Master Thesis

in Statistics

at the Ludwig-Maximilians-University Munich

Department Institute for Statistics

Data-free meta-learning via knowledge distillation from multiple teachers

Written by
Sebastian Gruber

Duty date
9th March 2021

Supervision
Prof. Dr. Volker Tresp
Ahmed Frikha

Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Die Arbeit wurde weder einer anderen Prüfungsbehörde vorgelegt noch veröffentlicht.

München, den 01.02.2021

.....
Sebastian Gruber

Acknowledgments

First of all, many thanks to Prof. Volker Tresp and Dr. Ulli Waltinger for making it possible to write this thesis as a Master's student at the Machine Intelligence Research Group of Siemens AG. Also, I want to thank Ahmed Frikha for his supervision in the form of valuable and appreciated feedback and his patience in our weekly discussions. With the handing in of this thesis, my 2.5 years at Siemens have come to an end, and every colleague in the different departments I worked with deserves a shoutout for the time spent together. Also, I have to thank Erdinger Alkoholfrei for keeping me well-nourished and energized. Last but not least, I want to thank my family for the support, especially once my childhood room turned into my daily office due to Coronavirus.

Abstract

This work addresses the novel and challenging problem of data-free few-shot learning. Previous few-shot learning works assume access to training data from auxiliary tasks to train a few-shot learner, i.e., a model able to learn new tasks with only a few examples. We consider a data-privacy-preserving version of the few-shot learning problem, where models trained on auxiliary tasks are available instead of the data itself. By the convention of knowledge distillation, these previously trained models are considered as teachers. We construct an extensive methodology to investigate possible solutions, including an algorithmic framework using data-free knowledge distillation to generate artificial data acting as training data from auxiliary tasks to enable few-shot learning. Baseline methods that leverage the teacher models challenge this framework and highlight promising alternatives. Several experiments are conducted to assess the solutions' performances and challenges on different variations of the problem. To the best of the author's knowledge, the problem has not been explored in the literature so far, and the proposed methods are novel solutions to a novel learning scenario.

List of Abbreviations

ANN	Artificial Neural Network
BN	Batch Normalization
DI	DeepInversion
FOMAML	First-Order Model-Agnostic
MAML	Model-Agnostic Meta-Learning Meta-Learning
MNIST	Modified National Institute of Standards and Technology
SGD	Stochastic Gradient Descent

Contents

1. Introduction

1.1. Motivation

In the last decade, deep learning has experienced an uprising hardly seen by other scientific fields [15]. A reason for its success is the wide range of possible applications to different fields. Examples are image recognition [30], learning a policy in a simulated environment [77], natural language processing [18] and speech recognition [28]. The results may even surpass human performance [78]. However, deep learning also has its limitations [56]. In most cases, to successfully train a deep learning model, big quantities of data are required. For this, the collection of high amounts of data in real-world or in simulations has to be performed, which require enormous compute resources in either ways. This becomes problematic for applications where data is intrinsically rare, its acquisition is expensive [3], or compute resources are not sufficient [37].

Classical scenarios for this are those where privacy, safety, or ethics issues are present, making the data instances with labels difficult or impossible to acquire [90]. A specific example is drug discovery. Here, new molecules' properties are searched for to identify new, useful drugs [3]. However, new molecules do not have many biological records on clinical candidates due to possible toxicity, low activity, and low solubility [90]. Other examples, where the target task only has few examples, are language translation [41], and cold-start item recommendation [88]. In the aforementioned scenarios, it is essential to learn effectively from a small number of samples. Due to the high cost and time consumption of manually labeling the data by human labor and the rarity of data, traditional deep learning methods cannot be successfully applied [11]. This is a stark contrast compared to the human visual ability to recognize new classes with very few annotated examples [11]. We refer to learning new tasks with limited amount of labeled instances as few-shot learning [11]. A task of this problem domain is also called a few-shot task.

Meta-learning provides an alternative paradigm compared to conventional learning algorithms to solve problems in the few-shot learning domain [33]. In meta-learning, a machine learning model gathers experience over several learning iterations over a distribution of tasks related to the target task [33]. This experience is then used to improve its future learning performance on the target task [33]. Meta-learning means 'learning-to-learn' [85] and can lead to various benefits such as data and compute efficiency. Thrun and Pratt [85] defines a learning-to-learn algorithm when the learner trained by the algorithm improves its performance at solving a task with respect

to the number of related tasks seen. In contrast, conventional machine learning improves its performance as more data instances from a single task are encountered [33]. Generally, meta-learning for the few-shot learning problem handles each task as an observation instance. A multitude of different tasks is required by prior meta-learning approaches. Often, the set of tasks consists in total of several thousands of unique data instances, even though each task includes only very few data instances [22] [79] [67]. Meta-learning has proven useful in areas spanning few-shot image recognition [22] [79], unsupervised learning [59], data efficient [20] [34] and self-directed [2] reinforcement learning, hyperparameter optimization [23] and neural architecture search [52] [72] [101]. Because different communities use the term meta-learning differently, multiple perspectives and interpretations can be found in the literature [33].

Performing few-shot learning requires big amounts of data from similar tasks. Due to data privacy concerns, these quantities may not be available. One possibility to protect data is to release pretrained models instead of data. A popular method to extract the knowledge from neural networks is knowledge distillation (KD). KD [32] was designed to transfer the task solving abilities learned by a large, highly parameterized model to a relatively simpler and smaller model. Possible realizations of the large model can be an ensemble of multiple models or a single model with a large parameter space, a long training phase, and strict regularization such as Dropout [81], or BatchNorm [38]. Generally, the large model is referred to as the teacher, while the smaller, newly trained model is called student [64]. Original knowledge distillation approaches use real data either from the teacher's training data distribution or a different transfer set to perform the distillation [64]. However, accessing the original training data of the teacher may not always be possible. Often the training dataset is too large for the given computational resources [53], e.g. ImageNet [74]. Another possible scenario is that datasets are proprietary and not shared publicly due to privacy or confidentiality concerns [64]. Furthermore, the data may be inaccessible due to General Data Protection Regulation (GDPR), IP restrictions, or the data is too difficult to extract again. For example GDPR protect the biometric data of people and healthcare data of patients [64]. In summary, data is highly precious in the modern information age and, as a consequence, access to data may not always be possible [64]. This is the main motivation for introducing data-free (or zero-shot) knowledge distillation approaches [53]. Data-free knowledge distillation (DFKD) tries to accomplish the same task as the original KD approaches, except that the teacher's training data is not available. Most DFKD approaches train the student on synthetic data generated by leveraging the teacher model [27].

Following the works regarding few-shot learning and data-free knowledge distillation, one of the present thesis's contributions is to formulate the novel problem of data-free few-shot learning. Concretely, we aim to train a few-shot learner without having access to data from related tasks, but, instead, by using multiple available teacher models that were trained on such tasks. For this, in the first stage, we use data-free knowledge distillation to extract knowledge from the given teachers in the form of generated

data. In the second stage, meta-learning is used to train a student model (the few-shot learner) from scratch on the generated data. Furthermore, we evaluate approaches using conventional learning and propose baselines for this novel problem, which use the teachers without relying on generated data.

The industrial scenario that motivated this thesis involves sensor data collected from different production machines during manufacturing. Usually, the labeling of such data requires specific human expert knowledge. This makes it very costly to construct the large labeled dataset necessary for conventional deep learning approaches. Furthermore, access to sensor data from similar production machines or processes, i.e., the tasks, may also not be possible, due to data privacy concerns or unavailability of the data, making traditional few-shot learning through meta-learning not possible. We assume access to models trained on these auxiliary tasks, which guarantees data privacy. We use these models as teachers for the approaches introduced by this thesis.

1.2. Thesis Structure

The present thesis is structured as follows. We begin by providing the reader with the background needed to understand the thesis' contributions in chapter 2. In the following chapter 3, an extensive overview of works handling related problems is given. Here, the differences between those works and the present thesis are also mentioned. The methodology developed to address the problem is described in chapter 4. This includes combining previous methods in a new algorithmic framework as well as novel baselines. Subsequently, the research questions investigated in this master thesis and the experiments designed to address these are defined in chapter 5. The results of these experiments are presented in chapter 6 and, thereafter, interpreted and discussed in chapter 7. At the end of the latter chapter, an outlook and suggested next steps guide future research regarding the defined problem setting. Finally, the conclusion of this work's contributions is summarized in chapter 8.

2. Background

In this chapter, the background necessary to understand the addressed problem and the methodology introduced in chapter 4 is presented. We start with a short introduction to probability theory. Some of the definitions and principles are used throughout the thesis. This is followed by introducing machine learning, which acts as a backbone for the following deep learning approaches. Emphasis is laid on the optimization procedure. This is a crucial cornerstone for understanding meta-learning approaches and hyperparameter optimization, which is extensively used for the experiments in chapter 5. In the deep learning section 2.3, neural networks, meta-learning, and knowledge distillation are described. For neural networks, we define different layer types used in our work. In the meta-learning subsection 2.3.4, a general description is given, followed by an introduction of an optimization-based and metric-based approach. Here, an important baseline challenging meta-learning advances is also explained. Section 2.3.6 presents data-free knowledge distillation and focuses on a specific approach called DeepInversion.

2.1. Probability theory

In the following, the mathematical foundations of randomness in experiments and observations of the real world are defined. Furthermore, we formally define random variables and introduce joint and conditional probability distributions. These distributions are common targets for modeling in machine learning, including deep learning [8; 36]. Next, the expectation of a random variable is defined. We show a way of how to approximate this integral with the law of the large numbers. This law is implicitly used throughout the thesis when the mean of a function of data instances is calculated.

Probability space

A probability space is a mathematical construct providing a formal framework for stochastic observations or experiments. It consists of a sample space, a set of events, and a probability measure. Formal definitions are presented in the following.

Let Ω be an arbitrary non-empty set. When making observations with stochastic outcomes, we define Ω as the set of all possible outcomes. It is called the sample space of an experiment or stochastic observations. A specific example would be the case of flipping a coin, which gives $\Omega =$ {"head", "tails"}, the set of all possible outcomes.

Usually, describing each outcome on its own is not sufficient, so another set is required, enclosing all possible combinations of outcomes. A combination of outcomes is seen as a

subset of Ω and called an event in the following. Then, a σ-field $\mathcal{F} \subset 2^{\Omega}$ is defined as a set of events, so a set of subsets of Ω, with the conditions

- $\Omega \in \mathcal{F}$ ($\mathcal{F}$ contains the sample space),
- $A \in \mathcal{F} \Rightarrow (\Omega \setminus A) \in \mathcal{F}$ (if an event is in $\mathcal{F}$, so is its complementary event),
- $A_i \in \mathcal{F}(i = 1, \ldots, n) \Rightarrow \bigcup_{i=1}^{n} A_i \in \mathcal{F}$ (every arbitrary union of events in $\mathcal{F}$ is also in $\mathcal{F}$) [57].

Following the previous coin flip example, $\mathcal{F}$ would be {$\emptyset$, {"tails"}, {"head"}, {"head", "tails"}}.

Next, we require a measurement to specify the frequency of an event having an outcome as element relative to the whole sample space Ω. This frequency is called the probability of an event. The probability of an event is meant as the probability of it holding an arbitrary outcome of the sample space. Then, a probability measure is defined as a function $P : \mathcal{F} \to [0, 1]$ on a σ-field $\mathcal{F} \in 2^{\Omega}$ such that the following holds:

- $P(\emptyset) = 0$ (the probability of an outcome appearing in the empty event is zero),
- $P(\Omega) = 1$ (the probability of an outcome appearing in the sample space is one),
- $P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$ for $A_i, A_j \in \mathcal{F}$ and $A_i \neq A_j$ $(i, j = 1, \ldots, \infty)$ (if multiple events do not share the same outcomes, then the probability of all the events is equal to the sum of the probability of each event) [57].

An example for P in the previous coin flipping context would be $P(\{"head"\}) = 0.5$, $P(\{"tails"\}) = 0.5$, $P(\{"tails","head"\}) = 1.0$, and $P(\emptyset) = 0.0$.

To combine all the previous definitions, a probability space is then defined as the tuple $(\Omega, \mathcal{F}, P)$ of a sample space Ω, a σ-field $\mathcal{F}$, and a probability measure P [57]. A probability space is the minimum required definition to assign probabilities to stochastic outcomes and observations of the real world consistently.

Random variable

A random variable is a real-valued function of the sample space [57]. So far, we can assign probabilities to events. To do further calculations, e.g., for receiving the mean outcome, the outcomes must be numeric values. For this, we need a function mapping an outcome to a value while preserving the original events' context. In probability theory, these functions are called random variables [57]. We define a random variable $X : \Omega \to \mathbb{R}$ as a function on a σ-field $\mathcal{F}$ if

$$\{\omega \in \Omega | X(\omega) \geq a\} \in \mathcal{F} \tag{2.1}$$

for all $a \in \mathbb{R}$ [57]. In the case of the previous coin flipping scenario, "heads" could be transformed to "0" and "tails" to "1". Based on these numerical values, calculations can be done to gain insights into the experiments and observations. Furthermore, if X is a random variable and f a real-valued function, then $f(X)$ is also a random variable [57]. This is important as we do calculations on arbitrary functions, like losses (introduced in section 2.2), which are only considered random variables if their input is.

The probability distribution P_X of a random variable X for a set of outcomes $A \in \mathcal{F}$ is defined as $P_X(A) \coloneqq P(\{\omega \in \Omega | X(\omega) \in A\})$ [57]. For simplicity's sake, we write $P(X)$ for the function $x \mapsto P(\omega \in \Omega | X(\omega) = x)$, denoting the probability of X assuming the value x.

Conditional probability

In stochastic experiments and observations, we are usually interested in several different outcomes of the same trial. For example, on a single day, one may want to observe the weather and the temperature. Then, having access to one random variable (the weather) may influence the probabilities of observations of the other random variable (the temperature). The conditional probability is used if we depend a random variable on already known information of another random variable [57]. Both random variables follow a joint distribution, i.e., let $(X, Y) : \Omega \to \mathbb{R}^2$ be a vector of two random variables defined on the same probability space $(\Omega, \mathcal{F}, P)$ then

$$P_{(X,Y)}(A) = P(\{\omega \in \Omega | (X(\omega), Y(\omega)) \in A\}) \tag{2.2}$$

for arbitrary $A \subset \mathbb{R}^2$ [57]. If there is no A in a given context, then the last equation is often shortened to the more common term $P(X, Y)$ [8]. In machine learning, $P(X, Y)$ is usually a function of the form $(x, y) \mapsto P(X = x, Y = y) = P(\{\omega \in \Omega | X(\omega) = x, Y(\omega)) = y\})$ [97].

The conditional probability of the random variable Y given the random variable X is then

$$P(Y \in A | X \in B) = \frac{P(Y \in A, X \in B)}{P(X \in B)} = \frac{P(\{\omega \in \Omega | Y(\omega) \in A, X(\omega) \in B\})}{P(\{\omega \in \Omega | X(\omega) \in B\})} \tag{2.3}$$

for arbitrary $A, B \subset \mathbb{R}$ [57] as long as $P(Y \in B) > 0$.

In machine learning, the term $P(X|Y)$ is more common and denotes a function of the form $(x, y) \mapsto P(Y = y | X = x) = \frac{P(\{\omega \in \Omega | X(\omega) = x, Y(\omega) = y\})}{P(\{\omega \in \Omega | X(\omega) = x\})}$ [8; 97]. The joint distribution $P(X, Y)$ and the conditional distribution $P(Y|X)$ are the target of most learning algorithms introduced in the machine learning section 2.2 [8].

Finally, two random variables X and Y are said to be independent if $P(Y, X) = P(Y)P(X)$ [8]. This gives $P(Y|X) = P(Y)$, meaning if we have information of the outcome of X, the probabilities of the outcomes of Y are unchanged. Often, we talk about independent variables when assessing the outcomes of repeated trials of the same

observation or experiment. Observing the same coin flipped twice gives two independent random variables while observing the weather of two consecutive days does not. This is important for the approximation of the expectation introduced in the following subsection.

Expectation of a random variable

Having access to a random variable, it may be useful to aggregate all possible outcomes. The most prominent aggregation is the expectation of a random variable [57]. It can be interpreted as the mean value of the random variable over an infinite amount of observations. The definition of the expectation of a random variable X on a probability space $(\Omega, \mathcal{F}, P)$ is the abstract integral given by equation 2.4 [57].

$$\mathbb{E}_{X\sim P}[X] = \int_{\Omega} X dP = \int_{-\infty}^{\infty} x dP_X(x) \tag{2.4}$$

This integral is often either inaccessible or infeasible [97]. Thus, in machine learning applications, a random variable's expectation is not be calculated by integrating. The transformation to Riemann-integrals or countable sums as done in Marek Capinski [57] is skipped as it is not relevant for this thesis.

The expectation of a random variable can be approximated with the results of several independent trials conducted in an experiment. More specifically, the approximation can be made with the help of the *law of the large numbers*. The weak version is defined as follows.

Let $\mathbb{E}_{X\sim P}[X]$ be the expectation of a random variable X we want to approximate. Additionally, $\{X_i | \mathbb{E}_{X_i\sim P}[X_i] = \mathbb{E}_{X\sim P}[X], \mathbb{E}_{X_i\sim P}[(X_i)^2] \leq K, i = 1, \ldots, n\}$ (for $\infty > K \in \mathbb{R}$) is a set of independent random variables we observe in an experiment. Then, the expectation $\mathbb{E}_{X\sim P}[X]$ can be approximated by their mean, since

$$\lim_{n\to\infty} P(\{\omega \in \Omega | \, |\frac{1}{n}\sum_{1}^{n} X_n(\omega) - \mathbb{E}_{X\sim P}[X]| > \epsilon\}) = 0 \tag{2.5}$$

for an arbitrary $\epsilon > 0$ [57]. In other words, with increasing number of X_i $(i = 1, \ldots, n)$, the probability of the difference between the mean and the unknown expectation to be higher than some ϵ goes towards zero. So, the higher n, the better is the approximation. We use this approximation through calculating the mean throughout the present thesis.

2.2. Machine learning

Machine learning aims to construct computer programs that learn from data [36]. A succinct definition for this is given by Mitchell [61] [36]: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Machine learning allows us to handle a task T, that is usually too difficult to solve by programs relying on a manually defined set of rules. Here, the task T denotes the problem that should be solved once learning is done. Examples of different tasks are [36]:

- **Classification:** Specifying to which of k categories some input belongs
- **Classification with missing inputs:** Predicting the correct category as in classification, except the input is incomplete
- **Regression:** Predicting a numeric value given some input
- **Machine translation:** Given a sequence of symbols of a certain language, return a sequence of symbols in the target language
- **Anomaly detection:** Specifying if the input is an anomaly

The performance measure P quantifies the algorithm's ability to perform task T [36]. P is usually used to improve the algorithm during its learning phase w.r.t. the given task. It is also used to assess the final performance after the algorithm's learning is done. An example in the classification case is the accuracy, which measures the frequency of correctly assigned labels [36].

Furthermore, E is the given dataset an algorithm should use for its learning. This dataset acts as a noisy description of the dynamics of the task T. It is assumed it consists of independent random variables of unknown distributions. In the case of unsupervised learning, the algorithm aims to learn useful properties of the structure of the dataset [36]. Unsupervised learning is not used in the scope of this thesis, but an extensive overview is given in [8]. Another category of machine learning approaches is supervised learning, where each random variable is associated with a label [36]. The labels are often seen as realizations of a random variable since the labeling process is not always consistent and random at times [8]. Here, the learning stage consists in associating an input variable X with a target variable Y. Usually, Y is difficult to collect as it is not available automatically but must be provided during the dataset construction [36]. In most cases, X is high-dimensional, while Y is of lower dimension. We assume both variables follow a joint distribution, i.e. $(X, Y) \sim \mathcal{P}$ [8]. In the following, we focus on supervised learning since it is more relevant for the problem addressed in this thesis.

To learn the dependencies between the input and target variable of a dataset, a model f is required to approximate the joint distribution $\mathcal{P}(X, Y)$ (generative approaches) or the conditional distribution $\mathcal{P}(Y|X)$ (discriminative approaches) [8]. In the generative case, a prediction is received by applying the Bayes' theorem $\mathcal{P}(Y|X) = \frac{\mathcal{P}(X,Y)}{\mathcal{P}(X)}$. This approach has the advantage to sample new synthetic data using f but is also computationally more demanding [8]. Thus, the focus is laid upon discriminative models in the following. The performance measure P is a loss function $\mathcal{L}$. Let $\mathcal{F}$ be a set of possible models we consider for our learning algorithm. The theoretical objective of discriminative supervised machine learning is to find the best $f^* \in \mathcal{F}$ that minimizes the expectation of the loss function $\mathcal{L}$ regarding the unknown data distribution $\mathcal{P}$:

$$f^* = \arg\min_{f \in \mathcal{F}} \mathbb{E}_{(X,Y)\sim\mathcal{P}}[\mathcal{L}(f(X), Y)] \tag{2.6}$$

A common loss for a classification problem with C classes and model output space $\mathcal{P}_M \subset [0,1]^C$ is the *cross-entropy loss* $\mathcal{L}_{CE}$, defined in equation 2.7 [8].

$$\begin{aligned} \mathcal{L}_{CE} :& \mathcal{P}_M \times \{1, \dots, C\} \to \mathbb{R} \\ & (p, y) \mapsto -\sum_{i=1}^{C} \mathbb{1}_{\{i=y\}} log(p_i) \end{aligned} \tag{2.7}$$

One issue here is that the true distribution $\mathcal{P}$ is not available in practice. Another issue is that some subsets of $\mathcal{F}$ are dependent on other subsets. For example, the parameter space of a chosen model depends on the model architecture space. So a framework that can cope with these limitations is needed.

Let $\mathcal{D} = \{(X_i, Y_i) | i = 1, \dots, n\}$ of independent random variables be the available dataset. It is assumed each X_i and Y_i is identically distributed as the original, unknown variables X and Y we want to model.

The available dataset $\mathcal{D}$ is split into a training set $\mathcal{D}_{train}$, a validation set $\mathcal{D}_{val}$, and a test set $\mathcal{D}_{test}$. Each subset has its own purpose, which is illustrated in the following. The purpose of the training set is to fit a model to improve its performance on the task considered [8]. This process is also called (model) training or tuning. More specifically, it is used for optimizing parameters of a specific model or as input of a non-parametric function. The result of a model f tuned on the training set $\mathcal{D}_{train}$ is in most cases one of the two forms

$$f_{\mathcal{D}_{train}} : \begin{cases} x \mapsto f(x, \eta^*), & \text{parametric} \\ x \mapsto f(x, \mathcal{D}_{train}), & \text{non-parametric} \end{cases} \tag{2.8}$$

with

$$\eta^* \coloneqq \arg\min_{\eta \in H_f} \frac{1}{|\mathcal{D}_{train}|} \sum_{(x,y)\in\mathcal{D}_{train}} \mathcal{L}(f(x, \eta), y). \tag{2.9}$$

Equation 2.9 denotes the training solution η^* for the parametric case (H_f is the weight space of the model f) [7]. The non-parametric case does not use parameter optimization. So, we refer to the insertion of the training set into a kernel- or metric-like function as the training.

During training, we cannot update parameters other model parameters depend on. Such parameters are called hyperparameters and do not necessarily require a numeric scale or be part of the model [68]. They can also shape the training process. The hyperparameters are considered a fixed part of the model or the optimization algorithm to solve the objective in equation 2.9. A bi-level optimization scheme is required to optimize the hyperparameters, where updates with the training set loss are the inner level [8]. As a consequence, training can also be called inner optimization [68]. The outer level is called hyperparameter optimization, and, here, the hyperparameters are

adjusted based on the validation set loss [8]. Since we make several algorithm, model, and parameter decisions based on the model performance on the training and validation set, the loss of the best configuration is too optimistic. Thus, the test set is supposed to be only used for a realistic, final estimate of the trained model's performance with the best validation set loss [7] [36]. The test set loss approximates the model's predictive performance when applied to new data in practice.

2.2.1. Hyperparameter optimization

In this subsection, we give a brief overview of important concepts regarding hyperparameter optimization. We optimize the hyperparameters on the validation set based on the solution of the inner optimization. Usually, the procedure used to optimize the hyperparameter optimization objective is quite simple. Different models f are manually chosen or randomly sampled from a set $\mathcal{F}$, and optimized on the training set. The model yielding the lowest loss on the validation set $\mathcal{D}_{val}$ is chosen to be the best model $\hat{f}^*_{\mathcal{D}_{train}}$ of all considered models in $\mathcal{F}$. Formally, the hyperparameter optimization solution minimizes the objective given in equation 2.10 through the approximation of the expected loss [92] [7].

$$f^*_{\mathcal{D}_{train}} \coloneqq \arg\min_{f \in \mathcal{F}} \frac{1}{|\mathcal{D}_{val}|} \sum_{(x,y) \in \mathcal{D}_{val}} \mathcal{L}(f_{\mathcal{D}_{train}}(x), y) \tag{2.10}$$

$f_{\mathcal{D}_{train}}$ is a model f trained on a training set $\mathcal{D}_{train}$. One common approach to select $f \in \mathcal{F}$ is *Random Search* [4], which takes an interval $h_i = [h_i^{lower}, h_i^{upper}]$ of possible values for each hyperparameter i and then samples uniformly or log-uniformly a set of configurations $H \subset h_1 \times \cdots \times h_m, m \in \mathbb{N}$. This results in $\mathcal{F} = \{f^h | h \in H\}$ as the set of considered models. Here, f^h denotes a model with hyperparameter configuration h. Log-uniform sampling is useful for bounded hyperparameters. A concrete example is to sample out of the interval $[0.001, 1000]$. Around 99.9% of the sampled values would be > 1, even though the problem may require values < 1. Using log-uniform sampling would approximately sample half of the values > 1, while the others are < 1.

Another strategy is called *Grid Search* and evaluates all configurations in $H = h_1 \times \cdots \times h_m$ of different sets $h_i = \{h_{i1}, \ldots, h_{ik_i}\}$ $(k_i \in \mathbb{N}, i \in \{1, \ldots, m\})$, where each element is manually specified and corresponds to a hyperparameter i [4] [7].

The main challenge of machine learning algorithms is to perform well on new, previously unseen data [36]. This ability is called generalization [8] [36]. The generalization error is then defined as the expected value of the loss on the original distribution [36]. Since the original distribution is not available, the generalization error has to be estimated on a dataset not used during any optimization. This excludes the estimated performance on the training set since the model was already trained with it. Furthermore, the validation set has also been used for evaluating several hyperparameters and, thus, can give a too optimistic loss similar to the training set. To give an example, think about two models trained to predict a fair coin flip. Model A predicts "Heads" by 51 percent due to training

variances, while model B predicts "Heads" by only 50 percent. If the amount of "Heads" are slightly more than the amount of "Tails" in the validation set due to randomness, the hyperparameter optimization procedure would call model A superior, even though, as we know in this imaginary example, model B would give a better prediction.

Of course, we do not know the real solution in practice, but we can re-estimate the real model performance. This is done by evaluating the hyperparameter optimization solution $\hat{f}^*$ of equation (2.10) on the unseen test set and computing the test set error. The test set error is equivalent to the estimated generalization error [36]. Formally, the estimated generalization error on the test set $\mathcal{D}_{test}$ is then given by equation 2.11.

$$\widehat{\mathcal{GE}} = \frac{1}{|\mathcal{D}_{test}|} \sum_{(x,y)\in\mathcal{D}_{test}} \mathcal{L}(f^*_{\mathcal{D}_{train}}(x), y) \tag{2.11}$$

It is often the case that $\mathcal{L}$ in equation (2.11) is different than during hyperparameter and inner optimization because this value is the one reported and, thus, should allow interpretability [7]. For classification, a classic choice is the accuracy of predicting the correct class label. Additionally, it is of utter importance that no further modeling decisions are made on this value. Otherwise, any estimation of the true performance is lost [36].

2.3. Deep learning

Conventional machine learning algorithms already perform well on tabular data – data where each feature is a single dimension of the input [8]. Issues appear once they are applied to non-structured data, like images [36]. Each pixel and color channel is seen as a variable in an image, but features of an image are randomly distributed across pixels and areas. For example, a set of face images does not have the eyes on the same x- and y-coordinates across several samples. Additionally, the resolution of images increases with technological progress, giving single data instances with thousands of random variable dimensions. As a consequence, a different group of methods arised to handle tasks containing such unstructured data. This group of methods is called artificial neural networks (ANN) and may just be called neural- or deep networks in the machine learning context [36]. Similar to other parametric methods, an ANN can be seen as a function mapping the input variable X to a target variable Y by an arbitrary (but manually predefined) amount of adjustable parameters θ [36]:

$$P(Y|X) \approx ANN(X, \theta)$$

The parameters θ are also commonly called weights in the deep learning context [36]. Being inspired by human brains, neural networks are originally constructed as links between several single nodes (analogous to neurons in nature) [75]. This analogy does not explain the sophisticated and complex structure of neural network architectures developed in recent years [30]. A deep ANN consists of several layers with the intention that each layer transforms its input to another feature space

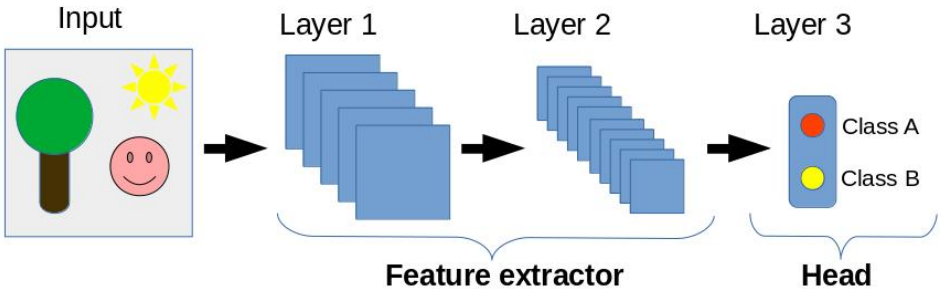


Figure 2.1.: Example scheme of a convolutional neural network with three layers. The first two layers transform their respective input into representations of more task relevant information. Due to this, random spatial information is removed and replaced by well-structured features. Each plane in the figure represents such a feature. These are then used by the output layer to make a prediction.

