Bonus-C：RNN 對 IMDB 的煉金實驗

實驗 0：RNN 的煉金術配方

實作了這麼多種的 RNN, 我們不禁想問到底多少神經元, 以及嵌入層的維度要設定多大, 才是最適當的, 因此筆者便進行了以下的測試實驗, 讓我們一起來探討這個深奧的問題。

實驗方法

❶ 使用 4-8.0.py 的架構來進行測試, 即資料處理方式和神經網路結構都一樣, 不過會移除 Dropout 層。

❷ 固定嵌入層維度為 16, 並讓 LSTM 的神經元數量從 4 個開始測試, 每次增加 4 個神經元, 依序遞增到 32 個 (4、8…32)。

❸ 每次的神經元數量測試會重複進行 10 遍, 每一遍有 10 個訓練週期 (Epochs), 訓練一遍後會取訓練週期中的最大驗證準確率, 10 遍後將每一遍的最大準確率平均起來, 當成此神經元數量的得分。

❹ 取最高分的神經元數量, 並固定此數量對嵌入層維度大小進行測試, 測試方法同步驟 2、3, 只是將神經元數量換成嵌入層維度。

實驗結果

經過實驗後我們可以得到不同神經元數量的得分表：

不同神經元數量在相同嵌入層維度下的得分

LSTM 神經元數量	得分
4	0.8496
8	0.8684
12	0.8790
16	0.8783
20	**0.8819**
24	0.8787
28	0.8791
32	0.8811

從上表可以看到得分隨著神經元數量上升而一起提升, 不過當數量超過 20 個以後便沒有明顯的效果了, 因此我們會固定神經元數量為 20, 並再接著對嵌入層維度進行測試, 如右為實驗結果：

不同嵌入層維度在相同神經元數量下的得分

嵌入層維度大小	得分
4	0.8538
8	0.8694
12	0.8785
16	0.8818
20	0.8808
24	**0.8830**
28	0.8784
32	0.8770

最後可以得知嵌入層大約是 24 維時會有最佳的結果。

實驗討論

以上的結果僅能說明 LSTM 處理 IMDB 分類問題時, 使用 20 個神經元和 24 維的嵌入層會有不錯的結果, 而無法代表所有的情況, 事實上實驗的結果差距也不大, 因此我們只能說 RNN 的效果會隨著神經元和嵌入層的維度增加而提升, 但並不是無限制的上漲, 而是有極限的, 所以這樣的實驗或許無法給出很精準的判斷, 但能讓我們大概知道要將參數設定成多少, 才能同時有足夠的效能, 又不會徒增運算的資源, 後續的實驗便會參考此數據來進行設計。

實驗 1：RNN 一定要搭配嵌入層嗎？

先前我們在跑 RNN 模型時都有用到嵌入層 (Embedding layer), 不過事實上, 這兩者並沒有一定要綁在一起。以下便會使用兩種資料格式來進行測試, 一個是數字序列, 即文字轉數字後的格式, 也就是 IMDB 載入的原始樣子, 另一個是 one-hot 編碼。最後會比較這兩者和使用嵌入層的效果。

數字序列在 LSTM 的測試

首先我們使用數字序列來測試, 在 4-4-2 節有說過, 這樣原始的格式會讓神經網路難以訓練, 不過正所謂眼見為憑, 測試過才知道真正的效果如何。

資料處理的方式與 4-5.0.py 一樣, 不過由於之後要將序列送入 LSTM, 所以還要將資料的 shape 轉為 (批次量大小, 時間長度, 特徵數), 由於我們的特徵就是一個數字, 所以特徵數就是 1。shape 的轉換可以使用本章工具模組 utilC.py 中的 seq_for_rnn()：

utilC.py　將數字序列轉為 RNN 可接受的 shape

```
def seq_for_rnn(seq):
    r_seq = seq.reshape(len(seq), -1, 1) ◄── 批次量大小不變　特徵數為 1
    return r_seq
```

BonusC.0.py　資料處理

```
…(略, 資料載入與處理方式同 4-5.0.py)

import utilC as u

train_data_seq = u.seq_for_rnn(train_data)
print(train_data_seq.shape)
```

```
(25000, 500, 1) ◄── 多了一軸
```

