Deep Learning

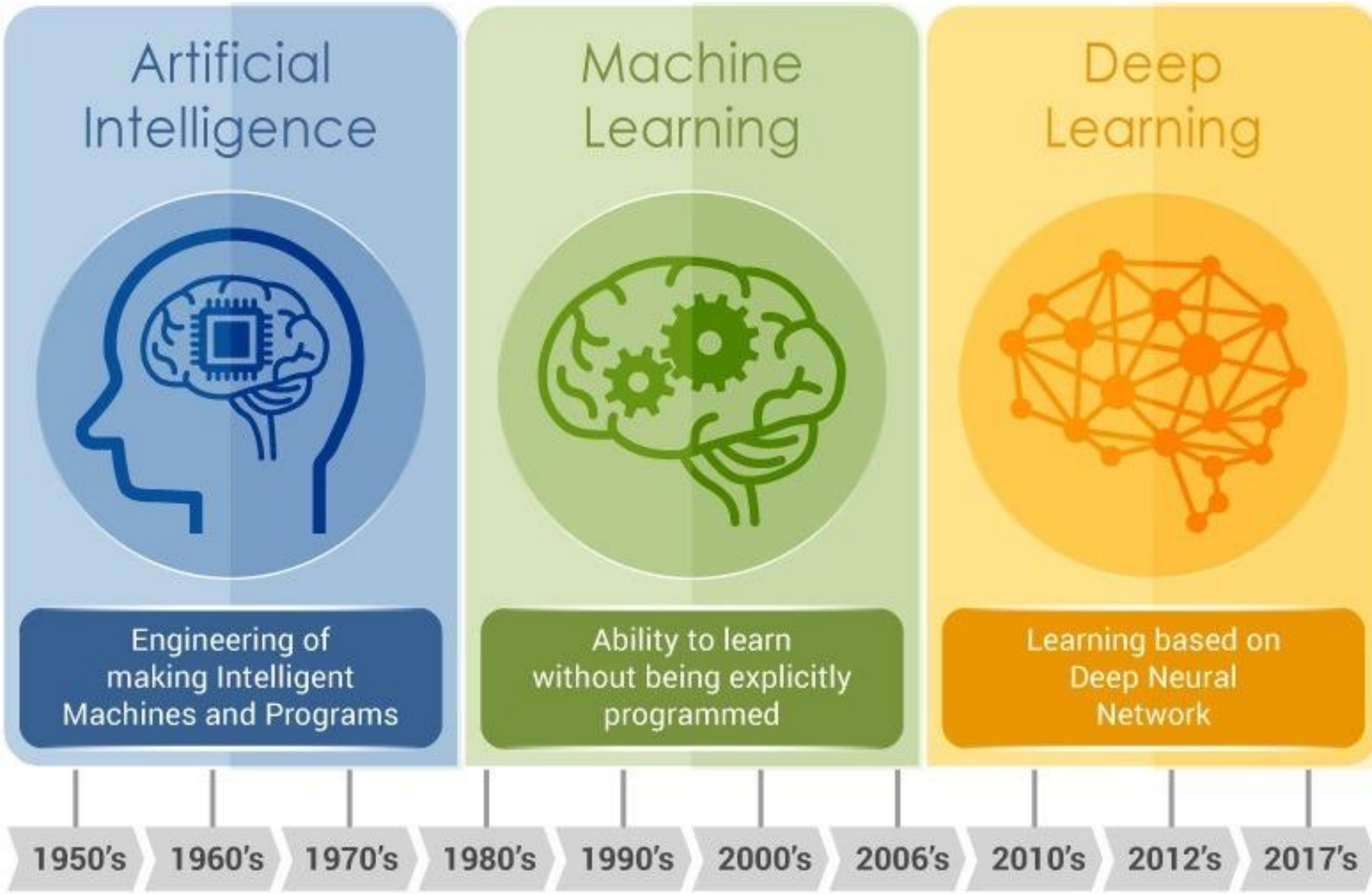
Optimization and Regularization

Puneet Kumar Jain

CSE Department

National Institute of Technology Rourkela

AI vs Machine Learning vs Deep Learning

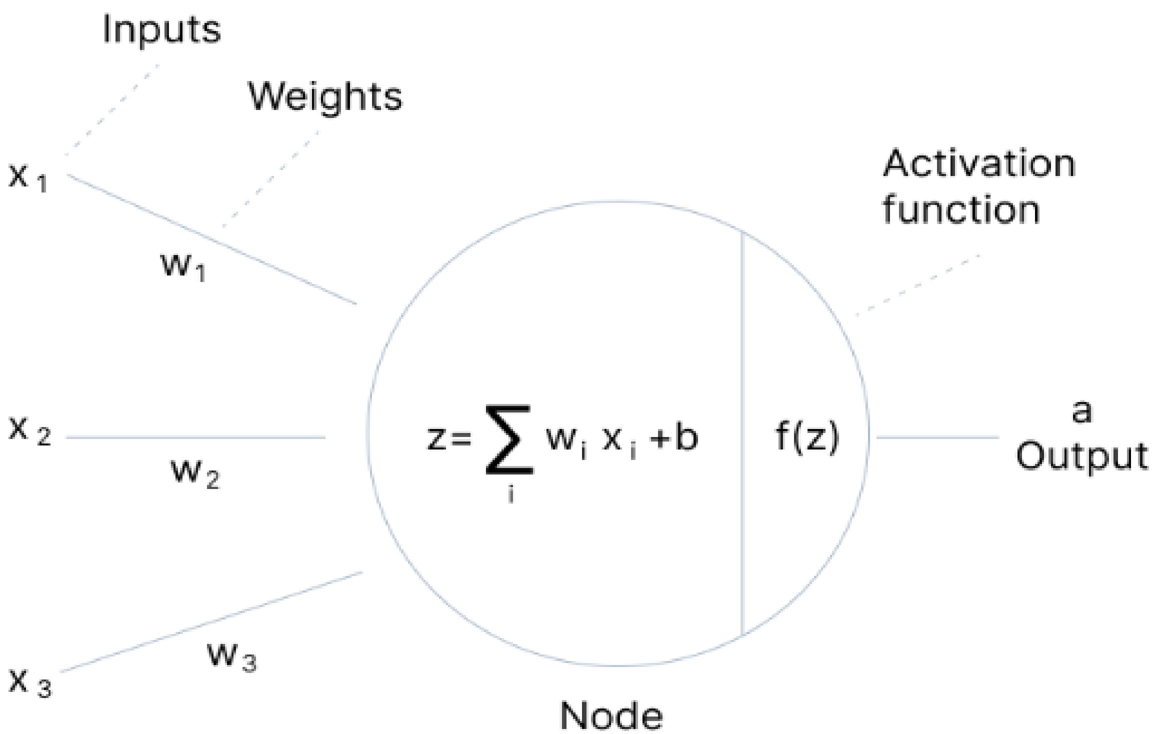


Algorithmic Advancements…

- Better ***Activation Functions*** for neural layers.
- Better ***Weight Initialization*** Schemes starting with layer-wise pretraining.
- To avoid Overfitting the Concepts like ***Dropout*** is Introduced.
- Better ***optimization schemes***, such as RMSProp and Adam.

