Understanding deep meta-learning

Huisman, M.

Understanding Deep Meta-Learning

Proefschrift

ter verkrijging van

de graad van doctor aan de Universiteit Leiden,

op gezag van rector magnificus prof.dr.ir. H. Bijl,

volgens besluit van het college voor promoties

te verdedigen op woensdag 17 januari 2024

klokke 10:00 uur

door

Mike Huisman

geboren te Delft

in 1997

Promotor:

Prof.dr. A. Plaat

Co-promotor:

Dr. J. N. van Rijn

Promotiecommissie:

Prof.dr. H. C. M. Kleijn

Prof.dr. K. J. Batenburg

Prof.dr. E. O. Postma (Universiteit van Tilburg)

Dr. T. M. Moerland

Dr. S. Magliacane (Universiteit van Amsterdam)

Universiteit
Leiden
The Netherlands

This work was fully funded by Leiden University. SIKS Dissertation Series No. 2024-04. The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 978-94-93330-43-6

Contents

Chapter 1

Introduction

Chapter overview

The invention of neural networks marks a critical milestone in the pursuit of true artificial intelligence. However, despite their impressive performance on various tasks, deep neural networks face limitations in learning efficiently. This chapter introduces the concept of *deep meta-learning*, an approach aimed at enhancing learning efficiency in neural networks. Despite the development of numerous new deep meta-learning algorithms, we identify that there remains a research gap in understanding their performance. We motivate the importance of addressing this research gap and make steps towards filling this in the remainder of the dissertation. Moreover, this chapter outlines the specific research questions explored in this dissertation and provides an overview of its structure.

1.1 Background

Artificial deep neural networks have demonstrated remarkable capabilities, achieving human-level or even super-human performance across various tasks (Krizhevsky et al., 2012; Silver et al., 2016; Brown et al., 2020). The performance of these networks, however, hinges on the availability of large volumes of training data, often necessitating access to high-end computational resources (LeCun et al., 2015). Without sufficient data, these networks struggle to capture the intricacies of the underlying task, leading to suboptimal performance and limited generalization. This data dependency and computational demand pose significant challenges, particularly in domains where data collection is arduous, expensive, or privacy-sensitive. Overcoming these limitations and improving the data efficiency of deep neural networks is crucial to unlocking their full potential, enabling them to learn and adapt from limited data, and facilitating their deployment in resource-constrained environments, thereby democratizing the use of deep learning.

The learning efficiency of artificial deep neural networks is in stark contrast to human learning, which is highly efficient (Lake et al., 2015; Salakhutdinov et al., 2012; Schmidt, 2009). That is, humans can learn new concepts from only a handful of examples or a limited amount of experience. For instance, if shown a picture of a unique animal species that they have never encountered before, humans can often grasp its distinguishing features and correctly identify similar instances with high accuracy.

In contrast, deep neural networks typically struggle to achieve comparable performance with such limited data.

One potential explanation for this gap in learning efficiency between human learning and deep learning is that humans can rely upon a large set of prior knowledge and experience to quickly learn to perform new tasks (Jankowski et al., 2011; Lake et al., 2017). For example, humans often first learn to crawl and walk before they learn how to run. This allows us to transfer our movement patterns and knowledge from crawling and walking to more quickly learn running than would be possible if we were directly to attempt to learn how to run. Coming back to the example of learning to identify a new animal species, humans can leverage their understanding of the visual world and feature detectors from all prior experiences to quickly identify new species from only a handful of examples.

Deep neural networks, in contrast to humans, are often characterized by their high degree of specialization towards specific tasks. They typically undergo end-to-end training, meaning they learn directly from available data without any pre-existing knowledge or prior experience. This lack of prior knowledge places a considerable burden on deep neural networks, as they essentially attempt to master complex tasks without having learned foundational skills or basic concepts. In a metaphorical sense, it is as if these networks are trying to run before learning to crawl or walk. In a more practical sense, this absence of previously learned movement patterns and task-specific knowledge requires deep neural networks to rely solely on the limited data provided during training, making the learning process more challenging and time-consuming. In order to overcome this limitation, deep neural networks would greatly benefit from the ability to leverage prior experience and existing data that could be harnessed to facilitate the learning of the given task at hand. By incorporating such prior knowledge, these networks could potentially enhance their learning efficiency, accelerate the training process, and achieve higher performance with limited data.

Deep meta-learning (Schmidhuber, 1987; Thrun, 1998; Naik and Mammone, 1992) is the field of research that aims to endow deep neural networks with the ability to reuse a large set of prior experiences in order to learn to perform new tasks more efficiently. Since any prior knowledge in neural networks is encoded in the hyperparameters (such as the initialization parameters when learning a new task, the learning rate of stochastic gradient descent, or the weight update rule), the goal of deep meta-learning can be formulated as extracting a good set of hyperparameters from a set of prior tasks such that new tasks can be learned more efficiently than a neural network without such prior knowledge. We note that this is a form of transfer learning (Pan and Yang, 2009; Taylor and Stone, 2009) as we transfer knowledge obtained from previous tasks to a new task in which we are interested.

A common testbed for deep meta-learning algorithms that we also employ throughout this dissertation is few-shot learning (Wang et al., 2020b; Ravi and Larochelle, 2017; Vinyals et al., 2016), where the neural network has to learn different tasks only from a handful of examples, or *shots* per class. In few-shot learning scenarios, deep neural networks are presented with a set of classes or categories, each accompanied by a limited number of labeled instances (shots). The objective is to enable the network to generalize from this sparse data and quickly adapt to new classes with only a few examples. This setting closely emulates the remarkable ability of humans to rapidly acquire new knowledge and skills from minimal exposure and allows us to assess the ability of the networks to reuse prior knowledge in order to quickly learn to perform new tasks.

1.2 Motivation

The concept of meta-learning with neural networks was pioneered in the 1980s - 2000s (Schmidhuber, 1987; Bengio et al., 1991; Hochreiter et al., 2001; Thrun, 1998; Naik and Mammone, 1992), but it is only in recent years that the field has experienced a surge in popularity, driven by promising results presented by Vinyals et al. (2016), Finn et al. (2017), and other researchers. This increased attention has led to the introduction of numerous novel techniques, resulting in significant advances in the few-shot learning performance of these methods. However, despite these achievements, the reasons behind why certain meta-learning algorithms outperform others often remain elusive. The underlying principles that determine the success of different meta-learning approaches have received limited attention and are, therefore, not well understood. This knowledge gap hampers our ability to make informed design choices and improve deep meta-learning algorithms.

It is crucial to address this research gap to gain knowledge about meta-learning algorithms beyond simple performance comparisons, such as method A outperforming method B. By delving deeper into the underlying principles, we can gain a comprehensive understanding of why certain meta-learning algorithms succeed while others fall short. This deeper insight, in turn, could allow us to design enhanced deep meta-learning algorithms and reason about the expected impact of specific design decisions on the performance of different algorithms. In essence, it empowers us to move beyond empirical observations and establish a theoretical foundation for deep meta-learning, enabling more robust and effective algorithm development.

Furthermore, an equally important aspect that has received limited attention is investigating how theoretical principles can be leveraged to improve the performance of meta-learning algorithms. By bridging the gap between theory and practice, we can uncover valuable insights into how theoretical foundations can be practically applied to enhance the capabilities of deep meta-learning algorithms. Understanding the interplay between theoretical principles and algorithmic design choices can guide the development of more efficient and effective meta-learning techniques, fostering advancements in few-shot learning and other related fields.

In summary, advancing the understanding of deep meta-learning algorithms is a critical research direction. By uncovering the underlying principles that govern their success, we can gain valuable insights beyond simple performance comparisons. This knowledge not only facilitates the development of improved deep meta-learning algorithms but also provides a theoretical framework to reason about the influence of different design decisions on their performance. Additionally, exploring the integration of theoretical principles into practical algorithm development opens up exciting avenues to enhance the capabilities of deep meta-learning approaches. In this dissertation, we aim to address these research gaps, so that we can move the field forward even further, paving the way for more robust and principled meta-learning techniques with broader applicability and superior performance.

1.3 Research questions

This dissertation is a collection of research articles that we have produced following this line of research. In this section, we give an overview of the research questions that we address as well as their contextual framework and relationships.

1.3.1 Stateless neural meta-learning

MAML (Finn et al., 2017) and the meta-learner LSTM (Ravi and Larochelle, 2017) are two popular deep meta-learning techniques. Interestingly, in writing the overview of deep meta-learning (see Chapter 2), we intuitively realized that the meta-learner LSTM is a slightly more expressive algorithm than MAML. In this dissertation, we mathematically prove this intuition: the meta-learner LSTM encompasses and can learn everything that MAML is capable of learning, and even more. In other words, the meta-learner LSTM subsumes MAML. Despite this fact, MAML consistently outperforms the meta-learner LSTM in various tasks in terms of the few-shot learning ability. The reasons behind this performance gap remain unknown and require further investigation.

Understanding the factors contributing to the superior performance of MAML compared to the meta-learner LSTM is of utmost importance. By unraveling the mechanisms responsible for this discrepancy, we can gain valuable insights into the inner workings of these meta-learning techniques. This research question prompts us to empirically explore potential factors such as architectural differences, optimization strategies, or inherent biases that might favor MAML's performance. Investigating these factors could shed light on the subtle nuances that influence the success or failure of deep meta-learning algorithms, leading to more informed design choices and ultimately driving improvements in the field. In short, we aim to answer the following research question.

RQ1: What causes the performance gap between MAML and the meta-learner LSTM?

1.3.2 Pre-training vs gradient-based meta-learning

A related method for improving the learning efficiency of deep neural networks compared with the deep meta-learning algorithms (MAML, meta-learner LSTM) investigated in the previous research question, is transfer learning by means of pre-training and finetuning. The idea of this method is to train a neural network to perform a given source task for which abundant data is available. This is in contrast to how deep meta-learning techniques are often trained for few-shot learning settings as they are trained on various tasks for which only a handful of examples are available to learn from. When presented with a new task, the finetuning approach transfers the parameters obtained by training on the source task to the target task and consequently fine-tunes these parameters on this target task. During finetuning, only the head of the network is adjusted whilst the body of the network is kept frozen. In the case of image classification, on which we focus in this dissertation, this corresponds to reusing the same feature detectors and only adapting how the different features are linearly combined to produce class scores.

Whilst the previous research question allowed us to gain a deep understanding of the differences between two deep meta-learning algorithms (MAML and the meta-learner LSTM), which have been observed to be successful at few-shot learning in various scenarios, recent results (Chen et al., 2019; Tian et al., 2020; Mangla et al., 2020) suggest that when evaluated on tasks from a different data distribution than the one used for training, the simple pre-training and finetuning baseline may be more effective than more complicated popular meta-learning techniques such as MAML and Reptile (Nichol et al., 2018). This is surprising as the learning behavior of MAML was shown to mimic that of finetuning: both rely on reusing learned features (Raghu et al., 2020). This begs the question of what causes the observed performance differences between these approaches, giving rise to the following research question.

RQ2: How do the learning behaviors of finetuning, MAML, and Reptile differ from each other and how does this influence their ability to quickly learn to perform new tasks?

1.3.3 LSTMs for few-shot learning

The study of research question 1 (RQ1) prompted us to look deeper into using an LSTM for meta-learning. More specifically, the meta-learner LSTM uses an LSTM at the *meta-level*: to perform weight updates for a base-learner network. In 2001, however, Hochreiter et al. (2001) showed that an LSTM that operates at the *data level* trained with backpropagation across different tasks is capable of meta-learning. The LSTM operating at the data level is fed an entire training set, and predictions for query inputs are then conditioned on the resulting hidden state. Despite the promising results of this approach on small problems, and more recently, also on reinforcement learning problems (Duan et al., 2016; Wang et al., 2016), the approach has received little attention in the supervised few-shot learning setting. In an earlier work that has tested the LSTM (Santoro et al., 2016), the performance was observed to be inferior compared with other meta-learning algorithms such as MAML and the meta-learner LSTM. This is surprising, as an LSTM at the data level is a maximally expressive meta-learning algorithm (Finn and Levine, 2018). We revisit this approach and test it on modern few-shot learning benchmarks. The research question we investigate is the following.

RQ3: Are LSTMs good few-shot learners when evaluated on modern benchmarks?

1.3.4 Subspace adaptation prior

The previous research questions have all focused on attempting to distill knowledge from empirical results. This begs the question of whether the opposite direction can also yield fruitful results, that is, whether the integration of machine learning knowledge can be used to enhance the few-shot learning capabilities of neural networks. Inspired by classical machine learning knowledge that a more expressive model (with more parameters) is more prone to overfitting than one with fewer parameters, we hypothesize that it may be beneficial for deep meta-learning methods to adjust only a subset of parameters rather than adapting *all* parameters of trainable layers when learning new tasks. More specifically, we hypothesize that it is beneficial to learn which subsets of parameters to adjust in every layer, rather than only the final layer as is common in the pre-training and finetuning strategy. The idea behind this hypothesis that simply adapting all paramaters neglects potentially more efficient learning strategies for a given task distribution and may be susceptible to overfitting, especially in few-shot learning where tasks must be learned from a limited number of examples. In short, we aim to answer the following research question.

RQ4: Can the few-shot learning ability of deep neural networks be improved by meta-learning which subsets of parameters to adjust?

1.4 Outline of the dissertation

This dissertation constitutes a collection of research papers and every chapter corresponds to one paper.

In Chapter 2, we survey the field of deep meta-learning. This serves as a detailed introduction and overview of the field. In this chapter, we provide the reader with the theoretical foundation for

describing deep meta-learning algorithms, and we investigate and summarize key methods, which are categorized into i) metric-, ii) model-, and iii) optimization-based techniques. In addition, we identify the main open challenges, such as performance evaluations on heterogeneous benchmarks, and reduction of the computational costs of meta-learning. This chapter corresponds to the following published research article.

Huisman, M., van Rijn, J. N., & Plaat, A. (2021). A survey of deep meta-learning. Artificial Intelligence Review, 54(6), 4483-4541. Springer.

In Chapter 3, we investigate our first research question based on the observed performance gap between the meta-learner LSTM and MAML. We show that the reason for this surprising performance gap is related to second-order gradients. We construct a new algorithm (named TURTLE) to gain more insight into the importance of second-order gradients. TURTLE is simpler than the meta-learner LSTM yet more expressive than MAML and outperforms both techniques at few-shot sine wave regression and 50% of the tested image classification settings (without any additional hyperparameter tuning) and is competitive otherwise, at a computational cost that is comparable to second-order MAML. We find that second-order gradients also significantly increase the accuracy of the meta-learner LSTM. This chapter is based on the following published research article.

Huisman, M., Plaat, A., & van Rijn, J. N. (2022). Stateless neural meta-learning using second-order gradients. Machine Learning, 111(9), 3227-3244. Springer.

In Chapter 4, we investigate our second research question. More specifically, we investigate the observed performance differences between finetuning, MAML, and another meta-learning technique called Reptile, and show that MAML and Reptile specialize for fast adaptation in low-data regimes of similar data distribution as the one used for training. Our findings show that both the output layer and the noisy training conditions induced by data scarcity in the few-shot learning setting play important roles in facilitating this specialization for MAML. Lastly, we show that the pre-trained features as obtained by the finetuning baseline are more diverse and discriminative than those learned by MAML and Reptile. Due to this lack of diversity and distribution specialization, MAML and Reptile may fail to generalize to target tasks that are more distant to the observed training tasks whereas finetuning can fall back on the diversity of the learned features. This chapter is based on the following research articles.

Huisman, M., Plaat, A. & van Rijn, J. N. (2021). A preliminary study on the feature representations of transfer learning and gradient-based meta-learning techniques. In Fifth Workshop on Meta-Learning at the Conference on Neural Information Processing Systems.

Huisman, M., Plaat, A. & van Rijn, J. N. (2023). Understanding Transfer Learning and Gradient-Based Meta-Learning Techniques. Accepted for publication in Machine Learning. Springer.

In Chapter 5, we investigate our third research question. That is, we revisit the classical LSTM approach to deep meta-learning and show that surprisingly, LSTM outperforms the popular meta-learning technique MAML on a simple few-shot sine wave regression benchmark, but that LSTM, expectedly, falls short on more complex few-shot image classification benchmarks. We identify two potential causes and propose a new method called *Outer Product LSTM (OP-LSTM)* that resolves these issues and displays substantial performance gains over the plain LSTM. Compared to popular

meta-learning baselines, OP-LSTM yields competitive performance on within-domain few-shot image classification, and performs better in cross-domain settings by 0.5% to 1.9% in accuracy score. While these results alone do not set a new state-of-the-art, the advances of OP-LSTM are orthogonal to other advances in the field of meta-learning, yielding new insights in how LSTM work in image classification, allowing for a whole range of new research directions. This chapter is based on the following research article.

Huisman, M., Moerland, T. M., Plaat, A., & van Rijn, J. N. (2023). Are LSTMs good few-shot learners? Machine Learning, 112, 4635–4662. Springer.

In Chapter 6, we investigate our fourth and final research question. That is, we investigate whether the few-shot learning performance of neural networks can be improved by meta-learning which parameters to adjust. To investigate this, we propose *Subspace Adaptation Prior* (SAP), a novel gradient-based meta-learning algorithm that jointly learns good initialization parameters (prior knowledge) and layer-wise *parameter subspaces* in the form of operation subsets that should be adaptable. In this way, SAP can learn which operation subsets to adjust with gradient descent based on the underlying task distribution, simultaneously decreasing the risk of overfitting when learning new tasks. We demonstrate that this ability is helpful as SAP yields superior or competitive performance in few-shot image classification settings (gains between 0.1% and 3.9% in accuracy). Analysis of the learned subspaces demonstrates that low-dimensional operations often yield high activation strengths, indicating that they may be important for achieving good few-shot learning performance. This chapter is based on the following research article.

Huisman, M., Plaat, A., & van Rijn, J. N. (2023). Subspace Adaptation Prior for Few-Shot Learning. Machine Learning. Springer.

In the following chapter, we give a detailed overview of the field deep meta-learning, serving as a basis for the rest of this dissertation.

Chapter 2

Overview of Deep Meta-Learning

Chapter overview

Deep neural networks can achieve great successes when presented with large data sets and sufficient computational resources. However, their ability to learn new concepts *quickly* is limited. Meta-learning is one approach to address this issue, by enabling the network to learn how to learn. The field of *deep meta-learning* advances at great speed, but lacks a unified, in-depth overview of current techniques. With this work[1], we aim to bridge this gap. After providing the reader with a theoretical foundation, we investigate and summarize key methods, which are categorized into i) metric-, ii) model-, and iii) optimization-based techniques. In addition, we identify the main open challenges, such as performance evaluations on heterogeneous benchmarks, and reduction of the computational costs of meta-learning.

2.1 Introduction

In recent years, deep learning techniques have achieved remarkable successes on various tasks, including game-playing (Mnih et al., 2013; Silver et al., 2016), image recognition (Krizhevsky et al., 2012; He et al., 2015), and machine translation (Wu et al., 2016). Despite these advances, ample challenges remain to be solved, such as the large amounts of data and training that are needed to achieve good performance. These requirements severely constrain the ability of deep neural networks to learn new concepts quickly, one of the defining aspects of human intelligence (Jankowski et al., 2011; Lake et al., 2017).

Meta-learning has been suggested as one strategy to overcome this challenge (Naik and Mammone, 1992; Schmidhuber, 1987; Thrun, 1998). The key idea is that meta-learning agents improve their own learning ability over time, or equivalently, learn to learn. The learning process is primarily concerned with tasks (set of observations) and takes place at two different levels: an inner- and an outer-level. At the *inner-level*, a new task is presented, and the agent tries to quickly learn the associated concepts from the training observations. This quick adaptation is facilitated by knowledge

[1]This chapter is based on the following published research article: Huisman, M., van Rijn, J. N., & Plaat, A. (2021). A survey of deep meta-learning. Artificial Intelligence Review, 54(6), 4483-4541. Springer.

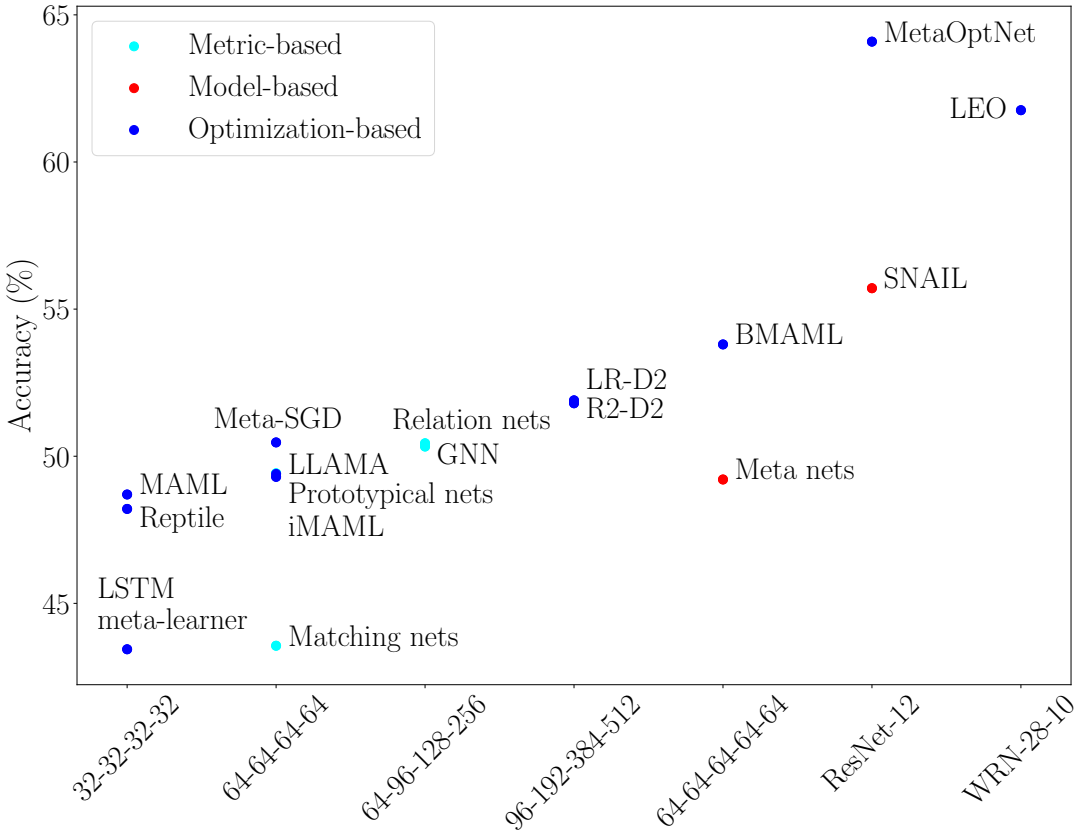


Figure 2.1: The accuracy scores of the covered techniques on 1-shot miniImageNet classification. The used feature extraction backbone is displayed on the x-axis. As one can see, there is a strong relationship between the network complexity and the classification performance.

that it has accumulated across earlier tasks at the *outer-level*. Thus, whereas the inner-level concerns a single task, the outer-level concerns a multitude of tasks.

Historically, the term meta-learning has been used with various scopes. In its broadest sense, it encapsulates all systems that leverage prior learning experience in order to learn new tasks more quickly (Vanschoren, 2018). This broad notion includes more traditional algorithm selection and hyperparameter optimization techniques for Machine Learning (Brazdil et al., 2008). In this chapter, however, we focus on a subset of the meta-learning field which develops meta-learning procedures to learn a good *inductive bias* for (deep) neural networks.[2] Henceforth, we use the term *deep meta-learning* to refer to this subfield of meta-learning.

The field of deep meta-learning is advancing at a quick pace, while it lacks a coherent, unifying overview, providing detailed insights into the key techniques. Vanschoren (2018) has surveyed meta-learning techniques, where meta-learning was used in the broad sense, limiting its account of deep meta-learning techniques. Also, many exciting developments in deep meta-learning have

[2]Here, inductive bias refers to the assumptions of a model which guide predictions on unseen data (Mitchell, 1980).

happened after the survey was published. A more recent survey by Hospedales et al. (2021) adopts the same notion of deep meta-learning as we do, but aims to give a broad overview, omitting technical details of the various techniques.

We attempt to fill this gap by providing detailed explications of contemporary deep meta-learning techniques, using a unified notation. More specifically, we cover modern techniques in the field for supervised and reinforcement learning, that have achieved state-of-the-art performance, obtained popularity in the field, and presented novel ideas. Extra attention is paid to MAML (Finn et al., 2017), and related techniques, because of their impact on the field. We show how the techniques relate to each other, detail their strengths and weaknesses, identify current challenges, and provide an overview of promising future research directions. One of the observations that we make is that the network complexity is highly related to the few-shot classification performance (see Figure 2.1). One might expect that in a few-shot setting, where only few examples are available to learn from, the number of network parameters should be kept small to prevent overfitting. Clearly, the figure shows that this does not hold, as techniques that use larger backbones tend to achieve better performance. One important factor might be that due to the large number of tasks that have been seen by the network, we are in a setting where similarly large amounts of observations have been evaluated. This result suggests that the size of the network should be taken into account when comparing algorithms.

This chapter can serve as educational introduction to the field of deep meta-learning, and as reference material for experienced researchers in the field. Throughout, we will adopt the taxonomy used by Vinyals (2017), which identifies three categories of deep meta-learning approaches: i) metric-, ii) model-, and iii) optimization-based meta-learning techniques.

The remainder of this chapter is structured as follows. Section 2 builds a common foundation on which we will base our overview of deep meta-learning techniques. Section 3, 4, and 5 cover the main metric-, model-, and optimization-based meta-learning techniques, respectively. Section 6 provides an overview of the field, and summarizes the key challenges and open questions. Table 2.1 gives an overview of notation that we will use throughout this paper.

2.2 Foundation

In this section, we build the necessary foundation for investigating deep meta-learning techniques in a consistent manner. To begin with, we contrast regular learning and meta-learning. Afterwards, we briefly discuss how deep meta-learning relates to different fields, what the usual training and evaluation procedure looks like, and which benchmarks are often used for this purpose. We finish this section by describing some applications and context of the meta-learning field.

2.2.1 The meta abstraction

In this subsection, we contrast base-level (regular) learning and meta-learning for two different paradigms, i.e., supervised and reinforcement learning.

Regular supervised learning

In *supervised learning*, we wish to learn a function $f_{\boldsymbol{\theta}} : X \to Y$ that learns to map inputs $\boldsymbol{x}_i \in X$ to their corresponding outputs $y_i \in Y$. Here, $\boldsymbol{\theta}$ are model parameters (e.g. weights in a neural network) that determine the function's behavior. To learn these parameters, we are given a data set of m

Expression	Meaning
Meta-learning	Learning to learn
$\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j})$	A task consisting of a labeled support and query set
Support set	The train set $D^{tr}_{\mathcal{T}_j}$ associated with a task $\mathcal{T}_j$
Query set	The test set $D^{test}_{\mathcal{T}_j}$ associated with a task $\mathcal{T}_j$
$\boldsymbol{x}_i$	Example input vector i in the support set
y_i	(One-hot encoded) label of example input $\boldsymbol{x}_i$ from the support set
k	Number of examples per class in the support set
N	Number of classes in the support and query sets of a task
$\boldsymbol{x}$	Input in the query set
y	A (one-hot encoded) label for input $\boldsymbol{x}$
$(f/g/h)_\circ$	Neural network function with parameters $\circ$
Inner-level	At the level of a single task
Outer-level	At the meta-level: across tasks
Fast weights	A term used in the literature to denote task-specific parameters
Base-learner	Learner that works at the inner-level
Meta-learner	Learner that operates at the outer-level
$\boldsymbol{\theta}$	The parameters of the base-learner network
$\mathcal{L}_D$	Loss function with respect to task/dataset D
Input embedding	Penultimate layer representation of the input
Task embedding	An internal representation of a task in a network/system
SL	Supervised Learning
RL	Reinforcement Learning

Table 2.1: Some notation and meaning, which we use throughout this chapter.

observations: $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m$. Thus, given a data set $\mathcal{D}$, learning boils down to finding the correct setting for $\boldsymbol{\theta}$ that minimizes an empirical loss function $\mathcal{L}_D$, which must capture how the model is performing, such that appropriate adjustments to its parameters can be made. In short, we wish to find

$$\boldsymbol{\theta}_{SL} := \arg\min_{\boldsymbol{\theta}} \mathcal{L}_D(\boldsymbol{\theta}), \tag{2.1}$$

where SL stands for "supervised learning". Note that this objective is specific to data set $\mathcal{D}$, meaning that our model $f_{\boldsymbol{\theta}}$ may not *generalize* to examples outside of $\mathcal{D}$. To measure generalization, one could evaluate the performance on a separate test data set, which contains unseen examples. A popular way to do this is through *cross-validation*, where one repeatedly creates train and test splits $D^{tr}, D^{test} \subset D$ and uses these to train and evaluate a model respectively (Hastie et al., 2009).

Finding globally optimal parameters $\boldsymbol{\theta}_{SL}$ is often computationally infeasible. We can, however, approximate them, guided by *pre-defined* meta-knowledge ω (Hospedales et al., 2021), which includes, e.g., the initial model parameters $\boldsymbol{\theta}$, choice of optimizer, and learning rate schedule. As such, we approximate

$$\boldsymbol{\theta}_{SL} \approx g_\omega(D, \mathcal{L}_D), \tag{2.2}$$

where g_ω is an optimization procedure that uses *pre-defined* meta-knowledge ω, data set $\mathcal{D}$, and loss

function $\mathcal{L}_D$, to produce updated weights $g_\omega(D, \mathcal{L}_D)$ that (presumably) perform well on $\mathcal{D}$.

Supervised meta-learning

In contrast, *supervised meta-learning* does not assume that any meta-knowledge ω is given, or predefined. Instead, the goal of meta-learning is to find the best ω, such that our (regular) base-learner can learn new *tasks* (data sets) as quickly as possible. Thus, whereas supervised regular learning involves one data set, supervised meta-learning involves a group of data sets. The goal is to learn meta-knowledge ω such that our model can learn many different tasks well. Thus, our model is learning to learn.

More formally, we have a probability distribution of tasks $p(\mathcal{T})$, and wish to find optimal meta-knowledge

$$\omega^* := \arg\min_\omega \underbrace{\mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}}_{\text{Outer-level}} [\underbrace{\mathcal{L}_{\mathcal{T}_j}(g_\omega(\mathcal{T}_j, \mathcal{L}_{\mathcal{T}_j}))}_{\text{Inner-level}}]. \tag{2.3}$$

Here, the inner-level concerns task-specific learning, while the outer-level concerns multiple tasks. One can now easily see why this is meta-learning: we learn ω, which allows for quick learning of tasks $\mathcal{T}_j$ at the inner-level. Hence, we are learning to learn.

Regular reinforcement learning

In *reinforcement learning*, we have an agent that learns from experience. That is, it interacts with an environment, modeled by a Markov Decision Process (MDP) $M = (S, A, P, r, p_0, \gamma, T)$. Here, S is the set of states, A the set of actions, P the transition probability distribution defining $P(s_{t+1}|s_t, a_t)$, $r : S \times A \to \mathbb{R}$ the reward function, p_0 the probability distribution over initial states, $\gamma \in [0, 1]$ the discount factor, and T the time horizon (maximum number of time steps) (Sutton and Barto, 2018; Duan et al., 2016).

At every time step t, the agent finds itself in state s_t, in which the agent performs an action a_t, computed by a policy function $\pi_{\boldsymbol{\theta}}$ (i.e., $a_t = \pi_{\boldsymbol{\theta}}(s_t)$), which is parameterized by weights $\boldsymbol{\theta}$. In turn, it receives a reward $r_t = r(s_t, \pi_{\boldsymbol{\theta}}(s_t)) \in \mathbb{R}$ and a new state s_{t+1}. This process of interactions continues until a termination criterion is met (e.g. fixed time horizon T reached). The goal of the agent is to learn how to act in order to maximize its expected reward. The reinforcement learning (RL) goal is to find

$$\boldsymbol{\theta}_{RL} := \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\text{traj}} \sum_{t=0}^{T} \gamma^t r(s_t, \pi_{\boldsymbol{\theta}}(s_t)), \tag{2.4}$$

where we take the expectation over the possible *trajectories* $\text{traj} = (s_0, \pi_{\boldsymbol{\theta}}(s_0), ...s_T, \pi_{\boldsymbol{\theta}}(s_T))$ due to the random nature of MDPs (Duan et al., 2016). Note that γ is a hyperparameter that can prioritize short- or long-term rewards by decreasing or increasing it, respectively.

Also in case of reinforcement learning it is often infeasible to find the global optimum $\boldsymbol{\theta}_{RL}$, and thus we settle for approximations. In short, given a learning method ω, we approximate

$$\boldsymbol{\theta}_{RL} \approx g_\omega(\mathcal{T}_j, \mathcal{L}_{\mathcal{T}_j}), \tag{2.5}$$

where again $\mathcal{T}_j$ is the given MDP, and g_ω is the optimization algorithm, guided by pre-defined meta-knowledge ω.

Note that in a Markov Decision Process (MDP), the agent knows the state at any given time step t. When this is not the case, it becomes a Partially Observable Markov Decision Process (POMDP), where the agent receives only observations O, and uses these to update its belief with regard to the state it is in (Sutton and Barto, 2018).

Meta reinforcement learning

The meta abstraction has as its object a group of tasks, or Markov Decision Processes (MDPs) in the case of reinforcement learning. Thus, instead of maximizing the expected reward on a single MDP, the meta reinforcement learning objective is to maximize the expected reward over various MDPs, by learning meta-knowledge ω. Here, the MDPs are sampled from some distribution $p(\mathcal{T})$. So, we wish to find a set of parameters

$$\boldsymbol{\omega}^* := \arg\min_{\boldsymbol{\omega}} \underbrace{\mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}}_{\text{Outer-level}} \left[\underbrace{\mathbb{E}_{traj} \sum_{t=0}^{T} \gamma^t r(s_t, \pi_{g_\omega(\mathcal{T}_j, \mathcal{L}_{\mathcal{T}_j})}(s_t))}_{\text{Inner-level}} \right]. \tag{2.6}$$

Contrast with other fields

Now that we have provided a formal basis for our discussion for both supervised and reinforcement meta-learning, it is time to contrast meta-learning briefly with two related areas of machine learning that also have the goal to improve the speed of learning. We will start with transfer learning.

Transfer Learning In Transfer Learning, one tries to *transfer* knowledge of previous tasks to new, unseen tasks (Pan and Yang, 2009; Taylor and Stone, 2009), which can be challenging when new task comes from a different distribution than the one used for training Iqbal et al. (2018). The distinction between Transfer Learning and Meta-Learning has become more opaque over time. A key property of meta-learning techniques, however, is their *meta-objective*, which explicitly aims to optimize performance across a distribution over tasks (as seen in previous sections by taking the expected loss over a distribution of tasks). This objective need not always be present in Transfer Learning techniques, e.g., when one *pre-trains* a model on a large data set, and *fine-tunes* the learned weights on a smaller data set.

Multi-task learning Another, closely related field, is that of multi-task learning. In multi-task learning a model is jointly trained to perform well on multiple fixed tasks (Hospedales et al., 2021). Meta-learning, in contrast, aims to find a model that can learn new (previously unseen) tasks quickly. This difference is illustrated in Figure 2.2.

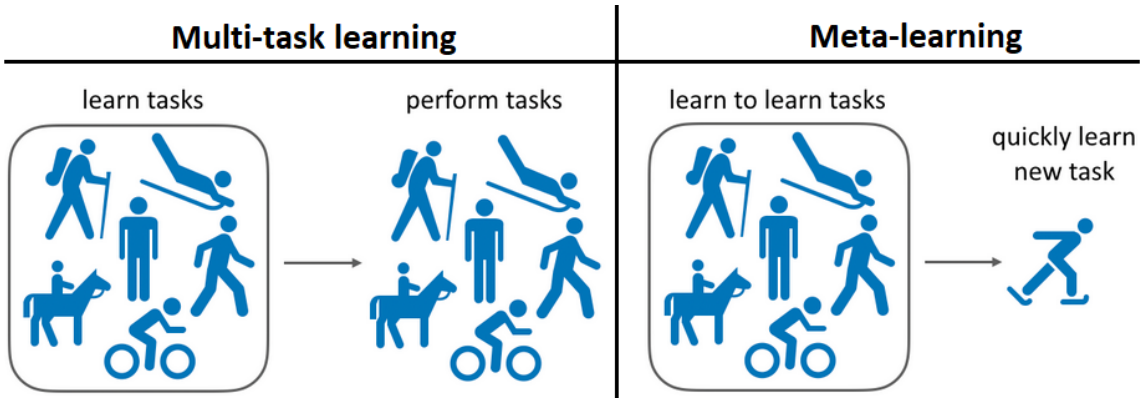


Figure 2.2: The difference between multi-task learning and meta-learning[3].

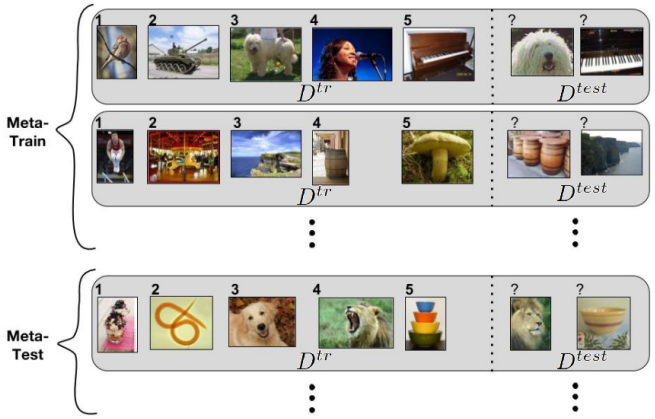


Figure 2.3: Illustration of N-way, k-shot classification, where $N = 5$, and $k = 1$. Meta-validation tasks are not displayed. Adapted from Ravi and Larochelle (2017).

2.2.2 The meta-setup

In the previous section, we have described the learning objectives for (meta) supervised and reinforcement learning. We will now describe the general setting that can be used to achieve these objectives. In general, one optimizes a meta-objective by using various tasks, which are data sets in the context of supervised learning, and (Partially Observable) Markov Decision Processes in case of reinforcement learning. This is done in three stages: the i) *meta-train* stage, ii) *meta-validation* stage, and iii) *meta-test* stage, each of which is associated with a set of tasks.

First, in the meta-train stage, the meta-learning algorithm is applied to the meta-train tasks. Second, the meta-validation tasks can then be used to evaluate the performance on unseen tasks, which were not used for training. Effectively, this measures the *meta-generalization* ability of the trained network, which serves as feedback to tune, e.g., hyper-parameters of the meta-learning algorithm. Third, the meta-test tasks are used to give a final performance estimate of the meta-learning technique.

[3]Adapted from `https://meta-world.github.io/`

N-way, k-shot learning

A frequently used instantiation of this general meta-setup is called N-way, k-shot classification (see Figure 2.3). This setup is also divided into the three stages—meta-train, meta-validation, and meta-test—which are used for meta-learning, meta-learner hyperparameter optimization, and evaluation, respectively. Each stage has a corresponding set of disjoint labels, i.e., $L^{tr}, L^{val}, L^{test} \subset Y$, such that $L^{tr} \cap L^{val} = \emptyset, L^{tr} \cap L^{test} = \emptyset$, and $L^{val} \cap L^{test} = \emptyset$. In a given stage s, *tasks/episodes* $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j})$ are obtained by sampling examples $(\boldsymbol{x}_i, y_i)$ from the full data set $\mathcal{D}$, such that every $y_i \in L^s$. Note that this requires access to a data set $\mathcal{D}$. The sampling process is guided by the N-way, k-shot principle, which states that every training data set $D^{tr}_{\mathcal{T}_j}$ should contain exactly N classes and k examples per class, implying that $|D^{tr}_{\mathcal{T}_j}| = N \cdot k$. Furthermore, the true labels of examples in the test set $D^{test}_{\mathcal{T}_j}$ must be present in the train set $D^{tr}_{\mathcal{T}_j}$ of a given task $\mathcal{T}_j$. $D^{tr}_{\mathcal{T}j}$ acts as a *support set*, literally supporting classification decisions on the *query set* $D^{test}_{\mathcal{T}_j}$. Importantly, note that with this terminology, the query set (or test set) of a task is actually used during the meta-training phase. Furthermore, the fact that the labels across stages are disjoint ensures that we test the ability of a model to learn *new* concepts.

The meta-learning objective in the training phase is to minimize the loss function of the model predictions on the query sets, conditioned on the support sets. As such, for a given task $\mathcal{T}_j$, the model 'sees' the support set, and extracts information from the support set to guide its predictions on the query set. By applying this procedure to different episodes/tasks $\mathcal{T}_j$, the model will slowly accumulate meta-knowledge ω, which can ultimately speed up learning on new tasks.

The easiest way to achieve this is by doing this with regular neural networks, but as was pointed out by various authors (see, e.g., Finn et al. (2017)) more sophisticated architectures will vastly outperform such network. In the remainder of this chapter, we will review such architectures.

At the meta-validation and meta-test stages, or evaluation phases, the learned meta-information in ω is fixed. The model is, however, still allowed to make task-specific updates to its parameters $\boldsymbol{\theta}$ (which implies that it is learning). After task-specific updates, we can evaluate the performance on the test sets. In this way, we test how well a technique performs at meta-learning.

N-way, k-shot classification is often performed for small values of k (since we want our models to learn new concepts quickly, i.e., from few examples). In that case, one can refer to it as *few-shot learning*.

Common benchmarks

Here, we briefly describe some benchmarks that can be used to evaluate meta-learning algorithms.

- **Omniglot (Lake et al., 2011):** This data set presents an image recognition task. Each image corresponds to one out of 1 623 characters from 50 different alphabets. Every character was drawn by 20 people. Note that in this case, the characters are the classes/labels.
- **ImageNet (Deng et al., 2009):** This is the largest image classification data set, containing more than 20K classes and over 14 million colored images. *miniImageNet* is a mini variant of the large ImageNet data set (Deng et al., 2009) for image classification, proposed by Vinyals et al. (2016) to reduce the engineering efforts to run experiments. The mini data set contains 60 000 colored images of size 84×84. There are a total of 100 classes present, each accorded by 600 examples. *tieredImageNet* (Ren et al., 2018) is another variation of the large ImageNet

data set. It is similar to miniImageNet, but contains a hierarchical structure. That is, there are 34 classes, each with its own sub-classes.

- **CIFAR-10 and CIFAR-100 (Krizhevsky, 2009)**: Two other image recognition data sets. Each one contains 60K RGB images of size 32×32. CIFAR-10 and CIFAR-100 contain 10 and 100 classes respectively, with a uniform number of examples per class (6 000 and 600 respectively). Every class in CIFAR-100 also has a super-class, of which there are 20 in the full data set. Many variants of the CIFAR data sets can be sampled, giving rise to e.g. *CIFAR-FS* (Bertinetto et al., 2019) and *FC-100* (Oreshkin et al., 2018).
- **CUB-200-2011 (Wah et al., 2011):** The CUB-200-2011 data set contains roughly 12K RGB images of birds from 200 species. Every image has some labeled attributes (e.g. crown color, tail shape).
- **MNIST (LeCun et al., 2010):** MNIST presents a hand-written digit recognition task, containing ten classes (for digits 0 through 9). In total, the data set is split into a 60K train and 10K test gray scale images of hand-written digits.
- **Meta-Dataset (Triantafillou et al., 2020):** This data set comprises several other data sets such as Omniglot (Lake et al., 2011), CUB-200 (Wah et al., 2011), ImageNet (Deng et al., 2009), and more (Triantafillou et al., 2020). An episode is then constructed by sampling a data set (e.g. Omniglot) and selecting a subset of labels to create train and test splits as before. In this way, broader generalization is enforced since the tasks are more distant from each other.
- **Meta-world (Yu et al., 2019)**: A meta reinforcement learning data set, containing 50 robotic manipulation tasks (control a robot arm to achieve some pre-defined goal, e.g. unlocking a door, or playing soccer). It was specifically designed to cover a broad range of tasks, such that meaningful generalization can be measured (Yu et al., 2019).

Some applications of meta-learning

Deep neural networks have achieved remarkable results on various tasks including image recognition, text processing, game playing, and robotics (Silver et al., 2016; Mnih et al., 2013; Wu et al., 2016), but their success depends on the amount of available data (Sun et al., 2017) and computing resources. Deep meta-learning reduces this dependency by allowing deep neural nets to learn new concepts quickly. As a result, meta-learning widens the applicability of deep learning techniques to many application domains. Such areas include few-shot image classification (Finn et al., 2017; Snell et al., 2017; Ravi and Larochelle, 2017), robotic control policy learning (Gupta et al., 2018; Clavera et al., 2019) (see Figure 2.4), hyperparameter optimization (Antoniou et al., 2019; Schmidhuber et al., 1997), meta-learning learning rules (Bengio et al., 1991, 1997; Miconi et al., 2018, 2019), abstract reasoning (Barrett et al., 2018), and many more. For a larger overview of applications, we refer interested readers to Hospedales et al. (2021).

2.2.3 The meta-learning field

As mentioned in the introduction, meta-learning is a broad area of research, as it encapsulates all techniques that leverage prior learning experience to learn new tasks more quickly (Vanschoren, 2018). We can classify two distinct communities in the field with a different focus: i) algorithm selection and hyperparameter optimization for machine learning techniques, and ii) search for inductive bias in deep neural networks. We will refer to these communities as group i) and group ii) respectively. Now, we will give a brief description of the first field, and a historical overview of the second.

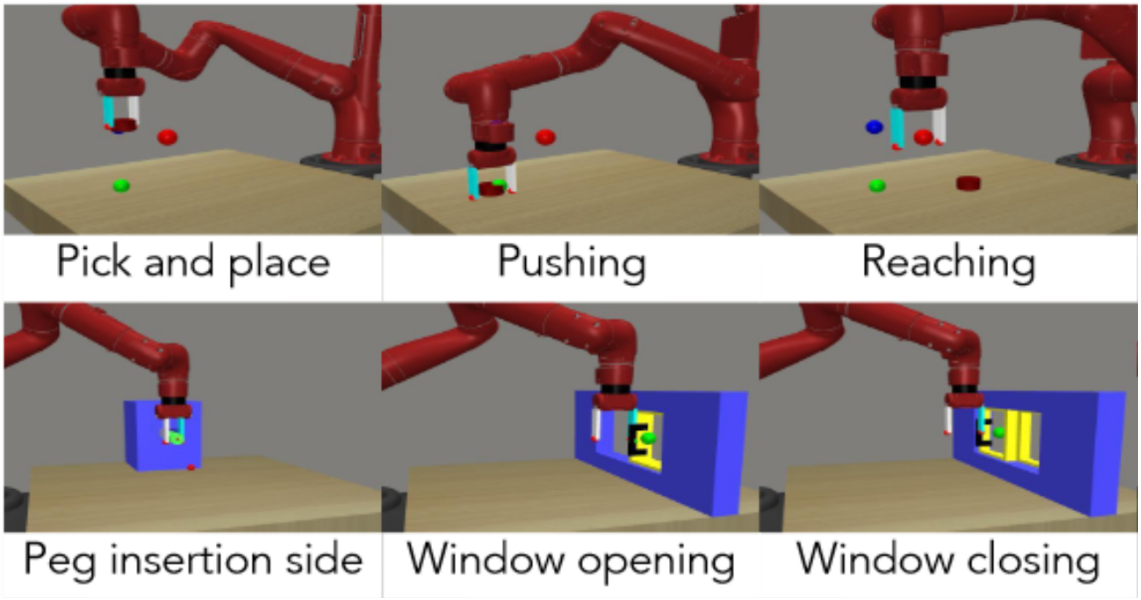


Figure 2.4: Learning continuous robotic control tasks is an important application of deep meta-learning techniques. Image taken from (Yu et al., 2019).

Group i) uses a more traditional approach, to select a suitable machine learning algorithm and hyperparameters for a new data set $\mathcal{D}$ (Peng et al., 2002). This selection can for example be made by leveraging prior model evaluations on various data sets D', and by using the model which achieved the best performance on the most similar data set (Vanschoren, 2018). Such traditional approaches require (large) databases of prior model evaluations, for many different algorithms. This has led to initiatives such as OpenML (Vanschoren et al., 2014), where researchers can share such information. The usage of these systems would limit the freedom in picking the neural network architecture as they would be constrained to using architectures that have been evaluated beforehand.

In contrast, group ii) adopts the view of a self-improving (neural) agent, which improves its learning ability over time by finding a good inductive bias (a set of assumptions that guide predictions). We now present a brief historical overview of developments in this field of deep meta-learning, based on Hospedales et al. (2021).

Pioneering work was done by Schmidhuber (1987) and Hinton and Plaut (1987). Schmidhuber developed a theory of *self-referential* learning, where the weights of a neural network can serve as input to the model itself, which then predicts updates (Schmidhuber, 1987, 1993). In that same year, Hinton and Plaut (1987) proposed to use two weights per neural network connection, i.e., *slow* and *fast* weights, which serve as long- and short-term memory respectively. Later came the idea of meta-learning learning rules (Bengio et al., 1991, 1997). Meta-learning techniques that use gradient-descent and backpropagation were proposed by Hochreiter et al. (2001) and Younger et al. (2001). These two works have been pivotal to the current field of deep meta-learning, as the majority of techniques rely on backpropagation, as we will see on our journey of contemporary deep meta-learning techniques.

	Metric	Model	Optimization
Key idea	Input similarity	Internal task representation	Optimize for fast adaptation
Strength	Simple and effective	Flexible	More robust generalizability
$p_{\boldsymbol{\theta}}(Y\|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j})$	$\sum_{(\boldsymbol{x}_i, y_i) \in D^{tr}_{\mathcal{T}_j}} k_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}_i) y_i$	$f_{\boldsymbol{\theta}}(\boldsymbol{x}, D^{tr}_{\mathcal{T}_j})$	$f_{g_{\boldsymbol{\varphi}}(\boldsymbol{\theta}, D^{tr}_{\mathcal{T}_j}, \mathcal{L}_{D^{tr}_{\mathcal{T}_j}})}(\boldsymbol{x})$

Table 2.2: High-level overview of the three deep meta-learning categories, i.e., i) metric-, ii) model-, and iii) optimization-based techniques, and their main strengths and weaknesses. Recall that $\mathcal{T}_j$ is a task, $D^{tr}_{\mathcal{T}_j}$ the corresponding support set, $k_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}_i)$ a kernel function returning the similarity between the two inputs $\boldsymbol{x}$ and $\boldsymbol{x}_i$, y_i are true labels for known inputs $\boldsymbol{x}_i$, θ are base-learner parameters, and $g_{\boldsymbol{\varphi}}$ is a (learned) optimizer with parameters $\boldsymbol{\varphi}$.

2.2.4 Overview of the rest of this chapter

In the remainder of this chapter, we will look in more detail at individual meta-learning methods. As indicated before, the techniques can be grouped into three main categories (Vinyals, 2017), namely i) metric-, ii) model-, and iii) optimization-based methods. We will discuss them in that order.

To help give an overview of the methods, we draw your attention to the following figure and tables. Table 2.2 summarizes the three categories, and provides key ideas, and strengths of the approaches. The terms and technical details are explained more fully in the remainder of this chapter. Table 2.3 contains an overview of all techniques that are discussed further on.

2.3 Metric-based meta-learning

At a high level, the goal of metric-based techniques is to acquire—among others—meta-knowledge ω in the form of a good feature space that can be used for various new tasks. In the context of neural networks, this feature space coincides with the weights $\boldsymbol{\theta}$ of the networks. Then, new tasks can be learned by comparing new inputs to example inputs (of which we know the labels) in the meta-learned feature space. The higher the similarity between a new input and an example, the more likely it is that the new input will have the same label as the example input.

Metric-based techniques are a form of meta-learning as they leverage their prior learning experience (meta-learned feature space) to 'learn' new tasks more quickly. Here, 'learn' is used in a non-standard way since metric-based techniques do not make any network changes when presented with new tasks, as they rely solely on input comparisons in the already meta-learned feature space. These input comparisons are a form of *non-parametric learning*, i.e., new task information is not absorbed into the network parameters.

More formally, metric-based learning techniques aim to learn a similarity kernel, or equivalently, *attention mechanism* $k_{\boldsymbol{\theta}}$ (parameterized by $\boldsymbol{\theta}$), that takes two inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, and outputs their similarity score. Larger scores indicate larger similarity. Class predictions for new inputs $\boldsymbol{x}$ can then be made by comparing $\boldsymbol{x}$ to example inputs $\boldsymbol{x}_i$, of which we know the true labels y_i. The underlying idea being that the larger the similarity between $\boldsymbol{x}$ and $\boldsymbol{x}_i$, the more likely it becomes that $\boldsymbol{x}$ also has label y_i.

Name	RL	Key idea	Bench.
Metric-based		**Input similarity**	-
Siamese nets	✗	Two-input, shared-weight, class identity network	1, 8
Matching nets	✗	Learn input embeddings for cosine-similarity weighted predictions	1, 2
Prototypical nets	✗	Input embeddings for class prototype clustering	1, 2, 7
Relation nets	✗	Learn input embeddings and similarity metric	1, 2, 7
ARC	✗	LSTM-based input fusion through interleaved glimpses	1, 2
GNN	✗	Propagate label information to unlabeled inputs in a graph	1, 2
Model-based		**Internal and stateful latent task representations**	-
RMLs	✓	Deploy Recurrent nets on RL problems	-
MANNs	✗	External short-term memory module for fast learning	1
Meta nets	✓	Fast reparameterization of base-learner by distinct meta-learner	1, 2
SNAIL	✓	Attention mechanism coupled with temporal convolutions	1, 2
CNP	✗	Condition predictive model on embedded contextual task data	1, 8
Neural stat.	✗	Similarity between latent task embeddings	1, 8
Opt.-based		**Optimize for fast task-specific adaptation**	-
LSTM optimizer	✗	RNN proposing weight updates for base-leaner	6, 8
LSTM ml.	✓	Embed base-learner parameters in cell state of LSTM	2
RL optimizer	✗	View optimization as RL problem	4, 6
MAML	✓	Learn initialization weights $\boldsymbol{\theta}$ for fast adaptation	1, 2
iMAML	✓	Approx. higher-order gradients, independent of optimization path	1, 2
Meta-SGD	✓	Learn both the initialization and updates	1, 2
Reptile	✓	Move initialization towards task-specific updated weights	1, 2
LEO	✗	Optimize in lower-dimensional latent parameter space	2, 3
Online MAML	✗	Accumulate task data for MAML-like training	4, 8
LLAMA	✗	Maintain probability distribution over post-update parameters $\boldsymbol{\theta}'_j$	2
PLATIPUS	✗	Learn a probability distribution over weight initializations $\boldsymbol{\theta}$	-
BMAML	✓	Learn multiple initializations $\boldsymbol{\Theta}$, jointly optimized by SVGD	2
Diff. solvers	✗	Learn input embeddings for simple base-learners	1, 2, 3, 4, 5

Table 2.3: Overview of the discussed deep meta-learning techniques. The table is partitioned into three sections, i.e., metric-, model-, and optimization-based techniques. All methods in one section adhere to the key idea of its corresponding category, which is mentioned in bold font. The columns RL and Bench show whether the techniques are applicable to reinforcement learning settings and the used benchmarks for testing the performance of the techniques. Note that all techniques are applicable to supervised learning, with the exception of RMLs. The benchmark column displays which benchmarks from Section 2.2.2 were used in the paper proposing the technique. The used coding scheme for this column is the following. 1: Omniglot, 2: miniImageNet, 3: tieredImageNet, 4: CIFAR-100, 5: CIFAR-FS, 6: CIFAR-10, 7: CUB, 8: MNIST, "-": used other evaluation method that are non-standard in deep meta-learning and thus not covered in Section 2.2.2. Used abbreviations: "opt.": optimization, "diff.": differentiable, "bench.": benchmarks.

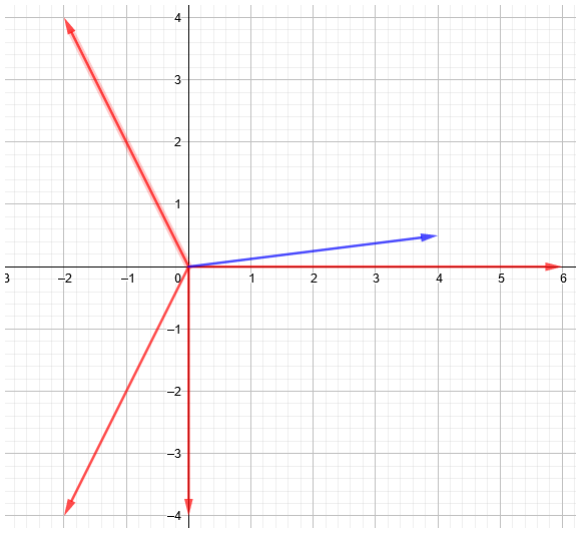


Figure 2.5: Illustration of our metric-based example. The blue vector represents the new input from the query set, whereas the red vectors are inputs from the support set which can be used to guide our prediction for the new input.

Given a task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j})$ and an unseen input vector $\boldsymbol{x} \in D^{test}_{\mathcal{T}_j}$, a probability distribution over classes Y is computed/predicted as a weighted combination of labels from the support set $D^{tr}_{\mathcal{T}_j}$, using similarity kernel $k_{\boldsymbol{\theta}}$, i.e.,

$$p_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j}) = \sum_{(\boldsymbol{x}_i, y_i) \in D^{tr}_{\mathcal{T}_j}} k_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}_i) y_i. \tag{2.7}$$

Importantly, the labels y_i are assumed to be *one-hot encoded*, meaning that they are represented by zero vectors with a '1' on the position of the true class. For example, suppose there are five classes in total, and our example $\boldsymbol{x}_1$ has true class 4. Then, the one-hot encoded label is $y_1 = [0, 0, 0, 1, 0]$. Note that the probability distribution $p_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j})$ over classes is a vector of size $|Y|$, in which the i-th entry corresponds to the probability that input $\boldsymbol{x}$ has class Y_i (given the support set). The predicted class is thus $\hat{y} = \arg\max_{i=1,2,\ldots,|Y|} p_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, S)_i$, where $p_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, S)_i$ is the computed probability that input $\boldsymbol{x}$ has class Y_i.

2.3.1 Example

Suppose that we are given a task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j})$. Furthermore, suppose that $D^{tr}_{\mathcal{T}_j} = \{([0, -4], 1), ([-2, -4], 2), ([-2, 4], 3), ([6, 0], 4)\}$, where a tuple denotes a pair $(\boldsymbol{x}_i, y_i)$. For simplicity, the example will not use an embedding function, which maps example inputs onto an (more informative) embedding space. Our query set only contains one example $D^{test}_{\mathcal{T}_j} = \{([4, 0.5], y)\}$. Then, the goal is to predict the correct label for new input $[4, 0.5]$ using only examples in $D^{tr}_{\mathcal{T}_j}$. The problem is visualized in Figure 2.5, where red vectors correspond to example inputs from our support set. The blue vector is the new input that needs to be classified. Intuitively, this new input is most similar to the vector

$\boldsymbol{x}_i$	y_i	$\|\|\boldsymbol{x}_i\|\|$	$\frac{\boldsymbol{x}_i}{\|\|\boldsymbol{x}_i\|\|}$	$\frac{\boldsymbol{x}_i}{\|\|\boldsymbol{x}_i\|\|} \cdot \frac{\boldsymbol{x}}{\|\|\boldsymbol{x}\|\|}$
$[0, -4]$	$[1, 0, 0, 0]$	4	$[0, -1]$	-0.12
$[-2, -4]$	$[0, 1, 0, 0]$	4.47	$[-0.48, -0.89]$	-0.58
$[-2, 4]$	$[0, 0, 1, 0]$	4.47	$[-0.48, 0.89]$	-0.37
$[6, 0]$	$[0, 0, 0, 1]$	6	$[1, 0]$	0.99

Table 2.4: Example showing pair-wise input comparisons. Numbers were rounded to two decimals.

$[6, 0]$, which means that we expect the label for the new input to be the same as that for $[6, 0]$, i.e., 4.

Suppose we use a fixed similarity kernel, namely the cosine similarity, i.e., $k(\boldsymbol{x}, \boldsymbol{x}_i) = \frac{\boldsymbol{x} \cdot \boldsymbol{x}_i^T}{||\boldsymbol{x}|| \cdot ||\boldsymbol{x}_i||}$, where $||\boldsymbol{v}||$ denotes the length of vector $\boldsymbol{v}$, i.e., $||\boldsymbol{v}|| = \sqrt{(\sum_n v_n^2)}$. Here, v_n denotes the n-th element of placeholder vector $\boldsymbol{v}$ (substitute $\boldsymbol{v}$ by $\boldsymbol{x}$ or $\boldsymbol{x}_i$). We can now compute the cosine similarity between the new input $[4, 0.5]$ and every example input $\boldsymbol{x}_i$, as done in Table 2.4, where we used the facts that $||\boldsymbol{x}|| = ||\,[4, 0.5]\,|| = \sqrt{4^2 + 0.5^2} \approx 4.03$, and $\frac{\boldsymbol{x}}{||\boldsymbol{x}||} \approx \frac{[4,0.5]}{4.03} = [0.99, 0.12]$.

From this table and Equation 2.7, it follows that the predicted probability distribution $p_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, D_{T_j}^{tr}) = -0.12y_1 - 0.58y_2 - 0.37y_3 + 0.99y_4 = -0.12[1, 0, 0, 0] - 0.58[0, 1, 0, 0] - 0.37[0, 0, 1, 0] + 0.99[0, 0, 0, 1] = [-0.12, -0.58, -0.37, 0.99]$. Note that this is not really a probability distribution. That would require normalization such that every element is at least 0 and the sum of all elements is 1. For the sake of this example, we do not perform this normalization, as it is clear that class 4 (the class of the most similar example input $[6, 0]$) will be predicted.

One may wonder why such techniques are meta-learners, for we could take any single data set $\mathcal{D}$ and use pair-wise comparisons to compute predictions. At the outer-level, metric-based meta-learners are trained on a distribution of different tasks, in order to learn (among others) a good input embedding function. This embedding function facilitates inner-level learning, which is achieved through pair-wise comparisons. As such, one learns an embedding function across tasks to facilitate task-specific learning, which is equivalent to "learning to learn", or meta-learning.

After this introduction to metric-based methods, we will now cover some key metric-based techniques.

2.3.2 Siamese neural networks

A Siamese neural network (Koch et al., 2015) consists of two neural networks $f_{\boldsymbol{\theta}}$ that share the same weights $\boldsymbol{\theta}$. Siamese neural networks take two inputs $\boldsymbol{x}_1, \boldsymbol{x}_2$, and compute two hidden states $f_{\boldsymbol{\theta}}(\boldsymbol{x}_1), f_{\boldsymbol{\theta}}(\boldsymbol{x}_2)$, corresponding to the activation patterns in the final hidden layers. These hidden states are fed into a distance layer, which computes a distance vector $\boldsymbol{d} = |f_{\boldsymbol{\theta}}(\boldsymbol{x}_1) - f_{\boldsymbol{\theta}}(\boldsymbol{x}_2)|$, where d_i is the absolute distance between the i-th elements of $f_{\boldsymbol{\theta}}(\boldsymbol{x}_1)$ and $f_{\boldsymbol{\theta}}(\boldsymbol{x}_2)$. From this distance vector, the similarity between $\boldsymbol{x}_1, \boldsymbol{x}_2$ is computed as $\sigma(\boldsymbol{\alpha}^T \boldsymbol{d})$, where σ is the sigmoid function (with output range [0,1]), and $\boldsymbol{\alpha}$ is a vector of free weighting parameters, determining the importance of each d_i. This network structure can be seen in Figure 2.6.

Koch et al. (2015) applied this technique to few-shot image recognition in two stages. In the first stage, they train the twin network on an *image verification* task, where the goal is to output whether two input images $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ have the same class. The network is thus stimulated to learn discriminative

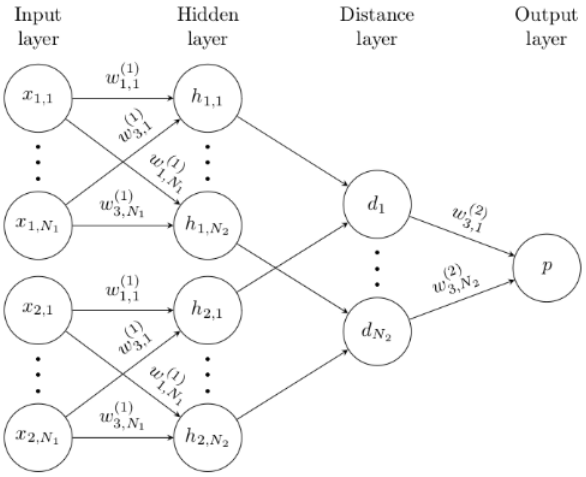


Figure 2.6: Example of a Siamese neural network. Source: Koch et al. (2015).

features. In the second stage, where the model is confronted with a new task, the network leverages its prior learning experience. That is, given a task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j})$, and previously unseen input $\boldsymbol{x} \in D^{test}_{\mathcal{T}_j}$, the predicted class $\hat{y}$ is equal to the label y_i of the example $(\boldsymbol{x}_i, y_i) \in D^{tr}_{\mathcal{T}_j}$ which yields the highest similarity score to $\boldsymbol{x}$. In contrast to other techniques mentioned further in this section, Siamese neural networks do not directly optimize for good performance across tasks (consisting of support and query sets). However, they do leverage learned knowledge from the verification task to learn new tasks quicker.

In summary, Siamese neural networks are a simple and elegant approach to perform few-shot learning. However, they are not readily applicable outside the supervised learning setting.

2.3.3 Matching networks

Matching networks (Vinyals et al., 2016) build upon the idea that underlies Siamese neural networks (Koch et al., 2015). That is, they leverage pair-wise comparisons between the given support set $D^{tr}_{\mathcal{T}_j} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{m}$ (for a task $\mathcal{T}_j$), and new inputs $\boldsymbol{x} \in D^{test}_{\mathcal{T}_j}$ from the query set which we want to classify. However, instead of assigning the class y_i of the most similar example input $\boldsymbol{x}_i$, matching networks use a weighted combination of *all* example labels y_i in the support set, based on the similarity of inputs $\boldsymbol{x}_i$ to new input $\boldsymbol{x}$. More specifically, predictions are computed as follows: $\hat{y} = \sum_{i=1}^{m} a(\boldsymbol{x}, \boldsymbol{x}_i) y_i$, where a is a non-parametric (non-trainable) attention mechanism, or similarity kernel. This classification process is shown in Figure 2.7. In this figure, the input to $f_{\boldsymbol{\theta}}$ has to be classified, using the support set $D^{tr}_{\mathcal{T}_j}$ (input to $g_{\boldsymbol{\theta}}$).

