Universiteit Leiden

Master Computer Science

Revisiting Learned Optimizers in Few-Shot Settings

Name:	Mike Huisman
Student ID:	s2581299
Date:	20-01-2021
Specialisation:	Data Science
1st supervisor:	Jan N. van Rijn
2nd supervisor:	Aske Plaat

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

A key characteristic of human intelligence is the ability to learn new tasks quickly. While deep learning techniques have demonstrated super-human level performance on some tasks, their ability to learn quickly is limited. Meta-learning is one approach to bridge this gap between human and deep learning. In this thesis, we show the theoretical relationship between two state-of-the-art meta-learning techniques. Combining this theoretical insight with recent empirical findings, we construct TURTLE, a new meta-learning algorithm that is more general than MAML yet simpler than the LSTM meta-learner. We find that TURTLE outperforms both techniques at sine wave regression and, without additional hyperparameter tuning or large increase in computational costs, exceeds their performance in challenging image classification settings by at least 1% accuracy. With further hyperparameter tuning, we expect to see a larger improvement. Our findings highlight the importance of second-order gradients for successfully training TURTLE and open the door to fruitful future research. All of our code has been made available online at: `https://github.com/mikehuisman/revisiting-learned-optimizers`.

1 Introduction

Human intelligence is characterized by the ability to learn quickly. While artificial neural networks can achieve (super-)human level performance on various tasks (He et al., 2015; Mnih et al., 2015; Silver et al., 2016), they require far more examples than their biological counterparts. Bridging this gap between artificial and human intelligence could extend the applicability of deep neural networks to domains where large data sets and/or sufficient computational resources are unavailable (Hospedales et al., 2020; Huisman et al., 2020).

Meta-learning is one approach to do that (Thrun, 1998; Schmidhuber, 1987; Naik and Mammone, 1992). The key idea is to extract a *prior* from learning experience on previous tasks to learn new ones from fewer examples (Vanschoren, 2018). This prior can take on many forms such as the neural architecture (Elsken et al., 2020), the initialization weights of a given network architecture (Finn et al., 2017; Nichol et al., 2018), or even a learned optimization procedure (Andrychowicz et al., 2016; Ravi and Larochelle, 2017).

In our work, we focus on two state-of-the-art meta-learning techniques: MAML (Finn et al., 2017) and the LSTM meta-learner (Ravi and Larochelle, 2017). These two techniques aim to learn a prior of the last two types. More specifically, MAML aims to find a set of initial parameters for a given neural network architecture from which it can learn new tasks quickly within few gradient update steps. The LSTM meta-learner also attempts to find such an initialization but instead of using vanilla gradient descent, it learns its own optimization procedure using a *meta-network* that updates the weights of the *base-learner* network. Note that there are thus two networks: the base-learner network which attempts to solve the task and make predictions from inputs, and the meta-network which updates the weights of the base-learner network in order to increase its performance. This architectural difference between MAML and the LSTM meta-learner is displayed in Figure 1.

Recently, there have been two observations in meta-learning that form a direct motivation for our work. The first observation is that Finn et al. (2017) have shown that their algorithm, MAML, outperforms the LSTM meta-learner (Ravi and Larochelle, 2017) on image classification. We find this result highly surprising for two reasons. Firstly, one would expect that the meta-network used by the LSTM meta-learner could learn to perform gradient descent and, as a consequence, perform at least as well as MAML (we prove this in Section 3). Secondly, findings by Andrychowicz et al. (2016) suggest that techniques that learn an optimizer, such as the LSTM meta-learner, can find better learning

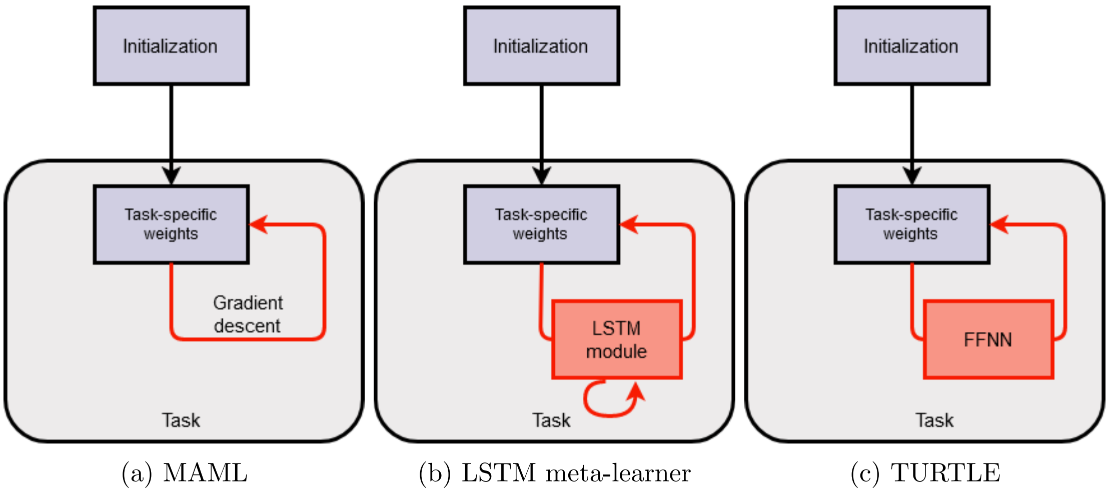


Figure 1: Architectural design of MAML, the LSTM meta-learner, and TURTLE (our proposed technique). Whereas MAML uses gradient descent to learn new tasks, the LSTM meta-learner and TURTLE use trainable meta-networks to update the weights of the base-learner network. The LSTM meta-learner uses a special LSTM module as meta-network, while TURTLE uses a simple feed-forward neural network (FFNN).

rules than vanilla gradient descent, which is used by MAML. In short, one would expect that learning an optimizer in addition to good base-learner initialization parameters would yield a better performance.

The second observation is that commonly used benchmarks for testing meta-learning algorithms may not have been challenging enough to properly assess their learning ability. That is, Chen et al. (2019) have shown that four state-of-the-art meta-learning techniques do not work so well compared with simple transfer learning techniques when presented with new tasks sampled from a different data set than the one used for training. These four techniques are: matching networks (Vinyals et al., 2016), prototypical networks (Snell et al., 2017), relation networks Sung et al. (2018), and MAML (Finn et al., 2017). Techniques that learn an optimizer such as the LSTM meta-learner, however, were not studied. We think that such techniques may outperform the four techniques studied by Chen et al. (2019) when task distribution shifts occur as (i) they can learn from the tasks coming from the new distribution in contrast to matching, prototypical, and relation networks which do not make network changes at test time (Huisman et al., 2020), and (ii) they may learn a learning procedure that can learn new tasks faster than vanilla gradient descent which is used by MAML.

In short, these two observations and expectations can be captured by the following single hypothesis, which we will investigate in the rest of this work.

Research hypothesis: *Learning an optimizer in addition to the meta-learned initial parameters should enable faster learning on regular few-shot learning benchmarks as well as more challenging settings where task distribution shifts occur*

We start investigating this hypothesis by establishing the theoretical relationship between MAML and the LSTM meta-learner. More specifically, we prove that the latter is more expressive than MAML, which means that—in theory—it could achieve the same performance. Inspired by this theoretical result, we formulate two hypotheses as to why the LSTM meta-learner fails to find a solution that works at least as well as the one found by MAML. These hypotheses lead us to construct a new meta-learning algorithm called TURTLE, which we design to be simpler than the LSTM meta-learner, yet more expressive than MAML. We empirically investigate and tune TURTLE, and find that it outperforms both MAML and the LSTM meta-learner on a simple regression problem and, without additional hyperparameter tuning, on a commonly used few-shot image classification benchmark, and a more challenging setup in which a task distribution shift occurs. The key to TURTLE's success is the fact that it uses second-order gradients, which are ignored by the LSTM meta-learner. Additionally, we compare the running time of TURTLE with that of MAML and the LSTM meta-learner and find that TURTLE is not much slower than the full version of MAML.

In short, our contributions can be summarized as follows:

- We prove the theoretical relationship between MAML and the LSTM meta-learner.
- We propose a new meta-learning algorithm called *TURTLE* which is more general than MAML, yet simpler than the LSTM meta-learner.
- We empirically demonstrate that TURTLE outperforms both the LSTM meta-learner and MAML at sine wave regression and various challenging settings involving commonly used image classification benchmarks.

The rest of this thesis is structured as follows. Section 2 provides a description of supervised learning in neural networks, the associated problems, and existing transfer- and meta-learning approaches to address these issues. In Section 3, we show the theoretical relationship between the LSTM meta-learner and MAML, relate this to recent empirical findings, and propose a new meta-learning algorithm called TURTLE. Section 4 covers our experimental setup and results. Section 5 discusses the obtained findings, and Section 6 concludes.

2 Background

In this section, we give an introduction to supervised learning and indicate its shortcomings. Next, we discuss some popular techniques from the fields of transfer learning and meta-learning to overcome these limitations. Table 1 contains an overview of the notation that we will use throughout this work.

2.1 Supervised learning

On a very high level, we wish to have computers that can perform certain tasks. Instead of manually programming the wanted behavior, we can write a program that learns how to perform a given task through experience (Goodfellow et al., 2016).

Suppose that we want our program P to learn some task $\mathcal{T}_h$. For example, the task could be to tell whether there are tumors in brain scan images. As input, we would then feed an image of a brain scan into the program, and we want the output of this program to be a binary classification decision (tumor or not a tumor).

Term	Meaning
$\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})$	A task consisting of a support set $D^{tr}_{\mathcal{T}_j}$ and query set $D^{te}_{\mathcal{T}_j}$
$D^{tr}_{\mathcal{T}_j}$	Support set of the task which is used for training
$D^{te}_{\mathcal{T}_j}$	Query set of the task which is used for evaluating the success of learning on the support set
N	The number of classes in the support set of a task in N-way k-shot classification
k	The number of examples per class in the support set of a task in few-shot learning
$\boldsymbol{\theta}$	Initialization weights of the base-learner network
$f_{\boldsymbol{\theta}}$	A neural network with parameters $\boldsymbol{\theta}$
$g_{\boldsymbol{\phi}}$	A meta-network with parameters $\boldsymbol{\phi}$. This network makes updates to the weights of the base-learner network f
$\nabla_{\circ}\varphi$	The gradients of φ with respect to $\circ$
Embedding module	The module of a neural network neural network without the output layer

Table 1: Notation that we will use throughout this work.

In the context of deep learning, our program corresponds to a neural network f with parameters $\boldsymbol{\theta}$. For simplicity, we will assume that the architecture of this network is fixed. Learning the task then corresponds to tweaking the parameters $\boldsymbol{\theta}$ of our network such that it can perform the given task as well as possible.

The learning objective can be formalized if we define a loss function $\mathcal{L}_{\mathcal{T}_h}$ that captures how well our model performs on the task. That is, we want to minimize the loss on the task by finding the optimal parameters

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_h}(f_{\boldsymbol{\theta}}). \tag{1}$$

In *supervised learning*, this learning process is based on some examples. This means that we show our network various examples of brain scan images and the corresponding labels (tumor or no tumor). The idea is that by exposing our network to these examples, it can learn to correctly classify such images. More generally, we have a finite data set $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{M}$ of M such examples where the $\boldsymbol{x}_i$ are inputs (e.g., brain scan images) and the y_i are the corresponding ground-truth outputs (e.g., tumor classification decision).

Since our model learns from a finite set of examples, the risk exists that the network memorizes all of the examples that we have given it, while it is not able to make good predictions on unseen inputs. In other words, our network may not *generalize* well to new inputs. To overcome this issue, we can split our entire data set into three disjoint partitions: (i) a training set D^{tr}, (ii) a validation set D^{val}, and (iii) a test set D^{te}. The training set is then used for learning (updating the network parameters to improve its performance), the validation set to measure the generalization ability of the network and to do hyperparameter tuning, and the test set for obtaining a final performance estimate of how good our model is at the given task (Goodfellow et al., 2016). Alternatively, one could use special techniques such as cross-validation.

There are various ways to leverage these data set splits for learning. For example, we could train our network on the examples in the training set and evaluate its performance on the validation set after every update of the network parameters in order to see whether the performance on the validation set (unseen examples) improves. Then, we can stop learning as soon as the validation performance no longer improves, or starts to worsen. In this way, we aim to prevent the problems associated with memorization, and maximize the generalization performance of the network.

In practice, finding the optimal parameters $\boldsymbol{\theta}^*$ from Equation 1 is often unfeasible. However, it is still possible to approximate these optimal parameters by, e.g., iterative gradient-based procedures such as Adam (Kingma and Ba, 2015). A generic training procedure is displayed in Algorithm 1. As one can see, a series of T updates (possibly until convergence) are made to the model parameters. At every time step t, a batch of data B from the training set D^{tr} (line 3) is used to compute the loss value (line 4). The gradients of this loss can then be used by an optimization procedure to update $\boldsymbol{\theta}_t$ (line 5).

Algorithm 1 Generic training procedure

```
1: Randomly initialize weights $\boldsymbol{\theta}_0$
2: for $t = 1, ..., T$ do                                    ▷ Or until convergence
3:     Sample batch $B$ of examples $(\boldsymbol{x}_i, y_i)$ from $D^{tr}$
4:     Compute $\mathcal{L}_{t-1} = \mathcal{L}_B(f_{\boldsymbol{\theta}_{t-1}})$
5:     Compute $\boldsymbol{\theta}_t$ using $\nabla_{\boldsymbol{\theta}_{t-1}}(\mathcal{L}_{t-1})$
6: end for
```

As mentioned in Section 1, great successes have been achieved with these iterative procedures. However, these successes heavily rely on the presence of abundant data, and the availability of sufficient computational resources. In other words, these techniques require many examples in order to achieve good performance. This makes deep learning techniques inapplicable to many real-world domains where data is not abundant, and computational resources are limited (Hospedales et al., 2020). Consequently, researchers have focused on the design of techniques that allow neural networks to learn more quickly, that is, from fewer data and with less computational resources.

This goal has given rise to two highly similar fields of study, namely *transfer learning* and meta-learning. We discuss both fields and some associated techniques which we will use in our work, in turn.

2.2 Transfer learning

The goal of transfer learning is to *transfer* knowledge from a source domain to a (different) target domain (Pan and Yang, 2009; Taylor and Stone, 2009). For example, suppose we have a neural network $f_{\boldsymbol{\theta}}$ which we trained on data set D. Now, suppose we wish to learn a program for a new data set D'. While many transfer learning techniques exist to do this, we limit ourselves to two transfer learning techniques[1] that we use as baselines in our work.

2.2.1 Finetuning (FT)

The *finetuning* approach that we use, attempts to transfer previously acquired knowledge from data set D to the new data set D'. This technique leverages the insight that layers of a neural networks follow a hierarchical representation pattern. That is, input representations become more abstract deeper down the network (closer to the output

[1]Chen et al. (2019) have shown that these techniques should not be underestimated as some of them are able to perform better than state-of-the-art meta-learning techniques in challenging scenarios.

layer) (Simonyan et al., 2014; Li et al., 2014; Mahendran and Vedaldi, 2016). Thus, the most abstract input representation is obtained in the final hidden layer, which is one layer before the output layer.

The finetuning technique works in two phases. In the pre-training phase, we train the network f_θ to minimize the loss on data set D using Algorithm 1. Then, in the finetuning phase, we freeze the input representation module (the hidden layers) and re-train the output layer on new data set D' (also using Algorithm 1). The intuition is that learning data set D' may be quicker as we have already learned all weights in the hidden layers.

2.2.2 Centroid-based finetuning (CFT)

A slight variation of the finetuning strategy, which is specific to classification problems, is called *centroid-based finetuning* (CFT) which uses a special output layer. This output layer learns class *centroid* vectors $\boldsymbol{w}_c$, which capture characteristics of classes $c \in C$ in the data. These centroid vectors can be interpreted as latent representations of class concepts in high-dimensional space. The technique makes class predictions as follows: given an input $\boldsymbol{x}_i$, we compute its embedding in this high-dimensional space, and compare the similarity between this input representation and all centroid vectors. The class of the centroid vector which yields the greatest similarity is then predicted.

To put this more precisely, class predictions $\hat{y}_i$ are made by comparing the abstract representation $f_{\boldsymbol{\theta}^{(1:L-1)}}(\boldsymbol{x}_i)$ of input $\boldsymbol{x}_i$ with every centroid $\boldsymbol{w}_c$ and to predict the class c of which the centroid is most similar to $f_{\boldsymbol{\theta}^{(1:L-1)}}(\boldsymbol{x}_i)$, given a similarity measure $s : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. Here, L is the number of layers in our network and d the dimensionality of the abstract input representations and class centroids. Equation 2 formalizes this prediction mechanism.

$$\hat{y}_i = \arg\max_{c \in C} s(f_{\boldsymbol{\theta}^{(1:L-1)}}(\boldsymbol{x}_i), \boldsymbol{w}_c) \tag{2}$$

Following Chen et al. (2019), we will use the cosine similarity as measure s, which is given by

$$s(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{\boldsymbol{x}_i^T \boldsymbol{x}_j}{||\boldsymbol{x_i}|| \cdot ||\boldsymbol{x}_j||}, \tag{3}$$

where $||\boldsymbol{x}|| = \sqrt{\sum_m x_m^2}$ denotes the 2-norm, or length, of vector $\boldsymbol{x}$.

Importantly, note that this technique is not applicable to regression problems, where outputs y_i are real-valued numbers. Also, the number of classes seen during training on data set D may greatly vary from the number of classes in the new data set D'. This is not a problem since all hidden layers will be frozen and an entirely new output layer will be initialized once data set D' is presented. This new output layer can then be fine-tuned for a number of update steps on the new data set D'.

Centroid-based finetuning is closely related to prototypical networks (Snell et al., 2017), which also learns class prototypes/centroids. We refer interested readers in this kind of distance-based classification to Gidaris and Komodakis (2018) and Qi et al. (2018).

2.3 Meta-learning

In its broadest sense, transfer learning techniques aim to leverage previously obtained knowledge to learn new tasks quicker. Under this interpretation, meta-learning can be seen as a subset of transfer learning (Huisman et al., 2020), as meta-learning techniques attempt to leverage previous learning experience to learn new tasks quicker. However, instead of training in regular fashion (on a single data set D) as done by non-meta-learning, transfer learning techniques, contemporary meta-learning techniques train on tasks $\mathcal{T}_j$ (special subsets of data). This special training procedure allows neural networks to explicitly train for fast adaptation to new tasks. This explicit goal is called a *meta-objective*.

Whereas the regular supervised learning objective is to minimize the loss on a single data set D (Equation 1), the meta-objective is to minimize the loss on a distribution of tasks $p(\mathcal{T})$, i.e.,

$$\mathcal{L}_{p(\mathcal{T})} = \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_j}(f_{\boldsymbol{\theta}'_j}), \tag{4}$$

where $\boldsymbol{\theta}'_j$ represent the task-specific network weights obtained by training on task $\mathcal{T}_j$.

