Modular Lifelong Machine Learning

Lazar Ignatov Valkov

Doctor of Philosophy

Institute for Adaptive and Neural Computation

School of Informatics

University of Edinburgh

2022

Abstract

Deep learning has drastically improved the state-of-the-art in many important fields, including computer vision and natural language processing (LeCun et al., 2015). However, it is expensive to train a deep neural network on a machine learning problem. The overall training cost further increases when one wants to solve additional problems. *Lifelong machine learning* (LML) develops algorithms that aim to efficiently learn to solve a sequence of problems, which become available one at a time. New problems are solved with less resources by transferring previously learned knowledge. At the same time, an LML algorithm needs to retain good performance on all encountered problems, thus avoiding catastrophic forgetting. Current approaches do not possess all the desired properties of an LML algorithm. First, they primarily focus on preventing catastrophic forgetting (Díaz-Rodríguez et al., 2018; Delange et al., 2021). As a result, they neglect some knowledge transfer properties. Furthermore, they assume that all problems in a sequence share the same input space. Finally, scaling these methods to a large sequence of problems remains a challenge.

Modular approaches to deep learning decompose a deep neural network into subnetworks, referred to as modules. Each module can then be trained to perform an atomic transformation, specialised in processing a distinct subset of inputs. This modular approach to storing knowledge makes it easy to only reuse the subset of modules which are useful for the task at hand.

This thesis introduces a line of research which demonstrates the merits of a modular approach to lifelong machine learning, and its ability to address the aforementioned shortcomings of other methods. Compared to previous work, we show that a modular approach can be used to achieve more LML properties than previously demonstrated. Furthermore, we develop tools which allow modular LML algorithms to scale in order to retain said properties on longer sequences of problems.

First, we introduce HOUDINI, a neurosymbolic framework for modular LML. HOUDINI represents modular deep neural networks as functional programs and accumulates a library of pre-trained modules over a sequence of problems. Given a new problem, we use program synthesis to select a suitable neural architecture, as well as a high-performing combination of pre-trained and new modules. We show that our approach has most of the properties desired from an LML algorithm. Notably, it can perform forward transfer, avoid negative transfer and prevent catastrophic forgetting, even across problems with disparate input domains and problems which require different neural architectures.

Second, we produce a modular LML algorithm which retains the properties of HOUDINI but can also scale to longer sequences of problems. To this end, we fix the choice of a neural architecture and introduce a probabilistic search framework, PICLE, for searching through different module combinations. To apply PICLE, we introduce two probabilistic models over neural modules which allows us to efficiently identify promising module combinations.

Third, we phrase the search over module combinations in modular LML as black-box optimisation, which allows one to make use of methods from the setting of hyper-parameter optimisation (HPO). We then develop a new HPO method which marries a multi-fidelity approach with model-based optimisation. We demonstrate that this leads to improvement in anytime performance in the HPO setting and discuss how this can in turn be used to augment modular LML methods.

Overall, this thesis identifies a number of important LML properties, which have not all been attained in past methods, and presents an LML algorithm which can achieve all of them, apart from backward transfer.

Lay Summary

Computers have been used to automate and augment many aspects of our life. Traditionally, people would write programs, which instruct the computer on how to perform a task. In contrast, machine learning allows the computer to learn from experience. For instance, it can learn from examples that show how a task should be performed. A computer usually learns how to perform each task from scratch, which can require considerable resources and a large number of examples. Instead, *lifelong machine learning* methods allow computers to build up knowledge over time and reuse it when learning how to perform a new task. This allows new tasks to be learned more quickly, requiring less resources.

This thesis presents a new method for lifelong machine learning. We introduce a more structured approach to storing and reusing the accumulated knowledge. We then improve our method by further reducing the amount of resources it requires.

Overall, our work makes three contributions. First, we reduce the number of examples a computer needs. Second, we extend the number of problems which our approach can learn to solve. Third, we make it easier to understand how the knowledge is being reused.

Acknowledgements

I would like to thank my supervisor, Dr. Charles Sutton, for giving me the opportunity to learn from him as well as for his valuable advice. I am grateful to my friend Akash SrivastaZa for all the time we have spent discussing exciting research ideas. I also thank my other collaborators: Cédric Archambeau, Dipak Chaudhari, Fela Winkelmolen, Rodolphe Jenatton and Swarat Chaudhuri.

I am indebted to my family, Maria, Ignat and Pepi, for their support throughout my studies. Finally, I thank Stefanie Speichert for her support as well as for her detailed feedback on my thesis.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Lazar Ignatov Valkov*)

To my grandmother, Penka Valkova.

Table of Contents

1 Introduction 1

1.1 Machine Learning - Benefits and Costs . . . 1

1.2 Lifelong Machine Learning - Reducing the Costs . . . 4

1.2.1 Knowledge Transfer to a Single Problem . . . 4

1.2.2 Knowledge Transfer Between Multiple Tasks . . . 5

1.2.3 Lifelong Knowledge Accumulation and Transfer . . . 6

1.3 Neurosymbolic Systems . . . 8

1.4 Thesis Overview . . . 8

1.5 Thesis Outline . . . 11

2 Background: Lifelong Learning 15

2.1 Supervised Learning . . . 15

2.2 Deep Learning . . . 18

2.2.1 Motivation . . . 18

2.2.2 Overview . . . 19

2.3 Transfer Learning . . . 21

2.3.1 Deep Transfer Learning . . . 24

2.4 Multi-Task Learning . . . 27

2.5 Lifelong Learning . . . 29

2.5.1 Common Experiments and Measurements . . . 33

2.5.2 Approaches . . . 35

3 Background - Related Areas 45

3.1 Modular Deep Learning . . . 45

3.1.1 Mixture of Experts . . . 46

3.1.2 Routing Networks . . . 47

3.1.3 Modular Deep Learning for Visual Question Answering . . . 48

3.2 Neurosymbolic Methods 50
3.3 Neural Programming 52
3.4 Neural Architecture Search 53
3.4.1 Search Space 54
3.4.2 Search Strategy 55
3.4.3 Performance Estimation Strategy 56
3.5 Black-box Optimisation 57
3.5.1 Sequential Model-based Optimisaition 57
3.5.2 Bayesian Optimisation 58

4 HOUDINI: Lifelong Learning as Program Synthesis 61
4.1 Related Work 88
4.2 Discussion 91

5 PICLE: A Probabilistic Framework for Modular Lifelong Learning 93
5.1 Introduction 94
5.2 Background 96
5.3 PICLE: A Probabilistic Search Framework 100
5.4 Scalable Perceptual and Few-shot Transfer 101
5.4.1 Probabilistic model 102
5.4.2 Search Strategy 107
5.5 Scalable Non-Perceptual Transfer 107
5.5.1 Probabilistic Model 108
5.5.2 Search Strategy 109
5.6 Combining The Two Search Strategies 110
5.7 BELL: Benchmark suite for Lifelong Learning 111
5.7.1 Compositional Problems 114
5.7.2 Realising the sequences 116
5.8 Experiments 117
5.8.1 BELL benchmarks 119
5.8.2 CTrL benchmarks 124
5.8.3 Ablation Experiments 126
5.8.4 Discussion 131
5.9 Related Work 132
5.10 Conclusion 136

6 A Transfer-Learning Extension of Hyperband **137**
6.1 Introduction 138
6.1.1 Black-Box Optimisation for Deep Learning 138
6.1.2 Hyperparameter Optimisation 140
6.2 Background 142
6.2.1 Adaptive Bayesian Linear Regression (ABLR) 144
6.2.2 Hyperband 146
6.3 Augmenting Hyperband with ABLR 148
6.4 Related Work on Hyperparameter Optimization 152
6.5 Experimental Evaluation 156
6.5.1 Tuning Stochastic Gradient Descent 158
6.5.2 Tuning XGBoost binary classifiers 159
6.5.3 Predictive Uncertainty with ABLR 162
6.6 HB-ABLR for Modular Lifelong Learning 163
6.7 Conclusion 165

7 Conclusion **167**
7.1 Future Work 168

Bibliography **171**

Chapter 1

Introduction

1.1 Machine Learning - Benefits and Costs

Machine Learning is the study of computer algorithms that improve automatically through experience (Mitchell, 1997). This experience is acquired by processing task-specific data. Of particular interest to this thesis are machine learning algorithms which define a parametric model of the data. These algorithms learn by selecting suitable values for the parameters, referred to as "training the model". A *machine learning problem* can roughly be defined by said task as well as by its input distribution. It is possible for two different problems to have the same task. For example, classifying an animal from an image or from a textual description are two separate machine learning problems with the same task.

A solution to a machine learning problem has two important considerations: generalisation performance, which reflects on how well it solves the target problem, and resources required. In general, there are three types of resources used in the development of a machine learning solution: computational resources, storage space and expert effort. An expert can be a machine learning expert, who understands the chosen machine learning algorithm, or a domain expert, who understands the target problem. Usually, computation and storage are more readily accessible, while experts are harder to access and expert effort is more costly in terms of finances and time required to complete a task. To further understand the benefits and the cost of developing a machine learning solution, one can consider *supervised learning*, which is one of the most commonly used types of machine learning.

Supervised learning is suitable for automating tasks, where one needs to make a prediction, given an input. For example, determining what object is displayed in an image, or predicting the price of a house, given some of its characteristics. To automate such a task using supervised learning, an expert needs to first create a training dataset, consisting of examples of inputs and their corresponding correct predictions (*labels*). Such a dataset is usually created by acquiring training inputs and manually labelling them. A supervised learning algorithm learns how to perform a task using the training dataset. It automatically selects a solution (*hypothesis*), consistent with the training dataset, from an expert-defined set of potential solutions (*hypothesis space*). For example, for a machine learning algorithm, based on training a parametric model, the set of all possible value assignments for the parameters constitutes the hypothesis space. Each value assignment is a hypothesis. Once a suitable hypothesis is found, it is used for predicting the labels of new inputs. Supervised learning has been applied to solve numerous problems, including spam filtering, online advertising and assessing loan applications.

Even though machine learning can and has been used to automate important tasks, using machine learning algorithms has multiple costs associated with it. A part of these costs increase with the size of the training dataset. This includes storage, computation and expert costs. Storage is needed in order to store the training dataset and the selected hypothesis. Computational resources are used to search for a well-performing hypothesis by processing the training dataset. Domain experts are needed in order to label example inputs when creating a training dataset for supervised learning. Furthermore, machine learning experts are necessary to select the hyperparameters of a machine learning algorithm. This is usually an iterative process that also depends on the size of the training dataset. Since all these costs increase with the size of the training dataset, one might consider simply reducing it. However, this is problematic for the following reason: if the hypothesis space is big and the training dataset is small, there can be multiple hypotheses which are consistent with the training dataset, but do not correctly predict the labels of new previously unobserved inputs (do not *generalise*). To address this, one needs to reduce the number of hypotheses that are consistent with the training dataset. This can be achieved by increasing the size of the training dataset. However, this in turn increases the aforementioned costs. Alternatively, one can reduce the hypothesis space or express a preference for some hypotheses over others. This is known as introducing *inductive bias* and can be used to express assumptions about the target solution (Battaglia et al., 2018). One way to introduce inductive bias is to have an expert analyse the target task, which could dramatically increase the expert cost and might still prove ineffective. As I'll discuss in the next section, there is another way to introduce inductive bias.

Most machine learning algorithms are limited in their ability to process raw natural data, such as images (LeCun et al., 2015). For these algorithms, an expert has to write a solution which pre-processes the data and transforms it into a lower-dimensional representation. This is referred to as *feature engineering* and requires an expert to invest a considerable amount of time in studying the input domain and the target task.

Deep learning refers to a subset of machine learning which aims to alleviate the need for feature engineering. The field trains parametric models, referred to as deep neural networks (DNNs), which stack learned nonlinear transformations of the data. This allows DNNs to learn a more abstract representation of the inputs from data. However, this also increases the hypothesis space and as a result, DNNs require large amounts of data to train. This can be seen as replacing expert cost spent on feature engineering with the costs associated with increasing the training dataset. However, note that the expert cost can be significantly higher than the cost of obtaining more training data. For instance, while it can be costly to pay an expert to develop a relevant image processing algorithm, it can be cheaper (in terms of both time and money) to use a tool such as Amazon Mechanical Turk where many people can be asked to label a few images each. Compared to other approaches, deep learning has demonstrated superior performance in multiple settings and has allowed machine learning to be used to automate a wider range of tasks (LeCun et al., 2015). Among other fields, deep learning has greatly advanced image processing (Jiao and Zhao, 2019), autonomous driving (Badue et al., 2020) and natural language processing (Deng and Liu, 2018). However, successful applications require a large number of parameters and a large training dataset, which in turn increases the computational, storage and expert costs associated with automating a task.

Typically, DNNs rely on all of their parameters for processing inputs and making predictions. Instead, modular deep learning (Chen, 2015) approaches deconstruct a DNN into a set of modules, in which each module is a parameterised non-linear function. This approach has been found useful for conditional computation (Bengio et al., 2013) where an input is processed by an input-specific composition of modules. It allows modules to specialise in performing a distinct atomic transformation which is useful for processing only some of the inputs. As a result, only a subset of the modules need to be used for a given input, leading to the reduction of the computational cost of a forward pass. Moreover, by composing the modules in novel ways a model can achieve better generalisation on the same problem (Andreas et al., 2016) and even solve related but previously unseen problems (Chang et al., 2018).

1.2 Lifelong Machine Learning - Reducing the Costs

Solving difficult problems with machine learning usually requires considerable resources. Deep learning allows one to reduce the expert time spent on feature engineering in exchange for requiring more data. In turn, increasing the training dataset increases the associated computational, storage and expert costs. As discussed above, it is possible to reduce the amount of training data required by introducing inductive bias into the learning algorithm. However, having an expert introduce such bias would significantly increase the expert cost. Alternatively, a machine learning algorithm can learn inductive bias from a related problem.

1.2.1 Knowledge Transfer to a Single Problem

Imagine having two related machine learning problems, S and T, which have similar input distributions or have similar tasks. It is then possible that their solutions share similarities. For example, the two problems can be spam detection and sentiment analysis of online user reviews. The solutions to both might involve the same latent representation of English sentences. Suppose further that a machine learning algorithm has been trained on problem S which resulted in a generalisable hypothesis h_S. One can then use this hypothesis to bias the machine learning algorithm when selecting a hypothesis h_T to solve problem T. For parametric models, order over the hypothesis space can be imposed by forcing the values of the parameters of h_T to be close to the values of the parameters of h_S, in terms of the Euclidean distance between them. Moreover, the size of the hypothesis space can be reduced by fixing some of the parameters of h_T to have values from h_S. This reduces the number of values that need to be learned and, thus, the number of possible hypotheses. This way of using a previously selected hypothesis to introduce inductive bias on a new problem can be seen as *knowledge transfer* and has been used in the field of *transfer learning*. There are other related fields such as meta-learning (Hospedales et al., 2021), self-supervised learning (Liu et al., 2021) and semi-supervised learning (Van Engelen and Hoos, 2020). They differ from transfer learning by the type of knowledge transferred or by the assumptions made about the involved problems.

In deep learning, one can differentiate between *perceptual transfer* and *non-perceptual* *transfer*. Perceptual transfer occurs when the transferred knowledge describes how to process the input domain. For instance, transferring the knowledge of processing English sentences between the problems of spam detection and sentiment analysis of English online user reviews. Non-perceptual transfer can occur between problems with disparate input domains but similar tasks. In this case, the transferred knowledge can describe how to predict the label, given a latent embedding of the input. For example, transferring the knowledge of performing sentiment analysis between two problems which involve online reviews of two different languages (Kanclerz et al., 2020).

Transfer learning has been successfully applied to multiple domains to significantly reduce the cost of solving a new problem. For instance, Devlin et al. (2018) use a deep learning model with 340 million parameters which is trained on the source problem for 4 days using 16 TPUs. Afterwards, they use transfer learning to solve a related problem with a small datatest using at most 1 hour for training and 1 TPU. This way, the authors achieved state-of-the-art on multiple natural language processing target problems.

1.2.2 Knowledge Transfer Between Multiple Tasks

Transfer learning focuses on performing well on a single problem - the target problem. However, one could be interested in automating multiple related tasks. For example, a social media service may want to automate sentiment analysis, the detection of hate speech and spam detection. Instead of tackling these problems individually, a machine learning algorithm can learn to simultaneously solve them. Knowledge transfer could then occur between the problems, thus, providing an inductive bias and selecting a more generalisable hypothesis (Caruana, 1997). This approach is studied by the field of *multi-task learning*, which develops algorithms that can decrease the training data required per task. In effect, this can reduce the combined cost, compared to training separate solutions for each problem. Multi-task learning has been applied to multiple areas, including computer vision, health informatics, speech and natural language processing (Zhang and Yang, 2017). However, one current limitation is that most multi-task learning methods assume that the input spaces of all problems are the same and that the input distributions are similar (Yang and Hospedales, 2014; Zhang and Yang, 2021).

1.2.3 Lifelong Knowledge Accumulation and Transfer

Both transfer learning and multi-task learning are concerned with solving a preset number of problems. However, there is a need for a machine learning algorithm which can continuously learn to solve new problems. This would allow each new problem to benefit from knowledge transfer from previously learned problems. For example, remember the social media service which has automated sentiment analysis, the detection of hate speech and spam detection. Given a pandemic caused by a virus, said service might want to also automate the problem of detecting misinformation about said virus. In such a time-critical setting with training data being scarce, it is necessary to transfer knowledge from the previously solved natural language processing problems.

Such settings are addressed by the field of *Lifelong Machine Learning* (LML) (or *Lifelong Learning*) (Silver et al., 2013). LML aims to build systems that can learn many tasks over a lifetime from one or more domains. Its goal is to retain learned knowledge and to selectively transfer that knowledge when learning a new task, in order to develop more accurate hypotheses. LML aims to significantly reduce the cost of learning to solve a new problem, similarly to transfer learning. As an LML method learns to solve more problems, it should acquire more knowledge which can be transferred, thus, further reducing the cost of learning to solve future problems. Moreover, the knowledge acquired from solving a new problem could be employed to improve the model's performance on a previous problem. LML can also be used to reduce the storage space required, compared to storing separate solutions for each problem of interest. In addition to reducing the costs, LML is necessary for developing agents that can efficiently accumulate knowledge in a continually changing environment. For example, LML appears to be necessary for developing more useful physical robots or chatbots that interact and assist humans in their daily lives (Chen and Liu, 2018). LML can also be applied to medical diagnosis, where more diseases need to be recognised as patient data becomes available (Li et al., 2020).

One could consider applying a multi-task learning method to the LML setting. This is problematic since such methods typically assume that the data for all problems of interest is available and that their model is trained offline over all of this data. However, the training datasets of past problems could become unavailable after training. One possible reason is the high storage cost, associated with storing all previous data, as the number of problems increases. Another possible reason is that storing past datasets would violate privacy-related restrictions. Even if previous data was available, re-training on all previous datasets every time a new problem is encountered could become prohibitively computationally expensive as the algorithm is applied to a long sequence of problems. To address these shortcomings, a subset of LML methods (rehearsal-based) adapt ideas from multi-task learning by relaxing the aforementioned constraints.

This thesis identifies a list of properties which an LML algorithm should have. Our list differs from the desiderata defined in related work (Schwarz et al., 2018b; Hadsell et al., 2020; Veniat et al., 2020; Delange et al., 2021) primarily by distinguishing between different capabilities of performing knowledge transfer to new problems. First, *plasticity* refers to an algorithm's ability to continuously learn to solve new problems. Ideally, the algorithm's generalisation performance on each problem should be at least as good as when said problem is solved in isolation. Transferring irrelevant knowledge on a new problem can introduce harmful inductive bias which hinders the algorithm's performance. This is referred to as *negative transfer* and should be avoided in order to improve plasticity. Second, *stability* refers to the ability to retain previously acquired knowledge. It is required in order to prevent *catastrophic forgetting* which can occur when an algorithm's performance on a past problem decreases drastically. Third, *forward transfer* refers to an algorithm's ability to transfer knowledge to a newly encountered problem. We identify three types of forward transfer: between problems with similar input distributions (*perceptual*), between problems with disparate input distributions or different input spaces (*non-perceptual*) and to problems with a few training examples (*few-shot*). For a diverse sequence of problems, LML algorithms which use DNNs should be able to transfer knowledge across similar problems but different neural architectures. For example, this is necessary in order to perform transfer across different input spaces. Fourth, an LML algorithm should be capable of *backward transfer*, allowing it to improve its performance of previously encountered problems after solving new ones. Finally, an LML algorithm should be applicable to large sequences of problems, with its resource demands scaling sub-linearly with the number of solved problems (*scalability*).

Current deep learning approaches to LML do not possess all of the aforementioned properties. Most focus on limiting catastrophic forgetting, while allowing for perceptual transfer (Díaz-Rodríguez et al., 2018; Delange et al., 2021). As a result, non-perceptual transfer, negative transfer and backward transfer are largely unaddressed. Moreover, current approaches use the same or a set of very similar neural architectures for each problem. This makes them unsuitable for solving a diverse set of problems, which can require different neural architectures. Finally, current approaches do not scale to large problem sequences.

1.3 Neurosymbolic Systems

While deep learning has been used to advance multiple fields (LeCun et al., 2015), it has a number of shortcomings (Garcez and Lamb, 2020). For instance, these methods require a large number of data points. Furthermore, it is difficult to interpret the predictions of a deep model. Another shortcoming is that deep models do not generalise well to inputs that are sampled from a different distribution than the one used for training (Nagarajan et al., 2020).

Symbolic methods are another approach to task automation. They involve manipulating human-readable symbols using pre-defined rules (Garnelo and Shanahan, 2019). These methods are data-efficient, interpretable and can generalise to new input distributions. However, symbolic methods operate on handcrafted symbols, which are not learned from sensory input (Harnad, 1990).

Neurosymbolic approaches combine deep learning and symbolic methods in order to take advantage of their complementary strengths (Garnelo and Shanahan, 2019; Garcez and Lamb, 2020). Among other potential benefits, this allows experts to introduce further structure into a deep learning system. For instance, Silver et al. (2016) combine learned heuristics, implemented by neural networks, with a Monte Carlo tree search.

1.4 Thesis Overview

This thesis presents a number of advancements towards developing a deep lifelong machine learning algorithm which can achieve all of the desiderata outlined above. We adopt a modular approach as it makes it possible the decompose the accumulated knowledge into disjoint groups of parameters which can then be flexibly reused when necessary.

Chapter 4 introduces HOUDINI, a neurosymbolic framework for supervised learning. We describe modular deep neural networks (mDNNs) as typed functional programs. The functions are parameterised and implemented as stacks of hidden layers, referred to as neural modules. Furthermore, the functions can have different types, which indicates that their inputs and outputs can be of different dimensionality and contain different sets of values. The functions can be composed in different ways to form different functional programs. Therefore, a functional program can describe an mDNN in terms of its neural architecture as well as its selection of neural modules. Given a new problem, we search for the program with the best generalisation performance. This process evaluates multiple programs, which involves training their randomly initialised parameters. We cast solving a new problem as program synthesis and employ a symbolic program synthesizer, which performs an exhaustive search over type-compatible programs. The program with the best generalisation performance is used to solve the given problem.

This modular approach is readily applicable to the LML setting. After finding the optimal program for a given problem, its newly trained neural modules are frozen and added to a library of functions. These functions can be reused on a new problem, which allows knowledge transfer. As more problems are solved, HOUDINI accumulates more transferable knowledge, stored as pre-trained modules in its library. As a result, it attains most of the properties, required from an LML algorithm. Our framework operates on problems from different input spaces. Furthermore, we demonstrate that HOUDINI achieves perceptual transfer, non-perceptual transfer, few-shot transfer, avoids negative transfer and prevents catastrophic forgetting on sequences of disparate problems. Moreover, it is capable of few-shot learning.

However, HOUDINI searches through the set of all possible combinations of pre-trained and new modules, for each of the possible neural architectures. At the same time, evaluating each item in this set is expensive as it involves training the new parameters of an mDNN. As a result, HOUDINI's main shortcoming is that its type-guided exhaustive search does not scale to large search spaces. This prevents it from being applicable to problems which require large modular neural architectures or to long sequences of problems, as both settings lead to enormous search spaces.

Chapter 5 introduces a scalable modular LML algorithm which maintains the LML properties of HOUDINI by also accumulating a library of pre-trained modules and reusing them on new problems as necessary. In contrast, the algorithm assumes that a problem-specific neural architecture is provided by an expert. As a result, for a new problem it searches through the set of all possible combinations of pre-trained and new modules. This search space is still rapidly expanding making a naive approach to search inapplicable. We address this challenge by introducing a probabilistic search framework called PICLE. PICLE divides the search space into subsets of similar module combinations and defines subset-specific probabilistic models which specify a distribution over the choice of pre-trained modules. PICLE then searches over the whole search space by using its probabilistic models to efficiently prioritise module combinations that have a high probability of success. We identify two large subsets of promising module combinations, which enable perceptual, few-shot and non-perceptual transfer, and define two distinct probabilistic models over them. We then use these models within PICLE and show that the resulting search method leads to a scalable LML algorithm which fulfils all of the outlined LML desiderata, apart from backward transfer. To verify these claims, we introduce a new benchmark suite for evaluating the properties of an LML algorithm. Our extensive experiments demonstrate that our method can outperform competitive baselines.

The modular LML algorithm which we present in Chapter 5 considers two large subsets of module compositions. However, this leaves another big subset, which contains the rest of the possible module compositions, unexplored. The items in this subset can enable useful knowledge transfer properties, such as simultaneously performing perceptual and non-perceptual transfer. Therefore, there is a need for an efficient search strategy that can be used to explore this subset. To address this, Chapter 6 phrases the discrete search over module combinations in a modular LML algorithm as black-box optimisation, which makes it possible for ideas from hyperparameter optimisation (HPO) to be applied to LML. However, we identify 3 shortcomings of previous HPO methods which make it difficult for them to be directly applied to modular LML. These shortcomings include the necessity for a manually-designed special input featurisation, the cold-start problem and the inefficiency incurred by evaluating every considered element using the same constant number of resources. In response to this, we develop a new method, HB-ABLR, which addresses all of these challenges. We arrive at it by augmenting a multi-fidelity approach to HPO (Li and Hoiem, 2017) with model-based adaptive sampling which uses a neural network (Perrone et al., 2018). Our choice of a surrogate model allows us to take advantage of data from similar optimisation problems and learn a suitable transformation of the input. Overall, we demonstrate that this allows HB-ABLR to achieve superior anytime performance in the setting of hyperparameter optimisation, compared to competitive benchmarks. This means that our method is capable of finding solutions which lead to better performance using a limited number of computational resources. Finally, we provide a discussion on how HB-ABLR can be applied to augment modular LML algorithms in order to enhance their transfer learning properties when applied to long sequences of problems.

Overall, this thesis presents a modular LML algorithm which can achieve all of the important LML properties which we had identified, apart from backward transfer. Furthermore, we demonstrate that a neurosymbolic approach can be used to automate lifelong machine learning across problems which have structured inputs, such as lists and graphs, and problems which require different neural architectures.

This thesis also paves the way for numerous research directions. Currently, we have demonstrated that HOUDINI can achieve most of the properties expected from an LML algorithm. An open question is whether we can augment our framework to allow backward transfer, thus, creating the first method with all of the desired properties. Moreover, we allowed HOUDINI to scale by fixing its choice of neural architecture. Therefore, allowing it to efficiently search through both neural architectures and module combinations is an exciting future research direction.

1.5 Thesis Outline

The rest of this thesis has the following structure:

– **Chapter 2** provides the background necessary to understand current LML methods. This includes supervised machine learning, deep learning, transfer learning, multi-task learning and lifelong learning.

– **Chapter 3** provides the background for all other fields, which are relevant to the presented research. This includes modular deep learning, neurosymbolic methods, neural programming, neural architecture search and Bayesian optimisation.

– **Chapter 4** introduces HOUDINI- a neurosymbolic framework, suitable for lifelong learning. Solving a new problem is posed as program synthesis. A type-directed exhaustive search is used to find the optimal program. We demonstrate that this approach contains most of the properties that an LML algorithm should have.

  The content of this chapter is based on a collaborative project. I led the project and was responsible for identifying relevant LML properties, helping design the modular approach to be applicable to LML, designing the experiments and implementing everything except for the program synthesis algorithm and the evolutionary search baseline.

  Our work was published as the following.

Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton and Swarat Chaudhuri. HOUDINI: Lifelong Learning as Program Synthesis.
In *International Conference on Neural Information Processing Systems*, 2018

– **Chapter 5**, to the best of my knowledge, introduces the first scalable modular LML algorithm which can achieve perceptual, non-perceptual and few-shot transfer. The algorithm achieves these properties by searching through a set of module combinations. We introduce PICLE, a probabilistic search framework which uses probabilistic models over pre-trained modules in order to efficiently explore the rapidly expanding search space. We define two probabilistic models over two disjoint subsets of module combinations for which we, respectively, introduce an efficient way of approximating a pre-trained module's input distribution, and a kernel between module combinations.

  Moreover, we list our lifelong learning desiderata and provide a discussion on how it differs from the lists of properties found in previous work. Finally, we introduce a new benchmark suite which can be used to diagnose the identified LML properties.

  I am mainly responsible for the content of this chapter. The following collaborators were involved in preliminary discussions, provided feedback on different iterations of the work, and provided assistance with the presentation of the method: Akash Srivastava, Dipak Chaudhari, Swarat Chaudhuri and Charles Sutton.

– **Chapter 6** presents HB-ABLR, a new hyperparameter optimisation method which augments Hyperband (Li and Hoiem, 2017) by replacing its random sampling procedure with model-based adaptive sampling using ABLR (Perrone et al., 2018). This leads to a multi-fidelity approach with a scalable surrogate model which can transfer knowledge from similar optimisation problems. We present experimental evidence that our method achieves better anytime performance than previous state-of-the-art approaches, in the setting of hyperparameter optimisation. Finally, we argue that modular LML algorithms can be phrased as black-box optimisation and provide a discussion on how HB-ABLR can benefit current modular LML approaches.

  The content of this chapter was based on a collaborative effort. As the first author, I was responsible for investigating how to best combine Hyperband and

ABLR and identifying the strengths and shortcomings of the resulting approach. I was also in charge of designing and implementing the experimental setup. I held frequent discussions with Rodolphe Jenatton who used my code to produce the final experimental results, after my internship had ended.

The work in this chapter was presented as the following.
Lazar Valkov, Rodolphe Jenatton, Fela Winkelmolen, Cédric Archambeau. A simple transfer-learning extension of Hyperband.
In *NeurIPS Workshop on Meta-Learning*, 2018

– **Chapter 7** presents a summary of the thesis and proposes future directions.

Chapter 2

Background: Lifelong Learning

This chapter describes the current work in lifelong learning and related areas. It aims to provide the necessary background knowledge to understand the desired properties of a lifelong learning algorithm, as well as the shortcomings of current approaches. The focus of this thesis is on approaches which make use of deep learning, as motivated in the introduction. The chapter begins by outlining supervised learning in order to establish terminology and notation. Afterwards, I describe deep learning and list the properties that make it suitable for lifelong learning. Next, I describe and review work in transfer learning and multi-task learning, as these are closely related areas. Thereafter, lifelong learning is defined and an overview of the literature is presented.

2.1 Supervised Learning

Supervised learning algorithms learn from a labelled *training dataset* - a set of input-output pairs. For instance, if one needs to automate the classification of an object, portrayed in an image, one would gather different images and manually assign each of them with a class. The resulting dataset is used by a supervised learning algorithm to yield a hypothesis, which is able to predict the labels of unseen images. The amount of inferred labels that match the true labels on new inputs characterise the predictor's *generalisation performance*. A higher number of training images usually result in better generalisation performance. The labels can be discrete, as in the example of object classification, which leads to a classification problem. Alternatively, one might want to predict a continuous quantity, such as the price of a house, which leads to a regression problem. The rest of this section provides a more precise definition of supervised learning.

A **supervised learning problem** requires an algorithm $\mathcal{A}$ to learn how to make a prediction $\hat{y}$ about the label y of a given input $\mathbf{x}$. The problem is specified by an *input domain* $\mathcal{D}$ and a *task* $\mathcal{T}$. The input domain $\mathcal{D} = (\mathcal{X}, p(X))$ is characterised by a feature space $\mathcal{X}$ and a marginal probability distribution $p(X)$ where $X \in \mathcal{X}$. The labels are defined by the given task $\mathcal{T}$, specified by a label space $\mathcal{Y}$ and a predictive function $f^* : \mathcal{X} \to \mathcal{Y}$, denoted $\mathcal{T} = (\mathcal{Y}, f^*(\cdot))$. While f^* is typically unknown it is assumed to be the underlying function, used to assign a label to each input: $y = f^*(\mathbf{x})$.

Definition 1. (Supervised Learning) Let $D = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^N$ be a training dataset of N examples where the inputs $\{\mathbf{x}^{(i)}\}_{i=1}^N$ are distributed according to some distribution $p(X)$. Let each label $y^{(i)}$ be generated by an unknown function $f^* : \mathcal{X} \to \mathcal{Y}$, so that $y^{(i)} = f^*(\mathbf{x}^{(i)})$. Given the training dataset D, discover a function $\hat{h} : \mathcal{X} \to \mathcal{Y}$ that approximates the true function f^*. (Russell and Norvig, 2009)

It is usually assumed that the inputs are independent and identically distributed according to $p(X)$. The function $\hat{h}$ is a *hypothesis*, which is selected from a *hypothesis space* $\mathcal{H}$ - the set of all functions being considered. For example, the hypothesis space can consist of all linear combinations of the input's dimensions, where each linear combination is a hypothesis. As stated above, the goal of the chosen hypothesis is to approximate the true function f^*. This would allow it to successfully predict unseen inputs, sampled from the same distribution $p(X)$ used to sample the inputs in the training dataset. The difference between the predicted and actual label is measured by a loss function $\mathcal{L}$. The expected loss is called *risk*. Ideally, the algorithm $\mathcal{A}$ should find the hypothesis $\hat{h}^*$ which achieves the lowest risk $\mathcal{R}(h)$:

$$\hat{h}^* = \operatorname*{argmin}_{h \in \mathcal{H}} \mathcal{R}(h) = \operatorname*{argmin}_{h \in \mathcal{H}} \mathbb{E}_{\mathbf{x} \sim p(X)} [\mathcal{L}(h(\mathbf{x}), f^*(\mathbf{x}))] \quad .$$

However, as one only has access to N labelled data points, it is not possible to evaluate the risk. Instead, an algorithm $\mathcal{A}$ can use the training data to minimise the *empirical risk* $\hat{\mathcal{R}}(h)$:

$$\hat{h} = \mathcal{A}(\mathcal{D}) = \operatorname*{argmin}_{h \in \mathcal{H}} \hat{\mathcal{R}}(h) = \operatorname*{argmin}_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(h(\mathbf{x}^{(i)}), y^{(i)}) \quad .$$

Thus, supervised learning algorithms select a hypothesis with low loss on the training dataset (low *training loss*). This method of selection can be problematic, as it can choose a hypothesis which fits the training dataset but generalises poorly to unseen inputs. This problem is referred to as *overfitting* and occurs when the empirical risk of the selected hypothesis is much smaller than its true risk, i.e. when $\mathcal{R}(\hat{h}) - \hat{\mathcal{R}}(\hat{h})$ is large (Shalev-Shwartz and Ben-David, 2014).

To combat this problem, one can either increase the size of the training dataset or introduce *inductive bias* in the hypothesis selection process. Inductive bias is introduced either by restricting the hypothesis space or by imposing an order over the hypotheses. First, restricting the hypothesis space removes a subspace from consideration. An example of this is restricting the hypothesis space to contain only linear combinations of the input's dimensions. Second, one can introduce inductive bias by imposing an order on the hypotheses. This way, if multiple hypotheses have similar performance on the training dataset, the algorithm will use the imposed order to choose between them. A commonly imposed ordering follows a principle named Occam's Razor and gives priority to simpler hypotheses. In the linear combinations example, this could be implemented by giving preference to hypotheses with lower parameter values. An algorithm $\mathcal{A}$ can introduce such a bias by performing *structural risk minimisation* (SRM):

$$\hat{h} = \mathcal{A}(\mathcal{D}) = \operatorname*{argmin}_{h \in \mathcal{H}} \hat{\mathcal{R}}(h) + \beta J(h) = \operatorname*{argmin}_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(h(\mathbf{x}^{(i)}), y^{(i)}) + \beta J(h) \quad .$$

Here, the function $J(h)$ is called a regularisation function and it evaluates the complexity of a model, and β is a weighting constant. The idea of biasing hypotheses spaces is used in the next sections to explain transfer learning.

After training a model on a training dataset, it is useful to understand how well the model will generalise to unseen data. The model's *generalisation* can be measured by the risk of the selected hypothesis, $\mathcal{R}(\hat{h})$. This can be approximated by evaluating the model's performance on a number of held-out data points, referred to as the test dataset. The test dataset can be used to approximate the risk by calculating:

$$\mathcal{R}(\hat{h}) \approx \frac{1}{N^{\text{test}}} \sum_{i=1}^{N^{\text{test}}} \mathcal{L}(h(\mathbf{x}^{(i),\text{test}}), y^{(i),\text{test}}) \quad .$$

For a classifier, another common measure of the generalisation performance is the its average accuracy across the input distribution, which can be approximated using the test dataset.

2.2 Deep Learning

This section first motivates the need for deep learning. Afterwards, I describe core concepts and properties, which make deep neural networks suitable for lifelong learning.

2.2.1 Motivation

In order for machine learning algorithms to generalise to unseen inputs, they need to make assumptions about the data (Wolpert, 1996). One common assumption is that the underlying labelling function is smooth, i.e. $f^*(x) \approx f^*(x+\varepsilon)$ (Goodfellow et al., 2016). However, machine learning methods which only rely on this assumption are affected by the curse of dimensionality, requiring the size of the training dataset to grow exponentially with the input dimensionality. This problem can be addressed by making additional assumptions about the data. Such assumptions are often present inside parametric models, which are statistical models with a fixed number of parameters (Murphy, 2012).

Logistic regression is a parametric model, which makes predictions based on a linear combination of the input features. To allow this model to represent a nonlinear function, one can augment the input with nonlinear transformations of its features, referred to as *basis functions*. One issue with this approach is that the number of basis functions is required to grow, often exponentially, with the number of input dimensions, in order to obtain good performance (Bishop, 2006). Furthermore, there are many basis functions to choose from, e.g. spline functions, Fourier basis and wavelets. Instead, one can represent each input using a pre-determined number of parameterised basis functions, each of which consists of a linear combination of the input, followed by a nonlinear function (a *non-linearity*), e.g. tanh. Then the parameters of each basis function can be trained along with those of the logistic regression on the training dataset. The resulting model is an example of a neural network (NN), where the parameterised basis functions are called hidden units and a group of hidden units is referred to as a hidden layer. By learning a predetermined number of parameterised basis functions, NNs can attain the desired accuracy using fewer units of computation compared to the alternative of using fixed basis functions (Gnecco, 2012; Kawaguchi, 2016; Du et al., 2018).

It is possible to increase the number of hidden layers, which leads to a deep neural network (DNN). It has been empirically demonstrated that DNNs outperform NNs with a single layer in many settings of interest. For instance, deep learning has been successfully used to obtain state-of-the-art performance across many fields, including image processing (Jiao and Zhao, 2019), autonomous driving (Badue et al., 2020) and natural language processing (Deng and Liu, 2018). DNNs' success can be partially attributed to their inductive bias, expressivity and effect on the optimisation process (Berner et al., 2021). First, by composing different types of layers (e.g. convolutional and fully connected), the model can be biased towards compositional hypotheses which exploit pre-known properties of the data. Second, it has been shown that there are functions which can be efficiently represented by a DNN with L layers, but would require an exponentially higher number of hidden units for a smaller number of layers (Gühring et al., 2020). Third, there are results which suggest that deep models have properties which facilitate non-convex optimization, which is used for selecting a hypothesis (Choromanska et al., 2015).

2.2.2 Overview

The field of *deep learning* (DL) studies deep neural networks. This includes understanding a DNN's properties, investigating new neural architectures, studying how to train DNNs and applying it to solve challenging problems. Advances in the field have led to breakthroughs in processing images, text, video, speech and audio (LeCun et al., 2015).

Deep neural networks (DNNs) are parametric models which represent a hypothesis as a composition of nonlinear functions, referred to as hidden layers. DNNs are characterised by their architecture, which describes the type of each hidden layer and how these hidden layers are connected. Common types of hidden layers are fully connected, convolutional and recurrent layers. The choice of hidden layers reflects further assumptions made about the data. For instance, convolutional layers apply the same parameterised function to all local regions of the input of a specified size. This makes convolutional layers suitable for applications like image processing since an object of interest can appear in any part of an image. Moreover, recurrent layers are used to process a sequence of items. Each item is processed with the same parameterised function, conditioned on information acquired from processing all of the previous items in the sequence. This makes recurrent layers suitable for applications such as natural language processing, where each word can be interpreted differently, depending on its context. A deep architecture which primarily relies on convolutional layers is called a convolutional neural network. An architecture that mainly relies on recurrent layers is referred to as a recurrent neural network. For further information about neural architectures and their implementation details, the reader is referred to Goodfellow et al. (2016).

At the time of writing, the most successful approach to train a DNN is by iteratively adjusting its parameters, using the gradient of the training loss with respect to the parameters. This involves repeating three steps: computing the loss, calculating the gradient and updating the parameters. First, in order to compute the loss, the DNN is used to process a different random subset of the training dataset. This can be used to provide an unbiased estimate of the gradient. Second, the gradient is computed efficiently using *backpropagation* (Rumelhart et al., 1986). Backpropagation is an algorithm based on dynamic programming, which takes advantage of the compositional structure of a deep neural network in order to avoid redundant computations. Third, the parameters are updated by an *optimiser* which uses the gradient to adjust their values in order to reduce the loss. Stochastic gradient descent (SGD) is a basic optimiser which can be used, but there are extensions, e.g. Adam (Kingma and Ba, 2014). In the literature, SGD is sometimes used to refer to an SGD-based optimiser, which can be replaced by its extensions.

The compositional nature of deep neural networks makes them suitable for the purposes of transfer learning and thus for lifelong learning. This is because it has been shown that neural networks represent the input data with a level of abstraction that increases with the index of the hidden layer (Kozma et al., 2018). It is possible to gain insight into this by looking at the inputs which increase the activation value of a hidden unit. This can provide intuition on what features from the training dataset does a unit react to. For example, Zeiler and Fergus (2014) visualise the features learned by every layer of a CNN, trained on an image dataset. Their results provide an insight into what each layer may have learned to detect. It could be interpreted that the first layer detects edges and colour blobs. Moreover, the second layer detects different combinations of edges and colours, while the third layer learns to detect textures. The fourth detects parts of the objects of interest, e.g. a dog's face. Finally, the fifth layer is shown to detect whole objects, e.g. dogs, with significant variation in their pose. This tendency has been observed in different papers. For instance, Nguyen et al. (2016) demonstrate an abstract hidden unit in the later layers of a CNN, which activates whenever the image depicts a store. This includes images of different sections of the store and images of the storefront. Overall, this suggests that lower hidden layers compute task-independent features which process the input domain, while higher layers perform more abstract and domain-invariant computation, which is more task-specific. Therefore, one could reuse or repurpose either the lower or the higher layers of a DNN, depending on the similarity between problems.

2.3 Transfer Learning

Deep learning's success can be partly attributed to its highly expressive hypothesis space. However, deep models require large datasets to train in order to find a generalisable hypothesis. For instance, consider the convolutional neural network, proposed by Krizhevsky et al. (2012), which won the ImageNet large-scale visual recognition challenge in 2012 (Russakovsky et al., 2015). It has 60 million parameters, which were trained on 1.2 million high-resolution images from the ImageNet dataset (Russakovsky et al., 2015). The training was done on two GPUs and took 5 days. This illustrates that training a deep neural network has great computational and storage costs involved. Moreover, there is a high expert cost for obtaining a large training dataset because, for most problems, labelling inputs requires manual labour by domain experts. Therefore, it is necessary to be able to apply deep learning to smaller datasets, as this would significantly reduce the associated costs and make it more widely applicable.

As discussed in Section 1.1, when the number of training data points is low, one can improve the generalisability of a learning algorithm by introducing additional inductive bias. One way to do this is to study the data and inject expert knowledge by providing a different featurisation of the inputs, which is more suitable for the target task. This would alleviate the need for a deep model to learn this featurisation and thus the number of parameters of the model could be reduced, which in turn should reduce the number of training datapoints required. For example, for an image processing problem, one can try pre-processing the training images with an edge-detection algorithm. This type of investigation requires expert knowledge and is very time-consuming. Alternatively, one can introduce inductive bias into the learning algorithm by using data from a similar machine learning problem. Transfer learning (TL) constitutes one class of such approaches and is motivated by the insight that similar problems should have similar hypotheses. TL can be used to obtain better generalisation performance with a limited amount of data or a limited amount of computation.

It is worth noting that there are other sub-fields which can be used improve the generalisability of a machine learning algorithm when the number of available training data is low. Meta-learning (Hospedales et al., 2021) introduces a *meta-level objective* which is used to optimise some properties of the learning algorithm, which in turn can allow said algorithm to better generalise on new tasks. For instance, MAML (Finn et al., 2017) uses data from related tasks in order to learn suitable initial parameters for a deep neural network, so that for a new problem the resulting algorithm can achieve good generalisation performance using only a few gradient updates on a small number of data points. In contrast, TL methods typically do not involve a meta-level objective and differ in the type of knowledge being transferred. Self-supervised learning (Liu et al., 2021) can be used to train a model on a large amount of unlabelled data in order to extract knowledge that can be transferred to supervised learning problems. For instance, auto-regressive models are trained to sequentially predict each of the input's dimensions, based on the previously observed ones. Among other areas, these have led to significant improvements in performance in natural language processing (Radford et al., 2018, 2019). In contrast, in TL, the data which knowledge is transferred from is typically assumed to be labelled (Yang et al., 2020). Semi-supervised learning considers the setting where both a small labelled dataset and unlabelled data is available for a problem of interest (Van Engelen and Hoos, 2020). For example, self-training methods iteratively use a model fit on the labelled dataset for labelling inputs from the unlabelled dataset, after which the model is further updated using all labelled data points (Ouali et al., 2020). Semi-supervised learning makes the assumption that the data comes from the same distribution which is not made in TL. This section focuses on transfer learning since it is the most relevant to the work described in this thesis.

Among other domains, transfer learning has been applied to computer vision, in order to achieve higher prediction accuracy on problems with small training datasets. For example, Donahue et al. (2014) apply transfer learning to image classification. They demonstrate that it is possible to reuse the hidden layers of a convolutional neural network, trained on a large set of natural images. They use the hidden layers to process images from related problems with a small training dataset. This way they obtain latent image representations of each of the processed images. The related problems include images of objects commonly found in the office (Saenko et al., 2010) and images of birds (Welinder et al., 2010). Further, they train a logistic regression to perform the new tasks, given the extracted latent image representations. In effect, they solve new problems using a deep neural network with fixed, pre-trained hidden layers and a trainable task-specific output layer. Therefore, inductive bias is introduced by fixing most parameter values and thus reducing the hypothesis space. The results show that this way of transfer learning achieves significantly higher performance than the baseline when the training dataset is small.

Transfer learning techniques are also applicable to other domains. For example, sentiment classification is a task within natural language processing, which involves classifying an online product review as positive or negative. Reviews of different types of products (e.g. cameras and ovens), can share the same vocabulary but will have a different word distribution, since a review is likely to use product-specific terms. Despite this difference, the same adjectives, e.g. “good”, can be indicative of a positive review. Therefore, instead of producing large amounts of labeled data for each product type, a model could be trained on a single product type, and then adapted to other product types, which would reduce the required training dataset size (Pan and Yang, 2009).

In this section, I will further examine transfer learning for supervised deep learning algorithms. For a different taxonomy of transfer learning, as well as an overview on how it is applied to different machine learning algorithms, the reader is directed to survey papers available online (Pan and Yang, 2009; Weiss et al., 2016; Taylor and Stone, 2009; Tan et al., 2018; Zhuang et al., 2020).

In transfer learning, one aims to make use of knowledge from a related supervised learning problem. The problem with a low amount of data that we are trying to solve is referred to as the *target problem* (denoted Ψ_T), while the related problem with more training data is known as the *source problem* (Ψ_S). Unless stated otherwise, I assume there is only one source problem. As defined in Section 2.1, the source problem is given by a dataset D_S, generated from a domain $\mathcal{D}_S$ and a task $\mathcal{T}_S$. The target problem is similarly defined.

Definition 2. (Transfer Learning) Given a source problem $\Psi_S = (\mathcal{D}_S, \mathcal{T}_S, D_S)$, a target problem $\Psi_T = \mathcal{D}_S, \mathcal{T}_S, D_S$ and a fixed number of computational resources, transfer learning aims to develop an algorithm $\mathcal{A}$, s.t.

$$\mathcal{R}_T(\mathcal{A}(D_S, D_T)) < \mathcal{R}_T(\mathcal{A}(\emptyset, D_T)) \quad ,$$

where $\mathcal{R}_T$ is the risk of a hypothesis on the target problem.

The definition states that the expected loss of a hypothesis, obtained using transfer learning, should be lower than a hypothesis, obtained only using the target training dataset. Compared to a standalone model, trained only on the training dataset of the target problem, a transfer learning approach can potentially result in three benefits: a higher start, a higher slope and/or a higher asymptote (Olivas et al., 2009). In other words, the performance could be higher at the start of training, during training or at the end of training. To capture this, some have also assessed knowledge transfer by comparing the *learning curve area* (LCA) (Chaudhry et al., 2018b). For example, this can be the area under the test accuracy curve, and captures how quickly a ML algorithm reaches good generalisation performance.

Approaches to transfer learning differ in the setting they assume. Firstly, they differ in the number of labelled examples available in the source and the target problem. Unless stated otherwise, this thesis assumes that the source problems have a large amount of labelled data, while a small amount of labelled data is available for the target problem. Secondly, approaches differ in the assumed difference between the source and target problems. For a target problem $\Psi_T = \{\mathcal{D}_T, \mathcal{T}_T\}$ to be different from the source problem $\Psi_S = \{\mathcal{D}_S, \mathcal{T}_S\}$, it is assumed that $\mathcal{D}_S \neq \mathcal{D}_T$ and/or $\mathcal{T}_S \neq \mathcal{T}_T$. *Domain adaptation* is a subfield of transfer learning, which operates in the setting where inputs from both problems have the same feature space but have different marginal distributions, while the tasks are the same ($\mathcal{X}_S = \mathcal{X}_T$, $P_S(X_S) \neq P_T(X_T)$, $\mathcal{T}_S = \mathcal{T}_T$). This is similar to the setting of *covariate shift*, which assumes a single problem, but a change in the input distribution in the test set (Storkey, 2009). One example is sentiment classification for reviews of books and electronics. Another common setting is when problems have the same inputs, but different label spaces ($\mathcal{D}_S = \mathcal{D}_T$, $\mathcal{Y}_S \neq \mathcal{Y}_T$). An example of this is doing part-of-speech tagging and sentiment classification on the same type of reviews. It is also possible to have different input marginal distributions and tasks ($\mathcal{X}_S = \mathcal{X}_T$, $P_S(X_S) \neq P_T(X_T)$, $\mathcal{Y}_S \neq \mathcal{Y}_T$). For instance, this is the case when the source problem is given by the ImageNet dataset and the target problem by Caltech-UCSD Birds, as in the motivating example explored earlier (Donahue et al., 2014).

Approaches to transfer learning can be divided into *shallow* and *deep*, depending on whether they use classical machine learning or deep learning models, respectively. Shallow methods have been described in surveys (Tan et al., 2018; Weiss et al., 2016; Csurka, 2017) and a detailed description of them is out of the scope of this thesis.

2.3.1 Deep Transfer Learning

Deep transfer learning encompasses a number of techniques which are surveyed in Tan et al. (2018) and Csurka (2017). This section focuses on a large portion of deep approaches to transfer learning which are based on reusing or repurposing pre-trained parameters. First, one trains a deep neural network, referred to as the source model, on the source problem. Second, one creates a target model for the target problem using usually the same neural architecture as the source model. A subset of the target model's parameters are initialised with the values of the corresponding source model's parameters. These are referred to as the *transferred parameters*. The rest of the target model's parameters are randomly initialised. Third, an important decision is how to train the target model. The transferred parameters can either be unchanged (*frozen*) (Donahue et al., 2014; Sharif Razavian et al., 2014; Oquab et al., 2014) or optimised on the target problem (*fine-tuned*) (Girshick et al., 2014; Raffel et al., 2019; Agrawal et al., 2014). Freezing the transferred parameters provides inductive bias by restricting the hypothesis space of the target model. This is because the target model has fewer parameters to optimise. Fine-tuning the transferred parameters provides inductive bias by re-ordering the hypothesis space. This is because stochastic gradient descent updates parameters by slightly changing their current value. Therefore, an implicit preference is given to the optimal parameter values near the initial ones. It is also possible to freeze some of the transferred parameters and finetune the rest. To better understand the benefits and considerations involved in making this decision, one needs to consider the related work.

One common approach to deep transfer learning is *deep feature extraction*. It transfers the hidden layers from the source model and freezes their parameters. In effect, the source model is used to provide a latent representation of the inputs from the target problem. An example of deep feature extraction was given at the beginning of the section (Donahue et al., 2014), where the hidden layers of a convolutional neural network, pre-trained on the ImageNet dataset are used to extract features from Caltech-UCSD Birds. A logistic regression is trained to perform classification, based on the extracted features.

Alternatively, one can transfer layers from the source model and choose to fine-tune the transferred parameters. Since the source and target problems are different, the weights of a deep model should ideally also be trained on the target problem. However, if the target training dataset is small, fine-tuning all of the transferred parameters can lead to overfitting and thus poor generalisation performance. This can be addressed by only fine-tuning a subset of the transferred parameters while freezing the rest. Therefore, one needs to decide which layers to freeze. It has been observed that the hidden layers of a trained deep neural network compute an increasingly more abstract representation of the input (Kozma et al., 2018). The lower hidden layers are specialised towards processing the input, while the higher hidden layers perform more task-specific computations. Therefore, one could try freezing different transferred layers, depending on how the source and target problems are related. If the two problems have similar input domains, then the parameters of the lower hidden layers could be frozen. Otherwise, if the two problems have the same task but different input domains, one could try freezing the higher layers, while fine-tuning the lower hidden layers. For a hidden layer with frozen parameters to be useful, it needs to be reusable across problems. Yosinski et al. (2014) explore the reusability of different hidden layers of a convolutional neural network. The authors create a source and a target problem by splitting the ImageNet dataset into two disjoint subsets, containing different objects. In this setting, the source and target input domains are similar, but the tasks are different. Each dataset contains about 645,000 examples, therefore training a model from random initialisation achieves a high generalisation performance. The reusability of the first n hidden layers of the source model is evaluated as follows: the first n hidden layers are transferred and frozen, while the rest of the parameters are randomly initialised and trained on the target dataset. If this does not decrease the performance, compared to a randomly initialised model, trained on this big dataset, the first n layers are said to be reusable. The results show that the first two layers of the eight-layered model are reusable, with the performance deteriorating increasingly, as more layers are transferred and frozen. This supports the idea that when applying transfer learning in this setting, one should freeze the lower layers, as higher layers are less readily reusable and might need to be fine-tuned. However, as pointed out in Guo et al. (2019), it is not clear whether this trend holds for multi-path architectures such as Residual Networks (He et al., 2016), because their skip connection could result in some later layers being more reusable than earlier ones. Overall, one should decide which layers to freeze and which to fine-tune by considering the difference between the source and the target problems as well as the properties of the particular neural architecture.

