A Comprehensive Guide on Optimizers in Deep Learning

What Are Optimizers in Deep Learning?

In deep learning, optimizers are algorithms that adjust the model's parameters during training to minimize a loss function. They enable neural networks to learn from data by iteratively updating weights and biases. Common optimizers include Stochastic Gradient Descent (SGD), Adam, and RMSprop. Each optimizer has specific update rules, learning rates, and momentum to find optimal model parameters for improved performance.

Optimizer algorithms are optimization method that helps improve a deep learning model's performance. These optimization algorithms or optimizers widely affect the accuracy and speed training of the deep learning model. But first of all, the question arises of what an optimizer is.

While training the deep learning optimizers model, modify each epoch's weights and minimize the loss function. An optimizer is a function or an algorithm that adjusts the attributes of the neural network, such as weights and learning rates. Thus, it helps in reducing the overall loss and improving accuracy. The problem of choosing the right weights for the model is a daunting task, as a deep learning model generally consists of millions of parameters. It raises the need to choose a suitable optimization algorithm for your application. Hence understanding these machine learning

algorithms is necessary for data scientists before having a deep dive into the field.

You can use different optimizers in the machine learning model to change your weights and learning rate. However, choosing the best optimizer depends upon the application. As a beginner, one evil thought that comes to mind is that we try all the possibilities and choose the one that shows the best results. This might be fine initially, but when dealing with hundreds of gigabytes of data, even a single epoch can take considerable time. So randomly choosing an algorithm is no less than gambling with your precious time that you will realize sooner or later in your journey.

This guide will cover various deep-learning optimizers, such as Gradient Descent, Stochastic Gradient Descent, Stochastic Gradient descent with momentum, Mini-Batch Gradient Descent, Adagrad, RMSProp, AdaDelta, and Adam. By the end of the article, you can compare various optimizers and the procedure they are based upon.

Important Deep Learning Terms

Before proceeding, there are a few terms that you should be familiar with.

- Epoch – The number of times the algorithm runs on the whole training dataset.
- Sample – A single row of a dataset.
- Batch – It denotes the number of samples to be taken to for updating the model parameters.

- Learning rate – It is a parameter that provides the model a scale of how much model weights should be updated.
- Cost Function/Loss Function – A cost function is used to calculate the cost, which is the difference between the predicted value and the actual value.
- Weights/ Bias – The learnable parameters in a model that controls the signal between two neurons.

Now let's explore each optimizer.

Gradient Descent Deep Learning Optimizer

Gradient Descent can be considered the popular kid among the class of optimizers. This optimization algorithm uses calculus to modify the values consistently and to achieve the local minimum. Before moving ahead, you might have the question of what a gradient is.

