Hw2 Report

1. Describe your model architecture details and PCA method

總共有兩個 model，第一個 model 為 2 layer FNN，輸入層與輸出層為固定的，hidden layers 為 512 個，然後根據助教的 coding，採用固定的 seed，loss function 採用 cross-entropy function，Update 的方式為 SGD，根據網路上對於 SGD 的定義[1]，其定義為要一筆一筆的輸入，有鑑於此，我也是使用每輸入一筆資料，就做一次更新，並且設計了 epoch，每一個 epoch 必須將 1470 筆資料訓練一次，最後是 forward pass and backward pass function 展示(兩個模型都有)，預設為訓練 3 次
兩個模型一樣的地方，都是使用 Relu function 作為隱藏層的激勵函數，並且在輸出層使用 softmax，將輸出值改成[0,1]之間。

```
def relu(self,X):

    for i in range(0,X.shape[0]):
        if(X[i] <= 0):
            X[i] = 0
    return X

def softmax2(self,z3):
    ans = []

    denom = sum(np.exp(z3))
    for i in range(0,z3.shape[0]):
        ans.append(np.exp(z3[i])/denom)
    ans = np.array(ans)

    return ans
```

並且計算 relu gradient，用來計算 backward pass

```
def relu_prime(self,z):
    ans = z;
    for j in range(0,z.shape[0]):
        if ans[j] <=0:
            ans[j] = 0
        else:
            ans[j] = 1
    return ans
```

Train, predict, score function 都是一樣的

```
def train(self, X, epoch=3):
    tot_loss = []
    for _ in range(0,epoch):
        for i in range(0,1470):
            y_hat = self.forward_pass(X[i,0:2])
            loss = self.loss(y_hat, X[i,2])
            self.backward_pass(y_hat,X[i,0:2],X[i,2])
            self._update()
            tot_loss.append(loss)
    return np.array(tot_loss)

def predict(self, X):
    ans = []
    for i in range(0
        y_hat = self.forward_pass(X[i,0:2])
        ans.append(np.argmax(y_hat))
    return np.array(ans)

def score(self, predict, y):
    cnt = np.sum(predict==y)
    return (cnt/len(y))*100
```

```
Built-in mutable sequence.

If no argument is given, the construc
The argument must be an iterable if s

Click anywhere in this tooltip for ad
```

初始設定

```
def __init__(self):
    self.input = 2 # feature numbers
    self.output = 3 # class number
    self.hidden_units = 512 # single layer
    seed = 123
    np.random.seed(seed)
    random.seed(seed)
    self.w1 = np.random.randn(self.input, self.hidden_units)*0.01
    self.w2 = np.random.randn(self.hidden_units, self.output) *0.01
    self.b1 = np.zeros(self.hidden_units)
    self.b2 = np.zeros(self.output)
```

Cross-entropy loss，並且設定 if，防止出現零的值，log 運算才不會出錯

```
#CEL
def loss(self,y_hat, y):

    log_probs= 0;

    if y_hat[int(y)] == 0:
        y_hat[int(y)] = 10**-10
    log_probs = -np.log2(y_hat[int(y)])
    loss = log_probs

    return loss
```

2 layers model update function

```
def _update(self, learning_rate=0.01):
    self.w1 = self.w1 - learning_rate*self.dw1
    self.b1 = self.b1 - learning_rate*self.db1
    self.w2 = self.w2 - learning_rate*self.dw2
    self.b2 = self.b2 - learning_rate*self.db2
```

2 layers model forward pass function

```
def backward_pass(self,y_pre, X, y):
    delta3 = y_pre
    delta3[int(y)] -= 1
    self.dw2 = np.dot(self.a2.reshape(self.hidden_units,1),delta3.reshape(3,1).T
    self.db2 = np.sum(delta3,axis = 0)

    self.dz1 = np.dot(delta3,self.w2.T) * self.relu_prime(self.a2)
    self.dw1 = np.dot(X.reshape(2,1),self.dz1.reshape(self.hidden_units,1).T)
    self.db1 = np.sum(self.dz1,axis = 0)
```