There are different ways of combining freezing and fine-tuning. In different settings, authors have fine-tuned all network parameters (Girshick et al., 2014) or only the parameters of the last few layers (Long et al., 2015). It is also possible to perform *discriminative fine-tuning*, assigning different learning rates to different layers, as suggested in Howard and Ruder (2018). The same paper also proposes "gradual unfreezing", in which they first unfreeze the last layer, train it for one epoch, and proceed to unfreeze the layer before it, until the whole network is finetuned. To train pretrained and new parameters, Wang et al. (2019) perform a two-stage training approach, where they first freeze the pretrained weights and train the new parameters. Secondly, they unfreeze the pre-trained weights and fine-tune the whole model together on the new task.

It is worth considering how one could make the source parameters more transferable. More transferable would mean that the parameters learned on the source problem are more likely to increase the model's performance on a target problem, compared to a standalone baseline. It is possible that improving the source model's performance on the source problem would make the parameters more transferable. One way to achieve this is to increase the number of datapoints in the source training dataset. Huh et al. (2016) report that, if the source dataset is ImageNet, adding more training datapoints to the source dataset, unsurprisingly improves the source model's performance on the source problem. However, the improved source model only resulted in a marginal improvement when transferred to the target problem. Another way to improve a model's performance on the source problem is to change the architecture of the deep neural network. Kornblith et al. (2019) show evidence that architectures which perform better on the source problem lead to a better transfer performance on the target problem. The authors reached this conclusion after examining 16 different neural architectures, using ImageNet as the source dataset and 12 different image classification target problems. Finally, one could try training the source parameters to be robust to adversarial attacks. Salman et al. (2020) found that adversarially robust models trained on ImageNet are more transferrable to target classification tasks.

2.4 Multi-Task Learning

Multi-task learning (MTL) addresses the scenario where one is given two or more problems and the goal is to learn to solve all of them simultaneously (Caruana, 1997; Zhang and Yang, 2021). Typically, the input space is assumed to be the same, while the input distributions might be different (Yang and Hospedales, 2014; Zhang and Yang, 2021). On the other hand, the tasks are assumed to be different. They are usually of equal importance, but one might be only interested in performing a main task, in which case the rest of the tasks being learned are considered auxiliary. MTL is similar to the transfer learning (TL) case with $T_S \neq T_D$, where the source task is considered auxiliary. However, while multi-task learning requires access to the source dataset, deep approaches to transfer learning do not. Approaches to MTL and TL can both make use of parameter sharing (Zhuang et al., 2020).

Definition 3. (Multi-Task Learning) Given V problems $\{\Psi_i = ((\mathcal{X}, p_i(\mathbf{x})), \mathcal{T}_i, D_i)\}_{i=1}^{V}$, which share an input space $\mathcal{X}$, multi-task learning aims to develop an algorithm $\mathcal{A}$, s.t.

$$\forall i \in 1,...,V : \mathcal{R}_i(\mathcal{A}(\{D_j\}_{j=1}^{V})) < \mathcal{R}_i(\mathcal{A}(\{D_i\})) \quad ,$$

where $\mathcal{R}_i$ is the risk of a hypothesis on the ith problem.

The definition states that an MTL algorithm should improve the expected loss of all of each of the tasks it learns (Zhang and Yang, 2021). Alternatively, one might be interested in improving the average loss across all tasks. MTL's idea is that by training on multiple related tasks simultaneously, the algorithm will select a more generalisable hypothesis, as it works across multiple tasks. Similarly to TL, if the tasks are not related, the algorithm might select a hypothesis which achieves lower performance on the problems than if the model was trained in a single-task setting. This is known as *negative transfer, task interference* or *catastrophic interference*. In deep MTL, negative transfer is mainly addressed by altering the way parameters are shared. Therefore, I outline different parameter sharing strategies next. Other research problems in MTL, e.g. task scheduling, are less relevant to the topic of this thesis and are not covered here. For an overview of issues and approaches with deep MTL, the reader is referred to Crawshaw (2020).

The most popular group of approaches to MTL is hard parameter sharing, in which all tasks share a subset of the parameters. In particular, a commonly used neural architecture is a DNN with L layers, in which the first H hidden layers are shared between the tasks and the rest $L-H$ layers have separate task-specific parameters. Each task is then trained using an individual loss function and dataset. Keeping this as a working example, there are several potential benefits of using MTL, as described in Caruana (1998). First, for tasks T_1, T_2 and a shared feature F, learning a common representation would mean that F can be learned using data samples from both tasks, which could improve its generalisability (*statistical data amplification*). Second, given tasks T_1, T_2 and a shared feature F, if F can be learned through T_2, but cannot be learned from labels of T_1, sharing representation layers with MTL would mean that F would be accessible to T_1, thus improving its performance (*eavesdropping*). Third, when training on multiple tasks simultaneously, the back-propagated gradients could destructively interfere and prevent the weights from going to a task-specific local minimum, increasing the chances of converging to a mutually beneficial minimum (*representation bias*). Importantly, in deep MTL, a hidden unit can be specialised for one task, while a task can choose to ignore a certain feature by assigning a small weight to it. As practical advice, the paper also suggests that one could try introducing a task-specific learning rate in order to make certain that all tasks reach the best performance around the same time and, therefore, are able to take advantage of each other's representations.

One important disadvantage of hard parameter sharing is that it can suffer from negative transfer, if the tasks learnt simultaneously are too different. Alternatively, soft parameter sharing approaches use completely separate parameters for solving each task. Instead, they encourage corresponding parameters for each task to be similar (Ruder, 2017). One way to achieve this is to penalise the difference between parameters using the l2 distance (Duong et al., 2015).

Another group of approaches aim to learn which parameters to share between tasks. For example, Misra et al. (2016) first shows empirically for two pairs of computer vision tasks that MTL is beneficial for the performance on each task and that the best DNN architecture for MTL depends on the given tasks. The paper proposes to learn a sharing structure between tasks using *cross-stitch units*. Initially, two separate models are trained on the two tasks. Afterwards, cross-stitch units are used to replace the task-specific hidden activation of each layer by a parameterised linear combination of the corresponding hidden activations for both tasks. A different approach to task-specific weight sharing is presented by (Rosenbaum et al., 2017). The paper introduces routing networks, in which the parameters of each hidden layer are selected based on its input. This selection is made by a separate parameterised model, called a routed. Routing networks are described in more detail in Section 3.1.

2.5 Lifelong Learning

In transfer learning and multi-task learning, the algorithm operates on a preset number of tasks. However, it is possible that after the initial model is trained on a number of tasks, one would want the model to also learn to solve a new problem, for which data had not been previously available. For example, having trained a model capable of accurate part-of-speech tagging and sentiment analysis, one can become interested in natural language inference (NLI). Instead of training solely on the NLI dataset, it could be better to adjust the previously trained model to the new problem, in order to benefit from the shared knowledge between the related tasks. As new problems arrive, it would be useful to be able to continue adapting the model to them, while benefiting from previous knowledge whenever possible. Furthermore, it is assumed that there is still interest in performing the previous tasks. Therefore, it is important to at least maintain the model's generalisation performance on previously solved problems. This would also preserve the previously obtained knowledge, allowing it to be reused on new problems that could arise at a later time.

Multi-task learning provides a possible solution. Whenever a new problem is available, a MTL algorithm can be used to train a new model on all problems. This approach requires the datasets for problems to be stored, which is problematic as it could require a lot of storage space, or datasets might need to be deleted for privacy reasons. Moreover, having to retrain on all tasks with every new dataset, could make the approach prohibitively computationally expensive (Zhang and Yang, 2017). These issues exacerbate as the number of problems and the sizes of the datasets increase, e.g. imagine a sequence of a thousand problems. Another issue is that MTL assumes that all of the problems have the same input space (Yang and Hospedales, 2014), which does not need to be the case in a general setting. In the example above, a new problem could involve performing sentiment analysis on audio, instead of text. Therefore, a new class of algorithms is needed to address this setting. The area which aims to develop them is known as *Lifelong Machine Learning* (LML) (or *Lifelong Learning*), *Continual Learning* and *Incremental Learning*.

Definition 4. (Lifelong Machine Learning) "Lifelong Machine Learning (LML), considers systems that can learn many tasks over a lifetime from one or more domains. They efficiently and effectively retain the knowledge they have learned and use that knowledge to more efficiently and effectively learn new tasks." (Silver et al., 2013)

Currently, work on deep approaches to lifelong learning is also referred to as "continual learning" (CL). Here, similar to De Lange et al. (2019), I focus on problem-incremental continual learning. In this setting, each problem is different. A new problem is defined by a whole dataset, which the algorithm can process offline, possibly over multiple epochs. Moreover, the datasets of previous or future problems are not accessible. When evaluating, the model is given an input and the index of the input's problem. There are two common instances of problem-incremental continual learning: domain-incremental and task-incremental. Domain-incremental sequences have different input domains but share the same task. Task-incremental sequences have different tasks but share input space.

Definition 5. (Problem-incremental Continual Learning) Given t problems $\{\Psi_i = (\mathcal{D}_i, \mathcal{T}_i, D_i)\}_{i=1}^t$, and a new problem Ψ_{t+1}, *Problem-incremental Continual Learning* aims to develop an algorithm $\mathcal{A}$, which outputs a hypothesis h_{t+1} such that:

$$h_{t+1} = \mathcal{A}(D_{t+1}, h_t)$$

$$h_i = \mathcal{A}(D_i, h_{i-1})$$

$$h_0 = \emptyset$$

s.t.

$$\forall i \in 1, \ldots, t+1 : \mathcal{R}_i(h_{t+1}) \leq \mathcal{R}_i(\mathcal{A}(D_i, \emptyset)) \quad ,$$

where $\mathcal{R}_i$ is the risk of a hypothesis on the ith problem. Ideally, $\mathcal{A}$ minimises the computational and memory requirements.

There are few things to note in this definition. First, the algorithm takes as input only the dataset of the latest problem, and the previous hypothesis. Therefore, the algorithm does not have direct access to the previous datasets. Second, the risk for each problem is computed using the latest hypothesis. This means that, after learning to solve a new problem, the algorithm should be able to improve, or at least maintain, its performance on previous problems. The following continual learning properties are commonly referenced in the literature: *positive forward transfer*, *avoiding negative forward transfer*, *backward transfer* and *avoiding catastrophic forgetting*. These are detailed next.

As an illustrative example, consider a sequence of 3 problems: Ψ_1, Ψ_2, Ψ_3, where Ψ_1 and Ψ_3 share similar hypotheses. First, to avoid forgetting, a CL algorithm that is provided with Ψ_2, should make sure that its new hypothesis does not perform worse on Ψ_1. In general, when a new problem Ψ_i is given, and the performance of the new hypothesis h_i on a previous problem decreases (when $\mathcal{R}_j(h_{i-1}) < \mathcal{R}_j(h_i), 1 \leq j < i$), then forgetting is said to have occurred. If said performance decreases dramatically, i.e. $\mathcal{R}_j(h_{i-1}) << \mathcal{R}_j(h_i), 1 \leq j < i$), *catastrophic forgetting* is said to have occurred. In practice, deep learning algorithms can experience catastrophic forgetting when they change previously learned parameters to better fit the new task. Second, when a CL algorithm is given Ψ_3, it should be able to further improve its performance on the third problem by transferring the knowledge acquired from the first problem Ψ_1, since both problems share similar hypotheses. If h_3's performance on Ψ_3 is higher than the performance of a model trained only on Ψ_3, *positive forward transfer* is said to have occurred. Conversely, if h_3's performance is lower, the algorithm has performed *negative forward transfer*. In general, for a new problem Ψ_i, a CL algorithm achieves positive transfer when $\mathcal{R}_i(h_i) < \mathcal{R}_i(\mathcal{A}(D_i, \emptyset))$ and negative transfer when $\mathcal{R}_i(h_i) > \mathcal{R}_i(\mathcal{A}(D_i, \emptyset))$.

Therefore, CL algorithms should selectively transfer previous knowledge, in order to promote positive transfer and avoid negative transfer. Finally, given Ψ_3, a CL algorithm should use the new dataset to also improve its performance on Ψ_1. This is referred to as *backward transfer* or *reverse transfer*. In general, backward transfer is defined similarly to catastrophic forgetting, as it is the opposite. Given a new problem Ψ_i, if the performance of the new hypothesis h_i on a previous problem increases ($\mathcal{R}_j(h_i) < \mathcal{R}_j(h_{i-1}), 1 \leq j < i,$), then reverse transfer is said to have occurred.

Another two properties which are commonly found in LML literature are *plasticity* and *stability*. Stability refers to an algorithm's ability to retain previously acquired knowledge and is required in order to avoid catastrophic forgetting. It is usually addressed by placing a restriction on how a model updates on new problems. Conversely, plasticity refers to an algorithm's ability to continuously learn to solve new problems. For optimal plasticity, an algorithm's generalisation performance on a given problem should be at least as good as when said problem is solved in isolation. We distinguish between two causes of reduced plasticity. First, if a model is severely restricted on how it can update on new tasks, this can affect its *capacity* to represent new hypotheses. Second, even if a model has enough capacity, using harmful inductive bias from previous problems would lead to negative transfer, and thus reduced performance on the new problem.

A similar idea to lifelong learning is that of never-ending learning, presented in Mitchell et al. (2018). In never-ending learning, the agent improves over a series of tasks in a semi-supervised way. Moreover, the agent is also responsible for formulating new problems, which it needs to solve. The paper presents a specific implementation of such an agent - the Never-Ending Language Learner (NELL). Given initial ontology, the agent processes web pages in a random-walk fashion and populates its knowledge base with confidence-weighted beliefs. It is shown that, with time, the agent improves and is able to gather more information from a web page. One disadvantage of NELL is that it organises its knowledge in a structured way, therefore, it is limited in the type of knowledge, which can be learned. There are other lifelong learning approaches, which do not make use of deep learning and are therefore omitted from this thesis. For more information on them, the reader is referred to Chen and Liu (2018). The rest of this section covers different deep learning approaches to continual learning.

2.5.1 Common Experiments and Measurements

Most approaches assume that the input space is the same for all encountered problems, while the input distributions can be different. Even though these are different problems, they are often referred to as different tasks. For consistency, I will adopt this terminology while describing different approaches. Moreover, most CL methods use a common architecture, which shares a learned hidden representation across tasks $\phi(\mathbf{x};\boldsymbol{\theta})$. Depending on the experiments, the output layer can be shared between tasks (*single-head architecture*) or be task-specific (*multi-head architecture*). After training on task t, the prediction for the jth input of task i, $\mathbf{x}_i^{(j)}$, is expressed as $\hat{h}_t^{(i)}(\mathbf{x}_i^{(j)})$.

2.5.1.1 Measurements

As detailed above, two desirable properties of a CL algorithm are positive transfer and avoiding negative transfer. Both are related to the algorithm's performance on a new task and are mutually exclusive. After training on task t, we can use a standalone baseline to evaluate which of the two has occurred:

$$TR_t = \mathcal{R}_t(\mathcal{A}(D_t,\emptyset)) - \mathcal{R}_t(h_t) \quad .$$

If TR_t is positive, then the algorithm has achieved positive transfer, and conversely, if Tr is negative, then negative transfer has occurred. Note that we can approximate $\mathcal{R}_t$ using a test dataset for task t. The other two desirable properties of a CL algorithm are backward transfer and avoiding catastrophic forgetting. They both relate to the performance of a past task i and a target task t, and can be evaluated by:

$$B_{t,i} = \mathcal{R}_i(h_i) - \mathcal{R}_i(h_t) \quad .$$

A positive $B_{t,i}$ value indicates that backward transfer has occurred on task i, while a negative $B_{t,i}$ value is evidence of catastrophic forgetting.

The aforementioned measures are evaluated one task at a time and can become difficult to analyse, as the number of tasks increase. Lopez-Paz and Ranzato (2017) propose evaluation metrics which summarise a CL algorithm's performance. For a sequence of T tasks, a matrix $C \in \mathbb{R}^{T\times T}$ is calculated, where $C_{i,j}$ is the test classification accuracy of h_i on task j. Moreover, $\mathbf{b} \in \mathbb{R}^{T\times 1}$ is the vector of test classification accuracy of the standalone baseline. The following measurements are proposed:

$$\textbf{Average Final Accuracy: } ACC = \frac{1}{T}\sum_{i=1}^{T} C_{T,i}$$

$$\textbf{Backward Transfer: } BWT = \frac{1}{T-1}\sum_{i=1}^{T-1} C_{T,i} - C_{i,i}$$

$$\textbf{Zero Shot Transfer: } ZST = \frac{1}{T-1}\sum_{i=2}^{T} C_{i-1,i} - \mathbf{b}_i \quad .$$

Zero-shot transfer evaluates the performance of a model h_t on the next task h_{t+1}. The authors name this "forward transfer", but I consider this naming to be misleading, thus have renamed it here. Instead, I define forward transfer to compare a model h_t's performance on task t, compared to a standalone baseline, in order to capture positive and negative transfer:

$$\textbf{Forward Transfer: } FWD = \frac{1}{T-1}\sum_{i=2}^{T} C_{i,i} - \mathbf{b}_i \quad .$$

Note that the measures imply that all tasks are equally important. The Average Final Accuracy is the most commonly used in papers to compare different approaches.

2.5.1.2 Common Datasets

To evaluate a CL algorithm, one needs a sequence of problems. The more diverse the sequence, the more challenging it is. For example, a sequence can introduce different label spaces with each task ($\mathcal{Y}_i \neq \mathcal{Y}_j$, if $i \neq j$). Moreover, datasets can be selected to increase the difference of the input distributions between the tasks. One commonly used sequence of tasks is *Permutted MNIST*, first described in Goodfellow et al. (2013). Each task consists of the MNIST (LeCun et al., 2010) image classification dataset, but has a different task-specific random permutation of the inputs. This sequence evaluates an algorithm's ability to reuse the underlying problem structure across tasks. Given a new task, a model should learn to associate new collections of pixels to pen strokes, without forgetting old connections between pixels and pen strokes (Goodfellow et al., 2013). Because the random permutation loses an image's local properties, on which convolutional layers rely, this sequence is used to train models only with fully-connected layers. Since the label space and the target labelling function are the same between tasks, a single-head architecture is used.

Another common task sequence is *Split MNIST*, introduced by Zenke et al. (2017). They divide the MNIST dataset into five binary classification problems: 0/1, 2/3, 4/5, 6/7, 8/9. Methods usually use a multi-head architecture for this experiment. Similarly, the authors define "Split CIFAR10" and "Split CIFAR100", based on the CIFAR10 and CIFAR100 datasets (Krizhevsky et al., 2009) respectively.

While there are benchmarks designed to evaluate LML's performance in a reinforcement learning setting (Schwarz et al., 2018a; Wołczyk et al., 2021; Nekoei et al., 2021), this thesis' focus is on supervised learning. Therefore, we only mention them for completeness and refer the reader to the review in (Khetarpal et al., 2020).

2.5.2 Approaches

One could try to apply a transfer learning approach, namely, fine-tuning, to continual learning (a baseline also referred to as SGD). However, at task t, fine-tuning optimises the network parameters in order to only minimise the new loss function $L_t(\boldsymbol{\theta})$. Therefore, the loss on previous tasks increases, leading to catastrophic forgetting. Goodfellow et al. (2013) investigate how different choices for a neural network's architecture and training affect catastrophic forgetting. These choices are evaluated on three sequences of length 2, which is similar to the transfer learning setting. The authors report that using Dropout (Srivastava et al., 2014) during training leads to the least forgetting (a baseline referred to as SGD+Dropout). However, SGD and Dropout do not scale to longer task sequences and are outperformed by methods specifically designed for continual learning (Kirkpatrick et al., 2017). Recently, Mirzadeh et al. (2020) have shown evidence that, using carefully tuned hyperparameters, along with well-known techniques, such as dropout and large learning rate with decay and shrinking batch size, can result in an approach which outperforms other more complex LML algorithms on three standard continual learning benchmarks.

Most of the current approaches to lifelong learning can be grouped into the following three categories: parameter-regularisation-based, replay-based and architecture-based approaches. Parameter regularisation methods restrict how much a model's parameters can change when learning to solve a new problem. Replay methods revisit data points from previous problems while training on a new problem. Finally, architecture-based methods change the model's neural architecture between problems. For example, this can involve adding new problem-specific parameters (Rusu et al., 2016) or computing a problem-specific parameter mask (Mallya et al., 2018). Catastrophic forgetting is usually prevented by freezing previously trained parameters.

2.5.2.1 Parameter Regularisation

Parameter regularisation methods address forgetting by storing additional parameter-specific information from past tasks, and using it to augment the training loss function of a new task.

Bayesian parameter regularisation methods are based on the idea of expressing the posterior distribution over the model parameters, given data from all tasks as follows:

$$\begin{aligned} p(\boldsymbol{\theta}|D_{1:T}) &= \frac{p(D_{1:T}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(D_{1:T})} = \frac{p(D_T|\boldsymbol{\theta})p(D_{1:T-1}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(D_T|D_{1:T-1})p(D_{1:T-1})} \\ &= \frac{p(D_T|\boldsymbol{\theta})p(\boldsymbol{\theta}|D_{1:T-1})}{p(D_T|D_{1:T-1})} = \frac{p(D_T|\boldsymbol{\theta})p(\boldsymbol{\theta}|D_{1:T-1})}{\int p(D_T|\boldsymbol{\theta})p(\boldsymbol{\theta}|D_{1:T-1})d\boldsymbol{\theta}} \quad . \end{aligned} \tag{2.1}$$

This means that it is not necessary to have access to all datasets. To approximate the latest posterior over parameters $P(\boldsymbol{\theta}|D_{1:T})$, one "only" requires the latest dataset D_T and the previous posterior $P(\boldsymbol{\theta}|D_{1:T-1})$. This is referred to as *Bayesian evidence accumulation* (BEA). Usually, computing the true posterior is computationally intractable, so approaches aim to estimate it. In addition to estimating the posterior, some methods add heuristic modifications to the resulting loss function in order to improve the empirical performance of their methods. Applying log on each side of the equation leads to:

$$\log p(\boldsymbol{\theta}|D_{1:T}) \propto \log p(D_T|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}|D_{1:T-1}) \quad .$$

This can be interpreted as $\log p(D_T|\boldsymbol{\theta})$ being the loss term defined over the latest dataset, and $\log p(\boldsymbol{\theta}|D_{1:T-1})$ being the regularization term. Therefore, methods which follow BEA are also referred to as *regularisation-based.*

Elastic weight consolidation (EWC) (Kirkpatrick et al., 2017) is a method inspired by BEA (Kirkpatrick et al., 2018). The method is a modified version of computing a diagonal Laplace approximation of the posterior. The posterior of the first task $p(\boldsymbol{\theta}|D_1)$ is approximated by a Gaussian distribution centered around the maximum likelihood estimate $\hat{\boldsymbol{\theta}}_1$. The Gaussian distribution has a diagonal precision given by the diagonal of the Fisher information matrix (FIM) F_1. The function minimised in EWC for the second task is

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_2(\boldsymbol{\theta}) + \sum_i \frac{\lambda}{2} F_1^{(i)} \left(\boldsymbol{\theta}^{(i)} - \hat{\boldsymbol{\theta}}_1^{(i)}\right)^2 \quad , \tag{2.2}$$

where $\mathcal{L}_2$ is the loss over the dataset of the second task, i is the index of each parameter and λ is a hyper-parameter, setting the importance of problem 1. When applied to multiple tasks, EWC computes a different approximation of the weights posterior of each task and then uses all of them for regularisation. This can be problematic, since the algorithm's computational and storage demands increase with the number of encountered tasks. To address this, Schwarz et al. (2018b) and Chaudhry et al. (2018a) augment EWC by accumulating all of the Fisher information matrices into a single one and allowing only one approximation to be stored at a time. EWC also assumes that FIM is a diagonal matrix, which is typically unlikely. To address this, Liu et al. (2018c) decompose the expression for computing an FIM and use it to derive a reparameterization of the layers of a neural network which encourages FIM to be approximately diagonal. On the other hand, Ritter et al. (2018a) aim to remove the diagonal restriction. They compute Kronecker factored Laplace approximation (Ritter et al., 2018b) which allows the resulting posterior to capture the interaction between weights within the same layer.

Variational Continual Learning (VCL) (Nguyen et al., 2018) applies BEA to lifelong learning and approximates the distributions over the model parameters with a isotropic multivariate Gaussian using Bayes by Backprop (BBB) (Blundell et al., 2015). This allows the approximation to be fitted during training instead of after finding the MAP as in when using Laplace approximation. Since a Gaussian might not capture the whole posterior of the parameters, repeating approximations for each task might lead to error accumulation, which in turn could cause catastrophic forgetting. In an attempt to remedy this, the authors suggest keeping a *coreset* - a fixed number of data points for all encountered tasks. They re-write the posterior over parameters in a way which includes the coreset and show that this leads to an improvement in performance. VCL can be seen as combining parameter regularisation with rehearsal-based methods.

EWC uses parameter uncertainty in the loss function, by allowing uncertain parameters to change from previously found optima. Similarly, Uncertainty-guided Continual Bayesian Neural Networks (UCB) (Ebrahimi et al., 2019) calculate parameter uncertainty using BBB, but use it to regulate the learning rate of the optimiser. The intuition is that more uncertain parameres should be allowed to change more rapidly, while more certain parameters' updates should be slow in order to prevent catastrophic forgetting.

There are other parameter regularisation approaches, which are not based on Bayesian approximations. Synaptic Intelligence (SI) (Zenke et al., 2017) optimises a loss function similar to EWC. At task t they minimise

$$\tilde{L}_t(\boldsymbol{\theta}) = L_t(\boldsymbol{\theta}) + c\sum_i \Omega_t^{(i)}(\boldsymbol{\theta}^{(i)} - \hat{\boldsymbol{\theta}}_{t-1}^{(i)})^2 \quad . \tag{2.3}$$

Here, L_t is the task-specific loss and c is a hyper-parameter which regularises the importance of remembering old tasks. Each parameter is penalised for changing from its optimal value on the previous task $\hat{\boldsymbol{\theta}}_{t-1}^{(i)}$, based on a parameter-specific coefficient $\Omega_t^{(i)}$, capturing how important the parameter has been to the loss of past tasks. To calculate $\Omega_t^{(i)}$, SI accumulates the partial derivatives of the loss with respect to the i-th parameter, observed across the training steps, which reflects on how important this parameter has been for changing the loss. This captures the change in a previous loss as the value of a parameter θ moves away from $\hat{\boldsymbol{\theta}}_{t-1}^{(i)}$ in the direction observed during training. However, this can underestimate the loss as the parameter's value changes in the other direction. Park et al. (2019) account for this by using a scalar to overestimate the loss in the unexplored direction. Memory Aware Synapses (MAS) (Aljundi et al., 2018) also optimise Eq. (2.3) for continual learning. In contrast to SI, MAS calculates a parameter's importance based on how much it affects the output of the neural network on average. The coefficient $\Omega_t^{(i)}$ is calculated using the partial derivative of the neural network's output $h(x;\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}^{(i)}$. Finally, Jung et al. (2020) propose to group the parameters that are part of the same hidden unit in a DNN, and regularise them with the same coefficient. The importance of each group is calculated using the average activation value of the associated hidden unit.

2.5.2.2 Memory-based Approaches

The parameter regularisation approaches which I discussed above maintain information about the optimal parameters on past tasks. This is then used to prevent important parameters to greatly change from their previously found optimal values. Instead, memory-based approaches assume that it is possible to retain some of the past data from previous problems, which can then be used to restrict further changes to the model in order to prevent catastrophic forgetting. One possible advantage of this approach is that it can allow for more flexibility in how the parameters change, as long as the constraints are satisfied. The main research questions revolve around how to best make use of the available data from past problems, how to select which past data to store and how to store past data.

Rehearsal-based approaches optimise their model on the available past data, while also training it to solve a new problem (Robins, 1995; Rolnick et al., 2019; Chaudhry et al., 2019b). In the supervised learning setting, this can involve simultaneously training the model to correctly predict the target labels of the stored inputs. Instead, Buzzega et al. (2020) store past inputs as well as the model's corresponding output logits. Motivated by knowledge distillation (Hinton et al., 2015), the stored logits are then used as targets during rehearsal.

Alternatively to directly training on it, past data can be used to modify the gradients computed using the new data. GEM (Lopez-Paz and Ranzato, 2017) modifies the gradient of the new loss $g = \Delta L_t(\mathbf{x};\boldsymbol{\theta})$ in order to align it with each of the gradients of the previous losses $g_j = \Delta L_j(D_j;\boldsymbol{\theta})$. This way, any updates to the parameters should either decrease or keep each of the expected losses on previous tasks L_j the same. However, each parameter update requires an extra optimisation step that involves all of the stored data, which prevents GEM from scaling to a large number of tasks and bigger number of stored data points. A-GEM (Chaudhry et al., 2018b) addresses this by computing an average gradient, g_{ref}, representative of the losses of past tasks, using a random subset of the past data. In contrast to GEM, Farajtabar et al. (2020) project g to be orthogonal to the gradients of a DNN's outputs on past data with respect to its parameters. This is designed to keep the past input-output mappings unchanged, as the DNN adapts to the new task.

Bayesian functional regularisation approaches (Titsias et al., 2019; Pan et al., 2020) compute a posterior over the function values of a selected subset of inputs from the training set. In effect, this represents a posterior over functions, given the training data. When training on a later task, they add a regularisation term to the objective, which ensures that shared parameters are constrained to change while being consistent with the previously computed functional posteriors.

Since it is typically not feasible to retain all of the previously observed data, methods store only a subset of it. As a result, a line of work investigates different ways of selecting which data point to retain. Isele and Cosgun (2018) outlines three approaches: random selection which preserves the underlying distribution, selecting the inputs associated to the highest prediction loss, selecting inputs by maximising the selected subset's coverage over the input space. Aljundi et al. (2019) suggest using the gradients in order to select the inputs which "pull" the model's parameters in the most diverse set of directions. Using hindsight to anchor past knowledge in continual learning. With respect to the number of the retained data points, Chaudhry et al. (2019b) presented evidence that even using a tiny amount of data points for rehearsal still leads to competitive LML performance. Instead of storing raw inputs, which can each be large in terms of their memory demands, Pellegrini et al. (2019) propose to store the latent embeddings. They assume that all tasks are related to image processing where the lower layers of a pre-trained CNN have been found to be reusable across tasks (Yosinski et al., 2014).

Another line of work develops approaches which avoid storing any of the past data. Given a new task, these approaches create a pseudo dataset which is considered to be representative of the data distribution of past tasks. Robins (1995) proposes to generate a pseudo dataset by first randomly sampling inputs from the input space and then labelling them using the model. Li and Hoiem (2017) propose to use the training inputs on the new task and label them with the task-specific output heads of their DNN. They then use a distillation-based multi-task objective to ensure that the task-specific outputs remain unchanged while the DNN is being trained. Shin et al. (2017) propose to use a generative model, namely a generative adversarial network (Goodfellow et al., 2014), to model the input distribution over all past tasks.

2.5.2.3 Architecture-based approaches

All previously described methods used a single-head or multi-head architecture, where the hidden layers of the neural network are shared between tasks. Even if data from all tasks is present, work on multi-task learning has shown that sharing all hidden weights can lead to catastrophic interference, reducing the performance on some of the tasks. Architecture-based continual learning methods change the model's architecture between tasks in order to reduce catastrophic forgetting, while allowing knowledge transfer.

One line of work contains methods which also utilise a multi-head architecture but use task-specific masks in order to restrict the parameter overlap between tasks. Piggyback (Mallya et al., 2018) assumes that the shared parameters are pre-trained on a large image dataset, on ImageNet. For each new task, the method learns a different binary mask $\mathbf{M}$ over the model's weights and the parameters of the task-specific output, while keeping the shared parameters frozen. After training, a binary mask over the shared parameters, costing 1 bit per parameter, is kept for each task. Since each task has a separate mask, and the shared parameters do not change, catastrophic forgetting is prevented by design. However, Piggyback is limited by its reliance on the initial values of the shared parameters. PackNet (Mallya and Lazebnik, 2018) allows the shared parameters to be updated on new tasks. After training on a new task, PackNet performs a step of iterative pruning. A task-specific binary mask is created by sorting the parameters of each layer by absolute value and discarding the lowest 50% by setting the associated mask value to 0. The rest of the mask values are set to 1. Next, the model is finetuned using the new mask, in order to adapt to the pruned weights. When training on following tasks, all previous masks are combined to freeze the parameters which useful for previous tasks. While PackNet relies on heuristic-based pruning to acquire a task-specific mask, *Hard Attention to the Task* (HAT) (Serra et al., 2018) learns a mask, together with the network parameters. Rather than masking parameters, HAT masks hidden activations. Moreover, instead of a strictly binary mask, the method allows intermediate values between 0 and 1 and a regularisation term is added to encourage mask values to be 0. After training on a new task, HAT computes cumulative attention $\mathbf{a}_{\leq t}^{(l,i)}$ which stores the maximum mask values for a given hidden activation. When training on following tasks, cumulative attention is used to restrict the gradient updates to parameters. This limits catastrophic forgetting by not allowing previously important parameters to change.

Constraining continual learning methods to use a constant amount of parameters limits the number of tasks which said methods can solve. This is why some approaches have the ability to grow the model's architecture given a new task. *Compacting, Picking and Growing* (CPG) (Hung et al., 2019) is a more parameter-efficient CL method which adds new parameters to its model on demand, while using pruning to keep its size small. The method has three main steps: pruning, masking and expanding on demand. After the model is trained on task 1, CPG uses gradual pruning (Zhu and Gupta, 2017) to prune the trainable parameters. The pruned parameters are marked as trainable for future tasks W_1^E, while the rest of the trainable parameters are added to a set of frozen weights W_1^F. Given a new task t, CPG trains the learnable parameters in W_{t-1}^E, as well as a task-specific binary mask M_t over W_{t-1}^F, similarly trained to the one in Piggyback. The mask is supposed to allow the model to selectively reuse part of the frozen weights from previous tasks W_{t-1}^F. After training, the final performance is evaluated. If the performance is not higher than a manually defined hyper-parameter, CPG increases the model's capacity by adding new parameters to W_{t-1}^F. Afterwards, M_t and W_{t-1}^F are re-trained. Since all the weights used in previous tasks are stored in W_{t-1}^F and frozen, CPG avoids catastrophic forgetting by design.

Progressive neural networks (PNNs) (Rusu et al., 2016) are another example of a method which expands its architecture. PNNs consider the usually used feedforward deep neural network as a column in their architecture. For each new task, a new task-specific column is introduced to the architecture and trained to solve said task. Moreover, the previously learned columns' parameters are frozen, thus avoiding catastrophic forgetting. As described, the setup is identical to training a different DNN for each task. In order to facilitate transfer between tasks, PNNs introduce lateral connections between columns. For a new task t, each input from the new dataset is processed by each column simultaneously. Moreover, each hidden activation $g_t^{(l)}$ for layer l of the new task t, is computed using the hidden activation from the previous layer $g_t^{(l-1)}$ as well as the hidden activations from the previous layers of the previous columns $g_{<t}^{(l-1)}$.

PNNs are evaluated on sequences of up to twelve Atari games and they demonstrate an ability to transfer better than the baselines. The authors also hint at possible optimisation difficulty, related to lateral connections, which prevents more effective transfer. The experiments further reveal that PNNs are prone to negative transfer. A big disadvantage of PNNs is that the number of parameters grows quadratically with the number of tasks. The authors demonstrate that only a fraction of the newly introduced parameters are used.

Another line of work considers a modular approach to LML. A deep neural network is decomposed into modules, where each module can be thought of as a sub-network that performs a parameterised non-linear transformation of its inputs. These methods select a task-specific subset of the available modules which is used to process the inputs of a given task. By grouping parameters into modules and only reusing a subset of modules per task, these methods allow their modules to specialise in performing an atomic transformation which is reusable across tasks. Furthermore, once trained, a module can be frozen in order to prevent forgetting. PathNet (Fernando et al., 2017) uses a modular deep neural network consisting of L module layers, each of which has d modules to choose from. For each task, they select a task-specific subset of modules, which involves selecting $k < d$ modules per layer. To make this selection, they use a genetic algorithm to search for a suitable subset of modules, while also optimising the modules' parameters. Rajasegaran et al. (2019) define a similar setting, however, they propose to use random search in order to select a task-specific subset of modules. Both of these methods use a constant amount of modules which limits the number of tasks which they can solve.

Veniat et al. (2020) also group parameters into multi-layer modules, with the hope that this would allow modules to perform atomic and more reusable transformations. Their method is called *Modular Networks with Task Driven Prior* (MNTDP). MNTDP considers a feedforward architecture of S modules. For task 1, all S modules $M_1^{(i)}$ are randomly initialised and learned. Afterwards, the modules are frozen for future tasks and added to the pool of modules which can be reused. For task 2, for each module index i, there is a choice between reusing the module $M_1^{(i)}$ and introducing a new randomly initialised module $M_2^{(i)}$. The authors refer to each unique combination of modules as a path π. Therefore, to solve task 2, MNTDP needs to select the best path and to train any new modules. Two strategies are proposed for this: Stochastic and Deterministic. The stochastic strategy (MNTDP-S) alternates between optimising the choice of a path using gradient descent via REINFORCE (Williams, 1992), and learning the new parameters using gradient descent and the supervised learning loss. The deterministic strategy (MNTDP-D) evaluates all possible paths by training a different copy of the new modules for each path. Either method can be used to select a path and train the new modules. Afterwards, the newly trained modules on task 2 are also frozen and added to the pool of options. For each task, the selected path can reuse some of the modules, however, the number of possible paths grows exponentially with the number of encountered tasks T: $O(T^S)$. This paper applies a number of heuristics to reduce the path search space. Given a new task t, the closest past task j^* is selected. This is done by processing the new dataset using each optimal path π_j for previous tasks. The search space is restricted to considering a combination of the modules from the closest path π_{j^*} and the new modules. This restricts the search space to $O(2^S)$. Another heuristic is to restrict the search to paths in which a pre-trained module can be connected to a new one, but a new module cannot feed into a pre-trained module. This restricts the method to only be able to transfer low-level perceptual knowledge. The authors notice that MNTDP-S leads to lower average accuracy than MNTDP-D, and reason that this occurs because training modules as part of different paths leads to sub-optimal convergence of their parameters.

Chapter 3

Background - Related Areas

This thesis presents a neurosymbolic framework in which neural networks are represented as functional programs. This leads to a modular lifelong learning approach which searches for a problem-specific neural architecture. In later chapters, we improve on our work by making use of Bayesian optimisation. Overall, our work benefits from the advantages of multiple sub-fields within deep learning. This chapter provides background into said sub-fields. Concretely, I present modular deep learning, neurosymbolic methods, neural programming, neural architecture search and Bayesian optimisation of black-box functions.

3.1 Modular Deep Learning

Typically, deep learning approaches have a fixed set of parameters, which are all used to make a prediction on a given input. Alternatively, one can assume that a given problem can be divided into sub-problems. For example, it can be assumed that the input space can be divided in two, with two different functions solving each part. In this case, modular approaches solve a problem by creating different sets of parameters, which are selectively applied, depending on the given input. These sets of parameters are sometimes referred to as modules and each module performs a nonlinear transformation of its input. The goal of modular approaches is that each module specialises in solving a different sub-problem. This divide-and-conquer approach can lead to a number of benefits (Rosenbaum, 2020). First, as different parts of the input data are processed by different modules, this could reduce the learning interference between data points and lead to higher performance. Second, since a different subset of modules can be used to process each input, the effective size of the whole model can be greatly increased, without affecting the time it takes to process an input (Shazeer et al., 2017). This increases the model's expressivity while keeping the computational complexity of a forward pass constant. Third, the parameters of a modular approach are more transferrable if each module focuses on a sub-problem. It is also possible to reassemble the modules in a novel way in order to solve a previously unseen problem (Chang et al., 2018). Fourth, as modules are reused only on related sub-tasks, this could reduce their catastrophic forgetting. The rest of the modules that are not relevant to a new problem are unaffected by any finetuning of the relevant parameters.

Given a problem, modular approaches have two main challenges. Selecting modules for each input and training the modules' parameters. If these challenges are not addressed properly, this could lead to different issues. For example, a modular approach could keep on selecting the same small subset of modules for every input - a problem known as *module collapse*. Another potential issue is *overfitting*, since having multiple modules increases the overall number of parameters.

3.1.1 Mixture of Experts

One modular approach is *Mixture of Experts* (MoE). (Jacobs et al., 1990). Each module is a neural network, referred to as an expert network, which makes a prediction about the label of a given input. MoE uses a gating network in order to combine the ouputs of all K experts e_i. The gating network $g(\mathbf{x})$ processes a given input and outputs an expert-specific weight $g_i = g(\mathbf{x})_i$. A MoE then outputs a linear combination of the experts' outputs:

$$\hat{y} = \sum_{i=1}^{K} g(\mathbf{x})_i e_i(\mathbf{x}) \quad . \tag{3.1}$$

This allows for a standard loss function to be used for training: $\mathcal{L} = \mathcal{L}(\hat{y}, y)$. However, this training procedure can lead to co-dependence between the experts, as all of them are used to collectively make a prediction. Jacobs et al. (1991) address this by changing the gating network g to output a categorical distribution over the choice of experts for a given input, i.e. $\sum_i^K g(\mathbf{x})_i = 1$. Afterwards, the loss function is changed by imagining that only one expert is sampled from the categorical distribution and used for prediction. As a result, the loss function can be changed to $\mathcal{L} = \sum_i^K g(\mathbf{x})_i \mathcal{L}(e_i(\mathbf{x}, y))$, which is the expected loss between a selected expert's prediction and the actual label. At test time, the prediction can still be made according to Eq. 3.1. Following this work, the gating networks in MoE are followed by a *softmax* activation by default. Overall, mixtures of experts require each of the experts to be run on the given input, which introduces memory and computational limitations that restrict the number of experts that can be used. Shazeer et al. (2017) scale this idea by introducing Sparsely-Gated Mixture-of-Experts (SGMoE). First, the authors treat a mixture of experts as a layer and introduce such a layer between two stacked LSTM layers. Setting the number of experts for each MoE layer to a high number dramatically increases the model's capacity. However, this model would be prohibitively computationally expensive. To address this, SGMoE uses the gating network to select the k experts with the highest probability out of all K experts. Since $K >> k$, some experts might never be selected. This is fixed by adding a learned input-specific noise to the logits of a gating network. The experiments demonstrate that SGMoE allows for the training of a model containing an MoE with up to 137 billion parameters. SGMoE achieves state-of-the-art performance on language modelling and machine translation benchmarks.

3.1.2 Routing Networks

Another popular direction for modular neural networks builds on *Routing Networks* (Rosenbaum et al., 2017). This framework has two main components: sets of modules and a router. Each input is processed by multiple layers of modules. For each layer, the router selects an input-specific module from a set of modules associated with said layer. Concretely, each layer l of the feedforward computation contains O_l modules. The router is a learnable function which uses the layer's input v_{l-1} to select a single module to be used. As a result, each input can be processed by a different composition of modules. Overall, there are different options for setting up the architecture of a routing network. For example, each layer could have the same list of modules which it should select from. In this case, the router can recursively apply the same list of modules, up to maximum depth d. As another example, if layers have a different number of modules, one could introduce multiple layer-specific routers. To train a routing network, we can use loss backpropagation to tune module parameters with SGD. On the other hand, training the router is more involved. Rosenbaum et al. (2017) use reinforcement learning to train the router. They evaluate routing networks in the multi-task learning setting. This is achieved by conditioning the router's decisions on the task's index as well.

Chang et al. (2018) show that routing networks can generalise to harder problems. They consider problems, where each is a combination of S sub-problems. In this setting, different combinations of sub-problems result in a different problem. For ex-

ample, multilingual arithmetic problems are considered, where each problem has an input encoded into a source language and a label encoded into a target language. Each source-target language pair is a different problem. The authors simultaneously train on multiple problems using multi-task learning. However, they want each of their modules to solve a sub-problem. Therefore, they make use of curriculum learning (Bengio et al., 2009), during training. Afterwards, they demonstrate that routing networks can generalise to previously unseen problems of higher difficulty. In the multilingual arithmetic example, they show that their model can generalise to previously unseen source-target language pairs (the target language is provided to the router). Moreover, the model generalises to longer arithmetic expressions.
Kirsch et al. (2018) propose a different way to train routing networks. They formulate a routing network as a probabilistic model and consider the choice of a module as a latent variable. Then, they train both the modules and the router using generalised Expectation-Maximisation (EM) (Jordan, 1998). The authors note that routing networks, as described in Rosenbaum et al. (2017), suffered from module collapse, i.e. only make use of a small subset of the modules, thus lacking module diversity. The experiments show that the new training algorithm alleviates module collapse, without the need for explicit diversity regularisation.

3.1.3 Modular Deep Learning for Visual Question Answering

In the previously discussed routing networks, the routers rely on the input and, optionally, on a task index in order to select the modules. It is possible to also supply meta-information about the given input, in order to specify the desired task. This finds application in visual question answering, where the question provides such meta-information. Here, I discuss neural module networks (Andreas et al., 2016), which can be seen as routing networks with a hand-engineered router that makes use of meta-information.

In visual question answering (VQA) datasets, a model is supplied with an image and a question and needs to learn to reason about the objects illustrated in the given image in order to successfully answer the given question. CLEVR (Johnson et al., 2017) is a synthetic dataset designed to evaluate a model's ability to reason. Unlike the previous datasets, it was designed to not contain statistical biases which can be exploited by the models.

Neural module networks (Andreas et al., 2016) is a modular deep learning approach to visual question answering. The authors notice that the input questions in VQA can be decomposed into sub-problems. For example, the question "where is the dog" can be decomposed into the sub-problems of *locating the dog in the image* and *recognising where it is situated*. The latter sub-problem is also contained in the question "where is the cat". For each question, the authors extract the sub-problems and assign a neural module to solve each unique sub-problem. The selected neural modules are assembled in an order consistent with the question, e.g. *where(dog(IMAGE))*. The output of the selected neural modules, together with an embedding of the question is used to provide an answer. This process is detailed next. First, the question is processed by a semantic parser to extract structure from sentences. The parser's output is then used to select a subset of neural modules and determine their layout. The input image is then processed by a shared sequence of convolutional layers (referred to here as CNN) and then by the neural modules. Next, the question is embedded using a recurrent neural network. The resulting embedding is combined with the output of the neural modules and used to output a final prediction. In the experiments, neural module networks are compared to a monolithic neural architecture, which uses the same parameters to process and answer all questions. It is found that this compositional approach outperforms the competing method on *SHAPES* - an easier two-dimensional version of CLEVR. On the other hand, it is only slightly better than the competitor on a dataset of natural images. The authors partially attribute this to the out-of-the-box parser, which they found made errors when parsing complicated questions.

To address this, the follow-up work presented in Hu et al. (2017), focuses on learning to assemble neural modules from data. They introduce an additional deep model which, given a question predicts a distribution over all possible layouts $p(l|q;\boldsymbol{\theta}_{layout})$. The whole pipeline is trained end-to-end on the target dataset, using backpropagation for the differentiable part and policy gradient for the non-differentiable part. The big search space over layouts makes this hard to train $\boldsymbol{\theta}_{layout}$ from random parameters, therefore the authors first train it on expert-provided question-layout pairs. On the *SHAPES* dataset, the new approach outperforms neural module networks. The authors then show that the new approach achieves 83.7 overall accuracy on the CLEVR dataset, which is significantly better than the competing monolithic methods. This is evidence that their approach is capable of reasoning about complex questions in the dataset. Overall, this line of work is able to exploit the problem compositionality by inducing a problem-specific bias. They augment neural approaches with the ability to symbolically describe neural networks, thus allowing for neural modules to specialise in sub-problems. Moreover, the question-specific layouts provide more transparency into how this approach interprets and answers a question.

3.2 Neurosymbolic Methods

Deep neural networks are flexible models that are good at learning from noisy inputs with a high dimension Despite their success, they currently have limitations. Firstly, DNNs require a considerable amount of data in order to train their many parameters. Secondly, DNNs typically do not generalise well to inputs sampled from a different distribution than the training distribution. Thirdly, DNNs make predictions based on many multiplications of their parameters with a given input. This makes it hard to interpret how a DNN's prediction is made.

Symbolic methods are another line of work, aimed at automating learning, knowledge discovery and decision making. These methods' basic unit is a symbol, which is a human-readable string and an abstract representation of an entity or a concept. Symbolic methods can use symbolic data to discover relations between different symbols. Moreover, formally specified inference rules can be used to deduce further sets of relations (Garnelo and Shanahan, 2019). In addition, symbolic methods can search through different symbol combinations in order to solve a problem. For example, in planning, given an initial state, a goal state and a set of actions, search algorithms can be used to find the string of actions leading to the goal state. Overall, symbolic methods allow for easy integration of human knowledge, making them data efficient. They can also operate on symbols from different tasks, making them generalisable. Finally, symbolic systems are interpretable, since every aspect of their design is human readable. One disadvantage of these methods is that the symbols and the rules are handcrafted, and are not learned from high-dimensional data. Moreover, it is hard for these methods to handle noisy data.

It can be seen that deep neural networks and symbolic approaches have complementary strengths. As a result, there is a line of work, referred to as *neurosymbolic* or *neural-symbolic* methods, which combines the two fields (Garnelo and Shanahan, 2019; Garcez and Lamb, 2020). In his AAAI 2020 keynote lecture, Kautz (2020) sorts neurosymbolic work into five categories, which are described next.

Category 1 encompasses standard deep learning methods since they can process symbolic inputs and produce symbolic outputs. Deep learning for translation and question answering are two examples from this category.

Category 2 includes methods which contain a neural sub-routine within a symbolic solver. One example of a category 2 system is AlphaGo (Silver et al., 2016) which is applied for playing the board game Go. This system has two deep neural networks - a policy network that outputs probabilities over possible moves, and a value network, which is trained to predict the winner, based on a given position. In addition, the system uses Monte Carlo tree search (MCTS) to search for the best move, since the search space is too big. MCTS makes use of the policy network in order to select subtrees, corresponding to moves which are more likely to be successful. After reaching maximum depth, MCTS uses the value network (in combination with fast rollout) to estimate the value of the leaf state.

Category 3 consists of methods, in which a system for manipulating symbols, using a set of rules is approximated by a neural network by training on the system's inputs and outputs. For example, Lample and Charton (2019) use deep learning for symbolic mathematics. They train a DNN to solve problems on function integration and ordinary differential equations. The input and labels are represented as trees and linearised using prefix notation. The authors make use of a sequence-to-sequence architecture, often used for translation. As a result, the trained model is shown to outperform commercial systems.

Category 4 encompasses systems in which the input is first processed by a DNN, and the DNN's output is processed by a symbolic system. This would allow the DNN to transform high dimensional input into a symbolic representation, and take advantage of a symbolic system's strong reasoning abilities. For example, Yi et al. (2018) present a neurosymbolic method for visual question answering (NS-VQA). NS-VQA has a scene parsing component, consisting of two DNNs, which transforms the image into a symbolic scene representation, describing each of the objects in a given image. There is also a question parsing component which transforms an input question into an executable program. This is based on an LSTM sequence-to-sequence architecture. Finally, NS-VQA uses a symbolic *program executor* to run the extracted program on the extracted symbolic scene representation of the input image. The program executor provides expert-written Python implementations for each possible function in the program.

Category 5 contains methods which embed symbolic reasoning inside a neural architecture. The difference from Category 4 is that Category 5 methods can selectively make calls to the symbolic component, when necessary.

3.3 Neural Programming

Lifelong learning involves re-using previously learned weights. At the same time, code reuse is an important concept in programming. The connection between programming and lifelong learning is made in Gaunt et al. (2016a); Reed and De Freitas (2015), and we explore a similar idea in Chapter 4. Therefore, literature, connecting deep neural networks and programming is described next. Given input-output examples, the methods try to approximate or synthesize the underlying algorithm, which conforms to the pairs.

Neural abstract machines are a class of methods, which aim to approximate an algorithm by fitting a deep neural network architecture on given input-output pairs. They are usually augmented with additional memory sources. Another common characteristic of these methods is that they don't produce source code, thus the induced programs aren't interpretable. For example, Neural Turing Machines (NTM) (Graves et al., 2014) are introduced in order to improve standard recurrent neural networks (RNN), enhancing their ability to learn algorithmic tasks. The architecture involves a neural controller, represented by an LSTM or an MLP, that operates over multiple time steps. A differentiable memory is also introduced. The controller receives input and produces an output by reading and writing to the differentiable memory, making the process end-to-end differentiable. Good generalization is observed, however, the target programs' difficulty is limited. What is more, a new model has to be trained for each program, as the program is contained within the learned weights.

Alternatively, Neelakantan et al. (2015) presents an architecture, capable of working on multiple programs. It includes a programmer - an RNN which processes a string database query and its last hidden state is considered to be a latent program. Using this program, the model performs multiple operations in order to retrieve the requested database information. To facilitate this, the neural networks have been augmented with arithmetic and logic operators. However, the model is evaluated only on simple queries and the authors report problems during training, which they addressed by injecting Gaussian noise to the gradients. Moreover, although the authors claim that the Neural Programmer can be applied to different domains, data sources and languages, each new problem would require a separate model to be trained.

On the other hand, the Neural Programmer Interpreter (Reed and De Freitas, 2015) is able to re-use weights across tasks. In this paper, programs are also represented as latent encodings and are stored as the value of a key-value persistent mapping. They can also interact with an environment, representing a more realistic scenario. The main component of an NPI is the NPI core, implemented as an LSTM, which processes an embedding of the environment, the program embedding, and the program arguments, and executes the program. An important feature of the core is that it's reusable for different problems. Therefore, after it is trained, new programs can be learned and executed using the same NPI core. The programs' embeddings are induced using samples from the target tasks. One shortcoming is that this method requires rich supervision, necessitating that the whole program execution trace is available for each input-output pair. On the other hand, this is shown to help achieve good generalisability. Furthermore, the NPI core works on program and environment embeddings, which allows the NPI to learn diverse tasks by having different parameters for embedding each environment. Catastrophic forgetting is still an issue in this architecture. To avoid it, when learning a new program, the authors concurrently train on previous programs as well. Therefore, it is assumed that datasets for multiple problems are available simultaneously, which is not desirable for a lifelong learning solution.

The alternative to learning a program using gradient descent is through symbolic program synthesis (Gulwani et al., 2017). These two approaches are compared in Gaunt et al. (2016b), which demonstrates that symbolic program synthesis outperforms inferring programs through gradient descent. Still, neural networks can assist symbolic methods. Balog et al. (2016) uses neural networks to guide the search of Inductive Program Synthesis (IPS) methods. The authors design a SQL-inspired domain-specific language, consisting of low-level functions (e.g. Head, Maximum) and high-level functions (e.g. Map, Filter, Scan). They then train a DNN model to predict which functions are likely to be used in a target program, by looking at the program's input-output pairs. For this purpose, a dataset of input-output-program triples is created. Empirical results show that this approach speeds up the baseline IPS methods.

