Comparative Analysis of Gradient Descent Methods and Optimization Techniques

Hamza Rashid

Math 387, McGill University

1 Introduction

In the field of deep learning, gradient descent is the predominant algorithm used to minimize the cost function and optimize model parameters. The size of the dataset on which the algorithm is performed can have a drastic impact on its accuracy and convergence behaviour. Stochastic Gradient Descent (SGD), Mini-batch Gradient Descent, and Batch Gradient Descent each offer various advantages and drawbacks in different computational and data scenarios. In addition, there are various optimization techniques built on top of Gradient descent that improve convergence speed, reduce oscillation, and adapt learning rates. This research aims to compare these methods systematically, focusing on their convergence rates, computational efficiency, and applicability to various machine learning models and datasets.

2 Background Information

2.1 Gradient Descent

Gradient Descent is a first-order iterative method for finding a minimum value of a function. In the context of deep learning, we wish to minimize the cost function

$$J : \mathbb{R}^d \to \mathbb{R}$$

which is typically the average of the loss function $\mathcal{L}$ – characterization of the model's error on a single instance – taken over a given quantity of training samples. The algorithm iteratively subtracts a scaling $\alpha > 0$, called the learning rate, of the cost's gradient ∇J from the current estimate of the minimum x_n. This effectively moves x_n along the direction of steepest descent, improving the estimate of a (not necessarily global) minimum in gradual, α-sized steps. Concretely, the estimate is updated as follows:

$$x_{n+1} := x_n - \alpha \nabla J(x_n) \tag{1}$$

where x_0 is an initial guess. Note that α can be adjusted across iterations.

In the case that J is γ-smooth (equivalently, ∇J is γ-Lipschitz) and convex, we can show that the Gradient Descent method has convergence rate $\mathcal{O}(\frac{1}{N})$, N the number of iterations:

Proof. The Lipschitz condition gives

$$\|\nabla J(x_{n+1}) - \nabla J(x_n)\| \leq \gamma \|x_{n+1} - x_n\| \tag{2}$$

where $\|\cdot\|$ is the Euclidean norm in $\mathbb{R}^d$.

And, the convexity of J yields

$$\nabla J(x_n) \leq \nabla J(x^*) + \nabla J(x_n)^\top (x_n - x^*) \tag{3}$$

where x^* is a minimizer of ∇J.

We then note that

$$\begin{aligned}
&\left|J(x) - J(y) - \nabla J(y)^\top (x-y)\right| \\
&= \left|\int_0^1 \nabla J(y + t(x-y))^\top (x-y)dt - \nabla J(y)^\top (x-y)\right| \\
&\leq \int_0^1 \left|(\nabla J(y+t(x-y)) - \nabla J(y))^\top (x-y)\right| dt \qquad \text{by triangle inequality} \\
&\leq \int_0^1 \|\nabla J(y+t(x-y)) - \nabla J(y)\| \cdot \|x-y\| \, dt \qquad \text{by Cauchy-Schwarz} \\
&\leq \int_0^1 \gamma t \|x-y\|^2 \, dt \qquad \text{by (2)} \\
&= \frac{\gamma}{2} \|x-y\|^2 \qquad (4)
\end{aligned}$$

where the first equality follows from the fundamental theorem of calculus.

Now suppose we run the algorithm with $\alpha := \frac{1}{\gamma}$. Then,

$$\begin{aligned}
\nabla J(x_{n+1}) &\leq \nabla J(x_n) + \nabla J(x_n)^\top (x_{n+1} - x_n) + \frac{\gamma}{2}\|x_{n+1} - x_n\|^2 \qquad \text{by (4)} \\
&= \nabla J(x_n) + \nabla J(x_n)^\top \left(-\frac{1}{\gamma}\nabla J(x_n)\right) + \frac{\gamma}{2}\left\|\frac{1}{\gamma}\nabla J(x_n)\right\|^2 \qquad \text{by (1)} \\
&= \nabla J(x_n) - \frac{1}{\gamma}\|\nabla J(x_n)\|^2 + \frac{1}{2\gamma}\|\nabla J(x_n)\|^2 \\
&= \nabla J(x_n) - \frac{1}{2\gamma}\|\nabla J(x_n)\|^2 \qquad (5)
\end{aligned}$$

Then (3) and (5) yield

$$\begin{aligned} J(x_{n+1}) - J(x^*) &\leq \nabla J(x_n)^\top (x_n - x^*) - \frac{1}{2\gamma} \left\|\nabla J(x_n)\right\|^2 \\ &= \gamma (x_n - x_{n+1})^\top (x_n - x^*) - \frac{\gamma}{2} \left\|x_n - x^*\right\|^2 \\ &= \frac{\gamma}{2} \left(\left\|x_n - x^*\right\|^2 - \left\|x_{n+1} - x^*\right\|^2 \right) \end{aligned}$$

Therefore,

$$\begin{aligned} N \cdot (J(x_{i+1}) - J(x^*)) &\leq \frac{\gamma}{2} \sum_{i=0}^{N-1} \left(\left\|x_i - x^*\right\|^2 - \left\|x_{i+1} - x^*\right\|^2 \right) \\ &= \frac{\gamma}{2}(x_0 - x^*) \end{aligned}$$

Thus,

$$J(x_{i+1}) - J(x^*) \leq \frac{\gamma}{2}(x_0 - x^*) \cdot \frac{1}{N}$$

$\square$

2.2 Batch Size

The size of the dataset on which the Gradient descent algorithm is performed, called the batch size, can have a drastic impact on its convergence behaviour. Batch Gradient Descent computes ∇J over the entire dataset. Mini-Batch Gradient Descent computes the gradient over fixed-sized subsets until covering the entire set, and Stochastic Gradient Descent is a variation of Mini-Batch, with batch size equal to one.

Formally, if the training dataset contains N examples and $\mathcal{L}_i$ denotes the error of the model's prediction on the i^{th} sample, then at the $(n+1)^{th}$ epoch (pass through the entire dataset) of Batch Gradient Descent, the update is given by:

$$x_{n+1} = x_n - \alpha \nabla J(x_n) = x_n - \frac{\alpha}{N} \sum_{i=1}^{N} \nabla \mathcal{L}_i(x_n) \tag{6}$$

The algorithm has the following structure:

Algorithm 1 Batch Gradient Descent

```
1: Initialize parameters $\theta$ randomly or with some predetermined values
2: Set learning rate $\alpha$
3: Set number of iterations $M$
4: for $i = 1$ to $M$ do
5:     $\nabla J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{L}_i(\theta)$
6:     $\theta = \theta - \alpha \nabla J(\theta)$
7: end for
```

In Mini-Batch Gradient Descent, the parameters are updated after processing each batch. Let B represent the batch size and k the batch index, with the assumption that the first batch includes training examples from 1 to B, the second batch from $B+1$ to $2B$, and so forth. The general update equation for any batch $k \geq 0$, is given by:

$$x_{n+1,k+1} = x_{n+1,k} - \frac{\alpha}{B} \sum_{i=kB+1}^{(k+1)B} \nabla \mathcal{L}_i(x_{n+1,k}) \tag{7}$$

Here, $x_{n+1,0}$ is initialized as $x_{n,\frac{N}{B}-1}$ at the start of each epoch.

An outline for the algorithm is given by:

Algorithm 2 Mini-Batch Gradient Descent

```
 1: Initialize parameters $\theta$ randomly or with some predetermined values
 2: Set learning rate $\alpha$
 3: Set number of iterations $M$
 4: Set batch size $B$
 5: for $i = 1$ to $M$ do
 6:     for $j = 1$ to (N / B) do
 7:         Select a mini-batch of $B$ training examples
 8:         Compute the gradient using the mini-batch and update the parameters:
 9:             $\nabla J(\theta) = \frac{1}{B}\sum_{i=1}^{B} \nabla \mathcal{L}_i(\theta)$
10:             $\theta = \theta - \alpha \nabla J(\theta)$
11:     end for
12: end for
```

Overall, in Batch Gradient Descent, the parameters of ∇J are updated once every epoch with the entire dataset, while in Mini-Batch, they are updated $\frac{N}{B}$ times per epoch using subsets of the data.

The Batch method achieves higher accuracy and stable convergence towards a local minimum near x_0 by utilizing the entire dataset to compute the gradient. In contrast, Mini-Batch Gradient Descent converges more rapidly towards the minimum as it updates the gradient more frequently using smaller batches - an advantage that is offset by less stable convergence and greater variance, issues that are particularly pronounced in Stochastic Gradient Descent.

Our analysis aims to highlight the advantages and limitations of each method in an experimental setting, facilitating more informed decisions in the selection of gradient descent techniques for specific machine learning challenges. In addition, this paper will delve into various optimization techniques that enhance the performance of gradient descent methods by addressing their inherent drawbacks. Specifically, we will examine the roles of Momentum, RMSprop, and Adam algorithms in improving convergence behavior and reducing variance. These optimizations represent significant advancements in training efficiency and are critical for achieving faster and more reliable results in complex machine learning tasks.

