



# 國立高雄科技大學

National Kaohsiung University of Science and Technology

## 深度學習理論與實作

### CH6 應用於文字資料與序列資料的深度學習

資工系 陳俊豪 教授



# Outline



6-1 文件資料處理

6-2 循環神經網路RNN

6-3 循環神經網路的進階使用方法

6-4 使用卷積神經網路進行序列資料處理

## 6-1 文件資料



- ✓ 文件資料是一種常見的序列資料
- ✓ 使用文件資料建立能理解自然語言的深度模型，可以用於：



文件分類



情感分析



有限條件的問答系統

## 6-1 文件資料



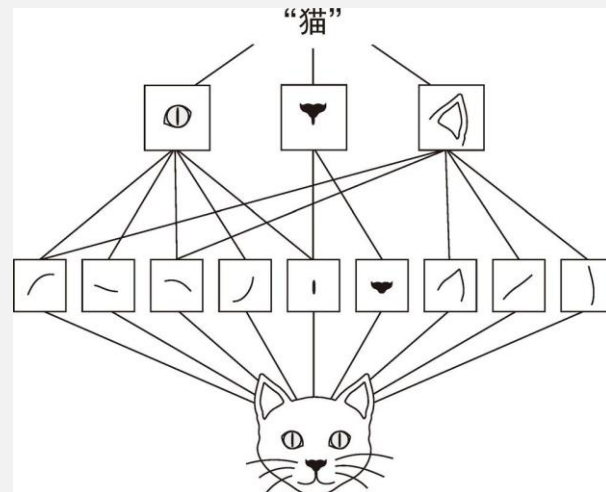
### ✓ 自然語言的深度學習

- 用pattern辨識的技術來處理單字、句子和文章的相關問題
- 其方式和電腦視覺用pattern辨識技術來處理影像的問題十分類似



### 自然語言

卷積神經網路：將文件拆分成小單元(單字、字元或n元語法)，再組合成想得到的答案(如：文章摘要、情感分析等)



### 電腦視覺

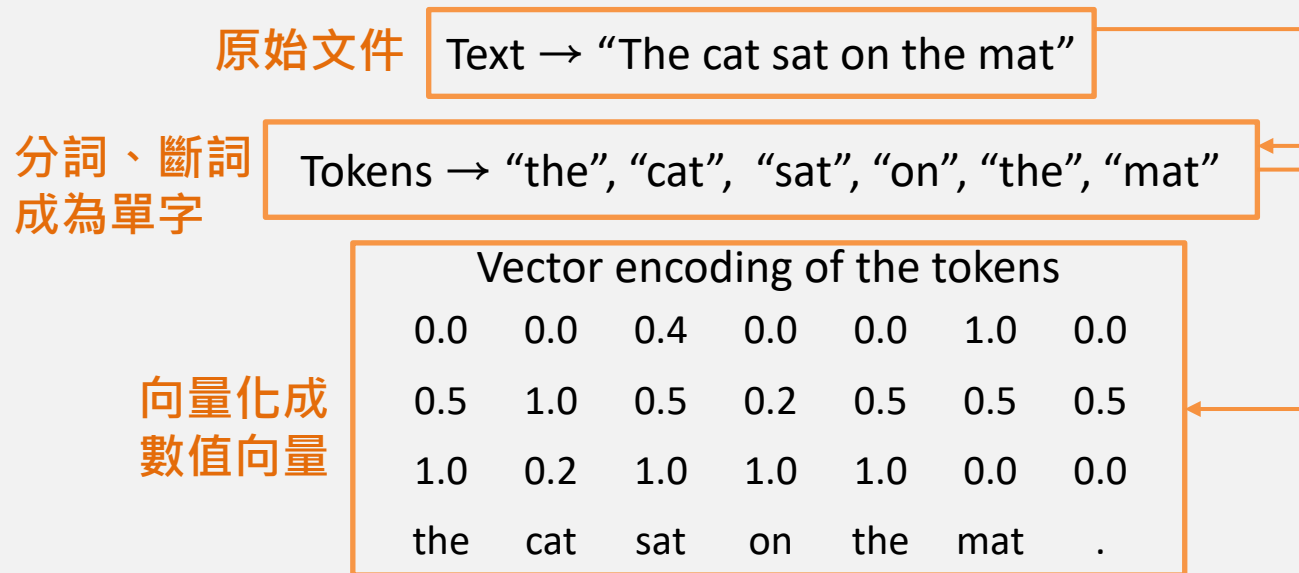
卷積神經網路：將人類視覺所建的圖，拆分為局部圖案，再組合成高階抽象事物(如：貓)

## 6-1 文件資料的預處理 (preprocessing)



### ✓ 文件向量化(text vectorizing)

- Step 1：「分詞」、「斷詞」：將文件分解成token小單元(單字、字元或n元語法)
- Step 2：透過字典對照表將token編碼成數值向量
- Step 3：數值向量打包成序列張量送入深度神經網路



從文件資料轉換成tokens，再轉換成數值向量

## 6-1 文件資料的預處理preprocessing



### ✓ Step 1 分詞方法

- 單字→向量
- 字元→向量
- N元語法→向量

說明：

N元語法=N元語法袋：從句子中取出N個連續文字的組合

- “The cat sat on the mat” 分解成2元語法
- { “the” , “ the cat” , “cat” , “cat sat” , “sat” , “sat on” , “on” , “on the” , “the” , “the mat” , “mat” }

### □ 語法袋(bag of words)

語法袋中的詞彙沒有順序性，如同把糖果放到一個袋子中，沒有順序性，因此多用於淺層模型，深度學習運用階層式特徵學習

Bag of words (BoW)

Very good drama although it appeared to have a few blank areas leaving the viewers to fill in the action for themselves. I can imagine life being this way for someone who can neither read nor write. This film simply smacked of the real world: the wife who is suddenly the sole supporter, the live-in relatives and their quarrels, the troubled child who gets knocked up and then, typically, drops out of school, a jackass husband who takes the nest egg and buys beer with it. 2 thumbs up... very very very good movie.



('the', 8),  
(',', 5),  
( 'very', 4),  
( ':', 4),  
( 'who', 4),  
( 'and', 3),  
( 'good', 2),  
( 'it', 2),  
( 'to', 2),  
( 'a', 2),  
( 'for', 2),  
( 'can', 2),  
( 'this', 2),  
( 'of', 2),  
( 'drama', 1),  
( 'although', 1),  
( 'appeared', 1),  
( 'have', 1),  
( 'few', 1),  
( 'blank', 1)  
.....

## 6-1 文件資料的預處理preprocessing



✓ Step 2 Token轉換成向量的方法

- Token的one-hot encoding
- Token嵌入法(Token embedding)

✓ 如果是專門用於文字資料，就稱為word embedding

		cat	mat	on	sat	the
the =>		0	0	0	0	1
cat =>		1	0	0	0	0
sat =>		0	0	0	1	0
...						...

One-hot encoding

Input		[CLS]	my	dog	is	cute	[SEP]	he	likes	play	##ing	[SEP]	[PAD]
Token Embeddings		E <sub>[CLS]</sub>	E <sub>my</sub>	E <sub>dog</sub>	E <sub>is</sub>	E <sub>cute</sub>	E <sub>[SEP]</sub>	E <sub>he</sub>	E <sub>likes</sub>	E <sub>play</sub>	E <sub>++ing</sub>	E <sub>[SEP]</sub>	
		+	+	+	+	+	+	+	+	+	+	+	
Segment Embeddings		E <sub>A</sub>	E <sub>A</sub>	E <sub>A</sub>	E <sub>A</sub>	E <sub>A</sub>	E <sub>A</sub>	E <sub>B</sub>	E <sub>B</sub>	E <sub>B</sub>	E <sub>B</sub>	E <sub>B</sub>	
		+	+	+	+	+	+	+	+	+	+	+	
Position Embeddings		E <sub>0</sub>	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	E <sub>7</sub>	E <sub>8</sub>	E <sub>9</sub>	E <sub>10</sub>	
input_ids	tokens_tensor	5566	1	2	3	4	9527	5	6	7	8	9527	0
	每個 token 的索引值												
token_type_ids	segments_tensor	0	0	0	0	0	0	1	1	1	1	1	0
	識別句子界限												
attention_mask	masks_tensor	1	1	1	1	1	1	1	1	1	1	1	0
	界定自注意力機制範圍												

Word embedding

## 6-1-1單字和字元的one-hot encoding



### ✓ 單字的 one-hot encoding (簡易版)

- 先建一個字典，把文章中的單字收錄為鍵(Key)
- 給鍵一個唯一的整數，作為鍵值(value)

```
import numpy as np
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
# 初始資料：此list的每一個元素是一個樣本，每一個樣本是一個輸入項目
# 在這範例中，樣本是一個句子，但也可以是整個文件

token_index = {} # 建立一個空字典，儲存資料中所有 tokens和其鍵值
for sample in samples: # 取samples中的一個sample
    for word in sample.split(): # 透過 split()方法對樣本進行分詞。在真實案例中，還要移除樣本中的標點符號與特殊字元
        if word not in token_index:
            token_index[word] = len(token_index) + 1 # 以新字token為鍵。請注意，不要把索引 0 指定給任何文字
```



## 6-1-1單字和字元的one-hot encoding



✓ (續)單字的 one-hot encoding (簡易版)

#以下開始將字串的token轉向量

max\_length = 10 # 將樣本向量化。每次只專注處理每個樣本中的前max\_length 文字

```
results = np.zeros(shape=(len(samples), # 用來儲存結果的 Numpy array
                        max_length,
                        max(token_index.values()) + 1))
print(results.shape)
```

Out[] : shape=(2, 10, 11)

說明：

Results是三軸張量

共 2 個樣本, 每個樣本只看前 10 個文字, 總樣本共有 10 個 token, 索引號到 11, 因為 0 不用

## 6-1-1單字和字元的one-hot encoding



✓ (續)單字的 one-hot encoding (簡易版)

```
for i, sample in enumerate(samples): #把第0軸第i元素、第1軸第j元素、第2軸第index元素設為1
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = token_index.get(word) #從字典取得word的鍵值(即位置)
        results[i, j, index] = 1.
print(token_index)
```

Out[ ] :

```
(2, 10, 11)
{'The': 1, 'cat': 2, 'sat': 3, 'on': 4, 'the': 5, 'mat.': 6, 'dog': 7, 'ate': 8, 'my': 9, 'homework.': 10}
```

說明：

list(enumerate(sample.split()))[:max\_length]拆解：

1. samples = 'The cat sat on the mat.'
2. sample.split()為一個list · ['The', 'cat', 'sat', 'on', 'the', 'mat.']
3. list放入enumerate() · 轉為[(0, 'The'), (1, 'cat'), (2, 'sat'), (3, 'on'), (4, 'mat.')]
4. [:max\_length]只看list的前max\_length項→10

## 6-1-1單字和字元的one-hot encoding



✓ 字元的 one-hot encoding (簡易版)

```
import string
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
characters = string.printable # 所有可印出的 ASCII 字元的字串, '0123456789abc...'
print(len(characters)) # 共100種字元(tokens)
token_index = dict(zip(characters, range(1, len(characters) + 1))) # 此字彙表字典儲存字元與對應的鍵值，
# 0一樣不使用
max_length = 50
results = np.zeros((len(samples), max_length, max(token_index.values()) + 1))
print(results.shape) # shape=(2,50,101)共有兩個樣本，只看每個樣本的前50個字，共有100種
# token(字元)，索引值最大為101，因為0不使用
```

```
Out[] : 100
        (2, 50, 101)
```

## 6-1-1單字和字元的one-hot encoding



✓ (續)字元的 one-hot encoding (簡易版)

```
for i, sample in enumerate(samples):  
    for j, character in enumerate(sample):  
        index = token_index.get(character)  
        results[i, j, index] = 1.  
  
print(results[0][0])
```

```
Out[] : [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  
 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  
 0. 0. 0. 0. 0.]
```

## 6-1-1 單字和字元的one-hot encoding



- ✓ 用 Keras 做文字的 one-hot encoding(Multi-hot encoding)
  - 可以從字串中刪除特殊字元
  - 專注於資料集中前N個最常用的文字(忽視大小寫)
  - 限制num\_words的數目可以控制運算量

```
from keras.preprocessing.text import Tokenizer # 匯入 Keras 分詞器
```

```
samples = ['The cat sat on the mat.', 'The dog ate my homework.'] # 初始資料 · Tokenizer測試用字串集
```

```
tokenizer = Tokenizer(num_words=1000) # 建立分詞器, 設定上僅考慮 1000 個最常用的文字 (token),  
也就是只會看初始資料的前 1000 個文字
```

```
tokenizer.fit_on_texts(samples)
```

```
# 建立字典(建立文字對應的索引值), 一樣依出現順序來決定, 0 一樣不使用
```

```
sequences = tokenizer.texts_to_sequences(samples)
```

```
# 將samples裡字串的每個單字用字典轉換成鍵值list
```

```
print(sequences)
```

```
Out[] : [[1, 2, 3, 4, 1, 5], [1, 6, 7, 8, 9]]
```

## 6-1-1單字和字元的one-hot encoding



✓ (續)用 Keras 做文字的 one-hot encoding

```
one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')  
# 可以直接取得 one-hot 的二進位表示方式  
# 此分詞 tokenizer 支援除了 one-hot 編碼以外, 也有支援其他的向量化方法
```

```
print(one_hot_results.shape) # (2, 1000) 共 2 個樣本, 每個樣本中的文字對應到的 token 位置 (1000個)  
word_index = tokenizer.word_index # 計算完成後, 取得文字與索引間的對應關聯  
print(word_index) # {'the': 1, 'cat': 2, 'sat': 3, ... 'my': 8, 'homework': 9}  
print('找到 %s 個唯一的 tokens.' % len(word_index))
```

Out[ ] :

```
(2, 1000)  
{'the': 1, 'cat': 2, 'sat': 3, 'on': 4, 'mat': 5, 'dog': 6, 'ate': 7, 'my': 8, 'homework': 9}  
找到 9 個唯一的 tokens.
```

## 6-1-1單字和字元的one-hot encoding



✓ 使用雜湊技巧的單字 one-hot encoding (簡易版)

- 當字典中的tokens數量太大時
- 不需要維護一個完整、明確的token字典
- 資料一邊進來一邊編碼，節省記憶體

```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
dimensionality = 1000 # 將文字儲存成大小為 1000 的向量。如果有接近1000 個文字(或更多),
max_length = 10      # 將會造成許多雜湊碰撞, 這會降低此編碼方法的準確性

results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = abs(hash(word)) % dimensionality
        results[i, j, index] = 1. # ← 將文字雜湊成 0 到 1000 之間的
                                   隨機整數索引
print(results.shape)
```

Out[] : (2, 10, 1000)

## 6-1-2 使用文字嵌入法 Word Embeddings



文字嵌入法(word embeddings)  
=密集文字向量 (dense word vector)



低維浮點數向量

One-hot encoding



高維稀疏向量

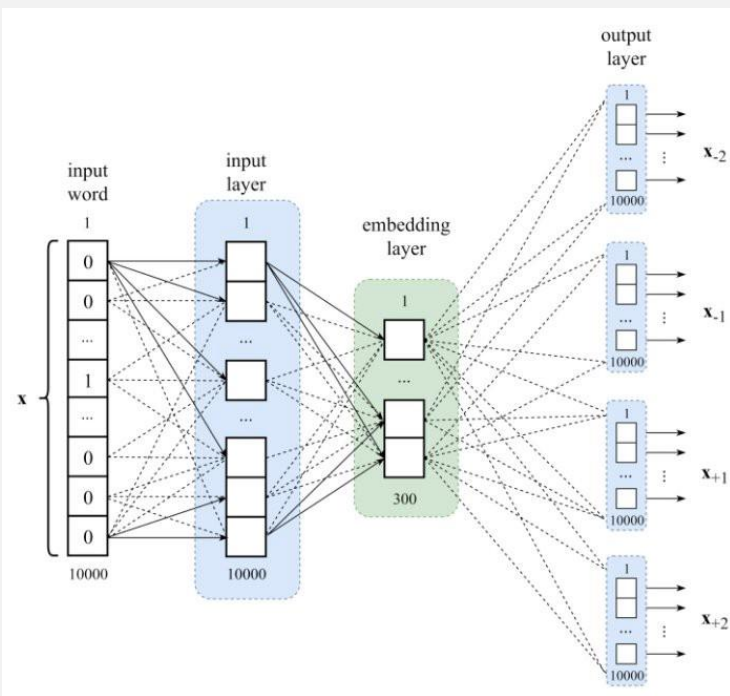


## 6-1-2 使用文字嵌入法 Word Embeddings

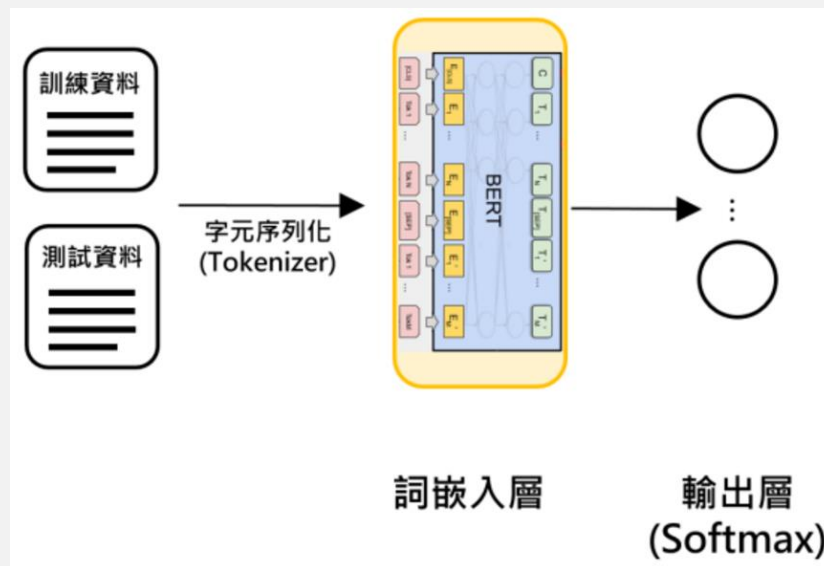


✓ 有兩種方法可以建立文字嵌入向量

1. 用 Embedding layer 同時學習文字嵌入向量(如同學習神經網路權重的方式)



