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.

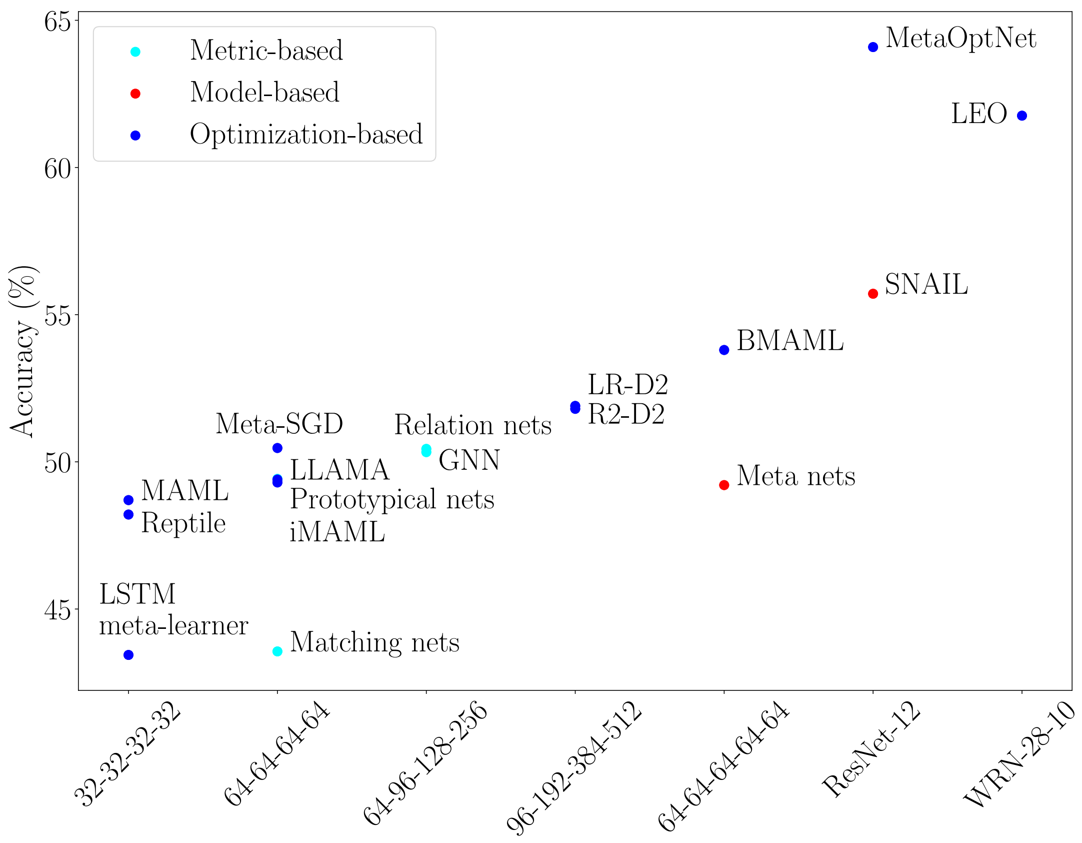


Figure 2.1: The accuracy scores of the covered techniques on 1-shot miniImageNet classification. The used feature extraction backbone is displayed on the x-axis. As one can see, there is a strong relationship between the network complexity and the classification performance.

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 $f_{\boldsymbol{\theta}} : X \rightarrow Y$ 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* 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.

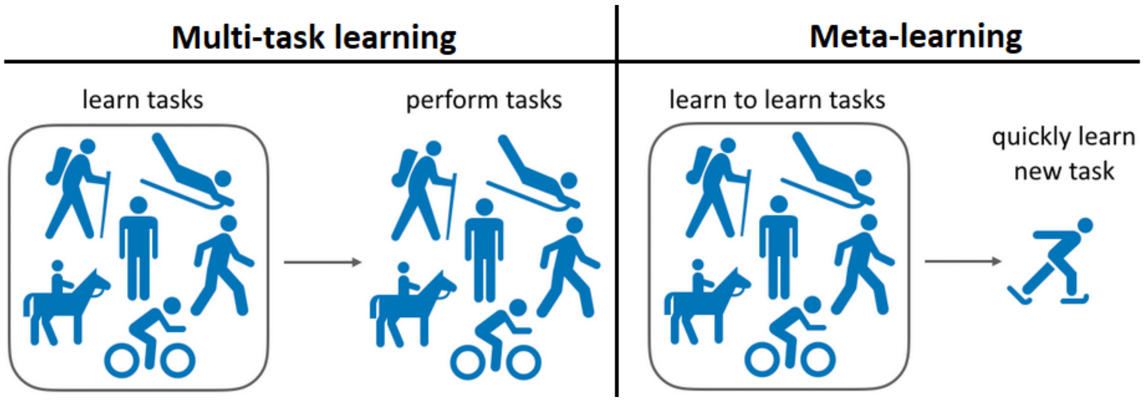


Figure 2.2: The difference between multi-task learning and meta-learning[3].

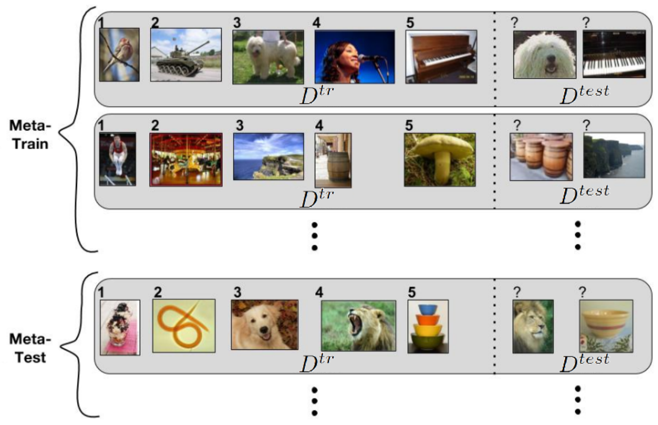


Figure 2.3: Illustration of N-way, k-shot classification, where $N = 5$, and $k = 1$. Meta-validation tasks are not displayed. Adapted from Ravi and Larochelle (2017).

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.

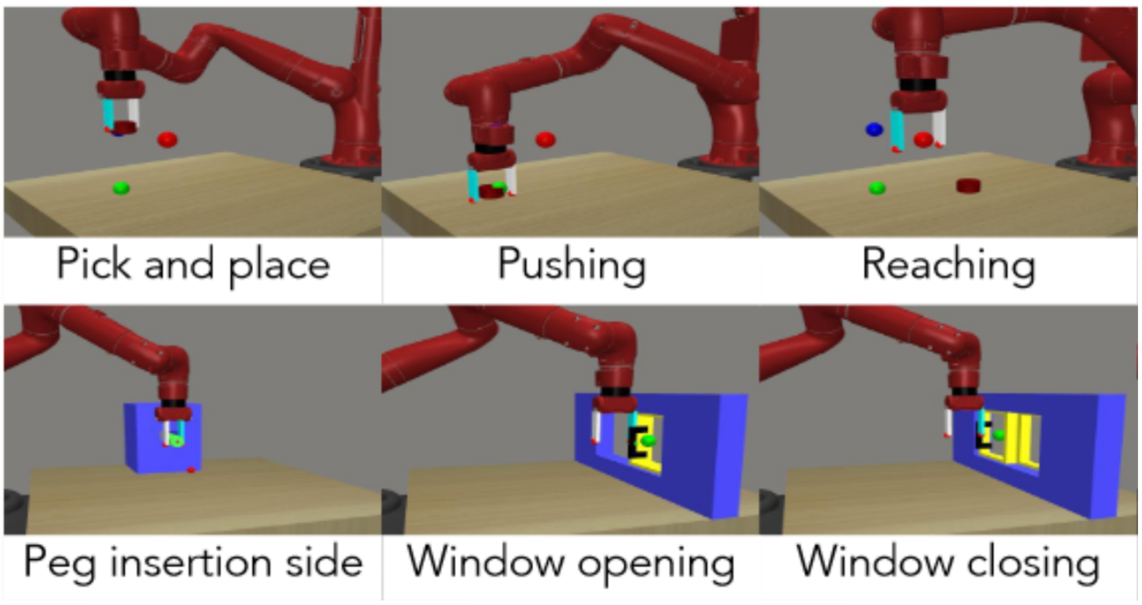


Figure 2.4: Learning continuous robotic control tasks is an important application of deep meta-learning techniques. Image taken from (Yu et al., 2019).

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
$p_{\boldsymbol{\theta}}(Y \vert \boldsymbol{x}, D^{tr}_{\mathcal{T}_j})$	$\sum_{(\boldsymbol{x}_i, y_i) \in D^{tr}_{\mathcal{T}_j}} k_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}_i) y_i$	$f_{\boldsymbol{\theta}}(\boldsymbol{x}, D^{tr}_{\mathcal{T}_j})$	$f_{g_{\boldsymbol{\varphi}(\boldsymbol{\theta}, D^{tr}_{\mathcal{T}_j}, \mathcal{L}_{D^{tr}_{\mathcal{T}_j}})}}(\boldsymbol{x})$

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.

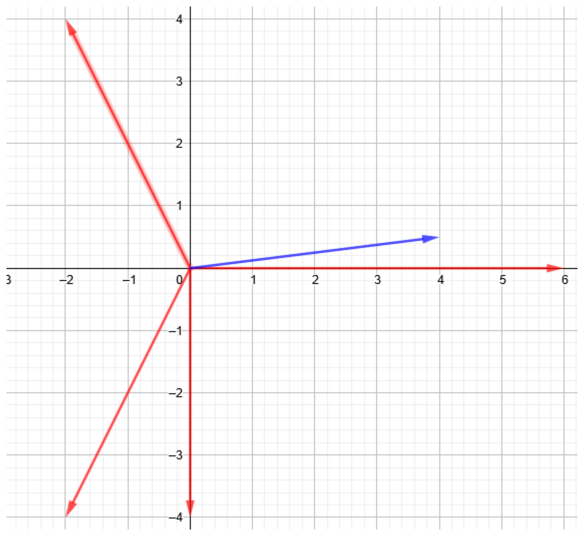


Figure 2.5: Illustration of our metric-based example. The blue vector represents the new input from the query set, whereas the red vectors are inputs from the support set which can be used to guide our prediction for the new input.

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 $p_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j}) = -0.12y_1 - 0.58y_2 - 0.37y_3 + 0.99y_4 = -0.12[1,0,0,0] - 0.58[0,1,0,0] - 0.37[0,0,1,0] + 0.99[0,0,0,1] = [-0.12, -0.58, -0.37, 0.99]$. 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

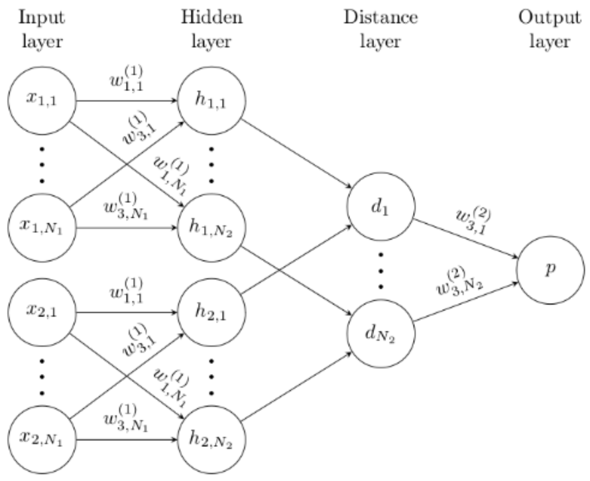


Figure 2.6: Example of a Siamese neural network. Source: Koch et al. (2015).

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 $D^{tr}_{\mathcal{T}_j} = \{(\boldsymbol{x}_i, y_i)\}^m_{i=1}$ (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: $\hat{y} = \sum^m_{i=1} a(\boldsymbol{x}, \boldsymbol{x}_i) y_i$, 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.,

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

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-

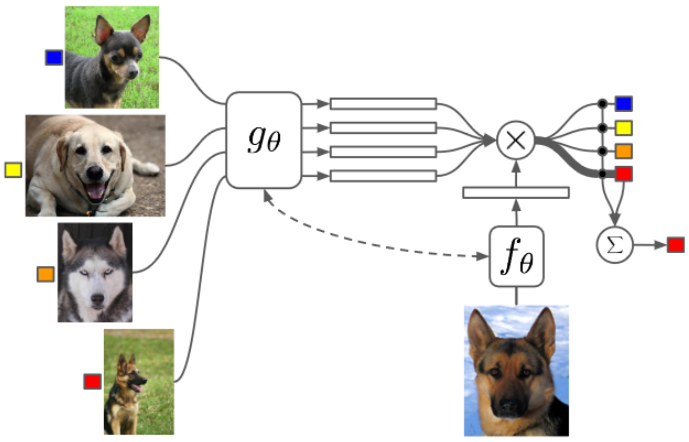


Figure 2.7: Architecture of matching networks. Source: Vinyals et al. (2016).

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

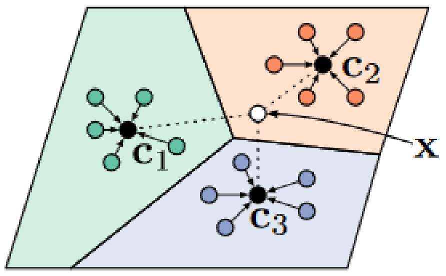


Figure 2.8: Prototypical networks for case of few-shot learning. The $\boldsymbol{c}_k$ are class prototypes for class k which are computed by averaging the representations of inputs (colored circles) in the support set. Note that the representation space is partitioned into three disjoint areas, where 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 $d : X \times X \to [0, +\infty)$ (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,

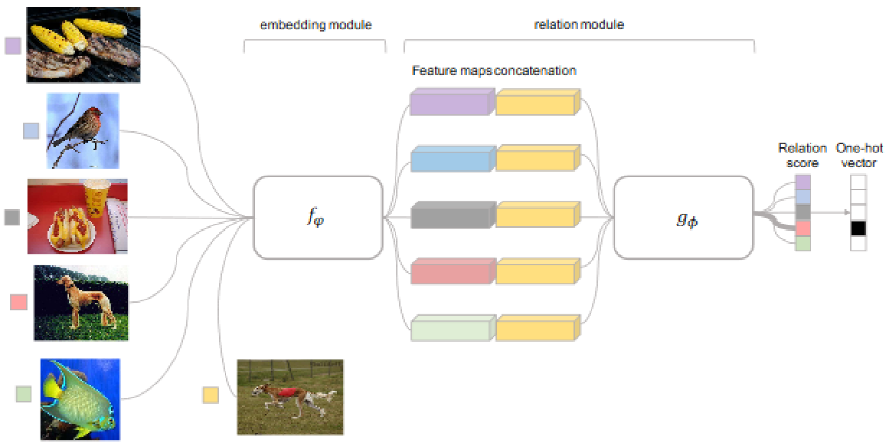


Figure 2.9: Relation network architecture. First, the embedding network f_φ embeds all inputs from the support set $D^{tr}_{\mathcal{T}_j}$ (the five example inputs on the left), and the query input (below the f_φ block). All support set embeddings $f_\varphi(\boldsymbol{x}_i)$ are then concatenated to the query embedding $f_\varphi(\boldsymbol{x})$. These concatenated embeddings are passed into a relation network g_ϕ, 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 $g_\phi([f_\varphi(\boldsymbol{x}), f_\varphi(\boldsymbol{x}_i)])$ 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_φ which is responsible for embedding inputs, and the *relation* network g_ϕ 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 $[f_\varphi(\boldsymbol{x}), f_\varphi(\boldsymbol{x}_i)]$, 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_ϕ. Finally, g_ϕ computes the relation score between $\boldsymbol{x}$ and $\boldsymbol{x}_i$ as

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

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: $y = [\frac{1}{N}, \dots, \frac{1}{N}]$. 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): $\boldsymbol{x}_i, \boldsymbol{x}, \dots, \boldsymbol{x}_i, \boldsymbol{x}$. 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

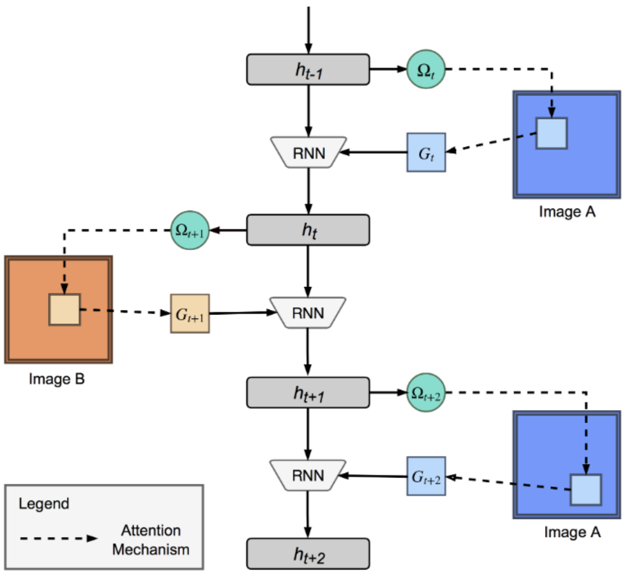


Figure 2.10: Processing in an attentive recurrent comparator. At every time step, the model takes a glimpse of a part of an image and incorporates this information into the hidden state h_t. The final hidden state after taking various glimpses of a pair of images is then used to compute a class similarity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

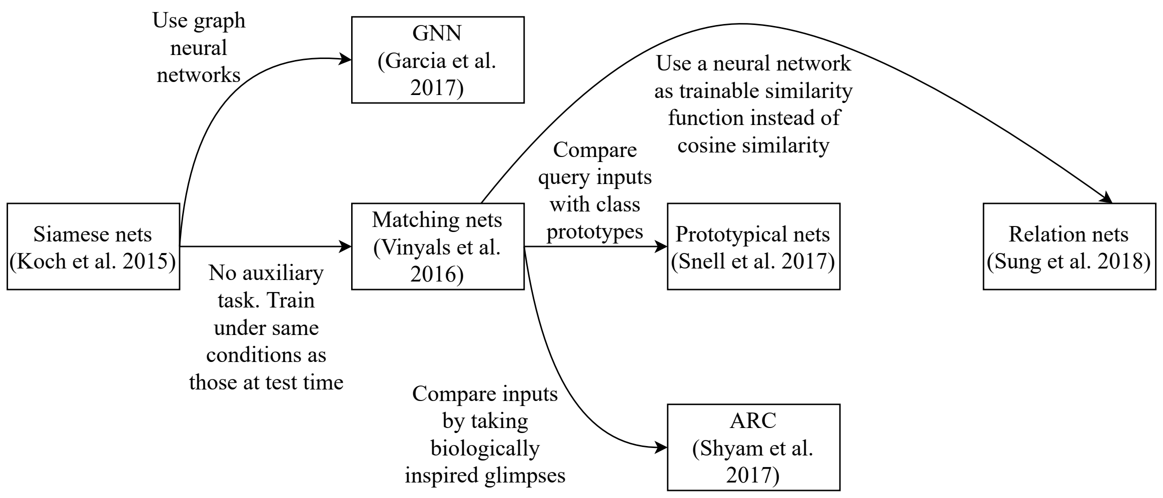

Siamese nets (Koch et al. 2015)
Use graph neural networks
GNN (Garcia et al. 2017)
No auxiliary task. Train under same conditions as those at test time
Matching nets (Vinyals et al. 2016)
Compare query inputs with class prototypes
Prototypical nets (Snell et al. 2017)
Use a neural network as trainable similarity function instead of cosine similarity
Relation nets (Sung et al. 2018)
Compare inputs by taking biologically inspired glimpses
ARC (Shyam et al. 2017)

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 $D^{tr}_{\mathcal{T}_j}$ corresponding to task $\mathcal{T}_j$, model-based techniques compute a class probability distribution for a new input $\boldsymbol{x}$ as

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

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 $D^{tr}_{\mathcal{T}_j} = \{([0, -4], 1), ([-2, -4], 2), ([-2, 4], 3), ([6, 0], 4)\}$, where a tuple denotes a pair $(\boldsymbol{x}_i, y_i)$. Furthermore, suppose our query set only contains one example $D^{test}_{\mathcal{T}_j} = \{([4, 0.5], 4)\}$. 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 $D^{tr}_{\mathcal{T}_j}$ and $D^{test}_{\mathcal{T}_j}$. 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 $D^{tr}_{\mathcal{T}_j}$ 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 $D^{tr}_{\mathcal{T}_j}$ 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

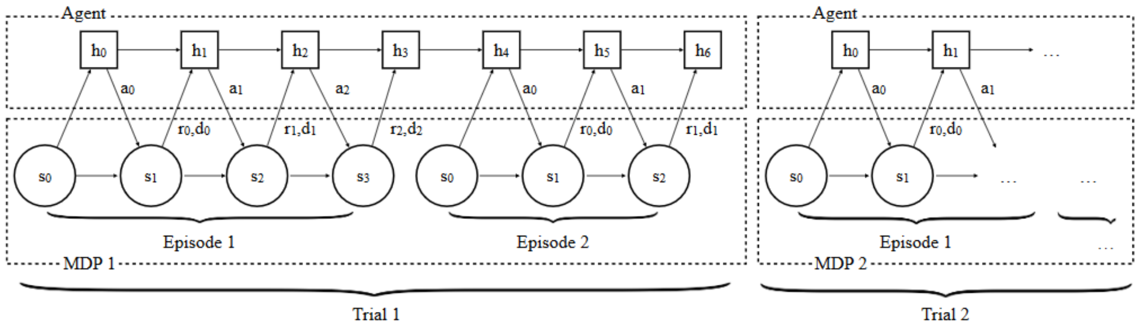


Figure 2.12: Workflow of recurrent meta-learners in reinforcement learning contexts. As mentioned in Section 2.2.1, s_t, r_t, and d_t denote the state, reward, and termination flag at time step t. h_t refers to the hidden state at time t. Source: Duan et al. (2016).

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).

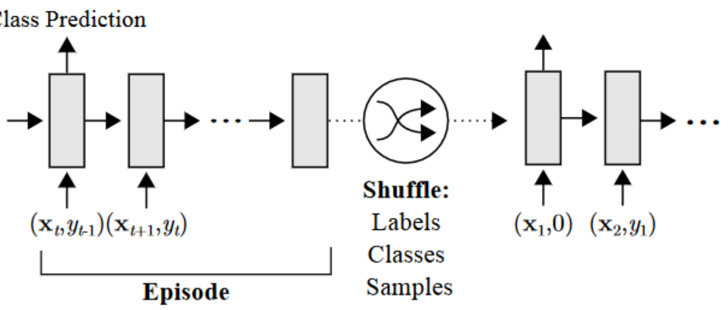

Class Prediction
$(\mathbf{x}_t, y_{t-1})(\mathbf{x}_{t+1}, y_t)$
Episode
Shuffle:
Labels
Classes
Samples
$(\mathbf{x}_1, 0)$ $(\mathbf{x}_2, y_1)$

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).

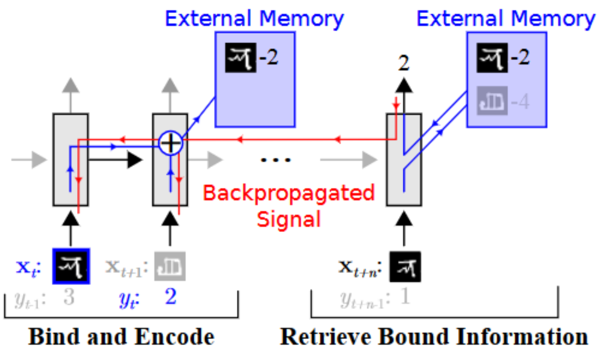


Figure 2.14: Controller-memory interaction in memory-augmented neural networks. Source: Santoro et al. (2016).

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.

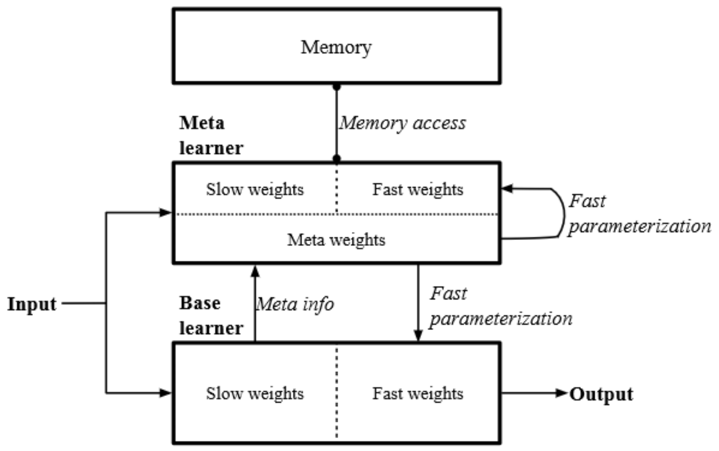


Figure 2.15: Architecture of a Meta Network. Source: Munkhdalai and Yu (2017).

