Universiteit Leiden

Master Computer Science

The Training of Neural Networks that can Train Neural Networks

Name:	Thijs Simons
Student ID:	s1830120
Date:	28-03-2023
Specialisation:	Artificial Intelligence
1st supervisor:	Mike Huisman
2nd supervisor:	Jan N. van Rijn
3rd supervisor:	Thomas Moerland

Master's Thesis in Computer Science

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

Abstract

Conventional neural network optimization relies heavily on handcrafted learning algorithms, mainly Stochastic Gradient Descent (SGD) or its variants. This study explores the possibility of harnessing machine learning to enhance or uncover novel neural network learning algorithms. We explore an existing method, the Message Passing Learning Protocol (MPLP), a technique designed to learn a neural network optimizer without explicitly providing gradients. An often-encountered problem of methods like MPLP is meta-training difficulties like early stagnation and long training times. It is often uncertain which hyperparameters lead to successful results.

To overcome these limitations, we will discuss existing strategic design choices and techniques that can facilitate the meta-training of MPLP-like methodologies. Especially, we draw attention to a regularization method that can make the meta-training of our configuration much more predictable. Finally, we reverse engineer a learned MPLP and show that it has learned an SGD-like learning algorithm.

Contents

1 Introduction

Recent years have witnessed significant advancements in deep learning techniques, particularly in the fields of computer vision and natural language processing [Goodfellow et al., 2016; LeCun et al., 2015; He et al., 2016; Vaswani et al., 2017]. Part of these advancements was thanks to the development of optimization algorithms. These optimization algorithms are designed by humans and can still be suboptimal. To improve optimization algorithms even further, meta-learning techniques have been proposed to automatically learn optimizers that could outperform existing ones.

There are different architectures for learning optimizers, ranging from lightweight techniques to more advanced, end-to-end approaches. Lightweight learned optimizers aim to enhance existing optimization techniques without introducing much overhead, but they may lack expressiveness. These types of learned optimizers are often based on SGD-like optimizers and often get explicit gradient information [Andrychowicz et al., 2016; Li and Malik, 2017a; Harrison et al., 2022; Metz et al., 2022]. Biasing the learned optimizer's architecture towards existing optimizers makes it less likely to discover novel optimization methods. Therefore, to discover potentially new optimization techniques, the learned optimizer should be less biased towards existing algorithms. By making the optimizer more expressive additional computational cost is introduced and therefore, these optimizers are still only mainly interesting from a discovery and scientific perspective.

For the existing more expressive learned optimizers that have been developed the training process is often underreported on [Sandler et al., 2021; Kirsch and Schmidhuber, 2021]. Problems during the training of the optimizers are mentioned but exact data, e.g. outer-learning curves, is not reported. It is unknown under what conditions some methods are able/not able to be trained. This makes the development of learned optimizers particularly painful.
The term *outer* here refers to the training of the optimizer itself. The term *inner* refers to the learned optimizer being applied to a neural network.

The research on outer-training dynamics that has currently been conducted focuses on, the more practical, lightweight SGD-like learned optimizers [Metz et al., 2019, 2022; Andrychowicz et al., 2016]. Some of this knowledge can be transferred to more expressive optimizers but not all. E.g., the training of more expressive optimizers is more likely to get stuck at the beginning of training [Randazzo et al., 2020; Kirsch and Schmidhuber, 2021].

In this thesis, we investigate the Message Passing Learning Protocol (MPLP) framework [Randazzo et al., 2020], a learning-to-optimize method that occupies a unique position between lightweight and end-to-end approaches. MPLP serves as an optimizer for the weights of a neural network while still having the flexibility to learn different optimization methods, such as those distinct from SGD. Unlike approaches with direct access to gradients, the MPLP relies on message passing through the neural network being optimized to determine how to update its weights. We will provide a detailed explanation of the framework's main

idea and its underlying mechanisms. While the MPLP-like optimizer has been reported as difficult to meta-train [Randazzo et al., 2020], the precise nature remains unclear. Our goal is to better understand the MPLP framework and its meta-training dynamics. We hope that the found knowledge will allow us to better understand the training dynamics of expressive learned optimizers in general.

We pose the following research questions:

Research Questions:

- How does the MPLP framework relate to SGD?
- What are the meta-training dynamics of MPLPs, and what techniques can we use to improve them?

Contributions:

- We show that the MPLP framework is expressive enough to mimic SGD.
- We train an MPLP and show that it has learned to do SGD.
- We give examples of what the outer-training curve of an MPLP can look like.
- We investigate various techniques to improve the training, including learning a learning rate, increasing message size, and exploring normalization methods.
- We show that in certain settings the MPLP can get stuck at the beginning of training.
- We show how a modification of batch entropy regularization can be used as a technique to help prevent the MPLP from getting stuck at the beginning of training.

Through this investigation, we hope to deepen our understanding of the MPLP framework, its training dynamics, and potential methods to improve its training. By doing so, we aim to contribute to the broader field of meta-learning.

2 Background

The background section of this thesis provides the necessary foundation for understanding the problem setting and the methods used in learning to optimize. We will discuss the key concepts and techniques relevant to our investigation of the Message Passing Learning Protocol (MPLP) framework.

Inner and outer optimization: In learning to optimize and meta-learning the naming can be confusing because there are multiple optimization processes and it can be confusing to which one is being referred to. Therefore, we use the following, much used, naming convention of the inner and outer optimization process. The outer-loop is referred to as the meta-learning process, in our case learning an optimizer. Outer and meta are used interchangeably.
The inner-loop is referred to the learning process of the optimizee, in our case a neural network that gets optimized for a certain task. We will stick to this formulation throughout this thesis.

Problem setting of learning to optimize: In the field of learning to optimize, the main objective is often to achieve the lowest possible test loss using a given training dataset. This requires discovering an optimization strategy $\mathcal{A}$ that can effectively generalize to new, unseen data and quickly adapt to novel tasks. To put it formally, we are given a training dataset $\mathcal{D}_{\text{train}} = (\mathbf{x}_i^{train}, \mathbf{y}_i^{train})$ and a test dataset $\mathcal{D}_{\text{test}} = (\mathbf{x}_j^{test}, \mathbf{y}_j^{test})$, which are both unseen to the optimization algorithm $\mathcal{A}$. Here, $\mathbf{x}_i^{train}$ and $\mathbf{x}_j^{test}$ denote vectors of the input features, while $\mathbf{y}_i^{train}$ and $\mathbf{y}_j^{test}$ represent vectors of the corresponding target values in the train and test datasets, respectively. Our goal is to find an optimization algorithm $\mathcal{A}$ that minimizes the loss $L(\mathcal{A}(\mathcal{D}_{\text{train}}), \mathcal{D}_{\text{test}})$ when assessed on the test dataset $\mathcal{D}_{\text{test}}$.

The optimization strategy $\mathcal{A}$ can be designed to perform well on specific tasks. For example, suppose we want to develop an optimization algorithm that excels in image classification tasks. In this case, we can train $\mathcal{A}$ on a variety of image classification datasets, such as MNIST or ImageNet [Lecun et al., 1998; Deng et al., 2009], to ensure that it learns effective optimization strategies for this type of task. Once the algorithm has been trained on these datasets, it might be capable of generalizing to new, unseen image classification tasks, thereby minimizing the loss when applied to their respective test datasets. By focusing on a specific task domain during meta-training, the learned optimization strategy $\mathcal{A}$ is more likely to exhibit strong performance when encountering similar tasks in the future.

Definition of stochastic gradient descent (SGD): SGD is a widely used *already existing* optimization algorithm that serves as a baseline for many learning tasks. The update rule for SGD is given by:

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial L(\theta_t; \mathbf{x}_t, \mathbf{y}_t)}{\partial \theta}, \tag{1}$$

where θ_t denotes the inner-parameters at iteration t, α is the learning rate, and $L(\theta_t; \mathbf{x}_t, \mathbf{y}_t)$ represents the loss function L parameterised by θ_t for the input-output pair $(\mathbf{x}_t, \mathbf{y}_t)$. We will

use the concept of SGD to provide a foundation for understanding how learning to optimize could be achieved.

Parameterization of the update function: One approach to enhancing the performance of optimization algorithms like SGD, is to parameterize the update function of SGD [Andrychowicz et al., 2016]. By introducing flexibility and adaptability in the optimization process, the model can learn better optimization strategies, potentially improving the performance of the underlying learning task. Let the update function be denoted by f. Then, the parameterized update rule can be expressed as:

$$\theta_{t+1} = \theta_t - f(\phi; x_t, y_t, \theta_t) \tag{2}$$

Here, ϕ represents the meta-parameters we want to learn. The main idea is that ϕ determines how the weights θ of the model should be updated. The next step is to find the meta-parameters ϕ such that the update function performs well. To do this we first specify the objective that should be minimized such that the performance is high, we call this the outer-loss.

Definition of the outer-loss: The outer-loss is a crucial concept in meta-optimization. It is defined as the weighted sum of inner-losses experienced while optimizing, and serves as an objective function for the meta-optimization process. This enables us to evaluate the performance of the learned optimizer in terms of its ability to minimize the loss on tasks. Mathematically, the outer-loss $L_{\text{outer}}(\phi)$ can be defined as:

$$L_{outer}(\phi) = \sum_{t=1}^{K} w_t \cdot L(\theta_t; \mathbf{x}_t, \mathbf{y}_t) \tag{3}$$

Here, K represents the number of update steps that are performed before calculating the outer-loss. w_t is a weighting factor for the different inner-losses, allowing for a trade-off between optimization speed and the final loss. In this thesis, we use $w_t = 1$ for all t, which is equivalent to minimizing the area under the inner-loss curve [Li and Malik, 2017b]. $L(\theta_t; \mathbf{x}_t, \mathbf{y}_t)$ is the inner-loss after t update steps.

ϕ are the meta-parameters that are used to update the weights in each update step. ϕ is only represented on the left side of the equation because the outer-loss here is shown without the explicit updates of the inner-parameters θ_t. In our setup ϕ determines how the inner-parameters θ_t are updated between loss calculations (See e.g. Eq. 2). This means that each loss has a dependence on the losses that were calculated before itself.

Note, how the meta-parameters ϕ affect the outer-loss in a recurrent way similar to how the parameters of RNNs do this [Vicol et al., 2021]. From the perspective of RNNs, Eq. 2 calculates the next state using the recurring parameters ϕ and the previous state θ_t. Figure 1 illustrates this well. The parameter α from the illustration is the recurring meta-parameter we are interested in optimizing. This recurrence can make the outer-loss highly sensitive to changes in the outer-parameters as explained by Metz et al. [2019].

