An empirical study of melded gradient descent algorithms

Student Name: Robert Lewis
Supervisor Name: Louis Aslett
Submitted as part of the degree of M.Sc. MISCADA to the
Board of Examiners in the Department of Computer Science, Durham University

Abstract – Since first order algorithms such as stochastic gradient descent (SGD) have been used to optimiser neural networks, there have been many improvements to their convergence speeds, the most notable being the adaptive moment estimation (ADAM) algorithm. However, even today, no other algorithms have been able to improve upon SGDs ability to generalise to new data; from this, a trade-off arises where one must choose between the speed of convergence and generalisability. In recent years there has been a lot of effort to minimise this trade-off using melded methods, which transition from ADAM to SGD in training to utilise ADAMs speed of convergence and SGDs generalisability. However, there are limitations in the current melded optimisers, such as the transition method and when to transition in training. This project proposes a new melded optimiser called MAWS (Melding ADAM with SGD), which aims to improve on the limitations of current melded optimisers in minimising this trade-off. Also, due to the lack of understanding of SGD generalisability, we aim to provide some insights into this. MAWS minimises this trade-off through incorporating an extra phase in-between the ADAM and SGD updates, this extra phase allows for a continuous transition from ADAM to SGD. We evaluate MAWS alongside existing competitors on two different datasets with two different neural network architectures. Our results successfully demonstrate that MAWS achieves its aims in improving the minimisation of the trade-off, providing faster convergence and better generalisability than its competitors. We also provide details on how SGDs generalisability is still superior on one of the experiments due to its enhanced ability to escape regions of high gradients such as local minima and ravines. Finally, we provide limitations to our experiments and MAWS, such as its additional hyperparameters and provide directions for future work to improve on minimising this trade-off.

Keywords – Optimisation, Gradient descent, Generalisability, Speed of convergence, ADAM, SGD.

1. INTRODUCTION

Optimisation problems appear everywhere, even in your day-to-day life. For instance, what clothes to wear depending on the weather or deciding which way to travel home, all of which revolve around finding the best solution from all possible solutions. The optimisation problem for a neural network is a particularly difficult one. As the amount of data and complexity of neural networks increases, so does the importance of the optimisation algorithm. Today neural networks are able to capture and learn complex nonlinear relationships between billions of parameters. This leads to optimisation problems over highly complex loss surfaces that have many local minima, plateaus, saddle points and cliffs. Optimisation in this context is further complicated by the need to avoid overfitting or memorisation of the training data to ensure that the fitted model can generalise well to future unseen data.

The powerhouse that is behind the neural network is the first order gradient descent optimiser [1]. The optimisation algorithm is of extreme importance and dictates the time it takes to train a neural network, but it can also affect the performance. The time it takes to train a deep neural network is highly dependent on the architecture of the neural network and on the size of the data. For example, on the extreme end of training times, a deep convolutional neural network that Google trained on a 100 million labelled image dataset took 6 months to train [2]. This demonstrates how difficult and time consuming it is to train deep neural networks; as such, it is great motivation to research optimisers in the hope to reduce training time and improve performance. Although neural networks have superior performance on image and speech data, the time it takes to train a neural network is its biggest drawback. Being able to reduce the training time of a deep neural network would be of a great benefit. Faster training times would allow models to be updated through re-training more frequently, which would be of particular use in the industry. For example, a telecommunication company can be more accurate in predicting churn since its models can be more frequently updated. Increasing the performance of a neural network can be even more of a benefit. For example, better neural network performance in areas such as self-driving cars, advertisement and finance would result in increased revenue and sales. These are just some of the many benefits of improving optimisers for neural networks.

For neural networks, the optimisation problem revolves around minimising the loss function, for this project it was the cross-entropy loss $L\left(x^{(i)}; y^{(i)}; \theta_{k-1}\right)$ that was of relevance:

$$L\left(x^{(i)}; y^{(i)}; \theta_{k\text{-}1}\right) = -\sum_{j=1}^{C} y^{(i,j)} \log \hat{y}\left(x^{(i)}; \theta_{k\text{-}1}\right)^{(j)} \quad (1)$$

where C is the number of classes, $x^{(i)}$ is the i^{th} training example and is a vector of length d where d is the number of features. $y^{(i)}$ is a vector of length C and is a one hot encoding of the true label for $x^{(i)}$. Since we are using softmax, $\hat{y}(x^{(i)}; \theta_{k-1})$ is a vector of probabilities that the training example $x^{(i)}$ belongs to each class j given $x^{(i)}$, it is a vector is of length C. θ_k are the parameters of the neural network at iteration k, commonly known as the weights and biases and is in the form of a tensor.

As mentioned previously, in a neural network setting, one minimises this loss function through a first order method called gradient descent [1]. Gradient descents goal is to find a global or local minimum of a function, in this case the loss function. The idea is to take repeated steps in the opposite direction to the gradient of the loss function, this is the direction of steepest descent and as such we will head in a direction of lower cost. Eventually, we will land in a local minimum, meaning we will have minimised the loss function and found our set of parameters for the network. We have formalised Cauchys gradient descent method in [1] in the context of neural networks:

$$L(\theta_{k-1}; X; Y) = \frac{\sum_{i=1}^{N} L\left(x^{(i)}; y^{(i)}; \theta_{k-1}\right)}{N} \qquad (2)$$

$$\theta_k = \theta_{k-1} - \alpha \cdot \nabla_\theta L(\theta_{k-1}; X; Y) \qquad (3)$$

where α is the learning rate that controls the size of the step, N is the total number of training examples. X and Y are all the examples and labels of the dataset. $L(\theta_{k-1}; X; Y)$ is the average of the total loss for all training examples and $\nabla_\theta L(\theta_{k-1}; X; Y)$ is the average gradient of the total loss w.r.t. the parameters. In section 2, we talk extensively about how gradient descent is performed for several different optimisers. We have purposely left out much of the details around training the neural network, such as the forward and backward propagation stages; see [3] if the reader needs to refresh their memory on the topic.

The difficulty of minimising the loss function comes from two main areas, the first is simply the number of parameters that need updating and thus differentiating. The second is the number of updates that need to be made to each parameter, each problem stems from the architecture of neural networks and cannot be avoided. The first problem is due to the need of having many parameters to capture the complex relationships in the data. The second stems from the complex parameter space as a result of the numerous amounts of parameters. In fact, neural networks have highly non-convex loss surfaces that have many local minima, plateaus, saddle points and cliffs. In which the first order methods cannot distinguish between the different critical points, making it difficult to navigate the parameter space and converge to a good local minimum. So why do we not use second order methods such as the Newtons method [4]? There are two main reasons; the first is due to them being extremely costly in terms of computation as compared to gradient descent, therefore increasing the already long time to train a neural network. The second main reason is due to saddle points being numerous in the parameter space, this is an issue since second order methods like Newtons have a tendency to be attracted to saddle points [5]. These saddle points are not where the optimal parameters for the network are located, which leads to poor performance of the network. As a result, first order methods are employed.

Today, there are various types of optimisers for neural networks. The most recent kind is what we call melded optimisers, they combine two different optimisers together in the hope to exploit the benefits of each. The two optimisers that are usually melded are stochastic gradient descent (SGD) [6] and adaptive moment estimation (ADAM) [7], a detailed discussion about these optimisers is made in section 2. SGD is utilised for its superior generalisability and ADAM is utilised for its speed of convergence. Generalisability in this context is defined as the performance of the neural network on unseen or test data; more specifically, in this paper we define it in terms of the predictive accuracy on our test data. As such generalisability is concerned with finding a good set of parameters at the end of training for the network to generalise well to new data. The speed of convergence is more loosely defined as the number of epochs it takes to train a neural network such that the accuracy or error rate is no longer changing. An epoch is the term to indicate the number of passes of the entire training data the neural network has processed. Without melded methods, one would have to choose between SGD and ADAM; from this, a trade-off arises as one must decide between the speed of convergence and generalisability.

Melded methods combine or switch from one method to the other while training, in order to reap the benefits of each optimiser and minimise the trade-off. This is the topic of this project, to investigate plain, adaptive and melded gradient descent methods and to propose a new melded method called MAWS (Melding ADAM with SGD), with the aim to improve upon current melded methods with the goal of minimising the trade-off. That is, we want MAWS to be as fast as ADAM but have SGDs generalisability. With this, we hope the MAWS optimiser eliminates the need to pick between SGD or ADAM. The last aim of this project revolves around generalisability and its lack of current understanding, we hope to propose possible insights into why SGD is superior in this area and how we can utilise it in a melded method.

Lastly, in this project several experiments are conducted. In section 2, we introduce in detail a variety of optimisers ranging from the standard optimiser to the most recent adaptive and melded optimisers. We provide advantages and disadvantages to each and discuss critical challenges all optimisers face. In section 3, we experiment on the discussed optimisers on simple toy problems, providing better intuition as to how these optimisers work through visualising the loss surface. In section 4, we propose our solution, which is the MAWS optimiser. We thoroughly detail and articulate each part of the algorithm and how it improves on the limitations of current melded optimisers. In Section 5, the results and evaluation are discussed. We compare our proposed optimiser to that of recent and popular optimisers on two datasets to help evaluate the performance of MAWS. The two datasets are MNIST [8] and CIFAR10[9], which are commonly used as benchmarks in the optimisation research field. The architecture used for MNIST is a simple one layer feedforward neural network, and for CIFAR10 a convolutional neural network called LeNet-5 is used [10]. More details are provided on the datasets and architectures in section 5. The results demonstrate that we have achieved the aim of reducing the trade-off with MAWS through adding the extra mixing phase. We also attempt to understand why SGD generalisability is superior in one of our experiments, and it turns out that it is in part due to its enhanced ability to escape regions of high gradients such as local minima and ravines. Additionally, in Section 5 and 6, we conclude with an overall analysis of the project, providing limitations and directions for future work.

2. Related Work

When applying a neural network to a problem, one of the most important choices that can affect the performance is the optimisation algorithm. In recent years there has been a plethora of adaptive gradient optimisers, making it difficult to know which one to use. In this section, we review a variety of optimisers: adaptive and non-adaptive, to gain a better understanding of the advantages and disadvantages of each. A summary of each optimiser can be found at the end of this section in table 1. If greater detail is needed, please look at the respective paper of the optimiser.

A. Standard Optimisers

There are three common approaches to standard gradient descent that can be used to train neural networks, each one differing by how much data is handled through the network at any one time. The differing amount of data used also causes a trade-off between the time taken for each update and the accuracy of each update, as well as some nuances.

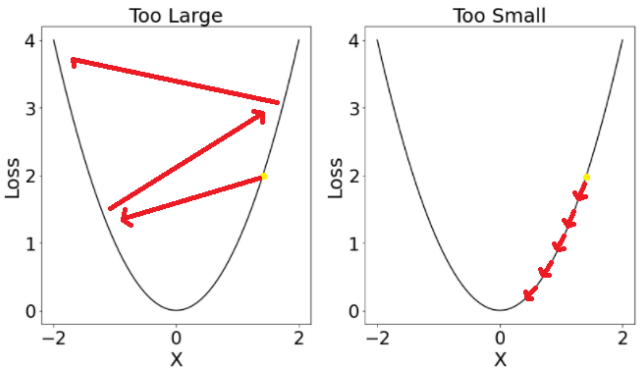

Figure 1: Illustration of the effects of when a too large and a too small of a learning rate is used.

The following subsection describes each optimiser and discusses the trade-offs between each in a deep neural network setting.

The first approach is called batch gradient descent (BGD), where the whole dataset is used to compute the gradient of the parameters w.r.t. the cost function [1]. In fact, we covered this approach in the introduction, see equation 3. As mentioned previously, α is the learning rate that determines how large of an update is performed. This is the most important hyperparameter to tune for all optimisers. If α is too small, the number of epochs required for convergence will increase because of small updates in the parameters. On the other hand, if α is too large, the parameters will rapidly change which can lead to divergent behaviours. One can see an illustration of these effects in figure 1. A large parameter update can lead to θ_k being in a sub optimal region in the parameter space where there is a high cost, these issues are discussed in section 2.E. Please note that we refer to weights and biases as the parameters of the neural network throughout this paper.

BGD can lead to more stable convergence for θ_k as it computes the exact gradient for each update. In a convex setting, BGD is guaranteed to converge to the global minimum [11], but in a non-convex setting such as a neural networks loss surface, this optimiser can only guarantee to converge to a local minimum [11]. Unfortunately, due to its stable updates it often finds itself converging to poor local minima in a neural network setting [12], it lacks the ability to escape these minima, unlike stochastic optimisers. Using BGD, one can utilise vectorisation to speed up computation. However, BGD requires a large amount of memory and due to big datasets not being able to fit in memory this vectorisation is much less efficient. As a result, this optimiser is slower at learning compared to other optimisers. Lastly, as BGD uses the whole dataset, it means that unnecessary computations are made as it recomputes gradients for very similar examples.

