# ANN AND DEEP LEARNING (CS636)

BY: Nidhi S. Periwal,
Teaching Assistant,
COED, SVNIT, Surat

# Feed Forward Neural Networks

- Feed forward neural networks are artificial neural networks in **which nodes do not form loops**.
- This type of neural network is also known as a **multi-layer neural network as all information is only passed forward**.
- During data flow, input nodes receive data, which travel through hidden layers, and exit output nodes.
- **No links** exist in the network that could get used to by sending **information back from the output node.**

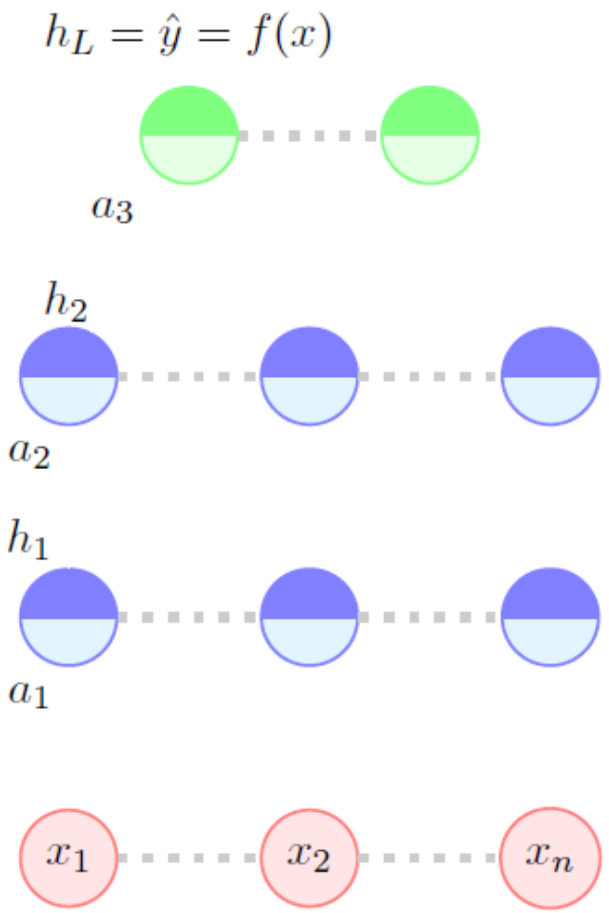
# Feed Forward Neural Networks

- "The process of receiving an input **to produce some kind of output to make some kind of prediction is known as Feed Forward**."
- Feed Forward neural network is the **core of many other important neural networks such as convolution neural network**
- In the feed-forward neural network, there are not any feedback loops or connections in the network.
- Here is simply an input layer, a hidden layer, and an output layer.

# Deep Feed Forward Neural Network

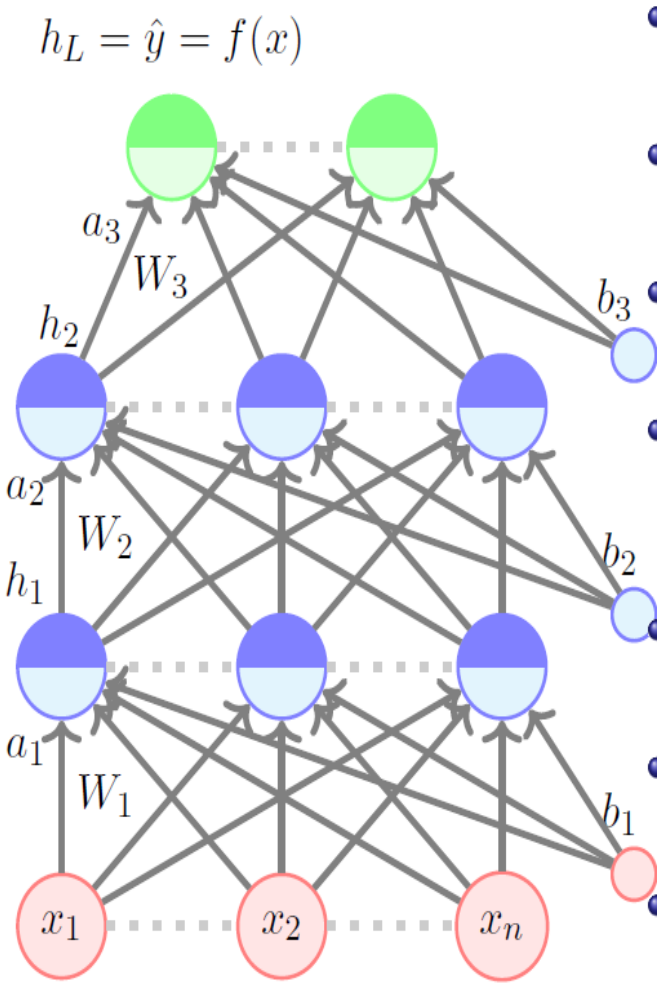
- Deep feed-forward networks are effective at learning from data with a **rich structure because the multiple layers of nonlinear activations force the data to follow specific types of patterns**.
- Furthermore, by adjusting the **structure of connections, one can incorporate domain-specific** insights in feed-forward networks. Examples of such settings include recurrent and convolutional neural networks.

# Deep L-Layer Neural Network




- The input to the network is an **n**-dimensional vector
- The network contains $\mathbf{L-1}$ hidden layers (2, in this case) having **n** neurons each
- Finally, there is one output layer containing **k** neurons (say, corresponding to **k** classes)
- Each neuron in the hidden layer and output layer can be split into two parts : pre-activation and activation ($a_i$ and $h_i$ are vectors)
- The input layer can be called the 0-th layer and the output layer can be called the $(L)$-th layer

# Deep L-Layer Neural Network

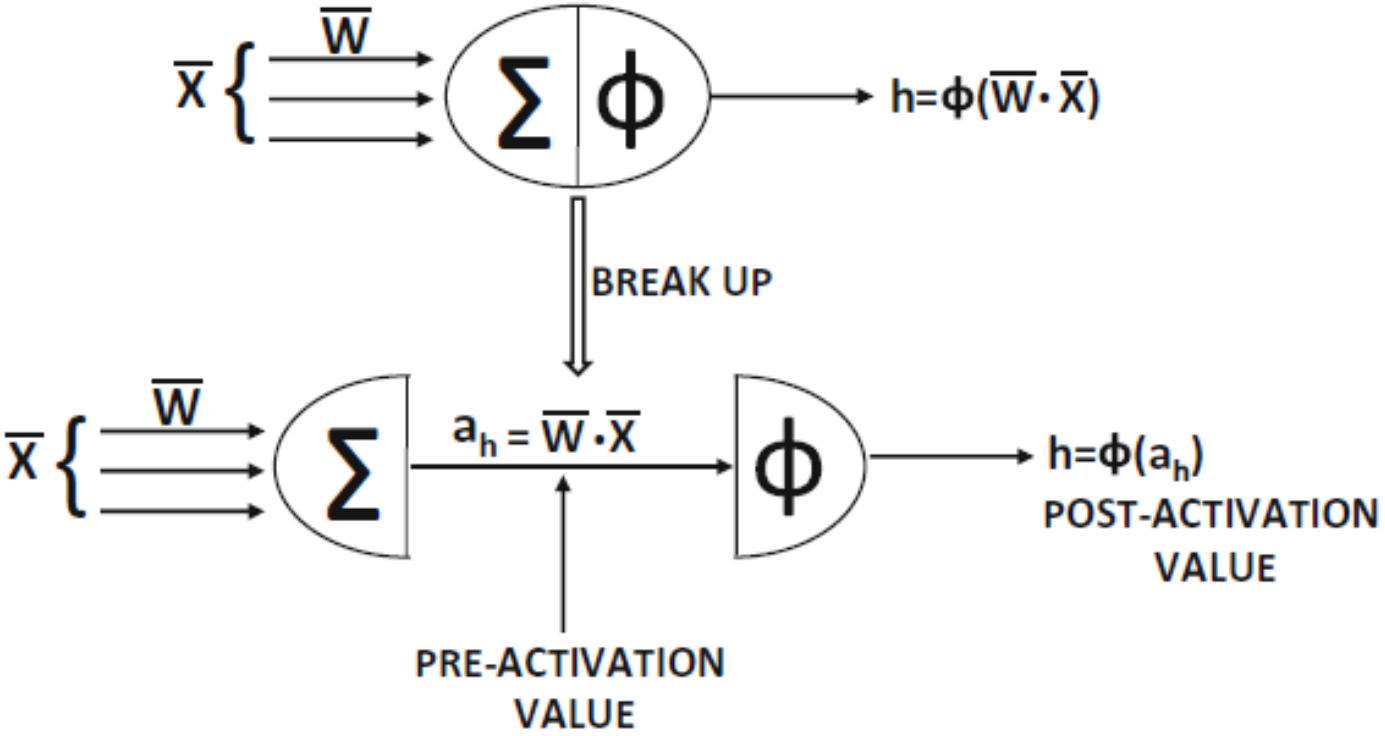



- $W_i \in \mathbb{R}^{n \times n}$ and $b_i \in \mathbb{R}^n$ are the weight and bias between layers $i-1$ and $i$ $(0 < i < L)$
- $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$ are the weight and bias between the last hidden layer and the output layer

# Deep L-Layer Neural Network

- Summation : $Z_1^{[1]} = W_1^{[1]} X + b_1^{[1]}$
- Activation : $A_1^{[1]} = f_1^{[1]}(Z_1^{[1]})$
- Superscript→ Layer in the network
- Subscript→ Sequence number of the specific neuron in the network

# Neural Network




Pre-activation and post-activation values within a neuron

# Parameters vs Hyperparameters

- Parameters are learned by the model during the training time,
  - Parameters of a deep neural network are W and b, which the model updates during the backpropagation step.
- Hyperparameters can be changed before training the model.
  - Hyperparameters for a deep NN, including:
    - Learning rate – $\alpha$
    - Number of iterations
    - Number of hidden layers
    - Units in each hidden layer
    - Choice of activation function