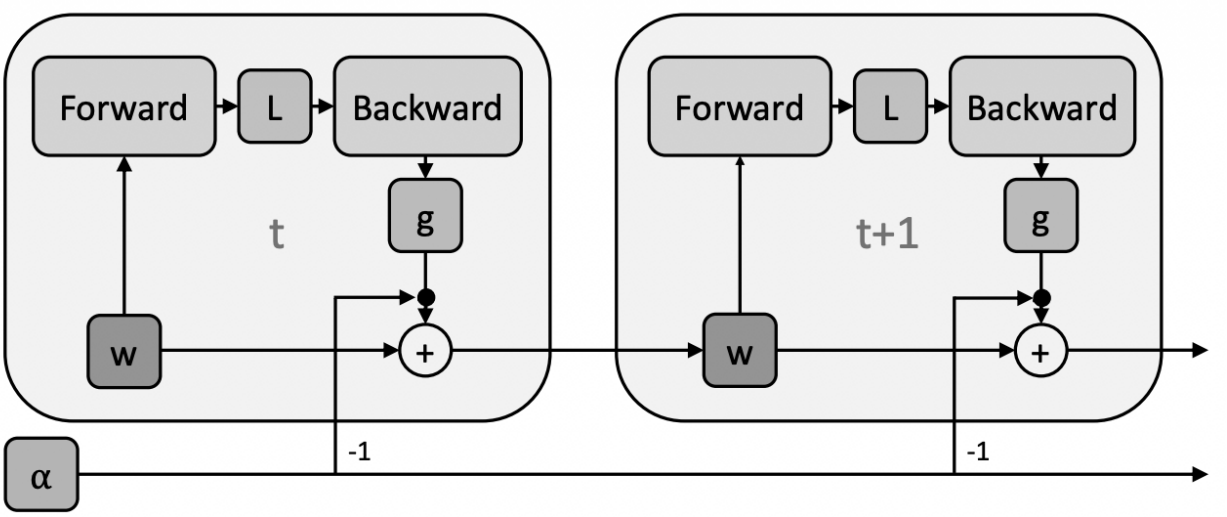


Figure 1: High level illustration of a computational graph of the optimization process of a neural network like model using gradient descent. α stands for the learning rate. Incorporating the forward, backward and update step in a single computational graph allows us to analytically calculate the gradients of the outer-loss with respect to α. Subsequently, allowing us to do gradient descent on α. Similar computational graphs are constructed when calculating the outer-loss. After each forward pass an inner-loss is calculated, this inner-loss is used in the calculation of the outer-loss (See Eq. 3). Source of illustration: [Wu et al., 2018]

Meta-parameter optimization: Our goal is to find the meta-parameters ϕ such that the outer-loss is minimized, effectively learning an optimizer. The most used method to optimize the parameters of an optimizer is that of stochastic meta descent [Schraudolph, 1999]. SMD is very similar to regular gradient descent. Given a parameterized loss function we want to minimize, we can calculate the gradient of the loss with respect to the parameters. We can then use this gradient as a descent direction to update the parameters. The only difference between SMD and regular gradient descent is that the gradient is calculated using the outer-loss instead of the inner-loss. In practice this means that the loss landscapes can look different and will give different training dynamics. E.g. the recursive occurrence of the outer-parameter ϕ can lead to different training dynamics than regular, non-meta, learning settings [Vicol et al., 2021].

There are multiple methods to calculate the gradients of the outer-loss with respect to the meta-parameters. Some of these are backpropagation through time (BPTT), evolutionary strategies (ES) and real-time recurrent learning (RTRL) [Mozer, 1995; Metz et al., 2019; Williams and Zipser, 1989]. After the gradients have been calculated, they can be used as our descent directions in the outer-parameter space. The advantage of using gradient descent compared to other optimization methods is that it can be used to optimize models with many parameters efficiently.

Meta-gradient calculation methods: There are several methods to compute the gradients required for meta-optimization, including:

- *(Truncated) Backpropagation through time ((T)BPTT):* A method for computing gradients in recurrent neural networks, which can also be applied to other sequence-based models. BPTT involves unfolding the computation graph over time and applying standard backpropagation to compute the gradients [Mozer, 1995].
- *Evolution strategies (ES):* A population-based optimization technique inspired by natural evolution, where gradients are approximated using a set of random perturbations applied to the meta-parameters. ES is particularly useful when the optimization landscape is non-differentiable or when gradient information is unavailable [Metz et al., 2019]. An advantage of ES is that it is better suited for longer truncations because we do not need to keep the computational graph on which the gradients are calculated in memory. The downside is that the variance gets too large when estimating the gradients of too many parameters. The variance can be reduced by increasing the number of samples at the cost of computation [Vicol et al., 2021].
- *Real-time recurrent learning (RTRL):* An online learning algorithm for recurrent neural networks that computes gradients incrementally as new input-output pairs are encountered. RTRL maintains a running estimate of the gradients, allowing for online parameter updates [Williams and Zipser, 1989]. The downside of this method is the large amount of computation and memory required. For most settings RTRL is infeasible.

Each of these methods has its advantages and disadvantages. In this thesis, we use TBPTT because for our setting it has the lowest computational costs. The main disadvantage of TBPTT is that truncation bias is introduced when the computational graph is not fully unrolled.

Everything can be parameterized: Using the concept of an outer-loss, it allows us to learn what would normally be considered hyperparameters. We will give a few examples of how we can parameterize and learn different parts of the optimization process. The most common thing to parameterize is the update function of an SGD-like optimizer [Metz et al., 2019; Li and Malik, 2017b; Andrychowicz et al., 2016; Vicol et al., 2021; Harrison et al., 2022]. The forward pass of the neural network can be parameterized. This is similar to doing architecture search [Liu et al., 2019; Sandler et al., 2021]. Very common optimizer hyperparameters like learning rate and weight decay can be parameterized [Maclaurin et al., 2015; Chandra et al., 2022] using this method. It is possible to learn the initialization parameters of the optimizee such that they generalize on a category of tasks [Finn et al., 2017]. Finally, it is also possible to parameterize the full backward pass, including gradient calculations, of the optimization algorithm. This is what we will focus on in this work.

Using the notion of an outer-loss to optimize single hyperparameters like the learning rate would be computationally expensive because more full optimizations runs have to be done

in comparison to black box methods like random search. It becomes more computationally efficient when large amounts of hyperparameters have to be optimized and if the hyperparameters can be transferred to new tasks. For example, it might be computationally expensive to learn an optimizer, but if the optimizer can be used for all problems without having to tune any of its hyperparameters it can save computation [Metz et al., 2022].

3 Related work

In the field of learning to learn there exists a spectrum of methods. At the beginning of the spectrum there are methods that are heavily based on existing learning algorithms. Often partially parameterized SGD-like algorithms that are applied to neural networks. At the other end of the spectrum, we have learning algorithms that are fully end-to-end. Which means that the method is fully black-box. The input is $\{(x_1, y_1), ..., (x_t, y_t), (x_{t+1}, 0)\}$ and the output is a prediction of y_{t+1}. The black box both represents the optimizee and the optimizer but they are intertwined within the box.
We have separated this section into methods that are based on update functions of SGD-like algorithms and into methods that are *not* based on update function of SGD-like algorithms. The latter includes end-to-end based learning methods but also neural network optimization based learning methods. E.g., [Sandler et al., 2021; Randazzo et al., 2020] are based on optimizing a neural network but are too distinct from SGD-like optimization methods to include them in that category.

3.1 Update function based

The first update functions based on SGD-like algorithms were learned by Bengio et al. [1995], they learned Hebbian-like update functions but also provided the gradients as input features in some of their experiments. Hebbian-like learning methods are biologically inspired algorithms that adjust neural network connections based on correlated activity, following the principle "neurons that fire together, wire together" [Brown, 2020].
Learning update functions was later popularized by the work of [Andrychowicz et al., 2016], in this work, at each weight of a neural network a RNN is used to calculate how the weight should be updated. The input to the RNN is the gradient and the previous state of the RNN. The inspiration to use an RNN to update the weights comes from optimizers that keep track of running statistics, like momentum and second-order momentum in Adam [Kingma and Ba, 2015]. This work also popularized backpropagating through the optimization process using TBPTT.

Many architectural changes have been made to the work of [Andrychowicz et al., 2016], e.g., adding more features as input to the optimizer [Lv et al., 2017], swapping the RNN for an MLP [Metz et al., 2019, 2020a], creating a hierarchical optimizer [Wichrowska et al., 2017; Metz et al., 2020b] etc. In addition to architectural changes there have also been changes in the training procedures. To increase generalization a larger set of training tasks

was used [Metz et al., 2022; Wichrowska et al., 2017]. Optimization methods like evolution strategies are used to increase training stability and reduce truncation biases [Metz et al., 2019; Vicol et al., 2021]. Arguably, the current best-performing learned optimizer is that of [Metz et al., 2022] which uses a combination of the developed techniques to train an optimizer that generalizes well and is competitive with non-learned optimizers.

3.2 Not update function based

To the best of our knowledge, the first work on learning an update rule not based on an SGD-like algorithm was by [Schmidhuber, 1987]. In this work, genetic programming is used to learn programs that can learn. The connection is made to neural networks but they were not used in the work. The first work that we could find that was more related to neural networks is that of [Bengio et al., 1991] and [Bengio et al., 1995]. The method tries to find new learning rules inspired by nature e.g., similar to Hebbian-like update rules. In this work mostly local information is used to calculate the weight updates and thus, the method is not very expressive. [Runarsson and Jonsson, 2000] learns to optimize a single hidden layer neural network using an evolutionary search algorithm and successfully learns to do backpropagation on a limited set of problems.

One of the first, more modern, is by Hochreiter et al. [2001]. The paper uses an end-to-end approach to learning to learn. It does this by using an RNN that does both the modeling and the optimization. At each timestep the RNN is given the input x_t and the ground truth of the previous timestep y_{t-1}. If at each timestep the RNN is given the ground truth of the current timestep y_t it would be able to cheat by outputting the ground truth y_t at each timestep. To prevent the RNN from cheating at each time step the input x_t and the *previous* ground truth y_{t-1} are passed to the RNN.
Intuitively, you can look at the RNN state that is being passed to the next timestep as the parameters of the optimizee and the size of the state is often in the order of hundreds. The parameters of the RNN could then be viewed as the optimization algorithm. A problem with systems like these is that the "optimizer" would have many more parameters than the "optimizee". Generally, it is preferred for the optimizer to have less parameters than the optimizee [Kirsch and Schmidhuber, 2021].

[Kirsch and Schmidhuber, 2021] recognizes the issue of the optimizer having many more parameters than the optimizee and tries to solve this by changing the architecture thoroughly. Instead of using a single RNN, multiple RNNs are placed in a network like structure that share the same parameters. By using multiple RNNs there are also multiple states at any moment in time. Thus, increasing the number of parameters the "optimizee" can have. The method is being optimized using Evolution Strategies (ES) because ES would be more stable than analytical methods. This work parameterizes both the forward and the backwards pass. A drawback of the method is that for a single forward pass all RNNs have to run for each layer in the network to mimic the sequential layer execution. This is computationally expensive. The authors of the paper show that the method can mimic the behavior of SGD. And

suggest that it can in an unrestricted setting also learn qualitatively different optimization algorithms.

[Sandler et al., 2021] uses a method that is a modification of how neural networks are generally constructed and trained. I.e. they do not use RNNs architectures to update the state but instead use the SGD update rule to update the state from Eq. 2. In this method the forward pass and the backward pass are dissected in smaller parts. Then these smaller parts are parameterized such that novel learning methods can be learned. They show how their method can learn gradient descent but interestingly they show evidence of how their method differs from gradient descent. In the work they also noted that normalization plays a crucial role in getting the meta-learning stable.

The Message Passing Learning Protocol [Randazzo et al., 2020] is similar to that of [Sandler et al., 2021] in the sense that it more closely resembles the SGD algorithm. MPLP does not modify the forward pass. Only the backward pass is modified. By looking at a neural network as a computational graph, we can place message passing neural networks that pass messages from output to input. These messages can then be used to update the parameters of the network. The MPLP method was only used on few-shot learning. The MPLP is at the core of our work and the details of the MPLP are described further in 4.1.

Finally, more recent work, has shown that, under the right circumstances, transformers can learn gradient descent [von Oswald et al., 2022; Akyürek et al., 2023]. Given the expressivity and the better-conditioned loss landscapes, transformers are a promising class of architectures to do meta-learning with [Kirsch et al., 2022]. The forward and backward passes are done entirely end-to-end. The disadvantage of this is that you are limited to the amount of data and the amount of "update steps" you can do. This would make using transformers as learning algorithms for very large datasets currently infeasible. On tabular data transformers have shown to be competitive with other machine learning methods like random forests [Hollmann et al., 2022].

4 Method

4.1 Message passing framework

As mentioned before, the learning to optimize framework we use is based on the MPLP framework [Randazzo et al., 2020]. In this section, we will explain the framework. In case we have deviated from the original framework we will mention this and explain why.

In summary, the MPLP framework is a regular MLP looked at from the perspective of a computational graph (See Fig. 2 for an example of such a computational graph). During the backward pass, messages are generated and passed through this graph to calculate how the parameters of the MLP should be updated.

