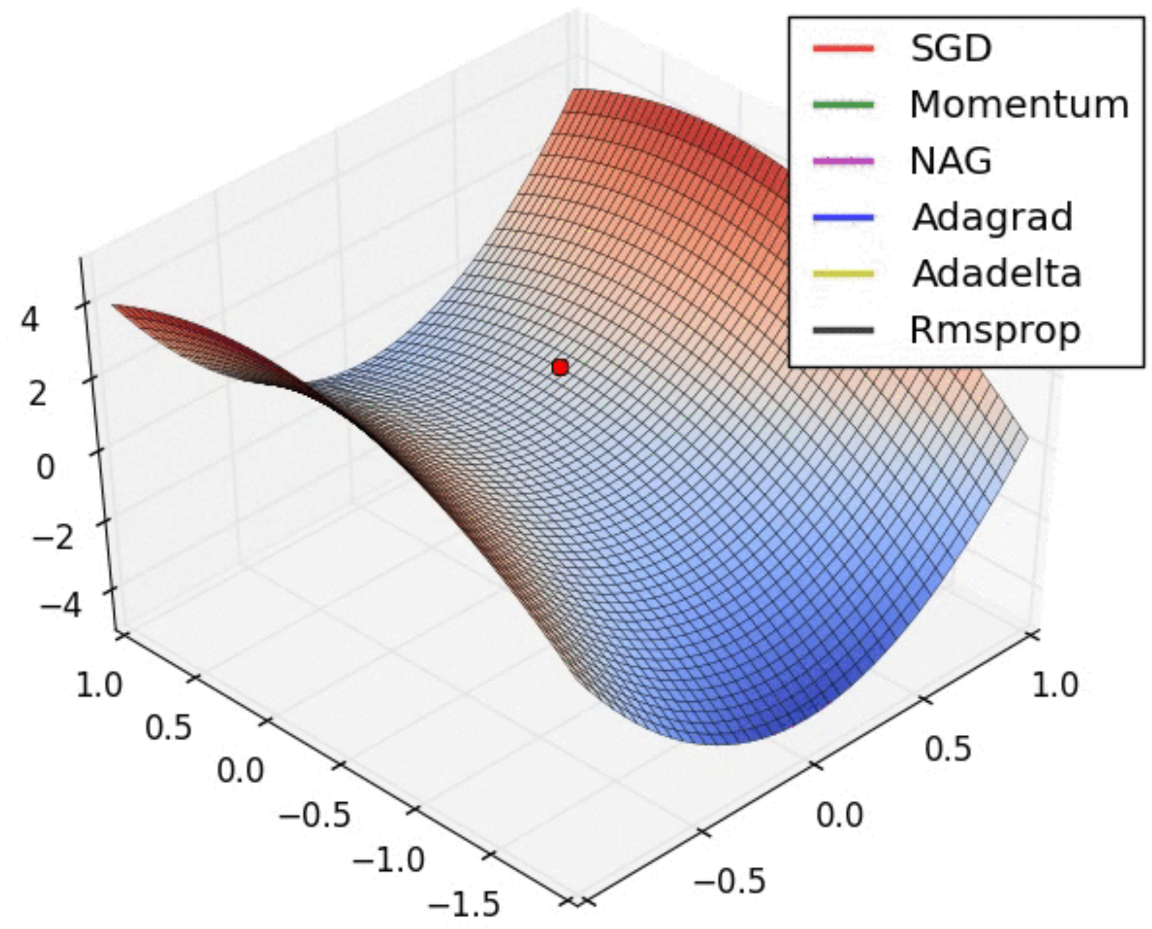
Optimization Algorithms in Neural Networks



In deep learning, we have the concept of loss, which tells us how poorly the model is performing at that current instant. We use this loss to train our network such that it performs better by essentially trying 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.

Optimizers are algorithms or methods used to change the attributes of the neural network (such as weights and the learning rate) to reduce the losses. Optimizers are used to solve optimization problems by minimizing the function.

How do Optimizer’s work?

For a useful mental model, you can think of a hiker trying to get down a mountain with a blindfold on. It’s impossible to know which direction to go in, but there’s one thing she can know: if she’s going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she’ll reach the base.

