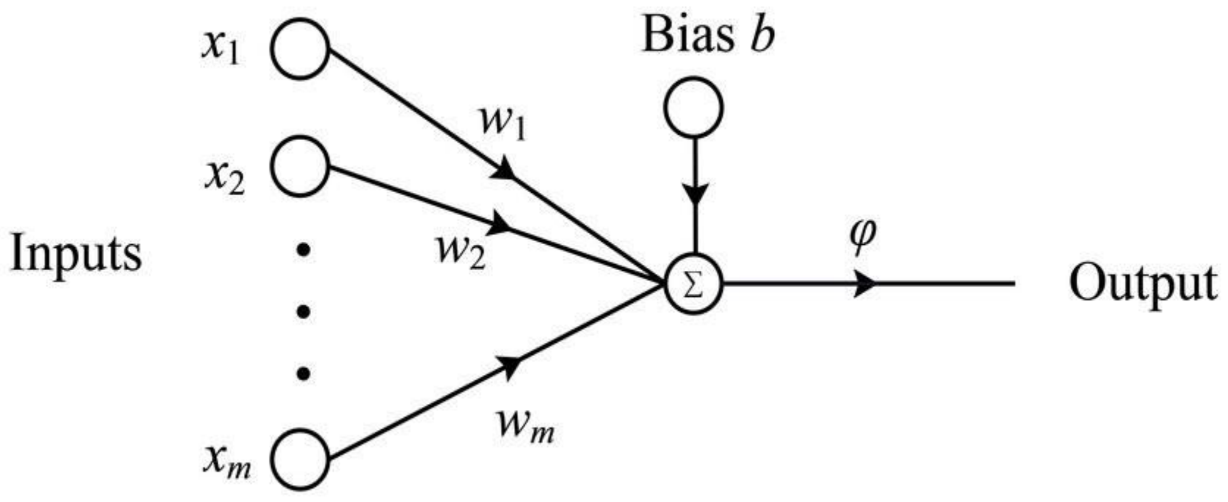
1)

→ A neural network is a mathematical function defined by a set of neurons which are connected to each other. A neuron is a mathematical model/function which takes single or multiple numbers as input and outputs a single number which can be defined as the activation. The below diagram will give us a clear idea: -



To describe the above image let me first define the term associated with the neurons,

X_1, X_2, …., X_m – These are the inputs of the function.

W_1, W_2, …, W_m – These are the weights assigned with each input. They are used for describing the strength of the connection between neurons.

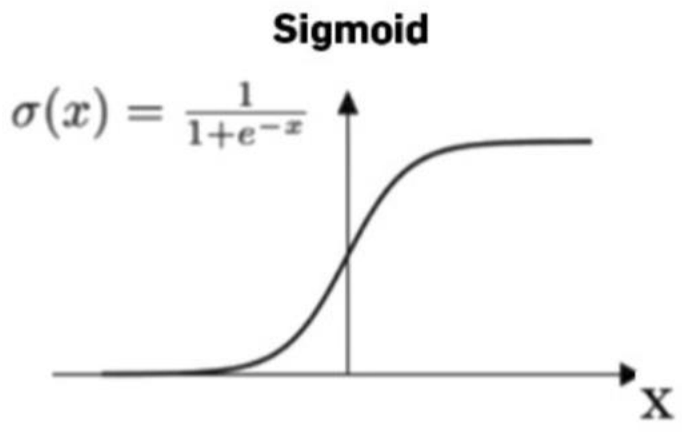
B – Bias can shift the activation function thus the neurons need a lower or higher value to fire. Basically, bias value activation function to be shifted to the right or left hence better fitting the data. Also, I think that bias always influences the output data and not the input data.

