1 Brief description of the algorithm

In this task, to invite the consideration of neighbourhood, I used the learning rule following equation 10.17 for 10 epochs, updating the learning rule and width at every iteration while the function goes through all the iris data. The total number of iteration is set to 1000.

2 Results

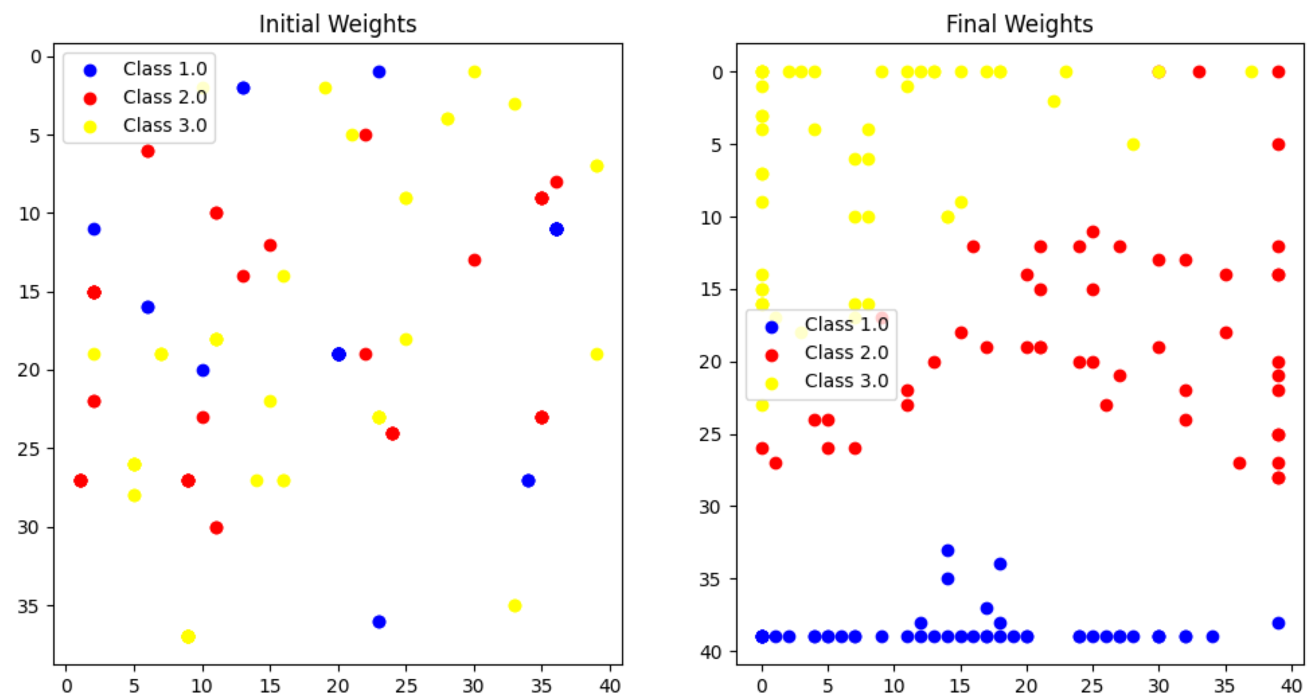

Figure 1: Example Test of Best Chromosome on Slope 5, Test Dataset

Here is the output where the weights shows 3 clusters eventually, with class 1 clearly separated while 2 and 3 a bit mixed at the corner. With an exponential increase on the number of iteration, no visible improved shows, meaning the corner ones could share similarity thus, causes difficulty to distinguish. Since no difference is found comparing introduce permutation in the iris data or not, I assume the original data has already possessed the property of stochastic.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
#Standardise
iris_label = pd.read_csv('iris-labels.csv')
iris_data = pd.read_csv('iris-data.csv')
maximal = iris_data.max()

dataS = iris_data.divide(maximal, axis=1)
labels = iris_label.iloc[:, 0].values
```

```
labels = iris_label.iloc[:, 0].values
```

```
#constants
batch_size=1
nEpoch = 10
eta_0 = 0.1
d_eta = 0.1
sigma_0 = 10
d_sigma = 0.05
output_size = (40,40)
feature = 4

nIterate = 100  #tune
```

```
def initializeW(output_size,feature):
    w = np.random.uniform(0,1,(output_size[0],output_size[1],feature))
    return w

def shuffle(data):
    dataS = data.sample(frac=1).reset_index(drop=True)
    return dataS

def find_winner(x,w):
    x_reshaped = x.reshape(-1)
    x_broadcasted = x_reshaped[np.newaxis, np.newaxis, :]

    distances = np.linalg.norm(w-x_broadcasted,axis=2)
    i0= np.unravel_index(np.argmin(distances,axis=None),distances.shape)

    return i0

def learn(j,k,h,eta,x,w):
    dw = eta * h * (x-w[j,k,:])
    return dw

def neighbour(ri,sigma,r0):
    dist = np.sum(np.square(np.array(r0) - np.array(ri)))
    s = -1/(2* (sigma ** 2))
    h = np.exp(s * dist )
    return h

def iterate(index0,dindex,epoch):
    index = index0 * np.exp(-dindex * epoch)
    return index

def get_winner_positions(data, weights):
    return [find_winner(x, weights) for x in data.values]

def shuffle(data, labels):
    combined = pd.concat([data, pd.DataFrame(labels, columns=['label'])], axis=1)
    shuffled = combined.sample(frac=1).reset_index(drop=True)
    data_shuffled = shuffled.drop(columns=['label'])
    labels_shuffled = shuffled['label'].values
    return data_shuffled, labels_shuffled
```

```
p = len(dataS)
w = initializeW(output_size,feature)
```

```
for epoch in range(1,nEpoch):
    #initialize
    #dataS, labels = shuffle(dataS, labels)
```

```
        #index update
        eta = iterate(eta_0,d_eta,epoch)
        sigma = iterate(sigma_0,d_sigma,epoch)

        for i in range(p):
            x = dataS.iloc[i, :].values
            r_winner = find_winner(x,w)

            for j in range(output_size[0]):
                for k in range(output_size[1]):
                    ri = (j, k)
                    h = neighbour(ri,sigma,r_winner)

                    dw = learn(j,k,h,eta,x,w)
                    w[j, k, :] += dw


    #plot
    final_w = w.copy()

    initial_winners = get_winner_positions(dataS, initial_w)
    final_winners = get_winner_positions(dataS,final_w)

    initial_x, initial_y = zip(*initial_winners)
    final_x, final_y = zip(*final_winners)

    fig, ax = plt.subplots(1,2,figsize=(12,6))

    scatter = ax[0].scatter(initial_x,initial_y,c=labels,cmap='viridis')
    ax[0].set_title('Initial Weights')
    ax[0].invert_yaxis()


    ax[1].scatter(final_x, final_y, c=labels, cmap='viridis')
    ax[1].set_title('Final Weights')
    ax[1].invert_yaxis()


    plt.show()
```