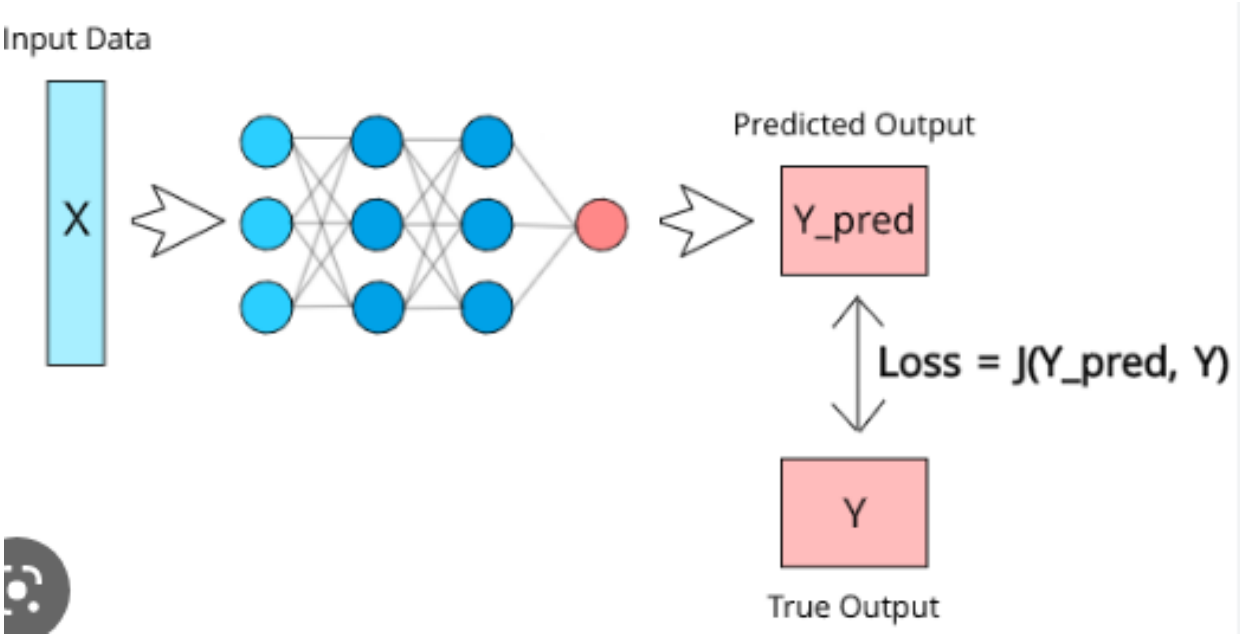
# Learning in NN

- Start with arbitrary value of Weights and bias.
- Weights and bias changes with **multiple iterations until there is no wrong prediction.**
- In supervised learning, the objective is to **reduce the number of erroneous prediction**.
- On increase in number of layers or interconnection weights, the problem becomes complex
- These parameters are learnt using back propagation.

# Gradient Descent

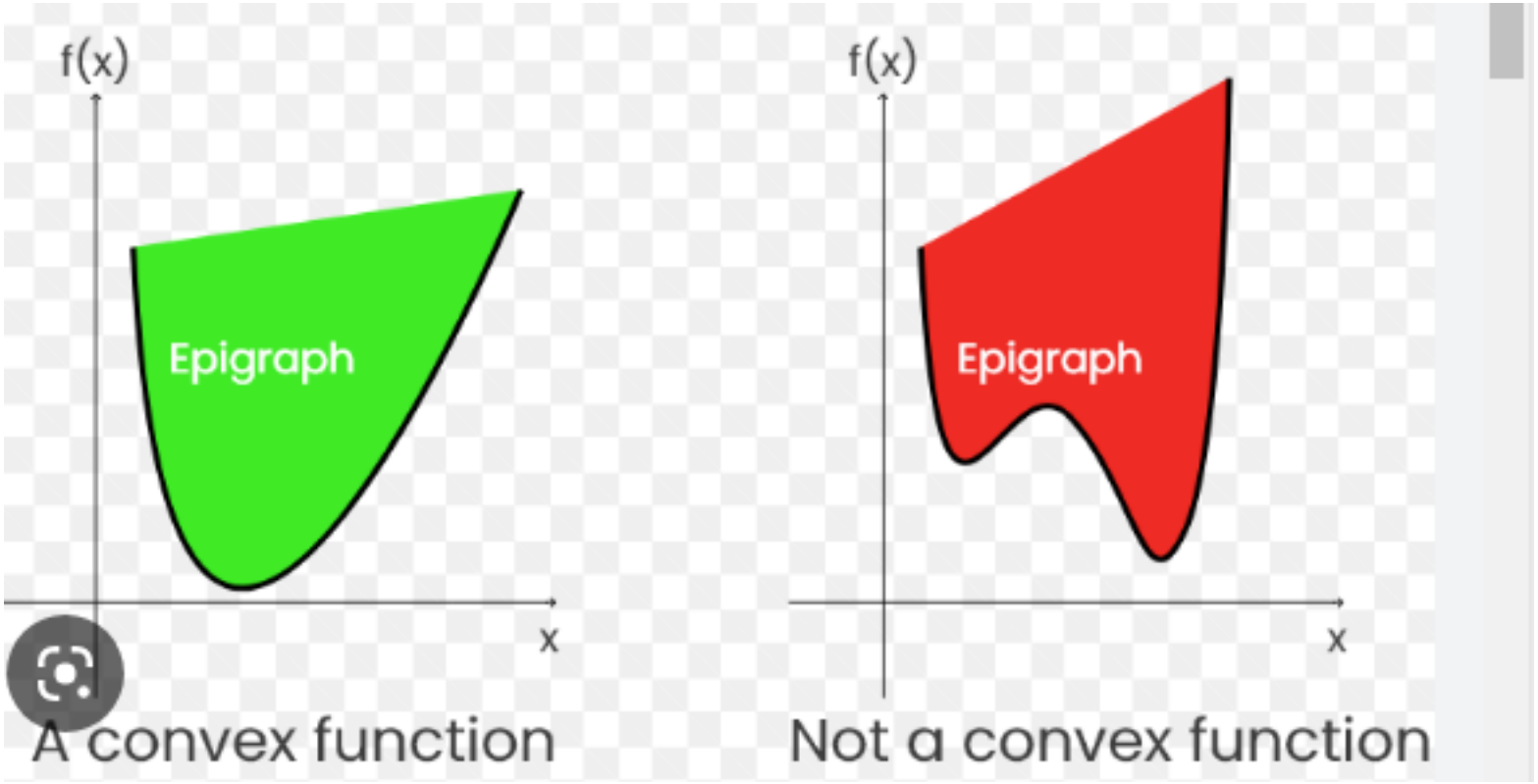
- Gradient Descent – key concept in back propagation
- *Gradient Descent is defined as* ***one of the most commonly used iterative optimization algorithms of machine learning to train the machine learning and deep learning models.*** *It helps in finding the local minimum of a function*
- *Gradient calculates* ***the partial derivative of loss function by each interconnection weight and bias.***
- ***i.e. identify the "gradient" or extent to change of weight or bias required to minimize loss function****.*

- **Loss Functions** are used to calculate the error between the known **predicted output and the actual output** generated by a model, Also often called Cost Functions.
- Measures extent of prediction error of supervised model

