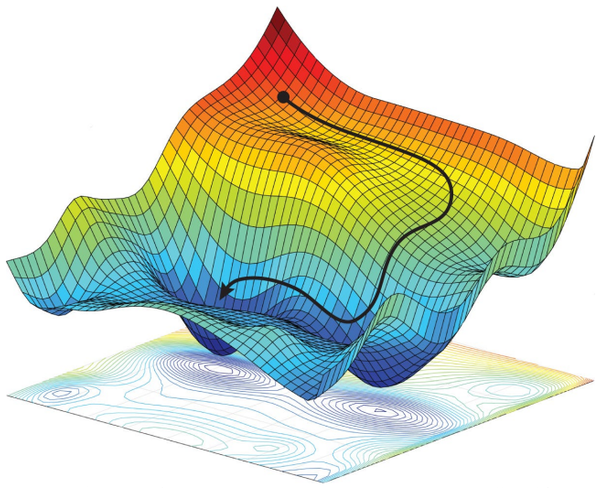
# Mathematical Exploration of Deep Learning Optimizers

Ishita Jain, Sudharsan Gopalakrishnan, Akshay Seetharam, Aabhas Senapati

December 2023



https://towardsdatascience.com/on-optimization-of-deep-neural-networks-21de9e83e1?gi=0a4a4c9b44de

# Contents

# 1 Introduction

Deep Learning is a field of Machine Learning that uses a vast amount of data for prediction. It is used in Natural Language Processing, Computer Vision, and Recommender Systems. To train on large amounts of data, Deep Learning models need fast and powerful optimizers. This paper will explore popular Deep Learning optimizers from a mathematical standpoint as well as their advantages and disadvantages.

# 2 Stochastic Gradient Descent

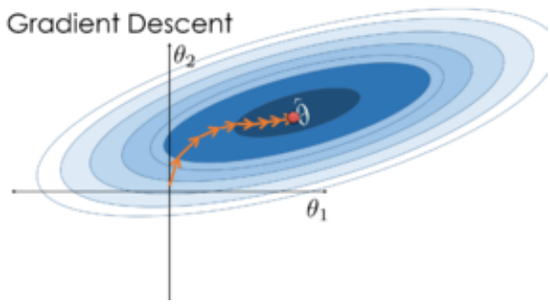



Figure 1: Visualization of how Gradient Descent finds minima

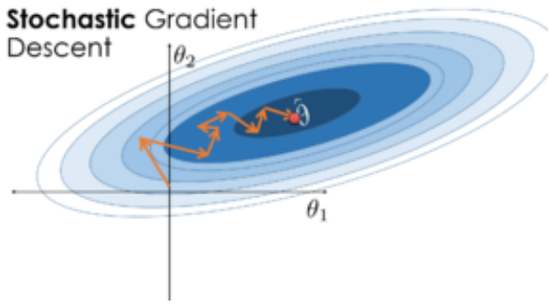



Figure 2: Visualization of how Stochastic Gradient Descent finds minima

## 2.1 What Is Gradient Descent?

To understand Stochastic Gradient Descent, let's first discuss Gradient Descent. Gradient Descent is the simplest optimizer on which almost all other optimizers are built.

In a neural network, the core structure for performing Deep Learning, there are various layers of neurons that pass information from layer to layer. Each neuron in a layer connects to one neuron in the next layer. These connections are represented with weights that determine how important each neuron is to the overall prediction; positive weights allow for successive neurons to be more important, while negative weights prompt successive neurons to be less important. The combination of the weights as a whole determines how well a neural network can model a particular data set. So, Deep Learning optimizers essentially try to find the most optimal combination of weights, that is a linear weighted sum given as follows:

$$\hat{y} = \sum_{i=1}^{n} w_i x_i + b = w_1 x_1 + w_2 x_2 + ... + w_n x_n + b$$

Here, $w_i$ represents each weight in a particular layer of $n$ neurons, $x_i$ represents the value (also known as an activation, a value between 0 and 1 that quantifies the importance of a neuron) of each corresponding neuron in the layer, and $b$ is an inherent bias value that serves as the intercept of the optimized linear model to be found. These weights, plus the bias value, compose the training parameters of a neural network. This is not to be confused with hyperparameters, which define the nature of the training process (i.e. learning rate and choice of optimizer).

