A Unified Framework for Gradient-based Hyperparameter Optimization and Meta-learning

Luca Franceschi

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Engineering

University College London

June 20, 2021

I, Luca Franceschi, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Machine learning algorithms and systems are progressively becoming part of our societies, leading to a growing need of building a vast multitude of accurate, reliable and interpretable models which should possibly exploit similarities among tasks. Automating segments of machine learning itself seems to be a natural step to undertake to deliver increasingly capable systems able to perform well in both the big-data and the few-shot learning regimes. Hyperparameter optimization (HPO) and meta-learning (MTL) constitute two building blocks of this growing effort. We explore these two topics under a unifying perspective, presenting a mathematical framework linked to bilevel programming that captures existing similarities and translates into procedures of practical interest rooted in algorithmic differentiation. We discuss the derivation, applicability and computational complexity of these methods and establish several approximation properties for a class of objective functions of the underlying bilevel programs. In HPO, these algorithms generalize and extend previous work on gradient-based methods. In MTL, the resulting framework subsumes classic and emerging strategies and provides a starting basis from which to build and analyze novel techniques. A series of examples and numerical simulations offer insight and highlight some limitations of these approaches. Experiments on larger-scale problems show the potential gains of the proposed methods in real-world applications. Finally, we develop two extensions of the basic algorithms apt to optimize a class of discrete hyperparameters (graph edges) in an application to relational learning and to tune online learning rate schedules for training neural network models, an old but crucially important issue in machine learning.

Impact Statement

One primary contributions of this dissertation is to develop, motivate and analyze a unifying framework for hyperparameter optimization and meta-learning. We use a concise but expressive formalism, grounded in mathematical bilevel programming, that abstracts from case-specific implementation details and allows us to highlight similarities and differences between various learning and meta-learning algorithms. We anticipate that the dissemination of this framework in the academic community will help scholars and researchers to approach more smoothly these two important fields of machine learning research. We further believe that the framework can contribute to accelerating the research activity in the area of meta-learning, providing a starting conceptual and analytical basis which researchers may leverage in order to quickly prototype and develop novel algorithms.

We study several practical procedures for solving bilevel problems that arise in machine learning, closing existing gaps in the literature and proposing novel schemes. By and large, the thesis presents a series of general-purpose computational tools which may be integrated into several machine learning pipelines, to optimize hyperparameters or to meta-learn tailored algorithms. The resulting procedures may increase the value of data both in the corporate and in the public sector, possibly unlocking previously unfeasible applications. In a broader context, these tools may also constitute a step forward to the important goal of democratizing machine learning. In fact, they can simplify and partially automate the selection of various hyperparameters of many popular learning algorithms, for instance by finding performing learning rate schedules for training deep neural networks.

Finally, the material that we present in the second last chapter of this thesis might constitute a stepping stone for the study and development of methods that learn discrete dependency structures from data. The framework and the practical procedure that we develop may help promote the adoption of relational learning techniques, enabling their usage in scenarios previously out of reach. Progresses in this direction could have a potential impact in many important sectors of public and private interest, such as digital health or transportation, where relevant interactions and phenomena are inherently discrete. In machine learning, future applications may include the development of meta-learning algorithms capable of manipulating symbolic structures such as computational graphs or logic expressions. In turn, these efforts might contribute to closing the gap between "classic" approaches to artificial intelligence and modern practices.

Acknowledgements

In these four and a half years of PhD at UCL and IIT (Istituto Italiano di Tecnologia), I had the luck to share and cross paths (both scientific and not) with many fantastic people who have supported, motivated and inspired me in so many ways. To all of you, a sincere thank you.

A huge thank goes to my supervisor Massimiliano Pontil who, very importantly, decided to hire me even if I did not pass the first public selection procedure. Without that decision, which came after several additional rounds of interviews, I would probably have been doing something quite different now. Massimilano's guide has been invaluable in these years, wisely modulated between free exploration and mathematical rigour, frank and trustful, filled with pieces of knowledge, advice and experience which he has always been very happy to share. The openness of Massi to new research areas, his intuition, and his judgment are truly remarkable, as much as his ability to find very fruitful and matching collaborations for each project. One of these, which started at the very beginning of my PhD, involves Paolo Frasconi. I sincerely enjoyed our long-lasting collaboration, admiring his passion and energy as well as his exceptional understanding of all that has to do with neural networks. His participation has far exceeded that of a collaborator on so many occasions. Paolo has been a mentor to me, and for this, I am very grateful.

I wish to thank all my colleagues both in London and Genova, especially Michele and Carlo (who provided invaluable feedback on the background chapters of the thesis), Luca, Leonardo, Dimitris, Giulia, Leonard, Mo, Riccardo, Saverio and Jordan with whom I had the pleasure to share many wonderful moments and exciting discussions, relaxing aperitifs with focaccia and pint at the pubs (viruses allowing). Many of you have become trusted friends by now: I surely count this as one of the biggest achievement of my PhD.

In 2019, I had the fortune to work for four months at NEC laboratories Europe in Heidelberg. Many thanks to Roberto Baldessari for giving me this opportunity, and to Xiao He and Mathias Niepert, with whom it has been – and still is – truly a pleasure to collaborate.

I wish to thank Jun Wang and Amos Storkey who had the patience to go through this quite long work and organized a very memorable defence, notwithstanding the strange period we are living through.

A warm thank you goes to my hometown friends, the Hormonauts. Their nerd-but-not-too-much-nerdy joyful attitude always lifts my spirit, as much as the certainty in the resilience and worth of our friendship. And an equally warm thank you goes to the friends of “casa Serena”. Our uncountable endless discussions, the lunches, the dinners and (hopefully only temporarily) the zoom calls truly light my days.

Last, but surely not least, I wish to deeply thank my family: my wife, Hanui, her parents and brother, my parents Gianni and Antonella, my sister Angela and Francois. I always found the task of explaining what I am working on in few comprehensible words as an engaging exercise, albeit a very challenging one (which entails also learning Korean!). But it is your love, your support and encouragement that are the greatest contribution to this work.

Contents

[1]The content of this section is attributed to my coauthors of [Grazzi et al., 2020]; included here for completion.

[2]The experiments reported in this section were performed by my coauthors of [Donini et al., 2020]. We include them here for completion.

List of Figures

List of Tables

Chapter 1

Introduction

The increasing complexity of machine learning algorithms has driven a large amount of research in the area of hyperparameter optimization and automatic machine learning. Likewise, increasing attention has been dedicated to the problem of meta-learning, or learning to learn (we will use these two terms interchangeably) as the research community shifts its focus from solving single problems in isolation to studying and developing more elaborate systems able to learn and adapt quickly to several different tasks.

The core idea of hyperparameter optimization (HPO) is relatively simple: given a measure of interest relative to the performance of a model (e.g. the misclassification error) HPO methods use a validation set to construct a *response function* [Bergstra and Bengio, 2012] of the hyperparameters and explore the hyperparameter space to seek for an optimum. This process constitutes an empirical approach for calibrating a learning algorithm in order to reduce the generalization error of the learned statistical models that it produces. Consider, for example, the problem of learning a phoneme classification model as a part of an automatic speech recognition system (ASR) [Yu and Deng, 2016]. Instances in the domain space – feature vectors extracted from few milliseconds of recorded speech – should be mapped to a probability distribution over phonemes through multiple layers of affine and nonlinear transformations. A priori, however, there is no prescription on the exact number and dimensionality of these transformations. Furthermore, secondary tasks, such as recognizing the speaker identity, might provide beneficial supervisory signal; but their strength should be calibrated and validated against the primary performance criterion.

Meta-learning (MTL) instead, in its general (inductive) formulation, deals with the problem of finding a good algorithm that performs well on a whole class of tasks [Baxter, 1998]. The central idea is that, by extracting and exploiting information originated by a multitude of different (but related) tasks, one can learn an inductive bias specifically tailored to the family of problems of interest. Continuing with the previous example, suppose now that we wish to incorporate into our ASR system the ability to adapt to different speakers' accents or to compensate for the presence of various noise sources such as the car engine while driving, or the background chats of a pub. It is quite unrealistic (as well as computationally wasteful) to think of training a single monolithic model by collecting enough data for all the different possible scenarios. We may, instead, imagine bootstrapping a core phoneme classifier by learning small situation-specific filters based on a few samples collected on the fly. To simulate the working conditions of the system, we can group the available data into different tasks, characterized by accent or noise types, thereby constructing a "set of datasets". Then, we can optimize the core model so that the (generalization) error incurred by the compositions of the task-specific filters (one per dataset) and the phoneme classifier is, on average, minimized. At test time, the resulting learning algorithm, upon receiving data from a novel task, will only need to tune a new small filter, since the largest part of the model has already been meta-learned.

A Unified View. In HPO the available data is most often associated with a single task and split into a training set (used to tune the parameters) and a validation set (used to tune the hyperparameters). In MTL, on the other hand, we assume we have access to a large quantity of (potentially small) datasets sampled from a common probability distribution (a *meta-distribution*). In MTL, the search space often incorporates choices associated with the hypothesis space and the features of the learning algorithm itself (e.g., how optimization of the training loss is performed) while in HPO it may include variables associated to regularizers, model capacity, data prepossessing or augmentation, and optimization routines. Although experimental protocols and specific design choices may substantially differ, we will show that it is possible, and indeed natural, to develop a framework that encompasses these two branches of machine learning. The central observation is that, while in standard supervised learning we seek the best hypothesis in a given space and with a given learning algorithm, in *both* HPO and MTL we seek a configuration so that the optimized learning algorithm will produce one or multiple models that generalize well to new data. Under this common perspective, both HPO and MTL essentially boil down to *nesting two search problems*: at the inner level we seek a good hypothesis (as in standard supervised learning) while at the outer level we seek a good configuration (including a good hypothesis space) where the inner search takes place.

Technical Approach and Challenges. The mathematical framework of bilevel programming [Colson et al., 2007], where an outer optimization problem is solved subject to the optimality of an inner optimization problem, offers a natural ground upon which to develop our unifying view. In fact, once we restrict the scope to learning algorithms that internally seek to solve an empirical risk minimization (ERM) problem, we will see that that the resulting framework describes most instances of hyperparameter optimization and encompasses many existing approaches to inductive meta-learning.

Bilevel programs that arise in machine learning are usually characterized by few and simple constraints but high dimensional inner and/or outer variables. In addition, the programs structure is also, usually, quite weak, as the objectives are non-linear and most often not even quadratic. This is quite the opposite scenario of typical problem instances considered in the operations research literature [Vicente and Calamai, 1994, Bard, 2013], where reference application areas include transportation, management and engineering design.

Unless inner and outer objectives coincide, the bilevel structure cannot be in general simplified. For all but the simplest cases (e.g. ridge regression), inner problems of interest do not have analytic solutions, meaning that it is not possible to rewrite the bilevel programs as a single level problem. Numerical methods are therefore required to find approximate minimizers. Due to the growing complexity of underlying learning models (e.g. deep neural networks), finding such approximate minimizer – and thus being able to evaluate the outer objective at a given point – may take hours, or even days.

Hence, we seek for practical techniques that require few function evaluations and, critically, scale in the dimensionality of the inner and outer variables. The technical direction that we take in this work stems from the simple observation that in most cases we "know" the learning system that we employ to solve the inner optimization problem and thus, in principle, we can "access" to it. This contrasts with the assumption of many classic approaches to HPO, from grid and random search [Bergstra and Bengio, 2012] to Bayesian optimization [Snoek et al., 2012, Shahriari et al., 2015], which consider the outer objective as a black-box function, essentially ignoring the inner problem[1]. In effect, many algorithms that follow the ERM principle are expressible as iterative procedures that find approximate (local) minimizers of a regularized empirical error in some predefined parameter space. The iterative procedure itself is most often the application of a gradient descent rule to a loss function computed on some training data (or a subset of it). Hence, by substituting the inner problem with the repeated application of an *optimization dynamics*, it becomes feasible – under appropriate smoothness assumptions – to compute approximations of the gradient of the outer objective. We refer to this object as the *hypergradient*, to semantically distinguish it from the gradient of a single-level, non-nested, objective function. We may then search for optimal configurations in the hyperparameter space with a gradient descent procedure, as Figure 1.1 exemplifies. Leveraging the well-known advantages of dimensionality independence of these optimization techniques [Polyak, 1987a], we may then formulate and effectively tackle learning problems where the dimensionality of the outer variables is very high, a particularly desirable scenario in meta-learning. Conversely, classic approaches to HPO quickly become impractical as the number of hyperparameters grows and are thus hardly applicable to MTL settings.

Computing the hypergradient constitutes a major technical challenge for the application of the approximate bilevel programming approach. There are multiple ways to tackle the problem, which have different trade-offs in terms of running time, space requirements, and required proprieties of the underlying bilevel problem. One

[1] This is, instead, not necessarily true for several classic approaches to MTL where the inner problem is often an integral part of the design of meta-learning algorithms.

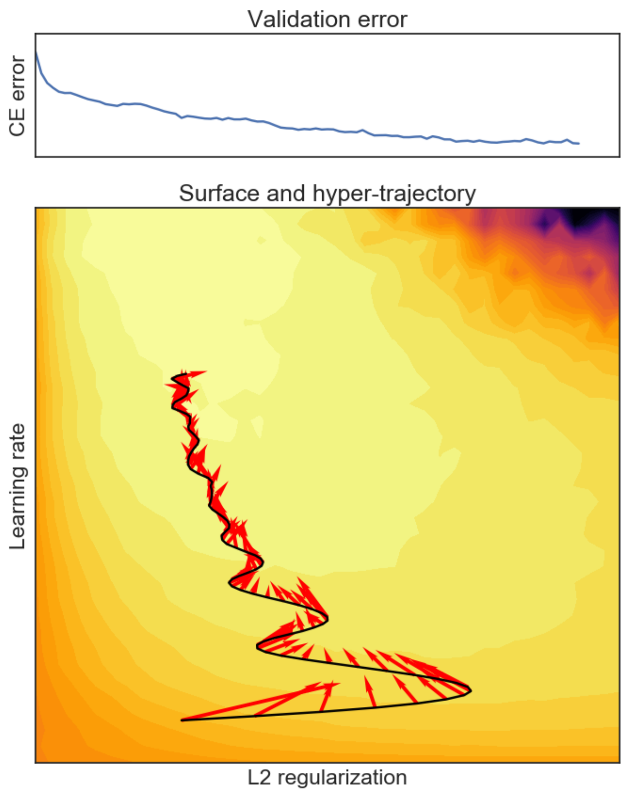

Figure 1.1: Bottom: response surface, obtained by extensive computation on a fine grid, and descent trajectory in the hyperparameter space for a small two-layers neural network trained on MNIST handwritten digits classification dataset. The inner and outer objectives are mean cross entropy over training and validation set, respectively. The inner objective (i.e. the training error) is optimized with stochastic gradient descent with momentum and includes an L^2 regularization term. The regularization coefficient coefficient is being optimized, alongside the learning rate, to reduce the validation error. The arrows in red indicate approximations of the negative hypergradients computed at each point, and the trajectory in black is generated by applying an accelerated optimization method (Adam). Lower to higher values are indicated with colors ranging from canary yellow to dark orange and black bean. **Top:** progression of the cross-validation validation error for the same simulation.

approach is based on a Lagrangian formulation associated with the parameter optimization dynamics. It encompasses the reverse-mode differentiation approach used by Maclaurin et al. [2015a], where the dynamics corresponds to stochastic gradient descent with momentum. A well-known drawback of reverse mode differentiation is its space complexity: we need to store the whole trajectory in the weight space in order to compute the hypergradient. An alternative approach that we consider overcomes this problem by computing the hypergradient in forward mode and it is efficient when the number of hyperparameters is much smaller than the number of parameters. Both methods require maintaining auxiliary system, called adjoint and tangent systems for reverse and forward mode, respectively. They involve the computation of the Jacobians of the optimization dynamics. These two approaches have a direct correspondence to two classic alternative ways of computing gradients for recurrent neural networks [Pearlmutter, 1995]: the Lagrangian (reverse) way corresponds to back-propagation through time [Werbos, 1990], while the forward way corresponds to real-time recurrent learning [Williams and Zipser, 1989]. We analyze also procedures that derive form the application of the implicit function theorem [Pedregosa, 2016, Koh and Liang, 2017] and a closely related approach that involves the differentiation of a fixed-point equation [Almeida, 1987, Liao et al., 2018]. In a comprehensive literature review on gradient-based HPO, we will further discuss other recently proposed techniques to compute or estimate hypergradients such as those employing randomized telescoping sums [Beatson and Adams, 2019] or hyper-networks [MacKay et al., 2019].

Our reformulations, which allow to derive practical algorithms to compute efficiently the hypergradient, give rise to families of problems that may be interpreted as "approximate bilevel programs". We prove that, for a class of inner objectives of practical interest, when replacing the inner problem with an iterative optimization dynamics the minimizers of the resulting approximate programs converge to those of the exact one. We provide non-asymptotic linear rates of convergence for the approximation errors of the hypergradient for two approaches, when the learning dynamics is a contraction. We will also investigate empirically the impact of various hypothesis on the quality of the solutions found by different approximation schemes.

We test the framework in a series of experiments on real-world inspired tasks that range from detecting noisy examples and discovering relationships between different learning tasks to quickly tuning few key hyperparameters of a large scale deep neural network for phoneme recognition. In MTL, by taking inspiration on early work on representation learning in the context of multi-task and meta-learning [Baxter, 1995, Caruana, 1998], we instantiate the framework in a simple, yet effective, way. We propose to treat the weights of the hidden layers of a neural networks as outer variables (hyperparameters), while we identify as inner variables the weights of multiple task-specific classifiers built upon the learned (hyper)representation.

Estimating Hypergradients Online. Even with the introduction of the optimization dynamics and the application of efficient algorithms, the nested structure of the computation may result in highly non-convex surfaces which may be difficult to optimize: often approximately solving bilevel problems requires hundreds of iterations of gradient descent to reach promising regions of in the hyperparameter space. This computational overhead could be an acceptable trade-off in some scenarios, since the bilevel framework may allow one to define more powerful and complex learning systems. However, in some others settings, speed is a key factor. We consider therefore the problem of devising an online hypergradient estimator, so that parameters and hyperparameters may be optimized *jointly*, in only one pass. To this end, we exploit the particular sparse structure of the hypergradient and devise an algorithm, which we call MARTHE (moving average real-time hyperparamter estimation), that makes use of moving average estimates to propose hyperparameter updates. We present an extensive study for the case of optimizing learning rate schedules, conducting small scale experiments to compare qualitatively optimal (static) schedules to those generated by MARTHE, and time-controlled real-world experiments to compare with alternative techniques.

Optimizing Discrete Hyperparameters. A fundamental assumption, necessary to perform the computation of the hypergradient, is that the objects involved (inner/outer objectives and optimization dynamics) should be sufficiently smooth. This means that the inner variables should be real-valued; excluding, in principle, important configuration parameters. We propose a second extension to allow for the gradient-based optimization of a class of discrete hyperparameters by introducing suitable discrete probability distributions. The result is the definition of a bilevel problem where the inner and outer objectives are minimized in expectation. As the expectations are intractable in all but the simplest cases, we resort to gradient estimators using stochastic computational graphs [Schulman et al., 2015] and the straight-through estimator [Bengio et al., 2013]. We call the resulting method LDS (learning discrete structures), and apply it to the problem of reconstructing or learning the edges of a graph in a relational learning context, where relationships among data-points are explicitly modelled by undirected unweighted adjacency matrices. LDS makes feasible to successfully apply graph neural network models [Scarselli et al., 2009], expressive discriminative models capable of exploiting relational information among data points, to semi-supervised transductive learning.

1.1 Contributions and Scope

The thesis contributes to the advancement of the field of gradient-based hyperparameter optimization and meta-learning. More specifically, the work is aimed at showing similarities and proposing a unifying approach to HPO and MTL, under both a conceptual and an algorithmic point of view. A core part of this research is also dedicated to the development of extensions of the fundamental algorithms, with the aim of broadening the range of applicability of gradient-based HPO techniques to previously unexplored scenarios.

Although, in principle, many techniques discussed may be extended to the unsupervised and reinforcement learning paradigms, throughout the thesis we will consider supervised or, occasionally, semi-supervised learning problems. In experiments, we will mostly use either linear (logistic/ridge regression) or deep neural network models to implement predictors and classifiers. Linear models, amenable to analytic investigation, provide often a simple but meaningful ground on which to probe the proposed methods. Experimenting with deep learning models, on the other hand, may potentially provide useful insight into the application of the developed algorithms to real-world problems. The thesis touches upon a number of topics in the general field of automatic machine learning, specifically in hyperparameter optimization and meta-learning (see Table 1.1 for an overview).

We make the following contributions:

- Formulation and analysis of a unifying framework for hyperparameter optimization and inductive meta-learning [Franceschi et al., 2018a], mathematically based on bilevel programming and algorithmic differentiation. Bilevel programming has been suggested before in machine learning in the context of kernel methods and support vector machines [Keerthi et al., 2007, Kunapuli et al., 2008], multitask learning [Flamary et al., 2014], and more recently HPO [Pedregosa, 2016], but, prior to [Franceschi et al., 2018a], never in the context of MTL. The framework will be presented in Chapter 5.

- Derivation of two general iterative procedures (reverse and forward mode) to compute hypergradients of approximate bilevel problems, which are directly applicable to both HPO and MTL settings. The reverse mode extends previous work by Domke [2012] and Maclaurin et al. [2015a], while the forward mode has never been studied before in these contexts, prior to [Franceschi et al., 2017]. The algorithms will be described in Chapter 5. Several numerical experiments are, instead, reported in Chapter 7.
- Study of the approximation proprieties of the approximate bilevel program that arise by replacing the inner problem with the iterations of an optimization dynamics [Franceschi et al., 2018a], establishing explicit error bounds for the computation of the hypergradient [Grazzi et al., 2020] for a class of inner objectives and dynamics. These results are reported in Chapter 6.
- Derivation of an online algorithm, MARTHE, for estimating hypergradients online and for jointly optimize parameters and hyperparameters of a learning system, with applications to optimizing learning rate schedules [Donini et al., 2020]. The algorithm uses a mechanism akin to momentum, exponentially discounting past information, and smoothly interpolates between two previously proposed methods, RTHO [Franceschi et al., 2017] and HD [Baydin et al., 2018a]. We show that MARTHE produces hyperparameter schedules which result in models with improved generalization on a variety of time-controlled real-world experiments. MARTHE and relative experiments are presented in Chapter 8.
- Adaptation of gradient-based HPO methods to work with a class of discrete hyperparameters (edges of a graph) and development of an algorithm, LDS, that

simultaneously learns the graph and the parameters of a graph neural network for semi-supervised classification [Franceschi et al., 2019]. This development is presented, alongside a series of numerical simulations, in Chapter 9.

Most of the material covered by the thesis has been presented in international conferences and workshops and is the result of collaboration with other scholars – the coauthors of the papers listed below. Chapter 3 provided the inspiration and initial material for a monograph on hyperparameter optimization, in preparation for Foundation & Trends in Machine Learning. The symbol * denotes equal contribution.

- CONFERENCE PAPERS:
  - Grazzi R., **Franceschi L.**, Pontil M., Salzo S. *On the Iteration Complexity of Hypergradient Computation*, Proceedings of the 37th International Conference on Machine Learning, ICML 2020, Online
  - Donini M*., **Franceschi L*.**, Majumder O., Pontil M., Frasconi P. *"MARTHE: Scheduling the Learning Rate Via Online Hypergradients"*, Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI 2020, Online
  - **Franceschi L.**, Niepert M., Pontil M., He X. *"Learning Discrete Structures for Graph Neural Networks"*, Proceedings of the 36th International Conference on Machine Learning, ICML 2019, Long Beach, USA
  - **Franceschi L.**, Frasconi P., Salzo S., Grazzi R., Pontil M. *"Bilevel Programming for Hyperparameter Optimization and Meta-Learning"*, Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholm, Sweden
  - **Franceschi L.**, Donini M., Frasconi P., Pontil M. *"Forward and Reverse Gradient-Based Hyperparameter Optimization"*, Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, Australia
- WORKSHOP PAPERS:

  - **Franceschi L.**, Niepert M., Pontil M., He X. *"Graph Strucutre Learning for GCNS"*, Representation Learning on Graphs and Manifolds at ICLR 2019, New Orleans, USA
  - **Franceschi L.**, Grazzi R., Pontil M., Salzo S., Frasconi P. *"Far-HPO: A Bilevel Programming Package for Hyperparameter Optimization and Meta-Learning"*, AutoML workshop at ICML 2018, Stockholm, Sweden.
  - **Franceschi L.**, Frasconi P., Donini M., Pontil M. *"A Bridge Between Hyperparameter Optimization and Learning-to-learn"*, Meta-Learning workshop at NIPS 2017, Long Beach, CA, USA.
  - **Franceschi L.**, Donini M., Frasconi P., Pontil M. *"On Hyperparameter Optimization in Learning Systems"*, 5th International Conference on Learning Representations (workshop track), Toulon, France.

- IN PREPARATION:
  - **Franceschi L.**, Donini M., Perrone V., Klein A., Seeger M., Archambeau C., Pontil M., Frasconi P. *Hyperparameter Search and Optimization in Machine Learning: Problems and Methods*, to appear in Foundations and Trends in Machine Learning

Collaboration with other researcher and research groups has lead to the following publications in application areas of robotics and speech recognition:

- Villarreal O., Barasuol V., Camurri M., Focchi M., **Franceschi L.**, Pontil M., Caldwell D.G., Semini C. *"Fast and Continuous Foothold Adaptation for Dynamic Locomotion through Convolutional Neural Networks"* IEEE Robotics and Automation Letters, vol. 4, no. 2, 2019
- Badino L., **Franceschi L.**, Donini M., Pontil M. *"A Speaker Adaptive DNN Training Approach for Speaker-independent Acoustic Inversion"*, Proc. Interspeech 2017, Stockholm, Sweden

Although related, these works are not explicitly covered in this thesis.

1.2 Outline

The work is divided into three parts. The first part is devoted to a detailed discussion of background concepts and topics instrumental in the development of the work. Chapter 2 serves as an introduction to the notation and to fundamental topics such as supervised learning and gradient-based optimization for machine learning. Chapters 3 and 4 are dedicated to a review of hyperparameter optimization and meta-learning, respectively. There, we will introduce the core problem settings, discuss similarities and differences with related fields and present relevant algorithms, strategies and techniques. Since in this work we propose a unifying framework between HPO and MTL, we believe that a thorough review of these two areas of research is essential to better frame and put into perspective our contributions. In particular, in Chapter 3, we will analyze the core components of the main approaches to HPO. This should help underpin the peculiar aspects of the gradient-based technique, highlighting its advantages and limitations. In Chapter 4, we will attempt to present a broad account of meta-learning that captures its central issues and questions. By discussing the internal working of several MTL algorithm developed over the last decades, we hope to foster the intuition behind the instantiation of our proposed framework in this area, exposing, at the same time, its limits.

The discussion of the contributions of our thesis, outlined in Section 1.1, starts in the second part. We introduce and discuss the proposed bilevel programming framework for HPO and MTL and derive the main algorithms in Chapter 5. We provide a theoretical analysis of various aspects of the procedures in Chapter 6, complementing our investigation with some targeted numerical simulations. In Chapter 7 we conduct a series of experiments with various instantiations of the proposed framework inspired by real-world applications in HPO and MTL, presenting comparisons with case-specific baselines and competing methods.

The final part covers extensions of the main algorithms with in-depth applications to two specific case studies. First, in Chapter 8, we focus on the problem of estimating hypergradients online, where we consider the important case of tuning learning rate schedules for deep neural networks. Second, we turn our attention to problems that

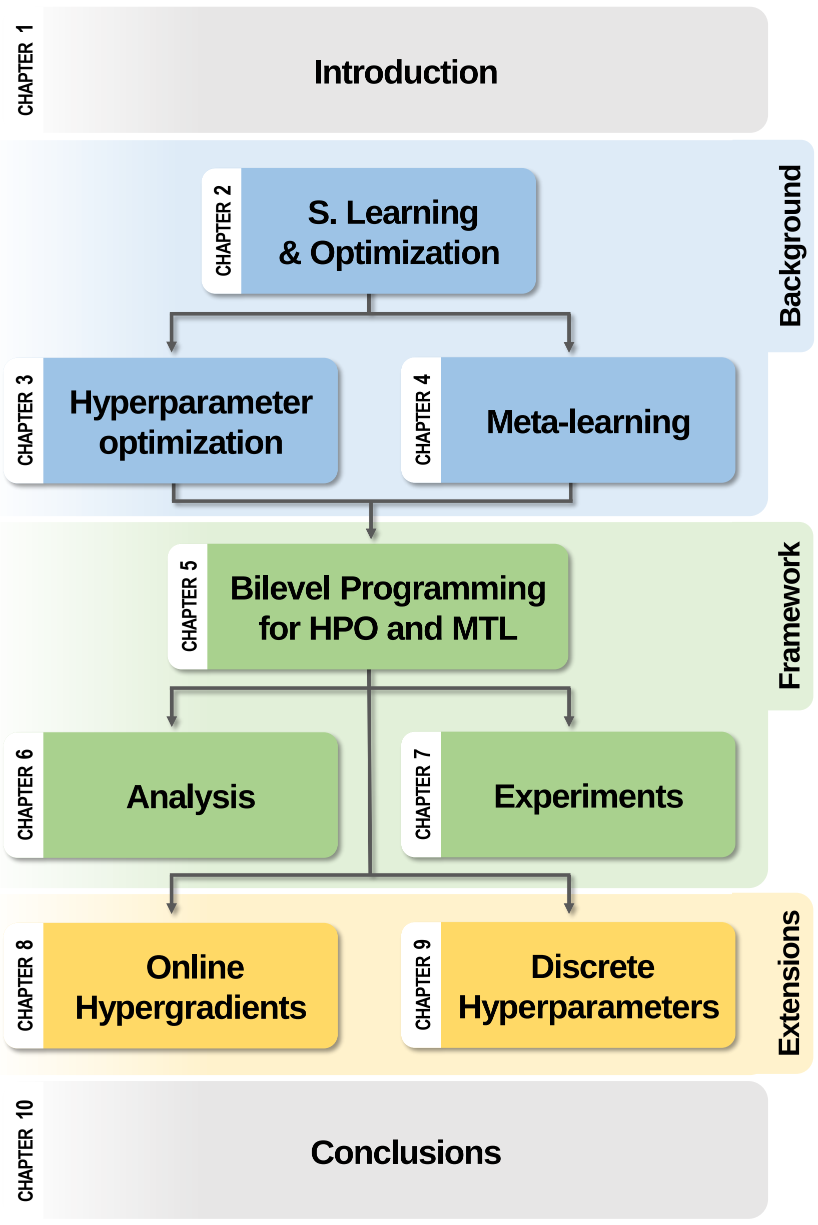

Figure 1.2: Structure of the thesis.

feature discrete outer variables (Chapter 9). We present applications to relational learning with graph neural networks where we learn discrete dependency structures between data points. Finally, we draw conclusions and discuss ongoing and possible future research directions in Chapter 10.

Figure 1.2 illustrate the structure of the thesis and Table 1.1 lists many of the topics discussed in the thesis, with references to chapter and sections

Table 1.1: List of topics discussed in the thesis, with references to chapters and sections, in loose apparition order.

Topics	*Occurrences*
Supervised learning	2.1; 7.1; 8.5
— Transductive learning	2.1; 9
— Learning algorithm (definition of)	2.1; 3.1
Regularization	2.1; 3.2.2 7.1
Linear models	2.2; 6.2.1; 7.1.1
Neural network models	2.3; 3.2.1
— Feed-forward neural nets	7.1.2; 8.4
— Convolutional neural nets	7.2.2; 8.5
— Recurrent neural nets	5.4.1; 5.4.2
— Graph neural nets	9
— Equilibrium models	5.4.3; 6.3.4
Gradient descent and variants	2.4
— Learning rates	3.2.3; 5.3.1; 8;
Algorithmic differentiation	5.4.1; 5.4.2; 6.1; A
Bilevel programming	5.2; 6; 9.3
Hyperparameter optimization	3; 4.3.5; 5.2.1; 7.1
— Online hyperparameter optimization	3.3; 5.4.2.1; 7.1.2; 8
— Neural architecture search	3.2.1; 9.6
Multitask learning	3.2.1; 3.2.2; 4.3.2; 7.1.1; 7.2.3
Meta-learning	4; 5.2.2; 7.2
— Meta-learning algorithm (definition of)	4.2
— Meta-distribution	4.2; 7.2.2
— Few-shot learning	4.1; 4.2; 7.2
— Meta-learning representations	4.4.1; 4.4.2.3; 7.2
— Meta-learning initialization (MAML)	4.4.2.3; 5.3.1
— Learning to optimize	3.2.3; 4.4.2.3

Part I

BACKGROUND

Chapter 2

Supervised Learning and Optimization

In the last decade, machine learning has established itself as one of the main drivers of innovation and economic growth in the industry [Society, 2017], while its research community is expanding at an unprecedented speed. Undoubtedly, a portion of its success is due to the paradigm shift brought by advances in representation and deep learning, which could easily capitalize on an increasing availability of data and compute. Rather than extracting hand-engineered rules and features, deep neural networks learn data-driven hierarchical representations which may be easily adapted to several downstream tasks, in an "end-to-end" fashion.

Using these techniques, several application problems that have vexed researchers since the seventeens have now sufficiently accurate solutions – handwritten characters recognition [Graves and Schmidhuber, 2009], natural language translation [Wu et al., 2016], game playing [Mnih et al., 2015] and language modelling [Brown et al., 2020] are some prominent examples. Finding these solutions, however, may require considerable computational and human effort, as frequently the best-performing systems are composed by complex models resulting from the application of heavily hyperparameterized algorithms. The generalization performances of these models are often sensitive to the configuration settings, to the point that the correct choice of a handful of these may make the difference between success and failure of the entire learning process[1]. The main goal of hyperparameter optimization is to automate the search process, thereby improving researchers productivity (and their well-being) and models' generalization performances, as well as allowing for a more flexible design of the underlying learning algorithms[2]. Furthermore, effective hyperparameter optimization may contribute to the democratization of machine learning. It may lead to the development of machine learning tools capable of "hiding under the hood" an increasing number of complex configuration parameters whose presence could otherwise hinder the access of non-expert users to more elaborate techniques.

Another bottleneck of many current approaches is represented by the vast quantity of data often required by the learning algorithms to achieve satisfactory results. In contrast, we are typically able to adapt quickly and generalize effectively to new situations and tasks, often based on very limited evidence. This is possible by accessing to acquired knowledge, by retrieving past relevant experience and by relating new tasks to previously encountered ones, among other cognitive processes that may potentially intervene. In short, we solve problems organically rather than in isolation. Recent advances in the fields of multitask, transfer and meta-learning are beginning to address this and related issues, whereby the idea of discovering entirely novel learning algorithms and relationships directly from data (rather than relying on strict hand-crafted routines and structures) is making its way in the research community.

Hyperparameter optimization and meta-learning lift the problem of learning from the space of features and functions to the space of entire learning algorithms.

[1] To cite a distinguished example, squashing nonlinearities such as the hyperbolic tangent and the logistic function have been for many years the uncontested choice for the activation function of neural networks. This prevented, for various reasons, the successful training of deeper models [Bengio et al., 1994, Glorot and Bengio, 2010]. The "simple" action of changing the value of this critical hyperparameter (e.g. by employing the rectifier linear unit [Glorot et al., 2011] or variants) has essentially removed a major roadblock for the development of an entire field – although, in this case, the change did not happen through the application of hyperparameter optimization tools.

[2]In fact, most of the hyperparameters contribute in defining what may be described as *"a space of hypothesis spaces"*. Informally speaking, the bigger this space, the more likely that it will include a hypothesis space that, in turns, contains a hypothesis which fits well the task at hand. Then again, the bigger the space, the more difficult the resulting optimization process may become, increasing the risks of "getting stuck" around poor hypothesis spaces. A good HPO technique, improving on the optimization in such space of spaces, should in principle allow for the "design" and effective utilization of richer spaces and, hence, more flexible learning algorithms. An additional potential problem of a too large space is the possibility of overfitting the objective function being optimized, that is typically a validation error. We will return to this point in Section 3.6.

In drawing a sort of parallelism with representation learning, they may have the potential to deliver a series of enabling techniques which could dramatically further the development of machine learning and its applications.

We will proceed with the formal treatment and review of hyperparameter optimization and meta-learning in Chapter 3 and Chapter 4, respectively. Before doing so, we introduce in this chapter some basic notation and concepts of supervised learning (Section 2.1) and present key models such as linear regressors, logistic classifiers (Section 2.2) and artificial neural networks (Section 2.3). The chapter concludes with a section dedicated to optimization techniques (Section 2.4), where we present standard procedures and theoretical results as well as algorithms and practices closer to applications to deep learning.

2.1 The Supervised Learning Problem

Broadly speaking,

> *Machine learning algorithms are computer programs characterized by the ability to improve their performances*[3] *at a class of tasks through experience.* [Mitchell, 1997]

The way in which the experience is collected and processed and the different types of performance measures and tasks define the three major paradigms of machine learning: supervised, unsupervised and reinforcement learning. In this work, we focus on the first of these three paradigms and touch upon some of its variants, such as semi-supervised learning (Chapter 9).

Supervised learning is marked by the presence of a supervisory signal in the learning environment, which typically associate a "correct" label or target $y \in \mathcal{Y}$ to each available datapoint $x \in \mathcal{X}$, where $\mathcal{X}$ and $\mathcal{Y}$ are an input and an output space. We often assume that $\mathcal{X}$ is a subset of $\mathbb{R}^n$ for some n. The output space may also be a subset of $\mathbb{R}^c$, in which case we talk about *regression problems* (e.g. predicting the selling price of a house, based on the location and size), or a categorical space, where $|\mathcal{Y}| = c$ and we are considering a *classification problem* (e.g. detecting the presence

[3]The concept of performance should be regarded as an externally defined measure.

of certain items in a picture). We assume that the phenomenon, or *concept*, from which the inputs are observed and linked to the outputs is described by an underlying joint probability distribution on $\mathcal{X} \times \mathcal{Y}$ which we denote[4] by $p_{x,y}$. Standard supervised learning algorithms infer mappings, or *hypothesis*, $h : \mathcal{X} \to \mathcal{Y}$ in a certain hypothesis space $\mathcal{H}$ that can capture the relation between inputs and outputs. The hypothesis may be either stochastic or deterministic. $\mathcal{H}$ could be, for instance, the space of linear functions between $\mathcal{X}$ and $\mathcal{Y}$. By introducing a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ (e.g. the least square error $\ell(y, y') = \|y - y'\|^2$) which measures the error between predictions and targets, we can define the expected risk, or *generalization error*, of h as

$$\mathcal{E}(h, p_{x,y}) = \mathbb{E}_{(x,y)\sim p_{x,y}} [\ell(h(x), y)] = \int_{\mathcal{X}\times\mathcal{Y}} \ell(h(x), y) \, \mathrm{d}\, p_{x,y}. \tag{2.1}$$

which expresses the overall performance of h on the task described by $p_{x,y}$ (according to ℓ). One would like to pick $h^* \in \arg\min_{h\in\mathcal{H}} \mathcal{E}(h, p_{x,y})$. This is, however, not possible since $p_{x,y}$ is unknown.

We can only observe a finite number of realizations of the phenomenon, i.e. the learning algorithm can only have access to a dataset of points $D = \{(x_i, y_i)\}_{i=1}^N$, which are typically regarded as i.i.d. samples from $p_{x,y}$. An ubiquitous approach in machine learning, called *empirical risk minimization* (ERM), involves replacing (2.1) with the computable quantity

$$\hat{\mathcal{E}}(h, D) = \frac{1}{N} \sum_{(x,y)\in D} \ell(h(x), y) \tag{2.2}$$

called *empirical risk* or error and search for hypothesis within $\mathcal{H}$ which minimize (2.2). One of the drawbacks of this approach is that the empirical risk minimizer may *overfit* the objective (2.2) at the expense of (2.1), leading to models that generalize poorly, possibly capturing noise present in D. To mitigate this phenomenon, often the empirical

[4] If x is a random variable, we denote by p_x its probability distribution. Writing $x \sim p_x$ simply means that x is distributed according to p_x (note the use of the same letter), while with $z \sim p_x$ we mean that we *draw* from p_x – z is a *realization*, i.e. a point in the codomain of x. If x is discrete, then p_x is a probability mass function, if x is continuous, then p_x is a probability density function. If y is a random variable or a realization, $p_x(\cdot|y)$ denotes the conditional distribution of x given y. We use this same notation also when y represents a parameters of the distribution. Throughout this work, to keep the presentation lean, when introducing a mapping we will indicate its deterministic domain and codomain. We may, however, evaluate it passing random variables (of the correct type) as arguments, leaving the definition of the relative higher-order mapping implicit.

error is augmented with a regularization term that favours certain solutions over others (e.g. "simpler" solution over complex one), based on background knowledge of the task, computational requirements or empirical observations. The study of regularization methods is of central importance in machine learning [see e.g. Vapnik, 2013, Friedman et al., 2001, Goodfellow et al., 2016]. We will return later on this topic when discussing the role and impact of regularization hyperparameters in Section 3.2.2. On the other hand, *underfitting* may happen when the empirical risk minimizer fails to sufficiently capture the complexity of the data. This, again, leads to models that generalize poorly, but, this time, for the opposite reason. Underfitting is most often associated with a wrong choice of the hypothesis space, e.g. when $\mathcal{H}$ contains too simple functions.

Rather than searching directly in spaces of functions, it is very common to parameterize $\mathcal{H}$ so that each hypothesis is described by its parameter (or weight) vector $w \in \mathcal{W}$, where $\mathcal{W}$ is most often a subset of $\mathbb{R}^d$, for some d. An inductive ERM problem (with Tychonov regularization) takes very often the form

$$\min_{w\in\mathcal{W}} L(w, D) = \min_{w\in\mathcal{W}} \frac{1}{N}\sum_{i=1}^{N} \ell(h_w(x_i), y_i) + \rho\Omega(w), \tag{2.3}$$

where Ω is a regularizer which may help prevent overfitting and ρ is a positive coefficient. The usage of appropriate regularization techniques is particularly important for the *overparametrized* case, when Problem (2.3) admits multiple solutions. When the context is clear, we will drop from L the dependency on the dataset D.

Inductive and Transductive Learning. What we have discussed so far pertains the so-called inductive learning setting: from a set of observations the goal is to *induce* a hypothesis that agrees with the data, but also generalizes (i.e. has a low expected error (2.1)). Among all the possible functions that may explain the data, the vast majority would not generalize at all: think about $h(x) = y_i$ if $x = x_i$ is a training point, and $h(x) = 0$ otherwise. The empirical risk (2.2) of h is 0, regardless of the task, but very likely its generalization error (2.1) would be quite high. One expects, in fact, that such a hypothesis would strongly overfit. When learning by induction, the *inductive* *bias*, implemented through the definition of the hypothesis space and regularization, discards hypotheses that are deemed “unlikely” in some sense (e.g. are discontinuous).

Another setting is that of *transductive learning* [Gammerman et al., 1998]. Closely linked to semi-supervised learning, transductive learning algorithms forgo the search for hypotheses that generalize over the entire domain $\mathcal{X}$, and only focuses on giving predictions for a finite set of points which are already part of the observations. Typically, given a dataset of training (or support) examples $D = \{(x_i, y_i)\}_{i=1}^{N_1}$ and testing (or query) points $D' = \{x_j\}_{j=1}^{N_2}$ a transductive learning problem consists in finding a hypothesis that minimizes the error over D'. Differently from the inductive setting, transductive hypotheses, being “one-use only”, take as input both D and D'. Since one is not interested in generalizing beyond D', usually the inductive bias of a transductive learning algorithm, although present, may be comparatively weaker. Furthermore, the algorithm may greatly benefit from additional information that relates support and query points, which could be represented by a graph. Consider, for instance, the problem of classifying the topic of a scientific article based on a feature vector that summarizes its content (e.g. bag of words). The task may become substantially easier if one has also access to the citations of other papers: one typically can expect that the probability of citing another article of the same topic is higher than that of citing a paper from a different topic. Thus, given a set of labelled and unlabelled articles with their *citation network* (which can be regarded as an unweighted directed graph), a transductive learning algorithm may leverage information from the entire set, not just from the labelled examples. The resulting hypothesis (e.g. implemented by a graph neural network) may propagate and aggregate the features of “neighbouring” articles to predict the topics of papers in the query set. It is, however, specialized on the dataset that has been used for training: classify new articles would, in principle, require rerun the algorithm, as the resulting hypothesis would differ (note that this is not the case for inductive learning algorithms). We will look more closely at the transductive learning scenario in Chapter 9, presenting also numerical experiments on tasks very similar to this example.

Learning as Executing Algorithms. Under a more general perspective, the process of learning may be regarded as the execution of an higher order function $\mathcal{A}$ (a learning algorithm) that maps a set of training data[5], representing available experience, to a hypothesis: the result of a search conducted in a hypothesis space $\mathcal{H}$. The search is often carried out by minimizing the empirical risk, whereby the minimization procedure should be considered as part of $\mathcal{A}$ itself. All the variables that are not part of this search may be considered hyperparameters of the learning algorithm. For instance, if $\mathcal{A}$ implements the minimization of (2.3) in a parameterized space of feed-forward neural networks (Section 2.3), then both the coefficient ρ and the regularizer mapping Ω may be regarded as hyperparameters.

We will expand on this view later on, starting from Section 3.1. We now proceed with the introduction of common loss functions ℓ employed in supervised learning and offer a brief overview of two prevalent classes of models which we will use extensively throughout this work.

2.2 Linear Models and Supervised Loss Functions

Linear (or affine) mappings are among the simplest and oldest model types routinely employed in supervised learning. They take the form of

$$h_w(x) = Wx + b \quad \text{with} \quad W \in \mathbb{R}^{c \times n},\, b \in \mathbb{R}^c \quad \text{and} \quad w = (W, b) \in \mathbb{R}^{c(n+1)}, \tag{2.4}$$

where n and c are the dimensions of input features and the target outputs, respectively. It is intended that w is obtained by concatenating and vectorizing as appropriate the single parameters. The hypothesis space for these type of models is therefore described by all the affine functions from $\mathbb{R}^n$ to $\mathbb{R}^c$. A common choice for the loss function for regression tasks is the squared error

$$\ell(h_w(x), y) = ||h_w(x) - y||^2. \tag{2.5}$$

For classification problems, a natural choice would be that of discretizing the

[5] In the case of transductive learning, this includes also the query set and, possibly, other (relational) information.

predictions of h_w by thresholding and check whether the result matches the target class. For instance, for a binary problem, encoding the positive class with $y = 1$ and the negative with $y = -1$ one could set

$$\ell(h_w(x), y) = [-y \operatorname{sign}(h_w(x))]_+ \in \{0, 1\}, \tag{2.6}$$

where $\operatorname{sigm}(z) = 1$ if $z \leq 0$ and -1 otherwise, and $[z]_+ = \max\{z, 0\}$ is the positive part; for multiclass problems, where $y \in \{1, \dots, c\}$, one could use

$$\ell(h_w(x), y) = \iota_{\{j \neq y\}}\left(\arg \max_{i=\{1,\dots,c\}} h_w(x)_i\right) \in \{0, 1\} \tag{2.7}$$

where ι is the indicator function. These are called 0-1 losses and have the clear disadvantage of having uninformative gradients almost everywhere. Seldom used as optimization objectives during training, (2.6) and (2.7) are often reported as final measures of accuracy. One common workaround to obtain a meaningful gradient is that of constructing a probability distribution from the models output using the softmax function

$$\operatorname{softmax}(h_w(x)) = \frac{e^{h_w(x)}}{\sum_{i=1}^{c} e^{h_w(x)_i}} \in (0, 1)^c \tag{2.8}$$

where the exponentiation at the numerator is element-wise, and employ the cross-entropy loss[6]

$$\ell(h_w(x), y) = -\log\left[\operatorname{softmax}(h_w(x))_y\right] = -\log\left(\frac{e^{h_w(x)_y}}{\sum_{i=1}^{c} e^{h_w(x)_i}}\right). \tag{2.9}$$

The quantity $h_w(x)_y$ is the component of the output vector relative to the y-th class and it is interpretable as the unnormalized probability that the model assigns to the event "x belongs to the class y" ($y \in \{1, \dots, c\}$). The denominator $\sum_{i=1}^{c} e^{h_w(x)_i}$ provides the normalization factor, so that from (2.8) one may easily forms a discrete probability distribution (a probability mass function) over the c classes. Hence, the cross-entropy loss (2.9), convex with respect to w, increasingly penalizes the model as

[6]Also called negative log-likelihood, the cross-entropy is equivalent to the logistic loss for the case of binary classification, up to a constant.

$[p(h_w(x))]_y$ departs from 1. The convexity implies that gradient descent procedures that minimize (2.9) converge to a minima (see Section 2.4.1). After training, frequently the output of classifications models is reconverted to be deterministic, setting $\tilde{h}_w(x) = \arg\max_i h_w(x)_i$.

In the case that instances may belong to more than one class (e.g. for detecting all the items in a picture rather than just one main object), one common approach it to employ the squared loss in conjunction with the logistic function:

$$\ell(h_w(x), y) = \left\| \left(1 + e^{-h_w(x)}\right)^{-1} - y \right\|^2$$

where the target $y \in [0,1]^c$ can hold more than one nonzero element and the exponentiation is element-wise. The i–th entry of $\left(1 + e^{-h_w(x)}\right)^{-1}$ may be interpreted as the probability that the model assign to the event "x contains the i-th object".

For scalar models ($c = 1$), typical regularization methods involve penalizing the L^p norm of W. Notably, the L^2 norm (standard Euclidean norm) promotes low variance while the L^1 norm promotes sparsity in the solution vector. When $c > 1$ (multivariate regression, multiclass classification), alongside entrywise equivalents of the previous norms (i.e. Frobenius norm for L^2 regularization), several regularization methods that relate different rows of W (interpreted as solution vectors of c different tasks) have been extensively studied in the context of multi-task learning. Among these, Laplacian or elliptic regularizers of the type $\Omega(w) = \sum_{i,j} c_{ij} \|w_i - w_j\|^2$, where w_i are the rows of W, promote similarities among tasks, while spectral regularizers such as the L^1 norm on the eigenvalues of W (nuclear norm) promotes low rank solutions [Mazumder et al., 2010]. See also Section 3.2.2 for more details and Section 7.1.1 for experiments in this setting, where we optimize the coefficients c_{ij} of Ω. Adding L^2 or Frobenius norm regularizers to convex objectives like (2.9) constitutes also a simple procedure to obtain strong convexity which, in turns, guarantees uniqueness of the minimizer.

Linear models attempt to explain outputs as weighted combinations of the raw inputs. They may therefore fail to capture complex relations and generally do not perform well when the raw features do not carry enough readily accessible information (e.g. x are image pixels). One way to overcome these limitation, linked to kernel methods [Murphy, 2012, ch. 14], is to transform the instances through some predefined features map $\chi : \mathcal{X} \to \mathcal{V}$, where $\mathcal{V}$ is a (possibly infinite dimensional) feature space and then fit a linear model from $\mathcal{V}$ to $\mathcal{Y}$. Another approach, which we describe next, is to directly learn parameterized feature maps from data, incorporating them into the model itself.

2.3 Deep Neural Networks

The collective name "deep neural networks" [Bengio, 2009, LeCun et al., 2015, Goodfellow et al., 2016] describes now a large class of models which share some typical design features and concepts such as the use of repeated compositions of simple parameterized mappings, called *layers*, the usage of distributed hierarchical representations and the centrality of modular approaches. Feed-forward (densely connected) neural networks are possibly the simplest non-trivial instantiation. In these type of models, affinities (2.4) are composed with nonlinear functions over multiple layers, pushing the input through several intermediate representations. Specifically, the model's output is given by $h_w(x) = z_L(x)$, obtained as

$$\begin{aligned} z_1(x) &= \sigma_1(W_1 x + b_1); \\ z_2(x) &= \sigma_2(W_2 z_1(x) + b_2); \\ \vdots \quad & \quad \vdots \\ z_L(x) &= W_L z_{L-1}(x) + b_L. \end{aligned} \tag{2.10}$$

The function σ_i are nonlinear element-wise operators, called *activations* (common instances are the logistic, ReLu [Glorot et al., 2011] or leaky ReLu [Maas et al., 2013] functions), while the variables $z_l(x)$ are are called *neurons* or (hidden) *units* – the jargon being inherited from the neuroscientific origins of these type of models [Rosenblatt, 1958]. The weight vector is hence given by $w = \{W_i, b_i\}_{i=1}^{L}$. The *depth* L and the dimensionalities of the hidden layers z_i specify the *architecture* of the network, defining the hypothesis space $\mathcal{H}$ for these type of models. The architecture is typically considered fixed or treated as a series of hyperparameters. Researchers have proposed over time several approaches that adapt depth, dimensionalities and also connectivity patterns during training (see e.g. [Fahlman and Lebiere, 1990] for early work and [Cortes et al., 2017] for a more recent attempt). Correlated to these studies, *neural architecture search* has emerged lately as a very active sub-branch of hyperparameter optimization and meta-learning. It aims to automatically discover novel architectures tailored to a given downstream task, starting from simple building blocks (see Section 3.1 for further discussion).

Densely connected layers and networks may be suitable for a series of applications, including, for instance, the phone recognition task mentioned in the introductory chapter. In other fields, however, more complex architectures have been proven to yield considerably better results. For instance, in visual tasks such as image classification, the affinities in (2.10) are replaced with convolutional operators[7]. Other now "standard" modification to the simple feed-forward architecture include adding pooling, attention [Vaswani et al., 2017] and normalization layers [Ioffe and Szegedy, 2015, Ba et al., 2016], as well as employing different connectivity patterns among layers. Researchers have developed a series of neural models for processing other types of inputs: recurrent neural networks maintain an internal state and are apt to process sequential data [Werbos, 1990, Hochreiter and Schmidhuber, 1997]. Graph and recursive neural networks employ message passing mechanisms to handle graph structured data [Scarselli et al., 2009, Frasconi et al., 1998, Kipf and Welling, 2017]. We will discuss these models and other variations more in depth in the following chapters, as the need arises.

Learning h_w is most often cast to an optimization problem using supervised loss functions such as those described in the previous section – in fact the last layer in (2.10) is essentially a linear model that acts on the features $z_{L-1}(x)$ rather than directly on x. This remains mostly true also for more complex neural models, with small variations depending on the type of output data. Differently from the linear case, however, the resulting objectives are no longer convex w.r.t. w, giving rise to a series of issues related to nonconvex optimization (multiple minima, saddle points, local proprieties of the optimization landscape, etc.) which have been (and still are) subject of much

[7]In fact, convolution and cross-correlation can still be still represented with the equations (2.10), where the weights W_i are (sparse and structured) block circulant matrices.

research in the field (see for instance [Baldi and Hornik, 1989, Dauphin et al., 2014, Li et al., 2018b]). Since neural networks are most often emploied in the presence of large datasets, and w may contain hundred of thousands or millions of parameters, the optimization routines of choice are typically stochastic first order methods, such as SGD or accelerated variants (see Sections 2.4.2, 2.4.3).

A Brief Historical Note. Neural networks were introduced in the fifties as simple models for simulating neural activity, supporting the argument that the brain acts (mainly) through connectionist, rather than symbolic, manipulation. The perceptron [Rosenblatt, 1958], given by $h_w = \text{sign}(Wx + b)$ is a famous example of these early models. Learning took place through the application of heuristic rules for adjusting the weights given a supervisory signal[8] such as Hebbian learning [Rosenblatt, 1958]. If, on the one hand, stacking multiple perceptrons did not yield appreciable results, mainly due to the underlying difficulties of finding solutions to the resulting combinatorial optimization problems, on the other hand, soon it became clear that the basic model had some serious limitations [Minsky and Papert, 1969]. This realization extinguished, at that time, much of the early interest of the community.

Toward the end of the eighties researchers started replacing "hard" nonlinearities with "soft" counterparts, e.g. by using the logistic activation function $\sigma(z) = (1 + e^{-z})^{-1}$. This modification of the earlier design pattern allowed us to cast effectively the search for good connections (weights) to a continuous optimization problem, quite similar to (2.3). In 1988, Rumelhart et al. [1986] demonstrated the application of the backpropagation algorithm[9] to learn a five layers neural network for recognizing kinship of a small group of people. The work is generally regarded as one of the first successful example of learning intermediate (or hidden) data-driven representations. Crucially, the gradient of the objective function could be computed accurately and efficiently, costing only a small multiplicative factor of the computation of the objective function itself.

Equipped with convenient differentiable parametrizations, neural networks could

[8] The gradient of any loss function that depends on this kind of h_w is not informative, being 0 almost everywhere; inf fact, a resulting optimization problem would be combinatorial in nature.

[9] An instantiation of reverse-mode algorithmic differentiation, see Chapter A.

effectively leverage the advances in smooth unconstrained optimization methods (see Section 2.4.3), and underwent a renewed, yet not lasting, period of interest. Overshadowed by emerging techniques more strongly rooted in statistical learning, such as support vector machines and kernel methods [Vapnik, 2013], research continued to later resurface roughly a decade ago. The field, re-branded as *deep learning*, has seen, since then, a number of advancements pertaining architecture design (convolutional layers [LeCun et al., 1998], rectifier linear units [Glorot et al., 2011], skip connections and residual layers [He et al., 2016], batch [Ioffe and Szegedy, 2015] and layer [Ba et al., 2016] normalization, to name a few), regularization methods (e.g. dropout [Srivastava et al., 2014]) and optimization techniques (e.g. weight initialization [Glorot and Bengio, 2010], accelerated adaptive methods [Kingma and Ba, 2015]). These advancements, alongside systematic development of software packages [Theano Development Team, 2016, Abadi et al., 2015, Paszke et al., 2017] and growing availability of computational resources (GPU acceleration) and training data, have elevated neural networks as the machine learning models of choice in many application scenarios, also in several other areas beside supervised learning [e.g. Goodfellow et al., 2014, Kingma and Welling, 2019, Mnih et al., 2013].

2.4 Optimization

Optimization is one of the pillars of machine learning, as problems of the type (2.3) arise routinely when fitting statistical models to observed data. While undoubtedly borrowing much from classic literature in operations research, convex and nonlinear programming [Polyak, 1987a, Nesterov, 2013, Nocedal and Wright, 2006, Bottou et al., 2018] optimization in machine learning exhibits a series of peculiar aspects and features that contribute to set it apart, to a certain extent, from its original birthplace. We review in this section some fundamental results and algorithms focusing on those we will use later in the thesis, commenting upon practices and issues that arise in optimization for machine learning. For the proofs of theorems and propositions we refer the reader to [Polyak, 1987a] where not otherwise stated.

Problem Setting. In this section, we consider the following *mathematical program*

$$\min_{w \in \mathcal{W}} f(w) \tag{2.11}$$

where w is the decision variable, the *domain* or *search space* $\mathcal{W} \subseteq \mathbb{R}^d$ is a convex closed set and $f : \mathcal{W} \to \mathbb{R}^+$ is a smooth non-negative objective function. In our context, the decision variable may be primarily thought of as the weights of the underlying model (thus we keep the same letter as in the previous sections), but could be also the hyperparameters of an hyperparameter optimization or meta-learning problem.

Problem (2.11) may not have solutions (e.g. consider the exponential function e^x on the real line); we characterize the existence of minimizers of f in the following proposition.

Theorem 2.4.1 (Weierstrass: existence of minimizers)**.** *If f is continuous and there exists a non-empty bounded sublevel set $\{w \in \mathcal{W} \, : \, f(w) \le a\}$ for some $a \in \mathbb{R}$, then f admits a (global) minimizer.*

The notation

$$\arg\min_{w \in \mathcal{W}} f(w) = \{u \in \mathcal{W} \, : \, f(u) \le f(w) \forall w \in \mathcal{W}\}$$

indicates the set of global minimizers of f (which may, in principle, contain more than a point). A point w is said to be a local minimizer if there exist an open neighbourhood U of w such that $f(w) \le f(u)$ for every $u \in U$. A minimizer is locally unique if there is an open neighbour in which there is no other minimizer. Points of the set $\{u \in \mathcal{W} : \nabla f(u) = 0\}$ where the gradient vanishes are called *stationary*. A stationary point may be a minimizer or maximizer (either local or global) or a saddle point. In general, a function may have multiple local and global minimizers and stationary points which are not necessarily minimizers; convex functions are an exception, as the next proposition describes.

Proposition 2.4.2 (Stationary points and minimizers of convex and strictly convex functions). *If f is a convex function, that is*

$$f(w+\tau(u-w)) \leq f(w)+\tau(f(u)-f(w)) \qquad \forall w,u \in \mathcal{W} \textit{ and } \forall \tau \in [0,1] \tag{2.12}$$

local minima are global minima. If f is also smooth, stationary points are (global) minimizers.

If f is strictly convex (i.e. (2.12) *is valid with the sign of strict inequality) then the minimizer is unique.*

When f admits a unique minimizer, we shall write $w^* = \arg\min_w f(w)$ to indicate such element.

2.4.1 Gradient Descent

The gradient of f represents the direction of local steepest increase in the value of f. The simplest algorithm to seek for minima is steepest gradient descent (GD) or batch gradient descent to better distinguish it from the stochastic variant (Section 2.4.2). When $\mathcal{W} = \mathbb{R}^d$ (unconstrained minimization), one can can start by picking a point $w_0 \in \mathbb{R}^d$, and then iteratively update the guess by following the direction given by the negative gradient

$$w_t = w_{t-1} - \eta_t \nabla f(w_{t-1}) \tag{2.13}$$

where $\eta_t > 0$ is a scalar step-size, also called learning rate in machine learning. Sometimes we will refer to the update step of an optimization method on the variable vector with the notation $\Delta_t w_t$, so that $w_t - w_{t-1} = \Delta_t w_t$. For GD $\Delta_t w_t = -\eta_t \nabla f(w_{t-1})$.

We provide two fundamental convergence results readapted from Polyak [1987a] considering a fixed step-size $\eta_t = \eta$. The first requires only f to be Lipschitz-smooth and merely shows that the iterates produced by gradient descent converge toward stationary points. The second pertains the narrower class of strongly convex functions, a subset of strictly convex functions, but provides much stronger results.

Proposition 2.4.3 (Convergence of gradient descent, general case). *Let $f \in C^1\left(\mathbb{R}^d\right)$ be a smooth function with Lipschitz continuous gradient:*

$$||\nabla f(w) - \nabla f(u)|| \leq \nu ||w-u|| \quad \forall \{w,u\} \subset \mathbb{R}^d. \tag{2.14}$$

If $\eta \in \left(0, \frac{2}{\nu}\right)$ then the iterates produced by (2.13) *are such that*

$$\lim_{t\to\infty} \|\nabla f(w_t)\| = 0 \qquad \textit{and} \qquad f(w_t) \le f(w_{t-1})$$

Proposition 2.4.4 (Convergence of gradient descent for strongly convex functions)**.** *Let $f \in C^1\left(\mathbb{R}^d\right)$ be an ν-smooth (corresponding to condition* (2.14)*) strongly convex function with modulus μ, i.e.*

$$f(w+\tau(u-w)) \le f(w) + \tau(f(u) - f(w)) + \mu\tau(1-\tau)\frac{\|u-w\|}{2}$$

for all $w, u \in \mathbb{R}^d$ and $\tau \in [0,1]$. Let $\eta \in \left(0, \frac{2}{\nu}\right)$, then the iterates produced by (2.13) *converge toward the unique minimizer w^* at the rate of*

$$\left\|w_t - w^*\right\| \le cq^k \qquad \textit{for} \quad q \in (0,1). \tag{2.15}$$

Moreover, if $f \in C^2\left(\mathbb{R}^d\right)$, the optimal fixed learning rate is given by

$$\eta^* = \frac{2}{\nu+\mu}$$

and, with this choice, the coefficient c and the contraction constant *q of* (2.15) *become*

$$c = \left\|w_0 - w^*\right\| \qquad q = \frac{\nu-\mu}{\nu+\mu}. \tag{2.16}$$

Even in the favorable setting of strongly convex functions, the speed of convergence can be remarkably slow when the difference between ν and μ increases, so that $q \to 1$.

Projected Gradient Descent. Gradient descent can be straightforwardly modified to handle the situation where $\mathcal{W}$ is a proper convex subset of $\mathbb{R}^d$. In this case the gradient does not necessarily vanishes at minimum points. A condition for w to be a minimizer of Problem (2.11) is that $\langle \nabla f(w), u - w\rangle \ge 0$ for all $u \in \mathcal{W}$. The method, called projected gradient descent, simply adds to (2.13) a projection step after the gradient update to ensure that the iterates remain inside $\mathcal{W}$:

$$\tilde{w}_t = w_{t-1} - \eta_t \nabla f(w_{t-1})$$
$$w_t = \mathrm{Proj}_{\mathcal{W}}(\tilde{w}_t)$$

where

$$\mathrm{Proj}_{\mathcal{W}}(w) = \arg\min_{u \in \mathcal{W}} \|u - w\|^2.$$

For strongly convex functions one can show a convergence result quite similar to 2.4.4 [Polyak, 1987a, Ch. 7, Th. 1]. The practicality of the method clearly depends on the difficulty of projecting onto $\mathcal{W}$. This is a very simple operations for sets such as balls, boxes or half-spaces, but can be as hard as solving the original problem in more complex cases.

Generalized Gradient Descent. Gradient descent may be adapted in a "visually simple" manner to optimize programs with non-smooth Lipschitz-continuous objectives, where the set of non-differentiable points of f has zero Lebesgue measure. The method, called generalized gradient descent, subgradient descent (especially when f is convex), or still, simply, gradient descent, consists in updating the decision variables by iterating

$$w_t = w_{t-1} - \eta_t u_t \quad \text{with} \quad u_t \in \partial f(w_{t-1}) \tag{2.17}$$

where η_t is a step-size and

$$\partial f(w) = \mathrm{Conv}\left\{ \lim_{i \to \infty} \nabla f(w_i) \,:\, w_i \to w \text{ and } \nabla f(w_i) \text{ exists} \right\} \tag{2.18}$$

is the generalized (Clarke) gradient [Clarke, 1990]. The set (2.18) is a singleton containing only the gradient wherever f is differentiable and coincides with the subgradient of convex functions (which we still denote by ∂f) defined by

$$\partial f(w) = \{q \in \mathbb{R}^d \,:\, f(u) - f(w) \geq \langle q, u - w \rangle \; \forall y \in \mathbb{R}^d\}.$$

The necessary (and sufficient, for convex functions) condition for w to be a minimizer of f becomes, for the non-smooth case, that $0 \in \partial f(w)$. When f is convex, it can be proved that the sequence of minimum values $\min_t f(w_t)$ with w_t obtained by (2.17) converge to the minimum of f [Ch. 5 Th. 1 Polyak, 1987a], provided that the learning rate sequences is such that

$$\eta_t \to 0 \qquad \text{and} \qquad \sum_t \eta_t = \infty.$$

We are not aware of convergence results to generalized stationary points, that is points for which $0 \in \partial f(w)$, for the general, non-convex, case. Non-smooth objectives arise often in machine learning due to non-differentiable loss functions or non-smooth activations (such as the ReLu activation) in neural models.

2.4.2 Stochastic Gradient Descent

Prototypical objective functions encountered in machine learning are often of the type

$$f(w) = \frac{1}{N}\sum_{i=1}^{N} f_i(w), \tag{2.19}$$

the right hand side representing an empirical version of an expectation $\mathbb{E}_\xi f_\xi(w)$. For instance, in a standard supervised learning setting, the random variable ξ may be distributed according to $p_{x,y}$, from which one samples a dataset containing N datapoints. The f_i's are then the sample-wise errors of h_w in (2.3); that is $f_i(w) = \ell(h_w(x_i), y_i)$ and $f(w) = L(w)$.

For most models, the cost of computing ∇f scales linearly with N (the size of the dataset) and – depending also on the complexity of the underline statistical model – this may translate into a prohibitive cost for each GD step (2.13) as N grows. Stochastic gradient descent (SGD) offers a remedy to this issue by computing at each iteration a random update direction g_t, incurring in a much lower cost then computing the full gradient. At each iteration, a subset of $K \geq 1$ indices are sampled uniformly and the parameters are updated according to

$$\begin{aligned} g_t(w_{t-1}) &= \frac{1}{K}\sum_{i\in\mathcal{B}_t} \nabla f_i(w_{t-1}) \quad \text{with} \quad \mathcal{B}_t \sim \mathcal{U}\{1, N\}^K \\ w_t &= w_{t-1} - \eta_t g_t(w_{t-1}) \end{aligned} \tag{2.20}$$

where $\mathcal{U}$ is the discrete uniform distribution. The parameter K offers a trade-off between computational cost and variance of the estimation. Especially in the neural network literature, $\mathcal{B}_t$ is often referred to as a mini-batch of examples and, accordingly, the algorithmic parameter K is called mini-batch size.

SGD and its accelerated variants are the optimization algorithms of choice in many machine learning applications. Beside very pragmatical computational constraints that may render (batch) GD impractical, there are several arguments in favour of SGD. We sketch some of these below. Others include regularization arguments [see e.g. Zhu et al., 2018], which are also central in machine learning applications, but are beyond the scope of the current discussion.

1. In many important applications (e.g. training neural models) the objective function is non-convex. Thus, gradient descent may be attracted toward a (poor) local minima or, worse, be trapped around saddle points where ∇f is close to 0. By following a stochastic direction at each iteration, it is less likely that g_t is close to 0 for many consecutive steps, especially if w_t is still far from w^*. In fact, SGD provably escapes from a class of saddle points [Ge et al., 2015] and, in general, is more robust to the presence of non-favourable stationary points.

2. When w_t is far from the optimal value, with high probability g_t will have a very similar direction to ∇f, yet costing much less. Thus SGD may progress faster[10] in the initial part of the optimization process. This behaviour does not carry over to the later stage, as the noise to signal ratio of g_t increases and typically batch methods achieve overall lower values of f if they are executed for long enough time.

3. Almost always in machine learning one is not necessarily interested in lowering the value of f as much as possible, but rather in finding points with a reasonable low objective value. This may be due to the fact that:

   (a) the ideal objective of optimization is the incomputable generalization error. However, when following the ERM principle, the objective (2.3) is the

[10] Where here the speed is measured in terms of runtime, not iterations.

function optimized in practice. Fully optimizing f may then be detrimental, as it may lead to overfitting. Unlike batch GD, SGD exhibits certain links with the generalization error (2.1) [Kuzborskij and Lampert, 2018]; in fact, for $N \to \infty$ (infinite examples) SGD optimizes for the expected risk, drawing samples directly form the data distribution, whereas GD would not be even applicable. Even in the finite case, one can show that for large enough sample sizes, SGD practically minimizes the expected risk up until t is small compared to N;

(b) even factoring out the previous point, f may still be far from the objective that one would naturally optimize. A prime example of this is given by classification problems, where arguably the most natural choice for a loss function is given by the discrete 0-1 loss (2.7), yet one in practice optimizes a differentiable surrogate, e.g. (2.9). Lower values of f may not necessarily correspond to lower values of the relative 0-1 loss, as this latter is clearly much less sensitive to small changes in the parameter space. In these cases, over-optimizing f may be wasteful, at best.

While points 1. and 2. are general, optimization-related, potential advantage of SGD for complex objective functions, the third point highlights some peculiarities of optimization in the contest of machine learning, mentioned at the beginning of the section. Unlike in some other fields, often, in machine learning applications, the performance of an optimization method in the initial stage weigh much more than that toward convergence. These peculiarities, alongside the ever increasing size of the available datasets, have determined the fortune of SGD in many learning scenarios.

The stochastic gradient g_t in (2.20) is an unbiased estimator of ∇f, hence, in expectation, it is a descent direction for f:

$$\langle \nabla f(w), \mathbb{E}_{\mathcal{B}_t}[g_t(w)] \rangle = \|\nabla f(w)\|^2 \geq 0.$$

If the second moment of g is bounded at stationary points and grows at most quadratically with the norm of the full gradient, i.e.

$$\mathbb{E}_{\mathcal{B}_t}[\|g_t(w)\|^2] \leq \zeta + \zeta_g \|\nabla f(w)\|^2, \tag{2.21}$$

one can show a series of convergence results regarding SGD [Bottou et al., 2018]. For ν-smooth μ-strongly convex functions, SGD with fixed step-size $\eta_t = \eta \leq (\nu\zeta_g)^{-1}$ converges linearly in value to a neighbourhood of $f(w^*)$:

$$\lim_{t\to\infty} \mathbb{E}[f(w_t) - f(w^*)] = \frac{\eta\nu\zeta}{2\mu}.$$

The speed of convergence increases for larger η, yet so does the expected optimality gap. To recover a convergence result similar to 2.4.4, although with sublinear rate, one needs to set a decreasing learning rate sequence. A classic result from Robbins and Monro [1951] requires that

$$\sum_{t=1}^{\infty} \eta_t = \infty \qquad \text{and} \qquad \sum_{t=1}^{\infty} \eta_t^2 < \infty.$$

For instance, one may set $\eta_t = \eta_0(t+\gamma)^{-1}$ for positive constants η_0 and γ.

To recover results similar to Proposition 2.4.3 for non-convex objectives, one needs additional assumptions on f [Bottou et al., 2018, Th. 4.12]. Finally we remark that the convergence results for SGD hold also in the case that the estimates g_t are biased; one only needs that in expectation g_t is a descent directions for $f(w_t)$ and that the expected norm of the estimator and that of the full gradient are comparable [Bottou et al., 2018].