We explain the MPLP framework in two parts, the forward and the backward pass. In the

forward pass, we will explain how the computational graph is constructed. In the backward pass, we will explain how the messages are passed and how the weights are updated.

Currently, only MLPs are supported but it could be extended to other architectures as well in the future. We have implemented the framework such that any arbitrary MLP can be constructed. The source code can be found here: `https://github.com/SimonsThijs/MPLP`.

Forward pass

As previously mentioned, the forward pass can be any arbitrary MLP. In this section we explain how we can represent the MLP as a computational graph such that messages can flow through this graph.
In Table 1 we define the 3 types of nodes that are used in the computational graph of an MLP. The nodes are defined as follows: the linear node is a node that performs a linear transformation on the input by multiplying it with a weight in the MLP. The linear node is statefull because it keeps track of a weight. The activation node is a node that performs a nonlinear transformation on the sum of the incoming values. The loss node is a node that computes the loss between the input and the ground truth. The loss nodes are the output of the computational graph. Note that we can have multiple loss nodes when we are e.g., doing multi-class classification. In the case of multiple loss nodes there is an additional node that sums the losses together. This sum node can be ignored when doing the message passing because we want to minimize each node individually. In Figure 2 we show an example of a computational graph of an MLP with a sum node. The sum node is highlighted red because it can be ignored in the message passing.

Table 1: The 3 types of nodes used in the forward pass of an MLP. The variables can be explained as follows: x is the intermediate result generated by the previous node during the forward pass (from input to output). If a node does not have incoming nodes, x represents the input data. θ_c is a single weight/parameter at the coordinate c of the node. y is the ground truth. The outputs of the linear nodes get summed before being passed to the activation node, this is done implicitly and not shown.

	Input	Computation/$k(\dots)$
Linear	x, θ_c	$x \cdot \theta_c$
Activation	x	$act(x)$
Loss	y, x	$loss(y, x)$

Within the framework one can have multiple types of activation functions, e.g., Relu in the hidden layers and sigmoid in the output layer. The exact loss function that is used can also be chosen, e.g., cross-entropy for classification and mean squared error for regression.

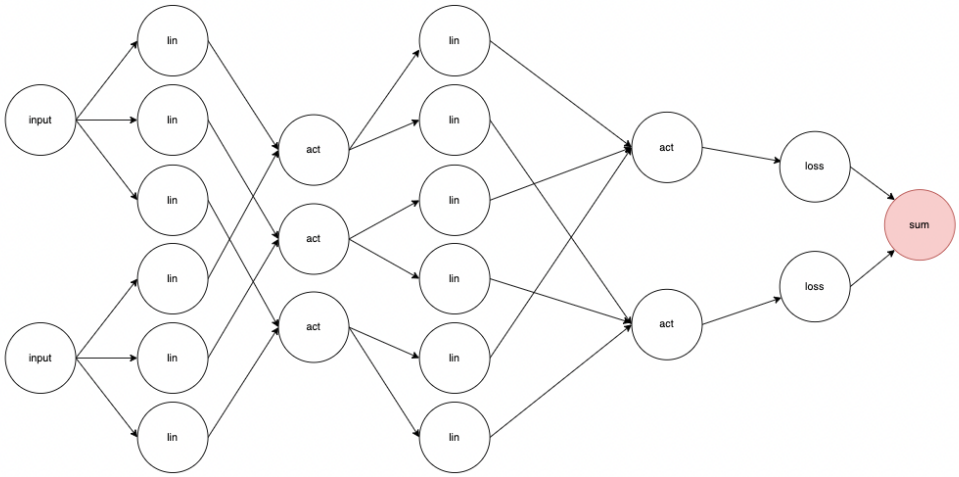


Figure 2: Example of how an MLP, with 2 input nodes, 1 hidden layer with 3 nodes and 2 output nodes, would look like in the MPLP framework. Note that the sum node does not pass messages in the backwards pass. Its mere purpose is to combine the losses at each output in the forward pass.

Backward pass

During the backward pass each node in the computational graph generates a message to be passed to the connected nodes in the direction of the input. This process is similar to the calculation of reverse mode differentiation [Griewank, 2012], where the error is propagated from the output to the input of the network. The messages are generated sequentially, starting from the loss nodes and ending at the input nodes.

For each node type, there exists a different function that generates and passes a message backward through the network:

$$g_{lin}(\phi_{lin}; m, \theta_c) \to \mathbb{R}^{|m|} \quad g_{act}(\phi_{act}; m, x) \to \mathbb{R}^{|m|} \quad g_{loss}(\phi_{loss}; m, x, y) \to \mathbb{R}^{|m|} \tag{4}$$

$|m|$ is the size of the generated messages. m is the incoming message. ϕ are the meta-parameters and are *not* shared between nodes of different types but *are* shared between nodes of the same type. There can be multiple types of activation nodes of which the meta-parameters are not shared. E.g. the message passing function of a ReLU node is different than from a Sigmoid node. In the original MPLP work there is also the option of not sharing any parameters but we have decided not to do this to keep the number of parameters low and decrease the risk of overfitting.

The incoming message of the loss node is the sum of the output of all the loss nodes. If there is only a single loss node then it is the same as the output of that node. This first message is not necessary for the MPLP to be able to learn but it can be hypothetically useful.

Activation nodes can receive multiple messages. In this case, the messages are summed together before being passed into the message generating function g,

$$m = \sum_{i=1}^{n} m_i$$

where n is the number of incoming messages. In the original MPLP framework the messages are not summed but averaged. We have decided to sum the results because this more closely resembles how gradients would normally be calculated. See Appendix B for further explanation.

With only message passing functions there would be no way to update the weights of the network. Therefore, in addition to the message passing functions there also exists a weight update function that calculates Δw (See Eq. 2). The weight update function is calculated for each linear node in a network:

$$f_{lin}(\phi_f; m, x) \to \mathbb{R}^1 \tag{5}$$

Similar to the message passing functions, m is the message that was passed backwards by the connected nodes. In the case of a MLP the message that f_{lin} receives is from the activation node (See Fig. 2). ϕ_f are the meta-parameters of the weight update function. These are shared between all linear nodes.

The update function f_{lin} is often extended by multiplying the output with an extra parameter, the learning-rate. The learning rate can then be learned. However, it is also possible to not learn the learning-rate and fuse the learning rate into f_{lin}. We investigate the effect of adding the extra learning-rate parameter in experiment 5.2. In any case, we scale the output of f_{lin} by 0.001 to prevent the weight updates to be too large at the beginning of outer-training. In addition to scaling, we clip the calculated weight updates between -1 and 1 such that the magnitude of the weights can not become too large too quickly. This helps in numerical stability.

In the original paper there are additional states introduced such that the MPLP can track information about the optimization process. E.g., momentum or second order momentum could be learned using these additional states. We have chosen to not use these additional states to keep the framework more simple and better understandable. These states could be added in the future if needed.
In the original paper bias nodes are treated as a separate node type with their own message generating function g_{bias}. We have decided to not do that in our implementation because of the similarities it has with linear nodes and thus we can treat them the same as linear nodes.

4.2 Expressiveness to imitate SGD

The way the architecture of the MPLP framework is constructed makes it able to imitate SGD without momentum. Table 2 shows what the functions g and f_{lin} have to be for the

MPLP framework to mimic SGD without momentum. The derivations of the functions can be found in Appendix A.

The reason the MPLP framework cannot imitate momentum or anything similar to momentum is that there is no state in which information can be saved between optimization steps.

Table 2: The definitions of g and f_{lin} for them to mimic SGD without momentum. Message size is 1. m is the incoming message. The way we derived the functions of Table 2 can be found in Appendix A.

Function	Definition
f_{lin}	$\alpha \cdot x \cdot m$
g_{lin}	$w \cdot m$
g_{act}	$act'(\sum_{i=1}^{n} x_i) \cdot m$
g_{loss}	$\frac{\partial}{\partial \bar{x}} loss(y, x)$

The crossentropy loss is a special case because it contains the softmax function which has cross-dependencies between the pre-activations and activations. The function $g_{crossentropy}$, such that SGD without momentum can be mimicked is defined and further explained in Appendix C.

4.3 Meta-training

Algorithm 1 shows the most standard L2O learning algorithm [Vicol et al., 2021]. The unroll function used in the algorithm is where the outer-loss gets calculated. This is the same as Eq. 3. After the full unroll is done, the resulting computational graph can be used to calculate the gradients of the outer-loss with respect to the outer-parameters. These gradients can then be used to update the outer-parameters using gradient descent. The algorithm is repeated for a certain amount of outer-optimization steps S. The algorithm can be extended by combining multiple gradient updates into a single update to create an outer-batch. The parameter T determines after how many inner-steps we sample a new task from our distribution of tasks. The truncation length K determines the amount of update steps we do before we calculate the gradients of our outer-parameters. K has a large impact on memory and computational resources needed for a single meta update step.

Algorithm 1 standard learning to optimize algorithm

Input: ϕ_0, initial outer-state
K, truncation length for unrolls
T, full horizon length
S, total number of outer optimization steps
$p(\mathcal{T})$, the task distribution
Initialize $\phi = \phi_0$
Initialize $\boldsymbol{t} = \infty$, current inner step
for $i = 1, \ldots, S$ **do**
  if $t + K \geq T$ **then** ▷ Reset the inner problem after T inner-update steps
    Randomly Initialize the inner-parameters, $\boldsymbol{\theta}$
    Sample $\mathcal{T} \sim p(\mathcal{T})$ ▷ Learned optimizer should work on multiple tasks
    $t = 0$
  end if
  $L, \boldsymbol{\theta} = \text{unroll}(\boldsymbol{\theta}, K, \phi, \mathcal{T})$ ▷ Calculate outer-loss and update the inner-parameters
  $g = \frac{\delta L}{\delta \phi}$ ▷ Can also be an estimation of the gradient
  $\boldsymbol{s} = \text{update}(\phi, g)$ ▷ do the meta descent step
  $t = t + K$
end for

We use Algorithm 1 to train the MPLP framework. The computational graph is being unrolled by alternating the forward and backward pass similar to how SGD works. After K iterations of unrolling we calculate the outer-loss using e.g., Eq. 3. We then update the meta-parameters such that we expect the outer-loss to decrease. This process is repeated many times. Every T iterations of unrolling we resample a task and reinitialize the inner-parameters. We need to resample a task because we want the optimizer to generalize to more than a single task.

In our implementation we clip the outer-gradients between -1 and 1 to counteract any potential exploding gradients.

For more details on how the unroll functions works, we have it further specified in Algorithm 2.

In the original paper the MPLP framework is only trained for few-shot learning. That means that the total number of unrolls T done on a new task and inner-initialization is very small. Then the truncation length K is set to be equal to T such that full unrolls are done. Instead of this, we use longer unrolls with more truncations to allow the optimizer to work for longer time horizons.

4.4 Batch entropy regularization

Batch entropy regularization is a technique that was initially developed to facilitate the trainability of deep neural networks (with at least 30 layers) without relying on additional methods such as normalization or residual connections [Peer et al., 2022]. In this work, we adapt the batch entropy regularization approach to enhance the trainability of MPLP-like learning to optimize methods. We begin by outlining the central concept of batch entropy regularization and subsequently illustrate its applicability as a valuable measure for improving the trainability of MPLP-like learning to optimize methods.

The key idea behind batch entropy regularization is to analyze and optimize the flow of information through individual layers in a neural network. This is achieved by quantifying the flow of information as the average amount of information propagated through each neuron within a layer. Intuitively, a single activation node that always fires or never fires is a superfluous activation node. Introducing a regularization term based on this idea enables the training of deep neural networks without additional training techniques. Batch entropy for a *single* node is defined as follows,

$$H = \frac{1}{2}\log(2\pi e\ \sigma_j^2 + 1), \tag{6}$$

where σ_j^2 is the standard deviation of a single activation node within a neural network of an inner-batch.
Then the batch entropy of an entire layer can be calculated by averaging over the nodes within that layer. Mathematically expressed as follows,

$$H^l = \frac{1}{2n}\sum_{j}^{n}\log(2\pi e\ \sigma_j^2 + \epsilon), \tag{7}$$

where σ_j^2 is the standard deviation of neuron j of layer l.