3.4 Neural Architecture Search

Deep learning has alleviated the need for a domain expert to design input features for different problems. Since the input transformation is learned from data, performance can be improved by modifying the architecture of a deep neural network. A neural architecture can be changed in order to bias the hypothesis space, e.g. convolutional neural networks represent hypotheses which apply the same transformations to every pixel in a given image. Moreover, a neural architecture can be changed in order to improve optimisation. For example, ResNet (He et al., 2016) introduces skip-connections in order to make it easier to train deep models with a high number of layers. Neural architecture search (NAS) aims to automate the process of discovering a neural architecture which achieves higher performance on a given problem. The process involves the evaluation of multiple architectures, which is costly. Therefore, some NAS research also focuses on reducing the computational demand. Here, NAS work is organised following the structure in Elsken et al. (2019). For more insightful surveys, the reader is referred to Wistuba et al. (2019) and Ren et al. (2020).

NAS methods have a common structure, which includes a search space, a search strategy and a performance estimation strategy. Firstly, a NAS method needs to define a *search space* which specifies a set of possible architectures. The search space captures the ranges of different properties of a neural architecture. As a simple example, for a fully-connected architecture, the search space could entail the number of hidden layers and the number of hidden units per layer. The bigger the search space, the fewer aspects of a neural architecture are manually specified, however, the harder the search. Secondly, a NAS method needs to specify a *search strategy*. The search strategy describes the order in which architectures are evaluated. For example, architectures can be selected at random from the search space. Ideally, the search strategy should decrease the time taken to find the best architecture. Thirdly, NAS needs to estimate the performance of each selected architecture on a given dataset. The simplest *performance estimation strategy* is to train a new deep model, described by an architecture, until convergence. NAS methods can run until reaching a pre-specified computational limit. Afterwards, the architecture with the highest performance, from those considered, is returned and used for the given problem. Next, I describe different approaches to designing each of the three components of a NAS method: search space, search strategy and performance estimation strategy.

3.4.1 Search Space

The search space describes the set of architectures, which a NAS model operates on. A *global search space* describes aspects of the whole neural architecture. For example, for convolutional neural networks, a search space can describe the number of layers, the type of each layer (e.g. convolutional, pooling), the hyperparameters of each layer (e.g. kernel size, stride) and the connectivity between layers (whether there are any skip connections). An expressive global search space captures a large set of architectures but also makes it more difficult to find high performing architectures. Moreover, it is easier to find an architecture that overfits on the target problem and does not generalise to new problems. To address these concerns, different works have introduced a modular approach. A neural architecture is divided into modules, referred to as cells, which are stacked to get the final result. The cells share hyper-parameters but have different parameters. Thus, the search space is constrained to describing an individual cell. For example, Zoph et al. (2018) describe a CNN in terms of two alternating types of cells: a normal cell, which preserves the dimension of its input, and a reduction cell, which reduces the input dimension. This modular approach reduces the search space. Moreover, the discovered cells can be reused on harder problems by increasing the number of cells added in an architecture. This makes the solutions in the modular search space potentially more reusable across problems, as shown in Zoph et al. (2018).

3.4.2 Search Strategy

With the search space defined, a NAS needs to select a search strategy, used to search for a high performing neural architecture. This is challenging because the search problem is non-continuous and relatively high dimensional (Elsken et al., 2019). Methods have addressed this using one of the following: random search (RS), reinforcement learning (RL), evolutionary algorithms (EA), Bayesian optimisation (BO) and continuous search (CS). *Random search* constitutes randomly selecting architectures from the search space for evaluation. Despite its simplicity, it is a strong baseline. Li and Talwalkar (2020) report that random search, combined with early stopping, performs at least as well as a more sophisticated approach to NAS (Pham et al., 2018). In *reinforcement learning*, an agent is trained to perform a sequence of actions in order to maximise a reward. RL can be applied to NAS by letting the agent select actions, which gradually define a neural architecture. *Evolutionary algorithms* maintain a population of samples from the search space. At each evolutionary step, at least one sample, referred to as a parent, is selected for reproduction, based on performance. If there are two parents, their properties are mixed in order to produce "offspring" samples. Moreover, each offspring sample exhibits mutations, which further modifies the sample's properties at random. This way, with each evolutionary step, the sample population should contain samples with higher average performance. Evolutionary NAS maintain a population of neural architectures, which are then evolved in search of ones with higher performance. Real et al. (2019) compare their evolutionary NAS method to RL and RS baselines on the CIFAR10 dataset. They find that the evolutionary method and the RL baseline converge to similar performance, however, the evolutionary method discovers more accurate architectures earlier, indicated that it is the preferable method if one has limited computational resources. Moreover, the authors show that RS's performance is only slightly lower, achieving around 1.5% lower accuracy. *Bayesian optimisation* usually uses Gaussian Processes (GPs) and is applied to low-dimensional continuous problems (Elsken et al., 2019).

In order to make use of GPs, Kandasamy et al. (2018) develop a kernel function, capturing the similarity between two neural architectures. They describe two concepts: *layer mass*, representing the amount of computation carried out by a layer, and *path length* between layers. These concepts are used to create a distance between architectures, which is in turn used to define a kernel. One could also use a different model for the surrogate function. For instance, Zela et al. (2018) use tree-structured Parzen estimator (TPE) (Bergstra et al., 2011) to propose neural architectures to evaluate. Alternatively, Liu et al. (2018a) propose to use an ensemble of 5 LSTMs, trained on different 4/5 subsets of the evaluations, as a surrogate. They explore the search space in an order of increasing difficulty, thus exploring simpler neural architectures first. This way the search can focus on only exploring the more complex architectures, which are an extension of the more successful simpler architectures. The authors use the mean of the surrogate ensemble, without taking the variance into account. Therefore, they categorise their approach as sequential model-based optimisation. *Continuous optimisation* refers to methods which relax the search space to make it continuous and use gradient descent to optimise the architecture. For example, DARTS (Liu et al., 2018b) uses a softmax, parameterised by hyper-parameters $\boldsymbol{\alpha}$, to select among different architecture choices. The neural network parameters are trained concurrently to optimise the architecture using gradient descent. The gradients of the neural network parameters are calculated using the loss on the training set, while the gradients of the hyper-parameters $\boldsymbol{\alpha}$ also use the loss calculated on the validation set.

3.4.3 Performance Estimation Strategy

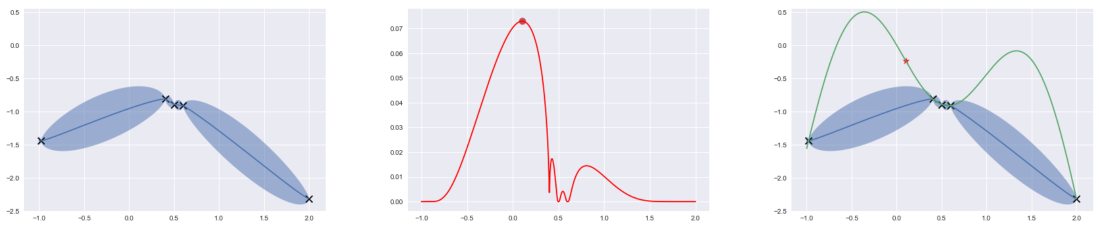
Having selected a strategy that proposes neural architectures, a NAS method needs to score each of them. A simple solution is to train a different set of parameters for each architecture and evaluate the resulting model on the validation dataset. However, this is too time-consuming, especially for big datasets. Therefore, different ideas have been explored in order to estimate an architecture's performance on the validation dataset, without training from random initialisation. One idea is to speed up the time it takes to train each new model by only training on a subset of the training dataset, or by reducing the number of training epochs. Another idea is to perform fewer training epochs and use an additional surrogate model to predict the final performance (Baker et al., 2017). Alternatively, one can reuse the weights from previous models, using them instead of random initialisation when evaluating a new architecture (Pham et al., 2018). Even though this is a widely used approach, there is evidence that it can result in the incorrect ranking of the proposed neural architectures (Yu et al., 2019). Another approach is to generate the trained weights of a neural network, conditioned on its architecture. Brock et al. (2017) present SMASH, which trains a Hypernetwork (Ha et al., 2016) to predict the trained parameters of a neural network, conditioned on its architecture. Once a hypernetwork is trained, it is used to cheaply evaluate architectures sampled at random.

3.5 Black-box Optimisation

Many problems can be described as searching for the arguments which maximise the output of a target function. In parametric machine learning models, one optimises a loss function of the training dataset, where the arguments are the model's parameters. In this case, gradient descent can be used in order to optimise this function's arguments. However, for other functions of interest, one can only evaluate their output for some arguments, but cannot evaluate the function's gradient with respect to the arguments. Optimising such a function is referred to as *black-box optimisation*. This section is concerned with the setting where it is costly to evaluate the target function. In this case, one needs to optimise the target function with as few evaluations as possible.

3.5.1 Sequential Model-based Optimisaition

When performing black-box optimisation, one can approximate the target function with a machine learning model, referred to as a *surrogate model*. This model is quick to evaluate, thus can be used to speed up the optimisation. Moreover, one can choose a differentiable surrogate model, which allows one to take advantage of gradient-based



(a) The surrogate model (b) The acquisition function (c) The target function

Figure 3.1: An illustration of the components of Bayesian optimisation.

optimisation techniques. Sequential model-based optimisation (Hutter et al., 2011) describes one way of how to use a surrogate model for black-box optimisation. Using this approach, one alternates between training a surrogate model on the current observations and using the model to propose new arguments to evaluate. Most commonly, the surrogate model is designed to also output uncertainty about its predictions. In this case, the optimisation procedure is referred to as Bayesian optimisation.

3.5.2 Bayesian Optimisation

Bayesian optimisation (BO) is a model-based sequential optimisation approach for black-box optimisation. Bayesian optimisation has been successfully applied to multiple areas. This includes machine learning hyperparameter optimisation, where BO is used to tune the hyperparamets of a machine learning algorithm. Moreover, BO has been used for neural architecture search, where the architecture of a deep learning model is optimised. In both of these cases, each evaluation of the target function is costly, as it involves training the parameters of a deep model. Next, I describe Bayesian optimisation in more detail, however, the reader is referred to Shahriari et al. (2015) for a comprehensive review.

Bayesian optimisation aims to find the optimal values $\mathbf{x}^*$ for the arguments of a given function f, s.t.

$$\mathbf{x}^* = \operatorname*{argmax}_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}) \quad ,$$

where $\mathcal{X}$ is the arguments' domain. It is assumed that it is possible to evaluate f at a proposed location $\mathbf{x}^{(i)}$ and obtain a noisy unbiased evaluation $y^{(i)} = f(\mathbf{x}^{(i)}) + \varepsilon$, where ε is the noise, usually assumed to have a 0-centered normal distribution. This allows one to model the function's output distribution $p(y^{(i)}|\varphi(\mathbf{x}^{(i)}))$ using a surrogate model. The mapping φ is used to underscore that the inputs might be embedded differently before being provided to the surrogate model. For example, a discrete variable might need to be one-hot encoded. This mapping is defined by an expert and can be specific to the choice of surrogate model.

Bayesian optimisation starts with R randomly selected arguments, which constitute the initial training dataset for the surrogate model h. Afterwards, the following four steps are repeated. First, the surrogate model is updated using the training dataset. For example, Figure 3.1a shows a Gaussian process which is fit on 5 observations. Note that the mean prediction of the surrogate model peaks near the already explored values in the middle. Therefore, a surrogate model needs to also output the uncertainty of its prediction, in order to provide information about other potentially high argument values. Second, Bayesian optimisation selects new arguments for evaluation using the surrogate model. To do this, one needs to specify an *acquisition function*, which combines the predicted mean and the predicted uncertainty into a measurement of the given argument's value. Acquisition functions are designed to balance exploration and exploitation. Exploration occurs when Bayesian optimisation selects arguments, which its surrogate model is uncertain about. Exploitation occurs when the argument values selected are close to already explored well-performing values. Figure 3.1b plots an acquisition function, namely expected improvement, over the argument space. The new arguments, selected for evaluation, are the ones that maximise the acquisition function. They are found by a search procedure, enabled by the cheap evaluation of the surrogate model and its optional gradient. Third, the Bayesian optimisation algorithm evaluates the selected arguments using the target function and returns the noisy prediction. Fourth, the selected arguments and the noisy prediction are added to the training dataset. In summary, Bayesian optimisation has the following components: the arguments domain, a surrogate model and an acquisition function.

The arguments domain is specific to the target function, however, it can also be used to guide the choice of the surrogate model. Some important considerations are the dimensionality of the arguments and whether or not they contain categorical variables. The choice of a surrogate model can impact the efficiency of Bayesian optimisation. One choice is to use a Gaussian process (GP), which is a commonly used nonparametric model (Rasmussen and Williams, 2006). A GP uses a kernel function, which captures the similarity between two inputs. Therefore, the choice of a kernel function is related to the input space and affects a GP's performance. Despite their wide use, GPs have multiple disadvantages (Falkner et al., 2018). Classical GPs have a cubic computational complexity $O(N^3)$ with respect to the size of the training dataset N. They require careful kernel customisation when applying it to complex argument spaces. Finally, GPs require hyper-parameter tuning. Other options for a surrogate model include random forests (Breiman, 2001; Hutter et al., 2011) and TPE (Bergstra et al., 2011). Common choices for a surrogate function are expected improvement (EI) and upper confidence bound (UCB). Let $\mathbf{x}$ be some arguments values and ν $h(\mathbf{x})$ a random variable distributed according to the distribution predicted by the surrogate model. Then EI (Mockus et al., 1978) captures how much the new function value is likely to increase the previously best observed function value τ:

$$EI(\mathbf{x}) = \mathbb{E}[(\nu - \tau)\mathbb{1}(\nu > \tau)] \quad ,$$

where $\mathbb{1}$ is 1 if the condition is true, and 0 otherwise. Let the predicted distribution of the surrogate model be Gaussian with mean μ and variance σ^2, i.e. $\mu, \sigma = h(\mathbf{x})$. UCB is an optimistic acquisition function (Shahriari et al., 2015), which sums the predicted mean and standard deviation:

$$UCB(\mathbf{x}) = \mu + \beta\sigma \quad ,$$

where β is a hyperparameter. If one seeks to minimise the target function, one would use the lower confidence bound (LCB), defined as

$$LCB(\mathbf{x}) = \mu - \beta\sigma \quad .$$

Bayesian optimisation approaches use machine learning models to approximate the target function. As a result, their good performance relies on the different assumptions made about the target function, such as the smoothness assumption. On the other hand, random search does not make any assumptions and randomly selects argument values for evaluation. This makes random search a strong baseline. Li et al. (2017) introduce *random 2x* which is random search with twice as many resources as Bayesian optimisation, i.e. run for twice as long. They show that *random 2x* performs competitively to the Bayesian optimisation methods on the selected data sets. This suggests that the efficiency increase provided by Bayesian optimisation may not necessarily be too big, compared to random search. What is more, when the arguments are high-dimensional, Bayesian optimisation performs similarly to random search (Wang et al., 2013). Therefore, successfully applying Bayesian optimisation requires careful consideration of the target function and the choice of a surrogate model.

Chapter 4

HOUDINI: Lifelong Learning as Program Synthesis

This chapter introduces HOUDINI- a neurosymbolic framework for solving supervised machine learning problems. For each problem, the framework employs a symbolic search procedure in order to find a problem-specific neural architecture which can achieve high generalisation performance. This allows it to operate on problems with different input spaces. We show that HOUDINI is readily applicable to the lifelong machine learning (LML) setting. Our results demonstrate that our framework provides multiple advantages over other LML approaches.

In contrast to preceding work in LML, we distinguish between different types of forward transfer, namely, perceptual, non-perceptual and few-shot transfer. *Perceptual transfer* shares the knowledge of how to process the inputs between two problems with similar input domains. For example, this can involve transferring the knowledge of extracting low-level features, like edges, from natural images. Conversely, non-perceptual transfer involves sharing knowledge between problems with disparate input domains but similar tasks. For example, transferring sentiment classification knowledge across different languages (Zhou et al., 2019). Finally, few-shot transfer involves solving a new problem which is represented by only a few training data points by reusing previous knowledge. For instance, if an algorithm can perform sentiment classification of reviews in English, and knows how to translate from French to English, it should be able to combine this knowledge in a novel way and perform sentiment classification of French sentences using only a few training examples.

Current LML approaches have several shortcomings. First, they employ the same or a similar neural architecture for all problems. For example, they use either the multi-head architecture (Lopez-Paz and Ranzato, 2017; Titsias et al., 2019; Nguyen et al., 2018) or an architecture with a preset depth and preset types of hidden layers (Rusu et al., 2016; Hung et al., 2019; Veniat et al., 2020). As a result, these approaches cannot be applied to a sequence of problems, which require different neural architectures. This prevents LML algorithms from being able to simultaneously solve problems from different input domains, such as natural language processing and image processing. Second, most of the current methods focus on avoiding catastrophic forgetting (Díaz-Rodríguez et al., 2018; Delange et al., 2021). Therefore, they do not address the challenges of avoiding negative transfer and achieving backward transfer. Furthermore, current methods do not allow non-perceptual transfer, in which only higher layers are transferred, allowing the model to learn new parameters to process a new input domain. Non-perceptual transfer is important, as it allows the reuse of abstract task-specific knowledge across disparate input domains.

Our framework can be used to address these shortcomings. HOUDINI represents deep neural networks as typed functional programs. The functions are implemented by neural modules, and different modules can process inputs of different types. Each functional program can contain a composition of modules and higher-order functions, e.g. *map*. This allows the functional programs to operate on a wide variety of input spaces. Given a problem, HOUDINI needs to find a functional program, which has a high generalisation performance. This is achieved by using an exhaustive search over type-compatible programs. Multiple programs are evaluated by having their neural modules trained and the highest-performing program is returned as the solution. HOUDINI can be adapted to the lifelong learning setting. After finding the correct program for the first problem, its modules can be frozen and added to a library of modules. For each new problem, the programs can then either contain pre-trained or randomly initialised modules. The new modules are trained on the new problem and added to the module library. This leads to a lifelong accumulation of knowledge, which can be reused on new problems.

As a result, HOUDINI contains many desirable properties of a lifelong learning algorithm. First, our framework can find a problem-specific neural architecture. It can adjust the types and the number of neural modules, as required by the problem. For example, it can choose between using a convolutional or a recurrent module. This allows our framework to operate on different input spaces. Second, catastrophic forgetting is prevented by design, since all pre-trained parameters are frozen. Third, HOUDINI is capable of transferring both input-processing functions as well as other more abstract functions. Thus, the framework can achieve both perceptual and non-perceptual transfer. Fourth, our framework can achieve few-shot transfer if the new problem can be solved by simply combining pre-trained modules in a novel way. Finally, HOUDINI can avoid negative transfer since, for each explored program, it firstly explores a standalone version of said program, which contains only randomly initialised modules.

Our neurosymbolic approach has the additional benefit of increased interpretability. We can inspect the optimal program found for a given problem. We can then examine which pre-trained modules are reused and check when said modules were originally trained. This allows us to better understand the knowledge transfer that occurs between problems.

Next, I present our paper, which was published at the *Thirty-second Annual Conference on Neural Information Processing Systems* (NeurIPS 2018). Afterwards, I conclude this chapter with a broader discussion. Note that the paper refers to perceptual transfer as *low-level transfer*, and to non-perceptual transfer as *high-level transfer*, reflecting the fact that these transfers can be achieved by reusing the first (low) and the later (high) layers respectively.

HOUDINI: Lifelong Learning as Program Synthesis

Lazar Valkov
University of Edinburgh
L.Valkov@sms.ed.ac.uk

Dipak Chaudhari
Rice University
dipakc@rice.edu

Akash Srivastava
University of Edinburgh
Akash.Srivastava@ed.ac.uk

Charles Sutton
University of Edinburgh,
The Alan Turing Institute, and Google Brain
charlessutton@google.com

Swarat Chaudhuri
Rice University
swarat@rice.edu

Abstract

We present a *neurosymbolic framework* for the lifelong learning of algorithmic tasks that mix perception and procedural reasoning. Reusing high-level concepts across domains and learning complex procedures are key challenges in lifelong learning. We show that a *program synthesis* approach that combines gradient descent with combinatorial search over programs can be a more effective response to these challenges than purely neural methods. Our framework, called HOUDINI, represents neural networks as strongly typed, differentiable functional programs that use symbolic higher-order combinators to compose a library of neural functions. Our learning algorithm consists of: (1) a symbolic program synthesizer that performs a type-directed search over parameterized programs, and decides on the library functions to reuse, and the architectures to combine them, while learning a sequence of tasks; and (2) a neural module that trains these programs using stochastic gradient descent. We evaluate HOUDINI on three benchmarks that combine perception with the algorithmic tasks of counting, summing, and shortest-path computation. Our experiments show that HOUDINI transfers high-level concepts more effectively than traditional transfer learning and progressive neural networks, and that the typed representation of networks significantly accelerates the search.

1 Introduction

Differentiable programming languages [25, 29, 8, 15, 10, 39, 34] have recently emerged as a powerful approach to the task of engineering deep learning systems. These languages are syntactically similar to, and often direct extensions of, traditional programming languages. However, programs in these languages are differentiable in their parameters, permitting gradient-based parameter learning. The framework of differentiable languages has proven especially powerful for representing architectures that have input-dependent structure, such as deep networks over trees [35, 2] and graphs [23, 19].

A recent paper by Gaunt et al. [14] points to another key appeal of high-level differentiable languages: they facilitate *transfer* across learning tasks. The paper gives a language called NEURAL TERPRET (NTPT) in which programs can invoke a library of trainable neural networks as subroutines. The parameters of these "library functions" are learned along with the code that calls them. The modularity of the language allows knowledge transfer, as a library function trained on a task can be reused in later tasks. In contrast to standard methods for transfer learning in deep networks, which re-use the first few layers of the network, neural libraries have the potential to enable reuse of higher, more abstract levels of the network, in what could be called *high-level transfer*. In spite of its promise, though, inferring the structure of differentiable programs is a fundamentally hard problem. While

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

NTPT and its predecessor TERPRET [15] allow some aspects of the program structure to be induced, a detailed hand-written template of the program is required for even the simplest tasks.

In this paper, we show that algorithmic ideas from *program synthesis* can help overcome this difficulty. The goal in program synthesis [3, 36, 13] is to discover programs (represented as terms following a specified syntax) that accomplish a given task. Many symbolic algorithms for the problem have been proposed in the recent past [16]. These algorithms can often outperform purely neural approaches on procedural tasks [15]. A key insight behind our approach is that these methods naturally complement stochastic gradient descent (SGD) in the induction of differentiable programs: while SGD is an effective way of learning program parameters, each step in a symbolic search can explore large changes to the program structure.

A second feature of our work is a representation of programs in a *typed functional language*. Such a representation is natural because functional combinators can compactly describe many common neural architectures [26]. For example, `fold` combinators can describe recurrent neural networks, and convolution over data structures such as lists and graphs can also be naturally expressed as functional combinators. Such representations also facilitate search, as they tend to be more canonical, and as the type system can help prune infeasible programs early on in the search [13, 27].

Concretely, we present a *neurosymbolic* learning framework, called HOUDINI, which is to our knowledge the first method to use symbolic methods to induce differentiable programs. In HOUDINI, a program synthesizer is used to search over networks described as strongly typed functional programs, whose parameters are then tuned end-to-end using gradient descent. Programs in HOUDINI specify the architecture of the network, by using functional combinators to express the network's connections, and can also facilitate learning transfer, by letting the synthesizer choose among library functions. HOUDINI is flexible about how the program synthesizer is implemented: here, we use and compare two implementations, one performing top-down, type-directed enumeration with a preference for simpler programs, and the other based on a type-directed evolutionary algorithm. The implementation for HOUDINI is available online [1].

We evaluate HOUDINI in the setting of *lifelong learning* [38], in which a model is trained on a series of tasks, and each training round is expected to benefit from previous rounds of learning. Two challenges in lifelong learning are *catastrophic forgetting*, in which later tasks overwrite what has been learned from earlier tasks, and *negative transfer*, in which attempting to use information from earlier tasks hurts performance on later tasks. Our use of a neural library avoids both problems, as the library allows us to freeze and selectively re-use portions of networks that have been successful.

Our experimental benchmarks combine perception with algorithmic tasks such as counting, summing, and shortest-path computation. Our programming language allows parsimonious representation for such tasks, as the combinators used to describe network structure can also be used to compactly express rich algorithmic operations. Our experiments show that HOUDINI can learn nontrivial programs for these tasks. For example, on a task of computing least-cost paths in a grid of images, HOUDINI discovers an algorithm that has the structure of the Bellman-Ford shortest path algorithm [7], but uses a learned neural function that approximates the algorithm's "relaxation" step. Second, our results indicate that the modular representation used in HOUDINI allows it to transfer high-level concepts and avoid negative transfer. We demonstrate that HOUDINI offers greater transfer than progressive neural networks [32] and traditional "low-level" transfer [40], in which early network layers are inherited from previous tasks. Third, we show that the use of a higher-level, typed language is critical to scaling the search for programs.

The contributions of this paper are threefold. First, we propose the use of symbolic program synthesis in transfer and lifelong learning. Second, we introduce a specific representation of neural networks as typed functional programs, whose types contain rich information such as tensor dimensions, and show how to leverage this representation in program synthesis. Third, we show that the modularity inherent in typed functional programming allows the method to transfer high-level modules, to avoid negative transfer and to achieve high performance with a small number of examples.

Related Work. HOUDINI builds on a known insight from program synthesis [16]: that functional representations and type-based pruning can be used to accelerate search over programs [13, 27, 20]. However, most prior efforts on program synthesis are purely symbolic and driven by the Boolean goals. HOUDINI repurposes these methods for an optimization setting, coupling them with gradient-based learning. A few recent approaches to program synthesis do combine symbolic and neural

Types τ:

$$\begin{array}{llllll} \tau & ::= & Atom \mid ADT \mid F & Atom & ::= & \texttt{bool} \mid \texttt{real} \\ TT & ::= & \texttt{Tensor}\langle Atom\rangle[m_1][m_2]\dots[m_k] & ADT & ::= & TT \mid \alpha\langle TT\rangle \\ F & ::= & ADT \mid F_1 \to F_2 & & . & \end{array}$$

Programs e over library $\mathcal{L}$: $\quad e \quad ::= \oplus_w : \tau_0 \mid e_0 \circ e_1 \mid \mathbf{map}_\alpha\ e \mid \mathbf{fold}_\alpha\ e \mid \mathbf{conv}_\alpha\ e.$

Figure 1: Syntax of the HOUDINI language. Here, α is an ADT, e.g., `list`; $m_1, \dots, m_k \geq 1$ define the shape of a tensor; $F_1 \to F_2$ is a function type; $\oplus_w \in \mathcal{L}$ is a neural library function that has type τ_0 and parameters w; and $\circ$ is the composition operator. $\mathbf{map}$, $\mathbf{fold}$, and $\mathbf{conv}$ are defined below.

methods [11, 6, 12, 28, 18]. Unlike our work, these methods do not synthesize differentiable programs. The only exception is NTPT [14], which neither considers a functional language nor a neurosymbolic search. Another recent method that creates a neural library is progress-and-compress [33], but this method is restricted to feedforward networks and low-level transfer. It is based on progressive networks [32], a method for lifelong learning based on low-level transfer, in which lateral connections are added between the networks for the all the previous tasks and the new task.

Neural module networks (NMNs) [4, 17] select and arrange reusable neural modules into a program-like network, for visual question answering. The major difference from our work is that NMNs require a natural language input to guide decisions about which modules to combine. HOUDINI works without this additional supervision. Also, HOUDINI can be seen to perform a form of *neural architecture search.* Such search has been studied extensively, using approaches such as reinforcement learning, evolutionary computation, and best-first search [42, 24, 31, 41]. HOUDINI operates at a higher level of abstraction, combining entire networks that have been trained previously, rather than optimizing over lower-level decisions such as the width of convolutional filters, the details of the gating mechanism, and so on. HOUDINI is distinct in its use of functional programming to represent architectures compactly and abstractly, and in its extensive use of types in accelerating search.

2 The HOUDINI Programming Language

HOUDINI consists of two components. The first is a typed, higher-order, functional language of differentiable programs. The second is a learning procedure split into a symbolic module and a neural module. Given a task specified by a set of training examples, the symbolic module enumerates parameterized programs in the HOUDINI language. The neural module uses gradient descent to find optimal parameters for synthesized programs; it also assesses the quality of solutions and decides whether an adequately performant solution has been discovered.

The design of the language is based on three ideas:

- The ubiquitous use of *function composition* to glue together different networks.
- The use of *higher-order combinators* such as $\mathbf{map}$ and $\mathbf{fold}$ to uniformly represent neural architectures as well as patterns of recursion in procedural tasks.
- The use of a strong *type discipline* to distinguish between neural computations over different forms of data, and to avoid generating provably incorrect programs during symbolic exploration.

Figure 1 shows the grammar for the HOUDINI language. Here, τ denotes types and e denotes programs. Now we elaborate on the various language constructs.

Types. The "atomic" data types in HOUDINI are booleans (`bool`) and reals. For us, `bool` is relaxed into a real value in $[0, 1]$, which for example, allows the type system to track if a vector has been passed through a sigmoid. *Tensors* over these types are also permitted. We have a distinct type $\texttt{Tensor}\langle Atom\rangle[m_1][m_2]\dots[m_k]$ for tensors of shape $m_1 \times \cdots \times m_k$ whose elements have atomic type $Atom$. (The dimensions $m_1, \dots, m_k$, as well as k itself, are bounded to keep the set of types finite.) We also have function types $F_1 \to F_2$, and abstract data types (ADTs) $\alpha\langle TT\rangle$ parameterized by a tensor type TT. Our current implementation supports two kinds of ADTs: $\texttt{list}\langle TT\rangle$, lists with elements of tensor type TT, and $\texttt{graph}\langle TT\rangle$, graphs whose nodes have values of tensor type TT.

Programs. The fundamental operation in HOUDINI is *function composition.* A composition operation can involve functions $\oplus_w$, parameterized by weights w and implemented by neural networks, drawn from a library $\mathcal{L}$. It can also involve a set of symbolic higher-order combinators that are guaranteed to preserve end-to-end differentiability and used to implement high-level network architectures. Specifically, we allow the following three families of combinators. The first two are standard constructs for functional languages, whereas the third is introduced specifically for deep models.

- Map combinators $\mathbf{map}_{\alpha\langle\tau\rangle}$, for ADTs of the form $\alpha\langle\tau\rangle$. Suppose e is a function. The expression $\mathbf{map}_{\mathtt{list}\langle\tau\rangle}\ e$ is a function that, given a list $[a_1, \ldots, a_k]$, returns the list $[e(a_1), \ldots, e(a_k)]$. The expression $\mathbf{map}_{\mathtt{graph}_\tau}\ e$ is a function that, given a graph G whose i-th node is labeled with a value a_i, returns a graph that is identical to G, but whose i-th node is labeled by $e(a_i)$.
- Left-fold combinators $\mathbf{fold}_{\alpha\langle\tau\rangle}$. For a function e and a term z, $\mathbf{fold}_{\mathtt{list}\langle\tau\rangle}\ e\ z$ is the function that, given a list $[a_1, \ldots, a_k]$, returns the value $(e\ (e\ \ldots (e\ (e\ z\ a_1)\ a_2) \ldots)\ a_k)$. To define fold over a graph, we assume a linear order on the graph's nodes. Given G, the function $\mathbf{fold}_{\mathtt{graph}\langle\tau\rangle}\ e\ z$ returns the fold over the list $[a_1, \ldots, a_k]$, where a_i is the value at the i-th node in this order.
- Convolution combinators $\mathbf{conv}_{\alpha\langle\tau\rangle}$. Let $p > 0$ be a fixed constant. For a "kernel" function e, $\mathbf{conv}_{\mathtt{list}\langle\tau\rangle}\ e$ is the function that, given a list $[a_1, \ldots, a_k]$, returns the list $[a'_1, \ldots, a'_k]$, where $a'_i = e\ [a_{i-p}, \ldots, a_i, \ldots, a_{i+p}]$. (We define $a_j = a_1$ if $j < 1$, and $a_j = a_k$ if $j > k$.) Given a graph G, the function $\mathbf{conv}_{\mathtt{graph}\langle\tau\rangle}\ e$ returns the graph G' whose node u contains the value $e\ [a_{i_1}, \ldots, a_{i_m}]$, where a_{i_j} is the value stored in the j-th neighbor of u.

Every neural library function is assumed to be annotated with a type. Using programming language techniques [30], HOUDINI assigns a type to each program e whose subexpressions use types consistently (see supplementary material). If it is impossible to assign a type to e, then e is *type-inconsistent*. Note that complete HOUDINI programs do not have explicit variable names. Thus, HOUDINI follows the *point-free* style of functional programming [5]. This style permits highly succinct representations of complex computations, which reduces the amount of enumeration needed during synthesis.

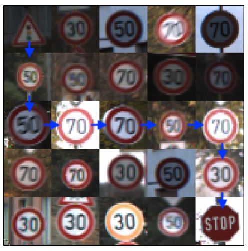


Figure 2: A grid of images from the GTSRB dataset [37]. The least-cost path from the top left to the bottom right node is marked.

HOUDINI for deep learning. The language has several properties that are useful for specifying deep models. First, any complete HOUDINI program e is differentiable in the parameters w of the neural library functions used in e. Second, common deep architectures can be compactly represented in our language. For example, deep feedforward networks can be represented by $\oplus_1 \circ \cdots \circ \oplus_k$, where each $\oplus_i$ is a neural function, and recurrent nets can be expressed as $\mathbf{fold}_{\mathtt{list}\langle\tau\rangle}\ \oplus\ z$, where $\oplus$ is a neural function and z is the initial state. Graph convolutional networks can be expressed as $\mathbf{conv}_{\mathtt{graph}\langle\tau\rangle}\ \oplus$. Going further, the language can be easily extended to handle bidirectional recurrent networks, attention mechanisms, and so on.

Example: Shortest path in a grid of images. To show how HOUDINI can model tasks that mix perception and procedural reasoning, we use an example that generalizes the navigation task of Gaunt et al. [14]. Suppose we are given a grid of images (e.g., Figure 2), whose elements represent speed limits and are connected horizontally and vertically, but not diagonally. Passing through each node induces a penalty, which depends on the node's speed limit, with lower speed limits having a higher penalty. The task is to predict the minimum cost $d(u)$ incurred while traveling from a fixed starting point *init* to every other node u.

One way to compute these costs is using the Bellman-Ford shortest-path algorithm [7], whose i-th iteration computes an estimated minimum cost $d_i(u)$ of travel to each node u in the graph. The cost estimates for the $(i + 1)$-th iteration are computed using a *relaxation* operation: $d_{i+1}(u) := \min(d_i(u), \min_{v \in Adj(u)} d_i(v) + w(v))$, where $w(v)$ is the penalty and $Adj(u)$ the neighbors of u. As the update to $d_i(u)$ only depends on values at u and its neighbors, the relaxation step can be represented as a graph convolution. As described in Section 4, HOUDINI is able to discover an approximation of this program purely from data. The synthesized program uses a graph convolution, a graph map, a neural module that processes the images of speed limits, and a neural module that approximates the relaxation function.

3 Learning Algorithm

Now we define our learning problem. For a HOUDINI program e_w parameterized by a vector w, let $e[w \mapsto v]$ be the function for the specific parameter vector v, i.e. by substituting w by v in e.

Suppose we have a library $\mathcal{L}$ of neural functions and a training set $\mathcal{D}$. As usual, we assume that $\mathcal{D}$ consists of i.i.d. samples from a distribution p_{data}. We assume that $\mathcal{D}$ is properly typed, i.e., every training instance $(x_i, y_i) \in \mathcal{D}$ has the same type, which is known. This means that we also know the type τ of our target function. The goal in our learning problem is to discover a program e_w^* of type τ, and values v for w such that $e_w^*[w \mapsto v] = \text{argmin}_{e \in Progs(\mathcal{L}), w \in \mathbb{R}^n}(\mathbf{E}_{x \sim p_{data}}[l(e, \mathcal{D}, x)])$, where $Progs(\mathcal{L})$ is the universe of all programs over $\mathcal{L}$, and l is a suitable loss function.

Our algorithm for this task consists of a symbolic program synthesis module called GENERATE and a gradient-based optimization module called TUNE. GENERATE repeatedly generates parameterized programs e_w and "proposes" them to TUNE. TUNE uses stochastic gradient descent to find parameter values v for e_w that lead to the optimal value of the loss function on a training set, and produces a program $e = e_w[w \mapsto v]$ with instantiated parameters. The final output of the algorithm is a program e^*, among all programs e as above, that leads to optimal loss on a validation set.

As each program proposed by GENERATE is subjected to training, GENERATE can only afford to propose a small number of programs, out of the vast combinatorial space of all programs. Selecting these programs is a difficult challenge. We use and compare two strategies for this task. Now we sketch these strategies; for more details, see the supplementary material.

- The first strategy is *top-down iterative refinement*, similar to the algorithm in the λ^2 program synthesizer [13]. Here, the synthesis procedure iteratively generates a series of "partial" programs (i.e., programs with missing code) over the library $\mathcal{L}$, starting with an "empty" program and ending with a complete program. A type inference procedure is used to rule out any partial program that is not type-safe. A cost heuristic is used to generate programs in an order of structural simplicity. Concretely, shorter programs are evaluated first.
- The second method is an *evolutionary algorithm* inspired by work on functional genetic programming [9]. Here, we use selection, crossover, and mutation operators to evolve a population of programs over $\mathcal{L}$. Types play a key role: all programs in the population are ensured to be type-safe, and mutation and crossover only replace a subterm in a program with terms of the same type.

In both cases, the use of types vastly reduces the amount of search that is needed, as the number of type-safe programs of a given size is a small fraction of the number of programs of that size. See Section 4 for an experimental confirmation.

Lifelong Learning. A *lifelong learning* setting is a sequence of related tasks $\mathcal{D}_1, \mathcal{D}_2, \ldots$, where each task $\mathcal{D}_i$ has its own training and validation set. Here, the learner is called repeatedly, once for each task $\mathcal{D}_i$ using a neural library $\mathcal{L}_i$, returning a best-performing program e_i^* with parameters v_i^*.

We implement transfer learning simply by adding new modules to the neural library after each call to the learner. We add all neural functions from e_i^* back into the library, freezing their parameters. More formally, let $\oplus_{i1} \ldots \oplus_{iK}$ be the neural library functions which are called anywhere in e_i^*. Each library function $\oplus_{ik}$ has parameters w_{ik}, set to the value v_{ik}^* by TUNE. The library for the next task is then $\mathcal{L}_{i+1} = \mathcal{L}_i \cup \{\oplus_{ik}[w_{ik} \mapsto v_{ik}^*]\}$. This process ensures that the parameter vectors of $\oplus_{ik}$ are frozen and can no longer be updated by subsequent tasks. Thus, we prevent catastrophic forgetting by design. Importantly, it is always possible for the synthesizer to introduce "fresh networks" whose parameters have not been pretrained. This is because the library always monotonically increases over time, so that an original neural library function with untrained parameters is still available.

This approach has the important implication that the set of neural library functions that the synthesizer uses is not fixed, but continually evolving. Because both trained and untrained versions of the library functions are available, this can be seen to permit *selective transfer*, meaning that depending on which version of the library function GENERATE chooses, the learner has the option of using or not using previously learned knowledge in a new task. This fact allows HOUDINI to avoid negative transfer.

4 Evaluation

Our evaluation studies four questions. First, we ask whether HOUDINI can learn nontrivial differentiable programs that combine perception and algorithmic reasoning. Second, we study if HOUDINI can transfer perceptual and algorithmic knowledge during lifelong learning. We study three forms of transfer: *low-level transfer* of perceptual concepts across domains, *high-level transfer* of algorithmic concepts, and *selective transfer* where the learning method decides on which known concepts to

Individual tasks

recognize_digit(d): Binary classification of whether image contains a digit $d \in \{0 \ldots 9\}$

classify_digit: Classify a digit into digit categories $(0 - 9)$

recognize_toy(t): Binary classification of whether an image contains a toy $t \in \{0 \ldots 4\}$

regress_speed: Return the speed value and a maximum distance constant from an image of a speed limit sign.

regress_mnist: Return the value and a maximum distance constant from a digit image from MNIST dataset.

count_digit(d): Given a list of images, count the number of images of digit d

count_toy(t): Given a list of images, count the number of images of toy t

sum_digits: Given a list of digit images, compute the sum of the digits.

shortest_path_street: Given a grid of images of speed limit signs, find the shortest distances to all other nodes

shortest_path_mnist: Given a grid of MNIST images, and a source node, find the shortest distances to all other nodes in the grid.

Task Sequences

Counting

CS1: Evaluate low-level transfer.
Task 1: recognize_digit($d1$); *Task 2*: recognize_digit($d2$); *Task 3*: count_digit($d1$); *Task 4*: count_digit($d2$)

CS2: Evaluate high-level transfer, and learning of perceptual tasks from higher-level supervision.
Task 1: recognize_digit($d1$); *Task 2*: count_digit($d1$); *Task 3*: count_digit($d2$); *Task 4*: recognize_digit($d2$)

CS3: Evaluate high-level transfer of counting across different image domains.
Task 1: recognize_digit(d); *Task 2*: count_digit(d); *Task 3*: count_toy(t); *Task 4*: recognize_toy(t)

Summing

SS: Demonstrate low-level transfer of a multi-class classifier as well as the advantage of functional methods like foldl in specific situations.

Task 1: classify_digit; *Task 2*: sum_digits

Single-Source Shortest Path

GS1: Learning of complex algorithms.

Task 1: regress_speed; *Task 2*: shortest_path_street

GS2: High-level transfer of complex algorithms.
Task 1: regress_mnist; *Task 2*: shortest_path_mnist; *Task 3:* shortest_path_street

Long sequence **LS**.

Task 1:count_digit($d1$); *Task 2*: count_toy($t1$); *Task 3*: recognize_toy($t2$); *Task 4*: recognize_digit($d2$); *Task 5*: count_toy($t3$); *Task 6*: count_digit($d3$); *Task 7*: count_toy($t4$); *Task 8*: recognize_digit($d4$); *Task 9*: count_digit($d5$)

Figure 3: Tasks and task sequences.

reuse. Third, we study the value of our type-directed approach to synthesis. Fourth, we compare the performance of the top-down and evolutionary synthesis algorithms.

Task Sequences. Each lifelong learning setting is a sequence of individual learning tasks. The full list of tasks is shown in Figure 3. These tasks include object recognition tasks over three data sets: MNIST [21], NORB [22], and the GTSRB data set of images of traffic signs [37]. In addition, we have three algorithmic tasks: *counting* the number of instances of images of a certain class in a list of images; *summing* a list of images of digits; and the *shortest path* computation described in Section 2.

We combine these tasks into seven sequences. Three of these (CS1, SS, GS1) involve low-level transfer, in which earlier tasks are perceptual tasks like recognizing digits, while later tasks introduce higher-level algorithmic problems. Three other task sequences (CS2, CS3, GS2) involve higher-level transfer, in which earlier tasks introduce a high-level concept, while later tasks require a learner to re-use this concept on different perceptual inputs. For example, in CS2, once count_digit(d_1) is learned for counting digits of class d_1, the synthesizer can learn to reuse this counting network on a new digit class d_2, even if the learning system has never seen d_2 before. The graph task sequence GS1 also demonstrates that the graph convolution combinator in HOUDINI allows learning of complex graph algorithms and GS2 tests if high-level transfer can be performed with this more complex task. Finally, we include a task sequence LS that is designed to evaluate our method on a task sequence that is both longer and that lacks a favourable curriculum. The sequence LS was initially randomly generated, and then slightly amended in order to evaluate all lifelong learning concepts discussed.

Experimental setup. We allow three kinds of neural library modules: multi-layer perceptrons (MLPs), convolutional neural networks (CNNs) and recurrent neural networks (RNNs). We use two symbolic synthesis strategies: top-down refinement and evolutionary. We use three types of baselines: (1) *standalone networks*, which do not do transfer learning, but simply train a new network (an RNN) for each task in the sequence, starting from random weights; (2) a traditional neural approach to *low-level transfer* (LLT) that transfers all weights learned in the previous task, except for the output layer that is kept task-specific; and (3) a version of the *progressive neural networks* (PNNs) [32]

Task	Top 3 programs	RMSE
Task 1: regress_mnist	1. nn_gs2_1 $\circ$ nn_gs2_2	1.47
Task 2: shortest_path_mnist	1. $(\mathbf{conv\_g}^{10}$ (nn_gs2_3)) $\circ$ ($\mathbf{map\_g}$ (lib.nn_gs2_1 $\circ$ lib.nn_gs2_2))	1.57
	2. $(\mathbf{conv\_g}^{9}$ (nn_gs2_4)) $\circ$ ($\mathbf{map\_g}$ (lib.nn_gs2_1 $\circ$ lib.nn_gs2_2))	1.72
	3. $(\mathbf{conv\_g}^{9}$ (nn_gs2_5)) $\circ$ ($\mathbf{map\_g}$ (nn_gs2_6 $\circ$ nn_gs2_7))	4.99
Task 3: shortest_path_street	1. $(\mathbf{conv\_g}^{10}$(lib.nn_gs2_3)) $\circ$ ($\mathbf{map\_g}$ (nn_gs2_8 $\circ$ nn_gs2_9))	3.48
	2. $(\mathbf{conv\_g}^{9}$(lib.nn_gs2_3)) $\circ$ ($\mathbf{map\_g}$ (nn_gs2_10 $\circ$ nn_gs2_11))	3.84
	3. $(\mathbf{conv\_g}^{10}$(lib.nn_gs2_3)) $\circ$ ($\mathbf{map\_g}$ (lib.nn_gs2_1 $\circ$ lib.nn_gs2_2))	6.91

Figure 4: Top 3 synthesized programs on Graph Sequence 2 (GS2). $\mathbf{map\_g}$ denotes a graph map (of the appropriate type); $\mathbf{conv\_g}^i$ denotes i repeated applications of a graph convolution combinator.

approach, which retains a pool of pretrained models during training and learns lateral connections among these models. Experiments were performed using a single-threaded implementation on a Linux system, with 8-core Intel E5-2620 v4 2.10GHz CPUs and TITAN X (Pascal) GPUs.

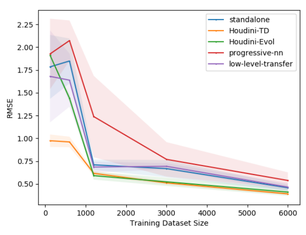
The architecture chosen for the standalone and LLT baselines composes an MLP, an RNN, and a CNN, and matches the structure of a high-performing program returned by HOUDINI, to enable an apples-to-apples comparison. In PNNs, every task in a sequence is associated with a network with the above architecture; lateral connections between these networks are learned. Each sequence involving digit classes d and toy classes t was instantiated five times for random values of d and t, and the results shown are averaged over these instantiations. In the graph sequences, we ran the same sequences with different random seeds, and shared the regressors learned for the first tasks across the competing methods for a more reliable comparison. We do not compare against PNNs in this case, as it is nontrivial to extend them to work with graphs. We evaluate the competing approaches on 2%, 10%, 20%, 50% and 100% of the training data for all but the graph sequences, where we evaluate only on 100%. For classification tasks, we report error, while for the regression tasks — counting, summing, regress_speed and shortest_path — we report root mean-squared error (RMSE).

Results: Synthesized programs. HOUDINI successfully synthesizes programs for each of the tasks in Figure 3 within at most 22 minutes. We list in Figure 4 the top 3 programs for each task in the graph sequence GS2, and the corresponding RMSEs. Here, function names with prefix "nn_" denote fresh neural modules trained during the corresponding tasks. Terms with prefix "lib." denote pretrained neural modules selected from the library. The synthesis times for Task 1, Task 2, and Task 3 are 0.35s, 1061s, and 1337s, respectively.

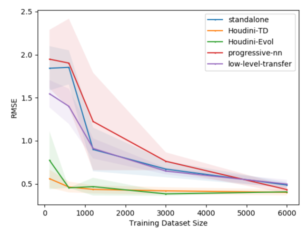
As an illustration, consider the top program for Task 3: $(\mathbf{conv\_g}^{10}$ lib.nn_gs2_3) $\circ$ ($\mathbf{map\_g}$ (nn_gs2_8 $\circ$ nn_gs2_9)). Here, $\mathbf{map\_g}$ takes as argument a function for processing the images of speed limits. Applied to the input graph, the map returns a graph G in which each node contains a number associated with its corresponding image and information about the least cost of travel to the node. The kernel for the graph convolution combinator $\mathbf{conv\_g}$ is a function lib.nn_gs2_3, originally learned in Task 2, that implements the *relaxation* operation used in shortest-path algorithms. The convolution is applied repeatedly, just like in the Bellman-Ford shortest path algorithm.

In the SS sequence, the top program for Task 2 is: ($\mathbf{fold\_l}$ nn_ss_3 zeros(1)) $\circ$ $\mathbf{map\_l}$(nn_ss_4 $\circ$ lib.nn_ss_2). Here, $\mathbf{fold\_l}$ denotes the fold operator applied to lists, and zeros(dim) is a function that returns a zero tensor of appropriate dimension. The program uses a map to apply a previously learned CNN feature extractor (lib.nn_ss_2) and a learned transformation of said features into a 2D hidden state, to all images in the input list. It then uses fold with another function (nn_ss_3) to give the final sum. Our results, presented in the supplementary material, show that this program greatly outperforms the baselines, even in the setting where all of the training data is available. We believe that this is because the synthesizer has selected a program with fewer parameters than the baseline RNN. In the results for the counting sequences (CS) and the long sequence (LS), the number of evaluated programs is restricted to 20, therefore $\mathbf{fold\_l}$ is not used within the synthesized programs. This allows us to evaluate the advantage of HOUDINI brought by its transfer capabilities, rather than its rich language.

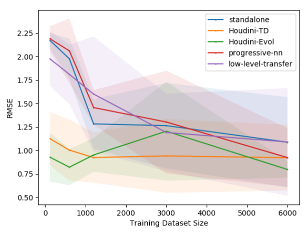
Results: Transfer. First we evaluate the performance of the methods on the counting sequences (Figure 5). For space, we omit early tasks where, by design, there is no opportunity for transfer; for these results, see the Appendix. In all cases where there is an opportunity to transfer from previous tasks, we see that HOUDINI has much lower error than any of the other transfer learning methods. The actual programs generated by HOUDINI are listed in the Appendix.



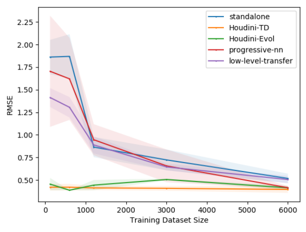
(a) CS1 Task 3: count_digit(d_1)



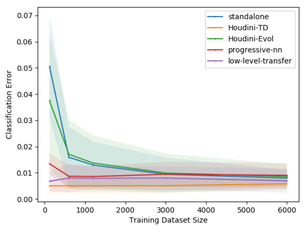
(b) CS2 Task 3: count_digit(d_2)



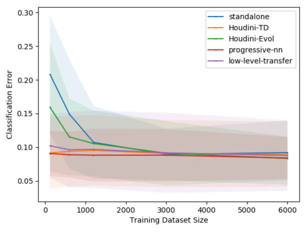
(c) CS3 Task 3: count_toy(t_1)



(d) CS1 Task 4: count_digit(d_2)



(e) CS2 Task 4: recognize_digit(d_2)



(f) CS3 Task 4: recognize_toy(t_1)

Figure 5: Lifelong "learning to count" (Sequences CS1 – CS3), demonstrating both low-level transfer of perceptual concepts and high-level transfer of a counting network. HOUDINI-TD and HOUDINI-EVOL are HOUDINI with the top-down and evolutionary synthesizers, respectively.

Task sequence CS1 evaluates the method's ability to selectively perform low-level transfer of a perceptual concept across higher level tasks. The first task that provides a transfer opportunity is CS1 task 3 (Figure 5a). There are two potential lower-level tasks that the methods could transfer from: recognize_digit(d_1) and recognize_digit(d_2). HOUDINI learns programs composed of neural modules $\mathrm{nn\_cs1\_1}$, $\mathrm{nn\_cs1\_2}$, $\mathrm{nn\_cs1\_3}$, and $\mathrm{nn\_cs1\_4}$ for these two tasks. During training for the count_digit($d1$) task, all the previously learned neural modules are available in the library. The learner, however, picks the correct module ($\mathrm{nn\_cs1\_2}$) for reuse, learning the program "$\mathrm{nn\_cs1\_7} \circ (\mathbf{map\_l}\ (\mathrm{nn\_cs1\_8} \circ \mathrm{lib.nn\_cs1\_2}))$" where $\mathrm{nn\_cs1\_7}$ and $\mathrm{nn\_cs1\_8}$ are fresh neural modules, and $\mathbf{map\_l}$ stands for a list map combinator of appropriate type. The low-level transfer baseline cannot select which of the previous tasks to re-use, and so suffers worse performance.

Task sequence CS2 provides an opportunity to transfer the higher-level concepts of counting, across different digit classification tasks. Here CS2 task 3 (Figure 5b) is the task that provides the first opportunity for transfer. We see that HOUDINI is able to learn much faster on this task because it is able to reuse a network which has learned from the previous counting task. Task sequence CS3 examines whether the methods can demonstrate high-level transfer when the input image domains are very different, from the MNIST domain to the NORB domain of toy images. We see in Figure 5c that the higher-level network still successfully transfers across tasks, learning an effective network for counting the number of toys of type t_1, even though the network has not previously seen any toy images at all. What is more, it can be seen that because of the high-level transfer, HOUDINI has learned a modular solution to this problem. From the subsequent performance on a standalone toy classification task (Figure 5f), we see that CS3 task 3 has already caused the network to induce a re-usable classifier on toys. Overall, it can be seen that HOUDINI outperforms all the baselines even under the limited data setting, confirming the successful selective transfer of both low-level and high-level perceptual information. Similar results can be seen on the summing task (see supplementary material). Moreover, on the longer task sequence LS, we also find that HOUDINI performs significantly better on the tasks in the sequence where there is an opportunity for transfer, and performs comparably the baselines on the other tasks (see supplementary material). Furthermore, on the summing sequence, our results also show low level transfer.

Finally, for the graph-based tasks (Table 2), we see that the graph convolutional program learned by HOUDINI on the graph tasks has significantly less error than a simple sequence model, a standalone

	Task	Number of programs		
		size = 4	size = 5	size = 6
No types	Task 1	8182	110372	1318972
	Task 2	12333	179049	2278113
	Task 3	17834	278318	3727358
	Task 4	24182	422619	6474938
+ Types	Task 1	2	20	44
	Task 2	5	37	67
	Task 3	9	47	158
	Task 4	9	51	175

Table 1: Effect of the type system on the number of programs considered in the symbolic search for task sequence CS1.

	Task 1	Task 2
RNN w llt	0.75	5.58
standalone	0.75	4.96
HOUDINI	0.75	1.77
HOUDINI EA	0.75	8.32
low-level-transfer	0.75	1.98

(a) Low-level transfer (llt) (task sequence GS1).

	Task 1	Task 2	Task 3
RNN w llt	1.44	5.00	6.05
standalone	1.44	6.49	7.
HOUDINI	1.44	1.50	3.31
HOUDINI EA	1.44	6.67	7.88
low-level-transfer	1.44	1.76	2.08

(b) High-level transfer (task sequence GS2).

