

---

# Deep Learning Optimization: Impact of Optimizers

---

Supervised by

Dr. Ahmed Abdelsamea

Prepared by

Seif Eldin 202200973 , Ziad Shaaban 202201093 ,

Ahmed Mostafa 202201114

**Draft 2**

**12/22/2024**

# Table of contents

Deep Learning Optimization: Impact of Optimizers	0
Table of contents	1
Abstract	2
Introduction	2
1. Stochastic Gradient Descent (SGD)	2
2. SGD with Momentum	3
3. Adaptive Moment Estimation (ADAM)	3
Problem definition	3
Methodology	5
Stochastic Gradient Descent (SGD)	5
SGD with Momentum	6
Adaptive Moment (ADAM)	7
Adaptive Gradient (AdaGrad)	7
Root Mean Square propagation (RMSprop)	7
Merging The Techniques	8
Results and Comparisons	10
Key Observations	12
Conclusion	12
References	14

# Abstract

This report shows how different optimizers affect deep learning models, as deep learning relies on optimization algorithms in the training process by minimizing the loss function, resulting in better performance, accuracy, and speed. And what different categories of optimizers. The report categorizes optimizers into three main types: Stochastic Gradient Descent (SGD), Momentum-based Optimizers, Adaptive Optimizers

For this study, we will focus on one representative optimizer from each category:

- **SGD (Stochastic Gradient Descent)** as a standard baseline optimizer.
- **SGD with Momentum** represents the momentum-based category.
- **Adam (Adaptive Moment Estimation)** is a widely used adaptive optimizer.

Showing the difference in performance, accuracy, and computational efficiency. Highlighting their respective advantages and trade-offs.[4] [6]

## Introduction

Optimizers are methods or “algorithms” that help in deep learning by minimizing the loss function, which quantifies the difference between predicted and actual outputs of the deep learning model, resulting in the adjustments of some of the model parameters (weights and biases) iteratively to achieve this goal (better accuracy, performance, and computational efficiency). By this, optimizers play a vital role in navigating the complex parameters of the model. Enabling efficient convergence, managing issues like vanishing or exploding gradients, and improving model performance. By this, choosing an optimizer can dramatically affect the model results.[5] [6][4]

2024; Optimizers

### 1. Stochastic Gradient Descent (SGD)

It is a type of Gradient descent but does not compute gradients using the entire data set. SGD injects an element of randomness and updates the data using small training examples at each iteration, making it much faster than the standard and making it more efficient in larger data sets. Additionally, the randomness in SGD can help the algorithm escape from local minima or saddle points, potentially allowing it to find a better global optimum, especially in complex, non-convex optimization problems. However, while randomness can lead to faster convergence, SGD can

exhibit high variance in its updates, introducing variability and causing oscillations around the optimal solution. [5][6][9][10]

## 2. SGD with Momentum

Momentum-based SGD extends the standard SGD method by incorporating momentum to modify the update rule in the normal gradient. This update includes a weighted moving average of past gradients. Additionally, it introduced normalized updates that ensure that the updates are determined by the direction of the gradient rather than the magnitude, which made a great improvement in non-convex optimization where magnitude can vary widely.

This reduces the variance of updates, provides better stability, accelerates the convergence along consistent descent directions, helps smooth out oscillations, and improves the model's performance, especially in a noisy or high-variance environment. All this without requiring large batch sizes, making it well-suited for modern machine-learning tasks.[3][8][2][9]

## 3. Adaptive Moment Estimation (ADAM)

ADAM is a widely used optimizer that uses two popular gradient descent algorithm extensions, which are **momentum** and **RMSProp**. Taking the momentum method's smoothness and the adaptive learning rate of **RMSProp**, it provides an efficient and adaptive approach to gradient-based optimization.

