Say Hi to CNN

Yung-Chin Yen

July 6, 2022

Outline

1 神經網路解題步驟

使用神經網路解決問題可大致分為兩個步驟:「學習」與「推論」。

- 學習指使用訓練資料進行權重參數的學習
- 推論指使用學習過的參數進行資料分類 CNN

而實際的動手實作可再細分為以下幾個步驟

1. 收集資料 (Gathering data)
2. 準備數據 (Preparing that data)
3. 選擇模型 (Choosing a model)
4. 訓練機器 (Training)
5. 評估分析 (Evaluation)
6. 調整參數 (Hyperparameter tuning)
7. 預測推論 (Prediction)

實際動手玩一下神經網路架構: Tensorflow Playground

2 收集資料

2.1 資料類型

1. 人工收集
   - 預測股市股價: 開盤、收盤、成交量、技術指標、財務指標、籌碼指標等等
   - 以物品識別: 大量物品照片並給予名稱 (label)
   - 以注音符號手寫辨識: 大量手寫照片及其對應答案 (label)

2. 現成資料集
   (a) MNIST 資料集由 0~9 的數字影像構成 (如圖1)，共計 60000 張訓練影像、10000 張測試影像。

Figure 1: MNIST 資料集內容範例

   (b) Boston housing Boston Housing 數據集包含有關波士頓不同房屋的數據 (數據，如年份、面積)，本資料集中有 506 個樣本和 13 個特徵變量，目標是使用給定的特徵預測房屋價格的價值。

   (c) Iris 鳶尾花資料集是非常著名的生物資訊資料集之一，由英國統計學家 Ronald Fisher 在 1936 年時，對加斯帕半島上的鳶尾屬花朵所提取的花瓣花萼的長寬數據資料，依照山鳶尾，變色鳶尾，維吉尼亞鳶尾三類進行標示，共 150 筆資料[1]。每筆資料有五個欄位：花萼長度 (Sepal Length)、花萼寬度 (Sepal Width)、花瓣長度 (Petal Length) 、花瓣寬度 (Petal Width)、類別 (Class)，其中類有 Setosa，Versicolor 和 Virginica 三個品種。

   (d) Cifar-10 由深度學習大師 Geoffrey Hinton 教授與其在加拿大多倫多大學的學生 Alex Krixhevsky 與 Vinoid Nair 所整理之影像資料集, 包含 6 萬筆 32*32 低解析度之彩色圖片, 其中 5 萬筆為訓練集; 1 萬筆為測試集, 是機器學習中常用的圖片辨識資料集

[1]機器學習資料集 - Iris dataset

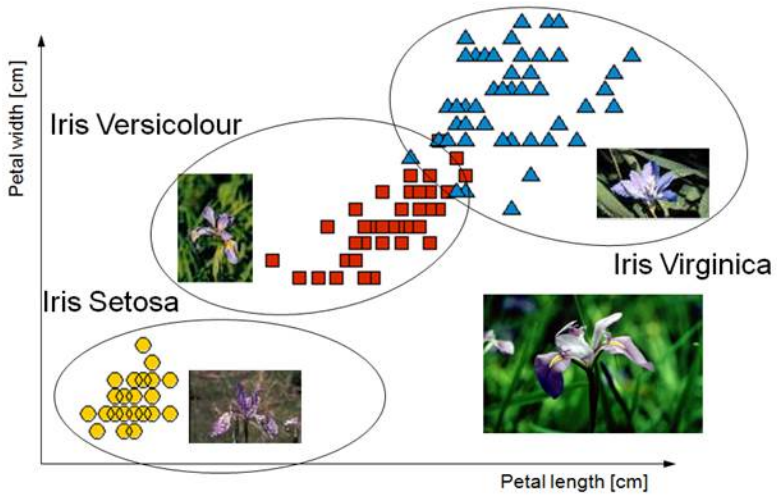


Figure 2: Iris 資料集

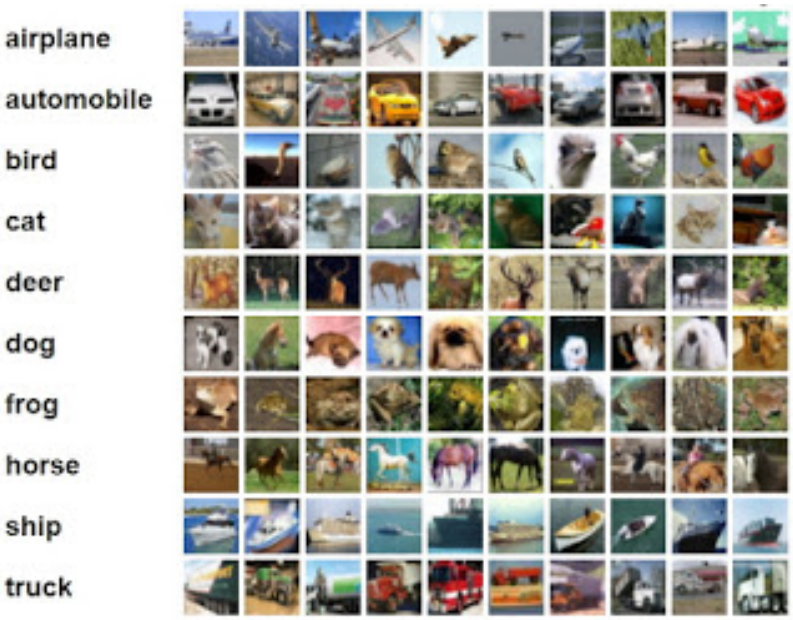


Figure 3: Cifar-10

2.2 DEMO

mnist.load_data(): 自網路下載資料集 (內容會包含 資料 (x) 和 標籤 (y) 兩部份), load_data() 這個 function 會把整份資料集分割成兩個子集合：訓練集 (training set) 和測試集 (testing set)。

```
from keras.datasets import mnist
(x_Train, y_Train), (x_Test, y_Test) = mnist.load_data()
```

3 準備數據 (Preparing that data)

當我們在比較分析兩組數據資料時，可能會遭遇因單位的不同 (例如：身高與體重)，或數字大小的代表性不同 (例如：粉專 1 萬人與滿足感 0.8)，造成各自變化的程度不一，進而影響統計分析的結果[2]。會造成什麼影響--那些變化量最大的因素（特徵）會主導分析結果。

3.1 特徵 (feature)

什麼是特徵? 每一筆資料由不同類型的資料 (數值、字串、布林值...) 組成，以 Iris 資料集為例，每朵花有四種數值，這些都是它的特徵 (feature)，我們的目的就是藉由分析這些特徵來預測花的種類。

下列程式利用 scikit learn 下載線上的資料集，先以 pandas 轉換 Iris 的四種特徵值為：

```
1 from sklearn import datasets
2 iris = datasets.load_iris()
3 print(iris.data[:5])
4 print(iris.feature_names)
```

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

```
1 from sklearn import datasets
2 import pandas as pd
3 iris = datasets.load_iris()
4 print(type(iris))
5
6 df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
7 print(df.head())
8 print(df.columns)
```

```
<class 'sklearn.utils.Bunch'>
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
```

[2]資料的正規化 (Normalization) 及標準化 (Standardization)

```
1                 4.9               3.0               1.4               0.2
2                 4.7               3.2               1.3               0.2
3                 4.6               3.1               1.5               0.2
4                 5.0               3.6               1.4               0.2
Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)'],
      dtype='object')
```

上述程式將 scikit learn 所下載的資料集先以 pandas 轉換 Iris 的四種特徵值為：

- sepal length (cm)
- sepal width (cm)
- petal length (cm)
- petal width (cm)