3 Optimizations

3.1 Polyak's Momentum (Heavy Ball Method)

The Gradient Descent algorithm can be optimized further by taking previous iterations into account. The Momentum method achieves this by updating its current estimate of the minimum x_n in the direction of an exponentially weighted moving average of the gradients. Formally, the update is given by:

$$x_{n+1} := x_n - \alpha V_{n+1} \tag{8}$$

where:

$$V_{n+1} := \beta V_n + (1-\beta)\nabla J(x_n) \tag{9}$$

The definition of V_{n+1} is the exponential smoothing formula applied to the cost's gradient $\nabla J(x_n)$, with smoothing factor $\beta \in [0,1]$ ($\beta := 0.9$ is often set in practice, roughly corresponding to a weighted average of the last ten gradients).

Algorithmically, we have

Algorithm 3 Gradient Descent with Momentum

```
 1: Initialize parameters $\theta$ randomly or with some predetermined values
 2: Set learning rate $\alpha$
 3: Set smoothing factor $\beta := 0.9$
 4: Set number of iterations $M$
 5: Initialize velocity $V := 0$
 6: for $i = 1$ to $M$ do
 7:         $\nabla J(\theta) = \text{ComputeGradient}(\theta)$
 8:     Update velocity using momentum:
 9:         $V = \beta V + (1-\beta)\nabla J(\theta)$
10:     Update parameters using velocity:
11:         $\theta = \theta - \alpha V$
12: end for
```

A common and useful reformulation of this method is:

$$x_{n+1} := x_n - V_{n+1} \tag{10}$$

where:

$$V_{n+1} := \beta V_n + \alpha \nabla J(x_n) \tag{11}$$

which achieves the exponential decay since the β's inductively multiply the past momentum terms, but changes the scale of the updates as the $1-\beta$ term is omitted and the learning rate only factors the gradient.

By updating the estimate with an averaging of the gradients, Momentum uses the dominant direction from past iterations to accelerate convergence towards the minimum. As a result, it can reduce oscillations in directions that diverge from the minimum, since it gives precedence to the dominant direction of convergence from previous gradients.

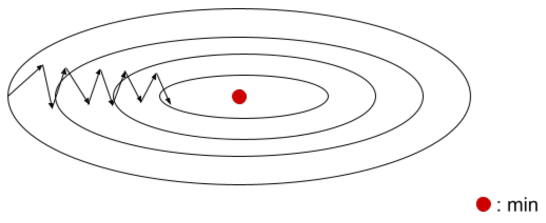


Figure 1: Example of Gradient Descent without Momentum

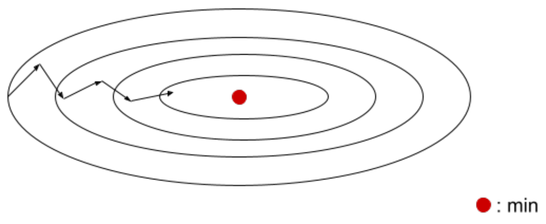


Figure 2: Example of Gradient Descent with Momentum

To understand Figures 1 and 2 more clearly, let the contours depict elevation within a neighborhood of a surface in $\mathbb{R}^3$, so that the fluctuations along directions that stray from the minimum tend to average out to zero, while progress in the z-component is given a bit more weight.

Such oscillations necessitate more iterations of the algorithm for convergence, and can impose a bottle-neck on the learning rate. By giving precedence to the dominant direction of convergence from past gradients, and thereby dampening oscillations in directions that diverge from the minimum, Momentum can decrease the number of iterations needed for convergence. Furthermore, if the cost function is non-convex (particularly in the case that multiple local minima exist), Momentum can help escape the sub-optimal local minima since it averages the gradients instead of only relying on the current one, which vanishes at such minima.

Despite offering faster convergence, the Momentum optimization does not always enhance performance[1]. Intuitively, this can occur if a neighborhood of the minimum is steep, in which case the accumulated momentum will likely cause the estimate to overshoot. In particular, when dealing with strongly convex functions (when there is quadratic lower bound on the growth of the function), Momentum Gradient Descent may perform worse than the standard method, and in some cases even fail to converge where the standard method succeeds. Nesterov-Accelerated Gradient, a variation of the Momentum method, addresses some of these problems; it converges for general convex functions, and has convergence rate $\mathcal{O}(\frac{1}{N^2})$.

3.2 Nesterov-Accelerated Gradient

Nesterov-Accelerated Gradient builds on top of Momentum by evaluating the gradient on a position that is slightly ahead of the accumulated average, called the look-ahead position. Concretely,

$$x_{n+1} := x_n - V_{n+1} \tag{12}$$

where:

$$V_{n+1} := \beta V_n + \alpha \nabla J(x_n - \beta V_n) \tag{13}$$

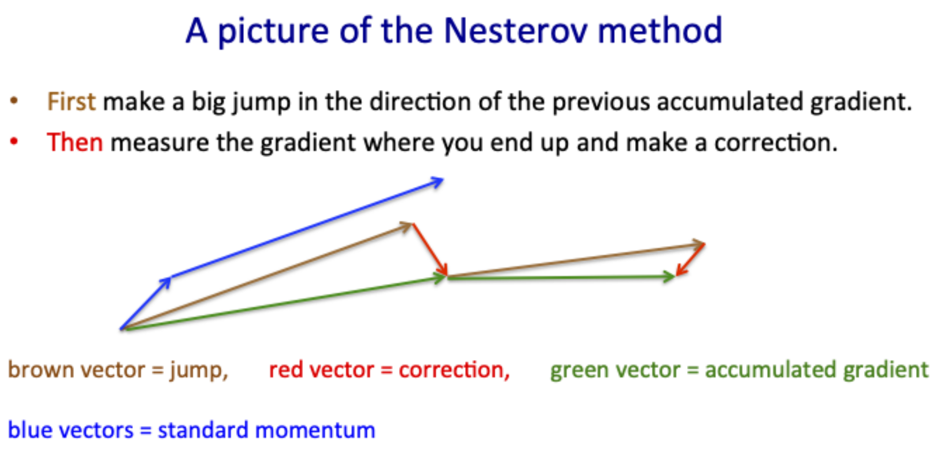
The connection to Momentum is in the reformulation given by (10) and (11). Since the term βV_n is used to update the parameters, the look-ahead $(x_n - \beta V_n)$ approximates the next update position.

The algorithm has the following form:

Algorithm 4 Nesterov Momentum

```
1: Initialize parameters $\theta$ randomly or with some predetermined values
2: Set learning rate $\alpha$
3: Set smoothing factor $\beta := 0.9$
4: Set number of iterations $M$
5: Initialize velocity $V := 0$
6: for $i = 1$ to $M$ do
7:     Compute the gradient of the cost function at the lookahead point:
8:         $\theta_{\text{lookahead}} = \theta - \beta V$
9:         $\nabla J(\theta_{\text{lookahead}}) = \text{ComputeGradient}(\theta_{\text{lookahead}})$
10:    Update velocity:
11:        $v = \beta v + \alpha \nabla J(\theta_{\text{lookahead}})$
12:    Update parameters using Nesterov momentum:
13:        $\theta = \theta - V$
14: end for
```

While Momentum computes the gradient at the current position before jumping in the direction of the updated accumulated gradients, Nesterov-Accelerated Gradient computes the gradient after updating the estimate in the direction of the previously accumulated gradients. Consequently, the latter method can often anticipate changes in the landscape more effectively, since computing the gradient from an approximation of the next position can help prevent the update from overshooting. Nesterov-Accelerated Gradient thereby inherits the advantages of Momentum, while also offering greater stability. Figure 3 gives a visual example.



A picture of the Nesterov method

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.

brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Figure 3: Nesterov Example (Source: Geoff Hinton Lecture 6C[4])

While Nesterov-Accelerated Gradient marks a significant advancement, the optimization techniques that follow are more closely aligned with Momentum.

3.3 RMSprop (Root Mean Square Propagation)

Similar in principle to the Momentum, RMSProp keeps a moving average of the (component-wise) squared gradient ∇J, and uses it normalize the learning rate for each weight and bias (components) of the input vector in each iteration. Concretely, the update is given by:

$$x_{n+1} := x_n - \alpha_n \nabla J \tag{14}$$

where, through flexibility of notation (which is ubiquitous in the surrounding literature),

$$\alpha_n := \frac{\alpha}{\sqrt{S_n} + \epsilon} \tag{15}$$

$$S_{n+1} := \beta Sn + (1 - \beta) \nabla J(x_n)^2 \tag{16}$$

Where $\epsilon > 0$ (typically, $\epsilon := 10^{-8}$) is intended to prevent division by a vanishing quantity, $\alpha > 0$ is initialized beforehand, and the operations involving ∇J and S_n are component-wise. It is recommended to set $\alpha := 0.001$, $\beta := 0.9$.

