## **What is Deep Learning**

Output

Mapping from features

Complex Features

Output

Output

Mapping from features

Hand Designed program

Handcrafted features

Input

Simple Features

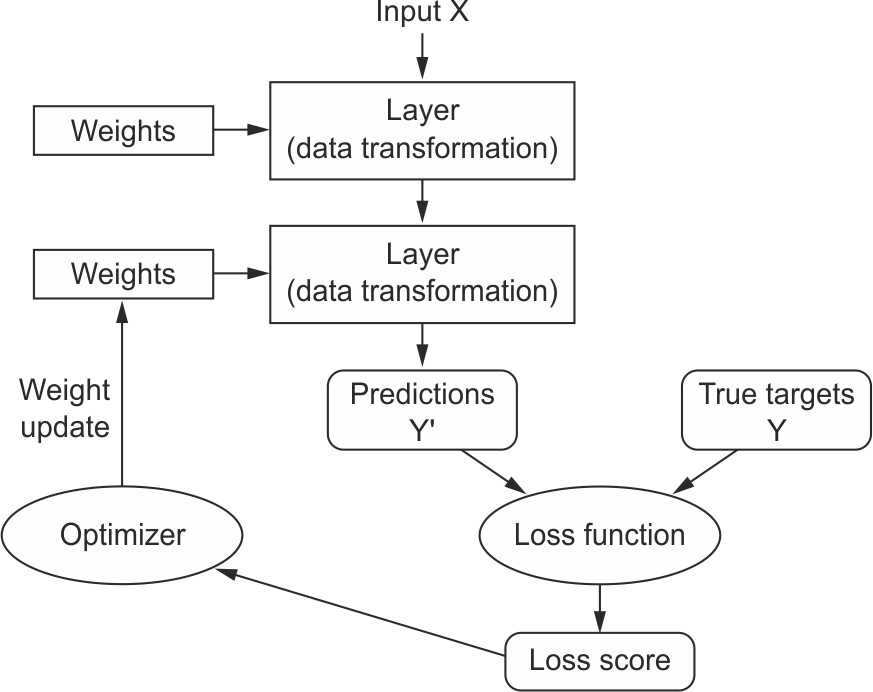
Input

Rule Based

Input

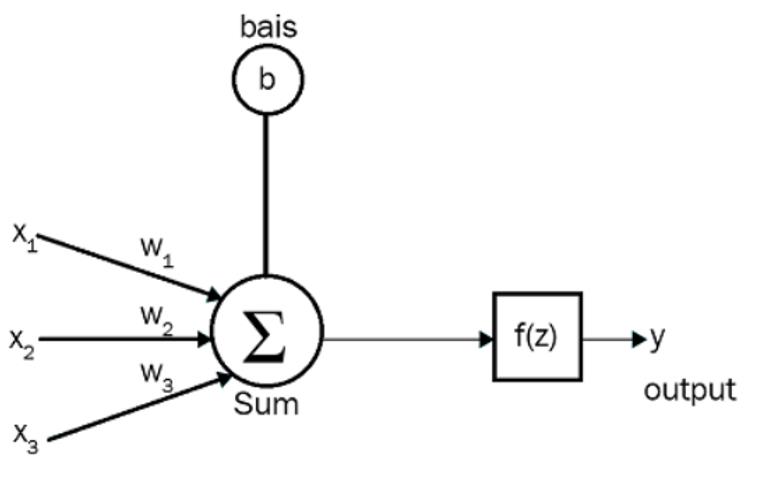
ML

DL



* *Layers*, which are combined into a *network* (or *model*)
* The *input data* and corresponding *targets*
* The *loss function*, which defines the feedback signal used for learning
* The *optimizer*, which determines how learning proceeds

## **Deep Learning – How it works**



**Car Buying Example**

Input- Model/version/Condition- New/Good --- , Miles- 1 lak -- >>> Price less

Output – Price – Y= 5 lasks- Desired out

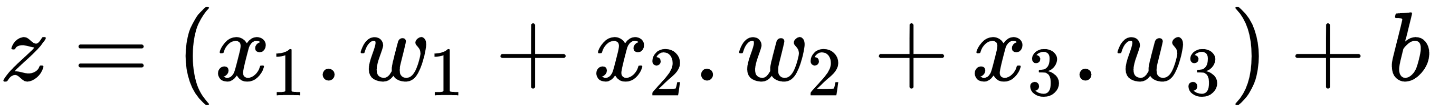
Y`= 2 lakhs – Predited output from model

Z= X Condition \* W1 Weight(Coefficient) + XMiles \* W2

= 0.5 \*0 + 0.8 \* 0 = 0 +1

a = f(Z)

**weights are used for strengthening the inputs**



Input layer- number of neurons in the input layer is the number of inputs we feed to the network.