2. 預先訓練的文字嵌入法(pretrained word embeddings)，用機器學習模型已經訓練好的文字嵌入向量

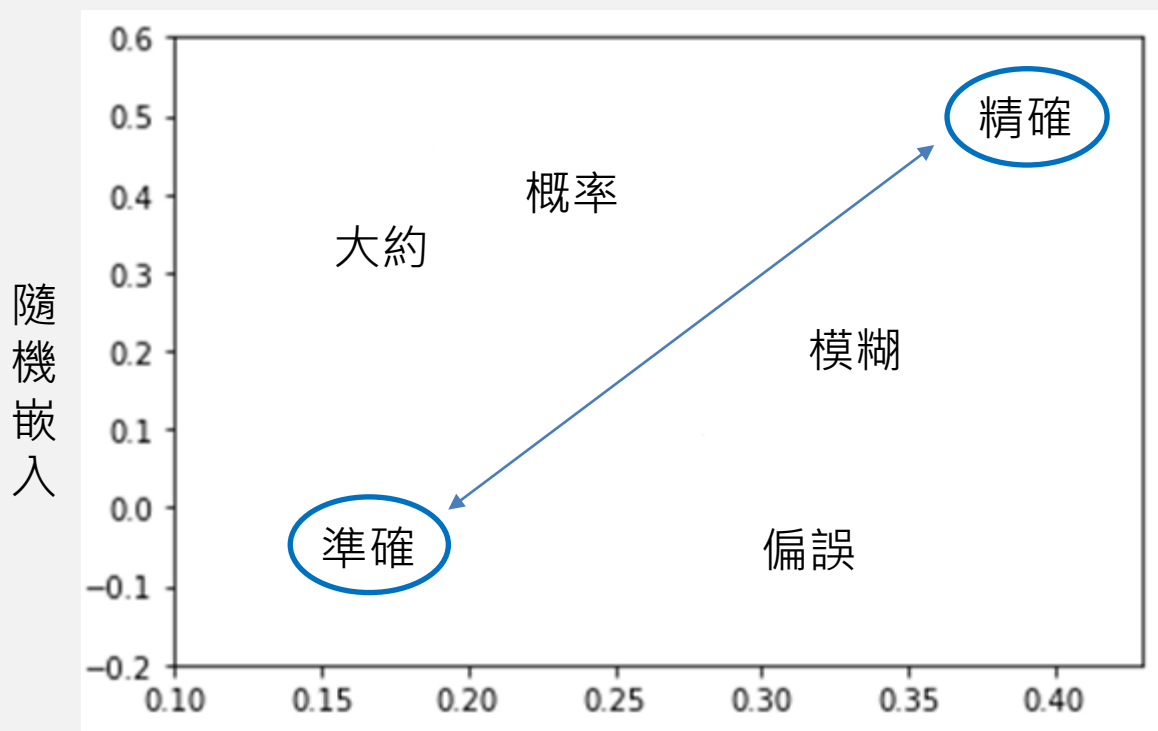


## 6-1-2 使用文字嵌入法 Word Embeddings



✓ 用Embedding layer同時學習文字嵌入向量

- 可以把任何文字映射(mapping)到向量，稱作「隨機嵌入」，同義字可能被隨機嵌入到很遠的位置，深度神經網路難以理解非結構化的嵌入空間

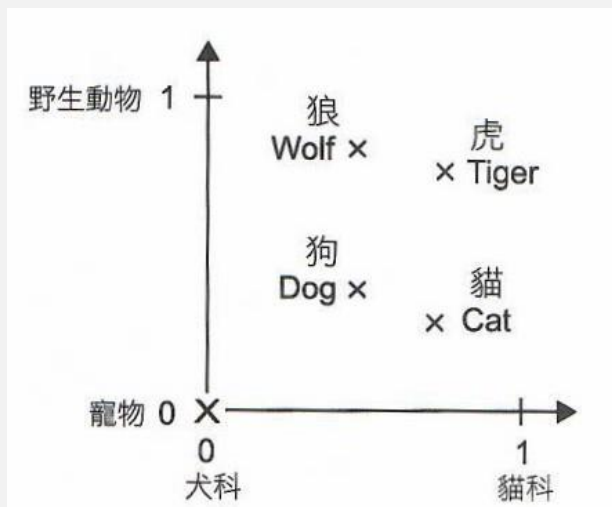


同義字被隨機嵌入到很遠的位置

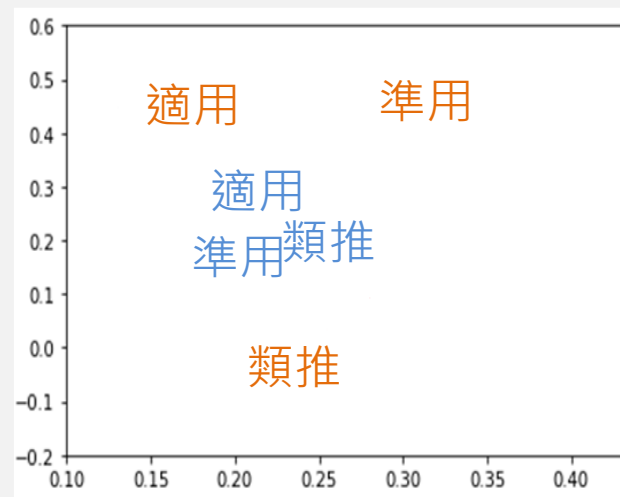
## 6-1-2 使用文字嵌入法 Word Embeddings



- ✓ 用Embedding layer同時學習文字嵌入向量
  - 除了兩個文字向量間的幾何距離(distance)外，還希望方向(directions)也和字義有所相關性，因此，文字向量(word vectors)之間的幾何空間關係應該反映這些文字的語義關係(semantic)



字義相近的距離較近，  
字義較遠的距離較遠



專業法律術語  
日常文章

某些字義會因為適用領域而有很大差異，可透過反向傳播演算法加強學習，使用Keras的Embedding layer將輸入累積到權重裡

## 6-1-2 使用文字嵌入法 Word Embeddings



### ✓ 建立一個嵌入層 (Embedding Layer)

```
from keras.layers import Embedding
# 建立嵌入向量層至少須指定兩個參數：
# 可能的tokens 數量 (此處為 1,000, 最少是 1+最大文字索引) 和嵌入向量的維數 (此處為 64)
embedding_layer = Embedding(1000, 64)
```

Embedding layer回傳3D浮點張量，  
shape是(samples, sequence\_length,  
embedding\_dimensionality)的3D張量

想像成字典，  
會把鍵值投射  
到密集向量

Word index  
文字編號(鍵值)

Embedding layer  
嵌入層

Corresponding word vector  
對應的文字密集向量

Input :

- shape是(samples, sequence\_length)的2D張量
- sequence\_length要相同長度，因為要打包到張量中

權重(token向量的內部字典)初始是隨機設定的，透過反向傳播調整

## 6-1-2 使用文字嵌入法 Word Embeddings



✓ 載入 IMDB, 整理成適合供 Embedding 層使用的資料

```
from keras.datasets import imdb
from keras_preprocessing.sequence import pad_sequences #!pip install Keras-Preprocessing

max_features = 10000 # 設定作為特徵的文字數量
maxlen = 20 # 在 20 個文字之後切掉文字資料 (在 max_features 最常見的文字中)

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features) # 將資料以整數 lists 載入
print(x_train.shape)
x_train = pad_sequences(x_train, maxlen=maxlen) # 將整數 lists 轉換為 2D 整數張量, 形狀為(樣本數
samples, 最大長度 maxlen)
print(x_train.shape)
print(x_train[0])
x_test = pad_sequences(x_test, maxlen=maxlen)
```

```
Out[] : (25000,)
        (25000, 20)
        [ 65  16  38 1334  88  12  16 283   5  16 4472 113 103  32
         15  16 5345  19 178  32]
```

## 6-1-2 使用文字嵌入法 Word Embeddings



- ✓ 把 IMDB 資料提供給 Embedding layer和分類器

```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Embedding

model = Sequential()
# 指定嵌入向量層的最大輸入長度, 以便之後可以攤平嵌入向量的輸入。在嵌入向量層之後, 啟動函數輸出的 shape 為 (樣本數 samples, 最大長度 maxlen, 8 )
model.add(Embedding(10000, 8, input_length=maxlen))
#將嵌入向量的 3D 張量展平為 2D 張量, 形狀為(樣本數 samples, 最大長度 maxlen * 8)
model.add(Flatten())
model.add(Dense(1, activation='sigmoid')) # ← 在頂部加上分類器
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

## 6-1-2 使用文字嵌入法 Word Embeddings



✓ (續)把 IMDB 資料提供給 Embedding layer和分類器

Out[] : Model: "sequential"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 20, 8)	80000
flatten (Flatten)	(None, 160)	0
dense (Dense)	(None, 1)	161

=====  
Total params: 80,161  
Trainable params: 80,161  
Non-trainable params: 0

Epoch 10/10

625/625 [=====] - 1s 2ms/step - loss: 0.3028 - acc: 0.8775 - val\_loss: 0.5197 - val\_acc: 0.7502

10000個字\*每個  
字8維的嵌入向量  
=80000(上一頁第  
4行程式碼)

說明：

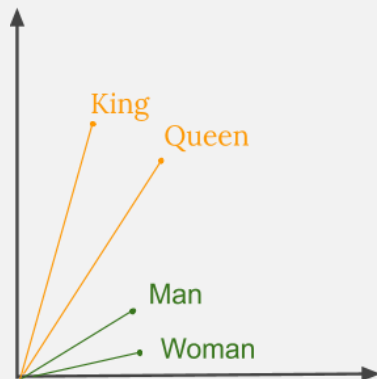
驗證準確度約75.02%，因為只用每個評論的前20個文字，所以結果還算不錯。

## 6-1-2 使用文字嵌入法 Word Embeddings



- ✓ 預先訓練的文字嵌入向量 (Pretrained word embeddings)
  - 使用時機：當訓練資料不足，無法自行提供足夠的資料來學習可用的文字嵌入向量時，可以拿普遍通用的特徵來使用，也就是其他問題上學習到的通用特徵
- ✓ 兩種最常見的文字嵌入向量：

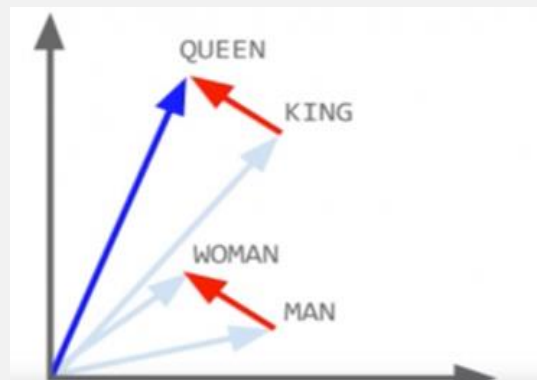
### Word2Vec演算法



Google開發

### 全域向量文字表示法

(Global Vectors for Word Representation, GloVe)



史丹佛大學團隊開發



## 6-1-3 整合實作：從原始資料到文字嵌入向量



- ✓ 從原始的IMDB資料集開始，將句子以嵌入向量方法轉換向量序列資料，然後展平並在頂部訓練一個密集層
  - Step 1: 處理原始 IMDB 資料的標籤，因為IMDB資料集是有正負評的，把是負評的部分轉為0，正評轉為1
  - Step 2: 對原始 IMDB 資料的文字資料進行向量化
  - Step 3: 解析 GloVe 文字嵌入向量檔案
  - Step 4: 準備 GloVe 文字嵌入向量矩陣
  - Step 5: 將預訓練的GloVe權重矩陣載入到嵌入向量層中
  - Step 6: 訓練
  - Step 7: 驗證
  - Step 8: 看結果

## 6-1-3 整合實作：從原始資料到文字嵌入向量



### ✓ Step 1. 處理原始 IMDB 資料的標籤

```
import os
imdb_dir = r'C:\Users\chche\aclImdb'
train_dir = os.path.join(imdb_dir, 'train')
labels = []
texts = []
for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname), encoding = 'utf8')
            texts.append(f.read()) #讀取評論字串資料存入texts
            f.close()
            if label_type == 'neg':
                labels.append(0) #負評為0
            else:
                labels.append(1) #正評為1
```

下載資料及網址：

<https://ai.stanford.edu/~amaas/data/sentiment/>

## 6-1-3 整合實作：從原始資料到文字嵌入向量



✓ Step 2. 對原始 IMDB 資料的文字資料進行向量化

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np
```

```
maxlen = 100 # 100 個文字後切斷評論 (只看評論的前 100 個字)
```

```
training_samples = 200 # 以 200 個樣本進行訓練
```

```
validation_samples = 10000 # 以 10,000 個樣本進行驗證
```

```
max_words = 10000 # 僅考慮資料集中的前 10,000 個單詞
```

```
tokenizer = Tokenizer(num_words=max_words)
```

```
tokenizer.fit_on_texts(texts)
```

```
sequences = tokenizer.texts_to_sequences(texts) # 將文字轉成整數 list 的序列資料
```

```
word_index = tokenizer.word_index # { 'the' : 1, 'cat' : 2, 'sat' : 3,... } 文字與索引的詞彙表
```

```
print('共使用了 %s 個 token 字詞.' % len(word_index))
```

```
Out[] : 88582 # 共使用了 88582 個 token 字詞
```

## 6-1-3 整合實作：從原始資料到文字嵌入向量



✓ (續)對原始 IMDB 資料的文字資料進行向量化

```
data = pad_sequences(sequences, maxlen=maxlen) # 只取每個評論的前 100 個字 (多切少補) 作為資料張量
labels = np.asarray(labels) # 將標籤 list 轉為 Numpy array (標籤張量)
print('資料張量 shape:', data.shape) # 資料張量shape: (25000, 100)
print('標籤張量 shape:', labels.shape) # 標籤張量shape: (25000,)
```

```
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]
```

```
x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

# 將資料拆分為訓練集和驗證集, 但首先要將資料打散, 因為所處理的資料是有順序性的樣本資料 (負評在前, 然後才是正評)

## 6-1-3 整合實作：從原始資料到文字嵌入向量



- ✓ Step 3. 解析 GloVe 文字嵌入向量檔案
  - Word2vec和GloVe都是著名預先訓練的嵌入向量

```
glove_dir = r'C:\Users\chche'
```

前往下列網址下載glove.6B.zip  
<https://nlp.stanford.edu/projects/glove/>

```
embeddings_index = {} #建立嵌入向量索引字典
```

```
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'), encoding='UTF-8')
```

```
for line in f:
```

```
    values = line.split() #以空格切開每一行字串
```

```
    word = values[0] #(字 & 100向量)
```

```
    coefs = np.asarray(values[1:], dtype='float32') #將座標值轉為numpy array, (100向量)
```

```
    embeddings_index[word] = coefs #儲存文字(鍵)與向量(值)
```

```
f.close()
```

```
print('共有 %s 個文字嵌入向量' % len(embeddings_index))
```

Out[] : 400000 #用GloVe這個預先訓練的嵌入向量，總共內建40,000個文字嵌入向量

## 6-1-3 整合實作：從原始資料到文字嵌入向量



### ✓ Step 4. 準備 GloVe 文字嵌入向量矩陣

```
embedding_dim = 100
```

```
embedding_matrix = np.zeros((max_words, embedding_dim)) #(10000, 100) , 只考慮前10000個字 , 每個字的向量維度為100
```

```
for word, i in word_index.items():
```

```
    if i < max_words:
```

```
        embedding_vector = embeddings_index.get(word)
```

```
        if embedding_vector is not None: #如果這個字有向量表示法
```

```
            embedding_matrix[i] = embedding_vector # ←將該向量表示法存入矩陣中 , 嵌入向量索引中找不到的文字將為 0
```

說明：

雖然GloVe中內建四萬個字，但還是不包含所有的字詞，所以如果沒有內建的字詞就會轉為0

## 6-1-3 整合實作：從原始資料到文字嵌入向量



### ✓ Step 5. 模型定義

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense
```

```
model = Sequential()
# 參數樣本數(可能的token數量), 嵌入向量維度, 只看評論的前 100 個字
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
#model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

(10000, 100, 100)

說明：

若要在embedding層後連接flatten層與dense層，需要設定input\_length，代表輸入序列的最大長度

## 6-1-3 整合實作：從原始資料到文字嵌入向量



✓ (續)模型定義

Out[] : Model: "sequential\_1"

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, 100, 100) 樣本數, 序列長度, 嵌入向量維度	1000000
flatten_1 (Flatten)	(None, 10000)	0
dense_1 (Dense)	(None, 32)	320032
dense_2 (Dense)	(None, 1)	33
Total params: 1,320,065		
Trainable params: 1,320,065		
Non-trainable params: 0		

$(10000+1)*32=320032$

$(32+1)*1=33$

說明：

- embedding層：10000個字\*每個字100維=1000000 (上一頁第4行程式碼)
- 全連接層：Param = (輸入數據維度+1) \*神經元個數，之所以要加1，是考慮到每個神經元都有一個Bias
- 輸入數據維度=上一層的輸出



## 6-1-3 整合實作：從原始資料到文字嵌入向量



- ✓ 將預訓練的GloVe權重矩陣載入到嵌入向量層中

嵌入向量層

設定層的權重

載入GloVe權重矩陣

```
model.layers[0].set_weights([embedding_matrix])
```

```
model.layers[0].trainable = False #凍結嵌入層，把trainable設為False，當模型的某些部份是預先訓練，則在訓練時不應更新預先訓練的部分
```

## 6-1-3 整合實作：從原始資料到文字嵌入向量



✓ Steps 6 & 7 訓練和驗證

```
model.compile(optimizer='rmsprop', #進行訓練
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                   epochs=10,
                   batch_size=32,
                   validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5') #儲存訓練後的模型
```

## 6-1-3 整合實作：從原始資料到文字嵌入向量



✓ 繪製結果

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
```