A sample algorithm is given below:

Algorithm 5 RMSprop (Root Mean Square Propagation)

```
1: Initialize parameters $\theta$ randomly or with some predetermined values
2: Set learning rate $\alpha := 0.001$
3: Set decay rate $\beta := 0.9$
4: Set small constant $\epsilon := 10^{-8}$
5: Set number of iterations $M$
6: Initialize cache $S := 0$
7: for $i = 1$ to $M$ do
8:     Compute the gradient of the cost function with respect to $\theta$:
9:         $\nabla J(\theta) = \text{ComputeGradient}(\theta)$
10:    Update cache:
11:        $S = \beta S + (1-\beta)(\nabla J(\theta))^2$
12:    Update parameters using RMSprop:
13:        $\theta = \theta - \alpha \left(\frac{\nabla J(\theta)}{\sqrt{S}+\epsilon}\right)$
14: end for
```

By normalizing the input parameters, RMSprop mitigates both exploding and vanishing gradients, leading to more stable convergence. Furthermore, the normalization process can decrease the parameters pointing in directions that diverge from the minimum if they are large, via division by the moving averages, and thereby reduce oscillation. Coupling the latter advantage with the improved convergence stability, RMSprop can allow one to increase the learning rate, as the updates are less likely to overshoot.

3.4 Adam (Adaptive Moment Estimation)

Adam directly combines Momentum with RMSprop. It updates the current estimate of the minimum x_n in the direction of an exponentially weighted moving average of the gradients, and keeps a moving average of the squared gradient ∇J to normalize the learning rate applied to each parameter. Letting β_1 denote the smoothing factor in (9) and β_2 that of (16), the update is given by:

$$x_{n+1} := x_n - \hat{\alpha_n}\hat{V_n} \tag{17}$$

where

$$\hat{\alpha_n} := \frac{\alpha}{\sqrt{\hat{S_n}} + \epsilon} \tag{18}$$

$$\hat{S_n} = \frac{S_n}{1-\beta_2^n} \tag{19}$$

$$\hat{V_n} = \frac{V_n}{1-\beta_1^n} \tag{20}$$

where α_n and V_n are given by (15) and (9), respectively, and (19) and (20) are bias corrections for the exponentially weighted moving averages, improving accuracy especially in early iterations. Adam Gradient Descent thereby inherits faster and more stable convergence by implementing the techniques in Momentum and RMSprop. It is recommended to set $\beta_1 := 0.9$ and $\beta_2 = 0.999$.

The algorithm can be outlined as follows:

Algorithm 6 Adam (Adaptive Moment Estimation)

```
 1: Initialize parameters $\theta$ randomly or with some predetermined values
 2: Set learning rate $\alpha := 0.001$
 3: Set decay rates for moment estimates $\beta_1 := 0.9$ and $\beta_2 := 0.999$
 4: Set small constant $\epsilon := 10^{-8}$
 5: Set number of iterations $M$
 6: Initialize moment variables $m = 0$, $v = 0$
 7: Initialize time step $t = 0$
 8: for $i = 1$ to $M$ do
 9:     Compute the gradient of the cost function with respect to $\theta$:
10:         $\nabla J(\theta) = \text{ComputeGradient}(\theta)$
11:     Update time step:
12:         $t = t + 1$
13:     Update biased first moment estimate:
14:         $m = \beta_1 m + (1-\beta_1)\nabla J(\theta)$
15:     Update biased second raw moment estimate:
16:         $v = \beta_2 v + (1-\beta_2)(\nabla J(\theta))^2$
17:     Correct bias in first moment:
18:         $\hat{m} = \frac{m}{1-\beta_1^t}$
19:     Correct bias in second moment:
20:         $\hat{v} = \frac{v}{1-\beta_2^t}$
21:     Update parameters using Adam:
22:         $\theta = \theta - \alpha\frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon}$
23: end for
```

Most of the recommended constants are given by the authors of the foregoing optimization techniques, and the results of empirical analyses. When it comes to the smoothing factor, the justification lies in finding a balance between averaging over all iterations or none, while also optimizing the model's performance.

4 Methodology

The first dataset consists of time series data, namely, we are given S&P 500 closing prices since the 1980's, and attempt to predict the closing prices on a subset of the data. The second dataset is the IMDB movie review sentiment dataset, constituting a binary classification problem - the review is either favourable or negative. For each dataset, we train a model over multiple batch sizes, testing each optimization technique for each batch size, namely, plain Gradient Descent, Momentum, RMSProp, and Adam.

The learning rate is initialized to 0.001 for each optimization method. We allow the number of epochs, choice of loss functions, and overall model architecture to vary between the two datasets since the metrics that measure the quality of the predictions on one dataset are not applicable to the other. For more details on the metrics, see [4], [5], and [7].

Due to hardware limitations on Google Colab, we perform tests on Mini-Batch Gradient Descent with batch sizes 1024, 512, and 128, in addition to Batch, for both datasets. For brevity, we will show the best performing optimization method for batch size 1024 and the full batch along with the standard method. The rest of the test results are available through the links:

- S&P 500 Times Series Notebook: `https://colab.research.google.com/drive/1Zx7kFs2HDtMjnKWyMSuR5iSDZVtZ7HTp?usp=sharing`
- IMDB Sentiment Analysis Notebook: `https://colab.research.google.com/drive/1FcYigXu8ZvxpnlmBoVG8vAI6pxomNYcC?usp=sharing`

For the time series dataset, which consists of 14022 samples, we train the model over 30 epochs, and test its performance under Batch Gradient Descent, and Mini-Batch with the aforementioned batch sizes.

For the sentiment analysis dataset, consisting of 25000 samples, we train the model over 12 epochs with the previously mentioned batch sizes, in addition to Batch.

The tables that follow display the metrics evaluated on the model after training on its last epoch. Note that the Loss metric is just the cost.

5 Results

5.1 Time Series Dataset

5.1.1 Batch Gradient Descent

Table 1: Plain (No Optimizer), Batch, S&P 500 Time Series Dataset

Metric	Value
Loss	0.0040
Mean Squared Error (MSE)	0.0081
Mean Absolute Error (MAE)	0.0542
Mean Absolute Percentage Error	79.6322
Loss Standard Deviation	4.7749×10^{-5}
MSE Standard Deviation	9.6422×10^{-5}
MAE Standard Deviation	8.3873×10^{-4}
MAPE Standard Deviation	5.3211

