# Introduction to Deep Learning and Tensorflow with Keras API

Recently I saw a post on the facebook by one of friend and he challenged me.
1 = 12
2 = 21
3 = 30
10 = 93
20 = 183
80 = 723
54 = 489
72 = 651
90 = 813
36 = ?

I was not able to answer the question immediately. After a while I found out the input number should be multipled by 9 and add 3 gives the result.
1 = 1*9 + 3*
*90 = 90*9 + 3
The same way 36 = 36*9 + 3.
So the final answer is 327 Now I can answer for any number just by writing a simple program. Wait.. How did I achieve the program logic. Because I am basically smart enough to identify these kind of solutions(just kidding) so I say this is of my natural intelligence. From now onwards I can do the calculation for any input number, because I was trained for the above set of input and output values. To make my job easy I wrote a program using simple python as mentioned below.

In [1]:
```
def estimate(x):
    return x*9 + 3
```

In [2]:
```
estimate(36)
```

Out[2]:
```
327
```

The above logic is so simple(atleast for me), but what if the logic is too complex, what if instead of single input if we need to calculate with more inputs to derive the final output.
Can we identify the function or logic to calculate the output value?
Can we write a program to identify the logic?
Can we build a brain that mimics you or me..?
Why not, we can build one using artificial intelligence that mimics us(you and me).
So our expected function is f(x) = 9x + 3 or y = 9x + 3

The above task can be achieved by traditional machine learning algorithms. But in this course we will build a Neural network model.
Just think of our brain. Our brain contains lot of nuerons and every neuron will process some input and provides the output to the next nueron or nuerons. Our brain nuerons or interconnected and responsible for processing the input for the actions.

We will use tensor flow to build a model to process the above input. We use tensorflow 2.x version

In [3]:
```
import tensorflow as tf
import numpy as np
```

Define the input and output to train our model. Our tensorflow accepts only the numerical values. So every input that we provide to the model should be converted to numerical values.

In [4]:
```
input = np.array([1, 2, 3, 10, 20, 80, 54, 72, 90],  dtype=float)
output = np.array([12, 21, 30, 93, 183, 723, 489, 651, 813],dtype=flo
at)
```

Now we are ready with input and output values to train our model. As a first step we will build a small neural network model to understand how the neural network model works. Every nueron you define in the model will accept the input and process the output. For example if a neuron receives an input x it calculates the output as a = wx + b where as

a = output
w = weight
b = bias.
Every input is multiplied by a weight(w) and adds the bias. These weights and bias will be adjusted as we train our model.

Our model structure is as below.

1. Input, Output Values for training.
2. Layer of single nueron
3. Output

In the first step we define the layer that accepts the input and calculates the output`

In [5]:
```
layer = tf.keras.layers.Dense(units=1, input_shape=[1])
```

Add the above defined layer to a model. We define the Sequential Model and add the layer by layer (if we have any multiple layers)

In [6]:
```
model = tf.keras.Sequential([layer])
```

We defined the model with layers. No we need to compile our model and tell it how to validate its output against the actual output. How to finetune itself, we teach the method to rectify itself.
Loss function - This tell what is the error(difference) value of each predicted output versus the actual output
Optimier function - Way to adjust the weights and biases. The minimum adjustment might take more time to train the model or overfitting. The large adjustment values may create low accurate models.
We will discuss more on these parameters going forward.

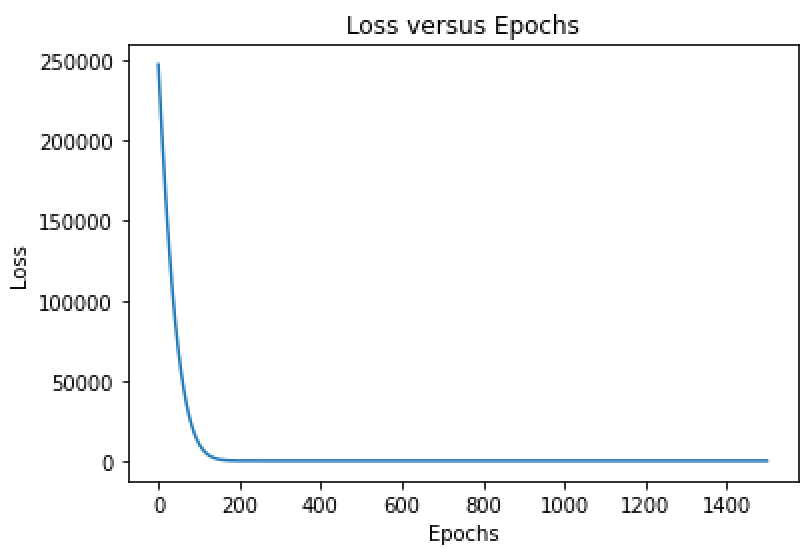
In [7]:
```
model.compile(loss=tf.keras.losses.mean_squared_error,
              optimizer=tf.keras.optimizers.Adam(0.1))
```

