

[Connect, Message, Like, Follow & Share, 100% Free Counselling → Thank You](#)



Neural Networks

A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature. Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria. The concept of neural networks, which has its roots in artificial intelligence, is swiftly gaining popularity in the development of trading systems.

Neural networks are a series of algorithms that mimic the operations of an animal brain to recognize relationships between vast amounts of data.

As such, they tend to resemble the connections of neurons and synapses found in the brain.

They are used in a variety of applications in financial services, from forecasting and marketing research to fraud detection and risk assessment.

Neural networks with several process layers are known as "deep" networks and are used for deep learning algorithms

The success of neural networks for stock market price prediction varies. Neural networks, in the world of finance, assist in the development of such processes as time-series forecasting, algorithmic trading, securities classification, credit risk modeling, and constructing proprietary indicators and price derivatives. A neural network works similarly to the human brain's neural network. A "neuron" in a neural network is a mathematical function that collects and classifies information according to a specific architecture. The network bears a strong resemblance to statistical methods such as curve fitting and regression analysis.

A neural network contains layers of interconnected nodes. Each node is known as a perceptron and is similar to a multiple linear regression. The perceptron feeds the signal produced by a multiple linear regression into an activation function that may be nonlinear.

Multi-Layered Perceptron

In a multi-layered perceptron (MLP), perceptrons are arranged in interconnected layers. The input layer collects input patterns. The output layer has classifications or output signals to which input patterns may map. For instance, the patterns may comprise a list of quantities for technical indicators about a security; potential outputs could be “buy,” “hold” or “sell.” Hidden layers fine-tune the input weightings until the neural network’s margin of error is minimal. It is hypothesized that hidden layers extrapolate salient features in the input data that have predictive power regarding the outputs. This describes feature extraction, which accomplishes a utility similar to statistical techniques such as principal component analysis.

Application of Neural Networks

Neural networks are broadly used, with applications for financial operations, enterprise planning, trading, business analytics, and product maintenance. Neural networks have also gained widespread adoption in business applications such as forecasting and marketing research solutions, fraud detection, and risk assessment. A neural network evaluates price data and unearths opportunities for making trade decisions based on the data analysis. The networks can distinguish subtle nonlinear interdependencies and patterns other methods of technical analysis cannot. According to research, the accuracy of neural networks in making price predictions for stocks differs. Some models predict the correct stock prices 50 to 60 percent of the time, while others are accurate in 70 percent of all instances. Some have posited that a 10 percent improvement in efficiency is all an investor can ask for from a neural network.

Components of a Neural Network

There are three main components: an input layer, a processing layer, and an output layer. The inputs may be weighted based on various criteria. Within the processing layer, which is hidden from view, there are nodes and connections between these nodes, meant to be analogous to the neurons and synapses in an animal brain.

Convolutional Neural Network

A convolutional neural network is one adapted for analyzing and identifying visual data such as digital images or photographs.

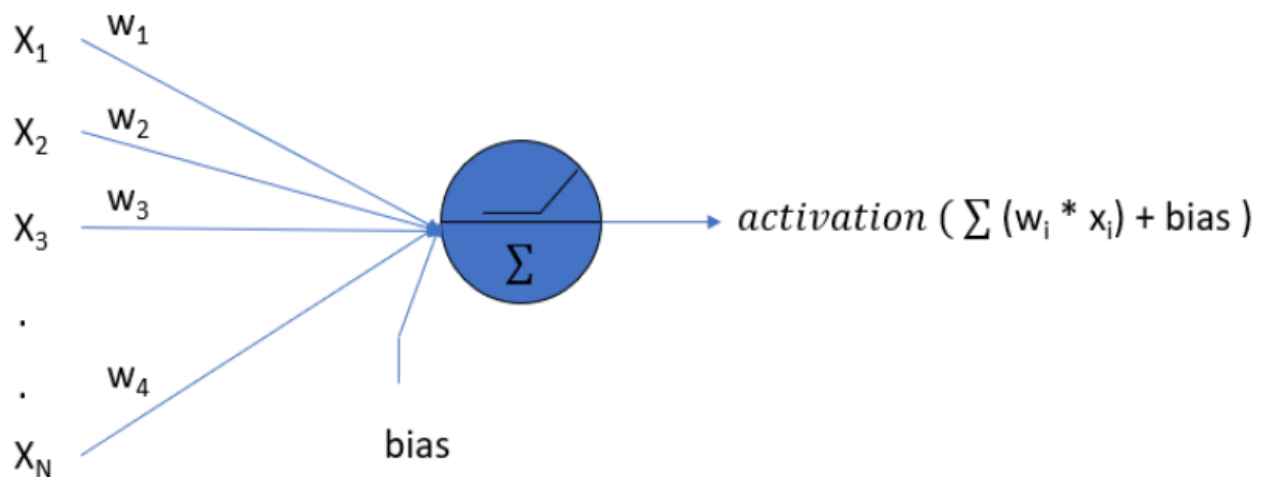
Recurrent Neural Network

A recurrent neural network is one adapted for analyzing time series data, event history, or temporal ordering.

Deep Neural Network

Also known as a deep learning network, a deep neural network, at its most basic, is one that involves two or more processing layers.

Neural networks are used to mimic the basic functioning of the human brain and are inspired by how the human brain interprets information. It is used to solve various real-time tasks because of its ability to perform computations quickly and its fast responses.



A single neuron shown with x_i inputs with their respective weights w_i and a bias term and applied activation function

An Artificial Neural Network model contains various components that are inspired by the biological nervous system.

Artificial Neural Network has a huge number of interconnected processing elements, also known as Nodes. These nodes are connected with other nodes using a connection link. The connection link contains weights, these weights contain the information about the input signal. Each iteration and input in turn leads to updation of these weights. After inputting all the data instances from the training data set, the final weights of the Neural Network along with its architecture is known as the Trained Neural Network. This process is called Training of Neural Networks. This trained neural network is used to solve specific problems as defined in the problem statement. Types of tasks that can be solved using an artificial neural network include Classification problems, Pattern Matching, Data Clustering, etc.

We use artificial neural networks because they learn very efficiently and adaptively. They have the capability to learn “how” to solve a specific problem from the training data it receives. After learning, it can be used to solve that specific problem very quickly and efficiently with high accuracy.

