CSE 641 | Deep Learning

Assignment 1

Part 1

PTA

- The parameters to be fed for the model are:
  - **X:** Input Matrix
  - **y:** Labels corresponding to Input Matrix
  - **numWeights:** Number of features in the input matrix
  - **threshold:** Threshold value to be considered while implementing the algorithm
  - **modelName:** Name of the model to be used while loading/saving the model
  - **weights:** We can feed in the predefined weights to the system. Default value for the same is Normal Distribution with mean = 0 and std = 1
  - **maxIters:** Maximum number of epochs to run the model for. Default value is 50
  - **loadModel:** Whether to load the model or not. If set true, it is assumed that there exist a compatible model in the same directory with modelName is the model's name.
- For training purposes, for "and" and "or" gates, normally generated weights with mean = 0 and standard deviation = 1 were used.
- For not gates, weights were predefined as [3, 2]
- For not gates, weights were predefined as [3, 2, 1]
- For and gate, it took 12 iterations to converge.
- For or gate, the model took 3 iterations to converge.
- For not gate, the model took 9 iterations to converge.
- For xor gate, the model didn't converge even after completion to 50 iterations.
- After 4 epochs, a cycle of weights started in the model, which repeated themselves after every 4 updates.

Madaline

Parameters for Madaline:

- **N_layers:** Number of layers in the model (Includes input layer, all hidden layers and output layer)
- **Layer_sizes:** Number of neurons in each layer (Includes input layer, all hidden layers and output layer)
- **learning_rate:** Learning rate for the model
- **weight_init:** Initial weight initializations
    - Available weight initializations:
        - random
        - normal
        - zero
- **num_epochs:** Number of epochs to run the model (Default value =200)
- **threshold**: Threshold value for the model

Helper Functions:

- Functions in madaline class:
    - **Zero_weight_init(), random_weight_init(), normal_weight_init():** Functions to initialize weights
    - **initialize():** Calls weight initialization function according to parameter weight_init and initializes bias
    - **forward_propagation():** Take input, weight, bias and the layer from which forward propagation should start. It returns final output as well as z (input sums) and y (output at each layer after applying threshold)
    - **checkConvergence():** To check convergence of model according to error limit defined
    - **fit():** Trains the model
    - **predict():** Predict output using weights and bias given by the model
    - **error():** Returns no of misclassified samples
    - **accuracy():** Returns no of correct samples/ total samples percentage
    - **saveModel():** Save model in file using pickle: `'fileModel.pkl'`
- Other functions:
    - **createDataset():** Creates dataset for f(x1,x2) where colored regions are 1, rest are 0
    - **loadModel():** Loads model from the given filename using pickle.

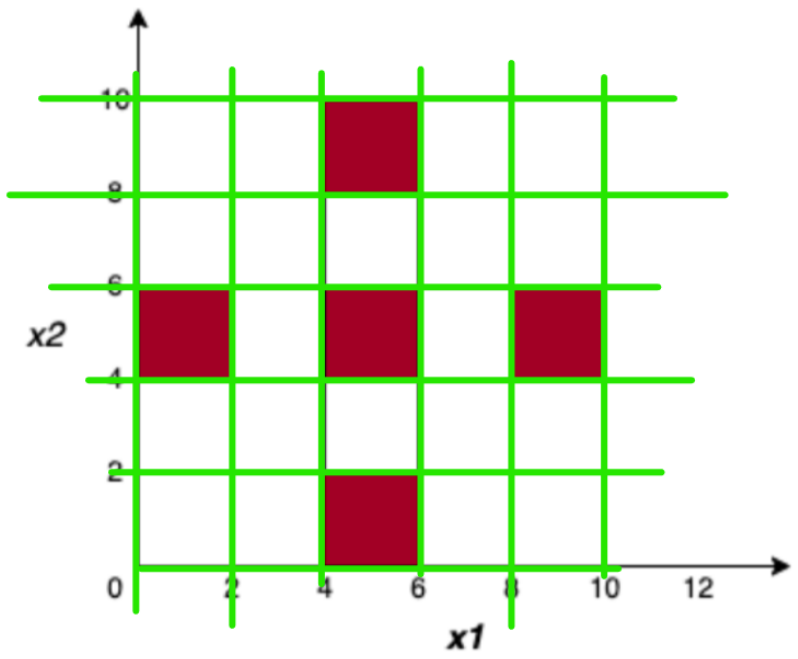
In the interest of time, we have assumed the model to have been converged if the error is less than 5%

Please note that the data points were generated in such a way that both the classes get almost comparable distributions.
Out of 121 samples, 51 have class label 1 (~43%) and 70 have label 0 (~57%).