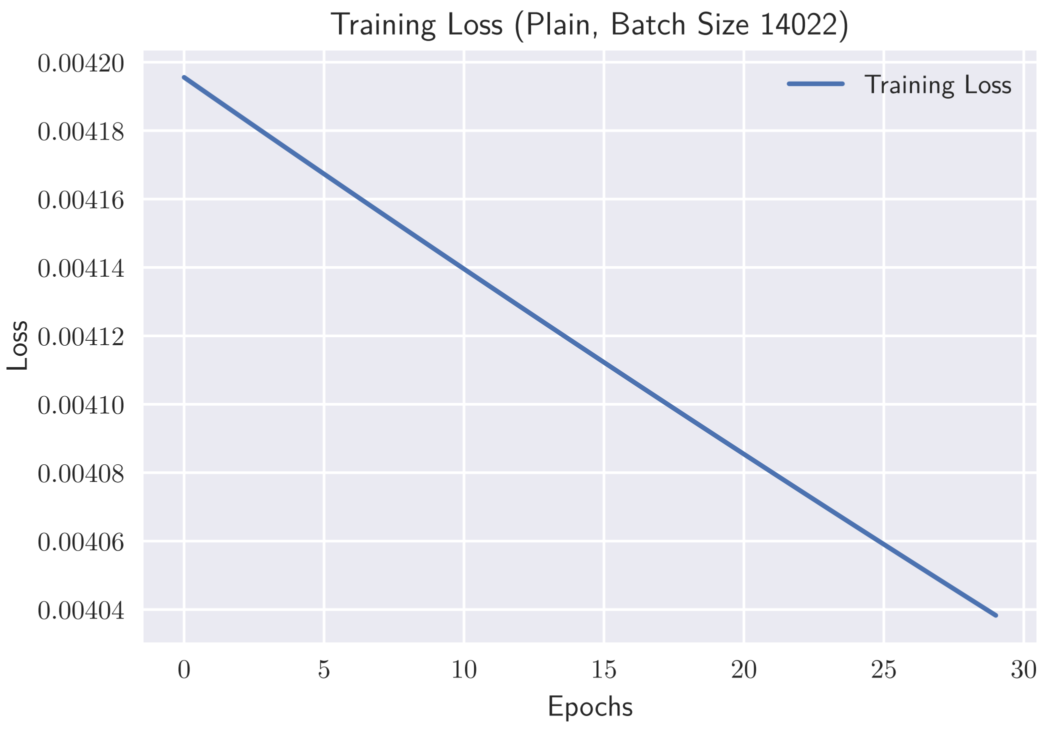


Figure 4: Training Loss: Plain, Batch, S&P 500 Time Series Dataset

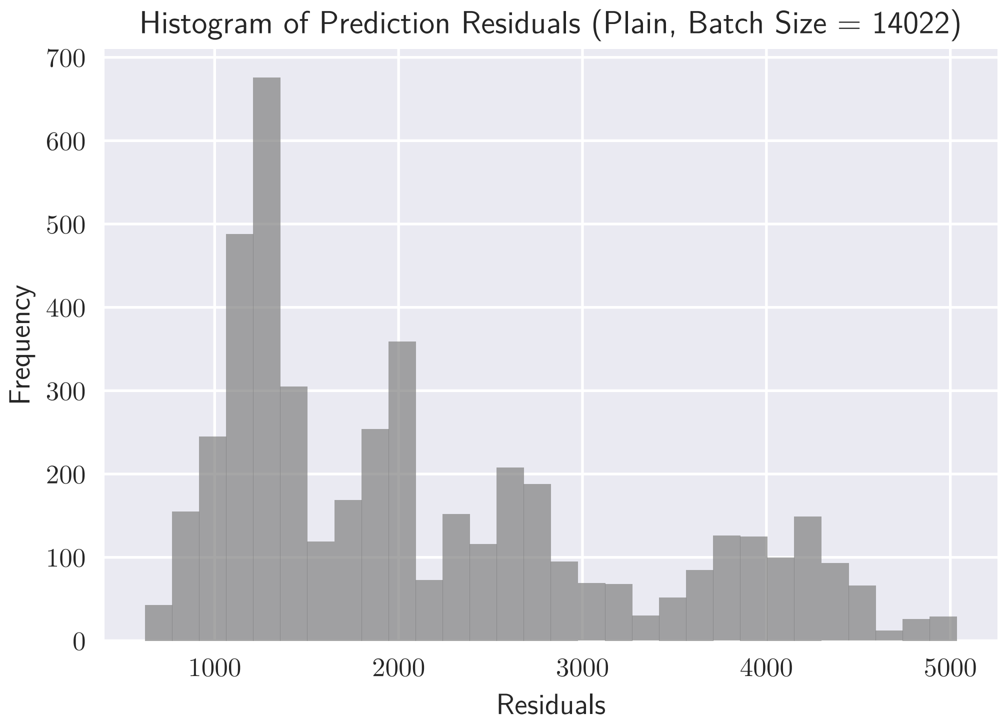


Figure 5: Residuals Histogram (Difference between Actual and Predicted Prices): Plain, Batch, S&P 500 Time Series Dataset

Table 2: Adam, Batch, S&P 500 Time Series Dataset

Metric	Value
Loss	0.0017
Mean Squared Error (MSE)	0.0035
Mean Absolute Error (MAE)	0.0491
Mean Absolute Percentage Error	361.2083
Loss Standard Deviation	0.0006442
MSE Standard Deviation	0.0013034
MAE Standard Deviation	0.0034232
MAPE Standard Deviation	125.2894

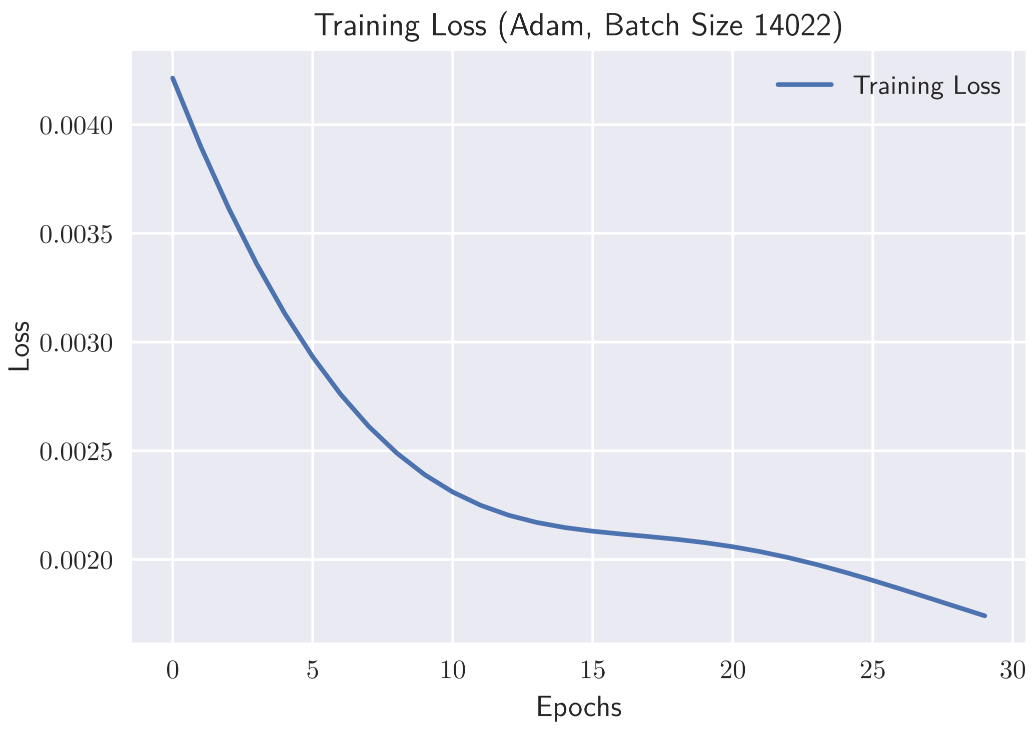


Figure 6: Training Loss: Adam, Batch, S&P 500 Time Series Dataset

A significant improvement as the model decreased its cost at a linear rate without any optimization algorithm.

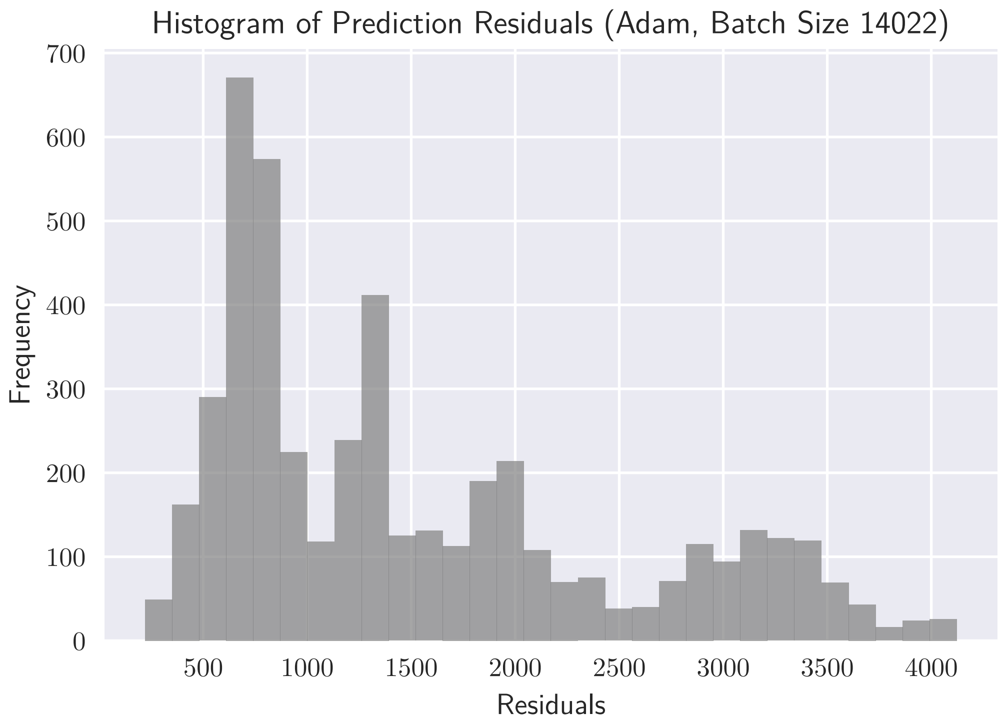


Figure 7: Residuals Histogram (Difference between Actual and Predicted Prices): Adam, Batch, S&P 500 Time Series Dataset

With the Adam optimization, the model's predictions are more frequently near the true value than without any optimization.

