A thesis submitted in partial fulfilment for the degree of Master of Science in the science of Artificial Intelligence

Combining adaptive-computation-time and learning-to-learn approaches for optimizing loss functions of base-learners

by

Jörg Sander
(10881530)

supervised by Patrick Putzky & Matthias Reisser

November 10, 2017
Abstract

Learning to learn approaches can be used to train a recurrent neural network (RNN) that learns an optimization algorithm. Adaptive computation time for RNNs allows a network to adjust its depth at each time step to the input received so far. In this work we developed two adaptive meta-learners that combine the learning to learn and adaptive computation time approaches in order to optimize loss functions of base-learners. Integrating both approaches is motivated by the idea that a learned, iterative optimization algorithm benefits from being able to adjust the number of optimization steps to the input, by learning to weight the time step losses. Our newly proposed models could be trained with less computational effort on convex and non-convex optimization tasks compared to the baseline optimizer introduced by Andrychowicz et al. (2016). Moreover the adaptive optimizers developed their own training regime that trades off computational effort against accuracy based on a prior injected preference.
Acknowledgement

I remember vividly when Patrick presented the possible master thesis projects to me that he had been thinking about. I kept asking and suddenly he came up with this paper from Alex Graves about adaptive computation time for recurrent neural networks that would somehow offer possibilities to be combined with the learning to learn approach from Andrychowicz. It is always easy for us humans to explain things by hindsight, after we have collected our experiences, but I genuinely remember that I was instantly excited. So thanks Patrick for coming up with such an interesting subject. It was probably a little bit too difficult for me in the first place and luckily Patrick had a good dose of patience with me (or may be not showing his impatience). It was a good decision that Patrick asked Matthias to join our team before he himself returned to Germany and became a father for the first time. Matthias was able to help me challenging Patrick’s mind and through this increased interaction I got a better grip on the subject. May be interesting to note for others, although we are all three native German speakers, I decided at a certain moment that my mind would benefit if we would talk English when discussing the work, my wiggling thought vector representations just would not easily decode into German and only slightly better into English. Anyway, nearly redundant to say that without the tremendous help of Matthias and Patrick it would have been very hard to finish this thesis in a good way I guess. Thanks a lot for supporting me!

For the rest, working on the thesis was pretty lonesome I have to say. Luckily I had my dear fellow students, Maartje, Maurits and Thijs. It was really a great pleasure to have you around and I am somehow grateful for the beautiful moments we had together. The collaboration during the study was very important for me and I learned a lot from all the three of you.

Without any doubt I received the greatest support from my wife Manja, the mental thesis coach. I was lucky that she did not travel that often in the last few months and worked mostly from home. Actually we enjoyed it a lot being at home the two of us, nursing ourself with great food and most importantly sharing our relentless sense of humor.

Rest me to say "sorry" to my mom for not having a lot of time to distract her from the terrible sickness she has to cope with on a daily base. I know she does not want to hear that. Without dad taking so good care of her I would have had more difficulties in spending so much time for myself.

Jörg Sander
November 2017
Den Haag, The Netherlands
# Contents

1 Introduction
1.1 Research questions .............................................. 3

2 Background
2.1 Preliminaries .................................................... 5
2.1.1 Terminology .................................................. 5
2.1.2 Learning an iterative optimization algorithm ............... 6
2.1.3 Meta learning .................................................. 7
2.1.4 Stick-breaking process ....................................... 8
2.1.5 Truncated Backpropagation Through Time .................. 9
2.2 LSTM optimizer (baseline model) ............................... 10
2.3 Adaptive Computation time for RNNs .......................... 11

3 Method
3.1 General idea and assumptions ................................... 14
3.2 Computation time adaptive meta-learner ........................ 15
3.3 Bayesian approach to computation time adaptive meta-learning ......... 17
3.3.1 Preliminaries .................................................. 17
3.3.2 Deriving an evidence lower bound for $p_{\phi}(x^{(n)}|T)$ ........ 18
3.3.3 Generating the trajectory weights ............................ 19
3.3.4 Sampling horizons .......................................... 21
3.4 Differences between M-ACT and M-PACT model ............... 22
3.5 Evaluation of models .......................................... 22

4 Related work ..........................................................................

5 Experimental results
5.1 General ............................................................. 26
5.2 Optimizing regression functions .................................... 27
5.2.1 Linear regression .............................................. 27
5.2.2 Evaluating the LSTM optimizer on convex regression task .... 27
5.2.3 Non-convex linear regression .................................. 30
5.2.4 Evaluating the LSTM optimizer on non-convex regression task .... 30
5.2.5 Conclusions .................................................... 31
5.3 Evaluating M-ACT and M-PACT model on linear regression tasks .... 33
5.3.1 Results on convex linear regression .......................... 34
5.3.2 Conclusions .................................................... 38
5.3.3 Results on non-convex linear regression ....................... 38
5.3.4 Conclusions .................................................... 42
5.4 Optimizing a small neural network on MNIST ..................... 44
5.4.1 Results .......................................................... 47
5.4.2 Conclusions .................................................... 49
1 | Introduction

An optimization algorithm is an essential and widely used tool in Machine Learning and other scientific disciplines in order to find the optima of an objective function. In this work we are focusing on the goal of finding a global minimum of a task specific objective function $f$:

$$\theta^* = \arg\min_{\theta \in \Theta} f(\theta),$$  \hspace{1cm} (1.1)

where $\theta$ denotes the parameters of the objective function $f$ and $\Theta \in \mathbb{R}^d$ is the continuous search space of interest (where $d \in \mathbb{N}$). We make the assumptions that the objective functions are differentiable and can be queried at arbitrary locations in $\Theta$.

In the setting of this work we are using objective functions that correspond to loss functions of so called base-learners which can in principle be any supervised learning system e.g. a regression model, a Support Vector Machine (SVM) or a simple Neural Network (NN). But instead of designing a new optimization algorithm or using hand-engineered optimizers like Adagrad [4], RMSprop [30] or ADAM [16] which are well established in the deep learning community, our work will lean on and extend earlier research, known as learning to learn (L2L) or meta-learning [1], [21], [19], in which the optimization algorithm is learned or in other words in which the product of (meta) learning is an algorithm that can be used for training base-learners on supervised learning tasks.

The work of Andrychowicz et al. [1] has shown that a learned optimization algorithm outperforms hand-engineered optimizers on tasks like solving quadratic functions and optimizing neural networks performing image classification on MNIST and CIFAR-10. Figure 1.1 visualizes the superior optimization performance of the meta-learner of [1] on solving a bivariate quadratic function. Clearly the convergence rate of the learned optimizer in [1], which we refer to as LSTM\textsuperscript{1} optimizer hereafter, is significantly faster if compared to the hand-engineered algorithm and this is without any doubt an important contribution of the work.

The research we present here focuses on the training procedure of the meta-learner itself and is motivated by the question whether it is possible to speed up the learning process of the learned optimizer without impeding the final performance of the model. The experiments conducted in [1] trained the optimizer by applying a fixed number of optimization steps (which we also refer to

\textsuperscript{1}Long Short-Term Memory
as fixed horizon) to all base-learners. We believe that it is interesting to investigate whether the meta-learner can learn itself which horizon length is sufficient for the optimization problem the base-learner is trained on. This could result in faster training of the model. The idea can be pushed one step further by scrutinizing the question whether the meta-learner can gather information about the objective functions that can be used to distinguish between specific functions. Such information could then be exploited to treat functions differently during the optimization trajectory i.e. to make the optimization trajectory input dependent. For example one could imagine that a function $f^{(1)}$ with an initial loss value $f^{(1)}(\theta^{(1)}_{t=0})$ is significantly closer to its global minimum $f^{(1)}(\theta^{*(1)})$ (where $\theta^*$ denotes the parameter values at the global minimum) than another function $f^{(2)}$

$$f^{(1)}(\theta^{(1)}_{t=0}) - f^{(1)}(\theta^{*(1)}) \ll f^{(2)}(\theta^{(2)}_{t=0}) - f^{(2)}(\theta^{*(2)}).$$  

(1.2)

In this work we developed a method that encourages the learned optimizer to more explicitly extract such information from the input functions by giving the model an opportunity to exploit the information during the optimization process.

The underlying idea was recently introduced in the adaptive computation time (ACT) approach for recurrent neural networks (RNN) by Alex Graves [8]. The primary idea of ACT for RNNs is to allow the network to adjust its depth (i.e. the number of hidden layers) at each time step to the input received so far, where normally the depth of an RNN is fixed. In its most ideal form the learned input dependency would let to learning systems that are able to dynamically adapt the amount of computation they employ to the complexity of the task they perform [8]. As will be outlined in more detail in section 2.3, an RNN using ACT is augmented with a so called sigmoidal halting unit that computes a halting probability associated with each time step\(^2\). The approach is inspired by the Neural Stack Architecture presented in [10] and reconciles with the so called Unrolled Iterative Estimation View (UIEV) developed in [11]. The UIEV augments the idea that increased network depth equates to an increased abstraction level (also referred to as representation view) with the viewpoint that within a functional block\(^3\) estimated features are iteratively refined and often earlier layers within a block contribute more than subsequent layers.

Inspired by both approaches we think that it is intuitive and beneficial to extend the learning to learn approach with the adaptive computation time approach. Our primary idea is simple, we endow the optimizer with the ability to adjust the number of optimization steps for each objective function during training instead of applying a fixed horizon to all functions. We surmise that a model which learns an iterative optimization algorithm could profit from this additional flexibility because it might be an advantage if the optimizer can explore shorter sequences in the beginning of training and extend to more difficult i.e. longer inference task when having mastered the shorter trajectories to perfection. We hypothesize that this could reduce the computational time and effort to train such a model without significantly impeding the performance of the optimizer. Furthermore we believe that such a versatibility encourages the model more explicitly to extract useful information from the objective function. Another motivation to augment the baseline model with the flexibility to adjust the number of optimization steps has to do with a limitation of the LSTM optimizer. In more demanding optimization tasks the model’s performance deteriorates remarkably when the horizon at test time exceeds the training horizon significantly (also recently revealed by the work of [3]). Hence this limitation can be mitigated if the meta-learner can carefully explore a sufficient long horizon that suits the complexity of the task at hand.

An additional reason to extend the learning to learn approach with yet another idea provided by the ACT method is related to the loss objective of the former approach. We first want to state that an iterative optimization procedure can be viewed as an arbitrary length trajectory or sequence in time in which each time step contributes to the final optimization result. For example one could interpret the loss reduction an optimizer achieves at each time step, as a kind of learning progress which is different for each time step and could therefore be weighted dissimilar. As we will outline in more detail in section 2.2 the LSTM optimizer credits each time step contribution equally and hence implies that earlier and later steps during the optimization process are having

\(^2\)Curious readers are referred to fig. 2.4 which visualizes the computational graph of an RNN using ACT.

\(^3\)E.g. a collection of layers with a gated skip connection (Highway Network) or identity skip connection (Residual Network).
uniform contributions to the final result. We believe that this is not always the case and that a model could benefit from being able to learn these step weights.

One could surmise that the step weights act as an attentional model that can shift during training. For example, exploring longer sequences in an early phase of training combined with the ability to assign low weights to later time steps, could be a way to attenuate high losses and still being able to explore longer sequences without paying too much penalty. As we noted above the ACT approach assigns a halting probability to each computational step and we think that these probabilities could be interpreted as step weights or trajectory weights (we will use both terms interchangeably hereafter). Most important the ACT approach utilizes these probabilities to determine the network depth and we will adopt this idea to control the number of iterative optimization steps performed for an objective function. Therefore we will extend the L2L approach by giving the LSTM optimizer the ability to learn the step weights during training.

Finally in order to prevent such a model from using unnecessary long optimization trajectories, it is necessary to constrain the meta-learner. It is not difficult to imagine that otherwise the optimizer would not be encouraged to limit itself w.r.t. the available computational resources (time and space). Once more we can adopt a crucial contribution of the ACT approach and extend the LSTM optimizer by augmenting the loss objective with a differentiable time penalty that enables us to inject a preference for speed versus accuracy when training the model. Our work contributes in the following ways. We developed two optimizers that combine the L2L and ACT approaches and where one method defines a Bayesian parametric model. The models extend the original LSTM optimizer such that they can be trained with less computational effort on different classes of optimization problems while achieving comparable optimization results as the baseline model. The newly developed models are able to dynamically adjust the number of optimization steps to the input and develop their own training regime that trades off speed versus accuracy based on a prior injected preference.

1.1 Research questions

In order to structure this study more stringently and give the reader a clear perspective on what follows, we formulate a couple of research questions (RQ) that this work answers by conducting a series of experiments. The research questions can be roughly divided into two groups. By answering the first three questions we will gain valuable insights that will corroborate why we have chosen a certain configuration of the LSTM optimizer that acts as a baseline for the performance assessment of our newly developed models. Therefore we will conduct a series of experiments solely with the baseline model in order to answer the following research questions:

RQ1.1 What is the effect on the convergence behavior of the model when training the optimizer for different fixed time horizons?

RQ1.2 How does training the optimizer on partial trajectories influence the convergence behavior of the model?

RQ1.3 Do the effects investigated in RQ1.1 and RQ1.2 depend on the difficulty of the optimization task?

The second part of the questions focuses on the evaluation of our new models and explores their optimization behavior:

RQ2.1 Is it possible to train the newly proposed models with less computational effort than the baseline model while achieving the same optimization performance?

RQ2.2 Are the models able to adapt to the varying difficulty of the objective functions within the same optimization task (also referred to as input dependency)?

RQ2.3 How does the injected preference for speed versus accuracy influence the performance of the models?

RQ2.4 Can the trajectory weights be harnessed to determine the time step when optimization should stop for a specific objective function?
We conclude the introduction by stating the overall structure of this work. In the next chapter 2 we will provide the reader with the necessary background that is essential to understand the underlying concepts of both approaches (L2L and ACT) and the new methods that we propose in this work. In chapter 3 we present a detailed description and formalization of the developed extensions to the LSTM optimizer. Research that is related to our work will be introduced in chapter 4 before we outline the experimental results in chapter 5. The former chapter is interleaved with short concluding paragraphs that answer some of the research questions and summarize the main insights. We conclude the work in chapter 6 by answering the remaining research questions, stating the overall conclusions, discussing the results and indicating directions for future work.
2 | Background

The goal of this chapter is to provide the reader with the necessary background on the approaches, concepts and terminologies that we are using throughout this research. Before we describe the baseline optimizer of [1] in more detail in section 2.2 and reveal the technicalities of the adaptive computation time approach for RNNs in section 2.3, we first state some preliminaries in the upcoming paragraphs.

We start by clarifying some important terms that we frequently use in this work. We then briefly explain the general setting of learning an optimization algorithm which is followed by an illustration of the concept of meta-learning as opposed to base-learning. One of our newly developed models uses the so called Stick-Breaking Process (SBP) to generate an infinite number of random weights and hence we introduce the process in this chapter. Finally we briefly explain Truncated Backpropagation Through Time (BPTT) which is the most practical method for training RNNs.

2.1 Preliminaries

2.1.1 Terminology

Optimizee

We adopt the terminology of [1] by referring to the objective function as the optimizee as opposed to the optimizer. Please note that we will use the terms optimizee and base-learner interchangeable in this work. Figure 2.1 visualizes briefly the relationship between optimizer and optimizee.

Optimization step

During a single time step $t$ the optimizer receives an error signal from the optimizee (e.g. the gradients of the optimizee’s loss w.r.t. the optimizee’s parameters) and processes this information in order to generate an update of the optimizee’s parameters. We will refer to these operations within the same time step $t$ as an optimization step that the optimizer performs in the parameter space of the optimizee.

Fixed horizon versus Number of optimization steps

The careful reader will probably have noticed that we are using two different terminologies and notations for something that seems to be the same, the number of optimization steps $T$ and the fixed horizon $H$. In the context of the LSTM optimizer these two measures are indeed equal because the model is not able to adjust $T$ for a specific objective function during training or testing. We made the deliberate choice to use the term number of optimization steps when we refer to an individual objective function and fixed horizon when denoting a fixed, hard limit that applies to all optimizees.
(where \( \{T, H : T \leq H\} \)). The architectural extensions of the LSTM optimizer that we introduce in chapter 3 are able to adjust the number of optimization steps for an optimizee.

**Halting step**

We refer to the last optimization step that an optimizer performs for a specific optimizee as the **halting step** which is numerically equal to the number of optimization steps \( T \).

**Computational effort**

In this work we often refer to the **computational effort** it takes to train a model. By this we mean the total number of optimization steps that a model performs during training (summed over epochs and optimizees). We provide an equation in section 3.5 when defining the evaluation metrics that we used.

### 2.1.2 Learning an iterative optimization algorithm

In the general setting of learning an optimization algorithm that we are considering in this work, the goal is to learn an algorithm \( A \), which we will refer to as optimizer hereafter, that minimizes a set of objective functions \( F \) drawn from a distribution \( p(f) \). We are focusing on functions that correspond to loss functions of so called base-learners (e.g. regression functions) which are parameterized by \( \theta \) (where \( \theta \in \Theta \) and \( \Theta \in \mathbb{R}^d \)). The optimizer takes as input a random location \( \theta_0 \sim p(\theta) \) in the domain of \( \Theta \) and from there on iteratively updates the parameter location by a step vector which we will denote by \( g_t \):

\[
 \theta_{t+1} = \theta_t + g_t \quad \text{where} \quad g_t = A(f, \theta_0) .
\]  

Hence the optimizer produces a sequence of iterates \( \{\theta_t\}_{t=1}^T \) where \( f(\theta_T) \) is the final minimum loss value found by the algorithm for the optimizee. \( T \) denotes the number of optimization steps taken. Generally, \( T \) is determined for the complete set of objective functions based on a convergence criteria. The goal of the optimizer is to minimize

\[
 \mathbb{E}_{f \sim p(f), \theta_0 \sim p(\theta)} [L(f, A(f, \theta_0))] ,
\]  

where \( L \) denotes the optimizer’s loss (also referred to as meta-loss). As we are interested in minimizing the objective functions, in its simplest form the expected loss is equal to the sum of the separate objective values \( f(\theta_t) \) at query points \( \{\theta_t\}_{t=1}^T \)

\[
 L^{total} = \mathbb{E}_{f \sim p(f), \theta_0 \sim p(\theta)} \left[ \sum_{t=1}^T f(\theta_t) \right] .
\]  

In the work we present here all optimizers will be implemented by means of an RNN \( m \) that models the update rule in (2.1) and learns the optimization algorithm \( A \). Please note that this allows the optimizer to have memory in order to track progression, curvature and other relevant information about the geometric structure of the error surface induced by the optimizees. We would like to emphasize that in the setting of this work the model that learns an optimization algorithm is not prohibited from learning anything about the task of the base-learners it is optimizing. At test time, the learned optimizer is evaluated on unseen optimizees, which correspond to loss functions used at training time (but newly sampled). Hence in our setting we pursue the goal of learning a task dependent optimizer.
2.1.3 Meta learning

Definitions of meta-learning (or learning to learn which we will use interchangeable hereafter) can vary between scientific disciplines and researchers, but in its most general meaning it studies how learning systems can increase in efficiency through experience. An essential goal of meta-learning is to understand how learning itself can become flexible according to the domain or task under study [31]. As Mitchell [22] points out in his early work, inductive learning is only possible in combination with an inductive bias that limits the hypothesis search space when a learning system has to generalize from past experience in order to deal with new unseen problems. In comparison to a base-learner where the bias to solve a particular task is a priori hardwired into the system, a meta-learner is able to learn the bias for a particular problem domain and therefore able to adapt to different tasks. Note that learning to learn is closely related to multi-task and (inductive) transfer learning [24]. The basic idea is that learning is not an isolated process and a learning system is expected to perform increasingly better when exhibited to problems from various domains. Nevertheless this requires that the learner acquires knowledge about learning that can be transferred across tasks.

