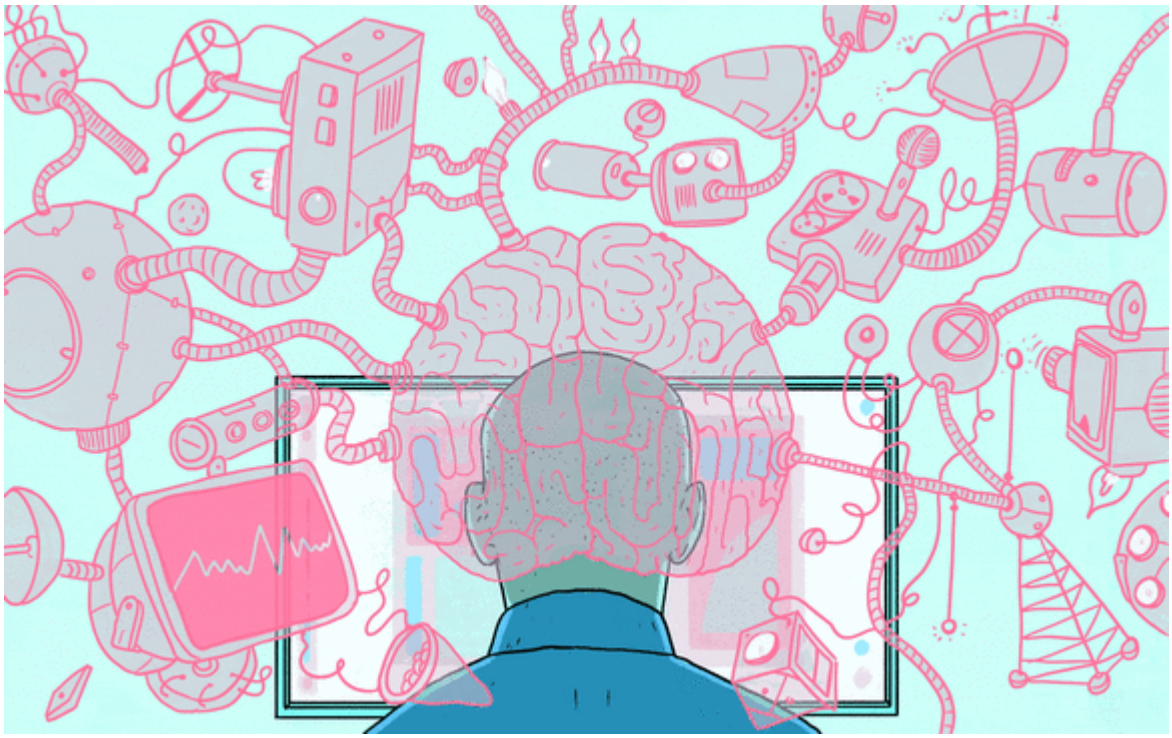
Explaination On Artificial Neural Netowrk



An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.

Importing Pandas

Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

```
import pandas as pd
```

Importing Dataset

Here i have imported dataset of churn modelling in which you have features like 'RowNumber', 'CustomerId', 'Surname', 'CreditScore', 'Geography', 'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard', 'IsActiveMember', 'EstimatedSalary', 'Exited'.

```
dataset = pd.read_csv("/content/Churn_Modelling.csv")
dataset
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Ba
0	1	15634602	Hargrave	619	France	Female	42	2	
1	2	15647311	Hill	608	Spain	Female	41	1	83
2	3	15619304	Onio	502	France	Female	42	8	159
3	4	15701354	Boni	699	France	Female	39	1	
4	5	15737888	Mitchell	850	Spain	Female	43	2	125
...	...	...	...	...	...	...	...	...	
9995	9996	15606229	Obijiaku	771	France	Male	39	5	
9996	9997	15569892	Johnstone	516	France	Male	35	10	57
9997	9998	15584532	Liu	709	France	Female	36	7	
9998	9999	15682355	Sabbatini	772	Germany	Male	42	3	75
9999	10000	15628319	Walker	792	France	Female	28	4	130

10000 rows × 14 columns

```
dataset.columns
```

```
Index(['RowNumber', 'CustomerId', 'Surname', 'CreditScore', 'Geography',
       'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard',
       'IsActiveMember', 'EstimatedSalary', 'Exited'],
      dtype='object')
```

As the *Exited* feature is to be in Y variable so below i am creating a y variable with *Exited* feature in that variable.