Table 2: Lifelong learning on graphs. Col 1: RMSE on speed/distance from image. Cols 2, 3: RMSE on shortest path (mnist, street).

baseline and the evolutionary-algorithm-based version of HOUDINI. As explained earlier, in the shortest_path_street task in the graph sequence GS2, HOUDINI learns a program that uses newly learned regress functions for the street signs, along with a "relaxation" function already learned from the earlier task shortest_path_mnist. In Table 2, we see this program performs well, suggesting that a domain-general relaxation operation is being learned. Our approach also outperforms the low-level-transfer baseline, except on the shortest_path_street task in GS2. We are unable to compare directly to NTPT because no public implementation is available. However, our graph task is a more difficult version of a task from [14], who report on their shortest-path task "2% of random restarts successfully converge to a program that generalizes" (see their supplementary material).

Results: Typed vs. untyped synthesis. To assess the impact of our type system, we count the programs that GENERATE produces with and without a type system (we pick the top-down implementation for this test, but the results also apply to the evolutionary synthesizer). Let the *size* of a program be the number of occurrences of library functions and combinators in the program. Table 1 shows the number of programs of different sizes generated for the tasks in the sequence CS1. Since the typical program size in our sequences is less than 6, we vary the target program size from 4 to 6. When the type system is disabled, the only constraint that GENERATE has while composing programs is the arity of the library functions. We note that this constraint fails to bring down the number of candidate programs to a manageable size. With the type system, however, GENERATE produces far fewer candidate programs. For reference, neural architecture search often considers thousands of potential architectures for a single task [24].

Results: Top-Down vs. Evolutionary Synthesis. Overall, the top-down implementation of GENERATE outperformed the evolutionary implementation. In some tasks, the two strategies performed similarly. However, the evolutionary strategy has high variance; indeed, in many runs of the task sequences, it times out without finding a solution. The timed out runs are not included in the plots.

5 Conclusion

We have presented HOUDINI, the first neurosymbolic approach to the synthesis of differentiable functional programs. Deep networks can be naturally specified as differentiable programs, and functional programs can compactly represent popular deep architectures [26]. Therefore, symbolic search through a space of differentiable functional programs is particularly appealing, because it can at the same time select both which pretrained neural library functions should be reused, and also what deep architecture should be used to combine them. On several lifelong learning tasks that combine perceptual and algorithmic reasoning, we showed that HOUDINI can accelerate learning by transferring high-level concepts.

Acknowledgements. This work was partially supported by DARPA MUSE award #FA8750-14-2-0270 and NSF award #CCF-1704883.

References

[1] Houdini code repository. `https://github.com/capergroup/houdini`.

[2] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. Learning continuous semantic representations of symbolic expressions. In *International Conference on Machine Learning (ICML)*, 2017.

[3] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*, FMCAD, pages 1–17, 2013.

[4] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 39–48, 2016.

[5] John Backus. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[6] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR)*, 2017. arXiv:1611.01989.

[7] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[8] Matko Bosnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable Forth interpreter. In *International Conference on Machine Learning (ICML)*, pages 547–556, 2017.

[9] Forrest Briggs and Melissa O'Neill. Functional genetic programming with combinators. In *Proceedings of the Third Asian-Pacific Workshop on Genetic Programming (ASPGP)*, pages 110–127, 2006.

[10] Rudy R. Bunel, Alban Desmaison, Pawan Kumar Mudigonda, Pushmeet Kohli, and Philip H. S. Torr. Adaptive neural compilation. In *Advances in Neural Information Processing Systems 29*, pages 1444–1452, 2016.

[11] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *International Conference on Machine Learning (ICML)*, 2017.

[12] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. Learning to infer graphics programs from hand-drawn images. *CoRR*, abs/1707.09627, 2017.

[13] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–239, 2015.

[14] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *International Conference on Machine Learning (ICML)*, pages 1213–1222, 2017.

[15] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016.

[16] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[17] Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to reason: End-to-end module networks for visual question answering. In *International Conference on Computer Vision (ICCV)*, 2017. CoRR, abs/1704.05526.

[18] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *International Conference on Learning Representations (ICLR)*, 2018.

[19] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[20] Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 542–553, 2014.

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[22] Yann LeCun, Fu Jie Huang, and Leon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition (CVPR)*, 2004.

[23] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*, 2016.

[24] Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *European Conference on Computer Vision (ECCV)*, 2018. arXiv:1712.00559.

[25] Dougal Maclaurin, David Duvenaud, Matthew Johnson, and Ryan P. Adams. Autograd: Reverse-mode differentiation of native Python, 2015.

[26] Christopher Olah. Neural networks, types, and functional programming, 2015. `http://colah.github.io/posts/2015-09-NN-Types-FP/`.

[27] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, volume 50, pages 619–630. ACM, 2015.

[28] Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *International Conference on Learning Representations (ICLR)*, 2016.

[29] 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.

[30] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[31] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning (ICML)*, 2017.

[32] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[33] Jonathan Schwarz, Jelena Luketina, Wojciech M. Czarnecki, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & compress: A scalable framework for continual learning. In *International Conference on Machine Learning (ICML)*, 2018.

[34] Asim Shankar and Wolff Dobson. Eager execution: An imperative, define-by-run interface to tensorflow. `https://research.googleblog.com/2017/10/eager-execution-imperative-define-by.html`, 2017.

[35] Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

[36] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[37] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, (0), 2012.

[38] Sebastian Thrun and Tom M. Mitchell. Lifelong robot learning. *Robotics and Autonomous Systems*, 15(1-2):25–46, 1995.

[39] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of NIPS Workshop on Machine Learning Systems (LearningSys)*, 2015.

[40] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems 27 (NIPS)*, 2014.

[41] Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. Practical network blocks design with q-learning. *CoRR*, abs/1708.05552, 2017.

[42] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

Supplementary material for HOUDINI: Lifelong Learning as Program Synthesis

Lazar Valkov
University of Edinburgh
L.Valkov@sms.ed.ac.uk

Dipak Chaudhari
Rice University
dipakc@rice.edu

Akash Srivastava
University of Edinburgh
Akash.Srivastava@ed.ac.uk

Charles Sutton
University of Edinburgh,
The Alan Turing Institute, and Google Brain
charlessutton@google.com

Swarat Chaudhuri
Rice University
swarat@rice.edu

A Assigning Types to HOUDINI Programs

A program e in HOUDINI is assigned a type using the following rules:

- $e = e' \circ e''$ is assigned a type iff e' has type $\tau\tau'$ and e'' has type $\tau'\tau''$. In this case, e has type $\tau\tau''$.
- $e =_{\alpha\tau}\ e'$ is assigned a type iff e' has the type $\tau\tau'$. In this case, the type of e is $\alpha\tau\alpha\tau'$.
- $e =_{\alpha\tau}\ e'\ z$ is assigned a type iff e' has the type $\tau'(\tau\tau')$ and z has the type τ'. In this case, e has type $\alpha\tau\tau'$.
- $e =_{\alpha\tau}\ e'$ is assigned a type iff e' has the type $\texttt{list}\tau\tau'$. In this case, e has type $\alpha\tau\alpha\tau'$.

If it is not possible to assign a type to the program e, then it is considered *type-inconsistent* and excluded from the scope of synthesis.

B Symbolic Program Synthesis

In this appendix we provide implementation details of our synthesis algorithms.

B.1 Synthesis Using Top-down Iterative Refinement

Now we give more details on the implementation of based on iterative refinement. To explain this algorithm, we need to define a notion of a *partial program*. The grammar for partial programs e is obtained by augmenting the HOUDINI grammar (Figure **??**) with an additional rule: $e ::= \tau$. The form τ represents a *hole*, standing for missing code. A program with holes has no operational meaning; however, we do have a type system for such programs. This type system follows the rules in Appendix A, but in addition, axiomatically assumes any subterm τ to be of type τ. A partial program that cannot be assigned a type is automatically excluded from the scope of synthesis.

Now, the initial input to the algorithm is the type τ of the function we want to learn. The procedure proceeds iteratively, maintaining a priority queue Q of *synthesis subtasks* of the form (e, f), where e is a type-safe partial or complete program of type τ, and f is either a hole of type τ' in e, or a special symbol $\perp$ indicating that e is complete (i.e., free of holes). The interpretation of such a task is to find a replacement e' of type τ' for the hole f such that the program e'' obtained by substituting f by e' is complete. (Because e is type-safe by construction, e'' is of type τ.) The queue is sorted according to a heuristic cost function that prioritizes simpler programs.

Initially, Q has a single element (e, f), where e is an "empty" program of form τ, and f is a reference to the hole in e. The procedure iteratively processes subtasks in the queue Q, selecting a task (e, f)

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

in the beginning of each iteration. If the program e is complete, it is sent to the neural module for parameter learning. Otherwise, the algorithm expands the program e by proposing a partial program that fills the hole f. To do this, the algorithm selects a production rule for partial programs from the grammar for partial programs. Suppose the right hand side of this rule is α. The algorithm constructs an expression e' from α by replacing each nonterminal in α by a hole with the same type as the nonterminal. If e' is not of the same type as f, it is automatically rejected. Otherwise, the algorithm constructs the program $e'' = e[f \mapsto e']$. For each hole f' in e'', the algorithm adds to Q a new task (e'', f'). If e'' has no hole, it adds to Q a task $(e'', \perp)$.

B.2 Evolutionary Synthesis

The evolutionary synthesis algorithm is an iterative procedure that maintains a population of programs. The population is initialized with a set of randomly generated type-safe parameterized programs. Each iteration of the algorithm performs the following steps.

1. Each program in the population is sent to the neural module , which computes a *fitness score* (the loss under optimal parameters) for the program.
2. We perform *random proportional selection*, in which a subset of the (parameterized) programs are retained, while the other programs are filtered out. Programs with higher fitness are more likely to remain in the population.
3. We perform a series of *crossover* operations, each of which draws a random pair of programs from the population and swaps a pair of randomly drawn subterms of the same type in these programs.
4. We perform a series of *mutation* operations, each of which randomly chooses a program and replaces a random subterm the program with a new subterm of the same type.

Because the crossover and mutation operations only replace terms with other terms of the same type, the programs in the population are always guaranteed to be type-consistent. This fact is key to the performance of the algorithm.

C Details of Experimental Setup

The initial library models, which have trainable weights, have the following architecture. MLP modules have one hidden layer of size 1024, followed by batch normalization and dropout, followed by an output layer. CNNs have two convolutional layers with 32 and 64 output channels respectively, each with a 5x5 kernel, stride 1 and 0 padding, and each followed by max pooling, followed by spatial dropout. RNN modules are long short-term memory (LSTM) networks with a hidden dimension of 100, followed by an output layer, which transforms the last hidden state. For a given task, we use the input and output types of the new function to decide between MLP, CNN, or RNN, and also deduce the output activation function.

The standalone baseline for counting uses an architecture of the form $\lambda x.\mathrm{RNN}(\ (\mathrm{MLP} \circ \mathrm{CNN}(x)))$, which is intuitively appropriate for the task, and also matches the shape of some programs commonly returned by HOUDINI. As for the shortest path sequences, the first task for GS1 and GS2 is regression, which we train using a network with architecture $\mathrm{MLP} \circ \mathrm{CNN}$, in which the last layer is linear. In the RNN baseline for the other tasks in the graph sequences, we map a learned $\mathrm{MLP} \circ \mathrm{CNN}$ regression module to each image in the grid. Afterwards, we linearize the grid row-wise, converting it into a list, and then we process it using an LSTM (RNN) with hidden state of size 100. The number was chosen so that both our implementation and the baseline have almost the same number of parameters.

For multi-class classification (Sequence SS - Task 1) and regression (GS1 - Task1, GS2 - Task 1), we used all training images available. For the rest of the tasks in GS1, GS2, GS3 and SS, we use 12000 data points for training, with 2100 for testing. The list lengths for training are [2, 3, 4, 5], and [6, 7, 8] for testing in order to evaluate the generalization to longer sequences. We train for 20 epochs on all list-related tasks and for 1000 epochs for the regression tasks. The training datasets for the graph shortest path tasks (GS1 - Task 2; GS2 - Task2, GS2 - Task3) consists of 70,000 3x3 grids and 1,000,000 4x4 grids, while the testing datasets consists of 10,000 5x5 grids. The number of epochs for these tasks is 5. In GS2 - Task3, the *low-level transfer* baseline reuses the regression function learned in GS2 - Task1, thus, the image dimensions from MNIST and the colored GTSRB need to match. Therefore, we expanded the MNIST digit images, used for the graph sequences GS1 and GS2, to 28x28x3 dimensionality and resized the images from GTSRB from 32x32x3 to 28x28x3.

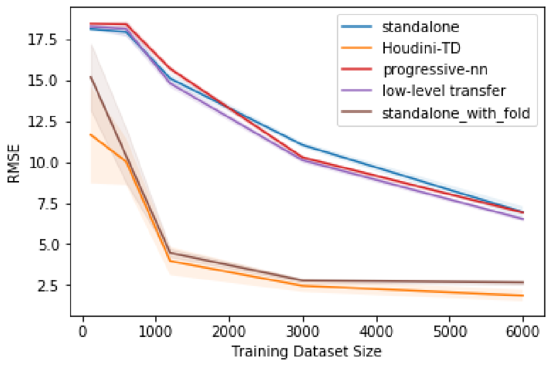
For all experiments, we use early stopping, reporting the the test error at the epoch where the validation error was minimized.

D Programs Discovered in Experiments

Tables 1-22 list the top 3 programs and the corresponding classification errors/RMSEs, on a test dataset, for most of our task sequences. The programs are ordered by their performance on a validaiton dataset. Finally, the presented programs are the ones evaluated for all (100%) of the training dataset. Here we use the syntax to denote function composition. Program terms with prefix "nn_" denote neural modules trained during the corresponding tasks whereas terms with prefix "lib." denote already trained neural modules in the library. For example, in Counting Sequence 1 (Table 1), "nn_cs1_1" is the neural module trained during Task 1 (recognize_digit(d_1)). After completion of this task, the neural module is added to the library and is available for use during the subsequent tasks. For example, the top performing program for Task 3 (count_digit(d_1)) uses the neural module "lib.nn_cs1_1" from the library (and a freshly trained neural module "nn_cs1_5") to construct a program for the counting task.

E Summing Experiment

In this section we present the result from task sequence SS in Figure 3 of the main paper. This sequence was designed to demonstrate low-level transfer of a multi-class classifier as well as the advantage of functional methods like foldl in specific situations. The first task of the sequence is a simple MNIST classifier, on which all competing methods do equally well. The second task is a regression task, to learn to sum all of the digits in the sequence. The standalone method, low level transfer one and the progressive neural networks all perform equally poorly (note that their lines are overplotted in the Figure), but the synthesized program from HOUDINI is able to learn this function easily because it is able to use a foldl operation. We also add a new baseline "standalone_with_fold", which reuses the program found by HOUDINI, but trains the parameter from a random initialization.



(a) Task 2: Sum digits

Figure 1: Lifelong learning for "learning to sum" (Sequence SS).

F Full Experimental Results on Counting Tasks

In the paper, we present results for the counting sequences on for the later tasks, in which transfer learning is possible. For completeness, in this section we present results on all of the tasks in the sequences. See Figures 2–4. We note that for the early tasks in each task sequence (e.g. CS1 tasks 1 and 2), there is little relevant information that can be transferred from early tasks, so as expected all methods perform similarly; e.g., the output of HOUDINI is a single library function.

Table 1: Counting Sequence 1(CS1). "CE" denotes classification error and "RMSE" denotes root mean square error.

Task	**Top 3 programs**	**Error**
Task 1: recognize_digit(d_1)	1. (nn_cs1_1, nn_cs1_2)	1% CE
Task 2: recognize_digit(d_2)	1. (nn_cs1_3, nn_cs1_4)	1% CE
	2. (nn_cs1_5, lib.nn_cs1_2)	1% CE
	3. (lib.nn_cs1_1, nn_cs1_6)	1% CE
Task 3: count_digit(d_1)	1. (nn_cs1_7, ((nn_cs1_8, lib.nn_cs1_2)))	0.38 RMSE
	2. ((nn_cs1_9, (nn_cs1_10)), (nn_cs1_11))	0.38 RMSE
	3. (nn_cs1_12, ((nn_cs1_13), (lib.nn_cs1_2)))	0.40 RMSE
Task 4: count_digit(d_2)	1. (nn_cs1_14, ((lib.nn_cs1_1), (nn_cs1_15)))	0.32 RMSE
	2. (lib.nn_cs1_7, ((nn_cs1_16, lib.nn_cs1_4)))	0.37 RMSE
	3. (lib.nn_cs1_7, ((nn_cs1_17), (lib.nn_cs1_4)))	0.37 RMSE

Table 2: Counting Sequence 2(CS2)

Task	**Top 3 programs**	**Error**
Task 1: recognize_digit(d_1)	1. (nn_cs2_1, nn_cs2_2)	1% CE
Task 2: count_digit(d_1)	1. (nn_cs2_3, ((nn_cs2_4), (nn_cs2_5)))	0.35 RMSE
	2. (nn_cs2_6, ((nn_cs2_7, nn_cs2_8)))	0.40 RMSE
	3. ((nn_cs2_9, (nn_cs2_10)), (nn_cs2_11))	0.41 RMSE
Task 3: count_digit(d_2)	1. (lib.nn_cs2_3, ((nn_cs2_12), (lib.nn_cs2_2)))	0.34 RMSE
	2. (lib.nn_cs2_3, ((nn_cs2_13, lib.nn_cs2_2)))	0.33 RMSE
	3. (lib.nn_cs2_3, ((nn_cs2_14), (nn_cs2_15)))	0.33 RMSE
Task 4: recognize_digit(d_2)	1. (nn_cs2_16, nn_cs2_17)	1% CE
	2. (nn_cs2_18, lib.nn_cs2_2)	1% CE
	3. (lib.nn_cs2_12, lib.nn_cs2_2)	1% CE

Table 3: Counting Sequence 3(CS3)

Task	**Top 3 Programs**	**Error**
Task 1: recognize_digit(d)	1. (nn_cs3_1, nn_cs3_2)	1% CE
Task 2: count_digit(d)	1. (nn_cs3_3, ((nn_cs3_4, nn_cs3_5)))	0.40 RMSE
	2. (nn_cs3_6, ((nn_cs3_7, lib.nn_cs3_2)))	0.40 RMSE
	3. (nn_cs3_8, ((nn_cs3_9), (lib.nn_cs3_2)))	0.41 RMSE
Task 3: count_toy(t)	1. (lib.nn_cs3_3, ((nn_cs3_10), (nn_cs3_11)))	-0.73
	2. (lib.nn_cs3_3, ((nn_cs3_12, nn_cs3_13)))	0.67 RMSE
	3. (lib.nn_cs3_3, ((lib.nn_cs3_1), (nn_cs3_14)))	0.96 RMSE
Task 4: recognize_toy(t)	1. (nn_cs3_15, lib.nn_cs3_11)	7% CE
	2. (nn_cs3_16, nn_cs3_17)	5% CE
	3. (lib.nn_cs3_10, lib.nn_cs3_11)	6% CE

Table 4: Summing Sequence(SS)

Task	**Top 3 programs**	**Error**
Task 1: classify_digit	1. (nn_ss_1, nn_ss_2)	1% CE
Task 2: sum_digits	1. ((nn_ss_3 zeros(1)), ((nn_ss_4, lib.nn_ss_2)))	2.15 RMSE
	2. ((nn_ss_5 zeros(1)), ((nn_ss_6, nn_ss_7)))	2.58 RMSE
	3. ((nn_ss_8 zeros(1)), ((lib.nn_ss_1, lib.nn_ss_2)))	4.30 RMSE

Table 5: Graph Sequence 1(GS1)

Task	**Top 3 Programs**	**Error**
Task 1: regress_speed	1. (nn_gs1_1, nn_gs1_2)	0.64 RMSE
Task 2: shortest_path_street	1. (^{10}(nn_gs1_3), ((lib.nn_gs1_1, lib.nn_gs1_2)))	1.88 RMSE
	2. (^{9}(nn_gs1_4), ((lib.nn_gs1_1, lib.nn_gs1_2)))	2.02 RMSE
	3. (^{10}(nn_gs1_5), ((nn_gs1_6, nn_gs1_7)))	6.76 RMSE

Table 6: Graph Sequence 2(GS2)

Task	**Top 3 Programs**	**Error**
Task 1: regress_mnist	1. (nn_gs2_1, nn_gs2_2)	1.47 RMSE
Task 2: shortest_path_mnist	1. (^{10}(nn_gs2_3), ((lib.nn_gs2_1, lib.nn_gs2_2)))	1.57 RMSE
	2. (^{9}(nn_gs2_4), ((lib.nn_gs2_1, lib.nn_gs2_2)))	1.73 RMSE
	3. (^{9}(nn_gs2_5), ((nn_gs2_6, nn_gs2_7)))	4.99 RMSE
Task 3: shortest_path_street	1. (^{10}(lib.nn_gs2_3), ((nn_gs2_8, nn_gs2_9)))	3.48 RMSE
	2. (^{9}(lib.nn_gs2_3), ((nn_gs2_10, nn_gs2_11)))	3.84 RMSE
	3. (^{10}(lib.nn_gs2_3), ((lib.nn_gs2_1, lib.nn_gs2_2)))	6.92 RMSE

Table 7: Long Sequence 1(LS1).

Task	Top 3 Programs	Error
Task 1: count_digit(7)	1. ((nn_ls1_1, (nn_ls1_2)), (nn_ls1_3))	0.46 RMSE
	2. (nn_ls1_4, ((nn_ls1_5, nn_ls1_6)))	0.49 RMSE
	3. (nn_ls1_7, ((nn_ls1_8), (nn_ls1_9)))	0.51 RMSE
Task 2: count_digit(4)	1. (lib.nn_ls1_1, ((nn_ls1_10), (nn_ls1_11)))	1.50 RMSE
	2. (lib.nn_ls1_1, ((nn_ls1_12, nn_ls1_13)))	1.61 RMSE
	3. ((lib.nn_ls1_1, (nn_ls1_14)), (nn_ls1_15))	1.64 RMSE
Task 3: recognize_toy(0)	1. (nn_ls1_16, lib.nn_ls1_11)	9.81% CE
	2. (nn_ls1_17, nn_ls1_18)	8.86% CE
	3. (nn_ls1_19, lib.nn_ls1_3)	12.86% CE
Task 4: recognize_digit(9)	1. (nn_ls1_20, nn_ls1_21)	1.38% CE
	2. (nn_ls1_22, lib.nn_ls1_3)	2.14% CE
	3. (lib.nn_ls1_2, nn_ls1_23)	1.95% CE
Task 5: count_digit(2)	1. (nn_ls1_24, ((nn_ls1_25), (nn_ls1_26)))	1.08 RMSE
	2. (lib.nn_ls1_1, ((nn_ls1_27, nn_ls1_28)))	1.02 RMSE
	3. (lib.nn_ls1_1, ((lib.nn_ls1_16, nn_ls1_29)))	0.95 RMSE
Task 6: count_digit(9)	1. (lib.nn_ls1_1, ((nn_ls1_30, nn_ls1_31)))	0.49 RMSE
	2. (lib.nn_ls1_1, ((nn_ls1_32, lib.nn_ls1_21)))	0.49 RMSE
	3. (lib.nn_ls1_1, ((nn_ls1_33, lib.nn_ls1_3)))	0.49 RMSE
Task 7: count_digit(0)	1. (nn_ls1_34, ((lib.nn_ls1_16, lib.nn_ls1_11)))	0.94 RMSE
	2. (lib.nn_ls1_1, ((nn_ls1_35, nn_ls1_36)))	0.81 RMSE
	3. (nn_ls1_37, ((nn_ls1_38, nn_ls1_39)))	0.85 RMSE
Task 8: recognize_digit(7)	1. (lib.nn_ls1_2, lib.nn_ls1_3)	0.86% CE
	2. (nn_ls1_40, lib.nn_ls1_3)	1.19% CE
	3. (nn_ls1_41, lib.nn_ls1_21)	1.05% CE
Task 9: count_digit(2)	1. (nn_ls1_42, ((nn_ls1_43, lib.nn_ls1_26)))	0.43 RMSE
	2. (lib.nn_ls1_1, ((nn_ls1_44, nn_ls1_45)))	0.45 RMSE
	3. (lib.nn_ls1_1, ((nn_ls1_46, lib.nn_ls1_3)))	0.45 RMSE

Table 8: Long Sequence 2(LS2).

Task	Top 3 Programs	Error
Task 1: count_digit(1)	1. (nn_ls2_1, ((nn_ls2_2, nn_ls2_3)))	0.43 RMSE
	2. (nn_ls2_4, ((nn_ls2_5), (nn_ls2_6)))	0.45 RMSE
	3. ((nn_ls2_7, (nn_ls2_8)), (nn_ls2_9))	0.48 RMSE
Task 2: count_digit(0)	1. (nn_ls2_10, ((nn_ls2_11), (nn_ls2_12)))	0.96 RMSE
	2. (lib.nn_ls2_1, ((nn_ls2_13, nn_ls2_14)))	0.84 RMSE
	3. ((lib.nn_ls2_1, (nn_ls2_15)), (nn_ls2_16))	0.92 RMSE
Task 3: recognize_toy(1)	1. (nn_ls2_17, nn_ls2_18)	5.05% CE
	2. (nn_ls2_19, lib.nn_ls2_12)	4.00% CE
	3. (lib.nn_ls2_2, nn_ls2_20)	10.52% CE
Task 4: recognize_digit(5)	1. (nn_ls2_21, nn_ls2_22)	0.76% CE
	2. (nn_ls2_23, lib.nn_ls2_3)	0.86% CE
	3. (lib.nn_ls2_17, nn_ls2_24)	0.81% CE
Task 5: count_digit(4)	1. (lib.nn_ls2_1, ((nn_ls2_25, nn_ls2_26)))	1.68 RMSE
	2. (lib.nn_ls2_1, ((nn_ls2_27, lib.nn_ls2_18)))	1.51 RMSE
	3. (lib.nn_ls2_1, ((nn_ls2_28, lib.nn_ls2_12)))	1.46 RMSE
Task 6: count_digit(5)	1. (nn_ls2_29, ((nn_ls2_30, nn_ls2_31)))	0.43 RMSE
	2. (nn_ls2_32, ((lib.nn_ls2_21, lib.nn_ls2_22)))	0.43 RMSE
	3. (lib.nn_ls2_1, ((nn_ls2_33, lib.nn_ls2_22)))	0.45 RMSE
Task 7: count_digit(1)	1. (nn_ls2_34, ((lib.nn_ls2_25, nn_ls2_35)))	0.64 RMSE
	2. (nn_ls2_36, ((nn_ls2_37, nn_ls2_38)))	0.74 RMSE
	3. (nn_ls2_39, ((nn_ls2_40, lib.nn_ls2_26)))	0.83 RMSE
Task 8: recognize_digit(1)	1. (nn_ls2_41, lib.nn_ls2_3)	0.29% CE
	2. (nn_ls2_42, lib.nn_ls2_12)	0.19% CE
	3. (nn_ls2_43, lib.nn_ls2_22)	0.24% CE
Task 9: count_digit(8)	1. (nn_ls2_44, ((nn_ls2_45, lib.nn_ls2_31)))	0.46 RMSE
	2. (nn_ls2_46, ((nn_ls2_47, lib.nn_ls2_26)))	0.45 RMSE
	3. (nn_ls2_48, ((nn_ls2_49, lib.nn_ls2_3)))	0.47 RMSE

Table 9: Long Sequence 3(LS3).

Task	Top 3 Programs	Error
Task 1: count_digit(9)	1. (nn_ls3_1, ((nn_ls3_2), (nn_ls3_3)))	0.46 RMSE
	2. (nn_ls3_4, ((nn_ls3_5, nn_ls3_6)))	0.48 RMSE
	3. ((nn_ls3_7, (nn_ls3_8)), (nn_ls3_9))	0.55 RMSE
Task 2: count_digit(1)	1. (lib.nn_ls3_1, ((nn_ls3_10, nn_ls3_11)))	0.63 RMSE
	2. (lib.nn_ls3_1, ((nn_ls3_12), (nn_ls3_13)))	0.68 RMSE
	3. ((lib.nn_ls3_1, (nn_ls3_14)), (nn_ls3_15))	0.63 RMSE
Task 3: recognize_toy(2)	1. (nn_ls3_16, nn_ls3_17)	8.19% CE
	2. (nn_ls3_18, lib.nn_ls3_11)	9.95% CE
	3. (lib.nn_ls3_2, nn_ls3_19)	14.00% CE
Task 4: recognize_digit(1)	1. (nn_ls3_20, lib.nn_ls3_3)	0.38% CE
	2. (nn_ls3_21, lib.nn_ls3_17)	0.48% CE
	3. (nn_ls3_22, nn_ls3_23)	0.24% CE
Task 5: count_digit(3)	1. (lib.nn_ls3_1, ((nn_ls3_24, nn_ls3_25)))	0.51 RMSE
	2. (lib.nn_ls3_1, ((nn_ls3_26, lib.nn_ls3_17)))	0.66 RMSE
	3. (lib.nn_ls3_1, ((nn_ls3_27, lib.nn_ls3_11)))	0.61 RMSE
Task 6: count_digit(1)	1. (nn_ls3_28, ((nn_ls3_29, lib.nn_ls3_11)))	0.38 RMSE
	2. (nn_ls3_30, ((nn_ls3_31, lib.nn_ls3_3)))	0.37 RMSE
	3. (lib.nn_ls3_1, ((nn_ls3_32, lib.nn_ls3_3)))	0.40 RMSE
Task 7: count_digit(2)	1. (nn_ls3_33, ((nn_ls3_34, lib.nn_ls3_17)))	0.96 RMSE
	2. (lib.nn_ls3_1, ((nn_ls3_35, nn_ls3_36)))	0.99 RMSE
	3. (lib.nn_ls3_1, ((nn_ls3_37, lib.nn_ls3_17)))	0.90 RMSE
Task 8: recognize_digit(9)	1. (nn_ls3_38, nn_ls3_39)	1.52% CE
	2. (lib.nn_ls3_2, nn_ls3_40)	2.43% CE
	3. (lib.nn_ls3_2, lib.nn_ls3_3)	1.43% CE
Task 9: count_digit(3)	1. (nn_ls3_41, ((nn_ls3_42, nn_ls3_43)))	0.39 RMSE
	2. (nn_ls3_44, ((nn_ls3_45, lib.nn_ls3_39)))	0.42 RMSE
	3. (nn_ls3_46, ((nn_ls3_47, lib.nn_ls3_3)))	0.44 RMSE

Table 10: Long Sequence 4(LS4).

Task	Top 3 Programs	Error
Task 1: count_digit(6)	1. (nn_ls4_1, ((nn_ls4_2), (nn_ls4_3)))	0.40 RMSE
	2. (nn_ls4_4, ((nn_ls4_5, nn_ls4_6)))	0.45 RMSE
	3. ((nn_ls4_7, (nn_ls4_8)), (nn_ls4_9))	0.48 RMSE
Task 2: count_digit(2)	1. (lib.nn_ls4_1, ((nn_ls4_10), (nn_ls4_11)))	0.89 RMSE
	2. (lib.nn_ls4_1, ((nn_ls4_12, nn_ls4_13)))	0.99 RMSE
	3. ((lib.nn_ls4_1, (nn_ls4_14)), (nn_ls4_15))	0.91 RMSE
Task 3: recognize_toy(3)	1. (nn_ls4_16, lib.nn_ls4_11)	4.95% CE
	2. (nn_ls4_17, nn_ls4_18)	4.00% CE
	3. (lib.nn_ls4_10, nn_ls4_19)	2.43% CE
Task 4: recognize_digit(8)	1. (nn_ls4_20, lib.nn_ls4_3)	0.71% CE
	2. (nn_ls4_21, nn_ls4_22)	0.52% CE
	3. (nn_ls4_23, lib.nn_ls4_11)	0.86% CE
Task 5: count_digit(1)	1. (lib.nn_ls4_1, ((nn_ls4_24, nn_ls4_25)))	0.64 RMSE
	2. (lib.nn_ls4_1, ((nn_ls4_26, lib.nn_ls4_11)))	0.57 RMSE
	3. (lib.nn_ls4_1, ((lib.nn_ls4_16, nn_ls4_27)))	0.70 RMSE
Task 6: count_digit(8)	1. (lib.nn_ls4_1, ((nn_ls4_28, lib.nn_ls4_3)))	0.39 RMSE
	2. (lib.nn_ls4_1, ((nn_ls4_29, nn_ls4_30)))	0.38 RMSE
	3. (lib.nn_ls4_1, ((nn_ls4_31, lib.nn_ls4_11)))	0.40 RMSE
Task 7: count_digit(3)	1. (lib.nn_ls4_1, ((nn_ls4_32, lib.nn_ls4_11)))	0.61 RMSE
	2. (lib.nn_ls4_1, ((nn_ls4_33, nn_ls4_34)))	0.54 RMSE
	3. (lib.nn_ls4_1, ((nn_ls4_35, lib.nn_ls4_25)))	0.60 RMSE
Task 8: recognize_digit(6)	1. (nn_ls4_36, lib.nn_ls4_3)	0.81% CE
	2. (lib.nn_ls4_20, nn_ls4_37)	0.90% CE
	3. (nn_ls4_38, nn_ls4_39)	0.86% CE
Task 9: count_digit(5)	1. (lib.nn_ls4_1, ((nn_ls4_40, nn_ls4_41)))	0.37 RMSE
	2. (lib.nn_ls4_1, ((nn_ls4_42, lib.nn_ls4_3)))	0.39 RMSE
	3. (lib.nn_ls4_1, ((nn_ls4_43, lib.nn_ls4_11)))	0.39 RMSE

Table 11: Long Sequence 5(LS5).

Task	Top 3 Programs	Error
Task 1: count_digit(4)	1. (nn_ls5_1, ((nn_ls5_2), (nn_ls5_3)))	0.45 RMSE
	2. (nn_ls5_4, ((nn_ls5_5, nn_ls5_6)))	0.46 RMSE
	3. ((nn_ls5_7, (nn_ls5_8)), (nn_ls5_9))	0.48 RMSE
Task 2: count_digit(3)	1. (nn_ls5_10, ((nn_ls5_11), (lib.nn_ls5_3)))	0.60 RMSE
	2. (lib.nn_ls5_1, ((nn_ls5_12), (nn_ls5_13)))	0.63 RMSE
	3. ((lib.nn_ls5_1, (nn_ls5_14)), (nn_ls5_15))	0.58 RMSE
Task 3: recognize_toy(4)	1. (nn_ls5_16, nn_ls5_17)	20.33% CE
	2. (lib.nn_ls5_11, nn_ls5_18)	17.76% CE
	3. (nn_ls5_19, lib.nn_ls5_3)	21.38% CE
Task 4: recognize_digit(7)	1. (nn_ls5_20, nn_ls5_21)	1.19% CE
	2. (nn_ls5_22, lib.nn_ls5_3)	0.90% CE
	3. (lib.nn_ls5_2, nn_ls5_23)	1.62% CE
Task 5: count_digit(0)	1. (lib.nn_ls5_10, ((nn_ls5_24, nn_ls5_25)))	0.90 RMSE
	2. (lib.nn_ls5_10, ((nn_ls5_26, lib.nn_ls5_17)))	0.90 RMSE
	3. (lib.nn_ls5_10, ((nn_ls5_27, lib.nn_ls5_3)))	0.86 RMSE
Task 6: count_digit(7)	1. (lib.nn_ls5_1, ((nn_ls5_28, nn_ls5_29)))	0.47 RMSE
	2. (lib.nn_ls5_1, ((nn_ls5_30, lib.nn_ls5_21)))	0.47 RMSE
	3. (nn_ls5_31, ((lib.nn_ls5_16, nn_ls5_32)))	0.47 RMSE
Task 7: count_digit(4)	1. (nn_ls5_33, ((nn_ls5_34, lib.nn_ls5_25)))	1.72 RMSE
	2. (nn_ls5_35, ((nn_ls5_36, lib.nn_ls5_17)))	1.50 RMSE
	3. (lib.nn_ls5_1, ((nn_ls5_37, nn_ls5_38)))	1.80 RMSE
Task 8: recognize_digit(4)	1. (nn_ls5_39, lib.nn_ls5_3)	0.29% CE
	2. (nn_ls5_40, lib.nn_ls5_21)	0.38% CE
	3. (lib.nn_ls5_20, nn_ls5_41)	0.48% CE
Task 9: count_digit(0)	1. (nn_ls5_42, ((nn_ls5_43, nn_ls5_44)))	0.37 RMSE
	2. (nn_ls5_45, ((lib.nn_ls5_24, nn_ls5_46)))	0.40 RMSE
	3. (nn_ls5_47, ((nn_ls5_48, lib.nn_ls5_21)))	0.40 RMSE

Table 12: Counting Sequence 1(CS1), Evolutionary Algorithm. "CE" denotes classification error and "RMSE" denotes root mean square error.

Task	Top 3 programs	Error
Task 1: recognize_digit(d_1)	1. (nn_cs1_1, nn_cs1_2)	0.57% CE
	2. (nn_cs1_3, nn_cs1_2)	0.38% CE
	3. (nn_cs1_4, nn_cs1_2)	0.76% CE
Task 2: recognize_digit(d_2)	1. (nn_cs1_5, nn_cs1_6)	0.38% CE
	2. (nn_cs1_7, nn_cs1_8)	0.48% CE
	3. (nn_cs1_9, nn_cs1_10)	0.43% CE
Task 3: count_digit(d_1)	1. (nn_cs1_11, ((nn_cs1_12, lib.nn_cs1_2)))	0.38 RMSE
	2. (nn_cs1_13, ((nn_cs1_14, lib.nn_cs1_2)))	0.38 RMSE
	3. (nn_cs1_15, ((lib.nn_cs1_1, lib.nn_cs1_2)))	0.40 RMSE
Task 4: count_digit(d_2)	No Solution	

Table 13: Counting Sequence 2(CS2), Evolutionary Algorithm.

Task	Top 3 programs	Error
Task 1: recognize_digit(d_1)	1. (nn_cs2_1, nn_cs2_2)	1% CE
	2. (nn_cs2_3, nn_cs2_4)	1% CE
	3. (nn_cs2_5, nn_cs2_2)	1% CE
Task 2: count_digit(d_1)	1. (nn_cs2_6, ((lib.nn_cs2_1, lib.nn_cs2_2)))	0.38 RMSE
	2. (nn_cs2_6, ((nn_cs2_7, lib.nn_cs2_2)))	0.38 RMSE
	3. (nn_cs2_8, ((nn_cs2_9, nn_cs2_10)))	0.39 RMSE
Task 3: count_digit(d_2)	No Solution	
Task 4: recognize_digit(d_2)	1. (nn_cs2_11, nn_cs2_12)	1% CE
	2. (nn_cs2_13, nn_cs2_14)	1% CE
	3. (lib.nn_cs2_1, nn_cs2_15)	1% CE

Table 14: Counting Sequence 3(CS3), Evolutionary Algorithm.

Task	Top 3 Programs	Error
Task 1: recognize_digit(d)	1. (nn_cs3_1, nn_cs3_2)	0.57% CE
	2. (nn_cs3_3, nn_cs3_4)	0.67% CE
	3. (nn_cs3_5, nn_cs3_6)	0.62% CE
Task 2: count_digit(d)	1. (nn_cs3_7, ((nn_cs3_8, lib.nn_cs3_2)))	0.36 RMSE
	2. (nn_cs3_7, ((nn_cs3_9, nn_cs3_10)))	0.39 RMSE
	3. (nn_cs3_11, ((nn_cs3_12, nn_cs3_13)))	0.39 RMSE
Task 3: count_toy(t)	1. (lib.nn_cs3_7, ((nn_cs3_14, nn_cs3_15)))	0.70 RMSE
	2. (lib.nn_cs3_7, ((nn_cs3_16, nn_cs3_17)))	0.61 RMSE
	3. (lib.nn_cs3_7, ((nn_cs3_18, nn_cs3_19)))	0.64 RMSE
Task 4: recognize_toy(t)	1. (nn_cs3_20, lib.nn_cs3_15)	5.62% CE
	2. (lib.nn_cs3_14, lib.nn_cs3_15)	5.38% CE
	3. (nn_cs3_21, lib.nn_cs3_15)	5.76% CE

Table 15: Summing Sequence(SS), Evolutionary Algorithm

Task	**Top 3 programs**	**Error**
Task 1: classify_digit	1. (nn_ss_1, nn_ss_2)	1% CE
Task 2: sum_digits	1. ((nn_ss_3 zeros(1)), ((nn_ss_4, lib.nn_ss_2)))	6.64 RMSE
	2. ((nn_ss_5 zeros(1)), ((nn_ss_6, lib.nn_ss_2)))	6.66 RMSE
	3. ((nn_ss_7 zeros(1)), ((nn_ss_8, lib.nn_ss_2)))	6.70 RMSE

Table 16: Graph Sequence 1(GS1), Evolutionary Algorithm.

Task	**Top 3 Programs**	**Error**
Task 1: regress_speed	1. (nn_gs1_1, nn_gs1_2)	0.80 RMSE
Task 2: shortest_path_street	1. ((nn_gs1_3, lib.nn_gs1_2))	8.36 RMSE
	2. ((nn_gs1_4, nn_gs1_5))	8.37 RMSE
	3. ((nn_gs1_6, lib.nn_gs1_2))	8.35 RMSE

Table 17: Graph Sequence 2(GS2), Evolutionary Algorithm.

Task	**Top 3 Programs**	**Error**
Task 1: regress_mnist	1. (nn_gs2_1, nn_gs2_2)	1.47 RMSE
Task 2: shortest_path_mnist	1. ((lib.nn_gs2_1, nn_gs2_3))	6.58 RMSE
	2. ((lib.nn_gs2_1, nn_gs2_4))	6.59 RMSE
	3. ((lib.nn_gs2_1, nn_gs2_5))	6.63 RMSE
Task 3: shortest_path_street	1. ((lib.nn_gs2_1, nn_gs2_6))	7.82 RMSE
	2. ((lib.nn_gs2_1, nn_gs2_7))	7.87 RMSE
	3. ((nn_gs2_8, nn_gs2_9))	7.96 RMSE

Table 18: Long Sequence 1(LS1), Evolutionary Algorithm.

Task	**Top 3 Programs**	**Error**
Task 1: count_digit(7)	1. (nn_ls1_1, ((nn_ls1_2, nn_ls1_3)))	0.42 RMSE
	2. (nn_ls1_4, ((nn_ls1_5, nn_ls1_6)))	0.44 RMSE
	3. (nn_ls1_1, ((nn_ls1_7, nn_ls1_8)))	0.50 RMSE
Task 2: count_digit(4)	1. (lib.nn_ls1_1, ((nn_ls1_9, nn_ls1_10)))	1.65 RMSE
	2. (lib.nn_ls1_1, ((nn_ls1_11, nn_ls1_12)))	1.53 RMSE
	3. (lib.nn_ls1_1, ((nn_ls1_13, nn_ls1_14)))	1.60 RMSE
Task 3: recognize_toy(0)	1. (nn_ls1_15, lib.nn_ls1_10)	9.81% CE
	2. (nn_ls1_16, nn_ls1_17)	9.76% CE
	3. (nn_ls1_18, lib.nn_ls1_10)	8.76% CE
Task 4: recognize_digit(9)	1. (nn_ls1_19, nn_ls1_20)	1.43% CE
	2. (nn_ls1_21, nn_ls1_22)	1.43% CE
	3. (nn_ls1_23, nn_ls1_24)	1.62% CE
Task 5: count_digit(2)	1. (nn_ls1_25, ((lib.nn_ls1_19, nn_ls1_26)))	0.90 RMSE
	2. (lib.nn_ls1_1, ((nn_ls1_27, nn_ls1_28)))	0.94 RMSE
	3. (lib.nn_ls1_1, ((nn_ls1_29, nn_ls1_30)))	0.98 RMSE
Task 6: count_digit(9)	No Solution	
Task 7: count_digit(0)	No Solution	
Task 8: recognize_digit(7)	1. (nn_ls1_31, lib.nn_ls1_26)	1.29% CE
	2. (nn_ls1_32, lib.nn_ls1_3)	0.71% CE
	3. (nn_ls1_33, nn_ls1_34)	1.19% CE
Task 9: count_digit(2)	1. (nn_ls1_35, ((nn_ls1_36, lib.nn_ls1_3)))	0.36 RMSE
	2. (lib.nn_ls1_25, ((nn_ls1_36, nn_ls1_37)))	0.38 RMSE
	3. (nn_ls1_38, ((nn_ls1_39, nn_ls1_40)))	0.37 RMSE

Table 19: Long Sequence 2(LS2), Evolutionary Algorithm.

Task	**Top 3 Programs**	**Error**
Task 1: count_digit(1)	No Solution	
Task 2: count_digit(0)	No Solution	
Task 3: recognize_toy(1)	1. (nn_ls2_1, nn_ls2_2)	5.43% CE
	2. (nn_ls2_3, nn_ls2_4)	5.81% CE
	3. (nn_ls2_5, nn_ls2_6)	5.05% CE
Task 4: recognize_digit(5)	1. (nn_ls2_7, nn_ls2_8)	0.71% CE
	2. (nn_ls2_9, nn_ls2_10)	0.43% CE
	3. (nn_ls2_9, nn_ls2_11)	0.62% CE
Task 5: count_digit(4)	No Solution	
Task 6: count_digit(5)	No Solution	
Task 7: count_digit(1)	No Solution	
Task 8: recognize_digit(1)	1. (nn_ls2_12, nn_ls2_13)	0.19% CE
	2. (nn_ls2_14, lib.nn_ls2_2)	0.29% CE
	3. (nn_ls2_15, lib.nn_ls2_2)	0.33% CE
Task 9: count_digit(8)	No Solution	

Table 20: Long Sequence 3(LS3), Evolutionary Algorithm.

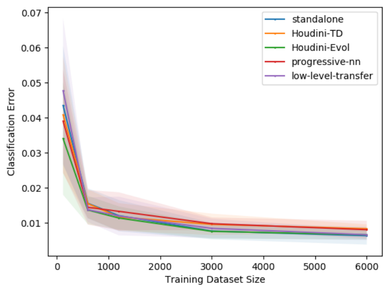
Task	Top 3 Programs	Error
Task 1: count_digit(9)	No Solution	
Task 2: count_digit(1)	No Solution	
Task 3: recognize_toy(2)	1. (nn_ls3_1, nn_ls3_2)	10.52% CE
	2. (nn_ls3_3, nn_ls3_2)	9.14% CE
	3. (nn_ls3_4, nn_ls3_2)	10.81% CE
Task 4: recognize_digit(1)	1. (nn_ls3_5, nn_ls3_6)	0.48% CE
	2. (nn_ls3_7, lib.nn_ls3_2)	0.33% CE
	3. (nn_ls3_8, nn_ls3_9)	0.24% CE
Task 5: count_digit(3)	No Solution	
Task 6: count_digit(1)	1. (nn_ls3_10, ((nn_ls3_11, lib.nn_ls3_6)))	0.38 RMSE
	2. (nn_ls3_12, ((nn_ls3_13, nn_ls3_14)))	0.37 RMSE
	3. (nn_ls3_15, ((nn_ls3_11, lib.nn_ls3_6)))	0.39 RMSE
Task 7: count_digit(2)	1. (lib.nn_ls3_10, ((nn_ls3_16, nn_ls3_17)))	1.02 RMSE
	2. (lib.nn_ls3_10, ((nn_ls3_18, nn_ls3_17)))	0.92 RMSE
	3. (lib.nn_ls3_10, ((nn_ls3_16, nn_ls3_19)))	0.96 RMSE
Task 8: recognize_digit(9)	1. (nn_ls3_20, nn_ls3_21)	1.05% CE
	2. (nn_ls3_22, nn_ls3_23)	1.14% CE
	3. (nn_ls3_24, lib.nn_ls3_17)	1.86% CE
Task 9: count_digit(3)	1. (lib.nn_ls3_10, ((nn_ls3_25, lib.nn_ls3_21)))	0.45 RMSE
	2. (lib.nn_ls3_10, ((nn_ls3_26, lib.nn_ls3_17)))	0.47 RMSE
	3. (lib.nn_ls3_10, ((nn_ls3_25, lib.nn_ls3_21)))	0.46 RMSE

Table 21: Long Sequence 4(LS4), Evolutionary Algorithm.

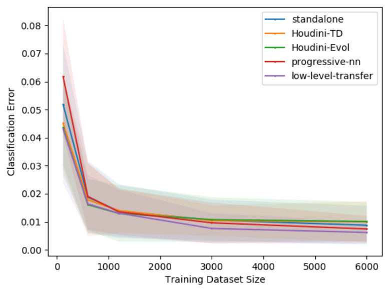
Task	Top 3 Programs	Error
Task 1: count_digit(6)	No Solution	
Task 2: count_digit(2)	No Solution	
Task 3: recognize_toy(3)	1. (nn_ls4_1, nn_ls4_2)	5.10% CE
	2. (nn_ls4_3, nn_ls4_4)	3.57% CE
	3. (nn_ls4_5, nn_ls4_6)	4.24% CE
Task 4: recognize_digit(8)	1. (nn_ls4_7, nn_ls4_8)	0.33% CE
	2. (nn_ls4_9, nn_ls4_10)	0.48% CE
	3. (nn_ls4_11, nn_ls4_12)	0.90% CE
Task 5: count_digit(1)	No Solution	
Task 6: count_digit(8)	1. (nn_ls4_13, ((nn_ls4_14, nn_ls4_15)))	0.41 RMSE
	2. (nn_ls4_16, ((nn_ls4_17, lib.nn_ls4_8)))	0.44 RMSE
	3. (nn_ls4_18, ((nn_ls4_19, lib.nn_ls4_8)))	0.44 RMSE
Task 7: count_digit(3)	1. (lib.nn_ls4_13, ((nn_ls4_20, nn_ls4_21)))	0.56 RMSE
	2. (lib.nn_ls4_13, ((nn_ls4_22, lib.nn_ls4_2)))	0.59 RMSE
	3. (lib.nn_ls4_13, ((nn_ls4_20, lib.nn_ls4_2)))	0.57 RMSE
Task 8: recognize_digit(6)	1. (nn_ls4_23, lib.nn_ls4_15)	0.48% CE
	2. (nn_ls4_24, lib.nn_ls4_15)	0.62% CE
	3. (nn_ls4_25, lib.nn_ls4_15)	0.71% CE
Task 9: count_digit(5)	1. (lib.nn_ls4_13, ((nn_ls4_26, nn_ls4_27)))	0.41 RMSE
	2. (lib.nn_ls4_13, ((nn_ls4_26, nn_ls4_27)))	0.41 RMSE
	3. (lib.nn_ls4_13, ((nn_ls4_28, nn_ls4_29)))	0.41 RMSE

Table 22: Long Sequence 5(LS5), Evolutionary Algorithm.

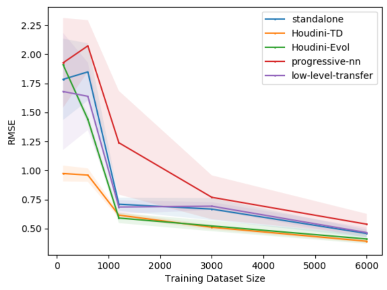
Task	Top 3 Programs	Error
Task 1: count_digit(4)	No Solution	
Task 2: count_digit(3)	No Solution	
Task 3: recognize_toy(4)	1. (nn_ls5_1, nn_ls5_2)	17.00% CE
	2. (nn_ls5_3, nn_ls5_4)	21.62% CE
	3. (nn_ls5_5, nn_ls5_6)	16.52% CE
Task 4: recognize_digit(7)	1. (nn_ls5_7, nn_ls5_8)	1.14% CE
	2. (nn_ls5_9, nn_ls5_10)	0.95% CE
	3. (nn_ls5_11, nn_ls5_12)	1.00% CE
Task 5: count_digit(0)	No Solution	
Task 6: count_digit(7)	No Solution	
Task 7: count_digit(4)	No Solution	
Task 8: recognize_digit(4)	1. (nn_ls5_13, nn_ls5_14)	0.38% CE
	2. (nn_ls5_15, lib.nn_ls5_8)	0.33% CE
	3. (nn_ls5_15, nn_ls5_16)	0.33% CE
Task 9: count_digit(0)	1. (nn_ls5_17, ((nn_ls5_18, lib.nn_ls5_8)))	0.38 RMSE
	2. (nn_ls5_17, ((nn_ls5_19, lib.nn_ls5_2)))	0.38 RMSE
	3. (nn_ls5_17, ((nn_ls5_20, lib.nn_ls5_8)))	0.40 RMSE



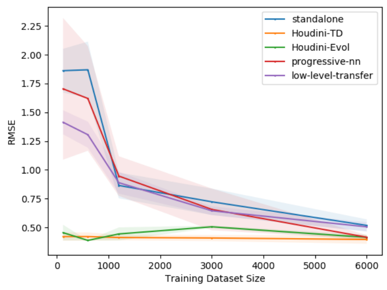
(a) Task 1: recognize_digit(d_1)



(b) Task 2: recognize_digit(d_2)



(c) Task 3: count_digit(d_1)

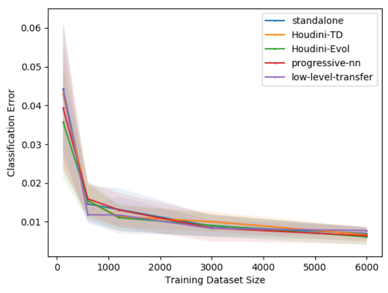


(d) Task 4: count_digit(d_2)

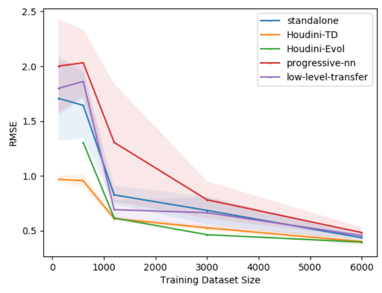
Figure 2: Lifelong learning for "learning to count" (Sequence CS1), demonstrating low-level transfer of perceptual recognizers.

G Results on Longer Task Sequence LS

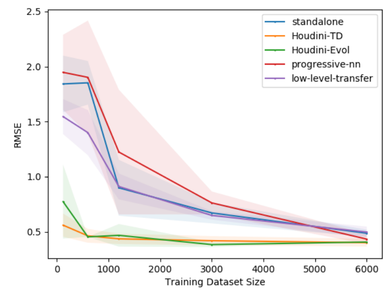
We report the performance of all methods on the longer task sequences on Figure 5. To save space, we report performance of all methods when trained on 10% of the data. The full learning curves follow similar patterns as the other task sequences. We report the classification and regression tasks from LS separately, because the error functions for the two tasks have different dynamic ranges. Please note that in the Figure, the tasks are labelled starting from 0. On the classification tasks, we note that all methods have similar performance. Examining the task sequence LS from Figure **??**, we see that these tasks have no opportunity to transfer from earlier tasks. On the regression tasks however, there is opportunity to transfer, and we see that HOUDINI shows much better performance than the other methods.