More closely related to our work is the well established idea that machine learning techniques can be used to optimize the process of learning itself [26], [29]. Our work falls under the umbrella of meta-learning because the product of learning is an algorithm that can be used to train base-learners. Therefore we will briefly explain how meta-learning is applied in our work.

We point the reader’s attention to figure 2.2 which visualizes the difference between base-learning and meta-learning in the context of machine learning that we are interested in. Base-learning in the context of our research corresponds to the general setting of supervised learning in machine learning (left side of figure 2.2). We train a model (base-learner) that learns to approximate a mapping function between input and output pairs that are provided to the model. An essential ingredient of the learning algorithm (blue rectangular box on the left side of fig. 2.2) is the formulation of a differentiable loss objective which quantifies how well the model performs on a specific task (e.g. regression problem). The goal during learning is to minimize the loss function...
which is achieved by computing the gradients of the loss w.r.t. the model’s parameters. At each iteration an optimizer updates the location of the current model’s parameters by a step vector \( \mathbf{g}_t \) in (2.1)). After training\(^1\) i.e. at test time the optimizer and the loss function become redundant and are **decoupled** from the model.

The right side of figure 2.2 visualizes the concept of meta-learning as opposed to base-learning. In meta-learning the target of learning is itself a learning algorithm\(^2\) and compared to base-learning the **learning algorithm**, which is in our case the **learned optimizer**, takes the place of the model in the base-learning setting (green rectangular box in the top right corner of fig. 2.2). The input of the learned optimizer must contain information about the error surface of the optimizees (green rectangular box in the bottom right corner of fig. 2.2). During training the learned optimizer exposes its parameters to a meta-learning algorithm (blue rectangular box on the right side of fig. 2.2), whose role mirrors that of the learning algorithm in the base-learning setting\(^3\). In the previous section we already gave a simple example of the loss objective of a model that learns an optimization algorithm (2.3). Note that in the meta-learning setting we are updating the base-learner’s and the learned optimizer’s parameters in an alternating sequence. After training the meta-learning algorithm becomes redundant and is therefore decoupled from the model while apparently the learned optimizer will still consume the error signals of the optimizees in order to optimize the base-learners.

All models presented in our work can be categorized as meta-learners that learn to optimize differentiable loss functions belonging to other base-learners (optimizees). The product of learning is an algorithm that has learned the update rule stated in (2.1) modeled as an RNN. These optimizers ingest the gradients of the objective functions which equates them to first-order optimization algorithms. There are a couple of advantages of learned optimization algorithms compared to hand-crafted optimizers:

- In the general machine learning setting it is necessary to choose a **learning rate** (frequently denoted by \( \eta \)) which is problem dependent. This hyper-parameter is often found by using random search whereas in our setting this parameter is learned and dynamically adjusted during learning (learning rate decay).

- Hand-engineered optimizers are often derived by analyzing objective functions which can in principle exhibit properties that the actual optimizees does not evince.

- Learned optimizers can extract and exploit problem specific patterns from the objective functions during learning.

### 2.1.4 Stick-breaking process

In section 3.3 we outline a new architectural extension to the LSTM optimizer that uses a Bayesian approach. The formalization makes use of the so called **Stick-Breaking Process** (SBP) \([27]\) which we are briefly introducing here. The goal of the process is to generate an infinite number of random weights \( \{ \pi_k \}_{k=1}^{\infty} \) that adhere to the following constraints:

\[
0 \leq \pi_k \leq 1 \quad \text{and} \\
\sum_{k=1}^{\infty} \pi_k = 1 \quad \text{almost surely ,}
\]

which implies that we can regard the set \( \{ \pi_k \}_{k=1}^{\infty} \) as valid probabilities. Informally the construction of the \( \pi_k \) values can be thought of as an iterative stick-breaking procedure, where at each stage one randomly and independently breaks what is left of a stick of unit length. The length of the newly broken piece is scaled by the remaining stick length from the previous step (except for the first time step). Scaling the current length of the break is essential to make sure that (2.4) holds.

\(^1\)We are not considering continuous learning in this research.

\(^2\)This is the **meta** part.

\(^3\)Actually the meta-learning algorithm is often the same as the learning algorithm in the base-learning setting.
Mathematically the procedure can be compactly written as:

\[
\pi_k = \begin{cases} 
\rho_1 & \text{if } k = 1 \\
\prod_{i=1}^{k-1} (1 - \rho_i) \rho_k & \text{for } k > 1 
\end{cases}, \tag{2.6}
\]

where \(\rho_k\) are independent random variables drawn from a Beta(\(\alpha, \beta\)) distribution. Also note that at each process step we can easily determine the length of the remaining stick:

\[
\pi_K = 1 - \sum_{i=1}^{K-1} \pi_i = \prod_{i=1}^{K-1} (1 - \rho_i), \tag{2.7}
\]

where \(K\) denotes the last process step.

### 2.1.5 Truncated Backpropagation Through Time

We assume that the reader is familiar with the concepts of backpropagation and backpropagation through time [32] (BPTT). In this paragraph we briefly review the common problems of training a recurrent neural network and close the section with a succinct explanation of the so called truncated BPTT algorithm which alleviates the high computational burden of applying BPTT during training of RNNs.

Since the work of [2] and [12] it is known that training RNNs with long-range temporal dependencies is difficult and suffers from two important problems:

1. **Exploding gradients**: due to the compounding effect that can occur when early time step changes have large effects on later time steps and hence result in exponentially large gradients.

2. **Vanishing gradients** [13]: due to the BPTT algorithm which can cause exponentially small gradients in earlier time steps and therefore resulting in slow or no learning at all.

One way to mitigate these problems is to use so called Long Short-Term Memory networks (LSTM) introduced by Hochreiter and Schmidhuber [14] and we remark that all RNNs implemented in our work use this architecture. However one of the main problems of BPTT is its high cost during parameter update and hence it can be difficult or even impossible to use large number of time steps i.e. iterations. In order to reduce the computational load of BPTT one can use truncated BPTT [33] which processes a sequence (with length \(T\)) one time step at a time, and every \(L_1\) steps, it runs BPTT for \(L_2\) time steps [28]. The algorithm is succinctly outlined here. It is important to note that the hidden states are never reset while executing intermediate BPTT runs. Therefore the procedure ensures that long-range temporal dependencies can be captured in the hidden states. The study of Andrychowicz et al. used a sub-sequence length of \(L_1 = L_2 = 20\) in all experiments. We will investigate the effect of the sub-sequence length, which we denote \(L\) or truncated BPTT(\(L\)), on the LSTM optimizer performance in one of our first experiments described in section 5.2.

**Input:** (1) Sequence length \(T\); (2) Number of time steps \(L_1\) after which BPTT is run; (3) Span of time steps \((L_2)\) that are captured during BPTT.

**Please note:** that \(y_t\) denotes the output and \(s_t\) represents the hidden state computed by the RNN at each time step \(t\).

**for** \(t = 1\) to \(T\) **do**

- Run the recurrent neural network for one step and compute \(y_t\) and \(s_t\).
- **if** \(L_1\) divides \(t\) or \(t\) equal to \(T\) **then**
  - Run BPTT from \(t\) down to \((t - L_2)\)

**Algorithm 1:** Truncated BPTT

Please note that in contrast to BPTT in truncated BPTT gradients computed at time step \(t\) do not flow back to all previous time steps \(t' < t\). As a consequence although the hidden states
are maintained and can in principle contain all relevant information from previous time steps, it is still theoretically possible that by truncating the gradient flow long-term dependencies are lost.

In the following two sections we outline in detail the baseline model from Andrychowicz et al. respectively the ACT framework introduced by Graves [8]. Both approaches form the foundation to our proposed architectural extensions that we present in chapter 3.

2.2 LSTM optimizer (baseline model)

As described in the introduction, the LSTM optimizer introduced in [1] uses a meta-learning approach and can be trained to optimize task specific objective functions. This model will be used as a baseline in our work and is implemented as a recurrent neural network (RNN) $m$ parameterized by $\phi$ (we refer to this model as LSTM optimizer and baseline model interchangeable hereafter). Figure 2.3 shows the computational graph of the model and visualizes equation (2.1) where the learned optimization algorithm $A$ is replaced by $m$. At each time step $t$ the gradients w.r.t. the optimizee’s parameters are ingested by the LSTM module which produces a step vector $g_t$ which is added to the current parameter values of the optimizee. In general, the optimizees will differ and contain a certain source of randomness due to the initialization procedures of e.g. the initial parameters $\theta_0$. Therefore the expected loss of the model for a discrete number of computational steps $T$ w.r.t. a given distribution over functions $f$ will be equal to:

$$
L^{total}(\phi) = \mathbb{E}_{f \sim p(f), \theta_0 \sim p(\theta)} \left[ \sum_{t=1}^{T} w_t f(\theta_t) \right] \text{ where } \theta_{t+1} = \theta_t + g_t \text{ and } w_t \in \mathbb{R}_{\geq 0}. 
$$

(2.8)

where $f$ is the function to be optimized which is parameterized by $\theta$ and $W = \{w_t\}_{t=1}^{T}$ denotes a set of arbitrary trajectory weights. The work of [1] used $w_t = 1$ for all time steps which implies that all optimization steps contribute equally to the final optimization result. The study mentions that the initial idea was to use $w_t = 1[t = T]$ which basically means that the sum in (2.8) over $T$ would be removed and $t$ would be replaced by $T$. The authors describe that this option was rejected because it would render truncated BPTT inefficient and they wanted to be able to train the model for partial trajectories. This is indeed an important property of the optimizer because the most interesting optimization problems will require a couple of hundred optimization steps.

Another important characteristic of the architecture of the LSTM optimizer is its coordinatewise network architecture. In order to be able to cope with the possible large number of parameters of the optimizee (e.g. a deep neural network) the parameters of the RNN are shared between the optimizee parameters while the model maintains a separate hidden state for each of them. The result is a relatively small RNN and we follow [1] by implementing the network using a two-layer LSTM which consists of a hidden and so called memory cell state which we denote $s_t$ resp. $c_t$. Please note that figure 2.3 which is taken from the original paper of [1] only visualizes the hidden state which is denoted by $h_t$ but we have deliberately chosen for the notation of $s_t$ in what follows.
The input of the RNN are the gradients of $f$ w.r.t. the parameters $\theta$ of the objective function denoted by $\nabla_{\theta} f(\theta_t)$. The model can be formalized as follows:

$$\begin{bmatrix} g_t \\ s_{t+1} \\ c_{t+1} \end{bmatrix} = m(\nabla_{\theta} f(\theta_t), s_t, c_t, \phi), \quad (2.9)$$

where the hidden and cell state are passed as input to the next time step $t+1$ and can be interpreted as compressed states of the current and previous time steps (note that $s_0$ and $c_0$ will be initialized with 0).

### 2.3 Adaptive Computation time for RNNs

In chapter 1 we shortly introduced the adaptive computation time approach for RNNs which Alex Graves presented in [8]. The basic idea is simple, whereas normally the amount of computation done by an RNN at each time step $t$ is fixed, an RNN using ACT is able to adjust it to the input. In other words the network depth becomes a dynamic function of the inputs received so far. In the setting of this work we equate amount of computation with network depth or number of optimization steps. Figure 2.4 displays the computational graph of an RNN with ACT and the $y$ axis which is labeled with intermediate ponder sequence, represents the adjustable dimension of the model. The RNN is augmented with a sigmoidal halting unit that generates a so called halting activation at each ponder step (denoted as $\{h_n^t\}_{n=1}^N$ in fig. 2.4).

At each time step $t$ the RNN outputs a single state $s_t$ and output $y_t$ that are equal to a weighted sum of intermediate (ponder) states resp. (ponder) outputs and where the halting probabilities are used as weights. The so called halting step is determined by means of the sum of halting activations and a fixed threshold value. In his work Graves successfully applies the approach to a couple of synthetic sequence learning and character prediction tasks. The work makes an important contribution by designing a time penalty (coined ponder cost) that is differentiable and depends on the number of time steps taken. In the following we state the formalization of the model.

![Figure 2.4: Computational graph of RNN with ACT taken from Alex Graves [8]](image)

As shown in figure 2.4, the architecture of a standard RNN is only slightly adjusted by:

- defining an intermediate sequence of (a) state transitions $(s_1^t, \ldots, s_{N(t)}^t)$ and (b) outputs $(y_1^t, \ldots, y_{N(t)}^t)$ at step $t$. $N(t)$ denotes the total number of computational steps performed at step $t$:

$$s_t^n = \begin{cases} S(s_{t-1}, x_t^n) & \text{if } n = 1 \\ S(s_{t}^{n-1}, x_t^n) & \text{otherwise} \end{cases}, \quad (2.10)$$

$$y_t^n = W_y s_t^n + b_y, \quad (2.11)$$
where $n \in \{1, \ldots, N(t)\}$, $W_y$ and $b_y$ denote the weight matrix respectively bias of the associated linear transformation and $S$ denotes a parametric state transition model.

- augmenting the RNN with a sigmoidal halting unit which activation $h^n_t$ is transformed into a halting probability $p^n_t$ that determines whether computation should continue:

$$ p^n_t = \begin{cases} R = 1 - \sum_{n=1}^{N(t)-1} h^n_t & \text{for } t = N(t) \\ h^n_t = \sigma(W_h s^n_t + b_h) & \text{otherwise} \end{cases} \quad (2.12) $$

where $W_h$ and $b_h$ denote the weight matrix respectively bias of the associated linear transformation and $\sigma(.)$ denotes the sigmoid function in order to make sure that $0 \leq h^n_t \leq 1$.

It is important to note that the last time step $t = N(t)$ is assigned the remainder of the probability mass which is denoted as $R$. This ensures that the halting probabilities sum to one.

The sum of halting activations together with a predefined deterministic threshold $\epsilon$ determine the halting step $N(t)$ when computation should stop:

$$ N(t) = \min\{n': \sum_{n=1}^{n'} h^n_t >= 1 - \epsilon\} \quad (2.13) $$

All experiments in [8] used a value of $\epsilon = 0.01$ and due to 2.13 the model is able to stop computation after one time step.

The halting probabilities are then used to define a weighted sum of intermediate network outputs $y_t$ and internal states $s_t$ that are both propagated along the sequence:

$$ s_t = \sum_{n=1}^{N(t)} p^n_t s^n_t \quad (2.14) $$

$$ y_t = \sum_{n=1}^{N(t)} p^n_t y^n_t \quad (2.15) $$

Finally the already introduced ponder cost (denoted $\mathcal{P}$) that penalizes the model for taking long sequences is defined as:

$$ \mathcal{P} = N(t) + R \quad (2.16) $$

This regularizer can be added to the loss function of the specific task we want to optimize:

$$ \hat{L}(\phi) = L(\phi) + \tau \mathcal{P} \quad (2.17) $$

As Graves notes the value of the hyper-parameter $\tau$, which trades off computational efficiency against accuracy of the learning result, has a significant effect on the model behavior and is difficult to choose. It is interesting to note that by minimizing the ponder cost the model should be encouraged to increase the halting unit activation of all steps preceding the final time step $N(t)$\textsuperscript{4}. An additional remark should be made on the differentiability of the ponder cost w.r.t. the halting activations. As noted by Graves the ponder cost is discontinuous with respect to $h^n_t$ at the points where $N(t)$ changes value. In addition $R$ is not a continuous function of the halting units which was outlined by Hugo Larochelle in [18]. Nevertheless we follow [8] by ignoring these discontinuities because they do not seem to matter as is corroborated by our results.

We would like to point the reader to a subtle difference w.r.t. the notation used in our work compared to the work of Graves. The green axes in figure 2.4 visualize:

\textsuperscript{4}Because the remaining probability mass not used in the preceding steps is added to the time penalty, see eq. (2.16).
• the intermediate ponder sequence (vertical) which we refer to with the superscript \( n \) in the notation above, where \( n \in \{1, \ldots, N(t)\} \);

• the time sequence (horizontal) which we denote with the subscript \( t \).

As outlined before, an RNN using ACT is able to adjust the number of computational steps \( N(t) \) per time step \( t \) where the sequence length in the time dimension is fixed for each task. In our work which applies the ACT approach to the meta-learner from [1] the number of computational steps per time step is fixed and hence we are not using the superscript \( n \). Instead in our approach the time sequence length is adjustable (\( x \) axis in fig. 2.4). Individual time steps will be denoted by \( t \) (often as subscript) and the total number of optimization steps for an optimizee will be denoted by \( T \) (also refer to section 2.1.1).
3 | Method

In this section we describe and formalize two newly developed optimizers that combine the learning to learn and adaptive computation time approaches we introduced in the previous chapter. We first convey the main ideas and assumptions underlying our approaches before we present the extensions in more detail.

![Computational graph of the computation time adaptive optimizer (M-ACT)](image)

Figure 3.1: Computational graph of the computation time adaptive optimizer (M-ACT)

### 3.1 General idea and assumptions

In section 2.2 we explained the LSTM optimizer in more detail and stated the original loss objective of the model in equation (2.8) which takes the step losses of the whole optimization trajectory into account. Intuitively one would think that in order to train a model that learns an optimization algorithm the loss function should only consider the loss of the last time step $E_{T \sim \rho(f)}[f(\theta_T)]$.

This loss was indeed first considered by Andrychowicz et al. but rejected because the amount of information conveyed by such an objective value would be temporally very sparse. As mentioned in the work of [3], by using instead the objective as in (2.8) the optimizer is provided with information from every time step along the optimization trajectory. It is also interesting to interpret the loss objective from the perspective of reinforcement learning (RL) because minimizing the sum of objective values is equivalent to finding a policy that minimizes the expected undiscounted cumulative regret. Note that in the original approach of [1] the loss objective (2.8) contains a trajectory weight $w_t$ associated with each time step which was set to a constant value of one for

---

1Because this should correspond to the lowest loss value.
all time steps and therefore equates to an *undiscounted* setting in RL terminology. We previously stated that such a setting implies that earlier and later steps during the optimization process are having uniform contributions to the final result.

In the introduction of this work we revealed our main motivation for combining the L2L with the ACT approach. An essential assumption underlying our work is the hypothesis that the contributions of the optimization steps to the final result should not be equally weighted. The loss reduction or *learning progress* an optimizer achieves at each time step is different and could therefore be weighted dissimilar. Hence in our approach these *step weights* are treated as extra parameters that can be learned during training. In order to be able to interpret the weights as valid probabilities or mixing coefficients we constraint the model and make sure the set of weights always sums to one. This implies that the original loss function in (2.8) turns into a weighted sum of step losses. Inspecting the original loss function in (2.8) and taking into account the last constraint that we formulated one would expect that the lowest loss value is achieved by setting the last time step weight to one and all preceding step weights to zero. We explained above that such a loss is undesirable and therefore we added an effective regularization term that prevents the optimizer from choosing this option.

Another important idea that is central to our approaches is the assumption that the training procedure for a learned optimizer can be more efficient if the model’s behavior becomes input dependent. We achieve this by allowing the optimizer to adapt the number of optimization steps $T$ for each individual optimizee during training. We belief this approach encourages the model more explicitly to extract useful information from the objective function. This contrasts the approach of [1] which uses a fixed training horizon for all optimizees.

