Machine learning of gradient-based optimization methods

Bachelorarbeit

von
Leonard Schröter
aus
Hamburg
Matrikelnummer: 50845
Studiengang: Computer Science

January 19, 2022

Erstprüfer: Prof. Dr. Daniel Ruprecht
Zweitprüfer: Dr. Sebastian Götschel
Betreuer: Dr. Sebastian Götschel

Eidestattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

„Machine learning of gradient-based optimization methods“

selbständig und ohne unzulässige fremde Hilfe verfasst habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich versichere, dass die eingereichte schriftliche Fassung der auf dem beigefügten Medium gespeicherten Fassung entspricht.

Ort, Datum

Unterschrift

Contents

List of Figures

1 Introduction

The invention of gradient-based optimization methods is attributed to Cauchy in 1847 [13]. Gradient-based optimization methods have therefore been known for almost two centuries. The original *gradient descent* algorithm by Cauchy works by iteratively taking small steps in the direction of the negative gradient. Many improvements and variations have been made to this original algorithm, so there exist a wide variety of different gradient-based optimization methods today. However, until today, the best results are still achieved by tuning the hyperparameters of the optimization method to make it suit better to a given problem. Therefore, we learn a gradient-based optimization method using recurrent neural networks instead of manually adjusting the hyperparameters. The primary motivation for this approach is to obtain an optimizer that generalizes well to many different problems. Such an algorithm could be a crucial component of a machine learning model that can adapt to every task it faces. The approach discussed and tested in this thesis was first proposed by Andrychowicz et al. in 2016 [1]. Since we try to learn the learning process itself, this approach falls into the domain of *meta-learning*. Meta-learning is a collective term for approaches that enhance optimization algorithms by algorithmically optimizing the learning process itself. Often, this area of machine learning is also referred to as *learning to learn*. The main application area of gradient-based optimization methods is to optimize the loss functions of neural networks. We, therefore, test how this method performs on loss functions of neural networks as well as simple quadratic functions.

The scope of this thesis contains the implementation of this approach. We also test this implementation and try to find improvements upon the work of Andrychowicz et al. [1]. Therefore, this thesis relies heavily on the aforementioned work. We start by giving an outlook on the different sections of this thesis.

1.1 Outlook

We start this thesis with a summary of the current research of meta-learning optimization algorithms. In that section, we summarize a variety of different approaches similar to the approach of this thesis. In the next section, we introduce some prerequisites for the later sections of this thesis. These prerequisites include an introduction to recurrent neural networks (RNNs) as well as Long Short-Term Memory (LSTM), which is an improvement on vanilla RNNs and was introduced by Schmidhuber and Hochreiter in 1997 [8]. We then continue with a theoretical description of the approach of this thesis, namely trying to learn a gradient-based optimization method using a recurrent neural network. After that, we describe a few interesting implementation details, including challenges we faced during implementation. The following section is about our results of experimenting with the implementation. This section includes discussing shortcomings of the approach and situations in which this approach does not work very well. In the final section, we finish this thesis by summarizing and making concluding remarks about the results.

We have given an overview of all topics in this thesis. In the next section, we summarize the current state of research of meta-learning optimization algorithms.

1.2 State of research

In this subsection, we briefly introduce a few approaches in the literature which are similar to the approach discussed in this thesis. There are many different approaches to meta-learn an optimization algorithm. These approaches often include *reinforcement learning*, sometimes as the learning objective and sometimes as the meta-learning strategy itself. Reinforcement learning is a machine learning strategy where an *agent* in an *environment* is given a *reward* for his *actions*. It then tries to find a *policy* which maximizes its cumulative reward. In this context, a policy refers to the strategy to pick an action provided a state of the environment. Since we receive limited information about the correct action in reinforcement learning, it sits between *supervised learning* (learning with labelled data) and *unsupervised learning* (learning with unlabelled data). A more in-depth introduction to reinforcement learning is, for example, given by Kaelbling et al. [10].

The first approach we want to discuss was introduced by Chen et al. in 2017 [2]. In contrast to the method in this thesis where we learn a gradient-based optimization method by using a recurrent neural network, i.e. by gradient-based optimization, Chen et al. learn reinforcement learning by gradient-based optimization. It is similar to the approach of this thesis in that it tries to meta-learn an optimizer using a recurrent neural network. The difference is that instead of having the recurrent neural network provide a step when given the gradient of the objective function as an input, they make the recurrent neural network output the new parameters of the objective function when it receives the last parameters and objective function value as input. The advantage of this method is that during testing of the learned optimizer, the objective function does not need to be differentiable. In training, the objective function still needs to be differentiable since we need to propagate the loss back through the network to learn the optimizer. In case the objective function is not differentiable during training, they, therefore, propose to approximate the gradient using one of several algorithms. By design, in our approach, the objective function always needs to be differentiable since we use the gradient of the objective function as input to our optimizer.

Another approach of learning a gradient-based optimization algorithm was introduced by Li and Malek in 2016 [14]. Later, they applied their approach to optimizing loss functions of neural networks [15]. Their approach is to learn the optimizer for gradient-based optimization methods using reinforcement learning. It is, therefore, a reverse of the approach by Chen et al. [2]. They generalize optimization techniques as a reinforcement learning policy and then try to find the optimal policy. In section 3 where we explain the approach of this thesis, we start by similarly generalizing gradient-based optimization methods.

In 2017 Wichrowska et al. [22] proposed a method that builds upon the method introduced by Andrychowicz et al. [1]. They enhance the approach by Andrychowicz et al. by employing a hierarchical network architecture. On the lowest level, they use coordinatewise recurrent neural networks similar to Andrychowicz et al. and therefore similar to the network structure that we describe in section 3.3.2. Additionally, they use so-called tensor RNNs, which combine outputs of a subset of the coordinatewise RNNs. Lastly, they combine the outputs of those tensor RNNs into a single global RNN. An

optimizer using this hierarchical network structure can, in contrast to the approach by Andrychowicz et al., use information about dependencies between coordinates.

Another very prominent approach called *Model-Agnostic Meta-Learning* (MAML) was introduced by Finn et al. in 2017 [4]. Instead of meta-learning the optimization algorithm directly, they instead learn the optimal starting parameters. The aim of this method is that basic gradient descent converges very fast on many different objective functions using these starting parameters. They do this by first performing gradient descent on the parameters of several different objective functions using the same initial parameters. After that, they perform gradient descent on the initial parameters with the final objective function values as the loss function.

There are a lot of different approaches which all have in common that they try to meta-learn the optimizer or part of the optimizer to generalize for a multitude of objective functions. In the next section, we introduce some prerequisites to the remainder of this thesis.

2 Prerequisites

In this section, we talk about some prerequisites to the remaining parts of this thesis. The approach of this thesis is to use recurrent neural networks (RNNs) to learn gradient-based optimization. In particular, we use Long Short-Term Memory (LSTM) networks. We, therefore, first give a short overview of recurrent neural networks. After that, we talk about general problems, which sometimes occur during the training of vanilla RNNs. Lastly, we discuss how LSTM networks try to solve these problems.

2.1 Recurrent neural networks

Recurrent neural networks are a special type of neural network which allows information to flow backwards in the network. This is in contrast to feed-forward networks, where information only flows from the input layer through the hidden layers towards the output layer. Recurrent neural networks allow the information to flow from one layer to another layer that came before it. They then use that information in the same way as any other information in the next step of passing data through the network. Usually, we take the output of the recurrent neural network and use it as an additional input in the next timestep. In this thesis, we refer to those recurrent neural networks as vanilla recurrent neural networks. We can see a simple schematic of this concept in figure 1.

As one can see, the recurrent neural network m works by receiving inputs x_t and h_t, which we call its internal state. It then produces an output y_t and the next internal state h_{t+1}, which we use as an input in the next timestep. We do not show the details about the specific layers of the RNN in this figure as they are not crucial to understanding the concept. We can, for example, assume the internal network structure of m to be a simple, fully connected feed-forward network with any number of layers. We will later see that there are different ways of designing the inner network structure of m.

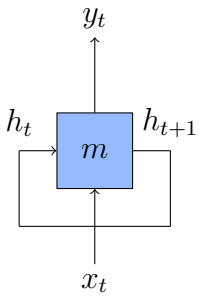


Figure 1: A schematic of a vanilla recurrent neural network called m. x_t describes the input and y_t the output of the network at timestep t. h_t describes the internal state that was passed on from the last timestep and h_{t+1} stands for the internal state that gets passed to the next step. Also, $y_t = h_{t+1}$ for each timestep t.

Due to the internal state getting passed from one step to the next, information can persist through multiple steps of passing data through the network. Recurrent neural networks are therefore inherently good at learning time series data. Although, in practice, vanilla recurrent neural networks fail to learn dependencies which are separated by large time gaps due to a problem called *vanishing gradients* that was, for example, described by Hochreiter in 1998 [6]. We cover this problem in slightly more detail in a later subsection and discuss how Long Short-Term Memory tries to solve that problem.

In general, gradient-based optimization methods, which take into account gradients not only from the current optimization step but also some steps which came before, have been observed to perform better than the standard gradient descent method. Ruder [20] gives an overview over many different gradient descent methods. These include vanilla gradient descent, as well as prominent algorithms like gradient descent using *momentum*, Nesterov Accelerated Gradient (NAG) [17] which is an improvement on momentum or ADAM [11]. The basic idea of momentum is to take bigger steps in a direction if a history of gradients points in the same direction. We can imagine this as a ball rolling down a surface getting faster the longer it rolls. One can, therefore, see that recurrent neural networks seem to be a suitable choice for learning the optimizer itself. Due to the nature of RNNs, it takes inputs, i.e. the gradients, from earlier timesteps into account, which might lead to it being able to learn desirable properties like momentum.

We have described the general idea of designing a neural network to persist information through multiple timesteps. In the next section, we talk about how we can train recurrent neural networks.

2.1.1 Backpropagation through time

We usually train neural networks by using a gradient-based optimization method. For this, we need to be able to compute the gradient of the loss function with respect to the network weights. We can compute this gradient by using an algorithm called *backpropagation*. The main idea of this algorithm is to go through all operations that we did

to attain the loss function in reverse order. During this, we successively compute the gradients using the chain rule until we reach the parameters in question. In this thesis, we do not go into the mathematical details of this algorithm. An in-depth description was, for example, given by Hecht-Nielsen in 1992 [5]. In this section, we instead describe how we can use backpropagation to compute the gradients for recurrent neural networks.

There is a different way compared to figure 1 that we can think of recurrent neural networks. That is, we can unroll a recurrent neural network in time. To see how this looks, we can take a look at figure 2.

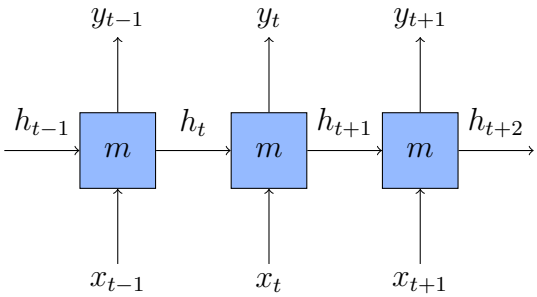


Figure 2: A schematic of a recurrent neural network that is unrolled in time. x_t describes the input and y_t the output of the network at timestep t. The internal state h_t is passed from one timestep to the next.

Instead of thinking of a neural network as a single network with a loop for the internal state, we can think of it as having a copy of the neural network for each timestep. The timesteps connect by the internal state h, which goes from one copy of the neural network to the next. By picturing the recurrent neural network unrolled in time, we can see that we can use the well-known backpropagation algorithm to compute the gradients. The only difference is that we let the gradients flow backwards through the different timesteps giving rise to the name backpropagation through time. We do not give a detailed mathematical description of this process in this thesis. For example, Werbos [21] provides a more detailed mathematical description of backpropagation through time.

In the next subsection, we discuss issues with vanilla recurrent neural networks, which make them unable to learn dependencies separated by large time gaps.

2.1.2 Problems with recurrent neural networks

In this subsection, we explore some issues with vanilla recurrent neural networks, which prevent them from learning dependencies that are separated by large time gaps.

Two main issues prevent vanilla recurrent neural networks from learning dependencies separated by large time gaps. They are called *vanishing gradients* and *exploding gradients*. These issues are, for example, explored by Hochreiter in 1991 [7] and 1998 [6]. Hochreiter is also one of the co-authors of the Long Short-Term Memory paper [8] which tries to solve these two problems. We introduce LSTM networks in the next section. On the one hand, the vanishing gradient problem describes the phenomenon that the gradients are

very small in magnitude. On the other hand, the exploding gradient problem describes the case of the gradients being very large in magnitude. Both issues are likely to occur in vanilla recurrent neural networks and render the network unable to learn efficiently. If the magnitude of the gradients is very small, the update steps of the network are also very small, making the optimization algorithm unable to perform meaningful parameter updates. If the gradient is very large in magnitude, the update steps become too large for the optimization algorithm to find the optimum. Hochreiter [7] proves that these problems stem from the fact that in vanilla RNNs, the influence of weights on the output depends on other weights. He shows that due to the chain rule, this dependency is multiplicative and, therefore, grows exponentially with the length of the time gap over which information is stored. The influence of weights on the output is described by the magnitude of the coordinates of the gradient. Due to this exponential dependency, the gradients decrease or increase very fast for growing time gaps, which can lead to the issue of vanishing and exploding gradients.

One approach to solving the exploding gradient problem is *gradient clipping* for example explored by Pascanu et al. in 2013 [18]. This method cuts off all gradient coordinates that exceed a certain magnitude. Another approach to solving both of these problems is Long Short-Term Memory. We give a short introduction to LSTM networks in the next section.

2.2 Long Short-Term Memory

In this section, we give an introduction to Long Short-Term Memory (LSTM) networks which we later use to learn a gradient-based optimization algorithm. LSTM networks were introduced by Schmidhuber and Hochreiter in 1997 [8]. Their main goal is to try to solve the issue of vanilla recurrent neural networks being unable to learn long-term dependencies. Before we start, it is important to note that there exist different variants of LSTM networks. In this section, we describe a basic LSTM variant which is the same as in Hua et al. [9].

In general, LSTM networks work like vanilla recurrent neural networks in that they pass an internal state from one timestep to the next. In vanilla recurrent neural networks, the internal state gets mutated a lot by being passed through an entire feed-forward network. The big difference of LSTM networks now is that there is an additional internal state called *cell state* which can only be changed at two places inside the LSTM network. First, the network decides which information it wants to forget, and then it decides which new information it wants to remember. That way, information can persist easier over a large number of steps. To see how this process works in detail, we take a look at the computational graph of the internal structure of an LSTM network in figure 3. Note that this figure is inspired by Hua et al. [9].

