Understanding Optimizers

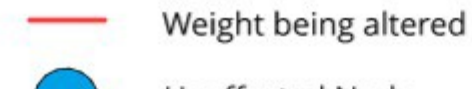
Harsha Bommana Follow
Oct 7, 2019 · 11 min read

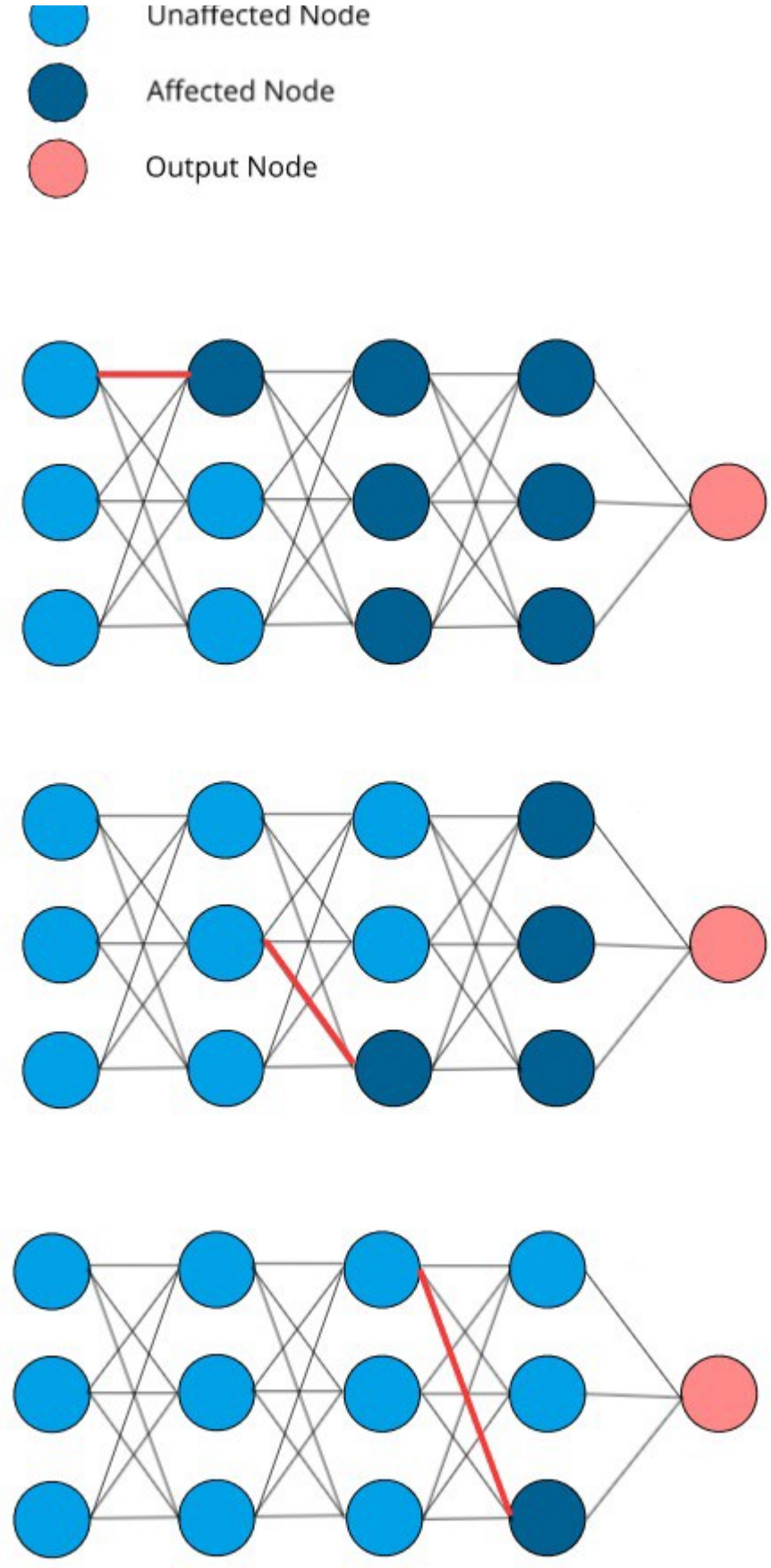
In deep learning we have the concept of loss, which tells us how poorly the model is performing at that current instant. Now we need to use this loss to **train** our network such that it performs better. Essentially what we need to do is to take the loss and try to **minimize** it, because a lower loss means our model is going to perform better. The process of minimizing (or maximizing) any mathematical expression is called **optimization** and we need now see how we can use these optimization methods for neural networks.

