edureka!
a Veranda Enterprise | ILLINOIS TECH

POST GRADUATE PROGRAM IN GENERATIVE AI AND ML

Deep Learning and Neural Network Architectures

Module Outline

edureka!
a Veranda Enterprise | ILLINOIS TECH

- Neural Networks and Deep Learning Foundation
- Tuning and Optimizing Deep Neural Networks
- Convolutional Neural Networks - I
- Convolutional Neural Networks - II
- Recurrent Neural Networks
- Long Short-Term Memory (LSTM) Networks

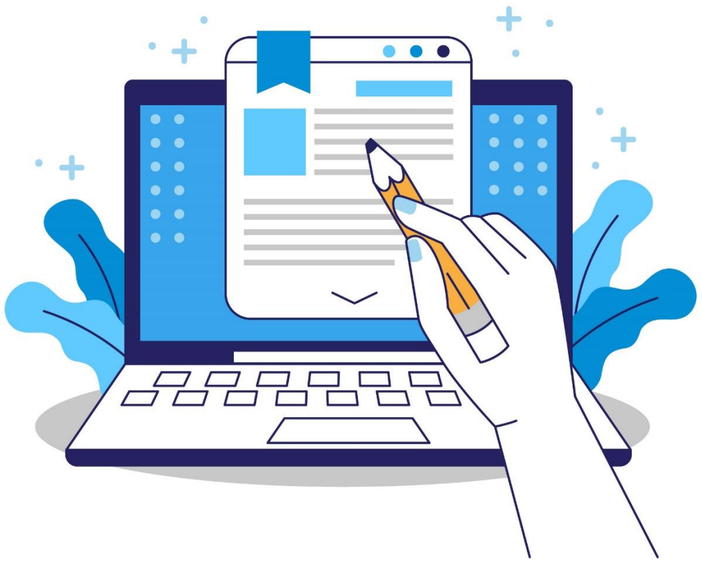
edureka!
a Veranda Enterprise

ILLINOIS TECH

Neural Networks and Deep Learning Foundation

Topics

- Introduction to Deep Learning
- Biological Neuron vs. Artificial Neuron
- Perceptron and Multilayer Perceptron
- Activation, Cost and Loss Functions
- Forward propagation
- Understanding Backpropagation Using Gradient Descent
- Process Of Building A Neural Network
- Weight initialization strategies
- Deep vs. Shallow Neural Networks
- Understanding Epochs, Batch Size, and Learning Rate
- Overfitting and Underfitting
- Computational Graphs
- Monitoring Training and Validation Loss
- Popular Deep Learning Frameworks
- Creating a simple DNN using Keras



Learning Objectives

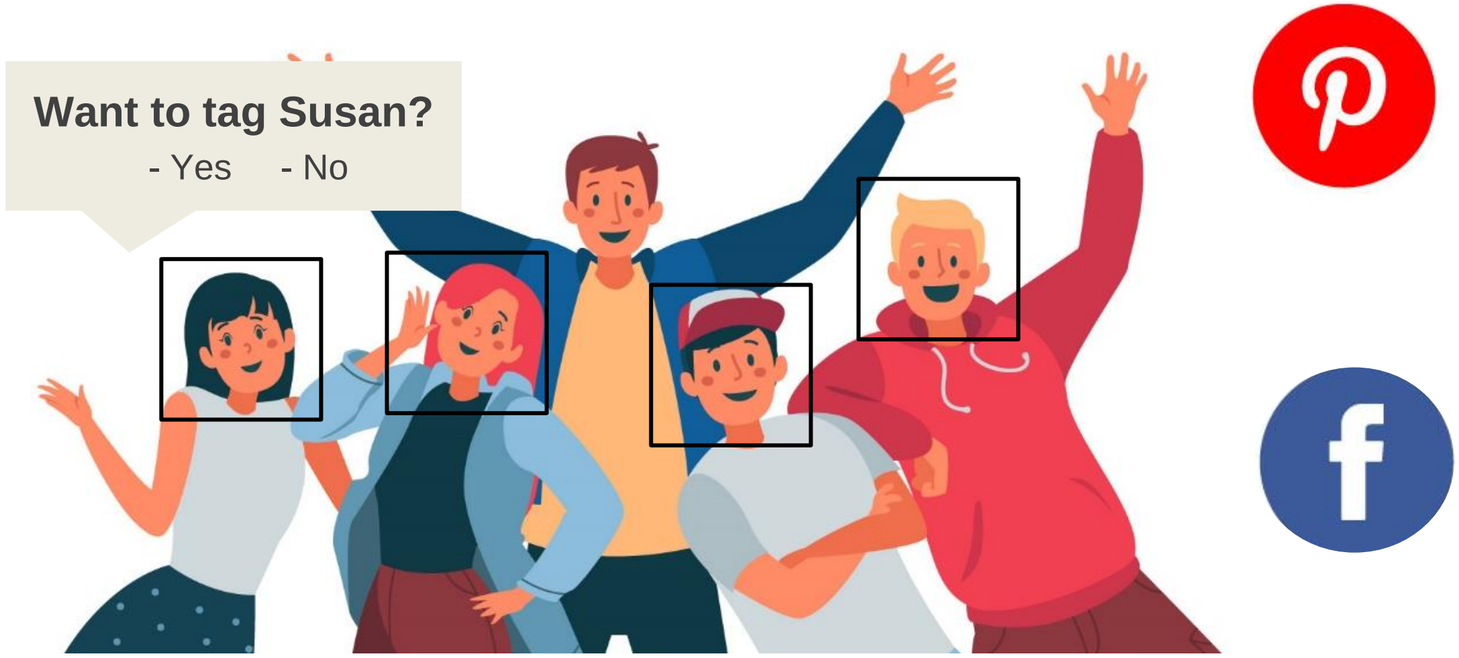
By the end of this lesson, you will be able to:

- Understand deep learning basics and neural network components (neurons, MLPs, activations).
- Explain forward/backward propagation, cost/loss functions, and gradient descent.
- Build and train neural networks with proper weight initialization and tuning.
- Use Keras to implement and monitor basic deep learning models.

Introduction To Deep Learning

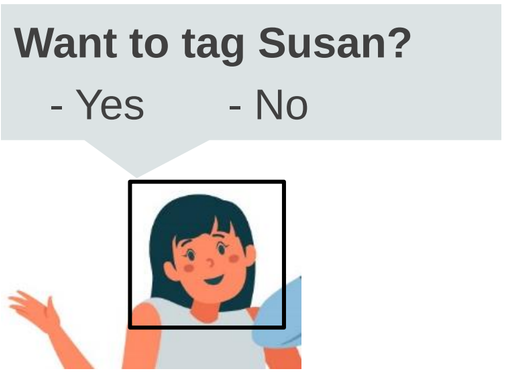
Use Case: Social Media

- Social media platforms like **Facebook & Pinterest** benefit heavily from Deep Learning technologies
- They are used for friend **recommendations**, pin **suggestions**, face **tagging** etc.



How Social Media Uses Deep Learning?

- **Example -** Face tagging on Facebook:



Facebook uses Deep Learning for **face recognition** in pictures uploaded by a user and then **tagging friends** from user's friend list

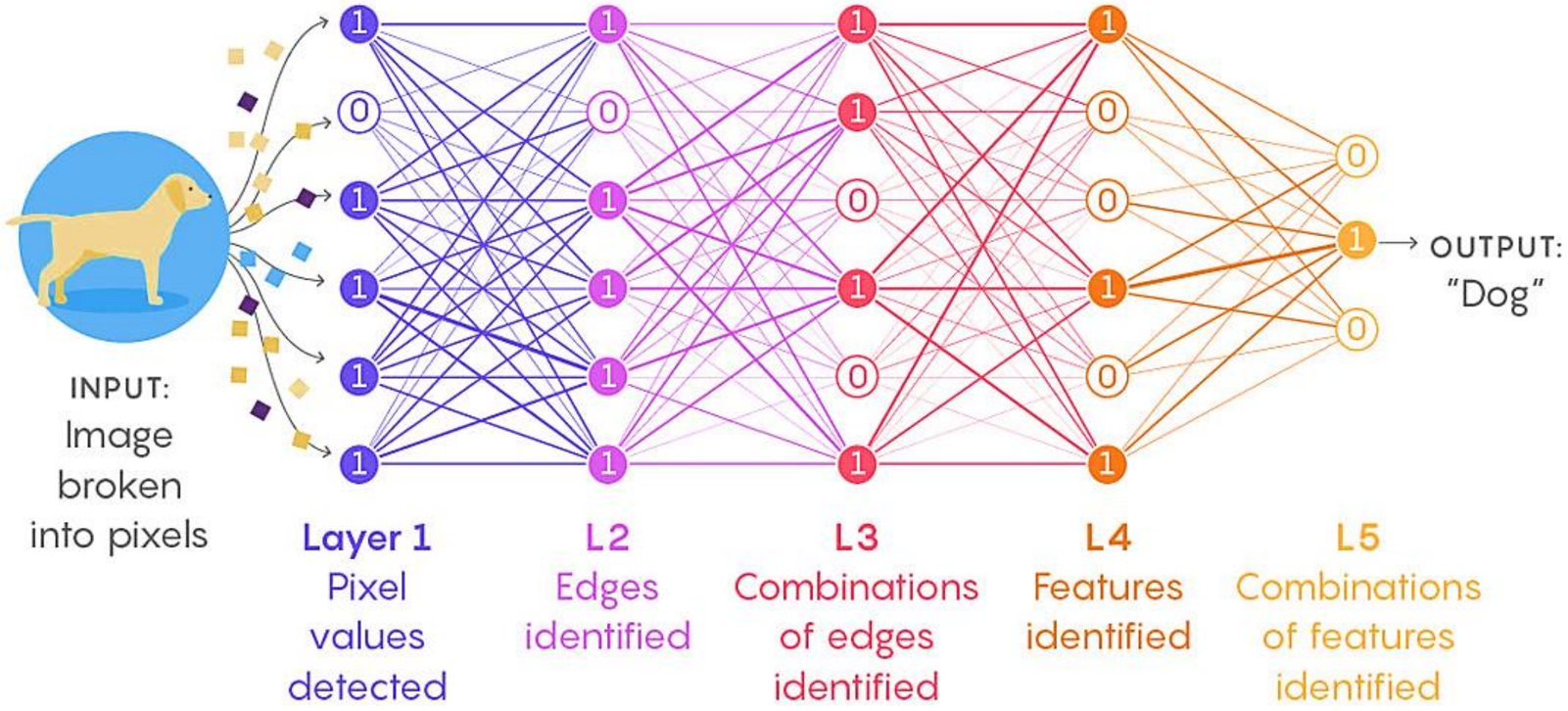
This information is also used to-

- Create **personalized feed** for the user
- Provide **friend suggestions**
- Show **relevant advertisements**

What is Deep Learning?

It is a **subset of machine learning** that takes the data, performs a function, and progressively gets better at it.

The algorithms are inspired by the structure and function of the brain, called artificial neural networks.



Why Deep Learning?

Deep learning is an Artificial Neural Network with many layers.

