COEN 6331: Neural Networks

ASSIGNMENT -2

Submitted by:

Rohan Kodavalla (40196377)

I certify that this submission is my original work and meets the Faculty's Expectations of Originality

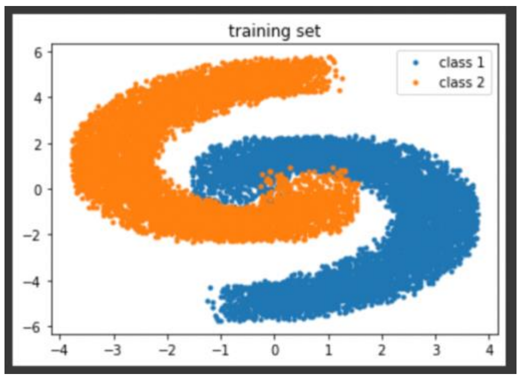
February 24, 2022

ABSTRACT

The objective of this assignment is to study a two-dimensional classification problem that involves nonconvex decision regions via Counter Propagation Network (CPN).

1. Generating two non-convex regions for training:

In order to generate the pattern above in the question a neural network architecture is designed with Keras, that will be used as the data to train the network.



As shown above data obtained from pattern is the dataset that contains two different spirals. Now from this data, choosing the samples in each of the regions, it is possible to train them and study CPN mechanism.

2. Selecting samples for the non-convex regions:

In this case, the number of samples selected are 7000 (random value) that is appended to value 'n_points' in the code.

The pattern generated is appended to a variable 'n' and from this 'n' value, the data sets for classes are derived and named 'd1x' & 'd1y' respectively in the code.

Further, to generate the datasets for training, all the points belonging to the pattern are normalized. But this normalization is done by arranging the data values in 'np.hstack', 'np.vstack' i.e., arranging horizontally and vertically in array stacks (just for simplification of coding process).

counter_propagation.py config.py generator_distributions.py distributions.py

```python
import numpy as np
from sources import generator_distributions,distributions
# famous data
P_trust = 0.95
t_critical_student = 2.26
number_drills = 20
number_stoutnes = 24

# for hi_square
decrease_of_freedom = number_stoutnes - 1
level_main = 0.05
hi_critical_23_005 = 35.5  # 23 and 0.05
hi_critical_11_005 = 19.7

n_points=7000
noise=1.5

n = np.sqrt(np.random.rand(n_points,1)) * 400 * (1.10*np.pi)/290
d1x = -np.cos(n)*n + np.random.rand(n_points,1) * noise -1
d1y = np.sin(n)*n + np.random.rand(n_points,1) * noise -1

# X_values data
drills = np.vstack((np.hstack((d1x,d1y)),np.hstack((-d1x,-d1y)))) ### combiing 2 rows
#print(d1x[0:5])
#drills=[item for sublist in drills_ for item in sublist]
# generated data from function generate_normal_distribution()
#normal_distribution = generator_distributions.generate_normal_distributions(batch = len(d1x), size = len(d1x))
#normal_distribution=distributions.generate_normal_distribution(size=len(d1x))
normal_distribution= []
# result from comparison with data from table and generated normal distribution.
# every number belong to data from table.
# 1 - distribution is normal
# 0 - distribution isn't normal
#print(drills[0:5],d1y[0:5])
y_train = np.hstack((np.zeros(n_points),np.ones(n_points))) #### normalization is happening here,, combing 2 cols
```

Code for configuring data sets values

counter_propagation.py config.py generator_distributions.py distributions.py

```python
'''
    Counter propagation neural network.
'''
import numpy as np
from sources.generator_distributions import mix_distributions, generate_normal_distributions, generate_expon_distributions, get_y_train
from sources import config
from math import sqrt
import time

X_values = config.drills
y_values = config.y_train

def sum_squares(arr):
    return sum([x ** 2 for x in arr])

# function normalization for vectors X
def normalization(arr_x, arr_y):
    sum_y_squares = sum_squares(arr_y)
    result = []

    for row_x in arr_x:
        middle_result = []
        sum_x_squares = sum_squares(row_x)
        for x in row_x:
            middle_result.append(round(x / sqrt(sum_x_squares + sum_y_squares), 2))

        result.append(middle_result)
    return result
```

Normalization

3. CPN weights, Kohonen & Grossberg Neurons:

a) Feeding the pattern data set as input to the learning of CPN network –

The pattern obtained in initial phase is now translated into values stored in d1x, d1y and are appended in an array to use it as a vector.