The ideas of learning the step weights and adjusting the number of computational steps per optimizee at training time are strongly entangled because the cumulative sum of trajectory weights determines the halting step. In order to achieve that our new optimizers are computationally par-simonious and at the same time exhibit *aggressive* optimization behavior, our models are penalized for taking too many optimization steps.

In the next section we will first introduce the *computation time adaptive meta-learner* (M-ACT) before we describe the M-PACT optimizer which uses a Bayesian parametric model.

### 3.2 Computation time adaptive meta-learner

In this section we present the computation time adaptive meta-learner (M-ACT) which augments the baseline LSTM optimizer described in section 2.2 with an additional sigmoidal halting unit $h$:

$$h_t = \sigma(W_h s_t + b_h), \quad (3.1)$$

where $W_h$ and $b_h$ denote the associated weight matrix respectively bias.

Figure 3.1 shows the computational graph of the new optimizer and we added two *partial views* of the graph in appendix A which guide the reader through the graph. In comparison to the computational graph of the baseline model presented in figure 2.3 we added a second *swim lane*\(^2\) that visualizes the computation of the loss objective which is equal to a weighted sum of optimizee time step losses plus a regularization term. The underlying idea of our approach is simple. In each step the model generates an additional halting activation $h_t$ where the sigmoid function makes sure that $0 \leq h_t \leq 1$. The halting activations are used to determine the time step weight $p_t$:

$$p_t = \begin{cases} R = 1 - \sum_{t=1}^{T-1} h_t & \text{for } t = T \\ h_t & \text{otherwise.} \end{cases} \quad (3.2)$$

In order to make sure that we can interpret $\{p_t\}_{t=1}^T$ as valid probabilities, the last halting activation $h_T$ is neglected and the so called *rest probability* $1 - \sum_{t=1}^{T-1} h_t$ is assigned to the last trajectory weight $p_T$.

\(^2\)Highlighted with a green background in fig. 3.1.
The M-ACT optimizer is formalized as follows:

\[
\begin{bmatrix}
g_t \\
p_t \\
s_{t+1} \\
c_{t+1}
\end{bmatrix} = m'(\nabla_\theta f(\theta_t), s_t, c_t, \phi, W_h, b_h).
\] (3.3)

The model exploits the generated halting activations \(\{h_t\}_{t=1}^T\) and trajectory weights \(\{p_t\}_{t=1}^T\) in two ways:

1. **Determine the number of computational steps**: we follow Graves and use the cumulative sum of activations and a fixed threshold value in order to determine the number of optimization steps \(T\) for an objective function. In order to prevent the model from using excessive computational resources through exploration of very long time sequences, we impose a hard limit \(H\) on the maximum number of time steps

\[
T = \min\{H, \min\{t' : \sum_{t=1}^{t'} h_t > 1 - \epsilon\}\}. 
\] (3.4)

We encountered the effect of the maximum horizon \(H\) when assessing the performance of the model while optimizing shallow neural networks (we refer the reader to section 5.4). In all experiments we set \(\epsilon\) to 0.01 which equates to a fixed threshold of 0.99.

2. **Compute weighted sum of time step losses**: as noted in equation (2.8) the loss function of the LSTM optimizer uses a fixed set of trajectory weights \(W\) (where \(w_t = 1\) for each time step). In our method the constant weights are replaced by the learned \(p_t\) values \(W = \{p_t\}_t\) and therefore the loss objective is equal to a weighted sum of time step losses.

Furthermore, we extend the original loss objective of the LSTM optimizer as formalized in equation (2.8) with a ponder cost i.e. time penalty that was introduced by the ACT approach (refer to eq. (2.16)). This results in the following loss function for the proposed model:

\[
L_{\text{total}}(\phi) = \mathbb{E}_{f(p)} \left[ \sum_{t=1}^{T(f)} p_t f(\theta_t) + \tau\left(T(f) + R(f)\right) \right],
\] (3.5)

where \(p_t\) is computed according to (3.2) and \(T(f)\) is determined based on (3.4). Please note that we have subsumed the parameters \(W_h\) and \(b_h\) into the set of model parameters \(\phi\) and by using the superscript notation \(T(f)\) and \(R(f)\) we make explicit that the number of optimization steps is determined per function. We will approximate the expectation in (3.5) by sampling minibatches of functions from \(p(f)\):

\[
L_{\text{total}}(\phi) \approx \bar{L}_{\text{total}}(\phi) = \frac{1}{M} \sum_{m=1}^{M} \left[ \sum_{t=1}^{T(f(m))} p_t f^{(m)}(\theta_t) + \tau\left(T(f(m)) + R(f(m))\right) \right],
\] (3.6)

where \(M\) denotes the size of the minibatch.

We emphasize that this formulation endows the model with three flexibilities that the baseline model lacks:

1. During training the number of optimization steps is not fixed and the model can explore different lengths of sequences.

2. The trajectory weights can be learned during training and are only constrained by the necessity to sum to one (\(\sum_{t=1}^{T} p_t = 1\)).

3. A trade-off preference between computational effort and accuracy can be injected into the model by choosing a specific value for the hyper-parameter \(\tau\). A higher value implies an increased preference for speed. In order to answer research question RQ2.3 we will investigate the effect of \(\tau\) when evaluating the model on convex and non-convex regression tasks.
As we have motivated earlier (1, 3.1), we belief that these versatilities will result in less computational effort to train the model if compared to the baseline model. Furthermore we think that the developed adjustments will encourage the model to behave in an input dependent way.

### 3.3 Bayesian approach to computation time adaptive meta-learning

In this section we outline the details of the computation time adaptive meta-learner that utilizes a Bayesian approach (M-PACT). The goal of the model is identical to the M-ACT optimizer i.e. we are enabling the meta-learner to adjust the computational effort during training by generating trajectory weights for each optimizee that quantify the different time step contributions. These probabilities are harnessed to determine the so called halting step. In contrast to the M-ACT model we use a Bayesian approach to the formulation of the loss objective. The trajectory weights correspond to the probabilities of a multinomial latent variable which specify the different time step credits. In contrast to the baseline and the M-ACT model we need to make sure that the objective value of the optimizee $f(\theta_t)$ can be expressed as a probability e.g. if the base-learners are regression functions the objective value conveys our uncertainty over the predicted target value. Furthermore we will regularize the model by introducing a prior distribution that expresses our a priori assumptions about the individual time step contributions. Another dissimilarity with the M-ACT model arises through the use of a stochastic threshold value that is utilized to determine the halting step per optimizee.

#### 3.3.1 Preliminaries

We are defining a Bayesian parametric model that contains a dataset $X = \{x^{(n)}\}_{n=1}^N$ consisting of $N$ i.i.d. samples of a continuous variable $x$. In the context of this work $x$ corresponds to the objective value of an optimizee $f(\theta_t)$. We presume the data is generated by some random process which involves a discrete latent variable $T \in \mathbb{N}^+$ that denotes an arbitrary length trajectory and $z$ a $T$-dimensional binary latent variable which has a 1-of-$T$ representation. The values of $z_t$ therefore satisfy

$$z_t \in \{0, 1\}$$

$$\sum_{t=1}^{T} z_t = 1 ,$$

and hence the vector $z$ has $T$ possible states. We define the joint distribution $p(x, z, T)$ by means of the following factorization:

$$p(x, z, T) = p(T) \ p(z|T) \ p(x|z) ,$$

which implies that $x$ and $T$ are conditionally independent given $z$. The probabilities of the conditional distribution $p(z|T)$ express the time step responsibilities. Hence we interpret them as coefficients $\pi_{t|T}$ of a mixture model where

$$p(z_t = 1|T) = \pi_{t|T} ,$$

and the coefficients $\{\pi_{t|T}\}_{t=1}^T$ must satisfy:

$$0 \leq \pi_{t|T} \leq 1$$

$$\sum_{t=1}^{T} \pi_{t|T} = 1 ,$$

\(^3\text{Where } \mathbb{N}^+ \text{ denotes the set of natural numbers without 0.}\)
in order to ensure they are valid probabilities. Based on these definitions and constraints we can write the conditional distribution \( p(z|T) \) as

\[
p(z|T) = \prod_{t=1}^{T} \pi_{zt}.
\]

(3.13)

We assume that the priors \( p_\phi(T) \), \( p_\phi(z|T) \) and the likelihood \( p_\phi(x|z) \) can be parameterized by an RNN network \( m \) with \( \phi \). Our goal is to maximize the marginal likelihood \( p_\phi(X) \) which is equal to a sum over the marginal likelihoods of individual datapoints \( \log p_\phi(X) = \sum_{n=1}^{N} \log p_\phi(x^{(n)}) \). The marginal likelihood of an individual datapoint can be decomposed into:

\[
\log p_\phi(x^{(n)}) = \log \sum_{T=1}^{\infty} p_\phi(T, x^{(n)}) = \sum_{T=1}^{\infty} p_\phi(T) p_\phi(x^{(n)}|T)
\]

(3.14)

\[
\geq \sum_{T=1}^{\infty} p_\phi(T) \log p_\phi(x^{(n)}|T)
\]

(3.15)

\[
= \mathbb{E}_{p_\phi(T)}[\log p_\phi(x^{(n)}|T)]
\]

(3.16)

by making use of the sum and product rules of probability and by lower bounding the marginal log-likelihood \( \log p_\phi(x^{(n)}) \) in equation (3.15) by applying Jensen’s inequality, making use of the concavity of the log function.

### 3.3.2 Deriving an evidence lower bound for \( p_\phi(x^{(n)}|T) \)

Our goal is to maximize the marginal likelihood \( p_\phi(x^{(n)}) \) and we could proceed from here with a variational inference (VI) approach for the complete-data likelihood function \( p_\phi(x^{(n)}, z, T) \) but we instead choose to solely tackle the conditional distribution \( p_\phi(x^{(n)}|T) \) with a VI approach and model \( p_\phi(T) \) with a deterministic function \( k(.) \) whose input is a random noise source \( \epsilon \) and whose output is \( T \). In the following we first derive an objective that can be used to maximize the marginal likelihood \( p_\phi(x^{(n)}|T) \). Subsequently we explain how we model \( p_\phi(T) \) before we put these parts together so that we are able to maximize the marginal likelihood \( p_\phi(x^{(n)}) \).

Instead of maximizing the marginal likelihood \( p_\phi(x^{(n)}|T) \) we maximize the complete-data likelihood function \( p_\phi(x^{(n)}, z|T) \) because this is significantly easier. We introduce the distribution \( q_\psi(z|x^{(n)}, T) \) which is an approximation of the true posterior distribution \( p_\psi(z|x, T) \) which we assume has the form of a truncated geometric distribution with shape parameter \( \nu \). Figure 4 (in appendix D) visualizes the prior distribution for three different shape parameter values \( \nu \). A larger value for \( \nu \) equates to a stronger regularization force because in those situations most of the probability mass is concentrated on earlier time steps and we assume that the model would naturally try to assign larger weights to later time steps because the objective loss will be lower then. In fact as we have pointed out in the beginning of this chapter (section 3.1), one would assume that a model has the tendency to assign a weight value of zero to all time steps preceding the last time step and we explained earlier that this undesirable. Therefore we believe that choosing a truncated geometric prior regularizes the model in a proper way. When learning progresses and the regularization force increases, we think that the model should be encouraged to take larger leaps in the optimizee’s parameter space in earlier time steps because this should be beneficial for all successive time steps.

We proceed by decomposing the marginal likelihood \( p_\phi(x^{(n)}|T) \) as follows:

\[
\log p_\phi(x^{(n)}|T) = \mathcal{L}(q, \phi; x^{(n)}) + D_{KL}(q_\phi(z|x^{(n)}, T) \parallel p(z|x^{(n)}, T))
\]

(3.17)

where \( \mathcal{L} \) denotes the evidence lower bound (ELBO) and \( D_{KL} \) the Kullback-Leibler (KL) divergence between the true posterior \( p_\phi(z|T) \) and the approximate posterior \( q_\phi(z|x, T) \). We could maximize \( \log p_\phi(x^{(n)}|T) \) by minimizing the KL-divergence but due to the non-negativity of the KL-divergence
the ELBO is a lower bound on $\log p(x^{(n)}|T)$ and hence we can instead maximize the lower bound $\mathcal{L}(q, \phi; x^{(n)})$ which decomposes into:

$$\log p_\phi(x^{(n)}|T) \geq \mathcal{L}(q, \phi; x^{(n)}) \geq E_{\tilde{p}_\phi(T)} \left[ \log p_\phi(x^{(n)}|T) \right] \geq E_{p_\phi(T)} \left[ \mathcal{L}(q, \phi; x^{(n)}) \right] = E_{q_\phi(z|x^{(n)},T)} \left[ \log p_\phi(x^{(n)}, z|T) - \log q_\phi(z|x^{(n)}, T) \right]$$

$$= E_{q_\phi(z|x^{(n)},T)} \left[ \log p_\phi(x^{(n)}|z) \right] - D_KL (q_\phi(z|x^{(n)}, T) \parallel p(z|T))$$

$$= \sum_{t=1}^{T} q_\phi(z_t|x^{(n)}, T) \log p_\phi(x^{(n)}|z_t) - q_\phi(z_t|x^{(n)}, T) \log \left( \frac{q_\phi(z_t|x^{(n)}, T)}{p(z_t|T)} \right).$$

where we made use of the fact that $q_\phi(z|x^{(n)}, T)$ can be factorized as follows:

$$q_\phi(z|x^{(n)}, T) = \prod_{t=1}^{T} q(z_t|x^{(n)}, T)^{z_t}.$$ (3.22)

Furthermore we deliberately omitted the conditioning on $T$ in $p(x|z_t)$ because $x$ is conditionally independent from $T$ given $z$.

As noted above our dataset consists of $N$ data points from a dataset $X$. We can construct an estimator of the evidence lower bound of the full dataset, based on minibatches:

$$\mathcal{L}(q, \phi; X^M) \approx \tilde{\mathcal{L}}^M(q, \phi; X^M) = \frac{1}{M} \sum_{m=1}^{M} \tilde{\mathcal{L}}^M(q, \phi; x^{(m)}),$$ (3.23)

where $M$ denotes the size of the minibatch and $X^M = \{x^{(m)}\}_{m=1}^{M}$.

Replacing $\log p_\phi(x^{(n)}|T)$ in equation (3.16) with the ELBO, we can rewrite the marginal likelihood $p_\phi(x^{(n)})$ as:

$$\log p_\phi(x^{(n)}) \geq E_{p_\phi(T)} \left[ \log p_\phi(x^{(n)}|T) \right] \geq E_{p_\phi(T)} \left[ \mathcal{L}(q, \phi; x^{(n)}) \right] = E_{q_\phi(z|x^{(n)},T)} \left[ \log p_\phi(x^{(n)}, z|T) - \log q_\phi(z|x^{(n)}, T) \right]$$

$$\approx E_{p_\phi(T)} \left[ \frac{1}{M} \sum_{m=1}^{M} \tilde{\mathcal{L}}^M(q, \phi; x^{(m)}) \right].$$ (3.27)

Before we outline the approximation of the expectation of $\mathcal{L}(q, \phi; X^{(n)})$ w.r.t. $T$ we explain how the $q_\phi(z_t|x^{(n)}, T)$ probabilities are computed for each time step.

### 3.3.3 Generating the trajectory weights

The proposed model must be able to generate a finite but arbitrary sequence length ($T$) of probabilities $\{q_\phi(z_t|x^{(n)}, T)\}_{t=1}^{T}$, where each $q_\phi(z_t|x^{(n)}, T)$ is associated with a specific time step $t$. Please note that hereafter we will refer to $q_\phi(z_t|x^{(n)}, T)$ as a compact notation for $q_\phi(z_t|x^{(n)}, T)$ to denote the approximated weight associated with step $t$ given $x^{(n)}$ and $T$. We are dealing with an arbitrary sequence length that must obey to the same constraints that we have already defined for the mixture coefficients $\pi_t/T$ of the prior $p_\phi(z|T)$:

$$0 \leq q_{z_t|x^{(n)},T} \leq 1 \quad \text{and} \quad \sum_{t=1}^{T} q_{z_t|x^{(n)},T} = 1.$$ (3.28)
In order for our model to be computational efficient it is necessary that we are able to compute the probability \( q_{z_t|x^{(n)},T} \) at time step \( t \) or in other words, we want each step probability to be independent of future time steps. On the other hand it is desirable that the model experiences the consequences of having generated the probabilities of earlier time steps (compared to the current time step \( t \)) because we conjecture that such feedback facilitates learning.

A well known process that adheres to these requirements is the so called stick-breaking process (SBP) which was introduced in section 2.1.4. Inherent to this procedure we will always be left with a remaining portion of the initial unit length stick (which suites the idea of an theoretically infinite horizon \( T \)). In order to meet the constraints defined in (3.28), (3.29) we have to make sure that the final step probability \( q_{z_T|x^{(n)},T} \) will be equal to the remaining probability i.e. stick length, denoted \( \delta_{q_{T-1}} \). Using the SBP, the generation of the trajectory weights can be formalized as follows:

\[
q_{z_t|x^{(n)},T} = \begin{cases} 
\rho_t & \text{if } t = 1 \\
\prod_{i=1}^{t-1} (1 - \rho_i) \rho_t & \text{for } 1 < t < T \\
\delta_{T-1} = 1 - \sum_{i=1}^{T-1} q_{z_i|x^{(n)},T} & \text{for } t = T \,.
\end{cases}
\]  

(3.30)

The \( \rho_t \) values are generated by the model which is implemented as an RNN \( m'' \) parameterized by \( \phi \):

\[
\begin{bmatrix} g_t \\ \rho_t \\ \sigma_t \\ c_{t+1} \end{bmatrix} = m''(x^{(n)}, s_t, c_t, \phi, \mathbf{W}_\rho, \mathbf{b}_\rho) \text{ where } \rho_t = \sigma(\mathbf{W}_\rho s_t + \mathbf{b}_\rho) \,,
\]

(3.31)

where \( \sigma(\cdot) \) denotes the sigmoid function and \( \mathbf{W}_\rho \) respectively \( \mathbf{b}_\rho \) are the parameters of the linear transformation that generates the \( \rho_t \) value. Hence the computed \( q_{z_t|x^{(n)},T} \) values are input dependent and can be interpreted as samples from the approximated posterior distribution \( q_\phi(z_T|x^{(n)},T) \).

An important subtlety must be noted here. Due to the stick-breaking procedure stated in (3.30), each step probability is conditioned on the previous time steps:

\[
q_{z_t|x^{(n)},T} = \begin{cases} 
q_\phi(z_t|x^{(n)}) & \text{if } t = 1 \\
q_\phi(z_t|x^{(n)}, z_{1:t-1}) & \text{for } 1 < t < T \\
q_\phi(z_T|x^{(n)}, z_{1:T-1}, T) & \text{for } t = T \,.
\end{cases}
\]

(3.32)

and all step probabilities with \( t < T \) are independent of the horizon \( T \). Only the last time step \( q_\phi(z_T|x^{(n)}, z_{1:T-1}, T) \) depends on \( T \).

