Thursday, 11 April 2019

Optimiser

Stochastic Gradient Descent (SGD)

SGD updates model parameters (*theta*) in the negative direction of the gradient (*g*) by taking a subset or a mini-batch of data of size (*m*):

$$g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$

$$\theta = \theta - \epsilon_k \times g$$

The neural network is represented by
f(x(i); theta)
x(i) are the training data and
y(i) are the training labels,
the gradient of the loss *L* is computed with respect to model parameters *theta*.

The learning rate (*eps_k*) determines the size of the step that the algorithm takes along the gradient (in the negative direction in the case of minimization and in the positive direction in the case of maximization).

The learning rate is a function of iteration *k* and is a single most important hyper-parameter. A learning rate that is too high (e.g. > 0.1) can lead to parameter updates that miss the optimum value, a learning rate that is too low (e.g. < 1e-5) will result in unnecessarily long training time. A good strategy is to start with a learning rate of 1e-3 and use a learning rate schedule that reduces the learning rate as a function of iterations (e.g. a step scheduler thathalves the learning rate every 4 epochs):

```
def step_decay(epoch):
    lr_init = 0.001
    drop = 0.5
    epochs_drop = 4.0
    lr_new = lr_init * \
             math.pow(drop, math.floor((1+epoch)/
epochs_drop))
    return lr_new
```

In general, we want the learning rate (*eps_k*) to satisfy the Robbins-Monroe conditions:

$$\sum_k \epsilon_k = \infty \qquad \sum_k \epsilon_k^2 < \infty$$

The first condition ensures that the algorithm will be able to find a locally optimal solution regardless of the starting point and the second one controls oscillations.

Momentum

Momentum accumulates exponentially decaying moving average of past gradients and continues to move in their direction:

$$v = \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_i L(f(x^{(i)}; \theta), y^{(i)}) \right)$$

$$\theta = \theta + v$$

Thus step size depends on how large and how aligned the sequence of gradients are, common values of momentum parameter alpha are 0.5 and 0.9.

Nesterov Momentum

Nesterov Momentum is inspired by Nesterov's accelerated gradient method:

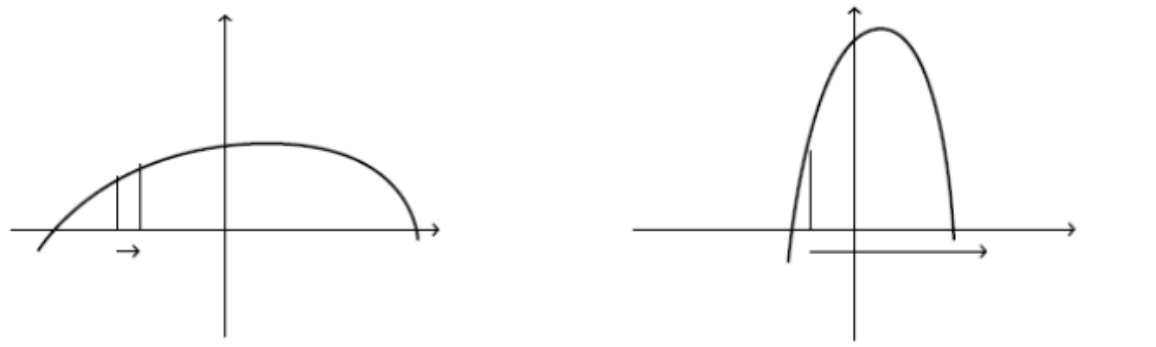
$$v = \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_i L(f(x^{(i)}; \theta + \alpha \times v), y^{(i)}) \right)$$

$$\theta = \theta + v$$

The difference between Nesterov and standard momentum is where the gradient is evaluated, with Nesterov's momentum the gradient is evaluated after the current velocity is applied, thus Nesterov's momentum adds a correction factor to the gradient.

AdaGrad

AdaGrad is an adaptive method for setting the learning rate [3]. Consider two scenarios in the Figure below.



slowly varying log-likelihood (small gradient)

rapidly varying log-likelihood (large gradient)

In the case of a slowly varying objective (left), the gradient would typically (at most points) have a small magnitude. As a result, we would need a large learning rate to quickly reach the optimum. In the case of a rapidly varying objective (right), the gradient would typically be very large. Using a large learning rate would result in very large steps, oscillating around but not reaching the optimum.

These two situations occur because the learning rate is set independent of the gradient. AdaGrad solves this by accumulating squared norms of gradients seen so far and dividing the learning rate by the square root of this sum:

$$g = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$$

$$s = s + g^T g$$

$$\theta = \theta - \epsilon_k \times g/\sqrt{s + eps}$$

As a result parameters that receive high gradients will have their effective learning rate reduced and parameters that receive small gradients will have their effective learning rate increased. The net effect is greater progress in the more gently sloped directions of parameter space and more cautious updates in the presence of large gradients.