This weighted input data from all the sources is then summed to produce a single value which is then feed to an activation function to turn the output into signal. The activation of the node in neural networks can be defined as follows: -

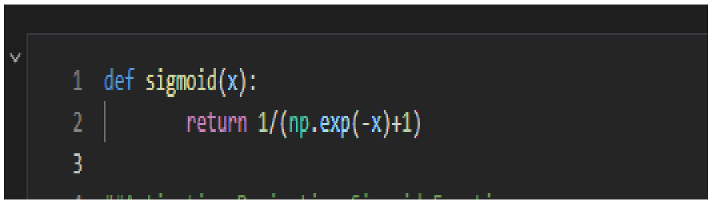
Output = Sigmoid (dot product (weights, inputs) + bias)

I think that the activation function helps in making the decision i.e., when it is given some weighted features from some data it will tell us whether the features are important enough to contribute to a classification.

For our code we are using a sigmoid function as our activation function.



In python we can define the sigmoid function as follows: -



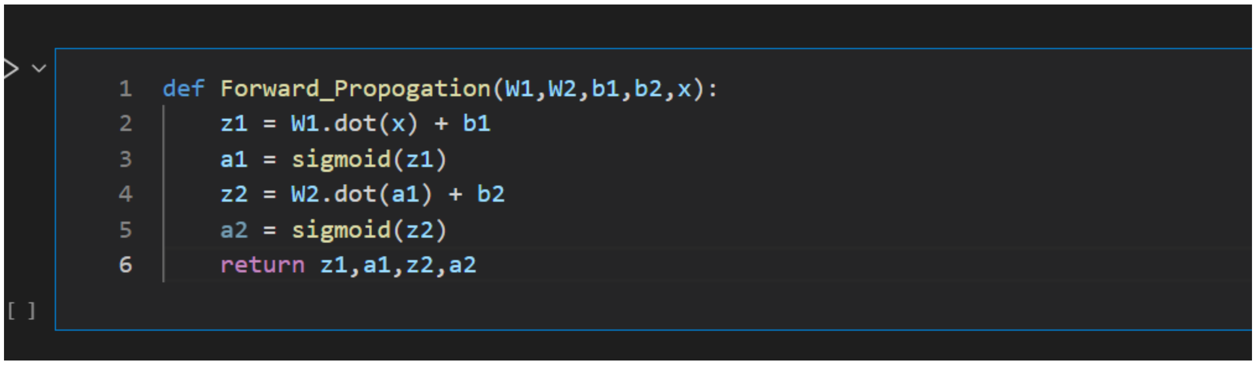
```python
def sigmoid(x):
        return 1/(np.exp(-x)+1)

```

Now that we have established a background, we will see how neural networks is trained:

Training of Neural Networks –

1) Forward Propagation

   In forward propagation we are multiplying the inputs with initialized weights and then adding the bias to every weighted input. This biased weighted input is then sent to the sigmoid function from there we get the nonlinear values. In case of multiple layer network, we multiply the output from the sigmoid with the weight of the next layer and add bias to the same. This output is again sent to the sigmoid function again to predict the output. In the code we have defined forward propagation as follows: -



```python
def Forward_Propogation(W1,W2,b1,b2,x):
    z1 = W1.dot(x) + b1
    a1 = sigmoid(z1)
    z2 = W2.dot(a1) + b2
    a2 = sigmoid(z2)
    return z1,a1,z2,a2
```

Here W1 = is the weight of layer 1(input layer), W2 = weights of the hidden layer,

Here B1 = bias for layer 1, B2 = bias for layer 2

2) Backward Propagation

This is learning step for network where we update the weights and biases based the difference between predicted and desired output. To measure the error of the networks we use the loss function. For loss we are using Mean squared error loss.

First, we will define the cost function,

$$C(w,b) = \frac{1}{2\mathrm{n}}\sum(y_{\mathrm{i}} - \hat{\mathrm{y}}(\mathrm{x_i}))^2$$

To minimize the error over time we look for a global minimum. To obtain this global minimum we are using gradient descent in our code. Here to calculate the descent quickly we use learning rate. We multiply the learning rate with the updated weights and subtract them from the original weights to get updated weights.

