碩管一甲 **M09218001** 周彥廷

高等類神經網路 作業 **1**：脂肪肝預測

作答說明：

請以 **FattyLiver_training.csv** 訓練你的 **MLP** 模型，並繪製你的網路結構、記錄你的最佳學習率、權重修正演算法等，記得要避免模式過擬合。

接著，再以 **FattyLiver_testing.csv** 計算模型預測的混淆矩陣與預測正確率。

目錄：

0.載入 **Python** 所需套件

```
import numpy as np
import pandas as pd
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.utils import np_utils
from sklearn.preprocessing import StandardScaler,MinMaxScaler
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from keras import layers
from keras import losses
from keras.layers import LeakyReLU
from keras import regularizers
from sklearn.metrics import classification_report, confusion_matrix
```

第一部分：依照作業題目建立 MLP 模型針對 FattyLiver 資料集訓練

1.導入資料

```
# 讀入訓練資料集
training_dataset = pd.read_csv('C:/Users/MCUT/Desktop/FattyLiver_training.csv', encoding='big5')
train_x = training_dataset.iloc[:, 1:42].values
train_y = np.where(training_dataset['是否有脂肪肝']== 'NO',0,1)
```

training_dataset - DataFrame

Index	是否有脂肪肝	年齡	性別	BMI	收縮壓	舒張壓	脈搏	有菸喝酒檳榔	腰圍	白血球	紅血球	血色素
0	NO	2	0	23.07	98.35	67.1	88.06	0	73.26	6.04	4.57	8.88
1	NO	2	0	20.72	103.23	72.77	94.56	0	67.94	4.69	5.53	12.06
2	NO	2	0	20.24	115.18	59.49	71.69	0	69.84	6.56	4.46	13.21
3	NO	3	0	21.25	98.2	62.03	68.74	0	72.33	3.82	4.16	12.64
4	YES	3	1	25.16	109.37	64.61	89.36	2	85.59	8.4	4.91	15.2
5	NO	2	0	20.48	93.38	61.89	74.11	0	62.34	6.44	4.36	13.34
6	NO	2	0	22.06	105.58	65.43	96.94	0	78.74	6.74	4.42	12.59
7	NO	2	1	21.9	104.62	69.32	66.02	1	77.46	6.57	5.62	17
8	NO	2	0	19.46	100.21	64.19	77.18	0	63.35	7.25	4.7	12.44
9	YES	2	0	29.73	102.03	60.18	61.6	0	84.39	8.53	4.59	13.36
10	NO	3	0	26.46	126.9	80.91	77.66	0	74.2	4.55	4.48	9.44
11	NO	2	0	20.67	92.5	56.75	76.54	0	72.15	4.46	4.58	12.14

train_x	Array of float64	(1500, 41)
train_y	Array of int32	(1500,)
training_dataset	DataFrame	(1500, 42)

```
# 讀入測試資料集
test_dataset = pd.read_csv('C:/Users/MCUT/Desktop/FattyLiver_testing.csv', header = None, encoding='big5')
test_x = test_dataset.iloc[:, 1:42].values
test_y = np.where(test_dataset[0]=='NO',0,1)
```

test_dataset - DataFrame

Index	0	1	2	3	4	5	6	7	8	9	10	11
0	YES	2	1	25.15	121.78	71.63	88.15	0	88.04	5.94	5.35	16.38
1	YES	1	1	23.58	121.84	72.24	89.48	2	80.95	8.21	5.12	15.81
2	NO	3	1	18.85	125.19	81.89	112.93	2	79.45	7.41	5.25	16.86
3	YES	2	1	30.22	133.21	87.2	93.17	0	91.02	8.3	5.07	14.51
4	NO	2	1	22.89	123.32	78.81	79.29	0	78.82	5.73	5.07	14.66
5	YES	2	1	22.05	116.14	71.91	72.64	0	79.82	4.27	5.45	17.33
6	YES	1	1	24.61	118.87	67.94	62.96	1	85.43	7.39	5.7	16.55
7	NO	1	1	19.55	103.14	61.09	83.02	2	66.71	8.86	5.13	16.27
8	YES	2	1	35.1	123.32	75.02	75.46	2	106.27	8.99	5.42	16.81
9	NO	2	1	22.76	113.5	68	67.28	0	78.96	6.29	4.83	14.89
10	YES	1	1	29.39	117.5	77.76	78.21	0	96.12	7.52	6.66	12.75
11	NO	2	1	22.07	115.58	70.77	80.03	0	69.9	8.17	5.19	15.01

test_dataset	DataFrame	(377, 42)
test_x	Array of float64	(377, 41)
test_y	Array of int32	(377,)

