Advanced Deep Learning Project

Customer Churn Prediction with State-of-the-art optimizers

Table of Contents

List of Figures

Introduction to Optimizers

Optimizer algorithms are optimization methods that help increase the performance of a deep learning model. These optimization techniques or optimizers have a significant impact on the accuracy and speed of deep learning model training. An optimizer is a function or algorithm that modifies the neural network's parameters such as weights and learning rates.

As a result, it helps in lowering overall loss while improving accuracy. Choosing the proper weights for the model is a challenging task because a deep learning model typically has millions of parameters. It emphasizes the importance of selecting an appropriate optimization algorithm for the application. As a result, understanding these machine learning techniques is essential for data scientists before diving further into the deep learning concepts.

To alter the weights and learning rate, different optimizers in the machine learning model can be used. The optimum optimizer, however, is determined by the application. One bad idea that comes to mind as a beginner is that we try all the choices and choose the one that produces the best results. This may seem fine at first, but when working with hundreds of gigabytes of data, even a single epoch can take a long time. So choosing an algorithm at random is no less than gambling with time.

To alter the weights and learning rate, different optimizers in the machine learning model can be used. The optimum optimizer, however, is determined by the application. One bad idea that comes to mind as a beginner is that we try all the choices and choose the one that produces the best results. This may seem OK at first, but when working with hundreds of gigabytes of data, even a single epoch can take a long time. So choosing an algorithm at random is no less than gambling with time.

In the present study, deep-learning optimizers like Adam, RMSProp and Adagrad, AdaDelta, are reviewed. This study will compare these optimizers and the procedure these optimizers are based upon.

Adam (ADAptive Moment Estimation)

Adam optimizer derives its name from adaptive moment estimation. This optimization approach is a stochastic gradient descent extension that updates network weights during training. Unlike SGD training, which maintains a single learning rate, Adam optimizer updates the learning rate for each network weight individually. The Adam optimization algorithm's researchers are aware of the advantages of the AdaGrad and RMSProp algorithms, which are extensions of the stochastic gradient descent techniques. As a result, the Adam optimizers inherit the characteristics of both the Adagrad and RMS prop algorithms. Adam, unlike RMS Prop, adapts learning rates based on the second moment of the gradients rather than just the first moment. We define the uncentered variance as the gradients' second moment (we do not subtract the mean).

The adam optimizer has various advantages that make it popular. It has been adopted as a benchmark for deep learning papers and is suggested as the default optimization algorithm. Furthermore, the algorithm is simple to implement, has a faster running time, utilizes less memory, and requires less tuning than any other optimization algorithm.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\left[\frac{\delta L}{\delta w_t}\right] \quad v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\delta L}{\delta w_t}\right]^2$$

The given formula represents the working of Adam optimizer. Here $\beta 1$ and $\beta 2$ represents the decay rate of the average of the gradients. (Kingma, 2014)

Adam Configuration Parameters

- alpha. Also referred to as the learning rate or step size. The proportion that weights are updated (e.g. 0.001). Larger values (e.g. 0.3) results in faster initial learning before the rate is updated. Smaller values (e.g. 1.0E-5) slow learning right down during training
- beta1. The exponential decay rate for the first moment estimates (e.g. 0.9).
- beta2. The exponential decay rate for the second-moment estimates (e.g. 0.999). This value should be set close to 1.0 on problems with a sparse gradient (e.g. NLP and computer vision problems).
- epsilon. Is a very small number to prevent any division by zero in the implementation (e.g. 10E-8).

The Adam optimizer prioritizes faster computing time, whereas algorithms such as stochastic gradient descent focuses on data points. As a result, algorithms like SGD generalize data more effectively at the expense of slow calculation performance.

The optimizer modifies the model's weights during training based on the computed gradients of the loss function with respect to the weights. It changes the learning rate for each weight individually, taking into account the gradient's first and second moments. This adaptive learning rate assists the optimizer in more effectively navigating the loss landscape and converges to a better solution.

The use of the Adam optimizer in this code allows the model to take advantage of its adaptable learning rates, efficient handling of sparse gradients, and reduced need for manual tuning. These parameters work together to improve the model's performance and capacity to understand complicated patterns and generate correct predictions.

