[ ] Start coding or generate with AI.



## What is Optimizer?

In deep learning, an optimizer is a crucial element that fine-tunes a neural network's parameters during training. Its primary role is to minimize the model's error or loss function, enhancing performance. Various optimization algorithms, known as optimizers, employ distinct strategies to converge towards optimal parameter values for improved predictions efficiently.

## What Are Optimizers in Deep Learning?

In deep learning, optimizers are crucial as algorithms that dynamically fine-tune a model's parameters throughout the training process, aiming to minimize a predefined loss function. These specialized algorithms facilitate the learning process of neural networks by iteratively refining the weights and biases based on the feedback received from the data. Well-known optimizers in deep learning encompass Stochastic Gradient Descent (SGD), Adam, and RMSprop, each equipped with distinct update rules, learning rates, and momentum strategies, all geared towards the overarching goal of discovering and converging upon optimal model parameters, thereby enhancing overall performance.

Optimizer algorithms are optimization method that helps improve a deep learning model's performance. These optimization algorithms or optimizers widely affect the

models 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.

## Gradient Descent Deep Learning Optimizer

Gradient Descent can be considered the popular kid among the class of optimizers in deep learning. This optimization algorithm uses calculus to consistently modify the values and achieve the local minimum. Before moving ahead, you might question 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)
```

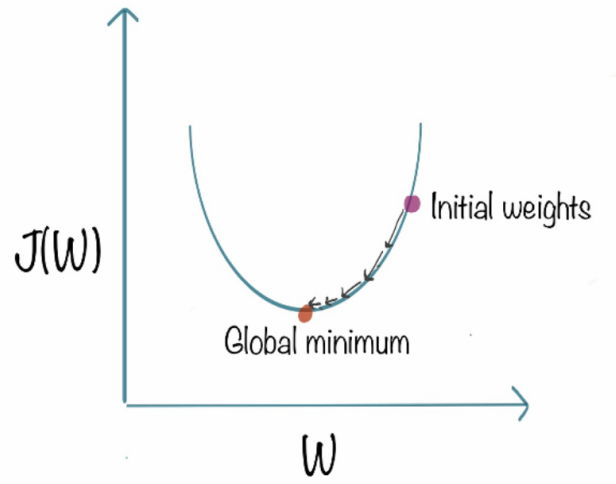
The above equation means how the gradient is calculated. Here alpha is the step size that represents how far to move against each gradient with each iteration.

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 there might be better options than using gradient descent on massive data. To tackle the challenges large datasets pose, we have stochastic gradient descent, a popular approach among optimizers in deep learning. The term stochastic denotes the element of randomness upon which the algorithm relies. In stochastic gradient descent, instead of processing the entire dataset during each iteration, we randomly select batches of data. This implies that only a few samples from the dataset are considered at a time, allowing for more

efficient and computationally feasible optimization in deep learning models.

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

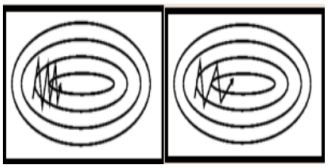
The procedure is first to select the initial parameters w and learning rate n. Then randomly shuffle the data at each iteration to reach an approximate minimum.

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 when addressing optimizers in deep learning. Due to this, 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.



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

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.

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_t \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}}$$

$$\eta'_t = \frac{\eta}{\mathrm{sqrt}(\alpha_t + \epsilon)}$$

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

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.

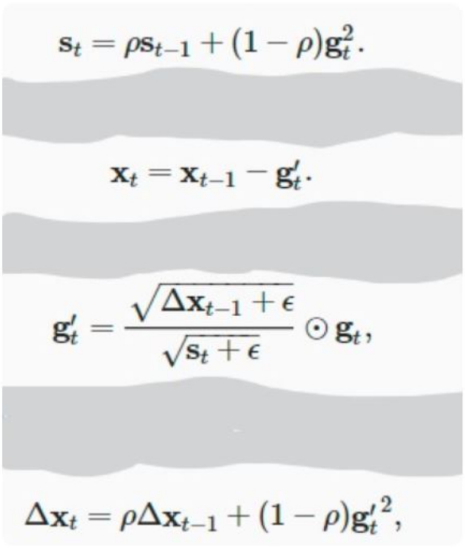
The problem with RMS Prop is that the learning rate has to be defined manually, and the suggested value doesn't work for every application.

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

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

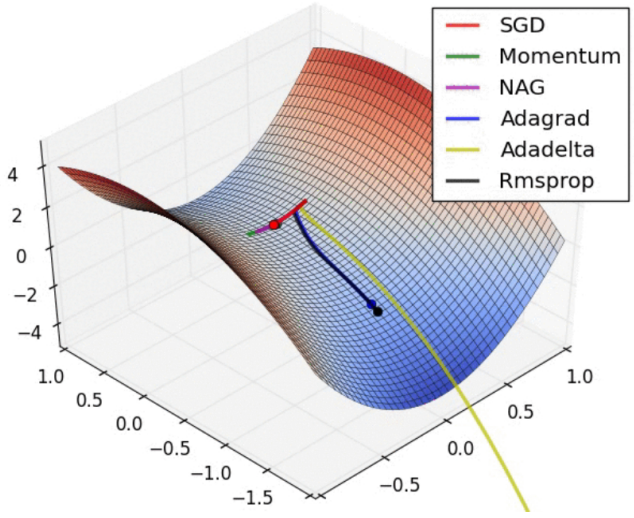
Adam Optimizer Formula

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.




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)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 28, 28) (60000,)
```

## 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, met

  return model
```

## 4. Train the Model

```
Adagrad', 'Adam', 'RMSprop', 'SGD']
```

```
_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(x_test,y_t
```

| Optimizer | Epoch 1<br>Val accuracy \| Val loss | Epoch 5<br>Val accuracy \| Val loss | Epoch 10<br>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 |

Double-click (or enter) to edit

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?

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.