In a neural network, we have many **weights** in between each layer. We have to understand that **each and every** weight in the network will affect the output of the network in some way, because they are all directly or indirectly connected to the output.

Hence we can say that if we change any particular weight in the neural network, the output of the network will also change.

As you can see in the illustration below, we are taking 3 different scenarios. In each scenario, we are selecting some random weight and we are altering it. We are also seeing what parts of the neural network get affected and which don't get affected **after** we change the selected weight. In all 3 scenarios, there is at least one affected node in the final layer of the network. Since all nodes of the last layer are also connected to the output node, it is safe to say that as long as some part of the last layer gets affected, the output node will also be affected.





Visualizing which parts of a network altering particular weights will affect

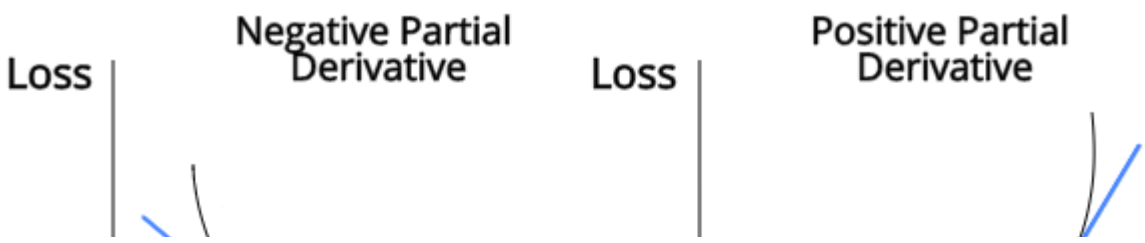
From the diagram we can also observe that the weights **further away** from the output node (closer to the start of the network), affect more nodes in between. Hence we can say that they affect the output very **indirectly** since there are a lot of weights between them and the output. The weights which are closer to the output affect fewer nodes in between, hence they have a more **direct** impact on the output node.

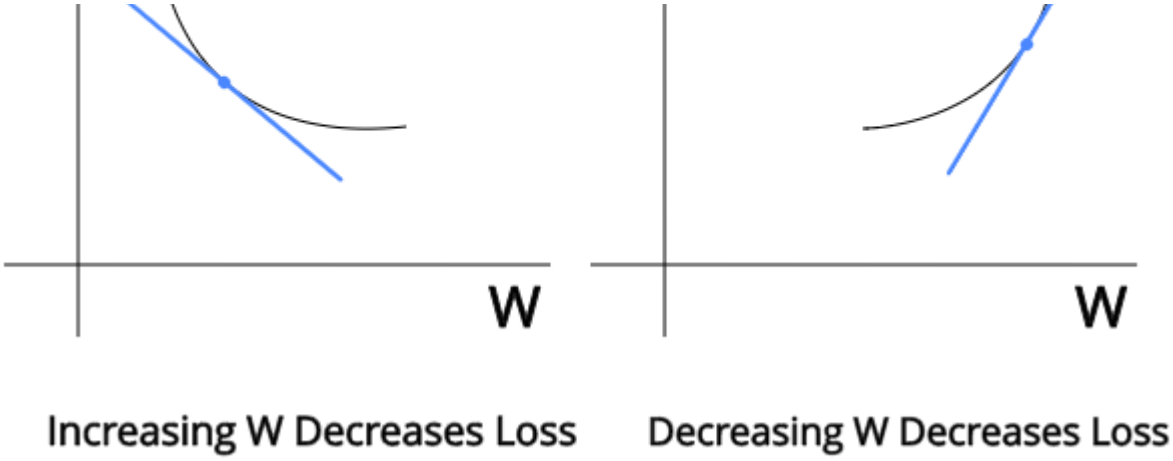
Now that we understand how to change the output of the network by changing the weights, let's go ahead to understand how we can minimize the loss. Changing the weights **changes the output.** Changing the output **changes the loss**, since loss is a function of the predicted value (Y_pred), which is basically the output of the network. Hence we can say that changing the weights will ultimately **change the loss.**

We have established the relationship between the weights and the final loss, but so far we have only talked about **changing.** Changing can mean increase or decrease, but we need to **decrease** the loss. So now we need to see how to **change** the weights in such a way that the loss **decreases.** This process is called optimization.

