機器學習 Machine Learning

劉昆琳 111061528 HW2 04/17/2023

Part 2. Programming assignment :

1. Describe your model architecture details and PCA method

原始資料集中的影像大小為 32x32x4，其中前兩個 32 代表影像的寬度和高度，最後的 4 代表 RGB 三種顏色和透明度。然而在使用 Python 的 cv2.imread 讀取影像時，透明度會被捨棄，因此影像大小為 32x32x3。若直接將整張影像當作輸入，會導致輸入參數會過大，相當於 3072 維的特徵，使模型無法高效訓練，因此，此次作業的第一步為利用 PCA 降維，將高維數據映射到低維空間中，期望在投影的維度上資料方差最大，以最大限度地分離資料並使用較少的數據維度，同時保留原數據的特性。

此次作業將利用 Python 中 sklearn 函式庫提供的 PCA 進行實驗，首先必須先將特徵正規化，因為每組特徵可能會因為單位的不同或數字大小的代表性不同，造成各自變化的程度不一，進而影響統計分析的結果，之後將 component 設成 2，代表希望留下的特徵維度，最後利用 pca.explained_variance_ratio_觀察降維後的每項特徵提供多少貢獻度，並將這些貢獻度總合起來，看看總共保留了多少特徵貢獻度，將結果繪製成圖 1，並將 2 維的資料分布繪製出來，如圖 2。

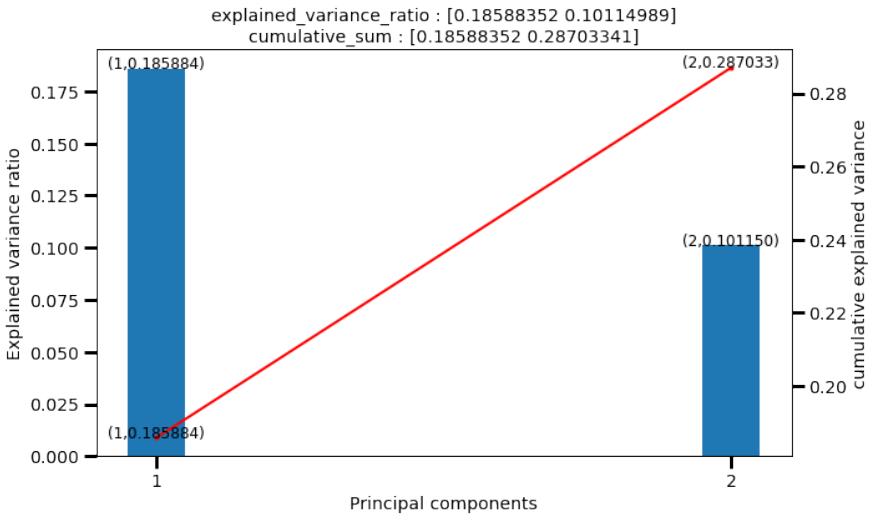

圖 1、explained variance ratio and cumulative sum

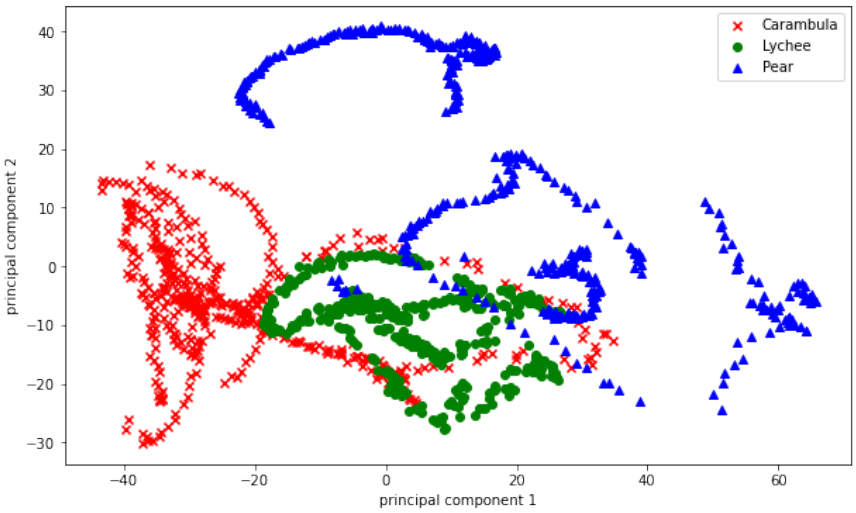

圖 2、降維後的資料分布

神經網路的架構如圖 3 所示，每個神經元由多個權重和 Bias 組成，經過激勵函數(activation function)後產生輸出。在兩層神經網路中，輸入先經過隱藏層(hidden layer)，再連接到輸出層。隱藏層使用 sigmoid 作為激勵函數，而輸出層因為需要進行多分類任務，因此採用 softmax 作為激勵函數。最後，輸出和 Groundtruth 進行損失函數(loss function)的計算，以更新參數。

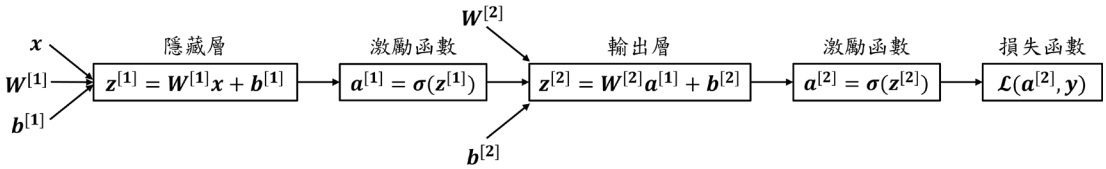

圖 3、神經網路架構

神經網路的第一步是初始化網路參數。為此設計了一個名為 initialize_parameters(n_x, n_h, n_y)的函數，其中 n_x、n_h 和 n_y 分別代表輸入層、隱藏層和輸出層的神經元數量。在函數中隨機生成符合正態分佈的數字來初始化權重和偏差，並儲存在一個字典中。需要注意的是，在初始化權重矩陣時會將權重值乘以 0.01，以將其限制在一個相對較小的範圍內。這樣做的原因是，如果權重矩陣中的值過大，會導致一些激活函數(如 sigmoid 和 tanh)的輸入值變得非常大或非常小時，梯度值會接近於零。這導致在反向傳播過程中梯度消失，使得權重更新變得非常緩慢，從而導致訓練過程變得非常困難。

```python
def initialize_parameters(n_x, n_h, n_y):
    '''
    初始化神經網路的參數
    n_x : 輸入層的神經元數量
    n_h : 隱藏層的神經元數量
    n_h : 輸出層的神經元數量
    函數會隨機生成一些符合正態分佈的數字來初始化權重和偏差，然後將它們儲存在一個字典中。

    常用的權重和偏差初始化方法是使用符合正態分佈的隨機值，並將它們乘以一個較小的常數，例如 0.01。
    這樣做的目的是將初始化值控制在一個較小的範圍內，從而避免出現過大或過小的初始化值。
    同時，較小的初始化值可以加速模型的訓練，因為它們可以使梯度下降算法更容易收斂。
    '''
    np.random.seed(1)
    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y, 1))
    parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2}
    return parameters
```

圖 4、初始化神經網路的參數

根據題目要求，需要使用 Stochastic gradient descent(SGD)這個梯度下降法來訓練神經網絡模型。SGD 是一種高效的優化算法，可以降低內存消耗，加快收斂速度，並有助於避免落入局部極小值。在程式中，我們需要設定 batch_size 這個參數，以決定每次從訓練集中隨機選取幾筆資料進行訓練。

前向傳播(forward propagation)是神經網絡中的一個重要步驟，用於將輸入信號傳遞到網絡中的每個神經元。在前向傳播過程中，每個神經元會計算加權輸入信號和偏差，並通過激活函數產生輸出信號，進而傳遞到下一層神經元。故設計一個名為 forward_propagation(X, parameters)的函數，如圖 5，其中 X 為輸入數據，parameters 為神經網路的參數，包含權重與 bias，用來計算前向傳播過程中的結果，即從輸入層開始，依次經過隱藏層和輸出層，得到最終的輸出值。

```python
def forward_propagation(X, parameters):
    '''
    神經網路的前向傳播過程，其作用是計算神經網路的輸出值，
    即從輸入層開始，依次經過隱藏層和輸出層，得到最終的輸出值。
    X : 輸入數據
    parameters : 神經網路的參數，包含權重與bias
    W1 : 輸入層與隱藏層之間的權重
    b1 : 輸入層與隱藏層之間的bias
    W2 : 隱藏層與輸出層之間的權重
    b2 : 隱藏層與輸出層之間的bias
    '''
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    Z1 = np.dot(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = softmax(Z2)

    # cache用來保存前向傳播的過程中計算出的結果 => 為了之後計算loss與反向傳播(backpropagation)
    cache = {"Z1": Z1, "A1": A1, "Z2": Z2, "A2": A2}

    return A2, cache
```

圖 4、前向傳播(forward propagation)

反向傳播(back propagation)是神經網絡中的一個核心算法，用於計算損失函數對每個參數的梯度。通過反向傳播，可以根據梯度來更新神經網絡的權重和偏差，從而使神經網絡能夠更好地擬合訓練數據，提高預測的準確性。在圖 3 的神經網路架構中，損失函數對於每個參數的梯度可以通過反向傳播進行推導，如圖 5 所示。首先計算輸出層的梯度，然後逐層向後計算每個隱藏層的梯度，最終得到每個參數的梯度。

以二分類來看：

損失函數：$L = -y \log a^{[2]} - (1-y) \log (1-a^{[2]})$

$$\frac{dL}{da^{[2]}} = \frac{-y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

$$\frac{dL}{dz^{[2]}} = \frac{dL}{da^{[2]}} \times \frac{da^{[2]}}{dz^{[2]}}$$

$$a^{[2]} = \text{sigmoid}(z^{[2]}) \Rightarrow \frac{da^{[2]}}{dz^{[2]}} = a^{[2]} \times (1-a^{[2]})$$

$$\frac{dL}{dz^{[2]}} = \left(\frac{-y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right) a^{[2]} (1-a^{[2]})$$

$$= -y(1-a^{[2]}) + (1-y)a^{[2]}$$

$$= -y + ya^{[2]} + a^{[2]} - ya^{[2]}$$

$$= a^{[2]} - y$$

$$\frac{dL}{dw^{[2]}} = \frac{dL}{dz^{[2]}} \times \frac{dz^{[2]}}{dw^{[2]}} \qquad \because z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$

$$= \frac{dL}{dz^{[2]}} \times a^{[1]} \qquad \therefore \frac{dz^{[2]}}{dw^{[2]}} = a^{[1]}$$

$$\frac{dL}{db^{[2]}} = \frac{dL}{dz^{[2]}} \times \frac{dz^{[2]}}{db^{[2]}}$$

$$= \frac{dL}{dz^{[2]}}$$

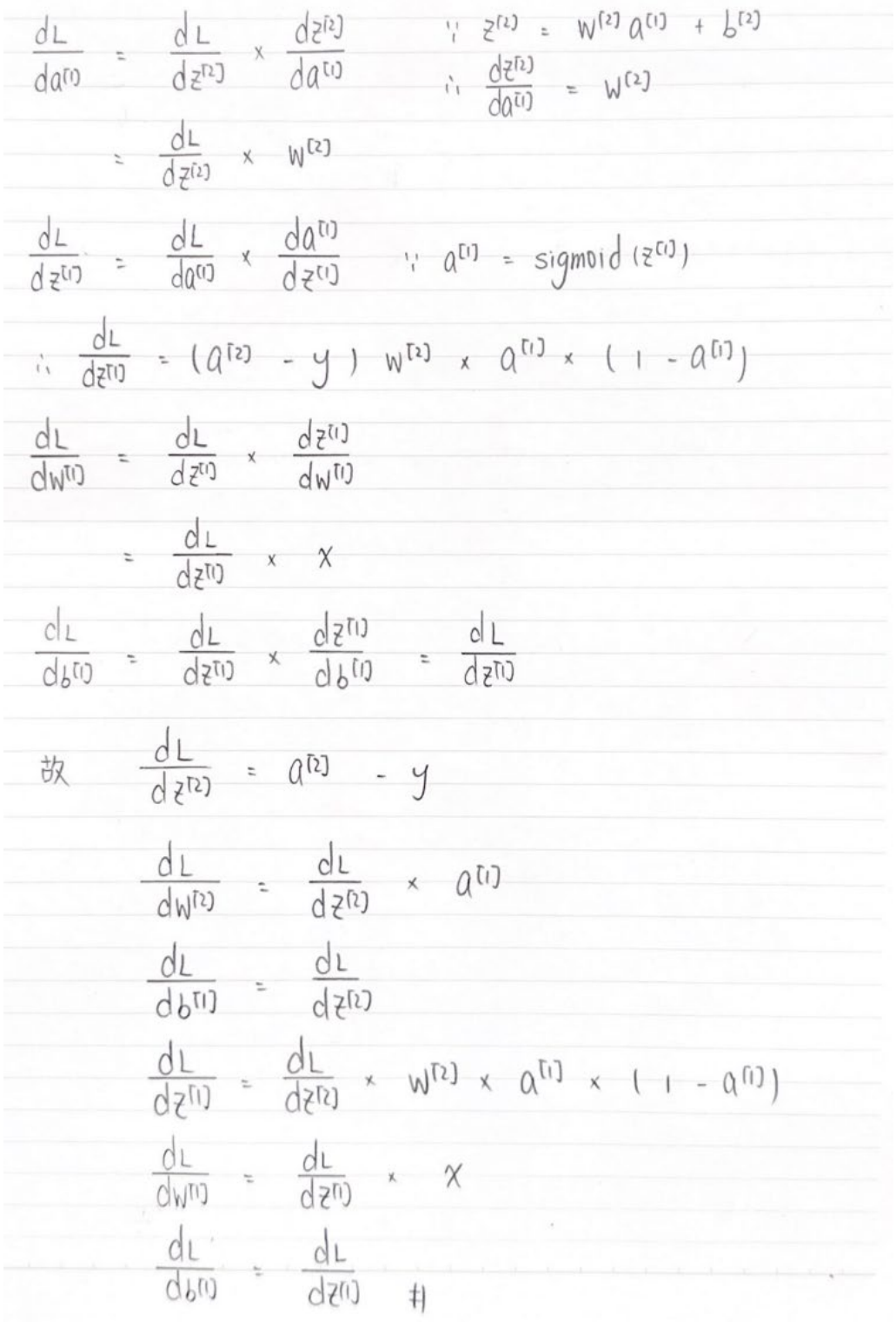

圖 5、二元分類的 back propagation 推導

而此次作業是一個多分類的問題，故 loss function 為 categorical crossentropy，且最後一層的 activation function 為 softmax，所以多元分類的 back propagation 推導如圖 6，可以發現結果會與二元分類一樣，故以此推導出的公式，建立函數

backward_propagation(parameters, cache, X, Y)，如圖 7，用於計算 loss 的梯度，以便進行參數更新，其中 parameters 為神經網路的參數，包含權重與 bias，cache 為前向傳播的過程中計算出的結果，X 為輸入數據，Y 為輸出層的數據。

若是一個多分類問題，則最後一層的 activation function 為 softmax，loss function 為 categorical cross - entropy

損失函數 $L = -y \log(a^{[2]})$

$$\frac{dL}{da^{[2]}} = \frac{-y}{a^{[2]}}$$

$$\frac{dL}{dz^{[2]}} = \frac{dL}{da^{[2]}} \times \frac{da^{[2]}}{dz^{[2]}}$$

$a^{[2]} = \text{softmax}(z^{[2]})$　　$\text{softmax} = \frac{e^{z_i}}{\sum_j e^{z_j}}$

對 softmax 偏微分，有兩種狀況

當 $i = j$：

$$\frac{da_i}{dz_i} = \frac{(e^{z_i})' \cdot \sum_j e^{z_j} - e^{z_i} e^{z_i}}{(\sum_j e^{z_j})^2}$$

$$= \frac{e^{z_i}}{\sum_j e^{z_j}} - \frac{e^{z_i}}{\sum_j e^{z_j}} \cdot \frac{e^{z_i}}{\sum_j e^{z_j}}$$

$$= a_i (1 - a_i)$$

當 $i \neq j$：

$$\frac{da_i}{dz_j} = \frac{0 - e^{z_i} e^{z_j}}{(\sum_j e^{z_j})^2} = -a_i a_j$$

$\frac{dL}{dz^{[2]}}$ for $i = j$

$$\frac{dL}{dz_i} = \frac{dL}{da_i} \times \frac{da_i}{dz_i} = \frac{-y_i}{a_i} \times a_i(1 - a_i)$$

$$= y_i(a_i - 1)$$

for $i \neq j$

$$\frac{dL}{dz_i} = \frac{dL}{da_j} \times \frac{da_j}{dz_i} = \frac{-y_j}{a_j} \times -a_i a_j = y_j a_i$$

合在一起，可得

$$\frac{dL}{dz_i} = y_i(a_i - 1) + \sum_{j, j \neq i} y_j a_i$$

$$= y_i(a_i - 1) + a_i \sum_{j, j \neq i} y_j$$

$\because \sum_j y_j = y_i + \sum_{j, j \neq i} y_j$

且 y 是一個 one-hot 向量

$\therefore \sum_j y_j = 1$

$$\frac{dL}{dz_i} = y_i(a_i - 1) + a_i \times (\underbrace{\sum_j y_j}_{1} - y_i)$$

$$= y_i(a_i - 1) + a_i \times (1 - y_i)$$

$$= y_i a_i - y_i + a_i - a_i y_i$$

$$= a_i - y_i$$

$$= a^{[2]} - y$$

從結果來看，多分類與二分類的 $\frac{dL}{dz^{[2]}}$ 是一樣的，故後面的推導結果也會相同 #

圖 6、多元分類的 back propagation 推導

```python
def backward_propagation(parameters, cache, X, Y):
    '''
    反向傳播算法，用於計算loss的梯度，以便進行參數更新
    parameters : 神經網路的參數，包含權重與bias
    cache : 保存前向傳播的過程中計算出的結果
    X : 輸入數據，維度為(n_x,m)，n_x表示為輸入層的神經元個數，m表示為樣本數量
    Y : 輸出數據，維度為(n_y,m)，n_y表示為輸出層的神經元個數，m表示為樣本數量
    '''
    m = X.shape[1]
    W2 = parameters["W2"]
    A1 = cache["A1"]
    A2 = cache["A2"]

    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis=1, keepdims=True) / m
    dZ1 = np.dot(W2.T, dZ2) * A1 * (1 - A1)
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis=1, keepdims=True) / m

    gradients = {"dW2": dW2, "db2": db2, "dW1": dW1, "db1": db1}
    return gradients
```

圖 7、back propagation

最後根據 back propagation 計算出來每層的梯度，更新神經網路的權重和 bias，並寫成函數 update_parameters (parameters, gradients, learning_rate)，如圖 8，其中 parameters 代表神經網路的參數，包含權重與 bias，gradients 代表 back propagation 計算出來的每層梯度，learning_rate 為學習率，其決定每次迭代中更新權重和偏差的步長大小，且學習率的選擇會影響神經網絡的訓練速度和收斂效，如果學習率過大，可能會導致模型震盪或無法收斂；如果學習率過小，則可能會導致模型收斂速度過慢或停留在局部極小值。

```python
def update_parameters(parameters, gradients, learning_rate):
    '''
    透過梯度下降算法更新神經網路的權重和bias。
    parameters : 神經網路的參數，包含權重與bias
    gradients : 梯度
    learning_rate : 學習率
    '''

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    dW1 = gradients["dW1"]
    db1 = gradients["db1"]
    dW2 = gradients["dW2"]
    db2 = gradients["db2"]

    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2

    parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2}

    return parameters
```

圖 8、更新網路的參數

2. Show your test accuracy.

在兩層的神經網路中，隱藏層的神經元數為 256，訓練次數為 1000，學習率為 0.005，batch_size 為 16，最終訓練集的準確率為 0.9073，loss 為 0.2766；驗證集的準確率為 0.8946，loss 為 0.2922，而**測試集的準確率為 0.8574**，如圖 9。

```
feature_test = Zscore(x_test.flatten().reshape(x_test.shape[0], x_test.shape[1] * x_test.shape[2] * x_test.shape[3]))[0]
pc1_test = pca.transform(feature_test)
X_test = pc1_test
Y_test = y_test
accuracy, _ = evaluate(X_test.T, Y_test.T, parameters)
print(f"Two layers NN accuracy : {accuracy : .4f}")
```

```
Two layers NN accuracy :  0.8574
```

圖 9、兩層網路之測試集準確率

而在三層的神經網路中，隱藏層的神經元數分別為 256、512，訓練次數為 1000，學習率為 0.01，batch_size 為 16，最終訓練集的準確率為 0.9082，loss 為 0.2307；驗證集的準確率為 0.8912，loss 為 0.2429，而**測試集的準確率為 0.8614**，如圖 10。

```
feature_test = Zscore(x_test.flatten().reshape(x_test.shape[0], x_test.shape[1] * x_test.shape[2] * x_test.shape[3]))[0]
pc1_test = pca.transform(feature_test)
X_test = pc1_test
Y_test = y_test
accuracy, _ = evaluate_3(X_test.T, Y_test.T, parameters)
print(f"Three layers NN accuracy : {accuracy : .4f}")
```

```
Three layers NN accuracy :  0.8614
```

圖 10、三層網路之測試集準確率

3. Plot training loss curves.

圖 11 為兩層神經網路的 accuracy 曲線圖與 loss 曲線圖，而圖 12 為三層神經網路的 accuracy 曲線圖與 loss 曲線圖。

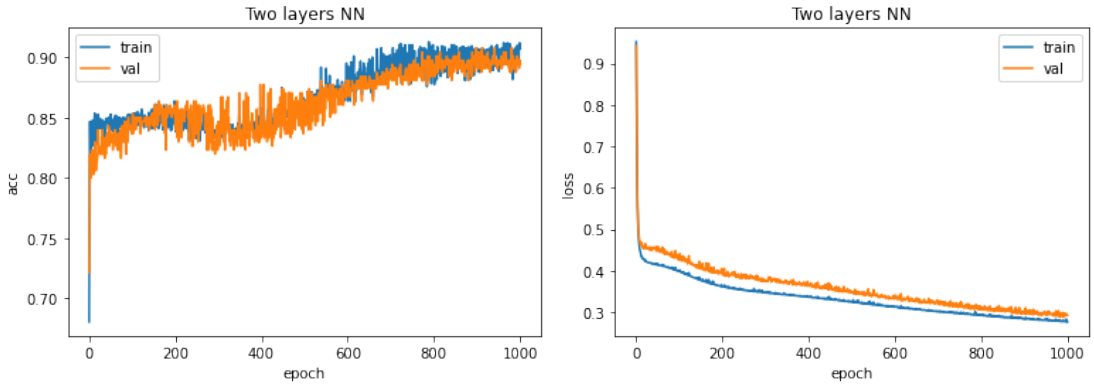

圖 11、兩層網路之 accuracy 與 loss 曲線圖

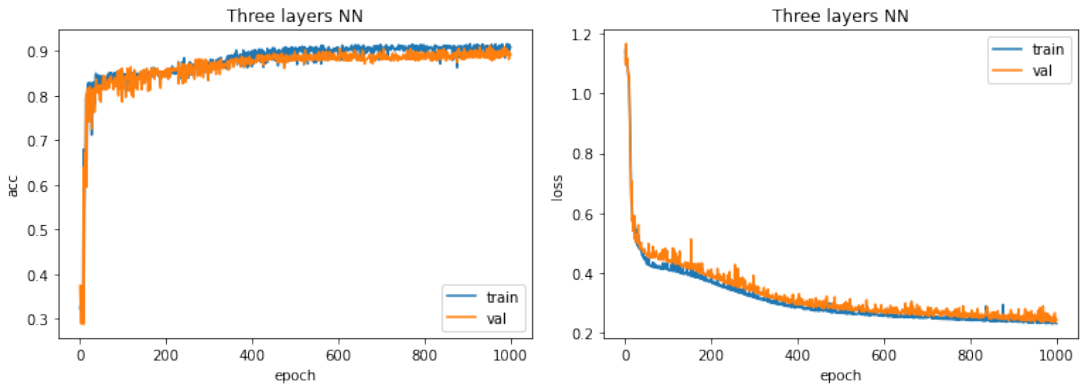

圖 12、三層網路之 accuracy 與 loss 曲線圖

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

圖 13 為兩層神經網路的 decision regions，而圖 14 為三層神經網路的 decision regions，從結果可以發現，在預測 Lychee 時，中間區域的表現是非常差的，容易被模型歸類到 Pear 或 Carambula，而在 principal component 1 為 0 的區域，也有幾筆 Carambula 被歸為 Lychee，這種現象可能是因為 PCA 降維後的特徵貢獻度不足所導致，如圖 1 所示，造成在降維後的資料分布中，這三類會有一定區域的重疊，如圖 2 所示。

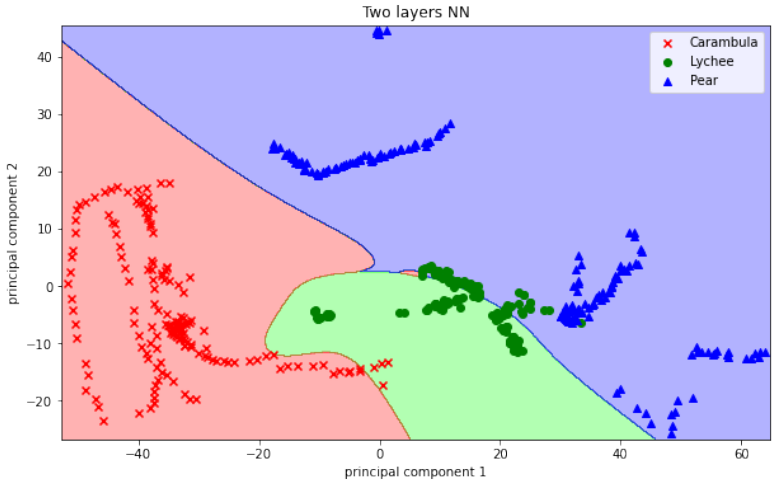

圖 13、兩層網路之 decision regions

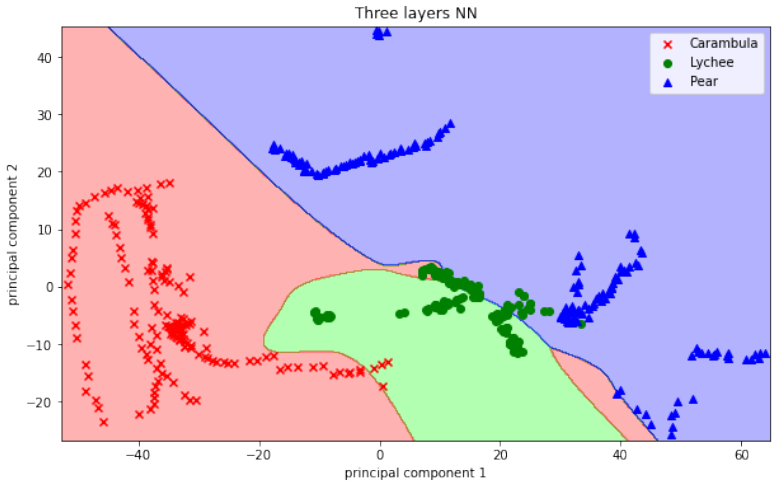

圖 14、三層網路之 decision regions

在以下的討論中，我們將保持初始訓練參數固定，並分別調整隱藏層的神經元數(n_h)、訓練次數(epoch)、學習率(lr)以及 batch_size，以觀察這些參數對模型輸出的影響。表 1 展示了通過調整神經元數目，同時保持其他參數固定的情況。從結果中，我們可以觀察到，增加神經元的數量能夠使模型變得更複雜，從而提高其表示能力和性能。但同時，這也會導致訓練時間的增加。過多的神經元可能會引起過擬合現象，即模型在訓練數據上表現良好，但在測試或新數據上表現不佳。相反，過少的神經元可能導致欠擬合，使模型的表示能力相對較弱。在這種情況下，模型可能無法有效地學習複雜的非線性關係，因此在擬合訓練數據方面可能表現不佳。

名稱	n_h	epoch	lr	batch size	train acc	val acc	test acc
Two layers NN	2	1000	0.005	16	0.8461	0.8401	0.7811
	4	1000	0.005	16	0.8520	0.8571	0.8092
	32	1000	0.005	16	0.8946	0.8741	0.8253
	128	1000	0.005	16	0.8903	0.8912	0.8373
	512	1000	0.005	16	0.8997	0.9014	0.8574

Three layers NN	2/4	1000	0. 005	16	0.8350	0.8265	0.7771
	4/8	1000	0. 005	16	0.8452	0.8265	0.8112
	32/64	1000	0. 005	16	0.8827	0.8639	0.9177
	128/256	1000	0. 005	16	0.9022	0.8810	0.8534
	512/1024	1000	0.005	16	0.9065	0.8946	0.8534

表 1、調整神經元數目

表 2 展示了通過調整 epoch，同時保持其他參數固定的情況。模型的性能會隨著 epoch 的增加而改善。越多的 epoch 可以使模型進行更多的訓練，提高模型的表現，但是當 epoch 數量過多時，模型可能會開始過度擬合訓練數據，導致泛化能力(generalization ability)下降，且加 epoch 的數量會增加訓練時間，計算資源也會隨之增加，另一方面，如果 epoch 數量太少，則可能無法充分訓練模型，導致模型性能不佳。

名稱	n_h	epoch	lr	batch size	train acc	val acc	test acc
Two layers NN	256	10	0.005	16	0.8342	0.8129	0.4645
	256	100	0.005	16	0.8486	0.8469	0.8032
	256	200	0.005	16	0.8299	0.8401	0.8112
	256	800	0.005	16	0.9022	0.8912	0.8574
	256	2000	0.005	16	0.9073	0.8980	0.8695
	256	4000	0.005	16	0.9158	0.9014	0.8655
Three layers NN	256/512	10	0. 005	16	0.3444	0.2891	0.3333
	256/512	100	0. 005	16	0.8376	0.8129	0.7631
	256/512	200	0. 005	16	0.8350	0.8231	0.7771

	256/512	800	0. 005	16	0.9014	0.8878	0.8514
	256/512	2000	0.005	16	0.9073	0.8912	0.8574
	256/512	4000	0.005	16	0.9184	0.9116	0.8655

表 2、調整 epoch

表 3 展示了通過調整學習(lr)，同時保持其他參數固定的情況。可以發現適當調整學習率可以提高模型的性能，如果學習率太小，權重更新的大小就越小，模型可能需要更長的時間才能收斂，或者收斂到 local minimum 就卡住了，導致無法找到 global minimum，反之如果學習率太大，模型可能會跳過最優解，甚至可能不收斂，且可能會導致訓練過程不穩定，甚至出現振盪現象，如圖 15，而從圖 16 至圖 25 可以看到不同學習率對整個訓練過程的影響。

名稱	n_h	epoch	lr	batch size	train acc	val acc	test acc
Two layers NN	256	1000	0.00001	16	0.8095	0.7993	0.7610
	256	1000	0.0001	16	0.8469	0.8197	0.7791
	256	1000	0.001	16	0.8393	0.8571	0.8092
	256	1000	0.01	16	0.9116	0.9082	0.8655
	256	1000	0.1	16	0.9133	0.9150	0.7831
Three layers NN	256/512	1000	0.00001	16	0.3444	0.2891	0.3333
	256/512	1000	0.0001	16	0.8010	0.8027	0.7108
	256/512	1000	0.001	16	0.8435	0.8367	0.7871
	256/512	1000	0.01	16	0.9022	0.8810	0.8394
	256/512	1000	0.1	16	0.9277	0.9048	0.8353

表 3、調整 lr

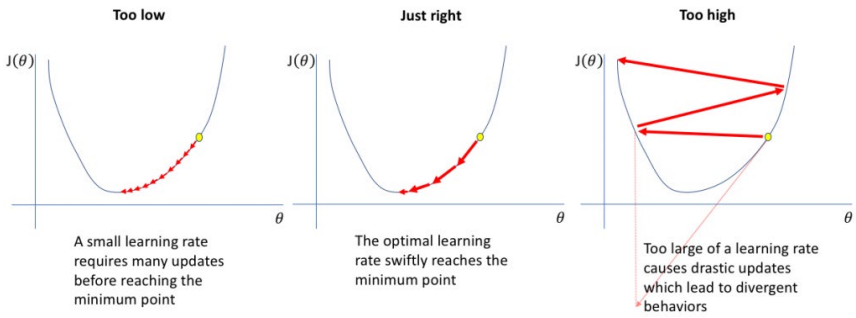

圖 15、學習率(lr)的大小對於模型的影響

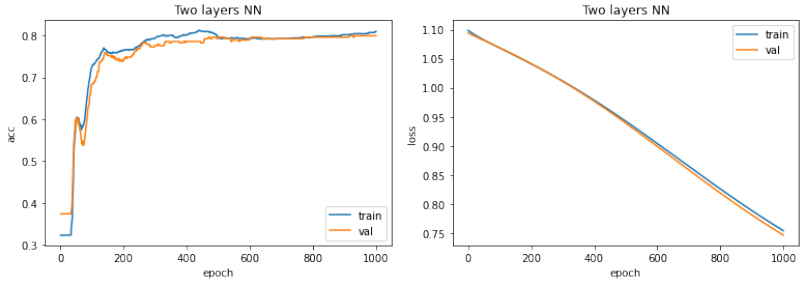

圖 16、兩層網路之 lr = 0.00001

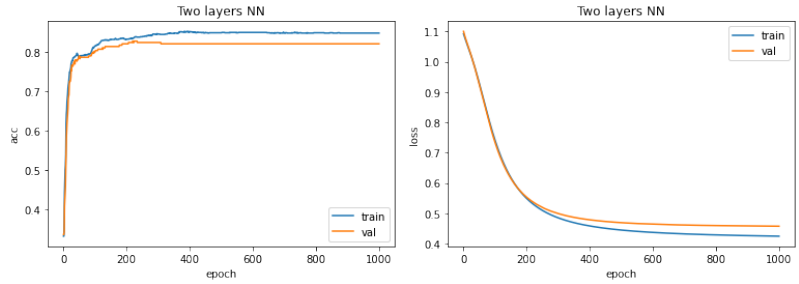

圖 17、兩層網路之 lr = 0.0001

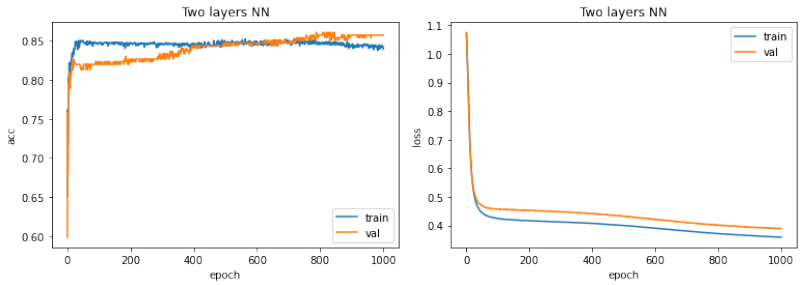

圖 18、兩層網路之 lr = 0.001

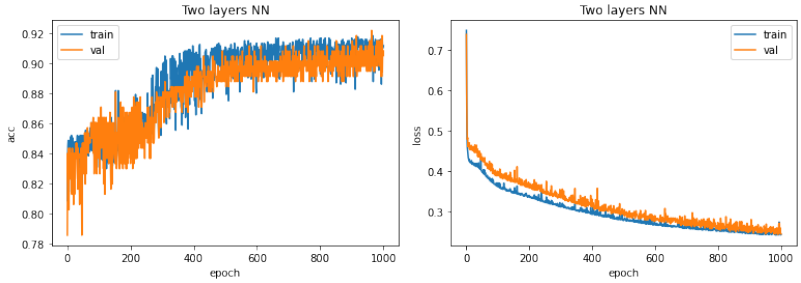

圖 19、兩層網路之 lr = 0.01

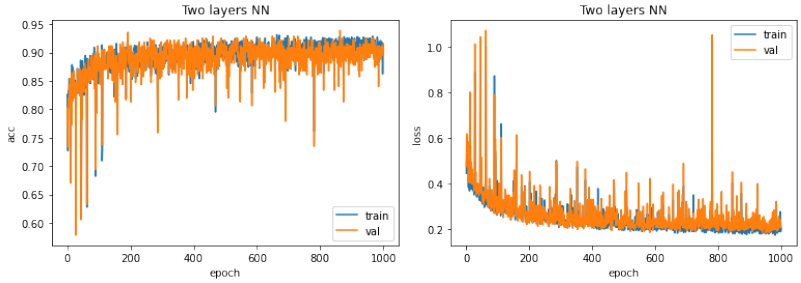

圖 20、兩層網路之 lr = 0.1

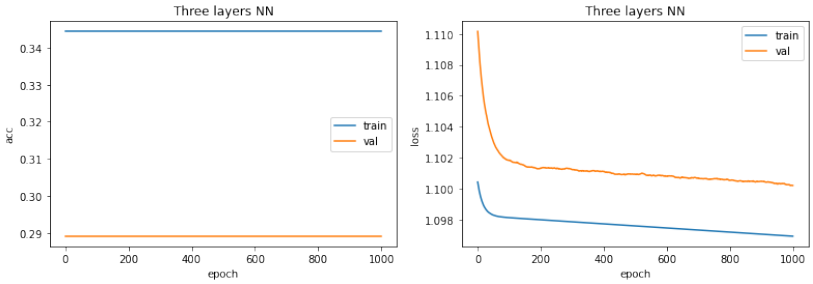

圖 21、三層網路之 lr = 0.00001

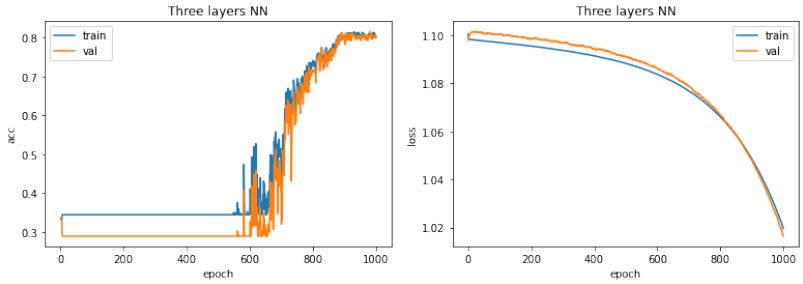

圖 22、三層網路之 lr = 0.0001

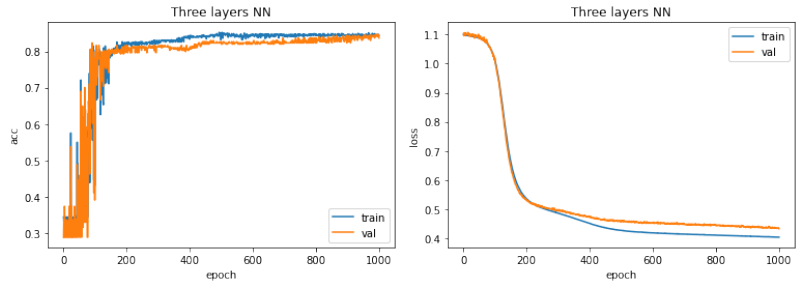

圖 23、三層網路之 lr = 0.001

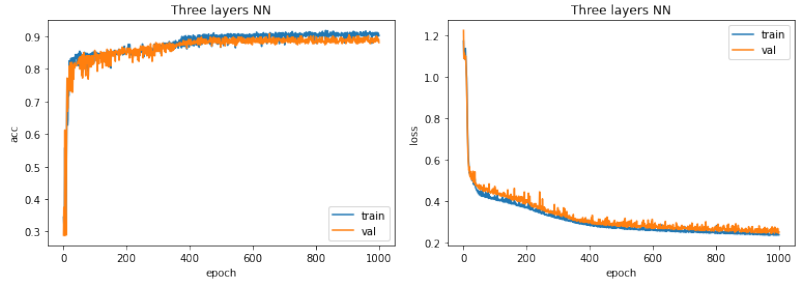

圖 24、三層網路之 lr = 0.01

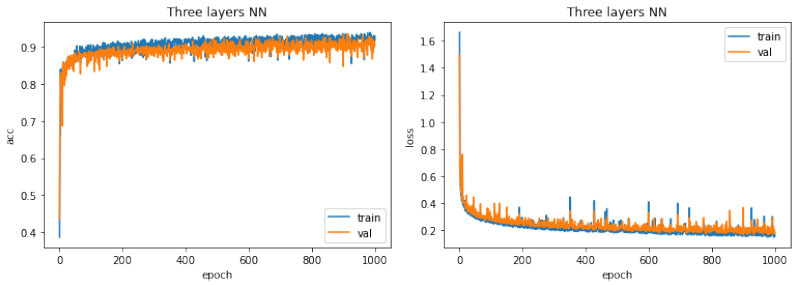

圖 25、三層網路之 lr = 0.1

表 4 展示了在固定其他參數的情況下，通過調整 batch size 對模型性能進行了評估。較大的 batch size 意味著模型在每次更新前會看到更多的數據，這有助於獲得更準確的梯度估計，加快收斂速度，並可以充分利用硬體平行運算能力，提高計算效率，但過大的 batch size 可能會降低模型的泛化能力，導致模型陷入 local minimum，而較小的 batch size 可能會使模型收斂速度較慢，因為模型需要經歷更多的梯度更新才能看到整個數據集，但較小的 batch size 可以增加隨機性，有助於模型跳出局部最優解，提高泛化性能。

名稱	n_h	epoch	lr	batch size	train acc	val acc	test acc
Two layers NN	256	1000	0.005	4	0.9090	0.9252	0.8373
	256	1000	0.005	32	0.8707	0.8605	0.8414
	256	1000	0.005	64	0.8316	0.8435	0.8112
	256	1000	0.005	256	0.8461	0.8367	0.7912
Three layers NN	256/512	1000	0.005	4	0.9184	0.9014	0.8614
	256/512	1000	0.005	32	0.8580	0.8639	0.8273
	256/512	1000	0.005	64	0.8469	0.8503	0.8032
	256/512	1000	0.005	256	0.8299	0.8129	0.7972

表 4、調整 batch size

綜合以上分析，適當地選擇超參數對模型的性能至關重要。當前已有許多方

法來改善訓練模型中的各種問題。例如，學習率衰減(Learning Rate Decay)可以提升梯度下降的效能。在訓練初期，較大的學習率有助於模型快速收斂，但隨著訓練的進行，較大的學習率可能導致模型在損失函數的全局最小值附近震盪，無法收斂到最佳解。此時，逐步降低學習率將有助於模型穩定地收斂到全局最小值；此外， Neural Network Intelligence(NNI)技術可以協助我們進行超參數的選擇。綜合以上，這些方法都是未來優化模型效果時值得嘗試的策略。

參考文獻

[1]https://blog.csdn.net/weixin_36815313/article/details/105341279

[2]https://zhuanlan.zhihu.com/p/161458241

[3]https://1305936314.github.io/2019/11/28/%E7%94%A8numpy%E5%AE%9E%E7%8E%B0%E4%B8%80%E4%B8%AA%E5%8F%8C%E5%B1%82%E7%9A%84softmax%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C/

[4] https://deepnotes.io/softmax-crossentropy

[5] https://zhuanlan.zhihu.com/p/25723112

[6] https://openai.com/blog/chatgpt