13

Metalearning for Deep Neural Networks

Mike Huisman, Jan N. van Rijn, and Aske Plaat

Summary. Deep neural networks have enabled large breakthroughs in various domains ranging from image and speech recognition to automated medical diagnosis. However, these networks are notorious for requiring large amounts of data to learn from, limiting their applicability in domains where data is scarce. Through metalearning, the networks can learn how to learn, allowing them to learn from fewer data. In this chapter, we provide a detailed overview of metalearning for knowledge transfer in deep neural networks. We categorize the techniques into (i) metric-based, (ii) model-based, and (iii) optimization-based techniques, cover the key techniques per category, discuss open challenges, and provide directions for future research such as performance evaluation on heterogeneous benchmarks.

13.1 Introduction

Although current deep learning methods obtain great successes, training is generally time consuming and large datasets are required to achieve good performance. Metalearning offers a solution to these issues. That is, metalearning deep neural networks can improve their learning ability over time, or equivalently, “learn to learn”: this allows them to learn new concepts from less data.

Most metalearning techniques, in the context of deep learning, learn at two levels. At the *inner level*, the agent is presented with a new task (dataset) and tries to quickly learn the associated concepts. This quick adaptation is facilitated by the knowledge that the agent has accumulated from other tasks, at the *outer level*. Thus, the inner level involves a single task, whereas the outer level involves a multitude of tasks. The accumulated knowledge is often directly embedded into the parameters of the agent (neural network). Note that this is in contrast to some other methods in this book (see, e.g., Chapter 6), where metalearning is used to optimize the hyperparameters of algorithms.

In this chapter, we provide a detailed overview of metalearning for knowledge transfer in deep neural networks, which was briefly introduced in the previous chapter. Following Vinyals (2017), we categorize this field into three groups: (i) metric-, (ii) model-, and (iii) optimization-based techniques. After introducing our notation and providing background information, we summarize key techniques of each category, identify the main challenges, and formulate open questions.

P. Brazdil et al., *Metalearning*, Cognitive Technologies,
https://doi.org/10.1007/978-3-030-67024-5_13

13.2 Background and Notation

In this section, we introduce and contrast base-level learning and metalearning in the context of deep learning. Additionally, we briefly discuss common training and evaluation procedures.

13.2.1 The meta-abstraction for deep neural networks

In *supervised learning*, we wish to learn a function $f_{\boldsymbol{\theta}} : X \rightarrow Y$ that maps inputs $\boldsymbol{x}_i \in X$ to their corresponding outputs $y_i \in Y$. In the context of deep learning, f is a neural network with parameters $\boldsymbol{\theta}$. Note that this is in contrast to previous chapters, where $\boldsymbol{\theta}$ was used to denote hyperparameters of algorithms. Given a dataset $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{m}$ of m examples, learning boils down to finding parameters $\boldsymbol{\theta}$ that minimize an empirical loss function $\mathcal{L}_D$. This loss function captures how well the model is performing by computing the difference between the model predictions $\hat{y}_i$ and correct dataset outputs y_i. In short, we wish to find the optimal parameters

$$\boldsymbol{\theta}^* := \arg\min_{\boldsymbol{\theta}} \mathcal{L}_D(\boldsymbol{\theta}). \tag{13.1}$$

Finding theoretically optimal parameters $\boldsymbol{\theta}^*$ is often infeasible. We can, however, approximate them, guided by *pre-defined* meta-knowledge ω (including, e.g., initial model parameters $\boldsymbol{\theta}$, choice of optimizer, and learning rate schedule) (Hospedales et al., 2020). As such, we approximate

$$\boldsymbol{\theta}^* \approx g_\omega(D, \mathcal{L}_D), \tag{13.2}$$

where g_ω is an optimization procedure that takes meta-knowledge ω, a dataset D, and loss function $\mathcal{L}_D$ to produce updated weights $g_\omega(D, \mathcal{L}_D)$ that presumably perform well on D. In practice, the optimizer g_ω often uses gradients of the loss function to update the model parameters $\boldsymbol{\theta}$. To measure the generalization performance of the found parameters $\boldsymbol{\theta}^*$, we can split the dataset into training and test sets D^{tr} and D^{test}, and use, e.g., cross-validation techniques (see Chapter 3).

In contrast, *supervised metalearning* for deep neural networks does not assume that some meta-knowledge ω is given, or pre-defined. Instead, the goal is to find the best ω such that new *tasks* $\mathcal{T}_j$ (datasets) can be learned as quickly as possible. Metalearning techniques often learn ω from a multitude of tasks $\mathcal{T}_j$. Note that this in contrast to regular supervised learning, where only one task (dataset) is used.

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

$$\omega^* := \arg\min_{\omega} \underbrace{\mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}}_{\text{Outer level}} \underbrace{[\mathcal{L}_{\mathcal{T}_j}(g_\omega(\mathcal{T}_j, \mathcal{L}_{\mathcal{T}_j}))]}_{\text{Inner level}}. \tag{13.3}$$

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

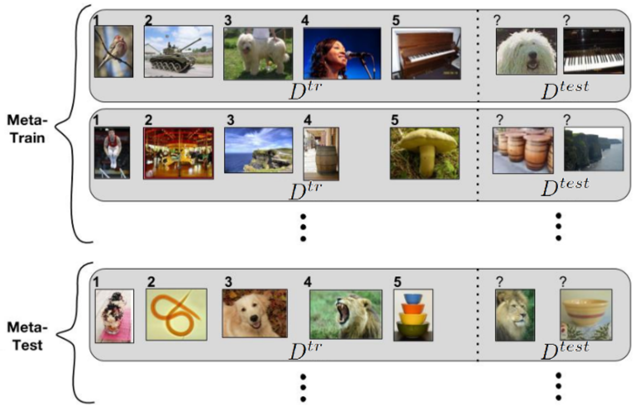

Fig. 13.1: Illustration of N-way k-shot classification, where $N = 5$, and $k = 1$. Meta-validation tasks are not displayed. Adapted from Ravi and Larochelle (2017)

13.2.2 Common training and evaluation procedures

In the previous subsection, we described the learning objectives for supervised learning and metalearning for deep neural networks. However, we remained agnostic with respect to the setup used in practice to achieve these objectives. In general, one optimizes a meta-objective by using various tasks. In practice, this is done in three stages: the (i) *meta-train*, (ii) *meta-validation*, and (iii) *meta-test* stages, each of which is associated with a set of tasks from a homogeneous source. Importantly, note that this is in contrast to some of the previous chapters, where heterogeneous data sources were used for knowledge transfer.

Now, in the meta-train stage, the metalearning algorithm is deployed on the meta-train tasks. 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., hyperparameters of the metalearning algorithm. Lastly, the meta-test tasks are used to give a final performance estimate of the metalearning technique.