Hidden Layer- identifies the pattern in the dataset. It is majorly responsible for learning the data representation and for extracting the features.

 we have to choose a number of hidden layers according to our use case. For a very simple problem, we can just use one hidden layer, but while performing complex tasks such as image recognition, we use many hidden layers, where each layer is responsible for extracting important features. The network is called a **deep neural network** when we have many hidden layers.

This transformation is then passed through an activation function, (here I am using **ReLU** or rectified linear units) to make make the output of the linear transformation non-linear. This allows the neural net to model complex non-linear relationships between input and output

**output layer**is the final layer in the model and, in this case, is size ten, one node for each label. We apply a **softmax activation** to this layer so that it outputs values between 0 and 1 across the final layer nodes — representing probabilities across the labels

If it is a multi-class classification say, with five classes, and if we want to get the probability of each class as an output, then the number of neurons in the output layer is five, each emitting the probability. If it is a regression problem, then we have one neuron in the output layer

**Loss- Cost function is a measure of how wrong the model is in terms of its ability to estimate the relationship between X and y.** This is typically expressed as a difference or distance between the predicted value and the actual value. The cost function (you may also see this referred to as loss or error.)

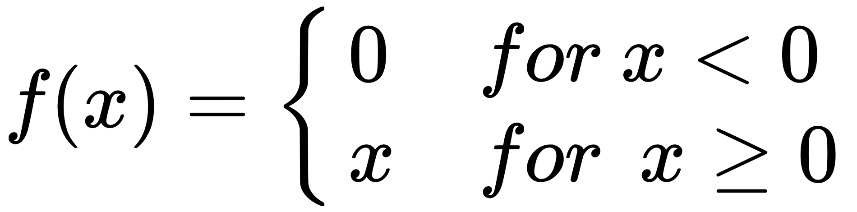
ML model, therefore, is to find parameters, weights or a structure that **minimises**the cost function.

We instantiate the weights with random values with very low variance, and fill the bias variable with zeros. We then define the matrix multiplication that takes place in the layer.

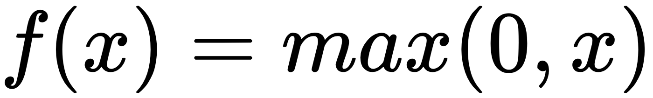
## **Activation Functions**

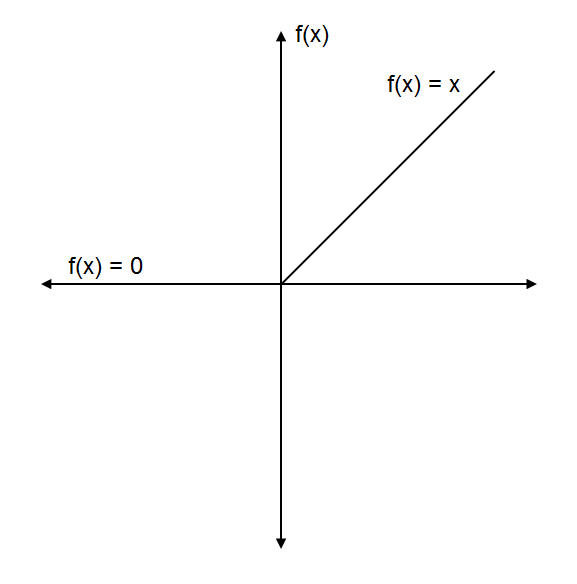
**The Rectified Linear Unit function**

The **Rectified Linear Unit** (**ReLU**) function is another one of the most commonly used activation functions. It outputs a value from o to infinity. It is basically a **piecewise** function and can be expressed as follows:



That is, F(x) returns zero when the value of *x* is less than zero and  f(x)  returns *x* when the value of *x* is greater than or equal to zero. It can also be expressed as follows:

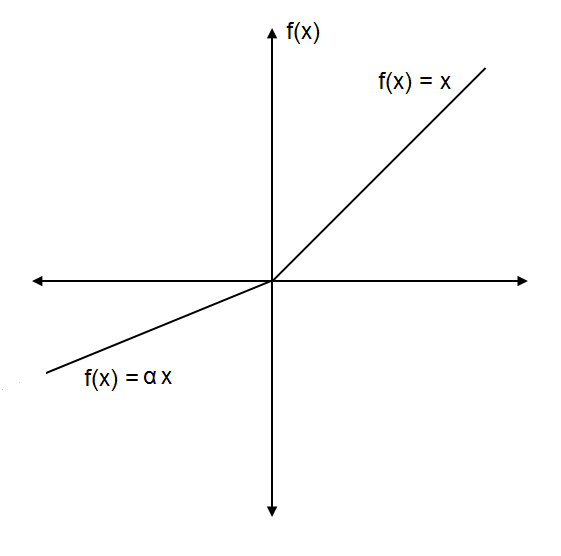


The ReLU function is shown in the following figure:

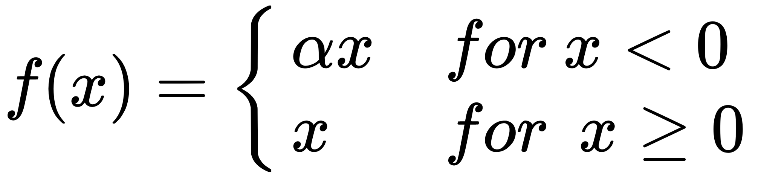
The snag for being zero for all negative values is a problem called **dying ReLU**, and a neuron is said to be dead if it always outputs zero

# The leaky ReLU function

**Leaky ReLU** is a variant of the ReLU function that solves the dying ReLU problem. Instead of converting every negative input to zero, it has a small slope for a negative value as shown:



Leaky ReLU can be expressed as follows:



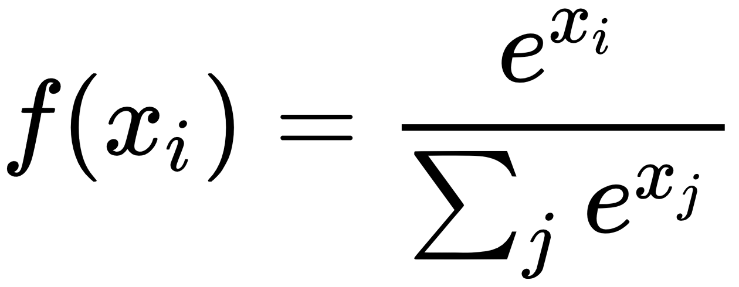
The value of  is typically set to 0.01. The leaky ReLU function is implemented as follows:

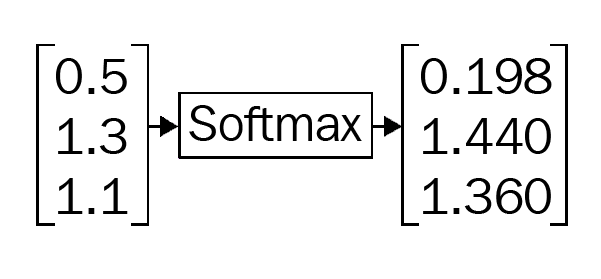
**Parametric ReLU- send parameter to the neural net**

**Randomized Relu- Set some Random value**

**Output layer**

The **softmax function** is basically the generalization of the sigmoid function. It is usually applied to the final layer of the network and while performing multi-class classification tasks. It gives the probabilities of each class for being output and thus, the sum of softmax values will always equal 1