Some real-life applications of neural networks include Air Traffic Control, Optical Character Recognition as used by some scanning apps like Google Lens, Voice Recognition, etc.

Types of Neural Networks

(i) ANN– It is also known as an artificial neural network. It is a feed-forward neural network because the inputs are sent in the forward direction. It can also contain hidden layers which can make the model even denser. They have a fixed length as specified by the programmer. It is used for Textual Data or Tabular Data. A widely used real-life application is Facial Recognition. It is comparatively less powerful than CNN and RNN.

(ii) CNN– It is also known as Convolutional Neural Networks. It is mainly used for Image Data. It is used for Computer Vision. Some of the real-life applications are object detection in autonomous vehicles. It contains a combination of convolutional layers and neurons. It is more powerful than both ANN and RNN.

(iii) RNN–

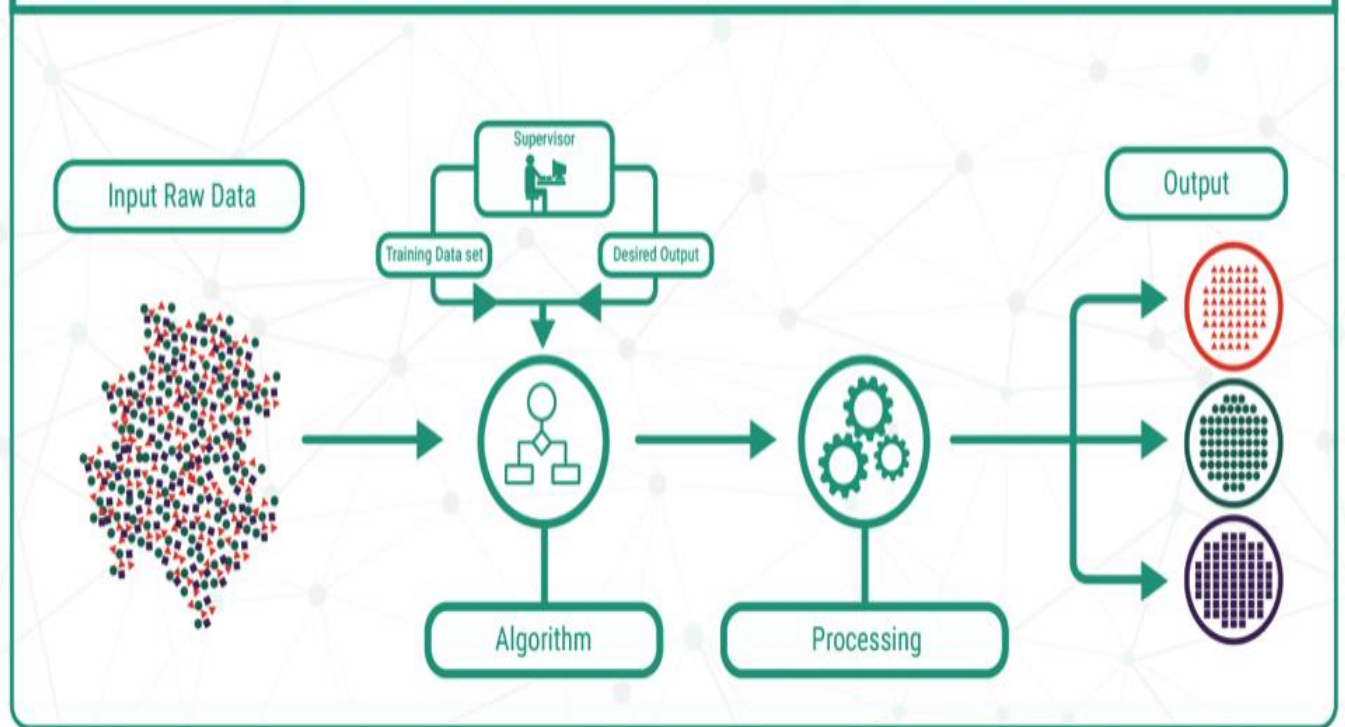
It is also known as Recurrent Neural Networks. It is used to process and interpret time series data. In this type of model, the output from a processing node is fed back into nodes in the same or previous layers. The most known types of RNN are **LSTM** (Long Short Term Memory) Networks

Now that we know the basics about Neural Networks, We know that Neural Networks' learning capability is what makes it interesting. There are 3 types of **learnings in Neural networks**, namely

1. **Supervised Learning**
2. **Unsupervised Learning**
3. **Reinforcement Learning**

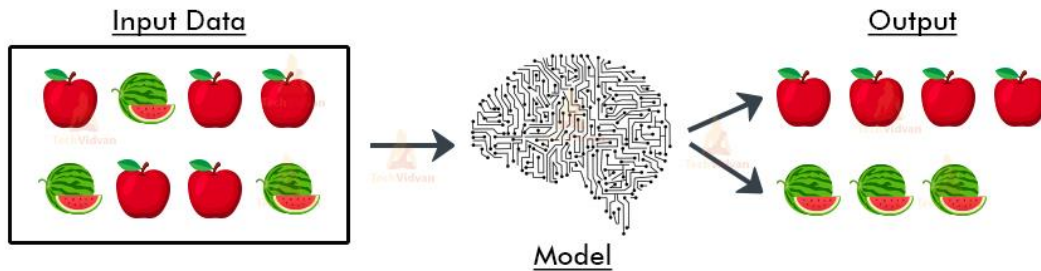
Supervised Learning: As the name suggests, it is a type of learning that is looked after by a supervisor. It is like learning with a teacher. There are input training pairs that contain a set of input and the desired output. Here the output from the model is compared with the desired output and an error is calculated, this error signal is sent back into the network for adjusting the weights. This adjustment is done till no more adjustments can be made and the output of the model matches the desired output. In this, there is feedback from the environment to the model.