Activation Functions…

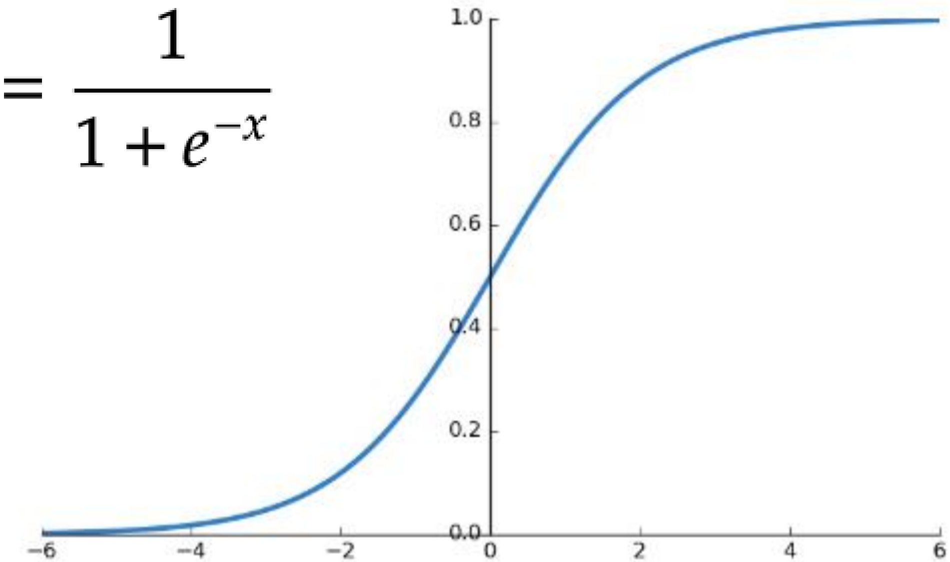
- An *Activation Function (Transfer Function)* maps the weighted summation of inputs to output.
- An Activation function is used to add ***Nonlinearity so that the network can learn complex patterns.***

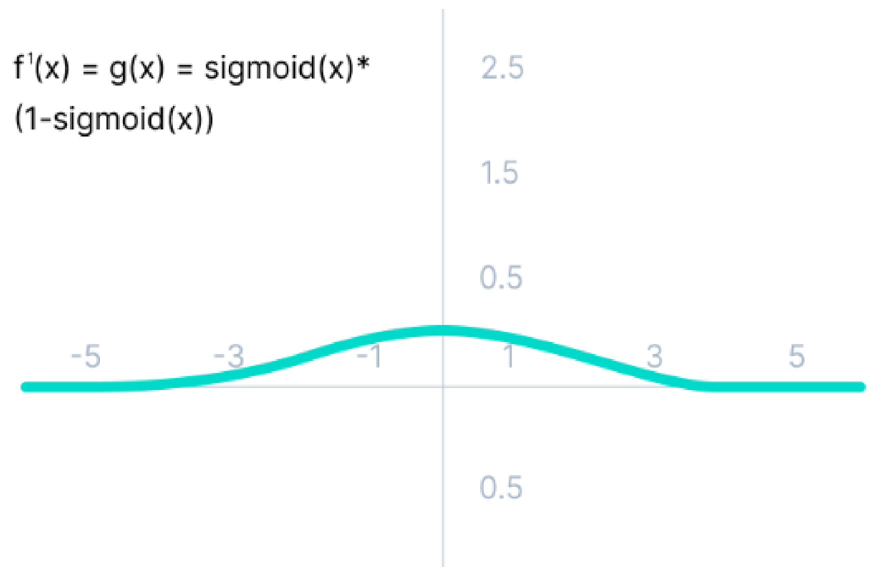


Sigmoid Activation Functions

- **Characteristics:**
  - Differentiable
  - Nonlinear
  - O/P lies in [0-1]
  - Fast
  - ***Vanishing Gradient Problem***

$$f(x) = \frac{1}{1+e^{-x}}$$





VANISHING GRADIENT PROBLEM

- Because of sigmoid activation function the derivative is ***less than 1*** and *when the derivatives are multiplied it gives a very small number* which ultimately changes the weight very less.
- *Usually occurs when the derivative is less than 1.*
- In case of ***sigmoid and tanh activation*** function it occurs frequently.

$$\frac{dL}{dw} = \frac{dL}{df_1} \times \frac{df_1}{df_2} \times \frac{df_2}{df_3} \times \cdots \ldots \ldots \ldots \ldots \ldots \times \frac{df_n}{dw}$$

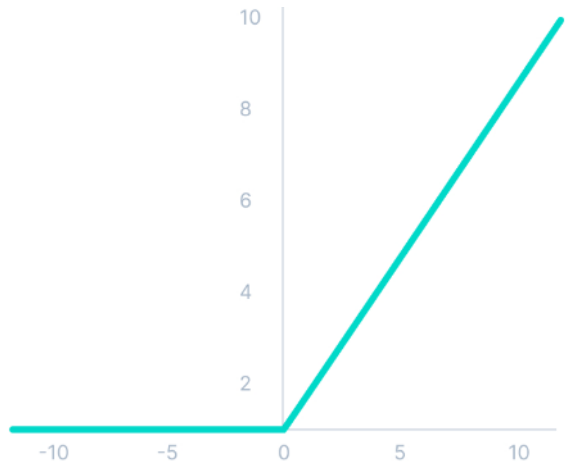
ReLU Activation Function

- f(x)= x,   when x>0

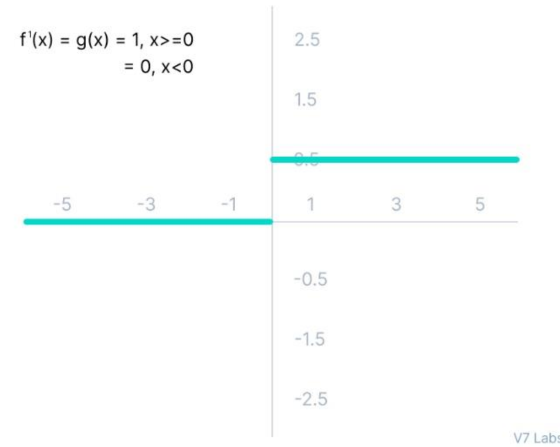
  = 0,   when x<=0
- Avoids Vanishing Gradient Problem.
- Derivative is Simple
  - f'(x)= 1 for x>=0