As before, x_t stands for the input to the network and y_t describes the output. As mentioned earlier, we have two internal states. First, the internal state h_t, which we know from vanilla recurrent neural networks, is the same as the output y_{t-1} of one timestep earlier. Additionally, we have another internal state c_t called the cell state. In general, all

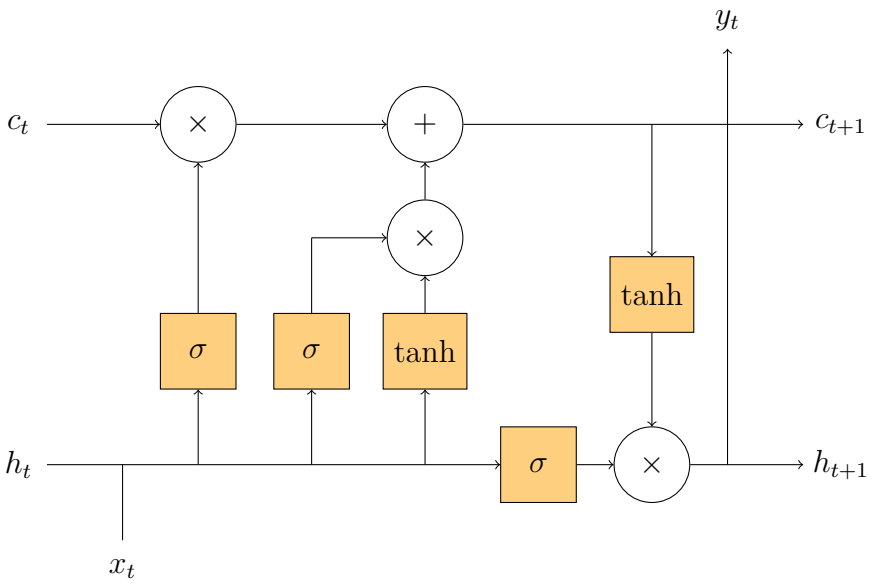


Figure 3: Computational graph of the inner structure of an LSTM network. x_t describes the input and y_t the output of the network. h_t describes the internal state and c_t describes the cell state that was passed from the last timestep. The circles stand for the coordinatewise variant of that operation and the yellow boxes represent fully connected neural network layers that use the function inside the box as activation function.

of these entities are vectors. The circles with mathematical operations all stand for the coordinatewise variant of that operation. Each orange box represents a neural network layer. The symbols inside those boxes describe the activation function of the given layer. σ stands for the well-known sigmoid activation function and tanh for the tanh activation function. We now go through all of these components in the order that an input x_t would take. Meaning we talk about the activation functions in order from left to right as depicted in figure 3.

The first operation is the concatenation of the input vector x_t and the internal state h_t. We do not depict this concatenation with any symbol but just the two lines joining. For simplicity, we also refer to this concatenated vector as the input vector. First, the input vector gets passed through a sigmoid layer which maps the input to a vector in $(0,1)^n$ where n corresponds to the size of the cell state c_t. This is because the next step involves the coordinatewise multiplication of this vector with the cell state. We can think of the first sigmoid layer as the network deciding which information it wants to forget from the cell state, and the coordinatewise multiplication is the actual process of forgetting. The second sigmoid layer again maps the input vector to a vector in $(0,1)^n$. We will describe this vector's purpose after talking about the next tanh layer. Opposed to the sigmoid layers, the tanh layer maps the input vector to a vector in $(-1,1)^n$. With this tanh layer, the network decides what the new information is that the cell state should store. Now, coming back to the second sigmoid layer, we can see that it gets multiplied coordinatewise with the output vector of the tanh layer, which corresponds to the new

information for the cell state to remember. We can interpret this multiplication as the sigmoid layer filtering the information that the tanh layer wants the cell state to store. After filtering the information, we add the new information to the cell state to store it for later timesteps. As we can see, these are all operations that are done to manipulate the cell state c_t. Therefore, the change of c_t in each step might be small, making it easier for the network to store information long-term inside the cell state.

The remaining operations determine the output y_t of the network that corresponds to the internal state h_{t+1} that gets passed to the next step. As we can see, the output gets determined by the already changed cell state. In particular, this is done by first passing the mutated cell state through a tanh layer which outputs a vector in $(-1, 1)^m$. Here, m corresponds to the size of the output vector y_t. The output of this tanh layer determines the values we want to output. Next, we pass the input vector through yet another sigmoid layer outputting a vector in $(0, 1)^m$. Similar to the step of filtering the information the cell state should remember, we multiply this vector with the output of the tanh layer. The result then finally determines the output y_t as well as the internal state h_{t+1} of the next step.

We have introduced a standard variation of Long Short-Term Memory networks. In the next section, we describe the approach of this thesis in a theoretical manner.

3 Approach

In this section, we talk about the theoretical details of learning gradient-based optimization using a recurrent neural network. This section is based heavily on Andrychowicz et al. [1]. Note that we also use the same notation as in Andrychowicz et al.

We refer to the objective function we want to optimize using a learned optimizer as f. Note that in most practical cases, f would be a loss function of a neural network, but we describe the approach in a more general manner to allow for any objective function f. Most gradient-based optimization methods have in common that they update the parameters θ of the objective function f in each timestep t by using the gradient of f with respect to θ evaluated at θ_t. We denote this gradient with $\nabla_\theta f(\theta_t)$. Here, θ_t stands for the values of θ at the timestep t. Probably the most basic gradient-based optimization method is called *gradient descent* and uses the following formula to update the parameters θ of the objective function f in each step t:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta f(\theta_t). \tag{3.1}$$

In this formula, α describes an arbitrary learning rate. As we can see, in every timestep t, we adjust the current parameters θ_t by a step in the direction of the negative gradient. There are much more sophisticated methods that work much better on rough objective functions, especially loss functions of neural networks. These methods include, for example, the already mentioned Nesterov Accelerated Gradient (NAG) [17] or ADAM [11]. All these methods have in common that they adjust the parameters θ in every timestep by a small step in a particular direction. This step always depends on the

gradient of the objective function with respect to the parameters at the current timestep. We can generalize these methods by the following formula:

$$\theta_{t+1} = \theta_t + g(\nabla_\theta f(\theta_t)). \tag{3.2}$$

Here, the function g describes how any of the earlier mentioned optimization methods generate the step from the gradient $\nabla_\theta f(\theta_t)$ as input. The main idea of this thesis which was introduced by Andrychowicz et al. [1] now consists of trying to learn g by using a recurrent neural network. This network should take $\nabla_\theta f(\theta_t)$ as input and output the optimal step to take, which we, for simplicity, also denote with g. In the following, we also use the term optimizer network for this network and call it m. We now slightly modify formula 3.2 to include the parameters of the optimizer network, which we call ϕ. In this case, ϕ corresponds to the weights and biases of the optimizer network. To do this, we let the output g of the optimizer network depend on the parameters ϕ:

$$\theta_{t+1} = \theta_t + g(\nabla_\theta f(\theta_t), \phi). \tag{3.3}$$

Note that some of the aforementioned gradient-based optimization methods may work with gradients from earlier optimization steps as additional information. In a later subsection, we discuss how this information can also be used by our learned optimizer.

To be able to learn the optimizer network m, which should output the step g, we first need to define a loss function for this network. We discuss this in the next subsection.

3.1 Loss of the optimizer

In this subsection, we define the loss function for learning the optimizer network m. Note that in the following, we use the symbol f to represent an entire distribution of objective functions and a single objective function interchangeably. We use this simplification for readability purposes, but we also want to illustrate that we want to learn an optimizer that can generalize over an entire distribution of objective functions instead of just working well on a single objective function.

Since we want to learn the optimizer network in a way such that it minimizes the objective function f, we define it through the objective function itself. Since we explicitly want the expected final values of the objective function to be minimal, we could use the following expression as the loss function for the optimizer network:

$$\mathcal{L}(\phi) = \mathbb{E}_f\Big[f\big(\theta^*(f, \phi)\big)\Big]. \tag{3.4}$$

First, we take the final value of a given objective function f after optimizing it. We then take the expected value of the final objective function values over the distribution of objective functions in question. θ^* represents the final values of the parameters θ after the optimization process. Note that we let θ^* be dependent on the objective function f itself as well as the parameters ϕ of the optimizer network. Andrychowicz et al. [1] describe this as being "a slight misuse of notation". We use this notation firstly to highlight that the final parameters θ^* depend on the sampled objective function itself. Secondly, we want

to highlight that the final parameters θ^* depend on the parameters ϕ of the optimizer network. This is the case due to θ^* being the result of updating θ in every step using the outputs g_t of the optimizer network m, which in turn depend on the parameters ϕ.

However, there are some issues with this definition of the loss function. First, we would have to execute an entire optimization process of the objective function for a single update of the parameters ϕ. This loss function would therefore render this method inefficient. Another issue is that throughout the optimization process of a single objective function, we would have to keep track of every operation which updates the current parameters θ_t with the step g_t. We would have to do this to be able to compute the gradient with respect to ϕ at the end of optimization since each g_t depends on ϕ. When optimizing for a large number of steps, this gives rise to a huge memory overhead, which, at least on the single GPU system on which we test the implementation, results in a memory overflow after a small number of steps. We discuss this problem of memory overhead in section 5.5 in more detail. Due to these issues with only using the information of the last step of optimizing the objective function, we can instead define the loss function for the optimizer network by using the information of multiple timesteps:

$$\mathcal{L}(\phi) = \mathbb{E}_f \left[\sum_{t=1}^{T} w_t f(\theta_t) \right] \text{ where } \theta_t = \theta_t(f, \phi). \tag{3.5}$$

The difference to the loss function defined in equation 3.4 is that we sum all values of the objective function over a time horizon T. This allows us to use the information of multiple steps of optimizing the objective function to learn the optimizer network. Furthermore, we can divide the optimization process of f into slices of length T and update the parameters ϕ once for each of these parts. This approach solves the first issue described in the last paragraph. For sufficiently small T, we also prevent the problem with memory overhead since we now only have to at most store operations from the last T steps of optimizing the objective function. Another small difference compared to equation 3.4 is that we weight each value of f by an arbitrary weight w_t. These weights give us another opportunity to finetune the learning of the optimizer. In section 5.6, we discuss different weighting strategies and test how they perform against each other. On a last note, θ_t in this equation again depends on f and ϕ similar to equation 3.4.

We have now defined the loss function for learning the optimizer network. Next, we need to compute the gradient of the loss function with respect to ϕ to be able to train the optimizer network. We discuss this in the following subsection.

3.2 Learning the optimizer

In this subsection, we talk about how to compute the gradient of the loss function that we defined in section 3.1 with respect to the optimizer parameters for learning the optimizer network.

We start by repeating some notations which we need for the remainder of this section. The following equation depicts the simple update rule that we want to use to optimize the objective function f:

$$\theta_{t+1} = \theta_t + g_t. \tag{3.6}$$

We derived this by generalizing the update rules of different gradient-based optimization methods. Additionally, we introduced the optimizer network m, which should take the gradient of f at θ_t as an input and provide the step g_t as output. Remember that the neural network m should be modelled as a recurrent neural network. Therefore m, besides outputting g_t, also outputs an internal state h_t[1] which is used as input to the next step. We can write this relation as

$$\begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} = m(\nabla_t, h_t, \phi). \tag{3.7}$$

Note that we use ∇_t as an abbreviation for $\nabla_\theta f(\theta_t)$. In this equation we wrote the optimizer network m as a function of its parameters ϕ, its internal state h_t as well as the gradient of the objective function for the current step. It outputs the step g_t for optimizing the objective function as well as the next internal state h_{t+1}. We now want to adjust the parameters ϕ in a way such that the output g_t minimizes the objective function f using equation 3.6. For this we need to compute the gradient $\partial\mathcal{L}(\phi)/\partial\phi$ where the loss function $\mathcal{L}(\phi)$ is the loss function that we defined in 3.5. Note that in practice we need to sample a specific objective function f from the provided distribution to approximate the expected value over the possibly infinitely large distribution of objective functions f. To see how we can compute the gradient, we can take a look at the computational graph in figure 4.

The computational graph shows three timesteps of the optimizing an objective function using the optimizer network m. Note that we use ∇_t as an abbreviation for $\nabla_\theta f(\theta_t)$ similar to equation 3.7. In each step, we use the gradient of the objective function f with respect to the parameters θ as input to the optimizer network m. Then we add the output g to θ and repeat the process. We now want to compute the gradient of the loss function $\mathcal{L}(\phi)$ with respect to the parameters ϕ of the optimizer network. Remember that we defined the loss function by adding the objective function over multiple timesteps. By the sum rule for derivatives, we, therefore, need to compute the gradient of the single timesteps. We do this by using backpropagation on the solid lines of the computational graph in figure 4. Since we do backpropagation over multiple timesteps, this is similar to the backpropagation through time algorithm that we explained in section 2.1.1. As one may notice, there are dashed lines going through the gradients in the computational graph. These indicate that we disregard these dependencies during backpropagation. In mathematical terms, we set

$$\frac{\partial \nabla_t}{\partial \phi} = 0. \tag{3.8}$$

[1]To be exact, the optimizer network m should be implemented by an LSTM network which carries its internal state as well as its cell state into the next step. For the sake of generality, we assume h_t to represent all internal states of every type of recurrent neural network that we could use to model the optimizer. Note that we could even use a simple feed-forward network to model the optimizer using this notation. Then $h_t = 0\ \forall t \in \{1, ..., T\}$. would hold.

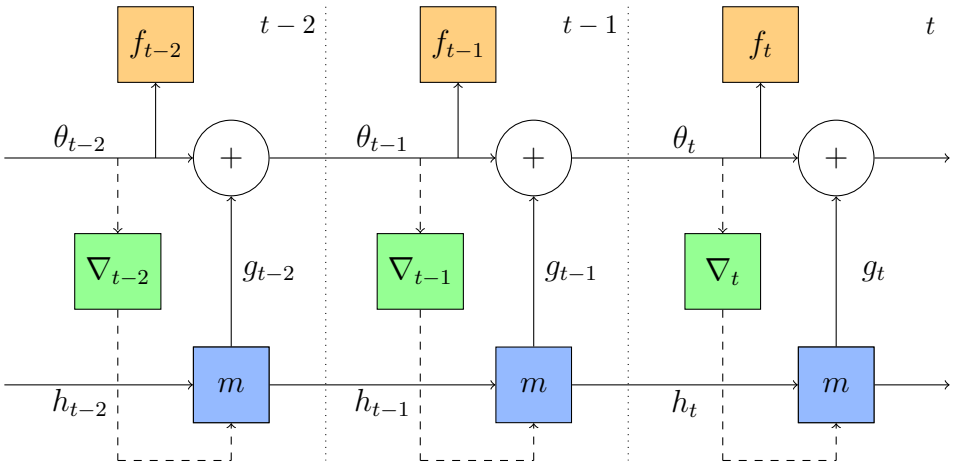


Figure 4: Computational graph representing multiple timesteps of optimizing an objective function f using the optimizer network m. ∇_t refers to the gradient of the objective function and g_t to the output of the optimizer network. θ_t describes the parameters of the objective function and h_t the internal state of the optimizer network.