```
y = dataset['Exited']
```

X will contains `['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard', 'IsActiveMember', 'EstimatedSalary']` features.

```
X = dataset[['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard',
                'IsActiveMember', 'EstimatedSalary']]
```

```
X.shape
```

```
(10000, 8)
```

One Hot Encoding For Geography Feature

Here as you can see in the *Geography* reason you have you have three columns after one hot encoding i.e Germany, Spain, France. So to avoid dummy trap i have used a fn from pandas i.e `get_dummies` in which i have use `drop_first=True` that will helps us to avoid dummy trap.

```
Geo = pd.get_dummies(dataset['Geography'], drop_first=True)
Geo
```

	Germany	Spain
0	0	0
1	0	1
2	0	0
3	0	0
4	0	1
...	...	...
9995	0	0
9996	0	0
9997	0	0
9998	1	0
9999	0	0

10000 rows × 2 columns

One Hot Encoding For *Gender* Feature

Here as you can see in the Geography reason you have you have three columns after `one hot encoding` i.e Germany, Spain, France. So to avoid dummy trap i have used a fn from pandas i.e `get_dummies` in which i have use `drop_first=True` that will helps us to avoid dummy trap.

```
Gender = pd.get_dummies(dataset['Gender'], drop_first=True)
Gender
```

	Male
0	0
1	0
2	0
3	0
4	0
...	...
9995	1
9996	1
9997	0
9998	1

Concatinating X, Geo, Gender

```
X = pd.concat([X, Geo, Gender], axis=1)
X.shape
```

```
(10000, 11)
```

importing *train_test_split*

Before feeding your data into the neural network you need to split that data into `training set` and `testing set`. this can be done using

`train_test_split` method from sklearn library and from model_selection module

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.20, random_state=40)
```

importing Sequrntial Model From keras

```
from keras.models import Sequential
model = Sequential()            # creating Empty model
```

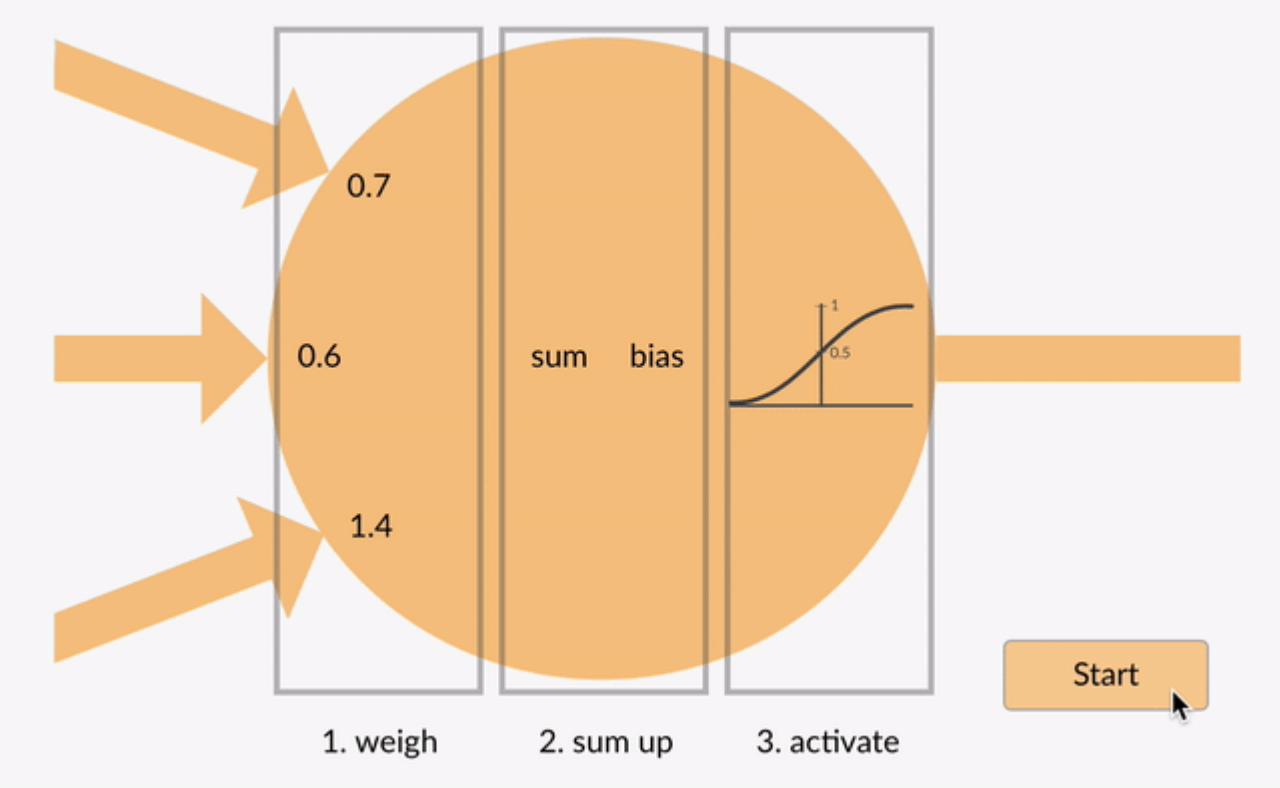
Importing Dense Layer

```
from keras.layers import Dense
```

