HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ADVANCED DEEP LEARNING

IT5422E

Recent SoTA Gradient Descent based Optimizers: Comparison and Evaluation

Team members:

Tran Thi Dinh - 20212255M

Le Hai Nam - 20212400M

Tran Thi Quyen - 20212247M

Supervisor:

Dr. Dinh Viet Sang

April, 2022

Contents

Chapter 1

Introduction

Deep learning has made substantial progress in recent years and has attracted a lot of interest from the research community and industry. It has been applied successfully to a wide range of problems ranging from image recognition, speech recognition, text classification, image generation, etc. However, there also exist many challenging problems in machine learning including minimization of loss (error) function, hyperparameters tuning, feature selection, dimensionality reduction, finding the optimum combination from a pool of base classifiers, etc. Optimization is a powerful and reliable tool for addressing some of these issues, and it has played a key role in statistical and deep learning since its inception. Deep learning models are increasingly reliant on optimization strategies in today's data-intensive technology era.

Gradient descent algorithms are the optimizers of choice for training deep neural networks, and generally change the weights of a neural network by following the direction of the negative gradient of some loss function with respect to the weight. While the classic, age-old gradient descent algorithms remain the most popular, there are empirical and theoretical reasons to consider newer alternatives. The advancement of optimization contributes significantly to the progress of machine learning. However, there are still many obstacles and unsolved problems in deep learning optimization. Improving optimization efficiency in deep neural networks with inadequate data is a challenging task. If there are not enough samples in the training of deep neural networks, significant variances and overfitting are likely to occur. Furthermore, one of the issues with deep neural networks has been non-convex optimization, which causes the optimization to produce a locally optimal solution rather than a global optimal one. When the sequence is too long for sequential models, the samples are frequently reduced by batches, resulting in deviation. It is critical to understand the deviation of

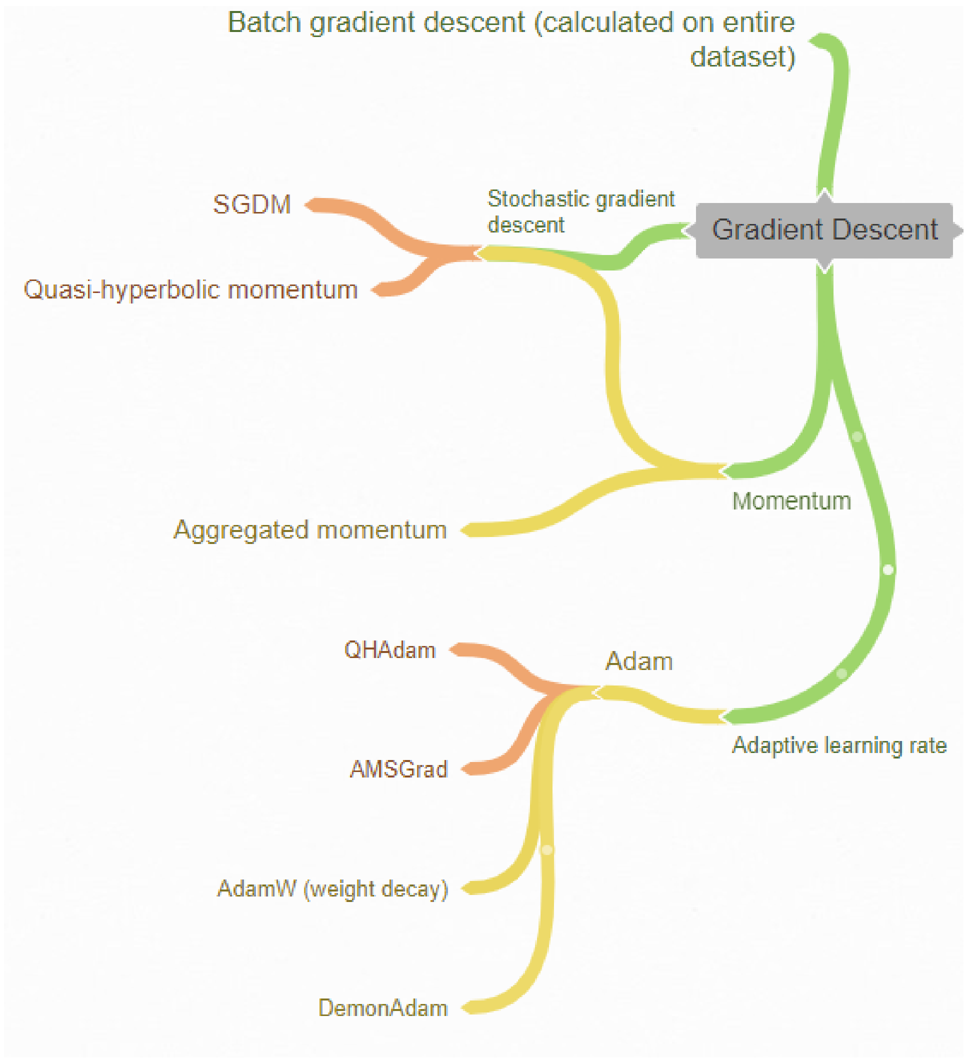


Figure 1.1: Overview of gradient descent optimization methods

stochastic optimization in this scenario. Thus, there are multiple challenges that need to be investigated and interpreted. Up to now, several optimizers have been proposed to tackle some specific challenges. The evolution of optimization methods can be summarized in Figure 1.1, which is represented by some famous algorithms. In this report, we research of some optimization methods that have been widely used in both research and practice. We detail the optimization algorithms in the next section, from the classical one to momentum and adaptive strategy improvements. In section 3, we show experiments on a variety of problems (image classification, text classification, and image generation) and provide some comments on the results. Finally, the conclusion of our research is presented in section 4.

In the rest of this report, we use some notations that are defined as follows:

- α: learning rate
- m: momentum buffer or velocity vector
- β: coefficient in momentum
- θ: weight/learnable parameter of model
- x: input data
- $L(x, \theta)$: loss function with input x and parameter θ
- g: gradient of loss function w.r.t θ, or ∇L

Chapter 2

Gradient descent optimization algorithms

2.1 Momentum (SGDM)

Momentum [12] is a commonly used technique. The main idea of momentum is to take advantage of the gradient information in the previous update step to motivate the model to overcome bad local points. In a word, momentum (or SGDM) is just adding a momentum term to SGD.

From the perspective of optimization by SGD: Consider two cases, a) the local loss landscape is a smooth hill and b) the local loss landscape is a steep ravine. As shown in figure 2.1. In the first case, even if it is clearly heading in the same direction, the gradient descent method may take a long time to reach the bottom of the hill. In the second one, the gradient descent algorithm may bounce back and forth between the steep ravine's walls without reaching the bottom quickly, and it would be fantastic if the back and forth gradients could be averaged to decrease variation. In a word, we are stuck at a local minimum.

These are the two common explanations, of why we need SGDM. SGDM is essentially an exponential moving average of past gradients parameterized by β (commonly equal to 0.9), and is given as follows:

$$\theta_t = \theta_{t-1} + \alpha m_{t-1}, \qquad m_t = \beta m_{t-1} - g_t \tag{2.1}$$

where

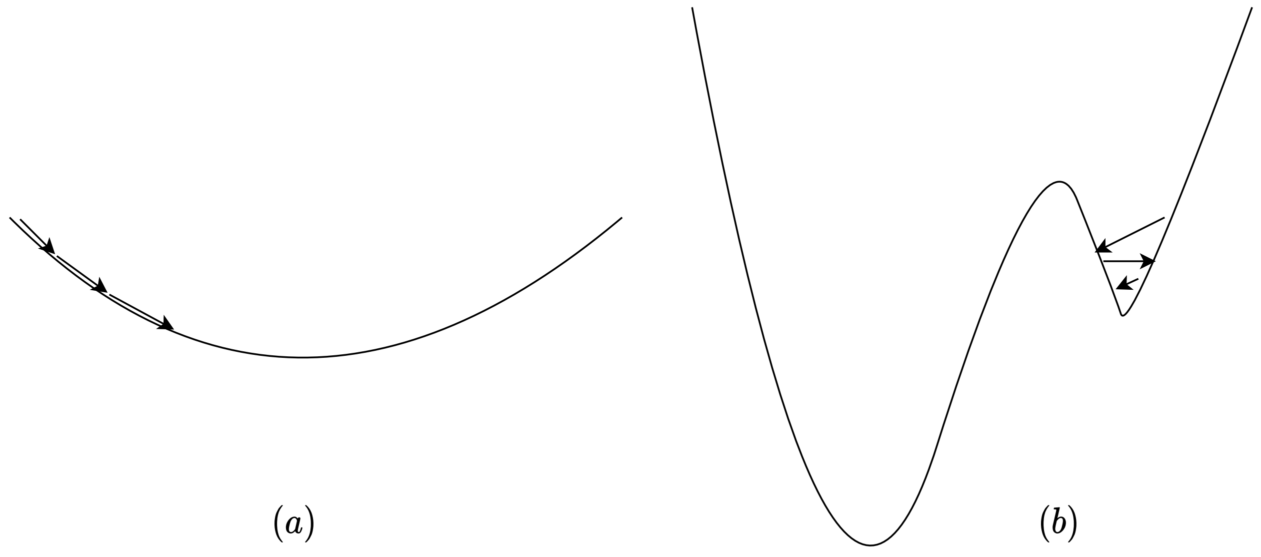


Figure 2.1: Motivation of SGDM

- Theta θ is the parameter
- Alpha α is learning rate.
- Beta β, a constant term. Also called the momentum coefficient.
- The term g_t is the gradient of loss function w.r.t θ at current time step.
- Last change (last update) to θ is called m_t. This includes the gradient information of previous update steps, called momentum.