Similarly, it’s impossible to know what your model’s weights should be right from the start. But with some trial and error based on the loss function (whether the hiker is descending), you can end up getting there eventually.

How you change the weights or learning rate of your neural network to reduce the loss, is defined by the optimizers you use. Optimization algorithms are responsible for reducing the losses and to provide the most accurate results possible.

Various optimizers have been discovered, each having its advantages and disadvantages.

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini Batch Stochastic Gradient Descent (MB-SGD)
4. SGD with momentum
5. Nesterov Accelerated Gradient (NAG)
6. Adaptive Gradient (AdaGrad)
7. AdaDelta
8. RMSprop
9. Adam

Gradient Descent

Gradient descent is based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum.
You start by defining the initial parameter's values and from there gradient descent uses calculus to iteratively adjust the values, so they minimize the given cost-function.

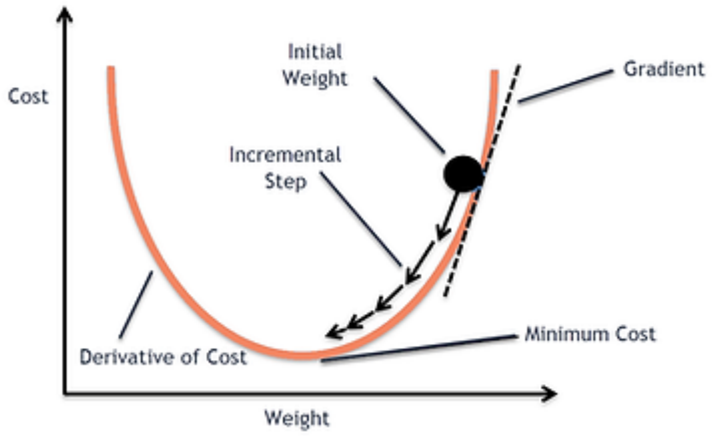
The weight is initialized using some type of initialization strategy (usually a random number) and is updated with each epoch according to the update equation.

$$\text{Repeat until convergence } \{$$

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\}$$

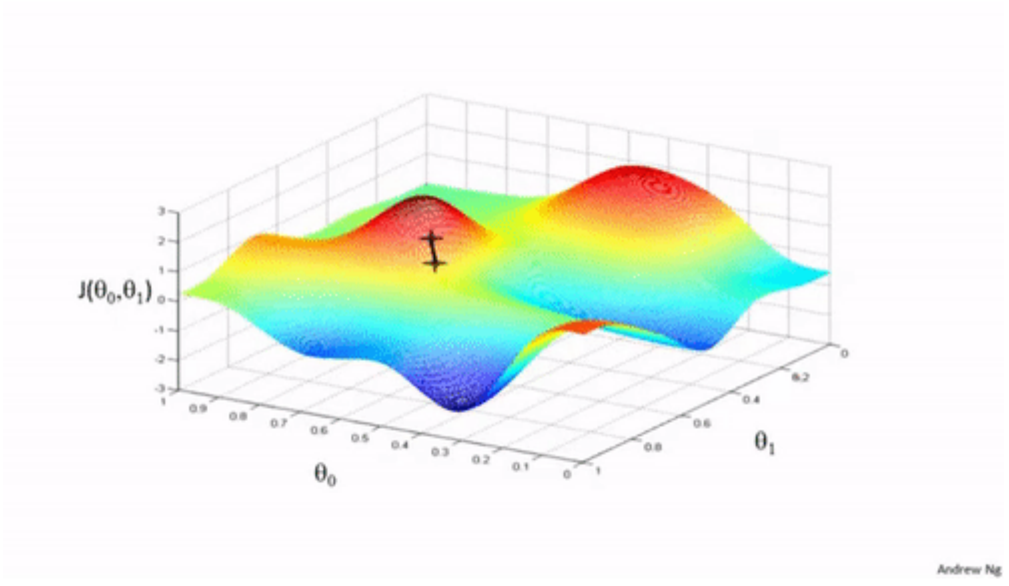
The above equation computes the gradient of the cost function J(θ) w.r.t. to the parameters/weights θ for the entire training dataset:



Our aim is to get to the bottom of our graph (Cost vs weights), or to a point where we can no longer move downhill–a local minimum.

Okay now, so what is a Gradient?