Now we will discuss how this batch entropy can be used in a learning to optimize setting. The main idea is that we can learn an optimizer to increase the entropy within a network by incorporating a batch entropy term within the outer-loss. Learning to increase the entropy could be a simpler task than learning to optimize. After the optimizer has learned to increase the entropy the optimizer can proceed to learn how to optimize. This approach is particularly beneficial when the optimizer encounters difficulties during the initial outer-training phase. For instance, as illustrated in Figure 9b, an increase in batch entropy is associated with the optimizer overcoming obstacles during the initial training stage. Consequently, optimizing for higher entropy could contribute to the training process of the optimizer.

For our purposes, we are not interested in finding the optimal batch entropy for each layer but rather want to regularize such that the batch entropy will never fall below a certain threshold value. This can be expressed as follows,

$$L_{be}^l = \begin{cases} p - H^l, & \text{if } H^l \leq p \\ 0, & \text{otherwise} \end{cases}, \qquad L_{be} = \frac{1}{l}\sum_{j}^{l} L_{be}^j \tag{8}$$

Where $p = 0.5$ is the threshold value. H^l is the entropy for a layer l in a network. The final batch entropy loss L_{be} is the average of all the layers in the network.
From anecdotal evidence, we found that the threshold parameter p is not sensitive. We can partly explain this by that the batch entropy regularization does not compete with the learning to optimize and the regularization merely serves as a way of finding a good starting state.

Concretely, we calculate the batch entropy after each forward pass during the unrolling of the computational graph. We scale the batch entropy loss such that $L_{be} \in [0, L_{final})$ where L_{final} is the final loss in a single unroll. The exact procedure is shown in Algorithm 2.

Algorithm 2 unrolling

Input: θ_0, starting inner-state
K, truncation length / number of unrolls
ϕ, outer state
$\mathcal{T}$, task
$\theta = \theta_0$
$\text{agg}_{L_{be}} = 0$ ▷ Keep track of the batch entropy loss
$\text{agg}_L = 0$ ▷ Keep track of the regular outer loss
for $i = 1, \ldots, K$ **do**
  $x, y = nextbatch(\mathcal{T})$
  $\hat{y}, L_{be}, o = forward(\theta, x)$ ▷ o is intermediate info that is needed in the backwards pass
  $L = loss(\hat{y}, y)$
  $\text{agg}_{L_{be}} \mathrel{+}= L_{be}$
  $\text{agg}_L \mathrel{+}= L$
  if $i < K$ **then** ▷ Only do backwards if loss is calculated afterwards
    $\Delta\theta = backward(\phi, x, y, \hat{y}, o)$ ▷ here we use o from the forward pass
    $\theta = \theta - \Delta\theta$
  end if
end for
$L_{be} = 2 \cdot \text{agg}_{L_{be}}/K$ ▷ scale the loss between 0 and 1
$L_{be} = L \cdot L_{be}$ ▷ Give same weight to L_{be} as to latest L, L is detached from comp. graph
$L_{outer} = \text{agg}_L$
$L_{total} = L_{outer} + L_{be}$
return L_{total}, θ

5 Experiments and results

In this section we describe the experimental settings and our results.

In the first experiment, we learn an optimizer with a message size of 1 such that we can visualize what the MPLP has learned. In the second experiment, we experiment with design

choices to see how they affect the training of the optimizer. In the third and final experiment, we train the MPLP on a more realistic problem, namely, MNIST. We find problems during the initial training phase which we solve using batch entropy regularization.

5.1 Learning SGD

In this experiment, we are going to train the MPLP on the sinewave problem. We use a message size of 1 such that we can easily visualize what functions the MPLP has learned. We compare the learned optimizer against SGD and see if they are similar.

The setting that was used in this experiment is one of the simplest. The function of the experiment is to get a better understanding of the training dynamics and to compare the learned optimizer to SGD.

5.1.1 Setup

We will train the MPLP and compare it to the functions found in Section 4.2. We have summarized the configuration of the experiment in Table 3.

Table 3: Overview of the setting that was used to train the MPLP. The outer-architecture was used for all functions f, g_{lin}, g_{act} and g_{mse}. $|m|$ is the message size, T is the full horizon length, K is the truncation length and S is the number of meta steps.

	Inner	**Outer**
Architecture	lin(1,20),ReLU,lin(20,20),ReLU,lin(20,1)	lin($\|i\|$,80),ReLU,lin(80,$\|o\|$)
Optimizer	MPLP	Adam β_1: 0.99 β_1: 0.999
Task family	$y = sin(x + p)$, $p \in \mathcal{U}(0, \pi)$	-
Additional params	batchsize: 32	T: 200, K: 10, S: $3 \cdot 10^5$, $\|m\|$: 1

The task family on which we train our optimizer is that of the sinewave. A sinewave task is defined as follows, $y = sin(x + p)$ where p is sampled from $\mathcal{U}(0, \pi)$. This task family has only a single parameter p, this is to keep the problem simple. Before every forward pass 32 new train points are sampled from the task where $x \in \mathcal{U}(-5, 5)$.

The architecture of the network that is optimized by the MPLP is static and not changed throughout the training. Thus the MPLP is only trained on a single MLP architecture and will probably not generalize to other architectures.

We set the message size to 1, this means that each node will only receive a single floating point during the backward pass. If we would use a higher message size, we would not be able to easily compare the learned optimizer to SGD (because SGD also uses message size 1 essentially). Higher message size would also be harder to visualize because of the extra dimensionalities. From the results in Section 5.2 we see that higher message sizes give much better results for this specific task. Therefore, it would be interesting to find a way to better understand the found optimizers with message sizes larger than 1.

The inputs to the message passing functions are the same as in Eq. 4 except for the message passing function of the loss node. Instead of the ground truth, the prediction and the incoming message we only use the ground truth and the prediction as our input. We do this to keep the dimensionality lower for the visualization. The incoming message is not necessary to imitate SGD (Table 2).
The inputs to the weight update function is the same as in 5.

We validate the optimizer on 10 different randomly initialized neural networks and 10 randomly chosen sinewave tasks using $p \in \mathcal{U}(0, \pi)$. The tasks and initializations on which we validate are chosen at the beginning of training and are not changed throughout the training.

5.1.2 Results

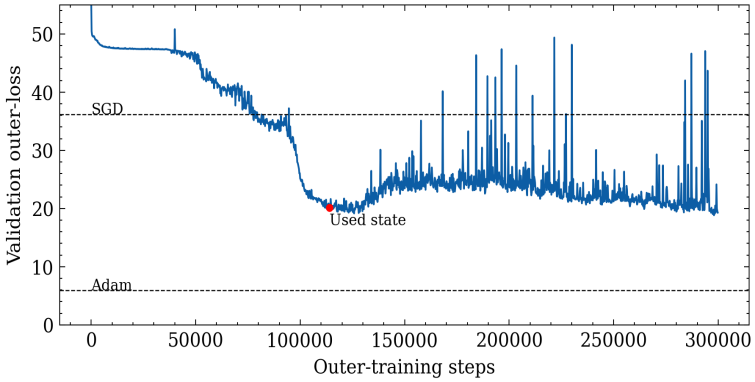
The outer-training process of the optimizer is shown in Figure 3a. We observe an initial flat area in the loss curve, after around 50000 steps it escapes this phase and starts learning something meaningful. The training of the MPLP is not very predictable i.e. the learning curve does not decrease monotonically and shows multiple phases of descent.

We randomly chose an optimizer state between 100000 and 130000 outer-training steps because this area showed relatively good performance without many outliers. This state is then used in the rest of the results section. The red dot in the Figure 3a shows the chosen state.

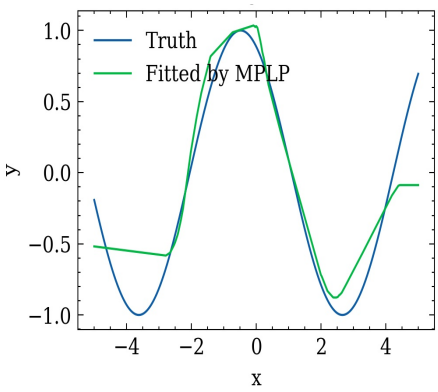
To give an intuition of how well the optimizer performs we have given an example in Figure 3b of the learned optimizer fitting a neural network to a sinewave from the same task family it was trained on.
In Figure 3a we see that our optimizer performs worse compared to learning-rate-tuned Adam on the outer-loss metric. It does, however, perform better than learning-rate-tuned SGD. Note that this comparison is made after 200 inner-update steps. If we would use the optimizer for more steps SGD would eventually outperform the learned MPLP.

It is convenient to visualize the learned functions because there are always only two inputs and a single output. Therefore, we can simply plot the functions in 3D space. Next to the message passing functions g and the update function f_{lin}, we also plot the functions that mimic SGD without momentum from Table 2.

The ranges of the plots were established by using the found optimizer to inner-train 20 newly sampled tasks and keeping track of the ranges of the inputs during this inner-training. We took the 1st and 99th percentile of the recorded values as the range.



(a) Outer-learning process of a MPLP on a family of sinewave tasks. The loss is the average of 10 meta-validation runs. Initial learning phase is flat until around 50000 steps, after which it starts converging. The red dot is the checkpoint we used to compare to SGD. The horizontal lines are the performance of learning-rate-tuned SGD and Adam.



(b) An example of how the learned MPLP can fit a sinewave. The optimizer that was used is the red dot from Figure 3a.

Figure 3: Outer-training (left) and example fit (right) of the learned optimizer.

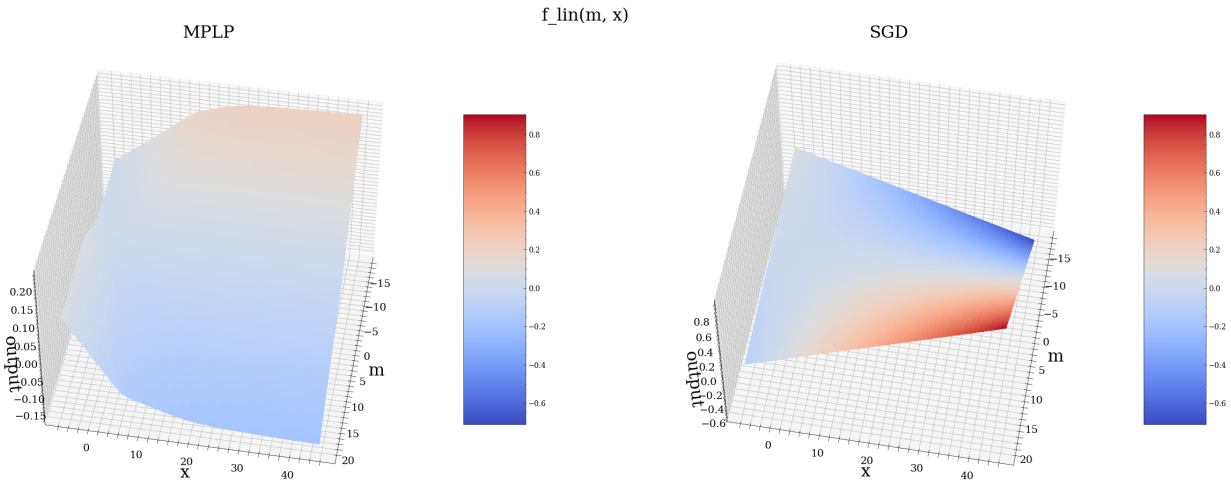


Figure 4: Left: learned update function. Right: update function of SGD. Shape of MPLP is negated version of the shape of SGD.