RMSProp

RMSProp modifies AdaGrad by changing the gradient accumulation into an exponentially weighted moving average, i.e. it discards history from the distant past [4]:

$$g = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$$

$$s = \text{decay\_rate} \times s + (1 - \text{decay\_rate})g^T g$$

$$\theta = \theta - \epsilon_k \times g/\sqrt{s + eps}$$

Notice that AdaGrad implies a decreasing learning rate even if the gradients remain constant due to accumulation of gradients from the beginning of training. By introducing exponentially weighted moving average we are weighing recent past more heavily in comparison to distant past. As a result, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam

Adam derives from "**adaptive moments**", it can be seen as a variant on the combination of RMSProp and momentum, the update looks like RMSProp except that a smooth version of the gradient is used instead of the raw stochastic gradient, the full Adam update also includes a bias correction mechanism [5]:

$$g = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$$

$$m = \beta_1 m + (1 - \beta_1)g$$

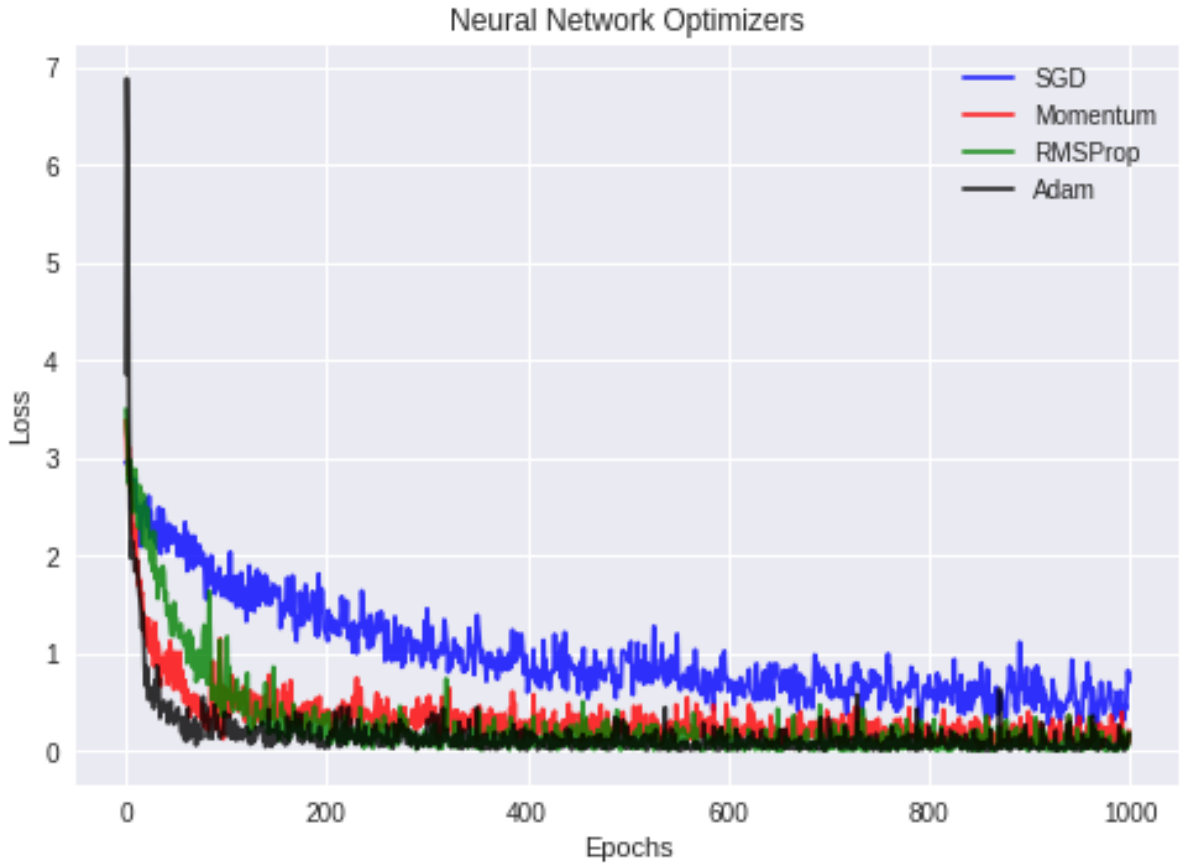
$$s = \beta_2 s + (1 - \beta_2)g^T g$$

$$\theta = \theta - \epsilon_k \times m/\sqrt{s + eps}$$

The recommended values are beta_1 = 0.9, beta_2 = 0.999, and eps = 1e-8.

Experiments

A simple CNN architecture was trained on MNIST dataset using TensorFlow with 1e-3 learning rate and cross-entropy loss using four different optimizers: SGD, Nesterov Momentum, RMSProp and Adam. The Figure below shows the value of the training loss vs iterations:



What are Optimization Algorithms ?

