DL Assignment 1

DRISHYA UNIYAL	MT21119
PRASHANT SHARMA	MT21227

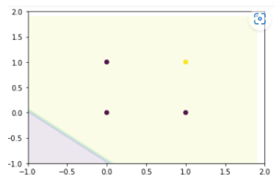
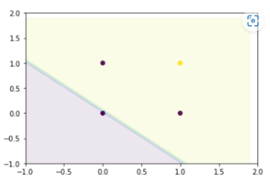
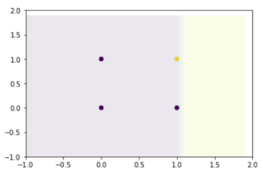
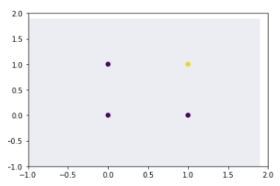
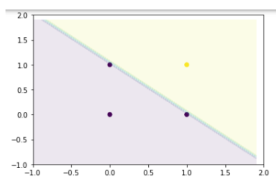
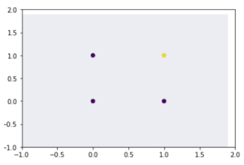
1.
Implement a Perceptron Training Algorithm (PTA) [15 Marks]
PTA updates weights when a mistake is made (DL Intro Slides 35). Follow the basic algorithm outlined in class, we require you to write a code and show the results for:
a**. Calculate the number of steps (i.e. weight updates) necessary for convergence for the following operations using: [2*3] marks**
i. Two variables: AND
ii. Two variables: OR
iii. One variable: NOT

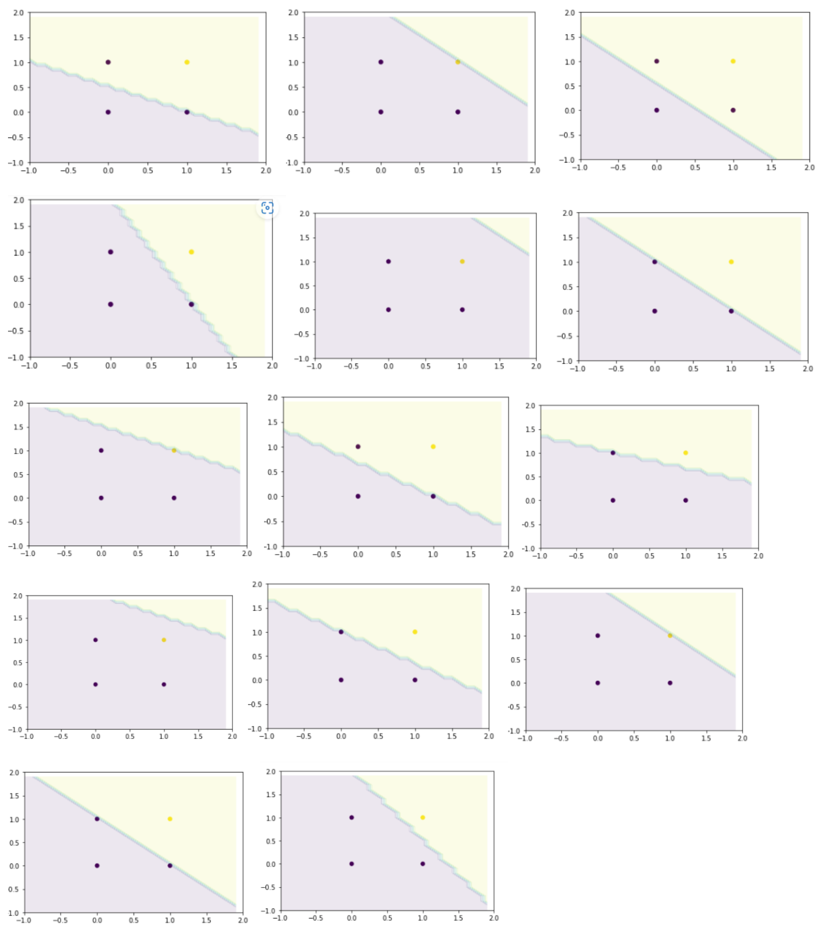
Results:
Perceptron trained from scratch

AND	20
OR	9
NOT	2

b. **Draw the decision boundary at each step of learning for all 3 operations (AND, OR, NOT).** [2*3] marks