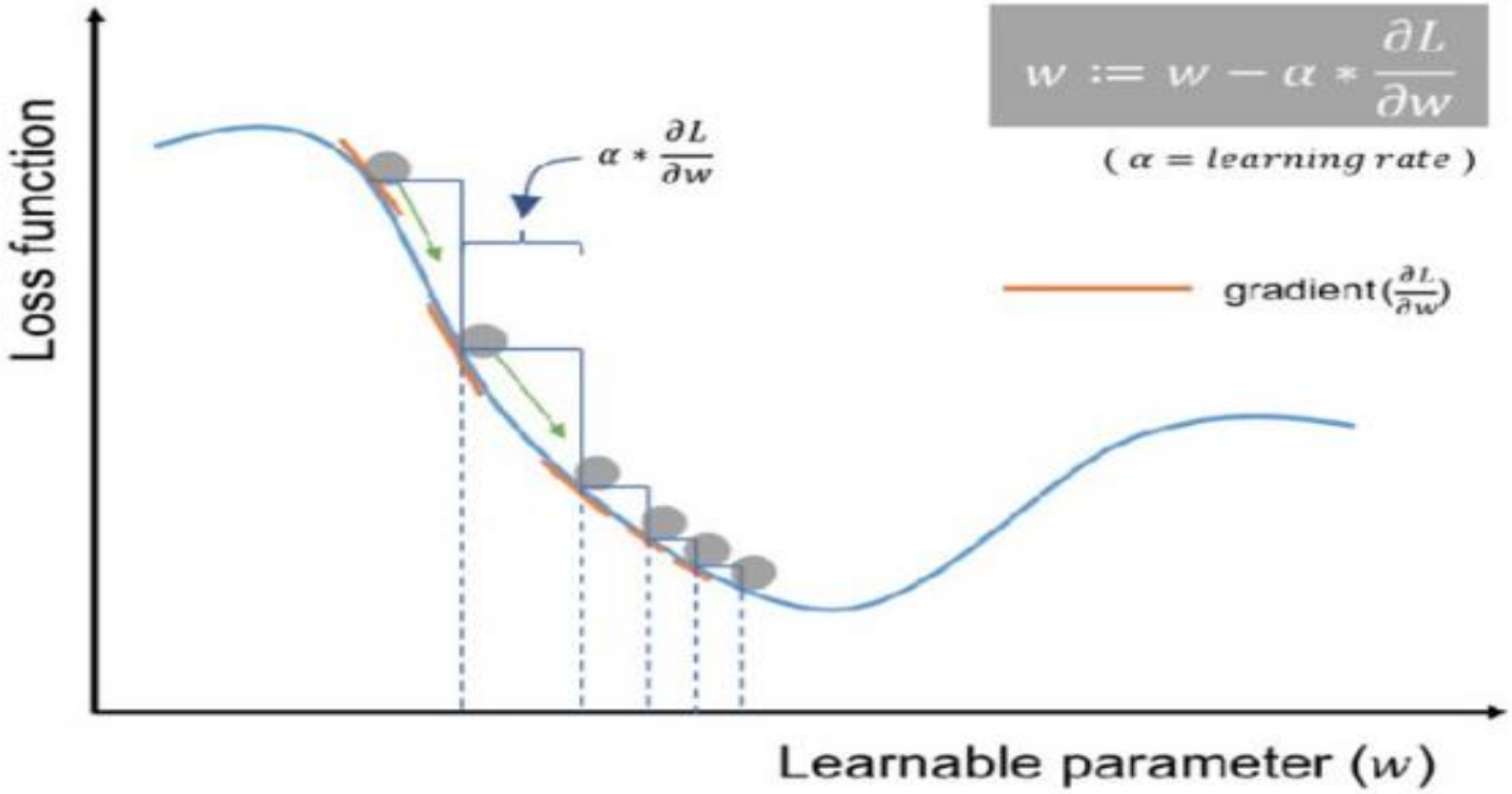


# Gradient Descent

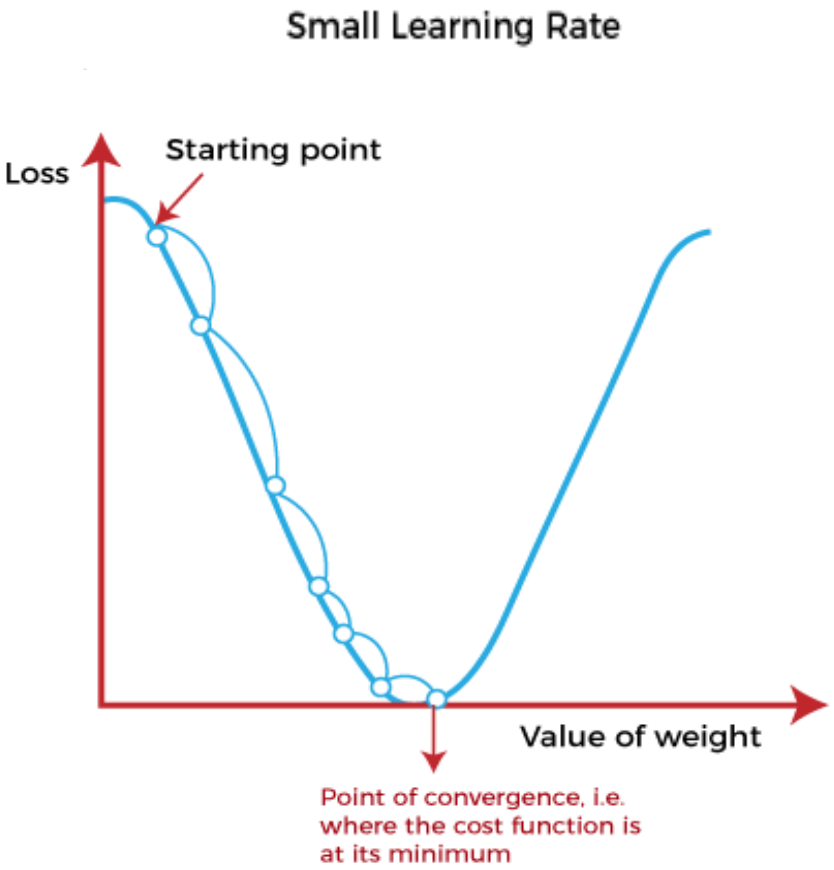
- Convex problems have only one minimum; that is, only one place where the slope is exactly 0. **That minimum is where the loss function converges**.
- **Gradient Descent** is an **iterative** optimization method for finding the minimum of a function. **On each iteration the parameters in a model are amended in the direction of the negative gradient of the output until the optimum parameters for the model are identified**

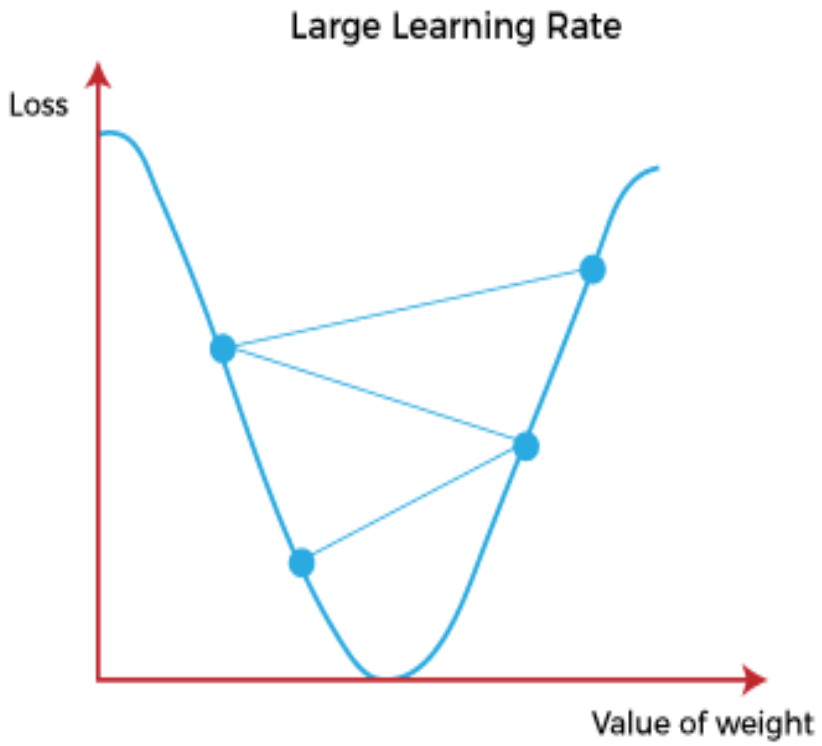

f(x)
Epigraph
x
A convex function
f(x)
Epigraph
x
Not a convex function

# Gradient Descent



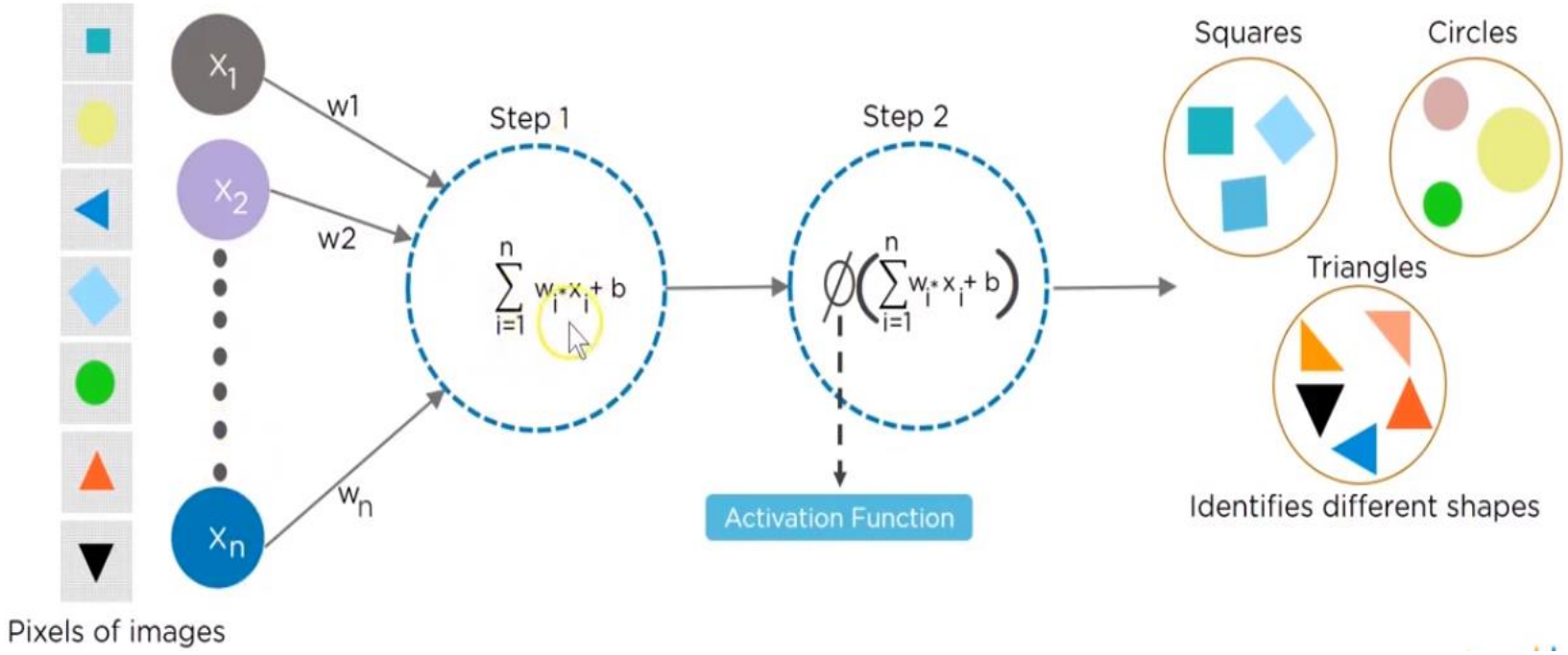
- **Learning Rate:**
- It is defined as the **step size** taken to reach the minimum or lowest point.
- This is typically **a small value that is evaluated and updated based on the behavior of the cost function**.
- If the learning rate is **high, it results in larger steps but also leads to risks of overshooting the minimum**.
- A **low learning rate shows the small step sizes, which compromises overall efficiency** but gives the advantage of more precision.

