EXPERIMENT 2:- DOCUMENTATION

Delhi Technological University
Shahbad Daulatpur Village, Rohini, New Delhi, Delhi, 110042

(Department of Computer science and technology)

SUBMITTED BY:-

Aryan Gahlaut

2K22/CO/102

6th semester (3rd year)

SUBMITTED TO:-

Dr Anil Parihar

EXPERIMENT 2

Aim: To train ANN (Artificial Neural networks) to successfully classify non linearly separable data points first via single neuron and second via Multi layer neural network and to observe their behaviour.

Dataset used: The Moons dataset is a very popular one used in classification tasks in machine learning. It has two classes that form crescent-like shapes, or "moons," making it an excellent test case for algorithms that need to separate complex, non-linearly separable data. The dataset generally consists of two features, which represent coordinates in a 2D space, and a target variable, indicating the class of each point.
The dataset can be generated with certain parameters, including the number of samples, the noise level, and the shape of the moons, which allows for a flexible testing environment in experimenting with classification algorithms. The dataset is generally used to assess the performance of classification models, especially those with non-linear decision boundaries such as SVM and neural networks.

Platform used: Google colab
It is an end-to-end cloud-based solution for writing and executing the Python code from a user's web browser called Google Colab, or also known as Google Colaboratory. This feature allows the user to work interactively in the same environment as many popular libraries: TensorFlow, PyTorch, and NumPy, especially in demand applications such as machine learning, data analysis, and research.
One of the advantages of Google Colab is access to free computing resources: GPUs and TPUs, that may accelerate the computation significantly and is especially needed for training big models , provided the access to these GPUs and TPUs are limited for a limited amount of time.
The platform integrates Google Drive and supports saving and easy sharing of the work. It also supports Jupyter notebooks, which provide a rich interface that allows users to combine code, visualizations, and text in a single document, making it an excellent tool for collaboration and learning.

Libraries and framework used:

1) Torch: Torch is a deep learning framework that has been developed to build and train neural networks. It supports tools for tensor computation, automatic differentiation, and GPU acceleration**.**

2) torch.nn: A submodule of PyTorch that contains all the building blocks for constructing neural networks, from layers to loss functions and optimization algorithms**.**

3) matplotlib.pyplot: This is a plotting library to draw graphs and charts. Often, it's used for the visualization of data and model performance**.**

4) sklearn.model_selection.train_test_split: It is the function of Scikit-learn which splits the data into two; this splits data into training set and unseen data set which tests the model against it.

5) sklearn.preprocessing.StandardScaler: It standardizes features using zero-mean unit-variance scaling, an operation important to most machine learning algorithms.

6) sklearn.datasets.make_moons: It is a synthetic, 2D data-generating function for generating data distributed in the form of moons and is usually applied to classification models.

7) pandas: is an efficient data manipulation library primarily used in handling structured data that includes tables. Data cleaning, analysis, and transformation are some of the tools that it provides.

8) torch.utils.data.TensorDataset: A PyTorch utility for creating datasets from tensors, which then can be used with DataLoader for batching and shuffling during training.

9) torch.utils.data.DataLoader: PyTorch is a tool for loading batches of the data during training where the data optionally gets shuffled.

10) torch.nn.functional: PyTorch module that provides functional versions of many operations, including activation functions and loss functions, used in the definition of models.

11) numpy: Core library for numerical computing in Python, especially for working with arrays and matrices. It also provides a wide range of mathematical functions.

Implementation:

Importing necessary libraries and framework

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons
import pandas as pd
```

The pytorch framework is imported as torch . It provides functionalities to work with neural networks.
Matplotlib library is used to visualize the performance of both single neuron and multi layer neural network.
Train_test_split is imported to split the dataset into training and testing data.
StandardScaler is used as a preprocessing in the pipeline to standardize that feature in such a way that it has a mean of 0 and standard deviation of 1.
make _moon is used to produce the data points.
Pandas is imported as pd and it is used to organize data in tabular structure so that preprocessing and other operations can be performed easily on the data.

Defining the dataset