Looking at it from a mathematical perspective, we can do this by using **partial derivatives.** A partial derivative will allow us to understand how two mathematical expressions **affect each other.** Let us take X and Y, which are connected by some arbitrary mathematical relationship. If we find the partial derivative of X **with respect to** Y, we can understand how changing X will affect Y. If the partial derivative is **positive,** that means **increasing X** will also **increase Y**. If it's **negative** that means **increasing X** will **decrease Y**.

Hence we need to find the partial derivatives of each and every **weight** in the neural network **with respect to** the loss. At a particular instant, if the weight's partial derivative is **positive,** then we will **decrease** that weight in order to decrease the loss. If the partial derivative is **negative,** then we will **increase** that weight in order to decrease the loss. Our ultimate goal is to decrease the loss afterall.





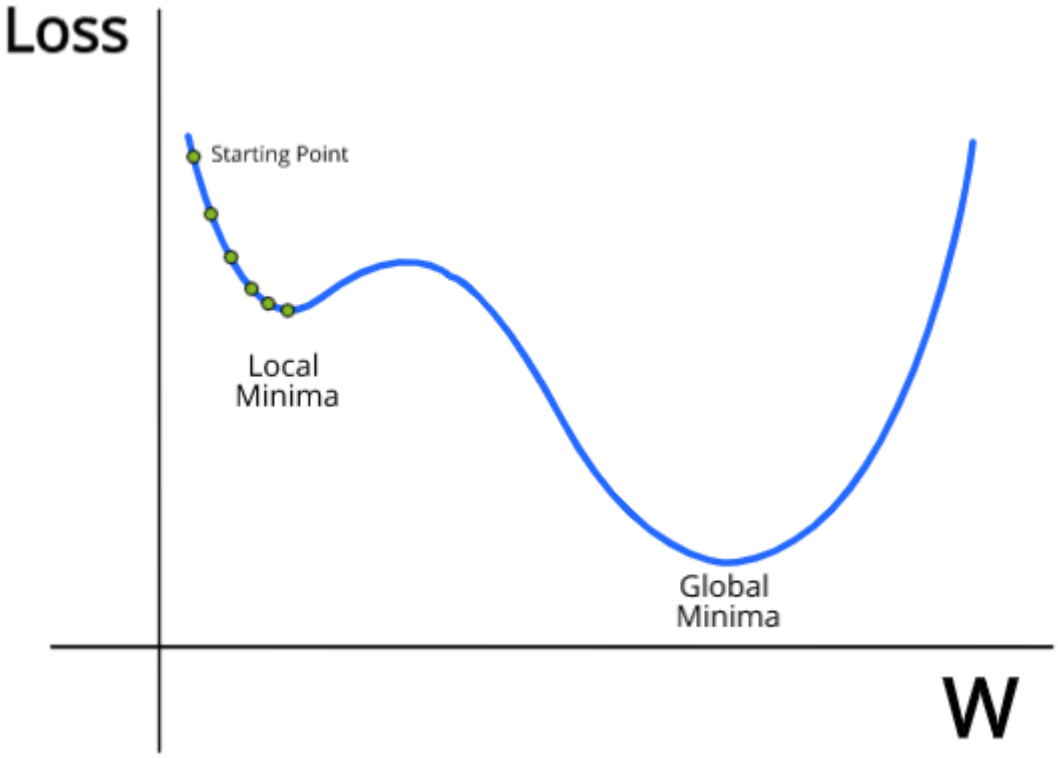
Partial Derivative for optimizing loss visual

This algorithm is called **Gradient Descent**, also called Stochastic Gradient Descent (SGD). And this is the most basic method of optimizing neural networks. This happens as an iterative process and hence we will update the value of each weight **multiple** times before the Loss converges at a suitable value. Let's look at the mathematical way of representing a single update:

$$W_{new} = W_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

Here the alpha symbol is the **learning rate.** This will affect the **speed** of optimization of our neural network. If we have a large learning rate, we will reach the minima for Loss faster because we are taking big steps, however as a result we may not reach a very good minimum since we are taking big steps and hence we might **overshoot** it. A smaller learning rate will solve this issue, but it will take a lot of steps for the neural network's loss to decrease to a good value. Hence we need to keep a learning rate at an optimal value. Usually keeping alpha $= 0.01$ is a safe value.

