FORWARD PROPAGATION

TENSORFLOW

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML-powered applications.
we use that often to load data, complex mathematical computations

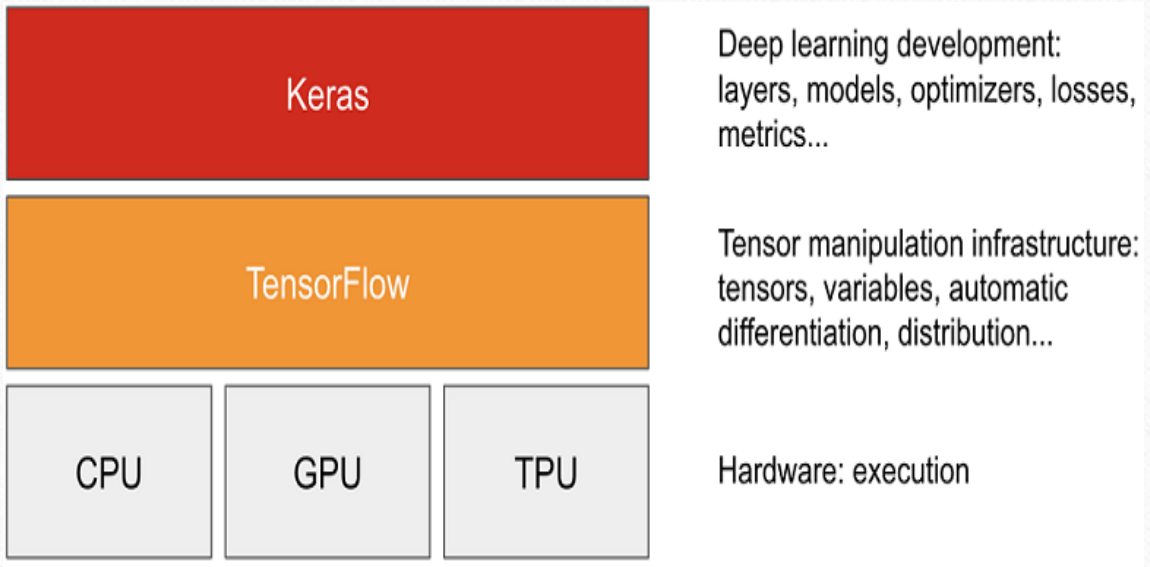
KERAS

- Keras is a high-level library that's built on top of Theano or TensorFlow. It provides a scikit-learn type API (written in Python) for building Neural Networks.
- Developers can use Keras to quickly build neural networks without worrying about the mathematical aspects of tensor algebra, numerical techniques, and optimization methods.
- **The key idea behind the development of Keras is to facilitate experimentations by fast prototyping.**

KERAS : high level

TENSORFLOW : both high level and low level

- Keras focuses on being modular, user-friendly, and extensible. It doesn't handle low-level computations; instead, it hands them off to another library called the Backend.
- Keras was adopted and integrated into TensorFlow in mid-2017. Users can access it via the tf.keras module. However, the Keras library can still operate separately and independently.



KERAS MODELS

SEQUENTIAL MODEL :

A **Sequential model** is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.
Sequential is the easiest way to build a **model** in Keras. It allows you to build a **model** layer by layer. Each layer has weights that correspond to the layer the follows it. We use the 'add()' function to add layers to our **model**.

FUNCTIONAL MODEL :

It is a more flexible API and is more advanced in nature. It is an alternative model against the Sequential Model that enables you to create models in a more complex manner. This model helps you to define multiple numbers of input and output.

A **deep neural network** (DNN) is an artificial **neural network** (ANN) with multiple layers between the input and output layers.