Gradient Descent works by modifying each parameter according to some loss function $J$. This loss function describes the error in the Deep Learning model's prediction, or the difference between the predicted and actual values. $J$ is calculated by finding the difference between the actual values and the predicted weighted sums of all the parameters. There are many popular loss functions, the most popular of which being Mean Squared Error (will be discussed more later). Gradient Descent is performed in the following series of steps:

1. Initially randomize the weights of the model.
2. Calculate $J$ for each data point, sum all the losses, and then take an average of them.
3. Compute gradient of the loss function, $\nabla J$, with respect to each weight, $w_i$.

4. Update each weight by subtracting the corresponding gradient with respect to the weight from the previous weight value.

5. Repeat the above steps until a desired threshold is met (i.e. an accuracy greater than 80%).

## 2.2 Stochastic Gradient Descent vs Gradient Descent

While Gradient Descent calculates the gradients for all data points in a data set, Stochastic Gradient Descent calculates the gradients for just a relatively small batch of the data points. As such, Stochastic Gradient Descent offers faster optimization, especially when there can exist lots of clusters or redundancies in a data set. However, this can result in noisier variations in the loss curve and might not land the overall loss at the perfect minima.

There is a common way to compromise between the two called mini-batch gradient descent or the other descent techniques discussed further in the paper. Note that each optimizer intends to update the weights of a model such that the error, or loss, between the actual and predicted values is minimized.

## 2.3 Stochastic Gradient Descent Equation

The weight updates performed using Stochastic Gradient Descent (and any simple form of Gradient Descent in general) is given by the following:

$$w_{t+1} = w_t - \gamma \nabla J$$

where $w_{t+1}$ represents the new weight, $w_t$ is the previous weight, and $\gamma \nabla J$ is the learning rate $\gamma$ multiplied by the gradient of the loss $\nabla J$.

## 2.4 Example calculation

We are going to assume we want to fit a linear model to our multi-variable data and get the optimal function.

Let's say we have a very simple neural network model (a graph of layers of nodes that pass data from one to another) like in the figure below. Each circle represents an input column, $x_i$, the $w_i$ represent the weight attached to each of the different columns and the output circle represents the final value predicted by our model. b represents the "y-intercept" which isn't multiplied to any input.

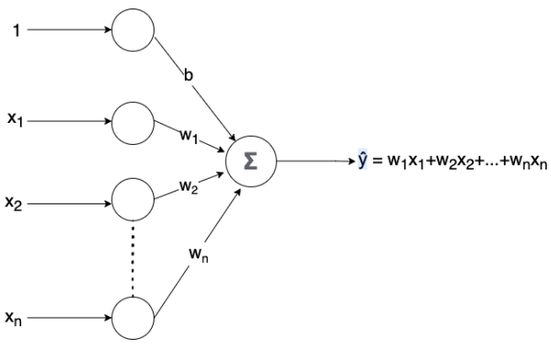



Figure 3: A very simple neural network, one layer deep

$$\hat{y} = w_1 x_1 + w_2 x_2 + ... + w_n x_n + b$$

Stochastic Gradient Descent is a way to find the weights of this function, in an optimal way. Remember that the gradient finds where the direction of steepest ascent is on a hill. Thus, if we want to maximize a function, we go where the gradient points us. But, if we want to minimize a function -namely the error of the function y, we want to go in the negative gradient direction.

Lets just say that our cost function - or the function that tells us how bad our current model is performing in predicting the data - is the Mean Squared Error (MSE) given by the below equation.

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=0}^{n}(y_i - (w_1 x_1 + w_2 x_2 + ... + w_n x_n + b)_i)^2$$

where y is the actual value and $\hat{y}$ is the predicted value and n is the total number of data points.

And, for the purposes of a sample calculation, our data table is below:

| Height | Weight | BMI |
|---|---|---|
| 0.5 | 1.4 | 4 |
| 2.3 | 1.9 | 4.1 |
| 2.9 | 3.2 | 3 |

(An important note to mention is that we wouldn't use such a small dataset on a neural network model)

We start by randomly generating a plane that we hope predicts y well. Let's say that we think BMI (y) = Height + Weight + 1 where $w_1 = 1, w_2 = 1$, and $b = 1$. We then need to find out how to minimize the cost function - which is helpful to do with calculus. We can take the partial derivative of the cost function with respect to each of the variables in it.