There is, however, one big problem with just gradient descent. We are not pushing the loss towards the **global minima**, we are merely pushing it towards the closest **local minima.** Let's see and illustration of this.



SGD Local Minima Problem

Here, we are starting from the labelled green dot. Every subsequent green dot represents the loss and new weight value after a **single update** has occurred. The gradient descent will only happen till the local minima since the **partial derivative (gradient)** near the local minima is **near zero.** Hence it will stay near there after reaching the local minima and will not try to reach the global minima.

This is a rather simple graph and in reality the graph will be much more complicated than this with **many** local minimas present, hence if we use just gradient descent we are not guaranteed to reach a good loss. We can combat this problem by using the concept of **momentum**.

Momentum

In momentum, what we are going to do is essentially try to capture some information regarding the **previous updates** a weight has gone through before performing the current update. Essentially, if a weight is constantly moving in a particular direction (increasing or decreasing), it will slowly accumulate some *"momentum"* in that direction. Hence when it faces some resistance and actually has to go the opposite way, it

will still continue going in the original direction for a while because of the accumulated momentum.

This is comparable to the actual momentum in physics. Let's consider a small valley and a ball. If we release the ball at one side of the valley, it will continuously roll down and in that process it is gaining momentum towards that direction. Finally when the ball reaches the bottom, it doesn't stop there. Instead it starts rolling up on the opposite side for a while since it has gained some momentum even though gravity is telling it to come down.

We are essentially trying to recreate this scenario mathematically so that the gradient descent algorithm can try to bypass local minima to try and reach the global minima. Let's look at the formula:

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

Momentum Update Formula

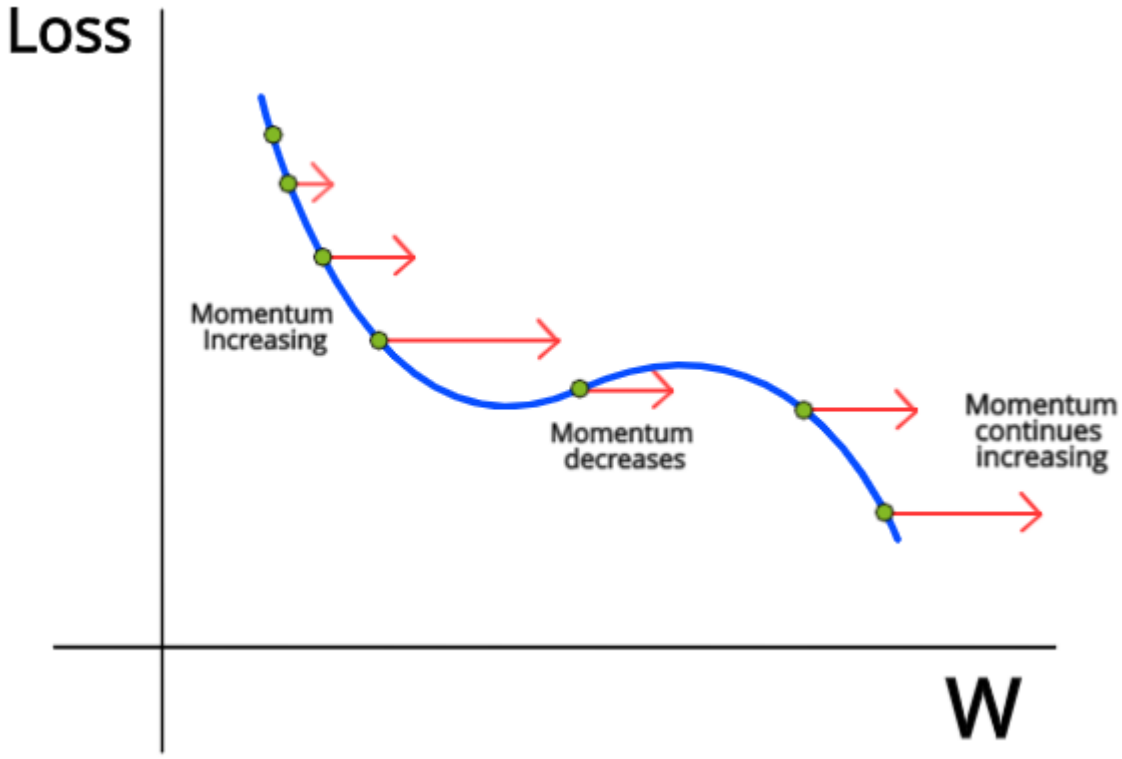
$$W_{new} = \nu_{new} + W_{old}$$

Weight Update Formula (Momentum)

Here, V is the **momentum factor.** As you can see, in each update it essentially adds the current derivative with a part of the **previous** momentum factor. And then we just add this to the weight to get the updated weight. n here is the **coefficient of momentum** and it decides how much momentum gets **carried forward** each time.