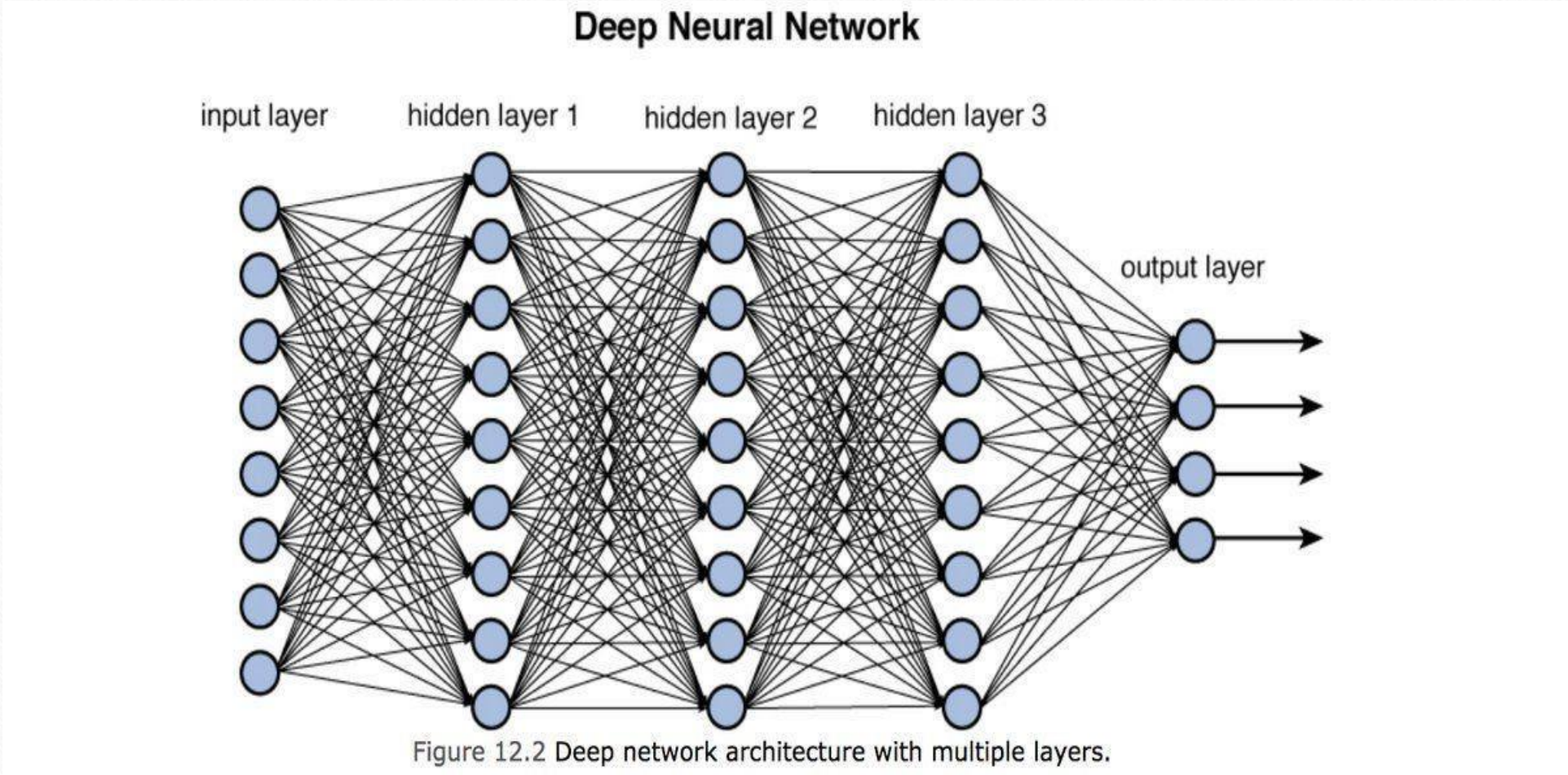