5.1.2 Mini-Batch Gradient Descent

Table 3: Plain, Batch Size 1024, S&P 500 Time Series Dataset

Metric	Value
Loss	0.0079
Mean Squared Error (MSE)	0.0159
Mean Absolute Error (MAE)	0.0893
Mean Absolute Percentage Error	488.91
Loss Standard Deviation	0.0014
MSE Standard Deviation	0.0029
MAE Standard Deviation	0.0038
MAPE Standard Deviation	125.33

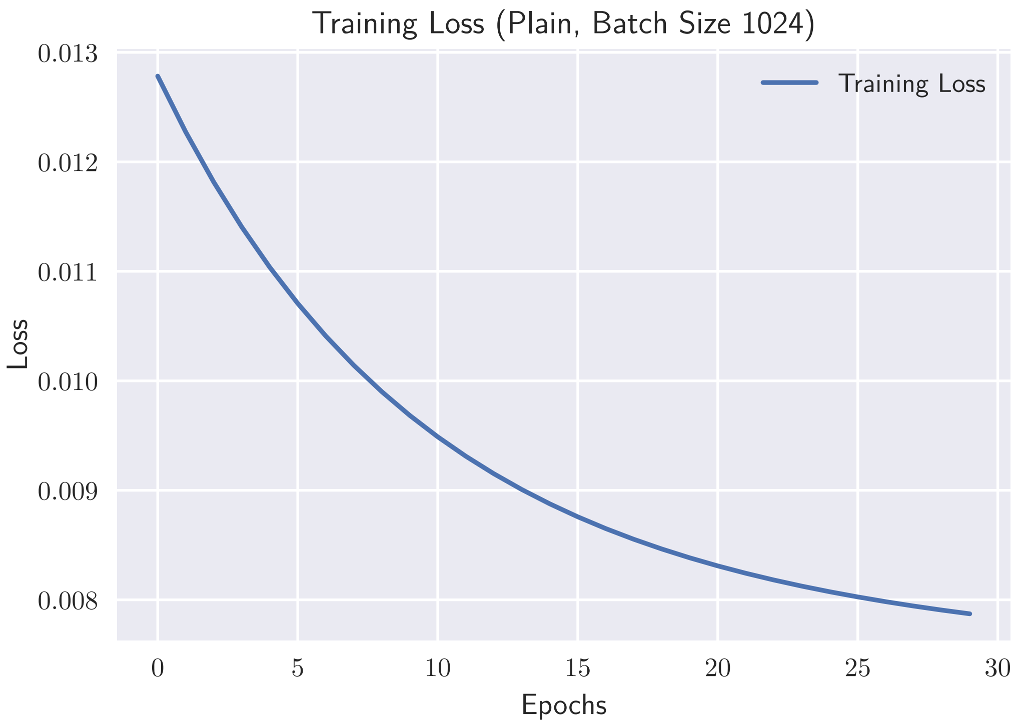


Figure 8: Training Loss: Plain, Batch Size 1024, S&P 500 Time Series Dataset

With a smaller batch size, the rate at which the model decreases its cost has significantly improved. However, the cost is larger than that obtained with the Batch method.

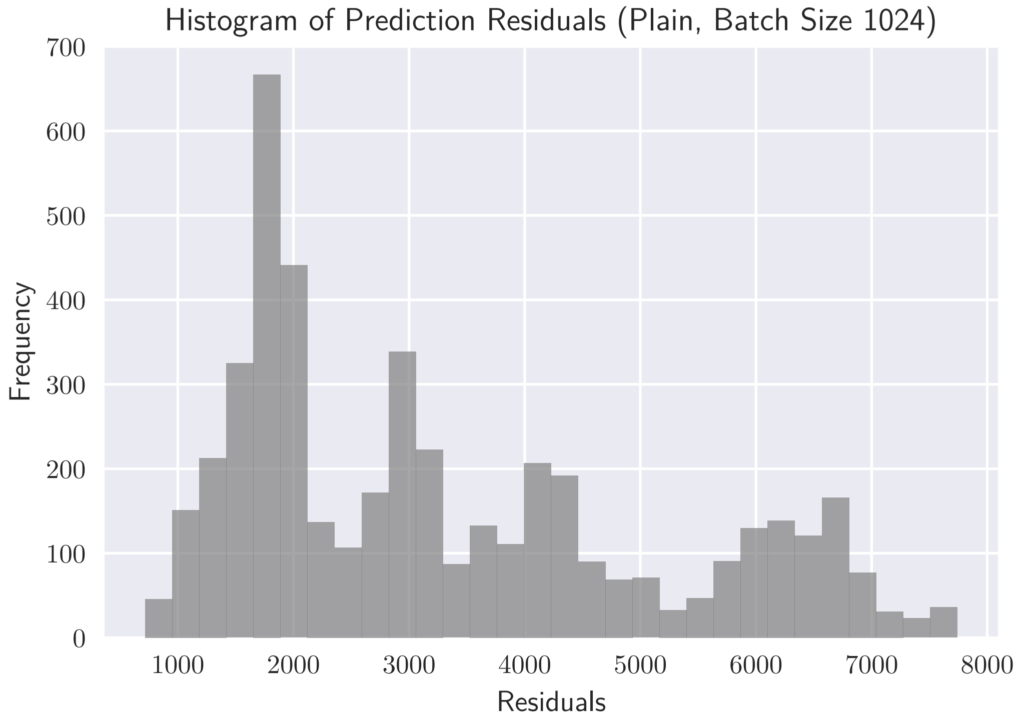


Figure 9: Residuals Histogram (Difference between Actual and Predicted Prices): Plain, Batch Size 1024, S&P 500 Time Series Dataset

The previous point is supported here, as the histogram does not show any considerable improvements.

Table 4: Adam, Batch Size 1024, S&P 500 Time Series Dataset

Metric	Value
Loss	2.51×10^{-6}
Mean Squared Error (MSE)	5.02×10^{-6}
Mean Absolute Error (MAE)	0.0012
Mean Absolute Percentage Error	5.07
Loss Standard Deviation	0.0011
MSE Standard Deviation	0.0022
MAE Standard Deviation	0.021
MAPE Standard Deviation	136.00

With this Mini-Batch configuration, the Adam optimizer yields a significant improvement in convergence, with the cost being orders of magnitude lower than that obtained without any optimization.

