Lecture-10-11
Course: Data Science

Introduction to Neural Network and Deep Learning

By

Dr. Sibarama Panigrahi

Senior Member, IEEE

Assistant Professor, Department of Computer Sc. & Engineering

National Institute of Technology, Ruorkela, Odisha, 769008, India

Mobile No.: +91-7377302566

Email: panigrahis[at]nitrkl[dot]ac[dot]in

panigrahi[dot]sibarama[at]gmail[dot]com

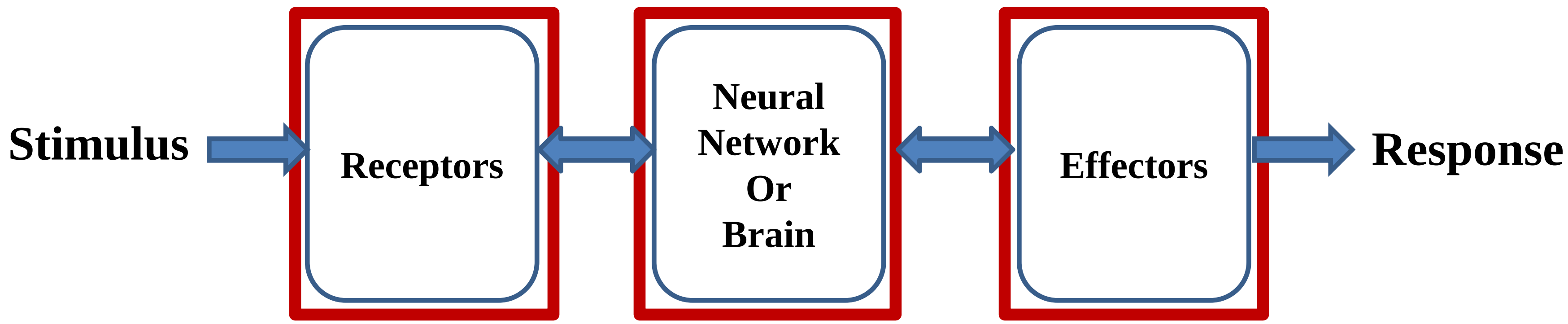
Outlines

- Introduction to Neural Network
- **Mathematical Model for Neural Network**
- Differentiation and its Application to Train Neural Network
- **Deep Neural Network**
- Recent Advances in Deep Learning
  - Activation Function, Weight Initialization (Xavier & Glorot, He)
  - Dropout and Regularization, Batch Normalization
  - Optimizers (SGD, NAG, AdaGrad, AdaDelta, RMSPROP, ADAM)
  - Building and Training Deep Neural Network using Python
- **Introduction to Hyper-Parameter Optimization**

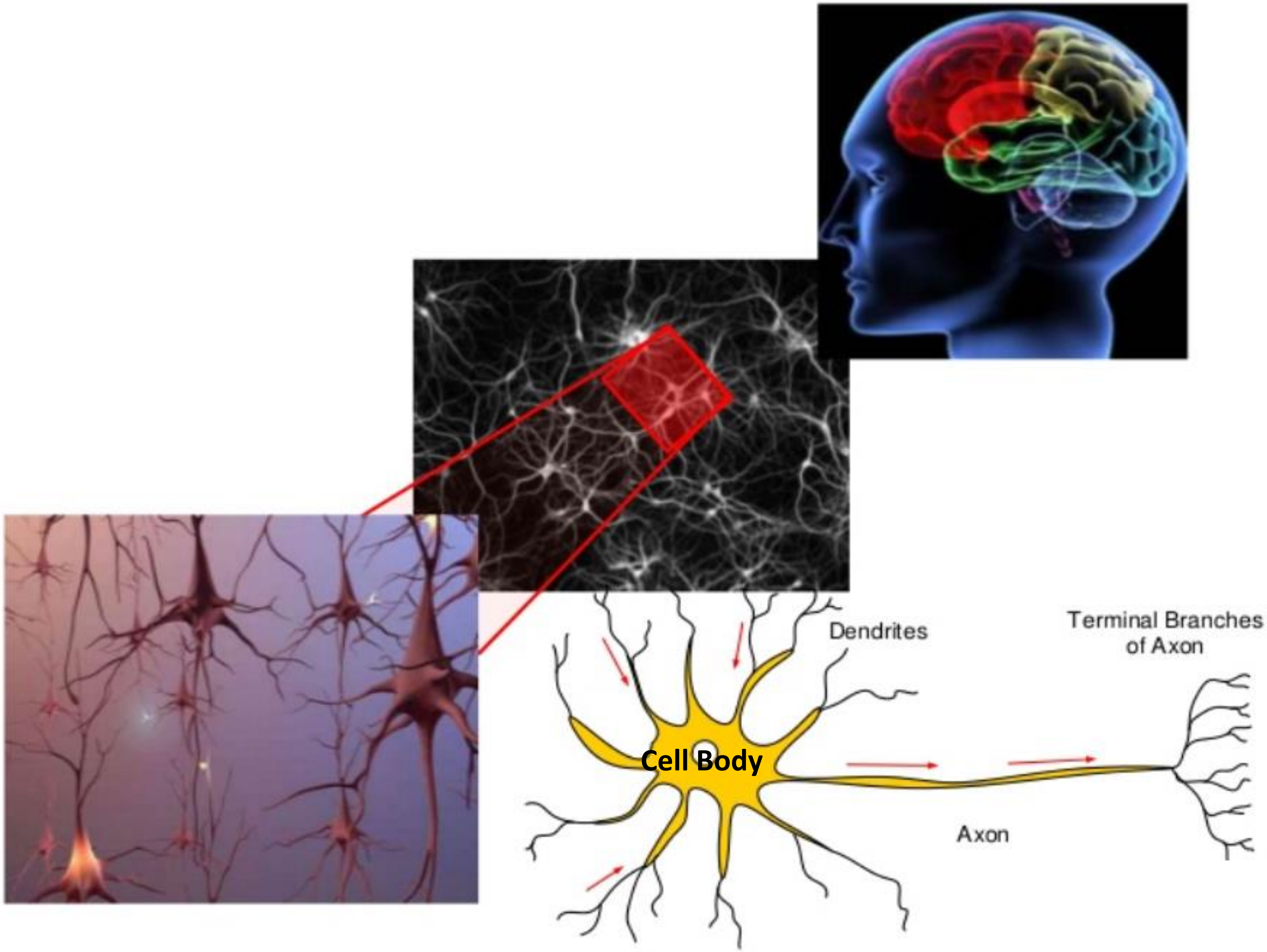
Artificial Neural Network

- An Artificial Neural Network (ANN) is a mathematical model that *loosely simulates* the structure and functionality of **Biological** nervous system to map the inputs to outputs.

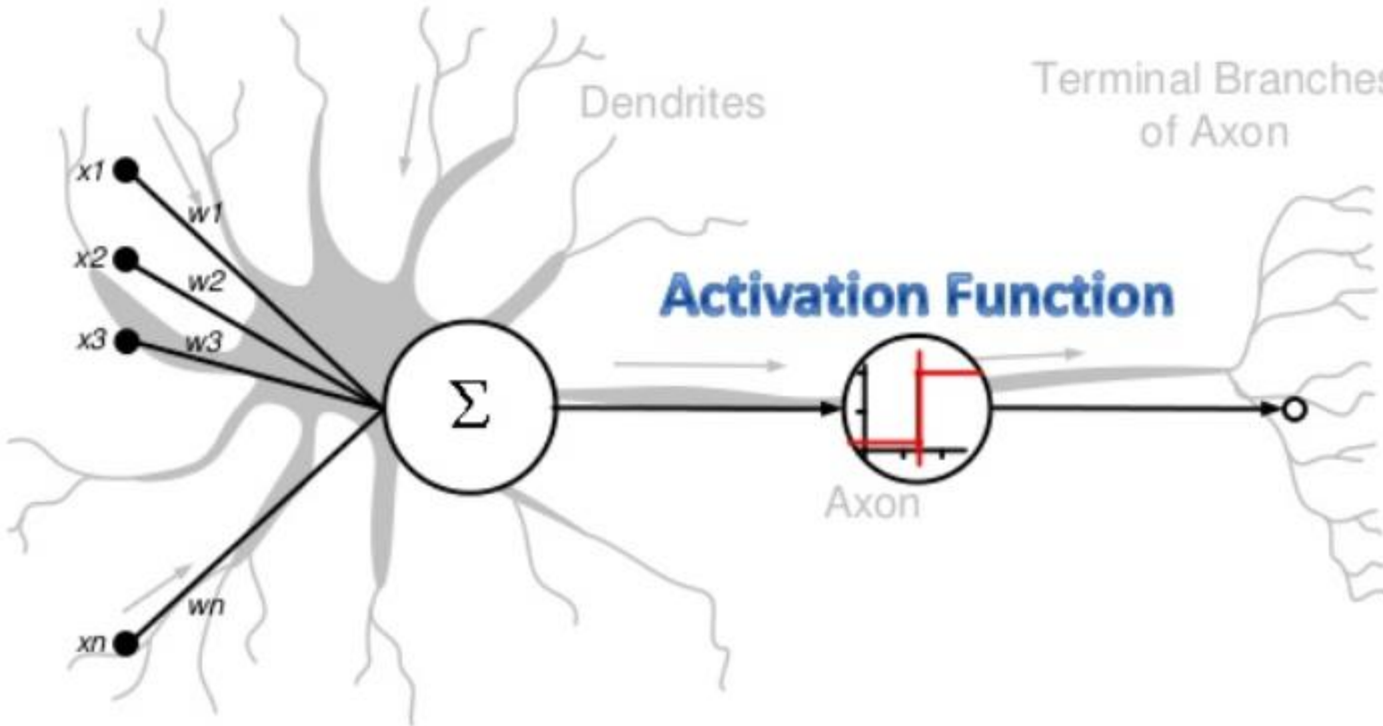
Block Diagram of Biological Nervous System



