Albert-Ludwigs University of Freiburg

Master Thesis

Learning to Optimize Deep Neural Networks

Author:
Muneeb Shahid

Supervisor:
Dr. Ilya Loshchilov

1st Examiner:
Dr. Frank Hutter

2nd Examiner:
Dr. Joschka Boedecker

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science
in the
Research Group Machine Learning for
Automated Algorithm Design,
Faculty of Engineering,
Department of Computer Science

Submitted on 25th October 2017

DECLARATION

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg,	
Place, Date	Signature

I would like to thank my family, my parents in particular, who always supported me when I needed them. And many thanks to my supervisor Ilya, his insights were of tremendous help during my thesis. Lastly, I am thankful to my floor mates who have been of incredible support throughout, especially towards the end of my thesis.

Abstract

With the advent of deep learning, machine learning community has started to move from handcrafted features to learned ones. These learned features often outperform manually crated features. Learning to optimize follows in the same footsteps and tries to learn an optimizer. In this thesis we present two approaches in the same direction, "Learning to Optimize With Normalized Inputs" and "Multiscale Adam". In both approaches we train meta optimizers such that given gradient information as input they output a gradient step. The first approach, "Learning to Optimize With Normalized Inputs" uses normalized history of gradients as inputs. We explore various configurations of this approach e.g. effect of using different history sizes. We also compare the learned optimizer with *Adam* and *RMSProp*. Our trained optimizer outperforms both *Adam* and *RMSProp* on the neural network it was trained on. Furthermore when applied on an unseen neural network (*Cifar10*), the learned optimizer shows competitive performance displaying its ability to generalize. The other approach, "Multiscale Adam" uses *Adam* running at different timescales as inputs and outputs a gradient step that is a weighted average of its *Adam* inputs. We then compare the performance of the learned optimizer with its individual *Adam* inputs. We do thorough testing of our approach on a small neural network on Mnist, we see that in most cases our trained optimizer outperforms its *Adam* inputs. Our trained optimizer also generalizes to other unseen networks, e.g. *Cifar10*, with success. In both approaches we start from simple models, test their capabilities, gradually make them more complex and report the results. Finally, we also discuss the limitations of both approaches as well.

Zusammenfassung

Mit dem Aufkommen von Deep Learning hat die Machine Learning Gemeinschaft begonnen, gelernte statt manuell gewählten Features zu verwenden. Solche gelernten Features erreichen zudem oft eine bessere Performance. Der Ansatz des Learning to optimize" verfolgt diese Idee und versucht einen Optimierer zu lernen. In dieser Arbeit verfolgen wir zwei Ansätze aus dieser Richtung,"Learning to Optimize With Normalized Inputs„'Multiscale Adam". In beiden Ansätzen trainieren wir Meta-Optimierer, welche als Eingabe Gradienten-Information erhalten und einen Gradienten-Update-Schritt ausgeben. Der erste Ansatz "Learning to Optimize With Normalized Inputs" benutzt die normalisierten vorhergegangenen Gradientenwerte als Eingabe. Wir untersuchen verschiedene Konfigurationen dieses Ansatzes, insbesondere betrachten wir wie sich die Anzahl betrachteter Schritte auf die Performance des Ansatzes auswirkt. Ferner vergleichen wir den gelerten Optimierer mit dem *Adam* und *RMSProp* Optimierer. Die Performance unseres gelernten Optimierers ist auf dem Netzwerk, auf welchem er trainiert wurde besser als die des *Adam* Optimierers. Auf einem ungesehenen Netzwerk erreicht unser gelernter Optimierer eine zum *Adam* Optimierer vergleichbare Performance. Dies demonstriert die Fähigkeit unseres Optimierers gelerntes Wissen zu verallgemeinern. Der andere Ansatz, "Multiscale Adamërhält die Ausgabe von Adam konfiguriert mit verschiedenen Zeitskalierungen als Eingabe und gibt einen Gradienten-Update-Schritt aus, der ein gewichteter Mittelwert seiner Eingaben ist. Wir vergleichen die Performance dieses gelernten Optimierers mit seinen individuellen Adam Eingaben. Wir führen ausführliche Tests unseres Ansatzes auf einem kleinen neuronalen Netz (Mnist) durch, wobei unser gelernter Optimierer besser abschneidet als seine Adam Eingaben. Unser gelernter Optimierer generalisiert auch auf andere ungesehene neuronale Netzwerke (Cifar10) mit Erfolg. Bei beiden Ansätzen beginnen wir mit einfachen Modellen, quantifizieren ihre Fähigkeiten und machen sie graduell immer komplexer und geben die Ergebnisse an. Zum Schluss diskutieren wir auch die Einschränkungen unserer beiden Ansätze.

Contents

Chapter 1

Introduction

Coordinatewise handcrafted optimizers such as gradient descent, Adam [6], RMSprop [14] form the backbone of optimizing neural networks. Given a problem $\mathbf{P}(x^t)$ that is a function of problem variables $x \in \mathbb{R}^{n\times 1}$, $\nabla_x L^t$ [1] $\in \mathbb{R}^{n\times 1}$: the gradient vector of problem loss $L^t \in \mathbb{R}^{1\times 1}$ with respect to the problem variables x^t a general update step is given by:

$$x_i^{t+1} = x_i^t - \delta x_i^t.$$

The computation of the step δx_i^t differs between methods. Due to high dimensional nature of deep networks; 2nd order methods such as Newton's method, BFGS etc are not considered viable and are hence excluded from further discussions.

1.1 Handcrafted Optimizers

Stochastic Gradient Descent (SGD) employs the following update step:

$$\delta x_i^t = \eta \cdot \nabla_x L_i^t.$$

where η is the learning rate.

SGD with momentum A simple SGD implementation is extremely vulnerable to noise and has trouble getting out of local minimas. Combining it with momentum helps it get over this weakness. The update step is given by:

$$\delta x_i^t = \gamma \cdot \delta x_i^{t-1} + \eta \cdot \nabla_x L_i^t$$

[1] Each row i of $\nabla_x L^t$: $\nabla_x L_i^t$ consists of $\frac{\partial L^t}{\partial x_i^t}$.

Nesterov Accelrated Gradient (NAG) When using normal momentum we compute gradients at x_i^t, but we know that the momentum term is going to shift the variables , NAG [9] caters for this by by computing gradients at the shifted position. The update step for NAG is given by:

$$\delta x_i^t = \gamma \cdot \delta x_i^{t-1} + \frac{\partial L}{\partial (x_i^t - \gamma \cdot \delta x_i^{t-1})}$$

Adagrad Unlike previous methods, Adagrad [4] is an adaptive gradient descent method. It adapts the learning rate to the problem variables i.e it has a different learning rate for each parameter. This allows it to perform large updates for infrequent parameters while performing small updates for frequent parameters. Its update step is given by:

$$\delta x_i^t = \frac{\eta}{\sqrt{G_{ii}^t + \epsilon}} \cdot \nabla_x L_i^t$$

where $G \in \mathbb{R}^{n \times n}$ is a diagonal matrix such that each entry G_{ii}^t is the sum of squares of gradients $\frac{\partial L^t}{\partial x_i^t}$, and ϵ is a stabilization constant. However this matrix is also Adagrad's biggest flaw, as it keeps accumulating sums in the denominator eventually causing the training to come to a halt.

RMSprop RMSprop [14] remedies the aforementioned pitfall of Adagrad by keeping a moving average of squared gradients instead. The update step for RMSprop is given by:

$$\begin{aligned} v_i^t &= \beta \cdot v_i^{t-1} + (1 - \beta) \cdot (\nabla_x L_i^t)^2 \\ \delta x_i^t &= \frac{\eta}{\sqrt{v_i^t + \epsilon}} \cdot \nabla_x L_i^t \end{aligned}$$

where β is the momentum timescale.

Adam Like gradient descent with momentum and Rmsprop, Adam [6] employs moving averages of the gradients and squared gradients respectively. As at the start the moving averages are biased towards zero, the authors perform bias correction. The update step for Adam is given by:

$$
\begin{aligned}
m_i^t &= \beta_1^t \cdot m_i^{t-1} + (1 - \beta_1^t) \cdot \nabla_x L_i^t \\
v_i^t &= \beta_2^t \cdot v_i^{t-1} + (1 - \beta_2^t) \cdot (\nabla_x L_i^t)^2 \\
\hat{m}_i^t &= \frac{1}{1 - \beta_1^t} \\
\hat{v}_i^t &= \frac{1}{1 - \beta_2^t} \\
\delta x_i^t &= \frac{\eta}{\sqrt{\hat{v}_i^t + \epsilon}} \cdot \hat{m}_i^t
\end{aligned}
$$

where β_1 and β_2 are momentum timescales.

Adamax a variation of Adam uses the l_∞ norm to update v_i^t instead of the l_2 norm, the update equation for v_i^t is given by:

$$v_i^t = \mathbf{max}(\beta_2^t \cdot v_i^{t-1}, |\nabla_x L_i^t|)$$

Notice that due to l_∞ norm, there is no need of bias correction.

1.2 Learned Optimizers

Learning to learn while still not commonly applied has a long history [13]. Schmidhuber [11] developed a network that was able to modify its own weights. Recently Li and Malik approached learning to learn from a reinforcement learning perspective. They learnt an optimization algorithm using guided policy search. They show that for the convex and non convex problems they trained for, the learnt optimizer outperforms manual optimizers. Sinha et al. trained an introspection network on training history of a *Mnist* network. The introspection network when applied to other networks predicted their weights after a certain number of iterations. Using the introspection network they were able to accelerate training of other networks i.e. *Cifar10*.

Learning to Learn Closer to our approach is the work done in *learning to learn* series. Andrychowicz et al. were the first ones, they trained a 2 layered LSTM [5] to optimize for different problems. They also demonstrated its performance on small *Mnist* and *Cifar10* networks. The learnt optimizer could only optimize for a specific number of iterations, after which it failed to perform well, change in activation functions also caused the learnt optimizer to diverge. They also had to train different meta optimizers for convolution and fully connected layers with in the same problem network (*Cifar10*). We believe a major reason for these issues is unnormalized inputs. As the scale of gradient inputs changes when they switch

problem networks or network activations, their learned optimizer fails to perform well, if at all, i.e. when sigmoid activations were changed to relu in an mnist network, the metaoptimizer diverged from the very start. We on the other hand use normalized inputs which makes our algorithms tolerant to change in gradient scale; enabling our trained models to generalize with much greater success. It should also be noted that compared to their we approach we use a much simpler architecture e.g a mlp.

Wichrowska et al. published their work on learning to learn as the next iteration. They introduced a novel hierarchical *RNN* architecture. They used *GRU*[3] as the building block of their model. Layers at different levels in hierarchy had a different focus. The lowest layer received information from the parameters of the problem network i.e *Mnist*, the middle layer focused on layer level information i.e. hidden layers in the problem network. Whereas as the top most layer had access to the whole problem level information. Similar to our work they also used normalized gradient inputs, however they used *Adam* like normalization, that is the moving averages of their inputs were normalized with moving averages of squared gradients. We on the other hand use a different normalization scheme, explained later on. Moreover they also use change in gradient magnitudes along with relative learning rate of parameters as inputs for their meta optimizer. They also learned the hyper parameters, such as learning rate, on the fly. They used small toy problems with added noise as training grounds. They report that after training on this corpus of small problems their optimizer generalized well to large neural networks. However for relatively larger networks i.e. Resnet V2, it failed to perform well. Contrary to them our architectures are much simpler, which was one of the goals for us as well, i.e. study how simple architectures perform and see if we find any motivation to use more complex ones. Finally an empirical evaluation of their approach could not be made possible as they released their code and the problems they trained their meta optimizer on, when it was already too late for us.

Chapter 2

Learning to Optimize Deep Neural Networks

In this chapter we present an overview of our approaches as a precursor to the details. Our work has two related directions, [1] we will present the approaches in the order they were originally attempted. We will start with simple models, test their limits and report the results. These results then provide the motivation for the next step we take. We refer to these two directions as:

- Learning to Optimize With Normalized Inputs.
- Multiscale Adam.

Both of these approaches use normalized inputs. Normalization helps meta optimizer in learning as it makes it impervious to the fluctuating gradient scale throughout the optimization. Prior to delineating the two approaches we will first describe the underlying learning algorithm that forms the base of both of these approaches.

2.1 The Learning Algorithm

In this section we present an overview of the underlying learning algorithm. While the aforementioned approaches differ in a number of ways, the underlying learning algorithm remains the same. The learning algorithm comprises of the following blocks:

- A handcrafted optimizer, $\mathbf{O}$. i.e Adam, RmsProp.

[1] While we tried to keep progress in both approaches as in sync as possible, one might notice some inconsistencies i.e some techniques that helped us in one of the approaches not being applied in the other approach, this is purely due to lack of enough time window.

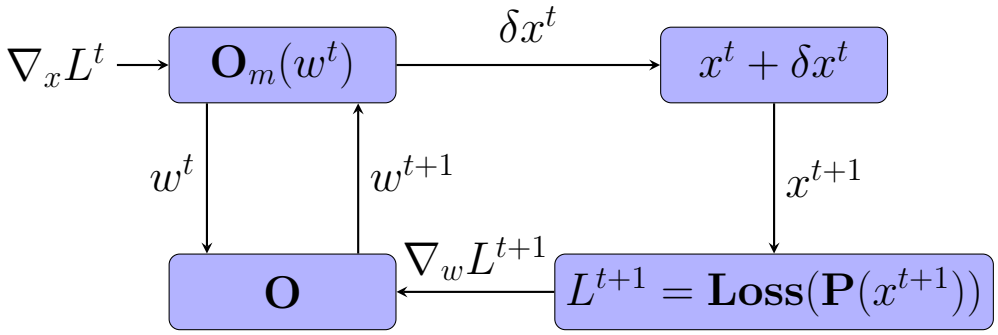


Figure 2.1: *The Learning Algorithm.*

- A meta optimizer, $\mathbf{O}_m(w^t, \nabla_x L^t)$. i.e a neural network with weights w^t that takes gradients $\nabla_x L^t$ as input.
- A training problem $\mathbf{P}(x^t)$. i.e *Rosenbrock, Mnist.*

Given a problem $\mathbf{P}(x^t)$ and problem variables $x_i^t \in \mathbb{R}^{n\times 1}$, at time step t we calculate $\nabla_x L^t \in \mathbb{R}^{n\times 1}$: the gradient vector of problem loss L^t with respect to the problem variables x^t. Each row i of $\nabla_x L^t$, $\nabla_x L_i^t$, consists of $\frac{\partial L^t}{\partial x_i^t}$. After normalization $\nabla_x L^t$ is fed as inputs to the meta optimizer. The meta optimizer then outputs a step δx^t. This step is applied to the the problem variables x^t to obtain x^{t+1}. We then calculate L^{t+1}. Finally, $\mathbf{O}$ utilizes $\nabla_w L^{t+1}$, the gradients of the loss L^{t+1} with respect to the meta optimizer parameters w^t, to output w^{t+1}. Figure 2.1 illustrates the flow chart.