2 layers model backward pass function

```
def forward_pass(self, X):

    self.z2 = np.dot(X, self.w1)+self.b1
    self.a2 = self.relu(self.z2)
    self.z3 = np.dot(self.a2, self.w2) + self.b2
    self.a3 = self.softmax2(self.z3)

    return self.a3
```

第二個 model 為 3 layers FNN，第一層隱藏層數量為 512，下一層為 128，為何會這樣做，是因為之前在使用 NN 的時候，總是會說越靠近輸出層的，要把 Node 數量降下來，以免在傳遞的時候，因為 Node 數量差距過大，丟失一些資訊，所以才採用這樣的方式，其更新的方式如同前一個，都是使用一筆資料進行一次更新，也一樣設定了 epoch，讓每一筆資料都可以訓練到。

初始設定

```
def __init__(self):
    self.input = 2 # feature numbers
    self.output = 3 # class number
    self.hidden_units_1 = 512 # single layer
    self.hidden_units_2 = 128

    self.seed = 123
    np.random.seed(self.seed)
    random.seed(self.seed)
    self.w1 = np.random.randn(self.input, self.hidden_units_1) *0.01
    self.w2 = np.random.randn(self.hidden_units_1, self.hidden_units_2) *0.01
    self.w3 = np.random.randn(self.hidden_units_2, self.output)  *0.01
    self.b1 = np.zeros(self.hidden_units_1)
    self.b2 = np.zeros(self.hidden_units_2)
    self.b3 = np.zeros(self.output)
```

3 layers update function

```
def _update(self, learning_rate=0.01):
    # SGD
    self.w1 = self.w1 - learning_rate*self.dw1
    self.b1 = self.b1 - learning_rate*self.db1
    self.w2 = self.w2 - learning_rate*self.dw2
    self.b2 = self.b2 - learning_rate*self.db2
    self.w3 = self.w3 - learning_rate*self.dw3
    self.b3 = self.b3 - learning_rate*self.db3
```

3 layers model forward pass function

```
def forward_pass(self, X):


    self.z1 = np.dot(X, self.w1) + self.b1
    self.a1 = self.relu(self.z1)
    self.z2 = np.dot(self.a1, self.w2) + self.b2
    self.a2 = self.relu(self.z2)
    self.z3 = np.dot(self.a2, self.w3) + self.b3
    self.a3 = self.softmax2(self.z3)

    return self.a3
```

3 layers model backward pass

```
def backward_pass(self,y_pre, X, y):
    delta3 = y_pre

    delta3[int(y)] -= 1
    self.dw3 = np.dot(self.a2.reshape(self.hidden_units_2,1),delta3.reshape(3,1).T)
    self.db3 = np.sum(delta3,axis = 0)

    self.dz2 = np.dot(delta3,self.w3.T) * self.relu_prime(self.a2)
    self.dw2 = np.dot(self.a1.reshape(self.hidden_units_1,1),self.dz2.reshape(self.hidden_units_2,1).T)
    self.db2 = np.sum(self.dz2,axis = 0)

    self.dz1 = np.dot(self.w2,self.dz2) * self.relu_prime(self.a1)
    self.dw1 = np.dot(X.reshape(2,1),self.dz1.reshape(self.hidden_units_1 ,1).T)
    self.db1 = np.sum(self.dz1,axis = 0)
```

將資料藉由初始權重傳遞到後面，到輸出層後，進行 backward pass，算其 cost function 對 weight 的微分，然後乘上 learning rate 進行 weight 與 bias 的更新，一筆資料進行一次更新，就可以訓練模型。

PCA 的方法，我對於這裡的理解，我是先將 train 資料輸入並收集成一個矩陣，對其做 PCA，PCA(n_component = 2)表示取最大的前兩個主軸(具有各項資料的最大變異數)，並且將訓練資料與測試資料投影在上面，就可以開始進行模型的訓練與預測。

