Introduction

This assignment presents a two-dimensional binary classification problem involving non-convex decision regions i.e., C1 & C2. The proposed solution employs a back propagation network to distinguish whether an input pattern belongs to class C1 or C2.

I have employed Google Colab, specifically Colab Notebook, which is a web-based interactive environment that allows you to write and execute code. The platform is especially designed and optimized for data science, machine learning, and data analysis applications.

For implementation, I have made use of Python programming language along with machine learning (ML) & deep learning (DL) platform TensorFlow and high-level API Keras, which focuses on solving ML problems through the provision of abstractions and building blocks for developing the model. In addition, to aid data pre-processing, manipulation, and visualization – scikit-learn, NumPy, pandas, matplotlib, seaborn etc., have been utilized.

The entire ML workflow has been adopted: dataset generation, model building, model evaluation, model optimization & results. These are discussed in depth in the subsequent sections.

Methodology & Justification

I. Multi-layer Perceptron

Multi-layer perceptron (MLP) is a class of fully connected feedforward neural networks. It consists of at least three layers: input layer, hidden layer, and the output layer. Each node/neuron apart from the input layer uses a non-linear activation function. MLP utilizes backpropagation technique for learning weights, a type of supervised learning, and can distinguish non-linearly separable data. It is also referred to as a Back Propagation Network.

II. Data Generation & Analysis

Dataset has been generated using piece-wise functions to map the curvatures of the regions and has been populated with data points in between the decision surfaces. This was achieved with the aid of python and excel workbook. The excel workbook is imported to the Colab environment as a comma separated (csv) file and is then merged with the python generated data points. Figure 1. depicts characteristics of the dataset.

There are 456 patterns/data points in total. Column ‘x’ represents the x-axis coordinate (feature) and column ‘y’ represents the accompanying y-axis coordinate (feature) to form a single pattern/data point in the two-dimensional space. The label or column ‘region’ defines the associated region the pattern belongs to i.e., two classes.

The value counts for each class are as shown below:

Class	Count	Percentage
C1	228	50%
C2	228	50%

Thus, we have a balanced dataset with equal representation from both classes.

	x	y	region
373	0.200000	0.200000	C2
39	0.020000	0.000000	C1
340	-0.600000	-0.600000	C2
218	0.700000	-0.800000	C1
155	-0.900000	1.600000	C1
...	...	...	...
106	-1.800000	-0.200000	C1
270	-0.376731	-0.789474	C2
348	-0.400000	-0.700000	C2
435	1.500000	-0.600000	C2
102	-1.900000	-0.200000	C1

456 rows × 3 columns

Figure 1.

Figure 2. represents the scatter plot of the dataset.

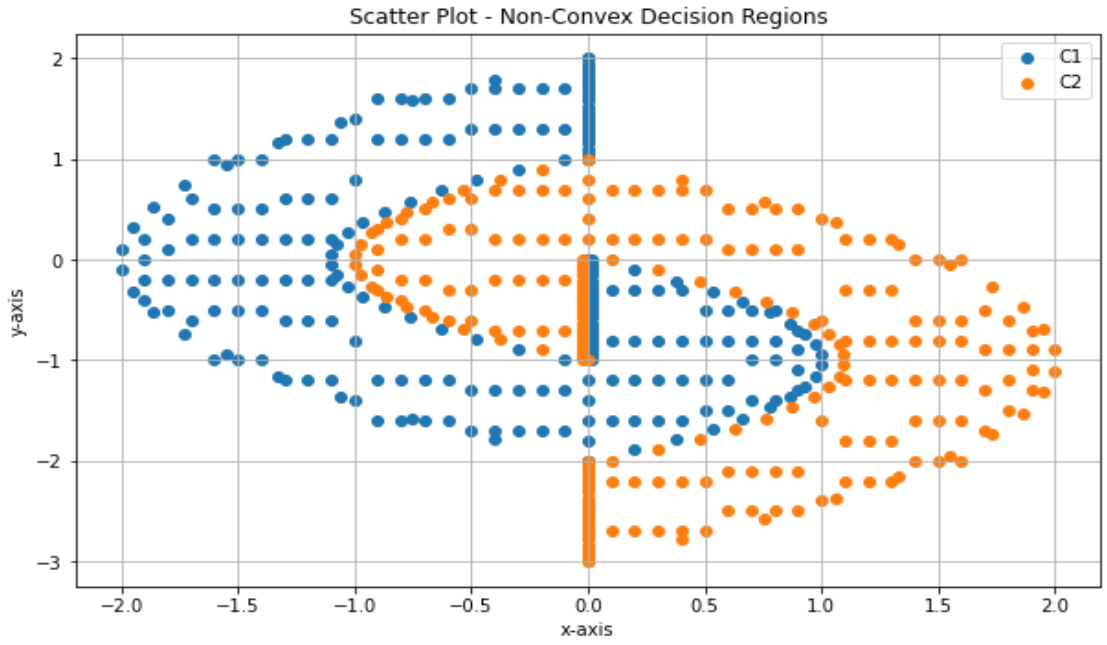


Figure 2.