2.探索數據

檢視讀入資料型態(觀察特徵)並確認是否有無遺漏值，

#檢查是否有缺失值

training_dataset.isnull().sum()

```
In [328]: training_dataset.isnull().sum()
Out[328]:
是否有脂肪肝        0
年齡          0
性別          0
BMI         0
收縮壓          0
舒張壓          0
脈搏          0
抽菸喝酒檳榔        0
腰圍          0
白血球          0
紅血球          0
血色素          0
血中紅血球百分比        0
紅血球平均容積        0
紅血球色素        0
```

#特徵欄位均無遺漏值

資料可視化

import seaborn as sns

import matplotlib.pyplot as plt

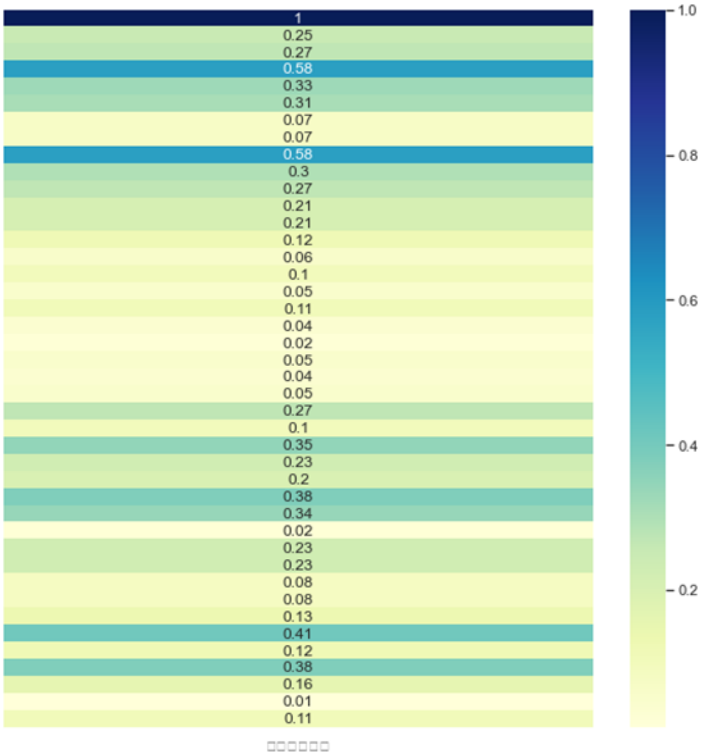
#繪製關聯矩陣(針對是否有脂肪肝)

correlation_matrix = training_dataset1.corr().round(2).loc[:,['是否有脂肪肝']].abs()

sns.heatmap(data=correlation_matrix, annot = True,cmap='YlGnBu')

#發現是否有脂肪肝指標與 BMI、腰圍與尿酸的關聯性最高。

是否有脂肪肝	1
年齡	0.25
性別	0.27
BMI	0.58
收縮壓	0.33
舒張壓	0.31
脈搏	0.07
抽菸喝酒檳榔	0.07
腰圍	0.58
白血球	0.3
紅血球	0.27
血色素	0.21
血中紅血球百分比	0.21
紅血球平均容積	0.12
紅血球色素	0.06



3.切分訓練集和測試集

題目已經切分好訓練資料集和測試資料集

4.數據預處理

#使用 StandardScaler 對資料進行預處理

print(f'預處理之前 x:{train_x[0]},y:{test_x[0]}')

ss_x = StandardScaler().fit(train_x)