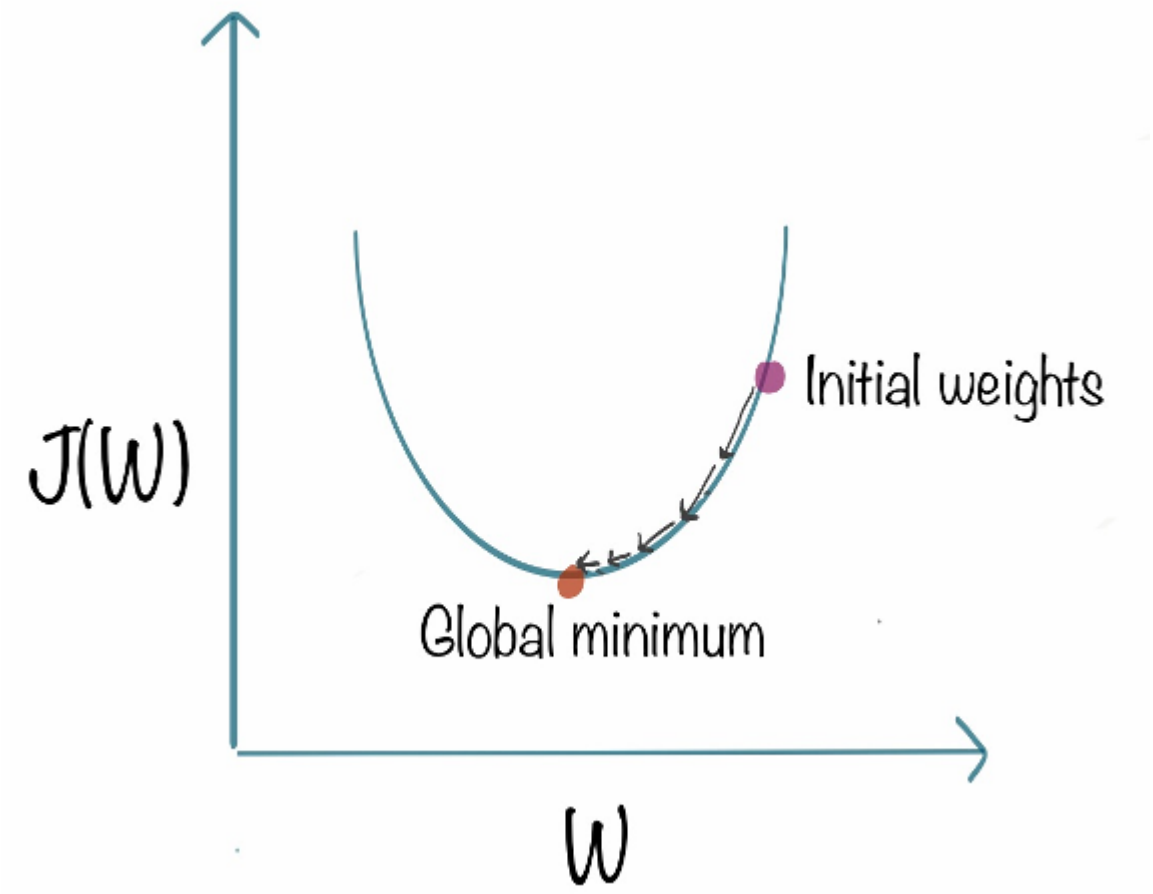
In simple terms, consider you are holding a ball resting at the top of a bowl. When you lose the ball, it goes along the steepest direction and eventually settles at the bottom of the bowl. A Gradient provides the ball in the steepest direction to reach the local minimum which is the bottom of the bowl.

$$x\_new = x - alpha * f'(x)$$

The above equation means how the gradient is calculated. Here alpha is the step size that represents how far to move against each gradient with each iteration.

Gradient descent works as follows:

- It starts with some coefficients, sees their cost, and searches for cost value lesser than what it is now.
- It moves towards the lower weight and updates the value of the coefficients.
- The process repeats until the local minimum is reached. A local minimum is a point beyond which it can not proceed.



Gradient descent works best for most purposes. However, it has some downsides too. It is expensive to calculate the gradients if the size of the data is huge. Gradient descent works well for convex functions, but it doesn't know how far to travel along the gradient for nonconvex functions.

Stochastic Gradient Descent Deep Learning Optimizer

At the end of the previous section, you learned why using gradient descent on massive data might not be the best option. To tackle the problem, we have stochastic gradient descent. The term stochastic means randomness on which the algorithm is based upon. In stochastic gradient descent, instead of taking the whole dataset for each iteration, we randomly select the batches of data. That means we only take a few samples from the dataset.

$$w := w - \eta \nabla Q_i(w).$$

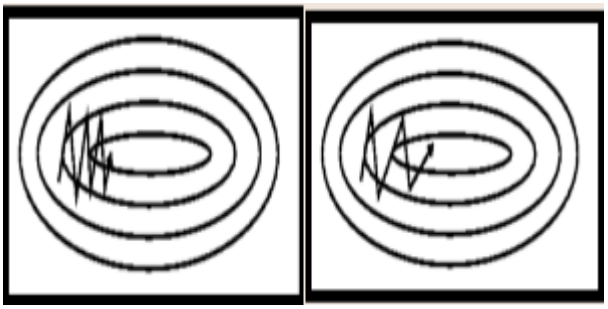
The procedure is first to select the initial parameters w and learning rate n. Then randomly shuffle the data at each iteration to reach an approximate minimum.

Since we are not using the whole dataset but the batches of it for each iteration, the path taken by the algorithm is full of noise as compared to the gradient descent algorithm. Thus, SGD uses a higher number of iterations to reach the local minima. Due to an increase in the number of iterations, the overall computation time increases. But even after increasing the number of iterations, the computation cost is still less than that of the gradient descent optimizer. So the conclusion is if the data is enormous and computational time is an essential factor, stochastic gradient descent should be preferred over batch gradient descent algorithm.