In colloquial language, this restriction says that the gradient of the objective function does not depend on the parameters of the optimizer network. We do this to prevent us from having to compute second-order derivatives of f. After we are done computing the gradients of the loss function, we use these gradients to update the parameters ϕ of the optimizer network. For this, we use the ADAM optimizer [11] since it generally produces good results.

We have now finished describing how we train the optimizer network. In the following subsection, we talk about the internal network structure of this network.

3.3 Network structure of the optimizer

In this section, we talk about the structure of our optimizer network m. In earlier sections, we have stated that we use a recurrent neural network for modelling the optimizer network. This is not an arbitrary choice. In the next subsection, we describe how recurrent neural networks have a beneficial property for implementing a learned optimizer.

3.3.1 Benefits of using recurrent neural networks

In section 3, we described how to generalize the update rule of gradient-based optimization methods. The main idea is that all methods work by updating the objective function parameters θ by a small step which they generate in different ways from the gradient of the objective function. See equation 3.2 for a general formula describing this process. One issue one may find with this generalization is that some methods like NAG [17] or ADAM [11] additionally to using the current gradient also use gradients from earlier timesteps to generate the step g_t. This does not get captured explicitly by formula 3.2. Now, the

step g_t is the output of the optimizer network m. The optimizer network generates g_t from the gradient of the objective function as input. The benefit of using a recurrent neural network for the optimizer network comes from the fact that recurrent neural networks can store information through multiple timesteps. Therefore, if we implement the optimizer network with a recurrent neural network, it can also use the information of earlier gradients instead of just the current gradient. Since gradient-based optimization methods which use earlier gradients are generally an improvement to methods that do not, we use recurrent neural networks to learn the optimizer.

We have now described why we use recurrent neural networks to model the optimizer. In the following subsection, we explain the coordinatewise network structure, which also provides certain benefits.

3.3.2 Coordinatewise network structure

In this section, we talk about the coordinatewise network structure of the optimizer network.

In section 3.3.1, we have described why we use a recurrent neural network to model the optimizer network m. In section 2.1.2, we also talked about some issues with vanilla recurrent neural networks. These issues are mostly solved by Long Short-Term Memory networks [8] as described in 2.2. Therefore, we use a Long Short-Term Memory network to implement the optimizer network m. The natural way to do this would be to have a single LSTM network that receives the entire vector $\nabla_\theta f(\theta_t)$ as input and outputs the entire vector g_t. However, this has two disadvantages. First, especially for a large parameter vector θ, this gets very computationally expensive. Second, the size of the parameter vector θ has to be constant over the entire distribution of objective functions f. It would therefore be impossible to test a trained optimizer network on another objective function that has a different parameter size. We, therefore, use another network structure where we use a separate small LSTM network for each coordinate in the parameter vector θ. If we trained a separate network for each parameter, we would have the same problem of not being able to generalize the optimizer to a problem with a different parameter size. We, therefore, let all the small networks share their parameters. This results in us having to deal with a much smaller parameter vector ϕ of the optimizer network, which reduces computational complexity drastically. Additionally, we can add or remove copies of this network to use the trained optimizer for a problem with a different number of parameters. An issue with this approach might be that we get the same behaviour in each coordinate of the parameter vector θ. We prevent this by letting each LSTM network carry its own internal state. One thing we lose using this approach is the ability to use dependencies between different coordinates in θ. As mentioned in section 1.2, there is an approach by Wichrowska et al. [22] which tries to solve this issue by using a hierarchical network structure for the optimizer network.

Note that the approach of using a coordinatewise network structure does not only work with LSTM networks but can be generalized to any type of recurrent neural network. However, the LSTM network will continue to be the network type for the optimizer that we use in this thesis.

We have now completed this section, where we explained the approach of this thesis in a theoretical way. In the next section, we talk about interesting implementation details that we faced during the implementation of the theoretical approach of this section.

4 Implementation details

In this section, we talk about some interesting implementation details of the theoretical approach described in section 3. We use Python for the implementation. As a machine learning library, we use the Keras framework [3] built on top of the TensorFlow 2 framework [16]. In the words of the developers, TensorFlow is "an interface for expressing machine learning algorithms". The most important task it undertakes is the computation of gradients using the backpropagation algorithm. The Keras framework provides abstractions for often used entities and algorithms for building neural networks. This includes a wide variety of different neural network types, including, for example, LSTM networks which we use for our implementation. They also provide a comprehensive selection of different optimizers, activation functions and a lot of other useful utilities for designing custom deep learning models. These two libraries, therefore, build the perfect basis for implementing the approach of meta-learning the optimizer. In the following subsections, we first describe one case in which we can misappropriate an interface of Keras to make implementing the coordinatewise network structure much easier. We then describe an issue with TensorFlow where it is impossible to compute the gradients in certain cases. We encounter this issue when trying to do a straightforward implementation of the approach of this thesis. After that, we introduce the workaround that we use to circumvent this issue.

4.1 Implementing a coordinatewise neural network structure

In this section, we explain how we can leverage a design choice of the Keras framework to implement the coordinatewise neural network structure that we introduced in section 3.3.2. We first introduce TensorFlow tensors, the basic building block of the TensorFlow framework. Tensors are the fundamental data structure that is used in TensorFlow (and therefore also Keras) to store and manipulate any data. This includes, for example, neural network weights and biases as well as neural network inputs and outputs. Tensors are essentially multidimensional arrays. They can only hold a single data type and cannot change their shape. The shape of a tensor is a tuple defining how many elements the tensor has in each axis. For example, a tensor with a shape defined as $(3, 2)$ could be interpreted as a 3×2 - matrix or an array containing three subarrays of length two.

All Keras layers have the property that they take inputs in a specific shape. Note that layers in Keras represent layers in neural networks. This shape is usually of the form

```
(batch, feature).
```

The first axis, called batch, defines how many samples we want to input together, and the second axis defines the number of features, i.e. numerical values for each input.

A sample, in this case, refers to a single input. For recurrent neural network layers in Keras, including LSTM network layers, when using the default configuration, the inputs have to have the shape

`(batch, timestep, feature).`

The difference is that we have another axis for inputting multiple timesteps at once. If we were to input multiple timesteps at once, Keras would implicitly carry the internal state and cell state from one timestep to the next. In the case of implementing the coordinatewise network structure, we set the number of timesteps to one since we have to add the output of the network to the objective function parameters in each timestep. Only then do we get access to the next input, i.e. the gradient of the objective function evaluated at the new parameters. We, therefore, need to take care of carrying the hidden and cell states through the timesteps ourselves. One option is to store the internal states which get output by the coordinatewise network in a variable and then provide it as input in the next step. We can alternatively use a configuration option of the Keras LSTM layer to let the network store the internal states after each step and apply them automatically in the next forward pass. Using this option, the network stores one dedicated internal state for each sample. We can use this implementation detail to implement a coordinatewise network structure. To do this, we let each sample represent one coordinate of the objective function parameters. This way, we achieve both main properties of a coordinatewise network structure. First, we share the network weights between all the copies of the network since we only define a single network where the copies are represented by the different samples in the input tensor. Second, we have a unique internal state for each coordinate. Therefore, the only thing we need to do to implement a coordinatewise network structure for the optimizer is to enable the aforementioned configuration option and set the input shape to

`(N, 1, 1).`

Here, N describes the number of parameters of our objective function. We do this to have one sample for each coordinate of the objective function parameters due to reasons described before. As also explained before, we set the number of timesteps per input to one. Note that we also set the number of features to one. This is because we input a scalar into each copy of the optimizer network, which corresponds to the gradient of the objective function in a single coordinate. However, there is a caveat. After inputting a tensor with a specific batch size into a Keras LSTM layer with this option enabled, we can only input tensors with the same batch size. Therefore, to be able to switch to a different objective function parameter size, we have to retrieve the weights from the current network and use them to initialize a new network.

We have seen how we can implement the coordinatewise network structure of the optimizer in Keras. In the next subsection, we describe a problem with tracking gradients through variable updates. After that, we discuss a workaround that we use in our implementation.

4.2 Tracking operations through variable updates

In the last subsection, we gave a short introduction to TensorFlow tensors and their properties. One additional important property of vanilla TensorFlow is that they are immutable. Immutability means that we cannot update the values inside a tensor that already exists. Instead, we have to create a new tensor every time we want the values of a tensor to change. But, there exist other types of TensorFlow tensors. One of these variations, which we also use in the implementation, is called a variable. They form a wrapper around TensorFlow tensors and can be used in any way a vanilla tensor can. Additionally, as the name suggests, they are mutable, meaning they can be changed in place using specific methods. These methods come in very handy when using TensorFlow variables as the data store for the weights and biases in a neural network layer. The weights need to be updated a lot of times during training which would make reassigning them in each training step very cumbersome. For this reason, Keras uses TensorFlow variables for storing weights and biases in neural network layers.

However, there exists one caveat concerning the differentiability of in-place assignments of TensorFlow variables. One purpose of TensorFlow tensors is to track operations to be able to compute gradients using the backpropagation algorithm. Unfortunately, operations of assigning new values to TensorFlow variables in place do not get tracked[2]. The new state after the in-place assignment of a TensorFlow variable acts like a completely fresh tensor in terms of computing the gradients. Therefore, computing the gradient of this variable with respect to any tensor which influenced its new value will result in zeros. However, this is most likely not the correct result in a mathematical sense. To explain how this is an issue when implementing the idea of this thesis, we take a look at the computational graph from figure 5.

The computational graph in figure 5 is the same as the computational graph from figure 4. As one can see, there is one path that is marked in red. This path is one example through which we need to perform backpropagation to compute the gradient of the loss function with respect to the parameters of the optimizer network m. Now, imagine the case where f represents the loss function of a neural network. In that case, the parameters θ correspond to the weights and biases of that neural network. In an implementation using default Keras layers, these weights and biases would be implemented using TensorFlow variables. Unfortunately, we reassign the value of the parameters θ by adding the optimizer output g to it. In a straightforward implementation, this update would be implemented using an assignment method of the TensorFlow variable class. However, as earlier mentioned, TensorFlow does not track dependencies through variable updates. Therefore, the parameters θ at timestep t do not know about their dependency on the optimizer output g_{t-1} and therefore on the parameters ϕ of the optimizer network. In a straightforward implementation, the gradient of the loss function with respect to the parameters of the optimizer network therefore always returns zero making it impossible to learn the optimizer network.

[2]In a GitHub issue in 2016, a TensorFlow developer confirmed that they "never track anything across variable updates" as this could lead to unexpected behaviour. https://github.com/tensorflow/tensorflow/issues/6193, Accessed: 15.12.2021

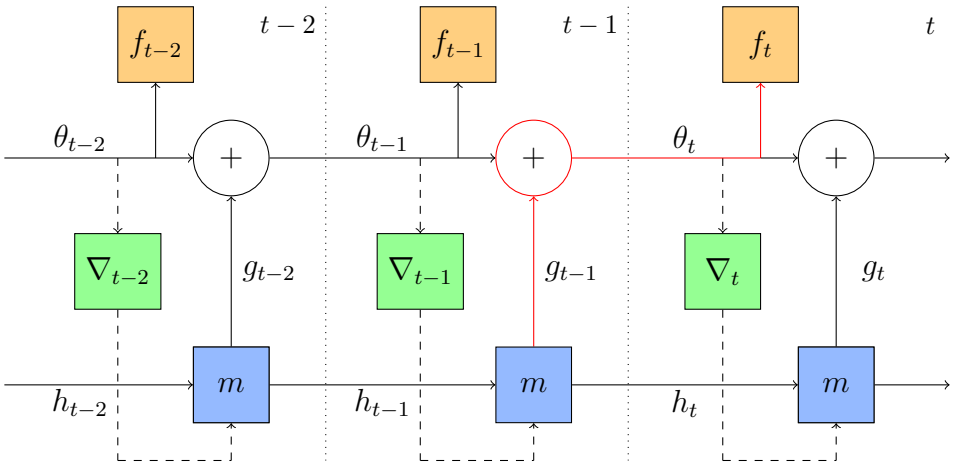


Figure 5: Computational graph representing multiple timesteps of optimizing an objective function f using the optimizer network m. This is the same as the computational graph from figure 4. We marked one path through which we need to perform backpropagation in red.

Incidentally, a similar issue occurs when trying to do a straightforward implementation using the PyTorch framework [19], another prominent machine learning framework. To solve this issue, we had to find a workaround which we discuss in the following subsection.

4.3 Solving the issue with tracking operations through variable updates

In this section, we describe a workaround that we use to circumvent the issue that originates from TensorFlow not tracking dependencies through variable updates. Note that we found this solution by looking at the implementation[3] by Andrychowicz et al. [1]. However, this implementation uses TensorFlow 1, which is very different from TensorFlow 2, which we use for our implementation. Therefore, studying their implementation gave us merely a hint in the right direction.

The issue that we described in the last section arises when we store the parameters of the objective function inside of TensorFlow variables. The first idea that comes to mind is to instead store them inside vanilla TensorFlow tensors that we manage ourselves. However, there is an issue with this approach. Since tensors are immutable, each time we update the values of the parameters, we have to initialize a new tensor. This would be no problem if we had control over the entire implementation of the objective function. The issue arises when we ultimately want the objective function to be any neural network implemented with Keras. This is due to Keras managing the weights and biases internally using TensorFlow variables. We, therefore, do not have direct control over these variables.

[3]Their repository is available on GitHub. https://github.com/deepmind/learning-to-learn, Accessed: 15.12.2021

However, there is a way to overwrite the default behaviour of storing the weights and biases of Keras layers, which we describe in the following.