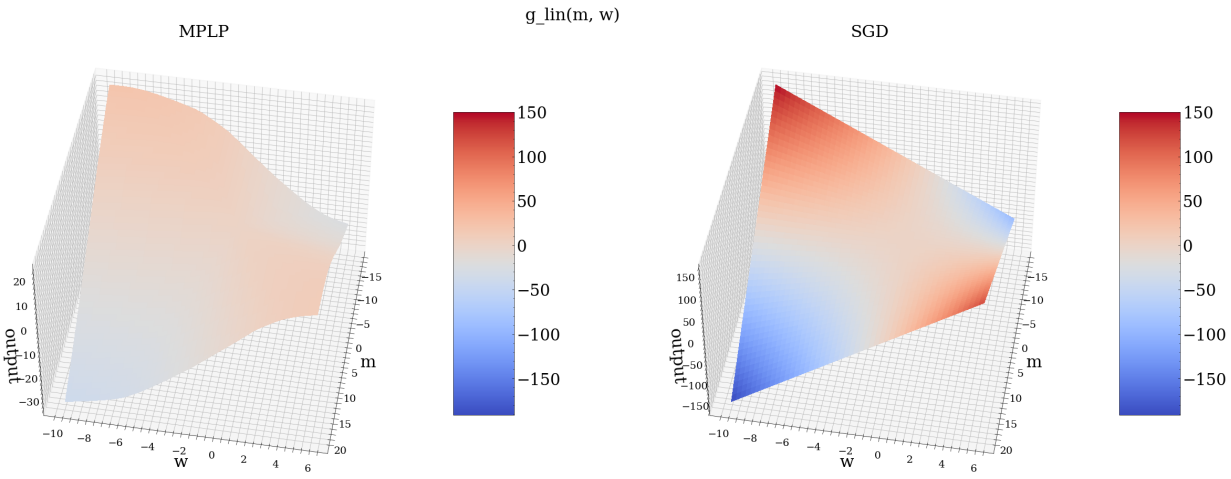


Figure 5: Left: learned message generating function of the linear node. Right: SGD message generating function of the linear node. Shapes are very much similar. Both represent multiplication.

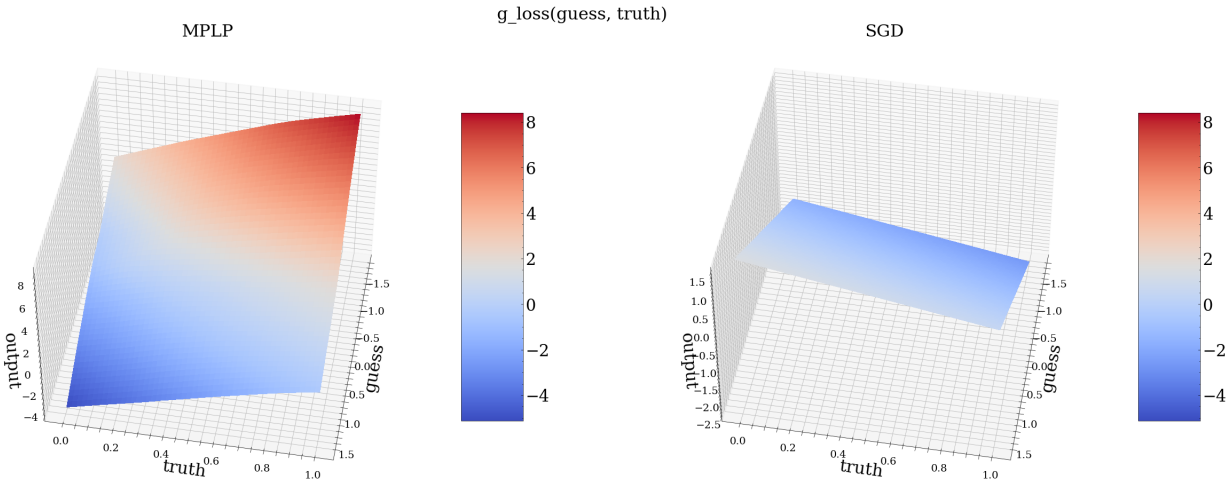


Figure 6: Left: learned message generating function of the loss node. Right: SGD message generating function of the loss node. Shapes are the same but MPLP has learned $\hat{y} - y$ while SGD is $y - \hat{y}$.

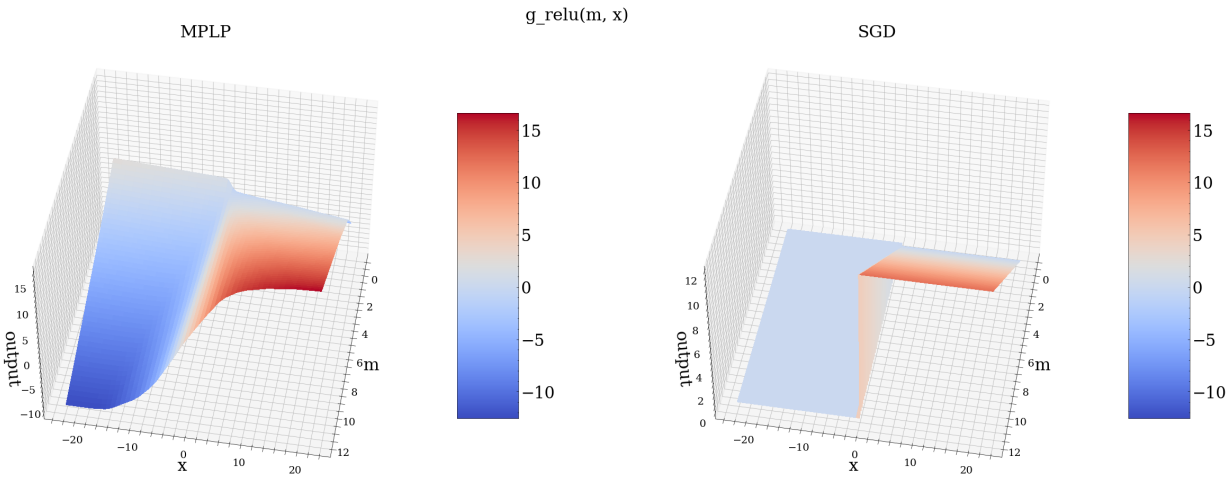


Figure 7: Left: learned message generating function of the ReLU node. Right: SGD message generating function of the ReLU node. Shapes show similar properties but MPLP can return negative values. Which is never the case for the ReLU activation function.

Looking at the results, we suspect that the learned optimizer has learned SGD but is less precise and only works for a certain input distribution. We can see that for some learned functions the shape of the surface is very similar. E.g., g_{relu} (Fig. 7) and especially g_{lin} (Fig. 5) are very similar to what the function of SGD looks like. g_{loss} (Fig. 6) is also similar to the SGD function however instead of $y - \hat{y}$ it has learned to do $\hat{y} - y$. This means that the "gradients" (the messages) that flow backwards are negated. This explains how the update function f_{lin} (Fig. 4) relates to SGD because the f_{lin} function is also the negated version of the SGD function.

We noted some interesting things from the surface plots. First of all, the learned g_{relu} shows negative values for $x < 0$. If we look at the learned function as if it would calculate the derivative of the ReLU activation function it suggests that there is a decreasing slope in the ReLU activation function, which there is clearly not. In Appendix D we show what activation function the learned g_{relu} is the derivative of. The differences we find in Figure 13 could be explained by that an approximation is good enough to reach the outer-loss that we did. Note that the outer-loss is fairly high (around 25).

The second interesting thing we noticed is that the messages that g_{relu} receives are mostly non-zero (minimum value is -0.57). It is unclear to us why this happens. It would be helpful to compare this to SGD and see if there is a large difference.

5.2 Training dynamics

From Figure 3a we observed that the outer-training behaviour of the MPLP is not well behaved. First of all, there is an initial phase in which it does not learn, this phase can take,

depending on your computing power and configuration, a few hours. During these few hours you are uncertain if it is going to train at all and this can make developing these systems very painful. Secondly, the training can generally take a long time, can we improve on the training time?

In this experiment we investigate what the effect of certain hyperparameters are on the outer-training dynamics.

5.2.1 Setup

For the experiments we use sinewave tasks with not only random phases but also random amplitude. This is done to make the MPLP also learn tasks of different scales which arguably gives a more realistic setting.

Table 4: Overview of the setting that was used to train the MPLP. The outer-architecture was used for all functions f, g_{lin}, g_{act} and g_{mse}. T is the full horizon length, K is the truncation length and S is the number of meta steps.

	Inner	**Outer**
Architecture	lin(1,20),ReLU,lin(20,20),ReLU,lin(20,1)	lin($\|i\|$,50),ReLU,lin(50,$\|o\|$)
Optimizer	MPLP	Adam β_1: 0.99 β_1: 0.999
Task family	$y = a \cdot sin(x + p)$, $p \in \mathcal{U}(0, \pi)$, $a \in \mathcal{U}(1, 5)$	-
Additional params	batchsize: 32	T: 100, K: 5, S: $1.2 \cdot 10^5$

The inputs that the message generating functions and weight update function receive are the same as in Eq. 4 and Eq. 5.

We will introduce a set of design choices that could affect the training behaviour of the MPLP. These different design choices are:

- *Learning a learning rate:* Up until this point there was no explicit learning rate that was learned during the training of the MPLP. The learning rate had been explicitly fused into the update function f. We are going to add an outer-parameter to the MPLP which is a scaler to the output of the update function f also known as the learning rate.
- *Normalization:* In the original paper it was mentioned that normalization can have a positive effect on the outer-training process [Randazzo et al., 2020]. E.g., it was mentioned that the MPLP could not be trained without normalization under certain configurations. We decided to use batch normalization because it gave the best performance compared to other normalization methods we have experimented with [Ioffe and Szegedy, 2015]. We do not keep running averages of the mean and std in the batch normalization because of the large changes in these statistics between tasks. The normalization is applied on the input of the functions f, g_{lin}, g_{act} and g_{mse}.

- *Increasing the message size:* Increasing the message size allows the MPLP to pass more information through the network. We hypothesize that a higher message size allows for more descent directions and could help to improve the initial phase of the training. In the original paper it is stated that both performance and outer-training speed improves with higher message sizes but no evidence is given [Randazzo et al., 2020]. We experiment with a message size of 1 and 8.

We validate the optimizer on 20 different randomly initialized neural networks and 20 randomly chosen sinewave tasks using the same task family we train on (random amplitude and random phase). The tasks and inner-initializations on which we validate are chosen at the beginning of training and are not changed throughout the training. We share the same inner-initializations and tasks used for validation across all experiments. We validate every 300 outer-training steps for computational reasons.

5.2.2 Results

We have run each configuration 5 times to take into account the randomness of the meta-initialization. We have plotted the median and the standard deviation over the 5 runs in Figure 8. Due to computational constraints, this is the maximum we could reasonably achieve.