Now we will provide the sample training data and will run the above defined mode to fit the input data.

In [8]:
```
history = model.fit(input, output, epochs=1500, verbose=False)
```

Lets plot the loss of our model over the epochs

In [9]:
```
import matplotlib.pyplot as plt
plt.figure()
plt.title("Loss versus Epochs")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.plot(history.history['loss'])
plt.show()
```



From the above plot we can say our model learnt pretty fast upto 125 epochs and after that the loss is almost flat but with consistent difference. As the epochs increases the loss will decrease. We will discuss on the number of epochs(early stops or finetuning the hyperparamets in the next sessions).

Lets Predict an input which is not part of our training model. 36 is not part of our model training.

In [10]:
```
model.predict([36])
```

Out[10]:
```
array([[328.14078]], dtype=float32)
```

Actual answer is $36 \times 9 + 3 = 327$ Looks like our model is doing pretty well.

Like wise we can predict other variables as well

In [11]:
```
validate_set = [25, 212, 49, 102]
```

In [12]:
```
model.predict(validate_set)
```

Out[12]:
```
array([[ 229.52406],
       [1906.0082 ],
       [ 444.6878 ],
       [ 919.84106]], dtype=float32)
```

Actual outputs

In [13]:
```
for val in validate_set:
    print(estimate(val))
```

```
228
1911
444
921
```

As we discussed above we defined the Neural network model with single nueron. Lets print the model summary

In [14]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense (Dense)                (None, 1)                 2         
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

Here it shows param as 2. How to calculate the params at each learning layer(will discuss about the learning layers later). Here it is single neuron it takes only two parameters as weight and bias. Lets see what are the weights and bias of that single neuron in the layer.

In [15]:
```
layer.get_weights()
```

Out[15]:
```
[array([[8.965156]], dtype=float32), array([5.395173], dtype=float3
2)]
```

As you see the above function prints the weights and bias. Here the weight is 8.986033 and bias is 3.9599

The single neuron constructed a function on the input as $8.986033 \times input + 3.9599$

If you observe the above equation it is close to our actual function. Here the bias was not adjusted close to 3 but we see the function is almost same as our actual function. As it is artificial there is always a chance for the error which can be negotiable at this point of time.

Lets make it little complex and add some more dense layer in between. Lets see the accuracy. Define the 3 layers.

1. Layer1 has 2 neurons accepting the single input
2. Layer2 has 4 neurons accepting the input from the 2 neurons from Layer1
3. Layer3 has single neuron accepts input from 4 neurons from Layer2

In [16]:
```
layer1 = tf.keras.layers.Dense(units=2, input_shape=[1])
layer2= tf.keras.layers.Dense(units=4)
layer3 = tf.keras.layers.Dense(units=1)
```

In [17]:
```
model2 = tf.keras.Sequential([layer1, layer2, layer3])
model2.compile(loss=tf.keras.losses.mean_squared_error,
               optimizer=tf.keras.optimizers.Adam(0.1))
```

In [18]:
```
model2.fit(input, output, epochs=1500, verbose=False)
```

Out[18]:
```
<tensorflow.python.keras.callbacks.History at 0x7fa54e6e9050>
```

In [19]:
```
model2.predict([36])
```

Out[19]:
```
array([[327.]], dtype=float32)
```

In [20]:
```
layer1.get_weights()
```

Out[20]:
```
[array([[ 1.425346 , -0.8117417]], dtype=float32),
 array([-0.18474889, -0.10386661], dtype=float32)]
```