The second standard optimiser is called stochastic gradient descent (SGD), where a single

training example is used to compute the gradient estimate of the parameters w.r.t. the cost function [6]:

$$\theta_k = \theta_{k-1} - \alpha \cdot \nabla_\theta L\left(\theta_{k-1}; x^{(i)}; y^{(i)}\right) \quad (4)$$

The difference between equation 4 and equation 3 is that equation 4 only uses a single training example and label, whereas equation 3 uses the entire dataset. Note that when we state gradient estimate, this is in fact a Monte Carlo estimate of the gradient. This is because SGD does not have access to the whole dataset. Due to using the estimate of the gradient, we improve the convergence speed but lose the stability of convergence when compared to BGD. This is due to having faster computations as it uses only one example for each update. Inherently this gradient estimate introduces stochastic behaviour in its updates, which can be a useful property that can lead it to escape poor local minima. Local minima will be discussed in more detail in section 2.E. Some issues with SGD are that its estimate of the gradient can be inadequate which leads to updates for θ_k in poor directions in the parameter space. Also, due to its noisy updates, SGD will often only converge around the minimum. The latter is due to SGD never having the exact gradient; as such, it cannot calculate where the exact true local minimum is, leading to stochastic oscillations in the vicinity of the local minimum.

The last standard optimiser is able to build upon the latter methods by combining them, this optimiser is called mini batch gradient descent (MBGD). We start by splitting the dataset into small batches, with this the batch is processed through the network, which computes the gradient estimate of each example w.r.t. the cost function in the batch and utilises the average to update the parameters [12]:

$$\nabla_\theta L(\theta_{k-1}; \mathcal{M}_k) = \frac{\sum_{i \in \mathcal{M}_k} \nabla_\theta L\left(\theta_{k-1}; x^{(i)}; y^{(i)}\right)}{|\mathcal{M}_k|} \quad (5)$$

$$\theta_k = \theta_{k-1} - \alpha \cdot \nabla_\theta L(\theta_{k-1}; \mathcal{M}_k) \quad (6)$$

where $\mathcal{M}_k$ is a set of randomly sampled numbers from {1, . . , N} without replacement and is of size M for all k, where M=$|\mathcal{M}_k|$ and is called the batch size. $\nabla_\theta L(\theta_{k-1}; \mathcal{M}_k)$ is the gradient estimate of the loss function w.r.t. the parameters over the batch of data.

MBGD lies in between the two former optimisers. It computes a more accurate gradient estimate as compared to SGD through averaging over a mini batch. Like BGD, it is also able to utilise vectorisations to speed up computations. Although we have a more accurate gradient estimate, we still have some stochastic behaviour in the updates of θ_k allowing it a chance to escape poor local minima. Expanding on this gradient estimate, MBGD has a more accurate estimate of the true loss and gradient than SGD. Therefore, from one update to the next, the loss surface MBGD 'sees' changes much less compared to SGD. Therefore, MBGD has less noise in its updates, which reduces the number of updates in poor directions in the parameter space, leading to faster convergence speeds. From this, we can see MBGD is an extremely useful optimiser when training a neural network and is arguably one of the most used optimisers even today. In the community, it is often referred to as SGD; from here on in the paper, we will adopt the same notation for simplicity.

Unfortunately, there are challenges with the optimisers above that must be overcome to train a neural network efficiently. Firstly, the stochastic behaviour of the latter two optimisers introduces two issues. The first is navigating down areas of steeper curvature and plateaus, the solution to this is discussed in the next subsection. The second issue is with converging to the exact local minimum [11]. Due to the noise in the updates of θ_k these optimisers oscillate around the minimum; however, as we discuss in section 2.E, this may not be an issue at all. But this issue is solved by using a learning rate scheduler during training, it is a predefined method that reduces the learning rate at a specific epoch. This allows for a smoother convergence to the minimum and greatly reduces the noisy updates of θ_k around the local minimum. Unfortunately, this introduces additional hyperparameters of selecting the correct threshold and the correct reduction in the learning rate, thus requiring more time and computations to tune these hyperparameters. Thirdly, as mentioned above, choosing the correct learning rate is very important for efficient and accurate training, thus it is the most important hyperparameter to tune. As a result, a range of values must be tried, increasing the time to train the neural network. Lastly, the learning rate is fixed between all parameters and therefore cannot capture nor incorporate the varying gradients between parameters. This denies us the ability to perform varying updates for parameters, to incorporate this, one must use an adaptive optimiser. The fixed learning rate can slow down convergence and training as more updates are required to navigate the parameter space for the parameters with smaller gradients.

B. Improving the Standard Optimisers

There have been several algorithms developed to combat some of the issues mentioned above; however, we only focus on one for this subsection. Firstly, local minima can often be found at the bottom of a well where surface curvature is much higher [13]. These types of regions in the loss surface often have an ill-conditioned hessian matrix. A hessian matrix is simply a square matrix of second order partial derivatives w.r.t. the loss function, and it describes the curvature of the function. It being ill-conditioned simply means that it is almost singular.

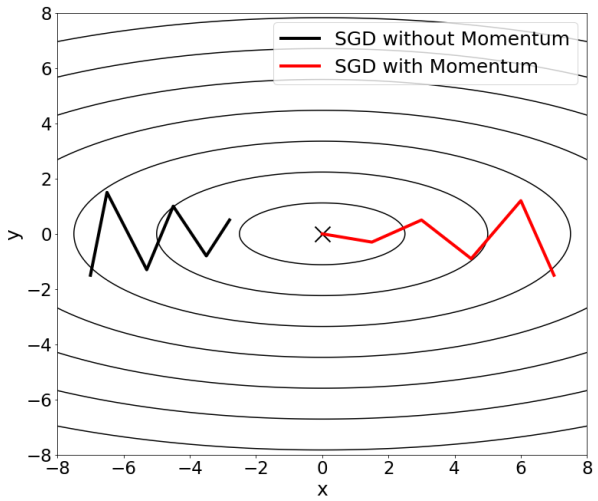

Figure 2: Illustration of the effects of momentum on a toy contour plot.

Although the hessian is not used nor calculated in first order methods, entering a region where the hessian is ill-conditioned and almost singular can result in increased oscillations and divergent behaviours in the learning rate. We can calculate a metric called the condition number, which is the ratio of the largest singular value to the smallest singular value. When the condition number is very large it means the hessian is ill-conditioned; as such, some parameters have very large curvature while some have very small curvature, leading to increased oscillations and divergent behaviours in the learning rate. Looking at the black line in figure 2, we see the consequences of ill-conditioning, resulting in slower progress in navigating the steep slopes. Additionally, areas of small gradients known as plateaus require many updates to navigate over, which again results in slow progress for convergence. To combat ill-conditioned hessians and the latter issue, a method called momentum was devised [14].

This method allows SGD to accelerate in the relevant direction in the parameter space and thus reduces oscillations in unwanted directions. Looking at figure 2, we can see these effects, resulting in faster convergence to the minimum. It is able to do this by adding a fraction ρ of the update vector used in the previous step to the current vector:

$$a_k = \rho \cdot a_{k-1} + (1-\rho) \cdot \nabla_\theta L(\theta_{k-1}; \mathcal{M}_k) \quad (7)$$

$$\theta_k = \theta_{k-1} - \alpha \cdot a_k \quad (8)$$

where a_k is initialised at zero and is an exponentially decaying average of the gradient, the fraction ρ has a default value of 0.9 and controls how much momentum is incorporated into the updates of θ_k. Note that SGD with momentum is denoted as SGDM throughout this paper. Intuitively, one can imagine a ball rolling down a hill, the velocity a_k captures the direction and speed at which the parameters move through the space. Therefore, as the ball moves down the hill, it becomes faster and faster with increasing momentum. This allows for faster navigation of the parameter space, which reduces the time of convergence. The same happens in the parameter space, looking at figure 2, a_k increases for dimensions where gradients are in the same direction, in this case the y direction. But also a_k decreases for dimensions whose gradients change directions, in this case the x direction. The latter is what reduces the oscillations in unwanted directions. It has been empirically shown that when momentum is used with SGD, the speed of convergence is faster, and the oscillations are reduced [14].

C. Adaptive Optimisers

The previous optimisers mentioned have all kept the learning rate fixed between parameters, as mentioned in section 2.A, this is not ideal when the size of the gradients of parameters vary. When a parameter θ_k has a small gradient and a fixed learning rate is used, the updates are very small. Thus, increasing the number of updates needed to find a suitable value for that parameter. Adaptive optimisers allow us to have a differing learning rate for each parameter. The most popular adaptive optimiser today is ADAM (adaptive moment estimation) [7], it computes individual learning rates for each parameter in the network from estimates of the first (mean) and second (uncentered variance) moments of the estimated gradients, m_k and v_k respectively. Firstly, the stochastic gradient estimate g_{k-1} for the mini batch is computed using:

$$g_{k-1} = \nabla_\theta L(\theta_{k-1}; \mathcal{M}_k) \quad (9)$$

This is the same gradient estimate as MBGD seen in section 2.A. After the gradient is computed, the first and second moments variables are updated, these are exponential decaying averages:

$$m_k = \beta_1 \cdot m_{k-1} + (1-\beta_1) \cdot g_{k-1} \quad (10)$$

$$v_k = \beta_2 \cdot v_{k-1} + (1-\beta_2) \cdot g_{k-1} \odot g_{k-1} \quad (11)$$

where β_1 and β_2 control the decay rates, these have default values of 0.9 and 0.999 respectively. $\odot$ represents element-wise matrix-matrix multiplication. As $m_k$and v_k are initialised to zero, they have an implicit bias towards zero, particularly at the beginning of training. To correct for this bias, we perform the following calculation:

$$\overline{m_k} = \frac{m_k}{1-\beta_1^{\mathbf{k}}} \qquad \overline{v_k} = \frac{v_k}{1-\beta_2^{\mathbf{k}}} \quad (12)$$

where the bold superscript **k** on $\beta_1^{\mathbf{k}}$ and $\beta_2^{\mathbf{k}}$ denotes the power instead of the iteration, $\overline{m_k}$ and $\overline{v_k}$ are the unbiased first and second moments of the estimated gradients. Finally, this is used to update the parameters:

$$\Delta_k^{ADAM} = \frac{\overline{m_k}}{\sqrt{\overline{v_k}} + \varepsilon} \quad (13)$$

$$\theta_k = \theta_{k-1} - \alpha \cdot \Delta_k^{ADAM} \quad (14)$$

where ε is 10^{-6} to avoid a division by zero.

Focusing on the first and second moments, the first moment m_k behaves similarly to the momentum discussed in section 2.B by incorporating some of the behaviour of past steps into our current update. This accelerates updates of θ_k in the relevant directions and reduces noise. The second moment v_k is the parameter that allows ADAM to provide different learning rates for different parameters. Intuitively, we want to use smaller updates for parameters with large gradients since it is easier for the update to cause too large of a change in our parameters, leaving them in a non-optimal region of the parameter space. Non-optimal regions of the parameter space are discussed in more detail in section 2.E. Thus, for parameters with small gradients, the update size will not be of significant size to see changes in the parameters θ_k. Therefore, we want a larger learning rate allowing us to make significant updates to our parameters θ_k.

ADAM is able to capture the size of the gradient associated with each parameter through the second moment estimate. For example, $\overline{v_k}$ will be small for parameters with smaller gradients due to the $g_{k-1} \odot g_{k-1}$ term, resulting in a larger update as $\overline{v_k}$ is in the denominator in equation 13. ADAM incorporated this idea from a popular adaptive optimiser called AdaGrad [15]. This is the biggest advantage adaptive optimisers have over SGD. As $\overline{v_k}$ is the uncentered variance we can interpret $\overline{v_k}$ as our uncertainty, for example, a large $\overline{v_k}$ means there is greater uncertainty around whether the direction of the gradient $\overline{m_k}$ is the direction of the true gradient; as such, we take smaller updates in parameters θ_k.

ADAM also incorporated a useful technique of using only a 'short-term' history of the past gradients, it does this through using exponential decaying averages. ADAM integrated this idea from a popular adaptive optimiser called RMSprop [12]. This is a small but powerful technique, as it ensures that the learning rates do not diminish rapidly. If we were to use the whole history (accumulating all past squared gradients), it would lead to aggressively diminishing learning rates. In fact, Adagrad does just this, and its diminishing learning rates turned out to be its main weakness [12]. Other advantages of using ADAM are its simple implementation, little hyperparameter tuning requirement due to its adaptive learning rates and its speed of convergence. The latter we speak in detail about in the next subsection. As a result, this makes ADAM one of the most widely used methods for optimising neural networks. The authors and many subsequent papers empirically show ADAM outperforming other adaptive optimisers and even SGD [7], with ADAMs superior convergence times being its main attraction.