The attention that is used consists of a softmax over the cosine similarity c between the input representations, i.e.,

$$a(\boldsymbol{x}, \boldsymbol{x}_i) = \frac{e^{c(f_{\boldsymbol{\phi}}(\boldsymbol{x}), g_{\boldsymbol{\varphi}}(\boldsymbol{x}_i))}}{\sum_{j=1}^{m} e^{c(f_{\boldsymbol{\phi}}(\boldsymbol{x}), g_{\boldsymbol{\varphi}}(\boldsymbol{x}_j))}}, \tag{2.8}$$

where $f_{\boldsymbol{\phi}}$ and $g_{\boldsymbol{\varphi}}$ are neural networks, parameterized by $\boldsymbol{\phi}$ and $\boldsymbol{\varphi}$, that map raw inputs to a (lower-

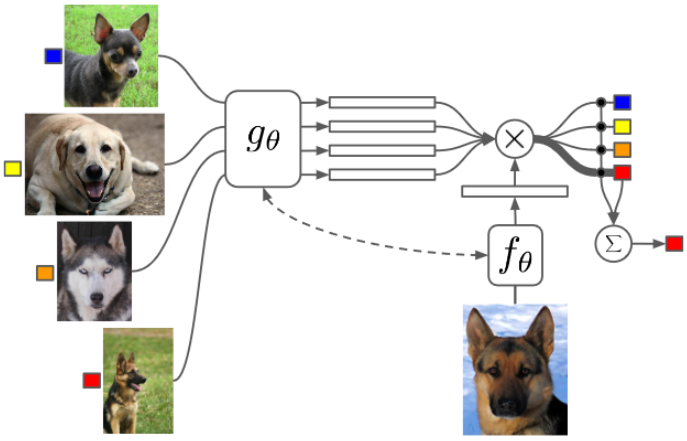


Figure 2.7: Architecture of matching networks. Source: Vinyals et al. (2016).

dimensional) latent vector, which corresponds to the output of the final hidden layer of a neural network. As such, the neural networks act as embedding functions. The larger the cosine similarity between the embeddings of $\boldsymbol{x}$ and $\boldsymbol{x}_i$, the larger $a(\boldsymbol{x}, \boldsymbol{x}_i)$, and thus the influence of label y_i on the predicted label $\hat{y}$ for input $\boldsymbol{x}$.

Vinyals et al. (2016) propose two main choices for the embedding functions. The first is to use a single neural network, granting us $\boldsymbol{\theta} = \boldsymbol{\phi} = \boldsymbol{\varphi}$ and thus $f_{\boldsymbol{\phi}} = g_{\boldsymbol{\varphi}}$. This setup is the default form of matching networks, as shown in Figure 2.7. The second choice is to make $f_{\boldsymbol{\phi}}$ and $g_{\boldsymbol{\varphi}}$ dependent on the support set $D^{tr}_{\mathcal{T}_j}$ using Long Short-Term Memory networks (LSTMs). In that case, $f_{\boldsymbol{\phi}}$ is represented by an attention LSTM, and $g_{\boldsymbol{\varphi}}$ by a bidirectional one. This choice for embedding functions is called *Full Context Embeddings* (FCE), and yielded an accuracy improvement of roughly 2% on miniImageNet compared to the regular matching networks, indicating that task-specific embeddings can aid the classification of new data points from the same distribution.

Matching networks learn a good feature space across tasks for making pair-wise comparisons between inputs. In contrast to Siamese neural networks (Koch et al., 2015), this feature space (given by weights $\boldsymbol{\theta}$) is learned across tasks, instead of on a distinct verification task.

In summary, matching networks are an elegant and simple approach to metric-based meta-learning. However, these networks are not readily applicable outside of supervised learning settings, and suffer from performance degradation when label distributions are biased (Vinyals et al., 2016).

2.3.4 Prototypical networks

Just like Matching nets (Vinyals et al., 2016), prototypical nets (Snell et al., 2017) base their class predictions on the entire support set $D^{tr}_{\mathcal{T}_j}$. However, instead of computing the similarity between new inputs and examples in the support set, prototypical nets only compare new inputs to *class prototypes* (centroids), which are single vector representations of classes in some embedding space. Since there are less (or equal) class prototypes than the number of examples in the support set, the amount of required pair-wise comparisons decreases, saving computational costs.

The underlying idea of class prototypes is that for a task $\mathcal{T}_j$, there exists an embedding function that

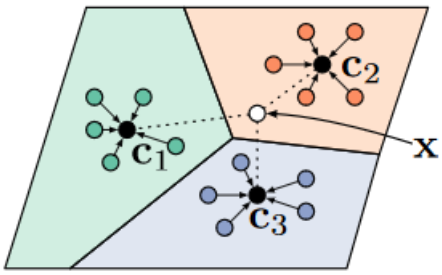


Figure 2.8: Prototypical networks for case of few-shot learning. The $\boldsymbol{c}_k$ are class prototypes for class k which are computed by averaging the representations of inputs (colored circles) in the support set. Note that the representation space is partitioned into three disjoint areas, where each area corresponds to one class. The class with the closest prototype to the new input $\boldsymbol{x}$ in the query set is then given as prediction. Source: Snell et al. (2017).

maps the support set onto a space where class instances cluster nicely around the corresponding class prototypes (Snell et al., 2017). Then, for a new input $\boldsymbol{x}$, the class of the prototype nearest to that input will be predicted. As such, prototypical nets perform nearest centroid/prototype classification in a meta-learned embedding space. This is visualized in Figure 2.8.

More formally, given a distance function $d : X \times X \rightarrow [0, +\infty)$ (e.g. Euclidean distance) and embedding function $f_{\boldsymbol{\theta}}$, parameterized by $\boldsymbol{\theta}$, prototypical networks compute class probabilities $p_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j})$ as follows

$$p_{\boldsymbol{\theta}}(y = k|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j}) = \frac{exp[-d(f_\theta(\boldsymbol{x}), \boldsymbol{c}_k)]}{\sum_{y_i} exp[-d(f_\theta(\boldsymbol{x}), \boldsymbol{c}_{y_i})]}, \tag{2.9}$$

where $\boldsymbol{c}_k$ is the prototype/centroid for class k and y_i are the classes in the support set $D^{tr}_{\mathcal{T}_j}$. Here, a class prototype for class k is defined as the average of all vectors $\boldsymbol{x}_i$ in the support set such that $y_i = k$. Thus, classes with prototypes that are nearer to the new input $\boldsymbol{x}$ obtain larger probability scores.

Snell et al. (2017) found that the squared Euclidean distance function as d gave rise to the best performance. With that distance function, prototypical networks can be seen as linear models. To see this, note that $-d(f_\theta(\boldsymbol{x}), \boldsymbol{c}_k) = -||f_\theta(\boldsymbol{x}) - \boldsymbol{c}_k||^2 = -f_\theta(\boldsymbol{x})^T f_\theta(\boldsymbol{x}) + 2\boldsymbol{c}_k^T f_\theta(\boldsymbol{x}) - \boldsymbol{c}_k^T \boldsymbol{c}_k$. The first term does not depend on the class k, and does thus not affect the classification decision. The remainder can be written as $\boldsymbol{w}_k^T f_\theta(\boldsymbol{x}) + \boldsymbol{b}_k$, where $\boldsymbol{w}_k = 2\boldsymbol{c}_k$ and $\boldsymbol{b}_k = -\boldsymbol{c}_k^T \boldsymbol{c}_k$. Note that this is linear in the output of network f_θ, not linear in the input of the network $\boldsymbol{x}$. Also, Snell et al. (2017) show that prototypical nets (coupled with Euclidean distance) are equivalent to matching nets in one-shot learning settings, as every example in the support set will be its own prototype.

In short, prototypical nets save computational costs by reducing the required number of pair-wise comparisons between new inputs and the support set, by adopting the concept of class prototypes. Additionally, prototypical nets were found to outperform matching nets (Vinyals et al., 2016) in 5-way, k-shot learning for $k = 1, 5$ on Omniglot (Lake et al., 2011) and miniImageNet (Vinyals et al., 2016), even though they do not use complex task-specific embedding functions. Despite these advantages,

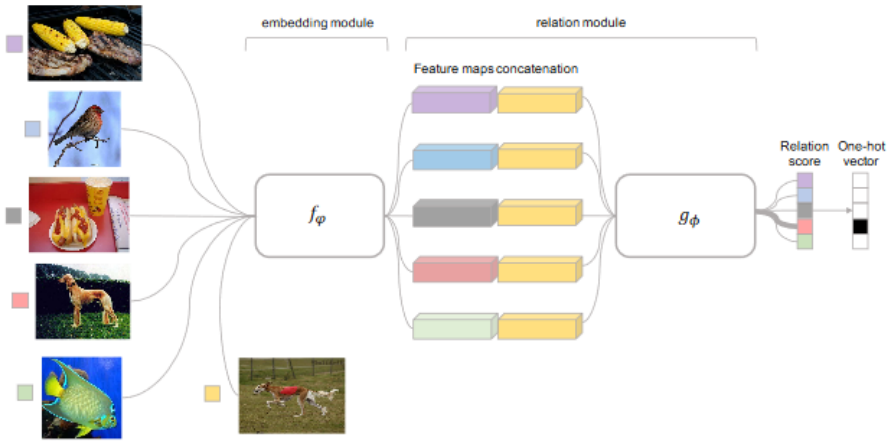


Figure 2.9: Relation network architecture. First, the embedding network $f_{\boldsymbol{\varphi}}$ embeds all inputs from the support set $D^{tr}_{\mathcal{T}_j}$ (the five example inputs on the left), and the query input (below the $f_{\boldsymbol{\varphi}}$ block). All support set embeddings $f_{\boldsymbol{\varphi}}(\boldsymbol{x}_i)$ are then concatenated to the query embedding $f_{\boldsymbol{\varphi}}(\boldsymbol{x})$. These concatenated embeddings are passed into a relation network $g_{\boldsymbol{\phi}}$, which computes a relation score for every pair $(\boldsymbol{x}_i, \boldsymbol{x})$. The class of the input $\boldsymbol{x}_i$ that yields the largest relation score $g_{\boldsymbol{\phi}}([f_{\boldsymbol{\varphi}}(\boldsymbol{x}), f_{\boldsymbol{\varphi}}(\boldsymbol{x}_i)])$ is then predicted. Source: Sung et al. (2018).

prototypical nets are not readily applicable outside of supervised learning settings.

2.3.5 Relation networks

In contrast to previously discussed metric-based techniques, Relation networks (Sung et al., 2018) employ a trainable similarity metric, instead of a pre-defined one (e.g. cosine similarity as used in matching nets (Vinyals et al., 2016)). More specifically, matching nets consist of two chained, neural network modules: the *embedding* network/module $f_{\boldsymbol{\varphi}}$ which is responsible for embedding inputs, and the *relation* network $g_{\boldsymbol{\phi}}$ which computes similarity scores between new inputs $\boldsymbol{x}$ and example inputs $\boldsymbol{x}_i$ of which we know the labels. A classification decision is then made by picking the class of the example input which yields the largest *relation score* (or similarity). Note that Relation nets thus do not use the idea of class prototypes, and simply compare new inputs $\boldsymbol{x}$ to all example inputs $\boldsymbol{x}_i$ in the support set, as done by, e.g., matching networks (Vinyals et al., 2016).

More formally, we are given a support set $D^{tr}_{\mathcal{T}_j}$ with some examples $(\boldsymbol{x}_i, y_i)$, and a new (previously unseen) input $\boldsymbol{x}$. Then, for every combination $(\boldsymbol{x}, \boldsymbol{x}_i)$, the Relation network produces a *concatenated* embedding $[f_{\boldsymbol{\varphi}}(\boldsymbol{x}), f_{\boldsymbol{\varphi}}(\boldsymbol{x}_i)]$, which is vector obtained by concatenating the respective embeddings of $\boldsymbol{x}$ and $\boldsymbol{x}_i$. This concatenated embedding is then fed into the *relation* module $g_{\boldsymbol{\phi}}$. Finally, $g_{\boldsymbol{\phi}}$ computes the relation score between $\boldsymbol{x}$ and $\boldsymbol{x}_i$ as

$$r_i = g_{\boldsymbol{\phi}}([f_{\boldsymbol{\varphi}}(\boldsymbol{x}), f_{\boldsymbol{\varphi}}(\boldsymbol{x}_i)]). \tag{2.10}$$

The predicted class is then $\hat{y} = y_{\arg\max_i r_i}$. This entire process is shown in Figure 2.9. Remarkably enough, Relation nets use the Mean-Squared Error (MSE) of the relation scores, rather than the more standard cross-entropy loss. The MSE is then propagated backwards through the entire architecture

(Figure 2.9).

The key advantage of Relation nets is their expressive power, induced by the usage of a trainable similarity function. This expressivity makes this technique very powerful. As a result, it yields better performance than previously discussed techniques that use a fixed similarity metric.

2.3.6 Graph neural networks

Graph neural networks (Garcia and Bruna, 2017) use a more general and flexible approach than previously discussed techniques for N-way, k-shot classification. As such, graph neural networks subsume Siamese (Koch et al., 2015) and prototypical networks (Snell et al., 2017). The graph neural network approach represents each task $\mathcal{T}_j$ as a fully-connected graph $G = (V, E)$, where V is a set of nodes/vertices and E a set of edges connecting nodes. In this graph, nodes $\boldsymbol{v}_i$ correspond to input embeddings $f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$, concatenated with their one-hot encoded labels y_i, i.e., $\boldsymbol{v}_i = [f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), y_i]$. For inputs $\boldsymbol{x}$ from the query set (for which we do not have the labels), a uniform prior over all N possible labels is used: $y = [\frac{1}{N}, \ldots, \frac{1}{N}]$. Thus, each node contains an input and label section. Edges are weighted links that connect these nodes.

The graph neural network then propagates information in the graph using a number of local operators. The underlying idea is that label information can be transmitted from nodes of which we do have the labels, to nodes for which we have to predict labels. Which local operators are used, is out of scope for this chapter, and the reader is referred to Garcia and Bruna (2017) for details.

By exposing the graph neural network to various tasks $\mathcal{T}_j$, the propagation mechanism can be altered to improve the flow of label information in such a way that predictions become more accurate. As such, in addition to learning a good input representation function $f_{\boldsymbol{\theta}}$, graph neural networks also learn to propagate label information from labeled examples to unlabeled inputs.

Graph neural networks achieve good performance in few-shot settings (Garcia and Bruna, 2017), and are also applicable in semi-supervised and active learning settings.

2.3.7 Attentive recurrent comparators

Attentive recurrent comparators (Shyam et al., 2017) differ from previously discussed techniques as they do not compare inputs as a whole, but by parts. This approach is inspired by how humans would make a decision concerning the similarity of objects. That is, we shift our attention from one object to the other, and move back and forth to take glimpses of different parts of both objects. In this way, information of two objects is fused from the beginning, whereas other techniques (e.g., matching networks (Vinyals et al., 2016) and graph neural networks (Garcia and Bruna, 2017)) only combine information at the end (after embedding both images) (Shyam et al., 2017).

Given two inputs $\boldsymbol{x}_i$ and $\boldsymbol{x}$, we feed them in interleaved fashion repeatedly into a recurrent neural network (controller): $\boldsymbol{x}_i, \boldsymbol{x}, \ldots, \boldsymbol{x}_i, \boldsymbol{x}$. Thus, the image at time step t is given by $I_t = \boldsymbol{x}_i$ if t is even else $\boldsymbol{x}$. Then, at each time step t, the attention mechanism focuses on a square region of the current image: $G_t = attend(I_t, \Omega_t)$, where $\Omega_t = W_g h_{t-1}$ are attention parameters, which are computed from the previous hidden state h_{t-1}. The next hidden state $h_{t+1} = \text{RNN}(G_t, h_{t-1})$ is given by the glimpse at time t, i.e., G_t, and the previous hidden state h_{t-1}. The entire sequence consists of g glimpses per image. After this sequence is fed into the recurrent neural network (indicated by RNN($\circ$)), the final hidden state h_{2g} is used as combined representation of $\boldsymbol{x}_i$ relative to $\boldsymbol{x}$. This process is summarized in Figure 2.10. Classification decisions can then be made by feeding the combined representations

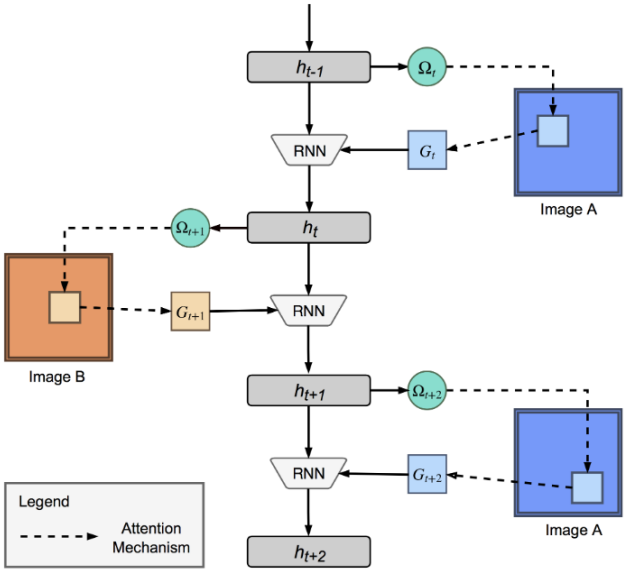


Figure 2.10: Processing in an attentive recurrent comparator. At every time step, the model takes a glimpse of a part of an image and incorporates this information into the hidden state h_t. The final hidden state after taking various glimpses of a pair of images is then used to compute a class similarity score. Source: Shyam et al. (2017).

into a classifier. Optionally, the combined representations can be processed by bi-directional LSTMs before passing them to the classifier.

The attention approach is biologically inspired, and biologically plausible. A downside of attentive recurrent comparators is the higher computational cost, while the performance is often not better than less biologically plausible techniques, such as graph neural networks (Garcia and Bruna, 2017).

2.3.8 Metric-based techniques, in conclusion

In this section, we have seen various metric-based techniques. The metric-based techniques meta-learn an informative feature space that can be used to compute class predictions based on input similarity scores. Figure 2.11 shows the relationships between the various metric-based techniques that we have covered.

As we can see, Siamese networks (Koch et al., 2015) mark the beginning of metric-based, deep meta-learning techniques in few-shot learning settings. They are the first to use the idea of predicting classes by comparing inputs from the support and query sets. This idea was generalized in GNNs (Hamilton et al., 2017; Garcia and Bruna, 2017) where the information flow between support and query inputs is parametric and thus more flexible. Matching networks (Vinyals et al., 2016) are directly inspired by Siamese networks as they use the same core idea (comparing inputs for making predictions), but directly train in the few-shot setting and use cosine similarity as similarity function. Thus, the auxiliary, binary classification task used by Siamese networks is left out, and matching networks directly train on tasks. Prototypical networks (Snell et al., 2017) increase the robustness of input comparisons by comparing every query set input with a class prototype instead of individual

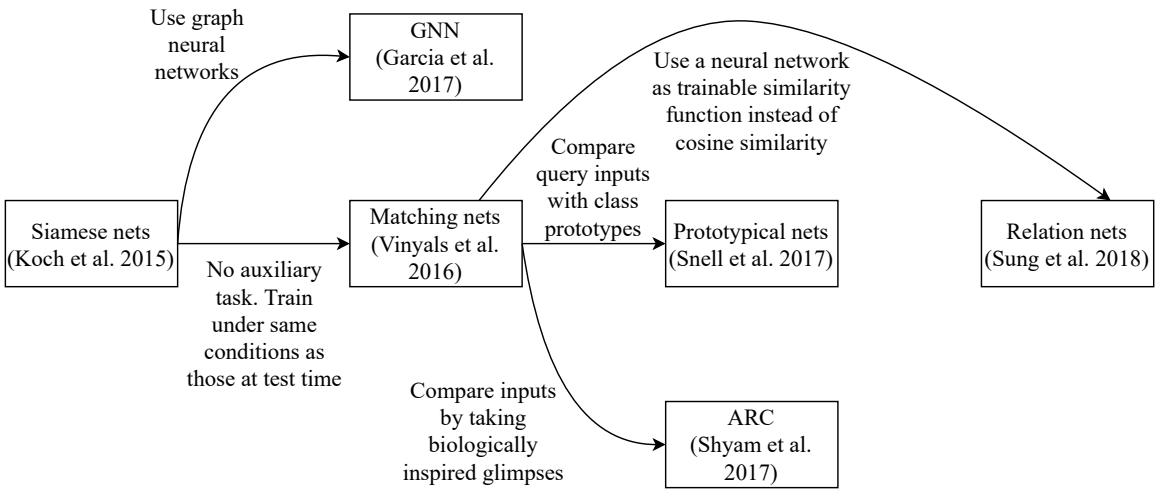


Figure 2.11: The relationships between the covered metric-based meta-learning techniques.

support set examples. This reduces the number of required input comparisons for a single query input to N instead of $k \cdot N$. Relation networks (Sung et al., 2018) replace the fixed, pre-defined similarity metrics used in matching and prototypical networks by a neural network, which allows for learning a domain-specific similarity function. Lastly, ARCs (Shyam et al., 2017) take a more biologically plausible approach by not comparing entire inputs but by taking multiple interleaved glimpses at various parts of the inputs that are being compared.

Key advantages of these metric-based techniques are that i) the underlying idea of similarity-based predictions is conceptually simple, and ii) they can be fast at test-time when tasks are small, as the networks do not need to make task-specific adjustments. However, when tasks at meta-test time become more distant from the tasks that were used at meta-train time, metric-learning techniques are unable to absorb new task information into the network weights. Consequently, performance may degrade.

Furthermore, when tasks become larger, pair-wise comparisons may become prohibitively expensive. Lastly, most metric-based techniques rely on the presence of labeled examples, which make them inapplicable outside of supervised learning settings.

2.4 Model-based meta-learning

A different approach to deep meta-learning is the model-based approach. On a high level, model-based techniques rely upon an adaptive, internal state, in contrast to metric-based techniques, which generally use a fixed neural network at test-time.

More specifically, model-based techniques maintain a stateful, internal representation of a task. When presented with a task, a model-based neural network processes the support set in sequential fashion. At every time step, an input enters, and alters the internal state of the model. Thus, the internal state can capture relevant task-specific information, which can be used to make predictions for new inputs.

Because the predictions are based on internal dynamics that are hidden from the outside, model-based

techniques are also called *black-boxes*. Information from previous inputs must be remembered, which is why model-based techniques have a memory component, either in- or externally.

Recall that the mechanics of metric-based techniques were limited to pair-wise input comparisons. This is not the case for model-based techniques, where the human designer has the freedom to choose the internal dynamics of the algorithm. As a result, model-based techniques are not restricted to meta-learning good feature spaces, as they can also learn internal dynamics, used to process and predict input data of tasks.

More formally, given a support set $D_{\mathcal{T}_j}^{tr}$ corresponding to task $\mathcal{T}_j$, model-based techniques compute a class probability distribution for a new input $\boldsymbol{x}$ as

$$p_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, D_{\mathcal{T}_j}^{tr}) = f_{\boldsymbol{\theta}}(\boldsymbol{x}, D_{\mathcal{T}_j}^{tr}), \tag{2.11}$$

where f represents the black-box neural network model, and $\boldsymbol{\theta}$ its parameters.

2.4.1 Example

Using the same example as in Section 2.3, suppose we are given a task support set $D_{\mathcal{T}_j}^{tr} = \{([0, -4], 1), ([-2, -4], 2), ([-2, 4], 3), ([6, 0], 4)\}$, where a tuple denotes a pair $(\boldsymbol{x}_i, y_i)$. Furthermore, suppose our query set only contains one example $D_{\mathcal{T}_j}^{test} = \{([4, 0.5], 4)\}$. This problem has been visualized in Figure 2.5 (in Section 2.3). For the sake of the example, we do not use an input embedding function: our model will operate on the raw inputs of $D_{\mathcal{T}_j}^{tr}$ and $D_{\mathcal{T}_j}^{test}$. As an internal state, our model uses an external *memory matrix* $M \in \mathbb{R}^{4\times(2+1)}$, with four rows (one for each example in our support set), and three columns (the dimensionality of input vectors, plus one dimension for the correct label). Our model proceeds to process the support set in sequential fashion, reading the examples from $D_{\mathcal{T}_j}^{tr}$ one by one, and by storing the i-th example in the i-th row of the memory module. After processing the support set, the memory matrix contains all examples, and as such, serves as internal task representation.

Given the new input $[4, 0.5]$, our model could use many different techniques to make a prediction based on this representation. For simplicity, assume that it computes the dot product between $\boldsymbol{x}$, and every memory $M(i)$ (the 2-D vector in the i-th row of M, ignoring the correct label), and predicts the class of the input which yields the largest dot product. This would produce scores $-2, -10, -6$, and 24 for the examples in $D_{\mathcal{T}_j}^{tr}$ respectively. Since the last example $[6, 0]$ yields the largest dot product, we predict that class, i.e., 4.

Note that this example could be seen as a metric-based technique where the dot product is used as similarity function. However, the reason that this technique is model-based is that it stores the entire task inside a memory module. This example was deliberately easy for illustrative purposes. More advanced and successful techniques have been proposed, which we will now cover.

2.4.2 Recurrent meta-learners

Recurrent meta-learners (Duan et al., 2016; Wang et al., 2016) are, as the name suggests, meta-learners based on recurrent neural networks. The recurrent network serves as dynamic task embedding storage. These recurrent meta-learners were specifically proposed for reinforcement learning problems, hence we will explain them in that setting.

The recurrence is implemented by e.g. an LSTM (Wang et al., 2016) or a GRU (Duan et al., 2016). The internal dynamics of the chosen Recurrent Neural Network (RNN) allows for fast adaptation

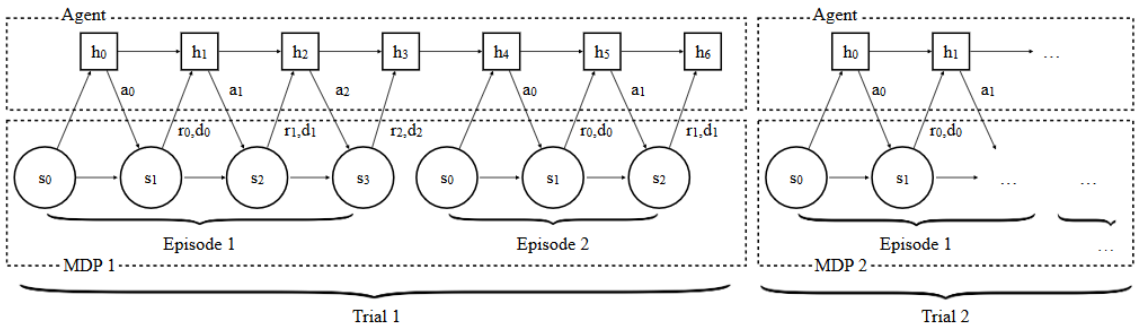


Figure 2.12: Workflow of recurrent meta-learners in reinforcement learning contexts. As mentioned in Section 2.2.1, s_t, r_t, and d_t denote the state, reward, and termination flag at time step t. h_t refers to the hidden state at time t. Source: Duan et al. (2016).

to new tasks, while the algorithm used to train the recurrent net gradually accumulates knowledge about the task structure, where each task is modelled as an episode (or set of episodes).

The idea of recurrent meta-learners is quite simple. That is, given a task $\mathcal{T}_j$, we simply feed the (potentially processed) environment variables $[s_{t+1}, a_t, r_t, d_t]$ (see Section 2.2.1) into an RNN at every time step t. Recall that s, a, r, d denote the state, action, reward, and termination flag respectively. At every time step t, the RNN outputs an action and a hidden state. Conditioned on its hidden state h_t, the network outputs an action a_t. The goal is to maximize the expected reward in each trial. See Figure 2.12 for a visual depiction. From this figure, it also becomes clear why these techniques are model-based. That is, they embed information from previously seen inputs in the hidden state.

Recurrent meta-learners have shown to perform almost as well as asymptotically optimal algorithms on simple reinforcement learning tasks (Wang et al., 2016; Duan et al., 2016). However, their performance degrades in more complex settings, where temporal dependencies can span a longer horizon. Making recurrent meta-learners better at such complex tasks is a direction for future research.

2.4.3 Memory-augmented neural networks (MANNs)

The key idea of memory-augmented neural networks (Santoro et al., 2016) is to enable neural networks to learn quickly with the help of an *external memory*. The main *controller* (the recurrent neural network interacting with the memory) then gradually accumulates knowledge across tasks, while the external memory allows for quick task-specific adaptation. For this, Santoro et al. (2016) used Neural Turing Machines (Graves et al., 2014). Here, the controller is parameterized by $\boldsymbol{\theta}$ and acts as the long-term memory of the memory-augmented neural network, while the external memory module is the short-term memory.

The workflow of memory-augmented neural networks is displayed in Figure 2.13. Note that the data from a task is processed as a sequence, i.e., data are fed into the network one by one. The support set is fed into the memory-augmented neural network first. Afterwards, the query set is processed. During the meta-train phase, training tasks can be fed into the network in arbitrary order. At time step t, the model receives input $\boldsymbol{x}_t$ with the label of the previous input, i.e., y_{t-1}. This was done to prevent the network from mapping class labels directly to the output (Santoro et al., 2016).

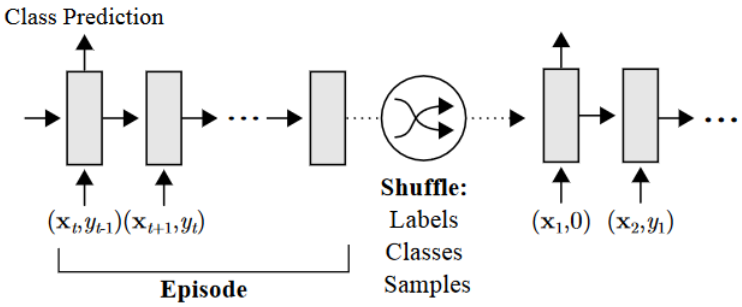


Figure 2.13: Workflow of memory-augmented neural networks. Here, an episode corresponds to a given task $\mathcal{T}_j$. After every episode, the order of labels, classes, and samples should be shuffled to minimize dependence on arbitrarily assigned orders. Source: Santoro et al. (2016).

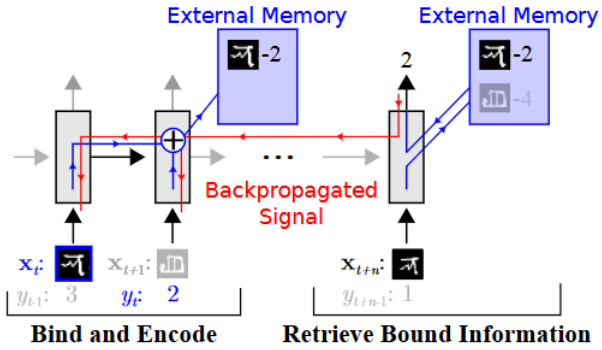


Figure 2.14: Controller-memory interaction in memory-augmented neural networks. Source: Santoro et al. (2016).

The interaction between the controller and memory is visualized in Figure 2.14. The idea is that the external memory module, containing representations of previously seen inputs, can be used to make predictions for new inputs. In short, previously obtained knowledge is leveraged to aid the classification of new inputs. Note that neural networks also attempt to do this, however, their prior knowledge is slowly accumulated into the network weights, while an external memory module can directly store such information.

Given an input $\boldsymbol{x}_t$ at time t, the controller generates a key $\boldsymbol{k}_t$, which can be stored in memory matrix M and can be used to retrieve previous representations from memory matrix M. When reading from memory, the aim is to produce a linear combination of stored keys in memory matrix M, giving greater weight to those which have a larger cosine similarity with the current key $\boldsymbol{k}_t$. More specifically, a read vector $\boldsymbol{w}_t^r$ is created, in which each entry i denotes the cosine similarity between key $\boldsymbol{k}_t$ and the memory (from a previous input) stored in row i, i.e., $M_t(i)$. Then, the representation $\boldsymbol{r}_t = \sum_i w_t^r(i)M(i)$ is retrieved, which is simply a linear combination of all keys (i.e., rows) in memory matrix M.

Predictions are made as follows. Given an input $\boldsymbol{x}_t$, memory-augmented neural networks use the external memory to compute the corresponding representation $\boldsymbol{r}_t$, which could be fed into a softmax

layer, resulting in class probabilities. Across tasks, memory-augmented neural networks learn a good input embedding function $f_{\boldsymbol{\theta}}$ and classifier weights, which can be exploited when presented with new tasks.

To write input representations to memory, Santoro et al. (2016) propose a new mechanism called Least Recently Used Access (LRUA). LRUA either writes to the least, or most recently used memory location. In the former case, it preserves recent memories, and in the latter it updates recently obtained information. The writing mechanism works by keeping track of how often every memory location is accessed in a usage vector $\boldsymbol{w}_t^u$, which is updated at every time step according to the following update rule: $\boldsymbol{w}_t^u := \gamma \boldsymbol{w}_{t-1}^u + \boldsymbol{w}_t^r + \boldsymbol{w}_t^w$, where superscripts u, w and r refer to usage, write and read vectors, respectively. In words, the previous usage vector is decayed (using parameter γ), while current reads ($\boldsymbol{w}_t^r$) and writes ($\boldsymbol{w}_t^w$) are added to the usage. Let n be the total number of reads to memory, and $\ell u(n)$ (ℓu for 'least used') be the n-th smallest value in the usage vector $\boldsymbol{w}_t^u$. Then, the least-used weights are defined as follows:

$$\boldsymbol{w}_t^{\ell u}(i) = \begin{cases} 0 & \text{if } w_t^u(i) > \ell u(n) \\ 1 & else \end{cases}.$$

Then, the write vector $\boldsymbol{w}_t^w$ is computed as $\boldsymbol{w}_t^w = \sigma(\alpha)\boldsymbol{w}_{t-1}^r + (1-\sigma(\alpha))\boldsymbol{w}_{t-1}^{\ell u}$, where α is a parameter that interpolates between the two weight vectors. As such, if $\sigma(\alpha) = 1$, we write to the most recently used memory, whereas when $\sigma(\alpha) = 0$, we write to the least recently used memory locations. Finally, writing is performed as follows: $M_t(i) := M_{t-1}(i) + w_t^w(i)\boldsymbol{k}_t$, for all i.

In summary, memory-augmented neural networks (Santoro et al., 2016) combine external memory and a neural network to achieve meta-learning. The interaction between a controller, with long-term memory parameters $\boldsymbol{\theta}$, and memory M, may also be interesting for studying human meta-learning (Santoro et al., 2016). In contrast to many metric-based techniques, this model-based technique is applicable to both classification and regression problems. A downside of this approach is the architectural complexity.

2.4.4 Meta networks

Meta networks are divided into two distinct subsystems (consisting of neural networks), i.e., the base- and meta-learner (whereas in memory-augmented neural networks the base- and meta-components are intertwined). The base-learner is responsible for performing tasks, and for providing the meta-learner with meta-information, such as loss gradients. The meta-learner can then compute fast task-specific weights for itself and the base-learner, such that it can perform better on the given task $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$. This workflow is depicted in Figure 2.15.

The meta-learner consists of neural networks $u_{\boldsymbol{\phi}}, m_{\boldsymbol{\varphi}}$, and $d_{\boldsymbol{\psi}}$. Network $u_{\boldsymbol{\phi}}$ is used as input representation function. Networks $d_{\boldsymbol{\psi}}$ and $m_{\boldsymbol{\varphi}}$ are used to compute task-specific weights $\boldsymbol{\phi}^*$ and example-level fast weights $\boldsymbol{\theta}^*$. Lastly, $b_{\boldsymbol{\theta}}$ is the base-learner which performs input predictions. Note that we used the term fast-weights throughout, which refers to task- or input-specific versions of slow (initial) weights.

In similar fashion to memory-augmented neural networks (Santoro et al., 2016), meta networks (Munkhdalai and Yu, 2017) also leverage the idea of an external memory module. However, meta networks use the memory for a different purpose. The memory stores for each observation $\boldsymbol{x}_i$ in the support set two components, i.e., its representation $\boldsymbol{r}_i$ and the fast weights $\boldsymbol{\theta}_i^*$. These are then used to compute a attention-based representation and fast weights for new inputs, respectively.

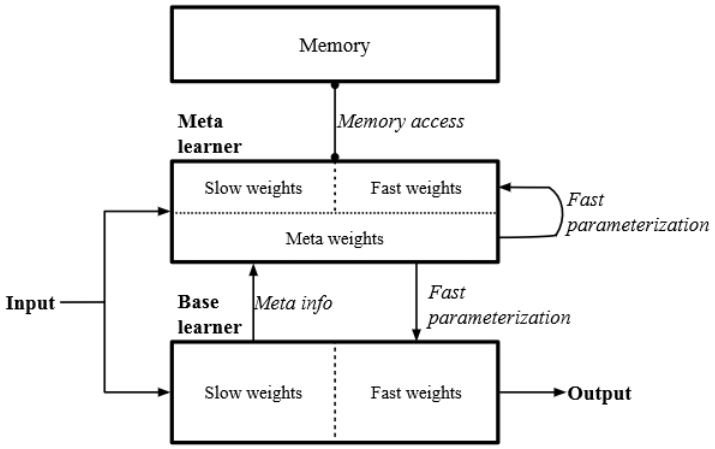


Figure 2.15: Architecture of a Meta Network. Source: Munkhdalai and Yu (2017).

The pseudocode for meta networks is displayed in Algorithm 1. First, a sample of the support set is created (line 1), which is used to compute task-specific weights $\boldsymbol{\phi}^*$ for the representation network $u_{\boldsymbol{\phi}}$ (lines 2-5). Note that $u_{\boldsymbol{\phi}}$ has two tasks, i) it should compute a representation for inputs ($\boldsymbol{x}_i$ (line 10 and 15), and ii) it needs to make predictions for inputs ($\boldsymbol{x}_i$, in order to compute a loss (line 3). To achieve both goals, a conventional neural network can be used that makes class predictions. The states of the final hidden layer are then used as representation. Typically, the cross entropy is calculated over the predictions of representation network $u_{\boldsymbol{\phi}}$. When there are multiple examples per class in the support set, an alternative is to use a contrastive loss function (Munkhdalai and Yu, 2017).

Then, meta networks iterate over every example $(\boldsymbol{x}_i, y_i)$ in the support set $D^{tr}_{\mathcal{T}_j}$. The base-learner $b_{\boldsymbol{\theta}}$ attempts to make class predictions for these examples, resulting in loss values $\mathcal{L}_i$ (line 7-8). The gradients of these losses are used to compute fast weights $\boldsymbol{\theta}^*$ for example i (line 8), which are then stored in the i-th row of memory matrix M (line 9). Additionally, input representations $\boldsymbol{r}_i$ are computed and stored in memory matrix R (lines 10-11).

Now, meta networks are ready to address the query set $D^{test}_{\mathcal{T}_j}$. They iterate over every example $(\boldsymbol{x}, y)$, and compute a representation $\boldsymbol{r}$ of it (line 15). This representation is matched against the representations of the support set, which are stored in memory matrix R. This matching gives us a similarity vector $\boldsymbol{a}$, where every entry k denotes the similarity between input representation $\boldsymbol{r}$ and the k-th row in memory matrix R, i.e., $R(k)$ (line 16). A softmax over this similarity vector is performed to normalize the entries. The resulting vector is used to compute a linear combination of weights that were generated for inputs in the support set (line 17). These weights $\boldsymbol{\theta}^*$ are specific for input $\boldsymbol{x}$ in the query set, and can be used by the base-learner b to make predictions for that input (line 18). The observed error is added to the task loss. After the entire query set is processed, all involved parameters can be updated using backpropagation (line 20).

Note that some neural networks use both slow- and fast-weights at the same time. Munkhdalai and Yu (2017) use a so-called augmentation setup for this, as depicted in Figure 2.16.

Algorithm 1 Meta networks, by Munkhdalai and Yu (2017)

Sample $S = \{(\boldsymbol{x}_i, y_i) \backsim D_{\mathcal{T}_j}^{tr}\}_{i=1}^{T}$ from the support set
for $(\boldsymbol{x}_i, y_i) \in S$ **do**
    $\mathcal{L}_i = \text{error}(u_{\boldsymbol{\phi}}(\boldsymbol{x}_i), y_i)$
end for
$\boldsymbol{\phi}^* = d_{\boldsymbol{\psi}}(\{\nabla_{\boldsymbol{\phi}} \mathcal{L}_i\}_{i=1}^{T})$
for $(\boldsymbol{x}_i, y_i) \in D_{\mathcal{T}_j}^{tr}$ **do**
    $\mathcal{L}_i = \text{error}(b_{\boldsymbol{\theta}}(\boldsymbol{x}_i), y_i)$
    $\boldsymbol{\theta}_i^* = m_{\boldsymbol{\varphi}}(\nabla_{\boldsymbol{\theta}} \mathcal{L}_i)$
    Store $\boldsymbol{\theta}_i^*$ in i-th position of example-level weight memory M
    $\boldsymbol{r}_i = u_{\boldsymbol{\phi}, \boldsymbol{\phi}^*}(\boldsymbol{x}_i)$
    Store $\boldsymbol{r}_i$ in i-th position of representation memory R
end for
$\mathcal{L}_{task} = 0$
for $(\boldsymbol{x}, y) \in D_{\mathcal{T}_j}^{test}$ **do**
    $\boldsymbol{r} = u_{\boldsymbol{\phi}, \boldsymbol{\phi}^*}(\boldsymbol{x})$
    $\boldsymbol{a} = \text{attention}(R, \boldsymbol{r})$ ▷ a_k is the cosine similarity between $\boldsymbol{r}$ and $R(k)$
    $\boldsymbol{\theta}^* = \text{softmax}(\boldsymbol{a})^T M$
    $\mathcal{L}_{task} = \mathcal{L}_{task} + \text{error}(b_{\boldsymbol{\theta}, \boldsymbol{\theta}^*}(\boldsymbol{x}), y)$
end for
Update $\Theta = \{\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\psi}, \boldsymbol{\varphi}\}$ using $\nabla_{\Theta} \mathcal{L}_{task}$

In short, meta networks rely on a reparameterization of the meta- and base-learner for every task. Despite the flexibility and applicability to both supervised and reinforcement learning settings, the approach is quite complex. It consists of many components, each with its own set of parameters, which can be a burden on memory-usage and computation time. Additionally, finding the correct architecture for all the involved components can be time consuming.

2.4.5 Simple neural attentive meta-learner (SNAIL)

Instead of an external memory matrix, SNAIL (Mishra et al., 2018) relies on a special model architecture to serve as memory. Mishra et al. (2018) argue that it is not possible to use Recurrent Neural Networks for this, as they have limited memory capacity, and cannot pinpoint specific prior experiences (Mishra et al., 2018). Hence, SNAIL uses a different architecture, consisting of 1D *temporal convolutions* (Oord et al., 2016) and a *soft attention* mechanism (Vaswani et al., 2017). The temporal convolutions allow for 'high band-width' memory access, and the attention mechanism allows to pinpoint specific experiences. Figure 2.17 visualizes the architecture and workflow of SNAIL for supervised learning problems. From this figure, it becomes clear why this technique is model-based. That is, model outputs are based upon the internal state, computed from earlier inputs.

SNAIL consists of three building blocks. The first is the *DenseBlock*, which applies a single 1D convolution to the input, and concatenates (in the feature/horizontal direction) the result. The second is a *TCBlock*, which is simply a series of DenseBlocks with exponentially increasing dilation rate of the temporal convolutions (Mishra et al., 2018). Note that the dilation is nothing but the temporal distance between two nodes in a network. For example, if we use a dilation of 2, a node at position p in layer L will receive the activation from node $p - 2$ from layer $L - 1$. The third block is the *AttentionBlock*, which learns to focus on the important parts of prior experience.

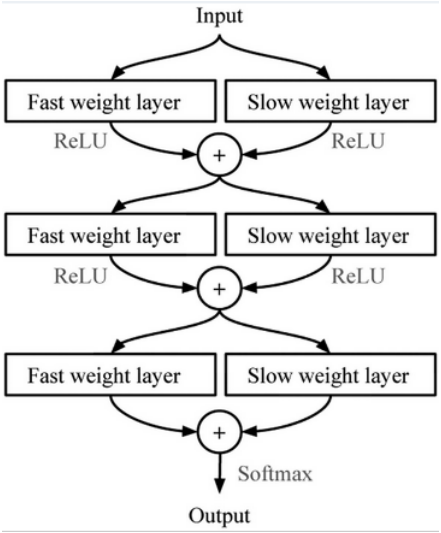


Figure 2.16: Layer augmentation setup used to combine slow and fast weights. Source: Munkhdalai and Yu (2017).

In similar fashion to memory-augmented neural networks (Santoro et al., 2016) (Section 2.4.3), SNAIL also processes task data in sequence, as shown in Figure 2.17. However, the input at time t is accompanied with the label at time t, instead of $t-1$ (as was the case for memory-augmented neural networks). SNAIL learns internal dynamics from seeing various tasks, so that it can make good predictions on the query set, conditioned upon the support set.

A key advantage of SNAIL is that it can be applied to both supervised and reinforcement learning tasks. In addition, it achieves good performance compared to previously discussed techniques. A downside of SNAIL is that finding the correct architecture of TCBlocks and DenseBlocks can be time consuming.

2.4.6 Conditional neural processes (CNPs)

In contrast to previous techniques, a conditional neural process (CNP) (Garnelo et al., 2018) does not rely on an external memory module. Instead, it aggregates the support set into a single aggregated latent representation. The general architecture is shown in Figure 2.18. As we can see, the conditional neural process operates in three phases on task $\mathcal{T}_j$. First, it observes the support set $D^{tr}_{\mathcal{T}_j}$, including the ground-truth outputs y_i. Examples $(\boldsymbol{x}_i, y_i) \in D^{tr}_{\mathcal{T}_j}$ are embedded using a neural network $h_{\boldsymbol{\theta}}$ into representations $\boldsymbol{r}_i$. Second, these representations are aggregated using operator a to produce a single representation $\boldsymbol{r}$ of $D^{tr}_{\mathcal{T}_j}$ (hence it is model-based). Third, a neural network $g_{\boldsymbol{\phi}}$ processes this single representation $\boldsymbol{r}$, new inputs $\boldsymbol{x}$, and produces predictions $\hat{y}$.

Let the entire conditional neural process model be denoted by $Q_{\boldsymbol{\Theta}}$, where Θ is a set of all involved parameters $\{\boldsymbol{\theta}, \boldsymbol{\phi}\}$. The training process is different compared to other techniques. Let $\boldsymbol{x}_{\mathcal{T}_j}$ and $\boldsymbol{y}_{\mathcal{T}_j}$ denote all inputs and corresponding outputs in $D^{tr}_{\mathcal{T}_j}$. Then, the first $\ell \backsim U(0, \ldots, k \cdot N - 1)$ examples in $D^{tr}_{\mathcal{T}_j}$ are used as a conditioning set $D^{c}_{\mathcal{T}_j}$ (effectively splitting the support set in a true training set and a validation set). Given a value of ℓ, the goal is to maximize the log likelihood (or minimize the

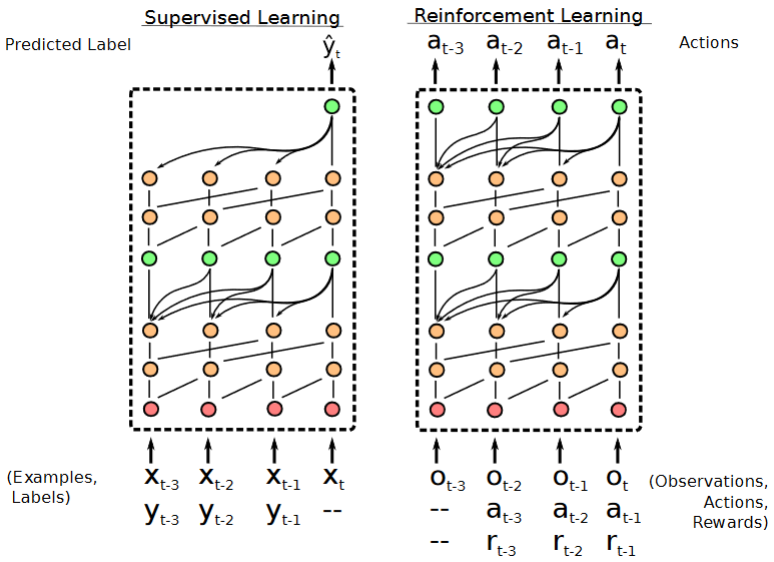


Figure 2.17: Architecture and workflow of SNAIL for supervised and reinforcement learning settings. The input layer is red. Temporal Convolution blocks are orange; attention blocks are green. Source: Mishra et al. (2018).

negative log likelihood) of the labels $\boldsymbol{y}_{\mathcal{T}_j}$ in the entire support set $D^{tr}_{\mathcal{T}_j}$

$$\mathcal{L}(\boldsymbol{\Theta}) = -\mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})} \left[\mathbb{E}_{\ell \backsim U(0,\ldots,k \cdot N-1)} \left(Q_{\boldsymbol{\Theta}}(\boldsymbol{y}_{\mathcal{T}_j} | D^{c}_{\mathcal{T}_j}, \boldsymbol{x}_{\mathcal{T}_j}) \right) \right]. \tag{2.12}$$

Conditional neural processes are trained by repeatedly sampling various tasks and values of ℓ, and propagating the observed loss backwards.

In summary, conditional neural processes use compact representations of previously seen inputs to aid the classification of new observations. Despite its simplicity and elegance, a disadvantage of this technique is that it is often outperformed in few-shot settings by other techniques such as matching networks (Vinyals et al., 2016) (see Section 2.3.3).

2.4.7 Neural statistician

A neural statistician (Edwards and Storkey, 2017) differs from earlier approaches as it learns to compute *summary statistics*, or *meta-features*, of data sets in an unsupervised manner. These latent embeddings (making the approach model-based) can then later be used for making predictions. Despite the broad applicability of the model, we discuss it in the context of deep meta-learning.

A neural statistician performs both *learning* and *inference*. In the learning phase, the model attempts to produce generative models $\hat{P}_i$ for every data set D_i. The key assumption that is made by Edwards and Storkey (2017) is that there exists a generative process P_i, which conditioned on a latent context vector $\boldsymbol{c}_i$, can produce data set D_i. At inference time, the goal is to infer a (posterior) probability

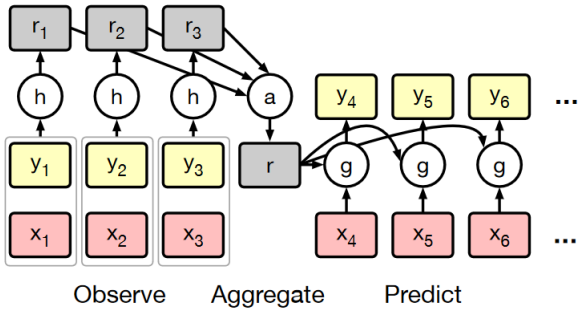


Figure 2.18: Schematic view of how conditional neural processes work. Here, h denotes a network outputting a representation for a observation, a denotes an aggregation function for these representations, and g denotes a neural network that makes predictions for unlabelled observations, based on the aggregated representation. Source: Garnelo et al. (2018).

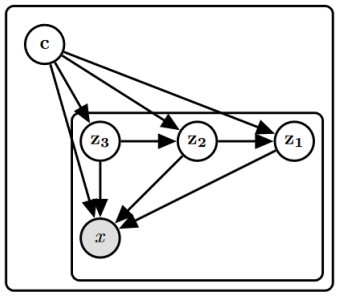


Figure 2.19: Neural statistician architecture. Edges are neural networks. All incoming inputs to a node are concatenated.

distribution over the context $q(\boldsymbol{c}|D)$.

The model uses a variational autoencoder, which consists of an encoder and decoder. The encoder is responsible for producing a distribution over latent vectors $\boldsymbol{z}$: $q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{\phi})$, where $\boldsymbol{x}$ is an input vector, and $\boldsymbol{\phi}$ are the encoder parameters. The encoded input $\boldsymbol{z}$, which is often of lower dimensionality than the original input $\boldsymbol{x}$, can then be decoded by the decoder $p(\boldsymbol{x}|\boldsymbol{z};\boldsymbol{\theta})$. Here, $\boldsymbol{\theta}$ are the parameters of the decoder. To capture more complex patterns in data sets, the model uses multiple latent layers $\boldsymbol{z}_1, ..., \boldsymbol{z}_L$, as shown in Figure 2.19. Given this architecture, the posterior over c and $\boldsymbol{z}_1, .., \boldsymbol{z}_L$ (shorthand $\boldsymbol{z}_{1:L}$) is given by

$$q(\boldsymbol{c}, \boldsymbol{z}_{1:L}|D;\boldsymbol{\phi}) = q(\boldsymbol{c}|D;\boldsymbol{\phi}) \prod_{\boldsymbol{x}\in D} q(z_L|\boldsymbol{x},\boldsymbol{c};\boldsymbol{\phi}) \prod_{i=1}^{L-1} q(\boldsymbol{z}_i|\boldsymbol{z}_{i+1},\boldsymbol{x},\boldsymbol{c};\boldsymbol{\phi}). \tag{2.13}$$

The neural statistician is trained to minimize a three-component loss function, consisting of the reconstruction loss (how well it models the data), context loss (how well the inferred context $q(\boldsymbol{c}|D;\boldsymbol{\phi})$ corresponds to the prior $P(\boldsymbol{c})$, and latent loss (how well the inferred latent variables $\boldsymbol{z}_i$ are modelled).

This model can be applied to N-way, few-shot learning as follows. Construct N data sets for every of the N classes, such that one data set contains only examples of the same class. Then, the neural statistician is provided with a new input $\boldsymbol{x}$, and has to predict its class. It computes a context posterior $N_{\boldsymbol{x}} = q(\boldsymbol{c}|\boldsymbol{x};\boldsymbol{\phi})$ depending on new input $\boldsymbol{x}$. In similar fashion, context posteriors are computed for all of the data sets $N_i = q(\boldsymbol{c}|D_i;\boldsymbol{\phi})$. Lastly, it assigns the label i such that the difference between N_i and $N_{\boldsymbol{x}}$ is minimal.

In summary, the neural statistician (Edwards and Storkey, 2017) allows for quick learning on new tasks through data set modeling. Additionally, it is applicable to both supervised and unsupervised settings. A downside is that the approach requires many data sets to achieve good performance (Edwards and Storkey, 2017).

2.4.8 Model-based techniques, in conclusion

In this section, we have discussed various model-based techniques. Despite apparent differences, they all build on the notion of task internalization. That is, tasks are processed and represented in the state of the model-based system. This state can then be used to make predictions. Figure 2.20 displays the relationships between the covered model-based techniques.

MANNs (Santoro et al., 2016) mark the beginning of the deep model-based meta-learning techniques. They use the idea of feeding the entire support set in sequential fashion into the model and then making predictions for the query set inputs using the internal state of the model. Such a model-based approach, where inputs sequentially enter the model was also taken by RMLs (Duan et al., 2016; Wang et al., 2016) in the reinforcement learning setting. Meta networks (Munkhdalai and Yu, 2017) also use a large black-box solution, but generate task-specific weights for every task that is encountered. SNAIL (Mishra et al., 2018) tries to improve the memory capacity and ability to pinpoint memories, which is limited in recurrent neural networks, by using attention mechanisms coupled with special temporal layers. Lastly, the neural statistician and CNP are two techniques that try to learn meta-features of data sets in an end-to-end fashion. The neural statistician uses the distance between meta-features to make class predictions, while the CNP conditions classifiers on these features.

Advantages of model-based approaches include the flexibility of the internal dynamics of the systems, and their broader applicability compared to most metric-based techniques. However, model-based techniques are often outperformed by metric-based techniques in supervised settings (e.g. graph neural networks (Garcia and Bruna, 2017); Section 2.3.6), may not perform well when presented with larger data sets (Hospedales et al., 2021), and generalize less well to more distant tasks than optimization-based techniques (Finn and Levine, 2018). We discuss this optimization-based approach next.

2.5 Optimization-based meta-learning

Optimization-based techniques adopt a different perspective on meta-learning than the previous two approaches. They explicitly optimize for fast learning. Most optimization-based techniques do so by approaching meta-learning as a bi-level optimization problem. At the inner-level, a base-learner makes task-specific updates using some optimization strategy (such as gradient descent). At the outer-level, the performance across tasks is optimized.

More formally, given a task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j})$ with new input $\boldsymbol{x} \in D^{test}_{\mathcal{T}_j}$ and base-learner parameters

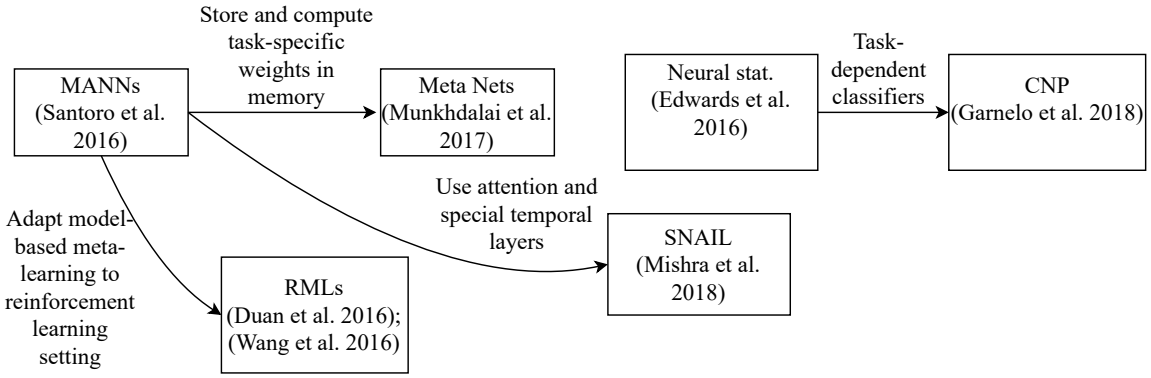


Figure 2.20: The relationships between the covered model-based meta-learning techniques. The neural statistician and CNP form an island in the model-based approaches.

$\boldsymbol{\theta}$, optimization-based meta-learners return

$$p(Y|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j}) = f_{g_{\boldsymbol{\varphi}(\boldsymbol{\theta}, D^{tr}_{\mathcal{T}_j}, \mathcal{L}_{\mathcal{T}_j})}}(\boldsymbol{x}), \tag{2.14}$$

where f is the base-learner, $g_{\boldsymbol{\varphi}}$ is a (learned) optimizer that makes task-specific updates to the base-learner parameters $\boldsymbol{\theta}$ using the support data $D^{tr}_{\mathcal{T}_i}$, and loss function $\mathcal{L}_{\mathcal{T}_j}$.

2.5.1 Example

Suppose we are faced with a linear regression problem, where every task is associated with a different function $f(x)$. For this example, suppose our model only has two parameters: a and b, which together form the function $\hat{f}(x) = ax + b$. Suppose further that our meta-training set consists of four different tasks, i.e., A, B, C, and D. Then, according to the optimization-based view, we wish to find a single set of parameters $\{a, b\}$ from which we can quickly learn the optimal parameters for each of the four tasks, as displayed in Figure 2.21. In fact, this is the intuition behind the popular optimization-based technique MAML (Finn et al., 2017). By exposing our model to various meta-training tasks, we can update the parameters a and b to facilitate quick adaptation.

We will now discuss the core optimization-based techniques in more detail.

2.5.2 LSTM optimizer

Standard gradient update rules have the form

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_t), \tag{2.15}$$

where α is the learning rate, and $\mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_t)$ is the loss function with respect to task $\mathcal{T}_j$ and network parameters at time t, i.e., $\boldsymbol{\theta}_t$. The key idea underlying LSTM optimizers (Andrychowicz et al., 2016) is to replace the update term $(-\alpha \nabla \mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_t))$ by an update proposed by an LSTM g with parameters $\boldsymbol{\varphi}$. Then, the new update becomes

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + g_{\boldsymbol{\varphi}}(\nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_t)). \tag{2.16}$$

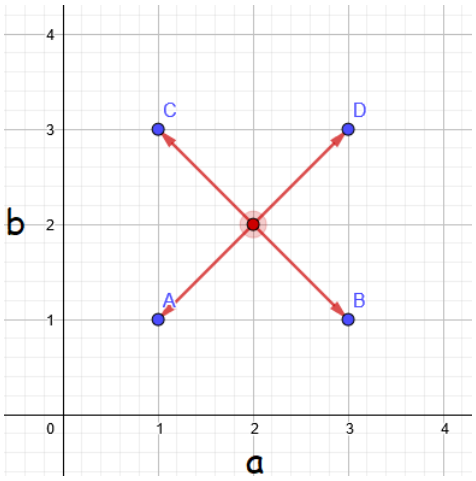


Figure 2.21: Example of an optimization-based technique, inspired by Finn et al. (2017).

This new update allows the optimization strategy to be tailored to a specific family of tasks. Note that this is meta-learning, i.e., the LSTM learns to learn. As such, this technique basically learns an update policy.

The loss function used to train an LSTM optimizer is:

$$\mathcal{L}(\boldsymbol{\varphi}) = \mathbb{E}_{\mathcal{L}_{\mathcal{T}_j}} \left[\sum_{t=1}^{T} w_t \mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_t) \right], \tag{2.17}$$

where T is the number of parameter updates that are made, and w_t are weights indicating the importance of performance after t steps. Note that generally we are only interested in the final performance after T steps. However, the authors found that the optimization procedure was better guided by equally weighting the performance after each gradient descent step. As is often done, second-order derivatives (arising from the dependency between the updated weights and the LSTM optimizer) were ignored due to the computational expenses associated with the computation thereof. This loss function is fully differentiable, and thus allows for training an LSTM optimizer (see Figure 2.22). To prevent a parameter explosion, the same network is used for every *coordinate*/weight in the base-learner's network, causing the update rule to be the same for every parameter. Of course, the updates depend on their prior values and gradients.

The key advantage of LSTM optimizers is that they can enable faster learning compared to hand-crafted optimizers, also on different data sets than those used to train the optimizer. However, Andrychowicz et al. (2016) did not apply this technique to few-shot learning. In fact, they did not apply it across tasks at all. Thus, it is unclear whether this technique can perform well in few-shot settings, where few data per class are available for training. Furthermore, the question remains whether it can scale to larger base-learner architectures.

2.5.3 LSTM meta-learner

Instead of having an LSTM predict gradient updates, Ravi and Larochelle (2017) embed the weights of the base-learner parameters into the cell state (long-term memory component) of the LSTM, giving

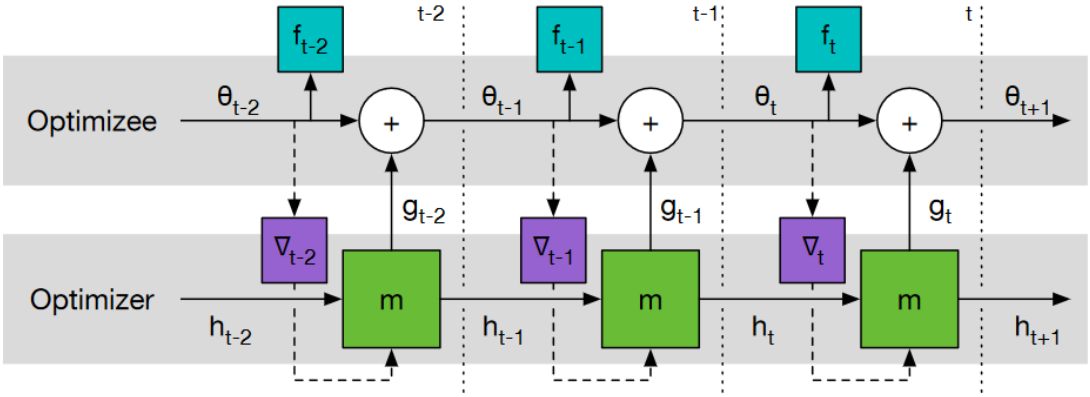


Figure 2.22: Workflow of the LSTM optimizer. Gradients can only propagate backwards through solid edges. f_t denotes the observed loss at time step t. Source: Andrychowicz et al. (2016).

rise to LSTM meta-learners. As such, the base-learner parameters $\boldsymbol{\theta}$ are literally inside the LSTM memory component (cell state). In this way, cell state updates correspond to base-learner parameter updates. This idea was inspired by the resemblance between the gradient and cell state update rules. Gradient updates often have the form as shown in Equation 2.15. The LSTM cell state update rule, in contrast, looks as follows

$$\boldsymbol{c}_t := f_t \odot \boldsymbol{c}_{t-1} + \alpha_t \odot \bar{\boldsymbol{c}}_t, \tag{2.18}$$

where f_t is the forget gate (which determines which information should be forgotten) at time t, $\odot$ represents the element-wise product, $\boldsymbol{c}_t$ is the cell state at time t, and $\bar{\boldsymbol{c}}_t$ the candidate cell state for time step t, and α_t the learning rate at time step t. Note that if $f_t = \mathbf{1}$ (vector of ones), $\alpha_t = \alpha$, $\boldsymbol{c}_{t-1} = \boldsymbol{\theta}_{t-1}$, and $\bar{\boldsymbol{c}}_t = -\nabla_{\boldsymbol{\theta}_{t-1}} \mathcal{L}_{\mathcal{T}_t}(\boldsymbol{\theta}_{t-1})$, this update is equivalent to the one used by gradient-descent. This similarity inspired Ravi and Larochelle (2017) to use an LSTM as meta-learner that learns to make updates for a base-learner, as shown in Figure 2.23.

More specifically, the cell state of the LSTM is initialized with $c_0 = \boldsymbol{\theta}_0$, which will be adjusted by the LSTM to a good common initialization point across different tasks. Then, to update the weights of the base-learner for the next time step $t+1$, the LSTM computes $\boldsymbol{c}_{t+1}$, and sets the weights of the base-learner equal to that. There is thus a one-to-one correspondence between $\boldsymbol{c}_t$ and $\boldsymbol{\theta}_t$. The meta-learner's learning rate α_t (see Equation 2.18), is set equal to $\sigma(\boldsymbol{w}_\alpha \cdot [\nabla_{\theta_{t-1}} \mathcal{L}_{\mathcal{T}_t}(\boldsymbol{\theta}_{t-1}), \mathcal{L}_{\mathcal{T}_t}(\boldsymbol{\theta}_t), \theta_{t-1}, \alpha_{t-1}] + \boldsymbol{b}_\alpha)$, where σ is the sigmoid function. Note that the output is a vector, with values between 0 and 1, which denote the the learning rates for the corresponding parameters. Furthermore, $\boldsymbol{w}_\alpha$ and $\boldsymbol{b}_\alpha$ are trainable parameters that part of the LSTM meta-learner. In words, the learning rate at any time depends on the loss gradients, the loss value, the previous parameters, and the previous learning rate. The forget gate, f_t, determines what part of the cell state should be forgotten, and is computed in a similar fashion, but with different weights.

