Optimization Algorithms for Machine Learning

Ong Yi Sheng Geoff

November 15, 2024

1 Introduction

Optimization algorithms such as Gradient Descent (GD) are fundamental to the field of machine learning, serving as the backbone for numerous pivotal algorithms, including gradient boosting, neural networks, and transformers. As neural networks gained prominence in the industry, the limitations of these methods became increasingly apparent.

GD encounters two primary challenges in navigating complex, high-dimensional loss landscapes: local minima and saddle points. Local minima are points where the loss function reaches a minimum in a specific region but is not the global minimum, which can cause GD to settle at suboptimal solutions. Saddle points, on the other hand, are critical points with mixed curvature; some directions slope downward, while others slope upward, creating instability. These points are prevalent in Neural network loss functions due to back-propagation.

Note: As dimensionality increases, the proportion of saddle points relative to local minima grows exponentially. Intuitively, the likelihood that all directions around a critical point lead upward (positive curvature) decreases exponentially (Dauphin et al., 2014). In deep learning, where the dimensionality is vast and the problem is highly non-convex, saddle points pose a more substantial challenge for optimization.

To address these issues, adaptive GD methods have been developed. These methods adjust the learning rate for each parameter based on past gradient information, aiming to improve convergence and handle sparse or noisy gradients more effectively. This paper will examine three of the most popular adaptive algorithms: AdaGrad, RMSProp, and Adam.

For each algorithm, we look at the mini-batch version, as it is the most general version: If the batch size is 1, it becomes the stochastic version, whereas if batch size is the total number of data points, it becomes the batch version. We also introduce a small term $\epsilon > 0$ in the denominator where necessary, to prevent dividing by zero.

2 AdaGrad

AdaGrad algorithm is given below in Algorithm 1:

Algorithm 1 AdaGrad for mini-batch setting

1: **Parameters:** Initialize the weights $\theta_0 \in \mathbb{R}^d$, batch size $B \in \mathbb{N}, B \leq n$, where n is the total number of data points, and parameters $\epsilon > 0, b_0 = 0, \alpha \in \mathbb{R}^+$
2: **for** $t = 0, 1, 2, ..., T$ **do**
3: **Mini-batch average gradient:** $g_t = \frac{1}{B} \sum_{i=1}^{B} \nabla f_i(\theta_t)$
4: **Moving update:** $b_t = b_{t-1} + g_t^2$
5: **Update:** $\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{b_t + \epsilon}} g_t$
6: **end for**

2.1 Motivation

AdaGrad aims to solve the issue of sparse features. In fields like natural language processing, different features can appear in varying frequencies. For standard GD methods, the learning rate scales all the feature gradients equally, which may cause the rare features to not learn effectively. AdaGrad addresses this problem by adapting the learning rate for each parameter. Parameters of higher frequency features

will have their learning rates reduced while that of lower frequency features will largely retain their larger learning rate. This property also helps it to escape saddle points faster, as compared to standard GD methods.

2.2 Remarks

We square the gradient to ensure the negative gradients does not cancel out the positive ones.
As the algorithm accumulates the squared gradients from the beginning of training, the effective learning rate will become very small. This causes slower convergence and in the worst case, even causing the model to stop learning.

3 RMSProp

RMSProp algorithm is given below in Algorithm 2:

Algorithm 2 RMSProp for mini-batch setting

1: **Parameters:** Initialize the weights $\theta_0 \in \mathbb{R}^d$, batch size $B \in \mathbb{N}, B \leq n$, where n is the total number of data points, and parameters $\gamma \in (0,1), \epsilon > 0, b_0 = 0, \alpha \in \mathbb{R}^+$
2: **for** $t = 0, 1, 2, ..., T$ **do**
3: **Mini-batch average gradient:** $g_t = \frac{1}{B} \sum_{i=1}^{B} \nabla f_i(\theta_t)$
4: **Moving average update:** $b_t = \gamma b_{t-1} + (1-\gamma) g_t^2$
5: **Update:** $\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{b_t + \epsilon}} g_t$
6: **end for**

A few things to take note:

- We take the Exponential Moving Average (EMA) of the squared gradients instead of just summing (as opposed to AdaGrad)
- We take the square root of the EMA of the squared gradient, hence the name "Root Mean Squared Propagation" (RMSProp).
- γ controls the balance between the contribution of past squared gradients and the current squared gradient, allowing adjustments to the emphasis on historical gradients relative to the most recent one.