    = 0 for x<0
- Problem:
  - Dead ReLU Units

$$f(x) = max\,(0, x)$$



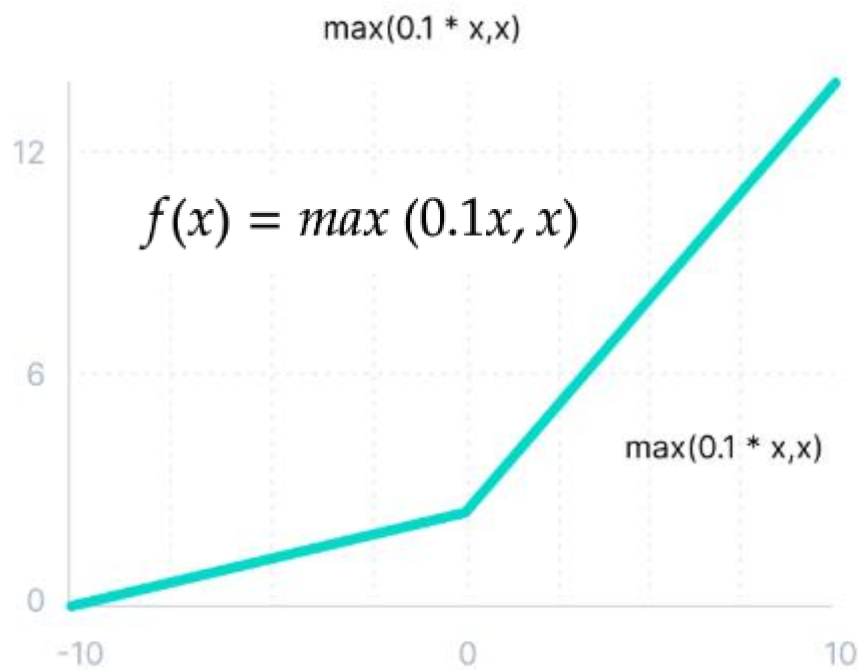
The Dying ReLU problem



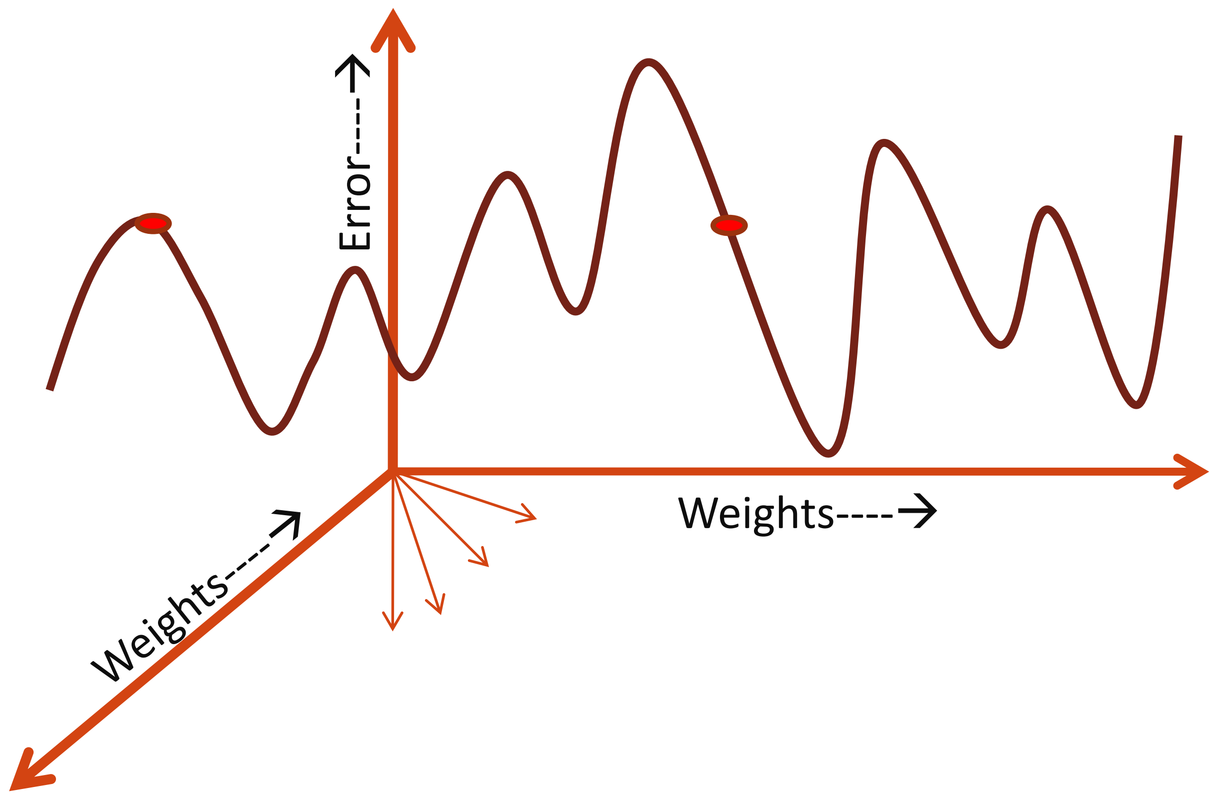
https://www.v7labs.com/blog/neural-networks-activation-functions

Leaky ReLU Activation Function

- f(x)= x,   when x>0
  = 0.1x,   when x<=0
- The advantages of Leaky ReLU are same as that of ReLU.
- In addition, it enables Backpropagation, even for negative input values.
- ***Avoids Dead ReLU***
- Simple Derivative
  - f'(x)= 1   for x>=0
    = 0.1 for x<0



WEIGHT INITIALIZATION



WEIGHT INITIALIZATION

- Mostly used
  - We should never initialize to same values.
    - Asymmetry is necessary
  - We should not initialize to large –ve values
    - Vanishing Gradient problems
  - Weights should be small (not too small)
  - Weights should have good variance
  - Weights should come from a Normal distribution with mean zero and small variance
  - Should have some +ve and Some –ve values