The momentum term (m_t), which consists of a constant β and the prior update βm_{t-1} to θ, helps us accumulate additional speed for each iteration. However, the change to θ includes the second most recent update to θ, and so on. Essentially, momentum helps us to store the calculations of the gradients (the updates) for use in all the next updates to a parameter θ. This exact property causes "the ball" to roll faster down the hill (first case, i.e. we converge faster because now we move forward faster. And this also gives us the opportunity to overcome the saddle points like the second case above (Figure 2.2)

Although, as compared to SGD, Momentum allows the model to converge quicker, it does have significant disadvantages in some particular instances, as follows: How do we ensure we're not missing the local minima as the ball speeds down the hill? If the momentum is too strong, we will most likely miss the local minimum, rolling past it just to roll backwards and

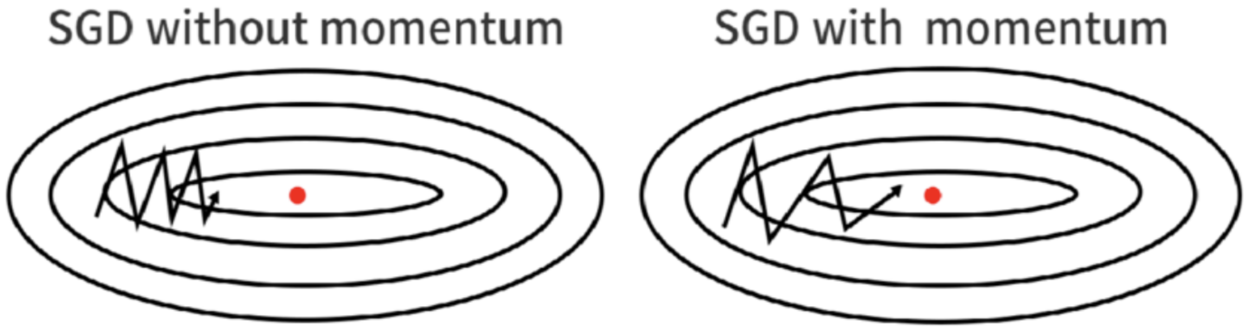


Figure 2.2: Momentum and not momentum

miss it again. If the momentum is too strong, we may just swing back and forth between the local minima.

2.2 Adam

Adaptive Moment Estimation is a method for optimizing stochastic objective functions using first-order gradients and adaptive estimates of lower-order moments. The method requires little memory, is insensitive to diagonal rescaling of the gradients, and is well suited to situations with a lot of data and/or parameters. The method can also be used to solve problems with non-stationary targets and/or very noisy and/or sparse gradients ([4]). It's essentially a hybrid of the 'gradient descent with momentum' and the 'RMSProp' algorithms (Tieleman & Hinton,2012).

Let $f(\theta)$ be a noisy objective function: a stochastic scalar function that is differentiable w.r.t. parameters θ. We are interested in minimizing the expected value of this function, $E[f(\theta)]$ w.r.t. its parameters θ. With $g_t = \nabla_\theta f_t(\theta)$ we denote the gradient, i.e. the vector of partial derivatives of f_t, w.r.t θ evaluated at timestep t.

The algorithm updates exponential moving averages of the gradient (m_t') and the squared gradient (v_t'), where the hyper-parameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages:

$$\begin{aligned} m_t &\leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &\leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{2.2}$$

The moving averages themselves are estimates of the first moment (the mean) and the second raw moment (the un-centered variance) of the gradient. However, these moving averages are initialized as (vectors of) 0's, leading to moment estimates that are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. the βs are close to 1). This can be addressed by using the fact that $\sum_{i=0}^{t} \beta^i = \frac{1-\beta^i}{1-\beta}$ to re-normalize terms. Correspondingly, the normalized state variables are given by:

$$\begin{aligned} \widehat{m_t} &\leftarrow m_t/(1-\beta_1^t) \\ \widehat{v_t} &\leftarrow v_t/(1-\beta_2^t) \end{aligned} \tag{2.3}$$

Then, the algorithm updates parameters as follow:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon} \tag{2.4}$$

2.3 AMSGrad

Adam integrates many strategies into a single learning algorithm that is both effective and efficient. As one might imagine, this method has grown in popularity as one of the more stable and successful optimization techniques for deep learning. The authors of "ON THE CONVERGENCE OF ADAM AND BEYOND" show, however, that even in basic one-dimensional convex situations, Adam can fail to converge to an optimal solution ([13]).

Consider the following simple sequence of linear functions for $\mathcal{F} = [-1, 1]$:

$$f_t(x) = \begin{cases} Cx & \text{t mode } 3 = 1 \\ -x & \text{otherwise} \end{cases}$$

Where $C > 2$. For this function sequence, it is easy to see that the point $x = -1$ provides the minimum regret. Suppose $\beta_1 = 0$ and $\beta_2 = 1/(1+C^2)$. They show that Adam converges to a highly suboptimal solution of $x = +1$ for this setting. Intuitively, the reasoning is as follows. The algorithm obtains the large gradient C once every 3 steps, and while the other 2 steps it observes the gradient -1, which moves the algorithm in the wrong direction. The large gradient C is unable to counteract this effect since it is scaled down by a factor of almost C for the given value of β_2, and hence the algorithm converges to 1 rather than -1.

They offer novel variations of the Adam algorithm that not only address the convergence concerns but also increase empirical performance by endowing such algorithms with "long-term memory" of prior gradients. The AMSGrad Algorithm is the name of their proposal.

AMSGrad uses a smaller learning rate in comparison to Adam and yet incorporates the intuition of slowly decaying the effect of past gradients on the learning rate as long as Γ_t is positive semi-definite. The key difference of AMSGrad with Adam is that it maintains the maximum of all v_t until the present time step and uses this maximum value for normalizing the running average of the gradient instead of v_t in Adam. AMSGrad neither increases nor decreases the learning rate and furthermore, decreases v_t which can potentially lead to non-decreasing learning rate even if gradient is large in the future iterations.

$$v_t \leftarrow \max(v_{t-1}, v_t) \tag{2.5}$$

Then, the algorithm updates parameters as follow:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \frac{m_t}{\sqrt{\widehat{v}_t} + \epsilon} \tag{2.6}$$

2.4 AdamW

For ordinary stochastic gradient descent (when rescaled by the learning rate), L2 regularization and weight decay regularization are identical, however this is not the case for adaptive gradient algorithms like Adam ([8]). The authors of the study "Decoupled weight decay regularization" explain that one of the reasons for the poor generalization of Adam, the most common adaptive gradient method, is because L2 regularization is not nearly as successful for it as it is for SGD. Their research of Adam, in particular, leads to the following conclusions:

L2 regularization and weight decay are not identical. For SGD, a reparameterization of the weight decay factor depending on the learning rate can make the two algorithms equal; however, this is not the case for Adam, as is sometimes ignored. L2 regularization, when paired with adaptive gradients, results in weights with significant historic parameter and/or gradient amplitudes being regularized less than when weight decay is used.

L2 regularization is not effective in Adam. One possible explanation why Adam and other adaptive gradient methods might be outperformed by SGD with momentum is that common deep learning libraries only implement L2 regularization, not the original weight decay. Therefore, on tasks/datasets where the use of L2 regularization is beneficial for SGD (e.g., on many popular image classification datasets), Adam leads to worse results than SGD with momentum (for which L2 regularization behaves as expected)

Weight decay is equally effective in both SGD and Adam. For SGD, it is equivalent to L2 regularization, while for Adam it is not.

Looking first at the case of SGD, the authors propose to decay the weights simultaneously with the update of θ_t based on gradient information.

SGD with L2 regularization	SGD with decoupled weight decay
$g_t \leftarrow \nabla L_t(\theta_{t-1}) + \lambda\boldsymbol{\theta_{t-1}}$	$g_t \leftarrow \nabla L_t(\theta_{t-1})$
$m_t \leftarrow \beta_1 m_{t-1} + \alpha g_t$	$m_t \leftarrow \beta_1 m_{t-1} + \alpha g_t$
$\theta_t \leftarrow \theta_{t-1} - m_t$	$\theta_t \leftarrow \theta_{t-1} - m_t - \lambda\boldsymbol{\theta_{t-1}}$

This yields their proposed variant of SGD with momentum using decoupled weight decay (SGDW). This simple modification explicitly decouples λ and α (although some problem-dependent implicit coupling may of course remain, as for any two hyperparameters).

They decouple weight decay and loss-based gradient updates in Adam as shown below; this gives rise to their variant of Adam with decoupled weight decay (AdamW).