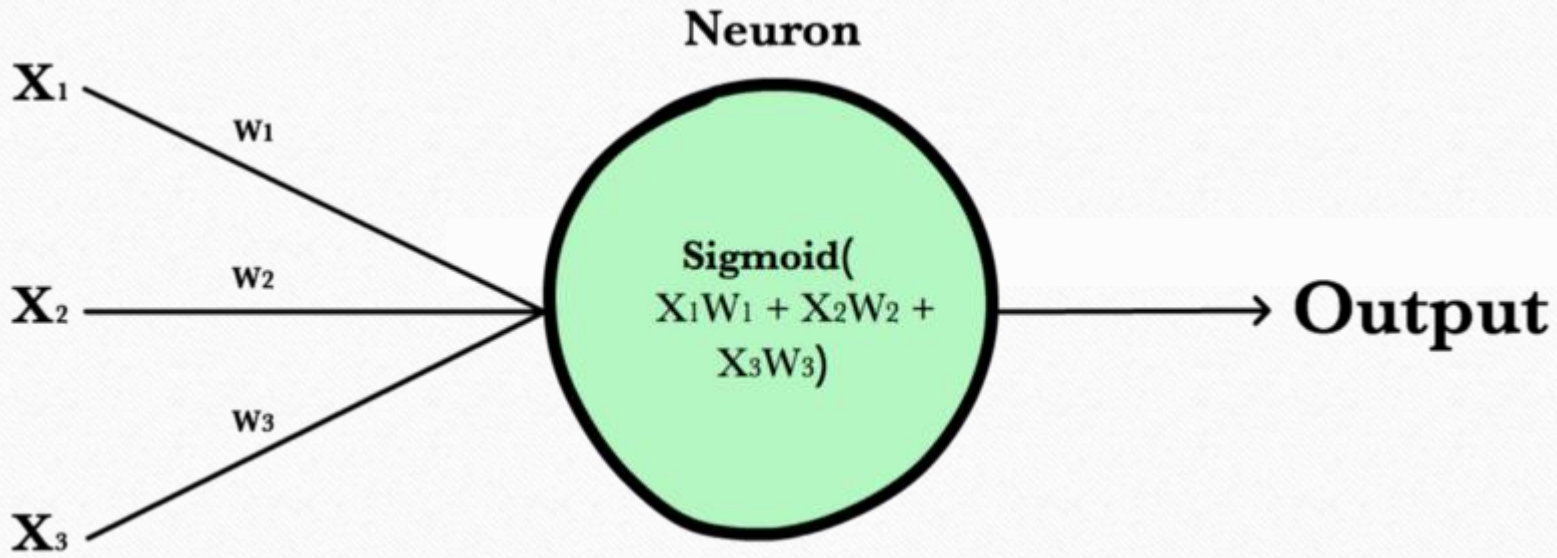