3.1 Motivation

RMSProp extends AdaGrad by addressing its slower convergence, which arises from the accumulation of squared gradients that can become excessively large over time. A property of EMA of squared gradients is that each parameters is bounded above by the supremum parameter of squared gradient. That is, $\forall t, \forall i \leq d$ where d is the number of features, $b_{ti} \leq \sup_t g_{ti}^2$. This means that the adjustments to the learning rate are less drastic compared to AdaGrad, which is the sum of squared gradients, effectively mitigating the issues of slow convergence or premature halting of learning. Furthermore, EMA reduces the impact of large or noisy gradient updates. Instead of reacting strongly to individual, potentially erratic gradient values, averaging process ensures that occasional large updates, which could destabilize the learning process, are tempered, preventing sudden jumps in parameter values.

3.2 Remarks

RMSProp converges much faster than AdaGrad, while still decreasing the learning rate when the gradient is getting smaller. This allows RMSProp to converge relatively quickly and stably.

4 Adam

Adam algorithm is given below in Algorithm 3:

Algorithm 3 Adam for mini-batch setting

1: **Parameters:** Initialize the weights $\theta_0 \in \mathbb{R}^d$, batch size $B \in \mathbb{N}, B \leq n$, where n is the total number of data points, and parameters $\gamma_1, \gamma_2 \in (0,1), \epsilon > 0, \alpha \in \mathbb{R}^+, b_0 = 0, m_0 = 0$
2: **for** $t = 0, 1, 2, ..., T$ **do**
3: **Mini-batch average gradient:** $g_t = \frac{1}{B}\sum_{i=1}^{B} \nabla f_i(\theta_t)$
4: **Moving average gradient:** $m_t = \gamma_1 m_{t-1} + (1-\gamma_1) g_t$
5: **Moving average update:** $b_t = \gamma_2 b_{t-1} + (1-\gamma_2) g_t^2$
6: **Bias correction:** $\hat{m}_t = \frac{m_t}{1-\gamma_1^t}$, $\hat{b}_t = \frac{b_t}{1-\gamma_2^t}$
7: **Update:** $\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{b}_t + \epsilon}} \hat{m}_t$
8: **end for**

A few things to take note:

- If $\gamma_1 = 0$, we will get back RMSProp.
- We perform bias correction as Adam is biased towards zero especially the beginning. This is due to the nature of exponential-moving-average and the initialization of $b_0 = 0, m_0 = 0$.

4.1 Motivation

Adam uses a combination of RMSProp and momentum, where it uses EMA for gradient and squared gradient. This allows Adam to solve both issues at hand: escaping saddle points and a chance to escape local minimas.

4.2 Remarks

Adam is one of the most popular machine learning optimization algorithms due to the ability to tune different hyperparameters. The combination of momentum and RMSProp makes Adam robust, and is generally a good choice for neural-network optimization.
However, Adam (without modification) has not been proven to converge for all convex functions. Many different variants of Adam such as AdamW, AMSGrad etc is created to circumvent this problem and to improve convergence in specific scenarios.

5 Experiments

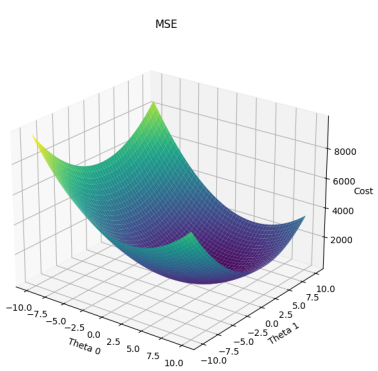
We design toy experiments to compare the performance of various optimization algorithms, including vanilla gradient descent (GD), gradient descent with momentum (GDM), RMSProp, AdaGrad, and Adam. Our objective is to illustrate the strengths and weaknesses of each method through practical examples, highlighting the specific scenarios where certain algorithms excel and where their limitations become more pronounced.