3.2 資料正規化

資料的正規化 (Normalization) 是將原始資料的數據按比例縮放於 [0, 1] 區間中，且不改變其原本分佈。

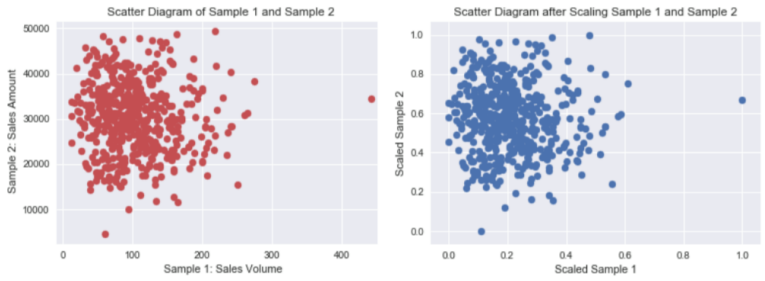


Figure 4: 資料正規化

正規化有兩種常用的方法，可以將不同規模的特徵轉化為相同的規模：常態化 (normalization) 和標準化 (standardization)：

3.3 常態化

將特徵值縮化為 0~1 間，這是「最小最大縮放」(min-max scaling) 的一個特例，某一特徵值的常態化做法如下：

$$x_{norm}^{i} = \frac{x^{i} - x_{min}}{x_{max} - x_{min}}$$

若以 scikit-learn 套件來完成實作，其程式碼如下：

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.fit_transform(X_test)
```

3.4 標準化

雖說常態化簡單實用，但對許多機器學習演算法來說 (特別是梯度下降法的最佳化)，標準化則更為實際，我們可令標準化後的特徵值其平均數為 0、標準差為 1，這樣一來，特徵值會滿足常態分佈，進而使演算法對於離群值不那麼敏感。標準化的公式如下：

$$x_{std}^{i} = \frac{x^{i} - \mu_{x}}{\sigma_{x}}$$

若以 scikit-learn 套件來完成實作，其程式碼如下：

```
from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

3.5 DEMO

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255



y_TrainOneHot = np_utils.to_categorical(y_Train)
y_TestOneHot = np_utils.to_categorical(y_Test)
```

4 選擇模型 (Choosing a model)

當數據都進行整理後，接下來就是要選擇訓練用的模型，像是決策樹、LSTM、RNN 等等都是機器學習中常使用的訓練模型，其中目前較常拿來訓練股市的是「LSTM」，中文叫做長短期記憶，是屬於深度學習中的一個模型。另一種 CNN 模型則適合處理圖形資料。

4.1 語法

Keras API reference / Models API / Model training APIs

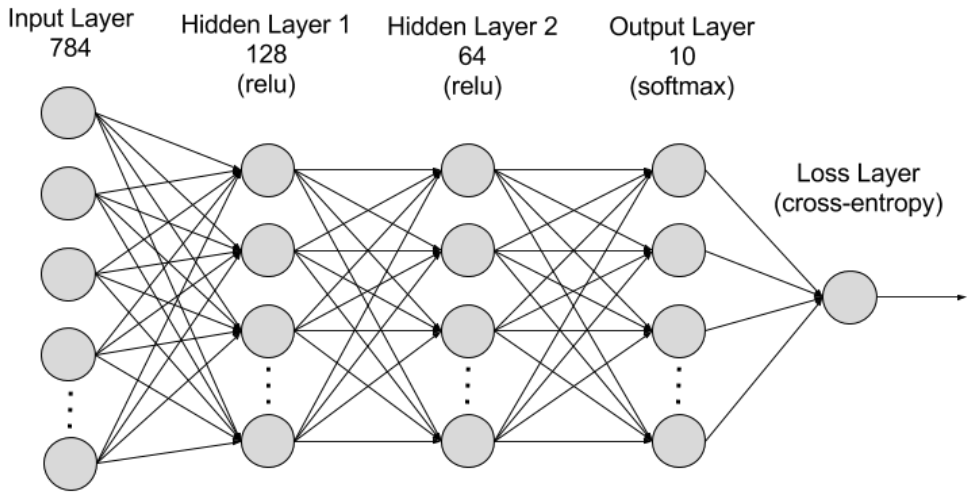
4.2 DEMO

1. LSTM 模型示例

```
model = Sequential()
model.add(LSTM(128,
               input_shape=(x_train.shape[1:]),
               activation='relu',
               return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(128, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

2. CNN 模型示例

```
model = Sequential()
model.add(Dense(units=128,
                input_dim=784,
                kernel_initializer='normal',
                activation='relu'))
model.add(Dense(64, activation='relu')
model.add(Dense(units=10,
                kernel_initializer='normal',
                activation='softmax'))
```



pxpx

Figure 5: MNIST-NeuralNet

5 訓練機器 (Training)

選擇好訓練模型後，再來要將訓練集資料丟進去模型中做訓練，每層要放多少神經元、要跑幾層等等都會影響模型訓練出來的結果，這部分只能靠經驗跟不斷嘗試去學習，或是上網多爬文看別人怎麼撰寫訓練模型。

在真正訓練前應該再設定好模型的 loss function, optimizer。

5.1 語法

Keras API reference / Models API / Model training APIs

5.2 DEMO

1. LSTM

```
# ptimizer, loss function
model.compile(optimizer=Adam(lr=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          e pochs=3,
          validation_data=(x_test, y_test))
```

2. CNN

```
# optimizer, loss function
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])

model.fit(x=x_Train,
          y=y_TrainOneHot,
          validation_split=0.2,
          epochs=5, batch_size=30, verbose=2)
```

6 評估分析 (Evaluation)

當模型訓練完成後，接下來就是判斷該模型是否有過度擬合 (overfitting)，這裡就是帶入測試集的資料進行評估，也可以嘗試利用交叉驗證的方式進行模型的擬合性判斷，以及利用 RESM、MSE 等統計計算來判斷模型的準確度

```
scores = model.evaluate(x_Train, y_TestOneHot)
```

7 調整參數 (Hyperparameter tuning)

到這大致上模型已經完成了 50%，最後的一步就是進行參數的微調，我們也稱為「超參數 (Hyperparamters)」，讓整個模型更加的精準，但也不能過度的調整，因為會造成 overfitting 的結果，這個取捨就只能依照無窮盡的反覆迭帶去尋找了，這部分也是相對較耗時間的地方

7.1 model 參數

- 調整 model 架構: [機器學習 ML NOTE] CNN 演化史 (AlexNet、VGG、Inception、ResNet)+Keras Coding
- loss function: `https://keras.io/api/losses/`
- optimizers: `https://keras.io/api/optimizers/`

7.2 Hyperparameters

- batch size：一次迭代放入進行訓練或測試的影像數量。

- epoch：一種單位，所有影像皆被計算過 1 次後即為 1 epoch

- CNN 筆記 - 超參數 (Hyperparamters)

8 預測推論 (Prediction)

到此，模型已經正式完成。那，怎麼知道這個模型夠不夠優秀？能不能正常運作？最簡單的評估方式是以一筆新的資料來測試。

```
prediction = model.predict_classes(x_Test4D_normalize)
print(prediction[:10])
```

以 MNIST 手寫數字辨識為例，我們可以自己寫一個數字餵給模型，看看模型是否能正確預測。

對於模型來說，這類全新的數據則是一個未知數，如果我們在訓練與測試階段都用同一批資料，這就好像上課與考試的內容都一樣，這會導致學生其實沒有學到真正的知識，而考試得到的分數也不會太準確。這種訓練資料與測試資料過於雷同而導致模型對新型別的資料欠缺處理能力的問題即為 過度擬合 (overfitting)；反之，如果我們的模型對於各種新型態資料都能辨識，則這個模型就具有 泛化 (Generalization) 的預測能力。

9 DEMO 1: Regression

CNN 雖然更多場合是用於處理影像相關的處理工作，然而他也可以用來幫我們解決一些數學問題，例如畫迴歸線。開啟 colab、新增一個筆記本，逐一將下列程式碼複製、貼到 colab 執行，觀察結果。