Using this last insight from (3.32) we can rewrite the result for the evidence lower bound of \( p(x^{(n)}|T) \) that we stated in (3.21) as follows:

\[
\mathcal{L}(q, \phi; x^{(n)}) = \sum_{t=1}^{T-1} q_\phi(z_t|x^{(n)}, z_{1:t-1}) \log p_{\phi}(x^{(n)}|z_t) - q_\phi(z_t|x^{(n)}, z_{1:t-1}) \log \left( \frac{q_\phi(z_t|x^{(n)}, z_{1:t-1})}{p(z_t|T)} \right) + q_\phi(z_T|x^{(n)}, z_{1:T-1}, T) \log p_{\phi}(x^{(n)}|z_T) - q_\phi(z_T|x^{(n)}, z_{1:T-1}, T) \log \left( \frac{q_\phi(z_T|x^{(n)}, z_{1:T-1}, T)}{p(z_T|T)} \right) \,.
\]

(3.33)

The formalization boils down to the point that we can dissect the evidence lower bound in a part that contains the interval from time step one to the penultimate time step \( T - 1 \) and a second part that only comprises the last time step \( T \). The approximated time step weights \( q_\phi(z_t|x^{(n)}, z_{1:t-1}) \) in the first part solely depend on the previous time steps but not on the final horizon \( T \) whereas the step weight that corresponds to the last time step does dependent on \( T \) as well \( q_\phi(z_T|x^{(n)}, z_{1:T-1}, T) \). This is important because the insight justifies how we sample \( T \) which we will outline in the next paragraph.
3.3.4 Sampling horizons

At the beginning of this section 3.3.1 we stated that $T$ is a discrete latent variable denoting an arbitrary length trajectory. We do not use a Bayesian approach to model $p_{\phi}(T)$ and hence we are not specifying an explicit prior density for $T$. Instead we will sample $T$ from an approximated posterior distribution $q_{\phi}(T|x)$ by defining a deterministic function $k(.)$ whose output is $T$. The input of $k(.)$ is the observable variable $x^{(n)}$, the set of stick-breaking probabilities $\{q_{\phi}(z_{i}|x^{(n)}, z_{1:i-1})\}_{i=1}^{\infty}$ which we denote by $Q$, and a random noise source $\epsilon$. The function acts as a simulator and is formulated as follows:

$$
T = k_{\phi}(x^{(n)}, Q, \epsilon) = \min \left\{ H, \min \left\{ t': \sum_{i=1}^{t'} q_{\phi}(z_{i}|x^{(n)}, z_{1:i-1}) \geq 1 - \epsilon \right\} \right\} 
$$

(3.34)

with $\epsilon \sim U(0,1)$,

where $H$ denotes a hard limit on the number of allowed optimization steps during training and $\epsilon$ denotes the random threshold value that we sample for each optimizee from $\epsilon \sim U(0,1)$. This implies that computation will stop when the accumulated probability up to step $t'$ will exceed a stochastic threshold value. This condition is similar to the deterministic threshold used by the M-ACT model. We argue that modeling $T$ through $k(.)$ the optimizer is unconstrained to progressively adjust the distribution of $T$ during training. One could also come up with an interesting interpretation of modeling $T$ through a stochastic threshold. The optimizer’s behavior must reflect the uncertainty about the trajectory length and act by accounting for the probability that the horizon will be reached now or later, i.e., it must average over the mixture of possible sequence lengths that might be available for the optimization procedure.

Combining our earlier results from (3.24) with the approximation of $q_{\phi}(T|x^{(n)})$ we can finally formalize an estimator for the marginal likelihood $p_{\phi}(x^{(n)})$ as follows:

$$
\log p_{\phi}(x^{(n)}) \geq \mathbb{E}_{p_{\phi}(T)} \left[ \mathcal{L}(q, \phi; x^{(n)}) \right] 
$$

(3.35)

$$
\approx \mathbb{E}_{q_{\phi}(T|x^{(n)})} \left[ \mathcal{L}(q, \phi; x^{(n)}) \right] 
$$

(3.36)

$$
= \mathbb{E}_{q_{\phi}(T|x^{(n)})} \left[ \mathbb{E}_{q_{\phi}(z|x^{(n)}, T)} \left[ \log p_{\phi}(x^{(n)}, z|T) - \log q_{\phi}(z|x^{(n)}, T) \right] \right] 
$$

(3.37)

$$
= \mathbb{E}_{q_{\phi}(T|x^{(n)})} \left[ \mathbb{E}_{q_{\phi}(z|x^{(n)}, T)} \left[ \log p_{\phi}(x^{(n)}|z) - D_{KL}(q_{\phi}(z|x^{(n)}, T) \parallel p(z|T)) \right] \right] 
$$

(3.38)

$$
= \mathbb{E}_{p(\epsilon)} \left[ \mathcal{L}(q, \phi; x^{(n)}, \epsilon) \right] 
$$

(3.39)

$$
\approx \frac{1}{L} \sum_{l=1}^{L} \frac{1}{M} \sum_{m=1}^{M} \tilde{\mathcal{L}}^{M} \left( q, \phi; x^{(m)}, \epsilon^{(m,l)} \right) 
$$

(3.40)

where $\epsilon^{(m,l)} \sim U(0,1)$ and $L$ is the number of horizons we sample for a specific datapoint $x^{(m)}$.

**Approximated posterior distribution $q_{\phi}(T|x)$**

Due to definition of the stick-breaking procedure (3.30) and the way we sample $T$ (3.34), the approximated posterior distribution $q_{\phi}(T|x)$ can be written as:

$$
q_{\phi}(T|x^{(n)}) = \begin{cases} 
\rho_{1} & \text{if } T = 1 \\
\prod_{i=1}^{T-1} (1 - \rho_{i}) \rho_{T} & \text{for } T > 1 .
\end{cases}
$$

(3.41)

The formalization looks very similar to the stick-breaking process due to the fact that we sum the individual time step probabilities (generated by the SBP) in order to determine $T$. If we sum the probabilities of individual events $\{A^{t}\}_{t=1}^{T}$ to determine another event $B$ the probability of $B$ should be equal to the product of the individual probabilities $p(A^{t})$. If we interpret the value of $\rho_{t}$ as the probability to stop at step $t$ and $(1 - \rho_{t})$ as the probability to continue then the result in (3.41) is what we would expect. Actually if all $\rho$ values would be equal, $T$ could be interpreted as a geometric random variable which fits the idea that $T$ denotes an arbitrary (possibly infinite) sequence length of optimization steps.
Approximated marginal posterior distribution \(q(z|x^{(n)})\)

Finally in appendix E we derive the formalization of the marginal posterior distribution \(q(z|x^{(n)})\) and state the result here:

\[
q(z_t|x^{(n)}) = \sum_{T=t}^{\infty} q(T|x^{(n)}) q(z_t|x^{(n)}, T) = q(T = t|x^{(n)}) \delta_{q_{t-1}} + q(z_t|x^{(n)}, T > t) \left(1 - \sum_{T=1}^{t} q(T|x^{(n)})\right),
\]

where \(\delta_{q_{t-1}}\) denotes the rest probability or remaining stick length after we broke the stick in the previous time step. Although the decomposition of the infinite sum into a part where \(T = t\) and a second part where \(T > t\) is comprehensible, we concede that the interpretability of the result is kind of hard. Nevertheless we believe that the result for the approximated posterior \(q(\phi|T|x^{(n)})\) (which is in fact induced by the approximation of \(q(z_t|x^{(n)}, T)\)) is useful because based on (3.41) we could interpret the arbitrary trajectory length \(T\) as a latent random variable having a geometric distribution.

### 3.4 Differences between M-ACT and M-PACT model

Before we describe the evaluation metrics in the next section we briefly note the differences between the proposed methods. Both loss functions consist of a weighted sum of step losses and a regularization term which differs between the approaches. The M-ACT optimizer uses the ponder cost introduced by the ACT approach. It penalizes the model for taking long time sequences and not using the available probability mass in the steps preceding the final time step. The M-PACT model is regularized by the KL-divergence using a prior distribution with shape parameter \(\nu\). Also note that due to the Bayesian approach, we need to make sure that the objective value of the optimizee \(f(\theta_t)\) can be expressed as a probability. This is not required for the M-ACT model.

Another important difference between the two approaches concerns the procedure generating the step weights. The M-ACT model uses the raw halting activations and only the last time step probability depends on the preceding activations (eq. (3.2)). In contrast the M-PACT approach utilizes the stick-breaking process to generate the step probabilities. Due to the SBP each step weight is conditioned on the previous steps.

Finally the models differ w.r.t. the procedure that determines the halting step for an optimizee during training. The M-ACT model uses the cumulative sum of halting activations in combination with a fixed threshold whereas the M-PACT model utilizes the cumulative sum of trajectory weights in combination with a stochastic threshold.

### 3.5 Evaluation of models

As mentioned in section 2.2, we compare the performance of our two new optimizers (M-ACT and M-PACT) against the LSTM optimizer. In a separate experiment not reported here, we first validated our implementation of the baseline optimizer on the quadratic function experiment described in [1] to make sure that the baseline model performs roughly the same as the original model.

When comparing the three models we focused on the following, quantitative performance metrics:

1. **Loss driven progress**: The loss reduction achieved in each time step compared to the initial loss at time step zero \((f(\theta_0) - f(\theta_t))\). Important to note, we are referring to the loss of the optimizee not of the optimizer. In general, we will often use the term **mean step loss** which refers to the mean loss of the optimizees or in other words the mean learning curves of
the optimizees\textsuperscript{4}. Most often it was sufficient to visually inspect these learning curves but in addition we always compared the precise numerical values for certain time steps.

2. **Final average loss value after \( T \) time steps**: During evaluation the models were given a fixed number of time steps (we used 100 and 200 steps) to minimize a sampled batch of optimizees. We used the mean loss value achieved in the final step to compare the performance of the models:

\[
E_{f \sim p(f)} \left[ f(\theta_T) \right].
\]  

(3.44)

3. **Total number of optimization steps during training**: In order to measure the computational effort to train the models we summed the optimization steps that the meta-learner performed during learning:

\[
S_{\text{total}} = \sum_{e=1}^{E} \sum_{m=1}^{M} T_{f}^{(m,e)},
\]  

(3.45)

where \( E \) denotes the number of training epochs, \( M \) denotes the number of functions per epoch and \( T_{f}^{(m,e)} \) represents the number of optimization steps performed for a specific optimizee \( f \). Note that \( T_{f}^{(m,e)} \) is fixed for the baseline model but was adjustable for the M-ACT and M-PACT models.

4. **Difficulty of the objective function**: As formulated in research questions **RQ2.2** we investigated whether the new models adjusted the computational effort to the difficulty of the objective function. Hence we defined a measure that approximates the difficulty of a task. In two of our experiments we used regression functions as optimizees. We conjecture that in this case a sensible measure for the difficulty is the initial negative log-likelihood (NLL) value of the function at time step \( t_0 \) assuming that larger NLL values correspond to a more difficult optimization task. In the last experiment the base-learners correspond to simple multilayer perceptrons (MLP) that perform image classification on the MNIST dataset. We surmised that the difficulty of the task is proportional to number of hidden units or layers\textsuperscript{5}. As we will explain in more detail in section 5.4 and 6, unfortunately we can currently only present preliminary results w.r.t. the learned input dependency the models obtained during the MLP task.

\textsuperscript{4}Note, there is a little subtlety here. Because the output of our learning procedure is an algorithm that is used to optimize base-learners we evaluate the performance of the optimizers indirectly through the learning curves of the optimizees.

\textsuperscript{5}Which is basically the same as assuming that the difficulty is proportional to the number of optimizee parameters.
4 Related work

This chapter provides an overview of previous research that is relevant to this work. In chapter 1 and 2 we have already described in detail the research of [1] and [8] which form the foundation for our work and therefore we will skip a lengthy description here.

Adaptive computation time

The work of [7] extends the ACT approach of Graves to Spatially Adaptive Computation Time (SACT) for Residual Networks which adapts the amount of computation between spatial positions. In their approach each residual unit\(^1\) within a residual block\(^2\) is augmented with a sigmoidal halting unit. Computation within a residual block stops as soon as the cumulative sum of halting activations reaches a predefined deterministic threshold value. By means of this approach the model learns a deterministic policy when to stop computation in a spatial position. This approach reconciles with the earlier mentioned "Unrolled Iterative Estimation View" (UIEW) presented by [11] which proposes that within a residual block the subsequent layers iteratively refine a feature map. Hence one could say that in SACT computation stops if the features are good enough.

Interesting is also the result that the so called computation time maps\(^3\) are well-correlated with human eye fixation positions which suggests that the model is implicitly learning an attentional mechanism. The work uses FLOPS\(^4\) as a measure to quantify the trade off between accuracy and computational speed.

In [21] the authors develop a dynamic computational time model to accelerate the average processing time for recurrent visual attention using a reinforcement learning approach. In addition to an optimal attention policy the model also learns a stopping policy. At each iteration the model attends to a more discriminative area and a binary stopping action indicates whether to continue or halt the recurrent fine-grained recognition task. Interesting for our work, the loss objective uses a negative delayed\(^5\) reward function and considers only training examples in which the last time step resulted in a correct classification of the image.

A compelling and more general approach to training a model that is able to dynamically adjust to new constraints introduced at inference time was presented by [23]. Using a reinforcement learning approach the model adaptively constructs computational graphs from sub-modules in an input dependent manner. Furthermore the model allows to augment the reward function of the controller to express a preference one has about its policy over computation graphs. This enables a practitioner to inject e.g. a preference for computational efficiency over accuracy at inference time. More general, the model’s behavior at test time can be changed by adding a constraint that has not been present at training time.

A very similar approach to ACT has been proposed in [15]. The work extends the basic Elman RNN unit [5] with a so called scheduler that decides at each time step which portion of the hidden state to change based on the current hidden state and input. This enables the model to adjust the computational effort performed at each time step to the sequence’s time structure which is the same objective that the ACT approach pursues.

---

\(^1\)A residual unit has the form \(F(x) = x + f(x)\) where the first term is referred to as shortcut connection and the second term is a residual function.

\(^2\)A residual block consists of stacked residual units.

\(^3\)Where the x and y axis indicate the spatial position and the z value equates to computational effort.

\(^4\)Floating Point Operations per Seconds

\(^5\)The loss does not consider the cumulative negative reward but only the last time step reward.
Finally the work of [6] proposes a Bayesian approach to ACT in the setting of 2D and 3D scene understanding. The authors present an unsupervised generative model based on the variational auto-encoder from [17] where the encoder (implemented as an RNN) sequentially infers the number of objects present in the scene i.e. it learns to choose the correct number of inference steps.

**Learning an optimization algorithm**

The research performed on learning optimization algorithms is relatively recent. Actually the research of Andrychowicz et al. was paralleled by the work of Malik and Li [19] which in contrast to [1] explores an approach to learning task *independent* optimization algorithms using reinforcement learning. The authors cast the problem of solving an optimization task as a finite-horizon Markov decision problem and hence any particular optimization algorithm corresponds to a policy. Applying the basic concepts of RL, an optimization algorithm that converges quickly is rewarded and those that do not are penalized. Formulating the task in this way enables the authors to apply existing guided policy search algorithms in order to train a model that learns an optimization algorithm. The results show that the learned optimizer outperforms hand-engineered optimizers on two optimization tasks.

In more recent work [20], Malik and Li extended their proposed framework to learning an optimization algorithm for solving high-dimensional stochastic optimization problems. The proposed model is trained in order to optimize a two-layer neural network performing image classification on MNIST. The experiments demonstrate that the learned optimizer can be easily transferred to new tasks like training neural nets for image classification on CIFAR-100 or the Toronto Faces Dataset. Furthermore the study explores the robustness of the proposed model to changes in the stochasticity of the gradients (by varying the minibatch size) and to larger neural network architectures (by increasing the number of input and hidden units).

A recent study by [3] which is related to the work of Andrychowicz et al., uses a meta-learning approach for training RNN optimizers to perform global optimization of black-box functions. Inspired by the Bayesian optimization (BO) framework the authors replace the raw time step loss used by the LSTM optimizer in equation (2.8), with the *expected posterior improvement* and *observed improvement* which should encourage the meta-learner to be more exploratory. The work is also interesting for us because it focuses on the issues of computation speed, horizon length and exploration versus exploitation trade offs. Their research reveals that the modified RNN optimizer is faster and tends to achieve better performance within the training horizon than the original meta-learner from [1] and the BO algorithms they compared with. The learned optimizer however underperforms against Bayesian optimization algorithms if evaluated on horizons that exceeded the training horizon. The authors think that this is caused by the fact that the RNN optimizer has not learned to explore these longer horizons.

Inspired by the approach of Andrychowicz et al., the work of [25] developed an LSTM optimizer that was applied to the setting of few-shot and few-classes learning. The proposed meta-learner was used to optimize simple convolutional neural networks performing image classification on the Mini-ImageNet dataset. Once more the model learns an update rule applied to the parameters of the base-learner but different to [1] the authors equate the parameters of the base-learner with the cell state of the LSTM and define parametric forms for the LSTM gates in order to enable the meta-learner to compute optimal parameter update values. In addition to the parameter update rule the meta-learner discovers good initialization for the base-learner parameters so they are suitable for fast gradient-based adaptation.

---

6 Training base-learners on regression tasks and optimizing two-layer neural networks performing a classification task.
In this chapter we describe in some detail the results that we obtained from the experiments conducted during this study. We assessed the performance of the models (LSTM optimizer, M-ACT and M-PACT) by means of three different tasks in which the meta-learners are trained to optimize loss objectives of different base-learners:

1. **Linear regression**: The base-learners are ten dimensional regression functions and the learned optimizers have to determine the regression coefficients by fitting a normal distribution to each of the 10 sampled data points. We will refer to this task as regression or convex regression hereafter.

2. **Non-convex linear regression**: The setting is identical to the one described above with the only difference that the models have to determine the regression coefficients by fitting a Student-t distribution (hereafter also referred to as t-distribution) to each data point which makes the problem non-convex and therefore more difficult. We will refer to this task as non-convex regression hereafter.

3. **Simple multilayer perceptron**: In the final task the base-learner is a simple multilayer perceptron performing image classification on the MNIST data set. We will relate to this optimization task with the shorthand *MLP task* hereafter.

We exploit the results to answer the research questions we pose in section 1.1. We divide this chapter in three parts.

The first part consists of a general description of the regression experiments and the specific experimental settings that we used to solely evaluate the LSTM baseline model. We started the experiments by assessing the performance of the baseline model on the convex, followed by the non-convex regression task. The results provide the answers to the research questions **RQ1.1**, **RQ1.2** and **RQ1.3**. We close the first part by summarizing our main insights and corroborate why we have chosen a certain configuration of the baseline model for the assessment of our newly developed models. In the second part we evaluate the performance of the M-ACT and M-PACT model on the two regression tasks and report the results by comparing their performance with the LSTM optimizer. Finally in the third part we outline the experiments and results we obtained when training the three meta-learners to optimize a small MLP on the MNIST data set. The experiments conducted in the second and third part reveal the answers to the research question **RQ2.1**, **RQ2.2** and **RQ2.3**.

### 5.1 General

In all experiments the optimizers used two-layer LSTMs with 40 hidden units. ADAM [16] was used to optimize the RNNs and in each experiment we performed random search to find the best learning rates. In order to determine the optimal final training epoch and hence prevent the models from overfitting, we evaluated the optimizers regularly during training. For the final evaluation we always tested the models on newly sampled instances and report the average performance. We implemented all models\(^1\) in *PyTorch*\(^2\) and ran the experiments on a single NVIDIA Titan X GPU\(^3\) and NVIDIA GTX 960M GPU.

\(^1\)https://github.com/toologicbv/meta_learner
\(^2\)http://pytorch.org/
\(^3\)on the DAS-4 cluster, http://www.cs.vu.nl/das4/
5.2 Optimizing regression functions

5.2.1 Linear regression

We used linear regression functions with ten parameters (denoted $\theta$)

$$f(x^{(n)}, \theta^*) = \sum_{j=1}^{M} \theta^*_j x^{(n)} + \epsilon,$$  

where $M = 10$, $x = (x^{(1)}, \ldots, x^{(N)})^T$ and $\theta^*$ denotes the true parameters to be determined by the optimizers. We added some fixed noise $\epsilon$ to the target value, where $\epsilon \sim N(0, \beta^{-1})$ and $\beta^{-1} = 0.01$.