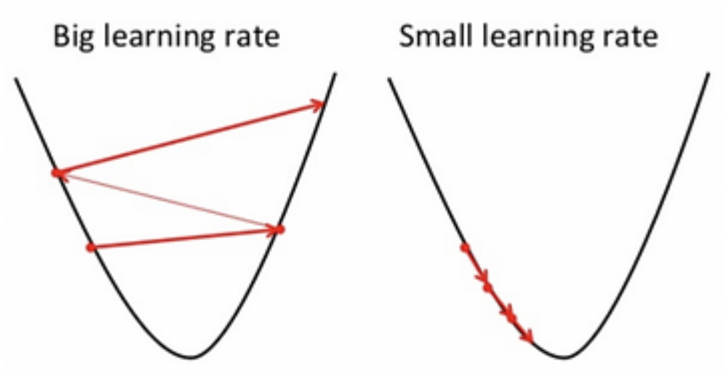
"A gradient measure's how much the output of a function changes if you change the inputs a little bit." — Lex Fridman (MIT)



Importance of Learning rate

How big the steps that the gradient descent takes in the direction of the local minimum are determined by the learning rate, which in turn determines how fast or slow we will move towards the optimal weights.

For gradient descent to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high. This is important because if the steps are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (see left image below). If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while (see the right image).



So, the learning rate should never be too high or too low. You can check if you're learning rate is doing well by plotting it on a graph.

In code, gradient descent looks something like this:

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

Advantages:

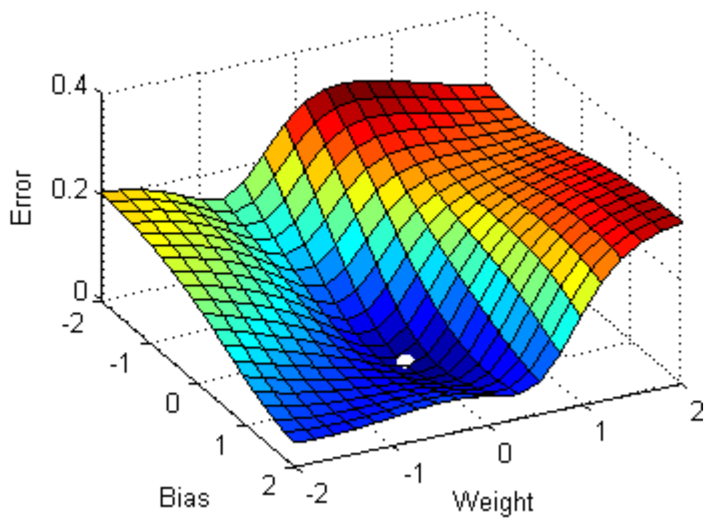
1. Easy computation.
2. Easy to implement.
3. Easy to understand.

Disadvantages:

1. May trap at local minima.
2. Weights are changed after calculating the gradient on the whole dataset. So, if the dataset is too large then this may take years to converge to the minima.
3. Requires large memory to calculate the gradient on the whole dataset.

Stochastic Gradient Descent (SGD)

SGD algorithm is an extension of the Gradient Descent, and it overcomes some of the disadvantages of the GD algorithm.
Gradient Descent has a disadvantage that it requires a lot of memory to load the entire dataset in order to compute the derivative of the loss function. In the SGD algorithm derivative is computed taking one point at a time



SGD performs a parameter update for each training example x(i) and label y(i):

$$\theta = \theta - \alpha \cdot \partial(J(\theta;x(i),y(i)))/\partial\theta$$

where {x(i) ,y(i)} are the training examples.

To make the training even faster we take a Gradient Descent step for each training example. Let's see what the implications would be in the image below.