2.4.3 Beyond Gradient Descent

The update directions of gradient descent and its stochastic variant rely only on local first order information carried by the gradient computed at the current iterate (w_{t-1}). However, access to higher order information such as the local curvature of f may potentially speed-up the progression toward a minimizer. The Newton's method, the prototype of second order optimization algorithms, locally approximates twice differentiable objectives up to the second order term of the Taylor expansion and sets the next iterate as the minimizer of this quadratic approximation. Concisely, a Newton's iteration takes the form of

$$w_t = w_{t-1} - \left[H_f(w_{t-1})\right]^{-1} \nabla f(w_{t-1}). \tag{2.22}$$

provided that the Hessian, denoted by H_f, is invertible at w_{t-1}. If the objective function is indeed quadratic and strictly convex, that is

$$f(w) = \langle w, Aw \rangle + \langle b, w \rangle + c \tag{2.23}$$

with $A > 0$, it is immediate to see that (2.22) always converges in one step to the solution of (2.23), irrespective of A and the initial point w_0. On the other hand, the convergence speed of GD is linked to the conditioning number[11] of A. This very favourable quadratic setting is, however, rather far from the practice: the conditions of (2.23) ($A > 0$) are not even verified locally for non-strongly convex objectives and saddle points are attractors for the Newton dynamics (2.22). Furthermore, the cost of inverting the Hessian may become prohibitive for larger problems. Many schemes have been proposed to approximate the inverse Hessian (quasi Newton's methods), whereby $\left[H_f(w_{t-1})\right]^{-1}$ is replaced by a positive defined preconditioning matrix B_t and the iteration assumes the form of

$$w_t = w_{t-1} - \eta_t B_t \nabla f(w_{t-1}).$$

where $\eta_t > 0$ is a step-size. The presence of a step-size limits the update to a neighbour – a *trust region* – of the current iterate. Among these, one of the most popular is the BFGS algorithm and its limited memory variant [Fletcher, 2013].

The machine learning community has devised a series of quasi-Newton methods for solving large scale non-convex learning problems. The so-called Hessian-free method [Martens and Sutskever, 2011] employs algorithmic differentiation tools and

[11] In fact, for (2.23), the constants ν and μ in Proposition 2.4.4 are simply the maximum and minimum eigenvalues of A. Then, the higher the conditioning number $\kappa(A) = \nu\mu^{-1} \geq 1$, the closer the contraction factor of (2.16) is to 1.

settles for rough approximate solutions of regularized quadratic problems obtained through few steps of conjugate gradient. The saddle free method [Dauphin et al., 2014] inverts the Hessian "in absolute value", thereby sidestepping the positive definiteness issue[12]. Notwithstanding reported successes in some application scenarios, higher order methods are sparingly used due to more complex implementations, unclear theoretical advantages (especially in the stochastic setting) and difficult to tune algorithmic hyperparameters. Optimization routines that employ only first order information, albeit in a more complex manner than GD and SGD, are, by far, more commonly applied: stochastic gradient descent with momentum and diagonal scaling methods are two prominent examples.

Gradient Descent with Momentum. The update step of gradient descent with momentum (GDM or SGDM for the stochastic variant) is given by an exponential running average of all the gradients up to the current iteration:

$$\Delta_t w_t = -\eta_t \sum_{j=1}^{t} \beta^{t-j} g_j(w_{j-1}) \tag{2.24}$$

where the step-size $\eta_t > 0$ and the momentum factor $\beta_t \in [0,1)$ are hyperparameters and g_j is either the full gradient ∇f or a stochastic estimation. By introducing an accessory variable $v_t \in \mathbb{R}^d$, sometimes referred to as *velocity*, one can conveniently write the GDM iteration as

$$\begin{aligned} v_t &= \beta_t v_{t-1} + g_t(w_{t-1}) \\ w_t &= w_{t-1} - \eta_t v_t \end{aligned} \tag{2.25}$$

with $v_0 = 0$. Maintaining a running average of the past gradients results in dampening the updates' magnitude for those coordinates that exhibit fast changing gradient directions. Conversely, updates are magnified for "stable" coordinates. Thus, GDM, by capturing some long term information, may display a more stable behaviour in presence of valleys and may progress faster than gradient descent on plateaus. A further intuitive advantage of SGDM (the stochastic variant) is that, by averaging consecutive gradient estimations, one can observe a reduction of the noise. This is also

[12]More precisely it involves rescaling the gradient along approximate Hessian's eigen-directions with the inverse absolute values of the corresponding Hessian's eigenvalues

suggested by the fact that the auxiliary variable v_t is an online (biased) estimator of the first moment of the stochastic gradient, that is its mean $\mathbb{E}[g_t]$: in the setting described in Section 2.4.2, one has $\mathbb{E}[g_t(w)] = \nabla f(w)$, and one could expect the random variable v_t to match "more closely" the gradient, leading to reduced variance. This argument, however, is not easily formalized, also due to the non-stationarity of the estimate.

In the batch case, when $\eta_t = \eta$ and $\beta_t = \beta$ are fixed, GDM is also known as the heavy ball method and provably yields faster convergence rates for quadratic strongly convex functions, achieving a contraction constant of

$$q_{\text{GDM}} = \frac{\sqrt{\nu} - \sqrt{\mu}}{\sqrt{\nu} + \sqrt{\mu}}$$

upon optimal choice of η and β, as shown by Polyak [1964] (cf. (2.16)).

RMSProp, method "informally" presented by Hinton in his lecture notes [Hinton et al., 2012], is a simple scaling scheme that employs a diagonal preconditioning matrix with $B_t = \text{diag}((\sqrt{v_t} + \varepsilon)^{-1})$ where v_t is a running average of squared gradients (RMS stands for running mean squares) and $\varepsilon \geq 0$ is a regularization coefficient, usually chosen very small. The RMSProp iterate reads

$$\begin{aligned} v_t &= \beta_t v_{t-1} + (1 - \beta_t) g_t(w_{t-1})^2 \\ w_t &= w_{t-1} + \frac{\eta_t}{\sqrt{v_t} + \varepsilon} g_t(w_{t-1}) \end{aligned} \tag{2.26}$$

where $\beta_t \in [0, 1)$ is a gain coefficient (Hinton proposes the default value of $\beta_t = 0.9$), $\eta_t > 0$ is a global learning rate, and g_t is a stochastic gradient[13]. One way to interpret the RMSProp update (2.26) – and, in fact, all diagonal scaling updates – is that every entry of the variable vector maintains its own learning rate. In (2.26), the component-wise learning rate is adapted by looking at the squared gradients, and it is increased when consecutive partial derivatives are consistently small in norm and decreased otherwise. Besides potentially helping in traversing large plateaus, such rescaling may be beneficial in contexts where the effective range of variables and gradients is spread

[13]The method has been specifically developed for the stochastic setting

over several orders of magnitudes throughout the optimization trajectory[14], a situation frequently observed in many neural network problems. Here, the accessory variable v_t is an online (biased) estimator of the second moment (2.21) of the stochastic gradient. Its direct usage in adapting the step-size in (2.26) potentially allows us to increase the effective learning rate over conservative choices of η_t and thus accelerate convergence, although in some cases it may also lead to divergence when v_t poorly underestimates (2.21).

Adam (adaptive moment estimation) [Kingma and Ba, 2015] is another method for stochastic optimization of large scale non-convex objectives. It essentially merges SGDM with RMSProp by maintaining a running average of both the first and the second moment of g_t, and further rescales the learning rate with bias correction terms. The method requires maintaining a set of two auxiliary variables, $\{v^1, v^2\} \in \mathbb{R}^{2d}$, both initialized at 0, and generate iterates according to

$$\begin{aligned} v_t^1 &= \beta_1 v_{t-1}^1 + (1-\beta_1) g_t(w_{t-1}) \\ v_t^2 &= \beta_2 v_{t-1}^2 + (1-\beta_2) g_t(w_{t-1})^2 \\ w_t &= w_{t-1} - \eta_t \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \frac{v_t^1}{\sqrt{v_t^2+\varepsilon}} \end{aligned} \tag{2.27}$$

where $\{\beta_1, \beta_2\} \in [0,1)^2$ are (constant) gains for the first and second running averages and $\eta_t > 0$ is the global learning rate. The authors suggest the default values of 0.9 and 0.999 for the gains and 10^{-3} for the learning rate. Despite the fact that it has been proved that for each setting of the algorithmic parameters (including considering decreasing step-sizes) there is an online convex optimization problem on which (2.27) does not converge to the global minimum [Reddi et al., 2018], the algorithms has gained considerable popularity[15] especially in the deep learning community, due to its empirical strong performances in the initial stage of the optimization and the relative simplicity of finding decent configuration parameters on many benchmark problems.

[14]For example, $[g_t]_i \in [10^{-6}, 10^{-3}]$ while $[g_t]_j \in [10^2, 10^4]$ for two indices $i \neq j$, for $t \geq 1$

[15]Based on the number of citations of [Kingma and Ba, 2015].

2.4.4 Early Stopping

As discussed in Section 2.4.2, for several reasons finding the (global) minimum of an objective f – identified as the training error – is not of primary importance in many optimization problems that arise in machine learning. An explicit technique often used in practice to avoid over-optimization is *early stopping* [Yao et al., 2007, Bengio, 2012]. At an high level, early stopping consists in terminating the optimization routine before full convergence, thereby accepting approximate solutions of (2.11) with an optimality gap that is higher than that achievable by running the algorithm for more iterations.

Usually, early stopping is implemented by regularly computing a score of interest, $a(w_t)$, which is different from the objective function f. This may be, for example, the accuracy on a validation set for a classification problem. The optimization is terminated if $a(w_t) < a(w_{t-1})$ (assuming higher is better). Additionally, since one cannot always expect a monotonic increase of a, very often a *patience* mechanism is employed, whereby early stopping is applied only if the score does not improve for several consecutive iterations. If $a(w_{t-\bar{t}})$ is the best value obtained so far, the optimization is terminated if $a(w_{t-k}) < a(w_{t-\bar{t}})$ for all $k \in \{0, \ldots, \bar{t} - 1\}$, and the algorithm return the iterate $w_{t-\bar{t}}$, irrespective of the value of the objective function f. The hyperparameter $\bar{t}$ is called *patience window*.

Early stopping acts as an implicit form of regularization, whose strength is regulated by $\bar{t}$, since it limits the effective search space. It has been interpreted also in terms of Bayesian variational inference [Maclaurin et al., 2015b] where the posterior distributions are implicitly generated by the (stochastic) optimization dynamics.

2.4.5 The General Scheme of Iterative Optimization

Abstracting away from particular implementations of the update rules of the various methods discussed so far, we present in Algorithm 1 the general scheme for the iterative optimization of an objective function f. It includes projection onto the domain if $\mathcal{W} \neq \mathbb{R}^d$ and early stopping. We will refer to Algorithm 1 and the notation introduced in this section in the second part of the thesis.

We denote by $\alpha \in A \subset \mathbb{R}^a$ the configuration parameters and with s the *state* of

Algorithm 1 Iterative optimization of f

```
1: Input: $\mathcal{W}$: domain; $w_0 \in X$: starting point; $\alpha$: configuration parameters; $T$: maximum number of iterations; EarlyStopping: optional early stopping procedure (2.4.4)
2: Output: Optimized parameters
3: $v_0 \leftarrow 0$                                          {Initialize accessory variables}
4: $s_0 = (w_0, v_0)$                                          {Construct state vector}
5: for $t = 1$ to $T$ do
6:     $(\tilde{w}_t, \tilde{v}_t) \leftarrow \Phi_t(s_{t-1}, \alpha)$          {Update state}
7:     $s_t \leftarrow (\mathrm{Proj}_{\mathcal{W}}(\tilde{w}_t), \tilde{v}_t)$       {Projection onto the domain}
8:     if EarlyStopping($w_t$) then break                    {Check for termination}
9: end for
```

the optimizer. This latter is the concatenation of the decision variable (the weights) and any other accessory variable v, such as the velocity for GDM described in Section 2.4.3. We denote by $\mathcal{W}'$ the resulting state's domain, which is typically given by $\mathcal{W} \times \mathbb{R}^{\dim(v)}$. For instance, for GDM $\dim(v) = d$ while for Adam $\dim(v) = 2d$. For (stochastic) gradient descent, which does not have any accessory variable, $\dim(v) = 0$ and $\mathcal{W}' = \mathcal{W}$. With this notations, setting as $s = (w, v)$ and $s' = (w', v')$, we define an *optimization dynamics* as a mapping

$$\Phi_t : \mathcal{W}' \times A \to \mathbb{R}^{d'}, \qquad \Phi_t(s, \alpha) = s', \tag{2.28}$$

that iteratively seeks for a minimizer of a give objective function. The subscript t refers to a potential stochastic evaluation. The objective function, gradient computations and variables' updates are wrapped into Φ_t to offer a concise, but convenient, representation of an iterative optimization process.

2.5 Interim Summary

We provided in this chapter a review of some essential concepts of supervised learning and optimization that we will use throughout the thesis. In particular, we presented linear and neural models that will constitute the hypothesis spaces of several numerical experiments that we will report in this work.

In the two following chapters, we will introduce the central notion of learning (and meta-learning) algorithms as abstract, higher-order mappings. Gradient-based iterative optimization is the backbone of several modern learning algorithms, from fitting logistic regressors to training deep neural networks. Thus, we dedicated Section 2.4 to the topic of optimization under a "machine learning perspective", discussing some differentiating aspects and features, especially in the context of stochastic methods. We stated some fundamental theoretical results and reviewed both classical and more recent techniques for gradient-based iterative optimization. We finally offered a schematic, general-purpose, algorithmic view (Section 2.4.5) of the resulting iterative search procedures that will help us formalize in Chapter 5 the main component of many learning algorithms, abstracting away from specific implementations. As one of the focus of the thesis is that of extending the applicability of gradient-based strategies to a higher level of abstraction (both in hyperparameter optimization and meta-learning), the theoretical results listed in Section 2.4 will also provide useful pointers for the analysis of Chapter 6.

Chapter 3

Review of Hyperparameter Optimization

We now turn our attention to hyperparameter optimization. We start the chapter by formally defining the problem in Section 3.1, then discuss the role and impact of various hyperparameters of representative supervised learning algorithms in Section 3.2 and in Section 3.3 we outline the online variant of the HPO problem. Section 3.4 is devoted to the review of the main approaches to HPO in machine learning, including model-free, model-based and population-based methods, delving in some details of the state-of-the-art approaches such as Bayesian optimization methods (with Gaussian processes) and population-based methods. As the thesis focuses on gradient-based HPO, we postpone an in-depth discussion of the technique to later chapters and use Section 3.5 to present relevant work and the state of the sub-field.

3.1 Problem Setting

Hyperparameter optimization [see e.g. Moore et al., 2011, Bergstra et al., 2011, Bergstra and Bengio, 2012, Maclaurin et al., 2015a, Bergstra et al., 2013, Hutter et al., 2015, Franceschi et al., 2017] is the problem of tuning the value of certain parameters that control the behavior of a learning algorithm. As already mentioned in Section 2.1, a (supervised) learning algorithm may be conveniently represented as a mapping that takes a dataset (representing a task) and a configuration (a list of hyperparameters/an hyperparameter vector) and returns an hypothesis:

$$\mathcal{A} : \mathcal{D} \times \Lambda \to \mathcal{H}; \qquad \mathcal{A}(D, \lambda) = h \tag{3.1}$$

where[1]

$$\mathcal{D} = \bigcup_{N \in \mathbb{N}} (\mathcal{X} \times \mathcal{Y})^N \tag{3.2}$$

is the space of finite dimensional dataset – $\mathcal{X}$ and $\mathcal{Y}$ are the input and output spaces – Λ is an hyperparameter space and $\mathcal{H}$ is an hypothesis space (e.g. linear functions between $\mathcal{X}$ and $\mathcal{Y}$). Since learning algorithm are generally applicable to multiple domains, one may also explicitly consider[2], setting

$$\mathcal{D} = \bigcup_{i \in I} \bigcup_{N \in \mathbb{N}} (\mathcal{X}_i \times \mathcal{Y}_i)^N. \tag{3.3}$$

which takes into account a class of possible input and output spaces. The hyperparamter space may not be equipped with any particular *global structure* (e.g. it need not be a vector space). Sometimes it is useful to think of $\lambda \in \Lambda = \Lambda_1 \times \cdots \times \Lambda_m$ as an m dimensional list where each component attends to a different aspect of $\mathcal{A}$. In a loose comparison with classical rule-based programming, hyperparameters may resemble the (inference) rules that the user passes to the program for it to make decisions.

As noted in Section 2.1, often the hypothesis spaces are parameterized by weight vectors $w \in \mathcal{W}$. When this is the case, we may equivalently think about $\mathcal{A}$ as a mapping

$$\mathcal{A} : \mathcal{D} \times \Lambda \to \mathcal{W}; \qquad \mathcal{A}(D, \lambda) = w, \tag{3.4}$$

where $\mathcal{H}$ in (3.1) is replaced by an appropriate weight space $\mathcal{W}$. Then, we implicitly

[1] Many learning algorithms are, in fact, stochastic, returning upon execution a randomized hypothesis. In this case, given a dataset and a configuration, $\mathcal{A}(D, \lambda)$ defines a probability distribution over $\mathcal{H}$. To ease the exposition, we consider here a (slightly idealized) deterministic case. We note that most arguments can be adapted to the stochastic setting by considering, where needed, expectations over the random components of $\mathcal{A}$.

[2] As we shall see, this very natural extension will prove especially useful when introducing the meta-learning problem in Chapter 4. Note that in (3.3), while the second union must be countable, as we deal with finite datasets, the same should not necessarily hold for the first one. We may in principle consider an uncountable collection of input/output spaces associated to different tasks.

assume the presence of a (total and surjective) mapping $w \to h_w$ that associates weights to hypothesis.

In principle $\mathcal{A}$ may be any arbitrary procedure that falls within the broad definition of learning algorithm given at the beginning of Section 2.1. It may be composed of various subroutines, branches and heuristics. However, the master example of $\mathcal{A}$ that we will refer and study in the following chapters takes the form of an iterative minimization of an empirical risk over a parameterized space of models (cf. (2.3)). As such, it is quite more structured in nature.

For instance, recalling the ASR example of the introduction, a learning algorithm that an user could think of executing for training a phoneme classifier may be informally described as

Example 3.1.1. Fit a multi-layer neural network using SGD with early stopping to minimize the average cross-entropy error on the available data.

The description above loosely designate an hypothesis space – feed-forward neural networks that map vectors of input speech features to probabilities over phoneme states – and sketches a training process – iterative minimization of a given objective function (the training error) with stochastic gradient descent. Yet, it does not describe many aspects that are necessary to run the algorithm in practice and may have a tremendous impact on the overall performances of the trained model. For example, details like how many layers and how many units per layer, what is it the learning rate (schedule), which value to use for early stopping patience window and whether to use any explicit forms of regularization are left undetermined. These additional information should be passed to $\mathcal{A}$ in the form of an hyperparamter (or configuration) vector, whose range of permitted values define the hyperparamter space Λ of $\mathcal{A}$.

It should be noted that there is no sharp border between "learning algorithm" and "hyperparameter space" and, consequently, "hypothesis space". For instance, one may consider algorithms that train CNNs as distinct from those involving RNNs, as they work preferentially on different type of input data. In this case, we would have two separate hyperparamter spaces whereby the first may contain choices relative to CNN architecture design (e.g. number of filters per layer, skip connections, ...), while the second may include hyperparameters that control the form of the recurrent cells. Conversely, one may prefer to think about any procedure that "works with and produces" neural networks as *a single learning algorithm* with a (considerably) larger hyperparameter and, hence, hypothesis space. The specific choice of which class of neural model to use (FFNN, CNN, RNN, GNN, etc.) could then be delegated to an hyperparameter, which, in turn, would lead to other conditional choices depending on the value it takes.

When formulating and developing a learning algorithm there is a trade-off between conceptual consistency and *generality* of use[3]. Bringing generality to one extreme, one may ideally collapse all machine learning methods into a single "general-purpose" algorithm that is able to tackle, in principle, every conceivable problem. Unfortunately, this idea, somewhat linked to the concept of *artificial general intelligence* [see Ford, 2018, for a recent reportage], while fascinating, would hardly constitute a solid ground for any constructive endeavour. In fact, we reckon that with our current level of understanding the cost to pay for such an extreme unification would be very likely an enormous inflation of the potential general-purpose algorithm's hyperparamter space, proverbially "sweeping the problems under the rug".

Likewise, the "level of detail" of a particular aspect of an algorithm's configuration space is not (or, rather, cannot) be uniquely determined and must be considered as a context-dependent concept. In the ASR example above, the iterative optimization rule is fixed to be an SGD update (2.20) and thus should be consider part of $\mathcal{A}$ itself. Only the learning rate schedule is an hyperparameter, as it is not determined by the learning algorithm during execution. Conversely, one may want to allow for a larger class of update rules (SGDM, RMSProp, Adam, etc.): in this case the categorical choice of the rule itself is to be considered part of Λ. Going further, one may even embrace the approach of "learning to optimize" (see Section 4.4.2), and richly parameterize the iterative update rule, e.g. with an LSTM recurrent network – the (real-valued) parameters of the resulting operations would, then, become themselves part of Λ [Andrychowicz et al., 2016, Bello et al., 2017, Wichrowska et al., 2017, Metz et al.,

[3]Informally defined as "the number of application scenarios" in which the algorithm may potentially work.

2019].

For the purpose of our discussion, we shall think of the hyperparameters of a learning algorithm as the variables that should be set before execution. These may take on different values without modifying "too much" the underlying learning process (see Section 3.2 for several examples). This essentially accounts to require a certain degree of coherency in the hypothesis space of the learning algorithm. The permitted level of detail should be stated upon defining Λ and all the remaining specifications shall be considered fixed and part of $\mathcal{A}$ itself. Thus, when discussing about hyperparamter optimization, we shall consider the mapping $\mathcal{A}$ as fixed and given. We consider the upstream problem of choosing which learning algorithm to use for each specific application scenario as a related, but different, field of study. This problem setting, oftentimes referred to as *algorithm selection*, has been addressed in e.g. [Thornton et al., 2013, Luo, 2016, Kotthoff et al., 2017, Fusi et al., 2018], although in relatively limited and homogeneous setups. It has also been studied by various authors in meta-learning contexts, see Section 4.4.2.

Regardless of the internal working of $\mathcal{A}$, one can expect that, for the same dataset D, different values of $\lambda \in \Lambda$ will result in different hypotheses. It is then natural to ask, among a set of candidate models $\{h_i = \mathcal{A}(D, \lambda_i)\}_i$, which one is to be preferred. Ideally, one would like to pick the hypothesis that minimizes the generalization error (2.1), but, of course, this is not possible, as $p_{x,y}$ is unknown. A common procedure is to use an empirical estimate of (2.1) by computing the average loss (2.2) on a set of data, D_{val}, that has not been used by $\mathcal{A}$:

$$\hat{\mathcal{E}}(\mathcal{A}(D_{\text{tr}}, \lambda), D_{\text{val}}) = \frac{1}{|D_{\text{val}}|} \sum_{(x,y)\in D_{\text{val}}} \ell(\mathcal{A}(D_{\text{tr}}, \lambda)(x), y). \tag{3.5}$$

Here, we have renamed the dataset used by the learning algorithm – the *training set* – as D_{tr} to better distinguish it form D_{val}. D_{val} is called *validation* or *held-out* set. From now on, we shall refer to the entirety of the available data as $D = D_{\text{tr}} \cup D_{\text{val}}$. The resulting quantity (3.5), called *validation error*, is an unbiased estimation of (2.1), provided that the points in D_{val} are i.i.d. samples from $p_{x,y}$ and that $D_{\text{tr}} \cap D_{\text{val}} = \emptyset$. This last fact is of primary importance: one cannot expect $\hat{\mathcal{E}}(\mathcal{A}(D_{\text{tr}}, \lambda), D_{\text{tr}})$ to be unbiased (it is, usually, too optimistic), given that $\mathcal{A}(D_{\text{tr}}, \lambda)$ picks a particular hypothesis by seen D_{tr} and (most likely) also $\hat{\mathcal{E}}(\cdot, D_{\text{tr}})$ itself. One may then pick the hypothesis that minimizes (3.5) among a number of available choices $\{h_i = \mathcal{A}(D, \lambda_i)\}_i$.

Going further, we may think of *optimizing the validation error* with respect to the variable λ, that is, finding the algorithm's configuration parameter that are the solution to the following optimization problem:

$$\lambda^* \in \arg\min_{\lambda \in \Lambda} \hat{\mathcal{E}}(\mathcal{A}(D_{\text{tr}}, \lambda), D_{\text{val}}). \tag{3.6}$$

This problem is the core of hyperparameter optimization and constitutes an empirical approach to hyperparameter tuning, akin to ERM for model fitting. Yet, unlike the parameterized ERM Problem (2.3) introduced in Section 2.1, the structure of the search space of (3.6) can be very complex and far from $\mathbb{R}^m$. Hyperparameters may be integer (number of layers in the ASR example) or categorical (which type of regularization to use). Further, there might be conditional relations between different components of λ. For instance, the number of units of the l-th layer is relevant only if the network is at least $l + 1$ layers deep). Thus, the validation error (3.5) is, in principle, neither smooth nor necessarily continuous, making up for a considerable challenge on the optimization side. Finally, as $\mathcal{A}$ typically involves itself the solution of a complex problem, even evaluating the objective of (3.6) at a point (namely one set of hyperparameters) may be computationally very intensive. Practical HPO methods should, therefore, rely on as few objective evaluations as possible and should benefit, whenever possible, from cheaper approximations of $\mathcal{A}(D_{\text{tr}}, \lambda)$.

There are other objectives considered in the HPO (and model selection) literature, beside (3.5). The cross-validation error [Stone, 1974] is a direct generalization of (3.5) obtained by repeatedly partitioning the available data into non-overlapping training and validation splits $\{(D^k_{\text{tr}}, D^k_{\text{val}})\}_{k=1}^K$ and by computing

$$\frac{1}{K} \sum_{k=1}^{K} \hat{\mathcal{E}}(\mathcal{A}(D^k_{\text{tr}}, \lambda), D^k_{\text{val}}). \tag{3.7}$$

Practical implementations include the so-called K-fold cross-validation, whereby the data is partitioned into $K \leq N$ subsets $\{D^k\}_{k=1}^K$ of roughly the same size, and the k-th split is given by $\left(\cup_{i\neq k} D^i, D^k\right)$. The special case where $K = N$ is known as leave-one cross-validation. Cross-validation objectives offer reduced variance over (3.5) as K grows, but are clearly more expensive to compute, since they require running multiple times the learning algorithm on the various data splits.

Possible variations of the validation objective include weighted error, precision, recall, F-measures and their smooth relaxations [Keerthi et al., 2007]. These may encode desiderata about the resulting statistical model (e.g. sensibility of a spam-email classifier) and may be implemented by appropriate choices of the point-wise loss ℓ within $\hat{\mathcal{E}}$. Researchers have also proposed several criteria that do not require splitting the available data into training and validation. These typically rely on various concepts of model's complexity. Among these, we mention the Stein's unbiased risk estimate [Stein, 1981], the Akaike information criterion [Akaike, 1998] and the Bayesian (or Schwartz) information criterion [Schwarz et al., 1978]. Other model specific criteria also exists: see e.g. [Chapelle et al., 2002] and references therein for SVM-related "goodness" measures. These criteria, replacing the empirical risk approach of $\hat{\mathcal{E}}$, make, however, assumptions on the underlying (statistical) properties of certain types of models and are less widely applicable than (3.5) and (3.7).

From now on, when the context is sufficiently clear, we may use the notation $f(\lambda)$ to refer the the objective of an hyperparamter optimization problem such as (3.6). The mapping f is often referred to as *response function*. Its image, $f(\Lambda)$, is called *response surface*. If not stated otherwise, we will assume that

$$f(\lambda) = \hat{\mathcal{E}}(\mathcal{A}(D_{\mathrm{tr}}, \lambda), D_{\mathrm{val}}). \tag{3.8}$$

In (3.8) the dependence on the input data (and the relative split) is wrapped into f: this is because we (primarily) think of an HPO problem as a *single-task problem*, whereby D is given and fixed[4]. Conversely, as we shall see in Chapter 4, in meta-learning we are interested in the behaviour of the learning algorithm on an entire distribution of

[4]This, nevertheless, does not exclude D from being a multi-task learning dataset.

tasks as well as in the generalization of $\mathcal{A}$ w.r.t. novel tasks. Thus, in the meta-learning context, the factor of variation represented by the input data cannot be likewise "hidden away" but shall rather be clearly highlighted. We will discuss in much greater details about the relations between HPO and MTL in Chapter 5.

3.2 Hyperparameters of Learning Algorithms

Each learning algorithm defines its particular set of hyperparameters and specifies, explicitly or implicitly, the respective role that these have in the resulting learning process. This may be as diverse as "leaf nodes must contain at least k data instances" when learning decision trees [Quinlan, 2014] to "initialize k convolutional filters at the 4-th layer of the network" when training CNNs. Whilst it is well beyond the scope of the current work to present a detailed account of all the possible occurrences, we offer below an overview of common hyperparameters typically found in learning algorithms and statistical models that we will employ or discuss in the following chapters. As we do so, we also take the opportunity to introduce and discuss relevant pieces of literature that, although linked to HPO, focus specifically on particular aspects of specific learning algorithms.

We semantically divide hyperparameters into three categories: *design hyperparameters* are closely tied to the specification of the hypothesis space itself, *regularization hyperparameters* control the (usually soft) constraints on the effective search space $\mathcal{H}$ and *optimization hyperparameters* intervene in the minimization of the empirical risk or, more generally, of a training objective. Clearly, these should not be considered as rigid distinctions as many hyperparameters could easily fit in multiple categories.

3.2.1 Design Hyperparameters

The definition of the hypothesis space $\mathcal{H}$ plays a crucial role in the development of a learning algorithm, as it determines which hypotheses (i.e. trained models) $\mathcal{A}$ can in principle output, regardless of the (admissible) training data it receives. Selecting a particular value $\bar{\lambda}$ of a design hyperparameter typically accounts for the process of restricting the search space to a subset $\mathcal{H}_{\bar{\lambda}} \subset \mathcal{H}$ where the search will then take place. At the highest level, the design of $\mathcal{H}$ depends on the type of data that the algorithm is supposed to process. For example, if $\mathcal{X}$ represents a space of (labelled) graphs describing, for instance, citation networks (Section 9.4) then $\mathcal{H}$ should include models that are able to effectively process this kind of structured information. While searching for linear hypothesis might still be a viable option, e.g. by including in $\mathcal{A}$ a preprocessing step that simply discards the relational structure, or extracts in some way a vector representation of the graph, it is likely that more specialized models such as graph neural networks will be more effective at capturing the underlying phenomenon of interest (possibly achieving a lower *irreducible error*). With few, partial. exceptions [Thornton et al., 2013], "standard" HPO treats this "macro-choice" as tied to the learning algorithm itself, rather than as a (categorical) hyperparamter. This second approach would, in fact, inject considerable conditionally in the rest of the hyperparameter space, further complicating the resulting HPO problem (3.6).

Table 3.1 lists a selection of design choices and their relative hyperparameters, divided into three parts. The topmost part groups choices shared among different classes of supervised learning models, the central part is relative to neural networks, while the last part present a few design choices relative to other types of models. We attempt to indicate the type of the relative hyperparamter (categorical, Boolean, integer, ...) in the third column of the table in what we identify as typical implementations, although we recognize that there may be different possible encoding options (especially for the entries marked with an asterisk).

The structure of simpler models may be fairly easily identified: affine models are essentially constrained by the input and output space dimensionalities. Possible choices in this case boil down to including or not a vector of bias terms (the term denoted by b in (2.4)) and possibly selecting a subset of input features. This latter choice, named precisely *feature selection* [James et al., 2013], is, in fact, a very general design process that has been studied from multiple point of views in several contexts [Guyon and Elisseeff, 2003]. The assumption of feature selection is that a subset of input features may be irrelevant or redundant to explain the dependent variables. The main goals are lowering computational and statistical complexity of the resulting model, potentially increasing interpretability. There are several ad hoc techniques for

Table 3.1: Examples of design hyperparameters. In the "Type" column, "Cat." stands for categorical, * indicates that there are other possible parameterizations and † denotes conditionality. Note that some conditional hyperparameters do not indicate any "standard" type, as this is heavily dependent on the parent choice. The table is divided into tree parts: from top to bottom the entries are relative to "general" design choices, neural networks, and kernel-based methods.

Name	*Description*	Type
Feature selection	*Selection of a subset of the n input features*	$\{0,1\}^n$
Loss function	*Selection of the (supervised) loss function and, when appropriate, the model's output (cf. 2.2)*	Cat.*
Sub-model sharing	*In multi-task learning, potion of the computational graph shared among multiple models*	Cat.
N. of layers/nodes	*Depth of a neural network*	$\mathbb{N}$
N. of units/channels	*Width of the layers of neural network*	$\mathbb{N}^\dagger$
Layer type	*Functional type of each layer: e.g. fully connected, convolutional, pooling or averaging, normalization [Ioffe and Szegedy, 2015, Ba et al., 2016], etc.*	Cat.†
Layer parameters	*Layer-specific configuration parameters (e.g. kernel width and stride for a convolutional layer)*	*,†
Activation	*Type of nonlinear activation*	Cat.*
Connectivity patterns	*Edges between layers of the directed computational graph that represents the network*	$\{0,1\}^\dagger$
Kernel type	*The functional expression for the kernel mapping*	Cat.
Kernel parameters	*Various hyperparameters that may define the kernel mapping (e.g. the degree of polynomial kernels, the bandwidth of RBF kernels, ...)*	*,†

performing feature selection, the earliest dating back at least to the sixties [Efroymson, 1960]. Sparsity inducing regularization techniques such as LASSO [Tibshirani, 1996] may be considered related alternatives. However, in all generality, selecting input features may be formulated as an HPO problem with n binary hyperparameters, one per each entry of the input vector.

The choice of the loss function, although mainly related to the downstream application, may be also considered as a categorical hyperparameter, alongside the type of the model's output; Section 2.2 indicates typical options. Very recently some authors have proposed to learn parameterized loss functions from data [Wu et al., 2018a, Bechtle et al., 2019]. Furthermore, sometimes it can be beneficial to weight examples separately, for instance to mitigate known data imbalances or noisiness

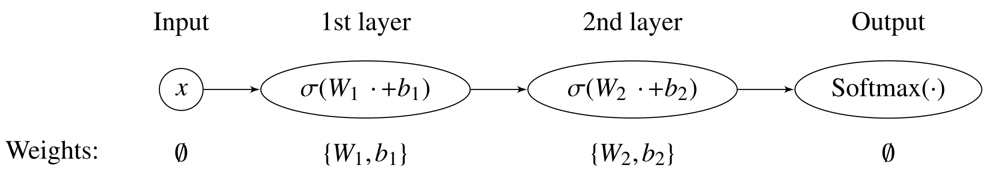

Figure 3.1: Representation of the computational graph of a two layers feed-forward neural network, where the "Softmax" function is defined in Section 2.2. Each node should be interpreted as a possibly parametric function whose input is denoted by ·. The parameters are "stored" in the node itself (shown in the row below the diagram). Other representation are possible: for instance, Theano [Theano Development Team, 2016] computational graph plotting package uses nodes to encode either single variables (inputs or weights) or operations.

[Franceschi et al., 2017, Ren et al., 2018]. In this case, one may associate a real-valued hyperparameter to each example, as we do experiments reported in [Franceschi et al., 2017].

Neural networks. The hypothesis space of neural networks is sensibly more complex than that of linear functions and it is controlled by a far greater number of design hyperparameter. Neural models can be though of as directed graphs $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, named *architectures*, where the nodes represent the computation carried out at each layer. The edges encode the relationship of composition; i.e. $(i, j) \in \mathcal{E}$ if the (output) of the i-th node is an input of the j-th node. The graphs of RNNs and some types of GNNs are cyclic. As a simple example, Figure 3.1 depicts the computational graph of a two layers feed-forward neural network (see (2.10)). We refer the reader to Section A.2.2 for a discussion of computational graph in the more general context of algorithmic differentiation[5].

A neural network computational graph $\mathcal{G}$ may be in principle treated as a "monolithic" hyperparameter. This requires specifying the function that each nodes implements (including designating the relative "trainable weights") and the connectivity patterns between the nodes, that is the set of edges $\mathcal{E}$. In practice, however, virtually all neural network-based learning algorithms deal with particular types of manually designed architectures (e.g feed-forward NN, CNN, CNN with skip connections [He

[5]The type of representation sketched here and shown in Figure 3.1 is more compact than that of Section A.2.2. It allows for vector or matrix operations and it groups input variables (which are both x and the various weights) with intermediate computations. However, the graph in Figure 3.1 may be easily expanded to follow the formalism of A.2.2.

et al., 2016], . . .). Therefore, very often only a much smaller number of adjustable components of $\mathcal{G}$ is exposed as a series of simpler integer or categorical hyperparameters. For instance, a learning algorithms for training feed-forward neural networks may restrict the possible architectures to sequential graphs such as the one depicted in Figure 3.1. The user may be required, however, to specify the number of nodes (i.e. the depth of the neural network), the output dimensionalities of each node (i.e. the number of units per layer) and the activation function σ to use at each layer. The central part of Table 3.1 shows an indicative selection of typical architectural hyperparameters in this context.

A series of advancements in the neural network community have derived form the development of novel architectures. Until recently, however, this research process has been driven almost exclusively by expert insight and analysis as well (as manual trial-and-error) and thus quite outside the scope of HPO. The recent emergence of a research area that specifically focuses on the optimization of neural architectures [e.g. Zoph and Le, 2017, Cai et al., 2018, Liu et al., 2019, Luo et al., 2018, Real et al., 2019], called *neural architecture search*, has altered the perspective in this sector and has lead to the development of specialised HPO algorithms to tackle this problem. Although the way in which the architecture space and the search method are implemented by the various authors is quite varied, the general idea is to parameterize a relatively small subspace of all the possible computational graphs by the means of a domain dependent language [Zoph and Le, 2017, Real et al., 2017] or with real-valued [Liu et al., 2019] or boolean hyperparameters. Warm starting and network transformation (morphism) strategies may be used to speed up the search process [Cai et al., 2018]. See [Elsken et al., 2019] for a review on the topic. Interestingly, various authors [Zoph and Le, 2017, Cai et al., 2018] have framed the neural architecture search problem under a reinforcement learning point of view, perspective that is rarely explored in the wider HPO field[6].

Kernel Methods. The main design choice in kernel-based methods such as SVM and Gaussian Processes (see also Section 3.4.5) is instead represented by the kernel

[6] Few exceptions include [Hansen, 2016], that focuses on the adaptation of learning rates.

mapping (e.g. linear, polynomial, Gaussian, ...). Conditional to this (categorical) choice, each type of kernel offers a series of configuration parameters that must be set; for instance the degree of polynomial kernels. Some learning algorithms combine different kernels during the execution [Aiolli and Donini, 2015, Jain et al., 2012], in which case a list of available kernels with their parameters – or, alternatively, a method to generate them – should be provided.

Multi-task Learning. In a multi-task learning setting, one way to transfer knowledge between tasks and statistical models is to share sub-components of the underlying computation. This is, for instance, quite straightforward for neural networks, where models trained on different tasks can share entire layers, thereby learning a common representation by exploiting information from multiple sources. The exact choice of which portions of the computational graph should be shared may, generally, be represented as a design hyperparameter.

3.2.2 Regularization Hyperparameters

Regularization hyperparameters control the strength and form of the various regularizers that may be used by a learning algorithm to reduce overfitting and inject previous knowledge of the task at hand. If design hyperparameters shape the hypothesis space in which a learning algorithm seeks for solutions, most regularization hyperparameters affect the prior distribution over possible solutions. That is, the probability that the resulting solution will be in a certain region of $\mathcal{H}$ *irrespective of* (or, before seeing any) particular dataset[7]. Regularization is often associated to the concepts of (model's) capacity and complexity, which in turn, are tied to the interpolation capabilities of the model in question. Table 3.2 shows a selection of regularization methods, with a short description and specification of the type of the associated hyperparameters.

Many regularization methods consist in adding a penalty term to the training error (empirical risk) in the form of a non-negative map $\Omega : \mathcal{W} \times \Lambda_\Omega \to \mathbb{R}^+$ that is independent from the data, where Λ_Ω is the space of hyperparameters for Ω. The

[7] As a matter of fact, to give more meaning to this informal definition, one should introduce and consider a distribution p_D over the space of datasets $\mathcal{D}$, so that one may reason about the distribution $\mathcal{A}(D)$ induced by applying the learning algorithm to the random variable $D \sim p_D$. This view relates to a meta-learning approach. We shall formalize it more precisely in the next chapter.

Table 3.2: Examples of regularization methods and relative hyperparameters. The "description" column briefly explains the effect of each regularization technique. Hyperparameters – whose admissible range is displayed in the third column – typically determine the strength of the relative regularizer. Further, in the "Type" column, "LT" indicates that the relative regularizer is a term added to the (training) objective and the symbol † denotes non-smooth (possibly discontinuous) regularizers. Note that LT regularizers may appear in some formulations also as constraints.

Name	*Description*	Type
L^0 penalty	*Penalizes number of non-zero weights components, induces sparsity, linked to network pruning [Louizos et al., 2018]*	$\mathbb{R}^{+,\dagger}$, LT
L^1 penalty	*Penalizes absolute value of weights, induces sparsity, linked to feature selection*	$\mathbb{R}^{+,\dagger}$, LT
L^2 penalty	*Penalizes squared weights; outlier mitigation*	$\mathbb{R}^{+}$, LT
Spectral reg.	*Penalizes singular values of weight matrices [Sedghi et al., 2019, Miller and Hardt, 2019], stabilizes training*	$\mathbb{R}^{+}$, LT
Jacobian reg.	*Penalty term as a function of the Jacobian of the hypothesis w.r.t. the input [Hoffman et al., 2019], increase smoothness*	$\mathbb{R}^{+}$, LT
Weight decay	*Reduces magnitude of weights, linked to L^2 regularization [Loshchilov and Hutter, 2019]*	$(0,1]$
Dropout	*Stochastically drops neural units from the computation, prevents feature co-adaptation [Srivastava et al., 2014]*	$(0,1]^{\dagger}$
Secondary tasks	*Favours representations beneficial also for other tasks [Caruana, 1998]; hyperparameters may balance the importance of each task*	$\mathbb{R}^{+}$, LT
multi-task reg.	*Set of techniques and relative regularizers for learning and sharing information among multiple, equally important, tasks; hyperparameters may set the strength of the interactions between pair of tasks*	$\mathbb{R}^{+}$, LT

hyperparameter $\lambda \in \Lambda_\Omega$ plays a somewhat similar role to that of the parameter vector of a statistical model, although it is usually of a much lower dimensionality (or even, simply, a scalar). A learning algorithm that encapsulates this type of regularization typically searches for hypothesis that minimize

$$L(w,\lambda,D) = \frac{1}{N}\sum_{i=1}^{N} \ell(h_w(x_i), y_i) + \Omega(w,\lambda) \tag{3.9}$$

where w is the model's weight vector and $D = \{x_i, y_i\}_{i=1}^N$ is a dataset.

L^p **Regularizers.** The simplest implementations of Ω are of the type

$$\Omega(w,\lambda) = \lambda\tilde{\Omega}(w) \qquad \text{where} \quad \tilde{\Omega}(w) = \frac{\|w\|_2^2}{2} \quad \text{or} \quad \tilde{\Omega}(w) = \|w\|_1 \tag{3.10}$$

where λ is a non-negative scalar hyperparameter ($\Lambda_\Omega = \mathbb{R}^+$) that controls the global strength of the penalty[8]. While the presence of only one scalar hyperparameter certainly reduces tuning efforts and simplifies the resulting HPO problem, some applications benefit from a finer control over the prior induced by the regularizer [Chen et al., 2019]. This may be achieved, for instance, by letting λ be and hyperparameter matrix and setting $\Omega(w,\lambda) = \tilde{\Omega}(\lambda w)$. In this case, λ is typically restricted to be a diagonal matrix (one regularization coefficient per weight) or presents some other additional structure (e.g. one coefficient per neural network layer).

L^2 (or quadratic) regularization promotes solutions where the magnitude of the entries is relatively homogeneous (smaller as λ increases for the case of (3.10)). This may potentially reduce the effect of outliers in the dataset or any subsequent feature map. The classic "ridge regression" learning algorithm fits L^2 regularized linear models with mean squared error. Quadratic regularization of the type (3.10), being smooth, strongly convex and very cheap to compute, is also commonly employed to impose uniqueness on the solutions of linear over-parameterized problems (where $d > n$) and to prevent the norm of neural network weights to arbitrarily increase during training. It is possibly one of the most ubiquitous form of regularization that finds its place in several learning algorithms, from neural networks to SVMs.

L^1 regularization is used to encourage sparsity in the solution vector, whereby in the simple implementation of (3.10) higher values of λ "increase the probability" that more components of w will be set to 0. The related mapping is convex and Lipschitz-continuous but not smooth. Notably, the learning algorithm that performs linear regression minimizing an empirical squared error with an added L^1 regularization term is known as LASSO (least absolute shrinkage and selection operator), and it is linked to feature selection. L^1 regularization is often introduced as a convex relaxation of the L^0

[8] In this case (3.9) reduces to (2.3) with $\lambda = \rho$. In (3.9) we have further highlighted the dependence of L from λ.

"norm". The L^0 norm counts the non-zero entries of a vector but, unlike the L^1 norm, it is not affected by the magnitude of the vector itself[9]. As such, it would be an ideal candidate for promoting sparsity, without causing the possibly undesirable "shrinkage" side-effect associated to the L^1 norm. The discontinuous nature of the L^0 norm has, however, discouraged its widespread application as it considerably complicates the resulting learning problem. Recently, Louizos et al. [2018] proposed a stochastic reformulation closely tied to L^0 regularization by introducing auxiliary semi-discrete random variables whose distribution is optimized alongside the model's parameters during training.

Spectral and Jacobian Regularizers. Other form of regularization act on the model itself, rather than directly on its weights. These include spectral regularizers, where a penalty term is applied to the singular values (the spectrum) of linear operators that define h_w, and Jacobian regularization, whereby Ω is a function of $\mathrm{D}h_w$ [see e.g. Hoffman et al., 2019, and references therein]. In the simple case of affine models $h_w(x) = Wx + b$, an example of the first class of regularizer is given by

$$\Omega(h_w, \lambda) = \lambda \|\sigma_{\mathrm{SV}}(W)\|_2^2$$

where $\lambda > 0$ is a scalar coefficient and σ_{SV} computes and returns the singular values of a matrix. Spectral regularization may be applied also to more complex models such as CNN [Sedghi et al., 2019] and it is particularly useful for dynamical models (such as RNNs) to promote – or enforce, when used as a constraint – contractiveness of the learnt dynamics [Miller and Hardt, 2019]. See also Section 6.3.4 for a series of experiments in this setting.

In contrast to the other regularization methods reviewed so far, Jacobian regularization mappings typically depend also on the input data and may take the form of

$$\Omega(h_w, \lambda, D) = \frac{\lambda}{N} \sum_{i=1}^{N} \|D h_w(x_i)\|_F^2. \tag{3.11}$$

[9] To be precise, the L^0 norm defined as $\|v\|_0 = \sum_i \iota_{v_i \neq 0}$ is not a proper norm since it is not homogeneous: in general $\|cv\|_0 \neq |c|\|v\|_0$ for scalar c and vector v.

where $\lambda > 0$ controls the strength of the regularizer. Jacobian regularization promotes local smoothness of the solution hypothesis by penalizing the local Lipschitz-constants of h_w around the sample points. For classification tasks this may lead to overall smoother decision boundaries, which, in turn, may improve the model's generalization performances, particularly against corrupted or adversarial [Szegedy et al., 2013] examples. Since computing (3.11) may be expensive, especially as the output dimensionality grows, some stochastic approximation schemes have been proposed [Hoffman et al., 2019]. Spectral and Jacobian regularizers remain, at the moment, far less common than simpler L^1 and L^2 regularizers, possibly because of their computational overhead. Additional methodological, theoretical and empirical results may be needed before these type of techniques would see a broader utilization.

Other Neural Network Regularizers. Among the regularization methods that cannot can be expressed as additive loss terms (as in (3.9)) we mention weight decay [Loshchilov and Hutter, 2019] and dropout [Srivastava et al., 2014]. Both techniques are applicable to iterative learning algorithms; the first consists in iteratively decaying the model parameters as $w_{t+1} = \lambda w_t$ for $\lambda \in (0, 1]$ and is closely related to L^2 regularization[10]; the second, specifically developed in the context of neural models, involves adding Bernoulli noise to the neurons of each layer. More precisely, for a feed-forward neural network (2.10), the output of a layer subject to dropout regularization is a random variable distributed as

$$z_l(x) \sim \sigma(W_l z_{l-1}(x) + b_2) \circ r_l \qquad \text{where} \quad r_l \sim \mathrm{Ber}(\lambda \mathbf{1}),$$

where $\circ$ denotes the element-wise product and $\mathbf{1}$ is a vector of ones of the appropriate dimension. The hyperparamter $\lambda \in (0, 1]$ controls the amount of noise injected (less noise for $\lambda \to 1$). Dropout helps to prevent the co-adaptation of (hidden) features by sampling at each iteration a sub-networks, thus "dropping" from the computation a

[10]If, e.g., the iterative optimization dynamics is GD, weight decay is equivalent to L^2 regularization (3.10) for $\lambda_{\mathrm{WD}} = 1 - \eta\lambda_{L^2}$ where η is the learning rate of GD and λ_{WD} and $\lambda_{L^2} < \eta^{-1}$ are the hyperparameters of weight decay and L^2 penalty, respectively. For adaptive optimization algorithms such as ADAM, L^2 regularization is, however, no longer equivalent to weight decay [Loshchilov and Hutter, 2019].

subset of units at each layer. Typically, dropout is applied only at training time, while the final model does not include it; for this reason we consider it as a regularization method rather than a design choice. Dropout has been shown to improve generalization on a multitude of tasks, and various researchers have proposed adaptations and extensions in several directions [e.g. Wan et al., 2013, Krueger et al., 2017].

Multi-task Learning Regularizers. When learning from multiple task simultaneously, a common path to integrate and share knowledge between different tasks is through regularization. Concerning regularization, multi-task learning may refer to two different setting depending on the presence of a hierarchy among the various tasks. In a first case a subset of the input tasks is marked as primary, while the others are deemed as secondary (or ancillary) tasks [Caruana, 1998]. These latter are utilized by the learning algorithm to provide additional supervisory signal, but are not necessarily part of the target concept. A typical procedure involves sharing a portion of the computational graph (see Section 3.2.1) between models that attend to primary and secondary tasks. The training error is then defined as a weighted sum of all the single tasks' training errors (both primary and secondary tasks). The weights of the resulting errors are then treated as hyperparameters. Secondary tasks may act as regularizers by promoting learning of more general and richer representations. We discuss a practical instance of this setup in the context of automatic speech recognition in Section 7.1.2.

In a second case, the user is interested in obtaining one model per task, attributing (equal) importance to each task. In this scenario, knowledge may be transferred between related tasks by the means of a regularization mapping that takes as input multiple models. For instance, the weight of T different linear models might be pushed closer together by employing the following regularizer:

$$\Omega(w_1, \dots, w_T, \lambda) = \sum_{i=1}^{T} \sum_{j=1}^{T} \lambda_{ij} \|w_i - w_j\|_2^2$$

whereby the (symmetric) matrix of coefficients $\lambda = \{\lambda_{ij}\}_{i,j=1}^{T}$ may be given or treated as an hyperparameter. In Section 7.1.1 we conduct some numerical experiments concerning this setting. Another possible option in this setting is to penalize the distance of the weights of the various models from a vector $\bar{w}$, called bias [Evgeniou et al., 2005]. The resulting regularizer may then be written as

$$\Omega(w_1, \dots, w_T, \bar{w}, \lambda) = \sum_{i=1}^{T} \lambda_i \|w_i - \bar{w}\|_2^2,$$

where λ is an hyperparamter vector. The bias $\bar{w}$ may be prescribed, computed (e.g. it could be the mean of $\{w_i\}_{i=1}^T$) or treated as an hyperparameter. Multi-task regularizers may also work by encouraging outputs of closely related tasks to be similar to each other around sample points (in contrast of using a single model for all the tasks); this may be a more appropriate form of regularization for non-linear models which may exhibit symmetries in the parameter space $\mathcal{W}$. We explored this setting in [Badino et al., 2017] for learning speaker-dependent acoustic inversion models.

Clearly, a learning algorithm may employ more than one form of regularization, whereby the resulting hyperparamter space may be defined as the product of each regulaizer's space. For instance, the algorithm that uses both L^1 and L^2 regularization is known as elastic net [Zou and Hastie, 2005] in the context of learning linear models.

3.2.3 Optimization Hyperparameters

The majority of learning algorithms require solving one or more mathematical programming problems to compute intermediate or final results. When the problems are smooth and continuous, optimization methods described in Section 2.4 provide principled and efficient techniques to search for solutions[11]. These methods present, however, a series of configuration parameters that need to be set before execution, which we collectively denoted by α in the general Algorithm 1. Even though convergence results may offer some guidance and conditions, it is most often the case that the optimization objectives of interest do not verify some required preconditions (e.g. they may not be globally strongly convex, as it is the case for the training error of deep neural networks). Even if they do, it may be unfeasible to compute precisely or even estimate some important quantities, such as the Lipschitz smoothness constant (cf.

[11] Alternatives are given by evolutionary or population-based methods, that have largely different types of hyperparameters. We do not discuss these methods here, but see Section 3.4 for applications of these techniques to hyperparameter optimization.

Table 3.3: List of typical optimization hyperparameters. In the "Type" column, "Cat." stands for categorical, "Dist." denotes a probability distribution and * indicates that there are other possible encoding options.

Name	*Description*	Type
Update rule	*General form of the iterative update rule* (2.28)	Cat.
Starting point	*Distribution from which to sample* w_0	Dist.
Learning rate	*Global step-size that determines the magnitude of the updates*	$\mathbb{R}^+$
Termination	*Total number of iterations or rule to determine termination (e.g. early stopping; see Section 2.4.4)*	$\mathbb{N}^*$
LR schedule	*Rule for changing (reducing) the learning rate as a function of the iteration*	Cat.*
Mini-batch size	*Number of samples to process at each iteration for stochastic methods; see* (2.20)	$\mathbb{N}$
Momentum factors	*Coefficients of accelerated methods (e.g.* β *in* (2.25)*)*	[0,1)
Stability constants	*Small constants that ensure numerical stability (e.g.* ε *for ADAM* (2.27) *for preventing division by 0)*	$\mathbb{R}$
Patience window	*Number of iterations to wait before early stopping*	$\mathbb{N}$

Theorem 2.4.4). For these reasons configuration parameters of optimization methods are usually treated as hyperparameters, although they may be optimized against the training error rather than the validation error. We report in Table 3.3 a list of typical optimization hyperparameters with a brief description and the relative hyperparameter type. The first half of the table contains configuration choices that are virtually "essential" for setting up an iterative (gradient-based) optimization routine, while the second half lists hyperparameters that are either "optional" or tied to a smaller subset of optimization methods, such as the momentum factors.

The first choice regarding iterative optimization methods is the (symbolic) form of the update dynamics, denoted by Φ_t in the general scheme of Algorithm 1. Possible alternatives include GD and SGD, accelerated or second order methods and their stochastic variants, described briefly in Section 2.4.3. Similarly to some design hyperparameters, this categorical choice is often "hardwired" to the learning algorithm itself; as otherwise it would inject substantial conditionality on the rest of the hyperparameter space. Learning to optimize, which we discuss in the next chapter in Section 4.4.2.3, is a sub-branch of meta-learning that focuses on this choice, exploring the possibility of learning update rules from data.

Initialization. Secondly, iterative procedures need a starting point w_0: this is typically either set to an arbitrary, fixed, value or sampled from a probability distribution. The first option may be sufficient for problems with strongly convex objective (e.g. L^2-regularized logistic regression), but it is inappropriate for non-convex problem, especially in the presence of symmetries in the parameter space. There is a series of classic and more recent works about initialization strategies for neural network [Glorot and Bengio, 2010, He et al., 2015], the shape and parameters of the initial distribution may be, in general, treated as an hyperparameter. The starting point, paired with aggressive termination conditions (e.g. very few steps of GD), or fast decreasing learning rate schedules, may be also utilized as a regularizer to induce a prior over the possible outputs of the learning algorithm, especially in the context of transfer or multi-task learning (see also Section 4.3.3).

Learning Rate. The (initial) step size, or learning rate, controls the magnitude of the parameter updates. Except for a handful of particular situation (strongly convex objective where the Lipschitz constant and the modulus of strongly convexity are computable) in which it is possible to compute its optimal value, in most practical scenarios the learning rate must be treated as an hyperparameter. Numerous empirical works have shown that it is one of the most sensitive hyperparameters when training neural networks [Bergstra and Bengio, 2012, Bergstra et al., 2011]. The step-size is considered to be among the most important hyperparameters that should be tuned [Bengio, 2012]. Some authors advocate for the benefits of maintaining different learning rates for different parameters (e.g. different neural network layers) [Maclaurin et al., 2015a, Antoniou et al., 2019]: for several optimization techniques, this corresponds to applying a (structured) diagonal scaling. Usually the value of the learning rate is changed as a function of the current iteration – the exact rule, alongside its parameters may be treated as an hyperparameter. Typical choices include steps-wise, exponential or linear decay. Decreasing the learning rate is particularly important when using stochastic optimization methods to ensure convergence to a fixed point (see Section 2.4.2) and aggressive decreasing schedules may work similarly to early stopping or L^2 regularization in some instances. Given the importance of the learning rate and its scheduling, research on adaptive methods is abundant and predates that on general HPO [e.g. Jones et al., 1998]. The aim of these works is typically to reduce the number of iteration needed to reach good candidate solutions (i.e. to speed-up the training) as well as simplifying and automating[12] the hyperparameter search for the initial learning rate and its schedule. Chapter 8 presents additional related work in this area and is dedicated to the development of an algorithm based on online hypergradient descent to adaptively tune the learning rate, hence automatically producing a schedule.

Other hyperparameters related to the optimization aspect of machine learning algorithms include the mini-batch size for stochastic methods, momentum factors for adaptive and accelerated methods and some numerical stability constants that may appear, for instance, to prevent division by 0 (e.g. in RMSprop or ADAM).

3.3 Hyperparameter Schedules and Online HPO

When the underlying learning algorithm is inherently sequential, as it is for the case of Example 3.1.1 where the iterative dynamics is given by the SGD update step (Equation (2.20)), it may make sense to consider *hyperparamter schedules* rather than "fixed values". This can be achieved either by explicitly maintaining one distinct hyperparameter vector per steps, i.e. $\lambda = \{\lambda_t\}_{t=1}^{T}$, where $T \in \mathbb{N}$ is a prescribed horizon, or by letting the hyperparameters at step t be an appropriate function of the iteration (or even of a "state of the algorithm" [Wu et al., 2018a]), that is, letting $\lambda_t = s_t(\lambda)$. Either way, there is no contradiction with the formulation of Problem (3.6): the optimization variable remains λ in both cases, and we shall consider that all operations instrumental in managing the resulting schedules are wrapped into the learning algorithm itself. It is worthwhile noting, however, that the hyperparameter space may become sensibly larger than that of the "static" counterpart, especially when embracing the first formulation. This, in turn, may further complicate the HPO problem and render classic approaches to HPO quickly impractical. In spite of this, scheduling is particularly appealing. On the one side, it allows us to introduce and control a dynamic behaviour in $\mathcal{A}$ that may be advantageous in certain circumstances (think of the case of decreasing learning

[12]Although it should be noted that often these methods introduce other set of hyperparameters.

rates for SGD dynamics); on the other side, if a static value was, ex post, the best performing choice, then a solution to Problem (3.6) would be simply given by the constant schedule $\lambda^* = \left\{\bar{\lambda}\right\}_{t=1}^{T}$. In principle, an HPO technique should be able to recover it – although unnecessarily expanding the hyperparameter space may raise concerns on the generalization of the HPO solutions.

Hyperparameter schedules also emerge form online formulations of the hyperparameter optimization problem. Online HPO techniques [Baydin et al., 2018a, Luketina et al., 2016, Jaderberg et al., 2017, Lorenzo et al., 2017, MacKay et al., 2019, Donini et al., 2020] typically attempt to find a performing hyperparamter schedule within only one execution of the learning algorithm, potentially speeding-up the HPO procedure. To do so, these methods must clearly forgo computing the performance measure at the end of the evaluation of $\mathcal{A}$, but rather should adopt other strategies or case-specific heuristics. In this sense, they do not (or rather, cannot) strictly attempt to solve Problem (3.6). We come back to this topic in more details in Chapter 8, where we discuss and present an online gradient-based optimization method and study the case of tuning learning rate schedules.

For some hyperparameters – for instance, optimization hyperparameters such as the learning rate – scheduling is entirely natural. Scheduling has been proposed also for several regularization hyperparameters [Maclaurin et al., 2015a, Luketina et al., 2016, Franceschi et al., 2017, Morerio et al., 2017, MacKay et al., 2019], while some methods typically associated to neural architecture search or pruning [Louizos et al., 2018, Liu et al., 2019] may also be loosely interpreted as forms of scheduling of design hyperparameters (specifically, of connectivity patterns). Another notable examples include [Wu et al., 2018a] where the authors focus on optimizing a dynamic loss function that depends on a concise representation of the model being fit. Among the works cited in this section, only [Maclaurin et al., 2015a, Wu et al., 2018a] adheres to the formulation of Problem (3.6), while [Maclaurin et al., 2015a] uses explicit schedules and [Wu et al., 2018a] follows a functional approach. The remaining studies follow online formulations.

3.4 Techniques for Hyperparameter Optimization

In this section we review the most common approaches to hyperparameter optimization, from simple manual, grid and random search, to more complex model-based and population-based techniques. We defer the discussion of gradient-based HPO to the next section, where we examine in greater details the state of the art and several current research directions in the subfield.

The peculiarities of the HPO problem setting and goals translate into the fact that HPO techniques have typically different evaluation criteria than standard continuous optimization algorithms; beside computational cost and (speed of convergence toward the) final objective value, one may consider also other factors.

- *Applicability*: given the large heterogeneity of learning algorithms and corresponding hyperparameter spaces, certain HPO techniques can only be applied (at least off-the-shelf) on a restricted subset of problems. Furthermore, some methods may require (different levels of) access to the underlying learning algorithm and its structure, while others need much less information. Techniques in the first class may, for instance, be inapplicable in contexts where HPO is offered as a service [e.g. Golovin et al., 2017] and the user does not want to disclose reserved information. Methods that require only function evaluations are often called *black-box*.

- *Accessibility*, *automation* and *implementation overhead*: closely related factors that mainly reflect the complexity of the method and the presence of configuration parameters whose correct setting might require expert knowledge. Since democratization of machine learning may be considered as a goal of hyperparamter optimization, an argument can be made in favour of techniques that require little to no expert knowledge.

- *Scale* and *scalability*: some techniques may be more amenable to parallelization than others, while, on the other hand, certain methods may require a minimum level of parallelization to yield meaningful results – potentially limiting their applicability to highly distributed computing environments.

Correlated to the last point, we may, in fact, roughly dived HPO techniques into sequential and parallel methods. Sequential methods use information collected from previous executions of $\mathcal{A}$ to determine the next trial point, i.e.

$$\lambda_{k+1} = \pi\Big(\{\lambda_j, f(\lambda_j)\}_{j=1}^{k}, \upsilon\Big) \tag{3.12}$$

while parallel methods simultaneously establish a number of trial points

$$\{\lambda_i\}_{i=1}^{k} = \pi(\upsilon) \tag{3.13}$$

on which to probe the learning algorithm. We denote by υ the set of configuration parameters for the HPO method, while π represents the map associated to the HPO technique itself. Given the ample variety and different nature of the various HPO methods that we touch upon in this section, we do not attempt to formalize further π at this stage, but rather think of π as a "notational hook", useful to amalgamate the discussion. Some sequential methods may determine more than a trial point at a time, while parallel methods may exchange information between different runs during execution, for instance to allocate increased computing power to more promising executions. Furthermore, some hybrid methods (e.g. evolutionary search) comprises both parallel and sequential steps.

Regardless of the nature of the method, we will assume that the best configuration found so far is always retrievable. Also called *incumbent solution*, such point typically corresponds to the lowest value of f measured so far. We will denote it by $\hat{\lambda}_k \in \Lambda$, where the subscript k may either indicate an iteration or an amount of elapsed time, as appropriate. Finally, we denote by $\hat{\lambda} \in \Lambda$ (without subscript) the final configuration returned by the method upon termination.

3.4.1 Manual Search

Traditionally, the problem of hyperparamter optimization has been tackled with a mix of manual and grid search. Pure manual search (MS) may be described as a sequential approach whereby the map π of (3.12) is provided by the user's judgement and insight. Manual search heavily relies on the experience and skill of the user, making it the least accessible method for HPO. MS constitutes a potentially very time consuming approach, especially when facing novel datasets and tasks, in the absence of a well established literature. As it is not possible to reliably recreate and retrace the (mental) steps that led to a certain hyperparamter configuration, MS suffers heavily form reproducibility issues. Typically the access to the learning algorithm is complete and, as the user manually explores the response surface, he or she may collect additional information from each single run (e.g. looking at the activation patterns of a neural network's layer during training), which may then be used to guide the search. If, from one side, this procedure may grant the user a superior degree of insight on $f(\Lambda)$ and $\mathcal{A}$ itself, thereby speeding up the search, on the other side, it may lead to the (accidental) leakage of reserved information (e.g. the test error), potentially prejudicing the outcome of the experiment. In some communities, this issue is particularly severe, especially when it is paired with the heavy (re)utilization of relatively few benchmark datasets when presenting empirical research results. In this regard, Recht et al. [2018] have recently measured the performance drop of several learning algorithms (executed with the hyperparameters indicated in each respective original work) on CIFAR10 [Krizhevsky and Hinton, 2009] when the resulting models are evaluated on a new curated set of test images rather than the "standard" test split. They report an absolute drop in accuracy between 4% and 12%.

Notwithstanding all these issues, manual search (especially when paired with grid search) remains to date one of the most popular method to perform HPO, as it causes virtually no implementation overhead and, in principle, it does not have any applicability restriction. Another potential advantage of manual search is that it has a low computational footprint, as the total number of trials generated by a typical run of MS is comparatively much smaller than that of other methods as the richer feedback loop and previously accumulated knowledge may tremendously boost the efficiency of the search policy of an experienced user.

3.4.2 Grid Search

The first formal description of grid search (GS) dates back at least to the thirties and is attributed to Fisher [1936]; in his foundational work on experimental design, Fisher refers to the method as *factorial design*. The classical approach to hyperparamter optimization in machine learning, grid search is a parallel method that consists in executing $\mathcal{A}$ over a predefined grid of values. Specifically, for each entry of λ, the user defines a list of values (trial sets) as

$$\bar{\lambda}^i = \{\bar{\lambda}^i_j\}_{j=1}^{n_i}, \quad \bar{\lambda}^i_j \in \Lambda_i, \quad \text{for } i = 1, \dots, m.$$

The grid of values $\upsilon = \{\bar{\lambda}^i\}_{i=1}^m$ is a configuration parameter for GS. The trial points are then given by all the $K = \prod_i n_i$ possible combinations on the Cartesian product generated by the m sets in υ. The learning algorithm is executed (in parallel, if possible) on the resulting set of points and the hyperparameter configuration that minimizes the objective function f is finally returned as $\hat{\lambda}$.

Grid search is a simple black-box method that may be reliably used for exploring low dimensional hyperparamter spaces, it requires very little implementation effort and it is easily scalable. The number of total experiments generated by GS, however, grows linearly in the size of the trial sets $\bar{\lambda}^i$ and is exponential in m; making the method largely impractical already form $m > 3$ and virtually inapplicable when m is in the order of 10. In the very favorable case that λ comprises only Boolean values, for $m = 10$ grid search generates already $2^{10} = 1024$ trial points. In addition, while Boolean and categorical hyperparameters naturally lend themselves to an evaluation on a predefined grid, unbounded integer and real-valued hyperparameters require manually setting up bounds and discretization. Extensive experimental practice [Bengio, 2012] has shown that some continuous hyperparameters, such as the learning rate, benefit from discretization on a logarithmic scale. One may, for instance, expect that, for some $a < b$ in $\mathbb{Z}$, with $p = b - a + 1$ the total number of trial points, a trial set of the type $\{10^j : j \in \{a, \dots, b\}\}$ would yield more significant results than an equally spaced grid between 10^a and 10^b defined as $\{10^a + j(p-1)^{-1}(10^b - 10^a) : j \in \{0, \dots p-1\}\}$. Thus, while requiring less expert knowledge than manual search, grid search may still be quite sensitive to a correct setting of the trial grid υ, especially for "non-standard" hyperparameters or problem settings for which there is lack of abundant previous experimental evidence.

To counter these limitations and contain the computational overhead, grid search is usually paired with manual search: the user alternates GS steps with adjustments and refinement to the trial sets (i.e. using a finer grid "centered" around the best found configuration, or moving the search bounds of continuous hyperparameters), until a satisfactory result is found.

3.4.3 Random Search

Perhaps the simplest procedure for global optimization, random search (RS) was probably first expounded in the fifties in a work by Brooks [1958]. Random search is a parallel black-box approach that consists in repeatedly drawing hyperparameter configuration from a predefined distribution over Λ. We will indicate such (joint) distribution with υ; π represents then the sampling process. While in principle υ could be any distribution, in practice, to keep the method accessible and uncomplicated, most of the times the joint distribution is given by a product of independent component-wise simple distributions relative to each of the specific entries of the hyperparameter vector, i.e. $\upsilon = \prod_{j=1}^{m} \upsilon_i$. For instance, if λ contains, in order, a categorical choice between K different neural architectures, the learning rate for SGD and a global dropout factor for regularization (see Section 3.2) then υ could be set as

$$\upsilon = \mathcal{U}\{1, K\} \times \mathrm{Log}\mathcal{U}(10^{-5}, 10^{-1}) \times \mathcal{U}(0.5, 1).$$

where $\mathcal{U}\{\cdot\}$ and $\mathcal{U}(\cdot)$ are the discrete and continuous uniform distributions and $\mathrm{Log}\mathcal{U}(\cdot)$ is the log-uniform (or reciprocal) distribution on the indicated intervals.

In an influential article, Bergstra and Bengio [2012] advocated using random search (instead of manual or grid search) as the "default" baseline method for HPO. They argue that RS retains the advantages of GS such as reproducibility, conceptual simplicity, very low implementation overhead and trivial parallelization, while also being more efficient in higher dimensional spaces, especially when the response function has a *low effective dimensionality*[13]; situation that they claim to be rather

[13] The authors informally define this condition as when a subset of factors (hyperparameters) accounts for most of the variation of the response function: e.g. for a bi-dimensional hyperparameter it could be that $f(\lambda_1, \lambda_2) \approx g(\lambda_1)$.

common in practice. In fact, as Brooks explains, the success of random search is linked to the relative size of the sub-region $\Lambda^* \subset \Lambda$ that leads to positive outcomes of the experiments (i.e. where f is sufficiently small). Assuming υ is the uniform distribution over Λ and that the user is satisfied whenever the search returns a configuration in Λ^*, one can simply compute the probability of success of random search after k trials as [Brooks, 1958]

$$\mathbb{P}\left(\hat{\lambda}_k \in \Lambda^*\right) = 1 - \left(1 - \frac{\mu(\Lambda^*)}{\mu(\Lambda)}\right)^k, \tag{3.14}$$

where μ is a suitable measure (e.g. the Lebesgue measure if Λ is continuous, or the counting measure if Λ is discrete). Equation (3.14) says that the minimum number of trials needed to assure a positive outcome with a given probability (say 95%) is independent from the dimensionality of the search space, and only depends on the relative size of the favourable region. This argument subsumes that of Bergstra and Bengio. Functions with low effective dimensionality are but a particular case, since one may "safely" disregard the insensitive factors of variation when estimating (3.14). This argument may also offer a simple and intuitive explanation to the stunning success of random search in seemingly complex settings such as neural architecture search [Li and Talwalkar, 2020]: repeated experimentation on few well-known benchmark datasets has lead to the development of rather amenable search spaces for neural architectures where the favourable subspace ratio is comparatively high.

However, for more general search spaces and problem settings the situation may be quite different: one may indeed expect that the ratio $\mu(\Lambda^*)/\mu(\Lambda)$ itself depends from the dimensionality of Λ. Consider, for instance, the m-dimensional hypercube $\Lambda = [0, 1]^m$ (so that $\mu(\Lambda) = 1$) and assume that, for each component, the favourable region is a tenth of the total. Then $\mu(\Lambda^*) = 0.1^m$, which goes quickly to 0 as m grows. Thus, while random search may indeed be more efficient than grid search on specific problems, especially when the search space is well understood, it still suffers, in general, from the same drawbacks of grid search regarding scalability to high dimensional settings. Furthermore, promising values of certain hyperparameters (e.g. the learning rate) may concentrate around a small region of the domain under the Euclidean measure, when sampling uniformly. One may then pick a distribution with more mass on that small region (e.g. using a log-uniform distribution for the learning rate), leading to a different measure in (3.14). This exercise, however, clearly requires a degree of prior knowledge which may not always be available. In other words, in order for random search to be noticeably more effective than grid search, one either needs to design a "benign" search space with a relative large region of good configurations, or must be able to provide a prior distribution that induces a favourable measure over Λ, either options demanding experience. Anyway, one clear advantage of random search is that it allows for a more natural treatment of a larger class of hyperparameters, not requiring discretization of continuous search spaces and allowing (in principle) for distributions with non-compact supports.