2.2 Training Regime

For our experiments we implement everything in Tensorflow [1]. We tested between *Adam* and *RMSProp* to select the optimizer $\mathbf{O}$ and noticed *Adam* to give better results and thus throughout we use *Adam* as $\mathbf{O}$. Optimizer $\mathbf{O}$, trains the meta optimizer $\mathbf{O}_m$, on a specific problem $\mathbf{P}(x^t)$, for a total of $train_{total}$ meta iterations. Since at the start $\mathbf{O}_m$ is untrained and primarily outputs noise, it is fairly common for it to push the problem $\mathbf{P}(x^t)$, that it is training to optimize, in to a minima from where it is incredibly hard to recover, if not impossible. Thus after every few random number of meta iterations $train_{reset}$ we reset the problem to a random starting position. After $\mathbf{O}_m$ has learned something useful, the minimum and maximum number of $train_{reset}$ is increased. Reinitializing the problem randomly every so often essentially translates to training on multiple problems. We make training dumps after every $train_{dump}$ iterations and validate performance of $\mathbf{O}_m$ on $\mathbf{P}_{val}(x^t)$. Note that during validation of $\mathbf{O}_m$, $\mathbf{O}$ does not

train $\mathbf{O}_m$. Finally we select the meta optimizer with best validation performance for further experiments and report the results.

2.3 Optimization Problems

In this section we list all the problems that we experimented with. Later on we will refer to the these problem when needed.

2.3.1 Sum of Squares

We consider a 1000 dimensional sum of squares. The variables were initialized from a uniform distribution ranging from -10, 10. Problem loss for the problem is given by the following equation:

$$L = \sum_{i}^{n} x_i^2 \tag{2.1}$$

2.3.2 Rosenbrock

We used the classic two dimensional *Rosenbrock* function [10] also known as *Rosenbrock's valley* or *Rosenbrock's banana function*. The global minima lies with in a banana shaped, long and narrow valley. While the valley can be found with ease, converging to the global minima is much harder. The loss is given by:

$$L = (a - x)^2 + b \cdot (y - x)^2 \tag{2.2}$$

where $a = 1$ and $b = 100$.

2.3.3 Mnist

Mnist is a collection of small images of handwritten digits. Each image is of dimension 28×28. The dataset consists of 55000 training data points, 10000 test data points and 5000 validation data points. For *Mnist* we consider 3 different architectures given as follows:

Mlp $MlpMnsit$ The *mlp* consists of a single hidden layer constituting of only 20 neurons with *sigmoid* activations. The final layer has 10 neurons corresponding to the 10 classes. The network computes *cross entropy* loss over the output. The network has a total of 15910 trainable parameters. We will refer to this network as $MlpMnsit$. This is the default network for *Mnist* that we will use in the bulk of our experiments, the reason being that its small size expedites the process of experimentation.

Convnet $ConvMnsit_l$ Next we consider a relatively larger convnet $ConvMnsit_l$. This is the default network available in Tensorflow for *Mnist*. It has two *convolutional* layers with *relu* activations of dimension [5, 5, 1, 32] and [5, 5, 32 ,64] respectively. Both layers are followed by a pooling layer (2x2 pool size and stride). Then comes a fully connected layer of dimensions [3136, 1024] with *relu* activations, this layer is connected to the output layer. The network then computes *cross entropy* loss over the final layer.

Convnet $ConvMnsit_s$ also has two *convolutional* layers with *relu* activations, each followed by a *max pool* layer (2x2 pool size and stride), their dimensions are [5, 5, 1, 16] and [5, 5, 1, 32]. Then comes a hidden layer of dimensions [1568, 512]. And finally the output layer computes *cross entropy* loss.

2.3.4 Cifar10

Cifar10 consists of a total of 60000 colored images of dimension 32x32. There are a total of 10 classes. The training dataset consists of 50000 images while the rest constitute the test set. We consider two closely related convents for *Cifar10*:

$ConvCifar_0$ is the default Cifar10 network available in Tensorflow it has 1,068,298 trainable parameters. The architecture is as follows: two convolution layers of dimensions $[5 \times 5 \times 3 \times 64]$ and $[5 \times 5 \times 64 \times 64]$. Both layers are normalized via LRN and *max pooled* (kernel 3×3, stride 2×2). The two fully connected layers have dimensions 2304×384 and 384×191. Finally *cross entropy* loss is computed on the output layer. The network uses *relu* activations through out. Some layers also employ weight decay.

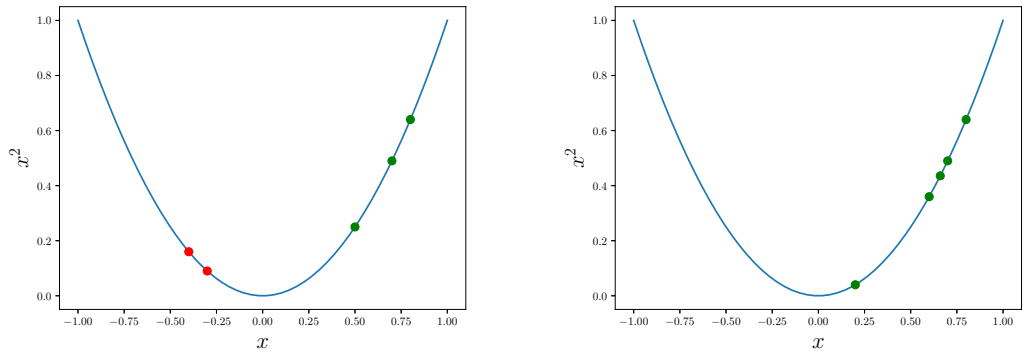
$ConvCifar_1$ has the same structure as $ConvCifar_0$ i.e *max pooling* and normalization, except for smaller number of trainable parameters. The two *convolutional* layers are given by $[5 \times 5 \times 3 \times 32]$ and $[5 \times 5 \times 32 \times 32]$. While the fully connected layers are $[576 \times 96]$ and $[96 \times 46]$. Like $ConvCifar_0$, $ConvCifar_1$ also uses relu activations and weight decay.

Chapter 3

Learning to Optimize With Normalized Inputs

3.1 Motivation

Consider the scenario that, while traversing some noise free optimization landscape our gradient signs change, this a good indicator we have just crossed a minima fig 3.1a illustrates the scenario. On the other hand while making our progress towards the minima if the signs stay the same then it is a good idea to keep going in the same direction fig 3.1b. These heuristics form the motivation of "Learning to Optimize With Normalized Inputs". That is we consider a history of gradients to output the gradient step δx_i^t.



(a) *Gradient signs changed after we crossed the minima.*

(b) *Gradient signs stay the same, keep going in the same direction.*

Figure 3.1: *Green circles denote points with positive gradient while red ones denote points with negative gradient.*

3.2 Algorithm

In order to better explain the algoritm we will now formalize our motivation. Consider a vector $\mathrm{H}^t_{x,i} \in \mathbb{R}^{1\times k}$ at time step t containing history of problem variable x_i over the last k time steps such that the gradients at these steps have mixed signs, then perhaps the local optima x^* lies somewhere between x_i^{max} [1] and x_i^{min} [2]. Figure 3.2a illustrates this scenario. On the other hand if all the gradients point in the same direction then it is a good indicator that x^* lies somewhere outside the history of k steps, Fig 3.2b, i.e $x^* < x^{max}$ or $x^* > x^{min}$. We study two variations of processing our history of gradients, namely "k Timesteps" that relies on last k timesteps and "k Momentums" that utilizes k momentums at different timescales. Both approaches maintain two history matrices

- $\mathrm{H}^t_x \in \mathbb{R}^{n\times k}$: History of problem variables $x^t \in \mathbb{R}^{n\times 1}$.
- $\mathrm{H}^t_{\nabla L} \in \mathbb{R}^{n\times k}$: Gradient history $\nabla_x L^t \in \mathbb{R}^{n\times 1}$ of the problem variables.

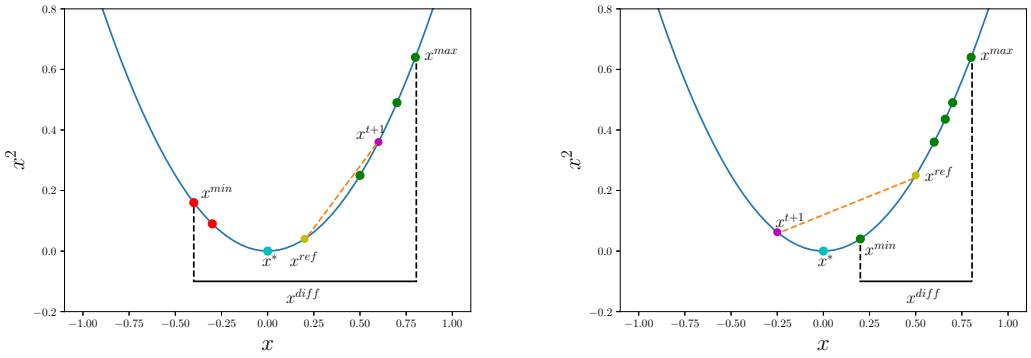
$\mathrm{H}^t_{\nabla L}$ is normalized coordinatewise and fed as inputs to the $\mathbf{O}_m$. The motivation here is that $\mathbf{O}_m$ should learn from the history of inputs and output an appropriate step $\delta x_i^t \in [-1.0, 1.0]$. Unless stated otherwise in our experiments the inputs are normalized by the infinity norm i.e whole row $\mathrm{H}^t_{\nabla L,i} \in \mathbb{R}^{1\times k}$, is normalized by the maximum element in the row $\mathrm{H}^t_{\nabla L,i}$. The update equations we use to calculate x_i^{t+1} are given as:

$$x_i^{t,ref} = (x_i^{t,max} + x_i^{t,min})/2 \tag{3.1}$$
$$x_i^{t,diff} = x_i^{t,max} - x_i^{t,min} \tag{3.2}$$
$$x_i^{t+1} = x_i^{t,ref} + \delta x_i^t \cdot x_i^{t,diff} \tag{3.3}$$

Where $x_i^{t,max} = \mathbf{max}(\mathrm{H}^t_{x,i}) \in \mathbb{R}^{1\times 1}$ and $x_i^{t,min} = \mathbf{min}(\mathrm{H}^t_{x,i}) \in \mathbb{R}^{1\times 1}$. $x_i^{t,ref}$ acts as a reference point, we then decide how away far we want to shoot from the reference point. We set $x_i^{t,ref}$ to be the mid point in our history as we suspected it to be a safe position to shoot from, to get to x_i^{t+1}. The computed step δx_i^t determines our shooting strength. But since it's an output to normalized inputs we need to scale it back to x_i^t space and thus it is then multiplied with $x_i^{t,diff}$. Figure 3.2 visualizes the update equations. Figure 3.2a demonstrates the case when we shoot with in our history range, where as figure 3.2b demonstrates the case when we need to look outside our history.

Note that x_i^{diff} also acts a variable learning rate for δx_i^t. To see how, consider $\mathrm{H}^t_{\nabla L,i}$ such that it has about the same magnitude of positive and negative gradients then the step δx_i^t should be almost zero and then the update will be $x_i^{t+1} \approx x_i^{ref}$ i.e x^* lies somewhere middle of our history record or close to the

[1] $x_i^{max} = \mathbf{max}(\mathrm{H}^t_{x,i}) \in \mathbb{R}^{1\times 1}$
[2] $x_i^{min} = \mathbf{min}(\mathrm{H}^t_{x,i}) \in \mathbb{R}^{1\times 1}$



(a) *The minima x^* lies within the range of observation history.*

(b) *The minima lies outside the range of observation history.*

Figure 3.2: *Illustration of how our algorithm computes x^{t+1}. Here we have $k = 5$. Green and red circles are the problem variables stored in $\mathrm{H}^t_{x,i}$. Green ones denote variables with positive gradient direction, while red ones denote problem variables with negative gradient direction. Our algorithm first determines the reference point x^{ref}, afterwards depending upon the gradient magnitudes and direction the algorithm outputs δx^t which we then use to move from x^{ref} to x^{t+1}.*

reference point (Fig 3.2a). This entails that the term x_i^{diff} will either contract and become smaller or stay the same, it will not grow.

On the other hand the more gradients point in a particular direction, the bigger the step δx_i^t magnitude and the more power we use to shoot in the opposite direction. With enough gradients in particular direction we shoot outside the collected history of last k steps (Fig 3.2b) i.e $x_i^{t+1} > x_i^{max}$ or $x_i^{t+1} < x_i^{min}$, this implies that the term x_i^{diff} would expand or stay the same.

Thus we have this contraction and expansion of the term x_i^{diff} which can be viewed as increasing and deceasing the learning rate of δx_i^t.

3.3 Architecture

We use a single layered *Mlp* as the meta optimizer $\mathbf{O}_m$. The hidden layer consists of 50 neurons with *relu* activations. We treat computing the step δx_i^t as a classification problem. The output layer constitutes of twelve neurons. Two of neurons are allocated for the sign of the step i.e $\{+1.0, -1.0\}$. Whereas, each of the 10 neurons correspond to ten linearly spaced δx_i^t magnitudes between 0 and 1 i.e $\{0.0, 0.111 \cdots, \cdots, 1.0\}$. Using *softmax* we compute expectation over the neurons for sign and magnitudes separately. Finally δx_i^t is the product of the two expectations. We also experimented with using *tanh* instead, but this approach was more stable and gave better results.