N-way k-shot learning

A frequently used instantiation of this general meta-setup is called N-way k-shot classification (see Figure 13.1). This setup is also divided into the three stages — meta-train, meta-validation, and meta-test — which are used for metalearning, metalearner 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 dataset D, such that every $y_i \in L^s$. Note that this requires access to a dataset D. Now, the sampling process is guided by the N-way k-shot principle, which states that every train dataset $D^{tr}_{\mathcal{T}_j}$ 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 training 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* $D^{test}_{\mathcal{T}_j}$. Throughout this chapter, we will use the terms training/support and test/query sets interchangeably. Importantly, note that with this terminology, the test set (or query 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 metalearning 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, 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 for different episodes/tasks, with different support and query sets, the model will slowly accumulate meta-knowledge ω, which can ultimately speed up learning on new tasks.

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. This way we test how well a technique performs at metalearning.

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*.

13.2.3 Overview of the rest of this chapter

In the remainder of this chapter we will look in more detail at individual metalearning methods. As indicated before, the methods can be grouped into three main categories (Vinyals, 2017), which we will discuss in sequence:

(i) Metric-based
(ii) Model-based
(iii) Optimization-based techniques

To help give an overview of the methods, we draw your attention to the following tables. Table 13.1 summarizes the three categories and provides the key ideas, strengths, and weaknesses of the approaches. The technical details are explained in the remainder of this chapter. Table 13.2 contains an overview of all the techniques that are discussed further on.

13.3 Metric-Based Metalearning

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 with example inputs (of which we know the labels) in the metalearned feature space. The greater 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 metalearning as they leverage their prior learning experience (metalearned feature space) to "learn" new tasks more quickly. Here,

	Metric	**Model**	**Optimization**
Key idea	Input similarity	Internal task representation	Bilevel optimization
Predictions	$\sum_{\boldsymbol{x}_i, y_i \in D^{tr}_{\mathcal{T}_j}} k_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}_i) y_i$	$f_{\boldsymbol{\theta}}(\boldsymbol{x}, D^{tr}_{\mathcal{T}_j})$	$f_{g_{\boldsymbol{\varphi}(\boldsymbol{\theta}, D^{tr}_{\mathcal{T}_j}, \mathcal{L}_{\mathcal{T}_j})}}(\boldsymbol{x})$
Strength	Simple and effective	Flexible	More robust generalizability
Weakness	Limited to supervised learning	Weak generalization	Computationally expensive

Table 13.1: High-level overview of the categorization of metalearning techniques for deep neural networks. See Section 13.6 for more details. Table extended from Vinyals (2017)

Name	Key idea
Metric-based	**Input similarity**
Siamese neural networks	Two-input, shared-weight, class identity network
Matching networks	Learn input embeddings for cosine-similarity weighted predictions
Graph neural networks	Propagate label information to unlabeled inputs in a graph
Attentive recurrent comparators	LSTM-based input fusion through interleaved glimpses
Model-based	**Internal task representations**
Memory-augmented neural networks	External short-term memory module for fast learning
Meta networks	Fast reparameterization of base-learner by distinct metalearner
Simple neural attentive meta-learner	Attention mechanism coupled with temporal convolutions
Conditional neural processes	Condition predictive model on embedded contextual task data
Optimization-based	**Bilevel optimization**
LSTM optimizer	RNN proposing weight updates for base-learner
Reinforcement learning optimizer	View optimization as reinforcement learning problem
Model-agnostic meta-learning	Learn initialization weights $\boldsymbol{\theta}$ for fast adaptation
Reptile	Move initialization towards task-specific updated weights

Table 13.2: Overview of the metalearning techniques discussed in this chapter

"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 metalearned 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 greater similarity. Class predictions for new inputs $\boldsymbol{x}$ can then be made by comparing $\boldsymbol{x}$ with example inputs $\boldsymbol{x}_i$, of which we know the true label y_i, the underlying idea being that the greater the similarity between $\boldsymbol{x}$ and $\boldsymbol{x}_i$, the more likely it becomes that $\boldsymbol{x}$ also has label y_i.

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 the similarity kernel $k_{\boldsymbol{\theta}}$:

$$P_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j}) = \sum_{(\boldsymbol{x}_i, y_i) \in D^{tr}_{\mathcal{T}_j}} k_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}_i) y_i. \tag{13.4}$$

Importantly, the labels y_i are assumed to be *one-hot encoded*, meaning that they are represented by zero vectors with a "1" at 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 ith entry corresponds to the probability that input $\boldsymbol{x}$ has class Y_i (given the support set). The predicted class is thus $\hat{y} = argmax_{i=1,2,\ldots,|Y|} P_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, S)_i$, where $P_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, S)_i$ is the computed probability that input $\boldsymbol{x}$ has class Y_i.

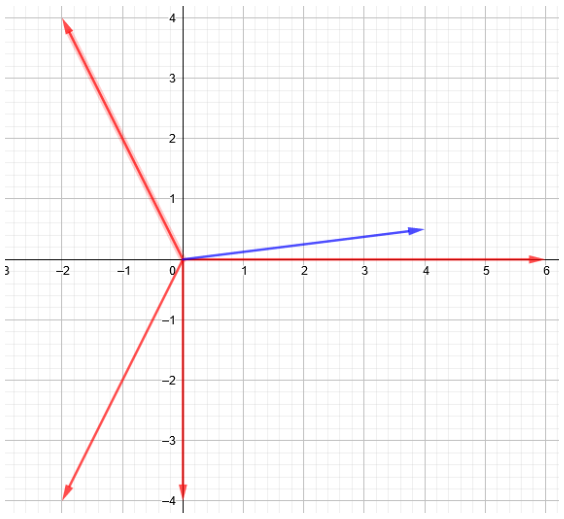

Fig. 13.2: Illustration of our metric-based example

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. Now, our test set only contains one example $D^{test}_{\mathcal{T}_j} = \{([4,0.5],y)\}$. Then, the goal is to predict the correct label for the new input $[4,0.5]$ using only examples in $D^{tr}_{\mathcal{T}_j}$. The problem is visualized in Figure 13.2, where red vectors correspond to example inputs from our training set. The blue vector is the new input that needs to be classified. Intuitively, this new input is most similar to the vector $[6,0]$, which means that we expect the label for the new input to be the same as that for $[6,0]$, i.e., 4.

Now, 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 nth 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 13.3, 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 13.4, 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.

$\boldsymbol{x}_i$	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 13.3: Example showing pairwise input comparisons. Numbers were rounded to two decimals

One may wonder why such techniques are metalearners, for we could take any single dataset D and use pairwise comparisons to compute predictions. Now, at the outer level, metric-based metalearners 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 pairwise comparisons. As such, one learns an embedding function across tasks to facilitate task-specific learning, which is equivalent to “learning to learn”, or metalearning.

In the remainder of this section we will discuss various key metric-based techniques, including

- Siamese networks (Koch et al., 2015)
- Matching networks Vinyals et al. (2016)

- Graph neural networks (Garcia and Bruna, 2017)
- Attentive recurrent comparators (Shyam et al., 2017)

13.3.1 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 ith 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 13.3.