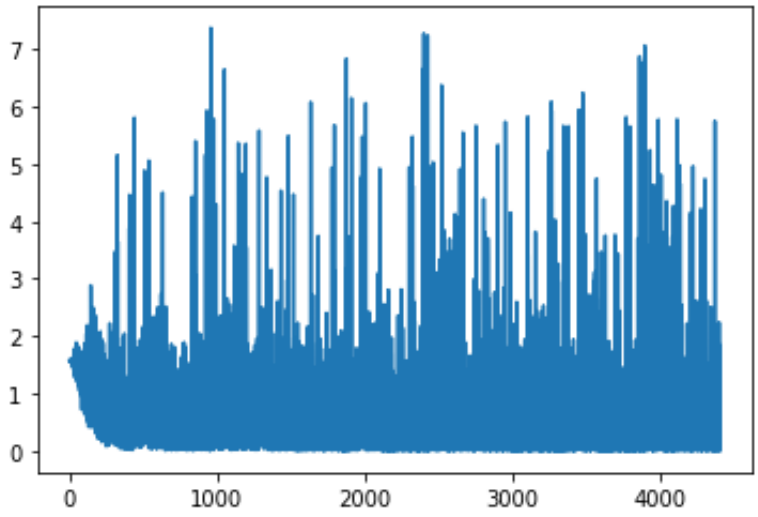
2. Show your test accuracy.
   會放上 training data 的準確度和 testing data 的準確度

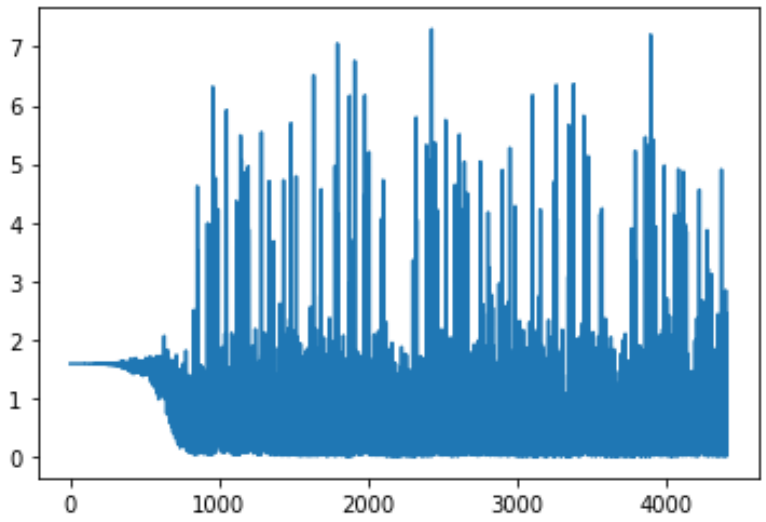
```
2 layer train score:  85.37414965986395
2 layer test score:  83.33333333333334
3 layer train score:  77.9591836734694
3 layer test score:  78.3132530120482
```