In [21]:
```
layer2.get_weights()
```

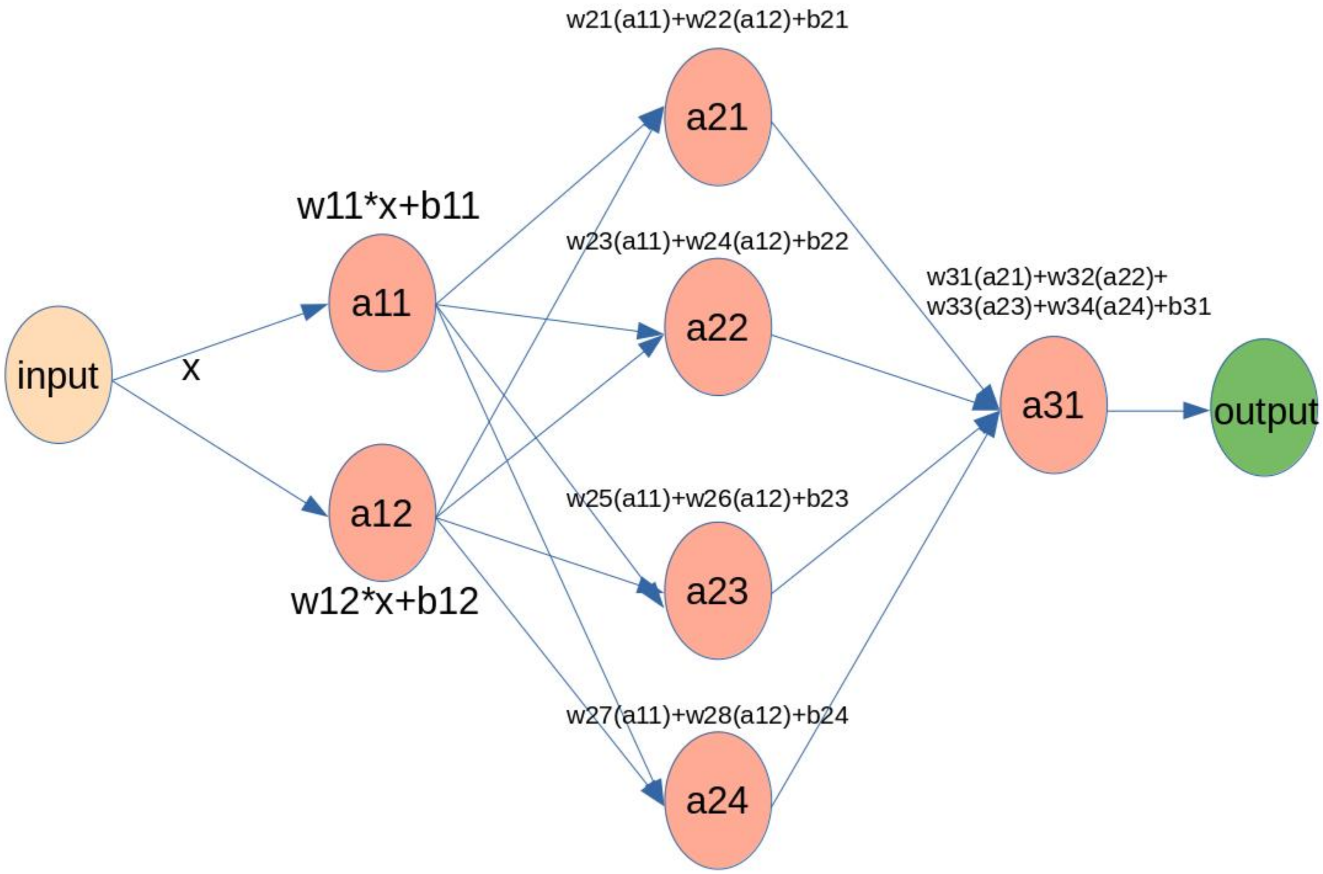
Out[21]:
```
[array([[ 1.2766095 , -1.7106092 , -1.1167812 , -0.5195032 ],
        [-1.0800608 ,  0.7388127 ,  0.44779032,  1.0377523 ]],
       dtype=float32),
 array([ 0.56280273, -0.46462837, -0.8300078 , -0.8535919 ], dtype=fl
oat32)]
```

In [22]:
```
layer3.get_weights()
```

Out[22]:
```
[array([[ 1.0267274 ],
        [-1.0397569 ],
        [-1.2010288 ],
        [-0.45772845]], dtype=float32),
 array([1.1137961], dtype=float32)]
```

The above weigths and biases looks complex but they also yield the better results.

