Introduction

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

I have employed Jupyter 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 data processing and manipulation libraries pandas, NumPy, and Sci-kit learn. In addition, for the purpose of data visualization – matplotlib & seaborn libraries 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. Counter Propagation Network

A counter propagation network (CPN) is a mapping neural network, whilst its architecture is a combination of the self-organizing map of Kohnen and outstar structure of Grossberg. It consists of three layers: input, output, and the hidden layer. The hidden layer or the competitive layer implements a winner-takes-all mechanism (unsupervised learning) to determine which output neuron is most like the input pattern. The neuron with the highest activation value is chosen as the winner, and its weight vector is updated to become more like the input pattern, essentially creating prototype vectors used to form a decision boundary. The output layer implements supervised learning, where it adjusts the weights between the hidden and output layer to produce the desired output for each input pattern. The prototypes act as attractors for the input patterns, pulling them towards the right appropriate class.

II. Data Generation & Analysis

The dataset has been generated using piece-wise functions to map curvatures of the regions and 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 Jupyter 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 depict a balanced dataset with equal representation from both classes.

Class	Count	Percentage
C1	228	50%
C2	228	50%

	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.

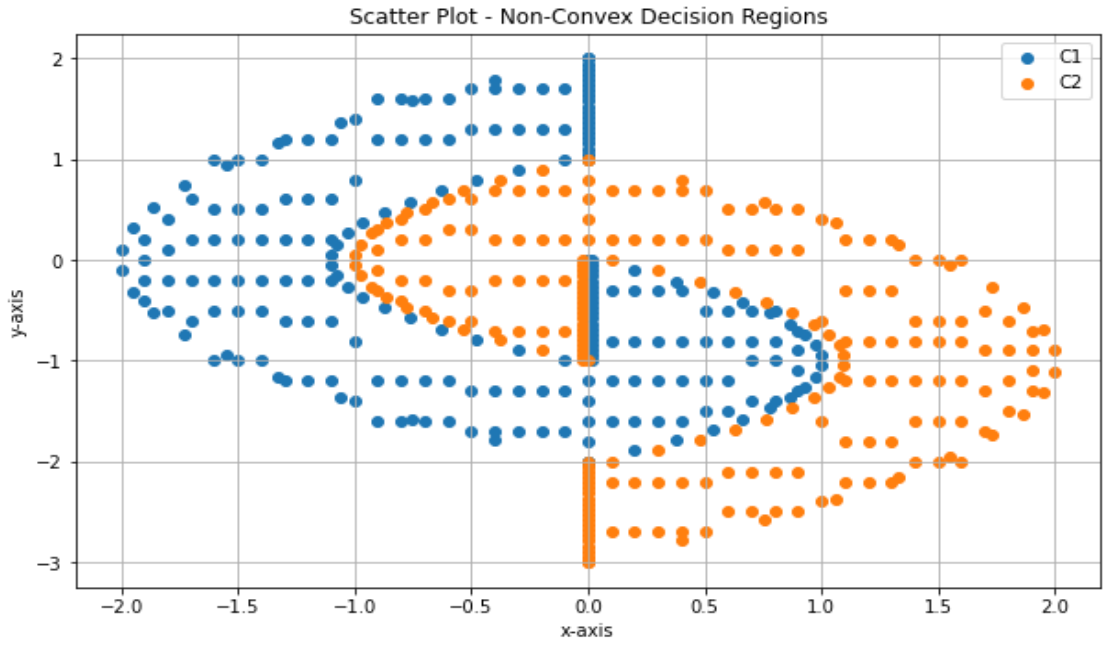


Figure 2.

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

Normalization is applied when you have features with different units and scales. It curtails the range of values to be between 0-1. This is done to effectively remove bias which can arise due to features having different scales and thus impacting the overall prediction.

Normalization applies the following formula to each numerical feature for scaling:

$$X_{new} = \frac{(x_i - x_{min})}{(x_{max} - x_{min})}$$

Where:

- x_i: The i^{th} value of the dataset.
- x_{min}: The minimum value in the dataset.
- x_{max}: The maximum value in the dataset.

