110061543 蔡昀修

I. About this homework:

本次 Programming 共分兩部分，第一部分為 Two-layer 的 neural network (nn)，第二部分則為 Three-layer nn，本報告說明將著重在 2-layer nn 的部分，因為 2-layer 跟 3-layer 架構跟方法都差不多，我在 2-layer nn 已設好參數，相同的網路更新及訓練方法可以直接套用到 3-layer nn。此報告中，兩個 nn 都會包含以下說明:

1. Model architecture,
2. PCA method,
3. Test accuracy,
4. Training loss curves,
5. Decision regions,
6. Different design settings

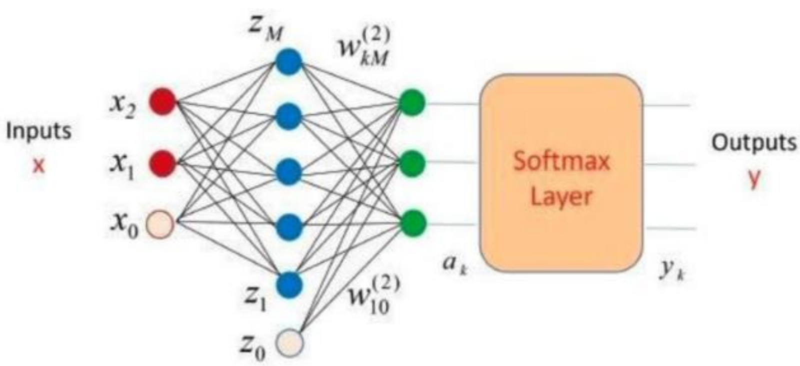
II. Two-layer neural network

1. Model architecture (Class: Neural_network):

(1) Code 描述：

我把 Neural network 設成 class，裡面包含這次手刻 nn 所需要的數學算式，包含：sigmoid, softmax, forward process, backward propagation 等。另外，這裡 default 設置兩個 parameters (batch_size, learning_rate)主要為了在最後一部分修改、比較不同參數間的效能。最後，我把 nn 的架構設成 list，方便直接 implement 2-layer 或 3-layer 的 nn，nn_arch = [2,64,3]表示 2-layer nn (1 hidden-layer)，而 64 表示 hidden-layer 有 64 個 neurons，第一部分練習的 nn 會類似作業中的網路架構圖。

```
class Neural_network:
    def __init__(self,
        batch_size: int = 32,
        epoch: int = 200,
        learning_rate: float = 0.1,
        nn_arch: list = [2,64,3]):
```



(2) Detailed implementation of the neural network (nn)

A. 網路架構

這次設計之網路架構是基本的 Fully Connected Neural Network (FNN)，架構及順序為 input layer→hidden-layer(s)→output layer→softmax→output。首先是 input 的部分，這次會輸入的資料是經 PCA 降維後的 2 維資料，而 output 則為三個水果種類的 labels。在 2-layer nn 的練習中，只會加入一層 hidden-layer (3-layer nn 有兩層 hidden-layer)，其 hiddenlayer_size=64，最後 output 層會經過一層 softmax 變成輸出的 output，所有訓練採用的 active function 是 sigmoid，主要是希望 acitivation 可以壓縮至 0~1。

B. 初始化

初始化每層的 weight 跟 bias，這裡用 dictionary 儲存以便之後做梯度更新。

```python
        for i in range(self.n_layer):
            self.param[f'weight{i+1}'] = np.random.rand(nn_arch[i], nn_arch[i+1])
            self.param[f'bias{i+1}'] = np.zeros((1, nn_arch[i+1]))