SUPERVISED LEARNING

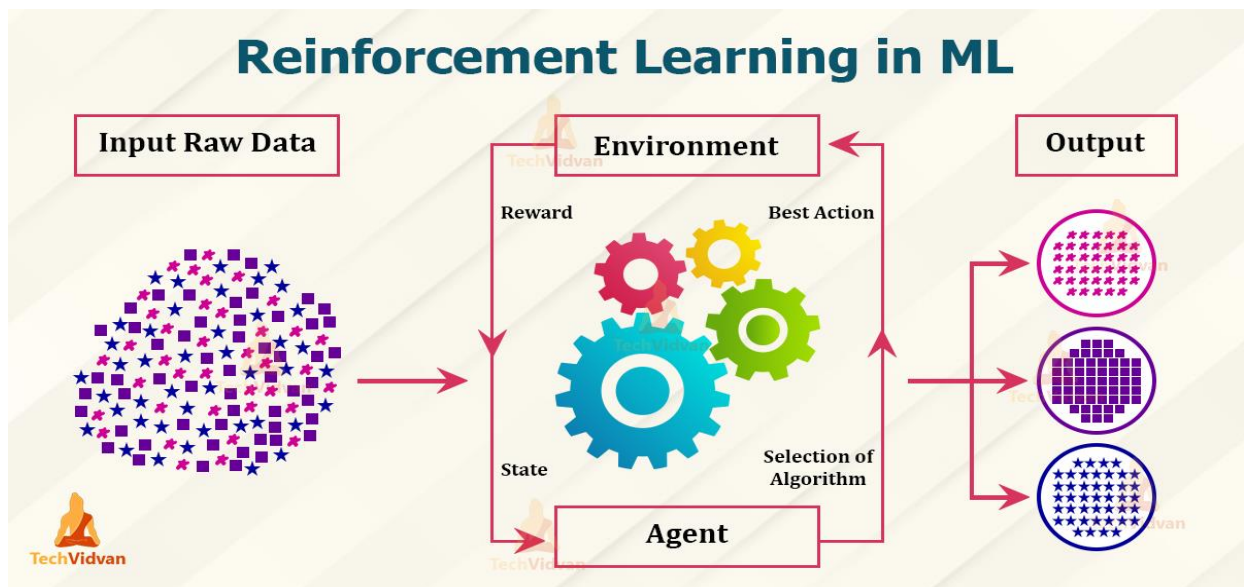


Unsupervised Learning: Unlike supervised learning, there is no supervisor or a teacher here. In this type of learning, there is no feedback from the environment, there is no desired output and the model learns on its own. During the training phase, the inputs are formed into classes that define the similarity of the members. Each class contains similar input patterns. On inputting a new pattern, it can predict to which class that input belongs based on similarity with other patterns. If there is no such class, a new class is formed.

Unsupervised Learning in ML

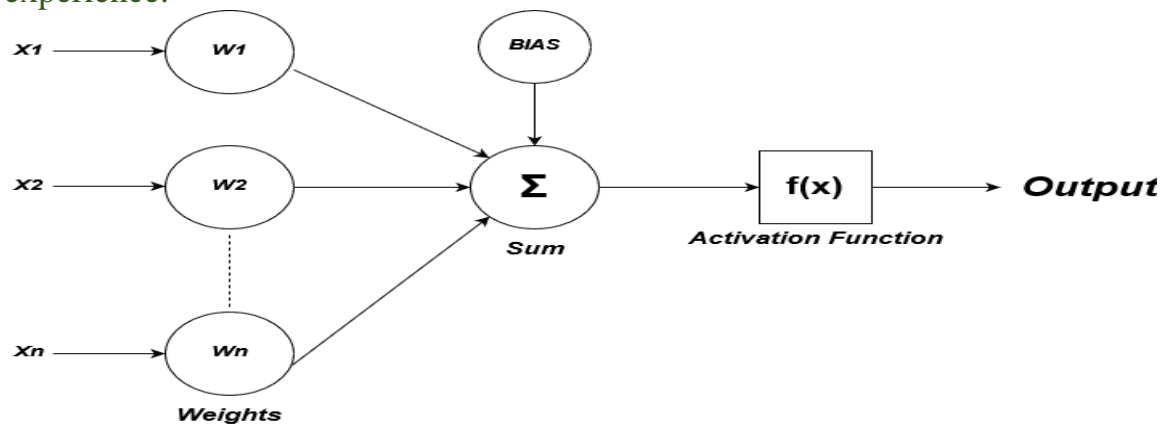


Reinforcement Learning: It gets the best of both worlds, that is, the best of both Supervised learning and Unsupervised learning. It is like learning with a critique. Here there is no exact feedback from the environment, rather there is critique feedback. The critique tells how close our solution is. Hence the model learns on its own based on the critique information. It is similar to supervised learning in that it receives feedback from the environment, but it is different in that it does not receive the desired output information, rather it receives critique information.

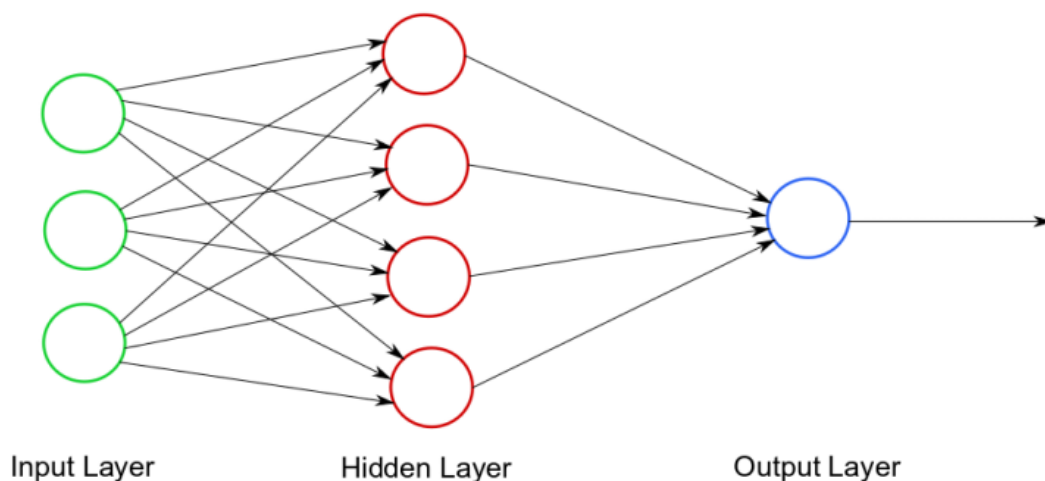


According to Arthur Samuel, one of the early American pioneers in the field of computer gaming and artificial intelligence, he defined machine learning as: Suppose we arrange for some automatic means of testing the effectiveness of any current weight assignment in terms of actual performance and provide a

mechanism for altering the weight assignment so as to maximize the performance. We need not go into the details of such a procedure to see that it could be made entirely automatic and to see that a machine so programmed would “learn” from its experience.



