# Unit 4
# Beyond The Gradient Descent

**Prepared By: Arjun Singh Saud**
**Asst. Prof. CDCSIT**

# Challenges with Gradient Descent

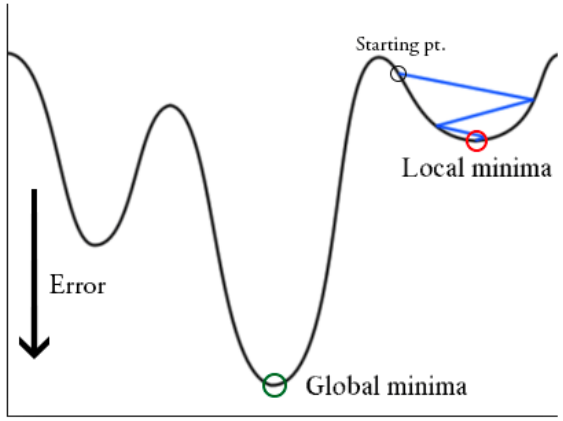
## Local Minima in Error Surface

- Local minima is the point mimic the shape of a global minimum, where the slope of the cost function increases on either side of the current point.
- For convex error surfaces, gradient descent can find the global minimum with ease, but with non-convex error surfaces, gradient descent struggles to find the global minimum.
- At local minima slope of cost function is near close to zero and the model stops learning. Hence the gradient descent faces difficulty to skip from it.

# Challenges with Gradient Descent

## Local Minima in Error Surface

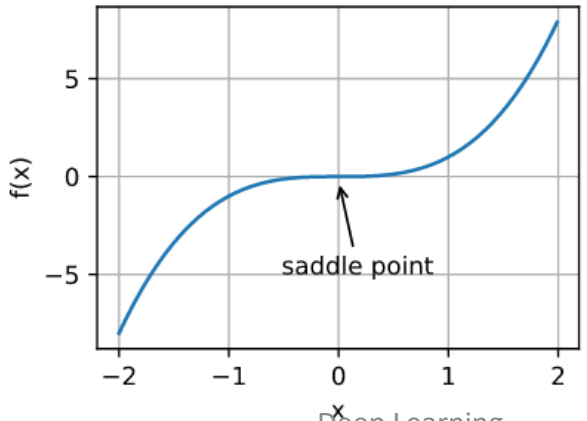
- There can be large number of local minima in error surface and the convergence point of gradient descent depends upon initialization point and learning rate.



# Challenges with Gradient Descent

**Flat Regions in Error Surface (Saddle Point)**

- Saddle point on a surface of error function is the point where seeing from one dimension, that point seems minimum while from other dimension it seems as a maximum point.
- Slope of the error surface is also near to zero at saddle points and hence the model stops learning and may converge there.



# Challenges with Gradient Descent

**Vanishing and Exploding Gradient**

- When training a deep neural network with gradient descent and backpropagation, we calculate the partial derivatives by moving across the network from the final output layer to the initial layer.
- With the chain rule, layers that are deeper in the network go through continuous matrix multiplications to compute their derivatives.

# Challenges with Gradient Descent

**Vanishing and Exploding Gradient**

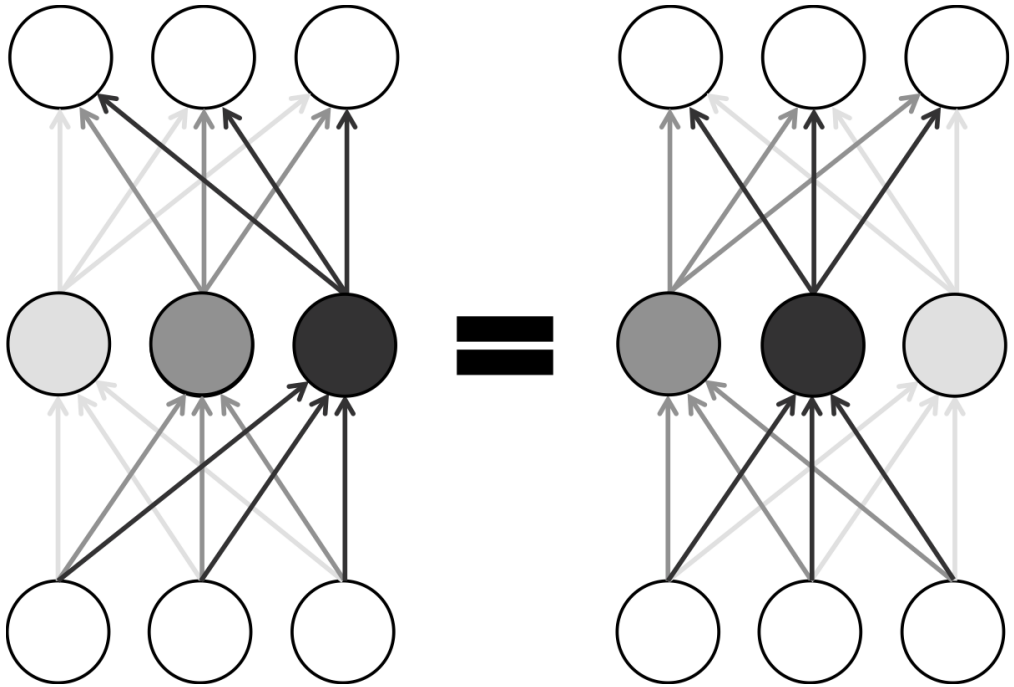
- In a network of n hidden layers, n derivatives will be multiplied together. If the derivatives are large, then the gradient will increase exponentially as we propagate down the model until they eventually explode, and this is what we call the problem of exploding gradient.
- Alternatively, if the derivatives are small, then the gradient will decrease exponentially as we propagate through the model until it eventually vanishes, and this is the vanishing gradient problem.