Optimization algorithms helps us to ***minimize (or maximize)*** an **Objective** function (*another name for* ***Error*** *function*) **E(x)** which is simply a mathematical function dependent on the Model's internal **learnable parameters** which are used in computing the target values(**Y**) from the set of *predictors*(**X**) used in the model. For example — we call the **Weights(W)** and the **Bias(b)** values of the neural network as its internal learnable *parameters* which are used in computing the output values and are learned and updated in the direction of optimal solution i.e minimizing the **Loss** by the network's training process and also play a major role in the ***training*** process of the Neural Network Model .

The internal parameters of a Model play a very important role in efficiently and effectively training a Model and produce accurate results. This is why we use various Optimisation strategies and algorithms to update and calculate appropriate and optimum values of such model's parameters which influence our Model's learning process and the output of a Model.

Types of optimization algorithms ?
Optimization Algorithm falls in 2 major categories -

- **First Order Optimization Algorithms** — These algorithms minimize or maximize a Loss function **E(x)** using its ***Gradient*** values with respect to the parameters. Most widely used First order optimization algorithm is **Gradient Descent.**The First order derivative tells us whether the function is decreasing or increasing at a particular point. First order Derivative basically give us a **line** which is ***Tangential*** *to a point on its Error Surface.*

What is a Gradient of a function?
A **Gradient** is simply a vector which is a multi-variable generalization of a ***derivative***(**dy/dx**) which is the *instantaneous rate of change of* ***y with respect to x.*** *The difference is that to calculate a derivative of a function which is dependent on more than one variable or multiple variables, a* ***Gradient takes its place. And a gradient is calculated using Partial Derivatives .*** *Also another major difference between the* ***Gradient*** *and a* ***derivative*** *is that a* ***Gradient*** *of a function produces a* ***Vector Field****.*

A **Gradient** is represented by a ***Jacobian*** Matrix — which is simply a Matrix consisting of ***first order partial Derivatives(Gradients)***.

Hence summing up, a derivative is simply defined for a function dependent on single variables , whereas a Gradient is defined for function dependent on multiple variables. Now let's not get more into Calculas and Physics.

2. **Second Order Optimization Algorithms** — Second-order methods use the **second order derivative** which is also called **Hessian** to minimize or maximize the **Loss** function.The **Hessian** is a Matrix of ***Second Order Partial Derivatives***. ***Since the second derivative is costly to compute, the second order is not used much*** .The second order derivative tells us whether the ***first derivative*** is increasing or decreasing which hints at the function's curvature.Second Order Derivative provide us with a **quadratic** surface which touches the curvature of the **Error Surface**.

Some Advantages of Second Order Optimization over First Order —
Although the Second Order Derivative may be a bit costly to find and calculate, but the advantage of a ***Second order Optimization*** *Technique is that is does not neglect or ignore the* ***curvature of Surface****.Secondly, in terms of Step-wise Performance they are better.*

So which Order Optimization Strategy to use ?

- Now **The First Order Optimization** techniques are easy to compute and less time consuming , converging pretty fast on large data sets.
- **Second Order Techniques** are faster only when the **Second Order Derivative** is known otherwise, these methods are always slower and costly to compute in terms of both time and memory.

Although ,sometimes ***Newton's Second Order Optimization*** *technique can sometimes Outperform* ***First Order Gradient Descent*** *Techniques because Second Order Techniques will not get stuck around paths of slow convergence around* ***saddle points*** *whereas* ***Gradient Descent*** *sometimes gets stuck and does not converges.*
Best way to know which one converges fast is to try it out yourself.