Finally, we note that the randomized nature of this HPO technique may have two other potential disadvantages. First, "unlucky" runs may end up with entire regions of Λ being under-explored or completely neglected. Secondly, also in view of the previous reason, an execution of a "round" of RS may not necessarily provide enough information to meaningfully modify the settings for an eventual subsequent round of RS. Thus, interleaving random search with manual search steps may be comparatively harder than doing so with grid search. To obviate these potential issues, one may resort to hybrid (grid/random) strategies [see Bousquet et al., 2017, and references therein]. One of these is Latin hypercube sampling, which involves dividing the search space into K hypercubes of roughly the same size and then drawing exactly one configuration per cube, either uniformly or according to a predefined prior distribution over each cube. More generally, one may resort to open-loop sampling strategies, whereby ν models a joint probability distribution over sequences of hyperparameters $\{\lambda_i\}_{i=1}^K$ (see (3.13)). In this latter context, Dodge et al. [2017] propose determinantal point processes to promote diversity among the configurations of the sampled sequences.

3.4.4 Model-based Methods

The techniques described so far do not make any attempt to model the response function $f(\lambda)$ but only rely on the evaluation of the objective on a set of selected or drawn points. Except for manual search, the information retrieved on previous runs of the learning algorithm does not influence the progression of the search. In contrast, model-based (MB) techniques construct an explicit *surrogate model* for $f(\lambda)$ – or for closely related quantities – on the basis of information collected from previous evaluations. We will denote such a model by

$$\hat{f} : \Lambda \times \mathcal{X} \to \mathcal{Y}, \tag{3.15}$$

where $\mathcal{X}$ may represent additional inputs (e.g. the performance of partial executions of $\mathcal{A}$, a baseline or the value of the incumbent solution) and $\mathcal{Y}$ is an appropriate output space. For instance, $\mathcal{Y}$ could be $\mathbb{R}^+$ if $\hat{f}$ models directly f (and f is a validation error of the type (3.5)), or $\mathcal{Y}$ could be the interval $[0,1]$ if $\hat{f}$ predicts the probability that a given configuration improves over a given baseline.

The panorama of model-based methods is strongly dominated by techniques that fall under the umbrella of *Bayesian optimization* (BO), which we describe in more details in the next section. In BO the surrogate model is a probabilistic mapping that also captures the uncertainty of the estimation. Nonetheless, some recent works go in different directions; Ilievski et al. [2017] use a deterministic radial basis function model for $\hat{f}$ which they interpolate on observations of f, while Hazan et al. [2018] focus on Boolean hyperparameters and use a sparse Fourier basis model for $\hat{f}$, in conjunction with heuristics for estimating the most sensitive hyperparameter entries. They then use a base global optimizer for performing HPO on a restriction of the original hyperparameter space. The works by Lorraine and Duvenaud [2018] and MacKay et al. [2019], linked to gradient-based hyperparameter optimization may also be interpreted as an attempt to learn a surrogate model of the entire learning algorithm (applied to a specific dataset), whereby the output space of $\hat{f}$ coincides with the hypothesis space of $\mathcal{A}$ itself, that is $\hat{\mathcal{Y}} = \mathcal{H}$ in (3.15). We will resume the discussion about these latter methods in Section 3.5.

Regardless of the particular implementation, one critical feature of surrogate models is that they should be much cheaper to evaluate than f itself (we recall that evaluating f requires executing the learning algorithm $\mathcal{A}$, which may be very expensive). This typically allows model-based techniques to compute $\hat{f}$ on several values at each step, e.g to seek for extrema. To conform with the notation introduced in (3.12) and (3.13), we let the surrogate model be passed to π as (part of) the configuration parameter υ; that is, we let $\upsilon = (\upsilon_1, \upsilon_2) = (\hat{f}, \upsilon_2)$, where υ_2 is a (possibly empty) set of other configuration parameters. $\hat{f}$ may be a parametric function itself, thus one may think of υ_1 as the set of variables that define the surrogate model.

The vast majority of model-based methods is sequential: the surrogate model suggests the next point to evaluate while being simultaneously adapted online as new data from the execution of $\mathcal{A}$ is collected and processed. In this case, π may be conveniently divided into two components π_1 and π_2, one corresponding to the iteration in the hyperparameter space, the other to the update of the surrogate model:

$$\begin{pmatrix} \lambda_{k+1} \\ \hat{f}_{k+1} \end{pmatrix} = \begin{pmatrix} \pi_1(\hat{f}_k, \upsilon_2) \\ \pi_2\left(\{\lambda_j, f(\lambda_j)\}_{j=1}^k, \hat{f}_k, \upsilon_2\right) \end{pmatrix} \tag{3.16}$$

Another possibility is to first learn $\hat{f}$ and then use it to suggest one or several hyperparameter configurations in a second stage, following a parallel scheme. Examples of this strategy include [Feurer et al., 2015, Fusi et al., 2018] which, however, assume the presence of multiple tasks and are closer in nature to meta or transfer learning settings. Still, even in the sequential approach, typically $\hat{f}_0$ is initialized by using several evaluations obtained in a "warm-up stage", e.g. by random search or Latin hypercube sampling. While the foremost representatives of these type of methods – tied to Bayesian optimization – are classically black-box procedures, model-based techniques may greatly benefit from direct access to the underlying learning algorithm. For instance, [Swersky et al., 2014, Klein et al., 2017, Falkner et al., 2018] use information gathered from partial execution of $\mathcal{A}$ to stop unpromising evaluations, which may be additionally resumed at a later stage [Swersky et al., 2014].

Model-based methods tend to be fairly more complex than other HPO techniques, both conceptually and from an implementation point of view. This is because of the additional challenges induced by the need of fitting a mapping – the surrogate model – based on relatively few function evaluations, as well as being able to extract meaningful information from said mapping (e.g. by using handcrafted acquisition functions, as we shall see in the next section). Thus, while there is a thriving ecosystem of free academic software, it is not rare that model-based HPO techniques are implemented and proposed as commercial products [Clark and Hayes, 2019, Golovin et al., 2017]. The development of MB techniques is often carved with several design choices – one above all, the hypothesis space for $\hat{f}$. Some configurations may be left to be set by the final user. Although, often, the authors provide "default values", the presence of many such parameters may potentially hider accessibility of some model-based techniques [see, e.g., the list of required inputs for the algorithms proposed in Hazan et al., 2018, Falkner et al., 2018].

3.4.5 Bayesian Optimization

Bayesian optimization has long been used as a method for global optimization of black-box functions (i.e., functions for which we can only observe the value $f(\lambda)$ at any point $\lambda \in \Lambda$), and dates back at least to the seventies with a series of works by J. Močkus and colleagues [Močkus, 1975, Močkus et al., 1978]. Since Bayesian optimization constitutes a sensible approach for optimizing (highly) multi-modal, non-convex and possibly expensive-to-compute functions – albeit limited to small-to-medium dimensional settings – it has attracted considerable attention for its potentiality to tackle HPO problems, especially following the publication of three influential papers in the early 2010s [Bergstra et al., 2011, Hutter et al., 2011, Snoek et al., 2012]. In this section, we offer a brief overview of the general methodology in the HPO context. We refer the reader to the work by Jones et al. [1998] for an intuitive introduction of the mathematical aspects of the methodology and to the review by Shahriari et al. [2015] for further insights into practical aspects and applications to machine learning.

BO is a model-based technique where the objective f is modelled with a stochastic process, fitted to the observations of the function's value at a series of points. In turn, the stochastic process is used to suggest the next value to probe. Typically the suggestion rule, express as the maximization of an *acquisition function*, balances *exploration* – that is, the need of collecting information to improve the model itself, for instance by proposing points far from previous observations – and *exploitation* – namely the exigency of "trusting" the surrogate model for rapidly finding good regions, for instance by choosing as next iterate a minimizer of the model inferred so far.

Referring to the notation of the previous section and to Equation (3.16), the stochastic process maps to the surrogate model $\hat{f}$, the selection rule to the first component of π while the fitting steps constitutes the second component, π_2. In the following, we describe a particular, "classical", instantiation of Bayesian optimization whereby the surrogate model is given by a Gaussian process [Williams and Rasmussen, 2006] and the acquisition function is the expected improvement [Močkus et al., 1978]. This approach is well suited for the global optimization of smooth multivariate real functions, and constitutes the core of the Spearmint [Snoek et al., 2012] HPO package.

We start by introducing a simple linear model h_w (Section 2.2) in a given feature space $\mathcal{V}$, to which we add independent Gaussian noise with variance σ^2:

$$h_w(\lambda) = \chi(\lambda)^\intercal w + \varepsilon; \qquad \varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2), \tag{3.17}$$

where $\chi : \Lambda \to \mathcal{V}$ is a feature map. While using linear maps directly in the input space would surely result in an overly simplistic model – we are trying to model an objective, the validation error, we believe to be (highly) non-convex and multimodal – working in a different feature space allows us to capture much richer behaviours, but still retain most of the benefits of the linear formulation [Murphy, 2012, Ch. 14]: evaluating f is expensive: we are expecting to work with only a few observations. The noise in (3.17) models potential errors in the observations. In hyperparameter optimization, the noise originates from the fact that the validation error is only an estimate of the true generalization error, but may also (partially) account for the underlying stochasticity of the learning algorithm.

Suppose now we are at the k-th iteration of the algorithm, after having already collected k observations. Let $Y_k = (f(\lambda_i))_{i=1}^k \in \mathbb{R}^k$ be the vector of observed values, and denote by $X_k = (\chi(\lambda_1), \dots, \chi(\lambda_k))$ the corresponding design matrix in the feature space (organized by columns). We are interested in obtaining a probability distribution over the output of f at a query point $\lambda_* \in \Lambda$; that is, in computing

$$y_* \sim \hat{f}_{k+1}(\lambda_*) = p_{y_*}(\cdot | x_*, X_k, Y_k),$$

where $x_* = \chi(\lambda_*)$ is the query point in the feature space. We could follow an empirical risk minimization approach, obtaining the weights $w_k \in \mathbb{R}^{\dim(\mathcal{V})}$ of the model (3.17) by minimizing, for example, a (regularized) mean squared error over the dataset (X_k, Y_k). We could then directly set $\hat{f}_{k+1} = h_{w_k}$. Such an estimate would, however, leave us with a rather meager predictive distribution $p_{y_*}(\cdot|x_*, X_k, Y_k, w_k)$, capable of capturing only our belief of the amount of noise present in the observations. Rather, we would like the stochasticity of $\hat{f}$ to (primarily) represent our uncertainty about the behaviour of f in unexplored regions[14]. To this aim, we take a Bayesian perspective on the fitting problem, and reason about all the linear models (in the feature space) that agree with the observations, weighted by their "realization probability".

More formally, we marginalize over the weights w and set

$$\hat{f}_{k+1}(\lambda_*) = \int p_{y_*}(\cdot|x_*, X_k, Y_k, w) p_w(\cdot|X_k, Y_k) \, \mathrm{d}w \tag{3.18}$$

Given our modelling assumption (3.17) and the hypothesis that the noise is independent, the first distribution simply reads as

$$p_{y_*}(\cdot|x_*, X_k, Y_k, w) = p_{y_*}(\cdot|x_*, w) = \mathcal{N}(h_w(\lambda_*), \sigma_\varepsilon^2) = \mathcal{N}(x_*^\intercal w, \sigma_\varepsilon^2). \tag{3.19}$$

By the Bayes' rule, $p_w(\cdot|X_k, Y_k)$ is, instead, given by

$$p_w(\cdot|X_k, Y_k) = \frac{p_y(\cdot|X_k, w) p_w(\cdot|X_k)}{p_y(\cdot|X_k)}, \tag{3.20}$$

[14] The following argument, mutated from [Jones et al., 1998], may help in better grasp the difference between these two aspects. Consider the case where f is a non-linear smooth deterministic function. We may then either omit the noise term ε in (3.17), or leave it for capturing modelling errors. In the first scenario, the ERM approach would yield a single point estimate for any $\lambda \in \Lambda$, which is very unlikely to be accurate, especially for points far from those previously sampled. Conversely, if we still wish to maintain a noise component, $\hat{f}_{k+1}$ would banally be inaccurate at λ_i for $i \leq k$. Thus, informally speaking, both ERM modelling options would leave us with a surrogate model that is inaccurate with probability 1 for "all except a few" f's. A way to escape this apparent modelling stalemate is to replace the independent Gaussian noise with one that depends on the evaluation point, i.e. $\varepsilon = \varepsilon(\lambda)$. Then, when fitting the model to the k observations, one may have $\varepsilon_k(\lambda_i) = 0$ for $i \leq k$ while letting $\varepsilon_k(\lambda) \neq 0$ for $\lambda \neq \lambda_i$. Furthermore, by continuity (ε_k is smooth and continuous as difference of C^1 functions), $\varepsilon_k(\lambda) \to \varepsilon_k(\lambda_i) = 0$ as $\lambda \to \lambda_i$. The Bayesian perspective offers a principled way to achieve this behaviour – but note that the noise term ε in (3.17) remains independent and indeed represents, there, the measurement noise.

Regarding the numerator of (3.20), we have that, for the same argument of (3.17),

$$p_y(\cdot|X_k, w) = \mathcal{N}(X_k^\intercal w, \sigma_\varepsilon^2 I).$$

This is the likelihood of the observations given the model. We must instead take a decision regarding the second term of the numerator, that is the distribution of the model parameters. We make the very natural assumption of independence from the design matrix and set

$$p_w(\cdot|X_k) = p_w = \mathcal{N}(0, \Sigma) \tag{3.21}$$

where Σ is a positive definite covariance matrix. This is a so-called *prior*. Equipped with (3.21), we can now also compute the denominator of (3.20) by marginalizing over w [Shahriari et al., 2015], obtaining

$$p_y(\cdot|X_k) = \int p_y(\cdot|X_k, w) p_w \, \mathrm{d}w = \mathcal{N}(0, X_k^\intercal \Sigma X_k + \sigma_\varepsilon^2 I) \tag{3.22}$$

We recognize, now, that the feature map only appears in a quadratic form where the (i, j)-th entry is given by $\chi(\lambda_i)^\intercal \Sigma \chi(\lambda_j)$: a dot product in the feature space. We then introduce the corresponding kernel

$$\kappa_\Sigma(\lambda, \lambda') = \chi(\lambda)^\intercal \Sigma \chi(\lambda') = \langle \chi(\lambda), \chi(\lambda') \rangle_\Sigma \tag{3.23}$$

and rewrite (3.22) in terms of the Gram matrix $K = \{\kappa_\Sigma(\lambda_i, \lambda_j)\}_{i,j=1}^k \in \mathbb{R}^{k \times k}$ as

$$p_y(\cdot|X_k) = \mathcal{N}(0, K + \sigma_\varepsilon^2 I).$$

We finally have all the terms to compute (3.18); which is, again, Gaussian [Williams and Rasmussen, 2006]:

$$\hat{f}_{k+1}(\lambda_*) = p_{y_*}(\cdot|x_*, X_k, Y_k) = \mathcal{N}(k_*^\intercal (K + \sigma_\varepsilon^2 I)^{-1} Y_k, \kappa(\lambda_*, \lambda_*) - k_*^\intercal (K + \sigma_\varepsilon^2 I) k_*) \tag{3.24}$$

where $k_* = \{\kappa_\Sigma(\lambda_*, \lambda_i)\}_{i=1}^k \in \mathbb{R}^k$ is the vector of kernel evaluations between the query and the observations points. Equation (3.24), despite its apparent complexity, describes a rich *posterior* distribution over the values of f that is analytically computable and whose design depends essentially only on the definition of the kernel map (3.23).

Examples of kernel functions include the $\mathcal{C}^\infty$ squared exponential

$$\kappa_\Sigma(\lambda,\lambda') = \sigma_0 e^{-r^2_\Sigma(\lambda,\lambda')/2}; \qquad r^2_\Sigma(\lambda,\lambda') = \lambda^\intercal \Sigma \lambda',$$

where $\sigma_0 > 0$ is a scale parameter and Σ a diagonal matrix with positive length scales $\{\sigma_i\}_{i=1}^m$ and the Matérn family, which contains kernels (defined as functions of r_Σ, above) with various degree of smoothness [Williams and Rasmussen, 2006]. Kernels intuitively encode the concept of distance or similarity between data points, and allow us to sidestep the definition of (possibly "less interpretable") feature maps and covariance matrices. The series of "free parameters[15] " that appears in (3.24) – usually called *hyperparameters* in the Bayesian literature (not to be confused with λ!) – do not have to be fixed constants, but may be estimated from the observations [Williams and Rasmussen, 2006, Shahriari et al., 2015] (automatic relevance determination). Thus, at this stage, the design choice left to the experimenter mainly boils down the choice of the kernel function.

As anticipated, what we have described so far may be conveniently summarized with the notion of *Gaussian process* (GP) [Williams and Rasmussen, 2006, Murphy, 2012], a particular type of stochastic process that is fully specified by its mean and covariance function. In particular, our derivation corresponds to a zero-mean Gaussian process (this is because of the zero mean prior over the weights (3.21)) with covariance function given by $\kappa_\Sigma + \sigma_\varepsilon^2\delta$ where $\delta(\lambda,\lambda') = 1$ if $\lambda = \lambda'$ and 0 otherwise. In short, we can write

$$\hat{f} = \mathcal{GP}(0, \kappa_\Sigma + \sigma_\varepsilon^2\delta).$$

The surrogate model update equation π_2 (see (3.16)) may then be written as

$$\hat{f}_{k+1} = \pi_2\left(\{\lambda_j, f(\lambda_j)\}_{j=1}^k, \xi_2\right) = \mathcal{GP}(0, \kappa_\Sigma + \sigma_\varepsilon^2\delta \,|\, X_k, Y_k), \qquad (3.25)$$

[15] That is, σ_ε, σ_0 and the diagonal entries of Σ when using the squared exponential or a Matérn kernel.

which expresses the update as the conditioning of the Gaussian process on the observations X_k, Y_k.

Having defined a surrogate model, we now turn our attention to the suggestion step π_1. One could think of minimizing directly the surrogate model and choose $\lambda_{k+1} \in \arg\min_\lambda \mathbb{E}[\hat{f}_k(\lambda)]$. This procedure would, however, be very sensitive to local minima and could lead to entire regions of the search space unexplored. Instead, the selection problem is usually viewed through the lens of an *acquisition* function that determines the utility of adding the point λ_{k+1} to the observations set. The particular expression of this mapping constitutes another design choice when employing BO methods. Research in the development of acquisition functions has been very active in the last two decades [see Shahriari et al., 2015, for a survey]; with information based criteria such as max-value entropy search [Wang and Jegelka, 2017] at the forefront. A classic choice, already introduced in the seminal work by Močkus et al. [1978] and used also in the context of HPO [Snoek et al., 2012], is the *expected improvement*. The expected improvement over a given baseline $\beta \in \mathbb{R}$ is defined, at iteration k, as

$$a_{\beta,\hat{f}_k}(\lambda) = \mathbb{E}_{\hat{f}_k}\left[\max\left\{\beta - \hat{f}_k(\lambda), 0\right\}\right].$$

and it can be computed in closed form when $\hat{f}_k$ is the conditional Gaussian process introduced above [Feurer and Hutter, 2018]. A sensible choice for the baseline β is the minimum value of f measured so far, i.e. $\beta = f(\hat{\lambda}_k)$. Then, assuming $a_{\beta,\hat{f}_k}$ has a unique maximizer, one may set

$$\lambda_{k+1} = \pi_1(\hat{f}_k, \nu_2) = \arg\max_{\lambda} a_{f(\hat{\lambda}_k),\hat{f}_k}(\lambda). \tag{3.26}$$

Note that the acquisition function is much cheaper to compute than f itself, meaning that the problem (3.26) may be, in general, optimized thoroughly.

Equations (3.26) and (3.25) specify, thus, a particular simple instantiation of a GP-based Bayesian optimization algorithm that may be used for HPO optimization. To summarize, the algorithm's configuration parameters ν_2 may contain choices regarding the mean and covariance functions of the Gaussian process, the method with which the process's hyperparamters are estimated and the acquisition function to use.

While undoubtedly elegant, the above presented approach has a number of drawbacks that may limit its applicability to real-world scenarios. It does not natively support parallel execution, it scales cubically in the number of points k (due to the matrix inversion in (3.24)) and does not scale well in the dimensionality of Λ. Finally, it may not be suitable for integer, categorical and conditional hyperparameters as showed in a comparative study by Eggensperger et al. [2013]. Snoek et al. [2012] the first problem, proposing an extension of the expected improvement that accounts for pending executions, and take on the scalability issues in [Snoek et al., 2015] by replacing the GP with a Bayesian neural network. Hutter et al. [2011] and Bergstra et al. [2011, 2013], focus both on scalability and applicability to discrete and conditional search spaces by using random forsets and tree Parzen estimators, respectively.

More recent advances in the field [Springenberg et al., 2016, Wu et al., 2017, Klein et al., 2017] focus on replacing GPs with more practical and scalable approaches, exploiting additional information beside function evaluations (thus "dropping" the black-box assumption for f). The additional information could be e.g. the performance of $\mathcal{A}$ on a small subsets of D [Klein et al., 2017] or (possibly noisy) gradients [Wu et al., 2017]. Finally, Falkner et al. [2018] propose to hybridize Bayesian optimization with bandit-based methods (see next section) to improve performances in the first stage of the HPO process, when the surrogate model may be still rather unreliable.

3.4.6 Population-based Methods

Population-based methods (PB) for HPO are a fairly large class of hybrid parallel-sequential algorithms that maintain a set of candidate configurations – a *population* – that changes or evolves after each iteration, or *round*. The defining characteristic of PB methods is the presence of a global update policy π that takes into account information (or observations) derived from each "individual" (single runs) in the population. Evolutionary search [Friedrichs and Igel, 2005, Loshchilov and Hutter, 2016, Real et al., 2020], swarm optimization [Jaderberg et al., 2017, Lorenzo et al., 2017] and bandit-based algorithms [Jamieson and Talwalkar, 2016, Li et al., 2017a] belong to this class. By contrast, simpler parallel methods such as grid or random search do not implement this type of behavior. In other words, given the initialization configuration (i.e. the grid or prior distribution) runs generated by GS or RS are mutually independent, while PB methods generate configurations based on observations from previous rounds.

Let $k = 1,2,\dots$ indicate the iteration; we denote by $o_k(\lambda)$ the set of observations collected at the k-th iteration, relative to the hyperparameter λ. Then, most BP methods can be expressed as

$$\left\{\lambda_i^{k+1}\right\}_{i=1}^{n_{k+1}} = \pi\left(\left\{\lambda_j^k, o_k(\lambda_j^k)\right\}_{j=1}^{n_k}, \upsilon\right),$$

where $n_k \in \mathbb{N}$ denotes the number of configurations maintained at the k-th rounds. In the evolutionary literature, a batch of configurations $\left\{\lambda_i^k\right\}_{i=1}^{n_k}$ is often referred to as a *generation* and o is often called *fitness* function. The observations collected may match with the original HPO objective (hence $o_k = f$ for $k \geq 1$); yet, several methods in this class [Jaderberg et al., 2017, Lorenzo et al., 2017, Jamieson and Talwalkar, 2016, Li et al., 2017a] exploit the iterative nature of many learning algorithms and use as observations (a validation error after) partial executions of $\mathcal{A}$. Additionally swarm optimization methods generate schedules rather than "static" hyperparameters (see Section 3.3). At $k = 0$, $o_0 = \emptyset$ and the population may be initiated by selecting n_0 points on a grid or by sampling from a prior distribution (as in random search).

The nature of the mapping π may greatly differ from method to method. Evolutionary search (ES) techniques typically maintain a fixed population of $n_k = n$ configurations. At each round, "individuals" are selected, eliminated, recombined and mutated according to their fitness score, loosely mimicking principles of evolutionary theory [Friedrichs and Igel, 2005]. For instance, configurations may be sorted according to $o_k(\lambda_j^k)$; the first, best performing, third may be kept unvaried, the second third may be mutated (e.g. some components may be varied according to some rule or distribution) and the last third of worst performing hyperparameters may be replaced either by configurations drawn from the initial distribution, or by the "offspring" (i.e. random combinations) of the best performing individuals. A particularly popular and effective instantiation of this framework is the CMA-ES strategy (covariance matrix adaptation evolutionary strategy) [Hansen and Ostermeier, 1996, 2001], where the mutation operation is modelled after an additive multivariate

Gaussian random variable whose covariance matrix is adapted online based on previous observations. Evolutionary search has been recently proposed as a competing, albeit rather computational-intensive, method for neural architecture search [Real et al., 2017, 2019].

Swarm optimization (SO) methods take, instead, inspiration from the collective behaviour of species that organize themselves in decentralized flocks or colonies (e.g. birds or ants). These methods typically fall in the online HPO category. SO methods maintain a constant population size that is updated with local perturbations toward the best performing individuals. Notably, some of these methods [Jaderberg et al., 2017] involve also changing the parameters of the underlying statistical models, alongside the hyperparameters, effectively giving raise to different learning algorithms.

Both ES and SO methods require a considerable amount of customization, task-specific heuristics and design choices (e.g. to define meaningful mutation or perturbation strategies), undermining their accessibility and applicability to novel algorithms and problems. Besides, they typically require a considerable computational power, benefiting usually from large population sizes.

Bandit-based HPO algorithms, instead, cast hyperparameter optimization as an instance of the best-arm identification problem, typical of the *multi-armed bandit* reinforcement learning setting [Lattimore and Szepesvári, 2020]. The algorithm proposed by Jamieson and Talwalkar [2016], called *successive halving*, conceptually maps hyperparameter configurations to "arms" and assumes that at each iteration k one can observe an "intermediate cost" given by $o_k(\lambda)$ and that the true cost associated to λ is given by[16] $\lim_{k\to\infty} o_k(\lambda) = f(\lambda)$. If the learning algorithm is iterative (e.g. fitting a neural network with SGD) and the objective is a validation error of the type (3.5), then o_k may be naturally given by the validation error of the model after $r_k \in \mathbb{N}$ iterations (with $r_k > r_{k'}$ for $k > k'$). Then, given a budget B and an initial number of configurations n_0, successive halving runs n_0 instances of $\mathcal{A}$ for $r_0 = B$ iterations, observes o_1, discards the worst performing half and continues running the remaining

[16] But, generally, $\mathbb{E}[o_k(\lambda)] \neq f(\lambda)$. Indeed, in the standard case where o_k and f are validation errors, typically $\mathbb{E}[o_k(\lambda)] > f(\lambda)$, especially during the first iterations.

instances doubling the iteration budget[17] $r_1 = 2r_0 = 2B$, and so on, until there is only one configuration left. Thus the mapping π is given by

$$\left\{\lambda_i^{k+1}\right\}_{i=1}^{n_0/2^{k+1}} = \pi\left(\left\{\lambda_j^k, o_k(\lambda_j^k)\right\}_{j=1}^{n_0/2^k}, \upsilon\right) = \left\{\lambda \in \left\{\lambda_j^k\right\}_{j=1}^{n_0/2^k} \ : \ o_k(\lambda) < o_k(\lambda_{\zeta_k})\right\},$$

where ζ_k denotes the index of the $n_0/2^{k+1}+1$ worst-performing configuration after $r_k = 2^k B$ steps (assuming, for simplicity, that n_0 is a power of 2). Hence, for successive halving $n_k = n_0/2^k$.

When the first n_0 configurations are randomly sampled, successive halving essentially boils down to random search with a simple and effective, globally shared, early stopping procedure, that allows for an increased number of trial points for the same budget. When fixing a total budget, successive halving, however, may be rather sensible to the ratio between n_0 and B, as it can possibly eliminate too early well-performing configurations that require longer computation, or, conversely, allocate too many resources to comparatively few hyperparamter settings, underexploring the search space. Li et al. [2017a] propose an hedging strategy to alleviate this issue.

3.5 Gradient-Based Hyperparameter Optimization

Gradient-based (GB) methods are sequential techniques that seek to tackle the hyperparameter optimization problem with classic procedure for continuous optimization, described in Section 2.4. The update (3.12) generally takes the simple form of

$$\lambda_{k+1} = \pi(\lambda_k, \nu) = \lambda_k - \beta g_k(\lambda_k) \tag{3.27}$$

where $g_k \in \mathbb{R}^m$ is a vector that ideally is "close to the gradient" of f and $\beta \in \nu$ is a step size. The main challenge for gradient-based methods is associated with the computation of g_k itself, often called *hypergradient*. While only a handful of learning algorithms (e.g. ridge regression whit a global L^2 regularization parameter) are differentiable in the most classical sense – that is, $\mathcal{A}(D_{\text{tr}}, \cdot) \in \mathcal{C}^1(\Lambda)$ with a computable

[17] Actually, the original algorithm in [Jamieson and Talwalkar, 2016] is formulated in term of a total budget, and intermediate quantities such as r_k are computed so that the total computation of the algorithm does not exceed the total budget.

closed form expression for $\mathrm{D}_\lambda \mathcal{A}(D_{\text{tr}}, \lambda)$ – it is possible, in several cases, to calculate a numerical approximation of the exact gradient for many real-valued hyperparameters.

The two main strategies for computing the hypergradient are iterative differentiation [Domke, 2012, Maclaurin et al., 2015a, Franceschi et al., 2017, 2018a, Sections 5.4.1 and 5.4.2] and implicit differentiation [Larsen et al., 1996, Chapelle et al., 2002, Seeger, 2007, Keerthi et al., 2007, Foo et al., 2008, Pedregosa, 2016, Lorraine et al., 2020, Section 5.4.3]. The first involves computing the exact gradient – up to numerical errors – of an approximate objective, defined thorough the recursive application of an optimization dynamics that "replaces and approximates" the learning algorithm $\mathcal{A}$; the second is based on the (numerical) application of the implicit function theorem [Lang, 2012] to the solution mapping $\mathcal{A}(D_{\text{tr}}, \cdot)$ when this is expressible via an appropriate equation. We refer the reader to Chapter 5 for a detailed exposition of the methods and resulting algorithms and to Chapter 6 for discussion of approximation properties and convergence analysis. Recent implementations of both these approaches rely heavily on an efficient use of algorithmic differentiation tools, which we describe in Chapter A. Instead, we discuss below some advantages and drawbacks of GB-HPO, comparing it with the main methods described in Section 3.4, and conclude with a thorough literature review on the topic.

Gradient-based methods require full access to the underlying learning algorithm, have a non-trivial implementation overhead and have more stringent applicability constraints than most of the methods discussed above: among the fundamental assumptions, f should be smooth and the hyperparameters being optimized should be real-valued. The first requirement is usually met by replacing discrete-valued objective functions such as the accuracy for classification problems with differentiable surrogates such as the cross-entropy error (2.9), in line with general practices for formulating smooth learning problems. The second requirement may instead represent a more limiting factor. Even though there exist several continuous [Sperduti and Starita, 1993, Tibshirani, 1996, Franceschi et al., 2017, Liu et al., 2019] or probabilistic [Franceschi et al., 2019] relaxation of discrete hyperparameters, these involve remodelling the underlying learning algorithms and have seen to date a limited usage. Thus, gradient-based HPO methods are mostly applied to the optimization of continuous regularization and optimization hyperparameters, although there is a growing effort of adapting the technique to neural architecture search [Liu et al., 2019, Luo et al., 2018]. Another potential drawback of GB-HPO methods is that they are inherently sequential procedures, though parallel computation may be exploited for speeding up the evaluation of the hypergradient or for computing the hypergradient of cross-validation errors of the type (3.7).

On the flip side, gradient-based hyperparameter optimization techniques, relying on the local properties of the response surface, are sample efficient and allow us to optimize several order of magnitudes more hyperparameters than most of the methods described above. This is made possible by the usage of algorithmic differentiation tools that allow us to keep the computational footprint relatively low (see Section A.4 for an overview of the computational complexity of algorithm differentiation procedures). The well known properties of gradient descent that assure monotonic decrease in f and rates of convergence[18] independent from the dimensionality of λ, even if they do not hold (or are hard to verify) in practice, further motivate the appropriateness of GB-HPO for tackling high dimensional HPO problem. Experimental evidence with learning settings containing thousand or even millions hyperparameters [Maclaurin et al., 2015a, Pedregosa, 2016, Franceschi et al., 2017, 2019, Lorraine et al., 2020] has shown that GB-HPO techniques make fast progresses within few function (and hypergradient) evaluations in various problems of practical interest, despite the local nature of these search techniques. As we shall see in the next chapters of the thesis, this feature allows us to formulate and successfully tackle many learning problems under the shared perspective of hyperparamter optimization, promoting the usage of increasingly general-purpose learning algorithms (with more hyperparameters) in contrast to more case-and-context-specific techniques (with less hyperparameters).

As discussed in a recent work by Grazzi et al. [2020], the choice of the hypergradient computation procedure should be mainly guided by the problem's properties (and by any eventual computational constraint). Once this choice has been made, the

[18]For strongly convex objectives, see Section 2.4.1.

configuration parameters of gradient-based techniques are essentially limited to those of the associated descent procedure (3.27) (or accelerated and stochastic variants – see Section 2.4). This means that, if we do not consider the initial point λ_0 to be part of ν, the gradient-based approach to HPO, when applicable, makes comparatively less assumptions and may require less "tuning effort" than other techniques described above.

Finally, as noted in Section 3.3, gradient-based techniques are well-suited for finding hyperparamter schedules and perform online HPO. In this setting, a popular route to obtain schedules is by updating the hyperparameters as

$$\lambda_{t+1} = \lambda_t - \beta \tilde{g}_t$$

where $\tilde{g}_t \in \mathbb{R}^m$ is a vector that approximates in some way a gradient of the objective function f (e.g. a validation error) evaluated after a *partial execution* of $\mathcal{A}$. This scheme may be traced back to the eighties, precisely to the "delta-delta" rule for adapting learning rates, introduced by [Jacobs, 1988]. The rule, based on previous work by Barto and Sutton [1981] and recently revived by Baydin et al. [2018a], involves setting

$$\tilde{g}_t = -\nabla L(w_t)^\intercal \nabla L(w_{t-1}) \tag{3.28}$$

where L is the training error of a parametric model (see (2.3)). In the specific case where the HPO and training objectives coincide, (3.28) may be interpreted as approximating the hypergradient of the learning rate by only considering one step of the (weight) optimization dynamics. A very similar approach is pursued by Luketina et al. [2016] for scheduling regularization hyperparameters. We refer to the reader to Chapter 8 for further discussion on this topic, particularly regarding the case study of tuning learning rates schedules.

Beside [Jacobs, 1988], among the few early works on gradient-based hyperparameter optimization, we mention an article by Larsen et al. [1996] which followed an implicit differentiation scheme to optimize regularization parameters for training neural networks, shortly followed by Andersen et al. [1997] that adapted the preceding scheme to an online warm-restart setting. Bengio [2000] proposed gradient-based optimization of regularization hyperparameters, mostly concerning the simple case of learning algorithms that minimize quadratic objectives, where one may use the analytical expression for $\mathcal{A}(D_{\text{tr}}, \cdot)$. Later on, Chapelle et al. [2002], Keerthi et al. [2007] and Seeger [2007] explored the optimization of SVM hyperparameters by implicit differentiation. In the first two articles, the authors also experiment with different objective functions other than the (cross-)validation error. Foo et al. [2008] extended the studies to conditional log-linear models. It is worth noting that these works deal with learning algorithms that internally solve strongly convex problems.

During the last decade, due to the increasing popularity of neural network models, whose learning algorithms comprise the minimization of non-quadratic and non-convex objectives, attention shifted to iterative differentiation methods. The work on the "back-optimization" method proposed by Domke [2012] in 2012 may be considered to be among the first of this kind, although in [Domke, 2012] the method was applied in a different context[19]. Maclaurin et al. [2015a] discussed reverse-mode iterative differentiation for the case that the learning algorithm is based on stochastic gradient descent with momentum. In this setting, they introduced a fixed-point numeric data-type to lower the memory footprint of the method. Franceschi et al. [2017] generalized the previous work beside reversible learning dynamics and proposed forward-mode iterative differentiation for the optimization of a few key hyperparameters, alongside an online variant (later refined in [Donini et al., 2020]). More recently, Shaban et al. [2019] investigated the effect of truncating the reverse-mode procedure to save computation, and Beatson and Adams [2019] proposed a stochastic hypergradient estimator based on randomized telescopic sums to address the issue of the bias introduced by using fixed approximations of $\mathcal{A}$ (namely, a fixed number of steps of the underlying optimization dynamics).

On the other hand, interest in implicit differentiation methods has been recently revived, especially in the form of the "fixed-point" method. There is some empirical evidence [Liao et al., 2018, Lorraine et al., 2020] that (small variants) of these types

[19]Specifically, on continuous energy-based models for image labelling and denoising. J. Domke suggested that a similar strategy could be suitable to tackle HPO problems in a more general context.

of hypergradient computation schemes may achieve satisfactory results despite the non-convexity of the learning algorithms' optimization problems. However, as we shall see in Section 6.3.4, they may also show quite unstable behaviours in some other cases.

A few recent GB methods depart from the categories of iterative or implicit differentiation. Mehra and Hamm [2019] present an optimization approach based on penalty functions that blend the learning algorithm with the HPO problem. The works on "self-tuning networks" [Lorraine and Duvenaud, 2018, MacKay et al., 2019] (already introduced in Section 3.4.4), may also be regarded as GB-HPO techniques. There, online hyperparameter updates are computed utilizing a surrogate model (also fitted online) that should locally approximate the response surface. The proposed schemes, however, require remodelling heavily the underlying learning algorithm and thus may be challenging to incorporate in scenarios different from those presented in the original works. For instance in [MacKay et al., 2019] the local surrogate model is implemented as an hypernetwork [Ha et al., 2017], which needs to be tailored to the task at hand. A similar approach is also explored by Brock et al. [2018] in the context of neural architecture search, where an hypernetwork is used to map architectures to weights, in such a way that the architecture may be quickly evaluated on the target task. There, the hypernetwork substitutes entirely the learning algorithm. While it is unlikely that such a model would be capable of learning close approximations of the original mapping $\mathcal{A}$, the authors empirically show that the validation error computed on the outputted parameters correlates well with the "correct one".

3.6 Generalization in Hyperparameter Optimization

As a hypothesis may overfit a given training set, there might be the possibility that a hyperparameter configuration overfits a validation set. This happens when the validation error (3.8) (or another quantity of interest, such as the leave-one-out cross-validation error) departs from (or does not correlate well with) the generalization error (2.1). Traditionally, grid search approaches only evaluate a fixed and comparatively small set of configurations, thereby limiting exploration. Yet, the issue of overfitting in hyperparameter optimization becomes more pressing when more advanced techniques which “actively” seek to minimize (3.8) are employed. There may be various sources of overfitting in the HPO context: a too small or not representative validation set, a too “thorough” optimization or a too large “complexity” of the hyperparameter space. In this last regard, recent advances in gradient-based hyperparameter optimization methods [Maclaurin et al., 2015b, Franceschi et al., 2017, Lorraine et al., 2020] now allow us to effectively optimize orders of magnitude more hyperparameters that what was previously possible. Unfortunately, whereas generalization and overfitting have been widely studied in the context of learning algorithms, hypothesis spaces and function complexity [Vapnik, 2013, Friedman et al., 2001], the same cannot be said in the context of hyperparameter optimization. Among the few work about the issue, Ng [1997] discusses the case where the validation set may be noisy proposing an ulterior validation step, while Ndiaye et al. [2019] more recently focus on grid search.

There are several difficulties in tackling this topic under a theoretical point of view. The nested nature of the HPO problem and the variety and diversity of the underlying learning algorithms are among these. Furthermore, overfitting in HPO is not necessarily tied to the complexity of the resulting hypothesis. The question largely depends on which hyperparameters one wishes to optimize. For instance, many optimization hyperparameters do no immediately relate to the complexity of h. One may then take an empirical approach to the issue, for instance, by further divining the validation set, and holding out one of the two splits to perform early stopping on the HPO procedure. We implement this strategy in Chapter 9 when learning edges of relational graphs. Generalization and overfitting in hyperparameter optimization is, however, considered to be a (difficult) open question for future research [Feurer and Hutter, 2018].

3.7 Interim Summary

This chapter reviews hyperparameter optimization in machine learning, offering a formal statement of the general problem and an overview of the principal HPO techniques. Machine learning algorithms use data to induce hypotheses that explain phenomena or concepts of interest, yet their behaviour is controlled by the setting of various configuration parameters. We saw that hyperparameters define the hypothesis space, both explicitly and implicitly and determine how the search will take place: a careful choice of their value is paramount for most learning procedures. We provided a series of examples in Section 3.2, dividing hyperparameters into three categories, explaining their role and impact on typical learning algorithms, as well as their possible encodings. The considerable diversity of both the learning algorithms and their respective configuration parameters, in conjunction with (potential) computational complexity issues, present a critical challenge for the resolution of HPO problems, which are nevertheless of crucial importance. Indeed, as we suggested in Section 3.1, the HPO problem (3.6) goes beyond the minimization of the empirical risk performed by many learning algorithms, and requires searching for configurations that lead to hypotheses that generalize well. This is still mostly carried out through an empirical, data-driven, approach (i.e. using a validation set) which requires very few assumptions. We follow this path in the thesis.

In Section 3.4 we studied and analyzed the major approaches to HPO, commenting on advantages and drawbacks of each family of methods, also in view of the particular aims and features of the field. One of the factor that emerged from the review of Section 3.4 is a general difficulty for most of the techniques to scale to high dimensional problems (many hyperparameters). This issue, that becomes even more relevant in a meta-learning context where the aim is to infer, rather than simply tune, a learning algorithm, is largely addressed by gradient-based approachs, albeit introducing some other restrictions. We offered a literature review of gradient-based HPO in Section 3.5, which can serve as context for the study we will present in the later chapters. Finally, we mentioned an important open question in HPO, that regards the generalization priorities of configurations found by hyperparameter optimization techniques.

Chapter 4

Review of Meta-Learning

While hyperparameter optimization might be viewed – if one accepts to leave generalization issues aside – as an exquisitely mathematical problem of minimizing a complex and expensive-to-evaluate function in a possibly intricate search space, the nature of meta-learning, or *learning to learn*, is certainly more tied to the fundamental questions of artificial intelligence and machine learning. Meta-learning algorithms add an additional layer of abstraction over the formulations of standard learning. They incorporate components that allow modifying the way learning itself takes place, based on the experience from a multitude of tasks. Learning to learn promotes a holistic approach to the problem of learning, reasoning about methods to extract, retain and improve a shared knowledge, shifting and adapting the inductive bias in a data-driven fashion to improve learning efficiency and efficacy. Following the three paradigms of machine learning, meta-learning also branches out into supervised, unsupervised and reinforced depending on the nature of the underlying tasks. We will focus almost exclusively on the the first type. See [Hospedales et al., 2020] and references therein for a very recent overview of meta-learning that discusses also meta-reinforcement and meta-unsupervised learning settings.

We start with a potential application example that helps illustrate some of the key concepts behind meta-learning[1]. We then proceed by defining and formalizing the

[1] The example tries to be "close enough" to some contemporary benchmark applications especially widespread in the popular sub-field of few-shot learning [Vinyals et al., 2016, Ravi and Larochelle, 2017, Finn et al., 2017], but also seeks to highlight some other (less frequently mentioned) peculiar aspects of the meta-learning paradigm, in an attempt to present a broader account of the subject.

meta-learning problem, touching upon its historical background and interpretations in Section 4.2. Contextually, we discuss similarities and differences with other related fields in Section 4.3 and finally conclude the review with a section dedicated to the description and analysis of a number of current techniques and strategies for meta-learning.

4.1 A Domestic Robot Example

Imagine that, in the near future, a company wants to produce domestic humanoid robots capable of helping their owners with daily activities [Parmiggiani et al., 2017]. An essential component of such a device would be a module for recognizing people from camera inputs[2]. A wealth of experimental evidence has shown that deep convolutional neural networks are very effective at tackling these types of discriminatory tasks, systematically achieving state-of-the-art results on benchmark problems since the early 2010s. Hence, the company may wish to implement such a recognition module capitalizing in some way on this kind of models. However, standard (single-task) supervised learning algorithms for training CNNs have the notorious drawback of being data and computation hungry. Yet, computational resources and the collection of labelled examples (as well as mistakes!) come at a premium when the robot is already in use, e.g. in a domestic environment. For this reason, it would be desirable to carry out as much work as possible at production time before the robot is shipped to the customers.

The members of the buyer's household would certainly be among the first individuals that the robot should be able to recognize, hopefully with great accuracy. However, at production time, it is unthinkable that the company has access to images of all its potential customers and their families. Besides, even if it did, it would be quite wasteful to implement a large model able to recognize thousands of individuals when most likely a few dozen would suffice. After all, a domestic robot is not supposed to wander the earth hailing at people, but should rather mostly remain within the

[2] In this example, for simplicity, we consider that a task corresponds to the binary classification problem of deciding whether a particular person is present in an image or not (hence there is a task per each person). Other tasks and problems in this context would comprise also (at least) localization, segmentation and pose estimation.

house confines. It is also hard to imagine that the company would ask to its potential customers to send over their family photo albums, at the very least for possible privacy concerns. Furthermore, it would be hardly practicable (or quite limiting) to foresee all the possible encounters the robot may have during its activity.

In short, we can safely assume that the robot cannot be trained on the tasks of interest – learning to recognize household members and potential guests – at production time. Hence, the recognition module must include a learning component that allows the robot to process the data (camera images) collected during the exercise of its activity, producing or adapting on demand models for recognizing newly encountered people. In principle, this could be achieved by including a standard supervised learning algorithm for training CNNs; perhaps improved with multi-task elements that leverage on the evident similarities between the various tasks and hyperparamter optimization routines that automate and improve the process. Such an implementation choice would, however, defer all the learning workload to the robot's activity time, which is not ideal for the reasons mentioned above.

Thus, if on the one side executing all the learning at production seems unfeasible, on the other side letting the robot learn only during its activity could prove quite problematic. Meta-learning provides a compelling middle ground between these two solutions. The basic observation is that at production time, while the company may not detain examples of the people (tasks) of interest, it can easily access to examples of different individuals (other tasks). This related data may be used to extract some knowledge about the concept of "recognizing people" in general, regardless of their specific attributes, inferring a *specialized learning algorithm* that is adapted to the relevant class of tasks. In this way most of the heavy lifting may be carried out in a first stage of learning (the *meta-level*), after which one may obtain a "lightweight learning routine" (the *base-level*) that can run locally as the need arises. The meta-level could comprise the learning of the first layers of a CNN "shared" among all the tasks, while the base-level lightweight and task-specific learning algorithm may be implemented as the adaptation of a few top layers, or even simply by fitting a logistic regression model on top of the features extracted by the CNN.

The resulting learning component is only supposed to work on a restricted set of tasks: it will not be called to learn to distinguish dog breeds or solve partial differential equations. This corresponds to the assumption that the relevant tasks are organized according to an unknown distribution, called *meta-distribution*, that describes relevant "operational boundaries" in which the learned algorithm is supposed to perform well. The specification of a meta-learning problem needs not be limited to the (abstract) concepts that lie behind the family of tasks of interest – that is recognizing people, in our case. In fact, machine learning problems are also characterized by the type of experience that may be collected and elaborated as well as the performance measure that guides the learning process. As meta-learning reasons exactly on the top of this level of abstraction, these are also two other potential factors of variation for meta-learning problems.

Consider, for instance, the difference between learning to recognize a member of the household and learning to recognize a guest invited one evening for dinner. In the first case, we can imagine that the robot will be able to collect a sizable number of samples (camera images) in a comparatively limited amount of time, with different light conditions, clothing and poses[3]. The robot will simply spend more time with an household member, beside the fact that he or she might willingly "pose" and provide accurate feedback for the new purchase. The same, however, cannot be expected from a guest invited for dinner. In this second case, the learning algorithm will likely have to rely on much fewer data – and, most importantly, much less diversified data (e.g. only indoor, evening images). To account for this, the base-level algorithm could adapt a number of layers proportional to the quantity and quality of the available data for each individual/task.

Furthermore, it may be acceptable that the robot will not be able to positively recognize the same guest if he or she comes back for a second time (say, for lunch, with different cloths and maybe even with a different hairstyle). After all, also for us, it may take a little while before being able to identify someone with the utmost confidence. On the other hand, we certainly would not appreciate if the robot starts mistaking

[3]Thus meta-learning is by no means restricted to a few-shot learning regime

other people (or, worse, objects or animals) for the guest only met a single time. The owners, instead, could be more patient to (few, initial) mistakes, but may expect to be positively recognized in every conceivable situation. These dissimilar expectations may be reflected in different performance measures for these two "subclasses" of tasks. For instance one may favor precision over recall in the guest case, while accuracy might be an adequate metric for household members.

4.2 Problem Setting

Summing up, the company of the above example, by running a *meta-learning algorithm* on a set of curated datasets (or *meta-training set*), may obtain a base-level learning algorithm to include in the recognition module that has the potential to be much more efficient, stable and accurate *on the tasks of interest* than general-purpose counterparts, because it incorporates a tailored, data-driven inductive bias extracted at the meta-level. Intuitively, the better the meta-training set reflects all the possible peculiarities and variations of the family of tasks of interest, the more effective the meta-learning procedure and, hence, the resulting learning algorithm will be. The example introduces some basic "ingredients" of the meta-learning setting: the presence of several tasks, the idea of acquiring in some way previous knowledge, the division of the learning process into two stages (the meta-level and the base-level), the unavailability of data pertaining some tasks of interests, the goal of generalizing at an algorithmic level, and so on. These concepts are present to some extent in most of the meta-learning literature, especially in the more recent works. However, over the years, various authors have utilized the term in different and not always compatible contexts, possibly hampering the emergence of a general, shared, definition of the meta-learning problem setting. To date, there is still an ongoing debate, as various currents of thoughts would either see meta-learning recognized as the "doorway" to artificial general intelligence [Schmidhuber, 2007], or have it demoted to a particular variation of supervised learning [Chao et al., 2020], with a whole spectrum of intermediate positions in between.

A Brief Historical Note. J. Schmidhuber is generally accredited as the first author to introduce, in the context of machine learning, the term "meta-learning" in the late eighties. In his Diploma thesis, Schmidhuber informally describes a meta-learning algorithm as a "system [that has] the ability to learn the methods how to learn" [Schmidhuber, 1987] (not necessarily stopping at the first meta-level). He takes an evolutionary approach at the problem, proposing meta-evolution and "self-referential associating learning mechanism" as possible implementations. In [Schmidhuber, 1987], the bulk of the effort is devoted to developing (symbolic) languages, initial states and environments that could exert an evolutionary pressure for supporting this kind of behaviour. Later, among other efforts, Schmidhuber continues the work on self-referential learning, proposing the so-called "Gödel machines" [Schmidhuber, 2007], ideal theoretical problem solvers which are capable of making "provably optimal self-improvements". The generality of Schmidhuber's view seems to suggest a strong tie between meta-learning and artificial general intelligence.

At least other three works [Hinton and Plaut, 1987, Bengio et al., 1991, Wolpert, 1992] published during the late eighties and the early nineties are deemed, retrospectively, as significant early contributions to the field of meta-learning [Vilalta and Drissi, 2002, Hospedales et al., 2020], although the term "meta-learning" does not appear in any of them. Hinton and Plaut [1987] describe a neural network where each connection parameter is given by the sum of two weights: a "slow" weight – same as the traditional connection parameter – is responsible for storing long-term knowledge and a "fast" weight, continuously decayed toward 0, capable of quicker adaptation to recent signals thanks to a larger learning rate. Hinton and Plaut's article inspired later research, e.g. on "meta-networks" [Munkhdalai and Yu, 2017] and, similarly to Schmidhuber's works, concerned a learning scenario that is temporal (or sequential) in nature. The agent would live throughout a single, lifelong, cycle.

Remaining in the field of neural networks, Bengio et al. [1991] investigated the possibility of inferring parametric "synaptic learning rules" (i.e. weight updates) using either gradient descent or evolution. While the main aim of the original work was that of introducing biologically plausible learning mechanisms for neural networks (in contrast to backpropagation), probably other aspects influenced the later literature, especially in the context of learning to optimize. Specifically, to promote generalization, the authors proposed weight sharing of the update rule's parameters, which should have been (meta-)learned from the simultaneous execution of several different representative tasks.

Under a more general perspective, "stacked generalization" [Wolpert, 1992] is also considered to be an early approach to meta-learning [Vilalta and Drissi, 2002]. Stacked generalization involves using a set of learning algorithms to train two levels[4] of models (generalizers), whereby the inputs to the "level-1" models are given by the outputs of the generalizers at the "level-0". These "levels" should not be considered akin to the meta and the base levels mentioned in the opening example. Indeed, Wolpert's original work is focused on single task learning and was presented as an extension to the standard cross-validation procedure (oriented to algorithm selection; see Section 3.1). However, interestingly, the proposed scheme bears a certain resemblance to the modern approaches to meta-learning (cf. (4.2)) in that it uses the validation (test/left-out) splits as an integral part of the training process.

Throughout the nineties, research in meta-learning and related fields such as multi-task and transfer learning intensifies. S. Thrun and L. Pratt take stock of a large portion of these works in an influential curated collection of articles [Thrun and Pratt, 1998b] that discuss meta-learning – or learning to learn, as they name the field[5] – under a methodological, theoretical and practical standpoint, touching upon all the paradigms of machine learning. Overall, the book emphasises the importance of learning representations (or metrics) through multiple tasks (including learning tasks relatedness measures) and promotes a clearer distinction between the meta-level and a the base-level of learning. Meta-learning is presented as strongly tied to the search for base-level inductive biases that should happen at the meta-level. Their delineation of the meta-learning scenario appears as a modification of Mitchel's definition of learning algorithm (quoted in our thesis at the beginning of Section 2.1), which, summarized, reads as follows:

[4] The procedure may be extended to multiple levels.

[5] We use the terms meta-learning and learning to learn interchangeably. It is likely that Thrun and Pratt introduced the idiom "learning to learn" also to distance the content of their collection from other interpretations of the field.

Given a family of tasks, with their respective experiences and performance measures, an algorithms is said to learn-to-learn if its performance at each task improves with experience and with the number of tasks. [Thrun and Pratt, 1998b]

While certainly posing the ground for a more systematic approach to the subject, we note that this definition could look quite related to a multi-task learning perspective, suggesting that the yardstick for a meta-learning algorithm is its ability to improve upon the tasks which it has already seen, rather that those (potentially) yet to see. We refer to Section 4.3.2 for further discussion on the relationship between these two fields.

A different perspective on the subject is embraced by some other authors [e.g Vilalta and Drissi, 2002, Smith-Miles, 2009, Vanschoren, 2019, and references therein] who view meta-learning as strongly tied to algorithm selection and linked to meta-heuristics [Smith-Miles, 2009]. A typical instance is given by a learning scenario where the experience is collected through several executions of standard learning algorithms on (single-task) learning problems, possibly recording so-called "meta-data". The meta-learning component is responsible for predicting (or ranking) the effectiveness of each learning algorithms on query tasks. Of central importance are the so-called "meta-features", which are hand-engineered [see e.g. Table 1 in Vanschoren, 2019] or learned statistics/features that should concisely describe each learning task or empirical dataset. Research branches that follow this spirit, strongly related to hyperparameter optimization and algorithm selection, include landmarking [Pfahringer et al., 2000], dynamic bias selection [Gordon and Desjardins, 1995] and loss curves prediction [Leite and Brazdil, 2005]. Section 4.4.2.1 further describes the general approach to meta-learning stemming from this perspective.

In the recent years *few-shot learning* has gained considerable attention, so much that, lately, the term "meta-learning" has been often associated to it, if not confused with it [see e.g. Sec. 2.5 of Chen and Liu, 2018]. The aim of few-shot learning is to generalize from very few examples by leveraging the presence of a vast multitude of related tasks. Possibly, the best known few-shot learning problem instances pertain supervised multiclass classification with datasets that contain from one (*one-shot learning*) to five or ten examples per class [Vinyals et al., 2016, Ravi and Larochelle, 2017]. Among the works that probably sparked a rising interest in this particular learning setting we recognize an article by Santoro et al. [2016] on memory augmented neural networks. Santoro et al. were among the first to clearly link few-shot learning to meta-learning, developing neural network based methods to tackle the problem in a substantially "general purpose" fashion. Prior work typically approached few-shot learning with non-parametric techniques adapted to the specific problem setting of interest [Fei-Fei et al., 2006]. The core idea was still that of sharing knowledge from available classes (or categories), but these were not necessarily interpreted as tasks. The 2016 article [Santoro et al., 2016] has been closely followed by a rather long list of related studies that quickly achieved increasing performance gains on selected benchmark problems. We will introduce and discuss the concepts behind some of these studies in Section 4.4.

The Meta-learning Problem. In this thesis, we mostly adopt Thrun and Pratt's view, with a few key differences and remarks aimed at highlighting the importance of the generalization at the meta-level. To do so, we need to formalize a few concepts, such as the meta-distribution and the generalization error of learning algorithms. Indeed, whereas in standard supervised learning one reasons about the quality of learned hypotheses, in meta-learning one reasons about the quality of learned algorithms, as the focus shifts to a higher level of abstraction. Hence, if a good learning algorithm produces hypotheses that generalize well[6], an effective meta-learning algorithm generates learning algorithms that, in turn, output hypotheses that generalize well. That is, for short, learning algorithms that generalize well. Yet, it would be unrealistic to expect that a learning algorithm, even if learned, would be capable of performing well on completely arbitrary tasks. Instead, we shall rather assume that the tasks of interest are organized according to a given (but unknown) *meta-distribution*. For instance, the meta-distribution of the introductory example informally describes a class of binary classification problems pertaining the recognition of specific individuals from images

[6] That is hypotheses that achieve low generalization errors, according to (2.1).

captured by a camera installed on a domestic robot.

Let D and τ denote data[7] and task, viewed as random variables that take values over a space of data (or experiences/observations) $\mathcal{D}$ and a space of tasks $\mathcal{T}$, respectively. In supervised meta-learning, $\mathcal{T}$ comprises supervised learning tasks, viewed (according to Section 2.1) as join probability distributions $p_{x,y}(\cdot|\tau)$ over input and output spaces $\mathcal{X}_\tau \times \mathcal{Y}_\tau$, endowed with appropriate loss functions ℓ_τ. In the most standard case, which we follow here, $\mathcal{D}$ is a space of finite dimensional datasets (cf. (3.3)), that is

$$\mathcal{D} = \bigcup_{\bar{\tau} \in \mathcal{T}} \bigcup_{N \in \mathbb{N}} (\mathcal{X}_{\bar{\tau}} \times \mathcal{Y}_{\bar{\tau}})^N .$$

Then, we define a meta-distribution $p_{D,\tau}$ as a joint probability distribution over $\mathcal{D} \times \mathcal{T}$. Drawing from $\bar{D}, \bar{\tau} \sim p_{D,\tau}$ means sampling a dataset $\bar{D} \in \mathcal{D}$ and a task $\bar{\tau}$ with associated input and output spaces $\mathcal{X}_{\bar{\tau}}$ and $\mathcal{Y}_{\bar{\tau}}$, distribution $p_{x,y}(\cdot|\bar{\tau})$ over $\mathcal{X}_{\bar{\tau}} \times \mathcal{Y}_{\bar{\tau}}$ and loss function $\ell_{\bar{\tau}} : \mathcal{Y}_{\bar{\tau}} \times \mathcal{Y}_{\bar{\tau}} \to \mathbb{R}^+$. To simplify the discussion and the notation, we shall assume in the following that the loss functions are the same for all the tasks and identify a task with its associated probability distribution (leaving input and output spaces implicitly defined through $p_{x,y}(\cdot|\bar{\tau})$). Meta-distributions of typical supervised meta-learning problems are non-zero only on outcomes where the points of $\bar{D}$ are themselves drawn from $\bar{\tau}$ (not necessarily independently). We indicate this property by saying that the conditional distribution $p_D(\cdot|\bar{\tau}) = 0$ unless $D \sim \bar{\tau}$.

In our example of Section 4.1, D represents the random variable associated to the supervised data the robot may collect about an individual: a sample of D is a series of camera images with the respective targets that say whether a specific person is present or not. Instead, τ represents the random variable associated to the tasks of recognizing a specific individual: a sample of τ is a joint probability distribution over camera images and binary labels indicating the person's presence. Then $p_{D,\tau}$ concerns the distribution of pairs of datasets-tasks that the robot may encounter during its activity. For instance, the meta-distribution may concentrate around cases where the data is either abundant (when the person to recognize is a member of the household) or scarce

[7] In this section, differently from 2.1, we think of D as a random variable: a list of examples – input and output pairs, themselves thought of random variables – of any size.

(when the person is a guest). It will assign zero probability to outcomes where the dataset regards an individual, but the task is about recognizing someone else (unless the company wants to consider also cases where somebody tries to fool the robot!). It may also describe the fact that when there are few samples, it is also likely that they are quite homogeneous, as positive instances are likely to pertain to a single particular event or circumstance (e.g. a dinner).

If one wants to look for analogies with standard supervised learning and the concept of task distribution $p_{x,y}$ introduced in Section 2.1, our construction of $p_{D,\tau}$ may perhaps seem counterintuitive at first glance. The order of the arguments of $p_{D,\tau}$ might seem to point toward "the wrong direction", as in standard supervised learning one generally thinks of the situation where y depends on/is a function of x. Here, instead, D "depends" on τ. While one may certainly think of $p_{D,\tau}$ as either $p_{\tau,D}$ or $p_D(\cdot|\tau)p_\tau$ – the latter also suggesting a hierarchical relationship – we care to note that we have made this choice to suggest a different analogy, whereby the training set D takes place of the input x as the observation and τ replaces y as the target. Only, this time, the target τ is typically unknown (but often pretend we can observe a representative sample of it during meta-training).

With our (perhaps non-standard) definition of meta-distribution[8] the concept of generalization in meta-learning writes down quite naturally. Recall that a learning algorithm is defined as a mapping $\mathcal{A} : \mathcal{D} \times \Lambda \to \mathcal{H}$ (Equation (3.1)). When speaking of base-level learning algorithms in meta-learning, we gloss over the hyperparameter space and assume that $\Lambda = \emptyset$. This omission, carried out partially to simplify the discussion and partially to reflect common practices in meta-learning where very seldom hyperparameters of base-level learning algorithms "play any active role", leave us with $\mathcal{A} : \mathcal{D} \to \mathcal{H}$. We may then define the generalization error of an algorithm with respect to a meta-distribution as

$$\boldsymbol{\mathcal{E}}(\mathcal{A}, p_{D,\tau}) = \mathbb{E}_{(D,\tau)\sim p_{D,\tau}}[\mathcal{E}(\mathcal{A}(D), \tau)] \tag{4.1}$$

where $\mathcal{E}$, defined in (2.1), is the generalization error of an hypothesis with respect to a

[8] We defer a further discussion and motivation of our definition to the end of the section.

(standard) data distribution. We will use bold characters to indicate objects related to the meta-level. The formulation of the generalization error at the meta-level shows a compelling similitude to the (standard) generalization error at the base-level (2.1). In fact, as we proceed with the definition of "meta-level counterparts" of the familiar objects of standard learning, much of what said in Sections 2.1 and 3.1 may be repeated here with only minor adaptation to accommodate the meta-learning scenario.

The meta-distribution $p_{D,\mathcal{T}}$ is generally unknown. One may then derive an empirical version of (4.1) based only on finite (computable) quantities. To do so we introduce the concept of *meta-dataset*, that is a finite collection of $\mathbf{N} \in \mathbb{N}$ datasets[9], often called *episodes* in few-shot learning,

$$\mathbf{D} = \left\{\left(D^j_{\mathrm{tr}}, D^j_{\mathrm{ts}}\right)\right\}_{j=1}^{\mathbf{N}} \in \boldsymbol{\mathcal{D}} \tag{4.2}$$

split into training and testing partitions, where $D^j_{\mathrm{tr}} = \left\{\left(x^j_i, y^j_i\right)\right\}_{i=1}^{N^j_{\mathrm{tr}}}$ and, likewise, $D^j_{\mathrm{ts}} = \left\{\left(\hat{x}^j_i, \hat{y}^j_i\right)\right\}_{i=1}^{N^j_{\mathrm{ts}}}$. The space of finite dimensional meta-datasets $\boldsymbol{\mathcal{D}}$ is defined as

$$\boldsymbol{\mathcal{D}} = \bigcup_{\mathbf{N}\in\mathbb{N}} (\mathcal{D} \times \mathcal{D})^{\mathbf{N}}. \tag{4.3}$$

The training splits D^j_{tr} are sampled directly (but jointly with D^j_{ts}) from the first component of $p_{D,\mathcal{T}}$. The test splits D^j_{ts} (sometimes referred to as validation split [Franceschi et al., 2018a, Rusu et al., 2019]) contain a number of i.i.d. samples drawn from τ^j, with which one may estimate the generalization error $\mathcal{E}\left(\mathcal{A}\left(D^j_{\mathrm{tr}}\right), \tau^j\right)$ of $\mathcal{A}$ on the j–th task. In other words, if a sample from a meta-distribution is given by $\left(D^j_{\mathrm{tr}}, \tau^j\right)$, then an episode, element of $\mathbf{D}$, is a pair $\left(D^j_{\mathrm{tr}}, D^j_{\mathrm{ts}}\right)$ with $D^j_{\mathrm{ts}} \sim \tau^j$ (i.i.d.). Additionally, for the reasons explained in Section 3.1, one typically asks that $D^j_{\mathrm{tr}} \cap D^j_{\mathrm{ts}} = \emptyset$. The domestic robots company may construct a meta-dataset starting from images of various people that may be collected e.g. during trial sessions at production time, organizing them in episodes that try to mimic real activity scenarios.

[9] Recently, various authors [e.g. Rusu et al., 2019, Tian et al., 2020, Chen et al., 2020b] have stressed on the importance of pretraining in meta-learning, whereby meta-datasets are "flattened" and used as standard meta-learning datasets. We note that this procedure is rather natural only for some meta-learning problems where input and output spaces share a very similar structure.

Some authors [e.g. Vinyals et al., 2016, Snell et al., 2017, Munkhdalai and Yu, 2017, Sung et al., 2018], especially in the context of few-shot learning, call D_{tr} and D_{ts} *support* and *query* sets, respectively. Often [Vinyals et al., 2016, Sung et al., 2018] the use of this terminology is associated with metric learning strategies and instance-based (base-level) algorithms. We note that the practice of constructing datasets for meta-training by splitting the available datapoints into training (support) and testing (query) sets is comparatively recent and does not appear in early works. We trace this "change" back to the 2016 article of Vinyals et al. [2016] on one-shot learning. The authors advocate developing meta-learning procedures where "test and train conditions [...] match". Although "dataset splitting" could have been initially tied to the instance-based non-parametric nature of the original method (where it would hardly make sense to compute a loss on the support/training points for the purpose of meta-training), it has since then been widely accepted as one of the drivers of performance gains in many other approaches to meta-learning[10].

Using the empirical error of an hypothesis $\hat{\mathcal{E}}$, introduced in (2.2), we define by

$$\hat{\boldsymbol{\mathcal{E}}}(\mathcal{A},\mathbf{D}) = \frac{1}{\mathbf{N}}\sum_{j=1}^{\mathbf{N}} \hat{\mathcal{E}}\left(\mathcal{A}\left(D^j_{\text{tr}}\right), D^j_{\text{ts}}\right). \tag{4.4}$$

the empirical error of a learning algorithm on a meta-dataset $\mathbf{D}$. A meta-learning equivalent of the ERM problem of Section (2.1) then simply involves minimizing (4.4) in a given space of learning algorithms, or *meta-hypothesis space*. We shall denote such space, predictably, as $\boldsymbol{\mathcal{H}}$.

As it was the case in standard learning, very often it is convenient to parameterize $\boldsymbol{\mathcal{H}}$ to simplify the search. One option is to utilize (a part of[11]) the hyperparameter space Λ of an otherwise standard learning algorithm to achieve this. In fact, one of the

[10] See also the discussion at the end of this section. Related experimental results are reported in Section 7.2.3. However, from a statistical and learning theory viewpoint, up to our knowledge, it is not clear what are the advantage of this recent practice over the standard learning-to-learn setting [Thrun and Pratt, 1998b] in which all the sampled points in a task are used for training. Some preliminary results seem to suggest that the benefits may depend on the specific structure of the problem at hand [Bai et al., 2020], although more research in this direction is clearly needed.

[11] Remaining hyperparameters are then considered as part of configuration space $\boldsymbol{\Lambda}$ of a meta-learning algorithm. See Section 4.4.2 for examples.

leitmotif of the present work is to show that many meta-learning algorithms precisely act on (or search in) the hyperparamter space of a base $\mathcal{A}$, establishing a link between meta-learning and hyperparameter optimization. We will lay the foundation of this point in Section 4.3 and later elaborate in the second part of the thesis.

By mirroring the definition of learning algorithm given in Section 3.1, we then represent a meta-learning algorithm by an higher-order mapping

$$\boldsymbol{\mathcal{A}} : \boldsymbol{\mathcal{D}} \times \boldsymbol{\Lambda} \to \boldsymbol{\mathcal{H}}; \qquad \boldsymbol{\mathcal{A}}(\mathbf{D}, \boldsymbol{\lambda}) = \mathcal{A} \tag{4.5}$$

where $\boldsymbol{\Lambda}$ is a configuration space (meta-learning algorithms are not immune to hyperparameters!). The specific implementation of $\boldsymbol{\mathcal{H}}$ is tied to the particular meta-learning algorithm; we shall see some significant instantiations in Section 4.4. One that has already been suggested in the introductory example of Section 4.1, which we will further elaborate in Section 7.2.1, pertains learning algorithms that fit logistic regression models on the top of meta-learned features. Many meta-learning algorithms attempt to minimize in some way the error (4.4), additionally introducing various regularization terms and strategies to process the meta-training set.

A Definition of Meta-learning. We may finally integrate the definition of meta-learning algorithm given by Thrun and Pratt [1998b] by saying that

> Given a family of data and tasks with their respective performance measures, an algorithm is said to meta-learn if, whilst using the same data, its performance[12] improves at each task (both seen and yet unseen) through experience.

By acquiring more information about the meta-distribution of interest, a meta-learning algorithm improves its “knowledge” and makes its learning algorithm more efficient and precise (on tasks belonging to it). The base-level learning algorithm improves because more episodes – which effectively constitute “the experience” in this context – are processed at the meta-level; much as a standard learning algorithm may refine its hypothesis as it sees more datapoints (“the experience”, in the context of standard

[12] That is the performance of its associated base-level learning algorithm.

learning). Of course, this does not preclude the base-level algorithm from exhibiting better performance at a task if more data become suddenly available. After all, one should expect that a meta-learned learning algorithm still behaves like a standard learning algorithm. Yet, our definition is aimed at empathising that, in meta-learning, the primary dimension of learning is the meta-level. By contrast, if one executes a standard learning algorithm on ninety-nine tasks, and then processes the hundredth, she or he will record the same performance of running the algorithm only on this last task (random factors aside). There is no meta-level capable of extracting and retaining information across tasks: the standard learning algorithm alone has no way of improving by seeing more of them.

Before proceeding with the localization of meta-learning within the broader panorama of related areas of machine learning, we return upon the concept of meta-distribution and discuss and motivate our choices.

On the Concept and Definition of Meta-distributions. Our definition of meta-distribution $p_{D,\tau}$ as a joint distribution over datasets and tasks somewhat differs from standard references[13] (see e.g. [Baxter, 1998] or [Finn et al., 2017] for a more recent account) where the meta-distribution is only presented as an object that operates over tasks. With our notation, this would be a distribution p_τ solely defined over $\mathcal{T}$. We reckon that a direct consequence of this latter choice is that the event of sampling a training set D, input to $\mathcal{A}$ must be left somewhat implicit. One has either to push the definition of the necessary information into τ itself [Finn et al., 2017] or to offer elsewhere a separate description of the sampling procedure [Baxter, 1998, Maurer, 2005, Vinyals et al., 2016]. We believe that the first exposition practice might reveal troublesome as the concept of "task" becomes hardly separable from that of "dataset" – how to define generalization, then? The latter approach, instead, might suggest demoting the process of sampling training data to a mere technical detail.

In contrast, we argue that decoupling dataset and task distributions returns a more accurate and natural view of the meta-learning setting: if on the one side

[13] Our view has some similarities with a recent work of Chao et al. [2020]. However, we do not postulate the existence of an optimal model for each task, nor require the definition of a meta-loss, as these may induce some artificiality in the definition of the meta-learning problem.

algorithms must work with tangible finite-dimensional datasets, on the other side we are still looking for generalization – this time at the meta-level. The first and the second components of $p_{D,\mathcal{T}}$ aim at capturing exactly this dichotomy, making it explicit. Besides, the way one expects to receive training data (also, possibly, depending on the specific sampled task) is an integral part of the meta-learning problem; one that should mirror the realistic scenario in which the (meta-learned) learning algorithm is supposed to operate. For instance, our notation allows us to seamlessly specify the targeted data regime: as a concrete example, in one-shot learning for classification, where we expect to receive only one example per class, we can simply write that $|\bar{D}| = |\mathcal{Y}_{\bar{\tau}}|$ for $\bar{D}, \bar{\tau} \sim p_{D,\mathcal{T}}$. Furthermore, decoupling observations and tasks allows us to naturally consider far more general meta-learning settings in which $\bar{D}$ is not limited to be a (supervised) dataset; it could be, for instance, a string that describes $\bar{\tau}$ in natural language.

We further believe that our definition may give an additional methodological justification to the late practice of partitioning datasets into training (support) and testing (query) splits and utilizing the test splits to provide feedback at the meta-level, during meta-training. Indeed, the first split D_{tr} relates to the first random variable and simply reflects the kind of data the base-level learning algorithm $\mathcal{A}$ is supposed to receive: it is the input to $\mathcal{A}$. The second split D_{ts} relates to the second random variable and should comprise a number of i.i.d. samples drawn from $\bar{\tau} = p_{x,y}$. It allows us to empirically test the generalization capabilities of $\mathcal{A}$ by computing an unbiased estimate of the generalization error of $h = \mathcal{A}(D_{\text{tr}})$. Conversely, we do not explicitly require training samples to be i.i.d., although this is an ubiquitous assumption in the literature.