## 6-1-3 整合實作：從原始資料到文字嵌入向量

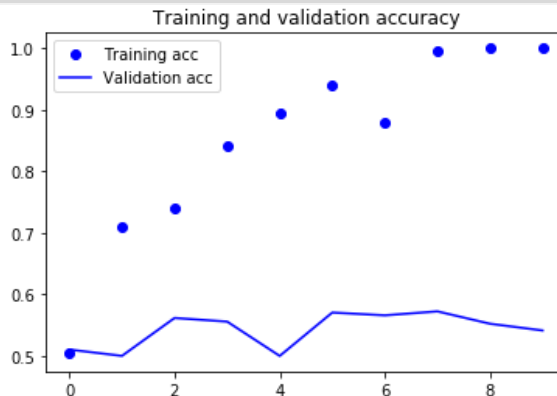
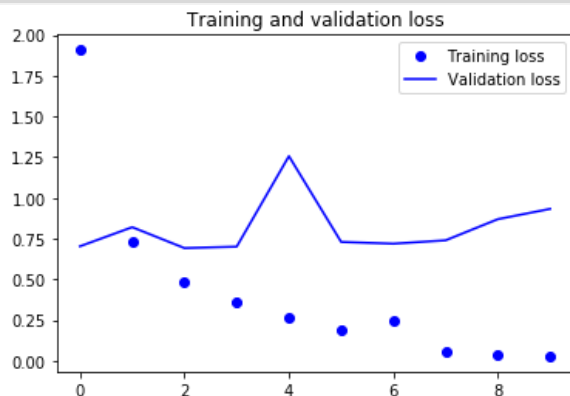


✓ 繪製結果(續)

```
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

Out[ ] :



說明：

- 驗證的準確度只有0.5，因為訓練樣本很少，馬上就過度配適
- 每個人執行的結果可能會有所不同，因為訓練樣本很少，所以隨機選擇後表現結果也會不同

## 6-1-3 整合實作：從原始資料到文字嵌入向量



- ✓ 訓練相同模型而**不使用**預先訓練的文字嵌入向量

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

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

history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_val, y_val))
```

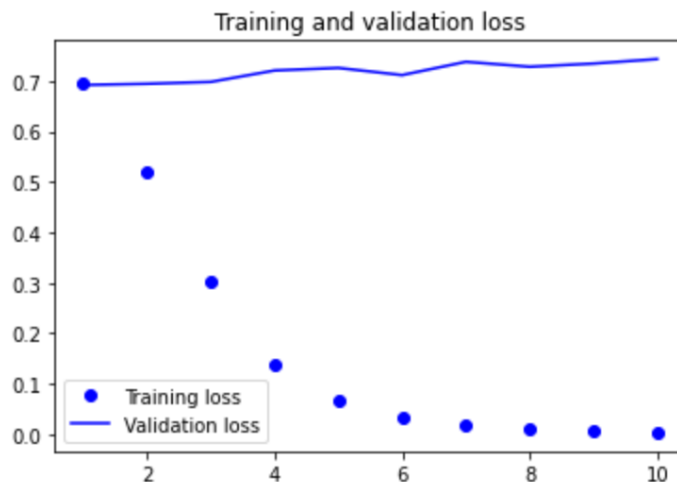
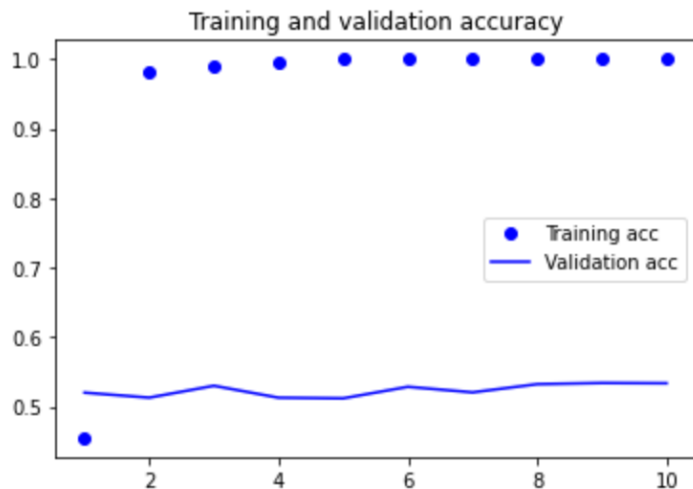
## 6-1-3 整合實作：從原始資料到文字嵌入向量



✓ 繪製結果

```
model.add(Dense(1, activation='sigmoid'))  
model.summary()
```

Out[ ] :



說明：

- 驗證準確度停滯於50%，由此可知在訓練樣本不足的情況下，使用預先訓練的文字嵌入向量優於從頭開始學習模型，如果我們增加訓練樣本的數量，則結果將會有所不同

## 6-1-3 整合實作：從原始資料到文字嵌入向量



### ✓ Step 8. (Testing) 對測試資料進行資料前處理

```
test_dir = os.path.join(imdb_dir, 'test')
labels = []
texts = []
for label_type in ['neg', 'pos']:
    dir_name = os.path.join(test_dir, label_type)
    for fname in sorted(os.listdir(dir_name)):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname), encoding='UTF-8')
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)
```

## 6-1-3 整合實作：從原始資料到文字嵌入向量



### ✓ Step 8. 載入訓練好模型並測試結果

```
#載入訓練好的模型，並評估測試結果
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

Out[ ] :

```
782/782 [=====] - 3s 4ms/step - loss: 0.7672 - acc: 0.5540
[0.7672020196914673, 0.5540400147438049]
```

說明：

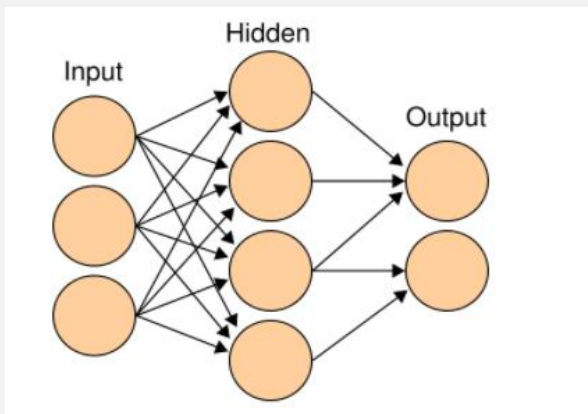
- 使用預訓練的文字嵌入向量建立之模型，測試準確度約**55%**，僅使用少量訓練樣本 (training\_samples = 200 **# 以 200 個樣本進行訓練**)是**很難得**可以得到這樣的結果!



## 6-2 循環神經網路 (Recurrent Neural Network)

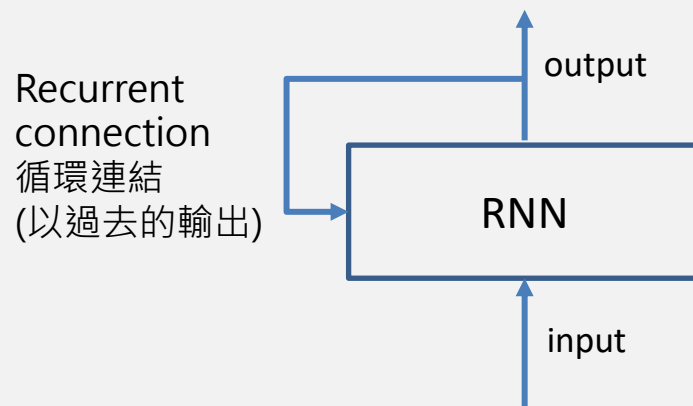


### 前饋式神經網路 (feedforward networks)



- 無使用記憶體：每個輸入資料都是獨立處理的
- 處理序列資料，需將整個資料轉成單一資料
  - ✓ 如：IMDB 電影評論轉成單一大型向量一次處理
- 包含：全連接神經網路、卷積神經網路

### 循環神經網路 (recurrent neural network, RNN)

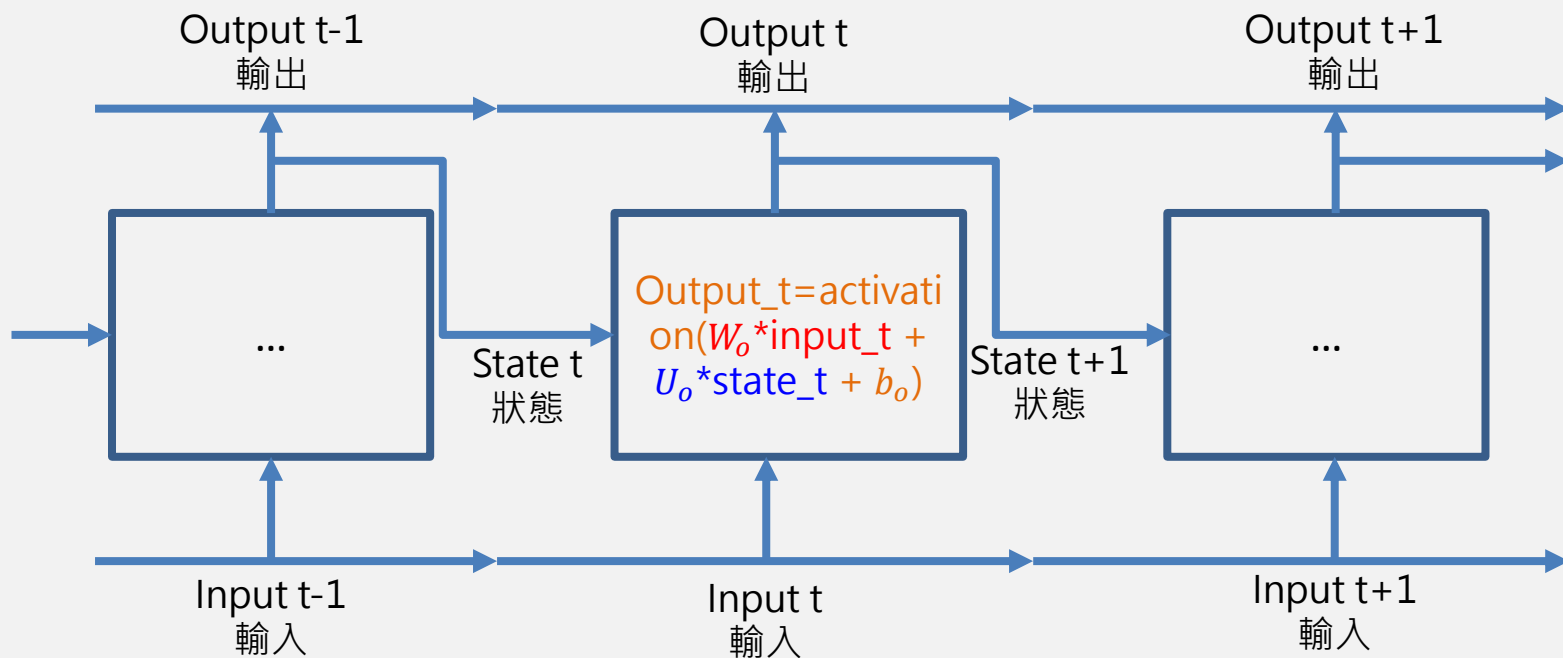


- 有記憶：透過一個持續與過去資訊有相關的狀態state來迭代處理序列資料，如同閱讀一本書，會依據前面章節的資訊更新
- 循環神經網路RNN
  - ✓ 具內部循環的神經網路
  - ✓ 在處理兩不同、獨立的序列資料時，RNN的狀態會被重置

## 6-2 循環神經網路RNN



- ✓ RNN的特徵在於**階躍函數**，重複使用迴圈中前一次迭代計算得到的數值結果



簡單的RNN，隨著時間的推移展開

## 6-2 循環神經網路



### ✓ 以Numpy實現簡單的RNN

```
import numpy as np

timesteps = 100 # 輸入序列資料中的時間點數量
input_features = 32 # 輸入特徵空間的維度數
output_features = 64 # 輸出特徵空間的維度數

inputs = np.random.random((timesteps, input_features)) # 輸入資料：隨機產生數值以便示範

state_t = np.zeros((output_features, )) # 初始狀態：全零向量 (沒有當前狀態)

W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features)) # 建立隨機權重矩陣
b = np.random.random((output_features, )) # 建立隨機偏移向量
```

補充：

- RNN將序列化向量作為資料，編碼成2D張量  $\text{shape}=(\text{timesteps}, \text{input\_features})$
- 每個時間點上，考慮該時間點t的當前狀態與輸入，一起處理以得到該點的輸出，供下一時間點使用

## 6-2 循環神經網路RNN



✓ (續)以Numpy實現簡單的RNN

```
successive_outputs = [] #用來儲存各個時間點的輸出
for input_t in inputs: # input_t 是個向量, shape 為 (input_features,)
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b) # 結合輸入與當前狀態(前一個輸出)
                                                                    # 以取得當前輸出
    successive_outputs.append(output_t) # 將此輸出儲存在列表中
    state_t = output_t #更新下一個時間點的神經網絡狀態

#串接所有時間點的輸出作為最終輸出
final_output_sequence = np.concatenate(successive_outputs, axis=0)
print(final_output_sequence.shape) # 100, 64
```

```
Out[] : shape=(6400,)
```

## 6-2-1 Keras中的一個循環層



✓ SimpleRNN是Keras中的其中一個循環層

差異	SimpleRNN (from Keras)	Numpy
資料類型	處理 <u>批次量</u> 的序列資料	處理 <u>單一個</u> 序列資料
input	3D張量 shape=(batch_size批次量大小, timesteps, input_features)	2D張量 shape=(timesteps, input_features)

- ✓ SimpleRNN由return sequences參數控制，且可以用兩種不同的方法執行：
1. 回傳每個時間點連續輸出的完整序列資料(3D張量)
  2. 回傳對每個輸入序列資料的最後一個輸出(2D張量)

## 6-2-1 Keras中的一個循環層



### SimpleRNN

僅在最後一個時間點回傳輸出的範圍

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN
model= Sequential()
model.add(Embedding(10000,32))
model.add(SimpleRNN(32))
model.summary()
```

(Token數量,嵌入向量的維度)

輸出特徵

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, None, 32)	320000
simple_rnn (SimpleRNN)	(None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

### SimpleRNN

回傳完整的狀態序列資料

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN
model= Sequential()
model.add(Embedding(10000,32))
model.add(SimpleRNN(32,return_sequences=True))
model.summary()
```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 32)	320000
simple_rnn_2 (SimpleRNN)	(None, None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

批次量、時間點數量未定所以為None

補充：

- 沒有設定 return\_sequences 則預設false

## 6-2-1 Keras中的一個循環層



✓ 堆疊多個循環層可增加神經網路表現能力

```
model= Sequential()  
model.add(Embedding(10000,32))  
model.add(SimpleRNN(32,return_sequences=True))  
model.add(SimpleRNN(32,return_sequences=True))  
model.add(SimpleRNN(32,return_sequences=True))  
model.add(SimpleRNN(32)) #最後一層僅傳回最後一個輸出  
model.summary()
```

#中間層需回傳完整的輸出序列資料

Out[] :

Layer (type)	Output Shape	Param #
embedding_3 (Embedding)	(None, None, 32)	320000
simple_rnn_3 (SimpleRNN)	(None, None, 32)	2080
simple_rnn_4 (SimpleRNN)	(None, None, 32)	2080
simple_rnn_5 (SimpleRNN)	(None, None, 32)	2080
simple_rnn_6 (SimpleRNN)	(None, 32)	2080

Total params: 328,320  
Trainable params: 328,320  
Non-trainable params: 0

## 6-2-1 Keras中的一個循環層



- ✓ 堆疊多個循環層於IMDB電影評論分類問題 - 準備資料

```
from keras.datasets import imdb
from keras_preprocessing.sequence import pad_sequences

max_features = 10000 #考慮做為特徵的文字數量
maxlen = 500 # 我們只看每篇評論的前 500 個文字
batch_size = 32

print('讀取資料...')
(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)
print(len(input_train), 'train sequences')
print(len(input_test), 'test sequences')
```

Out[] :

```
25000 #訓練用序列資料 (評論)
25000 #測試用序列資料
```



## 6-2-1 Keras中的一個循環層



✓ (續)堆疊多個循環層於IMDB電影評論分類問題 - 準備資料

```
print('Pad sequences (samples x time)')
```

```
input_train = pad_sequences(input_train, maxlen=maxlen) # 只看每篇評論的前 500 個文字, 多的  
              去除, 不足填補, 將序列填充轉化為固定序列長度, keras只接受長度相同的序列資料輸入
```

```
input_test = pad_sequences(input_test, maxlen=maxlen)
```

```
print('input_train shape:', input_train.shape)
```

```
print('input_test shape:', input_test.shape)
```

Out[ ] :