[36]. During this transformation, task-relevant information is preserved, while noise or irrelevant information is removed from the output. Stacking several such layers creates an operator decreasing the input size by several magnitudes to a usually smaller embedding of features. Usually, this embedding is purposely of small enough size to act as feature space for a light-weight classifier – the last layer of a neural network [36]. This final classifier is also referred to as the output layer, while the set of layers before it are called the *feature extractor* [70] [33]. The output layer is the simplest part of a neural network and is responsible for predicting Y or $P(Y|X)$ [36]. It is supposed to transform a feature vector $x_{feat} \in \mathcal{R}^n$ to a probability vector $p \in [0,1]^C$, with C being the number of classes in a classification task, with $\sum_i p_i = 1$. Usually, a fully connected layer (see section 2.3.2) or a similarity metric is used for this [42].

The feature extractor's main responsibility is to transform the high-dimensional, unstructured input into low-dimensional, well structured features $x_{feat} \in \mathcal{R}^n$. The features x_{feat} have to be structured so that the output layer can give accurate predictions [36]. The transformations in the feature extractor are done iteratively by processing the original input through different consecutive layers. Each intermediate layer takes the previous layer's output as the input and gives its output to the next layer. This architecture of several consecutive layers in a row is also why the field is called *deep* learning [36].

A sketch of a small neural network architecture for image data is given in figure 2.1. In the depicted example, a three layer neural network can be seen. Formally, its transformations would be of the following form with intermediate layer operators Φ giving results $hidden\_layer_1$ and $hidden\_layer_2$, an output layer operator σ and their

layer weights θ_1, θ_2 and θ_3, respectively:

$$hidden\_result_1 \leftarrow \Phi(input; \theta_1)$$
$$hidden\_result_2 \leftarrow \Phi(hidden\_result_1; \theta_2)$$
$$output \leftarrow \sigma(hidden\_result_2; \theta_3)$$

The forward processing of input through the different layers is also called *forward propagation* [36].

Due to such an architecture's recursive nature, it is possible to fit models for tasks of arbitrary complexity simply by increasing the number of layers and weights [14]. Nevertheless, there are also drawbacks of using models with such a high capacity:

- In general, neural networks have more weights than strictly required for a predictive task. This is because a small ANN with perfectly adjusted weights may not fully solve a task. On the other hand, starting with too many weights only has an increased computational effort as a disadvantage at first. As a consequence of this, ANNs are initialized with a high amount of weights. This makes them almost always capable of fitting more complex patterns than those appearing in the data. Therefore, after an unknown number of training iteration, ANNs start to fit random noise, introducing a high generalization error. This effect is called overfitting [36].
- With the number of weights of a neural network, the training's computational effort increases. Thus, the run time (and power cost) for training can be extremely high. Additionally, bigger datasets are required to fit the higher number of parameters, compared to conventional machine learning models [21]. These datasets may have several tens of thousands of images [17] [46] resulting in challenging memory requirements.
- Several layer operations back-to-back make the interpretability and transparency of intermediate layer results and model predictions extremely difficult. Interpreting neural networks is considered as a separate research direction [62].

2.3.1. Optimization

In this subsection, we give a brief introduction to finding a neural network model with good predictive performance on a task. This builds mostly on the elaborations in the previous section 2.2. First, we talk about how the weights are adjusted to find a good training solution. In general, it is not possible to find 'the' solution since usually the loss function w.r.t. the weights is non-convex and has an unknown amount of local solutions [36]. Furthermore, we discuss how different model architectures and hyperparameters are optimized.

Training

When a neural network is initialized, its parameters are in the general case initialized randomly. This model does not give accurate predictions, so the weights have to be changed in some way to give the correct predictions w.r.t. the input. For this, neural networks are trained to maximize the same objective as other machine learning algorithms, given by equation 2.9 in section 2.2.

To find a minimum of the estimated expected loss in the weight search space, a first-order optimization method called gradient descent is usually used. The more accurate second-order methods are not practical due to the quadratic memory consumption w.r.t. the number of weights [69].

The gradient descent algorithm consists of two steps. First, the objective function is derived w.r.t. the weights to receive a gradient. Second, the weights are updated with that gradient. These steps are applied repeatedly, each time moving the weights closer to a (local) minimum in the weight space. Formally, the update for the whole weight vector θ_{t-1} in iteration t looks like the following with an update step size ϵ [36]:

$$\theta_t \leftarrow \theta_{t-1} - \epsilon \nabla_{\theta_{t-1}} \mathcal{L}$$

ϵ is called *learning rate* in the context of training neural networks as it influences how quickly updates change the weights. The updates can theoretically be applied for an infinite amount of iterations, even if only marginal changes in the weights happen. At the start of the training, the training and validation set's loss functions decrease similarly. However, after some training updates, the validation set loss stales and increases in its value. This is due to fitting noise in the data. We can assume further training does not improve the predictive capabilities of the network. Therefore, once the validation set loss stops improving, training is stopped to prevent overfitting [36]. This procedure is called *early stopping*. In figure 2.2 one can see the training progress with overfitting, where early stopping was not done early enough.

Calculating the derivatives of the loss $\mathcal{L}$ is the most computationally expensive part of the whole training [36]. Between the weights and the objective function, multiple operations are performed for input data. To make the gradient calculations of all layers feasible, an approach called *backpropagation* is used [36]. Backpropagation allows information from the estimated loss to flow back through the ANN to calculate the weight gradient [36]. More specifically, it uses the chain rule and stores intermediate results to circumvent calculating the derivatives of higher layers several times to get the derivatives of lower layers. Even with the usage of backpropagation, there are too many weights in modern architectures to allow convergence with CPU computations. To increase computation speed, backpropagation and forward propagation are done in tensors on a GPU device [36]. This finally lowers the computation time to a feasible level, but, as a consequence, also possibly raises the following issue. GPU devices require their own memory, which is generally not as extendable and big as the normal working memory of a computer [76]. Consequently, calculating the gradients of a full dataset does not usually fit on a GPU device. This requires the use of another trick, which

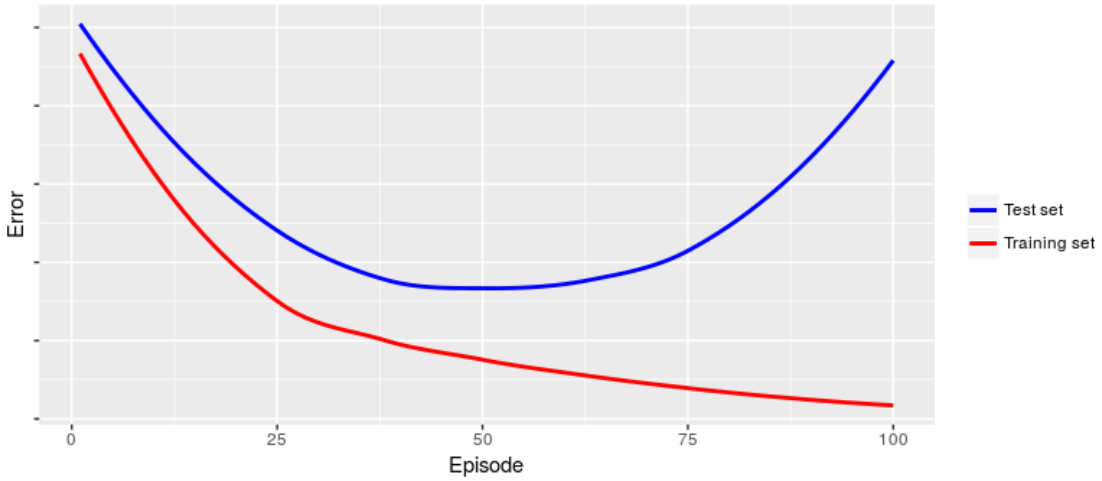


Figure 2.2.: A generic example of overfitting starting around episode 60 during the training phase.

splits the full dataset into minibatches. Minibatches are small subsets forming a random partition of the original set [36]. When calculating the derivatives for the weights update, the loss of only a single minibatch is used. Because each minibatch is also a sample of the original set, stochasticity is introduced in each weight update iteration, giving the procedure the name stochastic gradient descent (SGD) [36]. Generally, this introduced randomness does not worsen the generalization error.

Different optimization algorithms are applied in deep learning, besides SGD, even though they are mostly just an extension of the original version. A common one is *Adam*, which uses the first and second raw gradient update moment for the final weight update [44; 80]. A saddle point in the objective function gives tiny or zero gradient values. Consequently, no sufficiently sized weight update with (stochastic) gradient descent is possible to overcome this point. Adam makes use of momentum. Momentum increases gradient values if previous gradients had a similar direction in the weight space. When gradients gradually decrease in their value by encountering a saddle point, momentum keeps the gradient values high enough to overcome the saddle point. Due to momentum usage, Adam does not fail when faced with a saddle point in the objective landscape [44]. Compared to the SGD gradient update 2.3.1, an iteration t of Adam is formally defined as the sequence of operations in equation 2.12 [44]. Here, $\beta_1, \beta_2 \in]0, 1[$ act as update rates for the moments and $\eta > 0$ as a small constant for numeric stability.

$$
\begin{aligned}
g_t &\leftarrow \nabla_{\theta_{t-1}} \mathcal{L} \\
m_t &\leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &\leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &\leftarrow \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &\leftarrow \frac{v_t}{1-\beta_2^t} \\
\theta_t &\leftarrow \theta_{t-1} - \epsilon \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \eta}
\end{aligned}
\tag{2.12}
$$

Hyperparameters

Tuning the hyperparameters of a deep neural network is similar to conventional machine learning approaches and given by the same objective as in equation 2.10 [36]. In most cases, the biggest difference is the amount of time the training takes. This is one reason why we stop the training of a single hyperparameter configuration once it becomes clear that further training leads to decreasing performance. Some methods apply early stopping of the training more aggressively. E.g., the asynchronous successive halving algorithm (ASHA) and hyperband cancel training runs once it becomes likely a configuration does not beat the best other runs in terms of a specified loss [39] [48].

2.3.2. Layer types

The output layer of an artificial neural network has no scalability to higher input sizes due to the features of the input having to be semantically ordered [36]. It cannot deal with cases of features shifting positions as it appears in image data. As a consequence, the feature extractor is the main reason why convolutional neural networks can give accurate predictions [36]. A typical feature extractor consists of several layers, with possibly different layer types to pre-process any unstructured input, like images, for the output layer. An overview of the layer types used in this thesis is given in the following.

Fully connected layer

A fully connected layer transforms its input $x \in \mathcal{R}^n$ by multiplication with a weight matrix $W \in \mathcal{R}^{n \times m}$, addition of a bias vector $b \in \mathcal{R}^m$, and element-wise application of a non-linear function f (called activation function) [36]. The layer operation is then given by

$$
x \mapsto f(\, W^T x \, + \, b\,). \tag{2.13}
$$

A common choice for f is $ReLU : x \mapsto max(x, 0)$ (short for rectified linear unit) [36].

This layer type can lead to a high number of weight (every output is weighted with every input) and tends to generalize badly to unseen data, so it often only occurs once at the end of a neural network as the output layer [36; 30; 35]. In classification problems,

the activation function applied to the output of the last layer is usually the softmax operator

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{pmatrix} \mapsto \begin{pmatrix} \exp x_1 \\ \exp x_2 \\ \dots \\ \exp x_m \end{pmatrix} \left(\sum_{i=1}^{m} \exp x_i \right)^{-1} \tag{2.14}$$

applied to the m-dimensional output vector [36]. The softmax operator gives a meaningful output in the form of a probability vector.

Convolutional layer

In a fully connected layer, each additional input dimension adds m weights, with m being the output dimension. During training, these weights are adjusted. Fully connected layers lack scalability for higher-dimensional input. Furthermore, images usually grow quadratic in their pixel amount (because for a sharper image, one has to increase the resolution in the x-and in the y-direction). As a result, in the case of fully connected layers, the number of weights required would grow quadratically. To enable accurate predictions in this case, another layer type is introduced, called convolutional layer [36]. This layer type requires far fewer weights because, by design, it only has individual weights in a 'window' of predefined size [36]. These windows are usually called filters [50]. Each filter is moved over all neighboring variables of the input and creates new features as output for the next layer. All the outputs combined regarding a single filter are called feature map. In the context of image data, a neighborhood is meant as a part of an image cropped to the resolution $h \times h$.

A convolution is defined as an operation between an input x and a filter w at index t as

$$(x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) = \sum_{a=-\infty}^{\infty} x(t-a)w(a)[36]. \tag{2.15}$$

The second equation follows from the convolution being commutative.

For a two-dimensional input I, e.g., a gray scale image, the filter W is also two-dimensional and the convolution for pixel indices i, j is formulated as

$$(I * W)(i,j) = \sum_{a} \sum_{b} I(i-a, j-b)W(i,j)[36]. \tag{2.16}$$

Often, the input has an additional spatial dimension, denoting an image's color channel or the different feature maps of the previous layer. Then, the operation has to be further expanded to

$$(I * W)(i,j) = \sum_{a} \sum_{b} \sum_{c} I(i-a, j-b, c)W(i,j,c)[36]. \tag{2.17}$$

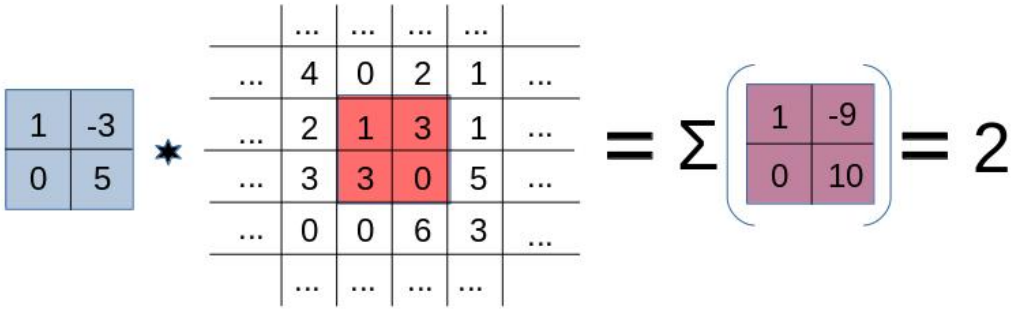


Figure 2.3.: One of the outputs of the convolution operator with kernel size $h = 2$. The input has two spatial dimensions and a single channel, resulting in weights $w \in \mathbb{R}^{2\times2\times1}$. The weights are in the blue box, a window of the input in the red box, and the intermediate element-wise products of these are shown in the purple box. The final result is the sum of all intermediate values.

The feature map of some input $I \in \mathbb{R}^{I_H \times I_W \times C}$ ($I_H, I_W, C \in \mathbb{N}$ input dimensions) and filter W is then returned by the function

$$\begin{aligned} \phi : \mathbb{R}^{I_H \times I_W \times C} &\to \mathbb{R}^{(I_H - h + 1)\times(I_W - h + 1)} \\ I &\mapsto ((I * W)(i,j))_{i,j\in\{1,\dots,h\}} + b \end{aligned} \tag{2.18}$$

with filter size h, filter weights $W \in \mathcal{R}^{h\times h\times C}$ and bias weight $b \in \mathbb{R}$. An example of this operation applied to a single neighborhood is depicted in figure 2.3.

Applying the same weights on different input variables is called weight sharing [36]. This is implicitly done by the definition of the convolutional layer operation ϕ in equation 2.18. An abstract depiction of this is shown in the case of one spatial dimension in figure 2.4. Here, only three weights have to be trained even though the input size is much larger. Similar to how a fully connected layer has several output dimensions, a convolutional layer has several feature maps as output resulting from several different filters.

Furthermore, applying the same weights on all inputs makes the output invariant to any shifts of the input [36]. In the case of image data, this is a critical property to detect similar features, e.g., a face shifted across different image samples. Because h has usually

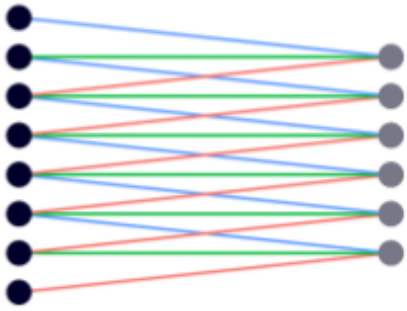


Figure 2.4.: Convolution operator with kernel size $h = 3$. Connections with the same colors share the same weights.

a low value ($h < 10$), this makes the convolutional layer very weight-efficient. Combined with the translation invariance, better generalization and computation time is achieved [36].

The drawback of having several feature maps as output is the lack of translation invariance of the new features [36]. Additionally, if the amount of filters is higher than the input channel size, the output can have a higher dimension than the input.

Consequently, an operation called *pooling* is used. It reduces the output to a smaller size. Pooling is often implied in a convolutional layer because it is exclusively – but not necessarily – used after the convolution [36]. Pooling works similar to the convolution operation. Furthermore, we replace the filter multiplication and the bias addition by a aggregation operator [36]. Each channel is handled independently of the other channels. The most common aggregation operator is the *max*-operator [36]. It returns the maximum value of its arguments. In all cases, pooling is supposed to make the output approximately invariant to small translations of the input [36]. However, especially max-pooling focuses on task-relevant information by giving the strongest signal in a neighborhood a pass-through to the output [36]. An abstract example of max-pooling is depicted in figure 2.5.

In neural networks, the pooling kernel size is often set to $k = 2$, so, reducing the output down to a quarter of the original size [30; 35]. By using pooling in several layers, the input is gradually reduced in its size layer by layer. Consequently, only a small feature vector resulting from the last convolutional layer is used as input for the output layer.

Batch normalization layer

During training an ANN, intermediate layer outputs often show a strong discrepancy in their value magnitudes [38].

To resolve this, the operation batch normalization (BN) was introduced in Ioffe and Szegedy [38]. The base idea is to normalize the outputs of each feature map of a whole batch of datapoints. This leads to more similarly distributed intermediate feature maps.

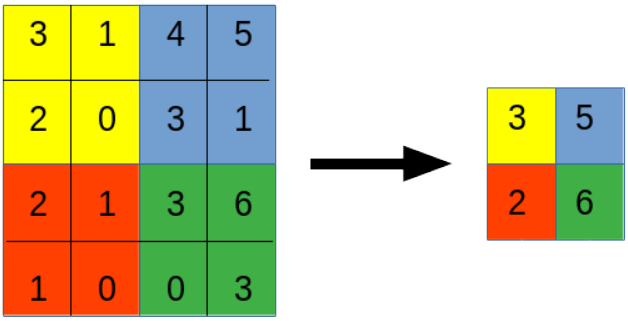


Figure 2.5.: Max-pooling operator with kernel size $h = 2$. Colors represent different neighborhoods. Unlike the convolution, the max operation happens in each channel on its own and not across all channels.

As a consequence, the training process of the whole network is stabilized and regularized [38]. Furthermore, training converges faster and is less sensitive to changes in learning rates. The formal definition of the BN operation on the c-th feature map of a convolutional layer output x is

$$x_c \mapsto \gamma_c \left(\frac{x_c - \mu_{B,c}}{\sqrt{\sigma^2_{B,c} - \epsilon}} \right) + \beta_c \tag{2.19}$$

with a small constant ϵ for numerical stability, mean $\mu_{B,c}$ and variance $\sigma^2_{B,c}$ of all values in a training (mini-)batch B. γ_c and β_c are scalar, real-valued weights optimized during the training, for an input channel c. From its first introduction, BN is used in almost all modern deep learning architectures due to its benefits [9].

To get meaningful batch statistics during validation and testing, the running mean of each batch statistic is calculated and stored during the training [38]. These values are then used for normalization on test data.

Residual layer

Deep neural networks are harder to train with an increasing number of layers [30]. As more layers are added to a network, the generalization error increases, even though the network has a higher capacity [30]. This observation is shown on the left side of figure 2.7.

To solve this problem, residual blocks were introduced by He et al. [30]. These result from a modification of the convolutional layer to allow the input of a layer to

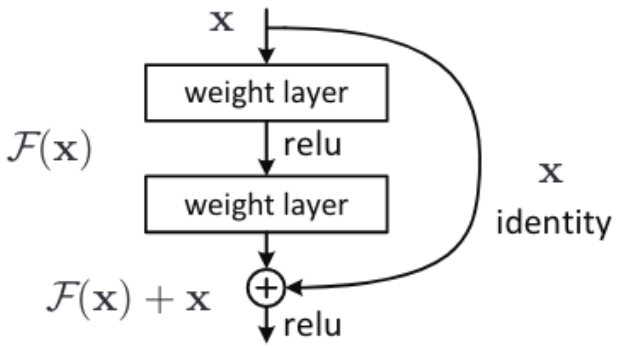


Figure 2.6.: Residual block with two convolutional layers as it appears in ResNet-34 or smaller. Figure used from [30].

circumvent the filter weights. Usually, a convolutional layer tries to fit the parameters of a full mapping $H(x)$ of some input x. In contrast to this, a residual operation fits only the residual mapping $F(x) := H(x) - x$ [30]. $F(x)$ can be approximated by several convolutional operations, as long as $H(x)$ and x are of equal dimensions. The architecture of a single residual block is depicted in figure 2.6.

A typical residual neural network consists of several residual blocks containing multiple convolutional layers each, with pooling applied between blocks. ReLU is applied once between the layers and once after the residuals are combined again with the input. BN is implicitly done right after each convolutional operation and is considered to be part of the weight layer. No pooling happens in a block. Such a model architecture is called *ResNet-X*, with X describing the number of layers in the full network. Bigger ResNets, like ResNet-50, use an even more sophisticated residual block. Since only small models are used in this master thesis, an extensive description of further residual blocks is not required to understand the results of the present thesis. More information is presented in the original work of He et al. [30]. Experiments showing how residual layers successfully enable deeper architectures can be seen in figure 2.7. The left and right sides depict the same model architecture. While the right side uses residual blocks, the left side does not.

Even though we only use smaller models (less than five million weights) in the experiment section, neural networks with residual blocks are widespread in recent neural network architectures [43]. As a consequence, ResNet based teacher models are extensively covered in the experiments.

2.3.3. Benchmark data sets

In this section, we give a short overview of benchmark datasets related to the ones used in the experiments (Chapter 5) this thesis.

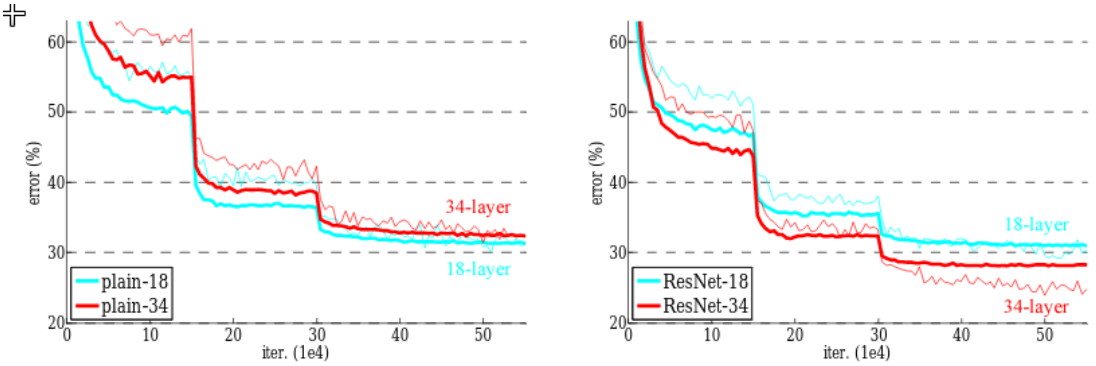


Figure 2.7.: Benchmark showing the improved scaling of residual layers in deep networks. On the left, normal convolutional layers have been used for a 18- and 34-layer network. On the right, residual layers were used in networks of equal size and number of weights. The thin lines represent the training errors, the bold lines the validation errors. [30].

MNIST

The MNIST (Modified National Institute of Standards and Technology) dataset consists of handwritten single digits [47]. It has 60000 train and 10000 test samples split evenly across ten different classes. The resolution of a single data instance is 28×28. The images have only a single channel since they are grayscale. MNIST is considered to be one of the easiest and earliest datasets used to benchmark image processing methods. Even older deep learning architectures show very high test accuracy with only little required computation time [47]. Sampled images can be seen in figure 2.8.

CIFAR-10

The CIFAR-10 (Canadian Institute For Advanced Research) dataset is a collection of 60000 colored images, split into 50000 training and 10000 testing instances [46]. There are ten classes in the set: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks, with each class being represented by 6000 images. Each image has a resolution of 32×32 with 3 channels. They show a wide variety of different features due to differing classes. This makes CIFAR-10 a lightweight dataset of medium difficulty.

ImageNet

ImageNet is a database containing over 15 million high-resolution images of around 22000 classes [45]. The images are not all of a similar resolution, so often only a smaller subset of ImageNet is used as a dataset. One such subset is ILSVRC (ImageNet Large-Scale Visual Recognition Challenge 2010). It contains 1.2 million training, 50.000 validation, and 150.000 testing images, spread across 1000 classes. It is common to scale all the images to a resolution of 224×224 with 3 channels [45]. This makes ImageNet and

Figure 2.8.: Sampled images of the MNIST dataset [47]. In recent years, the image values are used inverted, making the digits white and the background black.

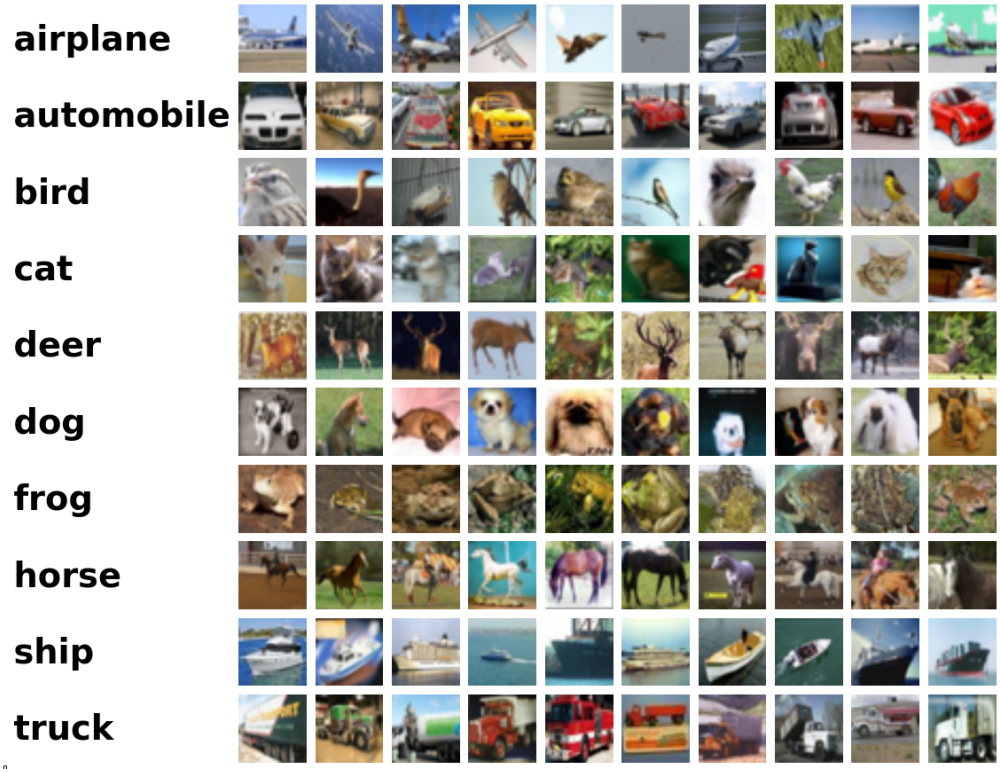


Figure 2.9.: Sampled images of the CIFAR-10 data set and their respective classes [46].

Figure 2.10.: Sampled images of the ImageNet data set and their respective classes [45].

its alterations a real challenge even for modern neural network architectures [1]. A few example images of ImageNet can be seen in figure 2.10.

2.3.4. Meta-learning

Meta-learning is commonly understood as the procedure of 'learning to learn' [33]. Its purpose is to improve a learning algorithm over several learning episodes. Recalling section 2.2, a conventional machine learning algorithm improves a model over several data instances. So, the task of meta-learning is not to train a model but to train a training algorithm. In this context, training the model is also called task adaptation [33]. Meta-learning makes use of a similar bi-level optimization scheme as conventional machine learning for hyperparameter optimization introduced in section 2.2. Here, the inner level is called the inner learning algorithm solving a task [33], e.g., image classification [46]. The outer level is called the meta-learning. It uses an outer (or upper/meta) algorithm to update the inner learning algorithm [33]. The outer updates are done in a way that improves an outer/meta objective based on the model training. An example is the model generalization error adapted to a task [33]. Based on these definitions, hyperparameter optimization can also be seen as a special form of meta-learning [33]. Most meta-learning algorithms in deep learning explicitly optimize the inner learning algorithm end-to-end w.r.t. the meta objective [33]. A gradient descent optimization algorithm usually conducts this since the model weights often define the search space [22] [79]. Meta-learning is applied with learning iterations on tasks sampled from a task distribution. This results in a base learning algorithm that generalizes to learning unseen tasks sampled from the same distribution [33].

2.3.5. Meta-learning for few-shot learning

In this subsection, we introduce the few-shot learning problem and how meta-learning is used to tackle it. In general, while meta-learning is considered as a class of algorithms, few-shot learning denotes a class of problems [33]. In this thesis, we only consider classification tasks for few-shot learning. The few-shot learning nomenclature is presented in the following. A (few-shot) task is considered as a small dataset containing only very few data instances. The training and test sets of a task are commonly called support and query sets, respectively, and form the task together [90]. Furthermore, we talk about a n-Way k-Shot task if the task has n classes and k data instances per class in the support set. A dataset in the context of few-shot learning is usually a set of (≥ 100) classes [16]. Based on these classes, a set of tasks is constructed. This set of tasks is considered as the task distribution of the few-shot learning problem. There are training, validation, and testing classes. They are used to construct the training, validation, and test tasks [16]. The training tasks are used for meta-learning training, also called meta-training [33]. The purpose of the validation tasks is to optimize the meta-learning algorithm's hyperparameters, also called meta-validation, in this thesis. After receiving a meta-trained inner learning algorithm, the test tasks are used to calculate the test error [33]. This stage is also called meta-testing. Here, the meta-trained inner learning algorithm is applied to a test task support set. The inner learning algorithm returns a model adapted to this support set. Then, the query set loss of the model is the generalization error. This error is a quantity of the inner learning algorithm's capability to adapt to new tasks with unseen classes.

The classes of the dataset must be strictly split into training, validation, and test classes. Otherwise, the generalization error of the task adapted model is computed on already seen data instances. As a consequence, it would be too optimistic.

To create the training, validation, and test tasks, algorithm 1 is executed on the respective classes [16]. In the implementation of Deleu et al. [16], these three sets of classes are already specified. Based on the returned sets of tasks, we can approximate the unknown task distribution.

To give a visual overview of how the final few-shot learning tasks are structured, figure 2.11 depicts the case of two training tasks and one test task, all with two classes ($n = 2$), four shots ($k = 4$), and one query set sample. Few-shot learning in practice requires many more training tasks to train a model, which generalizes to the test tasks successfully. The tasks, in summary, should cover several thousands of unique data points.

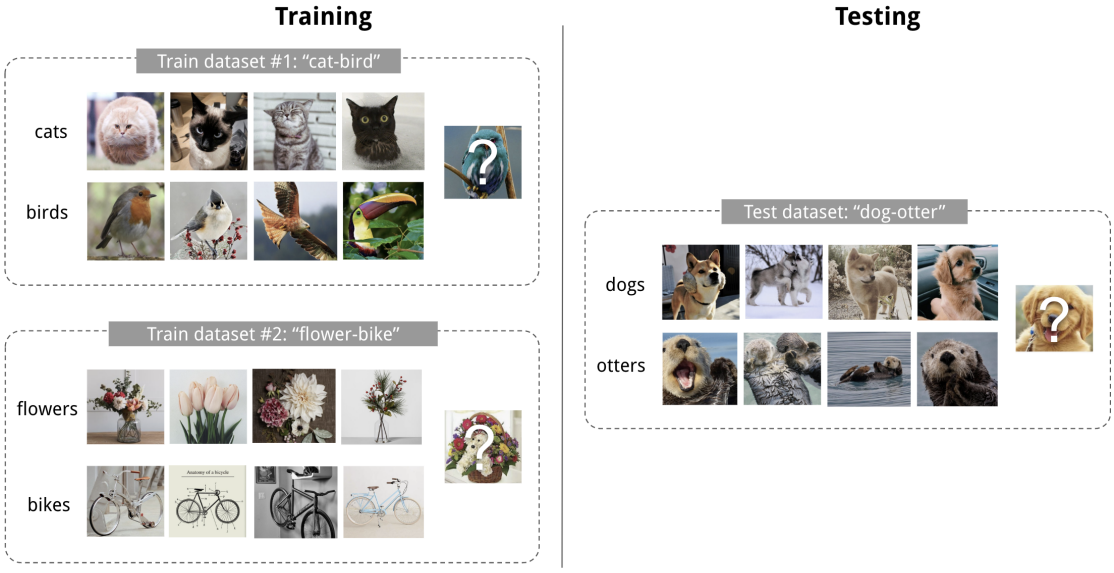
After defining the problem setup, we can formally define the optimization objective for a common meta-learning algorithm during the meta-training phase. The goal is to minimize the generalization error over the distribution of tasks $\mathcal{P}_\mathcal{T}$ concerning a set of hyperparameters of the inner learning algorithm similar to equation (2.8). The objective is given in equation 2.20 [33]. For consistency reasons with previous notations, we write $f_{\mathcal{D}_{sup},\theta}$ as the model f adapted to a support set $\mathcal{D}_{sup}$ by the inner learning algorithm with parameters θ.

Algorithm 1: Task sampling for few-shot learning as it is implemented by Deleu et al. [16]. $\hat{\mathcal{P}}_{class}$ is the empirical distribution of the available class instances.

Input : A set of classes $\mathcal{D}_{classes}$, n ways, k shots, l query shots
Output : A set of n-ways k-shots tasks with query set size l

$tasks \leftarrow \{\}$
\# iterate over all combinations of n sized pairs of classes
for *n-classes in n-combinations*($\mathcal{D}_{classes}$) **do**
    $support\_set \leftarrow \{\}$
    $query\_set \leftarrow \{\}$
    \# class index of the new task
    $i \leftarrow 0$
    for *class in n-classes* **do**
        for $\_ = 1 \dots k$ **do**
            \# sample support set class instance
            $x \sim \hat{\mathcal{P}}_{class}$ (no replacement)
            $support\_set \leftarrow support\_set \cup \{(x, i)\}$
        end
        for $\_ = 1 \dots l$ **do**
            \# sample query set class instance
            $x \sim \hat{\mathcal{P}}_{class}$ (no replacement)
            $query\_set \leftarrow query\_set \cup \{(x, i)\}$
        end
        $i \leftarrow i + 1$
    end
    $tasks \leftarrow tasks \cup \{(support\_set, query\_set)\}$
end
return *tasks*

Figure 2.11.: Example of a few-shot learning setup with two training tasks and one test task. The tasks are 4-shot 2-way, i.e. two classes with four examples each. This means the support set is of size $4 \times 2 = 8$. The query set is usually larger than the support set, but for demonstration purposes is only of size one here (the image covered by a question mark) [91].



$$\theta^* = \arg\min_{\theta \in \Theta} \mathbb{E}_{(\mathcal{D}_{sup}, \mathcal{D}_{query}) \sim \mathcal{P}_\mathcal{T}} [\frac{1}{|\mathcal{D}_{query}|} \sum_{(x,y) \in \mathcal{D}_{query}} \mathcal{L}(f_{\mathcal{D}_{sup},\theta}(x), y)] \tag{2.20}$$

θ^* is the best inner learning algorithm parameter configuration of the given task distribution. Θ is the search space of all parameters of the inner learning algorithm optimized by meta-learning. Examples for Θ are weight initializations [22], weights for an embedding function [79], or learning rates [51]. Most meta-learning algorithms try to solve the objective in equation 2.20 with a gradient descent like optimization algorithm [89; 79; 22], e.g., Adam introduced in equation 2.12. For this, the gradient is calculated on a batch of training tasks. This batch consists of B amount of tasks and approximates the true gradient by equation 2.21 [16; 22; 79].

$$\nabla_\theta \sum_{b=1}^{B} \frac{1}{|\mathcal{D}^b_{query}|} \sum_{(x,y) \in \mathcal{D}^b_{query}} \mathcal{L}(f_{\mathcal{D}^b_{sup},\theta}(x), y) \tag{2.21}$$

The full meta-learning training algorithm can be seen in algorithm 2. For consistency with previous notations, we imply with $f_{.,\theta}$ the model f combined with its inner learning algorithm with parameters θ. The inner learning algorithm solves its objective at once,

e.g., by a fixed amount of gradient steps. The task adapted model is denoted by $f_{\mathcal{D}_{sup},\theta}$ since it also depends on the parameters θ of the inner learning algorithm. The outer optimization algorithm minimizes the outer optimization loss $\mathcal{L}$ with a single gradient update on θ. For A_{outer} Adam is a common choice [79; 22]. A meta-trained learning algorithm $f_{.,\theta}$ with model f is returned at the end.

Algorithm 2: Meta-training algorithm for few-shot learning representing common, recent meta-learning approaches [25; 89; 79; 22; 71; 70; 73; 65].

Input : Base model f and inner learning algorithm with parameters θ, outer optimization updater A_{outer} (taking a loss value and a parameter set, and updating the latter to minimize the former), set of training classes $\mathcal{D}_{classes}$

Output : Meta-trained model with inner learning algorithm $f_{.,\theta}$

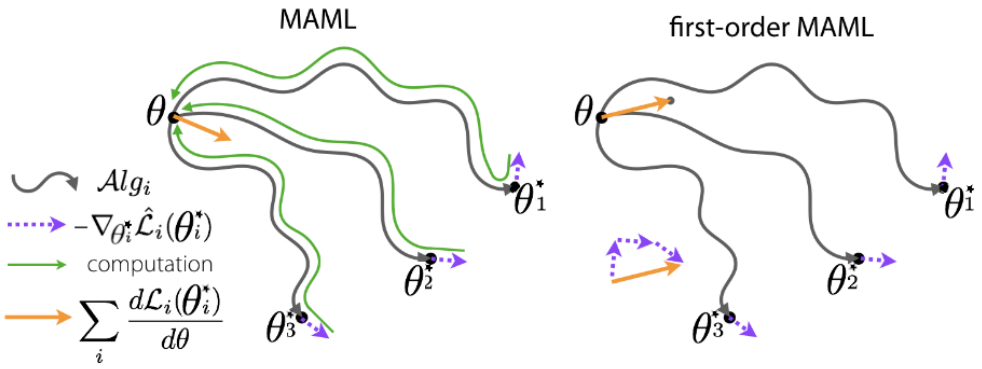
for *amount of iterations* **do**
  Sample batch of tasks $\{T_i\}_{i=1}^{B}$ from $\mathcal{D}_{classes}$ with algorithm 1.
  # set outer optimization loss
  $\mathcal{L} \leftarrow 0$
  for $(\mathcal{D}_{sup}, \mathcal{D}_{query})$ *in* $\{T_i\}_{i=1}^{B}$ **do**
    # do task adaptation with $f_{.,\theta}$ (approximately solving eq. 2.8)
    $f_{\mathcal{D}_{sup},\theta} \leftarrow (f_{.,\theta}, \mathcal{D}_{sup})$
    # update outer optimization loss
    $\mathcal{L} \leftarrow \mathcal{L} + \frac{1}{|\mathcal{D}_{query}|} \sum_{(x,y)\in\mathcal{D}_{query}} \mathcal{L}(f_{\mathcal{D}_{sup},\theta}(x), y)$
  end
  # do outer optimization step with A_{outer} on outer optimization loss $\mathcal{L}$
  $\theta \leftarrow A_{outer}(\mathcal{L}, \theta)$
end
return $f_{.,\theta}$

In general, the task adaptation depends on the meta-learning algorithm. Several modern meta-learning algorithms can be categorized in optimization-based and metric-based approaches [33]. Metric-based approaches, which use a non-parametric inner learning algorithm, do not use gradient updates [79]. On the other hand, optimization-based approaches do gradient updates on the model weights. Here, a gradient descent based optimization algorithm is a common choice [22]. Examples of these two classes of approaches, which we use in our experiments, are given in the following sections.

Optimization-based meta-learning

Optimization-based meta-learning makes use of gradient descent to approximate the inner optimization objective (2.9) [22; 33]. Usually, only a fixed amount of adaptation iterations are done on each task. An example for the hyperparameters of the inner learning algorithm are the weight initialization of the model before task adaptation [22]. Based on this, the outer optimization does gradient updates to receive the optimal weight

Figure 2.12.: Abstraction of optimization paths Alg_i of MAML and FOMAML [71]. θ is the weight initialization and θ_i^* the parameters adapted to task i after a fixed amount of gradient steps. The green line denotes the backpropagation path. FOMAML skips this calculation.



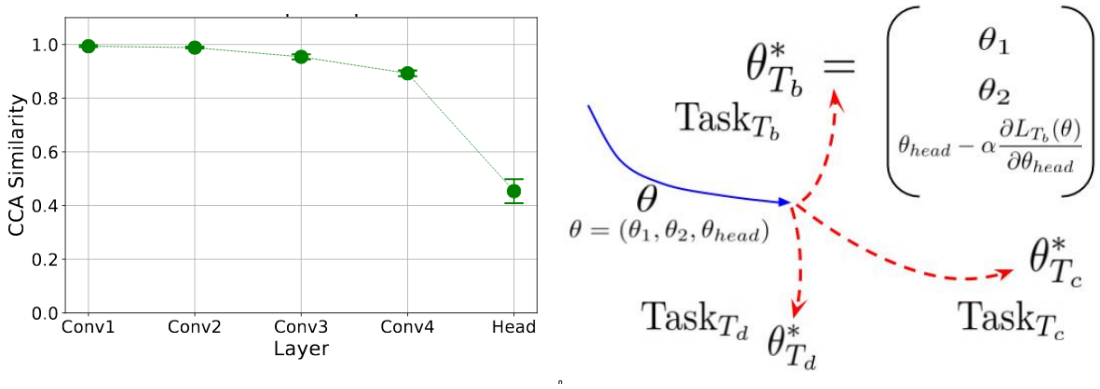
initialization.

MAML (Model-Agnostic Meta-Learning) introduced by Finn et al. [22] is one of the most famous algorithms that use this approach. Most works in the direction of optimization-based meta-learning are alterations [11; 51; 5; 71]. A major drawback of MAML is its computational costs. When the outer gradient is calculated, backpropagation through all the task adaptation steps is required [22]. For this, all intermediate updates of each gradient step have to be stored. Consequently, MAML requires increasingly more memory and computation the higher the number of task adaptation steps is.

Several solutions to overcome this issue have been suggested and are used in this section. One is to approximate the original meta gradient to the gradient of the last inner optimization update [22]. The sum of the last inner gradients of a batch of tasks is then used for the outer optimization update. This approach is called FOMAML (first-order MAML). FOMAML updates are an approximation of MAML updates, and empirical trials showed the impact on generalization performance as negligible [22]. As a consequence, FOMAML is computational more efficient at the cost of gradient accuracy. An abstract overview of the gradient update paths of MAML and FOMAML is depicted in figure 2.12.

Furthermore, it was discovered that the performance of MAML is a consequence of feature reuse in the convolutional layers [70]. This means, when adapting weights to a task, only the output layer of the neural network receives adaptation updates, while the other layers remain unchanged. This can be seen on the left side of figure 2.13, which shows the canonical correlation analysis (CCA) values of the layer weights before and after adapting to a task. The CCA value [29] of two matrices $A_k \in \mathcal{R}^{N \times p_k}$ $(k = 1, 2)$ is defined as

Figure 2.13.: MAML only adapts majorly the last layer of a deep neural network on a given task (left).
Consequently, ANIL was proposed (right), not updating the feature extractor during inner learning on each task (T_b, T_c, T_d) [70].



	MiniImagenet 1-shot 5-way		MiniImagenet 5-shot 5-way	
Method	Acc	Time	Acc	Time
MAML	46.9%	0.15s	63.1%	0.68s
ANIL	46.7%	0.084s	61.5%	0.37s

Table 2.1.: Comparing accuracy values and computation times of a single outer update between MAML and ANIL [70].

$$\begin{aligned} &\max_{u_1 \in \mathcal{R}^{p_1}, u_2 \in \mathcal{R}^{p_2}} corr(A_1 u_1, A_2 u_2) \\ =&\max_{u_1, u_2} \frac{u_1^T \Sigma_{A_1 A_2} u_2}{\sqrt{u_1^T \Sigma_{A_1 A_1} u_1} \sqrt{u_2^T \Sigma_{A_2 A_2} u_2}} \end{aligned} \tag{2.22}$$

with $\Sigma_{A_1 A_2}$ denoting the cross-covariance matrix of A_1 and A_2.

The authors proposed that the feature extractor learns sufficient feature representations during outer optimization [70]. An alteration of MAML called ANIL (Almost No Inner Loop) abuses this property. ANIL only allows the output layer of the neural network to be adjusted to a task. The feature extractor receives adjustments with the outer optimization update. This reduces memory and computation requirements. A sketch can be seen on the right side of figure 2.13.

Consequently, the lack of inner learning updates on most weights of the given neural network greatly improves computation and memory requirements. The generalization performance compared to MAML is similar, as shown in table 2.1.

Due to these findings, in the later experiment sections of this thesis, FOMAML is used

to approximate MAML. Furthermore, we use the principle of only adapting the output layer to a task for constructing baselines for data-free few-shot learning.

Metric-based meta-learning

Optimization-based meta-learning algorithms use gradient updates to adapt to each task. In the case of MAML, this requires additional computational time and introduces one more hyperparameter – the inner learning algorithm learning rate. Metric-based meta-learning does not require gradient updates and can be non-parametric for task adaptation [89; 79]. The task adaptation happens by evaluating a metric function on the support and query set feature embeddings. The feature embeddings are computed by forward propagating each set through a deep neural network feature extractor. The similarity between each query set embedding and the average class embeddings of the support set is computed. Then, the highest similarity of all classes indicates the predicted class. The output layer is not required, neither during meta-training nor during meta-testing. Consequently, in the context of metric-based meta-learning, we use the terms *feature extractor* and *neural network* interchangeably.

This category of meta-learning approaches can be interpreted as learning feature embeddings that are sufficiently general to make predictions based on metric differences between unseen classes.

A popular metric-based algorithm used in this thesis is Prototypical Networks (ProtoNet) [79]. ProtoNet calculates its output probability vector with the softmax operator applied on a negative distance metric $-d$. The distances are computed between the input and the class centeroids (called prototypes) $c_k \coloneqq \frac{1}{|\mathcal{D}^k_{sup}|} \sum_{(x,k)\in\mathcal{D}^k_{sup}} g_\theta(x)$. $\mathcal{D}^k_{sup} \subset \mathcal{D}_{sup}$ is each class-specific subset of the support set. k is the class index. The discriminative function is then given by equation 2.23 [79].

$$P_\theta(Y = k|X = x) \coloneqq \frac{\exp -d(g_\theta(x), c_k)}{\sum_{k'} \exp -d(g_\theta(x), c_{k'})} \tag{2.23}$$

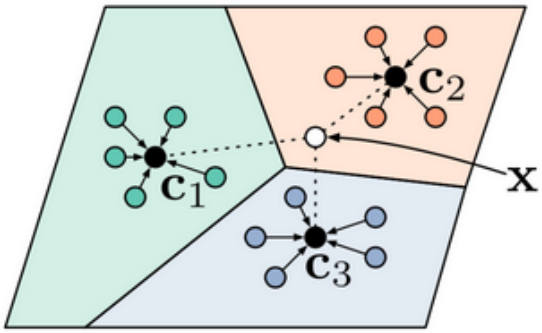
The outer optimization loss is the sum of the negative log-likelihood $-\log P_\theta(Y = k^*|X = x)$ of each query set instance x. Here, k^* is the true class label of an input x. A sketch of a feature embedding space with support set class centeroids and a query set instance can be seen in figure 2.14.

Baselines of meta-learning for few-shot learning

There are simple baselines that outperform some meta-learning algorithms, including MAML and ProtoNet [11]. Constructing baselines for few-shot learning is an active field of research [11] [12] [19]. Replacing meta-training with conventional neural network training gives a significant computation boost. As a consequence, we prefer baselines in cases where they show competitive generalization performance.

We define baselines for the problem addressed in this thesis based on Chen et al. [11]. We call the approach Baseline-Chen in the following to avoid confusion with other

Figure 2.14.: An abstraction of the embedding space of Prototypical Networks [79]. x is an instance of the query set. c_i are the prototypes of class i.



baseline methods. Baseline-Chen trains a deep neural network conventionally on all training classes combined in a single dataset. The output layer of this network is bigger compared to meta-learning approaches because all the classes are in a single task. For the k-shot n-way meta-validation and meta-testing phase, we remove the output layer and replace it with a randomly initialized output layer of output size n [11]. Furthermore, the feature extractor is fixed after training, so only the new output layer is updated based on the support set of validation or test tasks. The number of output layer updates by SGD for task adaptation is fixed, too.

For task adaptation, this makes Baseline-Chen equivalent to ANIL but with more gradient updates [11; 70]. We present an overview of the training phase, the task adaptation, and the output layer in figure 2.15.

Baseline-Chen showed comparable results to MAML and ProtoNet [11]. The generalization performance at domain drifts between training and testing classes is better [11].

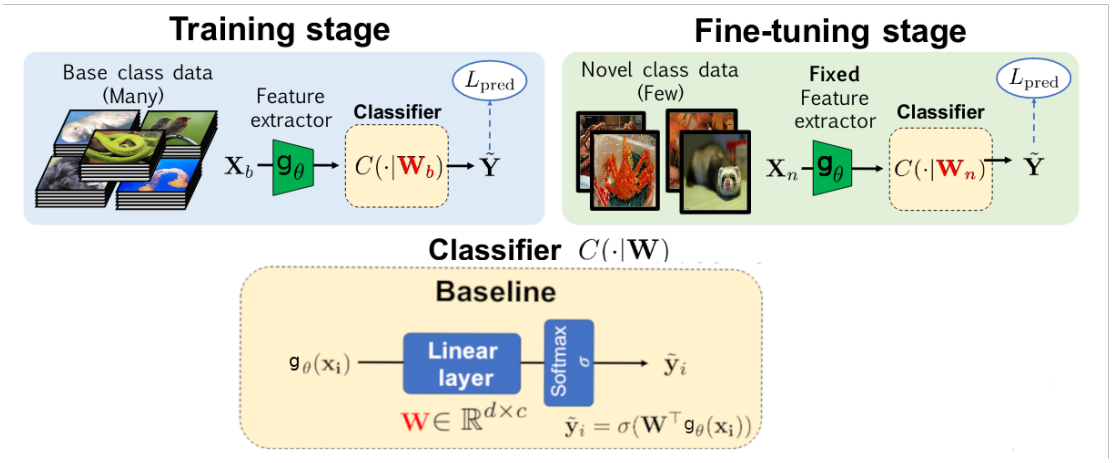
2.3.6. Data-free knowledge distillation

In modern deep learning applications, large neural networks are used [30; 18; 28]. Typical applications or large models are computer vision [30], natural language understanding [18], or speech recognition [28].

Consequently, knowledge distillation was introduced to transfer generalization performance from a large network to a smaller network [32]. In this context, the large network is called the teacher and the smaller network the student. As introduced in Hinton et al. [32], knowledge distillation trains the student network with the teachers' original training data. For this, the labels are discarded and replaced with the prediction probabilities of the teacher. This allows the new network to be deployed to smaller devices. The devices, e.g. cellphones, may not have enough computational capabilities to predict with the teacher network [27].

Due to data privacy concerns, the original training data may not be available. For

Figure 2.15.: The figure shows the different stages of Baseline-Chen and the output layer denoted as classifier C with parameters W [11]. The fine-tuning stage corresponds to the adaptation to a single validation or test task. The output layer has the parameters W_b during the training stage. After the training stage, the output layer is replaced with a new output layer. This new output layer is randomly initialized and according to the task's number of classes resized. W_n are the new task-specific parameters.



this scenario, methods performing knowledge distillation without the training set have been introduced [96] [60] [27]. Such methods perform data-free (or zero-shot) knowledge distillation (DFKD). To train a student, DFKD generate artificial data by using the teacher. We divide DFKD methods into two categories. We refer to one as generator-based since a generator network produces artificial training data [60; 10]. The student and generator are trained end-to-end in an adversarial manner [60; 10]. This can make the whole training difficult due to mode-collapse or catastrophic forgetting [84].

We call the other category noise-optimization-based. Here, random noise is initialized in the size of the desired training set. Then, the noise $\hat{x}$ is optimized by a gradient descent optimizer concerning an optimization objective in equation 2.24 [96].

$$\min_{\hat{x}} \mathcal{L}(p_T(\hat{x}), y) + \mathcal{R}(\hat{x}, p_T) \tag{2.24}$$

We denote the teacher model with p_T. $\mathcal{L}$ and $\mathcal{R}$ are specified so that the optimized noise resembles useful training data for the student [64; 96]. The noise-optimization-based approaches are simpler than generator-based approaches since a single optimization objective is used for each data instance [64; 96]. Generator-based approaches require an algorithm training two models, the generator and the student [60; 10]. A drawback is that the optimization has to start from scratch when new data is required. Out of all considered methods, the most promising for the proposed problem is introduced and explained in the following.

DeepInversion

In this section, we present a brief overview of DeepInversion (DI). DI optimizes noise with the objective given in equation 2.24 [96]. The primary loss $\mathcal{L}$ is the cross-entropy loss and responsible for the class assignment. The regularization term $\mathcal{R}$ optimizes the artificial data towards the domain of the original training data. This term of DI consists of several sub-terms, each serving a different purpose. These sub-terms are $\mathcal{R}_{TV}$, L_2, $\mathcal{R}_{feature}$, and $\mathcal{R}_{compete}$, and will be explained in the following.

The term $\mathcal{R}_{prior}$ was first introduced by Mordvintsev et al. [63], and is of the form

$$\mathcal{R}_{prior}(\hat{x}) = \alpha_{tv}\mathcal{R}_{TV}(\hat{x}) + \alpha_{l_2}L_2(\hat{x}) \tag{2.25}$$

with the L_2 norm applied on $\hat{x}$ flattened to a vector and the discrete total variation (TV) loss $\mathcal{R}_{TV}$ given in equation 2.26.

$$\begin{aligned}\mathcal{R}_{TV}(\hat{x}) \coloneqq \sum_{i,j} \Big(&(\hat{x}_{i+1,j} - \hat{x}_{i,j})^2 + (\hat{x}_{i,j+1} - \hat{x}_{i,j})^2 \\ &+(\hat{x}_{i+1,j+1} - \hat{x}_{i,j})^2 + (\hat{x}_{i+1,j} - \hat{x}_{i,j+1})^2 \Big)^{\frac{1}{2}}\end{aligned} \tag{2.26}$$

Equation 2.26 is a finite approximation of the *total variation* (TV) loss [55] on a continuous function $f : \mathcal{R}^{H\times W} \supset \Omega \to \mathcal{R}$ $(H, W \in \mathbb{N})$ given by the mapping

$$f \mapsto \int_\Omega \left(\Big(\frac{\partial f}{\partial u}(u,v)\Big)^2 + \Big(\frac{\partial f}{\partial v}(u,v)\Big)^2 \right)^{\frac{\beta}{2}} dudv. \tag{2.27}$$

It should be noted that the regularization term $\mathcal{R}_{TV}$ of DeepInversion uses additionally diagonal shifts unlike Mahendran and Vedaldi [55]. It is responsible for smoothing each image and minimize pixel artifacts. If the coefficients α_{tv} and α_{l_2} are high compared to the other values, the resulting images will look very blurry with a lack of sharp edges and corners [55].

DeepInversion introduces a further regularization term $\mathcal{R}_{feature}$, which helps to approximate the original data distribution. This requires accessing the BN statistics of the teacher network, learned on the original data. The difference between the running statistics μ_l and σ_l of a BN layer l of the teacher and the BN statistics $\hat{\mu}_l(\hat{\mathcal{B}})$ and $\hat{\sigma}_l(\hat{\mathcal{B}})$ of the current artificial data batch $\hat{\mathcal{B}}$ is minimized in equation 2.28 [96].

$$\mathcal{R}_{feature}(\hat{\mathcal{B}}) = \sum_l \|\hat{\mu}_l(\hat{\mathcal{B}}) - \mu_l\|_2 + \|\hat{\sigma}_l(\hat{\mathcal{B}}) - \sigma_l\|_2 \tag{2.28}$$

We assume that the running BN statistics accumulated during the teacher training are stored in a teacher. They give an approximation of the original distribution in the form of feature map distributions. As a consequence, this allows strong adaptation of

Teacher Network	VGG-11	VGG-11	ResNet-34
Student Network	VGG-11	ResNet-18	ResNet-18
Teacher Accuracy	92.34%	92.34%	95.42%
Noise ($\mathcal{L}$)	13.55%	13.45%	13.61%
$+\mathcal{R}_{prior}$ (DeepDream [63])	36.59%	39.67%	29.98%
$+\mathcal{R}_{feature}$ (DI)	84.16%	83.82%	91.43%
$+\mathcal{R}_{compete}$ (ADI)	90.78%	90.36%	93.26%

Table 2.2.: Accuracy values after training a student network with each additional loss term [96].

the otherwise noisy data to the training set domain. The DI regularization term $\mathcal{R}_{DI}$ is defined in equation 2.29 for a batch $\hat{\mathcal{B}}$ of artificial data [96].

$$\mathcal{R}_{DI}(\hat{\mathcal{B}}) = \frac{1}{|\hat{\mathcal{B}}|} \sum_{\hat{x} \in \hat{\mathcal{B}}} \mathcal{R}_{prior}(\hat{x}) + \alpha_f \mathcal{R}_{feature}(\hat{\mathcal{B}}) \tag{2.29}$$

Yin et al. [96] also introduce an variant of DI called Adaptive DeepInverson. This approach uses another regularization term $\mathcal{R}_{compete}$ given in equation 2.30. We refer to the student with p_S.

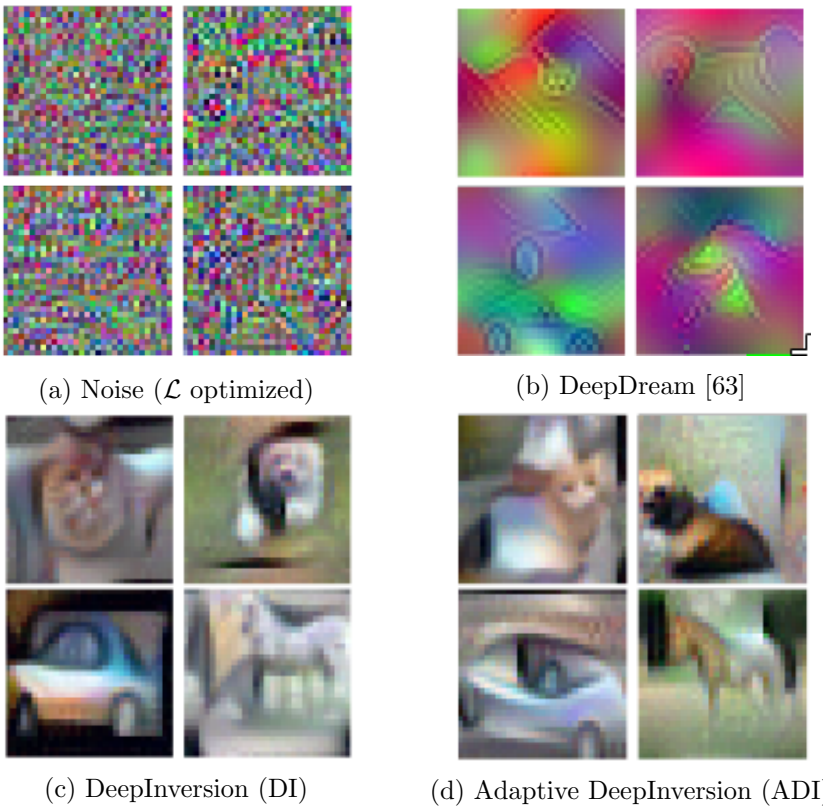
$$\mathcal{R}_{compete}(\hat{x}) = 1 - D_{JS}\big(p_T(\hat{x}) \| p_S(\hat{x})\big) \tag{2.30}$$

$\mathcal{R}_{compete}$ influences the noise optimization in parallel with the student training to optimize the data in directions where the student does not fit the teacher predictions. D_{JS} denotes the Jensen-Shannon divergence. Combining with the previous terms, this gives the total Adaptive DeepInversion regularization term in equation 2.31 [96].

$$\mathcal{R}_{ADI}(\hat{\mathcal{B}}) = \mathcal{R}_{DI}(\hat{\mathcal{B}}) + \alpha_c \frac{1}{|\hat{\mathcal{B}}|} \sum_{\hat{x} \in \hat{\mathcal{B}}} \mathcal{R}_{compete}(\hat{x}) \tag{2.31}$$

An overview of how each additional regularization term influences the generated images is given in figure 2.16. Furthermore, an overview of how the student accuracy is influenced is given in table 2.2. The differences between the performance yielded when using $\mathcal{R}_{DI}$ to $\mathcal{R}_{prior}$ are high. Using $\mathcal{R}_{ADI}$ gives further improvements. Using the $\mathcal{R}_{compete}$ introduces additional computations including the student model. Consequently, the optimization with this term is computationally slower.

An example of how effective DeepInversion can generate in-domain looking data is given in figure 2.17. By inspecting table 2.2, and comparing figure 2.16 with figure 2.17, we note that the results of DeepInversion may depend on the size of the teacher network and/or the resolution and/or size of the original training data.



(a) Noise ($\mathcal{L}$ optimized)

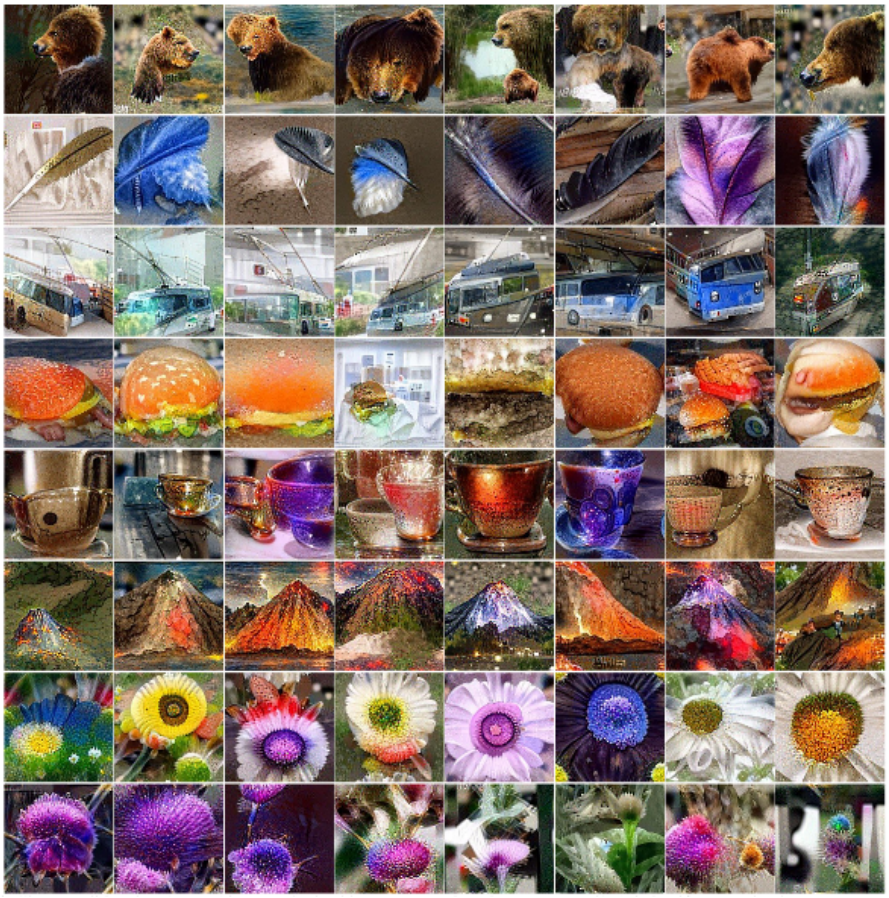
(b) DeepDream [63]

(c) DeepInversion (DI)

(d) Adaptive DeepInversion (ADI)

Figure 2.16.: Example images generated with each method applied to a ResNet-34 model trained on CIFAR-10 [96]. All images depict the same four classes: cat, dog, car, horse.

Figure 2.17.: Example images generated with DeepInversion applied to a ResNet-50v1.5 teacher trained on ImageNet [96]. Each row contains images of a single class.



3. Related Works

So far, data-free knowledge distillation [27] and few-shot learning via meta-learning [33] have been separately well studied. To the best of the author's knowledge, no published work has addressed their intersection, the data-free few-shot learning problem. We aim to distill the knowledge of multiple teachers in a way that enables few-shot learning in a student model. Works, that are built upon in this thesis, or that adress related problems, are presented in this section.

3.1. Meta-learning

Meta-learning algorithms are used to tackle the few-shot learning problem [90]. Since the problem addressed in this thesis is a variant of the few-shot learning problem, meta-learning is fundamentally related. A general survey of meta-learning applied to other settings is presented by [33]. In the scope of this thesis, we focus on meta-learning for few-shot learning. An overview of meta-learning algorithms is presented in the following.

3.1.1. Optimization-based approaches

One of the most influential methods for optimization-based few-shot learning is model-agnostic meta-learning (MAML) and its first-order approximation FOMAML introduced by Finn et al. [22]. MAML optimizes the weight initialization of a model, such that the model can successfully adapt to few-shot tasks with few gradient update steps. To update the weight initialization during meta-training, backpropagation through all task adaptation steps is done. This makes MAML memory and computationally expensive.

To address this issue, Nichol et al. [65] give an analysis on MAML and FOMAML. The authors propose to skip the backpropagation calculations, and introduce a simpler approach called Reptile. Reptile does not compute gradients for its outer optimization update. Instead, it moves the weight initialization in the direction of the task-specific weights.

Raghu et al. [70] show that MAML does not learn fast adaptation to new tasks. Instead, MAML learns feature reuse between tasks. Consequently, they proposed the ANIL (Almost No Inner Loop) algorithm. This algorithm adapts exclusively the output layer of a model to a task. The feature extractor is only updated by the outer optimization.

The recent work of Rajeswaran et al. [71] introduced implicit MAML (iMAML). iMAML enables a high number of task adaptation steps. This is achieved by using the implicit Jacobian of the task-specific parameters. It removes the requirement of doing backpropagation through all fine-tuning gradient steps as it exclusively depends on the final solution.

3.1.2. Metric-based approaches

Another category of meta-learning approaches uses a metric function. The metric function calculates distances between support set classes and query set instances to give a prediction. This does not require updating parameters of the model for task adaptation. One such method is called Matching Networks introduced by Vinyals et al. [89]. It compares the embeddings of the query set samples with the support set samples, and classifies them with their cosine similarity. The embeddings are given by the outputs of the last convolutional layer of a neural network. During outer optimization, the neural network weights are updated to decrease the angle between embeddings belonging to the same class.

Snell et al. [79] introduce Prototypical Networks (ProtoNet). ProtoNet compute the mean of all embeddings for each class in the support set. These mean embeddings are called prototypes. Distances between the prototypes and a query set instance are calculated with a metric function. The prediction is computed by applying softmax to the distances. The authors propose to use the Gaussian kernel as the metric function.

Sung et al. [83] replace the used metric with an additional neural network. This is similar to a parametrized metric, which allows better task adaptation. Thus, a better few-shot learning performance is achieved according to the authors' benchmarks. Rodríguez et al. [73] introduce Embedding Propagation for few-shot classification. It builds upon previous advances by regularizing the embedding space to have smoother decision boundaries. This is done by calculating a propagator matrix. The propagator matrix is multiplied with the embeddings to compute propagated embeddings. The propagated embeddings are used as inputs for the metric function.

3.1.3. Baselines for few-shot learning

There are other approaches for few-shot learning, besides meta-learning [90]. More simple baseline methods were introduced, too. A baseline may enable few-shot learning without a meta-training phase. Chen et al. [11] train a model conventionally on all training classes merged into a single dataset. The adaptation to an unseen task consists in optimizing a new output layer with a gradient-based optimization method. This is similar to how ANIL performs meta-testing [70] or to transfer learning [100]. Chen et al. [11] show competitive performance on few-shot tasks compared to meta-learning algorithms.

Chen et al. [12] introduce another baseline. Here, the model is trained conventionally, too. Afterwards, an additional meta-training stage happens. The meta-training stage resembles Prototypical Networks [79] with cosine similarity as the distance metric.

3.1.4. Meta-learning with artificial data

So far, all the mentioned methods work on thw original meta-training data. The problem setting considered in this thesis does not allow access to the original training data, so any meta-learning method is applied to artificial data. Using meta-learning approaches with artificial data is already done by the following works.

One approach introduced by Yin et al. [95] is ADML (ADversarial Meta-Learning). It aims to improve the meta-learning algorithm by using additional generated data. ADML uses the Fast Gradient Sign Method [26] to generate adversarial data with the given meta-training data. The data is used to calculate a sum of two gradients for the outer optimization update. One gradient is computed on the loss of adversarial data after fine-tuning on normal data. The other is computed on the loss of normal data after fine-tuning on adversarial data.

Another work published by Zhang et al. [99] introduces MetaGAN, which extends normal meta-training with an additional noise-class. For this, each n-way task is expanded with an additional class of samples produced by a generator. The task is considered to be a $(n + 1)$-way task. The generator is optimized to create fake images resembling the task classes. On the other hand, the classifier has to assign these fake images to the noise-class. This is done additionally to the original samples in the query set.

Both approaches share the property the artificial data to further enhance the meta-training on real data. In contrast, this work uses normal data and does meta-learning on artificially produced data.

3.2. Knowledge Distillation

In the context of this work, training data is inaccessible, and a set of trained models are available. As a consequence, methods of the topic data-free knowledge distillation have to be considered to successfully train a new model. An extensive survey of general knowledge distillation is presented by Gou et al. [27]. In original knowledge distillation, the goal is to compress a bigger model to a smaller one with minimal generalization performance reduction. Hinton et al. [32] is one of the first works introducing knowledge distillation. Here, a new student model is trained with the probability outputs of a single teacher model as labels. The original labels are discarded. The Kullback-Leibler divergence between the student's outputs and the new labels is used as the loss function. The training data of the teacher is used for training the student with the labels being changed to the teacher probability outputs.

The work of Zhang et al. [98] applies knowledge distillation to a teacher model trained with MAML. This gives a student capable of few-shot classification. The desired outcome of the presented thesis is similar to the mentioned work with a different setting. In Zhang et al. [98], the meta-training data of the teacher is available for the knowledge distillation procedure. Furthermore, the purpose of the introduced method is the compression of the teacher to a smaller model.

3.2.1. Data-free knowledge distillation

Further advances were made in the direction of data-free (zero-shot) knowledge distillation. Here, the teacher's training data is considered to be unavailable. Thus, methods focus

on generating artificial data. One approach for this is the usage of a generator network. The generator is tuned in parallel to the student network in an adversarial manner. Micaelli and Storkey [60] introduced Zero-shot Knowledge Transfer. It optimizes the student to minimize the Kullback-Leibler divergence between the student's and teacher's output. The data is produced by a generator, which maximizes the student's loss. To give the student more guidance an additional attention term is used. This loss reduces the difference between the student's and teacher's intermediate layer outputs. The authors show that their approach produces data, which gives a continuously high discrepancy between student and teacher throughout the training. This makes the student mimic the teacher without knowing the domain of the teacher's training data.

The same approach is used by Chen et al. [10] with a different loss. Here, for both the student and the generator the cross-entropy is used instead of the Kullback-Leibler divergence. Additionally, the generator receives two regularization terms. One is for producing data with high activation values in the teacher's last convolutional layer. The other term is for producing data with uniform class frequency.

The Kullback-Leibler divergence for the student is equal to the cross-entropy up to an additive constant, so the methods proposed in Micaelli and Storkey [60] and Chen et al. [10] are similar.

Another work by Choi et al. [13] builds upon this generator-based training and adds a generator pre-training phase. Here, before any update on the student is done, the generator is optimized with three different loss terms. One loss forces the generator to produce data close to the teacher training domain. This is done by minimizing the Kullback-Leibler divergence between the BN statistics of the original data stored in the teacher and the BN statistics received by forward propagating the generated data. The other two losses maximize the teacher confidence scores on the produced data and balance out the class frequencies.

A further approach for data-free knowledge distillation is by directly optimizing the artificial data. This usually involves starting with random noise and a target label. The noise is adjusted with the help of a gradient descent like method to resemble training data. Afterwards, the student is trained conventionally on the produced data. In the following we call such approaches noise-optimization-based. Compared to the generator-based approach, the amount of generated data is fixed with the initialization of the optimization. In contrast, a generator can theoretically produce an arbitrary amount of data after its training phase. On the other hand, optimizing noise is simpler and quicker than training two networks in an adversarial manner. The latter is prone to suffer from issues like mode-collapse or catastrophic forgetting [84].

One way to do noise-optimization-based data-free knowledge distillation was introduced by Nayak et al. [64]. The presented approach minimizes the cross-entropy between a class-specific Dirichlet-sample and the teacher output of the noise data. The Dirichlet-sample has concentration parameters according to the weights of each respective class of the teacher's output layer.

Yin et al. [96] introduce DeepInversion (DI) and its adversarial alteration Adaptive

DeepInversion (ADI). DI optimizes the noise data w.r.t. several criteria. Cross-entropy between teacher prediction and a class label is responsible for class affiliation of the noise data. BN statistics discrepancy similar to Choi et al. [13] is used to approximate the original training data distribution. Total variation and L_2 norm losses of the noise image are used to smooth out any pixel artifacts.

ADI uses the same losses as DI except for an additional adversarial loss. This loss combines the noise optimization and the student training to a single training procedure. The training happens in an adversarial manner, i.e. the student and the noise data are updated alternately. The adversarial loss is the Jensen-Shannon divergence between the student's and teacher's predictions on the noise data. Similar to Micaelli and Storkey [60] or Chen et al. [10], the optimization is drawn towards regions of higher discrepancy between the student and teacher outputs.

3.2.2. Multi-teacher knowledge distillation

Further works regarding the case of having multiple teacher networks have been published in the context of knowledge distillation. This scenario is often called knowledge amalgamation or ensemble distillation [27; 86].

The work of Ye et al. [93] uses unlabeled data and multiple teachers trained on different tasks to train a student. The student is capable of performing two pixel-prediction tasks: depth estimation and scene parsing.

Luo et al. [54] handle a more general case by training a student for an arbitrary task. Here, the requirement is the availability of unlabeled data, and the teacher networks can have differing architectures.

Another work of Tian et al. [86] tries to reformulate knowledge distillation by drawing connections to representation learning. The proposed method makes use of a critic model. The critic model forces the student to maximize the mutual information of its output with the teacher's output. The critic and the student are jointly optimized. Furthermore, the loss covers the case of having several teachers. The authors claim to have achieved state-of-the-art performance with this approach. Though, an extension for a data-free case has not been proposed.

One work, which does data-free knowledge distillation with multiple teachers, is of Ye et al. [94]. Here, every teacher layer receives its own generated input by a group-stack generator. The algorithm is designed to train each student layer individually with the teachers' intermediate layer outputs. The paper did not publish its source code, and the abstract algorithm description with a multitude of different losses makes reproducing the results unfeasible in the scope of this thesis.

3.3. Further privacy-preserving methods

The motivation of this thesis is to enable few-shot learning in a privacy-restricted setting without having access to the original training data of the available teacher models. A

category of algorithms for preserving the privacy of training data is federated learning. This was first introduced by McMahan et al. [58]. Federated learning aims to optimize a model with gradients received from several devices. Furthermore, the training data is exclusive to each device. Such a setting is given in the case of mobile phones, where the data is either inaccessible due to privacy reasons or simply due to size [40]. This is close to the proposed problem in this thesis. Instead of having gradients with no data, we consider the case where models trained on data are available. The work of Kairouz et al. [40] shows recent advances and problems in the field of federated learning.

4. Methodology

In this section, several methods for data-free few-shot classification given multiple pre-trained models are introduced. The pre-trained models are called teacher models or shortly teachers. At first, a detailed description of the novel problem setting addressed in this thesis is presented. Then, we define two orthogonal solution approaches and focus on one of them. We give an extensive specification of the proposed algorithm. Last, we introduce several baselines without (meta-) training stage. These act as challenging lower bounds for generative methods tackeling the defined problem.

4.1. Data-free few-shot learning

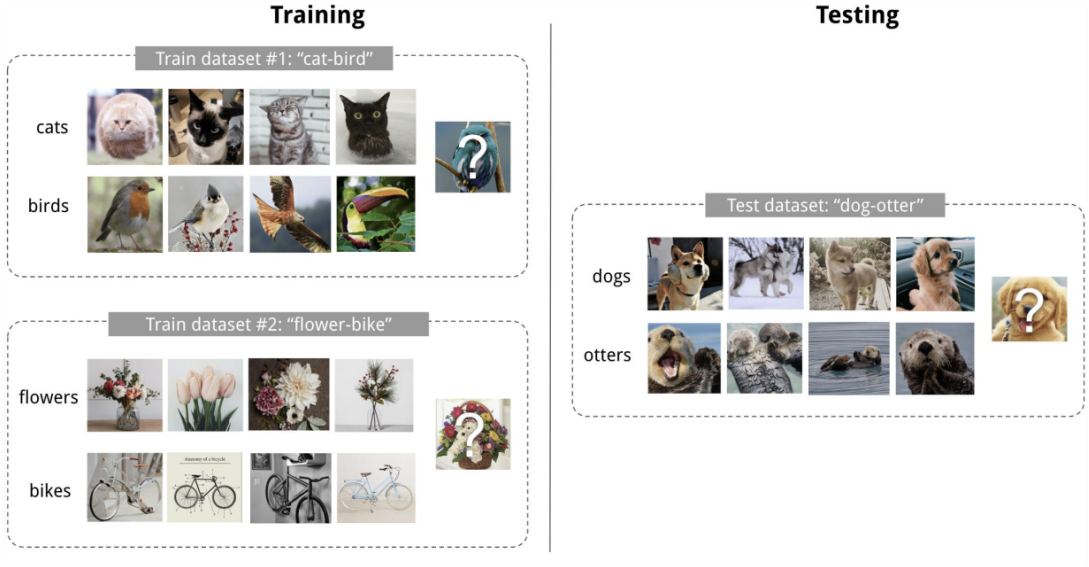
In the scope of this thesis, we introduce the novel problem setting of data-free few-shot learning. The problem is defined as follows. We are given a n-way k-shot classification task containing image data. This task is called the target task. We want to train a model on the task's support set such that the generalization error is as low as possible. For the generalization error, the error on the query set of the task is used. Furthermore, we assume a set $\mathcal{T} = \{T_1, \ldots, T_B\}$ of different pre-trained models is available. We call these models teachers, and neither their original training data is accessible nor any metadata. Each teacher is trained on data similar to the target task. Here, 'similar' means similarity according to the training tasks and test tasks defined by Deleu et al. [16]. The different training datasets are mutually exclusive, i.e., no two teachers are trained on the same class or data instance. This mimics the real-world scenario, where owners of different dataset are not willing to share data with each other, and a model is needed to learn a new similar task using only few datapoints.

A comparison between the original few-shot learning setting and the proposed data-free few-shot learning setting is depicted abstractly in figure 4.1.

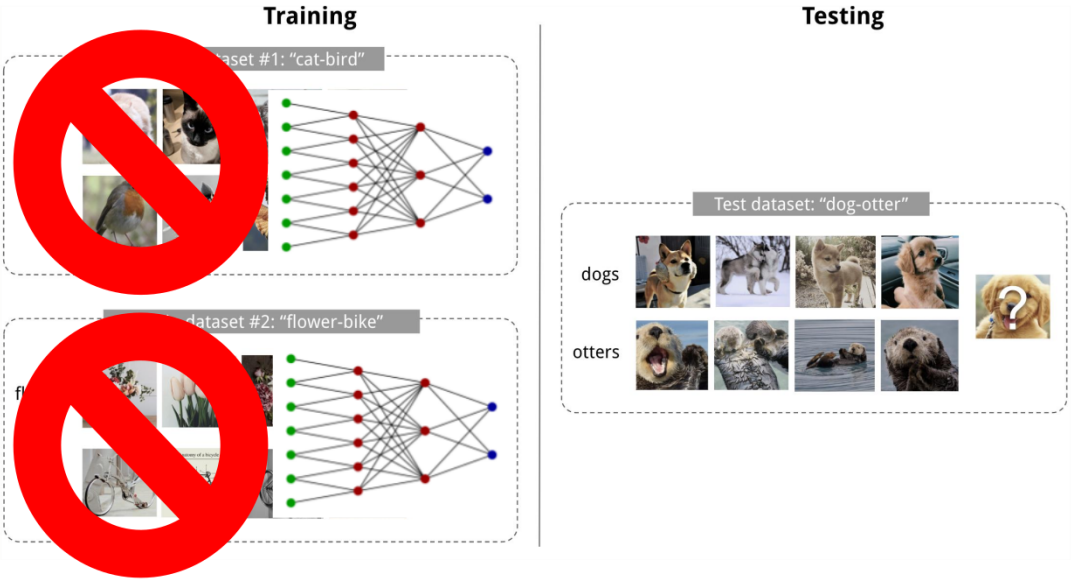
The teachers have arbitrary architectures. Thus, any approach trying to solve this scenario is supposed to work with arbitrary architectures. The proposed methods are designed to work according to these assumptions. Though, the outcome may be dependent on the given teachers. As a consequence, several different teacher architectures have to be evaluated.

4.2. Two classes of approaches

We introduce approaches combining meta-learning with knowledge distillation. We meta-train an inner learning algorithm with a student on generated data according to sections 2.3.4 and 2.3.6.



(a) Original few-shot learning provides training tasks similar to the target task.



(b) The novel data-free few-shot learning scenario provides teacher models trained on datasets similar to the target task.

Figure 4.1.: Comparison of the traditional few-shot learning and the proposed data-free few-shot learning setups. In both cases we want to adapt to an unseen task using few data instances.

To address our novel problem, we define two classes of approaches based on the data-free knowledge distillation literature. As mentioned in the chapter 2 and 3, there are currently two dominating classes of approaches in the latter. One is generator-based, where the student model and the generator are trained adversarially. The other is noise-optimization-based, where, noise data is optimized to resemble the original data and then used to train the student model. In the following, we discuss the proposed classes of approaches.

4.2.1. End-to-End data-free meta-learning

Similar to generator-based data-free knowledge distillation, we propose an approach that generates data optimized for meta-training the student. This can be achieved by optimizing the data generation and meta-training a few-shot learner End-to-End. In the following, a brief outline of this class of approaches is presented. In each optimization step, the negative value of the (meta-) training loss of the previous step is part of the data generation loss. Thus, both generating the data and training the student are part of the same optimization process. The generated data is optimized for the meta-training. On one hand, this may give better results than just trying to replicate the original training domain of the teacher. On the other hand, combining several optimization procedures by iterative updates and a shared optimization objective can make the training process non-transparent. Consequently, it can be expected the algorithm is prone to issues similar to those of generative adversarial network training [84].

We focus on the following approaches as a first attempt in tackling the proposed problem.

4.2.2. Consecutive data-free distillation and meta-learning

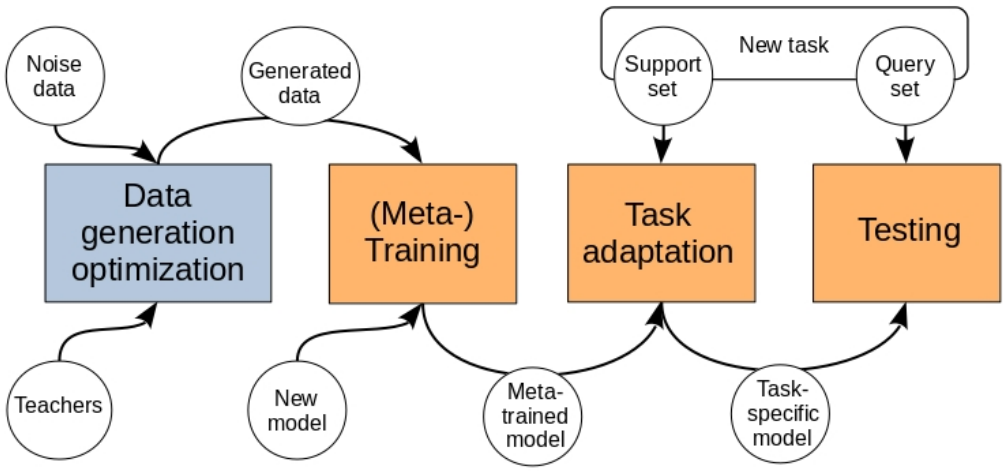
In this approach, we separate the generation of the artificial data and the training of the student. The first stage focuses on generating data resembling the original training data of the teachers. This is done prior to the meta-training. After generating a full set of artificially data, the second stage applies (meta-) training on the generated data. This allows any meta-learning method to be used.

This approach has the advantage of optimizing each part separately. Furthermore, different methods for each stage can be easily replaced and compared. This allows using state-of-the-art algorithms for both the data generation and the meta-training part. Figure 4.2 shows an abstract depiction of the algorithm for a test task.

The validation loss of each stage can be evaluated to perform hyperparameter tuning and algorithm selection.

Selecting the best method for the data generation is computational difficult compared to the meta-training stage. We cannot evaluate the data generation stage according to the mean validation performance of the considered meta-learning algorithms since this would be computationally infeasible. So, we will only evaluate the data generation stage with a computationally cheap algorithm trained on the generated data. We use Baseline-Chen

Figure 4.2.: An abstraction of the proposed data-free meta-learning algorithm. In the first stage (blue box), data is generated with the given teachers and an arbitrary data-free knowledge distillation algorithm. The second stage is conventional meta-learning (orange boxes). The generated data is used within a meta-learning algorithm to produce a meta-trained student. Then, the student is adapted to the support set of the target task. Last, the task adapted student is used to evaluate the generalization error of the target task query set.



as this cheap algorithm. The evaluation value is the generalization error of Baseline-Chen adapted to a new task. For validation, the generation process is optimized w.r.t. the validation loss of Baseline-Chen. The prior-generation-based approach with validation steps is summarized in algorithm 3. The trained student f_θ can be used on unseen test tasks. Or, a new (and possibly more computationally expensive) meta-learning algorithm can run on the generated set $\mathcal{X}$.

Algorithm 3: Consecutive data-free distillation and meta-learning algorithm for data-free few-shot learning.

Input : Set of teachers $\mathcal{T} = \{T_1, \ldots, T_B\}$, data generation algorithm DI, (meta-) training algorithm M, validation tasks $\mathcal{D}_{val}$
Output : Student/inner learning algorithm f_θ and artificial data $\mathcal{X}$

```
# initialize data for each teacher
for b = 1, ..., B do
    initialize data X_b
end
while validation loss L_val improves do
    # STAGE 1
    for b = 1, ..., B do
        # update data for each teacher independently
        X_b ← DI(X_b, T_b)
    end
    X ← {X_1, ... X_B}
    # STAGE 2
    initialize new student f_θ
    while validation loss L_val improves do
        # run M on X to update student
        f_θ ← M(f_θ, X)
        # calculate validation loss of student
        L_val ← f_θ(D_val)
    end
end
return (f_θ, X) according to the best L_val
```

Data generation stage

Like mentioned in section 2, optimizing noise is simpler than training a generator network. Since the proposed problem is novel, we prefer simplicity in the used method. Furthermore, compared to the other noise-optimization-based approach of Nayak et al. [64] DI introduced by Yin et al. [96] shows the most promising in-domain looking generated data. Consequently, we decided to use DI for the data generation stage [96]. For this, we require the teachers to have BN layers and stored running statistics of their

original training set. This assumption holds in all encountered modern convolutional neural network architectures [30; 35].

DI is applied to each teacher independently to generate a fixed number of images. Due to memory constraints, for each teacher, several batches of noise-data are optimized separately. After every batch of every teacher is optimized for a certain number of iterations, all the batches are merged into a single dataset. The amount of optimization iterations is determined by early stopping on the meta-validation loss of Baseline-Chen. For this, a randomly initialized model is trained on all the batches combined after every $i \in \mathbb{N}$ DI iterations. Since we assume the data noisy, we do not know how long Baseline-Chen should run without overfitting to the strongly present noise. As a consequence, another level of early stopping on top of Baseline-Chen early stopping is required. Baseline-Chen is trained as long as the meta-validation loss improves by at least one percent across multiple training iterations. The best validation loss of Baseline-Chen is used as the validation loss of the current DI iteration. If this validation loss does not improve more than one percent for several iterations, DI optimization is stopped. In our implementation, we use 100 DI iterations between validation evaluations. A sketch of this tuning procedure can be seen in figure 4.3. After the best hyperparameter configuration is determined, a new set is generated. We optimize it by the through early stopping determined amount of DI iterations.

Due to DI, we require that any teacher model was trained with several BN operations in its architecture. Additionally, the running batch statistics have to be stored in the teacher.

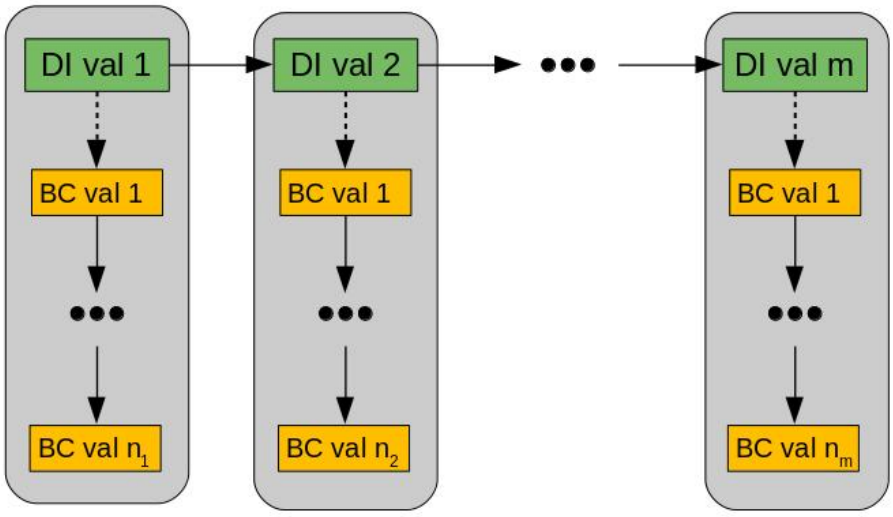
Meta-learning stage with generated data

The proposed algorithm 3 returns a model capable of learning a target few-shot task. To achieve this final result, a meta-learning training stage is required. After hyperparameter tuning the data generation stage, additional data is generated in higher volumes. This is necessary since we produced fewer data instances during the hyperparameter tuning. We expect better results with more generated data. We execute a meta-learning algorithm or one of the few-shot learning baseline methods on the generated data. The algorithm uses the generated data from the previous stage for its training.

To cover a variety of few-shot learning approaches, we decided to try an algorithm for optimization-based meta-learning, metric-based meta-learning, and a baseline using meta-training data, we present in the following.

We choose MAML [22] as the optimization-based meta-learning algorithm. Multiple optimization-based meta-learning methods are modifications of it [70; 65; 71]. We use its first-order approximation FOMAML due to its reportedly similar performance, yet quicker run time and smaller memory consumption. During meta-training, five inner learning update steps are executed for adaptation. For validation and test tasks, we use ten inner learning steps.

Figure 4.3.: An abstraction of the DI early stopping procedure during its hyperparameter tuning. After a predetermined amount of optimization iterations on each batch, a validation episode is triggered (green boxes). In each validation episode, all the batches are merged and used for training a new Baseline-Chen model (BC) (dotted arrows). Baseline-Chen is trained with early stopping resulting in $n_i \in \mathbb{N}$ ($i = 1 \dots m$, $m \in \mathbb{N}$) amounts of validation evaluations (orange boxes). The best validation value of a full Baseline-Chen run is used for the corresponding DI validation step. Of the final $\sum_{i=1}^{m} n_i$ amount of Baseline-Chen validation values, the best is picked as the validation loss for the used DI hyperparameter configuration. This nested early stopping is repeated for each configuration during hyperparameter tuning.



We use ProtoNet [79] as a representation of metric-based meta-learners. We consider it as a common method with a competitive performance compared to other approaches [73; 11; 6]. Since the metric function is non-parametric, task adaptation is computationally more efficient.

As the few-shot learning baseline approach, we use Baseline-Chen [11]. We consider it as a simple approach without a costly bi-level optimization scheme. This strongly improves run time and, as reported [11], maintains competitive generalization performance. It shows the quickest training improvements w.r.t. computation time of all evaluated algorithms for few-shot learning.

4.3. Few-shot learning without generated data

To assess the usefulness of the training on the artificial data, several baselines are constructed. These are designed to not use training tasks. Because we are given a few-shot task, we can optimize the model weights. This is done exclusively on the given validation or test task's support set.

Updates of one task are not allowed to leak to another task. So, for every encountered task, the baseline model has to be re-initialized fully from scratch. Furthermore, the baselines are designed to be always usable in any setup of the given scenario. In the following sections, three baselines are introduced. Since they work without any (meta-) training data, they can be considered a minimum lower bound for any data-free few-shot learning method.

4.3.1. Random initialization

The most naive approach is to train a randomly initialized model on the support set of the few-shot target task. The number of training iterations is determined using the validation tasks. More specifically, the number of updates on a validation task is specified with early stopping by calculating the generalization loss on the task's corresponding query set.

We expect that further updates give diminishing or harmful results. The optimal number of iterations averaged across meta-validation tasks is used for test tasks.

This baseline can be used for any few-shot learning setting, if no teacher models are given. It can be interpreted as a lower bound. If any algorithm is worse than this, the (meta-) training can be seen as harmful and should not be used. We refer to it as the Random-Initialization baseline in the following.

4.3.2. Best teacher initialization

As we mentioned in section 14, normally trained models are already decent at learning few-shot tasks. The teacher was trained on a task similar to the target task. So, we expect positive transfer when fine-tuning the teacher with a new, randomly initialized output layer. If the student cannot outperform every teacher in a few-shot task, training a new

student cannot be considered as beneficial. We aim for a learner with the best possible generalization performance. For this baseline, every teacher is used as initialization for fine-tuning on each validation task. We initialize a new output layer according to the target task. The teacher with the best validation loss is used for the test tasks. We refer to this baseline as Best-Teacher. The amount of update iterations on the support set is tuned as done in the random initialization baseline case.

4.3.3. Teachers' features concatenation

In this section, we propose another baseline combining the feature extractors of multiple teachers.

The Best-Teacher baseline uses each teacher separately. Consequently, the baseline fails to leverage the knowledge of several teachers. This may be neglectable if there are few teacher models trained on high data amounts. However, for scenarios with lots of smaller teachers, such a baseline may be inferior compared to combining the teachers. Considering optimization-based meta-learning in section 14, we make the assumption that the teachers' feature extractors do not require major weight updates. As a consequence, we introduce a baseline called Teacher-Concatenation. It concatenates the features of all teachers. The feature concatenation is the input for a randomly initialized output layer. The output layer is the only part of the model to be adapted to a task. Similar to the task adaptation phase of ANIL [70], the teachers' feature extractors remain frozen during fine-tuning. This makes it possible to concatenate many bigger teacher architectures since the output layer alone is relatively computationally cheap to optimize. The amount of update iterations on the support sets of the validation and test tasks is equally specified as in the other baselines, Best-Teacher and Random-Initialization.

This baseline allows using the feature extracting properties of all teachers combined during task adaptation. Meta-training is not required to achieve strong few-shot learning performance, as it was shown by Chen et al. [11]. Consequently, Teacher-Concatenation may be the strongest baseline for our problem setting. It may outperform the proposed data-free meta-learning method by a wide margin.

5. Experiments

In this chapter, a variety of experiments that have been conducted are explained in detail. At first, we formulate research questions relevant for the addressed problem setting. We designed our experiments to answer these as concisely as possible. Afterward, multiple public datasets for few-shot learning are introduced, which are part of the experimental evaluation and cover a range of possible real-world scenarios. This is followed by a description of the teacher training. In the end, the full experiment procedure is described in detail, including every hyperparameter optimization procedure with its evaluated configurations.

5.1. Research questions

The introduced methodology motivates a variety of different research questions. We want to determine which method performs best under a quick hyperparameter search and how the performance depends on the initial setting, namely the teacher models. Furthermore, we ask how much the generated data improves the final generalization performance and how much accuracy is lost by training any method on the generated data instead of the original data. Thus, an extensive description and the corresponding experimental setup are introduced and explained in the following.

5.1.1. Optimization-based versus metric-based meta-learning

As explained in chapter 2, optimization-based and metric-based meta-learning algorithms are often similar in their outer optimization update. On the other hand, fine-tuning on the support set of a task happens fundamentally different. This raises the question if there is a performance difference between meta-learning approaches. Additionally, Chen et al. [11] shows results suggesting a discrepancy between the respective performance with out-of-domain training tasks. Since we assume the generated data is not in the same domain as the validation and test tasks, we compare each meta-learning approach in the given setting. As a contender for optimization-based meta-learning, FOMAML is used. FOMAML is computationally more efficient than MAML yet yields comparable performance [22]. To achieve a fair comparison in the face of hyperparameter tuning, all few-shot learning algorithms are allowed to perform the same amount of hyperparameter runs. However, while the other methods' most sensitive one is the learning rate, FOMAML has two learning rates, of which the task adaptation learning rate has shown to be the more sensitive one. Due to this reason, FOMAML's meta-learning rate is fixed to the default value $1e-3$ [22; 6; 65]. This default appeared to work on all previously encountered datasets [22; 6; 65]. So, we optimize the inner learning rate for the task adaptation

during meta-validation. Additionally, five task adaptation steps are used during meta-training and ten during meta-validation and meta-testing. For the metric-based approach, ProtoNet was the selected algorithm.

5.1.2. Meta-learning versus few-shot learning baselines

In the work of [11], Baseline-Chen shows strong few-shot learning performance compared to meta-learning algorithms. This is remarkable since it trains a feature extractor conventionally on the training data. The training data consists of the meta-training tasks. So, a further research question is whether there is a benefit in using meta-learning algorithms on our generated data. Thus, Baseline-Chen is included in our experiments as a few-shot learning algorithm with the same hyperparameter optimization budget as FOMAML and ProtoNet.

5.1.3. Few-shot learning with generated data versus without generated data

We construct further experiments, which compare the few-shot learning baselines without generated data with the already mentioned algorithms that rely on the usage of generated data. The considered baselines are Random-Initialization, Best-Teacher, and Teacher-Concatenation. We give these baselines the same hyperparameter tuning budget under identical settings. The number of task adaptation steps is not predefined, in contrast to FOMAML and Baseline-Chen. This value is tuned on the validation tasks via early stopping. For fine-tuning on the test tasks, the mean early stopping iteration of the validation tasks is used. Because the learning rate has a substantial influence on the stopping iteration, the resulting number of adaptation steps may be more than thousand. This results in an advantage versus FOMAML using five or ten and Baseline-Chen using 100 task adaptation steps. Furthermore, this increases the tuning time drastically since these thousands of adaptation steps have to be done on each validation task. Due to this, we set an upper limit of 10000 adaptation steps.

5.1.4. Few-shot learning with generated data versus with original data

We investigate the performance decrease caused by using generated data instead of the original data. We train Baseline-Chen, FOMAML, and ProtoNet on the generated data and on the original training data of the teachers.

Baseline-Chen shows that a conventional training regime is sufficient to return a well-performing few-shot learner [11]. Thus, as a consequence of the shallow hyperparameter tuning and aggressive early stopping, teacher-based baselines (Best-teacher and teacher-concatenation) may perform better than algorithms tuned on the original data.

5.1.5. Teacher architectures and their training data amount

The problem adressed in this thesis depends on the dataset and the given teacher models. We investigate the impact of the model architecture on the quality of the generated data. In our case, there are two main differences between pre-trained teacher models: the type

of architecture and the amount of training data. We want to show how the proposed methodology works with a different given architecture and if more training data increases the quality of the generated images. For that, each experiment is repeated with Conv-4 and ResNet-10 as the teacher model architectures. Furthermore, we construct successive runs for teachers trained on 2000 training samples and 8000 training samples. We aim to use all the available data of a dataset. The datasets have 64 classes with 600 images per class [82; 89; 46]. If we use fewer training samples per teacher, we will be able to train more teachers. Thus, by using 2000 training samples, 16 different teachers can be trained on available data. 8000 training samples allow four teachers.

5.1.6. Few-shot versus many-shot learning

Originally, few-shot learning is the main focus of this work. Furthermore, we want to investigate if the number of shots of the test task influences the results. If so, specific methods are more recommendable for a given number of shots. To examine this, meta-training and meta-testing are repeated on tasks with shots in $\{1, 5, 50\}$. So, each (5-way) task has a total of 5, 25, and 250 image instances, respectively.

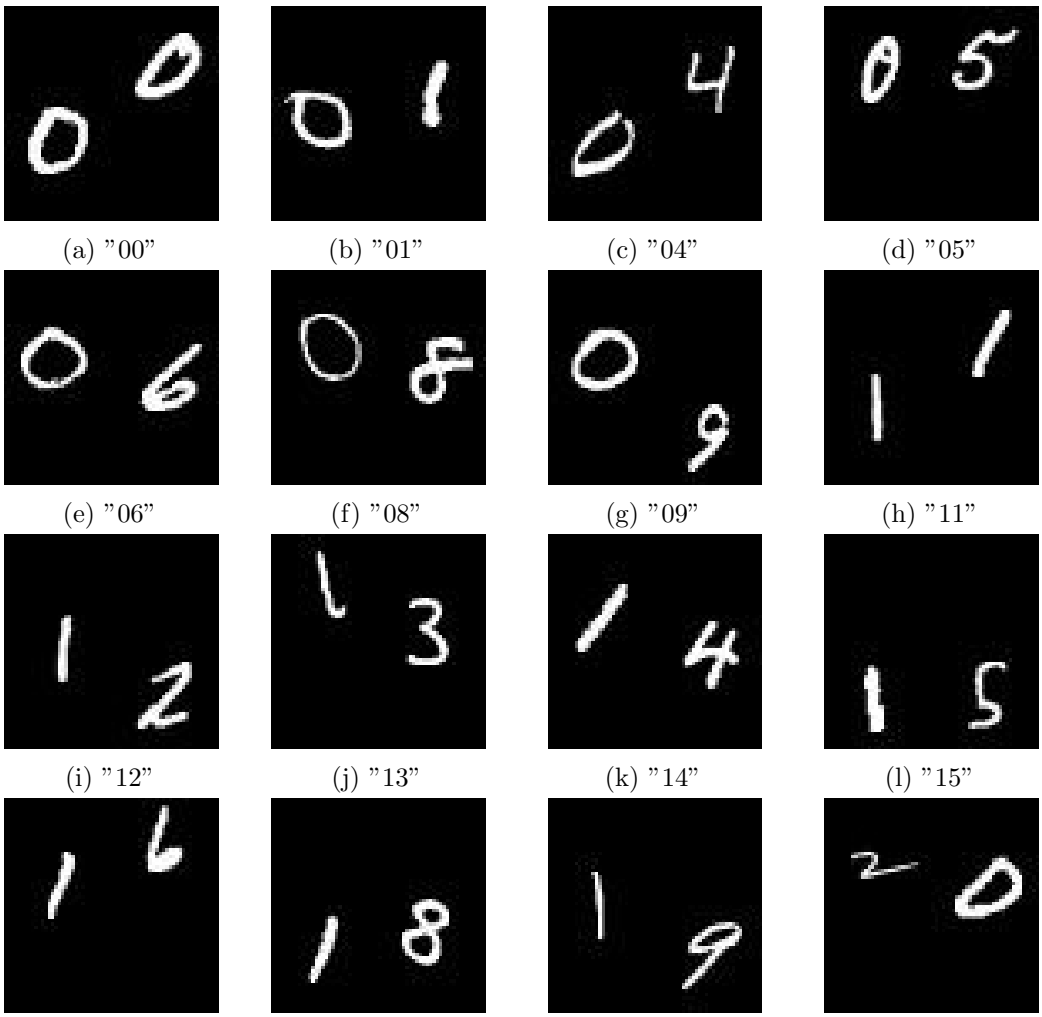
5.2. Datasets

In our proposed scenario, the teacher networks are conventionally trained convolutional neural networks with BN layers and no further specifications. This is a requirement of DI [96] and explained in chapter 2. To evaluate the proposed methods, we have to train the teachers additionally.

The library Torchmeta by Deleu et al. [16] offers a variety of few-shot learning datasets of image data. Here, some datasets have classes with several hundred instances [82; 89; 46]. We assume this are enough instances to train a convolutional neural network as long as enough classes are used for a task. The training data for each teacher is smaller in size and, as a consequence, only smaller teacher architectures are viable [31]. Nevertheless, they should be enough for a first assessment of the novel problem handled in this thesis. Constructing larger sized datasets allowing bigger teacher architectures is left for future research on this topic. An overview of the picked few-shot learning datasets is presented in the following.

5.2.1. DoubleMNIST

DoubleMNIST is an alteration of the original MNIST dataset [82]. Instead of single digits (0-9), it consists of 100 classes of double digits (00-99) with 64 classes used for training tasks, 16 for validation tasks, and 20 for testing tasks. 1000 images represent each class, and the images are grayscale with a resolution of 64×64. Example classes are depicted in figure 5.1. Because the classes are split arbitrarily for training, validation, and testing, every single digit (0-9) is seen in training tasks. Consequently, generalization from training tasks to validation and test tasks is easier compared to the other datasets, making DoubleMNIST the easiest dataset used in the experiments.



(a) "00" (b) "01" (c) "04" (d) "05"

(e) "06" (f) "08" (g) "09" (h) "11"

(i) "12" (j) "13" (k) "14" (l) "15"

(m) "16" (n) "18" (o) "19" (p) "20"

Figure 5.1.: Images of example classes out of 100 classes of DoubleMNIST [82].

5.2.2. CIFAR-FS

CIFAR-FS is a restructured copy of the CIFAR100 dataset with a total of 100 classes and 600 RGB images per class [46]. Each image has a resolution of 32×32. The classes are split into 64, 16, and 20 for training, validation, and testing, respectively. CIFAR-FS is of medium difficulty due to the highly different classes, and we assume the used models have significantly lower accuracy than in the case of DoubleMNIST. Example classes are depicted in figure 5.2.

5.2.3. MiniImagenet

MiniImagenet is a dataset based on ILSVRC-12 (ImageNet challenge of 2012) [89]. Instead of using the full set, 100 classes are sampled with each image colored and a resolution of 84×84. Like the other datasets, these classes are split into 64 for training tasks, 16 for validation tasks, and 20 for testing tasks. Example images out of 16 distinct classes are presented in figure 5.3. MiniImagenet is supposed to contain some of the complexity and difficulty of the original ImageNet challenge while being manageable by non-high-end GPU devices [89]. Thus, we intend it as the hardest challenge in our experiments, resulting in mean model accuracy lower than CIFAR-FS and DoubleMNIST.

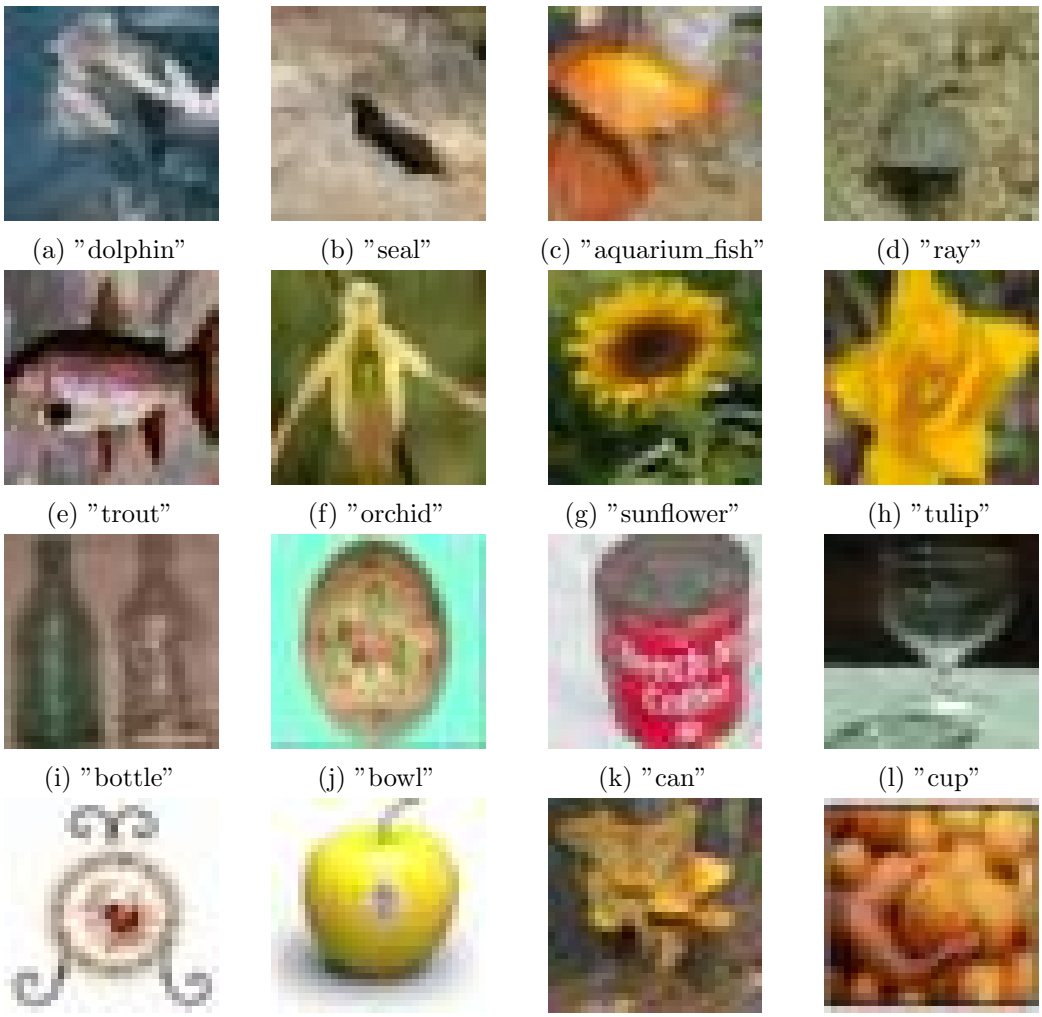
5.3. Teacher training

We train the teachers using only the meta-training data of the respective datasets. We restrict the teachers to not share data instances or classes during their training. For this, the classes of the dataset are evenly split by the amount of desired teachers so that each teacher has its own exclusive training classes. Then, each teacher is trained and its hyperparameters tuned on the images of the assigned classes. To do hyperparameter tuning and get a realistic and reliable generalization accuracy, each teacher's available set of images is split into a training, validation, and test sets. For the hyperparameter optimization, the learning rate is optimized. We consider the values $1e-3$, $1e-4$, and $1e-5$. The final teachers are used for the data-free few-shot learning problem.

5.4. Experiment procedure

Experiments on three datasets $\{\text{DoubleMNIST}, \text{CIFAR-FS}, \text{MiniImagenet}\}$ with each having four distinct teacher setups ($\{\text{Conv-4}, \text{ResNet-10}\}$ trained on $\{2000, 8000\}$ images) give twelve independent trial runs.

Conv-4 is a 4-layer CNN architecture, which consists of approximately 120.000 parameters (slightly fluctuating depending on the dataset) distributed across four convolutional layers with 64 feature maps and an output layer. This architecture is used by the teachers and the student independently of its training algorithm, i.e., FOMAML, ProtoNet, Baseline-Chen, as well as for the Random-Initialization baseline. ProtoNet does not use the output layer. Each convolutional layer consists of a convolution, a BN, a ReLU, and a Maxpool operation executed in this order.



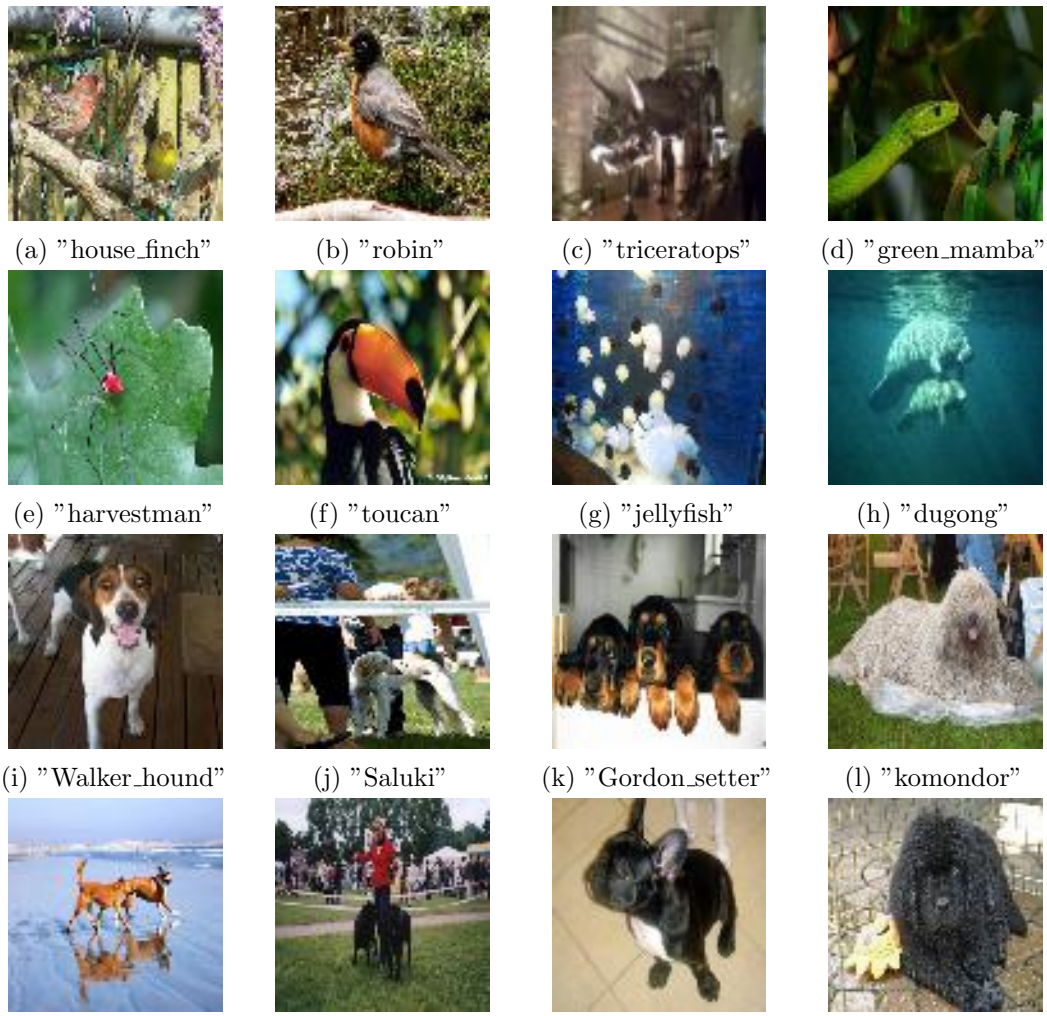
(a) "dolphin" (b) "seal" (c) "aquarium_fish" (d) "ray"

(e) "trout" (f) "orchid" (g) "sunflower" (h) "tulip"

(i) "bottle" (j) "bowl" (k) "can" (l) "cup"

(m) "plate" (n) "apple" (o) "mushroom" (p) "orange"

Figure 5.2.: Images of example classes out of 100 classes of CIFAR-FS.