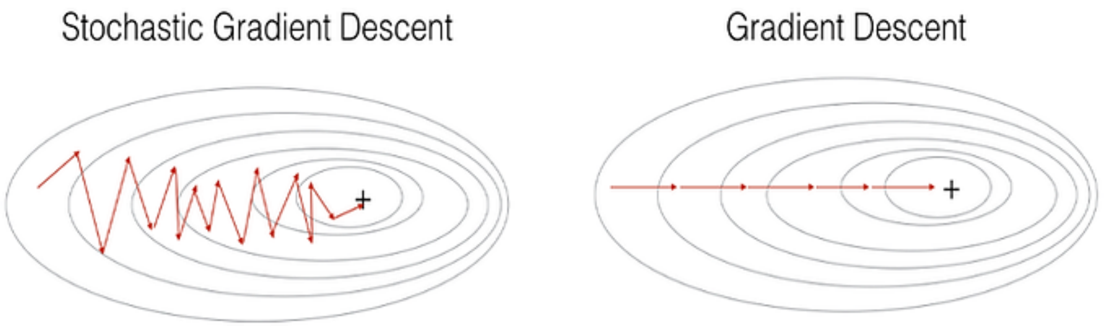


Figure 1 : SGD vs GD
"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

1. On the left, we have Stochastic Gradient Descent (where m=1 per step) we take a Gradient Descent step for each example and on the right is Gradient Descent (1 step per entire training set).
2. SGD seems to be quite noisy, at the same time it is much faster but may not converge to a minimum.
3. Typically, to get the best out of both worlds we use Mini-batch gradient descent (MGD) which looks at a smaller number of training set examples at once to help (usually power of 2 - 2^6 etc.).
4. Mini-batch Gradient Descent is relatively more stable than Stochastic Gradient Descent (SGD) but does have oscillations as gradient steps are being taken in the direction of a sample of the training set and not the entire set as in BGD.

It is can be observed that in SGD the updates take more iterations in comparison to gradient descent to reach minima. On the right, the Gradient Descent takes fewer steps to reach minima but the SGD algorithm is noisier and takes more iterations.
Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example.

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
      params_grad = evaluate_gradient(loss_function, example, params)
       params = params - learning_rate * params_grad
```

Advantage:

Memory requirement is less compared to the GD algorithm as the derivative is computed taking only 1 point at a time.

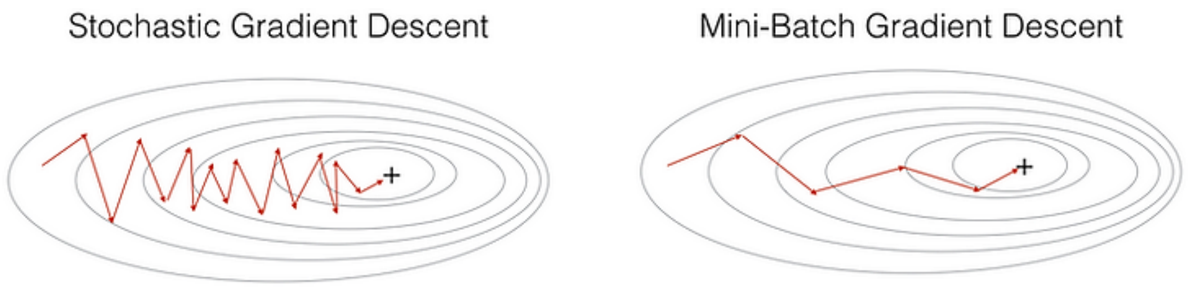
Disadvantages:

- The time required to complete 1 epoch is large compared to the GD algorithm.
- Takes a long time to converge.
- May stick at local minima.

Mini Batch Stochastic Gradient Descent (MB-SGD)

MB-SGD algorithm is an extension of the SGD algorithm, and it overcomes the problem of large time complexity. The MB-SGD algorithm takes a batch of points or subset of points from the dataset to compute the derivate.

It can be observed that the derivative of the loss function for MB-SGD is almost the same as a derivate of the loss function for GD after n iterations, however the number of iterations to achieve minima is large for MB-SGD compared to GD and the cost of computation is also large.



The updated weight is dependent on the derivate of loss for a batch of points. The updates in the case of MB-SGD are noisier because the derivative is not always towards minima.
MB-SGD divides the dataset into various batches and after every batch, the parameters are updated.

$$\theta = \theta - \alpha \cdot \partial(J(\theta;B(i)))/\partial\theta$$

where **{B(i)}** are the batches of training examples.

In code, instead of iterating over examples, we now iterate over mini-batches of a determined size eg. 50:

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for batch in get_batches(data, batch_size=50):
        params_grad = evaluate_gradient(loss_function, batch,params)
        params = params - learning_rate * params_grad
```

Advantages:

Less time complexity to converge compared to standard SGD algorithm.