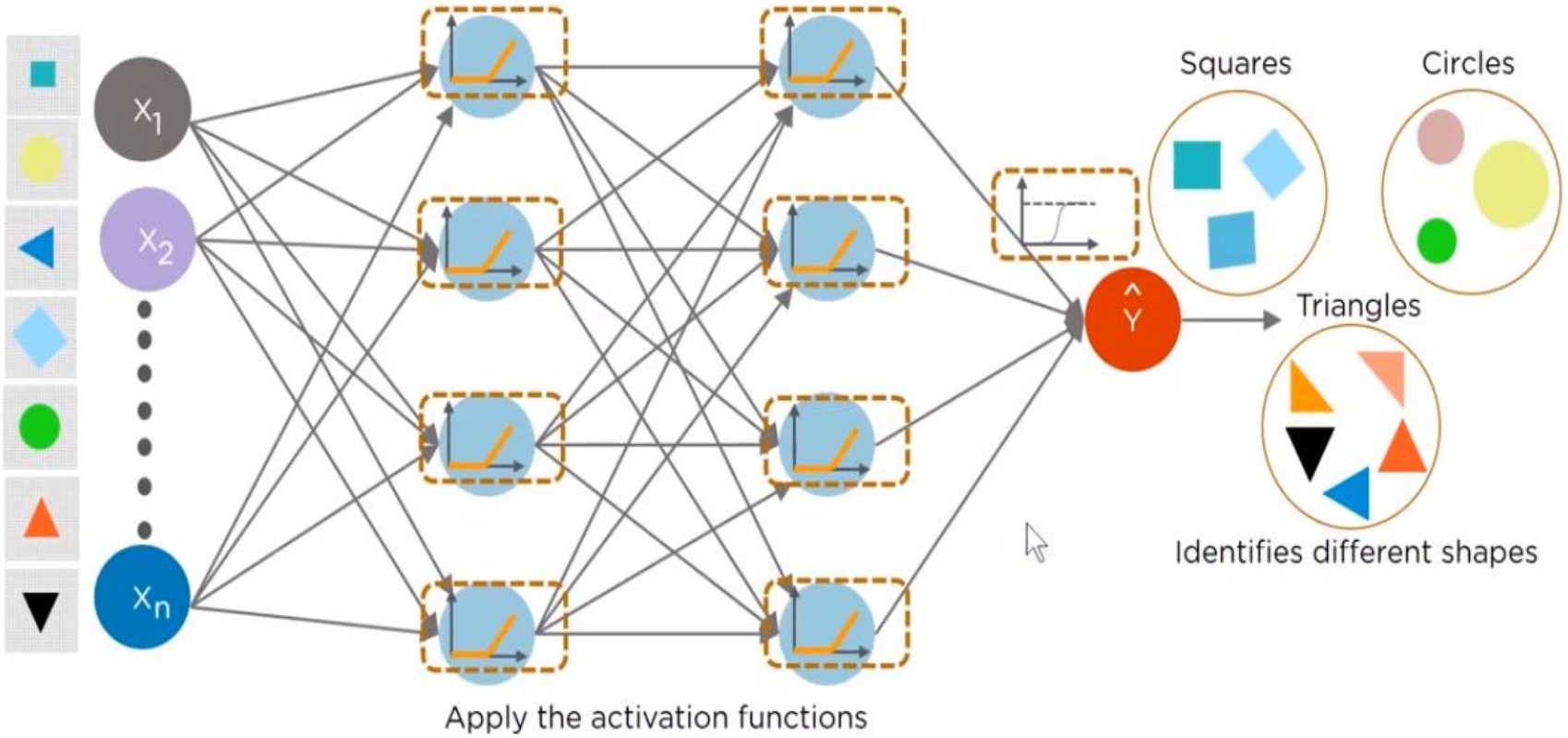

Small Learning Rate
Loss
Starting point
Value of weight
Point of convergence, i.e.
where the cost function is
at its minimum

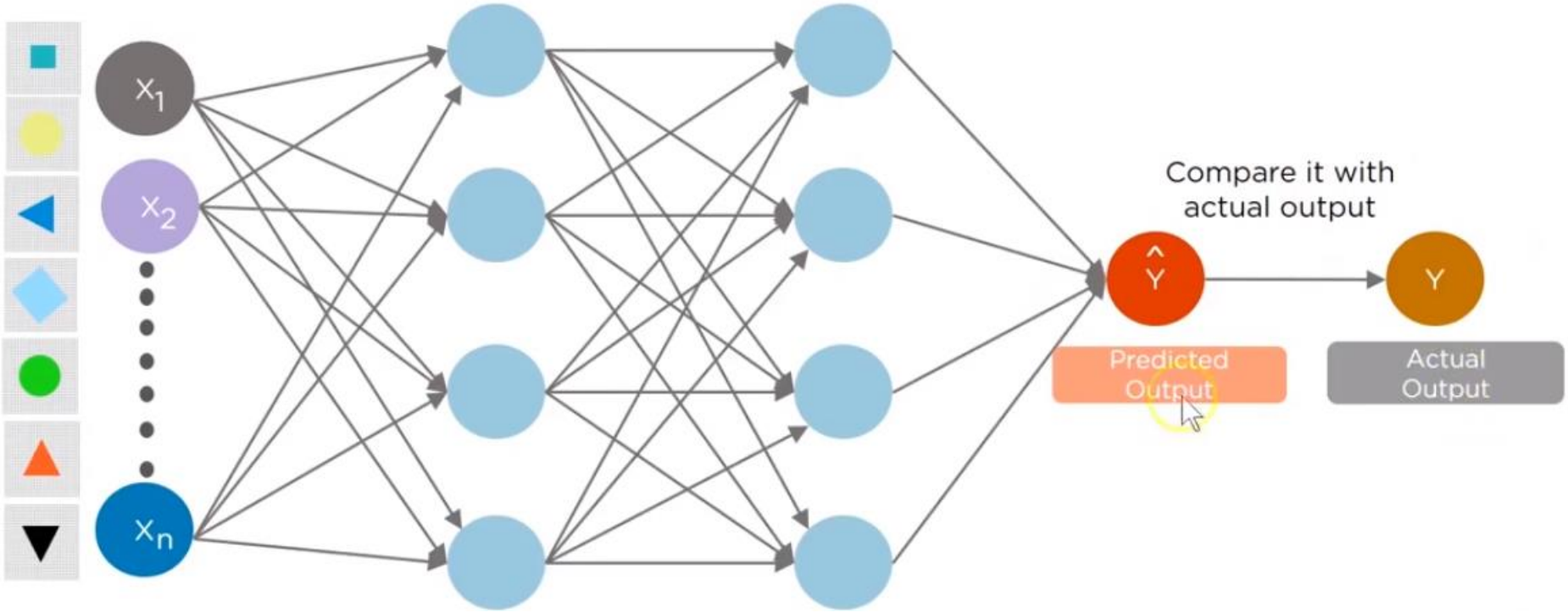

Large Learning Rate
Loss
Value of weight

# Working of Simple NN

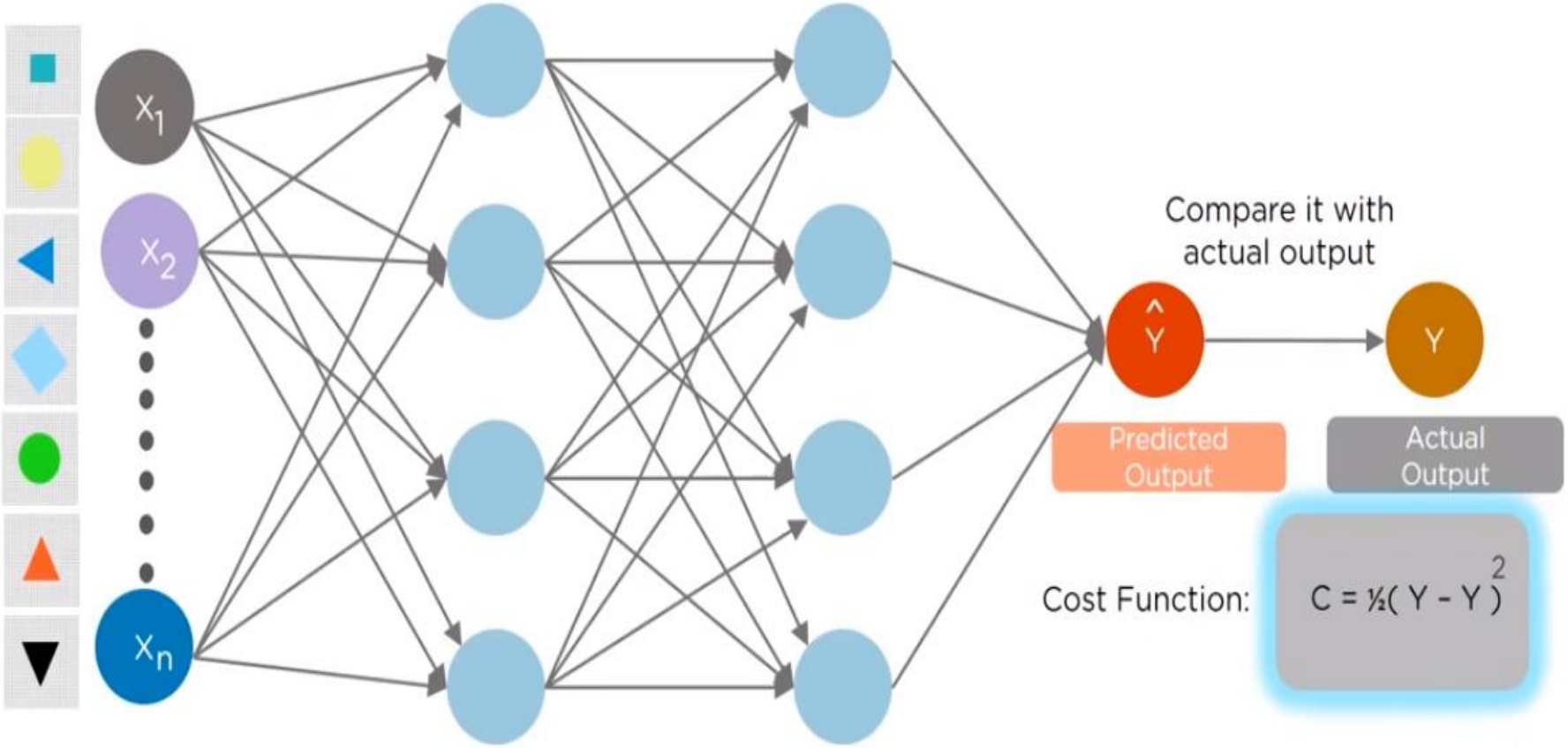
Lets find out how an Artificial Neural Network can be used to identify different shapes