- Artificial Neural Networks (ANNs) are the core of Deep Learning
- ANNs are powerful, versatile and scalable
- ANNs find applications in-

Google Images

Classifying billions of images

Apple's Siri

Powerful Speech Recognition

Netflix

Recommending movies to millions of users

DeepMind's AlphaGo

Beating the world champion in game of Go

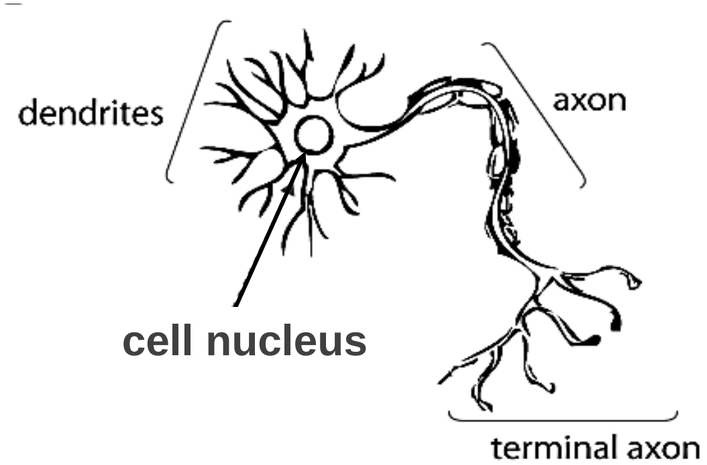
Biological Neuron vs. Artificial Neuron

Biological Vs. Artificial Neuron

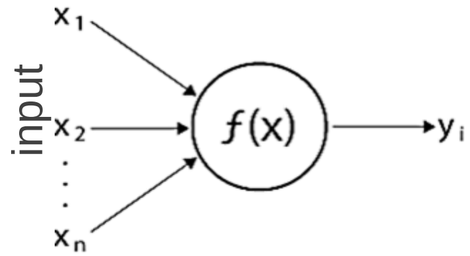
Biological Neuron

Artificial Neuron

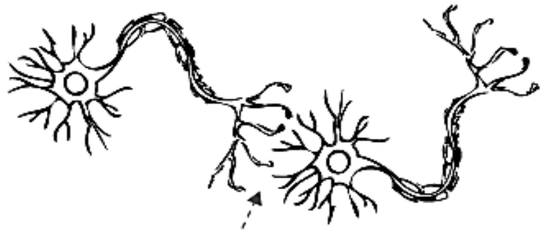
Single



output

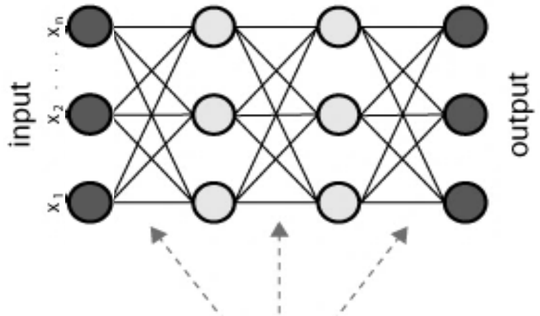


Multiple



(Information Transfer)

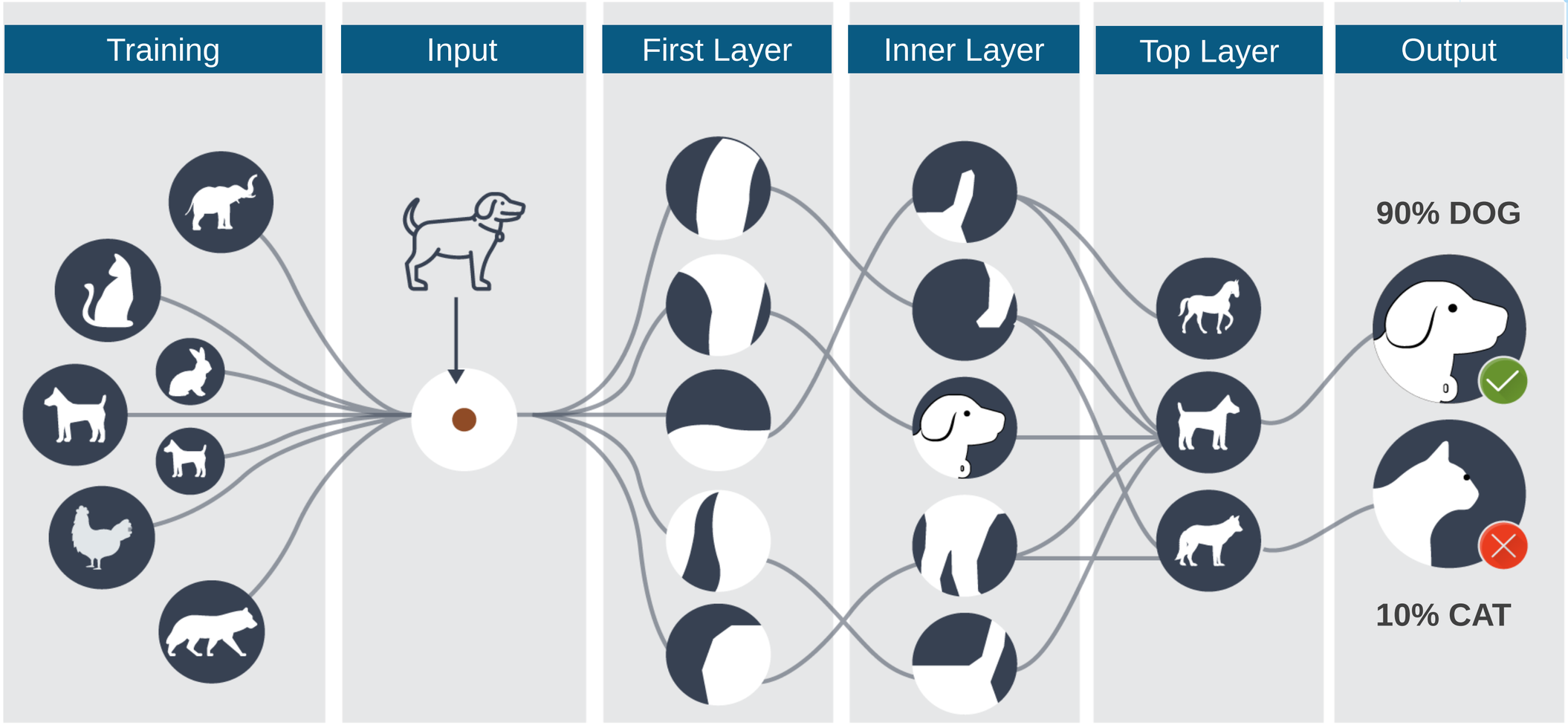
output



synapse

(Information Transfer)

How Does a Machine Recognize the Image?

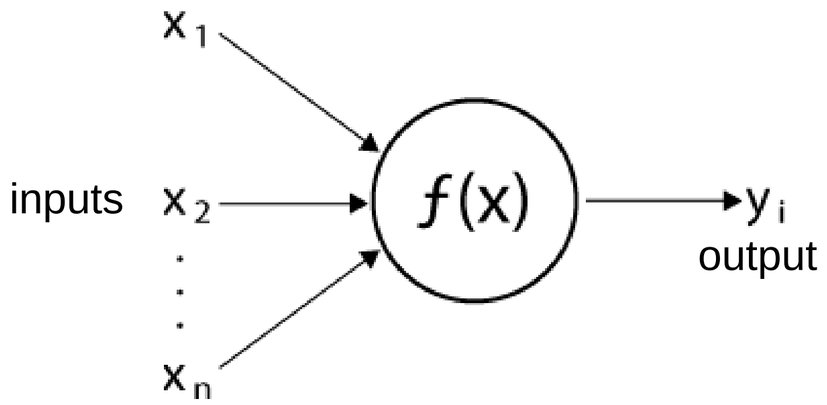


Introduction to Perceptron and Multilayer Perceptron

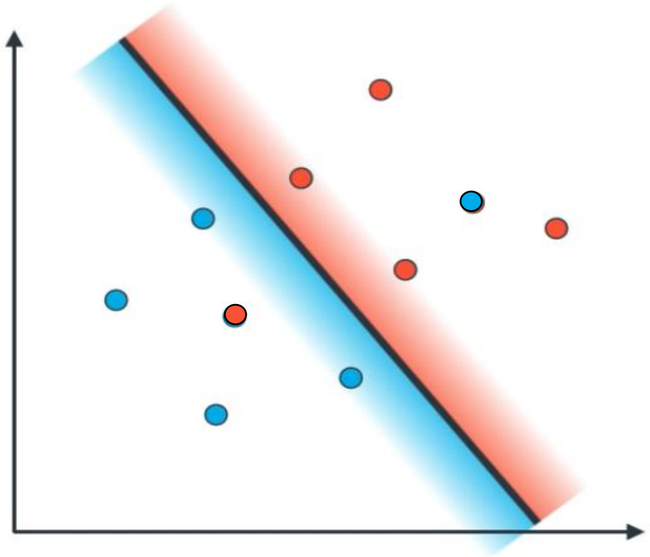
The Perceptron

A perceptron is the simplest type of artificial neural network unit, inspired by biological neurons. It takes **multiple inputs, applies weights, sums them up, and passes the result through an activation function** to produce a binary output.

Working of a Perceptron:



Limitations Of Single Layer Perceptron



A single-layer perceptron is **not capable of separating data points with a single line**

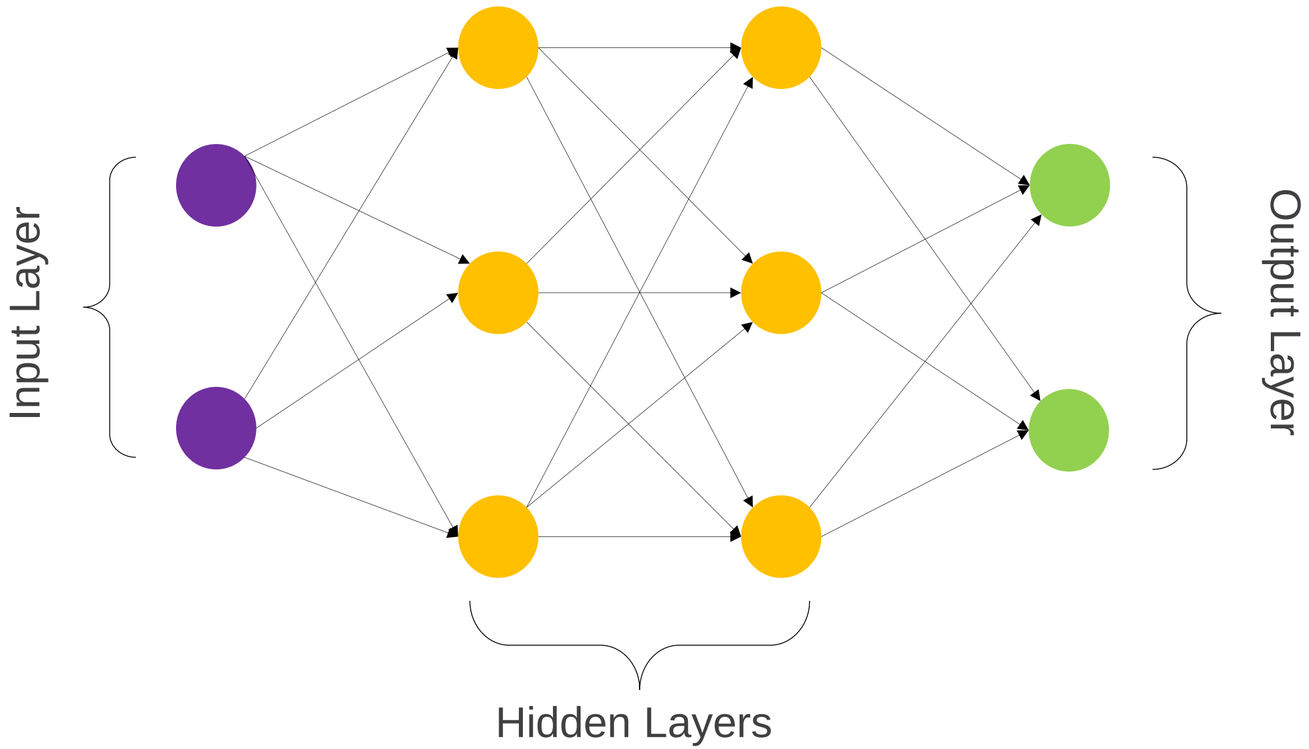
In addition, it **can not extract complex feature** dependency and thus not good at solving complex data problems