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:

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

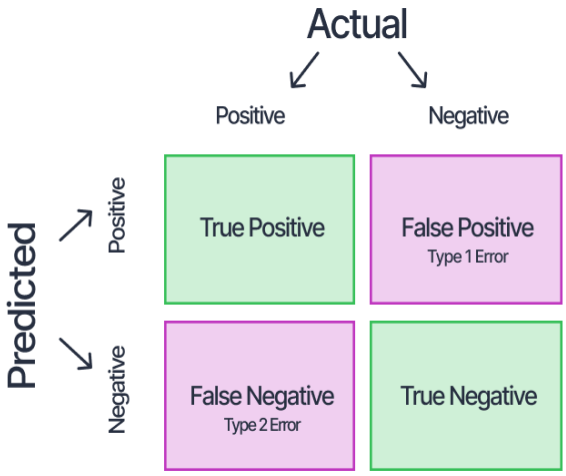
In addition, it binary classification accuracy can also be expressed in terms positives and negatives:

$$Accuracy\ = \frac{TP\ +\ TN}{TP\ +\ TN\ +\ FP\ +\ FN}$$

- 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 as positive.
• **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 as negative.



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:

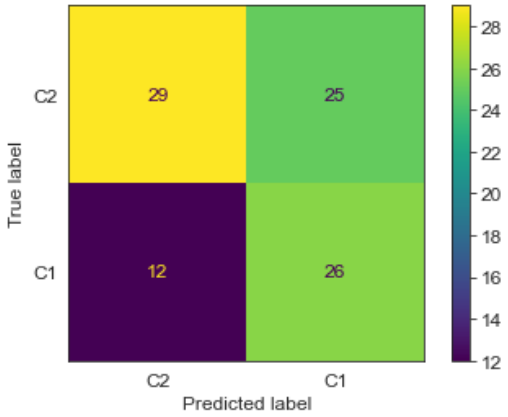
Hidden Layer Neurons	Kohnen Layer Learning Rate	Grossberg Layer Learning Rate	Decay Rate	Epochs	Shuffle
10	0.01	0.01	0.5	10	Yes

The performance recorded is as shown below:

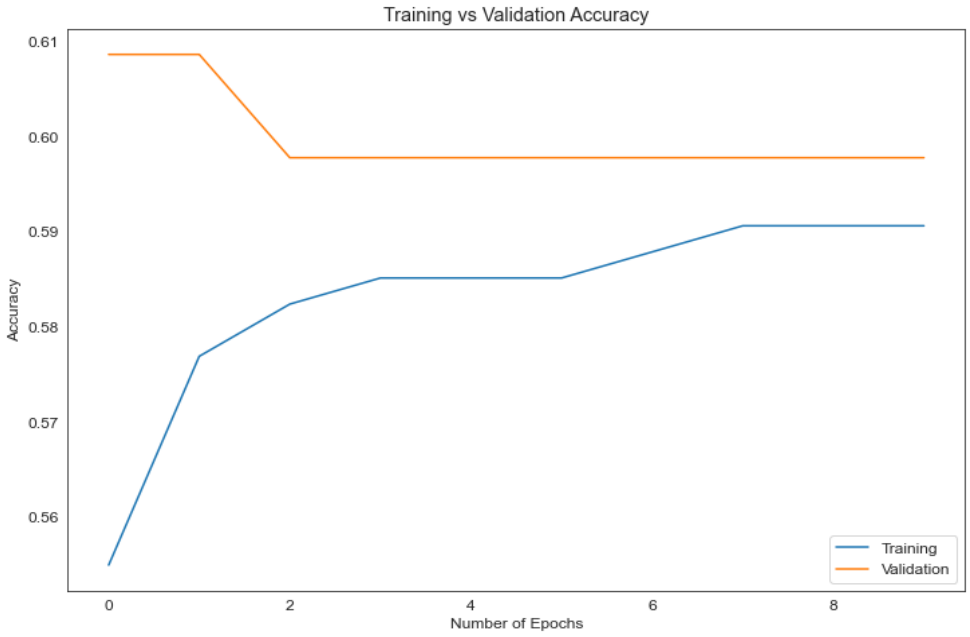
Accuracy	Training Time
59.78%	0.31 sec

The confusion matrix characteristics are as follows:

- TP: 29 (31.52%)
- TN: 26 (28.26%)
- FP: 12 (13.04 %)
- FN: 25 (27.17%)
- Total instances: 92



In addition, the training vs validation curve for 'accuracy' is shown below.