To calculate the loss change with respect to the error on the outer layer we use the chain rule,

Therefore,

$$\partial_j^L = \frac{\partial C}{\partial z_j^L}$$

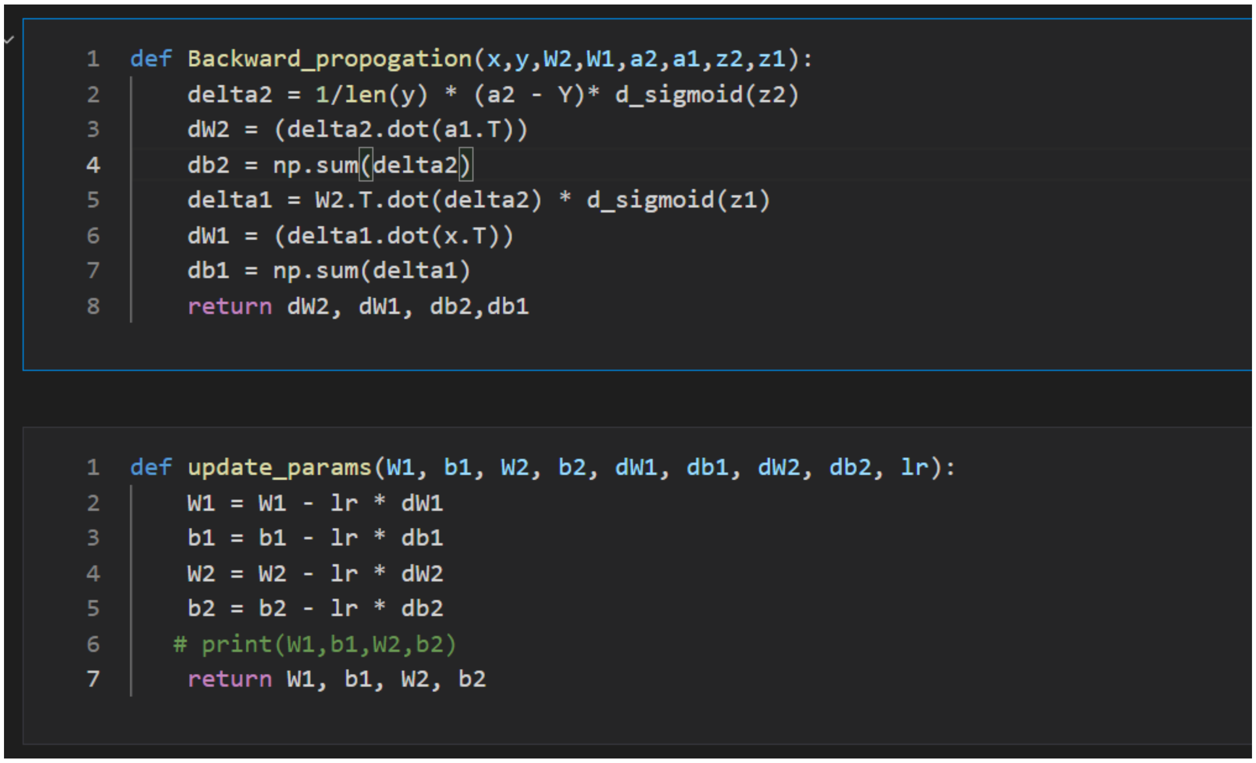
$$= \frac{\partial c}{\partial a_j^L} \cdot \sigma'(z_j^L)$$

Where, $\sigma'(z) = \sigma(z) \cdot \sigma(1 - z)$

From the above equation we can determine that we must take the derivation of our activation function Sigmoid. Therefore, in our backward propagation we calculate the ∂_j^L values and then update the weights $w_{j,k}^l \leftarrow w_{j,k}^l - \eta a_k^{l-1}\, \delta_j^l$.

To update the weights, we calculate the delta values of all the weights i.e., we take the dot product of the previous layer output and the weight. For the biases we sum all the values of delta. These updated weights and biases are then multiplied by the learning rate and then subtracted from the original weights/biases which have been initialized.