```

C. Forward propagation

從第一層開始，每層每個 neuron 之 acivation 值都會跟 weight 相乘並加上 bias，最後取 sigmoid，並傳入下層參數直到輸出 output 值，數學公式跟其對應的 code 如下圖。

a 第一層至 output 層:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = \sigma(z^{(l)})$$

b Output 用 softmax:

$$a_j^L = softmax(Z_j^L)$$

c Code:

```python
    def forward(self, X:np.ndarray, y:np.ndarray):
        self.prop = {}
        n_layer = self.n_layer
        self.feature = X
        self.label = y
        self.prop[f'a{0}'] = X

        for i in range(1, n_layer):
                self.prop[f'z{i}'] = self.prop[f'a{i-1}'] @ self.param[f'weight{i}'] + (self.param[f'bias{i}'])
                self.prop[f'a{i}'] = self.sigmoid(self.prop[f'z{i}'])
        self.prop[f'z{n_layer}'] = self.prop[f'a{n_layer-1}'] @ self.param[f'weight{n_layer}'] + (self.param[f'bias{n_layer}'])

        out = self.softmax(self.prop[f'z{n_layer}'])
        self.output = out
        return out
```

D. Backward propagation

為了要更新 weight 及 bias 值，這裡用了 gradient descent 來降低 loss。關於進行 SGD 之 batch_size 設定與其效能會在最後一部分描述。以下是數學式及 code，兩個皆分為三部份，因為從最後一層 output 至第一層的數值計算方式會稍微不同。

a Loss function:

$$Loss = \frac{1}{2N}\sum_{i=1}^{n}(\hat{y}-y)^2$$

b 以 weight 跟 bias 對 loss 做偏微分之 backpropagation 遞回關係式:

$$\frac{\partial C(W,b)}{\partial w^{(l)}} = \delta^{(l)} a^{(l-1)}$$

$$\frac{\partial C(W,b)}{\partial b^{(l)}} = \delta^{(l)}$$

其中:

$$\delta^{(l)} = \frac{\partial C(W,b)}{\partial z^{(l+1)}}\frac{\partial z^{(l+1)}}{\partial z^{(l)}} , \quad \delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)})$$

c 更新 weight & bias:

$$w^{(l)} = w^{(l)} - \alpha\frac{\partial C(w,b)}{\partial w^{(l)}}$$

$$b^{(l)} = b^{(l)} - \alpha\frac{\partial C(w,b)}{\partial b^{(l)}}$$

d Code:

```python
    def backward(self):
        n_layer = self.n_layer
        N = self.label.shape[0]

        # Calculate gradients of the output layer
        self.grad = {}
        self.grad["dz"+str(n_layer)] = (self.output - self.label) / N
        self.grad["dw"+str(n_layer)] = np.dot(self.prop["a"+str(n_layer-1)].T, self.grad["dz"+str(n_layer)])
        self.grad["db"+str(n_layer)] = np.sum(self.grad["dz"+str(n_layer)], axis=0, keepdims=True)

        # Backpropagate through the hidden layers
        for i in range(n_layer-1, 0, -1):
            self.grad["da"+str(i)] = np.dot(self.grad["dz"+str(i+1)], self.param["weight"+str(i+1)].T)
            self.grad["dz"+str(i)] = self.grad["da"+str(i)] * self.sigmoid_derivative(self.prop["z"+str(i)])
            self.grad["dw"+str(i)] = np.dot(self.prop["a"+str(i-1)].T, self.grad["dz"+str(i)])
            self.grad["db"+str(i)] = np.sum(self.grad["dz"+str(i)], axis=0, keepdims=True)

        # Update the parameters
        for i in range(1, n_layer+1):
            self.param["weight"+str(i)] -= self.learning_rate * self.grad["dw"+str(i)]
            self.param["bias"+str(i)] -= self.learning_rate * self.grad["db"+str(i)]
```