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

Table 1. records the effect of varying the number of neurons in the hidden layer on the model's performance metrics. All other hyperparameters are kept constant, same as the base model.

Table 1. Number of Neurons - Variable

Neurons	**Accuracy**	**Training Time**
10	64.13%	0.16 sec
30	64.13%	0.19 sec
60	76.08%	0.16 sec
90	76.08%	0.17 sec
120	76.08%	0.15 sec

Table 2. records the impact of varying the number of epochs on the model's performance metrics. All other hyperparameters are kept constant, same as that of the base model, with 10 neurons in the hidden layer.

Table 2. Epochs - Variable

Epochs	**Accuracy**	**Training Time**
10	60.86%	0.12 sec
40	60.86%	0.55 sec
80	63.04%	1.08 sec
140	65.23%	2.06 sec
200	63.04%	2.45 sec

Following observations are deduced from the tables above:

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

The following tables record the impact of varying the learning rates i.e., 'alpha' & 'beta', for the hidden/Kohnen and the output/Grossberg layer on the model's performance metrics. All other hyperparameters are kept constant (same as the base model), with the decay rate set to 0.5.

Table 3. Learning Rate (alpha) - Variable

Alpha	Beta	Accuracy	Training Time
0.001	0.01	60.87%	0.16 sec
0.01	0.01	60.87%	0.13 sec
0.05	0.01	73.91%	0.14 sec
0.1	0.01	65.22%	0.17 sec
0.5	0.01	58.70%	0.17 sec
0.9	0.01	58.70%	0.14 sec

Table 4. Learning Rate (beta) - Variable

Beta	Alpha	Accuracy	Training Time
0.001	0.01	51.87%	0.15 sec
0.01	0.01	60.87%	0.15 sec
0.05	0.01	65.23%	0.12 sec
0.1	0.01	76.09%	0.11 sec
0.5	0.01	76.09%	0.12 sec
0.9	0.01	76.09%	0.15 sec

Following observations are deduced from the table above:

- The learning rate is an important parameter which defines/quantifies how much change to introduce into the model based on the estimated error when adjusting weights during the network's training phase.
- The observation for each of the learning rate(s) is unique and is as described below:
- With large values of 'alpha', the network's capability to learn deteriorates and oscillates significantly when we increase the value.
- With small values of 'alpha', the model gets stuck at local minima and does not do well.
- Whilst with 'beta', we observe that the CPN learning capability increases until it stagnates when it reaches the steady-state value.
- With small values, the model's performance is poor since it is unable to best represent the data representation and most likely gets stuck at the numerous local minimums.
- The training times oscillate as values increase, with no definitive structure to them.

Table 5. records the impact of varying the decay rate hyperparameter on the model's performance metrics. All other hyperparameters are kept constant.

- Decay rate is a regularization hyperparameter used to control overfitting in models. It penalizes the weights during training to ensure better representation of the dataset is sought.
- With increase in the decay rate coefficient, the network's training time accelerates before slowing down for large values of the coefficient. Whilst accuracy improves for large values.

Table 5. Decay Rate – Variable

Decay Rate	Accuracy	Training Time
0.01	63.04%	0.18 sec
0.05	60.87%	0.16 sec
0.1	60.87%	0.12 sec
0.5	65.22%	0.12 sec
0.9	66.30%	0.14 sec

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 that of the model implemented, there may exist a more optimized/global optimum network for this problem with the given set of hyperparameters and a different CPN model.

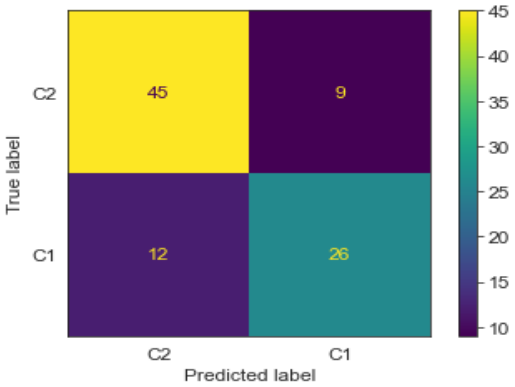
Hidden Layer Neurons	Kohnen Layer Learning Rate	Grossberg Layer Learning Rate	Decay Rate	Epochs	Shuffle
150	0.01	0.01	0.9	30	Yes

The following evaluation metrics were reported:

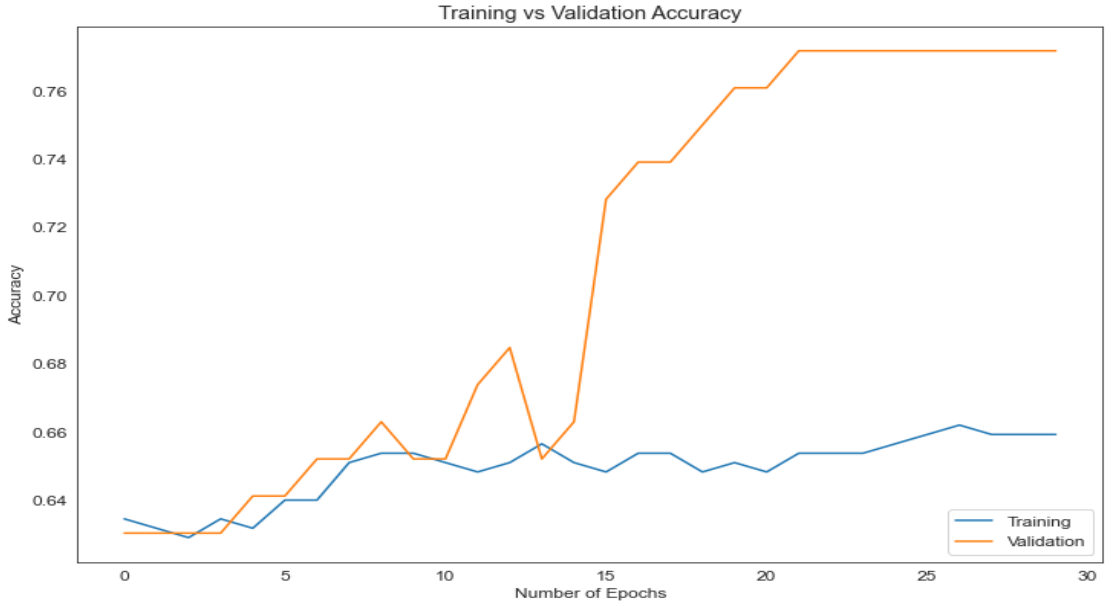
Accuracy	Training Time
78.26 %	1.17 sec

The confusion matrix characteristics are as follows:

- TP: 45 (48.91%)
- TN: 26 (28.26%)
- FN: 9 (9.78%)
- FP: 12(13.04%)
- Total instances: 92



In addition, the training vs validation curve for 'accuracy' is shown below. As one can observe, the model implemented has certain inefficiencies and is unable to generalize well on the dataset, resulting in it underfitting. However, the potential to improve certainly exists and will be explored further.



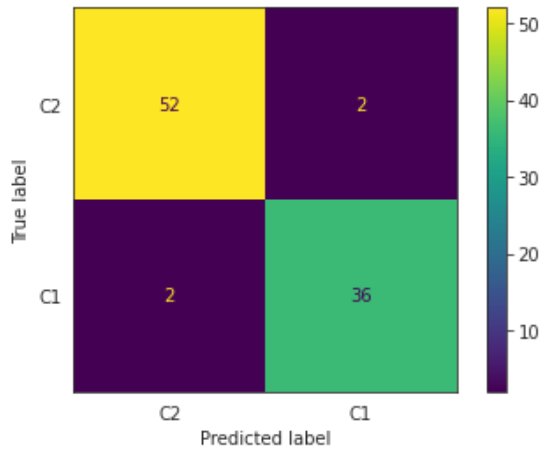
V. Conclusion and Comparison

We now look at a comparative analysis between the Back Propagation Network and the Counter Propagation Network on the problem at hand given the same dataset. We observe the optimal results produced by each model to objectively view their performance.

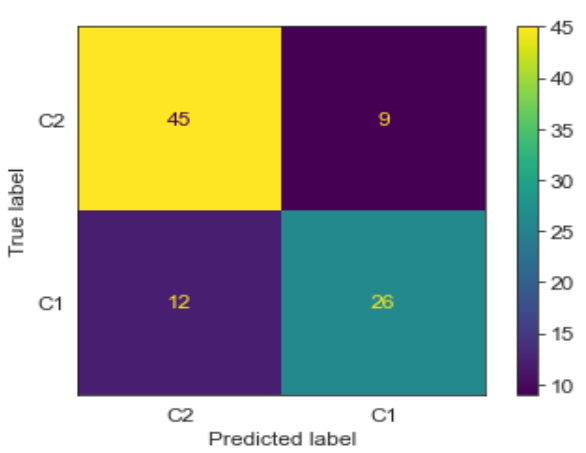
Table 6.