Following code screenshots will show backward propagation and update parameters function in python: -



```python
def Backward_propogation(x,y,W2,W1,a2,a1,z2,z1):
    delta2 = 1/len(y) * (a2 - Y)* d_sigmoid(z2)
    dW2 = (delta2.dot(a1.T))
    db2 = np.sum(delta2)
    delta1 = W2.T.dot(delta2) * d_sigmoid(z1)
    dW1 = (delta1.dot(x.T))
    db1 = np.sum(delta1)
    return dW2, dW1, db2,db1
```

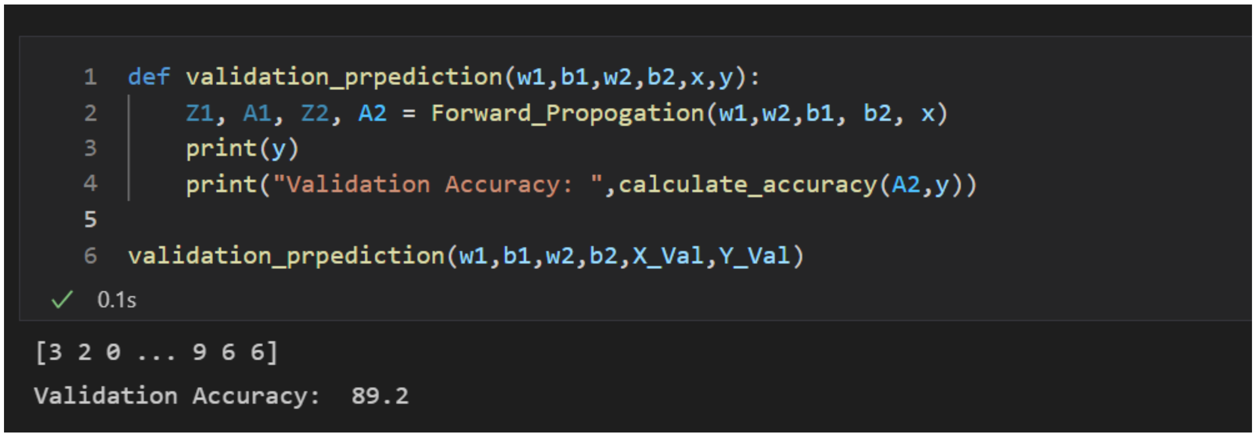
```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, lr):
    W1 = W1 - lr * dW1
    b1 = b1 - lr * db1
    W2 = W2 - lr * dW2
    b2 = b2 - lr * db2
   # print(W1,b1,W2,b2)
    return W1, b1, W2, b2
```

3) Stochastic Gradient Descent

To increase the efficiency of calculating the gradient for each sample we split the batch into mini batches and then compute the gradient between them. To maintain the stochastic part, we shuffle the dataset and then split the training set into multiple batches.

For Python program without using the Pytorch we get the following observations: -

1) Validation Accuracy

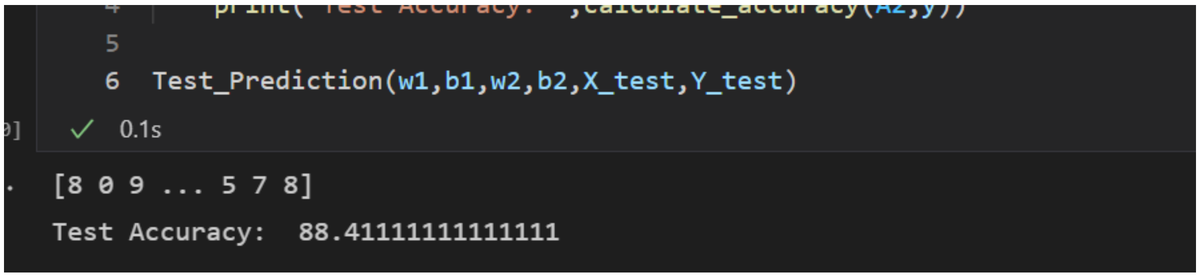


```python
def validation_prpediction(w1,b1,w2,b2,x,y):
    Z1, A1, Z2, A2 = Forward_Propogation(w1,w2,b1, b2, x)
    print(y)
    print("Validation Accuracy: ",calculate_accuracy(A2,y))

validation_prpediction(w1,b1,w2,b2,X_Val,Y_Val)
```