2. PCA method

(1) Dataloader：

首先處理 Data_train 以及 Data_test 的照片。原本提供的照片為 32*32 且 channel=3 的 rgb 照片，但是因為所有圖片皆為黑白，所以我直接用 opencv 轉為灰階來轉出 32*32*1 維度的照片。另外，我在這裡先將圖片做 flatten，將 32*32 的照片轉為 1024*1 的維度，以便後續直接輸入 neural network ；最後，將 train 及 test 資料的 data 及 label 分開，並都以 array 儲存。

```
# path
train_dir = os.path.join('./Data_train')
test_dir = os.path.join('./Data_test')
labels = ['Carambula', 'Lychee', 'Pear']

# training and test data
train_images = []
train_labels = []
for label_idx, label in enumerate(labels):
    # label_dir = os.path.join(train_dir, label)
    img_dir = glob.glob(os.path.join(train_dir, label, '*.png'))
    for i, imgs in enumerate(img_dir):
        img = cv2.imread(imgs)
        gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        resized_img = cv2.resize(gray_img, (32, 32))
        img_array = np.array(resized_img).flatten()
        train_images.append(img_array)
        train_labels.append(label_idx)

test_images = []
test_labels = []
for label_idx, label in enumerate(labels):
    img_dir = glob.glob(os.path.join(test_dir, label, '*.png'))
    for i, imgs in enumerate(img_dir):
        img = cv2.imread(imgs)
        gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        resized_img = cv2.resize(gray_img, (32, 32))
        img_array = np.array(resized_img).flatten()
        test_images.append(img_array)
        test_labels.append(label_idx)

# Convert the lists to numpy arrays
train_images = np.array(train_images) # already=1024
train_labels = np.array(train_labels)
test_images = np.array(test_images)
test_labels = np.array(test_labels)
```

(2) PCA:

這裡需要用 Principal Components Analysis (PCA)將 1024 維的資料降成 2 維，為了避免資料 scale 影響，我先用 sklearn 的 StandardScaler 標準化工具對資料進行處理(Fig.3)，處理完後才進行 PCA，針對 training data 做 fit，並分別 transform training 跟 testing data，PCA 要降為 2 維，就直接將 n_component 設成 2 即可，最後輸出的即為 features 跟 labels 。

```
    # Standardization
    scaler = StandardScaler()
    scaler.fit(train_images)
    train_images = scaler.transform(train_images)
    test_images = scaler.transform(test_images)
```

```
    # PCA (from 1024 to 2)
    pca = PCA(n_components=2, random_state=0)
    pca.fit(train_images)
    train_data_feature = pca.transform(train_images)
    test_data_feature = pca.transform(test_images)

    x_train, y_train = train_data_feature, train_labels
    x_test, y_test = test_data_feature, test_labels
```

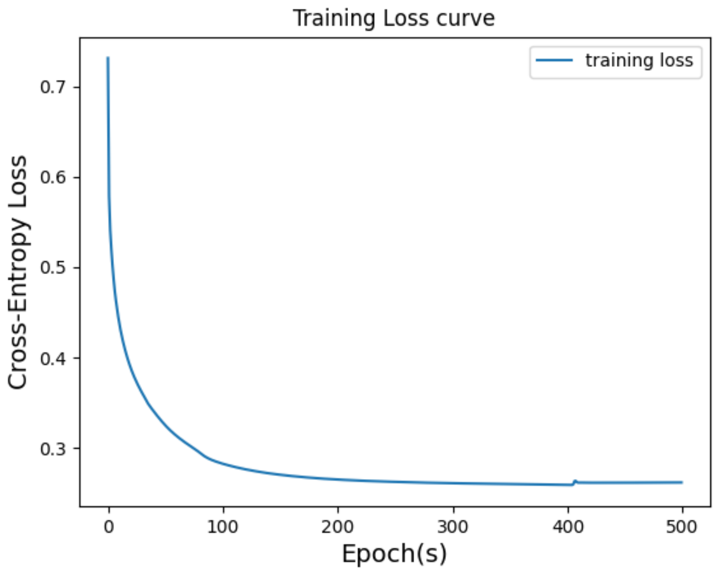
3. Test accuracy:

以 batch_size=32, learning_rate=0.1, epoch=500 來進行訓練，accuracy=96.78%。

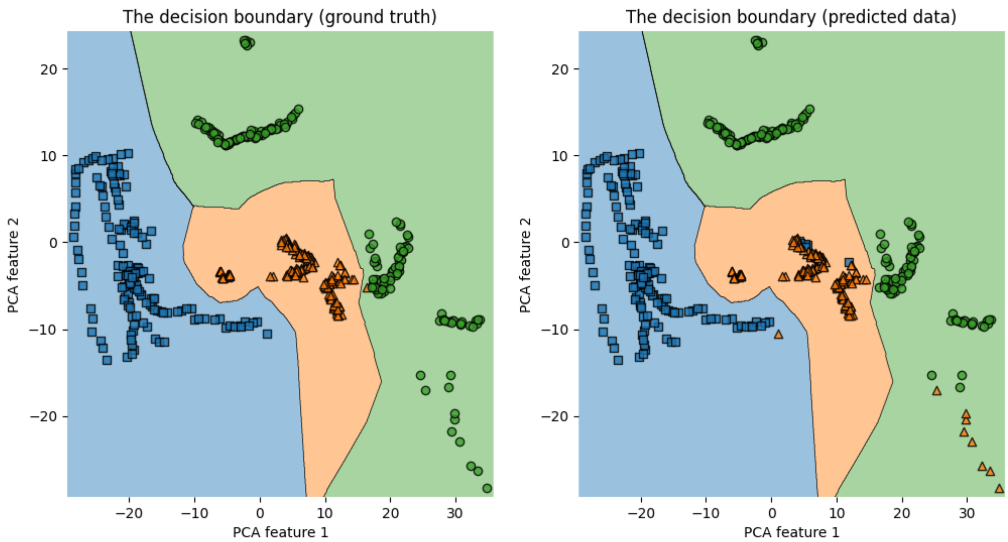
```
ezekiel@ezekiel-System-Product-Name: ~/Desktop/11120/M...
epoch 480/500: validation loss: 6.161299, Validation Accauacy: 94.22 %
epoch 481/500: validation loss: 6.157524, Validation Accauacy: 94.22 %
epoch 482/500: validation loss: 6.153755, Validation Accauacy: 94.22 %
epoch 483/500: validation loss: 6.149994, Validation Accauacy: 94.22 %
epoch 484/500: validation loss: 6.146239, Validation Accauacy: 94.22 %
epoch 485/500: validation loss: 6.142492, Validation Accauacy: 94.22 %
epoch 486/500: validation loss: 6.138752, Validation Accauacy: 94.22 %
epoch 487/500: validation loss: 6.135019, Validation Accauacy: 94.22 %
epoch 488/500: validation loss: 6.131292, Validation Accauacy: 94.22 %
epoch 489/500: validation loss: 6.127573, Validation Accauacy: 94.22 %
epoch 490/500: validation loss: 6.123861, Validation Accauacy: 94.22 %
epoch 491/500: validation loss: 6.120156, Validation Accauacy: 94.22 %
epoch 492/500: validation loss: 6.116457, Validation Accauacy: 94.22 %
epoch 493/500: validation loss: 6.112766, Validation Accauacy: 94.22 %
epoch 494/500: validation loss: 6.109081, Validation Accauacy: 94.22 %
epoch 495/500: validation loss: 6.105404, Validation Accauacy: 94.22 %
epoch 496/500: validation loss: 6.101733, Validation Accauacy: 94.22 %
epoch 497/500: validation loss: 6.098069, Validation Accauacy: 94.22 %
epoch 498/500: validation loss: 6.094412, Validation Accauacy: 94.22 %
epoch 499/500: validation loss: 6.090761, Validation Accauacy: 94.22 %
epoch 500/500: validation loss: 6.087118, Validation Accauacy: 94.22 %

accuracy: 0.9678714859437751
```

4. Training loss curves:

以 batch_size=32, learning_rate=0.1, epoch=500 來進行訓練。



5. Decision regions:

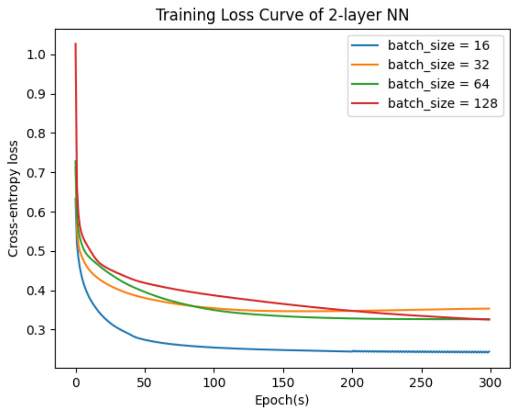


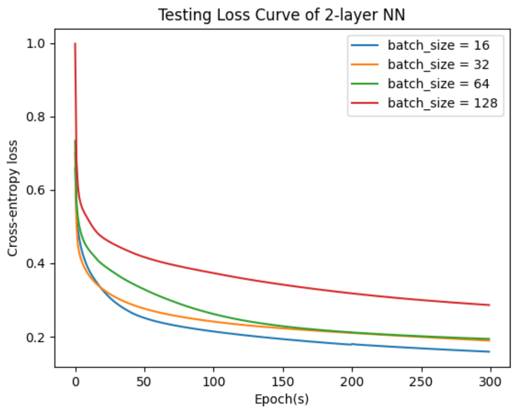
6. Different design settings:

以下比較三種常見的 neural network 參數 (Batch size, Learning rate, Neuron numbers)，觀察不同設定與 training loss, testing loss 的關係。

(1) Batch size: 在這個練習裡，根據 Training loss 跟 testing loss 圖表，發現 batch_size 較小時有比較好的訓練效能，表示梯度更新參數頻率較高，且方向較隨機的訓練比較符合題目需求。

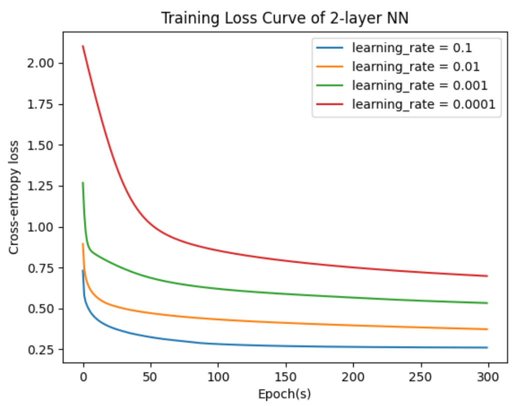
```
Testing accuracy with batch_size16: 0.963855421686747

Testing accuracy with batch_size32: 0.9658634538152611

Testing accuracy with batch_size64: 0.9578313253012049

Testing accuracy with batch_size128: 0.929718875502008
```

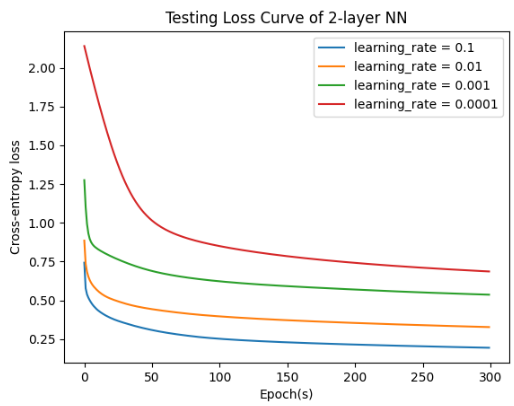




(2) Learning rate: Learning rate 的調整跟 batch size 相比較敏感，而在此範例，當 learning rate 設定較大時，更新權重較大，也收斂的最好。

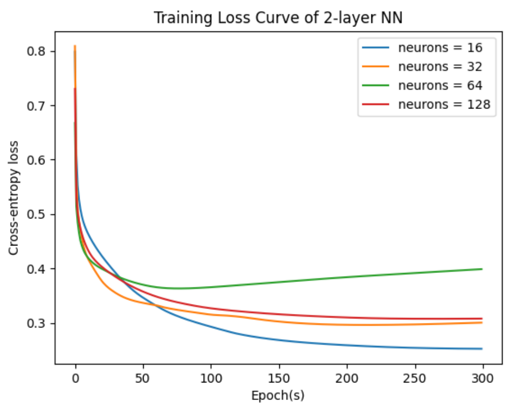
```
Testing accuracy with lr0.1: 0.9718875502008032

Testing accuracy with lr0.01: 0.9257028112449799

Testing accuracy with lr0.001: 0.7791164658634538

Testing accuracy with lr0.0001: 0.748995983935743
```