9.1 產生數據

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

x = np.random.uniform(0.0, 3, (10))
y = 78 + 7.8*x + np.random.normal(0.0, 3, len(x))

plt.scatter(x, y)
```

結果如下圖，一共有 10 個點，如何畫出一條迴歸線代表這 10 個點的趨勢？這個題目本身是什麼意思？

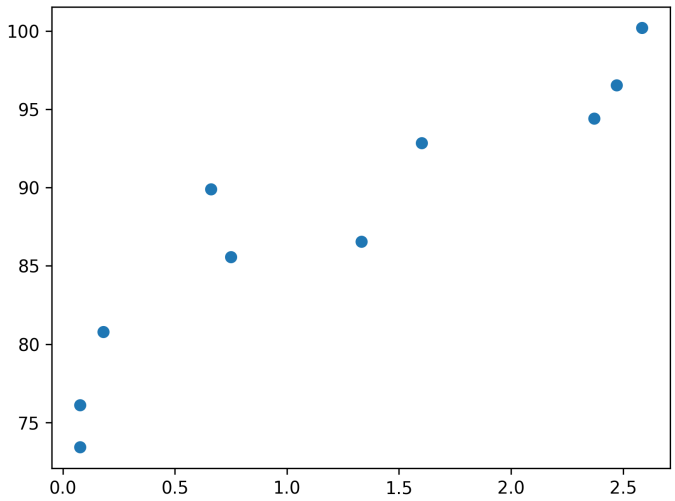


Figure 6: Caption

想像一下，上面是 10 個學生的資料，X 軸是學生對於數學的喜愛程度，Y 軸是該生的數學期末考成績，看起來二者間是存在某種關係的。那，如果已知某生對數學的喜愛程為 2.0，我們可以預測他的數學期末考成績嗎？---可以的，如果我們有一條像底下的迴歸線公式：

$$MathScore_i = a * MathLove_i + b$$

如此一來，我們就能輸入喜愛程度，得到對數學成績的預測結果。

當樣本數不太多，而你的計算能力也不弱時，也許你可以手動將上述公式中的 a 與 b 求出，得到一條 $y = ax + b$ 的迴歸線，但萬一樣本數有 200 個點呢

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

x = np.random.uniform(0.0, 3, (2000))
y = 78 + 7.8*x + np.random.normal(0.0, 3, len(x))
plt.cla()
plt.scatter(x, y)
plt.savefig("images/cnn-regression-2.png", dpi=300)
```

這段程式碼利用 numpy 套件的隨機函數產生 2000 個點，接下來我們想建一個 CNN 模型來幫我們畫一條迴歸線。

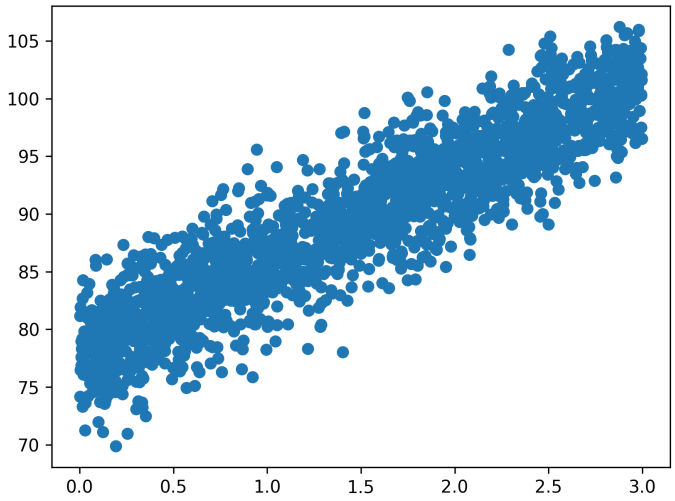


Figure 7: Caption

9.2 以 scikit learn 的線性迴歸模組來解決

```
from sklearn import linear_model
regr=linear_model.LinearRegression()

x = x.reshape(-1, 1)
# 改變x的格式形狀,
```

```
# 原本x為 [x1, x2, x3, ...]
# 要改格式符合scikit learn迴歸模組的需求, 變成: [ [x1], [x2], [x3], ...]
y = y.reshape(-1, 1)
regr.fit(x, y)

#短短兩行code, 模型已經建構完成了! 接下來, 我們來看看訓練集的成果。
plt.clf()
plt.scatter(x,y)
plt.plot(x, regr.predict(x), color='red', linewidth=4)
print("====實際(前三筆)====")
print(y[:3])
print("====預測(前三筆)====")
print(regr.predict(x[:3]))
```

```
====實際(前三筆)====
[[91.12584105]
 [94.36218262]
 [92.10536141]]
====預測(前三筆)====
[[92.70522144]
 [96.66162023]
 [89.95865672]]
```

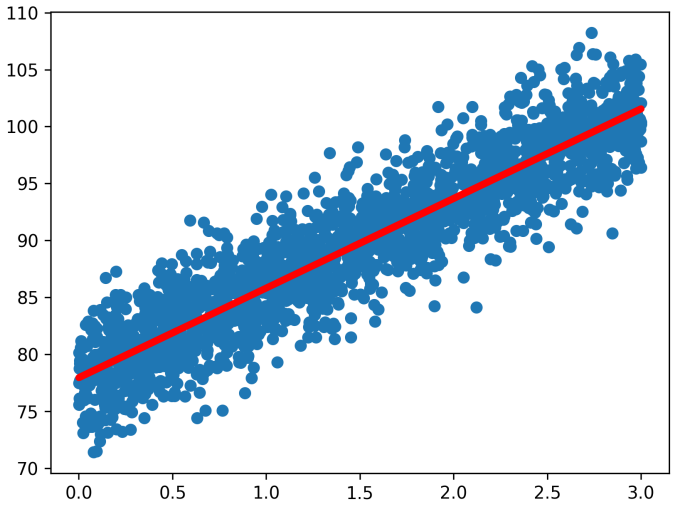


Figure 8: Caption

9.3 建立 model 來畫迴歸線

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout

# A simple regression model
model = Sequential()
model.add(Dense(4, input_shape=(1,)))
model.add(Dropout(0.5))
model.add(Dense(8, input_shape=(1,)))
model.add(Dropout(0.5))
model.add(Dense(1, input_shape=(1,)))
model.compile(loss='mse', optimizer='rmsprop')
print(model)
```

```
<keras.engine.sequential.Sequential object at 0x7f96fca66400>
```

9.4 訓練 model

```
# 將實際資料分為訓練集與測試集
x_Train = x[:1500]
x_Test = x[1500:]
y_Train = y[:1500]
y_Test = y[1500:]

# The fit() method - trains the model
train_history = model.fit(x=x_Train, y=y_Train,
                          validation_split=0.2,
                          epochs=100, batch_size=200,
                          verbose=2)
print(train_history)
```

```
Epoch 1/100
6/6 - 1s - loss: 8163.1743 - val_loss: 8152.0234 - 676ms/epoch - 113ms/step
Epoch 2/100
6/6 - 0s - loss: 8130.1035 - val_loss: 8134.6201 - 18ms/epoch - 3ms/step
Epoch 3/100
6/6 - 0s - loss: 8115.6943 - val_loss: 8119.0532 - 19ms/epoch - 3ms/step
Epoch 4/100
6/6 - 0s - loss: 8093.0952 - val_loss: 8105.3452 - 22ms/epoch - 4ms/step
Epoch 5/100
```