✓ 0.1s

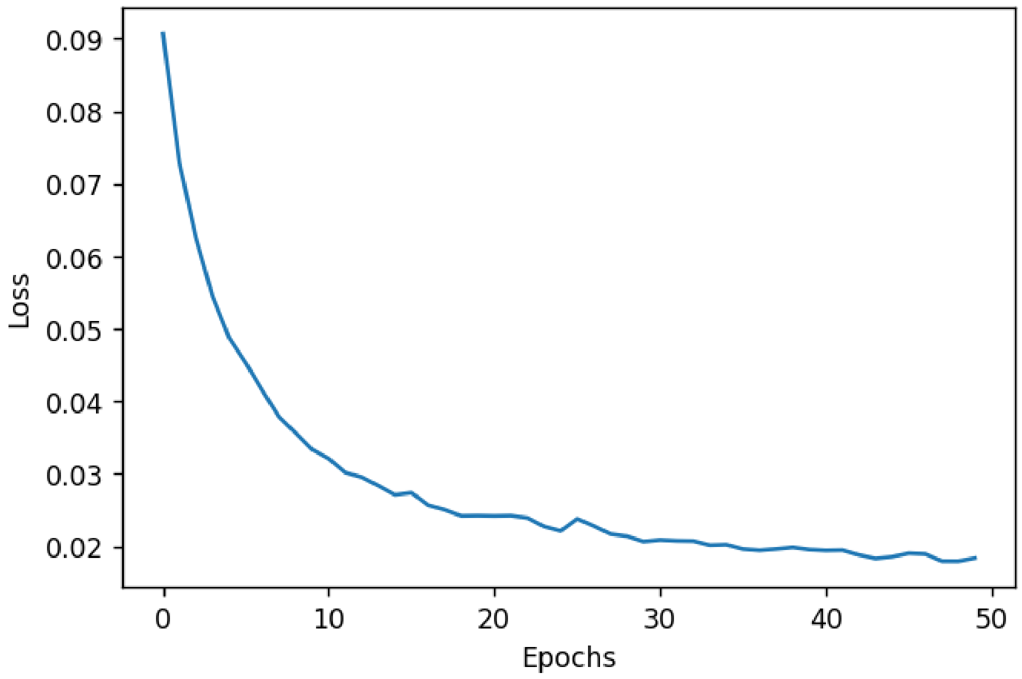
```
[3 2 0 ... 9 6 6]
Validation Accuracy:  89.2
```