   可以看出 test score 準確度極高，兩層的準確度比三層的還高一點。

3. Plot training loss curves

我使用的方法是，因為每一筆資料都會做一次更新，所以都會生成一個 loss，將所有的資料收集起來，並且畫出來。



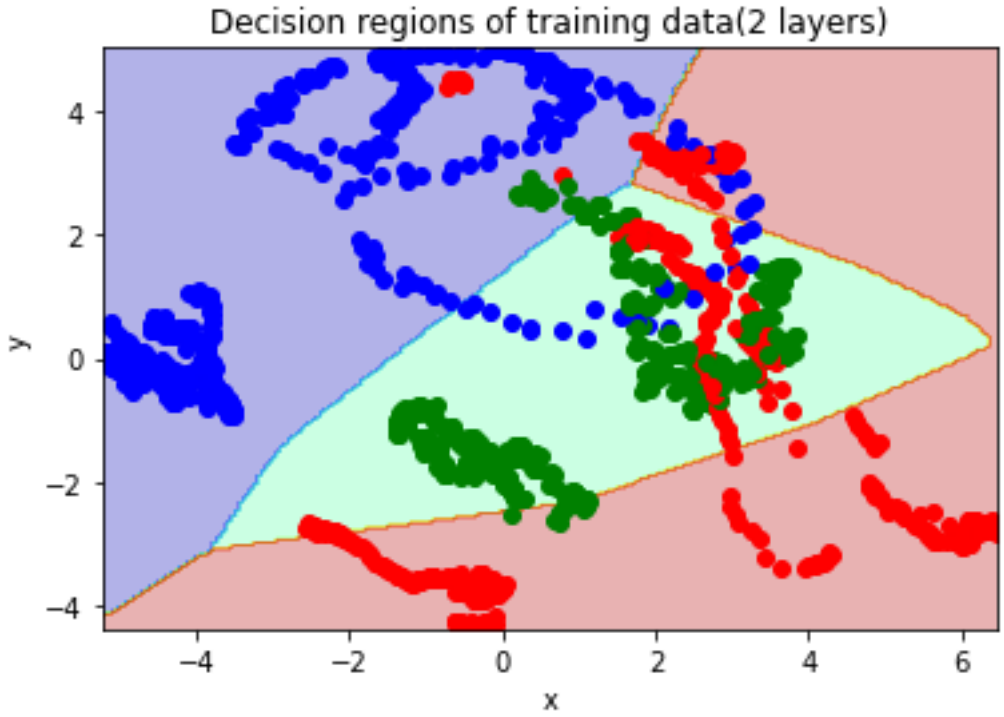


上面的第一張圖為 2 layer FNN，可以很直觀的看出，每一筆的 loss 是有慢慢降低的趨勢，所以可以看出其是有在訓練的，但有時 loss 還是會衝高，是因為一筆就更新一次所導致的，至於第二張圖的 loss 一開始幾乎沒有任何變動，我的理解是一開始可能權重訓練上還不夠多，所以沒有辦法精進模型，因為有兩層的 node，不一定所有的值都可以好好的更新到，所以一開始可能會看不出其 loss，後來可以看到有些 loss 下降，可以看出訓練成功，但是也有一些資料的 loss 相當大，個人猜測，也跟前者一樣，當其下一筆資料可能是不一樣的 label，然後 gradient 變化過大(因為一筆資料輸入並更新)，所以無法準確的預測其值，所以統整出使用 batch 的方式會有較好的更新，因為它可以統合所有資料的大小，給予較為固定方向的 gradient，所以可能會有更好的準確度。

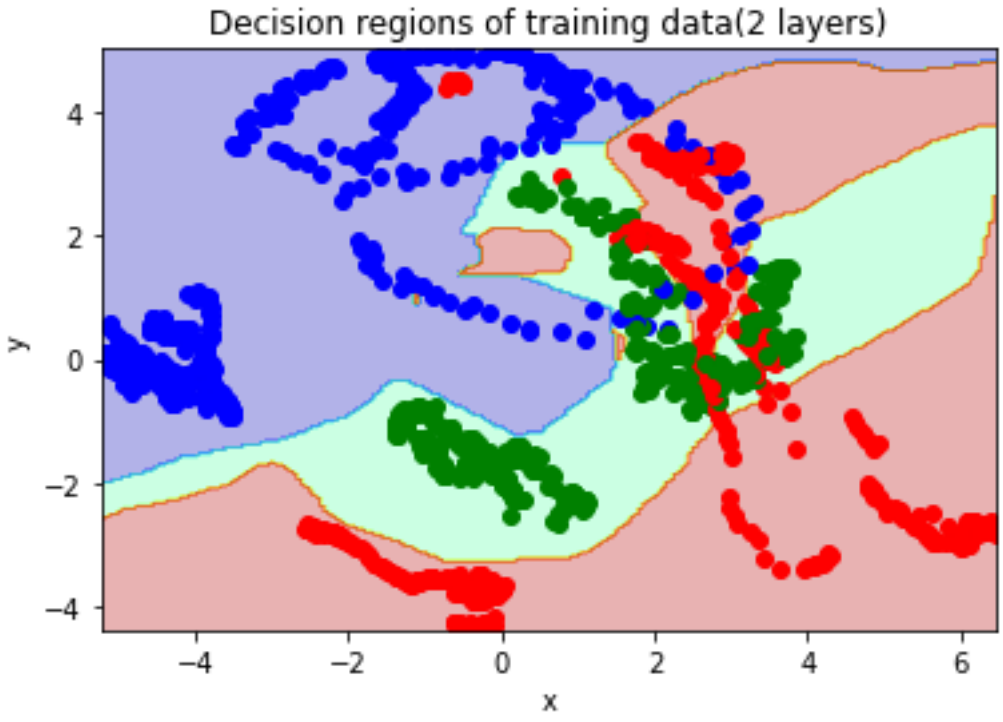
4. Plot decision regions and discuss the training / testing performance with different settings designed by yourself.