For each regression function we sampled the true parameters $\theta^*$ from a uniform distribution $\theta^* \sim U(0,1)$ and the initial values of the parameters $\theta_0$ are sampled from a normal distribution $\mathcal{N}(0, \alpha^{-1})$ where $\alpha^{-1} = 0.01$. In order to determine the log-likelihood of the parameters we uniformly sampled $N = 10$ observations for each regression function $f$. The negative log-likelihood (NLL) of a base-learner when fitting a normal distribution to each of the $N$ data points, is calculated as follows

$$-\log p(f(x, \theta^*)|f(x, \theta_t), \beta^{-1}) = \frac{\beta}{2} \sum_{n=1}^{N} \left\{ f(x^{(n)}, \theta_t) - f(x^{(n)}, \theta^*) \right\}^2$$

$$- \frac{N}{2} \log \beta + \frac{N}{2} \log(2\pi).$$

At each time step $t$ the optimizers input were the gradients of the negative log-likelihood w.r.t. the optimizee’s parameters.

5.2.2 Evaluating the LSTM optimizer on convex regression task

The baseline model minimizes the expected negative log-likelihood for a fixed number of optimization steps $T$ given a distribution over functions $p(f)$

$$L_{\text{total}}(\phi) = \mathbb{E}_{p(f)} \left[ \sum_{t=1}^{H} - \log p(f(x, \theta^*)|f(x, \theta_t), \beta^{-1}) \right]$$

$$\tilde{L}_{\text{total}}(\phi) = \frac{1}{M} \sum_{m=1}^{M} \sum_{t=1}^{H} - \log p(f^{(m)}(x, \theta^*)|f(x, \theta_t), \beta^{-1}),$$
where \( H \) refers to the fixed horizon length and \( M \) denotes the size of the minibatch of functions \( f \) in order to compute the Monte Carlo (MC) estimates. We used a minibatch size of \( M = 125 \) for all models in the regression task and each epoch consisted of 10,000 newly sampled regression functions. As table 5.1 shows the model was trained for a different number of epochs depending on the experimental settings using a learning rate of 4e-5. We varied the training horizon and the sub-sequence length when performing truncated BPTT:

1. In order to gain an answer to research question RQ1.1 we trained the model for the following fixed horizons: 5, 10, 25, 50 and 100 time steps. When \( H \) was greater than 20 steps (25, 50 and 100) we used truncated BPTT and unrolled for 20 steps.

2. In this setting the model was trained with a fixed horizon of 50 steps and we varied the sub-sequence length during truncated BPTT using: 50, 20, 10 and 5 steps. The results of this experimental setting provide us with an answer to research question RQ1.2.

Table 5.1: Experimental settings in both regression tasks for the LSTM optimizer. Shown are different combinations of training horizon and truncated BPTT sub-sequence length. Column three resp. four specify the sum of optimization steps (in Thousands) during training, where each epoch consisted of 10,000 functions to be optimized. Numbers between brackets equate to training epochs.

<table>
<thead>
<tr>
<th>Training horizon (H)</th>
<th>sub-sequence length BPTT(L)</th>
<th>Sum of optimization steps (in Thousands) during training</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Regression</td>
</tr>
<tr>
<td>5</td>
<td>n/a</td>
<td>1,000 (20)</td>
</tr>
<tr>
<td>10</td>
<td>n/a</td>
<td>2,000 (20)</td>
</tr>
<tr>
<td>25</td>
<td>20</td>
<td>13,750 (55)</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>50,000 (100)</td>
</tr>
<tr>
<td>50</td>
<td>30</td>
<td>n/a</td>
</tr>
<tr>
<td>50</td>
<td>20</td>
<td>22,500 (45)</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>20,000 (40)</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>15,000 (30)</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>40,000 (40)</td>
</tr>
</tbody>
</table>

The first insight we gained from the regression experiment is that the sub-sequence length used in truncated BPTT has an effect on the speed of learning. This effect can be observed in figure 5.1a, 5.1b, 5.1c and 5.1d in which the LSTM optimizer was trained with a fixed horizon of 50 time steps (\( H = 50 \)) and the model was unrolled for four different sub-sequence lengths during BPTT, \( L \in \{50, 20, 10, 5\} \). Notice that the condition in which \( L = H \) (in our case 50) means that the model was trained without truncated BPTT. The figures reveal that a model unrolled during training for five time steps achieved a better optimization result during the initial time steps than a model unrolled for 10, 20 or 50 time steps. The training procedure seems to benefit from early error feedback. Table 5.1 also shows that the computational effort to train the models decreased with reduced sub-sequence length. Furthermore we found that the sub-sequence length has no effect on the final performance of the model and hence it did not seem to matter for the ultimate optimizer’s performance whether or not we used truncated BPTT during training. Please refer to figure 3a and 3b in appendix B for details.

Another observation we made is that longer training horizons do result in better optimization performance. Figures 5.2a and 5.2b show the performance of the model in the different experimental settings on 10,000 newly sampled regression functions for the first five time steps respectively for the interval [5,100]. We notice that for the horizons 25, 50 and 100 (using a sub-sequence length of 20) the performance diverged roughly around the 30th optimization step and longer training horizons resulted in a superior final minimum value. Column three in table 5.1 specifies the total number of optimization steps (introduced in section 3.5) the model utilized during training. It
Figure 5.2: Evaluation of all models on linear regression task: Each of the four rows comprises two figures that capture the complete time horizon from 0 to 100 steps. The first column shows the mean average loss for the first five time steps and the second column provides the same information for the time steps five to 100. Shaded regions denote ± one standard deviation. Note that the y axis is using a log scale for the negative log-likelihood (NLL). **Row 1** (5.2a, 5.2b): The LSTM baseline trained for a horizon of 5, 10, 25, 50 and 100 time steps. **Row 2** (5.2c, 5.2d): The M-ACT model trained with three different hyper-parameter settings, $\tau \in \{0.008, 0.0006, 0.0003\}$. **Row 3** (5.2e, 5.2f): The M-PACT model trained with three different shape parameter settings, $\nu \in \{0.7, 0.5, 0.1\}$. **Row 4** (5.2g, 5.2h): Comparing the performance of three LSTM optimizers with the best performing M-ACT and M-PACT model, please see legend for different settings of hyper-parameters and horizon lengths.
Non-convex linear regression

**Figure 5.3:** Non-convex regression task: Learning progress during training for LSTM baseline model using different sub-sequence lengths during truncated BPTT. Shown are mean time step losses up to step ten for the first 20 training epochs. A darker color corresponds to a lower loss value. Units for the y axis are epochs and time steps for the x axis. The LSTM model was trained for 50 time steps unrolling the LSTM for 50, 30, 20 and 10 time steps during truncated BPTT. The figures clearly show the effect of the number of truncated BPTT steps on learning.

should be mentioned that training for the shorter horizons of five and ten steps did not exceed 20 epochs because the model would otherwise overfit on the first time steps resulting in an unstable optimization behavior if the evaluation horizon exceeded the training horizon. In addition we found that during training the model quickly achieved its lowest minimum value for the final time step $T$ and subsequent training epochs contributed especially to performance gains in the first time steps. This effect is stronger when using longer sub-sequence lengths during truncated BPTT which is in line with our earlier results visualized in figure 5.1.

### 5.2.3 Non-convex linear regression

The non-convex linear regression task consisted of the same functions we specified in equation 5.1 but this time we fitted a $t$-distribution to each data point which makes the optimization problem non-convex and hence more difficult. Using the heavy-tailed $t$-distribution makes the estimation of the regression coefficients more robust e.g. to outliers. The negative log-likelihood term that we need to compute for the loss functions of the meta-learners is equal to

$$
-\log p(f(x, \theta^*) | f(x, \theta_i), \lambda, \nu_s) = -N \left[ \log \Gamma \left( \frac{\nu_s}{2} + 1 \right) - \log \Gamma \left( \frac{\nu_s}{2} \right) - \frac{1}{2} \log(\pi \nu_s) - \log \lambda \right] + \frac{\nu_s + 1}{2} \sum_{n=1}^{N} \log \left( 1 + \left( \frac{f(x^{(n)}, \theta_i) - f(x^{(n)}, \theta^*)}{\lambda \sqrt{\sigma_s} \nu_s} \right)^2 \right),
$$

(5.5)

where $\lambda$ denotes the scale parameter (we used the standard value of one) and $\nu_s$ the *degrees of freedom* parameter of the Student-$t$ distribution. We used a value of $\nu_s = 1$ because we fitted individual distributions to each of the ten data points.

### 5.2.4 Evaluating the LSTM optimizer on non-convex regression task

In this section we describe the experiments performed to assess the baseline model’s performance on the non-convex regression task. We conducted these experiments in order to determine whether the insights gained from the convex regression task (section 5.2.2) also hold in case the optimization problem is more demanding (also refer to the research question RQ1.3). Hence our experimental settings were the same as described in section 5.2.1 and again table 5.1 specifies the different combinations of training horizon and sub-sequence lengths that we evaluated. Column four specifies the total number of optimization steps the model was trained on. Please note that during the final assessment of the optimizer we used a fixed horizon of 200 steps. The learning rate was set to 4e-5.
In order to investigate the effect of the sub-sequence length during truncated BPTT on learning, we trained the model with a fixed horizon of 50 steps and varied the sub-sequence length \( L \in \{50, 30, 20, 10\} \). Figure 5.3 shows the first ten mean time step losses for the initial 20 training epochs. We can observe that using truncated BPTT is generally beneficial for learning but we also notice that the shortest sub-sequence length of ten did not perform better than using \( L \) equal to 20. This is not in line with our findings from the previous experiment where we found that the shortest sub-sequence length of five did achieve faster learning in the first epochs compared to the longer sub-sequences that we evaluated. Moreover we once more detected that the final performance of the model did not depend on the sub-sequence length used during training (for details see figures 3c and 3d in appendix B).

In order to assess the influence of the training horizon length on the optimizer’s performance we tested the baseline model on 10,000 freshly sampled regression functions using a fixed horizon of 200 time steps. Figure 5.4a shows the step interval \([0, 100]\) and 5.4b visualizes the time steps from \([100, 200]\). We observe once more that a longer training horizon converges to lower values in the final time step \( T \) and this effect seems to be slightly stronger compared to the less demanding convex linear regression problem. Models trained with very short horizons (five and ten steps) still performed surprisingly well on this task although one clearly sees that their performance is already slightly inferior in the first time steps. We detected difficulties in training the model with a fixed horizon of 10 steps. Although the learning curves of the optimizer looked smooth for this setting we spotted significant irregularities w.r.t. the validation losses. This finding is in line with our previous observation w.r.t. to the sub-sequence length of 10 which performed inferior during training compared to a sub-sequence length of 20.

5.2.5 Conclusions

We briefly summarize our gained insights from the evaluation of the baseline model on the two regression tasks and provide answers to the research questions formulated in RQ1.1, RQ1.2 and RQ1.3. Finding the regression coefficients of 10-dimensional regression base-learners by fitting a normal distribution to a data point is an easy task for the LSTM optimizer whereas the non-convex problem is clearly more demanding. Longer training horizons result in superior convergence behavior of the model especially when evaluated on horizons that exceed the length of the training horizon (RQ1.1). This effect is stronger for the more difficult non-convex optimization problem (RQ1.3). Although this is in line with what one would normally expect we carefully note that we also observed that the baseline model performed reasonable when trained for relatively short, fixed horizons (e.g. \( H = 5 \) for convex and \( H = 25 \) for non-convex regression) while evaluated on significantly longer horizons (\( H = 100 \) in the regression resp. \( H = 200 \) in the non-convex regression task). We think this insight is important because the newly developed models are able to adjust the training horizon per optimizee and can therefore explore the necessary horizon length dynamically.

In both experiments we discovered that the sub-sequence length used during truncated BPTT has an effect on the learning speed but does not influence the final model performance (RQ1.2). This insight is important because our newly developed meta-learners do not use truncated BPTT during training (we will return to this subject in section 6.2.1) and for a fair performance comparison we want to be sure that the baseline model is not impaired if we use an LSTM that is using truncate BPTT. We finally decided to compare the performance of our new, adaptive meta-learners against a baseline that uses a sub-sequence length of 20 steps during truncated BPTT\(^4\). We want to stress that this also implies that the LSTM optimizer can be trained with significantly less computational effort. Table 5.1 states that training a model with a fixed horizon of 50 steps using no truncated BPTT, took more than twice the amount of computational effort compared to an LSTM that was using truncated BPTT(20).

Furthermore we will always compare the performance of the new meta-learners against several LSTM optimizers that were trained for different fixed horizon lengths. We think this is helpful and necessary when we compare accuracy against computational effort for all three models.

\(^4\)If \( H \leq 20 \) then \( L = H \).
Figure 5.4: Evaluation of all models on non-convex linear regression task: Each of the four rows comprises two figures that capture the complete time horizon from 0 to 200 steps. The first column shows the mean average loss (NLL) for the first 100 time steps and the second column provides the same information for the time steps 100 to 200. Shaded regions denote ± one standard deviation. Note that the y-axis of the figures in the right column is using a log scale. **Row 1** (5.4a, 5.4b): The baseline model when trained with a fixed horizon of 5, 10, 25, 50 and 100 time steps. **Row 2** (5.4c, 5.4d): The M-ACT model when trained for three different hyper-parameter settings, $\tau \in \{0.009, 0.0018, 0.0008\}$. **Row 3** (5.4e, 5.4f): The M-PACT model trained with three different shape parameter settings, $\nu \in \{0.5, 0.3, 0.05\}$. **Row 4** (5.4g, 5.4h): Comparing the performance of three LSTM optimizers with the best performing M-ACT and M-PACT model, please see legend for different settings of hyper-parameters and horizon lengths.
5.3 Evaluating M-ACT and M-PACT model on linear regression tasks

In this section we describe the experiments that assessed the performance of the M-ACT and M-PACT model on both regression tasks. Moreover we compared the results with the performance of the baseline model and hence these experiments are giving answers to the research questions RQ2.1 to RQ2.4. We used the same experimental settings as before i.e. during training each epoch contained 10,000 freshly sampled regression functions using a minibatch size $M$ of 125. Both models used a learning rate of $5e^{-5}$ and we did not use truncated BPTT during training.

In section 3.2 we formalized the loss function of the M-ACT optimizer in equation 3.6. Replacing the step loss $f^{(m)}(\theta_t)$ with the negative log-likelihood, we can rewrite the approximated, expected loss as:

$$L^{total}(\phi) \approx \tilde{L}^M(\phi) = \frac{1}{M} \sum_{m=1}^{M} \sum_{t=1}^{T(f^{(m)})} \left( \frac{-1}{p_t} \log p(f^{(m)}(x, \theta^*) | f(x, \theta_t), \beta^{-1}) + \tau \left( \tau^{(f^{(m)})} + R(f^{(m)}) \right) \right).$$

We approximated the expected loss for the M-PACT model based on minibatches as specified in equation (3.35) and we replaced the log likelihood term in equation (3.33) with the formalization we gave in equation (5.2) for the convex regression, respectively (5.5) for the non-convex regression task. We trained the M-PACT model with ten different values for the shape parameter $\nu$ of the prior distribution $\{0.0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$ and kept three settings (among them the best performing $\nu$ value) and report their results here.

We described in section 3 that both models have the flexibility to adjust the number of optimization steps $T$ per optimizee during training whereas the LSTM optimizer uses a fixed horizon. Throughout evaluation we tested all models on a fixed number of optimization steps i.e. 100 for the convex and 200 for the non-convex regression task. In table 5.3 we report the so called halting step statistics that we obtained during the final test run although the functions were optimized for a fixed number of steps. In this context the term "halting step" denotes the last optimization step $T$ after which the model would have normally stopped computation for a specific function. This step is determined for the M-ACT model based on equation (3.4) and is therefore equal to the condition used during training. However the M-PACT model used a stochastic threshold during training which we replaced for evaluation purposes in favor of the same deterministic threshold that we utilized for the M-ACT model.

We will report the results per regression task for both models. We look into the following subjects to summarize our findings:

- **Progression of learning** based on the achieved mean step losses in each training epoch. Moreover these figures will give us some insights into the exploration behavior of the models w.r.t. the sequence length.
- **Learning curves of the optimizees** that were acquired while the models had to optimize 10,000 newly sampled functions using a fixed time horizon (100 steps in the regression resp. 200 steps in the non-convex regression task).
- **Ability of adapting** the computational effort to the difficulty of the optimizees.
- **Distribution of trajectory weights** obtained during final evaluation which both models exploited to determine the halting step for an optimizee.
5.3.1 Results on convex linear regression

In this section we will describe the results obtained in the (convex) regression task when evaluating the new models.

Figure 5.5 visualizes the learning progress of both models during training on the linear regression task. The first row (figures 5.5a, 5.5b, 5.5c) shows the mean loss (NLL) per time step (x axis) per epoch (y axis) for the M-ACT model where each figure reveals this information for a different value of $\tau$. The same information is displayed for the M-PACT model in the bottom row (figures 5.5d, 5.5e, 5.5f) but then for the three different values of the shape parameter $\nu$. As expected, we observe that the M-ACT model explored significantly longer sequences with lower values for the time penalty parameter $\tau$ (lower $\tau$ values correspond to less regularization). It is interesting to mention that the model is resorting to shorter time horizons after having explored longer sequences as is visualized in figure 5.6a. The effect can be also detected for the M-PACT($\nu=0.1$) model in figure 5.6b but is absent for the stronger regularized models.

Furthermore we notice that the M-ACT model explored very short sequences (4 to 5) in the first 20 epochs where in contrast the M-PACT model immediately started with a sequence length above 10. Moreover we perceive that the M-ACT model quickly learned to take larger leaps in the parameter space of the optimizee than the M-PACT model (darker colors within the first five epochs). This is in line with our previous findings w.r.t. the effect of the sub-sequence length used during truncated BPTT when training the LSTM optimizer. Due to the simplicity of the task the M-ACT model could benefit from this behavior.

Figures 5.2c and 5.2d respectively 5.2e and 5.2f display the mean optimizer’s step losses achieved by the M-ACT resp. M-PACT models during final evaluation and visualize the effect

**Figure 5.5**: Learning progress during training for M-ACT and M-PACT model. Shown are mean step losses (NLL) per training epoch (y axis). Darker colors equate to lower loss values. Units of the x axis are time steps. **Row 1 (5.5a, 5.5b, 5.5c):** M-ACT, mean step losses per epoch for the different time-penalty settings $\tau = \{0.008, 0.0006, 0.0003\}$. **Row 2 (5.5d, 5.5e, 5.5f):** M-PACT, mean step losses per epoch for the different shape parameter settings $\nu = \{0.7, 0.5, 0.1\}$. 

![Regression](image)
Figure 5.6: Convex regression: Mean halting step during training. Units of $x$ axis are halting steps and $y$ axis units are training epochs. Shaded regions denote ± one standard deviation. Both figures show the effect of regularization on the model’s behavior during training. Large regularization results in less exploration in contrast to less penalization which encourages the model to explore longer time sequences.