2.3.1 N-way k-shot classification

The most commonly used method to train and evaluate meta-learning algorithms is called *N-way k-shot classification.* This method consists of three phases: the i) *meta-training*, ii) *meta-validation*, and iii) *meta-test* stages. As their names suggest, the meta-training phase is used for training the meta-learning algorithm, the meta-validation stage for tuning hyperparameters of the meta-learner to maximize generalization performance, and the meta-test phase for obtaining a final performance estimate. Every phase is associated with a disjoint set of classes, meaning that tasks from two distinct phases cannot contain examples with the same class.

Thus, suppose all classes from data set D are given by $C = \{c_1, ..., c_\ell\}$. After random shuffling, we can split this set of labels into three partitions: one for each phase, giving us C^{tr}, C^{val}, and C^{te}. Afterwards, we can create three data sets $D^{split} = \{(\boldsymbol{x}_i, y_i) \,|\, y_i \in C^{split}\}$ for $split \in \{tr, val, te\}$. These data sets D^{split} contain all examples from D with classes chosen from their respective class partition C^{split}.

Then, in a given stage with corresponding partition *split*, tasks are constructed following the N-way k-shot principle. Every task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})$ consists of a *support* set $D^{tr}_{\mathcal{T}_j}$ and a *query* set $D^{te}_{\mathcal{T}_j}$. The N-way k-shot principle states that every support set must contain exactly N classes and k examples per class. Furthermore, the principle requires that the every input in the query set $D^{te}_{\mathcal{T}_j}$ has a class that was present in the support set $D^{tr}_{\mathcal{T}_j}$. This is visualized in Figure 2. In this figure, every task $\mathcal{T}_j$ consists of a support set $D^{tr}_{\mathcal{T}_j}$ and query set $D^{te}_{\mathcal{T}_j}$. As we can see, every support set $D^{tr}_{\mathcal{T}_j}$ contains exactly 5 classes, implying that $N = 5$. Furthermore, there is precisely one example per class, which means that $k = 1$. Lastly, note that the query set $D^{te}_{\mathcal{T}_j}$ contains inputs with classes that are present in the support set $D^{te}_{\mathcal{T}_j}$.

The question remains how the task is presented to the model. In an *episode* with corresponding task $\mathcal{T}_j$, the model is presented with the support set $D^{tr}_{\mathcal{T}_j}$. It can use these examples to make updates to compute task-specific weights $\boldsymbol{\theta}_j$ from its initialization parameters $\boldsymbol{\theta}$. If it has learned well from the examples in $D^{tr}_{\mathcal{T}_j}$, we expect the performance to be good on the query set $D^{te}_{\mathcal{T}_j}$ of the same task since, by construction, it contains the same classes that were present in the support set which was used for training.

In short, for a given task, the objective is to maximize the performance on the query set, conditioned on the support set. Intuitively, the model is given a set of data to learn from (support set), and the goal is to learn from it in such a way that facilitates generalization (measured on the query set). Note that this is precisely what we want: we are after techniques that can learn well.

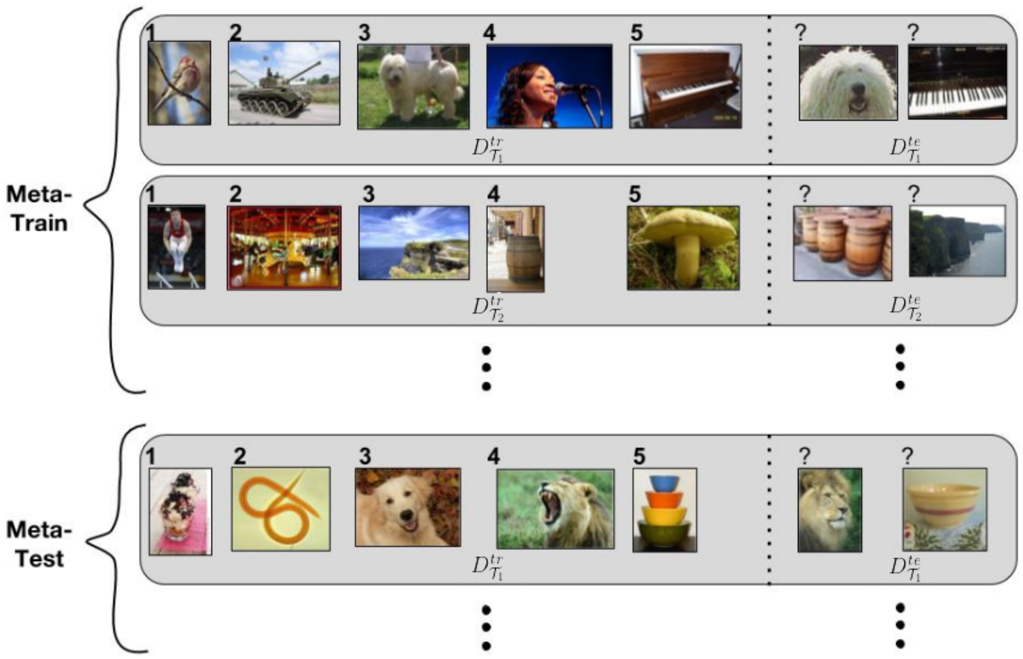


Figure 2: The N-way k-shot classification setup. Every box corresponds to a task consisting of a support set (left split) and query set (right split). There are $N = 5$ classes in every support set and $k = 1$ example per class. Note that the meta-validation stage is not displayed. Adapted from Ravi and Larochelle (2017)

.

When k is small, this N-way k-shot classification is also called *few-shot learning*, as the network is supposed to learn from only few examples. We now discuss two popular meta-learning techniques that can exploit these task structures to learn new tasks quicker.

2.3.2 Model-agnostic meta-learning (MAML)

Model-agnostic meta-learning (Finn et al., 2017), or MAML, is arguably the most popular technique for meta-learning. Its objective is to find good initialization parameters $\boldsymbol{\theta}$ for the base-learner, from which we can quickly learn new tasks $\mathcal{T}_j$ using vanilla, or regular, gradient descent (see Equation 7).

To clarify the intuition behind MAML, suppose that we have a regression problem where every task corresponds to a function $f(x) = ax + b$. Let us assume there are four of such tasks: A, B, C, and D. The optimal parameters for these tasks are displayed as blue dots in Figure 3. From here, it is easy to see that some initializations allow us to learn these tasks more quickly than others. In this case, the red dot corresponds to the best initialization point, since it allows us to quickly move towards the optimal parameters for all tasks (assuming we can move equally fast in every direction). Such a central point is precisely what MAML tries to find.

More generally, suppose we have a distribution of tasks $p(\mathcal{T})$ and wish to learn a good initialization $\boldsymbol{\theta}$ that can be adapted quickly by *one step* of gradient descent on the support

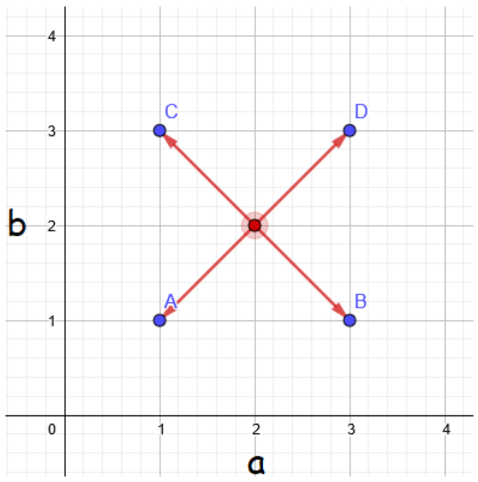


Figure 3: Intuition behind MAML (Finn et al., 2017). The red dot denotes such initialization point, and the arrows represent gradient update steps specific to the tasks A, B, C, and D. Source: Huisman et al. (2020).

set (see Equation 7). Thus, we wish to find an optimal initialization

$$\begin{aligned}\boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})} \left[\mathcal{L}_{D^{te}_{\mathcal{T}_j}} (f_{\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}})}) \right] \\ &= \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})} \left[\mathcal{L}_{D^{te}_{\mathcal{T}_j}} (f_{\boldsymbol{\theta}'_j}) \right], \end{aligned} \tag{5}$$

where $\boldsymbol{\theta}'_j$ denote the task-specific parameters $(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}}))$, also called *fast weights*. This theoretical loss takes the expectation over a distribution of all tasks. In practice, this expectation over tasks can be approximated by computing the loss on a randomly sampled batch of tasks $\mathcal{T}_j$. Gradients of this loss can then be used to update the initialization parameters $\boldsymbol{\theta}$.

Pseudocode for MAML is displayed in Algorithm 2. We start with a random initialization point $\boldsymbol{\theta}$ (line 1), and iteratively adapt this to facilitate quick task-specific adaptation. In every outer loop iteration, we sample a batch of task B (line 3) in order to approximate the expectation value in Equation 5. Then, for ever task, we initialize fast (task-specific) weights $\boldsymbol{\theta}'_j = \boldsymbol{\theta}$ (line 5). These fast weights are iteratively updated for T time steps on the support set (lines 6–9). The final fast weights $\boldsymbol{\theta}'_j$ are then used to compute the loss on the query set (line 10), which is propagated backwards to update the initialization point $\boldsymbol{\theta}$ (line 12).

A disadvantage of MAML is that updating the initialization weights according to Equation 5 requires one to perform back-propagation through the optimization trajectories. To see this, we can unpack the gradient term in line 12 of the algorithm

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j}) &= \mathcal{L}'_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j}) \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}'_j) \\ &= \mathcal{L}'_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j}) \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}}) - \ldots - \alpha \nabla_{\boldsymbol{\theta}_j^{(T-1)}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}^{(T-1)}})). \end{aligned} \tag{6}$$

Since the gradient of a sum is equal to the sum of the gradients of its parts, we obtain an expression that contains gradients of gradients in the second factor. These so-called *second-order* derivatives are computationally expensive to compute: quadratic in the

Algorithm 2 MAML, adapted from Finn et al. (2017)

```
1: Start with random initialization parameters $\boldsymbol{\theta}$
2: repeat
3:     Sample batch of $m$ tasks $B = \{(D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})\}^m_{j=1}$
4:     for $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j}) \in B$ do
5:         Fast weights $\boldsymbol{\theta}'_j = \boldsymbol{\theta}$
6:         for $t = 1, ..., T$ do
7:             Compute $\mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j})$ and $\nabla_{\boldsymbol{\theta}'_j}\mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j})$
8:             Update $\boldsymbol{\theta}'_j = \boldsymbol{\theta}'_j - \alpha\nabla_{\boldsymbol{\theta}'_j}\mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j})$
9:         end for
10:        Compute $\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j})$ and $\nabla_{\boldsymbol{\theta}}\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j})$
11:    end for
12:    Update initialization weights $\boldsymbol{\theta} = \boldsymbol{\theta} - \beta\sum_{\mathcal{T}_j \in B}\nabla_{\boldsymbol{\theta}}\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j})$
13: until convergence
```

number of base-learner parameters. Fortunately, it is possible to approximate the performance of second-order MAML (so-MAML) by ignoring the optimization trajectory, which is equivalent to stating that $\nabla_{\boldsymbol{\theta}}\boldsymbol{\theta}'_j = I$. This variant of MAML uses only first-order gradients (fo-MAML) and yields similar performance to so-MAML.

The intuition as to why the first-order approximation works just as well as the second-order variant is displayed in Figure 4. That is, first-order updates to the base-learner initialization correspond to moving the initialization into the opposite direction of the query loss gradient with respect to the fast weights $\boldsymbol{\theta}^{(T)}_j$, i.e., $\mathcal{L}'_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}^{(T)}_j})$. In this figure, $T = 4$ updates are made to the initial parameters on the support set $D^{tr}_{\mathcal{T}_j}$ of task $\mathcal{T}_j$ (red arrows). After adjusting the initial parameters to $\boldsymbol{\theta}^{(T)}_j$, the gradient descent direction of the fast weights $\boldsymbol{\theta}^{(4)}_j$ on the query set $D^{te}_{\mathcal{T}_j}$ is computed. If the error landscape is reasonably smooth, this blue vector—which is used to update the initialization parameters $\boldsymbol{\theta}$—presumably points towards the optimal parameters $\boldsymbol{\theta}^*_j$ for task $\mathcal{T}_j$. It is thus not hard to believe that the first-order MAML approximation will work just as well as the second-order variant. When multiple tasks are used, this blue update vector is constructed by averaging the negative query loss directions across the tasks, which presumably moves the initialization to a more central position from which it can learn these tasks quicker (see Figure 3).

2.3.3 LSTM meta-learner

On top of learning an initialization, the LSTM meta-learner (Ravi and Larochelle, 2017) attempts to learn the learning procedure, building upon findings by Andrychowicz et al. (2016). Thus, Ravi and Larochelle (2017) propose to replace gradient descent, which is used by MAML, by a trainable LSTM module $g_{\boldsymbol{\phi}}$ which makes updates to the base-learner parameters. More specifically, the base-learner parameters $\boldsymbol{\theta}$ are placed into the cell state of the LSTM meta-learner. As a consequence, cell state updates correspond to base-learner weight updates.

This one-to-one correspondence between base-learner parameters and cell states is visually depicted in Figure 5. That is, the base-learner parameters $\boldsymbol{\theta}$ are literally inside the LSTM meta-learner. In a given episode with corresponding task $\mathcal{T}_j$, the LSTM meta-

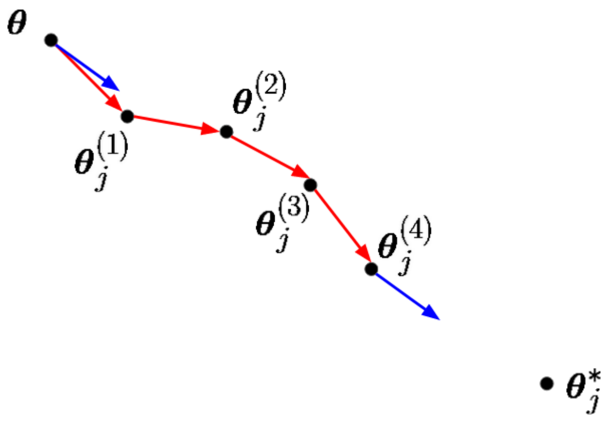

$\boldsymbol{\theta}$
$\boldsymbol{\theta}_j^{(1)}$
$\boldsymbol{\theta}_j^{(2)}$
$\boldsymbol{\theta}_j^{(3)}$
$\boldsymbol{\theta}_j^{(4)}$
$\boldsymbol{\theta}_j^*$

Figure 4: Demonstration of first-order MAML that makes 4 inner updates (red arrows) on the support set of a given task $\mathcal{T}_j$. By the first-order assumption, the initialization θ is moved in the direction proposed by gradient descent on the query set (blue arrow)—which presumably points towards the optimal parameters $\boldsymbol{\theta}_j^*$ for task $\mathcal{T}_j$. No backpropagation through the inner optimization trajectory takes place. This image is inspired by Rajeswaran et al. (2019).

learner randomly samples batches $(\boldsymbol{X}, \boldsymbol{Y})$ from the support set $D^{tr}_{\mathcal{T}_j}$ to update its cell state (and thus base-learner parameters). Then, after a fixed number of updates T, the loss of the final parameters is evaluated on the query set $D^{te}_{\mathcal{T}_j}$, on which we wish to maximize performance. The gradients of the observed loss can be then used to update the LSTM parameters $\boldsymbol{\phi}$ correspondingly.

This idea was inspired by the popular gradient descent update rule, given in Equation 7.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}_t}). \tag{7}$$

Ravi and Larochelle (2017) note that this update rule looks very similar to that of the cell state (long-term memory component) of an LSTM, given by

$$\boldsymbol{c}_{t+1} = \boldsymbol{p}_t \odot \boldsymbol{c}_t + \boldsymbol{i}_t \odot \bar{\boldsymbol{c}}_t. \tag{8}$$

Here, $\boldsymbol{c}_t$ is the cell state, $\boldsymbol{p}_t$ the forget gate, $\boldsymbol{i}_t$ the learning rate, $\bar{\boldsymbol{c}}_t$ the candidate cell state, and $\odot$ the element-wise product. When $\boldsymbol{p}_t = \mathbf{1}$, $\boldsymbol{c}_t = \boldsymbol{\theta}_t$, $\boldsymbol{i}_t = \alpha$, and $\bar{\boldsymbol{c}}_t = -\nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}_t})$, this cell state update rule is equivalent to gradient descent applied to base-learner parameters $\boldsymbol{\theta}_t$.

For this reason, the LSTM meta-learner sets the initial base-learner parameters and cell state to be equal, i.e., $\boldsymbol{\theta}_0 = \boldsymbol{c}_0$, and maintains this one-to-one correspondence over time, as depicted in Figure 5. Also, $\bar{\boldsymbol{c}}_t$ is set to $-\nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{D^{tr}_{\mathcal{T}_t}}(f_{\boldsymbol{\theta}_t})$. For increased expressivity in the meta-learner, Ravi and Larochelle (2017) define the learning rate $\boldsymbol{i}_t$ and forget gate $\boldsymbol{p}_t$ in a parametric way. More specifically, they make the learning rate vector $\boldsymbol{i}_t$ dependent on the loss gradient, loss value, previous parameter value, and previous learning rate, i.e.,

$$\boldsymbol{i}_t = \sigma(\boldsymbol{w}_I \odot [\nabla_{\boldsymbol{\theta}_{t-1}} \mathcal{L}_{D^{tr}_{\mathcal{T}_t}}(f_{\boldsymbol{\theta}_{t-1}}), \mathcal{L}_{D^{tr}_{\mathcal{T}_t}}(f_{\boldsymbol{\theta}_{t-1}}), \boldsymbol{\theta}_{t-1}, \boldsymbol{i}_{t-1}] + \boldsymbol{b}_I). \tag{9}$$

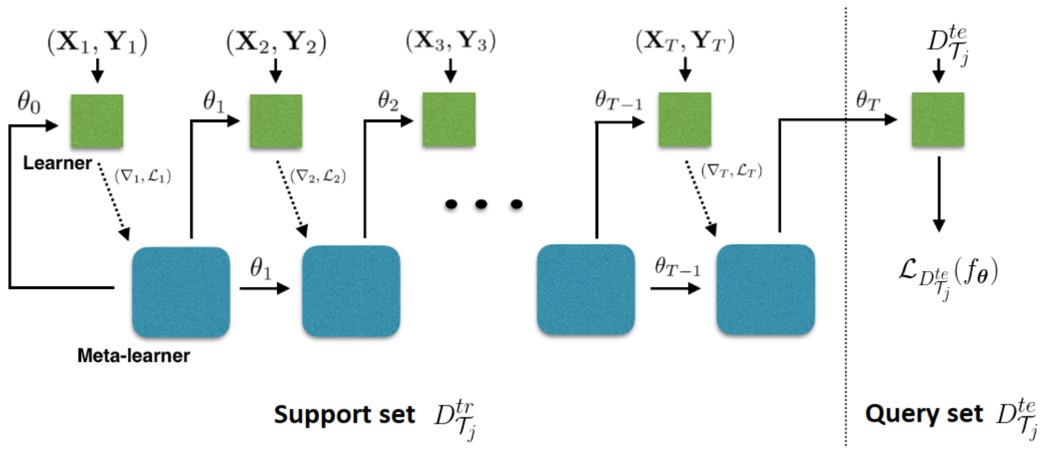

$(\mathbf{X}_1, \mathbf{Y}_1)$
$(\mathbf{X}_2, \mathbf{Y}_2)$
$(\mathbf{X}_3, \mathbf{Y}_3)$
$(\mathbf{X}_T, \mathbf{Y}_T)$
$D^{te}_{\mathcal{T}_j}$
θ_0
θ_1
θ_2
θ_{T-1}
θ_T
Learner
$(\nabla_1, \mathcal{L}_1)$
$(\nabla_2, \mathcal{L}_2)$
$(\nabla_T, \mathcal{L}_T)$
$\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}})$
θ_1
θ_{T-1}
Meta-learner
Support set $D^{tr}_{\mathcal{T}_j}$
Query set $D^{te}_{\mathcal{T}_j}$

Figure 5: The operation of an LSTM meta-learner in a single episode associated with task $\mathcal{T}_j$. $\mathcal{L}_t$ and ∇_t are shorthand for $\mathcal{L}_{(\boldsymbol{X}_t, \boldsymbol{Y}_t)}(f_{\boldsymbol{\theta}_{t-1}})$ and $\nabla_{\boldsymbol{\theta}_{t-1}} \mathcal{L}_{(\boldsymbol{X}_t, \boldsymbol{Y}_t)}(f_{\boldsymbol{\theta}_{t-1}})$ respectively. The meta-learner network makes T updates to the base-learner parameters using the support set. The resulting parameters $\boldsymbol{\theta}_T$ are then evaluated on the query set. The meta-learner network can be updated by propagating the query loss backwards through this computational graph. To avoid the computation of second-order derivatives, Ravi and Larochelle (2017) disallow gradients to flow backwards through dashed arrows. The image was slightly adjusted from Ravi and Larochelle (2017).