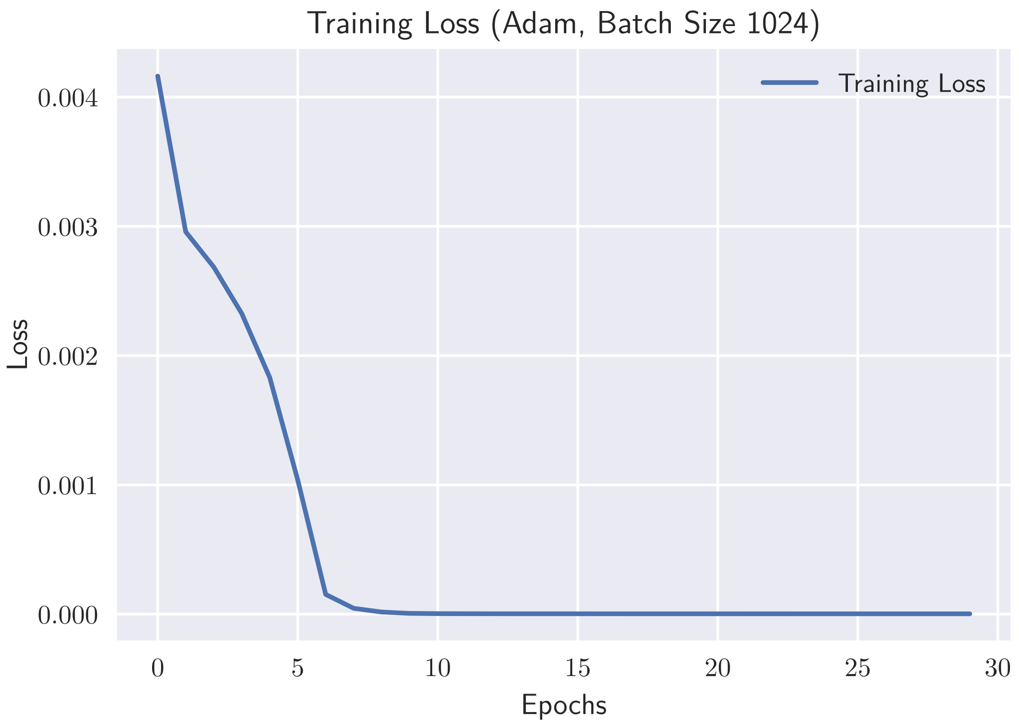


Figure 10: Training Loss: Adam, Batch Size 1024, S&P 500 Time Series Dataset

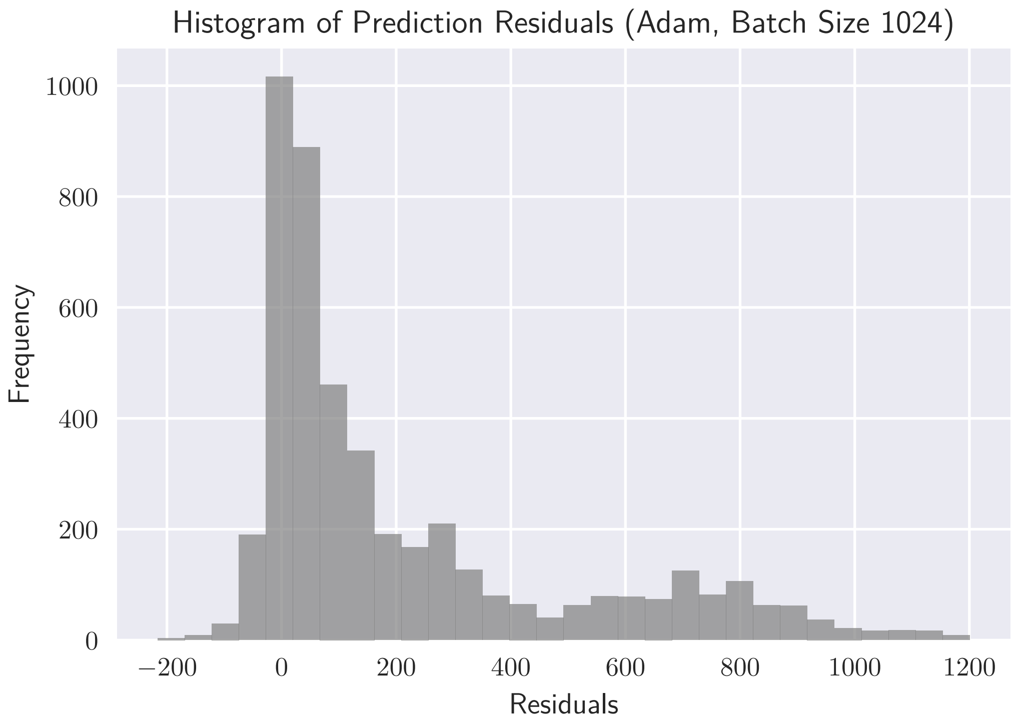


Figure 11: Residuals (Difference between Actual and Predicted Prices): Adam, Batch Size 1024, S&P 500 Time Series Dataset

The histogram indicates a higher frequency of predictions that are closer to the true value compared to the previous batch size configuration.

5.2 Sentiment Analysis Dataset

Table 5: Plain (No Optimizer), Batch, IMDB Sentiment Dataset

Metric	Value
Loss	0.6930
Accuracy	0.5121
Precision	0.5132
Recall	0.4700
Loss Standard Deviation	3.44×10^{-8}
Accuracy Standard Deviation	6.02×10^{-5}
Precision Standard Deviation	0.071
Recall Standard Deviation	0.075

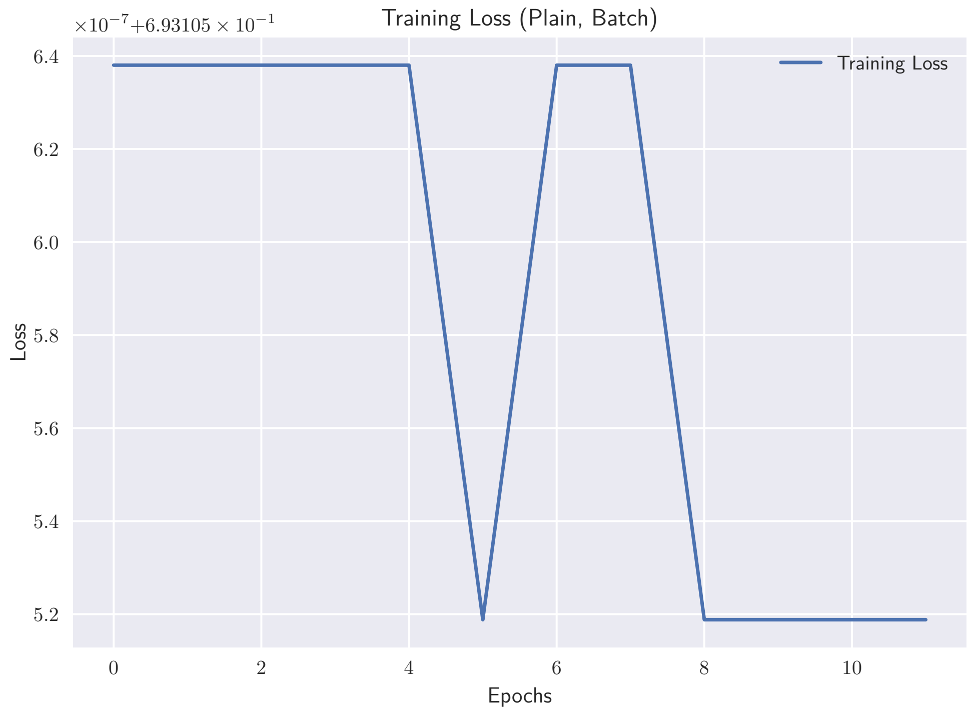


Figure 12: Loss: Plain Gradient Descent, Batch, IMDB Sentiment Dataset

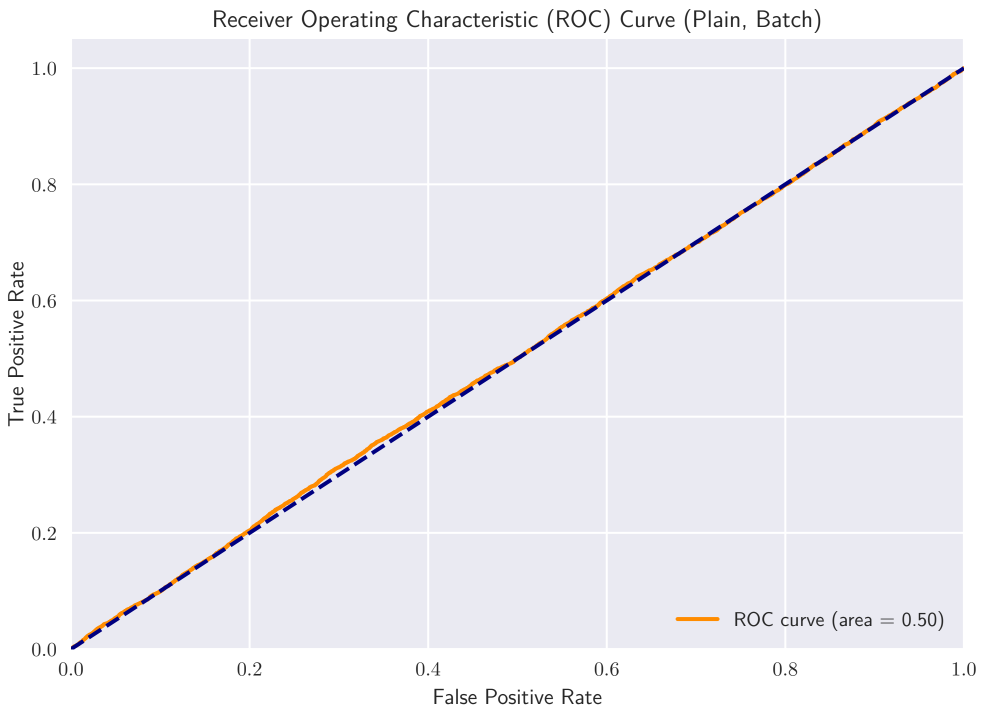


Figure 13: ROC Curve: Plain Gradient Descent, Batch, IMDB Sentiment Dataset

Table 6: Adam, Batch, IMDB Sentiment Dataset

Metric	Value
Loss	0.6320
Accuracy	0.7401
Precision	0.7490
Recall	0.7222
Loss Standard Deviation	0.0182
Accuracy Standard Deviation	0.0717
Precision Standard Deviation	0.0673
Recall Standard Deviation	0.1186