The variables which store the weights and biases of a layer only get initialized when the shape of the input tensor is known. This shape is initialized usually on the first forward pass through the neural network. We, therefore, need to overwrite the weight variables with our self managed tensors during this initialization process. This is not an easy task since we need to know the name of the property of the layer class in which the weights get stored internally. We need this name to overwrite this property with our tensors. Fortunately, all weights get initialized using an internal function that receives a name as an input parameter. This name corresponds to the name of the property under which the weights get stored. This is not guaranteed to be the case for every Keras layer. However, it is the case for all the standard layers that we checked. This is also the reason why our implementation is not a universal implementation that is guaranteed to work for any Keras layer or version of the two frameworks as the internal property names may change. We now employ our tensors as weights and biases by overwriting the default behaviour of the function that is used to initialize the weights and biases internally. For this, we use a Python library that can change the behaviour of a function inside a context. We do this such that we do not have to change the code of Keras directly. We first create a Python dictionary in which we later store these tensors holding the weights and biases. This dictionary is filled dynamically by our function that overwrites the function that initializes the weights and biases. This new function creates a nested dictionary inside the main dictionary for each layer and creates a new tensor for each of the weight and bias tensors of that layer. After the first forward pass through the objective neural network, the dictionary storing the weights and biases might look like the code snipped in figure 6.

```
1  import tensorflow as tf
2
3  # example state of the dictionary for storing the weights
4  # and biases after initialization of the objective neural network
5  objective_network_weights = {
6      "layer_1": {
7          "kernel": tf.Tensor(),
8          "bias": tf.Tensor(),
9      },
10     "layer_2": {
11         "kernel": tf.Tensor(),
12         "bias": tf.Tensor(),
13     },
14 }
```

Figure 6: A code snippet that shows an example state of the dictionary for storing the weights and biases of the objective neural network after initialization.

The example in this figure assumes a network with two layers where each has a kernel and a bias. However, this is just an example, and the implementation supports any

number of layers with any number of different weights and biases. We can now update the weights by reassigning a property of this nested dictionary with a tensor storing the new weights. However, there is one issue left. Namely, the weight properties inside the Keras layer do not yet carry the new tensors. To fix this, we overwrite the properties of the layers for storing the weights on our objective neural network with the new tensors inside our nested dictionary. We do this by overwriting another internal Keras function that Keras calls before each forward pass. For this, we use the same Python library as before. In this function, we first overwrite the weight properties of all the layers with the tensors inside our dictionary. After that, we execute the default behaviour of this function which executes the forward pass using our custom tensors as weights.

We now conclude the section where we talk about interesting implementation details. Apart from the issues described in this section, the remainder of the implementation is very straightforward. In the next section, we discuss the results of experimenting with the implementation from this section.

5 Experiments

This section is a collection of interesting results that we found when experimenting with the implementation described in the last section. The main goal of this section is to compare different configuration options. For a comparison of the learned optimizer against other state-of-the-art optimizers, we refer to the original work of Andrychowicz et al. [1]. They generally found that the learned optimizer performs better than other optimizers. As we later see, we get varying results when optimizing the loss function of a neural network with a learned optimizer. The learned optimizer outperforms other optimizers only in the best cases.

Since we include experimental results in most subsections, we first introduce the configuration options for our optimizer network in these experiments. Unless stated otherwise, we always use these options as a default. For the optimizer network, we use the coordinatewise network structure from section 3.3.2 with two LSTM layers each of size 20. The outputs of the last layer get added to form a scalar output for each coordinate[4]. We divide the timesteps for which we train the optimizer into slices of length T. We then update the optimizer parameters once for each slice by summing the objective function across all timesteps in each slice to get the loss function. Additionally, we set $T = 16$ in our experiments if not stated otherwise. T corresponds to the time horizon in the loss function 3.5. We explore why we use this optimizer update strategy and the choice of the time horizon in section 5.5.2. We also set the weights $w_t = 1$ for all timesteps in that loss function. In section 5.6, we explore different choices for these weights.

Next, we introduce the two objective functions used as benchmarks throughout this section.

[4]During the implementation, we first used a dense layer with a single output to attain a scalar output. This is also a viable option, although one has to be careful to not include a bias in this dense layer as the gradient with respect to this bias will be too large in comparison to the other gradients rendering the optimizer network unable to learn.

5.1 Example objective functions

We consider two objective functions in our experiments. The first one is a simple quadratic function. This objective function has a much smoother surface than the next objective function. The second objective function is the loss function of a neural network with a considerably rougher surface. We cover the details of these objective functions in the next two sections.

5.1.1 Quadratic function

The first objective function we consider is one of the most basic objective functions one could think of. It is a simple quadratic function that was also considered by Andrychowicz et al. [1]. We define this objective function as

$$f(\theta) = ||W\theta - y||_2^2. \tag{5.1}$$

Here, $W \in \mathbb{R}^{n \times n}$ and $y \in \mathbb{R}^n$ hold where n is an arbitrarily chosen positive integer. W and y are both sampled from a standard normal distribution upon initialization of the objective function. Unless stated otherwise, we set $n = 16$. We sample the initial θ from a standard normal distribution as well. Note that the minimum of this objective function cannot be smaller than zero due to it being the result of a euclidean norm. In our experiments, we first train the optimizer on different randomly sampled objective functions and initial parameters. In the tests that lead to the graphs representing the loss over time that we later see, we use the same objective function and initial parameters for each test in a single plot for better comparability. Additionally, we choose a learning rate of $\alpha = 0.001$. The output of the optimizer network gets scaled by this learning rate before being applied to the objective parameters. Note that we did not discuss this learning rate in the earlier sections of this thesis since it does not make a difference to the theoretical description of the approach. In practice, we have found that we achieve much better results using this learning rate. We, therefore, use it for all of the experiments. We usually optimize the quadratic function for 320 steps. We discuss this choice in section 5.2. Additionally, we do not use preprocessing on the gradients of the objective function, which we discuss in section 5.3.1. However, we do use preprocessing on the optimizer network gradients, which we discuss in section 5.3.2.

The area in which gradient-based optimization methods are most prominently used is for optimizing loss functions of neural networks. As a second example objective function, we, therefore, use the loss function of a simple neural network. We describe this neural network in the next section.

5.1.2 Neural network objective function

In this section, we introduce a neural network whose loss function we use as an objective function to test the approach of this thesis. We also refer to this network as the *objective network*. As a network structure, we choose a fully connected feed-forward network which we train on the MNIST dataset [12]. The MNIST dataset is a large dataset of

handwritten digits. It was introduced to be used as a benchmark to compare machine learning algorithms, and we, therefore, use it to test the learned optimizer. We use a cross-entropy loss function which is standard when training neural networks for classification tasks. We sample an objective function by randomly initializing the weights and biases θ and shuffling the MNIST dataset in each epoch. In general, we use batches of size 128 in our experiments. This is the same batch size that Andrychowicz et al. [1] use in their tests. We tried smaller batch sizes, but the loss fluctuates a lot, so we decided to use a batch size of 128 too. Additionally, we train and test the optimizer on a single epoch. The feed-forward network consists of one hidden layer with 32 neurons and one output layer with ten neurons. If not stated otherwise, we use the sigmoid activation function for both layers. We discuss the intricacies of different activations functions in sections 5.4.1 and 5.4.2. We use a learning rate of $\alpha = 0.01$ since we have observed that the learned optimizer performs much better using this learning rate. Lastly, we use preprocessing on the objective function gradients as well as the optimizer network gradients, which we discuss in sections 5.3.1 and 5.3.2 respectively.

During experiments with the implementation, we found that the attempt of learning the optimizer for a neural network fails in some cases. We have also observed that doing the same experiment with two different parameter initializations can lead to vastly different results. Therefore, when we display a graph of the loss of a neural network, we always use the best out of multiple attempts. In some cases, we explore the rate by which the learned optimizer works well to make it easier to compare different configuration options. In section 5.4 we explore these issues in more detail.

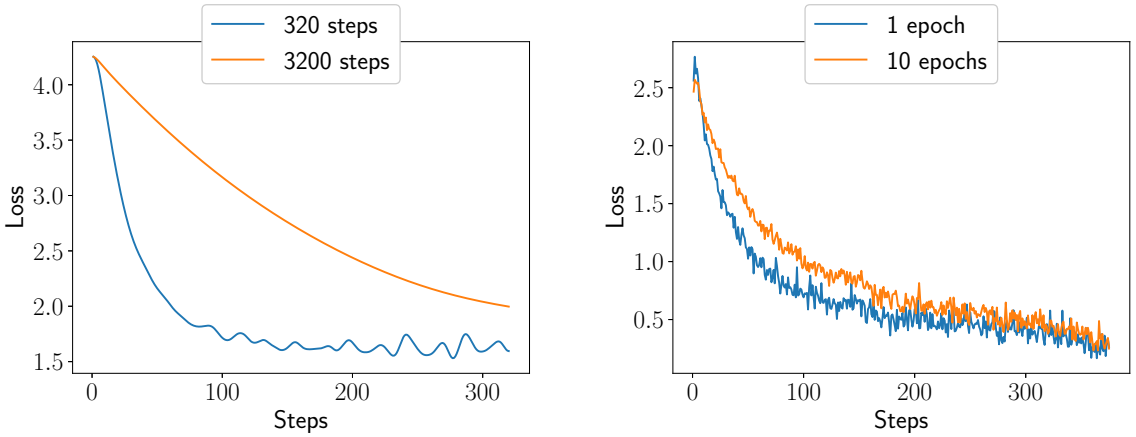
Note that we optimize both of these objective functions for a relatively small amount of timesteps. This is because training the optimizer for many timesteps is computationally more expensive. We discuss the choice of the number of timesteps for which we train the optimizer in the next section.

5.2 Choice of super epoch size

In this section, we discuss the choice of the number of timesteps for which we train the optimizer. We first introduce a definition. In the context of this thesis, a *super epoch* describes the number of timesteps of optimizing the objective function across which we train our optimizer. Naturally, we train the optimizer for multiple super epochs. A super epoch is different from an *epoch* in the machine learning context. One epoch usually describes going through the entire dataset once. The size of one super epoch is independent of the size of one epoch of optimizing the objective function. A super epoch could include multiple epochs, be equivalent to an epoch or only include parts of an epoch. Therefore, a super epoch to our optimizer network is similar to what an epoch is to a standard neural network.

We can observe our learned optimizer perform worse at the early steps of optimization if we train it on large super epochs. We can see this effect in figure 7.

We trained the optimizer for the quadratic objective function on a super epoch size of 320 and a super epoch size of 3200. We then tested both optimizers on a super epoch



(a) Quadratic objective function.

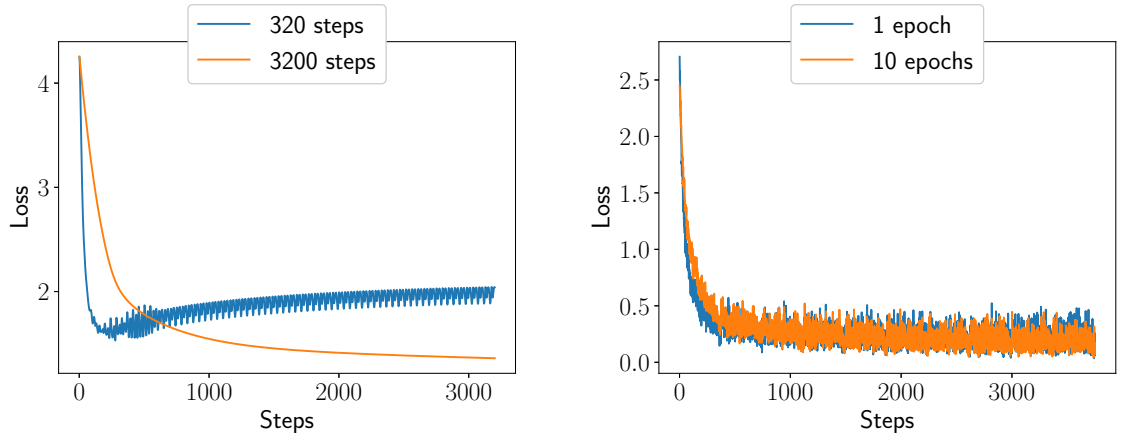
(b) Neural network objective function.

Figure 7: Objective function over time during optimization for a small number of steps using a learned optimizer that we trained on a large versus a small super epoch size.

size of 320. We discussed all other configuration options in section 5.1.1. We display the results in figure 7a. For the neural network objective function, we trained the optimizer on a super epoch size of one epoch and a super epoch size of ten epochs. We discussed all other configuration options in section 5.1.2. The results can be seen in figure 7b. Note that we also use preprocessing for both experiments, which we later discuss in section 5.3. For both objective functions, we can observe that if we train the optimizer on large super epochs, it performs worse on the earlier timesteps of optimization compared to an optimizer that was trained on smaller super epochs. This happens because the optimizer trained on a large super epoch size learns a smaller step size. This is due to the optimizer being trained much longer on the later timesteps where the objective is already close to the optimum. Therefore, a small step size is beneficial to not step out of the optimum. Next, we test how these four trained optimizers perform when being tested on a large super epoch size. For this, we test the optimizers for the quadratic objective function on a super epoch size of 3200 timesteps and the optimizers for the neural network objective function on a super epoch size of ten epochs. We show the results in figure 8.

We can observe that the optimizers trained on a large super epoch size perform better than the optimizers trained on a small super epoch size when testing them for a large number of steps. For the quadratic objective function, we can even observe the optimizer learned on a small super epoch size increasing the value of the objective function after we surpass the number of timesteps on which the optimizer was trained. This is most likely the case because it learns a larger step size. This large step size helps to minimize the objective faster at the start of optimization but makes the optimizer perform worse at the later steps of optimization where the objective is already close to the minimum.

All in all, it seems to be advisable to train the optimizer on the same number of timesteps that we later want to use it on. Due to the increasing time the training takes for larger super epochs, we continue to train and test the optimizer on smaller super



(a) Quadratic objective function.

(b) Neural network objective function.

Figure 8: Objective function over time during optimization for a large number of steps using a learned optimizer that we trained on a large versus a small super epoch size.

epochs. For the quadratic objective function, we stay with a super epoch size of 320, and for the neural network objective, we stay with a super epoch size of exactly one epoch. In our case, a single epoch has a total number of 375 timesteps.

5.3 Preprocessing the gradients

In this section, we talk about a means of preprocessing the gradients before using them for optimization. In the approach of this thesis, we deal with two different gradients. The first one is the gradient of the objective function with respect to the objective function parameters, which we use as input to the optimizer network. The second gradient is the gradient of the loss function described in formula 3.5 with respect to the optimizer parameters. We start by talking about preprocessing the gradient of the objective function.

5.3.1 Objective function gradients

Andrychowicz et al. [1] propose a logarithmic gradient preprocessing function to scale the coordinates of the input to the optimizer network to be of the same order of magnitude. Their reasoning to use this is because neural networks are better at working on input values that are roughly the same order of magnitude. In this section, we want to test how the learned optimizer performs with and without preprocessing the objective function gradients. We continue with giving the piecewise definition of this logarithmic function. After that, we compare how the learned optimizer performs with and without objective function gradient preprocessing.