# Challenges with Gradient Descent

## Model Non-Identifiability

- A model is identifiable, if with sufficiently large training set, we can rule out one optimal parameter setting of the model.
- Non-identifiability is opposite when there exists multiple equivalent models that give the same outputs.
- There are many causes of model non-identifiability. For example, weight symmetry is one cause of model non-identifiability, which happens when we rearrange hidden units and their incoming and outgoing weight vectors accordingly.

# Challenges with Gradient Descent



# Challenges with Gradient Descent

- Thus, we can get technically a different model that delivers the same output.
- This implies that model non-identifiablity cause many equivalent model configurations and hence potentially many local minima.

# How Pesky Are Spurious Local Minima in Deep Networks?

- All local minima that arise because of the non-identifiability of deep neural networks are not inherently problematic.
- This is because all nonidentifiable configurations learn equally from the training data and will have identical behavior during generalization to unseen examples.
- Instead, local minima are only problematic when they are *spurious*. A spurious local minimum corresponds to a configuration of weights in a neural network that incurs a higher error than the configuration at the global minimum.

# How Pesky Are Spurious Local Minima in Deep Networks?

- If these kinds of local minima are common, we quickly run into significant problems while using gradient based optimization methods because it only take into account local structure.
- Experiments have shown that local minima introduced by model non-identifiability are not truly annoying to gradient descent.
- Rather, finding the appropriate direction to move in is the major source of trouble in gradient descent.

# How Pesky Are Spurious Local Minima in Deep Networks?

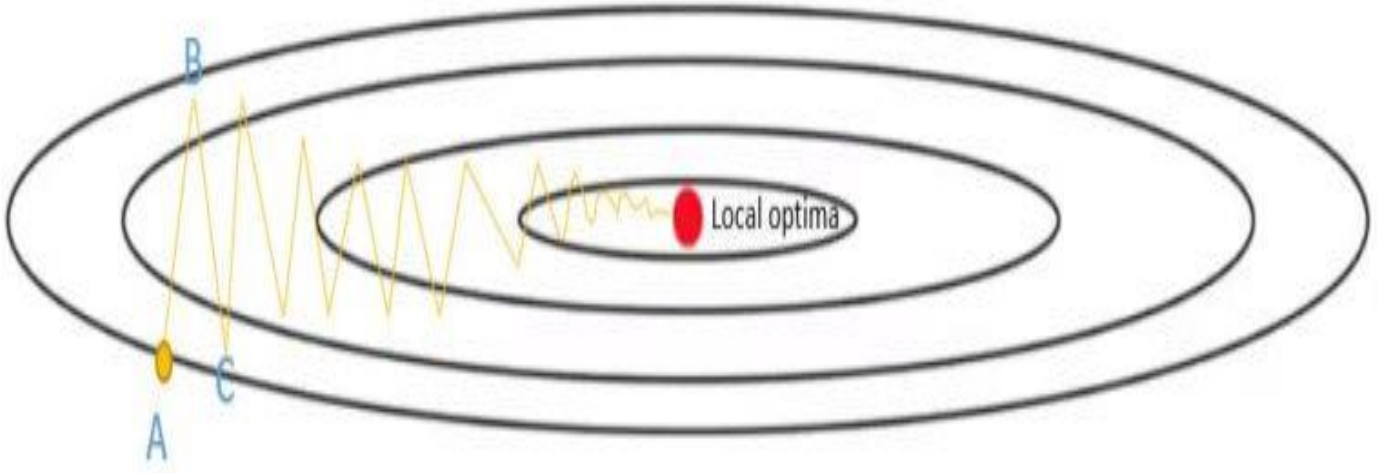
- If these kinds of local minima are common, we quickly run into significant problems while using gradient based optimization methods because it only take into account local structure.
- Experiments have shown that local minima introduced by model non-identifiability are not truly annoying to gradient descent.
- Rather, finding the appropriate direction to move in is the major source of trouble in gradient descent.