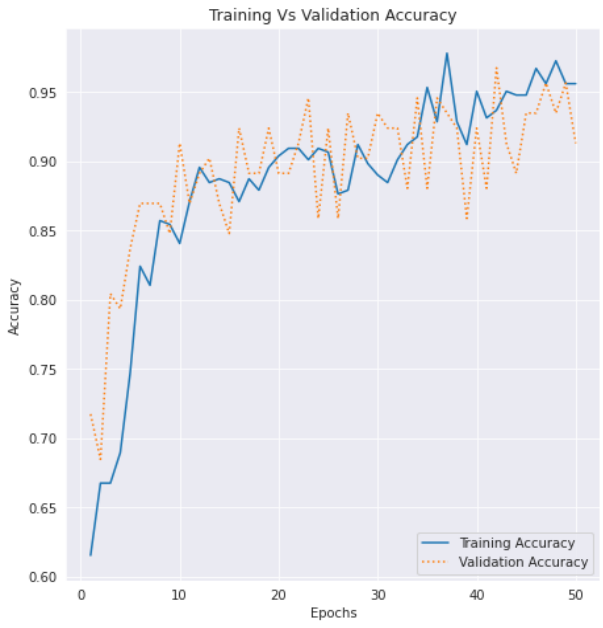
Model	**Accuracy**	**Training Time**
Back Propagation Network	95.65%	9.70 sec
Counter Propagation Network	78.26%	1.17 sec

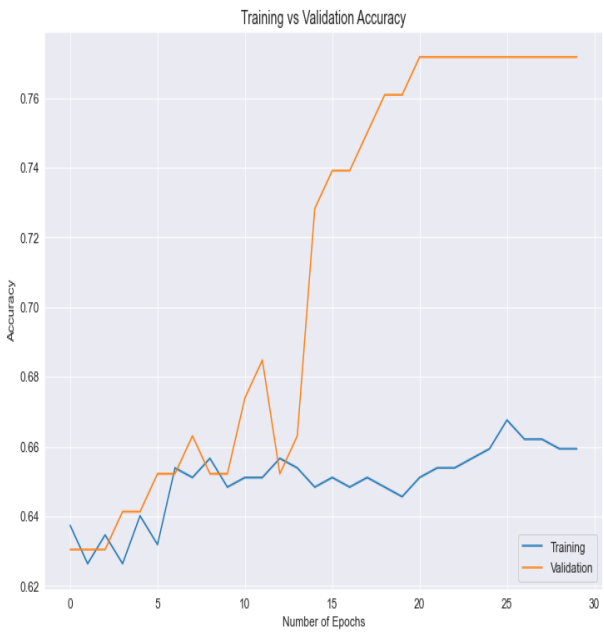
BPN



CPN







The following observations have been deducted:

- BPN performs thoroughly well depicted through the evaluation metrics.
- CPN is unable to generalize well and underfits as shown by the training vs validation accuracy curves, whilst BPN neither underfits nor overfits.
- The behavior of BPN can be termed as highly oscillatory, whereas with CPN it is less oscillatory and steadier.
- High ratio of FP and FN in CPN when compared to BPN.
- BPN is implemented with the aid of existing open-source libraries which have been optimized and refactored, thus the room for error is minimal.
- CPN has been implemented from scratch and may not be the optimized version of the model, attributing to the inaccuracy.
- BPN makes use of the ReLu activation function whilst CPN makes use of the Hebbian activation for the hidden layer neurons.
- CPN is a lot faster in terms of the training time taken.
- CPN is also less complex than BPN due to the presence of only three layers, whereas the optimal BPN has 5 layers.

I now list a few advantages and disadvantages of using CPN for the problem at hand.