3.4 k Timesteps

For "k Timesteps" the history matrices contain the last k timesteps of x and $\nabla_x L$; that is at time step t, $\mathrm{H}^t_{x,i}$ contains last k values of x i.e $\mathrm{H}^t_{x,i} = \{x_i^t, x_i^{t-1}, \ldots, x_i^{t-k-1}\}$. Similarly $\mathrm{H}^t_{\nabla L,i} = \{\nabla_x L_i^t, \nabla_x L_i^{t-1}, \ldots, \nabla_x L_i^{t-k-1}\}$. Before the algorithm starts execution both of the history matrices are populated with x^0 and $\nabla_x L^0$ accordingly i.e $\mathrm{H}^t_{x,i} = \{x_i^t, x_i^t, \cdots, x_i^t\}$ and $\mathrm{H}^t_{\nabla L,i} = \{\nabla_x L_i^t, \nabla_x L_i^t, \ldots, \nabla_x L_i^t\}$. Note that this poses a serious problem as this entails that the term x_i^{diff} is now zero, and as a consequence our algorithm will not make any progress. Additionally, noise can also cause x_i^{diff} to become very small which causes the convergence to slow down. Thus we add minimum learning rate η_{min} as an additional term. Update equation (3.3) now becomes

$$x_i^{t+1} = x_i^{t,ref} + \delta x_i^t \cdot (x_i^{t,diff} + \eta_{min}) \tag{3.4}$$

Both the history matrices are implemented as queues, when updating them we pop the record at the oldest timestep i.e. the record at the most distant past. And then we push a new record. Before moving towards *learning to optimize*, we present a very closely related handcrafted algorithm. We created and experimented with this algorithm to get an idea of how much potential there is for meta learning using update equation (3.4).

3.4.1 Mean Normalized Gradient (*MNG*)

MNG is a handcrafted optimizer, it computes coordinatewise mean of normalized gradients and uses the computed mean as the δx_i^t.

Algorithm

The algorithm starts by populating the history matrices with values at timestep 0, with the function **populate**(). It then normalizes $\mathrm{H}^t_{\nabla L}$ coordinatewise. Using the normalized $\mathrm{H}^t_{\nabla L}$ it calculates mean normalized gradients. Afterwards using the equation (3.4) the algorithm applies the calculated mean normalized gradients as the step δx^t to output x^{t+1}. Algorithm 1 shows the steps in detail.

Algorithm 1: Mean Normalized Gradient

Input: $\mathbf{P}(x^t)$, $x^t \in \mathbb{R}^{n\times 1}$ ▷ `Problem and its variables`

$L^t \leftarrow \mathbf{loss}(\mathbf{P}(x^t))$

$\nabla_x L^t \leftarrow \mathbf{BackProp}(L^t, x^t)$ ▷ $\nabla_x L^t \in \mathbb{R}^{n\times 1}$

$\mathrm{H}_x^t \leftarrow \mathbf{populate}(x^t)$ ▷ $\mathrm{H}_x^t \in \mathbb{R}^{n\times k}$

$\mathrm{H}_{\nabla L}^t \leftarrow \mathbf{populate}(\nabla_x L^t)$ ▷ $\mathrm{H}_{\nabla L}^t \in \mathbb{R}^{n\times k}$

def step($\bar{\mathrm{H}}_{\nabla L}^t \in \mathbb{R}^{n\times k}$: *Normalized gradients*):

  $sum \leftarrow \sum \bar{\mathrm{H}}_{\nabla L}^t$ ▷ $sum \in \mathbb{R}^{n\times 1}$

  return $\frac{sum}{k}$

def updateHistory($\mathrm{H} \in \mathbb{R}^{n\times k}$: *History*, $U \in \mathbb{R}^{n\times 1}$: *Update*):

  pop(H) ▷ `Remove the oldest history vector`

  push(H, U) ▷ `Add the update vector to history`

  return H

while $itr < itr_{max}$ **do**

  $\nabla_x L^{t,max} \leftarrow \|\mathrm{H}_{\nabla L}^t\|_\infty$ ▷ $\nabla_x L^{max} \in \mathbb{R}^{n\times 1}$

  $\bar{\mathrm{H}}_{\nabla L}^t \leftarrow {}^{\mathrm{H}_{\nabla L}^t}/_{\nabla_x L^{t,max}}$ ▷ $\bar{\mathrm{H}}_{\nabla L}^t \in \mathbb{R}^{n\times k}$

  $\delta x^t \leftarrow \mathbf{step}(\bar{\mathrm{H}}_{\nabla L}^t)$ ▷ $\delta x^t \in \mathbb{R}^{n\times 1}$

  $x^{t,max} \leftarrow \mathbf{max}(\mathrm{H}_x^t)$ ▷ $x^{t,max} \in \mathbb{R}^{n\times 1}$

  $x^{t,min} \leftarrow \mathbf{min}(\mathrm{H}_x^t)$ ▷ $x^{t,min} \in \mathbb{R}^{n\times 1}$

  $x^{t,ref} \leftarrow {}^{(x^{t,max} + x^{t,min})}/_2$

  $x^{t,diff} \leftarrow x^{t,max} - x^{t,min}$

  $x^{t+1} \leftarrow x^{t,ref} - \delta x^t \odot (x^{t,diff} + \eta_{min})$

  $L^{t+1} \leftarrow \mathbf{loss}(\mathbf{P}(x^{t+1}))$

  $\nabla_x L^{t+1} \leftarrow \mathbf{BackProp}(x^{t+1}, L^{t+1})$

  $\mathrm{H}_x^{t+1} \leftarrow \mathbf{updateHistory}(\mathrm{H}_x^t, x_i^{t+1})$

  $\mathrm{H}_{\nabla L}^{t+1} \leftarrow \mathbf{updateHistory}(\mathrm{H}_{\nabla L}^t, \nabla_x L^{t+1})$

  $itr = itr + 1$

end

Note that the algorithm works seamlessly with just gradient signs as well. However using only gradient signs results in inferior performance due to lack of information, i.e. gradient magnitudes. Figure 3.4 shows a comparison between sign only and the default algorithm.

Experiments

First we perform experiments with two toy problems before moving on to a neural network. We compare performance of the manual algorithm against *Adam* with varying learning rate η along with default momentum scale i.e. $\beta_1 = 0.9$ and$\beta_2 = 0.999$.

Sum of Squares First we test our algorithm on a sum of squares problem (section 2.3.1). We maintained $\mathrm{H}^t_{x,i}$ and $\mathrm{H}^t_{\nabla L,i}$ for $k = 5$ steps i.e H^t_x and $\mathrm{H}^t_{\nabla L} \in \mathbb{R}^{n\times 5}$. Figure 3.3 shows performance of *MNG* with different η_{min} against *Adam* with different learning rates η. *MNG* outperforms *Adam* by a significant margin. Note that the smaller the η_{min} the longer it takes to start making substantial progress for *MNG* and the later it tends to stall. The reason for the former is that since at start the $\mathrm{H}^t_{x,i}$ is populated with the same variable i.e x^0_i, term $x^{0,diff}_i$ is zero. Thus all the progress at the start comes from η_{min}, the smaller it is the longer it takes for the term $x^{t,diff}_i$ to get bigger and contribute something significant towards optimization. Note that $\eta_{min} = 1e-7$ proves to be too small and *MNG* makes no progress at all. On the other hand a big η_{min} also means that the algorithm will stall earlier, the reason being while $x^{t,diff}_i$ becomes small upon reaching the minima, but η_{min} is a constant and doesn't decrease. One modification here could be to decay η_{min} with time.

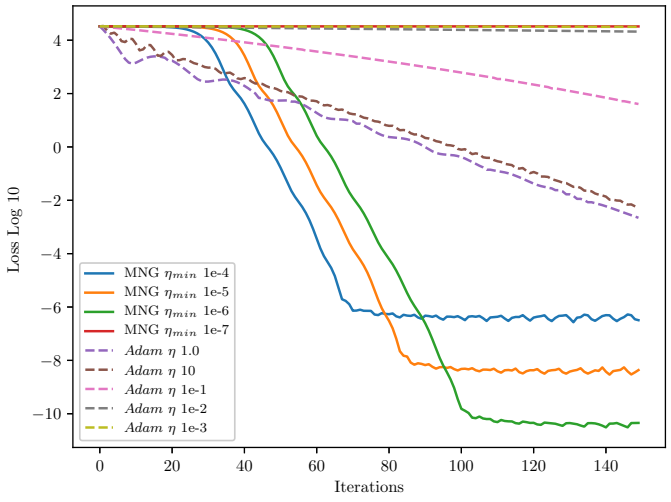


Figure 3.3: *Sum of squares: performance of MNG against Adam. MNG outperforms Adam across different η_{min} and η.*

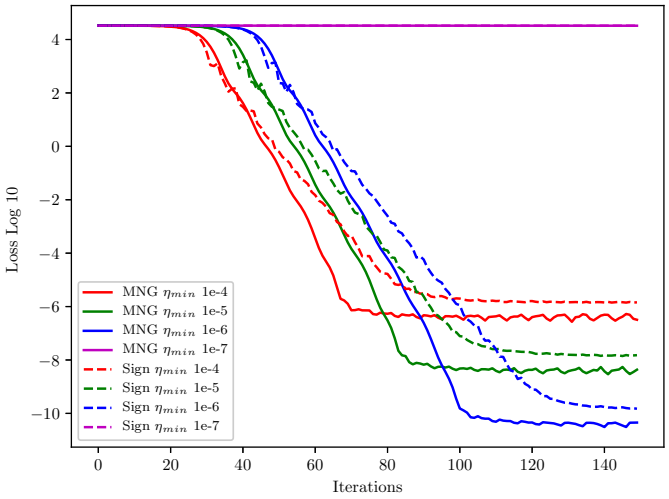


Figure 3.4: *Sum of squares: performance of MNG against 'Sign only version'. The default algorithm always outperforms its sign only counterpart for a given* η_{min}.

Rosenbrock Now we test our algorithm on the Rosenbrock function (section 2.3.2). We see that *Adam* tends to perform better than *MNG*. Notice the effect of increasing η_{min}. The higher it is, the faster the convergence. But similar to previous experiments η_{min} is a constant, thus the algorithm has a hard time around minima as it jumps around it e.g in figure 3.5 see the curve for *MNG* with $\eta_{min} = 1e-3$. Also it might be that for higher k, *MNG* performs better. We will explore the impact of choosing k in more detail in when we train optimizers..

Mnist Finally we test *MNG* with $k = 10$ on $MlpMnsit$ (section 2.3.3). Figure 3.6 shows the performance against *Adam*. *Adam* outperforms *MNG* by significant margin. Note that increasing η_{min} from $1e-4$ to $1e-2$ boosts the performance greatly. We suspect that noise plays a major role here causing x^{diff} $(k = 10)$ to become small and slowing down learning but since η_{min} is a constant, it is unaffected by noise allowing us to make progress. Finally since *MNG* does show some potential, we move towards the training the meta optimizer phase, where the hope is that the meta learned $\mathbf{O}_m$ will learn some non linear weighting of the gradients and output a better δx^t.

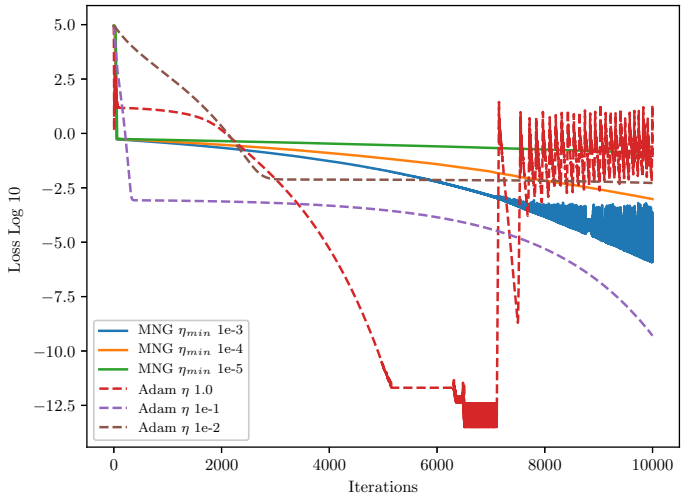


Figure 3.5: *Performance of MNG against Adam on Rosenbrock function. Adam outperforms MNG here.*

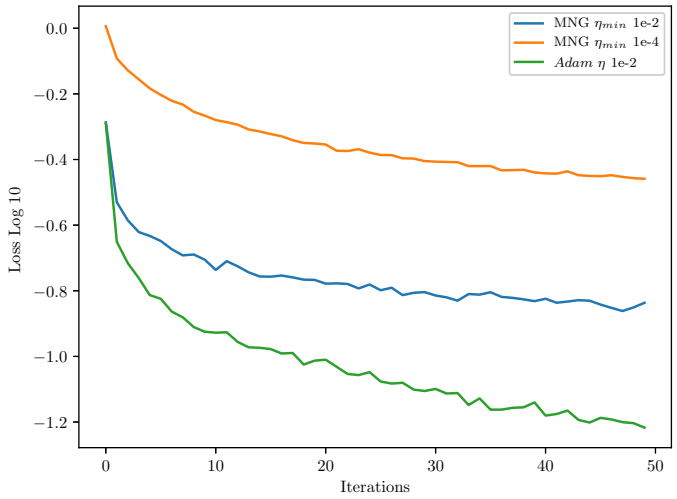


Figure 3.6: *Performance of MNG with $k = 10$ on Mnist. Adam outperforms MNG by a huge margin.*

3.4.2 Learning to Output δx^t

Algorithm

The full algorithm is almost the same as algorithm 1 but now our trained meta optimizer $\mathbf{O}_m$ i.e. *Mlp* (section 3.3) calculates the step δx_i^t. Also towards the

end of the main training loop, optimizer $\mathbf{O}$ updates the weights of the meta optimizer $\mathbf{O}_m$. Algorithm 2 lists the full algorithm. The while loop gives one full meta iteration.