Disadvantages:

- The update of MB-SGD is noisier compared to the update of the GD algorithm.
- Take a longer time to converge than the GD algorithm.
- May get stuck at local minima.

SGD with momentum

A major disadvantage of the MB-SGD algorithm is that updates of weight are noisy. SGD with momentum overcomes this disadvantage by denoising the gradients.

The idea is to denoise derivative using an exponential weighting average so that more weight is assigned to the update happening at the present than to the previous update.

It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by **'γ'**.

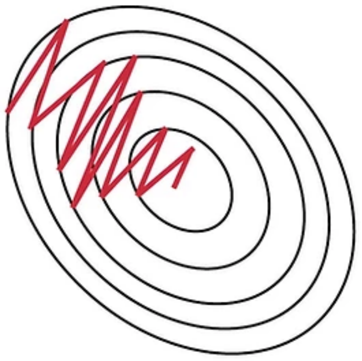
$$V(t) = \gamma.V(t-1) + \alpha.\partial(J(\theta))/\partial\theta$$

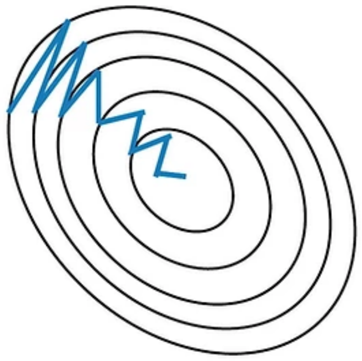
Now, the weights are updated by $\theta = \theta - V(t)$.

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

Momentum at time 't' is computed using all previous updates giving more weight to the immediate update compared to the previous update. This leads to speed up the convergence.

Essentially, when using momentum, we are trying to push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma<1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.


Stochastic Gradient Descent **withhout** Momentum


Stochastic Gradient Descent **with** Momentum

The diagram above demonstrates how SGD with momentum reduces noise in the gradients and converges faster as compared to SGD.

Advantages:

- Has all advantages of the SGD algorithm.
- Converges faster than the GD algorithm.

Disadvantages:

We need to compute one more variable for each update.

Nesterov Accelerated Gradient (NAG)

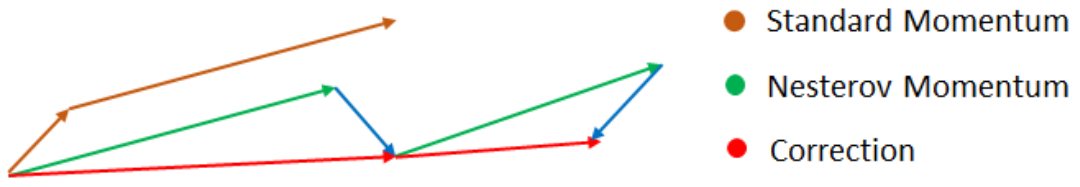
The idea of the NAG algorithm is very similar to SGD with momentum with a slight variant. In the case of SGD with momentum, the momentum and gradient are computed on the previous updated weight.

Momentum may be a good method to use but if the momentum is too high the algorithm may miss the local minima and may continue to rise up. So, to resolve this issue the NAG algorithm was developed. It is a " look ahead" method. We know we'll be using **$\gamma.V(t-1)$** for modifying the weights so, **$\theta-\gamma V(t-1)$** approximately tells us the future location. Now, we'll calculate the cost based on this future parameter rather than the current one.

$$\mathbf{V(t) = \gamma.V(t-1) + \alpha.\ \partial(J(\theta - \gamma V(t-1)))/\partial\theta}$$

and then update the parameters using **$\theta = \theta - V(t)$**

Again, we set the momentum term γ to a value of around 0.9. While Momentum first computes the current gradient (small brown vector below) and then takes a big jump in the direction of the updated accumulated gradient (big brown vector), NAG first makes a big jump in the direction of the previously accumulated gradient (green vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (red vector). This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs in tests.



Both NAG and SGD with momentum algorithms work equally well and share the same advantages and disadvantages.

Adaptive Gradient Descent(AdaGrad)

For all the previously discussed algorithms the learning rate remains constant. So the key idea of AdaGrad is to have an adaptive learning rate for each of the weights.