Figure 12.2 Deep network architecture with multiple layers.

Neuron

A neuron takes a group of weighted inputs, applies an activation function, and returns an output.



Inputs to a neuron can either be features from a training set or outputs from a previous layer's neurons. Weights are applied to the inputs as they travel along synapses to reach the neuron. The neuron then applies an activation function to the "sum of weighted inputs" from each incoming synapse and passes the result on to all the neurons in the next layer.

What do the weights in a Neuron convey to us?

1. **Importance of the feature**

Weights associated with each feature, convey the importance of that feature in predicting the output value.

2. **Tells the relationship between a particular feature in the dataset and the target value.**

let us consider an example of finding the likelihood of buying a car, with the dataset containing two input features like

1.Car price

2.Car Popularity

$$Buying\ Car\ \alpha\ \frac{1}{price\ of\ Car} \qquad buying\ car\ \alpha\ Popularity\ of\ Car$$

$$w_1 * (price\ of\ car) + w_2 * (polpularity\ of\ car) + b = 0$$

LAYERS

Think of a layer as a container of neurons. A layer groups a number of neurons together. It is used for holding a collection of neurons.

Input Layer

Holds the data your model will train on. Each neuron in the input layer represents a unique attribute in your dataset (e.g. age, colour)

Hidden Layer

There are often multiple hidden layers in a network. In traditional networks, hidden layers are typically fully-connected layers—each neuron receives input from all the previous layer's neurons and sends its output to every neuron in the next layer.

Output Layer

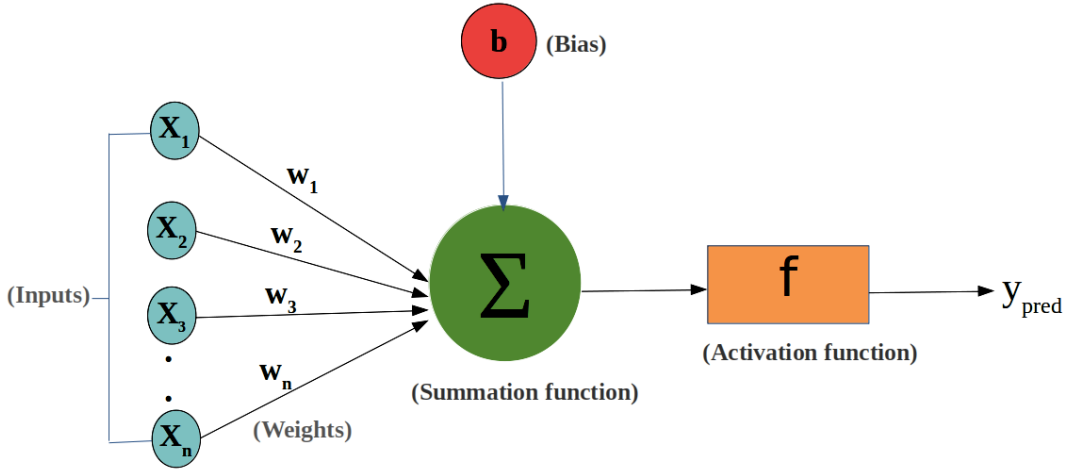
The final layer in a network. It receives input from the previous hidden layer, optionally applies an activation function, and returns an output representing your model's prediction.

SUMMATION FUNCTION

The first function implemented in the output layer is the summation of the products of all the feature data and the associated feature weights for each training data sample in each category/class. This is used as input into the activation function.

Bias allows you to shift the activation function by adding a constant (i.e. the given bias) to the input