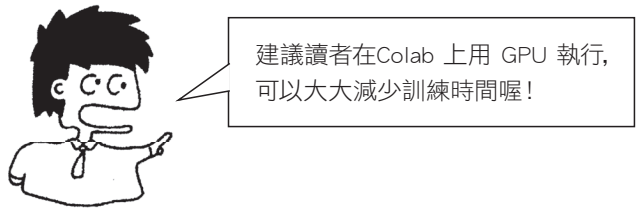
資料處理好後, 就能建立 LSTM 模型, 我們設定 LSTM 神經元數量為之前測試的最佳參數：20 個, 並讓它訓練：

BonusC.0.py(續)　建立模型並訓練

```
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

model = Sequential()
model.add(layers.InputLayer(input_shape=(maxlen, 1)))    ◄── 特徵數為 1
model.add(layers.LSTM(20))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()
```

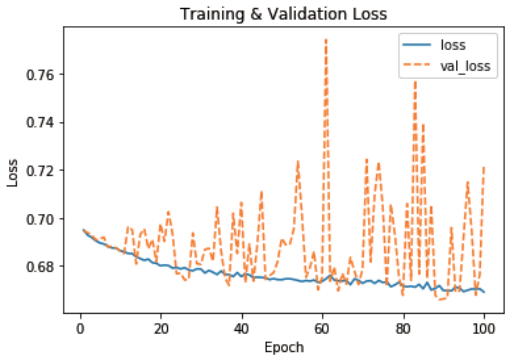
接下頁

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(train_data_seq, train_labels,
                    epochs=10,
                    batch_size=512,
                    validation_split=0.2)
```

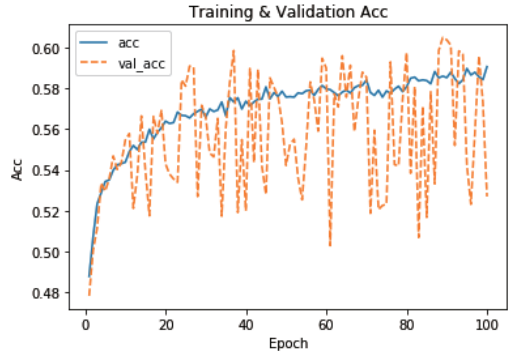


10 個訓練週期後, 可以看到不論訓練還是驗證準確率都只能到大概 54%, 不僅學習速度緩慢, 甚至可以說幾乎沒有學習能力, 即便嘗試將訓練週期提升到 100, 最好的驗證準確率也僅能到 60%。

訓練過程如下圖：



LSTM 處理數字序列的訓練和驗證損失



LSTM 處理數字序列的訓練和驗證準確率

以上結果便驗證了先前所說, 密集的數字序列資料會讓神經網路難以訓練, 因為資料被分布在又擠又亂的 1D 空間中。

one-hot 編碼在 LSTM 的測試

再來我們測試 one-hot 編碼的效果會如何, 事實上, 在 2016 年 Rie Johnson 和 Tong Zhang Rie 便提出了 oh-LSTM 的概念, 其中 oh 指的就是 one-hot。他們認為訓練 LSTM 文字模型時不一定需要嵌入層, 因為如果僅是把一個隨機的嵌入層作為模型的一部份一起訓練, 又沒有足夠的訓練資料, 這樣很難產生出轉換效果好的嵌入層, 除非嵌入層是預先訓練過的 (參見第 6 章), 不然乾脆直接使用 one-hot 編碼的文字資料來進行訓練。

以下我們便會將 IMDB 轉為 one-hot 編碼的文字資料, 並利用 LSTM 來進行訓練, 看看效果是否會超越有加入嵌入層的 LSTM, 首先載入 IMDB 資料集並進行序列對齊, 另外因為之後我們會用生成器 (generator) 來訓練, 這樣就無法用 fit() 中的 validation_split, 所以要先自行將資料分為訓練資料和驗證資料：

BonusC.1.py 載入 IMDB 資料集並進行序列對齊和資料切割

```
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences

num_words = 10000    ◀── 只處理常見的前 10000 個單字

(train_data, train_labels), (test_data, test_labels) = imdb.load_
data(num_words= num_words)

maxlen = 500         ◀── 設定序列長度為 500

train_data = pad_sequences(train_data, maxlen=maxlen,
truncating='post')   ◀── 只取前 500 個單字, 超過的截掉

val_data = train_data[-5000:]    ◀── 取後 5000 筆資料用作驗證資料, 與
                                     validation_split 設定為 0.2 是一樣的意思
train_data = train_data[:-5000] ◀── 取前 20000 筆資料用作訓練資料

val_labels = train_labels[-5000:]
train_labels = train_labels[:-5000]
```

接著使用本章工具模組 utilC.py 中的 seq2oh_generator() 來將資料轉成 one-hot 編碼生成器 (生成器的寫法請參見 4-4-3 節)：

BonusC.1.py(續)　將資料轉成 one-hot 編碼生成器

```
import utilC as u

batch_size = 50  ◄── 設定 generator 的批次量大小為 50

train_data_oh = u.seq2oh_generator(train_data, train_labels, batch_
size= batch_size, num_classes=num_words)
val_data_oh = u.seq2oh_generator(val_data, val_labels, batch_
size=batch_size, num_classes=num_words)
```

再來建立一個 LSTM 模型, 由於資料是 one-hot 編碼, 所以其輸入層的 shape 為 (序列長度, 詞彙量大小)：

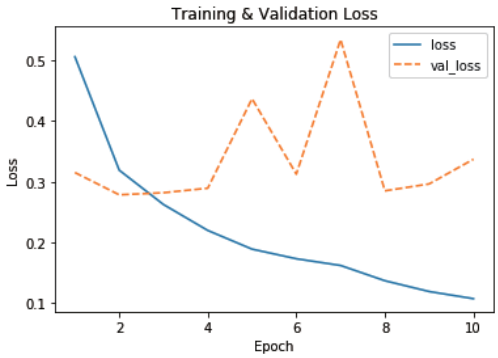
BonusC.1.py(續)　建立 oh-LSTM 模型

```
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

model = Sequential()
model.add(layers.InputLayer(input_shape=(maxlen, num_words)))
model.add(layers.LSTM(20))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()
```

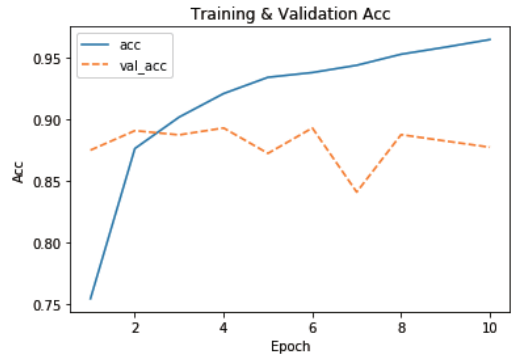
最後就能讓模型進行訓練：

BonusC.1.py(續)　訓練模型

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit_generator(train_data_oh,
                    steps_per_epoch=len(train_data)//batch_size,
                    validation_data=val_data_oh,
                    validation_steps=len(val_data)//batch_size,
                    epochs=10,
                    verbose=1)
```



LSTM 處理 one-hot 編碼的訓練和驗證損失



LSTM 處理 one-hot 編碼的訓練和驗證準確率

以上除了資料格式外，其餘參數皆比照先前有嵌入層的 LSTM 模型，從訓練過程中，我們可以看到最好的驗證準確率達到了 89%，這比有加入嵌入層的效果還要好。

LSTM 使用不同格式的比較

透過以上的實驗，最終我們可以歸納出右表：

格式	驗證準確率
20 維的嵌入層	0.88
原始序列	0.60
one-hot	**0.89**

實驗結果表明越稀疏的資料對神經網路的訓練越有幫助，然而也會帶來龐大的運算成本，如果想要使用稠密的資料，那就要有良好的壓縮 (整理) 方法。

實驗 2：自然語言處理的特徵工程

在前幾章中，神經網路輕輕鬆鬆就能在與影像處理的問題達到 90% 以上的準確率，然而到目前為止我們都沒有看到神經網路在自然語言處理的問題 (IMDB) 拿到什麼好成績，因此，接下來我們會將一些特徵工程加入 RNN，並進行實驗，看是否能幫助我們突破當前的門檻。

詞袋(Bag-of-Word)