In the introductory example, the domestic robot may acquire images of a guest under specific conditions (e.g. in evening, indoor scenes), whilst the general task of recognizing that person is clearly not limited to that particular circumstance. Thus, a meta-learning algorithm may be able to infer base-level learning algorithms able to compensate to expected or foreseeable biases – which $p_{D,\mathcal{T}}$ shall describe – by processing (meta-training on) relevant experience $\mathbf{D}$. The i.i.d. requirement of the test splits stems from a rather different rationale, i.e. that of *estimating* generalization. As we noted in Section 3.1, most learning algorithms internally use the empirical risk on D_{tr} (or some other correlated measure), and so do also many base-level algorithms in meta-learning. Thus, typically, one cannot obtain an unbiased estimate by only maintaining one single split. Under this view, the practice of "splitting the datasets" aligns with the effective goal of meta-learning: finding learning algorithms that generalize on a given meta-distribution.

4.3 Friends and Family

We now discuss several research areas considered to be closely related to meta-learning. For some of them, we try to reformulate the respective central questions as meta-learning problems, using the notation and concepts introduced in the previous section. These related fields are typically characterized by the study of learning scenarios that involve the presence of multiple tasks. Hyperparameter optimization is an exception as it finds a common denominator with meta-learning in the search for a learning algorithm.

4.3.1 Lifelong and Continual Learning

The ability of humans and animals to continually process external stimuli, react and adapt to changing environments and situations has for long motivated the study of artificial systems able to handle a stream of data and learn from it. Lifelong (LL) and continual learning (CL) study the online learning scenario where the data originate from a possibly infinite set of tasks. Early works of the eighties on meta-learning [Schmidhuber, 1987, Hinton and Plaut, 1987] considerated exactly this kind of setting. In fact, lifelong and continual learning could be considered as the online, non-stationary, counterparts (or instantiations) of meta-learning. However, typical algorithms that follow under the LL or CL umbrella do not explicitly distinguish between a meta and a base-level of learning and thus do not explicitly seek to infer base-level learning algorithms from data, with some recent exception [e.g. Denevi et al., 2018a]. We refer the reader to [Thrun, 1996] for early work on the subject and [Parisi et al., 2019] for a recent review that also emphasises on the biologically-inspired aspects and connotations of these fields.

A central topic in lifelong and continual learning is the issue of *catastrophic forgetting* or interference, whereby the learning system overrides abruptly past knowledge and the underlying model performance at past tasks quickly decays. Several strategies have been proposed to alleviate these phenomena, where the main difficulty is to devise methods that incur only in an acceptable (additional) computational cost.

4.3.2 Multi-task Learning

There are several links between multi-task and meta-learning. Both learning paradigms seek to leverage on the presence of a family of related tasks and make their own the idea of manipulating the inductive bias, aggregating and elaborating training signals that originate from multiple sources [Caruana, 1998, Maurer et al., 2016, Evgeniou et al., 2005]. In fact, some early work in meta-learning use almost unmodified versions of multi-task learning algorithms to perform meta-learning [Baxter, 1995]. Also several more recent methods undoubtedly inherit much in terms of algorithmic concepts and ideas from the multi-task learning literature, from weight sharing (Section 3.2.1) [Bertinetto et al., 2019, Franceschi et al., 2018a] to regularization (Section 3.2.2) [Denevi et al., 2019]. Whereas one can say that meta-learning experienced a quite "unruly" growth (and, arguably, has yet to come of age), the development of multi-task learning has been comparatively more balanced, accompanied by a steady stream of studies that initiated in the nineties. Several aspects and declination of multi-task learning have been incorporated, by now, in "standard machine learning pipelines" – it suffices to think about weight sharing up to the second last layer of neural networks for multiclass classification or multivariate regression problems. Yet, the multi-task learning problem is, inherently, a single level one. Indeed, the learning setting and data regime of multi-task learning involves the presence of a finite number of tasks and the availability (at training time) of a number of examples for each task. Hence, the dimension of learning is across examples, even though they may originate from multiple tasks. Multi-task learning algorithms ought to deduce rather than induce at the task level, as the generalization should happen on data coming from already seen tasks.

With our notation introduced in Section 4.2, the meta-distribution of a multi-task learning problem with $\mathbf{N}$ tasks $\{\tau^j\}_{j=1}^{\mathbf{N}}$ assumes the form of

$$\mathbb{E}_D[p_{D,\mathcal{T}}] = p_{\mathcal{T}} = \frac{1}{\mathbf{N}}\sum_{j=1}^{\mathbf{N}} \delta_{\tau^j} \quad \text{and} \quad p_D(\cdot|\tau^j) = 0 \;\text{ unless }\; D \sim \tau^j$$

where δ_z is the Dirac delta function with peak at z. Hence, the marginal task distribution is a discrete distribution with finite support and the conditional distribution of the data given a task is non-zero only on datasets sampled from that task. A multi-task dataset is then given by $\mathbf{D} = \{D_i\}_{i=1}^{\mathbf{N}}$ with $D^j \sim \tau^j$. This may be seen as sampling from the meta-distribution exactly $\mathbf{N}$ times without replacement (on the task side) and "discarding" the second elements of the splits. As in multi-task learning we do not seek for generalization at the algorithm level and only care about the given tasks, we do not need the second split. This restatement, although unnecessary convoluted, might suggest that multi-task learning is, in fact, a subset of meta-learning, one that allows for (and benefits from) a different, specialized, algorithmic approach.

4.3.3 Transfer Learning

In its broader sense, the term "transfer learning" refers to the general concept of sharing knowledge between various learning problems [Thrun and Pratt, 1998b], possibly (but not necessarily) distinguishing from source and target tasks. Under this meaning, meta-learning, lifelong and continual learning, as well as multi-task learning, constitute particular learning scenarios and problem settings within the larger area of transfer learning.

In a narrower sense, transfer learning designates the particular setting where there is a clear distinction between source and target tasks [Pan and Yang, 2009]. Ultimately, only these latter are the tasks of interest, while the first one are only considered as ancillary to improve performances on the target tasks. Let us consider, for simplicity, the case where there is exactly one source and one target task. We denote them by τ_s and τ_t respectively. At training time, one has access to a dataset of examples $D_\mathrm{s} \sim \tau_\mathrm{s}$. Available data for the target task, D_t, may be labelled (inductive transfer) or unlabelled (transductive/unsupervised transfer) and it is usually much scarcer than that of the source task (especially in the inductive case). Then, a transfer learning algorithm seeks to utilize D_s together with what is available from τ_t to obtain a model that generalizes well at τ_t – hopefully, better than what it could be otherwise possible by using D_t alone. In this restricted view, one may interpret transfer learning as a special case of meta-learning in which the marginal task distribution

$$\mathbb{E}_D[p_{D,\mathcal{T}}] = p_\mathcal{T} = \frac{1}{2}(\delta_{\tau_\mathrm{s}} + \delta_{\tau_\mathrm{t}})$$

is finite and the (final) performance measure is sensitive only to errors on τ_t (i.e. $\ell_\mathrm{s} = 0$). Note that this latter statement does not imply that one cannot use supervisory signal coming from the source task, but only that the quality of the resulting hypothesis is measured solely against ℓ_t.

With neural network models, a particularly simple yet effective technique for inductive transfer learning is a two-stage approach whereby in a first stage the network is trained only on the large set D_s. Labelled examples of D_t are then used to "fine-tune" the network weights[14] in the second stage, typically utilizing explicit or implicit regularization techniques (biased regularization, aggressive early stopping, small learning rate, etc.) to prevent overfitting. Finn et al. [2017] recently proposed an adaptation of this technique to the meta-learning (few-shot) setting which has been proven particularly effective and fruitful. We shall discuss it in detail in Section 4.4.2.3.

4.3.4 Domain Adaptation

The learning scenario of domain adaptation problems is very close to the narrower interpretation of transfer learning discussed above. Yet, domain adaptation focuses on a more restricted, but practically very relevant, setting where the factors of variations are bound to lie in changes of the distribution between source and target tasks (called *distribution shifts*), but the domains remain the same (i.e. the input and output space, as well as the performance measure, remain unvaried) [Pan and Yang, 2009]. Accordingly, in domain adaptation one usually speaks about source and target distributions rather than tasks. Often, also a temporal component is taken into consideration, whereby a

[14] In this context, "fine-tuning" refers to the practice of warm-starting the optimization related to the target tasks with the weights obtained in the previous learning stage.

system that receives and process data streams should face and adapt to several changes over time. Distribution shifts may, for instance, reflect an underlying non-stationarity of the phenomenon of interest (e.g. natural evolution of fashion trends), sudden and abrupt changes (e.g. a pandemic outbreak) or differences between the available data and the target scenario (e.g. simulation versus real-world [Christiano et al., 2016]). Decomposing $p_{x,y} = p_y(\cdot|x)p_x = p_x(\cdot|y)p_y$, one can consider the different situations where the shift affects only one term [Kouw and Loog, 2018]. *Prior shift* concern changes in p_y, *covariate shift* variations in p_x and *concept shift* in $p_y(\cdot|x)$. For instance, if the task is to diagnose a particular disease from a series of biometric readings and medical analysis, these special cases could occur, in order: in the event of an outbreak (the prior probability of having contracted the sickness rises as an increased number of people becomes infected); if there is a change in the tested population from the source data (different populations might have different characteristics, e.g. different weight and height distributions) or, finally, if new symptoms become associated with a disease, e.g. in case of a mutation. As it is the case for the restricted view of transfer learning, domain adaptation is traditionally tackled with ad-hoc singe-level or multi-stage learning algorithms [Pan and Yang, 2009]. However, meta-learning methodologies have been recently proposed [Li et al., 2018a] for domain generalization, which addresses the topic of developing learning algorithms that are robust by design to entire classes of domain shifts.

4.3.5 Hyperparameter Optimization

While meta-learning refers to a paradigm, hyperparameter optimization is rather related to a specific problem in machine learning and regards a set of techniques to tackle it. Hence, it is perfectly reasonable to perform hyperparameter optimization to tune the configuration parameters of a meta-learning algorithm or to apply a meta-learning technique to boost certain aspects of an hyperparameter optimization method. There is a branch of HPO that deals with the problem of developing methods for hyperparameter optimization capable of exploiting information stemming from multiple tasks and sources [Perrone et al., 2018], possibly with the explicit aim of generalizing to unseen (HPO) tasks [Schilling et al., 2015]. As we shall shortly see (Section 4.4.2.1), some instantiations of pool-based meta-learning algorithms are also quite related to this last aspects.

However, the ties between the two subjects go beyond these points, also when considering the standard applications of hyperparameter optimization. Whereas the relationships between meta-learning and the other machine learning sub-field discussed so far are mostly related to the circumstances of learning, hyperparameter optimization shares with meta-learning a common, general, aim. We will give substance to this argument in the second part of the thesis, but start here with the analysis of this relationship by restating the hyperparameter optimization problem discussed in Chapter 3 using the meta-learning terminology introduced so far.

A standard HPO problems (3.6) involves typically a single task $\bar{\tau} = p_{x,y}$. This corresponds to a (degenerate) meta-distribution of the type

$$\mathbb{E}_D[p_{D,\mathcal{T}}] = p_{\mathcal{T}} = \delta_{\bar{\tau}} \quad \text{and} \quad p_D(\cdot|\bar{\tau}) = 0 \;\text{ unless }\; D \sim \bar{\tau}.$$

Taking a sample from $p_{D,\mathcal{T}}$ effectively means drawing two datasets D_{tr} and D_{ts} from $p_{x,y}$ (where the latter comprises i.i.d. examples). By simply "renaming" D_{ts} into D_{val} we see that the expression of the empirical error of a learning algorithm defined in (4.4) is equivalent to the that of the validation error in (3.5), net of the formal absence of the hyperparameter vector λ in (4.4). The issue is quickly addressed by identifying configurations with algorithms. More precisely, given a (standard) learning algorithm $\mathcal{A}$, with its hyperparameter space Λ, we can denote by $\mathcal{A}_\lambda : \mathcal{D} \to \mathcal{H}_\lambda$ the corresponding "hyperparameterless" algorithm, for each $\lambda \in \Lambda$. Hence (4.4) reads

$$\hat{\boldsymbol{\mathcal{E}}}(\mathcal{A}_\lambda, \mathbf{D}) = \hat{\mathcal{E}}(\mathcal{A}_\lambda(D_{\text{tr}}), D_{\text{val}}) = \frac{1}{|D_{\text{val}}|} \sum_{(x,y)\in D_{\text{val}}} \ell(\mathcal{A}_\lambda(D_{\text{tr}})(x), y) = \hat{\mathcal{E}}(\mathcal{A}(D_{\text{tr}}, \lambda), D_{\text{val}}),$$

which is indeed the same as (3.5). This means that for the degenerate case of a meta-distribution that comprises only a single task, the (standard) objectives of hyperparameter optimization and meta-learning coincide. Note that this is not the case for the other sub-areas of machine learning described so far, where typically the splitting between training and validation (test) sets is not taken into account.

Going further with this view, we may consider that hyperparamter optimization techniques are particular implementations of meta-learning algorithms (4.5) which act in spaces of learning algorithms defined around the hyperparameters, i.e.

$$\boldsymbol{\mathcal{H}} = \{\mathcal{A}_\lambda : \mathcal{D} \to \mathcal{H}_\lambda : \lambda \in \Lambda\}.$$

On the other hand, as anticipated in the previous section, a number of meta-learning algorithms effectively search in algorithmic spaces parameterized by "standard hyper-parameters" (feature extraction, regularization, initialization, etc.). This provides the other side of the relationship between hyperparameter optimization and meta-learning. In the next section we shall look at some of these methods more closely. Specifically we dedicate Section 4.4.2 to the description of algorithmic strategies for MTL, which fall more naturally in this view.

4.4 Techniques for Meta-Learning

Here we describe various proposed implementations of the meta-learning algorithm mappings (4.5), discussing their relative search spaces $\boldsymbol{\mathcal{H}}$ and the expected data regimes. We focus on recent methods that often target (supervised) few-shot learning scenarios, being more closely related to the applications proposed in this work (Section 7.2), but also offer an overview of other classical approaches.

As meta-learning is a fast-paced field of research, categorizations tend to evolve quite quickly as well and vary depending on the particular perspective from which one looks at the field. Vanschoren [2019] sorts meta-learning algorithms depending on the type of experience (meta-data) they process – or on the nature of the meta-distribution $p_{D,\mathcal{T}}$, following our terminology. He considers that meta-data may pertain model evaluations[15], task properties or prior models. On rather different terms, Hospedales et al. [2020] propose a fine-grained cross-section taxonomy that develops around three independent axes. The meta-representation axis, the first and most diverse, concerns the way a meta-learning algorithm retains and uses past knowledge; the meta-optimization axsis regards the policy that $\boldsymbol{\mathcal{A}}$ implements to search in the space of algorithms $\boldsymbol{\mathcal{H}}$; the

[15] We regard this branch as an application of meta-learning to hyperparameter optimization.

meta-objective axis relates to the aim for which the meta-learning algorithm is designed. Since our analytical review covers comparatively less material than [Hospedales et al., 2020], we propose a simpler categorization related to the characteristics of $\mathcal{H}$. We differentiate between *model-based strategies* where $\mathcal{H}$ is comparable to a standard hypothesis space (RNNs, temporal convolutional NN, memory-augmented NNs ...), and *algorithmic strategies* that feature meta-hypothesis spaces $\mathcal{H}$ containing proper (even if elementary) learning algorithms (nearest neighbour, logistic regression, ERM with gradient descent, ...).

4.4.1 Model-based Meta-learning

The experience associated to the meta-level is composed of sets of examples. Hence, when developing a meta-learning algorithm, one may look at the various classes of statistical models which are designed to process sequences or sets of data. Some of these classes lend themselves well to play the role of learning algorithms in a meta-learning scenario, after some necessary adjustments and modifications. For instance, connectionist models (Section 2.3) such as recurrent [Hochreiter et al., 2001, Santoro et al., 2016] or temporal convolutional neural networks [Mishra et al., 2018], as well as adaptation of other feed-forward models to work on sets [Qiao et al., 2018] have been used for this purpose. The core idea is to embed entire training datasets D_{tr} into internal representations which may then be used to regress or classify query points. Model-based meta-learning (MB-MTL) algorithms implement their base-level with function evaluations ("forward passes"). Many of these algorithms find rather close counterparts in techniques of standard learning, beside the clear differences in the type of data that is used at the (meta-)training stage. The following illustrative example should help clarify these statements.

Let us consider an elementary meta-learning algorithm $\mathcal{A}$ that uses at the meta-level a simple (single layer) recurrent neural network. Let $\bar{D}, \bar{\tau} \sim p_{D,\tau}$ be a data-task distribution pair drawn from a meta-distribution, with $\bar{D} = \{(x_i, y_i)\}_{i=1}^{\bar{N}}$. Let us also assume, again, for simplicity, that the $p_{D,\tau}$ describes scalar regression tasks with input of fixed dimensionality. Starting with a given hidden state s_0, which could be randomly sampled or optimized, one option is to process $\bar{D}$ as

$$s_t = \sigma(W_1 x_t + W_2 y_t + W_3 s_{t-1} + b_1) \quad \text{for} \quad t = 1, \ldots, \bar{N} \tag{4.6}$$

obtaining the state $s_{\bar{N}}$. In principle, having seen all the $\bar{N}$ samples, $s_{\bar{N}}$ is a vector that depends on (and represents) the entire dataset $\bar{D}$. One may then regress test points x by computing

$$h_{w,\bar{D}}(x) = W_4 \sigma(W_1 x + W_3 s_{\bar{N}} + b_1) + b_2. \tag{4.7}$$

The equations (4.6)-(4.7) describe a simple RNN with a final linear output layer which returns a scalar value upon seen a sequence of (training) input/output pairs and a test input. In effect, this model may be interpreted as a learning algorithm $\mathcal{A}_w$, parameterized by $w = (W_1, W_2, W_3, W_4, b_1, b_2) \in \mathbb{R}^d$, that associate the hypothesis of Equation (4.7) to a dataset $\bar{D}$; or $\mathcal{A}_w(\bar{D}) = h_{w,\bar{D}}$. Assuming that the performance measure for each task is the mean squared error, the RNN weights w may be simply optimized by minimizing

$$\hat{\boldsymbol{\mathcal{E}}}(\mathcal{A}_w, \mathbf{D}) = \frac{1}{\mathbf{N}} \sum_{j=1}^{\mathbf{N}} \hat{\mathcal{E}}\left(\mathcal{A}_w\left(D^j_{\text{tr}}\right), D^j_{\text{ts}}\right) = \frac{1}{\mathbf{N}} \sum_{j=1}^{\mathbf{N}} \frac{1}{|D^j_{\text{ts}}|} \sum_{(x,y) \in D^j_{\text{ts}}} \left(h_{w,D^j_{\text{tr}}}(x) - y\right)^2$$

by gradient descent (backpropagation through time [Werbos, 1990]), where $\mathbf{D}$ is a meta-training set as described in (4.2). The optimization is performed at the meta-level, across tasks, in a space of learning algorithms given by $\boldsymbol{\mathcal{H}} = \{\mathcal{A}_w : w \in \mathbb{R}^d\}$. The hypotheses returned by $\mathcal{A}_w$, for any given w, do not have any variable that can be tuned on the specific task[16]. Hence, the base-level learning algorithm works by merely evaluating the RNN equations (4.6)-(4.7). The configuration space of $\boldsymbol{\mathcal{A}}$ may include optimization hyperparameters such as learning rates and update rule, as well as dimensionality of the RNN hidden state, type of activation functions and so on. This concludes the specification of this very simple meta-learning algorithm, which is indeed quite similar to a standard RNN learning algorithm for sequences, except for

[16] In fact, the base-level learning algorithm of $\boldsymbol{\mathcal{A}}$ can be interpreted as a fixed-weight neural network [Cotter and Conwell, 1990], as explained in [Hochreiter et al., 2001].

the different type of data that it processes.

One of the main difficulties encountered in developing MB-MTL algorithms is to devise a model with enough capacity and expressivity to serve as a full-fledged learning algorithm, while maintaining a manageable complexity. There are some critical issues with the model from the example. For instance, points processed earlier by the RNN may have a smaller impact on the final state $s_{\bar{N}}$ than those processed later. However, the presentation order should not matter (i.e. $\mathcal{A}_w$ should be permutation invariant) as the datasets do not (typically) possess any temporal structure. S. Hochreiter et al. already raised this issue in an early publication [Hochreiter et al., 2001] in which they propose to use LSTM networks [Hochreiter and Schmidhuber, 1997] instead. The setting described in [Hochreiter et al., 2001] is close to that of the example, except that they consider an online version of the problem (somewhat closer to lifelong learning) where the meta-learning system is supposed to process a stream of data stemming from multiple tasks (one task after the other). In [Hochreiter et al., 2001], the LSTM outputs a prediction for each time step, receiving as input the shifted pairs (x_t, y_{t-1}). During meta-training, it is adapted online using the error at each step as supervisory signal.

Santoro et al. [2016] explored the possibility of employing memory augmented neural networks [Graves et al., 2014, Sukhbaatar et al., 2015] as meta-hypotheses, in one of the paper that contributed sparking interest in meta-learning approaches to few-shot learning. The authors found memory network models superior to LSTMs on regression and multi-class classification meta-learning problems. Mishra et al. [2018] proposed a model-based meta-learning algorithm for few-shot learning based on temporal convolutional neural networks, with a data regime quite similar to that of our example (not online learning). The class of models developed in the paper features causal temporal convolution layers interleaved with attention layers [Vaswani et al., 2017] that allows for a parallel (rather than sequential) processing of the entire training set. The authors attribute the substantial performance gains (on some benchmark datasets) over previously proposed MB-MTL methods to the more direct information processing possible with feed-forward attention models. The cost to pay is, however, the limited and fixed data window that this type of models imposes, which can only be widened by deepening the network, raising its complexity. A slightly different modelling approach is pursued by Qiao et al. [2018], who focus on multi-class classification tasks. Given a standard feed-forward neural network such as a CNN, a class representation is obtained by summing up the embeddings at the second last layer of all examples belonging to a given class. Then, such representation is used as an input to a mapping that outputs a classifier vector for that class. The resulting family of models, which resemble deep sets networks [Zaheer et al., 2017], although limited in scope and applicability, seems particularly well suited to address few-shot learning problems in vision.

One reason of the success of model-based meta-learning framework may be sought in the possibility for the researchers to intervene more directly in the design of the hypothesis space of the underlying base-level learning algorithms, conceivably allowing for more specialization. Yet, the base-level is essentially limited to algorithms that only perform function evaluations. This lack of flexibility may render MB-MTL methods ill-suited to address meta-learning problems where the meta-distribution is highly multi-modal. For instance, this may occur when $p_{D,\mathcal{T}}$ represent cases where the training datasets (observations) have highly varying sizes or situation where tasks are structurally different one to another.

4.4.2 Algorithmic Meta-Learning

Whereas model-based meta-learning techniques adapt models of standard learning to work at a higher level of abstraction, algorithmic meta-learning (A-MTL) methods search in spaces composed of learning algorithms. Hence, at the base-level, they are not limited to produce hypothesis by solely evaluating functions, but can also perform more sophisticated computations such as finding approximate solutions to optimization problems. This entails the additional challenge of developing efficient computational methods or pipelines that allow effective learning at the meta-level, passing information through the execution of entire algorithms. On the other hand, the far greater flexibility of a more complex meta-hypothesis space may help devise more broadly applicable, general-purpose, meta-learning algorithms that are less sensitive to the particular application domain.

4.4.2.1 Pool-based Methods

Perhaps the most direct design pattern for A-MTL methods consists in modelling the base-level using a "pool" of (fixed) standard learning algorithms. For each task drawn from the meta-distribution, one would like to pick the algorithm from the pool that performs the best (executing as few runs as possible, or even none) and return as hypothesis the output of that algorithm. The core idea of pool-based A-MTL methods is to infer at the meta-level a mapping that, given a dataset, chooses (or recommends) a learning algorithm from the pool (ideally, the best performing one) or combines some of them (stacking). As we already mentioned in Section 4.2, this class of approaches is strongly linked to algorithm selection and has received a fairly good amount of attention from approximately the end of the nineties to the early 2000s. We give here an overview of the basic ideas and refer the reader to the reviews [Vilalta and Drissi, 2002, Smith-Miles, 2009, Vanschoren, 2019] and references therein for further details and links to particular implementations.

We describe the simple case of a pool-based meta-learning algorithm $\boldsymbol{\mathcal{A}}$ that prescribes a single recommendation. Consider a set of $K \in \mathbb{N}$ standard learning algorithms of interest $\{\mathcal{A}_k\}_{k=1}^K$ with relative hypothesis spaces $\{\mathcal{H}_k\}_{k=1}^K$ (and fixed configuration parameters) and define the "selection map"

$$\hat{\mathcal{A}} : [K] \times \mathcal{D} \to \bigcup_{k=1}^{K} \mathcal{H}_k; \qquad \hat{\mathcal{A}}(k, D) = \mathcal{A}_k(D),$$

where $[K] = \{1, \dots, K\}$. The main component of a pool-based A-MTL algorithms is given by a "recommender function" $r : \mathcal{D} \to |K| \times \mathcal{D}$ that returns an integer (alongside the received dataset), to pass over to $\hat{\mathcal{A}}$. Then, one can define the meta-hypothesis space of $\boldsymbol{\mathcal{A}}$ as

$$\boldsymbol{\mathcal{H}} = \{\hat{\mathcal{A}} \circ r : \mathcal{D} \to \cup_k \mathcal{H}_k \;:\; r \in \mathcal{R}\}$$

for some space of functions $\mathcal{R}$. Since $\hat{\mathcal{A}}$ is fixed, the meta-learning algorithm is only responsible for finding a good mapping $r \in \mathcal{R}$ that approximately solves

$$\min_{r \in \mathcal{R}} \hat{\mathcal{E}}(\hat{\mathcal{A}} \circ r, \mathbf{D})$$

where $\mathbf{D} \sim p_{D,\tau}$ is a meta-training set. This is, however, not a straightforward exercise. Indeed, the fact that r operates a discrete choice, paired with the potential non-differentiability of algorithms in the pool, makes the resulting meta-learning problem unsuitable for (end-to-end) gradient-based optimization. Furthermore, one remains with the issue of finding a space of functions $\mathcal{R}$ apt to meaningfully process entire datasets (akin to what we discussed in the previous section). In this last regard, much research in this area has focused on the development and usage of so-called meta-features to describe a task or an empirical dataset. Meta-features may include statistical and numerical quantities that describe a dataset, such as its mean, variance, the total number of points, the input and output dimensionalities, and so on. Researchers have also proposed to use as meta-features the performances of hypotheses outputted by simple and computationally light algorithms (landmarking). Hence, a prepossessing step (constituting a first, fixed, component of r) maps datasets into vector representations, on the top of which standard statistical models such as linear or logistic regression or decision trees may be used (constituting a second, learnable, component of r). More recently, Edwards and Storkey [2017] have proposed a variational auto-encoder [Kingma and Welling, 2019] approach for end-to-end learning of meta-features (or "statistics") which could potentially replace hand-engineered one.

To overcome the difficulties in optimization, several researchers have utilized two-stage approaches. The first stage may consist in collecting (meta-)data by executing the algorithms from the pool on the datasets in $\mathbf{D}$, recording e.g. a cross-validation error. One may then compile a table of results that can be processed into a dataset where the inputs are given by the meta-features representing the datasets and the outputs could be either the recorded cross-validation errors or the indices of the best performing algorithm for each task. Meta-training is then cast to a supervised classification or regression problem regarding the learnable component of r.

One of the advantages of pool-based A-MTL approaches lies in the possibility of including a large class of diverse learning algorithms (although meta-training may become increasingly difficult as K grows). The resulting base-level hypothesis space $\mathcal{H} = \bigcup_k \mathcal{H}_k$ may then be large enough to provide fair coverage for complex and highly multi-modal meta-distributions which could contain tasks that are sensibly different one to another. Alongside this, the computational cost of the learned base-level algorithm can be maintaining relatively low. Indeed, due to the recommender function r, only one learning algorithm from the pool (or a few cheap ones if using landmarking) must be run in order to return a hypothesis for any given (test) dataset. However, the two-stage meta-training process involves a series of hand-engineered steps that would hardly benefit from the advances in representation learning and end-to-end training of the last decade. Moreover, the choice of a fixed pool of learning algorithms may incur in underfitting issues in the presence of tight meta-distributions, as the $\mathcal{A}_k$ cannot be adapted, if needed. Clearly, a pool-based A-MTL base-learning algorithm such as the one described above may performs at most as well on a task as the best performing algorithm in its pool. Stacking or ensemble techniques may alleviate this issue to a certain extent, but do not solve it.

Following a quite opposite conceptual approach, more recently, researchers have pursued the development of A-MTL methods that use a single standard learning algorithm $\mathcal{A}$ as the "backbone" to design $\boldsymbol{\mathcal{H}}$, but allow for a thorough search and optimization of some of its components, carried out at the meta-level. Instance-based and parametric techniques, which we shall describe next, follow this latter pattern. Intuitively, this allows meta-learning systems to operate over a continuum of learning algorithms, rather than having to choose among a finite, hand-picked, set.

4.4.2.2 Instance-based Methods

Methods such as k-nearest neighbour (kNN) or kernel density estimation (KDE) are a class of simple and flexible non-parametric algorithms that do not require a "training stage", and lazily evaluate test point based on the distance to the training (or support) instances [see e.g. Murphy, 2012, Ch. 1 and 14]. Performances of kNN or KDE regression or classifications models are, however, rather sensitive to the choice of the feature space and distance metric, which could be considered as hyperparameters for these (standard) learning algorithms. Instance-based A-MTL methods [Vinyals et al., 2016, Snell et al., 2017, Sung et al., 2018] use these types of learning algorithms at the base-level, adapting some of their configurations parameters at the meta-level, across tasks. Many authors [Franceschi et al., 2018a, Rusu et al., 2019, Sung et al., 2018] refer to this class of techniques as *metric-based* strategies since the core idea is to meta-learn a distance metric adapted to the meta-distribution of interest.

Let $\bar{D}, \bar{\tau} \sim p_{D,\tau}$ be a data-task distribution pair drawn from a meta-distribution, with $\bar{D} = \{(x_i, y_i)\}_{i=1}^{\bar{N}}$ and $x_i \in \mathcal{X}$. Let $(x, y) \sim \bar{\tau}$ be a test datapoint. Hypotheses outputted by instance-based kNN or KDE algorithms can be written as [Vinyals et al., 2016]

$$h_{\kappa,\bar{D}}(x) = \sum_{i=1}^{\bar{N}} \kappa(x, x_i) y_i$$

where $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is akin to a kernel function. The map κ, which can be interpreted also as an attention mechanism or a relation mapping, is an hyperparameter of the corresponding (standard) learning algorithm (cf. Section 3.2.1). Following the notation introduced in Sections 4.2 and 4.3.5, we index the corresponding algorithms by κ: $\mathcal{A}_\kappa(\bar{D}) = h_{\kappa,\bar{D}}$. Then, a typical instance-based meta-learning algorithm searches for a base-level learning algorithm in a meta-hypothesis space given by $\boldsymbol{\mathcal{H}} = \{\mathcal{A}_\kappa \, : \, \kappa \in \mathcal{G}\}$ for some (possibly parameterized) set $\mathcal{G}$.

Vinyals et al. [2016] were the first to introduce this approach in a few-shot learning context for supervised classification. They decompose κ into feature map $\chi : \mathcal{X} \to \mathcal{V}$ and distance metric $\delta : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ with $\kappa(x, x') = \delta(\chi(x), \chi(x'))$ and fix

$$\delta(z, z_j) = e^{c(z,z_j)} \left(\sum_{i=1}^{\bar{N}} e^{c(z,z_i)} \right)^{-1} \qquad \text{where} \qquad c(z, z') = \frac{z^\mathsf{T} z'}{\|z\| \|z'\|} \tag{4.8}$$

is the cosine distance. The mapping χ is instead implemented by a deep neural network with parameters w (more precisely, a bidirectional LSTM on the top of a CNN feature extractor for applications to visual domains). Additionally, the feature map takes as input also the entire training set $\bar{D}$ to provide a "full contextual embedding [Vinyals et al., 2016]" (this is accomplished by the bidirectional LSTM). The author also propose to decouple support and query feature maps, which can be particularly useful when training observations are not points in the task domain. The search for a good metric at the meta-level is performed by maximizing the log-likelihood[17] on a meta-training set **D**

$$\hat{\mathcal{E}}(\mathcal{A}_\kappa, \mathbf{D}) = \frac{1}{\mathbf{N}} \sum_{j=1}^{\mathbf{N}} \frac{1}{|D^j_{\mathrm{ts}}|} \sum_{(x,y) \in D^j_{\mathrm{ts}}} \log \left[h_{\kappa, D^j_{\mathrm{tr}}}(x) \right]_y \tag{4.9}$$

with respect to w (the learnable parameter of κ), assuming $y \in \{1, \ldots, c\}$ indicates the class label. Because of (4.8), the hypotheses for each episode are in fact viewed in (4.9) as probability distributions over the episode classes. Since every component of the proposed base-level (non-parametric) algorithm is differentiable, the meta-training loss (4.9) may then be optimized simply by gradient ascent.

Snell et al. [2017] refine the algorithm of Vinyals et al. by introducing class prototypes, that consist in aggregating the embeddings of all example of a given class in the feature space, before computing the distance. In other words, the feature map χ is implemented as a set function, sothat in the feature space there is exactly one (prototype) vector per class. They further simplify the method dropping the full contextual embedding and extend the application range to zero-shot learning. Sung et al. [2018] propose to parameterize and meta-learn also the distance function δ and use a squared error on the resulting "relation scores" in place of the log-likelihood. Garcia and Bruna [2018] instead implement δ with a graph neural network.

Instance-based A-MTL algorithm retain meta-level knowledge in the parameters of the kernel function κ. Akin to MB-MTL and unlike pool-based A-MTL algorithms, the resulting meta-level optimization problems are typically smooth and unconstrained, rendering the search for meta-hypothesis potentially easier. One limitation, inherited by the instance-based methods which these meta-learning algorithms utilize, is that the computational cost of the base-level hypotheses increases with the size of the training sets. In fact, instance-based methods have been proposed until now only in few-shot learning contexts. The prototype approach [Snell et al., 2017] may, in part, alleviate

[17] Equivalent to minimizing the multi-class cross-entropy loss (Section 2.2).

this issue at the expense of expressiveness and limitation in application scope: for instance, it is not clear how it could be extended to regression tasks.

4.4.2.3 Parametric Methods

This family of A-MTL methods uses at the base-level learning algorithms associated to parametric models, adapting at the meta-level some of their configuration parameters. There is a variety of possible strategies to implement parametric A-MTL algorithms depending on which aspect of the underline algorithm one wishes to lift to the meta-level, with an abundant number of publications in the recent years [e.g. Andrychowicz et al., 2016, Ravi and Larochelle, 2017, Finn et al., 2017, Bello et al., 2017, Wichrowska et al., 2017, Franceschi et al., 2018a, Nichol et al., 2018, Bertinetto et al., 2019, Rusu et al., 2019, Zintgraf et al., 2019, Metz et al., 2019]. Typically, the base-level is more "complex" than that of model or instance-based meta-learning methods, as often parametric (standard) learning algorithms involve minimizing a loss function (e.g. regularized ERM) defined on each training set. For this reason, often this class of A-MTL algorithms is also called *optimization-based* [Hospedales et al., 2020]. The meta-level search is frequently implemented by gradient-based optimization, although also reinforcement learning [Bello et al., 2017] and simpler first order search schemes [Nichol et al., 2018] have been proposed. We will thoroughly discuss various aspects and implementation details of the gradient-based subset of methods belonging to this family in the following chapters of the thesis, linking the resulting meta-learning problems to (gradient-based) hyperparameter optimization. In Section 7.2 we will present our proposed implementation [Franceschi et al., 2018a] that uses logistic regression as the backbone algorithm for the base-level and learn representation mappings at the meta-level (or "hyper-representations", to highlight the ties with HPO). We describe here, instead, what it is possibly the best known and most popular instantiation of the parametric A-MTL class, proposed by Finn et al. [2017] in a few-shot learning context and extended by various authors thereafter. Later we discuss some central concepts behind the series of works which specifically focus on learning optimization rules [Andrychowicz et al., 2016, Bello et al., 2017, Wichrowska et al., 2017, Metz et al., 2019], a sub-field of research dubbed learning to optimize.

Consider a meta-distribution $p_{D,\mathcal{T}}$ where input and output spaces $\mathcal{X}$ and $\mathcal{Y}$ are structurally fixed[18]. Given some (standard) hypothesis space $\mathcal{H}$ (for instance neural network with a fixed architecture) the "model-agnostic meta-learning" (MAML) [Finn et al., 2017] algorithm $\boldsymbol{\mathcal{A}}$ performs at the base-level $K \geq 1$ steps of gradient descent on the task-specific empirical risk, meta-learning the starting point of the optimization routine (see Section 3.2.3). This may be interpreted as learning at the meta-level a model that, although not necessarily good for any specific task in $p_{D,\mathcal{T}}$, is easily adapted with few steps of gradient descent. Crucially, MAML differentiates from simpler fine-tuning approaches discussed in Section 4.3 in that the parameters of the "source" model are optimized end-to-end, taking into account the fine-tuning steps (on data and tasks drawn from a given meta-distributions). As before, let $\bar{D}, \bar{\tau} \sim p_{D,\mathcal{T}}$ be a data-task distribution pair. Let $h_w : \mathcal{X} \to \mathcal{Y}$ be an hypothesis parameterized by a vector $w \in \mathcal{W} \subseteq \mathbb{R}^d$ and call $L(w, \bar{D})$ the empirical error of h_w on the task observation points $\bar{D}$ (see Section 2.1 and Equation 2.3). Starting from $w_0 = \lambda \in \mathcal{W}$, the parameters are updated via K steps of gradient descent

$$w_k = w_{k-1} - \eta \nabla L(w_{k-1}, \bar{D}) \qquad \text{for} \quad k = 1, \dots, K; \tag{4.10}$$

where $\eta > 0$ is a learning rate (in the original method K and η are both regarded as configuration parameters of $\boldsymbol{\mathcal{A}}$). The final parameters w_K depend on the training set $\bar{D}$ and on the starting point[19] λ. Hence we write $w_K = w_K(\lambda, \bar{D})$ We may then define the family of base-level learning algorithms, parameterized by λ as $\mathcal{A}_\lambda(\bar{D}) = h_{w_K(\lambda,\bar{D})}$, with $w_K(\lambda, \bar{D})$ obtained from (4.10). The meta-hypothesis space of MAML is given by $\boldsymbol{\mathcal{H}} = \{\mathcal{A}_\lambda \, : \, \lambda \in \mathcal{W} \subseteq \mathbb{R}^d\}$. One may view h_λ as a "meta-model" since the knowledge is accumulated in the initialization weights λ. The search in $\boldsymbol{\mathcal{H}}$ may be carried out by minimizing the empirical error (4.4) of $\mathcal{A}_\lambda$ with respect to λ by gradient descent, as long as L and h are smooth mappings. Computing the gradient of the test (validation) loss requires computing higher order derivatives of the task-specific training error; we will discuss the details of this procedure in Chapter 5.

[18] However, the encoding of $\mathcal{Y}$ (e.g. the classes) may change depending on the tasks.

[19] We note that the parameters w_K would not depend on λ only in the case of exact optimization of an objective with unique global minimum, a situation quite far from that addressed in [Finn et al., 2017].

Several authors have proposed modifications and extensions of this algorithm, for instance by learning at the meta-level also the step-size η [Li et al., 2017b, Antoniou et al., 2019], adapting at the base-level only a portion of the parameters [Zintgraf et al., 2019], learning λ with a first-order search scheme [Nichol et al., 2018] and many others [Antoniou et al., 2019]. Rusu et al. [2019], instead, move the gradient-based optimization of the base-level (4.10) to a low-dimensional learned latent space, using a stochastic encoder-decoder architecture [Goodfellow et al., 2016, Ch. 14] that maps data to the parameters of h. The resulting base-level algorithm is then parameterized by the weights of the encoder and decoder mappings (besides other components) rather than directly by vectors in $\mathcal{W}$. The motive of [Rusu et al., 2019] (and also e.g. of [Zintgraf et al., 2019]) was to address possible overfitting issues caused by a too large hypothesis space at the base-level (adaptation of too many parameters) reported e.g. in [Mishra et al., 2018] on experiments in few-shot learning contexts. Interestingly, one possible interpretation of the work by Rusu et al. is that of replacing the single initialization vector of MAML, λ, by a conditional initialization $\lambda(\bar{D})$ that depends (via the encoder-decoder mapping) from the observed task-specific data. This extends the idea of conditional meta-learning, already present in some form in MB and instance-based methods [Vinyals et al., 2016, Sung et al., 2018, Oreshkin et al., 2018], to the parametric A-MTL family leading to very strong performances on a number of benchmark few-shot learning datasets. In this direction, very recently Wang et al. [2020] proposed a general method for conditional meta-learning based on a structured prediction approach that may be applied at the meta-level on the top of a vast range of base-level parameterizations.

Learning to Optimize. The update rule of gradient descent procedures for minimizing training errors constitutes a central component of many parametric learning algorithms, affecting runtime and generalization. This is particularly relevant especially when the underlying objective is non-convex, as it is the case when dealing with neural networks. It is ubiquitous practice to employ (structurally fixed) update mappings such as stochastic gradient descent (see Section 2.4) and only tune a few hyperparameters that specify the rule's behaviour. Analytic and empirical research in the optimization area has led to the development of more complex optimization methods (Section 2.4.3). Some of these routines may well be tailored to tackle optimization problems that arise e.g. in deep learning [see e.g. Goodfellow et al., 2016, ch. 8], but they are still general-purpose in nature, not depending on the data that instantiate the relative optimization objectives. One possible avenue for further improvements is to search for update rules that are specialized to small classes of problems, specifically taking into account the data regime and distribution, in an attempt to circumvent the "no free lunch theorem for optimization" [Wolpert and Macready, 1997]. This effort clearly fits into an (algorithmic) meta-learning context. In fact, early works that date back to the nineties and contributed shaping the meta-learning field [Bengio et al., 1991] follow this conceptual strand. Recently a number of publications [Andrychowicz et al., 2016, Bello et al., 2017, Ravi and Larochelle, 2017, Wichrowska et al., 2017, Metz et al., 2019] has marked a resurgent interest in this specific topic, although, to date, the interest seems still quite limited to the research community.

Mathematically, the meta-learning problem that derives from the search for update rules does not differ much from the one previously presented. Intuitively, at the base-level, instead of updating the parameters with a fixed gradient descent procedure as in (4.10) one may instead use a general update mapping Φ (see (2.28)) and perform

$$s_k = \Phi_k(s_{k-1}, \alpha, \bar{D}) \qquad \text{for} \quad k = 1, \dots, K;$$

where α are configuration parameters and $s_k = (w_k, v_k)$ is the optimizer state. There are mainly two different approaches to learning to optimize. On the one hand, [Andrychowicz et al., 2016, Wichrowska et al., 2017, Metz et al., 2019] implement Φ as a parameterized mapping such as an LSTM or a feed-forward neural network. The maps may take as input (also) the gradients of L at each step. In these cases, the parameters α is interpreted as the weight vector of the underlying model, which may then be optimized directly, at the meta-level, with gradient-based procedures. On the other hand, [Bello et al., 2017] takes a symbolic approach and represent Φ via its computational graph (see Sections 3.1 and A.2.2), in a work that is rather similar in spirit and methodology to [Zoph and Le, 2017] about neural architecture search. In [Bello et al., 2017], α

is a string of a custom defined “domain specific language”, that is then mapped to a computational graph, in a predefined manner. The string is the output of a “controller” model, implemented by a recurrent neural network. Then, the meta-level search is over the parameters of the controller RNN and is carried out with a reinforcement learning algorithm.

4.5 Interim Summary

In this chapter, we presented the meta-learning problem and reviewed various techniques and approaches that characterize the field. We started with an informal introduction of fundamental concepts with an example application in a simplified robotic scenario. We saw that the utility (or necessity) of meta-learning arises when one is interested in devising a learning system that performs well at (future) target tasks that are not observable at training time but share some commonalities, mathematically captured by the concept of meta-distribution. One may then use other available data, similar to the tasks of interest, to “bootstrap” the learning system, adapting it to the types of problems that it is expected to encounter “at test time”.

We formalized these concepts in Section 4.2. We defined the meta-distribution object as a joint distribution of observable data and tasks and argued that this perspective allows for a more natural treatment of the meta-learning problem. It also offers a conceptual justification to the practice of “splitting” datasets, a first factor that links meta-learning to hyperparameter optimization. We finally gave a definition of meta-learning algorithm that takes inspiration from established literature, but also clearly indicates that the (main) dimension of learning is the meta-level.

We then proceeded by studying similarities and differences between meta-learning and various other fields. We saw that, while most of the areas typically associated with meta-learning are marked by the presence of multiple tasks, hyperparameter optimization shares with meta-learning a common aim – argument that we will further develop in the following chapters.

The last section of this chapter reviews the main approaches to meta-learning, which we divided into model-based and algorithmic methods. For each family of techniques, we presented a “minimal instantiation”. These examples revealed that the design of many recent meta-learning algorithms (especially those introduced in Sections 4.4.2.2 and 4.4.2.3) relies on lifting to the meta-level some aspects of a standard learning algorithm – aspects that one may typically identify as hyperparameters.

4.6 Summary and Discussion of Concepts Introduced in Part I

In the background part of the thesis, we have introduced, discussed and touched upon a number of concepts, fields and subfields of machine learning that span from smooth optimization to multitask and meta-learning (refer also to Table 1.1). Here we offer a brief summary aimed at highlighting the relations and links among (some of) these areas of research.

Supervised learning [Mitchell, 1997, Friedman et al., 2001, Murphy, 2012], which, in a nutshell, consists in inferring functions from data, is deeply related to most of the concepts treated in the thesis. In hyperparameter optimization, the execution of a supervised learning algorithm becomes part of the objective function of the HPO problem (see Section 3.1); in meta-learning, supervised learning algorithms may constitute the base-level of learning (see Section 4.2), specifically, in the class of MTL techniques which we called "algorithmic" (Section 4.4.2).

Optimization [Polyak, 1987a, Nesterov, 2013, Nocedal and Wright, 2006, Bottou et al., 2018] is one of the pillars of machine learning and, as such, it is ubiquitous. As many optimization methods are general-purpose procedures, they typically have several configurations parameters which must be set when used within a (supervised) learning algorithm. In this regard, HPO is related to optimization in that one may formulate an HPO problem to find good hyperparameters of an optimization method (that is, to treat the configuration hyperparameters of an optimization procedure as hyperparameters of an HPO problem). In particular, tuning learning rates or step-sizes of gradient descent methods is a topic that has received much attention in (and to a certain extent, predates) HPO [Jacobs, 1988, Almeida et al., 1999, Schraudolph, 1999, Schaul et al., 2013, Baydin et al., 2018a, Wu et al., 2018b]. A particular subfield of meta-learning, dubbed learning to optimize [Andrychowicz et al., 2016, Bello et al., 2017, Wichrowska et al., 2017, Metz et al., 2019], explicitly seeks to learn from data optimization procedures, where often the update map is a richly parameterized neural net that takes as input gradients and possibly other local information (this may be seen as an highly parameterized version of the HPO regarding optimizers' parameters).

Hyperparameter optimization [Moore et al., 2011, Bergstra et al., 2011, Bergstra and Bengio, 2012, Maclaurin et al., 2015a, Bergstra et al., 2013, Hutter et al., 2015, Franceschi et al., 2017, Feurer and Hutter, 2018], the central topic of the previous chapter, is an essential tool in machine learning, where the aim is to seeking good values for configuration parameters which (inevitably) arise when developing learning and meta-learning algorithms. We presented it as a general problem, which may then be instantiated depending on which hyperparameters one seeks to optimize. For instance, in neural architecture search [Zoph and Le, 2017, Cai et al., 2018, Liu et al., 2019, Luo et al., 2018, Real et al., 2019], the focus is on finding computational structures, often interpreted as directed graph, for neural nets. Another fruitful source of HPO problems is multi-task learning [Caruana, 1998, Maurer et al., 2016, Evgeniou et al., 2005], where one often seeks to tune components of the mechanisms that regulate how information is shared among multiple tasks. We started analyzing in Section 4.3.5 how HPO intersect with MTL, and we will dive into this this topic in the second part of the thesis. One may see links between the online version of the HPO problem (see Section 3.3), lifelong learning [Thrun and Pratt, 1998b, Chen and Liu, 2018] and (online) domain adaptation [Jain and Learned-Miller, 2011], as these all share a temporal component (data streams).

Meta-learning[20], the topic of this chapter, is a quite vast field in rapid evolution [Thrun and Pratt, 1998a, Schmidhuber, 1987, Baxter, 1998, Finn et al., 2017, Franceschi et al., 2018a, Denevi et al., 2018a, Ravi and Larochelle, 2017, Vinyals et al., 2016, Hospedales et al., 2020]. As such, it escapes a general and widely accepted formalization, which we nevertheless tried to provide in Section 4.2. Our definition of MTL takes advantage of a parallelism with the classic definition of machine learning algorithm given by Mitchell [1997] that leverages the concept of meta-distribution and stresses that the experience in MTL (primarily) unfolds along the axis of the tasks or episodes. MTL, seen as a learning paradigm, intersects and is related to many other fields and subfields of machine learning, in that one may often conceptualize an MTL adaptation of practices and techniques of standard learning, as we saw e.g. in Section

[20]We remind the reader that in this work we use the term meta-learning as synonym of learning to learn.

4.4. In this way, for instance, one may think of meta-learning initialization [Finn et al., 2017, Antoniou et al., 2019, Rusu et al., 2019], representations [Baxter, 1995, Franceschi et al., 2018a, Bertinetto et al., 2019], update rules (learning to optimize), metrics [Vinyals et al., 2016, Snell et al., 2017], computational structures (neural architecture search) and so on. Yet, all these particular viewpoints should be interpreted, in our opinion, as specific instantiations of a more general paradigm, as much as k-nearest neighbour, support vector machine, decision trees, etc. are particular instantiations of (the concept of) supervised learning algorithm.

Part II

Framework

Chapter 5

Bilevel Programming for Gradient-based Hyperparameter Optimization and Meta-learning

This chapter begins the second part of the thesis, where we present, discuss and analyze a unifying framework for gradient-based hyperparameter optimization and meta-learning, rooted in bilevel programming and algorithmic differentiation.

After a brief introduction of Section 5.1 where we elaborate on few concepts introduced in Chapter 1, we present the framework in Section 5.2 and instantiate it for hyperparameter optimization and meta-learning. Contextually to our reviews of Part I, we discuss the class of learning and meta-learning algorithms that the bilevel framework naturally covers. Then, in Section 5.3 we introduce the gradient-based approach to solve an approximate version of the bilevel program, that branches in two main directions: *iterative* and *implicit*. We discuss how a broader interpretation of the iterative approach, on which we focus more closely, covers a larger class of algorithms and hyperparameters. Section 5.4 concludes with the derivation of the principal procedures to compute efficiently an approximate gradient of the outer objective (the *hypergradient*), including reverse-mode, forward-mode and implicit differentiation methods.

This chapter, as well as Chapters 6 and 7, is based on the articles *"Forward and Reverse Gradinet-based Hyperparameter Optimization"* [Franceschi et al., 2017], *"Bilevel Programming for Hyperparamter Optimization and Meta-learning"* [Franceschi et al., 2018a] and *"On the Iteration Complexity of Hypergradient Computation"* [Grazzi et al., 2020], published at ICML 2017, 2018 and 2020, respectively. We reorganized the material to provide a more fluent account of the work, taking the opportunity to add some extensions. This chapter pertains conceptual and algorithmic aspects of the framework, Chapter 6 is dedicated to an analytical investigation, providing convergence and complexity results for the algorithms of Section 5.4, and Chapter 7 collects a series of numerical simulations inspired by real-world applications.

5.1 Introduction

While in standard supervised learning problems we seek the best hypothesis in a given space and with a given learning algorithm, in hyperparameter optimization (Chapter 3) and meta-learning[1] (Chapter 4) we seek a configuration so that the optimized learning algorithm will produce a model that generalizes well to new data. The search space in MTL often incorporates choices associated with the hypothesis space and features of the learning algorithm itself (e.g., how optimization of the training loss is performed). Under this common perspective, both HPO and MTL essentially boil down to *nesting two search problems*: at the inner level we seek a good hypothesis (as in standard supervised learning) while at the outer level we seek a good configuration (including a good hypothesis space) where the inner search takes place. Depending on the specific setting, the outer variables take either the meaning of hyperparameters in a supervised learning problem or parameters of a meta-learner (also *meta-level parameters* or *meta-parameters*).

Surprisingly, before [Franceschi et al., 2018a] the literature on MTL had little overlap with the literature on HPO. One reason for this "historical" lack of intersection might be that these two fields typically differ substantially in terms of the experimental settings in which they are evaluated. While in HPO the available data is associated with a single task and split into a training set (used to tune the parameters) and a validation set (used to tune the hyperparameters), in MTL we deal with an entire distributions

[1] This statement reflects a large portion of MTL algorithms. In effect, we reckon that several pool-based methods, reviewed in Section 4.4.2.1, follow a quite different conceptual approach.

of related tasks and seek for generalization at an algorithmic level. A particularly interesting MTL setting is that of few-shot learning, where data comes in the form of short episodes (small datasets with few examples per class) sampled from a common probability distribution over supervised tasks. As we saw in the previous chapter, the relatively recent practice of constructing meta-datasets as in (4.2) (which contributes linking MTL to HPO) emerged precisely in this context.

A second reason may be detected in the different scopes of the prevailing methods to tackle HPO and MTL problems. Indeed, classic approaches to HPO have been only able to manage a relatively small number of hyperparameters, from a few dozens using random search (Section 3.4.3) to a few hundreds using Bayesian or model-based optimization (Sections 3.4.4 and 3.4.5). Yet, early works on MTL [Hinton and Plaut, 1987, Bengio et al., 1991, Hochreiter et al., 2001] already recognized the need for adapting a large number of parameters at the meta-level to extract and incorporate knowledge from the meta-distribution. Thus classic HPO methods seem not suitable to tackle meta-learning problems. Recent gradient-based techniques for HPO (Section 3.5), however, have significantly increased the number of hyperparameters that can be optimized and it is now possible to tune as hyperparameters entire weight vectors associated with a neural network layer. In this way, it becomes feasible to design models that possibly have more hyperparameters than parameters, blurring the border between the two fields. Such an approach is well suited for MTL since parameters are learned from a possibly small dataset, whereas hyperparameters leverage multiple available datasets and hence, by extension, many more examples.

5.2 A bilevel optimization framework

We view HPO and MTL within the natural mathematical framework of bilevel programming, where an outer optimization problem is solved subject to the optimality of an inner optimization problem. In HPO the outer problem involves hyperparameters while the inner problem is usually the minimization of an empirical loss. In MTL the outer problem could involve a shared representation among tasks while the inner problem could concern classifiers for individual tasks. Bilevel programming [Bard, 2013] has been suggested before in machine learning in the context of kernel methods and support vector machines [Keerthi et al., 2007, Kunapuli et al., 2008], multitask learning [Flamary et al., 2014], and more recently HPO [Pedregosa, 2016], but, prior to [Franceschi et al., 2018a], never in the context of MTL. Since [Franceschi et al., 2018a], a number of works have used the proposed formulation to introduce and present algorithms and applications setting in MTL and HPO, for instance in the context of neural architecture search [Liu et al., 2019], to learn the structure of group-lasso problems [Frecon et al., 2018], to describe various MTL approaches [Hospedales et al., 2020] and software [Grefenstette et al., 2019], to learn data simulators [Behl et al., 2020] and surrogate losses [Grabocka et al., 2019], among others.

We consider bilevel optimization problems [see e.g. Colson et al., 2007] of the form

$$\min_{\lambda\in\Lambda} f(\lambda) \tag{5.1}$$

where the function $f : \Lambda \to \mathbb{R}$ is defined at $\lambda \in \Lambda$ as

$$f(\lambda) = \inf\{E(w(\lambda), \lambda) : w(\lambda) \in \arg\min_{u\in\mathcal{W}\subseteq\mathbb{R}^d} L_\lambda(u)\}. \tag{5.2}$$

We call $E : \mathcal{W} \times \Lambda \to \mathbb{R}$ the *outer objective* and, for every $\lambda \in \Lambda$, we refer to $L_\lambda : \mathbb{R}^d \to \mathbb{R}$ as the *inner objective*: then $\{L_\lambda : \lambda \in \Lambda\}$ is a class of objective functions parameterized by λ. Since, in principle, the inner problem may have multiple solutions, the infimum that appears in (5.2) assures that f is well-defined as a function of λ. Specific instances of this problem include HPO and MTL, which we discuss next. We outline in Table 5.1 the links among bilevel programming, hyperparameter optimization and meta-learning, while the cartoon in Figure 5.1 depicts a simple example of the problem (5.1)-(5.2) where both w and λ are scalars.

The primary interpretation of the inner and outer objectives is that of an empirical risk computed over an appropriate set of data, depending on the context. Thus, the program (5.1)-(5.2) relates first and foremost to learning algorithms that operate by optimizing given objectives. That is, contextually to Section 3.1, (base-level) learning

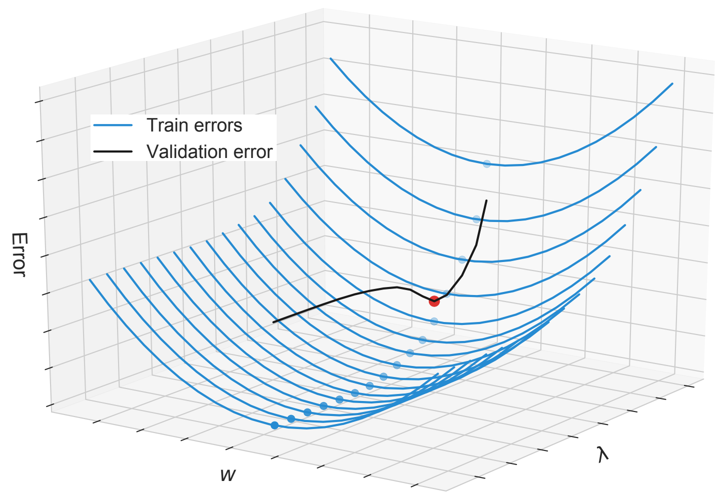

Figure 5.1: Each blue line represents the function $w \to L_\lambda(w)$ for fixed λ. The corresponding inner minimizer is shown as a blue dot. The outer objective E, evaluated at each minimizer, yields the black curve representing the function $\lambda \to f(\lambda)$, whose minimizer is shown as a red dot. The blue lines may be interpreted as the loss surfaces (training errors), while the black line corresponds to the response surface (validation error).

algorithms (defined on the weight space (3.4)) that may be written as

$$\mathcal{A}(D_{\mathrm{tr}}, \lambda) = \arg\min_{u \in \mathcal{W}} L_\lambda(u) = \mathcal{A}_\lambda(D_{\mathrm{tr}}), \tag{5.3}$$

where at the right hand side we use the notation of Section 4.3.5. This includes a large class of methods that perform empirical risk minimization, as we saw throughout the first part of the thesis. In (5.3), the definition of the hypothesis space $\mathcal{H}$ is left implicit and may depend on the value of λ, while the method with which the solution should be found is left unspecified. In fact, we note that (5.3) describes essentially an abstract algorithm: in practice, the minimization cannot be exact for all but a very few cases as the solution to the inner problem generally cannot be written analytically. One needs to resort to iterative optimization approaches as we will discuss in Section 5.3. Nevertheless, the representation (5.3) returns a compact view of the learning problem that also reflects a consistent body of literature ranging from support vector machines and kernel methods to deep learning.

Table 5.1: Links and naming conventions among different fields.

Bilevel programming	Hyperparameter optimization	Meta-learning
Inner variables	Parameters	Base-level parameters (task-specific parameters)
Outer variables	Hyperparameters	Meta-level parameters (meta-learner parameters)
Inner objective	Training error	Base-level objectives (task-specific errors)
Outer objective	Validation error	Meta-level objective (meta-training error)

5.2.1 Hyperparameter Optimization

In the context of hyperparameter optimization, we are interested in minimizing the validation error of a model $h_w : \mathcal{X} \to \mathcal{Y}$ parameterized by a vector w, with respect to a vector of hyperparameters λ (Section 3.1). For example, we may consider representation or regularization hyperparameters that control the hypothesis space or penalties, respectively. In this setting, a prototypical choice for the inner objective is the regularized empirical error

$$L_\lambda(w) = \frac{1}{|D_{\text{tr}}|} \sum_{(x,y)\in D_{\text{tr}}} \ell(h_w(x), y) + \Omega_\lambda(w) = \hat{\mathcal{E}}(h_w, D_{\text{tr}}) + \Omega_\lambda(w), \tag{5.4}$$

where $D_{\text{tr}} = \{(x_i, y_i)\}_{i=1}^N$ is a set of input/output points, ℓ is a prescribed loss function, and Ω_λ a regularizer parameterized by λ. The outer objective represents a proxy for the generalization error of h_w, and it may be given by the average loss on a validation set D_{val}

$$E(w, \lambda) = \frac{1}{|D_{\text{val}}|} \sum_{(x,y)\in D_{\text{val}}} \ell(h_w(x), y) = \hat{\mathcal{E}}(h_w, D_{\text{val}}). \tag{5.5}$$

or, in more generality, by a cross-validation error, as detailed in Appendix B. It is important to note that, in this setting, the outer objective E does not depend explicitly on the hyperparameters λ, but only implicitly through the solution $w(\lambda)$ on which E is computed (cf. Equation (3.5), considering $\mathcal{A}$'s of the type (5.3)). This is because in HPO λ is instrumental in finding a good model h_w, which is our final goal.

As a more specific example, consider linear models, $h_w(x) = \langle w, x\rangle$, let ℓ be the square loss and let $\Omega_\lambda(w) = \lambda\|w\|^2$, in which case the inner objective is ridge regression (Tikhonov regularization) and the bilevel problem optimizes the validation error of the ridge regressor over the regularization parameter.

5.2.2 Meta-Learning

In meta-learning the inner and outer objectives may be computed by averaging a training and a test (validation) error over multiple tasks (Section 4.2). The goal is to produce a learning algorithm that will work well on novel tasks. For this purpose, considering the typical supervised meta-learning case, we have available a meta-training set $\mathbf{D} = \left\{D^j = \left(D^j_{\text{tr}}, D^j_{\text{ts}}\right)\right\}_{j=1}^{\mathbf{N}}$, which is a collection of datasets sampled from a meta-distribution $p_{D,\mathcal{T}}$. Each dataset $D^j = \left\{\left(x^j_i, y^j_i\right)\right\}_{i=1}^{N^j}$ with $\left(x^j_i, y^j_i\right) \in \mathcal{X}^j \times \mathcal{Y}^j$ and $N^j = N^j_{\text{tr}} + N^j_{\text{ts}}$ is linked to a specific task. Recall that the training sets are sampled directly from a distribution over data (but jointly with the tasks), while the test sets are sampled (i.i.d.) form the task distributions; hence, generally $N^j_{\text{tr}} \neq N^j_{\text{ts}}$ (where N^j_{tr} and N^j_{ts} denote the number of training and testing examples, respectively). Note that the input and output spaces may be task dependent (e.g. a multi-class classification problem with variable number of classes).

The model for each task is a function $h_{w^j,\lambda} : \mathcal{X}^j \to \mathcal{Y}^j$, identified by a parameter vectors w^j and hyperparameters (or meta-level parameters) λ. We note that also the functional (structural) component of the base-level hypothesis $h_{w^j,\lambda}$ may vary depending on the task (e.g. neural networks with different structures), but we leave this fact implicit, to simplify the notation. A key point here is that λ is shared between base-level tasks. Denoting by $w = (w^j)_{j=1}^{\mathbf{N}}$ the concatenation of all the task-specific weight vectors, the inner and outer objectives are

$$L_\lambda(w) = \frac{1}{\mathbf{N}} \sum_{j=1}^{\mathbf{N}} L^j(w^j, \lambda, D^j_{\text{tr}}) = \frac{1}{\mathbf{N}} \sum_{j=1}^{\mathbf{N}} \hat{\mathcal{E}}(h_{w^j,\lambda}, D^j_{\text{tr}}), \tag{5.6}$$

$$E(w, \lambda) = \frac{1}{\mathbf{N}} \sum_{j=1}^{\mathbf{N}} L^j(w^j, \lambda, D^j_{\text{ts}}) = \frac{1}{\mathbf{N}} \sum_{j=1}^{\mathbf{N}} \hat{\mathcal{E}}(h_{w^j,\lambda}, D^j_{\text{ts}}). \tag{5.7}$$

The loss $L^j(w^j, \lambda, S)$ represents the empirical error of the pair (w^j, λ) on a set of examples S. Note that the inner and outer losses for task j use different train/test splits of the corresponding dataset D^j. Furthermore, unlike in HPO, in MTL the final goal is to find a good λ and the w^j are now instrumental.

If, again, we consider the base-level algorithm $\mathcal{A}_\lambda$ as the minimization of the the task-specific loss L^j as in (5.3), then we see that[2]

$$f(\lambda) = \hat{\mathcal{E}}(\mathcal{A}_\lambda, \mathbf{D}),$$

which is the empirical risk of the base-level learning algorithm $\mathcal{A}_\lambda$ (indexed by $\lambda \in \Lambda$) as defined in (4.4).

We can interpret the cartoon in Figure 5.1 as an MTL problem. The parameter λ indexes a family of hypothesis spaces within which the inner objective is minimized (the blue lines, with the dots as minimizers). At the meta-level (the black line), one seeks for the configuration λ (the red dot) that minimizes the average test error over the available datasets in $\mathbf{D}$. A particular example, detailed in Section 7.2.1, is to choose the model $h_{w^j,\lambda} = \langle w^j, r_\lambda(x)\rangle$, in which case λ parameterizes a feature mapping r_λ (e.g. a deep neural network). Yet another choice would be to consider $h_{w^j,\lambda}(x) = \langle w^j + \lambda, x\rangle$, in which case λ represents a common model around which task specific models are to be found [see e.g. Evgeniou et al., 2005, Finn et al., 2017, Khosla et al., 2012, Kuzborskij et al., 2013, and reference therein].

The nested structure of the resulting bilevel problem captures many of the algorithmic meta-learning strategies discussed in Section 4.4.2. It is, however, typically not quite as suitable at describing model-based strategies[3] (Section 4.4.1) and two stage-approaches [see e.g. Wang et al., 2019, Tian et al., 2020]. It is also superfluous for approaches whose base-level algorithms have an analytical (closed-form) expression, such as nearest neighbour classification [e.g. Vinyals et al., 2016, Snell et al., 2017], ridge [Bertinetto et al., 2019] or Gaussian process [Patacchiola et al., 2020] regression. In all these cases, indeed, the resulting problem can be reduced to a single level one.

[2] Assuming a unique minimizer of L_λ.

[3] Although, as we shall shortly see in Section 5.3.1, the iterative view offers a conceptual link to many model-based strategies.

5.3 The Gradient-Based Approach

We now discuss a general approach to solve Problem (5.1)-(5.2) when the hyperparameter (or meta-level parameter) vector λ is real-valued and $\Lambda \subset \mathbb{R}^m$. In principle, the outer objective could be optimized with a number of techniques that do not require computing a gradient, some of which we reviewed in the context of HPO in Section 3.4. However, when applicable, gradient-based optimization, has a number of advantages, both practical and theoretical that make it appealing (see Section 2.4), especially when the dimensionality of λ grows.

To simplify our discussion let us assume that $\mathcal{W} = \mathbb{R}^d$ (unconstrained optimization at the inner level) and that the inner objective has a unique minimizer w_λ. Even in this simplified scenario, Problem (5.1)-(5.2) remains challenging to solve. Indeed, in general there is no closed form expression for $w(\lambda)$, so it is not possible to directly optimize the outer objective function, nor compute ∇f. A possible strategy is to replace the inner problem with the first order optimality condition $\nabla L_\lambda = 0$ and apply the implicit function theorem [Pedregosa, 2016, Koh and Liang, 2017, Beirami et al., 2017, Lorraine et al., 2020]. However, one still remains with the problem of approximating the solution: the implicit equation $\nabla L_\lambda = 0$ can be only approximately satisfied. We return to this point (implicit differentiation) in Section 5.4.3, where we present some practical algorithms to compute an approximate gradient of f in this way.

Another compelling approach, on which we focus particularly in this thesis, is to replace the inner problem with a dynamical system. This point, discussed in [Domke, 2012, Maclaurin et al., 2015a, Franceschi et al., 2017, 2018a], allows us to compute an exact gradient (up to numerical errors) of an approximation of Problem (5.1)-(5.2). It also reflects the practical implementation of many learning algorithms, thereby making it possible to optimize variables (hyperparameter or meta-level parameters) that are associated to the way in which the search in the hypothesis space is carried out, i.e. that define the learning dynamics itself.

5.3.1 The Iterative View

Specifically, we let $[T] = \{1, \dots, T\}$ where T is a prescribed positive integer and consider the following Problem

$$\min_{\lambda} f_T(\lambda) = E(w_T(\lambda), \lambda), \tag{5.8}$$

where E is a smooth scalar function[4], and

$$s_0(\lambda) = \Phi_0(\lambda); \quad s_t(\lambda) = \Phi_t(s_{t-1}(\lambda), \lambda), \quad t \in [T], \tag{5.9}$$

where $s_t = (w_t, v_t) \in \mathbb{R}^{d'}$ is the state of the dynamical system, that potentially includes accessory variables v_i (Section 2.4.5). $\Phi_0 : \mathbb{R}^m \to \mathbb{R}^{d'}$ is a smooth initialization mapping and, for every $t \in [T]$, $\Phi_t : \mathbb{R}^{d'} \times \Lambda \subset \mathbb{R}^m \to \mathbb{R}^{d'}$ is a smooth update mapping. It typically represents the operation performed by the t-th step of an optimization algorithm, where the index t may represent stochastic evaluations of L. In this case – when the learning dynamics *is* an optimization dynamics – (5.8)-(5.9) may be seen as an approximation of Problem (5.1)-(5.2). We return on this topic in the next chapter, where discuss the approximation proprieties of (5.8)-(5.9). A major advantage of this reformulation is that it makes it possible to compute efficiently the gradient of f_T, the hypergradient, either in time or in memory, by making use of reverse or forward mode algorithmic differentiation (Sections 5.4.1 and 5.4.2). To simplify the notation, from now on we will often leave the dependency from λ implicit when denoting the state.

The optimization dynamics could be gradient descent, in which case $s_t = w_t$ (no auxiliary variables) and $\Phi_t(w_t, \lambda) = w_t - \eta_t \nabla L_\lambda(w_t)$ where $(\eta_t)_{t \in [T]}$ is a sequence of steps sizes. Another simple example of Φ_t occurs when training a neural network by gradient descent with momentum (Section 2.4.3), in which case

$$\begin{aligned} v_t &= \mu v_{t-1} + \nabla L_{\lambda,t}(w_{t-1}) \\ w_t &= w_{t-1} - \eta(\mu v_{t-1} - \nabla L_{\lambda,t}(w_{t-1}) \end{aligned} \tag{5.10}$$

where L_t is the objective associated with the t-th mini-batch, μ is the momentum factor

[4] We are not aware of outer objective functions that depend also on the auxiliary variables. Hence, we directly consider E as a function of the model variables only.

and η is (a fixed) learning rate. The initialization mapping could be given by

$$(w_0, v_0) = (\varepsilon, 0), \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2 I) \tag{5.11}$$

where the initial weights w_0 are sampled from a normal distribution with covariance matrix $\sigma^2 I$. In this example, λ could comprise the (optimization) hyperparameters (μ, η, σ).

In MTL, referring to Section 3.4, [Andrychowicz et al., 2016, Wichrowska et al., 2017] consider a mapping Φ_t implemented as a recurrent neural network (learning to optimize). The meta-level parameters λ are then the weights of the model. MAML [Finn et al., 2017] (which uses gradient descent as optimization routine) focuses on the initialization mapping by letting $\Phi_0(\lambda) = \lambda$. Note that these latter examples could not be easily expressed with the implicit view, although for MAML there exist closely related formulations (e.g. Rajeswaran et al. [2019] leverage the similarity of the effects of initialization in conjunction with early stopping with biased L^2 regularization).

The iterative view, where Φ_t represents an optimization dynamics, covers most of the parametric algorithmic strategies for MTL outlined in (4.4.2.3). The dynamics, however, need not be necessarily tied to an underlying optimization procedure. It can, for instance, relate to the extraction of representations, whereby the variables w_t are no longer interpreted as weights of underlying models, but rather as their internal representations. This perspective departs from the bilevel problem (5.1)-(5.2), but offers an interesting link to other learning and meta-learning algorithms. A very simple (and somewhat degenerate) instance is given by feed-forward neural networks (2.10), where one may identify the transformation at the t-th layer with Φ_t. The hidden units z_t instead identify with the weights w_t and the total iterations T with the depth of the network. Another more interesting example is given by the so-called equilibrium models (EQM) [Grazzi et al., 2020] that comprise, e.g., stable recurrent neural networks [Miller and Hardt, 2019], graph neural networks [Scarselli et al., 2009] and the formulations of Deep Equilibrium Networks by Bai et al. [2019] (see also experiments in Section 6.3.4). In this case, the internal representations are given by fixed points of learnable dynamics rather than compositions of a finite number of layers. The EQM learning problem relates to a more general formulation of the bilevel program of Section 5.2 that consist in replacing the inner problem with a fixed-point equation. We embrace this view in Section 6.3, and report in Section 6.3.4 a series of experiments with an instanciation of an EQM model.

In particular, this latter "broader" view points toward model-based (Section 4.4.1) and instance-based (Section 4.4.2.2) MTL techniques. For instance, Φ_t may be given by the dataset-encoding dynamics 4.7, where T would represent the number of datapoints in a given training set.