Thus, our partial derivatives of the cost function would be, where n = 3:

$$\frac{\partial costf}{\partial w_1} = -2x_1/3(y - w_1x_1 - w_2x_2 - b)$$

$$\frac{\partial costf}{\partial w_2} = -2x_2/3(y - w_1x_1 - w_2x_2 - b)$$

$$\frac{\partial costf}{\partial b} = -1/3(y - w_1x_1 - w_2x_2 - b)$$

Normally, Stochastic Gradient Descent would update learning rate for you - by stating with a large learning rate and making it smaller with each iteration but in our simple example, we can set the learning rate to 0.1.

If we calculate the values for the 3 partial derivatives by plugging in the point (0.5,1.4,4), we would get -0.37, - 1.03, 0.73. Next we would multiply each with the learning rate and adjust the new weights to be that value:

$$w_1 = x_1 - \frac{(\partial costf}{\partial w_1} * a)$$

$$w_2 = x_2 - \frac{(\partial costf}{\partial w_2} * a)$$

$$b = x_3 - \frac{(\partial costf}{\partial b} * a)$$

where a is the learning rate.

Finally, our new weights would be $w_1 = 1 + .036, w_2 = 1 + .103, b = 1 - 0.073$.

The new parameters give us the equation of a new function we can use to predict the data more effectively. We would repeat these steps, but by picking a new point. Once the algorithm reaches a certain threshold - such as not improving the loss function by a significant margin or iterating a certain amount of times.

# 3 Stochastic Gradient Descent With Momentum

As we work with Stochastic Gradient Descent, we notice that Stochastic Gradient Descent is especially slow in navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another. In such instances, Stochastic Gradient Descent oscillates across the slopes of the ravine with very little progress toward the local optimum and tends to get stuck in flat spots that have no gradient and aren't the actual minima.

Stochastic Gradient Descent, along with momentum, is an algorithm that builds inertia and overcomes the oscillations of Stochastic Gradient Descent and coasts along the flat spots that aren't local minima. It does this by using the history of the optimization function when updating the weights. This includes the addition of a new constant, $\beta$, like learning rate, that controls how much of the historical loss gradients the optimization function wants to use. Using momentum simply changes the update function defined in Stochastic Gradient Descent:

$$\nabla w_{1,t} = \frac{\partial costf}{\partial w_{1,t}} * (1 - \beta) + (\beta * \nabla w_{1,t-1})$$

$$\nabla w_{2,t} = \frac{\partial costf}{\partial w_{2,t}} * (1 - \beta) + (\beta * \nabla w_{2,t-1})$$

$$\vdots$$

$$\nabla b_t = \frac{\partial costf}{\partial b} * (1 - \beta) + (\beta * \nabla b_{t-1})$$

where $w_{i,t-1}$ is the previous weight function. And then, we would simply state

$$w_{i,t} = w_{i,t-1} - \nabla w_{i,t}$$

Think of the weight update $\nabla w_{i,t}$ as a kind of velocity that propels the optimizer towards the minimum error. This would mean that very large momentum will mean that the optimization function is affected very strongly by the past data while small momentum will mean that history isn't as important.

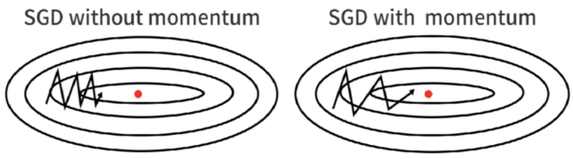



Figure 4: The differences in Stochastic Gradient Descent with and without momentum

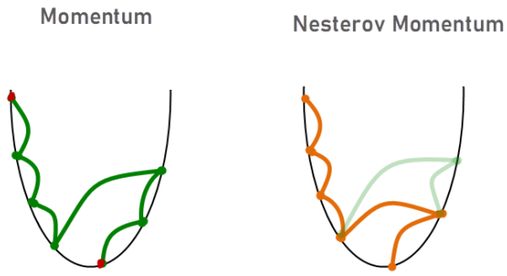



Figure 5: Nesterov based momentum versus non-nesterov based momentum

# 4 Nesterov

Once we understand how to modify gradient descent with momentum, Nesterov momentum is not actually that much of a step up. The fundamental question that motivates Nesterov momentum is, "Can we extract more information out of past steps?" If momentum looks at only the *size* of the past step, Nesterov momentum gleans more information by looking at the *rate of change* of step size. We can understand why by using the other common name of this method: Nesterov accelerated gradient descent.