```
shape=(25000, 500)
```

```
shape=(25000, 500)
```

## 6-2-1 Keras中的一個循環層



- ✓ 以嵌入向量 Embedding 層和 SimpleRNN 層訓練模型

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

## 6-2-1 Keras中的一個循環層



✓ 繪製結果 - IMDB電影評論分類問題

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()

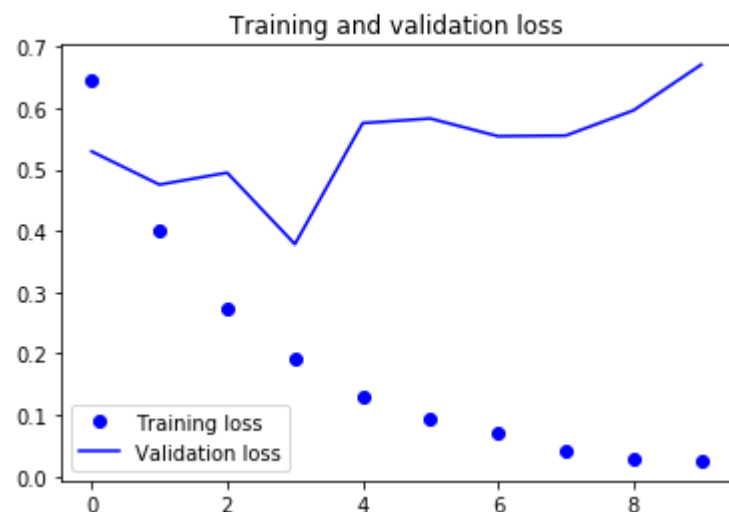
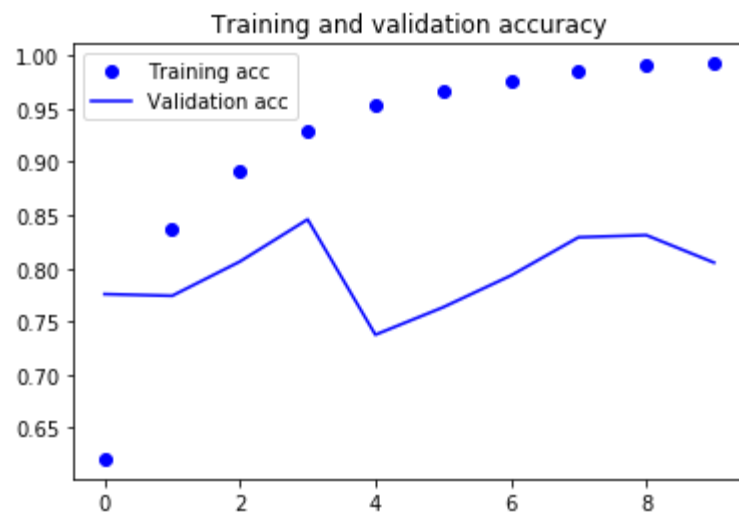
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

## 6-2-1 Keras中的一個循環層



✓ (續)繪製結果 - IMDB電影評論分類問題

Out[] :



說明：

✓ 驗證準確度差異：

- 第三章使用密集連接層後驗證準確度高達88%
- 循環神經網路驗證準確度85%

✓ 原因：

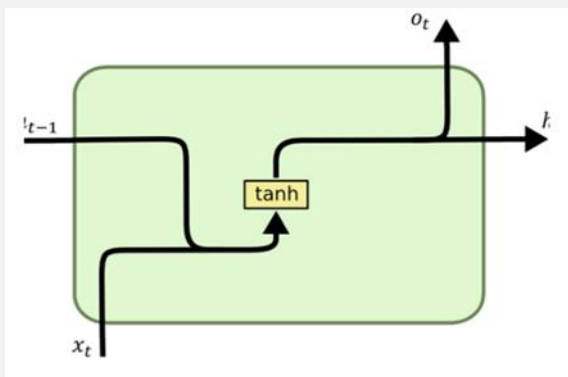
- 僅使用輸入資料前500字(非完整序列資料)
- SimpleRNN不擅長處理長型序列資料(ex: 文字資料)，資料太遠會遭遺忘

## 6-2-2 瞭解LSTM與GRU層

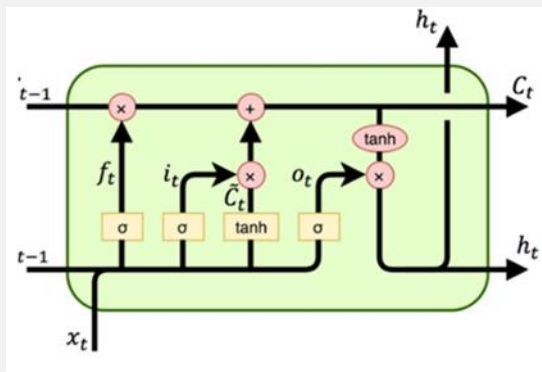


✓ Keras中的RNN循環層

### SimpleRNN



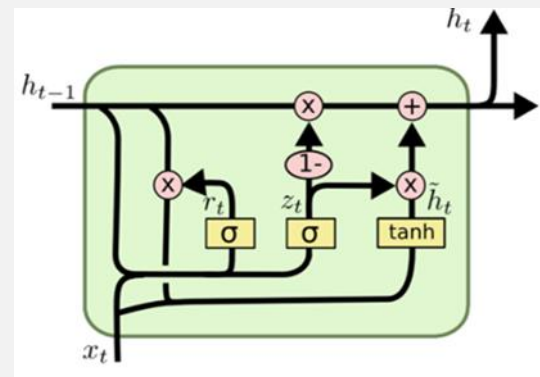
### LSTM



長期短期記憶  
(Long Short-Term Memory)

SimpleRNN層的改良版，增加跨多個時間點乘載資訊的方法，解決梯度消失問題

### GRU

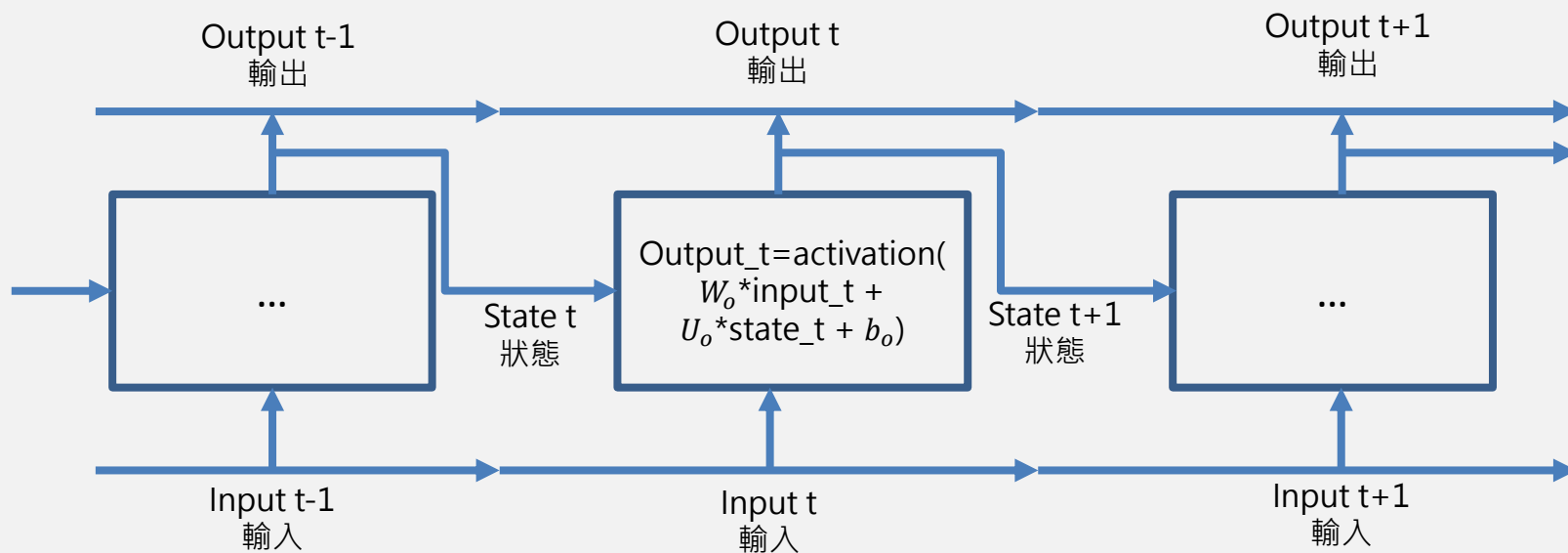


門控循環單元  
(Gated Recurrent Unit)

## 6-2-2 瞭解LSTM與GRU層



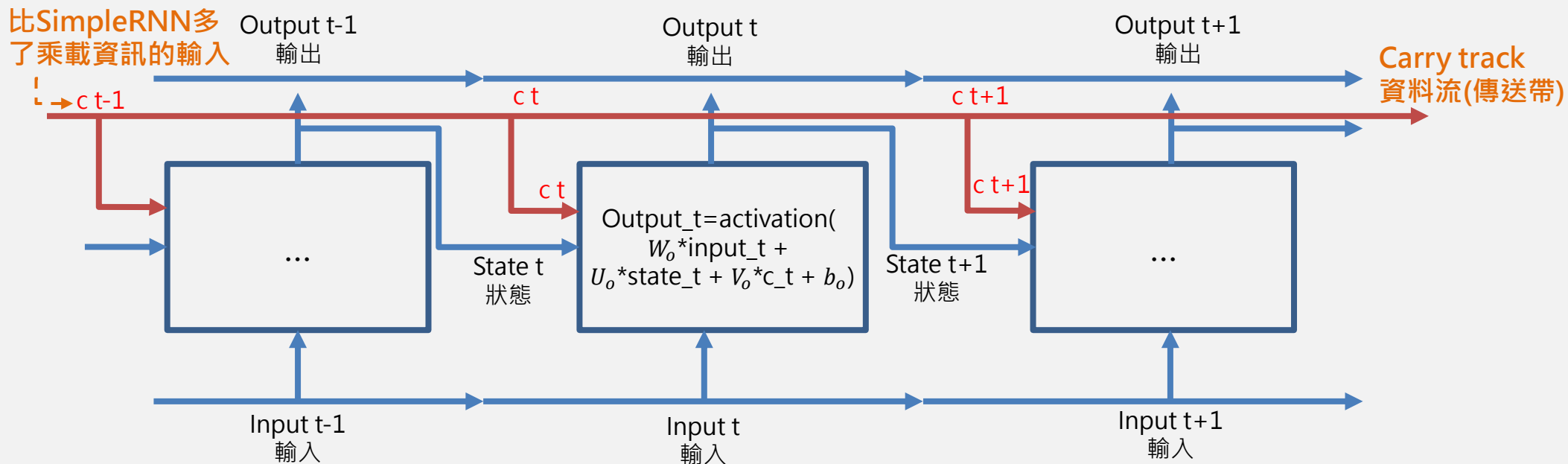
- ✓ LSTM的起點：SimpleRNN每個時間點有兩個資訊(輸入、狀態)



## 6-2-2 瞭解LSTM與GRU層



- ✓ 可在不同時間點 $t$ 取用  $C_t$  資訊
- ✓ 從SimpleRNN到LSTM
  - 增加一個承載資料軌道，每階段有三個資訊(輸入、狀態、承載資訊)



## 6-2-2 瞭解LSTM與GRU層



### ✓ 計算下一個乘載資訊流( $C_{t+1}$ )

- 轉換方式類似於SimpleRNN單元表達形式

$$y = \text{activation}(\text{dot}(\text{state\_t}, U) + \text{dot}(\text{input\_t}, W) + b) \quad \# \text{ SimpleRNN單元表達形式}$$

### ✓ LSTM 架構

- 以 $i, f, k$ 當索引

$$\text{Output\_t} = \text{activation}(\text{dot}(\text{state\_t}, U_o) + \text{dot}(\text{input\_t}, W_o) + \text{dot}(C\_t, V_o) + b_o)$$

$$i\_t = \text{activation}(\text{dot}(\text{state\_t}, U_i) + \text{dot}(\text{input\_t}, W_i) + b_i)$$

$$f\_t = \text{activation}(\text{dot}(\text{state\_t}, U_f) + \text{dot}(\text{input\_t}, W_f) + b_f)$$

$$k\_t = \text{activation}(\text{dot}(\text{state\_t}, U_k) + \text{dot}(\text{input\_t}, W_k) + b_k)$$

在時間點 $t$  計算 $i\_t$ 、 $f\_t$ 、 $k\_t$  取得新的乘載資訊狀態 $c\_t+1$

$$c\_t+1 = i\_t * k\_t + c\_t * f\_t$$

補充：

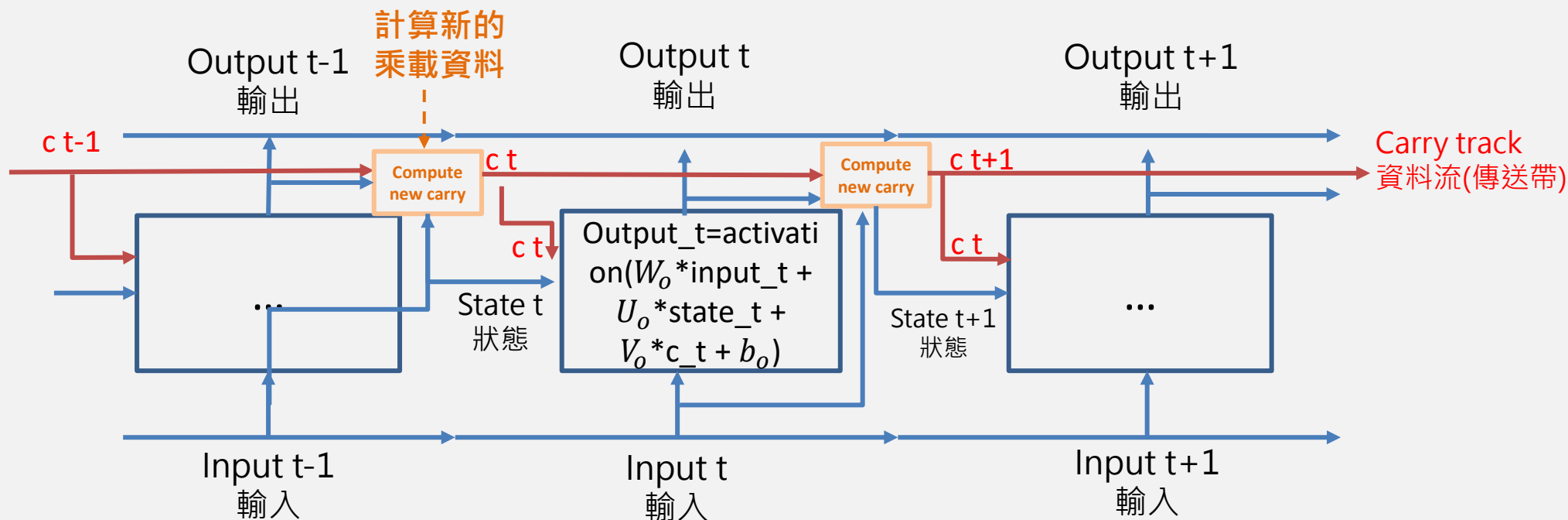
- $i\_t * k\_t$ ：提供有關當前的資訊以更新乘載資訊軌道
- $c\_t * f\_t$ ：遺忘承載資訊流中無關資訊的方法



## 6-2-2 瞭解LSTM與GRU層



✓ 將前頁描述的過程增加到圖中，即可得到LSTM模型剖析圖，如下



◆ LSTM模型允許之後重新注入過去的資訊，從而解決梯度消失的問題

## 6-2-3 以Keras實現具體的LSTM例子



- ✓ 以IMDB資料集進行訓練，在Keras中使用LSTM層

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

## 6-2-3 以Keras實現具體的LSTM例子



✓ 繪製結果 - IMDB資料集

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
```

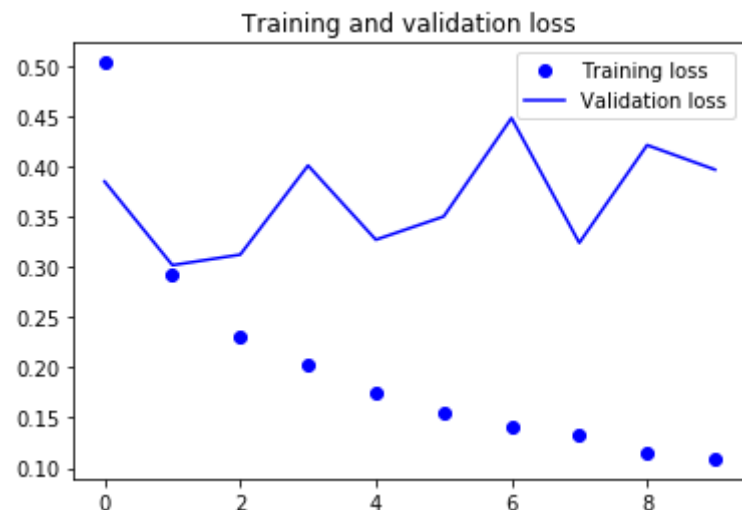
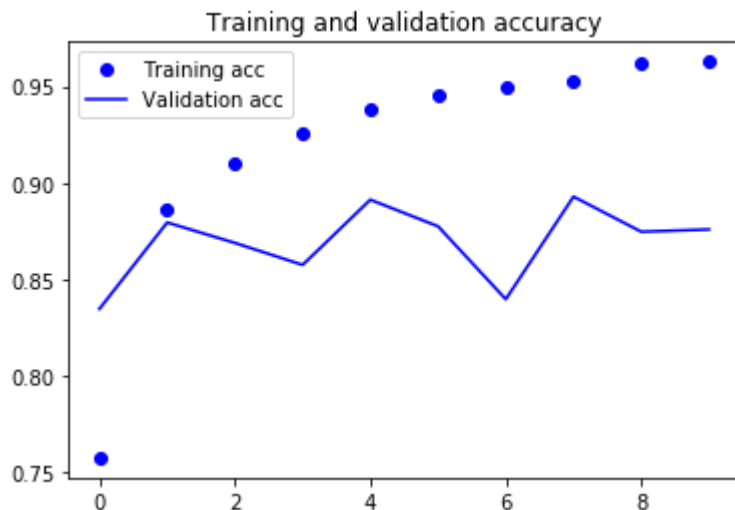
## 6-2-3 以Keras實現具體的LSTM例子



✓ (續) 繪製結果 - IMDB資料集

```
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

Out[] :

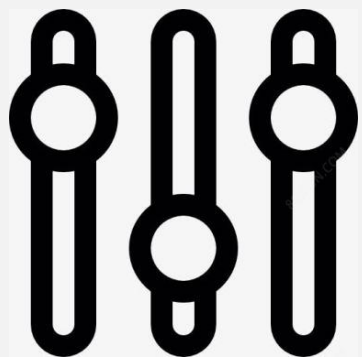


說明：LSTM模型驗證的準確度高達89%，優於SimpleRNN及全連接方法

## 6-2-3 以Keras實現具體的LSTM例子

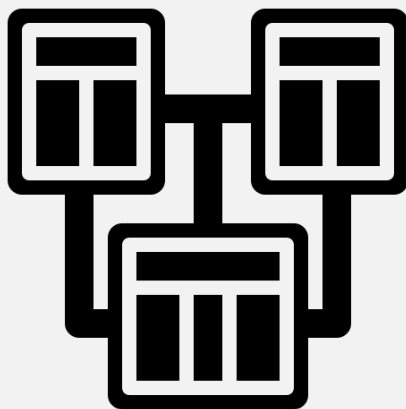


✓ 但相對於全連接方法，LSTM並不算技術突破，為什麼LSTM還不夠好？



未努力調整參數

例如：嵌入向量維度、  
LSTM輸出維度



缺乏正規化



長型架構對情感分析問題  
沒有幫助

查看每個評論中出現文字及頻率，  
密集連接層的方法仍比較適合

## 6-2-4 小結



- ✓ 此節從範例中學到
  - 什麼是RNN與其如何運作
  - 什麼是LSTM與其在長序列資料為何比simpleRNN好
  - 如何使用Keras RNN層處理序列資料
- ✓ 後續，將介紹更多RNN進階功能

## 6-3 循環神經網路的進階使用方法



- ✓ 如何改善循環神經網路性能和普適能力？
- ✓ 三項技術：
  - 循環丟棄(Recurrent dropout)
    - 用丟棄法解決過度配適的問題
  - 堆疊循環層(Stacking recurrent layers)
    - 增加神經網路的轉換表示能力
  - 雙向循環層(Bidirectional recurrent layers)
    - 以不同的方式給循環神經網路提供相同的資訊→減少遺忘問題
- ✓ 實作：溫度預測任務
  - 時間序列資料集，期間：2009~2016
  - 德國氣象站所收集的14種不同天氣數值(如：空氣溫度、氣壓、濕度、風向等)
  - 預測未來24小時的溫度

## 6-3-1 溫度預測任務



### ✓ 檢視天氣資料集的資料

```
import os
```

```
data_dir = r'C:\Users\chche' # jena_climate資料集路徑  
fname = os.path.join(data_dir, 'jena_climate_2009_2016.csv') # 資料集完整路徑  
f = open(fname) #檔案開啟  
data = f.read() #讀取資料  
f.close() #檔案關閉
```

```
lines = data.split('\n') #資料分隔  
header = lines[0].split(',') #標頭分隔  
lines = lines[1:] #標頭不讀入
```

```
print(header) #天氣資料項目  
print(len(header)) #表頭欄位數量  
print(len(lines)) #共420,551筆資料
```



## 6-3-1 溫度預測任務



✓ (續)檢視天氣資料集的資料

```
Out[]: ["Date Time", "p (mbar)", "T (degC)", "Tpot (K)", "Tdew (degC)",  
        "rh (%)", "VPmax (mbar)", "VPact (mbar)", "VPdef (mbar)", "sh (g/k  
g)", "H2OC (mmol/mol)", "rho (g/m**3)", "wv (m/s)", "max. wv (m/  
s)", "wd (deg)"]  
15  
420551
```

說明：

輸出420,551行資料，每行是一個時間點的天氣資料，其中包含一個日期紀錄和14個與天氣有關的數值

## 6-3-1 轉成Numpy並繪製氣溫圖



### ✓ 解析與處理資料

```
import numpy as np

float_data = np.zeros((len(lines), len(header) - 1)) #筆數, 天氣資料
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(',')[1:]] #日期資料不放入陣列中
    float_data[i, :] = values #[i, :]意思是[i, all columns]
print(float_data.shape)
```

Out[] : (420551, 14)

說明：

共有 420551 個時間點的天氣資料, 每個包含 14 種天氣數值

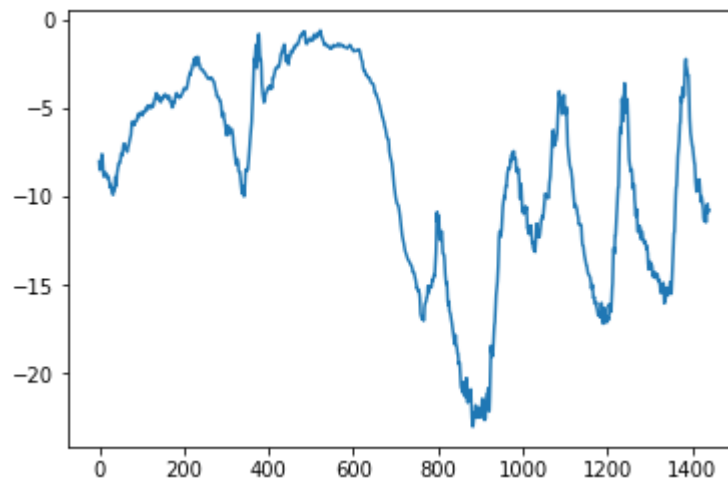
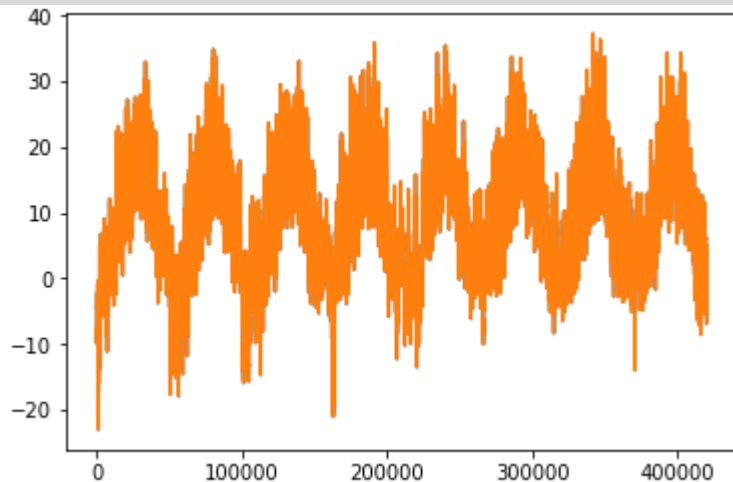
## 6-3-1 轉成Numpy並繪製氣溫圖



### ✓ 繪製氣溫圖

```
from matplotlib import pyplot as plt
temp = float_data[:, 1] # 索引 1 為溫度資料
plt.plot(range(len(temp)), temp)
plt.figure()
plt.plot(range(1440), temp[:1440]) # 10天資料, 每10分鐘記錄一次, 故共 1440個資料點
plt.show()
```

Out[] :



說明 :圖中可以清楚看到溫度的年度週期變化

## 6-3-2 準備資料



✓ 將問題以下列方式描述：

- 從過去lookback個時間點開始，以steps個時間點為間隔進行採樣，是否能預測delay個時間點的溫度？會用到以下參數：
  - ✓ Lookback = 720 → 觀察前五天的資料
  - ✓ Steps = 6 → 以每小時一個資料點進行採樣
  - ✓ Delay = 144 → 目標是未來24小時的溫度預測

✓ 開始前要先對資料進行兩件事：

- 標準化：將資料處理成神經網路可讀取的格式，獨立標準化每個時間點的序列資料，以便都可以用相似的基準取得相對小的數值
- 建立一個資料產生器：直接從原始資料即時產生訓練資料、驗證資料和測試資料。產生批次資料，因為原始資料每間隔十分鐘就會由觀測站紀錄，但以十分鐘來說，天氣的變化可能微乎其微，所以從原始的觀測資料，轉換後再作後續訓練

## 6-3-2 準備資料 – 標準化



### ✓ 標準化

- 因為溫度、氣壓...天氣數值的單位差距很大，所以需要先進行標準化，透過減去每個時間序列資料的平均值並除以標準差來預先處理資料

```
mean = float_data[:200000].mean(axis=0) #計算前200,000的資料點平均  
float_data -= mean #減去平均值  
std = float_data[:200000].std(axis=0) #計算前200,000的資料點標準差  
float_data /= std #除以標準差
```

## 6-3-2 準備資料 – 資料產生器



✓ 資料產生器 - 定義產生器函式以生產時間序列樣本資料及其目標資料

1. `def generator(data, lookback, delay, min_index, max_index, shuffle=False, batch_size=128, step=6):`
2.  `if max_index is None:`
3.  `max_index = len(data) - delay - 1`

說明：

`data`：經歷過標準化的原始浮點陣列

`lookback`：輸入資料回溯多少個時間點，本案例回溯1440點(前十天)資料

`delay`：目標溫度應該在未來多少個時間點，本案例為144點(一天後)資料

`min_index & max_index`：資料陣列中的最大和最小的索引值，用來劃分資料

`shuffle=False`：是否打亂資料

`batch_size=128`：每批次的樣本量

`step=6`：產生樣本的時間區隔，本案例為每小時

## 6-3-2 準備資料 – 資料產生器



✓ (續)資料產生器- 定義產生器函式以生產時間序列樣本資料及其目標資料

```
4.     i = min_index + lookback
5.     while 1:
6.         if shuffle:
7.             rows = np.random.randint(
8.                 min_index + lookback, max_index, size=batch_size)
9.         else:
10.            if i + batch_size >= max_index:
11.                i = min_index + lookback
12.            rows = np.arange(i, min(i + batch_size, max_index))
13.            i += len(rows)
```

## 6-3-2 準備資料 – 資料產生器



✓ (續)資料產生器- 定義產生器函式以生產時間序列樣本資料及其目標資料

```
14.     samples = np.zeros((len(rows), #訓練資料
15.         lookback // step, #一個小時取一數值
16.         data.shape[-1]))
17.     targets = np.zeros((len(rows), ))
18.     for j, row in enumerate(rows):
19.         indices = range(rows[j] - lookback, rows[j], step)
20.         samples[j] = data[indices]
21.         targets[j] = data[rows[j] + delay][1]
22.     yield samples, targets
```

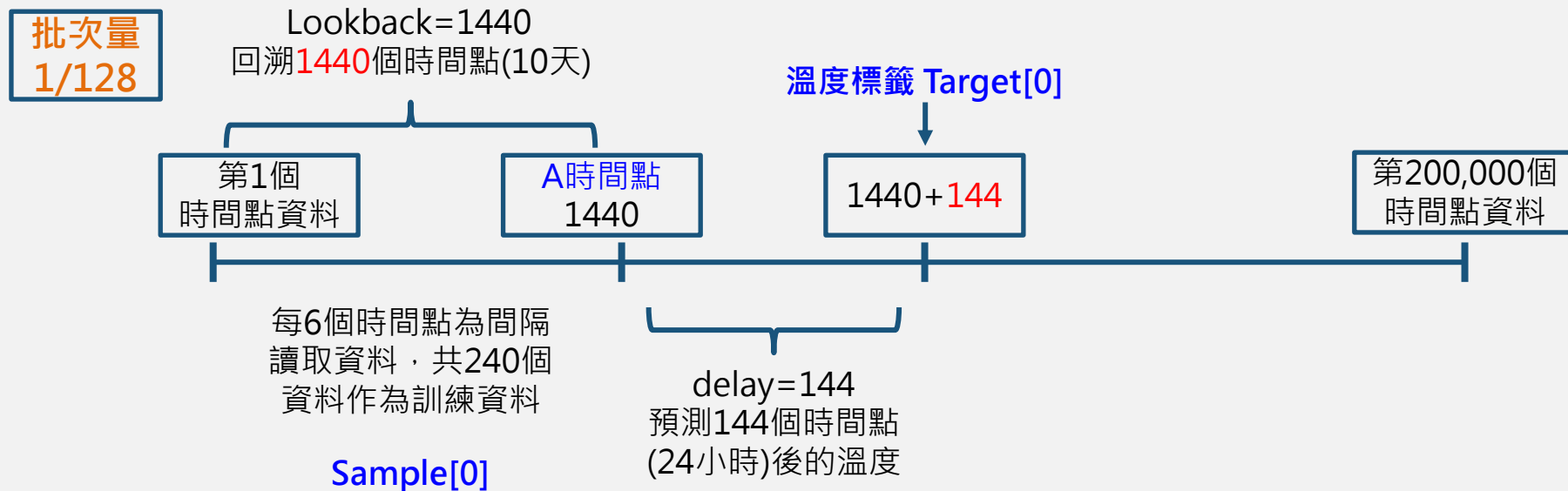


## 6-3-2 準備資料 – 補充



✓ 資料產生器的流程：

- 從A時間點往回看10天的資料，對這10天的資料進行取樣做為訓練資料，然後用A時間點後24個小時的溫度做為目標資料(標籤)
- 我們以訓練資料產生器train\_gen所產生的第一個批次量(128)資料來說明其產生過程，每個批次會產生128筆訓練資料與對應的標籤(溫度)：

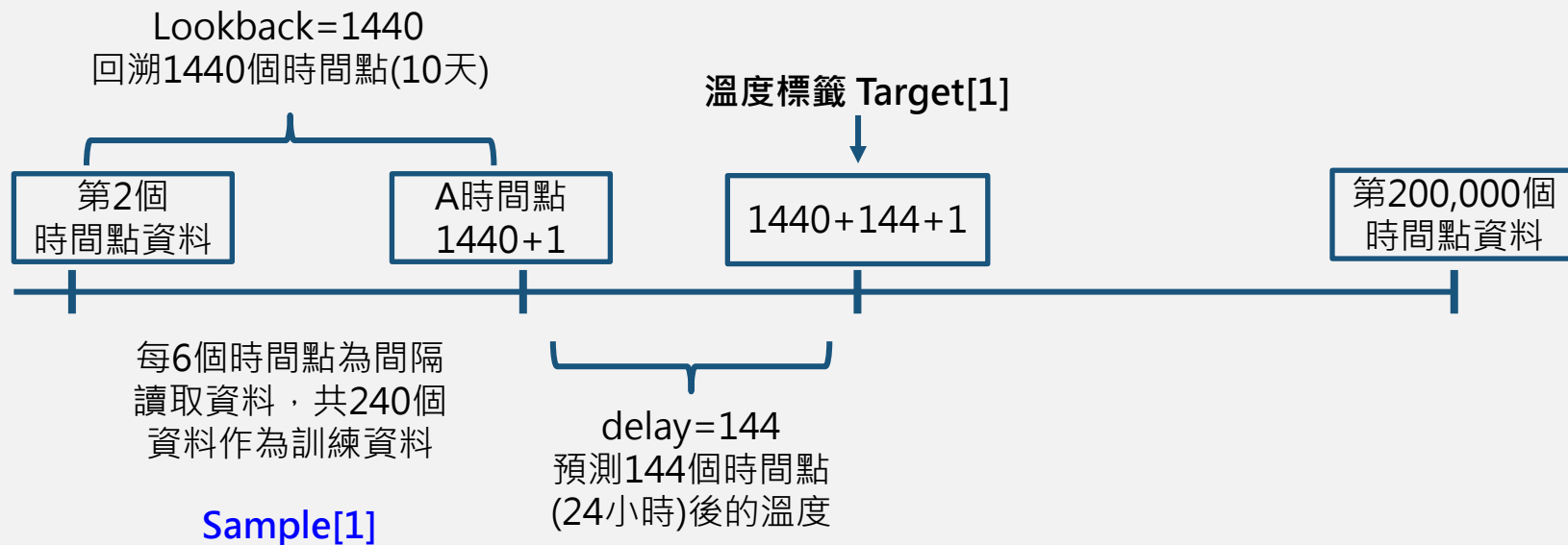


## 6-3-2 準備資料 – 補充



✓ (續)資料產生器的流程：

批次量：2/128

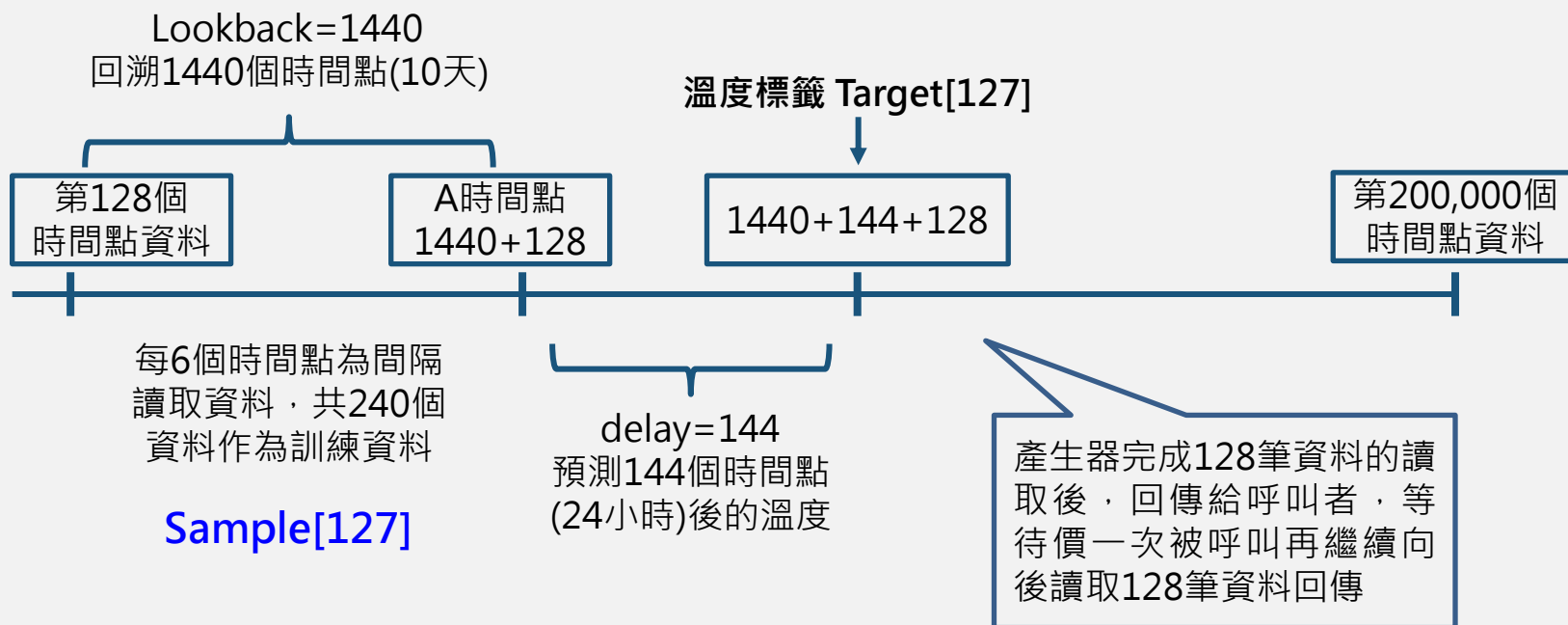


## 6-3-2 準備資料 – 補充



✓ (續)資料產生器的流程：

批次量：128/128



## 6-3-2 準備資料 – 建立資料



- ✓ 建立訓練資料、驗證資料和測試資料產生器