5.4 Hypergradient Computation

In this section, based on [Franceschi et al., 2017] and [Grazzi et al., 2020], we present various algorithms to compute an approximate gradient of the outer objective, that may then be plugged into a gradient descent procedure to optimize λ (Equation (3.27)). The iterative view of Section 5.3.1 gives rise to two possible ways, rooted in algorithmic differentiation (see Chapter A), to compute the hypergradient which have different trade-offs in terms of running time and space requirements. The reverse-mode (Section 5.4.1) efficient in time, is based on a Lagrangian formulation associated with the parameter optimization dynamics. It encompasses the reverse-mode approach presented in [Maclaurin et al., 2015a], where the dynamics corresponds to stochastic gradient descent with momentum. We do not assume reversible parameter optimization dynamics. A well-known drawback of reverse-mode differentiation is its space complexity: we need to store the whole trajectory of training iterates in order to compute the hypergradient. An alternative approach, first introduced in the HPO context in [Franceschi et al., 2017], is to compute the hypergradient in forward-mode (Section 5.4.2) and it is efficient in memory. These two approaches have a direct correspondence with two classic alternative ways of computing gradients for recurrent neural networks (RNN) [Pearlmutter, 1995]: the Lagrangian (reverse) way corresponds to back-propagation through time [Werbos, 1990], while the forward way corresponds to real-time recurrent learning (RTRL) [Williams and Zipser, 1989]. As RTRL allows one to update parameters after each time step, the forward approach is suitable for real-time hyperparameter updates, which may significantly speed up the overall hyperparameter optimization procedure in the presence of large datasets. We sketch an elementary version of the real-time procedure based on forward-mode in Section 5.4.2.1, and refer to Chapter 8 for a thorough discussion of a refined version of the algorithm.

Finally, in Section 5.4.3 we discuss implicit differentiation schemes [Pedregosa, 2016, Rajeswaran et al., 2019, Lorraine et al., 2020]. First, an (implicit) equation for $\nabla f(\lambda)$ is obtained through the implicit function theorem. Then, this equation is approximately solved by using a two-stage scheme. We present two specific implementation in this class: the *fixed-point method* [Lorraine et al., 2020] and the *conjugate gradient method* [Pedregosa, 2016].

5.4.1 Reverse-Mode

The reverse-mode computation leads to an algorithm closely related to the one presented in [Maclaurin et al., 2015a]. A major difference with respect to their work is that we do not require the mappings Φ_t defined in Equation (5.9) to be invertible. With respect to [Franceschi et al., 2017], we extend here the computation to include also the initialization mapping and the possible direct dependence of the outer objective from the hyperparameter vector λ. We note that the reverse-mode calculation is structurally identical to back-propagation through time [Werbos, 1990].

We consider the constraint optimization problem that arise from the iterative view

$$\begin{aligned} &\min_{\lambda, s} E(w_T, \lambda) = f_T(\lambda) \\ &\text{subject to} \quad s_0 = \Phi_0(\lambda), \quad s_t = \Phi_t(s_{t-1}, \lambda),\ t \in [T]. \end{aligned} \tag{5.12}$$

This formulation closely follows a classical Lagrangian approach used to derive the back-propagation algorithm [LeCun, 1988]. The vector $s = (s_t)_{t=0}^T$ is considered as a free variable, although it is uniquely determined by the dynamics.

The Lagrangian of problem (5.12) is

$$\mathcal{L}(s, \lambda, \alpha) = E(w_T, \lambda) + \sum_{t=1}^{T} \alpha_t(\Phi_t(s_{t-1}, \lambda) - s_t) + \alpha_0(\Phi_0(\lambda) - s_0)$$

where, for each $t \in \{0,\dots,T\}$, $\alpha_t \in \mathbb{R}^d$ is a row vector of Lagrange multipliers associated with the t-th step of the dynamics. We define, for every $t \in \{1,\dots,T\}$, the matrices

$$A_t = \frac{\partial \Phi_t(s_{t-1},\lambda)}{\partial s_{t-1}} \in \mathbb{R}^{d'\times d'}, \;\; B_t = \frac{\partial \Phi_t(s_{t-1},\lambda)}{\partial \lambda} \in \mathbb{R}^{d'\times m}. \tag{5.13}$$

The partial derivatives of the Lagrangian are given by

$$\frac{\partial \mathcal{L}}{\partial \alpha_0} = \Phi_0(\lambda) - s_0, \qquad \frac{\partial \mathcal{L}}{\partial \alpha_t} = \Phi_t(s_{t-1},\lambda) - s_t, \quad t \in [T] \tag{5.14}$$

$$\frac{\partial \mathcal{L}}{\partial s_t} = \alpha_{t+1}A_{t+1} - \alpha_t, \quad t \in \{0,\dots,T-1\} \tag{5.15}$$

$$\frac{\partial \mathcal{L}}{\partial s_T} = \frac{\partial E(w_T,\lambda)}{\partial s_T} - \alpha_T \tag{5.16}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \sum_{t=1}^{T} \alpha_t B_t + \frac{\partial E(w_T,\lambda)}{\partial \lambda} + \alpha_0 \frac{\partial \Phi_0(\lambda)}{\partial \lambda}, \tag{5.17}$$

where $\partial_s E(w,\lambda) = (\partial_w E(w,\lambda), 0) \in \mathbb{R}^{1\times d'}$, where 0 is a row vector with $d' - d$ entries.

The optimality conditions are then obtained by setting each derivative to zero. In particular, setting the right hand side of Equations (5.15) and (5.16) to zero gives

$$\alpha_t = \begin{cases} \partial_{s_T} E(w_T,\lambda) & \text{if } t = T, \\ \nabla \partial_{s_T} E(w_T,\lambda) A_T \cdots A_{t+1} & \text{if } t \in \{0,\dots,T-1\}. \end{cases} \tag{5.18}$$

Combining these equations with Eq. (5.17) we obtain that

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \partial_\lambda E(w_T,\lambda) + \partial_{s_T} E(w_T,\lambda) \sum_{t=0}^{T} \left(\prod_{s=t+1}^{T} A_s \right) B_t,$$

where we have set $B_0 = \partial_\lambda \Phi_0(\lambda)$. As we shall see this coincides with the expression for the differential of f_T (whose transposed is the gradient) in Eq. (5.22) derived in the next section. Pseudo-code of `Reverse-HG` is presented in Algorithm 2.

If an hyperparameters appears as a parameter of a probability distribution, such as the case of a random normal initialization of the neural network weight (5.11), one can compute an estimator of the differential using a smooth reparameterization, if it exists [Mohamed et al., 2020]. This procedure is also known as reparameterization trick

Algorithm 2 `Reverse-HG`

```
Input: $\lambda$ current values of the hyper-
parameters, dynamics $\{\Phi_t\}_{t=0}^{T}$
Output: Gradient of $f_T$
$S_0 \leftarrow \Phi_0(\lambda)$
for $t = 1$ to $T$ do
    $s_t \leftarrow \Phi_t(s_{t-1}, \lambda)$
end for
$\alpha_T \leftarrow \partial_{s_T} E(w_T, \lambda)$
$g \leftarrow \partial_\lambda E(w_T, \lambda)$
for $t = T - 1$ downto $0$ do
    $g \leftarrow g + \alpha_{t+1} B_{t+1}$
    $\alpha_t \leftarrow \alpha_{t+1} A_{t+1}$
end for
return $[g + \alpha_0 B_0]^\intercal$
```

[Kingma and Welling, 2014]. For the simple case of (5.11), where $w_0 = \Phi_0(\lambda) = \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, \sigma^2 I)$, assuming that the only hyperparamter is $\lambda = \sigma$ and that the dynamics is gradient descent, one can rewrite $\Phi_0(\lambda) = \sigma\varepsilon$ for $\varepsilon \sim \mathcal{N}(0, I)$ so that the estimator of the differential is computed as

$$\partial_\lambda \Phi_0(\lambda) = \varepsilon \in \mathbb{R}^d \qquad \varepsilon \sim \mathcal{N}(0, I)$$

Hence, the hypergradient of f with respect to $\lambda = \sigma$ would be given by the random variable

$$\nabla f_T(\lambda) = \alpha_0 \varepsilon \in \mathbb{R} \qquad \varepsilon \sim \mathcal{N}(0, I),$$

that is the scalar product between the vector of the "initial" adjoints α_0 and a vector ε drawn from the standard normal distribution. If no smooth reparameterization exists (e.g. for discrete random variables), one need to resort to other techniques. We will discuss this case, presenting an implementation that uses the so-called straight-through gradient estimator [Bengio et al., 2013], in Chapter 9 for optimizing edges of a graph (interpreted as hyperparameters).

5.4.2 Forward-Mode

The second approach to compute the hypergradient appeals to the chain rule for the (total) derivative of composite functions with implicit dependencies, to obtain that the gradient of f_T at λ, which satisfies[5]

$$[\nabla f_T(\lambda)]^\intercal = \partial_\lambda f_T(\lambda) = \partial_\lambda E(w_T,\lambda) + \partial_{s_T} E(w_T,\lambda)\frac{ds_T}{d\lambda} \tag{5.19}$$

where $\frac{ds_T}{d\lambda}$ is the $d' \times m$ matrix formed by the total derivative of the components of s_T with respect to the components of λ. This time s_t is viewed as a function of λ for each iteration t.

Recall that $s_t = \Phi_t(s_{t-1},\lambda)$. The operators Φ_t depend on the hyperparameter λ both directly by its expression and indirectly through the state s_{t-1}. Using again the chain rule we have that, for every $t \in \{1,\ldots,T\}$,

$$\frac{ds_0}{d\lambda} = \frac{\partial\Phi_0(\lambda)}{\partial\lambda}; \qquad \frac{ds_t}{d\lambda} = \frac{\partial\Phi_t(s_{t-1},\lambda)}{\partial s_{t-1}}\frac{ds_{t-1}}{d\lambda} + \frac{\partial\Phi_t(s_{t-1},\lambda)}{\partial\lambda}. \tag{5.20}$$

Defining $Z_t = \frac{ds_t}{d\lambda}$ for every $t \in \{0,\ldots,T\}$ and recalling Equation (5.13), we can rewrite Eq. (5.20) as the recursion

$$Z_0 = \partial_\lambda\Phi_0(\lambda) = B_0, \qquad Z_t = A_t Z_{t-1} + B_t, \quad t \in [T]. \tag{5.21}$$

Using Eq. (5.21), we obtain that

$$\begin{aligned}
[\nabla f_T(\lambda)]^\intercal &= \partial_\lambda E(w_T,\lambda) + \partial_{s_T} E(w_T,\lambda) Z_T \\
&= \partial_\lambda E(w_T,\lambda) + \partial_{s_T} E(w_T,\lambda)(A_T Z_{T-1} + B_T) \\
&= \partial_\lambda E(w_T,\lambda) + \partial_{s_T} E(w_T,\lambda)(A_T A_{T-1} Z_{T-2} + A_T B_{T-1} + B_T) \\
&\;\;\vdots \\
&= \partial_\lambda E(w_T,\lambda) + \partial_{s_T} E(w_T,\lambda) \sum_{t=0}^{T}\left(\prod_{s=t+1}^{T} A_s\right) B_t.
\end{aligned} \tag{5.22}$$

[5]The gradient of a scalar function is a column vector, while its differential (that coincides with its Jacobian) is a row vector.

Note that the recurrence (5.21) on the Jacobian matrix is structurally identical to the recurrence in the RTRL procedure described in [Williams and Zipser, 1989, eq. (2.10)].

From the above derivation it is apparent that $\nabla f_T(\lambda)$ can be computed by an iterative algorithm which runs in parallel to the training algorithm. Pseudo-code of `Forward-HG` is presented in Algorithm 3. At first sight, the computation of the terms in the right hand side of Eq. (5.21) seems prohibitive. However, in Section 6.1 we observe that if m is much smaller than d, the computation can be done efficiently.

Algorithm 3 `Forward-HG`

```
Input: $\lambda$ current values of the hyper-parameters, dynamics $\{\Phi_t\}_{t=0}^T$
Output: Gradient of $f_T$ w.r.t. $\lambda$
$s_0 \leftarrow \Phi_0(\lambda)$
$Z_0 \leftarrow B_0$
for $t = 1$ to $T$ do
    $s_t \leftarrow \Phi_t(s_{t-1}, \lambda)$
    $Z_t \leftarrow A_t Z_{t-1} + B_t$
end for
return $[\partial_\lambda E(w_T, \lambda) + \partial_{s_T} E(w_T, \lambda) Z_T]^\intercal$
```

5.4.2.1 Real-Time Forward-Mode

For every $t \in \{1, \ldots, T\}$ let $f_t : \mathbb{R}^m \to \mathbb{R}$ be the response function at time t: $f_t(\lambda) = E(w_t(\lambda), \lambda)$. Note that f_t for $t = T$ coincides with the definition of the response function in Eq. (5.8). A major difference between `Reverse-HG` and `Forward-HG` is that the *partial* hypergradients

$$[\nabla f_t(\lambda)]^\intercal = \frac{dE(w_t, \lambda)}{d\lambda} = \partial_\lambda E(w_t, \lambda) + \partial_{s_t} E(w_t, \lambda) Z_t \tag{5.23}$$

are available in the second procedure at each time step t and not only at the end.

The availability of partial hypergradients is significant since we are allowed to update hyperparameters several times in a single optimization epoch, without having to wait until time T. This is reminiscent of the real-time updates suggested by Williams and Zipser [1989] for RTRL. The real-time approach may be suitable in the case of a data stream (i.e. $T = \infty$), where `Reverse-HG` would be hardly applicable. Even in the case of finite (but large) datasets it is possible to perform one hyperparameter update after a hyper-batch of data (i.e. a set of minibatches) has been processed. Algorithm 3 can be easily modified to yield a partial hypergradient when $t \mod \Delta = 0$ (for some hyper-batch size Δ) and letting t run from 1 to ∞, reusing examples in a circular or random way. This is particularly meaningful when the outer objective does not depend explicitly from the hyperparameter vector λ; that is $E(w_t(\lambda), \lambda) = E(w_t(\lambda))$, as it is typical in HPO: in effect, computing the "direct gradient" $\partial_\lambda E(w_t) = 0$ (i.e. without considering the implicit dependence of s_t) would not yeield any update direction at all. We use this strategy (which we further develop in Chapter 8) in a phoneme recognition experiment for tuning critical hyperparameters of a large feed-forward neural network in Section 7.1.2.

5.4.3 Implicit Differentiation

In this section we present derivations and algorithms that arise from the implicit view of the bilevel program (5.1)-(5.2). We change here the notation for the inner objective to $L(w, \lambda) = L_\lambda(w)$ to improve readability. As mentioned in Section 5.3, the implicit approach involves replacing the inner problem with the first order optimality condition

$$\nabla_w L(w, \lambda) = 0. \tag{5.24}$$

Under some conditions which we will specify in Section 6.3, the (exact) gradient of f is given by

$$[\nabla f(\lambda)]^\top = \partial_\lambda f(\lambda) = \partial_\lambda E(w(\lambda), \lambda) + \partial_w E(w(\lambda), \lambda) \frac{dw(\lambda)}{d\lambda}. \tag{5.25}$$

By applying the implicit function theorem [Krantz and Parks, 2012] to the implicit equation given by (5.24) one obtains

$$\frac{dw(\lambda)}{d\lambda} = -\frac{\partial^2 L(w(\lambda), \lambda)}{\partial w^2}^{-1} \frac{\partial^2 L(w(\lambda), \lambda)}{\partial w \partial \lambda} \tag{5.26}$$

provided that the Hessian of the inner objective (w.r.t. w) is invertible[6] on the solution surface $w(\lambda)$ for $\lambda \in \Lambda$. However, typically one does not have access to an analytic expression of $w(\lambda)$ and thus must resort to approximations. Let us denote by w_T an approximate solution of the inner problem (e.g. found with an iterative algorithm). Then, after substituting $w(\lambda)$ with w_T and plugging (5.26) into (5.25), one can set $q = -\partial_w E \left[\partial^2 L\right]^{-2} \in \mathbb{R}^d$ and (approximately) solve the linear system (organized by rows)

$$q \frac{\partial^2 L(w_T, \lambda)}{\partial w^2} = -\partial_w E(w_T, \lambda). \tag{5.27}$$

Call $q_{T,K}$ an approximate solution of (5.27). Then an approximate gradient of the outer objective is given by

$$[\nabla f_{T,K}(\lambda)]^\intercal = \partial_\lambda E(w_T, \lambda) + q_{T,K} \partial^2_{w,\lambda} L(w_T, \lambda). \tag{5.28}$$

Hence the two possible sources of approximations stem from the solution of the inner problem and of the linear system (5.27). Particular implementation of this computation scheme depend on how (5.27) is solved; one classic procedure is the conjugate gradient (CG) method. It features a linear rate of convergence that depends on the conditioning number of the Hessian of the inner objective.

Another possible approach, somewhat linked to the iterative view, is given by the so-called fixed-point method. It is based on the observation that one can rewrite the condition (5.24) as a fixed point equation of an appropriate (contractive) dynamics[7]

$$w(\lambda) = \Phi(w(\lambda), \lambda),$$

which could be, for instance, gradient descent $\Phi(w, \lambda) = w - \eta \nabla L$. Then, the application

[6] In fact, one would only need local invertibility as the implicit function theorem is a local result. For simplicity we consider here the global case, which also allows us to derive convergence proprieties of the hypergradient approximation of the implicit differentiation procedures. See Chapter 6 for details.

[7] As in the iterative case, the dynamics may be defined on an augmented state, for example when using gradient descent with momentum.

of the implicit function theorem yields

$$\frac{dw(\lambda)}{d\lambda} = \left(I - \frac{\partial\Phi(w(\lambda),\lambda)}{\partial w}\right)^{-1} \frac{\partial\Phi(w(\lambda),\lambda)}{\partial w \partial\lambda}$$

and, repeating the procedure described above, Equation (5.27) becomes

$$q(I - \partial_w \Phi(w_T, \lambda)) = \partial_w E(w_T, \lambda). \tag{5.29}$$

Starting from any point $q_{T,0} \in \mathbb{R}^d$, one can iterate

$$q_{T,k} = q_{T,k-1}\partial_w \Phi(w_T, \lambda) + \partial_w E(w_T, \lambda) \qquad k \in [K] \tag{5.30}$$

to find an approximate fixed-point of (5.29) (which is an approximate solution of the linear system). Warm-starting $q_{T,0}$ with the vector obtained at the previous optimization iteration (of λ) may speed-up the convergence of (5.30). Algorithm 4 lists the resulting procedure. We note that this computation method may be used also in the more general case when Φ is not necessarily an optimization dynamics: Φ may represent a stable recurrent neural network [Miller and Hardt, 2019], a graph neural network [Scarselli et al., 2009] or an equilibrium model (Section 6.3.4). In this cases, the fixed-point algorithm is usually known as recurrent backpropagation [Almeida, 1987, Liao et al., 2018].

Algorithm 4 `Fixed-point-HG`

```
Input: $\lambda$ current values of the hyperparameters, (fixed-point) dynamics $\Phi$, approximate minimizer of inner objective $w_T$, number of iterations $K$, initial value $q_0$
Output: Approximate gradient of $f$
for $k = 1$ to $K$ do
    $q_{T,k} \leftarrow q_{T,k-1}\partial_w \Phi(w_T, \lambda) + \partial_w E(w_T, \lambda)$
end for
return $[\partial_\lambda E(w_T, \lambda) + q_{T,K}\partial_\lambda \Phi(w_T, \lambda)]^\intercal$
```

Implicit differentiation methods are "oblivious" to the way the approximate solution w_T has been found. This means that they have a computational advantage in memory over the reverse-mode differentiation as they do not require storing the entire dynamics. If implemented correctly, using algorithmic differentiation, they have a run-time advantage over forward-mode differentiation since they only deal with vector quantities (the $q_{T,K}$'s). However, for this very reason, they do not directly allow computing the gradient of hyperparameters related to the optimization of the inner problem, such as the initialization mapping or the learning rate. Furthermore, as we shall see in the next two chapters, they are also quite sensitive to the proprieties of the inner problem (e.g. for the fixed-point method, the dynamics must be a contraction), and therefore are applicable to a narrower class of problems.

5.5 Interim Summary

In this chapter, we have shown that both HPO and (a large portion of) MTL can be formulated in terms of bilevel programming, which allows us to compactly express several hyperparameter optimization problems and meta-learning algorithms. Our framework encompasses recently proposed methods for meta-learning, such as MAML and learning to optimize, but also suggests different design patterns for the inner learning algorithm which are the subject of ongoing studies and open up several routes for future research. We have indicated two main paths to reformulate the bilevel problem that lead to a series of algorithms to efficiently compute an approximate gradient of the outer objective. In particular, we argued that the iterative approach reflects more closely practical implementations of the underlying (base-level) learning algorithms. It further supports natural extensions that allow considering hyperparameters (or meta-parameters) related to how the algorithm searches for hypotheses (typically by optimization).

In the last section, we derived and presented two algorithms that arise from the iterative view. In this context, before [Franceschi et al., 2017], previous work has mainly focused on the reverse-mode computation, attempting to deal with its space complexity, that becomes prohibitive for very large models such as deep networks. Forward-mode differentiation, especially the real-time counterpart that we will study more closely in Chapter 8, constitute a compelling alternative for optimizing few critical hyperparameters, as we shall see in experiments of Section 7.1.2. We also derived algorithms that arise from the implicit view, presenting two particular implementations. All these algorithms have different trade-offs in terms of computational complexity and applicability, which we will analyze both theoretically and empirically in the next two chapters.

Chapter 6

Analysis of the Framework

In this chapter we report some analytical result concerning the algorithms presented in the previous section, focusing on the iterative procedures. We start by providing a complexity analysis in Section 6.1 that reveals the trade-off between forward and reverse mode, and establish the computational complexity of the fixed-point method.

The iterative procedure (5.8)-(5.9) raise the issue of the quality of the approximation to Problem (5.1)-(5.2). We provide sufficient conditions for assuring that the set of minimizers of f_T converges to the set of minimizers of f, first reported in [Franceschi et al., 2018a]. We observe that these conditions are reasonable and apply to concrete problems relevant to applications. We conclude the section with two experiments regarding the effect of tuning some hyperparameters of the optimization dynamics and the selection of the total number of iterations (the horizon) for iterative methods.

Next, we turn our attention to the convergence of the (approximate) hypergradients to the gradient of the outer objective ∇f, considering an extension of the bilevel program of Section 5.2, where the inner problem is given by a fixed-point equation. We provide iteration complexity results for iterative differentiation and mention those concerning implicit methods, when the mapping defining the fixed point equation is a contraction. In particular, we prove non-asymptotic linear rates for the approximation errors of both approaches. These results have appeared very recently in [Grazzi et al., 2020]. We finally investigate empirically the impact of the contractiveness hypothesis on the effectiveness of the iterative and the implicit methods in Section 6.3.4, considering as case study an implementation of the so-called equilibrium models, neural networks which use as internal representation fixed-points of a learnable dynamics.

6.1 Complexity Analysis

We discuss the time and space complexity of Algorithms 2, 3 and 4. The computational complexity of the implicit differentiation procedure with conjugate gradient is the same of that of Algorithm 4. We begin by recalling some basic results from the algorithmic differentiation literature. We refer the reader to Chapter A for an introduction to the topic. In particular, Section A.4 derives computational complexity bounds for the two modes of algorithmic differentiation.

Let $F : \mathbb{R}^n \mapsto \mathbb{R}^p$ be a differentiable function and suppose it can be evaluated in time $\mathtt{t}(n, p)$ and requires space $\mathtt{m}(n, p)$. Denote by $\mathrm{D}F$ the $p \times n$ Jacobian matrix of F. Then the following facts hold true [Griewank and Walther, 2008] (see also [Baydin et al., 2018b] for a shorter account):

(i) For any vector $r \in \mathbb{R}^n$, the product $\mathrm{D}Fr$ can be evaluated in time $O(\mathtt{t}(n, p))$ and requires space $O(\mathtt{m}(n, p))$ using forward-mode AD.

(ii) For any row vector $q \in \mathbb{R}^{1\times p}$, the product $q\mathrm{D}F$ has time and space complexities $O(\mathtt{t}(n, p))$ using reverse-mode AD.

(iii) As a corollary of item (i), the whole $\mathrm{D}F$ can be computed in time $O(n\mathtt{t}(n, p))$ and requires space $O(\mathtt{m}(n, p))$ using forward-mode AD (just use unitary vectors $r = e_i$ for $i = 1, \ldots, n$).

(iv) Similarly, $\mathrm{D}F$ can be computed in time $O(p\mathtt{t}(n, p))$ and requires space $O(\mathtt{t}(n, p))$ using reverse-mode AD.

Let $\mathtt{t}(d', m)$ and $\mathtt{m}(d', m)$ denote time and space, respectively, required to evaluate the update map Φ_t defined in Section 5.3.1. Then the response function $f_T : \mathbb{R}^m \mapsto \mathbb{R}$ defined in Equation (5.8) can be evaluated in time $O(T\mathtt{t}(d', m))$ (assuming the time required to compute the validation error $E(\lambda)$ does not affect the bound[1]) and requires space $O(\mathtt{m}(d', m))$ since variables s_t may be overwritten at each iteration. Then, a direct

[1]This is indeed realistic since the number of validation examples is typically lower than the number of training iterations.

application of Fact (i) above shows that Algorithm 3 runs in time $O(Tm\mathtt{t}(d',m))$ and space $O(\mathtt{m}(d',m))$. The same results can also be obtained by noting that in Algorithm 3 the product $A_t Z_{t-1}$ requires m Jacobian-vector products, each costing $O(\mathtt{t}(d',m))$ (from Fact (i)), while computing the Jacobian B_t takes time $O(m\mathtt{t}(d',m))$ (from Fact (iii)).

Similarly, a direct application of Fact (ii) shows that Algorithm 2 has both time and space complexities $O(T\mathtt{t}(d',m))$. Again the same results can be obtained by noting that $\alpha_{t+1}A_{t_1}$ and $\alpha_t B_t$ are left Jacobian-vector products that in reverse-mode take both time $O(\mathtt{t}(d',m))$ (from Fact (ii)). Unfortunately, in this case, the variables s_t cannot be overwritten, explaining the much higher space requirement.

As an example, consider training a neural network with d weights[2], using classic iterative optimization algorithms such as SGD (possibly with momentum) or Adam, where the hyperparameters are just learning rate and momentum terms. In this case, $d' = O(d)$ and $m = O(1)$. Moreover, $\mathtt{t}(d',m)$ and $\mathtt{m}(d',m)$ are both $O(d)$. As a result, Algorithm 2 runs in time and space $O(Td)$, while Algorithm 3 runs in time $O(Td)$ and space $O(d)$, which would typically make a dramatic difference in terms of memory requirements.

Regarding the fixed-point hypergradient procedure of Algorithm 4, let $\mathtt{t}(d,m)$ and $\mathtt{m}(d,m)$ be the time and space cost required for evaluating the mapping Φ, introduced in Section 5.4.3. Then, still assuming that evaluating the gradient of the validation error does not affect the bound, applying fact (ii) to the update (5.30), one obtains that Algorithm 4 has time complexity $O(K\mathtt{t}(d,m))$ and space complexity $O(\mathtt{t}(d,m))$, where K is the set number of iterations. Indeed, in this case, one only needs to store the last iterate w_T, while the variables q_t may be replaced. Furthermore we not that also the conjugate gradient method only needs to compute at each step exactly one Hessian vector product, hence the same bounds apply, where K denotes, this time, the number of CG updates. Note that these estimates do not take into account the cost of finding w_T, since for implicit methods the approximate minimizer is considered to be an input of the procedure.

[2]This includes linear SVM and logistic regression as special cases.

6.1.1 Empirical Validation

To complement the complexity analysis in Section 6.1, we study empirically the running time per *hyperiteration* (an optimization step for λ) and space requirements of `Reverse-HG` and `Forward-HG` algorithms[3]. We trained three layers feed-forward neural networks on MNIST dataset with SGDM, for $T = 1000$ iterations. In a first set of experiments (Figure 6.1, left) we fixed the number of weights at 199210 and optimized the learning rate, momentum factor and a varying number of example weights in the training error, similar to the experiment on data-hypercleaning reported in [Franceschi et al., 2017] As expected, the running time of Reverse-HG is essentially constant, while that of Forward-HG increases linearly. On the other hand, when fixing the number of hyperparameters (learning rate and momentum factor), the space complexity of Reverse-HG grows linearly with respect to the number of parameters (Figure 6.1, right), while that of Forward-HG remains constant.

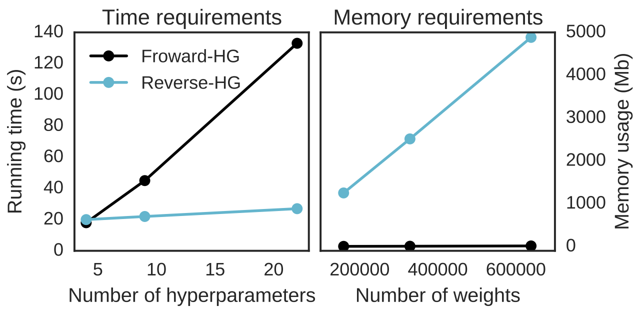

Figure 6.1: Time (left) and space (right) requirements for the computation of the hypergradient with Forward-HG and Reverse-HG algorithms.

6.2 Approximation Proprieties of Iterative Approach

Procedure (5.8)-(5.9), though related to the bilevel problem (5.1)-(5.2), may not be, in general, a good approximation of it. Indeed, making the assumptions (which sound perfectly reasonable) that, for every $\lambda \in \Lambda$, $w_T(\lambda) \to w(\lambda)$ for some $w(\lambda) \in \arg\min L_\lambda$, and that $E(\cdot, \lambda)$ is continuous, one can only assert that $\lim_{T\to\infty} f_T(\lambda) = E(w(\lambda), \lambda) \geq f(\lambda)$. This is because the optimization dynamics converge to some minimizer of the inner objective L_λ, but not necessarily to the one that also minimizes the function

[3]Code for this experiment, based on TensorFlow 1 [Abadi et al., 2015], is available at `https://github.com/lucfra/RFHO`.

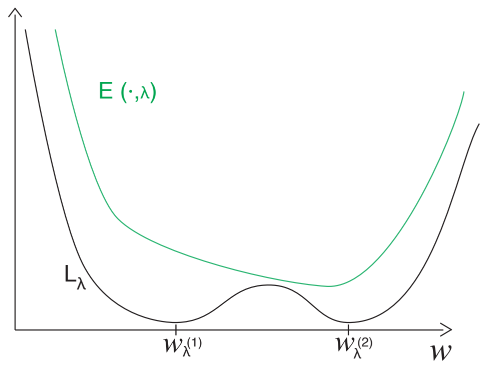

Figure 6.2: In this cartoon, for a fixed λ, argmin $L_\lambda = \{w_\lambda^{(1)}, w_\lambda^{(2)}\}$; the iterates of an optimization mapping Φ could converge to $w_\lambda^{(1)}$ with $E(w_\lambda^{(1)}, \lambda) > E(w_\lambda^{(2)}, \lambda)$.

E. This is illustrated in Figure 6.2. The situation is, however, different if the inner problem admits a unique minimizer for every $\lambda \in \Lambda$. Indeed in this case, it is possible to show that the set of minimizers of the approximate problems converge, as $T \to +\infty$ and in an appropriate sense, to the set of minimizers of the bilevel problem. More precisely, we make the following assumptions:

Assumption A.

1. Λ is a compact subset of $\mathbb{R}^m$;
2. $E : \mathbb{R}^d \times \Lambda \to \mathbb{R}$ is jointly continuous;
3. the map $(w, \lambda) \mapsto L_\lambda(w)$ is jointly continuous and such that $\arg\min L_\lambda$ is a singleton for every $\lambda \in \Lambda$;
4. $w(\lambda) = \arg\min L_\lambda$ remains bounded as λ varies in Λ.

Then, problem (5.1)-(5.2) becomes

$$\min_{\lambda \in \Lambda} f(\lambda) = E(w(\lambda), \lambda), \qquad w(\lambda) = \text{argmin}_u L_\lambda(u). \tag{6.1}$$

Under the above assumptions, in the following we give results about the existence of solutions of problem (6.1) and the convergence of the approximate problems (5.8)-(5.9) towards problem (6.1) — relating the minima as well as the set of minimizers. In this respect we note that, since both f and f_T are nonconvex, argmin f_T and argmin f are in general nonsingleton, so an appropriate definition of set convergence is required.

Theorem 6.2.1 (Existence)**.** *Under Assumptions A problem* (6.1) *admits solutions.*

Proof. See Appendix C.1.

The result below follows from general facts on the stability of minimizers in optimization problems [Dontchev and Zolezzi, 1993].

Theorem 6.2.2 (Set Convergence of Minimizers)**.** *In addition to Assumptions A, suppose that:*

Assumption B.

1. $E(\cdot, \lambda)$ is uniformly Lipschitz continuous;
2. The iterates $(w_T(\lambda))_{T\in\mathbb{N}}$ converge uniformly to $w(\lambda)$ on Λ as $T \to +\infty$.

   Then

(a) $\inf f_T \to \inf f$*,*

(b) $\operatorname{argmin} f_T \to \operatorname{argmin} f$*, meaning that, for every* $(\lambda_T)_{T\in\mathbb{N}}$ *such that* $\lambda_T \in \operatorname{argmin} f_T$*, we have that:*

- $(\lambda_T)_{T\in\mathbb{N}}$ *admits a convergent subsequence;*
- *for every subsequence* $(\lambda_{K_T})_{T\in\mathbb{N}}$ *such that* $\lambda_{K_T} \to \bar{\lambda}$*, we have* $\bar{\lambda} \in \operatorname{argmin} f$*.*

Proof. See Appendix C.1.

We stress that assumptions A are very natural and satisfied by many problems of practical interests. Thus, the above results provide full theoretical justification to the proposed approximate procedure (5.8)-(5.9). The following remark discusses Assumption B.2.

Remark 6.2.3. If L_λ is strongly convex, then many gradient-based algorithms (e.g., standard and accelerated gradient descent) yield linear convergence of the iterates $w_T(\lambda)$'s. Moreover, in such cases, the rate of linear convergence is of type $(\nu_\lambda - \mu_\lambda)/(\nu_\lambda + \mu_\lambda)$, where ν_λ and μ_λ are the Lipschitz constant of the gradient and the modulus of strong convexity of L_λ respectively, (see Proposition 2.4.4). So, this rate can be uniformly bounded from above by $\rho \in\,]0,1[$, provided that $\sup_{\lambda\in\Lambda} \nu_\lambda < +\infty$ and

$\inf_{\lambda\in\Lambda}\mu_\lambda > 0$. Thus, in these cases $w_T(\lambda)$ converges uniformly to $w(\lambda)$ on Λ (at a linear rate).

6.2.1 The Effect of T

Motivated by the theoretical findings, we empirically investigate how solving the inner problem *approximately* (i.e. using small T) affects convergence, generalization performances, and running time. Let us consider the following form of the inner objective:

$$L_H(w) = \|Y - XHw\|^2 + \mu\|w\|^2, \tag{6.2}$$

where $X \in \mathbb{R}^{N\times d}$ is a design matrix of N examples, $Y \in \mathbb{R}^{N\times c}$ is a target output, $\mu > 0$ is a fixed regularization parameter and $H \in \mathbb{R}^{d\times d}$ is the hyperparameter, representing a linear feature map. L_H is strongly convex, with modulus $\mu > 0$ (independent on the hyperparameter H), and Lipschitz smooth with constant $\nu_H = 2\|(XH)^\top XH + \mu I\|$, which is bounded from above, if H ranges in a bounded set of square matrices. In this case assumptions (i)-(vi) are satisfied.

The solution of (6.2) is given by

$$w_H = [(XH)^T XH + \mu I]^{-1}(XH)^T Y.$$

As outer objective, we utilize a non-regularized square loss on validation data. In this setting, the bilevel problem reduces to a (non-convex) optimization problem in H, allowing us to compare the approximated solutions against the closed-form analytical one.

We use a subset of 100 classes ($c = 100$) extracted from Omniglot dataset [Lake et al., 2017] to construct a HPO problem aimed at tuning H. A training set D_{tr} and a validation set D_{val}, each consisting of three randomly drawn examples per class, were sampled to form the HPO problem. A third set D_{test}, consisting of fifteen examples per class, was used for testing. Instead of using raw images as input, we employ feature vectors $x \in \mathbb{R}^{256}$ computed by the convolutional network trained on one-shot five-ways ML setting as described in Sec. 7.2.2.

For the approximate problems we compute the hypergradient using Algorithm

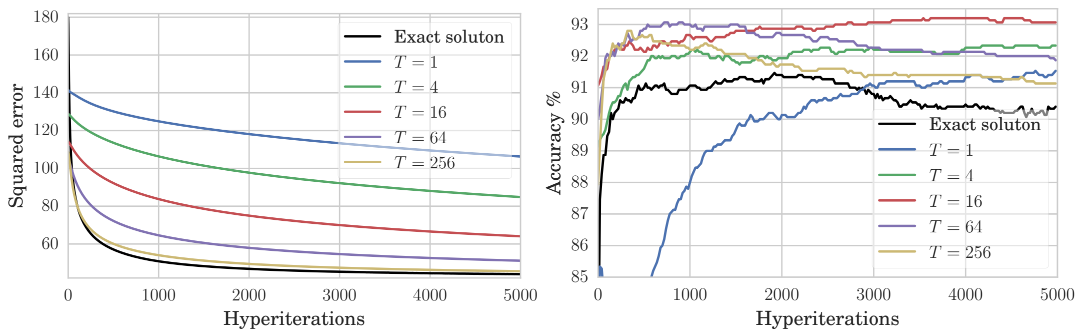

Figure 6.3: (Left) optimization of the outer objectives (validation error) f and f_T for exact and approximate problems. The optimization of H is performed with gradient descent with momentum, with same initialization, step size and momentum factor for each run. (Right) accuracy on D_{test} of exact and approximated solutions (training and validation accuracy scores reach almost 100% already for $T = 4$ and after few hundred hyperiterations and therefore are not reported). These two plot shows that for higher values of T, the approximate outer objectives converges toward the exact one, as expected from Theorem 6.2.2. However, in this experimental setting, lower values of f_T do not necessarily correspond to higher test accuracy scores and may lead to overfitting the validation error.

Table 6.1: Execution times on a NVidia Tesla M40 GPU.

T	1	4	16	64	256	Exact
Time (sec)	60	119	356	1344	5532	320

2. Figure 6.3 (left) shows the values of functions f and f_T (see Eqs. (5.1) and (5.8), respectively) during the optimization of H. As T increases, the solution of the approximate problem approaches the true bilevel solution (in value). However, in the setting of this experiment, performing a small number of gradient descent steps for solving the inner problem acts as an implicit regularizer. As it is evident from Figure 6.3 (right), the generalization error is better when T is smaller than the value yielding the best approximation of the inner solution. Here we see that tuning T helps contain this issue. As T increases, the number of hyperiterations required to reach the maximum test accuracy decreases, further suggesting that there is an interplay between the number of iterations used to solve the inner and the outer objective. Finally, because of the memory complexity of `Reverse-HG` (studied in Section 6.1) it is even more appealing to reduce the number of iterations when possible.

Despite the attractiveness of shorter horizons, we note, however, that some other problem settings may benefit instead from larger values of T. For instance, Wu et al.

[2018b] study the bias that stems from choosing too short horizons when optimizing learning rates (see also Chapter 8). In general, T may be considered as a configuration parameter of (iterative) gradient-based HPO and MTL methods which should be (meta-)validated; see also experiments in Sections 7.2.2 and 9.4.3. A correct horizon length may help avoid potential overfitting issues as in the numerical example of this section, or unwanted "short-horizon" biases, as noted in [Wu et al., 2018b].

We remind the reader to [Grazzi et al., 2020] for comparative experiments between iterative and implicit approaches regarding the hypergradient approximation errors. We now conclude the section with a simple example showing that, in certain cases, optimizing the parameters that control the dynamics Φ may lead to situations in which some assumptions are no longer satisfied, possibly causing divergence.

6.2.2 On Tuning the Hyperparameters of the Optimization Dynamics

With the iterative approximation scheme it becomes feasible to optimize also configuration parameters of the optimization dynamics Φ_t, such as the learning rate. This possibility has a great practical value (as we will see in Chapters 7 and 8) as it can substantially automate the search of critical hyperparameters, possibly speeding up the training procedure and leading to models with improved generalization. However, tuning these hyperparameters against the outer objective (e.g. a validation error) may result in situations in which some of the hypothesis of Theorem 6.2.2 are no longer satisfied. Consider, for instance, the very simple scalar bilevel problem

$$\min_{\lambda\in[-M,M]} \lambda^2 - \left(w(\lambda) - \frac{\lambda}{2}\right)^2, \quad \text{s.t.} \quad w(\lambda) = \operatorname{argmin}_{u\in\mathbb{R}} \frac{(u-\lambda)^2}{2} \tag{6.3}$$

for $M \in \mathbb{R}$, which satisfies all the hypothesis of Theorem 6.2.2 (but $E(w,\lambda)$ is not convex when seen as a function of two variables). The solution of the inner problem is simply given by $w(\lambda) = \lambda$ and hence $\lambda^* = 0$ is the unique minimizer of (6.3), with $f(\lambda) = 0$. By replacing the inner problem with a gradient descent optimization dynamics, one obtains

$$\min_{\lambda\in[-M,M]} \lambda^2 - \left(w_T(\lambda) - \frac{\lambda}{2}\right)^2, \tag{6.4}$$

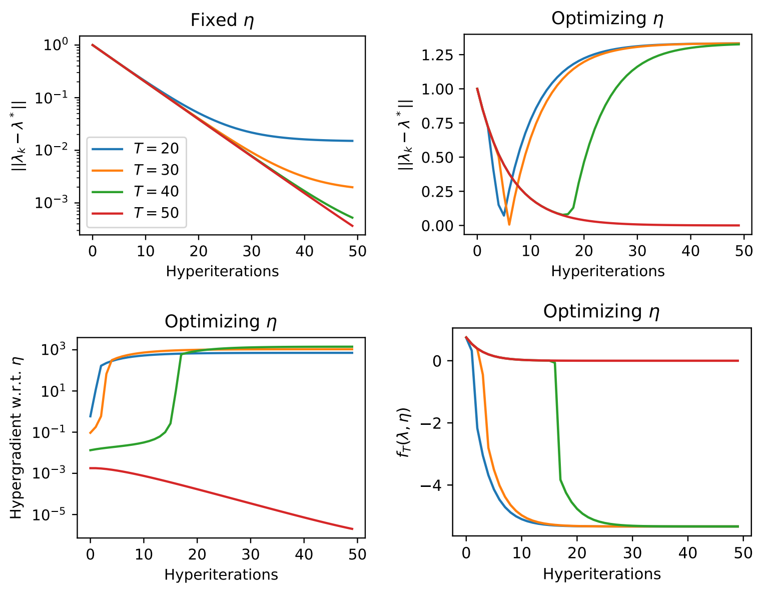

Figure 6.4: Results about the optimality gap (top rows) from the numerical solution of Problem (6.4)-(6.5) when keeping the learning rate constant (top left) and when tuning the learning rate as well (top right), for various horizons T. Bottom left: value of the hypergradient $\partial_\eta f_T$. Bottom right: value of the outer objective (which, in this case, is also a function of η).

$$\text{s.t.} \quad w_0 = \bar{w}, \quad w_t(\lambda) = \Phi(w_{t-1}, \lambda, \eta) = w_{t-1} - \eta(w_{t-1} - \lambda) \quad t \in [T]. \tag{6.5}$$

In Figure 6.4 top left we report the optimality gap ($\|\lambda_k - \lambda^*\| = |\lambda_k|$) as a function of the hyperiterations (updates of λ) for various values of T, keeping a constant learning rate of $\eta = 10^{-1}$ (we use `Forward-HG` to compute the hypergradient). The procedure approximately converges for all values of T, although for smaller horizons the gap remain larger, as w_T remains further away from $w(\lambda)$. The situation changes radically if we treat also the learning rate $\eta \geq 0$ as an hyperparameter to optimize. In this case we observe (Figure 6.4 top right) that for $T \leq 40$ the iterate λ_k no longer converges to 0. A closer inspection reveals that, during the optimization, the learning rate goes to 0. In fact, by letting $\eta \to 0$ the outer objective f_T (which becomes the single level objective $\lambda - \left(\bar{w} - \frac{\lambda}{2}\right)^2$) achieves a smaller value than $f(\lambda^*)$ (Figure 6.4 bottom right). The same does not happen for $T = 50$ although the reason is that the hypergradient w.r.t. η in this case, whilst still positive, vanishes toward 0 (bottom-left plot of the figure). Hence, in this case, if the horizon T is too small, by optimizing η one clearly ends up violating the hypothesis B.2 of Theorem 6.2.2 that $w_T \to w(\lambda)$.

This phenomenon does not necessarily occur in more realistic scenarios where typically the inner and outer objective are “better aligned”, but should nevertheless highlight the nature of the approximation that derives from the iterative approach. In fact, substituting the inner problem with an optimization dynamics gives raise to a constrained optimization problem. The solutions to this latter problem are those that minimize the outer objective f_T. Hence, if the formulation of the problem offers a configuration that decreases f_T (beyond $\min f$) while still respecting the T equality constraints (for instance, by zeroing the learning rate), this can be returned by the outer optimization procedure, irrespective of the optimality of the original inner problem.

6.3 Error Bounds for Hypergradient Approximation

We now derive non-asymptotic bounds on the hypergradient approximation error for the algorithms presented in Section 5.4, first shown in [Grazzi et al., 2020]. In this section we consider the following bilevel problem:

$$\min_{\lambda \in \Lambda} f(\lambda) := E(w(\lambda), \lambda) \qquad \text{subject to } \; w(\lambda) = \Phi(w(\lambda), \lambda), \tag{6.6}$$

where Λ is a closed convex subset of $\mathbb{R}^m$, $E\colon \mathbb{R}^d \times \Lambda \to \mathbb{R}$ and $\Phi\colon \mathbb{R}^d \times \Lambda \to \mathbb{R}^d$ are continuously differentiable functions. Note that this problem is strongly related to (6.1) and it represents, in a sense, an extension of it. Indeed, if in (6.1) we consider strictly convex inner objectives $L_\lambda \in C^2(\mathbb{R}^d)$, we can let Φ be such that $\Phi(w, \lambda) = w - \eta_\lambda \nabla L_\lambda(w)$, where η_λ is an appropriate step-size (that may depend on λ), then problem (6.1) and problem (6.6) are equivalent.

We make the following assumptions, related to the continuous differentiability of the outer objective and of the inner dynamics, and the contractiveness of the dynamics, which strengthen those in A and B.

Assumption C. For every $\lambda \in \Lambda$,

1. $\partial_w \Phi(w, \lambda)$ and $\partial_\lambda \Phi(w, \lambda)$ are Lipschitz continuous with constants $\nu_{1,\lambda}$ and $\nu_{2,\lambda}$ respectively.
2. $\partial_w E(w, \lambda)$ and $\partial_\lambda E(\cdot, \lambda)$ are Lipschitz continuous with constants $\xi_{1,\lambda}$ and $\xi_{2,\lambda}$

respectively.

3. For every $\lambda \in \Lambda$, $\Phi(\cdot, \lambda)$ is a contraction with constant $\rho_\lambda \in (0,1)$.

A direct consequence of Assumption C.3 is that, for each λ, the dynamics Φ has a unique fixed point $w(\lambda)$, and that the mapping $I - \partial_w \Phi(w, \lambda)$ is invertible. This further implies that, for the implicit function theorem [Krantz and Parks, 2012], $w(\cdot)$ and $f(\cdot)$ are differentiable on Λ. Specifically, for every $\lambda \in \Lambda$, it holds that

$$\partial_\lambda w(\lambda) = (I - \partial_w \Phi(w(\lambda), \lambda))^{-1} \partial_\lambda \Phi(w(\lambda), \lambda) \tag{6.7}$$

$$[\nabla f(\lambda)]^\intercal = \partial_\lambda E(w(\lambda), \lambda) + \partial_w E(w(\lambda), \lambda) \partial_\lambda w(\lambda). \tag{6.8}$$

When the dynamics Φ is gradient descent (with $\eta = 1$) applied to an inner objective L_λ, equation (6.7) reduces to (5.26). Furthermore, one has that

$$\left\| (I - \partial_w \Phi(w, \lambda))^{-1} \right\| \leq \frac{1}{1 - \rho_\lambda} \tag{6.9}$$

as a simple consequence of $\|\partial_w \Phi(w, \lambda)\| \leq \rho_\lambda < 1$, using the Neumann series for the inverse.

Remark 6.3.1. Assumption C.3 looks quite restrictive, however it is satisfied in a number of interesting cases:

(a) In the setting of Remark 6.2.3, for bilevel optimization problems of the form (6.1), when L_λ is μ_λ-strongly convex with Lipschitz constant ν_λ, the optimization dynamics of GD $\Phi(w, \lambda) = w - \eta_\lambda \nabla L_\lambda(w)$ is a contraction w.r.t. w with constant $\rho_\lambda = \frac{\nu_\lambda - \mu_\lambda}{\nu_\lambda + \mu_\lambda}$ when choosing the (optimal) learning rate $\eta_\lambda = \frac{2}{\mu_\lambda + \eta_\lambda}$.

(b) For strongly convex quadratic functions, accelerated methods like [Nesterov, 1983] or heavy-ball [Polyak, 1987b] can be formulated as fixed-point iterations of a contraction in the norm defined by a suitable positive definite matrix.

(c) In certain graph and recurrent neural networks of the form (6.15), where the transition function is assumed to be a contraction [Scarselli et al., 2009, Almeida, 1987, Pineda, 1987].

Before starting with the study of iterative and implicit differentiation, we give a lemma which introduces three additional constants that will occur in the error bounds.

Lemma 6.3.2. *Let $\lambda \in \Lambda$ and let $D_\lambda > 0$ be such that $\|w(\lambda)\| \leq D_\lambda$. Then there exist $\nu_{E,\lambda}, \nu_{\Phi,\lambda} \in \mathbb{R}_+$ such that*

$$\sup_{\|w\|\leq 2D_\lambda} \|\partial_w E(w,\lambda)\| \leq \nu_{E,\lambda}, \qquad \sup_{\|w\|\leq 2D_\lambda} \|\partial_\lambda \Phi(w,\lambda)\| \leq \nu_{\Phi,\lambda}$$

The proof [Grazzi et al., 2020] exploits the fact that the image of a continuous function applied to a compact set remains compact.

6.3.1 Iterative Differentiation

In this section we derive bounds for Algorithms 2 and 3 (which, we recall, constitute two different procedure to compute the same quantity) for the simplified case where $s_t = w_t$ (no auxiliary variables), $\Phi = \Phi_t$ for each t (fixed dynamics) and $\Phi_0 = 0$ (constant initialization). With Assumption C.3 in force and if $w_T(\lambda)$ is defined as in Equation (5.9), we have the following proposition that is essential for the final bound.

Proposition 6.3.3. *Suppose that Assumptions C.1 and C.3 hold and let $T \in \mathbb{N}$, with $T \geq 1$. Moreover, for every $\lambda \in \Lambda$, let $w_T(\lambda)$ be computed by Algorithm 2 or 3 (with the above specifications) and let D_λ and $\nu_{\Phi,\lambda}$ be as in Lemma 6.3.2. Then, $w_T(\cdot)$ is differentiable and, for every $\lambda \in \Lambda$,*

$$\|\partial_\lambda w_T(\lambda) - \partial_\lambda w(\lambda)\| \leq \left(\nu_{2,\lambda} + \nu_{1,\lambda}\frac{\nu_{\Phi,\lambda}}{1-\rho_\lambda}\right) D_\lambda T \rho_\lambda^{T-1} + \frac{\nu_{\Phi,\lambda}}{1-\rho_\lambda}\rho_\lambda^T, \tag{6.10}$$

where $\partial_\lambda w_T$ is obtained by the recursion[4]

$$\partial_\lambda w_t = \partial_w \Phi(w_{t-1},\lambda)\partial_\lambda w_{t-1} + \partial_\lambda \Phi(w_{t-1},\lambda) \qquad t \in [T]. \tag{6.11}$$

Proof. See Appendix C.2.

Leveraging Proposition 6.3.3, we give the main result of this section.

[4] Note that (6.11) is essential (5.20) in the simplified case with no auxiliary variables, fixed dynamics and constant initialization that we consider in this section.

Theorem 6.3.4. (Error bound for iterative differentiation). *Suppose that Assumptions C hold and let $T \in \mathbb{N}$ with $t \geq 1$. Let $D_\lambda, \nu_{E,\lambda}$, and $\nu_{\Phi,\lambda}$ be as in Lemma 6.3.2. Then, $f_T(\lambda) = E(w_T(\lambda), \lambda)$ is differentiable and, for every $\lambda \in \Lambda$,*

$$\|\nabla f_T(\lambda) - \nabla f(\lambda)\| \leq \Big(c_1(\lambda) + c_2(\lambda)\frac{T}{\rho_\lambda} + c_3(\lambda)\Big)\rho_\lambda^T, \tag{6.12}$$

where

$$\begin{aligned} c_1(\lambda) &= \left(\xi_{2,\lambda} + \frac{\xi_{1,\lambda}\nu_{\Phi,\lambda}}{1-\rho_\lambda}\right) D_\lambda, \\ c_2(\lambda) &= \left(\nu_{2,\lambda} + \frac{\nu_{1,\lambda}\nu_{\Phi,\lambda}}{1-\rho_\lambda}\right)\nu_{E,\lambda} D_\lambda, \\ c_3(\lambda) &= \frac{\nu_{E,\lambda}\,\nu_{\Phi,\lambda}}{1-\rho_\lambda}. \end{aligned}$$

Proof. See Appendix C.2.

In its generality this result provides a non-asymptotic linear rate of convergence for the gradient of f_T towards that of f that applies to Algorithms 2 and 3.

6.3.2 Implicit Differentiation [5]

In this section we derive approximation error bounds for the fixed-point implicit differentiation procedure described in Algorithm 4. We refer the reader to [Grazzi et al., 2020] for a more general result concerning the convergence of the broader computation scheme of Section 5.4.3 that includes also the implementation with conjugate gradient. The result in [Grazzi et al., 2020] relies on slightly less stricter assumptions, in particular not requiring global contractiveness of the mapping Φ. We recall that `Fixed-point-HG` regards $w_T(\lambda)$ as an input and that the the main computation is carried out by the recurrence

$$q_{T,k} = q_{T,k-1}\partial_w\Phi(w_T,\lambda) + \partial_w E(w_T,\lambda) \quad k \in [K], \tag{6.13}$$

with $q_{T,k} \in \mathbb{R}^{1\times d}$.

[5]The content of this section is attributed to my coauthors of [Grazzi et al., 2020]; included here for completion.

Theorem 6.3.5. (Error bound for fixed-point iterative differentiation). *Suppose that Assumptions C hold. Let $\nabla f_{T,K}(\lambda)$ be defined according to* (5.28)*, where $q_{T,K}$ is computed by iterating* (6.13). *Then, for every $T, K \in \mathbb{N}$,*

$$\|\nabla f_{T,K}(\lambda) - \nabla f(\lambda)\| \le \left(c_1(\lambda) + c_2(\lambda) \frac{1-\rho_\lambda^K}{1-\rho_\lambda} \right) \rho_\lambda(T) + c_3(\lambda) \rho_\lambda^K, \tag{6.14}$$

where $c_1(\lambda)$, $c_2(\lambda)$ and $c_3(\lambda)$ are given in Theorem 6.3.4 and $\rho_\lambda(t)$ is the convergence rate of $w_T \to w(\lambda)$

It is interesting to note that, if in Algorithm 4 we define $w_T(\lambda)$ as the $T-$th iterate of Φ as in the iterative differentiation approach (so that $\rho_\lambda(T) = \rho_\lambda^T$), and take $K = T$, then the bound for `Fixed-point-HG` (6.14) is lower than that of the iterative differentiation algorithms (6.12), since

$$\rho_\lambda(1-\rho_\lambda^T)/(1-\rho_\lambda) = \sum_{i=1}^{T} \rho_\lambda^i < T$$

for every $T \ge 1$. This analysis suggests that `Fixed-point-HG` may converge faster than `Reverse-HG` or `Forward-HG` (in terms of iterations), when Assumptions C are satisfied.

6.3.3 Discussion

Iterative differentiation for functions defined implicitly has been extensively studied in the automatic differentiation literature. In particular [Griewank and Walther, 2008, Chap. 15] derives asymptotic linear rates for iterative differentiation under the assumption that $\Phi(\cdot, \lambda)$ is a contraction. Iterative differentiation is also considered in [Shaban et al., 2019] where $\nabla f_t(\lambda)$ is approximated via a procedure which is reminiscent of truncated backpropagation. The authors bound the norm of the difference between $\nabla f_t(\lambda)$ and its truncated version as a function of the truncation steps. This is different from our analysis which directly considers the problem of estimating the gradient of f.

In the case of implicit differentiation, an asymptotic analysis is presented in [Pedregosa, 2016], where the author proves the convergence of an inexact gradient projection algorithm for the minimization of the function f defined in problem (6.1), using increasingly accurate estimates of $\nabla f(\lambda)$. Rajeswaran et al. [2019] present complexity results in the setting of meta-learning with biased regularization.

We also mention the papers by [Amos and Kolter, 2017] and [Amos, 2019], which present techniques to differentiate through the solutions of quadratic and cone programs respectively. Using such techniques allows one to treat these optimization problems as layers of a neural network and to use backpropagation for the end-to-end training of the resulting learning model. In the former work, the gradient is obtained by implicitly differentiating through the KKT conditions of the lower-level problem, while the latter performs implicit differentiation on the residual map of Minty's parametrization.

A different approach to solve bilevel problems of the form (6.1) is presented by [Mehra and Hamm, 2019], who consider a sequence of "single level" objectives involving a quadratic regularization term penalizing violations of the lower-level first-order stationary conditions. The authors provide asymptotic convergence guarantees for the method, as the regularization parameter tends to infinity, and show that it outperforms both iterative and implicit differentiation on different settings where the lower-level problem is non-convex.

All previously mentioned works except [Griewank and Walther, 2008] consider bilevel problems of the form (6.1). Another exception is [Liao et al., 2018], which proposes two improvements to recurrent backpropagation, one based on conjugate gradient on the normal equations, and another based on Neumann series approximation of the inverse.

6.3.4 The Effect of the Contractiveness Hypothesis

In this section we present experiments aimed at understanding the importance of the contractiveness hypothesis C.3 on the (empirical) stability and convergence of iterative and implicit hypergradient computation schemes. For this purpose we consider a setting in which the mapping Φ is not an optimization dynamics, but rather constitute (part of) the statistical model itself, as in case *(c)* of Remark 6.3.1. This help us better control the factor ρ_λ, allowing us to consider non-trivial settings where ρ_λ is less than 1 by design.

Specifically, we consider the following learning problem

$$\begin{aligned} &\min_{\lambda=(\gamma,\theta)\in\Lambda} f(\lambda) := \sum_{i=1}^{N} E_i(w^i(\gamma),\theta), \\ &\text{subject to } w^i(\gamma) = \phi_i(w^i(\gamma),\gamma), \text{ for } i \in [N], \end{aligned} \tag{6.15}$$

where the operators $\phi_i : \mathbb{R}^d \times \Lambda \to \mathbb{R}^d$ are associated to the training points x_i, and the error functions E_i are the losses incurred by a standard supervised algorithm on the transformed dataset $\{w^i(\gamma), y_i\}_{i=1}^N$. Here $\Phi = (\phi_i)_{i=1}^N$ and $w = (w^i)_{i=1}^N$ When iterating the dynamics ϕ (for T steps, or until convergence), one may consider that the w_t^i for $t \in [T]$ are internal intermediate representations akin to neural network activations.

We use a subset of $N = 5000$ instances randomly sampled from the MNIST dataset as training data and employ a multiclass logistic classifier paired with a cross-entropy loss. We picked a small training set and purposefully avoided stochastic optimization methods to better focus on issues related to the computation of the hypergradients itself, avoiding the introduction of other sources of noise. We model the learnable dynamics with parameters $\gamma = (C, C', c)$ as

$$\phi_i(w_i,\gamma) = \tanh\left(C \star w_i + \mu_{2\times2}(C' \star x_i) + c\right) \tag{6.16}$$

where $w_i \in \mathbb{R}^{h\times14\times14}$ are the state feature maps, $\star$ denotes multi-channel bidimensional cross-correlation, C and C' contain h 3×3 convolutional kernels each and $\mu_{2\times2}$ denotes the max-pooling operator with a 2×2 field and stride of 2. The state feature maps are passed through a max-pooling operator before being flattened and fed to a multiclass logistic classifier. We set $h = 10$ for all the experiments. We use the results and the code of Sedghi et al. [2019] to efficiently perform the projection of the linear operator associated to C into the unit spectral ball. This ensures that the transition mappings (6.16), and hence Φ, are contractions. This can be achieved during optimization by projecting the singular values of $\mathrm{p}(C)$ onto the interval $[0, 1-\varepsilon]$ for $\varepsilon > 0$, where $\mathrm{p}(C)$ is an $h\times h$ matrix of doubly block circulant matrices [see Sedghi et al., 2019, for details]. We note that regularizing the norm of $\partial_{w_i}\phi_i$ or adding L_1 or L_∞ penalty terms

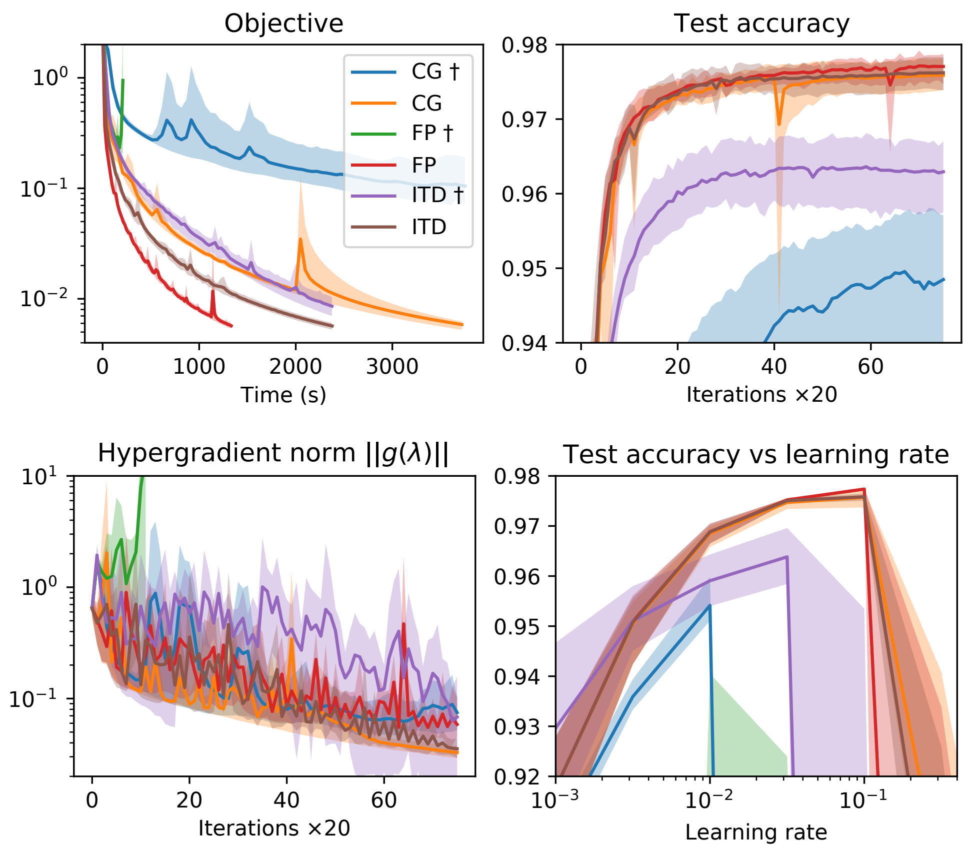

Figure 6.5: Experiments with convolutional equilibrium models. Mean (solid line) and point-wise minimum-maximum range (shaded region) across 5 random seeds. The seed only controls the initialization of λ. The estimated hypergradient $g(\lambda)$ is equal to $\nabla f_T(\lambda)$ for iterative differentiation and $\nabla f_{T,K}(\lambda)$ for implicit differentiation. We used $T = K = 20$ for all methods and Nesterov momentum (1500 iterations) for optimizing λ, applying a projector operator at each iteration except for the methods marked with †. In the first three plots we report for each method runs executed with the step-size that lead to the highest validation accuracy on the gird of the last plot (bottom-right). When the dynamics is contractive, all the three methods perform equally well, with FP being the fastest. Conversely, CG and especially FP are clearly outperformed by ITD in the unconstrained case.

on $\mathrm{p}(C)$ may encourage, but does not strictly enforce, $\left\|\mathrm{p}(C)\right\| < 1$.

The first three plots of Figure 6.5 report training objectives, test accuracy and norms of the estimated hypergradient for hypergradient computation methods[6], either applying or not the constraint on $\mathrm{p}(C)$. The bottom-right plot explores the sensitivity

[6] Specifically "CG" refers to implicit differentiation with conjugate gradient, "FP" to `Fixed-point-HG` and "ITD" to iterative differentiation implemented with "Reverse-HG". Note that, since here $\partial_w\Phi$ is not symmetric, the conjugate gradient method must be applied on the normal equations. Code for reproducing the experiments, developed on the top of Pythorch [Paszke et al., 2017], is available at `https://github.com/prolearner/hypertorch`.

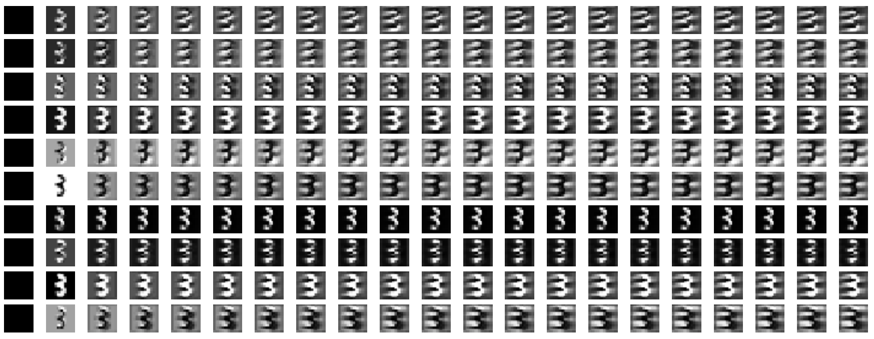

Figure 6.6: Visualization of the states filter maps $w_i \in \mathbb{R}^{10\times14\times14}$ for the image of a three from the MNIST dataset, learnt with the fixed-point method with contractive dynamics. Each of the ten rows represents a filter and the x-axis proceeds with the iterations of the dynamics (6.16) (for a total of $t = 20$ iterations). The states are initialized at 0 (black images on the left) and the mapping (6.16) is iterated 20 times to reach an approximate fixed point representation (rightmost images).

of the methods to the choice of the learning rate. Unconstrained runs are marked with †. Referring to the rightmost plot, it is clear that not constraining the spectral norm results in unstable behaviour of the "memory-less" implicit differentiation methods (green and blue lines) for all but a few learning rates, while iterative differentiation (violet), as expected, suffers comparatively less. Interestingly, when the projection is not performed, optimization with the fixed-point scheme (akin to recurrent backpropagation) does not reliably converge for all the probed values of the step-size, see green shaded region in the rightmost plot of Figure 6.5. This indicates the importance of the contractiveness assumption for implicit differentiation methods. On the contrary, when $\|\mathrm{p}(C)\| < 1$ is enforced, all the approximation methods are successful and stable, with `Fixed-point-HG` to be preferred being faster then CG on the normal equations and requiring substantially less memory than `Reverse-HG`.

We show some visual examples of the learned dynamics in Figure 6.6, where we plot the 10 state filter maps as the iterations of (6.16) proceed.

6.4 Interim Summary

The gradient-based algorithms for finding solutions to bilevel problems that arise in HPO and MTL, introduced in the previous chapter, have different trade-offs in terms of computational complexity, which we analyzed in Section 6.1. `Reverse-HG` is efficient when λ is high dimensional and the number of iterations or the dimensionality of the parameter vector w are comparatively low. `Forward-HG`, on the contrary, is effective when λ is low dimensional. `Fixed-point-HG`, needing to store only the last iterate and maintaining only vector adjoints, inherits the computational advantages of both the iterative procedures, at the cost of applying to a more restricted set of cases.

In the other two sections of the chapter we focused on the approximation proprieties of the algorithms. First, the iterative approach constructs a sequence of approximate problems indexed by the horizon length T. In Section 6.2 we showed that when the inner problem has a unique solution (e.g. is strongly convex) the minimizers of the approximate problems converge to those of the original bilevel program. Second, in Section 6.3, we turned our attention to the approximation error on the computation of the hypergradient, considering a more general version of the bilevel problem that involves a fixed-point equation at the inner level. Under the assumption that the equation is defined by a contraction mapping, we established results on the iteration complexity of the two strategies to compute the hypergradient, showing that they both exhibit liner rates of convergence.

We accompanied the theoretical analysis with four simple numerical simulations. The first provides empirical evidence of the complexity analysis. Then, two set of experiments investigate particular aspect of the iterative approach. The simulation of Section 6.2.1 focuses on the potential statistical (and computational) benefits of short horizons for problems with many hyperparameters, linking the effect to implicit regularization akin to early stopping (Section 2.4.4). The numerical simulation will also be relevant for experiments in few-shot meta-learning which we will report in the next chapter. In Section 6.2.2, instead, we showed that tuning the learning rate of the optimization dynamics may cause divergence in certain cases. While this occurrence mainly depends on the specific outer objective and problem formulation, the example highlights some aspects of the type of approximation arising from the iterative reformulation. In Section 6.3.4 we presented some comparative results concerning the stability of the hypergradient computation related to the findings of Section 6.3. We found that the implicit schemes (and in particular `Fixed-Point-HG`) are more sensitive to the contractiveness of the dynamics Φ and under-perform with this is not enforced. This suggests that, despite the computational advantages of the implicit approaches, iterative schemes represent a more robust choice, when computationally feasible.

Chapter 7

Experiments on Approximate Bilevel Programming

We present here a series of experiments on both hyperparameter optimization and meta-learning, instantiating the proposed bilevel framework in various learning scenarios. While each experiment in the previous chapter was directly related to theoretical aspects discussed there, the primary aim of the experiments of this chapter is to showcase the effectiveness of the framework in learning scenarios closer to applications. Specifically, we show how the framework seamlessly allows one to formulate different learning problems and applications following a unified "design pattern". Approximate solutions of these problems may then be sought by a gradient-descent procedure (3.5) that typically requires minimal adjustments, where the hypergradient is estimated with one of the algorithms introduced in Section 5.4.

In standard supervised learning, the possibility of effectively optimize over high-dimensional hyperparameter spaces utilizing `Reverse-HG` makes it feasible to consider learning algorithms that would be otherwise deemed too difficult to tune if tackled with classic approaches to HPO. In our experiments of Section 7.1, we explore this possibility, presenting a learning problem to discover the relationships between different learning tasks. The forward mode, instead, is appropriate to tune a small number of critical hyperparameters and, crucially, it applies to learning algorithms which make use of large-scale models. In this regard, we give experimental evidence that the real-time version of this approach (Section 5.4.2.1) is efficient enough to allow for the automatic tuning of important hyperparameters of a deep learning model and compare the results with a number of other HPO techniques.

In Section 7.2, by taking inspiration on early work on representation learning in the context of multi-task and meta-learning [Baxter, 1995, Caruana, 1998], we instantiate the framework for MTL in a simple way, treating the weights of the last layer of a neural network as the inner variables. The remaining weights, which parameterize the representation mapping, act as the outer variables. As shown in Section 7.2.2, the resulting MTL algorithm performs well in practice. At the time of publication [Franceschi et al., 2018a], it outperformed most of the existing strategies on two benchmark datasets on few-shot learning. We finally conclude the MTL section with a comparative study that consider different baselines and optimization techniques for the few-shot learning problem, including runs where we do not split the data of the meta-training episodes (see Section 4.2).

Our experimental experience led to the development of two open-source Python packages, both based on the deep learning library TensorFlow [Abadi et al., 2015]. The first, called RFHO[1] has been used for the experiments of the first section of this chapter. The second, called Far-HO[2] is an expansion and revision of RFHO that includes the implementation of all the algorithms derived in Section 5.4 and has been presented at the AutoML workshop hosted at ICML 2018 [Franceschi et al., 2018b]. Far-HO has been used for the experiments of the second section of this chapter and for those in Chapter 9.

7.1 Gradient-based Hyperparameter Optimization

In this section, we present a series of experiments on hyperparameter optimization tasks, where the aim is to minimize a validation objective $f(\lambda)$ as defined in (3.8). In all the simulations, hyperparameters were updated with the Adam algorithm [Kingma and Ba, 2015] in order to minimize an (approximate) response function. The experiments of this section were first reported in [Franceschi et al., 2017].

[1]Available at `https://github.com/lucfra/RFHO`

[2] Available at `https://github.com/lucfra/FAR-HO`)

7.1.1 Learning Task Interactions

This second set of experiments is in the multitask learning (MTT) context, where the goal is to find simultaneously the model of multiple related tasks. Many MTT methods require that a task interaction matrix is given as input to the learning algorithm. However, in real applications, this matrix is often unknown and it is interesting to learn it from data. Below, we show that our framework can be naturally applied to learning the task relatedness matrix.

We used CIFAR-10 and CIFAR-100 [Krizhevsky and Hinton, 2009], two object recognition datasets with 10 and 100 classes, respectively. As features we employed the pre-activation of the second last layer of Inception-V3 model trained on ImageNet[3]. From CIFAR-10, we extracted 50 examples as training set, different 50 examples as validation set and the remaining for testing. From CIFAR-100, we selected 300 examples as training set, 300 as validation set and the remaining for testing. Finally, we used a one-hot encoder of the labels obtaining a set of labels in $\{0,1\}^K$ ($K = 10$ or $K = 100$).