An artificial neuron can be thought of as a simple or multiple linear regression model with an activation function at the end. A neuron from layer i will take the output of all the neurons from the later $i-1$ as inputs calculate the weighted sum and add bias to it. After this is sent to an activation function as we saw in the previous diagram.



The first neuron from the first layer is connected to all the inputs from the previous layer, Similarly, the second neuron from the first hidden layer will also be connected to all the inputs from the previous layer and so on for all the neurons in

the first hidden layer. For neurons in the second hidden layer (outputs of the previously hidden layer) are considered as inputs and each of these neurons are connected to previous neurons, likewise. This whole process is called **Forward propagation**.

After this, there is an interesting thing that happens. Once we have predicted the output it is then compared to the actual output. We then calculate the loss and try to minimize it. But how can we minimize this loss? For this, there comes another concept which is known as Back Propagation. We will understand more about this in another article. I will tell you how it works. First, the loss is calculated then weights and biases are adjusted in such a way that they try to minimize the loss. Weights and biases are updated with the help of another algorithm called gradient descent. We will understand more about gradient descent in a later section. We basically move in the direction opposite to the gradient. This concept is derived from the Taylor series.

A Neural Network is analogous to the connections of neurons in our brain. In this article, we will see how to set up Neural Networks, Artificial Neural Networks, and Deep Neural Networks, and also how to design the model, how to train it, and finally how to use it for testing. Now we will see how to implement these using Python.

Before going into that, let us talk about Deep Learning and Neural Networks.

Deep Learning:

Deep Learning is a type of machine learning which is very similar to how humans think and gain knowledge about things. Previously invented algorithms are linear, and deep learning is complex.

Neural Networks:

A Neural Network is an assortment of some unit cells, that are interconnected and transmit signals from one to another cell, and they can learn and make decisions.

Deep Learning is very complex and needs to be simplified and broken into simple algorithms, and this can be made possible through a process called visualization.

Setting Neural Networks:

There are many libraries to set up Neural Networks, in this article we will use TensorFlow, which was developed by Google.

Install Tensorflow

```
pip install tensorflow
```

The next step is to import the notebook and main modules from TensorFlow Keras.

The code is given below.

```
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import utils
from tensorflow.keras import backend as K
import matplotlib.pyplot as plt
import shap
```

Artificial Neural Networks

An ANN has an input and an output and is made up of some layers. Neurons, or nodes, are units that connect inputs through a function. This activated function makes neurons switch off or switch on. The weighted inputs have weights, which are randomly initialized, but during training, they will get optimized like all the machine learning algorithms.

As we discussed before, ANNs are made of layers, which in turn are divided into 3 parts.

1-Input Layer

2-Hidden Layer

3-Output Layer

The explanation is similar to their names.

Input Layer- It transfers the input to the Neural Network.

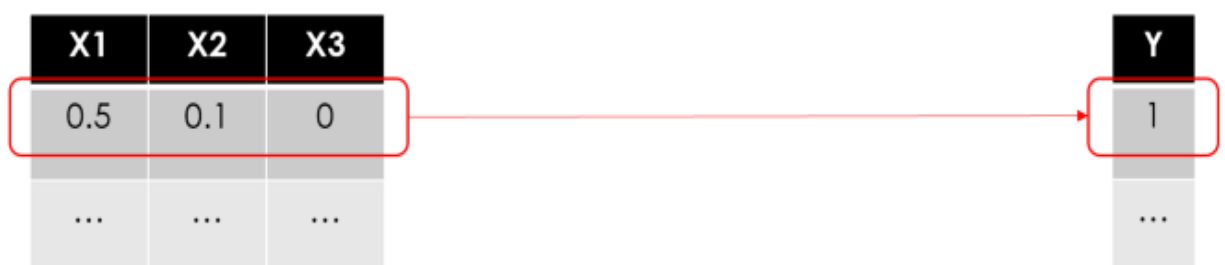
Hidden Layer- It is like an intermediate junction between the input layer and output layer.

Output Layer- It transfers the output to the Neural Network.

ANNs may have multiple layers with a minimum of 1 layer. ANN with 1 layer is called a perceptron. A perceptron behaves like a simple linear regression model.

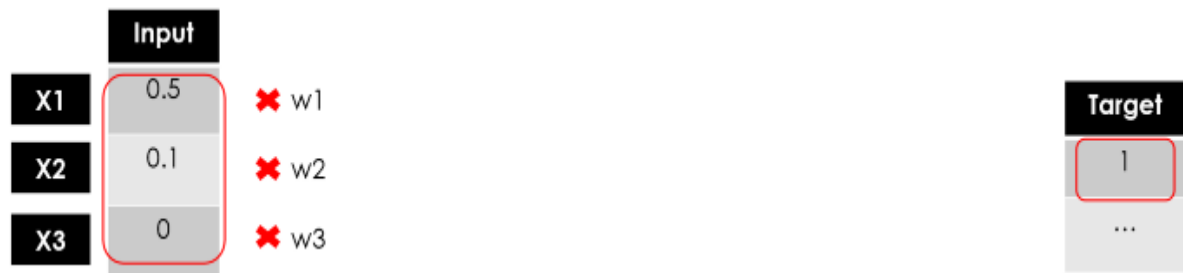
Let us go through some examples.

Let us take an example of a dataset having 3 inputs and one output.

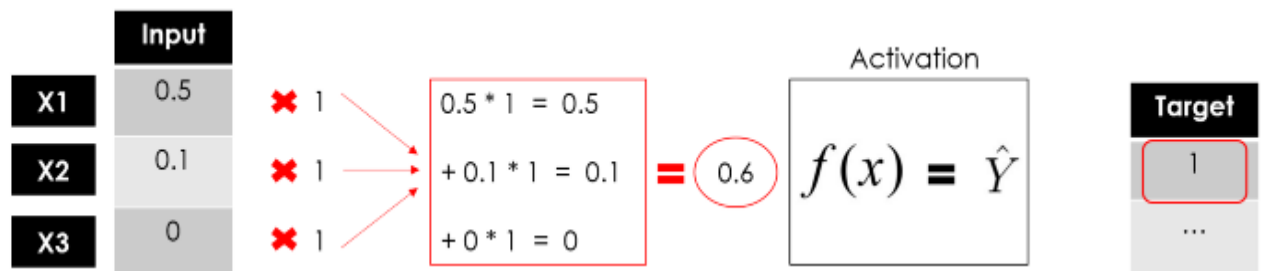


The common thing we need to do in machine learning models is training.