   a. 2 layers model

下面這張圖為初始 weight 乘上 0.01



下面這張圖為初始 Weight 乘上 0.9



發現其準確度上升，可以發現初始值會大大影響準確度，從 83.33%進化到 89.7590%

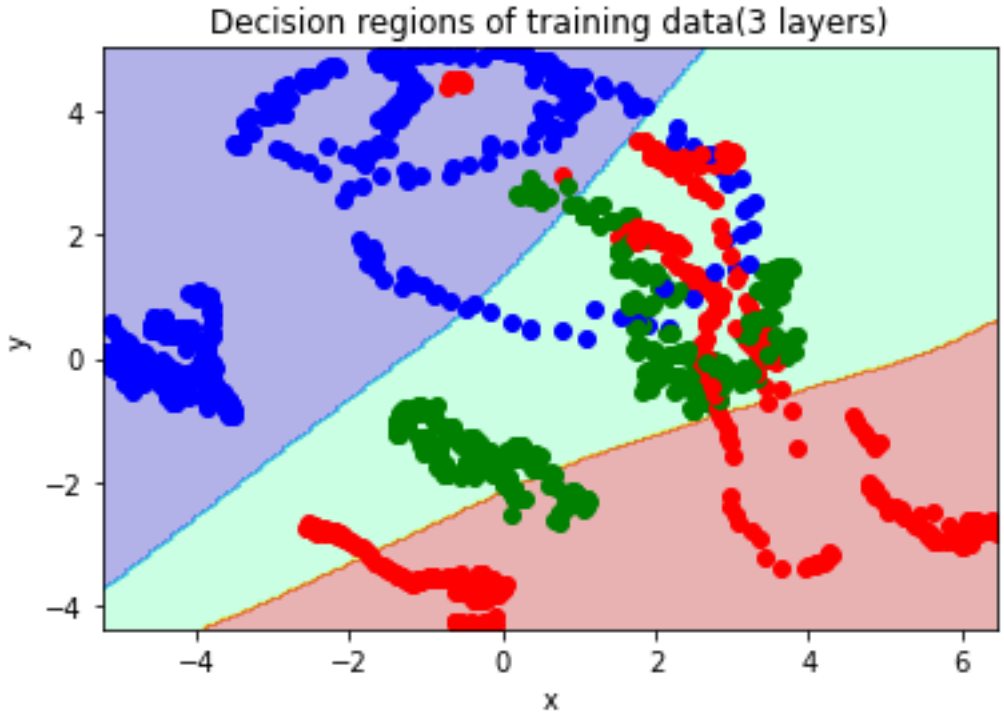
```
2 layer train score:  94.42176870748298
2 layer test score:  89.7590361445783
```