```
x_moon , y_moon=make_moons(n_samples=1500 , noise=0.1 , random_state=42)


x_moon=pd.DataFrame(x_moon)
x_moon.head()
```

There are 2 independent features (x_moon) and one target dependent feature (y_moon).
There are a total of 1500 such data points with noise ratio of 0.1 and random state of 42.
These two independent features are then converted to dataframe so that preprocessing can be done easily and efficiently.
Top 5 entries is the printed.

	0	1
0	0.981321	-0.499666
1	1.844094	0.072311
2	-0.060090	1.048577
3	-0.174089	1.099558
4	-0.464776	0.877356

```
y_moon=pd.DataFrame(y_moon)
y_moon.head()
```

Similarly , y_moon is also converted to dataframe and top 5 entries are also printed.

Splitting data and preprocessing

```
x_train ,x_test , y_train ,y_test=train_test_split(x_moon , y_moon ,
test_size=0.2 ,random_state=42,shuffle=True)    # splitting data into train and
test set
scaler=StandardScaler()                                                    #
to standardize the features in x_moon
x_train=pd.DataFrame(scaler.fit_transform(x_train),columns=x_train.columns)
# scaled data is then converted to dataframe
x_test=pd.DataFrame(scaler.transform(x_test),columns=x_test.columns)

x_train.head()
print('\n')
x_test.head()
```

The dataset is splitted into training and testing sample using train_test_split(). The test_size is kept 0.2. It means that about 20% of the dataset will be used for testing purpose and remaining 80% will be used for training purpose. X_train , x_test contains 2 features and y_train , y_test contains target dependent column. After splitting the dataset into training and testing set , scaler() is applied to them (StandardScaler()) to scale these independent features. After scaling the features , the scaled data is again converted to the dataframe .
On testing set only transformation is applied , not fitting , as the testing data will be used for prediction based on metrics of training set.

After all these the top entries of both train and test set are printed.

Note that the output feature is not scaled as it is categorical column. Scaling the feature would result in discrepancies in predictions and meaning of feature.

Converting train dataset into tensor dataset and grouping into batches

```
input_ds=torch.tensor(x_train.values , dtype=torch.float32)
label_ds=torch.tensor(y_train.values , dtype=torch.float32).view(-1,1)
from torch.utils.data import TensorDataset , DataLoader
import torch.nn.functional as F
loss_fn=F.binary_cross_entropy
train_ds=TensorDataset(input_ds,label_ds)
train_dl=DataLoader(train_ds,batch_size=50,shuffle=True
opt=torch.optim.SGD(model.parameters(),lr=1e-4)
```

X_train and y_train are converted to tensors by using torch.tensor() and reassigned to input_ds and label_ds.
Input_ds and label_ds are paired up together into trsin_ds respectively using TensorDataset.
Binary_Cross_entropy loss will be used since we are working for binary classification.
DataLoader is used to segregate the train_ds into groups of mini batches of fixed size of 50.
SGD is used as an optimizer with the learning rate of 1e-4.

Defining single layer neural network

```
class onelayernn(nn.Module):
  def __init__(self,input_size,hidden_size,output_size):
    super(onelayernn,self).__init__()

    self.layer1=nn.Linear(input_size,output_size)
    self.sigmoid=nn.Sigmoid()

  def forward(self,x):
    x=self.sigmoid(self.layer1(x))
    return x

input=2
hidden=0
output=1
model=onelayernn(input,hidden,output)
```

A simple neural network model named onelayernn is defined using PyTorch. This consists of only a single layer. This class inherits the nn.Module class and initializes a linear layer

of nn.Linear that maps the inputs of a given size, input_size, to an output size, output_size. Finally, it applies the sigmoid activation function, which maps the output in range between 0 and 1 [0,1], nn.Sigmoid, at the output of this linear transformation to introduce non-linearity. The forward method defines the data flow through the network, which is the input passing through the linear layer followed by the sigmoid activation. With input size 2, hidden size 0 which is not in use here, and output size set to 1, the model is instantiated.

Training single layer neuron