Adding First Layer to model with neurons=8, input_feature=11 and activation fn = relu (rectified linear unit)

```
model.add(Dense(units=8, activation='relu', input_dim=11))
```

`Relu` is an activation function i.e its will activate the neurons in the hidden layers. The main functions of `relu` is that all the output from a layers from all the neuron will pass to another layers of the respective neurons.



Adding Second Layer with neurons=6 and activation fn = relu

In this layer i have used `6 neurons` and with the `relu` activation fn.

```
model.add(Dense(units=6, activation='relu'))
```

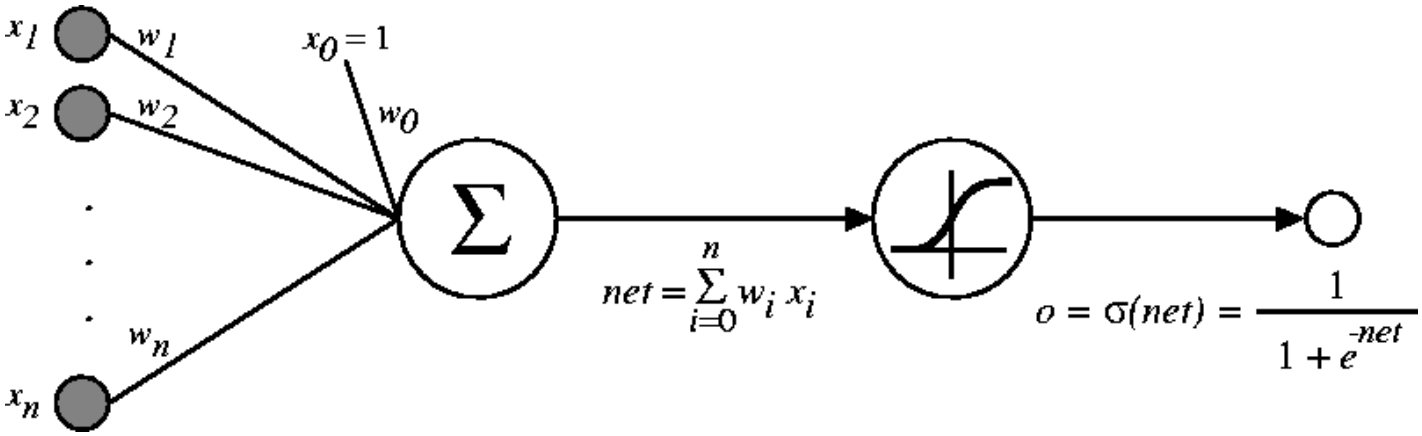
Adding third layer with neurons=6 and activation fn = relu

In this layer i have used `6 neurons` and with the `relu` activation fn.

```
model.add(Dense(units=6, activation='relu'))
```