WEIGHT INITIALIZATION

- Xavier/Glorot initialization in 2010- ***well for sigmoid activation function***
  - First Variation –

$$W_{ij} = N(0, \sigma_{ij}), \quad \sigma_{ij} = \frac{2}{Fanin + Fanout}$$

  - Second Variation–

$$W_{ij} = U\left(-\frac{\sqrt{6}}{\sqrt{Fanin + Fanout}}, \frac{\sqrt{6}}{\sqrt{Fanin + Fanout}}\right)$$

WEIGHT INITIALIZATION

- He Initializer, 2015 ***works well for ReLU***
  - First Variation – $$W_{ij} = N(0, \sigma_{ij}), \quad \sigma_{ij} = \sqrt{\frac{2}{Fanin}}$$
  - Second Variation– $$W_{ij} = U\left(-\frac{\sqrt{6}}{\sqrt{Fanin}}, \frac{\sqrt{6}}{\sqrt{Fanin}}\right)$$

BIAS-VARIANCE TRADE-OFF

No. of Layers Increases → **More No. of Weights** → **Chances to Overfit is High** → **Problem of High Variance**

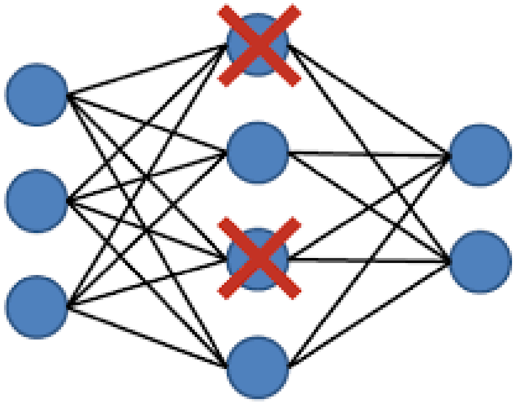
No. of Layers Decreases → **Less No. of Weights** → **Chances to Underfit is High** → **Problem of High Bias**

- Multilayer ANN has higher chance of overfitting.

Dropout and Regularization

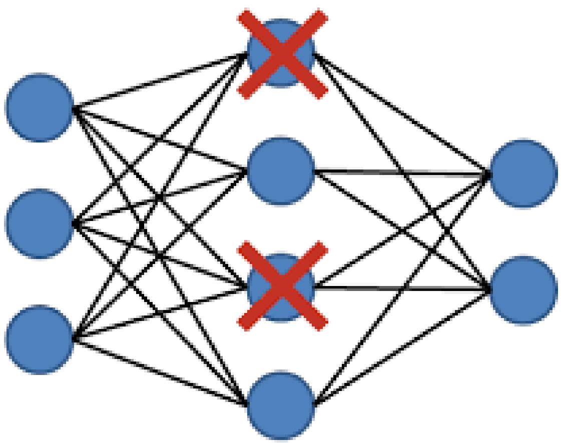
- Deep NN tend to overfit because of many layers and weights
- For this dropout and regularization is needed
- In Dropout, a certain percentage of inputs and hidden layer neurons are dropped out for an iteration
- Some call it as drop out network or layer.

‘



Dropout

- Procedure:
  - During training we decide with probability *p* to update a node's weights or not.
  - We set *p* to be typically 0.5
- Highly effective in deep learning:
  - Decreases overfitting
  - Reduces training time
- Can be loosely interpreted as ensemble of networks



BATCH NORMALIZATION

- Normalization is a data pre-processing tool used to bring the numerical data to a common scale without distorting its shape.

  - Decimal Scaling:

$$N_i = \frac{T_i}{10^p}$$

  - Median:

$$N_i = \frac{T_i}{\text{median}(T)}$$

  - Min-Max
  - Vector:

$$N_i = \text{Min}_N + \frac{T_i - \text{Min}_T}{\text{Max}_T - \text{Min}_T} \times (\text{Max}_N - \text{Min}_N)$$

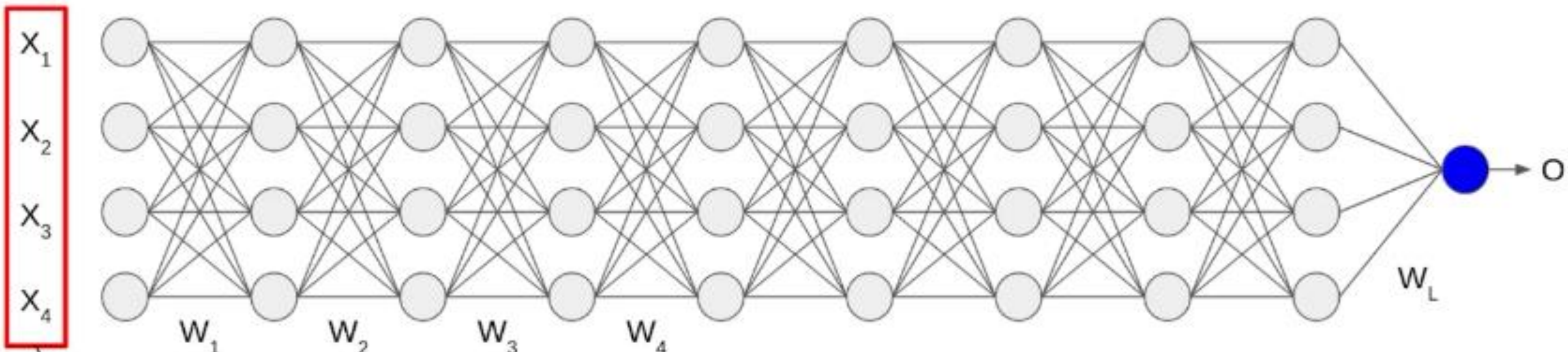
$$N_i = \frac{T_i}{\sqrt{\sum_{j=1}^{k} T_j^{\,2}}}$$