Stochastic Gradient Descent With Momentum Deep Learning Optimizer

As discussed in the earlier section, you have learned that stochastic gradient descent takes a much more noisy path than the gradient descent algorithm. Due to this reason, it requires a more significant number of iterations to reach the optimal minimum, and hence computation time is very slow. To overcome the problem, we use stochastic gradient descent with a momentum algorithm.

What the momentum does is helps in faster convergence of the loss function. Stochastic gradient descent oscillates between either direction of the gradient and updates the weights accordingly. However, adding a fraction of the previous update to the current update will make the process a bit faster. One thing that should be remembered while using this algorithm is that the learning rate should be decreased with a high momentum term.



In the above image, the left part shows the convergence graph of the stochastic gradient descent algorithm. At the same time, the right side shows SGD with momentum. From the image, you can compare the path chosen by both algorithms and realize that using momentum helps reach convergence in less time. You might be thinking of using a large momentum and learning rate to make the process even faster. But

remember that while increasing the momentum, the possibility of passing the optimal minimum also increases. This might result in poor accuracy and even more oscillations.

Mini Batch Gradient Descent Deep Learning Optimizer

In this variant of gradient descent, instead of taking all the training data, only a subset of the dataset is used for calculating the loss function. Since we are using a batch of data instead of taking the whole dataset, fewer iterations are needed. That is why the mini-batch gradient descent algorithm is faster than both stochastic gradient descent and batch gradient descent algorithms. This algorithm is more efficient and robust than the earlier variants of gradient descent. As the algorithm uses batching, all the training data need not be loaded in the memory, thus making the process more efficient to implement. Moreover, the cost function in mini-batch gradient descent is noisier than the batch gradient descent algorithm but smoother than that of the stochastic gradient descent algorithm. Because of this, mini-batch gradient descent is ideal and provides a good balance between speed and accuracy.

Despite all that, the mini-batch gradient descent algorithm has some downsides too. It needs a hyperparameter that is “mini-batch-size”, which needs to be tuned to achieve the required accuracy. Although, the batch size of 32 is considered to be appropriate for almost every case. Also, in some cases, it results in poor final accuracy. Due to this, there needs a rise to look for other alternatives too.

Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer

The adaptive gradient descent algorithm is slightly different from other gradient descent algorithms. This is because it uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training. The more the parameters get changed, the more minor the learning rate changes. This modification is highly beneficial because real-world datasets contain sparse as well as dense features. So it is unfair to have the same value of learning rate for all the features. The Adagrad algorithm uses the below formula to update the weights. Here the alpha(t) denotes the different learning rates at each iteration, n is a constant, and E is a small positive to avoid division by 0.

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_{\mathrm{t}} \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}} \qquad \eta'_{\mathrm{t}} = \frac{\eta}{\mathrm{sqrt}(\alpha_{\mathrm{t}} + \epsilon)}$$

The benefit of using Adagrad is that it abolishes the need to modify the learning rate manually. It is more reliable than gradient descent algorithms and their variants, and it reaches convergence at a higher speed.

One downside of the AdaGrad optimizer is that it decreases the learning rate aggressively and monotonically. There might be a point when the learning rate becomes extremely small. This is because the squared gradients in the denominator keep accumulating, and thus the denominator part keeps on increasing. Due to small learning rates, the model eventually becomes unable to acquire more knowledge, and hence the accuracy of the model is compromised.

RMS Prop (Root Mean Square) Deep Learning Optimizer

RMS prop is one of the popular optimizers among deep learning enthusiasts. This is maybe because it hasn't been published but is still very well-known in the community. RMS prop is ideally an extension of the work RPPROP. It resolves the problem of varying gradients. The problem with the gradients is that some of them were small while others may be huge. So, defining a single learning rate might not be the best idea. RPPROP uses the gradient sign, adapting the step size individually for each weight. In this algorithm, the two gradients are first compared for signs. If they have the same sign, we're going in the right direction, increasing the step size by a small fraction. If they have opposite signs, we must decrease the step size. Then we limit the step size and can now go for the weight update.

The problem with RPPROP is that it doesn't work well with large datasets and when we want to perform mini-batch updates. So, achieving the robustness of RPPROP and the efficiency of mini-batches simultaneously was the main motivation behind the rise of RMS prop. RMS prop is an advancement in AdaGrad optimizer as it reduces the monotonically decreasing learning rate.

