1. Perceptron Training Algorithm

Q1.a:

Pre-processing:
Inputs are given to the implemented gates in the form of 1 and 0.

Assumptions:
The bias input of 1 (always) is assumed to be the third input to the perceptron andis provided
along with the other inputs. To check convergence the whole training process is run for a certain
number of epochs (we took 10 in our case). Is the

Approach:
First of all the weights were initialized with ones (in all cases). For each of the input instances the weights are updated using the PTA weight update formula:
$w = w + yx$
Here, y = converted label (1 and -1, not 1 and 0)
Weights are updated for the same instance until the output becomes
correct. Then The Same is done for (10 in our case).
Also, for drawing the decision boundary the weights at each update are stored in an array.

Results:
(i) For AND, with weights initiated with ones it took a lot of **17** weight updates for the perceptron to get trained.
For OR, it took a total of **6** weight updates.
(ii) For NOT too, it took a total of **6** weight updates for the perceptron to get trained.

Q1.b:

Results:
The decision boundaries learnt during each step for all the implemented gates. We can see in code.

Q1.c:

Results:
We observed that the perceptron was not able to learn the boundary because with the increase in the number of epochs the weight updates keeps on increasing it.
never stopped, unlike the earlier (AND, OR, and NOT) cases where the update steps remained

the same even though we increased the epoch. It means the perceptron was not converging and
was not able to learn the required decision boundary.
Also, we found that the weights started to repeat after a certain number of update steps. It took a total of 54 weight updates to prove that perceptron is not converging because of the weights of steps.