D. Melded Methods

Before discussing the latest melded optimisers, we need to understand the difference between SGD and ADAM and why neither optimiser is completely satisfactory. There are two key metrics that separate SGD and ADAM, generalisability and the speed of convergence. As mentioned previously, we define generalisability as the predictive accuracy on our test data, it is concerned with finding a good set of parameters at the end of training for the network to generalise well to new data. It is often that adaptive optimisers produce better training performance than SGD, however, they fail to maintain this performance when working with test data [16]. This is because adaptive optimisers converge to sets of very low-cost parameters at the end of training, such that they overfit the data and thus generalise poorly to new data. Although adaptive optimisers have been observed to generalise poorly compared to SGD, the speed of convergence for adaptive methods is much faster. Therefore, one must choose between the two, resulting in a trade-off. This trade-off is still not fully understood, however, there have been several papers researching it [15, 16, 17]. This will be an important topic that is discussed in section 5.

Recently there has been a lot of effort in bridging the gap between generalisation and convergence speeds. In this paper, we will propose a new algorithm that helps minimise the trade-off between the two. See section 4 for our proposed optimiser. Firstly, we will discuss a recent melded optimiser called SWATS [17]. It is a strategy that simply switches from ADAM to SGD once a condition is satisfied. It aims to utilise the speed of convergence of ADAM at the beginning of training and SGDs generalisability at the end of training, in order to find a good set of parameters for the network. In [17], the authors show SWATS outperforming SGD and ADAM in several tasks, empirically demonstrating that the trade-off has been minimised. There are two problems one must overcome to develop a melded optimiser, firstly determining when to switch and secondly what learning rate to use after the switch. SWATS overcomes the latter issue through proposing the following subproblem to find the learning rate γ_k for use in SGD updates if a switch occurs:

$$proj_{-\gamma_k g_{k-1}} p_k = \frac{(p_k) \cdot (-\gamma_k g_{k-1})}{p_k \cdot p_k} p_k = \alpha \cdot \Delta_k^{ADAM} = p_k \quad (15)$$

where p_k is an ADAM update, $proj_{-\gamma_k g_{k-1}} p_k$ denotes the orthogonal projection of $-\gamma_k g_{k-1}$ onto p_k. SWATS solution to this problem is the following:

$$\gamma_k = \frac{p_k^T p_k}{-p_k^T g_{k-1}} \quad (16)$$

As stated in the paper, this can be interpreted as the scaling necessary for the stochastic gradient g_{k-1} that leads to its projection on the ADAM step p_k to be p_k itself. For more clarity, look at figure 3 illustrating the solution. γ_k will be a noisy estimate of the scaling required, as such an exponential average initialised at zero is computed:

$$\lambda_k = \beta_2 \lambda_{k-1} + (1 - \beta_2)\gamma_k \qquad (17)$$

where λ_k is initialised at zero and is an exponentially decaying average of γ_k. A very simple switching point is also proposed in the paper, which provides a solution to the problem of when to switch:

$$|\Lambda - \gamma_k| < t \quad where\ \Lambda = \frac{\lambda_k}{1 - \beta_2^{\boldsymbol{k}}} \qquad (18)$$

where again the bold superscript **k** on $\beta_2^{\boldsymbol{k}}$ denotes the power instead of the iteration, t has a default value of 10^{-6} and this condition is checked at every iteration. It compares Λ and the current value of γ_k. Due to the zero initialisation of λ_k, a bias correction is used giving us Λ.

We can interpret this switching point as when the Λ and γ_k approach each other and become equal, it means that the different learning rates are no longer changing for each parameter. At this point, we have utilised ADAMs speed of convergence and adaptive learning rates, as such, we switch to SGD to benefit from its generalisability. Once the condition is satisfied, the algorithm switches to SGD and uses Λ as its learning rate. The switch usually occurs in the first half of the total number of epochs, this is simply due to its design. In [17], they demonstrate that switching too early increases generalisability but reduces convergence speed and switching too late reduces generalisability and increases convergence speed. Inherently, there is still somewhat of a trade-off, although minimised. Unfortunately, there does not seem to be a way to completely avoid this trade-off. SWATS makes sure not to switch too early with the denominator of 1-$\beta_2^{\boldsymbol{k}}$. At the beginning, the denominator is very small, which ensures we do not satisfy the condition. As epochs increase, the denominator increases to 1, and at this point we are interested in if Λ and γ_k are close and hence will switch if the condition is satisfied. We discuss SWATS switching point in greater detail in section 5. Please note for all melded methods, we can incorporate momentum into the stochastic gradient g_{k-1} by replacing it with a_k in equation 7.

To conclude the SWATS algorithm, firstly, we start with ADAM to update parameters θ_k. Here we still need a learning rate α for ADAM which must be tuned. Using ADAM at the start of training we are able to utilise its adaptive learning rates, allowing parameters to be more efficiently updated as compared to SGD. The idea is to then switch to SGD at the correct time when the criterion in equation 18 is met, with this, we use the calculated learning rate Λ for the SGD updates. Note that this is not a hyperparameter; thus, no tuning is needed, this is why this method is so useful. When we switch to SGD, we utilise its generalisability at the end of training to find a good set of final parameters for the network.

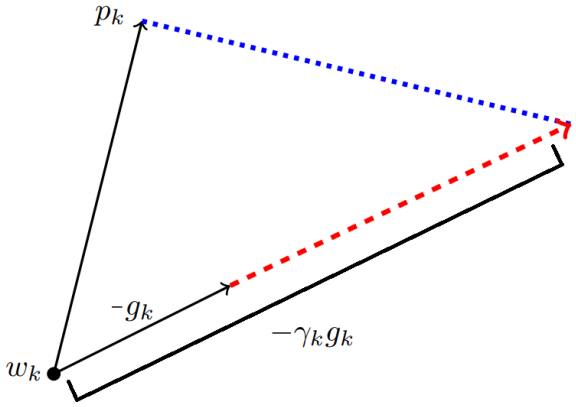

Figure 3: The biased learning rate γ_k for SGD after the transition, calculated from equation 16. given parameters θ_k, a stochastic gradient g_{k-1} and an ADAM step p_k.

SWATS is not the only melded optimiser that tries to combat this trade-off, we will briefly discuss two other melded optimisers to which we incorporate some of their ideas into MAWS. We omit some details around the following optimisers as they are less relevant to this paper and our proposed optimiser. In [18], the authors investigate a combined optimiser called MAS where each update of θ_k combines an ADAM update Δ_k^{ADAM} and learning rate α_{ADAM} and an SGD update g_{k-1} and learning rate α_{SGD} through a linear weighted sum:

$$\omega = 0.5 \cdot (\alpha_{ADAM} + \alpha_{SGD}) \qquad (19)$$

$$\theta_k = \theta_{k-1} + \omega \cdot 0.5 \cdot \left(\Delta_k^{ADAM} + g_{k-1}\right) \qquad (20)$$

where ω is an average of both optimisers learning rates; unfortunately, each learning rate needs tuning.

In this paper, MAS outperforms ADAM and SGD in a number of problems. The intuition behind MAS is that if ADAMs and SGDs update vectors are in a similar direction, then their combination will cause a large, boosted update in that direction. However, when they disagree, the combination of their updates will result in a smaller update in the parameters. Although a very simple yet powerful optimiser, it fails to provide an intuitive understanding of why it is beneficial to incorporate both optimisers at each step in training to combat the trade-off.

A recent optimiser that gained a lot of attention is AdaBound [19]. In this paper, they propose an optimiser that utilises dynamic bounds on learning rates to achieve a gradual and smooth transition from ADAM to SGD. It being similar to

SWATS demonstrates a common intuition of switching from ADAM to SGD. These bounds (upper and lower) on the learning rate are employed through gradient clipping, this is when we set a boundary for the gradient norms to lie within. These bounds for clipping are continuously restricted for each iteration, both bounds converge smoothly to a constant final learning rate. This results in the optimiser behaving like ADAM at the beginning of training and with each iteration behaving more and more like SGD as these bounds are restricted. Note that there is an additional hyperparameter that is the constant final learning rate. Again, in this paper Adabound is empirically shown to outperform ADAM and SGD on several tasks. As such, there is clear evidence of improved generalisability and speed of convergence.

E. Challenges for All Optimisers

Having discussed some of the most recent melded optimisers, it is important to understand the problems every single optimiser faces. There are three key challenges as described by Ian Goodfellow in [20]. In this subsection, we will explore each challenge in a bit more detail.

I. Local Minima

As mentioned several times, finding a good local minimum in the parameter space is a difficult task for non-convex problems. Surfaces of the space vary from problem to problem making it difficult for optimisers to be robust to all landscapes. Today, getting stuck at a local minimum at the beginning or middle of training is much less of a concern. It has been revealed that local minima are in more

Optimiser	Additional Variables	Transition Method	The update
SGD see [6]	-	-	$\alpha \cdot g_{k-1}$
Momentum see [14]	$a_k = \rho \cdot a_{k-1} + (1-\rho) \cdot g_{k-1}$	-	$\alpha \cdot a_k$
ADAM see [7]	$m_k = \beta_1 \cdot m_{k-1} + (1-\beta_1) \cdot g_{k-1}$ $\overline{m_k} = \frac{m_k}{1-\beta_1^k}$ $v_k = \beta_2 \cdot v_{k-1} + (1-\beta_2) \cdot g_{k-1} \odot g_{k-1}$ $\overline{v_k} = \frac{v_k}{1-\beta_2^k}$	-	$\alpha \cdot \frac{\overline{m_k}}{\sqrt{\overline{v_k}} + \varepsilon}$
SWATS see [17]	$p_k = \alpha \cdot \frac{\overline{m_k}}{\sqrt{\overline{v_k}} + \varepsilon}$ $\gamma_k = \frac{p_k^T p_k}{-p_k^T g_{k-1}}$ $\lambda_k = \beta_2 \lambda_{k-1} + (1-\beta_2)\gamma_k$ $\Lambda = \frac{\lambda_k}{1-\beta_2^k}$	Switch to SGD phase if: $\lvert \Lambda - \gamma_k \rvert < t$	ADAM phase: $\alpha \cdot \frac{\overline{m_k}}{\sqrt{\overline{v_k}} + \varepsilon}$ SGD phase: $\Lambda \cdot g_{k-1}$
MAWS (See section 4)	$\mu_k^A = \begin{cases} 1, & k < \eta_A \cdot S \\ 1 - \frac{k - \eta_A \cdot S}{\eta_M \cdot S}, & \eta_A \cdot S \le k \le (\eta_M + \eta_A) \cdot S \\ 0, & k > (\eta_M + \eta_A) \cdot S \end{cases}$ $\mu_k^S = 1 - \mu_k^A$ $\alpha^{Melded} = \mu_k^A \cdot \alpha + \mu_k^S \cdot \Lambda$ $\Delta_k^{Melded} = \mu_k^A \cdot \frac{\overline{m_k}}{\sqrt{\overline{v_k}} + \varepsilon} + \mu_k^S \cdot g_{k-1}$	Length of respective phases: $\eta_A = E \cdot \varphi_{Adam}$ $\eta_M = E \cdot \varphi_{MIX}$ $\eta_S = E - \eta_A - \eta_M$	ADAM phase: $\alpha \cdot \frac{\overline{m_k}}{\sqrt{\overline{v_k}} + \varepsilon}$ Mix phase: $\alpha^{Melded} \cdot \Delta_k^{Melded}$ SGD phase: $\Lambda \cdot g_{k-1}$

Table 1: Summary table of the optimisers covered. Our optimiser MAWS is proposed in section 4.

abundance at low costs, also in terms of the performance of the neural network, most local minima are equivalent to each other and even to the global minimum [20]. As a result, it is very unlikely to come across a local minimum in training; if one does get stuck in a local minimum, it will be at low costs. Therefore, it will be equivalent to many other local minima and will likely result in a good set of parameters for the network. In fact, research has shown that more times than not, optimisers for most neural networks will not get stuck at a critical point [20], due to the stochastic nature of all optimisers. This fact is not a concern as the goal is not necessarily to find a local minimum, just a good set of parameters for the network. There have been indications of local minima being the reason for a loss in generalisability [13, 16]. The equivalence of local minima stems from no identifiability as discussed in [20], that is, there are many sets of parameters for the network that result in a good performance for the network. There is a simple test proposed in [20] to check if local minima are the problem for a network, one plots the gradient norm over time. If the gradient norm shrinks to an insignificant size, the problem is local minima or critical points.

II. Plateaus, Saddle Points and Other Flat Regions

For high dimensional non-convex functions, local minima are rare, and saddle points are more common; in fact, the ratio of saddle points to local minima grows exponentially with dimensions [5]. Although saddle points are more likely to be found at a higher cost, which was also shown in [5], there are still possible concerns. As saddle points can have a large plateau of the same loss, it makes it difficult for optimisers to escape from due to near zero gradients. However, empirically gradient descent algorithms have been able to escape from these areas. Generally, large plateaus are much more of a concern; since the gradients are all near zero, little progress can be made in the parameter space. If these areas are in a high-cost region, it will cause parameters to be poorly tuned and result in poor performances by the network. Unfortunately, there is not a perfect solution, the best solution to escape a flat region is with the use of high momentum, however, too much momentum can cause issues later on in training.

III. Cliffs, Exploding Gradients and Non-Optimal Regions

Cliffs and exploding gradients are problematic if not identified. These are regions with extremely steep gradients and are a result of large weights being multiplied together, most commonly seen in recurrent neural networks as multiplication of many weights are performed for each step. When close to a cliff, an update in θ_k can move parameters far from their previous position to a non-optimal region in the parameter space. Cliffs can be avoided through gradient clipping, which prevents too large of a parameter update to be taken by clipping the update size. Non-optimal regions refer to areas of high cost or poor local regions; that is, the local descent will not point us to further regions of lower loss, due to the location the parameters are in the space. Non-optimal regions are usually a problem near the start of training; to avoid these, one must try several initial starting points to affirm that they have not started in such a region.

3. Toy Problems

Having discussed several optimisers alongside their strengths and challenges, we will now briefly look at some toy examples. This will allow us to develop a better intuition on these challenges and should highlight some differences between the optimisers discussed in section 2. For figures 4 and 5, the toy examples are not just simply static optimiser test functions. In reality, when training a neural network, these optimisers will not have access to the exact loss function, only an estimate due to training in mini batches. This means that from one update to the next, the estimate of the loss function and gradient is varying, so we wanted to incorporate this in our toy examples. To integrate this changing loss function, we incorporated slight deviations into how each function was centred for each epoch, we did this using the following:

$$f(x,y) = \frac{1}{N}\sum_{i=1}^{N} g(x + a_i,\ \ y + b_i) \qquad (21)$$

$$\bar{f}(x,y) = \frac{1}{|\mathcal{M}_k|}\sum_{i\in\mathcal{M}_k} g(x + a_i,\ \ y + b_i) \approx f(x,y) \qquad (22)$$

Where $a_i \sim Norm(0,\sigma^2)$ for i from 1 to N-1, $a_N = -sum(a_i)$ and similarly for b_i. $f(x,y)$ is the true toy function, $\bar{f}(x,y)$ is the stochastic approximation to the true toy function, each of which is used for each update for GD and SGD respectively. $g(x + a_i,\ \ y + b_i)$ is the slightly off centred true toy function, x and y are a fixed grid of points. N is the total number of a_i and b_i pairs, in terms of a neural network it represents the size of the data. $\mathcal{M}_k$ here is a set of randomly sampled a_i and b_i pairs from {1, . . , N} without replacement, it is of size M where M=$|\mathcal{M}_k|$ for all k, which represents the size of a mini batch. When performing GD, the true toy function

$f(x,y)$ is used to calculate each update. However, when performing SGD, $\bar{f}(x,y)$ is used to calculate each update, which is no longer the exact true toy function but an approximation.

The first example we will look at is in figure 4, this is a very simple saddle point with a true function of $g(x,y) = Z = x^2 - y^2$. It is important that an optimiser can escape a saddle point as the loss rapidly increases and decreases in different dimensions. Therefore, small initial updates can cause dramatic changes in the loss, this is another definition of ill-conditioning. GD was the only optimiser to get stuck at the critical point, this is due to the use of the true toy function and the specific starting point. This is an unlikely scenario as our starting y coordinate purposely aligns to that of the critical point. Because of this the optimiser only navigates in the x direction as the gradient is only in that direction, oscillating back and forth around the critical point until it converges. In fact, GD is unlikely to converge to any strict saddle point if the initial starting point is chosen randomly.

The reason why the other optimisers do not get stuck is due to mini batches incorporating variability in our toy function, it is no longer using the true function but the estimate; this estimate will be shifted slightly on the axes. Looking at the other optimisers in figure 4, we see after the first step the parameter in the x dimension still has the biggest update. However, due to using the estimated toy function the starting y coordinate is no longer exactly aligned with the critical point. Therefore, there is a small gradient and update in the y dimension. Larger and larger updates are then made in the y direction due to the increasing gradient. As mentioned earlier, these types of saddle points can cause the hessian to become ill-conditioned, which can result in slower training or for the case of GD halt training completely. Clearly, momentum helps avoid these consequences, looking at figure 4 and comparing SGD and SGDM, we see SGD taking four times as many updates to escape this saddle point. This demonstrates that momentum is very useful in escaping saddle points, allowing for faster convergence and reduced oscillations.

The reason that ADAM, MAWS and SWATS escape slightly faster than SGDM is due to their adaptive learning rates. In this case, the gradients in the y coordinate have been given a larger learning rate. On top of momentum, it allows it to make larger updates in the y direction, resulting in it escaping the saddle point more efficiently than SGDM. The fact that SGDM is slower at escaping saddle points is supported by [21], in which they explain in a bit more detail why this is exactly the case.

Reverting to section 2.E, we now have more confidence that our optimisers will not get stuck in a

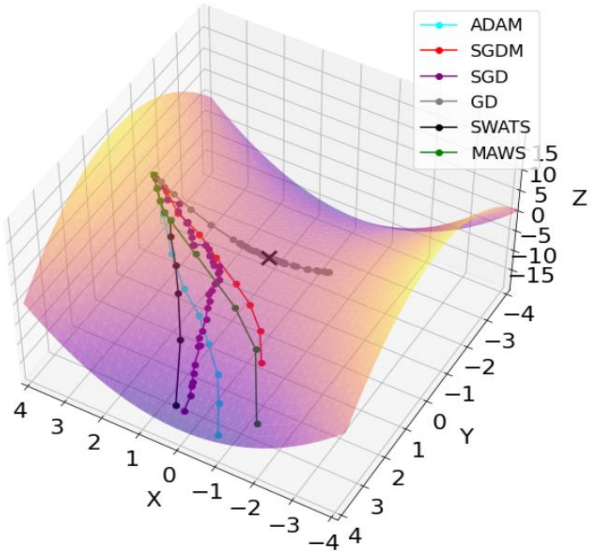

Figure 4: 3D surface plot of a saddle point with a function of $g(x,y) = Z = x^2 - y^2$. We show trajectories for 6 different methods. The critical point is at the origin (0,0) and the starting point is located at (3,0). The total number of iterations for ADAM, SGDM, SGD, GD, SWATS and MAWS are 8, 9, 39, 55, 10 and 9 respectively.

saddle point. In fact, as long as some form of momentum is present, the escape time does not seem to be an issue. Thus, it cannot be having a role in the differences in convergence speeds of these algorithms, since each optimiser apart from SGD escapes within ±2 epochs from one another. Lastly, we included the melded optimisers in this toy example just to illustrate the difference between the two transitions, this will be discussed in section 5 once we have covered our proposed optimiser. Since each toy example is just a small local region in an actual neural network loss function, we do not gain much intuition from these melded optimisers. As it is likely that when they approach regions similar to these toy examples, they will solely be in an ADAM or SGD phase. For example, in this scenario, we know that saddle points are in more abundance at higher losses, Therefore, we would expect the melded optimiser to still be in the ADAM phase, as higher losses are present at the beginning of training. Due to this and for clarity in the graphs, in the rest of the toy examples we focus on ADAM, SGD and simpler optimisers discussed in section 2.

Our second toy example seen in figure 5 is a simple function of $g(x,y) = Z = 0.1 * |x|^2 + |y|$, we aim to demonstrate the effects that batch size has on the path taken by the optimisers. Note that SGD and ADAM have a momentum of 0.7 in this example, and a learning rate scheduler was used for the SGD optimisers.

For the SGD optimisers, looking at figure 5, clearly the mini batch size is having an impact on the noise present in the updates. With smaller mini

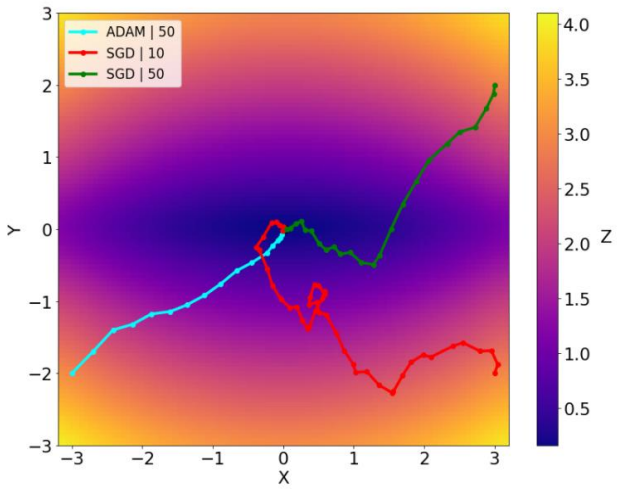

Figure 5: Contour plot of the function $g(x, y) = Z = 0.1 \cdot |x|^2 + |y|$. The different gradient descent paths taken by the respective optimisers are also plotted. The global minimum is at the origin (0,0) and due to symmetry the three equivalent starting points are located at (3,2), (3, -2) and (-3, -2). The numbers in the legend represent the batch size parameter M. The total number of iterations for convergence for ADAM | 50, SGD | 10 and SGD | 50 are 16, 43 and 22 respectively.

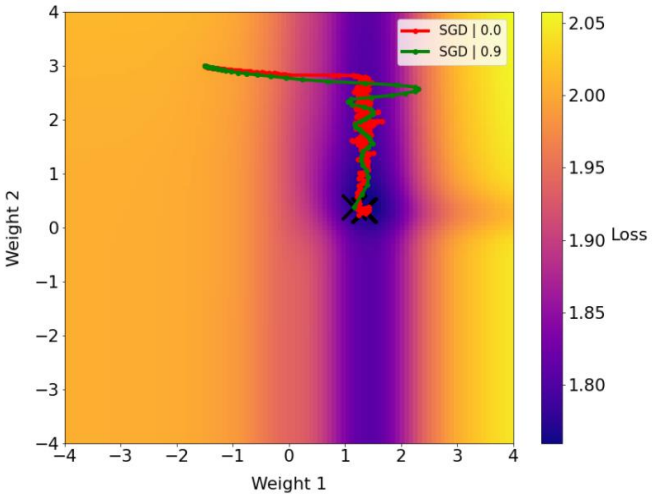

Figure 6: Contour plot of the loss surface of a one layered feedforward neural network. The different gradient descent paths taken by the respective optimisers are also plotted. The starting point is at (-1.5, 3.0) for weights 1 and 2 respectively. The numbers in the legend represent the momentum parameter ρ. The total number of epochs for convergence for SGD | 0.0 and SGD | 0.9 are 5 and 2 respectively.

batches, we have more noise in the updates as the estimate of the loss function and gradient is inherently less accurate. The larger the mini batch, the closer the estimate is to the true gradient, and thus less variance is in each update. It depends on the application as to how large a mini batch is needed, and it is an important hyperparameter to tune. With smaller mini batches, there is less memory requirements and reduced computational time, but more variance in the updates. Without a learning rate scheduler, one would see more variance specifically around the critical point even with momentum present, making it harder to converge to the exact minimum.

Lastly let us look at ADAM, it is the fastest optimiser to converge and seems to have much less variance in its updates. Its trajectory is in the relevant direction with very little deviation, this is again due to its first and second moments that allow it to utilise past updates and adaptive learning rates. We can see this in action in figure 5, comparing the size of the updates to the other optimisers, ADAM clearly has larger updates in the x direction than any other optimiser early on.

Our last example in figure 6 differs from the rest as it is not a toy function, it is a plot of an actual neural network surface. More specifically, we trained a single layer feedforward neural network on the MNIST dataset [8]. The MNIST dataset consists of 70,000 images of handwritten examples, we used a mini batch size of 64 for both SGD optimisers. In terms of the dimensions of the neural network, the input layer, hidden layer and output layer had dimensions of 784, 4 and 10 respectively. Due to the high dimensionality of the neural network, it makes plotting the loss surface very difficult. For this example, we first had to train the neural network to obtain the best fit parameters; with this, we chose two weights from differing layers. Weight 1 is from the hidden layer and weight 2 is from the output layer of the neural network. To plot the surface, we fixed all but the two weights and calculated the loss over a grid, the grid consisted of 400 different pairs of values for the respective two weights. Lastly, to plot the paths over this surface we picked initial starting values for the two weights, while also keeping the already fitted weights fixed and recorded the updates in these two weights while we trained the neural network.

For this example, we wanted to highlight the similarities that this surface has with our toy examples, validating our current findings. We also wanted to demonstrate how momentum can help traverse regions of small gradients, in this case a ravine. A ravine in this sense is a region where the gradient is steep in one dimension (weight 1) and flat in another (weight 2). Moving onto how the optimisers performed, we can see how momentum significantly reduced the number of oscillations associated with SGD. This resulted in better convergence speeds, with SGD | 0.9 only taking 2 epochs as opposed to 5 epochs taken by SGD | 0.0. Momentum accelerated our descent downwards to the ravine, but also increased update sizes significantly within the ravine. As explained in

section 2.E, momentum is a great solution for optimisers traversing flat regions.

4. SOLUTION

We have discussed several melded optimisers in the above sections, all differing by how they combine ADAM and SGD updates in order to minimise the trade-off between generalisability and convergence speed. In this section, we propose our algorithm which is called MAWS (Melding ADAM with SGD), the pseudo code for this algorithm can be found in the appendix in algorithm 1. Before making the algorithm, we had several problems to solve, with the first being how to meld the optimisers. We opted with transitioning from ADAM to SGD, instead of the MAS approach, where they combine each optimiser for every update. We opted for this since the former had the best performance in minimising the trade-off, which is the goal of any melded optimiser. As such, it was a sensible starting point.

The second problem is how to transition, SWATS and AdaBound are on two opposite sides of the spectrum when it comes to the transition. The former is abrupt, while the latter is continuous throughout training. In the Adabound paper, the authors suggest that more experimentation is needed to be done on how to conduct the transition. We took inspiration from this and wanted to explore the area in the spectrum in-between these optimisers. We decided that we wanted two distinct phases of ADAM and SGD like SWATS, however, we also wanted a smooth transition similar to Adabound. To meet these criteria, we decided on three distinct phases, ADAM, mixed and SGD. The first phase involves the ADAM optimiser to make updates to our parameters. The second phase uses a varying linearly weighted contribution of ADAM and SGD updates; an adaptation to the MAS optimiser. The last phase simply uses SGD to update our parameters. On the spectrum mentioned above, our approach does lie closer to SWATS. We chose to have distinct ADAM and SGD phases for our criteria as we want to maximise the benefits from the speed of convergence and generalisability of ADAM and SGD respectively. For example, in AdaBound, there is no real distinct phase, due to the continuous transition throughout training. Early on in training for AdaBound, we will lose some of the speed of convergence from ADAM as it is already slowly transitioning to SGD, a similar thing can also be said for generalisability. We believed this may hinder minimising this trade-off, thus we decided on having distinct ADAM and SGD phases. The smooth transition was also added to our criteria after experimenting with SWATS. The abrupt transition in SWATS led to some issues in learning; more specifically, it caused unwanted rapid increases in the gradient norm, which we will talk about in section 5. Therefore, to meet all our criteria, a third mixing phase had to be made for the smooth transition.

The third issue to solve was when to transition into each phase. This is particularly important as it has a big impact on the trade-off; as mentioned previously, transitioning too early or late results in a larger trade-off which is not what we want. Having experimented with SWATS, we had concerns around their switching criterion, found in equation 18. More specifically, in order to see any transition at all we had to tune t, this is not mentioned in the paper and must be considered as an extra hyperparameter. Additionally, once t was tuned, we saw transitions at non-optimal times during training. As a result, we did not implement the switching point of SWATS into MAWS. Instead, to solve this issue, we opted to make two explicit hyperparameters that decide how many epochs our algorithm stays in each of the three stages:

$$\eta_A = E \cdot \varphi_{Adam} \qquad where\ \varphi \in [0,1] \quad (23)$$

$$\eta_M = E \cdot \varphi_{MIX} \qquad where\ \tau \in [0,1] \quad (24)$$

$$\eta_S = E - \eta_A - \eta_M \quad (25)$$

where η_A, η_M and η_S are the number of epochs in the ADAM, mix and SGD phase respectively, E is the total number of epochs. φ_{Adam} and φ_{MIX} control the number of epochs in the ADAM, mix and SGD phase. Note that $\varphi_{Adam} + \varphi_{MIX} \leq 1$ such that $\eta_A + \eta_M + \eta_S = E$. These hyperparameters provide full control over when the respective transitions occur, unlike the SWATS optimiser. Another benefit of these hyperparameters for MAWS is that there easily interpretable, unlike the hyperparameter t for SWATS. One can see how we implemented this on lines 2 and 3 in the pseudo code in algorithm 1. We also discover that the performance of the neural network is much less sensitive to these hyperparameters in comparison to the SWATS hyperparameter t, thus reducing the need of tuning them. As such, we propose a default value of 0.2 and 0.3 for φ_{Adam} and φ_{MIX}. Although tuning will help maximise performance, the results will be near equivalent as we will demonstrate in section 5.

The last problem to solve was what learning rate to use for SGD during the mixing and SGD phase. This learning rate is just as important as the initial learning rate that needs tuning for the ADAM phase. We have mentioned earlier the consequences of too large or small of a learning rate, which still applies here. However, we do not want to simply add a second learning rate to our hyperparameters, as it is particularly difficult and time consuming to tune. After our experimentation with SWATS, we decided to incorporate their solution to this problem. That is using Λ in equation 18 for our SGD learning rate in

the mixed and SGD phases. Looking at lines 29, 36 and 44 in the pseudo code in algorithm 1, you can see how this is implemented and how we use this learning rate for both the mixing and SGD phases. Although we found difficulties with the transition point, the learning rate Λ was able to select an arguably good learning rate that minimised disruption in learning. By disruption, we mean minimal spikes in the gradient norm, loss and accuracy. It is an extremely useful method that allowed us to avoid the burden of tuning an extra hyperparameter.

We will now discuss the details of the mixing phase as this is where the main differences between MAWS and SWATS lie. During the mixing phase, we wanted as smooth of a transition as possible. Instead of taking the average of an ADAM and SGD update throughout the mixing phase like MAS, we decided to do linearly decreasing and increasing weighted contributions for ADAM and SGD respectively. Apart from the above, the method used to meld the learning rates and gradients was inspired by MAWS. Over η_M epochs we would linearly start increasing SGDs contribution and reduce ADAMs contribution equally. This is summarised in the following equations; however, we suggest looking at lines 3, 34, 35 and 38 in the pseudo code in algorithm 1 to help with understanding:

$$\mu_k^A = \begin{cases} 1, & k < \eta_A \cdot S \\ 1 - \dfrac{k - \eta_A \cdot S}{\eta_M \cdot S}, & \eta_A \cdot S \le k \le (\eta_M + \eta_A) \cdot S \\ 0, & k > (\eta_M + \eta_A) \cdot S \end{cases} \quad (26)$$

$$\mu_k^S = 1 - \mu_k^A \quad (27)$$

$$\alpha^{Melded} = \mu_k^A \cdot \alpha + \mu_k^S \cdot \Lambda \quad (28)$$

$$\Delta_k^{Melded} = \mu_k^A \cdot \Delta_k^{ADAM} + \mu_k^S \cdot g_{k-1} \quad (29)$$

$$\theta_k = \theta_{k-1} - \alpha^{Melded} \cdot \Delta_k^{Melded} \quad (30)$$

Where S = N/M which is the number of iterations per epoch, N and M are again the number of observations and batch size respectively. Please note we have converted the transition lengths from equations 23, 24 and 25 from the number of epochs to the number of iterations k, for equation 26. μ_k^A and μ_k^S are the wights for ADAM and SGD at iteration k respectively. Therefore, as k increases, μ_k^S increases towards 1 and μ_k^A decreases towards 0 by the same amount, achieving a smooth transition from ADAM to SGD over η_M epochs. Again, we have the option to incorporate momentum for the SGD updates. Note that the method used in the pseudo code is in terms of epochs and not iterations, but it is equivalent to the above.

We can derive the same intuition mentioned in the MAS optimiser in section 2.D, where if ADAM and SGD agree on the direction of the update, the resulting update will be boosted in that direction. However, for MAWS, the ADAM and SGD contributions will be varying for each update. We experimented with several types of nonlinear weights; however, the linear weights provided the smoothest transition. Having discussed the challenges we overcame, the reasoning behind our optimiser and the benefits it can bring, we want to disclose an area of concern. As mentioned above, we need to manually select the length of each phase using φ_{Adam} and φ_{MIX}. We have default values for the phase lengths and will soon show in section 5 that there is a range of values for these phases in which performance is not affected. However, there is still a possibility that other architectures of neural networks or even different datasets to what we tested on that could warrant completely different values. This would cause the need to tune these hyperparameters, resulting in them being a limitation to our design.

Lastly, we wanted to briefly explain how we implemented this algorithm into code. The coding language used was Python 3.9, and we implemented the optimiser in PyTorch. PyTorch is an open-source machine learning library specialising in deep learning. We chose Pytorch as it is the standard in machine learning research on optimisers, and also provides the option for low level customisation, which is required when creating a custom optimiser.

5. Results and Evaluation

Having discussed our proposed optimiser MAWS, we want to refer back to figure 4 to explain the subtle difference between SWATS and MAWS on this toy function. We can see that both optimisers transition after the fifth epoch, SWATS transitions into the ADAM phase, whereas MAWS transitions into the MIXED phase for a further 3 epochs. Due to the abrupt transition, SWATS takes a more direct path. However, we purposely prolonged the mixing phase for MAWS to demonstrate the contributions ADAM and SGD are having. In this mixed phase, we see some boosted steps as ADAM and SGD updates are in a similar direction, allowing MAWS to escape the saddle point by one less epoch compared to SWATS. The last point we want to make is that being able to tune when each transition occurs for MAWS is a great advantage. For the SWATS optimiser, we have little to no control over when the transition occurs. We can tune t in equation 18; however, we found it to be extremely unreliable. Tuning t only helped us see a transition occur rather than helping us tune when the transition occurs. It is also important to realise that it is difficult to intuitively interpret t, but the hyperparameters for MAWS are easy to interpret, which is another advantage MAWS has over SWATS.

A. Experimental setup

In order to test and evaluate our new optimiser, we decided to compare its results with ADAM and SGD since we are evaluating how well MAWS minimises the trade-off. But we are also comparing MAWS to SWATS as it is the closest in terms of style to our proposed optimiser. Thus, we should be able to better evaluate the transition phase of MAWS, as it is the main differing factor between it and SWATS. In terms of the evaluation method, we adopted the standard approach of dividing the dataset into train, validation and test sets with fractions of 0.6, 0.2 and 0.2 respectively. Other evaluation methods such as bootstrap are not as viable for neural networks due to the time it takes to train a neural network. As stated previously, we measure the performance of the neural networks using the accuracy metric since this provides a measure for generalisability and is the standard in this field. We evaluate each optimiser on two different datasets and two different neural network architectures. In terms of the data, we used the MNIST dataset [8]. This consists of 70,000 images. Each image consists of a greyscale handwritten number between 1-9, the size of the image is 28x28 pixels. The other dataset we used is CIFAR10 [9], this dataset consists of 60,000 images. These images are coloured, and each belongs to one of 10 classes and each class is of equal size (6000 images per class). The size of each image is 32x32 pixels. There are several reasons we chose these two datasets. Firstly, these two datasets are the standard datasets used in optimiser research papers, allowing for easy comparison across various papers, providing a more effective evaluation of each optimiser. Secondly, each dataset is highly dimensional, which is perfect for a neural network to excel and provide excellent performance. Lastly, these datasets are well known and understood and so are easy to implement. For example, one can find a good range of values for hyperparameters in papers making tuning less time consuming.

Moving onto the neural network architecture, for the MNIST dataset, we used a simple one layer feedforward neural network with a hidden dimension of 100. The reason we used this simple architecture is due to the simplicity of the dataset. We do not want to introduce any more parameters to the network than needed; otherwise, overfitting can become an issue. This simple network can produce excellent results quickly and reliably. It is also the standard architecture used in research papers for the MNIST dataset, again allowing for easy evaluation over different research papers.

For the CIFAR10 dataset, we used a convolutional neural network, the exact architecture is found in [10] and is called LeNet-5. It is a straightforward convolutional neural network with 5 layers with learnable parameters and a small memory footprint. This was particularly beneficial as we used CPUs to conduct our experiments on all of our neural networks; simple, efficient networks keep the computational time to a minimum. Unlike the case for the feedforward neural network, the LeNet-5 architecture is not the superior architecture for performance on CIFAR10. As such, LeNet-5 was not the standard in research papers. The reason we did not use the state-of-the-art network, called ResNet-32 [22], was due to the computational time it takes to run this network for this dataset. As reported in [17], even with 16 NVIDIA Tesla K80 GPUs, it would take roughly 3 weeks to train and evaluate the neural network. As such, we opted for a simpler architecture that still produces competitive results. For comparison, we used 4 CPUs which took 5 days to conduct our training and evaluation for the CIFAR10 dataset. Also, unlike other melded method papers, we have compared and evaluated our optimiser MAWS to that of another melded method, SWATS. Thus, there is no need to do cross comparisons over the SWATS paper and ours.

In terms of hyperparameters, for each experiment, we tuned the learning rate for all optimisers. We adopted the standard approach found in papers for tuning all of our hyperparameters. For ADAM, MAWS and SWATS, we chose the learning rate from a grid of 20 values linearly ranging from 0.0001 to 0.005. For SGD, we coarsely tune the learning rate on a logarithmic scale from 10^{-3} to 10^{2} and then fine-tuned the best performing learning rate. The reason we have a larger grid search for SGD is because its best initial learning rate can vary by several orders of magnitude. But ADAM and hence the melded optimisers are much less sensitive and so their best initial learning rate are of similar magnitudes. As mentioned previously for SWATS, we had to tune t in equation 18 to see any transition. We did a simple logarithmic grid search between 10^{-2} to 10^{-6} to find the best value for t; the size of the grid was 15. The hyperparameters that control the switching for MAWS $\boldsymbol{\varphi_{Adam}}$ and $\boldsymbol{\varphi_{MIX}}$ were chosen from a grid of {0.1, 0.15, 0.2,0.25, 0.3,0.35, 0.4,0.45, 0.5}. We conducted a simpler and smaller grid search for MAWS as the neural networks are less sensitive to these hyperparameters. Throughout these experiments, every other hyperparameter was kept constant between optimisers. Momentum was kept at 0.9, and β_1 and β_2 had values of 0.9 and 0.999 respectively. The batch size and regularisation hyperparameters like drop out and L2 regularisation, which have not been mentioned nor discussed previously, were all chosen by the recommended values of the base architecture. There was no need to tune the latter hyperparameters as they do not affect the performance of the optimisers. Although tuning them would improve performance, it is not needed

Name	φ_{Adam}	φ_{MIX}	Avg acc.	Acc max.
MNIST				
Adam	-	-	97.13	97.71
SGD	-	-	**97.44**	**98.25**
SWATS	-	-	97.08	97.27
MAWS	0.1	0.2	97.11	97.31
MAWS	0.2	0.3	97.17	97.53
MAWS	0.2	0.4	97.01	97.19
CIFAR10				
Adam	-	-	78.37	80.13
SGD	-	-	76.54	79.47
SWATS	-	-	72.15	73.97
MAWS	0.1	0.2	78.26	81.43
MAWS	0.25	0.3	**80.36**	**83.81**
MAWS	0.25	0.4	80.09	82.55

Table 2: Average accuracy and maximum accuracy over 6 runs of the MNIST and CIFAR10 dataset.

when we are solely comparing optimisers to one another. Lastly, the number of epochs was also chosen by considering the base architecture recommendations, but also the convergence and computational time it took to run each experiment.

B. MNIST Experiment

Moving onto the results of our experiments, we will first look at the MNIST results and follow that with the CIFAR10 results. We will be providing insights from the results and graphs we obtained, to finish the section we will talk about generalisability, possible limitations in our experiments and finally future work. Please note that for figures 7 to 12 the hyperparameters φ_{Adam} and φ_{MIX} for MAWS have the default values of 0.2 and 0.3. For the melded methods on the MNIST dataset, SWATS transitioned at epoch 15 and MAWS transitioned at epoch 30 and 60. Looking at the MNIST results in table 2, the first thing to notice is that SGD generalised best to our test data. Achieving the best average and maximum accuracy over 6 different runs. However, looking at figure 7 as expected and mentioned in previous sections, SGD had the slowest convergence speed of all optimisers. It is not blatantly obvious from looking at table 2 that the melded methods were not able to utilise SGDs generalisability, but when we look at figure 7 it is apparent. Comparing the other methods to SGD, we see much faster convergence times due to Adam adaptive learning rates early on; unfortunately, for the melded methods we see little progress in generalisability in the later stages of training, where SGD strives.

So why are the melded methods having poor generalisability even after switching to SGD? The cause must be coming from the initial ADAM updates as it is the only similarity between ADAM, MAWS and SWATS. If it were the SGD phase or

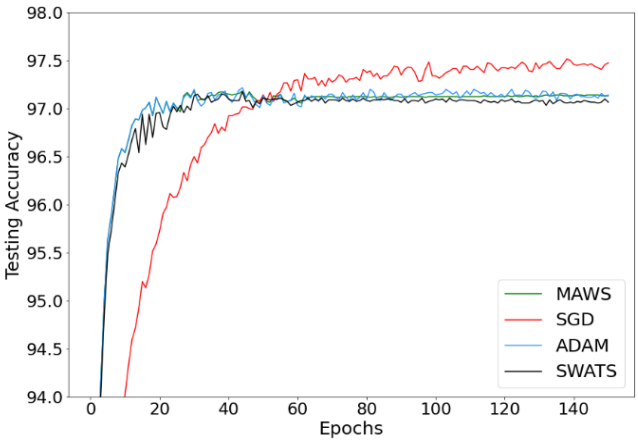

Figure 7: One layer feedforward neural network test accuracies on the MNIST dataset for the four respective optimisers.

transitions that were affecting the generalisability, we would not expect such a similarity in behaviour for the three optimisers that we do see in figure 7 and 8. Our first proposal is that ADAM and the melded optimisers are suffering from poor correspondence between local and global structures in the loss function. As mentioned in [20], this is simply when the direction of a local update in the parameters leads to a lower loss, however, the direction of the global update is in another direction which is optimal with even lower loss. Thus, leaving θ_k in a poor local region of the parameter space with respect to the global update. We understand that this poor correspondence must be occurring at the initial updates. This is because even when we reduce the length of the ADAM phase for MAWS hoping for increased generalisability, we see little difference in performance or generalisability. Thus, SGD must be taking updates in a direction closer to the global update than ADAM, MAWS and SWATS, such that SGD trajectory is in a direction of lower loss relative to the other three optimisers. Due to this poor correspondence, ADAM and the melded optimisers have worse generalisability.

Our second proposal to explain the reduced progress in generalisability after the initial stages of training is that ADAM and the melded optimisers are getting stuck in a critical point or a flat region. Although we argued this to be an unlikely scenario in section 2, Looking at figure 8, we see the gradient norms of the optimisers reduce to almost zero when compared to the gradient norm of SGD. As mentioned previously, if the gradient norm shrinks to an insignificant size, the problem revolves around critical points [20]. We must assume we are stuck in a local minimum as they are in more abundance at lower costs, to which we are at. As such, we must address that it is still possible for the melded methods to get stuck in a local minimum even in the SGD phase, especially when the gradient norm is so small,

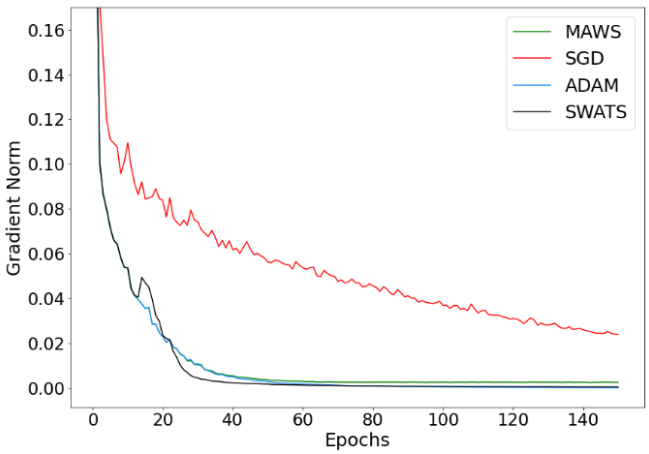

Figure 8: Gradient norm for the one layer feedforward neural network on the MNIST dataset for the four respective optimisers.

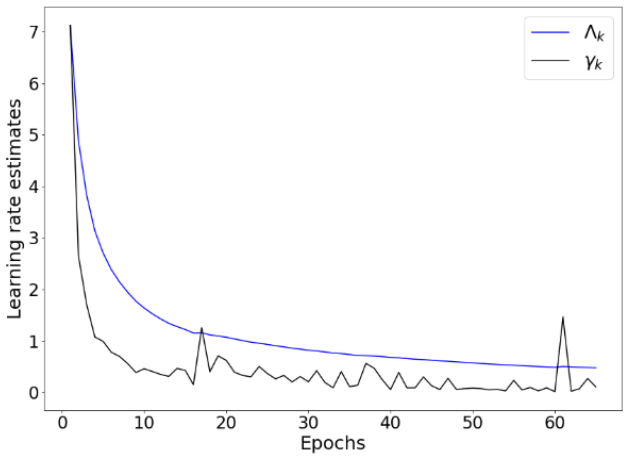

Figure 9: SWATS and MAWS learning rate estimates Λ and γ_k for the one layer feedforward neural network on the MNIST dataset.

which prevents large updates out of the minimum. Therefore, if we have poor correspondence between local and global structures, we will enter a region of higher cost than that of the SGD optimiser. With this and the fact that we are stuck in a local minimum, the insignificant gradient norm in figure 8 is explained but also the slight drop in accuracy of the optimisers since we are at a higher loss with respect to the SGD optimiser. Lastly, we want to add that the poor correspondence must be having a greater impact, since we see a consistent difference in accuracies when comparing the three optimisers to SGD over 6 runs. If it were solely the local minimum having the most significant impact, we would expect some runs not to get stuck in the local minimum and perform as well as SGD, but this is not the case. Lastly, in figure 8, we see SGDs gradient norm reducing at a slower rate as compared to the other methods. This could be explained by the fact that SGD has been found to converge to regions where flatter minima are present [23], such that we see the slowly reducing gradient norm. If it were a sharp minimum, one would expect a faster decrease in the gradient norm as it traverses down to the minimum much faster due to the larger gradients.

We will now investigate the differences between MAWS and SWATS. Looking at table 2 and figure 7, we can see that MAWS has better generalisability. We do not suspect this to be down to SWATS earlier transition point. As switching to SGD sooner should improve generalisability, not reduce it. We believe the main candidate for the reduced generalisability is down to the abrupt transition for SWATS. Looking at figure 8, it is clear the gradient norm spikes at the transition point for SWATS. However, the MAWS transition is much smoother, and no spikes can be seen. Allowing for a smooth transition in terms of the size of the updates from ADAM to SGD. Looking at figure 9, we can see at epoch 15 where SWATS transitions, Λ is larger than at epoch 30 where MAWS transitions. This could also be causing the spike in the gradient norm for SWATS. Due to the larger learning rate Λ and the abrupt transition, it causes a larger update in the model parameters. As mentioned in section 2, this impacts learning as it can cause θ_k to be in a non-optimal region in the parameter space after the large update. This could also explain the reduced generalisability for SWATS.

C. CIFAR10 Experiment

We will now turn our focus to the CIFAR10 dataset in which we used the LeNet-5 convolutional neural network. Looking at figure 10 and table 2, we can see that MAWS performs the best out of all the optimisers; as such, it had the best generalisation to the new data. As expected, SGD had better generalisation compared to ADAM, but ADAM had a much faster rate of convergence. We will touch upon this in subsection F.

Comparing MAWS and SWATS, we see a transition at epoch 41 for SWATS and at epoch 55 and 110 for MAWS. This is a significant difference allowing MAWS to have a faster rate of convergence due to its prolonged ADAM phase. These transitions are more clearly illustrated in figure 10, with gradient norms varying at the transition points. MAWS was still able to utilise SGD generalisability and perform the best in this area too, even though it was in the SGD phase for a shorter period. This is evidence that we have better minimised the trade-off between generalisability and speed of convergence as compared to SWATS. This also further justifies using extra hyperparameters for MAWS to have greater control over the transition points, allowing for optimal transition times, which SWATS cannot do. Again, there are 3 possible causes that are impacting SWATS performance in terms of minimising this trade-off when compared to MAWS.

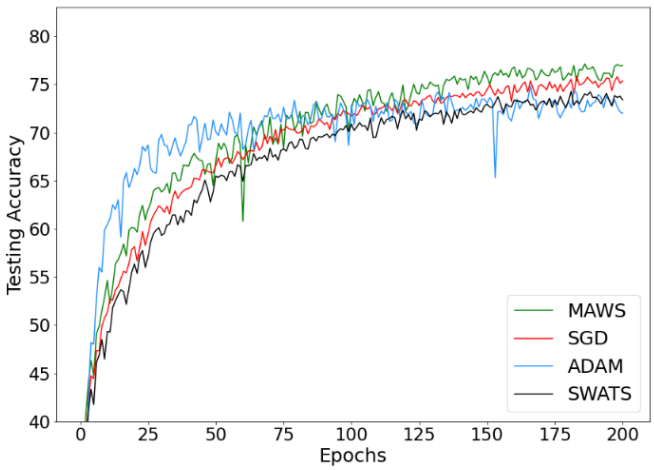

Figure 10: LeNet-5 test accuracies on the CIFAR10 dataset for the four respective optimisers.

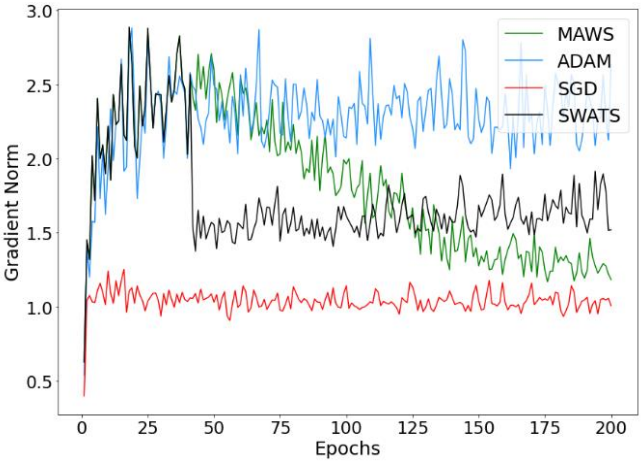

Figure 11: Gradient norm for LeNet-5 on the CIFAR10 dataset for the four respective optimisers.

Let us firstly look at the learning rate Λ after the transition in figure 12, as expected SWATS has a larger Λ than MAWS due to its premature transition. We believe this is affecting the update size in the SGD phase resulting in too large of updates in the parameter space. Looking at figure 11, we see the gradient norm for SWATS after the transition to be larger than SGDs, but also larger than MAWS when it is in the SGD phase. Although this is not having a massive impact on learning, it could still be detrimental in minimising the trade-off and could explain the difference in generalisability between SWATS and MAWS. Again, these large updates in the parameters can cause them to be in non-optimal regions.

The second possible cause could be down to the early transition of SWATS. With an early transition, we expect the convergence speed to reduce and generalisability to increase. However, looking at figure 10, SWATS generalisability did not increase with respect to MAWS, suggesting that either the earlier transition did not cause reduced performance or that the early transition reduced both generalisability and speed of convergence. The latter is less plausible as in [17] they demonstrate switching early improves generalisability. However, the former is plausible. Since for MAWS, the model was not sensitive to $\boldsymbol{\varphi_{Adam}}$ and $\boldsymbol{\varphi_{MIX}}$; this claim is discussed in subsection E. As a result, we expect this earlier transition to have little impact with respect to our other proposals.

Our third hypothesis as to why SWATS was not able to minimise the trade-off as efficiently as MAWS is that the transition is abrupt. Looking at figure 11, we see an abrupt drop in the gradient norm for SWATS at the transition. There is a clear distinction between this sharp drop and the sharp increase we see in the MNIST case. Unlike the MNIST case, the sharp decrease in the gradient norm does not cause θ_k to be in a non-optimal region in the parameter space, as the updates are not large. However, in this case it causes the updates in θ_k to become much smaller almost instantaneously after the transition. The smaller updates do not necessarily hinder the generalisation of SWATS; however, it does hinder the speed of convergence since we are making smaller and slower progress in updating our parameters. Therefore, we suspect that SWATS could achieve the same generalisability as MAWS but over more epochs, we see this slower convergence of SWATS in figure 10. The abrupt transition on the other hand could be affecting both generalisation and speed of convergence, hence it could explain why SWATS cannot minimise the trade-off to the same extent as MAWS.

Unfortunately, what we are about to propose has not been tested, nor has there been much research in the area of transitions and its effects on the trade-off. As stated in [16] and demonstrated in our toy plots, ADAM and SGD take different trajectories in the parameter space. This is due to ADAMs adaptive learning rates and exponential averaged gradient, which reduces the stochasticity in its updates. As mentioned previously, this reduces ADAMs ability to escape regions of high gradients like sharp local minima. As a result, ADAM is more likely to be in regions of sharp minima or areas where large gradients are present [16]. We propose that switching abruptly with SWATS impacts the ability to escape these regions even in the SGD phase. Whereas looking at figure 11, we see MAWS smooth transition from ADAM and SGD. Thus, in comparison to SWATS, we can make greater progress escaping these types of regions during the transition, as we have both ADAMs speed and SGDs improved ability to escape local minima. The limitation in this reasoning is that we are assuming that SWATS is in these types of regions at the end of the ADAM phase, which may not be the case. However, even if this is not the case, a smooth transition is still able to utilise both speed, generalisability and the property to escape sharp

minima to a greater extent. We believe this results in MAWS having better progress when transitioning to SGD, which results in efficient parameter updates and better generalisability. We see this in figure 10, after the switch, SWATS was not able to make as much progress in the parameter space; as such, its accuracy and generalisability was always slightly lower than MAWS.

We again conclude that MAWS was able to better minimise the trade-off between generalisability and speed of convergence as compared to SWATS. But also, for this model, MAWS was able to outperform both ADAM and SGD. We believe this is due to the mixing phase in MAWS, allowing for a smoother transition. The mixing phase prevents two things, a sudden increase in the gradient norm and a sudden decrease. Both of which have been demonstrated by our two experiments.

D. The Drawback of SWATS

We now want to look at why SWATS transition criterion caused us some problems and thus why we did not incorporate it in our optimiser. Looking at figures 9 and 12, we can see that each looks respectively similar in their trends. With Λ being an exponential trend and γ_k being a noisy estimate of the scaling required as mentioned in section 2.D. Early on in training, both quantities differ by a significant amount as designed, since we do not want to switch too early. However, later in training, with the foresight of knowing when the optimal time to switch from MAWS, we know particularly with the CIFAR10 dataset, the SWATS criterion does not switch at the optimal time. From the figures, we see that when SWATS switches and the criterion in equation 18 is met, it is due to a spike in γ_k for that epoch, allowing the difference between γ_k and Λ to be less than t in equation 18. This difference in subsequent epochs then grows, which is not ideal as we want and expect a continuingly decreasing difference as the number of epochs increases. This is not the case, the criterion is met by what seems to be somewhat random spikes in γ_k, which is causing issues in switching early as seen in our two experiments. This is the reason we decided to implement our own method of deciding when to transition using our two extra hyperparameters. This is not a perfect solution, and its limitations are discussed in the following subsection.

E. MAWS Hyperparameters

The last topic we wanted to talk about with respect to our experiments is the hyperparameters $\boldsymbol{\varphi_{Adam}}$ and $\boldsymbol{\varphi_{MIX}}$ for MAWS, which control when each transition occurs. Firstly, we tuned these hyperparameters for each experiment to obtain the best performance. Surprisingly, we see a value of 0.3

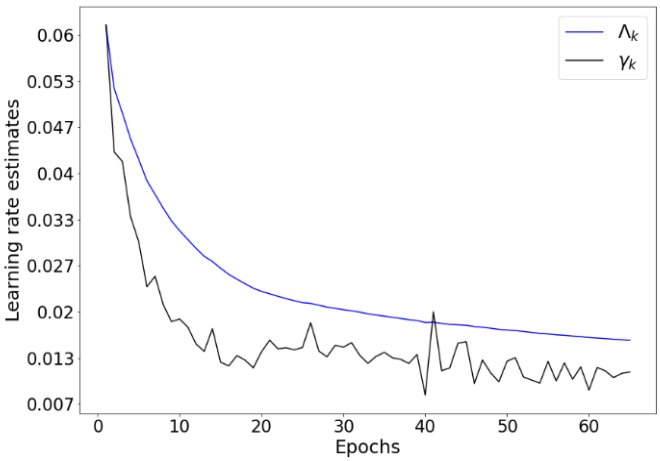

Figure 12: SWATS and MAWS learning rate estimates Λ and γ_k for LeNet-5 on the CIFAR10 dataset.

being consistently the best for $\boldsymbol{\varphi_{MIX}}$ and a value around 0.2 to be optimal for $\boldsymbol{\varphi_{Adam}}$. We also trained each network with 2 different pairs of values for $\boldsymbol{\varphi_{Adam}}$ and $\boldsymbol{\varphi_{MIX}}$, which are also tabulated in table 2, we did this to see how sensitive the networks are to these hyperparameters. We see from the table that the networks performance is reduced but only slightly with non-optimal values. However, they still perform better or equivalently to SWATS, ADAM and SGD. As a result, it is not crucial to tune these hyperparameters as we see values for $\boldsymbol{\varphi_{Adam}}$ and $\boldsymbol{\varphi_{MIX}}$ of 0.2 and 0.3 to be consistently optimal for the optimiser. Please note that out of the two hyperparameters, $\boldsymbol{\varphi_{Adam}}$ varies more, intuitively, this makes sense since the switching point should depend on the architecture and dataset that is being used. However, we can put constraints on the range of the optimal value for $\boldsymbol{\varphi_{Adam}}$, in all of our experiments we found $\boldsymbol{\varphi_{Adam}}$ lies in between 0.15-0.25. We speak about how this is a possible limitation in subsection G. The fact that 0.3 is consistently the optimal value for $\boldsymbol{\varphi_{MIX}}$ suggests that no matter the architecture or dataset, the mixing phase should always be around 30% of the total number of epochs. Although unexpected, it is a plausible result as it may just be the optimal mixing length to achieve the smoothest transition.

F. Generalisability

Throughout these experiments, we demonstrated the benefits of using melded optimisers to minimise this trade-off between generalisability and speed of convergence. However, these do not work perfectly and even in our MNIST experiment SGD significantly outperformed the most recent adaptive and melded methods. To make further progress in the melded optimisers field, we need to understand why SGDs generalisability is unmatched. But also, how to incorporate that into an adaptive method like ADAM to benefit from its speed of convergence.

There has been a lot of work done recently on understanding this gap between the speed of convergence and generalisability. Most recently, in [16], they investigated how the escape time of SGD and ADAM differ for minima, basins and asymmetric valleys, which leads to insights into why SGD generalises better. The authors found that the escape time from these regions depends on the Radon measure of the basin, minima or valley but also the gradient noise in these algorithms. Radon measure is a measure on the σ-algerbra of Borel sets of a Hausdorff topological space X that is finite on all compact sets, outer regular on all Borel sets, and inner regular on open sets. The authors demonstrate that SGD escapes faster than ADAM in these regions. This is due to ADAMs adaptive learning rate and its use of its exponential gradient average, both reduce the gradient noise, which results in larger Radon measure. As such, SGD is more locally unstable than ADAM at regions of sharp gradients, minima and valleys and thus can escape from these regions to flatter regions. As a result, ADAM gets stuck in these regions, which are known to overfit and thus have bad generalisation. This helps to explain why we see this generalisation gap between ADAM and SGD in both of our experiments. Particularly in the MNIST experiment where ADAM gets stuck at a local minimum. However, it does not help us in the context of melded methods. As we still see MAWS and SWATS being trapped at a local minimum with poor generalisability for the MNIST experiment, even when they are in the SGD phase. Additionally, we do not know the effects that switching from one method to another has on the gradient noise nor the radon measure, we hope future work can be done here.

G. Limitations to Our Experiments

For our experiments and results, there are important limitations that need to be discussed. The main limitation to our results is the lack of a strong theoretical foundation in this field. This limits our understanding of optimisation in the context of highly multimodal loss surfaces, such as those encountered in deep learning. This limitation had to be considered throughout the formulation of this paper, particularly when interpreting our results.

Our second biggest limitation is the number and variety of datasets and architectures we tested our optimiser on. This is a significant limitation, especially when we are trying to evaluate how well our proposed optimiser generalises to new data. This is because the results we calculated may not be indicative of MAWS general performance on other datasets and thus may threaten the validity of our results. To truly understand how well MAWS generalises to new data as compared to SGD, more experiments need to be conducted. For example, if we had access to more CPUs/GPUs we could have experimented with recurrent neural networks on audio data, implemented superior architectures like ResNet-32 as mentioned previously or even explored tabular data using a new architecture by google cloud AI called TabNet [24]. Doing this would allow for more robust testing and more confidence in our evaluation and conclusion. On a similar note, we did evaluate our optimiser with respect to several others, but with the number of new optimisers increasing, being able to test more optimisers would have been beneficial to our evaluation. This would have also allowed us to understand where our optimiser succeeds over other optimisers, but possibly even more importantly, it may have allowed us to identify areas of weaknesses. To address these limitations, we did select the most relevant optimisers to compare MAWS to and used two vastly different architectures.