```
def train(model, train_dl,loss_fn,opt,epoch):
  for e in range(epoch):
    model.train()
    for x , y in train_dl:
      opt.zero_grad()
      output=model(x)
      loss=loss_fn(output,y)
      loss.backward()
      opt.step()
      if (e + 1) % 10 == 0:
            print(f"Epoch [{e+1}/{epoch}], Loss: {loss.item():.4f}")

train(model,train_dl,loss_fn,opt,10000)
```

The train function is used to train this single layered neural network model over a 10000s of epochs. Inside the function, the model is set in training mode using model.train(). The function then iterates over the training data in train_dl, which yields batches of inputs (x) and their corresponding labels (y). For every group of samples, the gradients of the optimizer are zeroed by using opt.zero_grad() and a forward pass is run over x and the model to yield the output. The loss is calculated using loss_fn as follows: a comparison of output yielded by the model with the true labels, after which loss gradients are calculated with loss.backward(), and opt step() uses these gradients to update the parameters of the model. Every 10 epochs, the loss value is printed to observe the progress of training. The function call train with 10,000 epochs to train the model, passing model, train_dl, loss_fn, and opt as arguments, which corresponds to the model, training data, loss function, and optimizer.

Streaming output truncated to the last 5000 lines.
Epoch [7920/10000], Loss: 0.2178
Epoch [7920/10000], Loss: 0.2003
Epoch [7920/10000], Loss: 0.2822
.

.
.
.
.
Epoch [10000/10000], Loss: 0.2131
Epoch [10000/10000], Loss: 0.3274
Epoch [10000/10000], Loss: 0.2363

Testing the single layer neural network

```
x_test=pd.DataFrame(scaler.transform(x_test), columns=x_test.columns)

input_test=torch.tensor(x_test.values ,dtype=torch.float32)
label_test=torch.tensor(y_test.values ,dtype=torch.float32).view(-1,1)

test_ds=TensorDataset(input_test , label_test)
test_dl=DataLoader(test_ds,batch_size=40 , shuffle=True)

def test_onenn(model, loss_fn , test_dl):
    model.eval()
    total_loss = 0
    correct = 0
    wrong = 0
    with torch.no_grad():
        for x, y in test_dl:
            output = model(x)
            total_loss += loss_fn(output, y)



            predicted = (output > 0.5).float()

            correct+= (predicted==y).sum().item()
            wrong += (predicted != y).sum().item()

    # Print the number of correct and wrong predictions
    print(f"Correct predictions: {correct}")
    print(f"Wrong predictions: {wrong}")
    print(f"Total Loss: {total_loss.item() / len(test_dl):.4f}")