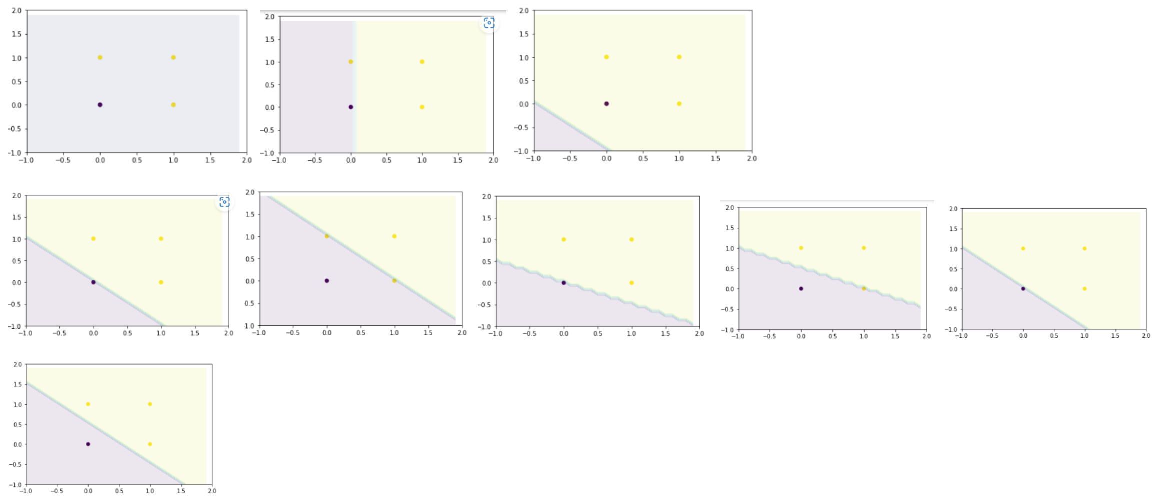
Results:
AND



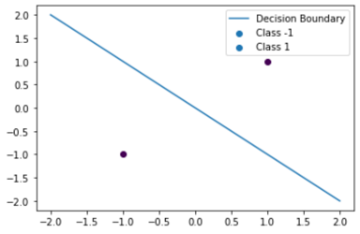
```
Number of steps necessary for convergence:  20
Best weight vector for AND function:  [3. 2.]
Best bias for AND function:  -4.0
```

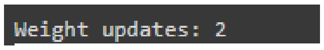
OR



```
Number of steps necessary for convergence:  9
Best weight vector for OR function:  [2. 2.]
Best bias for OR function:  -1.0
```

NOT



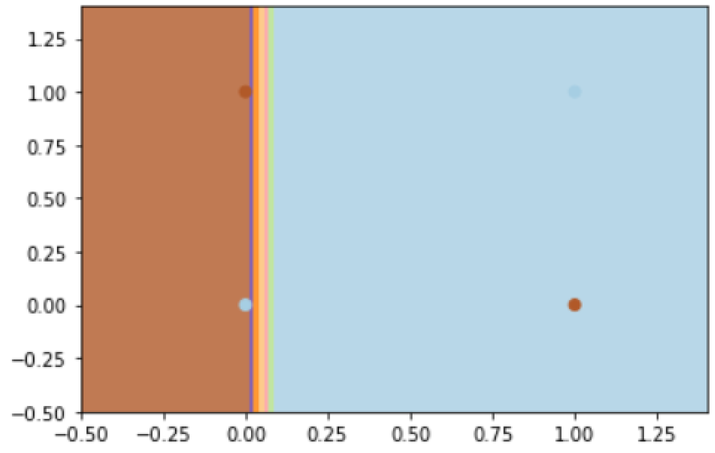


```
Weight updates: 2
```

Note: As the code runs a number of times the converge steps may differ but not very drastically!

c. **Theoretically, PTA can only converge on linearly separable data, as observed in 1a. Demonstrate via code that PTA cannot compute XOR operation. How many steps do you need to take to prove it? You can either prove this by showing the output or printing the decision boundary at each step.** [3] marks

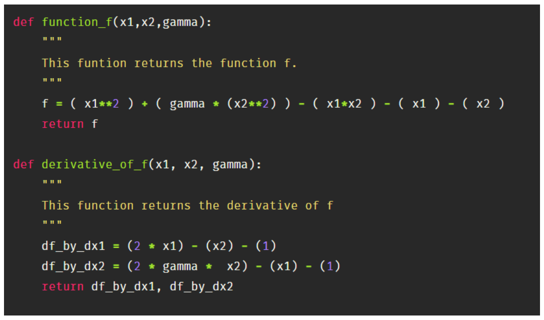
- No matter how many epochs we run, the weights and biases will never converge, so the number of steps can't be said as a single perceptron never works and never converges for XOR.
- The code ran for 10 epochs.