2) Test Accuracy

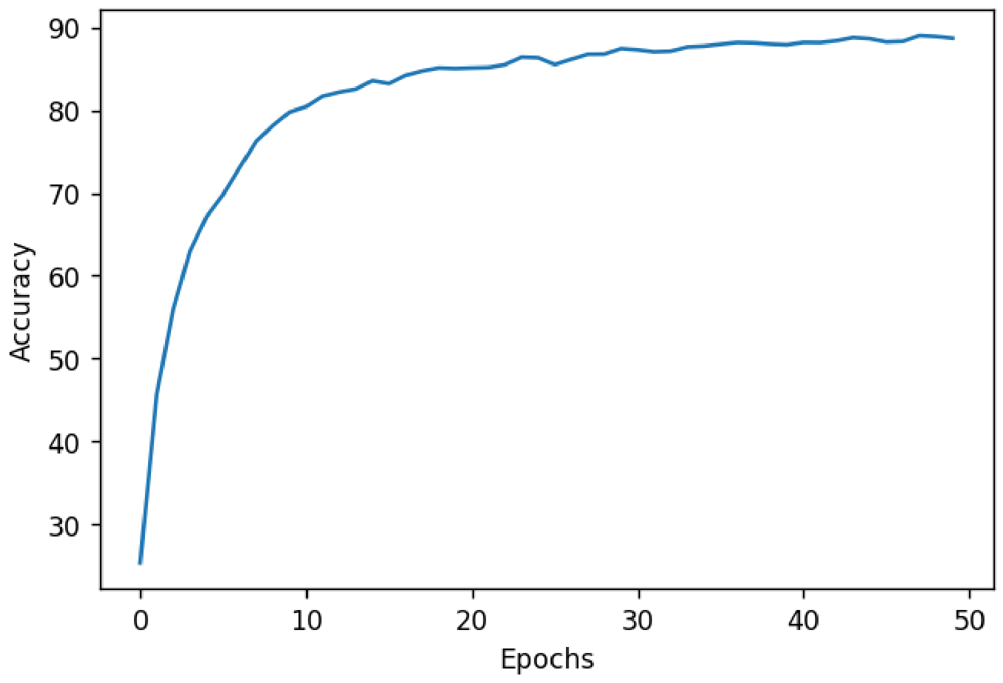


```
    print("Test Accuracy: ",calculate_accuracy(A2,y))

Test_Prediction(w1,b1,w2,b2,X_test,Y_test)
✓ 0.1s
[8 0 9 ... 5 7 8]
Test Accuracy:  88.41111111111111
```

The training loss is as follows:

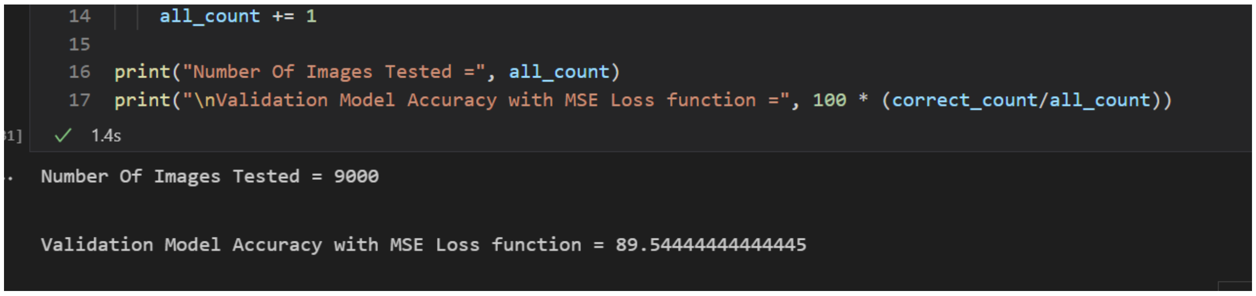


Also, to increase the training accuracy I have used batches of different sizes and then compared whether the loss is decreasing or not.



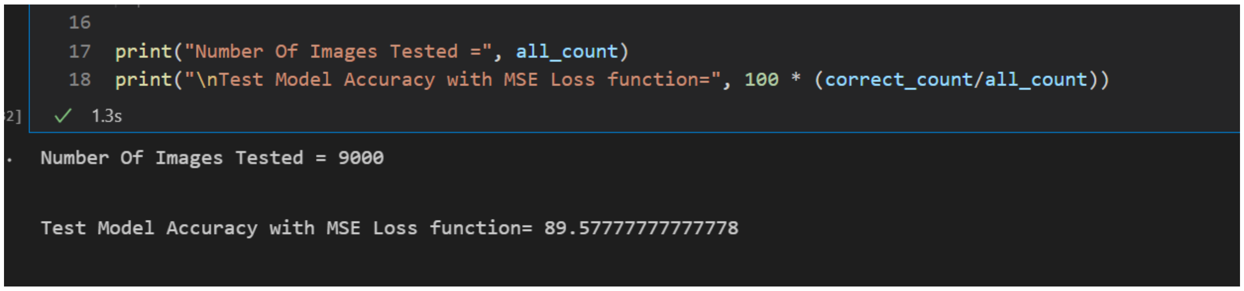
For python code with pytorch following is the output: -