X_train, X_test = ss_x.transform(train_x),ss_x.transform(test_x)

print(f'預處理之後 x:{X_train[0]},y:{X_test[0]}')

```
預處理之前 x:[2.0000e+00 0.0000e+00 2.3070e+01 9.8350e+01 6.7100e+01 8.8060
 0.0000e+00 7.3260e+01 6.0400e+00 4.5700e+00 8.8800e+00 3.0010e+01
 6.6200e+01 2.0080e+01 3.0570e+01 1.8700e+01 4.0632e+02 5.2190e+01
 3.4890e+01 8.6400e+00 4.9500e+00 7.6000e-01 1.6932e+02 1.2040e+01
 4.5050e+01 9.1530e+01 8.8000e-01 3.4690e+01 2.5820e+01 2.5200e+00
 2.8590e+01 7.2720e+01 4.4700e+00 4.4000e-01 2.3000e-01 4.8000e+00
 7.4300e+00 7.0940e+01 1.4085e+02 6.0000e+00 2.0000e+00],y:[2.0000e+00 1.0
 0.0000e+00 8.8040e+01 5.9400e+00 5.3500e+00 1.6380e+01 4.7090e+01
 8.8020e+01 3.1230e+01 3.5890e+01 1.1940e+01 2.5496e+02 6.3070e+01
 3.1420e+01 4.9200e+00 1.3400e+00 2.5000e-01 2.1475e+02 1.2060e+01
 1.6903e+02 8.8320e+01 1.0700e+00 4.0100e+01 2.3730e+01 7.4000e-01
 2.3850e+01 8.8550e+01 4.7800e+00 1.1300e+00 3.2000e-01 6.0900e+00
 7.2800e+00 5.8690e+01 2.0506e+02 7.0000e+00 0.0000e+00]
預處理之後 x:[ 0.04903168 -1.58003275 -0.28254469 -1.48079934 -0.70326852
 -0.64544401 -0.80099708 -0.48092454 -1.06901793 -3.72538223 -3.73385326
 -3.34694005 -3.60752438 -2.81150959  5.30742827  2.94001068 -0.51867983
 -0.14064074  2.00610448  1.19635385  0.63116537 -0.3077062   0.13320577
 -0.81906793  0.23675871 -0.90263412  0.30830908  0.02022232  0.62135465
  0.55711406  0.62653114 -1.21767015 -1.37433842 -0.85692073 -0.97407039
 -0.58094278  1.40122619 -0.90036908 -0.3578511   2.56205808],y:[ 0.049031
 -0.64544401  0.64479188 -0.54756966  0.40785744  0.99399264  0.59725511
  0.02536493  0.61279549  1.8236607  -1.12561049  0.00980272  0.91399188
 -0.63357374 -0.65205386 -0.76156261 -0.93915305  1.20117091  0.13931547
  0.49636972 -0.05951502  0.00885907  0.53686017 -0.07620188 -1.36328152
  0.15427083  1.55055983  0.1277418   0.49522715  0.13454566 -0.1204508
 -0.98722172  0.39408276  0.68033863  1.0793019  -0.59448165]
```

# 訓練標籤的獨熱編碼
encoding_train_y = np_utils.to_categorical(train_y)
# 測試標籤的獨熱編碼
encoding_test_y = np_utils.to_categorical(test_y)

5.模型建立

# 建立模式
model = Sequential()
model.add(Dense(10, input_dim=41, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(2, activation='softmax'))
#初步決定神經元個數與選擇激勵函數(relu)

```
Epoch 8/10
1500/1500 [==============================] - 1s 550us/step - loss: 0.5269 - acc: 0.7407
Epoch 9/10
1500/1500 [==============================] - 1s 553us/step - loss: 0.5310 - acc: 0.7373
Epoch 10/10
1500/1500 [==============================] - 1s 550us/step - loss: 0.5304 - acc: 0.7507
377/377 [==============================] - 2s 6ms/step

Accuracy: 72.15%
```