The gradient preprocessing function proposed by Andrychowicz et al. [1] is defined as

$$\rho(\nabla_t) = \begin{cases} \frac{\ln(|\nabla_t|)}{p} \cdot \operatorname{sgn}(\nabla_t) & \text{if } |\nabla_t| \geq e^{-p} \\ -e^p \cdot \nabla_t & \text{else} \end{cases}. \tag{5.2}$$

As before, we abbreviate $\nabla_\theta f(\theta_t)$ with ∇_t. This stands for the gradient of the objective function and, therefore, the input to our preprocessing function. The parameter p has to be greater than zero. Andrychowicz et al. [1] choose $p = 10$ for all their experiments, so we also choose this for all our experiments. In the following, we show a plot of this function to get a better understanding of it.

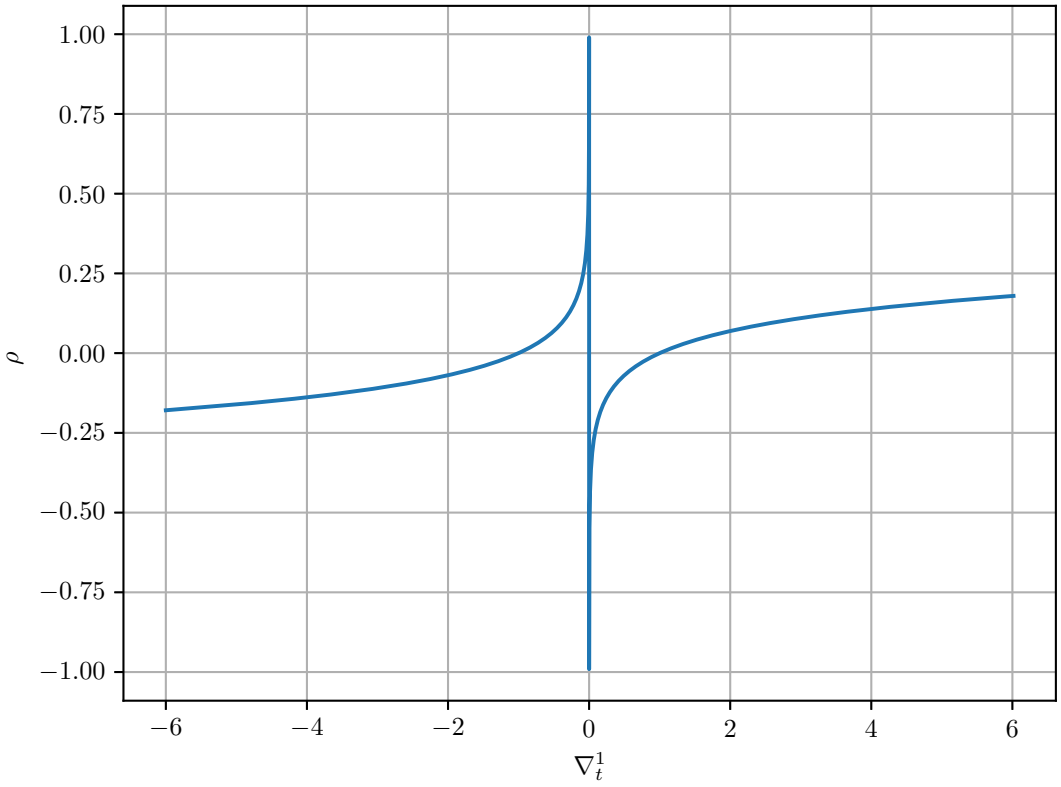
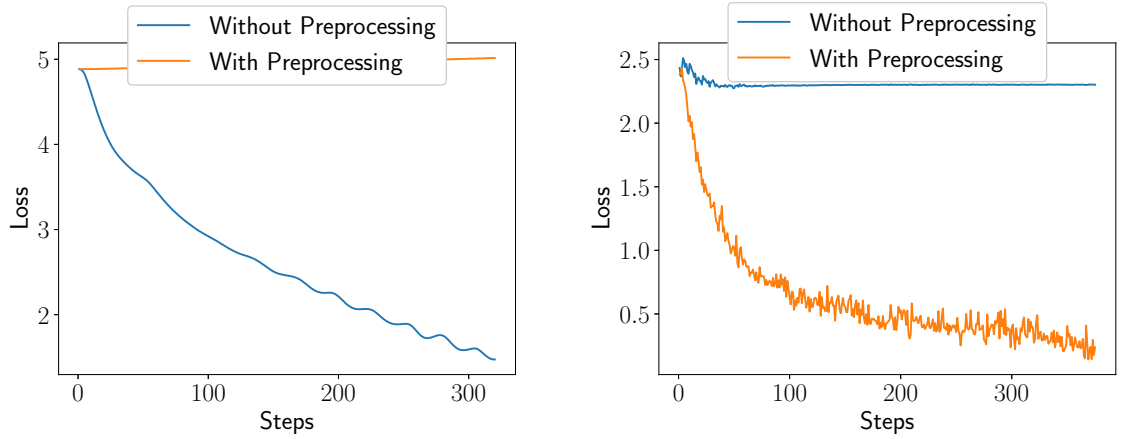


Figure 9: Plot of the objective function gradient preprocessing function described in equation 5.2 of a single coordinate using $p = 10$.

As one can see from figure 9, the preprocessing function ρ maps to values between one and minus one for sufficiently small input values. In general, this is a suitable input domain for neural networks and therefore serves our purpose well. One might think that there exists an issue since ρ is not bijective on all real numbers. This seems to be undesirable, as our optimizer network could then not tell apart different gradient values. Fortunately, the gradients which we input into this function are, especially for loss functions of neural networks, generally small in magnitude. In fact, for a domain

$1 > |\nabla_t| \geq e^{-p}$, this function is bijective, making it a good choice for gradients that fall into this domain. For $p = 10$, note that values with a magnitude roughly smaller than 10^{-5} fall outside this domain. Note that gradient coordinates that fall outside this domain have a very different optimal output than gradients that fall into this domain and map to the same value. For that reason, if the rate by which values fall outside this domain is sufficiently large during training, the optimizer tries to learn two different behaviours for the same input. Therefore, if the rate by which the gradient coordinates fall outside this domain is sufficiently small, we get a good performance using this preprocessing function. However, it is always possible that some gradient coordinates fall outside this domain. For those coordinates, the optimizer outputs a value that does not vary a lot in magnitude from other outputs. This is because a learned optimizer would interpret a gradient outside this domain as some gradient inside this domain. This behaviour is therefore comparable to gradient clipping [18] for gradients larger than one and roughly smaller than 10^{-5} in magnitude. By changing the parameter p, we can adjust the lower bound of this domain.

We can see how our learned optimizer performs with and without this gradient preprocessing function in figure 10.



(a) Quadratic objective function.

(b) Neural network objective function.

Figure 10: Objective function over time during optimization using a learned optimizer with and without preprocessing the objective function gradients.

As one can see from figure 10a, the learned optimizer does not work well with preprocessing for the quadratic function. An observation that we can make is that sufficiently many gradients are larger than one in magnitude and, therefore, lay outside the domain for which the preprocessing function is bijective. This makes it impossible for the optimizer to learn as it cannot differentiate between different gradients for sufficiently many cases. In the other experiments, we, therefore, do not use the preprocessing function for gradients of the quadratic objective function. Next, we can take a look at figure 10b. This figure shows how the learned optimizer performs with and without preprocessing on a loss function of a neural network as an objective function. As one can see, the

network does not seem to be able to learn without using the preprocessing function for the gradients of the objective function. It turns out that the loss converges to a value of around 2.3. Incidentally, this loss coincides with a prediction accuracy of about 10%. This is as good as guessing randomly because the MNIST dataset has ten classes. We discuss a possible explanation for why this happens in the following.

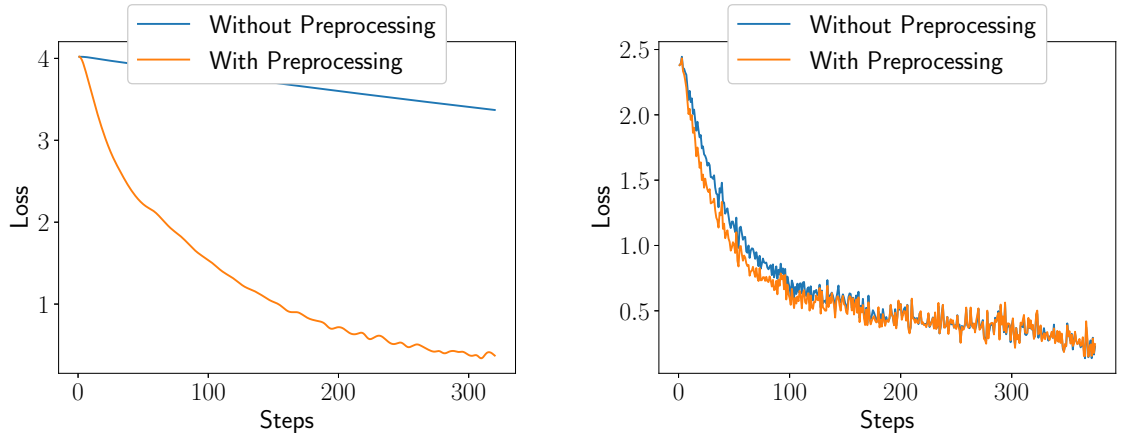
In general, gradients of loss functions with respect to parameters of layers that are closer to the output are larger in magnitude. The difference in magnitude comes from the fact that the gradients get computed using the chain rule. Therefore, multiple small values get multiplied to attain the gradient. For earlier layers, more small values get multiplied. Especially the bias of the output layer has much larger gradients than the other weights of the network. Since there is no preprocessing to approximately equalize the magnitude of the gradients, the optimizer will receive inputs that are drastically different in magnitude. Due to the random nature of the optimizer directly after initialization, the gradient of the bias of the output layer likely increases even more. This would make the gap in magnitude even larger. Due to this, for the optimizer network, it might seem that only the bias of the output layer influences the output. Since there is no connection to the inputs anymore, the best strategy is to assign the same probability to each class. We can observe this behaviour in the outputs of the objective network. Training the objective network to output the same probability for each class can be interpreted as a local minimum for our optimizer. We can observe that this sometimes also happens in other experiments with the neural network objective function. We discuss some of these occurrences in section 5.4.

We cannot definitively prove the reasoning in the above paragraph, but our observations coincide with the explanation. Due to the network not learning at all without preprocessing of the objective gradients, we continue to use preprocessing for all other experiments with a neural network objective function.

Next, we discuss gradient preprocessing for the gradients of the loss function with respect to the parameters of the optimizer network.

5.3.2 Optimizer network gradients

In this section, we talk about preprocessing the gradients of the loss function that we defined in equation 3.5 with respect to the parameters of the optimizer network. Especially for neural network objective functions, these gradients become very small in magnitude. In section 5.4.3 we discuss why this happens. We then also describe a case where these gradients become so small that some coordinates round to zero. Zero gradients make it hard for our optimizer network to learn. To counteract this issue, we normalize these gradients using the euclidean norm. This method uses the euclidean norm to scale the gradients such that each gradient vector has a length of one. Since the gradients are usually small in magnitude, euclidean normalization usually increases their magnitude. Note that when gradient coordinates round to zero, normalization does not help as scaling zero coordinates does nothing. In figure 11a we can see the results of training the optimizer with and without this normalization step. All other configuration options are the default configuration options that we introduced in sections 5.1.1 and 5.1.2 respectively.



(a) Quadratic objective function.

(b) Neural network objective function.

Figure 11: Objective function over time during optimization using a learned optimizer with and without normalizing the optimizer gradients.

As one can see, the learned optimizer does not work very well on the quadratic objective function if we do not normalize the gradients. We, therefore, use normalization on the optimizer gradients in all other experiments with the quadratic objective function. For the neural network loss function as an objective function, using normalization does not seem to make a difference. This is most likely the case because the gradients are large enough in magnitude for this simple two-layer objective network structure. Since normalization does not seem to have a negative impact on the results, we keep using it for the rest of the experiments with neural network objective functions in this section.

In the next section, we discuss issues that arise when optimizing the loss functions of neural networks using a learned optimizer.

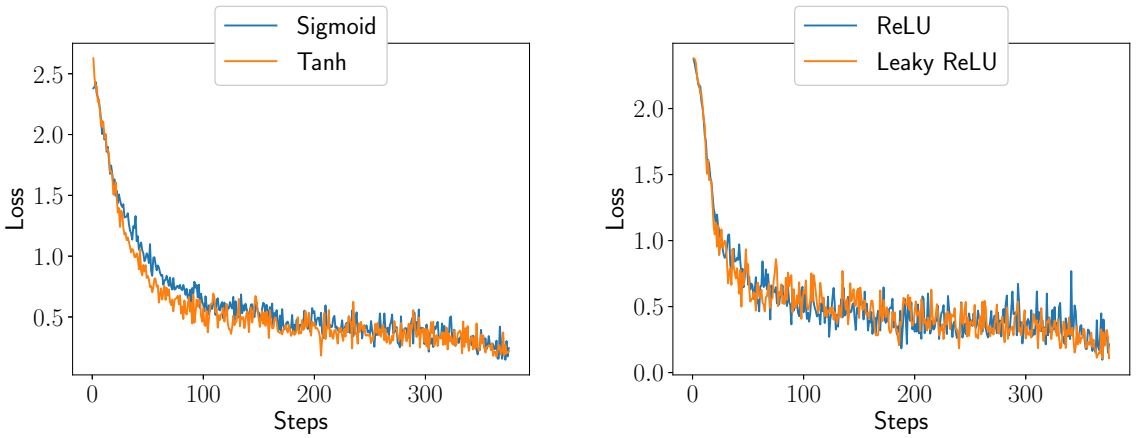
5.4 Issues with learning to minimize loss functions of neural networks

In this section, we talk about issues that occur when optimizing a loss function of a neural network using a learned optimizer. As already motivated, when doing experiments with the implementation, we observe that different initializations on the optimizer parameters can lead to vastly different results. In this section, we explore some cases where this happens and explore some hypotheses of why this happens.

5.4.1 Choice of activation functions in objective neural networks

First, we talk about the choice of activation function in our objective network. In all other experiments, we use a sigmoid activation function for both the hidden layer and the output layer of our objective network. In this section, we test the activation functions sigmoid, tanh, ReLU and leaky ReLU for the hidden layer with a softmax activation for the output layer. The other configuration options are the default options described in

section 5.1.2. Note that the leaky ReLU activation is the same as the ReLU activation in the positive domain. The difference is that it has a shallow negative slope in the negative domain. We use a slope of 0.1 in our experiments. We can see the results of these tests in figure 12. This figure shows the loss over time when using a learned optimizer to optimize the loss of the network structures introduced above.