Advantages	**Disadvantages**
Ability to perform unsupervised learning. This enables the functioning of the model even with unlabeled data.	Limited flexibility in terms of the model architecture, thus cannot be easily adapted to other tasks.
Ability to handle data quickly resulting in fast processing.	Prone to overfitting, resulting in poor generalization to new data.
Lower complexity due to the small number of layers.	Prone to underfitting, due to simplicity of the model and scarcity of the dataset.
Ability to perform back-mapping, resulting in a look-up table.	
Reduces dimensionality of the dataset, simplifying the analysis.	

VI. Appendix

```
#Importing requisite libraries
import pandas as pd
import numpy as np
import time
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.utils import shuffle
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
accuracy_score

#Dataset Generation
datapoints = pd.read_csv('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)

label_region = LabelEncoder()
coordinates['region'] = label_region.fit_transform(coordinates['region'])
scaler = MinMaxScaler()
coordinates[['x','y']] = scaler.fit_transform(coordinates[['x','y']])

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

```
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
class CounterpropagationNetwork:
    def __init__(self, input_size, hidden_size, output_size):

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights
        self.hidden_weights = np.random.rand(hidden_size, input_size)#Kohnen
Layer
        self.output_weights = np.random.rand(output_size,
hidden_size)#Grossberg Outstar Layer

    def train(self, X_train, y_train, X_test, y_test, num_epochs, alpha,
beta, decay_rate):
        history = {'epoch':[], 'training_accuracy': [],
'validation_accuracy':[] }
        for epoch in range(num_epochs):
            predictions = []
            for i in range(len(X_train)):

                # Forward pass
                hidden_activations =
self._compute_hidden_activations(X_train[i])
                output_activations =
self._compute_output_activations(hidden_activations)

                # Backward pass
                output_errors = y_train[i] - output_activations
                hidden_errors =
self._compute_hidden_errors(hidden_activations, output_errors)

                # Update weights
                self._update_output_weights(hidden_activations,
output_errors, beta)
                self._update_hidden_weights(X_train[i], hidden_errors, alpha)

                predictions = np.append(predictions,output_activations)

            # Learning rates diminishes after every epoch
```

```
            alpha*= decay_rate
            beta*= decay_rate
            train_accuracy =
accuracy_score(y_train,np.where(predictions>0.5,1,0))
            val_accuracy = accuracy_score(y_test,
np.where(self.predict(X_test)>0.5,1,0))
            history["epoch"].append(epoch)
            history["training_accuracy"].append(train_accuracy)
            history["validation_accuracy"].append(val_accuracy)

        return history

    def predict(self, X):
        predictions = []
        for i in range(len(X)):
            hidden_activations = self._compute_hidden_activations(X[i])
            output_activations =
self._compute_output_activations(hidden_activations)
            predictions = np.append(predictions,output_activations)
        return predictions


    def _compute_hidden_activations(self, x):
        return np.dot(self.hidden_weights, x)

    def _compute_output_activations(self, h):
        return np.dot(self.output_weights, h)

    def _compute_hidden_errors(self, h, output_errors):
        return np.multiply(np.dot(self.output_weights.T, output_errors),
np.multiply(h, 1 - h))

    def _update_hidden_weights(self, x, hidden_errors, alpha):
        delta_w = alpha * np.outer(hidden_errors, x)
        self.hidden_weights += delta_w

    def _update_output_weights(self, hidden_activations, output_errors,
beta):
        delta_w = beta * np.outer(output_errors, hidden_activations)
        self.output_weights += delta_w
```

```
if __name__ == '__main__':
    cpn = CounterpropagationNetwork(input_size=2, hidden_size = 10,
output_size=1)
    start_time = time.time()
    history = pd.DataFrame(cpn.train(X_train, y_train, X_test, y_test, 10,
0.01,0.01, 0.9))
    end_time = time.time()
    print("Training time: %.2f " %(end_time - start_time))
    print(history)

#Model Evaluation
y_pred = cpn.predict(X_test)
y_pred = np.where(y_pred>0.5,1,0)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:",accuracy)

# Accuracy Plot
plt.figure(figsize=(12,8))
plt.plot(history['epoch'], history['training_accuracy'])
plt.plot(history['epoch'], history['validation_accuracy'])
plt.xlabel('Number of Epochs')
plt.ylabel('Accuracy')
plt.title('Training vs Validation Accuracy')
plt.legend(["Training", "Validation"], loc ="lower right")
plt.show()

#Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["C2",
"C1"])
sns.set_style("white")
plt.rc('font', size=12)
disp.plot()
plt.show()
```