# 編譯模式
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

6.模型訓練

# 訓練模式
model.fit(X_train, encoding_train_y, epochs=10, batch_size=10)

7.模型評估

# 評估模式
scores = model.evaluate(test_x, encoding_test_y)
print("\nAccuracy: %.2f%%" % (scores[1]*100))

結論：訓練模式得到模型表現為 0.7507，評估模式得到 0.7215 正確率，第二部分會應用上課時類神經網路實務應用技巧使 MLP 效能提高。

第二部分：導入投影片所教技巧與法則改善模型效能

1.隱藏層活化函數的選擇

```
# 建立模式
model = Sequential()
model.add(Dense(10, input_dim=41, activation='sigmoid'))
model.add(Dense(20, activation='sigmoid'))
model.add(Dense(2, activation='softmax'))
# 編譯模式
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_560 (Dense)            (None, 10)                420
_________________________________________________________________
dense_561 (Dense)            (None, 20)                220
_________________________________________________________________
dense_562 (Dense)            (None, 2)                 42
=================================================================
Total params: 682
Trainable params: 682
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1500/1500 [==============================] - 7s 4ms/step - loss: 0.6848 - acc: 0.6000
Epoch 2/10
1500/1500 [==============================] - 1s 550us/step - loss: 0.6695 - acc: 0.6580
Epoch 3/10
1500/1500 [==============================] - 1s 560us/step - loss: 0.6548 - acc: 0.6640
Epoch 4/10
1500/1500 [==============================] - 1s 560us/step - loss: 0.6411 - acc: 0.6813
Epoch 5/10
1500/1500 [==============================] - 1s 560us/step - loss: 0.6184 - acc: 0.6987
Epoch 6/10
1500/1500 [==============================] - 1s 560us/step - loss: 0.6098 - acc: 0.6820
Epoch 7/10
1500/1500 [==============================] - 1s 552us/step - loss: 0.5912 - acc: 0.7080
Epoch 8/10
1500/1500 [==============================] - 1s 551us/step - loss: 0.5853 - acc: 0.7160
Epoch 9/10
1500/1500 [==============================] - 1s 545us/step - loss: 0.5723 - acc: 0.7260
Epoch 10/10
1500/1500 [==============================] - 1s 568us/step - loss: 0.5622 - acc: 0.7120
377/377 [==============================] - 2s 7ms/step

Accuracy: 74.01%
```

```
model = Sequential()
model.add(Dense(10, input_dim=41, activation='tanh'))
model.add(Dense(20, activation='tanh'))
```

model.add(Dense(2, activation='softmax'))

```
Epoch 1/10
1500/1500 [==============================] - 7s 4ms/step - loss: 0.6416 - acc: 0.6427
Epoch 2/10
1500/1500 [==============================] - 1s 558us/step - loss: 0.5952 - acc: 0.7040
Epoch 3/10
1500/1500 [==============================] - 1s 561us/step - loss: 0.6006 - acc: 0.6780
Epoch 4/10
1500/1500 [==============================] - 1s 569us/step - loss: 0.6005 - acc: 0.6927
Epoch 5/10
1500/1500 [==============================] - 1s 563us/step - loss: 0.5888 - acc: 0.7153
Epoch 6/10
1500/1500 [==============================] - 1s 571us/step - loss: 0.5873 - acc: 0.6780
Epoch 7/10
1500/1500 [==============================] - 1s 572us/step - loss: 0.5831 - acc: 0.7060
Epoch 8/10
1500/1500 [==============================] - 1s 579us/step - loss: 0.5761 - acc: 0.7187
Epoch 9/10
1500/1500 [==============================] - 1s 576us/step - loss: 0.5650 - acc: 0.7187
Epoch 10/10
1500/1500 [==============================] - 1s 569us/step - loss: 0.5657 - acc: 0.7153
377/377 [==============================] - 3s 7ms/step

Accuracy: 62.33%
```

使用 Sigmoid function 與 Tanh function 作為隱藏層活化函數在網路結構中會有 l 損失函數收斂的問題(紅色區塊觀察)與梯度消失問題，所以隱藏層活化函數選用 ReLU function 或是 Leaky ReLU function。

2.損失函數與輸出層活化函數的搭配

本案例是二元分類問題，因此損失函數選用 Binary Cross-entropy，輸出層的活化函數為 Softmax。

3.數據輸入批量設定

```
# 建立模式
model = Sequential()
model.add(Dense(10, input_dim=41, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(2, activation='softmax'))
# 編譯模式
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
# 訓練模式
model.fit(X_train, encoding_train_y, epochs=10, batch_size=1)
# 評估模式
scores = model.evaluate(test_x, encoding_test_y)
```

print("\nAccuracy: %.2f%%" % (scores[1]*100))