The choice of small training set sizes is due to the strong discriminative power of the selected features. In fact, using larger sample sizes would not allow us to appreciate the advantage of MTT. In order to leverage information among the different classes, we employed a multitask learning regularizer [Evgeniou et al., 2005]

$$\Omega_{C,\rho}(W) = \sum_{j,k=1}^{K} C_{j,k}\|w_j - w_k\|_2^2 + \rho\sum_{k=1}^{K}\|w_k\|^2,$$

where w_k are the weights for class k, K is the number of classes, and the symmetric non-negative matrix C models the interactions between the classes/tasks. We used a regularized training error defined as $L_\lambda(W) = \sum_{i\in D_{\mathrm{tr}}}\ell(Wx_i + b, y_i) + \Omega_{C,\rho}(W)$ where $\ell(\cdot,\cdot)$ is the categorical cross-entropy and $b = (b_1,\ldots,b_K)$ is the vector of biases associated with each linear model. Here $\lambda = (\rho, C)$. We wish solve the following optimization problem:

$$\min\{E(W_T, b_T) \;\text{ subject to }\; \rho \geq 0,\; C = C^\intercal,\; C \geq 0\},$$

[3]Available at `tinyurl.com/h2x8wws`

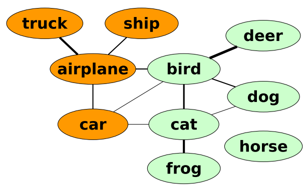

Figure 7.1: Relationship graph of CIFAR-10 classes. Edge thickness represents the interaction strength between classes.

Table 7.1: Test accuracy±standard deviation on CIFAR-10 and CIFAR-100 for single task learning (STL), naive MTT (NMTT) and our approach without (HMTT) and with (HMTT-S) the L1-norm constraint on matrix C.

	CIFAR-10	CIFAR-100
STL	67.47±2.78	18.99±1.12
NMTT	69.41±1.90	19.19±0.75
HMTT	70.85±1.87	21.15±0.36
HMTT-S	71.62±1.34	22.09±0.29

where (W_T, b_T) is the T-th iteration obtained by running gradient descent with momentum (GDM) on the training objective. We solve this problem using `Reverse-HG` and optimizing the hyperparameters by projecting Adam updates on the set $\{(\rho, C) : \rho \geq 0,\ C = C^\intercal,\ C \geq 0\}$. We compare the following methods:

- SLT: single task learning, i.e. $C = 0$, using a validation set to tune the optimal value of ρ for each task;
- NMTT: we considered the naive MTT scenario in which the tasks are equally related, that is $C_{j,k} = a$ for every $1 \leq j,k \leq K$. In this case we learn the two non-negative hyperparameters a and ρ;
- HMTT: our hyperparameter optimization method `Reverse-HG` to tune C and ρ;
- HMTT-S: Learning the matrix C with only few examples per class could bring the discovery of spurious relationships. We try to remove this effect by imposing the constraint that $\sum_{j,k} C_{j,k} \leq R$, where[4] $R = 10^{-3}$. In this case, Adam updates are projected onto the set $\{(\rho, C) : \rho \geq 0,\ C = C^\intercal,\ C \geq 0,\ \sum_{j,k} C_{j,k} \leq R\}$.

[4]We observed that $R = 10^{-4}$ yielded very similar results.

Results of five repetitions with different splits are presented in Table 7.1. Note that HMTT gives a visible improvement in performance, and adding the constraint that $\sum_{j,k} C_{j,k} \leq R$ further improves performance in both datasets. The matrix C can been interpreted as an adjacency matrix of a graph, highlighting the relationships between the classes. Figure 7.1 depicts the graph for CIFAR-10 extracted from the algorithm HMTT-S. Although this result is strongly influenced by the choice of the data representations, we can note that animals tends to be more related to themselves than to vehicles and vice versa.

Comparison with task-specific approaches. In Table 7.2, we report comparative results obtained with two state-of-the-art multitask learning methods ([Dinuzzo et al., 2011] and [Jawanpuria et al., 2015]) on the CIFAR-10 dataset.

Table 7.2: Test accuracy±standard deviation on CIFAR-10. Hyperparameters of MTT algorithms were validated by grid-search with the same experimental setting of Section 7.1.1. Jawanpuria et al. [2015] algorithm contains a p-norm regularizer for the task interaction matrix C, for $p \in (1,2]$. The value of p used in the experiment is specified in the third column.

	CIFAR-10	p
Dinuzzo et al. [2011]	69.96±1.85	
Jawanpuria et al. [2015]	70.30±1.05	2
Jawanpuria et al. [2015]	70.96±1.04	4/3
HMTT-S	71.62±1.34	

Both methods improve over STL and NMTT but perform slightly worse than HMTT-S. The task interaction matrix is treated as a model parameter by these algorithms, which may lead to overfitting for such a small training set, further highlighting the advantages of considering C as an hyperparameter. Computation times are comparable in the order of 2-3 hours.

Other approaches [e.g. Kang et al., 2011] tackle the same problem in a similar framework, but a complete analysis of MTT is beyond the scope of this work.

7.1.2 Phoneme Classification

The aim of the third set of experiments is to assess the efficacy of the real-time `Forward-HG` algorithm (RTHO) introduced in Section 5.4.2.1. We run experiments

Table 7.3: Frame level phone-state classification accuracy on standard TIMIT test set and execution time in minutes on one Titan X GPU. For RS, we set a time budget of 300 minutes.

Method	Accuracy %	Time (min)
STL	59.81	12
RS	60.36	300
SMBO	60.91	300
RTHO	61.97	164
RTHO-NT	61.38	289

on phoneme recognition in the multitask framework proposed in [Badino, 2016, and references therein]. Data for all experiments was obtained from the TIMIT phonetic recognition dataset [Garofolo et al., 1993]. The dataset contains 5040 sentences corresponding to around 1.5 million speech acoustic frames. Training, validation and test sets contain respectively 73%, 23% and 4% of the data. The primary task is a frame-level phoneme state classification with 183 classes and it consists in learning a mapping h_P from acoustic speech vectors to hidden Markov model monophone states. Each 25ms speech frame is represented by a 123-dimensional vector containing 40 Mel frequency scale cepstral coefficients and energy, augmented with their deltas and delta-deltas. We used a window of eleven frames centered around the prediction target to create the 1353-dimensional input to h_P. We consider a secondary (or auxiliary) task that consists in learning a mapping h_S from acoustic vectors to 300-dimensional real-valued vectors of context-dependent phonetic embeddings defined in [Badino, 2016].

As in previous work, we assume that the two mappings h_P and h_S share inputs and an intermediate representation, obtained by four layers of a feed-forward neural network with 2000 units on each layer. We denote by W the parameter vector of these four shared layers. The network has two different output layers with parameter vectors W^P and W^S each relative to the primary and secondary task. The network is trained to jointly minimize $L_\lambda(W, W^P, W^S) = L_P(W, W^P) + \rho L_S(W, W^S)$, where the primary error L_P is the average cross-entropy loss on the primary task, the secondary error L_S is given by mean squared error on the embedding vectors and $\rho \geq 0$ is a design hyperparameter. Since we are ultimately interested in learning h_P, we formulate the hyperparameter optimization problem as

$$\min\{E(W_T, W_T^P) \text{ subject to } \rho, \eta \geq 0, 0 \leq \mu \leq 1\},$$

where E is the cross entropy loss computed on a validation set after T iterations of stochastic GDM, and η and μ are defined in (5.10). Here $\lambda = (\rho, \eta, \mu)$ comprises both optimization and regularization/design hyperparameters. In all the experiments we fix a mini-batch size of 500. We compare the following methods:

1. STL: the secondary target is ignored ($\rho = 0$); η and μ are set to 0.075 and 0.5 respectively as in [Badino, 2016].

2. RS: random search (Section 3.4.3) with $\rho \sim \mathcal{U}(0,4)$, $\eta \sim \mathcal{E}(0.1)$ (exponential distribution with scale parameter 0.1) and $\mu \sim \mathcal{U}(0,1)$.

3. SMBO: sequential model-based Bayesian optimization with Gaussian processes[5] (Section 3.4.5). We set the following definition intervals for the hyperparameters: $\eta \in [10^{-5}, 1]$, $\mu \in [0, 0.999]$ and $\rho \in [0,4]$; we used expected improvement as acquisition function and initialized the Gaussian Process with the observed validation error of five randomly sampled configurations.

4. RTHO: real-time hyperparameter optimization with initial learning rate and momentum factor as in STL and initial ρ set to 1.6 (best value obtained by grid-search in [Badino, 2016]).

5. RTHO-NT: RTHO with "null teacher," i.e. when the initial values of ρ, η and μ are set to 0. We regard this experiment as particularly interesting: this initial setting, while clearly not optimal, does not require any background knowledge on the task at hand.

We also tried to run `Forward-HG` for a fixed number of epochs, not in real-time mode. Results are not reported since the method could not make any appreciable progress after running 24 hours on a Titan X GPU.

[5] Implementation available at `https://github.com/fmfn/BayesianOptimization/`.

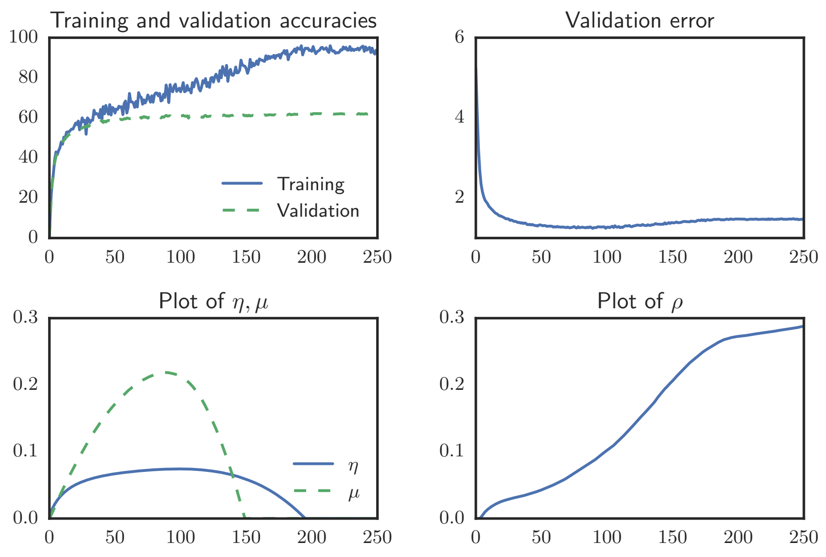

Figure 7.2: Learning curves and hyperparameter evolution for RTHO-NT: the horizontal axis runs with the hyper-batches. Top-left: frame level accuracy on mini-batches (Training) and on a randomly selected subset of the validation set (Validation). Top-right: validation error E on the same subset of the validation set. Bottom-left: evolution of optimizer hyperparameters η and μ. Bottom-right: evolution of design hyperparameter ρ.

Test accuracies and execution times are reported in Table 7.3. Figure 7.2 shows learning curves and hyperparameter evolutions for RTHO-NT. In Experiments 1 and 2 we employ a standard early stopping procedure on the validation accuracy, while in Experiments 4 and 5 a natural stopping time is given by the decay to 0 of the learning rate (see Figure 7.2 left-bottom plot). In Experiments 4 and 5 we used a hyper-batch size of $\Delta = 200$ (see Eq. (5.23)) and a hyper-learning rate of 0.005.

The best results in Table 7.3 are very similar to those obtained in state-of-the-art recognizers using multitask learning [Badino, 2016, 2017]. In spite of the small number of hyperparameters, random search yields results only slightly better than the STL network (the result reported in Table 7.3 are an average over 5 trials, with a minimum and maximum accuracy of 59.93 and 60.86, respectively). Within the same time budget of 300 minutes, RTHO-NT is able to find hyperparameters yielding a substantial improvement over the STL version, thus effectively exploiting the auxiliary task. Bayesian optimization finds in the same amount of time a model that outperforms that found by RS, but is slightly worse than RTHO-NT. This may also indicate that part of the performance gain with RTHO runs is due to use an hyperparameter scheduling[6] (Section 3.3) rather than fixed hyperparameter values. As a final remark, we note that the model trained has more that 15×10^6 parameters for a corresponding state of more than 30×10^6 variables. At the time of publication [Franceschi et al., 2017], reverse-mode or approximate implicit methods had not been applied to models of this size.

7.2 Meta-learning

We now turn our attention to the application of the framework described in Chapter 5 to the setting of meta-learning. We first describe a simple model in Section 7.2.1 which we call hyper-representation. We test our method in the context of few-shot learning on two benchmark datasets in Section 7.2.2 and, finally, we contrast the bilevel MTL approach against classical approaches to learn shared representations in Section 7.2.3. The meta-learning algorithm and the experiments of this section were presented in [Franceschi et al., 2018a].

7.2.1 Learning Hyper-Representations

Finding good data representations is a centerpiece of machine learning. Classical approaches [Baxter, 1995, Caruana, 1998] learn both the weights of the representation mapping and those of the base-level classifiers jointly on the same data. Here we follow the bilevel approach in the case of deep learning where representation layers are shared across episodes. According to the practice presented in Section 4.2, we split each dataset/episode in training and test sets.

[6]This may also partially explain the comparatively poor performances of random search in this setting. In fact, one may consider to tune hyperparameter schedules with random search. To do so, one could utilize predefined scheduling functions, or could seek for updates of the type $\lambda_t = \lambda_{t-1} + \varepsilon_t$, $\varepsilon_t \sim \nu_t$ where each ν_t are given probability distributions. However, both of these routes introduce a number of additional non-trivial design choices (i.e the shape of the scheduling function or the choice of the distributions ν_t) which are beyond the scope of our presentation.

Algorithm 5 Reverse-HG for Hyper-representation

```
Input: $\lambda$: current values of the hyperparameter; $T$: number of iteration of GD; $\eta$: base-level learning rate; $\mathcal{B}$: mini-batch of episodes from $\mathbf{D}_{\text{tr}}$
Output: Gradient of meta-training error w.r.t. $\lambda$ on $\mathcal{B}$
for $j = 1$ to $|\mathcal{B}|$ do
  $w_0^j = 0$
  for $t = 1$ to $T$ do
    $w_t^j \leftarrow w_{t-1} - \eta \nabla_w L^j(w_{t-1}^j, \lambda, D_{\text{tr}}^j)$
  end for
  $\alpha_T^j \leftarrow \nabla_w L^j(w_T^j, \lambda, D_{\text{val}})$
  $p^j \leftarrow \nabla_\lambda L^j(w_T^j, \lambda, D_{\text{val}})$
  for $t = T-1$ downto $0$ do
    $p^j \leftarrow p^j - \alpha_{t+1}^j \eta \nabla_\lambda \nabla_w L^j(w_t^j, \lambda, D_{\text{tr}}^j)$
    $\alpha_t^j \leftarrow \alpha_{t+1}^j \left[I - \eta \nabla_w \nabla_w L^j(w_t^j, \lambda, D_{\text{tr}}^j)\right]$
  end for
end for
return $\sum_j p^j$
```

Our method involves the learning of a cross-task intermediate representation $r_\lambda : \mathcal{X} \to \mathbb{R}^k$ (parametrized by a vector λ) on top of which task specific models $s^j : \mathbb{R}^k \to \mathcal{Y}^j$ (parametrized by vectors w^j) are trained. The final base-level model for task j is thus given by $s^j \circ r_\lambda$. To find λ, we solve Problem (5.1)-(5.2) with inner and outer objectives as in Equations (5.6) and (5.7), respectively. We follow the iterative scheme (Section 5.3.1) to define the approximate problem, that is given by:

$$\min_\lambda f_T(\lambda) = \sum_{j=1}^{N} L^j(w_T^j, \lambda, D_{\text{ts}}^j) \tag{7.1}$$

$$w_t^j = w_{t-1}^j - \eta \nabla_w L^j(w_{t-1}^j, \lambda, D_{\text{tr}}^j), \qquad t \in [T],\ j \in [\mathbf{N}]. \tag{7.2}$$

Starting from an initial value, the weights of the task-specific models are learned by T iterations of gradient descent. The gradient of f_T can be computed efficiently in time by making use of the extended reverse-hypergradient procedure of Section 5.4.1, which we specify for this setting in Algorithm 5. Since, in general, the number of episodes in a meta-training set is large, we compute a stochastic approximation of the gradient of f_T by sampling a mini-batch of episodes. At test time, given a new episode

$\bar{D}$, the representation r_λ is kept fixed, and all the examples in $\bar{D}$ are used to tune the weights $\bar{w}$ of the episode-specific model $\bar{s}$.

Our method belongs to the class of parametric algorithmic MTL techniques, outlined in Section 4.4.2.3. Like other initialization and optimization strategies for MTL, our method does not require lookups in a support set as the memorization and metric strategies do [Santoro et al., 2016, Vinyals et al., 2016, Mishra et al., 2018]. Unlike [Andrychowicz et al., 2016, Ravi and Larochelle, 2017] we do not tune the optimization algorithm, which in our case is plain empirical loss minimization by gradient descent, and rather focus on the hypothesis space. Unlike [Finn et al., 2017], that aims at maximizing sensitivity of new task losses to the model parameters, we aim at maximizing the generalization to novel examples during training episodes, with respect to λ. Our assumptions about the structure of the model are slightly stronger than in [Finn et al., 2017] but still mild, namely that some (hyper)parameters define the representation and the remaining parameters define the classification function. In [Munkhdalai and Yu, 2017] the meta-knowledge is distributed among fast and slow weights and an external memory; our approach is more direct, since the meta-knowledge is solely distilled by λ. A further advantage of our method is that, if the episode-specific models are linear (e.g. logistic regressors) and each loss L^j is strongly convex in w, the theoretical guarantees of Theorem 6.2.2 apply (see Remark 6.2.3). These assumptions are satisfied in the experiments reported in the next section.

Finally, we note that the method proposed by Bertinetto et al. [2019] (published short after [Franceschi et al., 2018a]) is related very closely to our proposed approach. In particular, Bertinetto et al. use ridge regression (rather than multinomial logistic regression) at the base-level[7]. Hence, by taking advantage of the closed-form solution, they obtain and solve a single level problem for learning the parameters of the feature extractor at the meta-level.

[7]Akin to the problem presented in Section 6.2.1; the linear transformation H is replaced by a deep learning model in [Bertinetto et al., 2016], as we do in the next section.

7.2.2 Few-shot Learning

We now turn our attention to few-shot supervised learning, implementing the MTL strategy outlined above on two different benchmark datasets:

• Omniglot [Lake et al., 2015], a dataset that contains examples of 1623 different handwritten characters from 50 alphabets. We downsample the images to 28×28.

• MiniImagenet [Vinyals et al., 2016], a subset of ImageNet [Deng et al., 2009], that contains 60000 downsampled images from 100 different classes.

Following the experimental protocol used in a number of recent works, we build a meta-training set $\mathbf{D}_{\text{tr}}$, from which we sample datasets to solve Problem (7.1)-(7.2), a meta-validation set $\mathbf{D}_{\text{val}}$ for tuning MTL hyperparameters, and finally a meta-test set $\mathbf{D}_{\text{ts}}$ which is used to estimate accuracy. Operationally, each meta-dataset consists of a pool of samples belonging to different (non-overlapping between separate meta-dataset) classes, which can be combined to form base-level classification datasets $D^j = D^j_{\text{tr}} \cup D^j_{\text{ts}}$ with 5 or 20 classes (for Omniglot). The D^j_{tr}'s contain 1 or 5 examples per class which are used to fit w^j (see Eq. 7.2). The D^j_{ts}'s, containing 15 examples per class, is used either to compute $f_T(\lambda)$ (see Eq. (7.1)) and its (stochastic) gradient if $D^j \in \mathbf{D}_{\text{tr}}$ or to provide a generalization score if D^j comes from either $\mathbf{D}_{\text{val}}$ or $\mathbf{D}_{\text{ts}}$. For MiniImagenet we use the same split and images proposed in [Ravi and Larochelle, 2017], while for Omniglot we use the protocol defined by [Santoro et al., 2016].

As base-level classifiers we use multinomial logistic regressors and as task losses ℓ^j we employ cross-entropy. The inner problems, being strongly convex, admit unique minimizers, yet require numerical computation of the solutions. We initialize base-level models parameters w^j to 0 and, according to the observation in Sec. 6.2.1, we perform T gradient descent steps, where T is treated as an hyperparameter of the meta-learning algorithm that has to be validated. Figure 7.3 shows an example of meta-validation of T for one-shot learning on MiniImagenet. We compute a stochastic approximation of $\nabla f_T(\lambda)$ with Algorithm 5 and use Adam with decaying learning rate to optimize λ.

Regarding the specific implementation of the representation mapping r_λ, we employ for Omniglot a four-layers convolutional neural network with strided convolutions

Table 7.4: Accuracy scores, computed on episodes from $\mathbf{D}_{\text{ts}}$, of various methods on 1-shot and 5-shot classification problems on Omniglot and MiniImagenet. For MiniImagenet 95% confidence intervals are reported. For Hyper-representation the scores are computed over 600 randomly drawn episodes. For other methods we show results as reported by their respective authors: [1] Koch et al. [2015]; [2] Vinyals et al. [2016]; [3] Edwards and Storkey [2017]; [4] Kaiser et al. [2017]; [5] Ravi and Larochelle [2017]; [6] Finn et al. [2017]; [7] Munkhdalai and Yu [2017]; [8] Snell et al. [2017]; [9] Mishra et al. [2018]

	Omniglot 5 cl.		Omniglot 20 cl.		MiniImagenet 5 classes	
Method	1-shot	5-shot	1-shot	5-shot	1-shot	5-shot
Siamese nets [1]	97.3	98.4	88.2	97.0	–	–
Matching nets [2]	98.1	98.9	93.8	98.5	43.44 ± 0.77	55.31 ± 0.73
Neural statistician [3]	98.1	99.5	93.2	98.1	–	–
Memory modules [4]	98.4	99.6	95.0	98.6	–	–
Meta-LSTM [5]	–	–	–	–	43.56 ± 0.84	60.60 ± 0.71
MAML [6]	98.7	99.9	95.8	98.9	48.70 ± 1.75	63.11 ± 0.92
Meta-networks [7]	98.9	–	97.0	–	49.21 ± 0.96	–
Prototypical Nets [8]	98.8	99.7	96.0	98.9	49.42 ± 0.78	68.20 ± 0.66
SNAIL [9]	99.1	99.8	97.6	99.4	55.71 ± 0.99	68.88 ± 0.92
Hyper-representation	98.6	99.5	95.5	98.4	50.54 ± 0.85	64.53 ± 0.68

and 64 filters per layer as in [Vinyals et al., 2016] and other successive works. For MiniImagenet we tried two different architectures:

• *C4L*, a four-layers convolutional neural network with max-pooling and 32 filters per layer;

• *RN*: a residual network [He et al., 2016] built of four residual blocks followed by two convolutional layers.

The first network architecture has been proposed in [Ravi and Larochelle, 2017] and then used in [Finn et al., 2017], while a similar residual network architecture has been employed in a more recent work [Mishra et al., 2018]. We report our results, using *RN* for MiniImagenet, in Table 7.4, alongside scores from various proposed methods (at the time of publication) for comparison. See 4.4 for a review of most of the methods cited in the table. Many recent works that report experimental results on MiniImagenet have used much more powerful feature extractors, often pretrained on vary large datasets [e.g. Rusu et al., 2019] and sometimes using higher resulution images [Sung et al., 2018]. These differences make it difficult to directly compare several results of the more recent literature with the one that we report here.

The proposed method achieves competitive results highlighting the relative im-

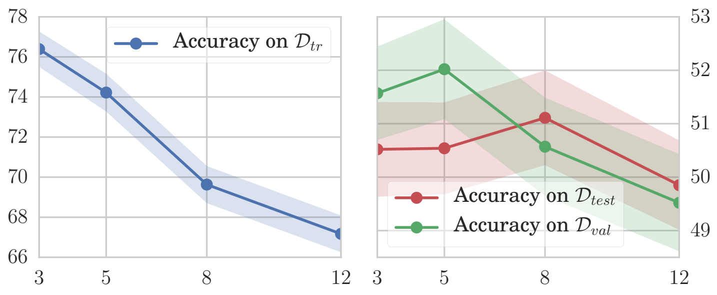

Figure 7.3: Meta-validation of the number of gradient descent steps (T) of the base-level models for MiniImagenet using the *RN* representation. Early stopping on the accuracy on meta-validation set during meta-training resulted in halting the optimization of λ after 42k, 40k, 22k, and 15k hyperiterations for T equal to 3, 5, 8 and 12 respectively; in line with our observation in Sec. 6.2.1.

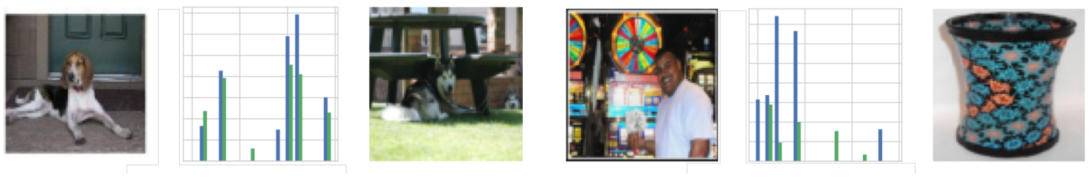

Figure 7.4: After sampling two datasets $D \in \mathbf{D}_{\mathrm{tr}}$ and $D' \in \mathbf{D}_{\mathrm{ts}}$, we show on the top the two images $x \in D$, $x' \in D'$ that minimize $\|r_\lambda(x) - r_\lambda(x')\|$ and on the bottom those that maximize it. In between each of the two pairs we compare a random subset of components of $r_\lambda(x)$ (blue) and $r_\lambda(x')$ (green).

portance of learning a task independent representation, on the top of which logistic classifiers trained with very few samples can perform, in average, well. Moreover, utilizing more expressive models such as residual network as representation mappings, is beneficial for our proposed strategy. Indeed, compared to *C4L*, *RN* achieves a relative improvement of 6.5% on one-shot and 4.2% on five-shot. Figure 7.4 provides a visual example of the goodness of the learned representation, showing that MiniImagenet examples (the first from meta-training, the second from the meta-testing sets) from similar classes (different dog breeds) are mapped near each other by r_λ and, conversely, samples from dissimilar classes are mapped afar.

7.2.3 On Variants of Representation Learning Methods

In this section, we examine if there are benefits of learning a representation within the proposed bilevel framework compared to other possible approaches that involve an explicit factorization of a classifier as $s^j \circ r$. The representation mapping r is either pretrained or learned with different meta-learning algorithms. We focus on the problem of one-shot learning on MiniImagenet and we use *C4L* as architecture for the representation mapping. In all the experiments the base-level models s^j are multinomial logistic regressor as in Sec. 7.2.2, tuned with 5 steps of gradient descent. We ran the following experiments:

• *Multiclass*: the mapping $r : \mathcal{X} \to \mathbb{R}^{64}$ is given by the linear outputs before the softmax operation of a network[8] pretrained on the totality of examples contained in the training meta-dataset (600 examples for each of the 64 classes). In this setting, we found that using the second last layer or the output after the softmax yields worst results;

• *Bilevel-train*: we use a bilevel approach but, unlike in Sec. 7.2.1, we optimize the parameter vector λ of the representation mapping by minimizing the loss on the training sets of each episode. The hypergradient is still computed with Algorithm 5, albeit we set $D^j_{\text{ts}} = D^j_{\text{tr}}$ for each training episodes;

• *Approx* and *Approx-train*: we consider an approximation of the hypergradient $\nabla f_T(\lambda)$ by disregarding the optimization dynamics of the inner objectives (i.e. we set $\partial_\lambda w^j_T = 0$). In *Approx-train* we just use the training sets;

• *Classic*: as in [Baxter, 1995], we learn r by jointly optimize $\hat{f}(\lambda, w^1, \ldots, w^{\mathbf{N}}) = \sum_{j=1}^{\mathbf{N}} L^j(w^j, \lambda, D^j_{\text{tr}})$ and treat the problem as standard multitask learning, with the exception that we evaluate $\hat{f}$ on mini-batches of 4 episodes, randomly sampled every 5 gradient descent iterations.

Table 7.5: Performance of various methods where the representation is either transfered or learned with variants of hyper-reprefsentation methods. The last raw reports, for comparison, the score obtained with hyper-representation.

Method	# filters	Accuracy 1-shot
Multiclass	64	43.02
Bilevel-train	32	29.63
Approx	32	41.12
Approx-train	32	38.80
Classic-train	32	40.46
Hyper-representation-C4L	32	47.51

In settings where we do not use the test sets, we let the training sets of each episode contain 16 examples per class. Using training episodes with just one example per

[8]The network is similar to C4L but has 64 filters per layer.

class resulted in performances just above random chance. While the first experiment constitutes a standard baseline, the others have the specific aim of assessing (*i*) the importance of splitting episodes of meta-training set into training and test and (*ii*) the importance of computing the hypergradient of the approximate bilevel problem with Algorithm 5. The results reported in Table 7.5 suggest that both the training/test splitting and the full computation of the hypergradient constitute key factors for learning a good representation in a meta-learning context. On the other side, using pretrained representations, especially in a low-dimensional space, turns out to be a rather effective baseline, stronger than many reported prior to [Franceschi et al., 2018a]. One possible explanation is that, in this context, some classes in the training and testing meta-datasets are rather similar (e.g. various dog breeds) and thus base-level classifiers can leverage on very specific representations.

7.3 Discussion

In this chapter we demonstrated some practical aspects of the proposed bilevel programming framework for gradient-based hyperparameter optimization and meta-learning, instantiating it in four different learning settings. The framework, in practice, translates into a simple and direct operative pipeline that adapts well to the modern computational environments. Broadly speaking, the pipeline comprises the following steps:

1. determine inner and outer objectives with their respective decision variables (weights and hyper or meta parameters);
2. formulate the exact and the approximate programs (here we have focused on the iterative approach);
3. find solutions to the approximate problem by gradient descent on the outer objective, computing the hypergradient with the most appropriate method.

On the one hand, many existing learning and meta-learning algorithms may be recast into this framework and adapted to the pipeline, allowing us to efficiently tune real-valued hyperparameters or extracting common knowledge by optimizing the parameters of a meta-learner. On the other hand, the pipeline provides a solid ground, backed by the analysis of Chapter 6, on which to develop novel methods, especially in the meta-learning setting.

We have empirically shown, with several numerical simulations, that the resulting optimized standard or base-level learning algorithms compare favourably with case-specific methods and baselines. For HPO, we have presented experiments in both conventional (Section 7.2) and less conventional (Section 7.1.1) learning settings. For MTL we presented an adaptation of classic strategies of connectionist multi-task learning that adheres to the meta-learning paradigm presented in Section 4.2 and practically follows the above-mentioned pipeline. From a series of ablative experiments conducted in Section 7.2.3 we concluded that the two peculiar "innovations" of computing the gradient through the base-level learning algorithm (multinomial logistic regression, in our case[9]) and splitting the data during meta-training are paramount for the success of the MTL procedure.

[9]Bertinetto et al. [2019] report ablative experiments suited to their MTL algorithm that agree with our findings.

Part III

Extensions

Chapter 8

Online Hypergradients

This and the next chapter compose the third part of the thesis, where we study two extensions of the framework and algorithms presented in the second part.

In this chapter, we focus on the real-time hypergradient computation based on the forward mode, introduced in Section 5.4.2.1; see also experiments of Section 7.1.2. The RTHO method offers a compelling strategy to quickly tune few critical hyperparameters on-the-fly, generating entire hyperparameter schedules (Section 3.3). However, it is not directly clear how the updates produced by RTHO compare to the "exact iterative hypergradient", as computed by either Algorithm 2 or 3. Moreover, as we shall see in Section 8.2.1, RTHO can be sensitive to sudden changes in the instantaneous response surface, making it potentially unstable. We investigate these issues taking as a case study the problem of tuning task-specific learning rate schedules. Based on our findings, we introduce MARTHE (moving average real time hyperparameter estimation) in Section 8.3, a novel online algorithm guided by cheap approximations of the hypergradient that uses discounted past information from the optimization trajectory to simulate future behaviour. It interpolates between RTHO and another recently proposed technique called hypergradient descent (HD) [Baydin et al., 2018a]. We show empirically that MARTHE produces learning rate schedules that are more stable and lead to models that generalize better in a series of time-controlled comparative experiments with deep neural networks, which we report in Section 8.5.

This chapter is based on [Donini et al., 2020].

8.1 Introduction

Learning rate (LR) adaptation for first-order optimization methods is one of the most widely studied aspects in optimization for learning methods, in particular neural networks. Recent research in this area has focused on developing complex schedules that depend on a small number of hyperparameters [Loshchilov and Hutter, 2017, Orabona and Pál, 2016] or proposed methods to optimize LR schedules w.r.t. the training objective [Schaul et al., 2013, Baydin et al., 2018a, Wu et al., 2018b]. While quick optimization is desirable, the true goal of supervised learning is to minimize the generalization error, which is commonly estimated by holding out part of the available data for validation. Hyperparameter optimization (Chapter 3), a related but distinct branch of the literature, specifically focuses on this aspect, with less emphasis on the goal of rapid convergence on a single task. Additionally, in meta-learning, works in the area of learning to optimize (Section 4.4.2) have focused on the problem of tuning parameterized optimizers on whole classes of learning problems but require prior expensive optimization and are not designed to speed up training on a single task.

One of the goal of this chapter is to automatically compute *in an online fashion* a learning rate schedule for stochastic optimization methods (such as SGD) only on the basis of the given learning task, aiming at producing models with associated small validation error. We study the problem of finding a LR schedule under the framework of (iterative) gradient-based hyperparameter optimization (Section 5.3.1): we consider an optimal schedule $\eta^* = (\eta^*_0, \dots, \eta^*_{T-1}) \in \mathbb{R}^T_+$ as a solution to the following constrained optimization problem

$$\begin{aligned} &\min\{f_T(\eta) = E(w_T(\eta)) : \eta \in \mathbb{R}^T_+\} \\ \text{s.t.} \quad & w_0 = \bar{w}, \quad w_{t+1}(\eta) = \Phi_t(w_t(\eta), \eta_t) \quad t \in [T] \end{aligned} \tag{8.1}$$

where $[T] = \{0, \dots, T-1\}$, where $E : \mathbb{R}^d \to \mathbb{R}_+$ is an objective function, $\Phi_t : \mathbb{R}^d \times \mathbb{R}_+ \to \mathbb{R}^d$ is a (possibly stochastic) weight update dynamics, $\bar{w} \in \mathbb{R}^d$ represents the initial model weights (parameters) and finally w_t are the weights after t iterations. This problem is related to (5.1)-(5.2), although we focus here solely on the iterative view of Problem (8.1) (cf. Equations (5.8)-(5.9)), since in this chapter we are explicitly interested in tuning the LR schedule. Most of the arguments developed in the following are, however, easily adjustable to the more general setting where at each step one may consider an hyperparameter vector λ_t that can comprise e.g. (also) regularization or design coefficients (as in Section 7.1.2).

We can think of E as either the training or the validation loss of the model, while the dynamics Φ describe the update rule (such as SGD, SGD-Momentum, Adam etc.). For example in the case of SGD, $\Phi_t(w_t, \eta_t) = w_t - \eta_t \nabla L_t(w_t)$, with $L_t(w_t)$ the training loss on the t-th minibatch. The horizon T should be large enough so that the training error can be effectively minimized to avoid underfitting. A too large value of T does not necessarily harm since $\eta_k = 0$ for $k > \bar{T}$ is still a feasible solution, implementing early stopping in this setting. The learning rate is among the critical hyperparameters affecting the performances of learnt statistical models such as neural networks [Bengio, 2012, Bergstra and Bengio, 2012]. Beside convergence arguments from the stochastic optimization literature, for all but the simplest problems, non-constant schedules yield generally better results [Bengio, 2012].

Problem (8.1) can be in principle solved by any HPO technique. However, most HPO techniques, including those based on hypergradients and the general bilevel programming framework of Chapter 5, would not be suitable for the present purpose since they require multiple evaluations of f (which, in turn, require executions of the weight optimization routine). This clearly defeats one of the main goals of determining LR schedules, i.e. speed. In fact, several other researchers [Almeida et al., 1999, Schraudolph, 1999, Schaul et al., 2013, Franceschi et al., 2017, Baydin et al., 2018a, Wu et al., 2018b] have investigated related solutions for deriving greedy update rules for the learning rate. A common characteristic of methods in this family is that the LR update rule does not take into account information from the future. At a high level, we argue that any method should attempt to produce updates that approximate the true and computationally unaffordable hypergradient of the *final* objective with respect to the current learning rate. In relation to this, [Wu et al., 2018b] discuss the bias deriving from greedy or short-horizon optimization and [Micaelli and Storkey, 2020] recently proposed hyperparameter sharing (similar to the "hyper-batch" of Section 5.4.2.1) and long horizon hypergradient computation as a partial remedy. In practice, different methods resort to different approximations or explicitly consider greedily minimizing the performance after a single parameter update [Almeida et al., 1999, Schaul et al., 2013, Baydin et al., 2018a]. The type of approximation and the type of objective (i.e. the training or the validation loss) are in principle separate issues, although, in literature, there is a lack of reported comparative experiments with both objectives and the same approximation.

A second goal of this chapter, not limited to the exercise of tuning LR schedules, is to make a step forward in understanding the behavior of real-time gradient-based hyperparameter optimization techniques. In this regard, we analyze in Section 8.2 the structure of the true hypergradient that could be used to solve Problem (8.1) if wall-clock time was not a concern, and conduct some numerical experiments to gain further insight (Section 8.4). We then study in Section 8.2.1 some failure modes of previously proposed methods along with a detailed discussion of the type of approximations that these methods exploit. Based on these considerations, we develop a new hypergradient-based algorithm, which we call MARTHE (Moving Average Real-Time Hyperparameter Estimation). MARTHE has a moderate computational cost and can be interpreted as a generalization of RTHO (Section 5.4.2.1, [Franceschi et al., 2017]) and the algorithm described by Baydin et al. [2018a].

8.2 Structure of the Hypergradient

In this section we analyze the specific structure of the "exact iterative hypergradient" of Problem 8.1 where the hyperparameter vector given by the learning rate schedule $\eta = (\eta_0, \dots, \eta_{T-1})$ is treated as a vector of hyperparameters and T is a fixed horizon. Since the learning rates are positive real-valued variables, assuming both E and Φ are smooth functions, we can compute the gradient of $f \in \mathbb{R}^T$. Recall from Section 5.4.2, that the gradient is given by

$$\nabla f_T(\eta) = \partial_\eta w_T^\intercal \nabla E(w_T), \quad \partial_\eta w_T = \frac{\mathrm{d} w_T}{\mathrm{d}\eta} \in \mathbb{R}^{d\times T}, \tag{8.2}$$

The total derivative $\partial_\eta w_T$ can be computed iteratively with forward-mode algorithmic differentiation (Section A.3.1) as

$$\partial_\eta w_0 = 0, \quad \partial_\eta w_{t+1} = A_t \partial_\eta w_t + B_t, \tag{8.3}$$

$$\text{with} \qquad A_t = \frac{\partial \Phi_t(w_t, \eta_t)}{\partial w_t}, \quad B_t = \frac{\partial \Phi_t(w_t, \eta_t)}{\partial \eta}. \tag{8.4}$$

The Jacobian matrices A_t and B_t depend on w_t and η_t, but we will leave these dependencies implicit to ease our notation. In the case of SGD[1],

$$A_t = I - \eta_t H_t(w_t), \quad \text{and} \quad [B_t]_j = -\delta_{tj} \nabla L_t(w_t),$$

where subscripts denote columns (starting from 0), $\delta_{tj} = 1$ if $t = j$ and 0 otherwise and H_t is the Hessian of the training error $L_t : \mathbb{R}^d \to \mathbb{R}^+$ on the t−th mini-batch.

Given the high dimensionality of η, reverse-mode differentiation would result in a more efficient (running-time) implementation. We use here forward-mode both because it is easier to interpret and because it is closely related to the computational scheme behind MARTHE, as we will show in Section 8.3. We note that stochastic approximations of Equation (8.2) may be obtained with randomized telescoping sums [Beatson and Adams, 2019] or hyper-networks based stochastic approximations [MacKay et al., 2019].

Equation (8.3) describes the so-called tangent system [Griewank and Walther, 2008] which is a discrete affine time-variant dynamical system that measures how the parameters of the model would change for infinitesimal variations of the learning rate schedule, after t iterations of the optimization dynamics. Notice that the "translation matrices" B_t are very sparse, having, at any iteration, only one non-zero column. This means that $[\partial_\eta w_t]_j$ remains 0 for all $j \geq t$: η_t affects only the future parameters

[1]Throughout we use SGD to simplify the discussion, however, similar arguments hold for any smooth optimization dynamics such as those including momentum terms. In such case one must consider the extended dynamics that includes also the auxiliary variables, as derived in Section 5.4.

trajectory. Finally, for a single learning rate η_t, the derivative (a scalar) is

$$\frac{\partial f_T(\eta)}{\partial \eta_t} = [\nabla f_T(\eta)]_t = \left[\left(\prod_{s=t+1}^{T-1} A_s\right) B_t\right]_t^{\mathsf{T}} \nabla E(w_T) \tag{8.5}$$

$$= -\nabla L_t(w_t)^{\mathsf{T}} P_{t+1:T-1} \nabla E(w_T), \tag{8.6}$$

where the last equality holds true for SGD. Equation (8.5) can be read as the scalar product between the gradients of the training error at the t-th step and the objective E at the final iterate, *transformed by* the accumulated (transposed) Jacobians of the optimization dynamics, shorthanded by $P_{t+1:T-1}$ in (8.6). As it is apparent from (8.5), given w_t, the hypergradient of η_t is affected only by the future trajectory and does not depend explicitly on η_t.

In its original form, where each learning rate is left free to take any permitted value, Problem (8.1) presents a highly nonlinear setup. Although in principle it could be tackled by a projected gradient descent method, in practice this is not feasible even for relatively small problems: evaluating the gradient with forward-mode is inefficient in time since it requires maintaining a large matrix tangent system. Evaluating it with reverse-mode is inefficient in memory since the entire weight trajectory $(w_i)_{i=0}^{T}$ should be stored[2]. Furthermore, it can be expected that several updates of η are necessary to reach convergence – each update requiring the computation of f_T and the entire parameter trajectory in the weight space. Since this approach is computationally very expensive, we turn out attention to online updates where η_t is required to be updated online based only on trajectory information up to time t.

8.2.1 Online Gradient-Based Adaptive Schedules

Before developing and motivating our proposed technique, we discuss two previous methods to compute the learning rate schedule online. The real-time hyperparameter optimization (RTHO) algorithm [Franceschi et al., 2017] introduced in Section 5.4.2.1, reminiscent of stochastic meta-descent [Schraudolph, 1999], is based on forward-mode differentiation and uses information from the entire weight trajectory by accumulating

[2]Techniques based on implicit differentiation and fixed-point equations (see Section 5.4.3) cannot be readily applied to compute ∇f_T since the training loss L does not depend explicitly on η.

partial hypergradients. Hypergradient descent (HD), proposed in [Baydin et al., 2018a] and closely related to the earlier work by Almeida et al. [1999], aims at minimizing the loss with respect to the learning rate after one step of optimization. It uses information only from the past and current iterate.

Both methods implement an update rules of the type (cf. Equation (3.27))

$$\eta_t = \max\left[\eta_{t-1} - \beta \Delta \eta_t, 0\right],$$

where $\Delta \eta_t$ is an online estimate of the hypergradient, $\beta > 0$ is a step-size or *hyper-learning rate* and the max ensures positivity[3]. To ease the discussion, we omit the stochastic (mini-batch) evaluation of the training error L and possibly of the objective E.

The update rules[4] are given by

$$\Delta^{\mathrm{RTHO}} \eta_t = \left[\sum_{i=0}^{t-1} P_{i+1:t-1} B_i\right]^{\intercal} \nabla E(w_t); \tag{8.7}$$

$$\Delta^{\mathrm{HD}} \eta_t = B_{t-1}^{\intercal} \nabla E(w_t) \tag{8.8}$$

for RTHO and HD respectively, where $P_{t:t-1} := I$. Thus $\Delta^{\mathrm{RTHO}} = \Delta^{\mathrm{HD}} + r((w_i, \eta_i)_{i=0}^{t-2})$: the correction term r can be interpreted as an "on-trajectory approximations" of longer horizon objectives as we will discuss in Section 8.3.

Although successful in some learning scenarios, we argue that both these update rules suffer from (different) pathological behaviors, as HD may be "shortsighted", being prone to underestimate the learning rate (as noted by Wu et al. [2018b]), while RTHO may be too slow to adapt to sudden changes of the loss surface or, worse, it may be unstable, with updates growing uncontrollably in magnitude. We exemplify these behaviors in Figure 8.1, using two bidimensional test functions[5] from the optimization

[3]Updates could be also considered in the logarithmic space, e.g. by Schraudolph [1999]; we find it useful, to let η reach 0 whenever needed, offering a natural way to implement early stopping.

[4] In [Franceschi et al., 2017], the hyperparameter is updated every K iterations. Here we focus on the case $K = 1$ which better allows for a unifying treatment. HD is developed using as objective the training loss L rather than the validation loss E. We consider here without loss of generality the case of optimizing E.

[5]We use the Beale function defined as $L(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$

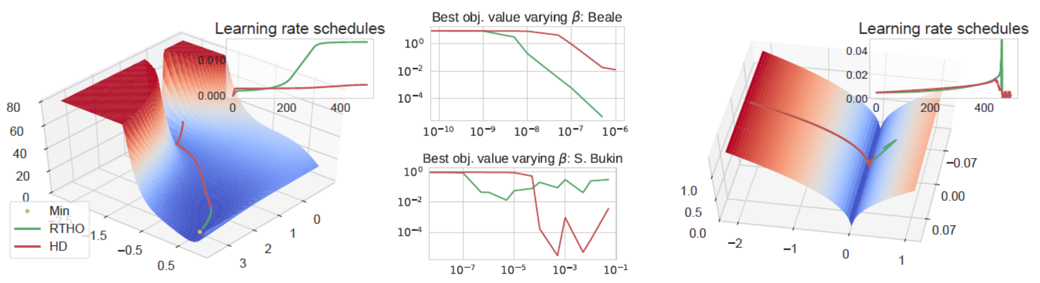

Figure 8.1: Loss surface and trajectories for 500 steps of gradient descent with HD and RTHO for Beale function (left) and (smoothed and simplified) Bukin N.6 (right). Center: best objective value reached within 500 iterations for various values of β that do not lead to divergence.

literature, where we set $E = L$ and we perform 500 steps of gradient descent. The Beale function, on the left, presents sharp peaks and large plateaus. RTHO consistently outperforms HD for all probed values of β that do not lead to divergence (Figure 8.1 upper center). This can be easily explained by the fact that in flat regions gradients are small in magnitude, leading to $\Delta^{\mathrm{HD}}\eta_t$ to be small as well. RTHO, on the other hand, by accumulating all available partial hypergradients and exploiting second order information, is capable of making faster progress. We use a simplified and smoothed version of the Bukin function N.6 to show the opposite scenario (Figure 8.1 lower center and right). Once the optimization trajectory closes the valley of minimizers $y = 0.01x$, RTHO fails to discount outdated information, bringing the learning rate first to grow exponentially, and then to suddenly vanish to 0, as the gradient changes direction. HD, on the other hand, correctly damps η and is able to maintain the trajectory close to the valley.

These considerations suggest that neither Δ^{RTHO} nor Δ^{HD} provide *globally useful* update directions, as large plateaus and sudden changes on the loss surface are common features of the optimization landscape of neural networks [Bengio et al., 1994, Glorot and Bengio, 2010]. Our proposed algorithm smoothly interpolates between these two methods, as we will present next.

and a simplified smoothed version of Buking N.6: $L(x,y) = \sqrt{((y-0.01x)^2+\varepsilon)^{1/2}+\varepsilon}$, with $\varepsilon > 0$.

8.3 MARTHE

In this section, we develop and motivate MARTHE, which we instantiate for the case of tuning LR schedules on-the-fly, during a single training run. This method maintains a moving-average over approximations of (8.5) of increasingly longer horizon, using the past trajectory and gradients to retain a low computational overhead. Further, we show that RTHO [Franceschi et al., 2017] and HD [Baydin et al., 2018a] outlined above, can be interpreted as special cases of MARTHE, shedding further light on their behaviour and shortcomings.

Shorter horizon auxiliary objectives. For $K > 0$, define $g_K(u,\xi)$, with $\xi \in \mathbb{R}_+^K$ as

$$g_K(u,\xi) = E(u_K(\xi)) \tag{8.9}$$

$$\text{s.t.} \quad u_0 = u, \quad u_{i+1} = \Phi(u_i, \xi_i) \;\text{ for }\; i = [K]. \tag{8.10}$$

The g_Ks define a class of shorter horizon objective functions, indexed by K, which correspond to the evaluation of E after K steps of optimization, starting from $u \in \mathbb{R}^d$ and using ξ as the LR schedule[6]. Now, the derivative of g_K with respect to ξ_0, denoted g'_K, is given by

$$g'_K(u,\xi) = \frac{\partial g_K(u,\xi)}{\partial \xi_0} = [B_0]_0^\intercal P_{1:K-1} \nabla E(u_K) \tag{8.11}$$

$$= -\nabla L(u)^\intercal P_{1:K-1} \nabla E(u_K), \tag{8.12}$$

where the last equality holds for SGD dynamics. Once computed on subsets of the original optimization dynamics $(w_i)_{i=0}^T$, the derivative reduces for $K = 1$ to

$$g'_1(w_t, \eta_t) = -\nabla E(w_{t+1}) \nabla L(w_t)^\intercal,$$

assuming SGD dynamics, and for $K = T - t$ to

$$g'_{T-t}(w_t, (\eta_i)_{i=t}^{T-1}) = [\nabla f(\eta)]_t .$$

[6] Formally, ξ and u are different from η and w from the previous sections; later, however, we will evaluate the g_K's on subsequences of the optimization trajectory.

Intermediate values of K yield cheaper, shorter horizon approximations of (8.5).

Approximating the future trajectory with the past. Explicitly using any of the approximations given by $g'_K(w_t,\eta)$ as $\Delta\eta_t$ is, however, still largely impractical, especially for $K \gg 1$. Indeed, it would be necessary to iterate the map Φ for K steps (in the future), with the resulting $(w_{t+i})_{i=1}^{K}$ iterations discarded after a single update of the learning rate. For $K \in [t]$, we may then consider evaluating g'_K *exactly K steps in the past*, that is evaluating $g'_K(w_{t-K},(\eta_i)_{i=t-K}^{t-1})$. Selecting $K = 1$ is indeed equivalent to Δ^{HD}, which is computationally inexpensive. However, when past iterates are close to future ones (such as in the case of large plateaus), using larger K's would allow us, in principle, to capture longer horizon dependencies present in the hypergradient structure of (8.5). Unfortunately the computational efficiency of $K = 1$ does not generalize to $K > 1$, since setting $\Delta\eta_t = g'_K$ would require maintaining K different tangent systems.

Discounted accumulation of g'_ks. The definition of the g_Ks, however, allows one to highlight the recursive nature of the *accumulation* of g'_K. Indeed, by maintaining the vector tangent system,

$$Z_0 = [B_0(u_0,\xi_0)]_0 \tag{8.13}$$

$$Z_{i+1} = \mu A_i(u_i,\xi_i)Z_i + [B_i(u_i,\xi_i)]_i \text{ for } i \geq 0, \tag{8.14}$$

with $Z_i \in \mathbb{R}^d$, computing the moving average

$$S_{K,\mu}(u,\xi) = \sum_{i=0}^{K-1} \mu^{K-1-i} g'_{K-i}(u_i,(\xi_j)_{j=i}^{K-1}) = Z_K^{\intercal}\nabla E(u_K) \tag{8.15}$$

from S_{K-1} requires only updating (8.14) and recomputing the gradient of E. We note that (8.15) is reminescent of the GDM update (2.24). The total cost of this operation is $O(c(\Phi))$ per step both in time and memory using fast Jacobians vector products [Pearlmutter, 1994] where $c(\Phi)$ is the cost of computing the optimization dynamics (typically $c(\Phi) = O(d)$). The parameter $\mu \in [0,1]$ allows us to control how quickly past history is forgotten. One can notice that

$$\Delta^{\mathrm{RTHO}}\eta_t = S_{t,1}(w_0,(\eta_j)_{i=0}^{t-1}),$$

Algorithm 6 **MARTHE**; requires β, μ, $\eta_0[=0]$

```
Initialization of $w$ and $Z_0 \leftarrow 0$
for $t = 0$ to $T$ do
    $\eta_t \leftarrow \max[\eta_{t-1} - \beta\Delta\eta_t, 0]$    {Update LR if $t > 0$}
    $Z_{t+1} \leftarrow \mu A_t(w_t, \eta_t)Z_t + [B_t(w_t, \eta_t)]_t$    {Equation (8.14)}
    $w_{t+1} \leftarrow \Phi_t(w_t, \eta_t)$    {Parameter update}
end for
```

while $\mu = 0$ recovers

$$\Delta^{\mathrm{HD}}\eta_t = S_{t,0}(w_0, (\eta_j)_{i=0}^{t-1}) = g_1'(w_{t-1}, \eta_{t-1}).$$

Values of $\mu < 1$ help discount outdated information, while as μ increases so does the horizon of the hypergradient approximations. The computational scheme of (8.13) is quite similar to that of forward-mode algorithmic differentiation for computing $\partial_\eta w$ (see Section 8.2 and Equation (8.3)); however, the "tangent system" in (8.13), exploiting the sparsity of the matrices B_t, only keeps track of the variations with respect to the first component ξ_0, drastically reducing the running time.

Algorithm 6 presents the pseudocode of MARTHE, which uses $S_{t,\mu}$ as learning rate update at the t-th iteration, where μ is a configuration parameter. The runtime and memory requirements of the algorithm are dominated by the computation of the variables Z. Being these structurally identical to the tangent propagation of forward mode algorithmic differentiation for scalar function, we conclude that the runtime complexity is only a multiplicative factor higher over the cost of the underlying optimization dynamics Φ and requires two times the memory (see [Griewank and Walther, 2008, Ch. 4] and Section A.4).

8.4 Optimized Schedules and Quality of MARTHE Approximations

In this section, we empirically compare the optimized LR schedules found by approximately solving Problem 8.1 by gradient descent (denoted LRS-OPT), where the

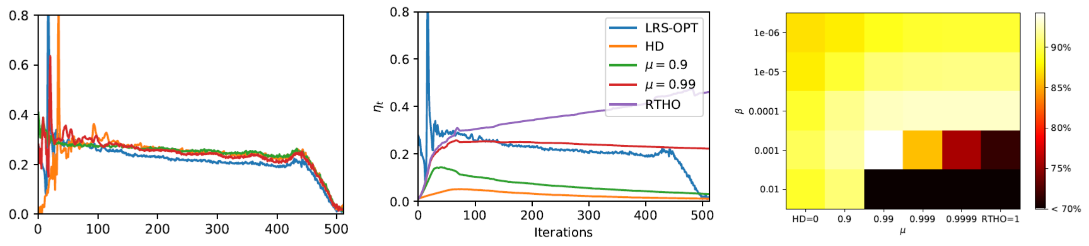

Figure 8.2: Left: schedules found by LRS-OPT (after 500 iterations of SGD) on 4 different random seeds. Center: comparison between optimized and MARTHE schedules for one seed, for indicative values of μ. We report the schedule generated with the hyper-learning rate β that achieves the best final validation accuracy. Right: Average validation accuracy of MARTHE over 20 random seeds, for various values of β and μ. The best performance of 94.2% is obtained with $\mu = 0.99$. For reference, the average validation accuracy of the network trained with $\eta = 0.01 \cdot \mathbf{1}_{512}$ is 87.5%, while LRS-OPT obtains an average accuracy of 96.1%. For $\mu \in [0.9, 1)$, when MARTHE converges it consistently outperforms HD and it performs at least as well as RTHO, but converges for a wider range of β.

hypergradient is given by (8.5), against those generated by MARTHE, for a wide range of hyper-momentum factors μ (including HD and RTHO) and hyper-learning rates β. We are interested in understanding and visualizing the qualitative similarities among the schedules, as well as the effect of μ and β on the final performance measure. To this end, we trained three-layers feed forward neural networks with 500 hidden units per layer on a subset of 7000 MNIST [LeCun et al., 1998] images. We used a cross-entropy loss and SGD as optimization dynamics Φ, with a mini-batch size of 100. We further sampled 700 images to form the validation set and defined E to be the validation loss after $T = 512$ optimization steps (about 7 epochs). For LRS-OPT, we randomly generated different mini-batches at each iteration, to prevent the schedule from unnaturally adapting to a specific progression of mini-batches[7]. We initialized $\eta = 0.01 \cdot \mathbf{1}_{512}$ for LRS-OPT and set $\eta_0 = 0.01$ for MARTHE, and repeated the experiments for 4 random seeds.

Figure 8.2 (left) shows the LRS-OPT schedules found after 5000 iterations of gradient descent: the plot reveals a strong initialization (random seed) specific behavior of η^* for approximately the first 100 steps. The LR schedule then stabilizes or slowly decreases up until around 50 iterations before the final time, at which point it quickly

[7]We retained, however, the random initialization of the network weights, to account for the impact that this may have on the initial part of the trajectory. This offers a fairer comparison between LRS-OPT and online methods, which compute the trajectory only once.

decreases (recall that, in this setting, all η_i, including η_0, are optimized "independently" and may take any permitted value). Figure 8.2 (center) present a qualitative comparison between the offline LRS-OPT schedule and the online ones, for indicative values of μ. Too small values of μ result in an underestimation of the learning rates, with generated schedules that quickly decay to very small values – this is in line with what observed in [Wu et al., 2018b]. For too high values of μ ($\mu = 1$ i.e. RTHO [Franceschi et al., 2017] in the figure) the schedules linger or fail to decrease, possibly causing instability and divergence. For certain values of μ, the schedules computed by MARTHE seems to capture the general behaviour of the optimized ones. Finally, Figure 8.2 (right) shows the average validation accuracy (rather than loss, for easier interpretation) of MARTHE methods varying β and μ. Higher values of μ translate to higher final performances – with a clear jump occurring between $\mu = 0.9$ and $\mu = 0.99$ – but may require a smaller hyper learning rate to prevent divergence.

8.5 Experiments[8]

We performed an extensive set of experiments in order to compare MARTHE, RTHO, and HD. We also considered a classic LR scheduling baseline in the form of exponential decay (Exponential) where the LR schedule is defined by $\eta_t = \eta_0 \gamma^t$. The purpose of these experiments is to perform a thorough comparison of various learning-rate scheduling methods, with a focus on those that are (hyper-)gradient-based, in the fairest possible manner: indeed, these methods have very different running-time per iteration – HD and Exponential being much faster than MARTHE and RTHO – as well as different configuration spaces. It would be unfair to compare them using the number of iterations as computational budget. We therefore designed an experimental setup that allowed us to account for it: we implemented a random search strategy over the respective algorithms' configuration spaces and early-stopped each run with a 10-epochs patience window. We repeatedly drew configurations parameters (hyper-hyperparameters) and run respective experiments until a fixed time budget of 36 hours was reached. The proposed experimental setting tries to mimic how machine learning

[8]The experiments reported in this section were performed by my coauthors of [Donini et al., 2020]. We include them here for completion.

practitioners may approach the parameter-tuning problem.

We used two alternative optimization dynamics: SGDM with the momentum hyperparameter fixed to 0.9 and Adam with the commonly suggested default values $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We fixed the batch size to 128, the initial learning rate $\eta_0 = 0.1$ for SGDM and 0.003 for Adam, and the weight decay to $5 \cdot 10^{-4}$. For the adaptive methods, we sampled β in $[10^{-3}, 10^{-6}]$ log-uniformly, and for our method, we sampled μ between 0.9 and 0.999. Finally, we picked the decay factor γ for Exponential log-uniformly in $[0.9, 1]$.

In our first set of experiments, we used VGG-11 [Simonyan and Zisserman, 2015] with batch normalization [Ioffe and Szegedy, 2015] after the convolutional layers, on CIFAR10 [Krizhevsky et al., 2014], and SGDM as the inner optimizer. In the second set of experiments, we used ResNet-18 [He et al., 2016] on CIFAR100 [Krizhevsky et al., 2014], in this case with Adam. This choice of models and learning settings has been made with the goal of striking a balance between diversity and computational time, so that in the first setting we could obtain in the 36 hours of budget a sizable number of runs with all the methods. The second setting instead uses a more recent model, which is possibly closer to practical applications. For both CIFAR10 and CIFAR100, we used 45000 images as training images and 5000 images as the validation dataset. An additional set of 10000 test images was finally used to estimate generalization accuracy[10]. We used standard data augmentation, including mean-std normalization, random crops and horizontal flips. Gradients were clipped to an absolute value of 100.0.

We kept track of the model with the best validation accuracy found so far, reporting in Figure 8.3 (left and center) the relative mean test accuracy (solid line) and

[10] Please note that the experiment reported in an earlier version of the reference work of this chapter, published on ArXiv on the 18th of October 2019 with the title *"Scheduling the Learning Rate via Hypergradients: New Insights and a New Algorithm"* (`https://arxiv.org/abs/1910.08525v1`), featured a different experimental protocol, whereby only two splits were used. Learning rate schedules were tuned on the second split, and the accompanying plots reported accuracy scores and errors on the same second split. We acknowledge that such practice may potentially hide overfitting issues and may unfairly advantage methods that exploit more information form that dataset, such as gradient-based one. The experiments reported in the final version of the reference paper [Donini et al., 2020] and in this section follow, instead, the more standard protocol of partitioning the data in training, validation and test splits.

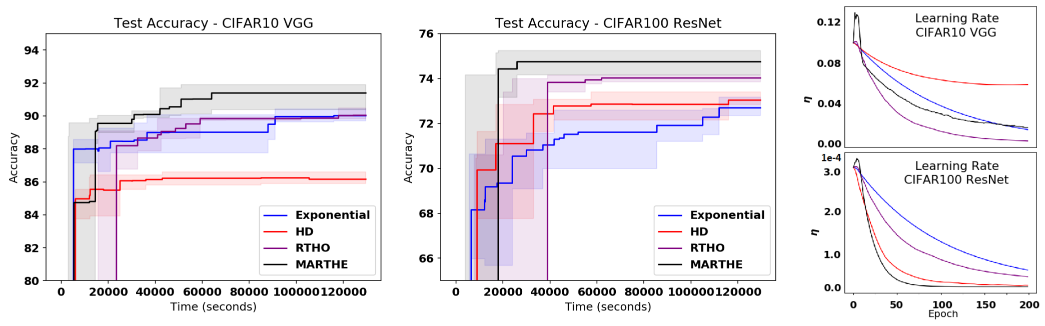

Figure 8.3: Left and center: we randomly draw parameters from each algorithm's configuration space (hyper-hyperparameters) and run the resulting experiments using early stopping with a patience window of 10 epochs. We keep track of the best model (i.e. the model with the highest validation accuracy) found so far and we report the relative test accuracy as a function of time. The solid line represents average accuracy, while shaded regions depict minimum and maximum accuracy across different seeds. On the left we show results for the experiments with VGG networks trained on CIFAR10 with SGDM as inner optimization method. The center plot reports experiments with ResNet models trained on CIFAR100 with Adam as optimization dynamics. Right: samples of learning rate schedules that lead to the best found model for each scheduling method and for the relative seed. Experiments on CIFAR10 (top) and CIFAR100 (bottom). Experiments conducted by my coauthors of [Donini et al., 2020] and included for completion.

minimum and maximum (limits of the shaded regions) across 5 repetitions. Inspecting the figure, it is possible to identify which method is the best performing one for any give time budget, both in average and in the worst/best case scenario. Figures 8.3 (right) shows examples of the LR schedules obtained by using the different methods.

We performed all experiments using AWS P3.2XL instances, each providing one NVIDIA Tesla V100 GPU. Finally, our PyTorch implementation of the methods and the experimental framework to reproduce the results is available at `https://github.com/awslabs/adatune`.

8.5.1 Analysis of Results

We will mainly focus on the accuracy on the test dataset achieved by different methods within a fixed time budget. For all the experiments, results summarized in Figure 8.3 show that both Exponential and HD were able to obtain a reasonably good accuracy within the first 4 hours, while RTHO and MARTHE required 6 hours at least to reach the same level of accuracy. This is due to the fact that the wall-clock time required to process a single minibatch is different: MARTHE takes approximately 4 times the wall-clock time of HD; there is negligible wall-clock time difference between MARTHE and RTHO or between HD and Exponential. MARTHE was able to surpass all the other methods consistently after 10 hours.

Our experimental protocol resulted in HD and Exponential getting more trials compared to RTHO and MARTHE (in average around 24 trials for the first two compared to 8 of RTHO and MARTHE). Despite the fact that MARTHE could only afford fewer trials, it could still achieve better performance, suggesting that it is able to produce better learning rate schedules more reliably. Moreover, MARTHE maintains a better peak accuracy compared to RTHO showing the effectiveness of down-weighting outdated information.

Our experimental setup helped us investigate the robustness of the methods with respect to the choice of the hyper-hyperparameters[13]. To that end, we can see from Figure 8.3 (left) that the average and worst case test set accuracies (measured across multiple seeds) of MARTHE are better in comparison to the other methods. This is a strong indication that MARTHE demonstrated superior adaptability with respect to different hyper-hyperparameters and seeds compared to other methods. This is also reflected by the result that MARTHE outperforms other strategies on a consistent basis if given a sufficient time budget (4-6 hours in our experiments): the higher computational cost of MARTHE is outbalanced by the fact that it needs fewer trials to reach the same performance level of faster methods like Exponential or HD.

Overall, our experiments reveal that RTHO and MARTHE provide better performance, giving a clear indication of the importance of the past information. Due to its lower computational overhead, Exponential should be still preferred under tight budget constraints, while MARTHE with $\mu \in [0.9, 0.999]$ should be preferred if enough time is available.

[13]We note that it is not common in the existing literature to mention the necessity of tuning hyper-hyperparameters for adaptive learning-rate methods, although different datasets and/or networks may require some tuning that may strongly affect the results.

8.6 Discussion

Finding a good learning rate schedule is an old but crucially important issue in machine learning. Here, we makes a step forward, analyzing previously proposed online gradient-based methods and introducing a more general technique to obtain performing LR schedules based on an increasingly long moving average over hypergradient approximations. MARTHE interpolates between HD and RTHO, and its implementation is fairly simple with modern automatic differentiation tools, adding only a moderate computational overhead over the underlying optimizer complexity.

In this chapter, we studied the case of optimizing the learning rate schedules; we note, however, that MARTHE is a general technique for finding online hyperparameter schedules (albeit its runtime scales linearly with the number of hyperparameters), possibly implementing a competitive alternative in other application scenarios, such as tuning regularization parameters. Interestingly, for that case of regularization, we note that the method proposed by Luketina et al. [2016] (called $T_1 - T_2$) is based on the same one-step approximation of HD [Baydin et al., 2018a]. Hence MARTHE would also generalize $T_1 - T_2$ in that setting.

As the RTHO method, presented in Section 5.4.2.1, is a special case of MARTHE with hyper-momentum factor $\mu = 1$, our derivation of Section 8.3 sheds also some lights in understanding the type of approximation that derives from computing the hypergradients in real-time. Specifically, we characterized the algorithm as the accumulation of the gradients of shifted shorter-horizon approximations of the outer objective. Unfortunately, it appears difficult to quantify exactly the amount of error introduced by these approximations. Nevertheless, MARTHE (and RTHO) updates allow reducing the computational complexity by orders of magnitude (either in time w.r.t. `Forward-HG` or in memory w.r.t. `Reverse-HG`), while still providing useful information that we found empirically superior to simpler, first-order (one-step) approximations. By discounting past and possibly outdated trajectory information, MARTHE reveals also more stable than RTHO, finding good schedules in a shorter amount of time when configuration parameters for both methods are randomly drawn from a very simple prior distribution.

Chapter 9

Learrning Discrete Structures

In this last content chapter, we extend the range of applicability of the gradient-based bilevel framework to a class of discrete hyperparameters (outer variables). We develop a variant of `Reverse-HG` (Section 5.4.1) that computes gradient estimates w.r.t. the parameters of a discrete probability distribution over which we define an appropriate outer objective.

We take as case study the problem of learning graph edges for node-level classification with graph neural networks (GNNs). GNNs are a popular class of machine learning models whose major advantage is their ability to incorporate a sparse and discrete dependency structure between data points. Unfortunately, GNNs can only be used when such a graph-structure is available. In practice, however, real-world graphs are often noisy and incomplete or might not be available at all. With this work, we propose to jointly learn the graph structure and the parameters of graph convolutional networks (GCNs) by approximately solving a bilevel program that learns a discrete probability distribution on the edges of the graph (Section 9.3). This allows one to apply GCNs not only in scenarios where the given graph is incomplete or corrupted but also in those where a graph is not available. In section 9.4, we report a series of experiments that analyze the behavior of the proposed method and demonstrate that it outperforms related methods by a significant margin.

This chapter is based on [Franceschi et al., 2019]. Alongside [Jiang et al., 2019], which was published concurrently to [Franceschi et al., 2019], the work reported here is one of the first attempts to simultaneously learn the graph and the parameters of a

GNN for semi-supervised (transductive) classification.

9.1 Introduction

Relational learning is concerned with methods that cannot only leverage the attributes of data points but also their relationships. Diagnosing a patient, for example, not only depends on the patient's vitals and demographic information but also on the same information about their relatives, the information about the hospitals they have visited, and so on. Relational learning, therefore, does not make the assumption of independence between data points but models their dependency explicitly. Graphs are a natural way to represent relational information and there is a large number of learning algorithms leveraging graph structure. Graph neural networks (GNNs) [Scarselli et al., 2009] are one such class of algorithms that are able to incorporate sparse and discrete dependency structures between data points.

While a graph structure is available in some domains, in others it has to be inferred or constructed. A possible approach is to first create a k-nearest neighbor (kNN) graph based on some measure of similarity between data points. This is a common strategy used by several learning methods such as LLE [Roweis and Saul, 2000] and Isomap [Tenenbaum et al., 2000]. A major shortcoming of this approach, however, is that the efficacy of the resulting models hinges on the choice of k and, more importantly, on the choice of a suitable similarity measure over the input features. In any case, the graph creation and parameter learning steps are independent and require heuristics and trial and error. Alternatively, one could simply use a kernel matrix to model the similarity of examples implicitly at the cost of introducing a dense dependency structure.

In this chapter, we follow a different route with the aim of learning *discrete* and *sparse* dependencies between data points while simultaneously training the parameters of graph convolutional networks (GCN), a class of GNNs. Intuitively, GCNs learn node representations by passing and aggregating messages between neighboring nodes [Kipf and Welling, 2017, Monti et al., 2017, Gilmer et al., 2017, Hamilton et al., 2017, Duran and Niepert, 2017, Velickovic et al., 2018]. We propose to learn a generative probabilistic model for graphs, samples from which are used both during training and at prediction time. Edges are modelled with random variables whose parameters are treated as hyperparameters in a bilevel learning framework (Chapter 5, [Franceschi et al., 2018a]). We iteratively sample the structure while minimizing an inner objective (a training error) and optimize the edge distribution parameters by minimizing an outer objective (a validation error).

9.2 Background

We first provide some background on graph theory and graph convolutional networks which complements the introduction to neural network models of Section 2.3.

9.2.1 Graph Theory Basics

A graph G is a pair (V,E) with $V=\{v_1,...,v_N\}$ the set of vertices and $E \subseteq V \times V$ the set of edges. Let N and M be the number of vertices and edges, respectively. Each graph can be represented by an adjacency matrix A of size $N \times N$: $A_{i,j}=1$ if there is an edge from vertex v_i to vertex v_j, and $A_{i,j}=0$ otherwise. The graph Laplacian is defined by $L=D-A$ where $D_{i,i}=\sum_j A_{i,j}$ and $D_{i,j}=0$ if $i \neq j$. We denote the set of all $N \times N$ adjacency matrices by $\mathcal{M}_N$.