Adding last layer with neurons=1 and activation fn = relu

In this layer i have used `1 neuron` and with the `sigmoid` activation fn. As you can see in the below image after summation from all the neurons then the output goes to `sigmoid` fn and that fn will gives you a `binary` output (1/0). As it is giving onlu one output then only `one neuron` is required.




```
model.add(Dense(units=1, activation='sigmoid'))
```

```
model.get_config()
```

```
        'batch_input_shape': (None, 11),
        'bias_constraint': None,
        'bias_initializer': {'class_name': 'Zeros', 'config': {}},
        'bias_regularizer': None,
        'dtype': 'float32',
        'kernel_constraint': None,
        'kernel_initializer': {'class_name': 'GlorotUniform',
         'config': {'seed': None}},
        'kernel_regularizer': None,
```

```
   'name': 'dense',
   'trainable': True,
   'units': 8,
   'use_bias': True}},
 {'class_name': 'Dense',
  'config': {'activation': 'relu',
   'activity_regularizer': None,
   'bias_constraint': None,
   'bias_initializer': {'class_name': 'Zeros', 'config': {}},
   'bias_regularizer': None,
   'dtype': 'float32',
   'kernel_constraint': None,
   'kernel_initializer': {'class_name': 'GlorotUniform',
    'config': {'seed': None}},
   'kernel_regularizer': None,
   'name': 'dense_1',
   'trainable': True,
   'units': 6,
   'use_bias': True}},
 {'class_name': 'Dense',
  'config': {'activation': 'relu',
   'activity_regularizer': None,
   'bias_constraint': None,
   'bias_initializer': {'class_name': 'Zeros', 'config': {}},
   'bias_regularizer': None,
   'dtype': 'float32',
   'kernel_constraint': None,
   'kernel_initializer': {'class_name': 'GlorotUniform',
    'config': {'seed': None}},
   'kernel_regularizer': None,
   'name': 'dense_2',
   'trainable': True,
   'units': 6,
   'use_bias': True}},
 {'class_name': 'Dense',
  'config': {'activation': 'sigmoid',
   'activity_regularizer': None,
   'bias_constraint': None,
   'bias_initializer': {'class_name': 'Zeros', 'config': {}},
   'bias_regularizer': None,
   'dtype': 'float32',
   'kernel_constraint': None,
   'kernel_initializer': {'class_name': 'GlorotUniform',
    'config': {'seed': None}},
   'kernel_regularizer': None,
   'name': 'dense_3',
   'trainable': True,
   'units': 1,
   'use_bias': True}}],
 'name': 'sequential'}
```