```
lookback = 1440
```

```
step = 6
```

```
delay = 144
```

```
batch_size = 128
```

### # 訓練資料產生器

```
train_gen = generator(float_data, lookback=lookback, delay=delay, min_index=0,  
                      max_index=200000, shuffle=True, step=step, batch_size=batch_size)
```

### # 驗證資料產生器

```
val_gen = generator(float_data, lookback=lookback, delay=delay, min_index=200001,  
                   max_index=300000, step=step, batch_size=batch_size)
```

### # 驗證資料產生器

```
val_gen = generator(float_data, lookback=lookback, delay=delay, min_index=200001,  
                   max_index=300000, step=step, batch_size=batch_size)
```

## 6-3-2 建立訓練資料、驗證資料看測試資料



✓ (續)建立訓練資料、驗證資料和測試資料產生器

### # 測試資料產生器

```
test_gen = generator(float_data,  
    lookback=lookback, #回溯多少個時間點  
    delay=delay, #目標溫度在甚麼時間點  
    min_index=300001,  
    max_index=None,  
    step=step, #產生樣本的時間區隔  
    batch_size=batch_size) #每批次的樣本量
```

```
val_steps = (300000 - 200001 - lookback) // batch_size #← 1...
```

```
test_steps = (len(float_data) - 300001 - lookback) // batch_size #← 2..
```

#1. val\_steps 產生器需要運行多少次才可以產生完整的驗證集

#2. test\_steps 產生器需要運行多少次才可以產生完整的測試集

### 6-3-3 一種基於常識性、非機器學習的基準方法



- ✓ 當面對一個尚有解決方案的新問題時，一些常識性的基準方法會很有用
- ✓ 例如：在一個不均衡的分類任務
  - 即其中一些類別比其他類別多的情況，方法就是在新樣本出現時，就直接預測為A類，因為這樣會有90%的機率會猜對，總體上會90%的精準度
  - 因此任何基於機器學習的方法都應該超過90%的分數以證明方法是有用的(但在某些特殊情況下，這些基準方法被證明仍難以擊敗)

#### 不均衡的分類任務



A類

B類

○ 哪一類？

● 答A類，會有90%的機率答對！

機器學習的模型應該擊敗這個基準方法

## 6-3-3 一種基於常識性、非機器學習的基準方法



- ✓ 在溫度預測的資料集中，一般常識性基準方法可以使用平均絕對誤差MAE

```
def evaluate_naive_method(): #評估程序
    batch_maes = []
    for step in range(val_steps): # 計算所有的驗證集資料
        samples, targets = next(val_gen) # 驅動產生器,用next()驅動val_gen產生器,
                                          輸出一個批次量的資料與標籤
        print(samples.shape) # shape=(128, 240, 14), 因為回朔為 1440 個時間點, 並以 6 個時間點為間隔
                              進行取樣,所以共產生 1440/6=240 個時間點資料
        print(targets.shape) # shape=(128,) 128 筆溫度答案
        preds = samples[:, -1, 1]
        mae = np.mean(np.abs(preds - targets)) #mae平均絕對誤差
        batch_maes.append(mae)
    print(np.mean(batch_maes))
```

```
evaluate_naive_method()
```

```
Out[] : celsius_mae = 0.29 * std[1]
```

說明：基準方法的結果為 $0.29 \times \text{溫度標準差} \approx 2.57^\circ\text{C}$

## 6-3-4 基本的機器學習方法



- ✓ 在研究複雜且計算成本高昂的模型前 → 先嘗試小型密集連接神經網路
  - 使用簡單、密集連接神經網路，訓練與評估密集連接模型

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
```

```
model = Sequential()
model.add(layers.Flatten(input_shape=(lookback // step, float_data.shape[-1]))) #2400(小時), 14
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))
```

```
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen, steps_per_epoch=500,
                             epochs=20, validation_data=val_gen, validation_steps=val_steps)
```



## 6-3-4 基本的機器學習方法



✓ 繪製結果

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

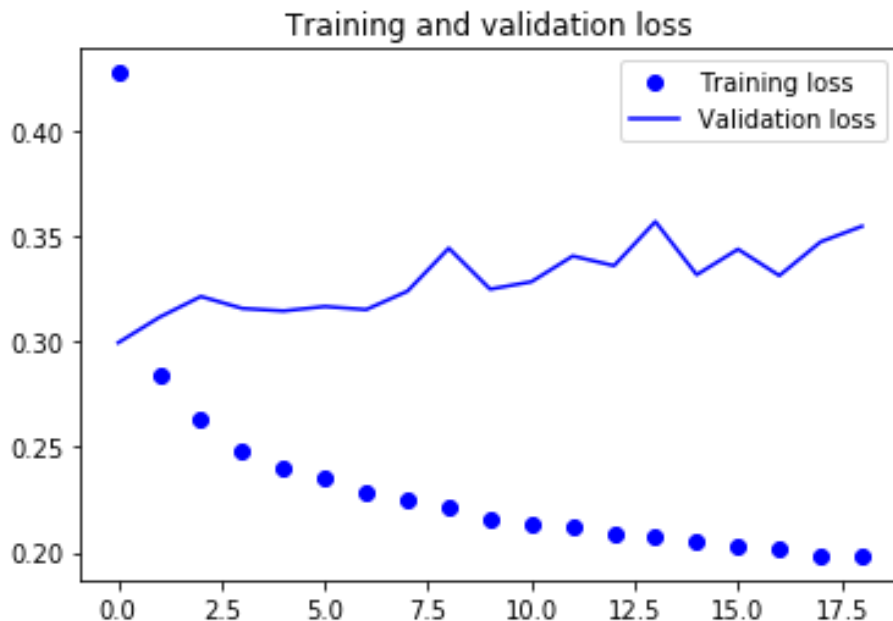
plt.show()
```

## 6-3-4 基本的機器學習方法



✓ (續)繪製結果

Out[] :




說明：損失大於0.29，大部分驗證損失都比非機器學習的基準方法結果0.29還差，表示這個模型的結果並不够好，代表我們的一般常識與認知包含許多機器學習模型沒有的高價值資訊

## 6-3-5 第一個循環基準方法



- ✓ 雖完全連接方法的成效不好，但不代表機器學習不適用於這個問題，之前的方法是先展平時間序列資料，這樣將從輸入資料中刪除了時間概念
- ✓ 但如果是序列資料，因果關係和順序都很重要，譬如：

序列不一樣，意思截然不同

我愛你  你愛我

- ✓ 現在來嘗試循環序列資料處理模型：
  - GRU (Gated Recurrent Unit) 閘控循環單元神經網路
  - 與LSTM的原理相同，但經過些許的簡化，使執行成本較低，與完全連接方法不同的點是GRU利用了資料點的時間順序屬性

## 6-3-5 第一個循環基準方法



- ✓ GRU (Gated Recurrent Unit) 閘控循環單元神經網路
- ✓ 模型的訓練與驗證

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
```

```
model = Sequential()
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
#None是為了要留給batch_size=200000/500=400，float_data.shape會等於(420551,14)，
float_shape.shape[-1]就是抓14
model.add(layers.Dense(1))
```

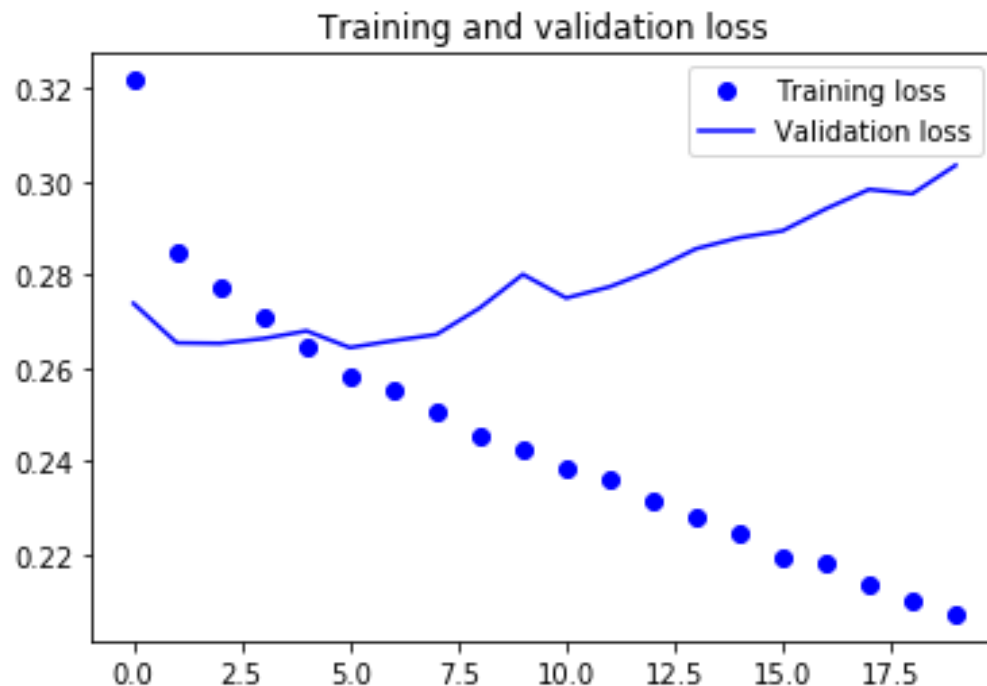
```
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen, steps_per_epoch=500, epochs=20,
                             validation_data=val_gen, validation_steps=val_steps)
```

## 6-3-5 第一個循環基準方法



✓ 繪製結果

Out[] :



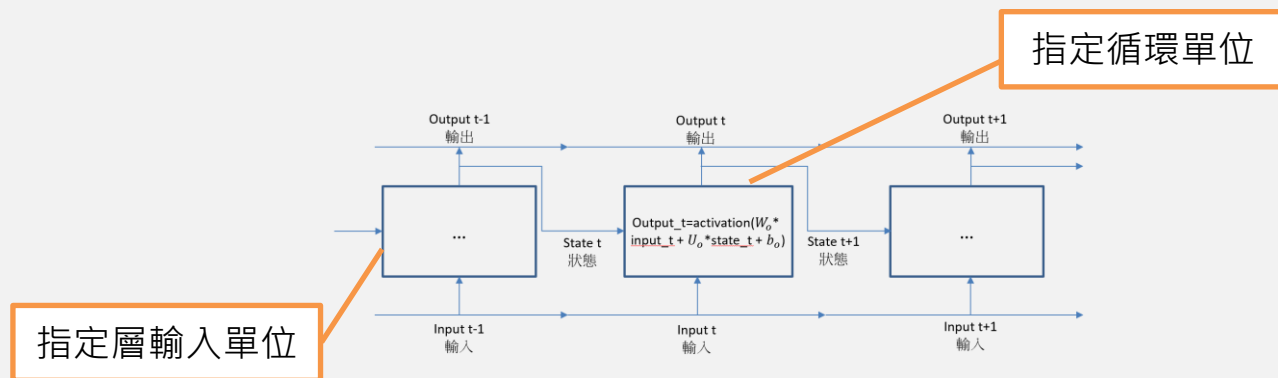
說明：新的驗證損失MAE約為 $0.265 \times \text{溫度標準差} = 2.35^\circ\text{C}$ ，比初始誤差 $2.57^\circ\text{C}$ 好，顯示出循環神經網路優於序列資料扁平化密集神經網路(0.29)，但仍有進步空間

## 6-3-6 使用循環丟棄法來降低過度配適



### ✓ 循環神經丟棄法

- 每個步驟中，使用相同的遮罩(相同的剔除單元pattern)，不可以隨機性的丟棄遮罩，這樣會破壞此誤差訊號的傳遞並對學習過程造成傷害
- Keras中的每個循環層都有兩個丟棄法相關的參數
  - ✓ dropout：一個指定輸入單元的丟棄率
  - ✓ recurrent\_dropout：指定循環單位的丟棄率



- ✓ 注意：透過丟棄法常規化神經網路需要更長的時間才能完全收斂，所以神經網路的訓練週期(次數)需要原來的兩倍

## 6-3-6 使用循環丟棄法來降低過度配適



### ✓ 訓練和驗證使用丟棄法常規化的GRU模型

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

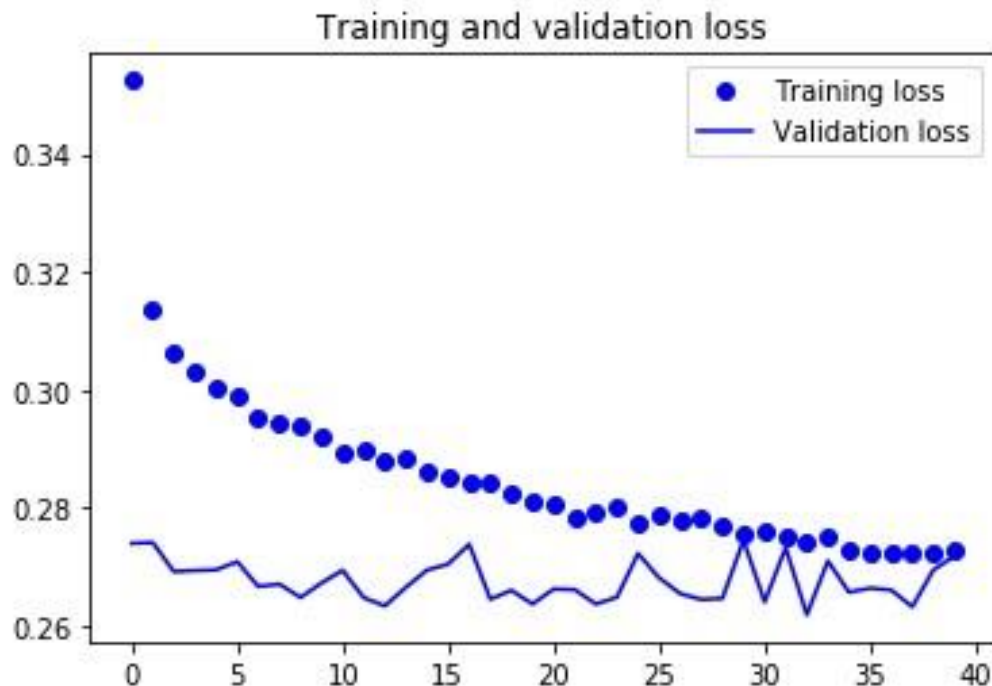
model = Sequential()
model.add(layers.GRU(32,
                    dropout=0.2, #指定層輸入單位的丟棄率
                    recurrent_dropout=0.2, #指定循環單位的丟棄率
                    input_shape=(None, float_data.shape[-1])))
#None是為要留給batch_size，float_data.shape[-1]就是抓14
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                             steps_per_epoch=500,
                             epochs=40,
                             validation_data=val_gen, validation_steps=val_steps)
```

## 6-3-6 使用循環丟棄法來降低過度配適



✓ (續)訓練和驗證使用丟棄法常規化的GRU模型

Out[] :



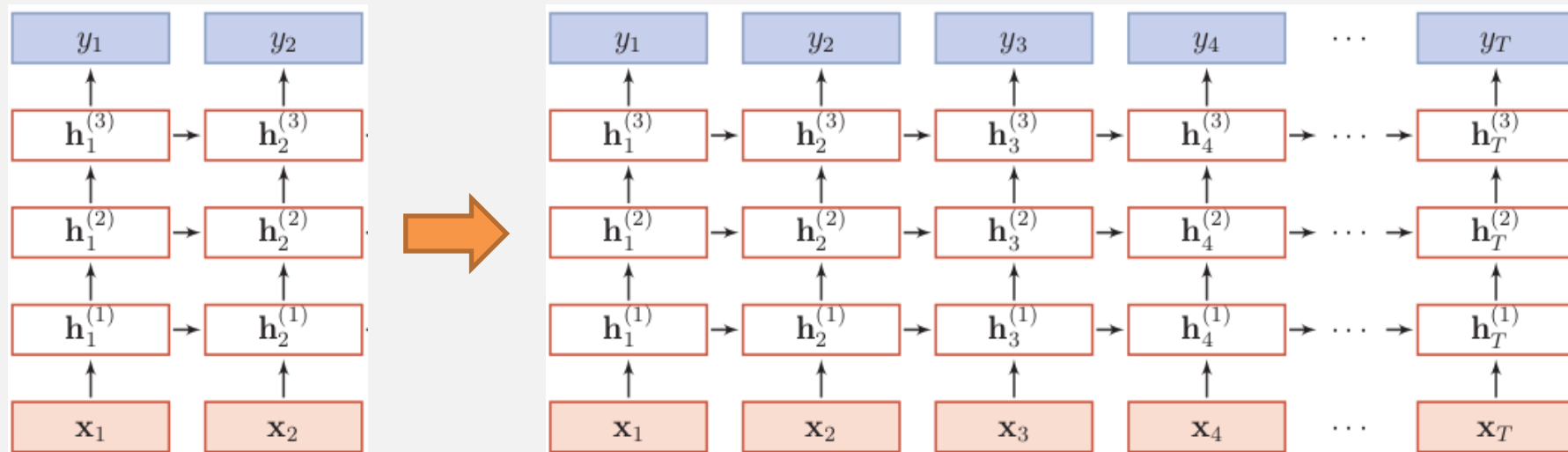
說明：訓練成功！在前30個訓練週期，不再過度配適，但是儘管驗證損失更穩定，但最佳損失沒有比以前的結果降低很多



## 6-3-7 堆疊循環層



- ✓ 模型雖不再過度配適，但遇到性能瓶頸 → 可考慮增加神經網路容量
- 可增加層中的單元數或增加更多層來提升神經網路容量
  - 在Keras中將循環層堆疊在原始架構上，故所有中間層應輸出其完整的輸出序列資料(3D張量) → 透過`return_sequences=True`來達成



原始

增加神經網路容量

## 6-3-7 堆疊循環層



- ✓ 訓練和驗證一個使用丟棄法的堆疊GRU模型

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                    dropout=0.1, #指定層輸入單位的丟棄率
                    recurrent_dropout=0.5, #指定循環單位的丟棄率
                    return_sequences=True, #中間層應輸出完整的序列資料(3D張量)
                    input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
                    dropout=0.1,
                    recurrent_dropout=0.5))
```