# Momentum Based Optimizations

- Mini-batch gradient descent makes a parameter update with just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will oscillate toward convergence.
- Gradient Descent with Momentum considers the past gradients to smooth out the update. It computes an exponentially weighted average of gradients, and then use that gradient to update weights instead. It works faster than the standard gradient descent algorithm.

# Momentum Based Optimizations

- Consider an example where we are trying to optimize a cost function which has contours like below and the red dot denotes the position of the local optima (minimum).



# Momentum Based Optimizations

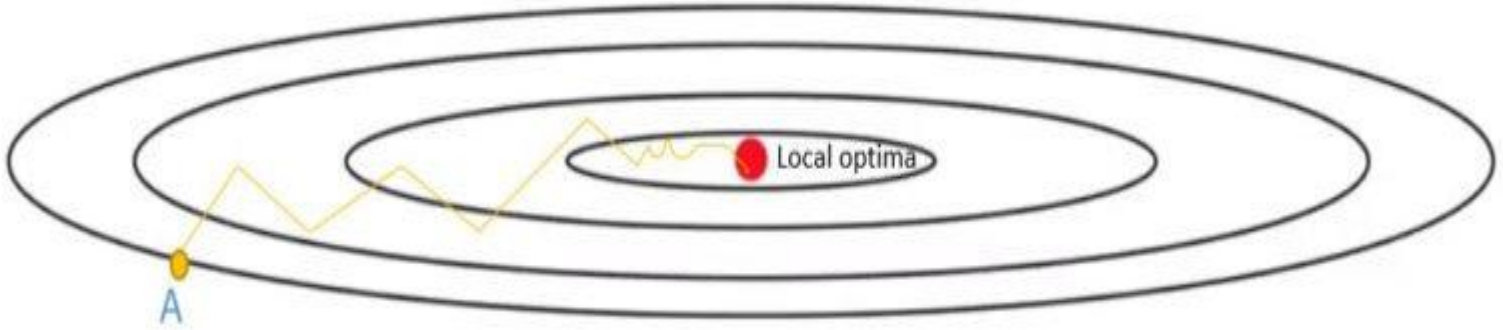
- With each iteration of gradient descent, we move towards the local optima with up and down oscillations. If we use larger learning rate then the vertical oscillation will have higher magnitude.
- So, this vertical oscillation slows down our gradient descent and prevents us from using a much larger learning rate.
- With each iteration of gradient descent, we move towards the local optima with up and down oscillations. If we use larger learning rate then the vertical oscillation will have higher magnitude.

# Momentum Based Optimizations

- So, this vertical oscillation slows down our gradient descent and prevents us from using a much larger learning rate.
- By using the exponentially weighted average values of dw and db, we tend to average out the oscillations in the vertical direction closer to zero as they are in both directions (positive and negative).
- Whereas, on the horizontal direction, all the derivatives are pointing to the right of the horizontal direction, so the average in the horizontal direction will still be pretty big.

# Momentum Based Optimizations

- It allows our algorithm to take more straight forwards path towards local optima and damp out vertical oscillations. Due to this reason, the algorithm will end up at local optima with a few iterations.



# Momentum Based Optimizations

- It computes an exponentially weighted average of gradients and then uses this gradient to update the weights as below.