We are essentially making use of the time derivative of momentum (with apologies to physicists), which is acceleration. The math adds one small complication—we just add one partial derivative term, and now we have a method of gradient descent that takes into account how quickly the curvature of the landscape changes. Nesterov momentum, for this reason, is well suited for applications where the cost function changes shape and direction quickly, and it can avoid falling into local minima because of its sensitivity to curvature. Of course, the computational and space costs are now even higher, but we can offset this by introducing more stochasticity with the understanding that Nesterov momentum might lead us a bit more astray.

Nesterov momentum is given by the same equations as states for SGD with momentum

$$\nabla w_{1,t} = \frac{\partial costf}{\partial w_{1,t}} * a + (\beta * \nabla w_{1,t-1})$$

$$\nabla w_{2,t} = \frac{\partial costf}{\partial w_{2,t}} * a + (\beta * \nabla w_{2,t-1})$$

$$\vdots$$

$$\nabla b_t = \frac{\partial costf}{\partial b} * a + (\beta * \nabla b_{t-1})$$

However, instead of just feeding the previous weight values $w_{i,t-1}$ into $\frac{\partial costf}{\partial w_i}$, we instead feed $w_{i,t-1} + (\beta * \nabla w_{2,t-1})$ in order to control the optimizer's acceleration to the minimum error. Specifically, this controls changes in the optimizer's velocity at every time step.

The weight updates, however, remain the same as before.

$$w_{i,t} = w_{i,t-1} - \nabla w_{i,t}$$

# 5 Adam

Before we cover the Adam optimizer, it is critical to discuss two other special optimizers: AdaGrad and RMSProp.

These two special optimizers are known as adaptive learning optimizers. With the optimizers we discussed above, the learning rate stays constant. However, AdaGrad and RMSProp modify the learning rate to achieve an optimal path to the minimum loss.

## 5.1 AdaGrad

Here, each weight $w_i$ and bias are updated as follows:

$$w_{i,t} = w_{i,t-1} - \frac{a}{\sqrt{\sum_{j=1}^{t} (\frac{\partial costf}{\partial w_{i,j}})^2 + \epsilon}} * (\frac{\partial costf}{\partial w_{i,t}})$$

In the denominator of the second term in the right-hand-side, we store the square root of the sum of squares of past gradients, thus lowering the overall learning rate (the entire fraction multiplied to the gradient of the current loss $\frac{\partial costf}{\partial w_{i,t}}$). The extra $\epsilon$ is simply a constant meant to prevent division by zero.

Ideally, for each new squared gradient $(\frac{\partial costf}{\partial w_{i,j}})^2$ added to the sum of squares in the denominator, if $\frac{\partial costf}{\partial w_{i,j}}$ is small, then a bigger overall learning rate is necessary. Similarly, if $\frac{\partial costf}{\partial w_{i,j}}$ is large, then a smaller overall learning rate is necessary.

To make the equations less overwhelming, let's write velocity equations as we did for the optimizers above.

$$\nabla w_{1,t} = (\frac{\partial costf}{\partial w_{1,t}})^2 + (\nabla w_{1,t-1})$$

$$\nabla w_{2,t} = (\frac{\partial costf}{\partial w_{2,t}})^2 + (\nabla w_{2,t-1})$$

$$\vdots$$

$$\nabla b_t = (\frac{\partial costf}{\partial b})^2 + (\nabla b_{t-1})$$

So, the weight update equation becomes the following:

$$w_{i,t} = w_{i,t-1} - \frac{a}{\sqrt{\nabla w_{i,t}} + \epsilon} * (\frac{\partial costf}{\partial w_{i,t}})$$

However, one issue with this optimizer is that the overall learning rate always decreases. Eventually, the learning rate might be so small that hardly any changes are made to the weights. So, we need to create a way to control how much the gradient of the current loss influences the weight updates and how much the previous "velocity" influences the new "velocity". We will do this with RMSProp.

## 5.2 RMSProp

The only thing that RMSProp changes is the velocity updates. Here, the "velocity" updates are given by the following:

$$\nabla w_{1,t} = (\frac{\partial costf}{\partial w_{1,t}})^2 * (1-\beta) + (\beta \nabla w_{1,t-1})$$

$$\nabla w_{2,t} = (\frac{\partial costf}{\partial w_{2,t}})^2 * (1-\beta) + (\beta \nabla w_{2,t-1})$$

$$\vdots$$

$$\nabla b_t = (\frac{\partial costf}{\partial b})^2 * (1-\beta) + (\beta \nabla b_{t-1})$$

Here, $\beta$ is another hyperparameter that is used to control the influence of the previous "velocity" and the gradient of the current loss on the new "velocity". More specifically, RMSProp uses an exponentially decaying average, as seen by the use of $\beta$ above, instead of just accumulating all previous squared gradients. $\beta$ is typically set to 0.9.