詞袋 (Bag-of-Word) 簡稱 BOW, 將文字資料經過詞袋處理後能讓神經網路一次看到多個詞, 就像人類在閱讀時, 其實也不是一個字一個字在看, 而是會分區來看 (詳細說明參見 6-1 節)。以下我們就來實作 BOW 並結合 LSTM, 首先使用本章工具模組 utilC.py 中的 bow_generator() 將資料處理成 BOW 的樣子, 以下為此 generator 的程式：

utilC.py　將序列轉成 BOW 的生成器

```
def bow_generator(data, y, batch_size=200, num_words=10000, bag_size=10):
    i = 0
    while True:
        if i*batch_size+batch_size > len(data)-1:
            i = 0
        samples = np.zeros((batch_size, data.shape[1]//bag_size,
                            num_words))

        sequences = data[(i*batch_size):(i*batch_size+batch_size)]
        for j, sequence in enumerate(sequences):
            for k in range(data.shape[1]//bag_size):
                word = sequence[k*bag_size:k*bag_size+bag_size]
                samples[j, k, word] = 1.

        targets = y[(i*batch_size):(i*batch_size+batch_size)]
        i+=1

        yield samples, targets
```

詞袋大小

若這一批次會超過資料長度, 則將 i 歸零

建立要輸出的 array, 初始的內容全為 0

先取出一個 batch 的資料

取出一個 bag 的詞

依據出現的詞將輸出 array 的對應位置設定為 1

使用這個生成器時, 要指定詞袋大小和批次量大小, 這裡分別指定為 20 和 100：

BonusC.2.py 將資料處理成 bow 生成器

```
…(略, 資料處理方式同 BonusC.1.py)

import utilC as u

bag_size = 20     ◄── 設定詞袋大小為 20
batch_size = 100  ◄── 設定生成器批次量大小為 100

train_data_bow = u.bow_generator(train_data, train_labels, batch_
size=batch_size, bag_size=bag_size)
val_data_bow = u.bow_generator(val_data, val_labels, batch_size=batch_
size, bag_size=bag_size)
```

接下來建立 BOW-LSTM 模型：

BonusC.2.py(續) 建立 BOW-LSTM 模型

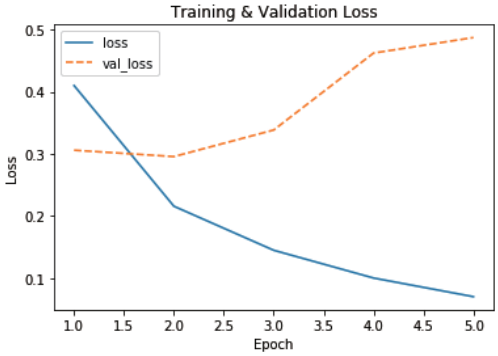
```
model = Sequential()
model.add(layers.InputLayer(input_shape=(maxlen//bag_size, num_words)))
model.add(layers.LSTM(20))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()
```