## 6-3-7 堆疊循環層



✓ (續)訓練和驗證一個使用丟棄法的堆疊GRU模型

```
model.add(layers.Dense(1))

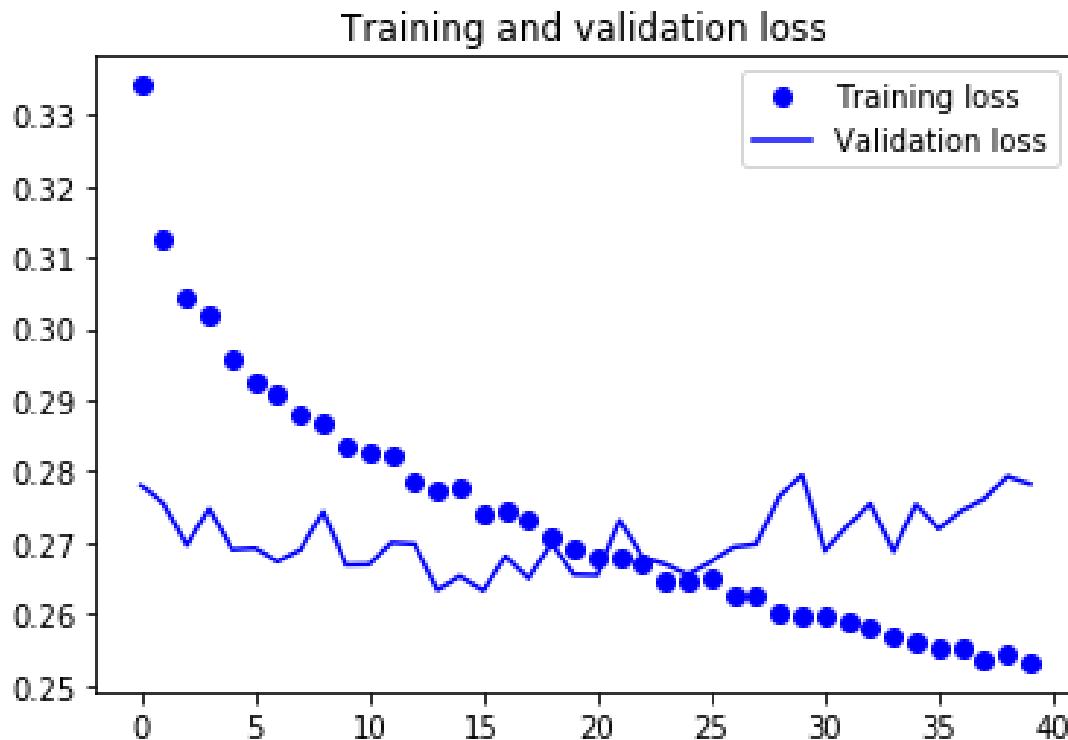
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

## 6-3-7 堆疊循環層



✓ (續)訓練和驗證一個使用丟棄法的堆疊GRU模型

Out[ ] :

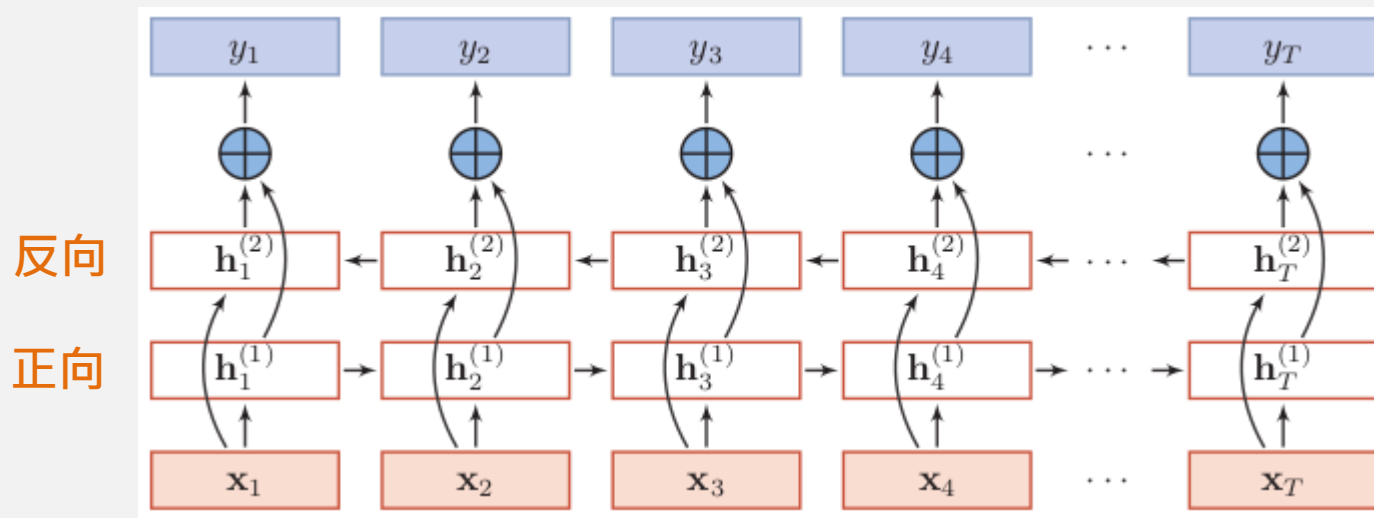


說明：增加一層後，仍然沒有太嚴重的過度配適問題，所以可以再增加層數，改善模型

## 6-3-8 使用雙向RNN



- ✓ 雙向RNN(bidirectional RNN)是RNN變體，某些特定任務上的表現會優於RNN
  - 例如：在自然語言處理方面可以稱為深度學習對於自然語言處理的瑞士刀！



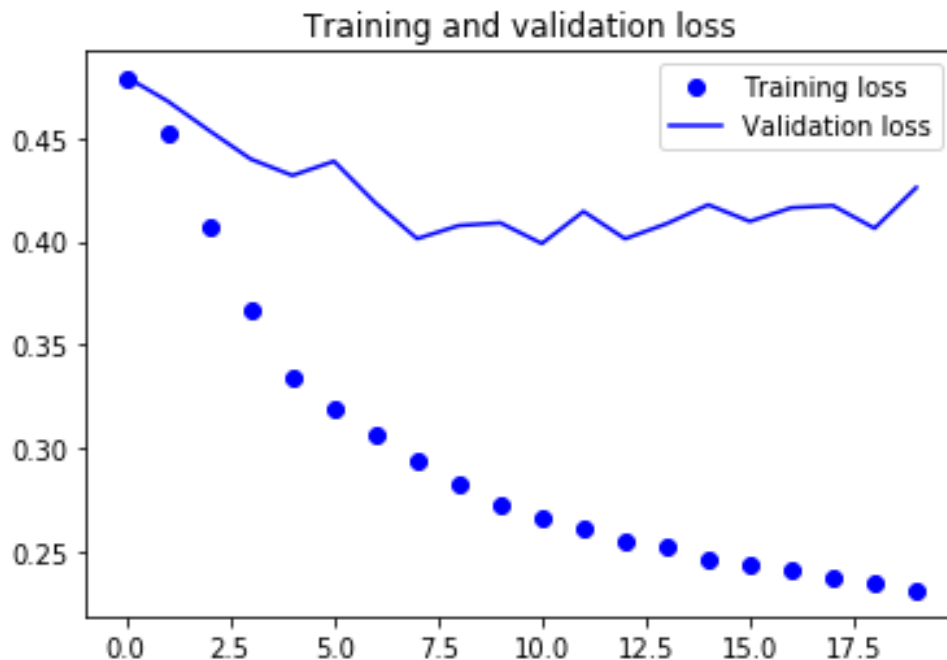
- ✓ 特性：資料順序性或時間性的處理，按順序處理輸入序列資料的時間點
- ✓ 適合：適合資料有順序意義的問題，如：溫度預測的問題
- ✓ 差異：可以取得單向RNN可能忽略的patterns

## 6-3-8 使用雙向RNN



✓ GRU使用反向序列資料訓練和驗證溫度預測任務

Out[] :



說明：針對此任務，正向序列版本優於反向順序版本，但對其他問題來說，可能不一樣

## 6-3-8 使用雙向RNN



### ✓ 使用反向序列資料訓練和驗證LSTM

```
from keras.datasets import imdb
from keras_preprocessing.sequence import pad_sequences
from keras import layers
from keras.models import Sequential
```

```
max_features = 10000 # 考慮作為特徵的文字數量
maxlen = 500 # 只看每篇評論的前 500 個字
```

```
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features) # 載入資料
x_train = [x[::-1] for x in x_train] # 將訓練資料進行反向順序排列
x_test = [x[::-1] for x in x_test] # 將測試資料進行反向順序排列
x_train = pad_sequences(x_train, maxlen=maxlen) # ← 填補序列資料
x_test = pad_sequences(x_test, maxlen=maxlen)
```

## 6-3-8 使用雙向RNN



✓ (續)使用反向序列資料訓練和驗證LSTM

```
model = Sequential()
model.add(layers.Embedding(max_features, 128))
model.add(layers.LSTM(32))
model.add(layers.Dense(1, activation='sigmoid'))
```

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

```
history = model.fit(x_train, y_train, #fit是一次載入所有的訓練集
                   epochs=10,
                   batch_size=128,
                   validation_split=0.2) #抓20%作為驗證集
```

結果正反向的成效幾乎一樣，證明雖然字詞的順序在理解語言方面很重要，但使用的順序並不重要

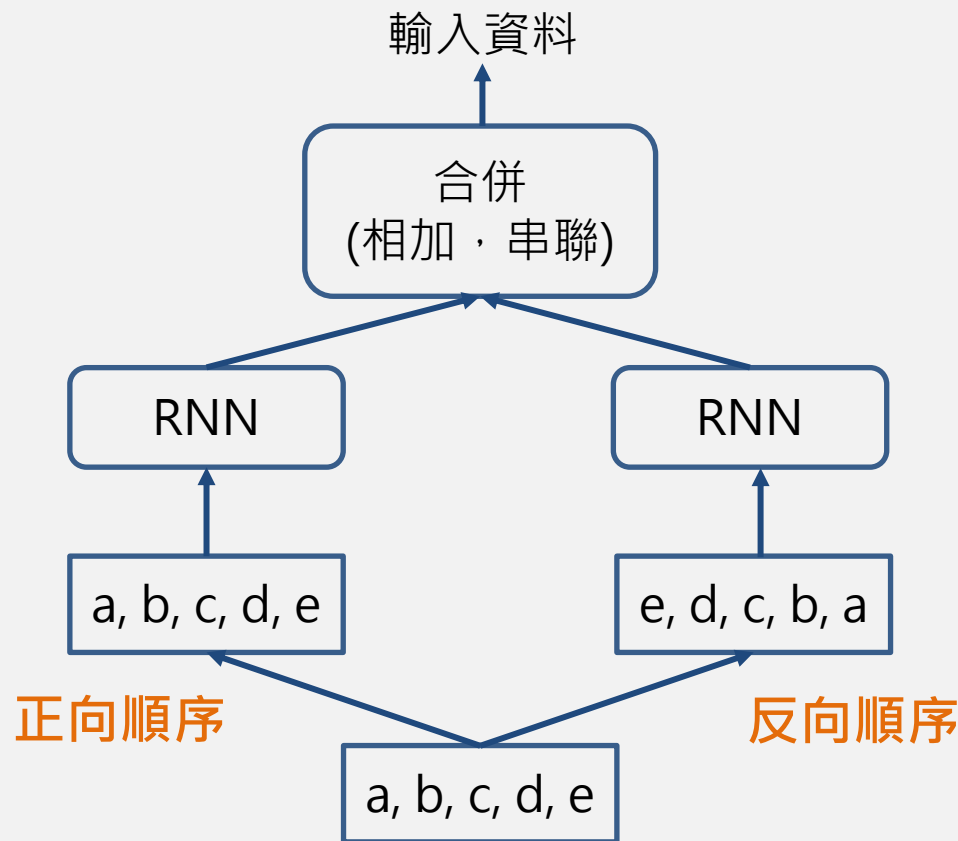


## 6-3-8 使用雙向RNN



### ✓ 雙向RNN工作原理

- 該層的第一個參數是個循環層物件，緊接著Bidirectional雙向層會自己建立第二個單獨的循環層物件
- 然後其中一個循環層按時間順序輸入序列資料，另一個循環層則以反向順序處理輸入序列資料
- 最後串接在一起



## 6-3-8 使用雙向RNN



### ✓ 訓練和驗證雙向LSTM

```
model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

## 6-3-8 使用雙向RNN



### ✓ (續)訓練和驗證雙向LSTM

Out[] :

Layer (type)	Output Shape	Param #
=====		
embedding_2 (Embedding)	(None, None, 32)	320000
=====		
bidirectional (Bidirectional)	(None, 64)	16640
=====		
dense_4 (Dense)	(None, 1)	65
=====		
Total params: 336,705		
Trainable params: 336,705		
Non-trainable params: 0		

說明：

✓  $32 * \text{max\_feature} = 320000$

✓  $2 * 4 * [(\text{numHiddenUnit} + \text{inputSize}) * \text{numHiddenUnits} + \text{numHiddenUnits}]$   
 $= 2 * 4 * ((32 + 32) * 32 + 32) = 16640$

## 6-3-8 使用雙向RNN



### ✓ 訓練雙向GRU進行溫度預測任務

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.Bidirectional(layers.GRU(32), input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen, steps_per_epoch=500,
                              epochs=40, validation_data=val_gen,
                              validation_steps=val_steps)
```

說明：成效幾乎與單向GRU一致，因為所有的預測能力都來自正向的資料，我們知道反向資料在此任務上表現不佳，也就是對預測溫度而言，最近的歷史資料比較遠的歷史資料更為重要

## 6-3-9 更進一步



✓ 還可以嘗試其他方法，來提高溫度預測問題的成效：

- 調整堆疊中每個循環層的單位數
- 調整RMSprop優化器使用的學習率
- 使用LSTM而不是GRU→模型表現能力較佳，但計算成本較高
- 更大的密集連接層或堆疊更多的密集連接層

## 6-3-10 小結



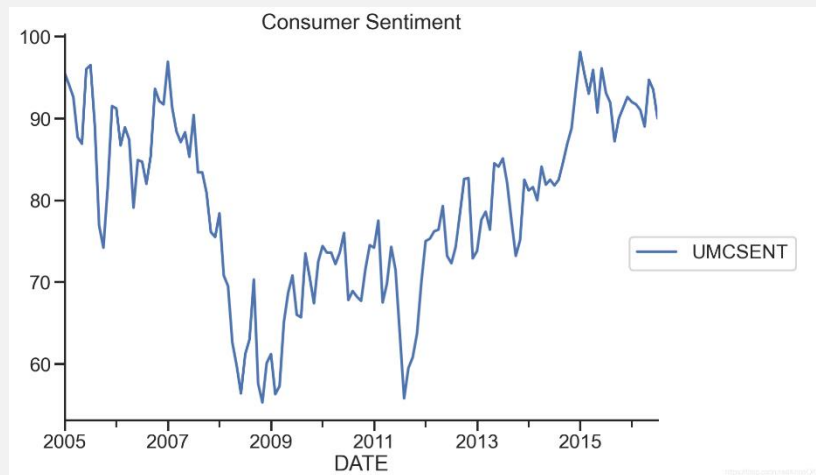
✓ 目前我們學習到：

- 當資料有**順序性**時→**循環神經網路**會比展平時間順序的模型**更合適**
- 循環神經網路如**使用丟棄法**→**時間一致丟棄**和**循環丟棄遮罩**，都內建於Keras
- **堆疊RNN**比單一RNN提供更強的**轉換表示能力**，可以在複雜問題(如機器翻譯)上提供明顯效益
- **雙向RNN**以兩種方向檢視序列資料，對**自然語言問題處理**很有用
- **雙向RNN**對時間序列問題幫助有限(**反向檢視時間序列問題沒太多幫助**)

## 6-4 使用卷積神經網路進行序列資料處理

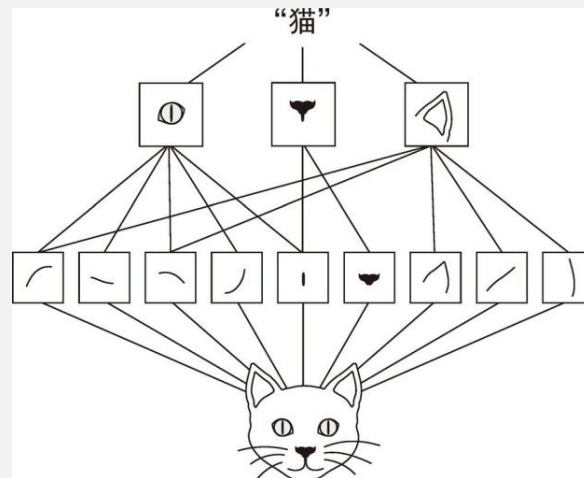


### ✓ 卷積神經網路



序列資料處理問題

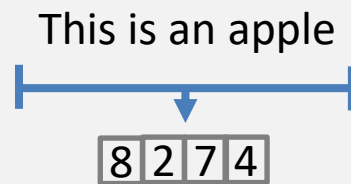
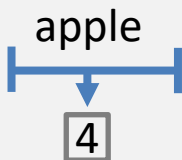
時間軸可被視為空間維度



電腦視覺

於局部輸入區塊中萃取特徵

### ✓ 序列資料類比圖片範例：



用特定整數代表一個文字，  
將**整數**視為**圖片像素值**

## 6-4 使用卷積神經網路進行序列資料處理



- ✓ 面對序列資料時，亦可使用1D卷積神經網路進行處理
  - 相較於RNN計算成本更低，最近與擴充內核一起使用的1D卷積神經網路已經成功用於：語音生成、機器翻譯
  - 除了這些案例外，也可針對文字資料分類、時間序列資料預測的簡單任務，用1D卷積神經網路取代RNN



## 6-4-1 瞭解序列資料的1D神經網路

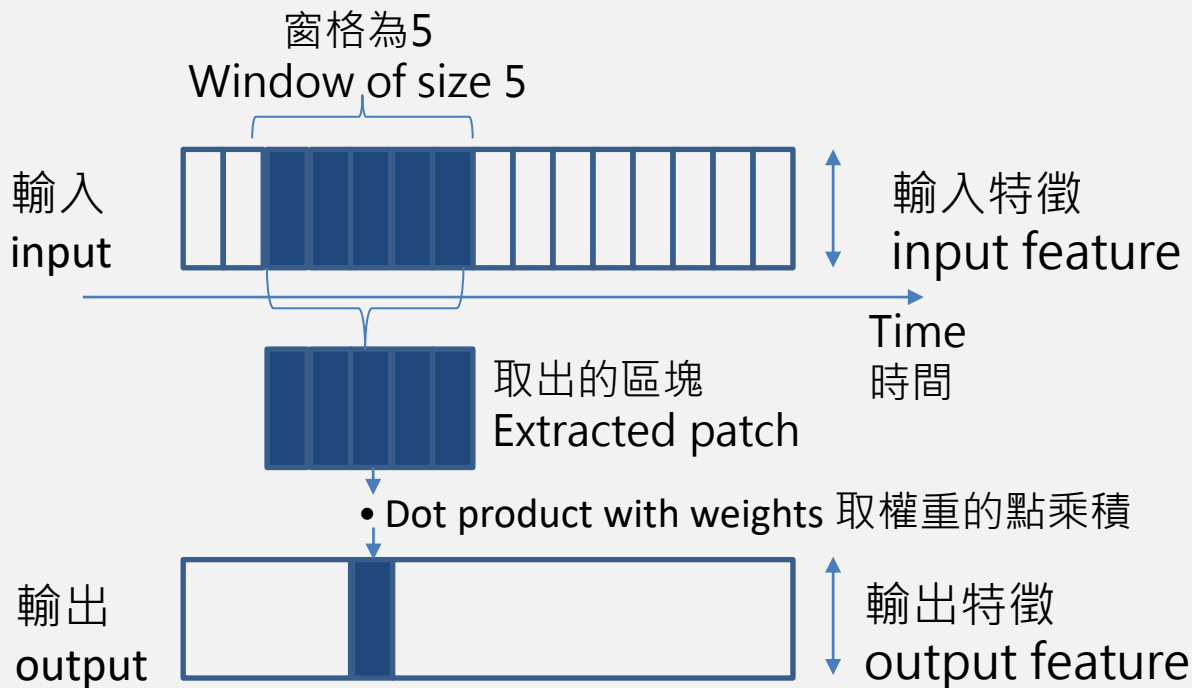


✓ 1D卷積層可識別序列資料中局部pattern

- Pattern平移不變性

- ✓ 在句中學到的pattern可在不同位置被識別

- 範例：以卷積窗格大小為5處理一維序列資料(用點乘積進行資料轉換)

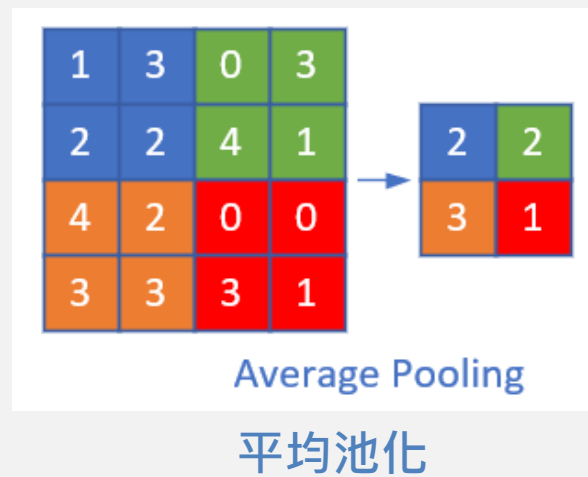


## 6-4-2 用於序列資料的1D池化方法



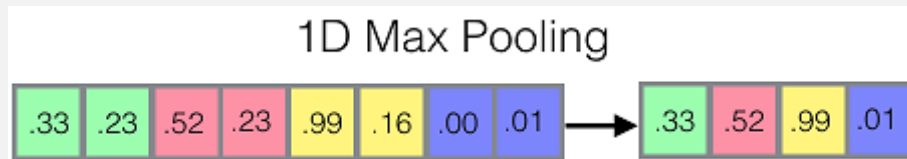
✓ **2D池化方法**：對神經網路中的圖片張量進行採樣

- 最大池化、平均池化



✓ **1D池化方法**

- 原理：與2D池化方法相同



- 目的：在神經網路中**減少**1D輸入的**長度**(子序列資料的採樣)

## 6-4-3 實作1D卷積神經網路



### ✓ 準備IMDB資料

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

max_features = 10000 #考慮做為特徵的文字數量
max_len = 500 # 我們只看每篇評論的前 500 個文字
batch_size = 32

print('Loading data... ')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences') # 25000 筆訓練用序列資料 (評論)
print(len(x_test), 'test sequences') # 25000 筆測試用序列資料
```

