淺度機器學習：類神經網路

December 12, 2022

類神經網路（Aritficial Neural Network, ANN）風行於 90 年代，帶動一波人工智慧學習（AI）的熱潮。不過幾年的功夫，便被看破手腳，於是逐漸退潮並匿跡於學術圈。並不是類神經網路不好，而是遇到實務面的瓶頸。當資料大，變數多時，電腦軟硬體的執行能力讓人卻步。另一方面，類神經網路雖然具備學習的「智慧」，但在開發測試的過程中，處處需要人工的介入，加上軟硬體的支援不力，常常搞得人仰馬翻，逐漸失去信心。

消失的人工智慧在電腦軟硬體逐步提升後，終於達到實務面能接受的效率，尤其配合影像處理的技術，在圖形辨識（Pattern Recognition）的領域上異軍突起，獲得廣大的注視，再度引領風騷，甚至成為深度機器學習（Deep Learning）的主力。本章介紹類神經網路的概念，並以 `sklearn.neural_network` 及 `neurolab` 套件為工具，以幾個典型的範例揭開類神經網路的面紗。

本章將學到關於程式設計

〈本章關於 Python 的指令與語法〉
指令：
numpy: outer, tile
matplotlib.pyplot:imshow, surface
scipy.io: loadmat
sklearn.neural_network:MLPRegressor, MLPClassifier
sklearn.model_selection:train_test_split
sklearn.metrics:mean_squared_eror, plot_confusion_matrix
neurolab:newff

1 背景介紹

1.1 前饋式（Feedforward）類神經網路的原理

圖 1 展示具備一個隱藏層的典型前饋式類神經網路，[1]其中幾個須留意的數字為左邊輸入端（Input）標示為 p 個變數個數（圖中 $p = 14$），中間隱藏層（Hidden Layer）有 q 個神經元（圖中 $q = 10$）及最右邊的輸出層（Output Layer），共有 r 個輸出變數（圖中 $r = 3$）。輸入與輸出變數的個數 p, r 依問題的結構而定，譬如下一節的機器手臂的範例中 $p = 2$ 代表平面座標位置，而 $r = 2$ 代表機器手臂的兩個角度。值得一提的是中間的隱藏層與輸出層，特別是隱藏層的結構與所含的神經元數量 q。q 越大，代表輸出與輸入之間的關係越複雜，從數學關係的角度來說，便是彼此間的非線性程度越高。

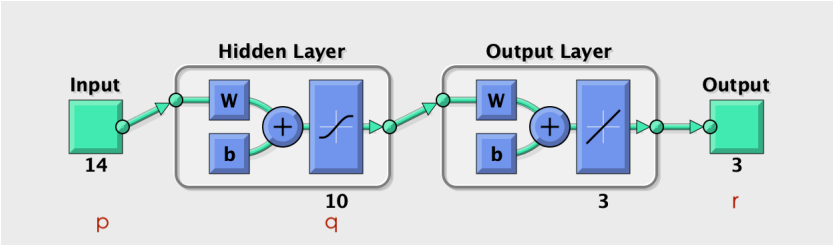


圖 1: 具備一個隱藏層的典型前饋式類神經網路

假設輸入端的 p 個變數表示為 $x_1, x_2, \cdots, x_p$，輸出端的 r 個變數表示為 $\hat{y}_1, \hat{y}_2, \cdots, \hat{y}_r$，則前饋式類神經網路的輸出與輸入間的數學關係寫成

$$\hat{y}_k = \sum_{i=1}^{q} w_{ki}^2 g\left(\sum_{j=1}^{p} w_{ij}^1 x_j + b_i^1\right) + b_k^2, \quad 1 \le k \le r \tag{1}$$

其中函數 $g(\cdot)$ 可以選擇為（$-1 \le g(z) \le 1$）

$$g(z) = c_1 \frac{1 - e^{-c_2 z}}{1 + e^{-c_2 z}} \tag{2}$$

函數 $g(z)$ 的長相便如圖 1 的隱藏層所繪製的函數圖。函數 $g(z)$ 也可以用在輸出層，以增加複雜度，不過通常使用如圖 1 的線性函數 $y = x$，也就是圖 1 的輸出層所繪製直線方程式。式 (1) 中的 w_{ij}^1 與 b_i^1 代表第一個隱藏層的第 i 個神經元與第 j 個輸入的權重係數（Weightings）與位階係數（Biases）。同樣地，w_{ki}^2 與 b_k^2 則是第二層（在此為輸出層）的第 k 個神經元與前一隱藏層的第 i 個輸出的權重係數與位階係數。

類神經網路根據已知的輸入與輸出資料 $x_j(n)$ 與 $y_k(n)$，調整係數 $w_{ij}^1, w_{ki}^2, b_i^1$ 與 b_k^2，使得真實資料 $y_k(n)$ 與類神經網路輸出資料 $\hat{y}_k(n)$ 的誤差為最小。假設共有

[1]本圖從 MATLAB 輸出

N 筆真實資料，則所謂類神經網路的訓練階段，寫成多變量函數的最小值問題，即

$$\min_{\Omega} e(\Omega) \tag{3}$$

其中誤差函數

$$\begin{aligned} e(\Omega) &= \sum_{n=1}^{N}\sum_{k=1}^{r}\left(y_k(n)-\hat{y}_k(n)\right)^2 \\ &= \sum_{n=1}^{N}\sum_{k=1}^{r}\left(y_k(n)-\sum_{i=1}^{q} w_{ki}^2 g\left(\sum_{j=1}^{p} w_{ij}^1 x_j + b_i^1\right)+b_k^2\right)^2 \end{aligned} \tag{4}$$

其中參數 $\Omega = \{w_{ij}^1, w_{ki}^2, b_i^1, b_k^2\}_{i=1,2,\cdots,q;j=1,2,\cdots,p;k=1,2,\cdots,r}$，共 $pq+qr+q+r$ 個參數。若以圖 1 的類神經網路架構（$p=14, q=10, r=3$）為例，式 (3) 的誤差函數 $e(\Omega)$ 的變數共 183 個。因此類神經網路可視為一個非常複雜、非線性程度很高到函數，能將輸入（X）與輸出（Y）的關係配適的很完美。

前文說到類神經網路曾在 90 年代刮起一陣旋風，雖然引起各界競相追逐並應用在許多領域，但如式 (3) 的高維度（變數多）最小值的計算問題卻是瓶頸。當輸入變數（p）越多，所選擇的隱藏層神經元數量（q）也越多時，若要求訓練的很完美（$e(\Omega)$ 越小），不可避免地需要更大的計算量，而造成計算時間過長，讓研發人員望之卻步。諸如此類的困擾隨著電腦軟硬體的提升，與演算法的成熟，已逐漸將類神經網路的學習優勢拉到「量產」的程度了，帶起新一波的人工智慧風潮，譬如機器學習中的「深度學習」便是充分利用了類神經網路的配適（fitting）能力。