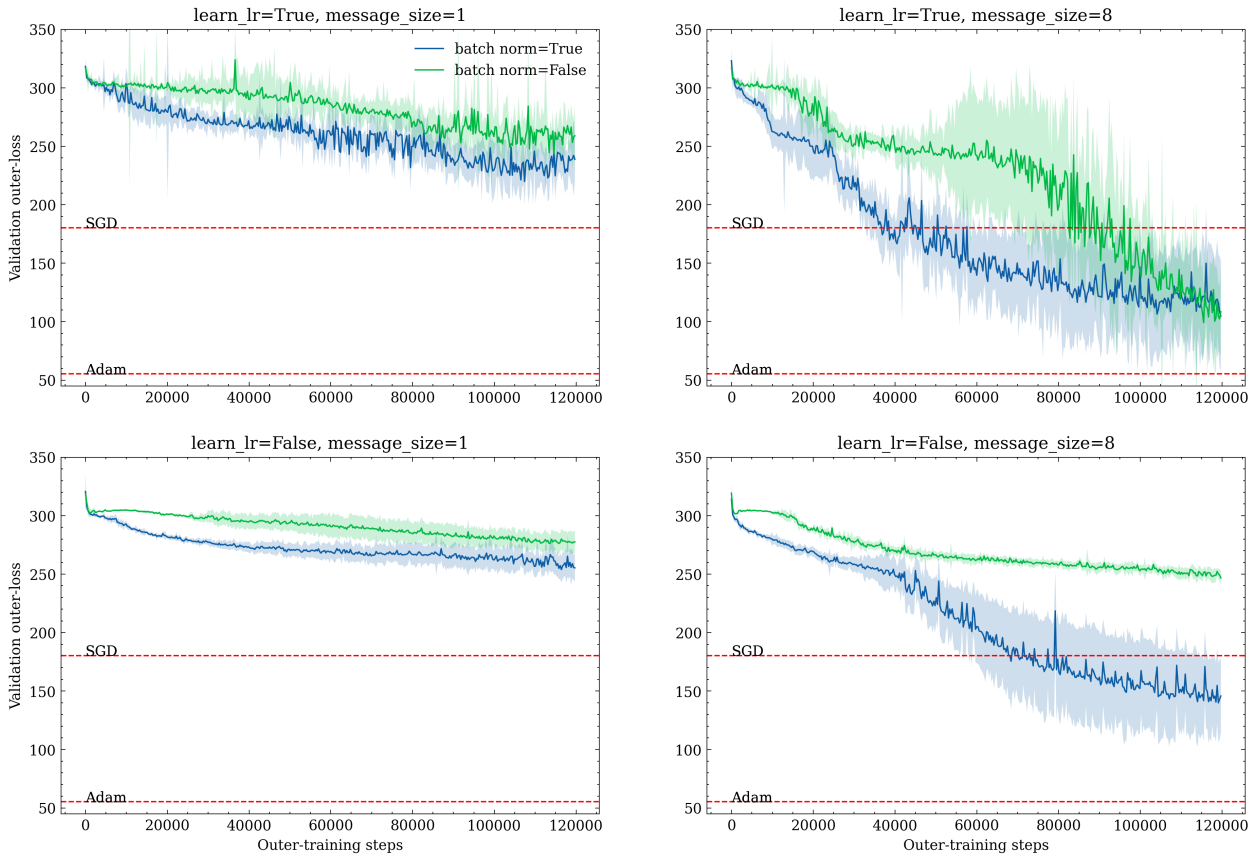


Figure 8: Outer-training curves of all possible combinations of the hyperparameters that we have experimented with. Batch normalization has shown faster convergence under all configurations. Increasing the message size leads to a much lower outer-loss. Learning the learning-rate especially has mainly shown advantages in the top-right configuration when there is no batch normalization applied. The red horizontal lines show the performance of learning-rate-tuned Adam.

From the results in Figure 8 we find that first of all, batch normalization improves convergence speed in all settings. Especially in the bottom-right configuration, we find a large difference for the use of batch normalization. Additionally, when using batch normalization the training of the optimizer less often gets stuck at the beginning of the training.
Secondly, learning the additional learning-rate parameter shows mixed results. We found that it would generally speed up convergence but in some cases more than others. E.g. when the message size is 1 the speed up is minimal, if even significant, but for message size 8 we find a large difference when no batch normalization is used. A disadvantage of the additional learning-rate parameter is that it seems to make the training more unstable. We hypothesize that this is due to the general sensitivity of the learning rate parameter [Metz et al., 2019].
Thirdly, increasing the message size has, as expected, increased the performance of the opti-

mizer significantly, it allows more information to be passed backwards through the network. Using these results we can not accept the hypothesis, that a higher message size can prevent the optimizer from getting stuck in the beginning phase, at least in this setting. Using the higher message size we still find scenarios where there is an initial flat training phase.

Using a learned learning-rate in combination with batch normalization and a higher message size gave the best performance. The performance of this configuration is still worse than learning-rate-tuned Adam. Especially since the learned MPLP is only trained to optimize for 100 update steps and has no guarantees to work well on other families of tasks. The optimizer does perform better than SGD for the best performing configuration.

5.3 Training on MNIST

Up until now we have only experimented with tasks from the sinewave family. To learn an optimizer that can optimize neural networks for more realistic problems, we have to also train the optimizer on more realistic problems. Therefore, in this experiment, we train the optimizer on the MNIST problem [Lecun et al., 1998].

5.3.1 Setup

Table 5: Overview of the setting that was used to train the MPLP. The outer-architecture was used for all functions f, g_{lin}, g_{act} and g_{mse}. T is the full horizon length, K is the truncation length, S is the number of meta steps and $|m|$ is the message size.

	Inner	**Outer**
Architecture	lin(28*28,32),ReLU,lin(32,20),ReLU,lin(20,10)	lin($\|i\|$,40),ReLU,lin(40,$\|o\|$)
Optimizer	MPLP	Adam β_1: 0.99 β_1: 0.999
Task family	MNIST train split	-
Additional params	*batchsize*: 32	T: 100, K: 5, S: $1.5 \cdot 10^5$, $\|m\|$: 24, *batchnorm*: True, *learnedlr*: True

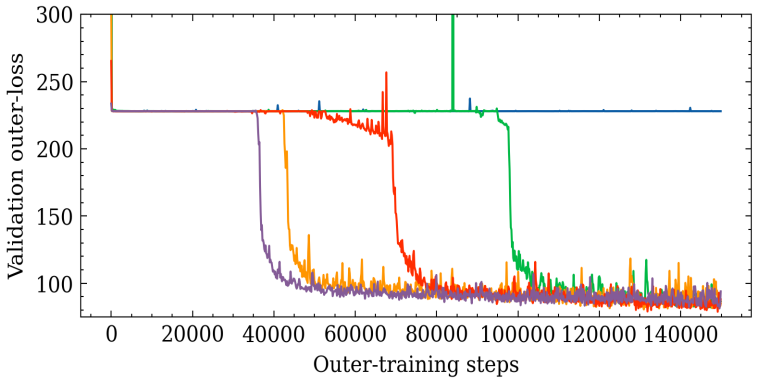
Because MNIST is a classification problem we use the crossentropy loss. The crossentropy includes a softmax activation to normalize the incoming logits. The definition of the crossentropy can be found in Eq. 23.

The inputs that the message generating functions and weight update function receive are the same as in Eq. 4 and Eq. 5.

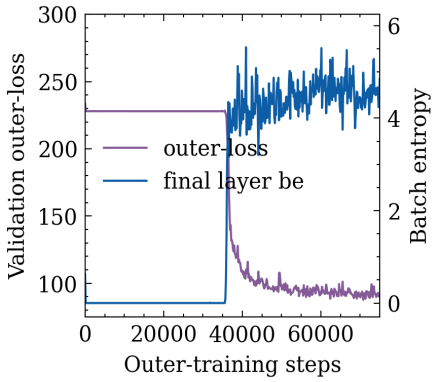
We validate the optimizer on the first T batches of the MNIST test set. After every 150 outer-training steps we validate the optimizer.

5.3.2 Results

We run the experiment 5 times to account for the random initialization of the meta-parameters. We have plotted the results in Figure 9a.



(a) Training the optimizer on MNIST shows a long initial phase where it gets stuck. It can start to learn but it is hard to predict when this happens. In one of the runs it did not start to learn even after 150000 steps.

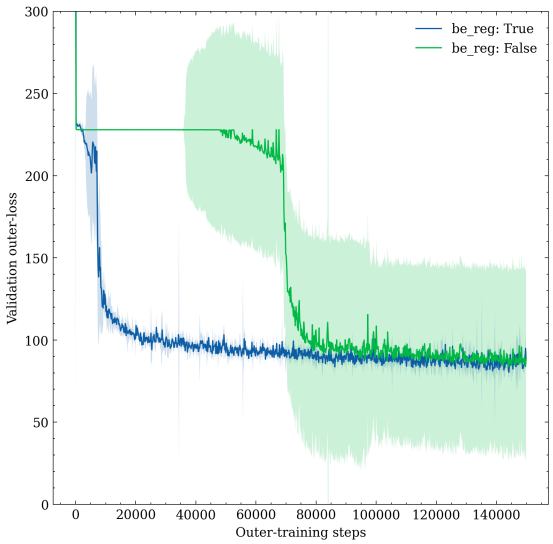


(b) Purple line is the same training session as the purple line in Figure 9a. Batch entropy of the final layer of the optimizee increases when the optimizer starts to learn how to optimize. *No* batch entropy regularization has been applied.

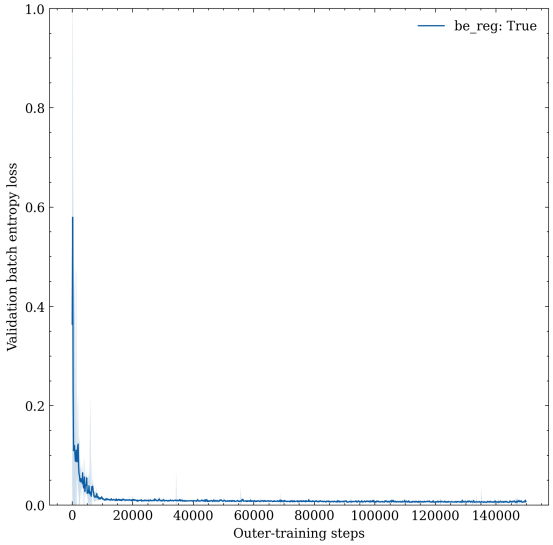
Figure 9

From Figure 9a we find that there is a long initial phase where it is not learning. It can however start to learn but it is hard to predict when this exactly happens. In one of the runs the optimizer did not start to learn after 150000 outer-training step. This is especially problematic when you are developing the optimizer, make changes and are not sure if the change you just made has caused the optimizer to not be able to be trained anymore.

In Figure 9b we find that the batch entropy seems to be correlated with the outer-loss. I.e. that regularization on the batch entropy property could prevent the initial flat phase. To test this hypothesis we add the regularization term as described in Section 4.4. The rest of the configuration stays the same. The results are visualized in Figure 10b.



(a) Batch entropy regularization removes the initial flat phase and allows for immediate training.



(b) The batch entropy component to the loss quickly decreases to a value close to 0. This shows the batch entropy does not compete with the other loss term.

Figure 10

From Figure 10a we find that batch entropy regularization has resulted in the disappearance of the initial phase in which the optimizer did not train. This enables us to train the optimizer with much more confidence.

Figure 10b shows the batch entropy loss during the training of the optimizer. We find that after the batch entropy loss is decreased it stays close to 0. This is in line with Figure 9b, which shows that learning the optimizer correlates with an increase in batch entropy. Note that the batch entropy is only regularized if it falls below a certain threshold (See Eq. 8).

We want to emphasize that we have only tested batch entropy regularization for this specific configuration. Different optimizer types, e.g, learned optimizers that only learn an update function might not benefit from batch entropy regularization. But there will probably also be configurations in which the optimizer would not be able to train without batch entropy regularization. Thus, batch entropy regularization should be seen as a tool to enable the training of optimizers when without it, the optimizer does not train.

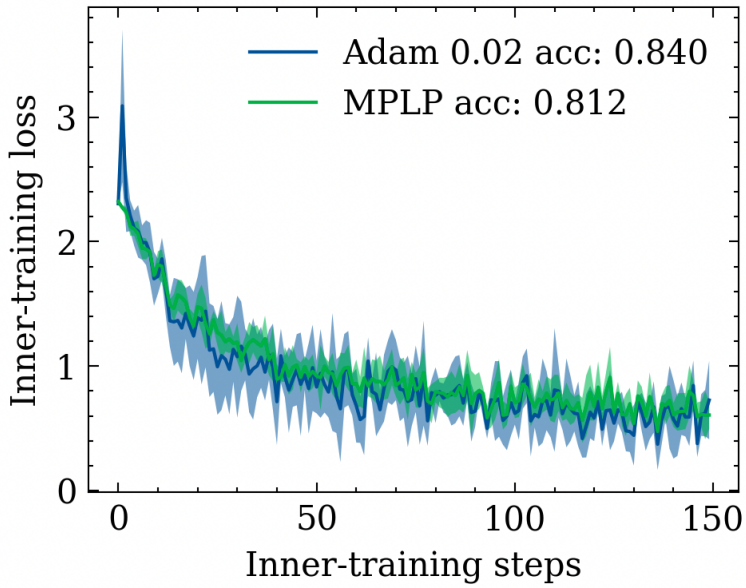


Figure 11: The learned MPLP and Adam show similar convergence-speed on MNIST.