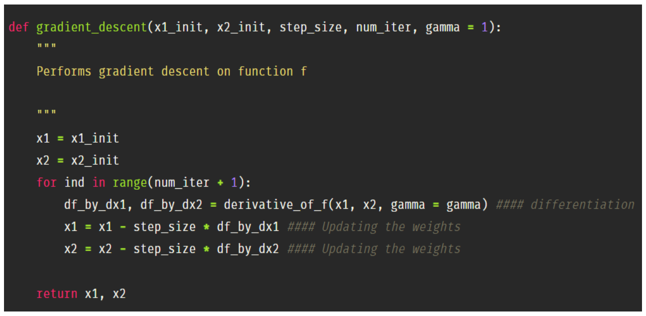
2.
Implement Gradient Descent Algorithm from scratch [25 marks]
Consider the following optimization problem:-
Min x1, x2 f(x1, x2) = x12 + γx22 − x1*x2 − x1 − x2
If we assume γ = 1 for the parts [a-d], then :

a. Write a function that implements the gradient descent algorithm (its subparts will involve differentiation and weight updation). [4] marks

- Below are the code snippets for functions writen to implements gradient descent algorithm, which involves tasks like differentiation, weight upgation, etc.



```python
def function_f(x1,x2,gamma):
    """
    This funtion returns the function f.
    """
    f = ( x1**2 ) + ( gamma * (x2**2) ) - ( x1*x2 ) - ( x1 ) - ( x2 )
    return f

def derivative_of_f(x1, x2, gamma):
    """
    This function returns the derivative of f
    """
    df_by_dx1 = (2 * x1) - (x2) - (1)
    df_by_dx2 = (2 * gamma *  x2) - (x1) - (1)
    return df_by_dx1, df_by_dx2
```



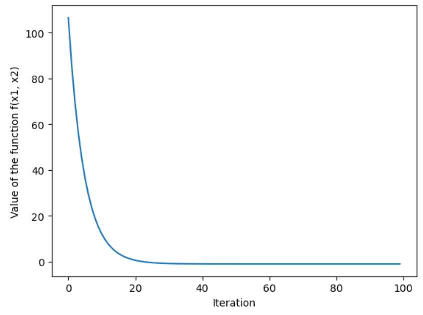
```python
def gradient_descent(x1_init, x2_init, step_size, num_iter, gamma = 1):
    """
    Performs gradient descent on function f

    """
    x1 = x1_init
    x2 = x2_init
    for ind in range(num_iter + 1):
        df_by_dx1, df_by_dx2 = derivative_of_f(x1, x2, gamma = gamma) #### differentiation
        x1 = x1 - step_size * df_by_dx1 #### Updating the weights
        x2 = x2 - step_size * df_by_dx2 #### Updating the weights

    return x1, x2
```

b. Using some initial values of x1 , x2 and step size = 0.1 as initial conditions, report the number of iterations required for convergence. [2.5] marks

- We used the initial values x1 = 13, and x2 = 12, and step size as 0.1 as specified.
- The number of iterations it took to converge is 134.

c. Plot the iteration v/s value of the function f in that iteration for the above setup. (x-axis = iteration no., y-axis = value of the function f in that iteration). [2.5] marks

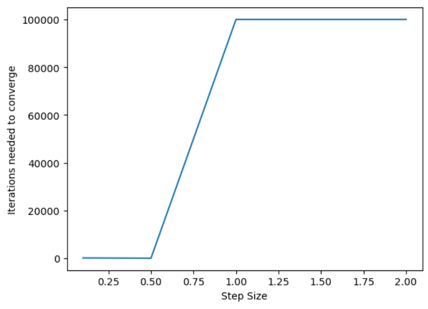
- We chose step size as 0.1 and ran the algorithm for 100 iterations.
- Below is the plot which shows the value of the function on Y-axis and step size on X-axis.



d. Arbitrarily choose 5 different values of step size in the interval [0,2]. Find out the number of iterations required to converge for each step size and plot step size v/s iterations graph (x-axis = step size, y-axis = iterations required to converge). You must keep the initial condition of x1 and x2 the same in all the cases. [6] marks

- *Configuration:* We ran the algorithm for 100000 iterations for step sizes = [0.1, 0.5, 1, 1.5, 2. Iterations needed for convergence is shown on the Y-axis, and step sizes are shown on X-axis.