## 5.3 AdaDelta

AdaDelta is an extension of RMSProp. For the weight update equation, there is an extra factor in the numerator of the overall learning rate multiplied to the gradient of the cost. This factor is very similar to the denominator

$$\nabla w_{1,t} = (\frac{\partial costf}{\partial w_{1,t}})^2 * (1-\beta) + (\beta \nabla w_{1,t-1})$$

$$\nabla w_{2,t} = (\frac{\partial costf}{\partial w_{2,t}})^2 * (1-\beta) + (\beta \nabla w_{2,t-1})$$

$$\vdots$$

$$\nabla b_t = (\frac{\partial costf}{\partial b})^2 * (1-\beta) + (\beta \nabla b_{t-1})$$

### 5.4 Adam, the best of all worlds

Adam is now a very simple idea because it simply is a combination of Stochastic Gradient Descent with momentum and RMSProp.

Let's define our velocity updates for Stochastic Gradient Descent with momentum as follows:

$$\nabla w_{1,t}^1 = \frac{\partial costf}{\partial w_{1,t}} * (1 - \beta_1) + (\beta_1 * \nabla w_{1,t-1})$$

$$\nabla w_{2,t}^1 = \frac{\partial costf}{\partial w_{2,t}} * (1 - \beta_1) + (\beta_1 * \nabla w_{2,t-1})$$

$$\vdots$$

$$\nabla b_t^1 = \frac{\partial costf}{\partial b} * (1 - \beta_1) + (\beta_1 * \nabla b_{t-1})$$

And, let's define our velocity updates for RMSProp as follows:

$$\nabla w_{1,t}^2 = (\frac{\partial costf}{\partial w_{1,t}})^2 * (1 - \beta_2) + (\beta_2 \nabla w_{1,t-1})$$

$$\nabla w_{2,t}^2 = (\frac{\partial costf}{\partial w_{2,t}})^2 * (1 - \beta_2) + (\beta_2 \nabla w_{2,t-1})$$

$$\vdots$$

$$\nabla b_t^2 = (\frac{\partial costf}{\partial b})^2 * (1 - \beta_2) + (\beta_2 \nabla b_{t-1})$$

Note that the subscripts on the beta hyperparameter in both velocity equations are used to distinguish between betas of the different equations. The same applies for the superscripts on the velocity updates (left hand side).

So, in Adam, the weights are updated as follows:

$$w_{i,t} = w_{i,t-1} - \frac{a}{\sqrt{\nabla w_{i,t}^2} + \epsilon} * (\nabla w_{i,t}^1)$$

In fact, for more efficient results, we scale the velocities $\nabla w_{i,t}^1$ and $\nabla w_{i,t}^2$ up to increase the influence of the previous velocities and gradients. This is done as follows:

$$\nabla w_{i,t}^{1,corrected} = \frac{\nabla w_{i,t}^1}{1 - \beta_1}$$

$$\nabla w_{i,t}^{2,corrected} = \frac{\nabla w_{i,t}^2}{1 - \beta_2}$$

So, the weights are then updated as follows:

$$w_{i,t} = w_{i,t-1} - \frac{a}{\sqrt{\nabla w_{i,t}^{2,corrected}} + \epsilon} * (\nabla w_{i,t}^{1,corrected})$$

## 6 Testing these optimizers

The optimizers are simulated on three different types of data: image and text. The text data requires a Recurrent Neural Network and the image data requires a Convolutional Neural Network.

### 6.1 RNN

One of the real world data sets we used was a stack of IMDb reviews, each of which are sentiment classified as 0 (negative sentiment) or 1 (positive sentiment). In order to effectively train a model to perform sentiment analisys, that is properly classify the sentiment of such reviews, we decided to use an RNN. RNN models are designed to process sequential data such as text or time series data.

For the RNN, we specifically used a Bidirectional Long-Short-Term-Memory (BiLSTM) model. Unlike a standard model, this model is much more efficient by avoiding the vanishing gradient problem and allowing data to flow both backwards and forwards. As such, the model can analyze both the past and future context of a word in a text sample and more accurately predict the sentiment.