To solve this problem, **Multi-layer perceptron** came into picture

Multilayer Perceptron (MLP)

MLP is a artificial neural network with **more than one hidden layers.**

MLP consists of **three main layers:**



Activation, Cost and Loss Functions

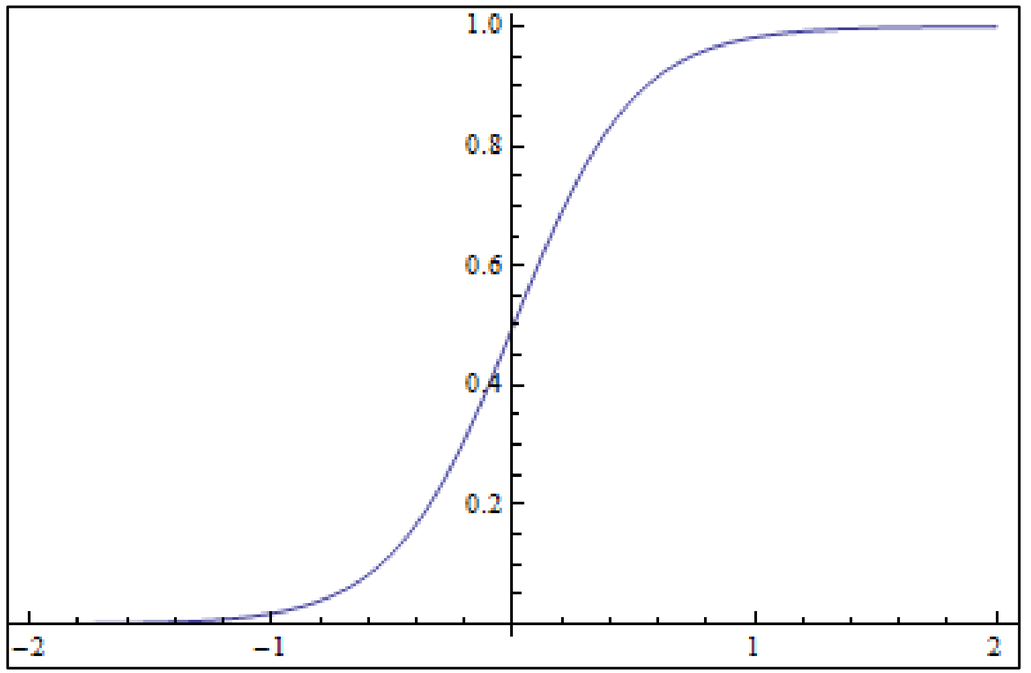
Activation Function

- Activation function, also known as **transfer function**, introduces non-linearity in neural networks.
- This non-linear transformation is introduced to **learn the complex underlying patterns** in the data.
- Without this function, a neuron simply resembles a linear regression.

Example: Sigmoid Function-

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

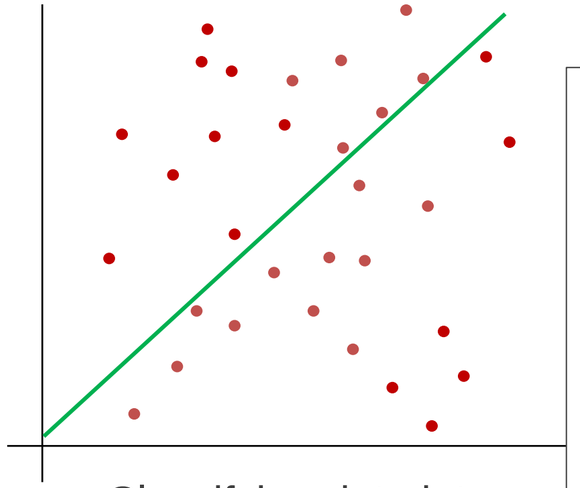
where z = b + $\sum_{i=1}^{na}$ x_i w_i



Importance Of An Activation Function

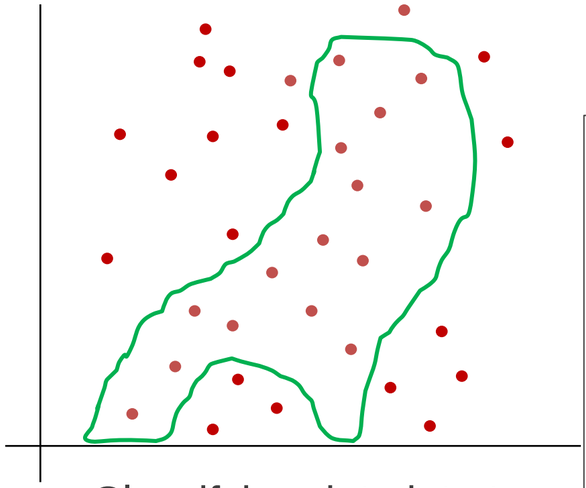
Purpose of an activation function is to **add non-linearity** into the neural network

Consider the below classification scenario:



Classifying data into two labels using a linear function

A linear function produces linear decision boundaries irrespective of how deep the network is

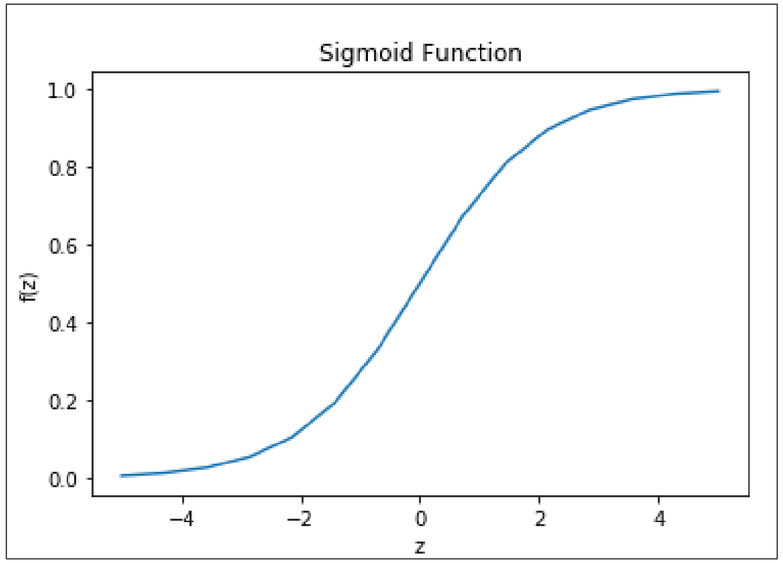


Classifying data into two labels using a non-linear function

Non-linearities allow us to model complex functions with improved capabilities

The Sigmoid Function

- One of the most commonly used functions.
- It scales all values between 0 and 1, thus used for predicting probabilities.



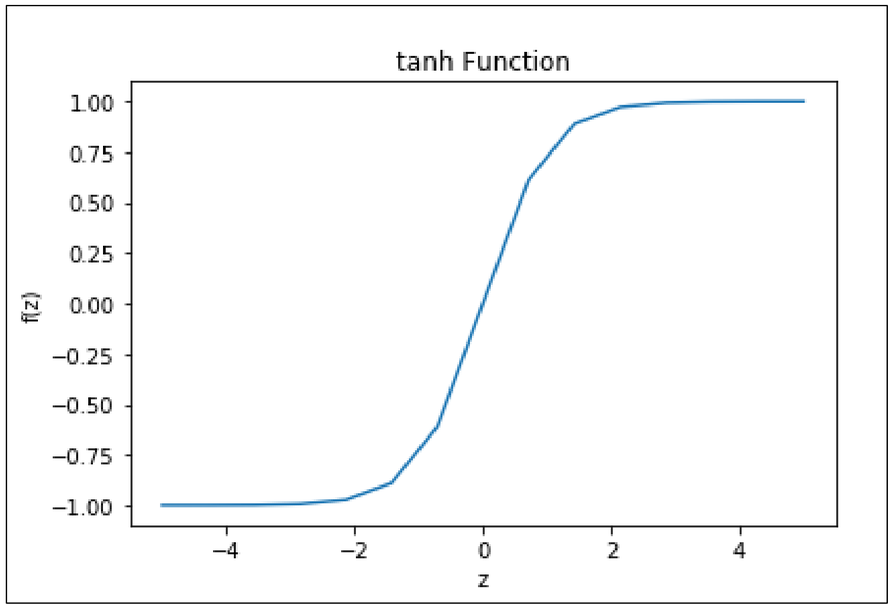
$$f(z) = \frac{1}{(1+ e^{-z})}$$

tf.nn.sigmoid()

It suffers from the problem of **"vanishing gradients"** which makes it unsuitable for backpropagation.