差距相當的大，也可以發現其 train 與 test 的分數都是上升，可以發現找到最適合這個問題的初始權重。

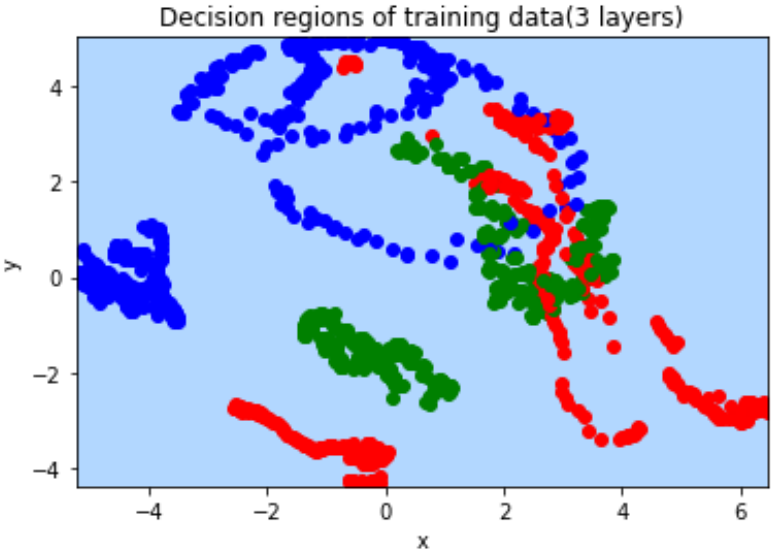
也可以從上面的 decision region 發現，原本少數綠色的點會出現再藍色區域內，因為權重提高，所以收斂到更好的位置，綠點的位置也變成綠色區塊

b. 3 layers model

下圖為初始權重乘上 0.01



下圖為乘上 0.9，發現其出現溢位，導致其權重過大，出現問題



為了使 testing 的準確度上升，還是保持初始權重乘上 0.01，原本的 epoch 為 3，現在改成 epoch = 5，發現其準確度大大的提升

```
3 layer train score:  88.91156462585033
3 layer test score:  93.37349397590361
```

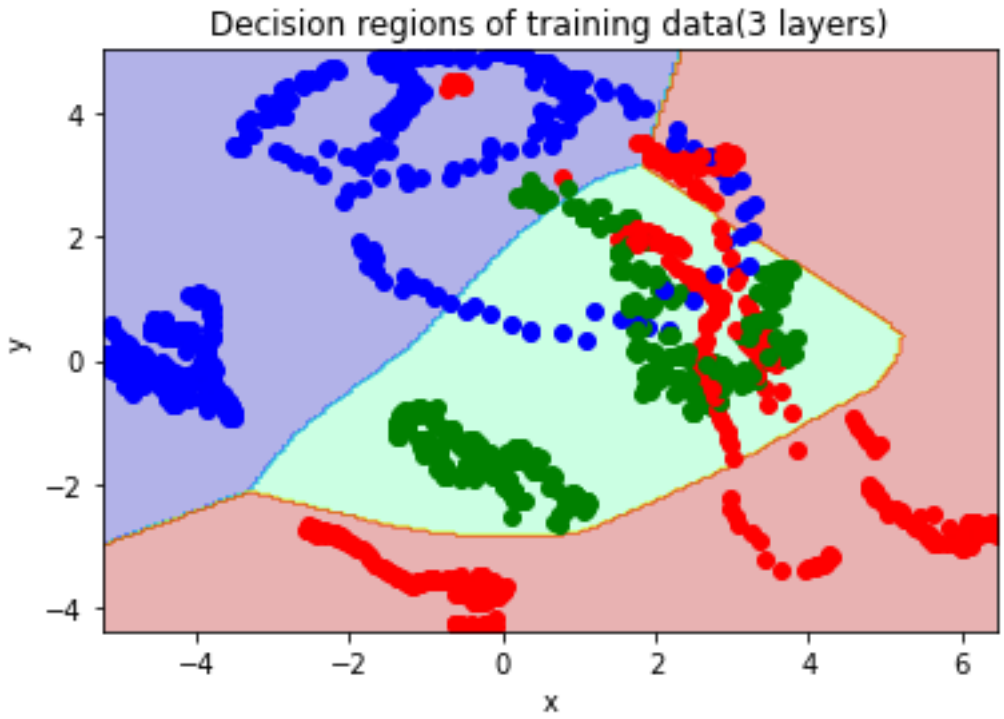
達到驚人的 93.37%，非常的高，training score 也相對提高

epoch = 6

```
3 layer train score:  89.59183673469387
3 layer test score:  89.95983935742971
```

準確度下降，可能是開始 overfitting，因為 train score 還是持續上升，但 test score 是下降，雖然還是比一開始訓練 epoch = 3 來的高。