```
6/6 - 0s - loss: 8088.0625 - val_loss: 8091.9082 - 21ms/epoch - 4ms/step
Epoch 6/100
6/6 - 0s - loss: 8076.4019 - val_loss: 8078.7568 - 21ms/epoch - 4ms/step
Epoch 7/100
6/6 - 0s - loss: 8065.8623 - val_loss: 8065.9023 - 22ms/epoch - 4ms/step
Epoch 8/100
6/6 - 0s - loss: 8049.2852 - val_loss: 8053.2207 - 21ms/epoch - 4ms/step
Epoch 9/100
6/6 - 0s - loss: 8037.1641 - val_loss: 8040.7090 - 21ms/epoch - 4ms/step
Epoch 10/100
6/6 - 0s - loss: 8017.3110 - val_loss: 8028.0156 - 22ms/epoch - 4ms/step
Epoch 11/100
6/6 - 0s - loss: 8007.4634 - val_loss: 8015.3794 - 21ms/epoch - 4ms/step
Epoch 12/100
6/6 - 0s - loss: 7997.4258 - val_loss: 8002.3467 - 22ms/epoch - 4ms/step
Epoch 13/100
6/6 - 0s - loss: 7980.9893 - val_loss: 7989.1118 - 21ms/epoch - 4ms/step
Epoch 14/100
6/6 - 0s - loss: 7971.9585 - val_loss: 7975.1201 - 22ms/epoch - 4ms/step
Epoch 15/100
6/6 - 0s - loss: 7952.8374 - val_loss: 7960.9668 - 23ms/epoch - 4ms/step
Epoch 16/100
6/6 - 0s - loss: 7943.7275 - val_loss: 7946.5649 - 21ms/epoch - 4ms/step
Epoch 17/100
6/6 - 0s - loss: 7924.2524 - val_loss: 7931.0952 - 21ms/epoch - 4ms/step
Epoch 18/100
6/6 - 0s - loss: 7918.0317 - val_loss: 7915.4448 - 21ms/epoch - 3ms/step
Epoch 19/100
6/6 - 0s - loss: 7889.4800 - val_loss: 7898.3384 - 21ms/epoch - 3ms/step
Epoch 20/100
6/6 - 0s - loss: 7884.0850 - val_loss: 7881.2534 - 21ms/epoch - 4ms/step
Epoch 21/100
6/6 - 0s - loss: 7857.5469 - val_loss: 7862.9492 - 21ms/epoch - 3ms/step
Epoch 22/100
6/6 - 0s - loss: 7836.8652 - val_loss: 7843.7798 - 20ms/epoch - 3ms/step
Epoch 23/100
6/6 - 0s - loss: 7831.5474 - val_loss: 7824.1558 - 21ms/epoch - 4ms/step
Epoch 24/100
6/6 - 0s - loss: 7799.3833 - val_loss: 7803.1401 - 21ms/epoch - 4ms/step
```

```
Epoch 25/100
6/6 - 0s - loss: 7788.5176 - val_loss: 7781.8169 - 21ms/epoch - 3ms/step
Epoch 26/100
6/6 - 0s - loss: 7762.7993 - val_loss: 7759.5225 - 21ms/epoch - 4ms/step
Epoch 27/100
6/6 - 0s - loss: 7742.5601 - val_loss: 7736.4365 - 21ms/epoch - 3ms/step
Epoch 28/100
6/6 - 0s - loss: 7710.9624 - val_loss: 7711.6831 - 20ms/epoch - 3ms/step
Epoch 29/100
6/6 - 0s - loss: 7686.9023 - val_loss: 7686.6118 - 21ms/epoch - 3ms/step
Epoch 30/100
6/6 - 0s - loss: 7654.4434 - val_loss: 7660.2993 - 22ms/epoch - 4ms/step
Epoch 31/100
6/6 - 0s - loss: 7616.2686 - val_loss: 7632.8784 - 21ms/epoch - 4ms/step
Epoch 32/100
6/6 - 0s - loss: 7601.3926 - val_loss: 7605.4316 - 21ms/epoch - 3ms/step
Epoch 33/100
6/6 - 0s - loss: 7580.7285 - val_loss: 7577.3242 - 21ms/epoch - 4ms/step
Epoch 34/100
6/6 - 0s - loss: 7539.0000 - val_loss: 7547.3091 - 21ms/epoch - 3ms/step
Epoch 35/100
6/6 - 0s - loss: 7543.3491 - val_loss: 7518.0386 - 21ms/epoch - 3ms/step
Epoch 36/100
6/6 - 0s - loss: 7490.0493 - val_loss: 7486.1777 - 21ms/epoch - 3ms/step
Epoch 37/100
6/6 - 0s - loss: 7471.3384 - val_loss: 7454.4219 - 22ms/epoch - 4ms/step
Epoch 38/100
6/6 - 0s - loss: 7425.4585 - val_loss: 7420.8564 - 21ms/epoch - 4ms/step
Epoch 39/100
6/6 - 0s - loss: 7391.1665 - val_loss: 7386.5859 - 21ms/epoch - 4ms/step
Epoch 40/100
6/6 - 0s - loss: 7364.9375 - val_loss: 7351.4951 - 21ms/epoch - 3ms/step
Epoch 41/100
6/6 - 0s - loss: 7325.8232 - val_loss: 7315.1899 - 19ms/epoch - 3ms/step
Epoch 42/100
6/6 - 0s - loss: 7300.6885 - val_loss: 7278.2241 - 20ms/epoch - 3ms/step
Epoch 43/100
6/6 - 0s - loss: 7231.8750 - val_loss: 7238.9932 - 21ms/epoch - 3ms/step
Epoch 44/100
```

```
6/6 - 0s - loss: 7228.6323 - val_loss: 7199.8735 - 21ms/epoch - 3ms/step
Epoch 45/100
6/6 - 0s - loss: 7155.6665 - val_loss: 7158.5669 - 21ms/epoch - 3ms/step
Epoch 46/100
6/6 - 0s - loss: 7117.6025 - val_loss: 7116.5669 - 21ms/epoch - 3ms/step
Epoch 47/100
6/6 - 0s - loss: 7098.7085 - val_loss: 7073.9907 - 20ms/epoch - 3ms/step
Epoch 48/100
6/6 - 0s - loss: 7033.6831 - val_loss: 7029.5801 - 20ms/epoch - 3ms/step
Epoch 49/100
6/6 - 0s - loss: 6993.9448 - val_loss: 6984.3750 - 20ms/epoch - 3ms/step
Epoch 50/100
6/6 - 0s - loss: 6963.6758 - val_loss: 6938.3232 - 21ms/epoch - 3ms/step
Epoch 51/100
6/6 - 0s - loss: 6873.2153 - val_loss: 6890.0005 - 21ms/epoch - 4ms/step
Epoch 52/100
6/6 - 0s - loss: 6834.0566 - val_loss: 6841.3877 - 21ms/epoch - 4ms/step
Epoch 53/100
6/6 - 0s - loss: 6839.6260 - val_loss: 6792.6797 - 21ms/epoch - 3ms/step
Epoch 54/100
6/6 - 0s - loss: 6761.8066 - val_loss: 6741.7358 - 20ms/epoch - 3ms/step
Epoch 55/100
6/6 - 0s - loss: 6719.5601 - val_loss: 6689.5752 - 21ms/epoch - 3ms/step
Epoch 56/100
6/6 - 0s - loss: 6646.5962 - val_loss: 6636.4600 - 21ms/epoch - 4ms/step
Epoch 57/100
6/6 - 0s - loss: 6676.9526 - val_loss: 6584.4219 - 21ms/epoch - 4ms/step
Epoch 58/100
6/6 - 0s - loss: 6580.0708 - val_loss: 6529.2104 - 21ms/epoch - 4ms/step
Epoch 59/100
6/6 - 0s - loss: 6487.5444 - val_loss: 6471.6338 - 22ms/epoch - 4ms/step
Epoch 60/100
6/6 - 0s - loss: 6457.2759 - val_loss: 6414.0923 - 21ms/epoch - 3ms/step
Epoch 61/100
6/6 - 0s - loss: 6427.3369 - val_loss: 6355.9165 - 21ms/epoch - 3ms/step
Epoch 62/100
6/6 - 0s - loss: 6307.2007 - val_loss: 6295.6885 - 21ms/epoch - 4ms/step
Epoch 63/100
6/6 - 0s - loss: 6279.2075 - val_loss: 6235.0156 - 21ms/epoch - 4ms/step
```