RMS Prop Formula

The algorithm mainly focuses on accelerating the optimization process by decreasing the number of function evaluations to reach the local minimum. The algorithm keeps the moving average of squared gradients for every weight and divides the gradient by the square root of the mean square.

$$v(w,t) := \gamma v(w,t-1) + (1-\gamma)(\nabla Q_i(w))^2$$

where gamma is the forgetting factor. Weights are updated by the below formula

$$w := w - \frac{\eta}{\sqrt{v(w,t)}} \nabla Q_i(w)$$

In simpler terms, if there exists a parameter due to which the cost function oscillates a lot, we want to penalize the update of this parameter. Suppose you built a model to classify a variety of fishes. The model relies on the factor ‘color’ mainly to differentiate between the fishes. Due to this, it makes a lot of errors. What RMS Prop does is, penalize the parameter ‘color’ so that it can rely on other features too. This prevents the algorithm from adapting too quickly to changes in the parameter ‘color’ compared to other parameters. This algorithm has several benefits as compared to earlier versions of gradient descent algorithms. The algorithm converges quickly and requires lesser tuning than gradient descent algorithms and their variants.

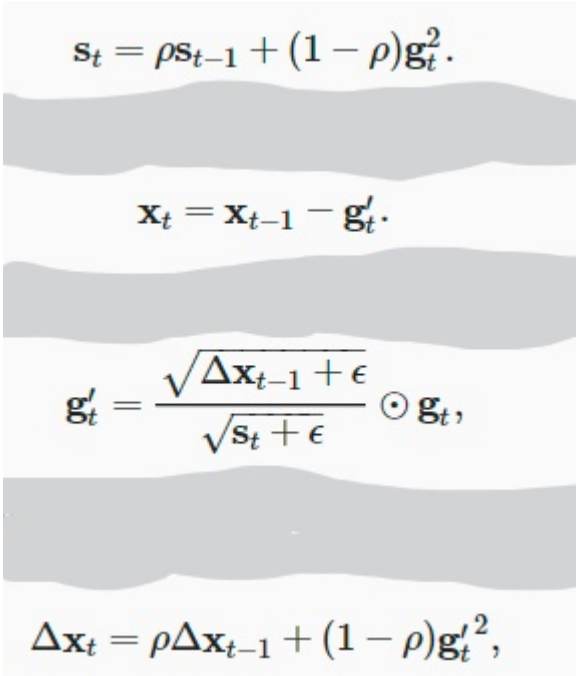
The problem with RMS Prop is that the learning rate has to be defined manually, and the suggested value doesn’t work for every application.

AdaDelta Deep Learning Optimizer

AdaDelta can be seen as a more robust version of the AdaGrad optimizer. It is based upon adaptive learning and is designed to deal with significant drawbacks of AdaGrad and RMS prop optimizer. The main problem with the above two optimizers is that the initial learning rate must be defined

manually. One other problem is the decaying learning rate which becomes infinitesimally small at some point. Due to this, a certain number of iterations later, the model can no longer learn new knowledge.

To deal with these problems, AdaDelta uses two state variables to store the leaky average of the second moment gradient and a leaky average of the second moment of change of parameters in the model.



Here St and delta Xt denote the state variables, g’t denotes rescaled gradient, delta Xt-1 denotes squares rescaled gradients, and epsilon represents a small positive integer to handle division by 0.

Adam Optimizer in Deep Learning

Adam optimizer, short for Adaptive Moment Estimation optimizer, is an optimization algorithm commonly used in deep learning. It is an extension of the stochastic gradient descent (SGD) algorithm and is designed to update the weights of a neural network during training.

The name “Adam” is derived from “adaptive moment estimation,” highlighting its ability to adaptively adjust the learning rate for each network weight individually. Unlike SGD, which maintains a single learning rate throughout training, Adam optimizer dynamically computes individual learning rates based on the past gradients and their second moments.

The creators of Adam optimizer incorporated the beneficial features of other optimization algorithms such as AdaGrad and RMSProp. Similar to RMSProp, Adam optimizer considers the second moment of the gradients, but unlike RMSProp, it calculates the uncentered variance of the gradients (without subtracting the mean).