$$v_t = \beta v_{t-1} + (1-\beta) dw_t$$

$$w_{t+1} = w_t + \alpha v_t$$

*Where beta 'β' is Hyperparameter called momentum and ranges from 0 to 1.*

- Sometimes above equation is also written as below. In such case it will be enough to scale value of α by (1-β),

$$v_t = \beta v_{t-1} + dw_t$$

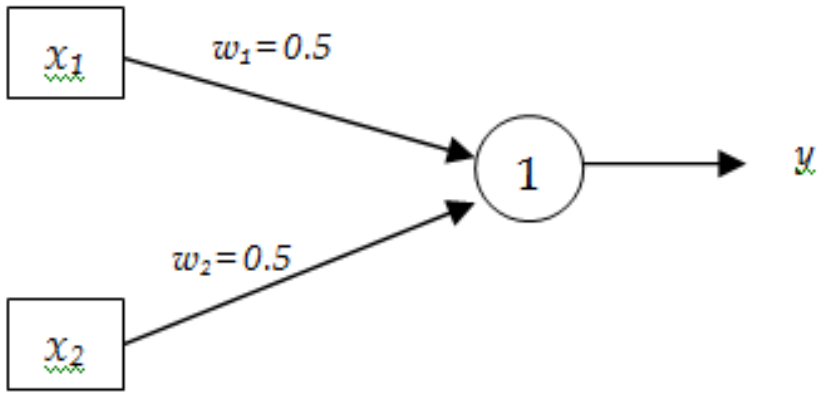
$$w_{t+1} = w_t + \alpha v_t$$

# Momentum Based Optimizations

**Example**

- Consider following simple ANN with logistic activation function. Assume that $\alpha = 0.2 \quad \beta = 0.8$. Calculate weight updates for the given training Sample using Momentum.

| $x_1$ | $x_2$ | $t$ |
|---|---|---|
| 1 | 0.6 | 0.8 |
| 0.5 | 0.5 | 0.6 |
| | | |



# Momentum Based Optimizations

**Solution**

For Training example:

(1,0.6,0.8)

$$y_{in} = 0.5*1 + 0.5*0.6 = 0.8$$

$$y = \frac{1}{1+\exp(-y_{in})} = 0.6899$$

$$(dw_1)_1 = (t-y)\times y \times (1-y) \times x_1 = 0.02354$$

$$(dw_2)_1 = (t-y)\times y \times (1-y) \times x_2 = 0.01412$$

$$(v_1)_1 = 0.8 \times (v_1)_0 + 0.2 \times 0.02354 = 0.004708$$

$$(v_2)_1 = 0.8 \times (v_2)_0 + 0.2 \times 0.01412 = 0.002824$$

$$(w_1)_2 = (w_1)_1 + 0.2 \times (v_1)_1 = 0.5 + 0.2 \times 0.004708 = 0.5009$$

$$(w_2)_2 = (w_2)_1 + 0.2 \times (v_2)_1 = 0.5 + 0.2 \times 0.002824 = 0.5006$$

# Momentum Based Optimizations

**Solution**

For Training example:

(0.5,0.5,0.6)

$$y_{in} = 0.5009*1 + 0.5006*0.6 = 0.7512$$

$$y = \frac{1}{1+\exp(-y_{in})} = 0.67944$$

$$(dw_1)_2 = (t-y)\times y\times(1-y)\times x_1 = ?$$

$$(dw_2)_2 = (t-y)\times y\times(1-y)\times x_2 = ?$$

$$(v_1)_2 = 0.8\times(v_1)_1 + 0.2\times(dw_1)_2 = ?$$

$$(v_2)_2 = 0.8\times(v_2)_1 + 0.2\times(dw_2)_2 = ?$$

$$(w_1)_3 = (w_1)_2 + 0.2\times(v_1)_2 = ?$$

$$(w_2)_2 = (w_2)_2 + 0.2\times(v_2)_2 = ?$$

# Bias Correction of Momentum

- When $V_0$ *is* initialized to 0, then $V_1$ will be calculated as:

$$v_1 = 0.8v_0 + (1-0.8)x_1 = 0.2x_1$$

- Obviously, this is a rather inaccurate (too small) averaging for the weighted average of this day. Similarly, $V_2$ will be calculated as:

$$v_2 = 0.8v_1 + (1-0.8)x_2 = 0.8 \times 0.2 \times x_1 + 0.2x_2 = 0.16x_1 + 0.2x_2$$

- Again, this value is also inaccurate (too small).
- In order to correct this inaccuracy, you should use the formula:

$$v_t = \frac{v_t}{1-\beta^t} \quad where\ 1-\beta^t\ is\ Bias\ Correction\ Factor$$

# Bias Correction of Momentum

- Now $v_1$ and $v_2$ will be,

$$v_1 = \frac{v_1}{1-\beta^1} = \frac{v_1}{1-0.8^1} = \frac{v_1}{0.2} \qquad v_2 = \frac{v_2}{1-\beta^2} = \frac{v_2}{1-0.8^2} = \frac{v_2}{0.36}$$