下圖為 epoch 為 5 的 decision region



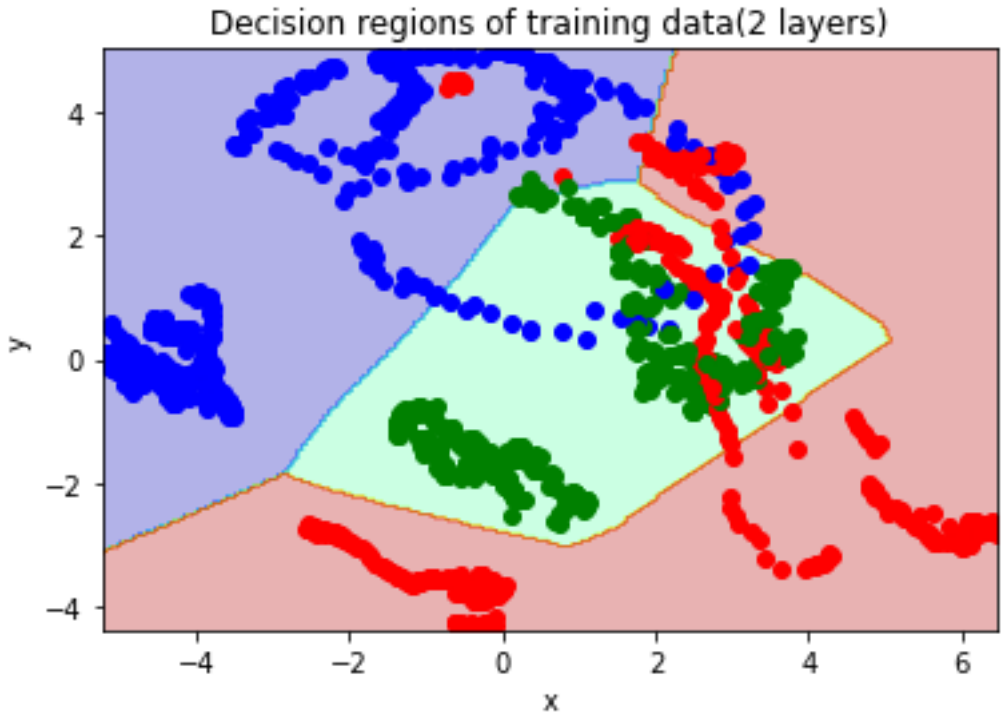
為了測試 2 layers 是否也是因為訓練次數過少的問題
將其 epoch 做多次測試，發現 epoch = 5 也有最高的準確度，其 train score and test score 都是提升。

```
2 layer train score:  88.843537414966
2 layer test score:  92.3694779116466
```

而且比更改權重還要來的高，至於 epoch = 6

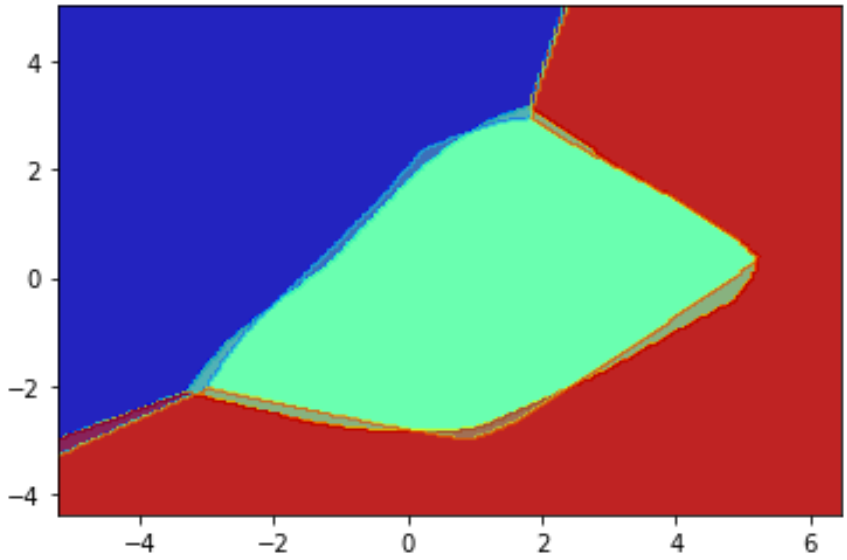
```
2 layer train score:  89.38775510204081
2 layer test score:  89.35742971887551
```