*We used Tensorflow to access the data set, model layers, and optimizers.*

Right below are two figures that display the training accuracy and loss after training the BiLSTM model on each optimizer.

As seen in Figure 7, the accuracy with the standard SGD optimizer seems to be stuck at 0.5 until after about 20 epochs when it starts to rise and then plateau. The optimizers Momentum, Nesterov, and AdaGrad observably perform much better and similar to one another. Momentum demonstrates a high

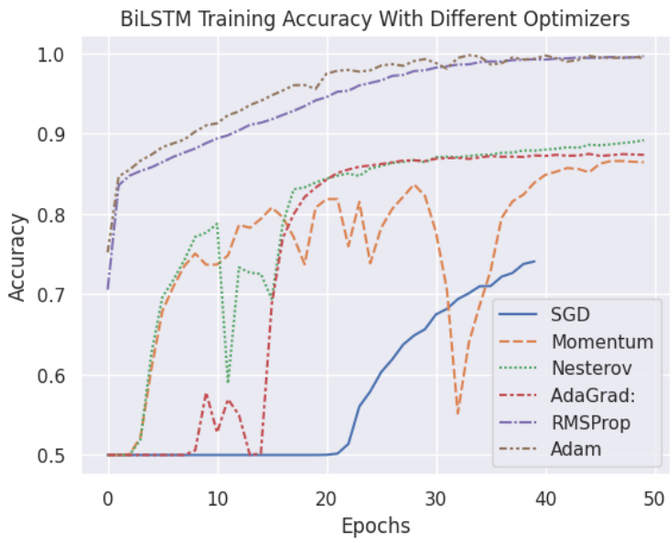



Figure 6: BiLSTM Accuracy Vs. Optimizer

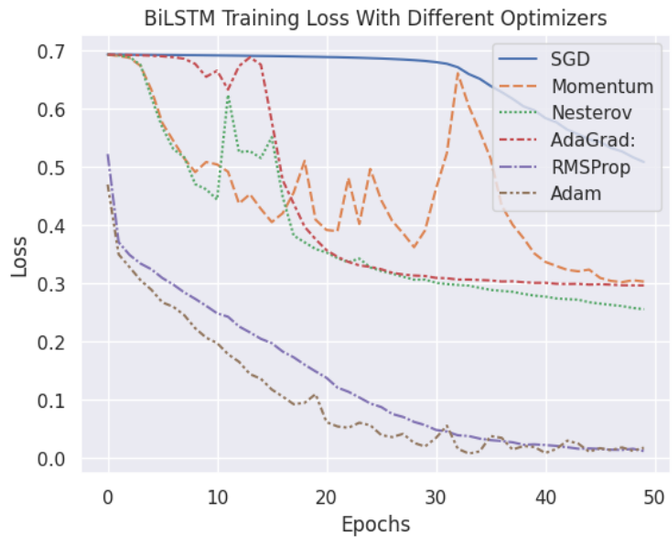



Figure 7: BiLSTM Loss Vs. Optimizer

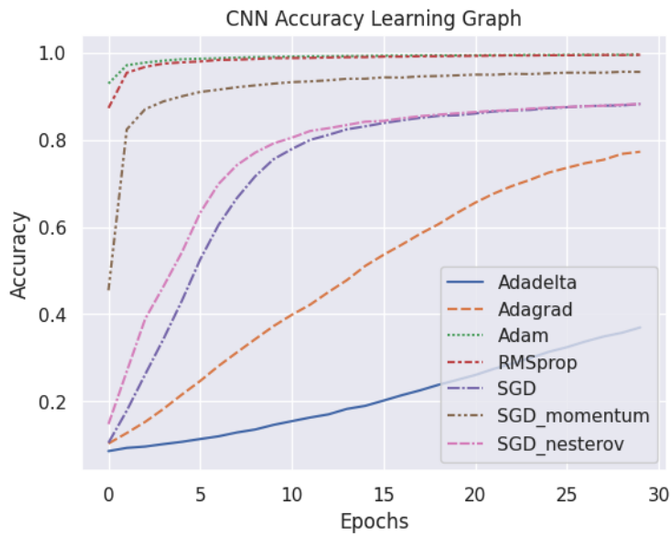
degree of fluctuation towards the end in comparison to these other two, especially Nesterov. This supports the assertion that Nesterov is used to control the velocity at which the error is minimized with an additional acceleration-like parameter. AdaGrad seems to do no better than these other two after many epochs, though it does noticibly begin its raise in accuracy much after and undergo a very steep incline. However, the sudden plateau post the incline is likely due to the sum of squared gradients becoming too large in the denominator of the velocity updates. As such, the overall learning rate becomes so small that the model's weights and bias are hardly updated. The last two, RMSProp and Adam, perform the best out of all optimizers and do not show any concievable difference among each other, other than that Adam brings the accuracy to a high enough value much faster before the accuracy starts to plateau with a slightly higher degree of fluctuation. This makes sense given that Adam is a combination of momentum and RMSProp, and Momentum may inevitably lead to fluctuation with higher velocities.

