Zewail City of Science and Technology
School of Computational Science and AI

Math 303
Linear and Nonlinear Programming

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

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 Grenadian 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 θ, 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 (η): $learning\ rate$

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

The above equation is used to update the hyperparameter (θ), 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]

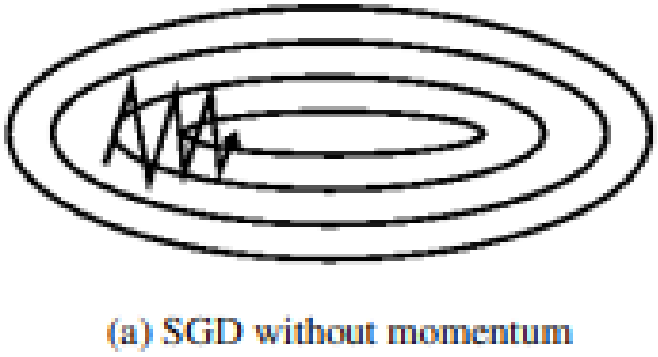


Figure 1:*hesitant progress towards local optimum. Adapted from [11]*

Python Implementation:

```
class SGD:
    def __init__(self, learning_rate=0.01):
        self.learning_rate = learning_rate
    def apply_gradients(self, gradients, variables):
        for grad, var in zip(gradients, variables):
            var.assign_sub(self.learning_rate * 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.

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

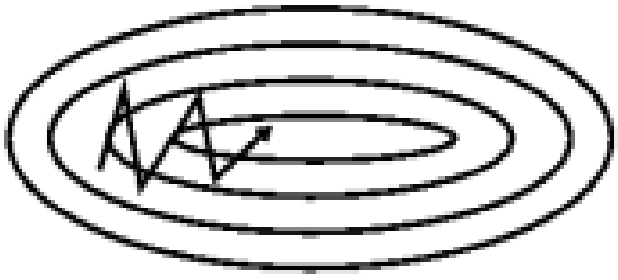
Velocity update equation incorporating update history

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

Parameter update

where:

- γ: $momentum\ coefficient$ (~ 0.9)
- v: $Direction\ vector$
- t: $time\ step$
- η: learning rate



(b) SGD with momentum

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

Python Implementation:

```
class SGDMomentum:
    def __init__(self, learning_rate=0.01, momentum=0.9):
        self.learning_rate = learning_rate
        self.momentum = momentum
        self.velocities = {}
    def apply_gradients(self, gradients, variables):
        for i, (grad, var) in enumerate(zip(gradients, variables)):
            if i not in self.velocities:
                self.velocities[i] = tf.zeros_like(var)
            velocity = self.momentum * self.velocities[i] + grad
            self.velocities[i] = velocity
            var.assign_sub(self.learning_rate * velocity)
```

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 θ_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]

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

where:

ϵ: *Constant to address division by zero*

η: learning rate

t: *time step*

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

$G_{t,ii}$: *Summation of the SG for* θ_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]

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1\, g^2{}_t \tag{5}$$

Updates the exponentially weighted moving average

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

Updates the hyperparameter θ

where:

ϵ: $Constant\ to\ address\ division\ by\ zero$

η: 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]

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

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.

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

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

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

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

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

where:

ϵ: *Constant to address division by zero*

η: 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*

β_1, β_2: *hyperparameters controlling decay rate*

Python Implementation:

```
class Adam:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-7):
        self.learning_rate = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = {}
        self.v = {}
        self.t = 0
    def apply_gradients(self, gradients, variables):
        self.t += 1
        for i, (grad, var) in enumerate(zip(gradients, variables)):
            if i not in self.m:
                self.m[i] = tf.zeros_like(var)
                self.v[i] = tf.zeros_like(var)

            self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * grad
            self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * tf.square(grad)

            m_hat = self.m[i] / (1 - self.beta1 ** self.t)
            v_hat = self.v[i] / (1 - self.beta2 ** self.t)
            var.assign_sub(self.learning_rate * m_hat / (tf.sqrt(v_hat) +
                           self.epsilon))
```

Results and Comparisons

The above implementation was trained, tested, and evaluated on the **MNIST** dataset. Using a custom neural network, the results were then compared to the built-in optimizers in the TensorFlow library. The full code can be found at this 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 Cross-Entropy
- Metric: Accuracy

Model Keras Implementation