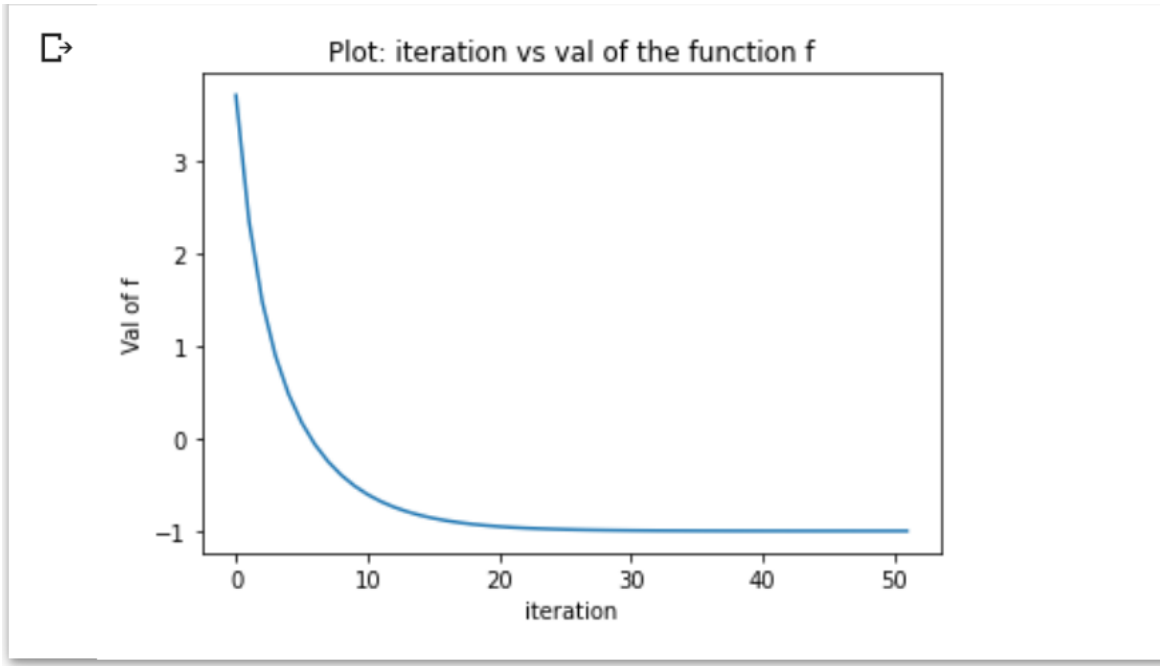
2. Gradient Descent

a) gradient descent function

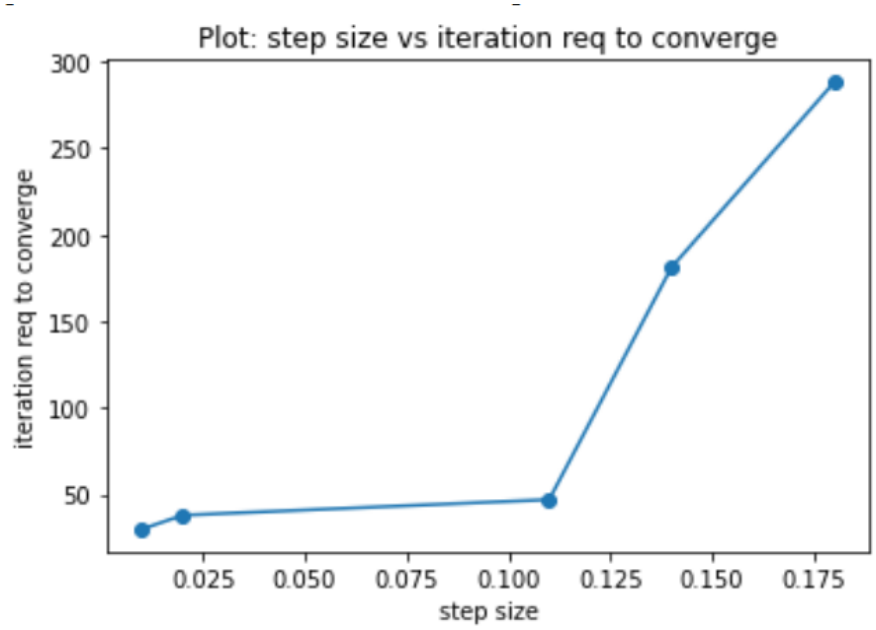
```
#defining gradient descent algorithm with parameters
def gradient_descent(func, grad, x0, Y, step_size, tolerence=1e-3):
  """returning
           x-> optimal point
           last_iter -> total iterations to converge
           fval_arr -> array of f values at each iteration
  """

  last_iter=0 #max iteration
  epoch=0
  fval_arr = [] #arr of f values at each iteration
  while(True):
    gradient = grad(x0)
    x = x0 - step_size*gradient

    # print("Iteration{}, x1={}, x2={}, f(x1,x2)={}".format(epoch, x[0].item(), x[1].item(), func(x,Y)))
    fval_arr.append(func(x,Y))
    #condition for convergence of the algorithm
    if np.linalg.norm(x-x0)/np.linalg.norm(x0)<tolerence:
      last_iter=epoch
      break
    x0 = x
    epoch += 1

  return x, last_iter, fval_arr
```

b) the number of iterations required for convergence is 52

c) Plot : Iteration vs value of function f(x1,x2)



d) Plot : step size vs iteration req to converge



When step size is 0.01 then iterations required to converge is 30.

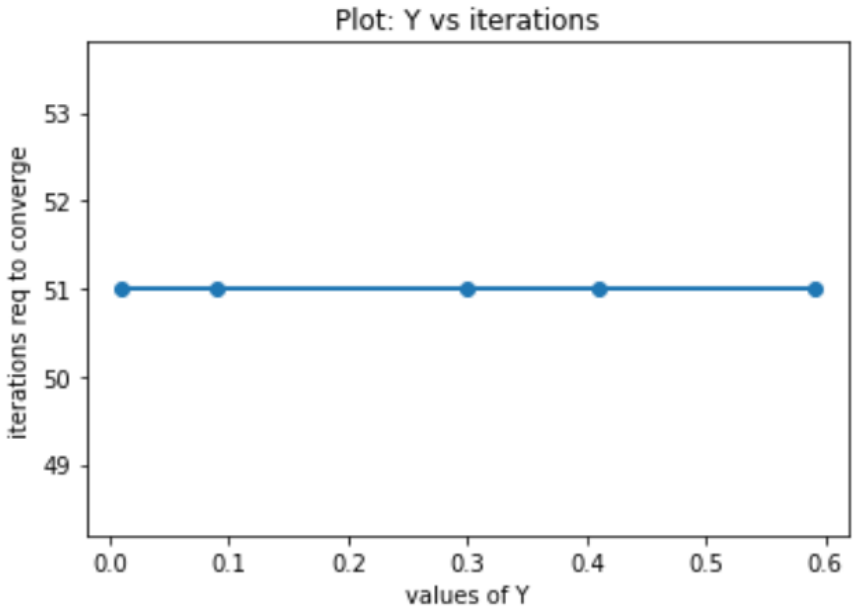
When step size is 0.02 then iterations required to converge is 38.