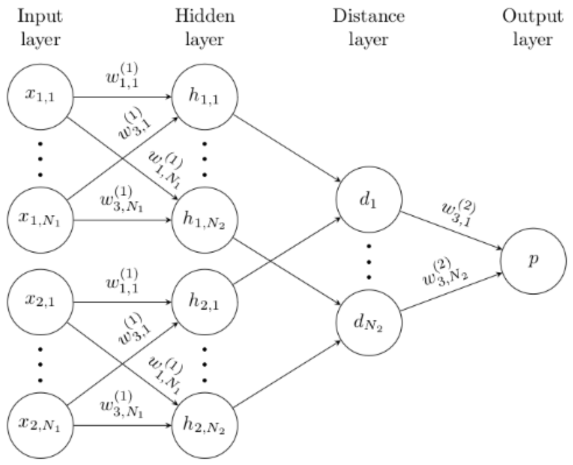

Fig. 13.3: Example of a Siamese neural network. Source: Koch et al. (2015)

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 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 chapter, Siamese neural networks do not optimize directly for good performance across tasks. 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.

13.3.2 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 pairwise comparisons between the given support set $D^{tr}_{\mathcal{T}_j} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m$ (for a task $\mathcal{T}_j$) and new inputs $\boldsymbol{x} \in D^{test}_{\mathcal{T}_j}$ from the query/test 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* the example labels y_i in the support set, based on the similarity of inputs $\boldsymbol{x}_i$ to new input $\boldsymbol{x}$. More specifically, predictions are computed as follows: $\hat{y} = \sum_{i=1}^m a(\boldsymbol{x}, \boldsymbol{x}_i) y_i$, where a is a non-parametric (non-trainable) attention mechanism, or similarity kernel. This classification process is shown in Figure 13.4. In this figure, the input to $f_{\boldsymbol{\theta}}$ has to be classified, using the support set $D^{tr}_{\mathcal{T}_j}$ (input to $g_{\boldsymbol{\theta}}$).

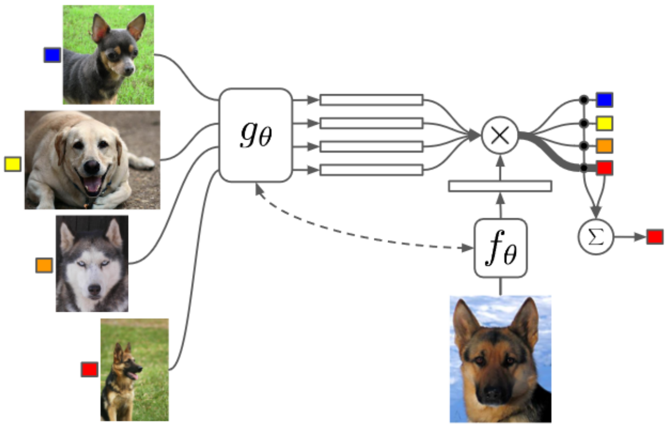

Fig. 13.4: Architecture of matching networks. Source: Vinyals et al. (2016)

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

$$a(\boldsymbol{x}, \boldsymbol{x}_i) = \frac{e^{c(f_{\boldsymbol{\phi}}(\boldsymbol{x}), g_{\boldsymbol{\varphi}}(\boldsymbol{x}_i))}}{\sum_{j=1}^m e^{c(f_{\boldsymbol{\phi}}(\boldsymbol{x}), g_{\boldsymbol{\varphi}}(\boldsymbol{x}_j))}}, \tag{13.5}$$

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-dimensional) latent vector (corresponding to the activation state in the final hidden layer of a neural network). As such, neural networks act as embedding functions. Now, 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 13.4. The second choice is to make f_ϕ and g_φ dependent on the support set $D^{tr}_{\mathcal{T}_j}$ using long short-term memory networks (LSTMs). In that case, f_ϕ is represented by an attention LSTM, and g_φ 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 with 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 pairwise comparisons between inputs. In contrast to Siamese neural networks (Koch et al., 2015), this feature space is learned across tasks instead of on a distinct verification task.

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

Some slight variations on matching networks have given rise to new metric-based techniques. Prototypical networks (Snell et al., 2017) use Euclidean distance as the basis for a similarity function, and reduce the required number of pairwise comparisons by comparing the new input $\boldsymbol{x}$ with class prototypes (mean vectors of inputs $\boldsymbol{x}_i$ from a class) in the support set. Instead of using fixed similarity metrics, relation networks (Sung et al., 2018) learn a similarity metric represented by a neural network, which allows for greater expressive power.

13.3.3 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 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/test 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 for which we do have the labels to nodes for which we have to predict labels. The used local operators are 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 not restricted to supervised learning settings.

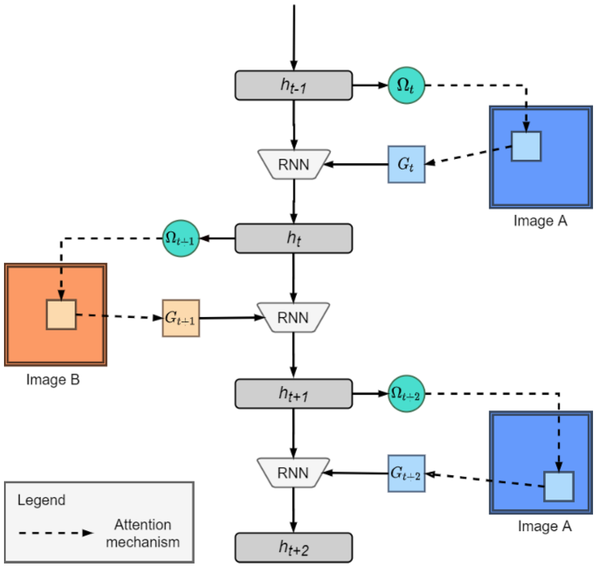

Fig. 13.5: Processing in an attentive recurrent comparator. Source: Shyam et al. (2017)

13.3.4 Attentive recurrent comparators

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

Given two inputs $\boldsymbol{x}_i$ and $\boldsymbol{x}$, we feed them in interleaved fashion repeatedly into a recurrent neural network (controller): $\boldsymbol{x}_i, \boldsymbol{x}, \ldots, \boldsymbol{x}_i, \boldsymbol{x}$. Thus, the image at time step t is given by $I_t = \boldsymbol{x}_i$ if t is even, else $\boldsymbol{x}$. Then, at each time step t, the attention mechanism focuses on a square region of the current image: $G_t = attend(I_t, \Omega_t)$, where $\Omega_t = W_g h_{t-1}$ are attention parameters, which are computed from the previous hidden state h_{t-1}. The next hidden state $h_{t+1} = \text{RNN}(G_t, h_{t-1})$ is given by the glimpse at time t, i.e., G_t, and the previous hidden state h_{t-1}. The entire sequence consists of g glimpses per image. After this sequence is fed into the recurrent neural network (indicated by RNN$(\ldots)$), the final hidden state h_{2g} is used as a combined representation of $\boldsymbol{x}_i$ relative to $\boldsymbol{x}$. This process is summarized in Figure 13.5. Classification decisions can then be made by feeding the combined representations 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).