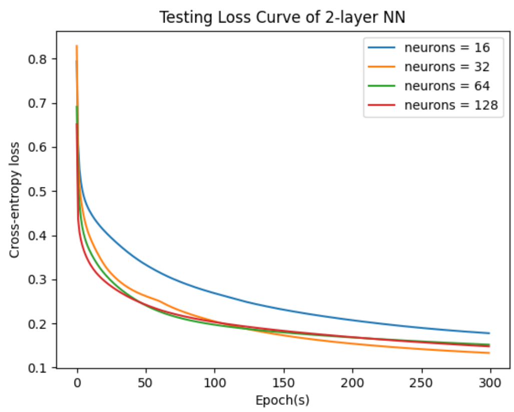




(3) Neuron numbers: 在 2-layer nn 中 testing 部分，我在 neurons=32 時有最好的效能，而當再增大 size 時，loss 表現卻比較差。

```
Testing accuracy with neurons16: 0.9497991967871486

Testing accuracy with neurons32: 0.9799196787148594

Testing accuracy with neurons64: 0.9598393574297188

Testing accuracy with neurons128: 0.9056224899598394
```





III. Three-layer neural network

1. Model architecture:

   nn_arch = [2,64,64,3]表示 3-layer nn (2 hidden-layers)，這裡比 2-layer nn 多加了一層 hidden-layer，而兩層 hidden-layer 之 neurons 皆為 64 個。

```
class Neural_network:
    def __init__(self,
        batch_size: int = 32,
        epoch: int = 200,
        learning_rate: float = 0.1,
        nn_arch: list = [2,64,64,3]):
```

2. PCA method

   同上方 Two-layer neural network 的作法，只是多了一層需要處理。

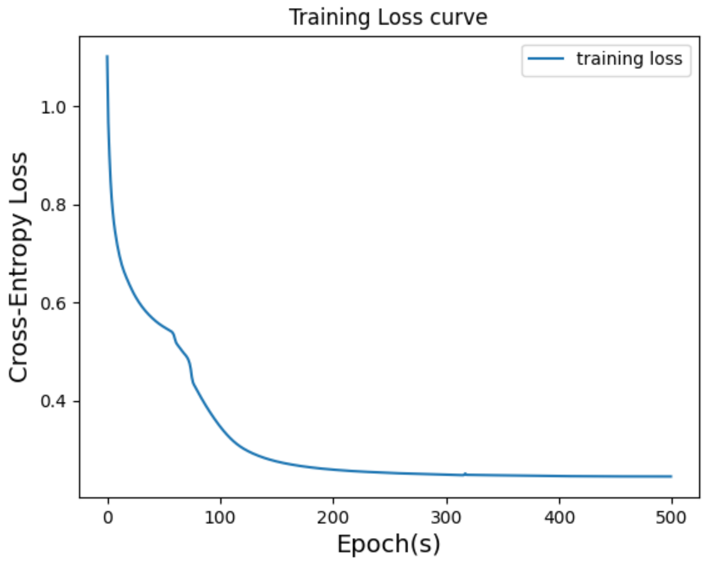
3. Test accuracy:

   以 batch_size=32, learning_rate=0.1, epoch=500 來進行訓練，accuracy=97.99%。

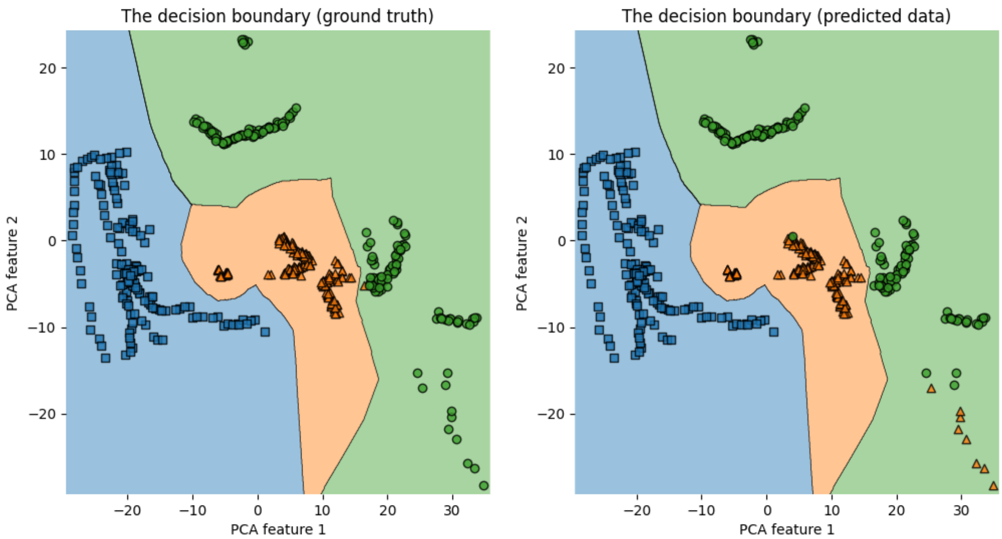
```
ezekiel@ezekiel-System-Product-Name: ~/Desktop/11120/M...
epoch 480/500: validation loss: 7.619836, Validation Accauacy: 90.48 %
epoch 481/500: validation loss: 7.616304, Validation Accauacy: 90.48 %
epoch 482/500: validation loss: 7.612776, Validation Accauacy: 90.48 %
epoch 483/500: validation loss: 7.609252, Validation Accauacy: 90.48 %
epoch 484/500: validation loss: 7.605731, Validation Accauacy: 90.48 %
epoch 485/500: validation loss: 7.602215, Validation Accauacy: 90.48 %
epoch 486/500: validation loss: 7.598701, Validation Accauacy: 90.48 %
epoch 487/500: validation loss: 7.595191, Validation Accauacy: 90.48 %
epoch 488/500: validation loss: 7.591684, Validation Accauacy: 90.48 %
epoch 489/500: validation loss: 7.588180, Validation Accauacy: 90.48 %
epoch 490/500: validation loss: 7.584679, Validation Accauacy: 90.48 %
epoch 491/500: validation loss: 7.581180, Validation Accauacy: 90.48 %
epoch 492/500: validation loss: 7.577683, Validation Accauacy: 90.48 %
epoch 493/500: validation loss: 7.574188, Validation Accauacy: 90.48 %
epoch 494/500: validation loss: 7.570695, Validation Accauacy: 90.48 %
epoch 495/500: validation loss: 7.567204, Validation Accauacy: 90.48 %
epoch 496/500: validation loss: 7.563715, Validation Accauacy: 90.48 %
epoch 497/500: validation loss: 7.560227, Validation Accauacy: 90.48 %
epoch 498/500: validation loss: 7.556740, Validation Accauacy: 90.48 %
epoch 499/500: validation loss: 7.553254, Validation Accauacy: 90.48 %
epoch 500/500: validation loss: 7.549769, Validation Accauacy: 90.48 %

accuracy: 0.9799196787148594
```

4. Training loss curves:

以 batch_size=32, learning_rate=0.1, epoch=500 來進行訓練。



5. Decision regions

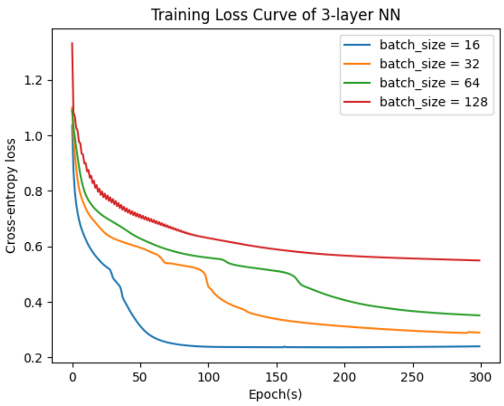


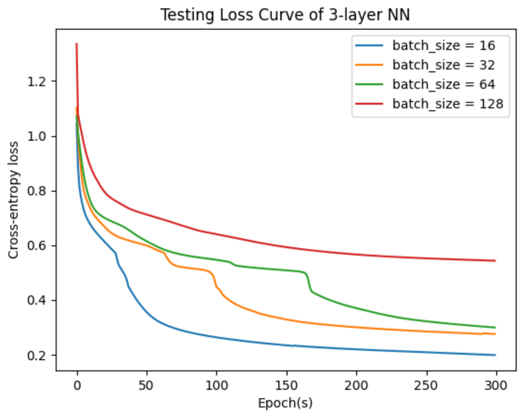
6. Different design settings:

以下比較三種常見的 neural network 參數 (Batch size, Learning rate, Neuron numbers)，觀察不同設定與 training loss, testing loss 的關係。