We compare the mean learning curves of the optimizees for the best M-ACT ($\tau=0.0003$) and M-PACT ($\nu=0.7$) model with the baseline model in figure 5.2g and 5.2h where the LSTM optimizer is trained for a fixed horizon of 5, 50 and 100 steps. Except for the baseline model denoted LSTM($H=5$), all models performed roughly the same in the first five time steps but differ in their convergence behavior during time step 5 to 100. One can spot a tiny but discernible lower minimum that the baseline model achieved when trained with a fixed horizon of 50 and 100 steps. The M-ACT and M-PACT optimizers performed nearly similar.

Table 5.2 specifies the exact mean loss values during evaluation at time step 1, 5, 30, 50 and 100. The second column reports the mean training horizon and the third column of the same table states the total number of optimization steps the models were trained on. We can detect that the computational effort that the best M-PACT model used, is only a fraction greater than the one utilized by the LSTM($H=5$) optimizer. As we explained previously, we computed halting step statistics when evaluating the newly proposed models and table 5.3 states those statistics for both regression tasks. We observe a clear effect of the time penalty and shape parameter on the halting step statistics. As expected stronger regularized models would be computationally more parsimonious during inference but the last column of table 5.3 also states that the mean loss value of the last time step would be too large, especially for the M-PACT model. Therefore this result already indicates that it is quite improbable that we can use the halting step at test time (also refer to research question RQ2.4).
Regression

M-ACT: Input dependency

M-PACT: Input dependency

M-ACT: Distribution of trajectory weights

M-PACT: Distribution of trajectory weights

Figure 5.7: Regression task: Visualization of input dependency and trajectory weights for M-ACT and M-PACT model at test time. Each column shows a different setting for the hyper-parameter $\tau$ resp. shape parameter $\nu$. Row 1 (5.7a to 5.7c): Input dependency for three M-ACT models. Row 2 (5.7d to 5.7f): Input dependency for three M-PACT models. Row 3 (5.7g to 5.7i): Distribution of trajectory weights for three M-ACT models. Row 4 (5.7j to 5.7l): Distribution of trajectory weights for three M-PACT models. Please note that (1) the distributions are averaged over different trajectory lengths; (2) the orange histogram visualizes the corresponding truncated prior geometric distribution.
Table 5.2: Comparison of computational effort versus accuracy for three LSTM baseline models and the best M-ACT and M-PACT model on (convex) regression tasks. The second column specifies the mean training horizon and the third column shows the total number of optimization steps executed during training (each epoch consisted of 10,000 functions). Column four to eight specify various mean step losses during the final evaluation. The last column shows the loss value for the mean halting step determined for the adaptive meta-learners using linear interpolation.

<table>
<thead>
<tr>
<th>Model</th>
<th>Mean training horizon</th>
<th>Optimization steps during training</th>
<th>Evaluation</th>
<th>Mean time step losses (NLL)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>LSTM (H=5)</td>
<td>5</td>
<td>1,000,000</td>
<td>20.72</td>
<td>12.22</td>
</tr>
<tr>
<td>LSTM (H=50)</td>
<td>50</td>
<td>22,500,000</td>
<td>17.58</td>
<td>11.99</td>
</tr>
<tr>
<td>LSTM (H=100)</td>
<td>100</td>
<td>30,000,000</td>
<td>17.48</td>
<td><strong>11.95</strong></td>
</tr>
<tr>
<td>M-ACT ($\tau=0.0003$)</td>
<td>13.18</td>
<td>6,812,364</td>
<td>17.94</td>
<td>12.26</td>
</tr>
<tr>
<td>M-PACT ($\nu=0.7$)</td>
<td>2.14</td>
<td>1,096,420</td>
<td><strong>17.37</strong></td>
<td><strong>11.95</strong></td>
</tr>
</tbody>
</table>

Input dependency

In order to explore the question whether our models learned to adjust the computational effort (measured as the halting step during evaluation) to the difficulty of the objective function we analyze figure 5.7a to 5.7f. Please note that we will also refer to this question as whether a model exhibited an input dependent optimization behavior. In all six figures the units of the $x$ axis are halting steps and the $y$ axis indicates the negative log-likelihood value. Each "red dot" represents an individual objective function (the models had to optimize 10,000 functions during the final test). The first three figures 5.7a, 5.7b and 5.7c visualize the input dependency for the different hyper-parameter settings of the M-ACT model and we observe that the model used a larger range of halting steps with less regularization. In all three settings we can detect that the model would use more computational steps for objective functions that have a higher initial NLL value. Purely by visual inspection the results suggest that a value of $\tau=0.0006$ results in the best computational adaptive behavior because we notice only higher NLL values for later halting steps. Both effects are also visible for the M-PACT model in figure 5.7d, 5.7e and 5.7f and a stronger regularized models (5.7d) seem to exhibit a better input dependency.

Trajectory weights

Finally figure 5.7g to 5.7l present the distributions of the so called trajectory weights (denoted $p(t)$ for the M-ACT and $q(z|x,T)$ for the M-PACT optimizer) that the models generated during the final evaluation. We would like to mention that the M-PACT model is more constrained than the M-ACT model to choose these weights because the former is forced to adjust the weights to the prior distribution (a larger value for the the shape parameter $\nu$ equates to a stronger force). We observe that the M-PACT optimizer accommodated well to these constraints for the larger regularization values of $\nu \in \{0.7, 0.5\}$. It seems that the best fit is achieved with a shape parameter value of 0.5. As expected we see that for the less regularized model the posterior starts to deviate more strongly from the prior distribution. Lower parameter values resulted in a larger support of the distribution which is in line with the halting step statistics that we reported earlier.

Figure 7b in appendix F corroborates our findings by visualizing the halting distributions of the M-PACT($\nu=0.7$) model for three different horizon lengths obtained at test time. Figure 5 in appendix E visualizes the corresponding marginal posterior distributions $q(z|x)$ that we obtained at test time. Comparing the marginal with the conditional distribution (orange histograms in fig. 5) we detect that the former accumulates more probability mass in the first time steps although the overall shape of both distributions looks very similar.

The M-ACT model generated different distributions especially for the lower hyper-parameter values (fig. 5.7h and 5.7i). The mode of the distributions is located shortly before the mean
Table 5.3: Halting step statistics for M-ACT and M-PACT model on regression tasks. Bold names indicate the models that we used in the final comparison with the baseline model.

<table>
<thead>
<tr>
<th>Evaluation</th>
<th>Regression Halting step statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>(hyper-parameter)</td>
</tr>
<tr>
<td>M-ACT (τ=0.008)</td>
<td>[2, 5]</td>
</tr>
<tr>
<td>M-ACT (τ=0.0006)</td>
<td>[14, 30]</td>
</tr>
<tr>
<td>M-ACT (τ=0.0003)</td>
<td>[20, 33]</td>
</tr>
<tr>
<td>M-PACT (ν=0.1)</td>
<td>[11, 38]</td>
</tr>
<tr>
<td>M-PACT (ν=0.5)</td>
<td>[7, 11]</td>
</tr>
<tr>
<td>M-PACT (ν=0.7)</td>
<td>[7, 9]</td>
</tr>
</tbody>
</table>

halting step (see table 5.3) and the shape of the histograms suggest that the model accumulates most of its probability mass just before the halting step of an optimizee. This is corroborated by figure 7a in appendix F which visualizes the halting distributions of the M-ACT(τ=0.0003) model for four different horizon lengths obtained at test time. The distributions clearly show an inclined slope towards the last time step. The time step weights in the first half of the trajectory have approximately constant values with the exception of the first time step. It seems that the regularization term works in the intended way i.e. it counteracts the natural tendency of the model to accumulate all probability mass on the last time step.

5.3.2 Conclusions

Before we continue with a detailed outline of the results that we obtained on the non-convex regression task, we briefly summarize our insights from the previous experiment and augment them with conclusions that contribute to the research questions RQ2.1 to RQ2.4. The results show that both models (M-ACT and M-PACT) achieved optimization results that were comparable to the baseline model. We admit that compared to the LSTM optimizer trained with long horizons (H ∈ {50, 100}) the performance of our models is slightly inferior in the last time steps. With the current results we cannot conclude whether our models can be trained faster than the baseline and we assume that the task is too simple to make a difference. We do observe that both models learn a certain degree of input dependency and that they exploit this information to adapt the necessary number of optimization steps. The behavior of both models during training reveals clear influences of the injected time penalty, where lower regularization resulted in the exploration of longer time sequences and in most of the cases the models were able to resort to shorter trajectories towards the end of training. We already concluded that the results indicate that we cannot exploit the halting step at test time because on average the base-learners did not converge at those time steps.

5.3.3 Results on non-convex linear regression

In this section we outline the results that we obtained for both adaptive optimizers while assessing their performance on the non-convex regression task. We cover once more the subjects listed at the beginning of the previous section to convey the main results. Please note that the evaluation procedure was identical to the previous experiment with the only difference that we optimized
Non-convex regression

Figure 5.8: Non-convex regression task: Training progress of M-ACT and M-PACT. Shown are mean step losses (NLL) per training epoch (y axis). Darker colors equate to lower loss values. Units of the x axis are time steps. **Row 1 (5.8a, 5.8b, 5.8c):** M-ACT, mean step losses per epoch for the different hyper-parameter settings $\tau \in \{0.009, 0.0018, 0.0008\}$. **Row 2 (5.8d, 5.8e, 5.8f):** M-PACT, mean step losses per epoch for the different shape parameter settings $\nu \in \{0.5, 0.3, 0.05\}$. 10,000 freshly sampled regression functions for a fixed horizon of 200 time steps (instead of 100 steps in the convex case). The optimization task is more demanding and hence it is interesting to assess the performance of the models in time steps that significantly exceeded the horizon they were able to explore during training.\(^6\)

We can inspect the learning progress of both models in figure 5.8 which contains in the top row the achieved mean step losses per training epoch for the three different hyper-parameter settings of the M-ACT model (fig. 5.8a, 5.8b, 5.8c). The bottom row displays the same information for the three different shape parameter settings of the M-PACT model (fig. 5.8d, 5.8e, 5.8f). In addition figure 5.9 visualizes the progression of the halting step histograms during training. The figure contains once more six sub-figures and has the same grouping we just stated for the previous figure.

It is interesting to notice that the mean step losses during training look very different for the two optimizers especially if we compare the results for the M-ACT($\tau=0.0018$) in figure 5.8b with the M-PACT($\nu=0.5$) model in figure 5.8d. The results suggest that the M-PACT model’s performance is significantly inferior compared to the M-ACT optimizer because if one inspects e.g. the last epoch the M-ACT model achieves obviously lower mean loss values at time step 20 compared to the M-PACT model for the same time step. At test time the performance of both models is very similar as the results in table 5.4 reveal. Taking into account the halting step histograms of both models in figure 5.9b and 5.9d we can detect that the halting step distributions (in the last epoch) are significantly dissimilar for both models (please also refer to figure 5.11 for a different horizon).

\(^6\)Note that due to the hard limit of 100 steps during training, the models cannot explore trajectories beyond this horizon.
Non-convex regression

Figure 5.9: Non-convex regression: Progression of halting step distribution during training. Shown are histograms of halting steps per training epoch for six different models (reader is looking from above). Darker colors equate to lower values. **Row 1** (5.9a, 5.9b, 5.9c): M-ACT, halting step histograms per epoch for the different hyper-parameter settings $\tau \in \{0.009, 0.0018, 0.0008\}$. **Row 2** (5.9d, 5.9e, 5.9f): M-PACT, halting step histograms per epoch for the different shape parameter settings $\nu \in \{0.5, 0.3, 0.05\}$. One can observe how both models shift the attention from shorter to longer time horizons and back.

Visualization of the same information). The mean loss value at time step 20 is generated by only a few optimizees in case of the M-PACT model whereas the M-ACT model trained all optimizees for at least 20 time steps. This does not entirely explain the performance of the models at test time but it does reveal that we need to be careful when interpreting the mean step losses during training.

In line with our expectation and the previous experiment we perceive a clear inverse relationship between the degree of regularization and the length of the explored time sequences when inspecting the progression of the halting step distributions (fig. 5.9). Furthermore we see that the non-convex regression task is more demanding because both models have the tendency to explore longer sequences compared to the convex regression task although the degree of constraint is similar (e.g. for the M-PACT($\nu=0.5$) model fig. 5.5e, 5.8d). Besides we observe once more that the models were able to resort on average to shorter time horizons after having explored longer sequences (e.g. fig. 5.9b, 5.9d, 5.9e). Especially for the M-PACT optimizer we observe that the model reduced the variance of the halting steps significantly towards the end of training. The effect is generally smaller for stronger regularized models. It seems that in most of the settings the optimizers were able to shift the attention from shorter to longer time horizons and back.

The lower part of table 5.3 summarizes the halting step statistics that we collected during final evaluation of the models. The data confirms the effect of the time penalty and shape parameter we described above. Please note as well that the range and standard deviation increased significantly when the models were less constrained. We return to this observation when describing the results w.r.t. the level of input dependency the models were able to learn.
In order to visualize the optimization performance of the models in the non-convex regression task on the 10,000 test functions, we again split the mean step losses generated by the optimizers into two separate time step intervals \([0,100]\) and \([100,200]\). Figures 5.4c and 5.4d respectively display the mean learning curves (of the optimizees) achieved by the M-ACT resp. M-PACT model for the different hyper- and shape parameter values. Large time-penalties for the M-ACT model resulted in inferior optimization performance after the fifth time step whereas low penalization led to deficiencies in later time steps (100 to 200). The shape parameter \(\nu\) seemed to have little effect in the first hundred time steps but was noticeable in later time steps. A larger \(\nu\) value equates to higher regularization and can be interpreted as making the model more myopic w.r.t. to later time steps. Nevertheless figure 5.4f reveals that higher regularization \((\nu=0.5)\) did not lead to inferior performance in later time steps.

Table 5.4: Comparison of computational effort versus accuracy for six LSTM baseline models and the best M-ACT and M-PACT model on the non-convex regression task. The second column specifies the mean training horizon. The third column shows the total number of optimization steps executed during training (each epoch consisted of 10,000 functions). Column four to eight specify various step losses during the final evaluation. The last column shows the loss value for the mean halting step determined for the adaptive meta-learners using linear interpolation. Please note that the results which are stated below the dashed horizontal line were obtained when training three LSTM models for exactly two million optimization steps which was roughly equivalent to the computational effort used to train the best M-PACT model on this task.

<table>
<thead>
<tr>
<th>Model</th>
<th>Mean training horizon</th>
<th>Optimization steps (training)</th>
<th>1</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>E[T]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM (H=25)</td>
<td>25</td>
<td>11,250,000</td>
<td>42.21</td>
<td>28.63</td>
<td>22.0</td>
<td>20.26</td>
<td>18.98</td>
<td>n/a</td>
</tr>
<tr>
<td>LSTM (H=50)</td>
<td>50</td>
<td>25,000,000</td>
<td>41.98</td>
<td>28.63</td>
<td>21.86</td>
<td>19.7</td>
<td>18.09</td>
<td>n/a</td>
</tr>
<tr>
<td>LSTM (H=100)</td>
<td>100</td>
<td>30,000,000</td>
<td>42.2</td>
<td>28.72</td>
<td>21.69</td>
<td>19.22</td>
<td>17.19</td>
<td>n/a</td>
</tr>
<tr>
<td>M-ACT ((\tau=0.0018))</td>
<td>15.65</td>
<td>7,296,844</td>
<td>43.21</td>
<td>29.61</td>
<td>22.14</td>
<td>20.2</td>
<td>18.92</td>
<td>24.59</td>
</tr>
<tr>
<td>M-PACT ((\nu=0.5))</td>
<td>3.85</td>
<td>1,982,095</td>
<td>42.58</td>
<td>29.04</td>
<td>22.35</td>
<td>20.49</td>
<td>19.16</td>
<td>27.75</td>
</tr>
<tr>
<td>LSTM (H=5)</td>
<td>5</td>
<td>2,000,000</td>
<td>42.55</td>
<td>30.01</td>
<td>23.61</td>
<td>22.4</td>
<td>21.54</td>
<td>n/a</td>
</tr>
<tr>
<td>LSTM (H=10)</td>
<td>10</td>
<td>2,000,000</td>
<td>45.35</td>
<td>29.13</td>
<td>23.54</td>
<td>22.16</td>
<td>20.91</td>
<td>n/a</td>
</tr>
<tr>
<td>LSTM (H=20)</td>
<td>20</td>
<td>2,000,000</td>
<td>51.86</td>
<td>30.88</td>
<td>23.25</td>
<td>21.47</td>
<td>20.07</td>
<td>n/a</td>
</tr>
</tbody>
</table>

The bottom row of figure 5.4 compares the optimization performance of three LSTM optimizers (trained for a fixed horizon of 25, 50 and 100) with the best M-ACT \((\tau=0.0018)\) and M-PACT \((\nu=0.5)\) model \(\text{fig. 5.4g, 5.4h}\). Except for the M-ACT model which performed slightly inferior all other models achieved nearly the same performance during the first 40 steps. In the consecutive time steps one can clearly see that the baseline models trained with a fixed horizon of 50 and 100 steps attained lower mean loss values and the advantage slowly increased over time. In the final time step the computation time adaptive models roughly obtained the same performance which is comparable to the baseline model when trained with a horizon of 25 steps. Table 5.4 specifies the exact mean loss values during evaluation at time step 1, 10, 50, 100 and 200. The third column of the same table states the total number of optimization steps the models were trained on. We note that the computational effort utilized by the best M-ACT resp. M-PACT model was significantly less than the computation consumed by the baseline model trained with a fixed horizon of 25 time steps.

For comparison purposes, we also included the evaluation results of three LSTM models that were trained for exactly two million optimization steps (please refer to the three rows below the dashed line in table 5.4). This is roughly equivalent to the computational effort that we used to train the best M-PACT model. We trained the models with a fixed horizon of 5, 10 and 20 steps and varied the number of epochs. If we compare the performance of these models with the
M-PACT optimizer we notice that the baseline models performed inferior but we concede that especially the LSTMs trained with a fixed horizon length of 5 and 10 steps achieved good results when taking into account the limited computational effort it took to train the models.\footnote{Please note that the result for the LSTM($H=10$) does not contradict our earlier findings because as we noted then the model showed significant irregularities w.r.t. the validation losses when it was trained until convergence, whereas in this setting it was only trained for 20 epochs.}

### Input dependency

The scatter plots in figure 5.10a to 5.10f visualize the dependency between the initial negative log-likelihood value of an optimizee ($y$ axis) and the halting step ($x$ axis) determined by the adaptive meta-learners at test time. The data is shown for 10,000 regression functions when evaluation the M-ACT (fig. 5.10a to 5.10c) resp. M-PACT (fig. 5.10d to 5.10f) model. Compared to the results from the previous experiment (convex regression, fig. 5.7) we notice that the correlation between these measures has decreased and that this decline is stronger in the case of the M-PACT model. Nevertheless we stress that one can still observe a relationship between both measures. Furthermore the results suggest that models with less regularization seem to get lazy w.r.t. their objective i.e. it appears as if these models did not use the extra computational time steps to optimize the more difficult optimizees, they just used it. Hence we perceive these models as being inferior w.r.t. their ability to adapt the number of computational steps to the difficulty of the objective function.

### Trajectory weights

Finally we inspect once more the distribution of the trajectory weights that we obtained during the final evaluation of the adaptive meta-learners on the non-convex regression task. Figures 5.10g to 5.10l show the distributions for the three different hyper-parameter settings of the M-ACT model. We notice that the form of these distributions is very similar to the ones we saw in the convex regression experiment. The mode of the distributions is located shortly before the mean halting step (see table 5.3) and the overall shape of the distribution looks very similar to a normal distribution although clearly the tail to the right of the mean gets increasingly thicker with less regularization. Figure 8a in appendix F visualizes the halting distributions of the M-ACT($\tau=0.0018$) model for four different horizon lengths obtained at test time. The figures are in line with the results we obtained in the convex regression task. The distributions clearly show an inclined slope towards the last time step.