b) Selection of Kohonen, Grossberg Neurons-

In this case, number of Kohonen neurons chosen are 2 , and Grossberg neurons are 1.

c) Generating random weights-

Initially the weights for these 2 layers are chosen randomly, based on length of the vector (set to 4 in our example). These weights are denoted as 'kohonen_weights' and 'grossberg_weights' respectively.

```python
class Counter_Propagation:
    def __init__(self, X_values, y_values, kohonen_neurons = 2, grossberg_neurons = 1, len_x_vector = 4):
        self.X_values = X_values
        self.y_values = y_values
        self.kohonen_neurons = kohonen_neurons
        self.grossberg_neurons = grossberg_neurons
        self.kohonen_weights = self.generate_weights(kohonen_neurons, len_x_vector)
        self.grossberg_weights = self.generate_weights(grossberg_neurons, len_x_vector)

    # generate weights for each part of network
    def generate_weights(self, num_neurons = 1, length=4):
        yy = np.random.rand(num_neurons, length)
        result = np.asarray(yy)
        print("Generating Random weights------", result)

        if len(result) == 1:
            return result[0]
        return result

```

Code for b), c) points above

d) Computing Euclidean distance-

Firstly, in order to compute this, the total number of samples in this case was chosen to be 7000 for which each of the samples in vector compute the shortest distance to given 2 neurons in Kohonen layer and 1 neuron in Grossberg layer.

The output of above (Euclidean distance) is stored in an array which generates almost 5000 values, and the winning node is printed corresponding to layer.

Post this the vectors of weights are updated with respect to the layer.

```python
    # for shorter Evklide's way
    def calculate_evklid_way(self, w_vector, x_vector):
        zz= sum([ ((w-x) ** 2) for w, x in zip(w_vector, x_vector)])
        return zz


    # calculate net for Grossberg lay
    def sum_activation(self, k_vector, w_vector):
        return sum([ k*w for k, w in zip(k_vector, w_vector)])

    # update vector kohonen weights
    def update_kohonen_weights(self, x_vector, w_vector, learning_rate = 0.7):
        weights = []

        for x, w in zip(x_vector, w_vector):
            w_new = w + learning_rate * (x - w)
            weights.append(w_new)
        return np.asarray(weights)

    # update weights for grossberg lay
    def update_grossberg_weights(self, y_value, w_value, learning_rate = 0.1, k = 1):
        w_new = w_value + learning_rate * (y_value - w_value) * k
        #print(f'grossberg update: {w_value}, {w_new}')
        return w_new

```

Code for d) point

e) Training counter propagation neural network-

The parameters chosen for this process is: - learning rate for kohonen layer , learning rate for Grossberg layer, epochs. Now the training of CPN network is done from below code snippets.

```python
    # training counter propagation neural network
    def fit(self, lr_kohonen = 0.7, lr_grossberg = 0.1, epochs = 10):
        print("kohonen_neurons used-----", self.kohonen_neurons, " || grossberg_neurons used------",self.grossberg_neurons )
        for epoch in range(epochs):
            if epoch % 5 == 0 and lr_kohonen > 0.1 and lr_grossberg > 0.01:
                lr_kohonen-= 0.05
                lr_grossberg -= 0.005

            good_counter = 0
            for x_vector, y_value in zip(self.X_values, self.y_values):
                kohonen_neurons = []

                for w_iter in range(len(self.kohonen_weights)):
                    kohonen_neurons.append(self.calculate_evklid_way(x_vector, self.kohonen_weights[w_iter]))

                neuron_min = min(kohonen (method) index: (__value: Any, __start: SupportsIndex = ..., __stop: SupportsIndex = ...) -> int
                index = kohonen_neurons.index(neuron_min)

                for i in range(len(kohonen_neurons)):
                    if i == index:
                        kohonen_neurons[i] = 1
                    else:
                        kohonen_neurons[i] = 0

                self.kohonen_weights[index] = self.update_kohonen_weights(x_vector, self.kohonen_weights[index], learning_rate= lr_kohonen)

                # grossberg neurons
                #print("index -", index)
                self.grossberg_weights[index] = self.update_grossberg_weights(y_value, self.grossberg_weights[index], learning_rate=lr_grossberg)
                grossberg_neuron_out = int(round(self.sum_activation(kohonen_neurons, self.grossberg_weights)))
                good_counter += self.good_count(y_value, grossberg_neuron_out)

                #print(f'{y_value} : {grossberg_neuron_out}')

            print(f'Success training {epoch+1} epoch: {int(good_counter/len(self.y_values) * 100)}%')