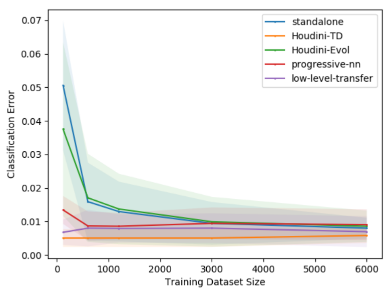
(a) Task 1: recognize_digit(d_1)



(b) Task 2: count_digit(d_1)

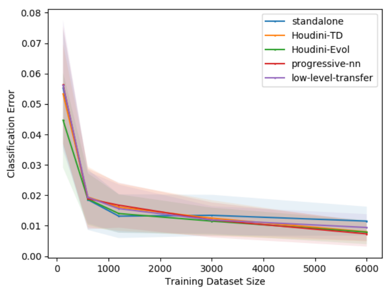


(c) Task 3: count_digit(d_2)

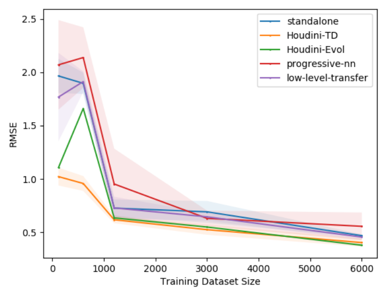


(d) Task 4: recognize_digit(d_2)

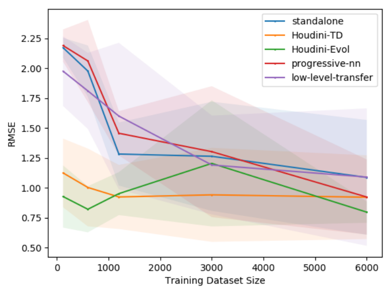
Figure 3: Lifelong learning for "learning to count" (Sequence CS2), demonstrating high-level transfer of a counting network across categories.



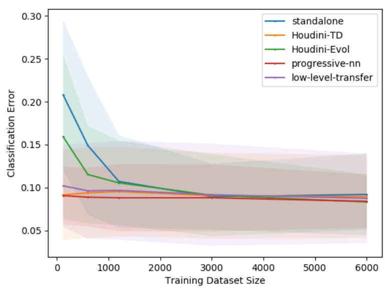
(a) Task 1: recognize_digit(d_1)



(b) Task 2: count_digit(d_1)



(c) Task 3: count_toy(t_1)



(d) Task 4: recognize_toy(t_1)

Figure 4: Lifelong learning for "learning to count" (Sequence CS3), demonstrating high-level transfer across different types of images. After learning to count MNIST digits, the same network can be used to count images of toys.

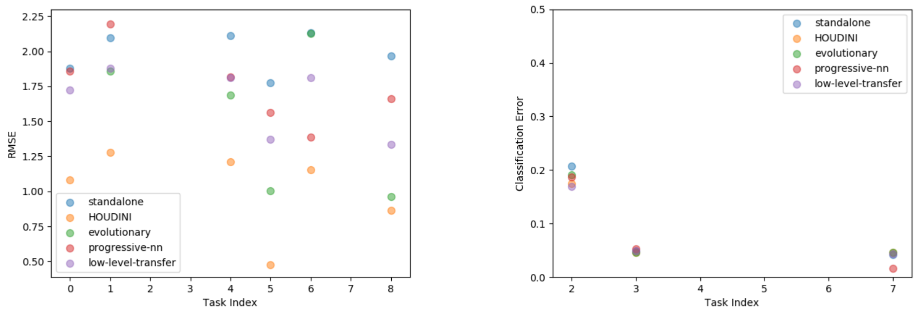


Figure 5: Performance of transfer learning systems on task sequence LS1. At top: regression tasks. At bottom: classification tasks

4.1 Related Work

Considering the neurosymbolic categorisation outlined in Section 3.2, HOUDINI is the closest to *Category 2*, in which the methods contain a neural sub-routine within a symbolic solver. This is because our symbolic program synthesis solves a given problem by repeatedly calling a neural-network-based sub-routine which trains a given program and returns its performance on a held-out dataset. Interestingly, by modifying HOUDINI's DSL, one could allow our framework to synthesise other neurosymbolic systems, as discussed in the next section.

HOUDINI operates over modular deep neural networks. The modular deep learning (MDL) algorithms discussed in Section 3.1 condition the choice of modules on inputs within the same problem. They concurrently optimise the parameters of all modules within different module compositions, which allows their modules to learn more reusable functions. In contrast, HOUDINI conditions the choice of modules (functions) which we use on the given problem. In our experiments, we encourage modules to learn reusable functions by ordering the problems of our sequences according to a curriculum. We note that applying MDL algorithms to the setting of lifelong learning would require augmenting them in order to address catastrophic forgetting. Still, such methods would have an inherent disadvantage because selecting modules based on the inputs would not allow for non-perceptual transfer.

HOUDINI possess advantages over the other lifelong learning methods discussed in Section 2.5. To begin with, the discussed replay-based and parameter regularisation methods use a fixed multi-head neural architecture. Sharing the same set of parameters for all tasks puts a limit on the number of different tasks that can be learned, which limits these algorithms' plasticity. Moreover, having to use the same parameters for each task can introduce harmful inductive bias which could lead to negative transfer. In contrast, HOUDINI can introduce new randomly initialised modules for a given task. This way it always has the capacity to learn a new task, and can also avoid negative transfer by simply choosing not to transfer anything, if no useful knowledge is available. A shared architecture also affects LML methods' stability since learning to perform a new task involves optimising all shared parameters. This introduces forgetting, which is mitigated to a different extent by different algorithms. In contrast, HOUDINI does not exhibit forgetting by design, as it freezes all modules' parameters after training. In terms of forward transfer, methods which share the same parameters for all tasks can potentially achieve both perceptual and non-perceptual transfer, however, the focus of preceding and most of following work remains on perceptual transfer. A disadvantage of HOUDINI, especially evident in perceptual transfer, is that freezing the modules does not allow it to adapt to slightly different input domains and might require a new module to be introduced. Instead, regularisation-based continual learning methods can adjust their parameters and account for the changes in the input distribution. When non-perceptual transfer needs to be done across disparate input domains, sharing the lower layers of a neural network could lead to negative transfer. In contrast, HOUDINI can introduce new lower modules, while reusing the higher ones. What is more, a multi-head architecture may not be suitable for few-shot transfer, as it introduces new parameters which need to be optimised on the new task. In comparison, HOUDINI can achieve few-shot transfer without introducing trainable parameters by re-arranging the available modules in a novel way in order to compute a new function. However, this is limited to LML settings in which the modules are learned in a way that makes them reusable, e.g. through a problem curriculum and/or through module architecture bias. One shortcoming of HOUDINI is that freezing the weights of modules in the library makes backward transfer impossible by design. In contrast, GEM (Lopez-Paz and Ranzato, 2017) exhibits some level of success in achieving backward transfer. In terms of required resources, regularisation-based methods' and replay-based methods' memory and computational requirements can remain constant with the number of solved problems. Similarly, HOUDINI can be implemented with a constant memory requirement by only loading modules from the library in the memory on demand. However, applying HOUDINI to long sequences can become prohibitively computationally expensive. This is because the search space of possible neural architectures and possible combination of modules can quickly become too big for our search method to be effective.

Considering architecture-based approaches to LML, Piggyback (Mallya et al., 2018) and PackNet (Mallya and Lazebnik, 2018) rely on the assumption that the underlying model's initial parameters are going to be relevant for all of the encountered tasks, which limits these methods' applicability to varied sequences. Our method does not share this limitation. HAT (Serra et al., 2018) focuses on avoiding catastrophic forgetting but also allows for graceful forgetting. Other than that, the method uses a fixed architecture with shared parameters at beginning of learning of each task, which is likely to lead to negative transfer. Progressive neural networks (Rusu et al., 2016) add new parameters for every task which makes them too memory-demanding. Moreover, they are limited to perceptual transfer and have been observed to exhibit negative transfer. Compacting, Picking and Growing (CPG) uses a heuristic and manually defined performance threshold to decide when to add new parameters. Unlike HOUDINI, it also incorporates a compacting phase which aims to restrict the size of the accumulated pre-trained parameters over time. PathNet (Fernando et al., 2017) is a modular approach to LML which was made available before the publication of our work. They maintain the same number of modules across all tasks which restricts their capacity to learn new problems. Moreover, it makes the method vulnerable to negative transfer, as observed in their results. In comparison, HOUDINI can better avoid negative transfer because it can always choose to introduce new modules if necessary. In terms of transfer, PathNet focuses perceptual transfer, while our work demonstrates that modular approaches can achieve non-perceptual and few-shot transfer. Moreover, we show that modular approaches can be used to transfer knowledge across different neural architecture. In terms of scalability, both PathNet's genetic search algorithm and HOUDINI's exhaustive search are unlikely to be effective in large search spaces which result from a large number of available modules.

Overall, one big difference of our approach, compared to previous LML methods is that we represent neural networks as functional programs, which allows us to search for a suitable neural architecture, as well as knowledge transfer. This increases the applicability of HOUDINI, allowing it to be used on sequences which require different neural architectures and have problems with different input spaces. Another advantage of our neurosymbolic approach is that it is more interpretable than the aforementioned approaches because the programs used to solve a problem can provide insight into which modules are reused and what tasks were these modules originally trained on.

Related LML work, developed after ours was published, has focused on developing modular approaches to LML which search for the best selection of pre-trained modules, given a user-specified neural architecture. MNTDP-D (Veniat et al., 2020), described in Chapter 2, is an efficient modular LML algorithm which requires that $L+1$ different modules be evaluated per problem, where L is the number of layers in the given neural architecture. The authors' approach aggressively restricts the search space which in turn limits MNTDP-D's transfer properties to only performing perceptual transfer. Ostapenko et al. (2021) present LMC - a modular LML algorithm which obtains a soft selection of pre-trained and new modules for each layer, using continuous optimisation. For this purpose, they model the output distribution of each module which is then used while obtaining the soft selection. The algorithm can achieve both perceptual transfer and few-shot transfer, however, is not capable of non-perceptual transfer. Moreover, computing a forward pass using a soft selection requires that all modules be stored in memory, which makes the algorithm very expensive in terms of memory and places a limit on the amount of pre-trained modules which can be accumulated. The authors also show evidence that LMC is outperformed by MNTDP on longer sequences. Further discussion on both methods is available in the next chapter.

HOUDINI searches for both an optimal neural architecture and an optimal selection of modules. At the same time, the field of neural architecture search (NAS), as described in Chapter 2, defines different ways to search for an optimal neural architecture. Focusing the discussion on the search for a suitable neural architecture, our search space is arguably more complex than the search spaces of the methods explored in Section 3.4. This is because the neural architectures which we consider have varying lengths and also can use different higher-order functions from functional programming to result in different patterns of computation. Still, these architectures are inherently compositional which is a property that could be exploited by a surrogate-based NAS method. HOUDINI uses types to significantly reduce the search space of all possible neural architectures, however, it then applies exhaustive search over the resulting search space. In contrast, NAS conduct a more sophisticated adaptive search. Exploring HOUDINI's restricted search space of type-correct neural architectures with a NAS approach is an interesting direction for future work. One potential challenge is that our LML setting requires that as few different neural architectures are evaluated as possible. In contrast, NAS methods can require evaluating a large number of different architectures before finding a solution. For instance, DARTS (Liu et al., 2018b) is orders of magnitude faster than other NAS techniques, and still takes one and a half days of runtime. Therefore, it is crucial to use an even more efficient NAS method.

4.2 Discussion

The presented work demonstrates that HOUDINI possesses most of the properties which are desired from a lifelong machine learning solution. While our work has limitations, they could be addressed in future work, which should lead to a general framework for lifelong machine learning.

HOUDINI freezes all pre-trained modules in order to prevent catastrophic forgetting. However, this leads to several shortcomings. First, it prevents backward transfer, as previous parameters are not modified. Second, not being able to finetune can lead to suboptimal results. For example, this could happen if we try to reuse a pre-trained module on a slightly different input distribution than the distribution it was trained on.

In this case, the module's weights might need to be adapted to the new problem. If the new data is small, HOUDINI would choose to use the suboptimal parameters. Otherwise, our approach would choose to train a new randomly initialised module. The latter case would result in the library having two modules which have very similar functionality. Therefore, the third shortcoming of freezing pre-trained modules is that it can introduce redundancies in the library.

One could address this by combining HOUDINI with a regularisation approach to continual learning. Whenever a pre-trained module is reused, one would need to augment the loss in order to reduce catastrophic forgetting. Current regularisation-based approaches have trouble scaling to a large sequence of problems. However, our framework selectively reuses pre-trained modules only on relevant problems. Therefore each module-specific regularisation would need to prevent catastrophic forgetting on only a fraction of all problems.

The main shortcoming of the work presented in this chapter is that it does not scale to a large number of problems. Currently, we use types to drastically reduce the program search space. However, the number of type-compatible programs grows exponentially with the number of pre-trained modules in the library. As a result, the currently used exhaustive search procedure becomes prohibitively computationally expensive for large sequences. The next chapter investigates how to address this limitation.

It is interesting to note that the HOUDINI programming language can be modified to describe some of the neurosymbolic approaches, discussed in Section 3.2. Remember that neurosymbolic approaches of category 1 are deep neural networks (DNNs), which operate on symbols and/or output symbols. Moreover, category 3 consists of DNNs, which approximate a symbolic system. Therefore, HOUDINI can easily represent neurosymbolic systems from these two categories. Category 4 consists of methods which combine a deep model and a symbolic system. The symbolic system can be added as a typed function inside our framework library. This could allow these neurosymbolic systems to be described by the HOUDINI programming language. Therefore, one can imagine extending our framework to lifelong learning of neurosymbolic methods.

Chapter 5

PICLE: A Probabilistic Framework for Modular Lifelong Learning

In the previous chapter we developed a neurosymbolic framework for modular lifelong machine learning (LML) and demonstrated that it can achieve a number of desirable properties. The main disadvantage of HOUDINI is that it does not scale to larger search spaces. Given a problem, it uses exhaustive search to search for a suitable modular neural architecture and a suitable selection of pre-trained modules to reuse. The number of pre-trained modules can grow linearly with the number of solved problems. As a result, the resulting search space is bounded by $O(nt^L)$ where n are the number of possible architectures, t is the index of the given problem and L is the maximum number of modules in the considered architectures. Therefore, as either quantity grows, it can quickly become prohibitively computationally expensive for exhaustive search to find a good solution. In this chapter, we assume that the neural architecture is fixed, which still results in a large search space bounded by $O(t^L)$. Our goal is to develop a modular LML algorithm that maintains the transfer learning properties of HOUDINI but also scales to large search spaces. Such an algorithm would be applicable to more realistic lifelong learning settings, involving much larger sequences of problems and more complex modular neural architectures which are comprised of more modules.

This chapter introduces PICLE, a probabilistic search framework for scalable modular LML. PICLE explores subsets of the search space using a probabilistic model to compute a probability distribution over the choice of pre-trained modules. We identify two important subsets of module combinations and show how to explore them efficiently using our framework. First, we identify a set of module combinations which transfer knowledge across similar input domains. We model each module's training input distribution and define a probabilistic model over the choice of pre-trained modules and the resulting hidden states for a given problem. Second, we identify of a set of module combinations which transfer knowledge across problems with disparate input distributions or different input spaces. We define a kernel between module combinations and show that applying our framework leads to Bayesian optimisation search. Finally, we show how the two search procedures can be combined and used to construct a scalable modular LML algorithm.

In order to assess our method, we develop a more challenging evaluation setting than the one in Chapter 4. To this end, we first define a list of desiderata for lifelong learning which places an emphasis on forward knowledge transfer. Second, we develop BELL, a benchmark suite which consists of sequences of compositional problems that each evaluate a different subset of the identified desired LML properties. The problems require a neural architecture which consists of 8 modules, so even for sequences of length 6, the search space for the last problem is upper bounded by $O(6^8) = O(1,679,616)$. Note that navigating this search space is challenging because evaluating most of its items involves training the parameters of a neural network.

We provide empirical evidence that our approach meets all of our desiderata, apart from backward transfer. It exhibits these properties across different input domains and neural architectures, which allows it to outperform competitive baselines.

5.1 Introduction

The lifelong machine learning (LML) setting calls for algorithms that can solve a sequence of learning problems (Thrun, 1998). Compared to solving the same problems separately, LML could lead to better generalisation performance on each problem as well a smaller memory footprint for the final model. An LML algorithm should avoid catastrophic forgetting — i.e., not allow later tasks to overwrite what has been learned from earlier tasks — and achieve transfer across a large sequence of problems. Ideally, the algorithm should be able to transfer knowledge across similar input distributions (*perceptual transfer*), dissimilar input distributions and different input spaces (*non-perceptual transfer*), and to problems with a few training examples (*few-shot transfer*).

Modular approaches (Valkov et al., 2018; Veniat et al., 2020; Ostapenko et al., 2021) have been proposed as a promising direction for LML. These approaches represent a neural network as a composition of modules, where each module is a parameterised function that is trained to perform an atomic transformation of its input and is reusable across tasks. Modular LML algorithms accumulate a library of diverse modules by solving the encountered problems in a sequence. Given a new problem, their goal is to find the most suitable combination of pre-trained and new modules, out of the set of all possible combinations, as measured by the performance on a held-out dataset. Unlike LML approaches which share the same parameters across all problems, modular algorithms can introduce new modules and, thus, do not have an upper bound on the number of solved problems.

However, *scalability* remains a key challenge in modular approaches to LML, as the set of module combinations is discrete and explodes combinatorially, rendering naive search strategies futile. It is important for an LML algorithm's resource demands to scale sub-linearly with the size of this set, because this would allow said algorithm to be applied to larger problem sequences as well as larger modular architectures. Prior work has sidestepped this challenge by introducing various restrictions. For example, by only handling perceptual transfer (Veniat et al., 2020) or by relaxing the search space and imposing an implicit limit on the number of pre-trained modules that can be accumulated (Ostapenko et al., 2021). The design of LML algorithms that relaxes these restrictions and can also scale remains an open problem.

In this work we present PICLE, a new probabilistic search framework for modular LML, in response to this challenge. For a given problem, PICLE divides the set of all module combinations into subsets of combinations with common properties. It then explores each subset using a subset-specific probabilistic model, designed to take advantage of the subset's properties. This model is used to compute a distribution over the choice of pre-trained modules, which can be queried efficiently without having to train new parameters. To apply our framework, we first develop a probabilistic model over a set of module combinations which can achieve perceptual and few-shot transfer. We design this model to take advantage of our insight that the input distribution on which a module is trained can indicate how successfully said module would process a set of inputs. Second, we identify a subset of module combinations capable of non-perceptual transfer and introduce a new kernel between them, allowing us to define a probabilistic model over this subset. The kernel is based on our insight that if the pre-trained modules of two paths compute similar functions, then said paths are likely to exhibit similar generalisation performance after their new modules are trained. Finally, we show that each of the two probabilistic models can be used to conduct separate searches through module combinations, which can then be combined into a scalable modular LML algorithm capable of perceptual, few-shot and non-perceptual transfer.

To evaluate our approach, we introduce a challenging evaluation setting for LML. To this end, we specify LML desiderata which differ from ones defined in previous work by distinguishing between different types of forward transfer. We then introduce a benchmark suite, BELL, which contains different sequences of problems which are designed to evaluate different subsets of LML properties. In contrast to a similar benchmark suite (Veniat et al., 2020), we use compositional tasks which allows us to define additional sequences which evaluate more LML properties.

5.2 Background

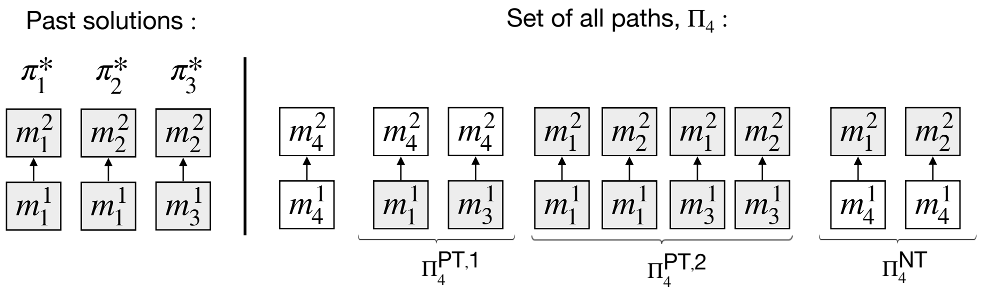

Past solutions :
Set of all paths, Π_4 :
π_1^*
π_2^*
π_3^*
m_1^2 m_2^2 m_2^2
m_1^1 m_1^1 m_3^1
m_4^2 m_4^1
m_4^2 m_4^2 m_1^1 m_3^1
m_1^2 m_2^2 m_1^2 m_2^2 m_1^1 m_1^1 m_3^1 m_3^1
m_1^2 m_2^2 m_4^1 m_4^1
$\Pi_4^{PT,1}$
$\Pi_4^{PT,2}$
Π_4^{NT}

Figure 5.1: This illustrates the set of all paths, Π_4, which a modular algorithm considers when solving the fourth problem Ψ_4 of some sequence, where the modular architecture has $L = 2$ layers. The library consists of all previously trained modules: $\Lambda_4 = \{m_1^1, m_3^1, m_1^2, m_2^2\}$, which are illustrated with a grey background in order to indicate that their parameters are frozen. Paths in $\Pi_4^{\text{PT},1}$ (Eq. 5.4) select a pre-trained module for the first layer, which allows them to perform perceptual transfer. Paths in $\Pi_4^{\text{PT},2}$ reuse modules in both layers. They can perform few-shot transfer since they only require a few examples, so that the correct path can be selected. Paths in $\Pi_4^{\text{NT},1}$ (Eq. 5.11) can achieve non-perceptual transfer by reusing a module in the second layer, allowing for it to be applied to a new input domain.

A lifelong machine learning algorithm $\mathcal{A}$ is tasked with solving a sequence of problems $S = (\Psi_1, ..., \Psi_T)$, usually provided one at a time. We consider the supervised setting, in which each problem is characterised by a tuple $\Psi = (\mathcal{D}, \mathcal{T})$, where $\mathcal{D}$ is the input domain, comprised of an input space and an input distribution, and $\mathcal{T}$ is a task, defined by a label space and a labelling function (Pan and Yang, 2009). an LML algorithm aims to transfer knowledge between the problems in a sequence in order to improve each problem's generalisation performance. The knowledge transfer to a single problem can be defined as the difference in an algorithm's performance, compared to when it is trained in the absence of the rest of the problems.

To maximise its performance, an LML algorithm should possess a number of desiderata. First, *plasticity* refers to an algorithm's ability to continuously learn to solve new problems and obtain at least as good generalisation performance as a standalone baseline. Plasticity can be negatively affected by a model's restricted capacity to learn new knowledge or by negative transfer. Second, *stability* is required in order to prevent *catastrophic forgetting*, which can occur when an algorithm's performance on a past problem decreases drastically. Third, *forward transfer* refers to an algorithm's ability to transfer knowledge to a newly encountered problem. We distinguish between three types of transfer: between problems with similar input distributions (*perceptual*), between problems with different input distributions or different input-spaces (*non-perceptual*) and to problems with a few training examples (*few-shot*). We note that knowledge transfer can involve transferring knowledge across similar problems which require different neural architectures, e.g. when performing transfer across different input spaces. Fourth, an LML algorithm should be capable of *backward transfer*, allowing it to improve its performance of previously encountered problems after solving new ones. Finally, an LML algorithm should be effective on large sequences of problems under constrained resources (*scalability*). In particular, the computational and memory requirements should scale sub-linearly with the number of encountered problems.

Modular approaches represent a deep neural network ζ_Θ as a composition of modules $\zeta_\Theta = m^L \circ ... \circ m^2 \circ m^1$, where the first module that is applied to the input is m^1. Each module m^i represents a nonlinear transformation, parameterised by $\theta(m^i)$. It can consist of one or more hidden layers, each with a potentially different type (e.g. convolutional or fully connected) and with a different activation function (e.g. ReLU or *tanh*). Given a problem, one needs to select optimal values for all parameters Θ and return the resulting neural network ζ_Θ as the solution.

Modular approaches have been successfully applied to lifelong learning. After solving $t-1$ problems, they accumulate a library of previously trained modules that each have been used to help perform one or more of the previously encountered tasks. The library $\Lambda_t = \bigcup_{i=1}^{L} \{m^i_j\}_{j<t}$ is a set of pre-trained modules, where m^i_j denotes a module in layer i that was trained on the j-th problem. It is then possible to construct different modular neural networks by either selecting a pre-trained module from the library or by training a new one from scratch for each of the L layers. We denote the set

of all modular neural networks as $\Pi_t = \{(\{m_j^1\}_{j<t} \cup \{m_t^1\}) \times ... \times (\{m_j^L\}_{j<t} \cup \{m_t^L\})\}$, where m_t^i denotes a new module used in layer i with randomly initialised parameters. We refer to each element of Π_t as a path, as it can be seen as guiding the forward computation through different modules. Figure 5.1 illustrates the set of all paths Π_4 for the fourth problem of some sequence, using a modular architecture with 2 module layers. Let a new problem Ψ_t be specified by a training and a validation datasets, $\mathbf{D}^{\text{tr}} = (\mathbf{X}^{\text{tr}}, \mathbf{Y}^{\text{tr}})$ and $\mathbf{D}^{\text{val}} = (\mathbf{X}^{\text{val}}, \mathbf{Y}^{\text{val}})$. To solve it, a modular LML algorithm searches for the path which leads to the best validation performance by optimising:

$$\begin{aligned} \pi_t^* &= \underset{\pi \in \mathbb{S}(\Pi_t)}{\operatorname{argmax}}\, p(\mathbf{Y}^{\text{val}} | \pi^{[\text{pre}]}, \pi^{[\text{new}]}_{\Theta^*_{\text{new}}}, \mathbf{X}^{\text{val}}) \\ \text{s.t. } \Theta^*_{\text{new}} &= \underset{\Theta_{\text{new}}}{\operatorname{argmax}}\, p(\mathbf{Y}^{\text{tr}} | \pi^{[\text{pre}]}, \pi^{[\text{new}]}_{\Theta_{\text{new}}}, \mathbf{X}^{\text{tr}}). \end{aligned} \tag{5.1}$$

Here $\pi^{[\text{pre}]}$ and $\pi^{[\text{new}]}$ denote the pre-trained and randomly initialised modules, respectively. If a path π contains new randomly initialised modules, their parameters $\Theta^*_{\text{new}} = \{\theta(m_j^i) : m_j^i \in \pi^{[\text{new}]}\}$ need to be trained before the path can be evaluated. In this case, said parameters are trained on the training dataset and the resulting model is evaluated on the validation dataset. As a result, evaluating a path can be computationally expensive and one wants to evaluate as few paths as possible. However, the search space grows quickly. If there are J pre-trained modules for each module type in the library, the search space consists of $(J+1)^L$ unique paths. This necessitates a search strategy $\mathbb{S}(\Pi_t)$ which can prioritise the most promising paths for evaluation. Under finite resources, the algorithm can evaluate only the first r paths suggested by the search strategy. Algorithm 1 summarises how the described modular LML algorithm solves a given problem. This algorithm, named MOLL, provides a template which can be combined different search strategies in order to create different modular LML algorithms.

Modular algorithms can achieve most of the LML desiderata (Valkov et al., 2018). Catastrophic forgetting is prevented by freezing the parameters of all pre-trained modules in the library. A path with only randomly initialised parameters can be selected, ensuring the methods' plasticity. As more problems are solved, the pre-trained modules in the library can be reused in order to achieve all types of forward transfer, as shown in Figure 5.1 and as discussed in the following sections. It is possible for two problems can require two different modular neural architectures which use the same modules for some of their layers. At the same time, modular LML algorithms can operate on different modular neural architectures for different problems, therefore, they can perform transfer across neural architectures. An additional benefit is that selected

Algorithm 1: MOLL

```
Input: $t$, Current problem's index
Input: $D_t^{\text{tr}}$, Training dataset
Input: $D_t^{\text{val}}$, Validation dataset
Input: $\Lambda_{t-1}$, Latest library
Input: $\mathbb{S}$, A search strategy for prioritising paths
Input: $r$, Number of paths that can be evaluated
Input: $\zeta$, The modular neural architecture
/* Use $\Lambda_{t-1}$ to generate the set of all paths.                    */
[1] $\Pi_t = \{(\{m_j^1\}_{j<t} \cup \{m_t^1\}) \times ... \times (\{m_j^L\}_{j<t} \cup \{m_t^L\})$
/* Evaluate the first $r$ paths recommended by $\mathbb{S}$ and select the
   one with the best performance on $D_t^{\text{val}}$                    */
[2]
    $\pi_t^* = \underset{\pi \in \mathbb{S}(\Pi_t)}{\text{argmax}}\, p(\mathbf{Y}^{\text{val}} | \pi^{[\text{pre}]}, \pi_{\Theta_{\text{new}}^*}^{[\text{new}]}, \mathbf{X}^{\text{val}})$
    $\text{s.t. } \Theta_{\text{new}}^* = \underset{\Theta_{\text{new}}}{\text{argmax}}\, p(\mathbf{Y}^{\text{tr}} | \pi^{[\text{pre}]}, \pi_{\Theta_{\text{new}}}^{[\text{new}]}, \mathbf{X}^{\text{tr}}).$
/* Update the library with the new modules                               */
[3] $\Lambda_t = \Lambda_{t-1} \cup \{m_t^i \text{ for } m_t^i \text{ in } \pi_t^*\}$
/* Return the found solution as well as the new library.                 */
[4] return $\zeta(\pi_t^*)$, $\Lambda_t$
```

paths can be interpreted in order to obtain insight into which problem(s) is knowledge being transferred from. However, a key challenge has been to retain all these transfer properties while using memory and computational resources that scale sub-linearly with the number of problems. To achieve this, one needs a search strategy which scales to enormous search spaces.

Prior to our work, Veniat et al. (2020) presented a scalable modular LML algorithm, MNTDP-D, whose search strategy always proposes a constant number of paths for evaluation when solving a problem. To achieve this, they severely restrict the search space which prevents them from achieving non-perceptual and few-shot transfer. Their approach requires that each previous solution is separately used to process the training data of a new problem, and that a K-nearest-neighbour classifier is fit to the extracted latent features. Therefore, technically, the computational requirement of this approach does not scale sub-linearly with the number of problems. However, this additional cost only becomes non-negligible in comparison to the rest, when the number of solved problems is very large, which is not a setting considered in current LML literature. Therefore, we still consider MNTDP-D to be scalable. In general, we consider a modular LML algorithm to be scalable if it can effectively achieve its transfer learning properties in large search spaces, while the number of modules which it simultaneously loads in memory and the number of paths which it explores both scale sub-linearly with the number of encountered problems.

5.3 PICLE: A Probabilistic Search Framework

We propose a probabilistic approach which uses the information available about the new problem to compute a probability distribution over different choices for pre-trained modules. The distribution does not model new modules which allows it to be queried efficiently, without having to train any new parameters.

Our approach is to split Π_t into subsets $\Pi_t^i \subset \Pi_t$ of paths in which the reused modules are at the same layer positions. For each subset, one can then define a probabilistic model and use it to compute a probability distribution over the choice of pre-trained modules $p(\pi^{[\mathrm{pre}]}|\mathbf{X}^{\mathrm{tr}}, \mathbf{Y}^{\mathrm{tr}}, \mathbf{X}^{\mathrm{tr}}, \mathbf{Y}^{\mathrm{val}}, E_j)$, where one can make use of the available datasets and the previously evaluated paths E_j. Computing this distribution can be done efficiently, as it does not involve randomly initialised modules. We can then define a search strategy for each path subset as selecting the unevaluated path with the highest posterior probability of its pre-trained modules:

$$\mathbb{S}_{\text{MAP}}(\Pi_t^i) = \left(\operatorname*{argmax}_{\pi \in \Pi_t^i} p(\pi^{[\text{pre}]} | \mathbf{X}^{\text{tr}}, \mathbf{Y}^{\text{tr}}, \mathbf{X}^{\text{tr}}, \mathbf{Y}^{\text{val}}, E_j) \right)_j \tag{5.2}$$

This allows us to define our search framework PICLE, which explores the set of all paths by combining the recommendations of $\mathbb{S}_{\text{MAP}}$ applied to different subsets:

$$\mathbb{S}_{\text{PICLE}} = \bigoplus_i \mathbb{S}_{\text{MAP}}(\Pi_t^i) \tag{5.3}$$

where $\bigoplus$ denotes the operation which is used to combine the recommendations. In this work, we simply concatenate the recommendations.

Next, we show how our framework can be used to create a scalable search strategy. First, we apply our probabilistic approach to subsets of paths which are capable of perceptual and few-shot transfer. Second, we apply it to a subset of paths capable of non-perceptual transfer. In both cases, we define a suitable probabilistic model, which takes advantage of each subset's properties. Finally, we show how the two search strategies can be combined. When used within Algorithm 1, this leads to a scalable LML algorithm which can achieve different types of transfer while always evaluating a constant number of paths per problem. Developing other probabilistic models necessary for the exploration of the remaining subsets of Π_t is left for future work.

5.4 Scalable Perceptual and Few-shot Transfer

In order to achieve perceptual transfer, a model needs to transfer knowledge on how to transform the input. For example, how to extract edges from natural images. For a path, this means reusing pre-trained modules for the first $l \in \{1, ..., L\}$ module layers. If all L layers are reused, this can allow for few-shot transfer, since there are no new parameters that need to be learned. Therefore, we identify subsets of paths $\Pi_t^{\text{PT},l} \in \Pi_t$ which can achieve perceptual transfer and few-shot transfer. In each path, the first l layers are selected from the library, while the rest are selected to be new randomly initialised modules, i.e.

$$\Pi_t^{\text{PT},l} = \{\pi : \pi = \{m_{<t}^i, i \leq l\} \cup \{m_t^i, i > l\}\}. \tag{5.4}$$

Figure 5.1 illustrates two examples of PT subsets, namely $\Pi_4^{PT,1}$ and $\Pi_4^{PT,2}$. Each subset grows polynomially with the size of the library and exponentially with the number of layers, which makes a naive search inapplicable when either of the two quantities is large. In this section, we use our probabilistic approach to devise a scalable search strategy.

5.4.1 Probabilistic model

In general, if a neural network is evaluated on inputs which are not sampled from its training distribution, its performance can degrade significantly (Csurka, 2017). We hypothesise that the density of an input under the model's training input distribution is indicative of how well the model can transform the input. We extend this idea to modules since they also are parameterised nonlinear transformations, each trained on some input distribution. Therefore, for a new problem and a given path, the density of each pre-trained module's inputs under said module's training input distribution should be indicative of the path's expected performance. For a PT path, the first l modules are pre-trained, which allows us to use them to process the training data and calculate the inputs which each of these modules needs to handle.

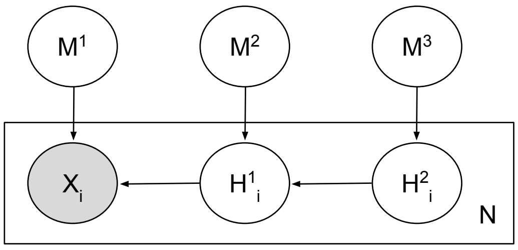


Figure 5.2: A graphical model depicting the joint distribution over the three pre-trained modules (M^1, M^2 and M^3) of a PT path $\pi \in \Pi^{\text{PT}, 3}$ and their inputs: X_i, H_i^1 and H_i^2 respectively.

Using this insight, for a given value of l, we define a probabilistic model over the choices of pre-trained modules and their N inputs (including X, the problem inputs, and H^i, the relevant hidden states). The graphical model for $l = 3$ is depicted in Figure 5.2 and the general joint probability distribution can be written as:

$$p(M^1,...,M^l,X_1,...,X_N,H_1^1,...,H_N^1,...,H_1^{l-1},...,H_N^{l-1}) \\ = \prod_{j=1}^{N}\left\{p(X_j|H_j^1,M^1)\prod_{i=2}^{l-1}\left[p(H_j^{i-1}|H_j^i,M^i)\right]p(H_j^{l-1}|M^l)\right\}\prod_{i=1}^{l}\left[p(M^i)\right]. \quad (5.5)$$

We can then model the posterior of a PT path of length l as only being dependent on the training inputs, $p(\pi^{[\text{pre}]}|\mathbf{X}^{\text{tr}}) := p(m^1,...,m^l|\mathbf{x}_1,...,\mathbf{x}_N)$, and express it in terms of quantities which we can approximate. To ease the presentation, and without loss of generality, we set the number of pre-trained modules $l = 3$. Next, we express the joint distribution in terms of quantities which we can approximate. We write

$$\begin{aligned} &p(M^1,M^2,M^3,X_1,...,X_N,H_1^1,...,H_N^1,H_1^2,...,H_N^2) \\ &\qquad = p(M^1)p(M^2)p(M^3)\prod_{j=1}^{N} p(X_j|H_j^1,M^1)p(H_j^1|H_j^2,M^2)p(H_j^2|M^3) \end{aligned}$$

where

$$p(X_j|H_j^1,M^1) = \frac{p(X_j,H_j^1,M^1)}{p(H_j^1,M^1)} = \frac{p(H_j^1|X_j,M^1)p(X_j|M^1)p(M^1)}{p(H_j^1)p(M^1)} = \frac{p(H_j^1|X_j,M^1)p(X_j|M^1)}{\sum_{m^{2\prime}} p(H_j^1|m^{2\prime})p(m^{2\prime})}$$

and

$$\begin{aligned} p(H_j^1|H_j^2,M^2) &= \frac{p(H_j^1,H_j^2,M^2)}{p(H_j^2,M^2)} = \frac{p(H_j^2|H_j^1,M^2)p(H_j^1|M^2)p(M^2)}{p(H_j^2)p(M^2)} \\ &= \frac{p(H_j^2|H_j^1,M^2)p(H_j^1|M^2)}{\sum_{m^{3\prime}} p(H_j^2|m^{3\prime})p(m^{3\prime})}. \end{aligned}$$

Therefore, the joint distribution can be expressed as:

$$\begin{aligned} &p(M^1,M^2,M^3,X_1,...,X_N,H_1^1,...,H_N^1,H_1^2,...,H_N^2) \\ &= p(M^1)p(M^2)p(M^3)\prod_{j=1}^{N} \frac{p(H_j^1|X_j,M^1)p(X_j|M^1)}{\sum_{m^{2\prime}} p(H_j^1|m^{2\prime})p(m^{2\prime})} \frac{p(H_j^2|H_j^1,M^2)p(H_j^1|M^2)}{\sum_{m^{3\prime}} p(H_j^2|m^{3\prime})p(m^{3\prime})} p(H_j^2|M^3). \end{aligned} \tag{5.6}$$

This expression contains three groups of distributions, which we need to define. First, we can define a prior over the choices of pre-trained modules for different module layers, $p(M^i)$. Second, we can approximate each module's training input distribution, leading to $p(H_j^i|m_{<t}^{i+1}) \approx q(H_j^i|m_{<t}^{i+1})$. The third group of distributions contains $p(H_j^i|H_j^{i-1},M^i)$ which is a distribution over the values of a hidden layer H^i given the previous hidden layer $\mathbf{h}^{i-1}$ and a module m^i. However, said hidden layer is given by the deterministic transformation $\mathbf{h}_j^i = m^i(\mathbf{h}_j^{i-1})$. Therefore, we can model $p(H_j^i|H_j^{i-1},M^i) = \delta(H_j^i - \mathbf{h}_j^i)$ using the Dirac delta function δ. This function has the property that $\int_{-\infty}^{\infty} f(z)\delta(z-$

$c)dz = f(c)$, which we use next in order to simplify the posterior. We write:

$$
\begin{aligned}
p(m^1,m^2,m^3|\mathbf{x}_1,...,\mathbf{x}_N) &= \frac{p(m^1,m^2,m^3,\mathbf{x}_1,...,\mathbf{x}_N)}{p(\mathbf{x}_1,...,\mathbf{x}_N)} \propto p(m^1,m^2,m^3,\mathbf{x}_1,...,\mathbf{x}_N) \\
&= \int_{-\infty}^{\infty} ... \int_{-\infty}^{\infty} p(m^1,m^2,m^3,\mathbf{x}_1,...,\mathbf{x}_N,\mathbf{h}_1^{'1},...,\mathbf{h}_N^{'1},\mathbf{h}_1^{'2},...,\mathbf{h}_N^{'2}) d\mathbf{x}_1...d\mathbf{h}_\mathbf{N}^{'} \\
= p(m^1)p(m^2)p(m^3) &\prod_{j=1}^{N} \frac{p(x_j|m^1)}{\sum_{m^{2'}} p(\mathbf{h}_j^1|m^{2'})p(m^{2'})} \frac{p(\mathbf{h}_j^1|m^2)}{\sum_{m^{3'}} p(\mathbf{h}_j^2|m^{3'})p(m^{3'})} p(\mathbf{h}_j^2|m^3) \\
\approx p(m^1)p(m^2)p(m^3) &\prod_{j=1}^{N} \frac{q(\mathbf{x}_j|m^1)}{\sum_{m^{2'}} q(\mathbf{h}_j^1|m^{2'})p(m^{2'})} \frac{q(\mathbf{h}_j^1|m^2)}{\sum_{m^{3'}} q(\mathbf{h}_j^2|m^{3'})p(m^{3'})} q(\mathbf{h}_j^2|m^3)
\end{aligned}
$$

In general, for l pre-trained modules, we approximate the numerator of the posterior using:

$$
\begin{aligned}
p(m^1,...,m^l|\mathbf{x}_1,...,\mathbf{x}_N) &\propto p(m^1,...,m^l,\mathbf{x}_1,...,\mathbf{x}_N) \\
&\approx \prod_{i=1}^{l} p(m^i) \prod_{j=1}^{N} \left\{ \prod_{i=1}^{l-1} \left[\frac{q(\mathbf{h}_j^{i-1}|m^i)}{\sum_{m^{i+1'}} q(\mathbf{h}_j^i|m^{i+1'})p(m^{i+1'})} \right] q(\mathbf{h}_j^{l-1}|m^l) \right\}
\end{aligned}
\tag{5.7}
$$

To compute this, we need to define a prior distribution over modules and approximate a module's training input distribution. These are detailed next.

5.4.1.1 Defining the prior

Computing Eq. 5.7 requires us to define a prior distribution over the choice of a pre-trained module, $p(M^i)$. Say two modules m_a^i and m_b^i are trained using two different paths on two different problems. Now, say that the model trained on problem a achieved $Acc(a)$ validation accuracy, while the model trained on problem b achieved a higher validation accuracy $Acc(b) = Acc(a) + d$, for $d > 0$. We hypothesise that the module, whose model achieved the higher accuracy after training, is likely to compute a more useful transformation of its input. Therefore, if m_a^i and m_b^i have a similar likelihood for a given set of training data points, we would give preference to using m_b^i. To this end, we define the prior distribution in terms of a module's original accuracy using the *softmax* function as follows:

$$
p(m_j^i) = \frac{e^{\frac{Acc(j)}{T}}}{\sum_{m_{j'}^i} e^{\frac{Acc(j')}{T}}}. \tag{5.8}
$$

Here T is the temperature hyperparameter which we compute as follows. Suppose that, for a given set of inputs $\{\mathbf{x}_1,...,\mathbf{x}_N\}$, we have selected the first $i-1$ modules and have computed the inputs to the ith module, $D^{H^{i-1}} = \{h_j^{i-1}\}_{j=1}^N$. Moreover, suppose that the likelihood of module m_a^i is slightly higher than the likelihood of m_b^i, i.e. that $p(D^{H^{i-1}}|m_a^i) > p(D^{H^{i-1}}|m_b^i)$. However, because the model of m_b^i was trained to a higher accuracy ($Acc(b) = Acc(a) + d$), we would like to give m_b^i preference over m_a^i. Therefore, we would like to set the hyperparameter T so that the posterior of the path using m_b^i is higher, i.e. $p(m_{<t}^1, ..., m_a^i|\mathbf{x}_1, ..., \mathbf{x}_N) < p(m_{<t}^1, ..., m_b^i|\mathbf{x}_1, ..., \mathbf{x}_N)$. Using Eq. 5.7 we can express this as:

$$
\begin{aligned}
p(m_{<t}^1, ..., m_a^i|\mathbf{x}_1, ..., \mathbf{x}_N) &< p(m_{<t}^1, ..., m_b^i|\mathbf{x}_1, ..., \mathbf{x}_N) \\
p(m_a^i)p(D^{H^{i-1}}|m_a^i) &< p(m_b^i)p(D^{H^{i-1}}|m_b^i) \\
\frac{p(m_a^i)}{p(m_b^i)} &< \frac{p(D^{H^{i-1}}|m_b^i)}{p(D^{H^{i-1}}|m_a^i)} \\
\frac{e^{\frac{Acc(a)}{T}}}{e^{\frac{Acc(a)+d}{T}}} &< \frac{p(D^{H^{i-1}}|m_b^i)}{p(D^{H^{i-1}}|m_a^i)} \qquad (5.9) \\
\frac{Acc(a)}{T} - \frac{Acc(a)+d}{T} &< \log \frac{p(D^{H^{i-1}}|m_b^i)}{p(D^{H^{i-1}}|m_a^i)} \\
T &> \frac{d}{\log p(D^{H^{i-1}}|m_b^i) - \log p(D^{H^{i-1}}|m_a^i)}.
\end{aligned}
$$

We can then use the inequality in Eq. 5.9 in order to determine the value T. To do this, one needs to decide how much difference in log likelihood should be overcome by a difference d in accuracy.

5.4.1.2 Approximating the likelihood

In order to compute Eq. 5.7, we need to approximate each module's training input distribution, $p(H^{i-1}|m_{<t}^i) \approx q(H^{i-1}|m_{<t}^i)$. Let m_c^i be a module in the ith layer trained on the c-th problem with inputs $\{\mathbf{x}_j\}_{j=1}^N$, using the path π_c^*. After training, we can compute this module's training inputs by processing the problem's inputs using the first $i-1$ modules of π_c^*. This way we compute a set of hidden activations $D^{\text{tr},H^{i-1}} = \{\mathbf{h}_j^{i-1}\}_{j=1}^N$ which can be used to approximate the module's training input distribution.

In this work, we use a multivariate Gaussian distribution because of its simplicity and efficiency. However, this has two disadvantages. First, for a module with inputs $h^{i-1} \in \mathbb{R}^d$ of dimensionality d, a Gaussian approximation would require $d^2 + d$ parameters, which could become undesirably large for high values of d. Second, if $N < d$ or if some of the input dimensions are not linearly independent, the sample covariance matrix would be rank-deficient (Mohammadi et al., 2016) which leads to an ill-defined density of the corresponding Gaussian distribution.

To address this, we assume that there are at least $k \ll d$ linearly independent dimensions. We then perform dimensionality reduction of each input $\mathbf{h}_j^{i-1}$ using a *random projection* with a randomly generated matrix $A^{i-1} \in \mathbb{R}^{k \times d}$ (Pedregosa et al., 2011). This projection has a number of useful properties. First, since A^{i-1} is a full column-rank matrix, it is highly likely that the k dimensions of the projected inputs will be linearly independent. Second, it has been shown that a random projection approximately preserves the distance between the projected data points (Johnson, 1984). Third, the matrix A^{i-1} is not data-dependant, so the same matrix can be reused to project points with the same dimensionality sampled from different input distributions. Fourth, the density of a projected data point $p(A^{i-1}\mathbf{h}_j^{i-1})$ is proportional to the average density of all the original data points that could be projected to it (Cunningham and Fiterau, 2021). Since the support of the data distribution usually lies on a manifold with a significantly lower dimension, it is unlikely that two different data points sampled from a data distribution will be projected to the same point. In this case the two densities would be proportional $p(\mathbf{h}_j^{i-1}) \propto p({}^{i-1}\mathbf{h}_j^{i-1})$ with a high probability.

Therefore, for a given set of hidden activations $D_j^{\text{tr},H^{i-1}} = \{\mathbf{h}_j^{i-1}\}_{j=1}^N$, representing the inputs that a module m_c^i was trained on, we approximate the projected input training distribution with a multivariate Gaussian $p(A^{i-1}H^{i-1}|m_c^i) \approx q(A^{i-1}H^{i-1}|m_c^i) = \mathcal{N}(\mu_k^i, \Sigma_k^i)$, where μ_k^i and Σ_k^i are respectively the sample mean and the sample covariance, computed using the projected module inputs in $D_j^{\text{tr},H^{i-1}}$. This addresses the challenge of singular sample covariance and each approximation requires much fewer parameters, $k^2 + k$. In our ablation experiments, we compare the utility of multivariate Gaussian approximations of the projected module inputs to that of multivariate Gaussian approximations over the original module inputs. Surprisingly, our results suggest that the former (our approximation) leads to a more reliable identification of the correct modules which should be selected to process some given set of inputs.

As a result of our approximations, our probabilistic model from Equation 5.5 becomes a model over the choice of modules and their projected inputs, modelling $p(M^1, \ldots, M^l, A^0X_1, \ldots, A^1H_1^1, \ldots, A^{l-1}H_N^{l-1})$. Because of the aforementioned proportionality of the likelihood, it is likely that the posterior distributions are proportional as well, i.e. $p(m^1, \ldots, m^l|\mathbf{x}_1, \ldots, \mathbf{x}_N) \propto p(m^1, \ldots, m^l|A^0\mathbf{x}_1, \ldots, A^0\mathbf{x}_N)$.

Having specified a probabilistic model and a way to compute the posterior distribution over pre-trained, we next show how to apply the search strategy defined in Eq. 5.2 to subsets of PT paths.

5.4.2 Search Strategy

The number of possible PT paths of length l increases exponentially with l. As a result, it is not computationally feasible to directly apply the search strategy $\mathbb{S}_{\text{MAP}}$ (Eq. 5.2) since it would require that $p(\pi^{[\text{pre}]}|\mathbf{X}^{\text{tr}})$ is evaluated for all PT paths of length l. Instead, we augment it with a greedy policy, such that after selecting the first l pre-trained modules, we freeze this selection and reuse the same l modules for PT paths which transfer more modules. As a result, $p(\pi^{[\text{pre}]}|\mathbf{X}^{\text{tr}})$ needs to be evaluated for at most $O(tL)$ paths. Moreover, for each value of l, the search only recommends the path which has the highest probability under our model for evaluation, since this is the path which is best equipped to handle the inputs. We define the augmented search strategy over all $\Pi_t^{\text{PT}} = \bigcup_{l=1}^{L} \Pi_t^{\text{PT},l}$ as:

$$\begin{aligned} &\mathbb{S}_{\text{G}}^{\text{PT}}(\Pi_t^{\text{PT}}) := \{\pi^{*\text{PT},l}\}_{l=1}^{L} \\ &\text{where } \pi^{*\text{PT},l} = \pi^{*\text{PT},l-1}[:l-1] \cup m^{*,l} \cup \{m_t^i\}_{i=l}^{L} \\ &\qquad m^{*,l} = \underset{m^l}{\text{argmax}}\, p(\pi^{*'\text{PT},l-1}[:l-1] \cup m^l|\mathbf{x}_1,\dots,\mathbf{x}_N) \end{aligned} \tag{5.10}$$

This search strategy always recommends a constant number of paths ($O(L)$ paths) for evaluation irrespective of the value of t and, thus, irrespective of the size of the search space. Moreover, it requires that only a constant number of modules be loaded in memory at a time. As a result, combining $\mathbb{S}_{\text{G}}^{\text{PT}}$ with Algorithm 1 would result in a scalable modular LML algorithm capable of perceptual and few-shot transfer.

5.5 Scalable Non-Perceptual Transfer

To achieve non-perceptual transfer, a model can use pre-trained modules for the last l module layers. This represents knowledge on how to transfer a latent representation of the input to a task-specific prediction. Therefore, we identify subsets $\Pi_t^{NT,l} \in \Pi_t$ which consists of paths capable of non-perceptual transfer. Each path has its first $L-l$ modules randomly initialised while the rest l modules are selected from a library:

$$\Pi_t^{\text{NT},l} = \left\{\pi_t^{\text{NT},l} : \pi_t^{\text{NT},l} = \{m_t^i\}_{i=1}^{l} \cup \{m_{<t}^i\}_{i=L-l+1}^{L}\right\}. \tag{5.11}$$

Figure 5.1 illustrates an example of a NT subset, namely $\Pi_4^{NT,1}$. Each subset grows polynomially with the size of the library and exponentially with l, which makes a naive search inapplicable when either of the two quantities is large. In this section we use our probabilistic approach to devise a scalable search strategy.

5.5.1 Probabilistic Model

For NT paths of length l, we define a probabilistic model over the choice of pre-trained modules, the labels of the validation dataset, the inputs of the validation dataset and the previous path evaluations E_j. The first two are treated as random variables while the latter two are treated as parameters. This allows us to express the posterior distribution over paths as:

$$p(\pi^{[\text{pre}]}|\mathbf{X}^{\text{val}},\mathbf{Y}^{\text{val}},\mathbf{E}_j) \propto p(\mathbf{Y}^{\text{val}}|\mathbf{X}^{\text{val}},\pi^{[\text{pre}]},\mathbf{E}_j)p(\pi^{[\text{pre}]}). \tag{5.12}$$

E_j is a set of tuples, each containing a previously evaluated NT path of length l and its performance on the validation dataset after its new parameters are trained.

We define a prior over $\pi^{[\text{pre}]}$ which assigns equal non-zero values only to pre-trained modules which have been used together to solve a previous problem. This reflects our prior assumption that using a novel combination of modules for non-perceptual transfer is unnecessary for the sequences which we consider. A similar prior is used in Veniat et al. (2020).

For the purposes of numerical stability, we approximate the log-likelihood, $\log p(\mathbf{Y}^{\text{val}}|\mathbf{X}^{\text{val}},\pi^{[\text{pre}]},\mathbf{E}_j)$ using a Gaussian process (GP), which is fit on previous path evaluations E_j. To enable this, we next define a kernel function between the pre-trained modules of two NT paths.

The difference between two NT paths π_i and π_j of length l is that their last l pre-trained modules compute different functions. We hypothesise that if said functions are similar, the two NT paths will exhibit similar performance after their random parameters are trained. To make use of this insight, we define an inner product between two vector-valued functions, $f : \Omega \to \mathbb{R}^r$ and $g : \Omega \to \mathbb{R}^r$ as: $\langle f\ ,\ g\rangle = \int_\Omega f(\mathbf{z}) \cdot g(\mathbf{z})d\mathbf{z}$. This allows us to compute the distance between two functions as:

$$d(f,g) := ||f-g|| = \sqrt{\langle f-g\ ,\ f-g\rangle} = \sqrt{\int_\Omega (f(\mathbf{z})-g(\mathbf{z}))\cdot(f(\mathbf{z})-g(\mathbf{z}))d\mathbf{z}}\ . \tag{5.13}$$

We approximate this value using Monte Carlo integration with a set of inputs, Z, from the functions' common input space. We can then define the kernel function between the chosen pre-trained modules of two same-length NT paths using the squared exponential kernel function and the distance between the functions computed by their last l modules:

$$\kappa(\pi_i^{[\text{pre}]},\pi_j^{[\text{pre}]};Z) = \sigma^2 \exp\left\{-(d(\pi_i^{[\text{pre}]},\pi_j^{[\text{pre}]};Z)^2)/(2\gamma^2)\right\} \tag{5.14}$$

where σ and γ are the kernel hyperparameters which are fit to maximise the marginal likelihood of a GP's training data (Rasmussen and Williams, 2006). To create Z for a value of l, we store a small number of samples from the input distribution of each pre-trained module at layer $L-l+1$. Given a new problem, we create Z by combining all the stored hidden activations. This allows us to compare functions on regions of the input space with higher density.