9.2.2 Graph Convolutional Neural Networks

Graph neural networks are a popular class of machine learning models for graph-structured data. We focus specifically on graph convolutional networks (GCNs) and their application to semi-supervised learning. All GNNs have the same two inputs. First, a feature matrix $X \in \mathcal{X}_N \subset \mathbb{R}^{N \times n}$ where n is the number of different node features, second, a graph $G=(V,E)$ with adjacency matrix $A \in \mathcal{H}_N$. Given a set of class labels $\mathcal{Y}$ and a labeling function $y: V \to \mathcal{Y}$ that maps (a subset of) the nodes to their true class label, the objective is, given a set of training nodes V_{tr}, to learn a function

$$h_w : \mathcal{X}_N \times \mathcal{M}_N \to \mathcal{Y}^N$$

by minimizing some regularized empirical loss

$$L(w,A) = \frac{1}{|V_{\text{tr}}|} \sum_{v \in V_{\text{tr}}} \ell(h_w(X,A)_v, y_v) + \Omega(w), \tag{9.1}$$

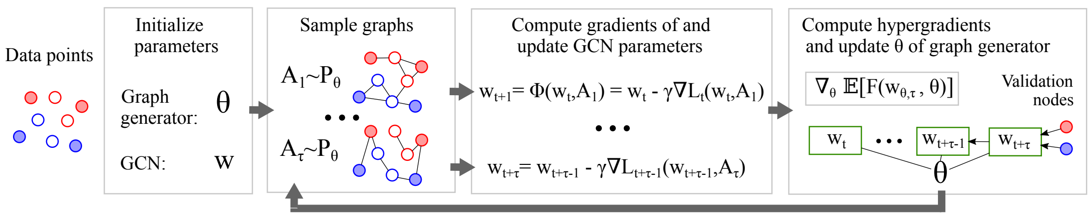
Data points
Initialize parameters
Graph generator: θ
GCN: w
Sample graphs
A1~Pθ
Aτ~Pθ
Compute gradients of and update GCN parameters
wt+1= Φ(wt,A1) = wt - γ∇Lt(wt,A1)
wt+τ= wt+τ-1 - γ∇Lt+τ-1(wt+τ-1,Aτ)
Compute hypergradients and update θ of graph generator
∇θ 𝔼[F(wθ,τ , θ)]
Validation nodes
wt
wt+τ-1
wt+τ
θ

Figure 9.1: Schematic representation of our approach for learning discrete graph structures for GNNs.

where $w \in \mathbb{R}^d$ are the parameters of h_w, $h_w(X,A)_v$ is the output of h_w for node v, $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ is a point-wise loss function, and Ω is a regularizer. An example of the function h_w proposed by Kipf and Welling [2017] is the following two hidden layer GCN that computes the class probabilities as

$$h_w(X,A) = \text{Softmax}(\hat{A}\ \text{ReLu}(\hat{A}\ X\ W_1)\ W_2), \tag{9.2}$$

where $w = (W_1, W_2)$ are the parameters of the GCN and $\hat{A}$ is the normalized adjacency matrix, given by $\hat{A} = \tilde{D}^{-1/2}(A+I)\tilde{D}^{-1/2}$, with diagonal, $\tilde{D}_{ii} = 1 + \sum_j A_{ij}$. GCNs for node-level classification may be seen as either transductive or inductive learning algorithms. Indeed, the training regime is typical of the transductive setting (Section 2.1), where the training set comprises also the inputs of the examples that one wishes to classify. However, once the the weights of the models are learnt, it may also make sense to compute predictions for another set of points with their respective adjacency matrix, provided that the this second graph with features concerns the same concept or phenomenon of the one used during training.

9.3 Learning Discrete Graph Structures

We address the challenging scenarios where a graph structure is either completely missing, incomplete, or noisy. To this end, we learn a *discrete* and *sparse* dependency structure between data points while simultaneously training the parameters of a GCN. We frame this as a bilevel programming problem whose outer variables are the parameters of a generative probabilistic model for graphs. The proposed approach, therefore, optimizes both the parameters of a GCN and the parameters of a graph generator so as to minimize the classification error on a given dataset. We developed a practical algorithm based on truncated reverse-mode algorithmic differentiation [Williams and Peng, 1990] and hypergradient estimation to approximately solve the bilevel problem. A schematic illustration of the resulting method is presented in Figure 9.1.

9.3.1 Jointly Learning the Structure and Parameters

Let us suppose that information about the true adjacency matrix A is missing or incomplete. Since, ultimately, we are interested in finding a model that minimizes the generalization error, we assume the existence of a second subset of instances with known target, V_{val} (the validation set), from which we can estimate the generalization error. Hence, we propose to find $A \in \mathcal{M}_N$ that minimizes the function

$$E(w(A), A) = \frac{1}{|V_{\text{val}}|} \sum_{v \in V_{\text{val}}} \ell(h_{w(A)}(X, A)_v, y_v), \tag{9.3}$$

where $w(A)$ is the minimizer, assumed unique, of L (see Equation (9.1)) for a *fixed adjacency matrix A*. We can then consider Equations (9.1) and (9.3) as the inner and outer objective of a mixed-integer bilevel programming problem where the outer objective aims to find an optimal discrete graph structure and the inner objective the optimal parameters of a GCN given a graph.

The resulting bilevel problem is intractable to solve exactly even for small graphs. Moreover, it contains both continuous and discrete-valued variables, which prevents us from directly applying Equation (5.25) or its iterative counterpart (5.19). A possible solution is to construct a continuous relaxation (see e.g. Frecon et al. [2018] for an application to group lasso, or Zheng et al. [2018] for estimating the structure of Bayesian networks), another is to work with parameters of a probability distribution over graphs. The latter is the approach we follow here. We maintain a generative model for the graph structure and reformulate the bilevel program in terms of the (continuous) parameters of the resulting distribution over discrete graphs. Specifically, we propose to model each edge with a Bernoulli random variable. Let $\overline{\mathcal{M}_N} = \text{Conv}(\mathcal{M}_N)$ be the convex hull of the set of all adjacency matrices for N nodes. By modeling all the possible edges as a set of mutually independent Bernoulli random variables with parameter matrix $\theta \in \overline{\mathcal{M}_N}$ we can sample graphs as $\mathcal{M}_N \ni A \sim \text{Ber}(\theta)$. Equations (9.1) and (9.3) can then be replaced by considering the expected loss over graph structures. The resulting bilevel problem becomes

$$\min_{\theta\in\overline{\mathcal{M}}_N} \mathbb{E}_{A\sim\mathrm{Ber}(\theta)}\left[E(w(\theta),A)\right] \tag{9.4}$$

$$\text{such that } w(\theta) = \arg\min_{w} \mathbb{E}_{A\sim\mathrm{Ber}(\theta)}\left[L(w,A)\right], \tag{9.5}$$

assuming that the inner problem has a unique solution for all $\theta \in \overline{\mathcal{M}}_N$. By taking the expectation, both the inner and the outer objectives become continuous (and smooth if E and L are smooth) functions of the Bernoulli parameters. The bilevel problem given by Equations (9.4)-(9.5) is still challenging to solve efficiently. This is because the solution of the inner problem is not available in closed form for GCNs (the objective is non-convex); and the expectations are intractable to compute exactly[1].

Before describing a method to solve the optimization problem given by Equations (9.4)-(9.5) approximately with hypergradient descent, we first turn to the question of obtaining a final GCN model that we can use for prediction. For a given distribution $p_A(\cdot|\theta)$ over graphs with N nodes and with parameters θ, the expected output of a GCN is

$$h_w^{\exp}(X) = \mathbb{E}_A[h_w(X,A)] = \sum_{A\in\mathcal{M}_N} p_A(\cdot|\theta) h_w(X,A). \tag{9.6}$$

Unfortunately, computing this expectation is intractable even for small graphs; we can, however, compute an empirical estimate of the output as

$$\hat{h}_w(X) = \frac{1}{S}\sum_{i=1}^{S} h_w(X,A_i), \tag{9.7}$$

where $S > 0$ is the number of samples we wish to draw. Note that $\hat{h}$ is an unbiased estimator of $h_w^{\exp}$. Hence, to use a GCN h_w learned with the bilevel formulation for prediction, we sample S graphs from the distribution $p_A(\cdot|\theta)$ and compute the prediction as the empirical mean of the values of h_w.

Given the parametrization of the graph generator with Bernoulli variables

[1] This is different from e.g. (model free) reinforcement learning, where the objective function is usually not observable, depending in an unknown way from the action and the environment.

($p_A(\cdot|\theta) = \text{Ber}(\theta)$), one can sample a new graph in $O(N^2)$. Sampling from a large number of Bernoulli variables, however, is highly efficient, trivially parallelizable, and possible at a rate of millions per second. Other sampling strategies such as MCMC sampling are possible in constant time. Given a set of sampled graphs, it is more efficient to evaluate a sparse GCN S times than to use the Bernoulli parameters as weights of the GCN's adjacency matrix[2]. Indeed, for GCN models, computing $\hat{h}_w$ has a cost of $O(SCd)$, rather than $O(N^2d)$ for a fully connected graph, where $C = \sum_{ij}\theta_{ij}$ is the expected number of edges, and d is the dimension of the weight vector. Another advantage of using a graph-generative model is that we can interpret it probabilistically which is not the case when learning a dense adjacency matrix.

9.3.2 Structure Learning via Hypergradient Descent

The bilevel programming formalism is a natural fit for the problem of learning both a graph generative model and the parameters of a GNN for a specific downstream task. Here, the outer variables θ are the parameters of the graph generative model and the inner variables w are the parameters of the GCN.

We now discuss a practical algorithm to approach the bilevel problem defined by Equations (9.4) and (9.5). Regarding the inner problem, we note that the expectation

$$\mathbb{E}_{A\sim\text{Ber}(\theta)}\left[L(w,A)\right] = \sum_{A\in\mathcal{M}_N} p_A(\cdot|\theta)L(w,A) \tag{9.8}$$

is composed of a sum in the order of 2^{N^2} terms, which is intractable even for relatively small graphs. We can, however, choose a tractable approximate learning dynamics Φ such as stochastic gradient descent (SGD, see also Section 2.4.2),

$$w_{t+1}(\theta) = \Phi(w_t(\theta), A_t) = w_t(\theta) - \gamma_t \nabla L(w_t(\theta), A_t), \tag{9.9}$$

where γ_t is a learning rate and $A_t \sim \text{Ber}(\theta)$ is drawn at each iteration. Under appropriate assumptions and for $t \to \infty$, SGD converges to a weight vector $w(\theta)$ that depends on the edges' probability distribution [Bottou, 2010].

[2]Note also that $\mathbb{E}h_w(X,A) \neq h_w(X,\mathbb{E}A) = h_w(X,\theta)$, as the model h_w is, in general, nonlinear.

Let $w_T(\theta)$ be an approximate minimizer of $\mathbb{E}[L]$ where T may depend on θ. We now need an estimator for the hypergradient $\nabla_\theta \mathbb{E}_{A\sim\mathrm{Ber}(\theta)}[E(w_T(\theta),A)]$. Let us first consider the more general case of estimating $\nabla_\theta \mathbb{E}_{z\sim p_z(\cdot|\theta)}[h(z)]$ for some distribution $z \sim p_z(\cdot|\theta)$ with parameters θ. If there exists a differentiable and reversible sampling path $\mathsf{sp}(\theta,\varepsilon)$ for $p_z(\cdot|\theta)$, with $z = \mathsf{sp}(\theta,\varepsilon)$ for $\varepsilon \sim p_\varepsilon$, then one can use the general form of the pathwise gradient estimator [see Mohamed et al., 2020, Sec. 5]:

$$\nabla_\theta \mathbb{E}_{z\sim p_z(\cdot|\theta)}[h(z)] = \mathbb{E}_{\varepsilon\sim p_\varepsilon}[\nabla_\theta h(\mathsf{sp}(\theta,\varepsilon))] = \mathbb{E}_{z\sim p_z(\cdot|\theta)}[\nabla_z h(z)\nabla_\theta z]. \tag{9.10}$$

Since we are concerned with discrete random variables, any sampling path would have discontinuities, making (9.10) not directly applicable. Nevertheless, by using an inexact but smooth reparameterization for $p_z(\cdot|\theta)$, we may employ an approximate version of (9.10) that allows us to derive a biased estimator of the gradient $\nabla\mathbb{E}[h]$. For $z = \mathsf{sp}(\theta,\varepsilon) = \theta$ the resulting gradient estimator is an instance of the class of straight-through estimators (STE) [Bengio et al., 2013].

Now, in our setting, we simply use the identity mapping $A = \mathsf{sp}(\theta,\varepsilon) = \theta$ and approximate

$$\nabla_\theta \mathbb{E}_{A\sim\mathrm{Ber}(\theta)}[E(w_T(\theta),A)] \approx \mathbb{E}_{A\sim\mathrm{Ber}(\theta)}[\nabla_A E(w_T(\theta),A)]. \tag{9.11}$$

The second line instantiates (9.10) since $\nabla_\theta A = \nabla_\theta \theta = \mathbf{I}$ with our choice of reparameterization for $p_A(\cdot|\theta)$. This allows us to both take discrete samples in the forward pass and to use an efficient (low variance) pathwise gradient estimator in the reverse pass. The cost of this operation is the introduction of a bias, as setting $A = \mathsf{sp}(\theta,\varepsilon) = \theta$ is not the same as sampling A from $\mathrm{Ber}(\theta)$ (see also Appendix D for further details on the STE). Recalling equation (5.19), we can further write $\mathbb{E}_{A\sim\mathrm{Ber}(\theta)}[\nabla_A E(w_T(\theta),A)]$ as

$$\mathbb{E}_A[\partial_w E(w_T(\theta),A)\nabla_A w_T(\theta) + \partial_A E(w_T(\theta),A)] \tag{9.12}$$

noting that $w_{\theta,T}$ depends on the distribution of A through the optimization dynamics (9.9).

Algorithm 7 LDS

```
1: Input data: $X$, $Y$, $Y'$[, $A$]
2: Input parameters: $\eta$, $\tau$[, $k$]
3: [$A \leftarrow$ kNN$(X,k)$]          {Initialize $A$ to $k$NN graph if $A = 0$}
4: $\theta \leftarrow A$          {Initialize $p_A(\cdot|\theta)$ as a deterministic distribution}
5: while Stopping condition is not met do
6:     $t \leftarrow 0$
7:     while Inner objective decreases do
8:         $A_t \sim \mathrm{Ber}(\theta)$          {Sample structure}
9:         $w_{\theta,t+1} \leftarrow \Phi_t(w_t(\theta), A_t)$          {Optimize inner objective}
10:        $t \leftarrow t+1$
11:        if $t = 0 \pmod{\tau}$ or $\tau = 0$ then
12:            $G \leftarrow$ computeHG$(E, Y, \theta, (w_{\theta,i})_{i=t-\tau}^{t})$
13:            $\theta \leftarrow \mathrm{Proj}_{\overline{\mathcal{M}}_N}[\theta - \eta G]$          {Optimize outer objective}
14:        end if
15:    end while
16: end while
17: return $w$, $p_A(\cdot|\theta)$          {Best found weights and prob. distribution}
```

Computing the hypergradient by fully unrolling the dynamics may be too expensive both in time and memory[3]. We propose to truncate the computation and estimate the hypergradient every τ iterations, where τ is a parameter of the algorithm. This is essentially an adaptation of truncated back-propagation through time [Werbos, 1990, Williams and Peng, 1990] and can be seen as a short-horizon optimization procedure with warm restart on w. A sketch of the method is presented in Algorithm 7. The procedure `computeHG` computes the hypergradient with the reverse mode (see Algorithm 2), sampling at each iteration a new adjacency matrix.

The algorithm contains stopping conditions at the outer and at the inner level. While it is natural to implement the latter with a decrease condition on the inner objective[4], we find it useful to implement the first with a simple early stopping criterion. A fraction of the examples in the validation set is held-out to compute, in each outer iteration, the accuracy using the predictions of the empirically expected model (9.7). The optimization procedure terminates if there is no improvement for some consecutive outer loops. This helps avoid overfitting the outer objective (9.4),

[3]Moreover, since we rely on biased estimations of the gradients, we do not expect to gain too much from a full computation.

[4]We continue optimizing L until $L(w_{t-1}, A)(1+\varepsilon) \geq L(w_t(\theta), A)$, for $\varepsilon > 0$ ($\varepsilon = 10^{-3}$ in the experiments). Since L is non-convex, we also use a patience window of p steps.

which may be a concern in this context given the quantity of (hyper)parameters being optimized and the relative small size of the validation sets (see also Section 3.6).

The hypergradients estimated with Algorithm 7 at each outer iteration are biased. The bias stems from both the straight-trough estimator and from the truncation procedure introduced in lines 11-13 [Tallec and Ollivier, 2017]. Nevertheless, we find empirically that the algorithm is able to make reasonable progress, finding configurations in the distribution space that are beneficial for the tasks at hand.

9.4 Experiments

We conducted a series of experiments with three main objectives. First, we evaluated LDS on node classification problems where a graph structure is available but where a certain fraction of edges is missing. Here, we compared LDS with graph-based learning algorithms including standard GCNs. Second, we wanted to validate our hypothesis that LDS can achieve competitive results on semi-supervised classification problems for which a graph is *not* available. To this end, we compared LDS with a number of existing semi-supervised classification approaches. We also compared LDS with algorithms that first create k-NN affinity graphs on the data set. Third, we analyzed the learned graph generative model to understand to what extent LDS is able to learn meaningful edge probability distributions even when a large fraction of edges is missing.

9.4.1 Datasets

Cora and Citeseer are two benchmark datasets that are commonly used to evaluate relational learners in general and GCNs in particular [Sen et al., 2008]. The input features are bag of words and the task is node classification. We use the same dataset split and experimental setup of previous work [Yang et al., 2016, Kipf and Welling, 2017]. To evaluate the robustness of LDS on incomplete graphs, we construct graphs with missing edges by randomly sampling 25%, 50%, and 75% of the edges. In addition to Cora and Citeseer where we removed all edges, we evaluate LDS on benchmark datasets that are available in scikit-learn [Pedregosa et al., 2011] such as Wine, Breast Cancer (Cancer), Digits, and 20 Newsgroup (20news). We take 10 classes from 20 Newsgroup and use words (TFIDF) with a frequency of more than 5% as features. We also use FMA, a dataset where 140 audio features are extracted from 7,994 music tracks and where the problem is genre classification [Defferrard et al., 2017].

9.4.2 Setup and Baselines

For the experiments on graphs with missing edges, we compare LDS to standard GCNs. In addition, we also conceived a method (GCN-RND) where we add randomly sampled edges at each optimization step of a standard GCN. With this method we intend to show that simply adding random edges to the standard training procedure of a GCN model (perhaps acting as a regularization technique) is not enough to improve the generalization.

When a graph is completely missing, GCNs boil down to feed-forward neural networks. Therefore, we evaluate different strategies to induce a graph on both labeled and unlabeled samples by creating (1) a sparse Erdős-Rényi random graph [Erdos and Rényi, 1960] (Sparse-GCN); (2) a dense graph with equal edge probabilities (Dense-GCN); (3) a dense RBF kernel on the input features (RBF-GCN); and (4) a sparse k-nearest neighbor graph on the input features (kNN-GCN). For LDS we initialize the edge probabilities using the k-NN graph (kNN-LDS). We further include a dense version of LDS where we learn a dense similarity matrix (kNN-LDS (dense)). In this setting, we compare LDS to popular semi-supervised learning methods such as label propagation (LP) [Zhu et al., 2003], manifold regularization (ManiReg) [Belkin et al., 2006], and semi-supervised embedding (SemiEmb) [Weston et al., 2012]. ManiReg and SemiEmb are given a k-NN graph as input for the Laplacian regularization. We also compare LDS to baselines that do not leverage a graph-structure such as logistic regression (LogReg), support vector machines (Linear and RBF SVM), random forests (RF), and feed-forward neural networks (FFNN). For comparison methods that need a kNN graph, $k \in \{2,3,\ldots,20\}$ and the metric (Euclidean or Cosine) are tuned using validation accuracy. For kNN-LDS, k is tuned from 10 or 20.

We use the two layers GCN given by (9.2) with 16 hidden neurons and ReLu activation. Given a set of labelled training instances V_{tr} (nodes or examples) we use the regularized cross-entropy loss

$$L(w,A) = -\sum_{v \in V_{\text{tr}}} y_v \circ \log\left[h_w(X,A)_v\right] + \rho \|w_1\|^2,$$

where y_v is the one-hot encoded target vector for the v-th instance, $\circ$ denotes the element-wise multiplication and ρ is a non-negative coefficient. As additional regularization technique we apply dropout [Srivastava et al., 2014] with $\beta = 0.5$ as in previous work. We use Adam [Kingma and Ba, 2015] for optimizing L, tuning the learning rate γ from {0.005, 0.01, 0.02}. The same number of hidden neurons and the same activation is used for SemiEmb and FFNN.

For LDS, we set the initial edge parameters $\theta_{i,j}$ to 0 except for the known edges (or those found by kNN) which we set to 1. We then let all the parameters (including those initially set to 1) to be optimized by the algorithm. We further split the validation set evenly to form the validation (A) and early stopping (B) sets. As outer objective we use the un-regularized cross-entropy loss on (A) and optimize it with stochastic gradient descent. with exponentially decreasing learning rate. Initial experiments showed that accelerated optimization methods such as Adam or SGD with momentum underperform in this setting. We tune the step size η of the outer optimization loop and the number of updates τ used to compute the truncated hypergradient. Finally, we draw $S = 16$ samples to compute the output predictions (see Equation (9.7)). For LDS and GCN, we apply early stopping with a window size of 20 steps.

LDS was implemented in TensorFlow [Abadi et al., 2015] and is available at `https://github.com/lucfra/LDS`. The implementations of the supervised baselines and LP are those from the scikit-learn python package [Pedregosa et al., 2011]. GCN, ManiReg, and SemiEmb are implemented in Tensorflow. The hyperparameters for all the methods are selected through grid search optimizing for the validation accuracy.

9.4.3 Results

The results on the incomplete graphs are shown in Figure 9.2 for Cora (left) and Citeseer (center). For each percentage of retained edges the accuracy on validation

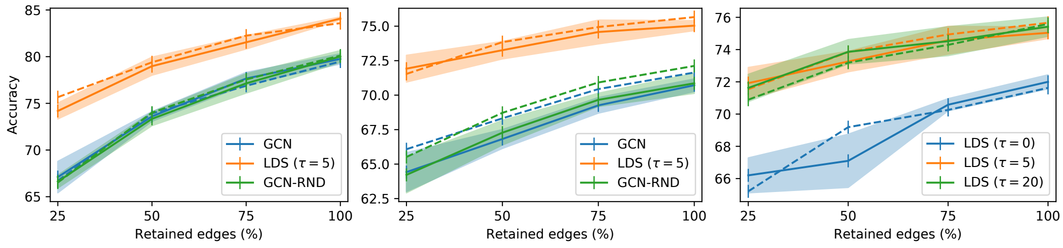

Figure 9.2: Mean accuracy ± standard deviation on validation (early stopping; dashed lines) and test (solid lines) sets for edge deletion scenarios on Cora (left) and Citeseer (center). (Right) Validation of the number of steps τ used to compute the hypergradient (Citeseer); $\tau = 0$ corresponds to alternating minimization. All results are obtained from five runs with different random seeds.

Table 9.1: Initial number of edges and expected number of sampled edges of learned graph by LDS.

% Edges	25%	50%	75%	100%
Cora Initial	1357	2714	4071	5429
Cora Learned	3635.6	4513.9	5476.9	6276.4
Citeseer Initial	1183	2366	3549	4732
Citeseer Learned	3457.4	4474.2	7842.5	6745.2

(used for early stopping) and test sets are plotted. LDS achieves competitive results in all scenarios and accuracy gains of up to 7 percentage points. Notably, LDS improves the generalization accuracy of GCN models also when the given graph is that of the respective dataset (100% of edges retained), by learning additional helpful edges. At the time of publication [Franceschi et al., 2019], the accuracy of 84.1% and 75.0% for Cora and Citeseer, respectively, exceed all previous state-of-the-art results. Conversely, adding random edges does not help decrease the generalization error. GCN and GCN-RND perform similarly which indicates that adding random edges to the graph is not helpful.

Figure 9.2 (right) depicts the impact of the number of iterations τ to compute the hypergradients. Taking multiple steps strongly outperforms alternating optimization[5] (i.e. $\tau = 0$) in all settings. Increasing τ further to the value of 20, however, does not yield significant benefits, while increasing the computational cost (cf. Section 6.2.1).

[5]For $\tau = 0$, one step of optimization of L w.r.t. w, fixing θ is interleaved with one step of minimization of E w.r.t. θ, fixing w. Even if computationally lighter, this approach disregards the nested structure of (9.4)-(9.5), not computing the first term of Equation (5.19).

Table 9.2: Test accuracy (± standard deviation) in percentage on various classification datasets. The best results and the statistical competitive ones (by paired t-test with $\alpha = 0.05$) are in bold. All experiments have been repeated with 5 different random seeds. We compare *k*NN-LDS to several supervised baselines and semi-supervised learning methods. No graph is provided as input. *k*NN-LDS achieves high accuracy results on most of the datasets and yields the highest gains on datasets with underlying graphs (Citeseer, Cora).

	Wine	Cancer	Digits	Citeseer	Cora	20news	FMA
LogReg	92.1 (1.3)	**93.3 (0.5)**	85.5 (1.5)	62.2 (0.0)	60.8 (0.0)	42.7 (1.7)	37.3 (0.7)
Linear SVM	93.9 (1.6)	**90.6 (4.5)**	87.1 (1.8)	58.3 (0.0)	58.9 (0.0)	40.3 (1.4)	35.7 (1.5)
RBF SVM	**94.1 (2.9)**	**91.7 (3.1)**	86.9 (3.2)	60.2 (0.0)	59.7 (0.0)	41.0 (1.1)	**38.3 (1.0)**
RF	93.7 (1.6)	**92.1 (1.7)**	83.1 (2.6)	60.7 (0.7)	58.7 (0.4)	40.0 (1.1)	**37.9 (0.6)**
FFNN	89.7 (1.9)	**92.9 (1.2)**	36.3 (10.3)	56.7 (1.7)	56.1 (1.6)	38.6 (1.4)	33.2 (1.3)
LP	89.8 (3.7)	76.6 (0.5)	**91.9 (3.1)**	23.2 (6.7)	37.8 (0.2)	35.3 (0.9)	14.1 (2.1)
ManiReg	90.5 (0.1)	81.8 (0.1)	83.9 (0.1)	67.7 (1.6)	62.3 (0.9)	**46.6 (1.5)**	34.2 (1.1)
SemiEmb	91.9 (0.1)	89.7 (0.1)	**90.9 (0.1)**	68.1 (0.1)	63.1 (0.1)	**46.9 (0.1)**	34.1 (1.9)
Sparse-GCN	63.5 (6.6)	72.5 (2.9)	13.4 (1.5)	33.1 (0.9)	30.6 (2.1)	24.7 (1.2)	23.4 (1.4)
Dense-GCN	90.6 (2.8)	90.5 (2.7)	35.6 (21.8)	58.4 (1.1)	59.1 (0.6)	40.1 (1.5)	34.5 (0.9)
RBF-GCN	90.6 (2.3)	**92.6 (2.2)**	70.8 (5.5)	58.1 (1.2)	57.1 (1.9)	39.3 (1.4)	33.7 (1.4)
*k*NN-GCN	93.2 (3.1)	**93.8 (1.4)**	**91.3 (0.5)**	68.3 (1.3)	66.5 (0.4)	41.3 (0.6)	**37.8 (0.9)**
*k*NN-LDS (dense)	**97.5 (1.2)**	**94.9 (0.5)**	**92.1 (0.7)**	**70.9 (1.3)**	**70.9 (1.1)**	**45.6 (2.2)**	**38.6 (0.6)**
*k*NN-LDS	**97.3 (0.4)**	**94.4 (1.9)**	**92.5 (0.7)**	**71.5 (1.1)**	**71.5 (0.8)**	**46.4 (1.6)**	**39.7 (1.4)**

In Table 9.1 we computed the average number of edges in a sampled graph for Cora and Citeseer, to analyze the properties of the graphs sampled from the learned graph generator. The expected number of edges for LDS is higher than the original number which is to be expected since LDS has better accuracy results than the standard GCN in Figure 9.2. Nevertheless, the learned graphs are still very sparse (e.g. for Cora, on average, less than 0.2% edges are present). This facilitates efficient learning of the GCN in the inner learning loop of LDS.

Table 9.2 lists the results for semi-supervised classification problems. The supervised learning baselines work well on some datasets such as Wine and Cancer but fail to provide competitive results on others such as Digits, Citeseer, Cora, and 20News. The semi-supervised learning baselines LP, ManiReg and SemiEmb can only improve the supervised learning baselines on 1, 3 and 4 datasets, respectively. The results for the GCN with different input graphs show that *k*NN-GCN works well and provides competitive results compared to the supervised baselines on all datasets. *k*NN-LDS significantly outperforms *k*NN-GCN on 4 out of the 7 datasets. In addition, *k*NN-LDS is among the most competitive methods on all datasets and yields the highest gains on datasets that have an underlying graph (Cora and Citeseer; see also Figure 9.5 for

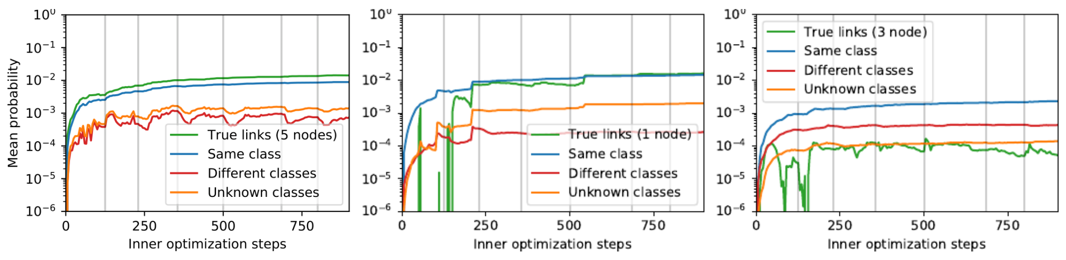

Figure 9.3: Mean edge probabilities to nodes aggregated w.r.t. four groups during LDS optimization, in log_{10} scale for three example nodes. For each example node, all other nodes are grouped by the following criteria: (a) adjacent in the ground truth graph; (b) same class membership; (c) different class membership; and (d) unknown class membership. Probabilities are computed with LDS ($\tau = 5$) on Cora with 25% retained edges. From left to right, the example nodes belong to the training, validation, and test set, respectively. The vertical gray lines indicate when the inner optimization dynamics restarts, that is, when the weights of the GCN are reinitialized.

a visual representation of the learned embeddings with various methods). Moreover, *k*NN-LDS performs slightly better than its dense counterpart where we learn a dense adjacency matrix. The added benefit of the sparse graph representation lies in the potential to scale to larger datasets.

In Figure 9.3, we show the evolution of mean edge probabilities during optimization on three types of nodes (train, validation, test) on the Cora dataset. LDS is able to learn a graph generative model that is, on average, attributing 10 to 100 times more probability to edges between samples sharing the same class label. LDS often attributes a higher probability to edges that are present in the true held-out adjacency matrix (green lines in the plots). In Figure 9.4 (left) we report the normalized histograms of the optimized edges probabilities for the same nodes of Figure 9.3, sorted into six bins in $\log_{10}$-scale. Edges are divided in two groups: edges between nodes of the same class (blue) and between nodes of unknown or different classes (orange). LDS is able to learn highly non-uniform edge probabilities that reflect the class membership of the nodes.

Figure 9.4 (right) shows similar qualitative results as Figure 9.4 (left), this time for three Citeseer test nodes, missclassified by *k*NN-GCN and correctly classified by *k*NN-LDS. Again, the learned edge probabilities linking to nodes of the same classes is significantly different to those from different classes; but in this case the densities

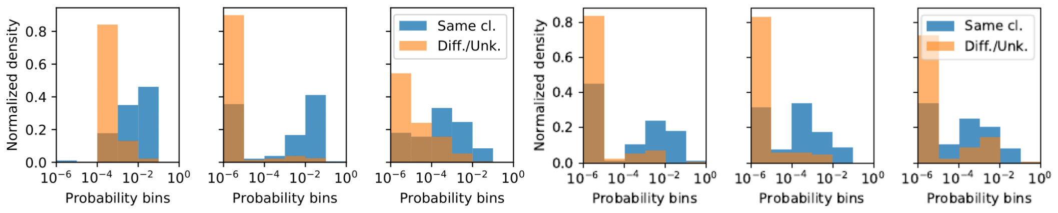

Figure 9.4: (Left) Normalized histograms of edges' probabilities for the same nodes of Figure 9.3. (Right) Histograms for three Citeseer test nodes, missclassified by *k*NN-GCN and rightly classified by *k*NN-LDS.

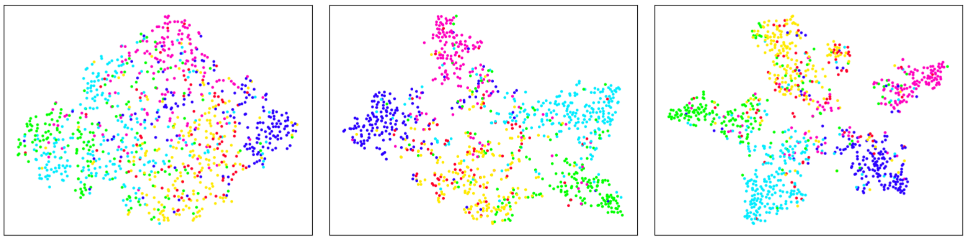

Figure 9.5: T-SNE visualization of the output activations (before the classification layer) on the Citeseer dataset. Left: Dense-GCN, Center: *k*NN-GCN, Right *k*NN-LDS

are more skewed toward the first bin. On the datasets we considered, what seems to matter is to capture a useful distribution (i.e. higher probability for links between same class) rather than pick exact links; of course for other datasets this may vary. We further visualize the embeddings learned by GCN and LDS using T-SNE [Maaten and Hinton, 2008] in Figure 9.5.

9.5 Related work

Semi-supervised Learning. Early works on graph-based semi-supervised learning use graph Laplacian regularization and include label propagation (LP) [Zhu et al., 2003], manifold regularization (ManiReg) [Belkin et al., 2006], and semi-supervised embedding (SemiEmb) [Weston et al., 2012]. These methods assume a given graph whose edges represent some similarity between nodes. Later, [Yang et al., 2016] proposed a method that uses graphs not for regularization but rather for embedding learning by jointly classification and graph context prediction. Kipf and Welling [2017] presented the first GCN for semi-supervised learning. There are now numerous GCN variants all of which assume a given graph structure. Contrary to all existing graph-based semi-supervised learning approaches, LDS is able to work even when the graph is incomplete or missing.

Graph Synthesis and Generation. LDS learns a probabilistic generative model for graphs. The earliest probabilistic generative model for graphs was the Erdős-Rényi random graph model [Erdos and Rényi, 1960], where edge probabilities are modelled as identically distributed and mutually independent Bernoullis. Several network models have been proposed to model well particular graph properties such as degree distribution [Leskovec et al., 2005] or network diameter [Watts and Strogatz, 1998]. Leskovec et al. [2010] proposed a generative model based on the Kronecker product that takes a real graph as input and generates graphs that have similar properties. Recently, deep learning based approaches have been proposed for graph generation [You et al., 2018, Li et al., 2018c, Grover et al., 2019, De Cao and Kipf, 2018]. The goal of these methods, however, is to learn a sophisticated generative model that reflects the properties of the training graphs. LDS, on the other hand, learns graph generative models as a means to perform well on classification problems and its input is not a collection of graphs. More recent work proposed an unsupervised model that learns to infer interactions between entities while simultaneously learning the dynamics of physical systems such as spring systems [Kipf et al., 2018]. Contrary to LDS, the method is specific to dynamical interacting systems, is unsupervised, and uses a variational encoder-decoder. Finally, we note that Johnson [2017] proposed a fully differentiable neural model able to process and produce graph structures at both input, representation and output levels; training the model requires, however, supervision in terms of ground truth graphs.

Link Prediction. Link prediction is a decades-old problem [Liben-Nowell and Kleinberg, 2007]. Several survey papers cover the large body of work ranging from link prediction in social networks to knowledge base completion [Lü and Zhou, 2011, Nickel et al., 2016]. While a majority of the methods are based on some similarity measure between node pairs, there has been a number of neural network based methods [Zhang and Chen, 2017, 2018]. The problem we study in this chapter is related to link prediction as we also want to learn or extend a graph. However, existing link prediction methods do not simultaneously learn a GNN node classifier. Statistical relational learning (SRL) [Getoor and Taskar, 2007] models often perform both link prediction and node classification through the existence of binary and unary predicates. However, SRL models are inherently intractable and the structure and parameter learning steps are independent.

Gradient Estimation for Discrete Random Variables. Due to the intractable nature of the two bilevel objectives, LDS needs to estimate the hypergradients through a stochastic computational graph [Schulman et al., 2015]. Using the score function estimator, also known as REINFORCE [Williams, 1992], would treat the outer objective as a black-box function and would not exploit E being differentiable w.r.t. the sampled adjacency matrices and inner optimization dynamics. Conversely, the pathwise estimator is not readily applicable, since the random variables are discrete. LDS borrows from a solution proposed before [Bengio et al., 2013], at the cost of having biased estimates. Recently, Jang et al. [2017] and Maddison et al. [2017] presented an approach based on continuous relaxations to reduce variance, which Tucker et al. [2017] combined with REINFORCE to obtain an unbiased estimator. Grathwohl et al. [2018] further introduced surrogate models to construct control variates for black-box functions. Unfortunately, these latter methods require to compute the function in the interior of the hypercube, possibly in multiple points [Tucker et al., 2017]. This would introduce additional computational overhead[6].

9.6 Discussion

In this chapter we presented LDS, a method that follows the bilevel framework to simultaneously learn the graph structure and the parameters of a GNN. While we have used a specific GCN variant [Kipf and Welling, 2017] in the experiments, the method is more generally applicable to other GNNs. The strengths of LDS are its high accuracy gains on typical semi-supervised classification datasets at a reasonable computational cost. Moreover, due to the graph generative model LDS learns, the edge parameters have a probabilistic interpretation. Bedsides its specific application to relational learning, we have shown that it is possible to compute efficiently update

[6]Recall that E can be computed only after (approximately) solving the inner optimization problem.

directions for parameters of discrete distributions that appear at the outer level by employing the STE. Suitable variants of LDS algorithm may also be applied to other problems such as neural architecture search or to tune other discrete hyperparameters.

The method has its limitations. While relatively efficient, it cannot currently scale to large datasets: this would require an implementation that works with mini-batches of nodes. We evaluated LDS only in the transductive setting, when all data points (nodes) are available during training. Adding additional nodes after training (the inductive setting) would currently require retraining the entire model from scratch. When sampling graphs, we do not currently enforce the graphs to be connected. This is something we anticipate to improve the results, but this would require a more sophisticated sampling strategy. All of these shortcomings motivate future work.

Chapter 10

Conclusions

In this thesis, we proposed and studied a unifying framework for hyperparameter optimization and meta-learning, motivated by the central observation that both HPO and MTL revolve around the search for good learning algorithms, albeit the scale and the experimental settings typically differ.

The framework hinges on the formulation of a bilevel program that abstracts away from these differences, allowing one to express a large portion of learning and meta-learning algorithms (in particular, those based on the ERM paradigm) concisely and effectively. We declined the framework for HPO, where the outer variables are interpreted as hyperparameters and are instrumental in finding hypotheses that generalize well, and for MTL, where the outer variables represent, instead, the object of learning, playing the role of parameters of meta-learners shared among different tasks. In MTL, the inner and outer problems naturally correspond to the base and the meta levels of learning, as identified by the seminal work of Thrun and Pratt. Contextually to our review of meta-learning of Chapter 4, we discussed the extent to which the proposed framework covers and reflects existing MTL methodologies, finding in the so-called parametric (optimization-based) algorithmic strategies the closest match. Various authors have suggested bilevel programming for hyperparameter optimization and other areas of machine learning. Yet, showing that the formalism adapts well also to the meta-learning context is a novel contribution of our work.

We saw that the inner problems linked to most learning algorithms do not typically admit closed-form solutions, except for a few cases such as ridge regression. Thus, in order to provide general and practical algorithms, one must resort to approximations. A compelling strategy is to replace the minimization of the inner objective with a set of constraints. These constraints can be either given by the first order optimality conditions (implicit view) or by the equations of an optimization dynamics (iterative view). We argued that the iterative view reflects more closely how learning algorithms are implemented in practice, crucially allowing one to consider as outer variables hyperparameters (or meta-parameters) that pertain the dynamics itself (e.g. learning rates or initialization points). In turns, this makes it possible to include into the framework a larger class of important HPO problems and popular MTL strategies.

When the inner and outer variables are real-valued and the objectives are smooth, the iterative and implicit approximation approaches give rise to two different families of procedures to compute approximate gradients of the outer objective. Concerning iterative differentiation, we derived the reverse mode, generalizing previous work by Domke [2012] and Maclaurin et al. [2015a], and proposed the forward mode, linked to classic algorithms for training recurrent neural networks presented by Williams and Zipser [1989] in the late eighties. For implicit differentiation, we considered a general strategy which uses the conjugate gradient method, explored by Pedregosa [2016] in HPO and Rajeswaran et al. [2019] in MTL, and one that involves differentiating a fixed point equation, linked to recurrent backpropagation and recently adopted by Liao et al. [2018] and MacKay et al. [2019]. We provided convergence results regarding the set of minimizers of the iterative approximate programs and proved non-asymptotic linear rates for the hypergradient approximation error of the iterative and fixed-point approaches when the underlying dynamics is contractive. Overall, our presentation brings together a series of methods and procedures previously scattered throughout the literature, generalizing from case-specific formulations and filling some existing gaps, especially under a theoretical standpoint.

Our experience with numerical simulations has been motivated by two fundamental goals. First, we wanted to complement our theoretical analysis by showing some limit cases and investigating the effect of various assumptions. We found that, for the iterative approach, small horizons may help improve generalization – possibly acting as a form of implicit regularization – while saving computation. However, for too small values, the computed hypergradient may incur in too high approximation errors, thus preventing successful optimization. We also found that the hypothesis of contractiveness of the inner dynamics that we used for deriving the hypergradient error bounds has a more visible impact on the implicit differentiation methods, which exhibited an unstable behaviour when this falls. Second, we wanted to provide evidence of the practical benefits of the proposed framework on real-world-inspired problems. With this aim, we showed the advantages of high dimensional gradient-based hyperparameter optimization, possible with the application of `Reverse-HG`, and the competitive performances of a real-time version of the forward mode for tuning few critical hyperparameters. We further adapted a simple and classic strategy for multitask learning (sharing a common representation) to the MTL setting and showed that it can extrapolate to novel tasks and perform comparably well. Contextually, we verified that the two main "innovations" associated to the MTL setting and our proposed framework[1] are essential in achieving good results with our meta-learning algorithm.

Some limitations of the main algorithms presented in the central part of the thesis led us to develop extensions in two main directions. First, we focused on the online version of the forward mode hypergradient computation scheme. An early version of the procedure, dubbed RTHO in Chapter 5, achieved promising results on some application settings, automatically generating schedules for few hyperparameters that could lead to trained models with improved generalization. Nonetheless, some aspects of the method remained unclear. We also encountered some stability issues (e.g. as reported in Section 8.4) which could prevent a broader utilization of RTHO. These facts motivated the development of MARTHE, We took as a specific case study the optimization of the learning rate schedule, widely recognized as one of the most critical hyperparameters for training deep neural networks – often researchers and practitioners spend a good amount of time to hand-tune it. By comparing the online update directions compute by RTHO with those calculated with exact (but

[1] Namely, optimize at the meta-level to achieve good generalization on the test splits and use hypergradients rather than performing alternating or joint optimization.

computationally unfeasible) iterative differentiation, we realized that a conceivable source of instability was the accumulation of undiscounted (and possibly outdated) information on the tangent system. Hence, we reinterpreted the computation of the online forward-mode system through the lens of a two-stage approximation process that involves the differentiation of shifted shorter-horizon auxiliary objectives. Under this perspective, it appeared natural to introduce a discount factor to reduce the impact of past information, effectively modelling MARTHE updates as moving averages rather than summations, mimicking, in a way, the update rule of gradient descent with momentum. Even though this modification adds a new configuration parameter (besides the hyper-learning rate), we empirically showed that it substantially helps improve the procedure's stability, enabling MARTHE to outperforms RTHO and other approaches on a series of carefully designed time-controlled experiments.

Next, we turned our attention to another challenging goal: extending the applicability of the proposed framework to the optimization of a class of discrete hyperparameters. We considered, in this case, a relational learning setting, picking graph convolutional neural networks as underlying models. Introducing a simple discrete probability distribution over adjacency matrices, we formulated a bilevel problem where the outer variables are the (real-valued) parameters of the distribution. We defined the inner and outer objectives as the expected training and validation errors of a GCN that takes as input (alongside the node features) the random variable representing the adjacency matrix. Following once again the iterative approximation approach, we used stochastic gradient descent as optimization dynamics (the stochasticity stemming from the graph distribution) and derived an hypergradient estimator leveraging the straight-through estimator. Accordingly, we developed a practical algorithm based on a truncated version of `Reverse-HG` which we named LDS to jointly learn discrete graph structures and the weights of the GCN. We showed that LDS manages to find meaningful dependencies among data point leading to improved performances in the presence of incomplete graphs. We empirically demonstrated that once a dependency structure is learned with LDS, GCNs are good candidate models for semi-supervised transductive learning, consistently matching or exceeding baselines and competitor techniques on several datasets. Despite some current limitations, this may be considered a fascinating result on its own which might sparkle interest in the area of learning or generating complex dependency structures, moving beyond the assumption of independence between data points assumed in many learning settings.

10.1 Future Work

The perspective we have taken in this work and the proposed framework contribute to blur the border between hyperparameter optimization and meta-learning. However, with the partial exception of Chapter 9, we have mostly investigated the descriptive power of the framework, proposing instantiations that "do not fall too far" from existing and classical learning and meta-learning problems and algorithms. Beside extending our descriptive analysis to the other major paradigms of machine learning, we believe that a first natural direction of future research is that of exploring the constructive potential of the framework. In this regard, designing learning systems that integrate more closely HPO (alongside algorithm selection) and MTL might be a particularly fruitful effort. Some core ideas of so-called pool-based algorithmic MTL methods (Section 4.4.2.1) could provide useful starting points.

As we mentioned in our review of Chapter 4 (see ending paragraph of Sections 4.2), interesting and challenging meta-learning problems may arise from (and be described by) joint distributions where the marginal distribution of the data observations may be quite far from (few) i.i.d. examples of a given concept or phenomenon. For instance, one may consider a situation where observed datasets are biased in some sense (e.g. representing unfair credit score assignment), or where observations are given in the form of natural language descriptions of a task rather than a set of supervised examples. We believe that another interesting line of research is that of investigating the applicability and effectiveness of our proposed framework in these meta-learning scenarios that substantially differ from few-shot learning.

Under a more technical standpoint, it would be valuable to extend our convergence results to settings in which inner problems do not satisfy the assumptions of our convergence analysis. These include bilevel problems in which the lower level dynamics is only locally contractive, non-smooth, possibly non-expansive (e.g. when the inner objective is only convex) or can only be solved via a stochastic procedure. Additionally another important step to close the gap between practical procedures and theoretical understanding is to study the statistical properties of bilevel strategies where the outer objectives are based on the generalization ability of the inner models. As our framework links meta-learning and hyperparameter optimization, an analysis in this direction may be useful to both fields. Ideas from [Maurer et al., 2016, Denevi et al., 2018b, Konobeev et al., 2020, Chen et al., 2020a] may be helpful in this direction.

Lastly, the algorithm that we have developed in Chapter 9 is a first step toward the development of gradient-based strategies to tune discrete hyperparameters or meta-parameters. For LDS, we have considered only a rather simple discrete probability distribution, where the straight through estimator has been proved effective in practice. However, settings in which the outer variables appear as parameters of more complex discrete distributions or define the cost or constraints of combinatorial optimization problems would very likely require different estimation strategies. Further developments in this directions could enable the effective design of a novel class of meta-learning methods. For example, one could imagine a MTL algorithm in which the structure of the base-level algorithm could be determined by the solution of an integer programming problem where the constraints could possibly encode previous knowledge or requirements specified by the user.

Appendix A

Fundamentals of Algorithmic Differentiation

In this chapter we review the fundamentals of algorithmic differentiation. After a introduction that defines the concepts and boundaries of this area of applied mathematics, in Section A.2 we discuss general structures for evaluating functions, representation schemes and assumptions employed in the context of algorithmic differentiation. We then continues with the introduction and derivation of the two main modes for computing derivatives, i.e. forward and reverse-mode differentiation (Section A.3). Finally, in Section A.4 we analyze the computational complexities these two modes. This chapter is intended as a convenient summary of the first five chapters of the book by Griewank and Walther [2008].

A.1 Introduction

Algorithmic differentiation, also known as automatic differentiation deals with the problem of automatically augmenting a given program with operations that allow the evaluations of its derivatives or other related quantities, such as Hessians or Jacobian vector products. It is based on the assumption that the vast majority of scientific programs executes a series of elementary operations that can be tracked and differentiated through efficiently. Perhaps one of the best known results of algorithmic differentiation is the so-called *cheap gradient principle*, which states that gradients of scalar functions can be obtained through reverse-mode algorithmic differentiation at a run-time cost which is only a small multiple of that of computing the original function (see Section A.4). Clearly, this fact has rather important practical implications e.g. when optimizing high dimensional objective functions, as having access to cheap and accurate gradients through automatically generated code may greatly simplify the task.

In the last decade, the systematic applications of algorithmic differentiation tools [Theano Development Team, 2016, Paszke et al., 2017, Abadi et al., 2015] has been recognized one of the key drivers behind the recent successes of machine learning [Baydin et al., 2018b], and deep learning in particular. The fact that compositionality of simple nonlinear mappings is among the core design features of artificial neural networks has certainly favoured the early adoption of (a subset of) such tools in the field – backpropagation [Rumelhart et al., 1986] and backpropagation through time [Werbos, 1990] being two major examples, implementing reverse-mode gradient computation for feed-forward and recurrent neural networks, respectively.

Before proceeding, we briefly remark the main differences between algorithmic differentiation and two other approaches for computing derivatives, namely numerical and symbolic differentiation. Algorithmic differentiation produces programs that return numerical values once evaluated at a point; yet, it greatly differs from numerical derivation, where the derivative of a function f in the direction e is often computed with difference quotients

$$\mathrm{D}^{+}_{h,e}f(x) = \frac{f(x+he)-f(x)}{h} \quad \text{or} \quad \mathrm{D}^{\pm}_{h,e}f(x) = \frac{f(x+he)-f(x-he)}{2h}, \quad h>0.$$

Numerical differentiation is inflexible and prone to truncation errors, worsened in high dimensionality settings; algorithmic differentiation, on the other hand, makes no additional approximation[1] and allows for seamless and efficient computation of gradients and higher order derivatives.

Algorithmic differentiation differs from symbolic differentiation since it does not manipulates nor return algebraic expressions and rather revolves around computing intermediate values. Consider as an example the multivariate function $f(x) = \prod_i x_i$.

[1]Of course algorithm differentiation is not immune to errors deriving from finite precision arithmetic, once implemented in a computer software.

Symbolically differentiating f would yield expressions such as $\frac{\partial f}{\partial x_i} = \prod_{j \neq i} x_j$. Computing ∇f would then require multiplying several times the independent variables, as there is in principle no relation between different components of the gradient. This would clearly neglect the particular structure of f, where partial multiplications could be stored and reused; algorithmic differentiation provides methods to automatically do so, leading to substantial computational savings.

A.2 Evaluating Functions

Let

$$f : \mathcal{X} \subseteq \mathbb{R}^n \to \mathcal{Y} \subseteq \mathbb{R}^m \tag{A.1}$$

be the function of interest we wish to differentiate. Algorithmic differentiation requires additional structure with respect to other differentiation techniques, as it requires knowing *how* f is evaluated, beside being able to compute its numerical value at a point. In this section we will specify the implications of this requirement and how this naturally leads to defining different and complementary ways to represent the evaluation of f, each capturing peculiar aspects of the computational process.

Let x_i for $i = 1, \dots, n$ and y_j, for $j = 1, \dots, m$ be the set of independent and dependent variables, respectively. The central assumption of algorithmic differentiation is that most of the functions of interest are computed (e.g. by a CPU or a GPU) using simple operations and intermediate variables. More formally, we denote by v_k the set of *intermediate* (or auxiliary) variables and let

$$v_k = \psi_k(x_1, \dots, x_n, v_1, \dots, v_{k-1}) \qquad \text{for } k = 1, \dots, l$$

where l depends on the specific function and $\psi_k : \mathcal{X}_k \to \mathbb{R}$ are scalar functions called *elementals*[2] belonging to a predefined *library* Ψ. The library may include from very simple arithmetic operations[3] such as addition and multiplication to look-up tables

[2] For the sake of simplicity, in this section we consider only scalar variables and elementals; the generalization to vector valued variables and elementals is straightforward.

[3] A somewhat minimal requirement is that Ψ contains at least the so-called *polynomial core*, i.e. addition, multiplication, unary sign switch and constant assignment. These elementals allow computing polynomial functions, which in turn may be used to approximate other operations such as trigonometric

and stochastic random variables. We assume that the the output of f is simply given by the last m intermediate variables, so that

$$y_j = v_{l-j} \qquad \text{for } j = 1, \ldots, m$$

For example, for the product function $f(x) = \prod_i x_i$, the intermediate variables may be defined as the partial products $v_k = \prod_{i=1}^{k} x_k = v_{k-1} * x_k$, and the scalar output $y_1 = f(x) = v_n$; the number of auxiliary variables l would be n in this case.

Clearly, there may be multiple choices for writing down a function in this way, which may in turn depend on the granularity of the available elementals. Some choices could be preferable to others when considering possible numerical errors or overflows. Algorithmic differentiation is strongly tied to the implementation of the evaluation procedure: informally, what is good for computing the function is also good for calculating derivatives. In any case, considering f as the results of a series of intermediate operations which make use of a number of auxiliary variables (rather than thinking of f as a monolithic expression), let us better reason about strategies to save and reuse computation and paves the way to think about differentiation under an operational (rather than symbolic) standpoint.

A.2.1 Evaluation procedure and evaluation trace.

To unify the notation we introduce the set of auxiliary variables with non-positive indices v_{i-n} for $i = 1, \ldots, n$ to account for the input assignment. The general evaluation procedure for algorithmically differentiable functions (with library Ψ) is listed in Table A.1. This is not a mere restatement of Eq. (A.1), as the existence of an evaluation procedure for a function f implies that f is the result of a *finite* list of explicitly known operations laid down in a precise known order. In short, we say that $f \in \mathrm{Span}(\Psi)$ if there exists a (not necessarily unique) evaluation procedure for f with elementals belonging to Ψ.

The evaluate of the function at a point $\hat{x} \in \mathcal{X}$ (*instantiation*) proceeds by simply assigning the numerical values of $\hat{x}$ to the first n variables and then continues by

and exponential functions. It is far more common, however, to include in Ψ also more complex operations.

Table A.1: General evaluation procedure for a function $f \in \mathrm{Span}(\Psi)$.

$v_{i-n} = x_i$	$i = 1, \ldots, n$	Assign inputs
$v_i = \psi_i(v_{1-n}, \ldots, v_{i-1})$	$i = 1, \ldots, l$	Compute intermediate variables
$y_i = v_{l-i}$	$i = 1, \ldots, m$	Copy output

computing the intermediate and output values $\hat{v}_i = \psi_i(\hat{v}_{1-n}, \ldots, \hat{v}_{i-1})$. This produces the so called *evaluation trace* $\{\hat{v}_i\}_{i=1-n}^{l}$ at $\hat{x}$, which essentially is the record of a particular execution of f.

A.2.2 Computational graph.

Most of elementals of practical interest are unary or binary functions, this means that very often the computation of ψ_i (second row of Table A.1) will only *directly* depend on few variables. For instance, in the product function example we have $v_i = v_{i-n} * v_{i-1} = \psi_i(v_{i-n}, v_{i-1})$: given the value of the two multiplicands, v_i is independent from the rest of the intermediate variables. Furthermore, it may be the case that operations may be carried out in parallel[4], as the computation of one does not affect the other. The *computational graph* offers a way to represent the evaluation of a function that concisely captures these two aspects. We can represent f by a directed acyclic graph $\mathcal{G}_f = (\mathcal{V}_f, \mathcal{E}_f)$ where the nodes $\mathcal{V}_f = \{v_i\}_{i=1-n}^{l}$ play the role of input, output and intermediate variables. The edges encode the relationship of direct dependence, denoted with the symbol $\prec$:

$$(v_j, v_i) \in \mathcal{E}_f \quad \Leftrightarrow \quad v_i \prec v_j \quad \Leftrightarrow \quad v_i = \psi_i(v_j, \ldots).$$

The direct dependence relation $\prec$ constitute a partial ordering of the set of indices. Its transitive closure, denoted $\prec^*$, represents the “usual” mathematical computational dependence. For instance, it is most often the case that $y_j \prec^* x_i$ but $y_j \nprec x_i$, i.e. the outputs depend, but not directly, from the input (as typically the input is transformed by a series of operations which define f). The roots of the graph are the independent variables while the leafs are the dependents. We refer to Figure A.1 for a concrete

[4] Sadly, this does not apply to the product example, as computing v_i requires knowing v_{i-1}.

example of a computational graph for a toy function.

The second row of Table A.1 can then be replaced with the more concise notation

$$v_i = \psi_i(v_j)_{j<i} = \psi_i(u_i) \qquad \text{for } i = 1,\dots,l$$

where, from now on, we will use the notation $u_i = (v_j)_{j<i} \in \mathbb{R}^{n_i}$.

A.2.3 System of equations

The evaluation procedure can also be thought of as system of $n+l$ equations

$$0 = E(x,v) = (v_i - \psi_i(u_i))_{i=1-n}^{l} \tag{A.2}$$

where the first n components are simply the input assignment functions $\psi_i(u_i) = x_i$, for $i = 1-n,\dots,0$. This representation mode is particularly useful when deriving the reverse mode algorithmic differentiation using the Lagrangian formulation. The square Jacobian matrix of E w.r.t. the intermediate variables v has, by construction, the block structure

$$D_vE(x,v) = \begin{pmatrix} I & 0 & 0 \\ C_1 & I-C_2 & 0 \\ C_3 & C_4 & I \end{pmatrix} = I - C,$$

where the strictly lower triangular matrix $C = (c_{ij})$ contains the partial derivatives of the elemental functions: $c_{ij} = \partial_{v_j}\psi_i$. Thus, $D_vE(x,v)$ is (globally) non-singular, and by the implicit function theorem the variables v are uniquely determined as functions of x. In particular this is true for the last m components of v, and we have that $y = (v_i)_{i=l-m}^{l} = f(x)$, as expected.

A.2.4 State Transformations

The evaluation procedure for f may be also represented with a dynamical system where the state is given by the vector of auxiliary variables $v = (v_i)_{i=1-n}^{l} \in \mathbb{R}^{n+l}$. We introduce the state transitions

$$\Phi_i : \mathbb{R}^{n+l} \to \mathbb{R}^{n+l} \qquad [\Phi_i(v)]_j = \begin{cases} \psi_i(u_i) & \text{if } j = i \\ v_j & \text{if } j \neq i \end{cases} \tag{A.3}$$

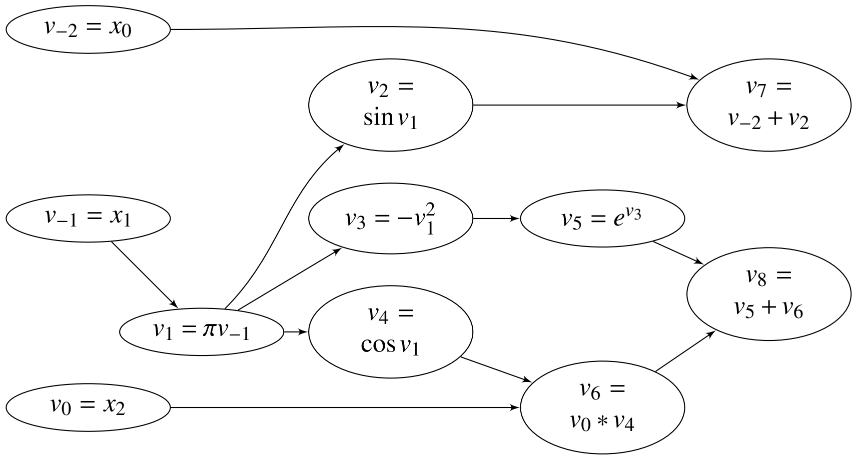

Figure A.1: Example of a computational graph for the function of Example A.2.1

In other words, the maps (A.3) perform the i-th elemental operation on the i-th component of the state vector, and leave everything else unvaried. Denoting by

$$P_n = \begin{pmatrix} I_n & 0 & \cdots & 0 \end{pmatrix} \in \mathbb{R}^{n \times n+l} \quad \text{and} \quad Q_m \in \begin{pmatrix} 0 & \cdots & 0 & I_m \end{pmatrix} \in \mathbb{R}^{m \times n+l}$$

the projections of the first n and last m components, then evaluating f is equivalent to compute

$$y = f(x) = Q_m \left(\Phi_l \circ \Phi_{l-1} \circ \cdots \circ \Phi_1 (P_n^\top x) \right) \tag{A.4}$$

Example A.2.1. We illustrate the computational graph and relative evaluation procedure for a a simple bivariate function given by

$$f : \mathbb{R}^3 \to \mathbb{R}^2; \qquad f(x_0, x_1, x_2) = \begin{pmatrix} x_0 + \sin(\pi x_1) \\ x_2 \cos(\pi x_1) + e^{-(\pi x_1)^2} \end{pmatrix}. \tag{A.5}$$

Assuming that our elemental library includes the polynomial core, trigonometric functions and exponentiation, we sketch in Figure A.1 a possible computational graph, organized from left to right[5]. Operations on nodes at the same horizontal coordinate may be carried out in parallel. The relevant evaluation procedure is listed in Table A.2.

[5] Some operations such as constant assignment (of π) and negative square at node v_3 have been merged to improve readability.

Table A.2: Evaluation procedure for the function in Equation (A.5).

$v_{-2} = x_0$	$v_5 = e^{v_3}$
$v_{-1} = x_1$	$v_6 = v_0 + v_4$
$v_0 = x_0$	$v_7 = v_{-2} + v_2$
$v_1 = \pi * v_{-1}$	$v_8 = v_5 + v_6$
$v_2 = \sin v_1$	$y_1 = v_7$
$v_3 = -v_1^2$	$y_2 = v_8$
$v_4 = \cos v_1$	

A.3 Forward and Reverse Mode Differentiation

Clearly, in order for $f \in \text{Span}(\Psi)$ to be differentiable, we require every elemental in Ψ to be $d \geq 1$ times continuously differentiable on the interior of its domain. Since f is writable as composition of functions in the library, we have that

$$f \in \text{Span}(\Psi) \in C^d$$

on the interior of its domain (which, however, may be in principle empty).

The forward and reverse modes offer efficient and truncation-errors free ways to compute

$$\dot{y} = \mathrm{D}f(x)\,\dot{x} \in \mathbb{R}^{m\times 1} \tag{A.6}$$

and

$$\bar{x}^\intercal = \bar{y}^\intercal\,\mathrm{D}f(x) \in \mathbb{R}^{1\times n}, \tag{A.7}$$

respectively, where[6] $\mathrm{D}f : \mathcal{X} \subseteq \mathbb{R}^n \to \mathbb{R}^{m\times n}$ is the differential operator and $\mathrm{D}f(x) \in \mathbb{R}^{m\times n}$ denotes the Jacobian of f at x. The column vector $\dot{x}$ is called *seed direction*, while the row vector $\bar{y}$ is named *weight functional*[7]. These vectors should be thought of as input of the two procedures that will compute Equations (A.6) and (A.7), respectively.

Notably, when f is a curve (i.e. $n = 1$), and $\dot{x} = 1$, $\dot{y}$ represents the tangent, or

[6] We follow Euler's notation for Jacobians, which we may also write as $D_x f$, $\partial_x f$ or $\frac{\partial f}{\partial x}$ to underline the variable with respect to which we differentiate. The dotted notation – reminiscent of the Newton's notation – is used for tangents as it reminds to velocities of curves, while the barred notation is, in a way, "proprietary" of the reverse-mode differentiation literature.

[7] Technically, the barred quantities $\bar{y}$ and $\bar{x}$ should belong to the dual space of $\mathbb{R}^m$ and $\mathbb{R}^n$, i.e. the space of linear functions between with range in $\mathbb{R}$. For simplicity, since we are dealing only with real spaces, here we identify them with their canonical representation as vectors.

velocity, of the curve, forward mode algorithmic differentiation allows one to compute $\mathrm{D}f$ in only a single pass. Conversely, when f is a scalar function (i.e. $m = 1$), with the choice of $\bar{y} = 1$, $\bar{x}$ corresponds to the gradient ∇f, which can be computed in only one pass using reverse mode differentiation. When full Jacobians are needed, forward or reverse procedures should be called multiple times with the corresponding seeds or weight functionals set to the relevant canonical base vectors. Neither of the two modes compute $\mathrm{D}f(x)$ explicitly, at any point during execution – and in this fact lies the ground for the computational saving that is achievable using these differentiation techniques.

A.3.1 Forward mode differentiation

The forward mode algorithmic differentiation, also known as *tangent propagation*, computes $\dot{y} = \mathrm{D}f(x)\dot{x}$ and may be essentially described as the systematic application of the chain rule to the evaluation procedure of f. Geometrically, it can be viewed as computing the first derivative of f along a smooth curve $\gamma : \mathbb{R} \to \mathcal{X} \subset \mathbb{R}^n$ such that $\gamma(0) = x$ and $\gamma'(0) = \dot{x}$. By the chain rule one has

$$\dot{y} = \frac{\partial}{\partial t} f(\gamma(t))_{|t=0} = \mathrm{D}f(\gamma(0))\gamma'(0) = \mathrm{D}f(x)\dot{x}.$$

The tangent of the curve γ' is propagated from the domain space to the codomain by the mapping

$$\dot{f} : \mathcal{X} \times \mathbb{R}^n \to \mathbb{R}^m, \qquad \dot{f}(x, \dot{x}) = \mathrm{D}f(x)\dot{x} = \dot{y}, \tag{A.8}$$

as represented in Figure A.2.

Table A.1 may be augmented in a straightforward way to implement the mapping (A.8) by introducing a set of dotted intermediate variables $(\dot{v}_i)_{i=1-n}^{l}$ and the dotted elementals

$$\dot{\psi}_i : \mathcal{X}_i \times \mathbb{R}^{n_i} \to \mathbb{R}; \qquad \dot{\psi}_i(u_i, \dot{u}_i) = \mathrm{D}\psi(u_i)\dot{u}_i = \sum_{j<i} \frac{\partial \psi}{\partial v_j}(u_i)\dot{v}_j \tag{A.9}$$

where $\dot{u}_i = (v_j)_{j<i}$. The resulting procedure is listed in Table A.3 which implements (A.8). One derivative statement follow each of the evaluation statements. If there

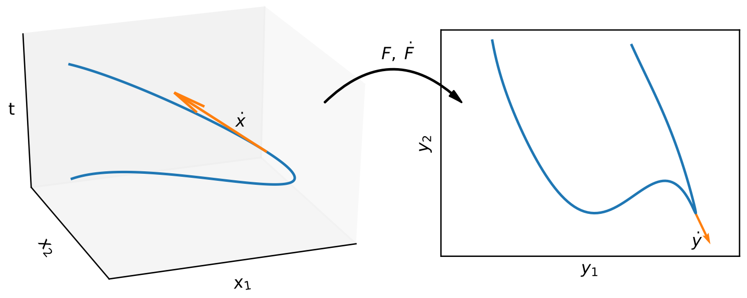

Figure A.2: A geometrical representation of the tangent propagation (forward mode algorithmic differentiation) for the toy function of Example A.2.1.

Table A.3: General forward mode (tangent) procedure.

$(v_{i-n}, \dot{v}_{i-n}) = (x_i, \dot{x}_i)$	$i = 1, \ldots, n$	Assign inputs and seed directions
$(v_i, \dot{v}_i) = \big(\psi_i(u_i), \dot{\psi}_i(u_i, \dot{u}_i)\big)$	$i = 1, \ldots, l$	Comp. intermediate var. and tangents
$(y_i, \dot{y}_i) = (v_{l-i}, \dot{v}_{l-1})$	$i = 1, \ldots, m$	Copy output and tangent

is no memory overwriting, the operations in the tangent space may be computed simultaneously with those to evaluate the function itself.

A.3.2 Reverse mode differentiation

If tangent propagation enables the efficient computation of velocities of curves, reverse mode algorithmic differentiation, or *gradient propagation*, provides a scheme for computing normals or cotangents of hyper-surfaces (which are gradients, if the function is real-valued) at a cost that is only a fixed multiple of that of evaluating f itself. Similar to forward mode (but note the different dimensionalities), reverse mode algorithmic differentiation can be thought of as an implementation of the mapping

$$\bar{f} : \mathcal{X} \times \mathbb{R}^m \to \mathbb{R}^n, \qquad \bar{f}(x, \bar{y}) = \bar{y}^\intercal \mathrm{D} f(x) = \bar{x}. \tag{A.10}$$

The function $\bar{f}$ propagates the normals to hyper-surfaces $\{y \in \mathcal{Y} : \bar{y}^\intercal y = c\}$ in the codomain space to normals of the hyper-surfaces $\{x \in \mathcal{X} : \bar{y}^\intercal F(x) = c\}$ in the domain space, for $c \in \mathbb{R}$; Figure A.3 offers a visualization for the toy example A.2.1; where, differently from forward propagation, the mapping (A.10) has been represented with an arrow from the output space to the input space since f must be evaluated before $\bar{f}$.

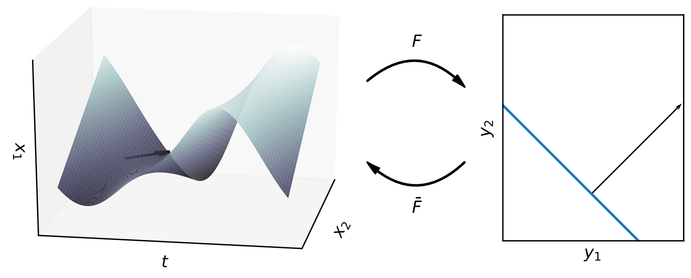

Figure A.3: Geometric visualization of the gradient propagation with reverse mode for the toy function A.2.1.

There are various routes to derive reverse mode gradient computation scheme. A first, more informal, way involves tracking backward the dependencies on the computational graph and computing sensitivities with respect to the inputs of each of the nodes. A second one, which we consider here, revolves around the representation of f as a dynamical system and the fundamental equation

$$\bar{x}^\intercal \dot{x} = \bar{y}^\intercal \dot{y}. \tag{A.11}$$

A third way, which makes use of the representation as system of equations and follows a Lagrangian derivation, will be explored in details in Section 5.4.1 in the context of gradient-based hyperparameter optimization and meta-learning.

Differentiating the state transformation (A.3)

$$y = f(x) = Q_m \left(\Phi_l \circ \Phi_{l-1} \circ \cdots \circ \Phi_1 (P_n^\intercal x) \right),$$

w.r.t. x and multiplying by $\dot{x}$ yields, by the chain rule, the expression

$$\dot{y} = \underbrace{Q_m A_l A_{l-1} \cdots A_1 P^\intercal}_{=\mathrm{D}f(x)} \dot{x},$$

where the matrices $A_i = \mathrm{D}\Phi_i(x) \in \mathbb{R}^{n+l \times n+l}$. Now, left multiplying by $\bar{y}^\intercal$ one has

$$\bar{y}^\intercal \dot{y} = \bar{y}^\intercal Q_m A_l A_{l-1} \cdots A_1 P_n^\intercal \dot{x}, \quad \Rightarrow \quad \bar{x} = P_n A_1^\intercal \cdots A_{l-1}^\intercal A_l^\intercal Q_m^\intercal \bar{y} \tag{A.12}$$

by transposing and by (A.11). Mirroring the dotted intermediate variable of the forward mode, we introduce now the barred intermediate variables $\bar{v} = (\bar{v}_i)_{i=1-n}^{l}$, called *adjoint* variables, which will incrementally keep track of the reverse mode computation of the cotangent. The last m components of $\bar{v}$ are initialized in the right-most side of (A.12) by the projection $Q_m^\top \bar{y}$ as $\bar{v}_{l-m+i} = \bar{y}_i$ for $i \in [m]$ and successively *backward* multiplied by the matrices $A_i^\top$ until the first n components of $\bar{v}$ are computed and then assigned to $\bar{x}$ through the projection P_n. The computational savings derive from the particular structure of the matrices A_i. Indeed, by inspecting the transposed differential of the state transformations, one may see that

$$A_i^\top = I_{n+l} + (\nabla\psi_i(u_i) - e_{n+i})e_{n+i}^\top$$

where, with a little abuse of notation, we embed the gradient of the elementals ψ_i into $\mathbb{R}^{n+l}$ by letting $[\nabla\psi_i(u_i)]_j = \partial_{v_j}\psi_i(u_i)$ for $j \prec i$ and 0 otherwise. This means that computing

$$A_i^\top \bar{v} = \bar{v} + (\nabla\psi_i(u_i) - e_{n+i})\bar{v}_i$$

only requires vector operations. In other words, at each iteration, $\bar{v}_j$ is incremented by $\partial_{v_j}\psi_i(u_i)\bar{v}_i$ if $j \prec i$, it is set to 0 if $i = j$ and left unchanged otherwise.

The resulting procedure is listed in Table (A.4), where, similarly to (A.9), we introduce the barred (adjoint) elementals

$$\bar{\psi}_i : \mathcal{X}_i \subseteq \mathbb{R}^{n_i} \times \mathbb{R} \to \mathbb{R}^{n_i} \qquad \psi_i(u_i, \bar{v}_i) = \nabla\psi_i(u_i)\bar{v}_i, \tag{A.13}$$

and $\bar{u}_i$ is the collection of the adjoints $(\bar{v}_j)_{j\prec i}$. Comparing with the forward mode in Table A.3, reverse mode does not allows for parallel evaluation of the function and the cotangent, as the adjoint variables must be computed backward once all the intermediate and output values are known. Furthermore, the values of the intermediate variables stored in the evaluation trace should be, in principle, maintained and not overwritten since they may be needed in the reverse pass (5th line in the Table). As we shall see in the next section, this has an impact on the space complexity of this scheme.

Table A.4: General reverse mode (gradient propagation) procedure; incremental formulation.