By incorporating both the first moment (mean) and second moment (uncentered variance) of the gradients, Adam optimizer achieves an adaptive learning rate that can efficiently navigate the optimization landscape during training. This adaptivity helps in faster convergence and improved performance of the neural network.

In summary, Adam optimizer is an optimization algorithm that extends SGD by dynamically adjusting learning rates based on individual weights. It combines the features of AdaGrad and RMSProp to provide efficient and adaptive updates to the network weights during deep learning training.

Adam Optimizer Formula

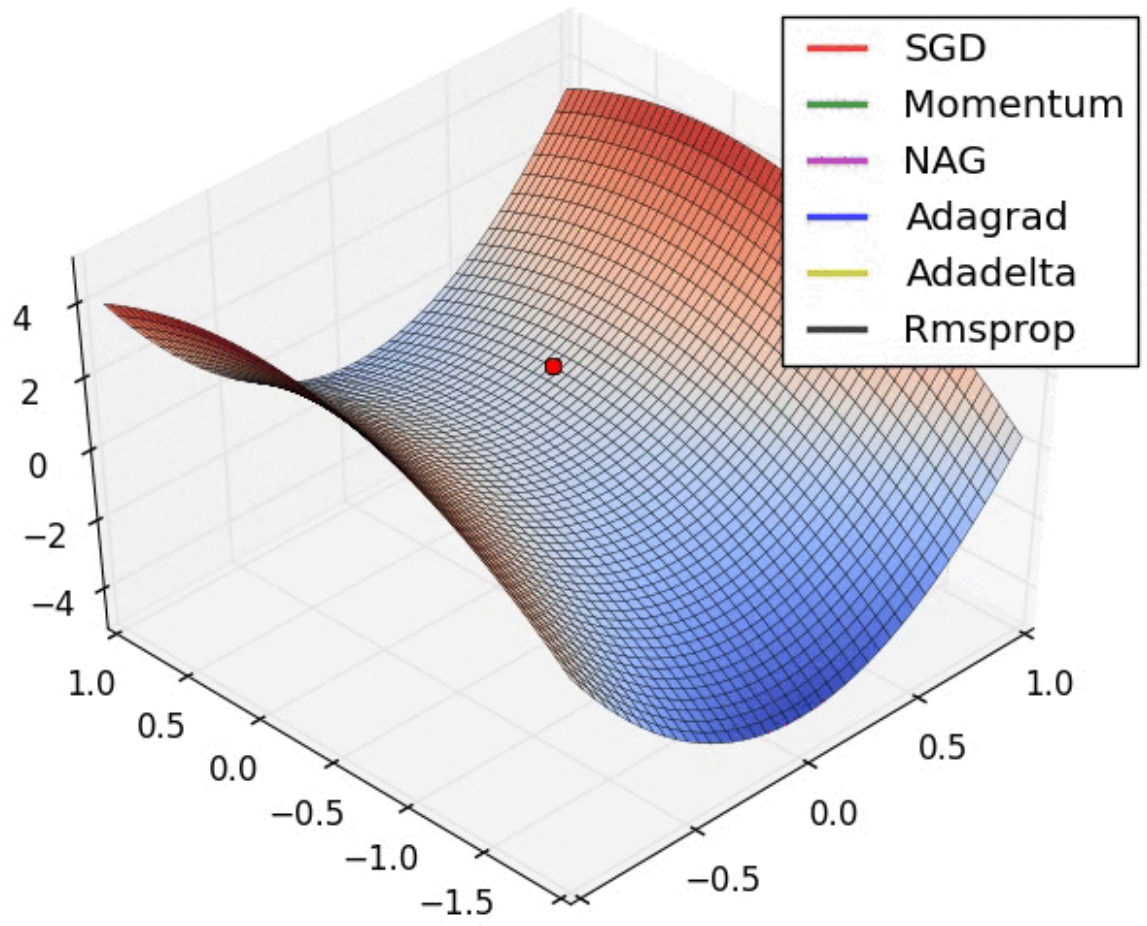
The adam optimizer has several benefits, due to which it is used widely. It is adapted as a benchmark for deep learning papers and recommended as a default optimization algorithm. Moreover, the algorithm is straightforward

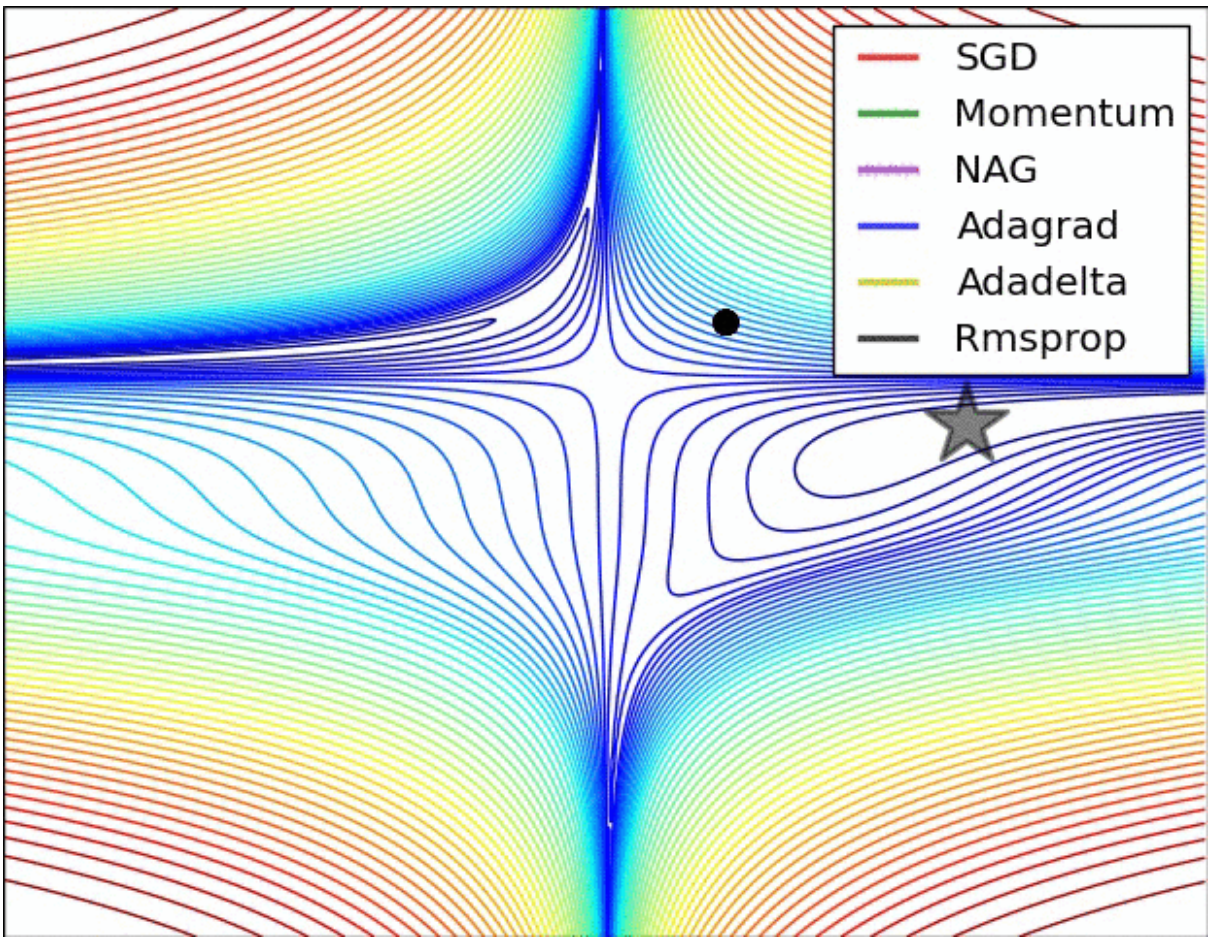
to implement, has a faster running time, low memory requirements, and requires less tuning than any other optimization algorithm.

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

The above formula represents the working of adam optimizer. Here B1 and B2 represent the decay rate of the average of the gradients.

If the adam optimizer uses the good properties of all the algorithms and is the best available optimizer, then why shouldn't you use Adam in every application? And what was the need to learn about other algorithms in depth? This is because even Adam has some downsides. It tends to focus on faster computation time, whereas algorithms like stochastic gradient descent focus on data points. That's why algorithms like SGD generalize the data in a better manner at the cost of low computation speed. So, the optimization algorithms can be picked accordingly depending on the requirements and the type of data.


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5



The above visualizations create a better picture in mind and help in comparing the results of various optimization algorithms.

Hands-on Optimizers

We have learned enough theory, and now we need to do some practical analysis. It's time to try what we have learned and compare the results by choosing different optimizers on a simple neural network. As we are talking about keeping things simple, what's better than the MNIST dataset? We will train a simple model using some basic layers, keeping the batch size and epochs the same but with different optimizers. For the sake of fairness, we will use the default values with each optimizer.

The steps for building the network are given below.

1. Import Necessary Libraries

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
(x_train, y_train), (x_test, y_test) = mnist.load_data()
print(x_train.shape, y_train.shape)
```

2. Load the Dataset

```
x_train= x_train.reshape(x_train.shape[0],28,28,1)
x_test=  x_test.reshape(x_test.shape[0],28,28,1)
input_shape=(28,28,1)
y_train=keras.utils.to_categorical(y_train)#,num_classes=)
y_test=keras.utils.to_categorical(y_test)#, num_classes)
x_train= x_train.astype('float32')
x_test= x_test.astype('float32')
x_train /= 255
x_test /=255
```

3. Build the Model

```
batch_size=64

num_classes=10

epochs=10

def build_model(optimizer):

model=Sequential()

model.add(Conv2D(32,kernel_size=(3,3),activation='relu',input_
shape=input_shape))

model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(256, activation='relu'))

model.add(Dropout(0.5))
```

```
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
optimizer= optimizer, metrics=['accuracy'])

return model
```

4. Train the Model

```
optimizers = ['Adadelta', 'Adagrad', 'Adam', 'RMSprop', 'SGD']

for i in optimizers:

model = build_model(i)