5.5.2 Search Strategy

In order to develop a suitable search strategy to search over NT paths, we modify $\mathbb{S}_{\text{MAP}}$ (Eq. 5.2) and apply it on a single subset $\Pi_t^{\text{NT},l_{\min}}$ where $l_{\min}$ is the user-defined minimum number of modules which need to be transferred in order to improve the performance. Due to our choice of prior, maximising the posterior using Eq. 5.12 is equal to maximising the log-likelihood over the paths with nonzero prior probability. However, our GP-based approximation provides a distribution over the log-likelihood's value, which reflects the uncertainty of the GP's prediction. To account for this, we use an acquisition function, namely Upper Confidence Bound (UCB) (Shahriari et al., 2015) which provides an optimistic prediction of the log-likelihood. The resulting search strategy resembles Bayesian Optimisation:

$$\mathbb{S}_{\text{BO-MAP}}(\Pi_t^{\text{NT},l_{\min}}) = \left(\underset{\pi \in \Pi_t^{\text{NT},l_{\min}}}{\operatorname{argmax}} \; UCB\left(\log p(\mathbf{Y}^{\text{val}} | \mathbf{X}^{\text{val}}, \pi^{[\text{pre}]}, \mathbf{E}_j)\right)\right)_j . \tag{5.15}$$

At each step j the search strategy updates the GP using the previously available path evaluations and then uses the GP to predict the log-likelihoods of all unevaluated paths. The path whose predicted distribution has the highest UCB value is selected as the most promising and recommended for evaluation. For the first two steps ($j \in \{1,2\}$) we use Equation 5.13 to select the NT paths whose pre-trained functions have the lowest average distance to others'. Our intuition is that these paths are the most related to others so their evaluation should be the most informative about the performance of other paths.

The GP makes it possible to detect when further improvement is unlikely, allowing us to perform early stopping. For this purpose we compute the Expected Improvement, $EI\left(\log p(\pi^{[\text{pre}]} | \mathbf{X}^{\text{val}}, \mathbf{Y}^{\text{val}}, \mathbf{E}_j)\right)$ of the path selected for evaluation at each step j. If it is lower than a certain threshold, the search strategy terminates and does not recommend any more paths for evaluation. EI-based early stopping has been previously suggested in Nguyen et al. (2017) and, similarly to Makarova et al. (2022), our preliminary experiments showed that it leads to fewer path evaluations, compared to using UCB for early stopping.

In the worst case, $\mathbb{S}_{\text{BO-MAP}}$ would propose $O(t-1)$ paths for evaluation which scales linearly with the number of solved problems. While early stopping reduces the number of evaluated paths, our experiments suggest said number still scales linearly with the number of problems. However, our ablation experiments demonstrate that our search strategy exhibits a good anytime performance. This means that our Bayesian optimisation algorithm is capable of quickly finding a path which achieves positive non-perceptual transfer. Therefore, it is possible to constrain $\mathbb{S}_{\text{BO-MAP}}$ to evaluating a constant number of paths and still achieve non-perceptual transfer across a long sequence of problems. Moreover, the strategy can be implemented in a way which requires only a constant number of modules to be stored in memory at a time. As a result, combining $\mathbb{S}_{\text{BO-MAP}}$ with Algorithm 1 would result in a scalable modular LML algorithm capable of non-perceptual transfer.

5.6 Combining The Two Search Strategies

Following our framework defined in Equation 5.3 we combine the two search strategies, $\mathbb{S}_{\text{G}}^{\text{PT}}$ (Eq. 5.10) and $\mathbb{S}_{\text{BO-MAP}}$ (Eq. 5.15). To do this, we simply invoke them consecutively, calling $\mathbb{S}_{\text{G}}^{\text{PT}}$ first since it always recommends a constant number of paths. This leads to the following search strategy:

$$\tilde{\mathbb{S}}_{\text{PICLE}}(\Pi_t^{\text{PT}} \cup \Pi_t^{\text{NT},l_{\min}}) = \mathbb{S}_{\text{G}}^{\text{PT}}(\Pi_t^{\text{PT}}) \cup \mathbb{S}_{\text{BO-MAP}}(\Pi_t^{\text{NT},l_{\min}}). \tag{5.16}$$

The resulting search strategy, $\tilde{\mathbb{S}}_{\text{PICLE}}$ benefits from the transfer learning properties of its constituent strategies. As discussed in earlier sections, both $\mathbb{S}_{\text{G}}^{\text{PT}}$ and $\mathbb{S}_{\text{BO-MAP}}$ can be used for recommending a constant number of paths, regardless of the size of the search space. Moreover, they require that only a constant number of modules be loaded in memory at a time. Therefore, $\tilde{\mathbb{S}}_{\text{PICLE}}$ can be combined with Algorithm 1 in order to develop a scalable modular LML algorithm which can perform perceptual, non-perceptual and few-shot transfer.

5.7 BELL: Benchmark suite for Lifelong Learning

In order to evaluate our ideas, we introduce *BELL* - a suite of benchmarks for evaluating the LML properties outlined in Section 5.2. We assume compositional tasks and then generate various problem sequences, with each evaluating one or two of the desired properties. Running an LML algorithm on all sequences then allows us to asses which LML properties are present and which are missing. This builds upon the CTrL benchmark suite, presented in Veniat et al. (2020), which defines different sequences of image classification tasks, namely S^{pl}, S^{-}, S^{out}, S^{in}, S^{+} and S^{long}. They evaluate plasticity, perceptual transfer, non-perceptual transfer, catastrophic forgetting, backward transfer and scalability. We define our sequences similarly but over problems with compositional tasks. This provides more flexibility in defining the types of knowledge transferred between problems. As a result, we introduce additional sequences which evaluate new LML properties (S^{sp} and S^{few}). We also introduce new more challenging sequences (S^{out*} and S^{out**}).

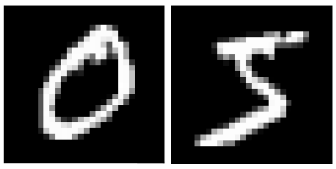


Figure 5.3: An example input for $\Psi = (D_{\text{MNIST}_1}, h_{\text{MNIST}_1}, g = (g^{(2)}_{\text{XOR}}, g^{(1)}_1))$. The 2-tuple of images are classified by h_{MNIST} and then are mapped to the coordinates (1, 6) by $g^{(1)}_1$ since they represent the first and sixth classes respectively. Afterwards, $g^{(2)}_{\text{XOR}}$ labels this input as 0, using the XOR pattern, shown in Fig. 5.4.

We assume that the task of each problem is compositional, i.e. that its labelling function can be decomposed into $f = g \circ h$. We then denote each problem as a 3-tuple, $\Psi_t = (D_j, h_j, g_k)$. Figure 5.3 presents an example of a problem where the input domain D_j is over 2-tuples of MNIST (LeCun et al., 2010) images; h_j maps a 2-tuple of images into a 2-tuple of classes; g_k maps a 2-tuple of classes to a Boolean value specifying whether or not the classes conform to some pattern. We can exploit the compositionality of the problems in order to evaluate different transfer learning properties. For instance, if two problems share the same D and h, then perceptual transfer can occur between them. If two problems share the same g, then non-perceptual transfer can occur between them. Finally, if a problem's 3-tuple combines previously encountered values for D_j, h_j and g_k, then this could allow for few-shot transfer.

The sequences we define rely on a pool of input domains and their input labelling sub-functions $\{(D_j, h_j)\}_j$ as well as a pool of upper labelling sub-functions $\{g_k\}_j$. Each of the following sequences is a pattern which specifies the number of problems, the common elements between the problems as well as the problems' dataset sizes. To evaluate an LML algorithm over one of the defined sequences, one randomly samples from the aforementioned pools and according to the sequence's pattern. Following Veniat et al. (2020), we set the sequence length of most sequences to 6 which, as we show in the experiments section, is sufficient for evaluating different LML properties. Specifically, we can still use these sequences to evaluate the scalability of a modular LML algorithm because our experiments use an 8-layered modular neural architecture which leads to a large search space in the last problem, upper bounded by $O(6^8 = 1679616)$. For a problem $\Psi_t = (D_j, h_j, g_k)$, we refer to h_j as the lower labelling sub-function and to g_k as the upper labelling sub-function. We use the indices j and k to indicate whether the corresponding labelling sub-function has occurred before in the sequence ($j < t$, $k < t$) or if it is new and randomly selected ($j = t$, $k = t$). For brevity, if $j = t$ and $k = t$, we don't write out the whole triple but only Ψ_t. By repeating previously encountered domains or labelling sub-functions, we can control what knowledge can be transferred in each of the define sequences. In turn, this allows us to evaluate different LML properties. We use Ψ^+ to indicate that the dataset generated for this problem is sufficient to learn a well generalising approximation without transferring knowledge. On the other hand, Ψ^- indicates that the LML algorithm cannot achieve good generalisation on this problem without transferring knowledge. Finally, Ψ^{--} indicates that the generated training dataset consists of only a few datapoints, e.g. 10. Next, we separately present each sequence, detailing which LML properties it evaluates.

Plasticity and **Stability**: The sequence $S^{pl} = [\Psi_1^+, \Psi_2^+, \Psi_3^+, \Psi_4^+, \Psi_5^+, \Psi_6^+]$ consists of 6 distinct problems, each of which has a different input domain and a different task. Moreover, each of the generated datasets has a sufficient number of data points as not to necessitate transfer. Therefore, this sequence evaluates an LML algorithm's ability to learn distinct problems, i.e. its plasticity. Moreover, this sequence can be used to evaluate an algorithm's stability by assessing its performance after training on all problems and checking for forgetting.

Forward Transfer: Most of our sequences are dedicated to evaluate different types of forward transfer. To begin with, in the sequence $S^- = [\Psi_1^+, \Psi_2^-, \Psi_3^-, \Psi_4^-, \Psi_5^-, \Psi_1^-]$ the first and the last datasets represent the same problem, however, the last dataset has fewer data points. Therefore, an LML algorithm would need to transfer the knowledge acquired from solving the first problem, thus, demonstrating its ability to perform forward transfer to a previously solved problem.

Perceptual Forward Transfer: We introduce three different sequences for evaluating perceptual transfer. First, in $S^{out} = [\Psi_1^+, \Psi_2^-, \Psi_3^-, \Psi_4^-, \Psi_5^-, \Psi_6^- = (D_1, h_1, g_6)]$ the last problem has the same input domain and input-processing target function h_1 as in problem 1. However, the last problem's dataset is small, therefore, an LML algorithm needs to perform perceptual transfer from the first problem, which is described by a large dataset. Second, $S^{out*} = [\Psi_1^-, \Psi_2^+ = (D_1, h_1, g_2), \Psi_3^-, \Psi_4^-, \Psi_5^-, \Psi_6^- = (D_1, h_1, g_6)]$ shares the same input distributions and lower labelling sub-function h_1 across problems Ψ_1, Ψ_2 and Ψ_6. Therefore, an LML algorithm needs to decide whether to transfer knowledge obtained from the first or from the second problem. Third, the sequence $S^{out**} = [\Psi_1^-, \Psi_2^+ = (D_1, h_1, g_2), \Psi_3^-, \Psi_4^-, \Psi_5^-, \Psi_1^-]$ is similar to the preceding one, with the distinction that the last problem is the same as the first. In this sequence, an LML algorithm needs to decide between reusing knowledge acquired from solving the same problem (Ψ_1), or to transfer perceptual knowledge from a more different problem (Ψ_2). Overall, these three sequences are designed to be increasingly more challenging in order to distinguish between different LML algorithms which are capable of perceptual transfer to a different extent.

Non-Perceptual Forward Transfer: Currently, we define two sequences to assess an algorithm's ability to transfer non-perceptual knowledge. Firstly, in $S^{in} = [\Psi_1^+, \Psi_2^-, \Psi_3^-, \Psi_4^-, \Psi_5^-, \Psi_6^- = (D_6, h_6, g_1)]$ the last problem has the same upper labelling sub-function as the first problem. However, the two problems' input distributions and lower labelling sub-functions are different. Therefore, an LML algorithm would need to transfer knowledge across different input domains. Secondly, the sequence $S^{sp} = [\Psi_1^+, \Psi_2^-, \Psi_3^-, \Psi_4^-, \Psi_5^-, \Psi_6^- = (D_6, h_6, g_1)]$ is simiarly defined, however, the input distribution of the last problem is also defined on a different input space from the input space of the first problem. Therefore, an algorithm would need to transfer knowledge across different input spaces.

Few-shot Forward Transfer: In order to evaluate this property, we introduce the following sequence, in which the first two problems are different from the rest of the sequences: $S^{few} = [\Psi_1^+ = (D_1, h_1), \Psi_2^+ = (D_2, h_2), \Psi_3^-, \Psi_4^- = (D_1, h_1, g_4), \Psi_5^-, \Psi_6^{--} = (D_2, h_2, g_4)]$. The labelling functions of the first two problems are simpler, each consisting only of a lower labelling sub-function. This is done in order to provide an LML algorithm with more supervision on how to approximate h_1 and g_1 more accurately.

The fourth problem Ψ_4 in this sequence then shares the same input domain and lower labelling sub-function as the first problem, but introduces a new upper labelling sub-function g_4. The last problem then shares the input domain and the lower labelling sub-function of Ψ_2, while also sharing the upper labelling sub-function of problem Ψ_4. Moreover, the last problem's training dataset consists of only a few data points. Therefore, an LML algorithm would need to reuse its approximations of h_2 and g_4 in a novel manner in order to solve the last problem.

Backward Transfer: The sequence $S^+ = [\Psi_1^-, \Psi_2^-, \Psi_3^-, \Psi_4^-, \Psi_5^-, \Psi_1^+]$ has the same first and last problem. However, the first dataset has significantly less data points than the last. Ideally, an LML algorithm should use the knowledge acquired after solving the last problem in order to improve its performance on the first problem. While this sequence represents a starting point for evaluating backward transfer, it is possible to introduce other sequences, representing more elaborate evaluations. For instance, introducing sequences which evaluate perceptual and non-perceptual backward transfer separately. However, as backward transfer is not the focus of this thesis, this is left for future work.

Scalability: This property can be evaluated using a long sequence of problems. For this purpose we define $S^{\text{long}} = [\Psi_i]_{i=1}^{60}$ which consists of 60 problems, each randomly selected with replacement from a set of problems. Most problems are represented by a small dataset, Ψ_t^-. Each of the first 50 problems has a $\frac{1}{3}$ probability of being represented by a large dataset, Ψ_t^+. Each problem also has a $\frac{1}{10}$ probability of being represented by an extra small dataset, Ψ_t^{--}. Overall, this sequence evaluates perceptual, non-perceptual and few-shot transfer on a longer sequence of problems.

The definitions of these sequences rely on two pools of (domains, lower labelling sub-functions) and upper labelling sub-functions, respectively. In turn, these can be used to create a set of problems. Next, we present a set of problems which can be used together with the aforementioned sequence definitions in order to evaluate LML algorithms.

5.7.1 Compositional Problems

To implement the sequences defined above, one needs to define a set of compositional problems. To this end, we define 9 different pairs of an input domain and a lower labelling sub-function, $\{(D_i, h_i)\}_{i=1}^{9}$. Moreover, we define 16 different upper labelling sub-functions $\{g_i\}_{i=1}^{16}$. These can be combined into a total of 144 different compositional problems.

First, we define 9 image multi-class classification tasks, which all share input and output spaces $\mathbb{R}^{28\times28} \rightarrow \mathbb{R}^8$, but each have a different input distribution D_i and a domain-specific labelling function h_i. Concretely, we start with image classification datasets which depict: digits (MNIST) (LeCun et al., 2010), fashion products (FMNIST) (Xiao et al., 2017), letters (EMNIST) (Cohen et al., 2017) and cursive Japanese characters (KMNIST) (Clanuwat et al., 2018). Since some of the classes in KMNIST have significantly fewer training data points, we only use the 33 classes with the following indices: $[0,1,2,4-12,15,17-21,24-28,30,34,35,37-41,46,47]$, as they have a sufficient number of associated data points. We split the image datasets into smaller 8-class classification datasets. We use i to denote the different splits of the same original dataset. For example MNIST$_1$ denotes the dataset consisting of the first 8 digits ($\{0,...,7\}$). There is no MNIST$_2$ because there are only 10 different classes available. As another example, EMNIST$_2$ represents the third split of EMNIST, corresponding to a classification task among the letters from 'i' to 'p'. Using this we end up with the following 9 image datasets: MNIST$_1$, FMNIST$_1$, $\{\text{EMNIST}_i\}_{i=1}^3$, $\{\text{KMNIST}_i\}_{i=1}^4$. For each of these image datasets, we set aside 4800 validation images from the training dataset. We also keep the provided test images separate.

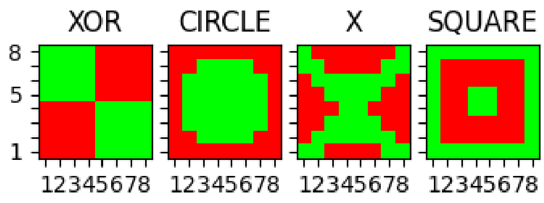


Figure 5.4: An illustration of the four two-dimensional patterns which are used by the four $g^{(2)}$ functions to label the input coordinates. Green indicates a positive label, and red indicates a negative label.

Second, we define a set of binary classification tasks, which map $\mathbb{R}^{16} \rightarrow \{0,1\}$. Each task's labelling function g_i receives two concatenated 8-dimensional one-hot encodings and returns a binary value, indicating if the given combination of 2 classes, represented by the input, fulfils a certain criteria. We further decompose the labelling function into $g_i(\mathbf{x}) = g_k^{(2)}(g_j^{(1)}(\mathbf{x}[:8]), g_j^{(1)}(\mathbf{x}[8:]))$.
Here, $g_j^{(1)}$ maps a one-hot encoding to an integer between 1 and 8. For instance, $g_1^{(1)}$ maps the first dimension to 1, the second to 2 and so on. As a result, we use $g^{(1)}$ to convert the initial input of two one-hot encodings to two-dimensional coordinates. We

define 4 different $g^{(1)}$ mappings, where $g_1^{(1)}$ is defined as above, and $g_1^{(2)}$, $g_1^{(3)}$ and $g_1^{(4)}$ each map the dimensions to a different randomly selected integer between 1 and 8.
At the same time, each $g_k^{(2)} : \mathbb{R}^2 \to \{0,1\}$ outputs whether a given two-dimensional coordinate is a part of a certain pattern or not. We define 4 different $g^{(2)}$ functions, each corresponding to one of 4 two-dimensional patterns, shown in Fig 5.4. In total, these functions need to label $8 * 8 = 64$ different two-dimensional coordinates.
We fuse the 4 different $g^{(1)}$ functions with the 4 different $g^{(2)}$ functions to define 16 different g functions:

$$\{g_{(k-1)*4+j}(\mathbf{x}) = g_k^{(2)}(g_j^{(1)}(\mathbf{x}[:8]), g_j^{(1)}(\mathbf{x}[8:])), k \in \{1,2,3,4\}, j \in \{1,2,3,4\}\}.$$

Finally, we can combine our 9 image classification datasets $\{(D_i, h_i)\}_{i=1}^{9}$ with our 16 binary classification tasks, in order to create 144 compositional problems $\{\Psi_{(k-1)*9+j} = (D_k, h_k, g_j), k \in \{1,...,9\}, j \in \{1,...,16\}\}$. The input to a problem $\Psi_i = (D_k, h_k, g_j)$ are two images sampled from D_i. Each image is labelled by h_i, each resulting in an eight-dimensional one-hot encoding of the corresponding image's class. The two one-hot encodings are then concatenated and labelled by g_j, which results in a binary label. An example for $\Psi = (D_{\text{MNIST}_1}, h_{\text{MNIST}_1}, g = (g_{\text{XOR}}^{(2)}, g_1^{(1)}))$ is shown in Fig 5.3.

Sequence S^{sp} involves transferring across input spaces by having its last problem's input domain be defined over a different input space. To create this domain we flatten any randomly selected domain from $\mathbb{R}^{28\times 28}$ to $\mathbb{R}^{784}$. This loses the images' spacial information and requires that a different neural architecture is applied to process those inputs.

5.7.2 Realising the sequences

To implement a sequence S of length l, we need to select l concrete compositional problems which fit the pattern specified by said sequence. Let the sequence have $l^{(1)}$ different pairs of image domain and lower labelling sub-function, and $l^{(2)}$ different upper labelling sub-functions. For all sequences, apart from S^{long}, we select $l^{(1)}$ pairs of (D_i, h_i) by sampling from the set of all possible image classification tasks, without replacement. Similarly, we select $l^{(2)}$ different upper labelling sub-functions by sampling without replacement from the set of available binary classification tasks $\{g_i\}_{i=1}^{16}$. For S^{long}, we use sampling with replacement.

If a problem's training dataset needs to be large, Ψ_i^+, we generate it according to the triple $n_{\text{tr}}^+ = (30000, \text{All}_{\text{tr}}, \text{All})$. The first value indicates that we generate 30000 data points in total. The second value indicates how many unique images from the ones set aside for training, are used when generating the inputs. In this case, we use all the available training images. The third value indicates how many out of the 64 unique two-dimensional coordinates, used by the upper labelling sub-function, are represented by the input images. In this case, we use all two-dimensional coordinates.

Some of the problems' training datasets are required to be small and to necessitate transfer. For sequences $S^{-}, S^{\text{out}}, S^{\text{out*}}, S^{\text{out**}}, S^{\text{few}}, S^{+}$, we generate the training datasets of each problem Ψ^{-} using the triple $n^{-}_{\text{tr}} = (10000, 100, \text{All})$. This way, only 100 unique images are used to generate the training dataset, so solving the problem is likely to be difficult without perceptual transfer. The subset of unique images is randomly sampled and can is different between two problems which share an input domain. For sequences S^{in} and S^{sp}, which evaluate non-perceptual transfer, we use the triple $n^{-}_{\text{tr}} = (10000, \text{All}_{\text{tr}}, 30)$. As a result, the generated datasets will only represent $30/64$ of the two-dimensional coordinates, which is not sufficient for learning the underlying two-dimensional pattern. Therefore, these problems will necessitate non-perceptual transfer. When generating a dataset for a problem Ψ^{-} in the sequence S^{long}, we randomly choose between the two, namely between $(10000, 100, \text{All})$ and $(10000, \text{All}_{\text{tr}}, 30)$.

For the problems in which the training dataset needs to contain only a few data points, Ψ^{--}, we use the triple $n^{--}_{\text{tr}} = (10, 20, 10)$. This creates only 10 data points, representing 20 different images and 10 different two-dimensional patterns.

For problems with Ψ^{--}, we use the triple $n^{--}_{\text{val}} = (10, 20, 10)$ for generating the validation dataset. For the rest of the problems, we use the triple $n^{--}_{\text{val}} = (5000, \text{All}_{\text{val}}, \text{All})$. Finally, we generate all test datasets using the triple $n^{--}_{\text{test}} = (5000, \text{All}_{\text{test}}, \text{All})$.

5.8 Experiments

Our experiments evaluate our approaches' ability to achieve the LML desiderata which we specified in Section 5.2. We are interested in our approaches' ability to scale to large search spaces and still achieve perceptual, non-perceptual and few-shot transfer. Furthermore, we assess their applicability to disparate input domains and neural architectures. For this purpose, we perform experiments on our BELL benchmark suite as well as on the CTrL (Veniat et al., 2020) benchmark suite. Finally, we conduct ablation experiments in order to verify our design choices.

We combine the search strategies which we defined earlier with Algorithm 1 in order to derive three modular LML algorithms: EMO which can perform perceptual and few-shot transfer using $\mathbb{S}_{\mathrm{G}}^{\mathrm{PT}}(\Pi_t^{\mathrm{PT}})$ (Eq 5.10); NOMO which can perform non-perceptual transfer using $\mathbb{S}_{\text{BO-MAP}}(\Pi_t^{\mathrm{NT},l_{\min}})$ (Eq. 5.2); PICLE which can perform all three of the aforementioned types of transfer using $\tilde{\mathbb{S}}_{\mathrm{PICLE}}(\Pi_t^{\mathrm{PT}} \cup \Pi_t^{\mathrm{NT},l_{\min}})$ (Eq. 5.16). Since the focus of this chapter is on augmenting modular LML methods, we compare our algorithms to a number of competitive modular LML baselines. First, we define MCL-RS which randomly selects paths from Π_t. A similar modular LML algorithm is described in Rajasegaran et al. (2019). Moreover, random search has been shown to be a competitive baseline in high-dimensional search spaces (Wang et al., 2013) and for neural architecture search (Li and Talwalkar, 2020). Second, we evaluate HOUDINI with a fixed neural architecture in order to keep the results comparable. Third, we evaluate MNTDP-D (Veniat et al., 2020) which is a scalable modular LML algorithm, capable of perceptual transfer. The algorithm evaluates a constant number L of different paths (where L is the number of module layers) and is detailed in Section 2.5.2.3. MTNDP-D has been shown to outperform parameter regularisation LML methods (Kirkpatrick et al., 2017) and rehearsal-based methods (Chaudhry et al., 2019a) on the CTrL benchmark suite. We selected MNTDP-D instead of its stochastic version MNTDP-S because the authors showed that the former achieves better performance. Finally, we also evaluate the standalone baseline (SA), which trains a new model on each problem. This can demonstrate the performances which can be achieved without knowledge transfer.

We use the same set of hyperparameters for our approaches across both benchmark suites, which suggests that this choice of hyperparameters is robust and applicable to different problems and neural architectures. For EMO and PICLE we project the hidden states to 20 dimensions before approximating the resulting distribution. Moreover, we use 0.001 as the softmax temperature parameter for the prior distribution over pre-trained modules. For NOMO and PICLE we store 40 of the training inputs of each of the pre-trained modules used in the $(L - l_{\min} + 1)$th layer. The value for $l_{\min}$ is 3 for both the BELL and the CTrL benchmarks. The jitter of the expected improvement is 0.001 and the threshold used for early stopping, is 0.001. To calculate the value of the upper confidence bound (UCB), we set the hyperparameter $\beta = 2$ in order to encourage exploration over exploitation. Finally, we use a problem-specific random seed to make the training process deterministic, so that the difference in performance can be accredited only to the LML algorithm, and not to randomness introduced during training.

For each baseline, we assess the performance on a held-out test dataset and report the average accuracy of the final model across all problems, $\mathbb{A}$, as well as the amount of forward transfer on the last problem, Tr^{-1}, computed as the difference in accuracy, compared to the standalone baseline.

All experiments are implemented using PyTorch 1.11.0 (Paszke et al., 2019). We also use GPy's (GPy, 2012) implementation of a Gaussian process. We run each LML algorithm on a single sequence, on a separate GPU. All experiments are run on a single machine with two Tesla P100 GPUs with 16 GB VRAM, 64-core CPU of the following model: "Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz", and 377 GB RAM.

5.8.1 BELL benchmarks

We first evaluate our algorithms and the baselines on the benchmark suite which we introduced in Section 5.7. In order to reduce the effect of randomness on the results, we create 3 versions of each sequence by randomly selecting different compositional problems. Then, for each sequence, we report the measurements, averaged over these 3 versions. As discussed next, for a sequence of length 6, the search space of possible paths is upper bounded by $O(6^8 = 1679616)$, making it possible to evaluate the scalability of the different baselines.

5.8.1.1 Neural Architecture

Here, we present the minimal neural architecture which we have found to be suitable for solving the problems we use in BELL. We arrived at this architecture by evaluating the performance of the SA baseline on randomly selected problems with a large amount of training data.

We first define a convolutional neural network $\zeta_{\text{CNN}} : \mathbb{R}^{28\times 28} \rightarrow \mathbb{R}^8$, suitable for processing images from the image classification datasets. We use a 5-layer architecture with *ReLU* hidden activations and a *softmax* output activation. The layers are as follows: *Conv2d(input_channels=1, output_channels=64, kernel_size=5, stride=2, padding=0)*, *Conv2d(input_channels=64, output_channels=64, kernel_size=5, stride=2, padding=0)*, *flatten*, *FC(4*4*64, 64)*, *FC(64, 64)*, *FC(64, 10)*. Here, *Conv2d* specified a two-dimensional convolutional layer and *FC* specifies a fully-connected layer.

Second, we define a fully-connected neural network for processing a concatenation of two 8-dimensional one-hot embeddings, $\zeta_{\text{MLP}} : \mathbb{R}^{16} \rightarrow \mathbb{R}^1$. It consists of 2 *FC* hidden layers with 64 hidden units and *RELU* hidden activations, followed by an output

FC layer with a *sigmoid* activation.

For a compositional problem $\Psi_k = (D_i, h_i, g_j)$ the input is a 2-tuple of images, $(\mathbf{x}^1, \mathbf{x}^2)$ and the expected output is a binary classification. We solve it using the architecture $\zeta_{\text{comp}} = \zeta_{\text{MLP}}(\mathit{concatenate}(\zeta_{\text{CNN}}(\mathbf{x}^2), \zeta_{\text{CNN}}(\mathbf{x}^2)))$. This architecture processes each of the 2 input images with the same ζ_{CNN} model. Then the 2 outputs are concatenated and processed by a ζ_{MLP} model.

We represent this as a modular neural architecture by considering each of the 8 parameterised nonlinear transformations to be a separate module. This increases the number of possible paths for each problem. As a result, for the 6th problem in a sequence, the number of possible paths is upper bounded by $O(6^8 = 1679616)$. Therefore, in this setting, even sequences of length 6 are challenging for modular LML approaches.

The input space of the last problem in the sequence S^{sp} is given by a flattened vector with 784 dimensions. Therefore, only for this problem, we replace ζ_{CNN} with a different architecture, ζ_{FC}, which consists of two fully connected layers, with a hidden size of 64, uses *ReLU* as a hidden activation and *softmax* as its output activation.

We train new parameters to increase the log-likelihood of the labels using the AdamW optimiser (Loshchilov and Hutter, 2017) with 0.00016 learning rate, and 0.97 weight decay. The training is done with a mini-batch size of 32 and across 1200 epochs. We apply early stopping, based on the validation loss. We stop after 6000 updates without improvement and return the parameters which were logged to have had the best validation accuracy during training.

5.8.1.2 Results

The results for the sequences which do not involve non-perceptual transfer are presented in Table 5.1. We present EMO and PICLE in the same column as they exhibit the same performance on these sequences. Overall, it can be observed that PICLE outperforms the rest of the baselines on all sequences, apart from S^+, with MNTDP-D coming second. PICLE's advantage is the biggest on S^{few} and $S^{\text{out**}}$ where we achieve significant improvement over all other methods. The low amount of transfer which MCL-RS and HOUDINI achieve on the last problems highlights the fact that naive search strategies are not suitable for large search spaces. We next review these results in more detail.

To begin with S^{pl} consists of 6 problems, represented by a large dataset and evaluates an algorithm's ability to learn a large number of problems without catastrophic

	SA	MCL-RS	HOUDINI	MNTDP-D	EMO, PICLE
$\mathbb{A}(S^-)$	73.88	76.67	79.59	81.67	**81.92**
$Tr^{-1}(S^-)$	0.	17.22	34.27	**34.29**	**34.29**
$\mathbb{A}(S^{\text{out}})$	74.25	76.16	74.40	77.95	**78.15**
$Tr^{-1}(S^{\text{out}})$	0.	5.64	0.	**15.41**	**15.41**
$\mathbb{A}(S^{\text{out*}})$	72.27	73.39	72.27	75.48	**75.72**
$Tr^{-1}(S^{\text{out*}})$	0.	0.43	0.	**12.53**	**12.53**
$\mathbb{A}(S^{\text{out**}})$	71.51	73.85	71.75	73.71	**75.73**
$Tr^{-1}(S^{\text{out**}})$	0.	4.61	1.46	1.74	**12.04**
$\mathbb{A}(S^{\text{few}})$	75.47	78.14	80.82	82.18	**88.12**
$Tr^{-1}(S^{\text{few}})$	0.	5.87	4.54	11.42	**46.07**
$\mathbb{A}(S^{\text{pl}})$	93.61	93.63	93.61	93.72	**93.79**
$Tr^{-1}(S^{\text{pl}})$	0.	0.	0.	**00.20**	**00.20**
$\mathbb{A}(S^+)$	73.61	**75.08**	73.61	74.54	74.49
$Tr^{-1}(S^+)$	0.	0.	0.	0.	0.

Table 5.1: The results on our compositional benchmarks which do not evaluate non-perceptual transfer. PICLE outperforms the rest of the baselines on all sequences, apart from S^+.

forgetting. Forgetting is not evaluated, since all of the baselines avoid it by design. In terms of plasticity, all baselines achieve equal or better performance than the standalone baseline. Moreover, our results indicate that knowledge transfer is still possible in this sequence, with PICLE achieving the highest average accuracy.

The sequence S^{-} evaluates knowledge transfer to an already encountered problem, represented by a much smaller dataset upon the second encounter. Both MNTDP-D and PICLE achieve the highest transfer on the final problem, but PICLE also achieves a higher average accuracy across problems. This indicates that our algorithm can more often find paths which lead to knowledge transfer.

The sequence S^{out} evaluates an algorithm's perceptual transfer between two problems with the same input distributions. Both MNTDP-D and PICLE achieve the best amount of transfer on the last problem, but again PICLE attains a higher average accuracy.

$S^{\text{out*}}$ is a more challenging sequence for evaluating perceptual transfer, in which the first two problems share the same input distribution as the last problem. Therefore, the algorithms needs to select the correct problem to transfer from. Surprisingly to us, both MNTDP-D and PICLE are equally successful in achieving the highest perceptual transfer on the last problem. Still, PICLE attains a higher average accuracy across problems.

$S^{\text{out**}}$ is an even more challenging sequence for evaluating perceptual transfer. The first and the last problems are the same and are both given by a small dataset. The second problem, represented by a large dataset, shares the same input distribution as the first and the last problems, but has a different labelling function. Therefore, an algorithm needs to transfer knowledge from the second problem to the last problem, even though the first and the last problems are the same. Our results demonstrate that PICLE achieves the highest perceptual transfer on the last problem, with a significant advantage over the other baselines. Our prior distribution over modules, which prioritises modules trained to a higher accuracy, helps us select the correct problem to transfer from. At the same time, MNTDP-D is outperformed by random search (MCL-RS).

S^{few} evaluates a modular algorithm's ability to recompose previously trained modules in a novel way in order to solve a previously unseen problem. Once again, our results indicate that PICLE significantly outperforms the baselines. On the last problem, it achieves 34.65 higher accuracy than the second-best, MNTDP-D. This demonstrates our algorithm's ability to achieve few-shot transfer.

	SA	MCL-RS	HOUDINI	MNTDP-D	EMO	NOMO	PICLE
$\mathbb{A}(S^{\text{in}})$	89.01	90.85	89.32	90.62	90.26	92.20	**92.82**
$Tr^{-1}(S^{\text{in}})$	0.	1.81	11.04	9.70	7.61	18.89	**22.28**
$\mathbb{A}(S^{\text{sp}})$	87.94	92.22	**92.99**	87.94	87.92	91.92	91.93
$Tr^{-1}(S^{\text{sp}})$	0.	25.68	**30.27**	0.	0.	23.65	23.65

Table 5.2: The results on our compositional benchmarks which evaluate non-perceptual transfer. The results demonstrate that PICLE is able to perform non-perceptual transfer across problems with different input distributions and different input spaces.

S^{+} evaluates an algorithm's ability to achieve backward knowledge transfer from the last problem to the first. However, none of the competing methods are capable of backward transfer by design. Surprisingly, MCL-RS is able to find the best average accuracy across problems.

Our results on sequences which evaluate non-perceptual transfer, presented in Table 5.2, demonstrate that $\mathbb{S}_{\text{BO-MAP}}$ enables NOMO and PICLE to transfer knowledge across different input distributions and input spaces. At the same time EMO and MNTDP-D can only achieve perceptual transfer on the last problem which leads to their significantly smaller performance. PICLE is able to achieve the highest performance on S^{in}. Surprisingly, it achieves a higher amount of knowledge transfer on the last problem than NOMO. Upon investigation, we found that for one of the three random realisations of S^{in}, it was better to transfer perceptual knowledge, than to transfer non-perceptual knowledge to the last problem, which PICLE was able to take advantage of. In S^{sp}, the different input space of the last problem necessitates a different modular architecture for the first 5 modules, resulting in a much smaller search space, $O(6^3 = 216)$. This allows non-scalable approaches, namely MCL-RS and HOUDINI, to also be effective on this sequence. Lastly, PICLE's performance demonstrates that combining $\mathbb{S}_{G}^{\text{PT}}$ and $\mathbb{S}_{\text{BO-MAP}}$ leads to better performance on sequences that allow for both perceptual and non-perceptual transfer.

Finally, we evaluated our approach on S^{long} - a sequence of 60 problems. EMO achieved $+7.37$ higher average accuracy than the standalone baseline demonstrating its ability to achieve perceptual transfer on a long sequence of problems. MNTDP-D achieved $+8.83$ higher average accuracy than SA which confirmed its scalability. Finally, PICLE performed the best, attaining $+12.25$ higher average accuracy than SA.

This shows that our approach can successfully attain perceptual and non-perceptual transfer across a long sequence of problems.

5.8.2 CTrL benchmarks

The CTrL benchmark suite was introduced in Veniat et al. (2020). They define a number of sequences, based on seven image classification tasks, namely: CIFAR10 and CIFAR100 (Krizhevsky et al., 2009), DTD (Cimpoi et al., 2014), SVHN (Netzer et al., 2011), MNIST (LeCun et al., 1998), RainbowMNIST (Finn et al., 2019), and Fashion MNIST (Xiao et al., 2017). All images are rescaled to 32x32 pixels in the RGB color format. CTrL was first to introduce the following sequences: S^{-}, S^{+}, S^{in}, S^{out}, S^{pl} and S^{long}, which are defined similarly to our definitions. However, the difference is that they are defined for and implemented by image classification tasks. The last task in S^{in}, which evaluates non-perceptual transfer, is given by MNIST images with a different background color than the first task. The last task in S^{out} is given by shuffling the output labels of the first task. S^{long} has 100 tasks. For each task, they sample a random image dataset and a random subset of 5 classes to classify. The number of training data points is sampled according to a distribution that makes it more likely for later tasks to have small training datasets. In contrast to BELL, they use only 1 selection of tasks for each sequence, i.e. 1 realisation of each sequence. To generate the sequences, we use the code provided by the authors (Veniat and Ranzato, 2021).

Our experimental setup mirrors the one used in Veniat et al. (2020), except that we make the training process deterministic, as discussed above. The neural architecture used is a small variant of ResNet18 architecture which is divided into 6 modules, each representing a different ResNet block (He et al., 2016). While the paper presenting the CTrL benchmark states that 7 modules are used, we used the authors' code (Veniat, 2021) for this method which specifies only 6 modules with the same total number of parameters. The difference from the architecture stated in the paper is that the output layer is placed in the last module, instead of in a separate module. All parameters are trained to reduce the cross-entropy loss with an Adam optimiser Kingma and Ba (2014) with $\beta_1 = 0.9$, $\beta_1 = 0.999$ and $\varepsilon = 10^{-8}$. For each task, each path is evaluated 6 times with different combinations of values for the hyperparameters of the learning rate ($\{10^{-2}, 10^{-3}\}$) and of the weight decay strength $\{0, 10^{-5}, 10^{-4}\}$. The hyperparameters which lead to the best validation performance are selected. Early stopping is

	SA	MNTDP-D	EMO	NOMO	PICLE
$\mathbb{A}(S^{\text{in}})$	58.77	61.36	61.78	**63.41**	63.10
$Tr^{-1}(S^{\text{in}})$	0.	22.12	24.67	**32.57**	**32.57**
$\mathbb{A}(S^{-})$	56.28	81.67	**81.92**	-	**81.92**
$Tr^{-1}(S^{-})$	0.	**34.29**	**34.29**	-	**34.29**
$\mathbb{A}(S^{\text{out}})$	74.25	77.95	**78.15**	-	**78.15**
$Tr^{-1}(S^{\text{out}})$	0.	**15.41**	**15.41**	-	**15.41**
$\mathbb{A}(S^{\text{pl}})$	58.25	93.72	**93.79**	-	**93.79**
$Tr^{-1}(S^{\text{pl}})$	0.	**00.20**	**00.20**	-	**00.20**
$\mathbb{A}(S^{+})$	73.61	**74.54**	74.49	-	74.49
$Tr^{-1}(S^{+})$	0.	0.	0.	-	0.

Table 5.3: The evaluations on the CTrL sequences, except for S^{long}. For each sequence, we report average accuracy $\mathbb{A}$ and the amount of forward transfer on the last problem Tr^{-1}.

employed during training. If no improvement is achieved in 300 training iterations, the parameters with the best logged validation performance are selected. Data augmentation is also used during training, namely random crops (4 pixels padding and 32x32 crops) and random horizontal reflection.

Our results are shown in Table 5.3. We do not present results on MCL-RS and HOUDINI as we have already demonstrated that these approaches do not scale to large search spaces. It can be observed that EMO and PICLE, achieve similar performance to MNTDP-D on S^{pl}, S^{+}, S^{-}, S^{out} which evaluate plasticity, backward and perceptual transfer. This demonstrates that our approach can successfully perform perceptual transfer in this setting. On S^{-}, NOMO and PICLE successfully perform non-perceptual transfer, leading to superior performance on the last problem of the sequence ($+10.45$ higher than MNTDP). CTrL also specifies S^{long} which has 100 problems but only evaluates perceptual transfer. EMO was successful in transferring knowledge, achieving an average accuracy that was $+14.48$ higher than SA. MNTDP-D was also successful in scaling to this long sequence, achieving $+19.34$ higher performance than SA. Overall, the results demonstrate that both $\mathbb{S}_{\text{G}}^{\text{PT}}$ and $\mathbb{S}_{\text{BO-MAP}}$ are also applicable to more complex modular architectures.

5.8.3 Ablation Experiments

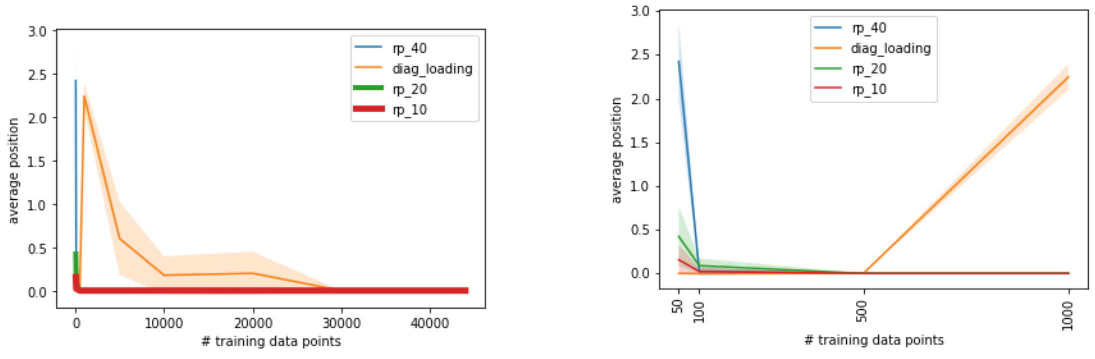
5.8.3.1 Random projections

The usage of $\mathbb{S}_{\mathrm{G}}^{\mathrm{PT}}$ (Eq. 5.10) relies on approximations of the training input distributions of pre-trained modules. Instead of modelling a module's training input distribution directly, we proposed to first project samples from it to k dimensions using random projection, and then we model the resulting distribution with a multivariate Gaussian. In this section we would like to evaluate three aspects of this approach. First, we would like to assess the usefulness of the resulting approximations for the purposes of selecting the correct input distribution. Second, we would like to assess the sensitivity of our approximations to the hyperparameter k. Third, we would like to compare our approach to Gaussian approximation of the original input space in order to determine whether we sacrifice performance.

To this end, we evaluate whether our approach is useful for distinguishing between a set of input distributions. We compare the approximations resulting from different choices of $k = \{0, 20, 40\}$. The resulting methods are referred to as *rp_10*, *rp_20* and *rp_40* respectively. Moreover, we compare to the method of computing a Gaussian approximation of a module's training input distribution, without a random projection. Since this can lead to a singular covariance matrix, we make use of diagonal loading (Draper and Smith, 1998) in which we add a small constant (10^{-8}) to the diagonal of the computed sample covariance matrix in order to make it positive definite. We refer to the resulting method as *diag_loading*.

We compare how well can these approaches distinguish between the 9 image datasets used in BELL. We chose to use the input images for our comparison since they have the highest dimension and should be the most difficult to approximate.

To evaluate one of the methods, we first use it to approximate all 9 input distributions using N data points, resulting in 9 approximations, denoted as $\{q_i\}_i^9$. Second, for each input distribution p_j, we sample 100 different data points and use them to order the approximations in descending order of their likelihood. Ideally, if the data points are sampled from the j-th distribution p_j, the corresponding approximation q_j should have the highest likelihood, and thus should be the first in the list, i.e. should have an index equal to 0. We compute the index of q_j in the ordered list and use it as an indication of how successfully the method has approximated p_j. We compute this index for each of the 9 distributions and report the average index, also referred to as the *average position*.



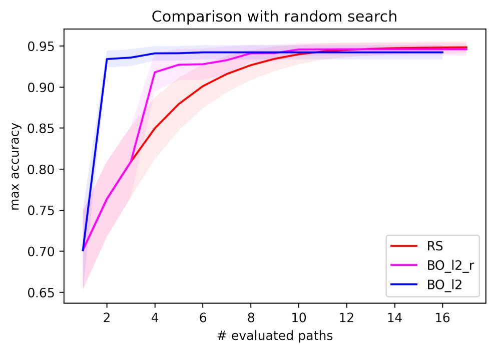
(a) (b)

Figure 5.5: Comparison of different methods for modelling a module's input distribution. The x axis represents the number N of data points used to compute an approximation. The y axis represents the average position, as defined in the main text, which indicates how well a method can approximate the distributions. The lower the average position is, the better the model performs. Figure a) presents a plot across all choices of N. Figure b) focuses on the first few values of N.

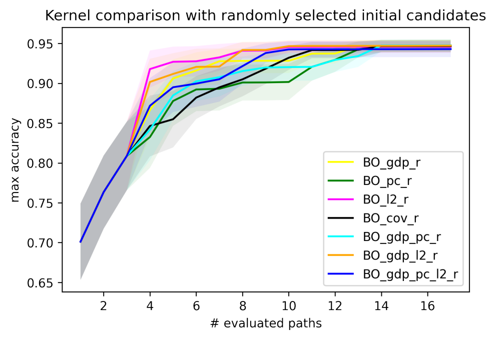
We evaluate each method for different choices of N, $N = \{50, 100, 500, 1000, 5000, 10000, 20000, 30000, 44000\}$. Moreover, we repeat all evaluations 5 times using different random seeds and report the mean and standard error of the average position. The results are reported in Fig. 5.5.

Our results show that directly modelling the original distribution with a Gaussian leads to sub-optimal performance. On the other hand, we observe that for $N \geq 500$, the methods which use random projection can always match the given data points with the correct distribution which they were sampled for. Surprisingly, for $N = 50$ and $N = 100$, diag_loading outperforms the other methods and can successfully identify the correct distribution of the given data points. Furthermore, we observe that for these values of N, decreasing the dimension k that the data points are projected to leads to better performance of the methods that are based on random projection.

Overall, our results suggest that the approximations which we use for $\mathbb{S}_{\mathrm{G}}^{\mathrm{PT}}$ are effective when the new modules are trained on more than 100 data points. This seems like a reasonable requirement, as fewer points are likely to result in a sub-optimal performance.



(a) Comparing to randomly selected initial points (BO_l2_rnd_init), and random search (RS).



(b) Comparing between different choices of GP kernels.

Figure 5.6: Comparing different design choices for our Bayesian optimisation algorithm $s_{\text{BO}}^{\text{NT},l}$.

5.8.3.2 Bayesian Optimisation

The search strategy $\mathbb{S}_{\text{BO-MAP}}$, which we use to search through NT paths for the opportunity of non-perceptual transfer, is equivalent to Bayesian optimisation. In this setting, we are interested in evaluating various aspects of $\mathbb{S}_{\text{BO-MAP}}$. First, we assess its ability to accelerate the search for the optimal NT path. Second, we assess its early stopping capabilities. Third, we compare our kernel function to other alternatives.

To begin with, we create a new sequence:

$$S^{\text{in}+} = [\Psi_1^+, \Psi_2^+, ..., \Psi_{15}^+, \Psi_{16}^- = (D_6, h_6, g_1)]$$

which involves all 16 upper labelling sub-functions g of BELL. The last problem's dataset is generated according to the triple $n_{\text{tr}}^- = (10000, \text{All}_{\text{tr}}, 30)$, which states that only 30 out of the 64 possible two-dimensional patterns are represented in the dataset. As a result, non-perceptual transfer is necessary in order to maximise the performance on the final problem. We create 5 realisations of $S^{\text{in}+}$ with different randomly selected problems.

Even though our method is deterministic, the baselines we compare it to involve randomly selecting 2 or all random paths. For each such baseline, and each of the 5 realisations of the sequence $S^{\text{in}+}$, we run the baseline 10 times with 10 different random seeds. This results in $10 * 5 = 50$ evaluations per method which we average over when reporting its performance. For each method, we plot its maximum accuracy achieved per number of programs evaluated (Fig 5.6), and we also report the average number of

	RS	bo_l2	bo_l2_r	bo_gdp_r	bo_pc_r	bo_cov_r	bo_gdp_pc_r	bo_gdp_l2_r	bo_gdp_l2_pc_r
#paths	17	11.5	**11.03**	14.57	13.53	11.98	15.27	13.93	13.61
$\mathbb{A}$	**94.70**	94.08	94.46	94.51	94.60	94.55	94.61	94.17	94.17

Table 5.4: A comparison of different discrete search algorithms.

programs it evaluates before early stopping (#paths) as well as the maximum accuracy it achieves at the end of its run ($\mathbb{A}$), as shown in Table 5.4.

There are 3 properties which we require from our approach: good any-time performance, good final performance, and effective early stopping. In other words, it should quickly find a good solution, then perform early stopping as soon as possible, and return a solution the performance of which is close to the best possible performance.

We begin by comparing our search strategy $\mathbb{S}_{\text{BO-MAP}}$ (which we denote BO_l2) to an augmented version which randomly selects the initial 2 paths (BO_l2_r), and to a random search baseline which recommends paths in a random order (RS). The results are shown in Fig. 5.6a and the first 3 columns of Table 5.4. The results demonstrate that *BO_l2* experiences a much stronger anytime performance, while evaluating a comparable number of paths and achieving a similar, albeit not the best, accuracy. Surprisingly, the 2 initial points chosen by BO_l2 already achieve high performance. As shown in Makarova et al. (2022), when early stopping based on the calculated Expercted Improvement, it is possible to sacrifice performance in exchange for evaluating fewer candidates. However, we consider the observed difference (also referred to as *regret*) to not be significant. The results also show that, even for randomly selected initial paths, the resulting method BO_l2_r exhibits a better anytime performance than random search, albeit worse than BO_l2. Moreover, BO_l2_r achieves better final accuracy than BO_l2 and evaluates slightly fewer paths, on average, than BO_l2. However, this method's reliance on randomly selected initial paths makes it less reliable than BO_l2.

Next, we investigate different choices of a kernel function used by the Gaussian process of $\mathbb{S}_{\text{BO-MAP}}$. Both BO_l2 and BO_l2_r use the RBF kernel, as defined in Eq. 5.14 in order to convert a distance between functions into a similarity between functions. Instead, we can directly calculate different similarity measures sim_v which can then be used with the following kernel:

$$k_v(\pi_j^{\text{NT},l}, \pi_k^{\text{NT},l}; Z) = \sigma_0^2 + \sigma_1^2 sim_v(\pi_j^{\text{NT},l}, \pi_k^{\text{NT},l}; Z) \tag{5.17}$$

where σ_0 and σ_1 are scalar hyperparameters that are optimised on the GP's training dataset, and Z is a set of points from the functions' input space, as defined in Section 5.5.1. We define the following similarity measurements for two scalar functions. First, we can compute the covariance between the function's outputs which computes the linear relationship between the functions' outputs but also reflects the magnitude of the outputs. This leads to the following similarity:

$$sim_{\mathrm{cov}}(f_1, f_2; Z) := COV(\cup_i f(z_i), \cup_i g(z_i))\ . \tag{5.18}$$

Second, we can compute the sample Pearson correlation coefficient (Lee Rodgers and Nicewander, 1988), denoted as PC, between the functions' outputs. This captures the linear correlation of the functions' outputs while ignoring their magnitude. As a result, the similarity ranges between $[-1, 1]$. The similarity is defined as:

$$sim_{\mathrm{pc}}(f_1, f_2; Z) := PC(\cup_i f(z_i), \cup_i g(z_i))\ . \tag{5.19}$$

Third, we note that the aforementioned similarities are computed based on the functions' outputs which only reflect points in their output space. Instead, we can compare the functions' curvatures around each evaluation input z_i. For each input, we compute the gradient of each function's output with respect to its input and normalise it to have a unit norm. The dot product between the two resulting normalised gradients is then computed in order to capture the alignment between the two curvatures. This leads to the following similarity:

$$sim_{\mathrm{gdp}}(f_1, f_2; Z) := \frac{1}{V}\sum_{i=1}^{V}\left(\frac{\nabla f_1(z_i)}{\left\|\nabla f_1(z_i)\right\|_2}\right)\cdot\left(\frac{\nabla f_2(z_i)}{\left\|\nabla f_2(z_i)\right\|_2}\right)\ . \tag{5.20}$$

The 3 similarities defined above lead to 3 kernels, which in turn result in the following 3 BO algorithms: BO_cov_r, BO_pc_r, BO_gpd_r. Moreover, one can sum kernel functions in order to result in a new kernel function, which uses a combination of similarities. We combine different kernels which leads to the following algorithms: BO_gpd_pc_r, BO_gpd_l2_r and BO_gpd_l2_pc_r. For all algorithms, we use randomly selected initial paths, because the paths selected deterministically perform too well which makes it harder to compare algorithms. The resulting comparison is presented in Fig. 5.6b and Table 5.4. It can be observed that early stopping prevents all of the BO algorithms from finding the optimal solution. However, they all achieve comparable performance while evaluating fewer paths. Overall, BO_l2_r achieves the best anytime performance by finding well-performing paths more quickly. Moreover BO_l2_r evaluates the fewest number of paths before early stopping. However, BO_l2_r has a slightly worse final accuracy, compared to using other similarity measures. Finally, we observe that combining the RBF kernel of BO_l2_r with other kernels does not result in a better algorithm.

Overall, our results suggest that $\mathbb{S}_{\text{BO-MAP}}$ can be used to accelerate the search for the optimal NT path. Moreover, its early stopping capabilities reduce the number of paths which need to be evaluated, if this number is not restricted by an expert. Finally, we show that our kernel function leads to quicker early stopping as well as better anytime performance.