As the weight gets updated, it will essentially store a *portion* of all the **previous gradients** in the momentum factor. Hence once it faces a change in the opposite direction, like a local minima, it will continue moving in the same direction until the magnitude of the momentum factor **gradually decreases** and starts pointing in the

opposite direction. In most cases, the momentum factor usually is enough to make the weights overcome the local minima.



Momentum visual example

Another added advantage of momentum is that because of the accumulation of gradients, the weight will converge at an appropriate loss much faster. Let's now look at another technique in optimization which allows gradient descent to happen in a smarter way.

Nesterov Accelerated Gradients (NAG)

In NAG, what we try to do is instead of calculating the gradients at the **current position** we try to calculate it from an approximate **future position**. This is because we want to try to calculate our gradients in a smarter way. Just before reaching a minima, the momentum will start reducing before it reaches it because we are using gradients from a future point. This results in improved stability and lesser oscillations while converging, furthermore, in practice it performs better than pure momentum.

Let's look at how exactly we use NAG in optimizing a weight in a neural network.

If we look at the whole momentum equation expanded:

$$W_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})} + W_{old}$$

Momentum full formula expanded

Here, we see that the next W value is essentially adding **n*v_old** with **alpha*current gradient** and the **current weight**. Considering the fact that alpha*current gradient is going to be a **very small** value, we can approximate the next W value by just adding n*v_old with the current W value. This will give an approximate **future W** value.

$$W_{future} = \eta * \nu_{old} + W_{old}$$

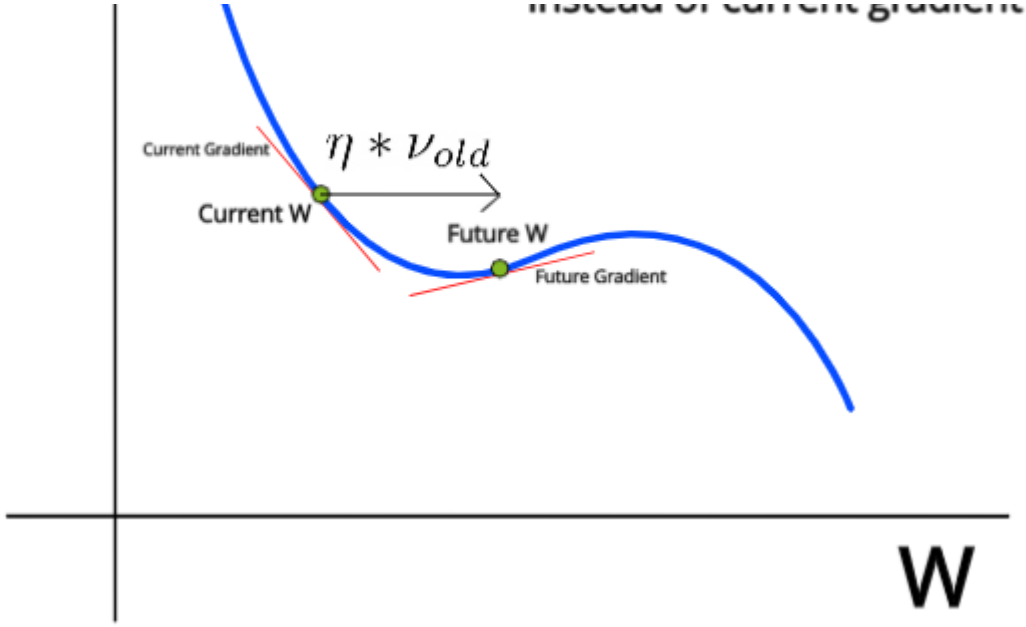
Calculating approximate future weight value

Now what we want to do is instead of calculating gradients with respect to the current W value, we will calculate them with respect to the future W value. This allows the momentum factor to start adapting to sharp gradient changes before they even occur, leading to increased stability while training.