Test score 也開始下降，訓練準確度是上升，因為 overfitting 發生，所以 train score 上升，test score 下降，所以可以得出測試出較好的訓練次數可以獲得較好的準確度，並沒有任何快速的方法，只能多測試才可以得知結果。
下圖為 2 layers model 的 decision region

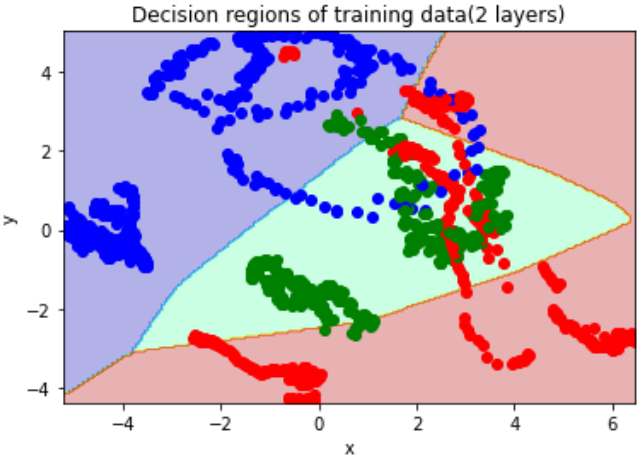


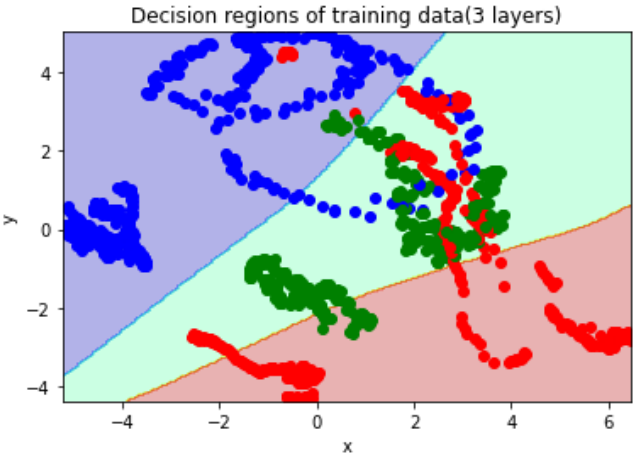
可以看出在同樣的情況下，並且 epoch = 5 的狀態，3 layers model 有最高的 test score 93.37 ，2 layers model 則只有 test score 92.3694，除此之外，這兩個模型所輸出的 decision region 長得非常像

下圖為兩張 decision region 疊加，可以看出非常像。



還發現一個特別的地方，就是 decision region 的形成，可以看出上面的圖為最佳狀態，但在 epoch = 3 的狀態下，兩個模型的 decision region 相差非常大





為何會有這樣的差異呢，我的理解是因為，三層隱藏層有較多的 Nodes，因為多了一個 hidden layer，所以從 decision region 可以看出，他沒有辦法像兩

層神經網路模型一樣，快速的收斂到較好的位置，主因應該是因為需要更新較多的權重，所以收斂速度較慢，因為權重的更新方式為最外層更新到最內層，所以需要花費較多的時間更新，層數越多應該會越明顯，但最後的收斂的準確度，卻是三層神經網路來的更好，這也是可以預期的，因為三層神經網路可以有較好的非線性模型，所以在兩個模型一樣的情況下，找到其最佳的訓練次數，就可以發現三層神經網路比兩層神經網路來的更好。

Source:

1. https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/
2. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e
3. https://medium.com/@qempsil0914/implement-neural-network-without-using-deep-learning-libraries-step-by-step-tutorial-python3-e2aa4e5766d1
4. https://medium.com/@qempsil0914/implement-neural-network-without-using-deep-learning-libraries-step-by-step-tutorial-python3-e2aa4e5766d1