#Mini-batch Size=1 ：Stochastic Gradient Descent

model.fit(X_train, encoding_train_y, epochs=10, batch_size=1)

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_574 (Dense)            (None, 10)                420
_________________________________________________________________
dense_575 (Dense)            (None, 10)                110
_________________________________________________________________
dense_576 (Dense)            (None, 10)                110
_________________________________________________________________
dense_577 (Dense)            (None, 2)                 22
=================================================================
Total params: 662
Trainable params: 662
Non-trainable params: 0

_________________________________________________________________
Epoch 1/10
1500/1500 [==============================] - 15s 10ms/step - loss: 1.2885 - acc: 0.6467
Epoch 2/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.6581 - acc: 0.6707
Epoch 3/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.6330 - acc: 0.6567
Epoch 4/10
1500/1500 [==============================] - 8s 6ms/step - loss: 0.6059 - acc: 0.6960
Epoch 5/10
1500/1500 [==============================] - 8s 6ms/step - loss: 0.5680 - acc: 0.7367
Epoch 6/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.5415 - acc: 0.7480
Epoch 7/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.5282 - acc: 0.7507
Epoch 8/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.5233 - acc: 0.7540
Epoch 9/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.5128 - acc: 0.7533
Epoch 10/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.5001 - acc: 0.7693
377/377 [==============================] - 3s 7ms/step

Accuracy: 76.66%
```

#網路結構如前，Mini-batch Size = 8 (2 的三次方)

model.fit(X_train, encoding_train_y, epochs=10, batch_size=8)

實驗顯示如下圖：

```
Epoch 1/10
1500/1500 [==============================] - 8s 5ms/step - loss: 1.2962 - acc: 0.6727
Epoch 2/10
1500/1500 [==============================] - 1s 791us/step - loss: 0.8166 - acc: 0.6767
Epoch 3/10
1500/1500 [==============================] - 1s 780us/step - loss: 0.6307 - acc: 0.7173
Epoch 4/10
1500/1500 [==============================] - 1s 785us/step - loss: 0.6172 - acc: 0.7213
Epoch 5/10
1500/1500 [==============================] - 1s 782us/step - loss: 0.5998 - acc: 0.7273
Epoch 6/10
1500/1500 [==============================] - 1s 788us/step - loss: 0.5922 - acc: 0.7333
Epoch 7/10
1500/1500 [==============================] - 1s 795us/step - loss: 0.5323 - acc: 0.7620
Epoch 8/10
1500/1500 [==============================] - 1s 791us/step - loss: 0.5242 - acc: 0.7633
Epoch 9/10
1500/1500 [==============================] - 1s 799us/step - loss: 0.5174 - acc: 0.7627
Epoch 10/10
1500/1500 [==============================] - 1s 798us/step - loss: 0.4917 - acc: 0.7773
377/377 [==============================] - 3s 7ms/step

Accuracy: 78.25%
```

#經過實驗證實設定適當的 Mini-batch size 相較 SGD 損失函數收斂表現較好，且在電腦執行上 SGD 計算時間所需時間較長。

4.權重學習準則

選擇有考量 learning rate 的權重學習方法以 Adam 和 RMSprop 為主，進行學習率超參數調整實驗。

開始使用 Leaky ReLU function 並嘗試更改 Dense 層數與神經元個數，並使用 Dropout 優化模型權重更新

網路結構圖：

```
model = Sequential()
model.add(Dense(30, input_dim=41))
model.add(LeakyReLU(alpha=0.01))
model.add(Dropout(0.3))      #使用 Dropout
model.add(Dense(30))
model.add(LeakyReLU(alpha=0.01))
model.add(Dense(30))
model.add(LeakyReLU(alpha=0.01))
model.add(Dense(30))
model.add(LeakyReLU(alpha=0.01))
model.add(Dense(units=2, activation = 'softmax'))
# 編譯模式
ADAM    = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
epsilon=None, decay=0.0, amsgrad=False)
```

```
model.compile(loss='binary_crossentropy', optimizer = ADAM, metrics=['accuracy'])
model.summary()