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{future})}$$

Nesterov Momentum Formula

This is the new momentum equation with NAG. As you can see, we are taking gradients from an approximate future value of W instead of the current value of W.



Nesterov VIsual Example

Here is an example of using NAG. At the current W value, we add n*v_old to approximate the future W value. Then we calculate the gradient at that point and use that instead of current gradient value while calculating the v_new value. As you can see in this example, even though the momentum is technically **supposed to increase** in this scenario, it will start **decreasing at this point** itself because the **future W** value has a gradient pointing in the **opposite direction**. Now let's look at some adaptive optimization methods.

Adaptive Optimization

In these methods, the parameters like learning rate (alpha) and momentum coefficient (n) will not stay constant throughout the training process. Instead, these values will constantly adapt for each and every weight in the network and hence will also change along with the weights. These kind of optimization algorithms fall under the category of *adaptive optimization*.

The first algorithm we are going to look into is **Adagrad.**

Adagrad

Adagrad is short for *adaptive gradients.* In this we try to change the learning rate (alpha) for each update. The learning rate changes during each update in such a way that it will

decrease if a weight is being updated too much in a short amount of time and it will increase if a weight is not being updated much.

First, each weight has its own **cache** value, which collects the squares of the gradients till the current point.

$$cache_{new} = cache_{old} + (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

Cache updation for Adagrad

The cache will continue to increase in value as the training progresses. Now the new update formula is as follows:

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adagrad Update Formula

This is of the same form as the original gradient descent formula except for the fact that the learning rate (alpha) constantly changes throughout the training. The E in the denominator is a very small value to ensure division by zero does not occur.

Essentially what's happening here is that if a weight has been having **very huge updates**, it's cache value is also going to **increase**. As a result, the learning rate will **become lesser** and that weight's update magnitudes will decrease over time. On the other hand, if a weight has not been having any significant update, it's cache value is going to be **very less**, and hence its learning rate will **increase**, forcing it to take **bigger updates**. This is the basic principle of the Adagrad optimizer.

However the disadvantage of this algorithm is that regardless of a weight's past gradients, the cache will **always increase** by some amount because square cannot be negative. Hence the learning rate of every weight will eventually decrease to a **very low value** till training does not happen significantly anymore.

The next adaptive optimizer, **RMSProp** effectively solves this problem.

RMSProp

In RMSProp the only difference lies in the cache updating strategy. In the new formula, we introduce a new parameter, the decay rate (gamma).

$$cache_{new} = \gamma * cache_{old} + (1 - \gamma) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

RMSProp Cache Update Formula

Here the gamma value is usually around 0.9 or 0.99. Hence for each update, the square of gradients get added at a very slow rate compared to adagrad. This ensures that the learning rate is changing constantly based on the way the weight is being updated, just like adagrad, but at the same time the learning rate does not decay too quickly, hence allowing training to continue for much longer.

Next we will look into Adam optimizer, which is widely regarded as one of the best go to optimizers for deep learning in general.

Adam

Adam is a little like combining RMSProp with Momentum. First we calculate our m value, which will represent the momentum at the current point.

$$m_{new} = \beta_1 * m_{old} - (1 - \beta_1) * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adam Momentum Update Formula

The only difference between this equation and the momentum equation is that instead of the learning rate we keep (1-Beta_1) to be multiplied with the current gradient.

Next we will calculate the accumulated cache, which is exactly the same as it is in RMSProp:

$$cache_{new} = \beta_2 * cache_{old} + (1 - \beta_2) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

Now we can get the final update formula:

$$W_{new} = W_{old} - \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * m_{new}$$

Adam weight updation formula

As we can observe here, we are performing accumulating the gradients by calculating momentum and also we are constantly changing the learning rate by using the cache. Due to these two features, Adam usually performs better than any other optimizer out there and is usually preferred while training neural networks. In the paper for Adam, the recommended parameters are 0.9 for beta_1, 0.99 for beta_2 and 1e-08 for epsilon.

That concludes this article in which I have covered most of the important optimizers which most deep learning practitioners use on a regular basis. I hope you have learnt something new! Thanks for reading!

```
Note: I am using notation d(Loss)/d(Wi) which actually represents
dLoss/dW at the value of Wi.
```

Machine Learning Deep Learning Artificial Intelligence Data Science Mathematics