In summary, Adam is a stochastic gradient descent replacement optimization technique for training deep learning models. Adam combines the best characteristics of the AdaGrad and RMSProp methods to create an optimization algorithm capable of dealing with sparse gradients on noisy problems. Adam is relatively simple to configure, and the default configuration parameters work well for the majority of problems.

Pros and Cons of Adam Optimizer

Advantages of Adam Optimizer:

Adaptive learning rates: Adam adjusts the learning rate for each parameter individually, which aids in effectively navigating different sections of the parameter space and speeds up convergence.

Momentum: Adam uses momentum to accelerate learning, particularly in the presence of sparse gradients. This improves handling of complex optimization landscapes.

Effective for large datasets: Adam performs effectively on huge datasets because it makes good use of memory and processing resources.

Disadvantages of Adam Optimizer:

Increased memory consumption: Adam maintains additional state variables for each parameter, which can result in increased memory requirements when compared to simpler optimizers.

Sensitive to learning rate choice: Sensitive to learning rate selection: Adam's performance may be affected by the learning rate selected. An incorrectly calibrated learning rate can result in slower convergence or even divergence in some instances.

May not generalize well: Adam's adaptive nature may cause it to overfit the training data and struggle to generalize adequately to previously encountered samples. Regularization techniques such as dropout can help to alleviate this problem.

RMSProp (Root Mean Square Propagation)

RMSProp, root mean square propagation, is an optimization algorithm or method designed for Neural Network training. RMSProp is an unpublished algorithm first proposed in the Coursera course "Neural Network for Machine Learning" lecture six by Geoff Hinton in 2012. RMSProp lies in the realm of adaptive learning rate methods, which have been growing in popularity in recent years because it is the extension of Stochastic Gradient Descent (SGD) algorithm, momentum method, and the foundation of Adam algorithm. One of the applications of RMSProp is the stochastic technology for mini-batch gradient descent. RMSProp is famous for not being published yet being very well-known; most deep learning frameworks include the implementation of it out of the box. RMSprop is a good, fast, and very popular optimizer and its popularity is only surpassed by Adam.

There are two ways to introduce RMSprop. First, is to look at it as the adaptation of the RProp algorithm for mini-batch learning. RProp was the initial motivation for developing this algorithm. Another way is to look at its similarities with Adagrad and view RMSprop as a way to deal with its radically diminishing learning rates.

RMSprop Configuration Parameters

Rprop

Rprop algorithm that's used for full-batch optimization. Rprop tries to resolve the problem that gradients may vary widely in magnitudes. Some gradients may be tiny and others may be huge, which result in a very difficult problem — trying to find a single global learning rate for the algorithm. If we use full-batch learning we can cope with this problem by only using the sign of the gradient. With that, we can guarantee that all weight updates are of the same size. This adjustment helps a great deal with saddle points and plateaus as we take big enough steps even with tiny gradients. Note, that we can't do that just by increasing the learning rates, because steps we take with large gradients are going to be even bigger, which will result in divergence. Rprop combines the idea of only using the sign of the gradient with the idea of adapting the step size individually for each weight. So, instead of looking at the magnitude of the gradient, we'll look at the step size that's defined for that particular weight. And that step size adapts individually over time, so that we accelerate learning in the direction that we need.

Rprop to RMSProp

The RProp algorithm does not really work when we have very large datasets and need to perform mini-batch weights updates because it violates the central idea behind stochastic gradient descent, which is when we have a small enough learning rate, it averages the gradients over successive mini batches. To solve this issue, consider the weight that gets the gradient 0.1 on nine mini-batches, and the gradient of -0.9 on tenths mini-batch. What we'd like is for those gradients to roughly cancel each other out, so that they stay approximately the same. But that's not what happens with RProp. With RProp, we increment the weight 9 times and decrement only once, so the weight grows much larger. So, achieving the robustness of RPROP and the efficiency of mini batches simultaneously was the main motivation behind the rise of RMS prop.

Rprop is equivalent to using the gradient but also dividing by the size of the gradient, so we get the same magnitude no matter how big a small that particular gradient is. The problem with mini-batches is that we divide by different gradients every time, so why not force the number we divide by to be similar for adjacent mini-batches?