## **Gradient Decent**

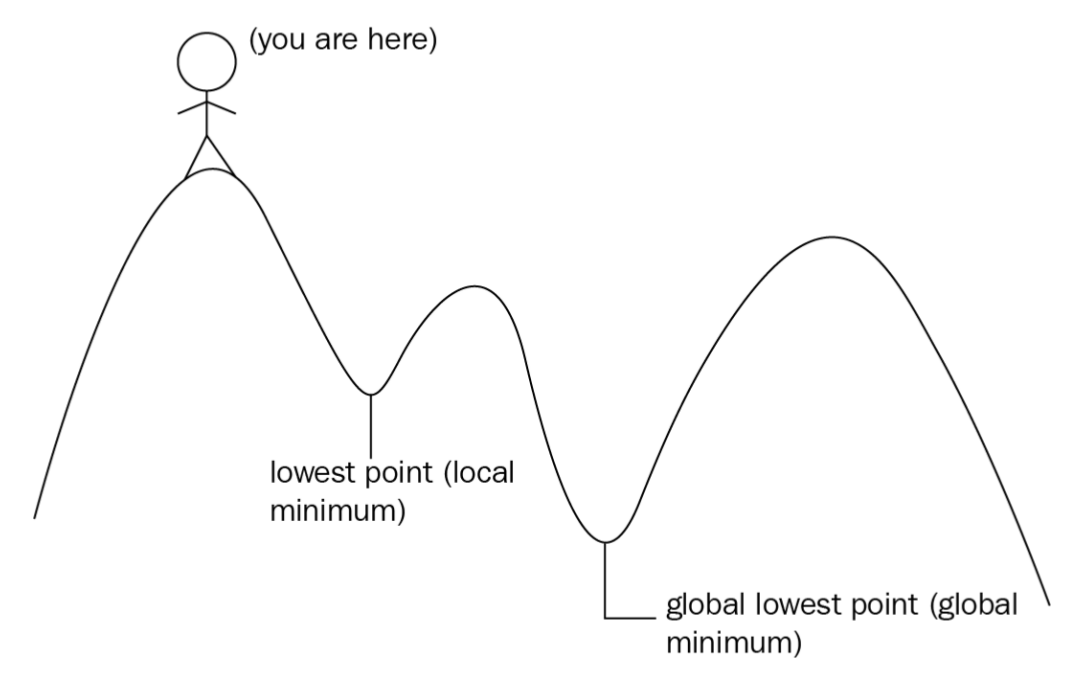
Gradient descent enables a model to learn the gradient or direction that the model should take in order to reduce errors (differences between actual y and predicted y). Direction in the simple linear regression example refers to how the model parameters b0 and b1 should be tweaked or corrected to further reduce the cost function. As the model iterates, it gradually converges towards a minimum where further tweaks to the parameters produce little or zero changes in the loss — also referred to as convergence

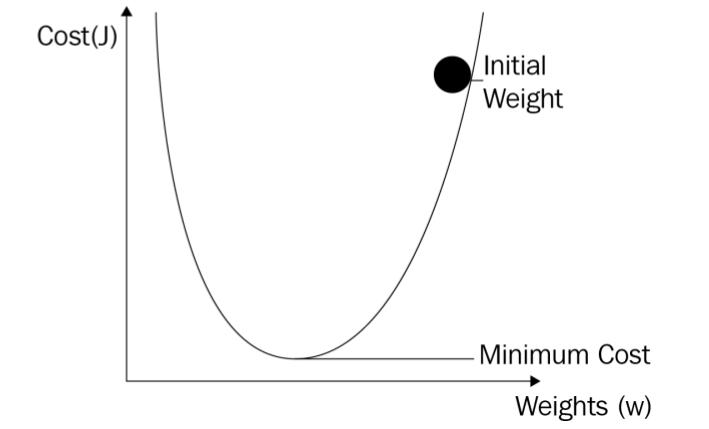
At this point the model has **optimized the weights** such that they minimize the cost function. This process is integral (no calculus pun intended!) to the ML process, because it greatly expedites the learning process — you can think of it as a means of receiving corrective feedback on how to improve upon your previous performance.

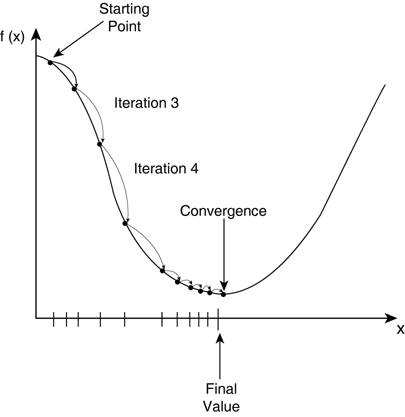
The aim of the activation function is to introduce a non-linear transformation to learn the complex underlying patterns in the data.

With gradient descent, the neural network learns the optimal values of the randomly initialized weight matrices. With the optimal values of weights, our network can predict the correct output and minimize the loss.

 Gradient descent is one of the most commonly used optimization algorithms. It is used for minimizing the cost function, which allows us to minimize the error and obtain the lowest possible error value







## **Neural Network Training Algorithm**

1. Initialize the weights and biases
2. Calculate the network output using forward propagation.
3. Calculate the error between the ground truth and the estimated or predicted output of the network.
4. Update the weights and the biases through backpropagation.
5. Repeat the above three steps until the number of iterations or epochs is reached or the error between the ground truth and the predicted output is below a predefined threshold.