```
Epoch 64/100
6/6 - 0s - loss: 6201.4722 - val_loss: 6172.8394 - 21ms/epoch - 4ms/step
Epoch 65/100
6/6 - 0s - loss: 6166.4458 - val_loss: 6110.2739 - 21ms/epoch - 4ms/step
Epoch 66/100
6/6 - 0s - loss: 6121.9590 - val_loss: 6046.8467 - 21ms/epoch - 3ms/step
Epoch 67/100
6/6 - 0s - loss: 6035.1187 - val_loss: 5981.7637 - 20ms/epoch - 3ms/step
Epoch 68/100
6/6 - 0s - loss: 5963.8081 - val_loss: 5914.8799 - 21ms/epoch - 4ms/step
Epoch 69/100
6/6 - 0s - loss: 5909.1968 - val_loss: 5848.0874 - 22ms/epoch - 4ms/step
Epoch 70/100
6/6 - 0s - loss: 5834.0771 - val_loss: 5780.1626 - 21ms/epoch - 4ms/step
Epoch 71/100
6/6 - 0s - loss: 5818.0547 - val_loss: 5712.0708 - 21ms/epoch - 4ms/step
Epoch 72/100
6/6 - 0s - loss: 5704.0469 - val_loss: 5641.2202 - 22ms/epoch - 4ms/step
Epoch 73/100
6/6 - 0s - loss: 5635.4438 - val_loss: 5570.3872 - 21ms/epoch - 3ms/step
Epoch 74/100
6/6 - 0s - loss: 5617.6133 - val_loss: 5499.5791 - 21ms/epoch - 4ms/step
Epoch 75/100
6/6 - 0s - loss: 5510.1392 - val_loss: 5426.1143 - 21ms/epoch - 4ms/step
Epoch 76/100
6/6 - 0s - loss: 5570.2881 - val_loss: 5354.3345 - 21ms/epoch - 4ms/step
Epoch 77/100
6/6 - 0s - loss: 5350.2866 - val_loss: 5278.6997 - 21ms/epoch - 4ms/step
Epoch 78/100
6/6 - 0s - loss: 5355.6577 - val_loss: 5203.9951 - 21ms/epoch - 4ms/step
Epoch 79/100
6/6 - 0s - loss: 5297.9756 - val_loss: 5127.9634 - 21ms/epoch - 3ms/step
Epoch 80/100
6/6 - 0s - loss: 5180.0483 - val_loss: 5050.3447 - 20ms/epoch - 3ms/step
Epoch 81/100
6/6 - 0s - loss: 5090.1357 - val_loss: 4971.6045 - 21ms/epoch - 3ms/step
Epoch 82/100
6/6 - 0s - loss: 5164.4390 - val_loss: 4894.9121 - 22ms/epoch - 4ms/step
Epoch 83/100
```

```
6/6 - 0s - loss: 4894.6182 - val_loss: 4814.1953 - 21ms/epoch - 4ms/step
Epoch 84/100
6/6 - 0s - loss: 4890.4429 - val_loss: 4734.3540 - 21ms/epoch - 4ms/step
Epoch 85/100
6/6 - 0s - loss: 4905.8848 - val_loss: 4654.4956 - 21ms/epoch - 4ms/step
Epoch 86/100
6/6 - 0s - loss: 4844.4067 - val_loss: 4574.1255 - 21ms/epoch - 3ms/step
Epoch 87/100
6/6 - 0s - loss: 4684.1406 - val_loss: 4492.3540 - 21ms/epoch - 3ms/step
Epoch 88/100
6/6 - 0s - loss: 4663.4800 - val_loss: 4409.6147 - 21ms/epoch - 4ms/step
Epoch 89/100
6/6 - 0s - loss: 4559.1958 - val_loss: 4326.4302 - 22ms/epoch - 4ms/step
Epoch 90/100
6/6 - 0s - loss: 4456.0454 - val_loss: 4242.9165 - 21ms/epoch - 4ms/step
Epoch 91/100
6/6 - 0s - loss: 4430.0552 - val_loss: 4159.1265 - 21ms/epoch - 4ms/step
Epoch 92/100
6/6 - 0s - loss: 4312.1738 - val_loss: 4075.7549 - 21ms/epoch - 3ms/step
Epoch 93/100
6/6 - 0s - loss: 4350.4619 - val_loss: 3991.5483 - 21ms/epoch - 3ms/step
Epoch 94/100
6/6 - 0s - loss: 4146.0176 - val_loss: 3907.2200 - 21ms/epoch - 3ms/step
Epoch 95/100
6/6 - 0s - loss: 4132.5640 - val_loss: 3822.6509 - 21ms/epoch - 4ms/step
Epoch 96/100
6/6 - 0s - loss: 4140.9741 - val_loss: 3737.9851 - 21ms/epoch - 4ms/step
Epoch 97/100
6/6 - 0s - loss: 4031.6101 - val_loss: 3652.7029 - 21ms/epoch - 3ms/step
Epoch 98/100
6/6 - 0s - loss: 3974.0295 - val_loss: 3570.1226 - 21ms/epoch - 4ms/step
Epoch 99/100
6/6 - 0s - loss: 3862.4404 - val_loss: 3484.5674 - 21ms/epoch - 3ms/step
Epoch 100/100
6/6 - 0s - loss: 3791.4417 - val_loss: 3399.6299 - 21ms/epoch - 3ms/step
<keras.callbacks.History object at 0x7f96fca9c760>
```

9.5 查看訓練過程

```
print(train_history.history)
print(train_history.history.keys())

import matplotlib.pyplot as plt
plt.cla()
plt.title('Train History')
plt.ylabel('loss')
plt.xlabel('Epoch')
plt.plot(train_history.history['loss'])
plt.plot(train_history.history['val_loss'])
#plt.show()
plt.savefig("images/cnn-regression-4.png", dpi=300)
```

```
{'loss': [8163.17431640625, 8130.103515625, 8115.6943359375, 8093.09521484375, 8088.0625, 8076
dict_keys(['loss', 'val_loss'])
```

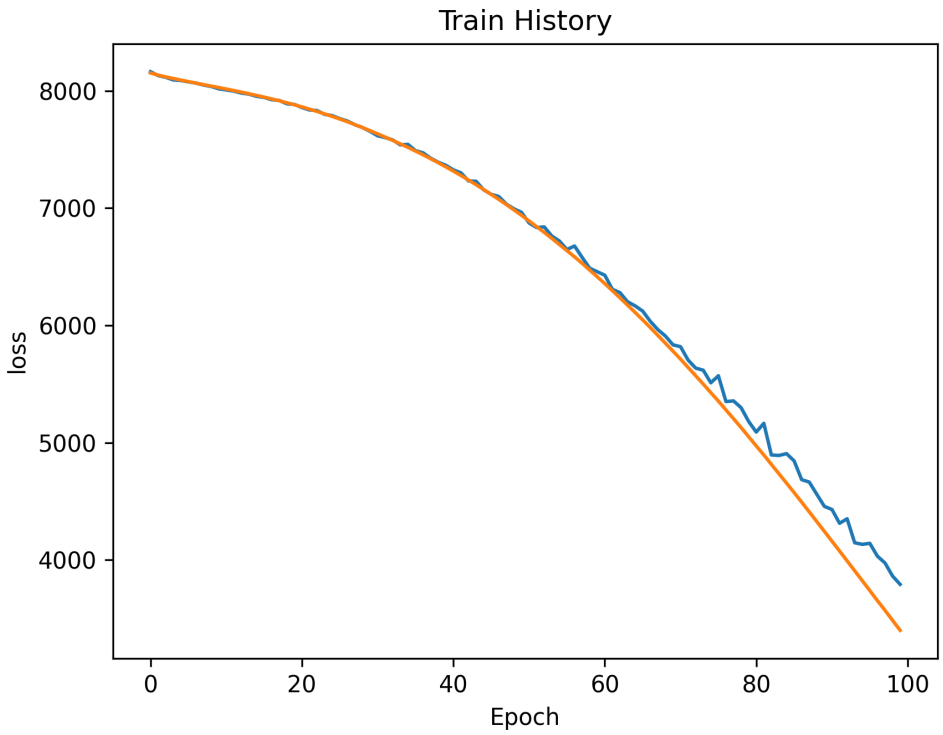


Figure 9: Caption

9.6 評估 model