5.8.4 Discussion

Our experiments evaluated our approach's LML capabilities on two separate benchmark suites. The results were consistent between sequences from the two benchmarks. This demonstrates our approach's applicability to different input domains and different neural architectures. Overall, the following properties can be observed. First, the results on both S^{pl} sequences demonstrate PICLE's plasticity as our method even surpasses the standalone baseline. Second, catastrophic forgetting is prevented by design, which assures our methods' stability. Third, we demonstrate all of the forward transfer properties which we distinguish between in our desiderata. Specifically, our method's perceptual transfer is evident from the results on both S^{out} sequences, both S^{long} sequences, and on the $S^{\text{out*}}$ and $S^{\text{out**}}$ BELL sequences. Moreover, the S^{few} sequence in BELL shows that PICLE can attain few-shot transfer. Furthermore, the results on both S^{in}, and on the S^{sp} and S^{long} BELL sequences, all demonstrate our approach's ability to achieve non-perceptual transfer. Fourth, the results on both S^{+} sequences show that PICLE is unable to perform backward transfer. Finally, our experiments were designed so that they also evaluate the methods' scalability, as the search spaces for even the short sequences of BELL and CTrL are upper-bounded by $O(167916)$ and $O(46656)$ respectively. Therefore PICLE's ability to attain these LML properties on these large search spaces provides evidence of its scalability. Moreover, the results on both S^{long} sequences show that PICLE can also scale to longer sequences of problems.

Our experimental results also demonstrate PICLE's advantages over other modular LML algorithms. We consistently outperform both MCL-RS and HOUDINI, demonstrating the advantages of our approach over modular LML algorithms which use a naive search strategy. On the other hand, MNTDP-D is a scalable modular LML algorithm which can achieve perceptual transfer. Our results demonstrate that both

MNTDP-D and PICLE achieve comparable performance on the short sequences that evaluate the LML properties which are shared between the methods. However, PICLE is able to outperform MNTDP-D on the S^{few}, S^{sp} and S^{long} BELL sequences as well as on both S^{in} sequences. This is due to MNTDP-D's inability to achieve non-perceptual and few-shot transfer. Interestingly, MNTDP-D demonstrated better performance than EMO on both S^{long} sequences, suggesting that MNTDP-D's search strategy is more effective at perceptual transfer than $\mathbb{S}_{\text{G}}^{\text{PT}}$ on long sequences. Conversely, $\mathbb{S}_{\text{G}}^{\text{PT}}$ demonstrated that it can utilise its prior distribution to achieve superior performance on the $S^{\text{out**}}$ BELL sequence, which evaluates perceptual transfer.

5.9 Related Work

Different desirable LML properties can be found across the literature. This includes catastrophic forgetting (Kirkpatrick et al., 2017), backward transfer (Lopez-Paz and Ranzato, 2017), scalability (Chaudhry et al., 2018b), non-perceptual transfer (Valkov et al., 2018), negative transfer (Rusu et al., 2016), few-shot transfer (Valkov et al., 2018; Ostapenko et al., 2021), privacy guarantees (Farquhar and Gal, 2019). Kemker and Kanan (2017) suggest that an algorithm should overcome catastrophic forgetting across different modalities, e.g. images and audio data, however, they modify both modalities so that they share the same input space. In this work, we present a list of properties, important for an LML algorithm, in which we focus on different transfer learning requirements.

Previous work also defines different LML desiderata. For instance PC (Schwarz et al., 2018b), EC (Hadsell et al., 2020), CS (Delange et al., 2021) and MC (Veniat et al., 2020) all define a list of desirable properties. All mention scalability, however, PC and EC do not further specify a requirement, CS asks that the algorithm's memory remains constant, and MC asks that the algorithm's memory and compute requirements scale sub-linearly with the number of encountered problems. While the rest aim to minimise catastrophic forgetting, CS also asks for "selective forgetting of trivial information". All these papers agree that positive forward transfer and positive backward transfer are necessary. However, only MC further decomposes it into what we call perceptual and non-perceptual transfer. Still, MC is missing the requirements for few-shot transfer and non-perceptual transfer across input spaces. EC requires an algorithm to be able to quickly adapt to domain shifts. While this can be interpreted as perceptual transfer, it is an important property which should be listed separately. In our work, we do not list it explicitly and our benchmark suite does not evaluate for it. EC also lists that an algorithm should require minimal access to previous problems. We do not list this, as we assume it is implied, however, it might be useful for this property to be stated explicitly. Other properties listed in the aforementioned work are not related to what we have defined as problem-incremental learning. For instance, PC, EC and CS argue that an algorithm shouldn't be provided with the index of the problem being solved. Moreover, CS asks that an algorithm be able to perform online learning on a continuous stream of data, as well as to be able to perform unsupervised learning on unlabelled data.

Different sequences and benchmarks have been proposed to evaluate different LML settings. For instance, CORe50 (Lomonaco and Maltoni, 2017) presents sequences for continuous object recognition based on a new dataset of natural images. Depending on the sequence, at each training batch they provide new images of a previously seen object, or new objects with previously unseen classes, or both. Therefore, even though they maintain the same problem, the data seen during training changes and an algorithm needs to adapt to it. Farquhar and Gal (2018) discuss a number of requirements for that an LML benchmark should have. For instance, the authors advocate for overlapping tasks and unclear task demarcations. However, in this work, we focus on incremental-problem setting in LML, in which problems are given one at a time and their index is specified. We do this because this setting is already challenging for current methods, without the need for an extra layer of complexity. Different sequences, relevant to this setting, have been proposed in the literature. Permutted MNIST Goodfellow et al. (2013), described in Section 2.5.1.2, presents a sequence in which each task is to classify MNIST images, but the inputs are permutted with a random problem-specific permutation. This random permutation means that convolutional layers can't be used to process the input, as they depend on local features in the image. As a result, this sequence is usually addressed using a neural architecture which consists of only fully-connected layers. However, in this case, only the first layer needs to change in order to accommodate for the new problem, meaning that it suffices to control catastrophic forgetting on the first layer only. Still, Permutted MNIST can be seen as evaluating non-perceptual transfer. We have not used it because of the limitations that it places on the applicable neural architectures. Two other common LML sequences are Split-CIFAR10 and Split-CIFAR100, also described in Section 2.5.1.2. The labels of either image dataset are split into disjoint subsets and each problem then involves classifying images from the corresponding subset of classes. As a result, all problems in

a sequence have a similar input domain, which allows perceptual knowledge transfer, but does not evaluate an algorithm's ability to avoid negative transfer. While such sequences are useful for evaluating catastrophic forgetting and some form of transfer, they are limited in the number of properties they evaluate.
Instead, in Chapter 4, we presented a number of different sequences of compositional problems, designed to evaluate different properties of LML. However, the image classification tasks that we used were primarily binary classification between two image classes, which reduces the difficulty. Moreover, there were only 1 or 2 different tasks operating on a list of binary classifications, namely, counting and summing. This restricts the number of different compositional problems and prevents us from constructing challenging longer sequences. In contrast, BELL uses multi-class image classification and contains a larger variety of compositional problems. The CTrL benchmark (Veniat et al., 2020) also introduced a number of sequences of image classification tasks, which allows us to diagnose the properties of an LML algorithm. Our work builds on this by extending it to compositional problems. This allows us to make S^{in} more challenging and to introduce new sequences, designed to be more difficult versions of already existing sequences, namely $S^{\text{out}*}$ and $S^{\text{out}**}$. Moreover, due to our problems' compositionality, we can add two new sequences, S^{in} and S^{few}, which evaluate for LML properties that were not evaluated in CTrL.

Continual learning methods can be categorised into ones based on regularisation, replay or a dynamic architecture (Parisi et al., 2019). The first two share the same set of parameters across all problems which limits their capacity and, in turn, their plasticity (Kirkpatrick et al., 2017). Dynamic architecture methods can share different parameters by learning problem-specific parameter masks (Mallya and Lazebnik, 2018) or adding more parameters (Rusu et al., 2016). This category includes modular approaches to CL, which share and introduce new modules, allowing groups of parameters to be trained and always reused together. Modular approaches mainly differ by their search space and their search strategy. PathNet (Fernando et al., 2017) uses evolutionary search to search through paths that combine up to 4 modules per layer. Rajasegaran et al. (2019) use random search on the set of all paths. HOUDINI (Valkov et al., 2018) uses type-guided exhaustive search on the set of all possible modular architectures and all paths. This method can attain the three types of forward transfer, but does not scale to large search spaces. Their results also show that exhaustive search leads to better performance than evolutionary search. MNTDP-D (Veniat et al., 2020) is a scalable approach which restricts its search space to perceptual transfer paths, derived from previous solutions. Similarly to $\mathbb{S}_G^{\mathrm{PT}}$, MNTDP-D evaluates only $L+1$ paths per problem, however, the approach does not allow for novel combinations of pre-trained modules which prevents it from achieving few-shot transfer. In contrast, our approach can achieve all three types of forward transfer. LMC (Ostapenko et al., 2021) makes a soft selection over paths. For each layer, they compute a linear combination of the outputs of all available pre-trained modules. To do this, the authors model the distribution over each module's outputs using a separate auto-encoder. In contrast, for $\mathbb{S}_{\mathrm{G}}^{\mathrm{PT}}$ we use multivariate Gaussian approximations of the projected inputs of a pre-trained module, which requires orders of magnitude fewer parameters. Concretely, in our experiments we need 420 parameters per approximation, while LMC needs $82,176$. Finally, LMC requires that all modules be kept in memory which limits its scalability to larger libraries, thus, larger search spaces.

The work presented in this chapter automates the process of choosing the best path. As such, the setting is similar to that of AutoML (He et al., 2021) in which the algorithms aim to automate different aspects of the process of applying machine learning to a problem. In particular, our setting is similar to hyperparameter optimisation (HPO) (Yu and Zhu, 2020) and neural architecture search (NAS) (Elsken et al., 2019). HPO aims to optimise different hyperparameters related to the training procedure (e.g. learning rate, choice of optimizer, magnitude of regularisation) and the neural architecture (e.g. number of hidden layers, number of hidden units, choice of activation functions). Categorical hyperparameters are typically encoded as a one-of-k embedding. However, in our setting, embedding all of the selected modules would result in an embedding of size $O(t^{l_{\min}})$ which would make the sample complexity of our surrogate model prohibitively high. NAS is a sub-setting of HPO which specialises to searching through more elaborate neural architectures. For this purpose, NAS methods also study how to best featurise a neural architecture, as described in Section 3.4. However, the approaches expressed in literature are specific to the respective neural architectures and aren't applicable to our setting. Despite the differences, there are also similarities between previous work in HPO and NAS, and our approach. For instance, most HPO and NAS approaches make use of a surrogate function. Moreover, it is common to use Bayesian optimisation with a Gaussian process (GP) as a surrogate function in HPO. Kandasamy et al. (2018) use Bayesian optimisation with a GP for NAS. Similarly to our work, they define their own kernel function to compute the similarity between two neural architectures. The idea of early stopping with Bayesian optimisation has also been explored before. For this purpose, Nguyen et al. (2017) use

EI, Lorenz et al. (2015) use the probability of improvement (PI) and Makarova et al. (2022) use a measure based on the lower confidence bound (LCB). Makarova et al. (2022) compare all 3 approaches and show that while using EI and PI has a higher chance of stopping before finding an optimal solution, they also evaluate significantly fewer different configurations.

5.10 Conclusion

In this chapter, we specified a list of LML desiderata, which distinguishes between different types of forward transfer. We then presented the first modular LML algorithm capable of achieving all of the listed LML properties, apart from backward transfer. To achieve this, we introduced a probabilistic search framework which can be used to perform efficient search over module combinations. We developed two probabilistic models over two different subset of module combinations. Used within our framework, these models enabled us to search for paths which achieve perceptual, few-shot and non-perceptual transfer, while scaling to large search spaces. Finally, we introduced a new benchmark suite which can be used to diagnose an LML algorithm's ability to attain the identified desiderata.

An exciting direction for future work is enabling modular LML algorithms to achieve backward transfer. This would lead to the first LML algorithm, which can achieve all of the desiderata that we have identified in this chapter. To achieve this, one would need to unfreeze the pre-trained modules, allowing their parameters to change on future tasks. An additional insight is necessary in order to then prevent forgetting and encourage backward transfer.

Another direction for future work is applying our probabilistic framework to more subsets of module combinations. This can bring additional benefits, such as being able to simultaneously perform perceptual and non-perceptual transfer, while also introducing new modules in the middle of a modular neural architecture. For this purpose, one would need to define a different probabilistic model over a subset of some of the currently unexplored module combinations.

Chapter 6

A Transfer-Learning Extension of Hyperband

In Chapter 5 we presented a template for modular lifelong machine learning (LML) algorithms (Algorithm 1) in which a key component is a search strategy $\mathbb{S}$ for navigating the set of all paths (all module combinations), Π_t, for a given problem. Furthermore, we showed that we can apply a divide-and-conquer approach by exploring different path subsets using different search strategies. However, the approach which we presented left a big subset of paths unexplored, namely $\Pi_t^{\text{rest}} = \Pi_t \setminus (\Pi_t^{\text{PT}} \cup \Pi_t^{\text{NT}})$. This subset includes paths which can simultaneously perform perceptual and non-perceptual transfer as well as other potentially useful paths which are more flexible in the way they transfer knowledge. This subset is also not explored by other modular LML algorithms (Veniat et al., 2020; Ostapenko et al., 2021). While HOUDINI searches through all paths (and neural architectures), its search algorithm prevents it from being applicable to large search spaces. Overall, there is a need for a scalable search strategy which can explore Π_t^{rest}.

In this chapter, we phrase the optimisation problem in Algorithm 1 (Eq. 5.1) as black-box optimisation (BBO) which allows us to make use of hyperparameter optimisation (HPO) methods when designing a relevant search strategy. However, previous HPO methods suffer from one or more out of three shortcomings. These shortcomings include the necessity for a manually-designed special input featurisation (e.g. a suitable GP kernel), the cold-start problem and the inefficiency incurred by evaluating every considered element using the same constant number of resources. As a result, previous HPO methods are difficult to apply for searching through Π_t^{rest}. To address this, this chapter presents a new HPO method which addresses all of the aforementioned shortcomings. The merits of our method are experimentally verified in the HPO setting, and using it for modular LML is left for future work.

We augment a popular multi-fidelity approach, Hyperband (Li et al., 2017), with adaptive sampling, using a neural-network-based surrogate model, ABLR (Perrone et al., 2018). As a result, our surrogate model is flexible enough to learn useful representations of the input, while also being able to address the cold-start problem by transferring data across HPO tasks. We evaluate the resulting method, which we refer to as HB-ABLR, on two HPO settings, where HB-ABLR outperforms the previous state-of-the-art (Falkner et al., 2018) approach. Afterwards, we present a discussion on how HB-ABLR can be applied to modular LML.

6.1 Introduction

6.1.1 Black-Box Optimisation for Deep Learning

Algorithms for black-box optimisation (BBO) (Section 3.5) present a useful set of tools that are applicable to a wide variety of settings. A black-box function is a function for which an analytical form is not available, making it impossible to obtain its derivative analytically (Li et al., 2021b). It is possible to evaluate the function on a selected input, however, doing so is usually assumed to be expensive in terms of required resources. For instance, evaluating a function can be time-consuming. The setting of single-objective BBO, also known as zeroth-order optimisation, requires one to find the input of a scalar black-box function $f : \Omega \to \mathbb{R}$ which optimises said function's output value. In order words, one needs to find $\mathbf{x}^*$ where $\mathbf{x}^* = \text{argmin}_{\mathbf{x}\in\Omega} f(\mathbf{x})$. Since each evaluation is expensive, the goal is to optimise the function using as few evaluations as possible.

The challenge of optimising some properties of a deep learning algorithm can be phrased as BBO. Concretely, one could look for the values of some properties of a deep neural network, which minimise its validation loss after training, thus, setting $f = \mathcal{L}^{\text{val}}$. For instance, the modular lifelong learning (MOLL) algorithm, defined in Algorithm 1 in Chapter 5, optimises the choice of a path $\pi \in \Pi_t$ which leads to the lowest validation loss (Eq. 5.1). Similarly, in *hyperparameter optimisation* (HPO), one searches for the hyperparameters of a machine learning algorithm, which lead to the lowest validation loss. Moreover, in *neural architecture search*, which can be seen a sub-field of HPO, the focus is on optimising the architecture of a deep neural network, in order to minimise its validation loss.

A common approach, which is applicable to all three settings is sequential model-based optimization (SMBO) (Section 3.5.1). It approximates the target function f with a surrogate model, using the previously obtained input evaluations. The resulting approximation is then used to find the most promising input to be evaluated next. Bayesian optimisation (BO) (Section 3.5.2) is a popular SMBO approach, in which the surrogate models the function's noisy output distribution $p(f(\mathbf{x})+\varepsilon|\varphi(\mathbf{x}))$, for some mean-centered normally distributed noise $\varepsilon \sim \mathcal{N}(\mathbf{0},\sigma^2)$. Here, φ is used to underscore that the input could be differently embedded when provided to the model, e.g. a discrete variable can be embedded using one-hot encoding. The conditional distribution also reflects the uncertainty of the surrogate model about its prediction, which allows BO algorithms to choose between exploitation and exploration. In Chapter 5, we used BO to define a search strategy over paths for non-perceptual transfer ($\mathbb{S}_{\text{BO-MAP}}$, Equation 5.15).

There are three important obstacles related to applying SMBO and BO algorithms to black-box optimisation. First, one needs to define the mapping φ in order to represent the inputs in a way which is suitable for the surrogate model. For example, in order to use $\mathbb{S}_{\text{BO-MAP}}$ in Chapter 5 we had to define a kernel function between paths which implicitly defines an embedding of each of the considered paths. This was done in order to allow the use of a Gaussian process (GP) as a surrogate model. Second, these algorithms can encounter the *cold-start* problem which occurs in the early iterations of these algorithms, especially when the input is high dimensional. When a few function evaluations are available, this is unlikely to be sufficient for the surrogate model to learn a useful approximation of the target function, which means that the model is unlikely to locate promising locations which should be explored next. This leads to poor performance in the initial iterations, where performance is defined by the lowest validation loss (target function value) found so far. Third, SMBO and BO can be resource inefficient which can hinder their performance when only a small amount of resources can be dedicated to an optimisation run. A potentially unnecessary cost can be incurred due to the fact that these methods require that each function be evaluated using the same amount of resources. For example, that a neural network be trained for the same number of epochs before its validation loss is evaluated.

In this chapter, we introduce an approach which addresses the aforementioned three obstacles. We present a multi-fidelity approach which uses Bayesian optimisation to select promising inputs. Moreover, our approach can address the cold-start problem by transferring knowledge from evaluations of similar black-box functions, obtained while performing BBO on said functions. We demonstrate our algorithm's effectiveness in the setting of hyperparameter optimisation. However, it is also applicable to neural architecture search, as evident by the application of a similar algorithm (Falkner et al., 2018) in this setting (Dong and Yang, 2020). Moreover, this chapter includes a discussion on how this method can be applied to modular lifelong learning.

6.1.2 Hyperparameter Optimisation

Applying a machine learning algorithm to solve a problem requires a user to set the values of algorithm-specific parameters, which are broadly referred to as *hyperparameters*. Examples of hyperparameters are the learning rate of stochastic gradient descent (Bottou and Cun, 2003) and the number of hidden units of a fully connected deep neural network (Bergstra and Bengio, 2012). Selecting the correct hyperparameters can significantly increase the performance of the resulting ML algorithm. While good performance hinges on a careful tuning of the hyperparameters, this process is time-consuming and crucially depends on the level of expertise of the practitioner. Therefore, it is desirable to automate the tuning process in order to reduce the associated costs.

Bayesian optimisation has been successfully used to automate hyperparameter search by casting it as black-box optimisation (Zhang et al., 2021; Melis et al., 2017; Jafar and Myungho, 2020). However, as discussed above, there are three obstacles to applying BO, which also arise in the setting of hyperparameter optimisation. First, finding the correct encoding for the hyperparameters, i.e. defining φ, can be challenging. For instance, if we wanted to find the best value for the learning rate between $(1\text{e}{-5}, 1)$, it can be seen that 99% of the values fall in the range $(1\text{e}{-2}, 1)$, leaving the rest unlikely to be explored during optimisation. This can be addressed by applying a log scale to the hyperparameter, and instead optimising for a value between $(\log_{10} 1\text{e}{-5}, \log_{10} 1)$. This spreads the values of interest more evenly. However, having an expert specify the scaling of each continuous hyperparameter is laborious and error-prone (Snoek et al., 2014), so it is preferable that the scaling be learned. Second, the issue of cold start, as described earlier, has also been observed when using BO for hyperparameter optimisation (Swersky et al., 2013). Third, most Bayesian optimisation methods evaluate all hyperparameter configurations with the same number of resources, which can incur an unnecessary cost, as discussed above.

ABLR (Perrone et al., 2018) is a surrogate model, which addresses the first two challenges. It computes a latent embedding of the hyperparameters using a neural network. It then feeds the latent embedding to a Bayesian linear regression (BLR), which then predicts the target function's conditional output distribution. As a result, ABLR can learn a suitable embedding of the hyperparameters using a neural network. In addition, a multi-head architecture can be used, in which the neural network is shared and a separate BLR is used to predict the outputs of different black-box functions. In effect, this transfers knowledge on how to embed the hyperparameters from previous BO runs on similar black-box functions. This addresses the cold-start problem as suitable parameter values of the surrogate neural network can be found, even if only a few evaluations are available from the black-box function of current interest.

The third challenge can be addressed using a multi-fidelity approach. Starting from the key observation that training ML models is an incremental process controlled by some *resources*, for instance, training set sizes, number of epochs or some amount of time, a recent line of work (Swersky et al., 2014; Klein et al., 2017; Li et al., 2017; Falkner et al., 2018) has proposed to exploit this feature to reallocate resources to the most promising HPs by making cheap and early decisions based on evaluations of f after having consumed few resources. Successive Halving (Jamieson and Talwalkar, 2016) selects n hyperparameter configurations uniformly at random, and evaluates them for r_0 resources before selecting a portion with the highest observed performance. It then iterates between further evaluating the remaining configurations with more resources and further filtering out configurations which exhibit lower performance. However, it is not clear what is the minimum number of resources r_0 after which one can reliably discard configurations. To address this, Hyperband (HB) (Li et al., 2017) invokes Successive Halving using different values for n anr r_0. Despite its simplicity, HB has been shown to outperform BO on many ML-related tuning tasks in regimes where solutions with moderate precision are acceptable. However, since the configurations are randomly sampled, the method does not exploit promising areas of hyperparameter values which can lead to sub-optimal final performance. To improve the precision resolution of Hyperband, several model-based extensions of Hyperband have been recently proposed (Bertrand et al., 2017; Wang et al., 2018; Falkner et al., 2018). Instead of randomly sampling the competing configurations, these methods use model-based adaptive sampling, which allows them to balance between exploitation and exploration and can lead to better final performance. However, the methods which use a Gaussian process (Bertrand et al., 2017) cannot scale to a large number of evaluations and the methods based on TPE (Wang et al., 2018; Falkner et al., 2018) are only trained on a subset of the available HB evaluations. Furthermore, these methods train their surrogate model only on evaluations of the current black-box of interest, thus, they do not address the cold-start problem.

In this work we augment Hyperband with adaptive sampling which uses ABLR as a surrogate model. Our approach models the amount of resources used to evaluate a hyperparameter configuration as a *contextual variable*, which allows us to fit the surrogate using all available evaluations. HB-ABLR can also make use of evaluations obtained from previous hyperparameter optimisation tasks. This knowledge transfer allows our method to increase both its anytime and final performances. Our work addresses all three of the aforementioned obstacles. Using a neural-network-based surrogate model allows us to learn a suitable transformation of the hyperparameters. Furthermore, HB-ABLR addresses the cold-start problem by transferring knowledge from previous evaluations. Finally, our method represents a multi-fidelity approach which can early stop under-performing configurations from being fully evaluated. We perform experiments on two hyperparameter optimisation tasks. Our results show that HB-ABLR can achieve competitive performance, in the absence of transfer. Using knowledge transfer, HB-ABLR consistently outperforms the previous state-of-the-art method (Falkner et al., 2018). Moreover, we perform ablation experiments which provide insight into the type of knowledge being transferred and into the robustness of single-task HB-ABLR.

6.2 Background

Say one is interested in performing hyperparameter optimisation for some machine learning algorithm $\mathcal{A}$ and for given training and validation datasets, D^{tr} and D^{val}. This can be cast as BBO, where the target black-box function maps a hyperparameter configuration $\mathbf{c}$ to a value l, which represents the validation performance, evaluated after training the ML algorithm, using the selected hyperparameters and R resources, on the given training dataset. Unless stated otherwise, we assume that l represents the validation loss. Evaluating the validation loss is assumed to lead to a noisy observation, since training the same algorithm with the same hyperparameters can lead to different validation losses. Therefore, the observation can be expressed as $l = f(\mathbf{c}) + \varepsilon = \mathcal{L}(\mathcal{A}(D^{\text{tr}}; \mathbf{c}, R), D^{\text{val}})$, where epsilon is a zero-centered normally distributed noise, $\varepsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2)$. After defining the target function, it is necessary to then define a set of hyperparameters Ω to be considered by defining the set of values which each of the hyperparameters of interest can take. One is then interested in finding the optimal hyperparameter configuration $\mathbf{c}^*$ which minimises the validation loss, $\mathbf{c}^* = \text{argmin}_{\mathbf{c}\in\Omega} \mathcal{L}(\mathcal{A}(D^{\text{tr}};\mathbf{c},R),D^{\text{val}})$. The performance of a hyperparameter optimisation approach can be defined by the minimal validation loss which has been found after evaluating a certain number of candidates.

Bayesian optimisation (BO) uses a surrogate model to model the conditional output distribution of the target function, $p(l|\varphi(\mathbf{c}))$ Here, φ is a mapping which maps the selected hyperparameters to a representation which is suitable for the chosen surrogate model. For instance φ can map a discrete variable with 3 values to a three-dimensional one-hot encoding. While φ is usually omitted, we use it to underscore the need for an expert to select a suitable embedding. After evaluating n configurations, BO fits its surrogate model using the set of available evaluations, $E = \{c^{(i)}, l^{(i)}\}_{i=0}^{n}$. BO uses an acquisition function g which computes a configuration $c^{(j)}$'s value, based on the surrogate model's prediction. The most promising configuration to be evaluated next is selected by minimising the acquisition function: $\hat{\mathbf{c}} = \text{argmin}_{c\in\Omega}\, g(p(l|\varphi(\mathbf{c});\mathbf{v}))$ (or maximising, depending on the acquisition function). When the number of previous evaluations, $|E|$, is low, the surrogate model might not approximate the target function well, which makes the chosen hyperparameter configuration unlikely to lead to an improvement over the previously observed minimum validation loss. In this case, the cold-start problem is said to have arisen.

When optimising the hyperparameters of a black-box function f_t with index t, it is possible to use evaluations $E_{<t}$ of similar, previously encountered, functions $f_{<t}$ in order to *warm-start* the optimisation process. A similar function could be the post-training validation loss of the same algorithm, computed using different training and validation datasets. One can then augment the surrogate model to take advantage of previous evaluations, modelling $p(l|\varphi(\mathbf{c}), E_t, E_{<t})$, where E_t are the available evaluations of the function of interest f_t. For instance Swersky et al. (2013) use multi-task Gaussian processes, which jointly model the output distributions of both the previously encountered functions and the black-box function of current interest. Such approaches are said to perform *knowledge transfer* to the target black-box function. Note that this transfer benefits the surrogate model, while in previous chapter we have discussed the knowledge transfer which benefits the learning algorithm $\mathcal{A}$. Transfer learning can be used to address the cold-start problem of Bayesian optimisation.

We now present a list of desiderata for a practical approach to hyperparameter optimisation, introduced in Falkner et al. (2018), which we later refer to when analysing our approach. **1. Strong Anytime Performance.** Evaluating a single hyperparameter configuration can be expensive in terms of time and computational resources, especially if it involves training a deep neural network. However, it is often the case that only a limited amount of resources can be allocated to a single run of HPO. Therefore, it is important for a HPO approach to be able to find acceptable solutions using a small number of overall resources. **2. Strong Final Performance.** On the other hand, when many resources can be allocated for a single HPO run, what matters is the performance that an HPO approach attains at the end. **3. Effective Use of Parallel Resources.** If multiple machines are available, e.g. a compute cluster, an HPO method needs to be able to make use of these effectively. **4. Scalability.** As the complexity of the target machine learning algorithm grows, there might be a need to tune more hyperparameters. Therefore, it is important that a HPO algorithm can scale to searching through a large number of hyperparameters. **5. Robustness & Flexibility.** To increase the applicability of a HPO method, it needs to be able to effectively search through different types to hyperparameters, such as binary, categorical, integer and continuous. Furthermore, it needs to be robust with respect to its own hyperparameters. **6. Simplicity.** It is desirable for an HPO method to limit its reliance on an expert for the selection of said method's hyperparameters. For instance, having to select a suitable kernel function. The authors also argue that it is important for a HPO approach to be easy to re-implement in different libraries. **7. Computational Efficiency.** The method should be able to scale with a large number of evaluations. This can become necessary for large experiments in which multiple parallel evaluations are possible using a compute cluster.

6.2.1 Adaptive Bayesian Linear Regression (ABLR)

Perrone et al. (2018) present a surrogate model for Bayesian optimisation (ABLR), which can be used to alleviate the need for manual input featurisation and can address the cold start problem by transferring knowledge from previous BO runs. Moreover, the model can scale linearly with the number of evaluations. To achieve this, the authors use Bayesian linear regression (BLR), as it can be modified to scale linearly with the number of inputs and cubically with its input dimension (Bishop, 2006). Since BLR has limited expressivity, the authors introduce a nonlinear transformation $\phi_{\mathbf{z}}$, parameterised by $\mathbf{z}$, which computes a suitable latent representation of a hyperparameter configuration. Transfer learning is achieved by using a multi-head architecture in which a shared $\phi_{\mathbf{z}}$ is used to help approximate multiple black-box functions.

Let the hyperparameter optimisation problem of interest have index t. Furthermore, let there be N_t already evaluated configurations, stored in a meta dataset $E_t = \{(\mathbf{c}_t^{(i)}, l_t^{(i)})\}_{i=1}^{N_t}$. A feedforward fully connected neural network is used to represent the nonlinear transformation $\phi_{\mathbf{z}}(\varphi(c)) : \mathbb{R}^P \to \mathbb{R}^D$, where $\mathbf{c}$ is a configuration with initial dimensionality $\varphi(\mathbf{c}) \in \mathbb{R}^P$. The feature matrix of all latent embeddings of the evaluated hyperparameter configurations is denoted as $\Phi_t = [\phi_{\mathbf{z}}(\varphi(\mathbf{c}_t^{(i)}))]_{i=1}^{N_t}$. ABLR models the observed values $\mathbf{l}_t$ and the prior distribution over the weights as follows:

$$p(\mathbf{l}_t|\mathbf{w}_t, \mathbf{z}, \beta_t) = \mathcal{N}(\Phi_t \mathbf{w}_t, \beta_t^{-1}\mathbf{I}_{N_t}),\ p(\mathbf{w}_t|\alpha_t) = \mathcal{N}(\mathbf{0}, \alpha_t^{-1}\mathbf{I}_D)$$

where $\beta_t > 0$ and $\alpha_t > 0$ are precision parameters which are used to specify the observational noise and the prior over the weights, respectively. For a new data point $\mathbf{c}_t^*$, the predicted noise-free value according to the model is $f_t^* = \mathbf{w}_t^\top \phi_{\mathbf{z}}(\varphi(\mathbf{c}_t^*))$. The *predictive distribution*

$$p(f_t^*|\mathbf{c}_t^*, E_t) = \int p(f_t^*|\mathbf{c}_t^*, \mathbf{w}_t) p(\mathbf{w}_t|E_t) d\mathbf{w}_t$$

is Gaussian and the authors show how to compute it by analytically integrating out the weights $\mathbf{w}_t$. As a result, the parameters that need to be learned are $\mathbf{z}$, α_t and β_t. These are found by jointly minimising the log *marginal likelihood*:

$$\rho(\mathbf{z}, \alpha_t, \beta_t) = -\log P(\mathbf{l}|\mathbf{z}, \alpha_t, \beta_t)$$

Overall, ABLR can be regarded as a deep neural network whose final linear layer is subject to Bayesian treatment. If evaluations of previously encountered black-box functions are available, ABLR models all evaluations with a multi-head neural architecture. Concretely, each black-box function is predicted using a shared $\phi_{\mathbf{z}}$ and a separate Bayesian linear regression for each black-box function. All parameters are then jointly learned by minimising the following criterion:

$$\rho(\mathbf{z}, \{\alpha_t', \beta_t'\}_{t'}) = -\sum_{t'} \log P(\mathbf{l}|\mathbf{z}, \alpha_{t'}, \beta_{t'}) \tag{6.1}$$

Knowledge transfer is achieved through the shared nonlinear transformation, which learns how to embed the hyperparameter configurations in a way which is useful for the prediction of the values of all encountered black-box functions.

All parameters of ABLR are optimised using LBFGS (Byrd et al., 1995). The authors show that the multi-task learning criterion and its gradient can be computed in $O(\sum_{t'} \max(N_{t'}, D) min(N_{t'}, D)^2)$. They note that for a large number of transfer data, it is possible to optimise $\phi_{\mathbf{z}}$ only on previous black-box functions and freeze its parameters. This would further improve the efficiency of the method.

Algorithm 2: SuccessiveHalving

```
Input: $R$, Maximum number of resource per configuration.
Input: $n$, Initial number of candidates.
Input: $s$, Number of iterations.
Input: $\eta$, Specifying the proportion of configurations to keep.
[1] $r = R\eta^{-s}$
[2] $T = \text{get\_hyperparameter\_configurations}(n)$
[3] foreach $i \in \{0,...,s\}$ do
[4]     $n_i = \lfloor n\eta^{-i} \rfloor$
[5]     $r_i = r\eta^{i}$
[6]     $L = \{\text{run\_then\_return\_val\_loss}(t, r_i) : t \in T\}$
[7]     $T = \text{top\_k}(T, L, \lfloor \frac{n_i}{\eta} \rfloor)$
[8] end
```

6.2.2 Hyperband

Hyperband (Li et al., 2017) is a hyperparameter optimisation method which considers a pool of randomly sampled configurations at a time. For each pool, the method iterates between evaluating the configurations using a small amount of resources and then selecting the best performing portion to be further evaluated with more resources, while discarding the rest. Hyperband has been shown to have strong anytime performance and to be able to find a good configuration much faster than resource-unaware Bayesian optimisation methods. On the other hand, Bayesian optimisation methods typically have a better final performance, as they find a better configuration given enough time (Falkner et al., 2018). Hyperband uses Successive Halving (Jamieson and Talwalkar, 2016) as a sub-routine. For this reason, we first present Successive Halving, using the parameterisation presented in Li et al. (2017) and then describe Hyperband.

Successive Halving (SH) iteratively evaluates a pool of hyperparameter candidates at different fractions of the maximum amount of resources possible, discarding a fraction of the candidates at every iteration. The algorithm is presented in Algorithm 3. Among its inputs, R is the maximum number of resources which a hyperparameter configuration can be evaluated for, e.g. maximum number of epochs necessary to train a neural network. Moreover, n is the initial number of configurations in the pool and η is used to specify the proportion of candidates which we would like to keep after each iteration, $\frac{1}{\eta}$. Finally, s specifies the number of iterations we would like *Succes-*

siveHalving to perform. After s iterations, each of the configurations left in the pool of configurations should be evaluated for R resources. Therefore, SH uses s and R to determine the number of resources r_i which need to be used at each iteration. SH creates the initial pool of candidates by sampling configurations uniformly at random from the defined hyperparameter space. The algorithm then performs $s+1$ iterations. At each iteration, the pool of remaining candidates T is evaluated for r_i resources and the $\frac{n_i}{\eta}$ candidates with the lowest observed validation loss are kept in the pool for further evaluation, while the rest are discarded. The resulting algorithm is heavily reliant on the selected values for its inputs. They determine the amount of resources r_0 on which all initial configurations are evaluated before a proportion of them are discarded. However, r_0 may not be enough to successfully distinguish between the promising configurations and the ones that should be discarded. Therefore, it can be challenging to manually set the input values.

Hyperband addresses this by repeatedly applying Successive Halving using different input values. Supported by theoretical guarantees for correctness and sample complexity, Hyperband follows a resource-allocation schedule determined from two parameters, η and R. The goal is for each run of SH to consume approximately B resources, which imposes a trade-off between the initial number of configurations n and the resources used before the first proportion of candidates is discarded, r_0. The algorithm is presented in Algorithm 3. Hyperband calls SH $s_{\max}+1$ times with different inputs, and each application of SH is referred to as a *bracket*. In the first bracket, $s = s_{\max}$ and the number of n is set to maximise exploration. As a result, r_0 is small and only 1 configuration is evaluated for the maximum number of resources R. In the last bracket, $s = 0$, a small pool of n configurations are all evaluated for $r_0 = R$ resources. During its run, Hyperband obtains a growing set of evaluations $E^{\text{HB}} = \{\mathbf{c}^{(i)}, r^{(i)}, l^{(i)}\}_i$, where r_i specifies the amount of resources used to obtain the evaluation.

Overall, Hyperband can automatically adapt to optimising different target functions, without requiring extra expert knowledge. While Hyperband has consistently demonstrated strong anytime performance, its final performance is typically weaker than Bayesian optimisation methods (Falkner et al., 2018). This is because candidates are always sampled uniformly at random, which does not focus on high-performing regions of the hyperparameter space.

Algorithm 3: Hyperband

```
Input: $R$, Maximum number of resource per configuration.
Input: $\eta$, Specifying the proportion of configurations to keep.
[1] $s_{\max} = \lfloor \log_\eta(R) \rfloor$
[2] $B = (s_{\max} + 1)R$
[3] foreach $s \in \{s_{max}, s_{max} - 1, ..., 0\}$ do
[4]     $n = \lceil \frac{B}{R} \frac{\eta^s}{(s+1)} \rceil$
[5]     $r = R\eta^{-s}$
[6]     SuccessiveHalving$(R, n, s, \eta)$
[7] end
[8] return Configuration with the smallest validation loss seen so far.
```

6.3 Augmenting Hyperband with ABLR

ABLR-based Bayesian optimisation and Hyperband have complementary strengths. ABLR can be used within BO to provide a scalable approach with competitive performance. Due to the flexibility of neural networks ABLR does not rely on an expert to provide problem-specific hyperparameter embeddings. Moreover, ABLR can use transfer learning to further improve the anytime and final performances of the resulting BO algorithm (Perrone et al., 2018). However, when used within a resource-unaware approach, which evaluates all hyperparameter configurations with R resources, the resulting optimisation algorithm incurs certain inefficiencies which hinder the algorithm's anytime performance.

On the other hand, Hyperband is a multi-fidelity approach which has demonstrated better anytime performance than resource-unaware algorithms. However, it randomly samples hyperparameter configurations, which can result in lower final performance, compared to methods which use adaptive sampling.

We augment Hyperband with adaptive sampling, based on ABLR. The resulting method, which we name HB-ABLR, achieves strong anytime and final performances, and can use transfer from previous evaluations to further increase both. We use HB to select different values for n and r, and modify its Successive Halving sub-routine as shown in Algorithm 4. We use ABLR to iteratively sample the initial pool of configurations, c at a time where c is the number of candidates that can be evaluated concurrently.

Before suggesting configurations, ABLR is fit on all previous evaluations E^{HB} from

Algorithm 4: SuccessiveHalvingWithABLR

```
Input: $R$, Maximum number of resource per configuration.
Input: $n$, Initial number of candidates.
Input: $s$, Number of iterations.
Input: $\eta$, Specifying the proportion of configurations to keep.
Input: $c$, Number of concurrent evaluations.
[1]  $r = R\eta^{-s}$
     /* Adaptively sample initial pool in the fist iteration.    */
[2]  $T = \{\}, L = \{\}, i = 0, n_0 = n, r_0 = r$
[3]  while $|T| < n$ do
[4]      $ablr = ABLR(E^{\text{HB}}, \{E^{\text{HB}}_{<t}\})$
[5]      $C = \text{suggest}(\min(c, n_0 - |T|), r_0, ablr, EI)$
[6]      $T = T \cup T$
[7]      $L = L \cup \{\text{run\_then\_return\_val\_loss}(t, r_0) : t \in C\}$
[8]  end
[9]  $T = \text{top\_k}(T, L, \lfloor \frac{n_0}{\eta} \rfloor)$
     /* Run the rest of the iterations as usual.                 */
[10] foreach $i \in \{1, ..., s\}$ do
[11]     $n_i = \lfloor n\eta^{-i} \rfloor$
[12]     $r_i = r\eta^{i}$
[13]     $L = \{\text{run\_then\_return\_val\_loss}(t, r_i) : t \in T\}$
[14]     $T = \text{top\_k}(T, L, \lfloor \frac{n_i}{\eta} \rfloor)$
[15] end
```

Figure 6.1: SH augmented with ABLR-based adaptive sampling, which we use within Hyperband.

the current HB run. Since the data $E^{\mathrm{HB}} = \{\mathbf{c}^{(i)}, r^{(i)}, l^{(i)}\}_i$ contains evaluations obtained with a different number of resources, we add r to the embedding of the hyperparameters $\varphi(c)$ and treat r as a *contextual input variable* because its value is provided by HB and not suggested by ABLR. Since r takes a limited number of discrete values, it is possible to embed it using a one-hot-encoding, or as $log(r)$ to reflect the fact that one can expect smaller performance gains as r increases. However, using one-hot-encoding wouldn't depict the ordering between the values of r, and manually selecting a transformation for a variable can be error-prone (Snoek et al., 2014). Instead, we normalise the values of r between $[-1, 1]$ and allow ABLR's neural network to learn the correct scaling for this dimension. Following Perrone et al. (2018), when optimising the parameters of ABLR, we start from the previously found optimal parameters, and use L-BFGS to train on all available evaluations. In practice, we also found that L-BFGS converges after a few iterations. Once ABLR is trained, we use it together with expected improvement (EI) to suggest c configurations. We optimise EI with respect to the hyperparameters, while keeping the resource contextual variable fixed $r = r_0$. This suggests that we are interested in hyperparameter configurations with promising performance after evaluating them on r_0 resources. Alternatively, we considered setting $r = R$, which would reflect the fact that we are interested in maximising the performance of a configuration after R resources. However, HB evaluates only a few configurations fully, making ABLR's predictions for $r = R$ less reliable. One could try a heuristic, in which suggestions are made using the maximum value of r, for which there are more than some manually-specified number of evaluations. Instead, in the interest of simplicity, we use $r = r_0$ for our algorithms' suggestions.

Our approach can take advantage of evaluations obtained on previous hyperparameter optimisation tasks. Let $\{E^{\mathrm{HB}}_{<t}\}$ be the sets of evaluations obtained by a HB-based algorithm on past HPO tasks. If such evaluations are available, we use them together with the new evaluations to train our ABLR model using the multi-task criterion from Eq. 6.1. This allows us to transfer knowledge by training the shared neural network of ABLR to represent the hyperparameters in a way which is useful for multiple tasks. Because we use a multi-task criterion, if the number of past HPO tasks is too big, the efficiency of updating the ABLR model could become a concern. In this case, it is possible to pre-train the neural network only on past evaluations and freeze its weights during the new HPO task, which would avoid having to optimise its weights across multiple tasks after every new batch of evaluations becomes available. A key benefit of our method is its ability to benefit from a large number of previous evaluations.

Combined with the method's flexibility, this allows for domain-specific knowledge to be effectively utilised, which can result in an increase in HB-ABLR's anytime and final performances.

At the moment, as in the original Successive Halving sub-routine, the configurations to be discarded from a pool of configurations are selected based on their observed performance after r_i resources. Alternatively, one could use the observed performances together with ABLR in order to make this selection. One way to do this by predicting each configuration's performance in a future iteration (r_{i+1} or R). This would be similar to HPO methods which try to approximate a ML algorithm's learning curve (Swersky et al., 2014; Klein et al., 2017). Since we use ABLR with a fully-connected neural networks, we cannot directly condition on the configurations' observed performances up to r_i. Instead, we can train ABLR on the new evaluations and then use it to make predictions about the future performances of the configurations in the current pool. Another way to make the selection would involve using the value of the acquisition function instead. For this purpose, we can again train ABLR on the newly obtained evaluations and then discard the configurations from the pool which have the lowest expected improvement for r_{i+1} resources, where the improvement can be calculated with respect to the best performance observed in the bracket. This second approach is more appealing, as it balances between exploration and exploitation. Utilising a surrogate model within HB in this way has not been discussed previously. While we think that this is an interesting approach, we made the decision to not incorporate it into our method for the sake of the method's simplicity. Therefore, we left evaluating this idea for future work.

Our method fulfils all of the desiderata for a HPO algorithm, described in Falkner et al. (2018). First, **strong anytime performance** is mainly achieved due to our use of HB, which allows us to evaluate poor-performing configurations after using fewer resources. In addition, our use of adaptive sampling and transfer learning both further improve the anytime performance. Second, HB-ABLR achieves **strong final performance** because of our use of adaptive sampling which allows the algorithm to focus on promising areas of the search space. The final performance can be further improved using our method by transferring knowledge from previous HPO tasks. Third, our algorithm can **effectively use parallel resources** as it can suggest and evaluate c candidates at a time. A potential bottleneck can arise if configurations take noticeably different amount of time. Since Successive Halving requires that all configurations in the current pool be evaluated before some of them are discarded, this could lead to idle compute time. If necessary, this could be addressed by simply starting a separate HB bracket, as suggested in Falkner et al. (2018). However, since all configurations are evaluated with the same amount of resources, we assume that their evaluation would take a similar amount of time, thus, we do not expect this modification to be necessary. Fourth, our algorithm uses a neural network to process the hyperparameter, which allows it to **scale** to a large number of hyperparameters, i.e. high-dimensional search spaces. Fifth, due to the use of a neural network, HB-ABLR is also **flexible** with regard to the types of hyperparameters that we search over. Furthermore, Perrone et al. (2018) show that ABLR can achieve good performance across different HPO tasks using the same neural architecture, which indicates that ABLR is **robust** with respect to its hyperparameters. Sixth, we used *simplicity* as a guiding principle in designing our algorithm. Separately, HB and ABLR can both be seen as simple in terms of the design choices necessary in order to apply to new problems, as well as the effort required to be implemented. To further this design philosophy, we maintain a single surrogate model across all HB brackets. Moreover, we train ABLR on all available evaluations, including the ones from past HPO tasks, without using a heuristic to select only a part of them. Seventh, our algorithm is **computationally efficient** as it scales linearly with the number of evaluations.

HB-ABLR also addresses the three challenges outlined in the introduction. First, the neural network used in ABLR can learn a suitable transformation for the hyperparameters of interest. This alleviates the necessity of injecting expert prior knowledge by defining hyperparameter-specific embeddings. Furthermore, our algorithm can address the cold-start problem by transferring knowledge from previous HPO tasks. Finally, HB-ABLR is a multi-fidelity algorithm, which reduces the inefficiency incurred by evaluating all hyperparameter configurations with a constant amount of resources.

6.4 Related Work on Hyperparameter Optimization

Among others, hyperparameter optimisation (HPO) has been applied to optimising the hyperparameters of random forests, support vector machines and neural networks (Shahriari et al., 2015). When applied to deep learning, HPO has led to performance improvements in image classification (Jafar and Myungho, 2020) natural language processing (Melis et al., 2017) and reinforcement learning (Zhang et al., 2021).

One approach to HPO is *grid search* in which a user first specifies a set of values for each hyperparameter, after which the search evaluates configurations from the Cartesian product of the specified sets. One shortcoming of this approach is that it can be difficult for an expert to manually identify potentially high-performing hyperparameter values. Alternatively, one can define a range of values for each hyperparameter and select configurations from the resulting search space uniformly at random. This approach is called *random search* and has been shown to outperform grid search (Bergstra and Bengio, 2012). Random search does not make any assumptions about the black-box function being optimised, e.g. the smoothness assumption. However, since all configurations are independent and identically distributed, random search does not exploit the potential information provided by past evaluations and does not focus on regions which can lead to higher performance.

On the other hand, Bayesian optimisation (BO) performs adaptive search by taking advantage of previous evaluations, and has been shown to be superior to RS for hyperparameter tuning of machine learning algorithms (Turner et al., 2021). Among others, BO has been successfully applied to tunning the hyperparameters of convolutional neural networks (Falkner et al., 2018) and has been used to win on 3 datasets in the AutoML challenge (Mendoza et al., 2016).
An important component of a BO algorithm is the choice of a surrogate model. Gaussian processes (GPs) are used most commonly (Shahriari et al., 2015) as they provide smooth and well-calibrated uncertainty estimates (Falkner et al., 2018). However, GPs require specialised kernels to be applied to more complex search spaces and require carefully set hyperpriors. Moreover, they scale cubically with the number of evaluated configurations. As an alternative, the Tree Parzen Estimator (TPE) (Bergstra et al., 2011) uses kernel density estimators (KDEs) to model densities over the configurations, instead of over the observed values. They support mixed continuous and discrete hyperparameter spaces and scale linearly with the number of observations. As originally described, TPE uses a hierarchy of one-dimensional KDEs which could fail to model important interactions between hyperparameters. This can be addressed by using a single multi-dimensional KDE, however, there is evidence that this approach fails at high dimensions (Lu et al., 2013; Falkner et al., 2018). ABLR (Perrone et al., 2018) can also scale linearly with the number of observations and is applicable to mixed spaces of hyperparameters. In contrast to TPE, ABLR does not have an inherent limitation on the hyperparameter dimensionality and is applicable to high dimensional inputs as well. Similarly, DNGO (Snoek et al., 2015) is a surrogate model which combines a neural network with an output Bayesian linear regression. DNGO first obtains the parameters of the neural network separately, and then the most suitable

BLR parameters. In contrast, ABLR optimises both sets of parameters jointly, which has been shown to lead to better performance (Perrone et al., 2018).
The scaling of the hyperparameters is another important consideration in Bayesian optimisation. For example, one can embed the learning rate j as $\log_{10}(j)$ to make sure that the values of interest are more evenly spread out. This also makes it easier for surrogate models to approximate the black-box function, as a small change in the input should always result in a change in the output of a similar magnitude. However, the correct scale of each hyperparameter may not be apparent. To address this, Snoek et al. (2014) propose to use are parameterised monotonic transformations, called *input warpings*, for each hyperparameter. In contrast to other surrogate models, ABLR contains a parameterised input transformation of the hyperparameters and does not require additional transformations.
Bayesian optimisation is prone to the cold-start problem. To address it, one needs to insert additional knowledge about the target black-box function, which makes it easier for a surrogate model to approximate it with a few data points. To address this, a line of work has focused on knowledge transfer using evaluations of previously encountered black-box functions, obtained from previous BO runs (Swersky et al., 2013; Bardenet et al., 2013; Yogatama and Mann, 2014; Wistuba et al., 2015; Feurer et al., 2015; Perrone et al., 2018). For example, (Feurer et al., 2015) propose to first evaluate hyperparameter configurations which have performed well in previous BO runs. Swersky et al. (2013) use a multi-task Gaussian process to fit the surrogate model on all available observations across multiple black-box functions. This approach is limited by the cubic complexity of GPs. On the other hand, if many observations across previous optimisations are available, the nonlinear mapping of ABLR can be trained offline, leaving only its BLR component to be trained. This results in a linear complexity in terms of the observed values of the target black-box function.

Multi-fidelity approaches detect poor-performing configurations early and stop their further evaluation. This leads to a more efficient use of resources and to better anytime performance. For example in Klein et al. (2017) the authors allow a configuration to be evaluated using a subset of the training dataset. They use a GP to predict the loss and the computational cost of evaluating the given configuration using the given percentage of the training dataset. The acquisition function then computes the amount of information gain per unit cost and is used to select both the next configuration to evaluate and the amount of training data to evaluate it with. Alternatively, one can evaluate a configuration using a different number of training iterations of the target machine learning algorithm, e.g. a different number of stochastic gradient descent updates. Freeze-thaw (Swersky et al., 2014) uses a GP to model the learning curve of a hyperparameter configuration, using a specifically designed kernel for training curves. They use the surrogate model to predict a configuration's final result from the observed performance after some iterations. Given a pool of configurations, the surrogate model is used to decide whether to pause the evaluation of a configuration, as well as which configuration to start or continue evaluating. Hyperband is a multi-fidelity approach which is agnostic about the type of resource being used.

Previous to our work, others have presented different ways of augmenting Hyperband by sampling the pool of configuration using Bayesian optimisation methods. Bertrand et al. (2017) use a GP as a surrogate model. Similar to our work, the authors model the budget as well as the hyperparameters to output a prediction of the validation loss. This allows their method to take advantage of all previous evaluations generated across Hyperband brackets. This approach shares the limitations of GPs described above. Alternatively, Wang et al. (2018) and Falkner et al. (2018) propose to use TPE as a surrogate model. Wang et al. (2018) use a separate TPE for each bracket and select the configurations in the initial pool sequentially using adaptive sampling, based on the performance of previous configurations within the bracket after using r_0 resources. As a result of the authors' design choice, their method cannot benefit from evaluations obtained in previous HB brackets. Instead, in BOHB (Falkner et al., 2018), a TPE is also fitted on some evaluations from previous brackets. The authors use all past evaluations which have been evaluated for $b_{\mathrm{c\_max}}$ resources, where $b_{\mathrm{c\_max}}$ is the highest number of resources for which there is a user-specified minimum number of evaluations available. In contrast, our work takes advantage of all previous evaluations across Hyperband brackets. Furthermore, the authors also show evidence that BOHB "struggles" when the hyperparameters' embedding is high dimensional. This could be partly attributed to the challenges of applying TPE to high dimensions, as discussed above. On the other hand, our method uses a neural network, which does not impose a limitation on the dimensionality. In contrast to all previous methods that augment Hyperband, our approach is capable of knowledge transfer from previous runs of hyperparameter optimisation. This allows us to warm-start the optimisation process, leading to better anytime performance. Following our work, Li et al. (2021a) propose a method which tries to model all evaluations obtained across the HB brackets. They train a separate surrogate for evaluations obtained with different budgets. Then they use a mixture-of-experts approach in order to combine the surrogates' predictions. In comparison, we only use a single surrogate to model all previous evaluations.

6.5 Experimental Evaluation

In this section, we evaluate our method's properties experimentally. We are interested in evaluating the advantage of combining HB and ABLR, as opposed to using these methods separately. Moreover, we would like to assess the benefit of training the surrogate on all previous evaluations within a single HB run, as opposed to a subset of them. Finally, we would like to evaluate the increase in performance gained by transferring knowledge from previous HB runs.

We compare our method to a number of competitive baselines, which we divide into two groups.
First, we consider resource-unaware approaches which always evaluate the blackbox function using the same, maximum amount of resources. These include random search (random) (Bergstra and Bengio, 2012) and GP-based BO (Snoek et al., 2012), implemented using GPyOpt (González and Dai, 2016). The kernel we used for GPs is Matérn-5/2 with automatic relevance determination hyperparameters, optimized by empirical Bayes (Rasmussen and Williams, 2006). We also compare our method to ABLR using the original implementation and with the default hyper-parameters, used in Perrone et al. (2018). Concretely, we use a fully-connected neural architecture with 3 hidden layers, where each has 50 hidden units and uses the $tanh$ activation function. We evaluate both ABLR without transfer learning (ABLR) and with transfer learning (ABLR-tr). Here, transfer learning is to be understood as reusing evaluations from previous HP optimization tasks, computed with a maximum number of resources.
Second, we consider resource-aware approaches, which assume that a function can be evaluated using different number of resources. We aim to compare our work to Hyperband and to previous method which extend it. Both Wang et al. (2018) and Falkner et al. (2018) propose to combine Hyperband with TPE-based BO. However, the latter, whose method is BO-HB, trains its surrogate on more evaluations and has demonstrated state-of-the-art performance across many experiments. Therefore, we compare to BO-HB, which we refer to as HBBO in our results to keep it consistent with notation used for the rest of the methods. We use the open-source implementation provided by the authors. We also compare to the combination of Hyperband and a GP-based BO (HB_GP). Our implementation is similar to the approach described in Bertrand et al. (2017), with the difference that they use a squared exponential kernel,

while we employ Matérn-5/2. The main difference between them is that the former assumes a higher degree of smoothness on the target black-box functions (Rasmussen and Williams, 2006). HB_GP models model and use the budget the same way as our method, making the choice of a surrogate the only difference.