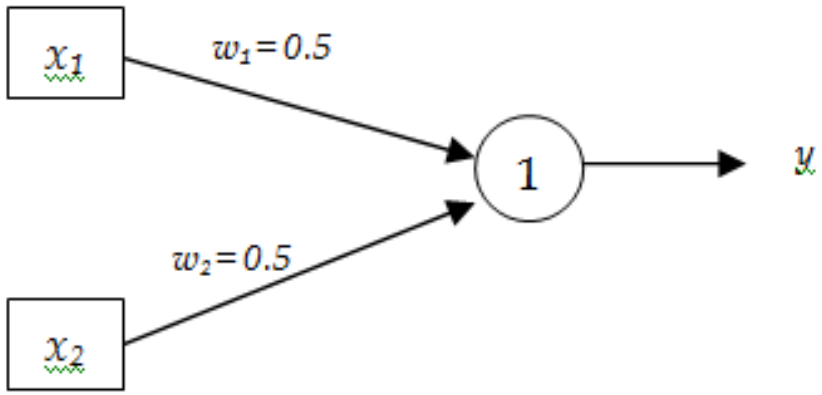
- When t is sufficiently large, bias correction factor has no effect because $1/1-\beta^t \approx 1$. Hence It only jacks up the initial values.

# Momentum Based Optimizations

**Example**

- Consider following simple ANN with logistic activation function. Assume that $\alpha = 0.2 \quad \beta = 0.8$. Calculate weight updates for the given training Sample using Momentum. Use Bias correction.

| $x_1$ | $x_2$ | $t$ |
|---|---|---|
| 1 | 0.6 | 0.8 |
| 0.5 | 0.5 | 0.6 |
| | | |



# Concept of Second Order Method

- The Backpropagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient).
- This is the direction in which the cost function decreases most rapidly.
- Although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence due to zigzag nature of convergence path.
- If gradient descent finds minimum in each search direction, successive search directions will be orthogonal to each other. This is the reason behind zigzag convergence path of the gradient descent.

# Concept of Second Order Method

- In Backpropagation, a learning rate is used to determine the length of the weight update (step size).
- The concept behind conjugate gradient descent is that since each step is taking the same general direction, the combination of search directions should minimize the zigzag nature of convergence path and capture the general trend towards the minimum.

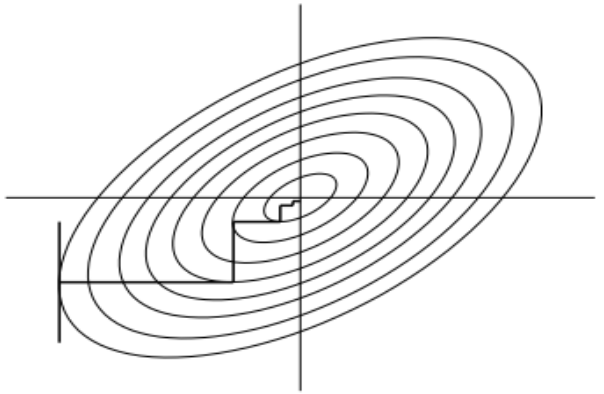


Fig. Convergence of Gradient Descent

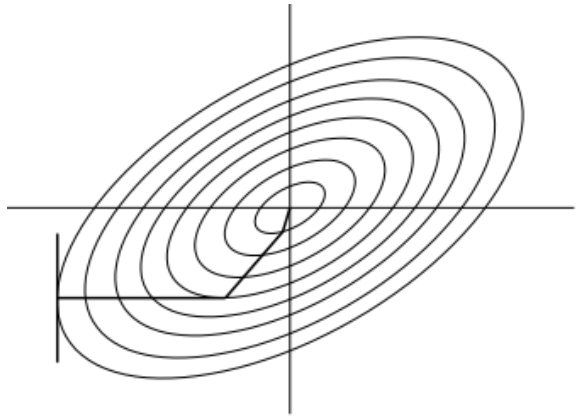


Fig. Convergence of Conjugate Gradient

# Concept of Second Order Method

- All of the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

  $d_0 = -g_0$

- Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction.