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)

```
 1: Sample $S = \{(\boldsymbol{x}_i, y_i) \backsim D^{tr}_{\mathcal{T}_j}\}^T_{i=1}$ from the support set
 2: for $(\boldsymbol{x}_i, y_i) \in S$ do
 3:     $\mathcal{L}_i = \text{error}(u_{\boldsymbol{\phi}}(\boldsymbol{x}_i), y_i)$
 4: end for
 5: $\boldsymbol{\phi}^* = d_{\boldsymbol{\psi}}(\{\nabla_{\boldsymbol{\phi}}\mathcal{L}_i\}^T_{i=1})$
 6: for $(\boldsymbol{x}_i, y_i) \in D^{tr}_{\mathcal{T}_j}$ do
 7:     $\mathcal{L}_i = \text{error}(b_{\boldsymbol{\theta}}(\boldsymbol{x}_i), y_i)$
 8:     $\boldsymbol{\theta}^*_i = m_{\boldsymbol{\varphi}}(\nabla_{\boldsymbol{\theta}}\mathcal{L}_i)$
 9:     Store $\boldsymbol{\theta}^*_i$ in $i$-th position of example-level weight memory $M$
10:     $\boldsymbol{r}_i = u_{\boldsymbol{\phi},\boldsymbol{\phi}^*}(\boldsymbol{x}_i)$
11:     Store $\boldsymbol{r}_i$ in $i$-th position of representation memory $R$
12: end for
13: $\mathcal{L}_{task} = 0$
14: for $(\boldsymbol{x}, y) \in D^{test}_{\mathcal{T}_j}$ do
15:     $\boldsymbol{r} = u_{\boldsymbol{\phi},\boldsymbol{\phi}^*}(\boldsymbol{x})$
16:     $\boldsymbol{a} = \text{attention}(R, \boldsymbol{r})$          ▷ $a_k$ is the cosine similarity between $\boldsymbol{r}$ and $R(k)$
17:     $\boldsymbol{\theta}^* = \text{softmax}(\boldsymbol{a})^T M$
18:     $\mathcal{L}_{task} = \mathcal{L}_{task} + \text{error}(b_{\boldsymbol{\theta},\boldsymbol{\theta}^*}(\boldsymbol{x}), y)$
19: end for
20: 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.

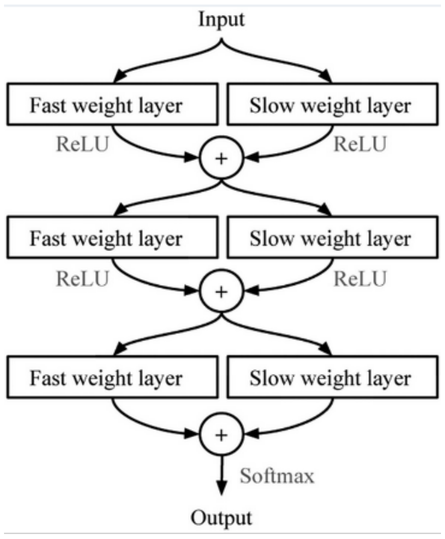


Figure 2.16: Layer augmentation setup used to combine slow and fast weights. Source: Munkhdalai and Yu (2017).

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 $\ell \backsim U(0, \dots, k \cdot N - 1)$ 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

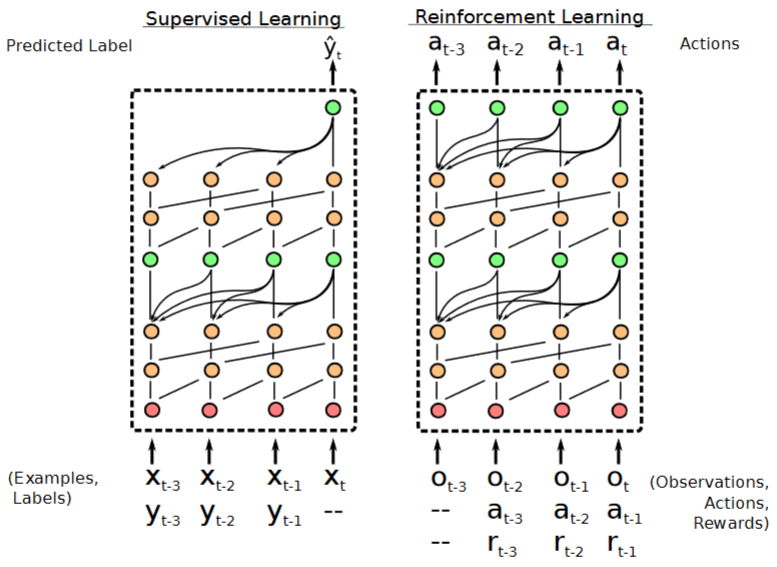

Supervised Learning
Reinforcement Learning
Predicted Label
$\hat{y}_t$
a_{t-3} a_{t-2} a_{t-1} a_t
Actions
(Examples, Labels)
x_{t-3} x_{t-2} x_{t-1} x_t
y_{t-3} y_{t-2} y_{t-1} --
o_{t-3} o_{t-2} o_{t-1} o_t
-- a_{t-3} a_{t-2} a_{t-1}
-- r_{t-3} r_{t-2} r_{t-1}
(Observations, Actions, Rewards)

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 $D^{tr}_{\mathcal{T}_j}$

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

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

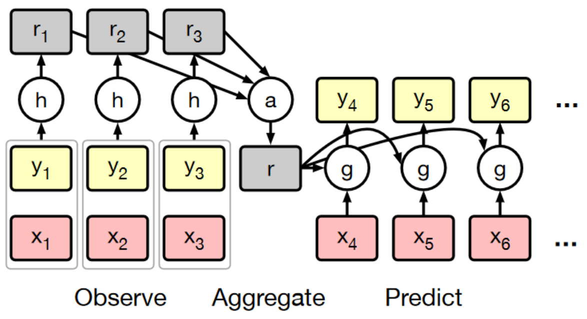


Figure 2.18: Schematic view of how conditional neural processes work. Here, h denotes a network outputting a representation for a observation, a denotes an aggregation function for these representations, and g denotes a neural network that makes predictions for unlabelled observations, based on the aggregated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