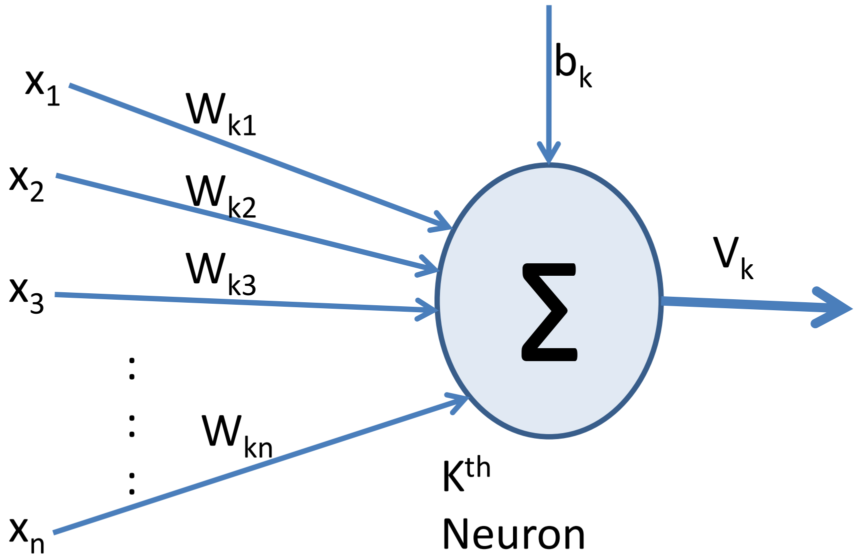
Typical Human Brain



Human Brain Neuron vs Artificial Neuron

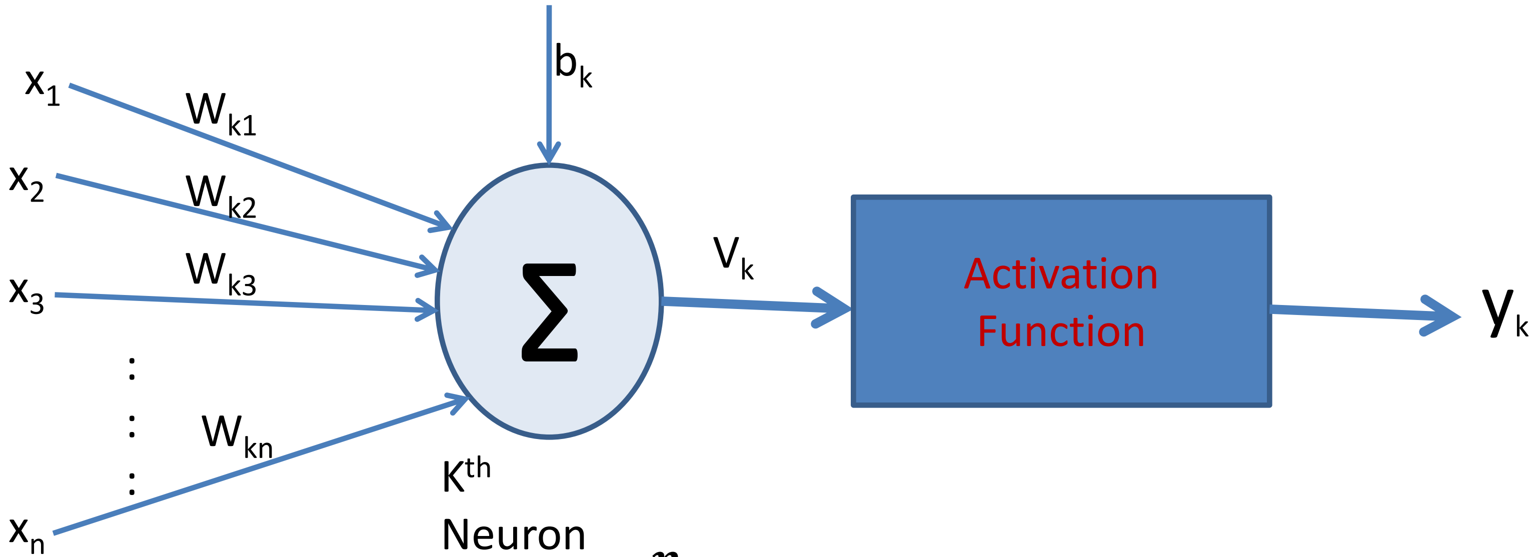


Artificial Neuron



$$V_k = W_{k1} * x_1 + Wk_2 * x_2 + Wk_3 * x_3 + \cdots + W_{kn} * x_n + b_k$$

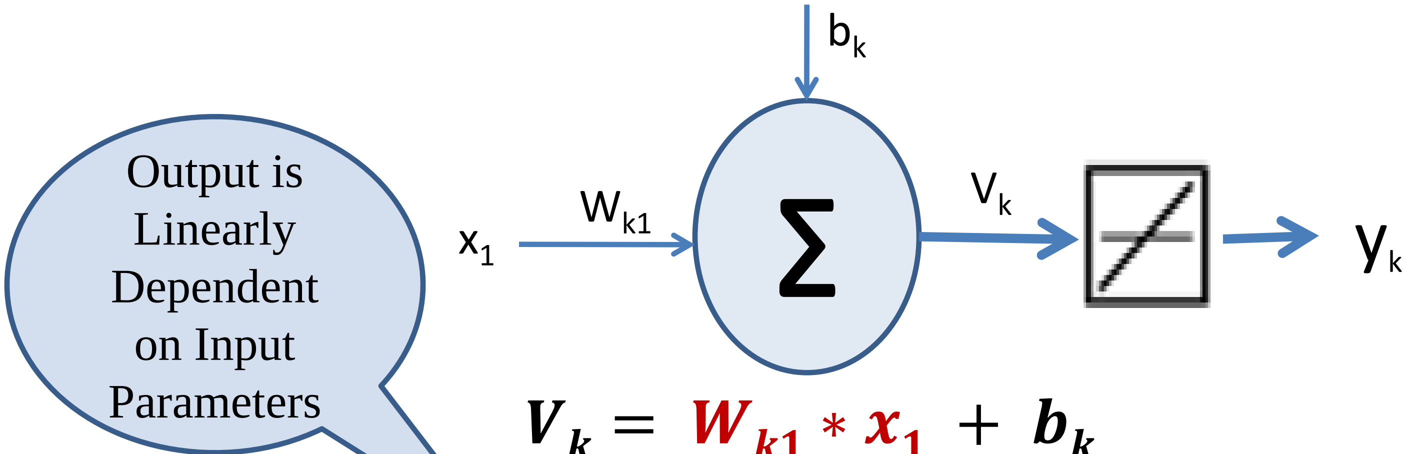
Artificial Neuron



$$V_k = \sum_{j=1}^{n} (W_{kj} * x_j) + b_k$$

$$y_k = f(V_k)$$

Single Neuron Model



$$V_k = W_{k1} * x_1 + b_k$$

$$y_k = f(V_k) = W_{k1} * x_1 + b_k$$

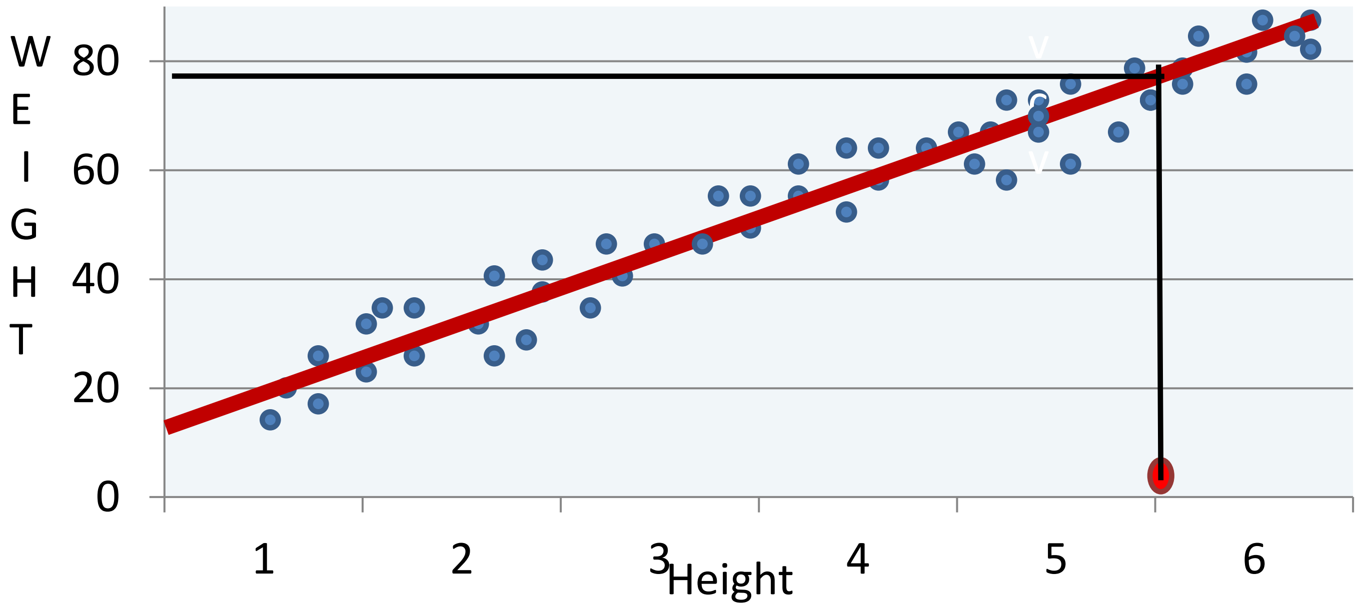
Single Neuron Model

- **Application**

y=mx+c

Where m=Slope Of Straight Line

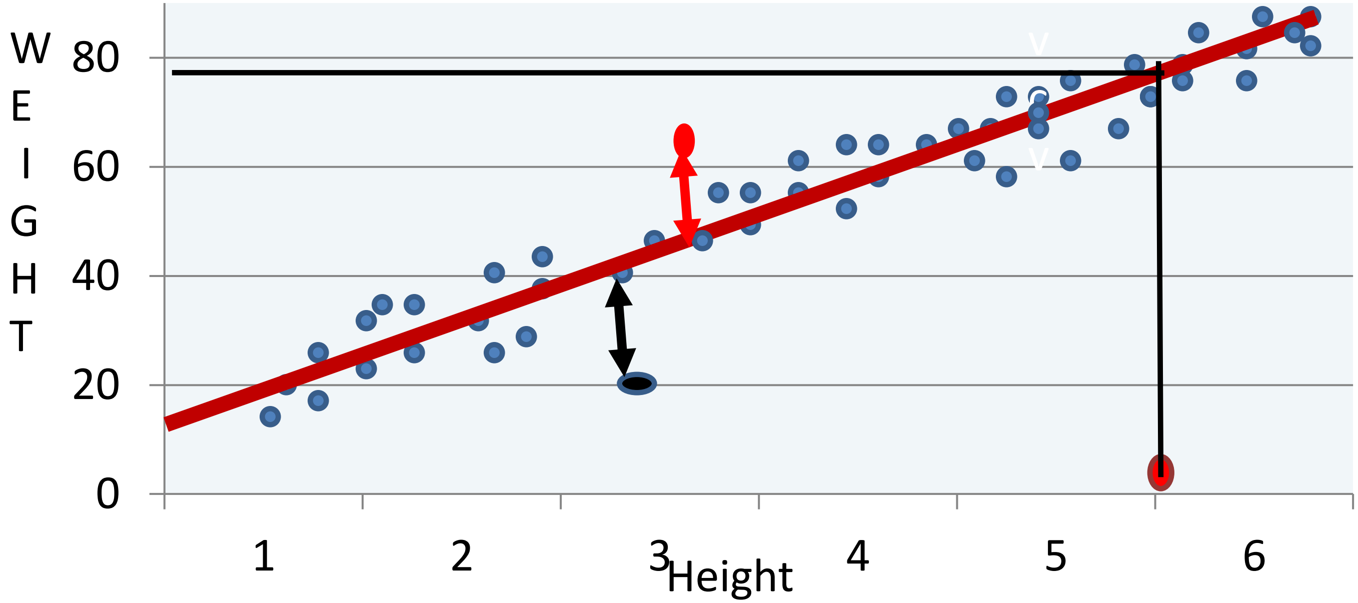
X=Height c=Intercept y=Weight



Single Neuron Model

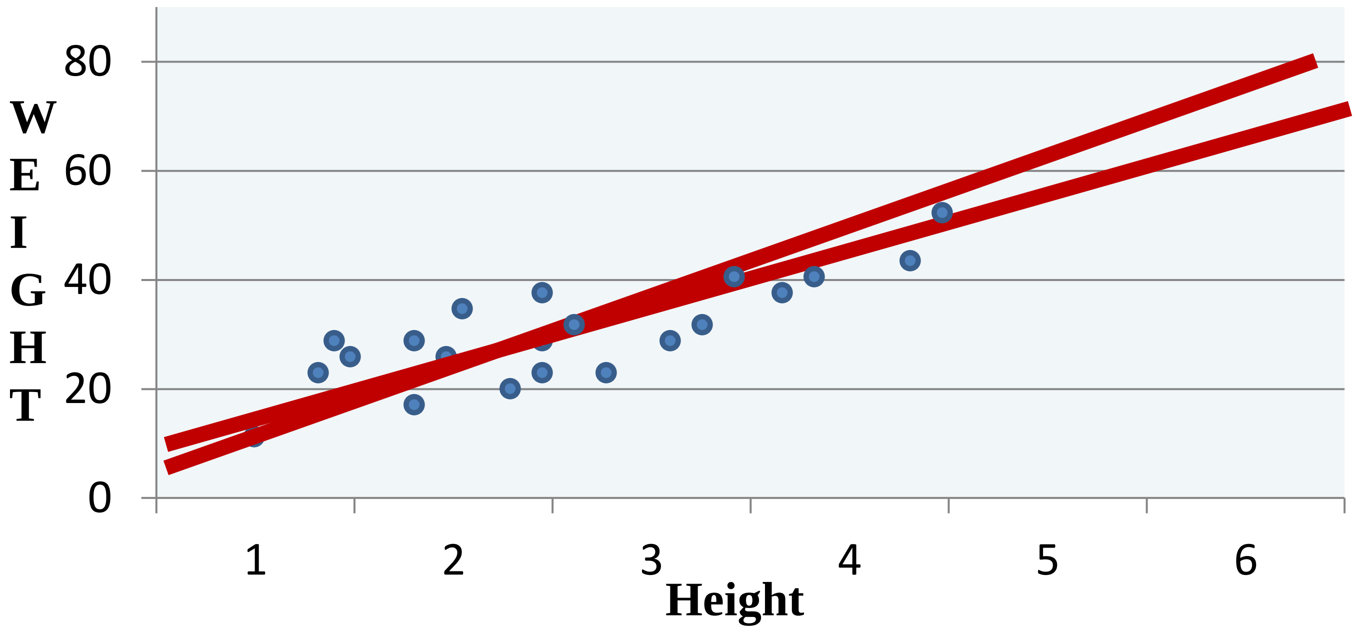
Error Calculation

- The error E_i=(Actual Value – Predicted value)=$(Ti - y_i)$
- For making +ve= $E_i = (T_i - y_i)^2$ [Error for i^{th} input instance]



Linear Neural Network

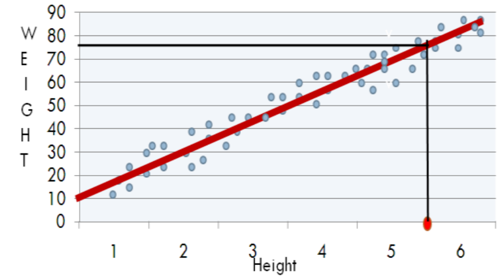
- **<u>Error Calculation</u>**
  - It is done to adjust the slope(m) and intercept for better fitting next time.

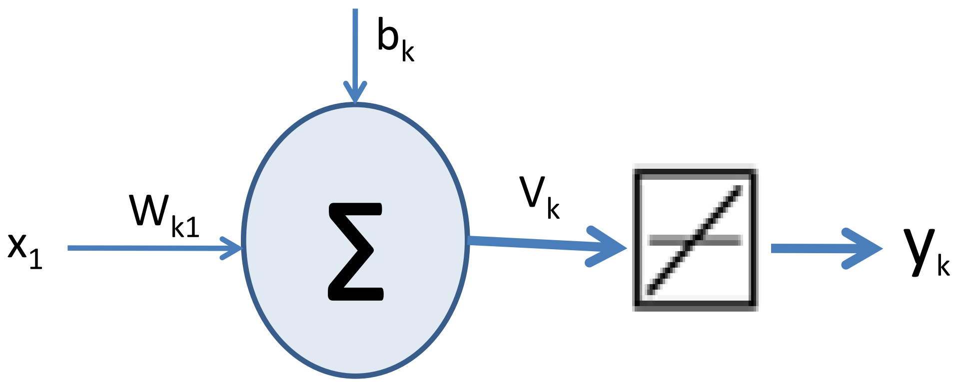


Linear Neural Network

$$y_k = W_{k1} * x_1 + b_k$$

$$y = m * x + c$$



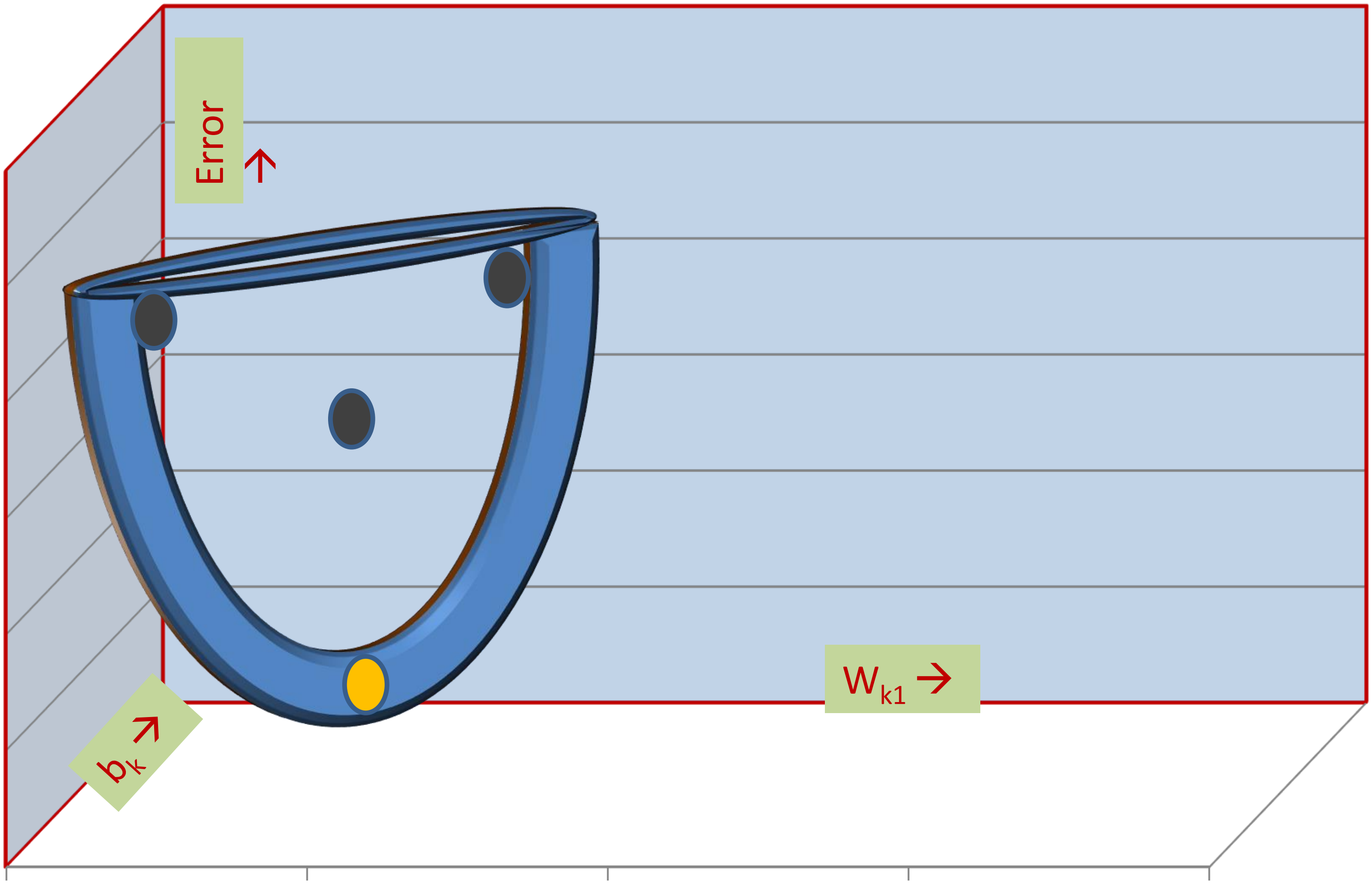


Output is Linearly Dependent on Input Parameters

$$V_k = W_{k1} * x_1 + b_k$$

$$y_k = f(V_k) = W_{k1} * x_1 + b_k$$

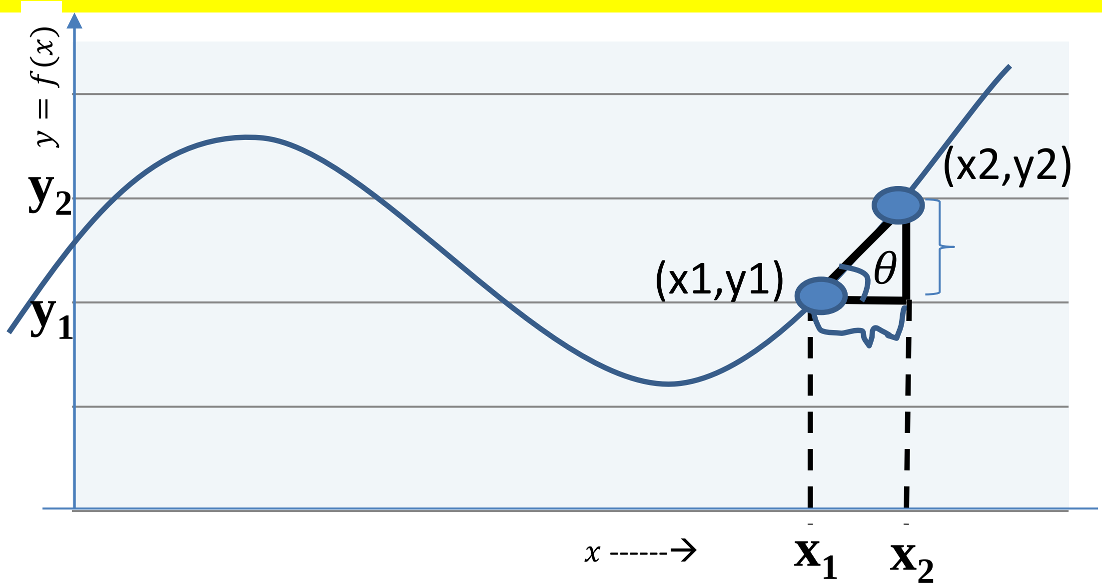
Plotting Error



Differentiation…

$$y = f(x)$$

$$\frac{dy}{dx} = \frac{df}{dx} = y' = f'$$



How much does y change as x changes= $\frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{p}{b} = \tan(\theta)$

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

Differentiation…

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

As $\Delta x \rightarrow 0$ we obtain a tangent at x.