Training is finding the best parameters so that error is minimized.



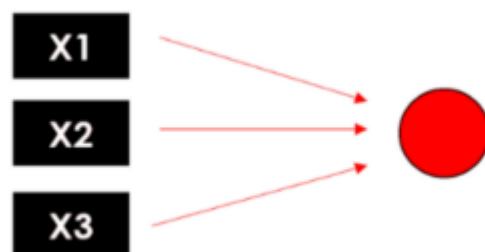
w_1 , w_2 , and w_3 are weights that can be initialized randomly. Let us initialize with 1.



Here we used a linear function to determine the output, so it is very much like linear regression.

As we saw, it is a perceptron. It has 3 inputs and one output. In this training, we will compare the output each time with the predicted one, optimize the parameters, or change the function so that it will be close to the expected value.

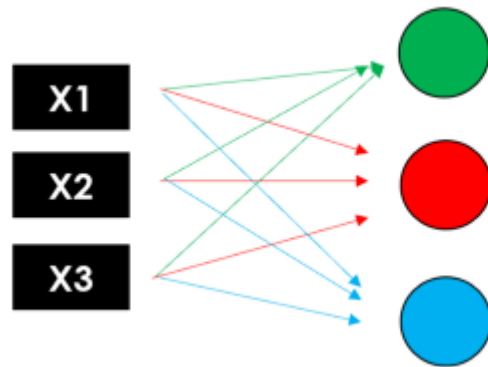
A Neuron is represented as,



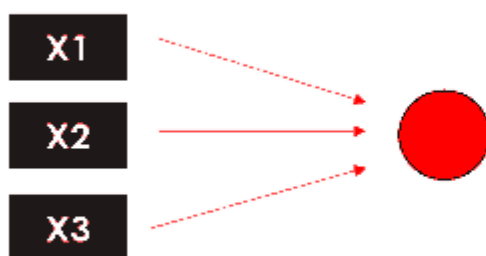
Deep Neural Networks

A neural network is said to be deep when it has at least 2 hidden layers so that it has 1 input layer, at least 2 hidden layers, and 1 output layer.

In a DNN, the process happens three times simultaneously and sends the output to the first hidden layer, which in turn becomes the input for the second hidden layer.



The second hidden layer takes the input from the first hidden layer and sends the output to the output layer.



Model Design

Now we will build a neural network with TensorFlow.

```

model = models.Sequential(name="Perceptron", layers=[ layers.Dense(
    name="dense",
    input_dim=3,
    units=1,
    activation='linear' #f(x)=x
)
])
model.summary()

```

Model: "Perceptron"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 1)	4
Total params: 4		
Trainable params: 4		
Non-trainable params: 0		

Source: https://miro.medium.com/max/700/1*nke8IcOb3b8xPwBq7tCDcw.png

If we want to use a binary function instead of the linear function,

define the function

```
import tensorflow as tf
```

```
def binary_step_activation(x):
```

```
    ##return 1 if x>0 else 0
```

```
    return K.switch(x>0, tf.math.divide(x,x), tf.math.multiply(x,0))
```

build the model

```

model = models.Sequential(name="Perceptron", layers=[
    layers.Dense(
        name="dense",
        input_dim=3,
        units=1,
        activation=binary_step_activation
    )
])

```

)
D)



We have discussed perceptron models. Let us go to the DNNs.

We need to know the following parameters before building a deep neural network:

1. Number of Layers
2. Number of neurons
3. Activation function

Number of Neurons = (Number of inputs + 1 output)/2

The activation function depends on the problem. For example, linear functions are used for regression models, and sigmoid functions are used for classification models.

Rectified linear unit (ReLU) ^[9]		$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ $= \max\{0, x\} = x \mathbf{1}_{x>0}$
Logistic, sigmoid, or soft step		$\sigma(x) = \frac{1}{1 + e^{-x}}$

Let us take an example with N inputs and one output.

```
n_features = 10
model = models.Sequential(name="DeepNN", layers=[
    ### hidden layer 1
    layers.Dense(name="h1", input_dim=n_features,
                  units=int(round((n_features+1)/2)),
                  activation='relu'),
    layers.Dropout(name="drop1", rate=0.2),
```



```

### hidden layer 2
layers.Dense(name="h2", units=int(round((n_features+1)/4)),
             activation='relu'),
layers.Dropout(name="drop2", rate=0.2),

### layer output
layers.Dense(name="output", units=1, activation='sigmoid')
)
model.summary()

```

Model: "DeepNN"

Layer (type)	Output Shape	Param #
h1 (Dense)	(None, 6)	66
drop1 (Dropout)	(None, 6)	0
h2 (Dense)	(None, 3)	21
drop2 (Dropout)	(None, 3)	0
output (Dense)	(None, 1)	4
Total params: 91		
Trainable params: 91		
Non-trainable params: 0		

Now we are going to use model class to build perceptron.

```

# Perceptron
inputs = layers.Input(name="input", shape=(3,))
outputs = layers.Dense(name="output", units=1,
                       activation='linear')(inputs)
model = models.Model(inputs=inputs, outputs=outputs,
                     name="Perceptron")

```

Next is using the model class to build a Deep Neural Network.

```

# DeepNN

#### layer input

inputs = layers.Input(name="input", shape=(n_features,))#### hidden layer 1
h1 = layers.Dense(name="h1", units=int(round((n_features+1)/2)),
activation='relu')(inputs)

h1 = layers.Dropout(name="drop1", rate=0.2)(h1)#### hidden layer 2
h2 = layers.Dense(name="h2", units=int(round((n_features+1)/4)),
activation='relu')(h1)

h2 = layers.Dropout(name="drop2", rate=0.2)(h2)#### layer output
output = layers.Dense(name="output", activation='sigmoid')(h2)
model = models.Model(inputs=inputs, outputs=output)