# 訓練模式
model.fit(X_train, encoding_train_y, epochs=20, batch_size=10) #batch_size

# 評估模式
scores = model.evaluate(X_test, encoding_test_y)
print("\nAccuracy: %.2f%%" % (scores[1]*100))
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_714 (Dense)            (None, 30)                1260
_________________________________________________________________
leaky_re_lu_366 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
dropout_48 (Dropout)         (None, 30)                0
_________________________________________________________________
dense_715 (Dense)            (None, 30)                930
_________________________________________________________________
leaky_re_lu_367 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
dense_716 (Dense)            (None, 30)                930
_________________________________________________________________
leaky_re_lu_368 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
dense_717 (Dense)            (None, 30)                930
_________________________________________________________________
dense_718 (Dense)            (None, 2)                 62
=================================================================
Total params: 4,112
Trainable params: 4,112
Non-trainable params: 0
```

```
Epoch 14/20
1500/1500 [==============================] - 1s 816us/step - loss: 0.3819 - acc: 0.8360
Epoch 15/20
1500/1500 [==============================] - 1s 799us/step - loss: 0.3625 - acc: 0.8393
Epoch 16/20
1500/1500 [==============================] - 1s 801us/step - loss: 0.3520 - acc: 0.8513
Epoch 17/20
1500/1500 [==============================] - 1s 827us/step - loss: 0.3532 - acc: 0.8500
Epoch 18/20
1500/1500 [==============================] - 1s 784us/step - loss: 0.3540 - acc: 0.8467
Epoch 19/20
1500/1500 [==============================] - 1s 776us/step - loss: 0.3319 - acc: 0.8593
Epoch 20/20
1500/1500 [==============================] - 1s 782us/step - loss: 0.3472 - acc: 0.8400
377/377 [==============================] - 4s 10ms/step

Accuracy: 80.11%
```

發現模型的損失函數有下降和正確率都有提升，模型驗證正確率保持水平。

Optimizer 設定 RMSprop：

RMSprop = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
model.compile(loss='binary_crossentropy', optimizer = RMSprop,
metrics=['accuracy'])
model.summary()

```
Epoch 11/20
1500/1500 [==============================] - 2s 1ms/step - loss: 0.3930 - acc: 0.8333
Epoch 12/20
1500/1500 [==============================] - 2s 1ms/step - loss: 0.3940 - acc: 0.8380
Epoch 13/20
1500/1500 [==============================] - 1s 980us/step - loss: 0.3670 - acc: 0.8380
Epoch 14/20
1500/1500 [==============================] - 2s 1ms/step - loss: 0.3761 - acc: 0.8387
Epoch 15/20
1500/1500 [==============================] - 2s 1ms/step - loss: 0.3870 - acc: 0.8393
Epoch 16/20
1500/1500 [==============================] - 1s 984us/step - loss: 0.3729 - acc: 0.8360
Epoch 17/20
1500/1500 [==============================] - 1s 997us/step - loss: 0.3818 - acc: 0.8440
Epoch 18/20
1500/1500 [==============================] - 1s 984us/step - loss: 0.3671 - acc: 0.8507
Epoch 19/20
1500/1500 [==============================] - 1s 980us/step - loss: 0.3684 - acc: 0.8547
Epoch 20/20
1500/1500 [==============================] - 1s 980us/step - loss: 0.3685 - acc: 0.8467
377/377 [==============================] - 4s 11ms/step

Accuracy: 79.84%
```

結論：使用 Adam 和 RMSprop 使用效果差不多，學習率不宜設定太大(ex：0.1)，損失函數容易卡住使正確率無法提升，本案例設定 0.001 左右為宜。

設定 lr = 0.1 執行結果如下：

```
Epoch 15/20
1500/1500 [==============================] - 1s 937us/step - loss: 4.0443 - acc: 0.7473
Epoch 16/20
1500/1500 [==============================] - 1s 931us/step - loss: 3.7831 - acc: 0.7640
Epoch 17/20
1500/1500 [==============================] - 1s 930us/step - loss: 4.6608 - acc: 0.7087
Epoch 18/20
1500/1500 [==============================] - 1s 945us/step - loss: 4.5312 - acc: 0.7173
Epoch 19/20
1500/1500 [==============================] - 1s 957us/step - loss: 4.6488 - acc: 0.7100
Epoch 20/20
1500/1500 [==============================] - 1s 963us/step - loss: 4.7343 - acc: 0.7047
377/377 [==============================] - 4s 10ms/step