**ADAM** maintains exponentially decaying averages of past gradients and the squared gradient during training, ensuring bias correction for their initialization, which allows it to achieve fast convergence while being a powerful tool for noisy and sparse gradient. Additionally, it needs minimal memory and makes excellent empirical performance across diverse settings, making it a perfect choice for training complex neural networks and a cornerstone in modern deep learning frameworks such as TensorFlow and PyTorch.[9][1][11]

## Problem definition

Optimization algorithms play a big role in image classification, impacting accuracy, convergence speed, stability, and final model performance. In this study, we aim to compare the performance and tradeoffs for all SGD, SGD with momentum, and Adam on an image classification problem. Focusing on evaluating their effects on training dynamics, convergence rates, and classification accuracy, providing insights into their strengths and limitations in solving image classification tasks effectively.

# Methodology

## Stochastic Gradient Descent (SGD)

The methodology implementation for SGD involves several steps to ensure that the objective function is optimized, probably

First, we define the objective function as  $J(\theta; x_i, y_i)$ . Which represents the loss function that we want to minimize. This function depends on the parameter  $\theta$ , the training examples  $x^{(i)}$ , and the labels  $y^{(i)}$ .

The Stochastic Gradient Descent(SGD) is defined by the following equation:

$$\theta = \theta - \eta \cdot \nabla J(\theta; x_i, y_i) \quad (1)$$

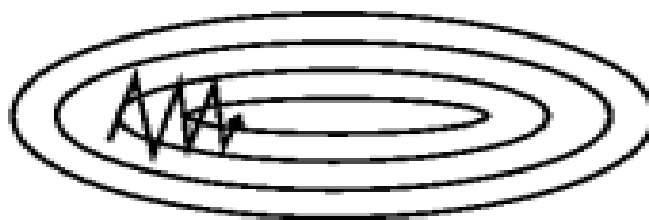
Where:

$\eta$  ( $\eta$ ): learning rate

$\eta \cdot \nabla J$ : loss functions's gradient

The above equation is used to update the hyperparameter ( $\theta$ ), which depending on the function could update weights, biases, boundaries, etc.

Secondly, we shuffle the data before training and choose a random small batch of the data to train on. While training, we decrease the learning rate gradually over time which mitigates the high variance caused by frequent updates, helps in stabilizing the convergence, and prevents cyclic patterns. [11]



(a) SGD without momentum

Figure 1: hesitant progress towards local optimum. Adapted from [11]

```
for i in range ( nb_epochs ):
    np . random . shuffle ( data )
    for example in data :
        params_grad = evaluate_gradient ( loss_function , example , params )
        params = params - learning_rate * params_grad
```

## SGD with Momentum

To resolve the hesitation issue, we'll implement a momentum parameter to address the oscillation problem and increase convergence speed. To incorporate this, we define a velocity vector to soften the update process in the relevant direction, achieving a faster, smoother descent. Similarly to SGD, an objective function is defined, training data is shuffled, and the learning rate is gradually decreased for stability.

$$vt = \gamma v_{t-1} + \eta \cdot \nabla \theta J(\theta) \quad (2)$$

*Velocity update equation incorporating update history*

$$\theta = \theta - vt \quad (3)$$

*Parameter update*

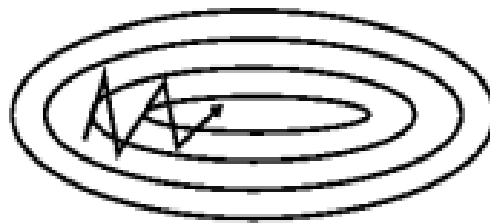
where:

$\gamma$ : momentum coefficient ( $\sim 0.9$ )

$v$ : Direction vector

$t$ : time step

$\eta$ : learning rate



(b) SGD with momentum

**Figure 2:** the effect of momentum on progress towards local optimum. Adapted from [11]

```
def momentum_optimizer(gradient_fn, theta, learning_rate=0.01,
momentum=0.9, num_iterations=1000):
    v = np.zeros_like(theta)
    history = [theta.copy()]
    for _ in range(num_iterations):
        gradient = gradient_fn(theta)
        v = momentum * v + learning_rate * gradient
        theta -= v
        history.append(theta.copy())
    return theta, history
```

## Adaptive Moment (ADAM)

Adam uses multiple techniques to optimize, it merges between Root Mean Square propagation (RMSprop) and momentum-based algorithms. A momentum-based approach was discussed in the previous section; here, a brief overview of RMSprop will be discussed as well as AdaGrad which RMSprop is based on. After that, a brief overview on how ADAM merges these techniques.