Algorithm 2: Learning to Optimize With Normalized Inputs

Input: $\mathbf{P}(x^t)$, $x^t \in \mathbb{R}^{n\times 1}$ ▷ `Problem and its variables`

$L^t \leftarrow \mathbf{loss}(\mathbf{P}(x^t))$

$\nabla_x L^t \leftarrow \mathbf{BackProp}(L^t, x^t)$ ▷ $\nabla_x L^t \in \mathbb{R}^{n\times 1}$

$\mathrm{H}_x^t \leftarrow \mathbf{populate}(x^t)$ ▷ $\mathrm{H}_x^t \in \mathbb{R}^{n\times k}$

$\mathrm{H}_{\nabla L}^t \leftarrow \mathbf{populate}(\nabla_x L^t)$ ▷ $\mathrm{H}_{\nabla L}^t \in \mathbb{R}^{n\times k}$

def step($\bar{\mathrm{H}}_{\nabla L}^t \in \mathbb{R}^{n\times k}$: *Normalized gradients*):

  return $\mathbf{O}_m(w^t, \bar{\mathrm{H}}_{\nabla L}^t)$

def updateHistory($\mathrm{H} \in \mathbb{R}^{n\times k}$: *History*, $U \in \mathbb{R}^{n\times 1}$: *Update*):

  pop(H) ▷ `Remove the oldest history vector`

  push(H, U) ▷ `Add the update vector`

  return H

while $itr < itr_{max}$ **do**

  $\nabla_x L^{t,max} \leftarrow \|\mathrm{H}_{\nabla L}^t\|_\infty$ ▷ $\nabla_x L^{max} \in \mathbb{R}^{n\times 1}$

  $\bar{\mathrm{H}}_{\nabla L}^t \leftarrow {}^{\mathrm{H}_{\nabla L}^t}/_{\nabla_x L^{t,max}}$ ▷ $\bar{\mathrm{H}}_{\nabla L}^t \in \mathbb{R}^{n\times k}$

  $\delta x^t \leftarrow \mathbf{step}(\bar{\mathrm{H}}_{\nabla L}^t)$ ▷ $\delta x^t \in \mathbb{R}^{n\times 1}$

  $x^{t,max} \leftarrow \mathbf{max}(\mathrm{H}_x^t)$ ▷ $x^{t,max} \in \mathbb{R}^{n\times 1}$

  $x^{t,min} \leftarrow \mathbf{min}(\mathrm{H}_x^t)$ ▷ $x^{t,min} \in \mathbb{R}^{n\times 1}$

  $x^{t,ref} \leftarrow {}^{(x^{t,max} + x^{t,min})}/_2$

  $x^{t,diff} \leftarrow x^{t,max} - x^{t,min}$

  $x^{t+1} \leftarrow x^{t,ref} - \delta x^t \odot (x^{t,diff} + \eta_{min})$

  $L^{t+1} \leftarrow \mathbf{loss}(\mathbf{P}(x^{t+1}))$

  $\nabla_x L^{t+1} \leftarrow \mathbf{BackProp}(x^{t+1}, L^{t+1})$

  $\nabla_w L^{t+1} \leftarrow \mathbf{BackProp}(w^t, L^{t+1})$

  $\mathrm{H}_x^{t+1} \leftarrow \mathbf{updateHistory}(\mathrm{H}_x^t, x_i^{t+1})$

  $\mathrm{H}_{\nabla L}^{t+1} \leftarrow \mathbf{updateHistory}(\mathrm{H}_{\nabla L}^t, \nabla_x L^{t+1})$

  $w^{t+1} \leftarrow \mathbf{O}(w^t, \nabla_w L^{t+1})$

  $itr = itr + 1$

end

Experiments

We train the meta optimizer $\mathbf{O}_m$ for 200k meta iterations on *Mnist* dataset coupled with the network $MlpMnsit$, using *Adam* optimizer. We decay *Adam*'s learning rate from $2.5e-4$ to $5e-6$ over the 200k meta iterations. We use the same training regime as described in 2.2. The range for $train_{reset}$ is 1000 meta iterations to 20000 meta iterations. For each meta iteration we calculate $\nabla_x L^t$ on a random batch of size 128 from the $Mnist$ dataset. $\nabla_x L^t$ is then fed to the meta optimizer as input as described in algorithm 2 and illustrated in figure 2.1. In our experiments we trained the meta optimizer $\mathbf{O}_m$ with several different number of timesteps k with $\eta_{min} = 1e-4$. We note that on average the training takes about half an hour on an Nvidia GTX TITAN Black.

Evaluation on $MlpMnsit$ After training the meta optimizer we validate it on mnist for 50 epochs with a batch size of 128. Figure 3.7 shows the loss curves for different k. Note that all of the configurations are very sensitive to η_{min} and for optimum performance η_{min} needs to be tuned after training. We noticed that for small number of k, optimum η_{min} is small as well and it increases as we increase k, i.e $k = 5$ performs the best with $\eta_{min} = 1e-3$ whereas for $k = 100$ optimum $\eta_{min} = 5e-2$. We also note that $k = 100$ performs the worst for $\eta_{min} = 1e-4$, but performs the best for $\eta_{min} = 5e-4$. This flip in performance for all k happens because for smaller number of k it is far easier to reach the maximum step length i.e $+1$ or -1. For example refer to the table 3.1 and note that for $k = 5$ the meta optimizer outputs 0.9999 step length with just 3 (out of 5) gradients with maximum magnitude. Whereas for larger k the minimum total magnitude required to output step of maximum length also increases as the meta optimizer looks much farther back in to the history, i.e. for $k = 10$ there need to be at least 6 gradients, instead of 3 like with the case of $k = 5$, with maximum magnitude to output step of maximum length or near maximum length. Thus configurations with smaller k require less η_{min}. We also note that even when training directly with higher η_{min}, the learning rate still needed to be tuned. From now on we only report results after tuning the learning rate. Figure 3.8 shows an evaluation run of a learned optimizer ($k = 100$) against *Adam* with best settings.

Table 3.1 shows the non linear relationship learned by the meta optimizer between inputs i.e normalized gradients, and step δx_i^t. Notice how δx_i^t decreases in magnitude, eventually flips sign and increases in magnitude again as we go from the top row to the sixth row. Other rows depict other non linear learned steps. Finally it's worth noticing that this particular learned optimizer is a bit biased towards the positive side, see the last row. Ideally one would expect δx_i^t to be close to zero as well. We notice that usually with more training this behavior goes away. Still however, more experiments need to be done to understand the bias's effect in a better way.

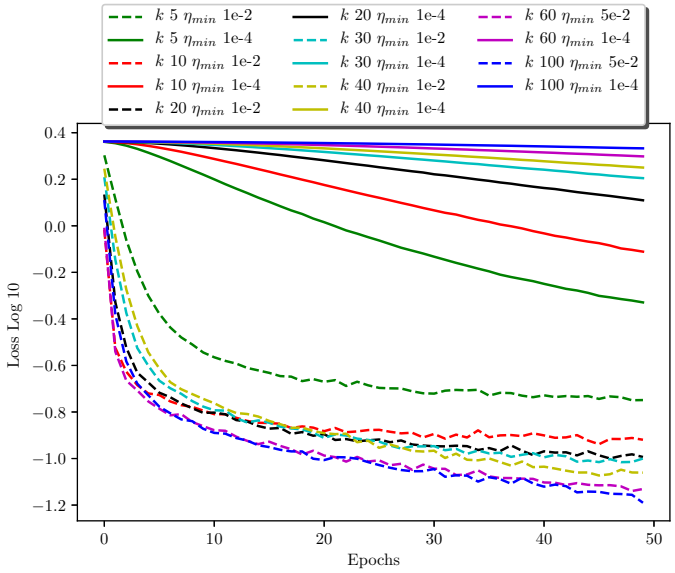


Figure 3.7: *Comparison of learned optimizers with different k, before and after tuning η_{min}. We see that increasing k leads to increase in performance.*

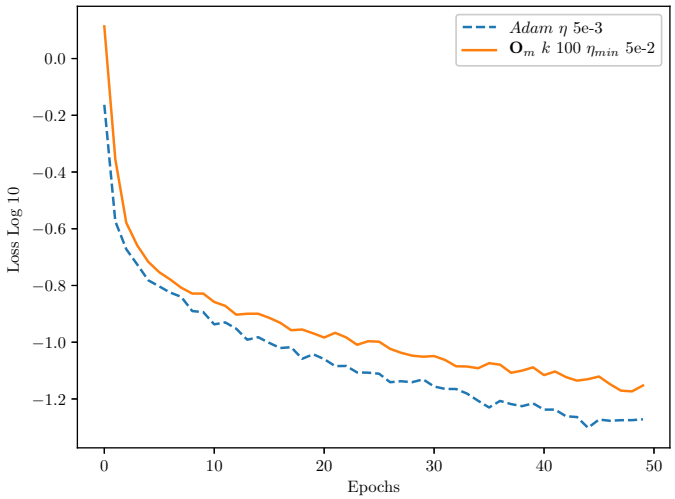


Figure 3.8: *k Timesteps with $k = 100$ vs Adam optimizer. Adam clearly shows better performance here.*

3.5 k Momentums

We see that while diminishing in nature, increasing number of timesteps does improve performance. But for practical purposes this is not a viable option, thus now we look towards using momentums at different timescales. This allows us

$\nabla_x L_i^{t-4}$	$\nabla_x L_i^{t-3}$	$\nabla_x L_i^{t-2}$	$\nabla_x L_i^{t-1}$	$\nabla_x L_i^t$	δx_i^t
1.0	1.0	1.0	1.0	1.0	0.99999666
1.0	1.0	1.0	1.0	-1.0	0.99984455
1.0	1.0	1.0	-1.0	-1.0	0.99128538
1.0	1.0	-1.0	-1.0	-1.0	-0.9901197
1.0	-1.0	-1.0	-1.0	-1.0	-0.99978876
-1.0	-1.0	-1.0	-1.0	-1.0	-0.99999416
-0.1	-0.1	-0.1	-0.1	1.0	0.96777165
-0.1	-0.1	-0.1	-0.5	1.0	0.74221277
-0.1	-0.1	-0.25	-0.25	1.0	0.84402364
-0.1	-0.25	0	0.25	1.0	0.41817734
1.0	0.25	0.0	-0.25	-1.0	0.0942957
0.0	0.0	0.0	-0.0	0.0	0.26363501

Table 3.1: *Steps learned by the meta optimizer* $\mathbf{O}_m$ *for* $k = 5$ *when using* k *Timesteps for a given history of gradients* $\nabla_x L^t$.

to encode history in a much more compact way. It should be noted that now k refers to number of timescales and not timesteps.

Using a different x_i^{ref} Recall that our reference point on page 10 is given by $x^{t,ref} = \frac{x^{t,max}+x^{t,min}}{2}$ and since momentums have the potential to look extremely far back in the past, $x^{t,ref}$ can be very far back in the past as well, equivalently the point where we shoot from, moves back further and further in time with increasing range of k. We note that this slows down convergence for $\mathbf{O}_m$, thus instead we opt for the more classic choice of using problem variables at timestep t i.e. $x_i^{t,ref} = x_i^t$. Update equation (3.4) is now given as follows:

$$x_i^{t+1} = x_i^t + \delta x_i^t \cdot (x_i^{t,diff} + \eta_{min}) \tag{3.5}$$

.

3.5.1 Algorithm

Apart from the new update equation (3.5), the only difference form algorithm 2 is that now we update the history differently. The new update function is given by algorithm 3.

Algorithm 3: updateHistory for k Momentums

Input: $m \in \mathbb{R}^{1\times k}$ ▷ `Momentum vector used for updates.`

```
def updateHistory(H ∈ R^{n×k} : History, U ∈ R^{n×1}: Update):
    return m · H + (1 − m) · U
```

3.5.2 Experiments

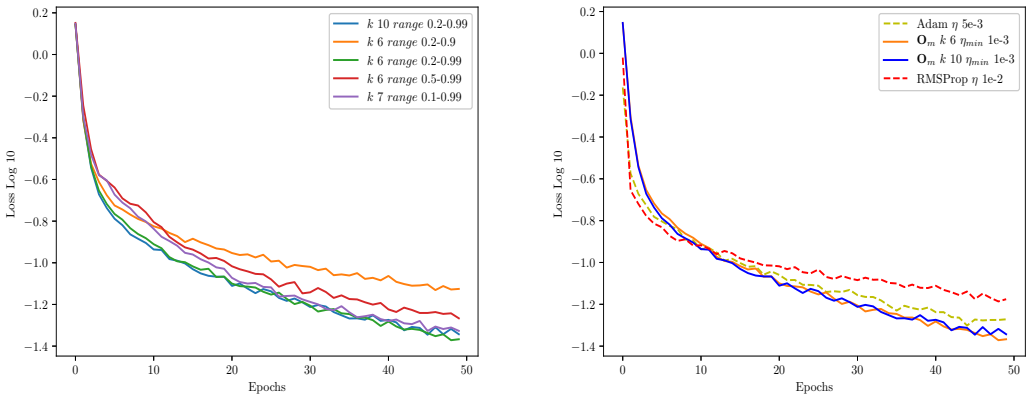
We train the meta optimizer with the following configurations:.

- $k = 6$, timescales linearly spaced from 0.2 - 0.9.
- $k = 6$, timescales linearly space from 0.5 - 0.99.
- $k = 6$, timescales linearly spaced from 0.2 - 0.99.
- $k = 7$, timescales linearly spaced from 0.1 - 0.99.
- $k = 10$, timescales linearly spaced from 0.2 - 0.99.

We use the same training setup as described in subsection 30. We noticed that on average training on an Nvidia GTX TITAN Black takes about 25 minutes.