To prevent an explosion of meta-learner parameters, weight-sharing is used, in similar fashion to LSTM optimizers proposed by Andrychowicz et al. (2016) (Section 2.5.2). This implies that the same update rule is applied to every weight at a given time step. The exact update, however, depends on the history of that specific parameter in terms of previous learning rate, loss, etc. For simplicity,

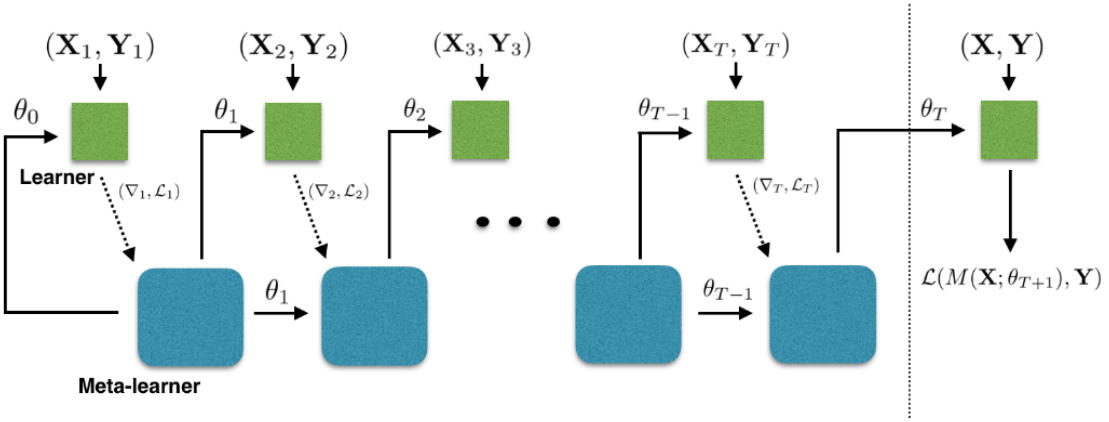


Figure 2.23: LSTM meta-learner computation graph. Gradients can only propagate backwards through solid edges. The base-learner is denoted as M. (X_t, Y_t) are training sets, whereas (X, Y) is the test set. Source: Ravi and Larochelle (2017).

second-order derivatives were ignored, by assuming the base-learner's loss does not depend on the cell state of the LSTM optimizer. Batch normalization was applied to stabilize and speed up the learning process.

In short, LSTM optimizers can learn to optimize a base-learner by maintaining a one-to-one correspondence over time between the base-learner's weights and the LSTM cell state. This allows the LSTM to exploit commonalities in the tasks, allowing for quicker optimization. However, there are simpler approaches (e.g. MAML (Finn et al., 2017)) that outperform this technique.

2.5.4 Reinforcement learning optimizer

Li and Malik (2018) proposed a framework which casts optimization as a reinforcement learning problem. Optimization can then be performed by existing reinforcement learning techniques. At a high-level, an optimization algorithm g takes as input an initial set of weights $\boldsymbol{\theta}_0$ and a task $\mathcal{T}_j$ with corresponding loss function $\mathcal{L}_{\mathcal{T}_j}$, and produces a sequence of new weights $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_T$, where $\boldsymbol{\theta}_T$ is the final solution found. On this sequence of proposed new weights, we can define a loss function $\mathcal{L}$ that captures unwanted properties (e.g. slow convergence, oscillations, etc.). The goal of learning an optimizer can then be formulated more precisely as follows. We wish to learn an optimal optimizer

$$g^* = argmin_g \, \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T}), \boldsymbol{\theta}_0 \backsim p(\boldsymbol{\theta}_0)} [\mathcal{L}(g(\mathcal{L}_{\mathcal{T}_j}, \boldsymbol{\theta}_0))] \tag{2.19}$$

The key insight is that the optimization can be formulated as a Partially Observable Markov Decision Process (POMDP). Then, the state corresponds to the current set of weights $\boldsymbol{\theta}_t$, the action to the proposed update at time step t, i.e., $\Delta\boldsymbol{\theta}_t$, and the policy to the function that computes the update. With this formulation, the optimizer g can be learned by existing reinforcement learning techniques. In their paper, they used an recurrent neural network as optimizer. At each time step, they feed it observation features, which depend on the previous set of weights, loss gradients, and objective functions, and use guided policy search to train it.

In summary, Li and Malik (2018) made a first step towards general optimization through reinforcement learning optimizers, which were shown able to generalize across network architectures and data sets. However, the base-learner architecture that was used was quite small. The question remains whether this approach can scale to larger architectures.

2.5.5 MAML

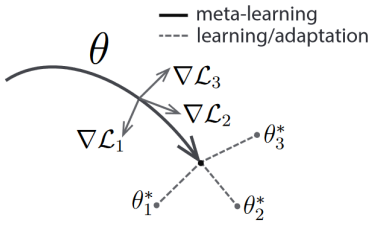


Figure 2.24: MAML learns an initialization point from which it can perform well on various tasks. Source: Finn et al. (2017).

Model-agnostic meta-learning (MAML) (Finn et al., 2017) uses a simple gradient-based inner optimization procedure (e.g. stochastic gradient descent), instead of more complex LSTM procedures or procedures based on reinforcement learning. The key idea of MAML is to explicitly optimize for fast adaptation to new tasks by learning a good set of initialization parameters $\boldsymbol{\theta}$. This is shown in Figure 2.24: from the learned initialization $\boldsymbol{\theta}$, we can quickly move to the best set of parameters for task $\mathcal{T}_j$, i.e., $\boldsymbol{\theta}_j^*$ for $j = 1, 2, 3$. The learned initialization can be seen as the *inductive bias* of the model, or simply the set of assumptions (encapsulated in $\boldsymbol{\theta}$) that the model makes with respect to the overall task structure.

More formally, let $\boldsymbol{\theta}$ denote the initial model parameters of a model. The goal is to quickly learn new concepts, which is equivalent to achieving a minimal loss in few gradient update steps. The amount of gradient steps s has to be specified upfront, such that MAML can explicitly optimize for achieving good performance within that number of steps. Suppose we pick only one gradient update step, i.e., $s = 1$. Then, given a task $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{test})$, gradient descent would produce updated parameters (fast weights)

$$\boldsymbol{\theta}_j' = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\boldsymbol{\theta}), \tag{2.20}$$

specific to task j. The *meta-loss* of quick adaptation (using $s = 1$ gradient steps) across tasks can then be formulated as

$$ML := \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}_{D_{\mathcal{T}_j}^{test}}(\boldsymbol{\theta}_j') = \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}_{D_{\mathcal{T}_j}^{test}}(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\boldsymbol{\theta})), \tag{2.21}$$

where $p(\mathcal{T})$ is a probability distribution over tasks. This expression contains an inner gradient ($\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_j)$). As such, by optimizing this meta-loss using gradient-based techniques, we have to compute second-order gradients. One can easily see this in the computation below

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} ML &= \nabla_{\boldsymbol{\theta}} \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \\
&= \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \\
&= \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}'_j) \\
&= \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}(\boldsymbol{\theta})}) \\
&= \underbrace{\sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)(\nabla_{\boldsymbol{\theta}} \boldsymbol{\theta} - \alpha \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}))}_{\text{FOMAML}},
\end{aligned} \tag{2.22}
$$

where we used $\mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)$ to denote the derivative of the loss function with respect to the query set, evaluated at the post-update parameters $\boldsymbol{\theta}'_j$. The term $\alpha \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})$ contains the second-order gradients. The computation thereof is expensive in terms of time and memory costs, especially when the optimization trajectory is large (when using a larger number of gradient updates s per task). Finn et al. (2017) experimented with leaving out second-order gradients, by assuming $\nabla_{\boldsymbol{\theta}} \boldsymbol{\theta}'_j = I$, giving us First Order MAML (FOMAML, see Equation 2.22). They found that FOMAML performed reasonably similar to MAML. This means that updating the initialization using only first order gradients $\sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)$ is roughly equal to using the full gradient expression of the meta-loss in Equation 2.22. One can extend the meta-loss to incorporate multiple gradient steps by substituting $\boldsymbol{\theta}'_j$ by a multi-step variant.

MAML is trained as follows. The initialization weights $\boldsymbol{\theta}$ are updated by continuously sampling a batch of m tasks $B = \{\mathcal{T}_j \backsim p(\mathcal{T})\}_{i=1}^{m}$. Then, for every task $\mathcal{T}_j \in B$, an *inner update* is performed to obtain $\boldsymbol{\theta}'_j$, in turn granting an observed loss $\mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)$. These losses across a batch of tasks are used in the *outer update*

$$
\boldsymbol{\theta} := \boldsymbol{\theta} - \beta \nabla_{\boldsymbol{\theta}} \sum_{\mathcal{T}_j \in B} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j). \tag{2.23}
$$

The complete training procedure of MAML is displayed in Algorithm 2. At test-time, when presented with a new task $\mathcal{T}_j$, the model is initialized with $\boldsymbol{\theta}$, and performs a number of gradient updates on the task data. Note that the algorithm for FOMAML is equivalent to Algorithm 2, except for the fact that the update on line 8 is done differently. That is, FOMAML updates the initialization with the rule $\boldsymbol{\theta} = \boldsymbol{\theta} - \beta \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)$.

Antoniou et al. (2019), in response to MAML, proposed many technical improvements that can improve training stability, performance, and generalization ability. Improvements include i) updating the initialization $\boldsymbol{\theta}$ after every inner update step (instead of after all steps are done) to increase gradient propagation, ii) using second-order gradients only after 50 epochs to increase the training speed, iii) learning layer-wise learning rates to improve flexibility, iv) annealing the meta-learning rate

Algorithm 2 One-step MAML for supervised learning, by Finn et al. (2017)

Randomly initialize $\boldsymbol{\theta}$
while not done **do**
  Sample batch of J tasks $B = \mathcal{T}_1, \ldots, \mathcal{T}_J \backsim p(\mathcal{T})$
  for $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j}) \in B$ **do**
    Compute $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})$
    Compute $\boldsymbol{\theta}'_j = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})$
  end for
  Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \beta \nabla_{\boldsymbol{\theta}} \sum_{\mathcal{T}_j \in B} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)$
end while

β over time, and v) some Batch Normalization tweaks (keep running statistics instead of batch-specific ones, and using per-step biases).

MAML has obtained great attention within the field of deep meta-learning, perhaps due to its i) simplicity (only requires two hyperparameters), ii) general applicability, and iii) strong performance. A downside of MAML, as mentioned above, is that it can be quite expensive in terms of running time and memory to optimize a base-learner for every task and compute higher-order derivatives from the optimization trajectories.

2.5.6 iMAML

Instead of ignoring higher-order derivatives (as done by FOMAML), which potentially decreases the performance compared to regular MAML, iMAML (Rajeswaran et al., 2019) approximates these derivatives in a way that is less memory-consuming.

Let $\mathcal{A}$ denote an inner optimization algorithm (e.g., stochastic gradient descent), which takes a support set $D^{tr}_{\mathcal{T}_j}$ corresponding to task $\mathcal{T}_j$ and initial model weights $\boldsymbol{\theta}$, and produces new weights $\boldsymbol{\theta}'_j = \mathcal{A}(\boldsymbol{\theta}, D^{tr}_{\mathcal{T}_j})$. MAML has to compute the derivative

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) = \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}'_j), \tag{2.24}$$

where $D^{test}_{\mathcal{T}_j}$ is the query set corresponding to task $\mathcal{T}_j$. This equation is a simple result of applying the chain rule. Importantly, note that $\nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}'_j)$ differentiates through $\mathcal{A}(\boldsymbol{\theta}, D^{tr}_{\mathcal{T}_j})$, while $\mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)$ does not, as it represents the gradient of the loss function evaluated at $\boldsymbol{\theta}'_j$. Rajeswaran et al. (2019) make use of the following lemma.

If $(\boldsymbol{I} + \frac{1}{\lambda} \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j))$ *is invertible (i.e.,* $(\boldsymbol{I} + \frac{1}{\lambda} \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j))^{-1}$ *exists), then*

$$\nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}'_j) = \left(\boldsymbol{I} + \frac{1}{\lambda} \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \right)^{-1}. \tag{2.25}$$

Here, λ is a regularization parameter. The reason for this is discussed below.

Combining Equation 2.24 and Equation 2.25, we have that

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) = \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \left(\boldsymbol{I} + \frac{1}{\lambda} \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \right)^{-1}. \tag{2.26}$$

The idea is to obtain an approximate gradient vector $\boldsymbol{g}_j$ that is close to this expression, i.e., we want the difference to be small

$$\boldsymbol{g}_j - \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \left(\boldsymbol{I} + \frac{1}{\lambda} \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \right)^{-1} = \boldsymbol{\epsilon}, \tag{2.27}$$

for some small tolerance vector $\boldsymbol{\epsilon}$. If we multiply both sides by the inverse of the inverse, i.e., $\left(\boldsymbol{I} + \frac{1}{\lambda} \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \right)$, we get

$$\boldsymbol{g}_j^T \left(\boldsymbol{I} + \frac{1}{\lambda} \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \right) \boldsymbol{g}_j - \boldsymbol{g}_j^T \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) = \boldsymbol{\epsilon}', \tag{2.28}$$

where $\boldsymbol{\epsilon}'$ absorbed the multiplication factor. We wish to minimize this expression for $\boldsymbol{g}_j$, and that can be performed using optimization techniques such as the conjugate gradient algorithm (Rajeswaran et al., 2019). This algorithm does not need to store Hessian matrices, which decreases the memory cost significantly. In turn, this allows iMAML to work with more inner gradient update steps. Note, however, that one needs to perform explicit regularization in that case to avoid overfitting. The conventional MAML did not require this, as it uses only a few number of gradient steps (equivalent to an early stopping mechanism).

At each inner loop step, iMAML computes the meta-gradient $\boldsymbol{g}_j$. After processing a batch of tasks, these gradients are averaged and used to update the initialization $\boldsymbol{\theta}$. Since it does not differentiate through the optimization process, we are free to use any other (non-differentiable) inner-optimizer.

In summary, iMAML reduces memory costs significantly as it need not differentiate through the optimization trajectory, also allowing for greater flexibility in the choice of inner optimizer. Additionally, it can account for larger optimization paths. The computational costs stay roughly the same compared to MAML (Finn et al., 2017). Future work could investigate more inner optimization procedures (Rajeswaran et al., 2019).

2.5.7 Meta-SGD

Meta-SGD (Li et al., 2017), or meta-stochastic gradient descent, is similar to MAML (Finn et al., 2017) (Section 2.5.5). However, on top of learning an initialization, Meta-SGD also learns learning rates for every model parameter in $\boldsymbol{\theta}$, building on the insight that the optimizer can be seen as trainable entity.

The standard SGD update rule is given in Equation 2.15. The meta-SGD optimizer uses a more general update, namely

$$\boldsymbol{\theta}'_j \leftarrow \boldsymbol{\theta} - \boldsymbol{\alpha} \odot \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}), \tag{2.29}$$

where $\odot$ is the element-wise product. Note that this means that alpha (learning rate) is now a vector—hence the bold font— instead of scalar, which allows for greater flexibility in the sense that

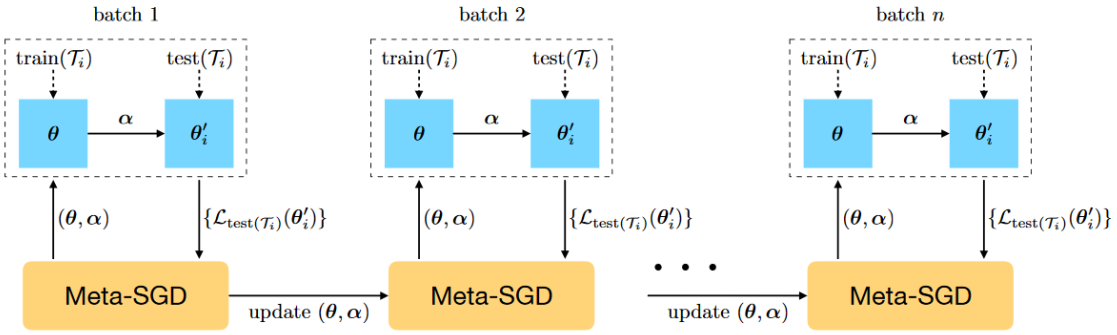


Figure 2.25: Meta-SGD learning process. Source: Li et al. (2017).

each parameter has its own learning rate. The goal is to learn the initialization $\boldsymbol{\theta}$, and learning rate vector $\boldsymbol{\alpha}$, such that the generalization ability is as large as possible. More mathematically precise, the learning objective is

$$min_{\boldsymbol{\alpha},\boldsymbol{\theta}} \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)] = \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta} - \boldsymbol{\alpha} \odot \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}))], \tag{2.30}$$

where we used a simple substitution for $\boldsymbol{\theta}'_j$. $\mathcal{L}_{D^{tr}_{\mathcal{T}_j}}$ and $\mathcal{L}_{D^{test}_{\mathcal{T}_j}}$ are the losses computed on the support and query set respectively. Note that this formulation stimulates generalization ability (as it includes the query set loss $\mathcal{L}_{D^{test}_{\mathcal{T}_j}}$, which can be observed during the meta-training phase). The learning process is visualized in Figure 2.25. Note that the meta-SGD optimizer is trained to maximize generalization ability after only one update step. Since this learning objective has a fully differentiable loss function, the meta-SGD optimizer itself can be trained using standard SGD.

In summary, Meta-SGD is more expressive than MAML as it does not only learn an initialization, but also learning rates per parameter. This, however, does come at the cost of an increased number of hyperparameters.

2.5.8 Reptile

Reptile (Nichol et al., 2018) is another optimization-based technique that, like MAML (Finn et al., 2017), solely attempts to find a good set of initialization parameters $\boldsymbol{\theta}$. The way in which Reptile attempts to find this initialization is quite different from MAML. It repeatedly samples a task, trains on the task, and moves the model weights towards the trained weights (Nichol et al., 2018). Algorithm 3 displays the pseudocode describing this simple process.

Algorithm 3 Reptile, by Nichol et al. (2018)

1: Initialize $\boldsymbol{\theta}$
2: **for** $i = 1, 2, \ldots$ **do**
3: Sample task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j})$ and corresponding loss function $\mathcal{L}_{\mathcal{T}_j}$
4: $\boldsymbol{\theta}'_j = SGD(\mathcal{L}_{D^{tr}_{\mathcal{T}_j}}, \boldsymbol{\theta}, k)$ ▷ Perform k gradient update steps to get $\boldsymbol{\theta}'_j$
5: $\boldsymbol{\theta} := \boldsymbol{\theta} + \epsilon(\boldsymbol{\theta}'_j - \boldsymbol{\theta})$ ▷ Move initialization point $\boldsymbol{\theta}$ towards $\boldsymbol{\theta}'_j$
6: **end for**

Nichol et al. (2018) note that it is possible to treat $(\boldsymbol{\theta} - \boldsymbol{\theta}_j')/\alpha$ as gradients, where α is the learning rate of the inner stochastic gradient descent optimizer (line 4 in the pseudocode), and to feed that into a meta-optimizer (e.g. Adam). Moreover, instead of sampling one task at a time, one could sample a batch of n tasks, and move the initialization $\boldsymbol{\theta}$ towards the average update direction $\bar{\boldsymbol{\theta}} = \frac{1}{n}\sum_{j=1}^{n}(\boldsymbol{\theta}_j' - \boldsymbol{\theta})$, granting the update rule $\boldsymbol{\theta} := \boldsymbol{\theta} + \epsilon\bar{\boldsymbol{\theta}}$.

The intuition behind Reptile is that updating the initialization weights towards updated parameters will grant a good inductive bias for tasks from the same family. By performing Taylor expansions of the gradients of Reptile and MAML (both first-order and second-order), Nichol et al. (2018) show that the expected gradients differ in their direction. They argue, however, that in practice, the gradients of Reptile will also bring the model towards a point minimizing the expected loss over tasks.

A mathematical argument as to why Reptile works goes as follows. Let $\boldsymbol{\theta}$ denote the initial parameters, and $\boldsymbol{\theta}_j^*$ the optimal set of weights for task $\mathcal{T}_j$. Lastly, let d be the Euclidean distance function. Then, the goal is to minimize the distance between the initialization point $\boldsymbol{\theta}$ and the optimal point $\boldsymbol{\theta}_j^*$, i.e.,

$$min_{\boldsymbol{\theta}}\, \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\frac{1}{2}d(\boldsymbol{\theta}, \boldsymbol{\theta}_j^*)^2]. \tag{2.31}$$

The gradient of this expected distance with respect to the initialization $\boldsymbol{\theta}$ is given by

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\frac{1}{2}d(\boldsymbol{\theta}, \boldsymbol{\theta}_j^*)^2] &= \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\frac{1}{2}\nabla_{\boldsymbol{\theta}}d(\boldsymbol{\theta}, \boldsymbol{\theta}_j^*)^2] \\ &= \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\boldsymbol{\theta} - \boldsymbol{\theta}_j^*],\end{aligned} \tag{2.32}$$

where we used the fact that the gradient of the squared Euclidean distance between two points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ is the vector $2(\boldsymbol{x}_1 - \boldsymbol{x}_2)$. Nichol et al. (2018) go on to argue that performing gradient descent on this objective would result in the following update rule

$$\begin{aligned}\boldsymbol{\theta} &= \boldsymbol{\theta} - \epsilon\nabla_{\boldsymbol{\theta}}\frac{1}{2}d(\boldsymbol{\theta}, \boldsymbol{\theta}_j^*)^2 \\ &= \boldsymbol{\theta} - \epsilon(\boldsymbol{\theta}_j^* - \boldsymbol{\theta}).\end{aligned} \tag{2.33}$$

Since we do not know $\boldsymbol{\theta}_{\mathcal{T}_j}^*$, one can approximate this by term by k steps of gradient descent $SGD(\mathcal{L}_{\mathcal{T}_j}, \boldsymbol{\theta}, k)$. In short, Reptile can be seen as gradient descent on the distance minimization objective given in Equation 2.31. A visualization is shown in Figure 2.26. The initialization $\boldsymbol{\theta}$ is moving towards the optimal weights for tasks 1 and 2 in interleaved fashion (hence the oscillations).

In conclusion, Reptile is an extremely simple meta-learning technique, which does not need to differentiate through the optimization trajectory like, e.g., MAML (Finn et al., 2017), saving time and memory costs. However, the theoretical foundation is a bit weaker due to the fact that it does not directly optimize for fast learning as done by MAML, and performance may be a bit worse than that of MAML in some settings.

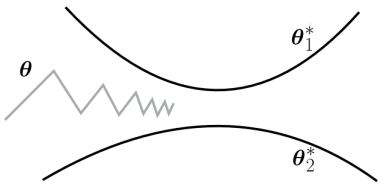


Figure 2.26: Schematic visualization of Reptile's learning trajectory. Here, $\boldsymbol{\theta}_1^*$ and $\boldsymbol{\theta}_2^*$ are the optimal weights for tasks $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. The initialization parameters $\boldsymbol{\theta}$ oscillate between these. Adapted from Nichol et al. (2018).

2.5.9 LEO

Latent Embedding Optimization, or LEO, was proposed by Rusu et al. (2019) to combat an issue of gradient-based meta-learners, such as MAML (Finn et al., 2017) (see Section 2.5.5), in few-shot settings (N-way, k-shot). These techniques operate in a high-dimensional parameter space using gradient information from only few examples, which could lead to poor generalization.

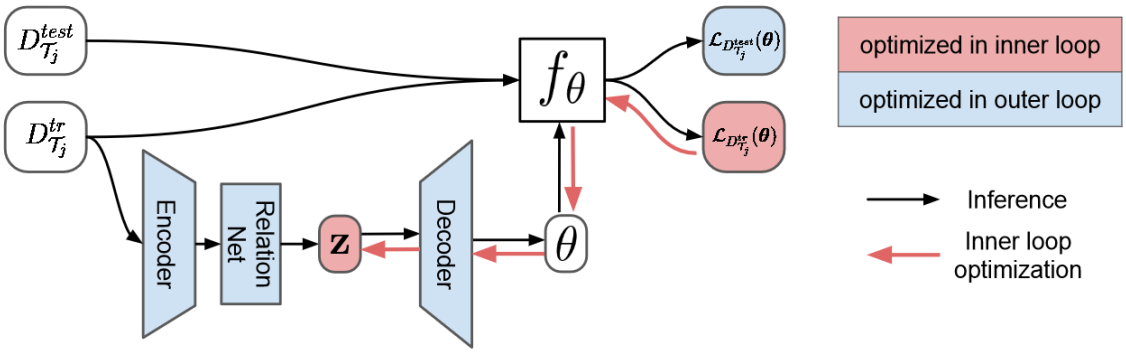


Figure 2.27: Workflow of Latent Embedding Optimization (LEO). adapted from Rusu et al. (2019).

LEO alleviates this issue by learning a lower-dimensional latent embedding space, which indirectly allows us to learn a good set of initial parameters $\boldsymbol{\theta}$. Additionally, the embedding space is conditioned upon tasks, allowing for more expressivity. In theory LEO could find initial parameters for the entire base-learner network, but the authors only experimented with setting the parameters for the final layers.

The complete workflow of LEO is shown in Figure 2.27. As we can see, given a task $\mathcal{T}_j$, the corresponding support set $D_{\mathcal{T}_j}^{tr}$ is fed into an encoder, which produces hidden codes for each example in that set. These hidden codes are paired and concatenated in every possible manner, granting us $(Nk)^2$ pairs, where N is the number of classes in the training set, and k the number of examples per class. These paired codes are then fed into a relation net (Sung et al., 2018) (see Section 2.3.5). The resulting embeddings are grouped by class, and parameterize a probability distribution over latent codes $\boldsymbol{z}_n$ (for class n) in a low dimensional space $\mathcal{Z}$. More formally, let $\boldsymbol{x}_n^\ell$ denote the ℓ-th example of class n in $D_{\mathcal{T}_j}^{tr}$. Then, the mean $\boldsymbol{\mu}_n^e$ and variance $\boldsymbol{\sigma}_n^e$ of a Gaussian distribution over latent codes for class n are computed as

$$\boldsymbol{\mu}_n^e, \boldsymbol{\sigma}_n^e = \frac{1}{Nk^2} \sum_{\ell_p=1}^{k} \sum_{m=1}^{N} \sum_{\ell_q=1}^{k} g_{\boldsymbol{\phi}_r} \left(g_{\boldsymbol{\phi}_e}(\boldsymbol{x}_n^{\ell_p}), g_{\boldsymbol{\phi}_e}(\boldsymbol{x}_m^{\ell_q}) \right), \tag{2.34}$$

where $\boldsymbol{\phi}_r, \boldsymbol{\phi}_e$ are parameters for the relation net and encoder respectively. Intuitively, the three summations ensure that every example with class n in $D_{\mathcal{T}_j}^{tr}$ is paired with every example from all classes n. Given $\boldsymbol{\mu}_n^e$, and $\boldsymbol{\sigma}_n^e$, one can sample a latent code $\boldsymbol{z}_n \backsim N(\boldsymbol{\mu}_n^e, diag(\boldsymbol{\sigma}_n^{e2}))$ for class n, which serves as latent embedding of the task training data.

The decoder can then generate a task-specific initialization $\boldsymbol{\theta}_n$ for class n as follows. First, one computes a mean and variance for a Gaussian distribution using the latent code

$$\boldsymbol{\mu}_n^d, \boldsymbol{\sigma}_n^d = g_{\boldsymbol{\phi}_d}(\boldsymbol{z}_n). \tag{2.35}$$

These are then used to sample initialization weights $\boldsymbol{\theta}_n \backsim N(\boldsymbol{\mu}_n^d, diag(\boldsymbol{\sigma}_n^{d2}))$. The loss from the generated weights can then be propagated backwards to adjust the embedding space. In practice, generating such high-dimensional set of parameters from a low-dimensional embedding can be quite problematic. Therefor, LEO uses pre-trained models, and only generates weights for the final layer, which limits the expressivity of the model.

A key advantage of LEO is that it optimizes in a lower-dimensional latent embedding space, which aids generalization performance. However, the approach is more complex than e.g. MAML (Finn et al., 2017), and its applicability is limited to few-shot learning settings.

2.5.10 Online MAML (FTML)

Online MAML (Finn et al., 2019) is an extension of MAML (Finn et al., 2017) to make it applicable to *online learning* settings (Anderson, 2008). In the online setting, we are presented with a sequence of tasks $\mathcal{T}_t$ with corresponding loss functions $\{\mathcal{L}_{\mathcal{T}_t}\}_{t=1}^T$, for some potentially infinite time horizon T. The goal is to pick a sequence of parameters $\{\boldsymbol{\theta}_t\}_{t=1}^T$ that performs well on the presented loss functions. This objective is captured by the $Regret_T$ over the entire sequence, which is defined by Finn et al. (2019) as follows

$$Regret_T = \sum_{t=1}^{T} \mathcal{L}_{\mathcal{T}_t}(\boldsymbol{\theta}_t') - min_{\boldsymbol{\theta}} \sum_{t=1}^{T} \mathcal{L}_{\mathcal{T}_t}(\boldsymbol{\theta}_t'), \tag{2.36}$$

where $\boldsymbol{\theta}$ are the initial model parameters (just as MAML), and $\boldsymbol{\theta}_t'$ are parameters resulting from a one-step gradient update (starting from $\boldsymbol{\theta}$) on task t. Here, the left term reflects the updated parameters chosen by the agent ($\boldsymbol{\theta}_t$), whereas the right term presents the minimum obtainable loss (in hindsight) from a single fixed set of parameters $\boldsymbol{\theta}$. Note that this setup assumes that the agent can make updates to its chosen parameters (transform its initial choice at time t from $\boldsymbol{\theta}_t$ to $\boldsymbol{\theta}_t'$).

Finn et al. (2019) propose FTML (Follow The Meta Leader), inspired by FTL (Follow The Leader) (Hannan, 1957; Kalai and Vempala, 2005), to minimize the regret. The basic idea is to set the parameters for the next time step $(t+1)$ equal to the best parameters in hindsight, i.e.,

$$\boldsymbol{\theta}_{t+1} := argmin_{\boldsymbol{\theta}} \sum_{k=1}^{t} \mathcal{L}_{\mathcal{T}_k}(\boldsymbol{\theta}'_k). \tag{2.37}$$

The gradient to perform meta-updates is then given by

$$g_t(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\mathcal{T}_k \backsim p_t(\mathcal{T})} \mathcal{L}_{\mathcal{T}_k}(\boldsymbol{\theta}'_k), \tag{2.38}$$

where $p_t(\mathcal{T})$ is a uniform distribution over tasks $1, ..., t$ (at time t).

Algorithm 4 contains the full pseudocode for FTML. In this algorithm, *MetaUpdate* performs a few (N_{meta}) meta-steps. In each meta-step, a task is sampled from B, together with train and test mini-batches to compute the gradient g_t in Equation 2.37. The initialization $\boldsymbol{\theta}$ is then updated ($\boldsymbol{\theta} := \boldsymbol{\theta} - \beta g_t(\boldsymbol{\theta})$), where β is the meta-learning rate. Note that the memory usage keeps increasing over time, as at every time step t, we append tasks to the buffer B, and keep task data sets in memory.

Algorithm 4 FTML by Finn et al. (2019)

Require: Performance threshold γ
Initialize empty task buffer B
for t = 1,... **do**
    Initialize data set $D_t = \emptyset$
    Append $\mathcal{T}_t$ to B
    while $|D_t| < N$ **do**
        Append batch of data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$ to D_t
        $\boldsymbol{\theta}_t = MetaUpdate(\boldsymbol{\theta}_t, B, t)$
        Compute $\boldsymbol{\theta}'_t$
        if $\mathcal{L}_{D_{\mathcal{T}_t}^{test}}(\boldsymbol{\theta}'_t) < \gamma$ **then**
            Save $|D_t|$ as the efficiency for task $\mathcal{T}_t$
        end if
    end while
    Save final performance $\mathcal{L}_{D_{\mathcal{T}_t}^{test}}(\boldsymbol{\theta}'_t)$
    $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t$
end for

In summary, Online MAML is a robust technique for online-learning (Finn et al., 2019). A downside of this approach is the computational costs that keep growing over time, as all encountered data are stored. Reducing these costs is a direction for future work. Also, one could experiment how well the approach works when more than one inner gradient update steps per task are used, as mentioned by Finn et al. (2019).

2.5.11 LLAMA

Grant et al. (2018) mold MAML into a probabilistic framework, such that a probability distribution over task-specific parameters $\boldsymbol{\theta}'_j$ is learned, instead of a single one. In this way, multiple potential

solutions can be obtained for a task. The resulting technique is called LLAMA (Laplace Approximation for Meta-Adaptation). Importantly, LLAMA is only developed for supervised learning settings.

A key observation is that a neural network $f_{\boldsymbol{\theta}'_j}$, parameterized by updated parameters $\boldsymbol{\theta}'_j$ (obtained from few gradient updates using $D^{tr}_{\mathcal{T}_j}$), outputs class probabilities $p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}'_j)$. To minimize the error on the query set $D^{test}_{\mathcal{T}_j}$, the model must output large probability scores for the true classes. This objective is captured in the maximum log likelihood loss function

$$\mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) = - \sum_{\boldsymbol{x}_i, y_i \in D^{test}_{\mathcal{T}_j}} log\, p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}'_j). \tag{2.39}$$

Simply put, if we see a task j as a probability distribution over examples $p_{\mathcal{T}_j}$, we wish to maximize the probability that the model predicts the correct class y_i, given an input $\boldsymbol{x}_i$. This can be done by plain gradient descent, as shown in Algorithm 5, where β is the meta-learning rate. Line 4 refers to ML-LAPLACE, which is a subroutine that computes task-specific updated parameters $\boldsymbol{\theta}'_j$, and estimates the negative log likelihood (loss function) which is used to update the initialization $\boldsymbol{\theta}$, as shown in Algorithm 6. Grant et al. (2018) approximated the quadratic curvature matrix $\hat{H}$ using K-FAC (Martens and Grosse, 2015).

The trick is that the initialization $\boldsymbol{\theta}$ defines a distribution $p(\boldsymbol{\theta}'_j|\boldsymbol{\theta})$ over task-specific parameters $\boldsymbol{\theta}'_j$. This distribution was taken to be a diagonal Gaussian (Grant et al., 2018). Then, to sample solutions for a new task $\mathcal{T}_j$, one can simply generate possible solutions $\boldsymbol{\theta}'_j$ from the learned Gaussian distribution.

Algorithm 5 LLAMA by Grant et al. (2018)

1: Initialize $\boldsymbol{\theta}$ randomly
2: **while** not converged **do**
3: Sample a batch of J tasks: $B = \mathcal{T}_1, ..., \mathcal{T}_J \backsim p(\mathcal{T})$
4: Estimate $\mathbb{E}_{(\boldsymbol{x}_i, y_i) \backsim p_{\mathcal{T}_j}}[-log\, p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta})] \, \forall \mathcal{T}_j \in B$ using ML-LAPLACE
5: $\boldsymbol{\theta} = \boldsymbol{\theta} - \beta \nabla_{\boldsymbol{\theta}} \sum_j \mathbb{E}_{(\boldsymbol{x}_i, y_i) \backsim p_{\mathcal{T}_j}}[-log\, p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta})$
6: **end while**

Algorithm 6 ML-LAPLACE (Grant et al., 2018)

1: $\boldsymbol{\theta}'_j = \boldsymbol{\theta}$
2: **for** k=1,...,K **do**
3: $\boldsymbol{\theta}'_j = \boldsymbol{\theta}'_j + \alpha \nabla_{\boldsymbol{\theta}'_j} log\, p(y_i \in D^{tr}_{\mathcal{T}_j} | \boldsymbol{\theta}'_j, \boldsymbol{x}_i \in D^{tr}_{\mathcal{T}_j})$
4: **end for**
5: Compute curvature matrix $\hat{H} = \nabla^2_{\boldsymbol{\theta}'_j}[-log\, p(y_i \in D^{test}_{\mathcal{T}_j} | \boldsymbol{\theta}'_j, \boldsymbol{x}_i \in D^{test}_{\mathcal{T}_j})] + \nabla^2_{\boldsymbol{\theta}'_j}[-log\, p(\boldsymbol{\theta}'_j|\boldsymbol{\theta})]$
6: **return** $-log\, p(y_i \in D^{test}_{\mathcal{T}_j} | \boldsymbol{\theta}'_j, \boldsymbol{x}_i \in D^{test}_{\mathcal{T}_j}) + \eta\, log[det(\hat{H})]$

In short, LLAMA extends MAML in probabilistic fashion, such that one can obtain multiple solutions for a single task, instead of one. This does, however, increase the computational costs. On top of that, the used Laplace approximation (in ML-LAPLACE) can be quite inaccurate (Grant et al., 2018).

2.5.12 PLATIPUS

PLATIPUS (Finn et al., 2018) builds upon the probabilistic interpretation of LLAMA (Grant et al., 2018), but learns a probability distribution over initializations $\boldsymbol{\theta}$, instead of task-specific parameters $\boldsymbol{\theta}'_j$. Thus, PLATIPUS allows one to sample an initialization $\boldsymbol{\theta} \backsim p(\boldsymbol{\theta})$, which can be updated with gradient descent to obtain task-specific weights (fast weights) $\boldsymbol{\theta}'_j$.

Algorithm 7 PLATIPUS training algorithm by Finn et al. (2018)

1: Initialize $\boldsymbol{\Theta} = \{\boldsymbol{\mu_\theta}, \boldsymbol{\sigma}^2_{\boldsymbol{\theta}}, \boldsymbol{v}_q, \boldsymbol{\gamma}_p, \boldsymbol{\gamma}_q\}$
2: **while** Not done **do**
3: Sample batch of tasks $B = \{\mathcal{T}_j \backsim p(\mathcal{T})\}_{i=1}^{m}$
4: **for** $\mathcal{T}_j \in B$ **do**
5: $D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j} = \mathcal{T}_j$
6: Compute $\nabla_{\boldsymbol{\mu_\theta}} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\mu_\theta})$
7: Sample $\boldsymbol{\theta} \backsim q = N(\boldsymbol{\mu_\theta} - \boldsymbol{\gamma}_q \nabla_{\boldsymbol{\mu_\theta}} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\mu_\theta}), \boldsymbol{v}_q)$
8: Compute $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})$
9: Compute fast weights $\boldsymbol{\theta}'_i = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})$
10: **end for**
11: $p(\boldsymbol{\theta}|D^{tr}_{\mathcal{T}_j}) = N(\boldsymbol{\mu_\theta} - \boldsymbol{\gamma}_p \nabla_{\boldsymbol{\mu_\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\mu_\theta}), \boldsymbol{\sigma}^2_{\boldsymbol{\theta}})$
12: Compute $\nabla_{\boldsymbol{\Theta}} \left[\sum_{\mathcal{T}_j} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\phi}_i) + D_{KL}(q(\boldsymbol{\theta}|D^{test}_{\mathcal{T}_j}), p(\boldsymbol{\theta}|D^{tr}_{\mathcal{T}_j})) \right]$
13: Update $\boldsymbol{\Theta}$ using the Adam optimizer
14: **end while**

The approach is best explained by its pseudocode, as shown in Algorithm 7. In contrast to the original MAML, PLATIPUS introduces five more parameter vectors (line 1). All of these parameters are used to facilitate the creation of Gaussian distributions over prior initializations (or simply priors) $\boldsymbol{\theta}$. That is, $\boldsymbol{\mu_\theta}$ represents the vector mean of the distributions. $\boldsymbol{\sigma}^2_{\boldsymbol{q}}$, and $\boldsymbol{v}_q$ represent the covariances of train and test distributions respectively. $\boldsymbol{\gamma}_x$ for $x = q, p$ are learning rate vectors for performing gradient steps on distributions q (line 6 and 7) and P (line 11).

The key difference with the regular MAML is that instead of having a single initialization point $\boldsymbol{\theta}$, we now learn distributions over priors: q and P, which are based on query and support data sets of task $\mathcal{T}_j$ respectively. Since these data sets come from the same task, we want the distributions $q(\boldsymbol{\theta}|D^{test}_{\mathcal{T}_j})$, and $p(\boldsymbol{\theta}|D^{tr}_{\mathcal{T}_j})$ to be close to each other. This is enforced by the Kullback–Leibler divergence (D_{KL}) loss term on line 12, which measures the distance between the two distributions. Importantly, note that q (line 7) and P (line 11) use vector means which are computed with one gradient update steps using the query and support data sets respectively. The idea is that the mean of the Gaussian distributions should be close to the updated mean $\boldsymbol{\mu_\theta}$, because we want to enable fast learning. As one can see, the training process is very similar to that of MAML (Finn et al., 2017) (Section 2.5.5), with some small adjustments to allow us to work with the probability distributions over $\boldsymbol{\theta}$.

At test-time, one can simply sample a new initialization $\boldsymbol{\theta}$ from the prior distribution $p(\boldsymbol{\theta}|D^{tr}_{\mathcal{T}_j})$ (note that q cannot be used at test-time as we do not have access to $D^{test}_{\mathcal{T}_j}$), and apply a gradient update on the provided support set $D^{tr}_{\mathcal{T}_j}$. Note that this allows us to sample multiple potential initializations $\boldsymbol{\theta}$ for the given task.

The key advantage of PLATIPUS is that it is aware of its own uncertainty, which greatly increases

the applicability of deep meta-learning in critical domains such as medical diagnosis (Finn et al., 2018). Based on this uncertainty, it can ask for labels of some inputs it is unsure about (active learning). A downside to this approach, however, are the increased computational costs, and the fact that it is not applicable to reinforcement learning.

2.5.13 Bayesian MAML (BMAML)

Bayesian MAML (Yoon et al., 2018) is another probabilistic variant of MAML that can generate multiple solutions. However, instead of learning a distribution over potential solutions, BMAML simply keeps M possible solutions, and optimizes them in joint fashion. Recall that probabilistic MAMLs (e.g., PLATIPUS) attempt to maximize the data likelihood of task $\mathcal{T}_j$, i.e., $p(\boldsymbol{y}_j^{test}|\boldsymbol{\theta}_j')$, where $\boldsymbol{\theta}_j'$ are task-specific fast weights obtained by one or more gradient updates. Yoon et al. (2018) model this likelihood using Stein Variational Gradient Descent (SVGD) (Liu and Wang, 2016).

To obtain M solutions, or equivalently, parameter settings $\boldsymbol{\theta}^m$, SVGD keeps a set of M *particles* $\boldsymbol{\Theta} = \{\boldsymbol{\theta}^m\}_{i=1}^M$. At iteration t, every $\boldsymbol{\theta}_t \in \boldsymbol{\Theta}$ is updated as follows

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon(\phi(\boldsymbol{\theta}_t)) \tag{2.40}$$

$$\text{where } \phi(\boldsymbol{\theta}_t) = \frac{1}{M}\sum_{m=1}^{M}\left[k(\boldsymbol{\theta}_t^m, \boldsymbol{\theta}_t)\nabla_{\boldsymbol{\theta}_t^m} log\, p(\boldsymbol{\theta}_t^m) + \nabla_{\boldsymbol{\theta}_t^m} k(\boldsymbol{\theta}_t^m, \boldsymbol{\theta}_t)\right]. \tag{2.41}$$

Here, $k(\boldsymbol{x}, \boldsymbol{x}')$ is a similarity kernel between $\boldsymbol{x}$ and $\boldsymbol{x}'$. The authors used a radial basis function (RBF) kernel, but in theory, any other kernel could be used. Note that the update of one particle depends on the other gradients of particles. The first term in the summation ($k(\boldsymbol{\theta}_t^m, \boldsymbol{\theta}_t)\nabla_{\boldsymbol{\theta}_t^m} log\, p(\boldsymbol{\theta}_t^m)$) moves the particle in the direction of the gradients of other particles, based on particle similarity. The second term ($\nabla_{\boldsymbol{\theta}_t^m} k(\boldsymbol{\theta}_t^m, \boldsymbol{\theta}_t)$) ensures that particles do not collapse (repulsive force) (Yoon et al., 2018).

These particles can then be used to approximate the probability distribution of the test labels

$$p(\boldsymbol{y}_j^{test}|\boldsymbol{\theta}_j') \approx \frac{1}{M}\sum_{m=1}^{M} p(\boldsymbol{y}_j^{test}|\boldsymbol{\theta}_{\mathcal{T}_j}^m), \tag{2.42}$$

where $\boldsymbol{\theta}_{\mathcal{T}_j}^m$ is the m-th particle obtained by training on the support set $D_{\mathcal{T}_j}^{tr}$ of task $\mathcal{T}_j$.

Yoon et al. (2018) proposed a new meta-loss to train BMAML, called the *Chaser Loss*. This loss relies on the insight that we want the approximated parameter distribution (obtained from the support set $p_{\mathcal{T}_j}^n(\boldsymbol{\theta}_{\mathcal{T}_j}|D^{tr}, \boldsymbol{\Theta}_0)$) and true distribution $p_{\mathcal{T}_j}^\infty(\boldsymbol{\theta}_{\mathcal{T}_j}|D^{tr} \cup D^{test})$ to be close to each other (since the task is the same). Here, n denotes the number of SVGD steps, and $\boldsymbol{\Theta}_0$ is the set of initial particles, in similar fashion to the initial parameters $\boldsymbol{\theta}$ seen by MAML. Since the true distribution is unknown, Yoon et al. (2018) approximate it by running SVGD for s additional steps, granting us the *leader* $\boldsymbol{\Theta}_{\mathcal{T}_j}^{n+s}$, where the s additional steps are performed on the combined support and query set. The intuition is that as the number of updates increases, the obtained distributions become more like the true one. $\boldsymbol{\Theta}_{\mathcal{T}_j}^n$ in this context is called the *chaser* as it wants to get closer to the leader. The proposed meta-loss is then given by

$$\mathcal{L}_{BMAML}(\boldsymbol{\Theta}_0) = \sum_{\mathcal{T}_j \in B} \sum_{m=1}^{M} ||\boldsymbol{\theta}_{\mathcal{T}_j}^{n,m} - \boldsymbol{\theta}_{\mathcal{T}_j}^{n+s,m}||_2^2. \tag{2.43}$$

The full pseudocode of BMAML is shown in Algorithm 8. Here, $\boldsymbol{\Theta}_{\mathcal{T}_j}^{n}(\boldsymbol{\Theta}_0)$ denotes the set of particles after n updates on task $\mathcal{T}_j$, and SG means "stop gradients" (we do not want the leader to depend on the initialization, as the leader must lead).

Algorithm 8 BMAML by Yoon et al. (2018)

Initialize $\boldsymbol{\Theta}_0$
for t=1,... until convergence **do**
  Sample a batch of tasks B from $p(\mathcal{T})$
  for task $\mathcal{T}_j \in B$ **do**
    Compute chaser $\boldsymbol{\Theta}_{\mathcal{T}_j}^{n}(\boldsymbol{\Theta}_0) = SVGD_n(\boldsymbol{\Theta}_0; D_{\mathcal{T}_j}^{tr}, \alpha)$
    Compute leader $\boldsymbol{\Theta}_{\mathcal{T}_j}^{n+s}(\boldsymbol{\Theta}_0) = SVGD_s(\boldsymbol{\Theta}_{\mathcal{T}_j}^{n}(\boldsymbol{\Theta}_0); D_{\mathcal{T}_j}^{tr} \cup D_{\mathcal{T}_j}^{test}, \alpha)$
  end for
  $\boldsymbol{\Theta}_0 = \boldsymbol{\Theta}_0 - \beta \nabla_{\boldsymbol{\Theta}_0} \sum_{\mathcal{T}_j \in B} d(\boldsymbol{\Theta}_{\mathcal{T}_j}^{n}(\boldsymbol{\Theta}_0), SG(\boldsymbol{\Theta}_{\mathcal{T}_j}^{n+s}(\boldsymbol{\Theta}_0)))$
end for

In summary, BMAML is a robust optimization-based meta-learning technique that can propose M potential solutions to a task. Additionally, it is applicable to reinforcement learning by using Stein Variational Policy Gradient instead of SVGD. A downside of this approach is that one has to keep M parameter sets in memory, which does not scale well. Reducing the memory costs is a direction for future work (Yoon et al., 2018). Furthermore, SVGD is sensitive to the selected kernel function, which was pre-defined in BMAML. However, Yoon et al. (2018) point out that it may be beneficial to learn the kernel function instead. This is another possibility for future research.

2.5.14 Simple differentiable solvers

Bertinetto et al. (2019) take a quite different approach. That is, they pick simple base-learners that have an analytical closed-form solution. The intuition is that the existence of a closed-form solution allows for good learning efficiency. They propose two techniques using this principle, namely **R2-D2** (Ridge Regression Differentiable Discriminator), and **LR-D2** (Logistic Regression Differentiable Discriminator). We cover both in turn.

Let $g_{\boldsymbol{\phi}} : X \rightarrow \mathbb{R}^e$ be a pre-trained input embedding model (e.g. a CNN), which outputs embeddings with a dimensionality of e. Furthermore, assume that we use a linear predictor function $f(g_{\boldsymbol{\phi}}(\boldsymbol{x}_i)) = g_{\boldsymbol{\phi}}(\boldsymbol{x}_i)W$, where W is a $e \times o$ weight matrix, and o is the output dimensionality (of the label). When using (regularized) Ridge Regression (done by R2-D2), one uses the optimal W, i.e.,

$$\begin{aligned} W^* &= \arg\min_W ||XW - Y||_2^2 + \gamma ||W||^2 \\ &= (X^T X + \gamma I)^{-1} X^T Y, \end{aligned} \tag{2.44}$$

where $X \in \mathbb{R}^{n \times e}$ is the input matrix, containing n rows (one for each embedded input $g_{\boldsymbol{\phi}}(\boldsymbol{x}_i)$),

$Y \in \mathbb{R}^{n \times o}$ is the output matrix with correct outputs corresponding to the inputs, and γ is a regularization term to prevent overfitting. Note that the analytical solution contains the term $(X^T X) \in \mathbb{R}^{e \times e}$, which is quadratic in the size of the embeddings. Since e can become quite large when using deep neural networks, Bertinetto et al. (2019) use Woodburry's identity

$$W^* = X^T (X X^T + \gamma I)^{-1} Y, \tag{2.45}$$

where $X X^T \in \mathbb{R}^{n \times n}$ is linear in the embedding size, and quadratic in the number of examples, which is more manageable in few-shot settings, where n is very small. To make predictions with this Ridge Regression based model, one can compute

$$\hat{Y} = \alpha X_{test} W^* + \beta, \tag{2.46}$$

where α and β are hyperparameters of the base-learner that can be learned by the meta-learner, and $X_{test} \in \mathbb{R}^{m \times e}$ corresponds to the m test inputs of a given task. Thus, the meta-learner needs to learn α, β, γ, and $\boldsymbol{\phi}$ (embedding weights of the CNN).

The technique can also be applied to iterative solvers when the optimization steps are differentiable (Bertinetto et al., 2019). LR-D2 uses the Logistic Regression objective and Newton's method as solver. Outputs $\boldsymbol{y} \in \{-1, +1\}^n$ are now binary. Let $\boldsymbol{w}$ denote a parameter row of our linear model (parameterized by W). Then, the i-th iteration of Newton's method, updates $\boldsymbol{w}_i$ as follows

$$\boldsymbol{w}_i = (X^T \text{diag}(\boldsymbol{s}_i) X + \gamma I)^{-1} X^T \text{diag}(\boldsymbol{s}_i) \boldsymbol{z}_i, \tag{2.47}$$

where $\boldsymbol{\mu}_i = \sigma(\boldsymbol{w}_{i-1}^T X)$, $\boldsymbol{s}_i = \boldsymbol{\mu}_i (1 - \boldsymbol{\mu}_i)$, $\boldsymbol{z}_i = \boldsymbol{w}_{i-1}^T X + (\boldsymbol{y} - \boldsymbol{\mu}_i)/\boldsymbol{s}_i$, and σ is the sigmoid function. Since the term $X^T \text{diag}(\boldsymbol{s}_i) X$ is a matrix of size $e \times e$, and thus again quadratic in the embedding size, Woodburry's identity is also applied here to obtain

$$\boldsymbol{w}_i = X^T (X X^T + \lambda \text{diag}(\boldsymbol{s}_i)^{-1})^{-1} \boldsymbol{z}_i, \tag{2.48}$$

making it quadratic in the input size, which is not a big problem since n is small in the few-shot setting. The main difference compared to R2-D2 is that the base-solver has to be run for multiple iterations to obtain W.

In the few-shot setting, the base-level optimizers compute the weight matrix W for a given task $\mathcal{T}_i$. The obtained loss on the query set of a task $\mathcal{L}_{D_{test}}$ is then used to update the parameters $\boldsymbol{\phi}$ of the input embedding function (e.g. CNN) and the hyperparameters of the base-learner.

Lee et al. (2019) have done similar work to Bertinetto et al. (2019), but with linear Support Vector Machines (SVMs) as base-learner. Their approach is dubbed **MetaOptNet**, and achieved state-of-the-art performance on few-shot image classification.

In short, simple differentiable solvers are simple, reasonably fast in terms of computation time, but limited to few-shot learning settings. Investigating the use of other simple base-learners is a direction for future work.

2.5.15 Optimization-based techniques, in conclusion

Optimization-based aim to learn new tasks quickly through (learned) optimization procedures. Note that this closely resembles base-level learning, which also occurs through optimization (e.g., gradient descent). However, in contrast to base-level techniques, optimization-based meta-learners can learn the optimizer and/or are exposed to multiple tasks, which allows them to learn to learn new tasks quickly. Figure 2.28 shows the relationships between the covered optimization-based techniques.

As we can see, the LSTM optimizer (Andrychowicz et al., 2016), which replaces hand-crafted optimization procedures such as gradient descent by a trainable LSTM, can be seen as the starting point for these optimization-based meta-learning techniques. Li and Malik (2018) also aim to learn a the optimization procedure with reinforcement learning instead of gradient-based methods. The LSTM meta-learner (Ravi and Larochelle, 2017) extends the LSTM optimizer to the few-shot setting by not only learning the optimization procedure, but also a good set of initial weights. This way, it can be used across tasks. MAML (Finn et al., 2017) is a simplification of the LSTM meta-learner as it replaces the trainable LSTM optimizer by hand-crafted gradient descent. MAML has received considerable attention within the field of deep meta-learning, and has, as one can see, inspired many other works.

Meta-SGD is an enhancement of MAML that not only learns the initial parameters, but also the learning rates (Li et al., 2017). LLAMA (Grant et al., 2018), PLATIPUS (Finn et al., 2018), and online MAML (Finn et al., 2019) extend MAML to the active and online learning settings. LLAMA and PLATIPUS are probabilistic interpretations of MAML, which allow them to sample multiple solutions for a given task and to quantify their uncertainty. BMAML (Yoon et al., 2018) takes a more discrete approach as it jointly optimizes a discrete set of M initializations. iMAML (Rajeswaran et al., 2019) aims to overcome the computational expenses associated with the computation of second-order derivatives, which is needed by MAML. Through implicit differentiation, they also allow for the use of non-differentiable inner loop optimization procedures. Reptile (Nichol et al., 2018) is an elegant first-order meta-learning algorithm for finding a set of initial parameters and removes the need of computing higher-order derivatives. LEO (Rusu et al., 2019) tries to improve the robustness of MAML by optimizing in lower-dimensional parameter space through the use of an encoder-decoder architecture. Lastly, R2-D2, LR-D2 (Bertinetto et al., 2019), and Lee et al. (2019) use simple classical machine learning methods (ridge regression, logistic regression, SVM, respectively) as classifier on top of a learned feature extractor.

A key advantage of optimization-based approaches is that they can achieve better performance on wider task distributions than, e.g., model-based approaches (Finn and Levine, 2018). However, optimization-based techniques optimize a base-learner for every task that they are presented with and/or learn the optimization procedure, which is computationally expensive (Hospedales et al., 2021).

Optimization-based meta-learning is a very active area of research. We expect future work to be done in order to reduce the computational demands of these methods, and improve the solution quality and level of generalization. We think that benchmarking and reproducibility research will play an important role in these improvements.

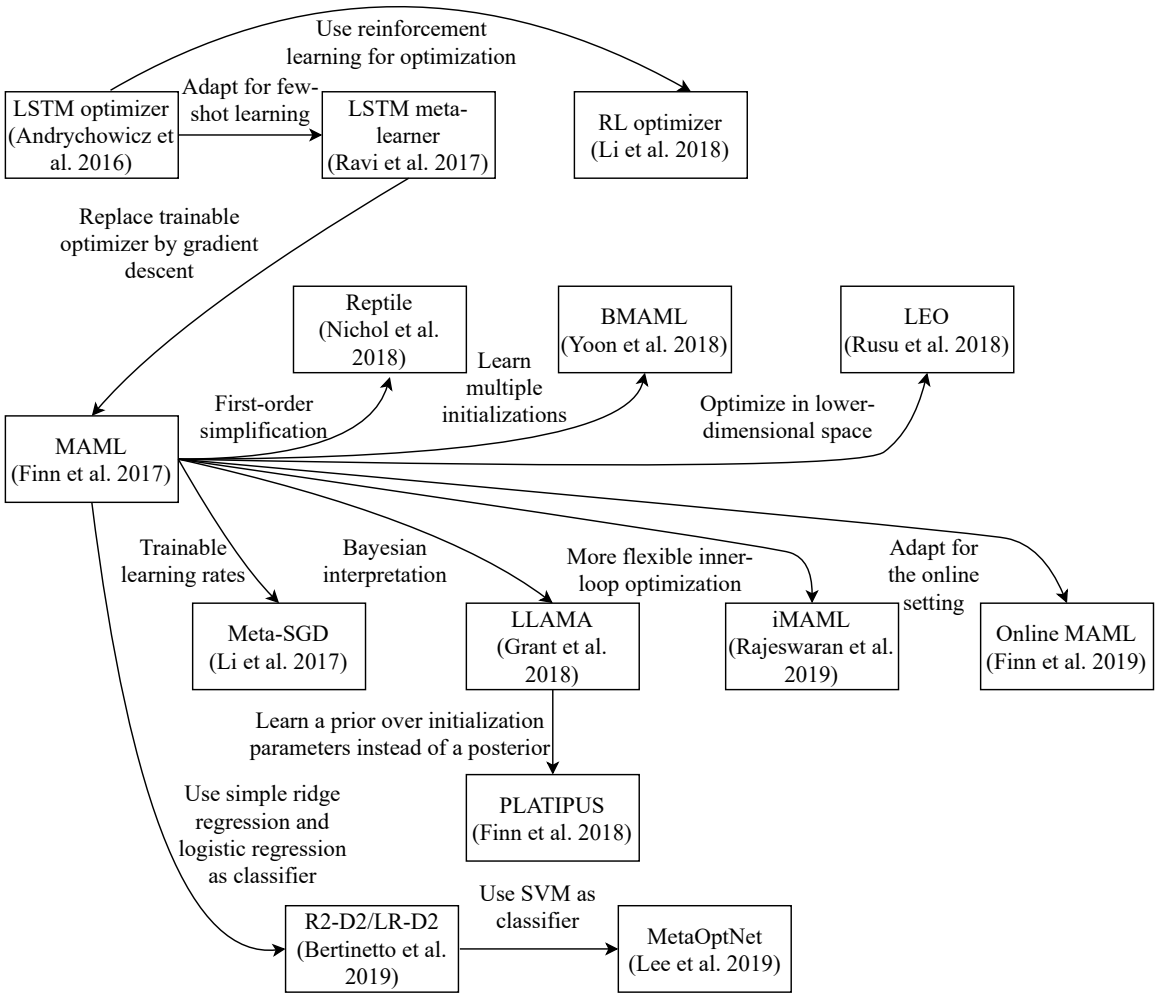


Figure 2.28: The relationships between the covered optimization-based meta-learning techniques. As one can see, MAML has a central position in this network of techniques as it has inspired many other works.

2.6 Concluding Remarks

In this section, we give a helicopter view of all that we discussed, and the field of deep meta-learning in general. We will also discuss challenges and future research.

2.6.1 Overview

In recent years, there has been a shift in focus in the broad meta-learning community. Traditional algorithm selection and hyperparameter optimization for classical machine learning techniques (e.g. Support Vector Machines, Logistic Regression, Random Forests, etc.) have been augmented by deep meta-learning, or equivalently, the pursuit of self-improving neural networks that can leverage prior learning experience to learn new tasks more quickly. Instead of training a new model from scratch for different tasks, we can use the same (meta-learning) model across tasks. As such, meta-learning can widen the applicability of powerful deep learning techniques to domains where fewer data are available and computational resources are limited.

Deep meta-learning techniques are characterized by their meta-objective, which allows them to maximize performance across various tasks, instead of a single one, as is the case in base-level learning objectives. This meta-objective is reflected in the training procedure of meta-learning methods, as they learn on a set of different meta-training tasks. The few-shot setting lends itself nicely towards this end, as tasks consist of few data points. This makes it computationally feasible to train on many different tasks, and it allows us to evaluate whether a neural network can learn new concepts from few examples. Task construction for training and evaluation does require some special attention. That is, it has been shown beneficial to match training and test conditions (Vinyals et al., 2016), and perhaps train in a more difficult setting than the one that will be used for evaluation (Snell et al., 2017).

On a high level, there are three categories of deep meta-learning techniques, namely i) metric-, ii) model-, and iii) optimization-based ones, which rely on i) computing input similarity, ii) task embeddings with states, and iii) task-specific updates, respectively. Each approach has strengths and weaknesses. Metric-learning techniques are simple and effective (Garcia and Bruna, 2017), but are not readily applicable outside of the supervised learning setting (Hospedales et al., 2021). Model-based techniques, on the other hand, can have very flexible internal dynamics, but lack generalization ability to more distant tasks than the ones used at meta-train time (Finn and Levine, 2018). Optimization-based approaches have shown greater generalizability, but are in general computationally expensive, as they optimize a base-learner for every task (Finn and Levine, 2018; Hospedales et al., 2021).

Table 2.2 provides a concise, tabular overview of these approaches. Many techniques have been proposed for each one of the categories, and the underlying ideas may vary greatly, even within the same category. Table 2.3, therefore, provides an overview of all methods and key ideas that we have discussed in this chapter, together with their applicability to supervised learning (SL) and reinforcement learning (RL) settings, key ideas, and benchmarks that were used for testing them. Table 2.5 displays an overview of the 1- and 5-shot classification performances (reported by the original authors) of the techniques on the frequently used miniImageNet benchmark. Moreover, it displays the used backbone (feature extraction module) as well as the final classification mechanism. From this table, it becomes clear that the 5-shot performance is typically better than the 1-shot performance, indicating that data scarcity is a large bottleneck for achieving good performance. Moreover, there is a strong relationship between the expressivity of the backbone and the performance. That is, deeper backbones tend to give rise to better classification performance. The best performance is achieved by MetaOptNet, yielding a 1-shot accuracy of 64.09% and a 5-shot accuracy of 80.00%.

Note however that MetaOptNet used a deeper backbone than most of the other techniques.

2.6.2 Open challenges and future work

Despite the great potential of deep meta-learning techniques, there are still open challenges, which we discuss here.

Figure 2.1 in Section 2.1 displays the accuracy scores of the covered meta-learning techniques on 1-shot miniImageNet classification. Techniques which were not tested in this setting by the original authors are omitted. As we can see, the performance of the techniques is related to the expressivity of the used backbone (ordered in increasing order on the x-axis). For example, the best performing techniques, LEO and MetaOptNet, use the largest network architectures. Moreover, the fact that different techniques use different backbones poses a problem as it is difficult to fairly compare their classification performance. An obvious question arises to which degree the difference in performance is due to methodological improvements, or due to the fact that a better backbone architecture was chosen. For this reason, we think that it would be useful to perform a large scale benchmark test where techniques are compared when they use the same backbones. This would also allow us to get a more clear idea of how the expressivity of the feature extraction module affects the performance.

Another challenge of deep meta-learning techniques is that they can be susceptible to the *memorization problem* (meta-overfitting), where the neural network has memorized tasks seen at meta-training time, and fails to generalize to new tasks. More research is required to better understand this problem. Clever task design and meta-regularization may prove useful to avoid such problems (Yin et al., 2020).

Another problem is that most of the meta-learning techniques discussed in this chapter are evaluated on narrow benchmark sets. This means that the data that the meta-learner used for training are not too distant from the data used for evaluating its performance. As such, one may wonder how well these techniques are actually able to adapt to more distant tasks. Chen et al. (2019) showed that the ability to adapt to new tasks decreases as they become more distant from the tasks seen at training time. Moreover, a simple non-meta-learning baseline (based on pre-training and fine-tuning) can outperform state-of-the-art meta-learning techniques when meta-test tasks come from a different data set than the one used for meta-training.

In reaction to these findings, Triantafillou et al. (2020) have recently proposed the Meta-Dataset benchmark, which consists of various previously used meta-learning benchmarks such as Omniglot (Lake et al., 2011) and ImageNet (Deng et al., 2009). In a similar spirit, Ullah et al. (2022) have released Meta-Album, a dataset consisting of 30 datasets from 10 different domains. This way, meta-learning techniques can be evaluated in more challenging settings where tasks are diverse. Following Hospedales et al. (2021), we think that this new benchmark can prove to be a good mean towards the investigation and development of meta-learning algorithms for such challenging scenarios.

As mentioned earlier in this section, deep meta-learning has the appealing prospect of widening the applicability of deep learning techniques to more real-world domains. For this, increasing the generalization ability of these techniques is very important. Additionally, the computational costs associated with the deployment of meta-learning techniques should be small. While these techniques can learn new tasks quickly, meta-training can be quite computationally expensive. Thus, decreasing the required computation time and memory costs of deep meta-learning techniques remains an open challenge.

Name	Backbone	Classifier	1-shot	5-shot
Metric-based				
Siamese nets	-	-	-	
Matching nets	64-64-64-64	Cosine sim.	43.56 ± 0.84	55.31 ± 0.73
Prototypical nets	64-64-64-64	Euclidean dist.	49.42 ± 0.78	68.20 ± 0.66
Relation nets	64-96-128-256	Sim. network	50.44 ± 0.82	65.32 ± 0.70
ARC	-	64-1 dense	$49.14 \pm -$	-
GNN	64-96-128-256	Softmax	50.33 ± 0.36	66.41 ± 0.63
Model-based				
RMLs	-	-	-	
MANNs	-	-	-	
Meta nets	64-64-64-64-64	64-Softmax	49.21 ± 0.96	-
SNAIL	Adj. ResNet-12	Softmax	55.71 ± 0.99	68.88 ± 0.92
CNP	-	-	-	
Neural stat.	-	-	-	
Opt.-based				
LSTM optimizer	-	-	-	
LSTM ml.	32-32-32-32	Softmax	43.44 ± 0.77	60.60 ± 0.71
RL optimizer	-	-	-	
MAML	32-32-32-32	Softmax	48.70 ± 1.84	63.11 ± 0.92
iMAML	64-64-64-64	Softmax	49.30 ± 1.88	-
Meta-SGD	64-64-64-64	Softmax	50.47 ± 1.87	64.03 ± 0.94
Reptile	32-32-32-32	Softmax	48.21 ± 0.69	66.00 ± 0.62
LEO	WRN-28-10	Softmax	61.76 ± 0.08	77.59 ± 0.12
Online MAML	-	-	-	
LLAMA	64-64-64-64	Softmax	49.40 ± 1.83	-
PLATIPUS	-	-	-	
BMAML	64-64-64-64-64	Softmax	53.80 ± 1.46	-
Diff. solvers				
R2-D2	96-192-384-512	Ridge regr.	51.8 ± 0.2	68.4 ± 0.2
LR-D2	96-192-384-512	Log. regr.	51.90 ± 0.20	68.70 ± 0.20
MetaOptNet	ResNet-12	SVM	$\mathbf{64.09 \pm 0.62}$	$\mathbf{80.00 \pm 0.45}$

Table 2.5: Comparison of the accuracy scores of the covered meta-learning techniques on 1- and 5-shot miniImageNet classification. Scores are taken from the original papers. The $\pm$ indicates the 95% confidence interval. The backbone is the used feature extraction module. The classifier column shows the final layer(s) that were used to transform the features into class predictions. Used abbreviations: "sim.": similarity, "Adj.": adjusted, and "dist.": distance, "log.": logistic, "regr.": regression, "ml.": meta-learner, "opt.": optimization.