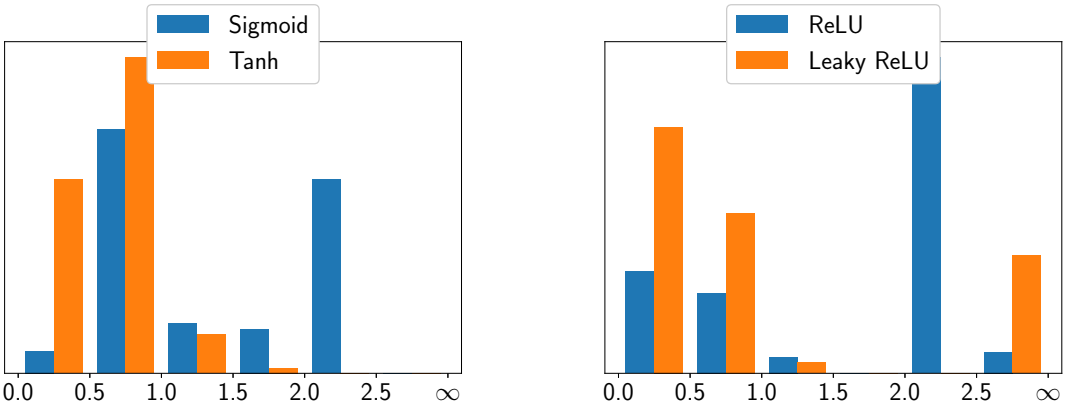
(a) Sigmoid and tanh activation functions. (b) ReLU and leaky ReLU activation functions.

Figure 12: Objective function over time during optimization using a learned optimizer. The objective neural network has different activation functions in the hidden layer.

As one can see, training an optimizer on all of these objective networks seems to work well. However, we ran the tests several times and picked the result with the smallest final loss. In figure 13 we, therefore, show the final losses of training the optimizer with different initializations of the optimizer parameters in a histogram. We used a smaller super epoch size and a smaller number of super epochs than we usually do during training to get these results. We did this to be able to run a significant amount of tests. This would not have been possible if we chose the same number of training steps for the optimizer as in the other experiments due to the much longer time this would have taken. We trained the optimizer for each objective network 100 times.

Note that a final loss of 2.3 coincides with an accuracy of 10%, which would be the accuracy of a random strategy. All final loss values below this indicate that the optimizer indeed learned to train the objective function to have a better than random strategy. From figure 13a, we can observe that the learned optimizer always learned to train the objective for a better than a random strategy for the tanh activation. We can also see that the optimizer usually works well when using an objective network with a sigmoid activation in the hidden layer and a softmax activation in the output layer. However, we can see a spike in the final losses of around 2.3. On the other hand, we can see from 13b that the ReLU activation does not work well in a lot of cases. In some cases, the optimizer even trained the objective network to have a final loss worse than picking randomly.

One hypothesis for why this happens is that this happens because of the negative part of the ReLU activation being zero. Due to the random nature of the optimizer at the start



(a) Sigmoid and tanh activation functions. (b) ReLU and leaky ReLU activation functions.

Figure 13: Distribution of the final loss values when training the optimizer multiple times with different initial parameters. We trained the optimizer for each objective network 100 times. The objective neural network has different activation functions in the hidden layer.

of training, in a lot of cases, we fall below zero in the inputs of the nodes of the hidden layer. The activation of those nodes then equates to zero. In that case, the gradients with respect to the weights of that layer also equal zero since the gradient of the ReLU activation is zero for the entire negative domain. If that happens, the optimizer network cannot learn the weights of the hidden layer. Therefore, it does not see a connection between the inputs and the outputs of the objective network. In that case, the best strategy is to output the same probability for each class. We can see that this happens a lot due to the spike of final loss values of around 2.3 for the ReLU activation.

Additionally, for the ReLU activation, we can observe that a few final losses are even larger and sometimes even go to infinity. These might be cases where the activation received large values, which, due to the ReLU activation not being bounded, result in large output values. These values might grow even larger due to large gradients and the optimizer making random steps at the start.

One idea to solve the problem of zero gradients when using the ReLU activation is by using a leaky ReLU activation instead of a ReLU activation. By doing this, the gradient of the negative part does not equal zero but some small constant which makes learning the network possible again. We observe that leaky ReLU does not have final losses in the range around 2.3, as can be seen in figure 13b. On the other hand, we again see cases where the loss is larger than 2.3, which might be due to the second reason explained above. This is because leaky ReLU, as well as ReLU, is unbounded.

All in all, we can observe that the rate by which the learned optimizer learns well depends heavily on the chosen activation function. We can observe two things that happen when the optimizer does not learn. The first one is that the final loss is around 2.3 corresponding which is as good as a random strategy, and the other is that the final

loss goes towards infinity. The final loss around 2.3 only happens if we use a sigmoid activation or a ReLU activation on the hidden layer. Incidentally, these two activations are the only ones of the four activations that go towards zero for small input values. This reinforces the reasoning described above. The final loss larger than 2.3 only happens for the ReLU and leaky ReLU activation in the hidden layer. These are the only two activation functions that are unbounded and go towards infinity for large input values. This reinforces the second reasoning that we gave above.

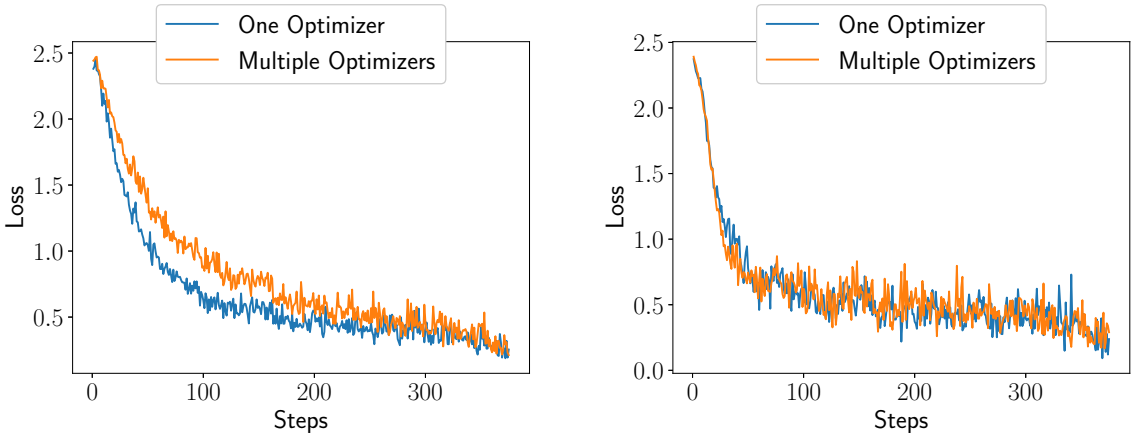
Another reason why the learned optimizer does not work well in some of the experiments in this section is that we use two different activations in the two layers. Andrychowicz et al. [1] show that the learned optimizer does not work well when being trained on one activation and tested on another. It may therefore be hard for the optimizer to learn the updates for two layers with different activations. We talk about a possible solution to this in the next section.

All in all, we cannot prove our reasonings above. However, our observations hint that there is a correlation between the probability that our learned optimizer works well and the activation function in our objective network. In general, activation functions should not go towards zero or infinity in any direction for the learned optimizer to perform well every time. We assume that a combination of the possible reasons we explained above is responsible for the learned optimizer sometimes not working well, especially with the ReLU activation function. We discuss a possible solution to the problem of drastically varying results for different initializations in section 5.4.4 where we use pretraining of the optimizer network.

5.4.2 Different layers in objective neural networks

In this section, we talk about what happens if we use different layer structures in our objective neural network. Andrychowicz et al. [1] found that it does not work to learn the optimizer for a network with multiple convolutional layers and one dense layer using a single coordinatewise LSTM network for all parameters. This seems to be the case because convolutional layers and dense layers are sufficiently different in their error surface. A solution that they propose is to use two different coordinatewise optimizers. One of these optimizers is responsible for the convolutional layers and one for the output layer. The optimizers still share their weights inside their respective layers and remain to have a different internal state in each coordinate. We try to use this approach on the examples from the last section. Namely, we use a single optimizer for each layer. This is similar to the approach by Wichrowska et al. [22]. They also use a different coordinatewise network for each layer as the lowest component of their hierarchical network structure. We test how this performs in comparison to using a single optimizer for all layers with some of the network structures from the last section. In particular, we choose to test a sigmoid activation and a ReLU activation in the hidden layer. In figure 14 we show the results.

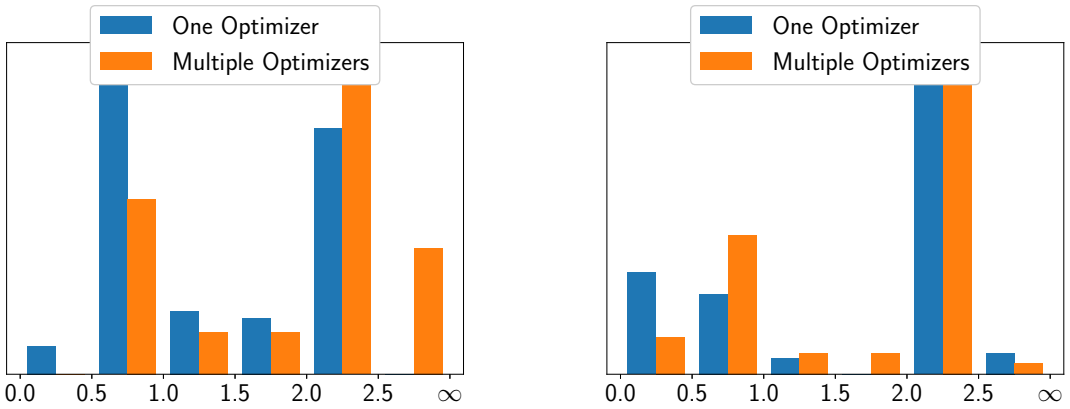
As one can see from figure 14b, for the ReLU activation function, this method of using multiple optimizers performs similarly well compared to using a single optimizer. For the sigmoid activation in figure 14a, it seems to perform slightly worse. However, the final loss values seem to be approximately the same for both methods. The results



(a) Sigmoid activation function. (b) ReLU activation function.

Figure 14: Objective function over time during optimization using a learned optimizer. We compare a single optimizer against a different optimizer per layer.

displayed are again the bests results that we got when starting the training with different initializations. We can, therefore, take a look at the distribution of the final values of the objective function when starting the training of the optimizer with different initial parameters. This distribution can be seen in figure 15.



(a) Sigmoid activation function. (b) ReLU activation function.

Figure 15: Distribution of the final loss values when training the optimizer multiple times with different initial parameters. We trained the optimizer for each objective network 100 times. We compare a single optimizer against a separate optimizer per layer.

We can observe that the distributions are very similar for both activations. For the sigmoid activation, multiple optimizers perform a little worse. An issue with using a separate optimizer per layer is that it is much slower than using a single optimizer. This is the case since TensorFlow now has to keep track of operations on twice as many optimizer

parameters. We also have to optimize both of our optimizers in each step, including computing the gradients. Due to the results not being better when using a separate optimizer per layer, we use a single optimizer for all other experiments. Nevertheless, this method is the only method that works if we use drastically different layer structures in a network, as shown by Andrychowicz et al. [1].

There is another issue with using a separate optimizer for each layer. This issue gets increasingly worse if we use more layers in our objective network. We explore this issue in the next section.

5.4.3 Vanishing gradients

In this section, we elaborate on an issue that occurs when using a different coordinatewise optimizer network for each layer.

In general, the gradient of the loss function with respect to the optimizer parameters is small in magnitude. We explain this intuitively as follows. The magnitude of the coordinates of a gradient of a function with respect to some parameters describes how much that function changes if we make a small change to the parameters. We can now think about how a small change in the parameters of the optimizer network influences the loss function. We defined the loss function by summing the objective function across multiple timesteps. Therefore, by using the linearity of the gradient, we can equivalently look at how much the optimizer parameters influence the objective function in each step. For this, we can follow the red path in the computational graph from figure 16.

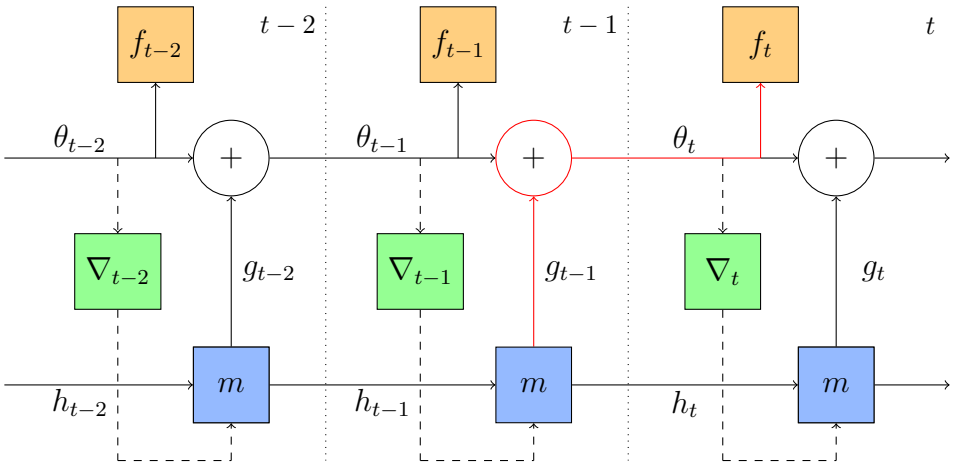


Figure 16: Computational graph representing multiple timesteps of optimizing an objective function f using the optimizer network m. This is the same as the computational graph from figure 4. We marked one path through which we need to perform backpropagation in red.

First, the parameters of the optimizer determine the output of the optimizer. Then this output gets added to the objective function parameters, which in turn get used to compute the objective function itself. When we now adjust the optimizer parameters

by a small amount, we can see that the optimizer's output also changes by a small amount. This output gets multiplied with the learning rate resulting in an even smaller change in the parameters of the objective function. Note that we do not show this in the computational graph, but we have already explained that we use this learning rate in practice. In summary, we can see that the optimizer parameters only have a small influence on the objective parameters which in turn have a small influence on the objective function. The influence of the optimizer parameters on the optimizer output corresponds to the magnitude of the coordinates of the gradient of the optimizer output with respect to the optimizer parameters. Similarly, the influence of the objective parameters on the objective function corresponds to the coordinates of the gradient of the objective function with respect to the objective parameters. These two gradients are multiplied due to the chain rule when computing the gradient of the objective function with respect to the optimizer parameters. We show this mathematically in the following.