The tanh Function

- A hyperbolic tangent (tanh) function maps input values to outputs between (-1 , 1)
- This function solves the “vanishing gradients” problem to some extent

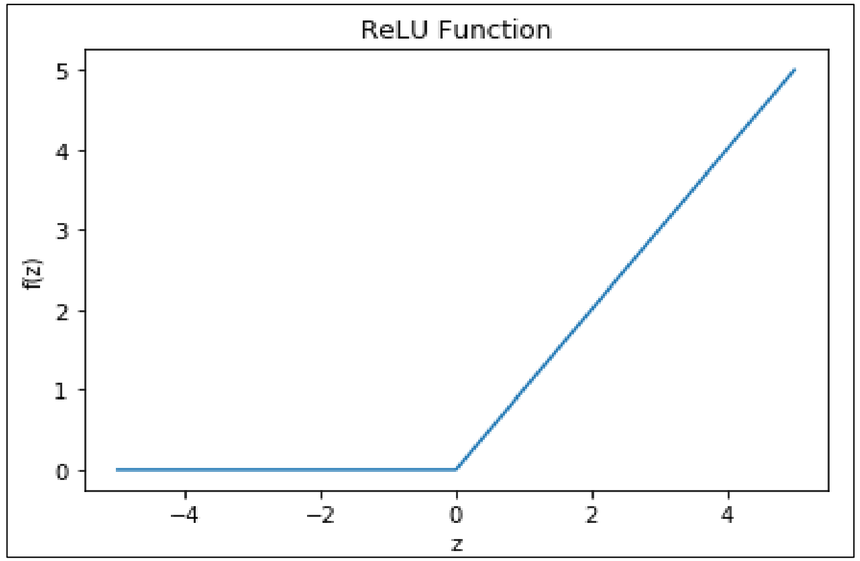


$$f(z) = \frac{(1 - e^{-2z})}{(1 + e^{-2z})}$$

tf.nn.tanh()

The Rectified Linear Unit (ReLU) Function

- One of the most commonly used activation functions
- It outputs a value from 0 to ∞ as per the below equation
- ReLU does not saturate unlike Sigmoid & tanh functions, which saturate in both the directions. Thus, ReLU minimises the vanishing gradients problem to a great extent



$$f(z) = \begin{cases} 0, & for\ z < 0 \\ z, & for\ z \geq 0 \end{cases}$$

tf.nn.relu()

The Softmax Function

- It is a generalization of Sigmoid function
- Softmax maps outputs to values between (0,1) such that their sum is 1
- Thus, Softmax is applied to the final layer of a network in a multi-class classification

z_n	$f(z_n)$
-5	0
-3	0.0003
0	0.0064
2	0.0471
5	0.9462

$$f(z_n) = \frac{e^{z_n}}{\sum_{i=1}^{n} e^{z_n}}$$ where, n is the number of classes

tf.nn.softmax()

Loss Function

- **Mean Squared Error (MSE):** for regression tasks

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- **Binary Cross-Entropy:** for binary classification

$$\text{BCE} = -\frac{1}{\text{n}}\sum_{i=1}^{\text{n}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

- **Categorical Cross-Entropy:** for multi-class classification.

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output} \\ \text{size}}} y_i \cdot \log \hat{y}_i$$

n: Number of training samples;

y_i**:** Actual Output;

$(y_i)\hat{}$**:** Predicted class probabilities;

Cost Function

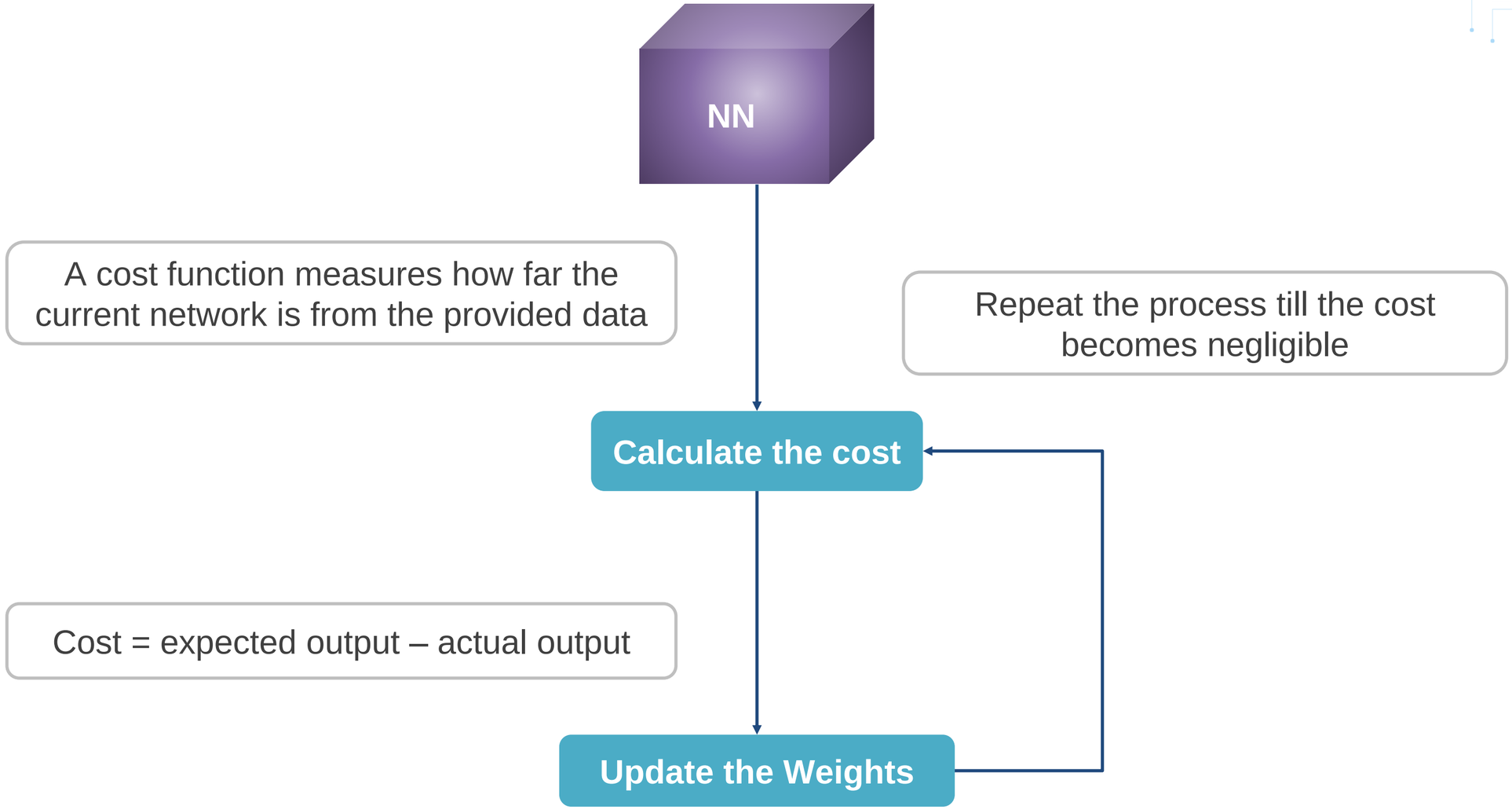
- A function, called Cost Function or Loss Function tells **how well a neural network performs**
- We will use a simple Cost Function defined as mean squared difference between the actual and the predicted output

Cost Function $J = \frac{1}{n}\sum_{i=1}^{n}(yi - \widehat{yi})^2$

- n: Number of training samples;
- yi: Actual Output;
- $\widehat{yi}$: Predicted Output

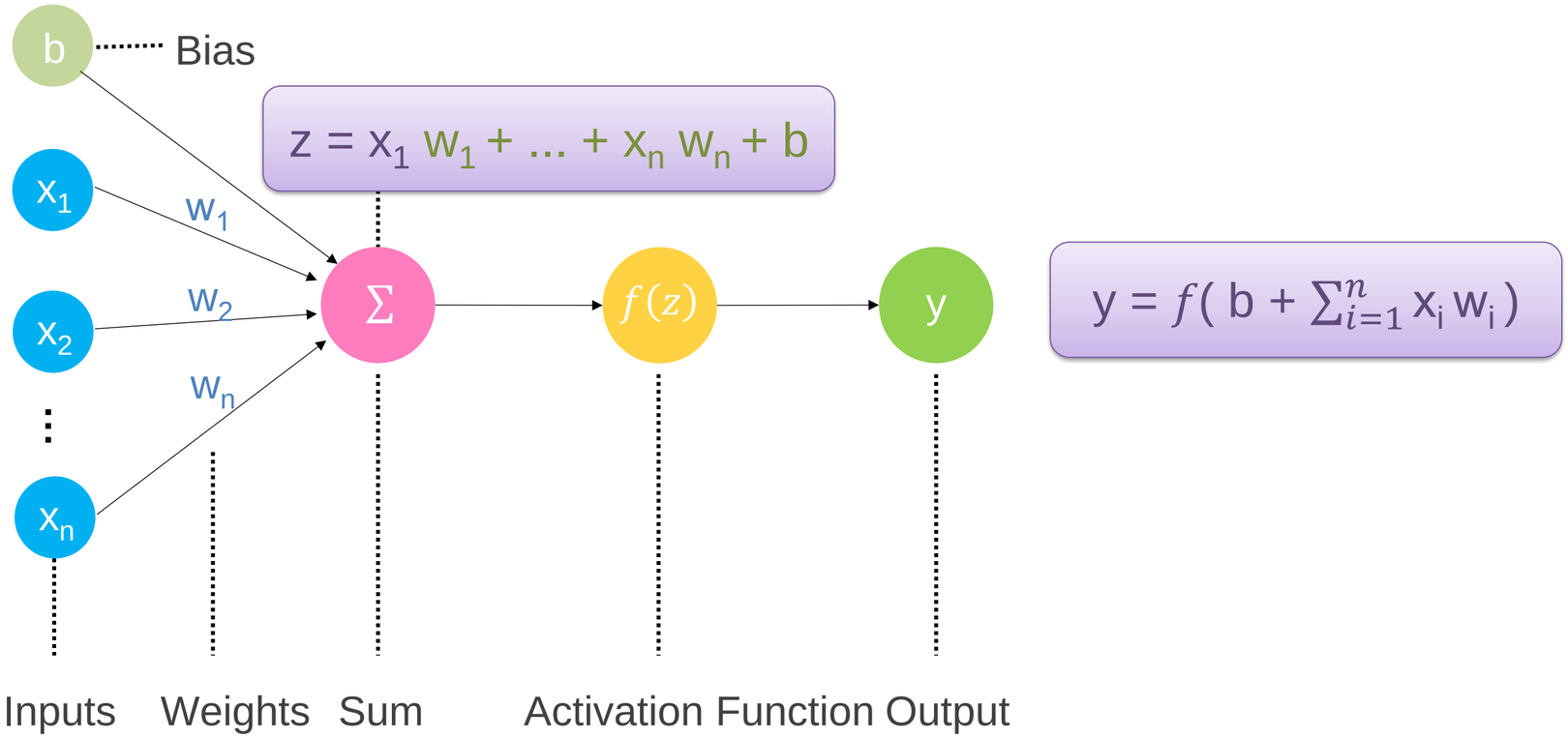
The aim is to minimize the cost to get the best performing neural network!