Accuracy: 69.23%
```

5.權重初始化方法

```
#使用 lecun 作為權重初始化方法
model = Sequential()
model.add(Dense(30, input_dim=41, kernel_initializer = 'lecun_uniform'))
                                                        #lecun_normal
model.add(LeakyReLU(alpha=0.01))
model.add(layers.BatchNormalization())     #作 BN 批量標準化
model.add(Dropout(0.3))
model.add(Dense(30))

model.add(LeakyReLU(alpha=0.01))
model.add(Dense(30))

model.add(LeakyReLU(alpha=0.01))
model.add(Dense(30))
model.add(Dense(units=2, activation = 'softmax'))
# 編譯模式
ADAM    = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
epsilon=None, decay=0.0, amsgrad=False)
model.compile(loss='binary_crossentropy', optimizer = ADAM, metrics=['accuracy'])
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_907 (Dense)            (None, 30)                1260
_________________________________________________________________
leaky_re_lu_493 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
batch_normalization_77 (Batc (None, 30)                120
_________________________________________________________________
dropout_81 (Dropout)         (None, 30)                0
_________________________________________________________________
dense_908 (Dense)            (None, 30)                930
_________________________________________________________________
leaky_re_lu_494 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
dense_909 (Dense)            (None, 30)                930
_________________________________________________________________
leaky_re_lu_495 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
dense_910 (Dense)            (None, 30)                930
_________________________________________________________________
leaky_re_lu_496 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
dense_911 (Dense)            (None, 2)                 62
=================================================================
Total params: 4,232
Trainable params: 4,172
Non-trainable params: 60
```

```
1500/1500 [==============================] - 3s 2ms/step - loss: 0.4299 - acc: 0.8053
Epoch 15/20
1500/1500 [==============================] - 2s 2ms/step - loss: 0.4313 - acc: 0.8067
Epoch 16/20
1500/1500 [==============================] - 2s 2ms/step - loss: 0.4209 - acc: 0.8073
Epoch 17/20
1500/1500 [==============================] - 2s 2ms/step - loss: 0.4185 - acc: 0.8133
Epoch 18/20
1500/1500 [==============================] - 2s 2ms/step - loss: 0.4109 - acc: 0.8213
Epoch 19/20
1500/1500 [==============================] - 2s 2ms/step - loss: 0.4064 - acc: 0.8247
Epoch 20/20
1500/1500 [==============================] - 2s 2ms/step - loss: 0.4233 - acc: 0.8040
377/377 [==============================] - 6s 16ms/step

Accuracy: 79.05%
```

```
#使用正則化(Regularizer)訓練神經網路
model = Sequential()
model.add(Dense(30, input_dim=41, kernel_regularizer=regularizers.l2(0.01),
                    bias_regularizer=regularizers.l1_l2(l1=0.01, l2=0.01),
                    activity_regularizer=regularizers.l1(0.01)))
model.add(LeakyReLU(alpha=0.01))
model.add(layers.BatchNormalization())    #作 BN 批量標準化
model.add(Dropout(0.3))
model.add(Dense(30))
model.add(LeakyReLU(alpha=0.01))
model.add(Dense(30))
model.add(LeakyReLU(alpha=0.01))
model.add(Dense(30))
model.add(Dense(units=2, activation = 'softmax'))
model.compile(loss='binary_crossentropy', optimizer = ADAM, metrics=['accuracy'])
```

```
Epoch 15/20
1500/1500 [==============================] - 3s 2ms/step - loss: 0.6247 - acc: 0.7893
Epoch 16/20
1500/1500 [==============================] - 3s 2ms/step - loss: 0.6003 - acc: 0.7960
Epoch 17/20
1500/1500 [==============================] - 3s 2ms/step - loss: 0.5836 - acc: 0.7967
Epoch 18/20
1500/1500 [==============================] - 2s 2ms/step - loss: 0.5998 - acc: 0.7973
Epoch 19/20
1500/1500 [==============================] - 2s 2ms/step - loss: 0.5869 - acc: 0.7893
Epoch 20/20
1500/1500 [==============================] - 2s 2ms/step - loss: 0.5797 - acc: 0.8027
377/377 [==============================] - 6s 16ms/step