(a) "house_finch" (b) "robin" (c) "triceratops" (d) "green_mamba"

(e) "harvestman" (f) "toucan" (g) "jellyfish" (h) "dugong"

(i) "Walker_hound" (j) "Saluki" (k) "Gordon_setter" (l) "komondor"

(m) "boxer" (n) "Tibetan_mastiff" (o) "French_bulldog" (p) "Newfoundland"

Figure 5.3.: Images of example classes out of 100 classes of MiniImagenet.

The ResNet-10 architecture consists of approximately 4.9 million parameters, nine convolutional layers and an output layer. Since ProtoNet does not use this architecture and to not vary from the ResNet naming conventions, the output layer is involved in the layer count. The input layer is a convolutional layer with BN and max-pooling. Four residual blocks follow with pairs of convolutional layers as described in section 2.3.2.

The hyperparameters of each teacher are tuned with grid search. $\{1e-3, 1e-4, 1e-5\}$ is the set of considered learning rates in each trial. The winning teachers of a trial are used as the given teachers for the following evaluations.

Hyperparameter tuning of each method is executed with early stopping on the validation loss with stopping patience of 3. I.e., if three consecutive evaluations of the validation loss show no improvements larger than 1%, the respective hyperparameter configuration run is stopped. The validation loss of Baseline-Chen is used for hyperparameter tuning DI with early stopping. We use grid search with the configurations $\{1e-1, 1e-2, 1e-3\}$, $\{1e5, 1e6, 1e7\}$, and $\{1e3, 1e4, 1e5\}$, for the learning rate of the optimizer, the TV loss term scaling, and the BN loss term scaling, respectively. For MiniImagenet, we use the configurations $\{1e6, 1e7, 1e8\}$ for the TV loss term scaling since manual tries showed slightly more pixel artifacts. The defined hyperparameter values of DI are several magnitudes greater than in the original DI implementation because we normalize each term to scale with the image, batch, and teacher network size [96]. These normalizations are not present in the original implementation [96]. There, a change of the BN layer amount, the image size, or the batch size requires a change of the loss term hyperparameters [96]. The L2 loss term is not used since it showed no influence. In the original work, the term was not used or of neglectable size [96]. The Baseline-Chen learning rate during the DI hyperparameter tuning is fixed to $1e-3$, a default used by the original work [11]. Since the defined DI tuning setup requires many evaluations, it was decided to use ASHA (asynchronous successive halving algorithm) introduced by Li et al. [49]). ASHA is used for DI hyperparameter tuning to terminate non-competitive configurations early.

Furthermore, during the hyperparameter tuning, we generate approximately 40 percent of the original training data's size due to computational limitations. After a winning configuration is found, a bigger set of approximately 160-200 percent (160 for MiniImagenet, 180 for DoubleMNIST, 200 for CIFAR-FS, due to computational reasons) of the original training data size is generated and used for the following.

After the data is generated, Baseline-Chen, ProtoNet, and FOMAML are hyperparameter tuned with validation tasks. Baseline-Chen and ProtoNet use the learning rate configurations $\{1e-2, 1e-3, 1e-4\}$. For FOMAML this is fixed to $1e-3$. We consider the task adaptation learning rate from $\{1e-1, 1e-2, 1e-3\}$ for FOMAML. We use Adam as optimizer for most algorithms. An exception is the task adaptation update of FOMAML, for which we use SGD.

The baselines without generated data receive hyperparameter tuning to the same extent for a fair comparison. Their learning rates are similarly considered from $\{1e-2, 1e-3, 1e-4\}$ as those of Baseline-Chen and ProtoNet. However, they are given the optimizer SGD instead of Adam since we make task adaptation

similar to MAML-like algorithms. Because possible thousands of update steps are done on every task, the maximum amount of fine-tuning steps is set to 10000. The Teacher-Concatenation baseline uses a maximum of eight teachers due to memory reasons.

All the hyperparameter ranges are picked considering the best validation results of manual trials or on defaults used by other works [96; 11; 79; 22]. The validation tasks during the hyperparameter tuning are always 5-shot 5-ways since, for different task setups, similar results can be expected. The increased computational effort of tuning different shot settings would be enormous.

After the hyperparameter optimization of all algorithms is done, the testing stage is executed. To reduce the influences of random initializations, the following is repeated for three seeds, different from the seed used for hyperparameter tuning. Baseline-Chen, Protonet, FOMAML, Random-Initialization, Best-Teacher, and Teacher-Concatenation are evaluated with test tasks of $\{1, 5, 50\}$-shots and 5-ways. For this, Baseline-Chen, Protonet, and FOMAML are trained from scratch on the generated data transformed to the test tasks shots- and ways-layout, or without transformation in the case of Baseline-Chen. The number of update iterations that are not fixed has been determined by early stopping during the previously described hyperparameter tuning phase of each method.

6. Results

In this chapter, we present the results of the few-shot test stage described in chapter 5. This chapter gives an extensive and detailed overview of the accuracy of individual methods before aggregating and discussing these in chapter 7. We present the results of all three datasets.

The following bar plots depict the mean and standard deviation of the accuracy across three seeds for each method. The exact accuracy values and the test losses are presented in tables in the appendix A.

Throughout the chapter, the following plotting scheme is used. The warm colors (yellow, orange, red) show the (meta-) training results with the original training data and without using generated data. These results represent an upper bound for the solutions in the assumed data-free setting. The light, medium and dark green colors represent the baselines without any generated data. The light, medium, and dark blue colors the case of different (meta-) learning algorithms using the data generated by DI. The bars are grouped with the shots number of the test tasks. Beneath each barplot, sampled generated images are shown for the respective teacher setting. These give a visual indication about the generated data's quality and domain similarity to the original data. Here, the images of a row belong to the same class. We depict the results of teachers with 2000 training instances on the left sub-figures. The results on the right sub-figures correspond to teachers with 8000 training instances.

6.1. DoubleMNIST

In this section, the results of DoubleMNIST, are presented. We present the results of the Conv-4 teachers and the ResNet-10 teachers.

6.1.1. Conv-4 teachers

Figure 6.1 presents the mean test task accuracy for each method (top) and some of the images generated by DI (bottom). Moreover, it depicts results of teachers trained with 2000 images (left) and 8000 images (right). As it can be seen, DI fails to produce images similar to the original training domain in both cases. The generated images look less like noise with 8000 teacher training instances. The produced images appear significantly more noisy than images produced by methods not considering the teacher source domain [60]. Consequently, we assume that any algorithm trained on the generated data fails to achieve competitive results. FOMAML achieves higher performance than ProtoNet and

Baseline-Chen. FOMAML performs worse than the baselines not using the generated data. On the other hand, the teacher-based baselines' high accuracy values (Best-Teacher and Teacher-Concatenation) compared to the algorithms accessing the original training data are remarkable.

6.1.2. ResNet-10 teachers

In figure 6.2 the results of teachers with the ResNet-10 architecture are presented. We present the mean test task accuracy for each method (top) and some of the images generated by DI (bottom). Here, we show teachers trained with 2000 images (left) and 8000 images (right). Like in previous results, DI does not generate images resembling the original training domain of the teachers. The resulting accuracy values of the algorithms trained on the generated data are worse than the results shown in figure 6.1. All algorithms using generated data are not predicting more accurate than a random classifier. The teacher-based baselines seem to benefit extremely from the ResNet-10 architecture compared to figure 6.1. They outperform algorithms accessing the original training data. Contrary to the results of Conv-4 teachers, the generated images with 2000 teacher training instances appears less noisy.

Overall, training on the DI generated data is counter-productive in all evaluated cases of the DoubleMNIST dataset. The teacher-based baselines show competitive accuracy values.

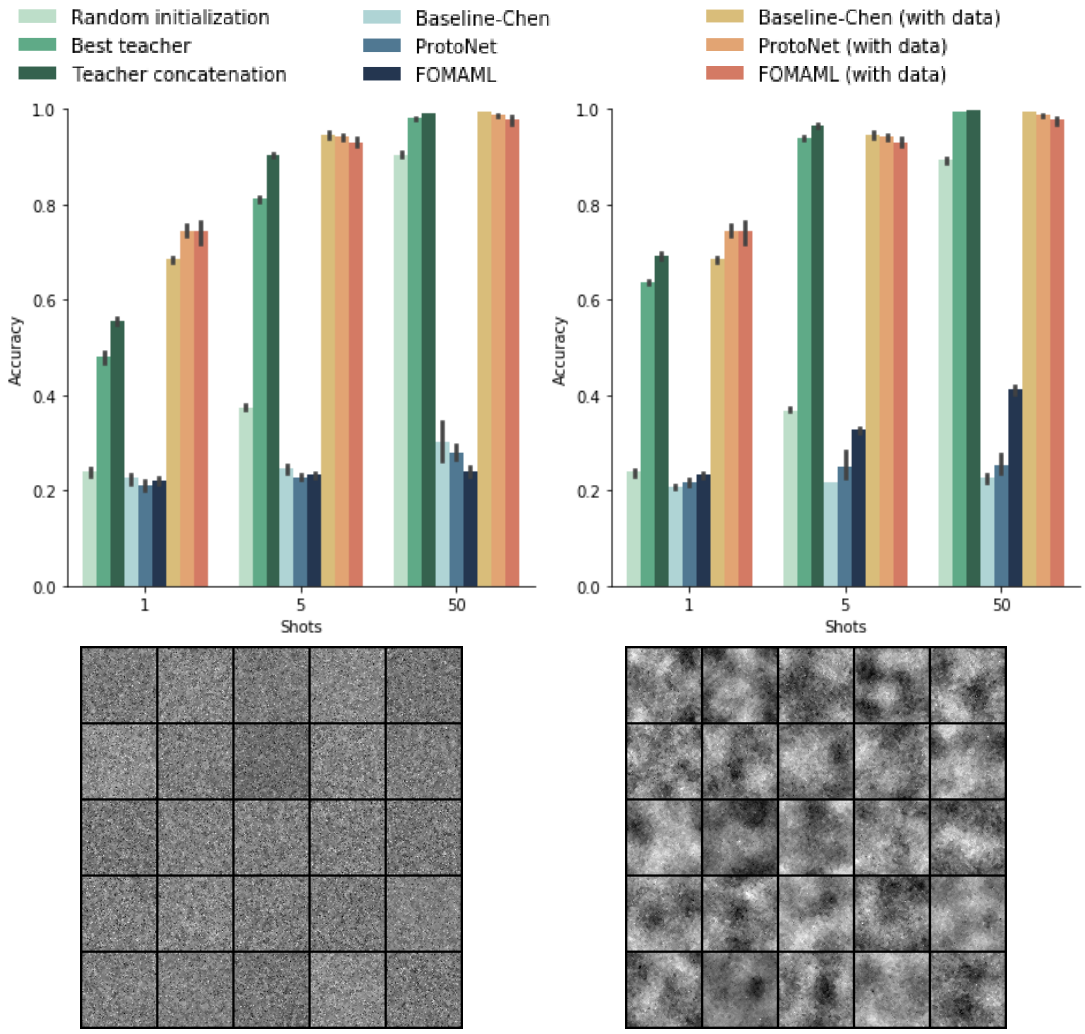


Figure 6.1.: Results on DoubleMNIST with Conv-4 teachers. Upper row: Testing accuracy of the various methods. Lower row: Samples of the DI generated images; each row represents a class. Left column: Teachers trained on 2000 images. Right column: Teachers trained on 8000 images.

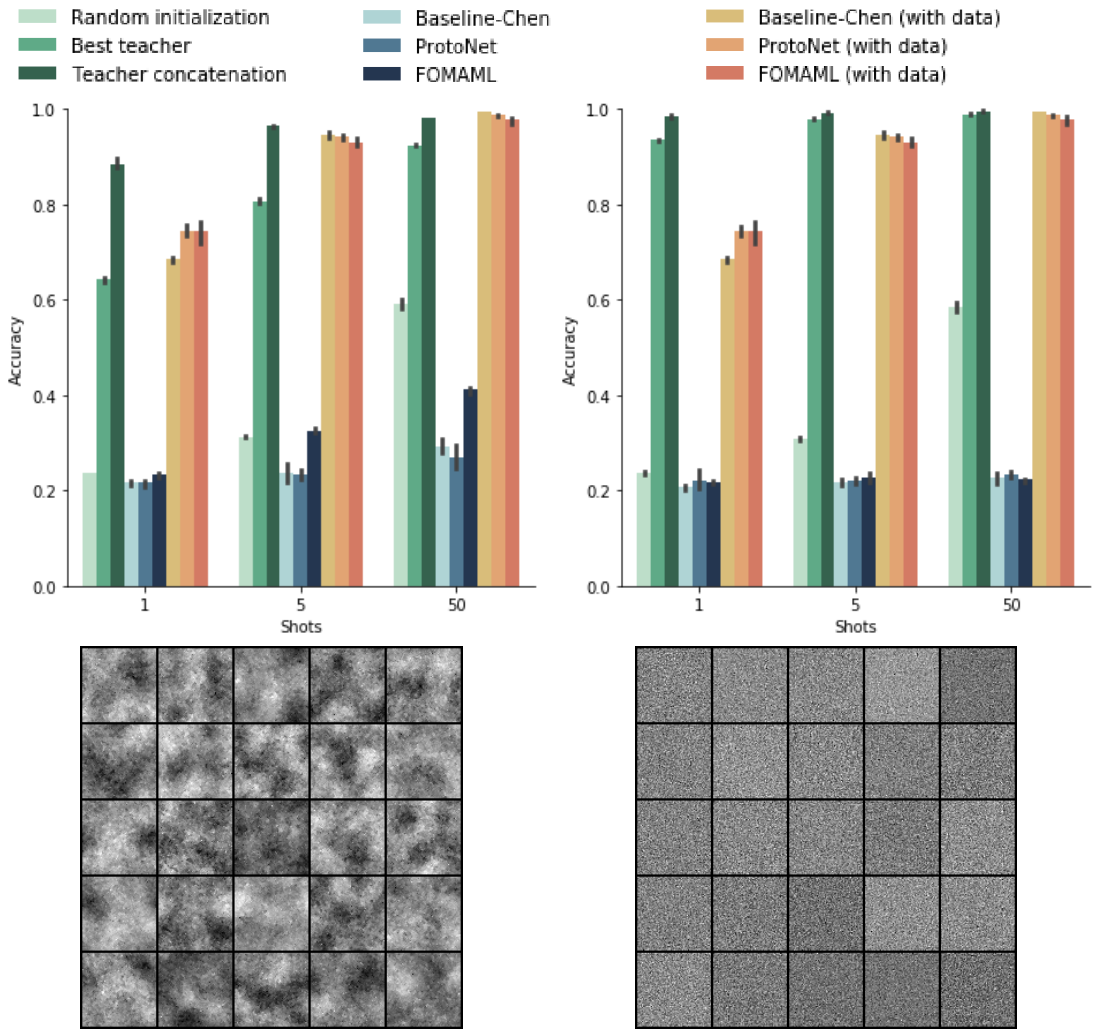


Figure 6.2.: Results on DoubleMNIST with ResNet-10 teachers. Upper row: Testing accuracy of the various methods. Lower row: Samples of the DI generated images; each row represents a class. Left column: Teachers trained on 2000 images. Right column: Teachers trained on 8000 images.

6.2. CIFAR-FS

In this section, the results for CIFAR-FS are presented. Compared to DoubleMNIST, CIFAR-FS is colored with a wider variety of different features, thus, presenting a higher challenge.

6.2.1. Conv-4 teachers

Figure 6.3 presents the mean test task accuracy for each method (top) and some of the images generated by DI (bottom). Moreover, the figure depicts results of teachers trained with 2000 images (left) and 8000 images (right).

The sampled examples of the generated data show that DI does not produce images similar to the original training domain. Unlike in the case of DoubleMNIST, training on the generated data is not counter-productive compared to the random initialization at lower shots. The Best-Teacher baseline benefits the most from additional training data of the teachers. Teacher-Concatenation shows the best results. With more teacher training data, it has better performance than the meta-learning algorithms and Baseline-Chen trained on the original training set.

6.2.2. ResNet-10 teachers

Figure 6.4 shows the mean test task accuracy for each method (top) and some of the images generated by DI (bottom). Moreover, it presents results of teachers trained with 2000 images (left) and 8000 images (right).

The sampled images generated by DI are substantially out of domain compared to the original training data. The algorithms using the generated data perform similarly like in the previous case of Conv-4 teachers depicted in figure 6.3. Nevertheless, the baselines without generated data are worse than in previous results shown in figure 6.3. They are the most competitive in this setting. More training data for the teachers improve the teacher-based baselines. We observe relatively low accuracy values of ProtoNet with 8000 teacher training instances.

Overall, the performance differences between generated-data-based algorithms and the baselines without generated data are smaller than in DoubleMNIST. However, we note that the pattern of Teacher-Concatenation being the most dominating approach is consistent.

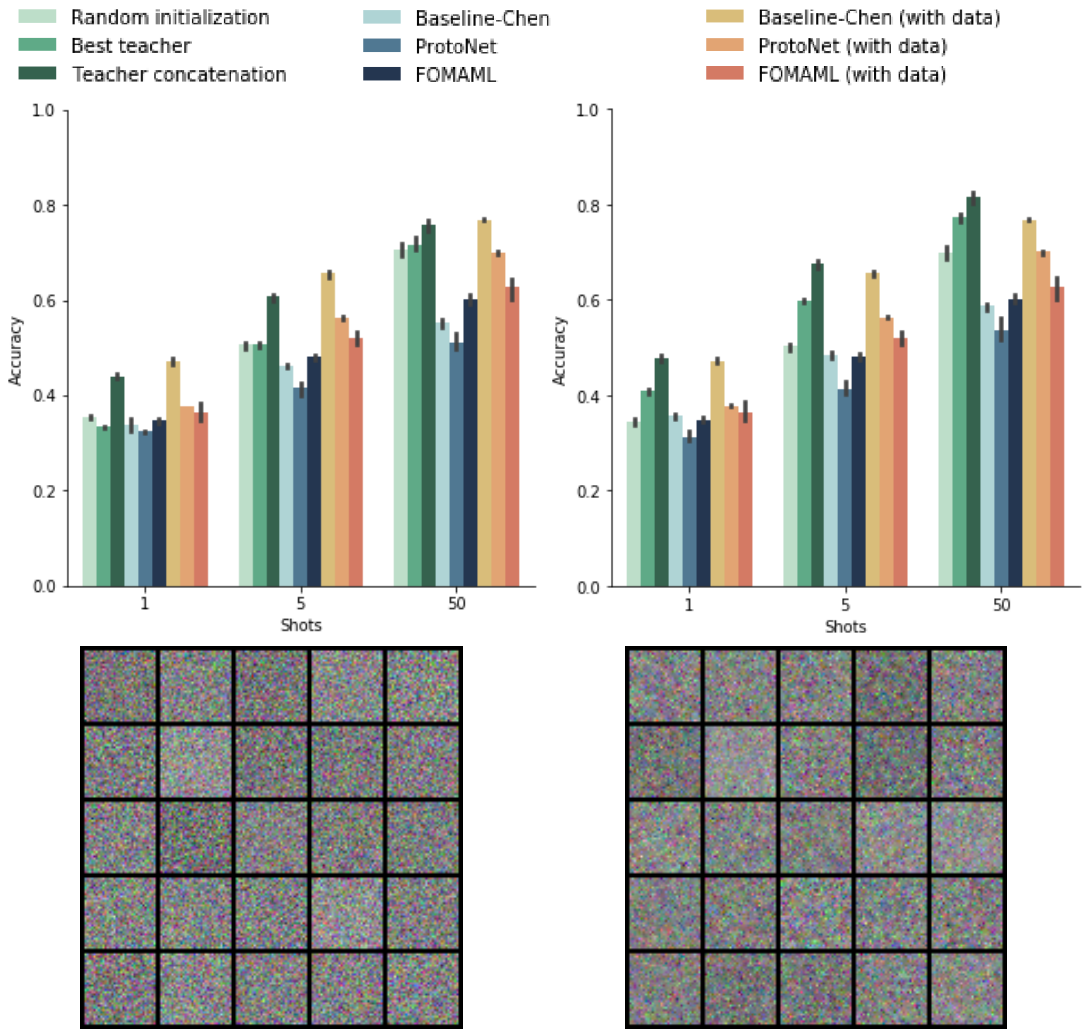


Figure 6.3.: Results on CIFAR-FS with Conv-4 teachers. Upper row: Testing accuracy of the various methods. Lower row: Samples of the DI generated images; each row represents a class. Left column: Teachers trained on 2000 images. Right column: Teachers trained on 8000 images.

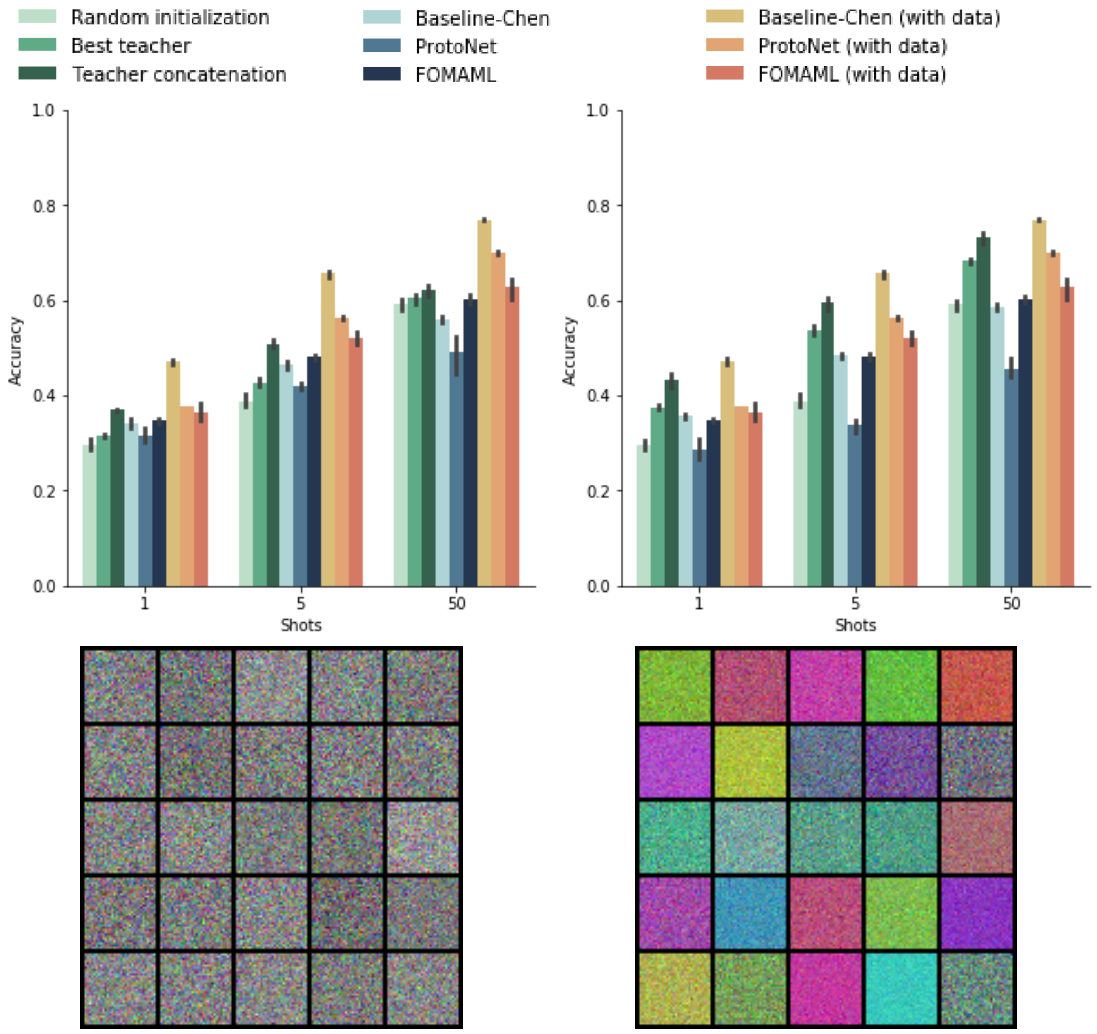


Figure 6.4.: Results on CIFAR-FS with ResNet-10 teachers. Upper row: Testing accuracy of the various methods. Lower row: Samples of the DI generated images; each row represents a class. Left column: Teachers trained on 2000 images. Right column: Teachers trained on 8000 images.

6.3. MiniImagenet

In this section, the experiments for MiniImagenet are presented. MiniImagenet is the most challenging of the used few-shot learning datasets in this thesis. The presentation layout of the different setups is kept consistent with the previous sections.

6.3.1. Conv-4 teachers

In figure 6.5, we present the mean test task accuracy for each method (top) and some of the images generated by DI (bottom). Moreover, we show results of teachers trained with 2000 images (left) and 8000 images (right) in the same figure.

Like in the previous cases, DI does not generate images close to the original training data domain. The algorithms using the generated images have low test task accuracy values. Baseline-Chen performs well in the 1-shot case compared to the other results. It almost beats all the baselines in the case of teachers trained with 2000 images. Teacher-Concatenation is the most dominant method. FOMAML performs remarkably badly and is outperformed by the random initialization in all cases. We observe that the meta-learning algorithms perform worse with 8000 teacher training images than with 2000 teacher training images.

6.3.2. ResNet-10 teachers

Figure 6.6 presents the mean test task accuracy for each method (top) and some of the images generated by DI (bottom). In the figure, we show results of teachers trained with 2000 images (left) and 8000 images (right).

Consistently to previous results, DI does not generate images resembling the original training domain of the teachers. Baseline-Chen performs well compared to the other algorithms. It performs the second highest after the teacher-based baselines without considering algorithms using the original data. The teacher-based baselines benefit from higher teacher training data sizes. We observe that Teacher-Concatenation performs the worst of all baselines in the 50-shot case with 2000 train images. FOMAML is consistently the worst performing method and does not perform better than the random initialization baseline in any case.

In all MiniImagenet cases, ProtoNet and FOMAML yield a lower performance compared to the other methods by a big margin. Using the generated data is more like a handicap than a helpful support for better generalization performance. However, comparing the performance of the algorithms trained with the original training data, it becomes clear that the hyperparameter tuning settings for FOMAML and ProtoNet are inappropriate for MiniImagenet and require a reassessment with more computational resources. Baseline-Chen, FOMAML, and ProtoNet are supposed to perform approximately equally well according to Chen et al. [11]. We present results confirming this claim in appendix B.

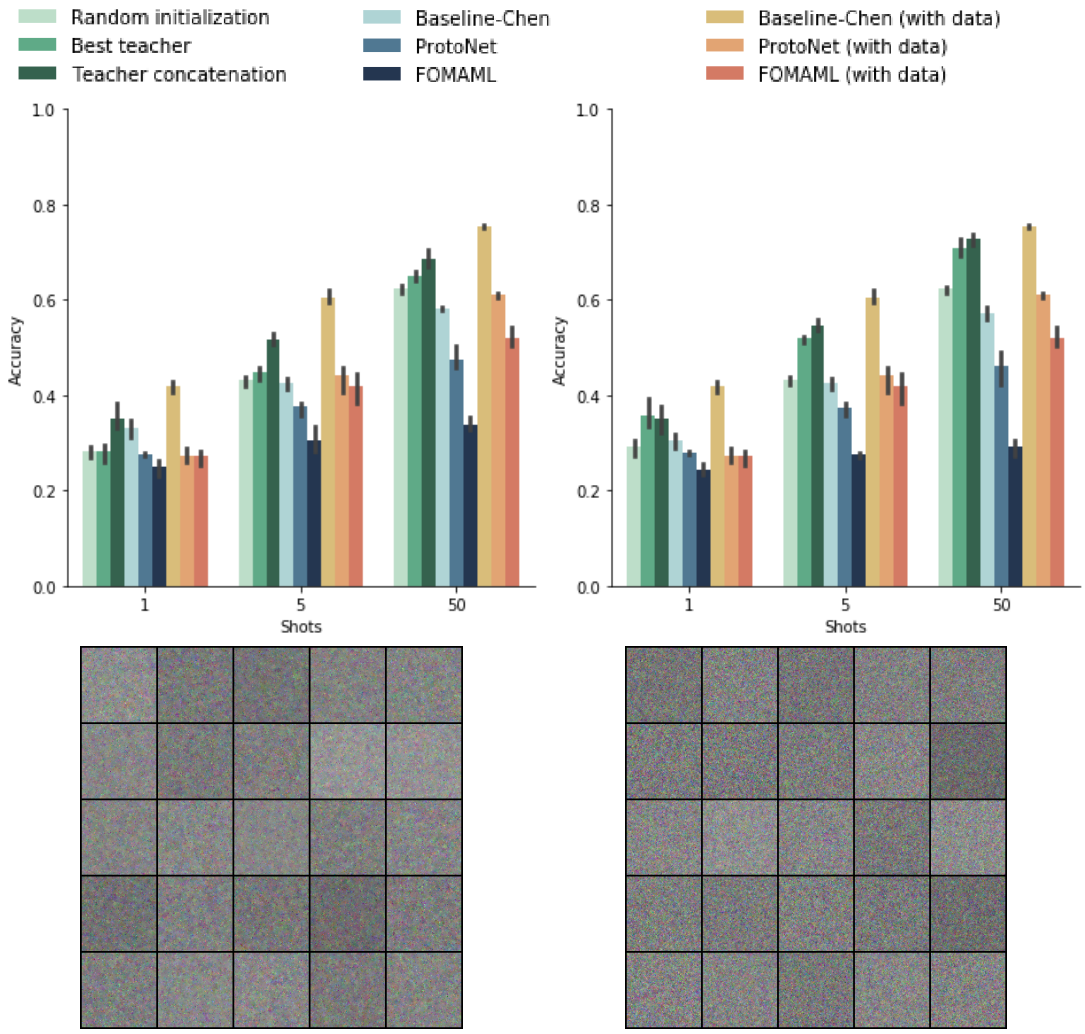


Figure 6.5.: Results on MiniImagenet with Conv-4 teachers. Upper row: Testing accuracy of the various methods. Lower row: Samples of the DI generated images; each row represents a class. Left column: Teachers trained on 2000 images. Right column: Teachers trained on 8000 images.