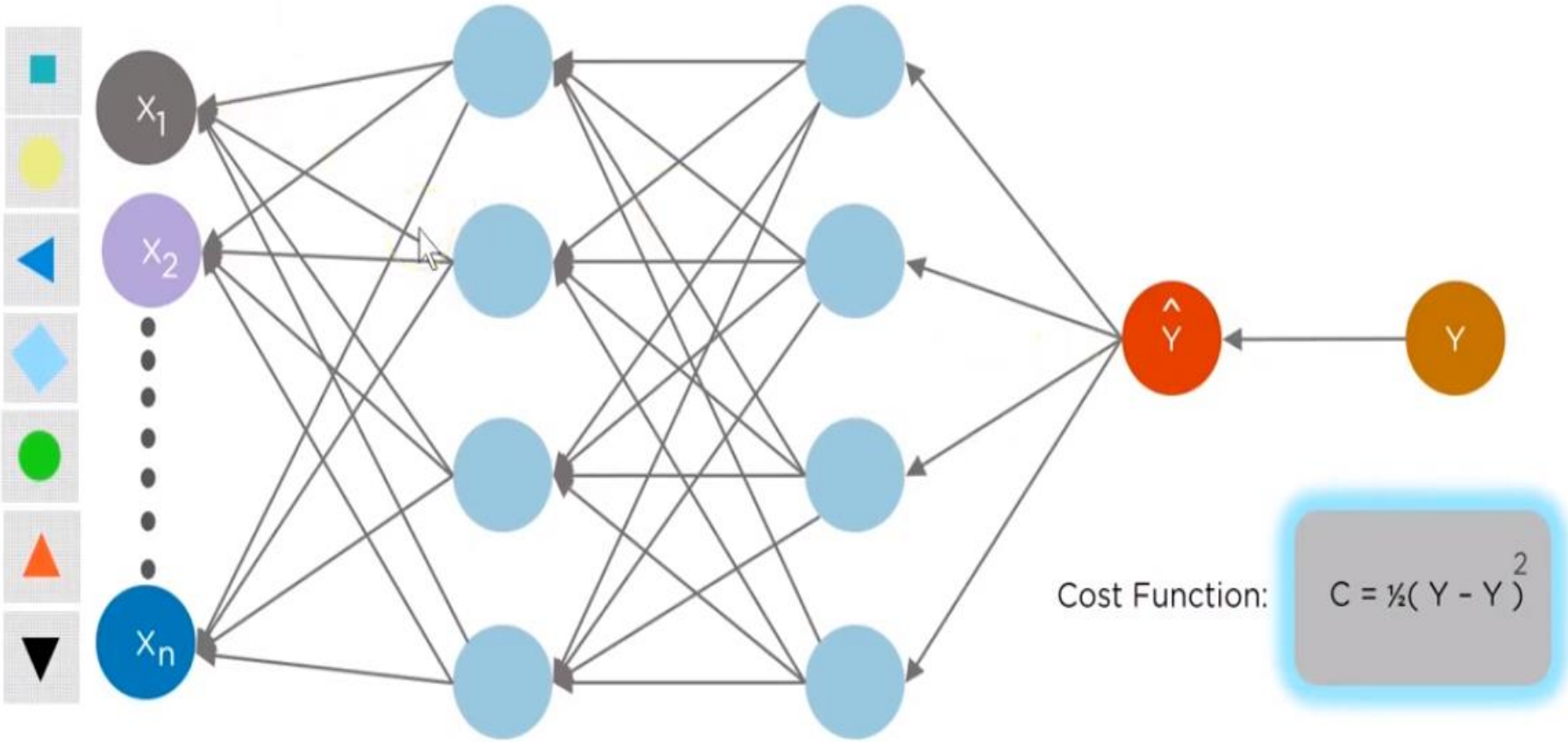

$x_1$
$x_2$
$x_n$
Apply the activation functions
Squares
Circles
Triangles
Identifies different shapes

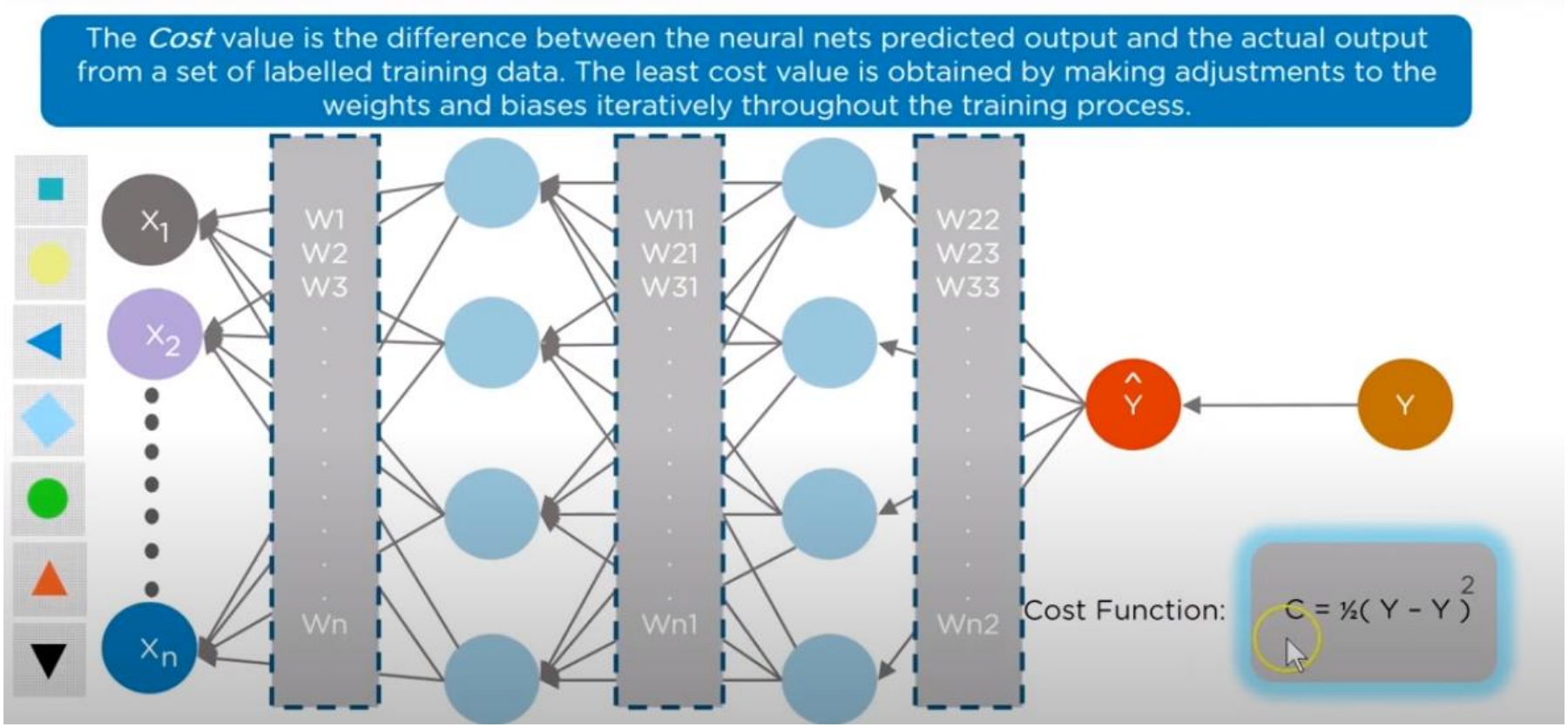

$x_1$
$x_2$
$x_n$
Compare it with actual output
$\hat{Y}$
Y
Predicted Output
Actual Output

Applying the cost function to minimize the difference between predicted and actual output using gradient descent algorithm