test_onenn(model, loss_fn , test_dl)
```

The test() function is used to test a one-layer neural network labeled onenn across a test dataset. First, the test data would be standardized using a scaler and then converted into a PyTorch tensor. The labels are similarly transformed to a tensor before they are reshaped

into a column vector. A TensorDataset is created to pair the input features and labels, and a DataLoader is used to batch the data with a batch size of 40 and shuffle the data for each epoch. The test_onenn function is designed to evaluate the performance of the trained model on this test dataset. Inside this function, the model is switched to evaluation mode with the line model.eval() and predictions are made without updating the model's weights during training using torch.no_grad(). For each batch, the model output is computed, loss is accumulated, and predictions are made by applying a threshold of 0.5 on the output. The number of correct and incorrect predictions is tracked by comparing predicted values to the actual labels. After all the batches are processed, the function prints out the total number of correct and wrong predictions along with the average loss across all the batches. Then it calls the test_onenn function passing the model, loss function, and test data loader as arguments to carry out the evaluation.

Result (single layered neural network)

Correct predictions: 243
Wrong predictions: 57
Total Loss: 15.5697

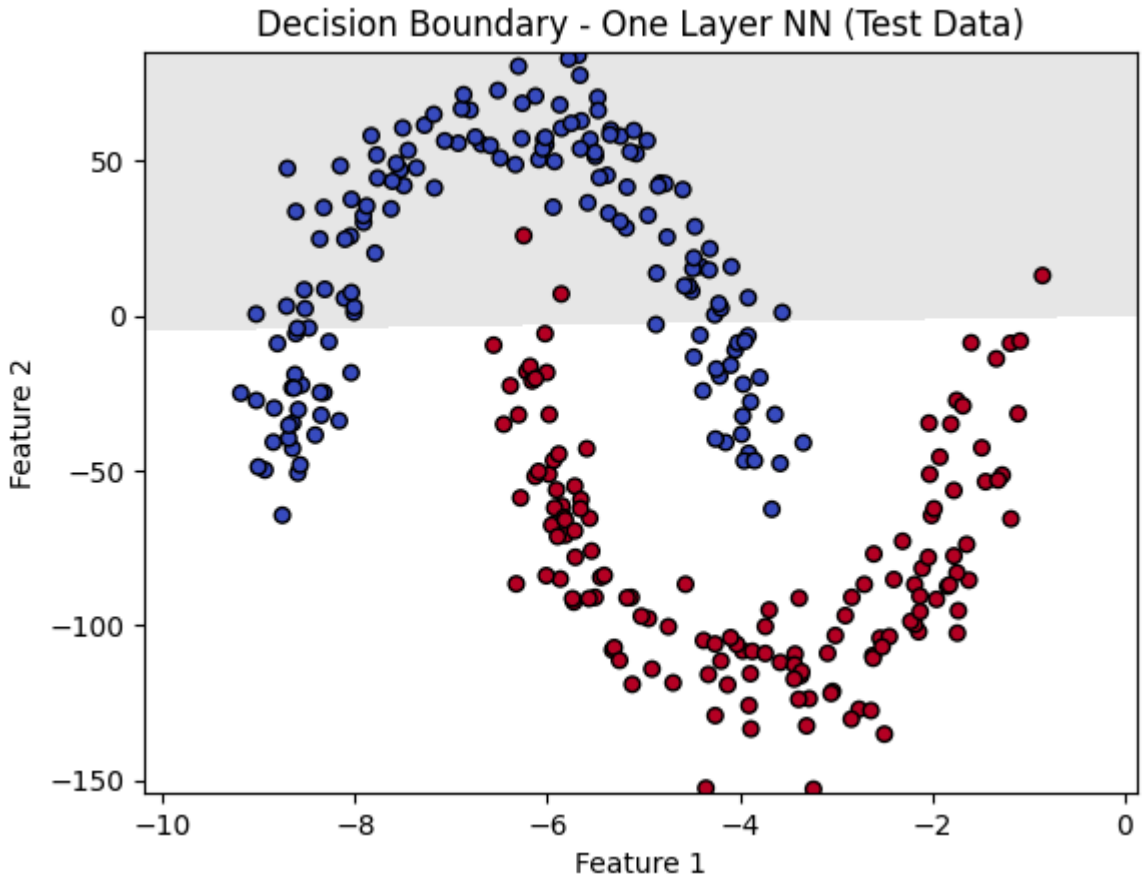
Accuracy=number of correct prediction / total number of cases
= 243/300
= **81%**

Visualizing single layer NN results

```
import numpy as np
def plot_decision_boundary(model, x_data, y_data, resolution=0.02):
    x_min, x_max = x_data[:, 0].min() - 1, x_data[:, 0].max() + 1
    y_min, y_max = x_data[:, 1].min() - 1, x_data[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, resolution),
                         np.arange(y_min, y_max, resolution))

    # Convert grid points to tensor and make predictions
    grid_points = torch.tensor(np.c_[xx.ravel(), yy.ravel()],
dtype=torch.float32)
    with torch.no_grad():
        model.eval()
        zz = model(grid_points).numpy().reshape(xx.shape)

    # Plot the contour map of the decision boundary
    plt.contourf(xx, yy, zz, levels=[0, 0.5], cmap='coolwarm', alpha=0.6)
    plt.scatter(x_data[:, 0], x_data[:, 1], c=y_data, cmap='coolwarm',
edgecolors='k', s=30)
```

```
    plt.title("Decision Boundary - One Layer NN (Test Data)")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()