Another possible limitation to our experiments is our hyperparameter tuning. Although we followed the standard procedure with respect to other optimisation papers, there are still weaknesses to what we did. More specifically, we kept momentum and the total number of epochs constant across optimisers, but also used recommended values for the batch size and regularisation hyperparameters like drop out and L2 regularisation. As explained previously, these hyperparameters are not explicitly for the optimisers and thus should not affect performances, as long as they are kept fixed among all optimisers. However, for optimisers like ADAM, which are known too overfit. If we had tuned the L2 regularisation parameter, we may have seen increased performances relative to the other optimisers. Therefore, this could be affecting the validity of our results. Due to the vast number of hyperparameters in a neural network, it is very difficult and time consuming to tune each one. Ideally, with enough time and computational power, we would have done this. In order to minimise this limitation, we and all other optimisation papers tune the hyperparameters associated with the optimisers. These are the most relevant and influential, allowing reduced training time.

Our last limitation is something we have mentioned previously, which is the additional hyperparameters that control how long we are in each phase for MAWS. We argued that the architectures used in the experiments were not sensitive to these hyperparameters. However, we do not know if this will be the case for all architectures. Additionally, we found that the architectures were more sensitive to the hyperparameter φ_{ADAM} than φ_{MIX}, which controls when to transition away from the ADAM phase. Suggesting there is not a fixed optimal value for all architectures and datasets, but also that the transition away from ADAM is the most important

one. This also ties into possible future work in this area since additional hyperparameters are not desirable due to the increased time of hyperparameter tuning. We ideally want a mechanism that allows automatic transitioning into the three separate phases without any hyperparameters. As covered in section 2.D, SWATS algorithm does this for two separate phases. Although, as mention above, we found difficulties in reproducing consistent results with this method. Therefore, we propose possible future work to be done in this area. More specifically, we think the SWATS method of using equation 18 for the criterion could be improved. Such that it does not rely on the spikes in γ_k that we see in figures 9 and 12 to meet the criterion in equation 18 for the transition to occur. We propose that using the gradient norms to help decide when to transition could be a better option. For instance, looking at figure 11 with the foresight of knowing the optimal transition away from the ADAM phase is at 55 epochs, this is around the time when the gradient norm of ADAM starts to stabilise around a value of 2.3. Possibly by monitoring this, we can automatically decide when to transition away from the ADAM phase. Additionally, one would have to decide the length of the mixing phase too. However, as our results suggest, the architectures were much less sensitive to the length of the mixing phase, suggesting that this could have a simple fixed optimal value across all architectures and datasets. There are a number of paths one can take to improve on our optimiser, and we hope more work can be done in the future for our suggested proposals.

6. CONCLUSION

In this project, we investigated minimising the trade-off between generalisability and the speed of convergence that arises due to having to choose between ADAM and SGD. We proposed a new optimiser called MAWS that uniquely transition from ADAM to SGD in three distinct phases. Our unique transition consisted of an additional phase that allows for a continuous transition from ADAM to SGD. This additional phase addressed the issues associated with SWATS abrupt transition, such as the spikes in the gradient norm and the disruption it caused in learning. Not only that, but MAWS also addressed the issue with transitioning at the correct time to minimise the trade-off. This was done through adding two extra hyperparameters, allowing more control as to when the optimiser transitions into its respective phases.

Following this, we provided some intuition behind the behaviour of our new algorithm compared to existing competitors by visualising some toy examples. We also evaluated our optimiser alongside ADAM, SGD and SWATS on common benchmark datasets allowing us to understand how well our optimiser performed. Doing so provided some insights as to why SGD generalisability was unmatched for our MNIST experiment, which confirmed the hypothesis in [16], that SGDs generalisability is in part due to its superior ability to escape regions of high gradients such as local minima and ravines. The evaluation also allowed us to conclude that MAWS was better able to minimise the trade-off than SWATS. With this, we successfully achieved our initial aims of the project, which were to minimise the trade-off and gain a better insight into why SGD generalisation was superior to other optimisers.

There are several possible areas where future work can be conducted, we have highlighted these specific areas at the end of the last section. Firstly, due to the limited understanding of SGDs superior generalisability, it is extremely important that future work is concentrated in this area as a whole. This could improve our understanding of melded optimisers and why it is so beneficial to transition from ADAM to SGD. It could also help us gain further understanding around the transitioning mechanism, which would help in deciding which method is best. Although it is important to work on melded optimisers to improve the speed of convergence, the most essential attribute in our opinion is the generalisability. At the end of the day, the generalisability of the algorithm dictates how well the model will perform. As mentioned previously, luckily a lot of recent work has been conducted on understanding SGDs generalisability. But unfortunately, there is yet a complete satisfying answer that explains its superiority.

Secondly, as mentioned previously, due to the limitation of the extra hyperparameters for MAWS that control when the transitions occur. It would be very beneficial for future work to investigate other methods that would eliminate the need of extra hyperparameters. We believe that future work could concentrate on monitoring the gradient norms throughout training to help decide when to transition. This has also been suggested in [19], where they also motivate future work being conducted on studying the effects of different transition methods.

Lastly, more work needs to be done in evaluating MAWS on different datasets and architectures. Although our results are promising, we have not explored the efficiency of MAWS on optimising recurrent neural networks nor on tabular or audio data. Only when future work is conducted can we truly understand the benefits and drawbacks of MAWS.

References

[1] A. Cauchy. “M´ethode g´en´erale pour la r´esolution des syst`emes d’´equations simultan´ees”, C. R. Acad. Sci. Paris, 25:536–538, 1847.

[2] G. Hinton, O. Vinyals and J. Dean, “Distilling the Knowledge in a Neural Network”, arXiv:1503.02531, 2015.

[3] W. Sarle, “Neural Networks and Statistical Models”, 19th SAS Group International Conference, 1994.

[4] Adi Ben-Israel, “A Newton-Raphson method for the solution of systems of equations”, Journal of Mathematical Analysis and Applications, vol: 15, Issue 2. Pp 243-252, 1966.

[5] Y. Dauphin et al., “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, arXiv:1406.2572, vol: 1, 2014.

[6] H. Robbins and S. Monro, “A stochastic approximation method”, The annals of mathematical statistics, pp. 400–407, 1951.

[7] D. Kingma and J. Ba, “ADAM: A Method for Stochastic Optimization.”, CoRR vol: 1412.6980, 2015.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,”, Proceedings of the IEEE, vol: 86, no. 11, pp. 2278–2323, 1998.

[9] A. Krihevsky and G. Hinton, “Learning multiple layers of features from tiny images”, Citeseer, 2009.

[10] Y. LeCun et al., "Backpropagation Applied to Handwritten Zip Code Recognition," In Neural Computation, vol: 1, no. 4, pp. 541-551, 1989.

[11] P. Cheridito et al., “A proof of convergence for gradient descent in the training of artificial neural networks for constant target functions”, arXiv:2102.09924, vol: 1, 2021.

[12] S. Ruder, “An overview of gradient descent optimization algorithms”, arXiv:1609.04747, vol: 2, 2017.

[13] H. He, G. Huang and Y. Tuan, “Asymmetric Valleys: Beyond Sharp and Flat Local Minima”, NeurIPS, vol: 19, 2019.

[14] D. Rumelhart, G. Hinton and R. Williams, "Learning representations by back-propagating errors", Nature, vol: 323, pp. 533–536, 1986.

[15] J. Duchi, E. Haan and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.", Journal of machine learning research, vol: 12.7, 2011.

[16] P. Zhou, J. Feng et al., “Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning”, NeurIPS, vol: 34, 2020.

[17] N. Keskar and R. Socher, “Improving Generalization Performance by Switching from ADAM to SGD”, arXiv:1712.07628, vol: 1, 2017.

[18] N. Landro, I. Gallo and R. Grassa, “Mixing ADAM and SGD: a Combined Optimization Method”, arXiv:2011.08042, vol: 1, 2020.

[19] L. Luo, Y. Xiong, Y. Liu and X. Sun, "Adaptive Gradient Methods with Dynamic Bound of Learning Rate.", In International Conference on Learning Representations, 2018.

[20] I. Goodfellow, Y. Bengio and A. Courville, “Deep Learning”, 1st ed MIT Press, 2017.

[21] M. Staib, S. Reddi et al., “Escaping Saddle Points with Adaptive Gradient Methods”, In International Conference on Machine Learning, pp. 5956-5965, 2019.

[22] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition", In 2016 IEEE Conference on Computer Vision and Pattern Recognition, pp. 770-778, 2016.

[23] N. Keskar, D. Mudigere et al., “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”, In International Conference on Learning Representations, 2017.

[24] S. Arik and T. Pfister, “TabNet: Attentive Interpretable Tabular Learning”, arXiv:1908.07442, vol: 5, 2020.

Appendix

Algorithm 1: MAWS

Input: Loss function L($\theta_{k-1}; M$), *weights* θ_{k-1}, *learning rate* α, β_1, β_2, ϵ, ϕ_{ADAM}, ϕ_{MIX}, *total number of epochs* N, ρ.

$m_0 = 0$; $v_0 = 0$; $\lambda_0 = 0$; b = -1 ; k = 0.

$\eta_{ADAM} = N \cdot \phi_{ADAM}$; $\eta_{MIX} = N \cdot \phi_{MIX}$; $\eta_{SGD} = N - \eta_{ADAM} - \eta_{MIX}$.

LS = linearspace(start = 0, end = 1, size = η_{MIX})

function ADAM_step($\theta_{k-1}, g_{k-1}, \beta_1, \beta_2, \epsilon$)

  $m_k = m_{k-1} \cdot \beta_1 + (1 - \beta_1) \cdot g_{k-1}$

  $v_k = v_{k-1} \cdot \beta_2 + (1 - \beta_2) \cdot g_{k-1} \odot g_{k-1}$

  $\overline{m_k} = \dfrac{m_k}{1 - \beta_1^{\mathbf{k}}}$

  $\overline{v_k} = \dfrac{v_k}{1 - \beta_2^{\mathbf{k}}}$

  $\Delta_k^{ADAM} = \dfrac{\overline{m_k}}{\sqrt{\overline{v_k}} + \epsilon}$

  return Δ_k^{ADAM}

end function

function SGD_step($\theta_{k-1}, g_{k-1}, \rho$)

  $v_k = v_{k-1} \cdot \rho + (1 - \rho) \cdot g_{k-1}$

  return v_k

end function

for epochs = 1 to N **do**

  if epochs $< \eta_{ADAM}$ **then**

    phase = ADAM

  else if $\eta_{ADAM} \leq epochs < \eta_{MIX} + \eta_{ADAM}$ **then**

    b = b + 1

    phase = MIX

  else

    phase = SGD

  end if

  for batches **do**

    k = k + 1

    $g_{k-1} = \nabla_\theta L(\theta_{k-1}; M)$

    if phase = SGD **then**

      $\theta_k = \theta_{k-1} - \Lambda \cdot SGD\_step(\theta_{k-1}, g_{k-1}, \rho)$

      continue

    else if phase = ADAM **then**

      $\theta_k = \theta_{k-1} - \alpha \cdot ADAM\_step(\theta_{k-1}, g_{k-1}, \beta_1, \beta_2, \epsilon)$

    else

      $\mu_b^{ADAM} = 1 - LS[b]$

      $\mu_b^{SGD} = LS[b]$

      $\alpha^{melded} = \mu_b^{ADAM} \cdot \alpha + \mu_b^{SGD} \cdot \Lambda$

      Δ_k^{ADAM} = ADAM_Step($\theta_{k-1}, g_{k-1}, \beta_1, \beta_2, \epsilon$)

      $\Delta_k^{Melded} = \mu_b^{ADAM} \cdot \Delta_k^{ADAM} + \mu_b^{SGD} \cdot SGD\_Step(\theta_{k-1}, g_{k-1}, \rho)$

      $\theta_k = \theta_{k-1} - \alpha^{melded} \cdot \Delta_k^{Melded}$

    end if

    $p_k = \alpha \cdot ADAM\_step(\theta_{k-1}, g_{k-1}, \beta_1, \beta_2, \epsilon)$

    $\gamma_k = \frac{p_k^T p_k}{-p_k^T g_{k-1}}$

    $\lambda_k = \beta_2 \lambda_{k-1} + (1 - \beta_2)\gamma_k$

    $\Lambda = \frac{\lambda_k}{(1 - \beta_2^{\mathbf{k}})}$

  end for

end for

1

Algorithm 1: Pseudo code for the MAWS optimiser.