Gradient Descent

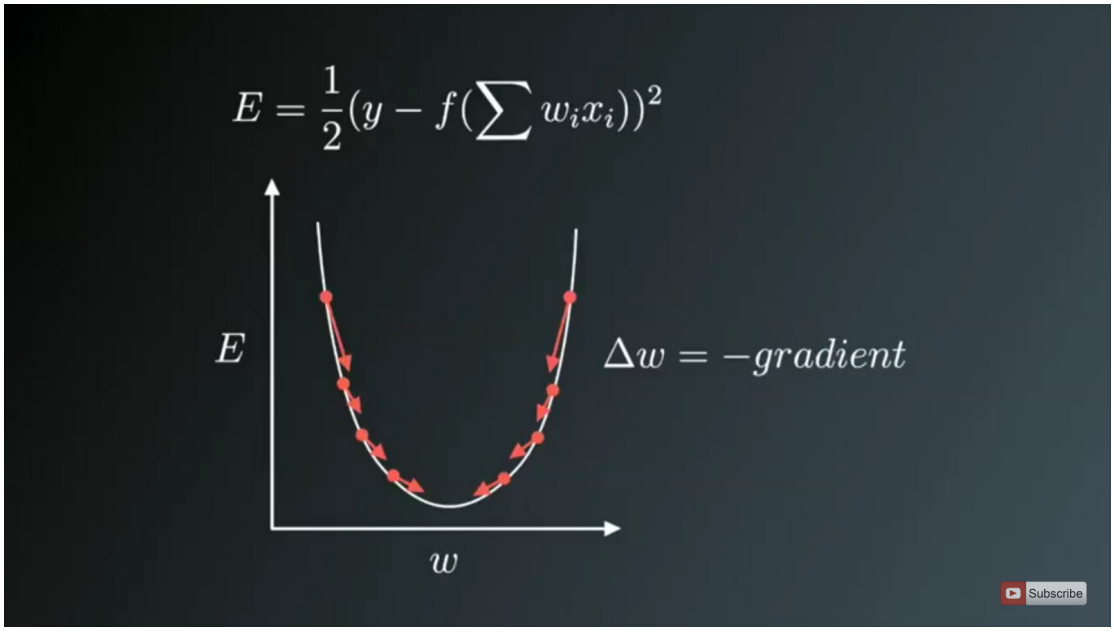
Gradient Descent is the most important technique and the foundation of how we train and optimize ***Intelligent Systems.***

What is does is **—**

"Oh Gradient Descent — Find the Minima , control the variance and then update the Model's parameters and finally lead us to Convergence"

$\theta = \theta - \eta \cdot \nabla J(\theta)$ — is the formula of the parameter updates, where '**η**' is the learning rate ,'**∇J(θ)**' is the **Gradient** of ***Loss function*-J(θ)** w.r.t *parameters-'θ'*.

It is the most popular Optimization algorithms used in optimizing a Neural Network. Now gradient descent is majorly used to do **Weights updates** in a Neural Network Model , i.e update and tune the Model's parameters in a direction so that we can minimize the **Loss function**. Now we all know a Neural Network trains via a famous technique called **Backpropagation ,** in which we first propagate forward calculating the dot product of Inputs signals and their corresponding Weights and then apply a ***activation function*** to those sum of products, which transforms the input signal to an output signal and also is important to model complex Non-linear functions and introduces **Non-linearities** to the Model which enables the Model to learn almost any *arbitrary functional mappings.*After this we propagate **backwards** in the Network carrying **Error** terms and updating **Weights** values using Gradient Descent, in which we calculate the gradient of **Error(E) function** with respect to the **Weights** (**W**) or the parameters , and update the parameters (here **Weights**) in the opposite direction of the Gradient of the Loss function w.r.t to the Model's parameters.



Weight updates in the opposite direction of the Gradient.The image on above shows the process of Weight updates in the opposite direction of the Gradient Vector of Error w.r.t to the Weights of the Network. The **U-Shaped** curve is the Gradient(slope). As one can notice if the Weight**(W)** values are too small or too large then we have large Errors , so want to update and optimize the weights such that it is neither too small nor too large , so we descent downwards opposite to the Gradients until we find a **local minima**.

Variants of Gradient Descent-

The traditional *Batch Gradient Descent* will calculate the gradient of the whole Data set but *will perform only* ***one update , hence it can be very slow and hard to control for datasets which are very very large and don't fit in the Memory****.* How big or small of an update to do is determined by the Learning Rate -**η ,** and it is guaranteed to converge to the ***global minimum*** for convex error surfaces and to a ***local minimum*** for non-convex surfaces.*Another thing while using Standard batch Gradient descent is that it computes redundant updates for large data sets.*

The above problems of Standard Gradient Descent are rectified in Stochastic Gradient Descent.