Metric-based techniques, in conclusion

In this section, we have seen various metric-based techniques. The metric-based techniques metalearn an informative feature space that can be used to compute class predictions based on input similarity scores.

Key advantages of these 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-training time, metric-learning techniques are unable to absorb new task information into the network weights. Consequently, performance may suffer.

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

13.4 Model-Based Metalearning

A different approach to metalearning for deep neural networks 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/training 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 pairwise 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 metalearning good feature spaces, as they can also learn internal dynamics, used to process and predict the 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 follows:

$$P_{\boldsymbol{\theta}}(Y|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j}) = f_{\boldsymbol{\theta}}(\boldsymbol{x}, D^{tr}_{\mathcal{T}_j}), \tag{13.6}$$

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

Example

Using the same example as in Section 13.3, suppose we are given a task training 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 test set only contains one example $D^{test}_{\mathcal{T}_j} = \{([4, 0.5], 4)\}$. This problem has been visualized in Figure 13.2 (in Section 13.3). Now, 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 ith example in the ith row of the memory module. After processing the support set, the memory matrix contains all examples and, as such, serves as an internal task representation.

Now, 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 ith row of M, ignoring the correct label) and predicts the class of the input which yields the largest dot product. This would produce scores of $-2, -10, -6$, 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.

This example was a bit simplistic for illustrative purposes. More advanced and successful techniques have been proposed, including

- Memory-augmented neural networks (Santoro et al., 2016)
- Meta networks (Munkhdalai and Yu, 2017)
- Simple neural attentive meta-learner (SNAIL) (Mishra et al., 2018)
- Conditional neural processes (Garnelo et al., 2018)

We discuss these techniques in order.

13.4.1 Memory-augmented neural networks

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 13.6. 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/training set is fed into the memory-augmented neural network first. Afterwards, the query/test set is processed. 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).

The interaction between the controller and memory is visualized in Figure 13.7. Given an input $\boldsymbol{x}_t$ at time t, the controller generates a key $\boldsymbol{k}_t$, which is used to either retrieve or store a memory from/in memory matrix M, i.e., $M_t(i)$. To read from memory using a key $\boldsymbol{k}_t$, a (column) read vector $\boldsymbol{w}^r_t$ is created with the same number of rows as the memory matrix M, where each entry i denotes the cosine similarity between the key $\boldsymbol{k}_t$, and the memory stored in row i, i.e., $M_t(i)$. Then, the memory $\boldsymbol{r}_t = \sum_i w^r_t(i)M(i)$ is retrieved, which is simply a linear combination of rows/memories in the memory matrix M.

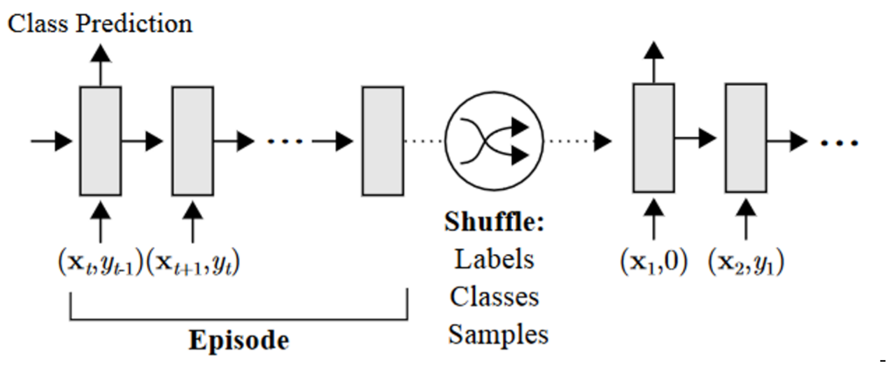
Class Prediction
$(\mathbf{x}_t, y_{t-1})(\mathbf{x}_{t+1}, y_t)$
Episode
Shuffle:
Labels
Classes
Samples
$(\mathbf{x}_1, 0)$ $(\mathbf{x}_2, y_1)$

Fig. 13.6: Workflow of memory-augmented neural networks. Source: Santoro et al. (2016)

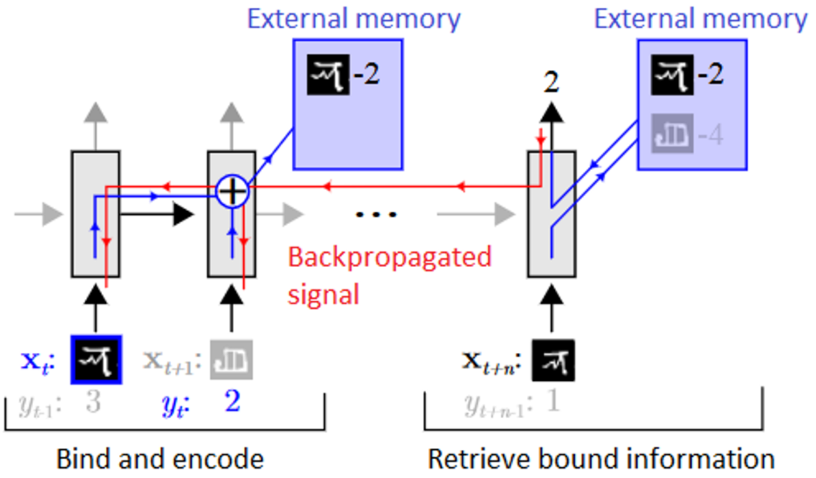

Fig. 13.7: Controller–memory interaction in memory-augmented neural networks. Source: Santoro et al. (2016)

To write to memory, Santoro et al. (2016) propose a new mechanism called least recently used access (LRUA). LRUA writes to either 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 the 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. Now, let n be the total number of reads to memory, and $\ell u(n)$ (ℓu for 'least used') be the nth smallest value in the usage vector $\boldsymbol{w}_t^u$. Then, the least-used weights are defined as follows:

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

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.

Predictions are made as follows: given an input $\boldsymbol{x}_t$, memory-augmented neural networks compute the corresponding memory $\boldsymbol{r}_t$ and feed that memory into a classifier. 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.

In summary, memory-augmented neural networks (Santoro et al., 2016) combine an external memory and a neural network to achieve metalearning. The interaction between a controller, with long-term memory parameters $\boldsymbol{\theta}$, and the memory M may also be interesting for studying human metalearning (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.