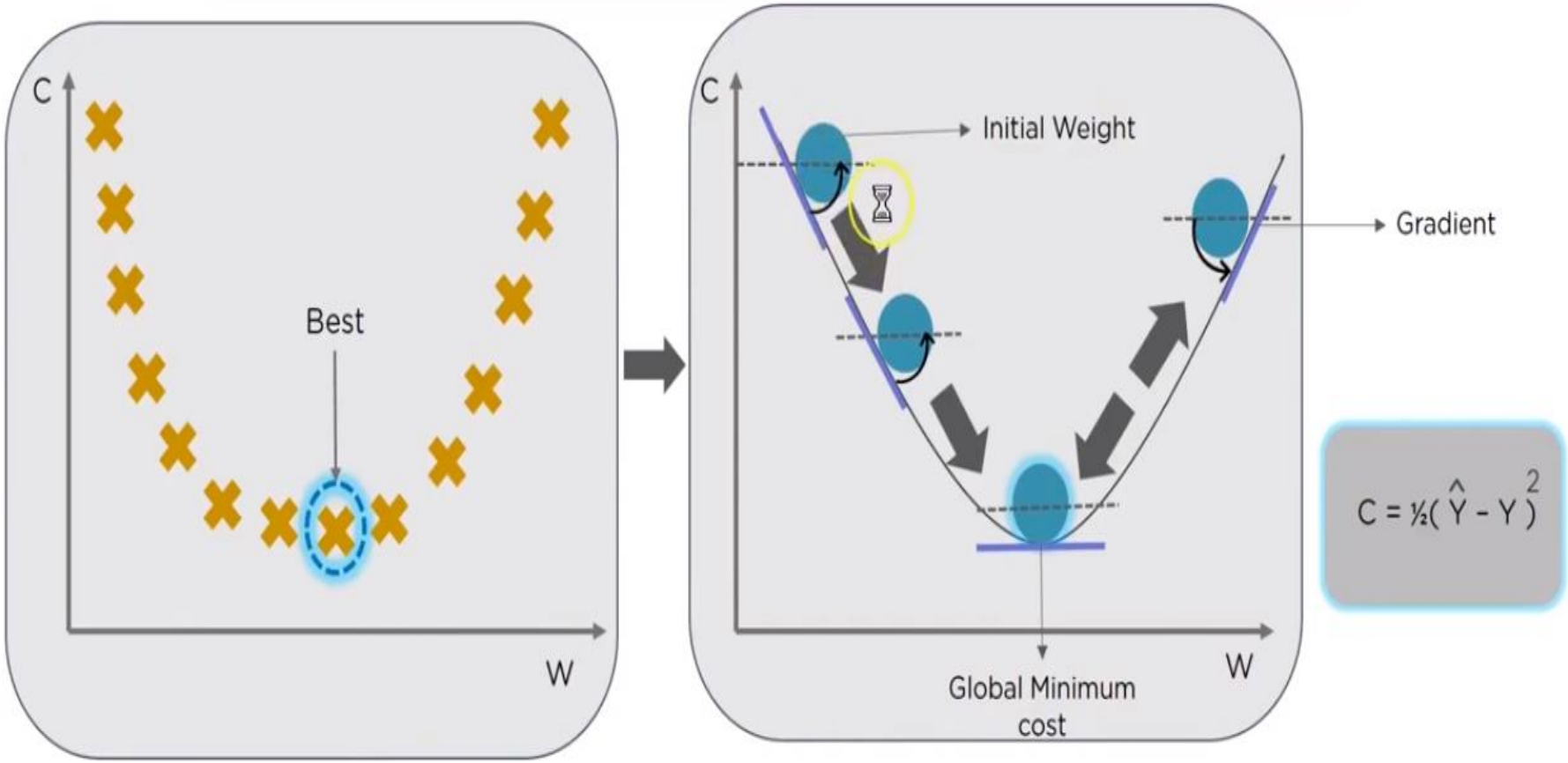


Neural Networks use *Backpropagation* method along with improve the performance of the Neural Net. A *cost function* is used to reduce the error rate between predicted and actual output.




The *Cost* value is the difference between the neural nets predicted output and the actual output from a set of labelled training data. The least cost value is obtained by making adjustments to the weights and biases iteratively throughout the training process.
X1
X2
Xn
W1
W2
W3
Wn
W11
W21
W31
Wn1
W22
W23
W33
Wn2
Ŷ
Y
Cost Function:
C = ½( Y - Y )2

Gradient Descent is an optimization algorithm for finding the minimum of a function



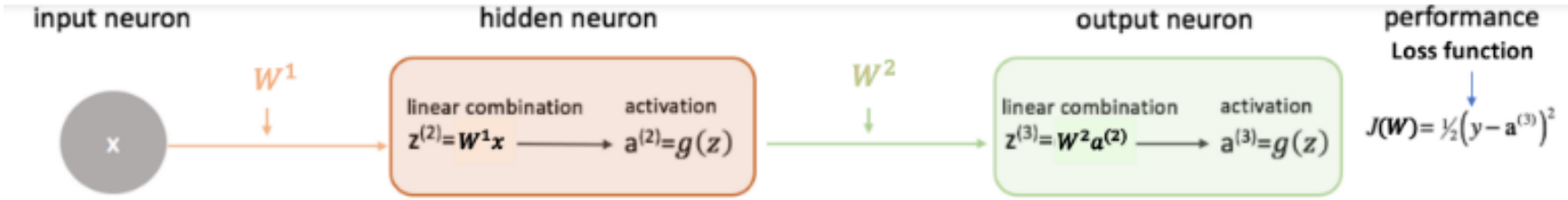
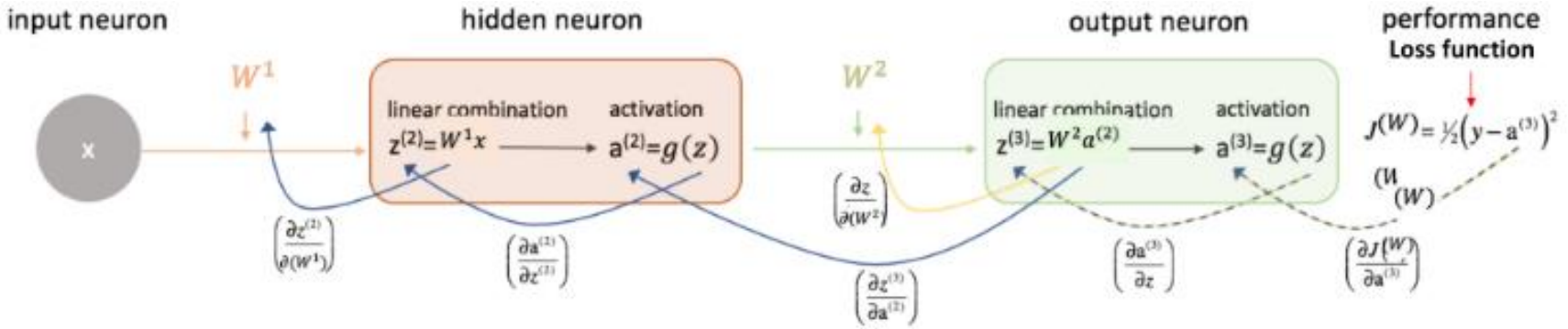



Fig. Forward Propagation

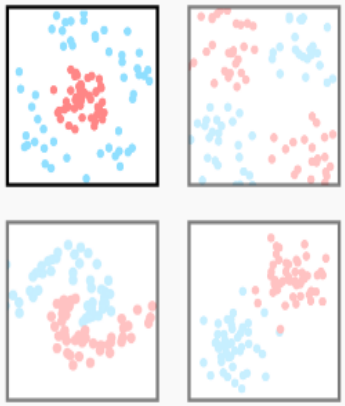


# Weight Initialization

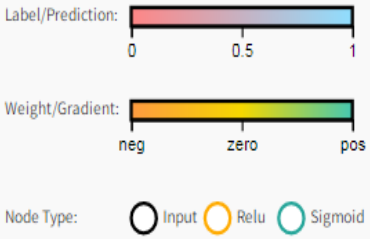
**https://www.deeplearning.ai/ai-notes/initialization/index.html**

**1. Choose input dataset**

Select a training dataset.



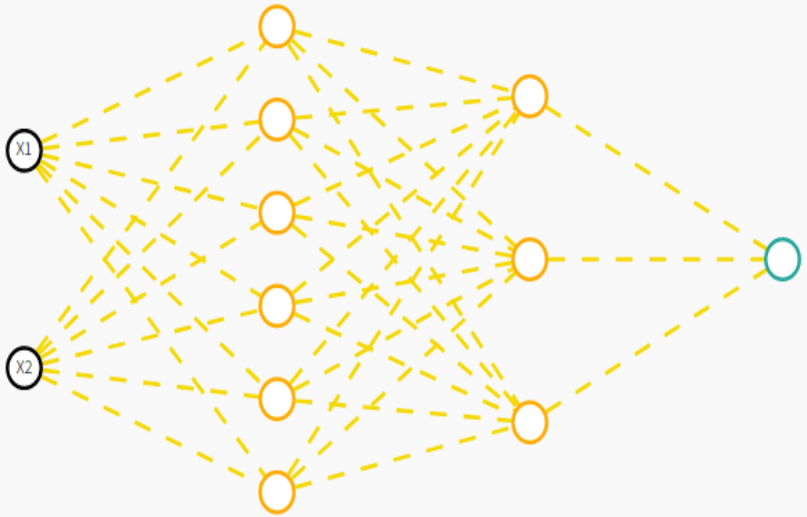
This legend details the color scheme for labels, and the values of the weights/gradients.




**2. Choose initialization method**

Select an initialization method for the values of your neural network parameters[1].

Zero  Too small  Appropriate  Too large

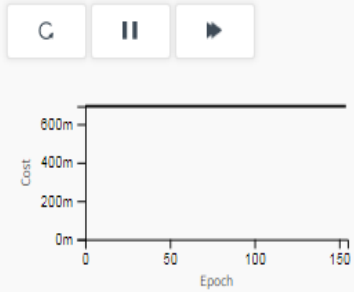


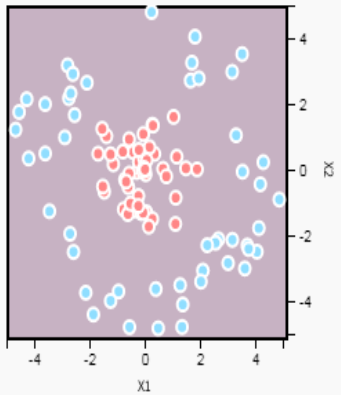

Select whether to visualize the weights or gradients of the network above.

Weight  Gradient

**3. Train the network.**

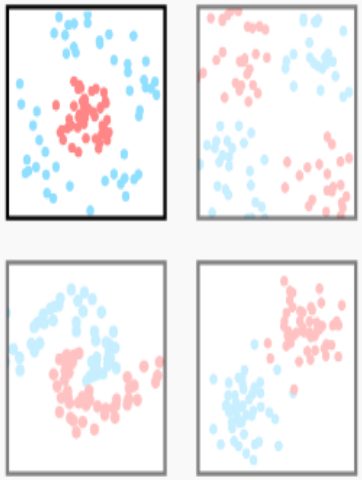
Observe the cost function and the decision boundary.



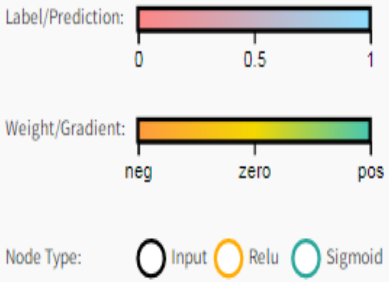



**1. Choose input dataset**

Select a training dataset.



This legend details the color scheme for labels, and the values of the weights/gradients.




**2. Choose initialization method**

Select an initialization method for the values of your neural network parameters[1].

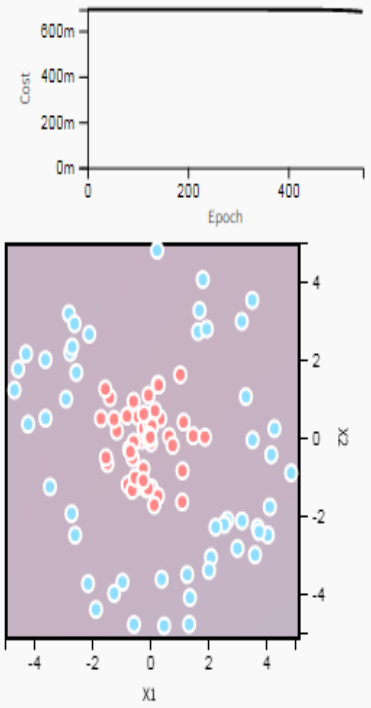
Zero  Too small  Appropriate  Too large


Select whether to visualize the weights or gradients of the network above.

Weight  Gradient

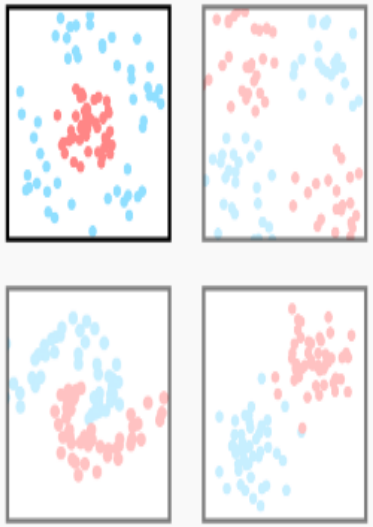
**3. Train the network.**

Observe the cost function and the decision boundary.

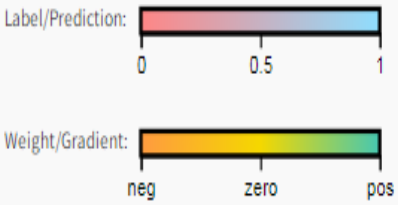


**1. Choose input dataset**

Select a training dataset.



This legend details the color scheme for labels, and the values of the weights/gradients.




**2. Choose initialization method**

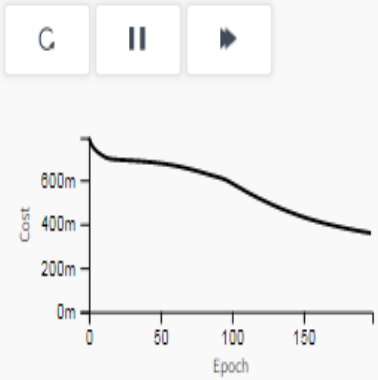
Select an initialization method for the values of your neural network parameters[1].

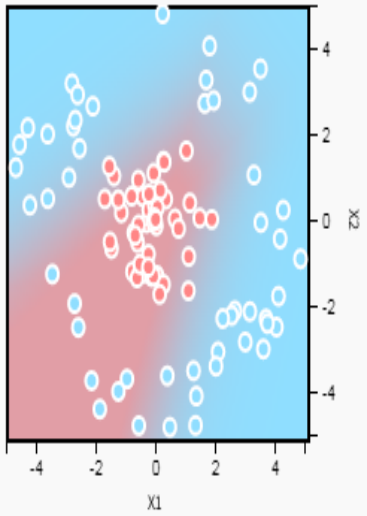
Zero  Too small  Appropriate  Too large


Select whether to visualize the weights or gradients of the network above.

**3. Train the network.**

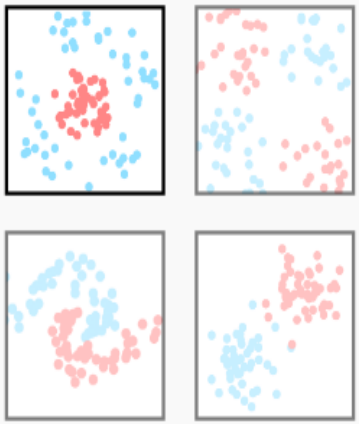
Observe the cost function and the decision boundary.



**1. Choose input dataset**

Select a training dataset.



This legend details the color scheme for labels, and the values of the weights/gradients.

Label/Prediction: 0 0.5 1

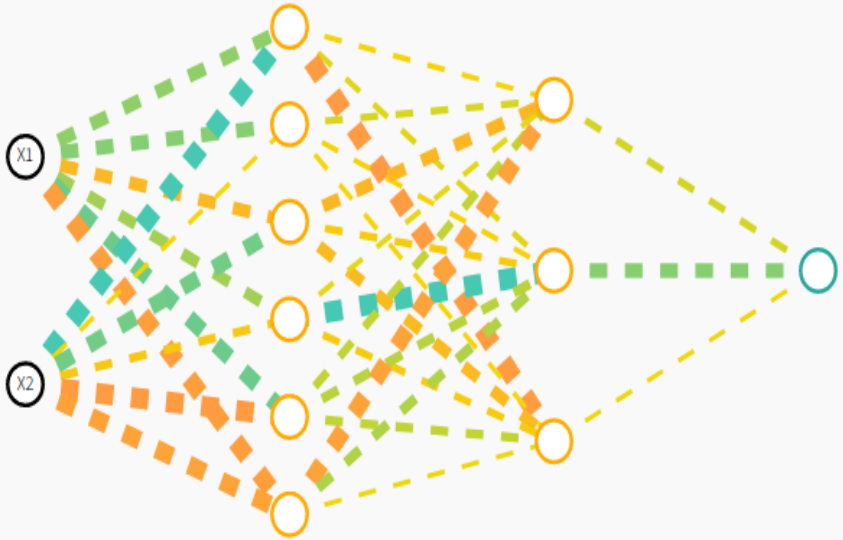
Weight/Gradient: neg zero pos

Node Type: Input Relu Sigmoid

**2. Choose initialization method**

Select an initialization method for the values of your neural network parameters[1].

Zero Too small Appropriate Too large

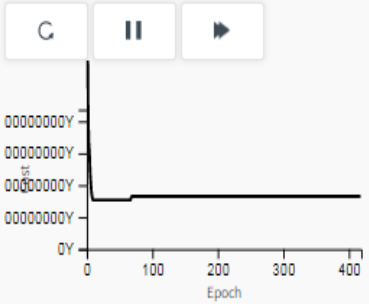


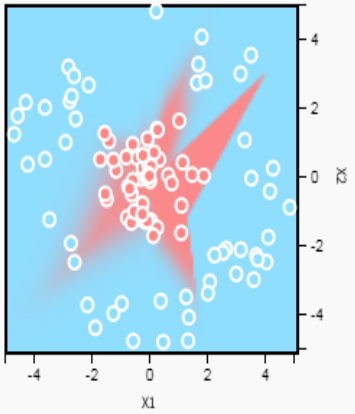

Select whether to visualize the weights or gradients of the network above.

Weight Gradient

**3. Train the network.**

Observe the cost function and the decision boundary.



# Weight Initialization

**Zero**

- *Initializing all the weights with* ***zeros leads the neurons to learn the same features during training***
- $\mathbf{W^k_i}$**=c for all i,** k , that is weights of all layers and in between any two **nodes are zero is same and has the value c .**
- Since the value is same for all neurons **, all the neurons would be symmetric(between neuron and all its subsequent connections) and will receive same updates** .
- We want **each** neuron to **learn a certain feature** and this initialization technique wont let that happen

# Weight Initialization

***Zero or Random Constant***

- Consider a *neural network* with two hidden units, and assume we initialize all the biases **to 0 and the weights with some constant $\alpha$.**
- If we forward propagate an input $(x_1,x_2)$ in this network, the output of both hidden units will be $relu(\alpha x1+\alpha x2)$.
- Thus, both hidden units will have identical influence on the cost, which will **lead to identical gradients.**
- Thus, both neurons will evolve symmetrically **throughout training, effectively preventing different neurons from learning different things**

# Weight Initialization

**A too-large initialization leads to exploding gradients**

- If the gradients get **LARGER** as our backpropagation progresses, we would end up with exploding gradients **having big weight updates, leading to the divergence of the gradient descent algorithm**
- **Initial weights** assigned to the neural nets creating **large losses**.
- The gradients of the cost with the respect to the parameters are too big.
- **This leads the cost to oscillate around its minimum value.**
- Model weights can become NaN very quickly