5.1 Convex loss function: Mean-Squared-Error

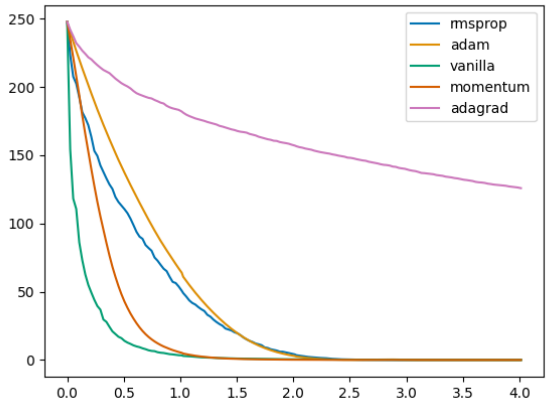
Our first experiment will be a straightforward one: minimize the standard Mean-Squared-Error (MSE) loss function for regression. That is,

$$MSE = \frac{1}{m}\|y - X\theta\|^2 \tag{1}$$

where m is the number of data points we use per batch.
For this experiment, we use sklearn's *make_regression* function to create a dataset of 8 features and 600 data points. We will use a batch-size of 16 and 4 epoch.



(a) MSE 3d curve



(b) MSE loss graph

The graph illustrates that GD and GDM converge more rapidly than the adaptive GD methods. This faster convergence can be attributed to the convex nature of the function, which features a straightforward downward slope. The learning rate used in GD and GDM only take into account the current iteration number, and since the function goes to 0 fast due to convexity of the function, the learning rate did not decrease much throughout. This enables them to make more significant progress toward the minimum in each iteration. In contrast, the adaptive GD methods employ a decreasing learning rate based on previous gradients, which slows their convergence over time. Note that AdaGrad have not converged after 4 epochs. This is due to the accumulation of squared gradients, which greatly reduce convergence rate. Hence, usage of AdaGrad is generally discouraged, unless a very stable descent is required.

5.2 Function with multiple local minima

For our next experiment, we will analyze how well each method are able to escape local minima. To do so, We aim to minimize the function:

$$f(x) = 0.015x^4 - 0.05x^3 - 0.35x^2 + 0.05x \tag{2}$$

An illustration of the curve can be seen below:

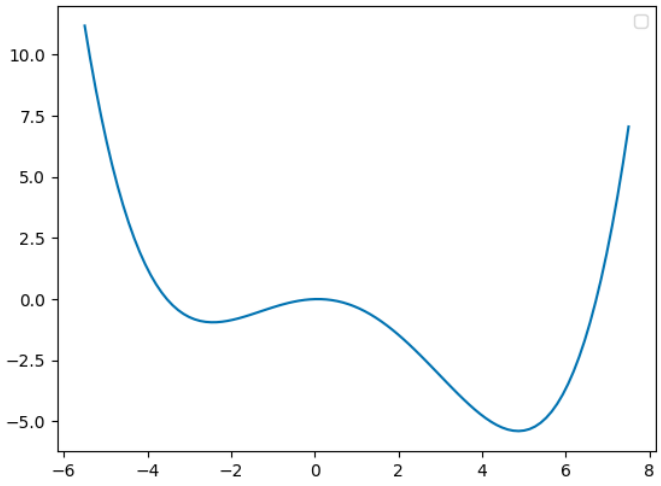


Figure 2: function with a local and global minimum

Note that there is a global minimum around $(5, -5)$ and a local minimum around $(2.5, -1)$.
We first perform gradient descent methods on this function. We start from $x_0 = -7$ for each methods

to observe whether the GD methods can move past local minima and successfully escape them in pursuit of the global minimum. We shall set momentum constant for GDM and Adam to be 0.99, to see its effects clearly.

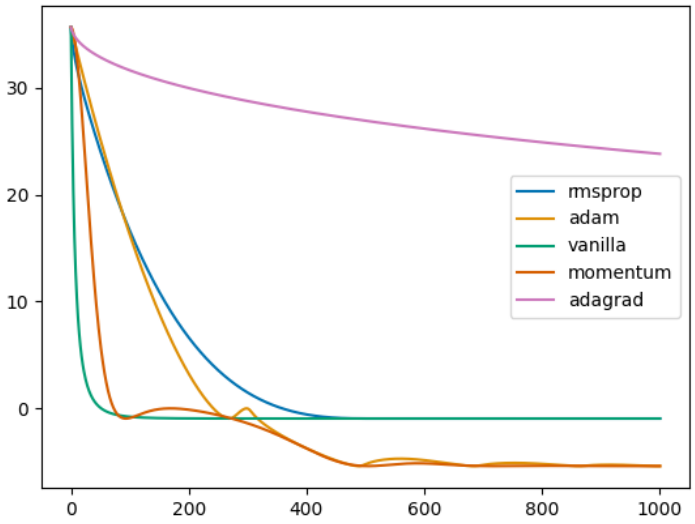


Figure 3: non-stochastic local-minima loss graph

We find that Adam and GDM is able to escape the local minima. This is because of the momentum buildup from "rolling" down the hill, which gives the EMA for gradient enough momentum to overcome the local minima. Do note that the rest of the gradient descent methods are stuck at the local minima.