Here, $\boldsymbol{w}_I$ and $\boldsymbol{b}_I$ correspond to the trainable weight matrix and bias vector for the learning rate. A similar parameterization is performed for the forget gate, i.e.,

$$\boldsymbol{p}_t = \sigma(\boldsymbol{w}_F \odot [\nabla_{\boldsymbol{\theta}_{t-1}} \mathcal{L}_{D^{tr}_{\mathcal{T}_t}}(f_{\boldsymbol{\theta}_{t-1}}), \mathcal{L}_{D^{tr}_{\mathcal{T}_t}}(f_{\boldsymbol{\theta}_{t-1}}), \boldsymbol{\theta}_{t-1}, \boldsymbol{p}_{t-1}] + \boldsymbol{b}_F), \tag{10}$$

where $\boldsymbol{w}_F$ and $\boldsymbol{b}_F$ are a weight matrix and bias vector, respectively.

Pseudocode for the LSTM meta-learner is displayed in Algorithm 3. After randomly initializing the parameters $\boldsymbol{\phi}$ of the LSTM $g_{\boldsymbol{\phi}}$ (line 1), several training iterations are performed (lines 3–16). In every episode, a task $\mathcal{T}_j$ is sampled (line 4), and the base-learner parameters $\boldsymbol{\theta}_0$ are initialized to the cell state $\boldsymbol{c}_0$ (line 5).

Then, the LSTM proposes T updates to the base-learner parameters. For every update, a batch of training examples $D^{tr}_{\mathcal{T}_j}$ is sampled (line 7), on which the loss of the base-learner is computed (line 8). The gradients of this loss are used to update the cell state (line 9), and thus the base-learner parameters (line 10; due to the one-to-one correspondence between the base-learner parameters and cell state).

At the end of an optimization trajectory of size T, the loss of the final parameters $\boldsymbol{\theta}_T$ is evaluated on the test set $D^{te}_{\mathcal{T}_j}$ (line 12). The gradients of this loss with respect to the LSTM parameters are used to update $\boldsymbol{\phi}_{e-1}$ (line 13).

Ravi and Larochelle (2017) follow Andrychowicz et al. (2016) by assuming that the base-learner loss gradients are independent of the LSTM parameters $\boldsymbol{\phi}$, to prevent the computation of higher-order derivatives. Furthermore, Ravi and Larochelle (2017) also preprocess the base-learner loss gradients using Equation 11. That is, given a gradient or loss x, it transform it into

$$p(x) \rightarrow \begin{cases} (\frac{\log|x|}{p}, \text{sign}(x)) & \text{if } |x| \geq e^{-p}, \\ (-1, e^p x) & \text{else} \end{cases}, \tag{11}$$

where p is the constant 10. Intuitively, this formula sqeezes values together, such that they are closer to each other. This is done to reduce the difference in magnitudes of losses and gradients, which can be a cause of training instability (Andrychowicz et al., 2016). On top of these technical tricks, the LSTM meta-learner use batch normalization to stabilize and speed up learning.

Algorithm 3 LSTM meta-learner, adapted from Ravi and Larochelle (2017)

```
 1: Randomly initialize LSTM parameters $\boldsymbol{\phi}_0$
 2: Initialize counter $e = 1$
 3: **repeat**
 4:     Sample $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})$
 5:     $\boldsymbol{\theta}_0 = \boldsymbol{c}_0$
 6:     **for** $t = 1, .., T$ **do**
 7:         Sample batch $\boldsymbol{X}_t, \boldsymbol{Y}_t$ from $D^{tr}_{\mathcal{T}_j}$
 8:         Compute loss of base-learner $\mathcal{L}_{(\boldsymbol{X}_t, \boldsymbol{Y}_t)}(f_{\boldsymbol{\theta}_{t-1}})$
 9:         Compute $\boldsymbol{c}_t$ using *Equation* 8
10:         $\boldsymbol{\theta}_t = \boldsymbol{c}_t$
11:     **end for**
12:     Compute test loss $\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}_T})$
13:     Compute $\boldsymbol{\phi}_e$ using $\nabla_{\boldsymbol{\phi}_{e-1}} \mathcal{L}_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}_T})$
14:     $\boldsymbol{c}_0 = \boldsymbol{c}_T$
15:     $e = e + 1$
16: **until** convergence
```

Note that while the LSTM meta-learner (Ravi and Larochelle, 2017) and MAML (Finn et al., 2017) are two well-known techniques in the field of meta-learning, they are no longer the best performing algorithms in few-shot learning settings. That is, many other meta-learning techniques have been proposed, such as latent embedding optimization (Rusu et al., 2018) and MetaOptNet (Lee et al., 2019), which outperform MAML and the LSTM meta-learner on few-shot image classification benchmarks. For a comprehensive overview of the current state-of-the-art in few-shot learning, we refer the reader to Lu et al. (2020). In contrast to the best performing techniques in these few-shot settings, MAML is also applicable to reinforcement learning settings, which is why it can still be considered a state-of-the-art technique.

3 TURTLE

In this section, we construct a small theoretical framework to capture the relationship between the LSTM meta-learner and MAML. Combining this theoretical knowledge with recent empirical findings, we construct a new meta-learning algorithm which we call *sTateless neURal meTa-LEarning*, or TURTLE. We prove that this technique is a generalized form of MAML, and thus has the theoretical capability of outperforming it.

3.1 Theoretical subsumption

There is a striking relationship between the LSTM meta-learner (Ravi and Larochelle, 2017) and MAML (Finn et al., 2017). Both seek to find a good set of initialization parameters $\boldsymbol{\theta}_0$ for the base-learner, which facilitates quick learning of new tasks. The two techniques differ, however, in their task-specific adaptation procedure. That is, MAML uses vanilla gradient descent updates, while the LSTM meta-learner uses a special LSTM module to update these initialization parameters (see Figure 1 in Section 1). Intuitively, one would say that because of this, the LSTM meta-learner is more expressive than MAML as the LSTM module could learn to perform gradient descent.

This leads us to formulate Theorem 1. Note that "technique X subsumes technique Y" means that technique X has enough expressive power to model the same solution as technique Y. The proof of this theorem is given below.

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

Proof. Assume that the LSTM meta-learner $g_{\boldsymbol{\theta}}$ and MAML start with the same base-learner initialization parameters $\boldsymbol{c}_0 = \boldsymbol{\theta}_0$, settings of the meta-optimizers (e.g., Adam), and optimization horizons, i.e., $T = T_{\text{LSTM}} = T_{\text{MAML}}$. Furthermore, assume that MAML uses a base-learning rate of α_{MAML}.

In order to prove that LSTM meta-learner subsumes MAML, we simply have to show that the theoretical possibility exists that the parameters of the LSTM meta-learner encode the same learning strategy as MAML (gradient descent).

More formally, we have to show that there exists a parameterization such that for an arbitrary task $\mathcal{T}_j \backsim p(\mathcal{T})$, the update sequence $[\boldsymbol{c}_0, \boldsymbol{c}_1, ..., \boldsymbol{c}_T]$ produced by the LSTM meta-learner is equivalent to the sequence $[\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_T]$ produced by MAML.

By assumption, we have that $\boldsymbol{c}_0 = \boldsymbol{\theta}_0$. Thus, it remains to be shown that there exists a *fixed* set of LSTM meta-learner parameters, such that at every time step $t \in \{0, ..., T-1\}$, $\boldsymbol{c}_{t+1} = \boldsymbol{\theta}_{t+1}$ holds, or equivalently,

$$\boldsymbol{p}_t \odot \boldsymbol{c}_t + \boldsymbol{i}_t \odot \bar{\boldsymbol{c}}_t = \boldsymbol{\theta}_t - \alpha_{\text{MAML}} \nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}_t}),$$

due to Equations 7 and 8. As discussed in Section 2.3.3, this equation is satisfied when $\boldsymbol{p}_t = \boldsymbol{1}$, $\boldsymbol{c}_t = \boldsymbol{\theta}_t$, $\boldsymbol{i}_t = \boldsymbol{\alpha}_{\text{MAML}}$, and $\bar{\boldsymbol{c}}_t = -\nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}_t})$. The requirements that $\boldsymbol{c}_0 = \boldsymbol{\theta}_0$ and $\bar{\boldsymbol{c}}_t = -\nabla_{\boldsymbol{\theta}_t} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}_t})$ can be satisfied by design. However, we still have to show that the remaining requirements can indeed be satisfied with the chosen parametric forms of $\boldsymbol{p}_t$ and $\boldsymbol{i}_t$.

Thus, we have to show it is possible for a meta-learner that $\boldsymbol{p}_t = \boldsymbol{1} = [1 ... 1]$, and $\boldsymbol{i}_t = \boldsymbol{\alpha}_{MAML} = [\alpha_{\text{MAML}} ... \alpha_{\text{MAML}}]$. We start with the former. From Equation 9, we know that

$$\boldsymbol{p}_t = \sigma(\boldsymbol{w}_F \odot [\nabla_{\boldsymbol{\theta}_{t-1}} \mathcal{L}_{D^{tr}_{\mathcal{T}_t}}(f_{\boldsymbol{\theta}_{t-1}}), \mathcal{L}_{D^{tr}_{\mathcal{T}_t}}(f_{\boldsymbol{\theta}_{t-1}}), \boldsymbol{\theta}_{t-1}, \boldsymbol{p}_{t-1}] + \boldsymbol{b}_F).$$

If we set $\boldsymbol{w}_F = [0 ... 0]$, we get

$$\begin{aligned} \boldsymbol{p}_t &= \sigma(\boldsymbol{b}_F) \\ &= \sigma([b_F^{(1)} ... b_F^{(n)}]) \\ &= \left(\left[\frac{1}{1 + e^{-b_F^{(1)}}} ... \frac{1}{1 + e^{-b_F^{(n)}}} \right] \right), \end{aligned} \tag{12}$$

where n is the number of base-learner parameters. From here, it is easy to see that $\boldsymbol{p}_t = [1 ... 1]$ if all $b_F^{(i)}$ are chosen to be sufficiently large, which yields $e^{-b_F^{(i)}} \approx 0$.

Finally, we have to show that it is possible to have $\boldsymbol{i}_t = [\alpha_{\text{MAML}} ... \alpha_{\text{MAML}}]$, or equivalently,

$$\sigma(\boldsymbol{w}_I \odot [\nabla_{\boldsymbol{\theta}_{t-1}} \mathcal{L}_{D^{tr}_{\mathcal{T}_t}}(f_{\boldsymbol{\theta}_{t-1}}), \mathcal{L}_{D^{tr}_{\mathcal{T}_t}}(f_{\boldsymbol{\theta}_{t-1}}), \boldsymbol{\theta}_{t-1}, \boldsymbol{i}_{t-1}] + \boldsymbol{b}_I) = [\alpha_{\text{MAML}} ... \alpha_{\text{MAML}}],$$

due to Equation 9. Again, if we pick the weight vector $\boldsymbol{w}_I = [0 \dots 0]$, we get

$$\begin{aligned} & \sigma(\boldsymbol{b}_I) = [\alpha_{\text{MAML}} \dots \alpha_{\text{MAML}}] \\ \equiv\ & \sigma([b_I^{(1)} \dots b_I^{(n)}]) = [\alpha_{\text{MAML}} \dots \alpha_{\text{MAML}}] \\ \equiv\ & \left(\left[\frac{1}{1+e^{-b_I^{(1)}}} \dots \frac{1}{1+e^{-b_I^{(n)}}} \right] \right) = [\alpha_{\text{MAML}} \dots \alpha_{\text{MAML}}]. \end{aligned} \tag{13}$$

We can then solve for

$$\begin{aligned} \alpha_{\text{MAML}} &= \frac{1}{1+e^{-b_I^{(i)}}} \\ \Rightarrow\ \alpha_{\text{MAML}} \left(1+e^{-b_I^{(i)}}\right) &= 1 \\ \Rightarrow\ e^{-b_I^{(i)}} &= \frac{1-\alpha_{\text{MAML}}}{\alpha_{\text{MAML}}} \\ \Rightarrow\ b_I^{(i)} &= -\ln\left(\frac{1-\alpha_{\text{MAML}}}{\alpha_{\text{MAML}}}\right). \end{aligned}$$

Since $0 < \alpha_{\text{MAML}} < 1$, it is thus indeed possible to have a parameterization of the LSTM meta-learner which yields exactly the same task-specific learning behavior as MAML. Hence, the proof is complete. □

3.2 Observations and empirical findings

This existence of a parameterization of the LSTM meta-learner that yields the same behavior as MAML means that in theory, the LSTM meta-learner could perform at least as well as MAML, or even better, as the learned update rule may be better than regular gradient descent (Andrychowicz et al., 2016). However, as mentioned in Section 2, neural networks, and thus the LSTM meta-learner, are trained by iterative procedures that navigate the error landscape. An example error landscape is displayed in Figure 6. If this landscape contains many local minima, or the input information is insufficient to compute the direction of the steepest descent, it may be possible that the meta-optimizer is unable to bring the LSTM meta-learner parameters towards the point where the behavior and thus performance is the same as that of MAML.

In fact, the findings of Finn et al. (2017) seem to indicate this. More specifically, they have found that MAML outperforms the LSTM meta-learner on 5-way, 1- and 5-shot image classification. This implies that the LSTM meta-learner was unable to arrive at the same point as MAML in its error landscape.

As we have seen before, the only difference between MAML and the LSTM meta-learner is that the latter learns an LSTM optimization network that can make updates to the base-learner weights. Consequently, the inability of the LSTM meta-learner to successfully navigate the error landscape must come from the additional complexity induced by learning an optimization procedure at the meta-level.

One hypothesis is that this is caused by the fact that the LSTM meta-learner learns a stateful optimization procedure in the form of an LSTM module, which needs to learn how to maintain and interact with a state, which requires additional trainable parameters. As a result, it may be that the complexity of the error landscape increases.

Another hypothesis as to why the LSTM meta-learner fails to find a good solution in the error landscape has to do with the first-order assumption that the algorithm makes in order to reduce the running time. That is, it ignores second-order derivatives by assuming

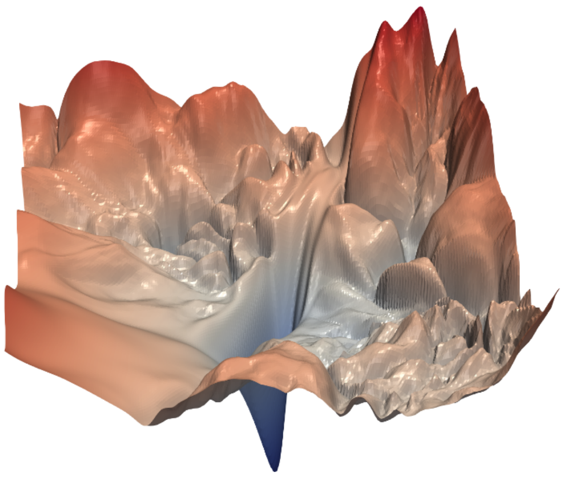


Figure 6: Example error landscape of a neural network. Lower points correspond to better solutions. Source: Li et al. (2018)

that inputs of the meta-network at time steps $t' > t$ do not depend on the meta-learner parameters of previous time steps. Ravi and Larochelle (2017) do not prove that this is a correct assumption to make and do not show any experiments that investigate the influence of this assumption on the performance of the LSTM meta-learner.

To understand why this assumption makes the meta-network unaware of the influence of its parameters on future inputs, one first has to develop an understanding of computation graphs that show the dependencies between variables over time. Figure 7 shows an example of such a graph for a meta-learning algorithm that learns the base-learner initialization parameters $\boldsymbol{\theta}$ and parameters $\boldsymbol{\phi}$ for the meta-network $g_{\boldsymbol{\phi}}$. This graph shows how the initial parameters are adjusted for $T = 2$ steps when presented with a new task $\mathcal{T}_j$. Black arrows from node a to node b indicate that node a influences node b. For example, we can see that the meta-network parameters $\boldsymbol{\phi}$ influence all updates $g_{\boldsymbol{\phi}}(\nabla_{\circ})$ made by this network, where $\circ = \nabla_{\boldsymbol{\theta}}, \nabla_{\boldsymbol{\theta}_j^{(1)}}$. Red arrows, on the other hand, display backward connections and indicate how gradients bubble up the graph in order to compute how the base-learner initialization $\boldsymbol{\theta}$ and meta-network parameters $\boldsymbol{\phi}$ should be adjusted.