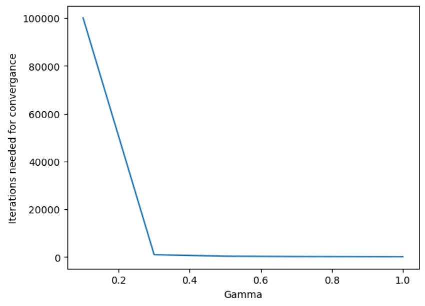
- *Observation:* For step_size >1; the convergance does not happen even at 100000 iterations, which indicates it is diverging beyond step_size 1.

- 

e. Arbitrarily choose 5 different values of γ in the interval [0,1] and find out the number of iterations required to converge for each value of γ. Plot γ v/s iterations graph (x-axis = γ, y-axis = iterations required to converge). You must keep the initial condition of x1 and x2 the same in all the cases. [6] marks

- *Configuration:* We ran the algorithm for 100000 iterations with step_size = 0.1, and the different values of gamma = [0.1, 0.3, 0.5, 0.7, 1].

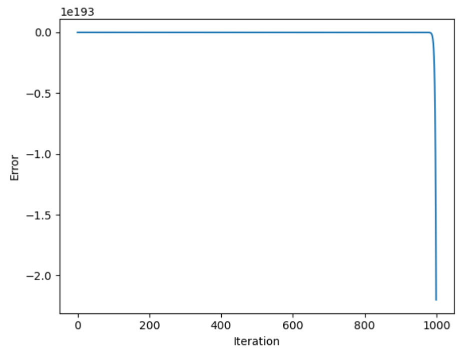
- *Observation:* For gamma < 0.3 the algorithm is diverging and for gamma > 0.3 it converges very quicky.



f. Run your gradient descent implementation for γ = -1 and plot Iteration v/s value of the function f in that iteration graph (x-axis = Iteration number, y-axis = value of the function f in that iteration). What do you observe? Can you think of an explanation for the observed behavior? ...Hint: think about the kind of curve a gradient descent draws. [2.5 marks + 1.5 marks]

- *Configuration:* We ran the algorithm for 1000 iterations, with step size of 0, and gamma as = -1.

- *Observation:* When gamma is negative, the function is not a convex function, and hence it may have multiple local minimas. The algorithm may get stuck in a local minimum instead of reaching the global minimum. The above plot shows this behavior as the value of the function oscillates around a certain value instead of continuously decreasing.



3. **Implement a Multilayer Perceptron from scratch [25 marks]**
Implement a modular Deep Learning Toolkit for training a multi-layer perceptron on the given Fashion-MNIST dataset.
Create the neural network using the descriptions given below. Implement forward propagation and backward propagation from scratch on the constructed neural network. You are required to create the toolkit.py file.
Network Structure: Input 784, output 10. You are free to use any number of hidden layers (>=1) (FNN) with any number of neurons. [4 for forward + 8 for backward] marks. After choosing the appropriate sizes of hidden layers, learning rate, and other hyperparameters. Analyze the model and report results for various combinations of the following:

a. **Activation**: Try using the activation functions for all hidden layers as either Sigmoid or ReLU. The output layer will always have Softmax activation. For each activation and its differential, write your implementation of it. [4Marks]

b. **Weight Initialization:** Try using the initial functions for all layers as either all zeros or Normal/Gaussian. [4 marks]

c. For the **4 combinations** of activation and weight initialization you try above, generate plots for [5 marks]
○ **Loss plot** - Training Loss and Validation Loss V/s Epochs.
○ **Accuracy plot** - Training Accuracy, Validation Accuracy V/s Epochs)
○ Analyze and Explain the plots obtained

Output:

- Relu/Sigmoid used for hidden layers
- Softmax for output layers.
- Zero/Normal Weight Initialization
- Code from scratch defining individual functions for the above.
- Different functions made for sigmoid, relu, and softmax.

```
def sigmoid(self, z):
    z = np.clip(z, -700, 700) # prevent overflow
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(self, a):
    return a * (1 - a)

def relu(self, z):
    return np.maximum(0, z)

def relu_derivative(self, z):
    return (z > 0) * 1.0

def softmax(self, x):
    x = np.array(x)
    x = np.where(np.isnan(x), -sys.float_info.max, x)
    x = np.where(np.isinf(x), sys.float_info.max, x)
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=1, keepdims=True)
```