Output:
Accuracy achieved: 95.87%
Number of misclassified samples (out of 121) = 5

Number of neurons needed: 12+5+1 (Hidden Layer 1 + Hidden Layer 2 + Output Layer)
Total neurons in hidden layer = 17
Total neurons including output layer = 18

b.

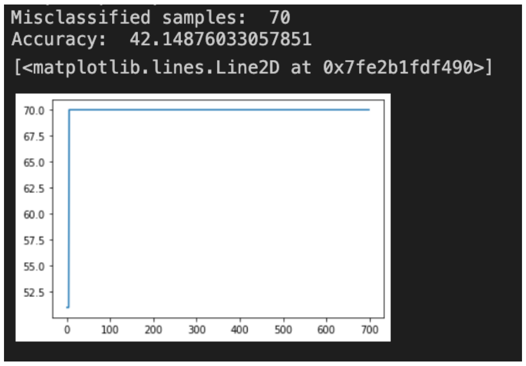


We **cannot** compute f(x1,x2) in <=2 neurons.
Each neuron in the model gives a single decision boundary that separates the data into 2 parts. Here f(x1,x2) as shown in the figure, needs at least **12 decision boundaries** (6 vertical and 6 horizontal together) to separate the red part from white. Lines x1=4, x1=6, x2=4 and x2=6 are able to cover all the edges for the middle red square, both vertical edges for top and bottom squares while both horizontal edges for left and right squares. x1=0 and x2=2 are needed for

vertical edges for left square, x1=8 and x1=10 are needed for right square, x2=0 and x2=2 are required for bottom square and x2=8 and x2=10 for top square. Thus in total 12 decision boundaries. Now these boundaries need to be combined for each square. Thus **5 more neurons** (5 squares) are needed to combine these decision boundaries in the next layer. Another additional layer is required in which **1 neuron** will be there which will combine all these 5 squares into 1 red region.

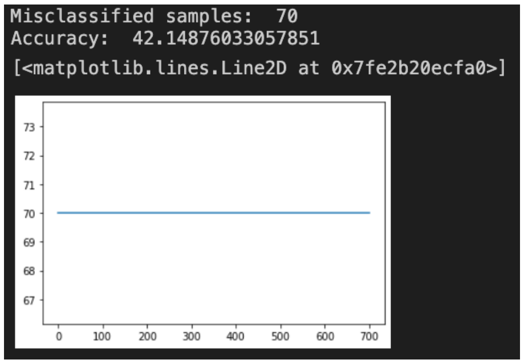
The same is also shown with the help of code. 2 different configurations were used:

1. 2 Layers with 1 neuron in each:



```
Misclassified samples:  70
Accuracy:  42.14876033057851
[<matplotlib.lines.Line2D at 0x7fe2b1fdf490>]
```

2. 1 Layer with 2 neurons:



```
Misclassified samples:  70
Accuracy:  42.14876033057851
[<matplotlib.lines.Line2D at 0x7fe2b20ecfa0>]
```

Final architecture required:
12 + 5: Hidden layers
12+5+1: Hidden layers + Output layer