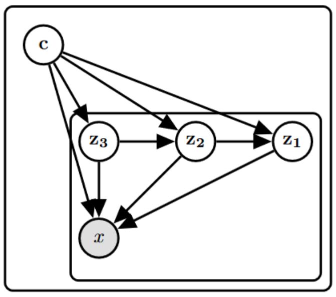


Figure 2.19: Neural statistician architecture. Edges are neural networks. All incoming inputs to a node are concatenated.

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

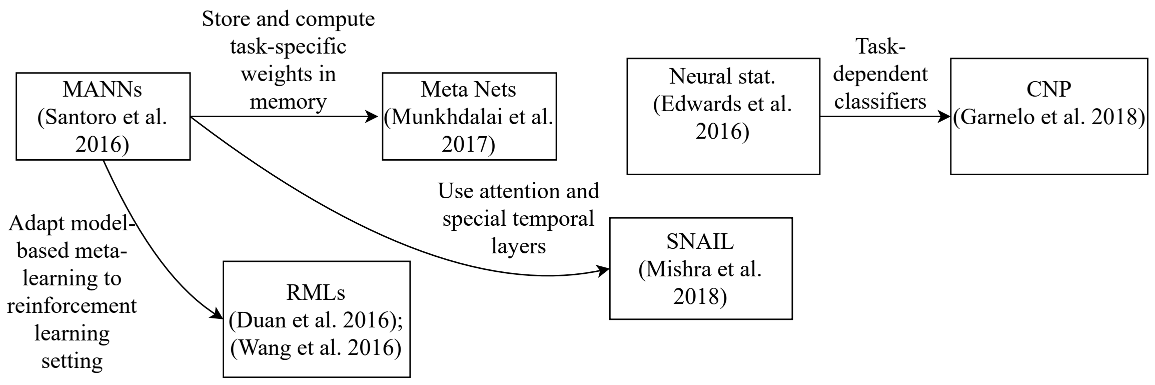

MANNs (Santoro et al. 2016)
Store and compute task-specific weights in memory
Meta Nets (Munkhdalai et al. 2017)
Neural stat. (Edwards et al. 2016)
Task-dependent classifiers
CNP (Garnelo et al. 2018)
Adapt model-based meta-learning to reinforcement learning setting
RMLs (Duan et al. 2016); (Wang et al. 2016)
Use attention and special temporal layers
SNAIL (Mishra et al. 2018)

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

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

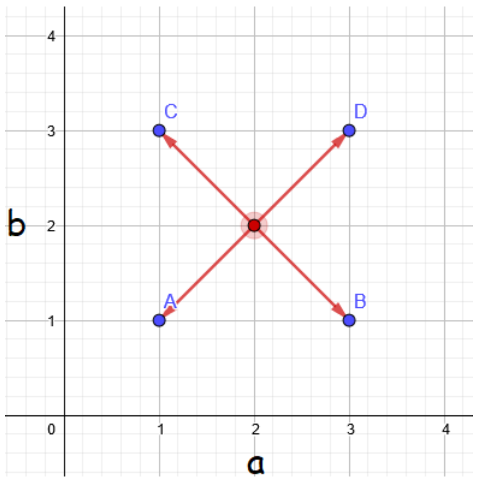


Figure 2.21: Example of an optimization-based technique, inspired by Finn et al. (2017).

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:

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

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

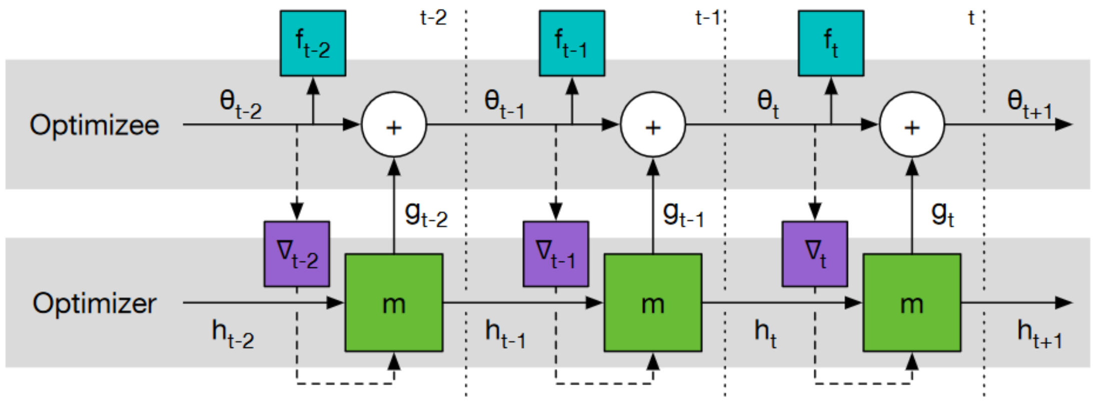


Figure 2.22: Workflow of the LSTM optimizer. Gradients can only propagate backwards through solid edges. f_t denotes the observed loss at time step t. Source: Andrychowicz et al. (2016).

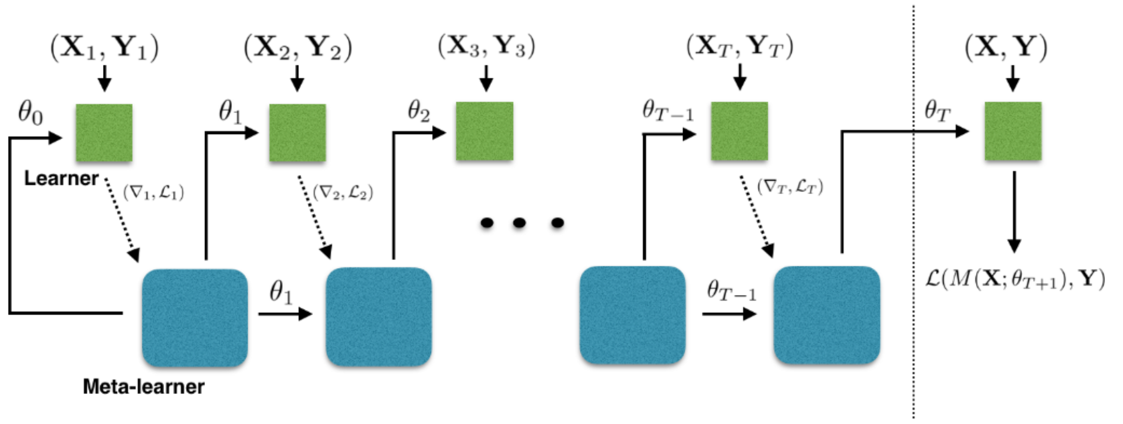
rise to LSTM meta-learners. As such, the base-learner parameters $\boldsymbol{\theta}$ are literally inside the LSTM memory component (cell state). In this way, cell state updates correspond to base-learner parameter updates. This idea was inspired by the resemblance between the gradient and cell state 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,



$(\mathbf{X}_1, \mathbf{Y}_1)$
$(\mathbf{X}_2, \mathbf{Y}_2)$
$(\mathbf{X}_3, \mathbf{Y}_3)$
$(\mathbf{X}_T, \mathbf{Y}_T)$
$(\mathbf{X}, \mathbf{Y})$
θ_0
θ_1
θ_2
θ_{T-1}
θ_T
Learner
$(\nabla_1, \mathcal{L}_1)$
$(\nabla_2, \mathcal{L}_2)$
$(\nabla_T, \mathcal{L}_T)$
θ_1
θ_{T-1}
$\mathcal{L}(M(\mathbf{X}; \theta_{T+1}), \mathbf{Y})$
Meta-learner

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

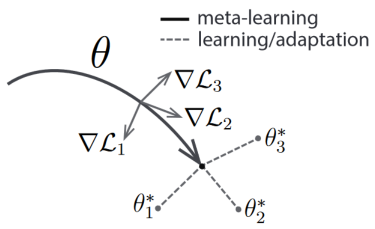


Figure 2.24: MAML learns an initialization point from which it can perform well on various tasks. Source: Finn et al. (2017).

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 $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j})$, gradient descent would produce updated parameters (fast weights)

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

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

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

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

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

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)