13.4.2 Meta networks

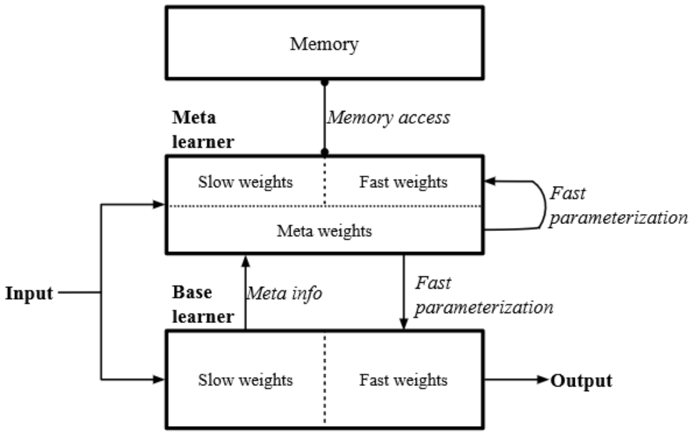

Fig. 13.8: Architecture of a meta network. Source: Munkhdalai and Yu (2017)

In a 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. Meta networks are divided into two distinct subsystems (consisting of neural networks), i.e., the base- and metalearner (whereas in memory-augmented neural networks the base- and meta-components are intertwined). Now, the base-learner is responsible for performing tasks, and for providing the metalearner with meta-information, such as loss gradients. The metalearner can then compute fast task-specific weights for itself and the base-learner, such that it can perform better on the given task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{test}_{\mathcal{T}_j})$. This workflow is depicted in Figure 13.8.

The metalearner consists of neural networks $u_{\boldsymbol{\phi}}, m_{\boldsymbol{\varphi}}$, and $d_{\boldsymbol{\psi}}$. The network $u_{\boldsymbol{\phi}}$ is used as an input representation function. The 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.

```
1: Sample $S = \{(\boldsymbol{x}_i, y_i) \backsim D^{tr}_{\mathcal{T}_j}\}_{i=1}^{T}$ 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: $\boldsymbol{\phi}^* = d_{\boldsymbol{\psi}}(\{\nabla_{\boldsymbol{\phi}}\mathcal{L}_i\}_{i=1}^{T})$
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 $\nabla_{\Theta}\mathcal{L}_{task}$
```

Algorithm 13.1: Meta networks, by Munkhdalai and Yu (2017)

The pseudocode for meta networks is displayed in Algorithm 13.1. First, a sample of the support set is created (line 1), which is then used to compute task-specific weights $\boldsymbol{\phi}^*$ for the representation network u (lines 2–5).

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 $\mathcal{L}_i$ (lines 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 ith row of memory matrix M (line 9). Additionally, input representations $\boldsymbol{r}_i$ are computed and stored in the 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 the memory matrix R. This matching gives us a similarity vector $\boldsymbol{a}$, where every entry k denotes the similarity between the input representation $\boldsymbol{r}$ and the kth row in the 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 the 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 the involved parameters can be updated using backpropagation (line 20).

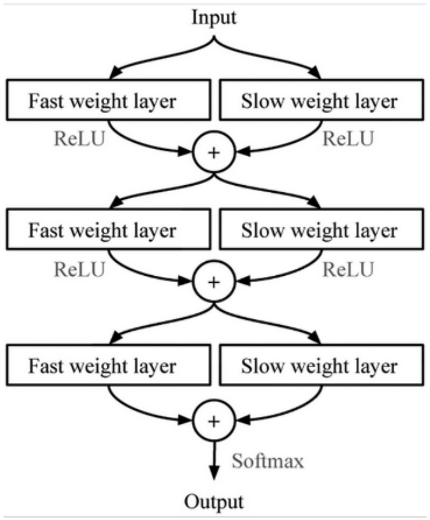

Fig. 13.9: Layer augmentation setup used to combine slow and fast weights. Source: Munkhdalai and Yu (2017)

Note that some neural networks use both slow and fast weights at the same time. Munkhdalai and Yu (2017) use an augmentation setup for this, as depicted in Figure 13.9.

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.

13.4.3 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 the 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-bandwidth” memory access, and the attention mechanism allows us to pinpoint specific experiences. Figure 13.10 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.

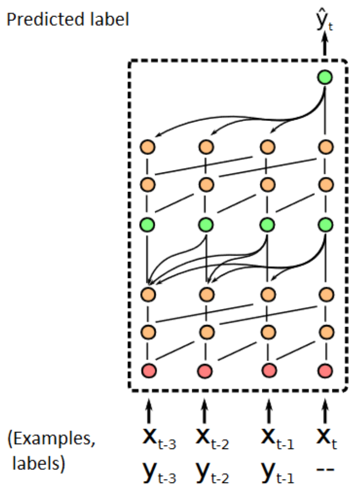
Predicted label
$\hat{y}_t$
(Examples, labels)
x_{t-3} x_{t-2} x_{t-1} x_t
y_{t-3} y_{t-2} y_{t-1} --

Fig. 13.10: Architecture and workflow of SNAIL. Temporal convolution blocks are orange; attention blocks are green. Source: Mishra et al. (2018)

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.

In a similar fashion to memory-augmented neural networks (Santoro et al., 2016) (Subsection 13.4.1), SNAIL also processes task data in sequence, as shown in Figure 13.10. 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 with previously discussed techniques. A downside of SNAIL is that finding the correct architecture of TCBlocks and DenseBlocks can be time consuming.

13.4.4 Conditional neural processes

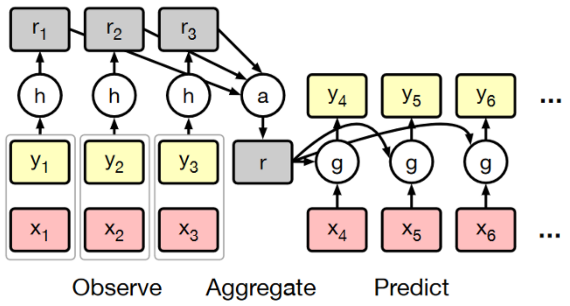

Fig. 13.11: Conditional neural process. Source: Garnelo et al. (2018)

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 13.11. As we can see, the conditional neural process operates in three phases on task $\mathcal{T}_j$. First, it observes the training 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 the 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 $\boldsymbol{r}$ and the new inputs $\boldsymbol{x}$ and produces the predictions $\hat{y}$.

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

$$\mathcal{L}(\boldsymbol{\Theta}) = -\mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})} \left[\mathbb{E}_{\ell \backsim U(0,\ldots,k\cdot N-1)} \left(Q_{\boldsymbol{\Theta}}(\boldsymbol{y}_{\mathcal{T}_j} | D^{c}_{\mathcal{T}_j}, \boldsymbol{x}_{\mathcal{T}_j}) \right) \right] . \tag{13.7}$$

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 Subsection 13.3.2).

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.

Advantages of model-based approaches include the flexibility of the internal dynamics of the systems and their broader applicability compared with 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); Subsection 13.3.3), may not perform well when presented with larger datasets (Hospedales et al., 2020), and generalize less well to more distant tasks than optimization-based techniques (Finn and Levine, 2018). We discuss this optimization-based approach next.