# Weight Initialization

**A too-small initialization leads to vanishing gradients**

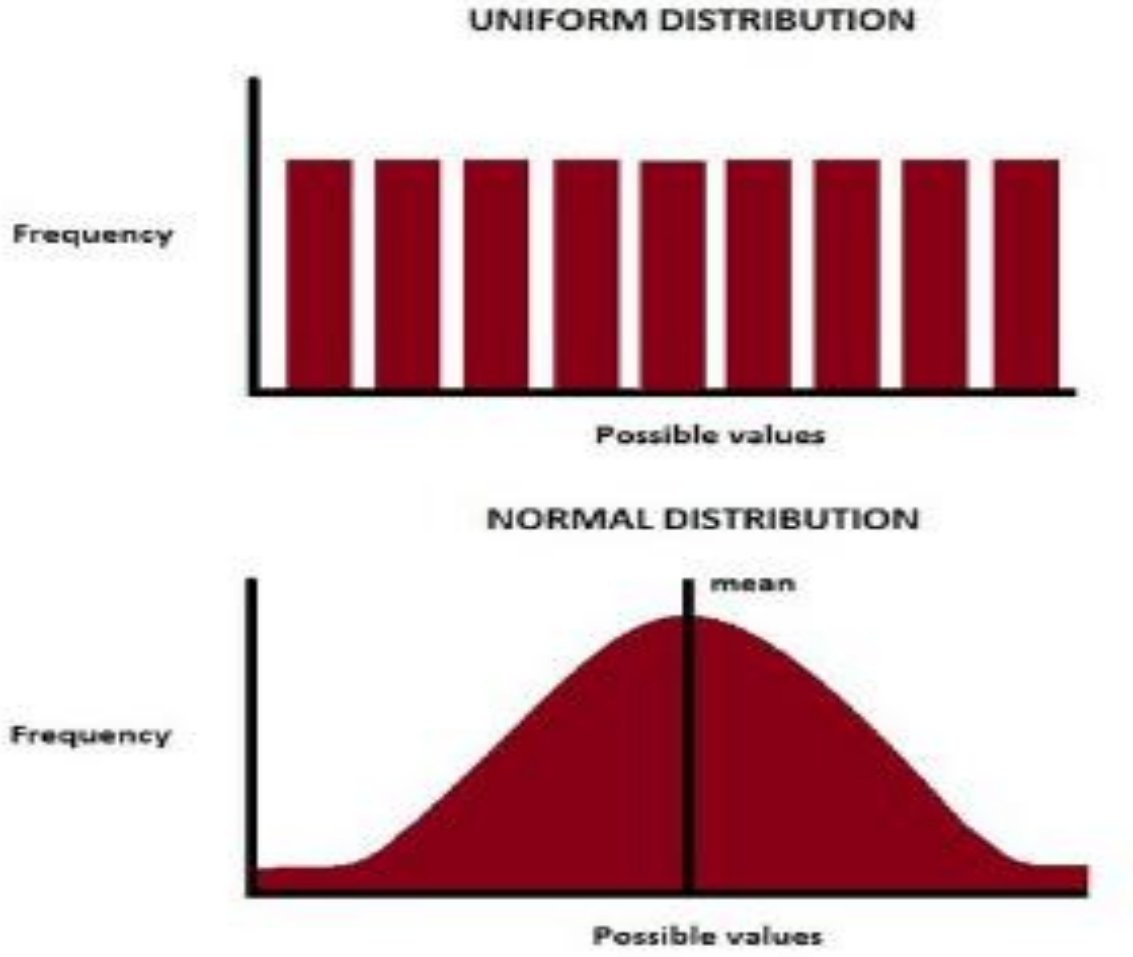
- The **gradients frequently become SMALLER until they are close to zero**, the new model weights (of the initial layers) will be virtually **identical to the old weights** without any updates.
- The gradient descent algorithm never converges to the optimal solution.
- It causes **unstable** behaviors of neural nets.
- The model learns **slowly and often times, training stops after a few iterations**

# Weight Initialization

- **Uniform Initialization**
- In uniform initialization of weights , weights belong to a uniform distribution in range a,b with values of a and b as below.
- Works well for Sigmod Distribution

$$W \approx U(a,b) \qquad a = \frac{-1}{\sqrt{f\_in}} \;,\; b = \frac{1}{\sqrt{f\_in}}$$

- f_in -> number of inputs to that neuron


UNIFORM DISTRIBUTION
Frequency
Possible values
NORMAL DISTRIBUTION
mean
Frequency
Possible values

# Weight Initialization

**Xavier initialization or Glorot initialization**

- Keeps the variance the same across every layer.
- Assume that our layer's activations are normally distributed around zero.
- Glorot and Xavier had a belief that if they **maintain variance of activations in all the layers going forward and backward convergence will be fast.**
- Works well for tanh/ sigmoid

# Weight Initialization

**Xavier initialization or Glorot initialization**

- In Xavier **Normal** Distribution, weights belong to normal distribution where mean is zero and standard deviation is as below

$$W \approx N(\mu, \sigma)$$

$$\mu = 0 \qquad \sigma = \sqrt{\frac{2}{f\_in + f\_out}}$$

# Weight Initialization

- **Xavier initialization or Glorot initialization**
- In Xavier **Uniform** Distribution , weights belong to uniform distribution in range as below:

$$W \approx U(a,b)$$

$$a = -\sqrt{\frac{6}{f\_in + f\_out}}, \qquad b = \sqrt{\frac{6}{f\_in + f\_out}}$$

# Weight Initialization

**He/Kaiming**

- **Kaiming Initialization**, or **He Initialization**, is an initialization method for neural networks that takes into account the non-linearity of activation functions, such as ReLU activations.
- Weights belong to normal distribution

$$W \approx N(\mu, \sigma)$$

$$\mu = 0 \qquad \sigma = \sqrt{\frac{2}{f\_in}}$$

# Benefits of Weight Initialization

- They serve as good starting points for weight initialization and they reduce the chances of exploding or vanishing gradients.
- Do not vanish or explode too quickly, as the weights are neither too much bigger than 1 nor too much less than 1.
- They help to avoid slow convergence and ensure that we do not keep oscillating off the minima.

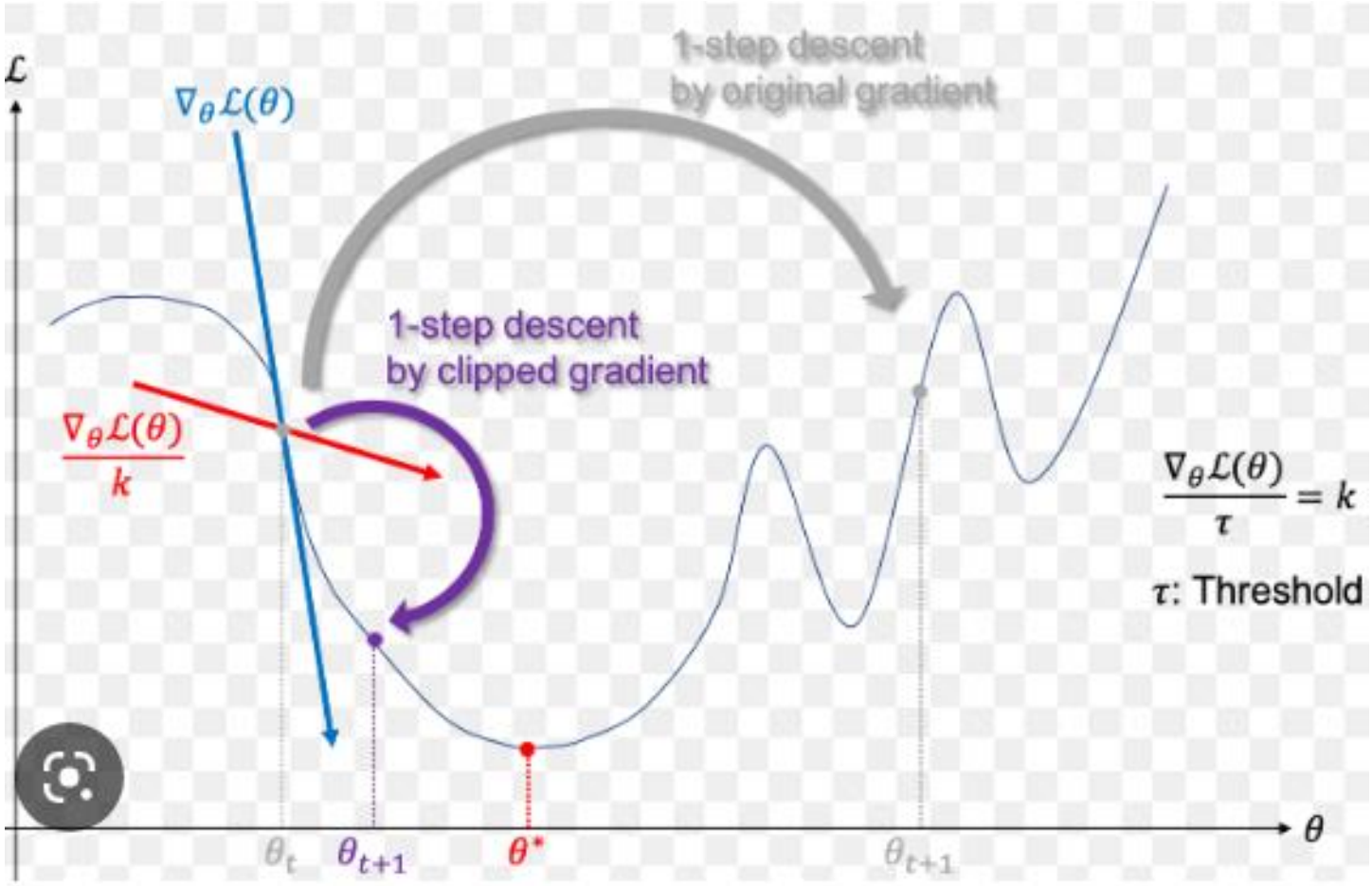
# Sum it up…..

- Zero initialization causes the neuron to memorize the same functions almost in each iteration.
- Random initialization is a better choice to break the symmetry. However, initializing weight with much high or low value can result in slower optimization.
- Using an extra scaling factor in **Xavier initialization, He-et-al Initialization**, etc can solve the above issue to some extent. That's why these are the more recommended weight initialization methods among all.

# Gradient Clipping

- Gradient Clipping is **a method where the error derivative is changed or clipped to a threshold during backward propagation through the network, and using the clipped gradients to update the weights**.
- **Gradient clipping** is a technique to prevent exploding gradients in very deep networks, usually in recurrent neural networks.
- *When the traditional gradient descent algorithm proposes to make a **very large step, the gradient clipping heuristic intervenes to reduce the step size to be small enough that it is less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.***
- It is a method that only addresses the **numerical stability of training deep neural network** models and does **not** offer any general improvement in **performance**.

# Gradient Clipping



Thank You!