Improving Performance of a Neural Network



Forward Propagation

Forward Propagation

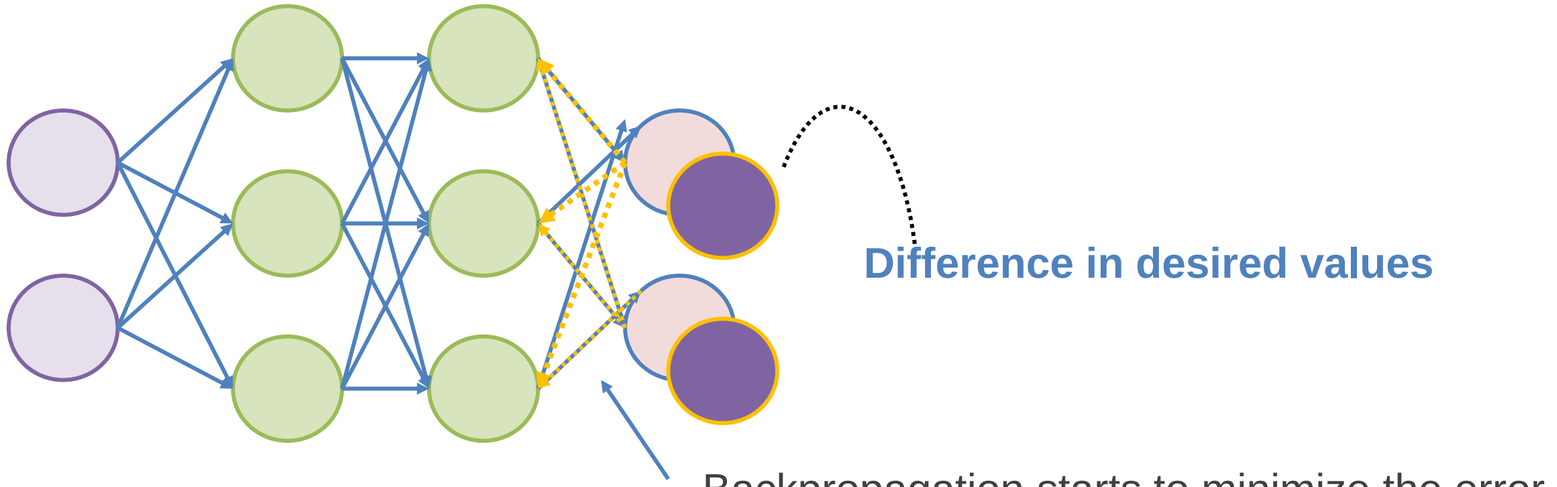


Understanding Backpropagation Using Gradient Descent

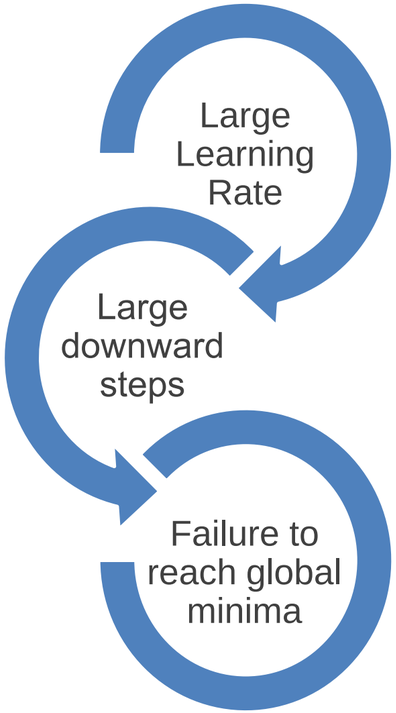
What Is Backpropagation?

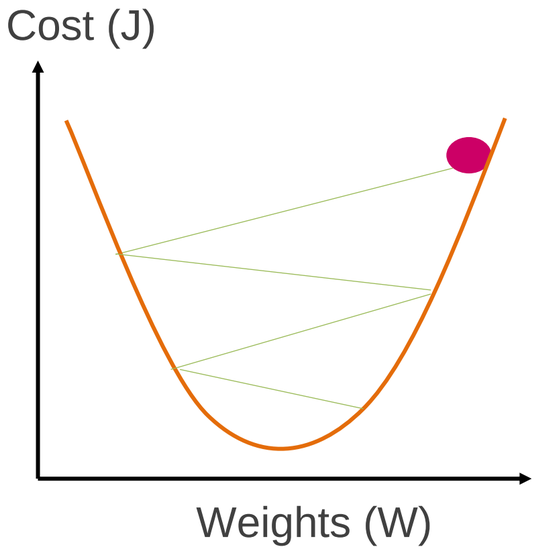
Algorithm to improve the accuracy of prediction and quick calculation of derivatives

- A Learning algorithm to compute gradient descent with respect to weight
- Weights values are manipulated to get the predicted result
- Backpropagation is used to update the weights from output to input

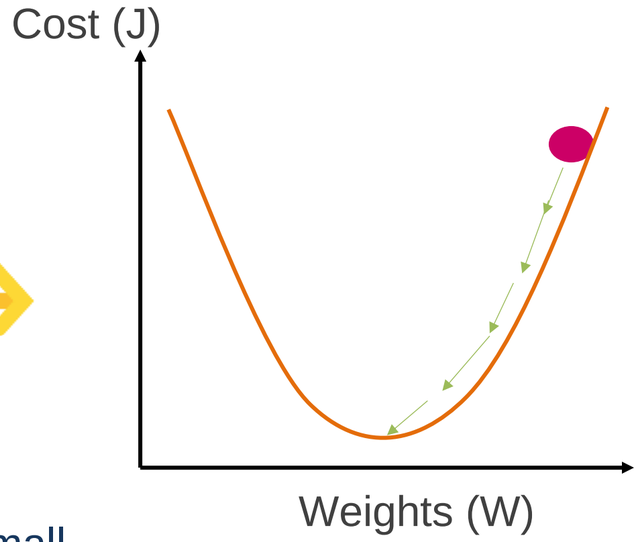


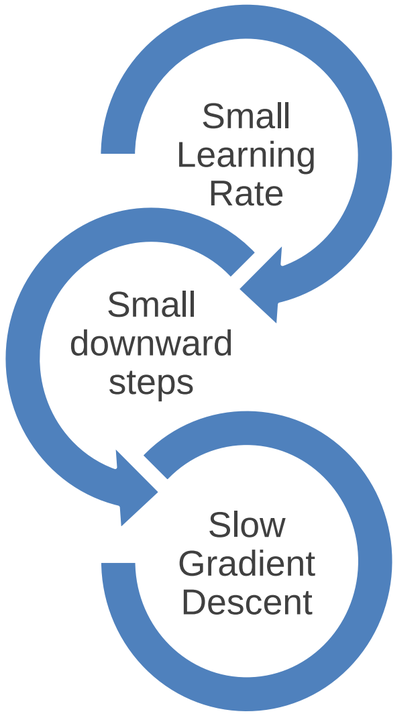
Learning Rate (α)



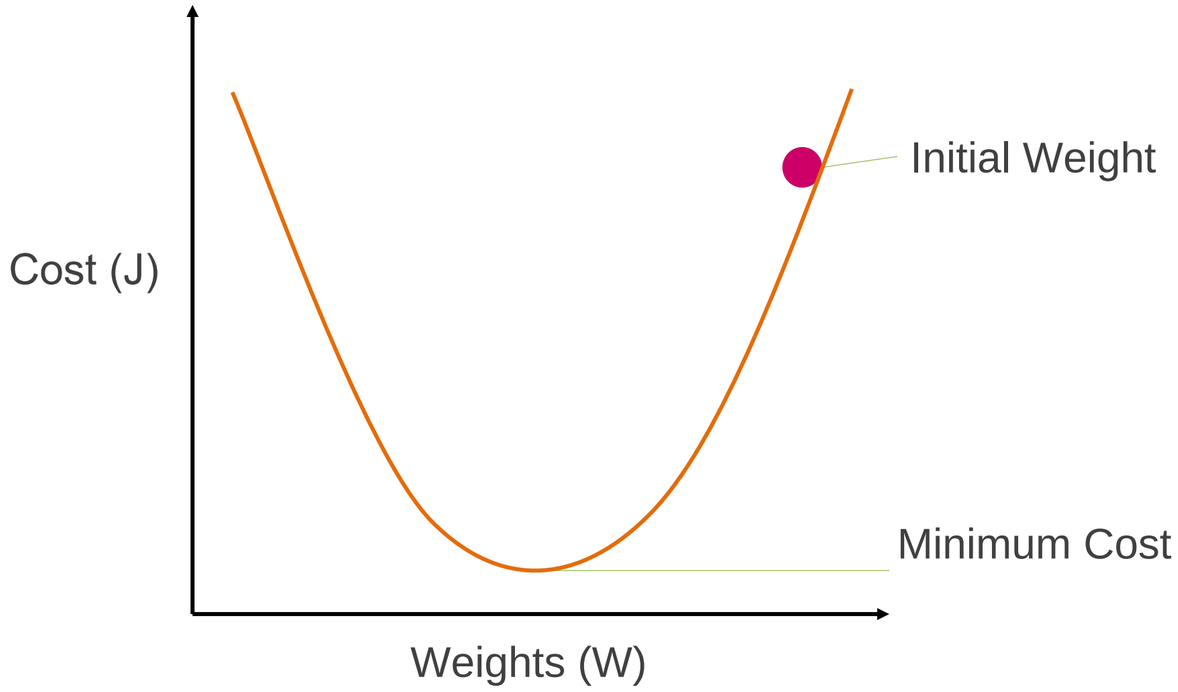


Large vs. small
Learning Rate





Applying Gradient Descent To Cost Function



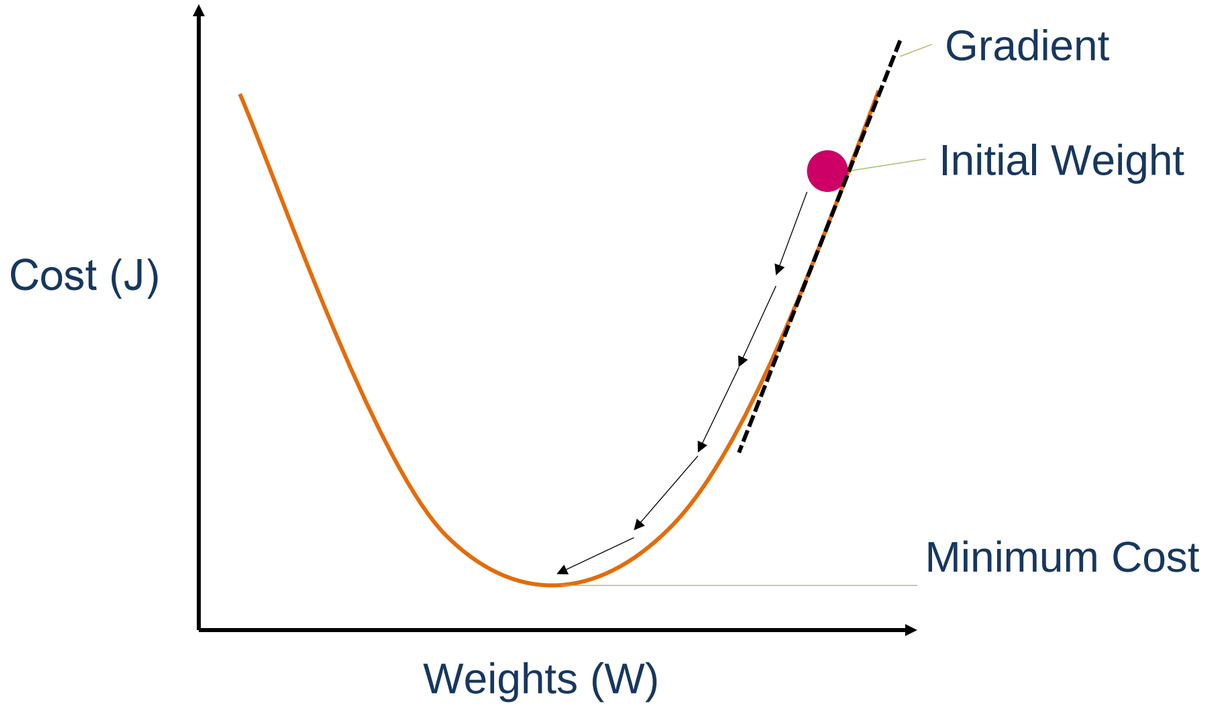
Gradients are used from moving from one point to another.