When step size is 0.11 then iterations required to converge is 47.

When step size is 0.14 then iterations required to converge is 181.

When step size is 0.18 then iterations required to converge is 288.

e) Plot : value Y vs Iterations



When the value of Y is 0.09 then iterations required to converge is 52.

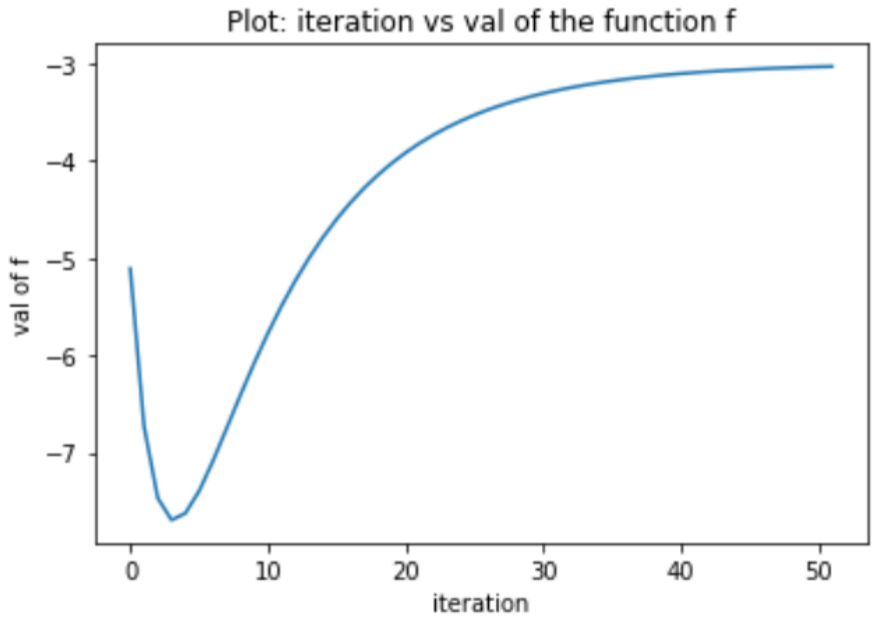
When the value of Y is 0.3 then iterations required to converge is 52.

When the value of Y is 0.59 then iterations required to converge is 52.

When the value of Y is 0.01 then iterations required to converge is 52.

When the value of Y is 0.41 then iterations required to converge is 52.

f) Plot : Iteration vs value of the function f(x1,x2)



Observation :
when the Y=-1 value of the function f(x1,x2) first increases then decreases as the number of iterations increases and on the other hand when Y=1 the graph decreases only as iteration increases.

3. Multi Layer Perceptron

Preprocessing:
The data is preprocessed in the sense that the Train set is split into a 70-30 ratio of train and validation.

Hyperparameters:
The neural network takes in as hyperparameters -
a) Activation Function (sigmoid/relu),
b) Learning Rate,
c) Epochs,
d) Number of hidden layers,
e) Number of neurons in each hidden layer

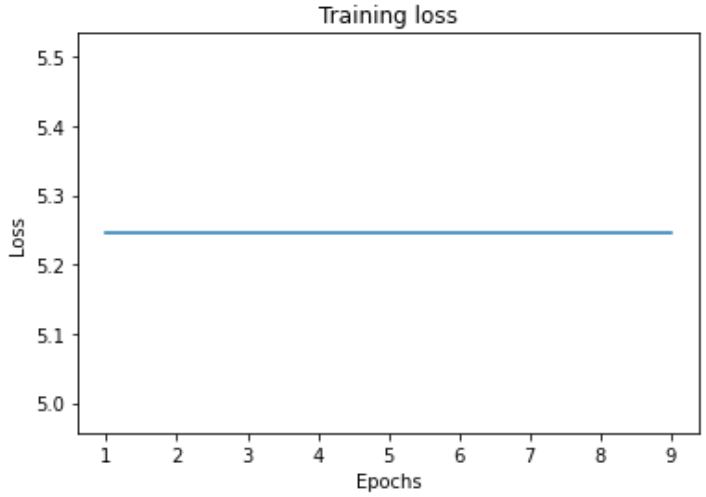
Helper Functions:
Helper Functions include-