The distributions we obtained for the M-PACT optimizer are significantly different than the ones we observed in the convex regression experiment. Furthermore we clearly see an effect of the degree of regularization in this setting (fig. 5.10j to 5.10l). We observe that especially in the setting where we trained the model with a more farsighted prior ($\nu=0.3$ and $\nu=0.05$) the approximated posterior distribution deviates significantly from the form of the corresponding prior. Figure 8b in appendix F visualizes the halting distributions of the M-PACT($\nu=0.5$) model for four different horizon lengths obtained at test time. The mode of the distribution shifts from earlier to later time steps when the trajectory length increases. This behavior expresses again the tendency of the optimizer to shift more weight to later time steps because this can decrease the overall loss if the regularization penalty is not too high.

Figure 6 in appendix E visualizes the marginal posterior distributions $q(z|x)$ that we obtained at test time. Comparing the marginal with the conditional distribution (orange histograms in fig. 6) we clearly observe that the former accumulates more probability mass in earlier time steps. Equivalent to the results in the convex regression task, the overall shape of the posterior and conditional distribution is very similar.

### 5.3.4 Conclusions

Prior to reporting the results of our last experiment we once more summarize the insights gained from the evaluation of the adaptive meta-learners on the non-convex regression task. The results suggest that the ACT optimizers can be trained with substantially less computational effort (RQ2.1) while still achieving reasonable optimization performance in the non-convex task.
Figure 5.10: Evaluation of M-ACT and M-PACT model on non-convex regression task. Visualization of input dependency and trajectory weights for M-ACT and M-PACT model on non-convex regression tasks during testing. Each column shows a different setting for the hyper-parameter $\tau$ resp. shape parameter $\nu$. Row 1 (5.10a to 5.10c): Input dependency for three M-ACT models. Row 2 (5.10d to 5.10f): Input dependency for three M-PACT models. Row 3 (5.10g to 5.10i): Distribution of trajectory weights for three M-ACT models. Row 4 (5.10j to 5.10l): Distribution of trajectory weights for three M-PACT models. Please note that (1) the distributions are averaged over different trajectory lengths; (2) the orange histogram visualizes the corresponding truncated prior geometric distribution.
we concede that the optimization performance is slightly inferior in later time steps compared to the LSTM optimizers trained with longer fixed time horizons \((H \in \{50, 100\})\). Nevertheless especially the results of the M-PACT model indicate that this optimizer does well in trading off computational effort versus accuracy. We cannot achieve the same results when training the baseline model with the same computational effort (we refer the reader to the last three rows of table 5.4).

The effect of the regularization term is significant for both models and inverse proportional i.e. stronger regularization results in shorter mean average training horizons (column two table 5.4) and hence less computational effort \((RQ2.3)\). Furthermore we conjecture that both ACT meta-learners are exploring longer time horizons after having mastered shorter sequences and finally resort again to more parsimonious horizons at the end of training (fig. 5.11). Strong regularization can suppress the exploration behavior whereas too little time-penalties can impede final model performance by making it slightly lazy \((RQ2.3)\). Finally the results reveal that the M-ACT and M-PACT optimizers exhibit a certain degree of input dependent optimization behavior i.e. they adjust the number of optimization steps to the difficulty of the objective function \((RQ2.2)\).

5.4 Optimizing a small neural network on MNIST

Andrychowicz et al. had shown \([1]\) that the LSTM optimizer could learn to optimize a small neural network on MNIST. Hence in our final experiment we assessed the performance of all three optimizers on this task. The base-learner is identical to the one used in \([1]\) which is a simple MLP with one hidden layer of 20 units using a sigmoid activation function. The error surface of the optimizee is the cross-entropy loss approximated by means of minibatches using the MNIST dataset. We discovered that it was necessary to scale the KL-divergence term in (3.35) with a factor of 0.03 (which we refer to as hyper-parameter \(\lambda\) hereafter) in order to size the regularization term appropriately. The loss objective of the baseline and the M-ACT model as specified in equation (5.3) resp. (5.6) did not change. The log-probability term in all equations was replaced by the cross-entropy loss. The configuration of the optimizers w.r.t. the number of hidden layers and units remained unchanged.

The simple MLP has roughly 16,000 parameters (including biases) and because we were limited in our computational resources we were forced to optimize one MLP at a time i.e. we could not train the optimizers by means of minibatches. Hence during training we had to deal with noisy updates of the optimizer parameters especially for the two adaptive computation time models. We
Figure 5.12: Evaluation of all models on the MLP task. Comparing the performance of two LSTM optimizers with the best performing M-ACT and M-PACT models, please refer to the legend for the different hyper-parameter settings and horizon lengths. All four models were tested on 100 newly sampled MLPs for a fixed horizon of 200 time steps. All three figures show the mean average loss values (logits) for a different time interval: 5.12a: [0,30], 5.12b: [30,100] and 5.12c: [100,200]. Shaded regions denote ± one standard deviation. Note that the y axis is using a log scale and that e.g. LSTM($H=50$) denotes the baseline model trained with a fixed horizon of 50 steps.

We trained the models with learning rate decay and averaged the optimizer loss over 15 optimizees where the last measure did not make a big difference and was later replaced by one MLP.

Neural networks are known to disregard small variations in input signals and instead concentrate on bigger input values. The input signal to our optimizers are the gradients w.r.t. the different optimizee parameters which can exhibit extremely varying magnitudes and hence the models are prone to ignore small variations. In order to tackle this problem we followed the advise of [1] by preprocessing the optimizee gradients as described in appendix C\textsuperscript{8} and rescaling the output of the RNNs by a constant factor of 0.1.

We also found that adding the raw optimizee parameter values to the optimizers input improved the final optimization result. We experimented with the bias terms $b_p$ in equation (3.31) resp. $b_h$ in equation (3.3) and observed that the training procedure for the M-ACT model would improve if $b_h$ was initialized to a negative value (we used -1). Due to this initialization the model took less epochs before it would explore longer sequences. During each epoch a model optimized 16 MLPs and approximated the error surface of the optimizees using a minibatch size of 128 images taken from the MNIST training set. We utilized the MNIST test set to validate the optimizers on freshly sampled base-learners. Please refer to table 5.5 for detailed information about the learning rates that were used for training the models.

Finding the most suitable hyper-parameter value for the M-ACT model (τ) was a subtle task because the final performance was very sensible to small changes of the value. As described earlier, although our two newly proposed models are able to adjust the optimization trajectory length during training we used a hard limit on the maximum horizon $H$ as stated in equation (3.4) and (3.34) to prevent the model from utilizing large amounts of computational resources. We discovered an effect of this limit when training the M-ACT model which we will describe in the result section below. Here we note that we set $H$ to 100 for the M-PACT model but used a value of 125 (which was never reached) for the M-ACT model.

In the next section we report the results for the best performing M-ACT and M-PACT model on the MLP task and compare their performance with two LSTM optimizers that were trained for a fixed horizon of 50 and 100 time steps. The first column of table 5.5 shows the hyper and shape parameter settings for the adaptive meta-learners. Also note that in the same column the horizon setting $H$ for the LSTM optimizer refers to the fixed training horizon. The final performance was assessed on 100 freshly sampled MLPs using a fixed horizon of 200 time steps. Note again that the adaptive meta-learners were also evaluated for 200 steps although during training they were able to adjust the sequence length of an optimizee.

\textsuperscript{8}Taken from [1].
MLP optimization task

Mean learning curve (of optimizees) per training epoch

![Loss per time step / M-ACT \( \tau = 0.003 \)](image)

(a) M-ACT \( \tau = 0.003 \)

![Loss per time step / M-PACT \( \nu = 0.1 \)](image)

(b) M-PACT \( \nu = 0.1 \)

Halting step histogram per training epoch

![Halting step histogram / M-ACT \( \tau = 0.003 \)](image)

(c) M-ACT: \( \tau = 0.003 \)

![Halting step histogram / M-PACT \( \nu = 0.1 \)](image)

(d) M-PACT: \( \nu = 0.1 \)

Figure 5.13: Training progress of best performing M-ACT and M-PACT models on MLP task. Shown are in row 1 (5.13a, 5.13b): mean step loss (logits) per training epoch \( (y \text{ axis}) \) and in row 2 (5.13c, 5.13d): histograms of halting steps per training epoch for two different models. Darker colors equate to lower values. Units of the \( x \text{ axis} \) are time steps. As a reader you are looking from above.
Unfortunately we were not able to investigate how well the M-ACT and M-PACT models could adjust to the difficulty of the task. By this we mean, that the only source of variability between the different base-learners is the initial parameter value $\theta_0$ and the selection of the (image) minibatches during training. Due to the fact that our optimizee loss surface is the cross-entropy function with ten classes the initial loss value is always approximately the same ($\approx \ln(10)$) and hence unusable as a measure for the difficulty of the task.

5.4.1 Results

We start this section with the inspection of the mean step losses obtained during the evaluation of the models on 100 MLPs for 200 optimization steps (please note that we also refer to this figures as learning curves of the optimizees). Figures 5.12a, 5.12b and 5.12c visualize this information for three specific time intervals which facilitates the analysis. Starting from left to right we notice that the LSTM and M-PACT model exhibited roughly the same performance in the first 15 time steps while the M-ACT model performs inferior.

| Table 5.5: Comparison of computational effort versus accuracy for two LSTM baseline models and the best M-ACT and M-PACT model on the MLP task. In column three we state the wall clock time in minutes needed to train the model. The fourth column specifies the mean training horizon and the fifth column shows the total number of optimization steps executed during training (each epoch consisted of 16 MLPs). Column six to ten specify various step losses during the final evaluation (units are logits). The last column shows the loss value for the mean halting step determined for the adaptive meta-learners using linear interpolation. |
|---|---|---|---|---|---|---|---|---|---|
| MLP task | Training | Evaluation |
| Model | Lr | Wall clock time (min) | $E[T]$ | Optimization steps | 1 | 10 | 50 | 100 | 200 | $E[T]$ |
| LSTM (H=50) | 5e-5 | 29 | 50 | **36,000** | 2.22 | 0.87 | 0.56 | 0.56 | 0.61 | n/a |
| LSTM (H=100) | 5e-5 | 57 | 100 | 72,000 | 2.23 | 0.84 | 0.57 | 0.48 | 0.42 | n/a |
| M-ACT ($\tau = 0.003$) | 1e-4 | 54 | 56.75 | 65064 | 2.21 | 0.95 | 0.52 | 0.47 | 0.41 | 0.47 |
| M-PACT ($\nu = 0.1$) | 1e-4 | 32 | 18.52 | **37,567** | **2.2** | **0.83** | 0.52 | **0.46** | **0.4** | 0.48 |

Around step 15 the M-ACT model achieved approximately the same optimization performance as the M-PACT whereas both LSTMs got slightly inferior. Except for the LSTM($H=50$) model which started to deteriorate significantly around step 70-80, the other three optimizers slowly converged to the same performance. Note that the $y$ axis is using a log scale and that the curves suffer from a certain degree of instability that is partially caused by the size of the test set but also by the randomness of the image selection. The final (at $t=200$) mean loss value is approximately equal for the LSTM($H=100$), M-ACT and M-PACT model while this value is substantially greater for the LSTM($H=50$). Column six to ten in table 5.5 show the exact mean loss values for the four models at step 1, 10, 50, 100 and 200.

Comparing the computational effort it took to train the four different optimizers we notice that the LSTM($H=50$) and the M-PACT model utilized roughly the same wall clock time (column three table 5.5) and number of optimization steps (column five table 5.5). Nevertheless the M-PACT model achieved better optimization results especially between time steps 50 to 200. As we noted earlier also the M-ACT optimizer performed well during evaluation yet it took only slightly less computational effort to train this model compared to the LSTM($H=100$). We highlight that the M-PACT model is much more parsimonious in its computational use than the baseline model trained with a fixed horizon of 100 time steps although the performance of both models is approximately the same.

It is interesting to study the learning progress of the two adaptive meta-learners during training. Figures 5.13a and 5.13b display the mean loss values of the optimizees for the M-ACT resp. M-
**Figure 5.14:** Evaluation of M-ACT and M-PACT model on MLP task. Visualization of trajectory weights for M-ACT and M-PACT model during testing. Each column shows a different setting for the hyper-parameter $\tau$ resp. shape parameter $\nu$. Distribution of trajectory weights for the best performing M-ACT 5.14a resp. M-PACT model 5.14b.

PACT model at different time steps\(^9\) during each training epoch (y axis). The units of the x axis are time steps and the mean loss value per time step is indicated by the color (darker colors equates to lower values). The figures reveal that while the M-ACT model explored longer time sequences the mean loss values of the newly explored time steps did not increase where this was clearly the case for the M-PACT model (we refer the reader to the mean step losses between epoch 50 and 70 in fig. 5.13b).

Figure 5.15 visualizes the mean halting step (including ± one standard deviation) for both models during training. The y axis is reversed and shows the training epochs where the units of the x axis are halting steps. We first state that although not shown here, we found the same effect of the hyper ($\tau$) resp. shape parameter ($\nu$) as in the regression tasks where an increased constraint resulted in shorter time sequences explored during training. Comparing the mean halting steps during training for both models one can see that the M-PACT model resorted to significantly shorter sequences after having explored longer horizons. On the other hand we also observe that the variance of the explored sequence lengths is significantly greater for the M-PACT than for the M-ACT optimizer and this is most probably due to the stochastic threshold the former is using. The mean halting step per epoch is significantly larger for the M-ACT optimizer and we notice in figure 5.15 that the model could hardly resort to slightly shorter time sequences after having explored longer ones.

**Table 5.6:** Halting step statistics for M-ACT and M-PACT model on MLP task during final evaluation on 100 freshly sampled MLPs.

<table>
<thead>
<tr>
<th>MLP</th>
<th>Evaluation Halting step statistics</th>
<th>Max horizon during training</th>
<th>Model (hyper-parameter)</th>
<th>Range</th>
<th>Mean</th>
<th>Std</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>M-ACT ($\tau = 0.003$)</td>
<td>125</td>
<td>[91,</td>
<td>91</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>M-PACT ($\nu = 0.1$)</td>
<td>100</td>
<td>[85,</td>
<td>85.95</td>
<td>0.22</td>
</tr>
</tbody>
</table>

As mentioned in the previous section, when searching for the optimal hyper-parameter value $\tau$ (M-ACT) we found that the fixed training horizon $H$ had an important effect on the performance of the M-ACT model. Initially we set the horizon to 100 but each time the model would reach this limit it could not resort to shorter time sequences and instead always utilized the maximum number of steps for all optimizers. We want to stress that the final performance of those models was comparable with the one we present here. As soon as we increased the horizon to 125 steps the

\(^9\)Note as a reader you are looking from above, the mean step loss values are color coded.
model would not reach this limit (but exceeded the earlier horizon of 100) and did finally converge to a time sequence length below 100. Figures 5.13c and 5.13d display the histogram of halting steps for the 16 MLPs optimized in each training epoch and they augment the visualization of the halting step statistics during training (please refer to figure 5.15). The figures once more convey the effect of the deterministic (M-ACT) resp. random (M-PACT) threshold that the models were using to determine the final time step for a specific optimizee.

**Trajectory weights**

To conclude we analyze the trajectory weights that we obtained during the final evaluation of the adaptive meta-learners on the MLP task and compare them to the results we acquired in the regression tasks (fig. 5.10, 5.7). Inspecting the weights generated by the M-ACT model (fig. 5.14a) in the MLP task we observe that except for the first 20 and the last time step (91) the values are nearly constant. The shape differs significantly from the distributions we saw in the regression experiments where the mode of the distribution was located towards the end of the trajectory. Unfortunately we can only surmise about this result. Our current assumption is that we throttled the model too much by means of the hard limit on the horizon and the time-penalty value.

The shape of the trajectory weight distribution for the M-PACT model (fig. 5.14b) resembles strongly the shape we saw in the regression tasks. The model is regularized by means of the Kullback-Leibler divergence and therefore tries to adjust to the shape of the prior geometric distribution with shape parameter $\nu = 0.1$. We note that the approximated posterior distribution has a thicker tail towards later time steps than in the previous tasks and this is most certainly caused by the less aggressive shape parameter value of the prior distribution and the necessity of longer optimization trajectories to solve the demanding task. Finally table 5.6 summarizes the halting step statistics for both models during the final evaluation on 100 MLPs. The halting step variance of the M-ACT optimizer is zero and practically negligible for the M-PACT model. This could suggest that the models exhibit no input dependency but we believe that this result is caused by the lack of diversity between different instances of the MLPs.

**5.4.2 Conclusions**

We conclude this section with a brief description of the main insights that we attained while assessing the performance of the three learned optimizers on the MLP task. The last task was
clearly the most demanding of all three and the results suggest that our adaptive meta-learners performed well while needing less computational effort to be trained (RQ2.1). Once more we observe that especially the M-PACT optimizer is well suited to trade off computational effort versus accuracy. This indicates that the regularization by means of the KL-divergence helps the model to achieve good optimization results while being extremely parsimonious in its computational use. Injecting an appropriate amount of regularization during training encouraged the optimizers to explore short optimization trajectories at the beginning before exploring longer time sequences during later training stages while resorting again to slightly shorter horizons at the end of training (RQ2.3). The M-PACT optimizer achieves the best results on the MLP task especially when we take into account the parsimonious computational effort it used during training.
6 | Discussion and conclusions

In this chapter we first return to some of the research questions that we formulated in section 1.1 before we state several limitations of the pursued approaches. We conclude the chapter by giving directions for future work.

6.1 Research questions

RQ2.1: Is it possible to train the newly proposed models with less computational effort than the baseline model while achieving the same optimization performance?  

We are aware of the fact that this study is not a "finished piece of work". Nevertheless our current results indicate that a computation time adaptive meta-learner can be trained with less computational effort compared to the baseline model which uses constant trajectory weights and a fixed training horizon. Particularly the M-PACT model which uses a Bayesian approach, is extremely parsimonious during training while achieving approximately the same results as the M-ACT model.

In the less demanding regression tasks we observed a slightly inferior convergence rate of the new models in later time steps compared to the baseline model when trained for long fixed horizons. The effect was absent in the more difficult MLP task. We believe that we would have found the same effect if we had trained the baseline model with a fixed horizon of e.g. 200 time steps in the MLP task. The interesting question to pose is how the computational training effort relates to the differences in the final optimizer's performance. In other words is there a linear relationship or is it very costly to achieve the last bit of performance growth? In order to answer this question in our setting one would need to set a challenging final loss objective for the base-learners and measure the computational effort needed to train the optimizers until they achieved at least this objective. Although we did not conduct these kind of experiments in this work we can be quite sure that the relationship between computational training effort and final performance is not linear. Inspecting solely the results of the baseline model in the non-convex regression task (we refer the reader to the first three rows of table 5.4) we clearly observe that one has to roughly double the computational effort to achieve a loss reduction of 4 to 5 percent.

However we believe that our results suggest that in the more difficult task the benefit of being computation time adaptive increased. This conclusion is corroborated by the fact that the baseline model’s performance in the MLP task deteriorated significantly when the training horizon was equal to 50 time steps compared to a mean horizon of 57 resp. 19 for the M-ACT and M-PACT model (please see table 5.5).

RQ2.2: Are the models able to adapt to the varying difficulty of the objective functions within the same optimization task (also referred to as input dependency)?