Evaluation on $MlpMnsit$ After training we evaluate the trained optimizers for 50 epochs with 128 batch size on the same network. Figure 3.9a shows their performance against each other with $\eta = 1e-3$. The keyword *range* in the figure denotes the range of the used timescales. We note that increasing k did not have a significant impact on the performance. Decreasing the range i.e 0.5 - 0.99, on the other hand had a strong impact. This indicates there is some information at a timescale as low as 0.2, that the meta optimizer deems necessary. Overall we see that k= 10 and 6 with *range* = [0.2, 0.99] performed the best. We also noted that using new x^{ref} impacts the performance positively giving more consistent and better results. Moreover now the algorithm relied relatively less on η_{min}, as it now only needed to increased by factor of 10 instead of 100 as for most of the k in subsection 30. Figure 3.9b compares performance of the the top two performers against *Adam* and $RMSProp$ with best settings. We see that while initially slower the meta optimizers outperform both *Adam* and $RMSProp$.

Evaluation on $ConvCifar_0$ In order to see how well the learnt optimizer generalize we now evaluate them on a much larger and and unseen network $ConvCifar_0$. We choose the best settings for both *Adam* and $RMSProp$ with $\eta = 1e-4$. Similarly for the learnt optimizers $\mathbf{O}_m$ we choose best setting and



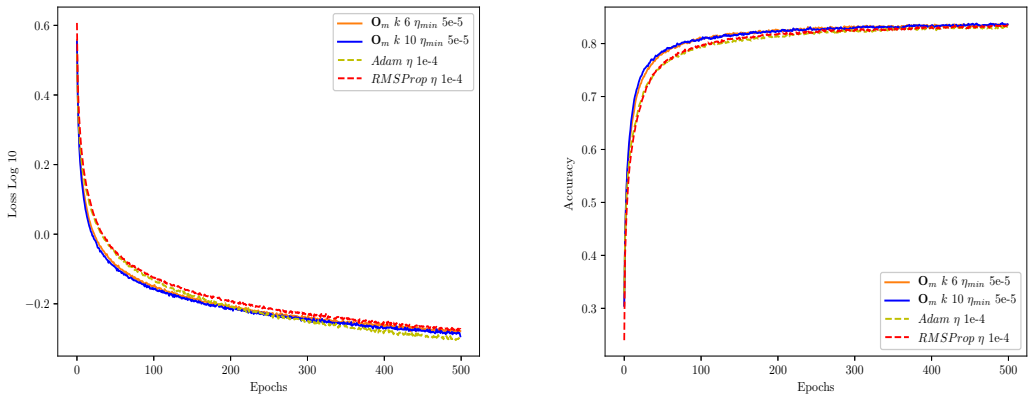
(a) *Comparison of the learned* $\mathbf{O}_m$*s.* **(b)** *Trained* $\mathbf{O}_m$ *vs Adam and RMSProp*

Figure 3.9: *Results obtained when evaluating on MlpMnsit with* $\eta_{min} = 1e-3$*. 'range' in fig (a) denotes the range of timescales used.*

set $\eta_{min} = 5e-5$. We ran the validation run for 500 epochs. Figure 3.10a shows the loss curves for the optimizers while figure 3.10b shows the accuracy, we see that despite not being trained on $ConvCifar_0$ both the learnt optimizers fare well against *Adam* and *RMSProp*. Outperforming *RMSProp* throughout the validation, while *Adam* on the other hand manges to take the lead towards the end. Although the learnt optimizers achieve higher accuracy on the test set, this is not the criteria we use to grade our optimizers as we are more concerned with the loss on the train set. Thus we consider *Adam* to be the better performer here, followed by $\mathbf{O}_m$ with $k = 10$ and $k = 6$. Amongst the learnt optimizer $k = 10$ showed better performance throughout. And then finally *RMSProp* performed the worst.

Using Normalized H_x^t **as inputs** Inspired by [12] we also feed normalized problem variables to $\mathbf{O}_m$ as inputs. The hope is that the meta optimizer will learn some relation between x_i^t and the minima and thus produce a better δx_i^t. However while evaluating on *MlpMnsit* we did not notice any performance improvement. And when the same model is applied on a different network i.e $ConvCifar_0$, we see a sharp decline in performance. This is because the optimizer does learn some relationship between x^t and δx^t on *MlpMnsit* that is just not there in $ConvCifar_0$, note that on the other hand the relationship between gradients and δx^t is much more universal.

***Adam* like Normalization** *Adam* normalizes the moving average of gradients with a moving average of square gradients. This accelerates learning for parameters with very small gradients due to squared gradient term in the de-



(a) *Loss curves on* $ConvCifar_0$. **(b)** *Accuracy achieved on* $ConvCifar_0$.

Figure 3.10: *Comparison of the learnt optimizers with Adam on* $ConvCifar_0$ *while using best settings for all optimizers.*

nominator 1.1. We implement a similar normalization scheme for our algorithms and experimented. However we did not notice any improvement when evaluated on $MlpMnsit$, thus we kept the normalization already in use. Moreover normalization with squared gradients will also require us to keep at least one more matrix in memory i.e the matrix of squared gradients.

3.6 Limitations

We notice that the learned optimizer steps are up to two times slower than a simple *Adam* step. If one considers time needed to be the main criteria for optimization, than handcrafted optimizers are definitely the better choice. Further more one needs to store 2 history matrices with a total of $n \times k$ parameters each in memory, which can incur large memory footprint for very large networks. Before being applicable more widely much more thorough evaluation of the algorithm's performance still needs to be done on different problems.

3.7 Future Work

We noticed the algorithm relies heavily on η_{min}, thus the effect of choosing different η_{min} needs to be studied more thoroughly, along with training the networks with a decaying η_{min}. Even better would be to try and get rid of η_{min} as then the algorithm will only be left with an automatic learning rate i.e. x^{diff}. Also the step δx^t is currently in the range [-1.0, 1.0], the effect of increasing and decreasing the range warrants attention as well and needs to be researched. One could also

add small Gaussian noise to the δx^t and study the effects. Furthermore effect of using more k is another hyper parameter that needs to be looked at more thoroughly. Also note that all of the updates presented after the handcrafted algorithm *MNG*, can be applied to *MNG* as well. Finally application on more datasets will give one more clear picture of the algorithm's performance.

Summary We started with stating our motivation, and then designed a manual algorithm as a precursor to *learning to optimize*. We then moved towards training our optimizer instead. We showed that the learnt optimizers outperformed *Adam* and *RMSprop* with best settings, on the problem on which they were trained. Furthermore the learnt optimizers generalized well to an unseen and much larger network, showing competitive performance against *Adam* and *RMSProp*. We also discussed some limitations of our approach. Finally, we looked at some future pathways for further research.

Chapter 4

Multiscale Adam

4.1 Motivation

As mentioned before *Adam*[6] uses normalized inputs, thus potentially lending its self as an ideal input for neural networks. We ourselves tried a similar normalization in 3.5.2. The fact that *Adam* steps already contain enough information for optimization forms the motivation for the current approach.

4.2 Algorithm

Given k *Adam* steps computed at different timescales as inputs, $\mathbf{O}_m$ outputs a weighted average over the inputs as the output step δx^t. The goal is that the learned models should perform among the top few input *Adam* timescales. Algorithm 4 lists the steps in detail.

Algorithm 4: Multiscale Adam

Input: $\mathbf{P}(x^t)$, $x^t \in \mathbb{R}^{n\times 1}$, **Adams** ▷ `Problem, problem variables, array of` *`Adam`* `optimizers at different timescales.`

$steps \leftarrow \mathbf{array}(len(\mathbf{Adams}))$
while $itr < itr_{max}$ **do**
  $L^t \leftarrow \mathbf{loss}(\mathbf{P}(x^t))$
  $\nabla_x L^t \leftarrow \mathbf{BackProp}(x^t, L^t)$ ▷ $\nabla_x L^t \in \mathbb{R}^{n\times 1}$
  for $i \leftarrow 1$ **to** $len(\mathbf{Adams})$ **do**
    $Adam \leftarrow \mathbf{Adams}[i]$
    $\delta x^t_{Adam_i} \leftarrow Adam(\nabla_x L^t)$
    $steps[i] \leftarrow \delta x^t_{Adam_i}$
  end
  $\delta x^t \leftarrow \mathbf{O}_m(w^t, steps)$
  $x^{t+1} \leftarrow x^t + \eta \cdot \delta x^t$
  $L^{t+1} \leftarrow \mathbf{loss}(\mathbf{P}(x^{t+1}))$
  $\nabla_w L^{t+1} \leftarrow \mathbf{BackProp}(w^t, L^{t+1})$
  $w^{t+1} \leftarrow \mathbf{O}(w^t, \nabla_w L^{t+1})$
  $itr \leftarrow itr + 1$
end

4.3 Architecture

We experiment with the following different architectures:

- Linear weighted average, (LA).
- Multi Layered Percpetron (MLP).
- Recursive Neural Network (RNN).

The model weights are initialized from a normal distribution with $\mu = 0$ and $\sigma = 0.01$.

4.3.1 Linear Weighted Average (LA)

Model LA has the same number of trainable variables as inputs i.e $len(w^t) = k$. The input weights are connected directly to the output $\in \mathbb{R}^{1\times 1}$. We experiment with the following configurations:

- Normalized weights, LA_a. i.e instead of w^t we apply $\bar{w}^t$ to the inputs.

$$\bar{w}_i^t = w_i^t / \sum w^t$$

- Normalized absolute values of weights, LA_b. i.e

$$\bar{w}_i^t = \|w_i^t\| / \sum \|w^t\|$$

  This forces the applied weights to be positive.

- No constraint, LA_c.

4.3.2 Multi Layered Percpetron (MLP)

We experiment with a single layered mlp. The hidden layer consists of 50 neurons with *relu* activations. The output layer has the same number of outputs as inputs i.e output layer $\in \mathbb{R}^{1\times k}$. The output layer has *softmax* activations, the output probability gives the weights for the inputs.

4.3.3 Recursive Neural Network (RNN)

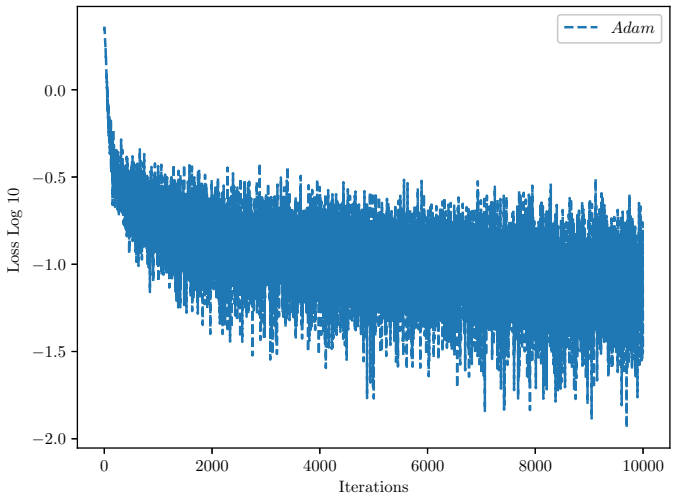


Figure 4.1: *A sample loss curve for Adam without any smoothing.*

The motivation for RNN came from the need to compute a smoothed loss. Figure 4.1 illustrates loss for *Adam* optimizer without any smoothing. As evident from the figure the loss for *Adam* is very noisy, which can prove to be very hard for $\mathbf{O}_m$ to learn to minimize. Using a RNN allows for averaging over the

loss over multiple timesteps thus enabling us to compute a relatively smoother approximation. We use the same architecture as MLP, except that now we do multiple updates to x^t before w^t are updated. This allows us to compute a smoothed loss over a single partial unroll. For an unroll of length u_{max}, loss L^t is given by.

$$L^t = \frac{1}{u_{max}} \sum_{u=0}^{u_{max}} L^u \tag{4.1}$$

Algorithm 5 lists the steps in detail. We experiment with a RNN using one layer and using two layers.

Algorithm 5: Multiscale Adam (RNN)

Input: $\mathbf{P}(x^t)$, $x^t \in \mathbb{R}^{n\times 1}$, **Adams** ▷ `Problem, problem variables, array of` ***Adam*** `optimizers at different timescales.`

$steps \leftarrow array(len(\mathbf{Adams}))$
while $itr < itr_{max}$ **do**
  $L^{total} = 0$
  $L^t \leftarrow \mathbf{loss}(\mathbf{P}(x^t))$
  while $u < u_{max}$ **do**
    $\nabla_x L^t \leftarrow \mathbf{BackProp}(x^t, L^t)$
    for $i \leftarrow 1$ **to** $len(\mathbf{Adams})$ **do**
      $Adam \leftarrow \mathbf{Adams}[i]$
      $\delta x^t_{Adam_i} \leftarrow Adam(\nabla_x L^t)$
      $steps[i] \leftarrow \delta x^i_{Adam_i}$
    end
    $\delta x^t \leftarrow \mathbf{O}_m(w^t, steps)$
    $x^{t+1} \leftarrow x^t + \eta \cdot \delta x^t$
    $L^{t+1} \leftarrow \mathbf{loss}(\mathbf{P}(x^{t+1}))$
    $L^{total} \leftarrow L^{total} + L^{t+1}$
    $u \leftarrow u + 1$
  end
  $L^{avg} \leftarrow \frac{L^{total}}{u_{max}}$
  $\nabla_w L^{avg} \leftarrow \mathbf{BackProp}(w^t, L^{avg})$
  $w^{t+1} \leftarrow \mathbf{O}(w^t, \nabla_w L^{avg})$
  $itr \leftarrow itr + 1$
end

4.4 Experiments

We train the meta optimizer $\mathbf{O}_m$ for 50k meta iterations using *Adam* optimizer as $\mathbf{O}$. We decayed $\mathbf{O}$'s learning rate from $2e-4$ to $5e-6$ over the course of 50k meta iterations.. $train_{reset}$ is set from 1000 meta iteration to 20000 meta iterations randomly.

In the bulk of our experiments we use *Adam* with $k = 6$ different timescales. For comparison we also train $k = 11$ and $k = 5$ [1]. We chose the timescales such that we cover the generally recommended timescales, e.g $\beta_1 = 0.9$ and $\beta_2 = 0.999$, along with timescales that are not recommended in practice, e.g $\beta_1 = 0.5$ and $\beta_2 = 0.555$. Time scales used for $k = 6$ are listed in the table 4.1a. For $k = 5$ we drop $Adam_{.5}$. For $k = 11$ we use the same timescales as $k = 6$ plus we refine the timescales further by using additional ones listed in 4.1b.