hist=model.fit(x_train, y_train, batch_size=batch_size,
epochs=epochs, verbose=1, validation_data=(x_test,y_test))
```

We have run our model with a batch size of 64 for 10 epochs. After trying the different optimizers, the results we get are pretty interesting. Before analyzing the results, what do you think will be the best optimizer for this dataset?

Optimizer	Epoch 1 Val accuracy | Val loss	Epoch 5 Val accuracy | Val loss	Epoch 10 Val accuracy | Val loss	Total Time
Adadelta	.4612 | 2.2474	.7776 | 1.6943	.8375 | 0.9026	8:02 min
Adagrad	.8411 | .7804	.9133 | .3194	.9286 | 0.2519	7:33 min
Adam	.9772 | .0701	.9884 | .0344	.9908 | .0297	7:20 min
RMSprop	.9783 | .0712	.9846 | .0484	.9857 | .0501	10:01 min
SGD with momentum	.9168 | .2929	.9585 | .1421	.9697 | .1008	7:04 min

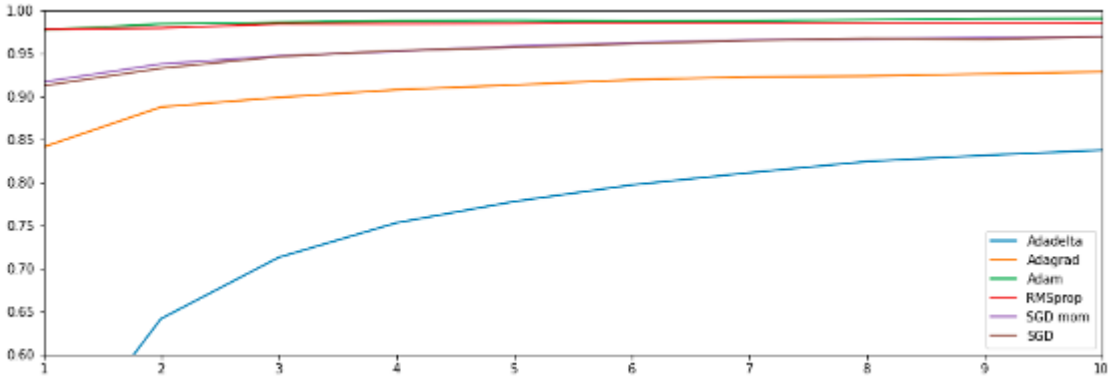
SGD	.9124 | .3157	.9569 | 1451	.9693 | .1040	6:42 min

Table Analysis

The above table shows the validation accuracy and loss at different epochs. It also contains the total time that the model took to run on 10 epochs for each optimizer. From the above table, we can make the following analysis.

- The adam optimizer shows the best accuracy in a satisfactory amount of time.
- RMSprop shows similar accuracy to that of Adam but with a comparatively much larger computation time.
- Surprisingly, the SGD algorithm took the least time to train and produced good results as well. But to reach the accuracy of the Adam optimizer, SGD will require more iterations, and hence the computation time will increase.
- SGD with momentum shows similar accuracy to SGD with unexpectedly larger computation time. This means the value of momentum taken needs to be optimized.
- Adadelta shows poor results both with accuracy and computation time.

You can analyze the accuracy of each optimizer with each epoch from the below graph.



We've now reached the end of this comprehensive guide. To refresh your memory, we will go through a summary of every optimization algorithm that we have covered in this guide. To refresh your memory, we will go through a summary of every optimization algorithm that we have covered in this guide.

Summary

SGD is a very basic algorithm and is hardly used in applications now due to its slow computation speed. One more problem with that algorithm is the constant learning rate for every epoch. Moreover, it is not able to handle saddle points very well. Adagrad works better than stochastic gradient descent generally due to frequent updates in the learning rate. It is best when used for dealing with sparse data. RMSProp shows similar results to that of the gradient descent algorithm with momentum, it just differs in the way by which the gradients are calculated.

Lastly comes the Adam optimizer that inherits the good features of RMSProp and other algorithms. The results of the Adam optimizer are generally better than every other optimization algorithm, have faster computation time, and require fewer parameters for tuning. Because of all

that, Adam is recommended as the default optimizer for most of the applications. Choosing the Adam optimizer for your application might give you the best probability of getting the best results.

But by the end, we learned that even Adam optimizer has some downsides. Also, there are cases when algorithms like SGD might be beneficial and perform better than Adam optimizer. So, it is of utmost importance to know your requirements and the type of data you are dealing with to choose the best optimization algorithm and achieve outstanding results.

Conclusion

This article taught us how an optimization algorithm could affect the deep learning model in terms of accuracy, speed, and efficiency. We learned about various algorithms, and hopefully, you were able to compare the algorithms with one another. We also learned when to use which algorithm and what could be the downsides of using that algorithm.

Key Takeaways

- Gradient Descent, Stochastic Gradient Descent, Mini-batch Gradient Descent, Adagrad, RMS Prop, AdaDelta, and Adam are all popular deep-learning optimizers.
- Each optimizer has its own strengths and weaknesses, and the choice of optimizer will depend on the specific deep-learning task and the characteristics of the data being used.

- The choice of optimizer can significantly impact the speed and quality of convergence during training, as well as the final performance of the deep learning model.

Frequently Asked Questions

Q1. What are some of the use cases where a deep learning model is trained?

A. Deep learning models can be trained for various use cases, such as image and speech recognition, natural language processing, recommendation systems, fraud detection, autonomous vehicles, predictive analytics, medical diagnosis, text generation, and video analysis.

Q2. How does artificial intelligence contribute to deep learning optimization through optimizers?

A. Artificial intelligence plays a significant role in deep learning optimizers by using various algorithms to automate and improve the learning process of deep neural networks. This includes techniques such as gradient descent, adaptive learning rates, and momentum, which help to optimize the network's performance and accuracy. Additionally, AI-powered optimizers such as Adam, Adagrad, and RMSProp can automatically adjust the learning rate and other hyperparameters, making the optimization process more efficient and effective.

Q3. What role do optimizers play in computer vision with deep learning?

A. In the field of computer vision, deep learning optimizers are a crucial aspect as they ensure optimal results are achieved during the training process. The optimizer's function is to minimize the loss, which measures the discrepancy between the predicted values and actual values, by

continuously adjusting the model's parameters. Choosing the right optimizer can significantly impact the speed and accuracy of the training and, ultimately, the final results. Hence, optimizers play a vital role in deep learning applications in computer vision.

Q4. What is the definition of an optimizer?
A. An optimizer, in the context of machine learning and optimization algorithms, is a computational technique or algorithm used to adjust the parameters of a model to minimize or maximize a specific objective function. It aims to find the optimal set of parameters that result in the best performance, such as minimizing loss in a neural network during training, by iteratively updating parameter values based on gradients or other criteria.

Github Url : https://github.com/Amirhossein1410/Optimizers_in_deep_learning