If you see the above results they are much closer than the previous "model". Adding more layers might increase the accuracy of the model as it is more prune to adjust the weights and biases.
But the more layers make the model complex and thus it is little difficult to understand.


w21(a11)+w22(a12)+b21
a21
w11*x+b11
w23(a11)+w24(a12)+b22
w31(a21)+w32(a22)+
w33(a23)+w34(a24)+b31
a11
a22
input
x
a31
output
a12
w25(a11)+w26(a12)+b23
a23
w12*x+b12
w27(a11)+w28(a12)+b24
a24

| Input Layer | Hidden Layer | | | | | | | | | Output Layer |
|---|---|---|---|---|---|---|---|---|---|---|
| | Layer1 | | | Layer2 | | | Layer3 | | | |
| | | | | w21 | 1.2766095 | a21 | | | | |
| | | | | w22 | -1.0800608 | | | | | |
| | | | | b21 | 0.56280273 | | | | | |
| | | | | | | | | | | |
| | | | | w23 | -1.7106092 | a22 | | | | |
| | w11 | 1.425346 | a11 | w24 | 0.7388127 | | w31 | 1.0267274 | a31 | |
| x | b11 | -0.18474889 | | b22 | -0.46462837 | | w32 | -1.0397569 | | y |
| | | | | | | | w33 | -1.2010288 | | |
| | w12 | -0.8117417 | a22 | w25 | -1.1167812 | a23 | w34 | -0.45772845 | | |
| | b12 | -0.10386661 | | w26 | 0.44779032 | | b31 | 1.1137961 | | |
| | | | | b23 | -0.8300078 | | | | | |
| | | | | | | | | | | |
| | | | | w27 | -0.5195032 | a24 | | | | |
| | | | | w28 | 1.0377523 | | | | | |
| | | | | b24 | -0.8535919 | | | | | |

The input from Input Layer goes to 2 neurons in Layer1. And the equation is as below.
w11*x+b11=a11*
*w12*x+b12=a12

Each output of Layer1 goes to 4 neurons in the Layer2 and the equation is as below.
w21(a11)+w22(a12)+b21=a21
w23(a11)+w24(a12)+b22=a22
w25(a11)+w26(a12)+b23=a23
w27(a11)+w28(a12)+b24=a24

The above output from Layer2 goes to 2 neurons in the Layer3
w31(a21) +w32(a22)+w33(a23)+w34(a24)+b31=a31(output)
The above final equation will yield a31=wx+b where the w, b close to 9 and 3

After many calculations the final equation is(It takes lot of time, try when you are free)
y=9*x+b

In [23]:
```
model2.predict(validate_set)
```

Out[23]:
```
array([[ 228.00002],
       [1911.     ],
       [ 444.00006],
       [ 920.99994]], dtype=float32)
```

In [24]:
```
for val in validate_set:
    print(estimate(val))
```

```
228
1911
444
921
```

In [26]:
```
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 2)                 4         
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 12        
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 5         
=================================================================
Total params: 21
Trainable params: 21
Non-trainable params: 0
_________________________________________________________________
```

In the model2 we have total of 7 neurons compared to 1 neuron in the earlier model. After our study we can say that model2 is giving more accurate results. Thought more neurons makes the model complex but it might result more accurate result(This statement is not true always).

The above example is a classic representation of Regression problem using Deep Neural Networks. In the next section we will discuss the Classification Model.

In [29]:
```
import os
print(os.environ['PATH'])
```

```
/home/rajeshpriyanka/anaconda3/bin:/home/rajeshpriyanka/anaconda3/con
dabin:/home/rajeshpriyanka/.local/bin:/usr/local/sbin:/usr/local/bi
n:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

In [ ]: