機器學習 HW11

姓名:林士恩

學號:B043011031

系級:108 電機甲

一、 原始程式碼:

LSTM:

```
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Embedding
from keras.layers import LSTM
from keras.datasets import imdb

max_features = 20000
maxlen = 80          # cut texts after this number of words (among top max_features most common words)
batch_size = 32 * 2      #一次訓練多少data

print('Loading data...')

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words = max_features) #考慮前20000個常用字元，將其他的非常用字編碼為oov_char
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')


#print(x_train)                #非narray object 其中包含list object
#print(y_train.shape)


print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen = maxlen)      #超過maxlen的就截掉, 少於的就補0
x_test = sequence.pad_sequences(x_test, maxlen = maxlen)
print('x_train shape:', x_train.shape)                          #換成25000x80的matrix
print('x_test shape:', x_test.shape)


print('Build model...')
model = Sequential()
model.add(Embedding(max_features, 128))                         #生成20000x128的矩陣來encoding, 而在訓練的過程當中更新這個矩陣的值
model.add(LSTM(128, dropout=0.2, recurrent_dropout = 0.2))      #輸出為128units(看成一層的神經元數量)
model.add(Dense(1, activation='sigmoid'))


# try using different optimizers and different optimizer configs
model.compile(loss = 'binary_crossentropy',
              optimizer = 'adam',
              metrics = ['accuracy'])

```

```
print('Train...')
model.fit(x_train, y_train,
          batch_size = batch_size,
          epochs = 8, verbose = 1,
          validation_data = (x_test, y_test))
score, acc = model.evaluate(x_test, y_test,
                            batch_size = batch_size)
print('Test score:', score)
print('Test accuracy:', acc)
```

```
25000/25000 [==============================] - 113s 5ms/step -
loss: 0.0570 - acc: 0.9807 - val_loss: 0.8034 - val_acc: 0.8149
25000/25000 [==============================] - 24s 977us/step
Test score: 0.803439349937
Test accuracy: 0.814879999962
```

(data 太少而 overfitting)

CNN:

```
from __future__ import print_function

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalMaxPooling1D
from keras.datasets import imdb

# set parameters:
max_features = 5000        #常用字彙改成5000個
maxlen = 400               #字串長度改成400
batch_size = 32
embedding_dims = 50
filters = 250
kernel_size = 3
hidden_dims = 250
epochs = 2

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)

print('Build model...')
model = Sequential()

# we start off with an efficient embedding layer which maps
# our vocab indices into embedding_dims dimensions
model.add(Embedding(max_features,                  # max_features = input_dim(字彙表大小)而非data總量
                    embedding_dims,                # output vector dim of the embedding layer(編碼的維度)
                    input_length = maxlen))        # the dim of the input to this layer
model.add(Dropout(0.2))

# we add a Convolution1D, which will learn filters
# word group filters of size filter_length:
model.add(Conv1D(filters,                          # filter輸出的張數(跟2D一樣)
                 kernel_size,                       # filter_length
                 padding='valid',                  # 不做zero padding
                 activation='relu',
                 strides=1))                       #convolution走的步長
# we use max pooling:
model.add(GlobalMaxPooling1D())                    #對於時間上的data做maxpooling, 一個

# We add a vanilla hidden layer:
#dim of the input must equal to the dim of the length of the output of the GlobalMaxPooling1D layer
model.add(Dense(hidden_dims))
model.add(Dropout(0.2))
model.add(Activation('relu'))

# We project onto a single unit output layer, and squash it with a sigmoid:
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs, verboise = 1,
          validation_data=(x_test, y_test))
```

```
25000/25000 [==============================] - 45s 2ms/step -
loss: 0.0448 - acc: 0.9839 - val_loss: 0.4497 - val_acc: 0.881
Epoch 8/8
25000/25000 [==============================] - 45s 2ms/step -
loss: 0.0357 - acc: 0.9867 - val_loss: 0.4812 - val_acc: 0.880
```

CNN + LSTM:

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/4
25000/25000 [==============================] - 89s 4ms/step -
loss: 0.3866 - acc: 0.8193 - val_loss: 0.3440 - val_acc: 0.8475
Epoch 2/4
25000/25000 [==============================] - 84s 3ms/step -
loss: 0.1986 - acc: 0.9242 - val_loss: 0.3425 - val_acc: 0.8580
Epoch 3/4
25000/25000 [==============================] - 89s 4ms/step -
loss: 0.0954 - acc: 0.9666 - val_loss: 0.4078 - val_acc: 0.8439
Epoch 4/4
25000/25000 [==============================] - 96s 4ms/step -
loss: 0.0422 - acc: 0.9863 - val_loss: 0.5677 - val_acc: 0.8423
25000/25000 [==============================] - 13s 525us/step
Test score: 0.567650891674
Test accuracy: 0.842319994521
```

雙向 LSTM:

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/4
25000/25000 [==============================] - 413s 17ms/step -
loss: 0.4125 - acc: 0.8102 - val_loss: 0.3474 - val_acc: 0.8464
Epoch 2/4
25000/25000 [==============================] - 410s 16ms/step -
loss: 0.2284 - acc: 0.9127 - val_loss: 0.3805 - val_acc: 0.8468
Epoch 3/4
25000/25000 [==============================] - 419s 17ms/step -
loss: 0.1297 - acc: 0.9529 - val_loss: 0.4743 - val_acc: 0.8373
Epoch 4/4
25000/25000 [==============================] - 417s 17ms/step -
loss: 0.0724 - acc: 0.9748 - val_loss: 0.6019 - val_acc: 0.8327
```

二、 程式碼解釋與討論:

LSTM:

這次的 LSTM 用來以影評分析來判斷情緒(以機率表示)，所以是實現 sequence to one(many to one)的架構，之所以用 LSTM 而不是傳統的 RNN 架構，在於 LSTM 能有效的解決梯度爆炸和梯度消失的問題，梯度爆炸的解決方法就是使權重矩陣限制在某個範圍內，而梯度消失就利用 Back Propagation 中 ct -> ct-1 的方向使 gradient 能順利傳下去(ResNet 高速公路的概念)，所以使得 LSTM 比起傳統 RNN 架構能有更好的效能，此外 GRU 為 LSTM 的進階延伸，

RNN 較 CNN 適合處理具有時間關聯的資訊，例如影片、文本、聲音，而 CNN 較擅長處理影像，處理具有空間中位置的影像較適合。

實現 multiple layer 如下:

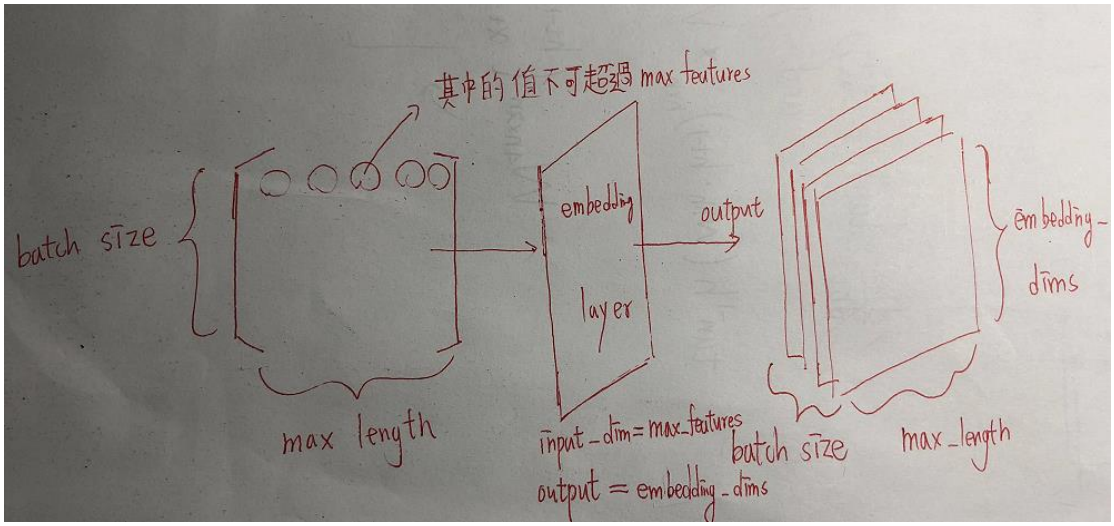
```
model = Sequential()
model.add(Embedding(max_features, 128))                    #生成20000x128的矩陣來encoding, 而在訓練的過程當中更新這個矩陣的值
model.add(LSTM(128, dropout=0.2, recurrent_dropout = 0.2, return_sequences=True))     #輸出為128units(看成一層的神經元數量)
model.add(LSTM(128, dropout=0.2, recurrent_dropout = 0.2))     #輸出為128units(看成一層的神經元數量)
model.add(Dense(1, activation='sigmoid'))
```

```
Epoch 8/8
25000/25000 [==============================] - 212s 8ms/step -
loss: 0.0439 - acc: 0.9851 - val_loss: 0.7881 - val_acc: 0.8094
25000/25000 [==============================] - 47s 2ms/step
Test score: 0.788063946095
Test accuracy: 0.809399999962
```