We aim to examine the impact of stochasticity on escaping local minima. Although this function is not a loss function, we can simulate stochastic gradient descent (SGD) methods by introducing random noise into the computed gradients. While this approach is not a perfect emulation, it demonstrates the potential of SGD. For this experiment, we will be using gaussian noise to simulate SGD methods. **Note:** In this context, we refer to GD methods with noise as an example of SGD, even though these methods are sometimes specifically referred to as 'perturbed GD'.

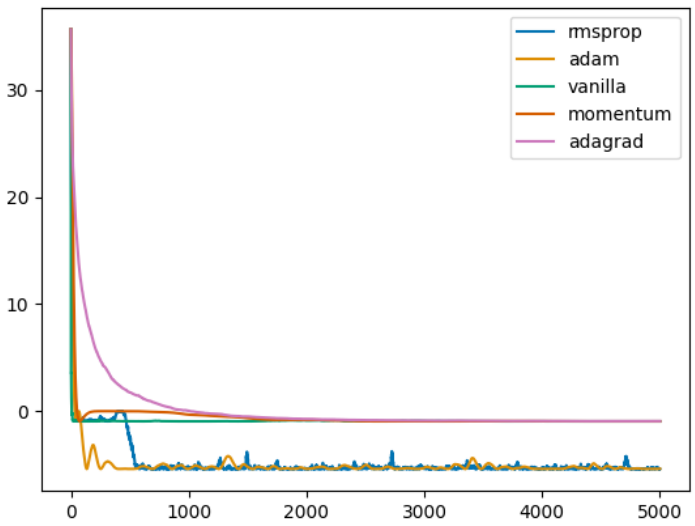


Figure 4: stochastic local-minima loss graph

We observe that, in this instance, RMSProp and Adam are able to escape local minima. This phe-

nomenon occurs because the injected noise provides a sufficient boost to push the SGD method (more specifically, RMSProp) out of local minima. However, it's worth noting that this outcome was observed only after repeating the experiment multiple times with noise of high variance $\sigma^2 = 9$, suggesting it is not a consistent effect. Interestingly, in this particular trial, momentum does not seem effective in overcoming the local minima. This may be due to the high variance of the noise we introduced. This in turn could disrupt the optimization trajectory, potentially masking the advantages of methods like momentum that rely on smooth directional updates.
Note: In this example, the local minima is not very deep, hence momentum is able to overcome it and converge to the global minima.

5.3 Saddle point function

Next, we want to see how well each GD methods escape the saddle point. For that, we aim to minimize the following function:

$$g(x_1, x_2) = x_1^2 - x_2^2 + 0.05x_2^4 \tag{3}$$

An illustration of the curve can be seen below:

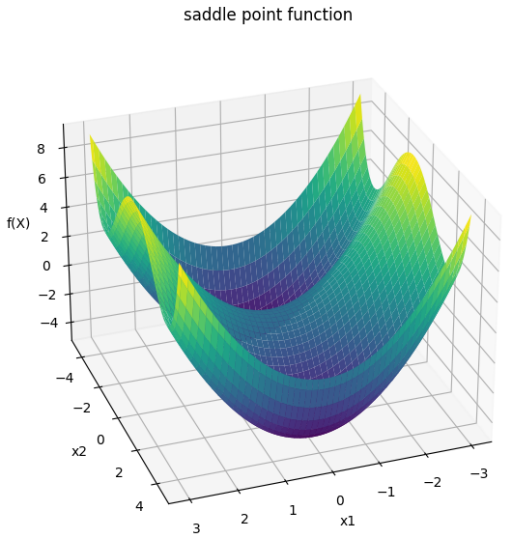


Figure 5: Saddle point function curve

Note that there is a simple (or non-degenerate) saddle point at $(0, 0)$, with slopes descending towards two equally valued local minima. If the gradient descent methods can overcome the saddle point, they will be able to converge to these local minima.
Like the previous experiment, we first perform GD methods on this function. We start from for each methods to observe if they are able to escape the saddle point, and how fast they are able to do so. To do this, we choose starting point $X = (5, 0)$.