$v_{i-n} = x_i$	$i = 1,\ldots,n$	Assign inputs
$v_i = \psi_i(u_i)$	$i = 1,\ldots,l$	Compute intermediate variables
$y_i = v_{l-i}$	$i = 1,\ldots,m$	Copy output
$\bar{v}_{l-m+i} = \bar{y}_i$	$i = 1,\ldots,m$	Initialize adjoints
$\bar{u}_i = \bar{u}_i + \bar{\psi}_i(u_i, \bar{v}_i)$	$i = l-m,\ldots,1-n$	Increment adjoints
$\bar{x}_i = \bar{v}_{i-n}$	$i = 1,\ldots,n$	Copy cotangent (gradient)

Table (A.4) corresponds to the incremental version of the reverse mode. The instructions in the second last row may be swapped with the non-incremental formulation

$$\bar{v}_j = \sum_{i \succ j} \frac{\partial \psi_i(u_i)}{\partial v_j} \bar{v}_i$$

that arises from a Lagrangian derivation and does not requires cycling through the elementals' inputs. The incremental version is however more widespread in practice, being easier to implement. In fact, the non-incremental version would require having access to a graph of backward dependencies (encoding and storing the relation $i \succ j$) which is not always easy to obtain automatically.

A.4 Computational Complexity Analysis

We provide in the following the main results concerning *relative* time and space complexity of forward and reverse mode algorithmic differentiation, in terms of the cost of computing the original function f, under an idealized time and space model.

We consider a simple memory complexity measure which we denote `mem`. Regarding the temporal complexity model we consider a vector value complexity measure, denoted `work`, designed to take into account the different costs of basic operations. For the sake of concreteness we consider four components: memory access and storage (`moves`), addition and subtraction (`adds`), multiplication (`mults`) and (unary) non-linear operations (`nlops`), which may include reciprocal, exponentials, trigonometrics, and so on. Thus, the computational cost of a `task` is given by the vector

$$\texttt{work}(\texttt{task}(f)) = \begin{pmatrix} \texttt{moves} \\ \texttt{adds} \\ \texttt{mults} \\ \texttt{nlops} \end{pmatrix} \in \mathbb{N}^4 \tag{A.14}$$

that counts the number of basic operations needed to execute it. Specifically we are interested in the tasks of evaluating functions, tangents and gradients with the procedures outlined in the previous sections and deriving temporal complexity bounds for the last two. As these tasks require (sequentially) evaluating each elemental once we shall assume that

$$\texttt{work}(\texttt{task}(f)) \leq \sum_{i=1}^{l} \texttt{work}(\texttt{task}'(\psi_i)), \tag{A.15}$$

with the task of evaluating f at a point being strictly additive:

$$\texttt{work}(f) = \sum_{i=1}^{l} \texttt{work}(\psi_i). \tag{A.16}$$

In (A.15), $\texttt{task}'$ denotes the series of operations to be performed on each elemental which, in principle, might differ from the original `task`. This is e.g. the case of gradient propagation, where $\texttt{task}'$ involves also storing and retrieving additional values in the forward and backward passes.

To simplify the analysis and allow for constructive proofs, we consider in the following an essential elemental library, slightly larger than the polynomial core, which mirrors the temporal complexity model (A.14) and contains assignment to constants (c), addition/subtraction, multiplication, and a "generic" non-linearity (σ):

$$\Psi = \{c, \pm, *, \sigma\}. \tag{A.17}$$

For example multiplication, as binary operation, requires 2 fetches, a `mults` operation and then a store, thus resulting in $\texttt{work}(*) = (3, 0, 1, 0)$.

In light of (A.16), The temporal complexity of evaluating f, $\texttt{work}(f)$, can be easily expressed as linear combination of elemental complexities

$$\texttt{work}(f) = W_{\texttt{eval}}|f| = \begin{pmatrix} 1 & 3 & 3 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \end{pmatrix} \tag{A.18}$$

where $|f|$ may be called the *elemental frequency vector* of f. Likewise, (A.15) implies that there exist a matrix $W_{\texttt{task}}$ such that $\texttt{work}(\texttt{task}(f)) \leq W_{\texttt{task}}|f|$, where the $\leq$ is intended component-wise. The columns of $W_{\texttt{eval}}$ and $W_{\texttt{task}}$ contain the cost of evaluating ψ and executing $\texttt{task'}(\psi)$, respectively, for each $\psi \in \Psi$.

The *runtime* is defined defined as a linear combination of $\texttt{work}$:

$$\texttt{time}(\texttt{task}(f)) = \omega^{\intercal}\texttt{work}(\texttt{task}(f)), \tag{A.19}$$

where ω is a system-dependent vector which may measure the clock-cycles needed for each of the four operations. By taking $\texttt{adds}$ as unit, we may assume that

$$\omega = (\mu, 1, \pi, \nu)^{\intercal} \quad \text{such that} \quad \mu \geq \pi \geq 1 \text{ and } \nu \geq 2\pi. \tag{A.20}$$

(Sub)-additiveness transfers to the the runtime and implies that

$$\frac{\texttt{time}(\texttt{task}(f))}{\texttt{time}(f)} \leq \frac{\sum_{i=1}^{l}\texttt{time}(\texttt{task}(\psi_i))}{\sum_{i=1}^{l}\texttt{time}(\psi_i)} \leq \sup_{\psi\in\Psi}\frac{\texttt{time}(\texttt{task}(\psi_i))}{\texttt{time}(\psi_i)} \tag{A.21}$$

where the supremum becomes a maximum for the case of Ψ defined in (A.17).

We are now ready to state the fundamental complexity bounds for the two main modes of algorithm differentiation.

Proposition A.4.1 (Complexity of forward mode differentiation)**.** *For every $x \in \mathcal{X}$ and seed direction $\dot{x} \in \mathbb{R}^n$, executing the procedure listed in Table A.3 has a runtime*

$$\texttt{time}([f(x), \dot{f}(x,\dot{x})]) \leq \omega_{\text{tang}}\texttt{time}(f) \quad \textit{with} \quad \omega_{\text{tang}} \in [2, 5/2] \tag{A.22}$$

and requires memory

$$\mathtt{mem}([f(x), \dot{f}(x, \dot{x})]) \leq 2\mathtt{mem}(f). \tag{A.23}$$

Proposition A.4.2 (Complexity of reverse mode differentiation)**.** *For every $x \in \mathcal{X}$ and weight functional $\bar{y} \in \mathbb{R}^m$, executing the procedure listed in Table A.4 has a runtime*

$$\mathtt{time}([f(x), \bar{f}(x, \bar{y})]) \leq \omega_{\text{grad}}\mathtt{time}(f) \quad \textit{with} \quad \omega_{\text{tang}} \in [3, 4] \tag{A.24}$$

and requires memory

$$\mathtt{mem}([f(x), \bar{f}(x, \bar{y})]) \sim l. \tag{A.25}$$

where $l \in \mathbb{N}$ is the number of elementals that compose f.

Thus, with the simple temporal complexity model described above, forward mode differentiation may be faster than reverse mode, as we will verify in practice for the case of computing hypergradient (cf. Figure 6.1) – the exact values of ω_{tang} and ω_{grad} depending on the system dependent constants of the basic operation (A.20). In the remainder of the section we give a sketch of the proof of (A.22) and refer the reader to [Griewank and Walther, 2008] for the rest.

The proof revolves around the concept of *bounded complexity* on Span(Ψ). For finite Ψ and sub-additive tasks there exists a square matrix C_{task} (that depends on the particular library) such that

$$\mathtt{work}(\mathtt{task}'(\psi)) \leq C_{\mathtt{task}}\mathtt{work}(\psi) \qquad \text{for all} \quad \psi \in \Psi. \tag{A.26}$$

This means that the temporal complexity of executing an additive task (on the elementals) can grow at most linearly with respect to the complexity of evaluating the elementals themselves. The property of bounded complexity carries over to functions in Span(Ψ), since

$$\begin{aligned} \mathtt{work}(\mathtt{task}(f)) &\leq \sum_{i=1}^{l} \mathtt{work}(\mathtt{task}'(\psi_i)) \\ &\leq \sum_{i=1}^{l} C_{\mathtt{task}}\mathtt{work}(\psi_i) = C_{\mathtt{task}}\mathtt{work}(f) \end{aligned} \tag{A.27}$$

Concerning the runtime one can show by using (A.21) that (A.27) implies

$$\mathtt{time}(\mathtt{task}(f)) \leq \omega_{\mathtt{task}} \mathtt{time}(f).$$

This proves the first part of the bound (A.22). The coefficient $\omega_{\mathtt{task}}$ is computed as

$$\omega_{\mathtt{task}} = \max_{\psi \in \Psi} \frac{\omega^{\mathsf{T}} \mathtt{work}(\mathtt{task}'(\psi))}{\omega^{\mathsf{T}} \mathtt{work}(\psi)} \leq \left\| \mathrm{diag}(\omega) C_{\mathtt{task}} \mathrm{diag}(\omega)^{-1} \right\|_1 .$$

which, for our case, becomes simply

$$\omega_{\mathtt{task}} = \max_{1 \leq i \leq 4} \frac{\omega^{\mathsf{T}} W_{\mathtt{task}} e_i}{\omega^{\mathsf{T}} W_{\mathtt{eval}} e_i} \tag{A.28}$$

where $W_{\mathtt{task}}$ is the matrix in (A.18). In order to prove the second prat of the statement (A.22) we only need to compute $W_{\mathtt{task}}$, with $\mathtt{task} = \mathtt{tang}$, which means computing the measures $\mathtt{work}([\psi_i, \dot{\psi}_i])$ for the four elementals in our library.

1. For the constant assignment C, one has $\dot{\psi}_c = 0$ which requires one additional store w.r.t. evaluating ψ_c.

2. For addition and subtraction, given bidimensional inputs and tangents u, $\dot{u}$, one has $\dot{v} = \dot{\psi}_{\pm}(u, \dot{u}) = \dot{u}_1 \pm \dot{u}_2$, requiring two (scalar) fetches for $\dot{u}$, an `adds` operation and a store of the result $\dot{v}$. Therefore $\mathtt{work}([\psi_{\pm}, \dot{\psi}_{\pm}]) = 2\mathtt{work}(\psi_{\pm})$.

3. Similarly to the previous case $\dot{v} = \dot{\psi}_*(u, \dot{u}) = \dot{u}_1 * u_2 + u_1 * \dot{u}_2$ which result in $\mathtt{work}(\dot{\psi}_*) = (5, 1, 2, 0)^{\mathsf{T}}$. Note, however, that fetching u is already executed when evaluating ψ_*, thus $\mathtt{work}([\psi_*, \dot{\psi}_*]) = (6, 1, 3, 0)^{\mathsf{T}}$.

4. Finally, differentiating nonlinearities requires $\mathtt{work}([\psi_\sigma, \dot{\psi}_\sigma]) = (4, 0, 1, 2)$ since, for scalar input and tangent, $\dot{v} = \dot{\psi}_\sigma(u, \dot{u}) = \psi'(u) * \dot{u}$. As in the previous case the cost of fetching u is counted as one `moves`, as the operation is carried out when evaluating the nonlinearity.

In summary, the task complexity matrix is

$$W_{\mathtt{tang}} = \begin{pmatrix} 2 & 6 & 6 & 4 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

which can be plugged in (A.28) together with $W_{\mathtt{eval}}$ from (A.18), leading to the interval estimation $\omega_{\mathtt{tang}} \in [2, 5/2]$ when taking into account the conditions for the runtime cost coefficients ω given in (A.20).

Appendix B

Cross-validation and Bilevel Programming

We note that the (approximate) bilevel programming framework easily accommodates also estimations of the generalization error generated by a cross-validation procedures. We describe here the case of K-fold cross-validation, which includes also leave-one-out cross validation.

Let $D = \{(x_i, y_i)\}_{i=1}^n$ be the set of available data; K-fold cross validation, with $K \in \{1, 2, \ldots, N\}$ consists in partitioning D in K subsets $\{D^j\}_{j=1}^K$ and fit as many models g_{w^j} on training data $D_{\text{tr}}^j = \bigcup_{i \neq j} D^i$. The models are then evaluated on $D_{\text{val}}^j = D^j$. Denoting by $w = (w^j)_{j=1}^K$ the vector of stacked weights, the K-fold cross validation error is given by

$$E(w, \lambda) = \frac{1}{K} \sum_{j=1}^{K} E^j(w^j, \lambda)$$

where $E^j(w^j, \lambda) = \sum_{(x,y) \in D^j} \ell(g_{w^j}(x), y)$. E can be treated as the outer objective in the bilevel framework, while the inner objective L_λ may be given by the sum of regularized empirical errors over each D_{tr}^j for the K models. Under this perspective, a K-fold cross-validation procedure closely resemble the bilevel problem for ML formulated in Sec. 5.2.2, where, in this case, the meta-distribution collapses on the data (ground) distribution and the episodes are sampled from the same dataset of points.

By following the procedure outlined in Sec. 5.3 we can approximate the minimization of L_λ with T steps of an optimization dynamics and compute the hypergradient of

$f_T(\lambda) = \frac{1}{K}\sum_j E^j(w_T^j, \lambda)$ by training the K models and proceed with either forward or reverse differentiation. The models may be fitted sequentially, in parallel or stochastically. Specifically, in this last case, one can repeatedly sample one fold at a time (or possibly a mini-batch of folds) and compute a stochastic hypergradient that can be used in a SGD procedure in order to minimize f_T. At last, we note that similar ideas for leave-one out cross-validation error are developed in [Beirami et al., 2017], where the hypergradient of an approximation of the outer objective is computed by means of the implicit function theorem.

Appendix C

Proofs of Results in Chapter 6

In this appendix chapter we provide proofs of the results of Chapter 6.

C.1 Approximation Proprieties of Iterative Approach

Proof of Theorem 6.2.1. Since Λ is compact, it follows from Weierstrass theorem that a sufficient condition for the existence of minimizers is that f is continuous. Thus, let $\bar{\lambda} \in \Lambda$ and let $(\lambda_n)_{n\in\mathbb{N}}$ be a sequence in Λ such that $\lambda_n \to \bar{\lambda}$. We prove that $f(\lambda_n) = E(w(\lambda_n), \lambda_n) \to E(w(\bar{\lambda}), \bar{\lambda}) = f(\bar{\lambda})$. Since $(w(\lambda_n))_{n\in\mathbb{N}}$ is bounded, there exists a subsequence $(w_{k_n})_{n\in\mathbb{N}}$ such that $w_{k_n} \to \bar{w}$ for some $\bar{w} \in \mathbb{R}^d$. Now, since $\lambda_{k_n} \to \bar{\lambda}$ and the map $(w, \lambda) \mapsto L_\lambda(w)$ is jointly continuous, we have

$$\forall\, w \in \mathbb{R}^d, \quad L_{\bar{\lambda}}(\bar{w}) = \lim_n L_{\lambda_{k_n}}(w_{k_n}) \leq \lim_n L_{\lambda_{k_n}}(w) = L_{\bar{\lambda}}(w).$$

Therefore, $\bar{w}$ is a minimizer of $L_{\bar{\lambda}}$ and hence $\bar{w} = w(\bar{\lambda})$. This prove that $(w_{\lambda n})_{n\in\mathbb{N}}$ is a bounded sequence having a unique cluster point. Hence $(w_{\lambda n})_{n\in\mathbb{N}}$ is convergence to its unique cluster point, which is $w(\bar{\lambda})$. Finally, since $(w(\lambda_n), \lambda_n) \to (w(\bar{\lambda}), \bar{\lambda})$ and E is jointly continuous, we have $E(w(\lambda_n), \lambda_n) \to E(w(\bar{\lambda}), \bar{\lambda})$ and the statement follows. □

We recall a fundamental fact concerning the stability of minima and minimizers in optimization problems [Dontchev and Zolezzi, 1993]. We provide the proof for completeness.

Theorem C.1.1 (Convergence)**.** *Let φ_T and φ be lower semicontinuous functions defined on a compact set Λ. Suppose that φ_T converges uniformly to φ on Λ as*

$T \to +\infty$. *Then*

(a) $\inf \varphi_T \to \inf \varphi$,

(b) $\operatorname{argmin} \varphi_T \to \operatorname{argmin} \varphi$, *meaning that, for every* $(\lambda_T)_{T \in \mathbb{N}}$ *such that* $\lambda_T \in \operatorname{argmin} \varphi_T$, *we have that:*

- $(\lambda_T)_{T \in \mathbb{N}}$ *admits a convergent subsequence;*
- *for every subsequence* $(\lambda_{K_T})_{T \in \mathbb{N}}$ *such that* $\lambda_{K_T} \to \bar{\lambda}$, *we have* $\bar{\lambda} \in \operatorname{argmin} \varphi$.

Proof. Let $(\lambda_T)_{T \in \mathbb{N}}$ be a sequence in Λ such that, for every $T \in \mathbb{N}$, $\lambda_T \in \operatorname{argmin} \varphi_T$. We prove that

1) $(\lambda_T)_{T \in \mathbb{N}}$ admits a convergent subsequence.

2) for every subsequence $(\lambda_{K_T})_{T \in \mathbb{N}}$ such that $\lambda_{K_T} \to \bar{\lambda}$, we have $\bar{\lambda} \in \operatorname{argmin} \varphi$ and $\varphi_{K_T}(\lambda_{K_T}) \to \inf \varphi$.

3) $\inf \varphi_T \to \inf \varphi$.

The first point follows from the fact that Λ is compact.
Concerning the second point, let $(\lambda_{K_T})_{T \in \mathbb{N}}$ be a subsequence such that $\lambda_{K_T} \to \bar{\lambda}$. Since φ_{K_T} converge uniformly to φ, we have

$$|\varphi_{K_T}(\lambda_{K_T}) - \varphi(\lambda_{K_T})| \leq \sup_{\lambda \in \Lambda} |\varphi_{K_T}(\lambda) - \varphi(\lambda)| \to 0.$$

Therefore, using also the continuity of φ, we have

$$\forall \lambda \in \Lambda, \quad \varphi(\bar{\lambda}) = \lim_T \varphi(\lambda_{K_T}) = \lim_T \varphi_{K_T}(\lambda_{K_T}) \leq \lim_T \varphi_{K_T}(\lambda) = \varphi(\lambda).$$

So, $\bar{\lambda} \in \operatorname{argmin} \varphi$ and $\varphi(\bar{\lambda}) = \lim_T \varphi_{K_T}(\lambda_{K_T}) \leq \inf \varphi = \varphi(\bar{\lambda})$, that is, $\lim_T \varphi_{K_T}(\lambda_{K_T}) = \inf \varphi$.
Finally, as regards the last point, we proceed by contradiction. If $(\varphi_T(\lambda_T))_{T \in \mathbb{N}}$ does not convergce to $\inf f$, then there exists an $\varepsilon > 0$ and a subsequence $(\varphi_{K_T}(\lambda_{K_T}))_{T \in \mathbb{N}}$ such that

$$|\varphi_{K_T}(\lambda_{K_T}) - \inf \varphi| \geq \varepsilon, \quad \forall T \in \mathbb{N} \tag{C.1}$$

Now, let $(\lambda_{K_T^{(1)}})$ be a convergent subsequence of $(\lambda_{K_T})_{T\in\mathbb{N}}$. Suppose that $\lambda_{K_T^{(1)}} \to \bar{\lambda}$. Clearly $(\lambda_{K_T^{(1)}})$ is also a subsequence of $(\lambda_T)_{T\in\mathbb{N}}$. Then, it follows from point 2) above that $\varphi_{K_T^{(1)}}(\lambda_{K_T^{(1)}}) \to \inf \varphi$. This latter finding together with equation (C.1) gives a contradiction. □

Proof of Theorem 6.2.2. Since $E(\cdot,\lambda)$ is uniformly Lipschitz continuous, there exists $\nu > 0$ such that for every $T \in \mathbb{N}$ and every $\lambda \in \Lambda$

$$
\begin{aligned}
|f_T(\lambda) - f(\lambda)| &= |E(w_T(\lambda),\lambda) - E(w(\lambda),\lambda)| \\
&\leq \nu \|w_T(\lambda) - w(\lambda)\|.
\end{aligned}
$$

It follows from assumption (vi) that $f_T(\lambda)$ converges to $f(\lambda)$ uniformly on Λ as $T \to +\infty$. Then the statement follows from Theorem C.1.1 □

C.2 Error Bounds for Hypergradient Approximation

In the following technical lemma we give two results which are fundamental for the proofs of the bounds for the iterative and fixed-point approach. The first result is standard (see [Polyak, 1987b], Lemma 1, Section 2.2).

Lemma C.2.1. *Let $(u_k)_{k\in\mathbb{N}}$ and $(\tau_k)_{k\in\mathbb{N}}$ be two sequences of real non-negative numbers and let $q \in [0,\infty)$. Suppose that, for every $k \in \mathbb{N}$, with $k \geq 1$,*

$$
u_k \leq q u_{k-1} + \tau_{k-1}. \tag{C.2}
$$

Then, the following hold.

1. *If $(\tau_k)_{k\in\mathbb{N}} \equiv \tau$, then $u_k \leq q^k u_0 + \tau(1-q^k)/(1-q)$.*

2. *If, for every integer $k \geq 1$, $\tau_k \leq q\tau_{k-1}$, then $u_k \leq q^k u_0 + kq^{k-1}\tau_0$.*

Proof. Let $k \in \mathbb{N}$, with $k \geq 1$. Then, we have

$$\begin{aligned} u_k &\leq q u_{k-1} + \tau_{k-1} \\ &\leq q(q u_{k-2} + \tau_{k-2}) + \tau_{k-1} \\ &= q^2 u_{k-2} + (\tau_{k-1} + q\tau_{k-2}) \\ &\vdots \\ &\leq q^k u_0 + \sum_{i=0}^{k-1} q^i \tau_{k-1-i}. \end{aligned} \tag{C.3}$$

Point 1: Suppose that $(\tau_k)_{k\in\mathbb{N}} \equiv \tau$. Then it follows from (C.3) that $u_k \leq q^k u_0 + \tau \sum_{i=0}^{k-1} q^i = q^k u_0 + \tau(1-q^k)/(1-q)$.

Point 2: Suppose that, for every integer $k \geq 1$, $\tau_k \leq q\tau_{k-1}$. Then, for every integers k, i with $i \leq k-1$, we have $\tau_{k-1-i} \leq q^{k-1-i}\tau_0$, which substituted into (C.3) yields

$$u_k \leq q^k u_0 + \sum_{i=0}^{k-1} q^i q^{k-1-i} \tau_0$$

and Point 2 follows. □

Proof of Proposition 6.3.3. We assume that $(w_T(\lambda))_{T\in\mathbb{N}}$ is defined through the iteration

$$w_T(\lambda) = \Phi(w_{t-1}(\lambda), \lambda) \tag{C.4}$$

starting from $w_0(\lambda) = 0 \in \mathbb{R}^d$. Let $t \in \mathbb{N}$ with $t \geq 1$. Then, the mapping $\lambda \mapsto w_t(\lambda)$ is differentiable since it is a composition of differentiable functions and $\partial_\lambda w(\lambda)$ exists for the implicit function theorem. Differentiating the lower-level equation in (6.6) and the recursive equation in (C.4), we get

$$\begin{aligned} \partial_\lambda w_t(\lambda) &= \partial_w \Phi(w_{t-1}(\lambda), \lambda) \partial_\lambda w_{t-1}(\lambda) + \partial_\lambda \Phi(w_{t-1}(\lambda), \lambda) \\ \partial_\lambda w(\lambda) &= \partial_w \Phi(w(\lambda), \lambda) \partial_\lambda w(\lambda) + \partial_\lambda \Phi(w(\lambda), \lambda). \end{aligned} \tag{C.5}$$

Therefore, we get

$$\begin{aligned}\|\partial_\lambda w_t(\lambda) - \partial_\lambda w(\lambda)\| \leq &\|\partial_w \Phi(w_{t-1}(\lambda), \lambda) - \partial_w \Phi(w(\lambda), \lambda)\| \, \|\partial_\lambda w(\lambda)\| \\ &+ \|\partial_w \Phi(w_{t-1}(\lambda), \lambda)\| \, \|\partial_\lambda w_{t-1}(\lambda) - \partial_\lambda w(\lambda)\| \\ &+ \|\partial_\lambda \Phi(w_{t-1}(\lambda), \lambda) - \partial_\lambda \Phi(w(\lambda), \lambda)\|\end{aligned}$$

and hence, we derive from Assumption C, Equations (6.7) and 6.9 and Lemmas 6.3.2, that

$$\begin{aligned}\|\partial_\lambda w_t(\lambda) - \partial_\lambda w(\lambda)\| \leq &(\nu_{2,\lambda} + \nu_{1,\lambda}\nu_{\Phi,\lambda}/(1-\rho_\lambda)) \, \|w_{t-1}(\lambda) - w(\lambda)\| \\ &+ \rho_\lambda \, \|\partial_\lambda w_{t-1}(\lambda) - \partial_\lambda w(\lambda)\| .\end{aligned}$$

Then, setting $p := \nu_{2,\lambda} + \nu_{1,\lambda}\nu_{\Phi,\lambda}/(1-\rho_\lambda)$, $\Delta_t := \|w_t(\lambda) - w(\lambda)\|$ and $\Delta'_t := \|\partial_\lambda w_t(\lambda) - \partial_\lambda w(\lambda)\|$, we get

$$\Delta_t \leq \rho_\lambda \Delta_{t-1} \quad \text{and} \quad \Delta'_t \leq \rho_\lambda \Delta'_{t-1} + p\Delta_{t-1}.$$

Therefore, it follows from Lemma C.2.1 Point 2 (with $u_t = \Delta'_t$ and $\tau_t = p\Delta_t$) that

$$\Delta'_t \leq \rho_\lambda^t \Delta'_0 + t\rho_\lambda^{t-1} p \Delta_0 \leq \frac{\nu_{\Phi,\lambda}}{1-\rho_\lambda}\rho_\lambda^t + pD_\lambda t \rho_\lambda^{t-1},$$

where in the last inequality we used the bounds (see (C.5) and Lemmas 6.3.2 and 6.9)

$$\begin{aligned}\Delta_0 &= \|w(\lambda) - w_0(\lambda)\| = \|w(\lambda)\| \leq D_\lambda \\ \Delta'_0 &= \|\partial_\lambda w(\lambda) - \partial_\lambda w_0(\lambda)\| = \|\partial_\lambda w(\lambda)\| \leq \frac{\nu_{\Phi,\lambda}}{1-\rho_\lambda}.\end{aligned} \tag{C.6}$$

Recalling the definitions of p and Δ'_t, (6.10) follows. □

Proof of Theorem 6.3.4. It follows from the definitions of f_T and f in Equation (6.6), respectively, and the chain rule for differentiation that

$$\nabla f_T(\lambda) = \nabla_\lambda E(w_t(\lambda),\lambda) + \partial_\lambda w_t(\lambda)^\top \nabla_w E(w_t(\lambda),\lambda)$$
$$\nabla f(\lambda) = \nabla_\lambda E(w(\lambda),\lambda) + \partial_\lambda w(\lambda)^\top \nabla_w E(w(\lambda),\lambda).$$

Therefore,

$$\begin{aligned}\|\nabla f_T(\lambda) - \nabla f(\lambda)\| \leq & \|\nabla_\lambda E(w_t(\lambda),\lambda) - \nabla_\lambda E(w(\lambda),\lambda)\| \\ & + \|\partial_\lambda w(\lambda)\| \, \|\nabla_w E(w_t(\lambda),\lambda) - \nabla_w E(w(\lambda),\lambda)\| \\ & + \|\partial_\lambda w_t(\lambda) - \partial_\lambda w(\lambda)\| \, \|\nabla_w E(w_t(\lambda),\lambda)\| .\end{aligned}$$

Now, we note that $\|w_t(\lambda)\| \leq \|w_t(\lambda) - w(\lambda)\| + \|w(\lambda)\| \leq (\rho_\lambda^K + 1)\,\|w(\lambda)\| \leq 2D_\lambda$. Therefore, it follows from Assumption C.2, Lemma 6.3.2 and Equation 6.9 that

$$\|\nabla f_T(\lambda) - \nabla f(\lambda)\| \leq (\xi_{2,\lambda} + \xi_{1,\lambda}\nu_{\Phi,\lambda}/(1-\rho_\lambda))\rho_\lambda^K D_\lambda + \nu_{E,\lambda}\,\|\partial_\lambda w(\lambda) - \partial_\lambda w_t(\lambda)\|,$$

where we used $\|w_t(\lambda) - w(\lambda)\| \leq \rho_\lambda^t \,\|w_0(\lambda) - w(\lambda)\| = \rho_\lambda^t \,\|w(\lambda)\| \leq \rho_\lambda^t D_\lambda$. Then, (6.12) follows from Proposition 6.3.3. □

Fixed-point Method. The following two propositions allow us to derive the iteration complexity bound for the fixed-point method.

Proposition C.2.2. *Suppose that* (6.13) *holds. Let* $\lambda \in \Lambda, T \in \mathbb{N}$. *Let* $u_{T,0}(\lambda) = 0 \in \mathbb{R}^{d\times n}$ *and for every integer* $k \geq 1$,

$$u_{T,K}(\lambda) = \partial_w \Phi(w_T(\lambda),\lambda) u_{T,K-1}(\lambda) + \partial_\lambda \Phi(w_T(\lambda),\lambda).$$

Then, for every $k \in \mathbb{N}$,

$$u_{T,K}(\lambda)^\top \nabla_w E(w_T(\lambda),\lambda) = \partial_\lambda \Phi(w_T(\lambda),\lambda)^\top q_{T,K}(\lambda). \tag{C.7}$$

Proof. We set $Y = \partial_1 \Phi(w_T(\lambda),\lambda) \in \mathbb{R}^{d\times d}$, $C = \partial_\lambda \Phi(w_T(\lambda),\lambda) \in \mathbb{R}^{d\times n}$, and $b =$

$\nabla_1 E(w_T(\lambda), \lambda) \in \mathbb{R}^d$. Let $K \in \mathbb{N}$, $K \geq 1$. Then,

$$\begin{aligned}
u_{T,K}(\lambda) &= Y u_{T,K-1}(\lambda) + C \\
&= Y^2 u_{T,K-2}(\lambda) + (1+Y)C \\
&\vdots \\
&= Y^K u_{T,0}(\lambda) + \sum_{i=0}^{K-1} Y^i C = \sum_{i=0}^{K-1} Y^i C.
\end{aligned}$$

In the same way, it follows from (6.13) that $q_{T,K}(\lambda) = Y^\top v_{T,K-1}(\lambda) + b = \sum_{i=0}^{K-1} (Y^\top)^i b$. Therefore, we have

$$u_{T,K}(\lambda)^\top b = C^\top \left(\sum_{i=0}^{K-1} Y^i \right)^\top b = C^\top \sum_{s=i}^{K-1} (Y^\top)^i b = C^\top q_{T,K}(\lambda)$$

and the statement follows. □

Using Proposition C.2.2, for the fixed-point method we can write

$$\nabla f_{T,K}(\lambda) = \nabla_2 E(w_T(\lambda), \lambda) + u_{T,K}(\lambda)^\top \nabla_1 E(w_T(\lambda), \lambda). \tag{C.8}$$

Then a result similar to Proposition 6.3.3 can be derived.

Proposition C.2.3. *Suppose that Assumption C holds. Let $\lambda \in \Lambda$ and $(u_{T,K}(\lambda))_{K\in\mathbb{N}}$ be defined as in Proposition C.2.2. Then, for every $T, K \in \mathbb{N}$, with $T \geq 1$,*

$$\left\| u_{T,K}(\lambda) - \partial_\lambda w(\lambda) \right\| \leq \left(\nu_{2,\lambda} + \nu_{1,\lambda} \frac{\nu_{\Phi,\lambda}}{(1-\rho_\lambda)} \right) \frac{D_\lambda (1-\rho_\lambda^K)}{1-\rho_\lambda} \rho_\lambda(T) + \frac{\nu_{\Phi,\lambda}}{1-\rho_\lambda} \rho_\lambda^K.$$

Proof. Let $T, K \in \mathbb{N}$, with $T, K \geq 1$. Recalling that

$$\begin{aligned}
u_{T,K}(\lambda) &= \partial_w \Phi(w_T(\lambda), \lambda) u_{T,K-1}(\lambda) + \partial_\lambda \Phi(w_T(\lambda), \lambda) \\
\partial_\lambda w(\lambda) &= \partial_w \Phi(w(\lambda), \lambda) \partial_\lambda w(\lambda) + \partial_\lambda \Phi(w(\lambda), \lambda)
\end{aligned}$$

we can bound the norm of the difference as follows

$$\begin{aligned}\|u_{T,K}(\lambda)-\partial_\lambda w(\lambda)\| \leq & \|\partial_w\Phi(w_T(\lambda),\lambda)-\partial_w\Phi(w(\lambda),\lambda)\|\,\|\partial_\lambda w(\lambda)\| \\ & +\|\partial_w\Phi(w_T(\lambda),\lambda)\|\left\|u_{T,K-1}(\lambda)-\partial_\lambda w(\lambda)\right\| \\ & +\|\partial_\lambda\Phi(w_T(\lambda),\lambda)-\partial_\lambda\Phi(w(\lambda),\lambda)\| \\ \leq & (\nu_{2,\lambda}+\nu_{1,\lambda}\nu_{\Phi,\lambda}/(1-\rho_\lambda))\,\|w_T(\lambda)-w(\lambda)\| \\ & +\rho_\lambda\left\|u_{T,K-1}(\lambda)-\partial_\lambda w(\lambda)\right\|,\end{aligned}$$

which gives a recursive inequality. Then, setting $p := \nu_{2,\lambda}+\nu_{1,\lambda}\nu_{\Phi,\lambda}/(1-\rho_\lambda)$, $\Delta_T := \|w_T(\lambda)-w(\lambda)\|$ and $\hat{\Delta}'_k := \left\|u_{T,K}(\lambda)-\partial_\lambda w(\lambda)\right\|$, we have $\hat{\Delta}'_k \leq \rho_\lambda\hat{\Delta}'_{K-1}+p\Delta_T$. Therefore, it follows from Lemma C.2.11 with $\tau = p\Delta_T$, that

$$\hat{\Delta}'_k \leq \rho_\lambda^K\hat{\Delta}'_0 + p\Delta_T\frac{1-\rho_\lambda^K}{1-\rho_\lambda} \leq \frac{\nu_{\Phi,\lambda}}{1-\rho_\lambda}\rho_\lambda^K + pD_\lambda\rho_\lambda(T)\frac{1-\rho_\lambda^K}{1-\rho_\lambda}.$$

The statement follows. □

Proof of Theorem 6.3.5. Let $T\in\mathbb{N}$ with $T\geq 1$ and let $(u_{T,K}(\lambda))_{k\in\mathbb{N}}$ be defined as in Proposition C.2.2. Then, the difference between exact and approximate gradients can be bound as follows

$$\begin{aligned}\|\nabla f_{T,K}(\lambda)-\nabla f(\lambda)\| \leq & \|\nabla_\lambda E(w_T(\lambda),\lambda)-\nabla_\lambda E(w(\lambda),\lambda)\| \\ & +\|\partial_\lambda w(\lambda)\|\,\|\nabla_w E(w_T(\lambda),\lambda)-\nabla_w E(w(\lambda),\lambda)\| \\ & +\left\|\partial_\lambda w(\lambda)-u_{T,K}(\lambda)\right\|\,\|\nabla_w E(w_T(\lambda),\lambda)\|.\end{aligned}$$

Now note that $\|w_t(\lambda)\| \leq \|w_t(\lambda)-w(\lambda)\|+\|w(\lambda)\| \leq (\rho_\lambda(T)+1)\,\|w(\lambda)\| \leq 2D_\lambda$. Then it follows from the assumptions and Lemmas 6.9 and 6.3.2 that

$$\|\nabla f_{T,K}(\lambda)-\nabla f(\lambda)\| \leq \left(\xi_{2,\lambda}+\frac{\xi_{1,\lambda}\nu_{\Phi,\lambda}}{1-\rho_\lambda}\right)\rho_\lambda(T)D_\lambda + \nu_{E,\lambda}\left\|u_{T,K}(\lambda)-\partial_\lambda w(\lambda)\right\|,$$

and the last term can be bounded using Proposition C.2.3. □

Appendix D

On the Straight-through Estimator

LDS borrows from an heuristic solution proposed before [Bengio et al., 2013], at the cost of having biased (hyper)gradient estimates. Given a function $h(z)$, where $z \sim P_\theta$ is a discrete random variable whose distribution depends on parameters θ, the STE is a technique that consists in computing a biased estimator of the gradient of $\ell(\theta) = \mathbb{E}_{z\sim P_\theta} h(z)$ by using an inexact, but smooth, reparameterization for z, together with the application of an approximate version of (9.10).

When z is Bernoulli distributed, such reparameterization can be simply chosen[1] as $z = \mathsf{sp}(\theta, \varepsilon) = \theta$ in which case the STE boils down to

$$\hat{g}(z) = \frac{\partial h(z)}{\partial z}, \quad z \sim P_\theta. \tag{D.1}$$

If h is *a smooth function of* z, Eq. D.1 is well defined and yields, in general, non-zero quantities. This operation may be viewed under different angles: e.g. as "setting" $\frac{\partial z}{\partial \theta}$ to the identity, or, as "ignoring" the hard thresholds in the backward pass. $\hat{g}$ is a random variable that depends, again, from θ. The true gradient $\nabla \ell(\theta)$ can be estimated by drawing one or more samples from $\hat{g}$.

As an illustrative example, consider the very simple case where $h(z) = (az - b)^2/2$ for scalars a and b, with $z \sim \mathrm{Ber}(\theta)$, $\theta \in [0, 1]$. The gradient (derivative) of $\mathbb{E}\,[h]$ w.r.T.

[1]An exact, but discontinuous reparameterization for $z \sim \mathrm{Ber}(\theta)$ is, for instance, $z = \mathsf{sp}(\theta, \varepsilon) = H(\theta - \varepsilon)$ for $\varepsilon \sim \mathcal{U}(0, 1)$, where H is the Heaviside function and $\mathcal{U}$ is the (continuous) uniform distribution.

θ can be easily computed as

$$\frac{\partial}{\partial\theta}\mathbb{E}_{z\sim\text{Ber}(\theta)}\left[h(z)\right] = \; = \frac{\partial}{\partial\theta}\left[\theta\frac{(a-b)^2}{2} + (1-\theta)\frac{(-b)^2}{2}\right] = \frac{a^2}{2} - ab,$$

whereas the corresponding straight-through estimator, which is a random variable, is given by

$$\hat{g}(z) = \frac{\partial h(z)}{\partial z} = (az-b)a, \quad z \sim \text{Ber}(\theta).$$

One has, however, that

$$\mathbb{E}_{z\sim\text{Ber}(\theta)}\left[\hat{g}(z)\right] = \theta(a-b)a + (1-\theta)(-ab) = \theta a^2 - ab,$$

resulting in $\hat{g}$ to be biased for $\theta \neq \frac{1}{2}$.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Fabio Aiolli and Michele Donini. Easymkl: a scalable multiple kernel learning algorithm. *Neurocomputing*, 169:215–224, 2015.

Hirotogu Akaike. Information theory and an extension of the maximum likelihood principle. In *Selected papers of hirotugu akaike*, pages 199–213. Springer, 1998.

Luis B Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Proceedings, 1st First International Conference on Neural Networks*, volume 2, pages 609–618, 1987.

Luís B Almeida, Thibault Langlois, José D Amaral, and Alexander Plakhov. Parameter adaptation in stochastic optimization. In *On-line learning in neural networks*, pages 111–134. Cambridge University Press, 1999.

Brandon Amos. *Differentiable optimization-based modeling for machine learning*. PhD thesis, PhD thesis. Carnegie Mellon University, 2019.

Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 136–145, 2017.

L Nonboe Andersen, Jan Larsen, Lars Kai Hansen, and Mads Hintz-Madsen. Adaptive regularization of neural classifiers. In *Neural Networks for Signal Processing VII. Proceedings of the 1997 IEEE Signal Processing Society Workshop*, pages 24–33. IEEE, 1997.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016. URL `http://papers.nips.cc/paper/6461-learning-to-learn-by-gradient-descent-by-gradient-descent`.

Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml. *International Conference on Learning Representations*, 2019.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Leonardo Badino. Phonetic context embeddings for dnn-hmm phone recognition. In *Proceedings of Interspeech*, pages 405–409, 2016.

Leonardo Badino. Personal communication, 2017.

Leonardo Badino, Luca Franceschi, Raman Arora, Michele Donini, and Massimiliano Pontil. A speaker adaptive dnn training approach for speaker-independent acoustic inversion. In *Proceedings of Interspeech*, pages 984–988, 2017.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. In *Advances in Neural Information Processing Systems*, pages 688–699, 2019.

Yu Bai, Minshuo Chen, Pan Zhou, Tuo Zhao, Jason D Lee, Sham Kakade, Huan Wang, and Caiming Xiong. How important is the train-validation split in meta-learning? *arXiv preprint arXiv:2010.05843*, 2020.

Pierre Baldi and Kurt Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1):53–58, 1989.

Jonathan F. Bard. *Practical bilevel optimization: algorithms and applications*, volume 30. Springer Science & Business Media, 2013.

Andrew G Barto and Richard S Sutton. Goal seeking components for adaptive intelligence: An initial assessment. Technical report, Massachussets Univ Amherst, 1981.

J. Baxter. Theoretical models of learning to learn. *Learning to learn*, pages 71–94, 1998.

Jonathan Baxter. Learning internal representations. In *Proceedings of the eighth annual conference on Computational learning theory*, pages 311–320. ACM, 1995.

Atilim Gunes Baydin, Robert Cornish, David Martínez-Rubio, Mark Schmidt, and Frank D. Wood. Online learning rate adaptation with hypergradient descent. In *International Conference on Learning Representations*, 2018a.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18 (153), 2018b.

Alex Beatson and Ryan P. Adams. Efficient optimization of loops and limits with randomized telescoping sums. *International Conference on Machine Learning*, 2019.

Sarah Bechtle, Artem Molchanov, Yevgen Chebotar, Edward Grefenstette, Ludovic Righetti, Gaurav Sukhatme, and Franziska Meier. Meta-learning via learned loss. *arXiv preprint arXiv:1906.05374*, 2019.

Harkirat Singh Behl, Atılım Güneş Baydin, Ran Gal, Philip HS Torr, and Vibhav Vineet. Autosimulate:(quickly) learning synthetic data generation. *ECCV*, 2020.

Ahmad Beirami, Meisam Razaviyayn, Shahin Shahrampour, and Vahid Tarokh. On optimal generalizability in parametric learning. In *Advances in Neural Information Processing Systems*, pages 3458–3468, 2017.

Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of Machine Learning Research*, 7: 2399–2434, 2006.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. *International Conference on Machine Learning*, 2017.

Y. Bengio, S. Bengio, and J. Cloutier. Learning a synaptic learning rule. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume ii, 1991.

Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900, 2000.

Yoshua Bengio. Learning deep architectures for AI. *Foundations and trends® in Machine Learning*, 2 (1):1–127, 2009.

Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012. URL `http://www.jmlr.org/papers/v13/bergstra12a.html`.

James Bergstra, Daniel Yamins, and David D. Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. *International Conference on Machine Learning*, 28:115–123, 2013.

James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.

Luca Bertinetto, João F Henriques, Jack Valmadre, Philip Torr, and Andrea Vedaldi. Learning feed-forward one-shot learners. In *Advances in neural information processing systems*, pages 523–531, 2016.

Luca Bertinetto, Joao F Henriques, Philip HS Torr, and Andrea Vedaldi. Meta-learning with differentiable closed-form solvers. *International Conference on Learning Representations*, 2019.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

Olivier Bousquet, Sylvain Gelly, Karol Kurach, Olivier Teytaud, and Damien Vincent. Critical hyper-parameters: No random, no cry. *arXiv preprint arXiv:1706.03200*, 2017.

Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. *International Conference on Learning Representations*, 2018.

Samuel H Brooks. A discussion of random methods for seeking maxima. *Operations research*, 6(2): 244–251, 1958.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. *AAAI*, 2018.

Rich Caruana. Multitask learning. In *Learning to learn*, pages 95–133. Springer, 1998. 02683.

Wei-Lun Chao, Han-Jia Ye, De-Chuan Zhan, Mark Campbell, and Kilian Q Weinberger. Revisiting meta-learning as supervised learning. *arXiv preprint arXiv:2002.00573*, 2020.

Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for support vector machines. *Machine learning*, 46(1-3):131–159, 2002.

Xinshi Chen, Yufei Zhang, Christoph Reisinger, and Le Song. Understanding deep architecture with reasoning layer. *Advances in Neural Information Processing Systems*, 33, 2020a.

Yihong Chen, Bei Chen, Xiangnan He, Chen Gao, Yong Li, Jian-Guang Lou, and Yue Wang. λopt: Learn to regularize recommender models in finer levels. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 978–986, 2019.

Yinbo Chen, Xiaolong Wang, Zhuang Liu, Huijuan Xu, and Trevor Darrell. A new meta-baseline for few-shot learning. *arXiv preprint arXiv:2003.04390*, 2020b.

Zhiyuan Chen and Bing Liu. Lifelong machine learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 12(3):1–207, 2018.

Paul Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. Transfer from simulation to real world through learning deep inverse dynamics model. *arXiv preprint arXiv:1610.03518*, 2016.

Scott Clark and Patrick Hayes. SigOpt Web page. `https://sigopt.com`, 2019. URL `https://sigopt.com`.

Frank H Clarke. *Optimization and nonsmooth analysis*, volume 5. Siam, 1990.

Benoît Colson, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Annals of operations research*, 153(1):235–256, 2007.

Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. Adanet: Adaptive structural learning of artificial neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 874–883. JMLR. org, 2017.

Neil E Cotter and Peter R Conwell. Fixed-weight networks can learn. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 553–559. IEEE, 1990.

Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

Nicola De Cao and Thomas Kipf. MolGAN: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018.

Michaël Defferrard, Kirell Benzi, Pierre Vandergheynst, and Xavier Bresson. FMA: A dataset for music analysis. In *18th International Society for Music Information Retrieval Conference*, 2017. URL `https://arxiv.org/abs/1612.01840`.

Giulia Denevi, Carlo Ciliberto, Dimitris Stamos, and Massimiliano Pontil. Learning to learn around a common mean. In *Advances in Neural Information Processing Systems*, pages 10169–10179, 2018a.

Giulia Denevi, Carlo Ciliberto, Dimitris Stamos, and Massimiliano Pontil. Incremental learning-to-learn with statistical guarantees. *UAI*, 2018b.

Giulia Denevi, Carlo Ciliberto, Riccardo Grazzi, and Massimiliano Pontil. Learning-to-learn stochastic gradient descent with biased regularization. In *Proc. 36th International Conference on Machine Learning*, volume 97 of *PMLR*, pages 1566–1575, 2019.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

Francesco Dinuzzo, Cheng S Ong, Gianluigi Pillonetto, and Peter V Gehler. Learning output kernels with block coordinate descent. In *Proceedings of the 28th International Conference on Machine Learning*, pages 49–56, 2011.

Jesse Dodge, Kevin Jamieson, and Noah A Smith. Open loop hyperparameter optimization and determinantal point processes. *arXiv preprint arXiv:1706.01566*, 2017.

Justin Domke. Generic methods for optimization-based modeling. In *Artificial Intelligence and Statistics*, pages 318–326, 2012.

Michele Donini, Luca Franceschi, Orchid Majumder, Massimiliano Pontil, and Paolo Frasconi. MARTHE: Scheduling the learning rate via online hypergradients. In *IJCAI*, 2020.

A. L. Dontchev and T. Zolezzi. *Well-posed optimization problems*, volume 1543 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1993.

Alberto Garcia Duran and Mathias Niepert. Learning graph representations with embedding propagation. In *Advances in Neural Information Processing Systems*, pages 5119–5130, 2017.

Harrison Edwards and Amos Storkey. Towards a Neural Statistician. *International Conference on Learning Representations*, 2017. URL `http://arxiv.org/abs/1606.02185`.

M A Efroymson. Multiple regression analysis. *Mathematical methods for digital computers*, pages 191–203, 1960.

Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, pages 1–5, 2013.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5 (1):17–60, 1960.

Theodoros Evgeniou, Charles A Micchelli, and Massimiliano Pontil. Learning multiple tasks with kernel methods. *Journal of machine learning research*, 6(Apr):615–637, 2005.

Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In *Advances in neural information processing systems*, pages 524–532, 1990.

Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *International Conference on Machine Learning*, 2018.

Li Fei-Fei, Rob Fergus, and Pietro Perona. One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence*, 28(4):594–611, 2006.

Matthias Feurer and Frank Hutter. Hyperparameter optimization. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Automatic Machine Learning: Methods, Systems, Challenges*, pages 3–38. Springer, 2018. In press, available at http://automl.org/book.

Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing Bayesian hyperparameter optimization via meta-learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1126–1135, 2017.

Ronald Aylmer Fisher. Design of experiments. *Br Med J*, 1(3923):554–554, 1936.

Rémi Flamary, Alain Rakotomamonjy, and Gilles Gasso. Learning constrained task similarities in graph-regularized multi-task learning. *Regularization, Optimization, Kernels, and Support Vector Machines*, page 103, 2014.

Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

Chuan-sheng Foo, Chuong B. Do, and Andrew Y. Ng. Efficient multiple hyperparameter learning for log-linear models. In *Advances in neural information processing systems*, pages 377–384, 2008.

Martin Ford. *Architects of Intelligence: The truth about AI from the people building it*. Packt Publishing Ltd, 2018.

Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1165–1173, 2017.

Luca Franceschi, Paolo Frasconi, Saverio Salzo, Riccardo Grazzi, and Massimiliano Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In *International Conference on Machine Learning*, pages 1563–1572, 2018a.

Luca Franceschi, Riccardo Grazzi, Massimiliano Pontil, Saverio Salzo, and Paolo Frasconi. Far-HO: A bilevel programming package for hyperparameter optimization and meta-learning. *AutoML workshop at International Conference on Machine Learning*, 2018b.

Luca Franceschi, Mathias Niepert, Massimiliano Pontil, and Xiao He. Learning discrete structures for graph neural networks. In *International Conference on Machine Learning*, pages 1972–1982, 2019.

Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A general framework for adaptive processing of data structures. *IEEE transactions on Neural Networks*, 9(5):768–786, 1998.

Jordan Frecon, Saverio Salzo, and Massimiliano Pontil. Bilevel learning of the group lasso structure. In *Advances in Neural Information Processing Systems 31*, pages 8311–8321, 2018.

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Springer series in statistics New York, 2001.

Frauke Friedrichs and Christian Igel. Evolutionary tuning of multiple svm parameters. *Neurocomputing*, 64:107–117, 2005.

Nicolo Fusi, Rishit Sheth, and Melih Elibol. Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems*, pages 3348–3357, 2018.

Alex Gammerman, Volodya Vovk, and Vladimir Vapnik. Learning by transduction. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 148–155, 1998.

Victor Garcia and Joan Bruna. Few-shot learning with graph neural networks. *International Conference on Learning Representations*, 2018.

John S. Garofolo, Lori F. Lamel, William M. Fisher, Jonathon G. Fiscus, and David S. Pallett. DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1. *NASA STI/Recon technical report*, 93, 1993.

Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle points: online stochastic gradient for tensor decomposition. In *Conference on Learning Theory*, pages 797–842, 2015.

Lise Getoor and Ben Taskar. *Introduction to statistical relational learning*, volume 1. MIT press Cambridge, 2007.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *International Conference on Machine Learning*, 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AIStat)*, pages 249–256, 2010.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016.

Diana F Gordon and Marie Desjardins. Evaluation and selection of biases in machine learning. *Machine learning*, 20(1-2):5–22, 1995.

Josif Grabocka, Randolf Scholz, and Lars Schmidt-Thieme. Learning surrogate losses. *arXiv preprint arXiv:1905.10108*, 2019.

Will Grathwohl, Dami Choi, Yuhuai Wu, Geoff Roeder, and David Duvenaud. Backpropagation through the void: Optimizing control variates for black-box gradient estimation. *International Conference on Learning Representations*, 2018.

Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in neural information processing systems*, pages 545–552, 2009.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Riccardo Grazzi, Luca Franceschi, , Massimilano Pontil, and Saverio Salzo. On the iteration complexity of hypergradient computation. In *International Conference on Machine Learning*, 2020.

Edward Grefenstette, Brandon Amos, Denis Yarats, Phu Mon Htut, Artem Molchanov, Franziska Meier, Douwe Kiela, Kyunghyun Cho, and Soumith Chintala. Generalized inner loop meta-learning. *arXiv preprint arXiv:1910.01727*, 2019.

Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, volume 105. Siam, 2008.

Aditya Grover, Aaron Zweig, and Stefano Ermon. Graphite: Iterative generative modeling of graphs. *International Conference on Machine Learning*, 2019.

Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

David Ha, Andrew Dai, and Quoc V. Le. HyperNetworks. *International Conference on Learning Representations*, 2017. URL `https://arxiv.org/abs/1609.09106`.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

Nikolaus Hansen and Andreas Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Proceedings of IEEE international conference on evolutionary computation*, pages 312–317. IEEE, 1996.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

Samantha Hansen. Using deep q-learning to control optimization hyperparameters. *arXiv preprint arXiv:1602.04062*, 2016.

Elad Hazan, Adam Klivans, and Yang Yuan. Hyperparameter optimization: A spectral approach. *International Conference on Learning Representations*, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a: overview of mini-batch gradient descent. *Coursera Lecture Notes*, 2012.

Geoffrey E Hinton and David C Plaut. Using fast weights to deblur old memories. In *Proceedings of the ninth annual conference of the Cognitive Science Society*, pages 177–186, 1987.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

Judy Hoffman, Daniel A Roberts, and Sho Yaida. Robust learning with jacobian regularization. *arXiv preprint arXiv:1908.02729*, 2019.

Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*, 2020.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Int. Conf. on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

Frank Hutter, Jrg Lcke, and Lars Schmidt-Thieme. Beyond Manual Tuning of Hyperparameters. *KI - Künstliche Intelligenz*, 29(4):329–337, November 2015. ISSN 0933-1875, 1610-1987. doi: 10.1007/s13218-015-0381-0. URL `http://link.springer.com/10.1007/s13218-015-0381-0`.

Ilija Ilievski, Taimoor Akhtar, Jiashi Feng, and Christine Annette Shoemaker. Efficient hyperparameter optimization for deep learning algorithms using deterministic rbf surrogates. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International Conference on Machine Learning*, 2015.

Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1 (4):295–307, 1988.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

Ashesh Jain, Swaminathan VN Vishwanathan, and Manik Varma. Spf-gmkl: generalized multiple kernel learning with a million kernels. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 750–758, 2012.

Vidit Jain and Erik Learned-Miller. Online domain adaptation of a pre-trained cascade of classifiers. In *CVPR 2011*, pages 577–584. IEEE, 2011.

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with Gumbel-softmax. *International Conference on Learning Representations*, 2017.

Pratik Jawanpuria, Maksim Lapin, Matthias Hein, and Bernt Schiele. Efficient output kernel learning for multiple tasks. In *Advances in Neural Information Processing Systems*, pages 1189–1197, 2015.

Bo Jiang, Ziyan Zhang, Doudou Lin, Jin Tang, and Bin Luo. Semi-supervised learning with graph learning-convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11313–11320, 2019.

Daniel D Johnson. Learning graphical state transitions. *International Conference on Learning Representations*, 2017.

Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

Lukasz Kaiser, Ofir Nachum, Aurko Roy, and Samy Bengio. Learning to remember rare events. *International Conference on Learning Representations*, 2017. URL `https://openreview.net/pdf?id=SJTQLdqlg`.

Zhuoliang Kang, Kristen Grauman, and Fei Sha. Learning with whom to share in multi-task feature learning. In *Proceedings of the 28th International Conference on Machine Learning*, pages 521–528, 2011.

S Sathiya Keerthi, Vikas Sindhwani, and Olivier Chapelle. An efficient method for gradient-based adaptation of hyperparameters in svm models. *Advances in Neural Information Processing Systems*, 19:673, 2007.

Aditya Khosla, Tinghui Zhou, Tomasz Malisiewicz, Alexei A Efros, and Antonio Torralba. Undoing the damage of dataset bias. In *European Conference on Computer Vision*, pages 158–171. Springer, 2012.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2015.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *International Conference on Learning Representations*, 2014.

Diederik P Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 2019.

Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, Max Welling, and Richard Zemel. Neural relational inference for interacting systems. *International Conference on Machine Learning*, 2018.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations*, 2017.

Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, pages 528–536, 2017.

Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *International Conference on Machine Learning Deep Learning Workshop*, volume 2, 2015. 00127.

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1885–1894, 2017.

Mikhail Konobeev, Ilja Kuzborskij, and Csaba Szepesvári. On optimality of meta-learning in fixed-design regression with weighted biased regularization. *Advances in Neural Information Processing Systems*, 2020.

Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830, 2017.

Wouter M Kouw and Marco Loog. An introduction to domain adaptation and transfer learning. *arXiv preprint arXiv:1812.11806*, 2018.

Steven G Krantz and Harold R Parks. *The implicit function theorem: history, theory, and applications*. Springer Science & Business Media, 2012.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *University of Toronto*, 2009.

Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, page 4, 2014.

David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Aaron Courville, and Chris Pal. Zoneout: Regularizing rnns by randomly preserving hidden activations. *International Conference on Learning Representations*, 2017.

G. Kunapuli, K.P. Bennett, Jing Hu, and Jong-Shi Pang. Classification model selection via bilevel programming. *Optimization Methods and Software*, 23(4):475–489, August 2008. ISSN 1055-6788, 1029-4937. doi: 10.1080/10556780802102586. URL `http://www.tandfonline.com/doi/abs/10.1080/10556780802102586`.

Ilja Kuzborskij and Christoph H Lampert. Data-dependent stability of stochastic gradient descent. *International Conference on Machine Learning*, 2018.

Ilja Kuzborskij, Francesco Orabona, and Barbara Caputo. From n to n+ 1: Multiclass transfer incremental learning. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 3358–3365, 2013.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40, 2017. ISSN 0140-525X, 1469-1825. doi: 10.1017/S0140525X16001837. URL `https://www.cambridge.org/core/journals/behavioral-and-brain-sciences/article/building-machines-that-learn-and-think-like-people/A9535B1D745A0377E16C590E14B94993`. 00152.

Serge Lang. *Fundamentals of differential geometry*, volume 191. Springer Science & Business Media, 2012.

Jan Larsen, Lars Kai Hansen, Claus Svarer, and M. Ohlsson. Design and regularization of neural networks: the optimal use of a validation set. In *Neural Networks for Signal Processing [1996] VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop*, pages 62–71. IEEE, 1996.

Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

Yann LeCun. A Theoretical Framework for Back-Propagation. In Geoffrey Hinton and Terrence Sejnowski, editors, *Proc. of the 1988 Connectionist models summer school*, pages 21–28. Morgan Kaufmann, 1988.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Rui Leite and Pavel Brazdil. Predicting relative performance of classifiers from samples. In *Proceedings of the 22nd international conference on Machine learning*, pages 497–503, 2005.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM, 2005.

Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11 (Feb):985–1042, 2010.

Da Li, Yongxin Yang, Yi-Zhe Song, and Timothy M Hospedales. Learning to generalize: Meta-learning for domain generalization. *AAAI*, 2018a.

Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, pages 6389–6399, 2018b.

Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Uncertainty in Artificial Intelligence*, pages 367–377, 2020.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017a.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018c.

Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. Meta-SGD: Learning to Learn Quickly for Few Shot Learning. *arXiv:1707.09835 [cs]*, July 2017b. URL `http://arxiv.org/abs/1707.09835`.

Renjie Liao, Yuwen Xiong, Ethan Fetaya, Lisa Zhang, KiJung Yoon, Xaq Pitkow, Raquel Urtasun, and Richard Zemel. Reviving and improving recurrent back-propagation. In *International Conference on Machine Learning*, pages 3088–3097, 2018.

David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *International Conference on Learning Representations*, 2019.

Pablo Ribalta Lorenzo, Jakub Nalepa, Michal Kawulok, Luciano Sanchez Ramos, and José Ranilla Pastor. Particle swarm optimization for hyper-parameter selection in deep neural networks. In *Proceedings of the genetic and evolutionary computation conference*, pages 481–488, 2017.

Jonathan Lorraine and David Duvenaud. Stochastic hyperparameter optimization through hypernetworks. *arXiv preprint arXiv:1802.09419*, 2018.

Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, pages 1540–1552, 2020.

Ilya Loshchilov and Frank Hutter. Cma-es for hyperparameter optimization of deep neural networks. *arXiv preprint arXiv:1604.07269*, 2016.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *International Conference on Learning Representations*, 2017.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *International Conference on Learning Representations*, 2019.

Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *International Conference on Learning Representations*, 2018.

Linyuan Lü and Tao Zhou. Link prediction in complex networks: A survey. *Physica A: statistical mechanics and its applications*, 390(6):1150–1170, 2011.

Jelena Luketina, Mathias Berglund, Klaus Greff, and Tapani Raiko. Scalable gradient-based tuning of continuous regularization hyperparameters. In *International Conference on Machine Learning*, pages 2952–2960, 2016.

Gang Luo. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 5(1):18, 2016.

Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in neural information processing systems*, pages 7816–7827, 2018.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning*, volume 30, page 3, 2013.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

Matthew MacKay, Paul Vicol, Jon Lorraine, David Duvenaud, and Roger Grosse. Self-tuning networks: Bilevel optimization of hyperparameters using structured best-response functions. *International Conference on Learning Representations*, 2019.

Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015a.

Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Early Stopping is Nonparametric Variational Inference. *arXiv preprint arXiv:1504.01344*, 2015b.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *International Conference on Learning Representations*, 2017.

James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In *International Conference on Machine Learning*, pages 1033–1040. Citeseer, 2011.

Andreas Maurer. Algorithmic stability and meta-learning. *Journal of Machine Learning Research*, 6 (Jun):967–994, 2005.

Andreas Maurer, Massimiliano Pontil, and Bernardino Romera-Paredes. The benefit of multitask representation learning. *The Journal of Machine Learning Research*, 17(1):2853–2884, 2016.

Rahul Mazumder, Trevor Hastie, and Robert Tibshirani. Spectral regularization algorithms for learning large incomplete matrices. *The Journal of Machine Learning Research*, 11:2287–2322, 2010.

Akshay Mehra and Jihun Hamm. Penalty method for inversion-free deep bilevel optimization. *arXiv preprint arXiv:1911.03432*, 2019.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pages 4556–4565, 2019.

Paul Micaelli and Amos Storkey. Non-greedy gradient-based hyperparameter optimization over long horizons. *arXiv preprint arXiv:2007.07869*, 2020.

John Miller and Moritz Hardt. Stable recurrent models. *International Conference on Learning Representations*, 2019.

M Minsky and S Papert. Perceptrons, 1969.

Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=B1DmUzWAW`.

Tom M. Mitchell. *Machine learning*. McGraw-hill New York, 1997.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Jonas Močkus. On Bayesian methods for seeking the extremum. In *Optimization techniques IFIP technical conference*, pages 400–404. Springer, 1975.

Jonas Močkus, Vytautas Tiesis, and Antanas Zilinskas. The application of Bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. Monte carlo gradient estimation in machine learning. *Journal of Machine Learning Research*, 2020.

Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proc. CVPR*, page 3, 2017.

Gregory Moore, Charles Bergeron, and Kristin P. Bennett. Model selection for primal SVM. *Machine Learning*, 85(1-2):175–208, October 2011. ISSN 0885-6125, 1573-0565. doi: 10.1007/s10994-011-5246-7. URL `http://link.springer.com/10.1007/s10994-011-5246-7`.

Pietro Morerio, Jacopo Cavazza, Riccardo Volpi, René Vidal, and Vittorio Murino. Curriculum dropout. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3544–3552, 2017.

Tsendsuren Munkhdalai and Hong Yu. Meta networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 2554–2563, 2017. URL `http://proceedings.mlr.press/v70/munkhdalai17a.html`.

Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

Eugene Ndiaye, Tam Le, Olivier Fercoq, Joseph Salmon, and Ichiro Takeuchi. Safe grid search with optimal complexity. In *International Conference on Machine Learning*, pages 4771–4780, 2019.

Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

Andrew Y. Ng. Preventing "overfitting" of cross-validation data. In *International Conference on Machine Learning*, volume 97, pages 245–253, 1997.

Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.

Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

Francesco Orabona and Dávid Pál. Coin betting and parameter-free online learning. In *Advances in Neural Information Processing Systems*, pages 577–585, 2016.

Boris Oreshkin, Pau Rodríguez López, and Alexandre Lacoste. Tadam: Task dependent adaptive metric for improved few-shot learning. In *Advances in Neural Information Processing Systems*, pages 721–731, 2018.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 2019.

Alberto Parmiggiani, Luca Fiorio, Alessandro Scalzo, Anand Vazhapilli Sureshbabu, Marco Randazzo, Marco Maggiali, Ugo Pattacini, Hagen Lehmann, Vadim Tikhanoff, Daniele Domenichelli, et al. The design and validation of the r1 personal humanoid. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 674–680. IEEE, 2017.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch, 2017.

Massimiliano Patacchiola, Jack Turner, Elliot J Crowley, Michael O'Boyle, and Amos Storkey. Bayesian meta-learning for the few-shot setting via deep kernels. 2020.

Barak A Pearlmutter. Fast exact multiplication by the hessian. *Neural computation*, 6(1):147–160, 1994.

Barak A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural networks*, 6(5):1212–1228, 1995.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Fabian Pedregosa. Hyperparameter optimization with approximate gradient. In *International Conference on Machine Learning*, pages 737–746, 2016.

Valerio Perrone, Rodolphe Jenatton, Matthias W Seeger, and Cédric Archambeau. Scalable hyperparameter transfer learning. In *Advances in Neural Information Processing Systems*, pages 6845–6855, 2018.

Bernhard Pfahringer, Hilan Bensusan, and Christophe G Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *International Conference on Machine Learning*, pages 743–750, 2000.

Fernando J Pineda. Generalization of back-propagation to recurrent neural networks. *Physical review letters*, 59(19):2229, 1987.

Boris Polyak. *Introduction to optimization*. Optimization Software, 1987a.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

Boris T. Polyak. *Introduction to Optimization*. Optimization Software Inc. Publication Division, New York, NY, USA, 1987b.

Siyuan Qiao, Chenxi Liu, Wei Shen, and Alan L Yuille. Few-shot image recognition by predicting parameters from activations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7229–7238, 2018.

J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

Aravind Rajeswaran, Chelsea Finn, Sham M Kakade, and Sergey Levine. Meta-learning with implicit gradients. In *Advances in Neural Information Processing Systems*, pages 113–124, 2019.

Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. *International Conference on Learning Representations*, 2017.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *International Conference on Machine Learning*, 2017.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

Esteban Real, Chen Liang, David R So, and Quoc V Le. Automl-zero: Evolving machine learning algorithms from scratch. *arXiv preprint arXiv:2003.03384*, 2020.

Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do CIFAR-10 classifiers generalize to CIFAR-10? *arXiv preprint arXiv:1806.00451*, 2018.

Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of Adam and beyond. *International Conference on Learning Representations*, 2018.

Mengye Ren, Wenyuan Zeng, Bin Yang, and Raquel Urtasun. Learning to reweight examples for robust deep learning. *International Conference on Machine Learning*, 2018.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Andrei A. Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. Meta-learning with latent embedding optimization. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=BJgklhAcK7`.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International Conference on Machine Learning*, pages 1842–1850, 2016. URL `http://proceedings.mlr.press/v48/santoro16.html`.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *International Conference on Machine Learning*, pages 343–351, 2013.

Nicolas Schilling, Martin Wistuba, Lucas Drumond, and Lars Schmidt-Thieme. Hyperparameter optimization with factorized multilayer perceptrons. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 87–103. Springer, 2015.

Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook.* PhD thesis, Technische Universität München, 1987.

Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial general intelligence*, pages 199–226. Springer, 2007.

Nicol N Schraudolph. Local gain adaptation in stochastic gradient descent. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99.(Conf. Publ. No. 470)*, volume 2, pages 569–574. IET, 1999.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3528–3536, 2015.

Gideon Schwarz et al. Estimating the dimension of a model. *The annals of statistics*, 6(2):461–464, 1978.

Hanie Sedghi, Vineet Gupta, and Philip M Long. The singular values of convolutional layers. *International Conference on Learning Representations*, 2019.

Matthias Seeger. Cross-validation optimization for large scale hierarchical classification kernel methods. In *Advances in neural information processing systems*, pages 1233–1240, 2007.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93, 2008.

Amirreza Shaban, Ching-An Cheng, Nathan Hatch, and Byron Boots. Truncated back-propagation for bilevel optimization. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1723–1732, 2019.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations*, 2015.

Kate A Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):1–25, 2009.

Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in neural information processing systems*, pages 4077–4087, 2017.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Mr Prabhat, and Ryan P. Adams. Scalable Bayesian Optimization Using Deep Neural Networks. In *International Conference on Machine Learning*, pages 2171–2180, 2015.

The Royal Society. Machine learning: the power and promise of computers that learn by example. `https://royalsociety.org/~/media/policy/projects/machine-learning/publications/machine-learning-report.pdf`, 2017.

Alessandro Sperduti and Antonina Starita. Speed up learning and network optimization with extended back propagation. *Neural networks*, 6(3):365–383, 1993.

Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian optimization with robust Bayesian neural networks. In *Advances in neural information processing systems*, pages 4134–4142, 2016.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Charles M Stein. Estimation of the mean of a multivariate normal distribution. *The annals of Statistics*, pages 1135–1151, 1981.

Mervyn Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974.

Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in neural information processing systems*, pages 2440–2448, 2015.

Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip HS Torr, and Timothy M Hospedales. Learning to compare: Relation network for few-shot learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1199–1208, 2018.

Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw Bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Corentin Tallec and Yann Ollivier. Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*, 2017.

Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

S. Thrun and L. Pratt. *Learning to learn*. Springer, 1998a.

Sebastian Thrun. Is learning the n-th thing any easier than learning the first? In *Advances in neural information processing systems*, pages 640–646, 1996.

Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 1998b. URL `https://books.google.it/books?hl=en&lr=&id=X_jpBwAAQBAJ&oi=fnd&pg=PA4&dq=theoretical+models+of+learning+to+learn&ots=gVMdZSEewl&sig=DtAvIIhSKPxl6uu9v12O8WqwlG8`. 00437.

Yonglong Tian, Yue Wang, Dilip Krishnan, Joshua B Tenenbaum, and Phillip Isola. Rethinking few-shot image classification: a good embedding is all you need? *arXiv preprint arXiv:2003.11539*, 2020.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

George Tucker, Andriy Mnih, Chris J Maddison, John Lawson, and Jascha Sohl-Dickstein. Rebar: Low-variance, unbiased gradient estimates for discrete latent variable models. In *Advances in Neural Information Processing Systems*, pages 2627–2636, 2017.

Joaquin Vanschoren. Meta-learning. In *Automated Machine Learning*, pages 35–61. Springer, Cham, 2019.

Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *International Conference on Learning Representations*, 2018.

Luis N Vicente and Paul H Calamai. Bilevel and multilevel programming: A bibliography review. *Journal of Global optimization*, 5(3):291–306, 1994.

Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial intelligence review*, 18(2):77–95, 2002.

Oriol Vinyals, Charles Blundell, Tim Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems*, pages 3630–3638, 2016.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*, pages 1058–1066, 2013.

Ruohan Wang, Yiannis Demiris, and Carlo Ciliberto. A structured prediction approach for conditional meta-learning. *arXiv preprint arXiv:2002.08799*, 2020.

Yan Wang, Wei-Lun Chao, Kilian Q Weinberger, and Laurens van der Maaten. Simpleshot: Revisiting nearest-neighbor classification for few-shot learning. *arXiv preprint arXiv:1911.04623*, 2019.

Zi Wang and Stefanie Jegelka. Max-value entropy search for efficient Bayesian optimization. *International Conference on Machine Learning*, 2017.

Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *nature*, 393 (6684):440, 1998.

Paul J Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Jason Weston, Frédéric Ratle, Hossein Mobahi, and Ronan Collobert. Deep learning via semi-supervised embedding. In *Neural Networks: Tricks of the Trade*, pages 639–655. Springer, 2012.

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Nando Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In *International Conference on Machine Learning*, pages 3751–3760, 2017.

Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Ronald J Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.

Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

Jian Wu, Matthias Poloczek, Andrew G Wilson, and Peter Frazier. Bayesian optimization with gradients. In *Advances in Neural Information Processing Systems*, pages 5267–5278, 2017.

Lijun Wu, Fei Tian, Yingce Xia, Yang Fan, Tao Qin, Lai Jian-Huang, and Tie-Yan Liu. Learning to teach with dynamic loss functions. In *Advances in Neural Information Processing Systems*, pages 6466–6477, 2018a.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger Grosse. Understanding short-horizon bias in stochastic meta-optimization. *International Conference on Learning Representations*, 2018b.

Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33nd International Conference on Machine Learning*, pages 40–48, 2016.

Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

Jiaxuan You, Rex Ying, Xiang Ren, William L Hamilton, and Jure Leskovec. Graphrnn: A deep generative model for graphs. *International Conference on Machine Learning*, 2018.

Dong Yu and Li Deng. *Automatic speech recognition*. Springer, 2016.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in neural information processing systems*, pages 3391–3401, 2017.

Muhan Zhang and Yixin Chen. Weisfeiler-Lehman neural machine for link prediction. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 575–583. ACM, 2017.

Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *arXiv preprint arXiv:1802.09691*, 2018.

Xun Zheng, Bryon Aragam, Pradeep Ravikumar, and Eric P Xing. Dags with no tears: Continuous optimization for structure learning. In *NeurIPS*, 2018.

Xiaojin Zhu, Zoubin Ghahramani, and John D Lafferty. Semi-supervised learning using Gaussian fields and harmonic functions. In *International Conference on Machine Learning*, pages 912–919, 2003.

Zhanxing Zhu, Jingfeng Wu, Bing Yu, Lei Wu, and Jinwen Ma. The anisotropic noise in stochastic gradient descent: Its behavior of escaping from sharp minima and regularization effects. *arXiv preprint arXiv:1803.00195*, 2018.

Luisa Zintgraf, Kyriacos Shiarli, Vitaly Kurin, Katja Hofmann, and Shimon Whiteson. Fast context adaptation via meta-learning. In *International Conference on Machine Learning*, pages 7693–7702, 2019.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *International Conference on Learning Representations*, 2017.

Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the royal statistical society: series B (statistical methodology)*, 67(2):301–320, 2005.