另一個阻礙機器學習進程的因素是實用性。固然類神經網路能將輸入與輸出透過其高度的非線性函數配適到完美的境地，但對訓練外的輸入測試資料，其輸出表現並不如預期，中間還牽涉到隱藏層數、隱藏層的神經元數量及訓練程度都非常有關係。於是過多的測試與過長的測試時間，終於讓類神經網路的實用價值受到質疑。下一節的機器人手臂的範例也會展示不同的訓練過程將導致不同的測試結果。

2 機器人手臂的範例

2.1 解「逆運動方程式」(Inverse Kinematic Equations)

圖 2 是一個低階自由度（DOF, Degree of Freedom）的機械手臂，有兩截手臂，長度分別為 $l_1=20, l_2=10$。兩截手臂依據兩個角度 θ_1, θ_2 的改變，可以使手臂

最前端的位置 (x, y) 覆蓋如圖 3 第一象限的陰影區域，也就是手臂前端的指頭或鉸子可以碰觸到的地方。

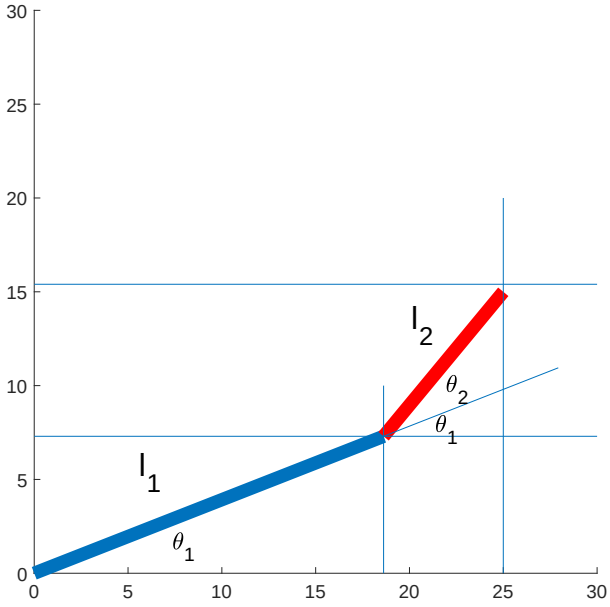


圖 2: 兩截式機械手臂

機械手臂藉由調整兩個角度 θ_1, θ_2 讓前端的鉸子移動到目的地 (x, y)。也就是給定 (x, y)，必須計算出 θ_1, θ_2 這兩個角度，這便是逆運動方程式的計算。因為維度低，角度少，因此這個逆運動方程式有解析解，如式 (6) 的 IKE

$$
\begin{aligned}
FKE \quad &: \\
x \quad &= \quad l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) \\
y \quad &= \quad l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2)
\end{aligned} \tag{5}
$$

$$
\begin{aligned}
IKE \quad &: \\
\theta_2 \quad &= \quad \cos^{-1}\left(\frac{x^2 + y^2 - l_1^2 - l_2^2}{2 l_1 l_2}\right) \\
\theta_1 \quad &= \quad \tan^{-1}\left(\frac{y}{x}\right) - \tan^{-1}\left(\frac{l_2 \sin(\theta_2)}{l_1 + l_2 \cos(\theta_2)}\right)
\end{aligned} \tag{6}
$$

當維度升高或機械手臂的角度變多之後，解析解變得不可能，必須尋求演算法的幫忙，找到近似解。在此，我們捨棄式 (6) 的解析解不用，轉而藉由類神經網

路的學習方式，在機器手臂能到達的範圍內找一些位置當作訓練資料（如圖 3 的 + 字位置），讓類神經網路透過這些訓練資料找到輸出角度與輸入位置的關係，再來看看學習過後，當面對新的位置資料 (x, y) 時，是否能準確輸出如式 (6) 正確的角度 (θ_1, θ_2)？

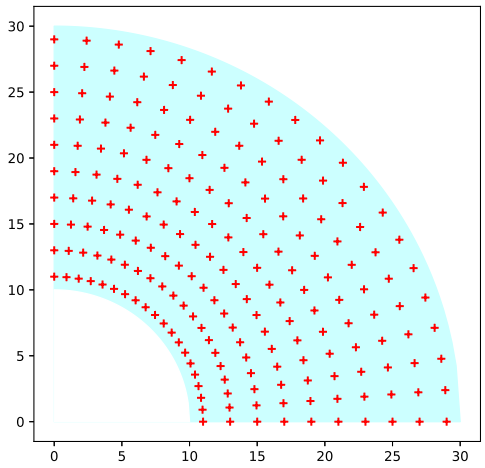


圖 3: 兩截式機械手臂所及範圍與訓練資料分布點

本範例採兩層的前饋式架構（Two Layer Feed-Forward Neural Network），根據前述機械手臂的設計，類神經網路的輸入是座標位置 (x, y)，輸出為機器手臂的兩個角度 (θ_1, θ_2)，架構如圖 4，其中隱藏層先試探性地採用 10 個神經元。[2]接著可以嘗試提高隱藏層的數量，並觀察輸出的擬合值與真實值的差異，直到某個適合的數量為止。譬如圖 5 展示隱藏層 10, 20, 40 與 80 時的真實位置（+）與擬合位置（o）的差異，並計算出均方根誤差（Root Mean Squared Error, rmse) 值。[3]從圖中可以看出當隱藏層數量增加時，均方根誤差隨之下降；當隱藏層數量提升至 80 時，均方根誤差反而增加了。不過，就類神經網路訓練而言，訓練資料所產生的誤差僅供參考，仍需參照測試資料的誤差而定。

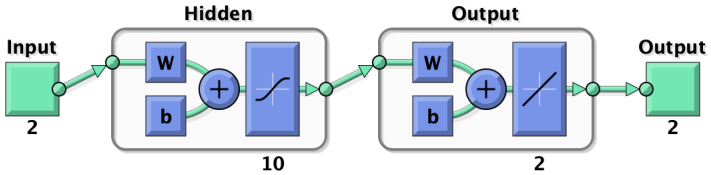
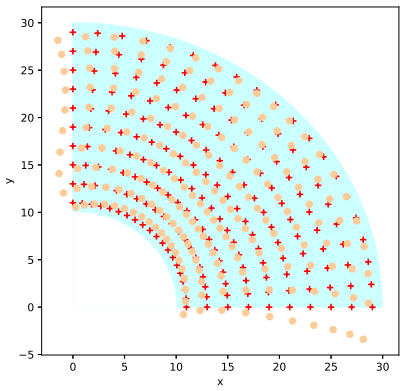


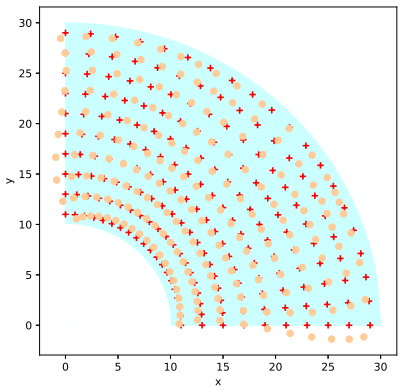
圖 4: 兩層的前饋式類神經網路架構

[2]類神經網路架構只含一個隱藏層，其中的神經元數量取決於輸入與輸出兩端的資料型態、資料量與問題本身的複雜程度。一般來說神經元數量必須通過若干次的訓練與測試方能決定，常常是件瑣碎無趣的工作。

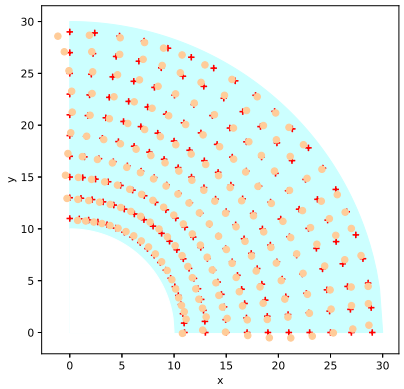
[3]rmse = $\sqrt{\frac{1}{2N}\sum_{i=1}^{N}(\hat{\theta}_1(i) - \theta_1(i))^2 + (\hat{\theta}_2(i) - \theta_2(i))^2}$



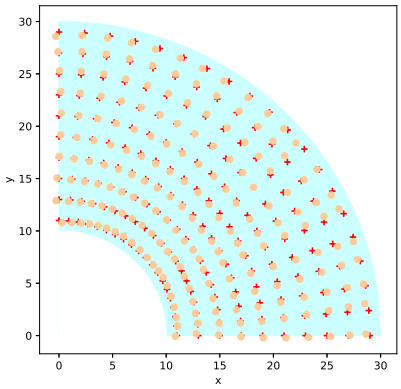
(a) 隱藏層 10, rmse = 0.0455



(b) 隱藏層 20, rmse = 0.0346



(c) 隱藏層 40, rmse = 0.0210



(d) 隱藏層 80, rmse = 0.0235

圖 5: 隱藏層數量與擬合值的準確度

圖 5 的類神經網路訓練採 `sklearn` 套件中的 `neural_network.MLPRegressor` 指令，其中 `MLPRegressor` 代表 Multi-Layer Perceptron Regressor。而 Multi-Layer Perceptron 便是像圖 1 的多層神經元（感知器 perceptrons）的架構，Regressor 代表其輸出/輸入的關係同於迴歸模型的概念（輸出與輸入皆是連續型資料，非類別型）。[4]下列程式碼呈現了訓練資料的準備（含輸出與輸出）與 `MLPRegressor` 的使用方式。

```
import numpy as np
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error

# Preaper training data (input)
l1, l2 = 20, 10
t = np.linspace(0, np.pi/2, 20)
```

[4]另有 `neural_network.MLPClassifier` 套件適合做分群。

```
l = np.arange(l1 - l2 + 1, l1 + l2 + 1, 2)
X = l.reshape(-1,1) @ np.cos(t.reshape(1,-1))
Y = l.reshape(-1,1) @ np.sin(t.reshape(1,-1))

# prepare training data (output)
theta2 = np.arccos((X.ravel()**2 + Y.ravel()**2 -\
        l1**2 - l2**2)/(2*l1*l2))
theta1 = np.arctan(Y.ravel()/X.ravel()) - \
        np.arctan(l2*np.sin(theta2)/(l1+l2*np.cos(theta2)))

# setup for ANN training
InputX = np.c_[X.ravel(), Y.ravel()]
OutputY = np.c_[theta1, theta2]
hidden_layers = (10, )
solver = 'lbfgs' # the best for robot data
# solver = 'sgd'
# solver = 'adam'
mlp_reg = MLPRegressor(max_iter = 8000, solver = solver,
    hidden_layer_sizes = hidden_layers, verbose = False,
    activation = 'logistic', # default activation = 'relu'
    tol=1e-6, random_state = 0)

mlp_reg.fit(InputX, OutputY) # Training ...
OutputY_hat = mlp_reg.predict(InputX) # Calculate fitted values
theta1_hat, theta2_hat = OutputY_hat[:,0], OutputY_hat[:,1]
# convert to (x,y) positions
x_hat = l1 * np.cos(theta1_hat) + \
        l2 * np.cos(theta1_hat+theta2_hat)
y_hat = l1 * np.sin(theta1_hat) + \
        l2 * np.sin(theta1_hat+theta2_hat)

rmse = np.sqrt(mean_squared_error(OutputY, OutputY_hat))
print('Root Mean square error is {:.4f}'.format(rmse))
```

上述指令中 MPLRegressor 宣告物件並設定該物件所需的參數。其中 max_iter=8000 與 tol=1e-6 是一般演算法具備的選項，都是指定演算法停止的條件。一般演算法屬於遞迴式的做法，用一個迴圈不斷地更新估計值，而 max_iter = 8000 代表最多的迴圈數，即使尚未達收斂條件，也強制停止；tol=1e-6 代表當目標函數（或損失函數 Loss function）隨著遞迴不再明顯改變（變化量小於 tol），則停止繼續遞迴，表示估計值差不多不再改變了，再繼續進行下去，進步有限，於是停止演算。

除了 max_iter=8000 與 tol=1e-6 之外，還有 max_fun, 與 early_stopping 等停止條件。讀者可以試試看，其中 early_stopping 牽涉到將訓練資料再

分出一部分做為驗證及停止演算法之用（Validation）。一個堪稱好的演算法指令，都會讓使用者介入演算法的運作細節，譬如 `learning_rate, momentum` 等。另外，設定 `verbose=True` 可以列印出每個迴圈的演算進度。這些參數常扮演關鍵角色，讓演算法走到正確的位置（不會在中途停止）。

此外，運用演算法指令最重要的選項，便是選擇一個適合的演算法，譬如，圖 5 選擇 `solver = 'lbfgs'` 即著名的 LBFGS 演算法。`MPLRegressor` 另提供 `adam` 與 `sgd` 兩種演算法，各有其優缺點，讀者最好都去試試看。最後是類神經網路中使用的非線性函數 `activation`，在此使用最經典的 `logistic` 函數，另有 `relu` 函數常用於深度學習的系統。如果想知道所用的 `MPLRegressor` 的所有參數，可以用指令 `print(mlp_reg.get_params())` 列印出所有參數與設定。

訓練後的類神經網路便可根據新的輸入資料，直接生成所需要的輸出資料。譬如，給予一個新的座標位置，就能迅速的計算出機器手臂所需要的角度。此時通常利用大量的測試資料（與訓練資料不同）所生成的角度與正確值相比，如果誤差在可接受的範圍內，則接受這個神經網路做為未來機器手臂的角度生成器（取代複雜或甚至不可能推演的逆運動方程式）。若不符使用的準確度，則必須調整後再測試。

2.2 訓練與測試資料的生成

圖 5 的訓練資料並非最理想的選擇，譬如，在半徑比較小的內側，資料較為密集，也就是訓練資料的選擇不夠均勻，當資料量不足時，這個問題必須被正視。

圖 6 展示另一種「看起來」較為均勻散佈的資料點。其作法，首先在正方形的區域產生均勻的亂數，再從中取位於第一象限內的所有點，最後挑選那些落在半徑為 10 與 30 的圓之內的亂數，這個方式也許不符合學理上的公平抽樣，但勉強可用。參考程式如下：

```
radius_in, radius_out =10, 30
X = uniform.rvs(loc = 0, scale = radius_out, size=(N, 2))
d = np.sqrt(X[:, 0] ** 2 + X[:, 1] ** 2)
Idx = (d < radius_out) & (d > radius_in)
TrainData = X[(d < radius_out) & (d > radius_in), :]
```

其實在圓內或球體內（甚至更高度空間），產生均勻亂數的方式已經很普遍，圖 7 呈現 1000 個亂數均勻散佈在圓內。參考的程式碼如下。

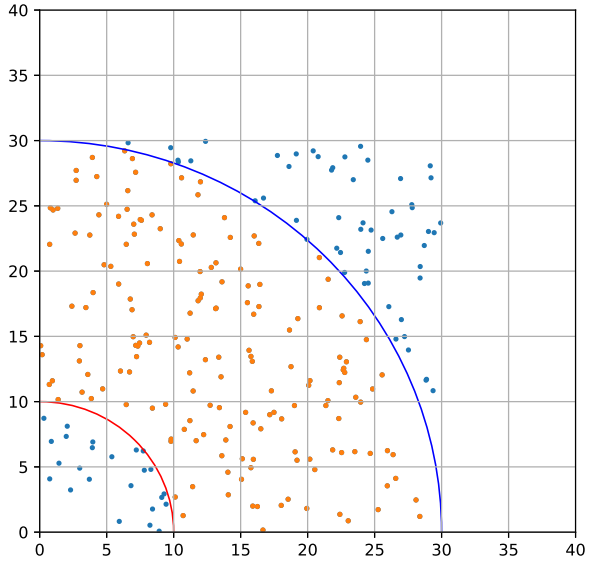


圖 6: 均勻落在扇形區的亂數

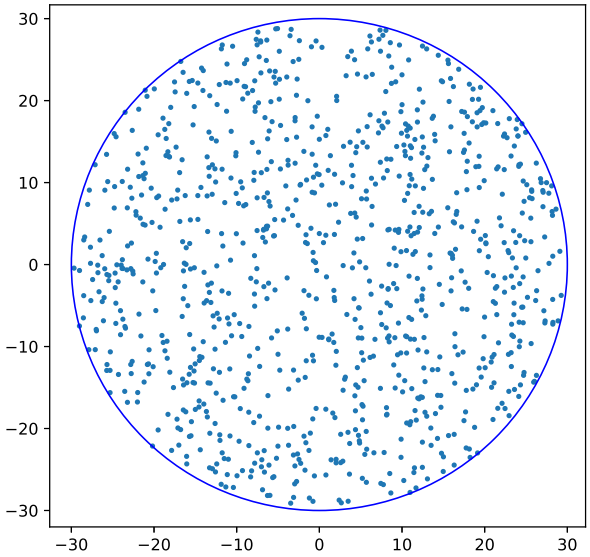


圖 7: 均勻落在圓形區域的亂數

```
import numpy as np
from scipy.special import gammainc

def randsphere(center, radius, n_per_sphere):
    """generate random numbers in a n-dimensional sphere
    i.e. in 2D, it is in a circle; in 3D, it is in a ball
```

```
    """

    r = radius
    ndim = center.size
    x = np.random.normal(size=(n_per_sphere, ndim))
    ssq = np.sum(x ** 2, axis=1)
    fr = r * gammainc(ndim / 2, ssq / 2) ** (1 / ndim)\
                / np.sqrt(ssq)
    frtiled = np.tile(fr.reshape(n_per_sphere, 1), (1, ndim))
    p = center + np.multiply(x, frtiled)
    return p

    p = randsphere(np.array([0, 0], 30, 1000)
```

其中函數 `randsphere` 便是用來產生高度空間球體內均勻散佈的點，其中第一個參數 `center` 代表中心點位置，其大小（`center.size`）也是實際空間的維度，`radius` 代表半徑，`n_per_sphere` 是樣本數；所以 `randsphere(np.array([0, 0], 30, 1000)` 代表在半徑 30 的圓內均勻產生 1000 個亂數。接著經過適度的「剪裁」，取得機器手臂的範圍空間。程式碼如：

```
p = p[(p[:,0] > 0) & (p[:,1] > 0), :] # 第一象限
d = np.sum(p**2, axis=1)
p = p[d >= radius_in**2, :] # 扇形內
```

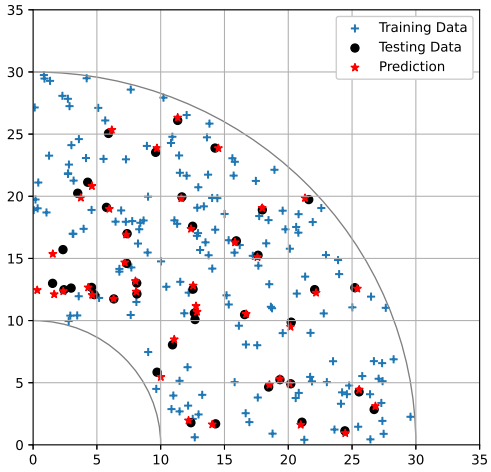
以下利用 `randsphere` 生成資料，並依 7:3 比例分成訓練與測試資料兩組。訓練資料用來訓練 ANN 網路，接著以測試資料檢驗訓練結果。圖 8 展示訓練資料的位置（+ 字符號）、測試資料（o 符號）與預測資料（* 符號）的關係位置。其中隱藏層設為 40，總樣本數分別 213 與 447。[5]測試資料（o 符號）與預測資料（* 符號）在某些位置有明顯的落差。此時可以考慮重新訓練並測試幾次比較看看，或直接調高隱藏層的神經元個數。另外，不同的訓練資料與測試資料的選擇，都會影響評估的結果，若不能快速、大量的執行各種測試評估直到滿意，再好的類神經網路也只是理想而已，這便是 90 年代的實際狀況。

2.3 NeuroLab 類神經網路套件的使用

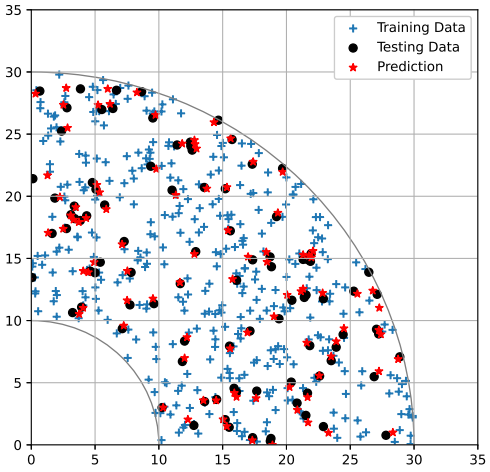
`NeuroLab` 是一組號稱「簡單有力 Simple and Powerful」的神經網路程式庫。[6]最大的特色是其使用方式近似著名的 `MATLAB` 神經網路套件（Neural Network Toolbox (NNT)），吸引原 `MATLAB` 使用者的青睞。

[5]因為採 `randsphere` 生成樣本，因此只能控制落在整個圓的樣本數，分別為 1000 與 2000。經過篩選後，落在扇形區的樣本數是 213 與 447。

[6]參考網路使用手冊 https://pythonhosted.org/neurolab/



(a) 隱藏層 40, 總樣本數 213



(b) 隱藏層 40, 總樣本數 447

圖 8: 隱藏層 $q = 40$ 下的訓練資料（+ 字符號）、測試資料（o 符號）與預測資料（* 符號）的關係位置。總樣本數 (a) 213, (b) 447。

本節介紹 NeuroLab 神經網路中與前一節相同的 Multilayer feed forward perceptron(newff)，[7]典型的設定如下程式碼：

```
import neurolab as nl
... 輸入資料 ...
InputX = np.c_[x_train, y_train] # inputs: N x 2
OutputY = np.c_[theta1, theta2] # output: N x 2

# create network
hidden_output_layers = [20, 2]#[hidden layers,output layer]

# set up activation functions for each hidden layer and output layer
transf = [nl.trans.TanSig(), nl.trans.PureLin()]

net = nl.net.newff([[x1.min(), x1.max()], [x2.min(), x2.max()]],
    size = hidden_output_layers, transf = transf)

#set up traning func
net.trainf = nl.train.train_bfgs # the default
# net.trainf = nl.train.train_cg # Newton-CG method
# net.trainf = nl.train.train_gd
# net.trainf = nl.train.train_gdx

# start training
err = net.train(InputX, OutputY, epochs = 5000, \
        show = 100, goal = 0.01)
```

[7]兩者名稱不同，但意思一樣。

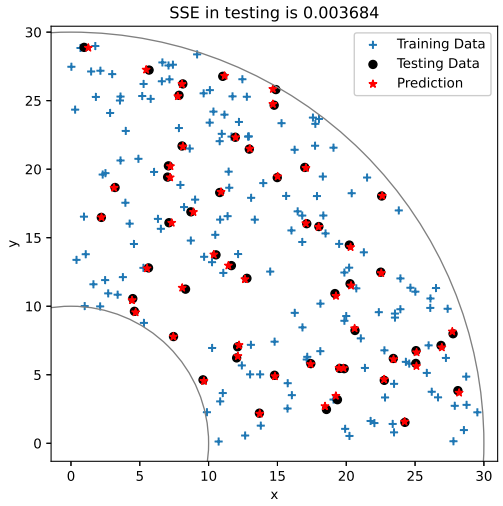
```
# Calculate fitted or prediction values
OutputY_hat = net.sim(InputX)
```

神經網路模型的設定程序與上一節的方式差不多。先建立神經網路的物件並設定必要的參數值，譬如含幾個隱藏層及每個隱藏層的神經元數量、每個隱藏層的輸出函數或稱轉換函數（Activation function, Transfer function）及訓練演算法。其中關於神經網路的層數算法，文獻上並沒有一致的說法，在這個套件裡，隱藏層包含了最後的輸出層（有就是除了輸入層以外，都稱隱藏層），上述程式碼的 `hidden_output_layers = [20, 2]` 代表輸出層有兩個變數，在輸入與輸出之間的一個隱藏層，神經元設為 20 個。中間隱藏層與輸出層的轉換函數分別為 Hyperbolic tangent sigmoid transfer function(`TanSig`) 及 Pure linear function(`PureLin`) 其中 `TagSig` 類似式 (2)，而 `PureLin` 則是純粹的線性輸出。

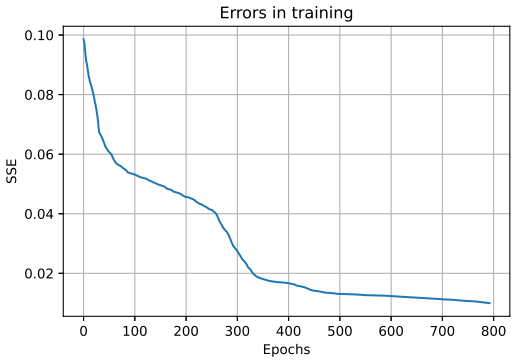
建立神經網路模型的指令 `net = nl.net.newff` 的第一個參數表明了輸入層的變數數量及其範圍值。而指令 `net.trainf = nl.train.train_bfgs` 則指定 BFGS 為訓練演算法，這也是預設的演算法。根據使用手冊上的說明，這個 BFGS 演算法出自 `scipy`。其他可使用的演算法也被註解在下面，僅供參考。不論是演算法還是前面提及的轉換函數，都與資料的型態與來源有關。指令 `net.train()` 開始對輸出入資料進行訓練，幾個常見的參數如 `epochs = 5000` 表示完整資料被訓練的次數，預設值為 500 次，若演算法執行到達這個次數尚未收斂，便強制結束。`goal = 0.01` 代表演算法停止的一項指標，在此為 Sum of Squared Error(SSE)，即輸出的真實值與擬和值的 SSE 誤差小於 0.01 即停止。[8]另一個參數 `show = 100` 表示每隔 100 個 epochs 列印出 SSE 的值。最後，`err = net.train()` 的輸出為演算法進程中每個 epoch 的 SSE 值。圖 9(a) 展示機器手臂的訓練與測試資料的預測誤差。與圖 8 相比，`NeuroLab` 的 `netff` 套件只用了 20 個神經元便取得很好的結果。圖 9(b) 展示訓練過程中 SSE 下降的趨勢，在將近 800 次 epochs 便下降到 0.01。

上述程式碼的最後一個指令 `net.sim` 等於其他套件的 predict 的意思，也就是使用訓練好的模型 (`net`) 對輸入資料進行計算，得到輸出值。

[8] $\text{SSE} = \sum_{i=1}^{N} ||\mathbf{y_i} - \hat{\mathbf{y}}_\mathbf{i}||^2$



(a) 測試資料的預測



(b) 訓練誤差

圖 9: NeuroLab 一個隱藏層（含 20 個神經元）的 (a) 預測能力與 (b) 訓練過程誤差的遞減。

3 圖形識別

類神經網路也普遍被應用在圖形識別上（Pattern Recognition），一般被歸類為「群組判別」，可以是監督式或非監督式。圖形識別指的是簡單線條或單純影像的判別，譬如，圖 10 從 0 到 9 的幾個手寫數字。早期應用在郵務系統分辨信件、包裹上的郵遞區號。由於手寫方式各異，每個相同數字的樣子也各異其趣，造成電腦自動判別時易產生困擾。而類神經網路的學習能力提供極有效率的解方，不妨親自來試試這個簡單的問題。

在進入神經網路的辨識訓練前，先岔題到影像資料的安排與呈現。在此我們準備了如圖 10 的手寫數字，[9]每個數字 1000 張，每張大小為 28×28 的黑白影像，作為類神經網路訓練用的資料。在進行訓練前，一定要先看看資料的長相，才能對接下來的神經網路訓練的難度有初步的看法。下列程式碼做出了圖 10。

```
from scipy.io import loadmat

data_dir = '../Data/'
D = loadmat(data_dir + 'Digits_train.mat')
# D.keys()
X = D['X'] # images
y = D['y'] # labels: single output in 0~9
```

[9]下載點 https://ntpuccw.blog/supplements/matlab-in-statistical-computing/

```
plt.figure(figsize = (9,6))
# prepare and diaplay a montage of digit images
n, m = 20, 30  # A n x m montage (total mn images)
sz = np.sqrt(X.shape[1]).astype('int') # image size sz x sz
M = np.zeros((m*sz, n*sz)) # montage image
A = X[:m*n,:] # show the first nm images
# Arrange images to form a montage
for i in range(m) :
    for j in range(n) :
        M[i*sz: (i+1)*sz, j*sz:(j+1)*sz] = \
            A[i*n+j,:].reshape(sz, sz)

plt.imshow(M.T, cmap = plt.cm.gray_r, \
        interpolation = 'nearest')
plt.xticks([])
plt.yticks([])
plt.title('The Montage of handwriting digits')
plt.show()
```

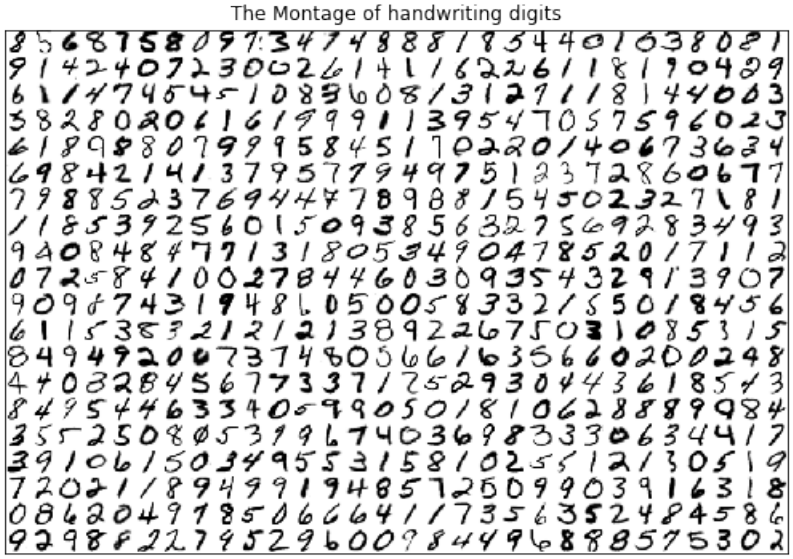


圖 10: 手寫數字

手寫數字檔以 MATLAB 的檔案格式儲存，因此採用 scipy.io 的 loadmat 指令取得。檔案內的 X 變數為 1000 × 784 的影像矩陣，即 1000 張影像，每張大小為 28 × 28 的正方形影像，拉開成一 1 × 784 的向量。將這樣的矩陣從最上面取 600 張，製作成如圖 10 的 20 × 30 蒙太奇圖陣（Montage），需要將 1 × 784 的向量 reshape 成 28 × 28 的影像矩陣，並一一置入蒙太奇圖陣的矩陣 M。程式碼中，迴圈內的矩陣空間安排是較為費心，請細心解讀，方便以後模仿。此外，檔案內的輸出變數 y 為 1000 的向量，輸出值為 0 到 9，代表輸入資料的類別（同手寫數字）。

接著開始進行神經網路的設定與訓練。與前段機器手臂的應用不同的是，手寫辨識屬於群組判別（共 10 個群組），其輸出為類別資料，因此採 sklearn.neural_network 的另一個模組 MLPClassifier，如其名所示，這是作為分類器（Classifier）用途的多層次感知器（Multi-Layer Perceptron）。以下程式碼展示 MLPClassifier 的使用方式（程序大致如上一節的 MLPRegressor）：

```
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import plot_confusion_matrix

# prepare data
X_train, X_test, y_train, y_test = \
    train_test_split(X/255, y.ravel(), test_size = 0.25)

# setup and run
hidden_layers = (30,) # one hidden layer
# solver = 'sgd'   # not efficient, need more tuning
# solver = 'lbfgs' # not suitable here
solver = 'adam'    # default solver
clf = MLPClassifier(max_iter = 10000, solver = solver,
    hidden_layer_sizes = hidden_layers, verbose = True,
    activation = 'logistic', tol = 1e-6, random_state = 0)
    # default activation = 'relu'
clf.fit(X_train, y_train)
y_test_hat = clf.predict(X_test)
```

程式碼 MLPClassifier() 建立了一個分類的學習器，其中幾個參數（Attributes）說明如后：max_iter = 10000 代表演算法的遞迴次數上限，不管收斂與否，強制停止；solver = 'adam' 演算法採預設的 adam，適合較大的數據量。讀者可以試試其他演算法，看看結果如何；verbose = True 將遞迴過程的目標函數值（在此為損失函數 Loss function）列印出來，可以藉由函數值遞減的速度，觀察學習的過程順利與否。有些時候，訓練過程會卡在某個區域極小值的坑洞裡爬不出來，最後的結果通常不理想，因此要考慮重新訓練，給予神經網路不同的初始值；random_state = 0 設定了固定的隨機亂數產生的位置，也就是每次執行，神經網路內部使用的隨機亂數都會一樣。當我們希望內部的隨機亂數每次都不同時，這個參數不能給固定的數字；activation = logistic 代表唯一的隱藏層的啟動函數（又稱轉換函數），[10] 其他的選項如 tanh, relu。

上述程式碼最後一行用於預測測試資料的輸出結果。如果要計算訓練資料或測

[10] logistic function 定義為 $f(x) = \frac{1}{1+e^{-x}}$

試資料的預測能力，可以使用現成的指令與 Confusion matrix，參考作法如下：

```
from sklearn.metrics import plot_confusion_matrix

score = clf.score(X_test, y_test)
# Confusion matrix
plot_confusion_matrix(clf, X_test, y_test,
    cmap = plt.cm.Blues, normalize = 'true')
```

圖 11 呈現了對 250 筆測試資料的混淆矩陣（Confusion matrix），格子內的數字代表該群組的預測準確率，其中以對「3」與「5」的判斷表現最差。[11]圖 12(a) 則是更改指令 plot_confusion_matrix 內的參數 normalized = 'false' ，呈現實際的準確數字，而圖 12(b) 展示訓練過程中損失函數遞減的趨勢，可以看出大約前 1/4 陡降，之後便緩步下降，直到滿足設定的損失函數在連續 10 次遞迴，其變化皆小於設定的 tol = 1e-6。

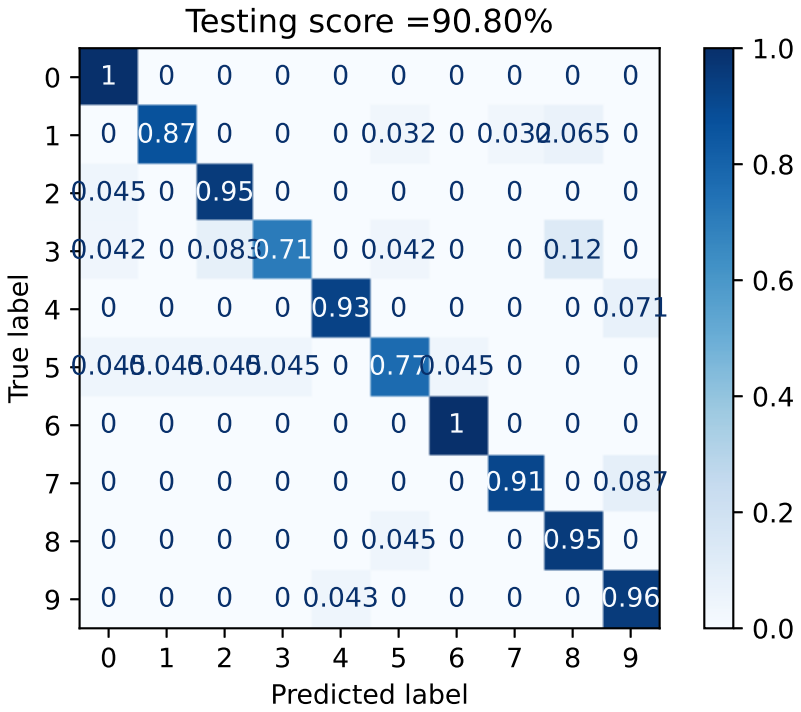
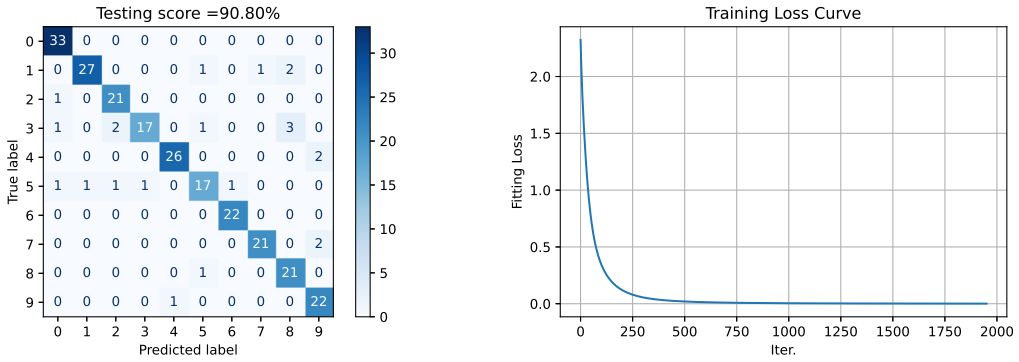


圖 11: MLPClassifier 含一個隱藏層（30 個神經元）對測試資料的預測能力以 Confusion matrix 表示。

圖 12(b) 的損失函數遞減趨勢，也在訓練過後被留在這裡：

```
plt.plot(clf.loss_curve_)
```

[11]看到這個結果，我們必須回頭去看原始的數字圖，如圖 10，感受一下是否這兩個數字圖特別容易混淆？譬如，回頭看圖 11 在 true label 為 3 的那一列，被判為 8 的比例高達 12%，是所有錯判率最高的一項。



(a) Confusion Matrix　　(b) 訓練誤差

圖 12: `MLPClassifier` 一個隱藏層（含 30 個神經元）的 (a) 測試資料的預測能力與 (b) 訓練過程誤差的遞減。

`clf = MLPClassifier()` 訓練過後，使用者可以從 `clf` 找到許多關於該神經網路的設定與執行過程中的數據，譬如，除了上述的 `clf.loss_curve_`，還有：

```
clf.loss_          # The current value of loss function
clf.best_loss_
clf.n_layers_      # input layer is counted
clf.n_outputs_     # Number of outputs
clf.out_activation_ # softmax is employed here
clf.n_iter_        # The total number of iterations
clf.t_             # The number of training samples
clf.classes_       # Class labels for each output.
clf.get_params(deep=True) # get all parameters
```

讀者不妨列印出來看看，必要時查詢使用手冊。其中值得一提的是 `clf.out_activation_`。這是紀錄神經網路的輸出函數，得到的答案是 `softmax`。這是群組判別最常用的輸出函數，一般來說，當有 `10` 個群組別時，`softmax` 有 `10` 個輸出，輸出值為屬於該類別的機率，全部相加為 `1`。`MLPClassifier` 將這 `10` 個輸出合併為一個，即機率值最高的那個群組別。

4 觀察與延伸

1. 延伸前述指令 randsphere 能在高維度球體內產生均勻的亂數，下述程式碼繪製了圖 13。

```
center = np.array([0, 0, 0])
radius = 1
n = 1000
```

```
fig = plt.figure(figsize=(6,6), dpi=300)
ax = fig.add_subplot(projection='3d')

u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
x = radius * np.outer(np.cos(u), np.sin(v))
y = radius * np.outer(np.sin(u), np.sin(v))
z = radius * np.outer(np.ones(np.size(u)), np.cos(v))

# Plot the surface
ax.plot_surface(x, y, z, color = '#CFF5D9', alpha = 0.3)
ax.plot(np.sin(u), np.cos(u), 0, color='k', linewidth = 1)
ax.plot([0]*100,np.sin(u),np.cos(u), color='k', \
        linewidth = 1, linestyle = 'dashed')
P = randsphere(center, radius, n)
for i in range(n):
    ax.scatter(P[i, 0], P[i, 1], P[i, 2], marker = '*', \
                c = '#E2A428', s = 2)

ax.view_init(elev = 25, azim = -61)
ax.set_xticks([-1, 0, 1])
ax.set_yticks([-1, 0, 1])
ax.set_zticks([-1, 0, 1])
plt.show()
```

2. 在機器人手臂的神經網路訓練，本文採用了 sklearn.neural_network.MLPRegresso 與 NeuroLab.net.newff 兩種套件。但是在圖形辨識方面，只採用 sklearn.neural_network 的另一個產品 MLPClassifier。讀者可以試著用 NeuroLab.net.newff 配合輸出層的轉換函數為 SoftMax，是否也能展現辨識能力？其實，不管 NeuroLab.net.newff 做為圖形辨識或群組分類的能力如何，它缺乏後援的支持，早已不是好的選項，譬如 NeuroLab.net 並沒有提供混淆矩陣的繪製，使用者必須自己繪製，便會令人卻步。

5 習題

1. 繪製式 (2) 的函數圖，其中 $c_1 = 1.005, c_2 = 1$。
2. 試著產生如圖 3 的訓練資料 100 筆，讓這些資料盡量均勻地散佈在機器手臂能及的範圍內。
3. 在機器手臂資料的訓練中測試參數 early_stopping 的驗證效果。根

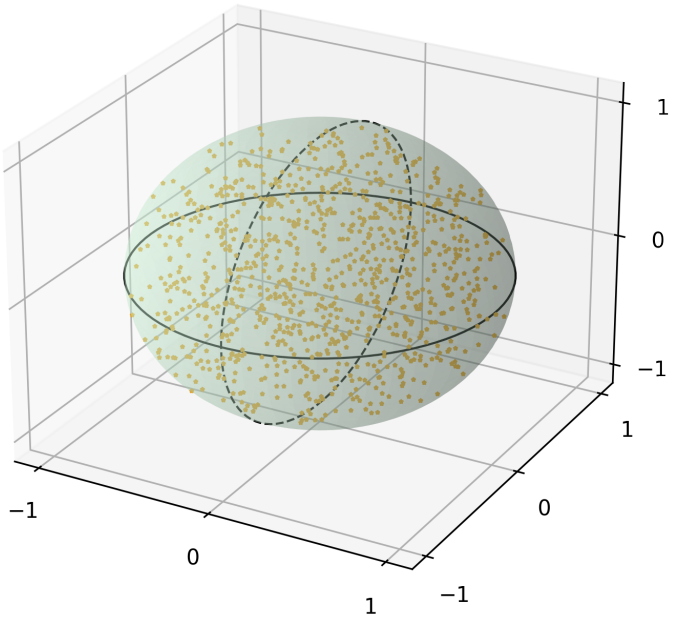


圖 13: 使用程式 randspere 繪製單位球體內均勻散佈的點

據 MLPRegresor 的使用規則，early_stopping 僅適用於演算法為 sgd, adam 兩種。

4. 比較 sklearn.neural_network 的 MLPRegresor 模組與 neuroLab 的 net.netff 模組，在相同隱藏層神經元數量與使用相同演算法的前提下，比較兩者的預測能力。

5. 自行找一個題目適用於 Neural Network Fitting 的類神經網路訓練（如機器手臂的學習）。輸入與輸出資料皆為連續型，來自現成資料或自行產生資料皆可。

6. 文中對數字資料的訓練與測試僅來自 1000 筆資料。請自原始資料中，擷取更多的資料作為訓練，評估訓練數量對預測精準度的影響。下列指令可以自 sklearn.datasets 中取得 70,000 筆資料：

```
from sklearn.datasets import fetch_openml

X, y = fetch_openml('mnist_784', version =1,\
        return_X_y = True)
```

7. 請自數字資料裡選取未接受訓練的數字資料，並寫一支程式來測試已訓練好的類神經網路的判斷能力。

8. 本章舉數字判讀作為圖形識別的範例，請讀者模仿本章的做法，但使用不同的圖形，譬如手寫英文字母的判別。所需的圖形可以上網搜尋取得。[12]

9. 在 sklearn.neural_network 的 MLPRegresor 與 MLPClassifier 兩個神經網路模組中，還有一個重要的參數 early_stopping。顧名思義，這個參數可以讓訓練過程提早結束。而需要提早結束的理由來自原來設定的結束條件，即損失函數在經過預設的 10 次遞迴後，變化的幅度小於 tol（預設為 1e-4）。這個停止條件毫無根據，只是強制演算法必須停止而已，至於變化的幅度要設定多小，也是經驗值。設定太小，會造成訓練時間過長，而且可能導致過度訓練（Over-fitting）；設定值太大，會造成訓練不足。early_stopping = True 則是在訓練過程中，撥出一部分資料（預設為 10%）做為查核之用（Validate），當查核資料的誤差持續 10 次遞迴不再遞減時（即不減反增），則演算法必須提早結束，以避免過度學習。此時，正式的結束點定位在查核資料誤差最小的地方。請讀者試著引入 early_stopping = True，並比較查核與否的對於實際預測能力的影響。

[12]https://www.kaggle.com/sachinpatel21/az-handwritten-alphabets-in-csv-format