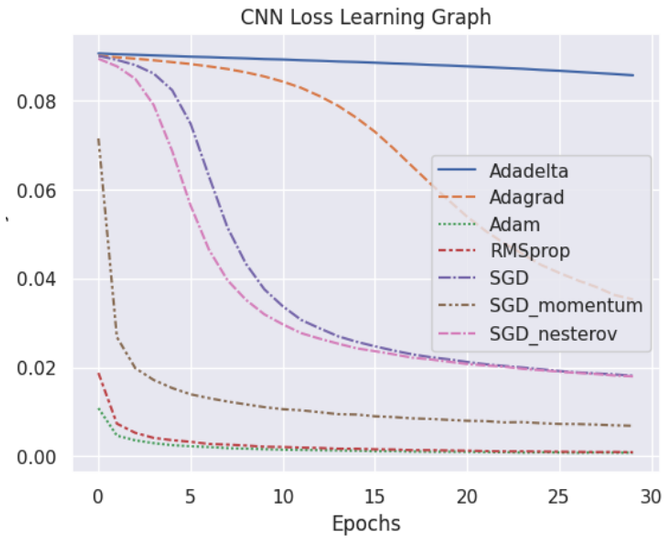
As per Figure 8, the loss for SGD initially reduces slowly until after about 30 epochs in which the loss starts reduces quicker to about 0.5. Momentum, Nesterov, and AdaGrad performed much better than SGD, with Momentum experiencing fluctuation as discussed above. And additionally as expected, RMSProp and Adam perform the best, with Adam's loss reducing the quickest with a slightly higher degree of fluctuation.

## 6.2 CNN

Convolutional Neural Networks are Neural Networks that are commonly used in image classification. An intuitive approach to image classification is to convert each pixel to a numerical value that would be the input to a neural network. However, this solution is impractical for big images or images that have more than one color. Additionally, a small shift in the image - such as to the right or left - would make the above neural network fail. To solve this issue, CNNs utilize the correlations found in images to create a "filter". After dotting the filter pixel values with the input image, the network reduces the pixel into a column vector that is then fed into a neural network to classify the image.

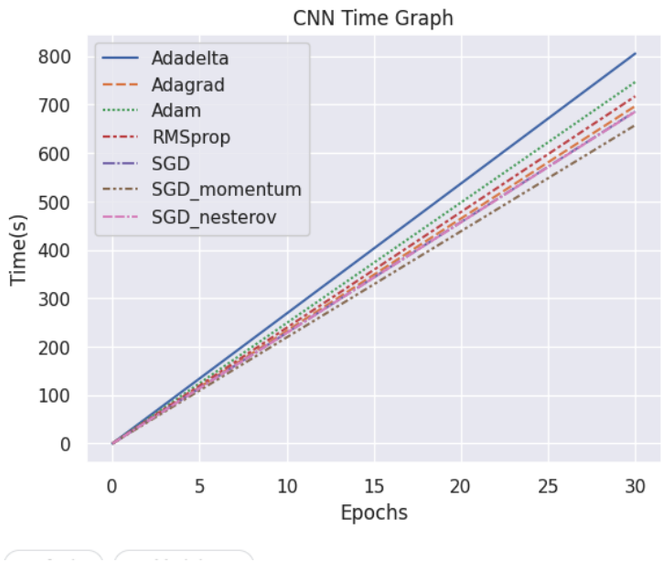
To test the optimizers on CNNs, we applied a CNN on the Modified National Institute of Standards and Technology database which has handwritten digits. The goal of the CNN is to classify these images into numerical text.




In terms of both accuracy and loss, Adam, RMSprop, and SGD momentum performed the best. Adam, predictably, performed well because it was able to adapt the learning rates and the other parameters to quickly achieve the optimal theta parameters.