CNN:

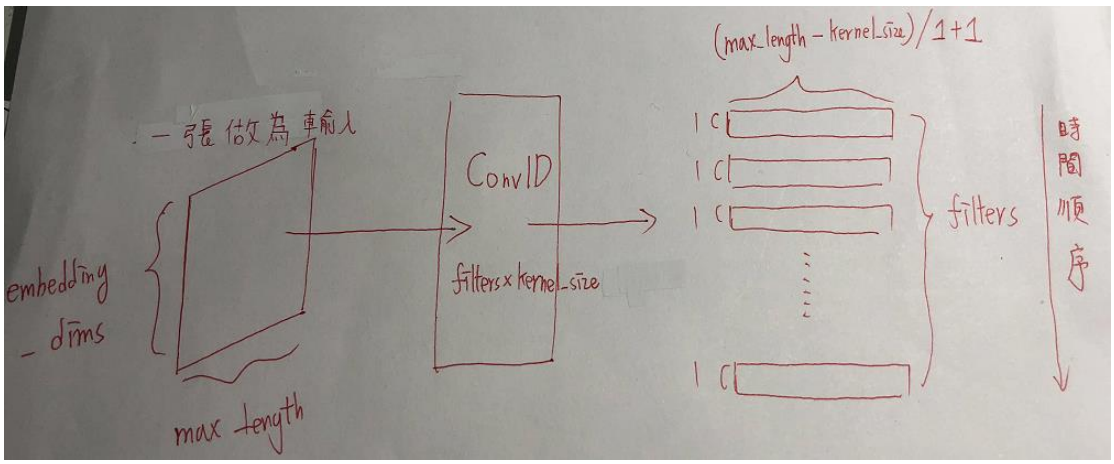
```
model.add(Embedding(max_features,                  # max_features = input_dim(字彙表大小)而非data總量
                    embedding_dims,                # output vector dim of the embedding layer(編碼的維度)
                    input_length = maxlen))        # the dim of the input to this layer
```

示意圖如下:




```
model.add(Conv1D(filters,                          # filter輸出的張數(跟2D一樣)
                 kernel_size,                       # filter_length
                 padding='valid',                  # 不做zero padding
                 activation='relu',
                 strides=1))                       #convolution走的步長
```