III. Data Preprocessing

Under this section, we look at the numerous preprocessing techniques that were employed to transform the dataset into clean and standardized representation for feeding the network.

i. Feature and Target Data Frame

To perform further preprocessing on the data set it is pertinent that we distinguish data by splitting it into two data frames, the feature set, and the target set. This enables us to perform feature scaling and encoding as per our need.

```
X = coordinates.drop('region', axis = 1)
y = coordinates['region']
```

The commands above have been employed in this case.

ii. Label Encoding

Using the *LabelEncoder* class from the sci-kit learn library, I have encoded the target variable classes into integers, where the new representation is as follows:

C1 → 0

C2 → 1

This enables ease in terms of classification for the network due to their machine-readable nature.

iii. Feature Scaling

Standardization is applied when you have features with different units and scale. It essentially wraps the values around the mean = 0 with standard deviation = 1. This is done to effectively remove bias which can arise due to features having different scales and thus impacting the overall prediction. It does not contain the variable within a range.
Standardization applies the following formula to each numerical feature for scaling:

$$X_{new} = \frac{X - mean}{std}$$

iv. Training and Test Split

After preprocessing data into the right format, I have split the data into training and test data sets. For this purpose, I have employed *train_test_split* class of the sci-kit learn library.
We have four data frames: *X_train* and *y_train*, used for training the model. *X_test* and y_test employed for validation and evaluation of the network. The training set is 80% of the data whilst validation/testing set is 20%.

IV. Model Building & Evaluation

Under this section, I implement the model and evaluate it based on a few evaluation metrics. The following sub-section outlines these metrics:

i. Evaluation Metrics

- Accuracy

  Accuracy is one of the metrics employed for evaluating classification models. It is represented as:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

- Binary Cross Entropy

  A loss function is a measurement between the algorithm's current output and expected output. It enables us to evaluate its performance.

  Binary cross entropy is a loss function that is employed for binary classification problems. It compares the predicted probabilities with the actual class output to compute the penalty score that penalizes probabilities based on their distance from the actual values.

- Confusion Matrix

  It is a table that visualizes the model's performance. It shows all the possible labels and how the model can correctly or incorrectly predict them. There are four quadrants in the matrix and our explained as follows:

  - **True Positive (TP)**: Positively labeled instance that the model correctly predicts.
  - **False Positive (FP)**: Negatively labeled instance that the model incorrectly predicts as positive.
  - **False Negative (FN)**: Positively labeled instance that the model incorrectly predicts as positive.
  - **True Negative (TN)**: Negatively labeled instance that the model correctly predicts.

  For us, in terms of the model's predictive accuracy, TP and TN hold great importance.

The underlying/base version of the model has the following characteristics:

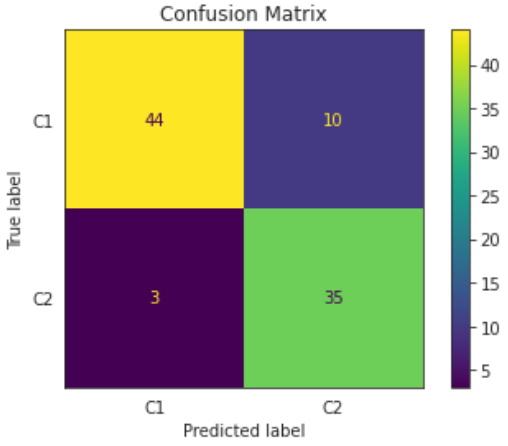
Hidden Layers	Hidden Layer Neurons	Activation Function	Optimization Algorithm	Learning Rate	Loss Function	Epochs
01	10	ReLu	Adam	0.01	Binary Cross Entropy	50

The performance recorded is as shown below:

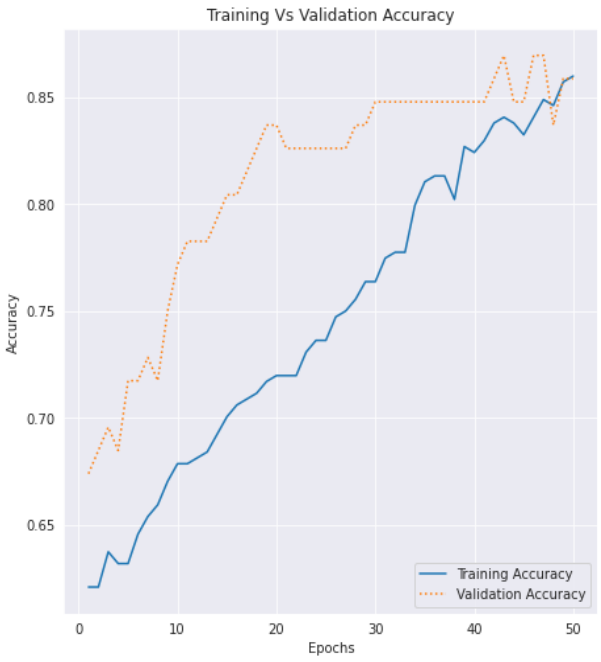
Accuracy	Loss	Training Time
85.87%	0.3018	11.57 sec

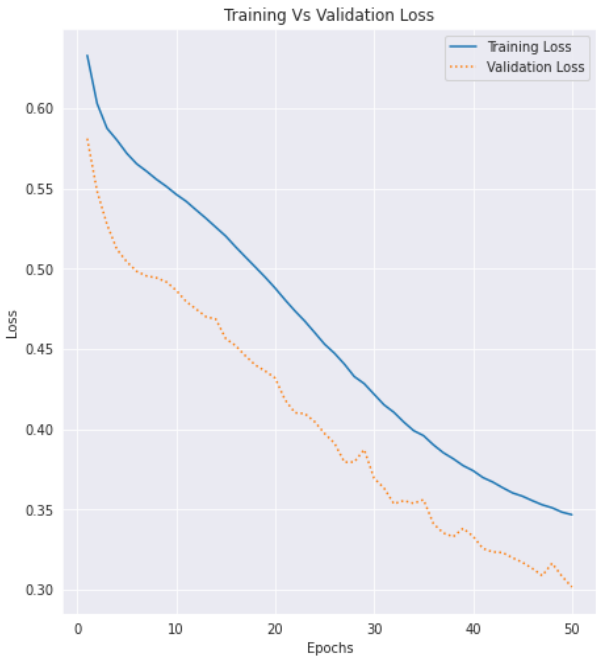
The confusion matrix characteristics are as follows:

- TP: 44 (47.83%)
- TN: 35 (38.43%)
- FN: 10 (10.87%)
- FP: 3 (3.26%)
- Total instances: 92



In addition, the training vs validation curves for 'accuracy' and 'loss' are shown below.





The following sub-section underlines a comparative analysis between various model versions obtained through parameter/hyperparameter tuning, and their impact on the classifier performance, learning speed etc.

The following table records the effect of varying the number of neurons in the hidden layer on the model's performance metrics. All other parameters/hyperparameters are kept constant (same as the base model), except for the hidden layer(s) which is only one.

Table 1. Number of Neurons - Variable

Hidden Layers	Neurons	Accuracy	Loss	Training Time
01	6	78.26%	0.4178	13.06 sec
01	12	83.70%	0.2992	22.27 sec
01	48	86.96%	0.2295	17.96 sec
01	128	89.13%	0.2002	20.92 sec
01	200	89.13%	0.2001	21.73 sec
01	400	90.02%	0.1971	22.07 sec

The following table records the impact of varying the number of hidden layers on the model's performance metrics. All other parameters/hyperparameters are kept constant (same as the base model), with 10 neurons per successive layer.

Table 2. Hidden Layers - Variable

Hidden Layers	Neurons per Hidden Layer	Accuracy	Loss	Training Time
01	10	88.04%	0.2902	13.24 sec
05	10	96.74%	0.1203	13.53 sec
10	10	92.39%	0.2100	16.59 sec
15	10	89.13%	0.1781	18.83 sec
20	10	41.30%	0.6948	19.16 sec

Following observations are deduced from the tables above:

- As we increase the number of hidden layers, the model's accuracy increases until it reaches a steady state point. As for loss, it decreases with increment in hidden layers.
- However, after a certain threshold the model begins to overfit and is unable to generalize well, increasing loss and decreasing predictive accuracy.
- A similar trend is observed when increasing the number of neurons in the hidden layer.
- Additionally, it is also noted that with many hidden layers and neurons, model complexity increases due to the increase in learnable parameters. This in turn impacts the learning speed/training time of the network.

The following table records the impact of varying the learning rate using the 'Stochastic Gradient Descent' (SGD) optimization algorithm on the model's performance metrics. All other parameters/hyperparameters are kept constant (same as the base model), with the number of hidden layers equal to one and momentum coefficient set to zero.