```

Visualization

The function for plotting an ANN using the TensorFlow model is given below.

```

def utils_nn_config(model):
    lst_layers = []
    if "Sequential" in str(model): #-> Sequential doesn't show the input layer
        layer = model.layers[0]
        lst_layers.append({"name":"input", "in":int(layer.input.shape[-1]), "neurons":0,
                           "out":int(layer.input.shape[-1]), "activation":None,
                           "params":0, "bias":0})
    for layer in model.layers:
        try:
            dic_layer = {"name":layer.name, "in":int(layer.input.shape[-1]),
"neurons":layer.units,
                           "out":int(layer.output.shape[-1]),
"activation":layer.get_config()["activation"],
                           "params":layer.get_weights()[0], "bias":layer.get_weights()[1]}
        except:
            dic_layer = {"name":layer.name, "in":int(layer.input.shape[-1]), "neurons":0,
                           "out":int(layer.output.shape[-1]), "activation":None,
                           "params":0, "bias":0}

```

```
    lst_layers.append(dic_layer)
return lst_layers
```

```
def visualize_nn(model, description=False, figsize=(10,8)):
```

```
    lst_layers = utils_nn_config(model)
    layer_sizes = [layer["out"] for layer in lst_layers]

    fig = plt.figure(figsize=figsize)
    ax = fig.gca()
    ax.set(title=model.name)
    ax.axis('off')
    left, right, bottom, top = 0.1, 0.9, 0.1, 0.9
    x_space = (right-left) / float(len(layer_sizes)-1)
    y_space = (top-bottom) / float(max(layer_sizes))
    p = 0.025
    ## nodes
    for i,n in enumerate(layer_sizes):
        top_on_layer = y_space*(n-1)/2.0 + (top+bottom)/2.0
        layer = lst_layers[i]
        colour = "green" if i in [0, len(layer_sizes)-1] else "blue"
        colour = "red" if (layer['neurons'] == 0) and (i > 0) else color
        ### add description
        if (description is True):
            d = i if i == 0 else i-0.5
            if layer['activation'] is None:
                plt.text(x=left+d*x_space, y=top, fontsize=10, color=colour,
s=layer["name"].upper())
            else:
                plt.text(x=left+d*x_space, y=top, fontsize=10, color=colour,
s=layer["name"].upper())
```

```

plt.text(x=left+d*x_space, y=top-p, fontsize=10, color=color,
s=layer['activation']+" (")

plt.text(x=left+d*x_space, y=top-2*p, fontsize=10, color=color,
s="Σ"+str(layer['in'])+"[X*w]+b")

out = " Y" if i == len(layer_sizes)-1 else " out"

plt.text(x=left+d*x_space, y=top-3*p, fontsize=10, color=color, s=") =
"+str(layer['neurons'])+out)

### circles
for m in range(n):

    color = "limegreen" if color == "green" else color

    circle = plt.Circle(xy=(left+i*x_space, top_on_layer-m*y_space-4*p),
radius=y_space/4.0, color=color, ec='k', zorder=4)

    ax.add_artist(circle)

    ### add text

    if i == 0:

        plt.text(x=left-4*p, y=top_on_layer-m*y_space-4*p, fontsize=10,
s=r'$X_{'+str(m+1)+'}$')

        elif i == len(layer_sizes)-1:

            plt.text(x=right+4*p, y=top_on_layer-m*y_space-4*p, fontsize=10,
s=r'$y_{'+str(m+1)+'}$')

        else:

            plt.text(x=left+i*x_space+p, y=top_on_layer-
m*y_space+(y_space/8.+0.01*y_space)-4*p, fontsize=10, s=r'$H_{'+str(m+1)+'}$')

    ## links

    for i, (n_a, n_b) in enumerate(zip(layer_sizes[:-1], layer_sizes[1:])):

        layer = lst_layers[i+1]

        color = "green" if i == len(layer_sizes)-2 else "blue"

        color = "red" if layer['neurons'] == 0 else color

        layer_top_a = y_space*(n_a-1)/2. + (top+bottom)/2. -4*p
        layer_top_b = y_space*(n_b-1)/2. + (top+bottom)/2. -4*p

        for m in range(n_a):

            for o in range(n_b):

                line = plt.Line2D([i*x_space+left, (i+1)*x_space+left],

```

```

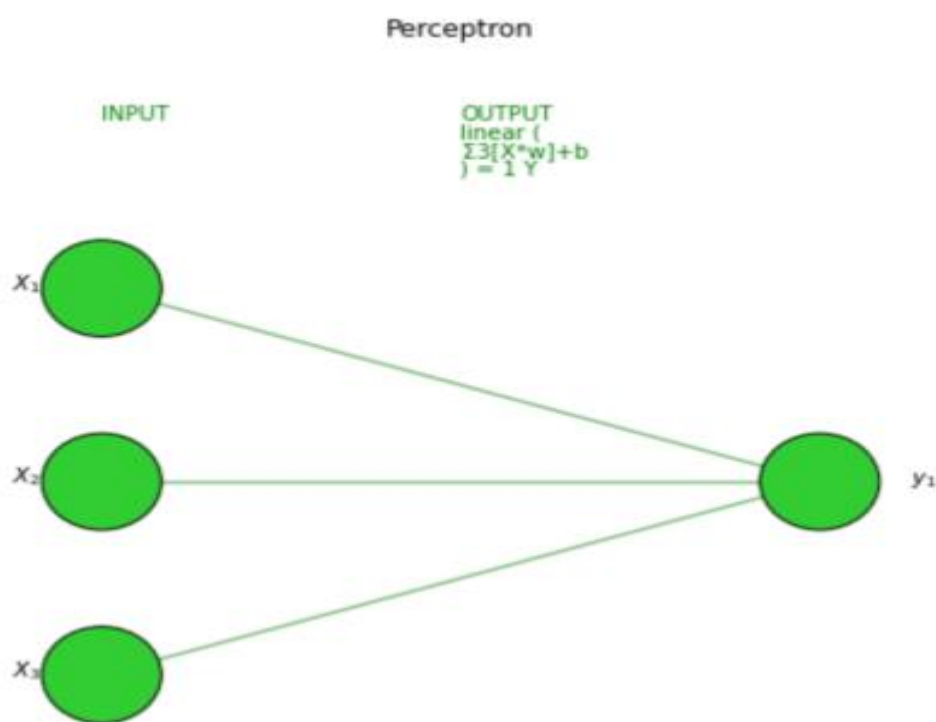
        [layer_top_a-m*y_space, layer_top_b-o*y_space],
        c=color, alpha=0.5)
    if layer['activation'] is None:
        if o == m:
            ax.add_artist(line)
        else:
            ax.add_artist(line)
plt.show()

visualize_nn(model, description=True, figsize=(10,8))