```
# The evaluate() method - gets the loss statistics
score = model.evaluate(x_Test, y_Test, batch_size=200)
print(score)
```

```
3/3 [==============================] - 0s 897us/step - loss: 3321.0034
3321.00341796875
```

9.7 預測結果

```
# The predict() method - predict the outputs for the given inputs
print(y_Test[:5])
print(model.predict(x_Test[:5]))
```

```
[[79.18318665]
 [76.04753786]
 [95.92318028]
 [92.51954163]
 [81.48265046]]
1/1 [==============================] - 0s 14ms/step
[[13.401583]
 [14.596603]
 [46.890038]
 [36.758396]
 [21.054796]]
```

9.8 調整 model/參數

1. model 架構
2. loss function
3. optimizer
4. hyper parameters
5. #1

```
# A simple regression model
model = Sequential()
model.add(Dense(4, input_shape=(1,)))
model.add(Dense(8, input_shape=(1,)))
model.add(Dense(4, input_shape=(1,)))
model.add(Dense(1, input_shape=(1,)))
model.compile(loss='mean_squared_error', optimizer='rmsprop')
#mean_squared_logarithmic_error
#mean_absolute_percentage_error
train_history = model.fit(x=x_Train, y=y_Train,
                          validation_split=0.2,
                          epochs=100, batch_size=200,
```

```
                          verbose=0)
score = model.evaluate(x_Test, y_Test, batch_size=200)
print(score)
print(y_Test[:5])
print(model.predict(x_Test[:5]))
```

```
3/3 [==============================] - 0s 1ms/step - loss: 396.9111
396.91107177734375
[[79.18318665]
 [76.04753786]
 [95.92318028]
 [92.51954163]
 [81.48265046]]
1/1 [==============================] - 0s 42ms/step
[[ 39.586174]
 [ 42.389393]
 [118.141884]
 [ 94.375534]
 [ 57.538727]]
```

6. #2

```
model = Sequential()
model.add(Dense(4, input_shape=(1,)))
model.add(Dense(8, input_shape=(1,)))
model.add(Dense(16, input_shape=(1,)))
model.add(Dense(32, input_shape=(1,)))
model.add(Dense(16, input_shape=(1,)))
model.add(Dense(4, input_shape=(1,)))
model.add(Dense(1, input_shape=(1,)))
model.compile(loss='mse', optimizer='rmsprop')
train_history = model.fit(x=x_Train, y=y_Train,
                          validation_split=0.2,
                          epochs=100, batch_size=200,
                          verbose=0)
score = model.evaluate(x_Test, y_Test, batch_size=200)
print(score)
print(y_Test[:5])
print(model.predict(x_Test[:5]))
```

```
3/3 [==============================] - 0s 838us/step - loss: 10.2370
```

```
10.237029075622559
[[79.18318665]
 [76.04753786]
 [95.92318028]
 [92.51954163]
 [81.48265046]]
WARNING:tensorflow:5 out of the last 25 calls to <function Model.make_predict_function.<l
1/1 [==============================] - 0s 55ms/step
[[76.99616]
 [77.70199]
 [96.77555]
 [90.79147]
 [81.51641]]
```

10 課堂練習: Regression

現在來看看另一組較複雜的數據資料，請你試著自己建個模型來預測這些詭異的資料...

10.1 數據

```python
import numpy as np

# Seed the random number generator for reproducibility
np.random.seed(0)

x_data = np.linspace(-10, 10, num=2000)
y_data = 2.9 * np.sin(1.5 * x_data) + np.random.normal(size=len(x_data))

plt.scatter(x_data, y_data)
```

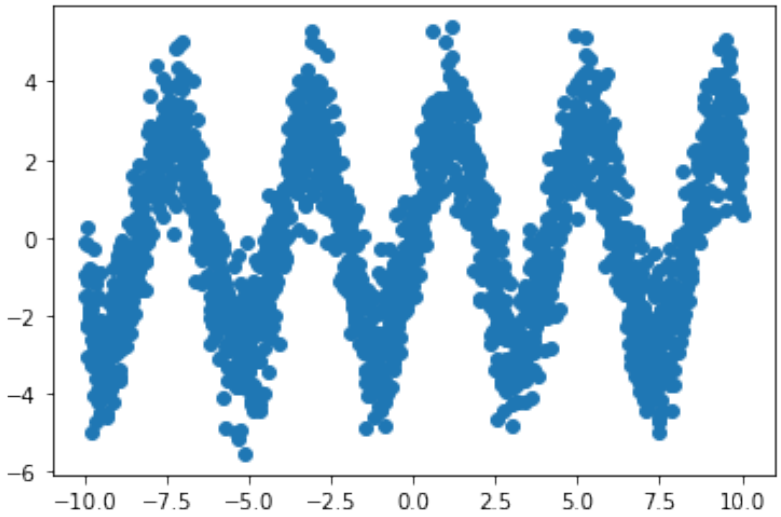


Figure 10: CNN 練習

11 DEMO 2: MNIST 資料集

11.1 MNIST

- MNIST 是機器學習領域中相當著名的資料集，號稱機器學習領域的「Hello world.」，其重要性不言可喻。

- MNIST 資料集由 0~9 的數字影像構成 (如圖1)，共計 60000 張訓練影像、10000 張測試影像。

- 一般的 MMIST 資料集的用法為：使用訓練影像進行學習，再利用學習後的模型預測能否正確分類測試影像。

準備資料是訓練模型的第一步，基礎資料可以是網上公開的資料集，也可以是自己的資料集。視覺、語音、語言等各種型別的資料在網上都能找到相應的資料集。

11.2 準備 MNIST 資料

MNIST 數據集來自美國國家標準與技術研究所, National Institute of Standards and Technology (NIST). 訓練集 (training set) 由來自 250 個不同人手寫的數字構成, 其中 50% 是高中學生, 50% 來自人口普查局 (the Census Bureau) 的工作人員. 測試集 (test set) 也是同樣比例的手寫數字數據。MNIST 數據集可在 `http://yann.lecun.com/exdb/mnist/` 獲取, 它包含了四個部分:

1. Training set images: train-images-idx3-ubyte.gz (9.9 MB, 解壓後 47 MB, 包含 60,000 個樣本)

2. Training set labels: train-labels-idx1-ubyte.gz (29 KB, 解壓後 60 KB, 包含 60,000 個標籤)

3. Test set images: t10k-images-idx3-ubyte.gz (1.6 MB, 解壓後 7.8 MB, 包含 10,000 個樣本)

4. Test set labels: t10k-labels-idx1-ubyte.gz (5KB, 解壓後 10 KB, 包含 10,000 個標籤)

雖然自己手動下載是個不錯的主意，可以讓你練習如何從網路抓檔案、解壓、設定這些檔案的下載路徑、然後再讀進來變成 numpy 的 array 或是 pandas 的 dataframe，再把這些資料餵給 tensorflow 模組去建模型..... 不過這對初學者來說著實是一件很麻煩的事，所以多數的套件都很貼心的幫你準備好了直接從網路抓下資料集的 function，像 tensorflow 的 load_data。

1. load data MNIST 資料集是一個適合拿來當作 TensotFlow 的練習素材，在 Tensorflow 的現有套件中，也已經有內建好的 MNIST 資料集，我們只要在安裝好 TensorFlow 的 Python 環境中執行以下程式碼，即可將 MNIST 資料成功讀取進來。

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()      (get-keras-mnist)
print(x_train.shape)
```