  - Z-Score

$$N_i = \frac{T_i - \mu_T}{\sigma_T}$$

BATCH NORMALIZATION

- Motivation



Normalize the inputs

$h_1 = \sigma(W_1 X)$

$h_2 = \sigma(W_2 h_1) = \sigma(W_2 \sigma(W_1 X))$

$O = \sigma(W_L h_{L-1})$

BATCH NORMALIZATION

$$\mu = \frac{1}{m}\sum h_i$$

$$\sigma = \sqrt{\frac{1}{m}\sum (h_i - \mu)^2}$$

- Where m: Number of Neurons at h_i

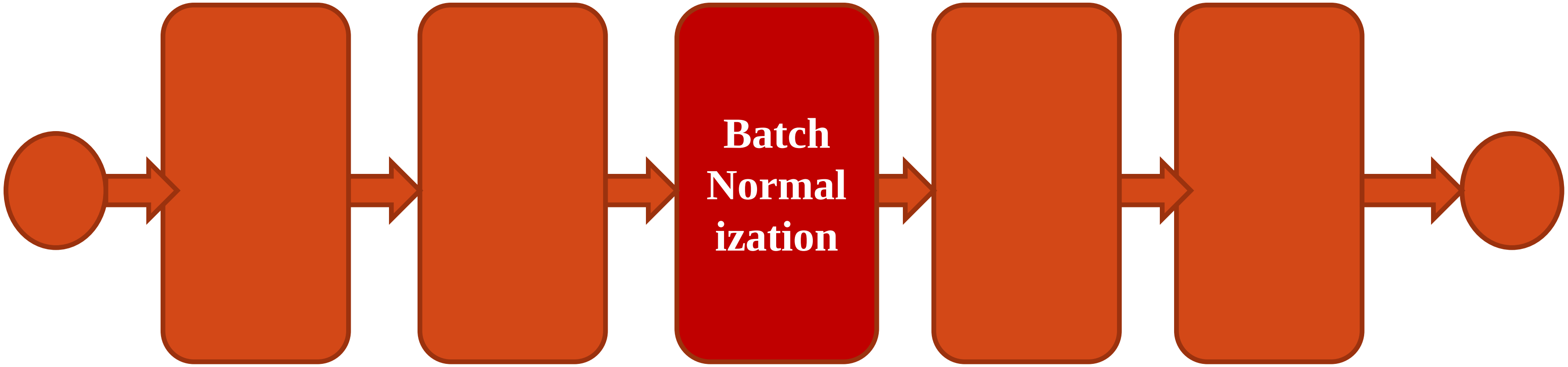
$$h_{i(norm)} = \frac{h_i - \mu}{\sigma + \epsilon}$$

- Where γ and β are hyper parameters.

$$h_i = \gamma . h_{i(norm)} + \beta$$

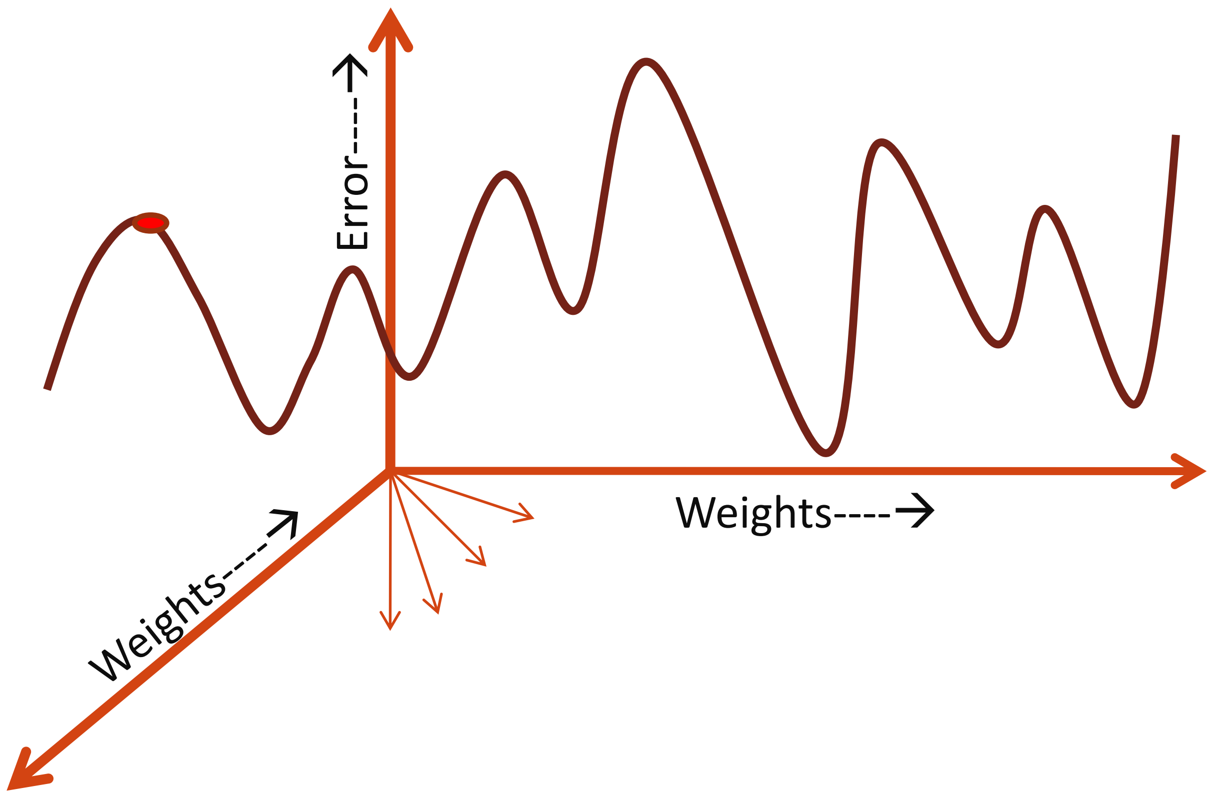
BATCH NORMALIZATION

- Advantages
  - Faster Convergence
  - Weak Regularizer (Batch Normalization + dropout)
  - Avoids internal covariate shift
- https://arxiv.org/pdf/1502.03167v3.pdf



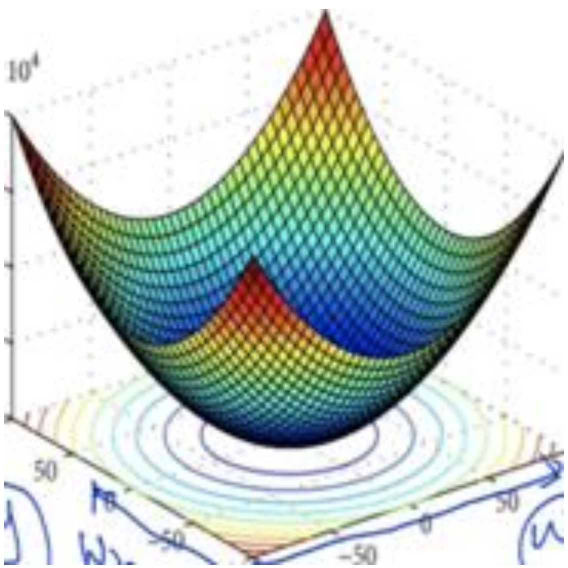
OPTIMIZERS

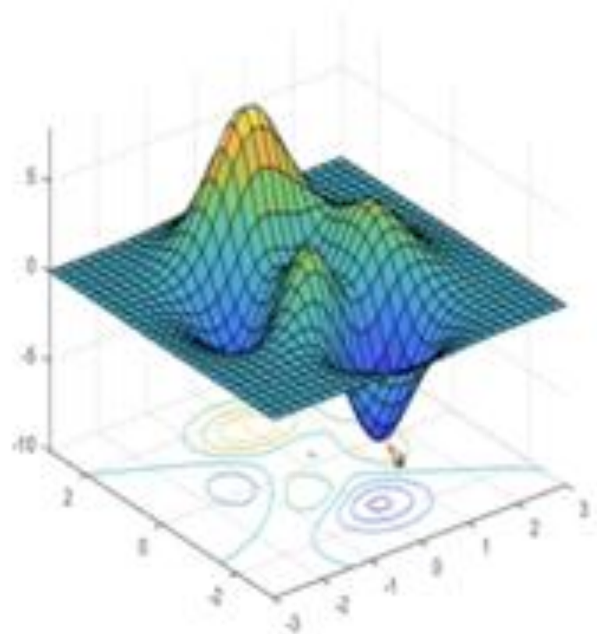
- At minima, maxima and saddle point, u have the gradient as Zero.

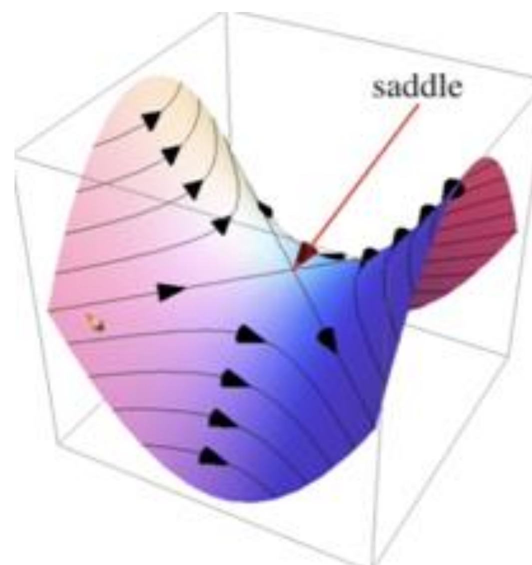


OPTIMIZERS

- Convex function and Non-Convex Function
- Convex functions have either 1 maxima or minima. (Local minima=global minima)
- Non-convex functions have more than one minima or maxima

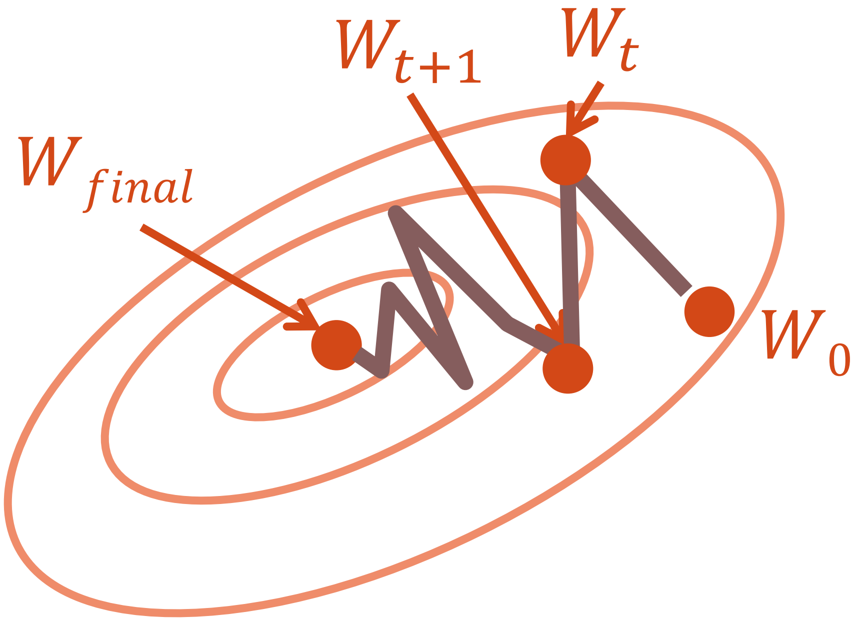






Stochastic gradient descent (SGD)

You take one point (Input Vector), Feed Forward it then update the weights by back-propagating the gradient of errors.



- Initialize W_0 randomly
- For t in $0,\ldots,T_{\text{maxiter}}$

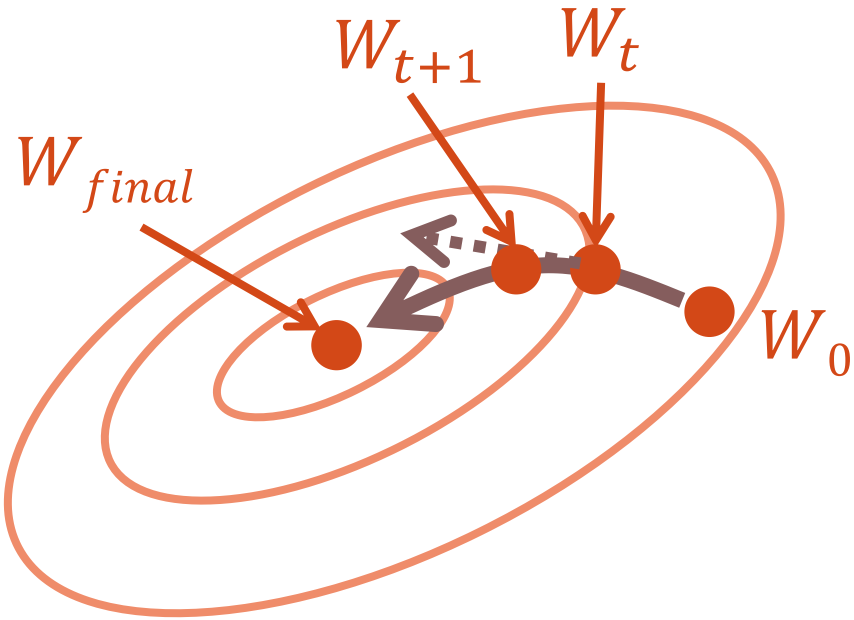
$$W^{t+1} = W^t - \eta_t \cdot \underbrace{\nabla Loss(f_w(x_i), y_i)}_{\text{Stochastic gradient}}$$

where index i is chosen randomly

- computation of $\nabla Loss(\ldots)$ requires only one training example
- Per-iteration comp. cost = O(1)

Gradient descent

You take all Input Vectors, Feed Forward it one by one, compute the error and get the mean error, then update the weights by back-propagating the gradient of errors.



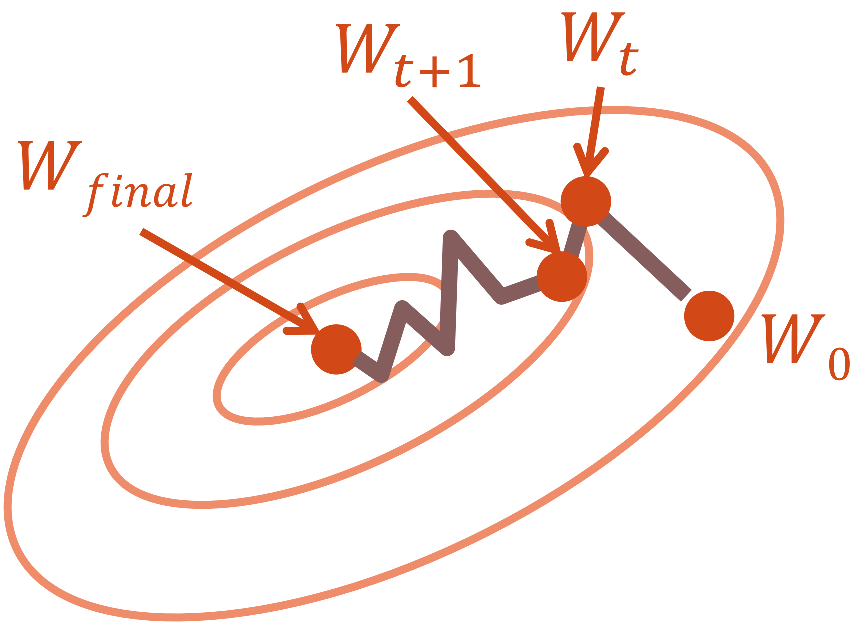
- Initialize W_0 randomly
- For t in $0, \ldots, T_{\text{maxiter}}$

$$W^{t+1} = W^{t} - \eta_t \cdot \nabla L(\underbrace{f_w(x_i), y_i}_{\text{Gradient of the objective}})$$

- computation of $\nabla L(W^t)$ requires a full sweep over the training data
- Per-iteration comp. cost = O(n)

Minibatch stochastic gradient descent

You take a subset of Input Vectors (more than one), Feed Forward it one by one, compute the error and get the mean error, then update the weights by back-propagating the gradient of errors.



- Initialize W_0 randomly
- For t in 0,…, T_{maxiter}

$$W^{t+1} = W^{t} - \eta_t \cdot \underbrace{\tilde{\nabla}_B L(W)}_{\text{minibatch gradient}}$$

where minibatch B is chosen randomly

- $\tilde{\nabla} L(\theta)$ is average gradient over random subset of data of size B
- Per-iteration comp. cost = O(B)

STOCHASTIC GRADIENT WITH MOMENTUM

- The **rate of convergence** of **Stochastic Gradient** can be improved by *adding a momentum* to the Gradient expression.
- This can be ***achieved by adding a fraction of previous weight change to the current weight change***.

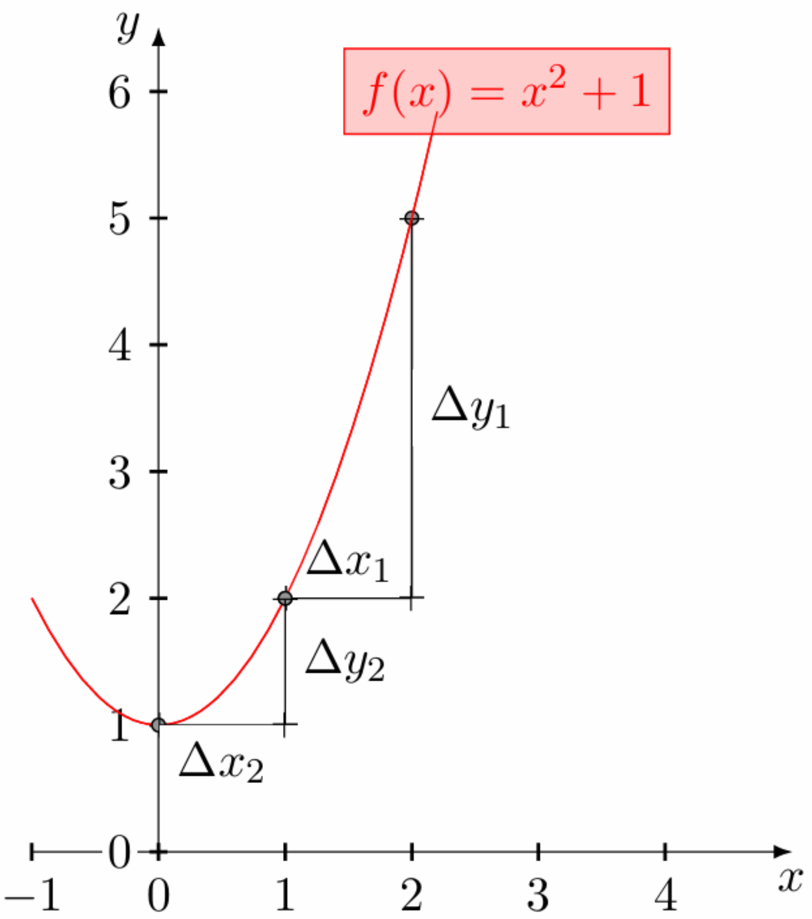
$$(w_i)_{new} = (w_i)_{old} - \eta \left[\frac{dL}{dw_i}\right]$$

$$(\mathrm{w_i})_{\mathrm{t}} = (\mathrm{w_i})_{\mathrm{t-1}} + \alpha . \Delta w_{t-1} - \eta \frac{dL}{dw}$$

Momentum

Learning Rate

Challenges with Gradient Descent



- When the curve is steep the gradient $(\frac{\Delta y_1}{\Delta x_1})$ is large
- When the curve is gentle the gradient $(\frac{\Delta y_2}{\Delta x_2})$ is small
- Recall that our weight updates are proportional to the gradient $w = w - \eta \nabla w$
- Hence in the areas where the curve is gentle the updates are small whereas in the areas where the curve is steep the updates are large

Momentum based GD

Intuition

- If I am repeatedly being asked to move in the same direction then I should probably gain some confidence and start taking bigger steps in that direction
- Just as a ball gains momentum while rolling down a slope

Update rule for momentum based gradient descent

$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

- In addition to the current update, also look at the history of updates.