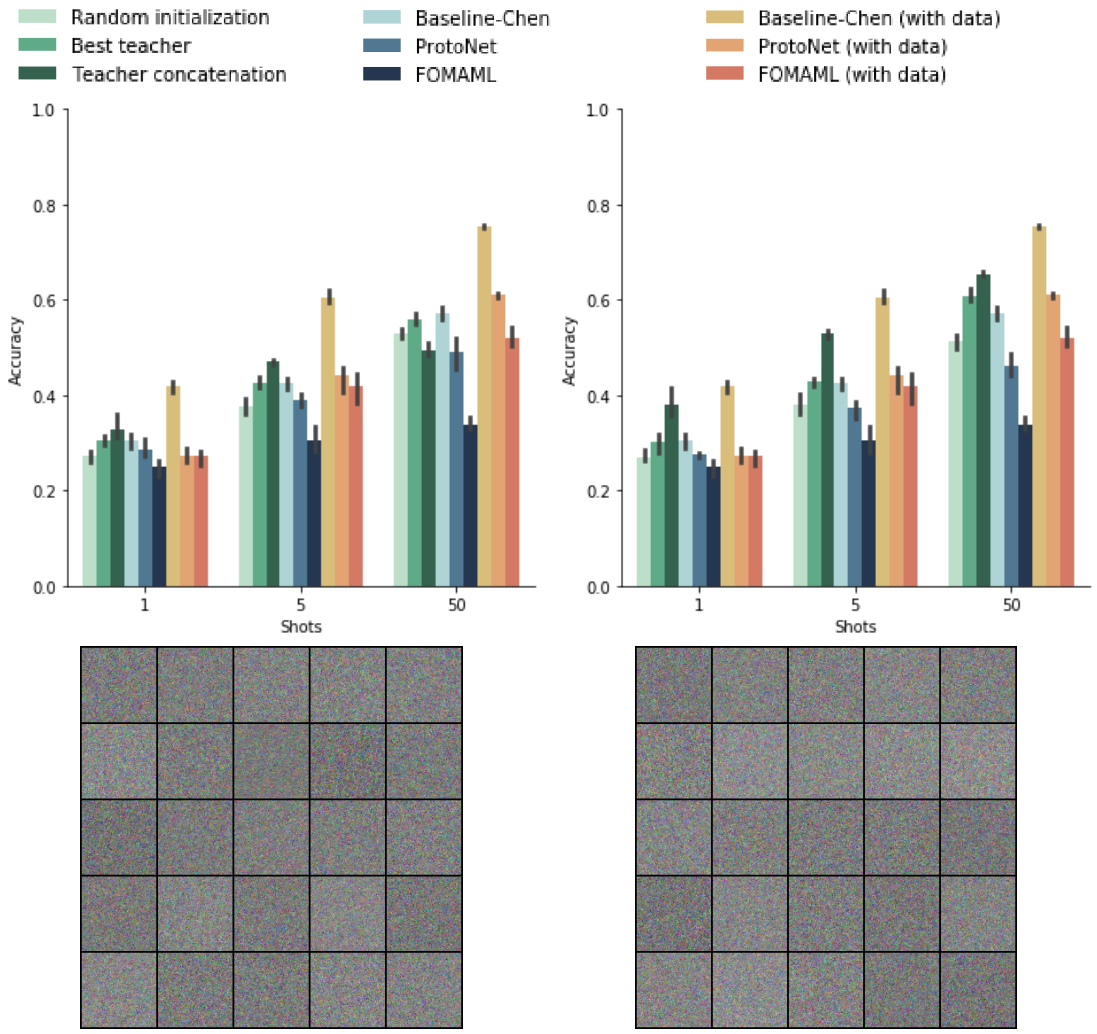


Figure 6.6.: Results on MiniImagenet with ResNet-10 teachers. Upper row: Testing accuracy of the various method. Lower row: Samples of the by DI generated images; each row represents a class. Left column: Teachers trained on 2000 images. Right column: Teachers trained on 8000 images.

7. Discussion

In this chapter, we discuss the experiments' results and aggregate subsets of these to answer the research questions defined in chapter 5. The presented information and figures are summaries of the previous chapter.

7.1. Research questions

This section aims to answer the research questions stated in section 5.1 by discussing and analyzing the produced results of chapter 6. We do not use all the results for each question and mention the aggregated subset in each subsection. In a new setting, we assume that not all algorithms are evaluated during validation. We assume one algorithm is selected randomly for evaluation. So, to not introduce a selection bias [66], we report the mean value instead of the maximum value of the aggregated algorithm results. Furthermore, we do not include results of the original training data in the aggregations unless explicitly stated otherwise.

Optimization-based versus metric-based meta-learning

In the given setting, we want to investigate if an optimization-based meta-learning algorithm performs better or worse than a metric-based one. Due to this, FOMAML and ProtoNet have been included in the experiments as basic representatives of these two categories of meta-learning algorithms. In figure 7.1 the results of all the different teacher and task settings have been averaged for each dataset for FOMAML and ProtoNet. As can be observed in the figure, FOMAML seems to do better on average on the easier datasets CIFAR-FS and DoubleMNIST, while ProtoNet outperforms in MiniImagenet. Based on these observations, we hypothesize that FOMAML is preferable for generated datasets of lower resolution and ProtoNet for higher resolutions. However, this may be due to too strict tuning settings. FOMAML with the given tuning settings shows worse results on the original training data than originally reported [11].

Meta-learning versus conventional learning

Furthermore, we asked if meta-learning on generated data is beneficial compared to conventional learning. Chen et al. [11] showed that Baseline-Chen is better for large domain shifts in their evaluated experimental setting. Because we train on generated data, the domain difference in our case is almost surely larger than between original training tasks and test tasks. Figure 7.2 shows how the conventional learning algorithm Baseline-Chen compares to the meta-learning algorithms FOMAML and ProtoNet. We observe

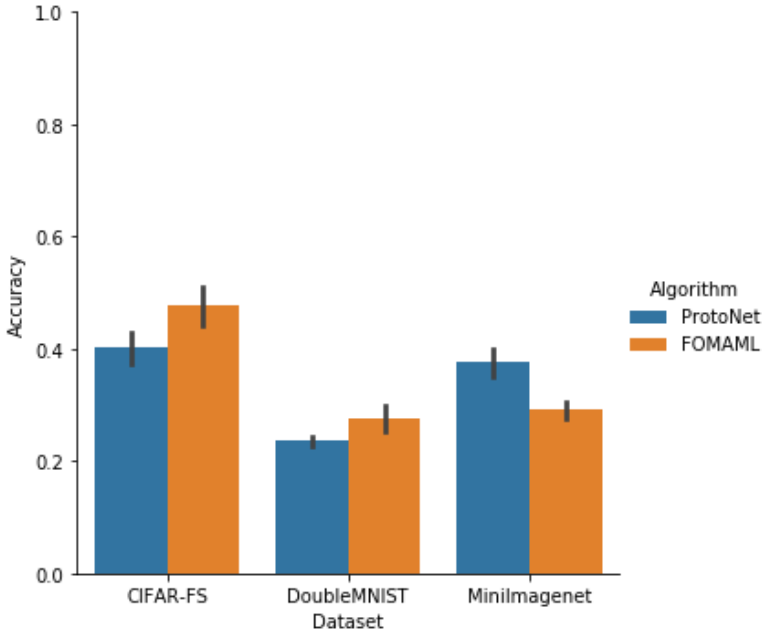


Figure 7.1.: Average accuracy values across all settings of a given dataset for ProtoNet and FOMAML.

no clear differences in the cases of CIFAR-FS and DoubleMNIST. In MiniImagenet, we see a higher accuracy of conventional learning.

Few-shot learning with generated data versus without generated data

In this section we analyze whether training on the generated data is beneficial for learning new tasks.

Figure 7.3 depicts the average performance of algorithms without generated data and algorithms with generated data for all settings of each dataset. Algorithms without generated data are Random-Initialization, Best-Teacher, and Teacher-Concatenation. Algorithms with generated data are FOMAML, ProtoNet, and Baseline-Chen. We observe that learning on generated data is unfavorable for CIFAR-FS and MiniImagenet, and highly unfavorable in the case of DoubleMNIST. Following the sampled generated images in chapter 6, we expect these results since the images generated by DI seem to be mostly noise. Improving the data generation process has most likely the biggest impact on performance.

Few-shot learning with generated data versus with original data

In the proposed setting, the original training data of the teachers is unavailable. Consequently, using the original data should be seen as an upper bound, and is not applicable in the adressed setting. We assess the performance decrease caused by the generation

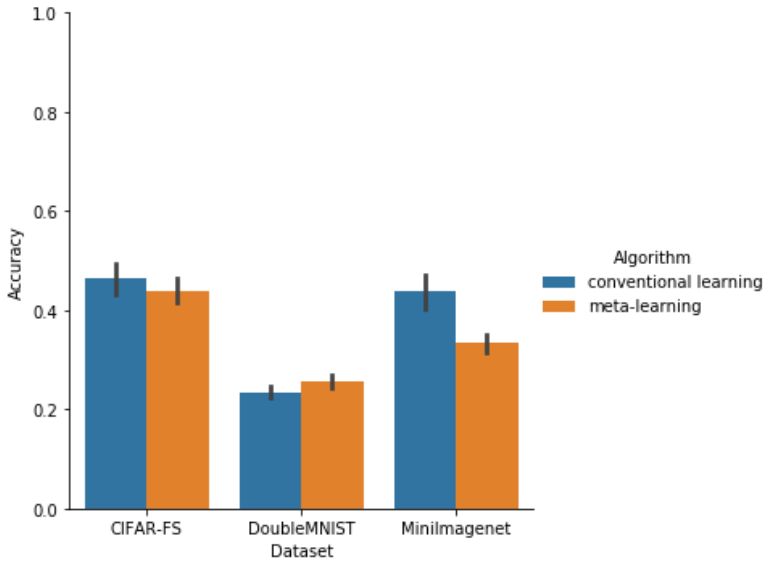


Figure 7.2.: Average accuracy values across all settings of a given dataset for meta-learning algorithms FOMAML and ProtoNet, and the conventional learning algorithms Baseline-Chen.

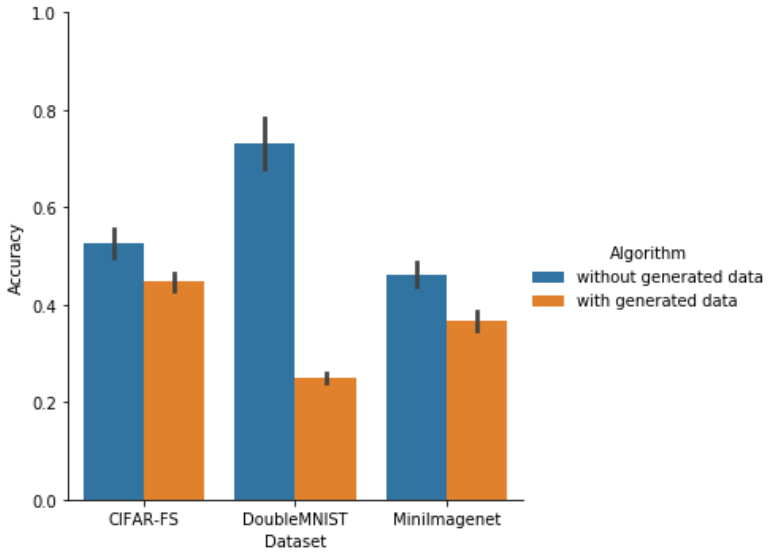


Figure 7.3.: Average accuracy values across all settings of a given dataset for algorithms without generated data, i.e. Random-Initialization, Best-Teacher, and Teacher-Concatenation, and algorithms with generated data, i.e. FOMAML, ProtoNet, Baseline-Chen.

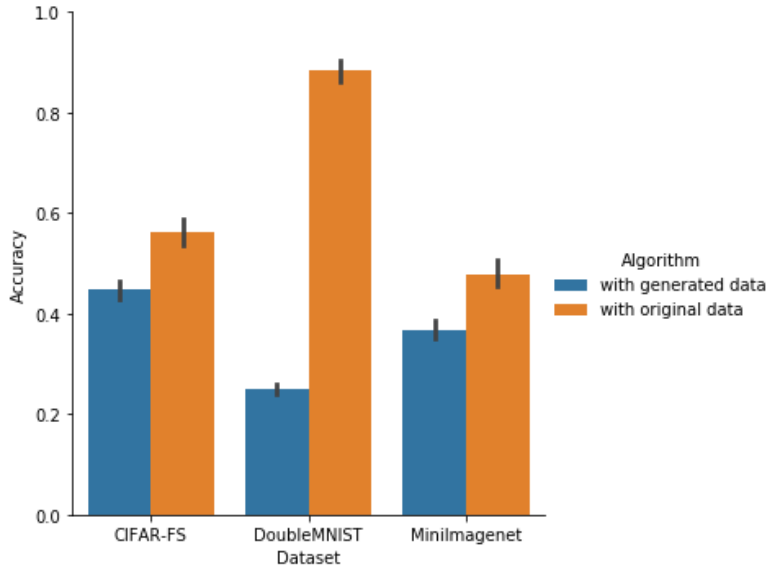


Figure 7.4.: Average accuracy values across all settings of a given dataset for algorithms, i.e. FOMAML, ProtoNet, and Baseline-Chen, trained with generated data and with the original training data of the teachers.

process by comparing the models trained in generated data to models trained on the original data. The resulting accuracy values are depicted in figure 7.4 with FOMAML, ProtoNet, and Baseline-Chen.. While the difference between MiniImagenet and CIFAR-FS is substantial, the difference in the case of DoubleMNIST is extreme. This observation can be interpreted as an indication that the data generation process for DoubleMNIST was tuned with wrong settings.

The test task domain is closer to the domain of the training tasks in the case of DoubleMNIST compared to CIFAR-FS and MiniImagenet. Every single digit occurring in the testing classes (a random subset of 00-99) occurs several times in the training classes (another subset of 00-99). E.g., if the class "79" was not seen in the training tasks, the digits "7" and "9" have been seen several times in other classes. This could explain the extreme gap between original data and generated data. Such similarities are not present in the CIFAR-FS and MiniImagenet dataset.

Teacher architectures and the teacher training data size

Another aspect we investigate is how different teacher architectures and teacher training data sizes influence the average performance of the introduced methods. We assume that larger teacher architectures and more data for the teacher training improve the representation of the training domain by the BN statistics. We aggregate all algorithms related to teachers over the different teacher settings. The considered algorithms are FOMAML, ProtoNet, Baseline-Chen, Best-Teacher, and Teacher-Concatenation. The

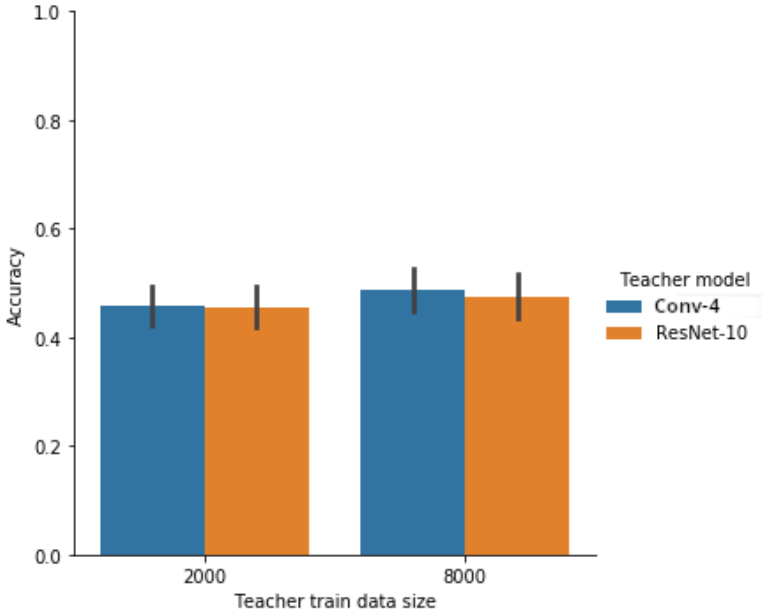


Figure 7.5.: Average accuracy values of all datasets and all teacher dependent methods, i.e. FOMAML, ProtoNet, Baseline-Chen, Best-Teacher, and Teacher-Concatenation, for two different teacher training set sizes.

result can be seen in figure 7.5. The presented results show that a larger architecture and more training data do not lead to better performance. This is contrary to the initial assumption. The trained teachers in this thesis seem to not be diverse enough as neither the architecture nor the amount of training data influence the outcome. On the other hand, more teacher models are available for smaller training data sizes, which might have covered up the expected discrepancy.

Few-shot versus many-shot learning

This chapter's last research question is how much the number of shots of the few-shot learning task influences the methods' performance. We consider Random-Initialization, Best-Teacher, and Teacher-Concatenation as the baseline algorithms without generated data. FOMAML, ProtoNet, and Baseline-Chen are algorithms using generated data.

The figure 7.6 shows that methods without generated data perform better on average on the evaluated datasets for all evaluated shots. Additionally, the baseline algorithms without generated data benefit more from higher number of shots than algorithms using generated data.

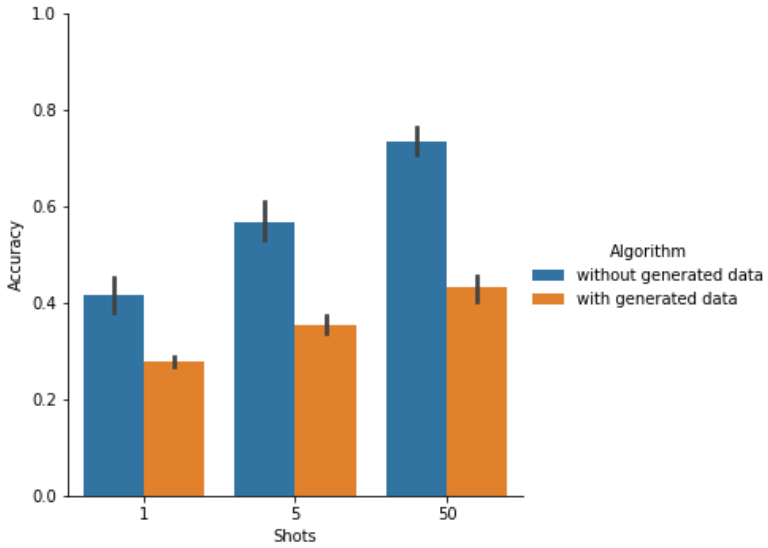


Figure 7.6.: Average accuracy values of all datasets of algorithms without generated data, i.e. Random-Initialization, Best-Teacher, and Teacher-Concatenation, and algorithms with generated data, i.e. FOMAML, ProtoNet, Baseline-Chen, for each evaluated amount of shots during testing.

8. Conclusion and Outlook

In this chapter, we present a conclusion of the findings and contributions in this work. An outlook giving guidance for future research in this direction follows.

8.1. Conclusion

The presented thesis introduced and conducted extensive experiments on methods for the novel setting of data-free few-shot learning given multiple pre-trained teacher models. We defined data-free few-shot learning as learning a few-shot task without the assistance of related auxiliary tasks. Instead, we assumed that a set of teacher models trained on such tasks are available.

Furthermore, we proposed several novel methods. These can be categorized into two groups. One group includes baselines, which do not require any generated data. Consequently, they do not use a meta-training stage. The other group consists of methods meta-training or pre-training a feature extractor with generated data. The considered algorithms in this group were FOMAML [22], ProtoNet [79], and Baseline-Chen [11]. The generated data was produced by a data-free knowledge distillation algorithm, DI [96], which was applied to each given teacher model. Moreover, we formulated the data-free few-shot learning algorithm to allow using other data generation methods. According to the evaluated experiments, the best performing solution is a novel baseline called Teacher-Concatenation, which combines the feature outputs of multiple teachers. The combined feature outputs are fed to a linear classifier, which is then adapted to the few-shot task.

The experiments showed inferior performances of the methods using the generated data. However, it is inconclusive if this is due to the data generation method or the available teacher models. This inconclusiveness is a consequence of using few-shot learning datasets of already available implementations [16]. The used datasets are too small in their size to train large teacher architectures. Consequently, the teacher models are significantly smaller than in the original work of DI [96]. Additionally, each teacher was trained with a smaller dataset compared to the work of Yin et al. [96].

8.2. Outlook

Based on the results in chapter 6 and the discussion in chapter 7, we propose potential next steps for further investigations of this topic.

8.2.1. Hyperparameter optimization

For all the evaluated experiments, early stopping patience of 3 was used for each hyperparameter optimization process. As discussed in section 7.1, training on the original dataset does not lead to the results reported in the original ProtoNet [79] and FOMAML [22] works. This is a clear indication that the early stopping was too restrictive, given that the hyperparameter ranges cover the optimal solution.

A small evaluation presented in appendix B strengthens this argument. The evaluated setting shows that higher accuracy values can be reached with a higher computational budget. Furthermore, we chose hyperparameter ranges for the learning methods based on previous experiences. For this, several implementations and literature exist that had acted as a guide [79; 22; 11; 6]. This may not be the case for DI. To the best of the author's knowledge, DI was not applied to such small teacher architectures trained with few images compared to the original work. We used models with five million weights or less, while Yin et al. [96] used models with at least ten million weights. Additionally, we trained each teacher with less than a quarter of the data used in the original work [96]. The hyperparameter ranges for DI were selected considering the results of manually specified values.

This motivates to rerun the hyperparameter tuning of DI with wider hyperparameter ranges. Furthermore, one could use a more sophisticated optimization algorithm given the extremely long evaluation times of a single configuration. E.g., bayesian optimization [24] may be an appropriate choice.

8.2.2. Investigation of data generation with smaller architectures

It is possible that the hyperparameter tuning was designed correctly, and the teacher models do not hold enough information for generating images similar to the original domain. Several different image generation methods [13; 94; 10] have to be compared in a wide range of experiments involving different teacher architectures to check this hypothesis.

So far, data-free knowledge distillation methods have been considered for the task of generating images. However, it is possible that different methods [87] in other research areas provide more suitable approaches for the given setting. The data generation process may require more experiments outside the few-shot learning setting of the present thesis.

Investigation of DeepInversion

We observed that the implementation of DI has more hyperparameters than have been reported in the paper [96]. We present examples of not reported hyperparameters in the following. The generated images are randomly flipped between DI iterations by a specific probability. In every optimization step, noise of certain strength is added to the images. The BN loss of the first layer receives an additional weighting compared to the BN loss values of the other layers. Considering these 'hidden' hyperparameters, the resulting

hyperparameter optimization space is above ten dimensions. This makes hyperparameter optimization challenging. In the original implementation, the hyperparameters scale directly with the image resolution, the number of BN layers in the teacher, the number of input channels, and the batch size. This is a consequence of using the sum in all DI optimization terms without normalizing. With changes in the mentioned sizes, the terms change differently in value through a changing number of summands. This makes it difficult to estimate what hyperparameter ranges of the loss coefficients are appropriate since a modification of the setting scales these ranges to different values.

Furthermore, we decided to use early stopping and ASHA for the DI optimization. I.e., DI was handled like the tuning of a machine learning model. However, DI optimizes noise and not model parameters. During the DI optimization, the feature maps of the noise are adjusted to follow the distribution given by the teacher's BN statistics. Furthermore, DI optimization removes noise in the generated data with the prior-losses. As a consequence, different optimization rules may. This raises the question whether overfitting can occur. If overfitting is not an appearing phenomenon during DI optimization, early stopping will not be required. The minimum of the DI optimization objective is equally good as the iteration with the best validation loss. Since a newly initialized model is trained on the generated data for every validation step, many computational evaluations could be saved. It could be possible to skip these computationally costly validation evaluations. Thus, more hyperparameter evaluations can be conducted with the same resources. DI may require more experiments in different problem settings to answer these questions.

8.2.3. Creation of novel datasets

As stated, the teacher architectures may have been too small to allow better results. Or, the the training data size of each teacher was not big enough. Unfortunately, the available few-shot learning datasets do not have more data instances [82; 89; 46]. So, if we want to evaluate teachers trained with more data instances, new datasets are required. To construct these, the openly accessible ImageNet database [17] can be used. The resulting datasets can be made several times (> 4) the size of CIFAR-FS. This would allow training the teacher architectures used in the original DI publication [96].

8.2.4. End-to-end-based data generation and meta-training

In the evaluated algorithm, the data generation process is independent of the (meta-) training stage. The validation loss is used for early stopping and hyperparameter tuning. Other works on data-free knowledge distillation [60] showed that a student model does not necessarily require generated images close to the original training domain. So, it might be beneficial to focus more on creating methods that integrate the meta-learning loss into the data generation objective as we suggested in subsection 4.2.1. Thus, the images do not have to have a similar domain to benefit the student (meta-) training stage.

8.2.5. Usage of validation tasks

The last and probably most critical point is the usage of validation classes and tasks. In a practical setting, no training data is available, and the target task has few data instances. This gives rise to the question of where the data for the hyperparameter tuning comes from.

A possible first trial to investigate this problem could be to optimize the hyperparameters of the meta-learning algorithms on the generated data instead of using the validation tasks. Furthermore, a data generation process with a small hyperparameter space and robust default hyperparameters could be used.

As an alternative, the generated images and their similarity to the original data domain could be visually evaluated by the user. This would lead to a costly manual evaluation of several datasets. In this scenario, it is impossible to construct an automatic hyperparameter tuning framework like it was done in this thesis. Furthermore, not using the validation data may drastically reduce the performance of the baselines without generated data.

Appendices

A. Results in table-format

In this chapter, the same results as in chapter 6 are presented, only this time in table-format instead of bar plots. The accuracy values are the mean of three seeds, and the 95% confidence interval is approximated by two times the standard deviation through the empirical rule under the assumption of a normal distribution.