Table 3. Learning Rate - Variable

Hidden Layers	Optimization Algorithm	Learning Rate	Accuracy	Loss	Training Time
01	SGD	0.001	66.30%	0.5927	5.91 sec
01	SGD	0.01	70.65%	0.5850	6.07 sec
01	SGD	0.05	73.91%	0.5142	10.02 sec
01	SGD	0.1	77.17%	0.4873	22.03 sec
01	SGD	0.4	84.78%	0.3526	11.45 sec
01	SGD	0.9	90.22%	0.2334	9.86 sec

Following observations are deduced from the table above:

- The learning rate is an important parameter which defines how much (quantifies) change to introduce into the model based on the estimated error when adjusting weights during the network's training phase.
- With large values of the learning rate, the network is able to quickly learn but oscillates significantly.
- With small values, the model's performance is poor since it is unable to learn with the constraints that exist and requires more training steps.

The following table records the impact of varying the momentum coefficient using the 'Stochastic Gradient Descent' optimization algorithm on the model's performance metrics. All other parameters/hyperparameters are kept constant (same as the base model), with the number of hidden layers equal to one and the learning rate fixed to a constant value of 0.01.

Table 4. Momentum Coefficient - Variable

Hidden Layers	Optimization Algorithm	Learning Rate	Momentum Coefficient	Accuracy	Loss	Training Time
01	SGD	0.01	0.01	73.91%	0.5366	12.63 sec
01	SGD	0.01	0.05	71.74%	0.5623	11.84 sec
01	SGD	0.01	0.2	64.13%	0.6027	5.00 sec
01	SGD	0.01	0.4	51.29%	0.6484	14.05 sec
01	SGD	0.01	0.7	72.83%	0.5107	21.64 sec
01	SGD	0.01	0.9	78.26%	0.4759	21.69 sec

Following observation is deduced from the table above:

- With increase in the momentum coefficient, the network's training accelerates before slowing down for large values of the coefficient. Accuracy & loss deteriorates before improving when coefficient values approach 1.

The optimal version of the model has the following characteristics. Note: This was obtained through an informed trial and error process. However, due to limitation of computation resources as well as the non-convex nature of the error surface, there may exist a more optimized/global optimum network for this problem with the given set of parameter/hyperparameters.

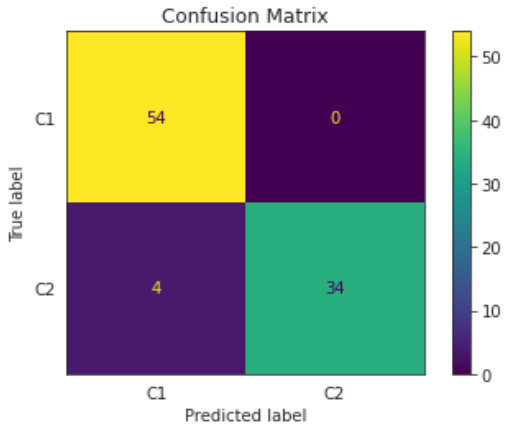
Hidden Layers	Activation Function	Optimization Algorithm	Learning Rate	Loss Function	Epochs	Shuffle
03	ReLu	Adam	0.01	Binary Cross Entropy	50	Yes

Note: The 'Adam' optimization algorithm is a variation of the classical stochastic gradient descent method as it computes a per-parameter learning rate which it adapts as learning progresses from the first (mean) and second (uncentered variance) moment of the gradient.

Accuracy	Loss	Training Time
95.65%	0.1059	9.70 sec

The confusion matrix characteristics are as follows:

- TP: 54 (58.67%)
- TN: 34 (36.96%)
- FN: 0 (0%)
- FP: 2 (4.35%)
- Total instances: 92



In addition, the training vs validation curves for 'accuracy' and 'loss' are shown below.