```

```python
    # work network on test values
    def evaluate(self, X_values, y_values):
        self.X_values = X_values
        self.y_values = y_values
        rr =[]
        rr1=[]
        good_counter= 0
        for x_vector, y_value in zip(self.X_values, self.y_values):
            kohonen_neurons = []

            for w_iter in range(len(self.kohonen_weights)):
                kohonen_neurons.append(self.calculate_evklid_way(x_vector, self.kohonen_weights[w_iter]))

            neuron_min = min(kohonen_neurons)
            index = kohonen_neurons.index(neuron_min)
            rr.append(neuron_min)
            rr1.append(index)
            for i in range(len(kohonen_neurons)):
                if i == index:
                    kohonen_neurons[i] = 1
                else:
                    kohonen_neurons[i] = 0

            grossberg_neuron_out = int(round(self.sum_activation(kohonen_neurons, self.grossberg_weights)))
            good_counter += self.good_count(y_value, grossberg_neuron_out)

        print(f'Success evaluate: {int(good_counter/len(self.y_values) * 100)}%')
        print("Shortest Euclidean Distance for each Training value------", rr)
        print("Winning node for each Training value------", rr1)
```

```python
# work neural network
if __name__ == '__main__':
    #print(X_values[0:5],y_values[0:5],y_values[-5:-1])\


    net = Counter_Propagation(X_values, y_values, kohonen_neurons=2, grossberg_neurons=1, len_x_vector=len(X_values[0]))

    t_start = time.perf_counter()
    lrk=0.7
    lrg= 1
    print("Learning rate for Kohonen ----", lrk , " || Learning rate for Grossberg----" , lrg)
    net.fit(lr_kohonen=0.7, lr_grossberg=1, epochs=5)
    t_stop = time.perf_counter()

    print(f"Time of fit: {round(t_stop - t_start, 3)}")

    # testing on synthetic values
    normal_distr = generate_normal_distributions(1000, 24)
    expon_distr = generate_expon_distributions(450, 24)

    X_test = mix_distributions(normal_distr, expon_distr)
    y_test = get_y_train(X_test)

    print("Evaluate")
    net.evaluate(X_test, y_test)
```

f) Studying effects of learning rate, epochs for training accuracy and winning nodes :

Number of Kohonen neurons	Number of Grossberg neurons	Learning rate for Kohonen layer	Learning rate for Grossberg layer	Epochs	Time taken (min)	Training Accuracy	Winner nodes count for 0^{th} & 1^{st} Neuron
2	1	0.7	1	5	01:316	99%	0- 1000 1- 450
2	1	0.7	0.1	5	01:198	100%	0- 998 1- 452
2	1	1.4	1	10	2:52	99%	1-1450

2	1	0.1	2	10	2.584	99%	0- 1000 1- 450

So overall, in this case we observe that for a same number of neurons, time taken for 5 and 10 epochs brings no change in training accuracy. Varying the learning rate for both layers only affects the number of winning nodes for the layers by a small amount.

So overall, maintaining the maximum accuracy and by varying learning rates for fixed number of neurons in Kohonen layer and Grossberg layer, the effects and working of a Counter Propagation Network CPN is studied.

APPENDIX: OUTPUTs // CODE

Generating data set –

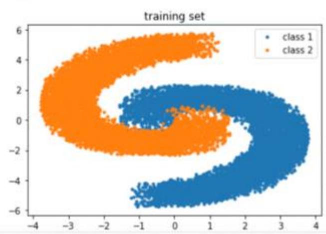
```python
import numpy as np
import matplotlib.pyplot as plt

def twospirals(n_points, noise=1.5):
    """
     Returns the two spirals dataset.
    """
    n = np.sqrt(np.random.rand(n_points,1)) * 400 * (1.10*np.pi)/290
    d1x = -np.cos(n)*n + np.random.rand(n_points,1) * noise -1
    d1y = np.sin(n)*n + np.random.rand(n_points,1) * noise -1
    return (np.vstack((np.hstack((d1x,d1y)),np.hstack((-d1x,-d1y)))),
            np.hstack((np.zeros(n_points),np.ones(n_points))))

X, y = twospirals(7000)

plt.title('training set')
plt.plot(X[y==0,0], X[y==0,1], '.', label='class 1')
plt.plot(X[y==1,0], X[y==1,1], '.', label='class 2')
plt.legend()
plt.show()

#generating data set
```