1) Validation Accuracy



```
            all_count += 1

    print("Number Of Images Tested =", all_count)
    print("\nValidation Model Accuracy with MSE Loss function =", 100 * (correct_count/all_count))
 ✓ 1.4s
Number Of Images Tested = 9000

Validation Model Accuracy with MSE Loss function = 89.54444444444445
```

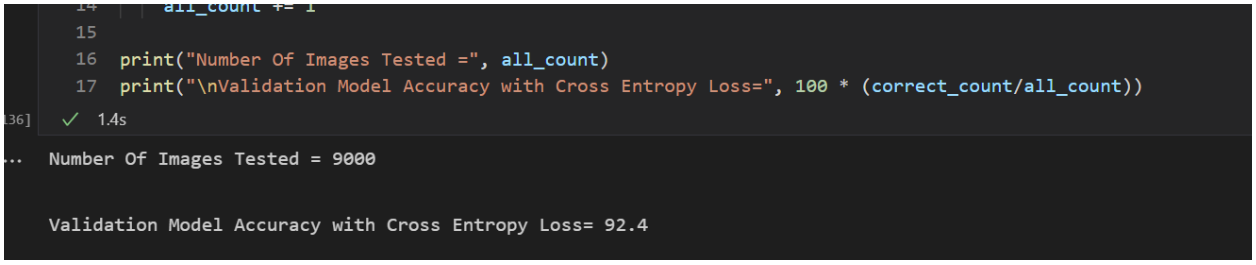
2) Test Accuracy



```

    print("Number Of Images Tested =", all_count)
    print("\nTest Model Accuracy with MSE Loss function=", 100 * (correct_count/all_count))
 ✓ 1.3s
Number Of Images Tested = 9000

Test Model Accuracy with MSE Loss function= 89.57777777777778
```

For increasing the accuracy, I have tried implementing the Cross Entropy Loss in the pytorch model to understand whether the accuracy is increasing or not. I was able to observe that the accuracy has increased.

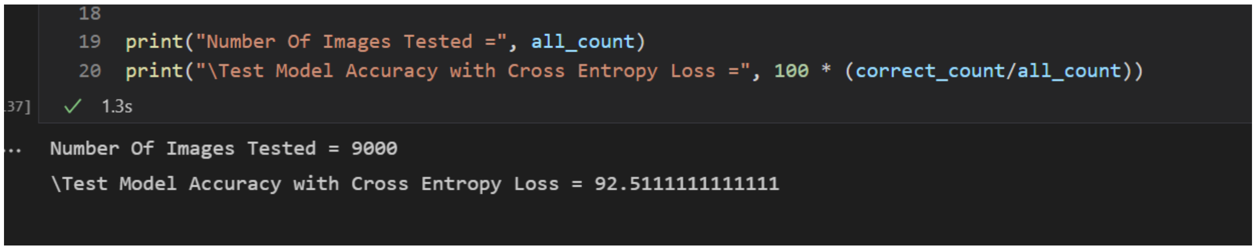
1) Validation Accuracy



```
    all_count += 1

print("Number Of Images Tested =", all_count)
print("\nValidation Model Accuracy with Cross Entropy Loss=", 100 * (correct_count/all_count))
```

```
Number Of Images Tested = 9000

Validation Model Accuracy with Cross Entropy Loss= 92.4
```