Thus, gradient of **Cost Function** is calculated with respect to **Initial Weight** to reach the

Minimum Cost - $\frac{\partial J}{\partial W}$

Gradient is simply differentiation of a function

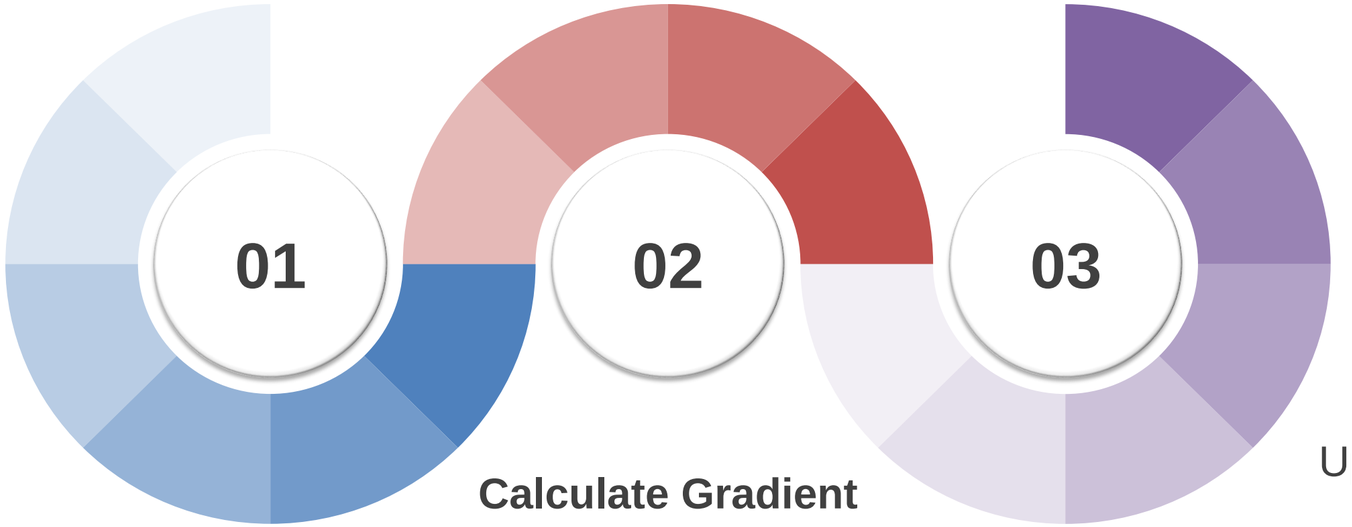
Applying Gradient Descent To Cost Function



- Calculate gradient at each step and move in the direction of descend
- With each step, weights are updated to a value where the cost is minimum

Process Of Building A Neural Network

Process Of Building A Neural Network



Perform Forward Propagation

Use random initial weights to compute the output

Calculate Gradient

Move backward from output layer to input, calculating gradient of Cost with respect to all Weights $\frac{\partial J}{\partial W}$

Update Weights

Update old Weights using the rule:

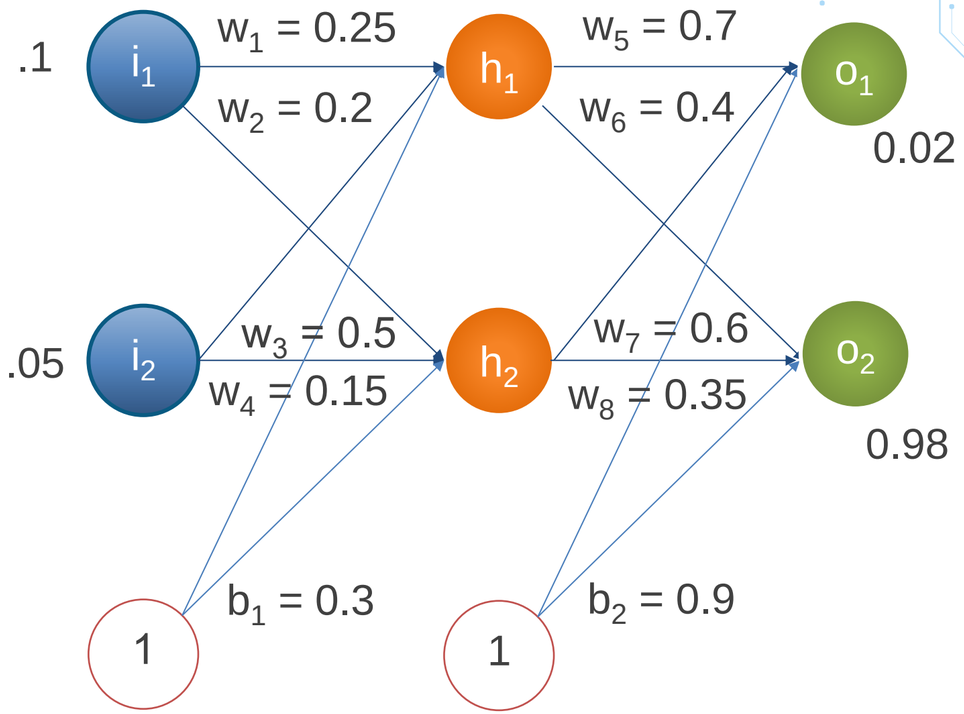
$$W_{new} = W_{old} - \alpha \frac{\partial J}{\partial W}$$

Backpropagation

Step-By-Step Learning Of A Neural Network

Consider the following neural network with:

- Two input neurons
- One hidden layer with two neurons
- Two output neurons



The goal of backpropagation is to optimize the weights so that the network can predict more accurate outputs for the given inputs

Learning Of A NN: Forward Propagation

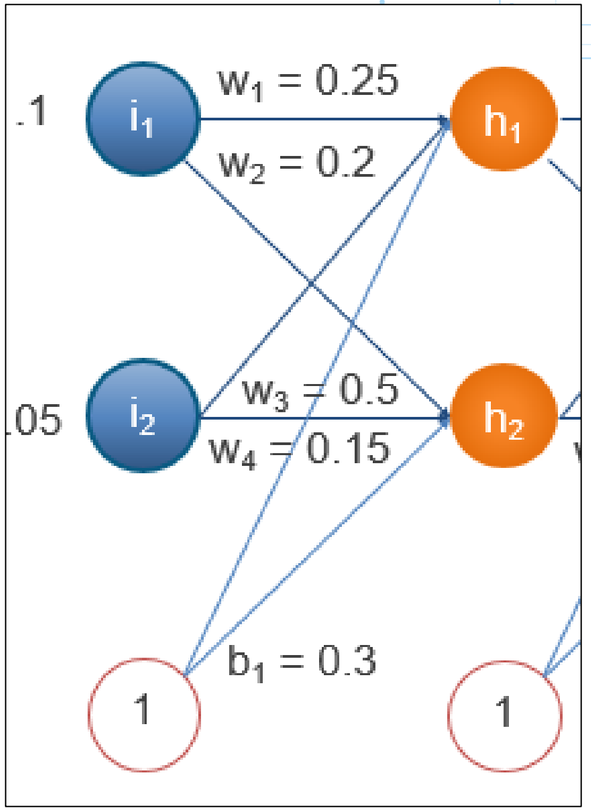
- Calculate the inputs to h1 and h2: Sum of weighted inputs

$$in_{h1} = w_1 \cdot i_1 + w_3 \cdot i_2 + b_1 \cdot 1 = 0.25*0.1 + 0.5*0.05 + 0.3 = \mathbf{0.35}$$

$$in_{h2} = w_2 \cdot i_1 + w_4 \cdot i_2 + b_1 \cdot 1 = 0.2*0.1 + 0.15*0.05 + 0.3 = \mathbf{0.3275}$$

- Calculate the outputs from h1 and h2: Application of Activation Function

$$out_{h1} = \frac{1}{1+ e^{-inh_1}} = \mathbf{0.5866} \qquad out_{h2} = \frac{1}{1+ e^{-inh_2}} = \mathbf{0.5812}$$



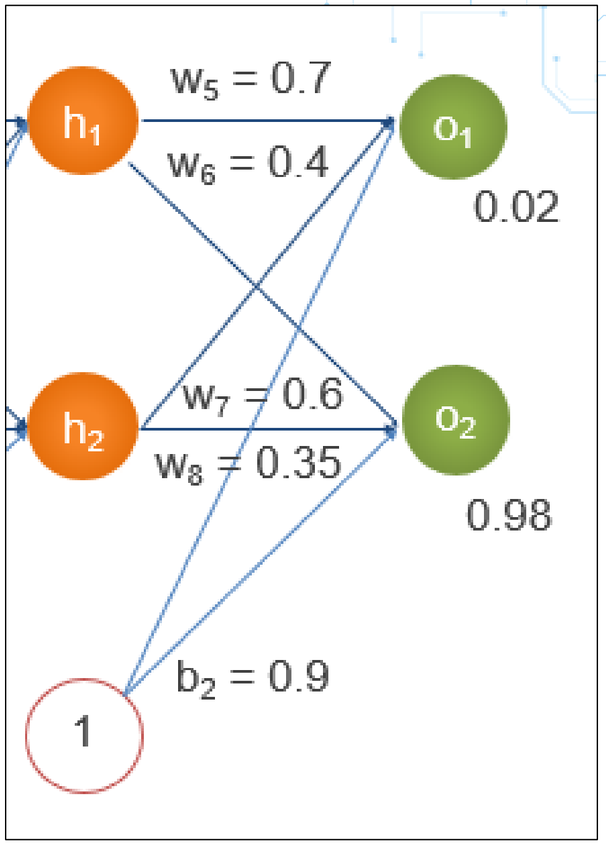
Learning Of A NN: Forward Propagation(Contd)

- Calculate the inputs to o1 and o2: Sum of weighted outputs from hidden layer

$$\boldsymbol{in_{o_1}} = w_5 \cdot h_1 + w_7 \cdot h_2 + b_2 \cdot 1 = \mathbf{1.3415}$$
$$\boldsymbol{in_{o_2}} = w_6 \cdot h_1 + w_8 \cdot h_2 + b_2 \cdot 1 = \mathbf{1.1546}$$

- Calculate the outputs from o1 and o2: Application of Activation function

$$\boldsymbol{out_{o_1}} = \frac{1}{1+e^{-ino_1}} = \mathbf{0.7927} \qquad \boldsymbol{out_{o_2}} = \frac{1}{1+e^{-ino_2}} = \mathbf{0.7603}$$



Learning Of A NN: Calculating The Error

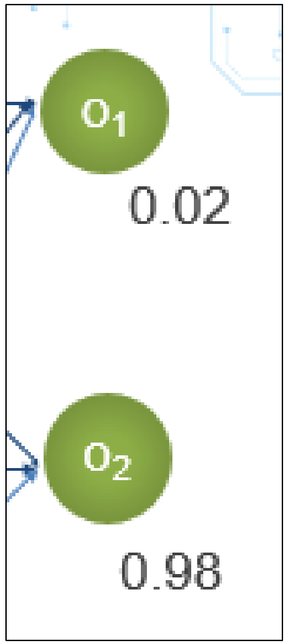
- Calculate the errors for o1 and o2

$$\boldsymbol{E_{o_1}} = \frac{1}{2}(target_{o_1} - outo_1)^2 = \mathbf{0.2985}$$

$$\boldsymbol{E_{o_2}} = \frac{1}{2}(target_{o_2} - outo_2)^2 = \mathbf{0.0241}$$