The central idea of RMSprop is to keep the moving average of the squared gradients for each weight. And then we divide the gradient by the square root of the mean square. Which is why it's called RMSprop (root mean square). RMSprop deals with the vanishing gradients problem by using a moving average of squared gradients to normalize the gradient. This normalization balances the step size (momentum), decreasing the step for large gradients to avoid exploding and increasing the step for small gradients to avoid vanishing.

Simply put, RMSprop uses an adaptive learning rate instead of treating the learning rate as a hyperparameter. This means that the learning rate changes over time. With math equations the update rule looks like this:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)\left(\frac{\delta C}{\delta w}\right)^2$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{E[g^2]_t}} \frac{\delta C}{\delta w}$$

Where E[g] is the moving average of squared gradients, δc/δw is is gradient of the cost function with respect to the weight, η is the learning rate and β is the moving average parameter (default value — 0.9, to make the sum of default gradient value 0.1 on nine mini-batches and -0.9 on tenths is approximate zero, and the default value η is 0.001 as per experience).

RMSprop is used to update the parameters of a neural network during training. It calculates an adaptive learning rate for each parameter by dividing the learning rate by the root mean square (RMS) of the exponentially weighted moving average (EWMA) of the squared gradients. The algorithm is defined as follows:

Initialize the learning rate (e.g., 0.001), decay rate (e.g., 0.9), and a small constant (e.g., 1e-8) for numerical stability.

For each parameter θ, compute the gradient g.

Update the EWMA of squared gradients:

S = decay_rate * S + (1 - decay_rate) * g^2

Update the parameter:

θ = θ - learning_rate * g / sqrt(S + small_constant)

Similarity with Adagrad:

Adagrad is an adaptive learning rate algorithm that looks a lot like RMSprop. Adagrad adds element-wise scaling of the gradient based on the historical sum of squares in each dimension. This means that we keep a running sum of squared gradients. And then we adapt the learning rate by dividing it by that sum.

What's this scaling do when we have a high condition number? If we have two coordinates — one that has always big gradients and one that has small gradients we'll be diving by the corresponding big or small number, so we accelerate movement among small direction, and in the direction where gradients are large, we're going to slow down as we divide by some large number.

What happens over the course of training? Steps get smaller and smaller and smaller, because we keep updating the squared grads growing over training. So, we divide by the larger number every time. In convex optimization, this makes a lot of sense, because when we approach minima, we want to slow down. In non-convex cases it's bad as we can get stuck on the saddle point. We can look at RMSprop as algorithms that address that concern a little bit.

With RMSprop we keep that estimate of squared gradients, but instead of letting that estimate continually accumulate over training, we keep a moving average of it.

Pros and Cons of RMSprop Optimizer

Advantages of RMSprop:

- Adaptive learning rate: RMSprop adapts the learning rate for each parameter individually based on the magnitude of the recent gradients. This adaptivity allows for faster convergence and improved performance by effectively scaling the learning rate according to the requirements of each parameter.
- Efficient memory usage: RMSprop only keeps track of the exponentially weighted moving average (EWMA) of the squared gradients for each parameter. This approach reduces the memory requirements compared to algorithms that store all past gradients. Consequently, RMSprop can be more memory-efficient, particularly for large-scale neural networks.
- Robustness to different learning rates: The adaptive learning rate mechanism of RMSprop makes it more resilient to selecting an optimal learning rate. This reduces the need for extensive manual tuning and allows the algorithm to handle a wider range of learning rates without sacrificing performance.

Disadvantages of RMSprop:

- Hyperparameter sensitivity: Like any optimization algorithm, RMSprop has hyperparameters that need to be set appropriately to achieve good performance. The learning rate, decay rate, and small constant can impact the convergence and overall effectiveness of RMSprop. Finding the optimal hyperparameter values can require some trial and error or careful tuning.
- Lack of momentum: RMSprop does not incorporate momentum, which is a term that accumulates past gradients to accelerate convergence. Momentum can help the optimizer navigate through flat regions or escape local minima more effectively. In scenarios where momentum plays a crucial

role, other optimizers like SGD with momentum or Adam, which combine adaptive learning rates with momentum, may be more suitable.