# Visualize decision boundary for the OneLayerNN on the test data
plot_decision_boundary(model, input_test.numpy(), label_test.numpy())
```

The plot_decision_boundary function plots a model's decision boundary on 2D datasets. It takes the minimum and maximum values to be plotted across each feature from x_data while adding a buffer of 1 to each of the axes for padding. Subsequently, the function generates the mesh grid in the form xx, yy of the same span as the extent of the input features.
The grid points are then converted to a tensor and passed through the model to obtain predictions. The predictions are reshaped to match the shape of the mesh grid. Using plt.contourf, the function plots a contour map representing the decision boundary, with the region where the model predicts class 0 and class 1 (with a decision threshold of 0.5). It also lays over the actual test data points (x_data and y_data) over this plot, colored according to their true labels, with a black edge around each point for clarity. This plot is titled "Decision Boundary - One Layer NN (Test Data)", has labeled axes ("Feature 1" and "Feature 2") and is presented by plt.show().
This function can help to visualize the decision boundary and how well it separates the different classes of data on the test dataset. This function will call to visually illustrate the decision boundary of the OneLayerNN model on the test data.



Defining multi layer neural network(5 layered in this case)

```
class fivelayernn(nn.Module):
  def __init__(self , input_size , hidden_size, output_size):
    super(fivelayernn,self).__init__()

    self.layer11=nn.Linear(input_size ,hidden_size)
    self.layer22=nn.Linear(hidden_size ,hidden_size)
    self.layer33=nn.Linear(hidden_size ,hidden_size)

    self.layer55=nn.Linear(hidden_size ,output_size)
    self.reLu=nn.ReLU()
    self.sigmoid=nn.Sigmoid()

  def forward(self , x):
    x=self.reLu(self.layer11(x))
    x=self.reLu(self.layer22(x))
    x=self.reLu(self.layer33(x))
    x=self.sigmoid(self.layer55(x))
    return x
```

```
model1=fivelayernn(input,10,output)
opt1=torch.optim.SGD(model1.parameters(),lr=1e-3)
```

The class of multi-layer neural network is defined using PyTorch's nn.Module as fivelayernn. This network has five layers:one input , three hidden layers and one output layer. Its __init__ initializes these layers along with activation functions. Among them, layer11, layer22, and layer33 are fully connected linear layers.
Each takes the input or previous hidden layer and transforms into a specified hidden_size:. The last layer is the output layer, layer55, which transforms the final hidden layer into output size. Rectified Linear Unit (ReLU) is applied following each of the first three as the activation after each of those. The Sigmoid activation function is used in the output layer. This function maps the output into a range between 0 and 1, making it more suitable for binary classification. Forward defines how data flows through the network. The output of every layer feeds into its activation function before moving on to the next layer.
The model is defined and instantiated with an input size of input, a hidden size of 10, and an output size of output. For model training, the choice of optimizer made here is SGD, which would update the weights of the model to minimize loss at a learning rate of 1e-3.

Training 5 layered neural network

```
def training(model1 ,loss_fn ,opt1 ,epoch ,train_dl):
  for e in range(epoch):
    model1.train()
    for x , y in train_dl:
      opt1.zero_grad()
      output=model1(x);
      loss=loss_fn(output,y)
      loss.backward()
      opt1.step()
      if (e + 1) % 10 == 0:
            print(f"Epoch [{e+1}/{epoch}], Loss: {loss.item():.4f}")

training(model1,loss_fn,opt1,10000,train_dl)
```

The training function is meant to train the fivelayernn model for a specified number of epochs , 10000 in this case. Inside the function, a loop runs for the given number of epochs (epoch), and for each epoch, model1.train() sets the model to training mode. The function proceeds as per the batch numbers passing through the training data loader, a train_dl.
The batch inside this loop will have two entries: the input items (x) and their corresponding labels (y). For every batch, gradients of the optimizer are zeroed using

opt1.zero_grad(), and a forward pass is performed by passing x through the model to get the output. Loss is calculated as a difference between model's output and true labels by using loss function (loss_fn). The gradients of loss with respect to model's parameters are computed by loss.backward(), and opt1.step() updates model's parameters in order to minimize the loss. Every 10 epochs, the loss for that epoch is printed to track the model's progress. Now let's call the function with 10,000 epochs, the model, the loss function, the optimizer and the training data loader as arguments to start training.