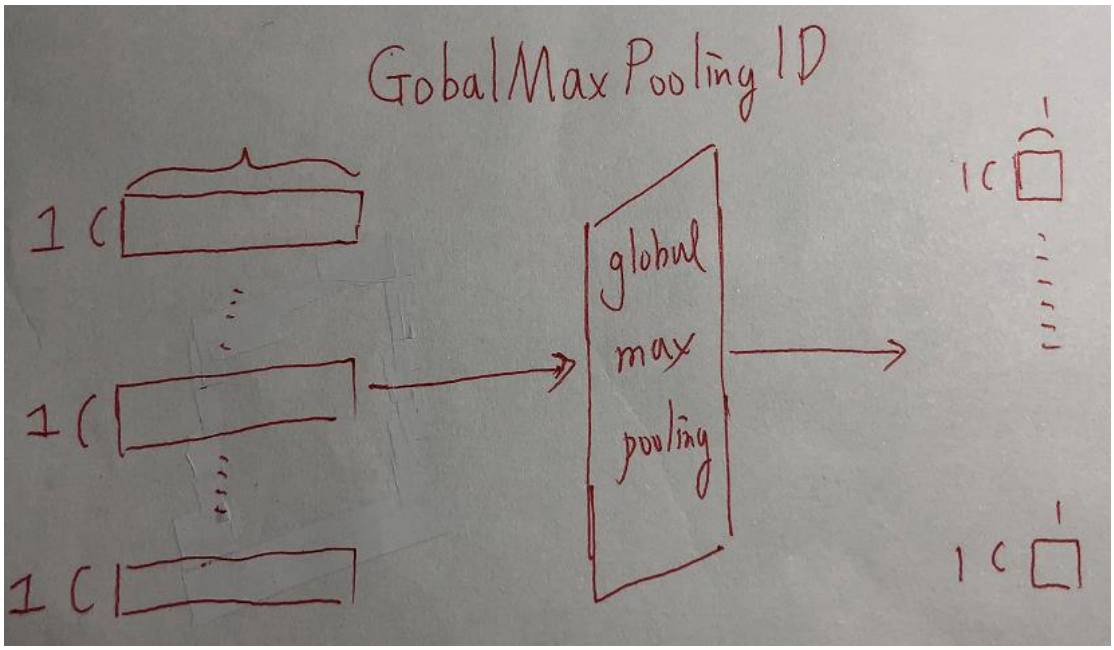
示意圖如下:



須注意的是 Conv1D 其實可以看成在做圖片的 Conv，只是做一次 Conv 只有做 1 個維度(1 個時間 t)，接下來進行第 2 次 Conv 是做 t+1 的時間點，以此類推。

```
55 model.add(GlobalMaxPooling1D())                    #對於時間上的data做maxpooling,
```

示意圖如下:



此 MaxPooling 大小為 input_length，就是把值壓縮成 1x1 的大小。

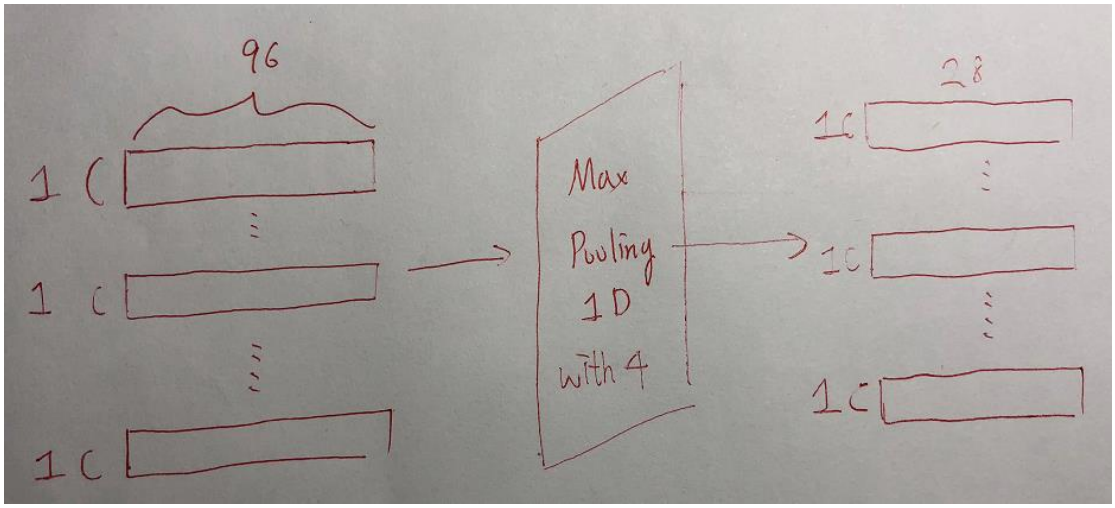
CNN 擅長處理靜態的資料(資料沒有時間前後連貫的關係)，所以較熱門的應用都在於影像處理方面。CNN 與 RNN 的很大差別之一為 CNN 容易實現”Deep”layers，而 RNN 如果深層的話會有梯度爆炸的問題。