Some real world problems demand systems that can perform well in online, or active learning settings. The investigation of deep meta-learning in these settings (Finn et al., 2018; Yoon et al., 2018; Finn et al., 2019; Munkhdalai and Yu, 2017; Vuorio et al., 2018) remains an important direction for future work.

Yet another direction for future research is the creation of *compositional* deep meta-learning systems, which instead of learning flat and associative functions $\boldsymbol{x} \rightarrow y$, organize knowledge in a *compositional* manner. This would allow them to decompose an input $\boldsymbol{x}$ into several (already learned) components $c_1(\boldsymbol{x}), ..., c_n(\boldsymbol{x})$, which in turn could help the performance in low-data regimes (Tokmakov et al., 2019).

The question has been raised whether contemporary deep meta-learning techniques actually learn how to perform rapid learning, or simply learn a set of robust high-level features, which can be (re)used for many (new) tasks. Raghu et al. (2020) investigated this question for the most popular deep meta-learning technique MAML, and found that it largely relies on feature reuse. It would be interesting to see whether we can develop techniques that rely more on fast learning, and what the effect would be on performance.

Lastly, it may be useful to add more meta-abstraction levels, giving rise to, e.g., meta-meta-learning, meta-meta-...-learning (Hospedales et al., 2021; Schmidhuber, 1987).

This chapter provides a comprehensive exploration of the field of deep meta-learning, offering a thorough analysis of different prominent deep meta-learning techniques, their interconnections, and their relative performances. The broad and in-depth overview of the field of deep meta-learning obtained in this chapter serves as a springboard for the research performed in the remaining chapters of this dissertation. For instance, in writing this chapter, we came to understand intuitively that the meta-learner LSTM and MAML are tightly connected in the sense that the former could emulate what MAML learns, but it could also do more. Surprisingly, however, the meta-learner LSTM is outperformed by MAML in terms of its few-shot learning performance. Subsequently, in Chapter 3, we aim to investigate the underlying factors that contribute to this performance gap. Chapter 4 and Chapter 5 follow a similar pattern, where we investigate empirically observed performance disparities between different methods and aim to understand the underlying factors contributing to these disparities.

Chapter 6, in contrast, takes the opposite perspective. That is, rather than trying to gain an increased understanding of deep meta-learning methods starting from empirical observations, we start from theoretical machine learning knowledge and investigate whether the integration into a deep meta-learning algorithm can improve the few-shot learning performance.

Chapter 3

Stateless Neural Meta-Learning using Second-Order Gradients

Chapter overview

Meta-learning can be used to learn a good prior that facilitates quick learning; two popular approaches are MAML and the meta-learner LSTM. These two methods represent important and different approaches in meta-learning. In this work[1], we study the two and formally show that the meta-learner LSTM subsumes MAML, although MAML, which is in this sense less general, outperforms the other. We suggest the reason for this surprising performance gap is related to second-order gradients. We construct a new algorithm (named TURTLE) to gain more insight into the importance of second-order gradients. TURTLE is simpler than the meta-learner LSTM yet more expressive than MAML and outperforms both techniques at few-shot sine wave regression and 50% of the tested image classification settings (without any additional hyperparameter tuning) and is competitive otherwise, at a computational cost that is comparable to second-order MAML. We find that second-order gradients also significantly increase the accuracy of the meta-learner LSTM. When MAML was introduced, one of its remarkable features was the use of second-order gradients. Subsequent work focused on cheaper first-order approximations. On the basis of our findings, we argue for more attention for second-order gradients.

3.1 Introduction

In this chapter, we study the relationship between two popular deep meta-learning methods: MAML (Finn et al., 2017) and the meta-learner LSTM (Ravi and Larochelle, 2017). Recall that MAML aims to learn a good weight initialization from which it can learn new tasks quickly using regular gradient descent. In addition to learning a good weight initialization, the meta-learner LSTM (Ravi and Larochelle, 2017) attempts to learn the optimization procedure in the form of a separate LSTM network. The meta-learner LSTM is more general than MAML in the sense that the LSTM can learn to perform gradient descent (see Section 3.4) or something better.

[1]This chapter is based on the following published research article: *Huisman, M., Plaat, A., & van Rijn, J. N. (2022). Stateless neural meta-learning using second-order gradients. Machine Learning, 111(9), 3227-3244. Springer.*

This suggests that the performance of MAML can be mimicked by the meta-learner LSTM on few-shot image classification. However, our experimental results and those by Finn et al. (2017) show that this is not necessarily the case. The meta-learner LSTM fails to find a solution in the *meta-landscape* that learns as well as gradient descent.

In this chapter, we aim to investigate the performance gap between MAML and the meta-learner LSTM. We hypothesize that the underperformance of the meta-learner LSTM could be caused by (i) the lack of second-order gradients, or (ii) the fact that an LSTM is used as an optimizer. To investigate these hypotheses, we introduce TURTLE, which is similar to the meta-learner LSTM but uses a fully-connected feed-forward network as an optimizer instead of an LSTM and, in addition, uses second-order gradients. Although both MAML and the meta-learner LSTM are by now surpassed by other state-of-the-art techniques, such as LEO (Rusu et al., 2019) and MetaOptNet (Lee et al., 2019) (see Section 3.2), they are still relevant and widely used. The aim of this chapter is to gain insight into the performance gap between the meta-learner LSTM and MAML, leading to the following contributions:

- We formally show that the meta-learner LSTM subsumes MAML.
- We formulate a new meta-learning algorithm called TURTLE to overcome two potential shortcomings of the meta-learner LSTM. We demonstrate that TURTLE successfully closes the performance gap to MAML as it outperforms MAML (and the meta-learner LSTM) on sine wave regression, and various settings involving miniImageNet and CUB by at least 1% accuracy without any additional hyperparameter tuning. TURTLE requires roughly the same amount of computation time as second-order MAML.
- Based on the results of TURTLE, we enhance the meta-learner LSTM by using raw gradients as meta-learner input and second-order information and show these changes result in a performance boost of 1-6% accuracy, indicating the importance of second-order gradients.

3.2 Related work

The success of deep learning techniques has been largely limited to domains where abundant data and large compute resources are available (LeCun et al., 2015). The reason for this is that learning a new task requires large amounts of resources. Meta-learning is an approach that holds the promise of relaxing these requirements by learning to learn. The field has attracted much attention in recent years, resulting in many new techniques, which can be divided into metric-based, model-based, and optimization-based approaches (see Chapter 2). In our work, we focus on an optimization-based approach, which includes both MAML and the meta-learner LSTM (see Figure 2.28).

MAML (Finn et al., 2017) aims to find a good weight initialization from which new tasks can be learned quickly within several gradient update steps. As shown in Figure 2.28, many works build upon the key idea of MAML, for example, to decrease the computational costs (Nichol et al., 2018; Rajeswaran et al., 2019), increase the applicability to online and active learning settings (Grant et al., 2018; Finn et al., 2018), or increase the expressivity of the algorithm (Li et al., 2017; Park and Oliva, 2019a; Lee and Choi, 2018). Despite its popularity, MAML does no longer yield state-of-the-art performance on few-shot learning benchmarks (Lu et al., 2020), as it is surpassed by, for example, latent embedding optimization (LEO) (Rusu et al., 2019) which optimizes the initial weights in a lower-dimensional latent space, and MetaOptNet (Lee et al., 2019), which stacks a convex model on top of the meta-learned initialization of a high-dimensional feature extractor. Although these approaches achieve state-of-the-art techniques on few-shot benchmarks, MAML is

elegant and generally applicable as it can also be used in reinforcement learning settings (Finn et al., 2017).

While the meta-learner LSTM (Ravi and Larochelle, 2017) learns both an initialization and an optimization procedure, it is generally hard to properly train the optimizer (Metz et al., 2019; Simons, 2022). As a result, techniques that use hand-crafted learning rules instead of trainable optimizers may yield better performance. It is perhaps for this reason that most meta-learning algorithms use simple, hand-crafted optimization procedures to learn new tasks, such as regular gradient descent (Bottou, 2004), Adam (Kingma and Ba, 2015), or RMSprop (Tieleman and Hinton, 2017). Andrychowicz et al. (2016), show that learned optimizers may learn faster and yield better performance than gradient descent.

The goal of our work is to investigate why MAML often outperforms the meta-learner LSTM, while the latter is at least as expressive as the former (see Section 3.4.1).[2] Finn and Levine (2018) have shown that the reverse also holds: MAML can approximate any learning algorithm. However, this theoretical result only holds for sufficiently deep base-learner networks. Thus, for a given network depth, it does not say that MAML subsumes the meta-learner LSTM. In contrast, our result that the meta-learner LSTM subsumes MAML holds for any base-learner network and depth.

In order to investigate the performance gap between the meta-learner LSTM and MAML, we propose TURTLE which replaces the LSTM module from the meta-learner LSTM with a feed-forward neural network. Note that Metz et al. (2019) also used a regular feed-forward network as an optimizer. However, they were mainly concerned with understanding and correcting the difficulties that arise from training an optimizer and do not learn a weight initialization for the base-learner network as we do. Baik et al. (2020) also use a feed-forward network on top of MAML but its goal is to generate a per-step learning rate and weight decay coefficients. The feed-forward network in TURTLE, in contrast, generates direct weight updates.

3.3 Preliminaries

In this section, we explain the notation and the concepts of the two methods that we investigate (MAML and the meta-learner LSTM). Whilst these methods are also described in Chapter 2, here we present the methods in a way that is helpful for this chapter, emphasizing certain properties using slightly different notation.

3.3.1 MAML

As mentioned before, MAML (Finn et al., 2017) attempts to learn a set of initial neural network parameters $\boldsymbol{\theta}$ from which we can quickly learn new tasks within T steps of gradient descent, for a small value of T. Thus, given a task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})$, MAML will produce a sequence of weights $(\boldsymbol{\theta}_j^{(0)}, \boldsymbol{\theta}_j^{(1)}, \boldsymbol{\theta}_j^{(2)}, ..., \boldsymbol{\theta}_j^{(T)})$, where

$$\boldsymbol{\theta}_j^{(t+1)} = \boldsymbol{\theta}_j^{(t)} - \alpha \nabla_{\boldsymbol{\theta}_j^{(t)}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}_j^{(t)}). \tag{3.1}$$

Here, α is the inner learning rate and $\mathcal{L}_D(\boldsymbol{\varphi})$ the loss of the network with weights $\boldsymbol{\varphi}$ on dataset D. Note that the first set of weights in the sequence is equal to the initialization, i.e., $\boldsymbol{\theta}_j^{(0)} = \boldsymbol{\theta}$.

[2]Link to our code: `https://github.com/mikehuisman/revisiting-learned-optimizers`

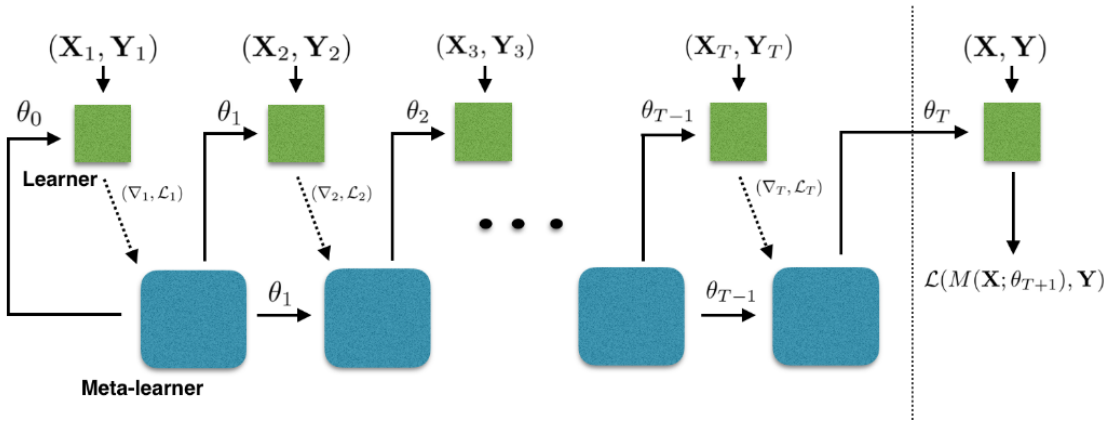


Figure 3.1: The workflow of the meta-learner LSTM (Ravi and Larochelle, 2017). The base-learner parameters are updated by an LSTM meta-learner network. The base-learner is denoted as M. (X_t, Y_t) are support sets, whereas (X, Y) is the query set. Note that the figure uses subscripts to indicate time steps and not tasks, i.e., θ_T are the parameters at step T and not the initial parameters for task T.

Given a distribution of tasks $p(\mathcal{T})$, we can formalize the objective of MAML as finding the initial parameters

$$\arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})} \left[\mathcal{L}_{D^{te}_{\mathcal{T}_j}} (\boldsymbol{\theta}_j^{(T)}) \right]. \tag{3.2}$$

Note that the loss is taken with respect to the query set, whereas $\boldsymbol{\theta}_j^{(T)}$ is computed on the support set $D^{tr}_{\mathcal{T}_J}$.

The initialization parameters $\boldsymbol{\theta}$ are updated by optimizing this objective in Equation 3.2, where the expectation over tasks is approximated by sampling a batch of tasks. Importantly, updating these initial parameters requires backpropagation through the optimization trajectories on the tasks from the batch. This implies the computation of second-order derivatives, which is computationally expensive. However, Finn et al. (2017) have shown that first-order MAML, which ignores these higher-order derivatives and is computationally less demanding, works just as well as the complete, second-order MAML version.

3.3.2 Meta-learner LSTM

The meta-learner LSTM by Ravi and Larochelle (2017) can be seen as an extension of MAML as it does not only learn the initial parameters $\boldsymbol{\theta}$ but also the optimization procedure which is used to learn a given task. Note that MAML only uses a single base-learner network, while the meta-learner LSTM uses a separate meta-network to update the base-learner parameters, as shown in Figure 3.1. Thus, instead of computing $(\boldsymbol{\theta}_j^{(0)}, \boldsymbol{\theta}_j^{(1)}, \boldsymbol{\theta}_j^{(2)}, ..., \boldsymbol{\theta}_j^{(T)})$ using regular gradient descent as done by MAML, the meta-learner LSTM learns a procedure that can produce such a sequence of updates, using a separate meta-network.

This trainable optimizer takes the form of a special LSTM module, which is applied to every weight in the base-learner network after the gradients and loss are computed on the support set. The idea

is to embed the base-learner weights into the cell state $\boldsymbol{c}$ of the LSTM module. Thus, for a given task $\mathcal{T}_j$, we start with cell state $\boldsymbol{c}_j^{(0)} = \boldsymbol{\theta}$. After this initialization phase, the base-learner parameters (which are now inside the cell state) are updated as

$$\begin{aligned}\boldsymbol{c}_j^{(t+1)} = & \underbrace{\sigma\left(\boldsymbol{W}_{\boldsymbol{f}} \cdot [\nabla_{\boldsymbol{\theta}_j^{(t)}}, \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}, \boldsymbol{\theta}_j^{(t)}, \boldsymbol{f}_j^{(t-1)}] + \boldsymbol{b}_{\boldsymbol{f}}\right)}_{\text{weight decay}} \odot \boldsymbol{c}_j^{(t)} \\ & + \underbrace{\sigma\left(\boldsymbol{W}_{\boldsymbol{i}} \cdot [\nabla_{\boldsymbol{\theta}_j^{(t)}}, \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}, \boldsymbol{\theta}_j^{(t)}, \boldsymbol{i}_j^{(t-1)}] + \boldsymbol{b}_{\boldsymbol{i}}\right)}_{\text{learning rate}} \odot \bar{\boldsymbol{c}}_j^{(t)}, \end{aligned} \tag{3.3}$$

where $\odot$ is the element-wise product, the two sigmoid factors σ are the parameterized forget gate $\boldsymbol{f}_j^{(t)}$ and learning rate $\boldsymbol{i}_j^{(t)}$ vectors that steer the learning process, $\nabla_{\boldsymbol{\theta}_j^{(t)}} = \nabla_{\boldsymbol{\theta}_j^{(t)}} \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\boldsymbol{\theta}_j^{(t)})$, $\mathcal{L}_{D_{\mathcal{T}_j}^{tr}} = \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\boldsymbol{\theta}_j^{(t)})$, and $\bar{\boldsymbol{c}}_j^{(t)} = -\nabla_{\boldsymbol{\theta}_j^{(t)}} \mathcal{L}_{D_{\mathcal{T}_j}^{tr}}(\boldsymbol{\theta}_j^{(t)})$. Both the learning rate and forget gate vectors are parameterized by weight matrices $\boldsymbol{W}_f, \boldsymbol{W}_i$ and bias vectors $\boldsymbol{b}_{\boldsymbol{f}}$ and $\boldsymbol{b}_{\boldsymbol{i}}$, respectively. These parameters steer the inner learning on tasks and are updated using regular, hand-crafted optimizers after every meta-training task. As noted by Ravi and Larochelle (2017), this is equivalent to gradient descent when $\boldsymbol{c}^{(t)} = \boldsymbol{\theta}_j^{(t)}$, and the sigmoidal factors are equal to $\mathbf{1}$ and $\boldsymbol{\alpha}$, respectively.

In spite of the fact that the LSTM module is applied to every weight individually to produce updates, it does maintain a separate hidden state for each of them. In a similar fashion to MAML, updating the initialization parameters (and LSTM parameters) would require propagating backwards through the optimization trajectory for each task. To circumvent the computational costs associated with this expensive operation, the meta-learner LSTM assumes that input gradients and losses are *independent* of the parameters in the LSTM.

3.4 Towards stateless neural meta-learning

In this section, we study the theoretical relationship between MAML and the meta-learner LSTM. Based on the resulting insight, we formulate a new meta-learning algorithm called TURTLE (stateless neural meta-learning) which is simpler than the meta-learner LSTM and more expressive than MAML.

3.4.1 Theoretical relationship

There is a subsumption relationship between MAML and the LSTM meta-learner. The gradient update rule used by MAML uses a fixed learning rate and no weight decay. The LSTM meta-learner, on the other hand, can learn a dynamic weight decay and learning rate schedule. These observations gives rise to the following theorem:

Theorem 1. *The meta-learner LSTM subsumes MAML*

Proof. We prove this theorem by showing that there is a parameterization of the LSTM meta-learner such that it updates the base-learner weights using gradient descent with a fixed learning rate α and without weight decay. In other words, we show that there exist $\boldsymbol{W}_f, \boldsymbol{b}_{\boldsymbol{f}}, \boldsymbol{W}_{\boldsymbol{i}}, \boldsymbol{b}_{\boldsymbol{i}}$ such that the update

made by the LSTM meta-learner is equivalent to that made by MAML

$$\boldsymbol{c}_j^{(t+1)} = \boldsymbol{\theta}_j^{(t)} - \alpha \nabla_{\boldsymbol{\theta}_j^{(t)}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}_j^{(t)}) \tag{3.4}$$

$$= \mathbf{1} \odot \boldsymbol{\theta}_j^{(t)} + \alpha \mathbf{1} \odot -\nabla_{\boldsymbol{\theta}_j^{(t)}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}_j^{(t)}). \tag{3.5}$$

The update of the meta-learner LSTM (Equation 3.3) satisfies this relationship when $\boldsymbol{c}_j^{(0)} = \boldsymbol{\theta}$ (satisfied by construction), the weight decay is equal a vector of ones $\mathbf{1}$, and the learning rate to $\alpha\mathbf{1}$. The weight decay condition condition can be met by setting $\boldsymbol{W_f}$ to a matrix of zeros and $\mathbf{b}_f$ to vector of sufficiently large values to push the output of the sigmoid near its saturation point (1). Since the learning rate $0 < \alpha < 1$ falls within the codomain of the sigmoid function, the learning rate condition can also be met by setting $\boldsymbol{W_i}$ to a matrix of zeros and $\boldsymbol{b}_f = -\ln(\frac{1-\alpha}{\alpha})\mathbf{1}$. Thus, we have shown that it is possible to parameterize the LSTM meta-learner so that it mimics gradient descent with any learning rate $0 < \alpha \leq 1$. □

3.4.2 Potential problems of the meta-learner LSTM

The theoretical insight that meta-learner LSTM subsumes MAML is not congruent with empirical findings which show that MAML outperforms the meta-learner LSTM on the miniImageNet image classification benchmark (Finn et al., 2017; Ravi and Larochelle, 2017), indicating that LSTM is unable to successfully navigate the error landscape to find a solution at least as good as the one found by MAML.

A potential cause is that the meta-learner LSTM attempts to learn a *stateful* optimization procedure, allowing it to employ dynamic weight decay and learning rate schedules for learning new tasks. While this gives the technique more flexibility in learning an optimization algorithm, it may also negatively affect meta-optimization as it may be harder to find a good dynamic optimization algorithm than a static one because the space of dynamic algorithms is less constrained. In addition, we conjecture that the loss landscape for dynamic algorithms is less smooth because the weight decay and learning rate schedules, which can have a large influence on the performance, depend on parameter trajectories (paths from initialization to task-specific parameters). We hypothesize that removing the stateful nature of the trainable optimizer may smoothen the meta-landscape as it constrains the space of possible solutions and removes the dependency of the learning rate on parameter trajectories, which can stabilize learning. For this reason, we replace the LSTM module in TURTLE with a regular fully-connected, feed-forward network, which is stateless.

Another potential cause of the underperformance could be the first-order assumption made by the meta-learner LSTM, which we briefly mentioned in Section 3.3.2. Effectively, this disconnects the computational graph by stating that weight updates made at time step t by the meta-network do not influence the inputs that this network receives at future time steps $t < t' < T$. Consequently, the algorithm ignores curvature information which can be important for stable training. While first-order MAML achieves similar performance to MAML, we think that the loss landscape of the LSTM meta-learner is less smooth (for reasons mentioned above), which can exacerbate the harmful effect of the first-order assumption. To overcome this issue, we use second-order gradients by default in TURTLE and investigate the effect of making the first-order assumption.

3.4.3 TURTLE

In an attempt to make the meta-landscape easier to navigate, we introduce a new algorithm, TURTLE, which trains a feed-forward meta-network to update the base-learner parameters. TURTLE is simpler than the meta-learner LSTM as it uses a *stateless* feed-forward neural network as a trainable optimizer, yet more expressive than MAML as its meta-network can learn to perform gradient descent.

The trainable optimizer in TURTLE is thus a fully-connected feed-forward neural network. We denote the batch of inputs that this network receives at time step t in the inner loop for task $\mathcal{T}_j$ as $I_j^{(t)} \in \mathbb{R}^{n \times d}$, where n and d are the number of base-learner parameters and the dimensionality of the inputs, respectively. The exact inputs that this network receives will be determined empirically, but two choices, inspired by the meta-learner LSTM, are: (i) the gradients with respect to all parameters and (ii) the current loss (repeated n times for each parameter in the base-network).

Moreover, we could mitigate the absence of a state in the meta-network by including a time step $t \in \{0, 1, ..., T-1\}$ and/or historical information such as a moving average of previous gradients or updates made by the meta-network. We denote the latter by $\boldsymbol{h}_j^{(t)}$ which is updated by

$$\boldsymbol{h}_j^{(t+1)} = \beta \boldsymbol{h}_j^{(t)} + (1 - \beta)\boldsymbol{v}_j^{(t)}, \tag{3.6}$$

where $0 \leq \beta \leq 0$ is a constant that determines the time span over which previous inputs affect the new state $\boldsymbol{h}_j^{(t+1)}$, and $\boldsymbol{v}_j^{(t)} \in \mathbb{R}^n$ is the new information (either the updates or gradients at time step t). When using previous updates, we initialize $\boldsymbol{h}_j^{(0)}$ by a vector of zeros.

Weight updates are then computed as follows

$$\boldsymbol{\theta}_j^{(t+1)} = \boldsymbol{\theta}_j^{(t)} + \boldsymbol{\alpha} \odot g_{\boldsymbol{\phi}}(I_j^{(t)}), \tag{3.7}$$

where $\boldsymbol{\alpha} \in \mathbb{R}^n$ is a vector of learning rates per parameter. Note that this weight update equation is simpler than the one used by the meta-learner LSTM (see Equation 3.3) as our meta-network $g_{\boldsymbol{\phi}}$ is stateless. Therefore, we do not have parameterized forget and input gates. Moreover, the learning rates per parameter in $\boldsymbol{\alpha}$ are not constrained to be within the interval $[0, 1]$ as is the case for the meta-learner LSTM due to the use of the sigmoid function.

In Algorithm 9 we show, in different colors, the code for MAML (red), the meta-learner LSTM (blue), and TURTLE (green). Although the code structure of the three meta-learners is similar, the update rules are quite different. Both the base- and meta-learner parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are updated by backpropagation through the optimization trajectories (line 11).

3.5 Experiments

In this section, we describe our experimental setup and the results that we obtained.

3.5.1 Hyperparameter analysis on sine-wave regression

Here, we investigate the effect of the order of information (first- versus second-order), the number of updates T per task, and further increasing the number of layers of the meta-network on the performance of TURTLE on 5-shot sine wave regression. The results are displayed in Figure 3.2. Note that in this experiment, we fixed the learning rate vector $\boldsymbol{\alpha}$ to be a vector of ones, which means

Algorithm 9 MAML meta-learner LSTM TURTLE

Initialize parameters $\boldsymbol{\Theta} =$ $\{\boldsymbol{\theta}\}$ $\{\boldsymbol{\theta}, \boldsymbol{\phi}\}$ $\{\boldsymbol{\theta}, \boldsymbol{\phi}\}$
Initialize $g_{\boldsymbol{\phi}}$ as N.A. LSTM feed-forward network
repeat
  Sample batch of J tasks $B = \{\mathcal{T}_j \backsim p(\mathcal{T})\}_{j=1}^{J}$
  for $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{te})$ in B **do**
    $\boldsymbol{\theta}_j^{(0)} = \boldsymbol{\theta}$
    for $t = 1, ..., T$ **do**
      Update $\boldsymbol{\theta}_j^{(t)}$ using Eq. 3.1 Eq. 3.3 Eq. 3.7
    end for
  end for
  Update $\boldsymbol{\Theta}$ using $\sum_{\mathcal{T}_j \in B} \mathcal{L}_{D_{\mathcal{T}_j}^{te}}(\boldsymbol{\theta}_j^{(T)})$
until convergence

that the updates proposed by the meta-network are directly added to the base-learner parameters without any scaling. Moreover, the only input that the meta-network receives is the gradient of the loss on the support set with respect to a base-learner parameter, and every hidden layer of the meta-network consists of 20 nodes followed by ReLU nonlinearities.

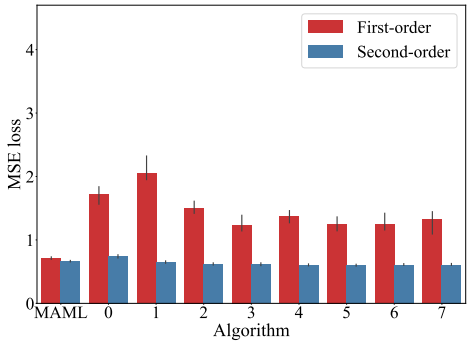
As we can see, the difference between first- and second-order MAML is relatively small, which was also found by Finn et al. (2017). In contrast, this is not the case for TURTLE, where the first-order variant fails to achieve a similar performance as second-order TURTLE. Furthermore, we see that the stability of TURTLE decreases as T increases as the confidence intervals become larger and the performance with fewer hidden layers deteriorates. Lastly, we find that 5 or 6 hidden layers yield the best performance across different values of T. For this reason, all further TURTLE experiments will be conducted with a meta-network of 5 hidden layers.

3.5.2 Few-shot sine wave regression

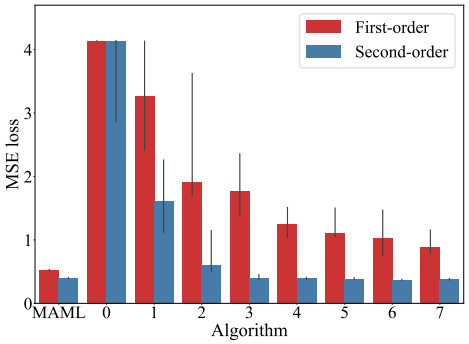
First, we compare the performance of TURTLE to that of MAML and the meta-learner LSTM on sine wave regression, which was originally proposed by Finn et al. (2017). We follow their experimental setup and use 70K, 1K, and 2K tasks for training, validation, and testing respectively. All experiments are performed 30 times with different random weight initializations of the base- and meta-learner networks. We perform meta-validation every 2.5K tasks for hyperparameter tuning. The meta-test results of this experiment are displayed in Figure 3.3. As we can see, TURTLE, which uses second-order gradients, systematically outperforms the meta-learner LSTM, which uses first-order gradients.

3.5.3 Few-shot image classification

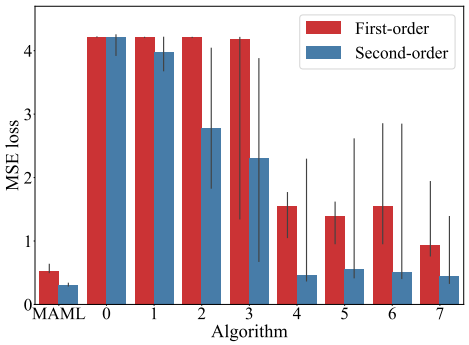
Without additional hyperparameter tuning, we now investigate the performance of 5-step TURTLE on few-shot image classification tasks, following the setup used in Chen et al. (2019). In addition, we investigate the importance of second-order gradients in this setting. For this, we use miniImageNet (Vinyals et al., 2016) (with the class splits proposed by Ravi and Larochelle (2017)) and CUB (Wah et al., 2011). We use the same base-learner network used by Snell et al. (2017) and Chen et al. (2019). Each algorithm is run with 5 different random weight initializations.



(a) $T = 1$



(b) $T = 5$



(c) $T = 10$

Figure 3.2: Influence of the order, number of update steps, and number of hidden layers (horizontal axis) on the meta-validation performance of TURTLE on 5-shot sine wave regression. We also plot the performance of first- and second-order MAML for comparison. Note that a lower MSE loss corresponds to better performance. The vertical bars indicate the 95% confidence intervals.

We compare the performance against three simple transfer-learning models, following Chen et al. (2019): train from scratch, finetuning, and baseline++. Based on our hyperparameter experiments for TURTLE, we also investigate an enhanced version of the meta-learner LSTM which uses raw gradients as meta-learner input and second-order information. The meta-test accuracy scores on 5-way miniImageNet and CUB classification are displayed in Table 3.1. Note that we use the best-reported hyperparameters for MAML and the meta-learner LSTM on miniImageNet, while we use the best hyperparameters found on sine wave regression for TURTLE. Despite this, TURTLE and second-order meta-learner LSTM outperform MAML and other techniques in 50% of the tested scenarios while they yield the competitive performance in the other scenarios. As we can see, the performances of all models are better on 5-shot classification compared with 1-shot classification. Looking at the results for miniImageNet, we see that the addition of second-order gradients increases the performance of both the meta-learner LSTM and TURTLE. An overview of the exact hyperparameter values that

[3]Our enhanced version of the meta-learner LSTM, which takes raw gradients as inputs, uses second-order gradients, and makes 8 updates per task.

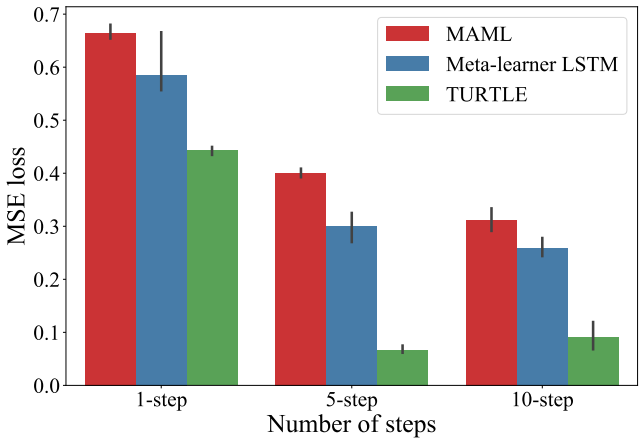


Figure 3.3: Median meta-test performance of MAML, the meta-learner LSTM, and TURTLE on 5-shot sine wave regression. Note that a lower error indicates better performance. The vertical bars indicate the 95% confidence intervals.

were used for all techniques can be found in Appendix A.

3.5.4 Cross-domain few-shot learning

We also investigate the robustness of the meta-learning algorithms when a task distribution shift occurs. For this, we train the techniques on miniImageNet and evaluate their performance on CUB tasks (and vice versa), following Chen et al. (2019). This is a challenging setting that requires a more general learning ability than for the experiments above. The results are shown in Table 3.4. Also in these challenging scenarios, second-order gradients are important to increase the performance of both the meta-learner LSTM and TURTLE. More specifically, the omission of second-order gradients can lead to large performance penalties, ranging from 1% to 5% accuracy.

3.5.5 Running time comparison

Lastly, we compare the running times of MAML, the meta-learner LSTM, and TURTLE on miniImageNet and CUB. A run comprises the time it costs to perform meta-training, meta-validation, and meta-testing on miniImageNet, and evaluation on CUB. We measure the average time in full hours across 5 runs on nodes with a Xeon Gold 6126 2.6GHz 12 core CPU and PNY GeForce RTX 2080TI GPU. The results are displayed in Figure 3.4. As we can see, the first-order algorithms are the fastest, while the second-order algorithms are slower (so-MAML and TURTLE). However, the performance of the first-order meta-learner LSTM and first-order TURTLE is worse than that of the second-order variants, indicating the importance of second-order gradients. For MAML, we do not observe such a difference between the first- and second-order variants. TURTLE is, despite its name, not much slower than the second-order MAML (SO-MAML), indicating that the time complexity is dominated by learning the base-learner initialization parameters. In fact, we observe that TURTLE is slightly faster than MAML, indicating that our implementation of the latter is not optimally efficient. In addition, we note that TURTLE is faster than the second-order (enhanced)

Table 3.1: Median meta-test accuracy scores and 95% confidence intervals over 5 runs of 5-way image classification on miniImageNet (left) and CUB (right). The best performance is displayed in bold font. Note that a higher accuracy indicates better performance.

	miniImageNet		CUB	
	1-shot	5-shot	1-shot	5-shot
TrainFromScratch	0.29 ± 0.00	0.40 ± 0.00	0.30 ± 0.00	0.46 ± 0.00
Finetuning	0.38 ± 0.00	0.56 ± 0.00	0.33 ± 0.01	0.53 ± 0.01
Baseline++	0.44 ± 0.00	0.58 ± 0.00	0.36 ± 0.01	0.53 ± 0.01
Meta-learner LSTM	0.45 ± 0.01	0.61 ± 0.00	0.50 ± 0.00	0.65 ± 0.01
Meta-learner LSTM^3	**0.48** ± 0.01	**0.63** ± 0.01	**0.53** ± 0.01	0.71 ± 0.00
FO-MAML	0.46 ± 0.01	**0.63** ± 0.00	0.52 ± 0.00	0.72 ± 0.01
MAML	0.47 ± 0.01	**0.63** ± 0.00	0.52 ± 0.00	**0.73** ± 0.01
First-order TURTLE	0.43 ± 0.01	0.59 ± 0.04	0.50 ± 0.00	0.64 ± 0.03
TURTLE	**0.48** ± 0.01	0.62 ± 0.01	**0.53** ± 0.00	0.72 ± 0.01

Table 3.2: Median meta-test accuracy scores and 95% confidence intervals over 5 runs of 5-way image classification on miniImageNet (left) and CUB (right). The best performance is displayed in bold font. Note that a higher accuracy indicates better performance.

	miniImageNet		CUB	
	1-shot	5-shot	1-shot	5-shot
Meta-learner LSTM	0.45 ± 0.01	0.61 ± 0.00	0.50 ± 0.00	0.65 ± 0.01
Meta-learner LSTM*	**0.48** ± 0.01	**0.63** ± 0.01	**0.53** ± 0.01	0.71 ± 0.00
FO-MAML	0.46 ± 0.01	**0.63** ± 0.00	0.52 ± 0.00	0.72 ± 0.01
MAML	0.47 ± 0.01	**0.63** ± 0.00	0.52 ± 0.00	**0.73** ± 0.01
First-order TURTLE	0.43 ± 0.01	0.59 ± 0.04	0.50 ± 0.00	0.64 ± 0.03
TURTLE	**0.48** ± 0.01	0.62 ± 0.01	**0.53** ± 0.00	0.72 ± 0.01

LSTM meta-learner.

3.6 Discussion and future work

In this chapter, we have formally shown that the meta-learner LSTM (Ravi and Larochelle, 2017) subsumes MAML (Finn et al., 2017). Experiments of Finn et al. (2017) and ourselves, however, show that MAML outperforms the meta-learner LSTM. We formulated two hypotheses for this surprising finding and, in turn, we formulated a new meta-learning algorithm named TURTLE, which is simpler than the meta-learner LSTM as it is stateless, yet more expressive than MAML because it can learn the weight update rule as it features a separate meta-network.

We empirically demonstrate that TURTLE is capable of outperforming both MAML and the (first-order) meta-learner LSTM on sine wave regression and—without additional hyperparameter tuning—on the frequently used miniImageNet benchmark. This shows that better update rules exist for fast adaptation than regular gradient descent, which is in line with findings by Andrychowicz

Table 3.3: Median meta-test accuracy scores and 95% confidence intervals over 5 runs of 5-way image classification on miniImageNet (left) and CUB (right). The best performance is displayed in bold font. Note that a higher accuracy indicates better performance.

	miniImageNet		CUB	
	1-shot	5-shot	1-shot	5-shot
Meta-learner LSTM	0.45 ± 0.01	0.61 ± 0.00	0.50 ± 0.00	0.65 ± 0.01
MAML	0.47 ± 0.01	**0.63** ± 0.00	0.52 ± 0.00	**0.73** ± 0.01
TURTLE	**0.48** ± 0.01	0.62 ± 0.01	**0.53** ± 0.00	0.72 ± 0.01

Table 3.4: Median meta-test accuracy scores in 5-way cross-domain classification (train on tasks from one dataset and evaluate on tasks from another dataset). The median accuracy and 95% confidence intervals were computed over 5 runs. The meta-learner LSTM^2 refers to our enhanced version of the meta-learner LSTM, which takes raw gradients as inputs, uses second-order gradients, and makes 8 updates per task.

	miniImageNet → CUB		CUB → miniImageNet	
	1-shot	5-shot	1-shot	5-shot
TrainFromScratch	0.30 ± 0.00	0.46 ± 0.00	0.29 ± 0.00	0.40 ± 0.00
Finetuning	0.33 ± 0.00	0.52 ± 0.01	0.29 ± 0.00	0.41 ± 0.00
Baseline++	0.35 ± 0.01	0.52 ± 0.01	0.26 ± 0.00	0.31 ± 0.01
Meta-learner LSTM	0.34 ± 0.01	0.52 ± 0.01	0.29 ± 0.01	0.37 ± 0.00
Meta-learner LSTM^2	0.37 ± 0.00	0.55 ± 0.01	**0.32** ± 0.01	0.43 ± 0.01
FO-MAML	0.34 ± 0.01	0.54 ± 0.01	0.31 ± 0.00	0.45 ± 0.01
MAML	0.35 ± 0.00	**0.56** ± 0.01	0.31 ± 0.00	**0.47** ± 0.00
fo-TURTLE	0.36 ± 0.00	0.54 ± 0.02	0.30 ± 0.00	0.31 ± 0.01
TURTLE	**0.38** ± 0.00	**0.56** ± 0.01	0.30 ± 0.00	0.44 ± 0.00

et al. (2016). Moreover, we enhanced the meta-learner LSTM by using raw gradients as meta-learner input and second-order gradient information, as they were found to be important for TURTLE. Our results indicate that this enhanced version of the meta-learner LSTM systematically outperforms the original technique by $1 - 6\%$ accuracy.

In short, these results show that second-order gradients are important for improving the few-shot image classification performance of the meta-learner LSTM and TURTLE, at the cost of additional runtime. In contrast, first-order MAML is a good approximation to second-order MAML as it yields similar performance (Finn et al., 2017). This finding supports our hypothesis that the loss landscape of MAML is smoother than that of meta-learning techniques that learn both the initialization parameters and a gradient-based optimization procedure.

Limitations and open challenges

While TURTLE and the enhanced meta-learner LSTM were shown to yield good performance, it has to be noted that this comes at the cost of increased computational expenses compared with first-order algorithms. That is, these second-order algorithms perform backpropagation through the

Table 3.5: Median meta-test accuracy scores in 5-way cross-domain classification (train on tasks from one dataset and evaluate on tasks from another dataset). The median accuracy and 95% confidence intervals were computed over 5 runs. The meta-learner LSTM^2 refers to our enhanced version of the meta-learner LSTM, which takes raw gradients as inputs, uses second-order gradients, and makes 8 updates per task.

	miniImageNet → CUB		CUB → miniImageNet	
	1-shot	5-shot	1-shot	5-shot
Meta-learner LSTM	0.34 ± 0.01	0.52 ± 0.01	0.29 ± 0.01	0.37 ± 0.00
Meta-learner LSTM*	0.37 ± 0.00	0.55 ± 0.01	**0.32** ± 0.01	0.43 ± 0.01
FO-MAML	0.34 ± 0.01	0.54 ± 0.01	0.31 ± 0.00	0.45 ± 0.01
MAML	0.35 ± 0.00	**0.56** ± 0.01	0.31 ± 0.00	**0.47** ± 0.00
fo-TURTLE	0.36 ± 0.00	0.54 ± 0.02	0.30 ± 0.00	0.31 ± 0.01
TURTLE	**0.38** ± 0.00	**0.56** ± 0.01	0.30 ± 0.00	0.44 ± 0.00

entire optimization trajectory which requires storing intermediate updates and the computation of second-order gradients. While this is also the case for MAML, it has been shown that first-order MAML achieves a similar performance whilst avoiding this expensive backpropagation process, yielding an excellent trade-off between performance and computational costs. For TURTLE, however, this is not the case, which means that other approaches should be investigated in order to reduce the computational costs. Future research may draw inspiration from Rajeswaran et al. (2019) who approximated second-order gradients in order to speed up MAML.

Our experiments also show that the training stability of TURTLE deteriorates as the number of inner updates increases. This is a known problem of meta-learning techniques that aim to learn the optimization algorithm. Metz et al. (2019) show that this instability is due to the fact that the meta-loss landscape becomes increasingly pathological as the number of inner updates increases. Future work is required to make it feasible to train such techniques for a large number of updates. Moreover, we note that we have only investigated the performances of MAML, TURTLE, and the LSTM meta-learner in 1- and 5-shot settings It would be interesting to investigate in future work how well MAML, TURTLE, and the LSTM meta-learner perform when more shots and ways (classes) are available per task. It may be possible that the performances of these techniques converge to the same point as the amount of available data increases. The intuition behind this is that given enough data, there is no need for a meta-learned prior to successfully learn the task.

Successfully using meta-learning algorithms in scenarios where task distribution shifts occur remains an important open challenge in the field of meta-learning. Our cross-domain experiment demonstrates that the learned optimization procedure by TURTLE generalizes to different tasks than the ones seen at training time, which is in line with findings by Andrychowicz et al. (2016). For this reason, we think that learned optimizers may be an important piece of the puzzle to broaden the applicability of meta-learning techniques to real-world problems. Future work can further investigate this hypothesis.

Our findings further show the benefit of learning an optimizer in addition to the initialization weights and highlight the importance of second-order gradients.

In this chapter, we have gained a deep understanding of the differences between two deep meta-

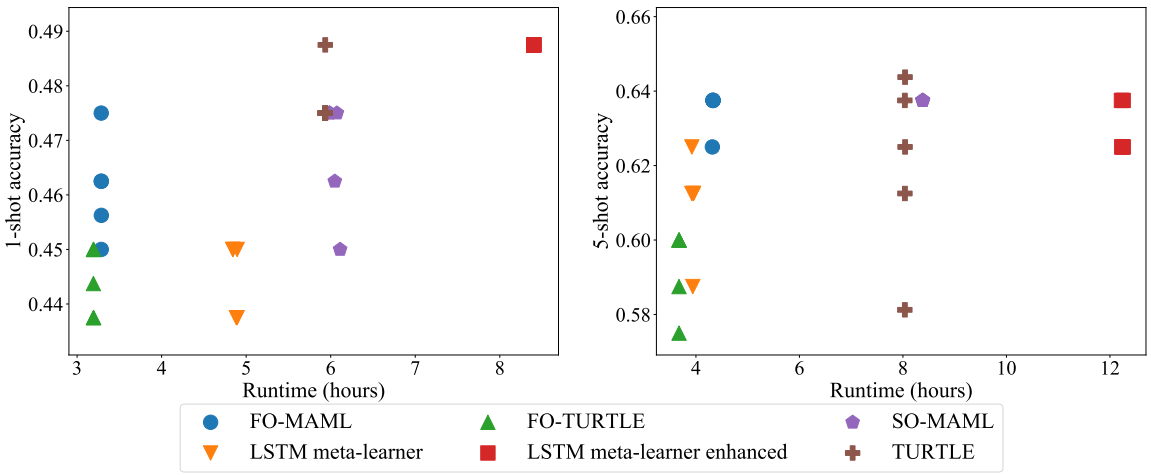


Figure 3.4: The running times and few-shot learning accuracy scores on 1-shot (left) and 5-shot (right) miniImageNet image classification of the different techniques for 5 runs with different random seeds.

learning algorithms (MAML and the meta-learner LSTM), which have been observed to be successful at few-shot learning in various scenarios. A related method for improving the learning efficiency of deep neural networks compared with the deep meta-learning algorithms (MAML, meta-learner LSTM) investigated in the previous research question, is transfer learning by means of pre-training and finetuning. The idea of this method is to train a neural network to perform a given source task for which abundant data is available. Recent results (Chen et al., 2019; Tian et al., 2020; Mangla et al., 2020) suggest that when evaluated on tasks from a different data distribution than the one used for training, the simple pre-training and finetuning baseline may be more effective than more complicated popular meta-learning techniques such as MAML and Reptile (Nichol et al., 2018). This is surprising as the learning behavior of MAML was shown to mimic that of finetuning: both rely on reusing learned features (Raghu et al., 2020). This begs the question of what causes the observed performance differences between these approaches. We investigate the differences in learning behaviors of finetuning, MAML, and Reptile in the next chapter.

Moreover, the study performed in this chapter prompts us to look deeper into using an LSTM for meta-learning. More specifically, the meta-learner LSTM uses an LSTM at the *meta-level*: to perform weight updates for a base-learner network. In 2001, however, Hochreiter et al. (2001) showed that an LSTM that operates at the *data level* trained with backpropagation across different tasks is capable of meta-learning. The LSTM operating at the data level is fed an entire training set, and predictions for query inputs are then conditioned on the resulting hidden state. This classical approach to meta-learning has received little attention in supervised few-shot learning scenarios, whilst it has been demonstrated to be successful at various reinforcement learning problems (Duan et al., 2016; Wang et al., 2016). In Chapter 5, We revisit this LSTM approach and test it on modern supervised few-shot learning benchmarks.

Chapter 4

Understanding Gradient-Based Meta-Learning and Transfer Learning

---

Chapter overview

Deep neural networks can yield good performance on various tasks but often require large amounts of data to train them. Meta-learning received considerable attention as one approach to improve the generalization of these networks from a limited amount of data. Whilst meta-learning techniques have been observed to be successful at this in various scenarios, recent results suggest that when evaluated on tasks from a different data distribution than the one used for training, a baseline that simply finetunes a pre-trained network may be more effective than more complicated meta-learning techniques such as MAML, which is one of the most popular meta-learning techniques. This is surprising as the learning behaviour of MAML mimics that of finetuning: both rely on re-using learned features. We investigate[1] the observed performance differences between finetuning, MAML, and another meta-learning technique called Reptile, and show that MAML and Reptile specialize for fast adaptation in low-data regimes of similar data distribution as the one used for training. Our findings show that both the output layer and the noisy training conditions induced by data scarcity play important roles in facilitating this specialization for MAML. Lastly, we show that the pre-trained features as obtained by the finetuning baseline are more diverse and discriminative than those learned by MAML and Reptile. Due to this lack of diversity and distribution specialization, MAML and Reptile may fail to generalize to out-of-distribution tasks whereas finetuning can fall back on the diversity of the learned features.

---

[1]This chapter is based on the following articles.

Huisman, M., Plaat, A. & van Rijn, J. N. (2021). A preliminary study on the feature representations of transfer learning and gradient-based meta-learning techniques. In Fifth Workshop on Meta-Learning at the Conference on Neural Information Processing Systems.

Huisman, M., Plaat, A. & van Rijn, J. N. (2023). Understanding Transfer Learning and Gradient-Based Meta-Learning Techniques. Accepted for publication in Machine Learning. Springer.

4.1 Introduction

Whilst the field of deep meta-learning has attracted much attention due to its promising prospect of enabling neural networks to learn from fewer data, recent results (Chen et al., 2019; Tian et al., 2020; Mangla et al., 2020) suggest that simply pre-training a network on a large dataset and *finetuning* only the final layer of the network (the final layer) may be more effective at learning new image classification tasks quickly than more complicated meta-learning techniques such as MAML (Finn et al., 2017) and Reptile (Nichol et al., 2018) when the data distribution is different from the one used for training. In contrast, MAML and Reptile often outperform finetuning when the data distribution is similar to the one used during training. These phenomena are not well understood and surprising as Raghu et al. (2020) have shown that the adaptation behaviour of MAML resembles that of finetuning when learning new tasks: most of the changes take place in the final layer of the network while the body of the network is mostly kept frozen.

In this chapter, we aim to find an explanation for the observed performance differences between MAML and finetuning. More specifically, we aim to answer the following two research questions:

1. Why do MAML and Reptile outperform finetuning in *within-distribution* settings?

2. Why can finetuning outperform gradient-based meta-learning techniques such as MAML and Reptile (Nichol et al., 2018) when the test data distribution diverges from the training data distribution?

Both questions focus on the **few-shot image classification settings**. We base our work on MAML, Reptile and finetuning, as these are influential techniques that have sparked a large body of follow-up methods that use the underlying ideas. Since the questions that we aim to answer are inherently harder than just a simple performance comparison, answering them for the models that are at the basis of this body of literature will be the right starting point. We think that developing a better understanding of these influential methods is of great value and can cascade further onto the more complex methods built on top of these.

Based on our analysis of the learning objectives of the three techniques (finetuning, MAML, Reptile), we hypothesize that MAML and Reptile specialize for adaptation in low-data regimes of tasks from the training distribution, giving them an advantage in within-distribution settings. However, since they may settle for initial features that are inferior compared with finetuning due to their negligence, or relative negligence, of the initial performance, they may perform comparatively worse when the test data distribution diverges from the training distribution.

The primary contributions of our work are the following. First, we show the importance of the output layer weights and data scarcity during training for Reptile and MAML to facilitate specialization for quick adaptation in low-data regimes of similar distributions, giving them an advantage compared with finetuning. Second, we show that the pre-trained features of the finetuning technique are more diverse and discriminative than those learned by MAML and Reptile, which can be advantageous in out-of-distribution settings.[2]

[2]All code for reproducing our results can be found at `https://github.com/mikehuisman/transfer-meta-feature-representations`

4.2 Related work

Meta-learning is a popular approach to enable deep neural networks to learn from a few data by learning an efficient learning algorithm. Many architectures and model types have been proposed, such as MAML (Finn et al., 2017), the meta-learner LSTM (Ravi and Larochelle, 2017), TURTLE (Chapter 3) and MetaOptNet (Lee et al., 2019). However, our understanding of newly proposed techniques remains limited in some cases. For example, different techniques use different backbones which raises the question of whether performance differences between techniques are due to new model-types or due to the difference in used backbones (see Chapter 2).

Chen et al. (2019) was one of the first that investigated this question by performing a fair comparison between popular meta-learning techniques, including MAML (Finn et al., 2017), on few-shot image classification benchmarks such as miniImageNet (Vinyals et al., 2016; Ravi and Larochelle, 2017) and CUB (Wah et al., 2011). Their results show that MAML often outperforms finetuning when the test tasks come from a similar data distribution as the training distribution when using shallow backbones. When the backbone becomes deeper and/or the domain differences between training and test tasks increase, however, this performance gap is reduced and, in some cases, finetuning outperforms MAML.

In addition to these findings by Chen et al. (2019), Tian et al. (2020) demonstrate that simply finetuning a pre-trained feature embedding module yields better performance than popular meta-learning techniques (including MAML) on few-shot benchmarks. Mangla et al. (2020) and Yang et al. (2021) further support this finding as they have proposed new few-shot learning techniques based on finetuning pre-trained networks which significantly outperform meta-learning techniques.

These performance differences between simple finetuning and more sophisticated techniques such as MAML may be surprising, as Raghu et al. (2020); Ding (2023) found that the learning behaviour of MAML is similar to that of finetuning on image classification benchmarks. More specifically, they compared the feature representations of MAML before and after task-specific adaptation, and show that MAML relies mostly on feature re-use instead of quick adaptation because the body of the network is barely adjusted, which resembles the learning dynamics of finetuning (see Section 4.3.3). Collins et al. (2020) compared the feature representations of MAML and the finetuning method (expected risk minimization) in linear regression settings and found that MAML finds an initialization closer to the hard tasks, characterized by their gentle loss landscapes with small gradients. We demonstrate a similar property: MAML has greater flexibility in picking an initialization as long as the post-adaptation performance is good.

In this chapter, we aim to unite the findings of Raghu et al. (2020) and Chen et al. (2019) by finding an answer to the question of why finetuning can outperform meta-learning techniques such as MAML and Reptile (Nichol et al., 2018) in some image classification scenarios while it is outperformed in other scenarios (when using a shallow backbone or when train/test task distributions are similar).

4.3 Background

In this section, we briefly revise supervised learning, few-shot learning (the main problem setting used in this chapter), finetuning, MAML, and Reptile, using slightly different notation compared with Chapter 2 to aid in understanding the rest of this chapter.

4.3.1 Supervised learning

In the *supervised learning* setting, we have a joint probability distribution over inputs $\mathbf{x}$ and corresponding outputs $\mathbf{y}$, i.e., $p(\mathbf{x}, \mathbf{y})$. In the context of deep learning, the goal is to build deep neural networks that can predict for any given input $\mathbf{x}$ the correct output $\mathbf{y}$. Throughout this chapter, we assume that the neural network architecture f is fixed and that we only wish to find a set of parameters θ such that the network predictions $f_\theta(\mathbf{x})$ are as good as possible. This can be done by updating the parameters θ in order to minimize a loss function $\mathcal{L}_{\mathbf{x}_i,\mathbf{y}_i}(\theta)$ that captures how well the network parameterized by θ is performing on input $\mathbf{x}_i$ and corresponding output $\mathbf{y}_i$. Here, network parameters θ are a weight matrix, where $\theta_{(i:j)}$ represent the weights of the i^{th} until the j^{th} layer (inclusive), where $0 < i < j \leq L$. Thus, under the joint distribution $p(\mathbf{x}, \mathbf{y})$, we wish to find

$$\arg\min_{\theta} \mathop{\mathbb{E}}_{\mathbf{x}_i,\mathbf{y}_i} \left[\mathcal{L}_{\mathbf{x}_i,\mathbf{y}_i}(\theta)\right], \tag{4.1}$$

where $(\mathbf{x}_i, \mathbf{y}_i)$ are sampled from the joint distribution $p(\mathbf{x}, \mathbf{y})$, i.e., $\mathbf{x}_i, \mathbf{y}_i \sim p(\mathbf{x}, \mathbf{y})$.

The most common way to approximate these parameters is by performing gradient descent on that loss function, which means that we update the parameters in the direction of the steepest descent

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_{\theta^{(t)}} \mathop{\mathbb{E}}_{\mathbf{x}_i,\mathbf{y}_i} \left[\mathcal{L}_{\mathbf{x}_i,\mathbf{y}_i}(\theta^{(t)})\right]. \tag{4.2}$$

Here, $\nabla_{\theta^{(t)}}$ is the gradient with respect to $\theta^{(t)}$, t indicates the time step, and α the learning rate or step size.

4.3.2 Few-shot learning

Few-shot learning is a special case of supervised learning, where the goal is to learn new tasks from only a limited number of examples, which is the main focus of this chapter and the techniques described below. In order to enhance the learning process on a limited number of examples, the learner is presented with an additional set of tasks, so that it can learn about the learning process. Here, every task $\mathcal{T}_j$ consists of a data distribution $p_j(\mathbf{x}, \mathbf{y})$ and a loss function $\mathcal{L}$. Since the loss function is often assumed to be fixed across all tasks, we henceforth use the term 'task' to refer to the task data distribution. The loss function is often assumed to be fixed, and therefore, we henceforth mean data distribution $p_j(\mathbf{x}, \mathbf{y})$ or a sample from this distribution, depending on the context. One notable exception is made in Section 4.5.1, where we abstract away from data distributions and define a task purely abstractly as a loss function.

Tasks are commonly sampled from a large meta-dataset $\mathcal{D} \backsim p_s(\mathbf{x}, \mathbf{y})$, which itself is a sample from a source distribution p_s. In the case of classification, this is often done as follows. Suppose that the source distribution from which dataset $\mathcal{D}$ is sampled, is defined over a set of classes $\mathcal{Y} = \{c_1, c_2, \ldots, c_n\}$. Then, we can create tasks $\mathcal{T}_j$ by considering only a subspace of this source distribution corresponding to a subset of classes $S_j \subseteq \mathcal{Y}$. The method can then be evaluated on tasks sampled from a disjoint subset of classes $S_m \subseteq \mathcal{Y}$, where $S_m \cap S_j =$.

Below, we give a concrete example of this procedure for the popular N**-way** k**-shot classification** setting (Finn et al., 2017; Vinyals et al., 2016; Snell et al., 2017). Suppose that we have a classification dataset $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y_1}), (\mathbf{x}_2, \mathbf{y_2}), \ldots, (\mathbf{x}_M, \mathbf{y_M})\}$ of examples. Then, we can create an N-way k-shot task $\mathcal{T}_j$ by sampling a subset of N labels $S_j \subseteq \mathcal{Y}$, where $|S_j| = N$. Moreover, we sample precisely k examples for every class to form a training set, or *support set* $D^{tr}_{\mathcal{T}_j}$, for that task, consisting of

$|D^{tr}_{\mathcal{T}_j}| = N \cdot k$ examples. Lastly, the test set, or *query set* $D^{te}_{\mathcal{T}_j}$, is obtained by sampling examples of the subset of classes S_j from $\mathcal{D}$ that are not present in the support set. Techniques then train on the support set and evaluated on the query set in order to measure how well they have learned the task. This is the problem setting that we will use throughout this chapter.

The deployment of an algorithm for few-shot learning is often done in three stages. In the *meta-training* stage, the algorithm is presented with training tasks and uses them to adjust the prior, such as the initialization parameters. After every X training tasks, the *meta-validation* stage takes place, where the learner is validated on unseen meta-validation tasks. Finally, after the training is completed, the learner with the best validation performance is evaluated in the *meta-test* phase, where the learner is confronted with new tasks that have not been seen during training and validation. Importantly, the tasks between meta-training, meta-validation, and meta-test phases are disjoint. For example, in image classification, the classes in the meta-training tasks are not allowed to occur in meta-test tasks as we are interested in measuring the learning ability instead of memorization ability. In regression settings, every task has its own ground-truth function (as in Section 4.5.1). For example, every task could be a sine wave with a certain phase and amplitude (Finn et al., 2017).

4.3.3 Finetuning

Achieving good generalization by minimizing the objective in Equation 4.1 using gradient-based optimization often requires large amounts of data. This raises the question of how we can perform few-shot learning of tasks. The transfer learning technique called *finetuning* tackles this problem as follows. In the *pre-training phase*, it minimizes Equation 4.1 on a given source distribution $p_s(\mathbf{x}, \mathbf{y})$ using gradient descent as shown in Equation 4.2. This leads to a sequence of updates that directly update the initialization parameters. Then, it freezes the feature extraction module of the network: all parameters of the network through the penultimate layer, i.e., $\theta_{(1:L-1)}$ where L is the number of layers. When presented with a target distribution $p_j(\mathbf{x}, \mathbf{y})$ from which we can sample fewer data, we can simply re-use the learned feature embedding module $f_{\theta_{(1:L-1)}}$ (all hidden layers of the network excluding the output layer) for this new problem. Then, in the *finetuning phase*, it only trains the parameters in the final layer of the network $\theta_{(L)}$ (the final layer).

By reducing the number of trainable parameters on the target problem, this technique effectively reduces the model complexity and prevents overfitting issues associated with the data scarcity in few-shot learning scenarios. This comes at the cost of not being able to adjust the feature representations of inputs. As a consequence, this approach fails when the pre-trained embedding module fails to produce informative representations of the target problem inputs.

4.3.4 Reptile

Instead of joint optimization on the source distribution, *Reptile* (Nichol et al., 2018) is a meta-learning algorithm and thus aims to learn how to learn. For this, it splits the source distribution $p_s(\mathbf{x}, \mathbf{y})$ into a number of smaller task distributions $p_1(\mathbf{x}, \mathbf{y}), p_2(\mathbf{x}, \mathbf{y}), \ldots, p_n(\mathbf{x}, \mathbf{y})$, corresponding to tasks $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$. On a single task $\mathcal{T}_j$ for $j \in \{1, \ldots, n\}$, its objective is to minimize Equation 4.1 under the task distribution $p_j(\mathbf{x}, \mathbf{y})$ using T gradient descent update steps as shown in Equation 4.2. This results in a sequence of weight updates $\theta \rightarrow \theta_j^{(1)} \rightarrow \ldots \rightarrow \theta_j^{(T)}$. After task-specific adaptation, the initial parameters θ are moved into the direction of $\theta_j^{(T)}$

$$\theta = \theta + \epsilon \left(\theta_j^{(T)} - \theta \right), \tag{4.3}$$

where ϵ is the step size. Intuitively, this update interpolates between the current initialization parameters θ and the task-specific parameters $\theta_j^{(T)}$. The updated initialization θ is then used as starting point when presented with new tasks, and the same process is repeated. It is easy to show that this update procedure corresponds to performing first-order optimization of the multi-step objective

$$\arg\min_{\theta} \mathop{\mathbb{E}}_{\mathcal{T}_j \sim p(\mathcal{T})} \left(\sum_{t=0}^{T-1} \mathop{\mathbb{E}}_{\mathbf{x}_i, \mathbf{y}_i \sim p_j} \left[\mathcal{L}_{t+1}(\theta_{\mathbf{j}}^{(\mathbf{t})}) \right] \right), \tag{4.4}$$

where $\mathcal{L}_{t+1}$ is shorthand for the loss on a mini-batch sampled at time step t.

4.3.5 MAML

Another popular gradient-based meta-learning technique is MAML (Finn et al., 2017). Just like Reptile, MAML also splits the source distribution $p_s(\mathbf{x}, \mathbf{y})$ into a number of smaller task distributions $p_1(\mathbf{x}, \mathbf{y}), p_2(\mathbf{x}, \mathbf{y}), \ldots, p_n(\mathbf{x}, \mathbf{y})$, corresponding to tasks $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$. On the training tasks, it aims to learn a weight initialization θ from which new tasks can be learned more efficiently. However, instead of optimizing a multi-step loss function, MAML only optimizes the final performance after task-specific adaptation. More specifically, this means that MAML is only interested in the performance of the final weights $\theta_j^{(T)}$ on a task and not in intermediate performances of weights $\theta_j^{(t)}$ for $t < T$. In other words, MAML aims to find

$$\arg\min_{\theta} \mathop{\mathbb{E}}_{\mathcal{T}_j \sim p(\mathcal{T})} \left(\mathop{\mathbb{E}}_{\mathbf{x}_i, \mathbf{y}_i \sim p_j} \left[\mathcal{L}_T(\theta_{\mathbf{j}}^{(\mathbf{T})}) \right] \right). \tag{4.5}$$

To find these parameters, MAML updates its initialization parameters as follows

$$\theta = \theta - \beta \nabla_\theta \mathcal{L}_{T+1}(\theta_j^{(T)}), \tag{4.6}$$

where β is the learning rate and $\nabla_\theta \mathcal{L}_{T+1}(\theta_j^{(T)}) = \nabla_{\theta_j^{(T)}} \mathcal{L}_{T+1}(\theta_j^{(T)}) \nabla_\theta \theta_j^{(T)}$. The factor $\nabla_\theta \theta_j^{(T)}$ contains second-order gradients and can be ignored by assuming that $\nabla_\theta \theta_j^{(T)} = I$ is the identity matrix, in a similar fashion to what Reptile does. This assumption gives rise to *first-order* MAML (fo-MAML) and significantly increases the training efficiency in terms of running time and memory usage, whilst achieving roughly the same performance as the *second-order* MAML version (Finn et al., 2017). In short, first-order MAML updates its initialization in the gradient update direction of the final task-specific parameters. In this chapter, we focus on first-order MAML, as Finn et al. (2017) have shown this to perform similarly to second-order MAML.

4.4 A common framework and interpretation

The three discussed techniques can be seen as part of a general gradient-based optimization framework, as shown in Algorithm 10. All algorithms try to find a good set of initial parameters as specified by their objective functions. The parameters are initialized randomly in line 1. Then, these initial parameters are iteratively updated based on the learning objectives (the loop starting from line 2).

This iterative updating procedure continues as follows. First, the data distribution is selected to

Table 4.1: Overview of the loss functions and corresponding focus of finetuning, Reptile, and MAML.

Algorithm	Loss function	Focus
Finetuning	$\mathop{\mathbb{E}}_{\mathbf{x}_i,\mathbf{y}_i}[\mathcal{L}_{\mathbf{x}_i,\mathbf{y}_i}(\theta)]$	Initial performance
Reptile	$\mathop{\mathbb{E}}_{\mathcal{T}_j\sim p(\mathcal{T})}\left(\sum_{t=0}^{T-1}\mathop{\mathbb{E}}_{\mathbf{x}_i,\mathbf{y}_i\sim p_j}\left[\mathcal{L}_{t+1}(\theta_{\mathbf{j}}^{(\mathbf{t})})\right]\right)$	Multi-step performance
MAML	$\mathop{\mathbb{E}}_{\mathcal{T}_j\sim p(\mathcal{T})}\left(\mathop{\mathbb{E}}_{\mathbf{x}_i,\mathbf{y}_i\sim p_j}\left[\mathcal{L}_{T}(\theta_{\mathbf{j}}^{(\mathbf{T})})\right]\right)$	Final performance

sample data from (line 3). That is, finetuning uses the full joint distribution $p_s(\mathbf{x},\mathbf{y})$ of the source problem, whereas Reptile and MAML select task distributions $p_j(\mathbf{x},\mathbf{y})$ (obtained by sub-sampling a set of instances coming from a subset of labels from the full distribution p_s). Next, we make T task-specific updates on mini-batches sampled from the distribution p that was selected in the previous stage (lines 4–8). Lastly, the initial parameters θ are updated using the outcomes of the task-specific adaptation phase.