```

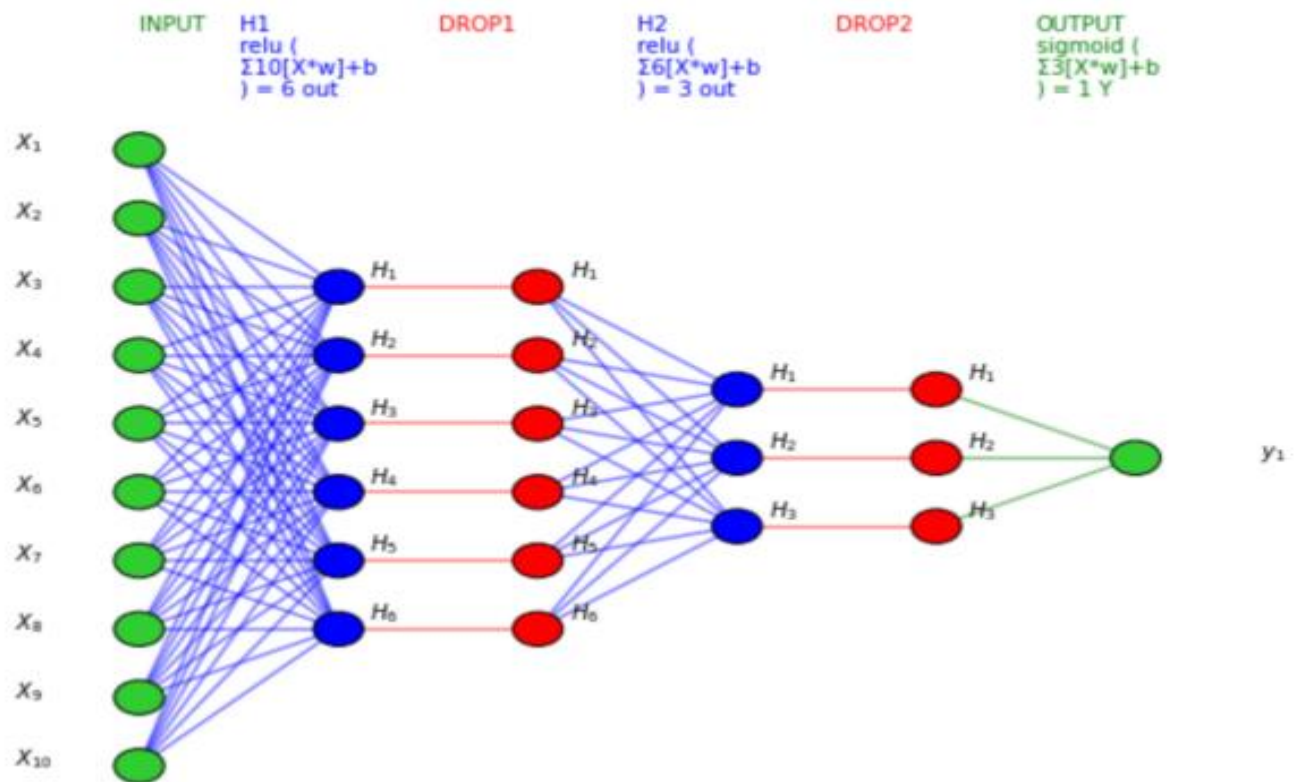
We are going to visualize our perceptron model and DNN model.

First, we will see the perceptron model.

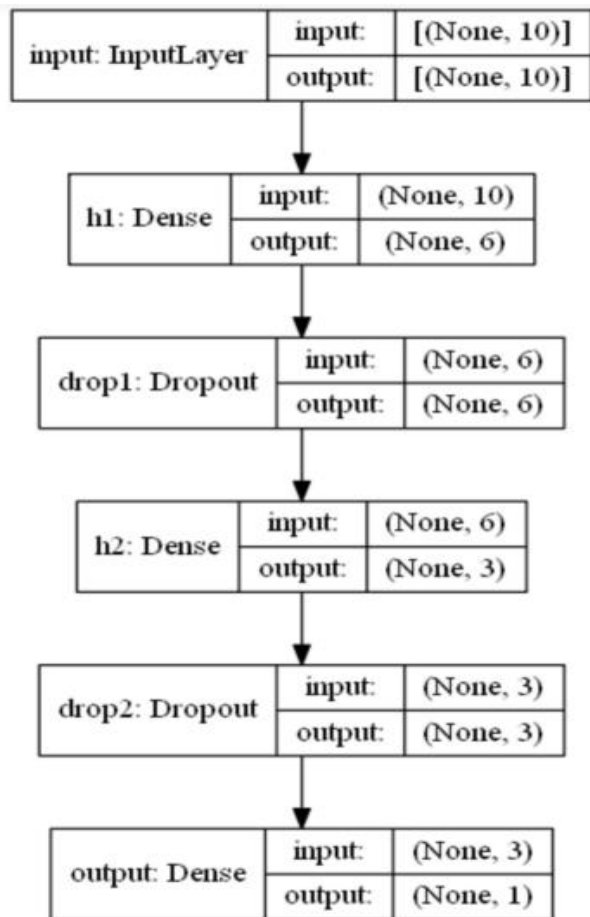


The below image is for a Deep Neural Network.