Accuracy: 76.92%
```

6. Summary

運用技巧

1.隱藏層活化函數的選擇 (Leaky ReLU function)

2.損失函數與輸出層活化函數的搭配 (Binary Cross-entropy)

3.數據輸入批量設定 (batch_size)

4.權重學習準則 (Adam)

5.權重初始化方法 (lecun)

經過實驗與調整超參數得出最佳網路結構程式碼如下

```
model = Sequential()
model.add(Dense(30, input_dim=41, kernel_initializer = 'he_normal'))
model.add(LeakyReLU(alpha=0.01))
model.add(layers.BatchNormalization())      #作 BN 批量標準化
model.add(Dropout(0.3))                     # Dropout
model.add(Dense(30))
model.add(LeakyReLU(alpha=0.01))
model.add(layers.BatchNormalization())      #作 BN 批量標準化
model.add(Dropout(0.3))                     # Dropout
model.add(Dense(30))
model.add(LeakyReLU(alpha=0.01))
model.add(Dropout(0.3))                     # Dropout
model.add(Dense(30))
model.add(LeakyReLU(alpha=0.01))
model.add(Dense(units=2, activation = 'softmax'))
# 編譯模式
ADAM    = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
epsilon=None, decay=0.0, amsgrad=False)
model.compile(loss='binary_crossentropy', optimizer = ADAM, metrics=['accuracy'])
model.summary()
# 訓練模式
model.fit(X_train, encoding_train_y, epochs=20, batch_size=8)
# 評估模式
scores = model.evaluate(test_x, encoding_test_y)
print("\nAccuracy: %.2f%%" % (scores[1]*100))
```

繪製網路結構如下

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_933 (Dense)            (None, 30)                1260
_________________________________________________________________
leaky_re_lu_513 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
batch_normalization_84 (Batc (None, 30)                120
_________________________________________________________________
dropout_89 (Dropout)         (None, 30)                0
_________________________________________________________________
dense_934 (Dense)            (None, 30)                930
_________________________________________________________________
leaky_re_lu_514 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
batch_normalization_85 (Batc (None, 30)                120
_________________________________________________________________
dropout_90 (Dropout)         (None, 30)                0
_________________________________________________________________
dense_935 (Dense)            (None, 30)                930
_________________________________________________________________
leaky_re_lu_515 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
dropout_91 (Dropout)         (None, 30)                0
_________________________________________________________________
dense_936 (Dense)            (None, 30)                930
_________________________________________________________________
leaky_re_lu_516 (LeakyReLU)  (None, 30)                0
_________________________________________________________________
dense_937 (Dense)            (None, 2)                 62
=================================================================
Total params: 4,352
Trainable params: 4,232
Non-trainable params: 120
```

權重學習使用 Adam，學習率(lr=0.001)，隱藏層激勵函數使用 LeakyReLU 並在輸入層使用拋棄法(Dropout)

7.計算模型預測的混淆矩陣與預測正確率

		Predicted class	
		P	*N*
Actual Class	*P*	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

$$
\begin{aligned}
precision &= \frac{TP}{TP+FP} \\
recall &= \frac{TP}{TP+FN} \\
F1 &= \frac{2 \times precision \times recall}{precision + recall} \\
accuracy &= \frac{TP+TN}{TP+FN+TN+FP} \\
specificity &= \frac{TN}{TN+FP}
\end{aligned}
$$

pred_model = model.predict(X_test)

pred = np.argmax(pred_model, axis=1)

pred_1 = np_utils.to_categorical(pred)

print(confusion_matrix(encoding_test_y.argmax(axis=1), pred_model.argmax(axis=1)))

```
In [506]: print(confusion_matrix(
[[136  19]
 [ 57 165]]
```

print(classification_report(encoding_test_y, pred_1))

```
              precision    recall  f1-score   support

           0       0.70      0.88      0.78       155
           1       0.90      0.74      0.81       222

   micro avg       0.80      0.80      0.80       377
   macro avg       0.80      0.81      0.80       377
weighted avg       0.82      0.80      0.80       377
 samples avg       0.80      0.80      0.80       377
```

結論：透過課程所學技巧配合反覆實驗得到預測準確率 80%結果。