13.5 Optimization-Based Metalearning

Optimization-based techniques adopt a different perspective on metalearning than the previous two approaches. They explicitly optimize for fast learning. Most optimization-based techniques do so by approaching metalearning as a bilevel 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 a input $\boldsymbol{x} \in D^{test}_{\mathcal{T}_j}$ and base-learner parameters $\boldsymbol{\theta}$, optimization-based metalearners return

$$P(Y|\boldsymbol{x}, D^{tr}_{\mathcal{T}_j}) = f_{g_{\boldsymbol{\varphi}(\boldsymbol{\theta}, D^{tr}_{\mathcal{T}_j}, \mathcal{L}_{\mathcal{T}_j})}}(\boldsymbol{x}), \tag{13.8}$$

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

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 13.12. In fact, this is the intuition behind the popular optimization-based technique MAML (Finn et al., 2017).

We now discuss the following optimization-based techniques, in turn:

- LSTM optimizer (Andrychowicz et al., 2016)
- Reinforcement learning optimizer (Ravi and Larochelle, 2017)
- Model-agnostic meta-learning (MAML) (Finn et al., 2017)
- Reptile (Nichol and Schulman, 2018)

13.5.1 LSTM optimizer

Standard gradient update rules have the form

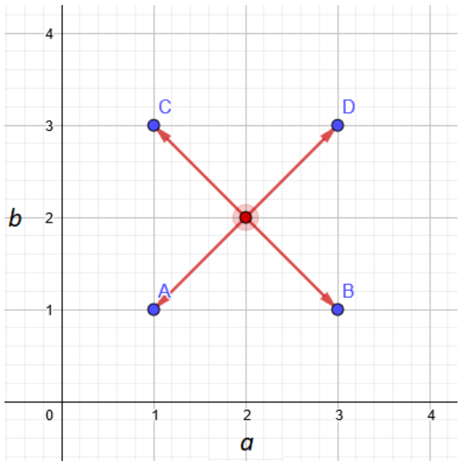

Fig. 13.12: Example of an optimization-based technique, inspired by Finn et al. (2017)

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_t), \tag{13.9}$$

where α is the learning rate and $\mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_t)$ is the loss function with respect to the 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 φ. Then, the new update becomes

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + g_{\varphi}(\nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_t)). \tag{13.10}$$

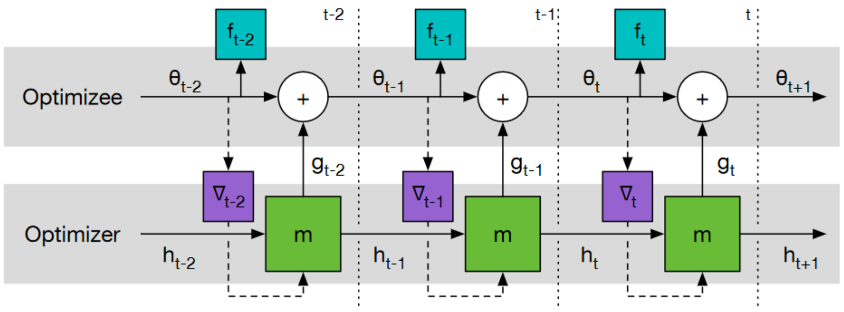

Fig. 13.13: Workflow of the LSTM optimizer. Gradients can only propagate backwards through solid edges. f_t denotes the observed loss at time step t. Source: Andrychowicz et al. (2016)

This new update allows the optimization strategy to be tailored to a specific family of tasks. Note that this is metalearning; i.e., the LSTM learns to learn.

The loss function used to train an LSTM optimizer is

$$\mathcal{L}(\boldsymbol{\varphi}) = \mathbb{E}_{\mathcal{L}_{\mathcal{T}_j}} \left[\sum_{t=1}^{T} w_t \mathcal{L}_{\mathcal{T}_j}(\boldsymbol{\theta}_t) \right], \tag{13.11}$$

where T is the number of parameter updates made, and w_t are weights larger than zero (set to the constant 1 in the original paper (Andrychowicz et al., 2016)). 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 expense associated with the computation thereof. This loss function is fully differentiable and thus allows for training an LSTM optimizer (see Figure 13.13). 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 with hand-crafted optimizers, also on different datasets 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 of whether it can scale to larger base-learner architectures.

13.5.2 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. Now, 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 a 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^*:

$$g^* = argmin_g \, \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T}), \boldsymbol{\theta}_0 \backsim p(\boldsymbol{\theta}_0)} [\mathcal{L}(g(\mathcal{L}_{\mathcal{T}_j}, \boldsymbol{\theta}_0))]. \tag{13.12}$$

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 a recurrent neural network as the 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 to be able to generalize across network architectures and datasets. However, the base-learner architecture that was used was quite small. The question remains of whether this approach can scale to larger architectures.

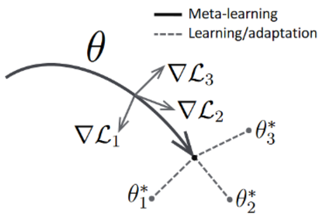

Fig. 13.14: MAML learns an initialization point from which it can perform well on various tasks. Source: Finn et al. (2017)

13.5.3 Model-agnostic meta-learning (MAML)

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 13.14: 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 (i.e., fast weights)

$$\boldsymbol{\theta}'_j = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}), \tag{13.13}$$

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

$$ML := \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) = \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})), \tag{13.14}$$

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:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} ML &= \nabla_{\boldsymbol{\theta}} \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \\
&= \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \\
&= \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}'_j) \\
&= \sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j) \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}(\boldsymbol{\theta})}) \\
&= \underbrace{\sum_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}'_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j)}_{\text{FOMAML}} (\nabla_{\boldsymbol{\theta}} \boldsymbol{\theta} - \alpha \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})), \qquad (13.15)
\end{aligned}
$$

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 test 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 13.15). They found that FOMAML performed reasonably similar implying that most of the benefit comes from the post-update gradients. 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 for the meta-loss in Equation 13.15. One can extend the meta-loss to incorporate multiple gradient steps by substituting $\boldsymbol{\theta}'_j$ by a multi-step variant.

Now, 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*:

$$
\boldsymbol{\theta} := \boldsymbol{\theta} - \beta \nabla_{\boldsymbol{\theta}} \sum_{\mathcal{T}_j \in B} \mathcal{L}_{D^{test}_{\mathcal{T}_j}}(\boldsymbol{\theta}'_j). \qquad (13.16)
$$

The complete training procedure of MAML is displayed in Algorithm 13.2. 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 13.2, 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 enhancements that can improve the 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 metalearning rate β over time, and (v) some batch normalization tweaks (keep running statistics instead of batch-specific ones, and using per-step biases).

```
1: Randomly initialize $\boldsymbol{\theta}$
2: while not done do
3:     Sample batch of $J$ tasks $B = \mathcal{T}_1, \ldots, \mathcal{T}_J \backsim p(\mathcal{T})$
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
```

Algorithm 13.2: One-step MAML for supervised learning, by Finn et al. (2017)

MAML has attracted great attention within the field of metalearning for deep neural networks, 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.