The 4 combinations made are

Sigmoid + Zero Initialization
Sigmoid + Normal Initialization
Relu + Zero Initialization
Relu + Normal Initialization

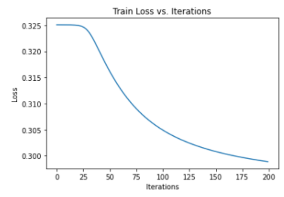
HyperParameters

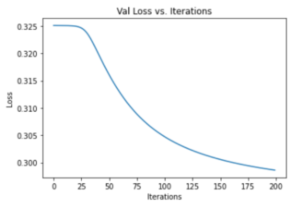
hidden_size	128
Learning Rate	0.01 (Zero WI), 0,1 (Normam WI)
Epochs	200
output_size	10

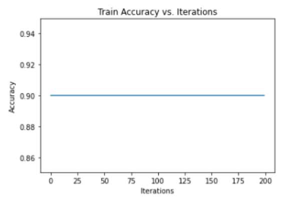
Sigmoid + Zero Initialization	90
Relu+ Zero Initialization	90
Sigmoid + Normal Initialization	87.019
Relu + Normal Initialization	90

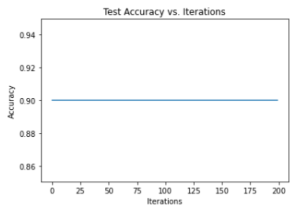
Sigmoid + Zero Initialization

```
Maximum accuracy on test data:  0.9
```



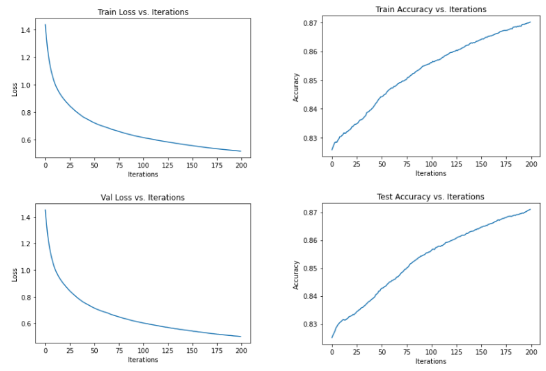




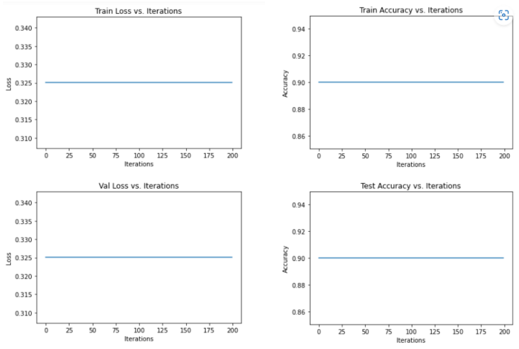


Sigmoid + Normal Initialization

```
Maximum accuracy on test data:  0.87019
```

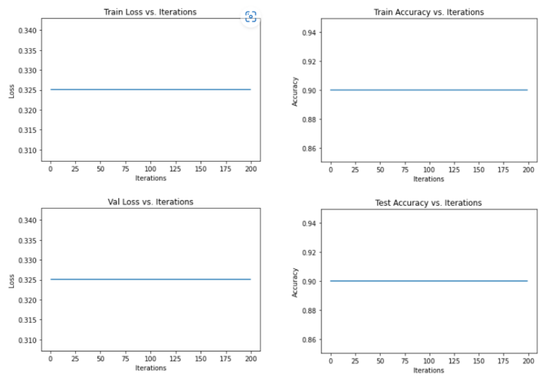


Relu + Zero Initialization



Relu + Normal Initialization

```
Maximum accuracy on test data:  0.9
```



The models have been saved using pickle.

```python
with open("model1.pickle", "wb") as file1:
    pickle.dump(model1, file1)
```

Toolkit.py files contains the basic structure for the model.

Graph Interpretations:

1. Seeing the plots, the data seems to overfit and hence the straight line.
2. This can be tackled using different techniques.
3. The result for sigmoid + normal seems to be converging as the other overfit.

Contribution:

Drishya - Q1 and Q3.
Prashant - Q1 part 3 and Q2
Report was made equally!