- Calculate the total error as shown below

$$\boldsymbol{E_{total}} = E_{o_1} + E_{o_2} = \mathbf{0.3226}$$



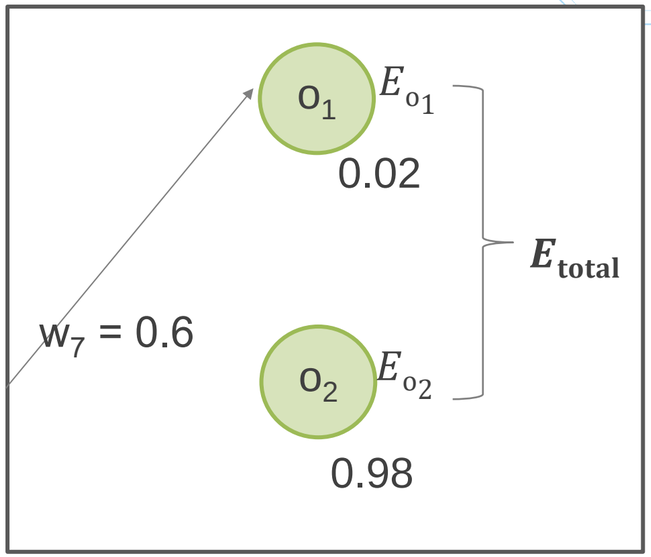
Learning Of A NN: Backpropagation

- Calculate the error gradient of $\boldsymbol{E}\mathbf{total}$ with respect to $\boldsymbol{w7}$
- Using Chain **Rule,** it can be calculated as:

$$\frac{\partial \boldsymbol{E}_{\text{total}}}{\partial \boldsymbol{w}_7} = \frac{\partial \boldsymbol{E}_{\text{total}}}{\partial \boldsymbol{out}_{o_1}} * \frac{\partial \boldsymbol{out}_{o_1}}{\partial \boldsymbol{in}_{o_1}} * \frac{\partial \boldsymbol{in}_{o_1}}{\partial \boldsymbol{w}_7} = \mathbf{0.0745}$$

- $\boldsymbol{w7}$ is updated as follows:

$$\boldsymbol{w7new} = w7old - \alpha * \frac{\partial E\text{total}}{\partial w7} = \mathbf{0.5628} \text{ ……..(Using } \alpha = 0.5)$$



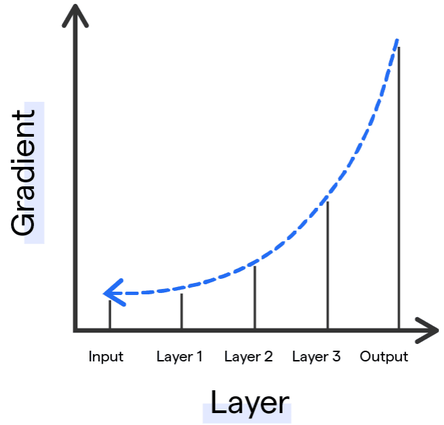
Similarly, all other weights are updated

Weight Initialization Strategies

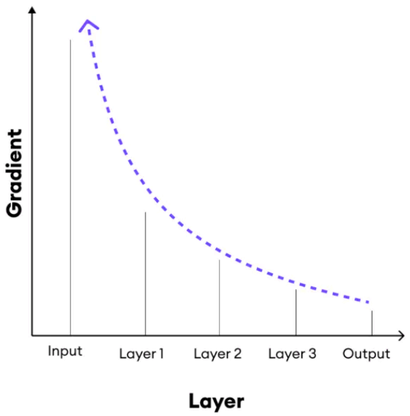
Weight initialization

What happens if there is no good initialization?

Slow convergence



Vanishing gradients



Exploding gradients

Zero and Constant Initialization

Point	Zero Initialization	Constant Initialization
Definition	All weights are initialized to exactly **zero**.	All weights are initialized to a **constant non-zero value**
Technique	Set every weight ***w = 0.***	Set every weight ***w = c*** where ccc is a constant.
Purpose	Start with simplicity, but it **fails for training**.	Introduces small differences but **still bad for deep learning**.
Problem	No symmetry breaking → all neurons learn the same thing → **no learning**.	Same as zero: **neurons behave identically** → poor learning.
Best Used For	Not recommended for any serious models.	Only for experiments or extremely basic models.
Example	***W = 0***	***W = 0.01 or w = 0.5***

Random and Xavier/Glorot Initialization

Point	Random Initialization	Xavier or Glorot Initialization
Definition	Weights are initialized with **random values**	Weights are initialized **randomly**, but with a **carefully calculated variance** based on network size.
Technique	Randomly pick values without considering input/output sizes.	Randomly pick values with variance $\frac{2}{n_{in}+n_{out}}$, balancing inputs and outputs.
Purpose	Breaks symmetry better than zero but can **still cause problems**.	Maintain stable activations and gradients across layers for **faster and safer training**.
Problem	Can cause **vanishing** or **exploding**	Solves vanishing/exploding problems by **scaling** weights.
Best Used For	Very shallow or experimental networks.	Deep neural networks, with **sigmoid** or **tanh** activations.
Example	$w \sim U(-1, 1)$ or $w \sim N(0, 1)$	$w \sim U\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$

Shallow vs. Deep Neural Networks

Shallow vs. Deep Neural Networks

Aspect	Shallow Neural Networks	Deep Neural Networks
Definition	Neural networks with few layers	Neural networks with many layers
Complexity	Low complexity	High complexity
Learning Capacity	Limited learning capacity	Higher learning capacity
Risk of Overfitting	Lower	Higher
Data Requirement	Requires less data	Requires more data for effective training
Parameter Count	Fewer parameters	Many more parameters
Computational Resources	Requires less computation	Needs more resources (e.g., GPUs)
Interpretability	Easier to interpret	More difficult to interpret
Examples	Single-layer Perceptron, Logistic Regression	CNNs, RNNs

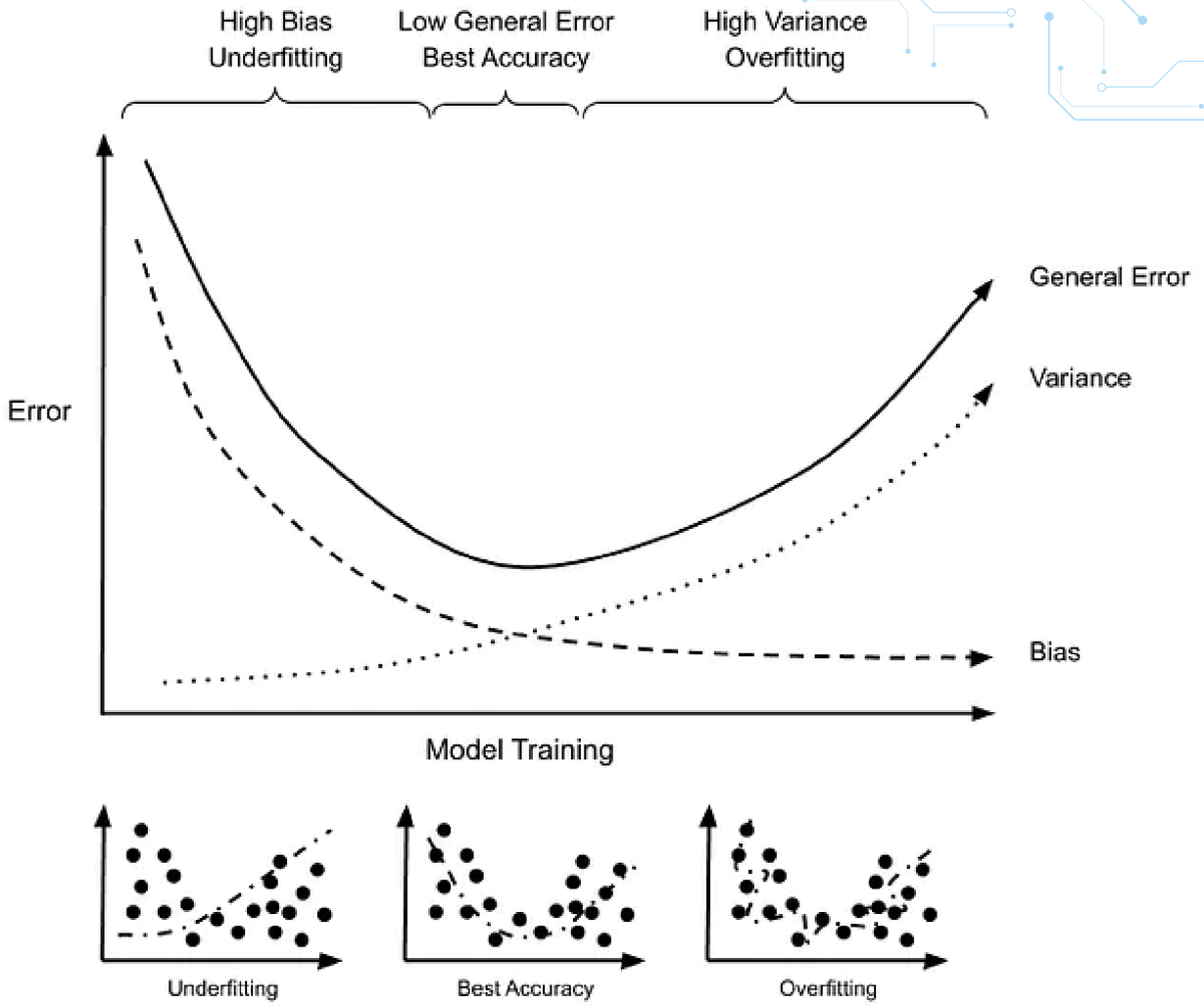
Bias-Variance Trade-off

Bias-Variance Trade Off

The bias-variance trade-off refers to the careful balance between **underfitting and overfitting**. The objective is to choose the **right model complexity** that enables it to perform well on new, unseen data.

In Deep Learning :

- Deep Neural Networks can easily **reduce bias** due to their complexity.
- But they are also **prone to high variance** if not regularized well.



Understanding Epochs, Batch Size and Learning Rate

Epochs, Batch Size and Learning Rate

Batch Size: The number of training samples processed together in one training step

Epoch: One full pass through the entire training dataset, composed of multiple iterations.

Learning Rate: A hyperparameter that determining model weights are adjusted during each update step.