  $d_k = -g_k + \beta_k d_{k-1}$

# Concept of Second Order Method

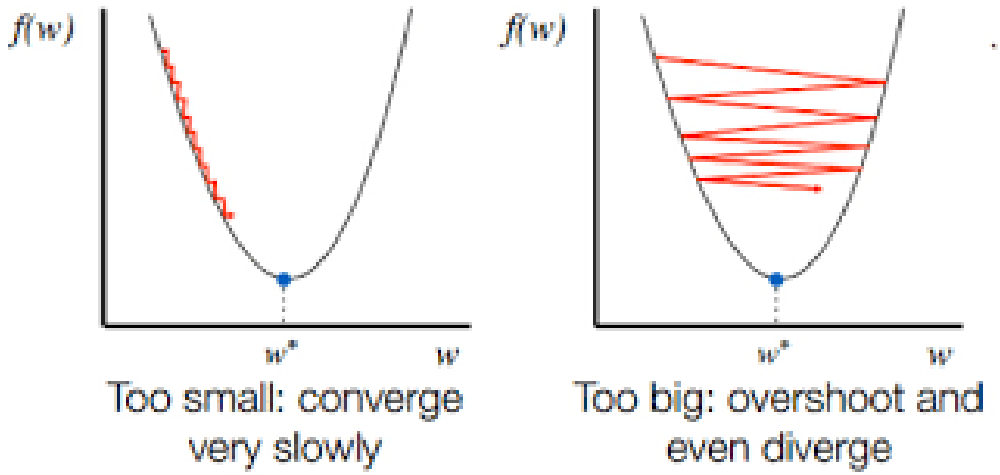
- Here$\beta$ is called the conjugate parameter, and there are different ways to calculate it. Using the Fletcher-Reeves procedure $\beta$ can be calculated as below.

$$\beta_k = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}}$$

This is the ratio of the squared norm of the current gradient to the squared norm of the previous gradient.

# Learning Rate Annealing

- One of the key hyper parameters to set in order to train a neural network is the *learning rate*.
- If learning rate is set too low, training will progress very slowly as we are making very tiny updates to the weights in our network.
- However, if learning rate is set too high, it can cause undesirable divergent behavior in our loss function.



# Learning Rate Annealing

- One approach of using learning rate value properly is to start with large value of learning rate and reduce value of the learning rate according to some predefined schedule. This approach is called learning rate annealing.
- Three commonly used learning rate annealing techniques or learning rate schedules are:
  - Time Based Decay
  - Step Decay
  - Exponential Decay

# Learning Rate Annealing

**Time Based Decay**

- Mathematical formulation of time-based learning rate decay is given below:

$$\alpha = \frac{\alpha_0}{1 + decay\_rate \times epoch\#}$$

***Example***

*Let* $\alpha_0$=0.2 Decay_Rate=1

For Epoch=1 α=0.2/2=0.1

For Epoch=2 α=0.2/3=0.067

For Epoch=3 α=0.2/4=0.05

*So on*

# Learning Rate Annealing

**Step Decay**

- Mathematical formulation of step decay of learning rate is given below. A typical way is to drop the learning rate by half afterv every 10 epochs.

$$\alpha = \alpha_0 \times drop\_rate^{(\lfloor epoch\#/epochs\_drop \rfloor)}$$

***Example***

*Let* $\alpha_0$=0.2 drop_rate=0.5 epochs_drop=5

For Epoch=5 $\alpha=0.2 \times 0.5^{5/5}=0.1$

For Epoch=10 $\alpha= 0.2 \times 0.5^{10/5}=0.05$

For Epoch=15 $\alpha= 0.2 \times 0.5^{15/5}=0.025$

*So on*

# Learning Rate Annealing

**Exponential Decay**

- Mathematical formulation of exponential decay of learning rate is given below.

$$\alpha = \alpha_0 e^{(-k \times epoch\#)}$$

***Example***

*Let* $\alpha_0$=0.2 k=0.1

For Epoch=1 α=0.181

For Epoch=2 α=0.164

For Epoch=3 α=0.148

*So on*

# Learning Rate Adaptation

- Learning rate annealing techniques reduces learning rate equally for all parameters and applies the same learning rate with all parameters.
- The better approach is to use the techniques that gives learning rate a chance to adapt.
- The main idea behind adaptive learning rate is to use larger learning rate for updating parameters that are modified with small scale in past and use smaller learning rate for updating parameters that are modified with large scale in the past.
- Commonly used learning rate adaptation techniques are: Adagrad, Adadelta, RMSProp, and Adam.

# Learning Rate Adaptation

# Learning Rate Adaptation

**Adagrad**

- In Adagrad, the variable $v$, called cache, keeps track of sum of the squared gradients, which in turn is used to normalize the parameter update.

$$v_t = v_{t-1} + dw_t^2$$

$$w_{t+1} = w_t + \frac{\alpha}{\sqrt{v_t + \varepsilon}} dw_t$$

- The effect of above equations is that Weights receiving high gradients will have their effective learning rate reduced.
- On the other hand, weights receiving small gradients will have their effective learning rate increased.