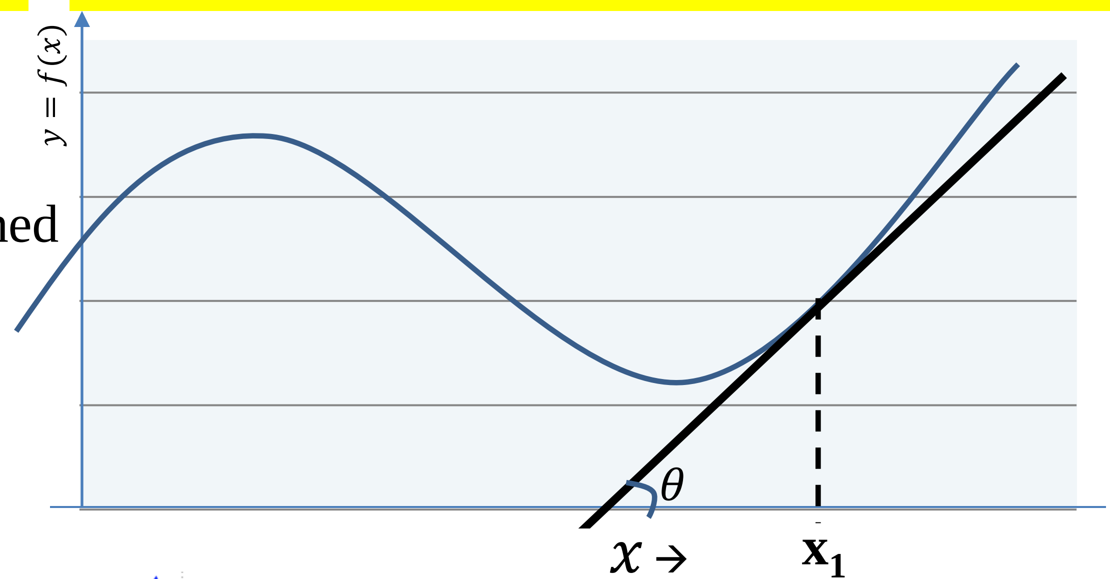
$\frac{dy}{dx} = \tan(\theta)$=slope of the tangent at x=x_1

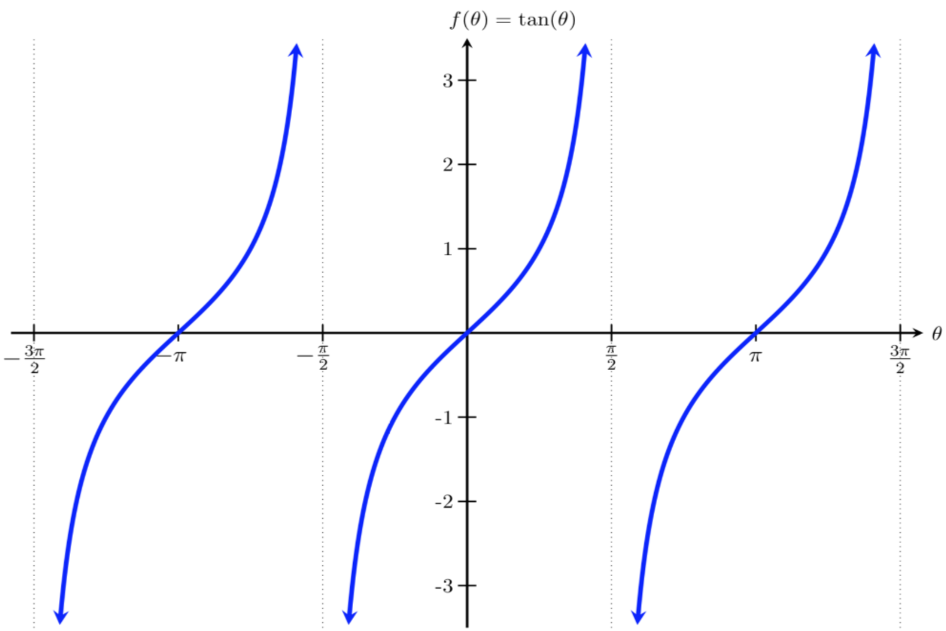
$\frac{dy}{dx} =$ Slope of the tangent to x-axis at x=x_1

Differentiation…

$0 < \theta < 90$ tan(θ)= +ve

90 tan(90)=Undefined

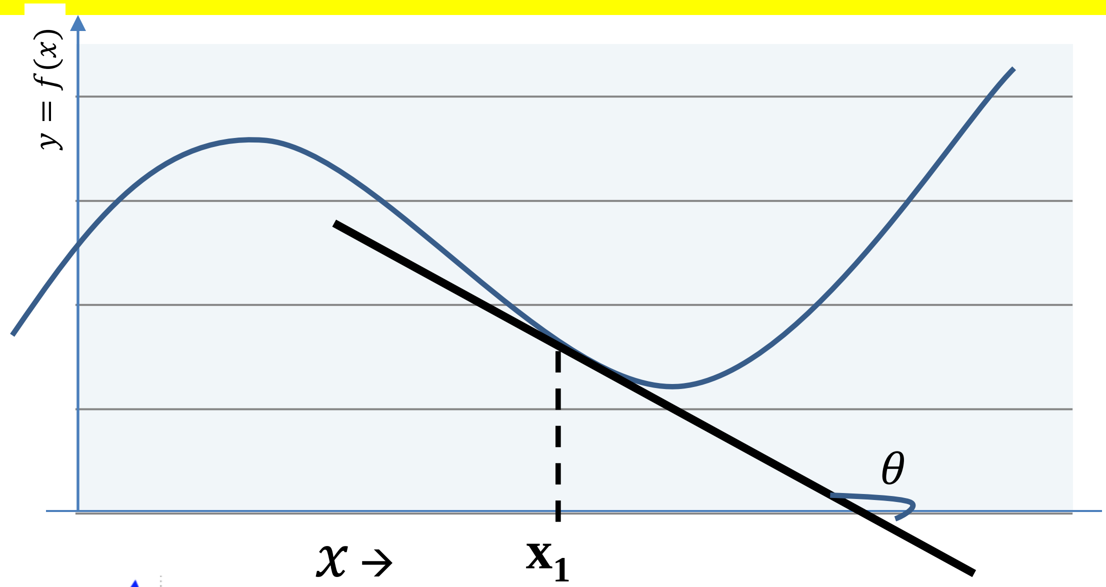


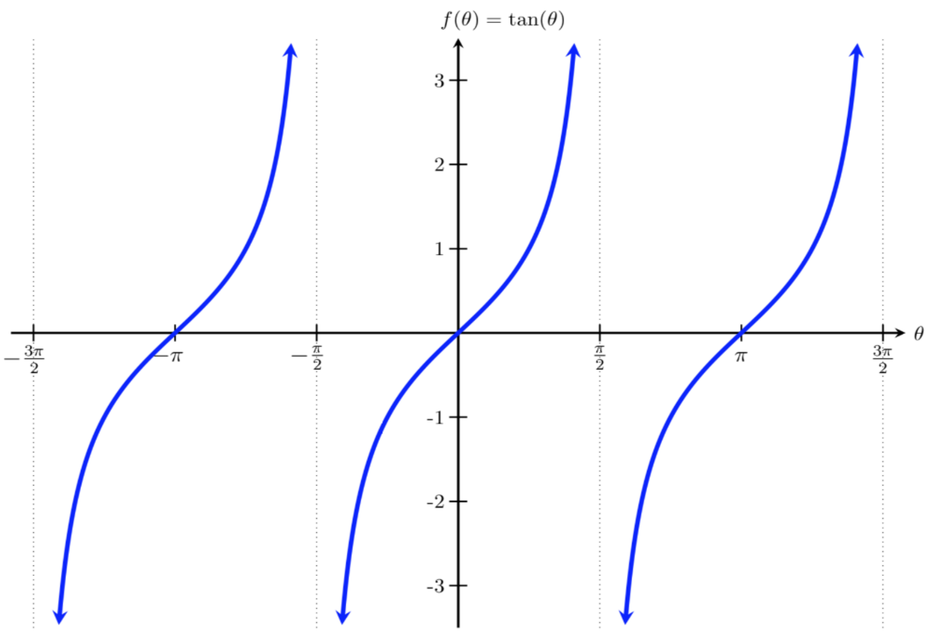


When theta > 90

Differentiation…

$\theta > 90$ $\tan(\theta)$= - ve

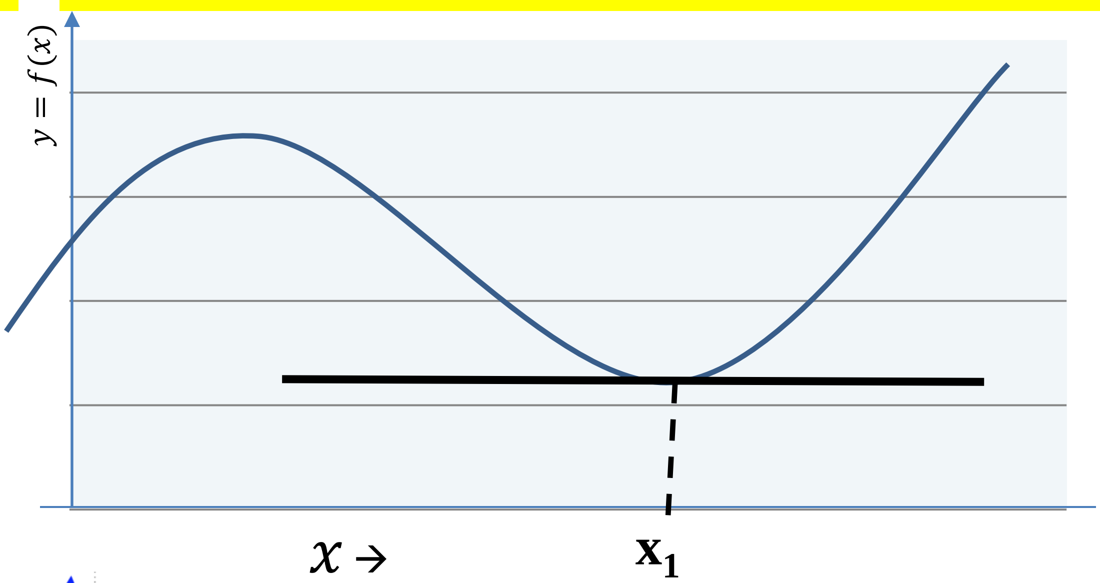


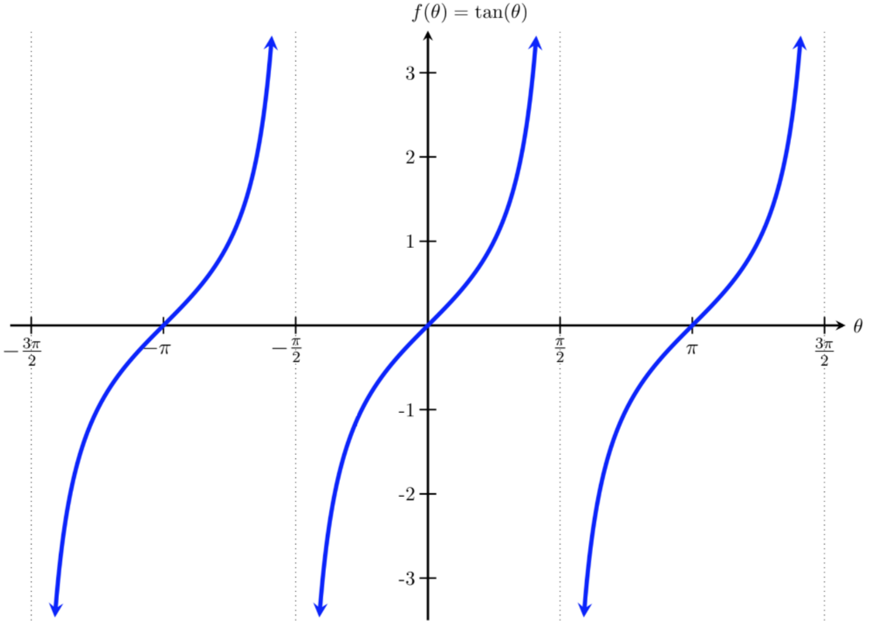


When theta=0

Differentiation…

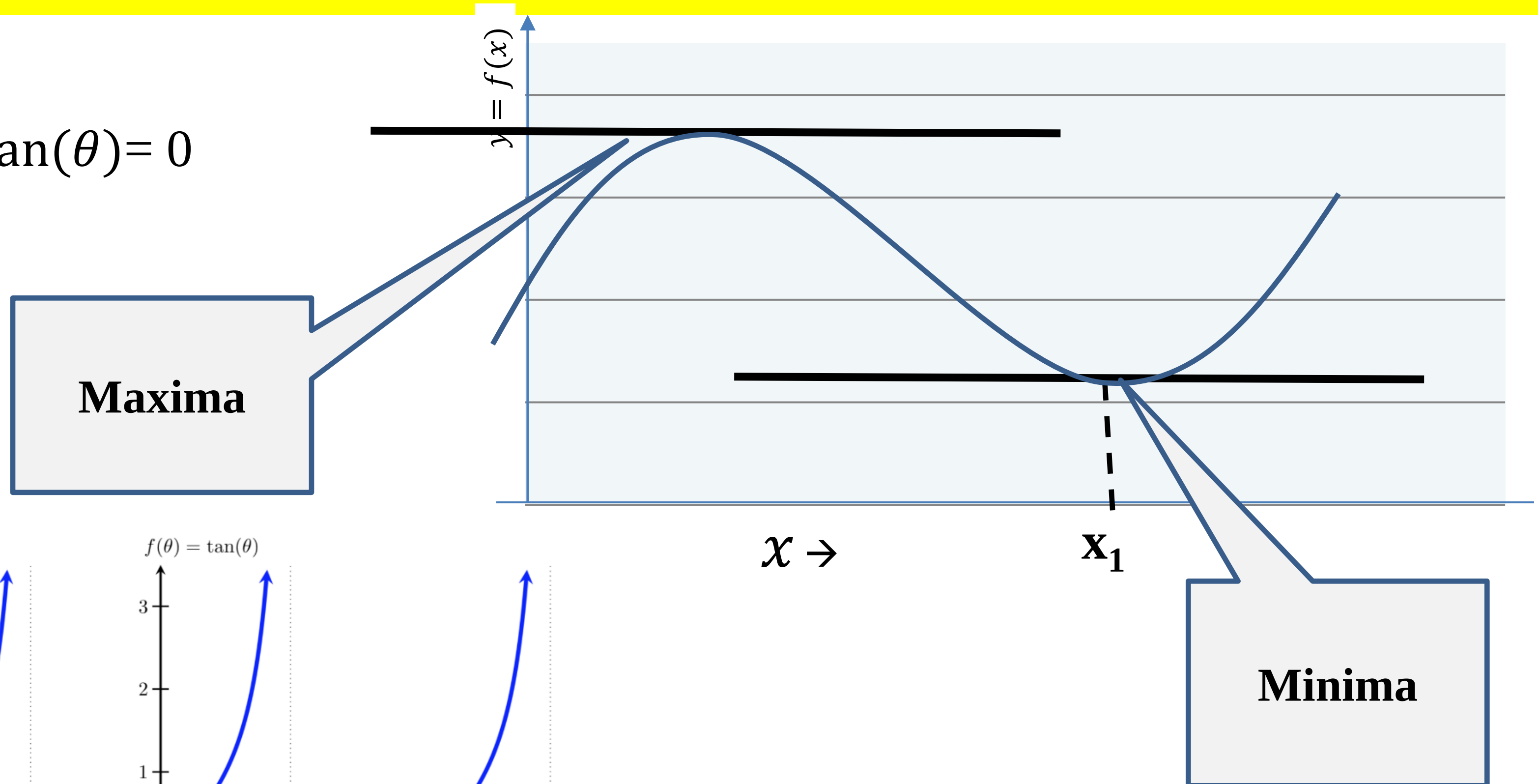
$\theta = 0 \quad \tan(\theta) = 0$

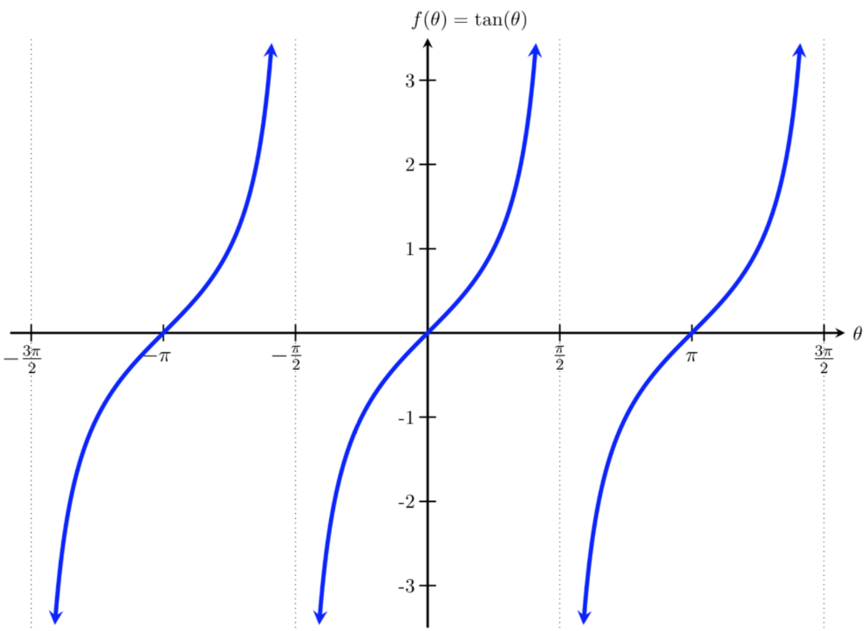




Differentiation…

$\theta = 0$ $\tan(\theta) = 0$





Note: At minima and Maxima the Slope is 0 ➔ $\tan(\theta)$=0 ➔ $\frac{dy}{dx} = 0$

Differentiation…

Distinguishing between a Minima & Maxima

Let f(x)= X^2 - 3X + 2

$\frac{df}{dx}$ =0

2X -3 =0

X=1.5

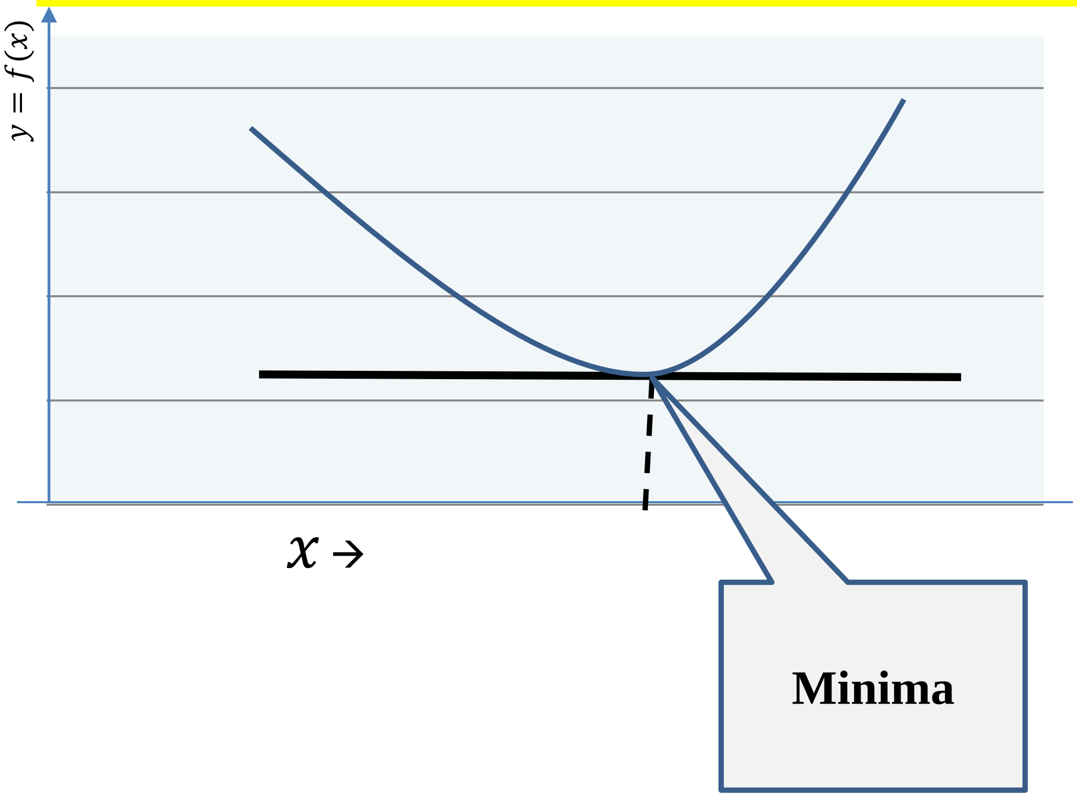
f(1.5)=-0.25

Take a point near 1.5, let X=1

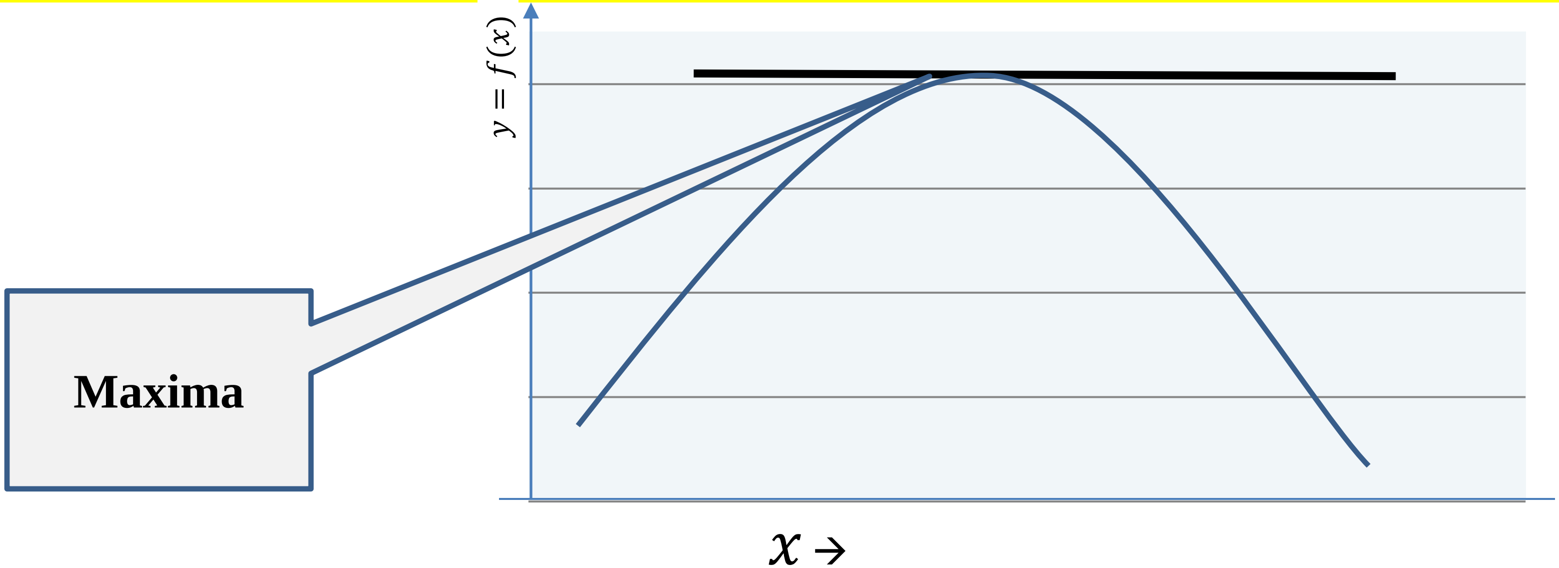
f(1)=1-3+2=0

X=1.5 can't be maxima. It is a minima.

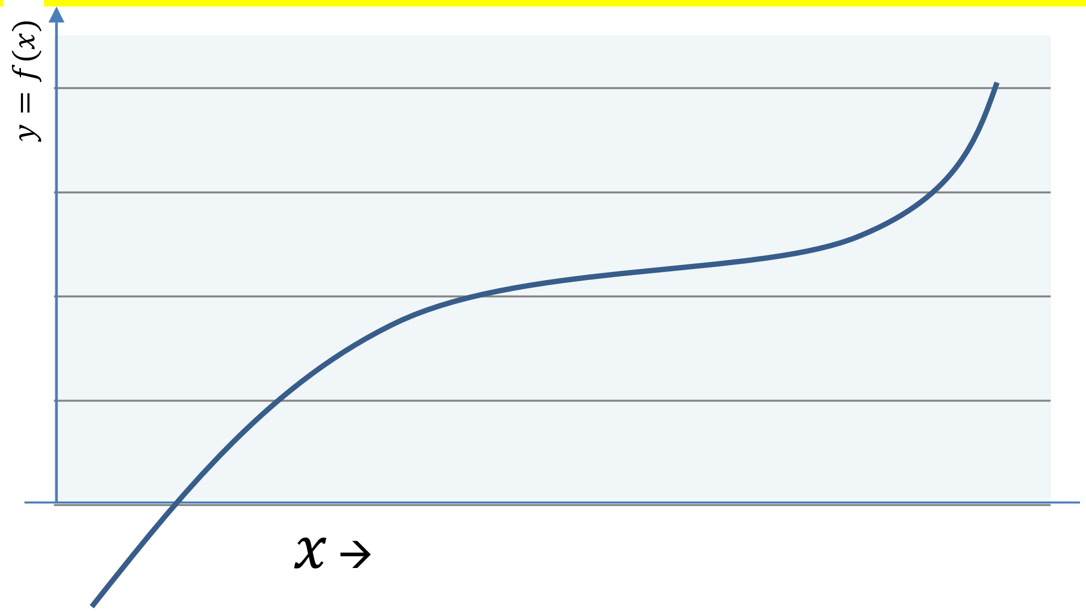
Error Function with Minima and No Maxima



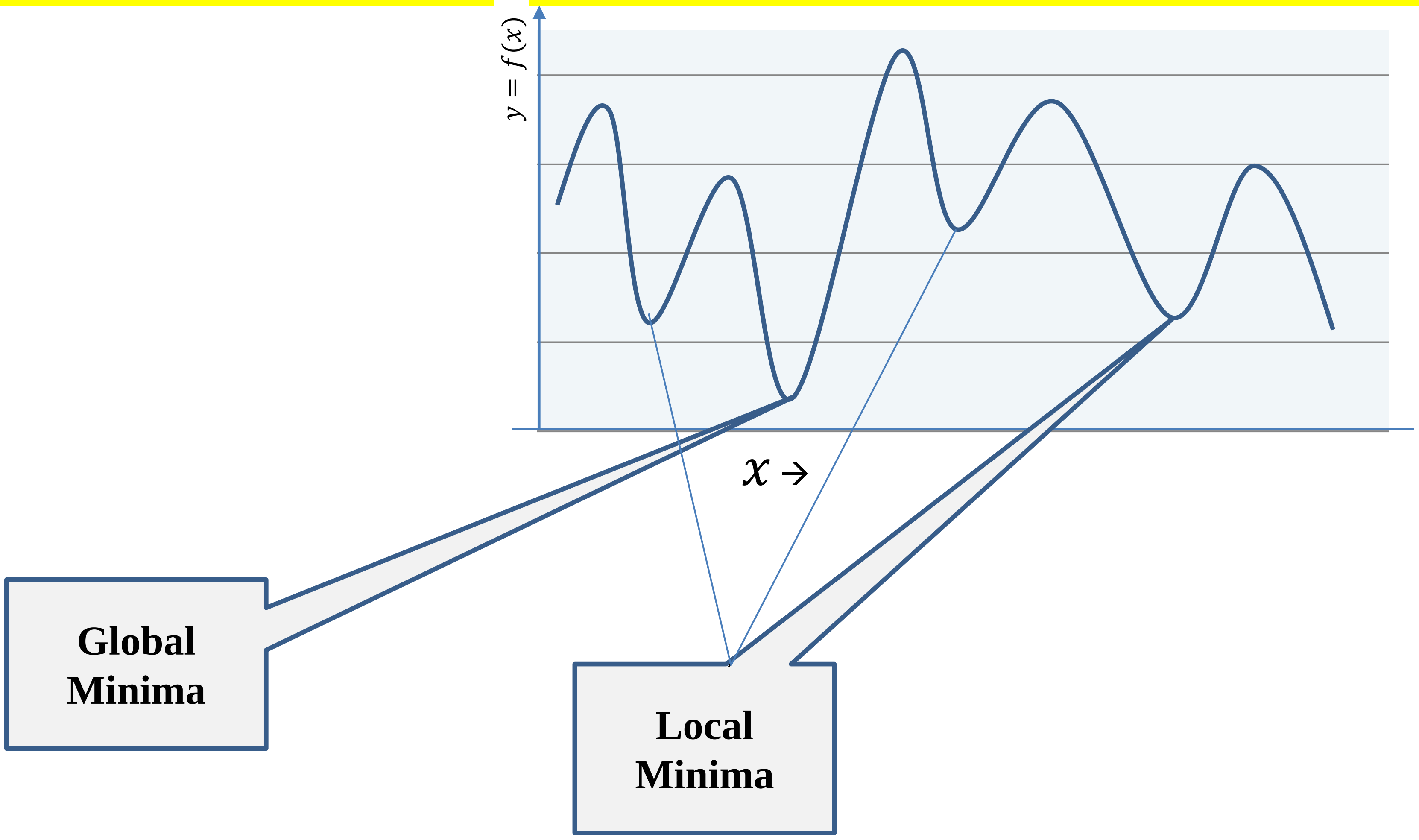
Error Function with a Maxima and No Minima



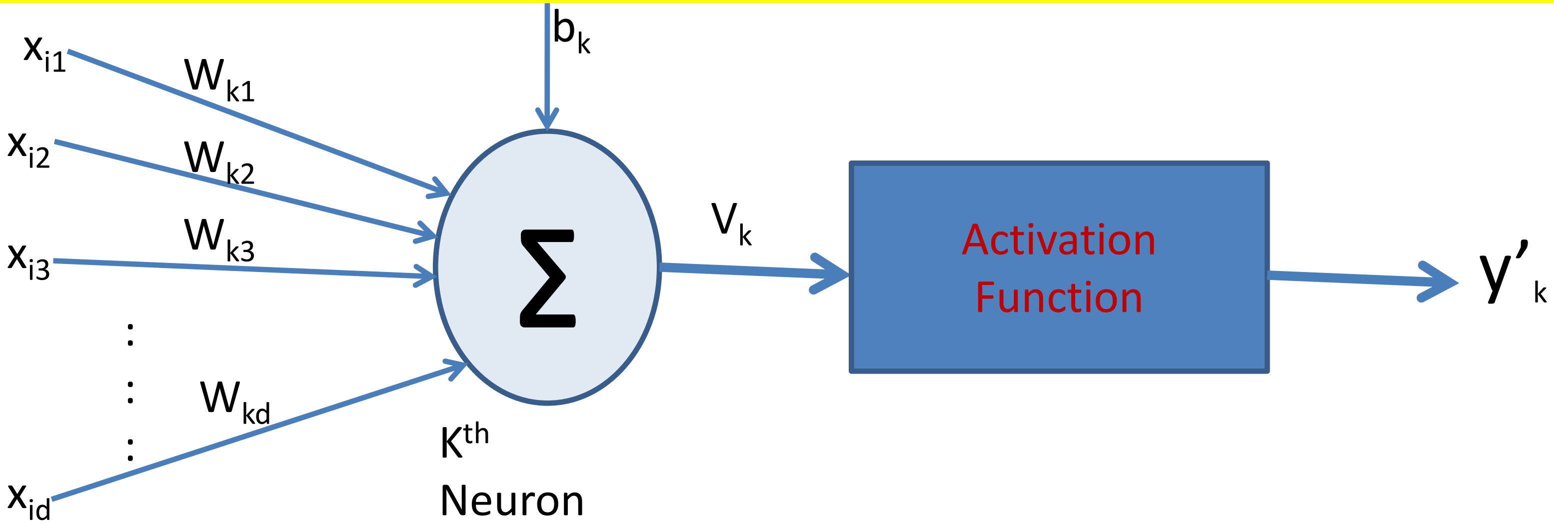
Error Function without a Maxima and Minima



Error Function with multiple Maxima and Minima



TRAINING A SINGLE-NEURON MODEL

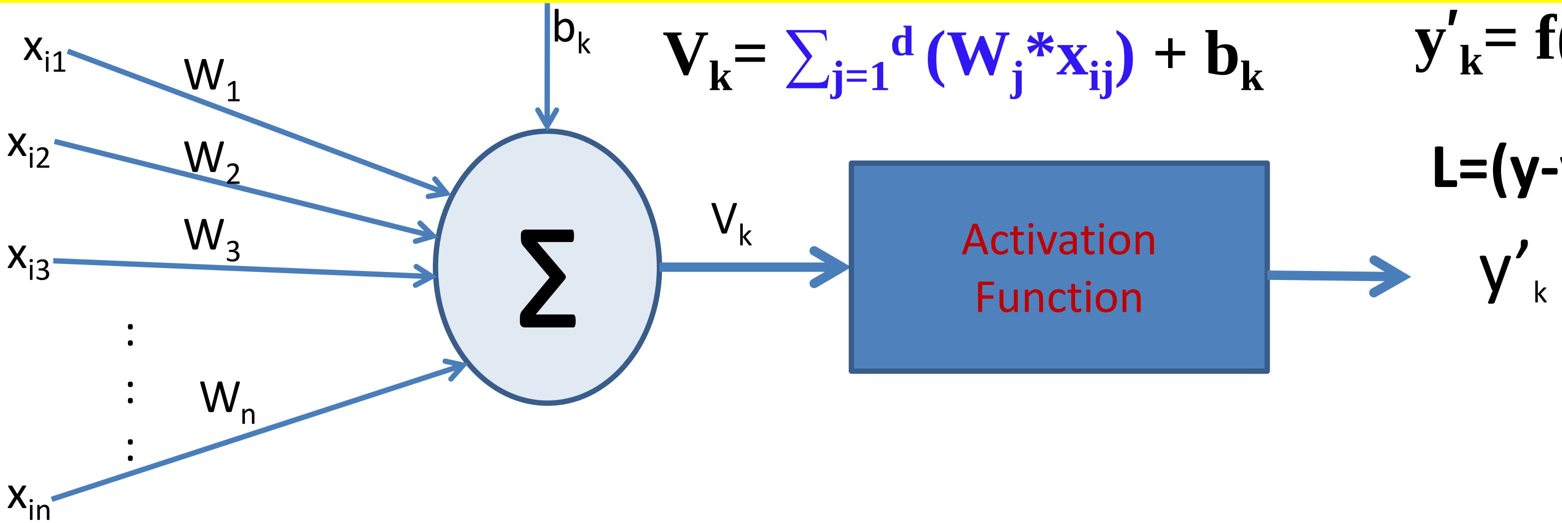


$V_k = \sum_{j=1}^{d} (W_{kj} * x_{ij}) + b_k$ $y'_k = f(V_k)$ $L = \sum_{i=1}^{n} (y_i - f(w^T x_i + b)^2$

- Step-1: Define the loss function $\sum_{i=1}^{n} (y_i - y_i')^2$
- Step-2: Define the optimization $\min_{\vec{w_i}} \sum_{i=1}^{n} (y_i - f(w^T x_i + b))^2 + reg$

Step-3Solve the Optimz Problem

TRAINING A SINGLE-NEURON MODEL



$$V_k = \sum_{j=1}^{d} (W_j * x_{ij}) + b_k$$

$$y'_k = f(V_k)$$

$$L = (y - y')^2$$

- Step-3: Solve the optimization problem
  - Randomly initialize the weights
  - Feed forward the inputs and compute the loss function
  - Update the weights

$$(w_i)_{new} = (w_i)_{old} - \eta \left[\!\left[\frac{dL}{dw_i}\right]\!\right] = \begin{cases} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \frac{\partial L}{\partial w_d} \end{cases}$$

-2(y-y')

x

$$\frac{dL}{dw_1} = \text{-2(y-y')*x}$$

TYPES OF NEURAL NETWORK

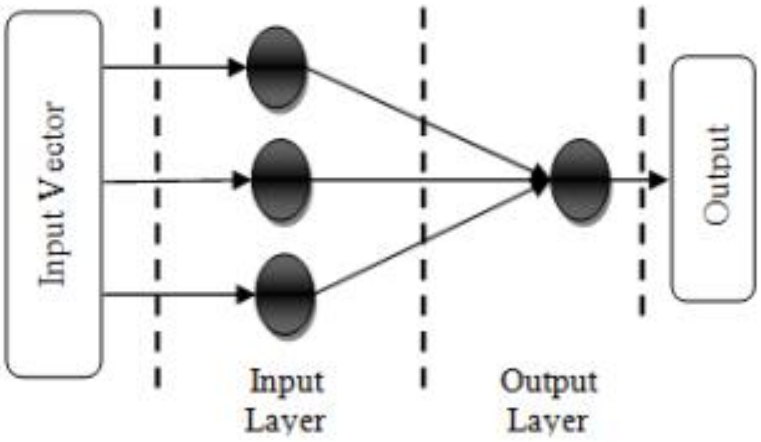


Figure 1.2: Single layer Neural Network

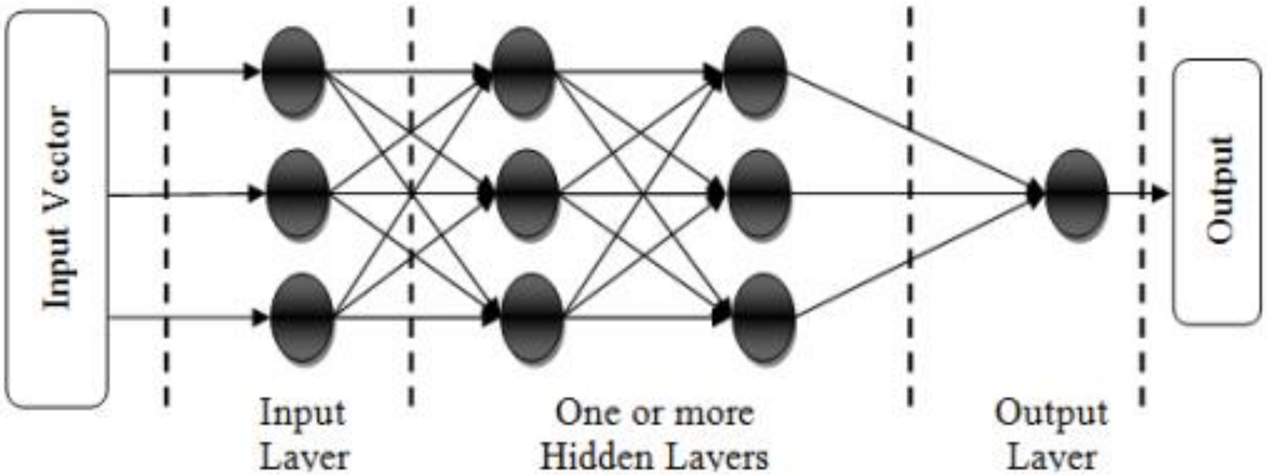
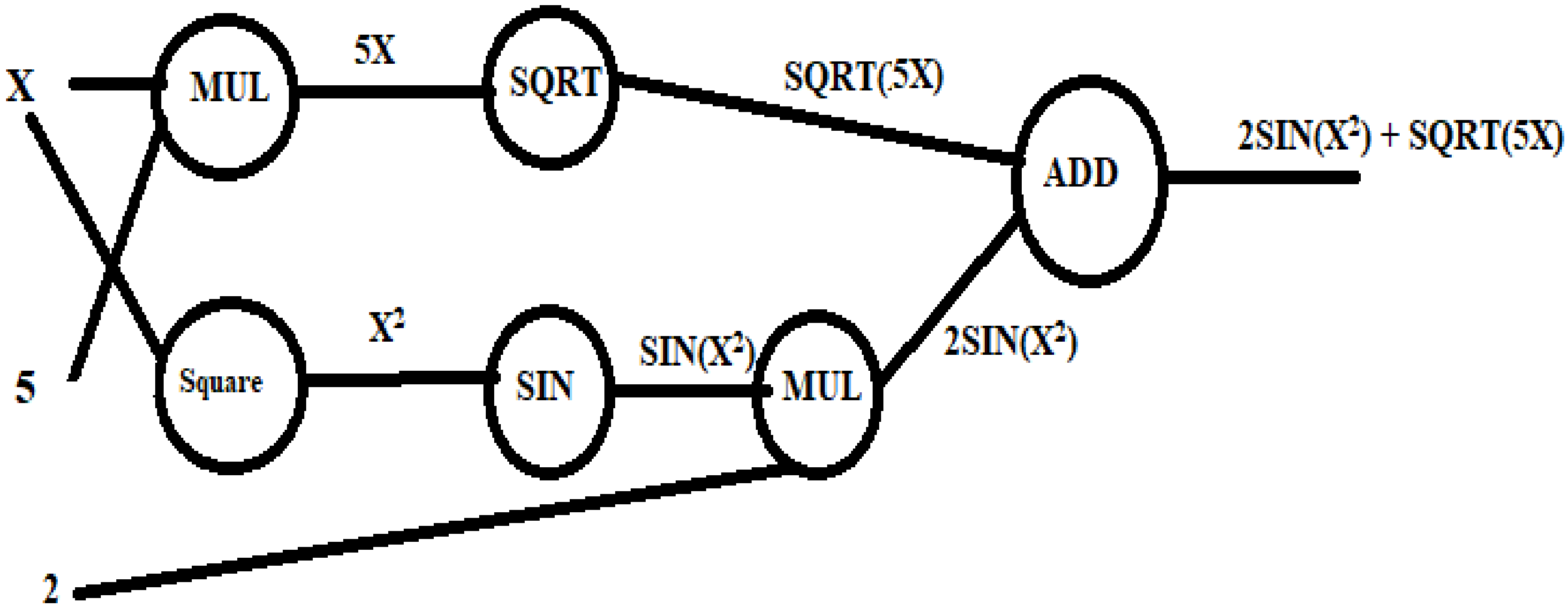


Figure 1.3: Multilayer Neural Network

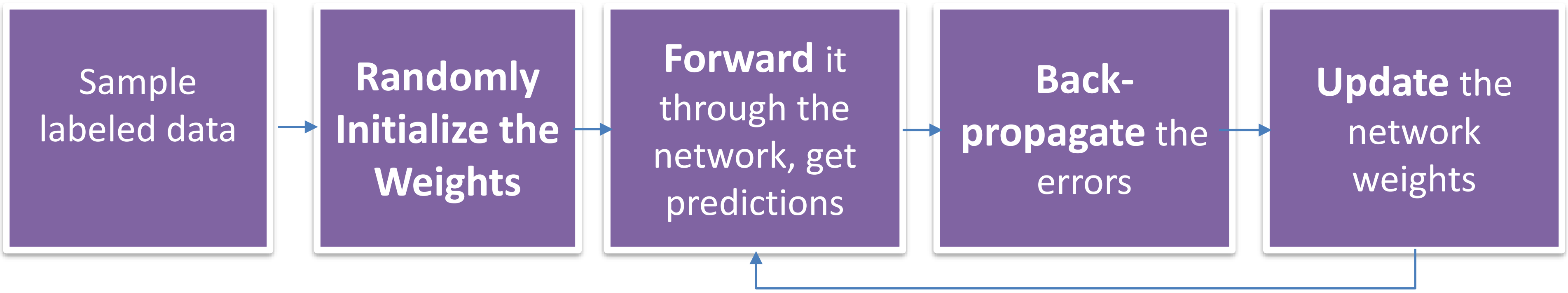
WHY MULTILAYER NEURAL NETWORK?

- **Biological Inspiration**
- **Universal Approximators:** Can approximate any nonlinear function to any desired level of accuracy.
- **Results in Powerful Models**

Graph for 2*sin(x^2)+sqrt(x*5)

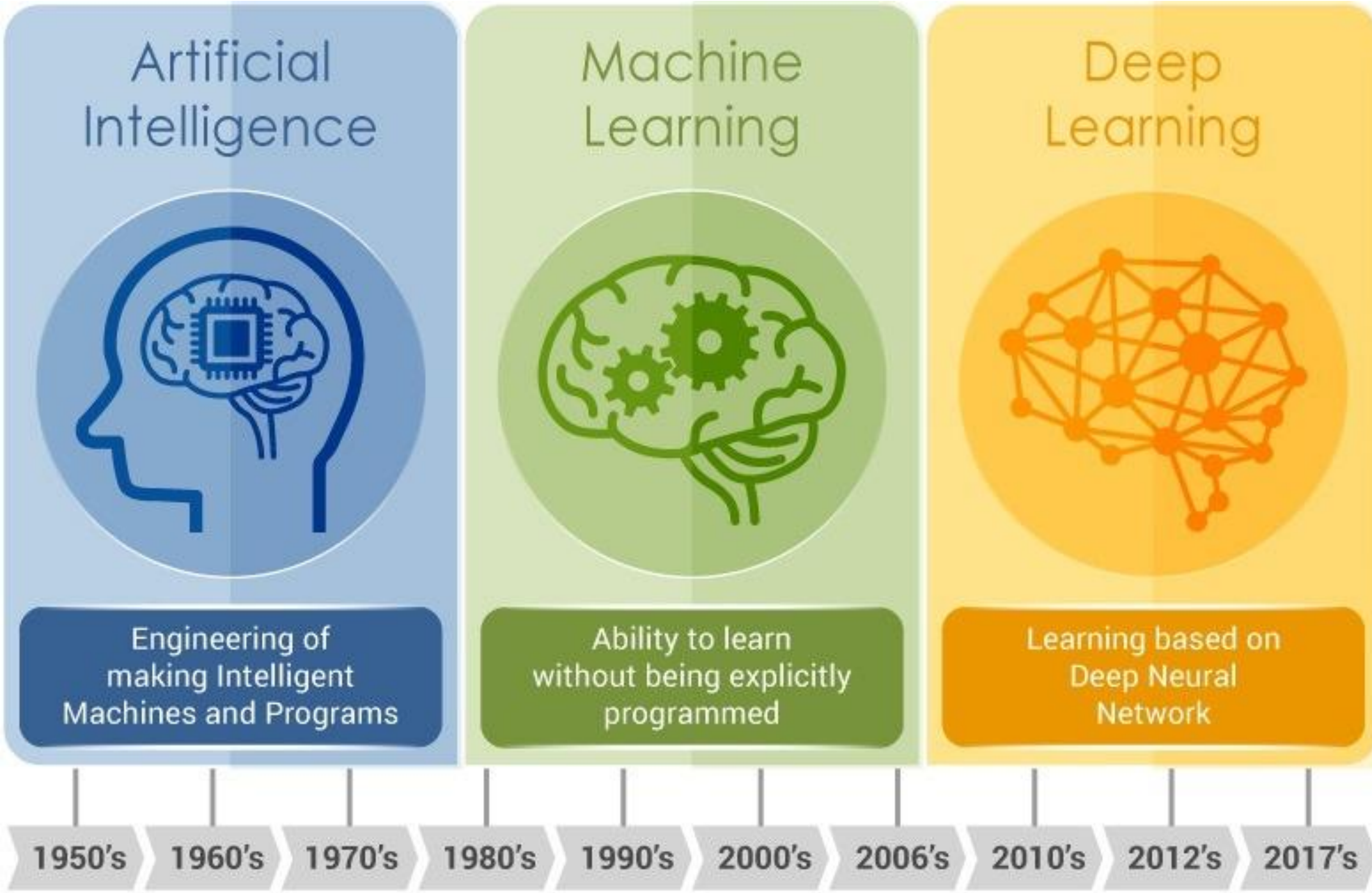


TRAINING MULTILAYER NEURAL NETWORK



- Back-Propagation: Chain Rule + Memoization
  - In Stochastic Gradient Descent (SGD) U take one point (Input Vector)
  - In Mini-Batch SGD, U take a set of points(input vectors)
  - In Gradient Descent, U take all the input vectors

AI vs Machine Learning vs Deep Learning



Deep Learning

- A type of ***machine learning*** based on ***artificial neural networks*** in which *multiple layers of processing* are used to *extract progressively higher level features* from data.

- "Deep Learning with Python" Francois Chollet

DEEP LEARNING APPROACH

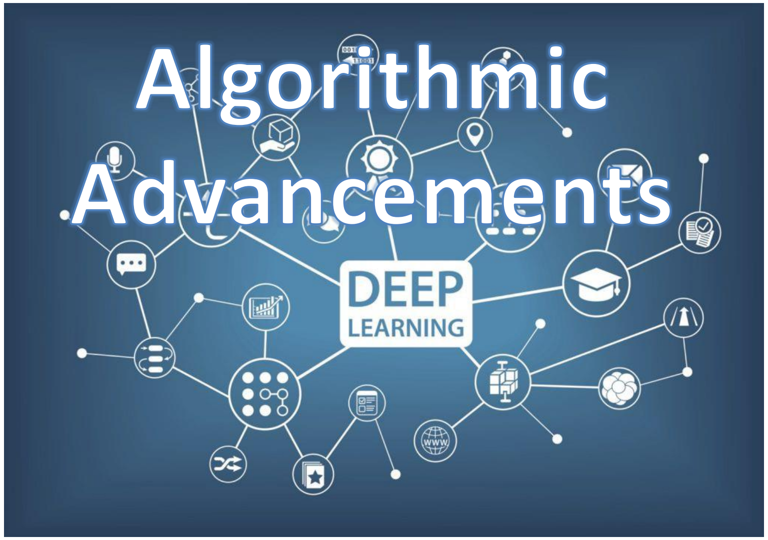
- Standard Approach (Mathematicians)
  - Build new theories
  - Perform Experiments
- New Deep Learning (Engineers way)
  - Given huge amount of computational power
  - People First Experiment and then try to build a theory

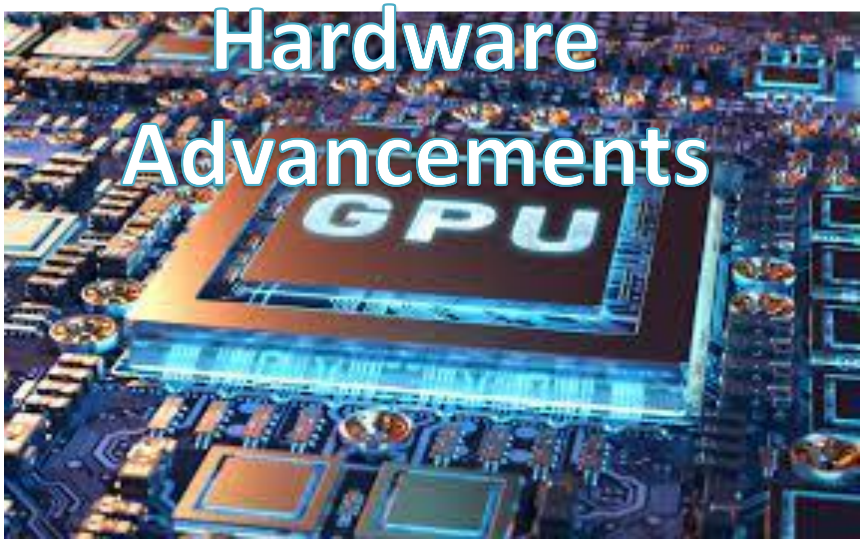
Why Deep Learning ? Why Now ?

- **Computer Vision-** *Convolutional Neural Networks* and *Backpropagation* —well understood since 1989
- **Time Series Forecasting-** *Long Short-Term Memory* — well understood since 1997

- "Deep Learning with Python" Francois Chollet

Why Deep Learning ? Why Now ?



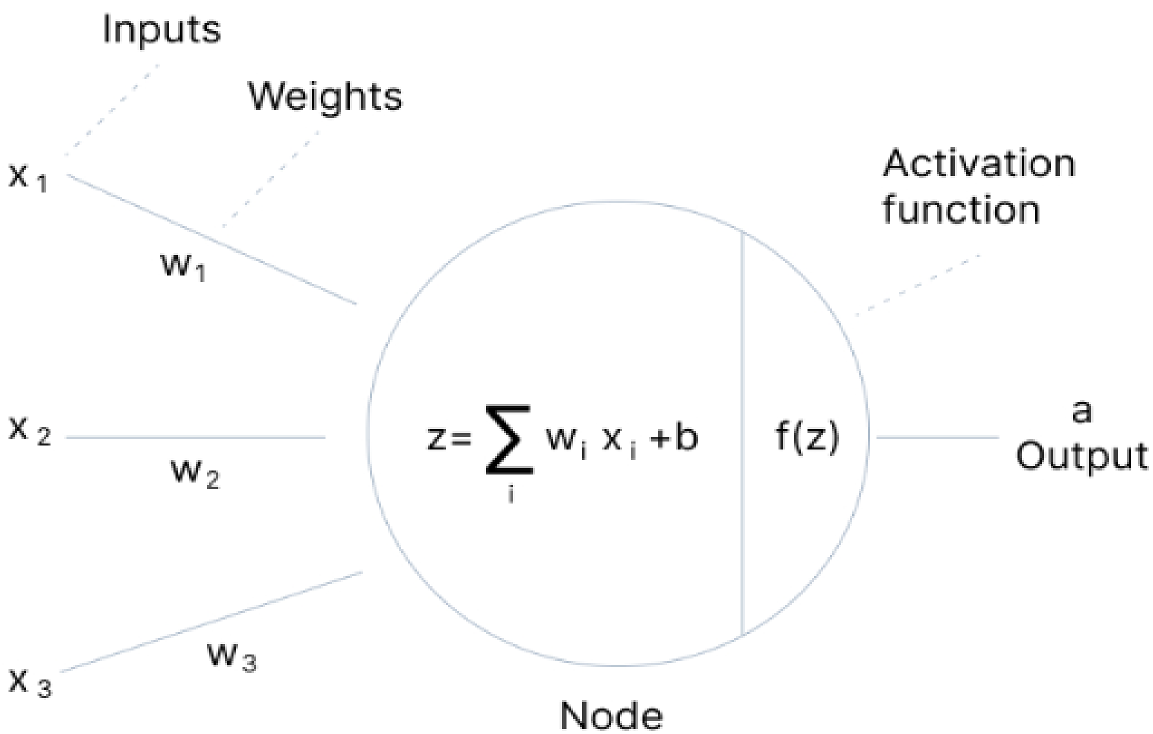


Algorithmic Advancements…

- 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…

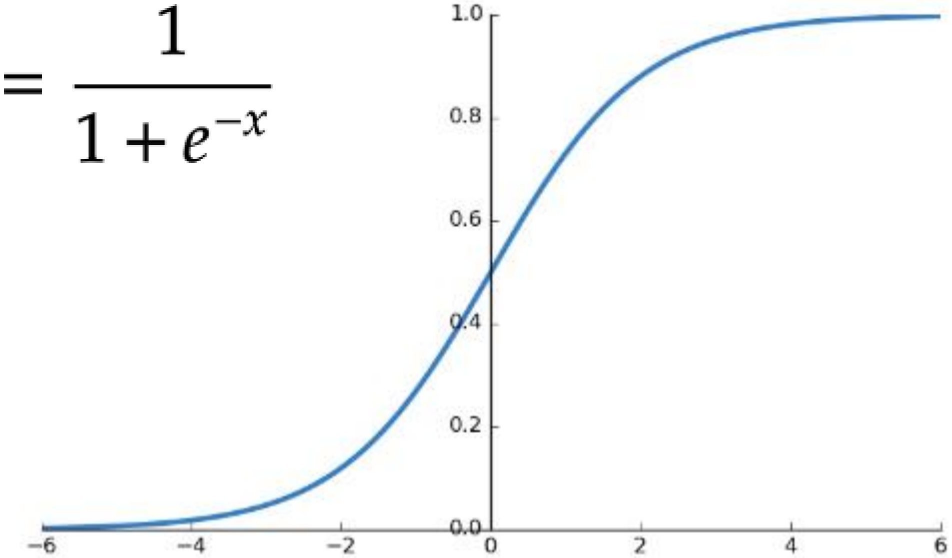
- An *Activation Function (Transfer Function)* maps the weighted summation of inputs to output.
- An Activation function is used to add ***Nonlinearity so that the network can learn complex patterns.***

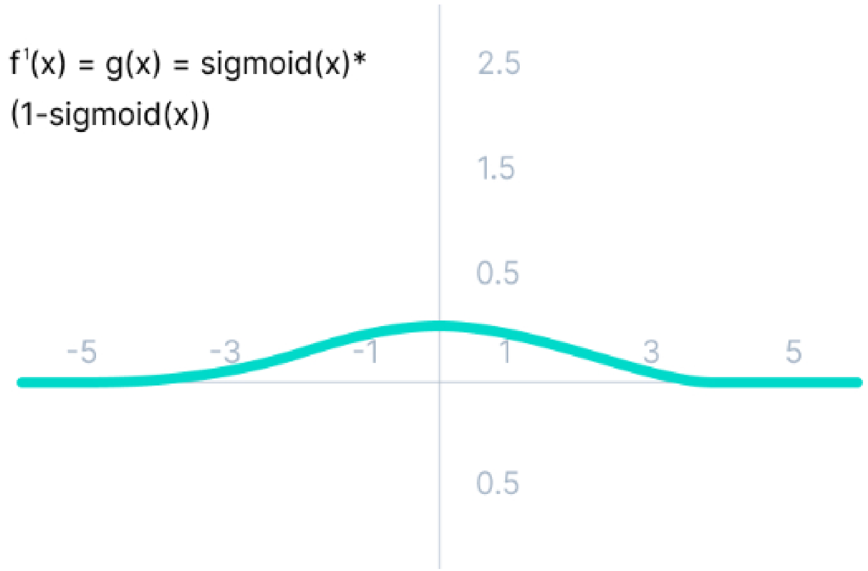


Sigmoid Activation Functions

- **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.

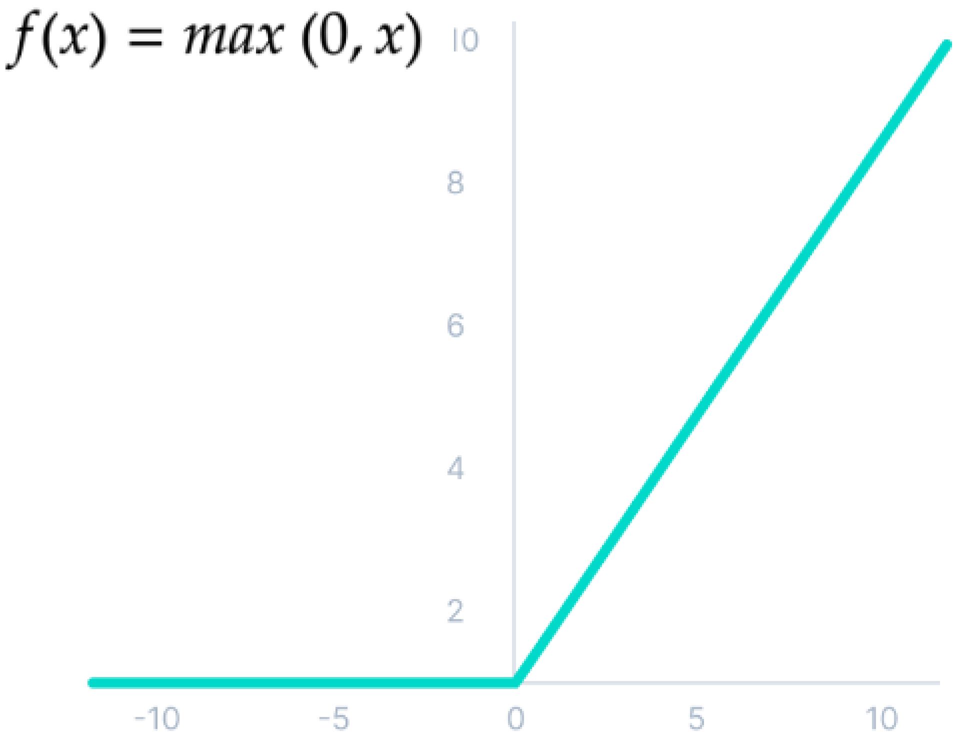
$$\frac{dL}{dw} = \frac{dL}{df_1} \times \frac{df_1}{df_2} \times \frac{df_2}{df_3} \times \cdots \ldots \ldots \ldots \ldots \ldots \times \frac{df_n}{dw}$$

ReLU Activation Function

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

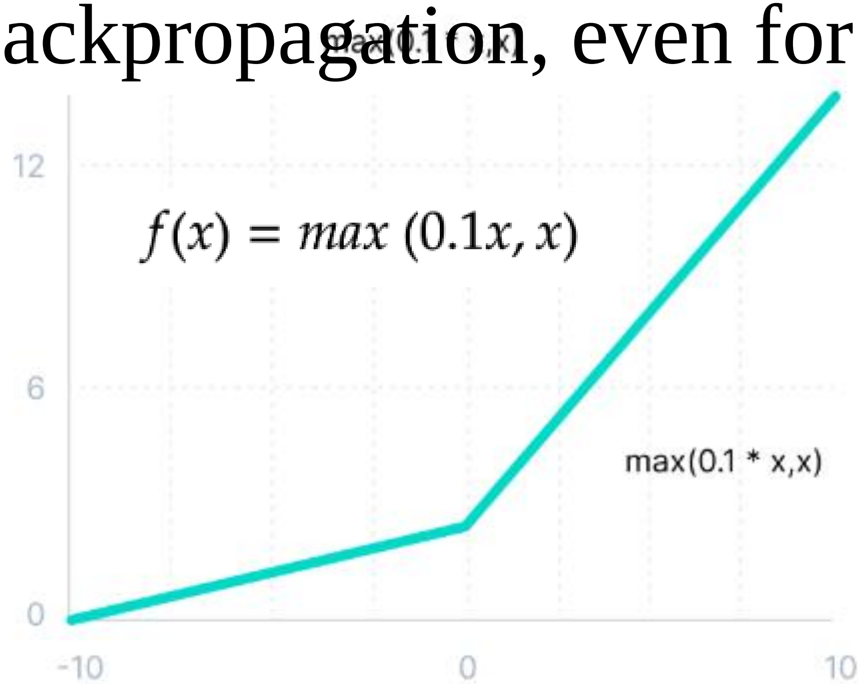
  = 0,   when x<=0
- Avoids Vanishing Gradient Problem.
- Derivative is Simple
  - f'(x)= 1 for x>=0

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

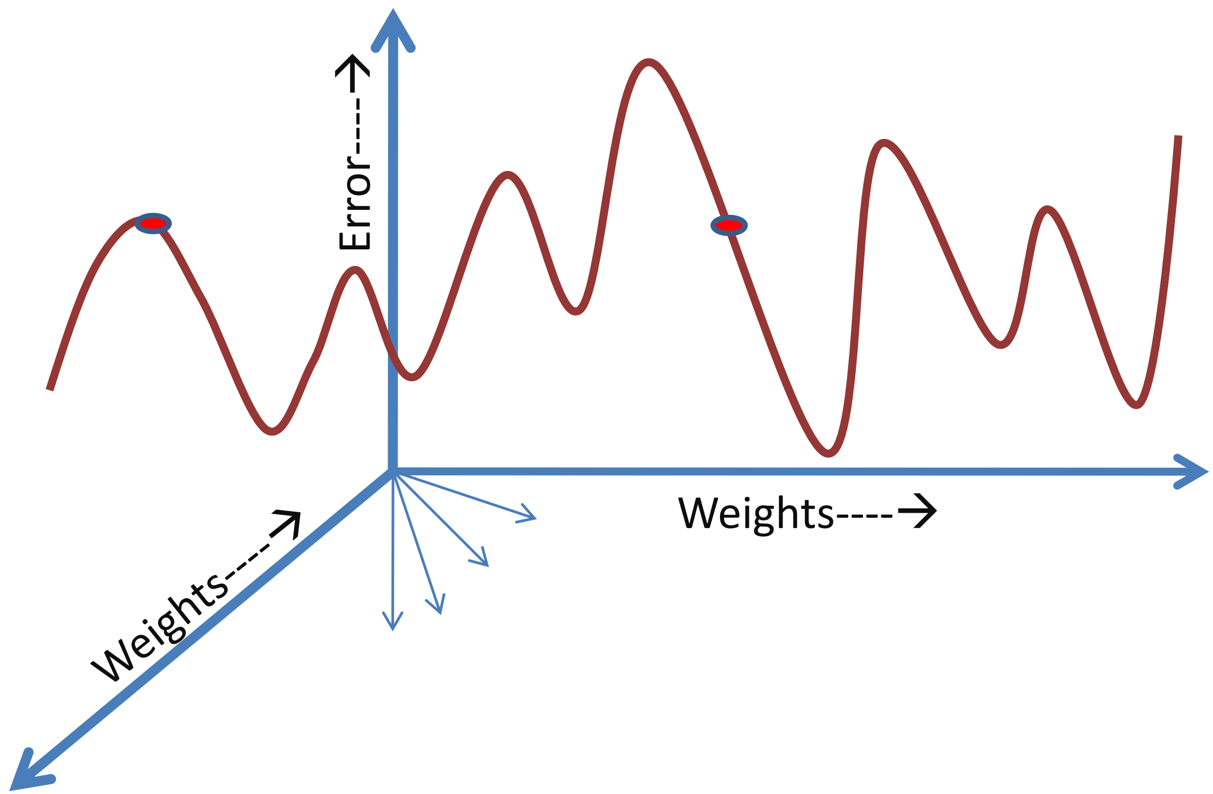


Leaky ReLU Activation Function

- 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

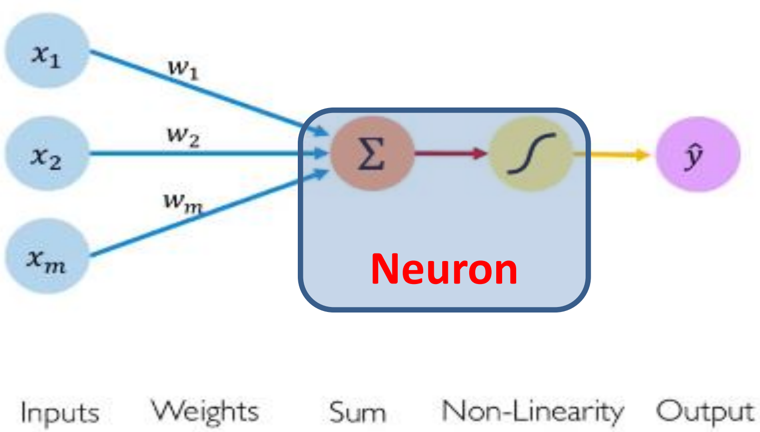


WEIGHT INITIALIZATION

- 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

- Better Strategies obtained from large experiments
  – Initialize weights based on Fan-in and Fan-out
  – Initialize your weights from a uniform distribution
    - $\left[-\frac{1}{\sqrt{fanin}}, \frac{1}{\sqrt{fanin}}\right]$
  – Works well for sigmoid activation function



WEIGHT INITIALIZATION

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

- First Variation – $W_{ij} = N\left(0, \sigma_{ij}\right), \quad \sigma_{ij} = \frac{2}{Fanin + Fanout}$
- 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 – $W_{ij} = N(0, \sigma_{ij}), \quad \sigma_{ij} = \sqrt{\dfrac{2}{Fanin}}$
- Second Variation– $W_{ij} = U\left(-\dfrac{\sqrt{6}}{\sqrt{Fanin}}, \dfrac{\sqrt{6}}{\sqrt{Fanin}}\right)$

BIAS-VARIANCE TRADE-OFF

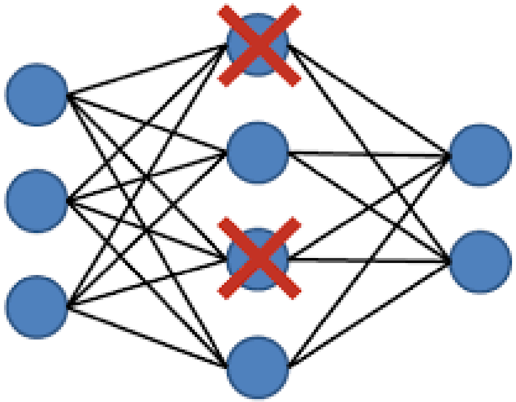
No. of Layers Increases → More No. of Weights → Chances to Overfit is High → Problem of High Variance

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

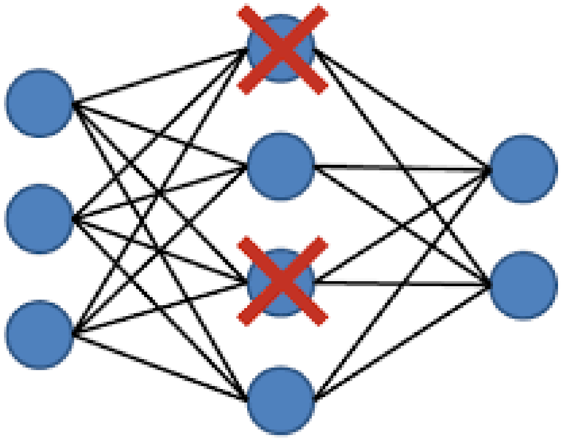
- Deep NN tend to overfit because of many layers and weights
- For this dropout and regularization is needed
- In Dropout, a certain percentage of inputs and hidden layer neurons are dropped out for an iteration
- Some call it as drop out network or layer.



‘

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



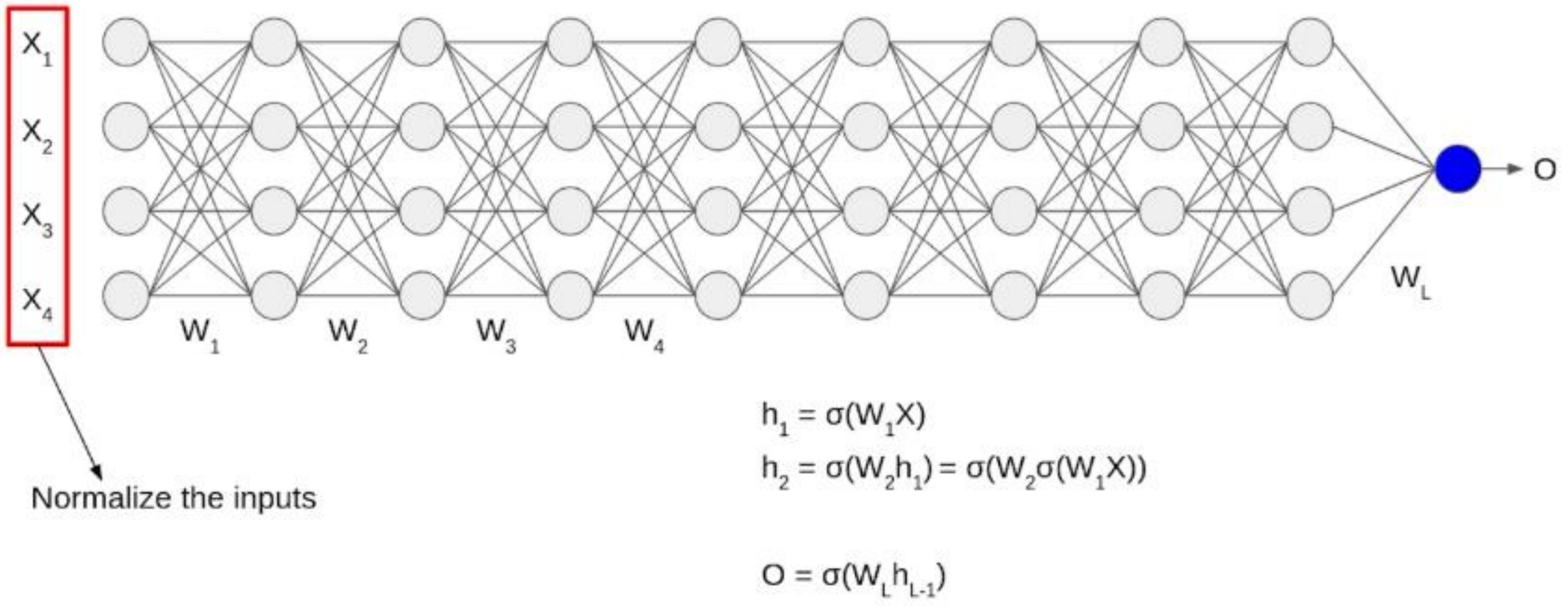
'

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: $N_i = \text{Min}_N + \frac{T_i - \text{Min}_T}{\text{Max}_T - \text{Min}_T} \times (\text{Max}_N - \text{Min}_N)$
  - Vector: $N_i = \frac{T_i}{\sqrt{\sum_{j=1}^{k} T_j^{\,2}}}$
  - Z-Score: $N_i = \frac{T_i - \mu_T}{\sigma_T}$

BATCH NORMALIZATION

- Motivation



Batch Normalization

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

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

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

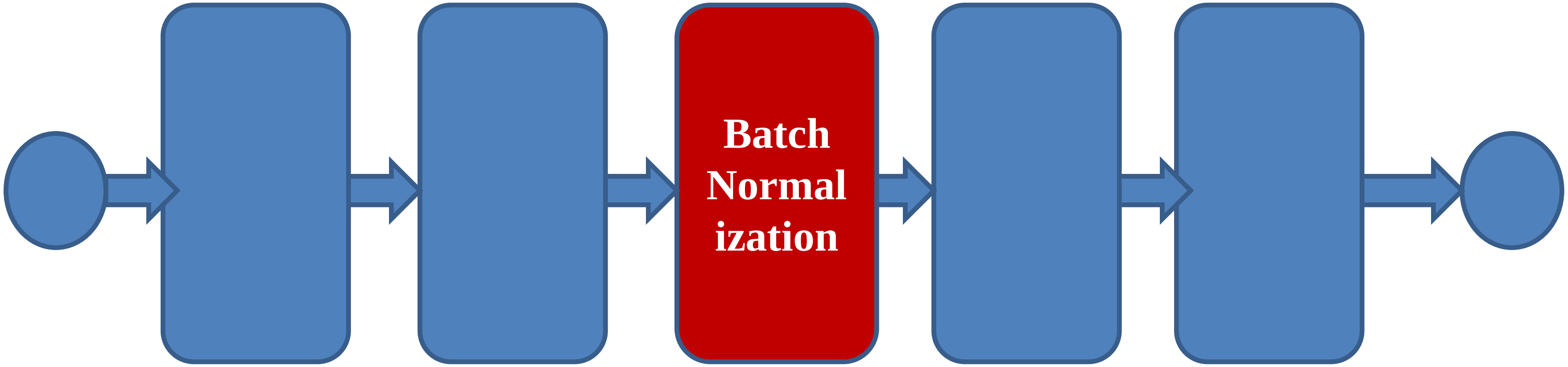
- Where m: Number of Neurons at h_i

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

- Where γ and β are hyper parameters.

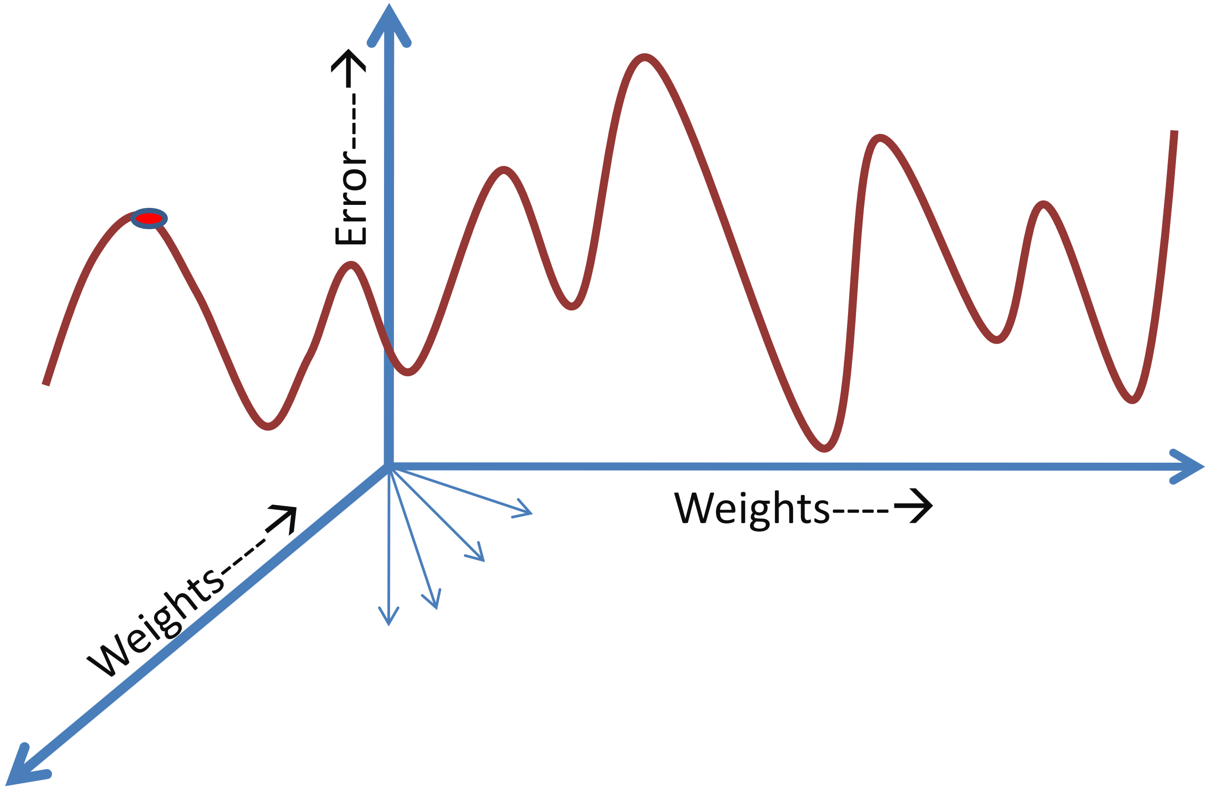
BATCH NORMALIZATION

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



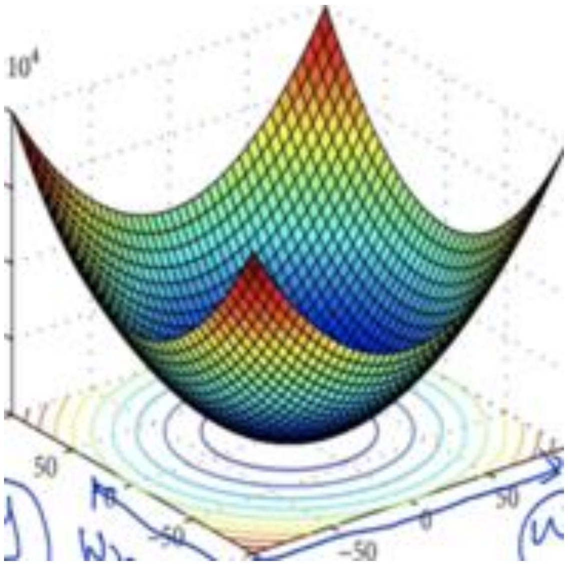
OPTIMIZERS

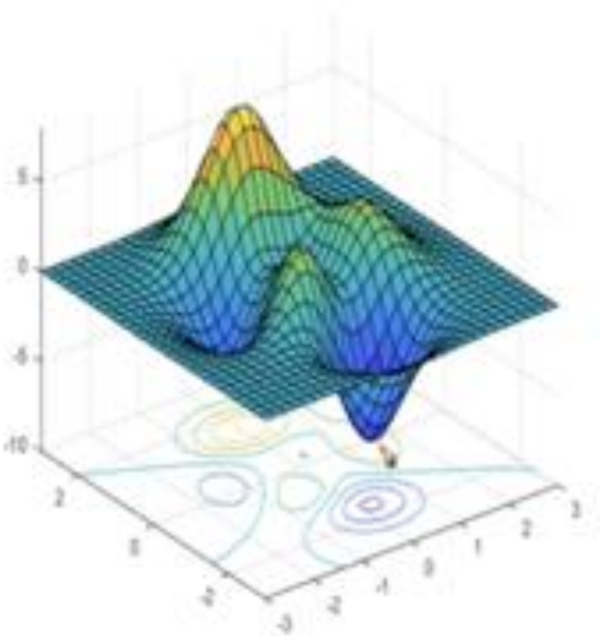
- At minima, maxima and saddle point, u have the gradient as Zero.

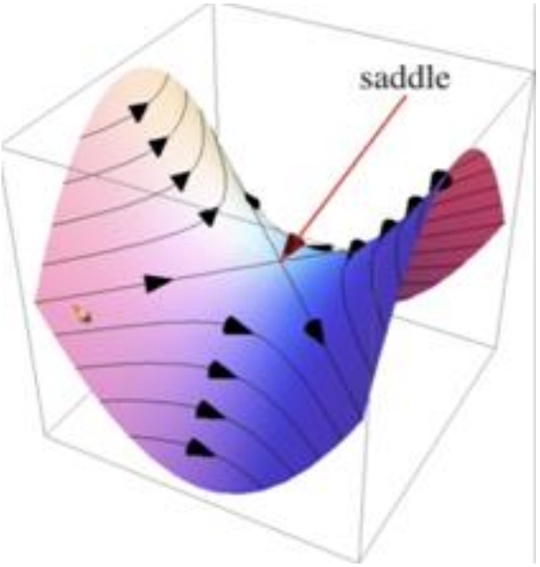


OPTIMIZERS

- 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

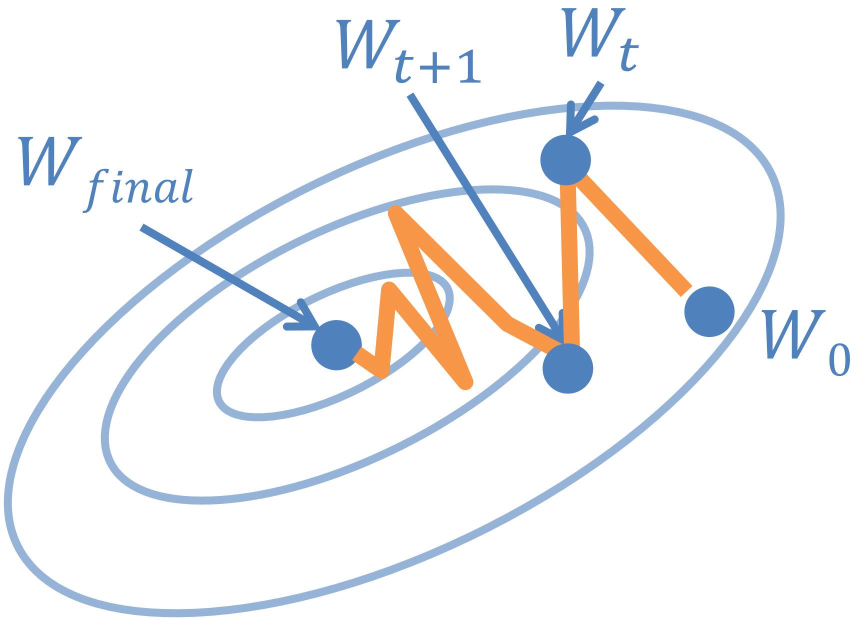






Stochastic gradient descent (SGD)

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 $0, \ldots, T_{\text{maxiter}}$

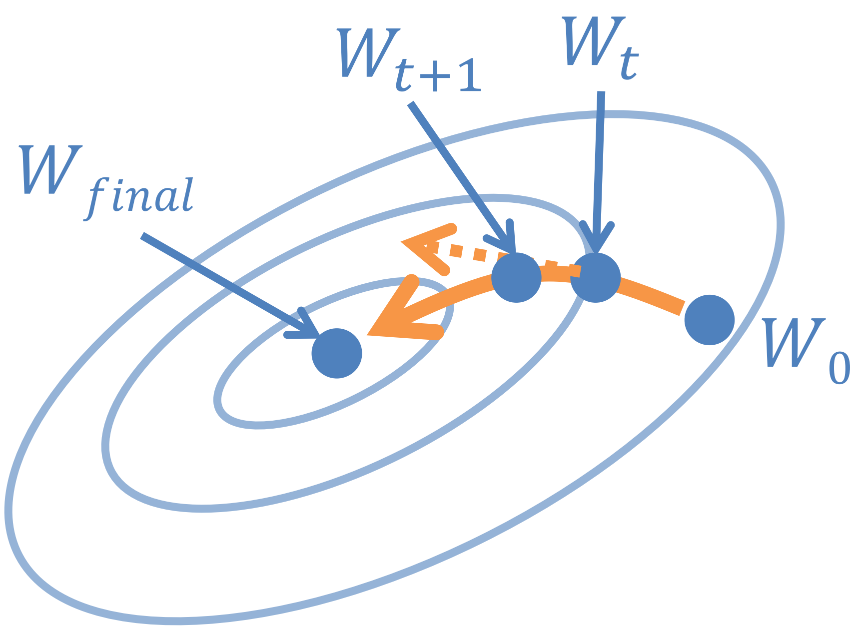
$$W^{t+1} = W^{t} - \eta_t \cdot \underbrace{\nabla Loss(f_w(x_i), y_i)}_{\text{Stochastic gradient}}$$

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.



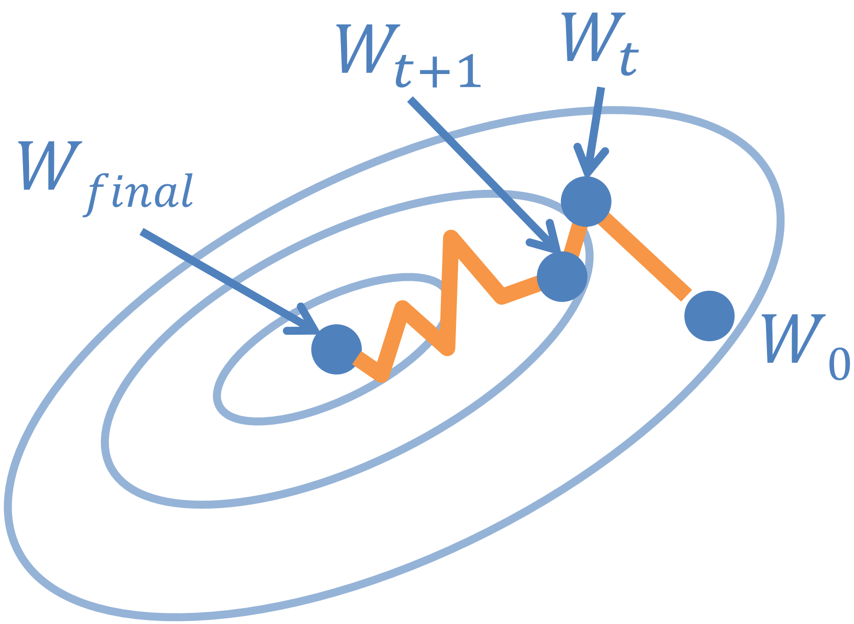
- Initialize W_0 randomly
- For t in $0,\ldots, T_{\text{maxiter}}$

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

- 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 $0, \ldots, T_{\text{maxiter}}$

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

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***.

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

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

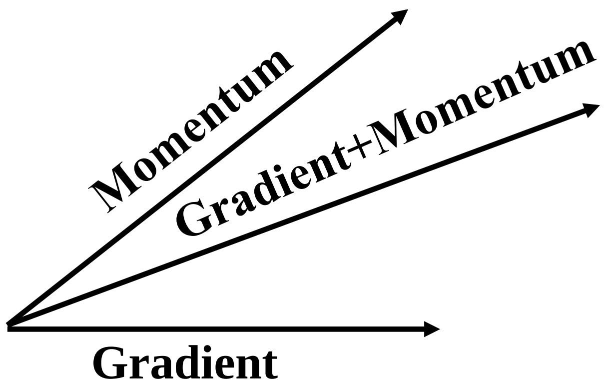
Momentum

Learning Rate

Nestrov Accelerated Gradient (NAG)

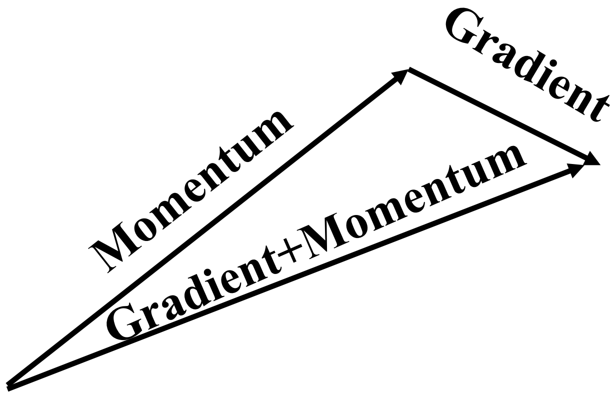
- SGD + Momentum

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

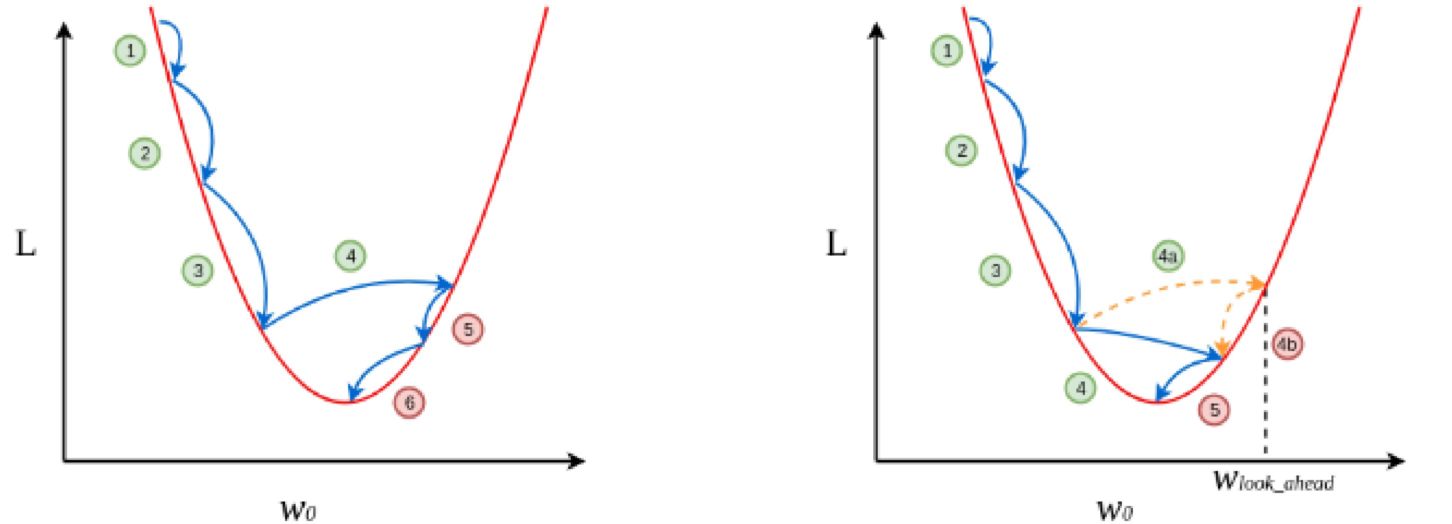


Nestrov Accelerated Gradient (NAG)

- 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

ADAPTIVE GRADIENT(ADAGRAD)

- SGD

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

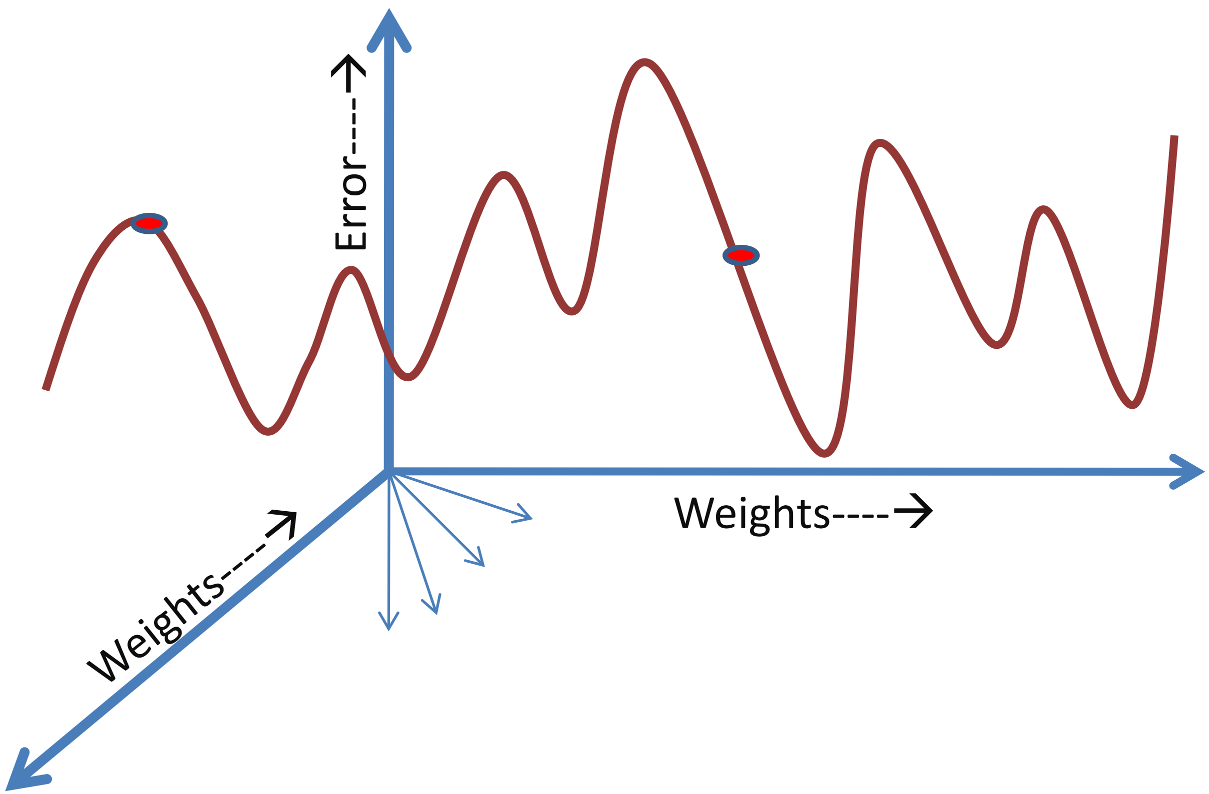
- Adagrad

$$\eta_t = \frac{\eta}{\sqrt{\alpha_{t-1} + \varepsilon}} \; with \; \alpha_t \geq \alpha_{t-1}$$

$$\alpha_{t-1} = \sum_{i=1}^{t-1} \left(\frac{dL}{dw}\right)_i^2$$

- As iteration increases the learning rate decreases.

ADAPTIVE GRADIENT(ADAGRAD)



ADAPTIVE GRADIENT(ADAGRAD)

- Advantages
  - No need of manual tuning
  - Works well for both Sparse and Dense Feature
- Disadvantages
  - As iteration increases, the learning rate will get low, which will result in Slow Convergence.
  - Computationally expensive.

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

Root Mean Square Propagation (RMSProp)

- It is same to AdaDelta however, it discards the history from extreme past while computing the exponentially decaying average.
- Converges faster once it finds a locally convex bowl as its error function.
- Faster convergence than AdaDelta.

ADAM(ADAPTIVE MOMENT ESTIMATION)

- https://arxiv.org/pdf/1412.6980.pdf

$$w_{t+1} = w_t - \alpha m_t$$

where,

$$m_t = \beta m_{t-1} + (1-\beta)\left[\frac{\delta L}{\delta w_t}\right]$$

m_t = aggregate of gradients at time t [current] (initially, m_t = 0)

m_{t-1} = aggregate of gradients at time t-1 [previous]

W_t = weights at time t

W_{t+1} = weights at time t+1

α_t = learning rate at time t

∂L = derivative of Loss Function

∂W_t = derivative of weights at time t

β = Moving average parameter (const, 0.9)

WHICH OPTIMIZER TO USE

- MiniBatch-SGD:::::::: Small/Shallow ANN
- Momentum & NAG::: Works well in most cases but Slower
- AdaGrad::::::::::::::::::: Sparse Features
- AdaDelta & RMSProp: Preferred Over AdaGrad
- Adam::::::::::::::::::::::: Most Favorite

How to Train a Deep Neural Network?: Not Limited

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

Thank You

For Your Valuable Time.