Adam with L2 regularization	Adam with decoupled weight decay
$g_t \leftarrow \nabla L_t(\theta_{t-1}) + \lambda\boldsymbol{\theta_{t-1}}$	$g_t \leftarrow \nabla L_t(\theta_{t-1})$
$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$	$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$	$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
$\widehat{m_t} \leftarrow m_t/(1-\beta_1^t)$	$\widehat{m_t} \leftarrow m_t/(1-\beta_1^t)$
$\widehat{v_t} \leftarrow v_t/(1-\beta_2^t)$	$\widehat{v_t} \leftarrow v_t/(1-\beta_2^t)$
$\theta_t \leftarrow \theta_{t-1} - \alpha\widehat{m_t}/(\sqrt{\widehat{v_t}} + \epsilon)$	$\theta_t \leftarrow \theta_{t-1} - \alpha\widehat{m_t}/(\sqrt{\widehat{v_t}} + \epsilon) - \lambda\boldsymbol{\theta_{t-1}}$

Having shown that L2 regularization and weight decay regularization differ for adaptive gradient algorithms raises the question of how they differ and how to interpret their effects. Their equivalence for standard SGD remains very helpful for intuition: both mechanisms push weights closer to zero, at the same rate.

However, for adaptive gradient algorithms they differ: with L2 regularization, the sums of the gradient of the loss function and the gradient of the regularizer (i.e., the L2norm of the weights) are adapted, whereas with decoupled weight decay, only the gradients of the loss function are adapted (with the weight decay step separated from the adaptive gradient mechanism).

With L2 regularization both types of gradients are normalized by their typical (summed) magnitudes, and therefore weights x with large typical gradient magnitude s are regularized by a smaller relative amount than other weights. In contrast, decoupled weight decay regularizes all weights with the same rate λ, effectively regularizing weights x with large s more than standard L2 regularization.

2.5 Aggregated momentum (AggMo)

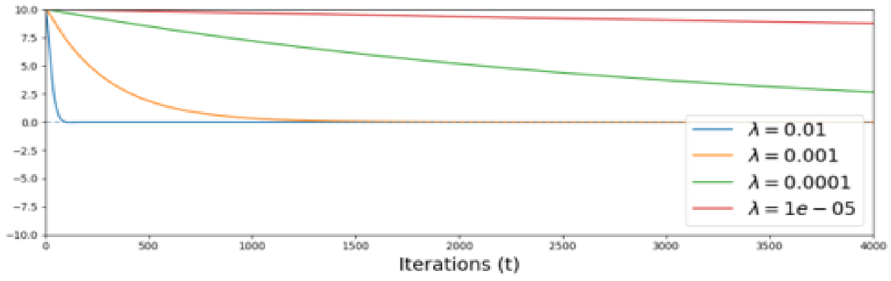
Aggregated momentum ([9]) is a variant of momentum which combines multiple velocity vectors with different damping coefficients to take advantage of both small and large damping coefficients. The motivation of AggMo starts from classical momentum.

Classical momentum (CM): consider a function $f : R^d \to R$ to be minimized with respect to some variable θ. CM minimizes this function by taking some initial point θ_0 and running the following iterative update steps:

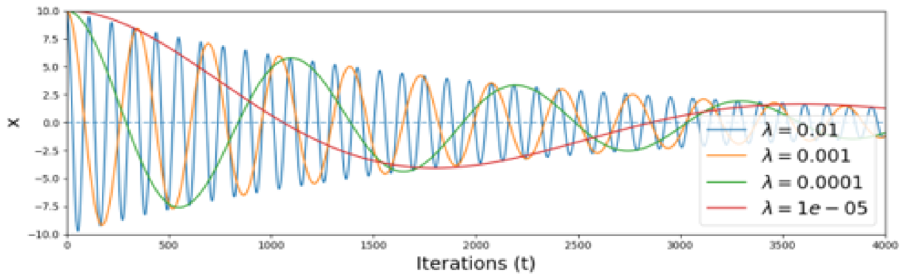
$$\begin{aligned} m_t &\leftarrow \beta m_{t-1} - \nabla L_{t-1}(\theta_{t-1}) \\ \theta_t &\leftarrow \theta_{t-1} - \alpha_t m_t \end{aligned} \tag{2.7}$$

Where m is called velocity vector/momentum buffer, α_t denotes a learning rate schedule, β is the damping coefficient and $v_0 = 0$. It is difficult to choose a suitable β. The choice of β imposes a tradeoff between speed and stability: small β gives stability in the optimizing process but can significantly increase training time, whereas large β values can potentially deliver much larger speed-ups, but are prone to oscillations and instability.

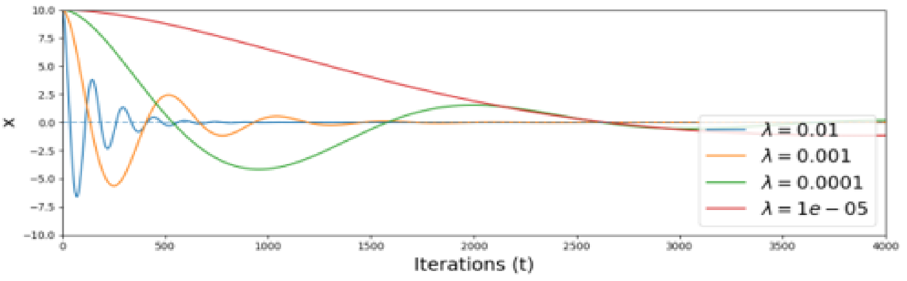
From those perspectives, AggMo try to find a way to dampen the oscillations while preserving the high terminal velocity of large beta values could dramatically speed up optimization. To resolve this issue, AggMo takes a linear combination of multiple momentum buffers. Specifically, it modifies the gradient descent algorithm by averaging several velocity vectors each with their own damping coefficient. AggMo's updated iterative procedure can be written



(a) CM ($\beta = 0.9$)



(b) CM ($\beta = 0.999$)



(c) AggMo ($\beta = [0, 0.9, 0.99, 0.999]$)

Figure 2.3: Minimizing a quadratic function $f = \lambda x^2$. All optimizers use a fixed learning rate of 0.33

as follows:

$$m_t^{(i)} \longleftarrow \beta^{(i)} m_t^{(i-1)} - \nabla_\theta L_{t-1}(\theta_{t-1}), \qquad \forall i = 1, ...K$$
$$\theta_t \longleftarrow \theta_{t-1} + \frac{\alpha_t}{K} m_t^{(i)} \tag{2.8}$$

Where $v_0^{(0)} = 0$ for each I and the vector $\beta = [\beta^{(1)}, \beta^{(2)}, ..., \beta^{(K)}]$ as the *damping vector*. The authors suggest choosing $\beta^{(i)} = 1 - \alpha^{i-1}$ with $\alpha = 0.1$. AggMo maintains K velocity vectors, each with a different damping coefficient, and averages them for the update at every timestep t. By taking advantage of several damping coefficients, AggMo can optimize well over ill conditioned curvature. This evidence can be observed in figure 2.3 that compares AggMo and Classical momentum in the optimization of several quadratic functions.

In figure 2.3, (a) and (b) show the results of CM using a small ($\beta = 0.9$) and a high damping coefficient ($\beta = 0.999$) respectively. We can see that when using a low damping coefficient, it takes many iterations to find the optimal solution. On the other hand, increasing the damping coefficient from 0.9 to 0.999 causes oscillations which prevent convergence. Figure 2.3 (c) shows that AggMo dampens oscillations quickly for all functions and converges faster than CM in the case of large β. It turns out that AggMo may converge slower in functions with low curvature ($\lambda = 0.01 and \lambda = 0.001$) but the gap with CM is negligible (just a few iterations), while it speeds up the training time remarkably in functions with high curvature.

Figure 2.4 displays the AggMo velocities during optimization. At point (1) the velocities are aligned towards the minima, with the $\beta = 0.999$ velocity contributing substantially more to each update. By point (2) the system has begun to oscillate. While the $\beta = 0.999$ velocity is still pointed away from the minima, the $\beta = 0.9$ velocity has changed direction and is damping the system. Combining the velocities allows AggMo to achieve fast convergence while reducing the impact of oscillations caused by large β values.

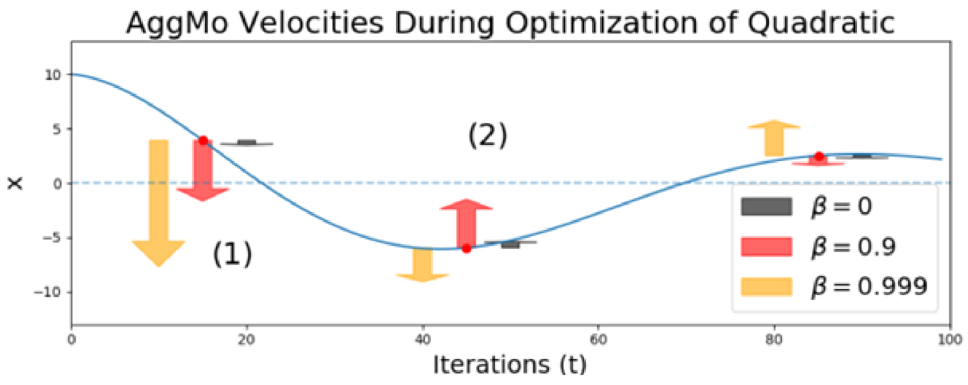


Figure 2.4: Breaking oscillations with passive damping. The arrows show the direction and relative amplitude. Source [9]

Computational/Memory cost: When compared to CM, AggMo has a very low additional computational cost because it just takes a few extra addition and multiplication operations on top of the single gradient evaluation. Due to store the K velocity vectors, there is some memory overhead.