## Adaptive Gradient (AdaGrad)

To address the update frequency issues of individual parameters usually faced in traditional SGD, AdaGrad assigns an adaptive learning rate for parameters  $\theta_i$  based on their update frequency, this provides an increase in update frequency for parameters that weren't frequently updated in prior implementations. Thus, AdaGrad incorporates historical feedback into the learning rate for each parameter given its frequency eliminating the need for the previously required manual tuning.[11]

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \quad (4)$$

where:

$\epsilon$ : Constant to address division by zero

$\eta$ : learning rate

$t$ : time step

$g$ : loss function gradient  $\rightarrow g_t = \nabla \theta J(\theta)$

$G_{t,ii}$ : Summation of the SG for  $\theta_i$  over  $t$  steps

## Root Mean Square propagation (RMSprop)

RMSprop was originally implemented to address the diminishing learning rate issues with the AdaGrad optimizer. It resolves this issue by calculating the root mean square of an exponentially weighted gradient, meaning that it continuously adapts the learning rate using the gradient weight. Thus, RMSprop is known as an adaptive learning rate optimizer.[11]

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \quad (5)$$

Updates the exponentially weighted moving average

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \quad (6)$$

Updates the hyperparameter  $\theta$

where:

$\epsilon$ : Constant to address division by zero

$\eta$ : learning rate

$t$ : time step

$g$ : loss function gradient  $\rightarrow g_t = \nabla \theta J(\theta)$

$E$ : exponentially weighted average

## Merging The Techniques

To create a more robust optimizer, ADAM combines the strengths of both Adaptive gradient, RMS propagation, and momentum-based algorithms by using the exponentially weighted average ( $m_t$ ) to achieve a smoother update towards the descent direction. [11]

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad (7)$$

Then by calculating the exponentially decaying average of the squared gradient ( $v_t$ ), ADAM ensures a controllable parameter value using the magnitude gradients when scaling the parameter updates.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad (8)$$

The above variables  $m_t$  &  $v_t$  are initialized to be zeros which are then updated to correct biases as follows:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (9, 10)$$

Finally, Adam combines both moments estimates for adaptively updating the parameters as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \cdot \hat{m}_t \quad (11)$$

where:

$\epsilon$ : Constant to address division by zero

$\eta$ : learning rate

$t$ : time step

$g$ : loss function gradient  $\rightarrow g_t = \nabla \theta J(\theta)$

$m_t$ : exponentially weighted average

$v_t$ : exponentially decaying average

$\beta_1, \beta_2$ : hyperparameters controlling decay rate



```

def adam_optimizer(gradient_fn, theta, learning_rate=0.001, beta1=0.9,
                   beta2=0.999, epsilon=1e-8, num_iterations=1000):

    m = np.zeros_like(theta)
    v = np.zeros_like(theta)
    t = 0
    history = [theta.copy()]

    for _ in range(num_iterations):
        t += 1
        gradient = gradient_fn(theta)
        m = beta1 * m + (1 - beta1) * gradient
        v = beta2 * v + (1 - beta2) * (gradient ** 2)
        m_hat = m / (1 - beta1 ** t)
        v_hat = v / (1 - beta2 ** t)
        theta -= learning_rate * m_hat / (np.sqrt(v_hat) + epsilon)
        history.append(theta.copy())

    return theta, history

```

# Results and Comparisons

We evaluated our implementation and trained them on **MNIST dataset**. [link](#) using a neural network, and then we compared them to the built-in optimizers in the TensorFlow library. Notebook [link](#)