Part 2

MLP

- The toolkit.py file consists of the neural network implementation and the functions to plot the loss vs epoch plots and the accuracy vs epoch plots.
- The data was normalized using the standard scalar function available in the sklearn library.
- Helper functions to draw various graphs were used.
- The neural network takes the following parameters:
- **N_inputs:** The number of inputs to the network/ Number of neurons in the input layer
- **N_outputs:** The number of outputs/ Number of neurons in the output layer
- **N_layers:** Number of hidden layers in the network(Default value of 2)
- **Layer_sizes:** Number of neurons in each hidden layer, (Default value = [10, 5])
- **activation:** The activation function to be used in the layers
    - Available activation functions:
        - sigmoid
        - tanh
        - ReLU
        - Softmax
    - Default activation function: sigmoid
- **learning_rate:** Learning rate for the network (defualt = 0.1)
- **weight_init:** Initial weight initializations
    - Available weight initializations:
        - random
        - normal
        - zero
    - Default weight initialization: random
- **batch_size:** Batch size for the network (default value = 1)
- **num_epochs:** Number of epochs to run the model for (Default value =200)
- **backpropogation:** Backpropogation algorithm to use
    - Available backpropogation algorithms:
        - Gradient Descent (gd)
        - Gradient Descent with Momentum (momentum)
        - Nesterov Accelerated Gradient (nag)
        - Adagrad (adagrad)
        - RMSProp (rmsprop)
        - Adam (adam)
    - Default backpropogation algorithm: gd,
- **beta:** Beta value for the backpropogation algorithms (Default = 0.9)
- **gamma:** Gamma values for the backpropogation algorithms (Default = 0.999)
- **modelName:** Name of the model to be used while loading/saving the model

- **loadModel:** Whether to load the model or not. If set true, it is assumed that there exist a compatible model with modelName is the model's name (Default = False)
- **saveModel:** Whether to save the model or not. If set true it is assumed that modelName is not empty. (Default = True)

Various Models with different learning rates, different configurations and different activation functions and their performances for 150 epochs

Activation Function: ReLU

	Learning Rate = 0.1	Learning Rate = 0.01	Learning Rate = 0.001
Layers = [200, 50]	63.34%	38.08%	11.58%
Layers = [256]	**83.17%**	63.3%	49.57%
Layers = [100]	82.59%	54.83%	37.76%

Activation Function: Sigmoid

	Learning Rate = 0.1	Learning Rate = 0.01	Learning Rate = 0.001
Layers = [200, 50]	21.25%	10.04%	10%
Layers = [256]	73.92%	61.95%	21.75%
Layers = [100]	71.66%	50.73%	13.32%

Activation Function: Tanh

	Learning Rate = 0.1	Learning Rate = 0.01	Learning Rate = 0.001
Layers = [200, 50]	69.99%	55.5%	13.75%
Layers = [256]	83.15%	68.66%	53.38%
Layers = [100]	82.51%	60.52%	50.32%

Best Results obtained:

Activation Function: ReLU
Number of Hidden Layers: 1
Hidden Layer Sizes: 256
Learning Rate: 0.1

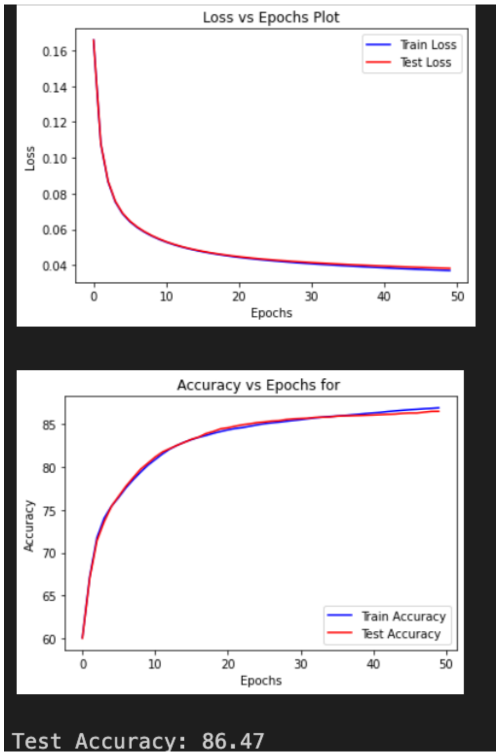
Results for Various Optimizers:

Optimizer	Accuracy on test data
Gradient Descent With Momentum	86.47%
Nesterov's Accelerated Gradient	86.48%
Adagrad	87.17%
RMSProp	89.73%
Adam	88.73%

Model Configurations for all of the above:

Activation Function: ReLU
Number of Hidden Layers: 1
Hidden Layer Sizes: 256
Learning Rate: 0.0001
Number of Epochs: 50

Analysis:



```
Test Accuracy: 86.47
```

Gradient Descent with Momentum

$m^{(t)} = \beta m^{(t-1)} + \eta \nabla J(w^{(t-1)})$
$w^{(t)} = w^{(t-1)} + m^{(t)}$
Accuracy: 86.47%

Gradient descent with momentum improves upon the network based on just gradient descent which had an accuracy of 83.1%. Momentum helps the model to get out of local minima and reach global minima. Here, first we step against the gradient and then move to the scaled version of the previous position given by the momentum.