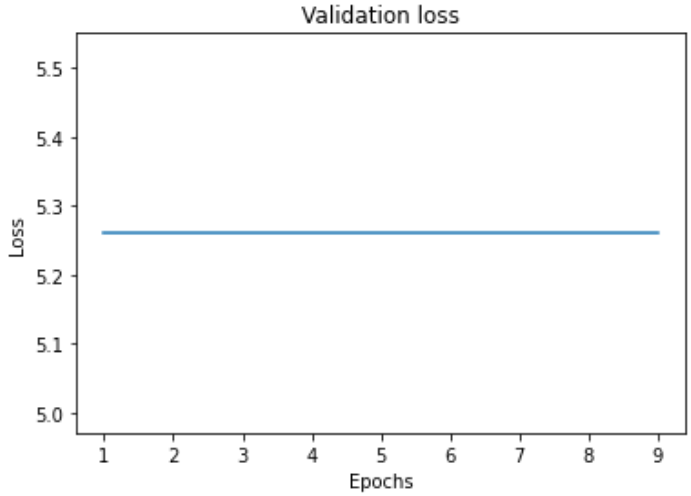
a) Activation functions (for output and for derivatives)
b) Calculation for gradients

c) Calculation for Loss and Accuracy
d) Forward pass for data inputs
e) Backpropagation and weight updation

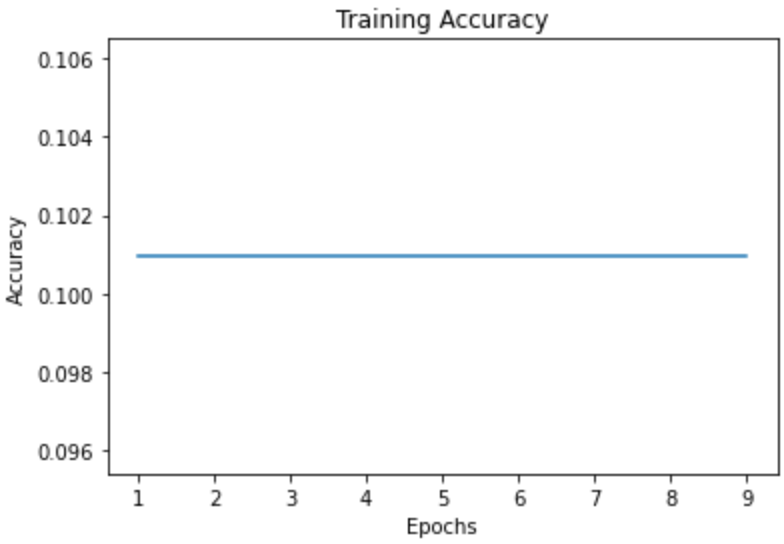
Results: Training and Validation loss remain almost constant for a small number of epochs, as training is slow and similar for a lesser number of trials of the samples.

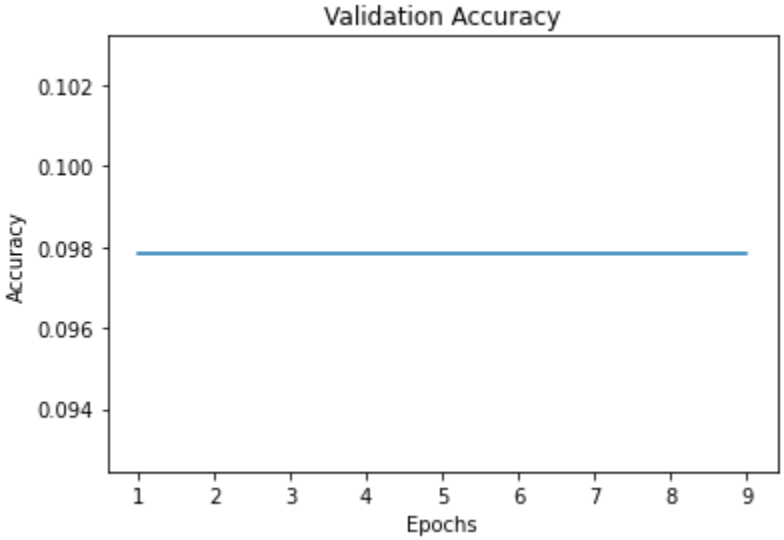
Training and Validation Loss vs Epochs;-





Training and Validation Accuracies vs Epochs:-


Training Accuracy
0.106
0.104
0.102
0.100
0.098
0.096
Accuracy
1
2
3
4
5
6
7
8
9
Epochs


Validation Accuracy
0.102
0.100
0.098
0.096
0.094
Accuracy
1
2
3
4
5
6
7
8
9
Epochs