Let us consider how the weights $\boldsymbol{\theta}_j^{(1)}$ are produced. We start at the root node of the base-learner, which corresponds to the weight initialization $\boldsymbol{\theta}$. The first step is then to compute the loss on the support set of a task, and the corresponding gradients $\nabla_{\boldsymbol{\theta}}$. Since these gradients depend on $\boldsymbol{\theta}$, there is a forward connection from this weight initialization $\boldsymbol{\theta}$ to the gradients $\nabla_{\boldsymbol{\theta}}$. These gradients $\nabla_{\boldsymbol{\theta}}$ can then be used as input to the meta-network $g_{\boldsymbol{\phi}}$ which, in turn, computes weight updates $g_{\boldsymbol{\phi}}(\nabla_{\boldsymbol{\theta}})$. This weight update is of course dependent on the inputs $\nabla_{\boldsymbol{\theta}}$ that the meta-network receives and the meta-network parameters $\boldsymbol{\phi}$, hence the forward connections between these nodes. Once we have computed the update step, we can simply add it to the previous parameters (in this case $\boldsymbol{\theta}$) and produce our first set of updated parameters $\boldsymbol{\theta}_j^{(1)}$. This same procedure is applied another time to compute the second set of updated parameters $\boldsymbol{\theta}_j^{(2)}$. After T updates are made (2 in this case), the loss on the query set is computed and propagated backwards through the red arrows in the graph to update the root nodes $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$.

The graph thus works in two phases: (i) the *forward* phase (black arrows) and (ii) the *backward* phase (red arrows). In the forward pass, the base-learner weights are adjusted on the support set of the given task by the meta-network in order to *learn* the task. At

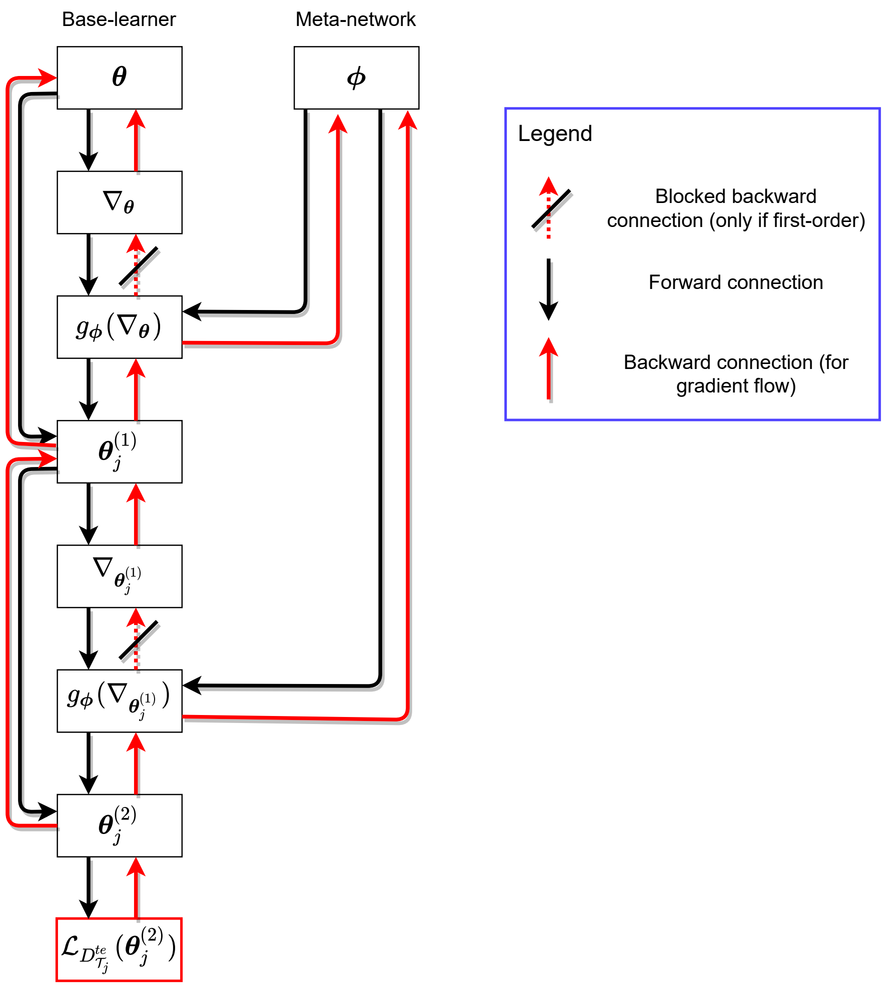

Base-learner
Meta-network
$\boldsymbol{\theta}$
$\boldsymbol{\phi}$
$\nabla_{\boldsymbol{\theta}}$
$g_{\boldsymbol{\phi}}(\nabla_{\boldsymbol{\theta}})$
$\boldsymbol{\theta}_j^{(1)}$
$\nabla_{\boldsymbol{\theta}_j^{(1)}}$
$g_{\boldsymbol{\phi}}(\nabla_{\boldsymbol{\theta}_j^{(1)}})$
$\boldsymbol{\theta}_j^{(2)}$
$\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(\boldsymbol{\theta}_j^{(2)})$
Legend
Blocked backward connection (only if first-order)
Forward connection
Backward connection (for gradient flow)

Figure 7: An example of a computation graph when $T = 2$ updates are made per task. In this example, the meta-network $g_{\boldsymbol{\phi}}$ updates the base-learner parameters based on their gradients with respect to the loss on the support set $\nabla_{\boldsymbol{\theta}_j^{(t)}} = \nabla_{\boldsymbol{\theta}_j^{(t)}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta}_j^{(t)})$. The initialization parameters $\boldsymbol{\theta}$ and meta-network parameters $\boldsymbol{\phi}$ are updated by propagating the loss on the query set (red node) backwards through the graph.

the end of this forward pass, we determine the loss of the resulting parameters on the query set of the task to assess how successful the learning process has been. Then, in the backward pass, gradients of this error signal are propagated backwards through the red arrows in the graph. In this way, we can compute the gradient of the query set loss with respect to our base-learner initialization and meta-network parameters, which can then be used to update them to increase the learning ability of our algorithm.

As shown in the graph, the first-order assumption blocks gradients from flowing through the red arrows with black cuts. If we look closely at the figure, we see that these blocked arrows always enter gradient nodes, meaning that gradients do not flow through gradients. Mathematically speaking, this means that we assume that $\nabla_{\boldsymbol{\theta}} \boldsymbol{\theta}_j^{(t)} = I$ and $\nabla_{\boldsymbol{\phi}} \nabla_{\boldsymbol{\theta}_j^{(t)}} = I$ for all values of t. As a consequence, approximated gradient expressions can be different from the full (second-order) expressions. This means that the base-learner initialization and meta-network parameters are adjusted with incomplete gradient signals.

While Finn et al. (2017) have shown that first-order MAML performs just as well as second-order MAML, this does not mean that first-order approximations are not hurtful when also learning the *parameters of a meta-network* as done by the LSTM meta-learner. In fact, we think that first-order approximations *are* hurtful when learning an optimizer. The reason for this is that the first-order assumption disconnects the backward graph (see the blocked red arrows in Figure 7), meaning that the optimizer will become unaware that updates at time step t affect gradient inputs at time steps $t' > t$. In short, the learned optimizer is navigating the meta-landscape with incomplete information.

3.3 Design of TURTLE

These theoretical and empirical insights motivate the construction of a new meta-learning algorithm (TURTLE) that, in theory, subsumes MAML, but makes fewer assumptions and has a less complex design such that the error landscape may be easier to navigate.

Learning objective and gradients The idea is simple. That is, we combine the LSTM meta-learner and MAML but replace the LSTM module with a *stateless* feed-forward neural network, which also has the task of optimizing the weights of the base-learner network. We call this technique sTateless neURal meTa LEarning, or TURTLE.

Figure 1 in Section 1 visually depicts the architectural differences between MAML, the LSTM meta-learner, and TURTLE. As we can see, all techniques learn initialization weights (top box). The differences become evident, however, when looking at how they learn a task by producing task-specific weights or, equivalently, fast weights. That is, MAML uses regular gradient descent to learn the new task. The LSTM meta-learner, on the other hand, replaces the gradient descent procedure by a trainable and stateful LSTM module. Lastly, TURTLE uses a simple feed-forward neural network (FFNN) as trainable optimizer, which does not maintain a state. Besides these architectural differences, there are also algorithmic differences between the three techniques, which will become clear further on in this subsection.

We now leverage the mathematical machinery that was used to develop MAML and the LSTM meta-learner in order to formalize TURTLE. Let $f_{\boldsymbol{\theta}}$ denote the base-learner neural network, and $g_{\boldsymbol{\phi}}$ our meta-feed-forward neural network, which takes input information I_j and outputs weight updates $g_{\boldsymbol{\phi}}(I_j)$ for the base-learner parameters. This input information can include, e.g., (processed) gradients of the base-learner parameters and the loss value. On a high level, our goal is to minimize the expected base-learner loss over a distribution of tasks $p(\mathcal{T})$. Suppose we are given initial base-learner parameters $\boldsymbol{\theta}$. Furthermore, assume that our meta-network $g_{\boldsymbol{\phi}}$ only makes a single update to these base-learner parameters $\boldsymbol{\theta}$. Then, we wish to find optimal parameters for our meta-network

$$\boldsymbol{\phi}^* = \arg\min_{\boldsymbol{\phi}} \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})} \left[\mathcal{L}_{D^{te}_{\mathcal{T}_j}} (\boldsymbol{\theta} + g_{\boldsymbol{\phi}}(I_j)) \right], \tag{14}$$

where I_j is meta-information specific to task $\mathcal{T}_j$, for example, a combination of the loss $\mathcal{L}_j = \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})$, and gradients $\nabla_j = \nabla_{\boldsymbol{\theta}} \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(\boldsymbol{\theta})$.

Note that this goal formulation assumes that some initial base-learner parameters $\boldsymbol{\theta}$ are given. Instead, we also include these in our learning objective, as a good initialization is a warm-start for quick learning, as shown by Finn et al. (2017). Consequently, we wish to obtain

$$\boldsymbol{\phi}^*, \boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\phi}, \boldsymbol{\theta}} \mathbb{E}_{\mathcal{T}_j \backsim p(\mathcal{T})} \left[\mathcal{L}_{D^{te}_{\mathcal{T}_j}} (\boldsymbol{\theta} + g_{\boldsymbol{\phi}}(I_j)) \right]. \tag{15}$$

Finding the optimal parameters $\boldsymbol{\phi}^*$ and $\boldsymbol{\theta}^*$ is infeasible. Hence, we resort to hand-crafted gradient-based optimizers, such as Adam. We now derive gradients of the loss function in Equation 15.

We start with the gradients with respect to base-learner parameters $\boldsymbol{\theta}$.

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j)) &= \mathcal{L}'_{D^{te}_{\mathcal{T}_j}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j))\nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j)) \\ &= \mathcal{L}'_{D^{te}_{\mathcal{T}_j}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j))\,(I+\nabla_{\boldsymbol{\theta}}g_{\boldsymbol{\phi}}(I_j)) \\ &= \mathcal{L}'_{D^{te}_{\mathcal{T}_j}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j))\,\left(I+g'_{\boldsymbol{\phi}}(I_j)\nabla_{\boldsymbol{\theta}}(I_j)\right) \end{aligned} \tag{16}$$

As one can see, this gradient expression follows the same pattern as in Equation 6.

For the meta-network parameters $\boldsymbol{\phi}$, the gradient is given by

$$\begin{aligned}\nabla_{\boldsymbol{\phi}}\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j)) &= \mathcal{L}'_{D^{te}_{\mathcal{T}_j}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j))\nabla_{\boldsymbol{\phi}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j)) \\ &= \mathcal{L}'_{D^{te}_{\mathcal{T}_j}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j))\,(\nabla_{\boldsymbol{\phi}}\boldsymbol{\theta}+\nabla_{\boldsymbol{\phi}}g_{\boldsymbol{\phi}}(I_j)) \\ &= \mathcal{L}'_{D^{te}_{\mathcal{T}_j}}(\boldsymbol{\theta}+g_{\boldsymbol{\phi}}(I_j))\,\left(g'_{\boldsymbol{\phi}}(I_j)\nabla_{\boldsymbol{\phi}}I_j\right). \end{aligned} \tag{17}$$

To compute these gradient expressions, we have to propagate backwards through the optimization trajectory—just like second-order MAML—which is computationally expensive as it requires the computation of second-order derivatives. Finn et al. (2017) found that these second-order can be discarded without sacrificing performance. The LSTM meta-learner (Ravi and Larochelle, 2017) also makes an assumption which allows it to side-step the computation of second-order derivatives. We do not build in such an assumption for TURTLE as it may be that second-order gradients are crucial for learning a good optimizer (see Section 3.2).

The algorithm The training algorithm for TURTLE is displayed in Algorithm 4. First, we initialize all involved parameters, namely the initialization point $\boldsymbol{\theta}$ for the base-learner, and weights $\boldsymbol{\phi}$ of the meta-learner network (lines 1–2). Then, for every task in the batch B, sampled in line 4, we make T updates to our base-learner initialization $\boldsymbol{\theta}$ (lines 7–11) to obtain fast (task-specific) weights $\boldsymbol{\theta}'_j$. To start this process, we initialize the fast weights to be equal to our initialization $\boldsymbol{\theta}$ (line 6). At every time step t, we compute meta-information with our current fast weights on the support set (line 8). This information is fed into the meta-learner network $g_{\boldsymbol{\phi}}$, which then proposes weight updates $\boldsymbol{u}_j$ (line 9), which are added to our fast weights (line 10).

With the final task-specific weights $\boldsymbol{\theta}'_j$, we compute the loss on the query set $D^{te}_{\mathcal{T}_j}$ (line 12), which is then propagated backwards to update both our initialization $\boldsymbol{\theta}$ and meta-learner network parameters $\boldsymbol{\phi}$ (line 14). Importantly, our meta-learner network works on a per-parameter basis, in similar fashion to the LSTM meta-learner. This means that the same meta-network is used to update every base-learner parameter.

Relationship with MAML We now show that TURTLE can be seen as a generalized form of MAML, i.e., TURTLE subsumes MAML. For this, we assume without loss of generality that TURTLE and MAML both start with the same base-learner initialization $\boldsymbol{\theta}$. Then, all we need to show is that the meta-network used by TURTLE can, in principle, perform regular gradient descent in similar fashion to MAML.

For this, assume task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})$ is presented to the models, leading to the observed loss $\mathcal{L}_j = \mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}})$ and corresponding gradients $\nabla_j = \nabla_{\boldsymbol{\theta}}\mathcal{L}_{D^{tr}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}})$. Now,

Algorithm 4 TURTLE

```
1: Randomly initialize base-learner weights $\boldsymbol{\theta}$
2: Randomly initialize meta-learner weights $\boldsymbol{\phi}$
3: repeat
4:     Sample batch of $m$ tasks $B = \{(D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})\}_{j=1}^{m}$
5:     for $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j}) \in B$ do
6:         Initialize fast weights $\boldsymbol{\theta}'_j = \boldsymbol{\theta}$
7:         for $t = 1, ..., T$ do
8:             Compute information $I_j$ by applying $f_{\boldsymbol{\theta}'_j}$ to the support set
9:             Compute one-step update $\boldsymbol{u}_j = g_{\boldsymbol{\phi}}(I_j)$
10:            Update fast weights $\boldsymbol{\theta}'_j = \boldsymbol{\theta}'_j + \boldsymbol{u}_j$
11:        end for
12:        Compute $\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j})$
13:    end for
14:    Update $\Theta = \{\boldsymbol{\theta}, \boldsymbol{\phi}\}$ by propagating $\sum_{\mathcal{T}_j \in B} \mathcal{L}_{D^{te}_{\mathcal{T}_j}}(f_{\boldsymbol{\theta}'_j})$ backwards
15: until convergence
```

MAML updates the base-learner parameters at a given time step using standard gradient descent: $\boldsymbol{\theta}_j = \boldsymbol{\theta} - \alpha\nabla_j$. In contrast, TURTLE updates the base-learner parameters using the meta-network $g_{\boldsymbol{\phi}}$, i.e., $\boldsymbol{\theta}'_j = \boldsymbol{\theta} + g_{\boldsymbol{\phi}}(I_j)$. When the update terms are equivalent, i.e., $-\alpha\nabla_j = g_{\boldsymbol{\phi}}[I_j]$, TURTLE learns a new task exactly like MAML.

In order to prove that TURTLE subsumes MAML, we must show that there exists a configuration of weights for meta-network $g_{\boldsymbol{\phi}}$ such that the proposed updates are equivalent to those made by gradient descent. Figure 8 contains example weights $\boldsymbol{\phi}$ for a fairly simple meta-network architecture that results in updates equivalent to those made by gradient descent. That is, given the gradient ∇_j of a base-learner parameter and loss value $\mathcal{L}_j$, the meta-network outputs the proposed update $-\alpha\nabla_j$, which is equivalent to the update made by gradient descent.

The same result holds for any other meta-network architecture, assuming that the gradients ∇_j are provided as input and that the meta-network uses linear or ReLU activation functions. To see this, one can imagine a meta-network in which all weights are set to zero except for a single path from the gradient input to the output node. Weights along this path to the final hidden layer would have a value of one, and the weight from the hidden layer to the output node a value of $-\alpha$. We can thus conclude that TURTLE is indeed a generalized form of MAML.

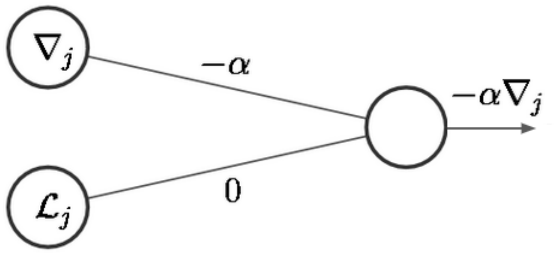


Figure 8: Example network configuration for $g_{\boldsymbol{\phi}}$ that proposes updates equivalent to regular gradient descent.

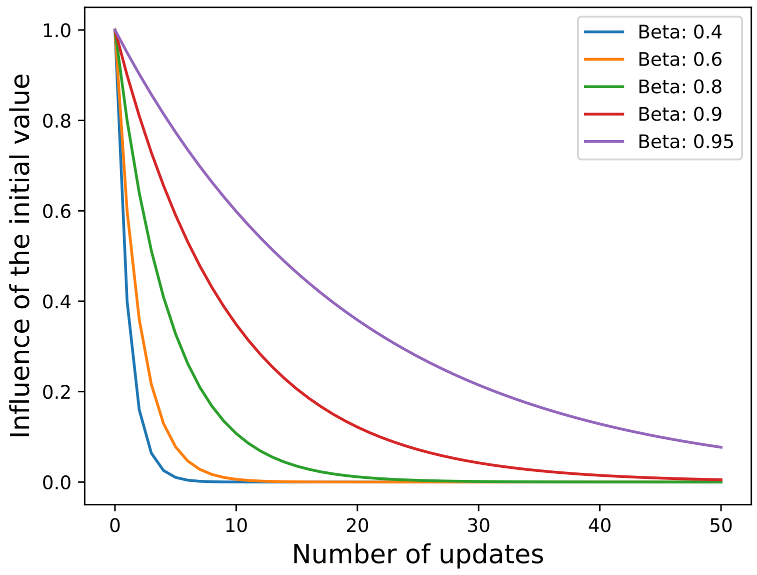


Figure 9: The influence of the β parameter on the decay of previous information.