1. Stochastic gradient descent

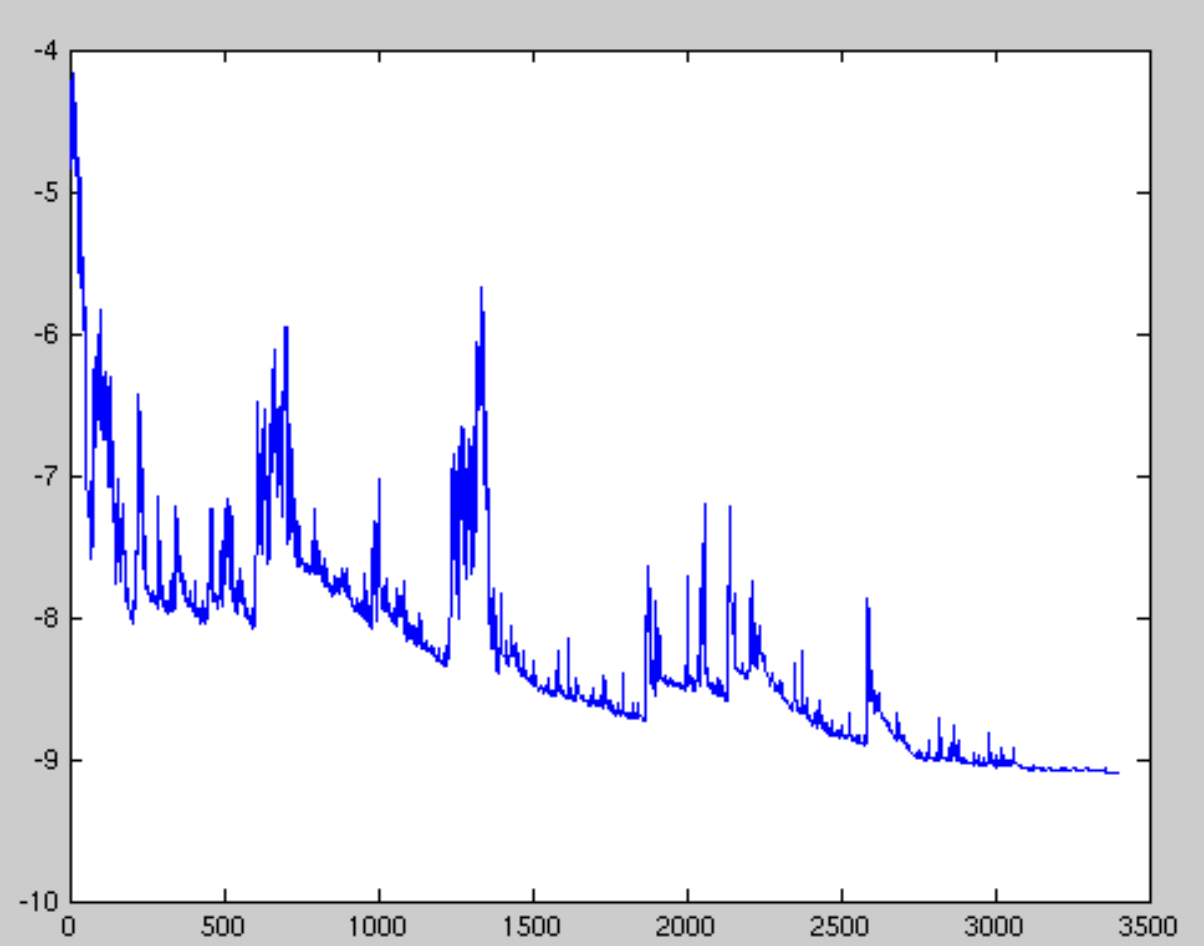
Stochastic Gradient Descent(SGD) on the other hand performs a parameter update for **each training example** .It is usually much faster technique.It performs one update at a time.

$\theta = \theta - \eta \cdot \nabla J(\theta; x(i); y(i))$, where {x(i) ,y(i)} are the training examples.

Now due to these ***frequent updates*** *,parameters updates have* ***high variance*** *and causes the* ***Loss function to fluctuate to different intensities****. This is actually a good thing because it helps us* ***discover new and possibly better local minima*** *, whereas Standard Gradient Descent will only converge to the minimum of the basin as mentioned above.*

But the problem with SGD is that due to the frequent updates and fluctuations it ultimately complicates the convergence to the exact minimum and will keep overshooting due to the frequent fluctuations .

Although, it has been shown that as we slowly decrease the learning rate-**η**, SGD shows the same convergence pattern as Standard gradient descent.



High Variance parameter updates for each training example cause the Loss function to fluctuate heavily due to which we might not get the minimum value of parameter which gives us least Loss value.

The problems of high variance parameter updates and unstable convergence can be rectified in another variant called *Mini-Batch Gradient Descent*.

2. Mini Batch Gradient Descent

An improvement to avoid all the problems and demerits of SGD and standard Gradient Descent would be to use **Mini Batch Gradient Descent** as it takes the best of both techniques and performs an update for every batch with n training examples in each batch.

The advantages of using Mini Batch Gradient Descent are —

- It Reduces the variance in the parameter updates , which can ultimately lead us to a much better and stable convergence.
- Can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.

- Commonly Mini-batch sizes Range from 50 to 256, but can vary as per the application and problem being solved.
- Mini-batch gradient descent is typically the algorithm of choice when training a neural network nowadays
- P.S —Actually the term SGD is used also when mini-batch gradient descent is used .

Challenges faced while using Gradient Descent and its variants —

- Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully *slow convergence i.e* will result in **small** baby steps towards finding optimal parameter values which minimize loss and finding that valley which directly affects the overall training time which gets too large. While a learning rate that is too **large** can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
- Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous ***sub-optimal local minima***. Actually, Difficulty arises in fact not from local minima but from ***saddle points***, i.e. *points where one dimension slopes up and another slopes down*. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

Optimizing the Gradient Descent

Now we will discuss about the various algorithms which are used to further optimize Gradient Descent.