Several other optimization-based techniques build upon MAML. Here, we briefly mention some of them. *Meta-SGD* (Li et al., 2017) does not only learn an initialization point, but also appropriate learning rates for every single parameter in the network. *Latent embedding optimization* (LEO) (Rusu et al., 2018) attempts to find a good initialization in a lower-dimensional space, which can aid generalization performance. Several probabilistic versions of MAML have also been proposed, which learn a distribution over initializations (Grant et al., 2018; Finn et al., 2018) or multiple initializations which are jointly optimized (Yoon et al., 2018). Lastly, one can combine a representation module (e.g., a CNN) with closed-form base-learners, for which one can derive analytical solutions (Bertinetto et al., 2019; Lee et al., 2019).

13.5.4 Reptile

Reptile (Nichol and Schulman, 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 and Schulman, 2018). Algorithm 13.3 displays the pseudocode describing this simple process.

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

Algorithm 13.3: Reptile, by Nichol and Schulman (2018)

Nichol and Schulman (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 and Schulman (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 $\boldsymbol{\theta}^*_j$:

$$min_{\boldsymbol{\theta}}\, \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\frac{1}{2}d(\boldsymbol{\theta}, \boldsymbol{\theta}^*_j)^2]. \tag{13.17}$$

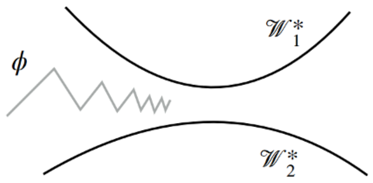

Fig. 13.15: Visualization of Reptile's learning trajectory. Source: (Nichol and Schulman, 2018)

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

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\frac{1}{2}d(\boldsymbol{\theta}, \boldsymbol{\theta}^*_j)^2] &= \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\frac{1}{2}\nabla_{\boldsymbol{\theta}} d(\boldsymbol{\theta}, \boldsymbol{\theta}^*_j)^2] \\ &= \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})}[\boldsymbol{\theta} - \boldsymbol{\theta}^*_j],\end{aligned} \tag{13.18}$$

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 and Schulman (2018) go on to argue that performing gradient descent on this objective would result in the following update rule:

$$\begin{aligned}\boldsymbol{\theta} &= \boldsymbol{\theta} - \epsilon\nabla_{\boldsymbol{\theta}}\frac{1}{2}d(\boldsymbol{\theta}, \boldsymbol{\theta}^*_j)^2 \\ &= \boldsymbol{\theta} - \epsilon(\boldsymbol{\theta}^*_j - \boldsymbol{\theta}).\end{aligned} \tag{13.19}$$

Since we do not know $\boldsymbol{\theta}^*_{\mathcal{T}_j}$, one can approximate this 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 13.17. A visualization is shown in Figure 13.15.

The initialization $\boldsymbol{\theta}$ (ϕ in the figure) is moving towards the optimal weights for tasks 1 and 2 in interleaved fashion (hence the oscillations).

In conclusion, Reptile is an extremely simple metalearning 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, and performance may be a bit worse than that of MAML.

Optimization-based techniques, in conclusion

Optimization-based techniques explicitly optimize for fast learning. 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, which is computationally expensive (Hospedales et al., 2020).

13.6 Discussion and Outlook

In this section, we give a helicopter view of the methods discussed in this chapter, and the field of metalearning for deep neural networks in general.

In recent years, this field has gained much popularity. The idea of self-improving neural networks that can leverage prior learning experience to learn new tasks more quickly is quite appealing. Instead of training a new model from scratch for different tasks, we can use the same (metalearning) model across tasks. As such, metalearning can widen the applicability of powerful deep learning techniques to domains where less data is available and computational resources are limited.

Metalearning techniques for deep neural networks 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 metalearning 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 to prevent the *memorization problem* (meta-overfitting), where the neural network has memorized tasks seen at training time but fails to generalize to new tasks. Clever task design and meta-regularization may prove useful to avoid such problems (Yin et al., 2020). Furthermore, to improve test performance, it is 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 metalearning in the context of deep learning, namely (i) metric-, (ii) model-, and (iii) optimization-based ones, which rely on computing input similarity, task embeddings with states, and 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., 2020). 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-training 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., 2020).

Table 13.1 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 13.2, therefore, provides an overview of all methods and key ideas that we have discussed in this chapter. Recall that $\mathcal{T}_j$ is a task, $D^{tr}_{\mathcal{T}_j}$ the corresponding training set, $\mathcal{L}_{\mathcal{T}_j}$ the loss function, $k_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}_i)$ a kernel function returning the similarity between the two inputs $\boldsymbol{x}$ and $\boldsymbol{x}_i$, y_i are true labels for example inputs $\boldsymbol{x}_i$, $\boldsymbol{\theta}$ are base-learner parameters, and $g_{\boldsymbol{\varphi}}$ is a (learned) optimizer with parameters $\boldsymbol{\varphi}$.

13.6.1 Open challenges

Despite the great potential of metalearning for deep neural networks, there are still open challenges. In addition to the challenges mentioned above (computational costs and the memorization problem), there are two other major challenges. Firstly, most of the metalearning techniques discussed in this chapter are evaluated on narrow benchmark sets. This means that the data that the metalearner 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. Increasing the degree of transferability of prior learning experience is an important and open challenge. As noted by Hospedales et al. (2020), the recently proposed meta-dataset (Triantafillou et al., 2019), could prove to be a great tool towards this end.

Secondly, Chen et al. (2019) have shown that a simple non-metalearning baseline yields competitive or better performance than some metalearning techniques on few-shot image classification. This begs the question of whether metalearning for deep neural networks is a good approach at all in the few-shot setting.

13.6.2 Future research

Future research is needed to address these challenges. Moreover, it is interesting to further investigate the applicability of metalearning for deep neural networks to online, active, and continual/lifelong learning settings (see, e.g., Finn et al. (2018); Yoon et al. (2018); Finn et al. (2019); Munkhdalai and Yu (2017); Vuorio et al. (2018)), as that could increase the applicability of deep learning tools in the real world.

Yet another direction for future research is the creation of *compositional* metalearning 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 $c_1(\boldsymbol{x}), \ldots, c_n(\boldsymbol{x})$, which in turn could help the performance in low-data regimes (Tokmakov et al., 2019). Lastly, the question has been raised of whether some contemporary metalearning techniques for deep neural networks 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 MAML (Finn and Levine, 2018) and found that it largely relies on feature reuse.

We end this chapter by formulating a list of open research questions concerning the challenges that we encounter when applying metalearning to deep neural networks:

- How can we design metalearners that are less susceptible to the memorization problem (Yin et al., 2020)?
- How well do current metalearning techniques generalize to more distant tasks, and how can we increase transferability (Finn and Levine, 2018)?
- How well do metalearning techniques perform in other domains such as active, online, and lifelong learning (Finn et al., 2019; Munkhdalai and Yu, 2017)?
- Can we reduce the computational costs of metalearning systems (Rajeswaran et al., 2019)?
- Can the principle of compositionality be successfully applied to metalearners (Barrett et al., 2018; Lake, 2019)?
- Can we understand why the non-metalearning baseline of Chen et al. (2019) outperforms some state-of-the-art metalearning techniques?
- Is it feasible and helpful to add more meta-abstraction levels, e.g. meta-metalearning, meta-meta-metalearning, etc. (Hospedales et al., 2020)?
- Can we design new metalearning techniques that rely more on rapid learning instead of reusing learned features (Raghu et al., 2020)?