```
1: Randomly initialize $\boldsymbol{\theta}$
2: **while** not done **do**
3:     Sample batch of $J$ tasks $B = \mathcal{T}_1, \dots, \mathcal{T}_J \backsim p(\mathcal{T})$
4:     **for** $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j}) \in B$ **do**
5:         Compute $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})$
6:         Compute $\boldsymbol{\theta}'_j = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})$
7:     **end for**
8:     Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \beta \nabla_{\boldsymbol{\theta}} \sum_{\mathcal{T}_j \in B} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)$
9: **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

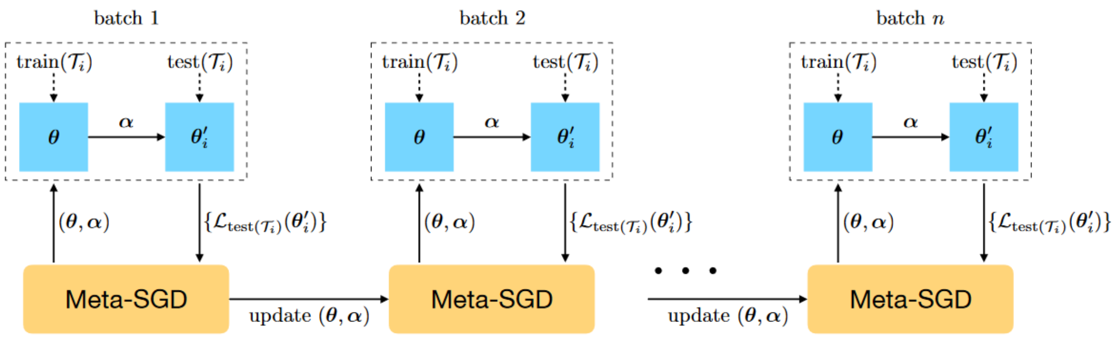


Figure 2.25: Meta-SGD learning process. Source: Li et al. (2017).

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 $i = 1, 2, \dots$ 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.

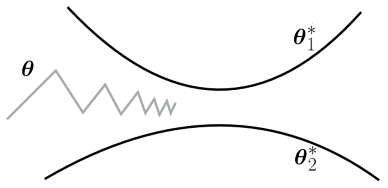


Figure 2.26: Schematic visualization of Reptile's learning trajectory. Here, $\boldsymbol{\theta}_1^*$ and $\boldsymbol{\theta}_2^*$ are the optimal weights for tasks $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. The initialization parameters $\boldsymbol{\theta}$ oscillate between these. Adapted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

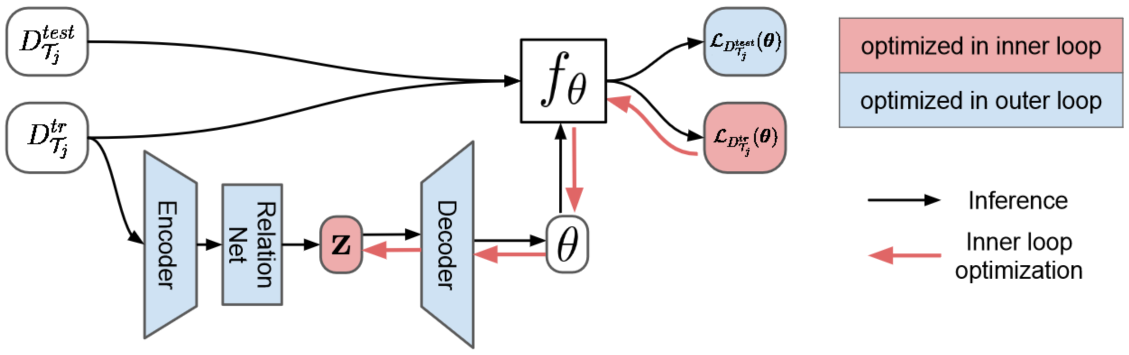


Figure 2.27: Workflow of Latent Embedding Optimization (LEO). adapted from Rusu et al. (2019).

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 $D^{tr}_{\mathcal{T}_j}$ 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 $\gamma$
 1: Initialize empty task buffer $B$
 2: for t = 1,... do
 3:     Initialize data set $D_t = \emptyset$
 4:     Append $\mathcal{T}_t$ to B
 5:     while $|D_t| < N$ do
 6:         Append batch of data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$ to $D_t$
 7:         $\boldsymbol{\theta}_t = MetaUpdate(\boldsymbol{\theta}_t, B, t)$
 8:         Compute $\boldsymbol{\theta}'_t$
 9:         if $\mathcal{L}_{D^{test}_{\mathcal{T}_t}}(\boldsymbol{\theta}'_t) < \gamma$ then
10:             Save $|D_t|$ as the efficiency for task $\mathcal{T}_t$
11:         end if
12:     end while
13:     Save final performance $\mathcal{L}_{D^{test}_{\mathcal{T}_t}}(\boldsymbol{\theta}'_t)$
14:     $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t$
15: 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 $\boldsymbol{\Theta} = \{\boldsymbol{\mu_\theta}, \boldsymbol{\sigma^2_\theta}, \boldsymbol{v}_q, \gamma_p, \gamma_q\}$
 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 $\boldsymbol{\theta} \backsim q = N(\boldsymbol{\mu_\theta} - \gamma_q \nabla_{\boldsymbol{\mu_\theta}} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\mu_\theta}), \boldsymbol{v}_q)$
 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:     $p(\boldsymbol{\theta}|D^{tr}_{\mathcal{T}_j}) = N(\boldsymbol{\mu_\theta} - \gamma_p \nabla_{\boldsymbol{\mu_\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\mu_\theta}), \boldsymbol{\sigma^2_\theta})$
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. $\boldsymbol{\sigma^2_q}$, and $\boldsymbol{v}_q$ represent the covariances of train and test distributions respectively. γ_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