A.1. DoubleMNIST

A.1.1. 4-layer CNN teachers

2000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	22.57 +/- 1.86
	5	24.67 +/- 1.22
	50	30.32 +/- 6.62
Baseline-Chen (with data)	1	68.45 +/- 0.71
	5	94.59 +/- 1.02
	50	99.51 +/- 0.16
Best teacher	1	48.14 +/- 2.11
	5	81.2 +/- 0.84
	50	98.07 +/- 0.34
FOMAML	1	22.09 +/- 1.1
	5	23.3 +/- 0.93
	50	23.95 +/- 1.68
FOMAML (with data)	1	74.29 +/- 3.54
	5	93.0 +/- 1.35
	50	97.76 +/- 1.31
ProtoNet	1	21.19 +/- 1.45
	5	22.78 +/- 0.91
	50	28.07 +/- 2.56
ProtoNet (with data)	1	74.59 +/- 1.87
	5	94.2 +/- 0.96
	50	98.8 +/- 0.27
Random initialization	1	23.93 +/- 1.42
	5	37.38 +/- 0.92
	50	90.52 +/- 0.84
Teacher concatenation	1	55.54 +/- 1.13
	5	90.31 +/- 0.58
	50	99.26 +/- 0.06

8000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	20.61 +/- 0.48
	5	21.75 +/- 0.1
	50	22.74 +/- 1.59
Baseline-Chen (with data)	1	68.45 +/- 0.71
	5	94.59 +/- 1.02
	50	99.51 +/- 0.16
Best teacher	1	63.67 +/- 0.53
	5	93.87 +/- 0.45
	50	99.53 +/- 0.16
FOMAML	1	23.32 +/- 0.83
	5	32.64 +/- 0.96
	50	41.16 +/- 1.37
FOMAML (with data)	1	74.29 +/- 3.54
	5	93.0 +/- 1.35
	50	97.76 +/- 1.31
ProtoNet	1	21.74 +/- 1.2
	5	24.93 +/- 4.82
	50	25.17 +/- 3.51
ProtoNet (with data)	1	74.59 +/- 1.87
	5	94.2 +/- 0.96
	50	98.8 +/- 0.27
Random initialization	1	23.87 +/- 1.12
	5	36.84 +/- 0.57
	50	89.27 +/- 0.68
Teacher concatenation	1	69.07 +/- 1.16
	5	96.51 +/- 0.5
	50	99.76 +/- 0.04

A.1.2. ResNet-10 teachers

2000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	21.59 +/- 0.74
	5	23.79 +/- 3.13
	50	29.16 +/- 2.99
Baseline-Chen (with data)	1	68.45 +/- 0.71
	5	94.59 +/- 1.02
	50	99.51 +/- 0.16
Best teacher	1	64.3 +/- 1.11
	5	80.67 +/- 1.06
	50	92.39 +/- 0.21
FOMAML	1	23.32 +/- 0.83
	5	32.61 +/- 0.89
	50	41.11 +/- 1.26
FOMAML (with data)	1	74.29 +/- 3.54
	5	93.0 +/- 1.35
	50	97.76 +/- 1.31
ProtoNet	1	21.61 +/- 1.02
	5	23.4 +/- 1.49
	50	26.82 +/- 4.24
ProtoNet (with data)	1	74.59 +/- 1.87
	5	94.2 +/- 0.96
	50	98.8 +/- 0.27
Random initialization	1	23.81 +/- 0.06
	5	31.23 +/- 0.43
	50	59.25 +/- 1.53
Teacher concatenation	1	88.58 +/- 1.5
	5	96.39 +/- 0.5
	50	98.2 +/- 0.16

8000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	20.67 +/- 0.81
	5	21.67 +/- 1.06
	50	22.62 +/- 1.84
Baseline-Chen (with data)	1	68.45 +/- 0.71
	5	94.59 +/- 1.02
	50	99.51 +/- 0.16
Best teacher	1	93.48 +/- 0.11
	5	97.96 +/- 0.28
	50	98.99 +/- 0.28
FOMAML	1	21.6 +/- 0.54
	5	22.69 +/- 1.65
	50	22.24 +/- 0.65
FOMAML (with data)	1	74.29 +/- 3.54
	5	93.0 +/- 1.35
	50	97.76 +/- 1.31
ProtoNet	1	22.16 +/- 3.25
	5	22.0 +/- 0.95
	50	23.29 +/- 1.09
ProtoNet (with data)	1	74.59 +/- 1.87
	5	94.2 +/- 0.96
	50	98.8 +/- 0.27
Random initialization	1	23.64 +/- 0.45
	5	31.0 +/- 0.72
	50	58.38 +/- 1.56
Teacher concatenation	1	98.36 +/- 0.51
	5	99.3 +/- 0.15
	50	99.56 +/- 0.17

A.2. CIFAR-FS

A.2.1. 4-layer CNN teachers

2000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	33.69 +/- 2.14
	5	46.13 +/- 0.57
	50	55.35 +/- 1.5
Baseline-Chen (with data)	1	47.04 +/- 0.9
	5	65.62 +/- 1.03
	50	76.86 +/- 0.31
Best teacher	1	33.49 +/- 0.39
	5	50.73 +/- 1.11
	50	71.69 +/- 2.28
FOMAML	1	34.72 +/- 0.8
	5	48.12 +/- 0.99
	50	60.11 +/- 1.54
FOMAML (with data)	1	36.33 +/- 3.22
	5	52.01 +/- 2.13
	50	62.66 +/- 3.79
ProtoNet	1	32.44 +/- 0.26
	5	41.68 +/- 2.31
	50	51.17 +/- 2.65
ProtoNet (with data)	1	37.69 +/- 0.21
	5	56.3 +/- 0.33
	50	70.04 +/- 0.57
Random initialization	1	35.44 +/- 0.62
	5	50.66 +/- 1.36
	50	70.56 +/- 2.18
Teacher concatenation	1	43.86 +/- 0.79
	5	60.73 +/- 1.22
	50	75.91 +/- 1.95

8000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	35.61 +/- 0.95
	5	48.31 +/- 0.99
	50	58.68 +/- 1.21
Baseline-Chen (with data)	1	47.04 +/- 0.9
	5	65.62 +/- 1.03
	50	76.86 +/- 0.31
Best teacher	1	40.96 +/- 1.12
	5	59.95 +/- 0.5
	50	77.51 +/- 1.46
FOMAML	1	34.72 +/- 0.8
	5	48.14 +/- 0.9
	50	60.15 +/- 1.56
FOMAML (with data)	1	36.33 +/- 3.22
	5	52.01 +/- 2.13
	50	62.66 +/- 3.79
ProtoNet	1	31.12 +/- 1.78
	5	41.24 +/- 2.49
	50	53.49 +/- 3.91
ProtoNet (with data)	1	37.69 +/- 0.21
	5	56.3 +/- 0.33
	50	70.04 +/- 0.57
Random initialization	1	34.38 +/- 0.88
	5	50.27 +/- 1.2
	50	69.77 +/- 2.08
Teacher concatenation	1	47.63 +/- 1.13
	5	67.56 +/- 1.56
	50	81.51 +/- 1.91

A.2.2. ResNet-10 teachers

2000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	33.98 +/- 1.64
	5	46.51 +/- 1.2
	50	55.86 +/- 0.88
Baseline-Chen (with data)	1	47.04 +/- 0.9
	5	65.62 +/- 1.03
	50	76.86 +/- 0.31
Best teacher	1	31.53 +/- 0.66
	5	42.54 +/- 1.36
	50	60.44 +/- 1.87
FOMAML	1	34.72 +/- 0.8
	5	48.09 +/- 0.94
	50	60.13 +/- 1.58
FOMAML (with data)	1	36.33 +/- 3.22
	5	52.01 +/- 2.13
	50	62.66 +/- 3.79
ProtoNet	1	31.46 +/- 2.39
	5	41.82 +/- 1.2
	50	49.07 +/- 6.65
ProtoNet (with data)	1	37.69 +/- 0.21
	5	56.3 +/- 0.33
	50	70.04 +/- 0.57
Random initialization	1	29.54 +/- 1.86
	5	38.84 +/- 2.28
	50	59.37 +/- 1.87
Teacher concatenation	1	36.91 +/- 0.25
	5	50.83 +/- 1.37
	50	62.17 +/- 2.05

8000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	35.61 +/- 0.95
	5	48.31 +/- 0.99
	50	58.68 +/- 1.21
Baseline-Chen (with data)	1	47.04 +/- 0.9
	5	65.62 +/- 1.03
	50	76.86 +/- 0.31
Best teacher	1	37.4 +/- 0.83
	5	53.82 +/- 1.79
	50	68.22 +/- 0.85
FOMAML	1	34.72 +/- 0.8
	5	48.13 +/- 0.98
	50	60.11 +/- 1.55
FOMAML (with data)	1	36.33 +/- 3.22
	5	52.01 +/- 2.13
	50	62.66 +/- 3.79
ProtoNet	1	28.45 +/- 3.44
	5	33.72 +/- 2.32
	50	45.68 +/- 3.03
ProtoNet (with data)	1	37.69 +/- 0.21
	5	56.3 +/- 0.33
	50	70.04 +/- 0.57
Random initialization	1	29.49 +/- 1.76
	5	38.71 +/- 2.29
	50	59.09 +/- 1.88
Teacher concatenation	1	43.22 +/- 2.32
	5	59.64 +/- 2.37
	50	73.34 +/- 1.88

A.3. MiniImagenet

A.3.1. 4-layer CNN teachers

2000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	33.04 +/- 2.99
	5	42.64 +/- 1.88
	50	58.09 +/- 0.56
Baseline-Chen (with data)	1	41.96 +/- 1.95
	5	60.58 +/- 2.11
	50	75.4 +/- 0.71
Best teacher	1	28.22 +/- 3.05
	5	44.71 +/- 2.31
	50	65.04 +/- 1.81
FOMAML	1	25.02 +/- 2.86
	5	30.67 +/- 4.37
	50	33.71 +/- 2.2
FOMAML (with data)	1	27.13 +/- 2.76
	5	41.8 +/- 5.1
	50	51.89 +/- 3.38
ProtoNet	1	27.51 +/- 0.49
	5	37.58 +/- 2.42
	50	47.56 +/- 3.79
ProtoNet (with data)	1	27.18 +/- 2.51
	5	44.07 +/- 4.82
	50	60.93 +/- 0.85
Random initialization	1	28.29 +/- 1.87
	5	43.16 +/- 1.75
	50	62.47 +/- 1.61
Teacher concatenation	1	35.02 +/- 4.81
	5	51.67 +/- 2.02
	50	68.56 +/- 3.02

8000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	30.4 +/- 2.47
	5	42.71 +/- 1.92
	50	57.07 +/- 2.21
Baseline-Chen (with data)	1	41.96 +/- 1.95
	5	60.58 +/- 2.11
	50	75.4 +/- 0.71
Best teacher	1	35.6 +/- 5.21
	5	51.96 +/- 1.45
	50	70.76 +/- 3.15
FOMAML	1	24.27 +/- 2.09
	5	27.49 +/- 1.05
	50	29.2 +/- 2.94
FOMAML (with data)	1	27.13 +/- 2.76
	5	41.8 +/- 5.1
	50	51.89 +/- 3.38
ProtoNet	1	27.91 +/- 0.55
	5	37.36 +/- 2.3
	50	46.22 +/- 5.72
ProtoNet (with data)	1	27.18 +/- 2.51
	5	44.07 +/- 4.82
	50	60.93 +/- 0.85
Random initialization	1	29.31 +/- 2.71
	5	43.07 +/- 1.42
	50	62.29 +/- 1.28
Teacher concatenation	1	35.07 +/- 4.33
	5	54.51 +/- 2.12
	50	72.67 +/- 1.85

A.3.2. ResNet-10 teachers

2000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	30.4 +/- 2.47
	5	42.71 +/- 1.92
	50	57.07 +/- 2.21
Baseline-Chen (with data)	1	41.96 +/- 1.95
	5	60.58 +/- 2.11
	50	75.4 +/- 0.71
Best teacher	1	30.36 +/- 1.66
	5	42.47 +/- 2.17
	50	55.82 +/- 1.96
FOMAML	1	25.02 +/- 2.86
	5	30.64 +/- 4.38
	50	33.71 +/- 2.2
FOMAML (with data)	1	27.13 +/- 2.76
	5	41.8 +/- 5.1
	50	51.89 +/- 3.38
ProtoNet	1	28.47 +/- 3.49
	5	38.89 +/- 2.13
	50	48.91 +/- 5.19
ProtoNet (with data)	1	27.18 +/- 2.51
	5	44.07 +/- 4.82
	50	60.93 +/- 0.85
Random initialization	1	27.24 +/- 2.24
	5	37.82 +/- 2.83
	50	52.98 +/- 1.58
Teacher concatenation	1	32.89 +/- 4.42
	5	46.96 +/- 0.82
	50	49.38 +/- 2.2

8000 train images per teacher

algorithm	shots	accuracy (%)
Baseline-Chen	1	30.4 +/- 2.47
	5	42.71 +/- 1.92
	50	57.07 +/- 2.21
Baseline-Chen (with data)	1	41.96 +/- 1.95
	5	60.58 +/- 2.11
	50	75.4 +/- 0.71
Best teacher	1	30.09 +/- 3.08
	5	42.91 +/- 1.4
	50	60.71 +/- 2.31
FOMAML	1	25.02 +/- 2.86
	5	30.62 +/- 4.43
	50	33.71 +/- 2.2
FOMAML (with data)	1	27.13 +/- 2.76
	5	41.8 +/- 5.1
	50	51.89 +/- 3.38
ProtoNet	1	27.47 +/- 0.65
	5	37.47 +/- 3.3
	50	46.13 +/- 4.54
ProtoNet (with data)	1	27.18 +/- 2.51
	5	44.07 +/- 4.82
	50	60.93 +/- 0.85
Random initialization	1	27.02 +/- 2.24
	5	38.13 +/- 3.27
	50	51.4 +/- 2.32
Teacher concatenation	1	38.0 +/- 5.03
	5	52.89 +/- 1.15
	50	65.47 +/- 0.86

B. Experiments with training data available and more computational resources

To give a hint of how the computational budget during the hyperparameter tuning affected the meta-learning algorithm results, they have been evaluated once more for 5-shot 5-way MiniImagenet tasks with more costly tuning parameters. The adjusted tuning parameters in the original MiniImagenet experiments are

- "meta batches between validation evaluations" = 20
- "meta batches for a single validation evaluation" = 10
- "meta batches for meta testing evaluation" = 20
- "meta batch size" = 1
- "early stopping patience" = 3

and in the rerun with more computational budget

- "meta batches between validation evaluations" = 100
- "meta batches for a single validation evaluation" = 100
- "meta batches for meta testing evaluation" = 100
- "meta batch size" = 4
- "early stopping patience" = 10

.

The number of batches seems only to influence the variance of the losses, which can be balanced by higher early stopping patience. So, the only critical values remaining are the meta batch size, which is set so low due to high memory requirements of the 50 shots tasks (exclusively setting the meta batch size higher for 1-shot and 5-shot tasks make a meaningful comparison of the few-shot cases to the many-shot case impossible), and the early stopping patience, which greatly increases the time until a hyperparameter tuning run is stopped. For Baseline-Chen, the batch size is kept unchanged, and the meta batch size has no influence. As shown in figure B.1, the influence of these settings on the performance of FOMAML and ProtoNet is large, so better accuracy values in the experiments can be expected by more access to GPU memory and computation time.

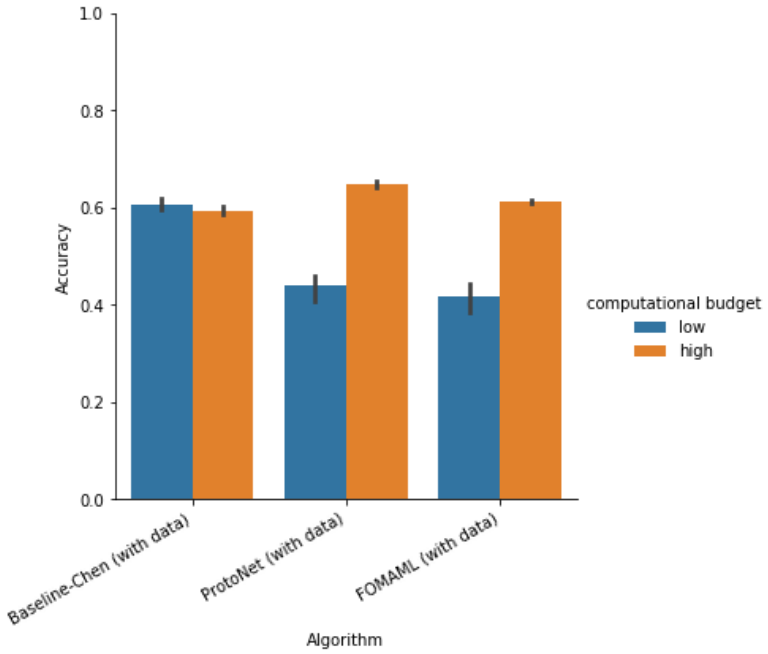


Figure B.1.: Average accuracy values of 5-shot 5-way MiniImagenet tasks of algorithms trained on the original MiniImagenet data but with different computational budget.

List of Figures

List of Tables

Bibliography

[1] I. 2017. Ilsvrc 2017. http://image-net.org/challenges/LSVRC/2017/results, 2017. abgerufen am 04.12.2020.

[2] F. Alet, M. F. Schneider, T. Lozano-Perez, and L. P. Kaelbling. Meta-learning curiosity algorithms, 2020.

[3] H. Altae-Tran, B. Ramsundar, A. S. Pappu, and V. Pande. Low data drug discovery with one-shot learning. *ACS Central Science*, 3(4):283–293, 2017. doi: 10.1021/acscentsci.6b00367. URL https://doi.org/10.1021/acscentsci.6b00367. PMID: 28470045.

[4] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24, pages 2546–2554. Curran Associates, Inc., 2011. URL https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf.

[5] L. Bertinetto, J. F. Henriques, P. H. S. Torr, and A. Vedaldi. Meta-learning with differentiable closed-form solvers. *CoRR*, abs/1805.08136, 2018. URL http://arxiv.org/abs/1805.08136.

[6] L. Bertinetto, J. F. Henriques, P. Torr, and A. Vedaldi. Meta-learning with differentiable closed-form solvers. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HyxnZh0ct7.

[7] B. Bischl, F. Scheipl, H. Seibold, L. Bothmann, D. Schalk, C. Molnar, and T. Pielok. Introduction to machine learning (i2ml), 2020. URL https://introduction-to-machine-learning.netlify.app.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[9] J. Bjorck, C. Gomes, B. Selman, and K. Q. Weinberger. Understanding batch normalization, 2018.

[10] H. Chen, Y. Wang, C. Xu, Z. Yang, C. Liu, B. Shi, C. Xu, C. Xu, and Q. Tian. Data-free learning of student networks. *CoRR*, abs/1904.01186, 2019. URL http://arxiv.org/abs/1904.01186.

[11] W. Chen, Y. Liu, Z. Kira, Y. F. Wang, and J. Huang. A closer look at few-shot classification. *CoRR*, abs/1904.04232, 2019. URL `http://arxiv.org/abs/1904.04232`.

[12] Y. Chen, X. Wang, Z. Liu, H. Xu, and T. Darrell. A new meta-baseline for few-shot learning, 2020.

[13] Y. Choi, J. Choi, M. El-Khamy, and J. Lee. Data-free network quantization with adversarial knowledge distillation, 2020.

[14] B. C. Csáji. Approximation with artificial neural networks. In *Approximation with Artificial Neural Networks*, 2001.

[15] J. Dean. The deep learning revolution and its implications for computer architecture and chip design. *CoRR*, abs/1911.05289, 2019. URL `http://arxiv.org/abs/1911.05289`.

[16] T. Deleu, T. Würfl, M. Samiei, J. P. Cohen, and Y. Bengio. Torchmeta: A Meta-Learning library for PyTorch, 2019. URL `https://arxiv.org/abs/1909.06576`. Available at: https://github.com/tristandeleu/pytorch-meta.

[17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[18] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL `http://arxiv.org/abs/1810.04805`.

[19] G. S. Dhillon, P. Chaudhari, A. Ravichandran, and S. Soatto. A baseline for few-shot image classification. *CoRR*, abs/1909.02729, 2019. URL `http://arxiv.org/abs/1909.02729`.

[20] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel. Rl$ˆ2$: Fast reinforcement learning via slow reinforcement learning. *CoRR*, abs/1611.02779, 2016. URL `http://arxiv.org/abs/1611.02779`.

[21] C. Edwards. Growing pains for deep learning. *Commun. ACM*, 58(7):14–16, June 2015. ISSN 0001-0782. doi: 10.1145/2771283. URL `http://doi.acm.org/10.1145/2771283`.

[22] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *CoRR*, abs/1703.03400, 2017. URL `http://arxiv.org/abs/1703.03400`.

[23] L. Franceschi, P. Frasconi, S. Salzo, R. Grazzi, and M. Pontil. Bilevel programming for hyperparameter optimization and meta-learning, 2018.

[24] P. I. Frazier. A tutorial on bayesian optimization, 2018.

[25] M. Goldblum, S. Reich, L. Fowl, R. Ni, V. Cherepanova, and T. Goldstein. Unraveling meta-learning: Understanding feature representations for few-shot tasks, 2020.

[26] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples, 2015.

[27] J. Gou, B. Yu, S. J. Maybank, and D. Tao. Knowledge distillation: A survey, 2020.

[28] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[29] D. R. Hardoon, S. Szedmak, and J. Shawe-Taylor. Canonical correlation analysis: An overview with application to learning methods. *Neural Computation*, 16(12): 2639–2664, 2004. doi: 10.1162/0899766042321814. URL `https://doi.org/10.1162/0899766042321814`.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

[31] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou. Deep learning scaling is predictable, empirically, 2017.

[32] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network, 2015.

[33] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: A survey, 2020.

[34] R. Houthooft, R. Y. Chen, P. Isola, B. C. Stadie, F. Wolski, J. Ho, and P. Abbeel. Evolved policy gradients. *CoRR*, abs/1802.04821, 2018. URL `http://arxiv.org/abs/1802.04821`.

[35] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely connected convolutional networks, 2018.

[36] A. C. Ian Goodfellow, Yoshua Bengio. *Deep Learning*. MIT Press, Cambridge, MA, 2016.

[37] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. V. Gool. AI benchmark: All about deep learning on smartphones in 2019. *CoRR*, abs/1910.06663, 2019. URL `http://arxiv.org/abs/1910.06663`.

[38] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

[39] K. G. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. *CoRR*, abs/1502.07943, 2015. URL `http://arxiv.org/abs/1502.07943`.

[40] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao. Advances and open problems in federated learning, 2019.

[41] L. Kaiser, O. Nachum, A. Roy, and S. Bengio. Learning to remember rare events. *CoRR*, abs/1703.03129, 2017. URL `http://arxiv.org/abs/1703.03129`.

[42] M. Kaya and H. Bilge. Deep metric learning: A survey. *Symmetry*, 11:1066, 08 2019. doi: 10.3390/sym11091066.

[43] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516, Apr 2020. ISSN 1573-7462. doi: 10.1007/s10462-020-09825-6. URL `http://dx.doi.org/10.1007/s10462-020-09825-6`.

[44] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[46] A. Krizhevsky et al. Learning multiple layers of features from tiny images. 2009.

[47] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[48] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016. URL `http://arxiv.org/abs/1603.06560`.

[49] L. Li, K. G. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht, and A. Talwalkar. Massively parallel hyperparameter tuning. *CoRR*, abs/1810.05934, 2018. URL `http://arxiv.org/abs/1810.05934`.

[50] Y. Li, S. Gu, L. V. Gool, and R. Timofte. Learning filter basis for convolutional neural network compression, 2019.

[51] Z. Li, F. Zhou, F. Chen, and H. Li. Meta-sgd: Learning to learn quickly for few shot learning. *CoRR*, abs/1707.09835, 2017. URL `http://arxiv.org/abs/1707.09835`.

[52] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search, 2019.

[53] R. G. Lopes, S. Fenu, and T. Starner. Data-free knowledge distillation for deep neural networks, 2017.

[54] S. Luo, X. Wang, G. Fang, Y. Hu, D. Tao, and M. Song. Knowledge amalgamation from heterogeneous networks by common feature learning. *CoRR*, abs/1906.10546, 2019. URL `http://arxiv.org/abs/1906.10546`.

[55] A. Mahendran and A. Vedaldi. Understanding deep image representations by inverting them. *CoRR*, abs/1412.0035, 2014. URL `http://arxiv.org/abs/1412.0035`.

[56] G. Marcus. Deep learning: A critical appraisal. *CoRR*, abs/1801.00631, 2018. URL `http://arxiv.org/abs/1801.00631`.

[57] E. K. Marek Capinski. *Measure, Integral and Probability*. Springer, London, UK, 2005.

[58] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016. URL `http://arxiv.org/abs/1602.05629`.

[59] L. Metz, N. Maheswaranathan, B. Cheung, and J. Sohl-Dickstein. Learning unsupervised learning rules. *CoRR*, abs/1804.00222, 2018. URL `http://arxiv.org/abs/1804.00222`.

[60] P. Micaelli and A. J. Storkey. Zero-shot knowledge transfer via adversarial belief matching. *CoRR*, abs/1905.09768, 2019. URL `http://arxiv.org/abs/1905.09768`.

[61] T. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997. ISBN 9780071154673. URL `https://books.google.de/books?id=EoYBngEACAAJ`.

[62] G. Montavon, W. Samek, and K. Müller. Methods for interpreting and understanding deep neural networks. *CoRR*, abs/1706.07979, 2017. URL `http://arxiv.org/abs/1706.07979`.

[63] A. Mordvintsev, C. Olah, and M. Tyka. Inceptionism: Going deeper into neural networks. *https://ai.googleblog.com/*, 2015. URL `https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html`.

[64] G. K. Nayak, K. R. Mopuri, V. Shaj, R. V. Babu, and A. Chakraborty. Zero-shot knowledge distillation in deep networks. *CoRR*, abs/1905.08114, 2019. URL http://arxiv.org/abs/1905.08114.

[65] A. Nichol, J. Achiam, and J. Schulman. On first-order meta-learning algorithms. *CoRR*, abs/1803.02999, 2018. URL http://arxiv.org/abs/1803.02999.

[66] Z. Ovaisi, R. Ahsan, Y. Zhang, K. Vasilaky, and E. Zheleva. Correcting for selection bias in learning-to-rank systems. *Proceedings of The Web Conference 2020*, Apr 2020. doi: 10.1145/3366423.3380255. URL http://dx.doi.org/10.1145/3366423.3380255.

[67] M. Patacchiola, J. Turner, E. J. Crowley, and A. J. Storkey. Deep kernel transfer in gaussian processes for few-shot learning. *CoRR*, abs/1910.05199, 2019. URL http://arxiv.org/abs/1910.05199.

[68] F. Pedregosa. Hyperparameter optimization with approximate gradient, 2016.

[69] J. Rafati and R. F. Marcia. Quasi-newton optimization methods for deep learning applications, 2019.

[70] A. Raghu, M. Raghu, S. Bengio, and O. Vinyals. Rapid learning or feature reuse? towards understanding the effectiveness of MAML. *CoRR*, abs/1909.09157, 2019. URL http://arxiv.org/abs/1909.09157.

[71] A. Rajeswaran, C. Finn, S. M. Kakade, and S. Levine. Meta-learning with implicit gradients. *CoRR*, abs/1909.04630, 2019. URL http://arxiv.org/abs/1909.04630.

[72] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search, 2019.

[73] P. Rodríguez, I. Laradji, A. Drouin, and A. Lacoste. Embedding propagation: Smoother manifold for few-shot classification, 2020.

[74] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge, 2015.

[75] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan 2015. ISSN 0893-6080. doi: 10.1016/j.neunet.2014.09.003. URL http://dx.doi.org/10.1016/j.neunet.2014.09.003.

[76] A. Shanbhag, S. Madden, and X. Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics (extended version), 2020.

[77] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484 EP –, Jan 2016. URL `http://dx.doi.org/10.1038/nature16961`. Article.

[78] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354 EP –, Oct 2017. URL `http://dx.doi.org/10.1038/nature24270`. Article.

[79] J. Snell, K. Swersky, and R. S. Zemel. Prototypical networks for few-shot learning. *CoRR*, abs/1703.05175, 2017. URL `http://arxiv.org/abs/1703.05175`.

[80] D. Soydaner. A comparison of optimization algorithms for deep learning. *International Journal of Pattern Recognition and Artificial Intelligence*, 34(13): 2052013, Apr 2020. ISSN 1793-6381. doi: 10.1142/s0218001420520138. URL `http://dx.doi.org/10.1142/S0218001420520138`.

[81] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

[82] S.-H. Sun. Multi-digit mnist for few-shot learning, 2019. URL `https://github.com/shaohua0116/MultiDigitMNIST`.

[83] F. Sung, Y. Yang, L. Zhang, T. Xiang, P. H. S. Torr, and T. M. Hospedales. Learning to compare: Relation network for few-shot learning. *CoRR*, abs/1711.06025, 2017. URL `http://arxiv.org/abs/1711.06025`.

[84] H. Thanh-Tung, T. Tran, and S. Venkatesh. On catastrophic forgetting and mode collapse in generative adversarial networks. *CoRR*, abs/1807.04015, 2018. URL `http://arxiv.org/abs/1807.04015`.

[85] S. Thrun and L. Pratt. *Learning to Learn: Introduction and Overview*, pages 3–17. Springer US, Boston, MA, 1998. ISBN 978-1-4615-5529-2. doi: 10.1007/978-1-4615-5529-2_1. URL `https://doi.org/10.1007/978-1-4615-5529-2_1`.

[86] Y. Tian, D. Krishnan, and P. Isola. Contrastive representation distillation. *CoRR*, abs/1910.10699, 2019. URL `http://arxiv.org/abs/1910.10699`.

[87] A. Triastcyn and B. Faltings. Generating artificial data for private deep learning, 2019.

[88] M. Vartak, A. Thiagarajan, C. Miranda, J. Bratman, and H. Larochelle. A meta-learning perspective on cold-start recommendations for items. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 6904–6914. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper/2017/file/51e6d6e679953c6311757004d8cbbba9-Paper.pdf`.

[89] O. Vinyals, C. Blundell, T. P. Lillicrap, K. Kavukcuoglu, and D. Wierstra. Matching networks for one shot learning. *CoRR*, abs/1606.04080, 2016. URL `http://arxiv.org/abs/1606.04080`.

[90] Y. Wang and Q. Yao. Few-shot learning: A survey. *CoRR*, abs/1904.05046, 2019. URL `http://arxiv.org/abs/1904.05046`.

[91] L. Weng. Meta-learning: Learning to learn fast. *lilianweng.github.io/lil-log*, 2018. URL `http://lilianweng.github.io/lil-log/2018/11/29/meta-learning.html`.

[92] L. Yang and A. Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, Nov 2020. ISSN 0925-2312. doi: 10.1016/j.neucom.2020.07.061. URL `http://dx.doi.org/10.1016/j.neucom.2020.07.061`.

[93] J. Ye, Y. Ji, X. Wang, K. Ou, D. Tao, and M. Song. Student becoming the master: Knowledge amalgamation for joint scene parsing, depth estimation, and more. *CoRR*, abs/1904.10167, 2019. URL `http://arxiv.org/abs/1904.10167`.

[94] J. Ye, Y. Ji, X. Wang, X. Gao, and M. Song. Data-free knowledge amalgamation via group-stack dual-gan, 2020.

[95] C. Yin, J. Tang, Z. Xu, and Y. Wang. Adversarial meta-learning. *CoRR*, abs/1806.03316, 2018. URL `http://arxiv.org/abs/1806.03316`.

[96] H. Yin, P. Molchanov, Z. Li, J. M. Alvarez, A. Mallya, D. Hoiem, N. K. Jha, and J. Kautz. Dreaming to distill: Data-free knowledge transfer via deepinversion, 2020.

[97] G. A. Young and R. L. Smith. *Essentials of Statistical Inference*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2005. doi: 10.1017/CBO9780511755392.

[98] M. Zhang, D. Wang, and S. Gai. Knowledge distillation for model-agnostic meta-learning. In *ECAI*, 2020.

[99] R. Zhang, T. Che, Z. Ghahramani, Y. Bengio, and Y. Song. Metagan: An adversarial approach to few-shot learning. In *Advances in Neural Information Processing Systems*, pages 2365–2374, 2018.

[100] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He. A comprehensive survey on transfer learning, 2020.

[101] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning, 2017.