Momentum

The high variance oscillations in SGD makes it hard to reach convergence , so a technique called ***Momentum*** was invented which ***accelerates*** **SGD** by navigating along the relevant direction and softens the oscillations in irrelevant directions.In other words all it does is adds a fraction '**γ**' of the update vector of the past step to the current update vector.

$$V(t)=\gamma V(t-1)+\eta \nabla J(\theta)$$.

and finally we update parameters by $\theta=\theta-V(t)$ **.**

The momentum term **γ** is usually set to 0.9 or a similar value.

Here the ***momentum*** *is same as the momentum in classical physics , as we throw a ball down a hill it gathers momentum and its velocity keeps on increasing.*

The same thing happens with our parameter updates —

- It leads to faster and stable convergence.
- Reduced Oscillations

The **momentum** term **γ** increases for dimensions whose gradients point in the same directions and *reduces updates* for dimensions whose gradients change directions. *This means it does parameter updates only for relevant examples. This reduces the unnecessary parameter updates which leads to faster and stable convergence and reduced oscillations*.

Nesterov accelerated gradient

A researcher named Yurii Nesterov saw a problem with Momentum —
However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.
What actually happens is that as we reach the minima i.e the lowest point on the curve ,the **momentum** is pretty high and it doesn't knows to ***slow*** *down* at that point due to the high momentum *which could cause it to miss the minima entirely and continue movie up. This problem was noticed by* **Yurii Nesterov.**
He published a research paper in 1983 which solved this problem of momentum and we now call this strategy ***Nestrov Accelerated Gradient.***
In the method he suggested we first make a big jump based on out previous momentum then calculate the Gradient and them make an correction which results in an parameter update. Now this anticipatory update prevents us to go too fast and not miss the minima and makes it more responsive to changes.

Nesterov accelerated gradient (NAG) is a way to give our momentum term this kind of prescience. We know that we will use our momentum term **γV(t−1)** to move the parameters **θ**. Computing **θ−γV(t−1)** thus gives us an *approximation of the next position of the parameters which gives us a rough idea where our parameters are going to be*. **We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:**
$V(t)=\gamma V(t-1)+\eta\nabla J(\ \theta-\gamma V(t-1)\ )$ and then update the parameters using **$\theta=\theta-V(t)$.**
One can refer more on ***NAGs*** here http://cs231n.github.io/neural-networks-3/.
Now that we are able to adapt our updates to the slope of our error function and speed up SGD in turn, we *would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.*

Adagrad

It simply allows the learning Rate -**η** to **adapt** based on the parameters. So it makes big updates for infrequent parameters and small updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data.

It uses a different learning Rate for every parameter θ at a time step based on the past gradients which were computed for that parameter.

Previously, we performed an update for all parameters **θ** at once as every parameter **θ(i)** used the same learning rate **η**. As ***Adagrad*** uses a different learning rate for every parameter **θ(i)** at every time step **t**, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we set **g(t,i)** to be the ***gradient of the loss function*** w.r.t. to the parameter **θ(i)** at time step **t .**

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

The formula for Parameter updates

***Adagrad* modifies the general learning rate η at each time step t for every parameter θ(i) based on the past gradients that have been computed for θ(i).**

*The main benefit of **Adagrad** is that we don't need to manually tune the learning Rate. Most implementations use a default value of 0.01 and leave it at that.*

Disadvantage —

- Its main weakness is that its learning rate-**η** is always Decreasing and decaying.

This happens due to the accumulation of each squared Gradients in the denominator , since every added term is positive.The accumulated sum keeps growing during training. This in turn causes the *learning rate to shrink and eventually become so small,* **that the model just stops learning entirely and stops acquiring new additional knowledge. Because we know that as the learning rate gets smaller and smaller the ability of the Model to learn fastly decreases and which gives very slow convergence and takes very long to train and learn i.e learning speed suffers and decreases.**

This problem of **Decaying learning Rate** is Rectified in another algorithm called **AdaDelta.**

AdaDelta

It is an extension of **AdaGrad** which tends to remove the *decaying learning Rate* problem of it. Instead of accumulating all previous squared gradients, ***Adadelta*** limits the window of accumulated past gradients to some fixed size **w**.

Instead of inefficiently storing **w** previous squared gradients, the sum of gradients is recursively defined as a *decaying* ***mean*** of all past squared gradients. The running average **$E[g^2](t)$** at time step **t** then depends (as a fraction **γ** similarly to the Momentum term) *only on the previous average and the current gradient.*

$E[g^2](t)=\gamma.E[g^2](t-1)+(1-\gamma).g^2(t)$, We set **γ** to a similar value as the momentum term, around 0.9.

$\Delta\theta(t)=-\eta \cdot g(t,i)$.

$\theta(t+1)=\theta(t)+\Delta\theta(t)$.