Metalearning for deep neural networks is a vibrant field. Despite great advances, ample challenges remain to be solved, which makes for exciting future work.

References

Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 3988–3996, USA. Curran Associates Inc.

Antoniou, A., Edwards, H., and Storkey, A. (2019). How to train your MAML. In *International Conference on Learning Representations*, ICLR'19.

Barrett, D. G., Hill, F., Santoro, A., Morcos, A. S., and Lillicrap, T. (2018). Measuring abstract reasoning in neural networks. In *Proceedings of the 35th International Conference on Machine Learning*, ICML'18, pages 4477–4486. JMLR.org.

Bertinetto, L., Henriques, J. F., Torr, P. H. S., and Vedaldi, A. (2019). Meta-learning with differentiable closed-form solvers. In *International Conference on Learning Representations*, ICLR'19.

Chen, W.-Y., Liu, Y.-C., Kira, Z., Wang, Y.-C., and Huang, J.-B. (2019). A closer look at few-shot classification. In *International Conference on Learning Representations*, ICLR'19.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, ICML'17, pages 1126–1135. JMLR.org.

Finn, C. and Levine, S. (2018). Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. In *International Conference on Learning Representations*, ICLR'18.

Finn, C., Rajeswaran, A., Kakade, S., and Levine, S. (2019). Online meta-learning. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, ICML'19, pages 1920–1930. JMLR.org.

Finn, C., Xu, K., and Levine, S. (2018). Probabilistic model-agnostic meta-learning. In *Advances in Neural Information Processing Systems 31*, NIPS'18, pages 9516–9527. Curran Associates Inc.

Garcia, V. and Bruna, J. (2017). Few-shot learning with graph neural networks. In *International Conference on Learning Representations*, ICLR'17.

Garnelo, M., Rosenbaum, D., Maddison, C., Ramalho, T., Saxton, D., Shanahan, M., Teh, Y. W., Rezende, D., and Eslami, S. M. A. (2018). Conditional neural processes. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *ICML'18*, pages 1704–1713. JMLR.org.

Grant, E., Finn, C., Levine, S., Darrell, T., and Griffiths, T. (2018). Recasting gradient-based meta-learning as hierarchical bayes. In *International Conference on Learning Representations*, ICLR'18.

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

Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. (2020). Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*.

Koch, G., Zemel, R., and Salakhutdinov, R. (2015). Siamese Neural Networks for One-shot Image Recognition. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *ICML'15*. JMLR.org.

Lake, B. M. (2019). Compositional generalization through meta sequence-to-sequence learning. In *Advances in Neural Information Processing Systems 33*, NIPS'19, pages 9791–9801. Curran Associates Inc.

Lee, K., Maji, S., Ravichandran, A., and Soatto, S. (2019). Meta-learning with differentiable convex optimization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 10657–10665.

Li, K. and Malik, J. (2018). Learning to optimize neural nets. *arXiv preprint arXiv:1703.00441*.

Li, Z., Zhou, F., Chen, F., and Li, H. (2017). Meta-SGD: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835*.

Mishra, N., Rohaninejad, M., Chen, X., and Abbeel, P. (2018). A simple neural attentive meta-learner. In *International Conference on Learning Representations*, ICLR'18.

Munkhdalai, T. and Yu, H. (2017). Meta networks. In *Proceedings of the 34th International Conference on Machine Learning*, ICML'17, pages 2554–2563. JMLR.org.

Nichol, A. and Schulman, J. (2018). Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2:2.

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

Raghu, A., Raghu, M., Bengio, S., and Vinyals, O. (2020). Rapid learning or feature reuse? towards understanding the effectiveness of MAML. In *International Conference on Learning Representations*, ICLR'20.

Rajeswaran, A., Finn, C., Kakade, S. M., and Levine, S. (2019). Meta-learning with implicit gradients. In *Advances in Neural Information Processing Systems 32*, NIPS'19, pages 113–124. Curran Associates Inc.

Ravi, S. and Larochelle, H. (2017). Optimization as a model for few-shot learning. In *International Conference on Learning Representations*, ICLR'17.

Rusu, A. A., Rao, D., Sygnowski, J., Vinyals, O., Pascanu, R., Osindero, S., and Hadsell, R. (2018). Meta-learning with latent embedding optimization. In *International Conference on Learning Representations*, ICLR'18.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. (2016). Meta-learning with memory-augmented neural networks. In *Proceedings of the 33rd International Conference on Machine Learning*, ICML'16, pages 1842–1850. JMLR.org.

Shyam, P., Gupta, S., and Dukkipati, A. (2017). Attentive recurrent comparators. In *Proceedings of the 34th International Conference on Machine Learning*, ICML'17, pages 3173–3181. JMLR.org.

Snell, J., Swersky, K., and Zemel, R. (2017). Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems 30*, NIPS'17, pages 4077–4087. Curran Associates Inc.

Sung, F., Yang, Y., Zhang, L., Xiang, T., Torr, P. H., and Hospedales, T. M. (2018). Learning to compare: Relation network for few-shot learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1199–1208.

Tokmakov, P., Wang, Y.-X., and Hebert, M. (2019). Learning compositional representations for few-shot recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 6372–6381.

Triantafillou, E., Zhu, T., Dumoulin, V., Lamblin, P., Evci, U., Xu, K., Goroshin, R., Gelada, C., Swersky, K., Manzagol, P.-A., et al. (2019). Meta-dataset: A dataset of datasets for learning to learn from few examples. *arXiv preprint arXiv:1903.03096*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems 30*, NIPS'17, pages 5998–6008. Curran Associates Inc.

Vinyals, O. (2017). Talk: Model vs optimization meta learning. `http://metalearning-symposium.ml/files/vinyals.pdf`. Neural Information Processing Systems (NIPS); accessed 06-06-2020.

Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., and Wierstra, D. (2016). Matching networks for one shot learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 3637–3645, USA. Curran Associates Inc.

Vuorio, R., Cho, D.-Y., Kim, D., and Kim, J. (2018). Meta continual learning. *arXiv preprint arXiv:1806.06928*.

Yin, M., Tucker, G., Zhou, M., Levine, S., and Finn, C. (2020). Meta-learning without memorization. In *International Conference on Learning Representations*, ICLR'20.

Yoon, J., Kim, T., Dia, O., Kim, S., Bengio, Y., and Ahn, S. (2018). Bayesian model-agnostic meta-learning. In *Advances in Neural Information Processing Systems 31*, NIPS'18, pages 7332–7342. Curran Associates Inc.