## Neural network structure

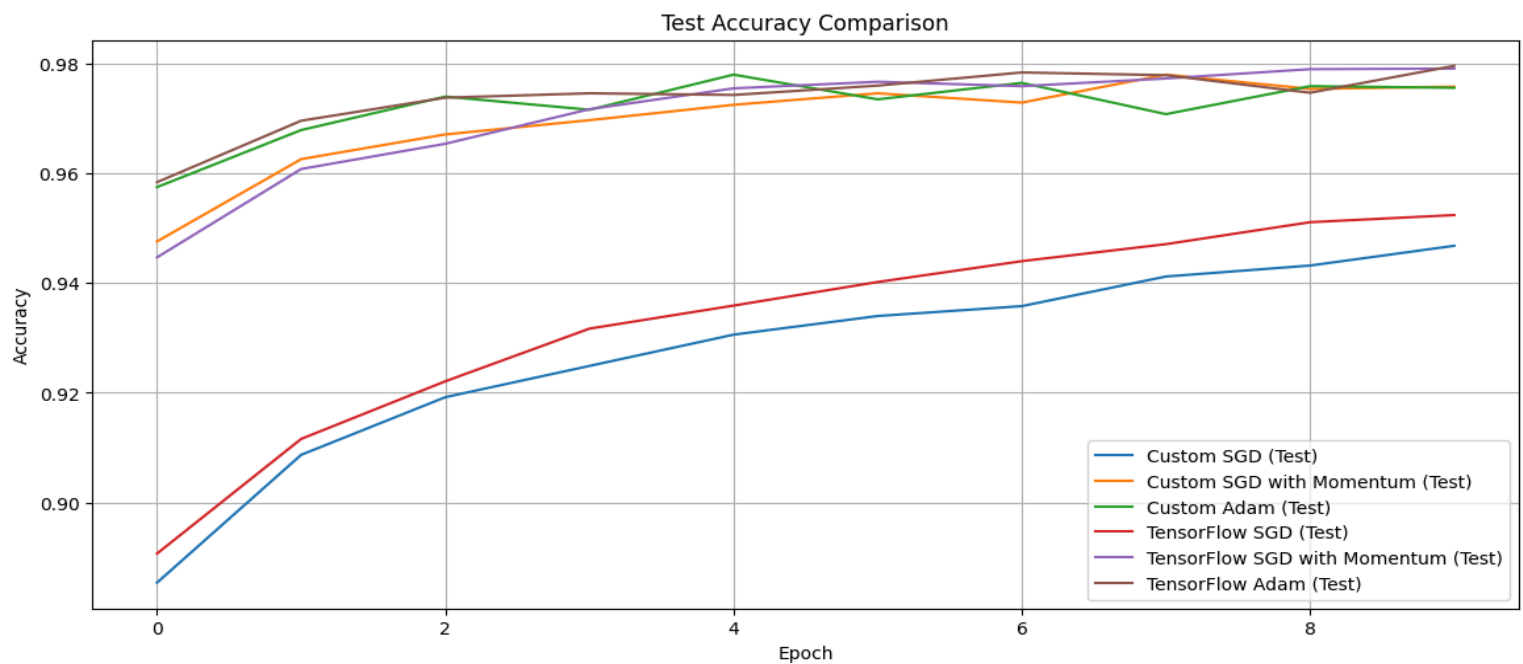
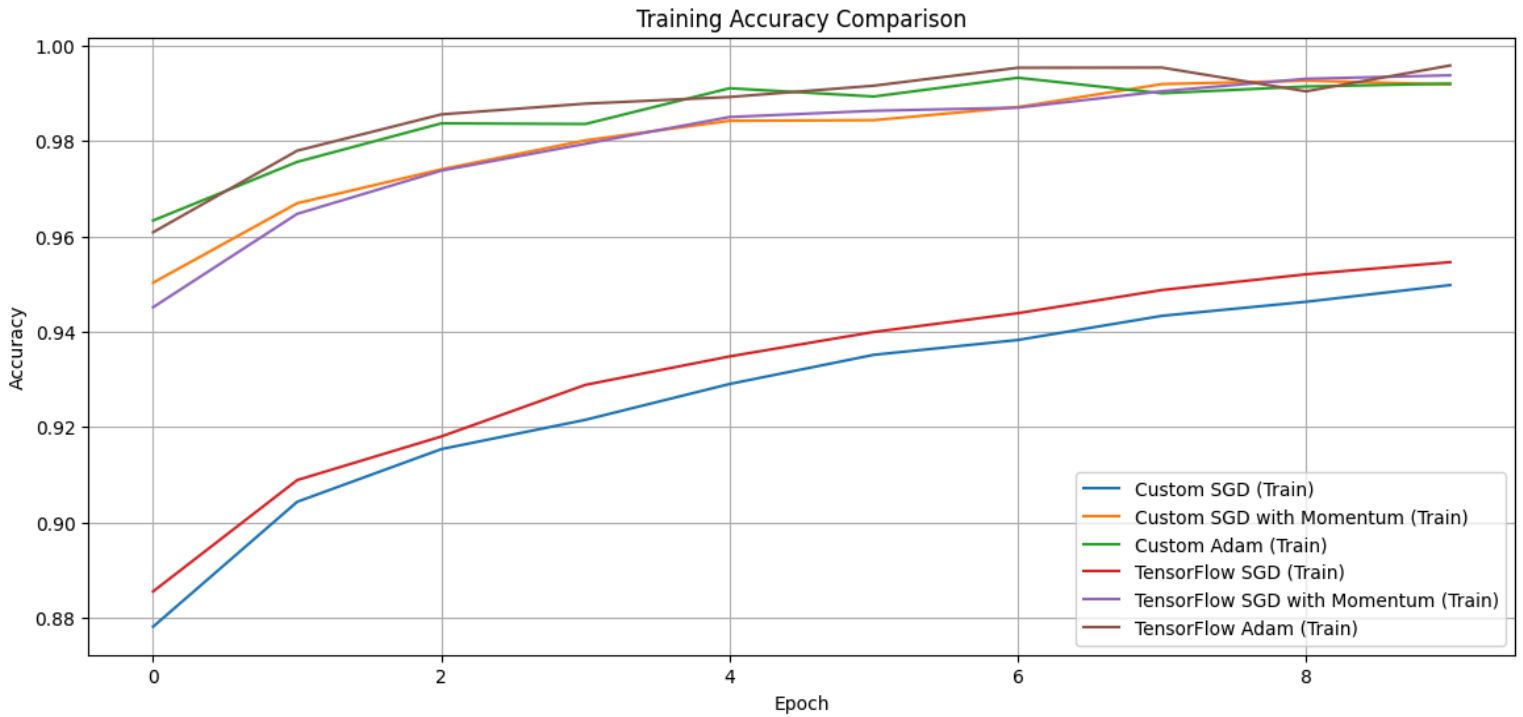
The neural network used for this evaluation is a simple yet effective fully connected feedforward neural network designed for the MNIST dataset. Its structure is as follows:

- Input Layer: 784 neurons (28x28 pixels flattened into a 1D vector)
- Hidden Layer 1: 128 neurons, ReLU activation
- Hidden Layer 2: 64 neurons, ReLU activation
- Output Layer: 10 neurons (one for each digit, 0–9), Softmax activation
- Loss Function: Sparse Categorical Crossentropy
- Metric: Accuracy

**This is a comparison table of all the results**

Optimizer	Final Test Accuracy
Custom SGD	0.9468
TensorFlow SGD	0.9524
Custom SGD with Momentum	0.9758
TensorFlow SGD with Momentum	0.9791
Custom Adam	0.9756
TensorFlow Adam	0.9796

## Visualization of the model's output



These two charts (Training Accuracy Comparison and Test Accuracy Comparison) illustrate the accuracy progression over epochs for both custom and TensorFlow-built optimizers.

## Key Observations

The results and comparisons of our custom implementations yield the following insights:

SGD achieved 94.68% accuracy on the MNIST dataset and showed its capability to optimize the neural network. However, it exhibited slower convergence compared to more advanced optimization methods. TensorFlow SGD outperformed it with a slight difference as it achieved an accuracy of 95.24%.

SGD with momentum achieved 97.58% accuracy on the MNIST dataset, which showed improvement on the normal SGD, which indicates that momentum helped overcome oscillations in the optimization path and accelerated convergence. TensorFlow SGD with Momentum achieved an accuracy of 97.91%, which closely matched the custom implementation.

ADAM achieved 97.56% accuracy on the MNIST dataset, highlighting the effectiveness of Adam's combination of momentum and adaptive learning rates. Which closely matched TensorFlow's built-in Adam optimizer

## Conclusion

This study demonstrated the critical role of optimization algorithms in training deep learning models, emphasizing their impact on accuracy, convergence speed, and computational efficiency. By focusing on three representative optimizers—SGD, SGD with Momentum, and Adam—we highlighted the nuances of their performance and trade-offs.

Key takeaways from the analysis include:

SGD serves as a baseline optimizer, offering simplicity and robust performance, though it struggles with slower convergence and oscillations in complex optimization landscapes.

SGD with Momentum effectively mitigates the limitations of standard SGD, achieving faster convergence and higher accuracy by incorporating historical gradients to smooth updates.

Adam, combining momentum and adaptive learning rates, stands out for its efficiency and adaptability, performing well across various tasks with minimal parameter tuning.

The results validate the effectiveness of advanced optimizers, with TensorFlow implementations slightly outperforming custom implementations due to additional optimizations and enhancements. The study underscores the importance of selecting the right optimizer based on the specific requirements of the task, as each optimizer has distinct strengths and limitations.

Future work could extend this comparison to more complex datasets and architectures, exploring the interplay between optimizers and other factors such as learning rate schedules, batch sizes, and regularization techniques, further enriching our understanding of optimization in deep learning.

# References

1. Bock, S., Goppold, J., & Weiß, M. (2018, April 27). **An improvement of the convergence proof of the ADAM-Optimizer**. arXiv.org. <https://arxiv.org/abs/1804.10587>
2. Bushaev, V. (2018, June 21). **Stochastic Gradient Descent with momentum** - Towards Data Science. Medium. <https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d>
3. Cutkosky, A., & Mehta, H. (2020, November 21). **Momentum improves normalized SGD**. PMLR. <https://proceedings.mlr.press/v119/cutkosky20b.html>
4. Goodfellow, I., Bengio, Y., & Courville, A. (2016). **Deep learning**. MIT Press.
5. Learning optimizers in deep learning made simple. (2024, October 28). ProjectPro. <https://www.projectpro.io/article/optimizers-in-deep-learning/983>
6. Optimizers: Maximizing accuracy, speed, and efficiency in deep learning. (2024, June 25). CloudThat Resources. <https://www.cloudthat.com/resources/blog/optimizers-maximizing-accuracy-speed-and-efficiency-in-deep-learning>
7. S. Boyd and L. Vandenberghe, **Convex Optimization**. Cambridge, UK: Cambridge University Press, 2004.
8. "SGD with Momentum Explained," Papers with Code. [Online]. Available: <https://paperswithcode.com/method/sgd-with-momentum> [Accessed: Dec. 31, 2024]
9. Ruder, S. (2016, September 15). **An overview of gradient descent optimization algorithms**. arXiv.org. <https://arxiv.org/abs/1609.04747>
10. "Optimization: Stochastic Gradient Descent," CS231n: Convolutional Neural Networks for Visual Recognition, Stanford University. [Online]. Available: <https://cs231n.github.io/optimization-1>. [Accessed: Dec. 31, 2024].
11. J. Brownlee, "Adam Optimization from Scratch," Machine Learning Mastery. [Online]. Available: <https://machinelearningmastery.com/adam-optimization-from-scratch>. [Accessed: Dec. 31, 2024].