Adam	β_1	β_2
$Adam_{.99}$	0.99	0.9999
$Adam_{.9}$	.9	0.999
$Adam_{.8}$	0.8	0.888
$Adam_{.7}$	0.7	0.777
$Adam_{.6}$	0.6	0.666
$Adam_{.5}$	0.5	0.555

(a) *Timescales for $k = 6$*

Adam	β_1	β_2
$Adam_{.95}$	0.995	0.9995
$Adam_{.85}$	.85	0.8855
$Adam_{.75}$	0.75	0.7775
$Adam_{.65}$	0.65	0.6665
$Adam_{.55}$	0.55	0.5555

(b) *Timescales for $k = 11$ include the above mentioned ones plus the ones for $k = 6$*

Table 4.1: *Used time scales.*

4.4.1 Mnist Mlp

First we use *MlpMnsit* as a testing ground for LA_a, LA_b and LA_c with $k = 6$. Then we look in to the effect of using different number of timescales i.e k. Finally we look at how our learned models with $k = 6$ perform with different learning rates. Mnist network architecture is the same as described in 2.3.3. For all experiments we evaluate on *Mnist* for 50 epochs with a batch size of 128.

Comparison LA_a vs LA_b vs LA_c ($k = 6$)

When comparing the 3 configurations as mentioned in 4.3.1, we notice that normalization helps improve the learning significantly Fig 4.2. Table 4.2, shows the final learned weights for the corresponding configurations. It is worth nothing

[1] This work was done before the results in 3.5.2, we did not test with timescales as low as 0.2 and 0.1 here.

that $Adam_{.99}$ and $Adam_{.5}$. consistently have the highest weights, which corresponds to fusing information from the most recent past and the most distant past. Intriguingly, when separate $Adam_{.5}$ performs the worst.

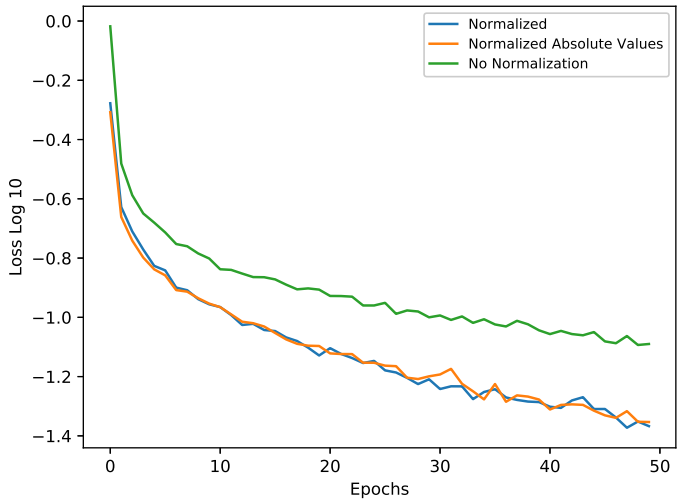


Figure 4.2: *Comparision of the 3 different approaches for linear weighted average. Normalized weights enable the meta optimizer to learn much more efficiently, and thus giving better results on validation.*

$Adam$	LA_a	LA_b	LA_c
$Adam_{.99}$	0.75668	0.66044	0.24911
$Adam_{.9}$	0.06197	0.00366	-0.08944
$Adam_{.8}$	0.68091	0.00065	-0.09293
$Adam_{.7}$	-1.21117	0.00064	0.00168
$Adam_{.6}$	-0.47439	0.00077	0.07846
$Adam_{.5}$	1.18599	0.33386	0.15519

Table 4.2: *Learned weights for* $k = 6$

Note that learned weights for LA_a include negative weights for $Adam_{0.7}$ and $Adam_{0.6}$, which can be equated to moving in the opposite direction i.e a negative learning rate. Whereas for $Adam_{0.5}$ the weight is more than 1.0 which can be interpreted as increasing the learning rate. While for all other inputs the learning rate is decreased. It should be kept in mind that weights were forced to sum upto one, thus when increasing weight for one input, weights for some other inputs must be lowered. Again for LA_b we see that the weights for timescale 0.99 and 0.5 are the highest. Note that we only have positive weights for LA_b as we only consider absolute values. For LA_c again we see that timescales 0.99 and 0.5 have

the most weights. There were a few evaluation instances where LA_c performed much better, but these cases were a rarity.

Finally, we noticed that using LA_b showed more consistent results compared LA_a and thus for further experiments we only consider the former. We will now refer to LA_b as simply LA.

Using different number of k

We now compare the relative performance for $k = 5$, $k = 6$ and $k = 11$ for learning rate $\eta = 1e - 2$. Figure 4.3 shows the performance for different choice of k. We notice that $k = 5$ performs slightly worse than $k = 6$, while $k = 11$ performs the best. The decrease in performance for $k = 5$ makes sense as we remove some information. As for $k = 11$, more information leads to better performance, however we notice that although common, this is not always the case i.e. increasing k does not always improve performance. Table 4.3a shows weights learned for $k = 5$ while table 4.3b shows weights learned for $k = 11$, again we see that the most distant and the most recent timescales have the highest weights.

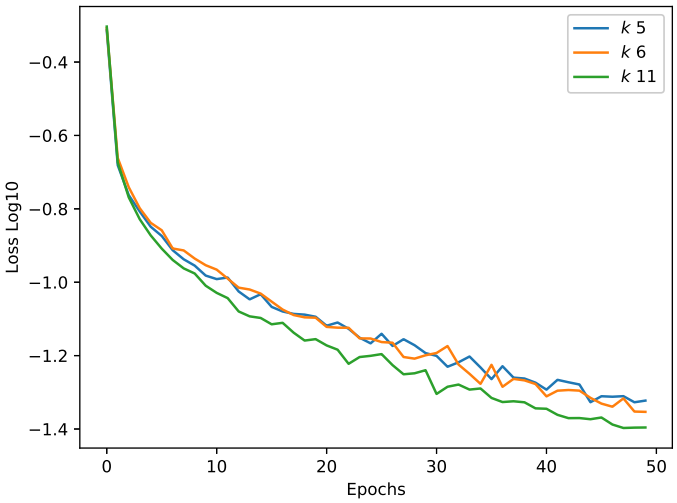


Figure 4.3: *Performance comparison of LA with $k = \{5, 6, 11\}$ on mnist. Using more k leads to better performance in this case.*

Finally Figure 4.4 shows performance of all three k against individual input *Adam* steps i.e. *Adam* with timescales same as what was used for training the meta optimizer. All 3 learned models perform better than individual *Adam* steps. In particular $k = 11$ garners a huge lead very swiftly.

Adam	Normalized (Abs)
$Adam_{.99}$	0.60268
$Adam_{.9}$	0.00327
$Adam_{.8}$	0.00654
$Adam_{.7}$	0.00760
$Adam_{.6}$	0.37990

(a) *Timescales for* $k = 5$

Adam	Normalized (Abs)
$Adam_{.99}$	0.66264
$Adam_{.95}$	0.00295
$Adam_{.9}$	0.00018
$Adam_{.85}$	0.00256
$Adam_{.8}$	0.00493
$Adam_{.7}$	0.00073
$Adam_{.75}$	0.00608
$Adam_{.65}$	0.00615
$Adam_{.6}$	0.00074
$Adam_{.55}$	0.00732
$Adam_{.5}$	0.29241

(b) *Timescales for* $k = 11$ *include the above mentioned ones plus the ones for* $k = 6$

Table 4.3: *Timescales used.*

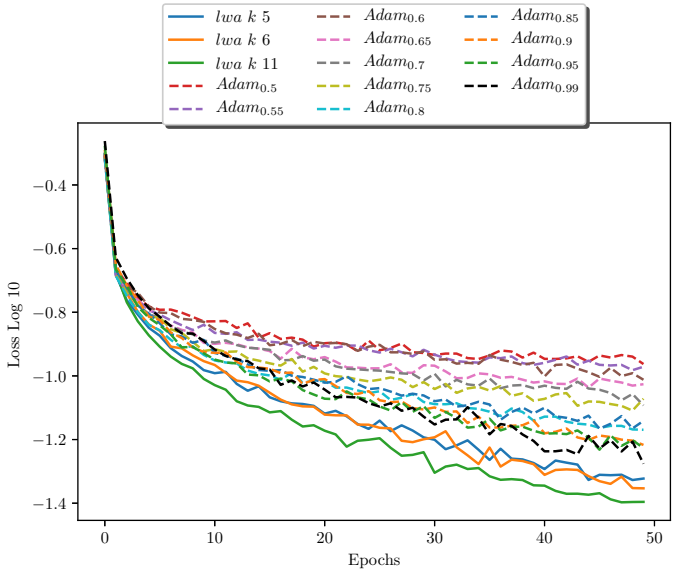


Figure 4.4: *Evaluation runs of LA with* $k = \{5, 6, 11\}$ *against 11 Adam optimizers with timescales same as the inputs of the learned optimizers. Note that the learned optimizers outperform Adam by a great margin.*

Using different learning rates with $k = 6$

In order to investigate the limits of our model, we train the model with multiple learning rates and compare with the performance of *Adam* at those learning rates. We test our model with these fixed learning rates: $\{1e-1, 5e-2, 1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5\}$. Furthermore, in order to check the effect of

learning rate decay we also do cosine annealing [8] and decay $1e-2$ and $5e-2$ to zero over the course of 50 epochs.

To compare performance we plot heatmaps of mean losses achieved over 50 epochs, with different learning rates along x-axis and used optimizers on y-axis. Each column is normalized with the infinity norm, which not only makes the highest performer most prominent but also shows how others perform relative to the best optimizer. The highest performer is the black block in each column. The poorer other optimizers perform relative to the best one the more lighter their shade is.

Heat map 4.5 shows performance grid for LA. Note that for the 4 highest learning rates LA does fairly well, achieving lowest loss for $5e-2$ and $1e-2$, 2nd lowest for $5e-3$ and 3rd lowest for $1e-1$. Also with learning rate decay it performs the best. But from $5e-3$ to $5e-5$ we see a gradual decrease in performance of the learned optimizer. However notice that in these cases the $Adam$ also doesn't perform much better either i.e the shades are quite similar. It should also be noted that the cases where LA does perform the best, it does so by a huge margin, i.e the difference in shades of $Adam$ and LA is very stark for $5e-2$.

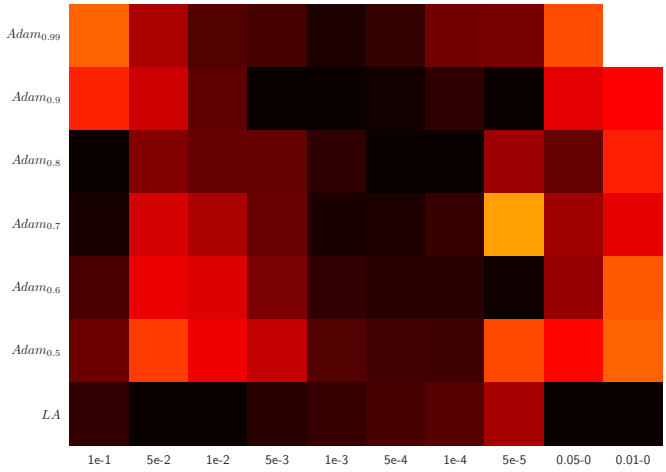


Figure 4.5: *Heat map with LA and Adam at the same timsecales as LA inputs. Learning rates are on x-axis, while optimizers are on y-axis. All columns are normalized by the infinity norm. We see that in 4 cases LA performs the best, but as learning rates start to go down its performance starts to drop as well.*

We focus on these worst performing cases and look further into what can be done to improve the performance. We first try a more powerful learning model i.e switch from linear weighted average LA to MLP. We notice that MLP

performs comparable to LA with very minor performance differences, thus we excluded MLP from further experimentation. Figure 4.6 shows the comparison of LA and MLP for two different learning rates.

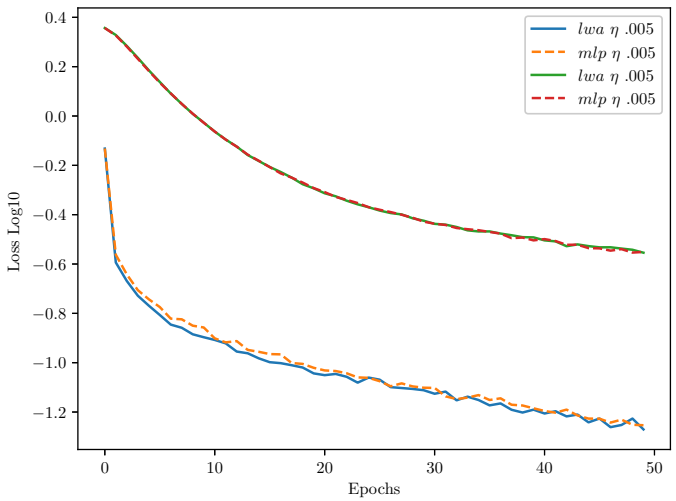


Figure 4.6: *LA vs MLP for 2 different learning rates evaluated on mnist. Despite being a more complex model MLP fails to perform better.*

We suspect this inability to learn something better is due to noisy loss, fig 4.1. Finally we apply our single layered RNN to the problem at hand and average the loss over unroll length of 20 iterations i.e. $u_{max} = 20$. Algorithm 5 lists the steps in detail. We still do 50k meta iterations, but note that now each meta iteration is more expensive in terms of time. We noticed that RNN takes about 3-4 hours to train on $MlpMnsit$. Although more expensive, we notice this vastly improves performance for the worst performing learning rates by achieving best results for $5e-4$, $1e-4$; and 3rd best for $5e-5$. The trained RNN also retains the best results from other learning rates. Figure 4.7 shows the heatmap for the learned RNN. We believe along with smoothing the loss, being able to *see* 20 steps ahead in the future and propagate gradients to the current timestep also helps in learning as well.

Finally we apply RNN with 2 layers to the same problems. We see that it improves the results achieved by the single layered RNN by a big margin. Figure 4.8 shows the new heatmap. Not only it achieves best results for all learning rates with the exception of $5e-3$, but it also improves on the existing best results, notice that the shade difference between $Adam$ and the learnt optimizers grows more stark. Figure 4.10 lists all loss curves for the 2 layered RNN against individual $Adam$ timescales. We see that even for the case of $5e-3$ RNN performs very close to the top performer.