We already touched upon this question when drawing conclusions based on the results we obtained from assessing the performance of our newly proposed models on the regression (section 5.3.2) and MLP task (section 5.3.4). Our results reveal that both models have acquired a certain degree of input dependency in the regression tasks because the relationship between the difficulty of an objective function and the assigned halting step exhibited some linearity. As we have previously noted, unfortunately we could not obtain similar results in the MLP task.

\(^{1}\text{For convenience we replicated the research question here.}\)
Table 6.1: Mixed MLP task: Halting step statistics for M-PACT model when trained on one- and two-layer MLPs. At test time the model was separately evaluated on 50 one-layer and two-layer MLPs using 200 optimization steps. We report the computed halting step statistics for both runs.

<table>
<thead>
<tr>
<th>Mixed MLP task</th>
<th>Evaluation Halting step statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-PACT(ν = 0.1)</td>
<td>Type of MLP Range Mean Std Median</td>
</tr>
<tr>
<td>one-layer</td>
<td>[117, 122] 119.18 1.03 119</td>
</tr>
<tr>
<td>two-layer</td>
<td>[119, 126] 122.56 1.55 123</td>
</tr>
</tbody>
</table>

We conducted a preliminary experiment in which we trained the M-PACT model on a mixture of one- and two-layer MLPs. We evaluated the model separately on 50 one-layer and two-layer MLPs and computed the halting step statistics which are stated in table 6.1. The results indicate that the model learned to distinguish between the two types of input because the statistics clearly differ. Without going into details it is interesting to observe that the results look very different to the ones we obtained when training the model only on one-layer MLPs. Hence future work should increase the diversity among the individual optimizees in the MLP task in order to investigate not only whether the models behave in an input dependent manner but also which effect increased input diversity has on the final model performance. Next to using one- and two-layer MLPs, we could also randomly choose the number of hidden units for the one-layer MLP while assuming that the difficulty of the task increases if the MLP has more hidden units.

RQ2.3: How does the injected preference for speed versus accuracy influence the performance of the models?

We conclude from the results that the introduction of a regularization term can function as a well calibrated counterforce to the general tendency of the models to explore very long time horizons during training. The time penalty and the KL-divergence term appear to be effective measures to inject our prior preferences for computational effort versus accuracy. Studying the learning behavior of both adaptive meta-learners throughout the experiments we conclude that in case the regularization term is adequately adjusted to the difficulty of the task, the adaptive meta-learners carefully explored longer optimization horizons. In most settings the models could resort to shorter horizons towards the end of training and reduced the halting step variance. We believe this is an optimal learning behavior because the models explored an adequate range of trajectory lengths and finally settled to a range that balanced computational cost and accuracy of the optimization result. However we admit that especially for the M-ACT model, it can be difficult to find the right amount of regularization.

RQ2.4: Can the trajectory weights be harnessed to determine the time step when optimization should stop for a specific objective function?

In order to answer this question we inspect the last column (headed \(E[T]\)) of table 5.2, 5.4 and 5.5 which states the mean loss values obtained at the mean halting step (only applicable for the adaptive meta-learners). This would have been on average the final performance of a base-learner trained by the adaptive optimizers. The mean loss values are clearly too high especially in the non-convex regression task. Note that we need to take the halting step variance into consideration that was induced by the randomness of the initialization procedure of the regression functions. We refer the reader to the penultimate column in table 5.3 and consider only the rows that are marked with a bold model name. The M-ACT model has clearly a significant greater mean halting step and variance then the M-PACT model which is in line with the insight that the M-PACT model can be trained with less computational effort then the M-ACT model. In the MLP task we noticed earlier that the lack of randomness in the initialization of the optimizees resulted in a lack of variance w.r.t. the halting step (please refer to table 5.6). Although taking the halting step variances into account it does not completely alleviate the fact that especially in the non-convex regression task the halting step does not coincide with the moment that the base-learners have converged.

We could influence the halting step at test time slightly by choosing a larger fixed threshold. This would probably increase the final optimization result non significantly. However we believe that based on our approach and the results it is not feasible to interpret the halting step as the...
moment when the optimization process (of the optimizee) has converged because the halting step used during training was not related to the moment of convergence. Intrinsic to both approaches, the models were penalized for not using the available probability mass in the time steps preceding the final time step \( T \) (we refer the reader to equation (3.5) resp. (3.33)). In other words the models were encouraged to increase the weights of all steps preceding the final time step \( T \). One of the underlying ideas was that this would force the model to be more aggressive, encouraging the optimizer to take larger leaps in the parameter space of the optimizee especially in the first time steps. Our results clearly reveal that this is not the case. Although these approaches resulted in shorter training horizons (aka halting steps) they do not correspond to the moment when the optimizee converged and can hence not be used during testing.

6.2 Limitations and future work

6.2.1 Truncated BPTT

Up to this point we have only briefly stated that we trained the new models without truncated BPTT. We have shown in the experiments on the regression tasks that the LSTM optimizer can be trained faster if using truncated BPTT as opposed to not using this technique. We may therefore carefully assume that our approaches would benefit from this training procedure. It will enable us to use larger "hard limits" on the maximum training horizon which is necessary in order to apply these approaches to more interesting optimization tasks that require significantly longer time sequences. As part of this work we made an attempt to implement truncated BPTT for both models. Unfortunately we observed that at training time the optimizers would quickly explore the maximum available horizon (i.e. hard limit \( H \)) and progressively assign this maximum horizon to all optimizers. The regularization term increased dramatically and the models were unable to lower this part of the loss objective which basically indicated that learning had stopped. In addition the models seem to be agnostic to the amount of regularization they were trained with. It is possible that we implemented the procedure inappropriately. On the other hand it could be the case that the method impairs the regularization of the models because in both cases the last time step probability plays an important role in the regularization term (refer to eq. (3.5) and (3.33)) but has no influence during intermediate BPTT runs. Frankly we did not investigate the problem in all depth and we concede that it is an essential improvement to enable truncated BPTT for both models.

6.2.2 M-PACT approach

In this section we would like to highlight some additional thoughts and critical remarks w.r.t. the approach that we pursued in order to develop the M-PACT model.

We start by pointing out that instead of defining a Bayesian probabilistic model with the latent variables \( T \) and \( z \) one could argue that a model with solely a discrete, one dimensional latent variable \( t \) denoting the number of optimization steps, would have been sufficient. We evaluated such a model during this study and briefly describe the problems we encountered here. This will provide the reader with additional background on why we have chosen the approach presented earlier.

Using a Bayesian approach we derived an evidence lower bound for the marginal likelihood \( p_\theta(\mathbf{x}^{(n)}) \) and approximated the true posterior distribution \( p_\phi(t|\mathbf{x}) \) which we assumed belongs to the family of geometric distributions, by introducing \( q_\phi(t|\mathbf{x}^{(n)}) \):

\[
\log p_\phi(\mathbf{x}^{(n)}) \geq \mathcal{L}(q, \phi) = \mathbb{E}_{q_\phi(t|\mathbf{x}^{(n)})} \left[ \log p_\phi(\mathbf{x}^{(n)}, t) - \log q_\phi(t|\mathbf{x}^{(n)}) \right] \tag{6.1}
\]

\[
= \mathbb{E}_{q_\phi(t|\mathbf{x}^{(n)})} \left[ \log p_\phi(\mathbf{x}^{(n)}|t) \right] - D_{KL}(q_\phi(t|\mathbf{x}^{(n)}) || p(t)) \tag{6.2}
\]

\[
= \sum_{t=1}^{\infty} q_\phi(t|\mathbf{x}^{(n)}) \log p_\phi(\mathbf{x}^{(n)}|t) - D_{KL}(q_\phi(t|\mathbf{x}^{(n)}) || p(t)). \tag{6.3}
\]

Due to the infinite sum the expectation taken w.r.t. \( t \) is intractable and one needs to resort to Monte Carlo expectations to approximate the ELBO. In order to sample \( t \) we used the same
sampling function that we stated in equation (3.34). We observed that such a model did not explore time sequences longer than four to five steps. Furthermore the model was agnostic to the value of the shape parameter of the geometric prior. Based on our analysis we concluded that the model could not learn how to use the generated step weights in order to influence the sequence length. It seemed that the learning signal produced by the samples was too weak and noisy to have an effect on the optimizer’s parameters. As soon as we computed a loss that was based on the complete optimization trajectory (i.e. the entire weighted sum of losses) the model would start to explore longer time sequences.

This insight encouraged us to introduce a second latent variable $z$, as we have described in detail in section 3.3.1, which enabled us to take a double expectation when deriving an estimator for the marginal likelihood. For convenience we repeat equation (3.37) here

$$\log p_\phi(x^{(n)}) \geq \mathbb{E}_{q_\phi(T|x^{(n)})} \left[ \mathbb{E}_{q_\phi(z|x^{(n)},T)} \left[ \log p_\phi(x^{(n)}, z|T) - \log q_\phi(z|x^{(n)}, T) \right] \right]. \quad (6.4)$$

The inner expectation w.r.t. $z$ permits us to compute the expectation over the entire optimization trajectory which increased the precision of the approximated KL-divergence. Furthermore this solution empowers the model to learn how to use the time step weights in order to influence the trajectory length. The outer expectation w.r.t. $T$ allows the model to explore different sequence lengths.

6.2.3 Computational effort

In this study we have defined computational effort in terms of the total number of optimization steps an optimizer performed during training (please refer to section 3.5 and equation (3.45)). We admit that this term does not automatically equate to computational speed. Although we have stated the wall clock time for training the models in the MLP task, we are aware of the fact that our current implementation of the adaptive meta-learners exhibit some significant computational overhead. Inherent to the method it is necessary to filter the minibatches at each time step w.r.t. the optimizees for which the optimization procedure must stop. In this work we did not optimize our implementation w.r.t. this point but we are confident that the overhead can be significantly reduced. This is a necessary enhancement although we stress that there will be always some additional computational costs in comparison to the baseline model. Furthermore the current results indicate that the extra computational toll starts to pay off in more demanding optimization tasks.

6.2.4 Miscellaneous

Throughout this work we highlighted several times that an essential assumption underlying our approaches states that the contributions of the optimization steps to the final result should be weighted dissimilar. Although we have shown that the pursued approaches have a positive effect on the computational training effort, it is questionable whether the term contribution is the correct interpretation for these weights. In other words, we think that the current loss objectives do not explicitly encourage the models to weight a time step according to its achieved loss reduction which we assume could quantify the contribution of a step to the final optimization result. Therefore future work could address this by defining a different loss objective that more explicitly expresses the learning progress achieved in a time step.

A related idea to improve the training procedure is inspired by the recent work of Alex Graves on automated curriculum learning [9]. One could consider the different sequence lengths the optimizer has to master as separate tasks that form an ensemble and hence corresponds to a curriculum. The difficulty of a task could equate to the sequence length. The idea is that an optimizer is learning a policy over tasks that maximizes overall progress. Graves defines different learning progress signals (e.g. loss-driven and complexity-driven) and we think it could be beneficial to replace the raw optimizee loss in our proposed loss objectives with a loss-driven learning progress signal. Again, using such a signal would fit with the idea that the step weights quantify the contribution of an optimization step to the overall result. The association between an optimization task and
curriculum learning fits our observation that the learned optimizers start with very short time sequences before they gradually explore longer time sequences.

Another, final idea on how one could enhance the training procedure is by using a separate set of optimizees that do not participate in the backpropagation procedure. The underlying idea is that such a model can more freely explore the error surface of the optimizees without being immediately punished for this behavior.
Bibliography


A Partial views on computational graph M-ACT model

Figure 1: Partial view 1 on the computational graph of the M-ACT model. Highlighting the information flow between optimizee and optimizer. At each time step $t$ the gradients w.r.t. the optimizee’s parameters are ingested by the LSTM module which produces a step vector $\Delta g_t$ and a halting activation $h_t$ (by means of the sigmoid function). The step vector is added to the current parameter values of the optimizee. Note that the halting activations are used (a) to compute a weighted sum of losses and (b) to determine the final optimization step $T$. There is a little subtlety not shown in the graph. The halting activation generated in the last time step is only used to determine the last time step $T$ (based on a fixed threshold value). Instead of $h_T$ the rest probability $(1 - \sum_{t=1}^{T-1} h_t)$ is utilized in the weighted sum of the loss objective.
Figure 2: Partial view 2 on the computational graph of the M-ACT model. Visualizing the composition of the loss objective of the optimizer. In each time step $t$ (except for the last step $T$) the objective value of the optimizee $f(\theta_t)$ is multiplied by the halting activation $h_t$ generated in the same time step. This results in a weighted sum of the optimizee’s time step losses. In the last time step $T$ the objective value $f(\theta_T)$ is multiplied by the rest probability $(1 - \sum_{t=1}^{T-1} h_t)$. The time penalty of the model is equal to the scaled sum of the number of time steps $T$ and the rest probability. $\tau$ is a hyper-parameter of the model.
B Additional figures linear regression experiments

Figure 3: Effect of truncated BPTT on final performance of the LSTM optimizer: Each row consists of two figures that visualize the complete time interval split in two intervals. Both rows compare the LSTM optimizer performance on four truncated BPTT settings. Shown are mean step losses. **Row 1**: convex linear regression task with fixed training horizon \( H = 50 \) and \( L \in \{5, 10, 20, 50\} \). The optimization trajectory of 100 time steps is split into two intervals (1) \([0, 5]\) left side and (2) \([5, 100]\) on the right side. **Row 2**: non-convex linear regression task with fixed training horizon \( H = 50 \) and \( L \in \{10, 20, 30, 50\} \). The optimization trajectory of 200 time steps is split into two intervals (1) \([0, 100]\) left side and (2) \([100, 200]\) on the right side. Although the different sub-sequence lengths used during truncated BPTT have an effect on the learning speed, the method does not influence the final performance of the LSTM optimizer.

C Gradient preprocessing

Please note that this appendix is taken from the original paper of [1] because we followed the advice to preprocess the gradients of the optimizer when optimizing simple multilayer perceptrons.

A potential challenge in training optimizers is that different input coordinates (i.e. the gradients w.r.t. the different optimizee parameters) can exhibit varying magnitudes. This is indeed the case e.g. when the optimizee is a neural network and different parameters correspond to weights in distinct layers. This can make training an optimizer difficult, because neural networks naturally disregard small variations in input signals and concentrate on bigger input values.

To this aim we propose to preprocess the optimizer’s inputs. One solution would be to give the optimizer \((\log(|\nabla|), \text{sgn}(\nabla))\) as an input, where \(\nabla\) is the gradient in the current time step. This has a problem that \(\log(|\nabla|)\) diverges for \(\nabla \to 0\). Therefore, we use the following preprocessing formula
\[ \nabla = \begin{cases} \left( \frac{\log(|\nabla|)}{p}, \text{sgn}(\nabla) \right) & \text{if } |\nabla| \geq e^{-p} \\ (-1, e^{p} \nabla) & \end{cases} \]

where \( p > 0 \) is a parameter controlling how small gradients are disregarded (we used \( p = 10 \) in all our experiments).

### D Geometric priors

![Truncated geometric prior](image)

(a) \( \nu = 0.05 \) and \( T = 50 \)  
(b) \( \nu = 0.1 \) and \( T = 50 \)  
(c) \( \nu = 0.3 \) and \( T = 50 \)

**Figure 4:** Examples of truncated geometric prior distributions with three different shape parameter values.

### E Marginal posterior \( q(z|x^{(n)}) \)

The individual marginal step probabilities \( q(z_t|x^{(n)}) \) can be calculated as follows

\[
q(z_t|x^{(n)}) = \sum_{T=t}^{\infty} q(T|x^{(n)}) \ q(z_t|x^{(n)}, T) 
\]

\[
= q(T = t|x^{(n)}) \ \delta_{q_{t-1}} + \sum_{T>t} q(T|x^{(n)}) \ q(z_t|x^{(n)}, T). 
\]

Note that we decomposed the infinite sum into a part where \( T = t \) and a second part where \( T > t \). Also note that the initial sum starts at \( T = t \) instead of \( T = 1 \) because \( q(z_t|x^{(n)}, T) = 0 \) for all \( T < t \).

We can then use (3.30) in order to replace \( q(z_t|x^{(n)}, T) \) with \( \prod_{i=1}^{t-1}(1 - \rho_i)\rho_t \) because \( t < T \). This results in:

\[
q(z_t|x^{(n)}) = q(T = t|x^{(n)}) \ \delta_{q_{t-1}} + \sum_{T>t} q(T|x^{(n)}) \ \prod_{i=1}^{t-1}(1 - \rho_i)\rho_t 
\]

\[
= q(T = t|x^{(n)}) \ \delta_{q_{t-1}} + \prod_{i=1}^{t-1}(1 - \rho_i)\rho_t \ \sum_{T>t} q(T|x^{(n)}) 
\]

\[
= q(T = t|x^{(n)}) \ \delta_{q_{t-1}} + \prod_{i=1}^{t-1}(1 - \rho_i)\rho_t \left( 1 - \sum_{T=1}^{t} q(T|x^{(n)}) \right) 
\]

\[
= q(T = t|x^{(n)}) \ \delta_{q_{t-1}} + q(z_t|x^{(n)}, T > t) \left( 1 - \sum_{T=1}^{t} q(T|x^{(n)}) \right) 
\]

where we have used the facts that all \( q(z_t|x^{(n)}, T > t) \) probabilities are independent of \( T \) and \( \sum_{T>t} q(T|x^{(n)}) = 1 - \sum_{T=1}^{t} q(T|x^{(n)}) \). \( \delta_{q_{t-1}} \) denotes the rest probability or remaining stick length.
after we broke the stick in the previous time step. Another subtlety relates to the fact that all probabilities in the set \( \{ q(z_t|\mathbf{x}^{(n)}, T) \}_{T>t} \) have the same value (due to the stick-breaking process).

Convex regression

Figure 5: Convex regression: Approximated marginal posterior distribution \( q(z|\mathbf{x}^{(n)}) \) for three different shape parameter values (M-PACT).

Non-convex regression

Figure 6: Non-convex regression: Approximated marginal posterior distribution \( q(z|\mathbf{x}^{(n)}) \) for three different shape parameter values (M-PACT).
F  Additional figures w.r.t. trajectory weights

Convex regression

**M-ACT**: Examples of trajectory weights $p(t)$ for different horizon lengths $T \in \{20, 23, 26, 30\}$

**M-PACT**: Examples of trajectory weights $q(z|x, T)$ for different horizon lengths $T \in \{7, 8, 9\}$

![Convex regression examples](image)

**Figure 7**: Convex regression: Distribution of trajectory weights at test time for different horizon lengths. Figure 7a shows four histograms for M-ACT($\tau=0.0003$) model and figure 7b visualizes three conditional distributions for the M-PACT($\nu=0.7$) model. Orange histogram shows the corresponding truncated prior geometric distribution.
Non-convex regression

**M-ACT:** Examples of trajectory weights $p(t)$ for different horizon lengths $T \in \{20, 30, 50, 70\}$

![Graphs of M-ACT](image)

(a) M-ACT($\tau=0.0018$)

**M-PACT:** Examples of trajectory weights $q(z|x,T)$ for different horizon lengths $T \in \{10, 12, 14, 16\}$

![Graphs of M-PACT](image)

(b) M-PACT($\nu=0.5$)

**Figure 8:** Non-convex regression: Distribution of trajectory weights at test time for different horizon lengths. Figure 8a shows four histograms for M-ACT($\tau=0.0018$) model and figure 8b visualizes four conditional distributions for the M-PACT($\nu=0.5$) model. Orange histograms show the corresponding truncated prior geometric distributions.