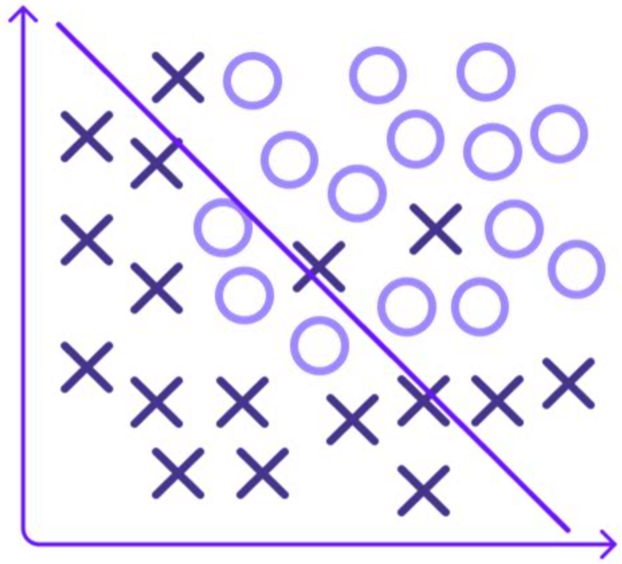
Let’s consider a simple example using a feedforward neural network for image classification. Suppose we have a **dataset of 10,000 images**, and we want to train our model with the following parameters:

- **Batch size:** 1000
- **Number of epochs:** 50
- **Learning rate:** 0.01

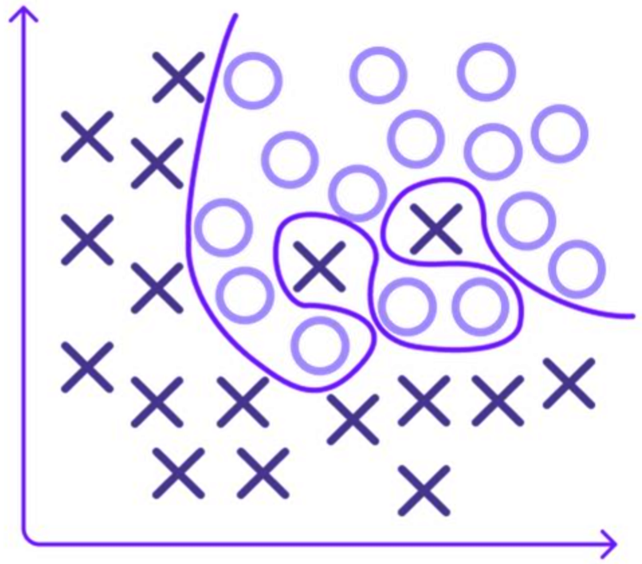
Overfitting and Underfitting

Overfitting and underfitting in deep learning

Underfitting occurs when a deep learning model is **too simple to capture** the underlying patterns in the training data.



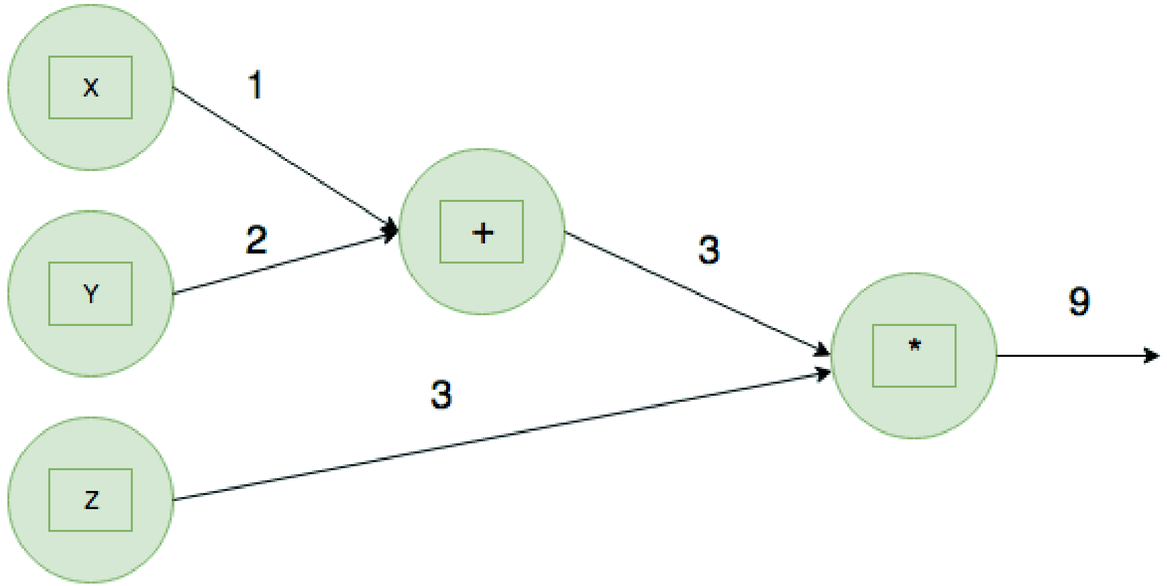
Overfitting happens when a deep learning model learns the **training data too well, including noise and irrelevant patterns.**



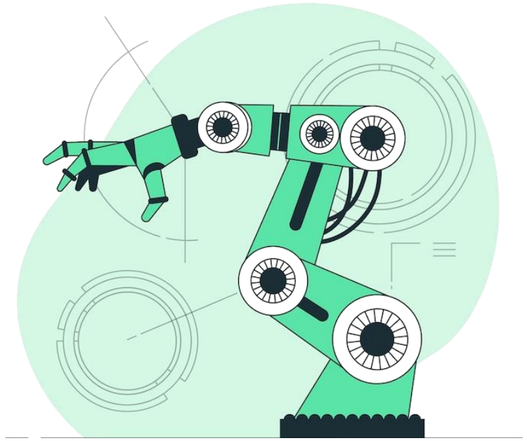
Computational Graphs in Deep Learning

Computational Graphs in Deep Learning

A computational graph in deep learning is a **structured representation** of the sequence of operations involved in a model, where nodes **represent mathematical operations** and edges represent the flow of data (tensors).



Why are they important?



Automatic Differentiation

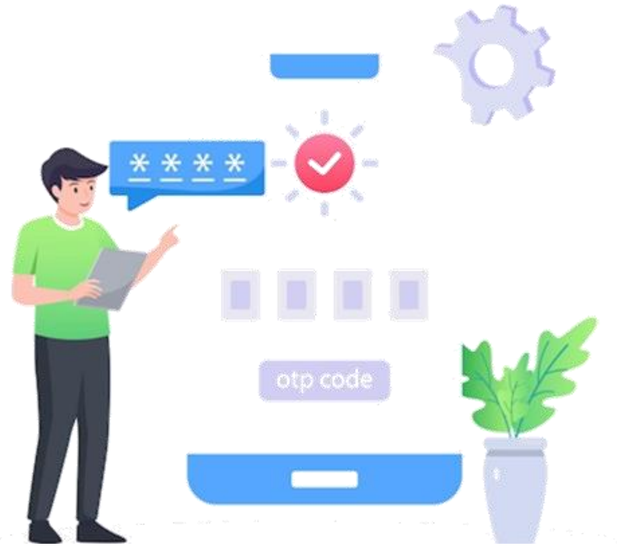
Modular Design

Optimization & Efficiency

Dynamic & Static Graph Support

Monitoring Training and Validation Loss

Training and Validation Loss in Deep Learning



Training Loss measures the error on the training dataset during learning.

Validation Loss measures the error on a separate validation dataset (unseen during training).

Ideal Behavior While Monitoring

Situation	Training Loss	Validation Loss	Interpretation
Good Fit	Decreasing	Decreasing	Model is learning correctly
Overfitting	Decreasing	Increasing	Model is memorizing, not generalizing
Underfitting	High	High	Model is too simple or not trained enough

Creating a simple DNN using Keras

Simple DNN using Keras

Code example

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.datasets import make_classification

# Step 1: Generate simple binary classification data
X, y = make_classification(n_samples=100, n_features=4, n_classes=2, random_state=42)

# Step 2: Define the simplest possible DNN
model = Sequential()
model.add(Dense(4, activation='relu', input_shape=(4,)))  # Hidden layer
model.add(Dense(1, activation='sigmoid'))                 # Output layer

# Step 3: Compile and train the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X, y, epochs=2, batch_size=10)
```

Output

```
Epoch 1/2
10/10 ━━━━━━━━━━━━━━━━━━━━ 4s 14ms/step - accuracy: 0.5475 - loss: 0.6564
Epoch 2/2
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.6058 - loss: 0.6476
```

Implementing a perceptron model (Demonstration)

Note: Refer to the Module 1: Demo 1 on LMS for detailed steps.

Training a basic MLP on the MNIST dataset (Demonstration)

Note: Refer to the Module 1: Demo 2 on LMS for detailed steps.

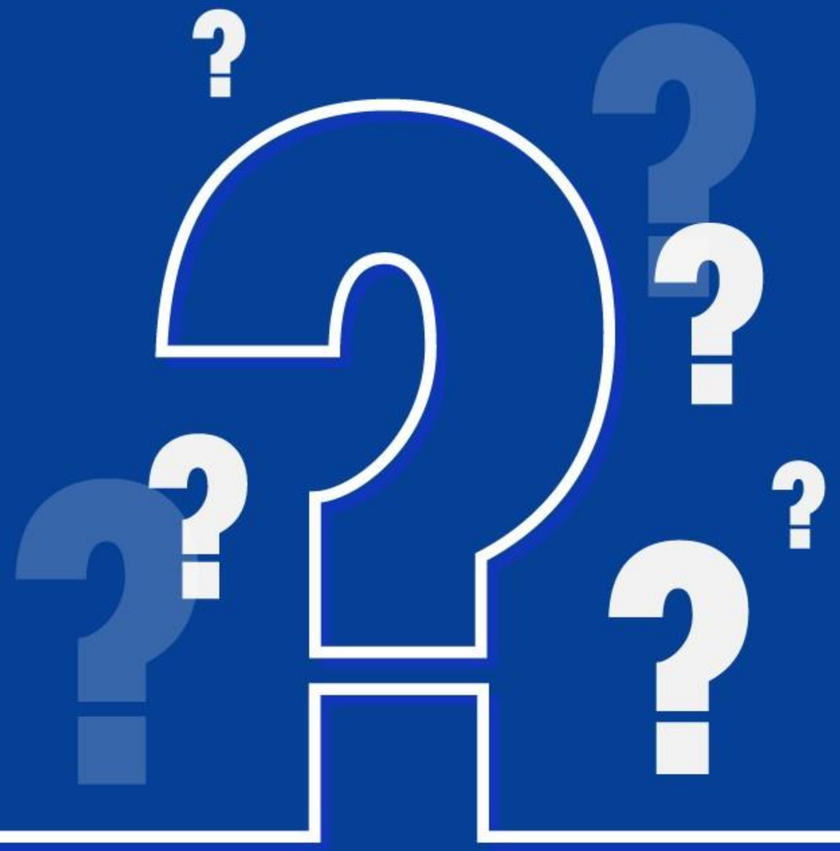
Summary

In this lesson, you have learned to:

- Explain NN basics: Perceptron, MLP, activation & loss functions.
- Understand forward/backpropagation & gradient descent.
- Grasp bias-variance, overfitting, and key training terms.
- Use Keras to build and monitor a simple DNN..

edureka! a Veranda ENTERPRISE

Questions



edureka! a Veranda ENTERPRISE

Feedback

edureka! a Veranda ENTERPRISE

Thank You

For information, Please Visit our Website
www.edureka.co