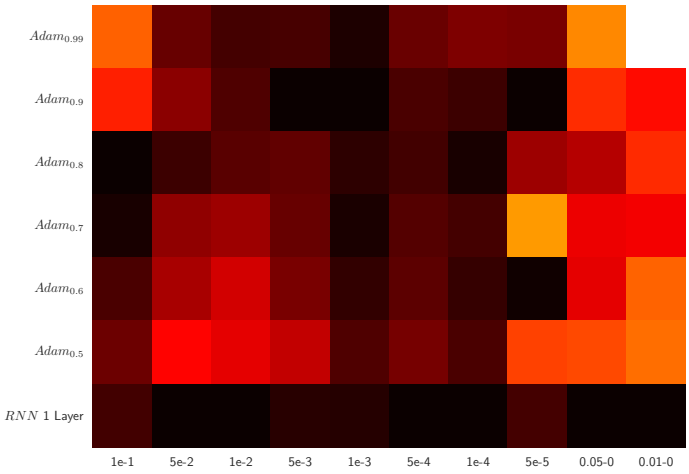


Figure 4.7: *Heat map with LA and Adam at the same timsecales as RNN inputs. Using the rnn vastly improves results over just a linear weighted average. It achieves best performance for 6 different learning rate schedules, while performing among the top ones for the rest.*

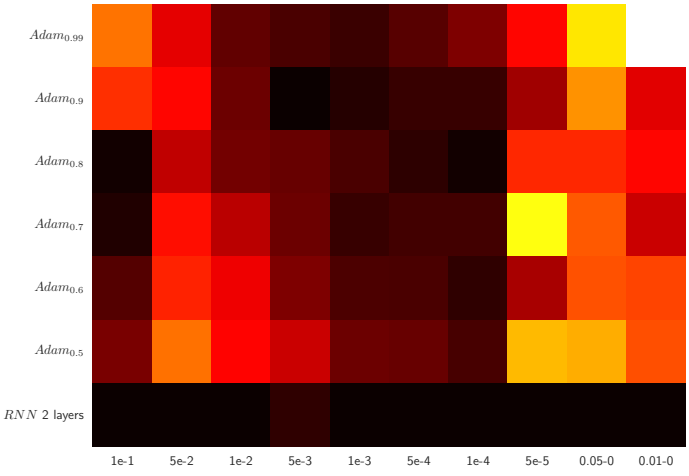
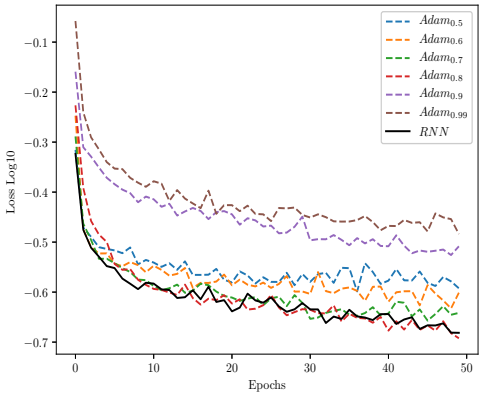
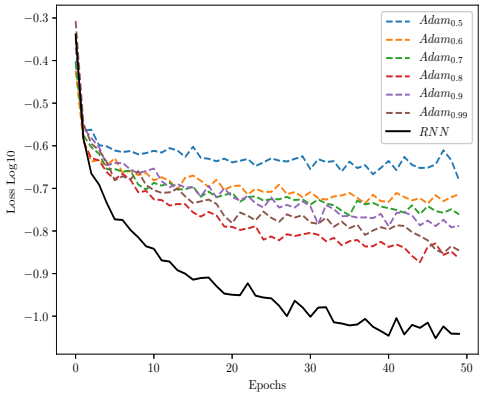


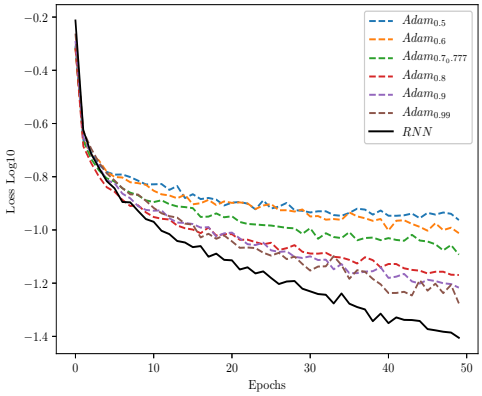
Figure 4.8: *Heat map with the 2 layer RNN and Adam at the same timsecales as the RNN inputs. This model achieved the best results amongst the models tried so far.*



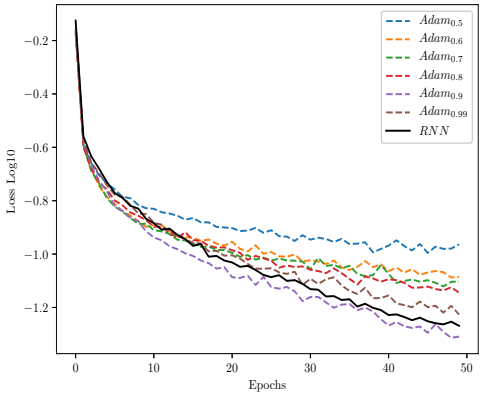
(a) $\eta = 1e-1$



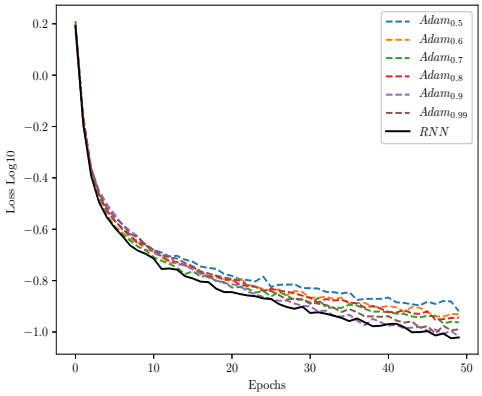
(b) $\eta = 5e-2$



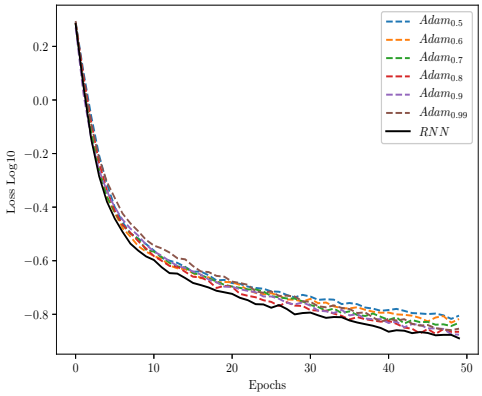
(c) $\eta = 1e-2$



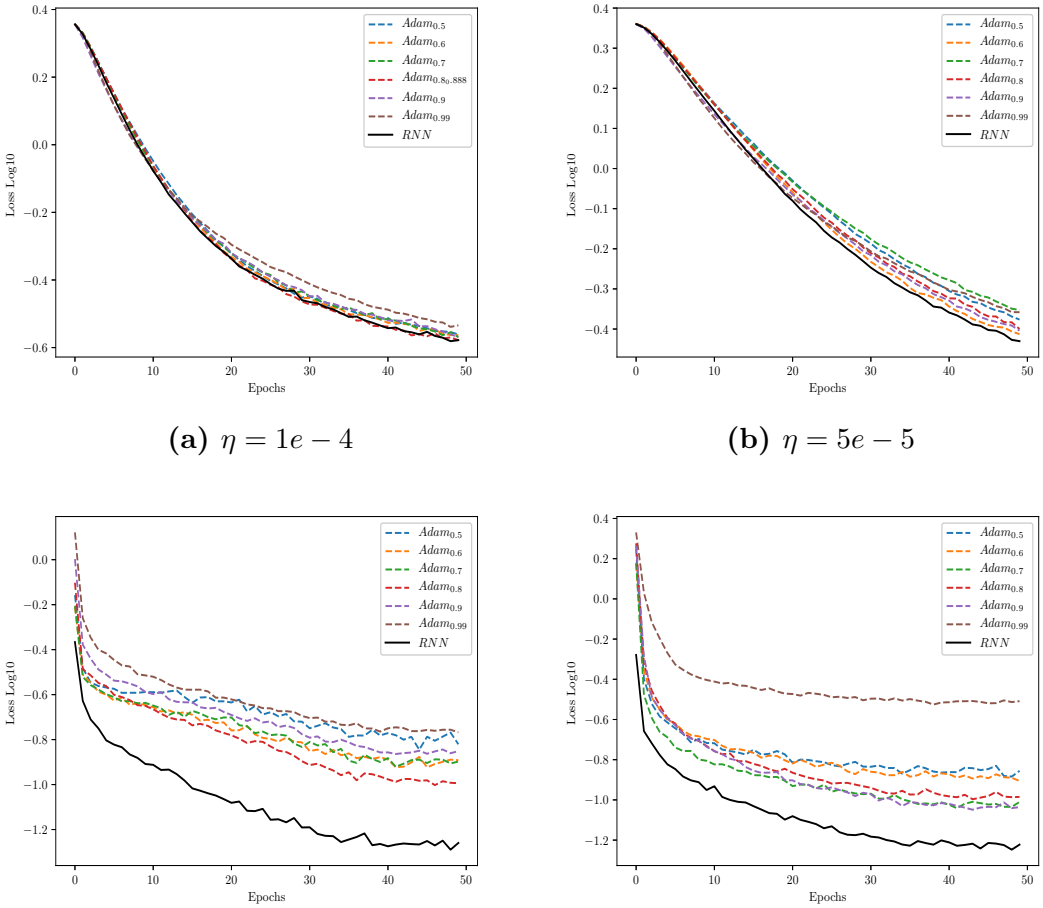
(d) $\eta = 5e-3$



(e) $\eta = 1e-3$



(f) $\eta = 5e-4$



(a) $\eta = 1e-4$ **(b)** $\eta = 5e-5$

(c) η *decayed from* $5e-2$ *to 0.0* **(d)** η *decayed from* $1e-2$ *to 0.0*

Figure 4.10: *Performance of 2 layered RNN across multiple learning rates, $k = 6$ on MlpMnsit. Notice that the learned rnn outperforms the individual Adam steps. The only case where it does not achieve the best result is $\eta = 5e-3$, there it achieves the 2nd best result.*

4.4.2 Mnist Convnet

In this section we train and test on two convolutional networks instead. We train $\mathbf{O}_m$ on a smaller network $ConvMnsit_1$ and test it on a larger network $ConvMnsit_0$. We train LA and RNN 2 layers on $ConvMnsit_1$ with a learning rate of $5e-4$ for 50k meta iterations and then apply the learned optimizers to $ConvMnsit_0$. Figure 4.11 shows how the learned models fare against individual $Adam$ timescales when applied on $ConvMnsit_0$. The trained RNN showed the best performance by far, despite being evaluated on a network it wasn't trained on. LA on the other hand fails to generalize as well, although it still does perform amongst the top performers. Again smoothing the loss proves to be of great use. We also noted that if we continue to train LA for longer durations than the normal budget of 50k updates then eventually it too learns to perform better. We suspect lack of smoothing causes this delay in learning.

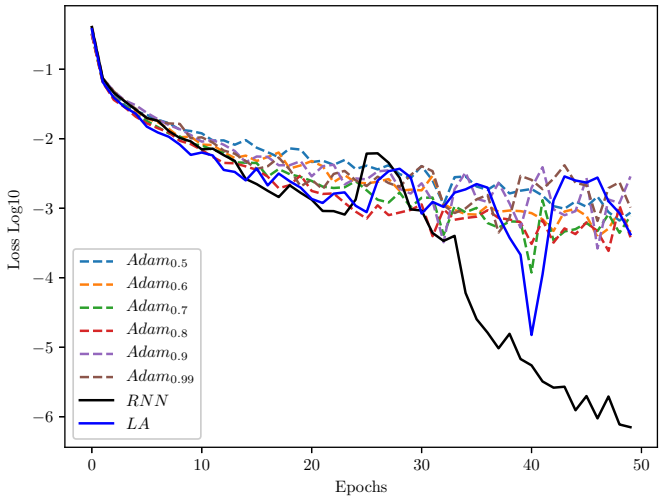


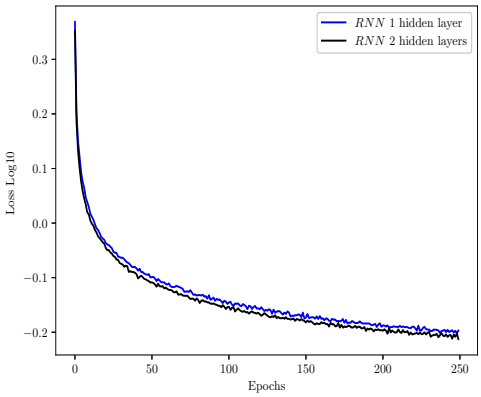
Figure 4.11: *Learned 2 layered RNN and LA against Adam running at the same timescales as the inputs for meta optimizers, evaluated on $MlpMnsit$. We see that the RNN shows the best performance here.*

4.4.3 Cifar10

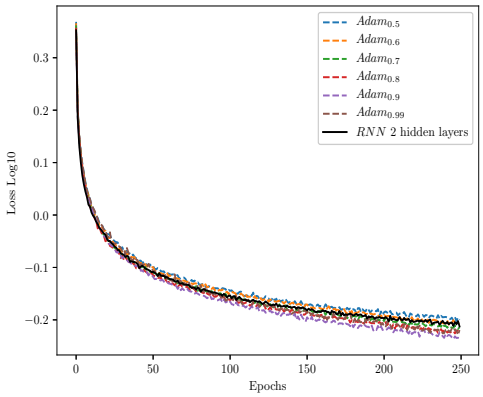
Training on Cifar10

For Cifar10 again we have two networks - a large one $ConvCifar_0$ and a small network $ConvCifar_1$. We trained two $RNNs$, one with a single hidden layer and one with 2 hidden layers, on $ConvCifar_1$ with a meta learning rate of $5e-4$. We used batch size 128 of $Cifar$ dataset for each meta iteration. Upon evaluation

on $ConvCifar_0$, RNN with two hidden layers performed better than the single layered one. However compared to the multiple input *Adam* timescales they both failed to achieve the achieve the best performance. Figure 4.12a shows their relative performance. We believe it might be due to simply too much noise which makes training $\mathbf{O}_m$ on *Cifar* hard. Figure 4.12b shows how RNN with two layers fares against *Adam*. Training directly on $ConvCifar_0$ did not yield any better results either.