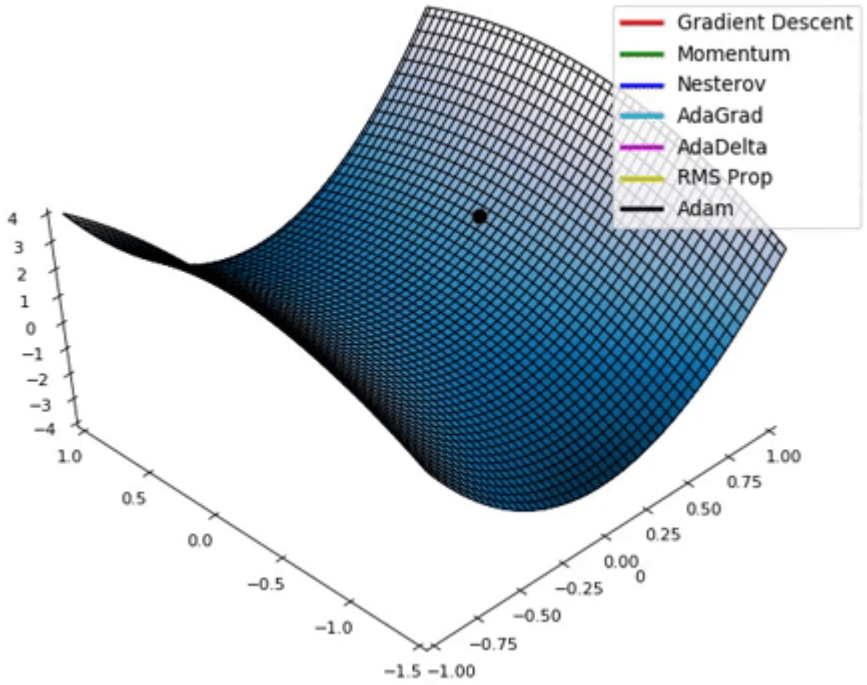

A visaul Explaination of the Artificial Neural Network

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense (Dense)                (None, 8)                 96        
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 54        
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 42        
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 7         
=================================================================
Total params: 199
Trainable params: 199
Non-trainable params: 0
_________________________________________________________________
```

As you can see in below **`Optimizers Comaprision`** You will find that there are different tyes of optimizers are shown. In which they have different speed to move. This speed denotes the learning from the weights. if your optimizers have less speed then it will learn more things but as you can see in terms of *`Ada Delta`* the speed is quite high. So its jups and moves out of the graph. where as

in case of *Adam* because i have used Adam you will find that speed of learning from the weights is quite slow compare to *Ada Delta*. Also you need to set the `learning rate` to move your optimizers very slowly. So, it can learn much more from the respective weights.




```
from keras.optimizers import Adam
```

As i have only one output that whether the Employee is exited from the company or not. i.e Binary output (Exited/notexited). So the loss will be generating in binary. To handle the binary loss we have `binary_crossentropy`.

```
model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.000001))
```

Below You will find the weights of the model

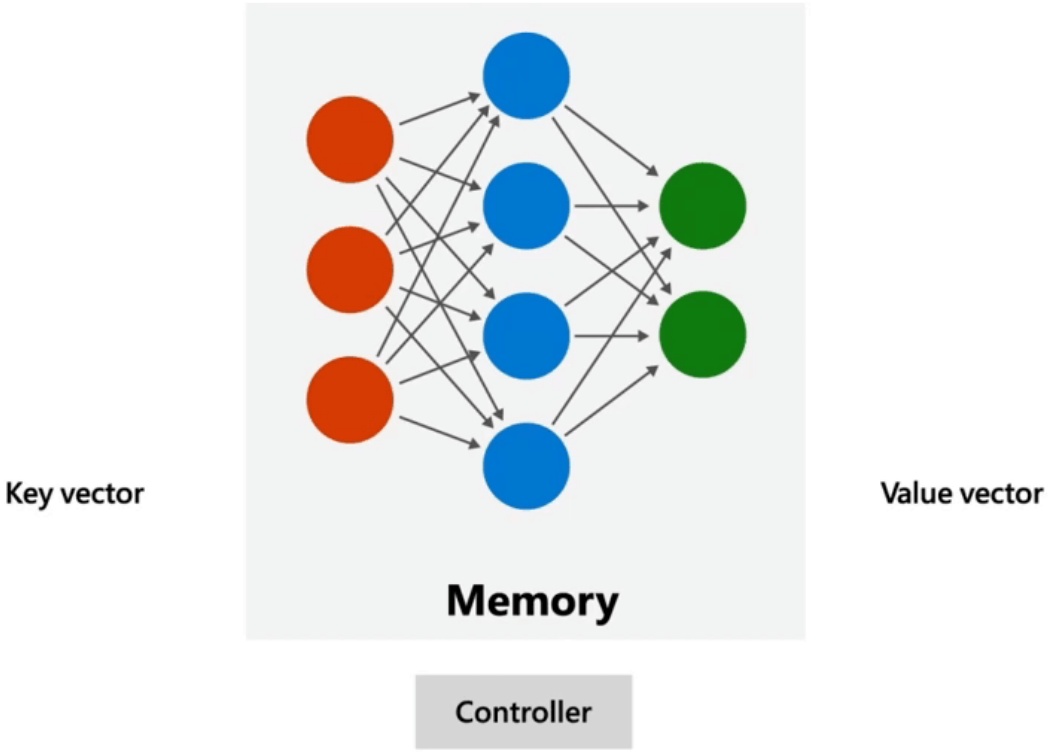
```
model.get_weights()
```

```
[array([[-0.02066076, -0.28968173,  0.2522359 , -0.11069694, -0.30354032,
         -0.35479504, -0.37940133,  0.12989408],
        [ 0.40807408,  0.33400285, -0.29044044,  0.12279463,  0.02866024,
          0.16505909, -0.43437654, -0.08052927],
        [ 0.08574432,  0.31294584,  0.34127384, -0.5403073 ,  0.22737062,
          0.3964677 , -0.4407091 , -0.11779466],
        [-0.47511214, -0.51364213, -0.5219625 ,  0.08389795,  0.0482738 ,
```