- Sensitivity to non-stationary gradients: RMSprop assumes that the gradients remain stationary over time. However, in situations where the gradients change significantly throughout training, RMSprop may struggle to adapt the learning rates effectively. Techniques like learning rate annealing or using different optimizers can be employed to address this issue.

The advantages and disadvantages of RMSprop are relative and context-dependent. While RMSprop has proven to be effective in many deep learning scenarios, the choice of optimizer should ultimately be based on empirical evaluation and experimentation with the specific problem and dataset at hand.

Comparison with other optimizers:

- AdaGrad: RMSprop is an improvement over AdaGrad because it addresses the diminishing learning rate issue by using an exponentially weighted moving average. RMSprop has been observed to converge faster and generalize better than AdaGrad in many cases.
- Adam: Adam is another popular optimizer that combines the benefits of RMSprop and momentum. It uses adaptive learning rates like RMSprop but also incorporates momentum to accelerate convergence. Adam is known for its robust performance and is widely used in deep learning.
- SGD with momentum: RMSprop lacks the momentum term, which is a factor that accumulates past gradients to accelerate convergence. As a result, SGD with momentum can be more effective in cases where the optimization landscape has long, narrow valleys.

Adagrad (Adaptive Gradient)

AdaGrad, also known as Adaptive Gradient, is a powerful optimization algorithm extensively employed in machine learning and deep learning. It was developed in 2011 by Duchi et al. to address the challenges posed by online learning and stochastic optimization problems. Unlike traditional optimization methods that rely on a fixed learning rate, Adagrad dynamically adjusts the learning rate for each parameter by considering their historical gradients. This adaptive approach mitigates the issues of slow convergence and unstable updates that arise from varying parameter sensitivities or data sparsity. By leveraging accumulated gradients, Adagrad tailors a personalized learning rate, enabling faster convergence for parameters with smaller updates and providing more stable updates for frequently updated parameters. This adaptiveness grants Adagrad a remarkable ability to handle sparse data and feature sets effectively.

AdaGrad Configuration Parameters

1. Accumulating Squared Gradients:
The core idea behind AdaGrad is to adaptively adjust the learning rate for each parameter based on the historical gradients. It achieves this by accumulating the squared gradients of each parameter over time. For a given parameter θ, the squared gradients are summed up as:

G += g^2,

where G is a matrix or vector that stores the accumulated squared gradients, and g represents the gradient of the parameter θ at a particular time step.

2. Individual Learning Rates:
Unlike traditional optimization algorithms that use a fixed learning rate for all parameters, AdaGrad assigns an individual learning rate to each parameter based on its historical gradients. The learning rate for parameter θ at time step t is computed as:

$\eta_t = \eta / \sqrt{G_t + \varepsilon}$,

where η is the initial learning rate, G_t represents the accumulated squared gradients up to time step t, and ε is a small constant (e.g., 10^{-8}) added for numerical stability to avoid division by zero.

The learning rate η_t is inversely proportional to the square root of the accumulated squared gradients. Consequently, parameters with large historical gradients will have smaller learning rates, allowing them to converge more slowly, while parameters with small historical gradients will have larger learning rates, enabling faster convergence.

3. Update Rule:
Once the learning rates are computed, AdaGrad performs the parameter update using the following rule:

$\theta_t = \theta_{t-1} - \eta_t * g$,

where θ_t represents the updated value of the parameter θ at time step t, θ_{t-1} is the previous value, η_t is the individual learning rate, and g is the gradient of the parameter at time step t.

By adjusting the learning rates based on the accumulated squared gradients, AdaGrad effectively adapts the optimization process to the characteristics of each parameter, facilitating faster convergence and improved performance.

Pros and Cons of Adagrad Optimizer:

- AdaGrad automatically adapts the learning rates for each parameter, reducing the need for manual tuning and providing a more efficient optimization process.
- It is particularly effective in scenarios with sparse data or uneven feature distributions, as it assigns larger learning rates to less frequent features or parameters with small gradients.
- AdaGrad can converge quickly, especially in the early stages of training when the gradients tend to be large.

However, AdaGrad has some limitations. It accumulates the squared gradients over time, which can lead to diminishing learning rates, causing slow convergence in later stages of training. To mitigate this issue, variants of AdaGrad such as RMSprop and Adam have been developed.

Datasets and model selection:

Dataset Description: Multinational Bank Customer Churn Prediction

The Multinational Bank Customer Churn dataset is chosen to apply a deep learning model with three different optimizers. The dataset is designed to facilitate the prediction of customer churn for the bank's

account holders. Customer churn refers to the phenomenon of customers discontinuing their relationship with the bank, which is a crucial concern for the bank's sustainability and growth. By analyzing various attributes of the customers, this dataset aims to develop a predictive model to identify potential churners.

The dataset contains the following information for each customer:

1. Customer ID: A unique identifier assigned to each customer.
2. Credit Score: The credit score of the customer, representing their creditworthiness.
3. Country: The country in which the customer resides.
4. Gender: The gender of the customer (e.g., Male or Female).
5. Age: The age of the customer in years.
6. Tenure: The number of years the customer has been with the bank.
7. Balance: The account balance of the customer.
8. Product Number: The number of products the customer has with the bank.
9. Credit Card: Indicates whether the customer has an active credit card (Yes or No).
10. Active Member: Indicates whether the customer is an active member of the bank (Yes or No).
11. Churn: The target variable indicating whether the customer has churned (discontinued the relationship with the bank) (Yes or No).

The dataset is intended to be used for developing deep learning models to predict customer churn. By utilizing the provided features, such as credit score, tenure, balance, and various customer characteristics, banks can gain insights into customer behavior and proactively take measures to retain customers who are likely to churn.

Model selection: ANN

Implementation of a deep learning model i.e., Artificial Neural Network (ANN) to predict customer churn.

- The create_model function creates and configures the ANN model. It initializes an instance of the Sequential class, which is a linear stack of layers.
- The first layer added to the model is a dense layer with 64 units and the ReLU activation function with input shape as 11 indicating features of the input data.
- Batch Normalization is added to the model to normalize the activations of the previous layer, helping with faster and more stable training.
- The next layer added is another Dense layer with 128 units and the ReLU activation function. This layer increases the number of units in the hidden layer, allowing for more complex representations to be learned. Another Dense layer with 64 units and the ReLU activation function is added
- Dropout layer is added in between to prevent overfitting with a dropout rate of 0.5. Dropout randomly sets a fraction of input units to 0 at each update during training.
- The final output layer is added using a Dense layer with 1 unit and the sigmoid activation function. The sigmoid activation function squashes the output between 0 and 1, representing the probability of churn.
- The model is compiled using three different optimizers and the loss function is set to 'binary_crossentropy' since this is a binary classification problem (churn or no churn).

Binary cross-entropy is a common loss function used for binary classification problems, where there are two possible classes or outcomes. It measures the dissimilarity between the predicted probabilities and the

true labels. It quantifies the difference between the predicted probabilities of belonging to the positive class (churn) and the true binary labels (0 or 1).

Mathematically, the binary cross-entropy loss function can be defined as:

binary_crossentropy(y_true, y_pred) = -(y_true * log(y_pred) + (1 - y_true) * log(1 - y_pred))

where y_true represents the true binary labels (0 or 1) and y_pred represents the predicted probabilities of belonging to the positive class (churn).

The loss function penalizes the model more when it predicts the wrong probability for a given sample. If the true label is 1 and the predicted probability is close to 0, the loss will be high. Similarly, if the true label is 0 and the predicted probability is close to 1, the loss will be high. The loss function encourages the model to adjust its parameters to minimize the discrepancy between the predicted probabilities and the true labels.

During training, the model tries to find the set of parameters that minimizes the average binary cross-entropy loss over the entire training dataset. This optimization process is performed using gradient descent-based methods like the Adam optimizer, which iteratively updates the model's parameters to minimize the loss.

We defined ANN model with input layer, two hidden layers, output layer and one dropout in our application.

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_44 (Dense)            (None, 64)                768

 batch_normalization_11 (Bat  (None, 64)               256
 chNormalization)

 dense_45 (Dense)            (None, 128)               8320

 dropout_11 (Dropout)        (None, 128)               0

 dense_46 (Dense)            (None, 64)                8256

 dense_47 (Dense)            (None, 1)                 65

=================================================================
Total params: 17,665
Trainable params: 17,537
Non-trainable params: 128
_________________________________________________________________
```

Figure 1: ANN Model Architecture

Results and Discussion

```
Test Loss:  0.35216042399406433
Test Accuracy:  0.8550000190734863
Training Time:  26.063394784927368 seconds
+------------+-----------------+-------------+
| Epoch      | Test Accuracy   | Test Loss   |
+============+=================+=============+
| Epoch 1    | 0.8135          | 0.4470      |
+------------+-----------------+-------------+
| Epoch 10   | 0.8590          | 0.3442      |
+------------+-----------------+-------------+
| Epoch 20   | 0.8589          | 0.3372      |
+------------+-----------------+-------------+
| Epoch 30   | 0.8673          | 0.3211      |
+------------+-----------------+-------------+
| Epoch 40   | 0.8694          | 0.3110      |
+------------+-----------------+-------------+
| Epoch 50   | 0.8767          | 0.3033      |
+------------+-----------------+-------------+
| Total Time |                 |             |
+------------+-----------------+-------------+
```

Figure 2: Model Performance with ADAM optimizer

```
Test Loss:  0.3503035604953766
Test Accuracy:  0.8579999804496765
Training Time:  42.629496335983276 seconds
+------------+-----------------+-------------+
| Epoch      | Test Accuracy   | Test Loss   |
+============+=================+=============+
| Epoch 1    | 0.8138          | 0.4466      |
+------------+-----------------+-------------+
| Epoch 10   | 0.8602          | 0.3417      |
+------------+-----------------+-------------+
| Epoch 20   | 0.8649          | 0.3312      |
+------------+-----------------+-------------+
| Epoch 30   | 0.8705          | 0.3195      |
+------------+-----------------+-------------+
| Epoch 40   | 0.8737          | 0.3099      |
+------------+-----------------+-------------+
| Epoch 50   | 0.8781          | 0.2998      |
+------------+-----------------+-------------+
| Total Time |                 |             |
+------------+-----------------+-------------+
```

Figure 3: Model Performance with RMSPro optimizer