(1) **Batch size:** 3-layer 模擬的情況跟 2-layer 很像，若是遇到更大的 batch_size，學習效果都越差。

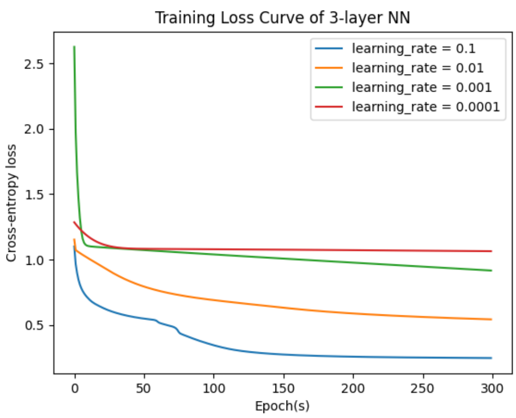
```
Testing accuracy with batch_size16: 0.9779116465863453

Testing accuracy with batch_size32: 0.9759036144578314

Testing accuracy with batch_size64: 0.9618473895582329

Testing accuracy with batch_size128: 0.7008032128514057
```

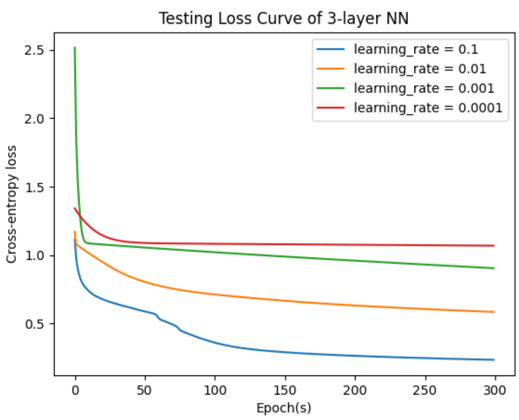




(2)Learning rate: 3-layer 對 learning rate(lr)的調整更敏感，只要 lr <0.01 就學不起來。

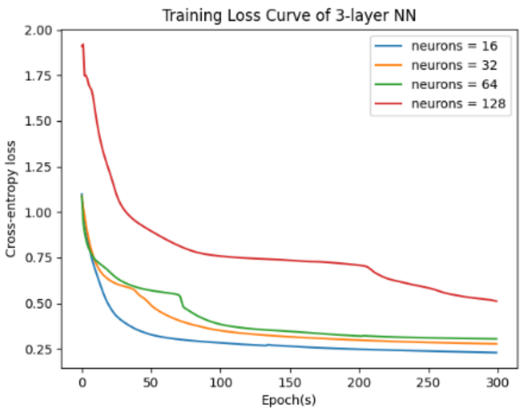
```
Testing accuracy with lr0.1: 0.9779116465863453

Testing accuracy with lr0.01: 0.6807228915662651

Testing accuracy with lr0.001: 0.6164658634538153

Testing accuracy with lr0.0001: 0.3493975903614458
```

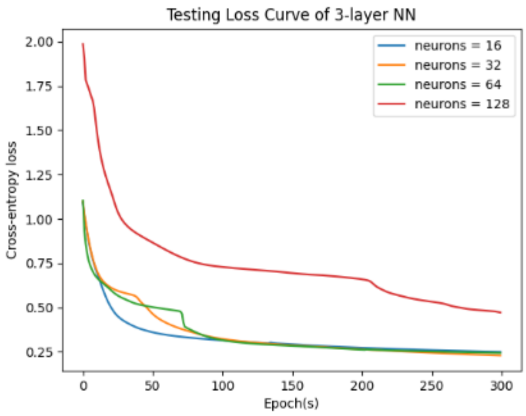




(3) Neuron numbers: 3-layer nn 跟 2-layer 一樣，都在 neurons=32 時有最好的效能，但 3-layer 的 loss 更低，增大 hidden_layer 數量確實可以有較好的學習效果，

```
Testing accuracy with neurons16: 0.9819277108433735

Testing accuracy with neurons32: 0.9839357429718876

Testing accuracy with neurons64: 0.9779116465863453

Testing accuracy with neurons128: 0.7389558232931727
```





Reference:
[1] https://zhuanlan.zhihu.com/p/421374471