```
         -0.21094859, -0.04436564,  0.3212192 ],
        [ 0.21555561, -0.07330978,  0.21048409,  0.51580375,  0.3181417 ,
         -0.37364948,  0.38994366,  0.35384363],
        [ 0.4174667 , -0.47016186, -0.18425742,  0.19694513, -0.42180163,
         -0.0538637 ,  0.02123272,  0.3795498 ],
        [-0.50346905,  0.22021902, -0.0587213 , -0.39206484, -0.11001834,
          0.39269185, -0.01426095, -0.34227383],
        [ 0.14976585,  0.4215495 , -0.4487199 , -0.10118306,  0.44541937,
         -0.02592146, -0.3683002 , -0.15695423],
        [ 0.04658729,  0.3677683 ,  0.50780183, -0.18169224, -0.40672716,
         -0.27327177,  0.416915  ,  0.33695632],
        [ 0.55054265, -0.0431208 ,  0.08847803, -0.4176634 ,  0.41596174,
          0.06407928,  0.22342652,  0.4530285 ],
        [-0.1881845 ,  0.55957144,  0.11970764, -0.53179413,  0.19073129,
          0.4368927 , -0.16713011, -0.24164882]], dtype=float32),
 array([0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32),
 array([[-0.38369834, -0.45466506,  0.08300167,  0.6431782 , -0.137074  ,
          0.62779343],
        [-0.00921959, -0.05148399, -0.43598944, -0.47586703, -0.3992074 ,
         -0.46359885],
        [ 0.02408886, -0.4984544 ,  0.06853729, -0.09909028,  0.11974955,
          0.47140014],
        [-0.418697  , -0.11011851,  0.34275085,  0.45947206, -0.39859602,
          0.00415146],
        [-0.56797343, -0.41449997,  0.37440336, -0.2053113 , -0.51773435,
          0.569739  ],
        [ 0.41156614, -0.1457048 , -0.04102856,  0.2835266 ,  0.49666166,
         -0.16993055],
        [ 0.25853467,  0.01449186,  0.23077959, -0.44923282,  0.1784932 ,
         -0.4885732 ],
        [-0.18333298, -0.39147   ,  0.06050986, -0.21145234, -0.38709328,
          0.0710482 ]], dtype=float32),
 array([0., 0., 0., 0., 0., 0.], dtype=float32),
 array([[ 0.31347698,  0.2991895 ,  0.24508625,  0.2747    ,  0.58385545,
          0.46211082],
        [-0.45942527, -0.28549853, -0.4627154 ,  0.10429323,  0.35801822,
         -0.4081499 ],
        [-0.5215674 , -0.36142457, -0.2062847 ,  0.39537436,  0.61899203,
         -0.57093555],
        [-0.14456117,  0.17730016, -0.35279912,  0.23425555,  0.23910499,
         -0.484769  ],
        [ 0.24732089,  0.07750112,  0.2086935 ,  0.1879257 , -0.53177184,
          0.12294507],
        [ 0.45292348, -0.3549444 ,  0.19589955, -0.08474702, -0.20897552,
         -0.52293193]], dtype=float32),
 array([0., 0., 0., 0., 0., 0.], dtype=float32),
 array([[-0.29135615],
        [-0.6119803 ],
        [ 0.6753768 ],
        [-0.05632192],
        [ 0.12343192],
        [ 0.11105609]], dtype=float32),
```

Training Phase

For training the model we need to fit the model and it require training data i.e `X_train, y_train`. And the `epochs is 100`. That means your training data will goes 100 times through the neural network which you have build above with the respective layers.