Our objective is to compute the gradient of the loss function from equation 3.5 with respect to the parameters ϕ of the optimizer network m. We defined this loss by summing the objective f across some number of timesteps. Due to the linearity of the gradient, we can equivalently take a look at the gradient of the objective function at timestep t with respect to the parameters ϕ and then generalize to all timesteps. To do this, we first write

$$\frac{\partial f_t}{\partial \phi} = \frac{\partial}{\partial \phi} f_t(\theta_t) \tag{5.3}$$

$$= \frac{\partial}{\partial \phi} f_t(\theta_t(\phi)). \tag{5.4}$$

This holds since f_t is computed using the parameters θ_t. Now, θ_t depends on the parameters ϕ of the optimizer network m. We can also see these dependencies by looking at the red path in the computational graph from figure 16. We can now use the chain rule to derive

$$\frac{\partial f_t}{\partial \phi} = \frac{\partial f_t}{\partial \theta_t} \cdot \frac{\partial \theta_t}{\partial \phi}. \tag{5.5}$$

Now, using equation 3.6 we can write

$$\frac{\partial f_t}{\partial \phi} = \frac{\partial f_t}{\partial \theta_t} \cdot \left(\frac{\partial}{\partial \phi} \theta_{t-1} + \alpha \cdot g_{t-1} \right) \tag{5.6}$$

$$= \frac{\partial f_t}{\partial \theta_t} \cdot \left(\frac{\partial \theta_{t-1}}{\partial \phi} + \alpha \cdot \frac{\partial g_{t-1}}{\partial \phi} \right). \tag{5.7}$$

Note that we included the learning rate α that we use in practice as already motivated in this section. We can now recursively apply the last step until we reach the start of our optimization horizon:

$$\frac{\partial f_t}{\partial \phi} = \frac{\partial f_t}{\partial \theta_t} \cdot \left(\frac{\partial \theta_1}{\partial \phi} + \alpha \cdot \frac{\partial g_1}{\partial \phi} + \cdots + \alpha \cdot \frac{\partial g_{t-1}}{\partial \phi} \right). \tag{5.8}$$

Since the initial parameters θ_1 do not yet depend on ϕ, the derivative of those parameters with respect to ϕ is equal to zero. We can therefore simplify to attain

$$\frac{\partial f_t}{\partial \phi} = \frac{\partial f_t}{\partial \theta_t} \cdot \left(0 + \alpha \cdot \frac{\partial g_1}{\partial \phi} + \cdots + \alpha \cdot \frac{\partial g_{t-1}}{\partial \phi}\right) \tag{5.9}$$

$$= \alpha \cdot \frac{\partial f_t}{\partial \theta_t} \cdot \sum_{\tau=1}^{t-1} \frac{\partial g_\tau}{\partial \phi}. \tag{5.10}$$

As a small remark, it might seem like we did not consider the recurrent nature of our optimizer network in the derivation above. However, this is because we stopped the derivation after reaching the gradient of the optimizer output with respect to the optimizer parameters since we do not need to go further for our argument. The optimizer output now depends on the optimizer parameters and the internal state, which also depends on the optimizer parameters. The following steps in the derivation would, therefore, in essence, correspond to the backpropagation through time algorithm 2.1.1.

We have shown how to compute the gradient of the objective function with respect to the optimizer parameters. To do this, we need to multiply the gradient of the objective with respect to the objective parameters with the gradient of the optimizer output with respect to the optimizer parameters. The second factor in the last term of the derivation above corresponds to the gradient of the objective function with respect to the objective parameters. In the context of this section, it corresponds to the gradient of the loss function of a neural network with respect to its weights. This gradient is usually clearly smaller than one. The last term sums the gradients of the outputs of our optimizer network with respect to its parameters. These gradients are again usually clearly smaller than one. The number of elements in this sum is dependent on our time horizon T over which we sum our loss. This value cannot be too large. If it were, the process would be computationally infeasible. We discuss this in section 5.5. Thus, summing multiple gradients does not give a large increase in magnitude. One can see that this multiplicative nature of the gradient of the objective with respect to the optimizer parameters leads to it being very small in magnitude. Note that to gain the gradient of the entire loss function, we have to sum this term across all timesteps in our time horizon T. Since this time horizon is usually small, we do not get a large increase in magnitude.

This issue was not very important yet since the gradients were still large enough in magnitude for the optimizer to learn. However, if we use a different optimizer for each layer, this issue can prevent our optimizer from learning. If we use an optimizer per layer, the parameters of one of these optimizers only influence the parameters of a single layer in the objective network. Therefore, the optimizer parameters have even less influence on the objective function making the gradient even smaller. The problem increases for optimizers responsible for layers closer to the input of the objective neural network. In general, due to the chain rule, the gradient of the loss function with respect to weights of layers close to the input is smaller than the gradient with respect to weights of layers close to the output. In our experiments in the last section, the magnitude of the gradient with respect to the optimizer parameters was still sufficiently large. However, if we use objective networks with more layers, this becomes a real problem. For example, if we

use an objective network with two convolutional layers and a dense layer as the output layer, we can already observe the consequences of this issue. Namely, the gradients of the loss function with respect to the parameters of the optimizer responsible for the first convolutional layer get rounded to zero. This is not desirable as we cannot learn a network if the gradient equals zero. Incidentally, when we trained the optimizer with this configuration for many super epochs, we saw the entire optimizer slowly making some progress by only learning the optimizers for the last two layers. This behaviour is still very undesirable. One can imagine that the gradient gets even smaller when using objective networks with many more layers.

In the next section, we discuss a means to make the behaviour of our trained optimizer more predictable on neural network objective functions.

5.4.4 Pretraining the optimizer

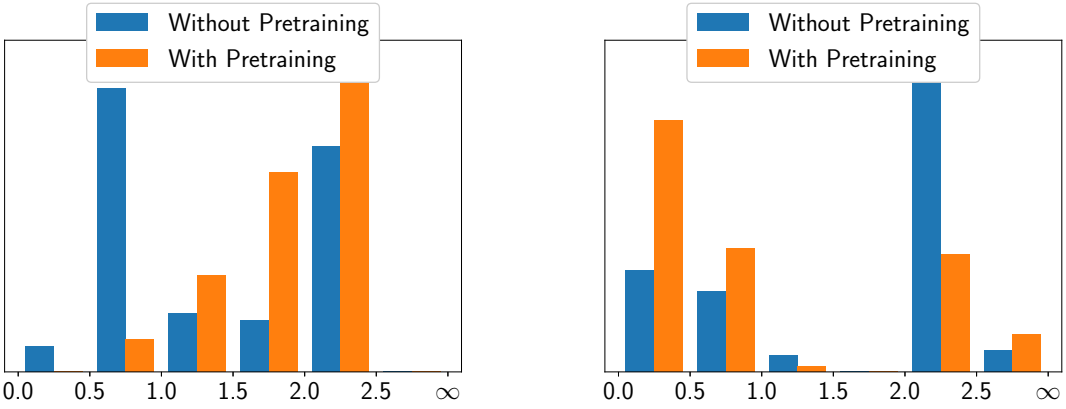
In this section, we discuss pretraining the optimizer network. Pretraining is an idea to solve the issue that the performance of a learned optimizer seems to be heavily dependent on the initial parameters. In this case, the main goal of pretraining is to level the performance of the optimizers with different initializations before the start of training. In general, when using pretraining, we use domain-specific knowledge to make the optimizer learn information that we already have before starting the actual learning process. We know that generally, the step should roughly point into the negative direction of the gradient. We can use this knowledge in our pretraining. We do this by training the optimizer network to output the negative input values times the learning rate. One advantage of the coordinatewise network structure is that we only need to train a single network since all networks responsible for the different coordinates share their weights.

However, there are some details we have to be careful about. First, we need to choose from which distribution we want to sample the pretraining dataset. We opt for a normal distribution with a mean of zero and a standard deviation of 0.001. We choose this distribution since the gradient of our neural network objective function usually falls into the same order of magnitude as the values sampled from that distribution. Secondly, we observed that our optimizer network learns the magnitude more so than the sign of the correct output when using a mean squared loss function or any other typical loss function. Since the sign of the output is much more relevant than the magnitude, we decided to use a different loss function that we define as

$$\mathcal{L}_p(\phi) = \frac{1}{n} \sum_{k=1}^{n} |g_k - g_k^*| \cdot \exp(-\omega \cdot \operatorname{sgn}(g_k \cdot g_k^*)) \tag{5.11}$$

Here, as before, ϕ corresponds to the optimizer parameters. g_k corresponds to the output of our optimizer in the k-th coordinate of our batch. g_k^* corresponds to the optimal output at that coordinate. The number of coordinates in a single batch is described by n. For the pretraining, we choose the same batch size as for the later training of 128. As one can see, this loss function takes the weighted average over the absolute value of the difference between the outputs and the optimal outputs. We discuss how these weights

look like in the following. Note that the sign function returns one if the input is positive and minus one if the input is negative. Therefore, if the sign of the output and the optimal output is the same, we get a positive sign. Otherwise, we get a negative sign. To gain the final weight, we input the inverted sign into an exponential function. In summary, if we have the same sign in the output and the optimal output, we input a negative value into the exponential function, which leads to a weight smaller than zero. In the same way, if we have different signs in the output and the optimal output, we get a weight larger than zero. Therefore, this loss function punishes different signs much more than different magnitudes. The difference between these weights can be controlled by the scalar $\omega > 0$. In our experiments we choose $\omega = 2$. We observed that the pretraining works much better if we use this custom loss function. In figure 17 we compare the distribution of the final objective function values using pretraining against not using pretraining.



(a) Sigmoid activation function.

(b) ReLU activation function.

Figure 17: Distribution of the final loss values when training the optimizer multiple times with different initial parameters. We trained the optimizer for each objective network 100 times. We compare optimizers trained with and without pretraining.

The objective functions are the same as in section 5.4.2. To get these distributions, we used slightly different configuration options during training. We used a smaller super epoch size and a smaller number of super epochs. We did this since the experiment is computationally expensive, and we ran the experiment 100 times to attain a meaningful distribution. As one can see from figure 17a, the pretraining made the learned optimizer perform much worse on a neural network with a sigmoid activation in the hidden layer. However, in figure 17b, we can see that it works much better than not using pretraining if we use an objective network with a ReLU activation in the hidden layer.

This concludes the section where we describe issues that arise when optimizing the loss function of a neural network using a learned optimizer. In general, we could see that the learned optimizer works well on simpler objective network structures. However, we also observed vastly different results when using different initial parameters. When trying to use the learned optimizer on more complicated network structures, we could

observe that issues appear that prevent the learned optimizer from performing well.

In the next section, we discuss different strategies for updating the parameters of the optimizer.

5.5 Learning strategy for the optimizer

In this section, we discuss different approaches to choosing the time horizon T for the loss function that we defined in equation 3.5. There are two possible options. The first option is to set T to the number of steps that we want to train the objective for. Using this approach, we would update the optimizer parameters once for each optimization process of the objective function. The other option is to choose T to be smaller than the number of steps that we want to optimize the objective for. If we choose the second option, there are two different approaches to meta-learn the optimizer. The first one is to divide the optimization steps into slices of size T and update the optimizer parameters once for each of these slices. The second approach is to keep a running history of the last T steps and update the optimizer parameters in each timestep of optimizing the objective function.

We first reintroduce the notion of a *super epoch* that we first introduced in section 5.2. In our implementation, we call one process of optimizing our objective function a super epoch. We use this term to make a distinction from an *epoch* which refers to optimizing over the entire dataset once. A super epoch could, for example, contain multiple epochs, be equivalent to an epoch or be a part of an epoch.

In the following subsection, we explore all three approaches introduced above.

5.5.1 Updating the optimizer parameters once every super epoch

The first approach to choosing the time horizon T is to set it equal to the total number of timesteps that we want to optimize our objective function for. This would imply that we sum the objective function across the entire optimization horizon to gain the loss function for the optimizer. Therefore, we could only update the optimizer parameters once for every optimization process of the objective function, making this approach computationally expensive.

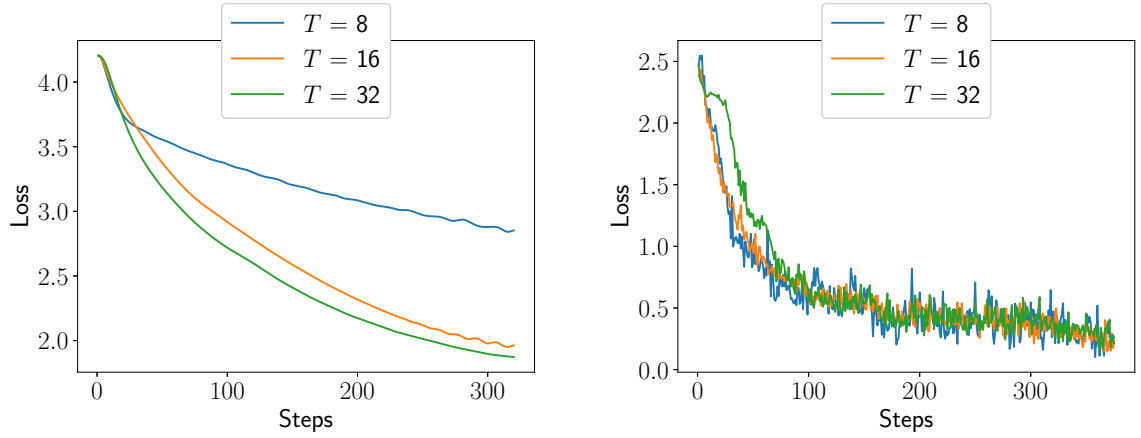
There are a few other issues with the approach of updating the optimizer once every super epoch. One of these issues is that we would have to know the exact number of steps for which we want to optimize our objective function in advance. This is not necessarily possible as we cannot say after how many timesteps our objective function reaches its optimum. Fortunately, Andrychowicz et al. [1] show that a learned optimizer trained only on the first 100 optimization steps generalizes well to the next 100 steps. Yet another issue arises when implementing this approach. If we choose a large super epoch and therefore a large time horizon we would have to sum the objective function across all timesteps to attain the loss function for the optimizer. Additionally, TensorFlow would have to keep track of all operations that influence the loss function. This includes all forward passes through the optimizer network in each timestep. For a large time horizon T, this can lead to a memory overflow rendering this approach computationally infeasible.

Due to these issues, this is not an approach that we propose to use. When working with a small time horizon or an objective function with a small number of parameters, then this approach might be a reasonable option. If we want to train the optimizer on more complicated objective functions and on larger time horizons, we have to employ another strategy which we explore in the following section.

5.5.2 Updating the optimizer parameters every T steps

In this section, we introduce the approach of setting the time horizon T to be smaller than the size of our super epoch. This leads to two options for updating the optimizer. The first option is to divide the timesteps in our super epoch into slices with the length of our time horizon T. For each slice, we then sum the objective function across all timesteps inside that slice and update the optimizer once using the respective loss function. This approach has a few advantages to the approach in the last section, where we set T to the size of our super epoch. We discuss this approach in this section.