AggMo is an efficient method that able to remain stable even with large damping coefficients and enjoys faster convergence rates. It also helps to reduce the difficulty in choosing right damping coefficient which is efficient in reducing hand-crafted effect. Moreover, the computational and memory overheads are still acceptable to use in practical.

2.6 Quasi-hyperbolic momentum (QHM)

Quasi-Hyperbolic momentum ([10]) is an adaptive momentum algorithm which decouples the momentum term from the current gradient when updating the weights. In other words, it is a weighted average of the momentum and Stochastic gradient descent (SGD). Its motivation starts with gradient variance reduction from momentum. We have updated iterative procedure of SGD and Momentum at time step t as follows:

- SGD:

$$\theta_{t+1} \leftarrow \theta_t - \alpha L_t(\theta_t) \tag{2.9}$$

- Momentum:

$$\begin{aligned} m_{t+1} &\leftarrow \beta m_t - (1-\beta)\nabla L_t(\theta_t) \\ \theta_{t+1} &\leftarrow \theta_t - \alpha m_{t+1} \end{aligned} \tag{2.10}$$

Where θ is learnable variable, α denotes the learning rate, L is the loss function, m is momentum buffer and β is discount factor. Note that $\beta = 0$ recovers plain SGD, and it also controls how slowly the momentum buffer is updated (the larger β is, the slower it will be to update). From the equation 2.10, it can be rewritten as:

$$\theta_{t+1} \leftarrow \theta_t - \alpha \underbrace{\left[(1-\beta)\sum_{i=0}^{t} \beta^i \nabla L_{t-i}(\theta_{t-i})\right]}_{Y} \tag{2.11}$$

The variance during updated procedure is the variance of Y element:

$$Cor[Y] = (1-\beta)^2 \sum_{i=0}^{t} \beta^{2i} Cor[L] = (1-\beta)^2 \frac{1-\beta^{2t}}{1-\beta^2} Cor[L] \tag{2.12}$$

When $t \to \infty$,

$$Cor[Y] = lim_{t\to\infty}(1-\beta)^2 \frac{1-\beta^{2t}}{1-\beta^2} Cor[L] = \frac{(1-\beta)^2}{1-\beta^2} Cor[L] \tag{2.13}$$

We can see that, increase β can reduce the $Cor[Y]$. However, higher β induces more bias in the momentum buffer with respect to the true gradient, as the momentum buffer becomes extremely slow to update. Thus, QHM aims to achieve variance reduction while guaranteeing that recent gradients contribute significantly to the update step. To do that they weigh the current gradient with an immediate discount factor ρ. So that it can reduce bias with respect to the true gradient. The update rule of QHM is introduced as follows:

$$\begin{aligned} &m_{t+1} \leftarrow \beta m_t - (1-\beta)\nabla L_t(\theta_t), \\ &\theta_{t+1} \leftarrow \theta_t - \alpha[(1-\rho)\nabla L_t(\theta_t) + \rho m_{t+1}] \end{aligned} \tag{2.14}$$

QHM can be seen as a v-weighted average of the momentum update step and the SGD update step. It recovers momentum and SGD when $v = 1$ and, $v = 0$ respectively. The equation 2.14 can also be written as:

$$\begin{aligned} \theta_{t+1} &\leftarrow \theta_t - \alpha \left[(1-\rho)\nabla L_t(\theta_t) + \rho(1-\beta)\sum_{i=0}^{t} \beta^i \nabla L_{t-i}(\theta_{t-i})\right] \\ &\leftarrow \theta_t - \alpha \underbrace{\left[(1-\rho\beta)\nabla L_t(\theta_t) + \rho(1-\beta)\sum_{i=0}^{t} \beta^i \nabla L_{t-i}(\theta_{t-i})\right]}_{Y} \end{aligned} \tag{2.15}$$

The variance then is calculated as:

$$\begin{aligned} Cor[Y] &= lim_{t\to\infty}\left[(1-\rho\beta)^2 Cor[L] + \rho^2(1-\beta)^2 \frac{\beta^2(1-\beta^{2t})}{1-\beta^2} Cor[L]\right] \\ &= \left[(1-\rho\beta)^2 + \frac{[\rho\beta(1-\beta)]^2}{1-\beta^2}\right] Cor[L] \end{aligned} \tag{2.16}$$

It can be verified that $Cor[Y]$ decreases (thus inducing variance reduction) with both increasing β and increasing ρ. Therefore, we can increase β to reduce variance and use v to mitigate the staleness of bias to last gradient. The authors recommend $\rho = 0.7$ and $\beta = 0.999$ as a good starting point.

2.7 Quasi-Hyperbolic Adam (QHAdam)

QHAdam is an adaptive learning rate extension of QHM to replace both momentum estimators in Adam with quasi-hyperbolic terms. Namely, QHAdam decouples the momentum term from the current gradient when updating the weights, and decouples the mean squared gradients term from the current squared gradient when updating the weights. In other words, it is a weighted average of the momentum and SGD, weighting the current gradient with an immediate discount factor ρ, divided by a weighted average of the mean squared gradients and the current squared gradient, weighting the current squared gradient with an immediate discount factor ρ_2. The update rule of QHAdam is as follows:

$$\begin{aligned} m_{t+1} &\leftarrow \beta_1 m_t - (1-\beta_1)\nabla L_t(\theta_t) \qquad m'_{t+1} \leftarrow (1-\beta_1^{t+1})^{-1} m_{t+1} \\ s_{t+1} &\leftarrow \beta_2 s_t - (1-\beta_2)(\nabla L_t(\theta_t))^2 \qquad s'_{t+1} \leftarrow (1-\beta_2^{t+1})^{-1} s_{t+1} \\ \theta_{t+1} &\leftarrow \theta_t - \alpha \left[\frac{(1-\rho_1)\nabla L_t(\theta_t) + \rho_1 m'_{t+1}}{\sqrt{(1-\rho_2)(\nabla L_t(\theta_t))^2 + \rho_2 s'_{t+1}} + \epsilon} \right] \end{aligned} \tag{2.17}$$

QHAdam can recover several variants of Adam algorithm. QHAdam recovers Adam when $\rho_1 = \rho_2 = 1$. Moreover, QHAdam recovers RMSProp when $\rho_1 = 0$ and $\rho_2 = 1$ and NAdam when $\rho_1 = \beta_1$ and $\rho_2 = 1$.

2.8 Demon

Decaying the learning rate is widely known as a technique that helps the network converge to a local minimum and avoid oscillation. Inspired by this, John Chen et al. proposed a new method called decay momentum (DEMON) [1], which reduces the impact of a gradient on current and future updates. This technique helps to improve model performance and hyperparameter robustness with simple techniques for the momentum parameter.

DEMON gives a rule for decay the momentum coefficient β, according to the following formula:

$$\beta_t = \beta_{init} \frac{1 - t/T}{(1-\beta_{init}) + \beta_{init}(1 - t/T)} \tag{2.18}$$

where $(1 - t/T)$ is the proportion of iterations remaining.

Formula 2.18 is derived from the following analysis: To minimize the objective function $\mathcal{L}(\theta)$, the most common momentum method, SGDM, is given by the following recursion (m_t is momentum term):

$$\theta_{t+1} = \theta_t + \alpha m_t, \qquad m_t = \beta m_{t-1} - g_t$$

Then, the main recursion can be unrolled into:

$$\theta_{t+1} = \theta t - \alpha g_t - \alpha\beta g_{t-1} - \alpha\beta^2 g_{t-2} + \alpha\beta^3 v_{t-2} = ... = \theta_t - \alpha g_t - \alpha \sum_{i=1}^{t} (\beta^i g_{t-i}) \qquad (2.19)$$

We can see that, a particular gradient term g_t contributes a total of $\alpha \sum_i \beta^i$ of its "energy" to all future gradient updates, and this results in the geometric sum:

$$\sum_{i=1}^{\infty} \beta^i = \beta \sum_{i=1}^{\infty} \beta^i = \frac{\beta}{1-\beta}$$

The authors aimed to design a schedule such that the cumulative momentum is decayed to 0. That is, at current step t with total T step, $\beta = \beta_{init}$:

$$\frac{\beta_t}{1-\beta_t} = (1 - t/T)\frac{\beta_{init}}{1-\beta_{init}}$$

This lead to (2.18).

Demon uses momentum to speed up in the early stages of training, but it depletes momentum as the training progresses. This keeps the weights of neural networks from rising too rapidly. The use of restraint weights has lately been investigated and shown to be crucial for optimal performance. For instance, [24] shows that simply using momentum leads to inferior performance, both conceptually and experimentally. It's critical to maintain a consistent rate of weight gain throughout time. Figure 2.5 presents a visualization of the proposed momentum decay rule and other common schedules.

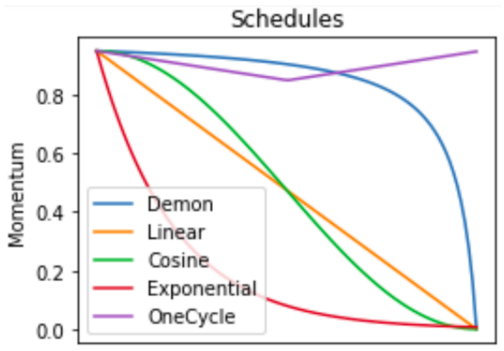
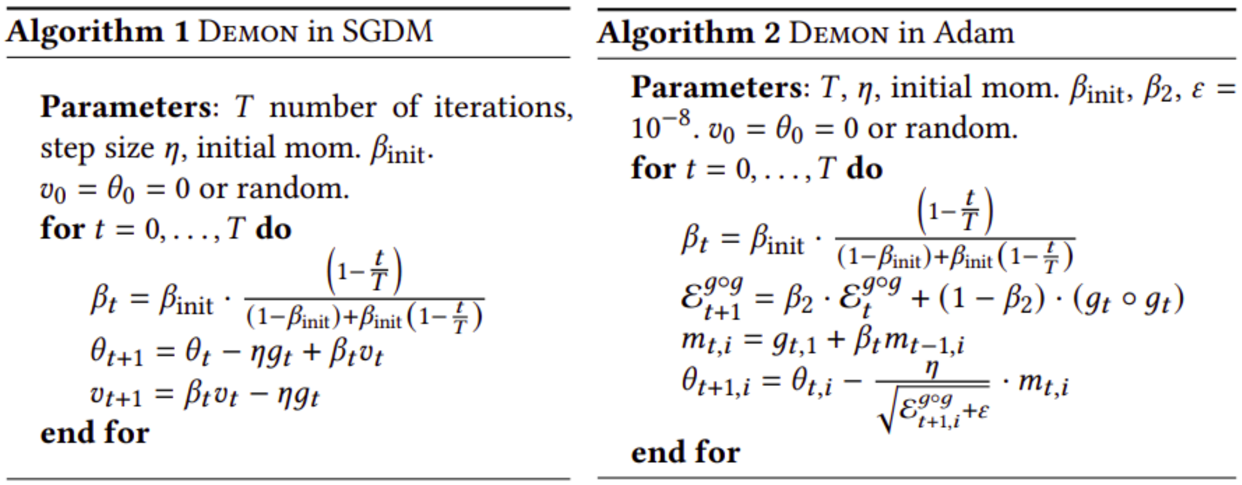


Figure 2.5: Non-linear DEMON schedule vs. and other schedules, from $\beta_{init} = 0.9$ to 0.

Demon can be applied to any gradient descent algorithm with a momentum parameter. The implementation is simply 1-2 lines of code before compute gradient. Thus, this technique requires only limited extra overhead and computation in comparison to the vanilla counterparts for the computation of β_t:

```
 p_t = (iters - t) / iters
beta_t = beta1 * (p_t / (1 - beta1 + beta1 * p_t))
```

The pseudocode for applying DEMON to SGDM and Adam (from the DEMON's paper) is in figure 2.6. The author also show that improved robustness to hyperparameter tuning for Demon relative to the popular learning rate step schedule.



Algorithm 1 Demon in SGDM

```
Parameters: $T$ number of iterations, step size $\eta$, initial mom. $\beta_{\text{init}}$.
$v_0 = \theta_0 = 0$ or random.
for $t = 0, \dots, T$ do
    $\beta_t = \beta_{\text{init}} \cdot \frac{\left(1-\frac{t}{T}\right)}{(1-\beta_{\text{init}})+\beta_{\text{init}}\left(1-\frac{t}{T}\right)}$
    $\theta_{t+1} = \theta_t - \eta g_t + \beta_t v_t$
    $v_{t+1} = \beta_t v_t - \eta g_t$
end for
```

Algorithm 2 Demon in Adam

```
Parameters: $T$, $\eta$, initial mom. $\beta_{\text{init}}$, $\beta_2$, $\varepsilon = 10^{-8}$. $v_0 = \theta_0 = 0$ or random.
for $t = 0, \dots, T$ do
    $\beta_t = \beta_{\text{init}} \cdot \frac{\left(1-\frac{t}{T}\right)}{(1-\beta_{\text{init}})+\beta_{\text{init}}\left(1-\frac{t}{T}\right)}$
    $\mathcal{E}_{t+1}^{g\circ g} = \beta_2 \cdot \mathcal{E}_t^{g\circ g} + (1-\beta_2)\cdot(g_t \circ g_t)$
    $m_{t,i} = g_{t,1} + \beta_t m_{t-1,i}$
    $\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\mathcal{E}_{t+1,i}^{g\circ g}+\varepsilon}} \cdot m_{t,i}$
end for
```

Figure 2.6: Apply DEMON to SGDM and Adam (η is our α)

In short, Demon is a decay momentum technique that can be effectively applied to any momentum based strategy (SGDM, Adam, etc.). Compared to other learning rate schedules and momentum schedules, Demon achieves outstanding results. This includes improvements over the popular learning rate step schedule, cosine decay schedule, OneCycle, and many others. Demon is computationally cheap, understand, and easy to implement.

Chapter 3

Experiments and results

3.1 Image classification

3.1.1 Experimental setup

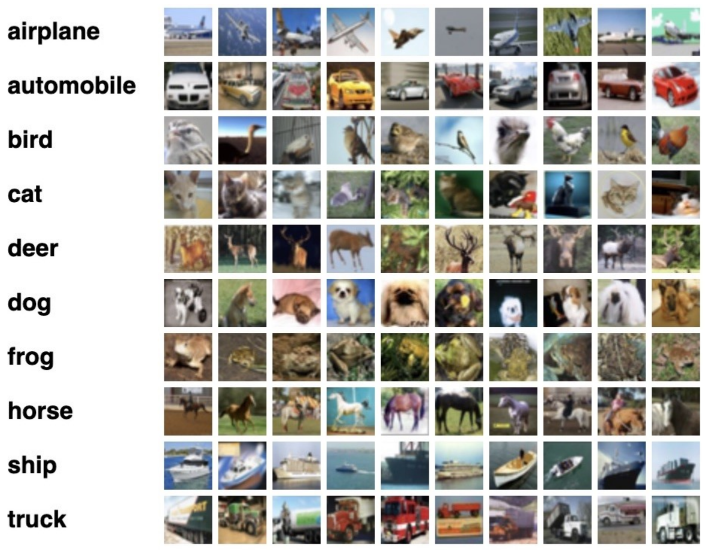


Figure 3.1: CIFAR-10 dataset

In this experiment, we examine the optimizers to classify image into some categories. We use CIFAR-10 dataset ([5]) in this experiment with 10 categories to classify. The dataset contains 50000 training images and 10000 test images. For all optimizer, we use 10000 images

Hyper-parameters	Range
Learning rate	**[0.0005, 0.001, 0.005, 0.008, 0.01, 0.05, 0.08, 0.1, 0.5]**
SGDM: β_1	**[0.8, 0.85, 0.9, 0.95, 0.99, 0.999]** - LeNet: $\beta_1 = 0.85, lr = 0.01$ - ResNet: $\beta_1 = 0.85, lr = 0.1$
Adam: β_1	- LeNet: $\beta_1 = 0.9, lr = 0.001$ - ResNet: $\beta_1 = 0.8, lr = 0.005$
AMSGrad: β_1	- LeNet: $\beta_1 = 0.9, lr = 0.001$ - ResNet: $\beta_1 = 0.8, lr = 0.01$
QHM: β	- LeNet: $\beta = 0.9, lr = 0.008$ - ResNet: $\beta = 0.9, lr = 0.5$
QHAdam: β_1	- LeNet: $\beta_1 = 0.9, lr = 0.001$ - ResNet: $\beta_1 = 0.8, lr = 0.005$
Demon-Adam: β_1	- LeNet: $\beta_1 = 0.85, lr = 0.001$ - ResNet: $\beta_1 = 0.9, lr = 0.005$
Demon-SGDM: β	- LeNet: $\beta = 0.95, lr = 0.001$ - ResNet: $\beta = 0.95, lr = 0.1$
AggMo: number of β	**[2, 3, 4, 5, 6]** - LeNet: num of $\beta = 2, lr = 0.0008$ - ResNet: num of $\beta = 2, lr = 0.08$
AdamW: weight decay	**[0.0001, 0.0002, 0.0005, 0.001, 0.002, 0.005]** - LeNet: $weight\_decay = 0.0005, lr = 0.0005$ - ResNet: $weight\_decay = 0.001, lr = 0.005$

Table 3.1: Tuned hyper-parameters range for image classification

of training set as validation set to select the best model's hyper-parameters based on the F1 score on this set. We use two convolutional networks in this experiment, one with a small number of parameters (LeNet) and one with a large number of parameters (ResNet18 - [3]). We set the number of epochs to 100 and batchsize to 128 in every experiment. Grid search

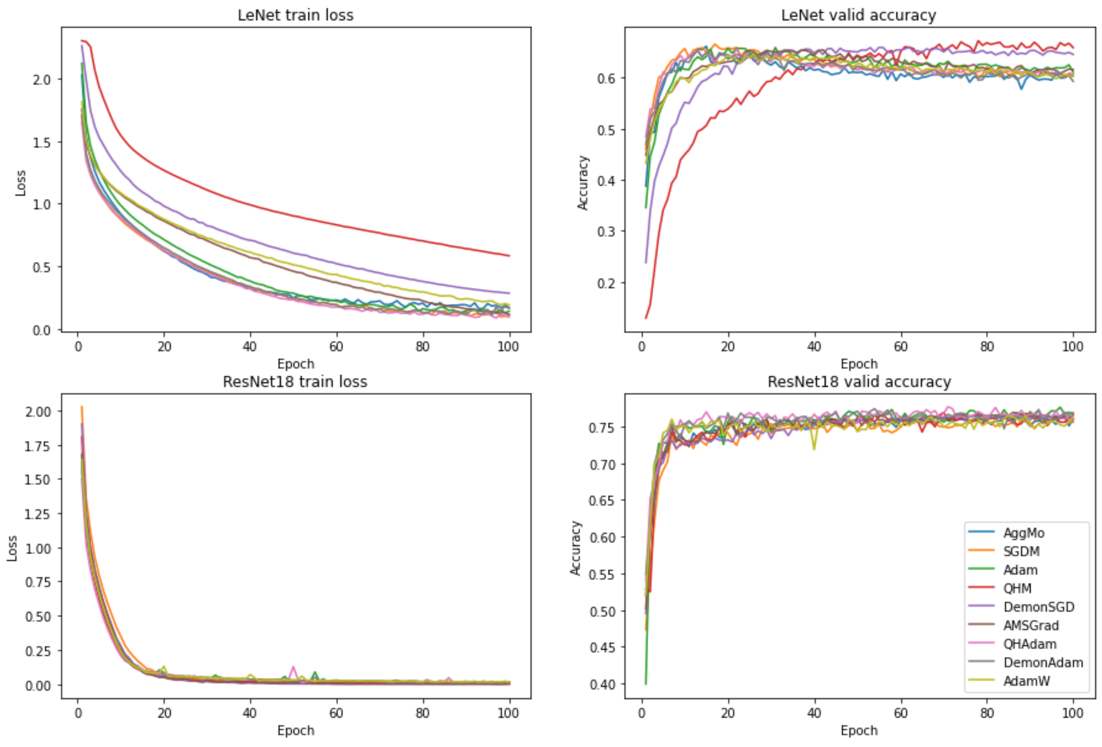


Figure 3.2: Train loss and valid accuracy on CIFAR-10

is adopted to find the best hyper-parameters, we adjust learning rate and an important parameter of each optimizer, the remaining parameters are fixed with the recommended value. No augmentation techniques or fine tune pretrained model are adopted in our experiment. The ranges for tuning parameters are shown in Table

3.1.2 Results

The results on test and validation sets are revealed in Table 3.2 and Figure 3.2 depicts the training loss and valid accuracy values during training. It can be witnessed that the results provided by the optimizers are comparative.

In the experiment of large network, all algorithms have a similar performance while there are more difference in the case of small network. Most algorithms seem to experience overfitting around 10 epochs (valid accuracy decreases in the subsequent epochs), except for QHM and DemonSGD. The loss on training set of QHM have the highest value, however it generalizes

CIFAR-10	LeNet				ResNet			
	Valid acc	Valid F1	Test acc	Test F1	Valid acc	Valid F1	Test acc	Test F1
SGDM	65.600	65.325	65.260	65.152	74.020	73.615	73.110	72.847
Adam	65.700	65.453	64.160	64.123	75.160	74.954	75.070	74.985
AggMo	65.560	65.031	65.490	65.109	72.760	72.367	71.990	71.641
QHM	**67.220**	**66.638**	**65.860**	**65.523**	74.320	74.064	73.140	73.000
DemonSGD	65.440	65.350	65.070	65.118	73.100	72.627	73.280	73.118
AMSGrad	65.020	64.684	64.700	64.700	75.120	74.793	73.660	72.512
QHAdam	65.340	64.953	64.600	64.554	75.560	75.672	**75.130**	**75.428**
DemonAdam	65.280	64.847	65.270	64.995	75.260	75.106	74.200	74.235
AdamW	64.540	64.081	63.110	62.755	**76.000**	**75.885**	75.030	75.081

Table 3.2: Results of validation and test sets on CIFAR-10

better for validation set. Unlike other algorithms, the accuracy score of QHM and Demon-SGD starts from a low value (especially QHM), however the trend is up but not down, this trend is also observed in the case of large network. It may indicate that the stability and generalization of these algorithms in this experiment to reduce overfitting.

second/epoch	LeNet	ResNet
SGDM	6.992	15.205
Adam	7.135	16.079
AggMo	6.966	16.027
QHM	7.111	16.006
DemonSGD	7.149	15.622
AMSGrad	7.313	15.965
QHAdam	7.277	16.959
DemonAdam	8.533	16.660
AdamW	8.294	16.723

Table 3.3: Time complexity of optimizers on image classification

Table 3.2 details the performance of the optimizers. Recent approach AdamW shows ineffectively in the small network, while QHM and its variant of Adam achieve highest F1 score in both networks. In addition, AggMo seem to perform worse than SGDM in large network experiments, even though it is an improvement of momentum. This can be explained as AggMo can probably help in speeding up training time, however it does not make sure that it can find a better local minimum than SGDM. Since many algorithms are tuned with the same range (Table 3.1) for fair comparison, we also visualized all search results to analyze

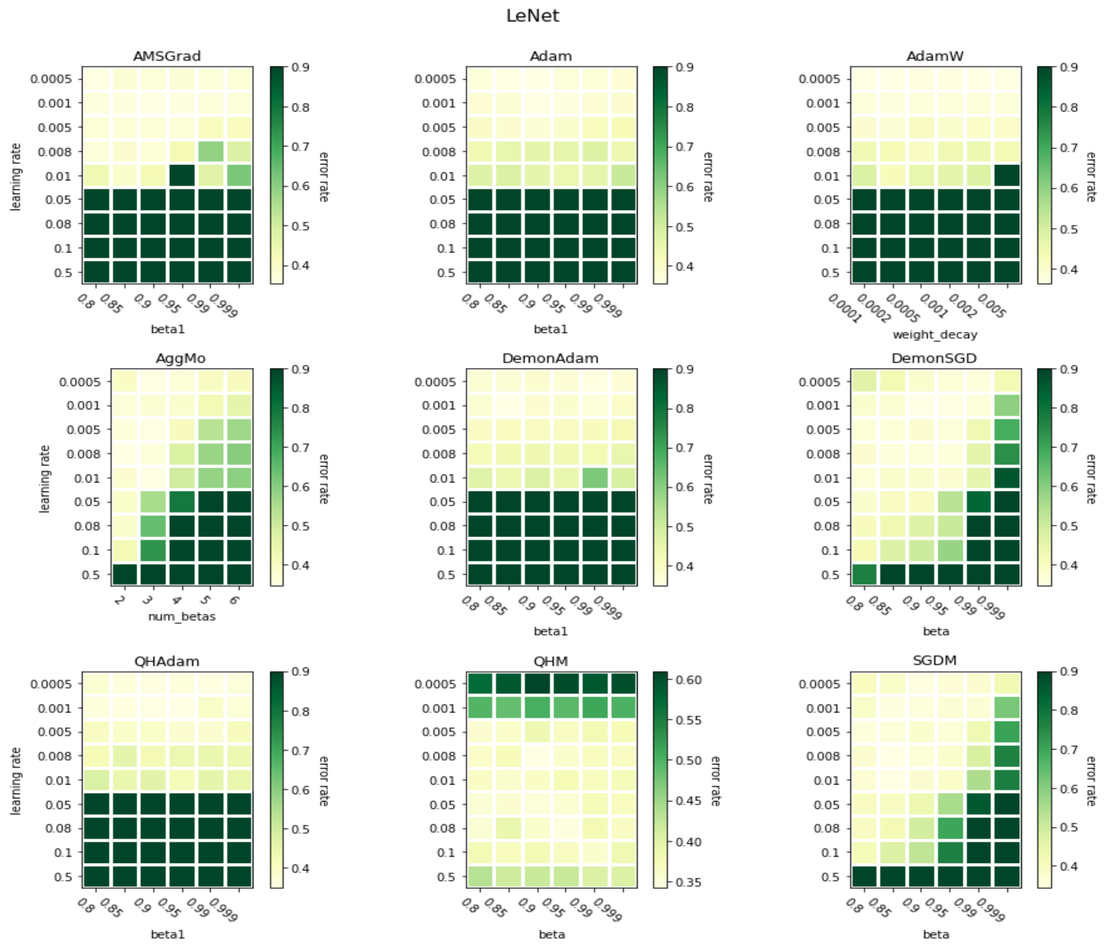


Figure 3.3: Error rate of all search parameters setting on LeNet

the sensitive aspect of the algorithm in Figure 3.3 and 3.4. We find that all methods using adaptive learning rate are sensitive with the large value of initial learning rate in the experiment of LeNet. This figure also reveals that SGD based methods with no adaptive learning rate perform worse with high value of β, since large β can cause instability in the training process. Moreover, increasing the number of β does not improve the performance of AggMo and QHM performs well with high value of learning rate while other algorithms do not. The result of QHM can be explained as the technique of variance reduction that allows the use of high learning rate. We also investigate the time complexity of optimizers which are illustrated in Table 3.3. It is obvious that classical method SGDM is the fasttest algorithm, and algorithms with adaptive learning rate combined with momentum take more time to update parameters in an epoch.

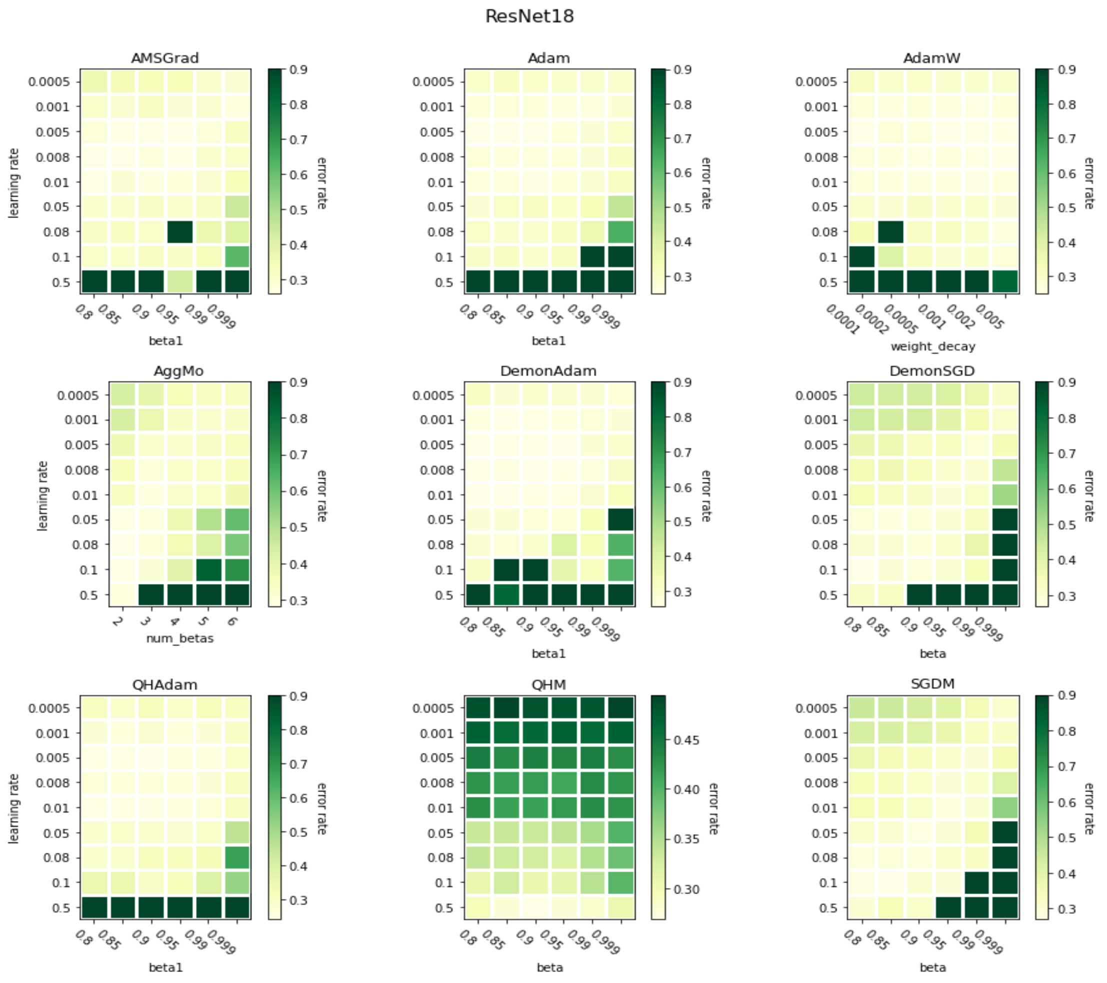


Figure 3.4: Error rate of all search parameters setting on ResNet18

3.2 Text classification

3.2.1 Experimental setup

In this subsection, we use the IMDb dataset ([7]) to investigate the optimization algorithms. The core dataset contains 50,000 reviews split evenly into 25k train and 25k test sets. The overall distribution of labels is balanced (25000 positive and 25000 negative). For all optimizers, we use two model in this experiment. The first model is a simple model that includes 1 LSTM, 1 Dropout and 2 Dense layers. For the word embeddings, we use the 100-dimensional GloVe embeddings of 400k words computed on a 2014 dump of English Wikipedia ([11]). We empirically set the max number of the epoch to 50 and save the best model on the validation set for testing. The second model is BERT base model (cased) ([2]) and trained with 10 epochs. We set the batchsize to 128 in every experiment. Table 3.4 shows the learning rate value of each optimizer, corresponding.

Optimizer	Leaning rate used in training process	
	LSTM	BERT
Adam	1e-3	1e-5
AMSGrad	1e-3	2e-5
AdamW	1e-3	1e-5
AggMo	1e-3	2e-5
QHM	2e-2	1e-3
QHAdam	1e-3	1e-5
DemonSGD	1e-3	2e-4
DemonAdam	1e-3	1e-5
SGDM	1e-3	2e-4

Table 3.4: Learning rate values in text classification models

3.2.2 Results

The figure 3.5 depicts the values of loss and accuracy during model training. In general, the optimal algorithms of non-adaptive learning rate group (SGDM, AggMo, QHM, Demon SGDM) have lower performance than those of adaptive learning rate group (Adam, AMS-Grad, AdamW, QHAdam, Demon Adam): the value of the loss function is higher and the accuracy is lower. The optimizers of the adaptive learning rate group performed similarly and converged earlier than the optimizers in the other group.

The table 3.5 details the performance of optimizers. In which AMSGrad and QHAdam are the two optimization algorithms giving the best results in both models. With the first model (use LSTM), SGDM and DemonSGD give relatively low results. The reason seems to be that we have chosen inappropriate hyperparameters for optimizers (e.g. initial learning rate).

IMDb	BERT				LSTM			
	Valid acc	Valid F1	Test acc	Test F1	Valid acc	Valid F1	Test acc	Test F1
AdamW	80.600	80.516	82.010	81.949	82.075	82.074	81.410	81.410
Adam	81.175	81.083	82.090	82.022	82.000	81.970	81.470	81.439
AggMo	81.450	81.422	81.170	81.151	81.700	81.697	80.620	80.618
AMSGrad	81.575	81.493	82.040	81.979	**82.625**	**82.625**	81.950	81.950
DemonAdam	81.425	81.353	82.380	82.316	82.200	82.197	81.890	81.888
DemonSGD	**82.525**	82.521	82.510	82.508	80.050	80.047	79.460	79.460
QHAdam	80.575	**82.541**	**82.790**	**82.762**	82.325	82.303	**82.070**	**82.047**
QHM	81.475	80.459	80.480	80.475	81.250	81.248	81.420	81.481
SGDM	81.200	81.199	80.900	80.898	79.000	78.985	79.790	79.783

Table 3.5: Results of validation and test sets on IMDb

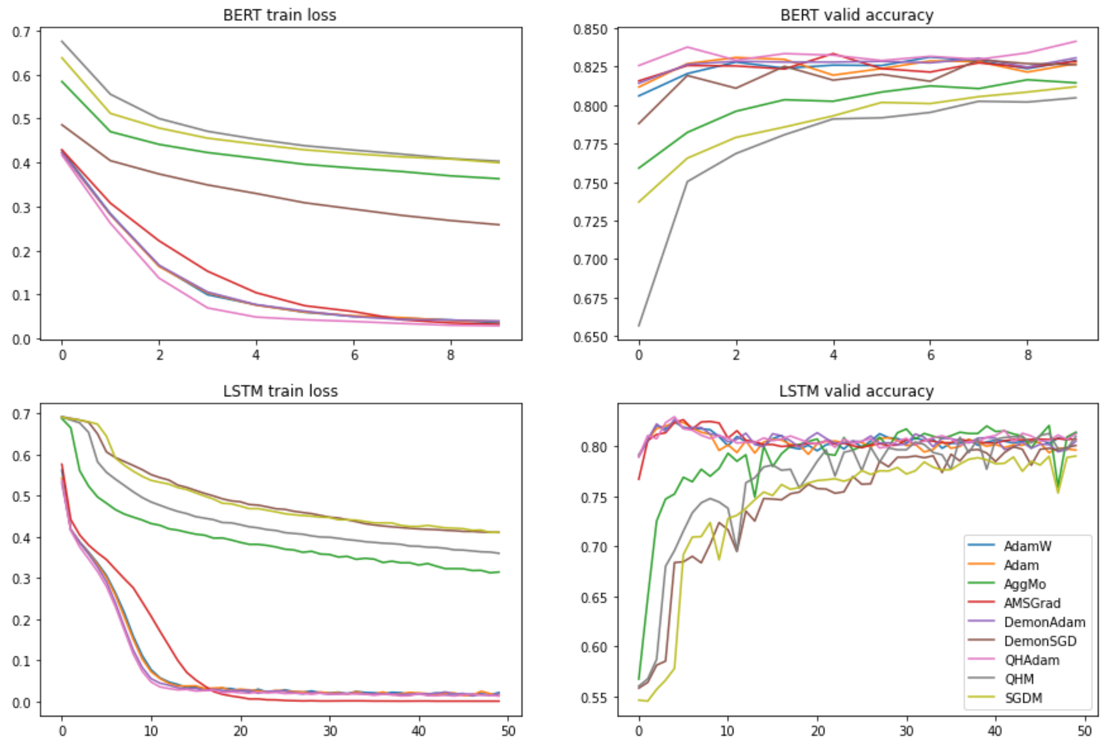


Figure 3.5: Loss and Accuracy during training process of LSTM and BERT

3.3 Image generation

3.3.1 Experimental setup

In this test, we use the MNIST dataset [6], in the image generation problem, to investigate the optimization algorithms. The dataset consists of 50000 training samples and 10000 test samples. For tuning the hyperparameter, we choose 10000 samples from the train set to make a validation set. And the best model will be selected based on the corresponding FID value. The network architecture we use is VAE with fully connected layers. We set epoch=100 and batch_size=128 for each experiment. The learning rate values corresponding to each setting are given in Table 3.6.

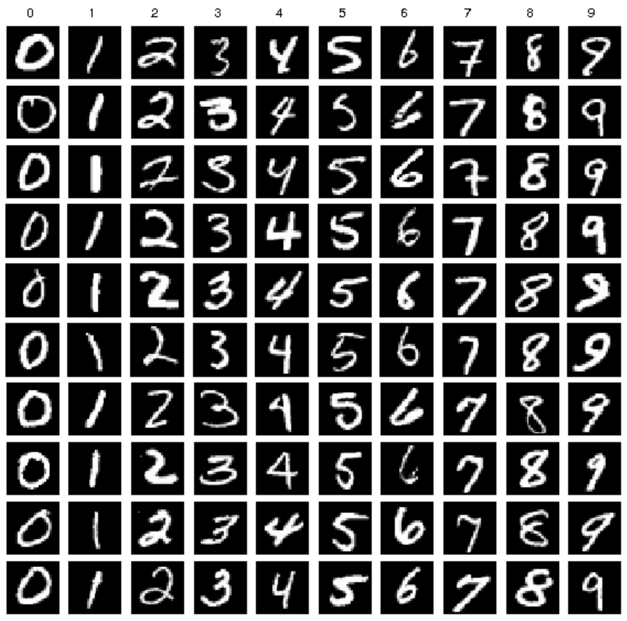


Figure 3.6: MNIST dataset

3.3.2 Results

Figure 3.7 shows the values of train loss and valid FID during training process of VAE models, corresponding to the different optimizers. When considering any pair of optimal methods, the method with the smaller train loss will be the method with the smaller valid FID. Among them, it can be clearly seen that methods based on the adaptive learning rate strategy are more effective. Moreover, when comparing DemonSGDM or DemonAdam with

Optimizer	Leaning rate used in training process
Adam	0.0001
AMSGrad	0.003
AdamW	0.001
AggMo	0.00001
QHM	0.0001
QHAdam	0.001
DemonSGD	0.00001
DemonAdam	0.001
SGDM	0.00001

Table 3.6: Learning rate values used in VAE

versions without decay momentum, Demon's significant performance improvement can be seen.

Table 3.7 is the details of FID, IS scores, and training time (per epoch) of VAE models when applying the optimization methods. We can see that the execution time of most of the methods does not differ too much, but AMSGrad seem to take more time than other methods. Methods for the lowest (best) FID include: QHAdam, AdamW, and DemonAdam. The methods for the highest (best) IS include: AdamW, DemonAdam and AMSGrad.

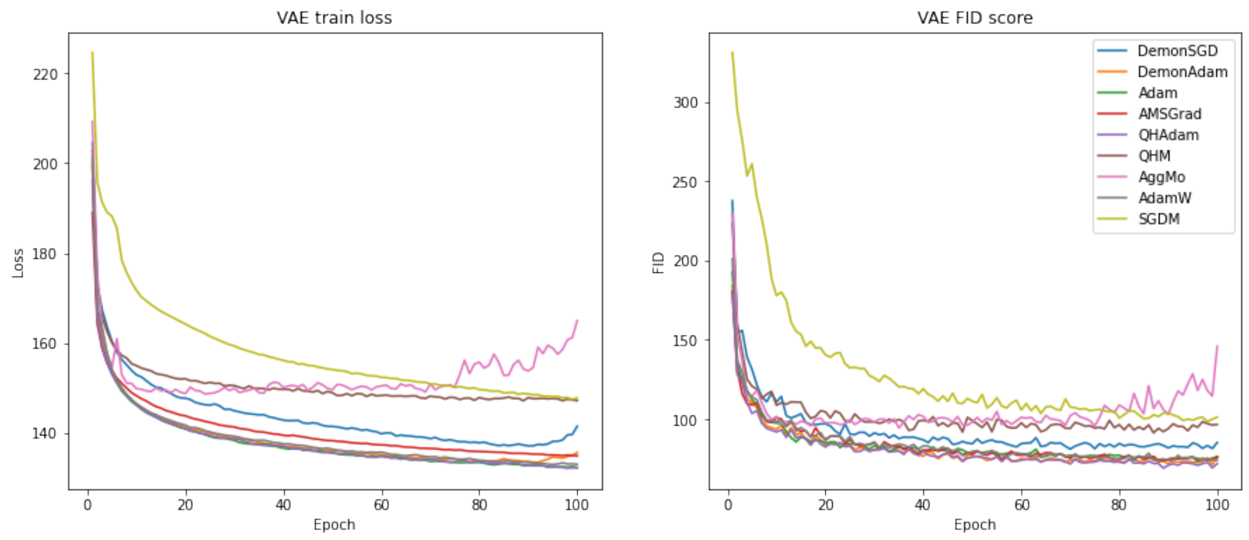


Figure 3.7: Train loss and Valid FID of VAE

Optimizer	FID	IS	time (minute)
SGDM	93.535	2.090	4.403
DemonSGDM	82.751	2.197	4.447
DemonAdam	74.398	2.256	4.646
Adam	74.447	2.151	4.565
AMSGrad	74.511	2.220	5.696
QHAdam	71.718	2.208	4.815
QHM	95.078	2.043	4.502
AggMo	99.923	2.077	4.543
AdamW	74.028	2.262	4.548

Table 3.7: FID, IS and time of VAEs

Chapter 4

Conclusion

In this project, we conducted a survey of gradient descent optimization algorithms. They are all techniques that have become popular in recent times. The experimental results show that: Methods belonging to the adaptive learning rate group give better performances. The Adam-based methods are often the leading methods of learning effectiveness (QHAdam, AdamW, Demon Adam). In addition, Demon's decay momentum strategy also showed a clear effect when significantly improving the original versions of SGDM and Adam.

Bibliography

[1] J. Chen and A. Kyrillidis. "Decaying momentum helps neural network training". *CoRR* abs/1910.04952 (2019). URL: http://arxiv.org/abs/1910.04952.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019,* J. Burstein, C. Doran, and T. Solorio (Eds.). 2019, pp. 4171–4186.

[3] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". *2016 IEEE Conference on Computer Vision and Pattern Recognition,CVPR 2016,* IEEE Computer Society, 2016, pp. 770–778.

[4] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". *3rd International Conference on Learning Representations, ICLR 2015*. Y. Bengio and Y. LeCun (Eds.). 2015.

[5] A. Krizhevsky, G. Hinton, et al. "Learning multiple layers of features from tiny images" (2009).

[6] Y. LeCun and C. Cortes. "MNIST handwritten digit database" (2010). URL: http://yann.lecun.com/exdb/mnist/.

[7] D. Lin, Y. Matsumoto, and R. Mihalcea. "Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies". *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. 2011.

[8] I. Loshchilov and F. Hutter. "Decoupled Weight Decay Regularization". *7th International Conference on Learning Representations, ICLR 2019,* 2019.

[9] J. Lucas, S. Sun, R. S. Zemel, and R. B. Grosse. “Aggregated Momentum: Stability Through Passive Damping”. *7th International Conference on Learning Representations, ICLR 2019*. 2019.

[10] J. Ma and D. Yarats. “Quasi-hyperbolic momentum and Adam for deep learning”. *7th International Conference on Learning Representations, ICLR 2019*. 2019.

[11] J. Pennington, R. Socher, and C. D. Manning. “Glove: Global vectors for word representation”. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[12] N. Qian. “On the momentum term in gradient descent learning algorithms”. *Neural Networks* 12.1 (1999), pp. 145–151. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6. URL: https://www.sciencedirect.com/science/article/pii/S0893608098001166.

[13] S. J. Reddi, S. Kale, and S. Kumar. “On the Convergence of Adam and Beyond”. *6th International Conference on Learning Representations, ICLR 2018*. 2018.