序列長度要除以詞袋大小

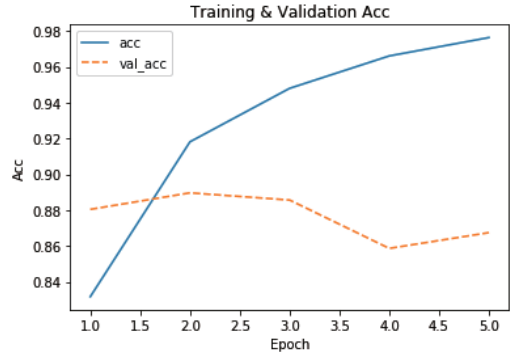
再來就能訓練模型了：

BonusC.2.py(續) 訓練模型

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit (train_data_bow,
                     steps_per_epoch=len(train_data)//batch_size,
                     validation_data=val_data_bow,
                     validation_steps=len(val_data)//batch_size,
                     epochs=5,
                     verbose=1)
```



LSTM 處理 BOW 的訓練和驗證損失



LSTM 處理 BOW 的訓練和驗證準確率

最好的驗證準確率大約到 88.9%, 比先前的嵌入層效果還要好一些, 接著我們使用和實驗 1 一樣的方法對詞袋大小進行測試, 看詞袋大小是否會影響神經網路的學習能力, 測試的大小分別為：10、20、50、100、250, 以下為測試結果：

詞袋大小	得分
10	0.888
20	0.889
50	0.890
100	0.891
250	**0.894**

結果顯示詞袋越大效果也越好, 事實上當詞袋大小等於序列長度時, 資料就和 multi-hot 編碼一模一樣了, 當然這樣也就失去 RNN 的意義了, 可見 BOW-LSTM 不是處理 IMDB 的最好方式, 因為效果不如直接使用 multi-hot 編碼。

N 元語法(N-gram)

N 元語法 (N-gram) 和 BOW 的概念很類似, 不一樣的是 N-gram 還會考慮 N 以下的其它詞語組合, 例如有一個句子為 "今天 天氣 真好", 如果使用 2 元語法, 就能得到以下資料："今天 天氣 真好 今天天氣 天氣真好", 如果是使用 3 元語法, 則能得到："今天 天氣 真好 今天天氣 天氣真好 今天天氣真好", 也就是 N 以下的所有連續組合, 這可以視為一種資料擴增法。

以下會將 N-gram 結合嵌入層和 LSTM, 首先自建一個函式 n_gram() 將資料進行擴增, 工作流程如下：

針對訓練資料：

❶ 找出 N 以下的所有連續組合。

❷ 如果這個組合已經在 N-gram 詞彙對照表中, 則將對應的號碼加入資料中。

❸ 如果詞彙對照表中無此組合, 則為這個組合建立編號並加入對照表中, 再將此編號加進資料中。

針對驗證資料：

❶ 找出 N 以下的所有連續組合。

❷ 如果這個組合已經在 N-gram 詞彙對照表中, 則將對應的號碼加入資料中。

❸ 如果詞彙對照表中無此組合, 則在資料中加入 0。

以上的流程已經實現在本章工具模組 utilC.py 中的 n_gram()：

utilC.py　N-gram 處理函式

```
def n_gram(data,n=1,num_words=10000,index=None,append=True):
    if index ==None:  N-gram 詞彙對照表 ┘                  └ 是否要進行資料擴增
        index = {}  ◄── 若 N-gram 詞彙對照表為 None, 建立一個空的對照表
    samples = []    ◄── 輸出資料, 一開始為空 list
    for seq in data:
        new_seq = seq.copy()  ◄── 複製一個序列
        for n_size in range(2,n+1):
            for i in range(len(seq)-(n_size-1)):
                word = tuple(seq[i:i+n_size]) ◄── 找出 N 以下的所有連續組合
                if word in index:
                    new_seq.append(index[word]) ◄┐
                elif append:                     如果這個組合已經在 N-gram 詞彙對
                    num_words+=1                 照表中, 則將對應的號碼加入資料中
                    index[word] = num_words
                    new_seq.append(num_words) ◄┐
                                               如果 N-gram 詞彙對照表中無此組合, 則為這個組合
                                               建立編號並加入對照表中, 再將此編號加進資料中
```

接下頁

```
                else:
                    new_seq.append(0)   ◄── 如果 N-gram 詞彙對照表中
                                            無此組合, 則在資料中加入 0
        samples.append(new_seq)   ◄── 將擴增完的資料加進輸出資料中

    samples = np.array(samples)
    return samples,index,num_words
```

接下來我們用這個函式來處理 IMDB 的資料, 要注意的是, 序列對齊應該放在資料擴增之後：

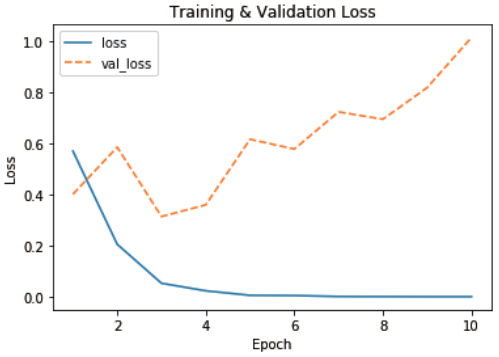
BonusC.3.py　載入 IMDB 資料集並進行資料處理

```
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences

num_words = 10000   ◄── 只處理常見的前 10000 個單字

(train_data, train_labels), (test_data, test_labels) = imdb.load_
data(num_words= num_words)

val_data = train_data[-5000:]   ◄── 取後 5000 筆資料用作驗證資料, 與
                                    validation_split 設定為 0.2 是一樣的意思
train_data = train_data[:-5000] ◄── 取前 20000 (25000-5000) 筆資料用作訓練資料

val_labels = train_labels[-5000:]
train_labels = train_labels[:-5000]

                                    設定 N 為 2,        處理訓練資料時,
import utilC as u                   即 2 元語法         要進行資料擴增