Meta-information In the above reasoning, we assumed that the meta-information I_j was given by the loss $\mathcal{L}_j$ and the gradients ∇_j. However, besides the loss and gradients, we think that it may be useful to add the gradient, or update history information, which is also done in hand-crafted optimization procedures, such as momentum (Rumelhart et al., 1986). For this, we maintain a running average of the previous updates or gradients for every parameter $\theta^{(i)}$ in the base-learner network. By placing this state in the inputs of meta-network, we mitigate the absence of an explicit state in the meta-learner network. When presented with a new task $\mathcal{T}_j$, we initialize empty history buffers, which is maintained over the optimization trajectory of length T. Note that when only a single update is made per task, i.e., $T = 1$, there is no point in keeping track of a history.

The historical information can be integrated in TURTLE by concatenating an additional column to the input matrix I_t at every time step. This column $\boldsymbol{h}_t \in \mathbb{R}^{n\times 1}$ represents the historical information for every of the n base-learner parameters. Suppose we use previous updates as historical information. Then, we can maintain an exponentially moving average given by

$$\boldsymbol{h}_{t+1} = \begin{cases} \mathbf{0} & \text{if } t = 0, \\ \beta \boldsymbol{h}_t + (1-\beta) g_{\boldsymbol{\phi}}(I_t) & \text{else} \end{cases}, \tag{18}$$

where $\beta \in [0, 1]$ is a parameter that influences the how long previous inputs will influence the updates. The influence of this parameter is demonstrated in Figure 9. As one can see, previous values exert more influence on the current value when beta is larger. In other words, a larger beta value slows down the decay of previous information. When using previous gradients instead of updates, one can simply replace the terms $g_{\boldsymbol{\phi}}(I_t)$ in the equation above by the gradients with respect to base-learner parameters at time step t. Also, one would not have to first initialize the history with a vector of zeros. Instead, one could initialize the history buffer with the first observed gradients on the support set.

4 Experiments

In this section, we empirically investigate the discussed techniques. First, we describe our experimental setup to test the algorithms. Second, we present the obtained results by following this setup.

4.1 Experimental setup

Our experimental setup consists of a regression problem and image classification problems. In the former case, the goal is to learn a real-valued function $f_{\boldsymbol{\theta}} : X \rightarrow \mathbb{R}$, while in the latter case, the goal is to learn a classifier $f_{\boldsymbol{\theta}} : X \rightarrow C$, where C is a set of classes. An overview of the techniques that we will experiment with in both settings are shown in Table 2. Note that the centroid-based finetuning model can only be applied in classification settings due to its design. Next, we discuss the two problem setups in more detail.

Technique	Key idea
TrainFromScratch (TFS)	Learn every task from a random initialization $\boldsymbol{\theta}$
Finetuning (FT)	Re-use the pre-trained embedding module and only learn a new output layer for every new task
Centroid-based finetuning (CFT)	Re-use the pre-trained embedding module and learn a special new output layer for every task based on cosine similarity
MAML	Learn good initialization parameters from which new tasks can be learned quickly using gradient descent
LSTM meta-learner (LSTM)	Learn good initialization parameters and an LSTM module that updates the base-learner weights
TURTLE	Learn good initialization parameters and a feed-forward neural network that updates the base-learner weights

Table 2: Brief summary of all techniques that we will experiment with.

4.1.1 Sine wave regression

Finn et al. (2017) proposed a simple sine regression problem, where the goal is to quickly learn a sine function that has generated the few examples in the support set that we are given access to.

Task setup Every task $\mathcal{T}_j$ is associated with a different sine wave function $s_j(x) = a \cdot \sin(x - p)$, where $a \in [0.1, 5.0]$ is the amplitude, and $p \in [0, \pi]$ the phase, which are both selected uniformly at random from their corresponding ranges. Then, given a task $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})$, the goal is to infer the sine wave that gives rise to observations (x_i, y_i) in the support set of a task. Note that the model receives no additional information about the shape or characteristics of the sine function other than these training examples. The quality of the inference is then determined based on the prediction error (MSE) on the observations from the query set $D^{te}_{\mathcal{T}_j}$.

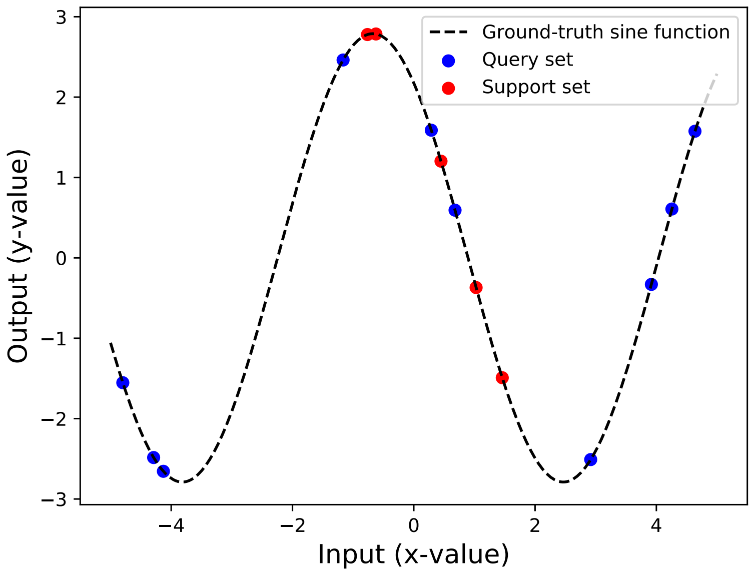


Figure 10: Example of a sine wave regression task.

An example of a task is given in Figure 10. In this case, the model has to infer the correct sine wave using $k = 5$ examples (shown in blue) from the support set. The degree to which the model succeeded, is evaluated on the 10 (red) examples in the query set.

This problem lends itself nicely for studying the behavior of meta-learning systems, and allows for a good comparison to transfer learning models. That is, the nature of the problem makes it very hard to succeed by training on flat batches of data which are sampled from all meta-training data $D^{tr} = (D^{tr}_{\mathcal{T}_1} \cup D^{te}_{\mathcal{T}_1}) \cup ... \cup (D^{tr}_{\mathcal{T}_J}, D^{te}_{\mathcal{T}_J})$. Transfer learning techniques, which train in this way, are thus unlikely to succeed (Finn et al., 2017) as the training examples do not come from the same sine function and, consequently, may contradict each other.

Base-learner In this problem setup, we have a base-learner neural network $f_{\boldsymbol{\theta}}$—with parameters $\boldsymbol{\theta}$—that attempts to learn the sine functions from few observations in $D^{tr}_{\mathcal{T}_j}$. We use the same network as used by Finn et al. (2017): a fully-connected feed-forward neural network of three layers. The input to the network is a single real-valued number $x_i \in [-5.0, 5.0]$. The first two hidden layers each have 40 nodes, which are followed by ReLU nonlinearities. The output layer maps the hidden state to a single real-valued output $f_{\boldsymbol{\theta}}(x_i)$.

Training details We generate 70 000 meta-training tasks and use the few-shot learning setup. Each task consists of a support set $D^{tr}_{\mathcal{T}_j}$ which contains k examples, and query set of size 50. In order to test well which system can learn the quickest from few data points, we choose k to be small. The number of data points in the query set is larger to ensure proper evaluation of generalization performance.

We train both the transfer learning baselines and meta-learning techniques for 70K episodes. The baselines do, however, not train on tasks, but on randomly sampled batches of data from all meta-training data $D^{tr} = (D^{tr}_{\mathcal{T}_1} \cup D^{te}_{\mathcal{T}_1}) \cup ... \cup (D^{tr}_{\mathcal{T}_{70\,000}}, D^{te}_{\mathcal{T}_{70\,000}})$. In accordance with Chen et al. (2019), the size of these flat batches is set to be 16. When the baselines are exposed to tasks during meta-validation or meta-test time, they train on mini-batches of size 4 sampled from the support set. Meta-learning algorithms, in contrast, train on every task in sequential fashion. They do not sample mini-batches from the support and query sets, and thus these sets are presented to the meta-learning models as a whole.

We validate the performance after every 2 500 episodes on a fixed set of 1 000 validation tasks. Note that this was not done by Finn et al. (2017) as they argue that meta-validation is unnecessary. We agree with this, as the models see every task only once which means that there is no risk of overfitting. However, meta-validation does allow us to perform hyperparameter tuning of TURTLE.

Evaluation details After a single run of meta-training, the final performance of the models is measured on a fixed set of 2 000 sine wave tasks. For this, we compute the mean squared error (MSE) between the models' predictions and the ground-truth outputs on the query sets of the meta-test tasks (after training on the support set). As a result, we get 2 000 MSE scores. We take the mean and median of these scores as final performance estimates of a model in a given run.

We perform 30 runs of every model with different initializations to investigate their robustness. Thus, every model performs is trained and evaluated 30 times on the same training and test data. We summarize the 30 evaluation scores using the mean, median, and 95% confidence intervals. Importantly, we ignore outliers as they can have a large impact on the mean and size of the confidence intervals. An observation x is said to be an outlier iff $x \notin [Q_1 - 1.5 \cdot (Q_3 - Q_1), Q_3 + 1.5 \cdot (Q_3 - Q_1)]$, where Q_1 and Q_3 denote the first and third quartiles of all the performance estimates across runs, respectively.

4.1.2 Image classification

For image classification, we re-implement a challenging setup proposed by Chen et al. (2019), which also allows us to evaluate the performance of the algorithms when *task distribution shifts* occur. This in in contrast to frequently used benchmarks which test the performance of algorithms on tasks from the same data set that was used for training.

Task setup The challenging setting of Chen et al. (2019) that we use, consists of two data sets:

- **miniImageNet**: As its name suggests, miniImageNet is a smaller version of the large ILSVRC data set (Russakovsky et al., 2015), originally proposed by Vinyals et al. (2016) to reduce the required computational costs and engineering efforts to run experiments with this data. The miniImageNet data set contains 100 classes and 60 000 RGB images. The examples are divided uniformly across the classes, which means that every class is accorded by 600 examples.

- **CUB**: The CUB data set (Wah et al., 2011) consists of roughly 12 000 colored bird images from 200 species (classes).

Both data sets are split into meta-training, meta-validation, and meta-test partitions. For miniImageNet, we use the splits proposed by Ravi and Larochelle (2017). For CUB, we randomly create our own splits with a 70/15/15 class ratio. All images are resized to 84×84 pixels.

A single task is then constructed following the N-way k-shot classification setup as described in Section 2.3.1. An example task is shown in Figure 2, where $N = 5$ and $k = 1$.

In contrast to this example, our query sets will contain 16 examples for each of the N classes, regardless of the value of k or N, following Chen et al. (2019).

Base-learner We use a convolutional neural network (CNN) consisting of four blocks as our base-learner, following Chen et al. (2019) and Snell et al. (2017). Every block consists of 64 convolutions of size $(3, 3)$, followed by a batch normalization layer, ReLU activation nonlinearity, and a 2D max-pooling layer with a kernel size of 2. The extracted features are then flattened and fed into a dense layer, with the exception of the centroid-based finetuning model, which uses its own special prototype output layer.

Training details All models are trained using the meta-training data from a single data set: either miniImageNet or CUB. Transfer learning baselines have an output layer with a size corresponding to the number of classes in the meta-training data, and are trained on flat batches of size 16 from $D^{tr} = (D^{tr}_{\mathcal{T}_1} \cup D^{te}_{\mathcal{T}_1}) \cup ... \cup (D^{tr}_{\mathcal{T}_J}, D^{te}_{\mathcal{T}_J)}$. Here J is the number of meta-training tasks. For $k = 1$ (1-shot setting), we randomly sample $J = 60\,000$ tasks to construct the meta-training set. For $k = 5$, we sample $J = 40\,000$ meta-training tasks.

In similar fashion to the sine regression case, we will use the hyperparameters reported by the original authors for all algorithms, unless explicitly stated otherwise. Additionally, we validate the performance of meta-learning algorithms every 2 500 episodes on 600 meta-validation tasks. The parameters that gave rise to the best validation performance will be evaluated on the meta-test tasks.

Evaluation details We evaluate the trained models on (i) the meta-validation and meta-test tasks of the same data set that was used for training, and (ii) on the meta-test tasks of the data set that was not used for training. Evaluation method (i) allows us to evaluate how well the models work within the same task distribution, whereas method (ii) also shows how well these models perform when a task distribution shift occurs.

Both meta-validation and meta-testing are performed using 600 randomly sampled tasks from the meta-validation and meta-test data respectively, following Chen et al. (2019). After a single run—consisting of meta-training, meta-validation, and meta-testing—we thus obtain 600 meta-test accuracy scores. We take the mean of these scores as representative performance. The mean, median, and 95% confidence intervals are then computed over a total of 5 runs as final performance estimate. For evaluation on the data set that was not used for training, we randomly sample 600 tasks from the meta-test tasks.

Table 3 contains a summary of our experimental setup.

4.2 Experimental results

In this subsection, we report on the experiments that we have conducted and the corresponding results.

4.2.1 Architecture of TURTLE

First, we study the behavior of 1-step TURTLE with different meta-learner architectures. More specifically, we perform an exhaustive grid search over three different activation functions (ReLU, sigmoid, tanh), three network architectures (0, 1, or 2 hidden layers with 20 nodes per layer), and three input types: raw gradients, raw gradients and loss, and

	Regression	Image classification
Training data	Sine waves	miniImageNet, CUB
Validation data	Sine waves	miniImageNet, CUB
Test data	Sine waves	miniImageNet, CUB
Meta-training tasks	70K	40K/60K for $k = 1/k = 5$
Meta-validation tasks	1K	600
Meta-test tasks	2K	600
Validate after	2.5K tasks	2.5K tasks
Base-learner network	Fully-connected feed-forward neural network with two hidden layers of 40 ReLU nodes followed by an output layer with a single node	Four stacked convolutional blocks (2D convolutions of 64 filters with kernel size 3, batch normalization, ReLU activation, 2D max-pooling) followed by a dense layer with N (number of classes) output nodes

Table 3: Summary of our experimental setup.

processed gradients and loss. We measure the performance of these models on validation tasks of the 5-shot sine wave regression problem because of its challenging nature: there are very few examples per task, which makes meta-learning important for achieving good performance.

The results are displayed in Table 4. Note that the confidence intervals are the same for the mean and median scores due to the fact that these statistics are computed over the maximum validation scores per run, which do not change, regardless of the statistic that is being computed.

Looking at this table, we see firstly that the raw gradient input type gives rise to good models compared with the other input types. This indicates that neither adding loss information nor processing gradients is helpful for achieving good performance. Secondly, we see that there seems to be a monotonically decreasing relationship between the MSE loss and the number of layers in the raw gradient meta-learners. This implies that we should investigate raw gradient meta-learner networks with more than two hidden layers.

Overall, we see that TURTLE with raw gradient information, two hidden layers, and the ReLU activation function performs best. For this reason, TURTLE will use the ReLU function in further experiments.

We also tested whether processed gradients without loss information could improve the performance of our best performing TURTLE so far. For this, we only include TURTLE models with the ReLU function, as it is clear that it outperforms the sigmoid and tanh functions. Table 5 shows the results of these experiments. As we can see, processing the gradients does not have a positive effect on the performance. Therefore, we omit all gradient preprocessing in the TURTLE models.

4.2.2 Layers and order of TURTLE

Here, we further investigate the influence of the number of layers on the performance of TURTLE, for different number of inner-optimization steps T. Moreover, we compare the performance of first-order TURTLE (fo-TURTLE), which ignores second-order gradients,

Input	HL	Activation	Mean MSE	Median MSE
Raw gradients	0	NA	1.04 ± 0.32	0.75 ± 0.32
	1	ReLU	0.67 ± 0.02	0.65 ± 0.02
	1	Sigmoid	0.83 ± 0.22	0.69 ± 0.22
	1	Tanh	0.71 ± 0.03	0.68 ± 0.03
	2	ReLU	**0.63** ± 0.02	**0.62** ± 0.02
	2	Sigmoid	0.69 ± 0.05	0.64 ± 0.05
	2	Tanh	0.65 ± 0.01	0.64 ± 0.01
Raw gradients and loss	0	NA	E ± E	2.74 ± E
	1	ReLU	2.90 ± 0.57	4.12 ± 0.57
	1	Sigmoid	1.43 ± 0.46	0.66 ± 0.46
	1	Tanh	0.93 ± 0.25	0.66 ± 0.25
	2	ReLU	1.48 ± 0.47	0.63 ± 0.47
	2	Sigmoid	1.12 ± 0.37	0.65 ± 0.37
	2	Tanh	1.37 ± 0.47	0.66 ± 0.47
Processed gradients and loss	0	NA	2.95 ± 0.27	2.85 ± 0.27
	1	ReLU	1.80 ± 0.31	1.43 ± 0.31
	1	Sigmoid	2.75 ± 0.32	2.53 ± 0.32
	1	Tanh	2.95 ± 0.41	2.86 ± 0.41
	2	ReLU	1.14 ± 0.20	0.97 ± 0.20
	2	Sigmoid	2.01 ± 0.41	1.35 ± 0.41
	2	Tanh	2.33 ± 0.45	2.43 ± 0.45

Table 4: MSE scores of various second-order TURTLE models on the validation tasks of the 5-shot sine wave regression problem. Column HL indicates the number of hidden layers of the meta-network used by TURTLE. The ± indicates the 95% confidence interval over 30 runs. "E" means that the value exceeds 100. The best mean and median performances are displayed in bold font.

and second-order TURTLE (so-TURTLE), which does not. In addition, we compare the performance with that of first-order MAML (fo-MAML; which ignores second-order gradients) and second-order MAML (so-MAML).

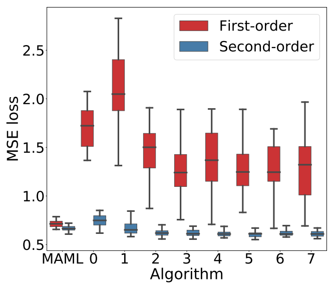
The results for optimization trajectories of size $T = 1, 5$, and 10 are shown in Figure 11. As we can see, fo-MAML is a good approximation to so-MAML for all tested trajectory sizes, as the difference between their performances is very small, which was also found by Finn et al. (2017). In contrast, the difference between fo- and so-TURTLE is much larger. This is a clear indication that second-order information is crucial for successful navigation of the meta-landscape.