Figure 11 shows the inner-training of the learned optimizer on the MNIST test set on 20 random inner-initializations. We find that the learned MPLP performs similarly to tuned Adam. This is interesting because the learned MPLP optimizer cannot calculate statistics similar to momentum. After all, it can not transfer information between update steps. This suggests that the learned MPLP has found a fundamentally different way to speed up the training process.
We are limited in the amount of inner-training steps we could do because the MPLP is only trained for a limited amount of inner-training steps. Therefore, if we would run the MPLP optimizer for a longer amount of time Adam would likely outperform MPLP.

6 Discussion & future work

In our experiments, we found that different configurations showed different outer training dynamics. E.g., in Section 5.2 (Sinewave) for all runs we could successfully train the MPLP without using batch entropy regularization, whereas in Section 5.3 (MNIST) batch entropy regularization was necessary. The main differences between the two experiments are the data on which the optimizer was trained on and the architecture of the optimizee. E.g., MNIST classification requires more inputs and outputs in the optimizee than Sinewave regression does. It is currently unclear what exactly causes the training problems in the initial training phase.
We hypothesize that the RNN-like training in combination with the nature of the problem makes the training process of these expressive optimizers sensitive to their exact configuration. Understanding what exactly has caused the training problems in the MNIST ex-

periments might not solve training problems that will be encountered in similar expressive optimizers. E.g., adding more input features to the message generating functions could cause new training problems which are not solved by understanding what caused the training to get stuck for the MNIST experiment. Therefore, to reliably train optimizers similar to the MPLP, some sort of regularization seems to be required.

Learning to optimize from scratch, e.g., without explicitly providing the gradients, seems to be a problem that is hard to solve using a greedy strategy during the beginning phase. At the beginning of the outer-training there is no clear descent direction that causes consecutive steps of decrease in the outer-loss. After the initial phase the training shows a more preferred monotonic trend. We think batch entropy regularization allows us to more quickly find the outer-state in which phase two of the outer-training starts.

Furthermore, in general, there are still many hyperparameters to the MPLP framework that can be experimented with. There are still many arbitrary choices that have been made. E.g. the architecture of the update functions and the message generating functions is still fairly arbitrary. It would be interesting to change the width and depth of the architecture or to change the activation functions. Currently, the output of the message generating functions is not activated. Also, the inputs to the update function and the message generating functions can be extended. Right now we have decided to use the same inputs as SGD would normally have access to. In the literature, often, running statistics of the weights, current timestep of the inner-optimization etc. are extra inputs that are available [Metz et al., 2022]. Often, the current weight is also given to the update function f_{lin} to allow the MPLP to learn to do weight decay.

Currently, we have only visualized a suboptimal performing optimizer because we are depending on the dimensionality of the message size for the visualization. From our experiments, we found that message size has a large effect on performance, therefore, we would like to be able to understand where this performance increase comes from. For this, we would have to think of ways to visualize the higher-dimensional messages. One direction to go in is to use dimensionality reduction on the messages. However, this would still make visualization difficult. E.g., reducing the dimensionality from 16 to 2 still leaves us with 5 dimensions (2 outputs, 3 inputs) to visualize. Therefore, the visualization methods have to be changed. What is possible is to look at how the weights of the optimizee get updated compared to other optimizers like Adam and/or SGD. Kirsch and Schmidhuber [2021] compared the accuracy of the optimizer during the inner-optimization process to SGD. But also, empirical step sizes could be compared to see if this is significantly different from how the status quo optimizers work. If we can find ways to better understand these learned optimizers, they can be used as a good source of inspiration to improve current optimizers.
Especially, optimizers that would generalize better on a wider set of optimizee architectures and tasks would be interesting to better understand if we see a performance increase. The distributions of the architectures and the tasks can be relatively small to make the training process more simple.

In this thesis, we have only experimented with TBPTT with short unrolls to train the MPLP. Despite seeing acceptable results using this method, it would be interesting to see how the training dynamics would change when we would e.g. use Evolution Strategies with longer unrolls to train the MPLP. The literature suggests that when using longer unrolls the outer-loss landscape becomes more chaotic [Metz et al., 2019] which are better suited to optimization methods that smooth the outer-loss landscape e.g. Evolution Strategies. Currently, this is too computationally expensive on our available hardware.

In general, this field would benefit from a more diverse benchmark dataset that can be used to train and test the optimizer. This could serve as a guideline for finding methods that better generalize but also make the training process computationally cheaper. E.g. it is currently unknown what would be a good schedule for truncation lengths during training to achieve good performance with relatively little computation. The datasets in this benchmark have to be relatively small and diverse. The benchmarks that exist are mostly designed for optimizers that are less computationally expensive to train/use [Metz et al., 2022].

7 Conclusion

The first research question we will try to answer is: *How does the MPLP framework relate to SGD?*

In Section 4.2 we show what form each component of the MPLP should take to imitate SGD. Thereafter in Section 5.1 we show in an experiment that the MPLP has highly likely learned an optimization algorithm similar to SGD.

For settings where the message size is larger than 1, it is harder to understand what kind of learning algorithm is learned. Because the messages are passed in a similar way to how gradients flow through a computational graph it is likely that for larger message sizes, in addition to other information, still some gradient-like information is learned.

What are the meta-training dynamics of MPLPs, and what techniques can we use to improve them?

This question consists of multiple questions. The first one, very broad, about the outer-training dynamics of MPLPs. First of all, the way we do the outer-training gives us similar dynamics as to how RNNs are trained, namely, there is a recurrent dependence because we calculate the gradients over the inner-optimization process. This can lead to unstable and chaotic training as discussed by Metz et al. [2019]. These dynamics apply to all learned optimizers.
What is more unique to the MPLP is that we have encountered situations where there is an initial phase where the training is stuck. We hypothesize that the additional expressivity of the MPLP gives these initial training problems.

To answer the second part of the question, we have experimented with multiple techniques to improve the training of the MPLP. Firstly, we have experimented with increasing the size

of the messages that get generated by the message generating functions. For our setting we saw that the message size has a large influence on the performance and convergence speed. Secondly, batch normalization increased convergence speed in all cases and generally produces a better performing MPLP. Furthermore, learning the learning-rate showed increased performance in our setting. It did however make the training more chaotic.

Additionally, we have modified an existing technique called batch entropy regularization to improve the trainability of the MPLP framework on the MNIST problem. We show how batch entropy correlates with certain training stages. We then show that this can be exploited by adding the batch entropy as a regularization term. Batch entropy regularization could especially be helpful to enable the training of expressive learned optimizers similar to the MPLP.
The learned MPLP shows similar performance to tuned Adam on the MNIST setting without making use of running statistics like momentum. This suggests that the MPLP has learned a fundamentally different technique to speed up the training.

Given the gained knowledge about training an MPLP, we think the most logical next step would be to train an MPLP to make it generalize on a wider set of optimizee architectures and tasks. If this optimizer would show better performance than e.g. tuned Adam it would especially be interesting to try to better understand the mechanics of this optimizer such that we can ultimately discover new optimization techniques.

References

Akyürek, E., Schuurmans, D., Andreas, J., Ma, T., and Zhou, D. (2023). What learning algorithm is in-context learning? investigations with linear models. In *The Eleventh International Conference on Learning Representations*.

Andrychowicz, M., Denil, M., Gómez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc.

Bengio, S., Bengio, Y., and Cloutier, J. (1995). On the search for new learning rules for anns. *Neural Processing Letters*, 2:26–30.

Bengio, Y., Bengio, S., and Cloutier, J. (1991). Learning a synaptic learning rule. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume ii, pages 969 vol.2–.

Brown, R. E. (2020). Donald o. hebb and the organization of behavior: 17 years in the writing. *Molecular Brain*, 13(1):55.

Chandra, K., Xie, A., Ragan-Kelley, J., and Meijer, E. (2022). Gradient descent: The ultimate optimizer. *Advances in Neural Information Processing Systems*, 35:8214–8225.

Deng, J., Dong, W., Socher, R., Li, L., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, pages 248–255. IEEE Computer Society.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135. PMLR.

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

Griewank, A. (2012). Who invented the reverse mode of differentiation.

Harrison, J., Metz, L., and Sohl-Dickstein, J. (2022). A closer look at learned optimization: Stability, robustness, and inductive biases. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A., editors, *Advances in Neural Information Processing Systems*, volume 35, pages 3758–3773. Curran Associates, Inc.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society.

Hochreiter, S., Younger, A. S., and Conwell, P. R. (2001). Learning to learn using gradient descent. In Dorffner, G., Bischof, H., and Hornik, K., editors, *Artificial Neural Networks - ICANN 2001, International Conference Vienna, Austria, August 21-25, 2001 Proceedings*, volume 2130 of *Lecture Notes in Computer Science*, pages 87–94. Springer.

Hollmann, N., Müller, S., Eggensperger, K., and Hutter, F. (2022). Tabpfn: A transformer that solves small tabular classification problems in a second.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. R. and Blei, D. M., editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456. JMLR.org.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y., editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Kirsch, L., Harrison, J., Sohl-Dickstein, J. N., and Metz, L. (2022). General-purpose in-context learning by meta-learning transformers. *ArXiv*, abs/2212.04458.

Kirsch, L. and Schmidhuber, J. (2021). Meta learning backpropagation and improving it. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W., editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 14122–14134.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Li, K. and Malik, J. (2017a). Learning to optimize. In *International Conference on Learning Representations*.

Li, K. and Malik, J. (2017b). Learning to optimize neural nets. *CoRR*, abs/1703.00441.

Liu, H., Simonyan, K., and Yang, Y. (2019). DARTS: Differentiable architecture search. In *International Conference on Learning Representations*.

Lv, K., Jiang, S., and Li, J. (2017). Learning gradient descent: Better generalization and longer horizons. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2247–2255. PMLR.

Maclaurin, D., Duvenaud, D., and Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. In Bach, F. R. and Blei, D. M., editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 2113–2122. JMLR.org.

Metz, L., Harrison, J., Freeman, C. D., Merchant, A., Beyer, L., Bradbury, J., Agrawal, N., Poole, B., Mordatch, I., Roberts, A., et al. (2022). Velo: Training versatile learned optimizers by scaling up. *arXiv preprint arXiv:2211.09760*.

Metz, L., Maheswaranathan, N., Freeman, C. D., Poole, B., and Sohl-Dickstein, J. (2020a). Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves. *CoRR*, abs/2009.11243.

Metz, L., Maheswaranathan, N., Nixon, J., Freeman, C. D., and Sohl-Dickstein, J. (2019). Understanding and correcting pathologies in the training of learned optimizers. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 4556–4565. PMLR.

Metz, L., Maheswaranathan, N., Sun, R., Freeman, C. D., Poole, B., and Sohl-Dickstein, J. (2020b). Using a thousand optimization tasks to learn hyperparameter search strategies. *CoRR*, abs/2002.11887.

Mozer, M. (1995). A focused backpropagation algorithm for temporal pattern recognition. *Complex Systems*, 3.

Peer, D., Keulen, B., Stabinger, S., Piater, J., and Rodriguez-sanchez, A. (2022). Improving the trainability of deep neural networks through layerwise batch-entropy regularization. *Transactions on Machine Learning Research*.

Randazzo, E., Niklasson, E., and Mordvintsev, A. (2020). MPLP: learning a message passing learning protocol. *CoRR*, abs/2007.00970.

Runarsson, T. and Jonsson, M. (2000). Evolution and design of distributed learning rules. In *2000 IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks. Proceedings of the First IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks (Cat. No.00*, pages 59–63.