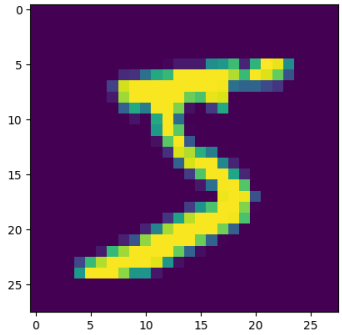
```
(60000, 28, 28)
```

在訓練模型之前，需要將樣本資料劃分為訓練集、測試集，有些情況下還會劃分為訓練集、測試集、驗證集。由上述程式第 get-keras-mnist 行可知，下載後的 MNIST 資料分成訓練資料 (training data) 與測試資料 (testing data)，其中 x 為圖片、y 為所對應數字。

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# ====================================
# 判斷資料形狀
print(x_train.shape)
print(x_test.shape)
# 第一個label的內容
print(y_train[0])
# 顯示影像內容
import matplotlib.pylab as plt
img = x_train[0]
plt.imshow(img)
plt.savefig("MNIST-Image.png")
```

```
(60000, 28, 28)
(10000, 28, 28)
5
```

由上述程式輸出結果可以看到載入的 x 為大小為 28*28 的圖片共 60000 張，每一筆 MNIST 資料的照片 (x) 由 784 個 pixels 組成 (28*28)，照片內容如圖11，訓練集的標籤 (y) 則為其對應的數字 (0～9)，此例為 5。



pxpx

Figure 11: MNIST 影像示例

x 的影像資料為灰階影像，每個像素的數值介於 0~255 之間，矩陣裡每一項的資料則是代表每個 pixel 顏色深淺的數值，如下圖12所示：

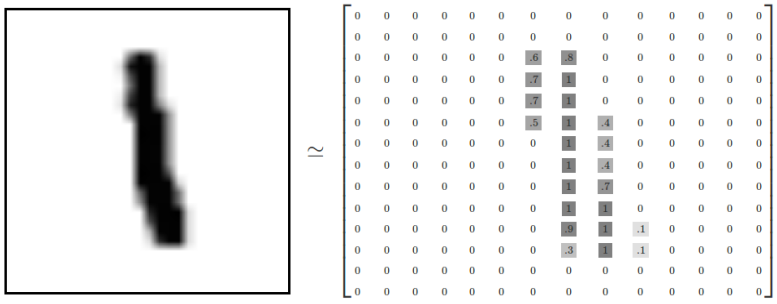


Figure 12: MNIST 資料矩陣

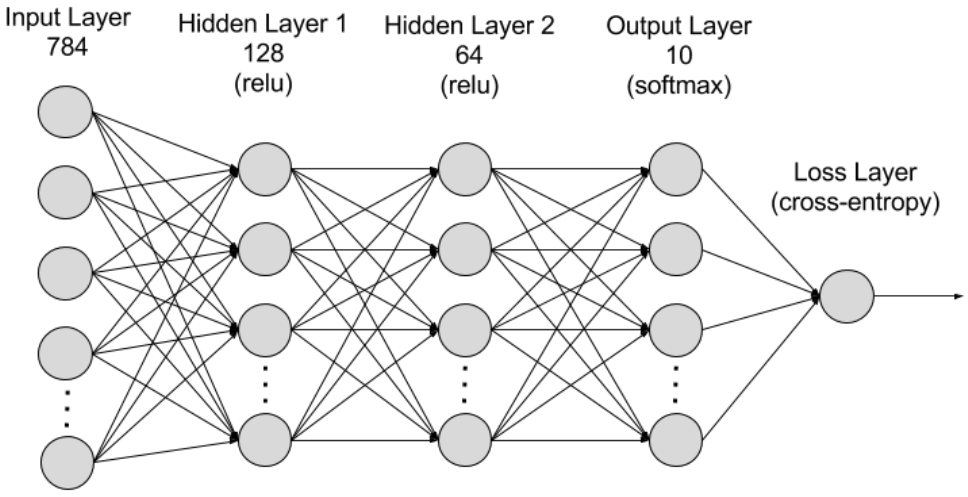
載入的 y 為所對應的數字 0~9,在這我們要運用 keras 中的 np_under_utils.to_under_categorical 將 y 轉成 one-hot 的形式，將他轉為一個 10 維的 vector，例如：我們所拿到的資料為 y=3，經過 np_utils.to_categorical，會轉換為 y=[0,0,0,1,0,0,0,0,0,0]。這部份的轉換程式碼如下：

```
from keras.datasets import mnist
from keras.utils import np_utils
import tensorflow as tf

mnist = tf.keras.datasets.mnist
```

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# ====================================
# 將圖片轉換為一個60000*784的向量, 並且標準化
x_train = x_train.reshape(60000, 784).astype('float32')
x_test = x_test.reshape(10000, 784).astype('float32')
x_train = x_train/255
x_test = x_test/255
# 將y轉換成one-hot encoding
y_train = np_utils.to_categorical(y_train, 10)
y_test = np_utils.to_categorical(y_test, 10)
# 回傳處理完的資料
print(y_train[0])

import numpy as np
np.set_printoptions(precision=2)
#print(x_train[0])
```

11.3 MNIST 的推論處理

如圖13所示，MNIST 的推論神經網路最前端的輸入層有 784 $(28 * 28 = 784)$ 個神經元，最後的輸出端有 10 個神經元 (0 9 個數字)，至於中間的隱藏層有兩個，第 1 個隱藏層有 50 個神經元，第 2 層有 100 個。此處的 50、100 可以設定為任意數（如，也可以是 128、64)。



pxpx

Figure 13: MNIST-NeuralNet

11.4 MNIST 資料集: 以 DNN Sequential 模型為例

此處以最簡單的 NN (Neural Network) 作為範例。以 Keras 的核心為模型，應用最常使用 Sequential 模型。藉由.add() 我們可以一層一層的將神經網路疊起。在每一層之中我們只需要簡單的設定每層的大小 (units) 與激勵函數 (activation function)。需要特別記得的是：第一層要記得寫輸入的向量大小、最後一層的 units 要等於輸出的向量大小。在這邊我們最後一層使用的激活函數 (activation function) 為 softmax。

1. Import Library

```
from keras.datasets import mnist
from keras.utils import np_utils
import numpy as np
np.random.seed(10)
```

2. 資料預處理

```
(x_Train, y_Train), (x_Test, y_Test) = mnist.load_data()
import matplotlib.pyplot as plt

##print(x_Train[1].shape)
#print(x_Train[1])
#plt.imshow(x_Train[1])
##y_Train[1]

x_Train=x_Train.reshape(x_Train.shape[0],28,28,1).astype('float32')
x_Test=x_Test.reshape(x_Test.shape[0],28,28,1).astype('float32')
#x_Train4D[1].shape
#print(x_Train4D[1])

x_Train = x_Train / 255
x_Test = x_Test / 255

y_Train = np_utils.to_categorical(y_Train)
y_Test = np_utils.to_categorical(y_Test)
```

3. 建立模型 activation function 的選擇: Keras API reference / Layers API / Layer activation functions