```
Streaming output truncated to the last 5000 lines.
Epoch [7920/10000], Loss: 0.0030
Epoch [7920/10000], Loss: 0.0040
Epoch [7920/10000], Loss: 0.0032
.
.
.
.
Epoch [10000/10000], Loss: 0.0020
Epoch [10000/10000], Loss: 0.0075
Epoch [10000/10000], Loss: 0.0046
```

Testing the multi layered network (5 layered)

```
def test_fivenn(model1, loss_fn , test_dl):
    model1.eval()
    total_loss1 = 0
    correct1 = 0
    wrong1 = 0
    with torch.no_grad():
        for x, y in test_dl:
            output = model1(x)
            total_loss1 += loss_fn(output, y)



            predicted = (output > 0.5).float()
            #print('\n',predicted , "-> ",y)
            correct1 += (predicted == y).sum().item()
            wrong1 += (predicted != y).sum().item()



    print(f"Correct predictions: {correct1}")
    print(f"Wrong predictions: {wrong1}")
    print(f"Total Loss: {total_loss1.item() / len(test_dl):.4f}")

test_fivenn(model1 , loss_fn , test_dl)
```

This function is used to test the performance of the trained fivelayernn model on a test dataset. Inside the function, it puts the model in evaluation mode by calling model1.eval(). It makes sure certain layers such as dropout behave properly at test time. It then initializes some variables that are going to keep track of total loss, the number of correct predictions, and the number of incorrect predictions.
Using torch.no_grad(), it is ensured that no gradients are computed, thus making the evaluation process faster since updates to the model are not needed during testing. The function iterates over test data in test_dl and, for each batch, the model's output is computed by passing the input x through the model. The loss is then computed by taking the difference of the model's output and true labels y against the given loss function (loss_fn). In order to get the predictions, a threshold value of 0.5 is applied to the model's output, which further converts it into binary predictions such as 0 or 1. The count of correct predictions is then determined by comparing the predicted values to the actual labels, and so is the case with the number of wrong predictions.
This function processes all the batches and then prints the correct and wrong prediction counts, and the average loss over all the batches. Then, the model is called using the trained model, the loss function, and the test data loader to evaluate the model.

Result (Multi layer neural network)

Correct predictions: 265
Wrong predictions: 35
Total Loss: 2.8345

Accuracy=total number of correct prediction / total number of labels
= 265/300
= **88.5%**

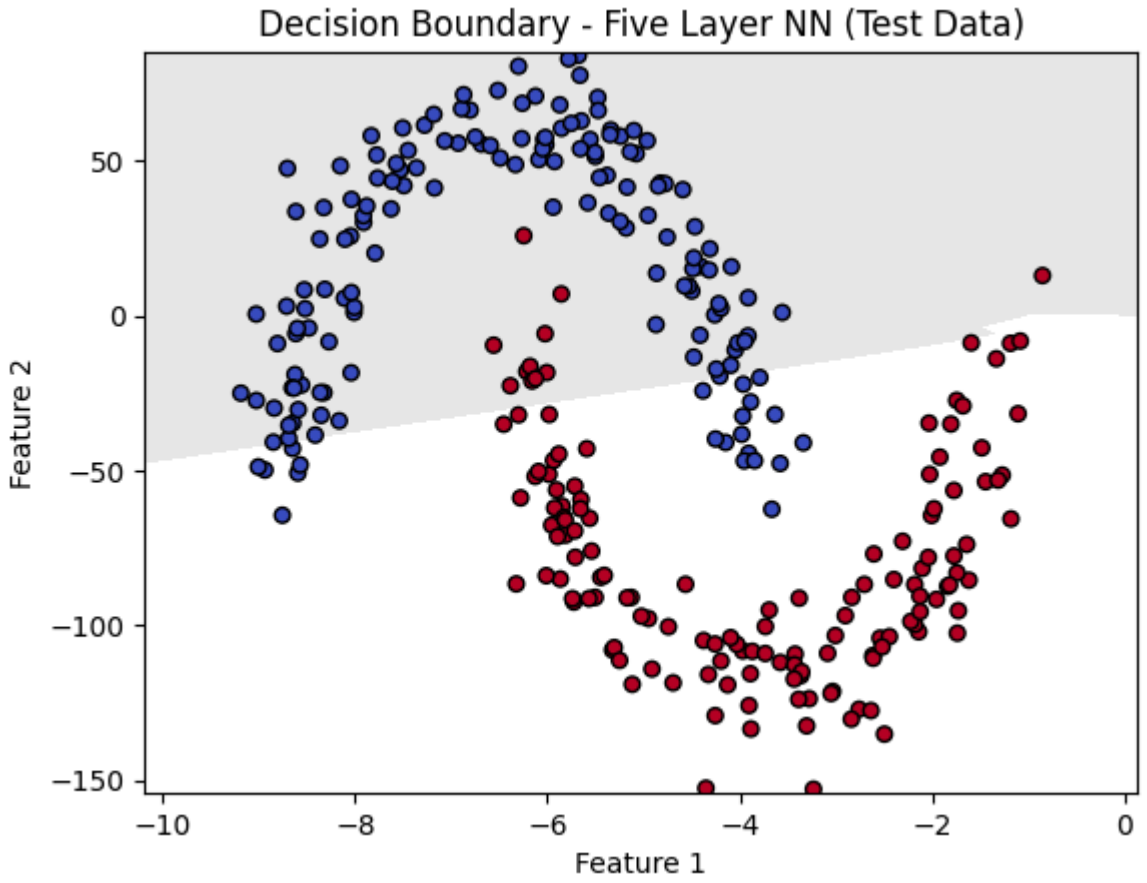
Visualizing results of multi layer neural network