$$S_D = \sum \{X_i * W_{i,D}\}$$

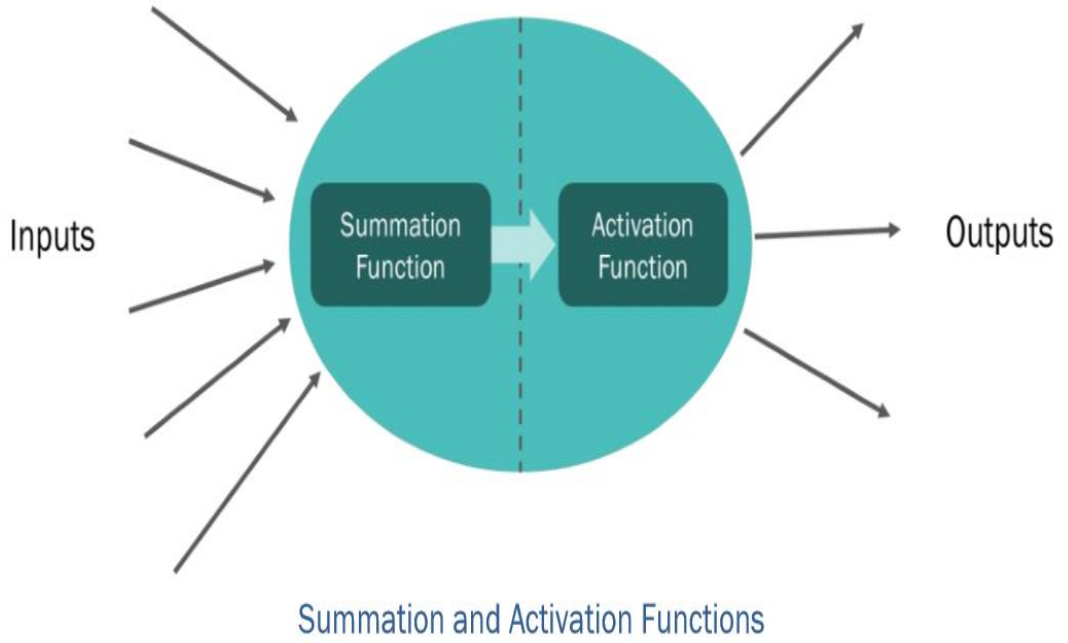


ACTIVATION FUNCTION

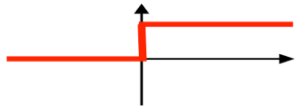
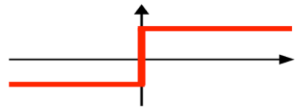
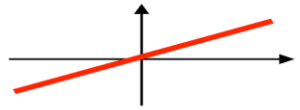
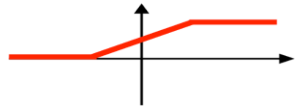
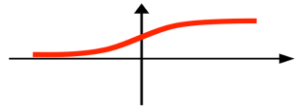
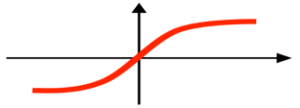
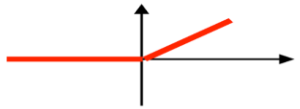
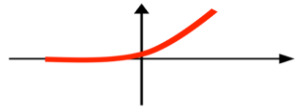
An activation function is a mapping of summed weighted input to the output of the neuron. It is called an activation/ transfer function because it governs the inception at which the neuron is activated and the strength of the output signal.