# Learning Rate Adaptation

**Adadelta**

- Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
- Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size W.

# Learning Rate Adaptation

## RMSProp

- Adagrad decays the learning rate very aggressively (as the denominator grows). As a result, after a while, parameters receiving larger gradients will start receiving very small updates because of the decayed learning rate.
- To avoid this why not decay the denominator and prevent its rapid growth.
- RMSProp stands for root mean squared propagation and avoids monotonically increasing denominator of Adagrad.

# Learning Rate Adaptation

## RMSProp

- The central idea of RMSProp is keep the moving average of the squared gradients for each weight. And then divide the learning rate by root mean square of the squared gradients.

$$v_t = \beta v_{t-1} + (1-\beta) dw_t^2$$

$$w_{t+1} = w_t + \frac{\alpha}{\sqrt{v_t + \in}} dw_t$$

# Learning Rate Adaptation

## Adam

- Adam update can be considered as RMSprop with momentum. In place of raw and noisy gradient vector *dw*, weighted average of gradients are used.

*Weighted average of gradients*

$$v_t = \beta_1 v_{t-1} + (1-\beta_1) dw_t$$

*Weighted average of squared gradients*

$$s_t = \beta_2 s_{t-1} + (1-\beta_2) dw_t^2$$

# Learning Rate Adaptation

## Adam

*Then Bias Correction is applied to moving averages as below*

$$v_t = \frac{v_t}{(1-\beta_1^t)} \qquad S_t = \frac{S_t}{(1-\beta_2^t)}$$

*Finally, update weights as below:*

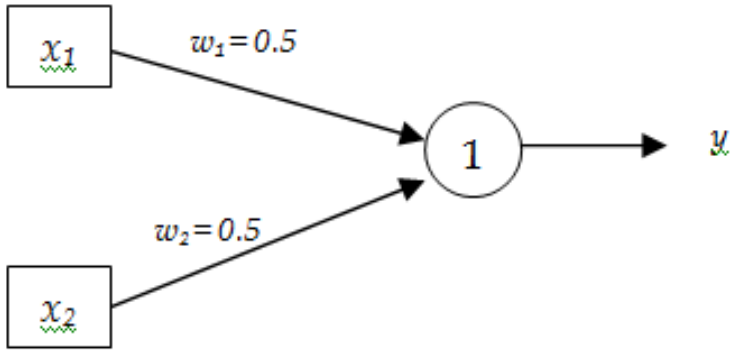
$$w_{t+1} = w_t + \frac{\alpha}{\sqrt{s_t + \in}} v_t$$

# Learning Rate Adaptation

**Example**

- Consider following simple ANN with logistic activation function. Assume that $\alpha = 0.2 \quad \beta = 0.8 \quad \beta_1 = 0.8 \quad \beta_2 = 0.8$ . Calculate weight updates for the given training Sample using (1) Adagrad (2) RMSProp and (3) Adam

| $x_1$ | $x_2$ | $t$ |
|---|---|---|
| 1 | 0.6 | 0.8 |
| 0.5 | 0.5 | 0.6 |
| 0.2 | 0.8 | 0.3 |



# Adaptive Control of Learning Rate

**Solution**

**For Adagrad**

For Training example:

(1,0.6,0.8)

$$y_{in} = 0.5*1 + 0.5*0.6 = 0.8$$

$$y = \frac{1}{1+\exp(-y_{in})} = 0.6899$$

$$dw_1 = (t-y)\times y \times (1-y) \times x_1 = 0.02354$$

$$dw_2 = (t-y)\times y \times (1-y) \times x_2 = 0.01412$$

$$(v_1)_1 = (v_1)_0 + 0.02354^2 = 0.000554$$

$$(v_2)_1 = (v_2)_0 + 0.01412^2 = 0.000199$$

$$(w_1)_2 = (w_1)_1 + \frac{0.2}{\sqrt{0.000554}} \times (0.02354) = 0.7$$

$$(w_2)_2 = (w_2)_1 + \frac{0.2}{\sqrt{0.000199}} \times (0.01412) = 0.7$$

# Adaptive Control of Learning Rate

**Solution**

**For Adagrad**

For Training example: (0.5,0.5,0.6)

$$y_{in} = 0.7 * 0.5 + 0.7 * 0.5 = 0.7$$

$$y = \frac{1}{1 + \exp(-y_{in})} = 0.6682$$

$$dw_1 = (y - t) \times y \times (1 - y) \times x_1 = 0.007559$$

$$dw_2 = (y - t) \times y \times (1 - y) \times x_2 = 0.007559$$

$$v_1 = 0.000554 + 0.007559^2 = 0.000611$$

$$v_2 = 0.000199 + 0.007559^2 = 0.000257$$

$$w_1 = w_1 - \frac{0.2}{\sqrt{0.000611}} \times 0.007559 = 0.639$$

$$w_2 = w_2 - \frac{0.2}{\sqrt{0.000257}} \times 0.007559 = 0.606$$

# Adaptive Control of Learning Rate

**Solution**

**For Adagrad**

For Training example:

(0.2,0.8,0.3)

$$y_{in} = ????$$

$$y = \frac{1}{1+\exp(-y_{in})} = ??$$

$$dw_1 = (y-t)\times y\times(1-y)\times x_1 = ??$$

$$dw_2 = (y-t)\times y\times(1-y)\times x_2 = ??$$

$$v_1 = ??$$

$$v_2 = ??$$

$$w_1 = ??? = 0.531$$

$$w_2 = ??? = 0.412$$

# Adaptive Control of Learning Rate

**Solution**

**For RMSProp**

For Training example:(1,0.6,0.8)

$$y_{in} = 0.5*1 + 0.5*0.6 = 0.8$$

$$y = \frac{1}{1+\exp(-y_{in})} = 0.6899$$

$$dw_1 = (y-t)\times y\times(1-y)\times x_1 = -0.02354$$

$$dw_2 = (y-t)\times y\times(1-y)\times x_2 = -0.01412$$

$$v_1 = 0.8*0 + (1-0.8)*(-0.02354)^2 = 0.000111$$

$$v_2 = 0.8*0 + (1-0.8)*(-0.01412)^2 = 0.0000399$$

$$w_1 = w_1 - \frac{0.2}{\sqrt{0.000111}}\times(-0.02354) = 0.947$$

$$w_2 = w_2 - \frac{0.2}{\sqrt{.0000399}}\times(-0.01412) = 4.971$$

# Adaptive Control of Learning Rate

**Solution**

**For RMSProp**

For Training example:(0.5,0.5,0.6)

$$y_{in} = 0.947 * 0.5 + 4.971 * 0.5 = ??$$

$$y = \frac{1}{1 + \exp(-y_{in})} = ??$$

$$dw_1 = (y - t) \times y \times (1 - y) \times x_1 = ???$$

$$dw_2 = (y - t) \times y \times (1 - y) \times x_2 = ???$$

$$v_1 = 0.8 * 0.000111 + (1 - 0.8) * (dw_1)^2 = ????$$

$$v_2 = 0.8 * 0.0000399 + (1 - 0.8) * (dw_2)^2 = ???$$

$$w_1 = w_1 - \frac{0.2}{\sqrt{v_1}} \times (dw_1) = ???$$

$$w_2 = w_2 - \frac{0.2}{\sqrt{v_2}} \times (dw_2) = ???$$

# Adaptive Control of Learning Rate

**Solution**

**For RMSProp**

For Training example:(0.2,0.8,0.3)

$$y_{in} = ???$$

$$y = \frac{1}{1+\exp(-y_{in})} = ??$$

$$dw_1 = (y-t)\times y\times(1-y)\times x_1 = ???$$

$$dw_2 = (y-t)\times y\times(1-y)\times x_2 = ???$$

$$v_1 = 0.8 * ??? + (1-0.8) * (dw_1)^2 = ????$$

$$v_2 = 0.8 * ??? + (1-0.8) * (dw_2)^2 = ???$$

$$w_1 = w_1 - \frac{0.2}{\sqrt{v_1}} \times (dw_1) = ???$$

$$w_2 = w_2 - \frac{0.2}{\sqrt{v_2}} \times (dw_2) = ???$$

# Adaptive Control of Learning Rate

**Solution**

**Adam**

??????---try yourself

# Adaptive Control of Learning Rate

**Example**

- Consider following simple ANN with logistic activation function. Assume that $\alpha = 0.2 \quad \beta = 0.8 \quad \beta_1 = 0.8 \quad \beta_2 = 0.8$ . Calculate weight updates for the given training Sample using (1) Momentum (2) Adagrad (3) RMSProp and (4) Adam.
- Note: Use bias correction feature in each case

| $x_1$ | $x_2$ | $t$ |
|---|---|---|
| 1 | 0.6 | 0.8 |
| 0.5 | 0.5 | 0.6 |
| 0.2 | 0.8 | 0.3 |