Note that in this general gradient-based optimization framework, all techniques update their initialization parameters based on a single distribution p at a time. One could also choose to use batches of distributions, or *meta-batches*, in order to update the initialization θ. This can be incorporated by using the average of the losses of the different distributions as an aggregated loss function.

Algorithm 10 General gradient-based optimization: finetuning reptile MAML

1: Randomly initialize θ
2: **while** not converged **do**
3: Select data distribution $p =$ p_s $p_j \sim p(\mathcal{T})$ $p_j \sim p(\mathcal{T})$
4: Set $\theta^{(0)} = \theta$
5: **for** $t = 0, ..., T-1$ **do**
6: Sample a batch of data $\mathbf{x},\mathbf{y} \sim p$
7: Compute $\theta^{(t+1)} = \theta^{(t)} - \nabla_{\theta^{(t)}}\mathcal{L}_{t+1}(\theta^{(t)})$
8: **end for**
9: Update θ by $\theta = \theta^{(T)}$ Equation 4.3 Equation 4.6
10: **end while**

Table 4.1 gives an overview of the three algorithms. As we can see, finetuning only optimizes for the initial performance and does not take into account the performance after adaptation. This means that its goal is to correctly classify any input $\mathbf{x}$ from the source problem distribution p_s. Reptile, on the other hand, optimizes both for initial performance, as well as performance after every update step. This means that Reptile may settle for an initialization with somewhat worse initial performance compared with finetuning, as long as the performance during task-specific adaptation makes up for this initial deficit. MAML is the most extreme in the sense that it can settle for an initialization with poor initial performance, as long as the final performance is good.

In short, Reptile and MAML can be interpreted as *look-ahead algorithms* as they take the performance

after task-specific adaptation into account whereas finetuning does not. Moreover, fo-MAML relies purely on the look-ahead mechanism and neglects the initial performance while Reptile also takes the initial and intermediate performances into account. This means that MAML may outperform finetuning with a *low-capacity* network (with the worst initial performance) where there is not enough capacity to store features that are directly useful for new tasks. The reason for this is likely that finetuning will be unable to obtain good embeddings for all of the training tasks and does not have a mechanism to anticipate what features would be good to learn future tasks better. MAML, on the other hand, does have this capability, and can thus settle for a set of features with worse initial performance that lends itself better for learning new tasks. In contrast, when we have *high-capacity* networks with enough expressivity to store all relevant features for a task, finetuning may outperform MAML as it optimizes purely for initial performance without any additional adaptation, which can be prone to overfitting to the training data of the tasks due to the limited amount of available data. Lastly, one may expect Reptile to take place between MAML and finetuning: it works better than finetuning when using low-capacity backbones while it may be slightly worse than finetuning when using larger-capacity networks (but better than MAML).

Although MAML focuses on the performance after learning, it has been shown that its learning behaviour is similar to that of finetuning: it mostly relies on feature re-use and not on fast learning (Raghu et al., 2020). This means that when a *distribution shift* occurs, which means that the test tasks become more distant from the tasks that were used for training, MAML may be ill-positioned due to poor initial performance compared with finetuning which can fall back on more directly useful initial features.

4.5 Experiments

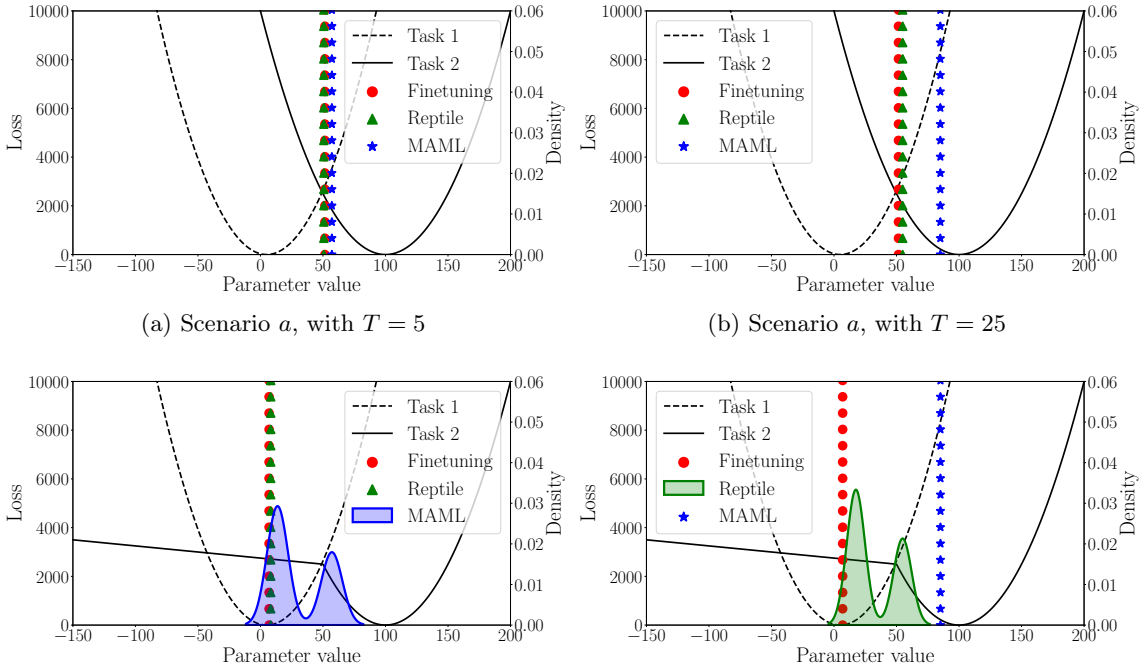
In this section, we perform various experiments to compare the learning behaviours of finetuning, MAML, and Reptile, in order to be able to study their within-distribution and out-of-distribution qualities that can help us answer the two research questions posed in Section 4.1. All experiments are conducted using single PNY GeForce RTX 2080TI GPUs. In order to study the question of why MAML and Reptile can outperform finetuning in within-distribution settings with a shallow Conv-4 backbone, we perform the following three first experiments. Moreover, to investigate why finetuning can outperform MAML and Reptile in out-of-distribution settings, addressing our second research question, we perform experiment four listed below.

1. **Toy problem** (Section 4.5.1) We study the behaviour of the algorithms on a *within-distribution* toy problems where there are only two tasks without noise in the loss signals caused by a shortage of training data. This allows us to investigate the initializations that the methods settle for after training. This allows us to see why MAML and Reptile may have an advantage over finetuning in within-distribution settings.

2. **The effect of the output layer** (Section 4.5.2) Finetuning removes the learned output layer and replaces it with a randomly initialized one when presented with a new task. MAML and Reptile, on the other hand, do not do this, and can directly start from the learned initialization weights for both the body and output layer of the network. To investigate whether this gives these two methods an advantage over finetuning in *within-distrbution* few-shot image classification, we investigate the effect of replacing the learned output layers with randomly initialized ones before learning a new task. This allows us to determine the importance of having a learned weight initialization for the output layer and whether this is something that can explain the advantage of MAML and Reptile over finetuning in these settings.

3. **Specialization for robustness against overfitting** (Section 4.5.2) Another difference between the methods is that finetuning is trained on regular mini-batches of data, whilst MAML and Reptile are trained explicitly for post-adaptation performance on noisy loss signals induced by the limited amount of available training data. To investigate the importance of explicitly training under noisy conditions, we study the performances of MAML and Reptile as a function of the number of examples present in the training condition. Here, the risk of overfitting is inversely related to the number of training examples k per task.

4. **Information content in the learned initializations** (Section 4.5.2) Lastly, we investigate the within-distribution and out-of-distribution learning performances of finetuning, MAML, and Reptile, with three different backbones of different expressive power (Conv-4, Resnet-10, Resnet-18). More specifically, we propose a measure of broadness or discriminative power of the features and investigate whether this is related to the few-shot learning abilities of these methods to see whether the discriminative power of the three methods differ and can account for the potential superiority of finetuning in the out-of-distribution setting.



(a) Scenario a, with $T = 5$

(b) Scenario a, with $T = 25$

(c) Scenario b, with $T = 5$

(d) Scenario b, with $T = 25$

Figure 4.1: Average initialization that finetuning, Reptile, and MAML converge to when using $T = 5$ or $T = 25$ adaptation steps per task. In scenario a (top figures), finetuning and Reptile both pick an initialization in the centre of the two optima where the initial loss is minimal. MAML neglects the initial performance and thus is freer to select an initialization point, especially when T is larger. In scenario b (bottom figures) the loss of task 2 is no longer convex and has a reasonably flat plateau. Finetuning and Reptile get stuck in the optimum of the first task and fail to learn the second task successfully, while MAML finds a location from which it can arrive at both optima.

4.5.1 Toy problem

First, we study the behaviour of finetuning, Reptile, and MAML in two synthetic scenarios a and b, consisting of two tasks each. In this subsection, we use a slightly more abstract notion of tasks compared with the rest of the text, and define tasks purely abstractly by loss functions. These tasks can be considered the meta-train set, and the goal of the algorithms is to find good initialization parameters on this task distribution. We represent tasks by their loss landscape, which we have constructed by hand for illustrative purposes. In scenario a, the two task loss landscapes are quadratic functions of a single parameter x. More specifically, the losses for this scenario are given by $\ell_1^a(x) = 1.3(x-5)^2$ and $\ell_2^a(x) = (x-100)^2$. In scenario b, the first task loss landscape is the same $\ell_1^b = \ell_1^a$ while the second task represents a more complex function:

$$\ell_2^b(x) = \begin{cases} (x-100)^2 & x > 50 \\ -5x + 2750 & x \leq 50 \end{cases} \tag{4.7}$$

The respective algorithms train by sampling tasks in an interleaved fashion, and by adapting the parameter x based on the loss landscape of the sampled task. We investigate the behaviour of Reptile and MAML when they make $T = 5$ or $T = 25$ task-specific adaptation steps. For this, we average the found solutions of the techniques over 100 different runs with initial x values that are equally spaced in the interval $[-200, +200]$. We find that finetuning converges to the same point regardless of the initialization and is thus represented by a single vertical line. For Reptile and MAML, the found solution depends on the initialization, which is why we represent the found solution as a probability density. A Jupyter notebook for reproducing these results can be found on our GitHub page.

Based on the learning objectives of the techniques, we expect finetuning to settle for an initialization that has a good initial performance on both tasks (small loss values). Furthermore, we expect that MAML will pick any initialization point from which it can reach minimal loss on both tasks within T steps. Reptile is expected to find a mid-way solution between finetuning and MAML.

The results of these experiments are displayed in Figure 4.1. In scenario a (top figures), we see that both finetuning and Reptile prefer an initialization at the intersection of the two loss curves, where the initial loss is minimal. MAML, on the other hand, neglects the initial performance when $T = 25$ and leans more to the right, whilst ensuring that it can reach the two optima within T steps. The reason that it prefers an initialization on the right of the intersection is that the loss landscape of task 1 is steeper, which means that task adaptation steps will be larger. Thus, a location at the right of the intersection ensures good learning of task 2 and yields comparatively fast learning on the first task.

In scenario b (bottom figures), the loss landscape of task 2 has a relatively flat plateau on the left-hand side. Because of this, finetuning and Reptile will be pulled towards the optimum (also the joint optimum) of the first task due to the larger gradients compared with the small gradients of the flat region of the second task when T is small. The solution that is found by MAML when $T = 5$ depends on the random initialization of the parameter, as can be seen in plot c). That is, when the random initialization is on the left of the plateau, MAML can not look beyond the flat region, implying that it will also be pulled towards the minimum of task 1. When $T = 25$, allowing the Reptile and MAML to look beyond the flat region, we see that Reptile either finds an initialization at $x = 50$ (when the starting point x_0 is on the right-hand side of the plateau) or at the joint optimum at $x = 0$ (when it starts with x_0 on the plateau). In the latter case, the post-adaptation performance of Reptile on both tasks is not optimal because it cannot reach the optimum of task 2. MAML, on the other hand, does not suffer from this suboptimality because it neglects the initial and intermediate performance

and simply finds an initialization at $x \approx 85$ from which it can reach both the optima of tasks 1 and 2.

4.5.2 Few-shot image classification

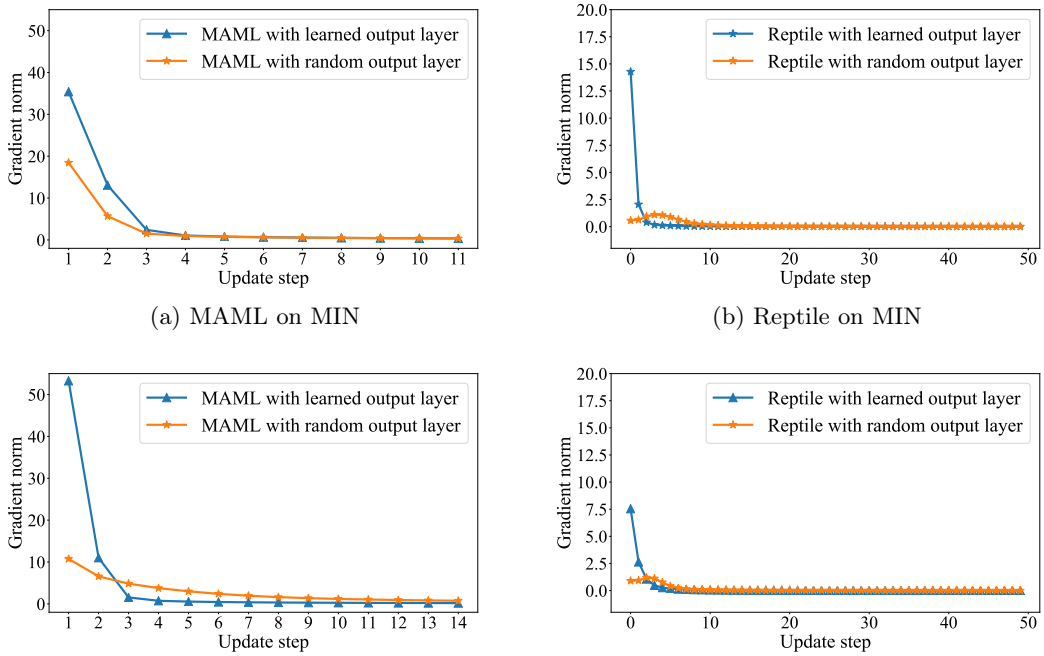
We continue our investigations by studying why MAML and Reptile can outperform finetuning in within-distribution few-shot image classification settings (see Section 4.3.2) when using a Conv-4 backbone. For these experiments, we use the N-way k-shot classification setting (see Section 4.3.2) on the miniImageNet (Vinyals et al., 2016; Ravi and Larochelle, 2017) and CUB (Wah et al., 2011) benchmarks. miniImageNet is a mini variant of the large ImageNet dataset (Deng et al., 2009) for image classification, consisting of 60 000 colored images of size 84×84. The dataset contains 100 classes and 600 examples per class. We use the same train/validation/test class splits as in Ravi and Larochelle (2017). The CUB dataset contains roughly 12 000 RGB images of birds from 200 species (classes). We use the same setting and train/validation/test class splits as in Chen et al. (2019).

Note that using real datasets entails that we move away from the abstract task definition as in the previous toy experiment, where the loss signal of the task was perfect. Instead, the loss signal is now approximated by sampling a finite set of data points for every task (for MAML and Reptile) or batch (for finetuning) and computing the performance of the methods on it.

For finetuning and MAML, we tune the hyperparameters on the meta-validation tasks using random search with a budget of 30 function evaluations for every backbone and dataset. We train MAML on 60 000 tasks in the 1-shot setting and on 40 000 tasks in the 5-shot setting, and validate its performance every 2 500 tasks. The checkpoint with the highest validation accuracy is then evaluated on 600 holdout test tasks. Similarly, finetuning is trained on 60 000 batches of data from the training split when we evaluate it in the 1-shot setting and on 40 000 batches when evaluating it in the 5-shot setting. Note that finetuning is trained on simple mini-batches of data instead of tasks consisting of a support and query set, and is later validated and tested on unseen validation and test tasks, respectively. In a similar fashion as for MAML, we validate its performance every 2 500 batches. Due to the computational expenses, for Reptile, we use the best-reported hyperparameters and training iterations on 5-way 1-shot miniImageNet as found by Nichol et al. (2018). We use Torchmeta for the implementation of the data loaders (Deleu et al., 2019). We note that a single run of MAML and finetuning finish within one day, while Reptile finished within 4 days, perhaps due to the absence of parallelism in the implementation we used.

The role of the output layer

Here, we investigate whether the fact that MAML and Reptile reuse their learned output layer when learning new tasks alter their inner-learning behaviour and give them an advantage in performance compared with finetuning, which removes the learned output layer and replaces it with a randomly initialized one when learning a new task. In short, we study the role of the output layer on the performance and inner-loop adaptation behaviour of MAML and Reptile. For this, we perform meta-training for MAML and Reptile on 5-way 1-shot miniImageNet classification, and study the effect of replacing the learned output layer initialization weights with random weights on their ability to learn new tasks. Note that even though the weight initialization of the output layer may be random, it is still trained on the support sets of unseen tasks, therefore, finetuned to the task upon which it will be evaluated. Figure 4.2 displays the effect of replacing the output layer of the meta-learned weight initialization by MAML and Reptile meta-trained on 5-way 1-shot miniImageNet, with a



(a) MAML on MIN

(b) Reptile on MIN

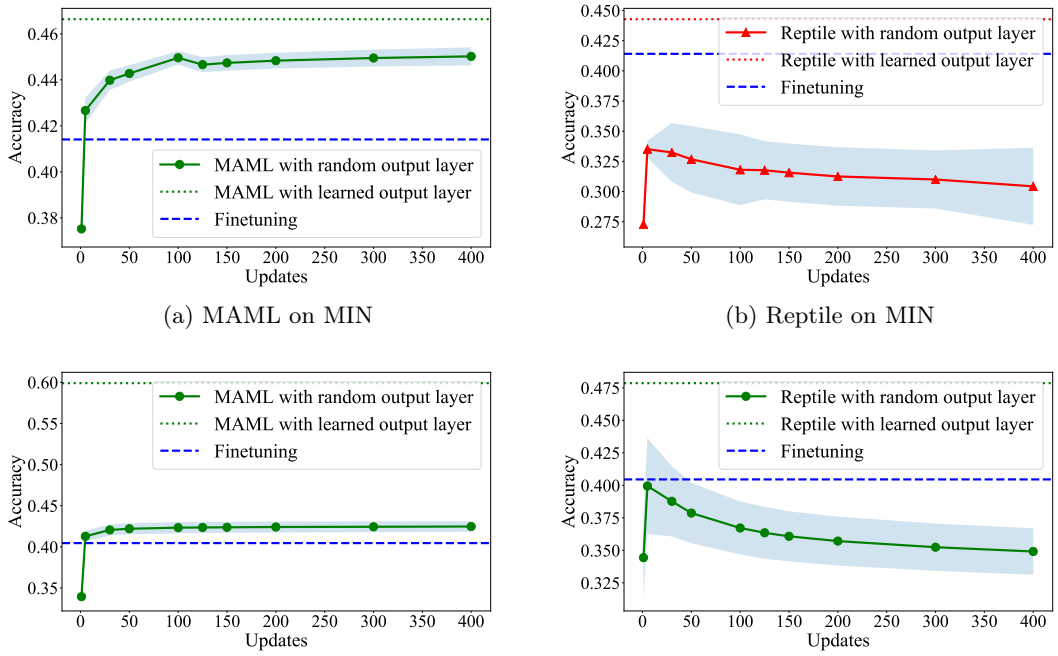
(c) MAML on CUB

(d) Reptile CUB

Figure 4.2: The difference in the average gradient norms during inner-loop adaptation between MAML (left) and Reptile (right) with a learned output layer and a randomly initialized one on 5-way 1-shot miniImageNet (MIN; top row) and CUB (bottom row). The 95% confidence intervals are within the size of the symbols. The learned output layers have a higher gradient norm at the beginning of the training phase.

randomly initialized one on the gradient norms during the inner-loop adaptation procedure. As we can see, the networks of the variants with a learned output layer receive larger gradient norms at the first few updates compared with the variants using a randomly initialized output layer, indicating that the learned output layer alters the learning behaviour of the algorithms. However, at the end of adaptation for a given task, the gradient norms are close to zero for both variants, indicating that both have converged to a local minimum. This implies that the learned initialization of the output layer has a distinct influence on the learning behaviour of new tasks. More specifically, using a learned output layer may aid in finding an initialization in the loss landscape that is sensitive to tasks and can be quickly adapted, explaining the larger gradient norms.

Next, we investigate whether reusing the learned output layers also leads to performance differences. For this, we investigate the influence of replacing the learned output layers in MAML and Reptile with randomly initialized ones when starting to learn new tasks on their learning performance for different numbers of update steps. The results are shown in Figure 4.3. As we can see, replacing the output layer with a random one leads to worse performance. Increasing the number of updates improves the performance for MAML, while the reverse is true for Reptile. In the end, the performance gap introduced by replacing the output layers with random ones is not closed, indicating that the output layers play an important role in successful inner-loop adaptation.



(a) MAML on MIN

(b) Reptile on MIN

(c) MAML on CUB

(d) Reptile on CUB

Figure 4.3: The difference in performance between MAML (left) and Reptile (right) with a learned output layer and a randomly initialized one on 5-way 1-shot miniImageNet (MIN; top row) and CUB (bottom row) for different numbers of update steps. The 95% confidence intervals are displayed as shaded regions. Learning new tasks starting with a random output layer fails to achieve the same end performance as with the learned output layer.

Specialization for robustness against overfitting

In this subsection, we investigate the influence of the level of data scarcity in the support set on the performance of MAML and Reptile. We hypothesize that both algorithms learn an initialization that is robust against overfitting when the number of examples in the support set per class (k) is small. This would imply that their performance would suffer when the number of examples in the support sets in training tasks is large due to the reduced need to become robust against overfitting, disabling the meta-learning techniques to become robust to overfitting during task-specific adaptation. We investigate this for 5-way miniImageNet image classification by varying the number of examples in the support set of meta-training tasks and measuring the performance on tasks with only one example per class (1-shot setting).

Figure 4.4 displays the results of these experiments. As we can see, there is an adverse effect of increasing the number of support examples per task on the final 1-shot performance of MAML. This shows that for MAML, it is important to match the training and test conditions so the initialization parameters can become robust against overfitting induced by data scarcity. In addition, we observe that Reptile is unstable due to its sensitivity to different hyperparameters on miniImageNet, even in the setting where $k = 1$. This is caused by the fact that Reptile is not allowed to sample mini-batches of data from the support set. Instead, we force it to use the full support set to investigate the effect of the number of support examples. When the number of examples is close to ten, which is the mini-batch size commonly used, as by the original authors (Nichol et al., 2018), there is a slight increase in performance for Reptile on miniImageNet, supporting the observation that it is sensitive to the chosen hyperparameters. On CUB, in contrast, we observe that the performance improves with the number of examples per class at training time, although the maximum number of examples investigated is 25 due to the fact that not every class has more examples than that. This illustrates that the sensitivity to hyperparameters depends on the chosen dataset.

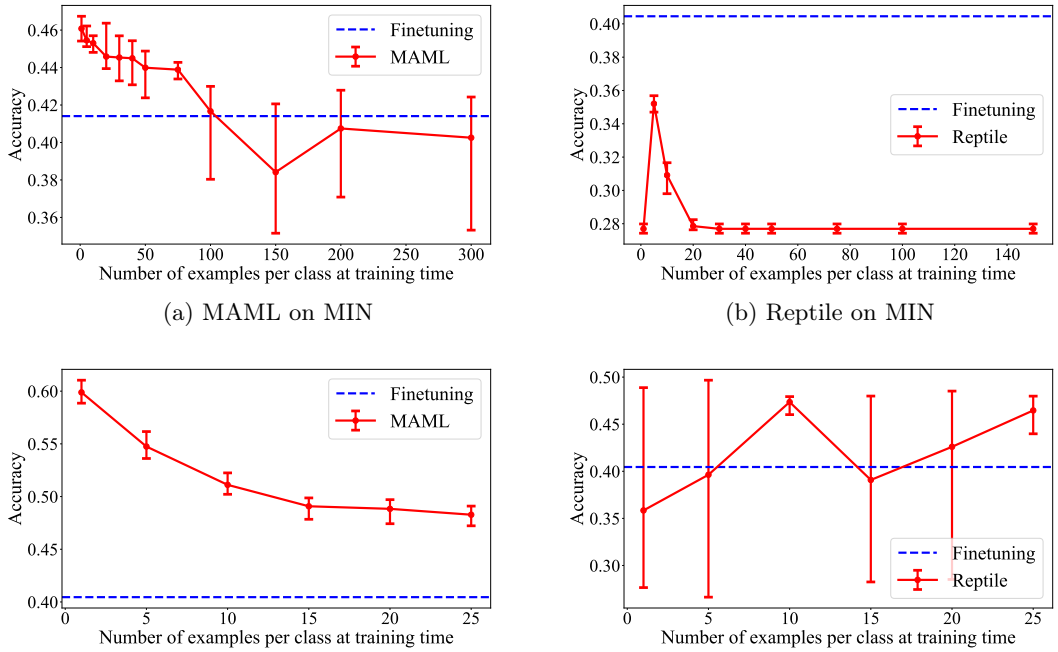
Information content in the learned initializations

Next, we investigate the relationship between the few-shot image classification performance and the discriminative power of the learned features by the three techniques for different backbones (Conv-4, ResNet10, ResNet18 (He et al., 2015)).

After deploying the three techniques on the datasets in a 5-way 1-shot manner, we measure the discriminative power of the learned initializations. Figure 4.5 visualizes this procedure for MAML and Reptile; finetuning follows a similar procedure. First, we extract the learned initialization parameters from the techniques. Second, we load these initializations into the base-learner network, freeze all hidden layers, and replace the output layer with a new one. The new output layer contains one node for every of the $|C_{test}|$ classes in the meta-test data. Third, we fine-tune this new output layer on the meta-test data in a *non-episodic* manner, which corresponds to regular supervised learning on the meta-test dataset. We use a 60/40 train/test split and evaluate the final performance on the latter. We refer to the resulting performance measure as the *joint classification accuracy*, which aims to indicate the discriminative power of the learned initialization, evaluated on data from unseen classes. Note that we use the expressions "discriminative power" and "information content" of the learned backbone synonymously.

The results of this experiment are shown in Figure 4.6. From this figure, we see that finetuning yields the best joint classification accuracy in all scenarios. From this figure, we see the following things.

- The within-distribution few-shot learning performance is better than the out-of-distribution



(a) MAML on MIN

(b) Reptile on MIN

(c) MAML on CUB

(d) Reptile on CUB

Figure 4.4: The effect of the number of training examples per class in the support set on the performance of MAML (left) and Reptile (right) on 5-way 1-shot miniImageNet (MIN; top row) and CUB (bottom row) classification. The larger the number of examples, the worse the few-shot learning performance of MAML. The error bars show the maximum and minimum performance over 5 runs with different random seeds. Note that the test tasks contain only a single example per class in the support set.

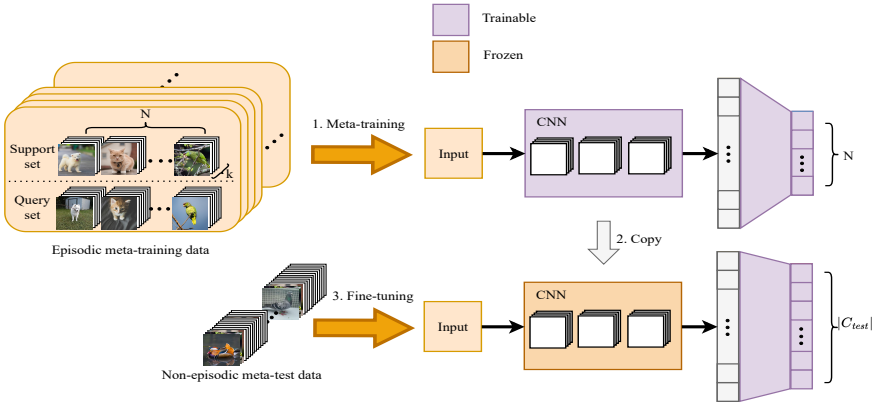


Figure 4.5: Flow chart for measuring the joint classification accuracy for meta-learning techniques. First, we train the techniques in an episodic manner on all data in the meta-train set. Second, we copy and freeze the learned initialization parameters and replace the output layer with a new one. Third, we fine-tune this new output layer on all meta-test data in a non-episodic manner. As such, the meta-test data is split into a non-episodic train and a non-episodic test set. Finally, we evaluate the learned evaluation on the hold-out test split of the meta-test data. We refer to the resulting performance measure as the joint classification accuracy. Note that finetuning follows the same procedure, with the exception that it trains non-episodically (on batches instead of tasks) on the meta-training data.

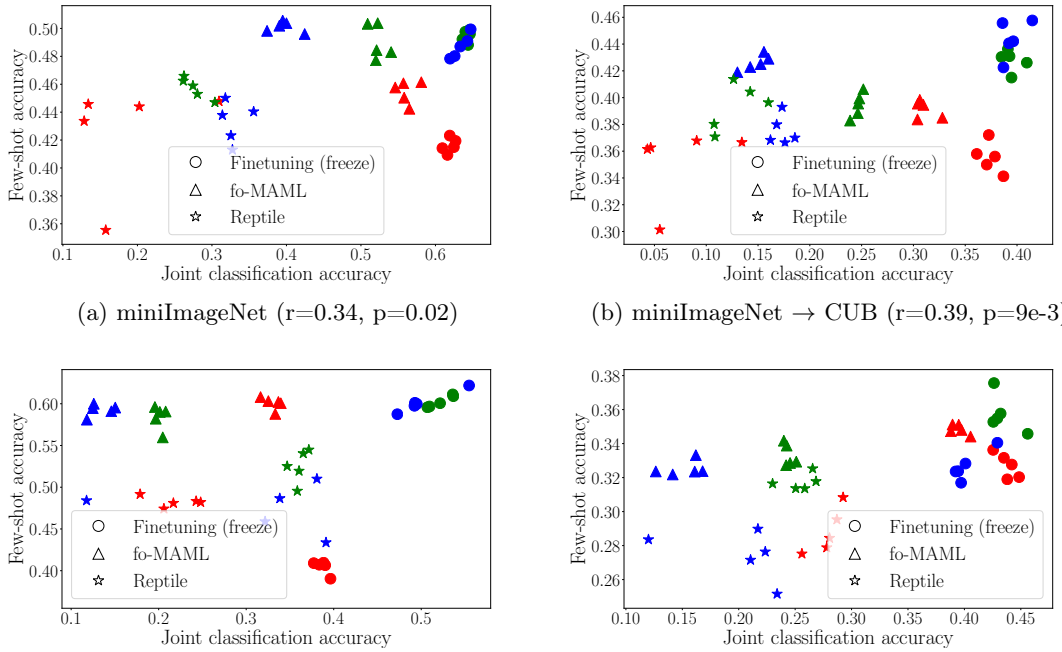
Table 4.2: Individual correlations between the joint classification accuracy and the few-shot learning performance. The Pearson correlation coefficients are indicated as r and corresponding p-values as p. We note that the results for each of the three few-shot learning techniques are produced with three different backbone networks. As such, correlations should be interpreted with utmost care. Significant correlations (using a threshold of $\alpha = 0.005$) are displayed in bold. “MIN": miniImageNet.

	MIN	MIN → CUB	CUB	CUB → MIN
Finetuning	**r=0.82, p=2e-4**	**r=0.71, p=3e-3**	**r=0.96, p=7e-9**	r=0.28, p=0.31
MAML	**r=-0.77, p=8e-4**	**r=-0.85, p=6e-5**	r=0.36, p=0.18	**r=0.90, p=4e-6**
Reptile	r=0.27, p=0.3	r=0.50, p=0.06	r=0.3, p=0.28	r=0.31, p=0.27

performance for all techniques

- MAML achieves the best few-shot learning performance when using a shallow backbone (conv-4)
- When the backbone becomes deeper, the features learned by MAML become less discriminative
- Finetuning learns the most discriminative set of features for direct joint classification on a large set of classes

However, we note that the joint classification performance either weakly correlates or does not correlate with the few-shot learning performance across the different techniques. We note that these correlation patterns may be affected by the fact that we used the best-reported hyperparameters for Reptile for the Conv-4 backbone, while we also use ResNet-10 and ResNet-18 backbones (He et al., 2015) in different settings. For finetuning, however, we do observe an improvement in few-shot learning performance as the backbone becomes deeper.



(a) miniImageNet (r=0.34, p=0.02)

(b) miniImageNet → CUB (r=0.39, p=9e-3)

(c) CUB (r=0.07, p=0.63)

(d) CUB → miniImageNet (r=0.53, p=2e-4)

Figure 4.6: The joint classification accuracy (x-axes) plotted against the 5-way 1-shot performance (y-axis) on all test classes. For every technique, there are 15 results plotted, corresponding to 3 backbones (Conv-4=red, ResNet-10=green, ResNet-18=blue) and 5 runs per setting. The Pearson correlation coefficients (r) and p-values are displayed in the subcaptions. The general correlations between the few-shot learning performance and joint classification accuracy range from weak to mild.

Next, we investigate whether there are statistically significant relationships per technique between the joint classification accuracy and the few-shot performance. Table 4.2 displays the Pearson correlation and corresponding p-values for individual techniques for the experiment in Section 4.5.2. As we can see, there are strong and significant ($\alpha = 0.005$) correlations between the joint classification accuracy and the few-shot learning performance of finetuning in three settings. For MAML, there are strong negative correlations on miniImageNet and miniImageNet → CUB, indicating that a lower joint classification accuracy is often associated with better few-shot learning performance. For Reptile, the correlations are non-significant and mild to weak.

4.6 Conclusion

In this chapter, we investigated 1) why MAML and Reptile can outperform finetuning in *within-distribution settings*, and 2) why finetuning can outperform gradient-based meta-learning techniques such as MAML and Reptile when the test data distribution diverges from the training data distribution.

We have shown how the optimization objectives of the three techniques can be interpreted as maximizing the direct performance, post-adaptation performance, and a combination of the two,

respectively. That is, finetuning aims to maximize the direct performance whereas MAML aims to maximize the performance *after* a few adaptation steps, making it a look-ahead objective. Reptile is a combination of the two as it focuses on both the initial performance as well as the performance after every update step on a given task. As a result, finetuning will favour an initialization that jointly minimizes the loss function, whereas MAML may settle for an inferior initialization that yields more promising results after a few gradient update steps. Reptile picks something in between these two extremes. Our synthetic example in Section 4.5.1 shows that these interpretations of the learning objectives allow us to understand the chosen initialization parameters.

Our empirical results show that these different objectives translate into different learned initializations. We have shown that MAML and Reptile specialize for adaptation in low-data regimes of the training tasks distribution, which explains why these techniques can outperform finetuning as observed by Chen et al. (2019); Finn et al. (2017); Nichol et al. (2018), answering our first research question. Both the weights of the output layer and the data scarcity in training tasks play an important role in facilitating this specialization, allowing them to gain an advantage over finetuning.

Moreover, we have found that finetuning learns a broad and diverse set of features that allows it to discriminate between many different classes. MAML and Reptile, in contrast, optimize a look-ahead objective and settle for a less diverse and broad feature space as long as it facilitates robust adaptation in low-data regimes of the *same* data distribution (as that is used to optimize the look-ahead objective). This can explain findings by Chen et al. (2019), who show that finetuning can yield superior few-shot learning performance in out-of-distribution settings. However, we do not observe a general correlation between the feature diversity and the few-shot learning performance across finetuning, Reptile, and MAML.

Another result is that MAML yields the best few-shot learning performance when using the Conv-4 backbone in all settings. Interestingly, the features learned by MAML become less discriminative as the depth of the backbone increases. This may indicate an over-specialization, and it may be interesting to see whether adding a penalty for narrow features may prevent this and increase the few-shot learning performance with deeper backbones and in out-of-distribution settings, which has been observed to be problematic by Rusu et al. (2019) and Chen et al. (2019) respectively. As this is beyond the scope of our research questions, we leave this for future work. Another fruitful direction for future work would be to quantify the distance or similarity between different tasks and to investigate the behaviour of meta-learning algorithms as a function of this quantitative measure. An additional benefit of such a measure of task similarity would be that it could allow us to detect when a new task is within-distribution or out-of-distribution, which could inform the choice of which algorithm to use.

In summary, our results suggest that the answer to our second research question is that MAML and Reptile may fail to quickly learn out-of-distribution tasks due to their over-specialization to the training data distribution caused by their look-ahead objective, whereas finetuning learns broad features that allow it to learn new out-of-distribution concepts. This is supported by the fact that in almost all scenarios, there are statistically significant relationships between the broadness of the learned features and the few-shot learning ability for finetuning.

In this chapter, we explored an alternative approach to deep meta-learning for increasing the learning efficiency of deep neural networks, namely pretraining and finetuning. More specifically, we compared the learning behaviors of finetuning, MAML, and Reptile. In the following chapters, we return to the realm of deep meta-learning to investigate LSTMs for few-shot learning (Chapter 5), and to

investigate whether the integration of classical machine learning knowledge can improve the few-shot learning behavior of deep neural networks (Chapter 6).

Chapter 5

Are LSTMs Good Few-Shot Learners?

Chapter overview

Deep learning requires large amounts of data to learn new tasks well, limiting its applicability to domains where such data is available. Meta-learning overcomes this limitation by learning how to learn. In 2001, Hochreiter et al. showed that an LSTM trained with backpropagation across different tasks is capable of meta-learning. Despite promising results of this approach on small problems, and more recently, also on reinforcement learning problems, the approach has received little attention in the supervised few-shot learning setting. We revisit[1] this approach and test it on modern few-shot learning benchmarks. We find that LSTM, surprisingly, outperform the popular meta-learning technique MAML on a simple few-shot sine wave regression benchmark, but that LSTM, expectedly, fall short on more complex few-shot image classification benchmarks. We identify two potential causes and propose a new method called *Outer Product LSTM (OP-LSTM)* that resolves these issues and displays substantial performance gains over the plain LSTM. Compared to popular meta-learning baselines, OP-LSTM yields competitive performance on within-domain few-shot image classification, and performs better in cross-domain settings by 0.5% to 1.9% in accuracy score. While these results alone do not set a new state-of-the-art, the advances of OP-LSTM are orthogonal to other advances in the field of meta-learning, yield new insights in how LSTM work in image classification, allowing for a whole range of new research directions. For reproducibility purposes, we publish all our research code publicly.

5.1 Introduction

In this chapter, we investigate a classical approach to meta-learning with neural networks that was pioneered by Hochreiter et al. (2001) and Younger et al. (2001). More specifically, they have shown that LSTMs trained with gradient descent are capable of meta-learning. At the inner level—when presented with a new task—the LSTM ingests training examples with corresponding ground-truth

[1]This chapter is based on the following research article: *Huisman, M., Moerland, T. M., Plaat, A., & van Rijn, J. N. (2023). Are LSTMs good few-shot learners? Machine Learning, 112, 4635–4662. Springer.*

outputs and conditions its predictions for new inputs on the resulting hidden state (the general idea for using recurrent neural networks for meta-learning has been visualized in Figure 5.1). The idea is that the training examples that are fed into the LSTM can be remembered or stored by the LSTM in its internal states, allowing predictions for new unseen inputs to be based on the training examples. This way, the LSTM can implement a learning algorithm in the recurrent dynamics, whilst the weights of the LSTM are kept frozen. During meta-training, the weights of the LSTM are only adjusted at the outer level (across tasks) by backpropagation, which corresponds to updating the inner-level learning program. By exposing the LSTM to different tasks which it cannot solve without learning, the LSTM is stimulated to learn tasks by ingesting the training examples which it is fed. The initial experiments of Hochreiter et al. (2001) and Younger et al. (2001) have shown promising results on simple and low-dimensional toy problems. Meta-learning with LSTMs has also been successfully extended to reinforcement learning settings (Duan et al., 2016; Wang et al., 2016), and demonstrates promising learning speed on new tasks.

To the best of our knowledge, the LSTM approach has, in contrast, not been studied on more complex and modern supervised few-shot learning benchmarks by the research community, which has already shifted its attention to more developing new and more complex methods (Finn et al., 2017; Snell et al., 2017; Flennerhag et al., 2020; Park and Oliva, 2019b). In our work, we revisit the idea of meta-learning with LSTMs and study the ability of the learning programs embedded in the weights of the LSTM to perform few-shot learning on modern benchmarks. We find that an LSTM outperforms the popular meta-learning technique MAML (Finn et al., 2017) on a simple few-shot sine wave regression benchmark, but that it falls short on more complex few-shot image classification benchmarks.

By studying the LSTM architecture in the context of meta-learning, we identify two potential causes for this underperformance, namely 1) the fact that it is not invariant to permutations of the training data and 2) that the input representation and learning procedures are intertwined. We propose a general solution to the first problem and propose a new meta-learning technique, *Outer Product LSTM (OP-LSTM)*, where we solve the second issue by learning the weight update rule for a base-learner network using an LSTM, in addition to good initialization parameters for the base-learner. This approach is similar to that of Ravi and Larochelle (2017), but differs in how the weights are updated with the LSTM and that in our approach, the LSTM does not use hand-crafted gradients as inputs in order to produce weight updates. Our experiments demonstrate that OP-LSTM yields substantial performance gains over the plain LSTM.

In this chapter, we investigate the following matters.

- We study the ability of a plain LSTM to perform few-shot learning on modern few-shot learning benchmarks and show that it yields surprisingly good performance on simple regression problems (outperforming MAML (Finn et al., 2017)), but is outperformed on more complex classification problems.
- We identify two problems with the plain LSTM for meta-learning, namely 1) the fact that it is not invariant to permutations of the training data and 2) that the input representation and learning procedures are intertwined, and propose solutions to overcome them by 1) an average pooling strategy and 2) decoupling the input representation from the learning procedure.
- We propose a novel LSTM architecture called *Outer Product LSTM (OP-LSTM)* that overcomes the limitations of the classical LSTM architecture and yields substantial performance gains on few-shot learning benchmarks.

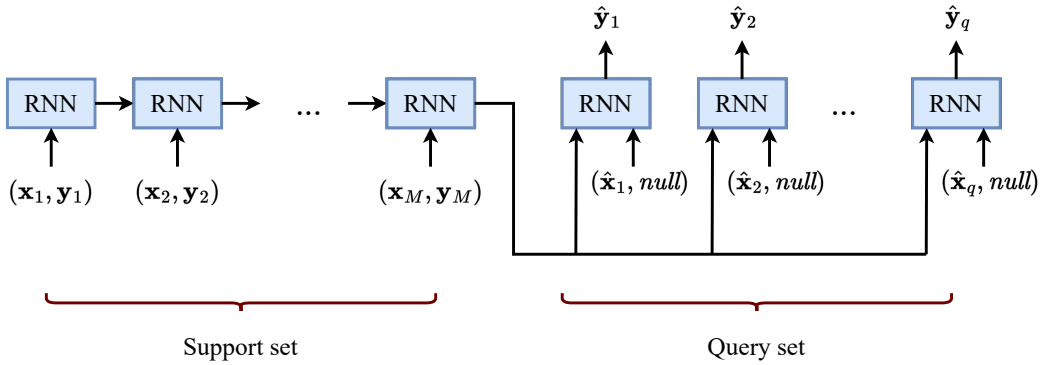


Figure 5.1: The use of a recurrent neural network for few-shot learning. The support set $D_{\mathcal{T}_j}^{tr} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_M, \mathbf{y}_M)\}$ is fed as a sequence into the RNN. The predictions $\hat{\mathbf{y}}_j$ for new query points $\hat{x}_j$ are conditioned on the resulting state. We note that feeding the tuples $(\mathbf{x}_i, \mathbf{y}_i)$ does not lead to the RNN directly outputting the presented labels (drastic overfitting) as the goal is to make predictions for query inputs, for which the ground-truth outputs are unknown. Alternatively, the support set could be fed into the RNN in a temporally offset manner (e.g., feed support tuples $(\mathbf{x}_i, \mathbf{y}_{i-1})$ into the RNN) as in Santoro et al. (2016) or in different ways (for example feed the error instead of the ground-truth target) (Hochreiter et al., 2001).

- We discuss that OP-LSTM can approximate MAML (Finn et al., 2017) as well as Prototypical network (Snell et al., 2017) as it can learn to perform the same weight matrix updates. Since OP-LSTM does not update the biases, it can only approximate these two methods.

Compared to popular meta-learning baselines, including MAML (Finn et al., 2017), Prototypical network (Snell et al., 2017), and Warp-MAML (Flennerhag et al., 2020), OP-LSTM yields competitive performance on within-domain few-shot image classification, and outperforms them in cross-domain settings by 0.5% to 1.9% in raw accuracy score. While these results alone do not set a new state-of-the-art, the advances of OP-LSTM are orthogonal to other advances in the field of meta-learning, allowing for a whole range of new research directions, such as using OP-LSTM to update the weights in gradient-based meta-learning techniques (Flennerhag et al., 2020; Park and Oliva, 2019b; Lee and Choi, 2018) rather than regular gradient descent. For reproducibility and verifyability purposes, we make all our research code publicly available.[2]

5.2 Related work

Earlier work with LSTMs Meta-learning with recurrent neural networks was originally proposed by Hochreiter et al. (2001) and Younger et al. (2001). In their pioneering work, Hochreiter et al. (2001) also investigated other recurrent neural network architectures for the task of meta-learning, but it was observed that Elman networks and vanilla recurrent neural networks failed to meta-learn simple Boolean functions. Only the LSTM was found to be successful at learning simple functions. For this reason, we solely focus on LSTM in our work.

The idea of meta-learning with an LSTM at the data level has also been investigated and shown to achieve promising results in the context of reinforcement learning (Duan et al., 2016; Wang et al.,

[2]See: `https://github.com/mikehuisman/lstm-fewshotlearning-oplstm`

2016; Alver and Precup, 2021). In the supervised meta-learning community, however, the idea of meta-learning with an LSTM at the data level (Hochreiter et al., 2001; Younger et al., 2001) has not gained much attention. A possible explanation for this is that Santoro et al. (2016) compared their proposed memory-augmented neural network (MANN) to an LSTM and found that the latter was outperformed on few-shot Omniglot (Lake et al., 2015) classification. However, it was not reported how the hyperparameters of the LSTM were tuned and whether it was a single-layer LSTM or a multi-layer LSTM. In addition, the LSTM was fed the input data as a sequence which is not permutation invariant, which can hinder its performance. We propose a permutation-invariant method of feeding training examples into recurrent neural networks and perform a detailed study of the performance of LSTM on few-shot learning benchmarks.

In concurrent work, Kirsch et al. (2022) investigates the ability of transformer architectures to implement learning algorithms, a baseline with a similar name as our proposed method was proposed ("Outer product LSTM"). We emphasize, however, that their method is different from ours (OP-LSTM) as it is a model-based approach that ingests the entire training set and query input into a slightly modified LSTM architecture (with an outer product update and inner product read-out) to make predictions, whereas in our OP-LSTM, the LSTM acts on a meta-level to update the weights of a base-learner network. Hollmann et al. (2023) investigates the ability of a transformer architecture to implement a general learning algorithm to quickly solve tabular classification problems. This work is similar to ours in the sense that it also investigates meta-learning general purpose learning algorithms, but differs in that they are interested in tabular problems and transformers, whereas we focus on LSTMs for few-shot image classification. Moreover, we do not generate synthetic data for meta-training on the main problem of interest (few-shot image classification).

In concurrent works done by Kirsch et al. (2022) and Chan et al. (2022), the ability of the classical LSTM architecture to implement a learning algorithm was also investigated. They observed that it was unable to embed a learning algorithm into its recurrent dynamics on image classification tasks. However, the focus was not on few-shot learning, and no potential explanation for this phenomenon was given. In our work, we investigate the LSTM's ability to learn a learning algorithm in settings where only one or five examples are present per class, dive into the inner working mechanics to formulate two hypotheses as to why the LSTM architecture is incapable of learning a good learning algorithm, and as a result, propose OP-LSTM which overcomes the limitations and performs significantly better than the classical LSTM architecture.

Different LSTM architectures for meta-learning Santoro et al. (2016) used an LSTM as a read/writing mechanism to an external memory in their MANN technique. Kirsch and Schmidhuber (2021) proposed to replace every weight in a neural network with a recurrent neural network that communicates through forward and backward messages. The system was shown able to learn backpropagation and can be used to improve upon it. Our proposed method OP-LSTM can also learn to implement backpropagation (see Section 5.6). Other works (Ravi and Larochelle, 2017; Andrychowicz et al., 2016) have also used an LSTM for meta-learning the weight update procedure. Instead of feeding the training examples into the LSTM, as done by the plain LSTM (Hochreiter et al., 2001; Younger et al., 2001), the LSTM was fed gradients so that it could propose weight updates for a separate base-learner network. Our proposed method OP-LSTM is similar to these two approaches that meta-learn the weight update rules as we use an LSTM to update the weights (2D hidden states) of a base-learner network. Note that this strategy thus also deviates from the plain LSTM approach, which is fed raw input data. In our approach, the LSTM acts on predictions and ground-truth targets or messages. In addition, we use a coordinate-wise architecture where the same LSTM is applied to different nodes in the network. A difference with other learning-to-optimize

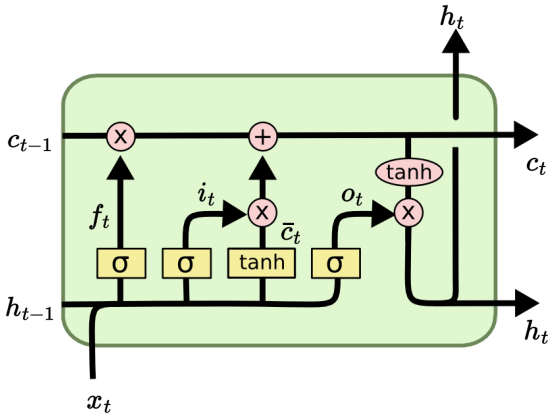


Figure 5.2: The architecture of an LSTM cell. The LSTM maintains an inner cell state $\mathbf{c}_t$ and hidden state $\mathbf{h}_t$ over time that are updated with new incoming data $\mathbf{x}_t$. The forget $\mathbf{f}_t$, input $\mathbf{i}_t$, output $\mathbf{o}_t$, and cell $\bar{\mathbf{c}}_t$ gates regulate how these states are updated. Image adapted from Olah (2015).

works (Ravi and Larochelle, 2017; Andrychowicz et al., 2016) is that we do not feed gradients into the LSTM and that we update the weights (2D hidden states) through outer product update rules.

5.3 Meta-learning with LSTM

In this section, we briefly review the LSTM architecture (Hochreiter and Schmidhuber, 1997), explain the idea of meta-learning with an LSTM through backpropagation as proposed by Hochreiter et al. (2001) and Younger et al. (2001), discuss two problems with this approach in the context of meta-learning and propose solutions to solve them.

Additionally, we propose solutions to this problem. We prove that a single-layer RNN followed by a linear layer is incapable of embedding a classification learning algorithm in its recurrent dynamics and show by example that an LSTM adding a single linear layer is sufficient to achieve this type of learning behavior in a simple setting.

5.3.1 LSTM architecture

LSTM (Hochreiter and Schmidhuber, 1997) is a recurrent neural network architecture suitable for processing sequences of data. The architecture of an LSTM cell is displayed in Figure 5.2. It maintains an internal state and uses four gates to regulate the information flow within the network

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f), \tag{5.1}$$
$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i), \tag{5.2}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o), \tag{5.3}$$
$$\bar{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c). \tag{5.4}$$

Here, $\theta = \{\mathbf{W}_f, \mathbf{W}_c, \mathbf{W}_i, \mathbf{W}_o, \mathbf{b}_f, \mathbf{b}_c, \mathbf{b}_i, \mathbf{b}_o\}$ are the parameters of the LSTM, $[\mathbf{a}, \mathbf{b}]$ represents the concatenation of $\mathbf{a}$ and $\mathbf{b}$, σ is the sigmoid function (applied element-wise) and $\mathbf{f}_t, \mathbf{i}_t, \mathbf{o}_t, \bar{\mathbf{c}}_t \in \mathbb{R}^{d_h}$

are the forget, input, output, and cell gates, respectively. These gates regulate the information flow within the network to produce the next cell and hidden states

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \bar{\mathbf{c}}_t, \tag{5.5}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \tag{5.6}$$

The hidden state and cell state are obtained by applying LSTM g_θ to inputs $(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1)$, i.e.,

$$[\mathbf{h}_t, \mathbf{c}_t] = g_\theta(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1) \tag{5.7}$$
$$= m_\theta(\mathbf{x}_t; \mathbf{h}_{t-1}, \mathbf{c}_{t-1}). \tag{5.8}$$

5.3.2 Meta-learning with LSTM

Hochreiter et al. (2001) and Younger et al. (2001) show that the LSTM can perform learning purely through unrolling its hidden state over time with fixed weights. When presented with a new task $\mathcal{T}_j$—denoting the concatenation of an input and its target as $\mathbf{x}'_t = (\mathbf{x}_t, \mathbf{y}_t)$—the support set $D^{tr}_{\mathcal{T}_j} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_M, \mathbf{y}_M)\} = \{\mathbf{x}'_1, \mathbf{x}'_2, \ldots, \mathbf{x}'_M\}$, is fed as a sequence, e.g., $(\mathbf{x}_1, \mathit{null}), (\mathbf{x}_2, \mathbf{y}_1), \ldots, (\mathbf{x}_M, \mathbf{y}_{M-1})$, into the LSTM to produce a hidden state $\mathbf{h}_M(D^{tr}_{\mathcal{T}_j})$. Predictions for unseen inputs (queries) $\hat{\mathbf{x}}$ are then conditioned on the hidden state $\mathbf{h}_M(D^{tr}_{\mathcal{T}_j})$ and cell state $\mathbf{c}_M(D^{tr}_{\mathcal{T}_j})$, where we have made it explicit that $\mathbf{h}_M$ and $\mathbf{c}_M$ are functions of the support data. More specifically, the hidden state of the query input $\hat{\mathbf{x}} = [\mathbf{x}, \mathbf{y}_M]$ is computed as $[\hat{\mathbf{h}}, \hat{\mathbf{c}}] = m_\theta(\hat{\mathbf{x}}; \mathbf{h}_M(D^{tr}_{\mathcal{T}_j}), \mathbf{c}_M(D^{tr}_{\mathcal{T}_j}))$, and this hidden state is used either directly for prediction or can be fed into a classifier function (which also uses fixed weights). Since the weights of the LSTM are fixed when presented with a new task, the learning takes place in the recurrent dynamics, and the hidden state $\mathbf{h}_M(D^{tr}_{\mathcal{T}_j})$ is responsible for guiding predictions on unseen inputs $\hat{\mathbf{x}}$. Note that there are different ways to feed the support data into the LSTM, as one can also use additional data such as the error on the previous input or feed the current input together with its target tuples $(\mathbf{x}_t, \mathbf{y}_t)$ (as done in Figure 5.1 and our implementation). We use the latter strategy in our experiments as we found it to be most effective.

This recurrent learning algorithm can be obtained by performing meta-training on various tasks which require the LSTM to perform learning through its recurrent dynamics. Given a task, we feed the training data into the LSTM, and then feed in query inputs to make predictions. The loss on these query predictions can be backpropagated through the LSTM to update the weights across different tasks. Note, however, that during the unrolling of the LSTM over the training data, the weights of the LSTM are held fixed. The weights are thus only updated across different tasks (not during adaptation to individual tasks) to improve the recurrent learning algorithm. By adjusting the weights of the LSTM using backpropagation across different tasks, we are essentially changing the learning program of the LSTM and hence performing meta-learning.

5.3.3 Problems with the classical LSTM architecture

The classical LSTM architecture suffers from two issues that may limit its ability to implement recurrent learning algorithms.

Non-temporal training data LSTMs work with sequences of data. When using an LSTM in the meta-learning context, the recurrent dynamics should implement a learning algorithm and process the support dataset. This support dataset $D^{tr}_{\mathcal{T}_j} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_M, \mathbf{y}_M)\} = \{\mathbf{x}'_1, \mathbf{x}'_2, \ldots, \mathbf{x}'_M\}$,

however, is a *set* rather than a sequence. This means that we would want the hidden embedding after processing the support data to be invariant with respect to the order in which the examples are fed into the LSTM. Put more precisely, given any two permutations of the M training examples $\pi = (\pi_1, \pi_2, \ldots, \pi_M)$ and $\pi' = (\pi'_1, \pi'_2, \ldots, \pi'_M)$, we want to enforce

$$g_\theta(\mathbf{x}'_{\pi_1}, \mathbf{x}'_{\pi_2}, \ldots, \mathbf{x}'_{\pi_M}) = g_\theta(\mathbf{x}'_{\pi'_1}, \mathbf{x}'_{\pi'_2}, \ldots, \mathbf{x}'_{\pi'_M}), \tag{5.9}$$

where $\mathbf{x}'_{\pi_i}$ is the i-th input (possibly containing target or error information) under permutation π and $\mathbf{x}'_{\pi'_i}$ the input under permutation π'.

Intertwinement of embedding and learning In the LSTM approach proposed by Hochreiter et al. (2001) and Younger et al. (2001), the recurrent dynamics implement a learning algorithm. At the same time, however, the hidden state also serves as an input embedding. Thus, in this approach, the input embedding and learning procedures are intertwined. This may be problematic because a learning procedure may be highly complex and nonlinear, whilst the optimal input embedding may be simple and linear. For example, suppose that we feed convolutional features into a plain LSTM. Normally, we often compute predictions using a linear output layer. Thus, a simple single-layer LSTM may be the best in terms of input representation. However, the learning ability of a single-layer LSTM may be too limited, leading to bad performance. In other words, stacking multiple LSTM layers may be beneficial for finding a better learning algorithm, but the resulting input embedding may be too complex, which can lead to overfitting. On the other hand, a good but simple input embedding may overly restrict the search space of learning algorithms, resulting in a bad learning algorithm.

An LSTM with sufficiently large hidden dimensionality may be able to separate the learning from the input representation by using the first N dimensions of the hidden representations to perform learning and to preserve important information for the next time step, and using the remaining dimensions to represent the input. However, this poses a challenging optimization problem due to the risk of overfitting and the large number of parameters that would be needed.

5.3.4 Towards an improved architecture

These potential issues of the classical LSTM architecture inspire us to develop an architecture that is better suited for meta-learning.

Non-temporal data → average pooling In order to enforce invariance of the hidden state and cell state with respect to the order of the support data, we can *pool* the individual embeddings. That is, given an initial state of the LSTM $\mathbf{s}_t = [\mathbf{h}_t, \mathbf{c}_t]$, we update the state by processing the support data as a batch and by average pooling, i.e.,

$$\mathbf{s}_{t+1} = [\mathbf{h}_{t+1}, \mathbf{c}_{t+1}] = \frac{1}{M} \sum_{i=1}^{M} m_\theta(\mathbf{x}'_i; \mathbf{h}_t, \mathbf{c}_t). \tag{5.10}$$

Note that one time step now corresponds to processing the entire support dataset once, since $\mathbf{s}_{t+1}$ is a function thereof. Our proposed batch processing for a single time step (during which we ingest the support data) has been visualized in Figure 5.3.

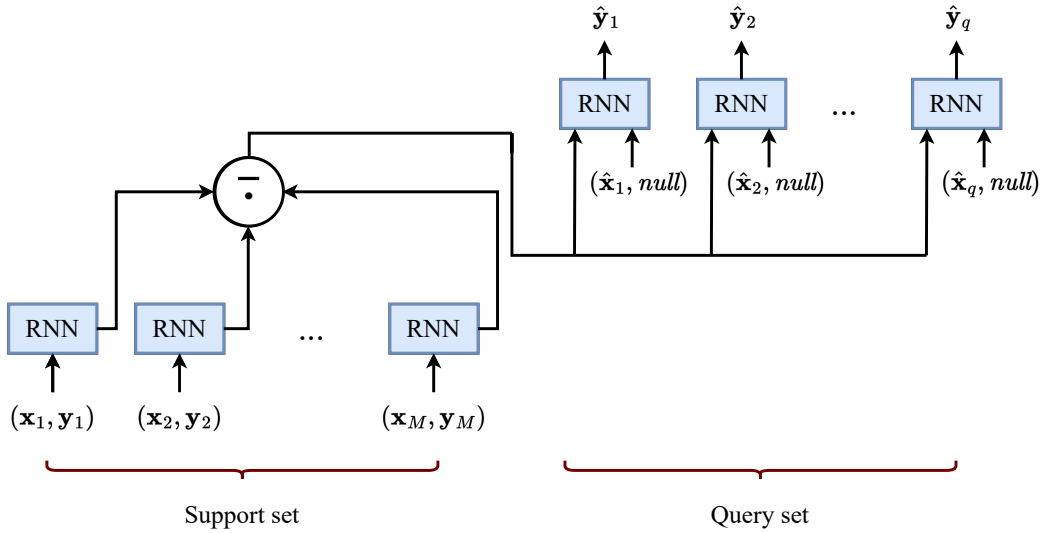


Figure 5.3: Our proposed batch processing of the support data, resulting in a state that is permutation invariant. Every support example $(\mathbf{x}_i, \mathbf{y}_i)$ is processed in parallel, and the resulting hidden states are aggregated with mean-pooling (denoted by the symbol $\bar{\cdot}$). The predictions $\hat{\mathbf{y}}_j$ for new query points $\hat{x}_j$ are conditioned on the resulting permutation-invariant state. Note that the support data is only fed once into the RNN (a single time step t), although it is possible to make multiple passes over the data, by feeding the mean-pooled state into the RNN at the next time step.

Intertwinement of embedding and learning The problems associated with the intertwinement of the embedding and learning procedures can be solved by decoupling them. In this way, we create two separate procedures: 1) the embedding procedure, and 2) the learning procedure. The embedding procedure can be implemented by a base-learner neural network, and the learning procedure by a meta-network that updates the weights of the base-learner network.

In the plain LSTM approach, where the learning procedure is intertwined with the input representation mechanism, predictions would be conditioned on the hidden state $\mathbf{h}(\mathbf{x}, \mathbf{h}(D^{tr}_{\mathcal{T}_j}), \mathbf{c}(D^{tr}_{\mathcal{T}_j}))$. Instead, we choose to use the inner product between the hidden state (acting as weight vector) and the embedding of current input $\mathbf{a}^{(L)}(\mathbf{x})$, i.e.,

$$\hat{y}(\mathbf{x}) = a^{(L)}(\mathbf{x}) = \underbrace{\mathbf{h}^{(L)}(D^{tr}_{\mathcal{T}_j})^T}_{\text{learning}} \underbrace{\mathbf{a}^{(L-1)}(\mathbf{x})}_{\text{embedding}}, \tag{5.11}$$

where $\mathbf{a}^{(L-1)}(\mathbf{x})$ is the representation of input $\mathbf{x}$ in layer $L-1$ of some base-learner network (consisting of L layers), whose weights are updated by a meta-network. We use the inner product to force interactions between the learning and embedding components, so that the predictions can not rely on either of the two separately. Note that by computing predictions in this way, we effectively decouple the learning algorithm implemented by hidden state dynamics from the input representation. A problem with this approach is that the output is a single scalar. In order to obtain an arbitrary output dimension $d_{out} > 1$, we should multiply the input representation $\mathbf{a}^{(L-1)}(\mathbf{x})$ with a matrix $\mathbf{H} \in \mathbb{R}^{d_{out} \times d_{in}}$, i.e., $\hat{\mathbf{y}}(\mathbf{x}) = \mathbf{H}^{(L)}\mathbf{a}^{(L-1)}(\mathbf{x})$. In order to obtain $\mathbf{H}^{(L)}$, one could use a separate LSTM with a hidden dimension of d_{in} per output dimension, but the number of required LSTMs would

grow linearly with the output dimensionality. Instead, we use the outer product, which requires only one hidden vector of size d_{in} that can be outer-multiplied with a vector of size d_{out}. We detail the computation of 2D weight matrices $\mathbf{H}$ (hidden states) with the outer product rule in the next section.

Note that the 2D hidden state $\mathbf{H}$ can be seen as a weight matrix of a regular feed-forward neural network, which allows us to generalize this approach to networks with an arbitrary number of layers, where we have a 2D hidden state $\mathbf{H}^{(\ell)}$ for every layer $\ell \in \{1, 2, \ldots, L\}$ in a network with L layers. Our approach can then be seen as meta-learning an outer product weight update rule for the base-learner network such that it can quickly adapt to new tasks.

5.4 Outer product LSTM (OP-LSTM)

Here, we propose a new technique, called *Outer Product LSTM (OP-LSTM)*, based on the problems of the classical LSTM architecture for meta-learning and our suggested solutions. We begin by discussing the architecture, then cover the learning objective and algorithm, and end by studying the relationship between OP-LSTM and other methods.

5.4.1 The architecture

Since we can view the 2D hidden states $\mathbf{H}^{(\ell)}$ in OP-LSTM as weight matrices that act on the input, the OP-LSTM can be interpreted as a regular fully-connected neural network. The output of the OP-LSTM for a given input $\mathbf{x}$ is given by

$$f_\theta(\mathbf{x}, D^{tr}_{\mathcal{T}_j}, T) = \sigma^{(L)}(\mathbf{H}^{(L)}_T \mathbf{a}^{(L-1)}_T(\mathbf{x}) + \mathbf{b}^{(L)}), \tag{5.12}$$

where $D^{tr}_{\mathcal{T}_j}$ is the support dataset of the task, L the number of layers of the base-learner network, and T the number of time steps that the network unrolls (trains) over the entire support set. Here, $\sigma^{(L)}$ is the activation function used in layer L, $\mathbf{b}^{(L)}$ the bias vector in the output layer, and $\mathbf{a}^{(L-1)}_T(\mathbf{x})$ the input to layer L after making T passes over the support set and having received the query input.