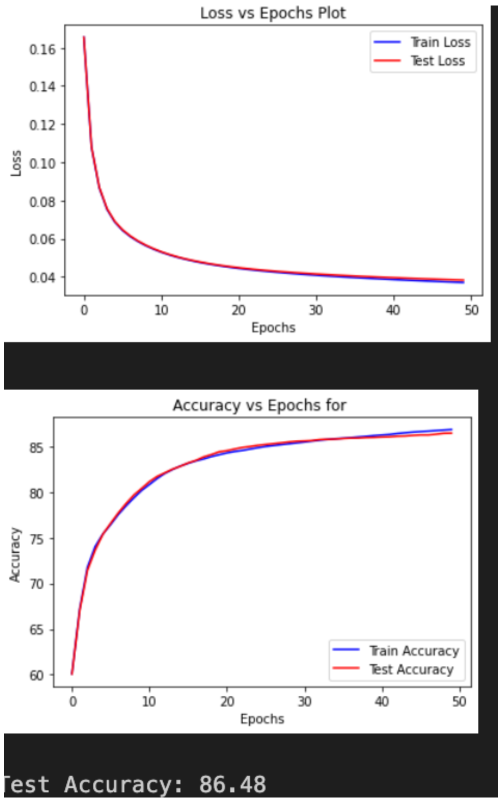
Nesterov's Accelerated Gradient

$m^{(t)} = \beta m^{(t-1)} + \eta \nabla J(w^{(t-1)} + \beta m^{(t-1)})$

$w^{(t)} = w^{(t-1)} + m^{(t)}$

Accuracy: 86.48%

In NAG, first we move on to scaled position and then step against the gradient. Here we got better accuracy than momentum



```
est Accuracy: 86.48
```

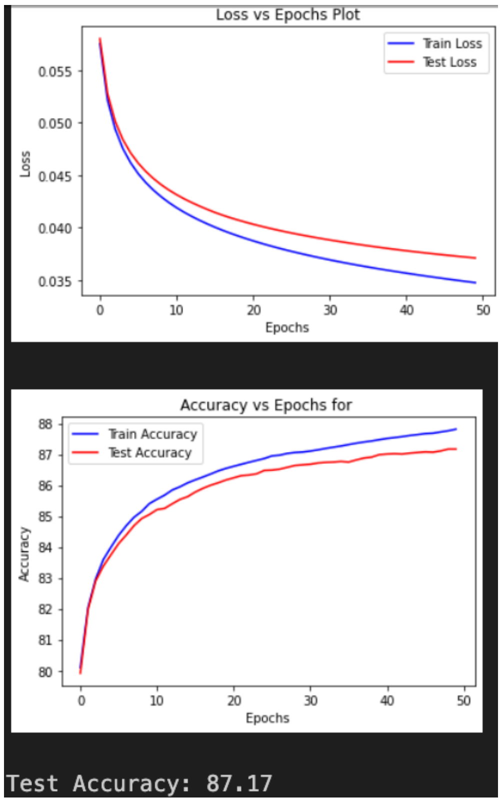
AdaGrad

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{G_i^{(t)} + \epsilon}} \nabla J(w_i^{(t)})$$

$$G_i^{(t)} = \sum_{\tau=1}^{t} \nabla J(w_i^{(\tau)}) \cdot \nabla J(w_i^{(\tau)})$$

Accuracy: 87.17%

AdaGrad has improved performance than both momentum and NAG. It even converged faster than both of the previous optimizers. It basically adapted its learning rate as the no of epochs increased.
But AdaGrad includes accumulating squared gradient which resulted into a large model file.