(a) *Comparison of RNNs with 1 and 2 hidden layers on* $ConvCifar_0$.



(b) *2 layer RNN against multiple Adam timescales on* $ConvCifar_0$.

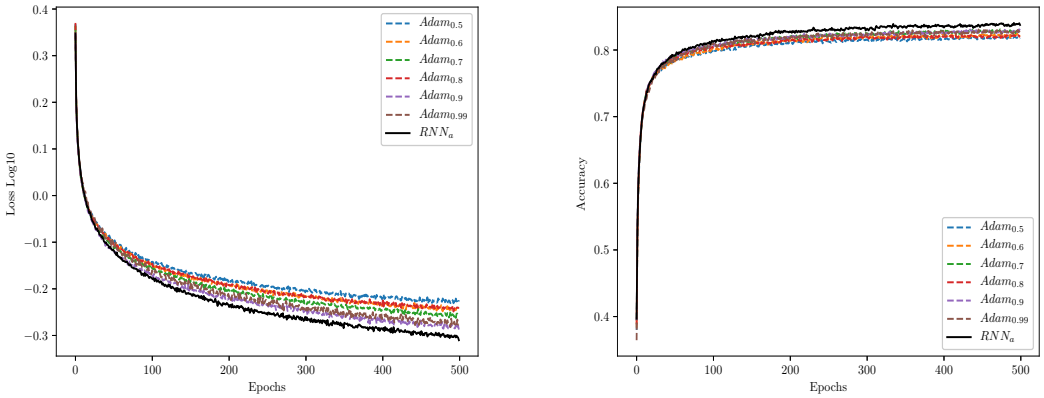
Figure 4.12: *Performance of RNN trained on* $ConvMnsit_1$ *and evaluated on* $ConvMnsit_0$. *We see that while the learned optimizer does show progress, it fails to achieve the same performance level as the top performing Adam timescale.*

Using pre trained meta optimizers

Next we decided to apply pre trained models on $ConvMnsit_0$ as a test to see how well they perform. Thus, next we took following two models already trained on $MlpMnsit$:

- Two layer RNN_a trained with learning rate $\eta = 0.01$.
- Two layer RNN_b trained with learning rate decay from $\eta = 0.01$ to $\eta = 0$ using cosine annealing.

First we tested RNN_a on the same setting where RNN trained on $ConvMnsit_1$ failed. i.e we set $\eta = 5e-4$ for all *Adam* timescales and RNN_a and apply it on $ConvMnsit_0$. We noted when applied on $ConvMnsit_0$, RNN_a outperforms all individual *Adam* timescales. As the evaluation progresses its lead grows stronger, until the evaluation was stopped after 500 epochs. Figure 4.13a shows



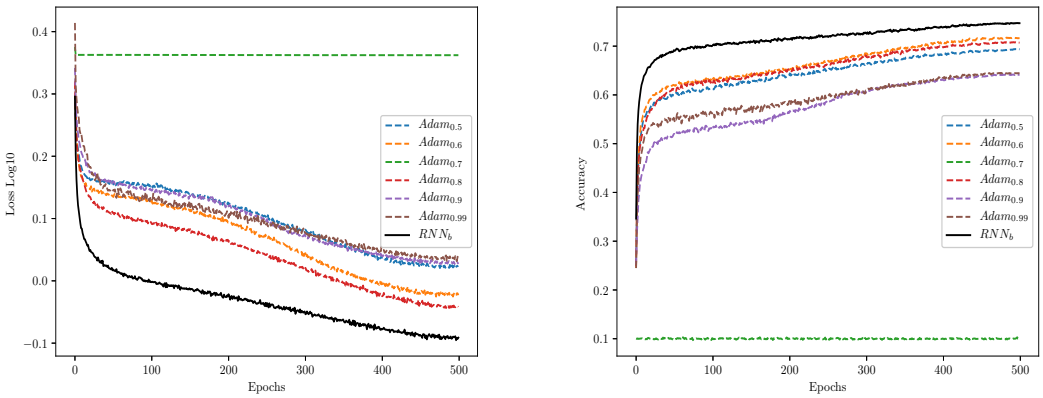
(a) *Loss on the training set.* **(b)** *Accuracy on test set.*

Figure 4.13: *Performance comparison of RNN_a and input Adam at different timescales on $ConvMnsit_0$ over 500 epochs, with $\eta = 5e-4$. The pre trained optimizer outperforms all of the individual Adam inputs.*

loss curves of RNN_a and individual *Adam* scales. Whereas 4.13b shows the accuracy achieved on test set by all the optimizers.

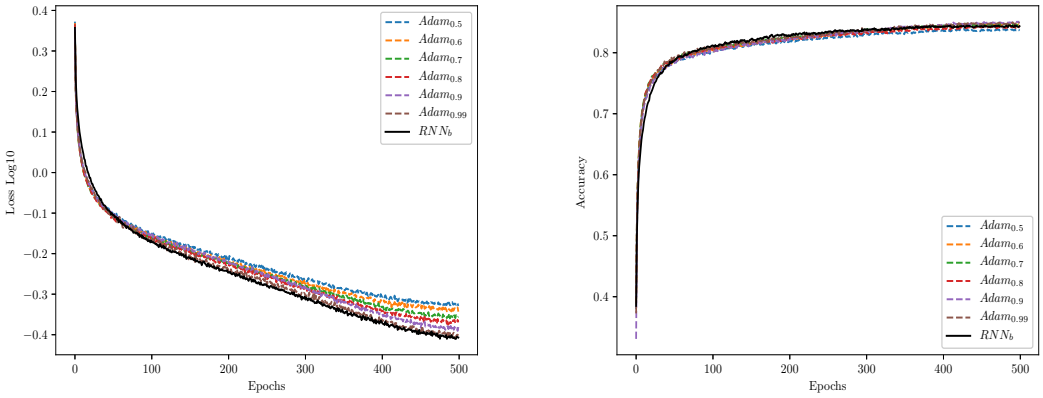
Afterwards we tested how the RNN_b trained with learning rate decay, performs when we evaluate it on $ConvMnsit_0$ while using the same learning rate schedule it was originally trained with on $MlpMnsit$. Again we ran the experiment for 500 epochs with a batch size of 128. We notice the learning rate schedule proves to be too harsh for all optimizers, as they do not achieve a good accuracy on test set. However, still the learned optimizer, i.e RNN_b outperforms its input *Adam* timescales by a huge margin, achieving the lowest loss curve and the highest accuracy. Also we notice that $Adam_{0.7}$ completely fails to perform with this learning rate schedule.

Finally we evaluate at another learning schedule where we observed *Adam* to perform the best. Now we decay the learning rate η from $5e-4$ to $5e-5$ over the course of 500 epochs using cosing annealing. We see that *Adam* performs much better compared to the last experiment with this schedule. Even so, RNN_b outperforms *Adam* by achieving the lowest loss curve. However when considering accuracy achieved on test set we notice that $Adam_{0.7}$ and $Adam_{0.99}$ perform marginally better, but since the main criteria for is the loss on the training set we still consider the meta optimizer to be the best performer in the current case. It is worth mentioning here that the learned optimizer was not trained with this learning rate schedule nor has it seen $ConvCifar_0$ or *Cifar10* before.



(a) *Loss on the training set.* **(b)** *Accuracy on the test set.*

Figure 4.14: *Performance comparison of RNN_b and input Adam at timescales on $ConvMnsit_0$ over 500 epochs, with learning rate decay from $\eta = 1e-2$ to $\eta = 0$ using consine annealing. The learned optimizer shows the best performance $ConvMnsit_0$ despite not having seen Cifar10 or $ConvMnsit_0$ before.*



(a) *Loss on the training set.* **(b)** *Accuracy on the test set.*

Figure 4.15: *Performance of comparison of RNN_b and input Adam timescales on $ConvMnsit_0$ over 500 epochs, with learning rate decay from $\eta = 5e-4$ to $\eta = 5e-5$ using consine annealing. The learned optimizer achieves the lowest loss curve through out the validation.*

4.5 Limitations

While we do notice very good training and validation results in most of the cases, for a few we were not able to achieve good results i.e. training on both cifar netwokrs $ConvCifar_0$ and $ConvCifar_1$. Although we suspect that noise might

be the cause, this is just a speculation at the moment. Furthermore even though optimizers trained on small mnist network $MlpMnsit$ performed very well on $ConvCifar_0$, this success of transferability might not always hold true. It could be that momentum time scales that worked well for $MlpMnsit$ might not be the best for some other random network thus leading to suboptimal performance on that network. One would then need to directly train on that network which can be very time consuming if the network is big. Moreover if that network refuses to lend itself for training easily, e.g. in the case of $Cifar10$, then even after training one would not obtain the best results.

4.6 Future Work

In the light of the limitations mentioned in the previous section, future work could include using models that allow for much longer partial unrolling that is they do not suffer from exploding and vanishing gradients, e.g. LSTM and GRU. One would then be able to smooth the loss over much longer time horizons.

Summary In this chapter we presented a new approach of training the meta optimizer $\mathbf{O}_m$. The approach uses *Adam* optimizer running at different timescales as inputs for meta optimizer, and the meta optimizer learns to output a weighted average of the inputs. We explored different models, starting from simple ones. We showed that the trained models tend to outperform the individual *Adam* timescales on the problems they were trained with. The learned optimizer also generalized well to larger networks that we experimented with, outperforming *Adam* timescales there as well.

Chapter 5

Conclusion

In this thesis we briefly discussed hand crafted optimizers and then state of the art in learning to optimize. We then presented two novel approaches for *learning to optimize*. First we presented "Learning to Optimize With Normalized Inputs". We started by giving the motivation from simple examples. We briefly looked at a handcrafted algorithm *MNG*, which formed a base for our further work. We saw that *MNG* performed very well on a sum of squares problem compared to *Adam*, but did not do so well on other problems. Afterwards we explored two major variants of "Learning to Optimize With Normalized Inputs", "k Timesteps" and "k Momentums". We trained our first learned optimizer using "k Timesteps". We saw that increasing k increased the performance of our trained optimizer. However just increasing the number of timesteps only provided diminishing returns in terms of improving the performance. Also considering the memory requirements one can not simply keep increasing the timesteps. Therefore we made a move towards using momentums instead i.e. "k Momentums". This allowed us to look much farther back in to the history while still keeping k small. We also made changes to our update equations to account for momentums. Both of these changes enhanced the performance of the learned optimizer by a great margin. We then compared the effects of using different momentum timescales. We trained and evaluated the learned models on $MlpMnsit$ and saw the learned models outperformed *Adam* and $RMSProp$ in evaluation runs. When applied on $ConvCifar_0$ both the learned models showed very competitive performance, outperforming $RMSProp$, despite not being trained on $ConvCifar_0$. At the end *Adam* took the lead. We also noted some drawbacks of our approach i.e. that it needs more memory then conventional handcrafted optimizers. Also when evaluating it is slower than handcrafted optimizers.

The second approach "Multiscale Adam" uses multiple *Adam* running at different timescales as inputs. It then learns to output a δx_i^t as a weighted mean on the inputs i.e *Adam* at different timescales. We started with a very simple model calculating only a linear weighted average. We saw that in some cases even something as simple as linear weighted average performs very well. There

we also that saw normalizing the weights showed better performance. Furthermore we noted that increasing k lead to better performance. However, for some cases the linear weighted average did not perform very well. We then explored further and found out smoothing the loss can be of great help. This gave us the motivation to switch to a RNN. The trained RNN enhanced the performance greatly. Upon switching to a 2 layered RNN we found out that it performed even better. Until now a small MLP for *mnist* had been our playground. Afterwards we moved towards a larger convolutional network for mnist. We first trained on a smaller convolutional network and then validated on a larger one. We saw that the 2 layered RNN out performed other optimizers by a significant margin, that displayed its ability to generalize. Moving onto *Cifar10*, we saw that training directly on *Cifar10* dataset with $ConvCifar_0$ and $ConvCifar_1$ did not yield the best results. But when we applied the pre trained models on $MlpMnsit$, to $ConvCifar_0$, we saw the pre trained optimizers outperformed the individual *Adam* inputs. While the approach showed promising results more experiments need to be done to see how it fares on other networks. Also we saw it failed to train on *Cifar10*, it might be that we need to smooth the loss more, which provides motivation to use more complex architectures.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[2] Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. *CoRR*, abs/1606.04474.

[3] Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555.

[4] Duchi, J., Hazan, E., and Singer, Y. (2010). Adaptive subgradient methods for online learning and stochastic optimization.

[5] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(9):1735–1780.

[6] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

[7] Li, K. and Malik, J. (2016). Learning to optimize. *CoRR*, abs/1606.01885.

[8] Loshchilov, I. and Hutter, F. (2016). SGDR: stochastic gradient descent with restarts. *CoRR*, abs/1608.03983.

[9] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). doklady ansssr (translated as soviet.math.docl.).

[10] Rosenbrock, H. H. (1960). An automatic method for finding the greatest or least value of a function. *The Computer Journal.*

[11] Schmidhuber, J. (1993). A neural network that embeds its own meta-levels. In *In Proc. of the International Conference on Neural Networks '93*. IEEE.

[12] Sinha, A., Sarkar, M., Mukherjee, A., and Krishnamurthy, B. (2017). Introspection: Accelerating neural network training by learning weight evolution. *CoRR*, abs/1704.04959.

[13] Thrun, S. (1996). Learning to learn: Introduction. In *In Learning To Learn*. Kluwer Academic Publishers.

[14] Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.

[15] 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. *CoRR*, abs/1703.04813.