It performs smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequently occurring features.

For brevity, we use gt to denote the gradient at time step t. **gt,i** is then the partial derivative of the objective function w.r.t. to the parameter **θi** at time step **t, η** is the learning rate and ∇θ is the partial derivative of loss function **J(θi)**

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}).$$

The SGD update for every parameter θ_i at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θi based on the past gradients for θi:

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

where **Gt** is the sum of the squares of the past gradients w.r.t to all parameters **θ.**

The benefit of AdaGrad is that it eliminates the need to manually tune the learning rate; most leave it at a default value of 0.01.
Its main weakness is the accumulation of the squared gradients(**Gt**) in the denominator. Since every added term is positive, the accumulated sum keeps growing during training, causing the learning

rate to shrink, and becoming infinitesimally small and further resulting in a vanishing gradient problem.

Advantage:

No need to update the learning rate manually as it changes adaptively with iterations.

Disadvantage:

As the number of iteration becomes very large learning rate decreases to a very small number which leads to slow convergence.

AdaDelta

The problem with the previous algorithm AdaGrad was learning rate becomes very small with a large number of iterations which leads to slow convergence. To avoid this, the AdaDelta algorithm relies upon an exponentially decaying average.

Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates finished.
Compared to Adagrad, in the original version of Adadelta, you don't have to set an initial learning rate.
Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average **E[g2]t** at time step t then depends only on the previous average and current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

Usually γ is set to around 0.9. Rewriting SGD updates in terms of the parameter update vector:

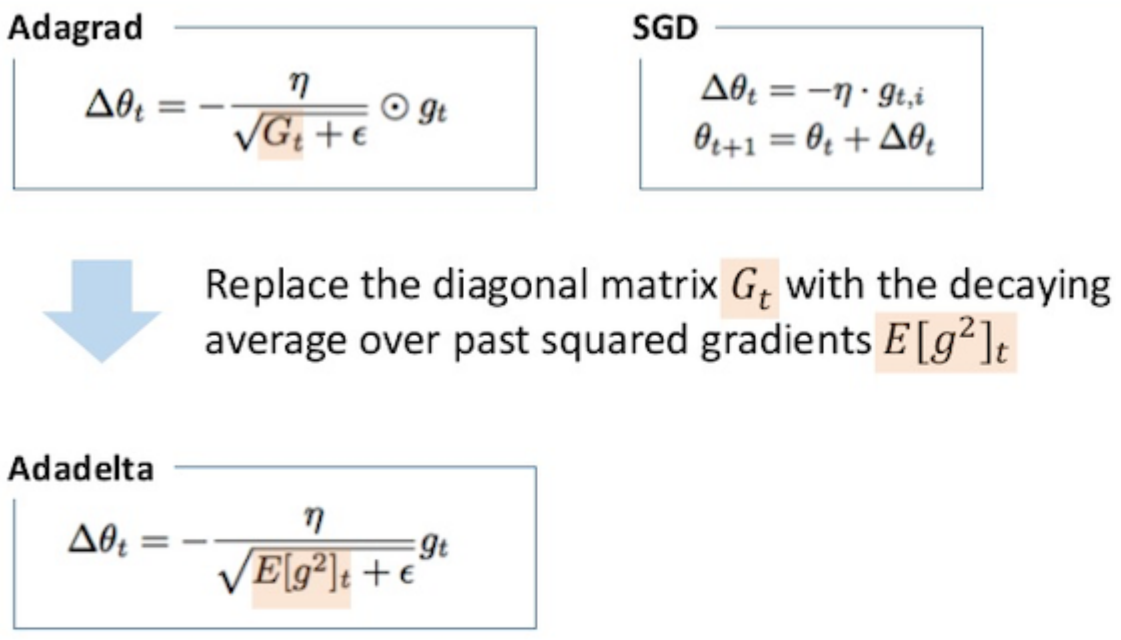
$$\Delta\theta_t = -\eta \cdot g_{t,i}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

AdaDelta takes the form:

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

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.


Adagrad
$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$
SGD
$\Delta\theta_t = -\eta \cdot g_{t,i}$
$\theta_{t+1} = \theta_t + \Delta\theta_t$
Replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$
Adadelta
$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$

RMSprop

RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

RMSprop also divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests **γ** be set to 0.9, while a good default value for the learning rate **η** is 0.001.