The first advantage is that we can update the optimizer multiple times during a single super epoch. The number of updates depends on the time horizon T. This makes the entire learning process much more computationally feasible since we can change the optimizer parameters much faster to make them converge on their optimum. The number of times that we update the optimizer in each super epoch corresponds to the factor by how much faster we train the optimizer. Note that this might not entirely be true as we use far fewer timesteps in each optimization step of the optimizer. Therefore, using all timesteps of the entire super epoch as proposed in the last section would lead to a more educated update of the optimizer parameters. In practice, we have observed that even for time horizons that are significantly smaller than the number of optimization steps per super epoch, the updates to the optimizer parameters are of high quality. In figure 18 we can see the results of using different time horizons.



(a) Quadratic objective function.

(b) Neural network objective function.

Figure 18: Objective function over time during optimization using a learned optimizer. We trained the optimizer with different time horizons T.

As one can see in figure 18a, the learned optimizer for a quadratic objective function trained on a time horizon of $T = 8$ performs worse than with a time horizon of $T = 16$ or $T = 32$. On the other hand, the difference between a time horizon of $T = 16$ and $T = 32$ is insignificant. For the objective network trained on the MNIST dataset, we can see that all choices of T seem to lead to similarly performing optimizers. We have observed that the runtime of our implementation grows very fast when increasing T. Since there does not seem to be an advantage of using a time horizon of $T = 32$ instead of using a time horizon of $T = 16$, we use a time horizon of $T = 16$ for all other experiments in this thesis.

Another advantage of this approach is that it is the only approach that is computationally feasible for large super epoch sizes. In the last section, we explained why the approach of setting T to the size of our super epoch is computationally infeasible. The reason for this is that TensorFlow has to keep track of too many operations, which can result in a memory overflow very quickly. Using the approach of this section, we do not have these issues as we only have to keep track of operations of at most T timesteps. Another advantage is that we can adjust the time horizon T to be suitable for the current hardware that the implementation is run on.

The approach in this section is similar to how *mini-batch gradient descent* [20] works. In mini-batch gradient descent, we divide the entire dataset into batches and then update the neural network parameters once for each batch. This usually provides high-quality updates despite not taking into account the entire dataset. It is also more efficient than using the entire batch for each update as it produces multiple updates for each epoch.

5.5.3 Updating the optimizer parameters in each step

In this section, we again set the time horizon T to be smaller than the size of all optimization steps of the objective function. In contrast to the last section, instead of summing the objective function across slices of T timesteps and updating the parameters of the optimizer network once for each of those slices, we now sum the objective function across the last T timesteps and update the optimizer once in each timestep.

By updating the optimizer in each timestep, we get a lot more updates of the optimizer parameters compared to the other two approaches in this section. This leads to our optimizer network learning faster than with the other methods. However, the updates over one super epoch cumulatively use the same information as the approaches in the last two sections. What this approach does is that it uses the objective function in each timestep at most T times inside the loss function instead of using it once as in the last two sections. Therefore, in theory, we should not find a better optimum as we do not use more information.

In practice, there is an issue with this approach. In an implementation, we would need to keep track of a moving window over the last T objective function values. As soon as we generate a new objective function value, we can drop the value which has been a member of our window the longest. Therefore, this resembles a FIFO queue. TensorFlow has to store all operations which lead to each objective function value to be able to compute the gradients. However, we do not need to compute the gradient of an objective function

value that we dropped from the FIFO queue. Therefore, TensorFlow should not need to keep track of the operations, which lead to the values that were removed from the queue. Unfortunately, there is no way of telling TensorFlow to stop tracking operations that lead to a specific value without initializing a new tracking context. However, initializing a new tracking context does not work since we cannot add operations from the past to this new tracking context. Initializing a new tracking context would also remove all other objective function values which should remain inside our FIFO queue. Therefore, the only straightforward way to implement the approach of this section is to keep track of all operations during an entire super epoch. Unfortunately, this runs into the same issue that we already covered in section 5.5.1, where this gets computationally infeasible for large time horizons T.

Due to this, we suggest not using this approach in practice. In all of the experiments in this thesis, we, therefore, use the approach of updating the optimizer once for every T timesteps that we explained in section 5.5.2. This is also the approach that Andrychowicz et al. [1] use.

In the next section, we cover another detail that we can adjust inside the loss function 3.5. Namely, we discuss strategies to set the weights w_t in this loss function.

5.6 Choice of weights in the loss function of the optimizer

In this section, we explore different strategies of setting the weights w_t in the loss function that we described in equation 3.5.

Up until now, we have used $w_t = 1$ in all of the experiments in this section. This is the same weights that Andrychowicz et al. [1] use. An idea to improve this might be to weight objective function values at the end heavier than objective function values at the start. This is desirable since we ultimately want the objective function at the last timestep to be minimal. One idea to do this would be to linearly increase the weights. We could then describe the weights as a function of the timestep as follows:

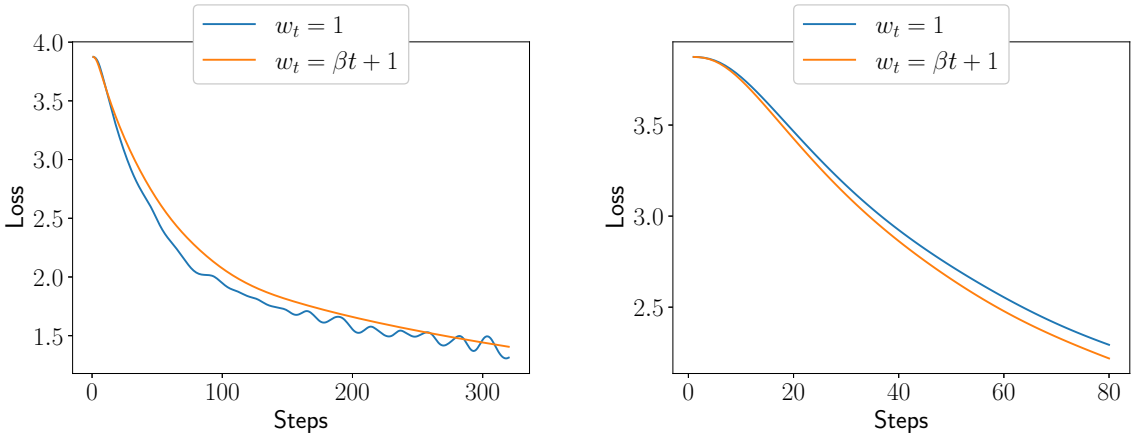
$$w_t = \beta t + 1. \tag{5.12}$$

Here, $\beta > 0$ corresponds to an arbitrary constant. Note that by adding a one, we make our weights not scale the objective function value down. We do this since we already have issues with small gradients, and we do not want to make them even smaller. Another idea would be to use exponential scaling to weight later entries even more heavily. In that case, we could define w_t as

$$w_t = \gamma^t. \tag{5.13}$$

Here, $\gamma > 1$ again corresponds to an arbitrary constant. However, there is one issue with this. These weights grow exponentially, and if we choose a large time horizon T, the difference of later objective values to earlier objective values would be way too large. In our tests, we, therefore, stick with the first option of scaling the weights linearly over time.

Note that the apparent usefulness of using increasing weights depends on the choice of the optimizer update strategy, which we discussed in section 5.5. As discussed in that section, we use the approach described in section 5.5.2, which works by updating the optimizer parameters once for every T timesteps. If we use increasing weights on these slices, we can see that additionally to the final value of the objective function, we also weight seemingly random intermediate objective function values higher than other values. Nevertheless, we show how the weighting approach compares to each other in the following. If we instead use the update strategy discussed in section 5.5.1, where we update the optimizer parameters once for every super epoch, then we achieve the goal that we had of weighting the timesteps in a monotonically increasing manner. We, therefore, also test increasing weights with that approach. The last update strategy that we discussed in section 5.5.3 was to sum the last T objective function values in each step. As one can see, weighting the timesteps differently does not serve a real purpose using that strategy since across an entire super epoch, the mean weight of each timestep would be the same. In figure 19 we therefore only test the first two approaches.



(a) Update the optimizer every T timesteps.

(b) Update the optimizer once every super epoch.

Figure 19: Objective function over time during optimization using a learned optimizer. We trained the optimizer on loss functions with different weighting strategies.

Note that we only test the approach on the quadratic objective function. We do this because using the approach of summing the objective functions across the entire super epoch is very computationally expensive. Additionally, for the experiments where we need to sum the objective across the entire super epoch, we choose a super epoch size of only 80 timesteps compared to the 320 timesteps for the approach of updating the optimizer every T timesteps. Note that we chose $\beta = 1$ in our experiments.

As one can see from figure 19a, using the strategy of updating the optimizer parameters once every T timesteps, we cannot see a significant performance difference between weighting the objective functions constantly or in a linearly increasing manner during training of the optimizer. In figure 19b, we can see that when updating the optimizer

once per super epoch, increasing weights perform very slightly better, but this may be the result of happenstance.

This now concludes the section where we do experiments with the approach of this thesis to see how well it performs and what issues arise in practice. We conclude this thesis in the next section.

6 Conclusion

This is the final section of this thesis. First, we summarize the most important parts of this thesis. After that, we make some concluding remarks about the approach of this thesis.

First, we theoretically described the approach of this thesis of meta-learning the optimizer for gradient-based optimization that was first introduced by Andrychowicz et al. [1]. This approach works by learning the optimal update step with a recurrent neural network. The recurrent neural network is trained by performing gradient descent on its weights using the objective function over multiple timesteps as the loss function. We then described issues that arise when trying to implement this approach in Python using state-of-the-art machine learning frameworks. The biggest issue is that it is impossible to track operations through variable updates. We solved this issue by storing the parameters of the objective function inside custom TensorFlow tensors. Lastly, we did experiments using the implementation. We tried different configuration options to see what works best when using the approach of meta-learning the optimizer in practice. We found that the preprocessing function for the gradients of the objective function proposed by Andrychowicz et al. [1] only makes sense for gradients that fall sufficiently often into the domain for which the preprocessing function is bijective.

Next, we found that learning the optimizer for optimizing a loss function of a neural network has some issues. We observed that the success of learning the optimizer depends heavily on its initial parameters. The success also depends on the activation function used inside the objective network. We observed that the ReLU activation function performs worst out of all activation functions that we tried. We, therefore and due to other observations, made the hypothesis that activation functions that neither converge to zero nor diverge to infinity in any direction work much better than activation functions that do. If we use an activation function that either converges towards zero or diverges, the gradients seem to sometimes get too small or too large for the optimizer to start learning properly. We tried solving these issues first by using a different optimizer for each layer. This approach did not yield the desired results but might be the only possible approach if the objective network structure gets too complicated. After that, we tried pretraining the optimizer to equalize the magnitude of the updates that the optimizer proposes at the start of training. Unfortunately, we did not get the desired result of substantially improving the success rate of learning the optimizer for all objective functions. However, we saw pretraining improve the success rate for the objective network with a ReLU activation function on the one hand but worsen it for the objective network with a sigmoid activation function on the other.

We also found a general problem of vanishing gradients of the loss function with respect to the optimizer parameters. If we use one optimizer per layer, this issue becomes so bad that some coordinates of the gradient get rounded to zero. It increases even more for objective network structures with a lot of layers. Unfortunately, as discovered by Andrychowicz et al. [1], we have to use a different optimizer per layer if we use an objective network structure with structurally different layers. Additionally, we discussed different update strategies for the optimizer parameters. We concluded that for large super epoch sizes, the approach of dividing the super epoch into slices and updating the optimizer parameters once for each slice is the only viable option. We also tried different weighting strategies for the timesteps inside the loss function, but we did not get any relevant performance improvement when using a more sophisticated weighting strategy. In summary, we have observed that the approach of learning the optimizer works very well for simple quadratic functions but starts to perform worse the more complex the objective function becomes.

One motivation for the approach of this thesis is to not have to design the optimal optimization algorithm for a given problem by hand. This means that we want to avoid needing to pick the optimizer and adjust its hyperparameters to work best for a given problem. Unfortunately, there are still a lot of hyperparameters that we can tune to make the learned optimizer perform better. These hyperparameters, for example, include configuration options that we discussed in this thesis, like the update strategy of the optimizer parameters or the weights of the timesteps of the objective function inside the loss function. They also include choosing a network structure for the optimizer as well as a learning rate.

In conclusion, there are likely too many issues with this approach for using it in practice. However, if one would be able to solve the major issues, then the approach seems to be very promising for the future of meta-learning and generalizing gradient-based optimization algorithms to different objective functions. One can imagine a variant of this algorithm being used inside a machine learning model, which can quickly learn unknown tasks due to the optimizer being able to generalize to every task. Unfortunately, this is likely impossible with the current state of this method, and there is a lot of work to be done to make this happen. One approach by Wichrowska et al. [22] tries to build upon the approach of this thesis. They use a hierarchical network structure for the optimizer to exchange information between the different coordinates. This is not possible with the approach of this thesis due to the coordinatewise network structure of the optimizer. They also train the optimizer on multiple objective classes and achieve better generalization between different objective classes. For future research, this might be an interesting starting point to explore.

References

[1] Marcin Andrychowicz et al. "Learning to learn by gradient descent by gradient descent". In: *NIPS*. 2016.

[2] Yutian Chen et al. "Learning to learn without gradient descent by gradient descent". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 748–756.

[3] François Chollet et al. *Keras*. https://keras.io. 2015.

[4] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-agnostic meta-learning for fast adaptation of deep networks". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 1126–1135.

[5] Robert Hecht-Nielsen. "Theory of the backpropagation neural network". In: *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[6] Sepp Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[7] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". In: *Diploma, Technische Universität München* 91.1 (1991).

[8] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[9] Yuxiu Hua et al. "Deep learning with long short-term memory for time series prediction". In: *IEEE Communications Magazine* 57.6 (2019), pp. 114–119.

[10] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[11] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[12] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[13] Claude Lemaréchal. "Cauchy and the gradient method". In: *Doc Math Extra* 251.254 (2012), p. 10.

[14] Ke Li and Jitendra Malik. "Learning to optimize". In: *arXiv preprint arXiv:1606.01885* (2016).

[15] Ke Li and Jitendra Malik. "Learning to optimize neural nets". In: *arXiv preprint arXiv:1703.00441* (2017).

[16] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[17] Yurii E Nesterov. “A method for solving the convex programming problem with convergence rate O (1/k^ 2)”. In: *Dokl. akad. nauk Sssr*. Vol. 269. 1983, pp. 543–547.

[18] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *International conference on machine learning*. PMLR. 2013, pp. 1310–1318.

[19] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[20] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[21] Paul J Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560.

[22] Olga Wichrowska et al. “Learned optimizers that scale and generalize”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 3751–3760.