For $T = 10$, we find something surprising. That is, both fo- and so-TURTLE seem very sensitive to the initialization point, as the variance is very large. This indicates that the meta-landscape has grown more complex, with higher local minima than for smaller values of T. This could be explained by the fact that TURTLE makes more updates on a single task, whilst neglecting information from other tasks. Since single tasks may be noisy due to their small sizes ($k = 5$), it is clear that this may guide the meta-network parameters in the wrong direction.

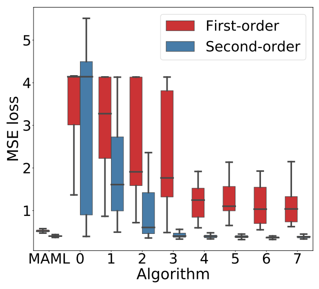
Overall, we see that TURTLE with five or six hidden layers seems to yield the most robust performance across the various optimization trajectory sizes. Since the difference

Type	HL	Mean MSE	Median MSE
Raw gradients	2	**0.63** ± 0.02	**0.62** ± 0.02
Processed gradients	0	2.89 ± 0.08	2.91 ± 0.08
	1	1.39 ± 0.28	1.39 ± 0.28
	2	0.90 ± 0.00	0.90 ± 0.00

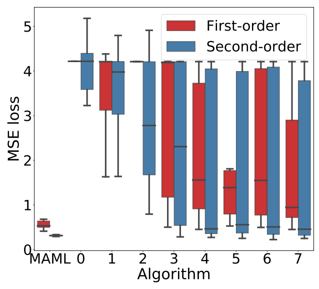
Table 5: Meta-validation performance of TURTLE with processed gradients as input. The best mean and median performances are displayed in bold font.



(a) $T = 1$



(b) $T = 5$



(c) $T = 10$

Figure 11: Influence of the gradient order and number of layers on the meta-validation performance on 5-shot sine wave regression. Integers on the x-axis denote the number of hidden layers in TURTLE's meta-network.

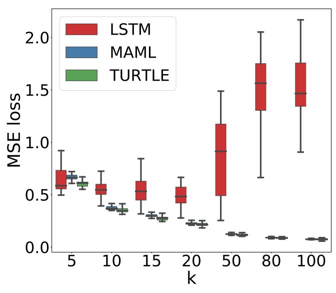
between the number of hidden layers is small, we pick five for future experiments. With this new setup for TURTLE, we experimented again with various input types to see which one yields the best performance, but again we found that raw gradients gave rise to the most robust performance (see Figure 19 in the appendix). For this reason, future experiments will be conducted with this input type.

4.2.3 Influence of task size

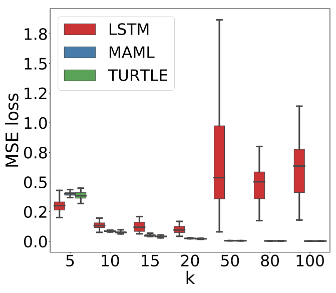
Next, we investigate whether the observed instability of TURTLE is caused by noise in update directions due to the small sizes of tasks ($k = 5$) examples. Therefore, we investigate how the performances of so-MAML and so-TURTLE are affected by increasing k: the number of examples in the support sets of tasks. We also compare them with the LSTM meta-learner.

The results are depicted in Figure 12. Firstly, we note that the LSTM meta-learner, while sometimes unstable, is able to achieve the best performance for $k = 5$. For larger values of k, however, it performs poorly and becomes highly unstable. We do not know what causes this, but it may indicate that the chosen architecture is not optimal for sine wave regression. In contrast, TURTLE and MAML grow more stable as k increases, which could be explained by the fact that task-level noise decreases.

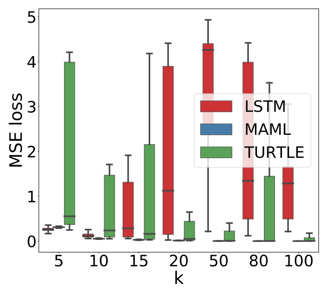
Secondly, we note that the LSTM meta-learner outperforms TURTLE when making only a single update step while it cannot leverage its memory component. This could be explained by the fact that the LSTM meta-learner learns a learning rate for every base-learner parameter. Below, we also investigate whether TURTLE benefits from learning a learning rate per parameter.



(a) $T = 1$



(b) $T = 5$



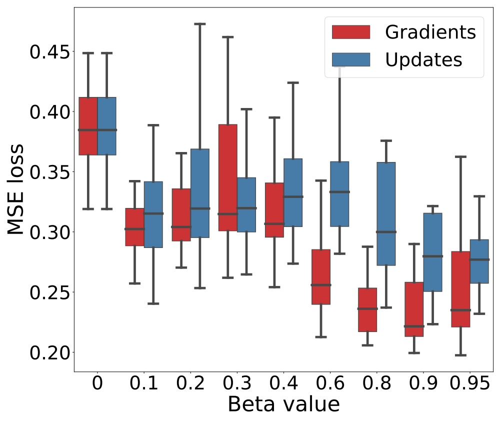
(c) $T = 10$

Figure 12: Influence of the number of data points per support set (k) on the meta-validation performance of so-MAML, the LSTM meta-learner, and so-TURTLE on 5-shot sine wave regression with various values of T.

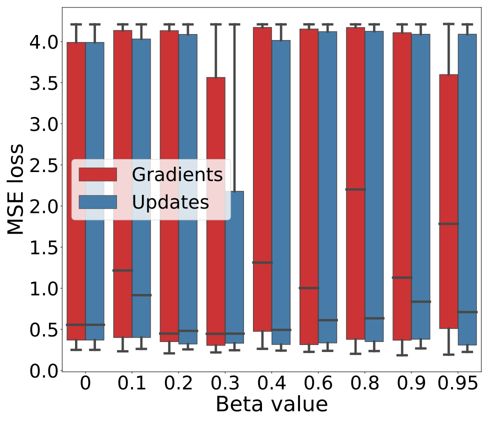
Overall, we observe that the meta-learning process becomes less stable as T increases, for both TURTLE and the LSTM meta-learner, although TURTLE is more robust for $T = 5$. While increasing the task size k improves TURTLE's training stability, we think that the gain is too small to attribute to the task size. For this reason, we will aim to improve the stability of TURTLE by making architectural changes.

4.2.4 Stabilizing TURTLE

Inclusion of historical information A potential cause of instability is the fact that regular TURTLE does not take previous updates or gradients into account, while this could provide useful information, for without such information, the meta-network can abruptly shift directions in which it moves the base-learner parameters. Including historical information could prevent this from happening, as the update of a base-learner parameter at time step t would be influenced by previous gradients or updates.



(a) $T = 5$



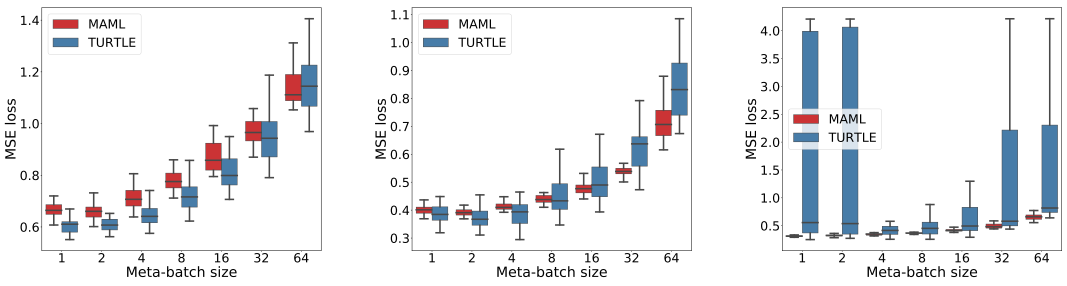
(b) $T = 10$

Figure 13: Influence of historical information on the meta-validation performance of 5-HL so-TURTLE on 5-shot sine wave regression.

Figure 13 displays the results from including historical information as input (see Section 3.3). For 5-step TURTLE, we can clearly see that the inclusion of historical information is beneficial. Overall, we see that gradient history information is more helpful than previous updates. Moreover, larger beta values give rise to better performance, indicating that gradients from the distant past can be helpful for deciding the next update.

For $T = 10$, the results are not so clear. Nonetheless, the inclusion of historical information also has a slight beneficial effect here, albeit for small values of beta.

Aggregating gradients Another cause of instability could be the fact that TURTLE has so far only been tested with meta-batch sizes of 1. This means that it updates its meta-network parameters after every task it encounters. Therefore, TURTLE suffers from tunnel-vision which may be detrimental to stability, especially when T becomes larger.



(a) $T = 1$ (b) $T = 5$ (c) $T = 10$

Figure 14: Influence of meta-batch size on the meta-validation performance of MAML and 5-HL so-TURTLE on 5-shot sine wave regression.

For this reason, we experiment with larger meta-batch sizes, such that meta-network updates are based upon multiple tasks instead of a single one. The results are shown in Figure 14. Looking at the results for $T = 1$, we see that both TURTLE and MAML do not benefit from larger meta-batch sizes than one. In fact, the performance decreases as the meta-batch size increases. A similar pattern can be seen for $T = 5$, although a meta-batch size of two is slightly beneficial for MAML. For $T = 10$, we see a dramatic increase in stability for TURTLE when using a meta-batch size of four compared with one and two. MAML, on the other hand, does not gain in performance when the meta-batch size is increased.

Overall, there seems to be a trade-off. Whenever possible, it seems like both MAML and TURTLE perform best for small meta-batch sizes while the performance drops quickly when larger meta-batches are used. This may be caused by the fact that the norm of the meta-update increases as the meta-batch size increases. Since we use a fixed learning rate and gradient clipping, this can indeed result in training instability. Additionally, smaller batch sizes introduce some small random noise which may actually be helpful for generalization performance.

However, as T grows larger, it is clear that learning rule updates should not be based upon a single task, as learning rules should be general. We expect that this is the reason for which increasing the meta-batch size to 4 for TURTLE greatly stabilizes the training process.

Including the update step as input It may be the case that the inclusion of the current update step allows TURTLE to learn a useful learning rate schedule that can stabilize the optimization process. More specifically, this means that TURTLE would receive an additional non-negative integer indicating the current step $t \in \{0, ..., T-1\}$. The results of this experiment is shown in Figure 15. As we can see, time as an additional input worsens the performance of TURTLE when $T = 10$, while it increases performance

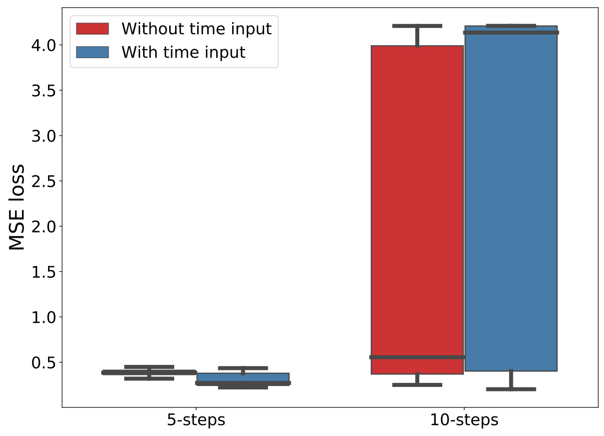


Figure 15: Influence of time input on the meta-validation performance of 10-step so-TURTLE with 5 hidden layers on 5-shot sine wave regression.

for $T = 5$.

4.2.5 Putting it all together

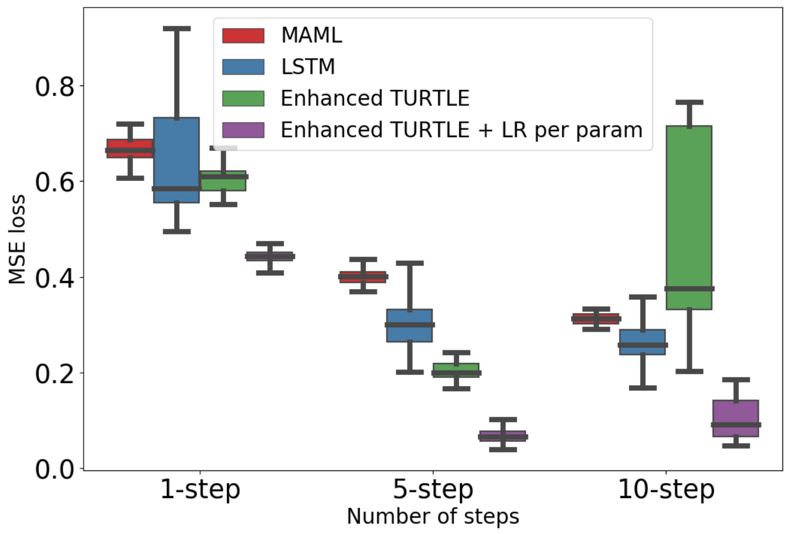


Figure 16: Comparison of the validation performance of 1-, 5-, and 10-step so-TURTLE, the LSTM meta-learner, and so-MAML on 5-shot sine wave regression.

Now it is time to combine all findings to produce the best performing TURTLE. Regardless of the number of steps, all TURTLE meta-networks are fully-connected and will use five hidden layers with 20 ReLU nodes per layer. For 1-step TURTLE, we simply use raw gradients as input, and no historical information as there is no historical information (we only make one step). The 5-step TURTLE will receive raw gradients, the time step t, and an exponentially weighted average of previous gradients (with $\beta = 0.9$)

as input. Furthermore, it will use meta-batch sizes of 2 to further stabilize the training. For 10-step TURTLE, we will use meta-batch sizes of 4 and previous gradient information with a beta-value of 0.3 (see Figure 18 in the appendix). The time step t is not included, as we found that it decreased performance (also after fixing meta-batch size at 4).

Inspired by the good 1-step results obtained by the LSTM meta-learner, we also experiment whether it is useful to learn a learning rate per parameter of the base-learner network. The final performances of all TURTLE models are shown in Figure 16.

As we can see in the figure, having a learning rate per base-learner parameter provides a boost to the performance of TURTLE, regardless of the value of T. This indicates that some parts of the base-network benefit from larger updates while other parts should stay relatively stable. This is intuitive, as some features may be important across various tasks (e.g., general concept of a sine curve), while some features are very task-specific (e.g., amplitude and phase). Furthermore, we see that TURTLE performs best when five steps are made per task, perhaps indicating that slight overfitting occurs when making 10 updates per task. Lastly, we observe that the enhanced TURTLE models (with learning rate per parameter) all outperform the MAML and LSTM meta-learner models.

4.2.6 Evaluating test performance on sine-wave regression

Next, we evaluate the TURTLE models that were found to perform best against the transfer baselines, MAML, and the LSTM meta-learner on the test tasks, which were not used for selecting hyperparameters. This allows us to get a more unbiased perspective on the performance of TURTLE. The results are shown in Table 6.

	Model	5-shot	10-shot	15-shot	20-shot
	TrainFromScratch	7.38 ± 0.12	4.61 ± 0.03	4.24 ± 0.01	3.85 ± 0.01
	Finetuning	3.16 ± 0.01	2.96 ± 0.01	3.00 ± 0.01	2.89 ± 0.01
1-step	LSTM	0.98 ± 0.14	0.76 ± 0.05	0.94 ± 0.14	1.06 ± 0.12
	so-MAML	0.68 ± 0.01	0.39 ± 0.01	0.31 ± 0.01	0.24 ± 0.00
	so-TURTLE	<u>0.47</u> ± 0.01	<u>0.26</u> ± 0.00	<u>0.20</u> ± 0.00	<u>0.15</u> ± 0.00
5-step	LSTM	0.37 ± 0.03	0.21 ± 0.03	0.21 ± 0.04	0.18 ± 0.04
	so-MAML	0.39 ± 0.01	0.10 ± 0.00	0.05 ± 0.00	0.03 ± 0.00
	so-TURTLE	**<u>0.08</u>** ± 0.01	**<u>0.02</u>** ± 0.00	**<u>0.02</u>** ± 0.00	**<u>0.01</u>** ± 0.00
10-step	LSTM	0.30 ± 0.02	0.22 ± 0.03	0.25 ± 0.07	0.40 ± 0.22
	so-MAML	0.30 ± 0.04	<u>0.07</u> ± 0.00	<u>0.03</u> ± 0.00	<u>0.02</u> ± 0.00
	so-TURTLE	<u>0.10</u> ± 0.02	0.08 ± 0.02	0.07 ± 0.02	0.34 ± 0.24

Table 6: Average MSE scores and 95% confidence intervals on $2,000$ sine-wave test tasks across 30 runs. The best performances for every task size and optimization length combination are underlined. The best performances for a specific task size are made bold.

We see that training from scratch on every new task is not a good strategy, indicating that a good prior is necessary for learning new tasks from very little data. The transfer learning baseline (finetuning) also fails to find a good prior by training on joint data from all training tasks. In contrast, the meta-learning techniques all outperform these baselines, showing that they are able to find a more useful prior (and learning strategies).

Looking at the 1- and 5-step models, we see that TURTLE outperforms the LSTM meta-learner and MAML in every setting in terms of robustness and average performance.

This shows that it is beneficial to learn priors both on the base-level and the meta-level. For the 10-step models, we observe that MAML outperforms both TURTLE and the LSTM meta-learner for $k \in \{10, 15, 20\}$. Moreover, MAML is the only algorithm that performs better when allowed to make 10 updates per task instead of 5. This could mean that the LSTM and TURTLE start to overfit their optimization strategy to these small tasks, while MAML does not have this issue as it uses a universal, hand-crafted optimizer that does not change based on the tasks.

Overall, we see that 5-step TURTLE achieves the best performance for all settings.

4.2.7 Image classification

Next, we investigate the performance of all techniques on image classification problems. In this setup, we perform 5 runs with different random initializations due to the computational costs and the fact that results are more consistent across runs.

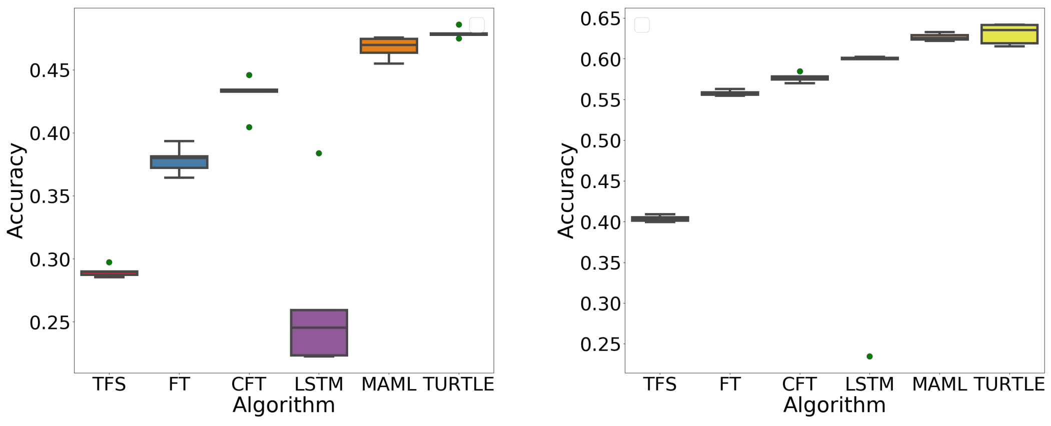
To begin with, we compare our results to those reported in other works. Other works do not report the results over various runs. Instead, they only report the best found mean classification performance and a confidence interval over all test tasks. Since the tasks that they used may differ from the ones we use, we only compare the best accuracy scores from 5 runs. The results are shown in Table 7.

	1-shot		5-shot	
	Reported	Our impl.	Reported	Our impl.
TrainFromScratch	-	0.29	-	0.41
Finetuning	0.36	0.39	0.54	0.56
Centroid finetuning	-	0.45	-	0.58
LSTM	0.43	0.38	0.61	0.60
MAML	0.49	0.48	0.63	0.63
TURTLE	-	**0.49**	-	**0.64**

Table 7: Difference between our meta-test accuracy scores and those reported in other works for 5-way image classification on miniImageNet. Our re-implemtations are shown in the column "our impl.". Note that higher accuracy indicates better performance. Reported results from finetuning, LSTM, and MAML come from Chen et al. (2019), Ravi and Larochelle (2017), and Finn et al. (2017) respectively. While Chen et al. (2019) did use the centroid finetuning model, they did not perform experiments without data augmentation on miniImageNet (hence the blank).

As one can see, our results are close to those reported by other authors. This suggests that our re-implementations are working properly. The only notable difference in performance is observed for the LSTM meta-learner on 1-shot classification. However, we have observed that the LSTM meta-learner suffers from training instability, causing the results between runs to vary by large amounts. Thus, it is possible that Ravi and Larochelle (2017) performed more runs to find a model with better performance. Also, since our 5-shot performance is very close to that reported by Ravi and Larochelle (2017), we think that our implementation is working correctly.

Moreover, we see that TURTLE (slightly) outperforms all other tested algorithms. To ensure that the difference between MAML and TURTLE is valid, we reported the best performance of MAML over 15 runs instead of 5 as done for the other algorithms. Although the difference between MAML and TURTLE is small, it has to be noted that



(a) 1-shot (b) 5-shot

Figure 17: Meta-test accuracy scores on 1- and 5-shot miniImageNet. Used abbreviations are "TFS": TrainFromScratch, "FT": Finetuning, "CFT": Centroid finetuning. The dots represent outliers.

we did not perform additional hyperparameter tuning on miniImageNet for TURTLE, whereas Finn et al. (2017) did do this for MAML. Moreover, TURTLE only makes five updates whereas MAML makes ten per task.

Figure 17 shows the distribution of accuracy scores across five runs on 1- and 5-shot miniImageNet tasks. This figure clearly shows the instability of the LSTM meta-learner in the 1-shot setting. This problem is largely resolved by moving to the 5-shot setting, indicating that the gradient signals in the 1-shot setting may be too noisy and lead to non-generalizable learned update rules. Note, however, that there is still an outlier with poor performance. Looking at TURTLE, we see that its performance is stable and slightly better than that of MAML. In the 5-shot setting, TURTLE seems a bit more unstable, but still outperforms MAML in 50% of the runs.

	1-shot	5-shot
TrainFromScratch	0.30 ± 0.00	0.46 ± 0.00
Finetuning	0.33 ± 0.00	0.53 ± 0.00
Centroid finetuning	0.36 ± 0.01	0.53 ± 0.02
LSTM	0.47 ± 0.01	0.64 ± 0.01
MAML	0.52 ± 0.00	**0.73** ± 0.01
TURTLE	**0.53** ± 0.01	0.71 ± 0.01

Table 8: Comparison of the average meta-test accuracy scores and 95% confidence intervals on 5-way CUB image classification.

We also compare the performances of the models on tasks from the CUB data set. The result are displayed in Table 8. First, we note that the accuracy scores are higher than on miniImageNet, indicating that the CUB tasks are easier to learn. This may be caused by the fact that all inputs are bird images, which narrows down the required features to correctly classify inputs. Second, we see that TURTLE (slightly) outperforms all other tested techniques in the challenging 1-shot setting but not in the 5-shot setting, where

MAML performs best.

	miniImageNet → CUB		CUB → miniImageNet	
	1-shot	5-shot	1-shot	5-shot
TrainFromScratch	0.31 ± 0.00	0.47 ± 0.00	0.29 ± 0.00	0.40 ± 0.00
Finetuning	0.33 ± 0.00	0.52 ± 0.01	0.29 ± 0.00	0.41 ± 0.00
Centroid finetuning	0.35 ± 0.01	0.52 ± 0.01	0.26 ± 0.00	0.31 ± 0.01
LSTM	0.27 ± 0.01	0.57 ± 0.01	0.29 ± 0.01	0.37 ± 0.00
MAML	0.39 ± 0.00	0.58 ± 0.00	**0.32** ± 0.00	**0.47** ± 0.00
TURTLE	**0.42** ± 0.01	**0.59** ± 0.01	0.31 ± 0.00	0.44 ± 0.00

Table 9: Average meta-test accuracy scores with 95% confidence intervals over 5 runs in task distribution shift settings.

Additionally, we investigate the performance of the algorithms when a shift in task distribution occurs. To achieve this, all models are trained on tasks from one data set and evaluated on tasks from another data set. More specifically, we investigate the scenarios (miniImageNet → CUB) and (CUB → miniImageNet), where ($x \to y$) means that we train on tasks from data set x and evaluate its performance on data set y. The results are shown in Table 9. As we can see, TURTLE outperforms the other tested techniques when trained on miniImageNet and evaluated on CUB tasks in both the 1- and 5-shot settings. When trained on CUB tasks, however, TURTLE is outperformed by MAML in both the 1- and 5-shot settings.

4.2.8 Running time

Lastly, we compare the running times of 5-step MAML, the LSTM meta-learner, and TURTLE across 5 runs. Each run includes meta-training, meta-validation, and meta-testing on miniImageNet, and evaluation on the CUB data set. The average runtimes are displayed in Table 10. In this table, we see a clear grouping of running times by the order of gradients that the algorithms use. That is, fo-MAML and the LSTM meta-learner both use first-order gradients and are equally fast. On the other hand, TURTLE and so-MAML both use second-order gradients and are slower than the first-order algorithms. TURTLE is slower than MAML due to the meta-network that has to be trained, but differences are small due to the fact that the complexity is dominated by the base-learner gradients.

Algorithm	1-shot	5-shot
fo-MAML	3h	3.7h
LSTM	3h	3.7h
so-MAML	5.8h	7.9h
TURTLE	6h	8.1h

Table 10: Average running times (hours) across 5 runs of 5-step MAML, the LSTM meta-learner, and TURTLE on miniImageNet and CUB.

5 Discussion

In this section, we interpret and discuss our results and their significance. Additionally, we describe promising directions for future research.

To begin with, we have shown the theoretical relationship between two state-of-the-art meta-learning algorithms: the LSTM meta-learner (Ravi and Larochelle, 2017) and MAML (Finn et al., 2017). More specifically, we have shown that the LSTM meta-learner is—in theory—more expressive than the latter as it can learn to perform gradient descent, which is also used by MAML to learn tasks. However, MAML was shown to outperform the LSTM meta-learner by Finn et al. (2017), which gave rise to our hypothesis that the meta-landscape, in which the LSTM meta-learner operates, is too complex to find a good solution (such as gradient descent).

As a result, we formulated a new algorithm named TURTLE, which is simpler than the LSTM meta-learner as it is stateless, yet more expressive than MAML because it can learn gradient descent as the weight update rule. TURTLE indeed outperforms both MAML and the LSTM meta-learner on the frequently used miniImageNet benchmark, which is in line with our theoretical expectations. This shows that the meta-landscape in which the LSTM meta-learner attempts to find a learning procedure is too complex to successfully navigate. Our hyperparameter analysis of TURTLE shows, however, that this complexity may not be an intrinsical property of the meta-landscape. Instead, the assumption made by LSTM meta-learner which effectively neglects second-order gradients may hurt its navigation ability.

In contrast, first-order MAML is a good approximation to second-order MAML as it yields similar performance (Finn et al., 2017). This finding highlights the distinction between the base- and meta-level goals. On the base-level, we wish to find an initialization that is close to the optimal parameters for tasks $\mathcal{T}_j \backsim p(\mathcal{T})$. Assuming reasonable smoothness of the base-level landscape, we could ignore our optimization trajectory $\boldsymbol{\theta} \rightarrow \boldsymbol{\theta}_j^{(1)} \rightarrow ... \rightarrow \boldsymbol{\theta}_j^{(T)}$ as the gradient at $\boldsymbol{\theta}_j^{(T)}$ will presumably point towards the direction of the optimal parameters for task $\mathcal{T}_j$, and hence we can move the initialization $\boldsymbol{\theta}$ in that direction. This smoothness assumption seems validated by the fact that fo-MAML is a good approximation to so-MAML. On the meta-level, in contrast, we wish to learn an optimization strategy, something which is sequential in nature. This means that an update at time step t influences the gradient inputs that the meta-network will receive at time steps $t' > t$. The consequence of ignoring second-order gradients is that the computation graph becomes disconnected (see Section 3.2), which makes the meta-network unaware of this simple fact.

A promising direction for future work is thus to alter the LSTM meta-learner by including second-order gradient information. Moreover, it would be interesting to see whether using meta-batches of size larger than one also increase its training stability as was the case for TURTLE. We expect that with these enhancements, the performance of the LSTM meta-learner will be even better than that of TURTLE as it can learn stateful update rules in a more flexible manner. As a side note, we think that the performance of TURTLE could also be improved on miniImageNet by performing hyperparameter tuning on that specific data set.

On the CUB data set, we found that TURTLE outperforms MAML in the challenging 1-shot setting, while the reverse is true in the 5-shot setting. This last observation shows that TURTLE was unable to find a similar solution as MAML, indicating that the chosen hyperparameters may not be optimal, which may be the consequence of the fact that we did not perform any additional hyperparameter tuning for any of the tested algorithms on the CUB data set. This means that further investigation is required before we can draw conclusions from this result. We leave this for future work.

In contrast to Chen et al. (2019), we have not found that simple transfer learning baselines outperform meta-learning algorithms when a task distribution shift (from miniImageNet to CUB or vice versa) occurs. This may be caused by the difference in experimental setup: they used a higher capacity base-learner network than us, which was shown to reduce the gap between meta-learning and transfer learning approaches. This is an indication that meta-learning approaches succeed in finding a more informative prior in constrained capacity settings, while transfer learning methods fail to do so. In other words, meta-learning techniques may have a better information compression rate which allows them to find a dense, informative prior. In settings where large capacity networks are used, the need for finding a such a dense prior may decrease as the models can store more information. Nonetheless, further investigating the difference between transfer and meta-learning techniques under various circumstances seems like another useful pursuit.

Interestingly, our results show that TURTLE outperforms MAML when trained on miniImageNet tasks and evaluated on CUB tasks, while the reverse was found to be true in the opposite scenario. We think that this is caused by the fact that miniImageNet has a broader input spectrum than CUB as the latter only contains bird images. Thus, TURTLE outperforms MAML when moving from a broader to a more narrow set of tasks, while it performs worse when trained on the narrow tasks and evaluated on the more diverse tasks. This can be explained by the *no free lunch theorem* for optimization (Wolpert and Macready, 1997), which suggests that domain specialization is important for achieving better performance (Andrychowicz et al., 2016). As a consequence, it is logical that TURTLE does not perform well on the diverse miniImageNet tasks when it has specialized to the more narrow CUB tasks. The fact that TURTLE does outperform MAML when applied to more narrow tasks than the ones used for training is then explained by the fact that it has specialized to a more general domain, which is more transferable to other domains.

Successfully using meta-learning techniques in settings where task distribution shifts occur, remains an important challenge within the field of meta-learning (Hospedales et al., 2020; Huisman et al., 2020). Our results indicate that in some of these challenging settings, it is beneficial to learn an optimizer. This is in line with the expectations of Chen et al. (2019) who argue that fast adaptation is crucial when task distribution shifts occur. In short, further investigating learned optimizers in these scenarios is another promising direction for future research.

Lastly, while our proposed technique TURTLE outperforms the LSTM meta-learner (and MAML) in some cases, it has to be noted that this technique does not yield state-of-the-art performance on the frequently used miniImageNet benchmark (Lu et al., 2020). Moreover, TURTLE requires propagating backwards through the entire optimization trajectory which requires storing intermediate updates and the computation of second-order gradients. While this is also the case for MAML, it has been shown that first-order MAML achieves similar performance whilst avoiding this expensive back-propagation process. Our results clearly indicate that this cannot be done for TURTLE, which means that other approaches should be investigated in order to reduce the computational costs. Future research along these lines may draw inspiration from Rajeswaran et al. (2019) who approximated second-order gradients in order to increase the speed.

6 Conclusions

In our work, we investigated a recent empirical result showing that MAML (Finn et al., 2017) outperforms the LSTM meta-learner (Ravi and Larochelle, 2017) on few-shot image classification. We find this result surprising as we can show that the latter is more expressive than the former (see Section 3.1). As a consequence, our research hypothesis was that it should be beneficial to learn an optimizer in addition to the initial parameters of the base-learner, both in regular few-shot settings as well as more challenging scenarios where task distribution shifts occur.

We formulated two possible hypotheses as to why the LSTM meta-learner failed to find a solution at least as good as that of MAML: (i) the fact that the LSTM meta-learner maintains a state increases the complexity of the meta-landscape which makes its harder to successfully navigate, and (ii) by ignoring second-order gradients, the meta-learner LSTM is navigating the error landscape with incomplete information, which makes it unable to find a good solution.

To investigate these hypotheses, we created and investigated the performance of TURTLE: a combination of MAML and the LSTM meta-learner, where we replace the stateful LSTM module by a stateless feed-forward meta-network and omit the default first-order assumption. We found that first-order TURTLE fails to achieve the same good results as the second-order variant, highlighting the importance of second-order gradients for learning an optimizer in few-shot settings, showing that hypothesis (ii) is likely to be correct.

Moreover, second-order TURTLE was found to outperform MAML on sine wave regression, the frequently used benchmark miniImageNet, and on CUB after training only on miniImageNet. However, because TURTLE was outperformed by MAML when trained on CUB and evaluated on miniImageNet tasks, we cannot conclude that a learned optimizer is always beneficial. Instead, we conclude that, in some cases, it is beneficial to learn the optimization procedure on top of the weight initialization.

Our work opens the door to fruitful future research which is likely to result in better performing meta-learning techniques.

References

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*, pages 3981–3989.

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

Elsken, T., Staffler, B., Metzen, J. H., and Hutter, F. (2020). Meta-learning of neural architectures for few-shot learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12365–12375.

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 - Volume 70*, ICML'17, page 1126–1135. JMLR.org.

Gidaris, S. and Komodakis, N. (2018). Dynamic few-shot visual learning without forgetting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4367–4375.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

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

Huisman, M., van Rijn, J. N., and Plaat, A. (2020). A survey of deep meta-learning. *arXiv preprint arXiv:2010.03522*.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y., editors, *International Conference on Learning Representations*.

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, H., Xu, Z., Taylor, G., Studer, C., and Goldstein, T. (2018). Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, pages 6389–6399.

Li, S., Liu, Z.-Q., and Chan, A. B. (2014). Heterogeneous multi-task learning for human pose estimation with deep convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 482–489.

Lu, J., Gong, P., Ye, J., and Zhang, C. (2020). Learning from very few samples: A survey. *arXiv preprint arXiv:2009.02653*.

Mahendran, A. and Vedaldi, A. (2016). Visualizing deep convolutional neural networks using natural pre-images. *International Journal of Computer Vision*, 120(3):233–255.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

Naik, D. K. and Mammone, R. J. (1992). Meta-neural networks that learn by learning. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 1, pages 437–442. IEEE.

Nichol, A., Achiam, J., and Schulman, J. (2018). On First-Order Meta-Learning Algorithms. *arXiv preprint arXiv:1803.02999*.

Pan, S. J. and Yang, Q. (2009). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

Qi, H., Brown, M., and Lowe, D. G. (2018). Low-shot learning with imprinted weights. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5822–5830.

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

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

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323:533–536.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115:211–252.

Rusu, A. A., Rao, D., Sygnowski, J., Vinyals, O., Pascanu, R., Osindero, S., and Hadsell, R. (2018). Meta-learning with latent embedding optimization. *arXiv preprint arXiv:1807.05960.*

Schmidhuber, J. (1987). Evolutionary principles in self-referential learning. *On learning how to learn: The meta-meta-... hook.) Diploma thesis, Institut f. Informatik, Tech. Univ. Munich*, 1(2).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2014). Deep inside convolutional networks: Visualising image classification models and saliency maps.

Snell, J., Swersky, K., and Zemel, R. (2017). Prototypical networks for few-shot learning. In *Advances in neural information processing systems*, pages 4077–4087.

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.

Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7).

Thrun, S. (1998). Lifelong learning algorithms. In *Learning to learn*, pages 181–209. Springer.

Vanschoren, J. (2018). Meta-learning: A survey. *arXiv preprint arXiv:1810.03548.*

Vinyals, O., Blundell, C., Lillicrap, T., Wierstra, D., et al. (2016). Matching networks for one shot learning. In *Advances in neural information processing systems*, pages 3630–3638.

Wah, C., Branson, S., Welinder, P., Perona, P., and Belongie, S. (2011). The caltech-ucsd birds-200-2011 dataset.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82.

A Additional experimental information

Here we report on some additional experiments that we have done.

A.1 Additional details of techniques

For all techniques mentioned below, we performed meta-validation after every 2500 training tasks. The best resulting configuration was evaluated at meta-test time.

Transfer baselines Note that these models (TrainFromScratch, finetuning, centroid finetuning) pre-trained on minibatches of size 16 sampled from the joint data obtained by merging all meta-training tasks. At test time, they were trained for 100 steps on minibatches of size 4 sampled from new tasks following Chen et al. (2019). Every 25 steps, we evaluated their performance on the entire support set to select the best configuration to test on the query set.

LSTM meta-learner For selecting the hyperparameters of the LSTM meta-learner[2], we followed Ravi and Larochelle (2017). That is, we use a 2-layer architecture, and Adam as meta-optimizer with a learning rate of 0.001. The batch size was set equal to the size of the task. Meta-gradients were clipped to have a norm of at most 0.25, following. The meta-network receives four inputs obtained by preprocessing the loss and gradients using Equation 11. On miniImageNet and CUB, the LSTM optimizer is set to perform 12 updates per task when the number of examples per class is $k = 1$ and 5 updates when $k = 5$.

MAML Again, we follow Finn et al. (2017) for selecting the hyperparameters, except for the meta-batch size on sine wave regression as we found it not to help performance. This means that the inner learning rate was set to 0.01 and the outer learning rate to 0.001, with Adam as meta-optimizer. These settings hold for both sine wave regression and image classification. When $T > 1$, we use gradient value clipping with a threshold of 10. On image classification, MAML was set to optimize the initial parameters based on $T = 5$ update steps, but an additional 5 steps were made afterwards to further increase the performance. Moreover, we used a meta-batch size of 4 and 2 for 1- and 5-shot image classification respectively.

TURTLE We performed many experiments with the hyperparameters of TURTLE. Here, we only report the settings that were found to give the best performance on sine wave regression, which were also used on the image classification problems. That is, the meta-network consists of 5 hidden layers of 20 nodes each. Every hidden node is followed by a ReLU nonlinearity. The input consists of a raw gradient, a historical real-valued number indicating the moving average of the previous input gradients with a (with a beta decay of 0.9), and a time step integer $t \in \{0, ..., T-1\}$. The output layer consists of a single node which corresponds to the proposed weight update. For training, we used meta-batches of size 2. Additionally, TURTLE maintains a separate learning rate for all weights in the base-learner network. Lastly, TURTLE uses second-order gradients and Adam as meta-optimizer with a learning rate of 0.001.

[2]We used code from `https://github.com/twitter/meta-learning-lstm`.

A.2 Reducing the number of layers

In an attempt to reduce the number of layers of the meta-network, we experiment with 2-hidden layer meta-network architectures with various numbers of nodes per layer.

The results are shown in Table 11. As we can see, two hidden layers are not sufficient to achieve the same performance as the 5-HL so-TURTLE model, regardless of the number of update steps per task.

Steps	Nodes	Mean MSE	Median MSE
1	Base	0.60 ± 0.01	0.61 ± 0.01
	5 Nodes	0.63 ± 0.02	0.63 ± 0.02
	40 Nodes	0.64 ± 0.02	0.62 ± 0.02
	60 Nodes	0.65 ± 0.02	0.65 ± 0.02
	80 Nodes	0.65 ± 0.03	0.63 ± 0.03
	100 Nodes	0.73 ± 0.13	0.65 ± 0.13
5	Base	0.50 ± 0.23	0.38 ± 0.23
	5 Nodes	0.59 ± 0.27	0.37 ± 0.27
	40 Nodes	2.53 ± 0.55	2.54 ± 0.55
	60 Nodes	3.14 ± 0.45	3.65 ± 0.45
	80 Nodes	3.14 ± 0.43	3.64 ± 0.43
	100 Nodes	1.04 ± 0.47	0.41 ± 0.47
10	Base	1.94 ± 0.19	0.69 ± 0.19
	5 Nodes	1.57 ± 0.53	0.92 ± 0.53
	40 Nodes	3.57 ± 0.42	4.13 ± 0.42
	60 Nodes	3.57 ± 0.43	4.04 ± 0.43
	80 Nodes	3.71 ± 0.32	3.73 ± 0.32
	100 Nodes	4.17 ± 0.34	4.25 ± 0.34

Table 11: The meta-validation performance of so-TURTLE with two hidden layers compared to the best performing 5-hidden layer model (base).

A.3 Historical information

Figure 18 displays the effect of historical input information on the performance of 10-step TURTLE which uses meta-batches of size 4. As we can see, using previous gradients with a beta value of 0.3 yields the best performance.

A.4 Effect of input type on 5-HL TURTLE

Figure 19 shows the influence of input types on the performance of 1-step, 5-HL so-TURTLE. Again, we find that raw gradients yields the most robust performance across the various runs. However, the best performance is achieved by the combination of raw gradients and loss information. This may be the result of chance, as inclusion of loss information does not seem helpful when the gradients are processed ("Ploss and Pgrads" performs worse than "Pgrads").

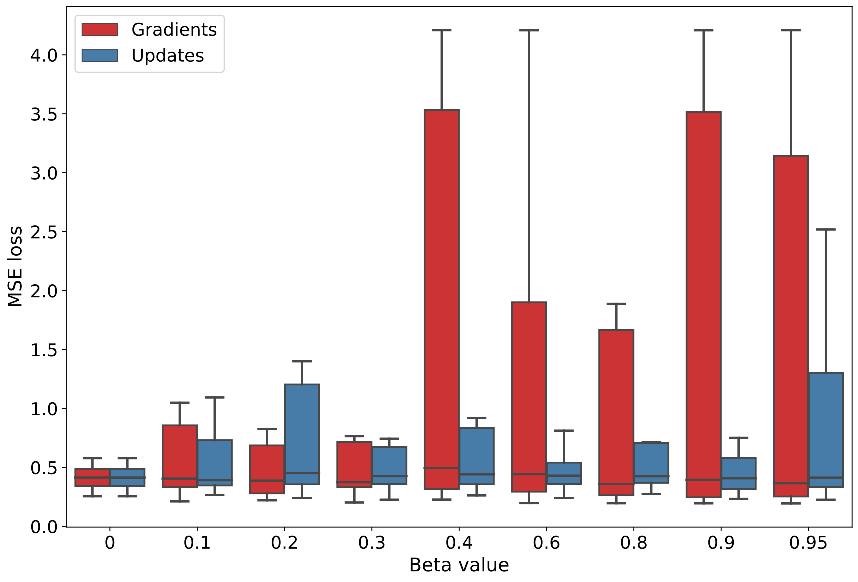


Figure 18: Influence of historical information on the meta-validation performance of 10-step, 5-HL so-TURTLE on 5-shot sine wave regression using a meta-batch size of 4.

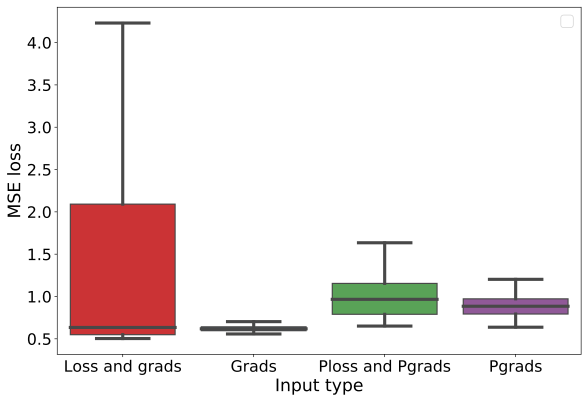


Figure 19: Influence of input information on the meta-validation performance of so-TURTLE with 5 hidden layers on 5-shot sine wave regression for $T = 1$. Here, "Ploss" and "Pgrads" refer to processed loss and processed gradients respectively.

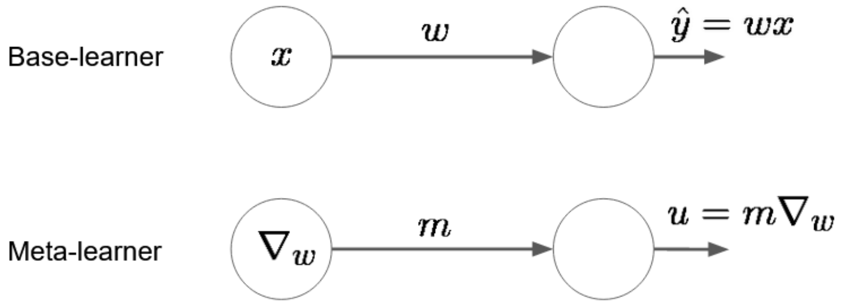


Figure 20: Example of a base-learner network with weight w and a meta-learner network with a single parameter $\boldsymbol{\phi} = m$.

B Example of TURTLE

In this section, we show an example of how TURTLE works on a single task. Suppose we have base- and meta-learner networks as shown in Figure 20. Furthermore, suppose that TURTLE takes makes two update to the base-learner parameters per task, i.e., $T = 2$.

Assume that we have a single task. A task consists of a *support set* and *query set.* The support set can be seen as training data, and the query set as validation data. For simplicity, we assume that our task looks as follows:

- Support set: Input $x_{tr} \in \mathbb{R}$ with output $y_{tr} \in \mathbb{R}$
- Query set: Input $x_{te} \in \mathbb{R}$ with output $y_{te} \in \mathbb{R}$

Lastly, assume that we want our base-learner network to minimize the MSE loss function $\mathcal{L}(y, \hat{y}) = (y - \hat{y}(w))^2$.

Normally, we would perform the following steps:

1. Randomly initialize base-learner weights w_0
2. For $t = 0, \ldots, T-1$:
3. a) Compute loss on train set $\mathcal{L}(y_{tr}, \hat{y}_{tr}(w_t))$
   b) Update the base-learner parameters (compute w_{t+1}) with SGD or Adam using $\nabla_{w_t}\mathcal{L}(y_{tr}, \hat{y}_{tr}(w_t))$

In our meta-learning context, however, we replace SGD or Adam in step b) by our meta-learning network. Also, since we introduce a trainable network, we will have to add a few steps to update our meta-learner network. Thus, the final procedure looks as follows.

1. Randomly initialize base- and meta-learner weights w_0 and m_0
2. For $t = 0, \ldots, T-1$:
3. a) Compute loss on train set $\mathcal{L}(y_{tr}, \hat{y}_{tr}(w_t))$
   b) Compute weight update by feeding the loss gradient $\nabla_w\mathcal{L}(y_{tr}, \hat{y}_{tr}(w_t))$ into our meta-learner network. Update the weights using $w_{t+1} := w_t + m\nabla_{w_t}\mathcal{L}(y_{tr}, \hat{y}_{tr}(w_t))$

c) Update the meta-learner parameters with SGD or Adam using $\nabla_m \mathcal{L}(y_{te} - \hat{y}_{te}(w_T))^2$

Thus, instead of updating our base-learner with hand-crafted rules (as SGD or Adam), we feed the gradient into our meta-learning network, which then computes weight updates for our base-learner. After updating the base-learner for a fixed number of steps T, we compute the loss on the validation/test set (x_{te}, y_{te}) in order to update our meta-learner network. Note that the latter is performed with hand-crafted rules such as SGD or Adam.

B.1 Numerical demonstration

Suppose our support and query sets are generated by a true function $f : \mathbb{R} \to \mathbb{R}$, where $f(x) = 2x + 1$. Furthermore, we assume that we make two updates to our base-learner per task, i.e., $T = 2$.

Lastly, suppose our task looks as follows:

- Support set: $x_{tr} = 2$ with $y_{tr} = 2 \times 2 + 1 = 5$
- Query set: $x_{te} = 4$ with $y_{te} = 2 \times 4 + 1 = 9$

We now work through the entire procedure for 2 updates, under the assumption that we initialize $w_0 = 1$ and $m_0 = -0.01$.

Step t = 0

1. Loss on the train set is $\mathcal{L}(y_{tr}, \hat{y}_{tr}(w_0)) = (y_{tr} - \hat{y}_{tr}(w_0))^2 = (y_{tr} - w_0 x_{tr})^2 = (5-2)^2 = 9$.
2. The partial derivative with respect to w_0 is given by $\frac{\partial}{\partial w_0}(y_{tr} - w_0 x_{tr})^2 = -2(y_{tr} - w_0 x_{tr}) \cdot x_{tr} = -4(5-2) = -12$.
3. We feed this partial derivative into meta-learner network, to get update $u_0 = m \frac{\partial}{\partial w_0}(y_{tr} - w_0 x_{tr})^2 = -0.01 \cdot -12 = 0.12$
4. We compute new base-learner weights $w_1 = w_0 + u_0 = 1 + 0.12 = 1.12$

Step t = 1

1. Compute $\mathcal{L}(y_{tr}, \hat{y}_{tr}(w_1)) = (y_{tr} - \hat{y}_{tr}(w_1))^2 = (y_{tr} - w_1 x_{tr})^2 = (5 - 1.12 \cdot 2)^2 = 2.76^2 = 7.6176$
2. Compute the derivative w.r.t. w_1, i.e., $\frac{\partial}{\partial w_1}(y_{tr} - w_1 x_{tr})^2 = -2(y_{tr} - w_1 x_{tr}) \frac{\partial}{\partial w_1} w_1 x_{tr} = -2(y_{tr} - w_1 x_{tr}) x_{tr} = -4(5 - 1.12 \cdot 2) = -4 \times 2.76 = -11.04$
3. Feed thid derivative as input to the meta-learner network to get update $u_1 = m \frac{\partial}{\partial w_1}(y_{tr} - w_1 x_{tr})^2 = -11.04m = -11.04 \times -0.01 = 0.1104$
4. Compute new base-learner weights $w_2 = w_1 + u_1 = 1.12 + 0.1104 = 1.2304$

B.1.1 Second-order TURTLE

We now show how the base-learner initialization w_0 and meta-learner parameter m are updated when using second-order TURTLE.

1. Compute the derivative of the loss with respect to the base-learner initialization w_0, i.e.,

$$\begin{aligned}
\frac{\partial}{\partial w_0}\mathcal{L}(y_{te} - \hat{y}_{te}(w_2)) &= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial w_0}\hat{y}_{te}(w_2) \\
&= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial w_0}w_2 x_{te} \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial w_0} \qquad (19)
\end{aligned}$$

Here, we can make use of the fact that

$$\begin{aligned}
\frac{\partial w_{t+1}}{\partial w_t} &= \frac{\partial}{\partial w_t}\left[w_t + m\frac{\partial}{\partial w_t}(y_{tr} - \hat{y}_{tr}(w_t))^2\right] \\
&= \frac{\partial}{\partial w_t}w_t + m\frac{\partial^2}{\partial^2 w_t}(y_{tr} - \hat{y}_{tr}(w_t))^2 \\
&= \frac{\partial}{\partial w_t}w_t - 2m\frac{\partial}{\partial w_t}\left[(y_{tr} - \hat{y}_{tr}(w_t))x_{tr}\right] \\
&= 1 - 2m\frac{\partial}{\partial w_t}\left[y_{tr}x_{tr} - \hat{y}_{tr}(w_t)x_{tr}\right] \\
&= 1 + 2m\frac{\partial}{\partial w_t}\hat{y}_{tr}(w_t)x_{tr} \\
&= 1 + 2m\frac{\partial}{\partial w_t}w_t x_{tr}^2 \\
&= 1 + 2mx_{tr}^2
\end{aligned}$$

Filling this in in Equation 19, we get

$$\begin{aligned}
-2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial w_0} &= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))(1 + 2mx_{tr}^2)(1 + 2mx_{tr})^2 \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\left[1 + 4mx_{tr}^2 + 4m^2x_{tr}^4\right] \\
&= -8(9 - 4 \cdot 1.2304)\left[1 + 4(-0.01)2^2 + 4(-0.01)^2 2^4\right] \\
&= -8(4.0784)(0.8464) \\
&\approx -27.616
\end{aligned}$$

2. Compute the derivative of this loss w.r.t. the meta-learner parameter m, i.e.,

$$\begin{aligned}
\frac{\partial}{\partial m}(y_{te} - \hat{y}_{te}(w_2))^2 &= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}\hat{y}_{te}(w_2) \\
&= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}w_2 x_{te} \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}(w_0 + u_0 + u_1) \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}(w_0 + m\frac{\partial}{\partial w_0}(y_{tr} - w_0 x_{tr})^2 + m\frac{\partial}{\partial w_1}(y_{tr} - w_1 x_{tr})^2) \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}(-2m(y_{tr} - w_0 x_{tr})x_{tr} - 2m(y_{tr} - w_1 x_{tr})x_{tr}) \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}(-2m x_{tr} y_{tr} + 2m x_{tr}^2 w_0 - 2m x_{tr} y_{tr} + \\
&\qquad 2m x_{tr}^2 (w_0 + m\frac{\partial}{\partial w_0}(y_{tr} - \hat{y}_{tr}(w_0))^2)) \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}(-4m x_{tr} y_{tr} + 4m x_{tr}^2 w_0 + 2m^2 x_{tr}^2 \frac{\partial}{\partial w_0}(y_{tr} - \hat{y}_{tr}(w_0))^2) \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}(-4m x_{tr} y_{tr} + 4m x_{tr}^2 w_0 - 4m^2 x_{tr}^3 (y_{tr} - w_0 x_{tr})) \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}(-4m x_{tr} y_{tr} + 4m x_{tr}^2 w_0 - 4m^2 x_{tr}^3 y_{tr} + 4m^2 x_{tr}^4 w_0) \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))(-4 x_{tr} y_{tr} + 4 x_{tr}^2 w_0 - 8m x_{tr}^3 y_{tr} + 8m x_{tr}^4 w_0) \\
&= -2 \cdot 4(9 - 1.2304 \cdot 4)(-4 \cdot 2 \cdot 5 + 4 \cdot 2^2 + 0.08 \cdot 2^3 \cdot 5 - 0.08 \cdot 2^4) \\
&= -32.6272(-40 + 16 + 3.2 - 1.28) \\
&= -32.6272 \cdot -22.08 \\
&\approx 720.409
\end{aligned}$$

B.1.2 First-order TURTLE

In first-order TURTLE, we ignore second-order derivatives. In mathematical terms, we assume for example that $b = \frac{\partial w_2}{w_0}$, which occurred in the gradient of the initialization of the second-order variant, is set to 1. This can be seen in the computations below.

1. Compute the derivative of the loss with respect to the base-learner initialization w_0, i.e.,

$$\begin{aligned}
\frac{\partial}{\partial w_0}\mathcal{L}(y_{te} - \hat{y}_{te}(w_2)) &= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial w_0}\hat{y}_{te}(w_2) \\
&= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial w_0}w_2 x_{te} \\
&= -2x_{te}(y_{te} - \hat{y}_{te}(w_2))\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial w_0} \\
&\approx -2x_{te}(y_{te} - \hat{y}_{te}(w_2)) \\
&\approx -8(9 - 1.2304 \cdot 4) \\
&\approx -32.627
\end{aligned}$$

2. Compute the derivative of this loss w.r.t. the meta-learner parameter m, i.e.,

$$\begin{aligned}
\frac{\partial}{\partial m}(y_{te} - \hat{y}_{te}(w_2))^2 &= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}\hat{y}_{te}(w_2) \\
&= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}w_2 x_{te} \\
&= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}(w_0 + u_0 + u_1)x_{te} \\
&= -2(y_{te} - \hat{y}_{te}(w_2))\frac{\partial}{\partial m}(w_0 + m\frac{\partial}{\partial w_0}(y_{tr} - w_0 x_{tr})^2 \\
&\qquad\qquad + m\frac{\partial}{\partial w_1}(y_{tr} - w_1 x_{tr})^2)x_{te} \\
&= -2(y_{te} - \hat{y}_{te}(w_2))(\frac{\partial}{\partial w_0}(y_{tr} - w_0 x_{tr})^2 \\
&\qquad\qquad + \frac{\partial}{\partial w_1}(y_{tr} - w_1 x_{tr})^2)x_{te} \\
&= -8(9 - 1.2304 \cdot 4)(-12 - 11.04) \\
&= -8 \times 4.0784 \times -23.04 \approx 751.730688
\end{aligned}$$