Code output -1 :

```
!python3 /content/Neural_Networks/counter_propagation.py
```

```
Generating Random weights------ [[0.10451289 0.27258738]
 [0.42827559 0.00331772]]
Generating Random weights------ [[0.78307658 0.07353826]]
Learning rate for Kohonen ---- 0.7  || Learning rate for Grossberg---- 1
kohonen_neurons used----- 2  || grossberg_neurons used------ 1
Success training 1 epoch: 100%
Success training 2 epoch: 100%
Success training 3 epoch: 100%
Success training 4 epoch: 100%
Success training 5 epoch: 100%
Time of fit: 1.316
Evaluate
Success evaluate: 100%
Shortest Euclidean Distance for each Training value------ [7.0725088636247015, 452.2080374625433, 511.48328711717915, 538.5469875750558, 7.072
Winning node for each Training value------ [1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
```

```python
[22] import numpy as np
a = np.array([1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1
unique, counts = np.unique(a, return_counts=True)
dict(zip(unique, counts))
```

```
{0: 1000, 1: 450}
```

Code output -2 :

```
!python3 /content/Neural_Networks/counter_propagation.py
```

```
Generating Random weights------ [[0.43753171 0.24688073]
 [0.79827069 0.54397452]]
Generating Random weights------ [[0.14482283 0.12450256]]
Learning rate for Kohonen ---- 0.7  || Learning rate for Grossberg---- 0.1
kohonen_neurons used----- 2  || grossberg_neurons used------ 1
Success training 1 epoch: 99%
Success training 2 epoch: 99%
Success training 3 epoch: 99%
Success training 4 epoch: 99%
Success training 5 epoch: 99%
Time of fit: 1.198
Evaluate
Success evaluate: 100%
Shortest Euclidean Distance for each Training value------ [482.43719246539933, 521.4448656117901, 528.0170317224772, 381.7258889739552, 488.38
Winning node for each Training value------ [0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1,
```

```
[23] [0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0,

import numpy as np
a = np.array([0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1
unique, counts = np.unique(a, return_counts=True)
dict(zip(unique, counts))
```

```
{0: 998, 1: 452}
```

Code output -3 :

```
[8] !python3 /content/Neural_Networks/counter_propagation.py
```

```
Generating Random weights------ [[0.92550424 0.03577393]
 [0.56863496 0.29513009]]
Generating Random weights------ [[0.23586267 0.89567496]]
Learning rate for Kohonen ---- 1.4  || Learning rate for Grossberg---- 1
kohonen_neurons used----- 2  || grossberg_neurons used------ 1
Success training 1 epoch: 99%
Success training 2 epoch: 99%
Success training 3 epoch: 99%
Success training 4 epoch: 99%
Success training 5 epoch: 99%
Success training 6 epoch: 99%
Success training 7 epoch: 99%
Success training 8 epoch: 99%
Success training 9 epoch: 99%
Success training 10 epoch: 99%
Time of fit: 2.52
Evaluate
Success evaluate: 100%
Shortest Euclidean Distance for each Training value------ [463.46378153047146, 19.769211374042282, 458.3508770353984, 479.70528450649624, 19.7
Winning node for each Training value------ [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

import numpy as np
a = np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
unique, counts = np.unique(a, return_counts=True)
dict(zip(unique, counts))
```

```
{1: 1450}
```

Code output -4:

```
!python3 /content/Neural_Networks/counter_propagation.py
```

```
Generating Random weights------ [[0.02824559 0.65261106]
 [0.19534586 0.00488573]]
Generating Random weights------ [[0.30654624 0.14717478]]
Learning rate for Kohonen ---- 0.1  || Learning rate for Grossberg---- 2
kohonen_neurons used----- 2  || grossberg_neurons used------ 1
Success training 1 epoch: 99%
Success training 2 epoch: 99%
Success training 3 epoch: 99%
Success training 4 epoch: 99%
Success training 5 epoch: 99%
Success training 6 epoch: 99%
Success training 7 epoch: 99%
Success training 8 epoch: 99%
Success training 9 epoch: 99%
Success training 10 epoch: 99%
Time of fit: 2.584
Evaluate
Success evaluate: 100%
Shortest Euclidean Distance for each Training value------ [17.3115775475725, 319.1260375394163, 17.3115775475725, 260.8723078472197, 329.61607
Winning node for each Training value------ [1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0,
```

```
[19] import numpy as np
a = np.array([1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0
unique, counts = np.unique(a, return_counts=True)
dict(zip(unique, counts))
```

```
{0: 1000, 1: 450}
```