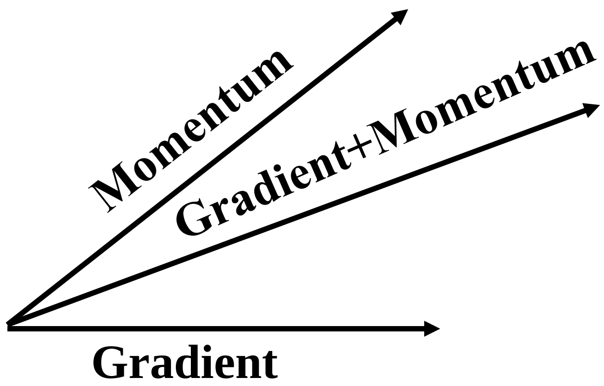
Some observations and questions

- Even in the regions having gentle slopes, momentum based gradient descent is able to take large steps because the momentum carries it along
- Is moving fast always good? Would there be a situation where momentum would cause us to run pass our goal?

Nestrov Accelerated Gradient (NAG)

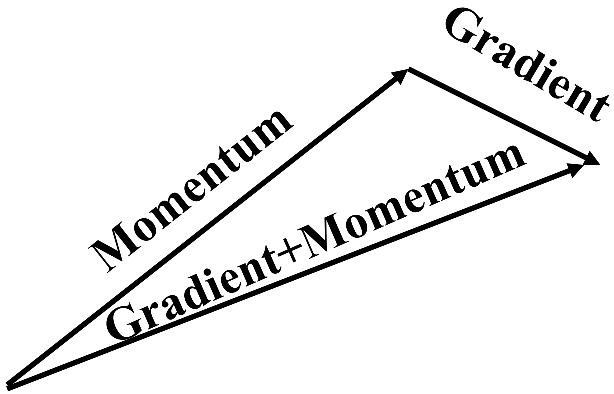
- SGD + Momentum

$$(\mathrm{w_i})_\mathrm{t} = (\mathrm{w_i})_\mathrm{t-1} + \alpha.\Delta w_{\mathrm{t-1}} - \eta \frac{dL}{dw}$$

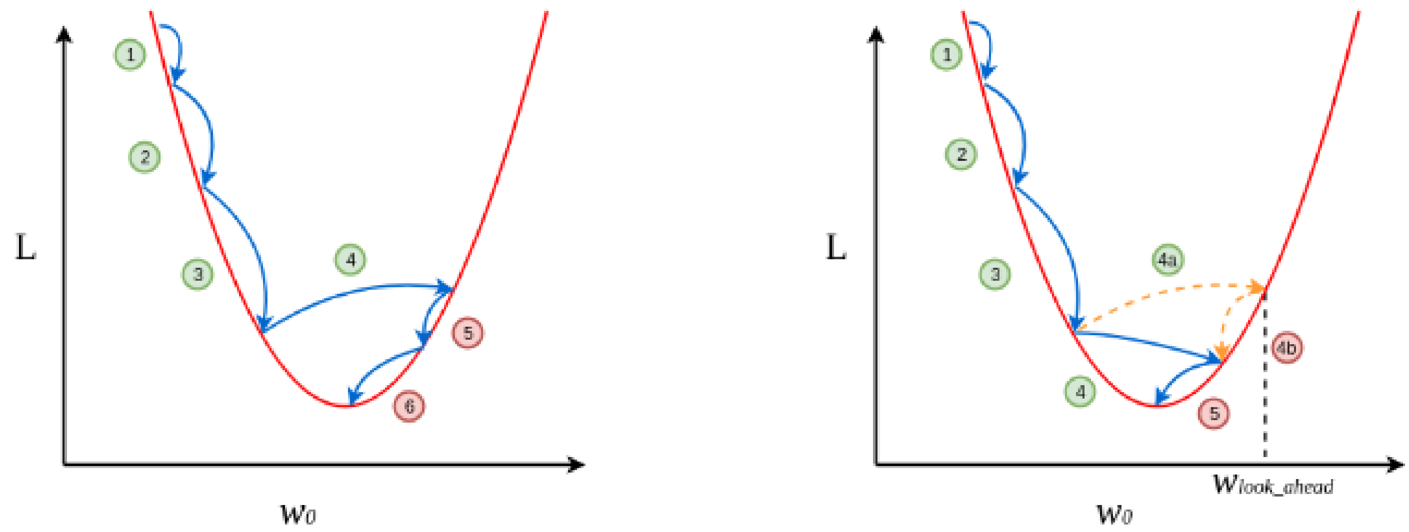


Nestrov Accelerated Gradient (NAG)

- NAG



Nestrov Accelerated Gradient (NAG)



(a) Momentum-Based Gradient Descent

(b) Nesterov Accelerated Gradient Descent

Adaptive Gradient(AdaGrad)

- In SGD, SGD+Momentum and NAG, the learning rate is same for each weight.
- However, in Adagrad you have different learning rate for different weights.
- Why
  - Sparse Feature
  - Dense Feature

Adagrad: GD with adaptive learning rate

Intuition

- Decay the learning rate for parameters in proportion to their update history (more updates means more decay)

Update rule for Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

... and a similar set of equations for b_t

AdaDelta

$$\eta_t' = \frac{\eta}{\sqrt{Exponentially\ Decaying(\alpha)_{t-1} + \epsilon}}$$

- $EDA_{t-1} = \gamma * EDA_{t-1} + (1-\gamma)\left(\frac{dL}{dw}\right)_{t-2}^{2}$

- Avoids the Problem of slow convergence of AdaGrad

RMSprop

Intuition

- Adagrad decays the learning rate very aggressively (as the denominator grows)
- As a result after a while the frequent parameters 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

Update rule for RMSProp

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

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

... and a similar set of equations for b_t

Adam

Intuition

- Do everything that RMSProp does to solve the decay problem of Adagrad
- Plus use a cumulative history of the gradients
- In practice, $\beta_1 = 0.9$ and $\beta_2 = 0.999$

Update rule for Adam

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla w_t$$

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * (\nabla w_t)^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

... and a similar set of equations for b_t

How to Train a Deep Neural Network?

1. **Pre-processing:** Data Narmalization
2. **Weight Initialization**
   - Xavier & Glorot (For Sigmoid)
   - He Initializer (For ReLU)
3. **Choose the Activation Function** (ReLU-Most Favourite)
4. **Batch Normalization** (Especially for later layers close to O/P Layer)
5. **Use Dropout**
6. **Choose the Optimizer** (Favourite- Adam)
7. **Hyper-parameters:** Architecture(# Layers, # Neurons), etc…
8. **Loss Function**
   - 2-Class Classification : Log Loss
   - Multi-Class Classification: Multi-Class Log Loss
   - Regression: Squared Loss

- **End of topic**