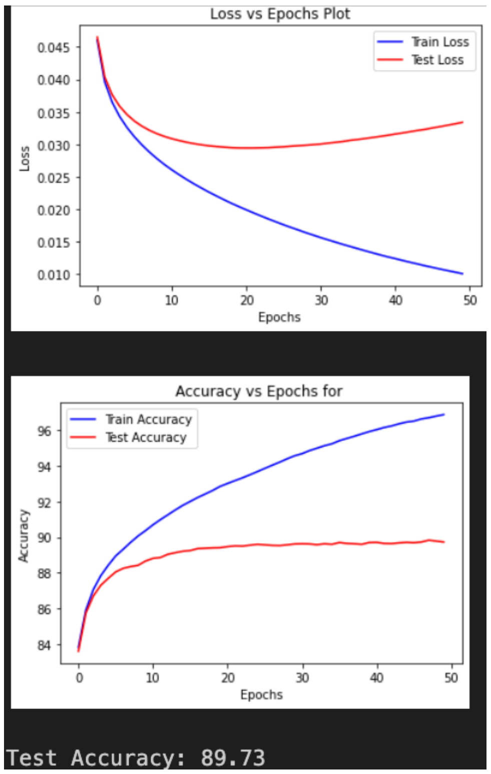
```
Test Accuracy: 87.17
```

RMSProp

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{E_i^{(t)} + \epsilon}} \nabla J(w_i^{(t)})$$

$$E_i^{(t)} = \gamma E_i^{(t-1)} + (1-\gamma)\nabla J(w_i^{(t)})^2$$

Accuracy: 89.73%
It handles the issue of accumulating squared gradient in adaGrad. It gives less weightage to previous gradients. Thus it has the best accuracy out of adaGrad, NAG and momentum and in the case of only 50 epochs it's even better than Adam.



```
Test Accuracy: 89.73
```

Adam

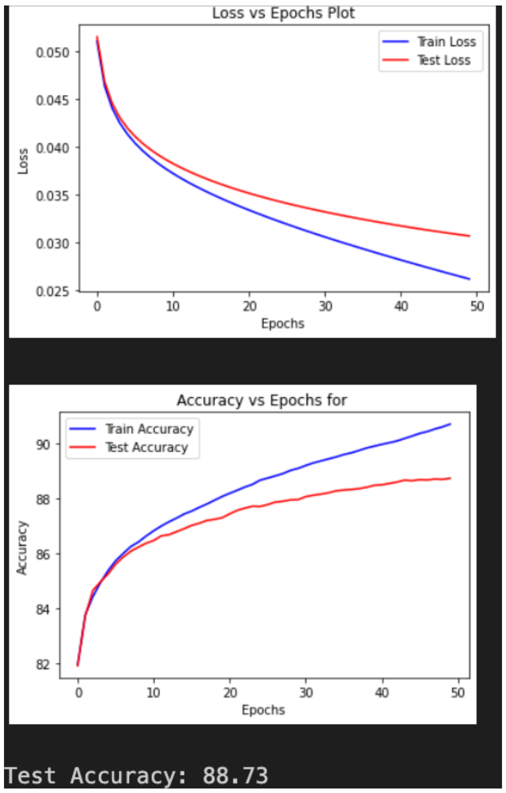
$$m^{(t)} = \beta m^{(t-1)} - \eta \nabla J(w^{(t)})$$

$$v^{(t)} = \gamma v^{(t-1)} + (1-\gamma)\nabla J(w^{(t)})^2$$

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta^t} \qquad \hat{v}^{(t)} = \frac{v^{(t)}}{1-\gamma^t}$$

$$w^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{\hat{v}^{(t)} + \epsilon}} \hat{m}^{(t)}$$

Accuracy: 88.73%
Adam combines RMSProp and momentum. In our case it couldn't converge in just 50 epochs (We didn't increase the epochs because we needed to compare all the optimizers in same architecture.). It has performed far better than NAG, AdaGrad and momentum. From figure, we can see that if more iterations would have been given it would have converged and greater accuracy would have been achieved.



```
Test Accuracy: 88.73
```

Contribution:
Both team members contributed equally on all the aspects of all the questions.