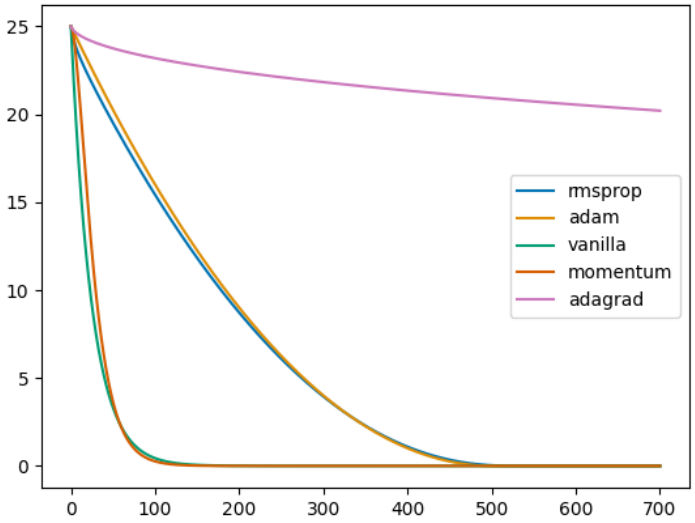


Figure 6: non-stochastic saddle point loss graph

It seems that all the GD methods are stuck at the saddle point, which may seem surprising as adaptive GD methods are supposed to solve this issue. However, when inspecting, we realized that since we start from x_2 is directly above the saddle point ($x_2 = 0$), the gradient in the x_2 direction is 0 throughout, hence the descent methods are not able to move in the x_2 direction at all, which means they are unable to escape the saddle point.

Like the previous experiment, this time, we add stochasticity into the GD methods. We use a noise of $\sigma^2 = 0.25$.

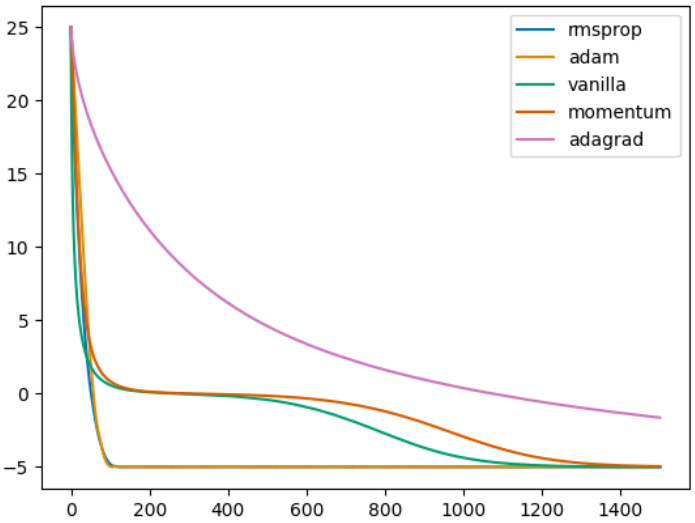


Figure 7: stochastic saddle point loss graph

Due to the noise added, the SGD methods is able to move in the x_2 direction, hence allowing them to escape the saddle point. Notably, RMSProp, Adam and AdaGrad escape saddle points quickly, while the other SGD methods get temporarily stuck in the saddle point before eventually breaking free. This difference arises because RMSProp, Adam and AdaGrad adaptively adjust the learning rate for each feature, enabling these methods to respond dynamically to the geometry of the loss surface. Their per-feature learning rate adjustments allow them to accelerate through saddle points. Do note that SGD methods are all able to escape the saddle point. Indeed, given a number of iterations and

non-zero noise, SGD methods can escape the simple saddle points and converge into a local minima.
Note: The simple saddle point is also called a non-degenerate saddle point. There is another type of saddle point that is called degenerate saddle point, where the Hessian matrix does not have full rank or is singular. This makes convergence much more complex, as the saddle point does not clearly indicate increasing or decreasing values in some direction.