```
def plot_decision_boundary(model, x_data, y_data, resolution=0.02):
    x_min, x_max = x_data[:, 0].min() - 1, x_data[:, 0].max() + 1
    y_min, y_max = x_data[:, 1].min() - 1, x_data[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, resolution),
                         np.arange(y_min, y_max, resolution))


    grid_points = torch.tensor(np.c_[xx.ravel(), yy.ravel()],
dtype=torch.float32)
    with torch.no_grad():
```

```
        model.eval()
        zz = model(grid_points).numpy().reshape(xx.shape)

    # Plot the contour map of the decision boundary
    plt.contourf(xx, yy, zz, levels=[0, 0.5], cmap='coolwarm', alpha=0.6)
    plt.scatter(x_data[:, 0], x_data[:, 1], c=y_data, cmap='coolwarm',
edgecolors='k', s=30)
    plt.title("Decision Boundary - Five Layer NN (Test Data)")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()

plot_decision_boundary(model1, input_test.numpy(), label_test.numpy())
```

The `plot_decision_boundary` function has been modified to plot the decision boundary for a **five-layer neural network** consisting of three hidden layers, one input layer, and one output layer. It first finds the minimum and maximum values of the two features in the input data (`x_data`) with 1 added for both axes. It then creates a mesh grid using `np.meshgrid`, which spans the entire feature space defined by these min and max values, with a specified resolution for the grid points. The function converts these grid points into a tensor, which is passed through the model to predict the class for each grid point.

Because we are not updating the model during testing, we use `torch.no_grad()` to prevent gradient calculations. The predictions are re-shaped to resemble the shape of the grid, and the decision boundary is presented. The `plt.contourf` tool is used for visualizing the decision boundary regions where the model predicts class 0 and class 1, with a threshold of 0.5 on the boundary coloring. The actual test data points, `x_data` and `y_data`, are overlaid over the contour plot, colored by their true class labels with black borders around the points so they stand out better.
The plot is titled and axis labeled appropriately, and `plt.show()` is used to display it. This way, it will clearly show how the five-layer neural network separates the data into classes based on the learned decision boundary.
Then the function is called to visualize the decision boundary of the `model1` which is a five-layer neural network on the test data `input_test` and `label_test`.



Observations:

Single layered neural network has an accuracy of **81%** and multi layered neural network has an accuracy of **88.5%** along with much lower total loss of **2.8345** against the total loss of **15.5697** . The accuracy of MLNN is significantly higher than that of single layer network. Also during the training phase , it achieved the loss of **0.0046** vs **0.2365** in case of single layer neural network. This shows that multi layer neural network is much more capable of classifying non linearly separable data points due to it's much more superior ability to learn non linear more complex relationships than that of single neuron , which is nothing more than a logistic regression instance (single neuron with sigmoid applied is equivalent to logistic regression model).

Conclusion:

We have successfully observed the behaviour of single neuron and the multi layer neural network and the difference between the result achieved by both.

Executable file(.ipynb) Linear separation