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t.$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t.$$

The final formula for Parameter Updates

Another thing with AdaDelta is that we don't even need to set a default learning Rate .

What Improvements we have done so far —

- We are calculating *different learning Rates* for each parameter.
- We are also calculating *momentum*.
- Preventing **Vanishing(decaying) learning Rates**.

What more improvements can we do ?

Since we are calculating individual ***learning rates*** for each parameter , why not calculate individual **momentum** changes for each parameter and store them separately. This is where a new modified technique and improvement comes into play called as **Adam.**

Adam
Adam stands for **Adaptive Moment Estimation.** Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like **AdaDelta** ,***Adam*** *also keeps an exponentially decaying average of past gradients M****(t)****, similar to momentum*:
M(t) and V(t) are values of the first moment which is the ***Mean*** and the second moment which is the ***uncentered variance*** of the *gradients* respectively.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}.$$

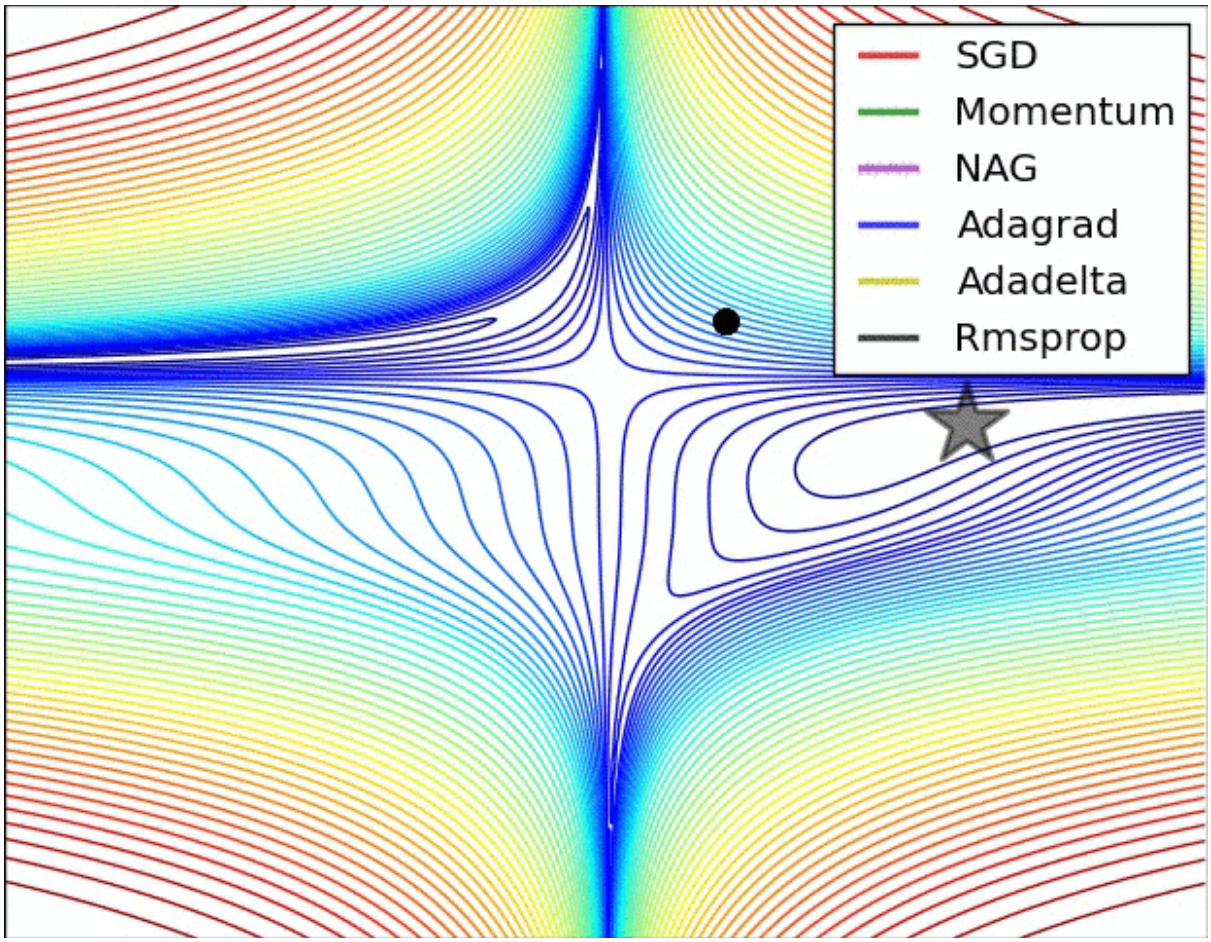
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

The formulas for the first Moment(mean) and the second moment (the variance) of the Gradients
Then the final formula for the Parameter update is —