RMSprop and Adadelta were both developed independently around the same time due to the need to resolve Adagrad's radically diminishing learning rates

Adaptive Moment Estimation (Adam)

Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum.

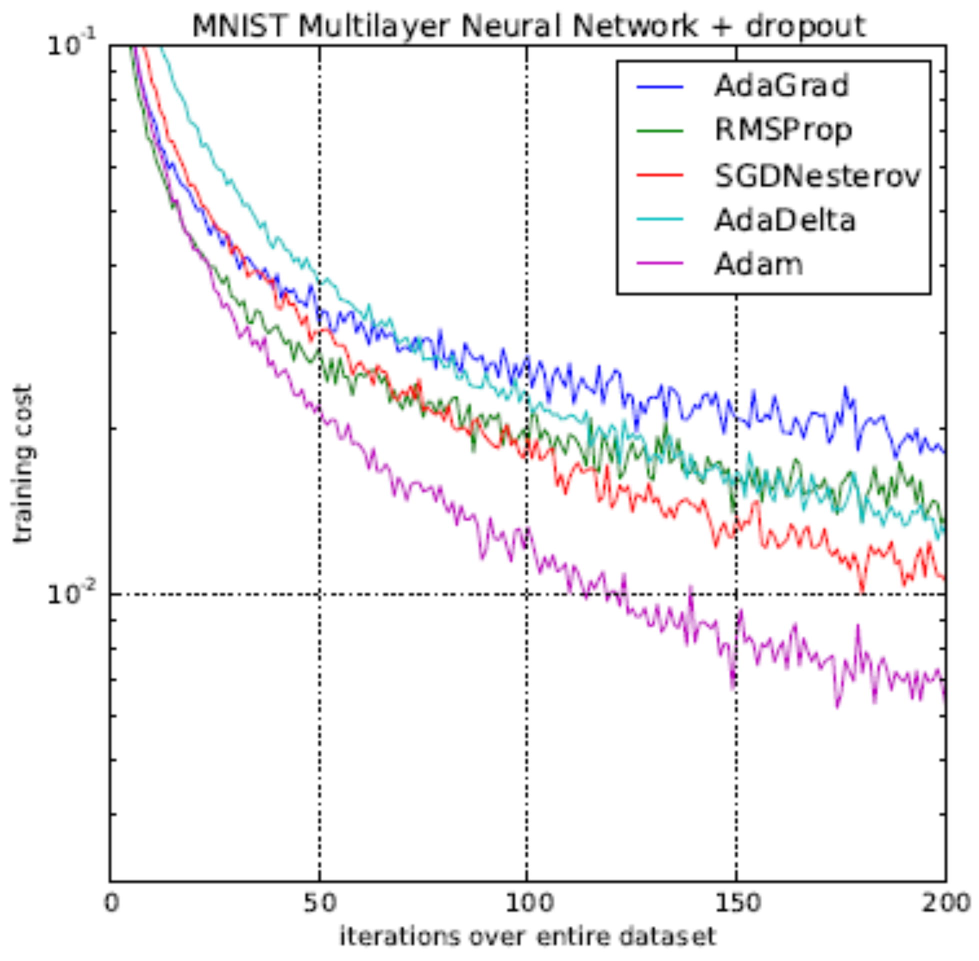
Adam computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients **vt** like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients **mt**, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.

Hyper-parameters **β1, β2 ∈ [0, 1)** control the exponential decay rates of these moving averages. We compute the decaying averages of past and past squared gradients **mt** and **vt** respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

mt and **vt** are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.

When to choose which algorithm?



As you can observe, the training cost in the case of Adam is the least.
Now observe the animation at the beginning of these notes and consider the following points:

1. It is observed that the SGD algorithm (red) is stuck at a saddle point. So SGD algorithm can only be used for shallow networks.
2. All the other algorithms except SGD finally converges one after the other, AdaDelta being the fastest followed by the momentum algorithms.
3. AdaGrad and AdaDelta algorithm can be used for sparse data.
4. Momentum and NAG work well for most cases but are slower.
5. Adam is considered the most efficent algorithm amongst all the algorithms discussed