                                                  ↓                ↓
n_train_data, index, num_words = u.n_gram(train_data, n=2, num_
words=num_words)

n_val_data, _, num_words = u.n_gram(val_data, n=2, num_words=num_words,
index=index, append=False)
          ↑                  ↑
將處理訓練資料得到的        處理驗證資料時,
N-gram 詞彙對照表傳入       不須進行資料擴增

maxlen = 1000   ◄── 由於資料擴增後序列會變長, 所以最大序列長度設定為 1000
n_train_data = pad_sequences(n_train_data, maxlen=maxlen,
truncating='post')
n_val_data = pad_sequences(n_val_data, maxlen=maxlen, truncating='post')
```

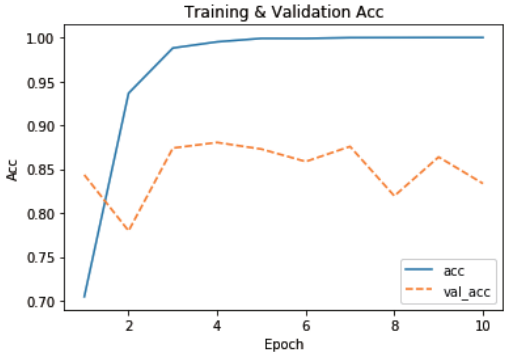
資料處理完後, 就能建立模型並訓練了：

BonusC.3.py(續)　建立模型並訓練

```
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

model = Sequential()
model.add(layers.Embedding(num_words, 24, input_length=maxlen))
model.add(layers.LSTM(20))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(n_train_data, train_labels,
                    epochs=10,
                    batch_size=128,
                    validation_data=(n_val_data, val_labels)
                    )
```



LSTM 處理 N-gram 的訓練和驗證損失



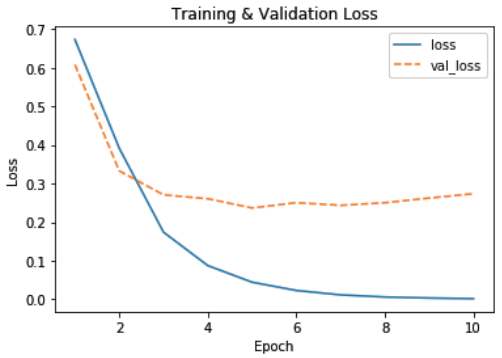
LSTM 處理 N-gram 的訓練和驗證損失

最好的驗證準確率為 88%, 並沒有什麼出色的表現, 在做過這麼多努力後, 依然沒什麼顯著的成果, 或許這再次說明了 IMDB 的分類問題不適合使用 RNN。最後為了驗證這個說法, 我們嘗試使用 N-gram, 但不使用 RNN, 看看效果如何。

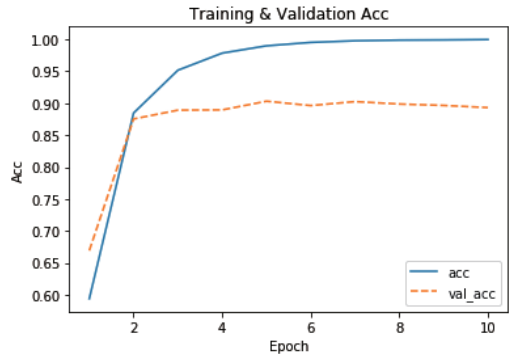
資料處理的方式與 BonusC.3.py 相同, 只要更改神經網路架構如下即可：

BonusC.4.py　使用 N-gram 的密集神經網路來處理 IMDB 問題

```
…(略, 資料處理方式同 BonusC.3.py)

model = Sequential()
model.add(layers.Embedding(num_words, 24, input_length=maxlen))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(n_train_data, train_labels,
                    epochs=10,
                    batch_size=128,
                    validation_data=(n_val_data, val_labels)
                    )
```



使用 N-gram 密集神經網路的訓練和驗證損失



使用 N-gram 密集神經網路的訓練和驗證準確率

從結果可以看到, 最好的驗證準確率終於超過 90% 了！這就說明了特徵工程是確實有效的, 不過 IMDB 不太適合使用 RNN, 這也不代表所有的文字資料都不適合用 RNN, 在第 6 章, 我們便能看到 RNN 在自然語言處理中發揮其作用。