Mathematically,

$$Y = \Sigma(weight * input) + bias$$

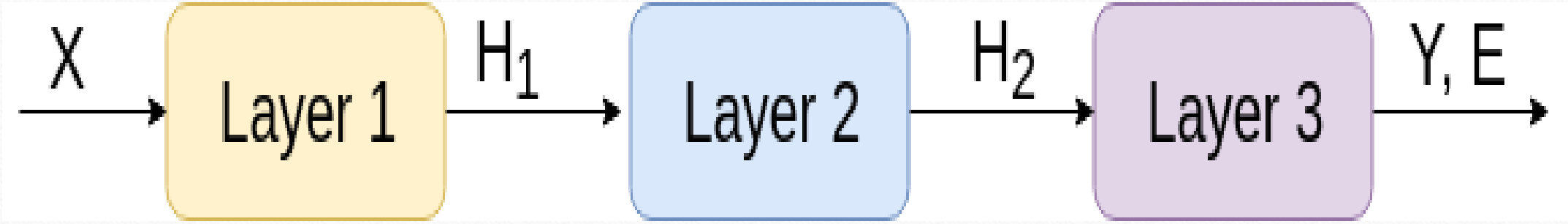


Summation and Activation Functions

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

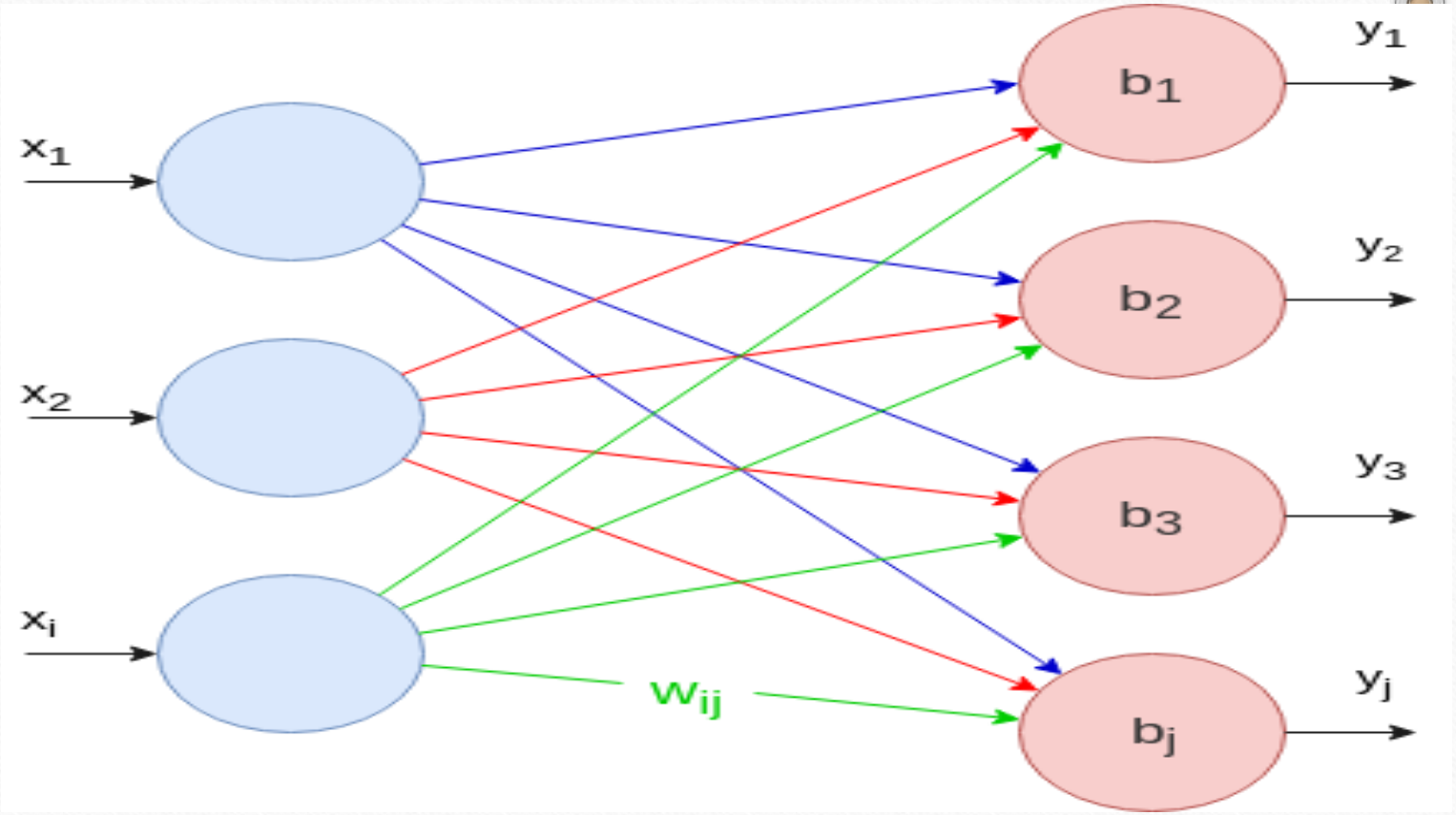
FORWARD PROPAGATION

Forward propagation (or forward pass) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer



Fully Connected Layer

FC layers are the most basic layers as every input neurons are connected to every output neurons.
With matrices, we can compute this formula for every output neuron in one shot using a **dot product** :



$$Y = XW + B$$

$$X = \begin{bmatrix} x_1 & \dots & x_i \end{bmatrix} \quad W = \begin{bmatrix} w_{11} & \dots & w_{1j} \\ \vdots & \ddots & \vdots \\ w_{i1} & \dots & w_{ij} \end{bmatrix} \quad B = \begin{bmatrix} b_1 & \dots & b_j \end{bmatrix}$$