```
Test Loss:  0.38822194933891296
Test Accuracy:  0.8389999866485596
Training Time:  37.47569918632507 seconds
+------------+-----------------+-------------+
| Epoch      | Test Accuracy   | Test Loss   |
+============+=================+=============+
| Epoch 1    | 0.7232          | 0.5795      |
+------------+-----------------+-------------+
| Epoch 10   | 0.8035          | 0.4553      |
+------------+-----------------+-------------+
| Epoch 20   | 0.8059          | 0.4396      |
+------------+-----------------+-------------+
| Epoch 30   | 0.8124          | 0.4310      |
+------------+-----------------+-------------+
| Epoch 40   | 0.8202          | 0.4205      |
+------------+-----------------+-------------+
| Epoch 50   | 0.8232          | 0.4097      |
+------------+-----------------+-------------+
| Total Time |                 |             |
+------------+-----------------+-------------+
```

Figure 4: Model Performance with ADAGrad optimizer

A comparison of three optimizers, namely Adam, RMSprop, and Adagrad, was conducted to evaluate their performance in a neural network model. The Adam optimizer exhibited the most favorable results, achieving a test accuracy of 0.8580 and a test loss of 0.3503. This optimizer outperformed the others in terms of accuracy and loss, indicating its superior ability to optimize the model's performance. Despite its strong performance, Adam required a longer training time of 42.63 seconds. The RMSprop optimizer demonstrated slightly lower performance with a test accuracy of 0.8570 and a test loss of 0.3557, while the Adagrad optimizer yielded comparatively lower results with a test accuracy of 0.8390 and a test loss of 0.3882. These findings highlight the significance of selecting an appropriate optimizer, as the choice can significantly impact the overall performance and efficiency of the model.

Comparing Accuracy Curve

Figure 5: Model accuracy in training

When examining the accuracy curves of the three optimizers, it is evident that Adam achieved the highest and most consistent accuracy throughout the training process, followed by RMSprop and Adagrad. Adam's accuracy curve showcased a steady upward trend, indicating continuous improvement in the model's performance over time. This suggests that the optimizer was successful in finding the optimal parameters, resulting in higher accuracy rates.

Similarly, RMSprop demonstrated a gradually increasing accuracy curve, although with slightly more fluctuations compared to Adam. While there were intermittent periods of minor fluctuations, the overall trend showed improvement in accuracy, albeit at a slightly slower pace than Adam. This indicates that RMSprop effectively optimized the model's performance, albeit with slight variations during the training process.

In the case of Adagrad, the accuracy curve exhibited a slower and less consistent growth. Although there was an overall upward trend, the rate of improvement was relatively slower compared to Adam and RMSprop. Consequently, Adagrad may be less effective in optimizing the accuracy of the model compared to the other two optimizers.

Overall, when considering the accuracy curves, Adam achieved the highest and most consistent accuracy throughout the training process. RMSprop also followed a similar trend, but Adagrad exhibited a slower and less consistent growth in accuracy, with potential limitations in further enhancing the model's performance.

Comparing Loss Curve

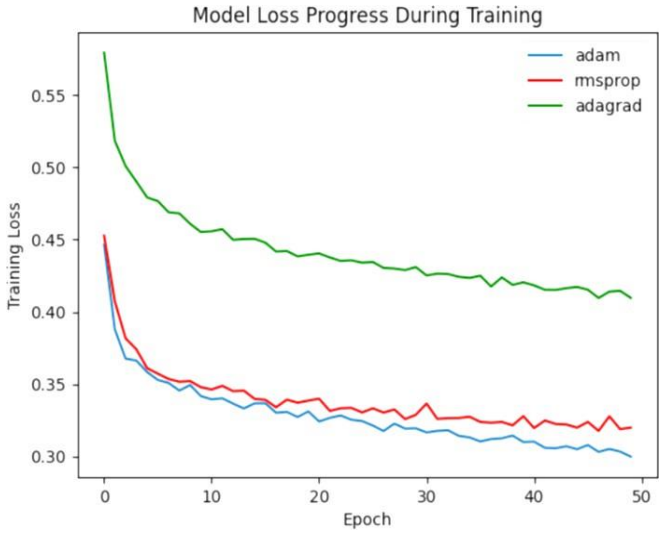

Figure 6: Model loss in training

When comparing the loss curves of the three optimizers, it is evident that Adam showcased the most favorable results, followed by RMSprop and Adagrad. In the case of Adam, the loss curve exhibited a steady and consistent downward trend throughout the training process, indicating effective convergence. This suggests that the model's loss was continuously reduced, resulting in improved performance over time.

In the case of Adam, the loss curve exhibited a steady and consistent downward trend throughout the training process, indicating effective convergence. This suggests that the model's loss was continuously reduced, resulting in improved performance over time.

RMSprop demonstrated a relatively similar trend but with slightly higher fluctuations in the loss curve compared to Adam. While the loss decreased over the course of training, there were intermittent periods of slight increases, suggesting that the optimization process might have encountered minor challenges in finding the global minimum.

On the other hand, Adagrad's loss curve exhibited a less favorable pattern. Although there was a general downward trend, the rate of reduction was relatively slower compared to Adam and RMSprop. This implies that Adagrad might be less effective in optimizing the model's performance compared to the other two optimizers. Nonetheless, the overall downward trend indicated that the model's performance was gradually improving.

Conclusion

Overall, when considering the loss curves, Adam demonstrated the most desirable pattern with a consistent and steady reduction in loss. RMSprop followed a similar trend with slightly more fluctuations, while Adagrad exhibited a slower reduction rate and a potential limitation in further minimizing the loss. The changes in the optimization procedures utilized by Adam, RMSprop, and Adagrad can be linked to the observed differences in the loss curves. Adam makes use of adaptive learning rates and momentum, which allows for faster convergence and better handling of sparse gradients. RMSprop also changes the learning rate adaptively but without momentum, resulting in some fluctuations in the loss. Adagrad, on the other hand, adjusts the learning rate depending on historical gradients, which may result in slower convergence and might result in limitations in reducing the loss.

References

Duchi, John & Hazan, Elad & Singer, Yoram. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research. 12. 2121-2159.

Kingma, D. P. (2014, December 22). Adam: A Method for Stochastic Optimization. arXiv.org. https://arxiv.org/abs/1412.6980

Understanding RMSprop — faster neural network learning | by Vitaly Bushaev. (2018, September 2). Towards Data Science. Retrieved June 23, 2023, from https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a