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
```

```
model = Sequential()
#將模型疊起
model.add(Dense(input_dim=28*28,units=128,activation='relu'))
model.add(Dense(units=64,activation='relu'))
model.add(Dense(units=10,activation='softmax'))
# model.summary()
```

4. 訓練模型 Model training APIs

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',metrics=['accuracy'])

train_history=model.fit(x=x_Train,
                        y=y_Train,validation_split=0.2,
                        epochs=6, batch_size=300,verbose=2)
```

5. 查看訓練過程

```
import matplotlib.pyplot as plt
def show_train_history(train_acc,test_acc):
    plt.plot(train_history.history[train_acc])
    plt.plot(train_history.history[test_acc])
    plt.title('Train History')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()

show_train_history('accuracy','val_accuracy')
show_train_history('loss','val_loss')
```

6. 評估模型準確率

```
scores = model.evaluate(x_Test , y_Test, batch_size = 200)
scores[1]
```

7. 實際預測結果

```
prediction=model.predict_classes(x_Test)
prediction[:10]

import matplotlib.pyplot as plt
def plot_images_labels_prediction(images,labels,prediction,idx,num=10):
    fig = plt.gcf()
    fig.set_size_inches(12, 14)
```

```
    if num>25: num=25
    for i in range(0, num):
        ax=plt.subplot(5,5, 1+i)
        ax.imshow(images[idx], cmap='binary')

        ax.set_title("label=" +str(labels[idx])+
                     ",predict="+str(prediction[idx])
                     ,fontsize=10)

        ax.set_xticks([]);ax.set_yticks([])
        idx+=1
    plt.show()
plot_images_labels_prediction(x_Test,y_Test,prediction,idx=0) #要用到原始的值
```

8. confusion matrix

```
import pandas as pd
pd.crosstab(y_Test,prediction,
            rownames=['label'],colnames=['predict'])
```

```
# 載入資料
from keras.datasets import mnist
from keras.utils import np_utils

def load_data():
    # 載入minst的資料
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    # 將圖片轉換為一個60000*784的向量, 並且標準化
    x_train = x_train.reshape(x_train.shape[0], 28*28)
    x_test = x_test.reshape(x_test.shape[0], 28*28)
    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    x_train = x_train/255
    x_test = x_test/255
    # 將y轉換成one-hot encoding
    y_train = np_utils.to_categorical(y_train, 10)
    y_test = np_utils.to_categorical(y_test, 10)
    # 回傳處理完的資料
    return (x_train, y_train), (x_test, y_test)

import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense,Activation
```

```
from keras.optimizers import  Adam

def build_model():#建立模型
    model = Sequential()
    #將模型疊起
    model.add(Dense(input_dim=28*28,units=128,activation='relu'))
    model.add(Dense(units=64,activation='relu'))
    model.add(Dense(units=10,activation='softmax'))
    model.summary()
    return model

# 開始訓練模型，此處使用了Adam做為我們的優化器，loss function選用了
    categorical_crossentropy。
(x_train,y_train),(x_test,y_test)=load_data()
model = build_model()
#開始訓練模型
model.compile(loss='categorical_crossentropy',optimizer="adam",metrics=['accuracy'])
model.fit(x_train,y_train,batch_size=100,epochs=5)
#顯示訓練結果
score = model.evaluate(x_train,y_train)
print ('\nTrain Acc:', score[1])
score = model.evaluate(x_test,y_test)
print ('\nTest Acc:', score[1])

### 進行預測
prediction = model.predict_classes(x_Test4D_normalize)
print(prediction[:10])
plot_images_labels_prediction("CNN_MNist", x_Test, y_Test, prediction, idx=0)
import pandas as pd
p = pd.crosstab(y_Test, prediction, rownames=['label'], colnames=['predict'])
print(p)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 500)               392500
_________________________________________________________________
dense_2 (Dense)              (None, 500)               250500
_________________________________________________________________
dense_3 (Dense)              (None, 500)               250500
_________________________________________________________________
```

```
dense_4 (Dense)              (None, 10)                5010
=================================================================
Total params: 898,510
Trainable params: 898,510
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20

  100/60000 [..............................] - ETA: 2:55 - loss: 2.2917 - acc: 0.1300
  800/60000 [..............................] - ETA: 25s - loss: 1.6424 - ACM: 0.5362
.......
16300/60000 [=======>......................] - ETA: 4s - loss: 0.3752 - acc: 0.8898
17000/60000 [=======>......................] - ETA: 4s - loss: 0.3681 - acc: 0.8916
.......
50600/60000 [========================>.....] - ETA: 0s - loss: 0.2232 - acc: 0.9335
51300/60000 [========================>.....] - ETA: 0s - loss: 0.2220 - acc: 0.9338
.......
59700/60000 [============================>.] - ETA: 0s - loss: 0.2078 - acc: 0.9377
60000/60000 [==============================] - 5s 81us/step - loss: 0.2074 - acc: 0.9379
Epoch 2/20

  100/60000 [..............................] - ETA: 5s - loss: 0.0702 - acc: 0.9800
......
60000/60000 [==============================] - 5s 77us/step - loss: 0.0832 - acc: 0.9740
Epoch 3/20
......
Epoch 29/20

   32/60000 [..............................] - ETA: 1:10
 1440/60000 [..............................] - ETA: 3s
......
58496/60000 [============================>.] - ETA: 0s
60000/60000 [==============================] - 2s 34us/step

Train Acc: 0.9981666666666666

   32/10000 [..............................] - ETA: 0s
 1568/10000 [===>..........................] - ETA: 0s
 3104/10000 [========>.....................] - ETA: 0s
```

```
 4640/10000 [============>.................] - ETA: 0s
 6176/10000 [=================>............] - ETA: 0s
 7680/10000 [======================>.......] - ETA: 0s
 9184/10000 [===========================>...] - ETA: 0s
10000/10000 [==============================] - 0s 33us/step

Test Acc: 0.9823
```

12 個人作業二

12.1 背景

某醫學研究中心針對旗下醫院 800 名疑似患有「無定向喪心病狂間歇性全身機能失調症」的患者做了一份病徵研究，針對以下這些可能病徵進行程度檢驗

1. 抑鬱
2. 癲癇
3. 精神分裂
4. 輕挑驕傲
5. 沒大沒小
6. 有犯罪傾向
7. 月經前緊張 (男患者嚴重的話也有)
8. 有自殺傾向

這 800 份資料可以點選這裡下載，每筆資料有九個欄位，前八欄分別對應到上述八項病徵，最後一欄為 0/1，代表病患是否患有該病。

請你建立一個預測 MODEL，以利該中心將來遇到類似病情的患者時只要先針對這些特徵值進行檢驗，即可了解該病例是否為此病患者，並即時予以適當治療。

12.2 作業要求

- 嗯，基本上就是自由心證，你能交多少就交多少，你想只交一張圖也行，你要從頭交待你在做什麼、每一個步驟有啥意義、一共測了幾種 CASE、最後成果如何、你的心得.... 也行，看你的誠意啦-_-(這向來是最坑人的一句話)

- 我是這樣覺得啦... model 隨便疊一疊，精確度至少也不應該低於 0.8 吧... QQ

13 小組作業三

13.1 Resources

1. **如何讀取自己的資料**: Loading Custom Image Dataset for Deep Learning Models: Part 1
2. Typical steps for loading custom dataset for Deep Learning Models
   (a) Open the image file. The format of the file can be JPEG, PNG, BMP, etc.

   (b) Resize the image to match the input size for the Input layer of the Deep Learning model.

   (c) Convert the image pixels to float datatype.

   (d) Normalize the image to have pixel values scaled down between 0 and 1 from 0 to 255.

   (e) Image data for Deep Learning models should be either a numpy array or a tensor object.

3. Data augmentation
   - Kaggle **知識點：資料擴增方法**
   - **影像資料擴增** (Image Data Augmentation) **的原理與實作**
   - **深度學習領域的資料增強**