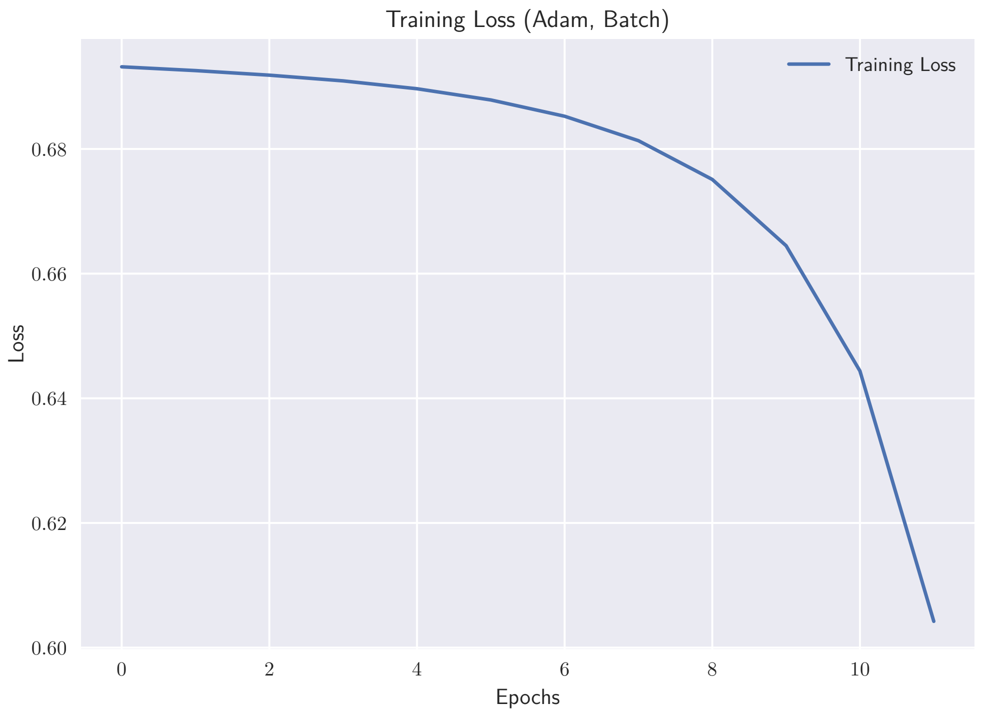


Figure 14: Loss: Adam, Batch, IMDB Sentiment Dataset

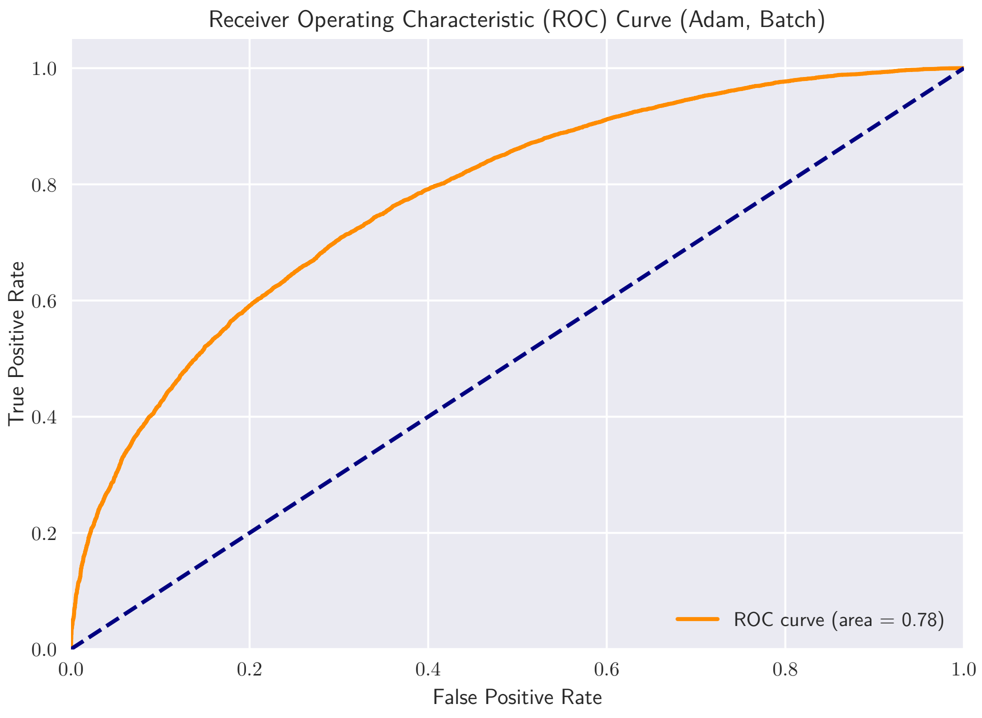


Figure 15: ROC Curve: Adam, Batch, IMDB Sentiment Dataset

5.2.1 Mini-Batch Gradient Descent

Table 7: Plain, Batch Size 1024, IMDB Sentiment Dataset

Metric	Value
Loss	0.6932
Accuracy	0.4971
Precision	0.4971
Recall	0.4966
Loss Standard Deviation	1.72×10^{-6}
Accuracy Standard Deviation	3.48×10^{-4}
Precision Standard Deviation	0.0360
Recall Standard Deviation	0.0312

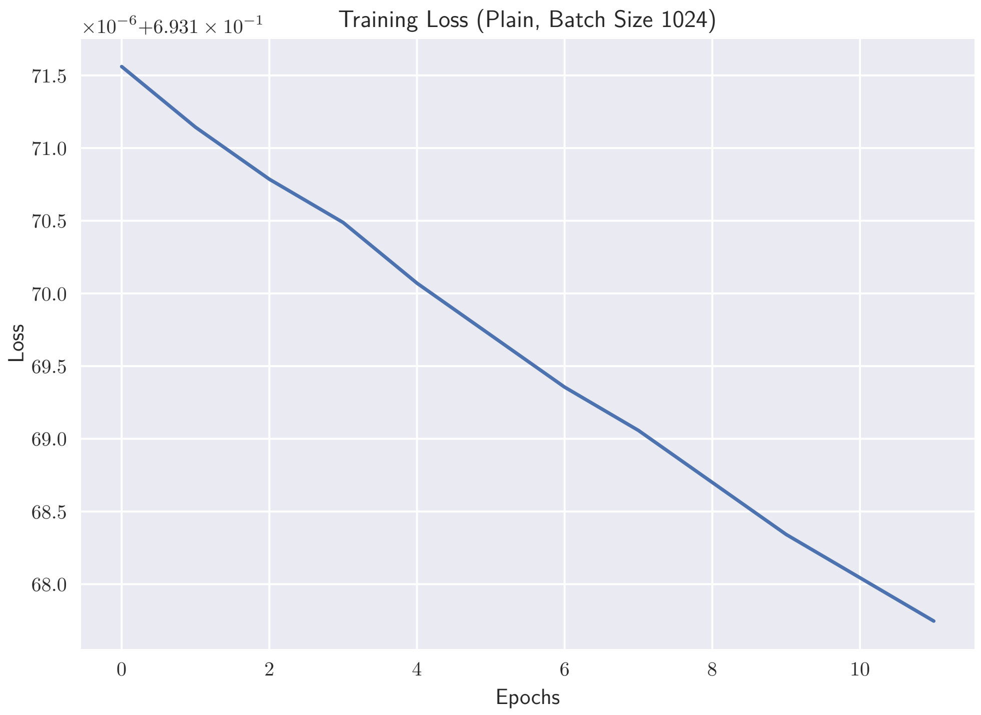


Figure 16: Loss: Plain Gradient Descent, Batch Size 1024, IMDB Sentiment Dataset

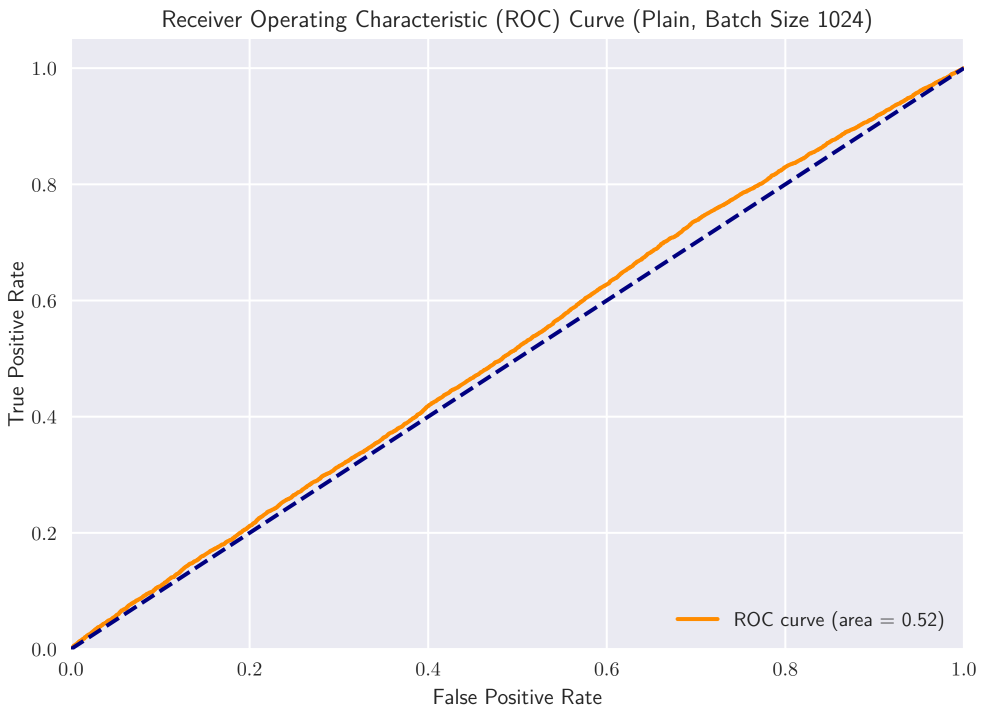


Figure 17: ROC Curve: Plain Gradient Descent, Batch Size 1024, IMDB Sentiment Dataset

Table 8: Adam, Batch Size 1024, IMDB Sentiment Dataset

Metric	Value
Loss	0.0115
Accuracy	0.9971
Precision	0.9974
Recall	0.9969
Loss Standard Deviation	0.1802
Accuracy Standard Deviation	0.1034
Precision Standard Deviation	0.0819
Recall Standard Deviation	0.0492