2) Test Accuracy

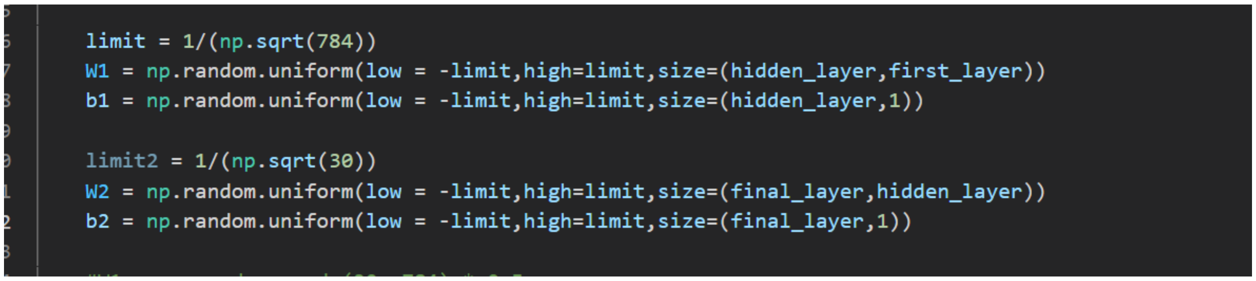


```

print("Number Of Images Tested =", all_count)
print("\Test Model Accuracy with Cross Entropy Loss =", 100 * (correct_count/all_count))
```

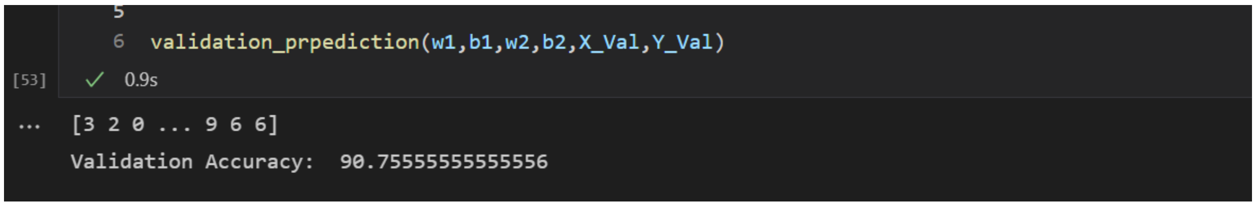
```
Number Of Images Tested = 9000
\Test Model Accuracy with Cross Entropy Loss = 92.5111111111111
```

Ques 4)

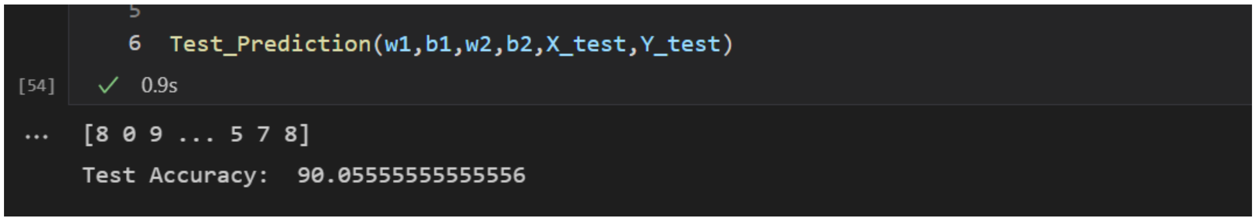
➔ To increase accuracy most of the modification I have tried to explain earlier. One of the changes to increase accuracy which I did was to initialize weight using different method. In the earlier version I had initialized weight directly or rather based on the size of the input. To increase the output accuracy I had initialized the weights using np.random.uniform function to uniformly distribute the weights across all the inputs.



```
limit = 1/(np.sqrt(784))
W1 = np.random.uniform(low = -limit,high=limit,size=(hidden_layer,first_layer))
b1 = np.random.uniform(low = -limit,high=limit,size=(hidden_layer,1))

limit2 = 1/(np.sqrt(30))
W2 = np.random.uniform(low = -limit,high=limit,size=(final_layer,hidden_layer))
b2 = np.random.uniform(low = -limit,high=limit,size=(final_layer,1))
```



```
validation_prpediction(w1,b1,w2,b2,X_Val,Y_Val)
[53] ✓ 0.9s

... [3 2 0 ... 9 6 6]
    Validation Accuracy:  90.75555555555556
```



```
Test_Prediction(w1,b1,w2,b2,X_test,Y_test)
[54] ✓ 0.9s

... [8 0 9 ... 5 7 8]
    Test Accuracy:  90.05555555555556
```

Code Link

1) Google Drive Link