Finally, we use artificial neural networks (NN) to model a synthetic dataset generated using sklearn's *make_regression* function with 25 features and 1024 data points. We design the NN with the following architecture: 1 input layer with 25 units, 5 hidden layers with sizes 55, 20, 104, 10 and 150 units respectively, and 1 output layer with a single unit. This makes the NN loss function non-convex due to the large number of hidden layers and the changing hidden layer size. All layers use the sigmoid activation function. For each experiment, we use a batch size of 16 and train the model for 25 epochs to assess convergence speed. We test the performance of different optimizers, observing how each affects the reduction in training loss.

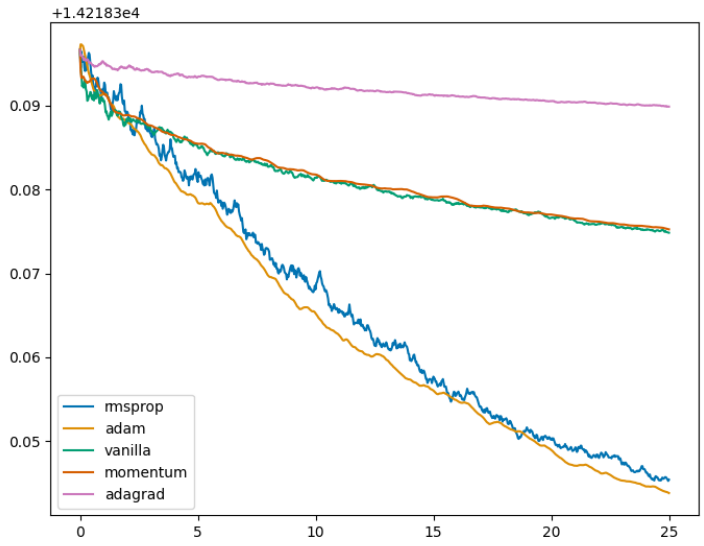


Figure 8: NN loss graph

In the graph, we observe an initial spike in training loss across all optimizers, likely due to a poor starting point near a saddle point. This behavior is particularly notable for AdaGrad, which appears to stagnate early on, while the other optimizers successfully escape this region. The stagnation in AdaGrad could be due to its cumulative squared gradient becoming large, effectively reducing its step size and causing slower convergence (I also set the initial learning rate to be the same across all optimizers to be fair, hence AdaGrad is disadvantaged here). In contrast, Adam and RMSProp avoid this issue by adapting their learning rates dynamically, which allows them to minimize the loss more quickly. The smoother convergence paths of Adam and momentum-based optimizers are noteworthy. This stability can be attributed to their use of momentum, which accelerates the optimization in a consistent direction, dampens oscillations, and promotes more stable progress toward the minimum. In comparison, momentum-free optimizers like vanilla SGD exhibit more oscillatory behavior, as they lack the directional consistency provided by momentum. Overall, Adam and RMSProp demonstrate both faster convergence and smoother trajectories, thanks to their adaptive learning rates.
Note: This graph was generated after extensive trial and error with the learning rate, which is why both the vanilla optimizer and momentum optimizer did not experience divergence. This highlights the importance of careful hyperparameter tuning when working with standard optimizers to ensure stable and effective training.
Another challenge encountered is the initialization of weight layers. To ensure a fair comparison, it is crucial that all networks share the same initialization. Therefore, we use specific predicates to initialize the weights for all NNs, while maintaining consistency in the initial weights and biases across all models.

6 Conclusion

After examining the performance of various GD methods, we observed distinct advantages across different types. Through the local minima function example, we found out that using momentum has proven effective for escaping local minima, as the accumulated gradient helps move past potential traps. For escaping simple saddle points, adaptive SGD methods such as RMSProp, AdaGrad and Adam show strong performance, though standard SGD methods can also handle saddle points due to its inherent randomness. For the NN example, we see each optimizers in action, and how momentum smooths out the noise.

Adam stands out as one of the most popular optimizers, known for its robustness and versatility in handling diverse issues across tasks, as it is able to solve both issues handily. However, in practice, it has been shown that SGD with momentum can outperform Adam in certain cases and with careful hyperparameter tuning. Its stochasticity not only enables it to escape saddle points but also allows it to converge more rapidly.

References

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Advances in neural information processing systems*, *27*.