For our method, we use the same architecture as the one for ABLR. We encode the resources used as a contextual variable r, normalised within $[-1,1]$. We evaluate our method's performance without transfer (HB-ABLR) and with transfer (HB-ABLR-tr). All methods which have BO-based sampling use Expected Improvement as the acquisition function. All HB-based methods use $\eta = 3$. For ABLR-tr and HB-ABLR-tr, use use transfer learning data resulting from 1 run of random search and HB, respectively, per previous optimisation task.

Our experimental protocol follows a leave-one-task-out procedure, where a BO problem is solved for a given held-out task using the data from the $T-1$ remaining tasks. We thereafter report results averaged over, both, T leave-on-task-out folds and 30 random replications of the experiments. The performance of each baseline for a certain number of resources is defined by the minimum performance found up to that point. Inspired by Golovin et al. (2017), we aggregate results across tasks by first normalising each performance according to the final performance of the random search baseline within each task (random search then has unit final performance across tasks). This allows us to better capture the relative performance gap between HP optimisation methods across tasks, irrespective of the performance magnitude, which can vary greatly across tasks. To aggregate the normalised performances of a method across tasks and random replications, we compute the mean and the standard error. We plot the normalised performances per amount of resources used, as shown in Figures 6.2, 6.3 and 6.4. When plotting performances, we apply a log-scale to better illustrate the difference between the competing methods. Each x axis in our plots represents r/R , where r is the number of resources used by the HP optimisation method, and R is the maximum number of resources used to evaluate a single hyperparameter configuration. We apply log scale to better distinguish between methods' performance at the early stages of optimisation.

6.5.1 Tuning Stochastic Gradient Descent

First, we evaluate our methods' ability to tune the hyperparameters of stochastic gradient descent (SGD). The machine learning (ML) algorithm whose performance we want to optimise is linear regression, optimised using SGD. For this purpose we define a linear regression model with parameters $\theta \in \mathbb{R}^{p\times 1}$, which transforms an input $\mathbf{x} \in \mathbb{R}^{p\times 1}$ into a scalar prediction. One task of the ML algorithm is specified by a training dataset $D^{\mathrm{tr}} = \{(\mathbf{x})_i^{\mathrm{tr}}, y_i^{\mathrm{tr}}\}_{i=1}^{N}$ and a validation dataset $D^{\mathrm{val}} = \{(\mathbf{x})_i^{\mathrm{val}}, y_i^{\mathrm{val}}\}_{i=1}^{N}$, where $N = 81$ and $p = 9$. For a single ML task, we use a standard normal distribution to sample all the inputs as well as the target parameter θ^*. For the targets, we consider a noisy linear model $y_i = \mathbf{x}_i^\top \theta^* + \varepsilon_i$. The ML algorithm uses SGD to optimise the parameters by minimising the training loss $\mathcal{L}^{\mathrm{tr}}$. Concretely, SGD solves:

$$\operatorname*{argmin}_{\theta\in\mathbb{R}^{p\times 1}} \sum_{i=1}^{N} \mathcal{L}_i^{\mathrm{tr}}(\mathbf{x_i^{\mathrm{tr}}}, \mathbf{y_i^{\mathrm{tr}}}, \theta) \text{ where } \mathcal{L}_i^{\mathrm{tr}}(\mathbf{x_i}, y_i, \theta) = \frac{1}{2}(\mathbf{x}_i^\top \theta - y_i)^2$$

, using the following update rule at step k:

$$\begin{cases} \mathbf{v} = \gamma\mathbf{v} + \frac{\nu}{1+\nu\lambda k} \nabla \mathcal{L}_i^{\mathrm{tr}}(\mathbf{x}_i^{\mathrm{tr}}, y_i^{\mathrm{tr}}, \theta^{(k)}) \\ \theta^{(k+1)} = \theta^{(k)}) - \mathbf{v} \end{cases} .$$

Here, we use the momentum $\gamma \in [0.3, 0.999]$, the learning rate $\nu \in [0.001, 1.0]$ and $\lambda \in [0.001, 1000.0]$ as hyperparameters of the SGD, as advocated in Bottou (2012). Given a ML task, the hyperparameter optimisation (HPO) task is to tune the hyperparameters of SGD, namely γ, ν and λ, in order to minimise the validation root mean squared error (RMSE):

$$\mathcal{L}^{\mathrm{val}}(\theta) = \sqrt{\frac{\sum_{i=1}^{N} (\mathbf{x}_i^{\mathrm{val}} \cdot \theta - y_i^{\mathrm{val}})^2}{N}}.$$

We define a unit of resource as three SGD updates and set the maximum amount of resources to $R = 243$. We generate $T = 30$ tasks using the method described above.

Our results are shown in Figure 6.2. From the resource-unaware BO methods, we see that GP's performance is marginally better than that of random. ABLR and ABLR-tr exhibit much better anytime and final performances. As expected, using HB results in a significant improvement in the anytime performance but, surprisingly, HB also achieves a similar final performance to ABLR and ABLR-tr. Methods which combine HB and BO demonstrate even further improvement in anytime performance. However, HB-GP has the same final performance as HB. Combined with the results of GP suggests that the Gaussian process used as a surrogate model is not suitable for this

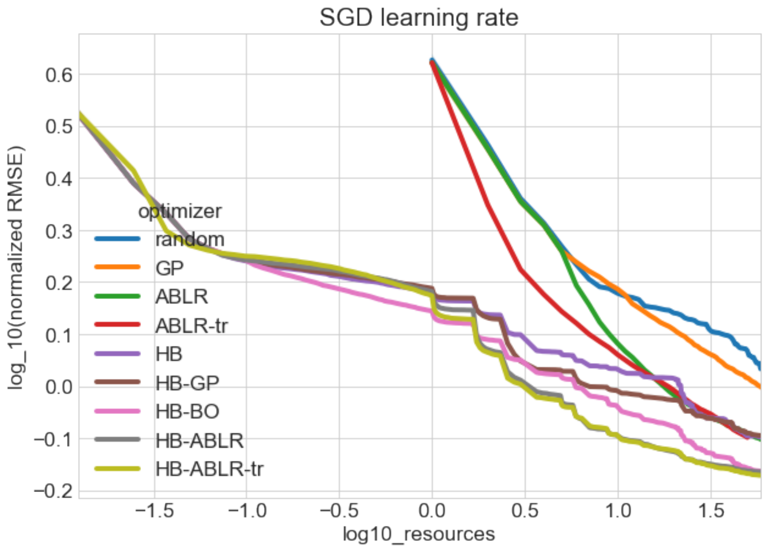


Figure 6.2: Comparison of our method to the baselines for the tuning of the SGD learning rate used to optimise linear regression problems. The x axis plots $log_{10}(r/R)$ where the resources used are the number of SGD iterations, and the maximum number of resources per configuration is $R = 243$.

task. Overall, HB-BO, HB-ABLR and HB-ABLR-tr demonstrate the best anytime and final performances, while HB-ABLR and HB-ABLR-tr are better after R resources are consumed. Surprisingly, both these methods have the same performance. In contrast, ABLR-tr achieves better anytime performance than ABLR. This suggests that, while knowledge transfer is possible on this task, HB-ABLR-tr does not benefit from it.

6.5.2 Tuning XGBoost binary classifiers

Now we consider a more complex setting which involves tuning more hyperparameters and more challenging datasets. We begin by selecting $T = 28$ binary classification datasets from the *libsvm* repository (Chang and Lin, 2011), namely: `australian`, `fourclass`, `german.numer`, `gina_agnostic`, `madelon`, `splice`, `breast-cancer`, `higgs_small`, `a6a`, `a7a`, `a8a`, `ijcnn1`, `mushrooms`, `phishing`, `rcv1.binary`, `skin_nonskin`, `spambase`, `susy`, `svmguide1`, `w6a`, `w7a`, `w8a`, `cod-rna`, `a1a`, `w1a`. Each dataset represents a separate ML problem which we solve using XGBoost (Chen and Guestrin, 2016). For each dataset, the HPO task is to optimise the hyperparameters of XGBoost in order to maximise the validation area under the curve (AUC). We tune the following 8 hyperparameters: eta $\in [0, 1]$, subsample $\in [0.5, 1]$, colsample_bytree $\in [0.3, 1]$, gamma $\in [2^{-20}, 64]$, min_child_weight $\in [2^{-8}, 64]$, alpha

$\in [2^{-20}, 256]$, lambda $\in [2^{-10}, 256]$, max_depth $\in [2, 128]$. We define a unit of resource as one round of XGBoost and set the maximum amount of resources to $R = 81$.

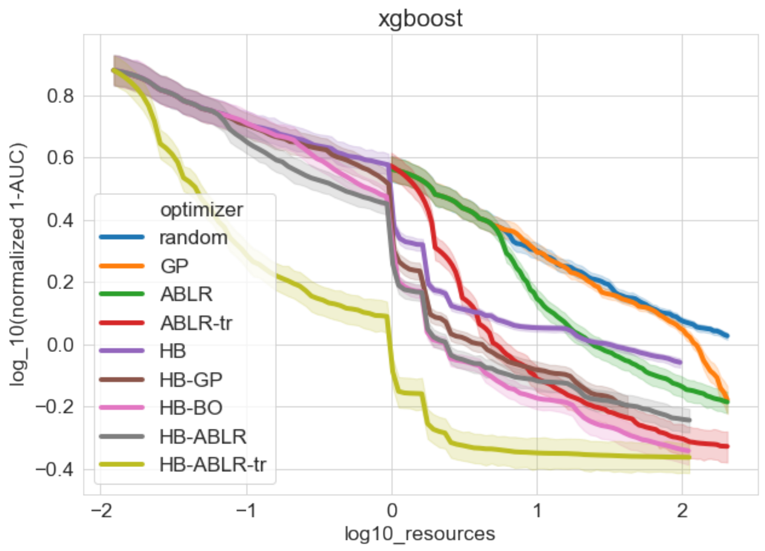


Figure 6.3: Comparison of our method to the baselines for the tuning of the hyperparameters of XGBoost binary classifiers. The x axis plots $log_{10}(r/R)$ where the resources used are the number XGBoost rounds, and the maximum number of resources per configuration is $R = 81$.

Our results are presented in Figure 6.3. It can be seen that GP has a similar anytime performance to random, but a stronger final performance. ABLR demonstrates an improvement over GP in terms of anytime performance. Interestingly, ABLR-tr is able to transfer knowledge from previous HB runs and achieves a performance which is comparable to the methods which combine HB and BO. HB demonstrates an improved anytime performance over the other BO methods, but its final performance is lower because it uses random search. HB-ABLR achieves an improvement over both HB and HB-GP, but it is outperformed by HB-BO. This suggests that combining HB with TPE is more suitable for the setting, even though it does not use all previous HB evaluations like HB-ABLR and HB-GP do. Finally, the results show that HB-ABLR-tr can benefit from knowledge transfer in this setting and achieves a significant improvement in terms of anytime performances, compared to the other methods.

In both our previous experimental settings we have considered above, tuning of SGD and XGBoost, we have not made assumptions about the search spaces. In particular, we have not used any prior information in the form of warping transformations, e.g. logarithmic transformations of some hyperparameter ranges spanning several order of magnitudes. To assess the effect of injecting prior knowledge in the form of

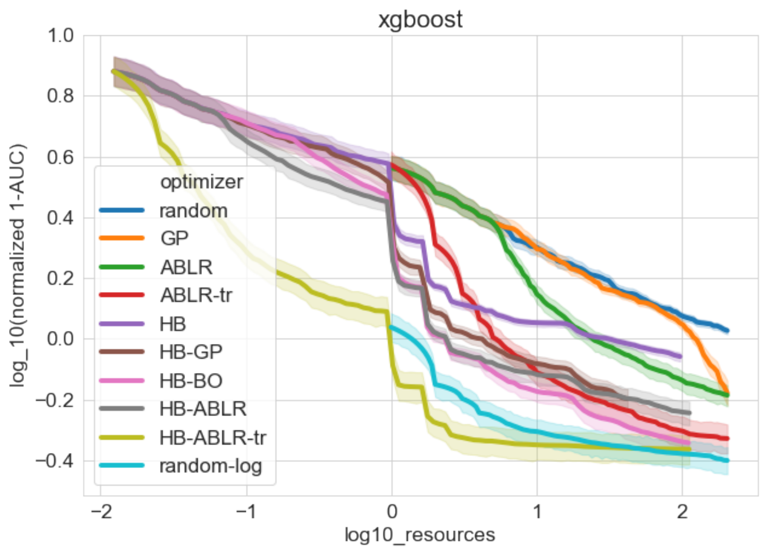


Figure 6.4: Comparing the performance of our baselines operating in the raw search space to the random search in the appropriately transformed search space specified by an expert (random-log). The x axis plots $log_{10}(r/R)$ where the resources used are the number XGBoost rounds, and the maximum number of resources per configuration is $R = 81$.

warping transformations, we change our setting for XGBoost by appropriately transforming some of its hyperparameter ranges. Concretely, we apply $\log_2$ scaling to the gamma, min_child_weight, lambda and alpha hyperparameters, and encode the discretised list of max_depth=$[2,3,4,6,8,11,16,23,32,45,64,91,128]$ as an integer corresponding to the indices in the list. We then run random search on this augmented hyperparameter search space and refer to the resulting method as random-log. Figure 6.4 shows the comparison of this new method to the rest. We can observe that appropriate transformations in the search space, if available, lead to a significant improvement. HB-ABLR-tr is the only method operating in the "raw" search space that can achieve a better anytime performance, and match the final performance of random-log. This suggests that the knowledge being transferred from previous tasks through a neural network relates to the correct embedding of the hyperparameters. Despite the transfer, we notice that random-log starts from a better performance than both ABLR-tr and HB-ABLR-tr. We attribute this to the fact that both ABLR-based methods sample their initial values randomly in the "raw" search spaces, which makes randomly sampling better-performing values highly unlikely.

6.5.3 Predictive Uncertainty with ABLR

In our preliminary investigation, we replicated the experiment conducted in Perrone et al. (2018), in which the target black-box function is given by a quadratic function: $f(\mathbf{x}) = \frac{1}{2}a_2||\mathbf{x}||_2^2 + a_1\mathbf{1}^\top\mathbf{x} + a_0$, where a_0, a_1 and a_2 are scalars sampled uniformly at random between $[0.1, 10]$, and $\mathbf{x} \in \mathbb{R}^3$. The HPO task is to find the $\mathbf{x}$ which minimises the black-box function. Our experiments matched the observations show in Figure 1a) in Perrone et al. (2018). Concretely, we observed that using a GP leads to better anytime performance than ABLR, in the absence of transfer. To investigate, we fit both a GP and an ABLR on the same set of observations and compared their predictions. We observed that ABLR's predictive uncertainty was much smaller than that of GP, meaning that ABLR was overconfident in its predictions. As a result, the BO procedure which uses ABLR, could prioritise exploitation over exploration. On the other hand, for our primary experiments, we observed that ABLR exhibited competitive performance, with ABLR-based methods outperforming GP-based method.

Further insight into this intermittent issue can be obtained by considering another line of work, namely *deep kernel learning* (DKL) (Calandra et al., 2016; Wilson et al., 2016a,b). These approaches use a neural network to obtain a latent representation of given inputs, which is in turn used by a GP to predict the output's distribution. *Neural linear* models represent a DKL approach which uses a linear kernel for the GP (Riquelme et al., 2018; Ober et al., 2021). Therefore, ABLR can be seen as a neural linear model, due to the equivalence between GPs with a linear kernel and Bayesian linear regression (Rasmussen and Williams, 2006). Ober and Rasmussen (2019) investigate the capabilities of neural linear models on simple regression tasks selected from the UCI and UCI "gap" datasets. They compare different neural linear models, which differ in the way their parameters are trained. One of the models is equivalent to ABLR, with the difference that the authors train the parameters using Adam instead of L-BFGS. Their results show that, on some of the datasets, ABLR's average test log-likelihood can be significantly improved by tuning the hyperparameters for ABLR's optimiser. This suggests that the uncertainty estimates of ABLR can be affected by the choice of ABLR hyperparameters for some tasks. In another paper, Ober et al. (2021) find that DKLs trained using the marginal likelihood, like ABLR, can still overfit on the training dataset. Furthermore, Tran et al. (2019) find that DKLs can be overconfident in their predictions, despite the Bayesian treatment of their last layer.

Overall, this suggests that ABLR can exhibit overconfidence on some tasks. While this may be addressed by tuning the hyperparameters of ABLR, doing so for a HPO task may be infeasible. As a result, the performance of HB-ABLR may not be better than BO-HB or HB-GP, in the absence of transfer from previous HPO tasks.

6.6 HB-ABLR for Modular Lifelong Learning

The section presents a discussion on how HB-ABLR can be used to improve previous modular lifelong learning algorithms. In Chapter 5 we outlined a general algorithm for modular lifelong machine learning, MOLL, described in Algorithm 1. Given a problem with index t, MOLL searches for the optimal path $\pi^* \in \Pi_t$ which leads to the best validation performance. Evaluating a single path π involves training any randomly initialised modules which the path specifies. This restricts the number of paths which can be evaluated, which necessitates a search strategy which can quickly identify promising paths and suggest them for evaluation. Our approach in Chapter 5 was to split the set of all possible paths Π_t into subsets and manually define a separate efficient search strategy for each subset. We explored two subsets Π_t^{PT} and Π_t^{NT}, containing paths capable of perceptual and non-perceptual transfer, respectively. However, this still leaves a big portion of potentially beneficial paths $\Pi_t^{\mathrm{rest}} = \Pi_t \setminus (\Pi_t^{\mathrm{PT}} \cup \Pi_t^{\mathrm{NT}})$ for which we have not defined a search strategy. For example, Π_t^{rest} contains paths which can simultaneously achieve perceptual and non-perceptual transfer by transferring both the first l_1 and the last l_2 modules, while letting the remaining $L-(l_1+l_2)$ modules be randomly initialised, where L is the number of modular layers in the given neural architecture. The subset Π_t^{rest} is also ignored by (Veniat et al., 2020; Ostapenko et al., 2021). Therefore, there is a need for a search strategy which can explore Π_t^{rest} efficiently. Such a search strategy could be combined with others' in order to improve the transfer learning capabilities of current scalable modular LML algorithms.

MOLL, as defined in Algorithm 1 can be seen as a resource-unaware algorithm for black-box optimisation, operating on a discrete search space. If target black-box function is the validation loss $\mathcal{L}_t^{\mathrm{val}}$ evaluated after training, the optimisation problem in Equation 5.1 is equivalent to:

$$\operatorname*{argmin}_{\pi \in \Pi_t} \mathcal{L}_t^{\mathrm{val}}(\pi).$$

Furthermore, one can evaluate $\mathcal{L}^{\mathrm{val}}$ after training the new parameters using a different number of resources (a different number of epochs), which allows for a multi-fidelity optimisation algorithm to be employed. Therefore, we can use HB-ABLR to search for the best path $\pi \in \Pi_t^{\text{rest}}$. The advantage of this approach is that it does not require a manually-defined search strategy tailored to paths in Π_t^{rest}. Moreover, a multi-fidelity approach would allow many more paths to be evaluated using the same amount of resources as the resource-unaware algorithms presented in Chapter 5.

One challenge of applying HB-ABLR is deciding how to encode a path so that it can be processed by ABLR's neural network, i.e. defining a mapping $\varphi(\pi)$. Let a path π specify L modules for each of the layers in a given neural architecture and let there be t options to choose from, for each of the L modular layers. A naive approach would be to use one-hot encoding to represent each of the L modules selected by π which would lead to a high-dimensional encoding of size tL. Since ABLR uses a fully-connected neural network, which does not incorporate knowledge about the structure of its inputs, having very high-dimensional inputs would increase the number of path evaluations necessary before the neural network can learn a transformation which is useful for processing unseen paths. Another approach would be to introduce a separate parameterised embedding for each module. Then, a path can be encoded as a concatenation of the selected modules' embeddings. To train ABLR's parameters, one would then optimise both the weights of the NN and the input's parameters. The resulting approach could lead to overfitting. Overall, in the absence of other expert-provided biases, applying HB-ABLR is likely to be ineffective without using transfer learning from past path optimisation tasks, i.e. path evaluations of past problems in the lifelong learning sequence. Therefore, HB-ABLR is unlikely to be effective for short sequences, such as the sequences with 6 problems used in Chapter 5. However, we expect it to be applicable to long sequences of problems, for which a large number of path evaluations will become available to transfer from.

The second challenge of applying HB-ABLR is determining how to suggest a path that should be evaluated. Computing the value of the acquisition function for all unevaluated path would be computationally infeasible. Typically, Bayesian optimisation algorithms make suggestions by optimising the surrogate model's input using the gradient of an acquisition function. Since our search space is discrete, the suggested input $c*$ is unlikely to correspond to an actual path. To address this, one could choose the unevaluated path whose embedding is the closest to the suggested input, minimising the distance between $c*$ and $\varphi(\pi)$.

Overall, applying HB-ABLR could by applied to modular lifelong learning in order to search through paths which are not considered by current modular LML algorithms. This would alleviate the need of manually specifying search strategies for subsets of paths. However, in the absence of additional prior knowledge, we expect to HB-ABLR to become useful for long problem sequences, as it needs many previous evaluations on which to train its surrogate model. Still, due to its multi-fidelity approach and its ability to transfer knowledge, HB-ABLR is the most applicable to modular LML our of the other HPO methods which we compare to in this chapter. We leave the application HB-ABLR to modular LML and the evaluation of the resulting approach, for future work.

6.7 Conclusion

In this chapter, we developed HB-ABLR which is a model-based multi-fidelity hyperparameter optimisation (HPO) approach. HB-ABLR addressed three challenges of HPO, namely, the necessity for a manually-designed special input featurisation (e.g. a suitable GP kernel), the cold-start problem and the inefficiency incurred by evaluating every considered element using the same constant number of resources. This allows our algorithm to exhibit better anytime performance than competitive baselines, as shown in our experiments. Finally, we presented a discussion on how HB-ABLR can be used to augment current scalable modular LML algorithms in order to improve their knowledge transfer capabilities.

Chapter 7

Conclusion

This thesis introduced a modular framework for lifelong machine learning (LML). Moreover, it developed two approaches, which can be used to improve the framework's computational efficiency. Here, I present a summary of our contributions and discuss possible future research directions.

In Chapter 4, we introduced a neurosymbolic framework for LML, called HOUDINI. We introduced a specific representation of neural networks as typed functional programs. We then showed that symbolic program synthesis can be applied to transfer learning and LML. Our method uses function types in order to reduce the search space to well-typed programs. We demonstrated that our approach fulfils almost all properties an LML algorithm should have. Compared to previous work, our method can operate across input spaces, perform non-perceptual transfer and avoid negative transfer. Given a new problem, HOUDINI searches for the problem-specific optimal program, which allows it to operate on different input spaces. The modularity of functional programs allows us to perform non-perceptual transfer by reusing higher modules which operate on a more abstract, latent space. Moreover, our algorithm can avoid negative transfer because we first evaluate the standalone version of each program we consider. Finally, the pre-trained modules can be combined in novel ways, which allows our framework to achieve high performance even with a small number of training examples.

In Chapter 5, we listed a set of desirable LML properties. Our main contribution is introducing a probabilistic search framework over module combinations, which allowed us to develop the first modular LML algorithm, which can achieve all of the listed properties, apart from backward transfer. For this purpose we developed two distinct probabilistic models which modelled the choice of pre-trained modules. For the first model, we showed how a module's input distribution can be efficiently approximated. For the second model, we introduced a kernel function between module combinations. Apart from this, we presented a benchmark suite which can be used to diagnose all of the listed desirable LML properties.

In Chapter 6, we augmented Hyperband Li et al. (2017) by replacing its random sampling with model-based adaptive sampling. Our choice of a surrogate model Perrone et al. (2018) allowed us to make use of previous evaluations from past searches. We demonstrated that our approach outperforms the state-of-the-art baselines in the hyperparameter optimisation (HPO) setting. As a secondary contribution, we phrased the search over module combinations performed by modular LML algorithm as black-box optimisation problem. We then presented a discussion on how the HPO method introduced in this chapter can be used to augment modular LML algorithms.

7.1 Future Work

The work presented in this thesis opens up a number of potential research directions. Some of them are discussed next.

One avenue is to extend HOUDINI to describe neurosymbolic systems as functional programs. Currently, all of the functions are implemented as neural modules. However, any symbolic function can be added to the library and then be used on a new problem. For example, HOUDINI could be used to describe the neurosymbolic method for visual question answering NS-VQA, presented out Yi et al. (2018). The resulting functional program should combine a convolutional network, an LSTM and a symbolic program executor. Overall, our framework could be modified to allow one to automatically discover a problem-specific neurosymbolic solution.

Another research question is how to augment the modular LML algorithm presented in Chapter 5 in order to allow for backward transfer. This would create the first deep lifelong learning algorithm which possesses all of the desired LML properties listed in this thesis. One advantage of our framework towards this goal is that neural modules are selectively reused only on relevant problems. It is possible that this makes it easier to achieve backwards transfer.

The scalable modular LML algorithm presented in Chapter 5 assumes that a problem-specific neural architecture is provided by the user. An interesting research direction is to investigate how to combine this algorithm with ideas from neural architecture search. This would further automate the process of applying our modular LML algorithm to sequences with disparate problems.

The method proposed in Chapter 6 performed well in the setting of hyper-parameter optimisation. The chapter also presented a discussion on how this method an be used to augment modular LML algorithms. Evaluating the merits of such a combination is another interesting research direction.

Bibliography

Agrawal, P., Girshick, R., and Malik, J. (2014). Analyzing the performance of multilayer neural networks for object recognition. In *European conference on computer vision*, pages 329–344. Springer.

Aljundi, R., Babiloni, F., Elhoseiny, M., Rohrbach, M., and Tuytelaars, T. (2018). Memory aware synapses: Learning what (not) to forget. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 139–154.

Aljundi, R., Lin, M., Goujaud, B., and Bengio, Y. (2019). Gradient based sample selection for online continual learning. *Advances in neural information processing systems*, 32.

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. (2016). Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 39–48.

Badue, C., Guidolini, R., Carneiro, R. V., Azevedo, P., Cardoso, V. B., Forechi, A., Jesus, L., Berriel, R., Paixao, T. M., Mutz, F., et al. (2020). Self-driving cars: A survey. *Expert Systems with Applications*, page 113816.

Baker, B., Gupta, O., Raskar, R., and Naik, N. (2017). Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. (2016). Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Bardenet, R., Brendel, M., Kégl, B., and Sebag, M. (2013). Collaborative hyperparameter tuning. In *International conference on machine learning*, pages 199–207. PMLR.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. (2018).

Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

Bengio, Y., Léonard, N., and Courville, A. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *25th annual conference on neural information processing systems (NIPS 2011)*, volume 24. Neural Information Processing Systems Foundation.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2).

Berner, J., Grohs, P., Kutyniok, G., and Petersen, P. (2021). The modern mathematics of deep learning. *arXiv preprint arXiv:2105.04026*.

Bertrand, H., Ardon, R., Perrot, M., and Bloch, I. (2017). Hyperparameter optimization of deep neural networks: Combining hyperband with bayesian model selection. In *Conférence sur l'Apprentissage Automatique*.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural network. In *International Conference on Machine Learning*, pages 1613–1622. PMLR.

Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer.

Bottou, L. and Cun, Y. (2003). Large scale online learning. *Advances in neural information processing systems*, 16.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2017). Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*.

Buzzega, P., Boschini, M., Porrello, A., Abati, D., and Calderara, S. (2020). Dark experience for general continual learning: a strong, simple baseline. *Advances in neural information processing systems*, 33:15920–15930.

Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208.

Calandra, R., Peters, J., Rasmussen, C. E., and Deisenroth, M. P. (2016). Manifold gaussian processes for regression. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 3338–3345. IEEE.

Caruana, R. (1997). Multitask learning. *Machine learning*, 28(1):41–75.

Caruana, R. (1998). Multitask learning. In *Learning to learn*, pages 95–133. Springer.

Chang, C.-C. and Lin, C.-J. (2011). Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27.

Chang, M. B., Gupta, A., Levine, S., and Griffiths, T. L. (2018). Automatically composing representation transformations as a means for generalization. *arXiv preprint arXiv:1807.04640*.

Chaudhry, A., Dokania, P. K., Ajanthan, T., and Torr, P. H. (2018a). Riemannian walk for incremental learning: Understanding forgetting and intransigence. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 532–547.

Chaudhry, A., Ranzato, M., Rohrbach, M., and Elhoseiny, M. (2018b). Efficient lifelong learning with a-gem. *arXiv preprint arXiv:1812.00420*.

Chaudhry, A., Rohrbach, M., Elhoseiny, M., Ajanthan, T., Dokania, P. K., Torr, P. H., and Ranzato, M. (2019a). Continual learning with tiny episodic memories.

Chaudhry, A., Rohrbach, M., Elhoseiny, M., Ajanthan, T., Dokania, P. K., Torr, P. H., and Ranzato, M. (2019b). On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*.

Chen, K. (2015). Deep and modular neural networks. In *Springer Handbook of Computational Intelligence*, pages 473–494. Springer.

Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM.

Chen, Z. and Liu, B. (2018). Lifelong machine learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 12(3):1–207.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2015). The loss surfaces of multilayer networks. In *Artificial intelligence and statistics*, pages 192–204. PMLR.

Cimpoi, M., Maji, S., Kokkinos, I., Mohamed, S., and Vedaldi, A. (2014). Describing textures in the wild. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3606–3613.

Clanuwat, T., Bober-Irizar, M., Kitamoto, A., Lamb, A., Yamamoto, K., and Ha, D. (2018). Deep learning for classical japanese literature. *arXiv preprint arXiv:1812.01718*.

Cohen, G., Afshar, S., Tapson, J., and Schaik, A. V. (2017). Emnist: Extending mnist to handwritten letters. *2017 International Joint Conference on Neural Networks (IJCNN)*.

Crawshaw, M. (2020). Multi-task learning with deep neural networks: A survey. *arXiv preprint arXiv:2009.09796*.

Csurka, G. (2017). Domain adaptation for visual applications: A comprehensive survey. *arXiv preprint arXiv:1702.05374*.

Cunningham, E. and Fiterau, M. (2021). A change of variables method for rectangular matrix-vector products. In *International Conference on Artificial Intelligence and Statistics*, pages 2755–2763. PMLR.

De Lange, M., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G., and Tuytelaars, T. (2019). A continual learning survey: Defying forgetting in classification tasks. *arXiv preprint arXiv:1909.08383*.

Delange, M., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G., and Tuytelaars, T. (2021). A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Deng, L. and Liu, Y. (2018). *Deep learning in natural language processing*. Springer.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Díaz-Rodríguez, N., Lomonaco, V., Filliat, D., and Maltoni, D. (2018). Don't forget, there is more than forgetting: new metrics for continual learning. *arXiv preprint arXiv:1810.13166*.

Donahue, J., Jia, Y., Vinyals, O., Hoffman, J., Zhang, N., Tzeng, E., and Darrell, T. (2014). Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pages 647–655.

Dong, X. and Yang, Y. (2020). Nas-bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*.

Draper, N. R. and Smith, H. (1998). *Applied regression analysis*, volume 326. John Wiley & Sons.

Du, S. S., Zhai, X., Poczos, B., and Singh, A. (2018). Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*.

Duong, L., Cohn, T., Bird, S., and Cook, P. (2015). Low resource dependency parsing: Cross-lingual parameter sharing in a neural network parser. In *Proceedings of the 53rd annual meeting of the Association for Computational Linguistics and the 7th international joint conference on natural language processing (volume 2: short papers)*, pages 845–850.

Ebrahimi, S., Elhoseiny, M., Darrell, T., and Rohrbach, M. (2019). Uncertainty-guided continual learning with bayesian neural networks. *arXiv preprint arXiv:1906.02425*.

Elsken, T., Metzen, J. H., Hutter, F., et al. (2019). Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20(55):1–21.

Falkner, S., Klein, A., and Hutter, F. (2018). Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR.

Farajtabar, M., Azizan, N., Mott, A., and Li, A. (2020). Orthogonal gradient descent for continual learning. In *International Conference on Artificial Intelligence and Statistics*, pages 3762–3773. PMLR.

Farquhar, S. and Gal, Y. (2018). Towards robust evaluations of continual learning. *arXiv preprint arXiv:1805.09733*.

Farquhar, S. and Gal, Y. (2019). Differentially private continual learning. *arXiv preprint arXiv:1902.06497*.

Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., Pritzel, A., and Wierstra, D. (2017). Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*.

Feurer, M., Springenberg, J., and Hutter, F. (2015). Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR.

Finn, C., Rajeswaran, A., Kakade, S., and Levine, S. (2019). Online meta-learning. In *International Conference on Machine Learning*, pages 1920–1930. PMLR.

Garcez, A. d. and Lamb, L. C. (2020). Neurosymbolic ai: The 3rd wave. *arXiv preprint arXiv:2012.05876*.

Garnelo, M. and Shanahan, M. (2019). Reconciling deep learning with symbolic artificial intelligence: representing objects and relations. *Current Opinion in Behavioral Sciences*, 29:17–23.

Gaunt, A. L., Brockschmidt, M., Kushman, N., and Tarlow, D. (2016a). Differentiable programs with neural libraries. *arXiv preprint arXiv:1611.02109*.

Gaunt, A. L., Brockschmidt, M., Singh, R., Kushman, N., Kohli, P., Taylor, J., and Tarlow, D. (2016b). Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*.

Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587.

Gnecco, G. (2012). A comparison between fixed-basis and variable-basis schemes for function approximation and functional optimization. *Journal of Applied Mathematics*, 2012.

Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., and Sculley, D. (2017). 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.

González, J. and Dai, Z. (2016). GPyOpt: A bayesian optimization framework in python. http://github.com/SheffieldML/GPyOpt.

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680.

Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., and Bengio, Y. (2013). An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*.

GPy (since 2012). GPy: A gaussian process framework in python. http://github.com/SheffieldML/GPy.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Gühring, I., Raslan, M., and Kutyniok, G. (2020). Expressivity of deep neural networks. *arXiv preprint arXiv:2007.04759*.

Gulwani, S., Polozov, O., and Singh, R. (2017). Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119.

Guo, Y., Shi, H., Kumar, A., Grauman, K., Rosing, T., and Feris, R. (2019). Spottune: transfer learning through adaptive fine-tuning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4805–4814.

Ha, D., Dai, A., and Le, Q. V. (2016). Hypernetworks. *arXiv preprint arXiv:1609.09106.*

Hadsell, R., Rao, D., Rusu, A. A., and Pascanu, R. (2020). Embracing change: Continual learning in deep neural networks. *Trends in cognitive sciences*, 24(12):1028–1040.

Harnad, S. (1990). The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1-3):335–346.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

He, X., Zhao, K., and Chu, X. (2021). Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622.

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531.*

Hospedales, T. M., Antoniou, A., Micaelli, P., and Storkey, A. J. (2021). Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence.*

Howard, J. and Ruder, S. (2018). Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146.*

Hu, R., Andreas, J., Rohrbach, M., Darrell, T., and Saenko, K. (2017). Learning to reason: End-to-end module networks for visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 804–813.

Huh, M., Agrawal, P., and Efros, A. A. (2016). What makes imagenet good for transfer learning? *arXiv preprint arXiv:1608.08614.*

Hung, S. C., Tu, C.-H., Wu, C.-E., Chen, C.-H., Chan, Y.-M., and Chen, C.-S. (2019). Compacting, picking and growing for unforgetting continual learning. *arXiv preprint arXiv:1910.06562.*

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer.

Isele, D. and Cosgun, A. (2018). Selective experience replay for lifelong learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Jacobs, R. A., Jordan, I. M. I., and Berta, A. G. (1990). Task decomposition through competition.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87.

Jafar, A. and Myungho, L. (2020). Hyperparameter optimization for deep residual learning in image classification. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 24–29. IEEE.

Jamieson, K. and Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In *Artificial intelligence and statistics*, pages 240–248. PMLR.

Jiao, L. and Zhao, J. (2019). A survey on the new generation of deep learning in image processing. *IEEE Access*, 7:172231–172263.

Johnson, J., Hariharan, B., Van Der Maaten, L., Fei-Fei, L., Lawrence Zitnick, C., and Girshick, R. (2017). Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2901–2910.

Johnson, W. B. (1984). Extensions of lipschitz mappings into a hilbert space. *Contemp. Math.*, 26:189–206.

Jordan, M. I. (1998). *Learning in graphical models*, volume 89. Springer Science & Business Media.

Jung, S., Ahn, H., Cha, S., and Moon, T. (2020). Adaptive group sparse regularization for continual learning. *arXiv preprint arXiv:2003.13726*.

Kanclerz, K., Miłkowski, P., and Kocoń, J. (2020). Cross-lingual deep neural transfer learning in sentiment analysis. *Procedia Computer Science*, 176:128–137.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. (2018). Neural architecture search with bayesian optimisation and optimal transport. *arXiv preprint arXiv:1802.07191.*

Kautz, H. (2020). The third ai summer, aaai robert s. engelmore memorial lecture, thirty-fourth aaai conference on artificial intelligence, new york, ny, february 10, 2020.

Kawaguchi, K. (2016). Deep learning without poor local minima. *Advances in neural information processing systems*, 29.

Kemker, R. and Kanan, C. (2017). Fearnet: Brain-inspired model for incremental learning. *arXiv preprint arXiv:1711.10563.*

Khetarpal, K., Riemer, M., Rish, I., and Precup, D. (2020). Towards continual reinforcement learning: A review and perspectives. *arXiv preprint arXiv:2012.13490.*

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2018). Reply to huszár: The elastic weight consolidation penalty is empirically valid. *Proceedings of the National Academy of Sciences*, 115(11):E2498–E2498.

Kirsch, L., Kunze, J., and Barber, D. (2018). Modular networks: Learning to decompose neural computation. *arXiv preprint arXiv:1811.05249.*

Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2017). Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial intelligence and statistics*, pages 528–536. PMLR.

Kornblith, S., Shlens, J., and Le, Q. V. (2019). Do better imagenet models transfer better? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2661–2671.

Kozma, R., Ilin, R., and Siegelmann, H. T. (2018). Evolution of abstraction across layers in deep learning neural networks. *Procedia computer science*, 144:203–213.

Krizhevsky, A., Hinton, G., et al. (2009). Learning multiple layers of features from tiny images.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *NIPS*.

Lample, G. and Charton, F. (2019). Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

LeCun, Y., Cortes, C., and Burges, C. (2010). Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2.

Lee Rodgers, J. and Nicewander, W. A. (1988). Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2017). Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816.

Li, L. and Talwalkar, A. (2020). Random search and reproducibility for neural architecture search. In *Uncertainty in Artificial Intelligence*, pages 367–377. PMLR.

Li, Y., Shen, Y., Jiang, J., Gao, J., Zhang, C., and Cui, B. (2021a). Mfes-hb: Efficient hyperband with multi-fidelity quality measurements. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 8491–8500.

Li, Y., Shen, Y., Zhang, W., Chen, Y., Jiang, H., Liu, M., Jiang, J., Gao, J., Wu, W., Yang, Z., et al. (2021b). Openbox: A generalized black-box optimization service. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 3209–3219.

Li, Z. and Hoiem, D. (2017). Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947.

Li, Z., Zhong, C., Wang, R., and Zheng, W.-S. (2020). Continual learning of new diseases with dual distillation and ensemble strategy. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 169–178. Springer.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2018a). Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34.

Liu, H., Simonyan, K., and Yang, Y. (2018b). Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.

Liu, X., Masana, M., Herranz, L., Van de Weijer, J., Lopez, A. M., and Bagdanov, A. D. (2018c). Rotate your networks: Better weight consolidation and less catastrophic forgetting. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 2262–2268. IEEE.

Liu, X., Zhang, F., Hou, Z., Mian, L., Wang, Z., Zhang, J., and Tang, J. (2021). Self-supervised learning: Generative or contrastive. *IEEE Transactions on Knowledge and Data Engineering*.

Lomonaco, V. and Maltoni, D. (2017). Core50: a new dataset and benchmark for continuous object recognition. In *Conference on Robot Learning*, pages 17–26. PMLR.

Long, J., Shelhamer, E., and Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440.

Lopez-Paz, D. and Ranzato, M. (2017). Gradient episodic memory for continual learning. *arXiv preprint arXiv:1706.08840*.

Lorenz, R., Monti, R. P., Violante, I. R., Faisal, A. A., Anagnostopoulos, C., Leech, R., and Montana, G. (2015). Stopping criteria for boosting automatic experimental design using real-time fmri with bayesian optimization. *arXiv preprint arXiv:1511.07827*.

Loshchilov, I. and Hutter, F. (2017). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Lu, L., Jiang, H., and Wong, W. H. (2013). Multivariate density estimation by bayesian sequential partitioning. *Journal of the American Statistical Association*, 108(504):1402–1410.

Makarova, A., Shen, H., Perrone, V., Klein, A., Faddoul, J. B., Krause, A., Seeger, M., and Archambeau, C. (2022). Automatic termination for hyperparameter optimization. In *First Conference on Automated Machine Learning (Main Track)*.

Mallya, A., Davis, D., and Lazebnik, S. (2018). Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 67–82.

Mallya, A. and Lazebnik, S. (2018). Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7765–7773.

Melis, G., Dyer, C., and Blunsom, P. (2017). On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*.

Mendoza, H., Klein, A., Feurer, M., Springenberg, J. T., and Hutter, F. (2016). Towards automatically-tuned neural networks. In *Workshop on automatic machine learning*, pages 58–65. PMLR.

Mirzadeh, S. I., Farajtabar, M., Pascanu, R., and Ghasemzadeh, H. (2020). Understanding the role of training regimes in continual learning. *Advances in Neural Information Processing Systems*, 33:7308–7320.

Misra, I., Shrivastava, A., Gupta, A., and Hebert, M. (2016). Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3994–4003.

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill.

Mitchell, T., Cohen, W., Hruschka, E., Talukdar, P., Yang, B., Betteridge, J., Carlson, A., Dalvi, B., Gardner, M., Kisiel, B., et al. (2018). Never-ending learning. *Communications of the ACM*, 61(5):103–115.

Mockus, J., Tiesis, V., and Zilinskas, A. (1978). The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2.

Mohammadi, H., Riche, R. L., Durrande, N., Touboul, E., and Bay, X. (2016). An analytic comparison of regularization methods for gaussian processes. *arXiv preprint arXiv:1602.00853*.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

Nagarajan, V., Andreassen, A., and Neyshabur, B. (2020). Understanding the failure modes of out-of-distribution generalization. *arXiv preprint arXiv:2010.15775*.

Neelakantan, A., Le, Q. V., and Sutskever, I. (2015). Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*.

Nekoei, H., Badrinaaraayanan, A., Courville, A., and Chandar, S. (2021). Continuous coordination as a realistic scenario for lifelong learning. In *International Conference on Machine Learning*, pages 8016–8024. PMLR.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning.

Nguyen, A., Yosinski, J., and Clune, J. (2016). Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks. *arXiv preprint arXiv:1602.03616*.

Nguyen, C. V., Li, Y., Bui, T. D., and Turner, R. E. (2018). Variational continual learning. In *International Conference on Learning Representations*.

Nguyen, V., Gupta, S., Rana, S., Li, C., and Venkatesh, S. (2017). Regret for expected improvement over the best-observed value and stopping condition. In *Asian conference on machine learning*, pages 279–294. PMLR.

Ober, S. W. and Rasmussen, C. E. (2019). Benchmarking the neural linear model for regression. *arXiv preprint arXiv:1912.08416*.

Ober, S. W., Rasmussen, C. E., and van der Wilk, M. (2021). The promises and pitfalls of deep kernel learning. In *Uncertainty in Artificial Intelligence*, pages 1206–1216. PMLR.

Olivas, E. S., Guerrero, J. D. M., Martinez-Sober, M., Magdalena-Benedito, J. R., Serrano, L., et al. (2009). *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques: Algorithms, Methods, and Techniques*. IGI Global.

Oquab, M., Bottou, L., Laptev, I., and Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724.

Ostapenko, O., Rodriguez, P., Caccia, M., and Charlin, L. (2021). Continual learning via local module composition. *Advances in Neural Information Processing Systems*, 34:30298–30312.

Ouali, Y., Hudelot, C., and Tami, M. (2020). An overview of deep semi-supervised learning. *arXiv preprint arXiv:2006.05278*.

Pan, P., Swaroop, S., Immer, A., Eschenhagen, R., Turner, R. E., and Khan, M. E. (2020). Continual deep learning by functional regularisation of memorable past. *arXiv preprint arXiv:2004.14070*.

Pan, S. J. and Yang, Q. (2009). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., and Wermter, S. (2019). Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71.

Park, D., Hong, S., Han, B., and Lee, K. M. (2019). Continual learning by asymmetric loss approximation with single-side overestimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3335–3344.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830.

Pellegrini, L., Graffieti, G., Lomonaco, V., and Maltoni, D. (2019). Latent replay for real-time continual learning. *arXiv preprint arXiv:1912.01100*.

Perrone, V., Jenatton, R., Seeger, M., and Archambeau, C. (2018). Scalable hyperparameter transfer learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 6846–6856.

Pham, H., Guan, M., Zoph, B., Le, Q., and Dean, J. (2018). Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, pages 4095–4104. PMLR.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). Improving language understanding by generative pre-training.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Rajasegaran, J., Hayat, M., Khan, S., Khan, F. S., and Shao, L. (2019). Random path selection for incremental learning. *Advances in Neural Information Processing Systems*, 3.

Rasmussen, C. E. and Williams, C. K. (2006). Gaussian processes for machine learning.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2019). Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789.

Reed, S. and De Freitas, N. (2015). Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.

Ren, P., Xiao, Y., Chang, X., Huang, P.-Y., Li, Z., Chen, X., and Wang, X. (2020). A comprehensive survey of neural architecture search: Challenges and solutions. *arXiv preprint arXiv:2006.02903*.

Riquelme, C., Tucker, G., and Snoek, J. (2018). Deep bayesian bandits showdown: An empirical comparison of bayesian deep networks for thompson sampling. *arXiv preprint arXiv:1802.09127*.

Ritter, H., Botev, A., and Barber, D. (2018a). Online structured laplace approximations for overcoming catastrophic forgetting. *Advances in Neural Information Processing Systems*, 31.

Ritter, H., Botev, A., and Barber, D. (2018b). A scalable laplace approximation for neural networks. In *6th International Conference on Learning Representations, ICLR 2018-Conference Track Proceedings*, volume 6. International Conference on Representation Learning.

Robins, A. (1995). Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146.

Rolnick, D., Ahuja, A., Schwarz, J., Lillicrap, T., and Wayne, G. (2019). Experience replay for continual learning. *Advances in Neural Information Processing Systems*, 32.

Rosenbaum, C., Klinger, T., and Riemer, M. (2017). Routing networks: Adaptive selection of non-linear functions for multi-task learning. *arXiv preprint arXiv:1711.01239*.

Rosenbaum, C. G. (2020). Dynamic composition of functions for modular learning.

Ruder, S. (2017). An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.

Russell, S. J. and Norvig, P. (2009). *Artificial Intelligence: a modern approach*. Pearson, 3 edition.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. (2016). Progressive neural networks. *arXiv preprint arXiv:1606.04671*.

Saenko, K., Kulis, B., Fritz, M., and Darrell, T. (2010). Adapting visual category models to new domains. In *European conference on computer vision*, pages 213–226. Springer.

Salman, H., Ilyas, A., Engstrom, L., Kapoor, A., and Madry, A. (2020). Do adversarially robust imagenet models transfer better? *arXiv preprint arXiv:2007.08489*.

Schwarz, J., Altman, D., Dudzik, A., Vinyals, O., Teh, Y. W., and Pascanu, R. (2018a). Towards a natural benchmark for continual learning. In *NeurIPS Workshop on Continual Learning*.

Schwarz, J., Czarnecki, W., Luketina, J., Grabska-Barwinska, A., Teh, Y. W., Pascanu, R., and Hadsell, R. (2018b). Progress & compress: A scalable framework for continual learning. In *International Conference on Machine Learning*, pages 4528–4537. PMLR.

Serra, J., Suris, D., Miron, M., and Karatzoglou, A. (2018). Overcoming catastrophic forgetting with hard attention to the task. In *International Conference on Machine Learning*, pages 4548–4557. PMLR.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and De Freitas, N. (2015). Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175.

Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

Sharif Razavian, A., Azizpour, H., Sullivan, J., and Carlsson, S. (2014). Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

Shin, H., Lee, J. K., Kim, J., and Kim, J. (2017). Continual learning with deep generative replay. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 2994–3003.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Silver, D. L., Yang, Q., and Li, L. (2013). Lifelong machine learning systems: Beyond learning algorithms. In *2013 AAAI spring symposium series*. Citeseer.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, M., and Adams, R. (2015). Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pages 2171–2180. PMLR.

Snoek, J., Swersky, K., Zemel, R., and Adams, R. (2014). Input warping for bayesian optimization of non-stationary functions. In *International Conference on Machine Learning*, pages 1674–1682. PMLR.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Storkey, A. (2009). When training and test sets are different: characterizing learning transfer. *Dataset shift in machine learning*, pages 3–28.

Swersky, K., Snoek, J., and Adams, R. P. (2013). Multi-task bayesian optimization. *Advances in neural information processing systems*, 26.

Swersky, K., Snoek, J., and Adams, R. P. (2014). Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*.

Tan, C., Sun, F., Kong, T., Zhang, W., Yang, C., and Liu, C. (2018). A survey on deep transfer learning. In *International conference on artificial neural networks*, pages 270–279. Springer.

Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7).

Thrun, S. (1998). Lifelong learning algorithms. In *Learning to learn*, pages 181–209. Springer.

Titsias, M. K., Schwarz, J., Matthews, A. G. d. G., Pascanu, R., and Teh, Y. W. (2019). Functional regularisation for continual learning with gaussian processes. In *International Conference on Learning Representations*.

Tran, G.-L., Bonilla, E. V., Cunningham, J., Michiardi, P., and Filippone, M. (2019). Calibrating deep convolutional gaussian processes. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1554–1563. PMLR.

Turner, R., Eriksson, D., McCourt, M., Kiili, J., Laaksonen, E., Xu, Z., and Guyon, I. (2021). Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In *NeurIPS 2020 Competition and Demonstration Track*, pages 3–26. PMLR.

Valkov, L., Chaudhari, D., Srivastava, A., Sutton, C., and Chaudhuri, S. (2018). Houdini: lifelong learning as program synthesis. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 8701–8712.

Van Engelen, J. E. and Hoos, H. H. (2020). A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440.

Veniat, T. (2021). Mntdp. https://github.com/TomVeniat/MNTDP.

Veniat, T., Denoyer, L., and Ranzato, M. (2020). Efficient continual learning with modular networks and task-driven priors. *arXiv preprint arXiv:2012.12631*.

Veniat, T. and Ranzato, M. (2021). Continual transfer learning benchmark. https://github.com/facebookresearch/CTrLBenchmark.

Wang, J., Xu, J., and Wang, X. (2018). Combination of hyperband and bayesian optimization for hyperparameter optimization in deep learning. *arXiv preprint arXiv:1801.01596*.

Wang, R., Su, H., Wang, C., Ji, K., and Ding, J. (2019). To tune or not to tune? how about the best of both worlds? *arXiv preprint arXiv:1907.05338*.

Wang, Z., Zoghi, M., Hutter, F., Matheson, D., De Freitas, N., et al. (2013). Bayesian optimization in high dimensions via random embeddings. In *IJCAI*, pages 1778–1784.

Weiss, K., Khoshgoftaar, T. M., and Wang, D. (2016). A survey of transfer learning. *Journal of Big data*, 3(1):9.

Welinder, P., Branson, S., Mita, T., Wah, C., Schroff, F., Belongie, S., and Perona, P. (2010). Caltech-ucsd birds 200.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

Wilson, A. G., Hu, Z., Salakhutdinov, R., and Xing, E. P. (2016a). Deep kernel learning. In *Artificial intelligence and statistics*, pages 370–378. PMLR.

Wilson, A. G., Hu, Z., Salakhutdinov, R. R., and Xing, E. P. (2016b). Stochastic variational deep kernel learning. *Advances in Neural Information Processing Systems*, 29.

Wistuba, M., Rawat, A., and Pedapati, T. (2019). A survey on neural architecture search. *arXiv preprint arXiv:1905.01392*.

Wistuba, M., Schilling, N., and Schmidt-Thieme, L. (2015). Learning hyperparameter optimization initializations. In *2015 IEEE international conference on data science and advanced analytics (DSAA)*, pages 1–10. IEEE.

Wołczyk, M., Zajac, M., Pascanu, R., Kuciński, Ł., and Miłoś, P. (2021). Continual world: A robotic benchmark for continual reinforcement learning. *Advances in Neural Information Processing Systems*, 34:28496–28510.

Wolpert, D. H. (1996). The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390.

Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

Yang, X., He, X., Liang, Y., Yang, Y., Zhang, S., and Xie, P. (2020). Transfer learning or self-supervised learning? a tale of two pretraining paradigms. *arXiv preprint arXiv:2007.04234*.

Yang, Y. and Hospedales, T. M. (2014). A unified perspective on multi-domain and multi-task learning. *arXiv preprint arXiv:1412.7489*.

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., and Tenenbaum, J. B. (2018). Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *arXiv preprint arXiv:1810.02338*.

Yogatama, D. and Mann, G. (2014). Efficient transfer learning method for automatic hyperparameter tuning. In *Artificial intelligence and statistics*, pages 1077–1085. PMLR.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? *arXiv preprint arXiv:1411.1792*.

Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzmann, M. (2019). Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142*.

Yu, T. and Zhu, H. (2020). Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689*.

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.

Zela, A., Klein, A., Falkner, S., and Hutter, F. (2018). Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *arXiv preprint arXiv:1807.06906*.

Zenke, F., Poole, B., and Ganguli, S. (2017). Continual learning through synaptic intelligence. In *International Conference on Machine Learning*, pages 3987–3995. PMLR.

Zhang, B., Rajan, R., Pineda, L., Lambert, N., Biedenkapp, A., Chua, K., Hutter, F., and Calandra, R. (2021). On the importance of hyperparameter optimization for model-based reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 4015–4023. PMLR.

Zhang, Y. and Yang, Q. (2017). A survey on multi-task learning. *arXiv preprint arXiv:1707.08114*.

Zhang, Y. and Yang, Q. (2021). A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering*.

Zhou, J. T., Pan, S. J., and Tsang, I. W. (2019). A deep learning framework for hybrid heterogeneous transfer learning. *Artificial Intelligence*, 275:310–328.

Zhu, M. and Gupta, S. (2017). To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*.

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., and He, Q. (2020). A comprehensive survey on transfer learning. *Proceedings of the IEEE*.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710.