CNN + LSTM:

```
model.add(Embedding(max_features, embedding_size, input_length=maxlen))        #embedding layer
model.add(Dropout(0.25))
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1))

model.add(MaxPooling1D(pool_size = pool_size))                                  #MaxPooling -> 產生 96 / 4 = 28x1的vector(某時刻的語句)
model.add(LSTM(lstm_output_size))                                               #輸出成128維的output
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

是意圖如下:



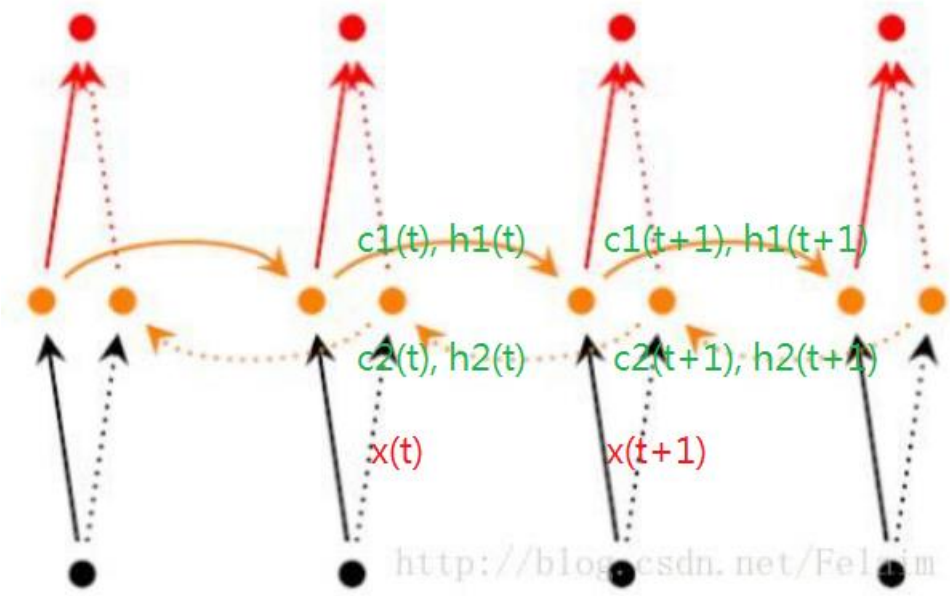
需注意的是在進行 CNN 的 trainning 時，經過 Conv1D 後就已經把一個 Batch_size 的訓練資料 Conv 成一個個的 row vector，所以經過 MaxPooling1D 後可以直接跟 LSTM 連接，不需要再做額外的維度轉換。

這次的測試資料是文本而非圖片，所以較沒辦法凸顯 CNN+LSTM 的優點。

CNN+LSTM 的優點在於，CNN 擅長於靜態的特徵抓取，所以先利用 CNN 進行影像的特徵訓練，再利用 RNN 擅長的動態訓練，以 CNN 傳過來的特徵當成 input，生成語句。可以應用於:圖片問答、圖片描述(標示)。

雙向 LSTM:

雙向 LSTM 的優點在於某時間點訓練時，除了可以利用歷史訊息，也可以利用未來的訊息一起訓練，增加更多的資訊量，需注意的是，雙向 RNN 其實就是 2 個相反方向的 RNN 組成，而這兩個 RNN 彼此並不共用同一個 hidden layer，但是 output layer 需要兩者共同組成輸出，是意圖如下:




```
model.add(Embedding(max_features, 128, input_length = maxlen))      #一個input會輸出為128 x maxlen的矩陣
model.add(Bidirectional(LSTM(64)))                                  #雙向LSTM層, 輸出為64 units(看成一層的神經元數量)
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

進行 LSTM 時一樣需要 embedding 層，並且加入雙向的 hidden layer(兩個 LSTM)，輸出成 64 units 的，再加入 sigmoid 中判斷機率。

Bidirectional LSTM 較傳統的 RNN、LSTM 有更強大的效能，並應用在處理自然語言領域較突出，因為常常判斷一句話，要回答時，往往不只需要前面歷史的資訊，如果加上未來的資訊的話，這樣就等同於可以得到更多的文本特徵來訓練模型。

並且 RNN 如果堆多層一點(垂直方向)，也能達到抓取更多特徵的功用。