```
def create_model():
    return tf.keras.Sequential([
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
```

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

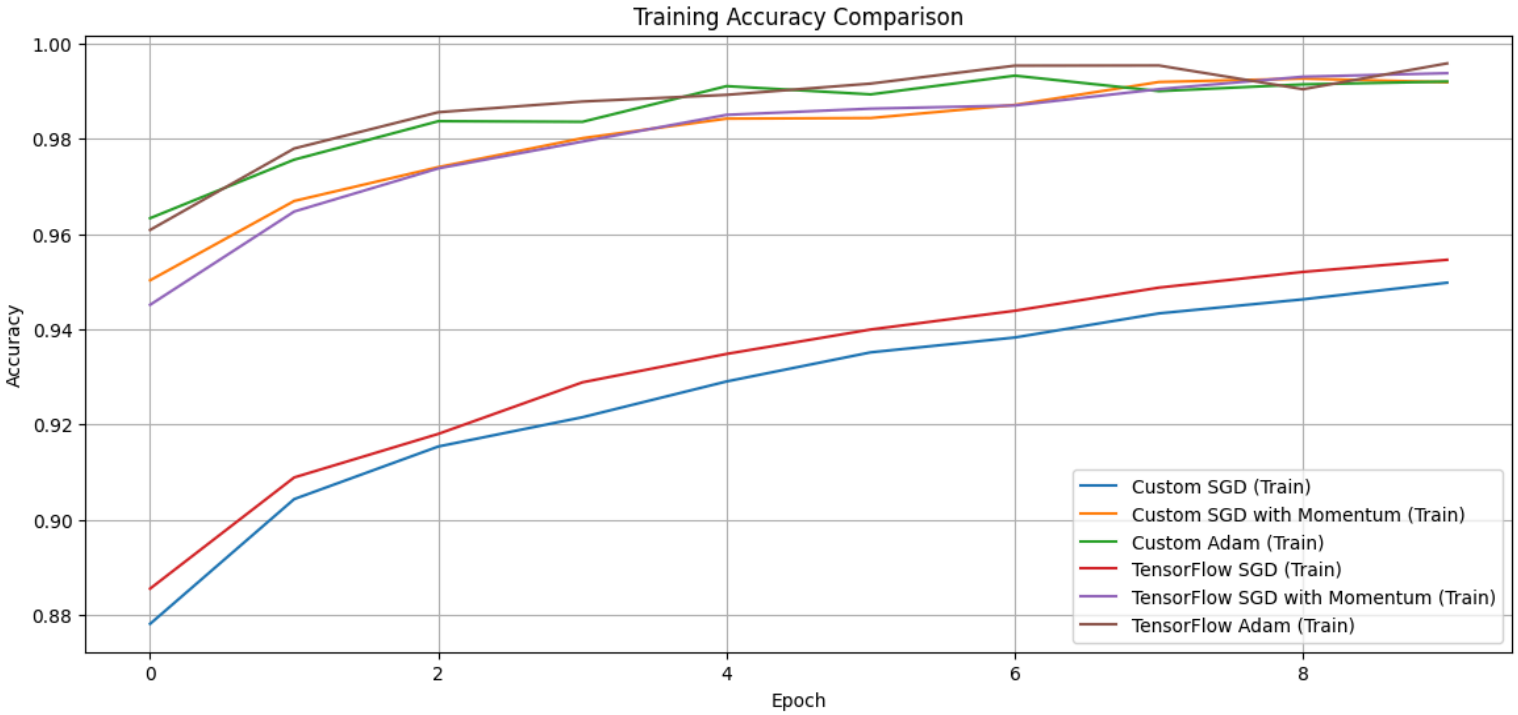


Figure 3: *A comparison chart between optimizers performance during training*

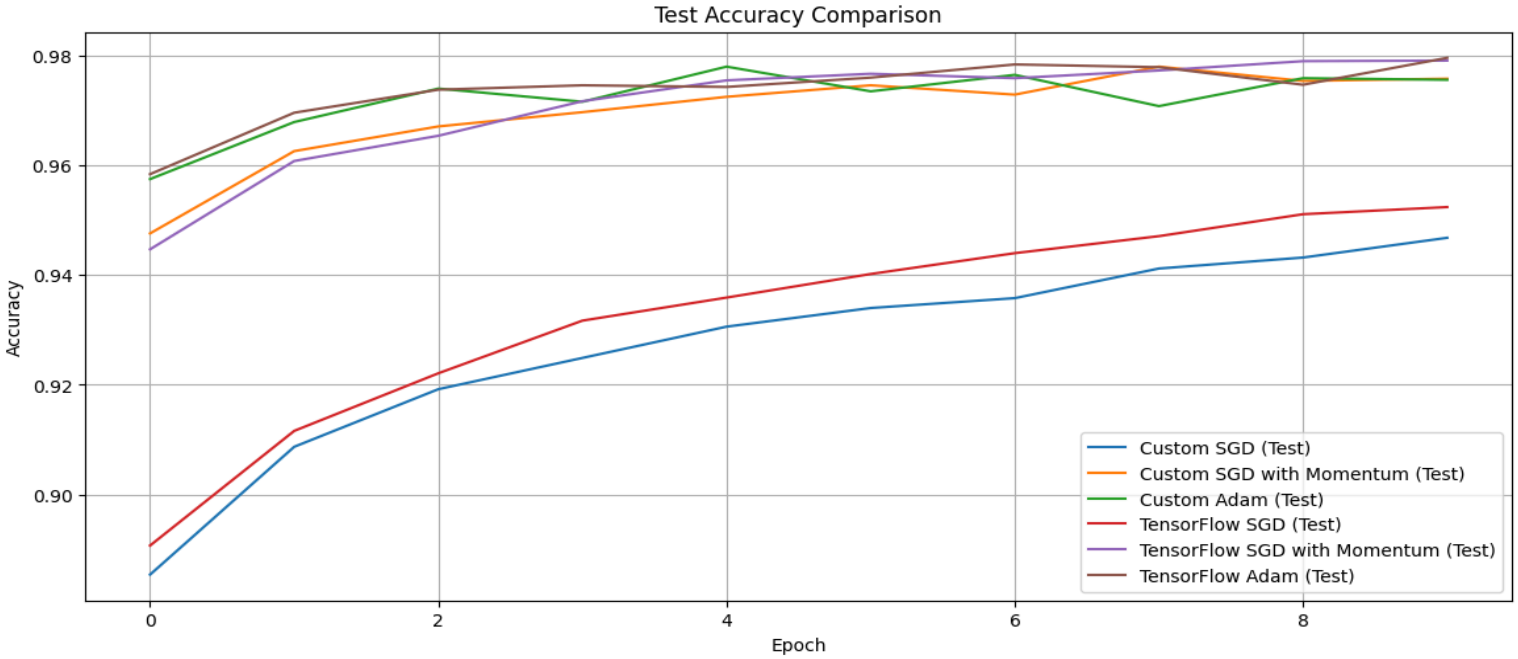


Figure 3: *A comparison chart between optimizers performance during testing*

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

Throughout this report, various optimization techniques and their implementation has been discussed. It was shown that the optimizer’s efficiency has a great impact on the training speed and the model’s accuracy and efficiency. For each of stochastic gradient descent (SGD), Momentum based stochastic gradient descent, and adaptive momentum approaches (ADAM) it was found that each approach suffered a critical issue which hindered its performance. Building on the previous models, the momentum based approach provided a smoother, faster descent than traditional SGD. While adam combined multiple optimizers such as AdaGrad, RMSprop, and traditional momentum approaches to resolve the learning rate inconsistency and address the vanishing gradient problems faced by prior approaches. Adam was able to achieve a decent accuracy although it is worth noting that SGD with momentum performed better. This is due to the dataset size and training iterations. For larger sized datasets and higher order loss functions, Adam is known to perform exceptionally well with a smooth steady decent as it incorporates the merits of having a direction vector, frequently updated parameters with adaptive updates based on the parameters update frequency, historical data, and a momentum parameter to achieve a robust optimizer.

In conclusion, All optimizers were noticed to perform well on the given dataset and it is worth testing on larger datasets to show higher contrast in performance and modifying the existing implementation for potential improvements.

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].