RMSprop performed well for CNNs. A reason could be because RMSprop updates a velocity component as well that helps the model to look past local minimums which is helpful for CNNs because the image is built by convolutions and then scaled which could represent some inconsistencies in the prediction step.




Predictably, Stochastic Gradient Descent and its variations took the least amount of time while Adadelta, Adam, and RMSprop took the most.

# 7 Advantages and Disadvantages of these optimizers

## 7.1 Gradient Descent

Let's start with Gradient Descent, a groundbreaking algorithm that shook the world of shallow Machine Learning when it came out. However, when used in Deep Learning, the incredibly large size of the dataset made it very computationally expensive to run through every single possible iteration of the dataset. Instead, Stochastic Gradient Descent was used where you only used a single point to evaluate how well your prediction function is performing. This created an incredible amount of variance: if a point that was almost an outlier was used, the prediction function would become incorrect at prediction. When used in incredibly large datasets, the law of Large Numbers does indeed hold and helps the function be an accurate prediction of data but with the cost that there was a lot of variance in the process of getting there and it wouldn't be the most optimal function.

## 7.2 Momentum

To combat the problem above, Momentum was introduced where history of the weight function was used to minimize the effect of a single almost-outlier point. Additionally, it helped in faster calculations as the step size wasn't a constant amount no matter how far the prediction was from the optimal prediction function. Instead, the steps the weights took was adjusted to include how far the prediction is from the optimal value. However this also means that the optimal prediction function might be passed over and might also not result in extremely accurate predictions.

## 7.3 Adagrad, RMSProp, and Adam

The AdaGrad and RMSProp optimizers, unlike the ones above, are adaptative-learning optimizers, meaning that they update the learning rate throughout the learning process. AdaGrad ensures an efficient traversal of the optimization landscape by reducing the learning rate at each time step by accumulating previous gradients. However, the problem here is that the accumulation of many gradients overtime can prompt the overall learning rate of the AdaGrad optimizer to become very small. The gradients of the loss would thus have little effect, and the weights and bias would hardly be updated. So, RMSProp adds a slight nuance to AdaGrad's weight update equations such that the learning rate is allowed to both increase and decrease as necessary, allowing for an even more efficient travesal of the optimization landscape.

Adam simply combines the optimization methods for Momentum and RMSProp, making optimization extra efficient (accelerated optimization with adaptive learning rates).

# References

[1] Figure 1: "Stochastic Gradient Descent." Stochastic Gradient Descent - Cornell University Computational Optimization Open Textbook - Optimization Wiki, optimization.cbe.cornell.edu/index.php?title=Stochasticgradientdescent. Accessed 4 Dec. 2023.

[2] Figure 2: "Stochastic Gradient Descent." Stochastic Gradient Descent - Cornell University Computational Optimization Open Textbook - Optimization Wiki, optimization.cbe.cornell.edu/index.php?title=Stochasticgradientdescent. Accessed 4 Dec. 2023.

[3] Figure 3: Kershawani, Aashita. GitHub, github.com/AashitaK/A-Hands-On-Workshop-In-Machine-Learning-Fall-2023/blob/main/Session

[4] Figure 4: "Papers with Code - Stochastic Gradient Descent with Momentum Explained." Explained — Papers With Code, paperswithcode.com/method/Stochastic Gradient Descent-with-momentum. Accessed 4 Dec. 2023.

[5] Figure 5: Ponraj, Asha. "Optimizers in Neural Network " Devskrol." DevSkrol, 23 Jan. 2022, devskrol.com/2020/09/05/optimizer-in-neural-network/.

[6] Brownlee, Jason. "Gentle Introduction to the Adam Optimization Algorithm for Deep Learning - MachineLearningMastery.com." Machine Learning Mastery, 13 January 2021, https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. Accessed 5 December 2023.

[7] Brownlee, Jason. "Gradient Descent with Nesterov Momentum From Scratch." Machine Learning Mastery, 12 Oct. 2021. machinelearningmastery.com/gradient-descent-with-nesterov-momentum-from-scratch/

[8] 3blue1brown. "Gradient Descent, How Neural Networks Learn: Chapter 2, Deep Learning." YouTube, 16 Oct. 2017

[9] Oppermann, Artem. "Optimization in Deep Learning: AdaGrad, RMSProp, ADAM." KI Tutorials, 20 October 2021, https://artemoppermann.com/optimization-in-deep-learning-adagrad-rmsprop-adam/. Accessed 4 December 2023.