Put more precisely, the activation in layer ℓ at time step t, as a function of an input $\mathbf{x}$, is denoted $\mathbf{a}^{(\ell)}_t(\mathbf{x})$ and defined as follows

$$\mathbf{a}^{(\ell)}_t(\mathbf{x}) = \begin{cases} \mathbf{x} & \text{if } \ell = 0 \text{ (input layer)}, \\ \sigma^{(\ell)}(\mathbf{H}^{(\ell)}_t \mathbf{a}^{(\ell-1)}_t(\mathbf{x}) + \mathbf{b}^{(\ell)}) & \text{otherwise.} \end{cases} \tag{5.13}$$

Note that this defines the forward dynamics of the architecture. Here, the $\mathbf{H}^{\ell}_t \in \mathbb{R}^{d^{(\ell)}_{out} \times d^{(\ell)}_{in}}$ is the 2D hidden state that is updated by pooling over the normalized 2D *outer product* hidden states $\mathbf{h}^{(\ell)}_{t+1}(\mathbf{x}'_i)\mathbf{a}^{(\ell-1)}_t(\mathbf{x}'_i)^T$ associated with individual training examples $\mathbf{x}'_i = (\mathbf{x}_i, \mathbf{y}_i)$, i.e.,

$$\mathbf{H}^{(\ell)}_{t+1} = \mathbf{H}^{(\ell)}_t + \frac{\gamma}{M} \sum_{i=1}^{M} \frac{\mathbf{h}^{(\ell)}_{t+1}(\mathbf{x}'_i)\mathbf{a}^{(\ell-1)}_t(\mathbf{x}_i)^T}{||\mathbf{h}^{(\ell)}_{t+1}(\mathbf{x}'_i)\mathbf{a}^{(\ell-1)}_t(\mathbf{x}_i)^T||_F}, \tag{5.14}$$

where γ is the step size of the updates and $||\cdot||_F$ is the Frobenius norm. We perform this normalization for numerical stability. Note that this update using average pooling ensures that the resulting hidden

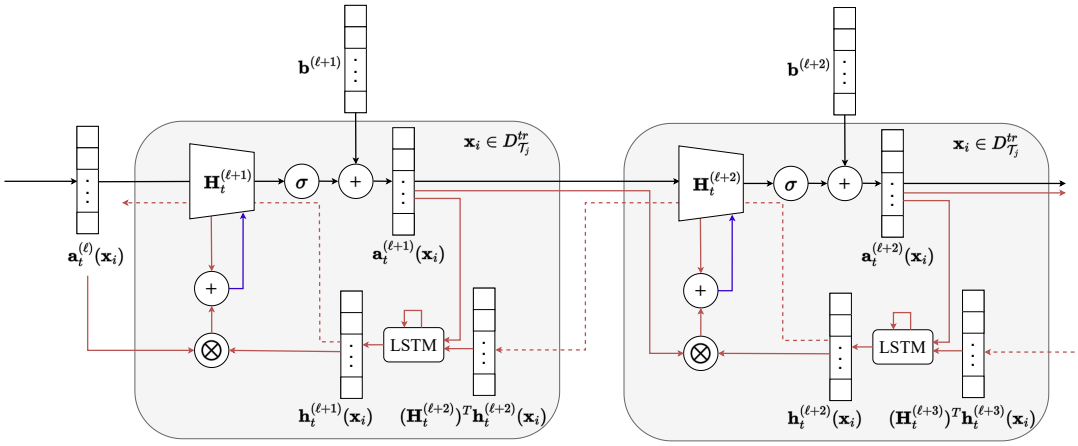


Figure 5.4: The workflow of OP-LSTM. We have visualized two layers of the base-learner network. During the forward pass, the 2D hidden states $\mathbf{H}_t^{(\ell)}$ act as weight matrices of a feed-forward neural network that act on the input of that layer $\mathbf{a}_t^{(\ell-1)}$. This linear combination $\mathbf{H}_t^{(\ell)}\mathbf{a}_t^{(\ell-1)}(\mathbf{x}_i)$ is passed through a nonlinearity σ and added with a bias vector $\mathbf{b}^{(\ell)}$ to produce the activation $\mathbf{a}_t^{(\ell)}(\mathbf{x}_i)$. The entire forward pass is displayed by the black arrows. The red arrows, on the other hand, indicate the backward pass using the coordinate-wise LSTM. The outer product ($\otimes$) of the resulting hidden state $\mathbf{h}_t^{(\ell+1)}$ and the inputs from the previous layer $\mathbf{a}_t^{(\ell)}$ are added to the 2D hidden state $\mathbf{H}_t^{(\ell+1)}$ to produce $\mathbf{H}_{t+1}^{(\ell+1)}$ (blue arrow), which can be interpreted as the updated weight matrix.

states $\mathbf{H}_t^{(\ell)}$ are invariant to permutations of the support data. Moreover, we observe that this equation defines the backward dynamics of the architecture (updating the 2D hidden states). However, this equation does not yet tell us how the hidden states $\mathbf{h}_{t+1}^{(\ell)}(\mathbf{x}_i')$ are computed.

We use a coordinate-wise LSTM so that the same LSTM can be used in layers of arbitrary dimensions, in similar fashion as Ravi and Larochelle (2017); Andrychowicz et al. (2016). This means that we maintain a state $s_{t,j}^{(\ell)} = [h_{t,j}^{(\ell)}, c_{t,j}^{(\ell)}]$ for every individual node j in the state vector and every layer $\ell \in \{1, 2, \ldots, L\}$ over time steps t. In order to obtain the hidden state vector for a given layer ℓ and time step t, we simply concatenate the individual hidden states computed by the coordinate-wise LSTM, i.e., $\mathbf{h}_t^{(\ell)} = [h_{t,1}^{(\ell)}, h_{t,2}^{(\ell)}, \ldots, h_{t,d^{(\ell)}}^{(\ell)}]^T$, where $d^{(\ell)}$ is the number of neurons in layer ℓ. The LSTM weights to update these states are shared across all layers and nodes with the same activation function. For classification experiments, we often have two LSTMs: one for the final layer which uses a softmax activation function, and one for the body of the network, which uses the ReLU activation. This allows OP-LSTM to learn weight updates akin to gradient descent, where the backward computation is tied to each nonlinearity in the base-learner network (as this can not be done by a single LSTM, due to the different non-linearities of the softmax and the RELU activation). We use pooling over the support data in order to update the states using a coordinate-wise approach, where every element of the hidden state $\mathbf{h}_t^{(\ell)}$ of a given layer ℓ is updated independently by a single LSTM.

In order to compute the next state $s_{t+1,j}^{(\ell)}$ of node j in layer ℓ, we need to have the previous state consisting of the previous hidden state $h_{t,j}^{(\ell)}$ and cell state $c_{t,j}^{(\ell)}$ of that node. Moreover, we need to

feed the LSTM an input, which we define as $z_{t,j}^{(\ell)}$. In OP-LSTM, we define this input as

$$z_{t,j}^{(\ell)}(\mathbf{x}_i') = \begin{cases} [(\mathbf{a}_t^{(\ell)}(\mathbf{x}_i))_j, (\mathbf{y}_i)_j] & \text{if } \ell = L \text{ (output layer)}, \\ [(\mathbf{a}_t^{(\ell)}(\mathbf{x}_i))_j, ((\mathbf{H}_t^{(\ell+1)})^T \mathbf{h}_t^{(\ell+1)}(\mathbf{x}_i))_j] & \text{otherwise.} \end{cases} \tag{5.15}$$

Note that for the output layer L, the input to the LSTM corresponds to the current prediction and the ground-truth output, which share the same dimensionality. For the earlier layers in the network, we do not have access to the ground-truth signals. Instead, we view the hidden state of the output layer LSTM as errors and propagate them backward through the 2D hidden states $\mathbf{H}_t^{(\ell+1)}$, hence the expression $(\mathbf{H}_t^{(\ell+1)})^T \mathbf{h}_t^{(\ell+1)}(\mathbf{x}_i)$ for earlier layers. We note that this is akin to backpropagation, where error messages $\delta^{(\ell+1)}$ are passed backward through the weights of the network.

Given an input $\mathbf{x}_i'$, the next state $s_{t+1,j}^{(\ell)}$ can then be computed by applying the LSTM m_θ to the input $z_{t,j}^{(\ell)}(\mathbf{x}_i')$, conditioned on the previous hidden state $h_{t,j}^{(\ell)}$ and cell state $c_{t,j}^{(\ell)}$.

$$s_{t+1,j}^{(\ell)}(\mathbf{x}_i') = [h_{t+1,j}^{(\ell)}(\mathbf{x}_i'), c_{t+1,j}^{(\ell)}(\mathbf{x}_i')] = m_\theta(z_{t,j}^{(\ell)}(\mathbf{x}_i'); h_{t,j}^{(\ell)}, c_{t,j}^{(\ell)}), \tag{5.16}$$

where $z_{t,j}^{(\ell)}(\mathbf{x}_i')$ is the input to the LSTM used to update the state. These individual states are averaged over all training inputs to obtain

$$s_{t+1,j}^{(\ell)} = [h_{t+1,j}^{(\ell)}, c_{t+1,j}^{(\ell)}] = \frac{1}{M} \sum_{i=1}^{M} s_{t+1,j}^{(\ell)}(\mathbf{x}_i'). \tag{5.17}$$

Note that we can obtain a state vector, hidden vector, and cell state vector, by concatenation, i.e., $\mathbf{s}_{t+1,j}^{(\ell)}(\mathbf{x}_i') = [s_{t+1,1}^{(\ell)}(\mathbf{x}_i'), s_{t+1,2}^{(\ell)}(\mathbf{x}_i'), \ldots, s_{t+1,d^{(\ell)}}^{(\ell)}(\mathbf{x}_i')]$, $\mathbf{h}_{t+1,j}^{(\ell)}(\mathbf{x}_i') = [h_{t+1,1}^{(\ell)}(\mathbf{x}_i'), h_{t+1,2}^{(\ell)}(\mathbf{x}_i'), \ldots, h_{t+1,d^{(\ell)}}^{(\ell)}(\mathbf{x}_i')]$, and $\mathbf{c}_{t+1,j}^{(\ell)}(\mathbf{x}_i') = [c_{t+1,1}^{(\ell)}(\mathbf{x}_i'), c_{t+1,2}^{(\ell)}(\mathbf{x}_i'), \ldots, c_{t+1,d^{(\ell)}}^{(\ell)}(\mathbf{x}_i')]$.

5.4.2 The algorithm

OP-LSTM is trained to minimize the expected loss on the query sets conditioned on the support sets, where the expectation is with respect to a distribution of tasks. Put more precisely, we wish to minimize $\mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})} \left[\mathcal{L}_{D_{\mathcal{T}_j}^{te}}(\Theta) \right]$, where $\Theta = \{\theta, \mathbf{H}_0^{(1)}, \mathbf{H}_0^{(2)}, \ldots, \mathbf{H}_0^{(L)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \ldots, \mathbf{b}^{(L)}\}$. This objective can be approximated by sampling batches of tasks, updating the weights using the learned outer product rules, and evaluating the loss on the query sets. Across tasks, we update Θ using gradient descent. In practice, we use the cross-entropy loss for classification tasks and the MSE loss for regression tasks.

The pseudocode for OP-LSTM is displayed in Algorithm 11. First, we randomly initialize the initial 2D hidden states $\mathbf{H}_0^{(\ell)}$ and the LSTM parameters θ. We group these parameters as $\Theta = \{\theta, \mathbf{H}_0^{(1)}, \mathbf{H}_0^{(2)}, \ldots, \mathbf{H}_0^{(L)}\}$, which will be meta-learned across different tasks. Given a task $\mathcal{T}_j$, we make T updates on the entire support set $D_{\mathcal{T}_j}^{tr}$ by processing the examples individually, updating the 2D hidden states $\mathbf{H}_t^{(\ell)}$, and computing the new hidden states of the coordinate-wise LSTM for every layer $\mathbf{s}_t^{(\ell)}$. After having made T updates on the support data, we compute the loss of the model on the query set $D_{\mathcal{T}_j}^{te}$. The gradient of this loss with respect to all parameters Θ is added to the gradient buffer. Once a batch of tasks B has been processed in this way, we perform a gradient update on Θ and repeat this process until convergence or a maximum number of iterations has been

reached.

Algorithm 11 Meta-learning with outer product LSTM (OP-LSTM)

Randomly initialize $\mathbf{H}_0^{(\ell)}$ and biases $\mathbf{b}^{(\ell)}$ for all $1 \leq \ell \leq L$
Randomly initialize LSTM parameters θ, set $\mathbf{h}_o^{(\ell)} = \mathbf{0}$, $\mathbf{c}_0^{(\ell)} = \mathbf{0}$
repeat
    Initialize gradient buffer $\zeta = \mathbf{0}$
    Sample batch of J tasks $B = \{\mathcal{T}_j \backsim p(\mathcal{T})\}_{j=1}^{J}$
    for $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{te})$ in B **do**
        set $\mathbf{h}_o^{(\ell)} = \mathbf{0}$, $\mathbf{c}_0^{(\ell)} = \mathbf{0}$ for all $1 \leq \ell \leq L$
        for $t = 1, ..., T$ **do**
            for $\mathbf{x}_i' = (\mathbf{x}_i, \mathbf{y}_i) \in D_{\mathcal{T}_j}^{tr}$ **do**
                Compute predictions $\mathbf{a}_{t-1}^{(L)}(\mathbf{x}_i)$ (Equation 5.13)
                Compute $\mathbf{z}_t^{(\ell)}(\mathbf{x}_i')$ and $\mathbf{h}_t^{(\ell)}(\mathbf{x}_i)$ for $1 \leq \ell \leq L$ with backward message passing (see Equation 5.15 and Equation 5.16)
                Update $\mathbf{H}_t^{(\ell)}$ for $1 \leq \ell \leq L$ (Equation 5.14)
            end for
            Compute $\mathbf{s}_t^{(\ell)}$ for $1 \leq \ell \leq L$ (Equation 5.17) through concatenation
        end for
        Compute query predictions $\mathcal{L}_{D_{\mathcal{T}_j}^{te}}(\{\mathbf{H}_T^{(\ell)}\}_{\ell=1}^{L})$
        Update gradient buffer $\zeta = \zeta + \frac{1}{J}\nabla_{\Theta}\mathcal{L}_{D_{\mathcal{T}_j}^{te}}(\{\mathbf{H}_T^{(\ell)}\}_{\ell=1}^{L})$
    end for
    Update $\mathbf{\Theta} = \Theta - \beta\zeta$
until convergence

5.5 Experiments

In this section, we aim to answer the following research questions:

- How do the performance and training stability of a plain LSTM compare when processing the support data as a sequence versus as a set with average pooling? (see Section 5.5.1)
- How well does the plain LSTM perform at few-shot sine wave regression and within- and cross-domain image classification problems compared with popular meta-learning methods such as MAML (Finn et al., 2017) and Prototypical network (Snell et al., 2017)? (see Section 5.5.2 and Section 5.5.3)
- Does OP-LSTM yield a performance improvement over the simple LSTM and the related approaches MAML and Prototypical network in few-shot sine wave regression and within- and cross-domain image classification problems? (see Section 5.5.2 and Section 5.5.3)
- How does OP-LSTM adjust the weights of the base-learner network? (see Section 5.5.4)

For our experiments, we use few-shot sine wave regression (Finn et al., 2017) as an illustrative task, and popular few-shot image classification benchmarks, namely Omniglot (Lake et al., 2015), miniImageNet (Ravi and Larochelle, 2017; Vinyals et al., 2016), and CUB (Wah et al., 2011). We

use MAML (Finn et al., 2017), prototypical network (Snell et al., 2017), SAP (see Chapter 6) and Warp-MAML (Flennerhag et al., 2020) as baselines. The former two are popular meta-learning methods and can both be approximated by the OP-LSTM (see Section 5.6), allowing us to investigate the benefit of OP-LSTM's expressive power. The last two baselines are used to investigate how OP-LSTM compares to state-of-the-art gradient-based meta-learning methods in terms of performance, although it has to be noted that that OP-LSTM is orthogonal to that method, in the sense that OP-LSTM could be used on top of Warp-MAML. However, this is a nontrivial extension and we leave this for future work. We run every technique on a single GPU (PNY GeForce RTX 2080TI) with a computation budget of 2 days (for detailed running times, please see Section B.2.3). Each experiment is performed with 3 different random seeds, where the random seed affects the random weight initialization of the neural networks as well as the used training tasks, validation tasks, and testing tasks. Below, we describe the different experimental settings that we use. Note that we do not aim to achieve state-of-the-art performance, but rather investigate whether the plain LSTM is a competitive method for few-shot learning on modern benchmarks and whether OP-LSTM yields improvements over the plain LSTM, MAML, and Prototypical network.

Sine wave regression This toy problem was originally proposed by Finn et al. (2017) to study meta-learning methods. In this setting, every task $\mathcal{T}_j$ corresponds to a sine wave $s_j = A_j \cdot \sin(x - p_j)$, where A_j and p_j are the amplitude and phase of the task, sampled uniformly at random from the intervals [0.1, 5.0] and [0, π], respectively. The goal is to predict for a given task the correct output y given an input x after training on the support set, consisting of k examples. The performance of learning is measured in the query set, consisting of 50 input-output. For the plain LSTM approach, we use a multi-layer LSTM trained with Backpropagation through Time (BPTT) using Adam (Kingma and Ba, 2015). During meta-training, the LSTM is shown 70 000 training tasks. Every 2 500 tasks, we perform meta-validation on 1 000 tasks, and after having selected the best validated model, we evaluate the performance on 2 000 meta-test tasks.

Few-shot image classification In case of few-shot image classification, all methods are trained for 80 000 episodes on training tasks and we perform meta-validation every 2 500 episodes. The best learner is then evaluated on 600 hold-out test tasks, each task having a number of examples per class in the support set as indicated by the experiment (ranging from 1–10) as well as a query set of 15 examples per class. We repeat every experiment 3 times with different random seeds, meaning the that weight initializations and tasks are different across runs, although the class splits for sampling training/validation/testing tasks are kept fixed. For the Omniglot image classification dataset, we used a fully-connected neural network as base-learner for MAML and OP-LSTM, following Santoro et al. (2016) and Finn et al. (2017). The network consists of 4 fully-connected blocks with dimensions 256-128-64-64. Every block consists of a linear layer, followed by BatchNorm and ReLU activation. Every layer of the base-learner network is an OP-LSTM block. The plain LSTM approach uses an LSTM as base-learner. For MAML, we use the best reported hyperparameters by Finn et al. (2017). We performed hyperparameter tuning for LSTM and OP-LSTM using random search and grid search, respectively (details can be found in Appendix B.2). Note that as such, the comparison against MAML and Prototypical networks is only for illustrative purposes, as the hyperparameter optimization procedure on these methods has, due to computational restrictions, not been executed under the same conditions.

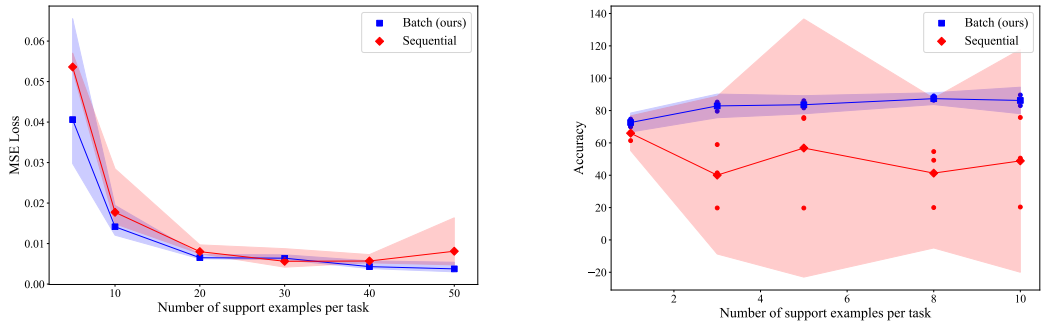
For the miniImageNet and CUB image classification datasets, we use the Conv-4 base-learner network for all methods, following Snell et al. (2017); Finn et al. (2017). This base-learner consists of 4 blocks, where every block consists of 64 feature maps created with 3×3 kernels, BatchNorm, and ReLU

nonlinearity. MAML uses a linear output layer to compute predictions, the plain LSTM operates on the flattened features extracted by the convolutional layers (as an LSTM taking image data as input does not scale well), whereas OP-LSTM uses an OP-LSTM block (see Figure 5.4) on these flattened features. Importantly, OP-LSTM is only used in the final layer as it does currently not support propagating messages backward through max pooling layers.

We first study the within-domain performance of the meta-learning methods, where test tasks are sampled from the same dataset as the one used for training (albeit with unseen classes). Afterward, we also study the cross-domain performance, where the techniques train on tasks from a given dataset and are evaluated on test tasks from another dataset. More specifically, we use the scenarios miniImageNet → CUB (train on miniImageNet and evaluate on CUB) and vice versa.

5.5.1 Permutation invariance for the plain LSTM

First, we investigate the difference in performance of the plain LSTM approach when processing the support data as a sequence $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_k, \mathbf{y}_k)$ or as a set $\{(\mathbf{x}_1, \mathbf{y}1), \ldots, (\mathbf{x}_k, \mathbf{y}_k)\}$ (see Section 5.3.4) on few-shot sine wave regression and few-shot Omniglot classification. For the former, every task consists of 50 query examples, whereas for the latter, we have 10 query examples per class. We tuned the LSTM that processes the support data sequentially with random search (details in appendix). We compare the performance of this tuned sequential model to that of an LSTM with batching (with the same hyperparameter configuration) to see whether the resulting permutation invariance is helpful for the performance and training stability of the LSTM. To measure the stability of the training process, we compute the confidence interval over the mean performances obtained over 3 different runs rather than over all performances concatenated for the different runs, as done in later experiments for consistency with the literature.



(a) Sinewave - Average MSE loss

(b) Omniglot - Average accuracy (%)

Figure 5.5: The average accuracy score of a plain LSTM with sequential and batch support data processing on few-shot sine wave regression (left) and Omniglot classfication (right) for different numbers of training examples per task. Note that a lower MSE (left) or a higher accuracy (right) corresponds to better performance. The results are averaged over 3 runs (each measured over 600 meta-test tasks) with different random seeds and the 95% confidence intervals over the mean performances of the runs are shown as shaded regions. Additionally, in the right plot, we have added scatter marks to indicate the average performances per run (dots, unconnected, 3 per setting). Batch processing performs on par or outperforms sequential processing and improves the training stability over different runs.

The results of this experiment are shown in Figure 5.5. In the case of few-shot sine wave regression (left subfigure), the performance of the LSTM with batching is on par or better compared with the sequential LSTM as the MSE score of the former is smaller or equal. We also note that the performance tends to improve with the amount of available training data. A similar, although more convincing, pattern can be seen in the case of few-shot Omniglot classification (right subfigure), where the LSTM with batching significantly outperforms the sequential LSTM across the different numbers of training examples per class. Surprisingly, in this case the performance of the LSTM does not improve as the number of examples per class increases. We found that this is due to training stability issues of the plain LSTM (as shown by the confidence intervals): for some runs, the LSTM does not learn and yields random performance, and in other runs the learning starts only after a certain period of burn-in iterations and fails to reach convergence within 80K meta-iterations (see appendix Section B.2.2 for detailed learning curves for every run). For the LSTM with batching, we do not observe such training stability issues. This shows that batching not only helps improve the performance, but also greatly increases the training stability. Note that the fact that the shaded confidence interval of the sequential LSTM goes above the performance obtained by the batching LSTM is an artifact of using symmetrical confidence intervals above and below the mean trend: the sequential LSTM never outperforms the batching LSTM. As we can see, the MSE loss for both approaches decreases as the size of the support set increases, as more training data is available for learning. Furthermore, we see that the performance of the LSTM with batching improves with the number of available training data, whereas this is not the case for the sequential LSTM, which struggles to yield competitive performance. Overall, the results imply that the permutation invariance is a helpful inductive bias to improve the few-shot learning performance. Consequently, we will use the LSTM with batching henceforth.

5.5.2 Performance comparison on few-shot sine wave regression

Next, we compare the performance of the plain LSTM with batching, our proposed OP-LSTM, as well as MAML (Finn et al., 2017). To ensure a fair comparison with MAML, we tuned the hyperparameters in the same way as for the plain LSTM as done in the previous subsection on 5-shot sine wave regression. For this tuning, we used the default base-learner architecture consisting of two hidden layers with 40 ReLU nodes, followed by an output layer of 1 node. Afterward, we searched over different architectures with different numbers of parameters such that the expressivity in terms of the number of parameters does not limit the performance of MAML. We used the same base-learner architecture for the OP-LSTM as MAML without additional tuning.

Table 5.1: Average test MSE on few-shot sine wave regression. The 95% confidence intervals are displayed as $\pm x$, and calculated over all meta-test tasks. We used batch processing for the LSTM and OP-LSTM. The best performances are dislpayed in bold font.

	Parameters	5-shot	10-shot	20-shot
MAML	17 018	0.18 ± 0.009	0.033 ± 0.003	0.005 ± 0.001
LSTM	20 201	**0.04** ± 0.002	0.010 ± 0.001	0.007 ± 0.000
OP-LSTM	18 107	0.11 ± 0.009	**0.008** ± 0.001	**0.003** ± 0.000

The test performances on the sine wave regression taska are displayed in Table 5.1. We note MAML, despite having a comparable number of parameters (models with more parameters than LSTM and OP-LSTM performed worse), is outperformed by LSTM and OP-LSTM, indicating that LSTM and OP-LSTM have discovered more efficient learning algorithms for sine wave tasks. Comparing LSTM

with OP-LSTM, we see that the former yields the best performance in the 5-shot setting, whereas OP-LSTM outperforms LSTM in the 10-shot and 20-shot settings.

5.5.3 Performance comparison on few-shot image classification

Within-domain Next, we investigate the within-domain performance of OP-LSTM and LSTM on few-shot image classification problems, namely, Omniglot, miniImageNet, and CUB. The results for the Omniglot dataset are displayed in Table 5.2. Note that the LSTM has many more parameters than the other methods as it consists of multiple fully-connected layers with large hidden dimensions, which were found to give the best validation performance. As we can see, the plain LSTM (with batching) does not yield competitive performance compared with the other methods, in spite of the fact that it has many more parameters and, in theory, could learn any learning algorithm. This shows that the LSTM is hard to optimize and struggles to find a good solution in more complex few-shot learning settings, i.e., image classification. OP-LSTM, on the other hand, which separates the learning procedure from the input representation, yields competitive performance compared with MAML and ProtoNet in both the 1-shot and 5-shot settings, whilst using fewer parameters than the plain LSTM.

Table 5.2: The mean test accuracy (%) on 5-way Omniglot classification across 3 different runs. The 95% confidence intervals are displayed as $\pm x$, and calculated over all runs and meta-test tasks (600 per run). The plain LSTM is outperformed by MAML. All methods (except LSTM) used a fully-connected feed-forward classifier. The best performances are dislpayed in bold font.

Technique	parameters	1-shot	5-shot
MAML	247 621	84.1 ± 0.90	**93.5** ± 0.30
ProtoNet	247 621	83.6 ± 0.88	93.4 ± 0.29
LSTM	13 530 097	72.6 ± 0.90	84.8 ± 0.50
OP-LSTM (ours)	249 167	**84.3** ± 0.90	91.8 ± 0.30

The results for miniImageNet and CUB are displayed in Table 5.3. Note that again, the LSTM uses more parameters than other methods as it consists of multiple large fully-connected layers which were found to yield the best validation performance. Nonetheless, it is applied on top of representations computed with the Conv-4 backbone, which is also used by all other methods. As we can see, the plain LSTM approach performs at chance level, again suggesting that the optimization problem of finding a good learning algorithm is too complex for this problem. The OP-LSTM, on the other hand, yields competitive or superior performance compared with all tested baselines on both miniImageNet and CUB, regardless of the number of shots, which shows the advantage of decoupling the input representation from the learning procedure.

Cross-domain Next, we investigate the cross-domain performance of the LSTM and OP-LSTM, where the test tasks come from a different a different dataset than the training tasks. We test this in two scenarios: train on miniImageNet and evaluate on CUB (MIN $\rightarrow$ CUB) and vice verse (CUB $\rightarrow$ MIN). The results of this experiment are displayed in Table 5.4. Again, the plain LSTM does not outperform a random classifier, whilst the OP-LSTM yields superior performance in every tested scenario, showing its versatility in this challenging setting.

Table 5.3: Meta-test accuracy scores on 5-way miniImageNet and CUB classification over 3 runs. The 95% confidence intervals are displayed as ± x, and calculated over all runs and meta-test tasks (600 per run). All methods used a Conv-4 backbone as a feature extractor. The "-" indicates that the method did not finish within 2 days of running time. The best performances are dislpayed in bold font.

		miniImageNet		CUB	
Technique	parameters	1-shot	5-shot	1-shot	5-shot
MAML	121 093	48.6 ± 1.04	63.0 ± 0.54	57.5 ± 1.04	**74.8** ± 0.51
Warp-MAML	231 877	50.4 ± 1.04	65.6 ± 0.53	59.6 ± 1.00	74.2 ± 0.51
SAP	412 852	**53.0** ± 1.08	67.6 ± 0.51	**63.5** ± 1.00	73.9 ± 0.51
ProtoNet	121 093	50.1 ± 1.04	65.4 ± 0.53	50.9 ± 1.01	63.7 ± 0.55
LSTM	55 879 349	20.2 ± 0.20	19.4 ± 0.20	-	-
OP-LSTM (ours)	141 187	51.9 ± 1.04	**67.9** ± 0.50	60.2 ± 1.04	73.1 ± 0.52

Table 5.4: Average cross-domain meta-test accuracy scores over 5 runs using a Conv-4 backbone. Techniques trained on tasks from one data set and were evaluated on tasks from another data set. The 95% confidence intervals are displayed as ± x, and calculated over all runs and meta-test tasks. The "-" indicates that the method did not finish within 2 days of running time. The best performances are dislpayed in bold font.

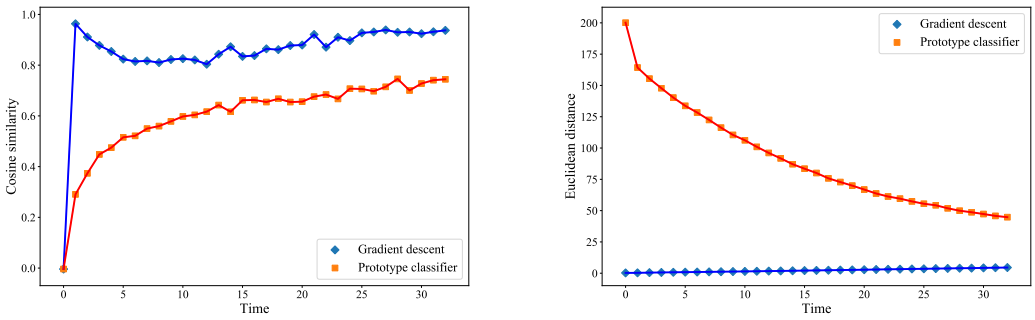
	MIN → CUB		CUB → MIN	
	1-shot	5-shot	1-shot	5-shot
MAML	37.9 ± 0.40	53.6 ± 0.40	31.1 ± 0.36	45.8 ± 0.39
Warp-MAML	42.0 ± 0.43	56.9 ± 0.42	31.1 ± 0.35	41.3 ± 0.36
SAP	41.5 ± 0.44	58.0 ± 0.41	33.3 ± 0.39	47.1 ± 0.39
ProtoNet	39.7 ± 0.41	56.0 ± 0.41	31.7 ± 0.34	45.3 ± 0.38
LSTM	20.1 ± 0.28	20.0 ± 0.25	-	-
OP-LSTM (ours)	**42.3** ± 0.42	**58.5** ± 0.41	**35.8** ± 0.40	**49.0** ± 0.40

5.5.4 Analysis of the learned weight updates

Lastly, we investigate how OP-LSTM updates the weights of the base-learner network. More specifically, we measure the cosine similarity and Euclidean distance between the OP-LSTM updates and updates made by gradient descent or Prototypical network. Denoting the initial final classifier weight matrix as $\mathbf{H}_0^{(L)}$, the OP-LSTM update direction after T updates is $\Delta_{OP} = \vec{\mathbf{H}}_T^{(L)} - \vec{\mathbf{H}}_0^{(L)}$, where $\vec{\mathbf{M}}$ means that we vectorize the matrix by flattening it. Similarly, we can measure the update compared with the initial weight matrix and those obtained by employing nearest-prototype classification ($\mathbf{H}_{Proto}^{(L)}$) as done in Prototypical network or gradient descent $\mathbf{H}_{GD}^{(L)}$, where the latter is obtained by performing T gradient update steps (with a learning rate of 0.01). These updates are associated with the update direction vectors $\Delta_{Proto} = \vec{\mathbf{H}}_{Proto}^{(L)} - \vec{\mathbf{H}}_0^{(L)}$ and $\Delta_{GD} = \vec{\mathbf{H}}_{GD}^{(L)} - \vec{\mathbf{H}}_0^{(L)}$. We can then measure the distance between the update direction Δ_{OP} of the OP-LSTM and Δ_{Proto} and Δ_{GD}. As a distance measure, we use the Euclidean distance. In addition, we also measure the cosine similarity between the update directions as an inverse distance measure that is invariant to the scale

and magnitudes of the vectors. After every 2 500 episodes, we measure these Euclidean distances and cosine similarity scores on the validation tasks, and average the results over 3 runs.

The results of this experiment are displayed in Figure 5.6. As we can see, the cosine similarity between the weight update directions of OP-LSTM and gradient descent and prototype-based classifiers increases with training time. OP-LSTM very quickly learns to update the weights in a similar direction as gradient descent, followed by a gradual decline in similarity, which is later followed by a gradual increase. This gradual decline may be to incorporate more prototype-based updates. Looking at the Euclidean distance, we observe the same pattern for the similarity compared with the prototype-based classifier, as the distance between the updates decreases (indicating a higher similarity). The Euclidean distance between OP-LSTM updates and gradient updates slightly increase over time, which may be a side effect of the sensitivity to scale and magnitude of this distance measure. Thus, even if both would perform gradient descent, but with different learning rates, The cosine similarity gives a better idea of directional similarity as it abstracts away from the magnitude of the vectors.



(a) Cosine similarity

(b) Euclidean distance

Figure 5.6: The average cosine similarity (left) and Euclidean distance (right) between the weight update directions of the OP-LSTM and a prototype-based and gradient-based classifier as a function of time on 5-way 1-shot miniImageNet classification. Each point on the x-axis indicates a validation step, which is performed after every 2 500 episodes. The results are averaged over 3 runs with different random seeds and the 95% confidence intervals are shown as shaded regions. The confidnce intervals are within the size of the symbols and imperceptible. As time progresses, the updates performed by OP-LSTM become more similar to those of gradient descent and prototype-based classifiers (increasing cosine similarity).

5.6 Relation to other methods

Here, we study the relationship of OP-LSTM to other existing meta-learning methods. More specifically, we aim to show that OP-LSTM is a general meta-learning approach, which can *approximate* the behaviour of different classes of meta-learning, such as optimization-based meta-learners (e.g., MAML, Finn et al. (2017)) and metric-based methods (e.g., Prototypical network, Snell et al. (2017)).

Model-agnostic meta-learning (MAML) MAML (Finn et al., 2017) aims to learn good initialization parameters for a base-learner network $\theta = \{\mathbf{W}_0^{(1)}, \mathbf{W}_0^{(2)}, \ldots, \mathbf{W}_0^{(L)}, \mathbf{b}_0^{(1)}, \mathbf{b}_0^{(2)}, \ldots, \mathbf{b}_0^{(L)}\}$ such that new tasks can be learned efficiently using a few gradient update steps. Here, $\mathbf{W}_0^{(\ell)}$ is the initial weight matrix of layer ℓ and $\mathbf{b}_0^{(\ell)}$ the initial bias vector of layer ℓ when presented with a new task.

The initial 2D hidden states $\mathbf{H}_0^{(\ell)}$ in OP-LSTM can be viewed as the initial weights $\mathbf{W}_0^{(\ell)}$ of the neural network in MAML. In MAML, the weights in layer ℓ for a given input are updated as $\mathbf{W}_{t+1}^{(\ell)} = \mathbf{W}_t^{(\ell)} - \eta\delta^{(\ell)}(\mathbf{p}^{(\ell-1)}(\mathbf{x}))^T$, where $\mathbf{a}^{(i)}(\mathbf{x})$ (with $1 \leq i \leq L$) is the vector of post-activation values in layer i as a result of the input $\mathbf{x}$, and $\delta^{(i)} = \nabla_{\mathbf{a}^{(i)}}\mathcal{L}(\mathbf{x})$, where $\mathcal{L}(\mathbf{x}, \mathbf{y})$ is the loss on input $\mathbf{x}$ given the ground-truth target $\mathbf{y}$, and η is the learning rate.

Instead of using this hand-crafted weight update rule, OP-LSTM learns the update rule using the outer product of LSTM hidden states and the input activation. From Equation 5.14 it follows that OP-LSTM is capable of updating the weights $\mathbf{H}_t^{(\ell)}$ with gradient descent by setting $\mathbf{h}_{t+1}^{(\ell)}(\mathbf{x}) = -\eta\delta^{(\ell)} = -\eta\nabla_{\mathbf{a}^{(\ell)}}\mathcal{L}(\mathbf{x}, \mathbf{y})$. (Note that in Equation 5.14 the gradient is also normalized by the Frobenius norm, which is formally not part of MAML.) We note that the inputs to the coordinate-wise LSTM contain the necessary information to compute the errors $\delta^{(\ell)}$ in every layer. That is, for the output layer, the LSTM receives the ground-truth output and prediction in the output layer. For earlier layers, the LSTM receives the backpropagated messages (the errors), as well as the activations. Consequently, OP-LSTM can update the 2D hidden states $\mathbf{H}^{(\ell)}$ with gradient descent, as MAML. OP-LSTM is thus an approximate generalization of MAML as it could learn to perform the same weight matrix updates, although OP-LSTM does not update the bias vectors given a task.

Prototypical network Prototypical network (Snell et al., 2017) aims to learn good initial weights $\theta = \{\mathbf{W}_0^{(1)}, \mathbf{W}_0^{(2)}, \ldots, \mathbf{W}_0^{(L-1)}, \mathbf{b}_0^{(1)}, \mathbf{b}_0^{(2)}, \ldots, \mathbf{b}_0^{(L-1)}\}$ for all parameters except for the final layer, such that a nearest-prototype classifier yields good performance. Let $f_\theta(\mathbf{x}_i)$ be the embeddings produced by this $(L-1)$-layered network for a given observation $\mathbf{x}_i$ (from the support set). Note that the network has $(L-1)$ layers as this is the feature embedding module without the output layer. Prototypical network computes centroids $\mathbf{c}_n = \frac{1}{|X_n|}\sum_{\mathbf{x}_i \in X_n} f_\theta(\mathbf{x}_i)$ for every class n, where X_n is the set of all support inputs with ground-truth class n, and $f_\theta(\mathbf{x})$ is the embedding of input $\mathbf{x}$. Then, the predicted score of a new input $\hat{\mathbf{x}}$ for class n is then given by $\hat{y}_n(\hat{\mathbf{x}}) = \frac{\exp(-d(f_\theta(\hat{\mathbf{x}}), \mathbf{c}_n))}{\sum_{n'}\exp(-d(f_\theta(\hat{\mathbf{x}}), \mathbf{c}_{n'}))}$, where $d(\mathbf{x}_i, \mathbf{x}_j) = ||\mathbf{x}_i - \mathbf{x}_j||_2^2$ is the squared Euclidean distance, and n' is a variable iterating over all classes.

This nearest-prototype classifier can be seen as a regular linear output layer (Triantafillou et al., 2020). To see this, note that the prediction score for class j is given by

$$\hat{y}_j(\hat{\mathbf{x}}) = ||f_\theta(\hat{\mathbf{x}}) - \mathbf{c}_n||_2^2 = (f_\theta(\hat{\mathbf{x}}) - \mathbf{c}_n)^T(f_\theta(\hat{\mathbf{x}}) - \mathbf{c}_n) \tag{5.18}$$

$$= f_\theta(\hat{\mathbf{x}})^T f_\theta(\hat{\mathbf{x}}) - 2f_\theta(\hat{\mathbf{x}})^T\mathbf{c}_n + \mathbf{c}_n^T\mathbf{c}_n \tag{5.19}$$

$$\propto -2f_\theta(\hat{\mathbf{x}})^T\mathbf{c}_n + \mathbf{c}_n^T\mathbf{c}_n, \tag{5.20}$$

where we ignored the first term $(f_\theta(\hat{\mathbf{x}})^T f_\theta(\hat{\mathbf{x}}))$ as it is constant across all classes n. The prediction score for class j is thus obtained by taking the dot product between the input embedding $f_\theta(\hat{\mathbf{x}})$ and $-2\mathbf{c}_n$ and by adding a bias term $b_n = \mathbf{c}_n^T\mathbf{c}_n$. Thus, the prototype-based classifier is equivalent to a linear output layer, i.e., $\hat{\mathbf{x}} = \mathbf{W}^{(L)}f_\theta(\hat{\mathbf{x}}) + \mathbf{b}^{(L)}$ where the n-th row of $\mathbf{W}^{(L)}$ corresponds to $-2\mathbf{c}_n$ and the n-th element of $\mathbf{b}^{(L)}$ is equal to $\mathbf{c}_n^T\mathbf{c}_n$. OP-LSTM can approximate the behavior of Prototypical

network with $T = 1$ steps per task as follows. First, assume that the underlying base-learner network is the same for Prototypical network and OP-LSTM, i.e., the initialization of the hidden state is equivalent to the initial weights of the base-learner used by Prototypical network $\mathbf{H}_0^{(\ell)} = \mathbf{W}_0^{(\ell)}$ for $\ell \in \{1, 2, \ldots, L-1\}$, and that the hidden state of the output layer in OP-LSTM is a matrix of zeros, i.e., $\mathbf{H}_0^{(L)} = \mathbf{0}$. Second, let the hidden states of the LSTM in OP-LSTM be be a vector of zeros $\mathbf{h}^{(\ell)} = \mathbf{0}$ for every layer $\ell < L$, and let the hidden state of the output layer given the example $\mathbf{x}'_i = (\mathbf{x}_i, \mathbf{y}_i)$ be the label identity function $\mathbf{h}^{(L)}(\mathbf{x}'_i) = \mathbf{y}_i$ (which can be learned by an LSTM). Then, OP-LSTM will update the hidden states as follows using Equation 5.14. The n-th row of $\mathbf{H}^{(L)}$ will equal $\gamma \frac{1}{M} \sum_{\mathbf{x}_i \in X_n} \frac{\mathbf{a}^{(L-1)}(\mathbf{x}_i)}{||\mathbf{a}^{(L-1)}(\mathbf{x}_i)||_F}$, where $X_n = \{\mathbf{x}_i \in D_{\mathcal{T}_j}^{tr} | \mathbf{y}_i = \mathbf{e}_n\}$ is the set of training inputs with class n, and γ and M are the learning rate of OP-LSTM and number of examples respectively. Note that this expression corresponds to the scaled prototype (mean of the embeddings) of class n, that is, $\gamma \bar{\mathbf{c}}_n$, where $\bar{\mathbf{c}}_n = \frac{1}{M} \sum_{\mathbf{x}_i \in X_n} \frac{\mathbf{a}^{(L-1)}(\mathbf{x}_i)}{||\mathbf{a}^{(L-1)}(\mathbf{x}_i)||_F}$. The prediction for the n-th class for a given input $\hat{\mathbf{x}}$ is thus given by $\gamma \bar{\mathbf{c}}_n^T \mathbf{a}^{(L-1)}(\mathbf{x}) + b_n^{(L)}$, where we omitted the time step for $\mathbf{a}^{(L-1)}$ and b_n is a fixed bias in the output layer. Note that for $\gamma = -2$, the first term $(-2\bar{\mathbf{c}}_n^T \mathbf{a}^{(L-1)}(\mathbf{x}))$ resembles the first term in the prediction made by Prototypical network for class n, which is given by $-2\mathbf{c}_n^T \mathbf{a}^{(L-1)}(\mathbf{x})$, where $\mathbf{a}^{(L-1)}(\mathbf{x}) = f_\theta(\mathbf{x})$. Hence, OP-LSTM can learn to approximate (up to the bias term) a normalized Prototypical network classifier.

We have thus shown that OP-LSTM can learn to implement a parametric learning algorithm (gradient descent) as well as a non-parametric learning algorithm (prototype-based classifier), demonstrating the flexibility of the approach.

5.7 Conclusions

Meta-learning is a strategy to enable deep neural networks to learn from small amounts of data. The field has witnessed an increase in popularity in recent years, and many new techniques are being developed. However, the potential of some of the earlier techniques have not been studied thoroughly, despite promising initial results. In our work, we revisited the plain LSTM approach proposed by Hochreiter et al. (2001) and Younger et al. (2001). This approach simply ingests the training data for a given task, and conditions the predictions of new query inputs on the resulting hidden state.

We analysed this approach from a few-shot learning perspective and uncovered two potential issues for embedding a learning algorithm into the weights of the LSTM: 1) the hidden embeddings of the support set are not permutation invariant, and 2) the learning algorithm and the input embedding mechanism are intertwined, which leads to a challenging optimization problem and an increased risk of overfitting. In our work, we proposed to overcome issue 1) by mean pooling the embeddings of individual training examples, rendering the obtained embedding permutation invariant. We found that this method is highly effective and increased the performance of the plain LSTM on both few-shot sine wave regression and image classification. Moreover, with this first solution, the plain LSTM approach already outperformed the popular meta-learning method MAML (Finn et al., 2017) on the former problem. It struggled, however, to yield good performance on few-shot image classification problems, highlighting the difficulty of optimizing this approach.

In order to resolve this difficulty, we proposed a new technique, Outer Product LSTM (OP-LSTM), that uses an LSTM to update the weights of a base-learner network. By doing this, we effectively decouple the learning algorithm (the weight updates) from the input representation mechanism (the base-learner), solving issue 2), as done in previous works (Ravi and Larochelle, 2017; Andrychowicz et al., 2016). Compared with previous works, OP-LSTM does not receive gradients as inputs. Our

theoretical analysis shows that OP-LSTM is capable of performing an approximate form of gradient descent (as done in MAML (Finn et al., 2017)) as well as a nearest prototype based approach (as done in Prototypical network (Snell et al., 2017)), showing the flexibility and expressiveness of the method. Empirically, we found that OP-LSTM overcomes the optimization issues associated with the plain LSTM approach on few-shot image classification benchmarks, whilst using fewer parameters. It yields competitive or superior performance compared with MAML (Finn et al., 2017) and Prototypical network (Snell et al., 2017), both of which it can approximate.

Future work When the base-learner is a convolutional neural network, we applied OP-LSTM on top of the convolutional feature embeddings. A fruitful direction for future research would be to propose a more general form of OP-LSTM that can update also the convolutional layers. This would require new backward message passing protocols to go through pooling layers often encountered in convolutional neural networks.

Moreover, we note that OP-LSTM is one way to overcome the two issues associated with the plain LSTM approach, but other approaches could also be investigated. For example, one could try to implement a convLSTM (Shi et al., 2015) such that the LSTM can be applied directly to raw inputs, instead of only after the convolutional backbone in case of image classification problems.

Another fruitful direction for future work would be to investigate different recurrent neural architectures and their ability to perform meta-learning. In the pioneering work of Hochreiter et al. (2001) and Younger et al. (2001), it was shown that only LSTMs were successful whereas vanilla recurrent neural networks and Elman networks failed to meta-learn simple functions. It would be interesting to explore how architectural design choices influence the ability of recurrent networks to perform meta-learning.

Lastly, OP-LSTM is a method to learn the weight update rule for a base-learner network, and is thus orthogonal to many advances and new methods in the field of meta-learning, such as Warp-MAML (Flennerhag et al., 2020) and SAP (see Chapter 6). Since this is a nontrivial extension of these methods, we leave this for future work. We think that combining these methods could yield new state-of-the-art performance.

In this chapter, in a similar fashion to previous chapters, we aimed to gain an increased understanding of deep meta-learning starting from empirical observations. In the following chapter, we reverse this approach and investigate whether the integration of classical machine learning knowledge can improve the few-shot learning behavior of deep neural networks.

Chapter 6

Subspace Adaptation Prior for Few-Shot Learning

Chapter overview

Gradient-based meta-learning techniques aim to distill useful prior knowledge from a set of training tasks such that new tasks can be learned more efficiently with gradient descent. While these methods have achieved successes in various scenarios, they commonly adapt *all* parameters of trainable layers when learning new tasks. This neglects potentially more efficient learning strategies for a given task distribution and may be susceptible to overfitting, especially in few-shot learning where tasks must be learned from a limited number of examples. To address these issues, we propose *Subspace Adaptation Prior* (SAP)[1], a novel gradient-based meta-learning algorithm that jointly learns good initialization parameters (prior knowledge) and layer-wise *parameter subspaces* in the form of operation subsets that should be adaptable. In this way, SAP can learn which operation subsets to adjust with gradient descent based on the underlying task distribution, simultaneously decreasing the risk of overfitting when learning new tasks. We demonstrate that this ability is helpful as SAP yields superior or competitive performance in few-shot image classification settings (gains between 0.1% and 3.9% in accuracy). Analysis of the learned subspaces demonstrates that low-dimensional operations often yield high activation strengths, indicating that they may be important for achieving good few-shot learning performance. For reproducibility purposes, we publish all our research code publicly.

6.1 Introduction

In this chapter, we revisit the popular deep meta-learning technique model-agnostic meta-learning (MAML) (Finn et al., 2017) that learns a prior in the form of the initialization parameters of the neural network. Learning a new task is then done by performing gradient descent starting from this meta-learned initialization. This approach, which is also widely used by techniques that are based on MAML (Lee and Choi, 2018; Flennerhag et al., 2020; Park and Oliva, 2019b; Yoon et al., 2018;

[1]This chapter is based on the following research article: *Huisman, M., Plaat, A., & van Rijn, J. N. (2023). Subspace Adaptation Prior for Few-Shot Learning. Machine Learning. Springer.*

Nichol et al., 2018), updates all of the parameters of every *trainable* layer with gradient descent when learning new tasks, which may be suboptimal for a given task distribution and may lead to overfitting since there are more degrees of freedom to fit the noise in the data. Especially in few-shot learning, where tasks are noisy due to the fact that only limited examples are available, these issues could hinder performance.

To address these issues and investigate the research question of whether the few-shot learning performance of deep neural networks can be improved by meta-learning which subsets of parameters to adjust, we propose a new gradient-based meta-learning technique called *Subspace Adaptation Prior* (SAP) that jointly learns good initialization parameters as well as layer-wise subspaces in which to perform gradient descent when learning new tasks. More specifically, SAP is given access to a candidate pool of *operations* for every layer that transforms the hidden representations, and it learns which of these subsets to adjust in order to learn new tasks quickly, similar to DARTS (Liu et al., 2019). Here, every operation corresponds to a parameter subspace. Note that this method serves as a form of regularization and allows SAP to find more efficient adaptation strategies than adjusting all parameters of trainable layers. In addition, it utilizes implicit gradient modulation to warp (Lee and Choi, 2018; Flennerhag et al., 2020) these subspaces per layer such that gradient descent can quickly adapt to new tasks, if they share a common structure.

We empirically demonstrate that SAP is able to find efficient parameter subspaces, or operation subsets, that match the underlying task structure in simple synthetic settings and yield good few-shot learning. Moreover, SAP outperforms gradient-based meta-learning techniques—that do not have the ability to learn in which structured subspaces to perform gradient descent—on few-shot sine wave regression and performs on-par or favorably in various few-shot image classification settings. In short, the contributions of this chapter are the following:

- We propose SAP, a new meta-learning algorithm for few-shot learning that jointly learns good initialization parameters and parameter subspaces in the form of operation subsets in which to perform gradient descent.
- We demonstrate the advantage of learning parameter subspaces as SAP outperforms existing methods by at least 18% on few-shot sine wave regression and yields competitive or superior performance on popular few-shot image classification benchmarks (improvements in classification accuracy scores range from 0.1% to 3.9%).
- We investigate the learned layer-wise parameter subspaces on synthetic few-shot sine wave regression and image classification problems and find that small subsets of adjustable parameters (simple parameter subspaces), including feature transformations such as element-wise scaling and shifting are assigned large weights, suggesting that they play an important role in achieving good performance with SAP.
- For reproducibility and verifyability purposes, we make all our research code publicly available.[2]

6.2 Related work

We review related work on optimization-based meta-learning, neural architecture search and gradient modulation.

[2]See: `https://github.com/mikehuisman/subspace-adaptation-prior`

Optimization-based meta-learning Our proposed technique belongs to the category of optimization-based meta-learning (Vinyals, 2017) (also see Chapter 2), which employs optimization methods to learn new tasks (Yoon et al., 2018; Bertinetto et al., 2019; Lee et al., 2019). These methods aim to meta-learn good settings for various hyperparameters, such as the initialization parameters, such that new tasks can be learned quickly using optimization methods. These methods vary from regular stochastic gradient descent, as used in MAML (Finn et al., 2017) and Reptile (Nichol et al., 2018), to meta-learned procedures where a network updates the weights of a base-learner (Ravi and Larochelle, 2017; Andrychowicz et al., 2016; Li et al., 2017; Rusu et al., 2019; Li and Malik, 2018) (also see Chapter 3). SAP aims to learn good initialization parameters such that new tasks can be learned quickly with regular gradient descent, similar to MAML.

This is a form of transfer learning (Taylor and Stone, 2009; Pan and Yang, 2009) where we transfer knowledge—in this case the initialization parameters—obtained on a set of source tasks to a new target task that we are confronted with. The idea is also related to the idea of domain adaptation (DA) (Daumé III, 2009; Farahani et al., 2021), although in DA it is often assumed that we have a single task but two different data distributions (a source distribution and a target distribution). Note that in deep meta-learning (Hospedales et al., 2021) (also see Chapter 2), we have set of various different training tasks and aim to transfer knowledge to a new target task, different from the ones seen at training time.

Neural architecture search (NAS) for meta-learning The techniques mentioned above assume a pre-specified network architecture. Recently, there has been some work on combining meta-learning with neural architecture search, where the architecture can also be learned. Kim et al. (2018) performs meta-learning as a subroutine to NAS, meaning that meta-training is performed for every candidate architecture, which can be computationally expensive. This problem can be overcome by combining gradient-based meta-learning with gradient-based neural architecture search such that the architecture and initialization parameters can be optimized jointly instead of separately. A popular gradient-based meta-learning algorithm is DARTS (Liu et al., 2019) which starts with a candidate pool of operations (as in SAP) and learns which of them to use, thereby learning an appropriate architecture. Learning which subspaces or subsets of operations to use per layer, as done in SAP, can be seen as applying DARTS over the candidate operation sets. A difference between DARTS and SAP is that we fix the base-learner parameters when adapting to new tasks, which can then serve to warp, or transform, the gradients such that gradient descent can quickly move to a good solution for new tasks (see below). Moreover, SAP updates the initialization parameters of all meta-trainable parameters with a MAML-like update (to maximize post-adaptation performance), while DARTS uses a Reptile-like update (to maximize multi-step performance). We describe DARTS in full detail in Section 6.3.

Lian et al. (2019) were the first to combine DARTS (Liu et al., 2019) with gradient-based meta-learning in order to learn a base-learner architecture that can be quickly adapted to new tasks. They perform hard-pruning, which requires re-running the meta-training phase for every new task, which is computationally expensive. In parallel to this work, Elsken et al. (2020) proposed a similar approach (MetaNAS) that does not perform hard-pruning and thus side-steps these expensive re-running procedures. In contrast to these works, which learn and adapt the base-learner network architecture as well as all of the parameters to every new task, SAP assumes a fixed base-learner architecture as a starting point and aims to learn a set of operations that are inserted per layer (see Section 6.4) that are responsible for quickly adapting to new tasks. In SAP, the architecture of the network is frozen at test time, in contrast to, for example, the architecture of the networks learned by MetaNAS (Elsken et al., 2020).

Gradient modulation in gradient-based meta-learning Recent works that build upon MAML have shown that gradient modulation can improve the generalization of optimization-based techniques (Sun et al., 2019). Explicit gradient modulation techniques directly transform the gradient updates when learning new tasks (Simon et al., 2020) through, for example, diagonal matrix multiplication (Li et al., 2017), or block-diagonal preconditioning (Park and Oliva, 2019b). Implicit gradient modulation techniques do not directly operate on the gradients but rely on indirect transformations. CAVIA (Zintgraf et al., 2019) separates shared parameters from context parameters. The latter serve as additional inputs to one or more layers of the neural network and are adjusted when learning a new task, whilst the shared parameters are kept fixed. Other examples of implicit gradient modulation methods are T-Net (Lee and Choi, 2018) and Warp-MAML (Flennerhag et al., 2020). SAP also performs implicit gradient modulation in a similar fashion to these two techniques.

T-Net inserts linear projection transformations directly after every matrix multiplication in the base-learner. The weights of these transformations are frozen when learning new tasks, and only the base-learner weights are adjusted. The goal is to meta-learn good initialization parameters of the base-learner weights as well as the transformation weights, such that new tasks can be learned more quickly. These transformation layers serve to implicitly modulate the gradients of the base-learner parameters so that gradient descent can quickly move to good solutions for new tasks. MT-Net is an extension to T-Net, which also learns to mask certain features, preventing them from being adapted when learning new tasks. We also investigated whether feature masking was useful for SAP, but found that it decreased performance. Warp-MAML is a generalization of T-Net as it does not require that the inserted transformation layers are linear, that is, the theoretical framework allows these transformation layers to be non-linear and consist of multiple layers (arbitrary neural networks).

Both T-Net and Warp-MAML adjust all parameters of trainable layers, as is common in gradient-based meta-learning. However, this may be suboptimal for a given task distribution and lead to overfitting due to the large degree of freedom to fit the noise in the data. MT-Net, on the other hand, freezes certain features, which, in turn, also requires certain weights to be frozen but this is rather inflexible as that does not allow us to perform simple operations such as element-wise scaling of all features, which may be helpful for a given task distribution. To overcome these issues, we propose SAP, which learns per trainable layer which operations from a pre-defined candidate pool to use and adapt when learning new tasks, instead of resorting to regular matrix multiplications in which all weights are adjusted when learning new tasks (as done by other methods). While the expressivity of SAP is equivalent to T-Net and Warp-MAML (when using linear warp layers), the candidate pool of operations allows SAP to learn which operations are important for the given task distribution, thereby structuring the weight updates.

SAP is similar to T-Net and Warp-MAML in the sense that the linear base layers $\mathbf{W}^\ell$ (see Section 6.4) of the network in SAP can be seen as the warp layers or transformation layers that are used in T-Net and Warp-MAML, which act as implicit preconditioning layers that warp the loss surface to aid gradient descent in finding a good solution. Due to the similarities between T-Net, Warp-MAML, and SAP, they serve as excellent baselines to investigate whether the ability of SAP to learn which operation subsets to adapt when learning new tasks is helpful for few-shot learning. Concurrently to our work, Jiang et al. (2022) have proposed a subspace meta-learning algorithm. Whilst the title is similar, they explicitly meta-learn the bases for K subspaces. Then, when learning a new task, they aim to find linear combinations of the basis vectors of each of the subspaces that give rise to the best parameters for the given task in the subspaces. The subset containing the parameters with the lowest training loss is then used to obtain predictions for the query/test set. Note that their work is

different in that we do not learn basis vectors for different subspaces, but instead insert candidate operations that act to transform intermediate representations in the base-learner network to allow for faster learning and modulating the gradients.

6.3 Background on DARTS

In this section, we briefly describe DARTS (Liu et al., 2019), which is a gradient-based neural architecture search method that we build upon in this chapter. The goal of DARTS is to find a suitable neural architecture for a given problem. To do this, DARTS assumes a set of candidate operations that can be used to transform an input into an output. These candidate operations form a weighted graph as shown in Figure 6.1. In the figure, every node $o_i(\mathbf{x})$ corresponds to a candidate operation and the weights of the edges correspond to the activation strengths of the different operations. These weights are initially unknown and DARTS aims to learn them jointly with the initial parameters of every operation. The output of the layer in the figure is given by

$$\mathcal{O}(\mathbf{x}) = \sum_{i=1}^{n} w_i o_i(\mathbf{x}), \tag{6.1}$$

where w_i is the weight of operation i and $\sum_{i=1}^{n} w_i = 1$ (e.g., by using a softmax). For our purposes, we only consider DARTS for searching over operations for a single layer, but it can be used for multi-layer architectures as well.

In addition to learning the weights w_i, DARTS simultaneously learns good parameters for every operation o_i. We denote the group of all activation weights as $\lambda = \{w_1, w_2, \ldots, w_n\}$ and all operation parameters as θ. DARTS adopts a method similar to MAML for learning λ and θ. That is, given a training task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})$, DARTS performs a gradient update step on the operation weights θ as follows

$$\theta'_j = \theta - \alpha \nabla_{\theta,\lambda} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\theta, \lambda). \tag{6.2}$$

Note that this is similar to Equation 2.20 in Chapter 2 with the exception that we now have activation strength parameters λ, which are kept constant during this inner-loop adaptation step. After updating the operation parameters θ, DARTS computes the loss of the new model on the query set, i.e., $\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(\theta'_j, \lambda)$ and updates the activation strengths using gradient descent on this loss

$$\lambda = \lambda - \beta \nabla_{\lambda} \mathcal{L}_{D^{te}_{\mathcal{T}_j}}(\theta'_j, \lambda). \tag{6.3}$$

Similarly to MAML (see Chapter 2 and Chapter 3), this update also contains second-order gradients, but first-order approximations can be made. In DARTS, the weights of the operations θ are simply updated to their new values, that is, $\theta = \theta'_j$, i.e., after every task in the meta-train set, we update the initialization parameters θ to the parameters that were obtained after training on task $\mathcal{T}_j$.

6.4 Subspace Adaptation Prior

In this section, we motivate and present our proposed technique called *Subspace Adaptation Prior* (SAP).

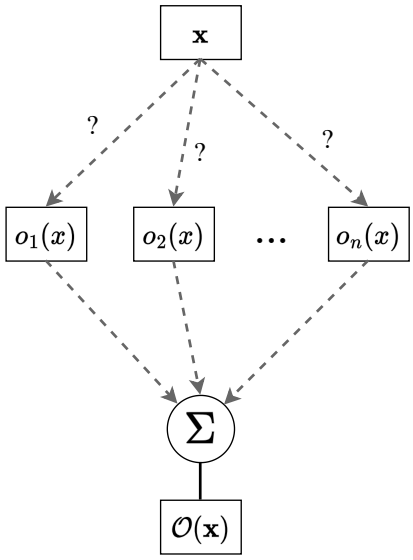
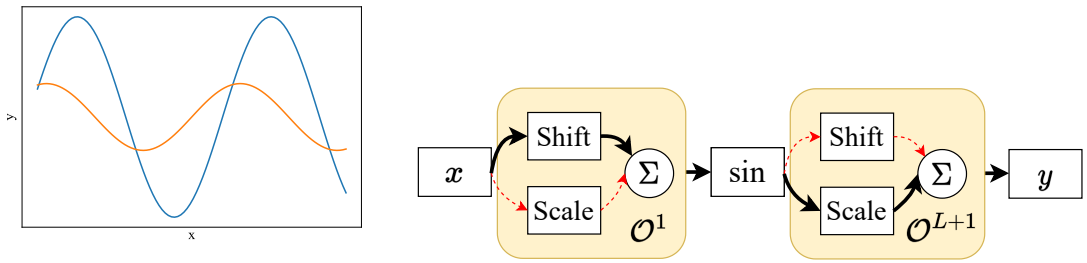


Figure 6.1: Intuitive visualization of DARTS. It is given a set of candidate operations $\mathcal{O}$ and aims to learn the weights of the edges (indicated as ?), corresponding to the strengths of the different operations $o_i(\mathbf{x})$. The output of the weighted graph is a convex combination of the different operations $\mathcal{O}(\mathbf{x}) = \sum_{i=1}^{n} w_i o_i(\mathbf{x})$.

6.4.1 Intuition and operations

Our method (SAP) builds on MAML as we also aim to learn good initialization parameters such that good performance can be achieved after a small number of gradient updates. However, MAML adapts all of its network parameters when presented with a new task, which may be suboptimal for the given task distribution and lead to overfitting. Our method, SAP, is given a pool of candidate operations per layer (described below) and it learns per layer which subset of operations should be adjusted to adapt to a new task. Since all of the operations that SAP can choose from per layer are subsumed in terms of expressivity by a full-rank matrix multiplication (or convolution in the case of image data), this can be understood as learning in which *parameter subspaces* to perform gradient descent so that new tasks can be learned more efficiently.

This is a form of regularization and can help the network to exploit structures in problems. For example, take the distribution of tasks $\mathcal{T}_j$ corresponding to different sine waves $g_j(x) = A_j \cdot \sin(x - p_j)$, where A_j is the amplitude and p_j the phase. There exists a common structure amongst these tasks: a given sine wave can be transformed into any other sine wave by simply shifting the input and scaling the output. This has been visualized in Figure 6.2. Techniques that adapt all parameters may overwrite the sine function and overfit to the noise, whereas theoretically, SAP could learn to keep these parameters fixed and that shifting the input and scaling the output are the most important operations and consequently, that gradient descent should be performed in the parameter subspaces corresponding to these operations. Sine waves form a simplistic example to demonstrate the idea of SAP, however, we note that also for image classification tasks, simple operations such as scaling and shifting feature maps can be useful too (Sun et al., 2019; Perez et al., 2018; Requeima et al., 2019). SAP can discover such underlying structures and use them to enhance its few-shot learning abilities.



(a) Sine wave tasks (b) Learned subspaces

Figure 6.2: SAP can learn the activation strengths of candidate operations $\mathcal{O}^\ell$ (corresponding to parameter subspaces) that match the problem structure. Suppose we are given a sine wave task distribution, where every task $\mathcal{T}_j$ is a sine wave $g_j(x) = A_j \cdot \sin(x - p_j)$, where p_j is the phase and A_j the amplitude. Instead of adapting all parameters of the network on a new task, SAP can learn to keep the sine network parameters (sin) frozen and that the input shift (shift in $\mathcal{O}^1$) and output scale (scale in $\mathcal{O}^{L+1}$) are the most important operations to adjust (bold and dark-colored arrows), matching the role of the phase and amplitude, respectively.

Candidate operations The candidate operations that SAP uses are specified by hand before meta-training. In order to preserve the original expressivity of the base-learner network, the operations are elementary linear algebra operations that are subsumed by full-rank matrix multiplication.

Table 6.1 displays all the operations that we use for both fully-connected and convolutional layers. The MTL scale operation was proposed by Sun et al. (2019). By construction, we require that the output of an operation set must have the same dimensionality as the input. Recall that in the case of fully-connected layers, all candidate operations can be expressed by a single matrix multiplication where only a subset of the entries is used. For example, an element-wise scale can be performed by multiplying the input with a diagonal matrix where the diagonal entries correspond to the element-wise scalars, and the non-diagonal entries are zero. In this way, every candidate operation occupies a *part* of the full operation set matrix. This also holds for convolutions, which can be seen as a stack of matrices.

We also include singular value (SVD) decomposition operations, where three v-rank matrices $A = U\Sigma V^T$ are multiplied to obtain a transformation matrix $A \in \gtrdot \times \ltimes$ with the same dimensionality as a full-rank transform $T \in \mathbb{R}^{m \times n}$ (although with a lower rank). Here, $U \in \mathbb{R}^{m \times v}$, $\Sigma \in \mathbb{R}^{v \times v}$, and $V^T \in \mathbb{R}^{v \times n}$. The obtained transformation A is then applied to the input.

Below, we describe how these operations are interleaved with the base-learner network and how SAP learns which subsets to adjust.

6.4.2 The algorithm

Architecture Let f_θ be a neural network with parameters θ, where the output, or prediction, is given by

$$f_\theta(\mathbf{x}) = \mathbf{W}^L \sigma(\dots \sigma(\mathbf{W}^2 \sigma(\mathbf{W}^1 \mathbf{x}))). \tag{6.4}$$

Fully-connected		Convolutional	
Operation	Dimensionality	Operation	Dimensionality
Identity	N.A.	Identity	N.A.
Matrix multiplication	$d \times d$	Convolution	$C \times C \times k \times k$
SVD-matrix multiplication	$d \times v$	SVD convolution	$C \times C \times k \times v$
Element-wise scale	d	1x1 convolution	$C \times C$
Scalar scale	1	MTL scale	$C \times C$
Vector shift	d	Channel-wise scale	C
Scalar shift	1	Channel-wise shift	C
		Scalar shift	1

Table 6.1: The candidate operations for fully-connected and convolutional network layers and the corresponding dimensionality of the subspace in which gradient will be performed. Here, d is the dimensionality of the input in the case of a fully-connected layer and C is the number of input and output dimensions of candidate operations in the case of convolutional layers. k is the kernel size of convolutions and $v < k$ is a variable dimension for SVD matrices.

Here, L is the number of layers of the network, σ is a non-linear activation function, and $\mathbf{W}^\ell$ is the weight matrix for layer $\ell \in \{1, 2, \ldots, L\}$ (which can also include the bias by concatenating a 1 at the top of the input vector). Note that $\theta = \{\mathbf{W}^1, \mathbf{W}^2, \ldots, \mathbf{W}^L\}$ is the set of all base-learner weight matrix parameters. In SAP, we insert sets of candidate operations $\mathcal{O}^\ell = \{o_1^\ell, \ldots, o_{n_\ell}^\ell\}$ before the application of weight matrices $\mathbf{W}^\ell$ and after computing the final output, as shown in Figure 6.3. Here, n_ℓ is the number of operations in the candidate set $\mathcal{O}^\ell$ in layer ℓ. Each of these operations $o_i^\ell \in \mathcal{O}^\ell$ act on the input, giving rise to partial outputs $o_i^\ell(\mathbf{z}^\ell)$ of the same dimensionality of the inputs, where $\mathbf{z}^\ell$ is the input to the ℓ-th operation layer. The final output of applying the candidate operations is a convex combination of the partial outputs, that is,

$$\mathcal{O}^\ell \mathbf{z}^\ell = \sum_{i=1}^{n_\ell} w_i^\ell o_i^\ell(\mathbf{z}^\ell), \tag{6.5}$$

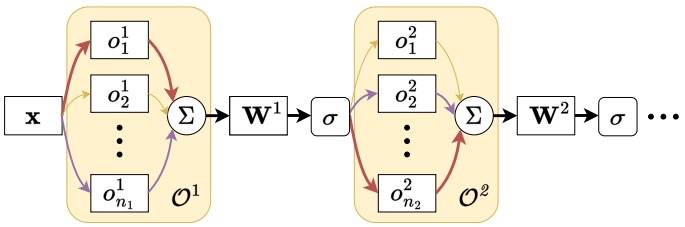


Figure 6.3: A diagram of a feed-forward pass in SAP. Sets of operations $\mathcal{O}^\ell$ are interleaved with base-learner weights $\mathbf{W}^\ell$. The operation sets perform a convex combination of a number of operations $\{o_1^\ell, \ldots, o_{n_\ell}^\ell\}$. SAP learns the strengths of each of the candidate operations and thereby learns in which parameter subspaces gradient descent can effectively adapt the network to learn new tasks. The operation strengths and the weight matrices $\mathbf{W}^\ell$ are frozen when adapting to new tasks. Only the operation parameters are adjusted at test time.

where $\mathbf{z}^1 = \mathbf{x}$ and w_i^ℓ is the activation strength of operation o_i^ℓ. We require that $\sum_{i=1}^{n_\ell} w_i^\ell = 1$ and $0 \leq w_i \leq 1$. Learning these activation strengths can be seen as neural architecture search. Thus, the output of the neural network in SAP is given by

$$f_{\mathbf{\Theta}}(\mathbf{x}) = \mathcal{O}^{L+1}\mathbf{W}^L\mathcal{O}^L\sigma(\dots\sigma(\mathbf{W}^2\mathcal{O}^2\sigma(\mathbf{W}^1\mathcal{O}^1\mathbf{x}))), \tag{6.6}$$

where $\mathbf{\Theta} = \{\theta, \phi, \lambda\}$ is the set of the initial hyperparameter values for the base-learner weights (θ), the operation weights (ϕ), and the activation weights (λ). Note that $\phi = \{\mathcal{O}^1, \mathcal{O}^2, \dots, \mathcal{O}^{L+1}\}$ are the parameters corresponding to the operations in all layers (see Section 6.4.1), and λ is the set containing all w_i^ℓ for all layers $\ell \in \{1, 2, \dots, L+1\}$.

Importantly, each of these candidate operations o_i^ℓ are *subsumed* or equivalent in terms of expressivity with full-rank matrix multiplication. For example, candidate operations can include element-wise shifting or multiplication of the input by a fixed scalar or by a vector, which can also be done by weight matrix multiplication. Since the application of a set of operations $\mathcal{O}^\ell$ of such expressivity can be seen as a single matrix multiplication (hence the suggestive notation), *the expressivity of an SAP network is equivalent to that of the original network.* To see this, note that the application of two weight matrices to an input can be written as the application of a single weight matrix to the input $\mathbf{x}$, that is, $\mathbf{W}(\mathcal{O}\mathbf{x}) = (\mathbf{W}\mathcal{O})\mathbf{x} = \mathbf{W}'\mathbf{x}$, where $\mathbf{W}$ and $\mathcal{O}$ are weight matrices, and $\mathbf{W}' = \mathbf{W}\mathcal{O}$. For the sake of another example, suppose that we have a set of two operations in $\mathcal{O}$: scalar multiplication $s \cdot \mathbf{z}$ and matrix multiplication $\mathbf{M}\mathbf{z}$ (preserving the dimensionality of $\mathbf{z}$). Furthermore, suppose that the two operations are applied with activation strengths w_1 and w_2, granting us the output $\mathbf{z}' = w_1 s \cdot \mathbf{z} + w_2\mathbf{M}\mathbf{z}$. We can rewrite this as $\mathbf{z}' = w_1 sI\mathbf{z} + w_2\mathbf{M}\mathbf{z} = (w_1 sI + w_2\mathbf{M})\mathbf{z} = \mathcal{O}\mathbf{z}$, where $\mathcal{O} = (w_1 sI + w_2\mathbf{M})$ and I is the identity matrix. For a more intuitive example, suppose that a base-layer is a fully-connected layer, and we add a fully-connected operation to alter the resulting representation, maintaining the original dimensionality. The composition of the two fully-connected layers is effectively linear and equally expressive as a single fully-connected layer. **Thus, introducing the operations used by SAP does not alter the expressivity of the original base-learner network.**

Crucially, this insight that we can write the weighted combination of different operations as a single weight matrix multiplication $\mathcal{O}\mathbf{x}$, where $\mathcal{O}$ is a weighted combination of different structured matrices, reveals that SAP effectively learns what subset of parameters of this weight matrix $\mathcal{O}$ and thus of $\mathbf{W}\mathcal{O}$ to adjust by learning the activation strengths λ. In this work, we use the expressions "learning which subsets of parameters to adjust" and "learning in what subspaces to perform gradient descent" synonymously.

Meta-learning The activation strengths w_i^ℓ are meta-learned by SAP in addition to the initialization parameters of the operations $\mathcal{O}^\ell$ and the base-learner weights $\mathbf{W}^\ell$. Note that learning the w_i^ℓ corresponds to learning in which parameter subspaces gradient descent is performed when learning new tasks, which can be done through the layer-wise application of the gradient-based neural architecture search technique DARTS (Liu et al., 2019). Let θ denote the initial parameters of the weight matrices $\mathbf{W}^\ell$, ϕ the parameters of all candidate operations $\mathcal{O}^\ell$, and λ the activation strengths w_i^ℓ of all individual candidate operation. Recall that $\mathbf{\Theta} = \{\theta, \phi, \lambda\}$.

When presented with a new task $\mathcal{T}_j$, the candidate operation activation strengths λ and the base-learner parameters θ are frozen, and only the candidate operation parameters ϕ are updated using

gradient descent for T steps

$$\phi_j^{(t+1)} \leftarrow \phi^{(t)} - \alpha \nabla_{\phi^{(t)}} \mathcal{L}_{\mathcal{T}_j}(\theta, \phi^{(t)}, \lambda), \tag{6.7}$$

where $\phi^{(0)}$ is initialized with ϕ. At the meta-level, the goal is to find good initial parameter settings for all involved parameters such that the task-specific performance is maximized. Thus, we wish to find

$$\underset{\boldsymbol{\Theta}=\{\theta,\phi,\lambda\}}{\arg\min} \underset{\mathcal{T}_j \backsim p(\mathcal{T})}{\mathbb{E}} \mathcal{L}_{\mathcal{T}_j}(\theta, \phi_j^{(T)}, \lambda), \tag{6.8}$$

where $\phi_j^{(T)}$ denotes the task-specific parameters obtained through one or more gradient update steps on task $\mathcal{T}_j$. In other words, we wish to find good initial values for the parameters θ, ϕ, and λ such that new tasks can be learned quickly by updating the operation parameters ϕ. This meta-objective can also be optimized through gradient descent by updating

$$\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \beta \nabla_{\boldsymbol{\Theta}} \sum_{\mathcal{T}_j \in \mathcal{B}} \mathcal{L}_{\mathcal{T}_j}(\theta, \phi_j^{(T)}, \lambda). \tag{6.9}$$

The full algorithm for application to few-shot learning is shown in Algorithm 12. At the start (line 1), we initialize the parameters of the base-learner θ randomly. The candidate operation parameters ϕ are initialized to leave the input unaffected (for example, scalars are initialized to 1 and biases to 0). The layer-wise activation strengths w_i^ℓ of the candidate operations are initialized to the uniform distribution. After this initialization, we repeat the following steps until a stopping criterion is met, such as having sampled a certain number of task batches, or observing decreasing performance on held-out validation tasks. We randomly sample batches of tasks (line 3), initialize the task-specific parameters $\phi^{(0)} = \phi$, and make T gradient update steps on the support set of every task (lines 6–8), and perform meta-updates to the initialization parameters $\boldsymbol{\Theta}$ (line 11) using the query sets of the tasks. Note that the meta-update requires the computation of second-order gradients as we have to compute the gradient of the inner-level gradients. The complexity of this is quadratic in the number of parameters, but can be avoided by using the first-order assumption $\nabla_\phi \phi_j^{(T)} = I$.

Algorithm 12 Subspace Adaptation Prior (SAP)

Require: $p(\mathcal{T})$
Require: α, β
1: initialize θ, ϕ, λ
2: **while** not converged **do**
3: sample batch of tasks $\mathcal{B} = \{\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{te}) \backsim p(\mathcal{T})\}_{j=1}^{M}$
4: **for** task $\mathcal{T}_j = (D_{\mathcal{T}_j}^{tr}, D_{\mathcal{T}_j}^{te}) \in \mathcal{B}$ **do**
5: initialize task-specific parameters $\phi_j^{(0)} = \phi$
6: **for** $t = 0, \ldots, T-1$ **do**
7: compute gradient update $\phi_j^{(t+1)}$ using Equation 6.7 on $D_{\mathcal{T}_j}^{tr}$
8: **end for**
9: **end for**
10: update initial parameters $\boldsymbol{\Theta} = \{\theta, \phi, \lambda\}$ using Equation 6.9
11: **end while**

Pruning The scores w_i^ℓ represent the *activation strengths* of the different candidate operations/subspaces, and can also be used for pruning the operations, for example, in a layer-wise or regular top-K fashion. By default, we do not hard-prune operations and maintain a convex combination of different candidate operations unless explicitly mentioned otherwise. Note that we cannot simply drop low activation strength operations from the network as that changes the composite features and layerwise activation statistics. Hard-pruning requires re-training the network with only the selected (non-pruned) subspaces/operations.

6.4.3 Analysis

One may wonder what the role is of inserting operation sets $\mathcal{O}^\ell$ in the base-learner network since they have the same expressivity as weight matrices. In other words, why do we have two consecutive matrix multiplications $\mathbf{W}^\ell\mathcal{O}^\ell\mathbf{x}$ if that is equivalent to having one matrix multiplication $\mathcal{U}\mathbf{x}$, where $\mathbf{U} = \mathbf{W}^\ell\mathcal{O}^\ell$. There are two reasons for maintaining two separate matrices, which we describe below.

Regularization First, having a set of operations $\mathcal{O}^\ell$ allows SAP to learn which sets, corresponding to weight subspaces of a full-rank matrix, are relevant for a given task distribution. Choosing lower-dimensional subspaces is a form of regularization, as fewer parameters can be adjusted to fit the noise in the data.

Gradient modulation Second, when computing gradient updates for the operation parameters ϕ^ℓ of a given layer ℓ, the frozen base-layers $\mathbf{W}^\ell$ implicitly modulate the gradients since the error signal traverses backward through $\mathbf{W}^\ell$ to $\mathcal{O}^\ell$. This method of gradient modulation was proposed by Lee and Choi (2018). Below, we borrow the analysis performed in that paper to illustrate the modulation.

Suppose we are presented with a task $\mathcal{T}_j$ and that the output for a given layer in the network is given $\mathbf{v} = \mathbf{W}\mathcal{O}\mathbf{x}$, where $\mathbf{x}$ is the input to the layer. Moreover, assume that the loss of the network on task $\mathcal{T}_j$ is given by $\mathcal{L}_{\mathcal{T}_j}$ Then, the parameters of the operations $\mathcal{O}$ are updated using a gradient update step, and we obtain the new output

$$\mathbf{v}^{new} = \mathbf{W}(\mathcal{O} - \alpha\nabla_{\mathcal{O}}\mathcal{L}_{\mathcal{T}_j})\mathbf{x} \tag{6.10}$$

$$= \mathbf{v} - \alpha(\mathbf{W}\nabla_{\mathcal{O}}\mathcal{L}_{\mathcal{T}_j})\mathbf{x}. \tag{6.11}$$

Note that we slightly abuse notation here since the parameters of the operations are denoted as ϕ. As we can see, the change in the layer's output $\Delta(\mathbf{v}^{new}, \mathbf{v})$ is negatively proportional to the $(\mathbf{W}\nabla_{\mathcal{O}}\mathcal{L}_{\mathcal{T}_j})$. Here, $\mathbf{W}$ *warps* the gradients with respect to the operation parameters. The warping of these gradients is meta-learned across tasks such that within a few gradient updates in warped space, a good performance can be achieved (Flennerhag et al., 2020; Lee and Choi, 2018; Park and Oliva, 2019b).

As a consequence, *SAP can learn both in which parameter subspaces to perform gradient descent by learning appropriate subsets of operations, as well as learn how to warp these subspaces so that few gradient updates yield good performance.*

6.5 Experiments

In this section, we aim to answer the following research questions:

- Does learning suitable layer-wise operations/subspaces improve meta-learning performance on sine wave regression? (Section 6.5.1)
- Do the learned strengths of subspaces/operations match the task structure in a simple synthetic setting? (Section 6.5.2, Section 6.5.3)
- How well does SAP perform at few-shot image classification? (Section 6.5.4)
- How well does SAP perform at cross-domain few-shot image classification? (Section 6.5.5)
- Is hard subspace pruning beneficial for the performance of SAP? (Section 6.5.6)
- What is the influence of second-order gradients on the performance of SAP? (Section 6.5.7)
- What operations are important for few-shot image classification? (Section 6.5.8)
- How does SAP compare in terms of the running time and and number of trainable parameters compared to the baselines? (Section 6.5.9)

6.5.1 Sine wave regression

First, we study the few-shot learning performance of SAP on few-shot sine wave regression, which is commonly used in the meta-learning community (Finn et al., 2017; Li et al., 2017; Park and Oliva, 2019b). Here, the goal is to learn sine wave regression tasks $\mathcal{T}_j$ corresponding to sine curves $g_j(x) = A_j \cdot \sin(x - p_j)$ from a limited set of k examples. The amplitudes A_j and phases p_j of these sine curves are randomly sampled from the intervals $[0.1, 5.0]$ and $[0, \pi]$, respectively. While the results on sine-wave regression are not our main contribution, the structure of these problems were a motivation for the development of this method, and therefore this is a good test-case on which we expect SAP to perform well. Of course, SAP can only be considered a valuable contribution when it also works on more relevant problem types, which we explore in the following sections.

We use the same base-learner architecture, a fully-connected neural network with 2 hidden ReLU layers of 40 nodes each, as in (Finn et al., 2017). For the SVD operations (see candidate operations in Section 6.4.2), we use ranks 5, 10, and 15 in the candidate pools. All candidate operations were initialized to have no effect on the network predictions at the start (transformation matrices were initialized to identity matrices, biases to 0, and scale operations to 1). All techniques are meta-trained on 70 000 tasks using one update step per task and a meta-batch size of 4. We perform validation every 2 500 tasks to select the best performing model, which will be tested after 1 and 10 gradient update steps on 2 000 meta-test tasks consisting of k support examples and 50 query data points.

As baselines, we compare against MAML, T-Net, and MT-Net (Lee and Choi, 2018) as well as Warp-MAML (Flennerhag et al., 2020) as are highly similar to SAP, which allows us to investigate the advantage of SAP's ability to learn which subsets of operations to adjust. We refrain from comparing against MetaNAS (Elsken et al., 2020), as this technique also adjusts the architecture at meta-test time and is orthogonal to SAP and the methods we compare against. For all methods, we use the same hyperparameters as reported in (Finn et al., 2017; Lee and Choi, 2018). In this case, however, Warp-MAML is equivalent to T-Net as both use insert linear "transformation" or "warp" layers in the base-learner network. The results of the experiments are displayed in Table 6.2. In this

		5-shot		10-shot	
	params	T=1	T=10	T=1	T=10
MAML	1 761	0.73 ± 0.016	0.42 ± 0.011	0.49 ± 0.011	0.15 ± 0.005
T-Net	4 962	0.53 ± 0.014	0.24 ± 0.009	0.33 ± 0.009	0.09 ± 0.004
MT-Net	5 043	0.55 ± 0.013	0.19 ± 0.005	0.34 ± 0.008	0.06 ± 0.002
SAP (ours)	10 013	**0.47** ± 0.012	**0.10** ± 0.003	**0.28** ± 0.008	**0.04** ± 0.001

Table 6.2: The mean MSE meta-test loss on 5- and 10-shot sine wave regression after $T = 1$ and $T = 10$ update steps. The results are averaged over 5 runs with different random seeds and the 95% confidence intervals are displayed as ± x. The number of parameters is shown in the column "params", even though the used backbones are equally expressive.

table, we can see that SAP consistently outperforms all tested baselines, supporting the hypothesis that it is indeed beneficial to learn in which subspaces to perform gradient descent. We have also performed experiments with SAP and the feature masking method used in MT-Net, where some features are frozen based on learned feature masking probabilities, but found that it decreases the performance, which may be due to the low-dimensional operations present in the architecture, which are more susceptible to being completely frozen as soon as a single feature is masked.

6.5.2 The learned subspaces for sine regression

Next, we investigate (in the same setting as above) the importance of the different candidate operations for quick adaptation to new tasks to see whether the operations match the task structure. We hypothesize that shifting the input and scaling the output are important operations as they are inherent in the definition of a sine wave $g_j(x) = A_j \cdot \sin(x - p_j)$. To investigate this, we inspect the activation strengths w_i^ℓ of the operations of the best models across 5 different runs with different random seeds. The operations that were used are were introduced in Table 6.1 (left side). The results for SAP with $T = 1$ are displayed in Figure 6.4 (similar results are obtained when making $T = 10$ updates and therefore omitted for brevity). As we can see, the most important transformations on the input and output are a scalar shift and multiplication, respectively. In other words, SAP has learned that shifting the input and scaling the output are effective operations to learn new tasks. Note that these operations match the structure of sine waves. While this confirms our hypothesis, SAP also assigns relatively large importance to operations that are not directly observable in the mathematical definition of sine curves such as an output shift and intermediate shifts.

6.5.3 Matching the problem structure

To further investigate the ability of SAP to match the learned candidate operation strengths to the structure of the problem, we investigate whether changes in the problem structure amount to changes in the learned activation strengths by SAP for the different operations. For this, we consider a synthetic sine wave regression problem that generalizes the settings studied by Finn et al. (2017) and Li et al. (2017). In this setting, we create different *task families* (task distributions) that are characterized by the mathematical operations inherent in the ground-truth function. All task families share the following template for the ground-truth function $g(x) = A \cdot \sin(f \cdot x - p) + \beta$, where A is the amplitude, f the frequency, p the phase, and β the output offset. What distinguishes task families is which of these parameters they include in the functional description. For example, task family A

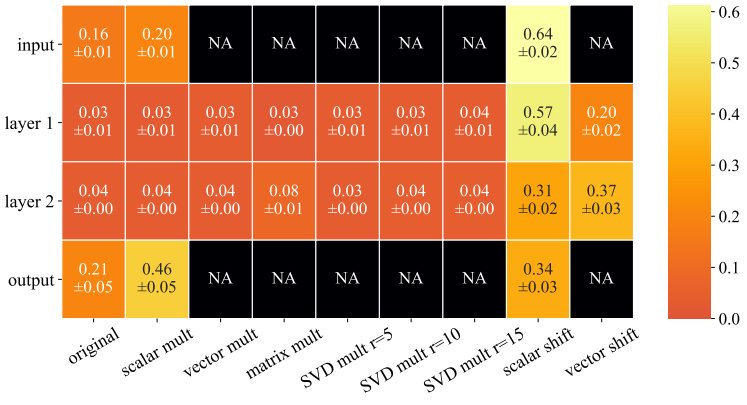


Figure 6.4: The importance of the different operations in SAP for 5-shot sine wave regression. The results are averaged across 5 runs with different random seeds and the standard deviations are shown as $\pm x$. NA entries indicate that these operations were not in the candidate pool for that layer, and "mul" means multiplication. The y-axis indicates the layer on which the operations act, and the x-axis displays the different candidate operations. Simple scalar multiplication and shifting, and vector shifts obtain high activation strengths in all layers. The input shift and output scale (inherently present in the definition of a sine wave) obtain high activation strengths.

may fix the amplitude and vary the frequency, phase, and output offset, whereas task family A may vary the amplitude and fix the rest. Each task family is thus defined by which of these parameters are varied among tasks from that family and which are kept constant. If a given parameter is not varied, we fix it to a value that leaves the function unaltered (i.e., $A, f, p = 1$ and $\beta = 0$). In total, there are $2^4 = 16$ task families that can be constructed by varying or fixing these parameters.

We perform meta-training on each of these task families separately and investigate whether SAP discovers the operations that are inherently present in the task structure. The experimental details follow those used in Section 6.5.1 with the exception that only operations were included that could be present in the task families to be able to measure whether SAP correctly detects and uses them. We use 20-shots per task and set the number of inner updates to $T = 1$. The results of this experiment are displayed in Figure 6.5. As we can see, SAP assigns higher activation strengths to operations that are inherently present in the task families in three out of four cases, i.e., input scale (frequency), input shift (phase), and output shift. A statistical T-test shows that these differences in mean activation strengths are statistically significant, using a threshold of 0.05. For the input scale, however, we observe that SAP assigns similar activation strength to the input scale activation, regardless of whether such an operation was present in the task family. This may indicate that SAP uses other operations to compensate for this, such as vector multiplications or matrix multiplications in later layers. Overall, these results suggest in this simple synthetic setting, SAP is capable of learning to use operations that appear in the problem structure in 75% of the scenarios.

6.5.4 Few-shot image classification

Next, we investigate the performance of SAP in few-shot image classification settings, where the goal is to learn new image classification tasks from a few examples. For this, we use the popular

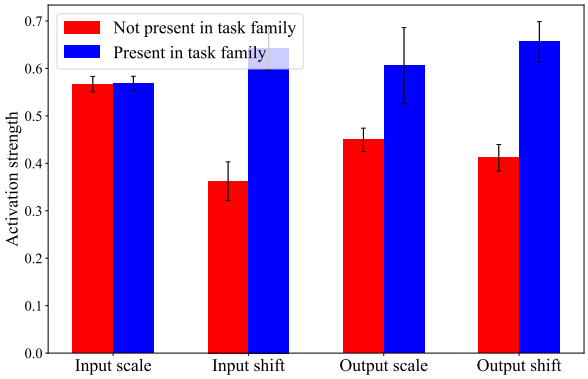


Figure 6.5: The mean activation strengths of the different operations corresponding to the intrinsic parameters that were varied within task families. The vertical bars display 95% confidence intervals over 5 runs with different random seeds. Each task family has the following template $g(x) = A \cdot \sin(f \cdot x - p) + \beta$ and differs in which of these operations are varied among tasks. If operations are inherently present in a task family, SAP assigns higher activation strengths to them than if they are not present in 3 out of 4 cases, indicating that the operations often match the problem structure.

Hyperparameter	Range
Inner learning rate	LogUniform(1e-3, 6e-1)
Inner update steps (training)	Uniform(1,10)
Inner update steps (testing)	Uniform(inner steps training, 15)
Meta-batch size	Uniform(1,10)
Gradient masking	Uniform({False, True})

Table 6.3: The ranges and sampling types for the hyperparameters, which were tuned with random search. The bounds are inclusive.

N-way k-shot classification setup (see Chapter 2) on miniImageNet (Vinyals et al., 2016; Ravi and Larochelle, 2017) and tieredImageNet (Ren et al., 2018). We use the frequently used Conv-4 backbone (Finn et al., 2017; Lee and Choi, 2018; Flennerhag et al., 2020), consisting of four blocks, where each block contains 3×3 convolutions, a max pooling layer, 2D BatchNorm, and a ReLU nonlinearity. In the literature, this backbone has been used with 64 channels for every convolutional block (Snell et al., 2017; Vinyals et al., 2016) as well as 32 channels (Finn et al., 2017; Nichol et al., 2018). For this reason, we present the results for SAP on both variants. The final feature representations are flattened and fed into a softmax output layer. All techniques were trained for 60 000 episodes and were validated after every 2 500 tasks and we use the best-reported hyperparameters by the original authors.

We tuned a subset of the hyperparameters for SAP on the *meta-validation* tasks using random search with a function evaluation budget of 30 runs. Each run was restricted to finish within 7 days on a single PNY GeForce RTX 2080TI GPU. Runs that took longer (e.g., because of a large meta-batch size) were discarded from the hyperparameter search. The used hyperparameter ranges and sampling

types that were used for the random search are displayed in Table 6.3. Due to computational constraints, we adopted the best reported hyperparameters of the baseline methods as reported in their respective papers. As such, the comparison against these baselines is in this experiment only for illustrative purposes, as the hyperparameter optimization procedure on these methods has not been executed under the same conditions.

	1-shot		5-shot	
	32 channels	64 channels	32 channels	64 channels
MAML	48.0 ± 0.8	46.7 ± 0.8	64.4 ± 0.4	63.6 ± 0.4
T-Net	48.9 ± 0.8	48.7 ± 0.8	65.3 ± 0.4	-
MT-Net	48.5 ± 0.8	49.3 ± 0.8	63.0 ± 0.4	-
Warp-MAML	49.5 ± 0.8	49.8 ± 0.8	63.9 ± 0.4	64.6 ± 0.4
SAP (ours)	**51.6** ± 0.8	**52.8** ± 0.8	**65.9** ± 0.4	**67.4** ± 0.4

Table 6.4: Meta-test accuracy scores on 5-way miniImageNet classification over 5 runs with two variants of the Conv-4 backbone, that is, with 32 or 64 channels per block. The 95% confidence intervals are displayed as ± x. "-" indicates that the experiments required more GPU VRAM than available.

The results for the experiments on 5-way miniImageNet and tieredImageNet classification are displayed in Table 6.4 and Table 6.5. Note that the results for 5-shot T-Net and MT-Net are missing as they were unable to run on our GPU with 12GB of VRAM. As we can see, the performance of the techniques improves when using 64 channels compared with 32, with the exception of MAML on miniImageNet and T-Net in the 1-shot setting on miniImageNet. As we can see, SAP consistently outperforms all tested baselines in all tested settings (with gains between 1.1% to 3.3% accuracy), indicating that it is beneficial to learn subsets of operations on which gradient descent is performed in the case of few-shot image classification.

	1-shot		5-shot	
	32 channels	64 channels	32 channels	64 channels
MAML	50.7 ± 0.8	51.5 ± 0.8	65.2 ± 0.4	66.6 ± 0.4
T-Net	49.4 ± 0.8	51.7 ± 0.8	64.6 ± 0.4	-
MT-Net	49.8 ± 0.9	51.5 ± 0.8	64.6 ± 0.4	-
Warp-MAML	51.8 ± 0.8	53.3 ± 0.8	66.0 ± 0.4	68.2 ± 0.4
SAP (ours)	**52.9** ± 0.8	**54.5** ± 0.8	**69.3** ± 0.3	**71.3** ± 0.4

Table 6.5: Meta-test accuracy scores on 5-way tieredImageNet classification over 5 runs with two variants of the Conv-4 backbone, that is, with 32 or 64 channels per block. The 95% confidence intervals are displayed as ± x. "-" indicates that the experiments required more GPU VRAM than available.

6.5.5 Cross-domain few-shot image classification

Next, we study the performance of SAP in a more challenging *cross-domain* few-shot image classification setting. In this setting, techniques are trained on tasks from dataset A and evaluated on tasks

from another dataset B, in contrast to the setting used above, where the techniques were evaluated on *unseen* tasks from the same dataset used for training. We use the same setting as Chen et al. (2019), in which we train on miniImageNet and evaluate on CUB (Wah et al., 2011). In addition, we also train on tieredImageNet (Ren et al., 2018) and test on CUB. All other experimental details are the same as above.

The results of this experiment are shown in Table 6.6. As we can see, SAP performs on par with Warp-MAML in the 1-shot setting for MIN → CUB. Both outperform the other tested baselines in that scenario. In other cases, however, SAP yields performance improvements ranging from 0.5% to 3.9% accuracy. This supports the hypothesis that it is beneficial to learn which subsets of operations to adjust when learning new tasks.

	MIN → CUB		Tiered → CUB	
	1-shot	5-shot	1-shot	5-shot
MAML	37.3 ± 0.3	54.7 ± 0.3	38.1 ± 0.3	55.1 ± 0.3
T-Net	38.0 ± 0.3	55.6 ± 0.3	37.5 ± 0.3	54.8 ± 0.3
MT-Net	37.1 ± 0.3	53.1 ± 0.3	38.0 ± 0.3	55.5 ± 0.3
Warp-MAML	**41.0** ± 0.3	55.3 ± 0.3	40.9 ± 0.3	56.8 ± 0.3
SAP (ours)	40.9 ± 0.3	**55.8** ± 0.3	**41.1** ± 0.3	**60.7** ± 0.3

Table 6.6: Average cross-domain meta-test accuracy scores over 5 runs a 32-channel Conv-4 backbone. Techniques trained on tasks from one data set were evaluated on tasks from another data set. The 95% confidence intervals are displayed as ± x.

6.5.6 Effect of hard pruning

Next, we investigate the effect of hard pruning the number of operations per layer, which is a common feature of DARTS (Liu et al., 2019), and therefore also inherited by SAP. For this, we compare the performance of SAP without hard pruning and SAP where we only retain the top-K operations as indicated by their strength scores. The hard-pruned SAP is re-trained using only the candidate operations which were not pruned. The results of this experiment with a 32-channel Conv-4 backbone are displayed in Table 6.7 (for the 64-channel variant, please see Table 5.4 in the appendix). As we can see, hard pruning can have a mild positive effect on the meta-learning performance, whilst reducing computational costs due to the fact that fewer parameters have to be trained. This also implies that some operations may indeed be suboptimal for a given task distribution, which soft-pruning is not able to completely filter out, and that a model which fully excludes these, can achieve better performance. We note, however, that the 95% confidence intervals are overlapping, suggesting that these performance increases are not significant.

6.5.7 The effect of the gradient order

All tested techniques require the computation of second-order gradients by default. Here, we investigate how the performance of SAP is affected by making a first-order approximation. We compare this first-order variant with the regular second-order variant, using the same experimental settings as used in Section 6.5.4. The results of this experiment are shown in Table 6.8. As we can see, the first-order approximation is consistently outperformed by the regular variant, with differences between 0.2% and 7.3 % accuracy, indicating that second-order gradients play an important role in

	miniImageNet		tieredImageNet	
	1-shot	5-shot	1-shot	5-shot
No pruning	51.6 ± 0.8	65.9 ± 0.4	52.9 ± 0.8	69.3 ± 0.3
Top-1	51.4 ± 0.8	65.8 ± 0.4	52.8 ± 0.8	69.4 ± 0.4
Top-2	**51.8** ± 0.8	**66.3** ± 0.4	**53.4** ± 0.8	69.4 ± 0.4
Top-3	**51.8** ± 0.8	**66.3** ± 0.4	53.0 ± 0.9	69.9 ± 0.4

Table 6.7: Mean meta-test accuracy scores on miniImageNet and tieredImageNet with 95% confidence intervals over 5 different runs. We used a Conv-4 backbone with 32 channels for these results.

achieving good performance.

	miniImageNet		tieredImageNet	
	1-shot	5-shot	1-shot	5-shot
SAP (first-order)	51.4 ± 0.8	63.7 ± 0.4	47.2 ± 0.8	62.0 ± 0.4
SAP (second-order)	**51.6** ± 0.8	**65.9** ± 0.4	**52.9** ± 0.8	**69.3** ± 0.3

Table 6.8: Meta-test accuracy scores on miniImageNet and tieredImageNet classification over 5 runs using the Conv-4 backbone with 32 channels. The 95% confidence intervals are displayed as ± x.

6.5.8 The learned subspaces for image classification

In order to gain insight into what operations are important for achieving good few-shot learning performance in SAP, we investigate the learned activation strengths for the different candidate operations. The operations that were used are were introduced in Table 6.1 (right side). In Figure 6.6, we can see these learned strengths in SAP on 1-shot 5-way miniImageNet using the Conv-4 backbone with 32 channels (similar patterns are seen for the backbone with 64 channels as can be seen in Figure C.1 in the appendix). As we can see, high-dimensional convolutional operations (conv1x1, conv3x3, convSVD) obtain low activation strengths, while lower-dimensional subspaces/operations such as shifts (scalar and vector) and MTL scale yield larger strengths. The greatest strength is assigned to the former throughout all layers. This may indicate that the higher-dimensional operations lead to overfitting, while the lower-dimensional operations are more suited for adapting to tasks when only limited data is available. Consequently, this implies that it is indeed beneficial to adapt subsets of operations when learning new tasks.

6.5.9 Number of parameters and running time

Lastly, we compare the running times and the number of parameters used by the different methods on few-shot image classification. These statistics were measured whilst performing the experiments in Section 6.5.4 and the results are displayed in Table 6.9. As we can see, SAP has the largest number of parameters, even though the backbone is equally expressive as that used by others. The running time of SAP, however, is often less than that of the baselines. This is caused by the fact that all methods use different hyperparameter settings in order to optimize the performance, which relates to the running time. For example, a larger meta-batch size or number of updates per task leads to an increase in running time. SAP uses the smallest meta-batch size and number of updates and

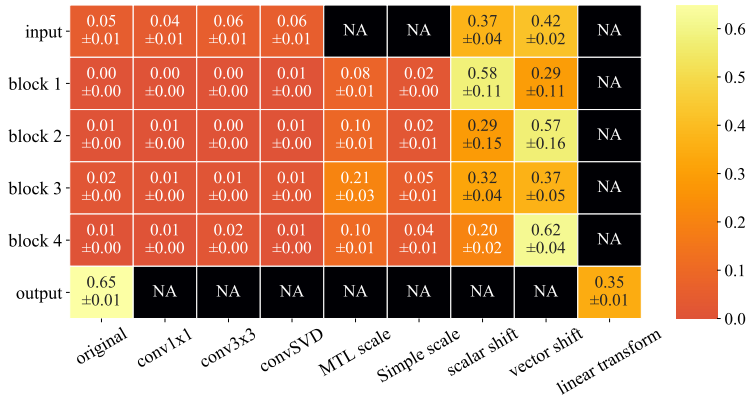


Figure 6.6: The importance of the different subspaces/operations in SAP on 5-way 1-shot miniImageNet using Conv-4 with 32 channels. The results are averaged across 5 runs with different random seeds and the standard deviations are shown as $\pm x$. NA entries indicate that these operations were not in the candidate pool for that layer. Simple scalar shift and vector shift operations obtain the highest activation strengths throughout the convolutional network.

hence yields the quickest running time. Note that these runtimes do not include the hyperparameter optimization that was performed, which adds a factor to the runtimes.

		miniImageNet		tieredImageNet	
	params	1-shot	5-shot	1-shot	5-shot
MAML	32 901	11h36min ± 7min	8h20min ± 1min	11h34min ± 5min	8h25min ± 4min
T-Net	37 022	33h05min ± 19min	30h20min ± 7min	33h25min ± 23min	30h30min ± 17min
MT-Net	37 150	33h33min ± 8min	30h22min ± 14min	33h49min ± 40min	30h47min ± 18min
Warp-MAML	60 645	7h19min ± 6min	7h17min ± 8min	7h35min ± 13min	7h19min ± 5min
SAP (first-order)	106 196	1h26min ± 2min	1h4min ± 0min	1h51min ± 6min	2h12min ± 4min
SAP	106 196	3h59min ± 0min	6h09min ± 0min	1h51min ± 6min	9h24min ± 19min

Table 6.9: The number of trainable parameters ("params") and mean running times on miniImageNet and tieredImageNet classification over 5 runs using the Conv-4 backbone with 32 channels. The standard deviations are displayed as ± x min. In spite of the differences in the number of parameters, the backbones are equally expressive. SAP was found to work best with a small meta-batch size and number of updates per task compared with the other approaches and hence yields the quickest running time.

6.6 Conclusions

In this chapter, we introduced, *Subspace Adaptation Prior* (SAP), a novel meta-learning algorithm that jointly learns a good neural network initialization and good parameter subspaces (or subsets of operations) in which new tasks can be learned within a few gradient descent updates from a few data. SAP overcomes the limitations of current state-of-the-art gradient-based meta-learning techniques

which perform gradient descent in *full parameter space* as they adjust all parameters (Finn et al., 2017; Lee and Choi, 2018; Flennerhag et al., 2020), which may be suboptimal, and may lead to overfitting during few-shot learning. Note, however, that our goal is not to yield state-of-the-art performance. Instead, we investigate the question of whether the few-shot learning performance of deep neural networks can be improved by meta-learning which subsets of parameters to adjust.

Our experiments show that SAP outperforms similar existing gradient-based meta-learners in few-shot sine wave regression, yields better performance in single-domain few-shot image classification settings, and yields competitive or superior performance in cross-domain few-shot image classification. This highlights the advantage of learning suitable subspaces in which to perform gradient descent when learning new tasks. This could be due to the regularization effect of not having to adjust all parameters as well as due to the ability to match structures inherently present in task families. Our experiments in Section 6.5.3 on synthetic task families demonstrate that the SAP is able to learn operations that match the task structure in simple settings in 75% of the cases. In other cases, it may compensate by using other operations that are not inherently present in the task structure.

Inspection of the subspace activation strengths in few-shot image classification reveals that simple and low-dimensional operations, such as shifting features by a single scalar or element-wise by a vector, are important. This is in line with recent work and findings (Triantafillou et al., 2021; Requeima et al., 2019; Bateni et al., 2020) which show that adapting pre-trained embeddings by means of such low-dimensional transformations, such as FiLM layers (Perez et al., 2018), can yield excellent performance. Furthermore, we found that hard-pruning the subspaces in SAP, or operations, such that only a discrete subset is used instead of a convex combination, was slightly beneficial, although no statistically significant differences were found.

Future work One limitation of SAP is that it requires the computation of second-order gradients by default during meta-training in order to update the initialization parameters, in a similar fashion as other gradient-based meta-learners such as MAML (Finn et al., 2017), (M)T-Net (Lee and Choi, 2018), and Warp-MAML (Flennerhag et al., 2020). These second-order gradients require $O(N^2)$ storage, where N is the number of total network parameters, which is prohibitive for deep networks. This limitation can be bypassed by using a first-order approximation, which comes at the cost of a performance penalty (between 0.2% and 7.3% accuracy in our experiments).

Gradient-based meta-learning methods struggle to scale well to deep networks as recent work suggests that simple pre-training and fine-tuning of the output layer (Tian et al., 2020; Chen et al., 2021) (also see Chapter 4) can yield superior performance on common few-shot image classification benchmarks. This is also the reason, besides searching for energy-efficient few-shot learners, that in our experiments we focus on relatively shallow backbones that adapt all layers when learning new tasks, instead of only the output layer.

Other limitations are that SAP introduces more parameters and that the candidate pools of operations are selected by hand, despite the fact that these operations are general. One direction for future work could be to design a method to discover such subspaces from scratch, instead of relying on a candidate set of operations, perhaps using an auto-encoder that generates the weights of a layer based on latent codes as used by Rusu et al. (2019). Masking the adaptation of these latent codes using Gumbel-softmax (Jang et al., 2017; Maddison et al., 2017) as done by MT-Net (Lee and Choi, 2018) would amount to adjusting only a subset of the parameters when performing gradient descent. This can reduce the number of parameters and may also help to scale gradient-based meta-learners, including SAP, to deep networks and make them competitive with approaches relying on pre-trained features, which is an open challenge.

Finally, orthogonal work has proposed a method that can also adjust the architecture during the meta-test phase (Elsken et al., 2020). Since this showed great potential, it would be worthwhile to combine this with SAP. Moreover, it would be interesting to investigate the sensitivity of SAP related methods such as MetaNAS to the chosen operations or blocks that these methods can select to use. We leave these ideas for future work, which has the potential to further advance the state-of-the-art.

This chapter is the final chapter that presents research results. In the next chapter, we take a step back and return to the big picture, revisiting and answering our original research questions, and proposing directions for future work.

Chapter 7

Conclusions

Chapter overview

In this dissertation, we have investigated the field of deep meta-learning and focused on addressing several research questions related to the performance and understanding of deep meta-learning algorithms with the aim of extracting knowledge about deep meta-learning algorithms beyond simple performance comparisons (method A outperforms method B). In addition, we have investigated how theoretical principles can be used to improve the performance of deep meta-learning algorithms. In this chapter, we revisit and answer the research questions listed in Chapter 1, discuss the implications and limitations of our work, and propose several fruitful directions for future research.

7.1 Answers to research questions

RQ1: What causes the performance gap between MAML and the meta-learner LSTM?

For the investigation of this research question, we have proposed a simplified version of the meta-learner LSTM called TURTLE, by using a feedforward neural network instead of an LSTM and raw inputs rather than preprocessed inputs. When using a first-order approximation of the meta-gradients in a similar fashion to the original meta-learner LSTM, TURTLE performs on par with the meta-learner LSTM. When using second-order gradients, however, TURTLE systematically yields better performance. Moreover, we have shown that by enhancing the meta-learner LSTM with the same inputs as TURTLE and second-order gradients, the observed performance gap to MAML was closed. This shows that the answer to this research question is the fact that the original meta-learner LSTM used processed inputs and a first-order approximation of the meta-gradients to perform meta-updates rather than the exact meta-gradients, which contain second-order effects.

Interestingly, the effect of making first-order approximations is different for MAML compared with the meta-learner LSTM and TURTLE. For MAML, it was shown by Finn et al. (2017) that using first-order approximations of the meta-gradients yields comparable performance compared with the exact second-order gradients. For the other two methods, however, this is not the case, suggesting that learning a weight update procedure (as done by TURTLE and the meta-learner LSTM) introduces

more curvature into the meta-loss landscape. Nonetheless, learning a weight update procedure can be beneficial for performance, as also noted by Andrychowicz et al. (2016).

Lastly, we note that whilst we have managed to close the observed performance gap by using second-order gradients, this comes at the cost of increased computational cost as there are $O(N^2)$ second-order gradients to be computed, where N is the number of parameters of the neural network. This can be intractable for larger neural networks with millions of parameters.

RQ2: How do the learning behaviors of finetuning, MAML, and Reptile differ from each other and how does this influence their ability to quickly learn to perform new tasks?

These techniques have distinct optimization objectives and exhibit varying behaviors, which in turn affect their ability to learn new tasks quickly.

In the research conducted in Chapter 4, the optimization objectives of these techniques were interpreted as maximizing different performance aspects. Finetuning aims to maximize direct performance with the current set of parameters. It focuses on improving performance without explicitly considering future adaptation steps. On the other hand, MAML aims to maximize performance after a few adaptation steps, making it a look-ahead objective. It searches for an initialization that may not yield the best immediate performance but leads to promising results after a few gradient update steps. Reptile combines both direct performance and performance after each update step, striking a balance between the two extremes. As a result of these differing objectives, finetuning tends to favor an initialization that jointly minimizes the loss function, emphasizing immediate performance. In contrast, MAML may settle for an inferior initialization if it shows potential for improvement after adaptation steps. Reptile finds a middle ground, considering both initial performance and subsequent adaptation.

Furthermore, our research has demonstrated that MAML and Reptile exhibit a specialization for adaptation in low-data scenarios within the distribution of training tasks. This specialization is facilitated by two key factors: the weights of the output layer and the scarcity of data in the training tasks. These factors allow MAML and Reptile to explicitly specialize for few-shot learning, giving them an advantage over finetuning when the test tasks are similar to those seen at training time. It is worth noting finetuning cannot specialize for few-shot learning using these two factors, as it begins with a random output layer when learning to perform a new task, and the finetuning method is not pre-trained on simulated low-data tasks.

Our results further show that finetuning learns a broad and diverse set of features that allows it to discriminate between many different classes. MAML and Reptile, in contrast, optimize a look-ahead objective and settle for a less diverse and broad feature space as long as it facilitates robust adaptation in low-data regimes. This can explain findings by Chen et al. (2019), who show that finetuning can yield superior few-shot learning performance when tasks become distant from the training tasks, which is further supported by the fact that there are statistically significant relationships between the broadness of the learned features and the few-shot learning ability for finetuning.

Another result is that MAML yields the best few-shot learning performance when using a shallow convolutional backbone. Interestingly, the features learned by MAML become less discriminative as the depth of the backbone increases. This may indicate an over-specialization. We did not observe such a phenomenon for the finetuning method.

RQ3: Are LSTMs good few-shot learners when evaluated on modern benchmarks?

In Chapter 5, we revisited the plain LSTM approach–originally proposed by Hochreiter et al. (2001) and Younger et al. (2001)–and analyzed it from a few-shot learning perspective. The main idea of this approach is that given a new task, we can enter the entire training dataset into the LSTM, and afterward enter query inputs for which we want to obtain predictions. This plain LSTM approach thus ingests training data for a specific task and conditions predictions for new query inputs on the resulting hidden state. However, as we have argued, there are two potential issues associated with this approach:

1. The hidden embeddings of the support set are not permutation invariant.
2. The learning algorithm and the input embedding mechanism are intertwined, leading to a challenging optimization problem and an increased risk of overfitting.

To address the first issue, mean pooling is proposed to make the embeddings permutation invariant. This method significantly improves the performance of the plain LSTM on few-shot sine wave regression and image classification tasks. It even outperforms the popular meta-learning method MAML on the sine wave regression problem. However, it still struggles to achieve good performance on few-shot image classification tasks, highlighting the optimization difficulties of the plain LSTM approach.

To overcome these difficulties, a new technique called Outer Product LSTM (OP-LSTM) is proposed. OP-LSTM uses an LSTM to update the weights of a base-learner network, effectively decoupling the learning algorithm from the input representation mechanism. This resolves the second issue and allows for more effective optimization. The theoretical analysis shows that OP-LSTM can perform an approximate form of gradient descent (similar to MAML) and a nearest prototype-based approach (similar to Prototypical Networks), demonstrating its flexibility and expressiveness. Empirically, we demonstrate that OP-LSTM overcomes the optimization issues associated with the plain LSTM approach in few-shot image classification benchmarks while using fewer parameters. It achieves competitive or superior performance compared to MAML and Prototypical Networks, which it can approximate.

In summary, while the plain LSTM approach initially had limitations, improvements such as mean pooling and the development of OP-LSTM have made LSTMs more effective as few-shot learners, yielding competitive performance in various few-shot learning tasks.

RQ4: Can the few-shot learning ability of deep neural networks be improved by learning which subsets of parameters to adjust?

In Chapter 6, we proposed a new deep meta-learning algorithm called *Subspace Adaptation Prior* (SAP). SAP jointly learns a good neural network initialization and good parameter subspaces (or subsets of operations) in which new tasks can be learned within a few gradient descent updates from a few data. That is, instead of adapting all parameters when learning a new task, which may be suboptimal and may lead to overfitting during few-shot learning, SAP learns which parameters to adjust when adapting to new tasks.

Our experiments show that SAP outperforms similar existing gradient-based meta-learners in few-shot sine wave regression, yields better performance in single-domain few-shot image classification settings, and yields competitive or superior performance in cross-domain few-shot image classification, where

the test tasks are more distant to the training tasks. This highlights the advantage of learning suitable subspaces in which to perform gradient descent when learning new tasks.

This could be due to the regularization effect of not having to adjust all parameters as well as due to the ability to match structures inherently present in task families. Our experiments in Section 6.5.3 on synthetic task families demonstrate that the SAP is able to learn operations that match the task structure in simple settings in 75% of the cases. In other cases, it may compensate by using other operations that are not inherently present in the task structure.

Inspection of the subspace activation strengths in few-shot image classification reveals that simple and low-dimensional operations, such as shifting features by a single scalar or element-wise by a vector, are important. This is in line with recent work and findings (Triantafillou et al., 2021; Requeima et al., 2019; Bateni et al., 2020) which show that adapting pre-trained embeddings by means of such low-dimensional transformations, such as FiLM layers (Perez et al., 2018), can yield excellent performance.

One limitation of SAP is that it requires the computation of second-order gradients by default during meta-training in order to update the initialization parameters, in a similar fashion as other gradient-based meta-learners such as MAML (Finn et al., 2017), (M)T-Net (Lee and Choi, 2018), and Warp-MAML (Flennerhag et al., 2020). This limitation can be bypassed by using a first-order approximation, which comes at the cost of a performance penalty (between 0.2% and 7.3% accuracy in our experiments).

Gradient-based meta-learning methods struggle to scale well to deep networks as recent work suggests that simple pre-training and fine-tuning of the output layer (Tian et al., 2020; Chen et al., 2021; Huisman et al., 2021) can yield superior performance on common few-shot image classification benchmarks. This is also the reason, besides searching for energy-efficient few-shot learners, that in our experiments we focus on relatively shallow backbones that adapt all layers when learning new tasks, instead of only the output layer. Other limitations are that SAP introduces more parameters and that the candidate pools of operations are selected by hand, despite the fact that these operations are general.

7.2 Future work

This dissertation represents significant progress in advancing our understanding of deep meta-learning algorithms beyond mere performance evaluations. However, with each step forward, we uncover a multitude of unanswered questions. Consequently, our research opens up numerous promising avenues for future exploration, aimed at delving even deeper into the inner workings of deep meta-learning algorithms, which, in turn, can lead to enhanced performance. In addition to the future work directions mentioned in the individual research chapters, below we describe future research directions that we think are most fruitful based on a high-level view of this entire dissertation.

Quantification of task distances A popular criticism of deep meta-learning methods is that their few-shot learning performance is often evaluated on tasks that are quite similar to the tasks seen a training time (within-distribution) (Chen et al., 2019; Triantafillou et al., 2020; Ullah et al., 2022), thereby giving an overly optimistic picture of their ability to generalize from limited data. The reason for this is that the test tasks are often sampled from the same dataset from which the training tasks were sampled. This critique has sparked interest in evaluating deep meta-learning techniques on test tasks sampled from different datasets than those used for training (out-of-distribution), which is also what we have done throughout Chapter 3, Chapter 4, Chapter 5, and Chapter 6. While

this is an important step forward, we argue that it would be even more beneficial to use a measure that quantifies how similar a given test task is compared to the training task distribution, which is something that has also been studied in the field of algorithm selection (Brazdil et al., 2022; Peng et al., 2002). This would allow us to perform a deeper investigation into the inner workings of deep meta-learning algorithms and evaluate their performance as a function of the novelty of tasks. This increased understanding could translate itself into new algorithms or training objectives and improved performance in more challenging scenarios, where the test tasks deviate from the observed training tasks.

Deep meta-learning on different data modalities One of the limitations of the present dissertation lies in the fact that our results were solely obtained on few-shot image classification benchmarks. Recently, the field has also moved towards testing meta-learning algorithms on different data modalities such as text (Lee et al., 2022; Sun et al., 2021), audio (Heggan et al., 2022; Shi et al., 2020), and video (Alet et al., 2021; Wang et al., 2020a). We think that it is an important direction for future work to explore the impact of the data modality, such as images, text, video, and speech, on the performance and behavior of deep meta-learning algorithms. Investigating this promising direction of future work will enable a more comprehensive understanding of how different types of data influence the effectiveness and adaptability of meta-learning techniques. By delving into these unexplored territories, we can uncover potential variations in the capabilities and limitations of meta-learning algorithms across various modalities, paving the way for more informed and tailored approaches in future research.

Towards more general deep meta-learning In Chapter 5 we have revisited the idea of using an LSTM as a general meta-learning algorithm (Hochreiter et al., 2001; Younger et al., 2001) that is fed the training data and can make predictions on new inputs conditioned on the resulting hidden state. A problem with this approach is that the learning algorithm and the input representations are intertwined, which can render the optimization difficult and make the approach susceptible to overfitting. That is, when applying this approach to image data, the LSTM module (the learning algorithm) is applied after a convolutional feature extractor (the input representation). Normally, however, the input representation would be directly fed into a linear layer. By feeding the input representation into an LSTM module, which can consist of multiple stacked LSTMs, the input representation is passed through multiple nonlinearities that could lead to overfitting. We solved this issue by decoupling the input representation from the learning algorithm by having the LSTM module parameterize an outer-product weight update function. However, one could also argue that the main problem is not the intertwinement of the learning algorithm and input representation, but rather the fact that the learning algorithm is applied after the input representation has been computed.

An alternative would be to intertwine the learning algorithm with the input representation from the lowest level of abstraction to the computation of the prediction. This could be implemented through a convolutional LSTM (convLSTM) (Shi et al., 2015) that maintains a state also in the convolutional layers. This can facilitate learning at lower levels of input representation. It is also possible to investigate other architectures for this purpose such as general transformer architectures. Given the success of other prompt-based models, we think that this approach, whilst computationally expensive, could lead to novel learning algorithms and improved few-shot learning performance.

Understanding black-box learning algorithms While the general-purpose meta-learning algorithms belonging to the black-box/model-based category of deep meta-learning techniques have the potential to improve the few-shot learning capabilities of deep neural networks, it is difficult to interpret their inner workings (how they adapt to new tasks). Future work can aim to address

this interpretability challenge, and allow us to gain valuable insights that, in turn, can enable us to develop new hand-crafted learning algorithms. That is, by studying the behavior of these black-box/meta-based meta-learning models, we can uncover novel learning strategies, principles, or heuristics that could be employed in the development of even more efficient interpretable learning algorithms. Increasing our understanding of black-box models can also help bridge the gap between cutting-edge deep learning techniques and traditional, interpretable approaches. Additionally, by unraveling the inner workings of these powerful meta-learning algorithms, we can enhance their trustworthiness and facilitate the integration of these models into real-world applications where explainability and interpretability are essential. Overall, addressing the challenges associated with interpretability not only contributes to the advancement of deep meta-learning techniques but also opens up new avenues to assisting human researchers in designing new learning algorithms.

Bibliography

Alet, F., Doblar, D., Zhou, A., Tenenbaum, J., Kawaguchi, K., and Finn, C. (2021). Noether networks: meta-learning useful conserved quantities. *Advances in Neural Information Processing Systems*, 34:16384–16397.

Alver, S. and Precup, D. (2021). What is going on inside recurrent meta reinforcement learning agents? *arXiv preprint arXiv:2104.14644*.

Anderson, T. (2008). *The Theory and Practice of Online Learning*. AU Press, Athabasca University.

Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems 29*, pages 3988–3996. Curran Associates Inc.

Antoniou, A., Edwards, H., and Storkey, A. (2019). How to train your MAML. In *International Conference on Learning Representations (ICLR'19)*.

Baik, S., Choi, M., Choi, J., Kim, H., and Lee, K. M. (2020). Meta-learning with adaptive hyperparameters. In *Advances in Neural Information Processing Systems 33*.

Barrett, D. G., Hill, F., Santoro, A., Morcos, A. S., and Lillicrap, T. (2018). Measuring abstract reasoning in neural networks. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, pages 4477–4486. JLMR.org.

Bateni, P., Goyal, R., Masrani, V., Wood, F., and Sigal, L. (2020). Improved few-shot visual classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14493–14502.

Bengio, S., Bengio, Y., Cloutier, J., and Gecsei, J. (1997). On the optimization of a synaptic learning rule. In *Optimality in Artificial and Biological Neural Networks*. Lawrance Erlbaum Associates, Inc.

Bengio, Y., Bengio, S., and Cloutier, J. (1991). Learning a synaptic learning rule. In *International Joint Conference on Neural Networks (IJCNN'91)*, volume 2. IEEE.

Bertinetto, L., Henriques, J. F., Torr, P., and Vedaldi, A. (2019). Meta-learning with differentiable closed-form solvers. In *International Conference on Learning Representations (ICLR'19)*.

Bottou, L. (2004). Stochastic Learning. In *Advanced lectures on machine learning*, pages 146–168. Springer.

Brazdil, P., Carrier, C. G., Soares, C., and Vilalta, R. (2008). *Metalearning: Applications to Data Mining*. Springer-Verlag Berlin Heidelberg.

Brazdil, P., van Rijn, J. N., Soares, C., and Vanschoren, J. (2022). *Metalearning: Applications to Automated Machine Learning and Data Mining*. Springer, 2nd edition.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Chan, S. C., Santoro, A., Lampinen, A. K., Wang, J. X., Singh, A. K., Richemond, P. H., McClelland, J., and Hill, F. (2022). Data distributional properties drive emergent in-context learning in transformers. In *Advances in Neural Information Processing Systems*.

Chen, W.-Y., Liu, Y.-C., Kira, Z., Wang, Y.-C. F., and Huang, J.-B. (2019). A closer look at few-shot classification. In *International Conference on Learning Representations (ICLR'19)*.

Chen, Y., Liu, Z., Xu, H., Darrell, T., and Wang, X. (2021). Meta-baseline: Exploring simple meta-learning for few-shot learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9062–9071.

Clavera, I., Nagabandi, A., Liu, S., Fearing, R. S., Abbeel, P., Levine, S., and Finn, C. (2019). Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. In *International Conference on Learning Representations (ICLR'19)*.

Collins, L., Mokhtari, A., and Shakkottai, S. (2020). Why does maml outperform erm? an optimization perspective. *arXiv preprint arXiv:2010.14672*.

Daumé III, H. (2009). Frustratingly easy domain adaptation. *arXiv preprint arXiv:0907.1815*.

Deleu, T., Würfl, T., Samiei, M., Cohen, J. P., and Bengio, Y. (2019). Torchmeta: A Meta-Learning library for PyTorch. Available at: https://github.com/tristandeleu/pytorch-meta.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE.

Ding, B. (2023). Understanding maml through its loss landscape. Master's thesis, Leiden University.

Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. (2016). RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning. *arXiv preprint arXiv:1611.02779*.

Edwards, H. and Storkey, A. (2017). Towards a Neural Statistician. In *International Conference on Learning Representations (ICLR'17)*.

Elsken, T., Staffler, B., Metzen, J. H., and Hutter, F. (2020). Meta-learning of neural architectures for few-shot learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'20)*, pages 12365–12375.

Farahani, A., Voghoei, S., Rasheed, K., and Arabnia, H. R. (2021). A brief review of domain adaptation. *Advances in Data Science and Information Engineering: Proceedings from ICDATA 2020 and IKE 2020*, pages 877–894.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*, pages 1126–1135. PMLR.

Finn, C. and Levine, S. (2018). Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. In *International Conference on Learning Representations (ICLR'18)*.

Finn, C., Rajeswaran, A., Kakade, S., and Levine, S. (2019). Online Meta-Learning. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, pages 1920–1930. JLMR.org.

Finn, C., Xu, K., and Levine, S. (2018). Probabilistic Model-Agnostic Meta-Learning. In *Advances in Neural Information Processing Systems 31*, pages 9516–9527.

Flennerhag, S., Rusu, A. A., Pascanu, R., Visin, F., Yin, H., and Hadsell, R. (2020). Meta-learning with warped gradient descent. In *International Conference on Learning Representations (ICLR'20)*.

Garcia, V. and Bruna, J. (2017). Few-Shot Learning with Graph Neural Networks. In *International Conference on Learning Representations (ICLR'17)*.

Garnelo, M., Rosenbaum, D., Maddison, C., Ramalho, T., Saxton, D., Shanahan, M., Teh, Y. W., Rezende, D., and Eslami, S. M. A. (2018). Conditional neural processes. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, volume 80, pages 1704–1713.

Grant, E., Finn, C., Levine, S., Darrell, T., and Griffiths, T. (2018). Recasting gradient-based meta-learning as hierarchical bayes. In *International Conference on Learning Representations (ICLR'18)*.

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

Gupta, A., Mendonca, R., Liu, Y., Abbeel, P., and Levine, S. (2018). Meta-Reinforcement Learning of Structured Exploration Strategies. In *Advances in Neural Information Processing Systems 31*, pages 5302–5311. Curran Associates Inc.

Hamilton, W., Ying, Z., and Leskovec, J. (2017). Inductive representation learning on large graphs. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30.

Hannan, J. (1957). Approximation to bayes risk in repeated play. *Contributions to the Theory of Games*, 3:97–139.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY, 2nd edition.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

Heggan, C., Budgett, S., Hospedales, T., and Yaghoobi, M. (2022). Metaaudio: A few-shot audio classification benchmark. In *International Conference on Artificial Neural Networks*, pages 219–230. Springer.

Hinton, G. E. and Plaut, D. C. (1987). Using Fast Weights to Deblur Old Memories. In *Proceedings of the 9th Annual Conference of the Cognitive Science Society*, pages 177–186.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hochreiter, S., Younger, A. S., and Conwell, P. R. (2001). Learning to Learn Using Gradient Descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer.

Hollmann, N., Müller, S., Eggensperger, K., and Hutter, F. (2023). Tabpfn: A transformer that solves small tabular classification problems in a second. In *The Eleventh International Conference on Learning Representations, ICLR 2023*. OpenReview.net.

Hospedales, T. M., Antoniou, A., Micaelli, P., and Storkey, A. J. (2021). Meta-learning in neural networks: A survey. *IEEE Transactions on Pattern Analysis & Machine Intelligence*.

Huisman, M., van Rijn, J. N., and Plaat, A. (2021). A preliminary study on the feature representations of transfer learning and gradient-based meta-learning techniques. In *Fifth Workshop on Meta-Learning at the Conference on Neural Information Processing Systems*.

Iqbal, M. S., Luo, B., Khan, T., Mehmood, R., and Sadiq, M. (2018). Heterogeneous transfer learning techniques for machine learning. *Iran Journal of Computer Science*, 1(1):31–46.

Jang, E., Gu, S., and Poole, B. (2017). Categorical reparameterization with gumbel-softmax. In *5th International Conference on Learning Representations, (ICLR'17)*.

Jankowski, N., Duch, W., and Grąbczewski, K. (2011). *Meta-Learning in Computational Intelligence*, volume 358. Springer-Verlag Berlin Heidelberg.

Jiang, W., Kwok, J., and Zhang, Y. (2022). Subspace learning for effective meta-learning. In *Proceedings of the 39th International Conference on Machine Learning*, pages 10177–10194. PMLR.

Kalai, A. and Vempala, S. (2005). Efficient algorithms for online decision problems. *Journal of Computer and System Sciences*, 71(3):291–307.

Kim, J., Lee, S., Kim, S., Cha, M., Lee, J. K., Choi, Y., Choi, Y., Cho, D.-Y., and Kim, J. (2018). Auto-meta: Automated gradient based meta learner search. *arXiv preprint arXiv:1806.06927*.

Kingma, D. P. and Ba, J. L. (2015). Adam: A method for stochastic gradient descent. In *International Conference on Learning Representations (ICLR'15)*.

Kirsch, L., Harrison, J., Sohl-Dickstein, J., and Metz, L. (2022). General-purpose in-context learning by meta-learning transformers. *arXiv preprint arXiv:2212.04458*.

Kirsch, L. and Schmidhuber, J. (2021). Meta learning backpropagation and improving it. In *Advances in Neural Information Processing Systems 34*, pages 14122–14134.

Koch, G., Zemel, R., and Salakhutdinov, R. (2015). Siamese Neural Networks for One-shot Image Recognition. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*, volume 37. JMLR.org.

Krizhevsky, A. (2009). Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105.

Lake, B., Salakhutdinov, R., Gross, J., and Tenenbaum, J. (2011). One shot learning of simple visual concepts. In *Proceedings of the annual meeting of the cognitive science society*, volume 33, pages 2568–2573.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2017). Building machines that learn and think like people. *Behavioral and brain sciences*, 40.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

LeCun, Y., Cortes, C., and Burges, C. (2010). MNIST handwritten digit database. `http://yann.lecun.com/exdb/mnist`. Accessed: 7-10-2020.

Lee, H.-y., Li, S.-W., and Vu, N. T. (2022). Meta learning for natural language processing: A survey. *arXiv preprint arXiv:2205.01500*.

Lee, K., Maji, S., Ravichandran, A., and Soatto, S. (2019). Meta-learning with differentiable convex optimization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 10657–10665.

Lee, Y. and Choi, S. (2018). Gradient-based meta-learning with learned layerwise metric and subspace. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, pages 2927–2936. PMLR.

Li, K. and Malik, J. (2018). Learning to Optimize Neural Nets. *arXiv preprint arXiv:1703.00441*.

Li, Z., Zhou, F., Chen, F., and Li, H. (2017). Meta-SGD: Learning to Learn Quickly for Few-Shot Learning. *arXiv preprint arXiv:1707.09835*.

Lian, D., Zheng, Y., Xu, Y., Lu, Y., Lin, L., Zhao, P., Huang, J., and Gao, S. (2019). Towards fast adaptation of neural architectures with meta learning. In *International Conference on Learning Representations (ICLR'19)*.

Liu, H., Simonyan, K., and Yang, Y. (2019). DARTS: Differentiable architecture search. In *International Conference on Learning Representations (ICLR'19)*.

Liu, Q. and Wang, D. (2016). Stein Variational Gradient Descent: A General Purpose Bayesian Inference Algorithm. In *Advances in neural information processing systems 29*, pages 2378–2386. Curran Associates Inc.

Lu, J., Gong, P., Ye, J., and Zhang, C. (2020). Learning from very few samples: A survey. *arXiv preprint arXiv:2009.02653*.

Maddison, C. J., Mnih, A., and Teh, Y. W. (2017). The concrete distribution: A continuous relaxation of discrete random variables. In *5th International Conference on Learning Representations, (ICLR'17)*.

Mangla, P., Kumari, N., Sinha, A., Singh, M., Krishnamurthy, B., and Balasubramanian, V. N. (2020). Charting the right manifold: Manifold mixup for few-shot learning. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2218–2227.

Martens, J. and Grosse, R. (2015). Optimizing Neural Networks with Kronecker-factored Approximate Curvature. In *Proceedings of the 32th International Conference on Machine Learning (ICML'15)*, pages 2408–2417. JMLR.org.

Metz, L., Maheswaranathan, N., Nixon, J., Freeman, D., and Sohl-Dickstein, J. (2019). Understanding and correcting pathologies in the training of learned optimizers. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, pages 4556–4565. PMLR.

Miconi, T., Rawal, A., Clune, J., and Stanley, K. O. (2019). Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity. In *International Conference on Learning Representations (ICLR'19)*.

Miconi, T., Stanley, K., and Clune, J. (2018). Differentiable plasticity: training plastic neural networks with backpropagation. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, pages 3559–3568. JLMR.org.

Mishra, N., Rohaninejad, M., Chen, X., and Abbeel, P. (2018). A simple neural attentive meta-learner. In *International Conference on Learning Representations (ICLR'18)*.

Mitchell, T. M. (1980). The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Munkhdalai, T. and Yu, H. (2017). Meta networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*, pages 2554–2563. JLMR.org.

Naik, D. K. and Mammone, R. J. (1992). Meta-neural networks that learn by learning. In *International Joint Conference on Neural Networks (IJCNN'92)*, volume 1, pages 437–442. IEEE.

Nichol, A., Achiam, J., and Schulman, J. (2018). On First-Order Meta-Learning Algorithms. *arXiv preprint arXiv:1803.02999*.

Olah, C. (2015). Understanding LSTM Networks. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. Accessed: 23-01-2023.

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). WaveNet: A Generative Model for Raw Audio. *arXiv preprint arXiv:1609.03499*.

Oreshkin, B., López, P. R., and Lacoste, A. (2018). Tadam: Task dependent adaptive metric for improved few-shot learning. In *Advances in Neural Information Processing Systems 31*, pages 721–731. Curran Associates Inc.

Pan, S. J. and Yang, Q. (2009). A Survey on Transfer Learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

Park, E. and Oliva, J. B. (2019a). Meta-curvature. In *Advances in Neural Information Processing Systems 32*, NIPS'19, pages 3314–3324.

Park, E. and Oliva, J. B. (2019b). Meta-curvature. In *Advances in Neural Information Processing Systems 32*, pages 3309–3319.

Peng, Y., Flach, P. A., Soares, C., and Brazdil, P. (2002). Improved Dataset Characterisation for Meta-learning. In *International Conference on Discovery Science*, volume 2534 of *Lecture Notes in Computer Science*, pages 141–152. Springer.

Perez, E., Strub, F., De Vries, H., Dumoulin, V., and Courville, A. (2018). Film: Visual reasoning with a general conditioning layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Raghu, A., Raghu, M., Bengio, S., and Vinyals, O. (2020). Rapid Learning or Feature Reuse? Towards Understanding the Effectiveness of MAML. In *International Conference on Learning Representations (ICLR'20)*.

Rajeswaran, A., Finn, C., Kakade, S. M., and Levine, S. (2019). Meta-Learning with Implicit Gradients. In *Advances in Neural Information Processing Systems 32*, pages 113–124.

Ravi, S. and Larochelle, H. (2017). Optimization as a Model for Few-Shot Learning. In *International Conference on Learning Representations (ICLR'17)*.

Ren, M., Ravi, S., Triantafillou, E., Snell, J., Swersky, K., Tenenbaum, J. B., Larochelle, H., and Zemel, R. S. (2018). Meta-learning for semi-supervised few-shot classification. In *International Conference on Learning Representations (ICLR'18)*.

Requeima, J., Gordon, J., Bronskill, J., Nowozin, S., and Turner, R. E. (2019). Fast and flexible multi-task classification using conditional neural adaptive processes. In *Advances in Neural Information Processing Systems 32*, pages 7957–7968.

Rusu, A. A., Rao, D., Sygnowski, J., Vinyals, O., Pascanu, R., Osindero, S., and Hadsell, R. (2019). Meta-learning with latent embedding optimization. In *International Conference on Learning Representations (ICLR'19)*.

Salakhutdinov, R., Tenenbaum, J., and Torralba, A. (2012). One-shot learning with a hierarchical nonparametric bayesian model. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, pages 195–206. JMLR Workshop and Conference Proceedings.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. (2016). Meta-learning with Memory-augmented Neural Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML'16)*, pages 1842–1850.

Schmidhuber, J. (1987). Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. Master's thesis, Technische Universität München.

Schmidhuber, J. (1993). A neural network that embeds its own meta-levels. In *IEEE International Conference on Neural Networks*, pages 407–412. IEEE.

Schmidhuber, J., Zhao, J., and Wiering, M. (1997). Shifting Inductive Bias with Success-Story Algorithm, Adaptive Levin Search, and Incremental Self-Improvement. *Machine Learning*, 28(1):105–130.

Schmidt, L. A. (2009). *Meaning and compositionality as statistical induction of categories and constraints*. PhD thesis, Massachusetts Institute of Technology.

Shi, B., Sun, M., Puvvada, K. C., Kao, C.-C., Matsoukas, S., and Wang, C. (2020). Few-shot acoustic event detection via meta learning. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 76–80. IEEE.

Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-K., and Woo, W.-c. (2015). Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems 28*.

Shyam, P., Gupta, S., and Dukkipati, A. (2017). Attentive Recurrent Comparators. In *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*, pages 3173–3181. JLMR.org.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Simon, C., Koniusz, P., Nock, R., and Harandi, M. (2020). On modulating the gradient for meta-learning. In *European Conference on Computer Vision*, pages 556–572. Springer.

Simons, T. (2022). The training of neural networks that can train neural networks. Master's thesis, Leiden University.

Snell, J., Swersky, K., and Zemel, R. (2017). Prototypical Networks for Few-shot Learning. In *Advances in Neural Information Processing Systems 30*, pages 4077–4087. Curran Associates Inc.

Sun, C., Shrivastava, A., Singh, S., and Gupta, A. (2017). Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 843–852.

Sun, P., Ouyang, Y., Zhang, W., and Dai, X. (2021). Meda: Meta-learning with data augmentation for few-shot text classification. In *IJCAI*, pages 3929–3935.

Sun, Q., Liu, Y., Chua, T.-S., and Schiele, B. (2019). Meta-transfer learning for few-shot learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 403–412.

Sung, F., Yang, Y., Zhang, L., Xiang, T., Torr, P. H., and Hospedales, T. M. (2018). Learning to Compare: Relation Network for Few-Shot Learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1199–1208. IEEE.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT press, 2nd edition.

Taylor, M. E. and Stone, P. (2009). Transfer Learning for Reinforcement Learning Domains: A Survey. *Journal of Machine Learning Research*, 10(7).

Thrun, S. (1998). Lifelong Learning Algorithms. In *Learning to learn*, pages 181–209. Springer.

Tian, Y., Wang, Y., Krishnan, D., Tenenbaum, J. B., and Isola, P. (2020). Rethinking few-shot image classification: a good embedding is all you need? *arXiv preprint arXiv:2003.11539*.

Tieleman, T. and Hinton, G. (2017). Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning. *Technical Report*.

Tokmakov, P., Wang, Y.-X., and Hebert, M. (2019). Learning Compositional Representations for Few-Shot Recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 6372–6381.

Triantafillou, E., Larochelle, H., Zemel, R., and Dumoulin, V. (2021). Learning a universal template for few-shot dataset generalization. In *Proceedings of the 38th International Conference on Machine Learning (ICML'21)*, pages 10424–10433. PMLR.

Triantafillou, E., Zhu, T., Dumoulin, V., Lamblin, P., Evci, U., Xu, K., Goroshin, R., Gelada, C., Swersky, K., Manzagol, P.-A., and Larochelle, H. (2020). Meta-dataset: A dataset of datasets for learning to learn from few examples. In *International Conference on Learning Representations (ICLR'20)*.

Ullah, I., Carrión-Ojeda, D., Escalera, S., Guyon, I., Huisman, M., Mohr, F., van Rijn, J. N., Sun, H., Vanschoren, J., and Vu, P. A. (2022). Meta-album: Multi-domain meta-dataset for few-shot image classification. *Advances in Neural Information Processing Systems*, 35:3232–3247.

Vanschoren, J. (2018). Meta-Learning: A Survey. *arXiv preprint arXiv:1810.03548*.

Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. (2014). OpenML: Networked Science in Machine Learning. *SIGKDD Explorations*, 15(2):49–60.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention Is All You Need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates Inc.

Vinyals, O. (2017). Talk: Model vs optimization meta learning. `http://metalearning-symposium.ml/files/vinyals.pdf`. Neural Information Processing Systems (NIPS'17); accessed 06-06-2020.

Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., and Wierstra, D. (2016). Matching Networks for One Shot Learning. In *Advances in Neural Information Processing Systems 29*, pages 3637–3645.

Vuorio, R., Cho, D.-Y., Kim, D., and Kim, J. (2018). Meta Continual Learning. *arXiv preprint arXiv:1806.06928*.

Wah, C., Branson, S., Welinder, P., Perona, P., and Belongie, S. (2011). The Caltech-UCSD Birds-200-2011 Dataset. Technical Report CNS-TR-2011-001, California Institute of Technology.

Wang, G., Luo, C., Sun, X., Xiong, Z., and Zeng, W. (2020a). Tracking by instance detection: A meta-learning approach. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6288–6297.

Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. (2016). Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*.

Wang, Y., Yao, Q., Kwok, J. T., and Ni, L. M. (2020b). Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys*, 53(3):1–34.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Łukasz Kaiser, Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's Neural

Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144*.

Yang, S., Liu, L., and Xu, M. (2021). Free lunch for few-shot learning: Distribution calibration. In *International Conference on Learning Representations (ICLR'21)*.

Yin, M., Tucker, G., Zhou, M., Levine, S., and Finn, C. (2020). Meta-learning without memorization. In *International Conference on Learning Representations (ICLR'20)*.

Yoon, J., Kim, T., Dia, O., Kim, S., Bengio, Y., and Ahn, S. (2018). Bayesian Model-Agnostic Meta-Learning. In *Advances in Neural Information Processing Systems 31*, pages 7332–7342. Curran Associates Inc.

Younger, A. S., Hochreiter, S., and Conwell, P. R. (2001). Meta-learning with backpropagation. In *International Joint Conference on Neural Networks (IJCNN'01)*, volume 3. IEEE.

Yu, T., Quillen, D., He, Z., Julian, R., Hausman, K., Finn, C., and Levine, S. (2019). Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning. *arXiv preprint arXiv:1910.10897*.

Zintgraf, L., Shiarli, K., Kurin, V., Hofmann, K., and Whiteson, S. (2019). Fast context adaptation via meta-learning. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, pages 7693–7702. PMLR.

Appendix A

Hyperparameter details for TURTLE

This is the appendix for Chapter 3, where we describe additional hyperparameter details.

A.1 Used hyperparameters

For all techniques mentioned below, we performed meta-validation after every 2,500 training tasks. The best-resulting configuration was evaluated at meta-test time.

For sine wave regression, we use the same base-learner network as Finn et al. (2017), i.e., a fully-connected feed-forward network consisting of a single input node followed by two hidden layers with 40 ReLU nodes each and a final single-node output layer.

For few-shot image classification problems, we use the same base-learner network as used by Snell et al. (2017) and Chen et al. (2019). This network is a stack of four identical convolutional blocks. Each block consists of 64 convolutions of size 3×3, batch normalization, a ReLU nonlinearity, and a 2D max-pooling layer with a kernel size of 2. The resulting embeddings of the $84 \times 84 \times 3$ input images are flattened and fed into a dense layer with N nodes (one for every class in a task). The base-learner is trained to minimize the cross-entropy loss on the query set, conditioned on the support set.

Transfer learning baselines Note that these models (TrainFromScratch, finetuning, baseline++) pre-trained on minibatches of size 16 sampled from the joint data obtained by merging all meta-training tasks. At test time, they were trained for 100 steps on mini-batches of size 4 sampled from new tasks following Chen et al. (2019). Every 25 steps, we evaluated their performance on the entire support set to select the best configuration to test on the query set.

LSTM meta-learner For selecting the hyperparameters of the LSTM meta-learner[1], we followed Ravi and Larochelle (2017). That is, we use a 2-layer architecture, and Adam as meta-optimizer with a learning rate of 0.001. The batch size was set equal to the size of the task. Meta-gradients were

[1]Used code: `https://github.com/markdtw/meta-learning-lstm-pytorch`.

clipped to have a norm of at most 0.25, following. The meta-network receives four inputs obtained by preprocessing the loss and gradients using in similar fashion to Andrychowicz et al. (2016) and Ravi and Larochelle (2017). On miniImageNet and CUB, the LSTM optimizer is set to perform 12 updates per task when the number of examples per class is $k = 1$ and 5 updates when $k = 5$.

MAML Again, we follow Finn et al. (2017) for selecting the hyperparameters, except for the meta-batch size on sine wave regression as we found it not to help performance. This means that the inner learning rate was set to 0.01 and the outer learning rate to 0.001, with Adam as meta-optimizer. These settings hold for both sine wave regression and image classification. When $T > 1$, we use gradient value clipping with a threshold of 10. On image classification, MAML was set to optimize the initial parameters based on $T = 5$ update steps, but an additional 5 steps were made afterwards to further increase the performance. Moreover, we used a meta-batch size of 4 and 2 for 1- and 5-shot image classification respectively.

TURTLE We performed many experiments with the hyperparameters of TURTLE on sine wave regression. Here, we only report the settings that were found to give the best performance, which were also used on the image classification problems. That is, the meta-network consists of 5 hidden layers of 20 nodes each. Every hidden node is followed by a ReLU nonlinearity. The input consists of a raw gradient, a historical real-valued number indicating the moving average of the previous input gradients with a (with a beta decay of 0.9), and a time step integer $t \in \{0, ..., T - 1\}$. The output layer consists of a single node which corresponds to the proposed weight update. For training, we used meta-batches of size 2. Additionally, TURTLE maintains a separate learning rate for all weights in the base-learner network. Lastly, TURTLE uses second-order gradients and Adam as meta-optimizer with a learning rate of 0.001.

Appendix B

Additional experimental results for OP-LSTM

This is the appendix for Chapter 5, where we present additional experimental results.

B.1 Sine wave regression: additional results

We also performed an experiment to investigate the effect of the input representation on the performance of the plain LSTM approach (proposed by Younger et al. (2001); Hochreiter et al. (2001)) on the 5-shot sine wave regression performance. The experimental setting follows the setup described in Section 5.5.1. For every input format, we performed hyperparameter tuning with the same randomly sampled hyperparameter configurations using Table B.2. The performances of the best validated models per input format are displayed in Table B.1. The best performance is obtained by feeding the current input, previous target, and the previous prediction into the LSTM, although the differences with other inputs are small.

Table B.1: The influence of different input information on the performance of the LSTM on 5-shot sine wave regression. 95% confidnce intervals are displayed as $\pm x$.

Input $\mathbf{x}_t$	Prev target y_{t-1}	Prev pred $\hat{y}_{t-1}$	Prev error e_{t-1}	5-shot MSE
✓	✓			0.04 ± 0.002
✓	✓	✓		$\mathbf{0.03} \pm 0.002$
✓	✓		✓	0.05 ± 0.004
✓	✓	✓	✓	0.06 ± 0.011

B.2 Hyperparameter tuning

B.2.1 Permutation invariance experiments

For the permutation invariance experiments on few-shot sine wave regression, we sampled 20 random configurations for the plain LSTM from the distributions displayed in Table B.2 and validated their performance on 5-shot ($k = 5$) sine-wave regression. We selected the best configuration and evaluated it on the meta-test tasks,

Table B.2: The used ranges and distributions for tuning the hyperparameters with random search for sine wave regression.

Hyperparameter	Range
Number of layers	Uniform({1,2,3,4})
Hidden dimensions	Uniform({1,3,8,20,40})
Meta-batch size	Uniform({1,2,3,4})
Learning rate	LogUniform(1e-5, 4e-2)
Unroll steps	Uniform({1,2,...,14})

For Omniglot, we performed random search with a function evaluation budget of 100, with a fixed learning rate of 0.001. The architecture of the plain LSTM with sequential data processing was sampled uniformly at random from {1024-512-256-128-64,2048-1024-512-128-64,2048-1024-512-256-128,1024-600-400-200-92,1024-512-512-256-128-64,1024-512-512-256-256-128-64, 612-400-256-128-64,1024-1024-1024-512-256-128-64,2048-1024-512-180-100,1024-580-280-160-80,256-128-64, 512-256-128-64,128-64-64-64,256-128-64,512-256-64,256-128-100,128-64-64-64-64,64-64-64-64,50-50}, the number of passes over the support data T was sampled uniformly at random from $\{1, 2, \ldots, 10\}$, and the meta-batch size from $\{1, 2, \ldots, 32\}$. We used the best hyperparameter configuration of the sequential plain LSTM for the plain LSTM with batching to compare the differences in performance.

B.2.2 Omniglot

For the **plain LSTM** approach, we used the best hyperparameter configuration found for the permutation invariance experiments.

For **OP-LSTM**, we performed a grid search, varying the meta-batch size within {1,4,8,16,32}, the architecture of the coordinate-wise LSTM within {20-1, 10-10-1, 40-5, 40-20-1, 20-20-20-5} (note that the last element is always 1 because it operates per coordinate), and the number of passes over the support set within {1,3,5,10}.

Detailed learning curves for the plain LSTM on Omniglot Here, we show the validation learning curves of the sequential LSTM and the LSTM which uses batching to complement the results displayed in Section 5.5.1. Figure B.1 displays the validation learning curves of the LSTM with batch data ingestion (top row) and the LSTM with sequential data processing (bottom row). As we can see, batching increases the stability of the training process and makes the LSTM less sensitive to the random initialization, as every run succeeds to reach convergence in contrast to the sequential LSTM.

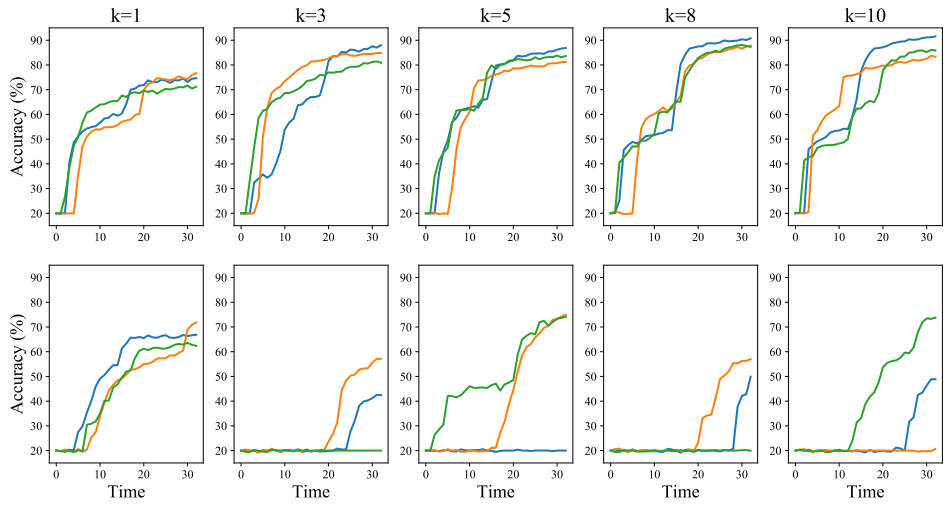


Figure B.1: The mean validation accuracy of the LSTM over time on Omniglot for every of the three different runs, for different numbers of examples per class k. **Top row:** LSTM with batching (mean-pooling). **Bottom row:** LSTM with sequential data ingestion. As we can see, batching improves the stability of the training process.

B.2.3 miniImageNet and CUB

For **plain LSTM**, we used random search with a budget of 130 function evaluations, the meta-batch size was sampled uniformly between 1 and 48, the number of layers between 1 and 4, the hidden size log-uniformly between 32 and 3200, and the number of passes T over the support dataset uniformly between 2 and 9.

For **OP-LSTM**, we performed the same grid search as for Omniglot. We use the best found hyperparameters for both methods on miniImageNet also on CUB.

We also measured the running times of the techniques on miniImageNet and CUB, as shown in Table B.3. We note that the running times may be affected by the server's load and thus can only give a rough estimation of the required amount of compute time. As we can see, the plain LSTM is the slowest method, despite achieving random performance on miniImageNet. OP-LSTM, in contrast, is more efficient.

B.3 Robustness to random seeds

Here, we investigate the robustness of the investigated methods to the random seed for the few-shot image classification experiments performed in Section 5.5.3. We perform th Instead of computing the confidence intervals over the performances of all test tasks for all seeds, we now compute the confidence interval over the mean test performance per run. As we perform three runs per method, we compute the confidence intervals over three observations per method. Note that the mean performance does not change as taking the mean of the three means will be equivalent (as the means

Table B.3: Mean running times on 5-way miniImageNet and CUB classification over 3 runs. All methods used a Conv-4 backbone as a feature extractor. "xh$y$min" means x hours and y minutes. The "-" indicates that the method did not finish within 2 days of running time.

		miniImageNet		CUB	
Technique	params	1-shot	5-shot	1-shot	5-shot
MAML	121 093	13h9min	12h1min	26h57min	17h39min
Warp-MAML	231 877	12h25min	12h30min	13h6min	12h48min
SAP	412 852	5h40min	11h14min	7h11min	11h17min
ProtoNet	121 093	4h14min	5h6min	31h18min	38h46min
LSTM	55 879 349	40h14min	46h47min	-	-
OP-LSTM (ours)	141 187	4h50min	5h31min	31h58min	40h8min

are based on an equal number of task performances).

B.4 Within-domain

Here, we present additional results for the conducted within-domain image classification experiments.

Omniglot The mean test performance and confidence intervals over the random seeds for Omniglot image classification are shown in Table B.4. As we can see, the confidence intervals are higher than in previous experiments because the intervals are computed over 3 observations instead of 1800 individual test task performances (600 per run). As we can see, the LSTM is unstable, supporting the hypothesis that the optimization problem is difficult. OP-LSTM, on the other hand, is less sensitive to the chosen random seed and has a stability that is comparable to that of MAML.

Table B.4: The mean test accuracy (%) on 5-way Omniglot classification across 3 different runs. The 95% confidence intervals, computed over the mean performances of the 3 different random seeds, are displayed as $\pm x$. The plain LSTM is outperformed by MAML. All methods (except LSTM) used a fully-connected feed-forward classifier.

Technique	parameters	1-shot	5-shot
MAML	247 621	84.1 ± 3.10	**93.5** ± 0.70
ProtoNet	247 621	83.6 ± 0.52	93.4 ± 1.48
LSTM	13 530 097	72.6 ± 3.87	84.8 ± 6.12
OP-LSTM (ours)	249 167	**84.3** ± 3.18	91.8 ± 0.70

MiniImageNet and CUB The mean test performance and confidence intervals over the random seeds for miniImageNet and CUB image classification are shown in Table B.5. In contrast to what we observed on Omniglot, the LSTM is now more stable. This is caused by the fact that it consistently fails to learn a learning algorithm that performs better than random guessing, and thus performs stably at chance level.

Table B.5: Meta-test accuracy scores on 5-way miniImageNet and CUB classification over 3 runs. The 95% confidence intervals, computed over the mean performances of the 3 different random seeds, are displayed as ± x. All methods used a Conv-4 backbone as a feature extractor. The "-" indicates that the method did not finish within 2 days of running time.

		miniImageNet		CUB	
Technique	params	1-shot	5-shot	1-shot	5-shot
MAML	121 093	48.6 ± 4.00	63.0 ± 0.33	57.5 ± 0.83	**74.8** ± 2.10
Warp-MAML	231 877	50.4 ± 2.58	65.6 ± 0.98	59.6 ± 2.15	74.2 ± 2.51
SAP	412 852	**53.0** ± 3.71	67.6 ± 0.47	**63.5** ± 6.24	73.9 ± 1.57
ProtoNet	121 093	50.1 ± 4.06	65.4 ± 2.84	50.9 ± 2.35	63.7 ± 0.47
LSTM	55 879 349	20.2 ± 0.60	19.4 ± 0.47	-	-
OP-LSTM (ours)	141 187	51.9 ± 2.52	**67.9** ± 2.40	60.2 ± 1.58	73.1 ± 1.57

Table B.6: Average cross-domain meta-test accuracy scores over 5 runs using a Conv-4 backbone. Techniques trained on tasks from one data set and were evaluated on tasks from another data set. The 95% confidence intervals, computed over the mean performances of the 3 different random seeds, are displayed as ± x. The "-" indicates that the method did not finish within 2 days of running time.

	MIN → CUB		CUB → MIN	
	1-shot	5-shot	1-shot	5-shot
MAML	37.9 ± 2.22	53.6 ± 0.67	31.1 ± 1.19	45.8 ± 2.06
Warp-MAML	42.0 ± 0.85	56.9 ± 4.16	31.1 ± 1.59	41.3 ± 1.37
SAP	41.5 ± 3.72	58.0 ± 1.79	33.3 ± 2.33	47.1 ± 1.28
ProtoNet	39.7 ± 4.11	56.0 ± 4.89	31.7 ± 0.20	45.3 ± 1.84
LSTM	20.1 ± 0.77	20.0 ± 0.40	-	-
OP-LSTM (ours)	**42.3** ± 1.90	**58.5** ± 1.49	**35.8** ± 2.98	**49.0** ± 0.80

B.5 Cross-domain

Lastly, we compute the confidence intervals in cross-domain settings and display the results in Table B.6. Again, the LSTM is a stable random guesser. The other algorithms are less stable, but do yield a better performance. We cannot observe a general pattern of stability in the sense that one algorithm is consistently more stable than others.

Appendix C

Additional experimental results for SAP

In this appendix for Chapter 6, we show additional experimental results on few-shot image classification.

C.1 Validation of re-implementation

	1-shot		5-shot	
	Reported	Local Repr	Reported	Local repr
MAML	48.7 ± 1.8	48.0 ± 0.8	63.2 ± 0.9	64.4 ± 0.4
T-Net	50.9 ± 1.8	48.9 ± 0.8	-	65.3 ± 0.4
MT-Net	51.7 ± 1.8	48.5 ± 0.8	-	63.0 ± 0.4
Warp-MAML*	-	49.5 ± 0.8	-	63.9 ± 0.4
SAP (ours)	-	**51.6** ± 0.8	-	**65.9** ± 0.4

Table C.1: Mean meta-test accuracy scores on 5-way miniImageNet classification over 5 runs using a Conv-4 backbone with 32 channels. The 95% confidence intervals are displayed as ± x. * Flennerhag et al. (2020) only reported the performance of Warp-MAML with 128 feature maps per convolutional block instead of 32, as displayed in the table.

We re-implemented the baselines to ensure a fair comparison in the used setting, and because the code of Warp-MAML has not been made available for other researchers. To verify our re-implementations of the baselines (T-Net, MT-Net, and Warp-MAML), we compare the reported performances to the ones that we obtain. The results of the image classification experiments are displayed in Table C.1. As we can see, there are minor differences between the reported performances and our local reproduction of their results. Also with the original code of T-Net and MT-Net, we were unable to reproduce their results. Other people have encountered similar issues reproducing the reported numbers of meta-learning techniques, including MAML, T-Net, and MT-Net.[1]

[1]There is an open issue on the GitHub repository of MT-Net about the inability to reproduce their reported results

C.2 Cross-domain few-shot image classification

In Table C.2, we show the cross-domain few-shot learning classification results when using 64 channels with the Conv-4 backbone. Also in this case, SAP outperforms other tested baselines. We also note that the performance of SAP is improved when using 64 channels compared with 32 (see Section 6.5.5).

	MIN → CUB		Tiered → CUB	
	1-shot	5-shot	1-shot	5-shot
MAML	37.1 ± 0.3	53.7 ± 0.3	38.8 ± 0.3	56.8 ± 0.3
T-Net	38.3 ± 0.3	OOM	39.9 ± 0.3	OOM
MT-Net	37.3 ± 0.3	OOM	39.1 ± 0.3	OOM
Warp-MAML	40.7 ± 0.3	56.2 ± 0.3	42.5 ± 0.3	58.9 ± 0.3
SAP (ours)	41.6 ± 0.3	57.8 ± 0.3	43.3 ± 0.3	64.3 ± 0.3

Table C.2: Average cross-domain meta-test accuracy scores over 5 runs using a 64-channel Conv-4 backbone. Techniques trained on tasks from one data set were evaluated on tasks from another data set. The 95% confidence intervals are displayed as ± x.

C.3 The effect of hard pruning

Table C.3 displays the effect of hard pruning when using 64 channels instead of 32. As we can see, hard pruning is slightly beneficial, but again, not significantly.

	miniImageNet		tieredImageNet	
	1-shot	5-shot	1-shot	5-shot
No pruning	52.8 ± 0.8	67.4 ± 0.4	54.5 ± 0.8	71.3 ± 0.4
Top-1	52.8 ± 0.8	**67.6** ± 0.4	**55.1** ± 0.8	**72.7** ± 0.4
Top-2	**52.9** ± 0.8	**67.6** ± 0.4	54.1 ± 0.8	**72.7** ± 0.4
Top-3	52.6 ± 0.8	67.4 ± 0.4	55.0 ± 0.8	72.4 ± 0.4

Table C.3: Mean meta-test accuracy scores on 5-way miniImageNet and tieredImageNet classification with 95% confidence intervals computed over 5 different runs. We used a Conv-4 backbone with 64 channels for these results.

C.4 The learned subspaces for image classification

Figure C.1 displays the learned activation strengths of SAP on 5-way 1-shot miniImageNet using Conv-4 with 64 channels. Similar patterns are observed for the 32-channel case.

on miniImageNet. See `https://github.com/yoonholee/MT-net/issues/5`. Other researchers such as Antoniou et al. (2019) have also reported issues reproducing MAML.

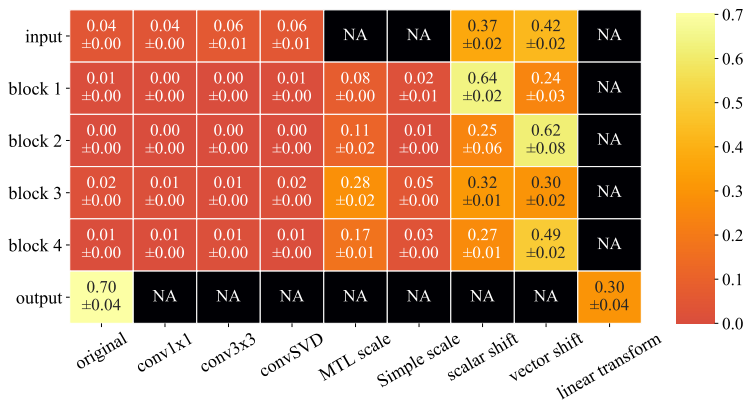


Figure C.1: The importance of the different subspaces/operations in SAP on 5-way 1-shot miniImageNet using Conv-4 with 64 channels. The results are averaged across 5 runs with different random seeds and the standard deviations are shown as $\pm x$. NA entries indicate that these operations were not in the candidate pool for that layer. Simple scalar shift and vector shift operations obtain the highest activation strengths throughout the convolutional network.

Summary

Deep neural networks have demonstrated impressive performance in various domains, often achieving human-level or super-human capabilities. However, their success heavily relies on the availability of extensive training data, which becomes problematic in domains where data collection is challenging, costly, or privacy-sensitive. Enhancing the data efficiency of deep neural networks and overcoming these limitations is of utmost importance. By doing so, we can unlock the full potential of these networks, enabling them to learn and adapt effectively even when presented with limited data. This improvement also paves the way for their deployment in resource-constrained environments, promoting the democratization of deep learning by making it more accessible and applicable across various domains.

One of the potential causes of their inefficient learning ability is the fact that deep neural networks are often trained from scratch in an end-to-end way. Deep meta-learning has emerged as a rapidly growing field to improve the learning efficiency of deep neural networks by endowing them with the ability to reuse prior experiences and knowledge. This dissertation focuses specifically on the application of deep meta-learning in few-shot learning scenarios, where networks need to rapidly adapt to new tasks with only a limited number of examples.

Despite recent progress in few-shot learning, the underlying principles that drive the success of meta-learning algorithms are still poorly understood, impeding algorithm development and design choices. In this dissertation, we take steps to bridge this knowledge gap by gaining a comprehensive understanding of the fundamental principles of popular deep meta-learning algorithms, enabling more informed algorithm development, and establishing robust theoretical foundations. Additionally, this work explores the integration of theoretical principles into practical algorithm design to enhance the capabilities of deep meta-learning approaches. By addressing these research gaps, this dissertation aims to advance the field, paving the way for more effective and principled meta-learning techniques that offer broader applicability and superior performance. Below is a brief outline of the dissertation.

Chapter 2 serves as an extensive introduction and overview, providing readers with a solid theoretical foundation for understanding deep meta-learning algorithms. We delve into key methods and categorize them into three main categories: i) metric-based techniques, ii) model-based techniques, and iii) optimization-based techniques. By exploring these approaches, we aim to provide a holistic understanding of the diverse methodologies employed in deep meta-learning. Furthermore, we identify and discuss the primary open challenges in the field. These challenges include the need for performance evaluations on heterogeneous benchmarks to ensure the robustness and generalizability of meta-learning algorithms.

In Chapter 3, we investigate an empirically observed performance gap between two popular and highly related deep meta-learning algorithms: the meta-learner LSTM and MAML. We found

this performance gap surprising based on our work in Chapter 2 as the meta-learner LSTM is more expressive than MAML and could, in theory, emulate the behavior of MAML. To gain a deeper understanding of this performance gap, we introduce a novel algorithm called TURTLE. The design and analysis of TURTLE reveal that the notable performance gap can be attributed to the influence of second-order gradients. We find that second-order gradients can also significantly increase the accuracy of the meta-learner LSTM with slight modifications of the inputs provided to the LSTM.

A related method to deep meta-learning is the transfer learning method commonly known as pre-training and fine-tuning. In Chapter 4, we investigate the observed performance differences between finetuning, MAML, and another meta-learning technique called Reptile. We present evidence indicating that MAML and Reptile exhibit a tendency to specialize in rapidly adapting to low-data regimes characterized by similar data distributions as the ones used during training. Our findings highlight the importance of both the output layer and the presence of noisy training conditions induced by data scarcity in the few-shot learning setting. These factors contribute significantly to enabling the specialization observed in MAML and Reptile. Additionally, we demonstrate that the pre-trained features obtained through the finetuning baseline exhibit greater diversity and discriminative power compared to those learned by MAML and Reptile. This lack of diversity and distribution specialization in MAML and Reptile may hinder their ability to generalize effectively to target tasks that differ significantly from the observed training tasks. In contrast, finetuning can leverage the diverse set of learned features to adapt more successfully to such distant target tasks.

In Chapter 5, we revisit a classical LSTM approach to deep meta-learning, where the idea is to feed a training dataset into an LSTM and to condition the predictions of query inputs on the resulting hidden state. This approach is known to be maximally expressive, that is, the LSTM could learn to implement any learning algorithm. Despite the promising results of this approach on small problems and on reinforcement learning problems, the approach has received little attention in the supervised few-shot learning setting. We show that LSTM outperforms the popular meta-learning technique MAML on a simple few-shot sine wave regression benchmark, but that LSTM, expectedly, falls short on more complex few-shot image classification benchmarks. We identify two potential factors contributing to the observed limitations and propose a novel method called Outer Product LSTM (OP-LSTM) to address these issues effectively. OP-LSTM surpasses the performance of plain LSTM and exhibits substantial performance gains. While these results alone do not set a new state-of-the-art, the advances of OP-LSTM are orthogonal to other advances in the field of meta-learning, yielding new insights in how LSTM works in image classification, allowing for a whole range of new research directions.

In Chapter 6, we investigate whether the integration of the fact that more expressive models are more likely to overfit can improve the few-shot learning performance by meta-learning which parameters to adjust. To investigate this, we propose *Subspace Adaptation Prior* (SAP), a novel gradient-based meta-learning algorithm that jointly learns good initialization parameters (prior knowledge) and layer-wise *parameter subspaces* in the form of operation subsets that should be adaptable. In this way, SAP can learn which operation subsets to adjust with gradient descent based on the underlying task distribution, simultaneously decreasing the risk of overfitting when learning new tasks. We demonstrate that this ability is helpful as SAP yields superior or competitive performance in few-shot image classification settings (gains between 0.1% and 3.9% in accuracy). Analysis of the learned subspaces demonstrates that low-dimensional operations often yield high activation strengths, indicating that they may be important for achieving good few-shot learning performance.

As such, in this dissertation, we have analyzed and performed empirical validation of various meta-learning systems, including MAML, Reptile, finetuning and various LSTM-based approaches. Additionally, we have explored the integration of theoretical principles for practical algorithm development. In short, we have made a small step toward understanding deep meta-learning algorithms, paving the way for more robust and principled meta-learning techniques with broader applicability and superior performance.

Dutch summary

Diepe neurale netwerken hebben indrukwekkende prestaties laten zien in verschillende domeinen, vaak met capaciteiten op menselijk of bovenmenselijk niveau. Hun succes hangt echter sterk af van de beschikbaarheid van uitgebreide trainingsdata, wat problematisch wordt in domeinen waar het verzamelen van zulke data uitdagend, kostbaar of privacygevoelig is. Het verbeteren van de data-efficiëntie van diepe neurale netwerken en het overkomen van deze beperkingen is van groot belang. Door dit te doen, kunnen we het volledige potentieel van deze netwerken ontsluiten, waardoor ze effectief kunnen leren en zich kunnen aanpassen, zelfs wanneer ze met beperkte gegevens worden geconfronteerd. Deze verbetering maakt ook de weg vrij voor hun inzet in omgevingen met beperkte middelen, waardoor de democratisering van *deep learning* wordt bevorderd door het toegankelijker en toepasbaarder te maken in verschillende domeinen.

Een van de mogelijke oorzaken van hun inefficiënte leervermogen is het feit dat diepe neurale netwerken vaak zonder enige voorkennis worden getraind. *Deep meta-learning* is naar voren gekomen als een snelgroeiend veld om de leerefficiëntie van diepe neurale netwerken te verbeteren door ze de mogelijkheid te geven om eerdere ervaringen en kennis te hergebruiken. Dit proefschrift richt zich specifiek op de toepassing van *deep meta-learning* in scenario's voor leren met weinig trainingsdata, waarbij netwerken zich snel moeten aanpassen aan nieuwe taken met slechts een beperkt aantal voorbeelden per taak.

Ondanks de recente vooruitgang op dit gebied van *few-shot learning*, worden de onderliggende principes die het succes van meta-learning-algoritmen stimuleren nog steeds slecht begrepen, wat de ontwikkeling van algoritmen en ontwerpkeuzes belemmert. In dit proefschrift nemen we stappen om deze kenniskloof te overbruggen door een uitgebreid begrip te verwerven van de fundamentele principes van populaire *deep meta-learning* algoritmen, waardoor een beter geïnformeerde ontwikkeling van algoritmen mogelijk wordt en een robuuste theoretische basis wordt gelegd. Bovendien onderzoekt dit werk de integratie van theoretische principes in praktisch algoritmeontwerp om de prestaties van *deep meta-learning* technieken te verbeteren. Door deze onderzoekshiaten aan te pakken, beoogt dit proefschrift het veld vooruit helpen en de weg vrij te maken voor meer effectieve en principiële *meta-learning* technieken die een bredere toepasbaarheid en superieure prestaties bieden. Hieronder volgt een kort overzicht van het proefschrift.

Hoofdstuk 2 dient als een uitgebreide inleiding en overzicht en biedt lezers een solide theoretische basis voor het begrijpen van *deep meta-learning* algoritmen. We verdiepen ons in de belangrijkste methoden en categoriseren ze in drie hoofdcategorieën: i) op metrische gegevens gebaseerde technieken, ii) op modellen gebaseerde technieken en iii) op optimalisatie gebaseerde technieken. Door deze benaderingen te onderzoeken, willen we een holistisch begrip bieden van de diverse methodologieën die worden gebruikt bij *deep meta-learning*. Verder identificeren en bespreken we de belangrijkste

openstaande uitdagingen in het veld. Deze uitdagingen omvatten de behoefte aan prestatie-evaluaties op heterogene benchmarks om de robuustheid en generaliseerbaarheid van *meta-learning* algoritmen te waarborgen.

In Hoofdstuk 3 onderzoeken we een empirisch waargenomen prestatiekloof tussen twee populaire en sterk verwante *deep meta-learning* algoritmen: de meta-learning LSTM en MAML. We vonden deze prestatiekloof verrassend op basis van ons werk in Hoofdstuk 2 omdat de meta-learning LSTM expressiever is dan MAML en in theorie het gedrag van MAML zou kunnen nabootsen. Om deze prestatiekloof beter te begrijpen, introduceren we een nieuw algoritme genaamd TURTLE. Het ontwerp en de analyse van TURTLE laten zien dat de opmerkelijke prestatiekloof kan worden toegeschreven aan de invloed van gradiënten van de tweede orde. We vinden dat gradiënten van de tweede orde ook de nauwkeurigheid van de meta-learning LSTM aanzienlijk kunnen verhogen met kleine aanpassingen van de invoer die aan de LSTM wordt geleverd.

Een verwante methode aan *deep meta-learning* is de overdrachtsleermethode die algemeen bekend staat als pre-training en fine-tuning. In Hoofdstuk 4 onderzoeken we de waargenomen prestatieverschillen tussen finetuning, MAML en een andere *meta-learning* techniek genaamd Reptile. We presenteren resultaten die aangeven dat MAML en Reptile de neiging vertonen om zich te specialiseren in snelle aanpassing aan *low-data-regimes* die worden gekenmerkt door vergelijkbare datadistributies als degene die tijdens de training worden gebruikt. Onze bevindingen benadrukken het belang van zowel de uitvoerlaag als de aanwezigheid van *noisy* trainingsomstandigheden veroorzaakt door gegevensschaarste in de *few-shot* leeromgeving. Deze factoren dragen aanzienlijk bij aan het mogelijk maken van de specialisatie die wordt waargenomen in MAML en Reptile. Bovendien tonen we aan dat de vooraf getrainde functies die zijn verkregen via de finetuning-baseline een grotere diversiteit en onderscheidend vermogen vertonen in vergelijking met die geleerd door MAML en Reptile. Dit gebrek aan diversiteit en distributiespecialisatie in MAML en Reptile kan hun vermogen belemmeren om effectief te generaliseren naar doeltaken die aanzienlijk verschillen van de geobserveerde trainingstaken. Finetuning daarentegen kan gebruikmaken van de diverse reeks aangeleerde functies om zich met meer succes aan te passen aan dergelijke verre doeltaken.

In Hoofdstuk 5 gaan we terug naar een klassieke LSTM-benadering van *deep meta-learning*, waarbij het idee is om een trainingsdataset in een LSTM in te voeren en de voorspellingen voor nieuwe datapunten te conditioneren op de resulterende verborgen staat van het netwerk. Het is bekend dat deze benadering maximaal expressief is, dat wil zeggen dat de LSTM zou kunnen leren om elk leeralgoritme te implementeren. Ondanks de veelbelovende resultaten van deze aanpak bij kleine problemen en bij *reinforcement learning* problemen, heeft de aanpak weinig aandacht gekregen in de *supervised learning* hoek. We laten zien dat LSTM beter presteert dan de populaire *meta-learning* techniek MAML op een eenvoudige sinusgolf regressiebenchmark, maar dat LSTM zoals verwacht tekort schiet op complexere *few-shot* beeldclassificatie benchmarks. We identificeren twee potentiële factoren die bijdragen aan de waargenomen beperkingen en stellen een nieuwe methode voor, *Outer Product LSTM* (OP-LSTM) genaamd, om deze problemen effectief aan te pakken. OP-LSTM overtreft de prestaties van gewone LSTM en vertoont aanzienlijke prestatieverbeteringen. Hoewel deze resultaten geen nieuwe state-of-the-art vormen, staan de vorderingen van OP-LSTM orthogonaal op andere vorderingen op het gebied van *meta-learning*, wat nieuwe inzichten oplevert in hoe LSTM werkt bij beeldclassificatie, en nieuwe onderzoeksrichtingen oplevert.

In Hoofdstuk 6 onderzoeken we of de integratie van het feit dat meer expressieve modellen eerder geneigd zijn tot overfitten, de leerprestaties van neurale netwerken kan verbeteren door te meta-leren welke parameters moeten worden aangepast. Om dit te onderzoeken, stellen we *Subspace Adaptation Prior* (SAP) voor, een nieuw op gradiënt gebaseerd *meta-learning* algoritme dat gezamenlijk goede

initialisatieparameters (voorkennis) en laaggewijze *parameter subspaces* leert in de vorm van subsets van bewerkingen die aanpasbaar moeten zijn. Op deze manier kan SAP leren welke bewerkingssubsets moeten worden aangepast op basis van de onderliggende taakverdeling, terwijl tegelijkertijd het risico van overfitting wordt verminderd bij het aanleren van nieuwe taken. We tonen aan dat deze mogelijkheid nuttig is, aangezien SAP superieure of concurrerende prestaties levert bij classificatie-instellingen voor weinig opnamen (winsten tussen 0,1% en 3,9% in nauwkeurigheid). Analyse van de aangeleerde deelruimten laat zien dat laag-dimensionale operaties vaak hoge activeringssterktes opleveren, wat aangeeft dat ze belangrijk kunnen zijn voor het bereiken van goede 'few-shot' leerprestaties.

Als zodanig hebben we in dit proefschrift analyse en empirische validatie uitgevoerd van verschillende *meta-learning* systemen, waaronder MAML, Reptile, finetuning en verschillende op LSTM gebaseerde benaderingen. Daarnaast hebben we de integratie van theoretische principes voor praktische algoritme-ontwikkeling onderzocht. Kortom, we hebben een kleine stap gezet in de richting van een beter begrip van *deep meta-learning*-algoritmen, waarmee we de weg hebben vrijgemaakt voor robuustere en meer principiële meta-learning-technieken met een bredere toepasbaarheid en superieure prestaties.

Acknowledgements

Here, I would like to thank a few people who have contributed toward this dissertation or have worked with me in the process. I restrict myself here to people who have had a direct influence on this dissertation in a professional manner and omit people to whom I am indebted for their immense support throughout not only this dissertation but life in general. People who belong to the latter category are already aware of my gratefulness.

First and foremost, I would like to thank my two supervisors, Aske and Jan, for providing me with the opportunity to work as a PhD candidate, which had been a dream for a long time. Moreover, I thank you for the freedom that you have given me in pursuing my passion: coming up with novel research ideas from scratch, and researching them autonomously throughout. Also, I would like to thank you for allowing me to take on a large responsibility in teaching, which has led me to be a co-teacher for Automated Machine Learning, and Algorithms and Data Structures. Moreover, I have also been involved in the supervision of three master thesis projects. I thoroughly enjoyed these teaching activities with all my heart. Lastly, I would like to thank you for supporting me in finishing the dissertation.

I would like to thank Matthias Mueller-Brockhausen and Zhao Yang for working with me on a project that aimed to meta-learn a loss function. Unfortunately, the project did not survive, but it was nice working with you. Also, I would like to thank Thomas Moerland for collaborating with me on the OP-LSTM project and sharing the same passion for human intelligence and the bigger picture.

Thanks to Herke van Hoof for an insightful discussion on LLAMA. Thanks to Pavel Brazdil for his encouragement and feedback on a preliminary version of our survey paper.

This work was performed using the compute resources from the Academic Leiden Interdisciplinary Cluster Environment (ALICE) provided by Leiden University and the Dutch national e-infrastructure with the support of SURF Cooperative.

Curriculum Vitae

Education

Leiden University *March 2021 - August 2023*
Doctor of Philosophy (PhD) in Computer Science,
Dissertation: Understanding Deep Meta-Learning

Leiden University *September 2019 - February 2021*
Master of Computer Science: Data Science, *Summa Cum Laude*
Thesis: Revisiting Learned Optimizers in Few-Shot Settings

Utrecht University *September 2016 - July 2019*
Bachelor of Science in Artificial Intelligence, *Cum Laude*
Thesis: Machine Learning in the Fight against Internet Toxicity

University of Applied Sciences Leiden *September 2015 - July 2016*
First-year diploma Computer Science

List of publications

- Huisman, M., Moerland, T. M., Plaat, A., & van Rijn, J. N. (2023). Are LSTMs good few-shot learners? Machine Learning, 112, 4635–4662. Springer.
- Huisman, M., Plaat, A. & van Rijn, J. N. (2023). Subspace Adaptation Prior for Few-Shot Learning. Machine Learning. Springer.
- Huisman, M., van Rijn, J. N., & Plaat, A. (2023). Understanding Transfer Learning and Gradient-Based Meta-Learning Techniques. Accepted for publication in Machine Learning. Springer.
- Ullah, I., Carrión-Ojeda, D., Escalera, S., Guyon, I., Huisman, M., Mohr, F., ... & Vu, P. A. (2022). Meta-album: Multi-domain meta-dataset for few-shot image classification. Advances in Neural Information Processing Systems, 35, 3232-3247.
- El Baz, A., Ullah, I., Alcobaça, E., Carvalho, A.C., Chen, H., Ferreira, F., Gouk, H., Guan, C., Guyon, I., Hospedales, T. & Hu, S. (2021). Lessons learned from the NeurIPS 2021 MetaDL challenge: Backbone fine-tuning without episodic meta-learning dominates for few-shot learning image classification. In NeurIPS 2021 Competition and Demonstration Track.
- Huisman, M., Plaat, A. & van Rijn, J. N. (2021). A preliminary study on the feature representations of transfer learning and gradient-based meta-learning techniques. In Fifth Workshop on Meta-Learning at the Conference on Neural Information Processing Systems.
- Huisman, M., Plaat, A., & van Rijn, J. N. (2022). Stateless neural meta-learning using second-order gradients. Machine Learning, 111(9), 3227-3244. Springer.
- Huisman, M., van Rijn, J. N., Plaat, A. (2022). Metalearning for deep neural networks. In Brazdil, P., van Rijn, J. N., Soares, C., & Vanschoren, J. (2nd Edition). Metalearning: Applications to Automated Machine Learning and Data Mining, chapter 13, pages 237-267. Springer Nature.
- Huisman, M., van Rijn, J. N., & Plaat, A. (2021). A survey of deep meta-learning. Artificial Intelligence Review, 54(6), 4483-4541. Springer.

SIKS dissertation series

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior

	26	Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
	27	Wen Li (TUD), Understanding Geo-spatial Information on Social Media
	28	Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
	29	Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
	30	Ruud Mattheij (UvT), The Eyes Have It
	31	Mohammad Khelghati (UT), Deep web content monitoring
	32	Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
	33	Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
	34	Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
	35	Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
	36	Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
	37	Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
	38	Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
	39	Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
	40	Christian Detweiler (TUD), Accounting for Values in Design
	41	Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
	42	Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
	43	Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
	44	Thibault Sellam (UVA), Automatic Assistants for Database Exploration
	45	Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
	46	Jorge Gallego Perez (UT), Robots to Make you Happy
	47	Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
	48	Tanja Buttler (TUD), Collecting Lessons Learned
	49	Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
	50	Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains
2017	01	Jan-Jaap Oerlemans (UL), Investigating Cybercrime
	02	Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
	03	Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
	04	Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
	05	Mahdieh Shadi (UVA), Collaboration Behavior
	06	Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
	07	Roel Bertens (UU), Insight in Information: from Abstract to Anomaly

08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TUE), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UVA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VU), Logics for causal inference under uncertainty
23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VU), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (UvT), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VU), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR

	40	Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
	41	Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
	42	Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
	43	Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
	44	Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
	45	Bas Testerink (UU), Decentralized Runtime Norm Enforcement
	46	Jan Schneider (OU), Sensor-based Learning Support
	47	Jie Yang (TUD), Crowd Knowledge Creation Acceleration
	48	Angel Suarez (OU), Collaborative inquiry-based learning
2018	01	Han van der Aa (VUA), Comparing and Aligning Process Representations
	02	Felix Mannhardt (TUE), Multi-perspective Process Mining
	03	Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
	04	Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
	05	Hugo Huurdeman (UVA), Supporting the Complex Dynamics of the Information Seeking Process
	06	Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
	07	Jieting Luo (UU), A formal account of opportunism in multi-agent systems
	08	Rick Smetsers (RUN), Advances in Model Learning for Software Systems
	09	Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
	10	Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
	11	Mahdi Sargolzaei (UVA), Enabling Framework for Service-oriented Collaborative Networks
	12	Xixi Lu (TUE), Using behavioral context in process mining
	13	Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
	14	Bart Joosten (UVT), Detecting Social Signals with Spatiotemporal Gabor Filters
	15	Naser Davarzani (UM), Biomarker discovery in heart failure
	16	Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
	17	Jianpeng Zhang (TUE), On Graph Sample Clustering
	18	Henriette Nakad (UL), De Notaris en Private Rechtspraak
	19	Minh Duc Pham (VUA), Emergent relational schemas for RDF
	20	Manxia Liu (RUN), Time and Bayesian Networks
	21	Aad Slootmaker (OUN), EMERGO: a generic platform for authoring and playing scenario-based serious games
	22	Eric Fernandes de Mello Araújo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
	23	Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
	24	Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
	25	Riste Gligorov (VUA), Serious Games in Audio-Visual Collections

	26	Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
	27	Maikel Leemans (TUE), Hierarchical Process Mining for Scalable Software Analysis
	28	Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
	29	Yu Gu (UVT), Emotion Recognition from Mandarin Speech
	30	Wouter Beek, The "K" in "semantic web" stands for "knowledge": scaling semantics to the web
2019	01	Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
	02	Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
	03	Eduardo Gonzalez Lopez de Murillas (TUE), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
	04	Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
	05	Sebastiaan van Zelst (TUE), Process Mining with Streaming Data
	06	Chris Dijkshoorn (VU), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
	07	Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
	08	Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
	09	Fahimeh Alizadeh Moghaddam (UVA), Self-adaptation for energy efficiency in software systems
	10	Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
	11	Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
	12	Jacqueline Heinerman (VU), Better Together
	13	Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
	14	Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
	15	Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
	16	Guangming Li (TUE), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
	17	Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
	18	Gerard Wagenaar (UU), Artefacts in Agile Team Communication
	19	Vincent Koeman (TUD), Tools for Developing Cognitive Agents
	20	Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
	21	Cong Liu (TUE), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
	22	Martin van den Berg (VU),Improving IT Decisions with Enterprise Architecture
	23	Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
	24	Anca Dumitrache (VU), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
	25	Emiel van Miltenburg (VU), Pragmatic factors in (automatic) image description
	26	Prince Singh (UT), An Integration Platform for Synchromodal Transport
	27	Alessandra Antonaci (OUN), The Gamification Design Process applied to (Massive) Open Online Courses

	28	Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
	29	Daniel Formolo (VU), Using virtual agents for simulation and training of social skills in safety-critical circumstances
	30	Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
	31	Milan Jelisavcic (VU), Alive and Kicking: Baby Steps in Robotics
	32	Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
	33	Anil Yaman (TUE), Evolution of Biologically Inspired Learning in Artificial Neural Networks
	34	Negar Ahmadi (TUE), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
	35	Lisa Facey-Shaw (OUN), Gamification with digital badges in learning programming
	36	Kevin Ackermans (OUN), Designing Video-Enhanced Rubrics to Master Complex Skills
	37	Jian Fang (TUD), Database Acceleration on FPGAs
	38	Akos Kadar (OUN), Learning visually grounded and multilingual representations
2020	01	Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour
	02	Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
	03	Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
	04	Maarten van Gompel (RUN), Context as Linguistic Bridges
	05	Yulong Pei (TUE), On local and global structure mining
	06	Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
	07	Wim van der Vegt (OUN), Towards a software architecture for reusable game components
	08	Ali Mirsoleimani (UL),Structured Parallel Programming for Monte Carlo Tree Search
	09	Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research
	10	Alifah Syamsiyah (TUE), In-database Preprocessing for Process Mining
	11	Sepideh Mesbah (TUD), Semantic-Enhanced Training Data AugmentationMethods for Long-Tail Entity Recognition Models
	12	Ward van Breda (VU), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment
	13	Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming
	14	Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases
	15	Konstantinos Georgiadis (OUN), Smart CAT: Machine Learning for Configurable Assessments in Serious Games
	16	Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling
	17	Daniele Di Mitri (OUN), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
	18	Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
	19	Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
	20	Albert Hankel (VU), Embedding Green ICT Maturity in Organisations
	21	Karine da Silva Miras de Araujo (VU), Where is the robot?: Life as it could be
	22	Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar

23 Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
24 Lenin da Nóbrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots
25 Xin Du (TUE), The Uncertainty in Exceptional Model Mining
26 Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization
27 Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
28 Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality
29 Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
30 Bob Zadok Blok (UL), Creatief, Creatiever, Creatiefst
31 Gongjin Lan (VU), Learning better – From Baby to Better
32 Jason Rhuggenaath (TUE), Revenue management in online markets: pricing and online advertising
33 Rick Gilsing (TUE), Supporting service-dominant business model evaluation in the context of business model innovation
34 Anna Bon (MU), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
35 Siamak Farshidi (UU), Multi-Criteria Decision-Making in Software Production

---

2021 01 Francisco Xavier Dos Santos Fonseca (TUD),Location-based Games for Social Interaction in Public Space
02 Rijk Mercuur (TUD), Simulating Human Routines: Integrating Social Practice Theory in Agent-Based Models
03 Seyyed Hadi Hashemi (UVA), Modeling Users Interacting with Smart Devices
04 Ioana Jivet (OU), The Dashboard That Loved Me: Designing adaptive learning analytics for self-regulated learning
05 Davide Dell'Anna (UU), Data-Driven Supervision of Autonomous Systems
06 Daniel Davison (UT), "Hey robot, what do you think?" How children learn with a social robot
07 Armel Lefebvre (UU), Research data management for open science
08 Nardie Fanchamps (OU), The Influence of Sense-Reason-Act Programming on Computational Thinking
09 Cristina Zaga (UT), The Design of Robothings. Non-Anthropomorphic and Non-Verbal Robots to Promote Children's Collaboration Through Play
10 Quinten Meertens (UvA), Misclassification Bias in Statistical Learning
11 Anne van Rossum (UL), Nonparametric Bayesian Methods in Robotic Vision
12 Lei Pi (UL), External Knowledge Absorption in Chinese SMEs
13 Bob R. Schadenberg (UT), Robots for Autistic Children: Understanding and Facilitating Predictability for Engagement in Learning
14 Negin Samaeemofrad (UL), Business Incubators: The Impact of Their Support
15 Onat Ege Adali (TU/e), Transformation of Value Propositions into Resource Re-Configurations through the Business Services Paradigm
16 Esam A. H. Ghaleb (UM), Bimodal emotion recognition from audio-visual cues
17 Dario Dotti (UM), Human Behavior Understanding from motion and bodily cues using deep neural networks

	18	Remi Wieten (UU), Bridging the Gap Between Informal Sense-Making Tools and Formal Systems - Facilitating the Construction of Bayesian Networks and Argumentation Frameworks
	19	Roberto Verdecchia (VU), Architectural Technical Debt: Identification and Management
	20	Masoud Mansoury (TU/e), Understanding and Mitigating Multi-Sided Exposure Bias in Recommender Systems
	21	Pedro Thiago Timbó Holanda (CWI), Progressive Indexes
	22	Sihang Qiu (TUD), Conversational Crowdsourcing
	23	Hugo Manuel Proença (LIACS), Robust rules for prediction and description
	24	Kaijie Zhu (TUE), On Efficient Temporal Subgraph Query Processing
	25	Eoin Martino Grua (VUA), The Future of E-Health is Mobile: Combining AI and Self-Adaptation to Create Adaptive E-Health Mobile Applications
	26	Benno Kruit (CWI & VUA), Reading the Grid: Extending Knowledge Bases from Human-readable Tables
	27	Jelte van Waterschoot (UT), Personalized and Personal Conversations: Designing Agents Who Want to Connect With You
	28	Christoph Selig (UL), Understanding the Heterogeneity of Corporate Entrepreneurship Programs
2022	01	Judith van Stegeren (UT), Flavor text generation for role-playing video games
	02	Paulo da Costa (TU/e), Data-driven Prognostics and Logistics Optimisation: A Deep Learning Journey
	03	Ali el Hassouni (VUA), A Model A Day Keeps The Doctor Away: Reinforcement Learning For Personalized Healthcare
	04	Ünal Aksu (UU), A Cross-Organizational Process Mining Framework
	05	Shiwei Liu (TU/e), Sparse Neural Network Training with In-Time Over-Parameterization
	06	Reza Refaei Afshar (TU/e), Machine Learning for Ad Publishers in Real Time Bidding
	07	Sambit Praharaj (OU), Measuring the Unmeasurable? Towards Automatic Co-located Collaboration Analytics
	08	Maikel L. van Eck (TU/e), Process Mining for Smart Product Design
	09	Oana Andreea Inel (VUA), Understanding Events: A Diversity-driven Human-Machine Approach
	10	Felipe Moraes Gomes (TUD), Examining the Effectiveness of Collaborative Search Engines
	11	Mirjam de Haas (UT), Staying engaged in child-robot interaction, a quantitative approach to studying preschoolers' engagement with robots and tasks during second-language tutoring
	12	Guanyi Chen (UU), Computational Generation of Chinese Noun Phrases
	13	Xander Wilcke (VUA), Machine Learning on Multimodal Knowledge Graphs: Opportunities, Challenges, and Methods for Learning on Real-World Heterogeneous and Spatially-Oriented Knowledge
	14	Michiel Overeem (UU), Evolution of Low-Code Platforms
	15	Jelmer Jan Koorn (UU), Work in Process: Unearthing Meaning using Process Mining
	16	Pieter Gijsbers (TU/e), Systems for AutoML Research
	17	Laura van der Lubbe (VUA), Empowering vulnerable people with serious games and gamification
	18	Paris Mavromoustakos Blom (TiU), Player Affect Modelling and Video Game Personalisation
	19	Bilge Yigit Ozkan (UU), Cybersecurity Maturity Assessment and Standardisation
	20	Fakhra Jabeen (VUA), Dark Side of the Digital Media - Computational Analysis of Negative Human Behaviors on Social Media

	21	Seethu Mariyam Christopher (UM), Intelligent Toys for Physical and Cognitive Assessments
	22	Alexandra Sierra Rativa (TiU), Virtual Character Design and its potential to foster Empathy, Immersion, and Collaboration Skills in Video Games and Virtual Reality Simulations
	23	Ilir Kola (TUD), Enabling Social Situation Awareness in Support Agents
	24	Samaneh Heidari (UU), Agents with Social Norms and Values - A framework for agent based social simulations with social norms and personal values
	25	Anna L.D. Latour (LU), Optimal decision-making under constraints and uncertainty
	26	Anne Dirkson (LU), Knowledge Discovery from Patient Forums: Gaining novel medical insights from patient experiences
	27	Christos Athanasiadis (UM), Emotion-aware cross-modal domain adaptation in video sequences
	28	Onuralp Ulusoy (UU), Privacy in Collaborative Systems
	29	Jan Kolkmeier (UT), From Head Transform to Mind Transplant: Social Interactions in Mixed Reality
	30	Dean De Leo (CWI), Analysis of Dynamic Graphs on Sparse Arrays
	31	Konstantinos Traganos (TU/e), Tackling Complexity in Smart Manufacturing with Advanced Manufacturing Process Management
	32	Cezara Pastrav (UU), Social simulation for socio-ecological systems
	33	Brinn Hekkelman (CWI/TUD), Fair Mechanisms for Smart Grid Congestion Management
	34	Nimat Ullah (VUA), Mind Your Behaviour: Computational Modelling of Emotion & Desire Regulation for Behaviour Change
	35	Mike E.U. Ligthart (VUA), Shaping the Child-Robot Relationship: Interaction Design Patterns for a Sustainable Interaction
2023	01	Bojan Simoski (VUA), Untangling the Puzzle of Digital Health Interventions
	02	Mariana Rachel Dias da Silva (TiU), Grounded or in flight? What our bodies can tell us about the whereabouts of our thoughts
	03	Shabnam Najafian (TUD), User Modeling for Privacy-preserving Explanations in Group Recommendations
	04	Gineke Wiggers (UL), The Relevance of Impact: bibliometric-enhanced legal information retrieval
	05	Anton Bouter (CWI), Optimal Mixing Evolutionary Algorithms for Large-Scale Real-Valued Optimization, Including Real-World Medical Applications
	06	António Pereira Barata (UL), Reliable and Fair Machine Learning for Risk Assessment
	07	Tianjin Huang (TU/e), The Roles of Adversarial Examples on Trustworthiness of Deep Learning
	08	Lu Yin (TU/e), Knowledge Elicitation using Psychometric Learning
	09	Xu Wang (VUA), Scientific Dataset Recommendation with Semantic Techniques
	10	Dennis J.N.J. Soemers (UM), Learning State-Action Features for General Game Playing
	11	Fawad Taj (VUA), Towards Motivating Machines: Computational Modeling of the Mechanism of Actions for Effective Digital Health Behavior Change Applications
	12	Tessel Bogaard (VUA), Using Metadata to Understand Search Behavior in Digital Libraries
	13	Injy Sarhan (UU), Open Information Extraction for Knowledge Representation
	14	Selma Čaušević (TUD), Energy resilience through self-organization
	15	Alvaro Henrique Chaim Correia (TU/e), Insights on Learning Tractable Probabilistic Graphical Models
	16	Peter Blomsma (TiU), Building Embodied Conversational Agents: Observations on human nonverbal behaviour as a resource for the development of artificial characters

17 Meike Nauta (UT), Explainable AI and Interpretable Computer Vision – From Oversight to Insight
18 Gustavo Penha (TUD), Designing and Diagnosing Models for Conversational Search and Recommendation
19 George Aalbers (TiU), Digital Traces of the Mind: Using Smartphones to Capture Signals of Well-Being in Individuals
20 Arkadiy Dushatskiy (TUD), Expensive Optimization with Model-Based Evolutionary Algorithms applied to Medical Image Segmentation using Deep Learning
21 Gerrit Jan de Bruin (UL), Network Analysis Methods for Smart Inspection in the Transport Domain
22 Alireza Shojaifar (UU), Volitional Cybersecurity
23 Theo Theunissen (UU), Documentation in Continuous Software Development
24 Agathe Balayn (TUD), Practices Towards Hazardous Failure Diagnosis in Machine Learning
25 Jurian Baas (UU), Entity Resolution on Historical Knowledge Graphs
26 Loek Tonnaer (TU/e), Linearly Symmetry-Based Disentangled Representations and their Out-of-Distribution Behaviour
27 Ghada Sokar (TU/e), Learning Continually Under Changing Data Distributions
28 Floris den Hengst (VUA), Learning to Behave: Reinforcement Learning in Human Contexts

2024 01 Daphne Miedema (TU/e), On SQL Learning: Disentangling concepts in data systems education
02 Emile van Krieken (VUA), Optimisation in Neurosymbolic Learning Systems
03 Feri Wijayanto (RUN), Automated Model Selection for Rasch and Mediation Analysis
04 Mike Huisman (Leiden University), Understanding Deep Meta-Learning