```
model.fit(X_train, y_train, epochs=100)
```

```
Epoch 72/100
250/250 [==============================] - 0s 1ms/step - loss: 17.1282
Epoch 73/100
250/250 [==============================] - 0s 1ms/step - loss: 18.0116
Epoch 74/100
250/250 [==============================] - 0s 1ms/step - loss: 17.9715
Epoch 75/100
250/250 [==============================] - 0s 1ms/step - loss: 17.0897
Epoch 76/100
250/250 [==============================] - 0s 1ms/step - loss: 16.8372
Epoch 77/100
250/250 [==============================] - 0s 1ms/step - loss: 16.8302
Epoch 78/100
250/250 [==============================] - 0s 1ms/step - loss: 16.5838
Epoch 79/100
250/250 [==============================] - 0s 1ms/step - loss: 16.4868
Epoch 80/100
250/250 [==============================] - 0s 1ms/step - loss: 15.5716
Epoch 81/100
250/250 [==============================] - 0s 1ms/step - loss: 14.5191
Epoch 82/100
```

```
250/250 [==============================] - 0s 1ms/step - loss: 14.1772
Epoch 83/100
250/250 [==============================] - 0s 1ms/step - loss: 14.3615
Epoch 84/100
250/250 [==============================] - 0s 1ms/step - loss: 13.8075
Epoch 85/100
250/250 [==============================] - 0s 1ms/step - loss: 13.5754
Epoch 86/100
250/250 [==============================] - 0s 1ms/step - loss: 13.4503
Epoch 87/100
250/250 [==============================] - 0s 1ms/step - loss: 13.3970
Epoch 88/100
250/250 [==============================] - 0s 1ms/step - loss: 12.4848
Epoch 89/100
250/250 [==============================] - 0s 1ms/step - loss: 12.5361
Epoch 90/100
250/250 [==============================] - 0s 1ms/step - loss: 11.9010
Epoch 91/100
250/250 [==============================] - 0s 1ms/step - loss: 12.0105
Epoch 92/100
250/250 [==============================] - 0s 1ms/step - loss: 11.4991
Epoch 93/100
250/250 [==============================] - 0s 989us/step - loss: 10.9268
Epoch 94/100
250/250 [==============================] - 0s 1ms/step - loss: 10.6491
Epoch 95/100
250/250 [==============================] - 0s 1ms/step - loss: 10.7060
Epoch 96/100
250/250 [==============================] - 0s 1ms/step - loss: 10.2463
Epoch 97/100
250/250 [==============================] - 0s 979us/step - loss: 10.1727
Epoch 98/100
250/250 [==============================] - 0s 1ms/step - loss: 10.1703
Epoch 99/100
250/250 [==============================] - 0s 1ms/step - loss: 9.7232
Epoch 100/100
250/250 [==============================] - 0s 1ms/step - loss: 9.7264
<tensorflow.python.keras.callbacks.History at 0x7f03367dfe50>
```

```
loss = pd.DataFrame(model.history.history)
loss
```

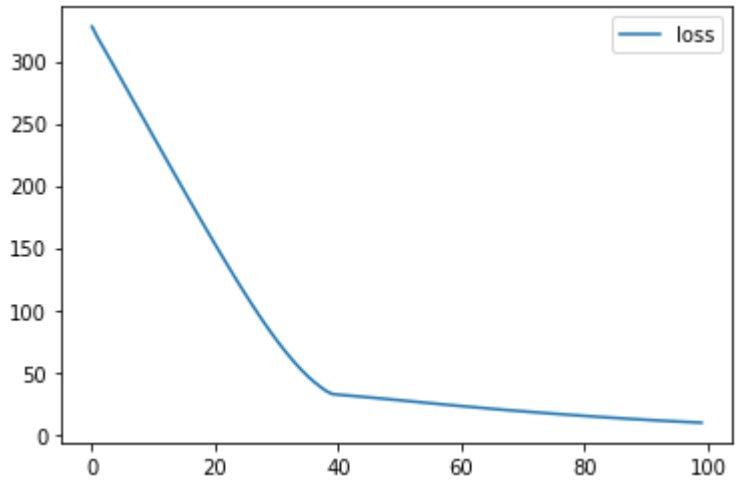
	loss
0	327.813019
1	318.202637
2	309.605621
3	300.935791

Plotting Loss Graph

As you can see the graph of the loss is slowly decreasing. So this can be possible because of `Adam` Optimizers

```
loss.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0334101390>
```



Prediction

To predict i am just giving random iputs but you can use the right values and it will predict on that case.

```
print("The employee will  :", model.predict([[1,2,3,4,5,6,7,8,9,10,11]])[0][0])
```

```
The employee will  : 0.4933369
```