V. Appendix

```
#Importing requisite libraries
import io
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from itertools import product
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
from sklearn.utils import shuffle
from tensorflow import keras
from keras.models import Sequential, load_model
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint
from keras.layers import Dropout
from keras import regularizers
from google.colab import files
uploaded = files.upload()

#Dataset Generation

datapoints = pd.read_csv((io.BytesIO(uploaded["datapoints.csv"])))

y_1 = np.linspace(1,2,20)
x_1 = np.zeros(20)

y_2 = np.linspace(-1,0,20)
x_2 = np.full((20),0.02)

y_3 = np.linspace(-2,0,20)
x_3 = -1*(y_3+1)**2 + 1

y_4 = np.linspace(-1,1,20)
x_4 = y_4**2 - 1.1

y_5 = np.linspace(-2,2,20)
x_5 = (1/2)*y_5**2 - 2

x_c1 = datapoints['x_c1']
y_c1 = datapoints['y_c1']
```

```
y_6 = np.linspace(-3,-2,20)
x_6 = np.zeros(20)

y_7 = np.linspace(-1,0,20)
x_7 = np.full((20),-0.02)

y_8 = np.linspace(-1,1,20)
x_8 = y_8**2 - 1

y_9 = np.linspace(-2,0,20)
x_9 = -1*(y_9+1)**2 + 1.1

y_10 = np.linspace(-3,1,20)
x_10 = (-1/2)*(y_10+1)**2 + 2

x_c2 = datapoints['x_c2']
y_c2 = datapoints['y_c2']

C1_x = np.concatenate((x_1, x_2, x_3, x_4, x_5, x_c1 ))
C1_y = np.concatenate((y_1, y_2, y_3, y_4, y_5, y_c1 ))

C2_x = np.concatenate((x_6, x_7, x_8, x_9, x_10, x_c2))
C2_y = np.concatenate((y_6, y_7, y_8, y_9, y_10, y_c2))

C1 = np.array([list(pair) for pair in zip(C1_x, C1_y)])
C2 = np.array([list(pair) for pair in zip(C2_x, C2_y)])

fig = plt.figure(figsize = (10, 6))
plt.grid()
plt.scatter(C1[:,0], C1[:,1],)
plt.scatter(C2[:,0], C2[:,1])
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.legend(["C1", "C2"], loc ="upper right")
plt.title('Scatter Plot - Non-Convex Decision Regions')
plt.show()

label_1 = ['C1']*len(C1)
label_2 = ['C2']*len(C2)
label = label_1 + label_2

coordinates = pd.DataFrame(np.concatenate((C1,C2)))
coordinates.columns = ['x', 'y']
coordinates['region'] = label
coordinates = shuffle(coordinates, random_state= 42)
```

```
#Data Preprocessing

X = coordinates.drop('region', axis = 1)
y = coordinates['region']

label_region = LabelEncoder()
y = label_region.fit_transform(y)
transform = StandardScaler()
X = transform.fit_transform(X)

X_train,X_test,y_train,y_test = train_test_split(X, y, random_state=42, test_
size=0.2) # Splitting data into the training and testing sets


#Model Building and Training

model = Sequential([
      keras.Input(shape=(2,)),
      Dense(units=24, activation='relu', name="hidden_1"),
      Dense(units=48, activation='relu', name="hidden_2"),
      Dense(units=24, activation='relu', name="hidden_3"),
      Dense(units=1, activation='sigmoid', name="output")
])

model.compile(
      optimizer = keras.optimizers.Adam(learning_rate=0.01),
      loss = keras.losses.BinaryCrossentropy(),
      metrics = ['accuracy']
)

filepath = 'my_best_model.hdf5'
checkpoint = ModelCheckpoint(filepath=filepath, monitor='val_loss',
verbose=1, save_best_only=True, mode='min')

callbacks = [checkpoint]
t0 = time.time()
history = model.fit(
      x = X_train,
      y = y_train,
      validation_data=(X_test, y_test),
      batch_size = 64,
      epochs = 50,
      callbacks=callbacks)
print("Training time: ", time.time()-t0)
model.summary()
```

```
#Model Evaluation

model = load_model(filepath)
model.evaluate(X_test, y_test,)

#Loss and Accuracy Plots

sns.set_style("darkgrid") #Enabling grid of the graphs
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1,len(accuracy)+1)
plt.figure(figsize=[16,8])
plt.subplot(1, 2, 1) #Plotting Training vs Validation Accuracy
plt.plot(epochs, accuracy, '-', label = 'Training Accuracy')
plt.plot(epochs, val_accuracy, ':', label = 'Validation Accuracy')
plt.title('Training Vs Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc = 'lower right')

plt.subplot(1, 2, 2) #Plotting Training vs Validation Loss
plt.plot(epochs, loss, '-', label = 'Training Loss')
plt.plot(epochs, val_loss, ':', label = 'Validation Loss')
plt.title('Training Vs Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc = 'upper right')
plt.show()

#Confusion Matrix

sns.set_style("white")
#convert the predicted probabilities to True/False labels (where True represe
nts lablel 'C1', while False represents label 'C2')
predictions = model.predict(X_test)> 0.5
#confusion matrix to display classification results
cm = confusion_matrix(y_test, predictions)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['C1','C2']
)
disp.plot()
plt.title('Confusion Matrix')
plt.show()
```