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

The full pseudocode of BMAML is shown in Algorithm 8. Here, $\mathbf{\Theta}_{\mathcal{T}_j}^{n}(\mathbf{\Theta}_0)$ 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)

```
1: Initialize $\mathbf{\Theta}_0$
2: **for** t=1,... until convergence **do**
3:     Sample a batch of tasks B from $p(\mathcal{T})$
4:     **for** task $\mathcal{T}_j \in B$ **do**
5:         Compute chaser $\mathbf{\Theta}_{\mathcal{T}_j}^{n}(\mathbf{\Theta}_0) = SVGD_n(\mathbf{\Theta}_0; D_{\mathcal{T}_j}^{tr}, \alpha)$
6:         Compute leader $\mathbf{\Theta}_{\mathcal{T}_j}^{n+s}(\mathbf{\Theta}_0) = SVGD_s(\mathbf{\Theta}_{\mathcal{T}_j}^{n}(\mathbf{\Theta}_0); D_{\mathcal{T}_j}^{tr} \cup D_{\mathcal{T}_j}^{test}, \alpha)$
7:     **end for**
8:     $\mathbf{\Theta}_0 = \mathbf{\Theta}_0 - \beta \nabla_{\mathbf{\Theta}_0} \sum_{\mathcal{T}_j \in B} d(\mathbf{\Theta}_{\mathcal{T}_j}^{n}(\mathbf{\Theta}_0), SG(\mathbf{\Theta}_{\mathcal{T}_j}^{n+s}(\mathbf{\Theta}_0)))$
9: **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 $g_{\boldsymbol{\phi}} : X \to \mathbb{R}^e$ 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

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

where $XX^T \in \mathbb{R}^{n \times n}$ 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

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

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.

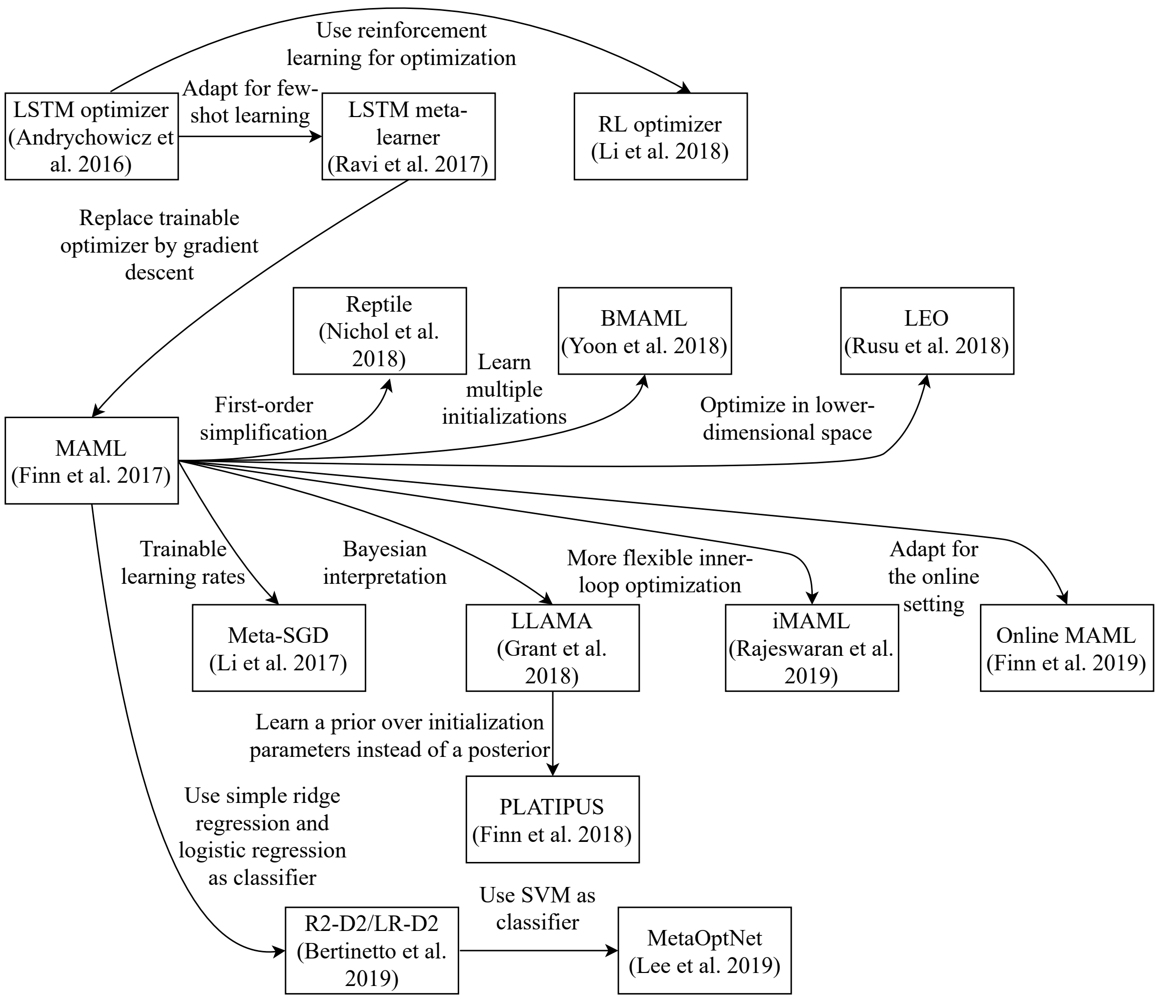

Use reinforcement learning for optimization
LSTM optimizer (Andrychowicz et al. 2016)
Adapt for few-shot learning
LSTM meta-learner (Ravi et al. 2017)
RL optimizer (Li et al. 2018)
Replace trainable optimizer by gradient descent
Reptile (Nichol et al. 2018)
BMAML (Yoon et al. 2018)
LEO (Rusu et al. 2018)
Learn multiple initializations
First-order simplification
Optimize in lower-dimensional space
MAML (Finn et al. 2017)
Trainable learning rates
Bayesian interpretation
More flexible inner-loop optimization
Adapt for the online setting
Meta-SGD (Li et al. 2017)
LLAMA (Grant et al. 2018)
iMAML (Rajeswaran et al. 2019)
Online MAML (Finn et al. 2019)
Learn a prior over initialization parameters instead of a posterior
Use simple ridge regression and logistic regression as classifier
PLATIPUS (Finn et al. 2018)
Use SVM as classifier
R2-D2/LR-D2 (Bertinetto et al. 2019)
MetaOptNet (Lee et al. 2019)

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 ± 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.