Sandler, M., Vladymyrov, M., Zhmoginov, A., Miller, N., Madams, T., Jackson, A., and y Arcas, B. A. (2021). Meta-learning bidirectional update rules. In Meila, M. and Zhang, T., editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 9288–9300. PMLR.

Schmidhuber, J. (1987). Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Diploma thesis, Technische Universitat Munchen, Germany.

Schraudolph, N. (1999). Local gain adaptation in stochastic gradient descent. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 569–574 vol.2.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Vicol, P., Metz, L., and Sohl-Dickstein, J. (2021). Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In Meila, M. and Zhang, T., editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 10553–10563. PMLR.

von Oswald, J., Niklasson, E., Randazzo, E., Sacramento, J., Mordvintsev, A., Zhmoginov, A., and Vladymyrov, M. (2022). Transformers learn in-context by gradient descent. *arXiv preprint arXiv:2212.07677*.

Wichrowska, O., Maheswaranathan, N., Hoffman, M. W., Colmenarejo, S. G., Denil, M., de Freitas, N., and Sohl-Dickstein, J. (2017). Learned optimizers that scale and generalize. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3751–3760. PMLR.

Williams, R. J. and Zipser, D. (1989). Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1:87–111.

Wu, Y., Ren, M., Liao, R., and Grosse, R. B. (2018). Understanding short-horizon bias in stochastic meta-optimization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Appendices

A Finding the message passing functions for SGD

In this section we find the functions from Eq. 4 and 5 such that the backwards pass mimics ordinary SGD without momentum. Finding these functions will show us that this framework has at least the capacity to imitate SGD without momentum.

Ordinary stochastic gradient descent without momentum is defined as follows,

$$\theta_c^{t+1} \leftarrow \theta_c^t - \alpha \Delta\theta_c \tag{9}$$

$$\Delta\theta_c = \frac{\partial L}{\partial \theta_c} \tag{10}$$

Where c is the coordinate of a linear node. θ_c is then the weight/parameter at node c. L is the loss and α is the learning rate which is constant. We assume no batches are used for now. This is done to simplify the derivation. It can be easily shown that batching is essentially taking the average of the $\Delta\theta_c$ over the batches as the final weight update.

We start with finding the function $f_{lin}(..., m, c)$. The purpose of $f_{lin}(..., m, c)$ is to calculate the weight update $\alpha \cdot \Delta\theta_c$. We find that for a linear node i

$$f_{lin}(..., m, i) = \alpha \frac{\partial L}{\partial \theta_i} \tag{11}$$

We can rewrite this to

$$f_{lin}(..., m, i) = \alpha \frac{\partial L}{\partial k_i(x_i, \theta_i)} \frac{\partial k_i(x_i, \theta_i)}{\partial \theta_i} \tag{12}$$

Where $k_i(...)$ is the computation that corresponds to the node i and can be found in Table 1. In this case, because $f_{lin}(...)$ is always calculated at a linear node $k_i(x_i, \theta_i) = x_i \cdot \theta_i$.

This gives us

$$\frac{\partial k_i(x_i, \theta_i)}{\partial \theta_i} = x \tag{13}$$

We then assume the following

$$\frac{\partial L}{\partial k_i(x_i, \theta_i)} = m \tag{14}$$

We can then rewrite Eq. 12

$$f_{lin}(x, m, c) = \alpha \cdot x \cdot m \tag{15}$$

Where x_i is the input to the i node at which f_{lin} is calculated.

We have now found the function f_{lin} such that it mimics ordinary SGD without momentum. However, the found function is dependent on the message m that is received for it to work.

We call the node that generates this message node j. The node j can be of any type. We found in Eq. 14 that the function g that generates this message must be the following

$$g_{nodetype}(..., m, j) = \frac{\partial L}{\partial k_i(x, \theta_i)} \tag{16}$$

Again, j is the node that generates and passes a message backwards to node i.

We can rewrite this to

$$g_{nodetype}(..., m, j) = \frac{\partial L}{\partial k_j(..., x_j)} \frac{\partial k_j(..., x_j)}{\partial k_i(x_i, \theta_i)} \tag{17}$$

Because the output of node i is the input to node j,

$$\frac{\partial k_j(..., x_j)}{\partial k_i(x_i, \theta_i)} = \frac{\partial k_j(..., x_j)}{\partial x_j} \tag{18}$$

We also assume,

$$\frac{\partial L}{\partial k_j(..., x_j)} = m \tag{19}$$

We can then rewrite Eq. 17

$$g_{nodetype}(..., m, j) = \frac{\partial k_j(..., x_j)}{\partial x_j} \cdot m \tag{20}$$

We have now also found the mssage passing generation functions g. Again we see that $g_{nodetype}(..., m, j)$ is dependent on the message that is generated from the node after j. Because the message is always of the same form (see Eq. 14 and 19) we get this recursive behaviour.

The recursion ends at the loss node. Where,

$$g_{loss}(..., j) = \frac{\partial L}{\partial k_i(x_i, \theta_i)} = \frac{\partial k_j(x_j, y)}{\partial x_j} \tag{21}$$

B Summing more closely resembles gradients

Whenever a node within a computational graph receives multiple messages as input we need to decide how we are going to aggregate the messages. There are multiple options to do this e.g. summing, averaging, maximizing etc. We have chosen to sum the messages because it more closely resembles how gradients would normally be calculated.

A node receives multiple messages when it has multiple occurrences in the mathematical representation of the computational graph. This means that a single node influences multiple

other nodes in the computational graph. In mathematical notation, this can be written down as follows,

$$\frac{\partial L}{\partial z} L(p(z), q(z))$$

Where z is the node that receives multiple messages. $p()$ and $q()$ are operations that are connected to z. And L is the function we ultimately want to minimize.

We can rewrite this to:

$$\frac{\partial L}{\partial z} L(f(z), g(z)) = \frac{\partial L}{\partial g}\frac{\partial g}{\partial z} + \frac{\partial L}{\partial f}\frac{\partial f}{\partial z}$$

Which is the sum of the gradients of the two functions that are connected to z. This justifies the use of summing as the aggregation method.

C Cross-entropy message generator function to mimic SGD

The softmax activation function is a normalization function and is defined as follows,

$$\sigma(x, i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{22}$$

Where x is the output vector of the final hidden layer, e.g. $x = x_{prev} \cdot W + b_{prev}$. Note that because of the denominator, $\sigma(x, i)$ is dependent on all the elements in x. This is different from e.g., Relu or Sigmoid where there is only a dependence on a single element of x: x_i.

For message passing, this is a problem because in our current implementation, there are no cross connections between the pre-softmax layer and the softmax. However, if we combine the cross-entropy loss with the softmax activation function this cross-dependence disappears. The cross-entropy for a single loss node is defined as follows:

$$L(y, x) = -\sum_{i=1}^{n} y_i \log(\sigma(x, i)) \tag{23}$$

We are now interested in how a single pre-softmax output node x_i influences the loss. It turns out the cross-dependence dissapears[1]:

$$\frac{\partial L}{\partial x_i} = -\sum_{j=1}^{n} y_j \frac{\partial \log(\sigma(x, j))}{\partial x_i} = \sigma(x, i) - y_i \tag{24}$$

This means that the message passing function for the softmax and the cross-entropy combined can be written as follows:

$$g_{crossentropy}(x, y_i, m) = \sigma(x, i) - y_i \tag{25}$$

[1] An example of how to obtain this derivation can be found here: `https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1`

Where i is the index of one of the output nodes of the MLP. This means we don't need the cross-connections between the pre-softmax layer and the softmax layer to be able to mimic SGD. Therefore, it is also a logical decision to combine the softmax and the cross-entropy into a single node.

D Learned ReLU anti-derivative

In the experiment from Section 5.1 we noted that the results from Figure 7 are interesting in the sense that they do not seem to correspond to the ReLU activation function.

From 2 we find that,

$$g_{act} = act'(x) \cdot m \tag{26}$$

We can therefore divide by m to obtain the derivative of the activation function. We do this both for the learned g_{relu} and the SGD imitation.

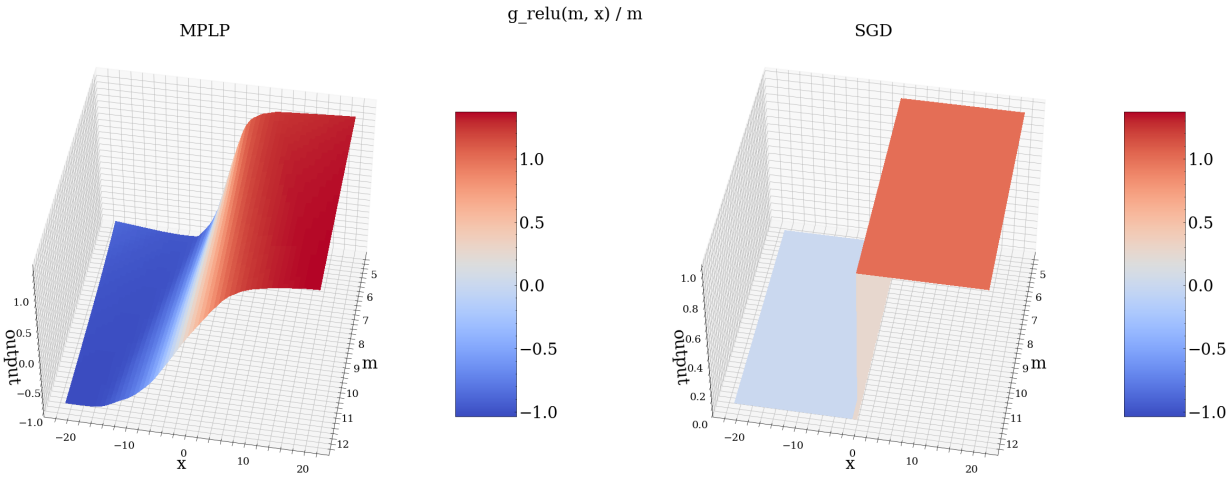


Figure 12: Same as Figure 7 but we divide by m. The result is supposedly the derivative of its activation.

From Figure 12 we see that MPLP function has learned to multiply by the incoming message/"gradient". I.e. after the division by m, m no longer affects the outcome. Note, that we cut off the values for $m \in [0, 5]$ to remove a chaotic area.

We are now left with the derivative of the activation. If we take the anti-derivative of this function we can see which activation function it corresponds to.

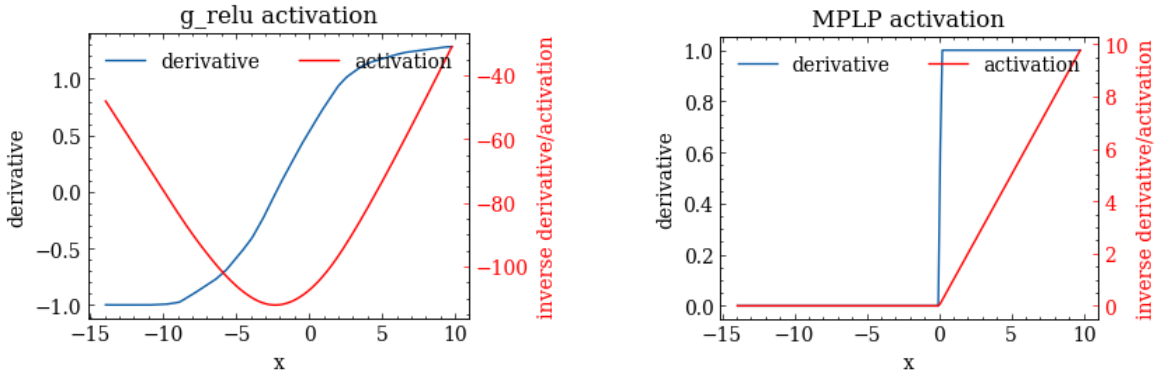


Figure 13: The activation functions corresponding to the message generating functions. g_{relu} does not correspond to ReLU.

We find that the activation function that corresponds to learned message generating function is very different than the ReLU that is used in the forward pass.