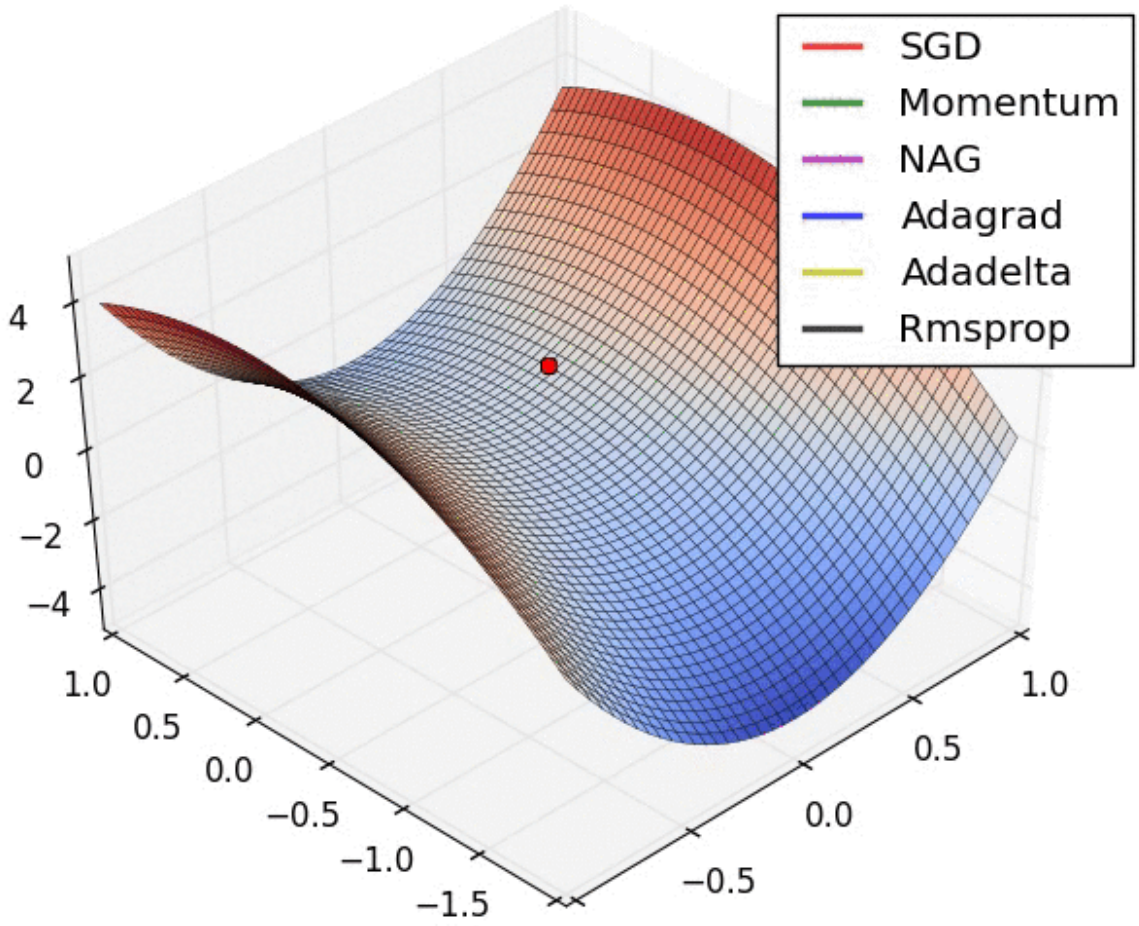
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$

The values for β1 is 0.9 , 0.999 for β2, and (10 x exp(-8)) for **ϵ**.
Adam *works well in practice and compares favorably to other adaptive learning-method algorithms as it converges very fast and the learning speed of the Model is quiet Fast and efficient and also it rectifies every problem that is faced in other optimization techniques such as* ***vanishing Learning rate , slow convergence or High variance in the parameter updates which leads to fluctuating Loss function***

Visualization of the Optimization Algorithms



SGD optimization on loss surface contours



SGD optimization on saddle point

From the above images one can see that The ***Adaptive algorithms converge*** *very fast and quickly find the right direction in which parameter updates should occur. Whereas standard* ***SGD , NAG and momentum*** *techniques are very slow and could not find the right direction.*

Conclusion

Which optimizer should we use?

The question was to choose the best optimizer for our Neural Network Model in order to converge fast and to learn properly and tune the internal parameters so as to minimize the Loss function .

Adam *works well in practice and outperforms other Adaptive techniques.*

If your input data is sparse then methods such as **SGD,NAG and momentum** are inferior and perform poorly. **For sparse data sets one should use one of the *adaptive learning-rate* methods**. An additional benefit is that we won't need to adjust the learning rate but likely achieve the best results with the default value.

If one wants fast convergence and train a deep Neural Network Model or a highly complex Neural Network then **Adam or any other Adaptive learning rate techniques** should be used because they outperforms every other optimization algorithms.