DeepNN



```
utils.plot_model(model, to_file='model.png', show_shapes=True,  
show_layer_names=True)
```



Training and Testing

We need an optimizer, a loss function, and metrics. We can use Adam Optimizer.

define metrics

```
def Recall(y_true, y_pred):
```

```
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
```

```
    possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
```

```
    recall = true_positives / (possible_positives + K.epsilon())
```

```
    return recall
```

```
def Precision(y_true, y_pred):
```

```
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
```

```
    predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
```

```
    precision = true_positives / (predicted_positives + K.epsilon())
```



```
return precision
```

```
def F1(y_true, y_pred):  
    precision = Precision(y_true, y_pred)  
    recall = Recall(y_true, y_pred)  
    return 2*((precision*recall)/(precision+recall+K.epsilon()))
```

```
model.compile(optimizer='adam', loss='binary_crossentropy',  
              metrics=['accuracy',F1])
```

```
# define metrics
```

```
def R2(y, y_hat):  
    ss_res = K.sum(K.square(y - y_hat))  
    ss_tot = K.sum(K.square(y - K.mean(y)))  
    return ( 1 - ss_res/(ss_tot + K.epsilon()) )
```

```
model.compile(optimizer='adam', loss='mean_absolute_error',  
              metrics=[R2])
```

During training, we can find that accuracy increases and loss decreases. We will keep some data sets for testing.

If we don't have our data ready, we can generate random data using the following code.

```
import numpy as np  
X = np.random.rand(1000,10)  
y = np.random.choice([1,0], size=1000)  
# train/validation  
training = model.fit(x=X, y=y, batch_size=30, epochs=100, shuffle=True)  
  
## training  
ax[0].set(title="Training")
```

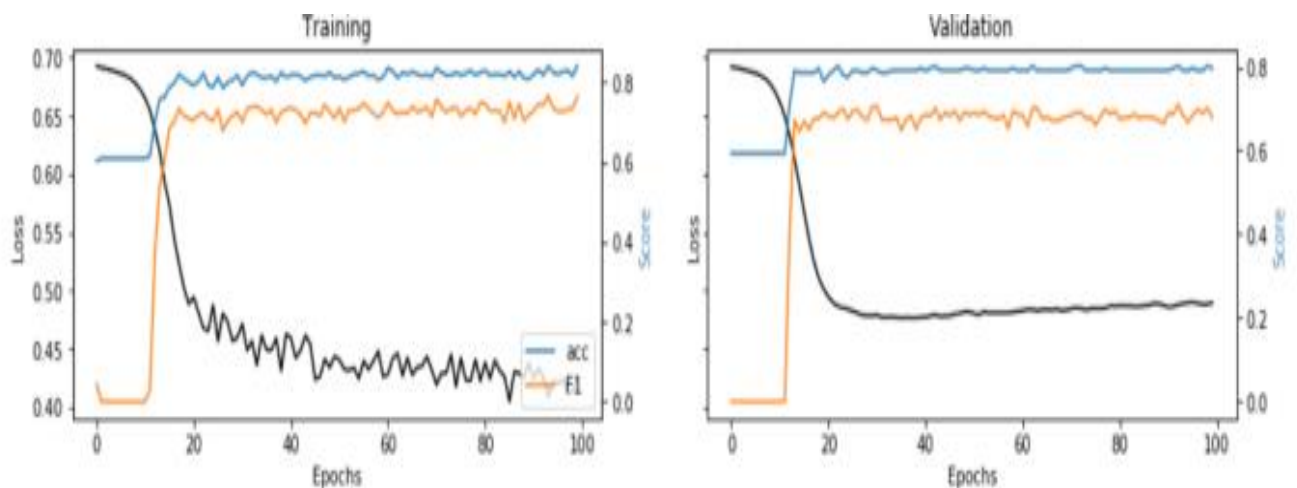
```

ax11 = ax[0].twinx()
ax[0].plot(training.history['loss'], color='black') ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('Loss', color='black')
for metric in metrics:
    ax11.plot(training.history[metric], label=metric) ax11.set_ylabel("Score",
color='steelblue')
ax11.legend()

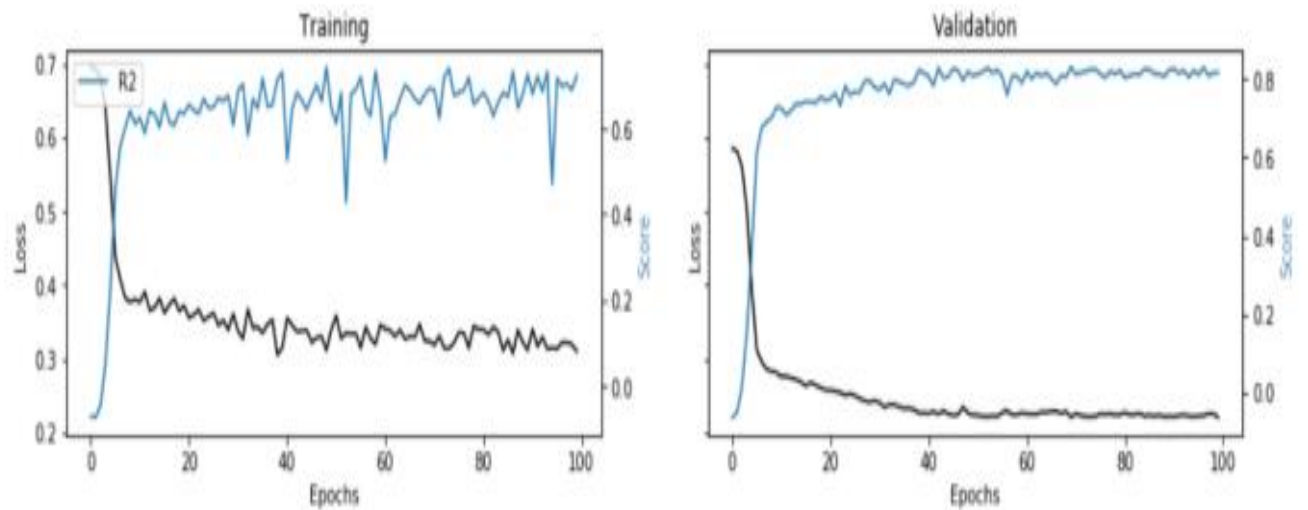
## validation
ax[1].set(title="Validation")
ax22 = ax[1].twinx()
ax[1].plot(training.history['val_loss'], color='black') ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Loss', color='black')
for metric in metrics:
    ax22.plot(training.history['val_'+metric], label=metric) ax22.set_ylabel("Score",
color="steelblue")
plt.show()

```

For Classification example,



For Regression example,



Now coming to the testing part,

```
i = l explainer_shap(model,
    X_names=list_feature_names,
    X_instance=X[i],
    X_train=X,
    task="classification", #task="regression"
    top=10)
```