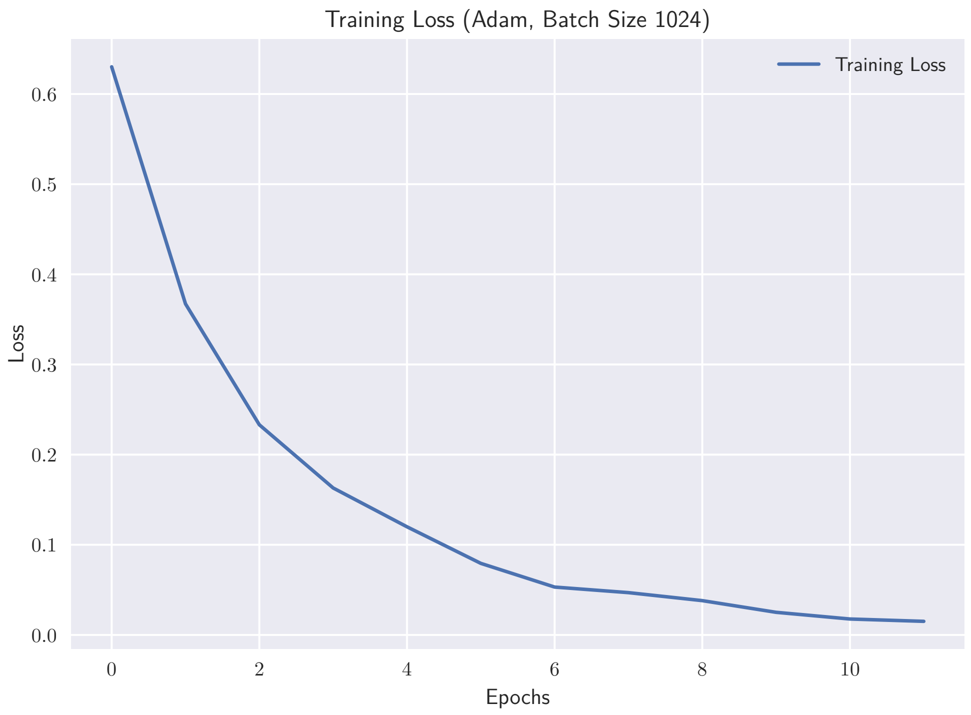


Figure 18: Loss: Adam, Batch Size 1024, IMDB Sentiment Dataset

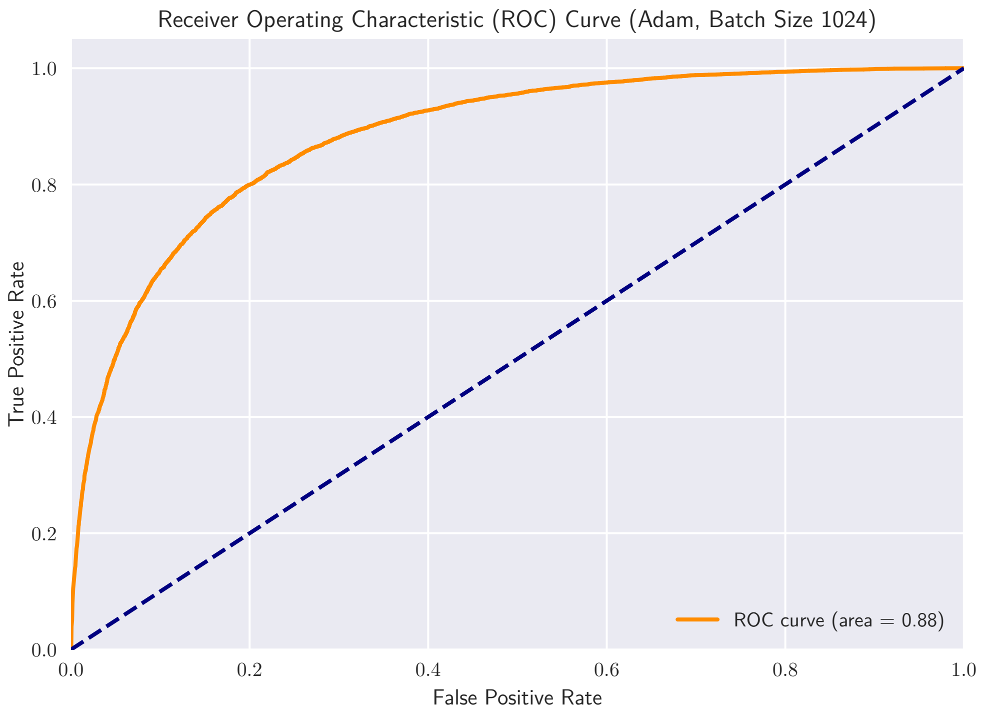


Figure 19: ROC Curve: Adam, Batch Size 1024, IMDB Sentiment Dataset

6 Discussion

Overall, Adam and RMSprop reliably bring the cost orders of magnitude lower than the standard method, especially when training with mini-batches. Furthermore, Momentum did not provide a dramatic improvement over the standard method, even performing worse on some occasions.

The normalization process in RMSprop and Adam seems to generalize more effectively than the acceleration process in Momentum. In a sense, the normalization gives the iteration process a better idea of where not to go, while Momentum takes a more aggressive approach in prioritizing acceleration in a given direction.

In addition, there were a few instances where RMSprop outperformed the Adam method in minimizing the cost function. This might be a result of the first moment of the gradient dominating the adaptive learning rate in Adam, leading the estimate to overshoot.

Finally, our experiments demonstrate the critical role of the batch size on the convergence rate, with our tests showing that Mini-Batch can help improve a linear rate of convergence (with respect to the number of epochs) to one that resembles a quadratic rate. However, while a smaller batch size can lead to a faster convergence rate, with respect to the number of epochs, it can also increase the duration of an epoch due to the numerous computations that take place in a single pass through the dataset. Therefore, in selecting the appropriate batch size, it is crucial to strike a balance between the theoretically enhanced rate of convergence and its typically inverse relationship with the training duration.

7 Conclusion

In conclusion, our exploration into the impact of batch size on the convergence behavior of optimization algorithms underlines its significant role in the performance of a model. This study has highlighted that Momentum is a foundational technique that paved the way for more sophisticated methods by providing a crucial understanding of acceleration in convergence. However, our experiments demonstrated that while Momentum enhances speed, it lacks the stability and adaptability provided by RMSprop and Adam. Furthermore, Adam, synthesizing the strengths of both Momentum and RMSprop, presents a robust approach by optimizing both convergence speed and stability. Our breakdown and analysis of optimization algorithms not only clarifies their individual attributes but also showcases their cumulative evolution. Future research should continue to refine these techniques, potentially exploring hybrid or adaptive methods that can dynamically adjust parameters according to real-time feedback from the training process. The ongoing development of optimization algorithms will likely continue to be a cornerstone in the advancement of deep learning methods.

References

[1] Peyré, G. *Advanced Gradient Descent Methods.* Retrieved from `https://www.ceremade.dauphine.fr/~waldspurger/tds/22_23_s1/advanced_gradient_descent.pdf` [Accessed: April 10, 2024].

[2] Deisenroth, M. P. (2021). *ECE 6270 / CS 6476: Variational Methods for Visual Recognition.* Retrieved from `https://mdav.ece.gatech.edu/ece-6270-spring2021/notes/06-grad-desc-analysis.pdf` [Accessed: April 10, 2024].

[3] Bubeck, S. (2015). *Convex Optimization: Algorithms and Complexity.* Retrieved from `http://sbubeck.com/Bubeck15.pdf` [Accessed: April 10, 2024].

[4] Hinton, G. *Unsupervised Feature Learning and Deep Learning: Lecture 6.* Retrieved from `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf` [Accessed: April 10, 2024].

[5] Keras. (n.d.). *Keras API Documentation: Metrics.* Retrieved from `https://keras.io/api/metrics/` [Accessed: April 10, 2024].

[6] CoHere. (n.d.). *A Comprehensive Guide to Classification Evaluation Metrics.* Retrieved from `https://cohere.com/blog/classification-eval-metrics` [Accessed: April 17, 2024].

[7] Wikipedia contributors. (n.d.). *Receiver operating characteristic.* Retrieved from `https://en.wikipedia.org/wiki/Receiver_operating_characteristic` [Accessed: April 23, 2024].

[8] Wikipedia contributors. (n.d.). *Exponential smoothing.* Retrieved from `https://en.wikipedia.org/wiki/Exponential_smoothing` [Accessed: April 10, 2024].

[9] Zhou, X. (2017, April 29). *Strong convexity.* Retrieved from `https://xingyuzhou.org/blog/notes/strong-convexity` [Accessed: April 8, 2024].

[10] Ahmadi, A. A. (2016). *Lecture 7: Conjugate Gradient Descent.* Princeton University. Retrieved from `https://www.princeton.edu/~aaa/Public/Teaching/ORF523/S16/ORF523_S16_Lec7_gh.pdf` [Accessed: April 17, 2024].