## 6-4-3 實作1D卷積神經網路



✓ (續)準備IMDB資料

```
print('Pad sequences (samples x time)')  
# 只看每篇評論的前 500 個文字, 多的去除, 不足填補  
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) # shape=(25000, 500)  
print('x_test shape:', x_test.shape)  # shape=(25000, 500)
```

Out[ ] :

```
25000 train sequences  
25000 test sequences  
Pad sequences (samples x time)  
x_train shape: (25000, 500)  
x_test shape: (25000, 500)
```

說明：完成資料預處理

## 6-4-3 實作1D卷積神經網路



✓ 以IMDB資料訓練和驗證簡單的1D卷積神經網路

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len))
#Token數量,維度,文字長度
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D()) #全域最大池化層
#對張量中第一軸的整個數據求一個最大值
model.add(layers.Dense(1))
model.summary()
```

## 6-4-3 實作1D卷積神經網路



✓ (續)以IMDB資料訓練和驗證簡單的1D卷積神經網路

```
model.compile(optimizer=RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                   epochs=10,
                   batch_size=128,
                   validation_split=0.2)
```

Out[] : Model: "sequential\_1"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 500, 128)	1280000
conv1d (Conv1D)	(None, 494, 32)	28704
max_pooling1d (MaxPooling1D)	(None, 98, 32)	0
conv1d_1 (Conv1D)	(None, 92, 32)	7200
global_max_pooling1d (Global	(None, 32)	0
dense (Dense)	(None, 1)	33

Total params: 1,315,937  
Trainable params: 1,315,937  
Non-trainable params: 0

## 6-4-3 實作1D卷積神經網路



### ✓ 繪製結果

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()

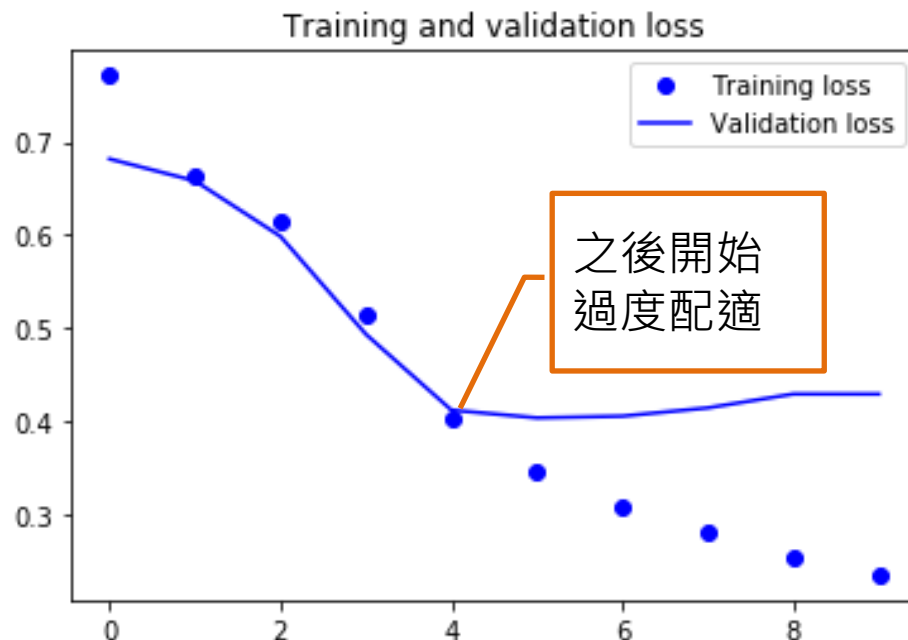
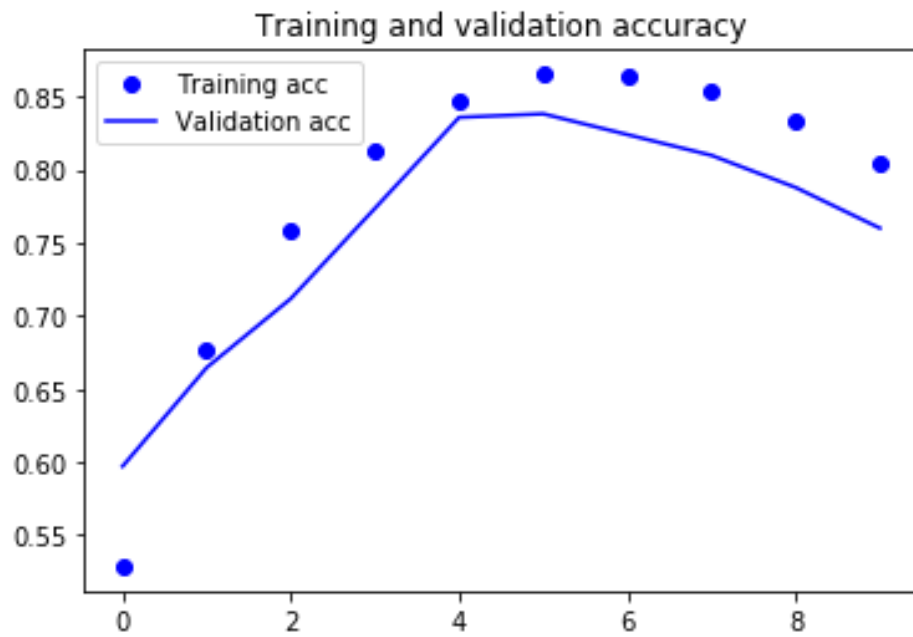
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

## 6-4-3 實作1D卷積神經網路



✓ (續)繪製結果

Out[ ] :



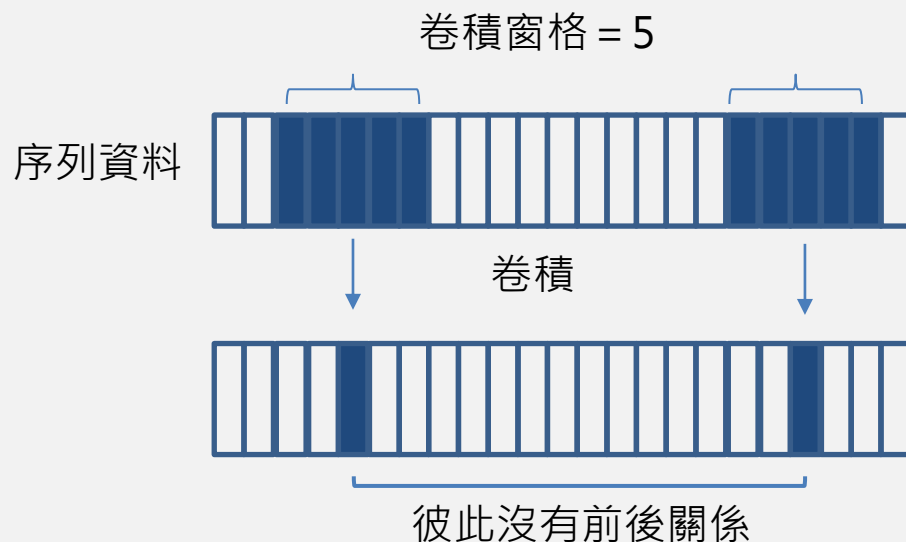


## 6-4-4 結合CNN與RNN處理長序列資料



### ✓ 1D卷積神經網路的問題

- 獨立處理輸入區塊，故對時間點的順序(超過卷積窗格大小)不敏感，與RNN不同
- 為了識別較長的出現patterns，我們可以堆疊許多卷積層和池化層，使神經網路上層可以看到原始輸入中更長的區塊，但對具順序靈敏度的資料來說，仍無太大的幫助
- 對資料中的局部區塊進行特徵萃取，但局部之外就較難產生關聯，無法知道前後關聯



故可以結合CNN與  
RNN處理長序列資料!

## 6-4-4 結合CNN與RNN處理長序列資料



- ✓ 以預測溫度的資料訓練和驗證一個簡單的1D卷積神經網路

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu', input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')

history = model.fit_generator(train_gen, steps_per_epoch=500, epochs=20,
                             validation_data=val_gen, validation_steps=val_steps)
```

## 6-4-4 結合CNN與RNN處理長序列資料



✓ 繪製結果

```
import matplotlib.pyplot as plt
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
```

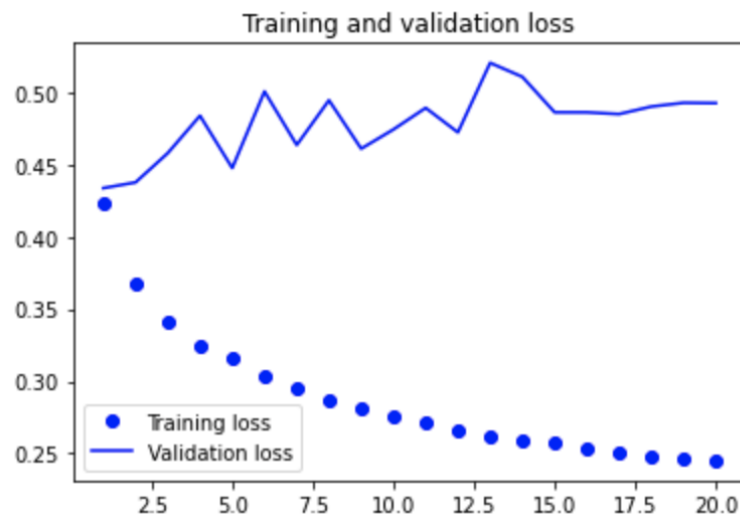
## 6-4-4 結合CNN與RNN處理長序列資料



✓ (續)繪製結果

```
plt.legend()  
plt.show()
```

Out[] :



說明：驗證損失保持在0.47左右，代表我們無法使用這小型卷積神經網路擊敗一般常識基準方法(0.29)

## 6-4-4 結合CNN與RNN處理長序列資料

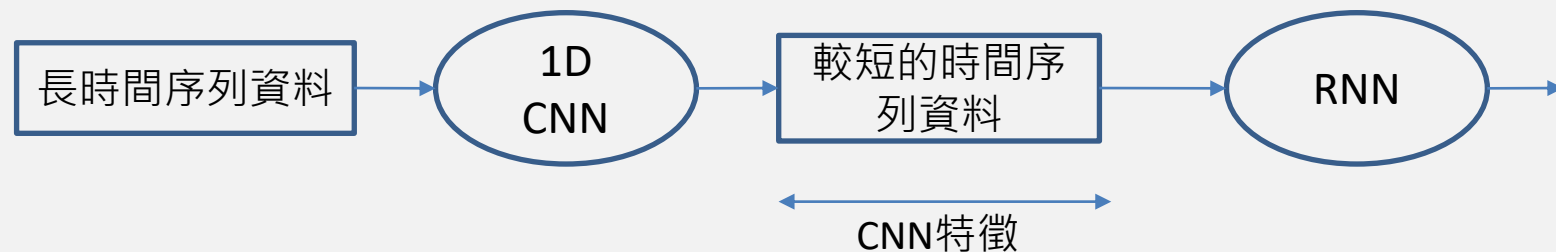


### ✓ 為什麼案例中卷積神經網路無法擊敗一般常識基準方法？

- 卷積神經網路在輸入序列資料中檢視pattern出現，卻不知其出現位置(開頭or結尾)
- 但如果放在IMDB電影評論資料集則不受影響，因為影響正負評的因素在於是否出現某些關鍵字(出現在哪個位置不影響)

### ✓ 卷積神經網路CNN+RNN

- 1D卷積神經網路作為RNN前的預處理步驟
- 例如：CNN將長時間序列資料轉換為較短的高階特徵序列資料，這些萃取資料再作為RNN的輸入



結合1D卷積神經網路和RNN以處理長時間序列資料

## 6-4-4 結合CNN與RNN處理長序列資料



✓ 為溫度預測數據集準備高解析率的資料產生器

```
step = 3 # 先前設定為 6(每小時 1 個時間點), 現在設定為 3(每 30 分鐘 1 個時間點)
lookback = 720 # 觀察前5天的資料
delay = 144 # 未來24小時的溫度預測
train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step)
```

說明：

1. 選擇使用一半的步驟step，使時間序列資料變成兩倍長
2. 其中溫度資料以每30分鐘1個時間點的速率採樣

## 6-4-4 結合CNN與RNN處理長序列資料



✓ (續)為溫度預測數據集準備高解析率的資料產生器

```
val_gen = generator(float_data,
                    lookback=lookback,
                    delay=delay,
                    min_index=200001,
                    max_index=300000,
                    step=step)
test_gen = generator(float_data,
                    lookback=lookback,
                    delay=delay,
                    min_index=300001,
                    max_index=None,
                    step=step)
val_steps = (300000 - 200001 - lookback) // 128
test_steps = (len(float_data) - 300001 - lookback) // 128
```

## 6-4-4 結合CNN與RNN處理長序列資料



✓ 結合 1D 卷積層和 GRU 層的模式

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GRU(32, dropout=0.1, recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(), loss='mae')
```



## 6-4-4 結合CNN與RNN處理長序列資料



✓ 結合 1D 卷積層和 GRU 層的模型(續)

```
history = model.fit_generator(train_gen,  
steps_per_epoch=500,  
epochs=20,  
validation_data=val_gen,  
validation_steps=val_steps)
```

## 6-4-4 結合CNN與RNN處理長序列資料



✓ (續)結合 1D 卷積層和 GRU 層的模式

Out[] :

Layer (type)	Output Shape	Param #
conv1d_8 (Conv1D)	(None, None, 32)	2272
max_pooling1d_5 (MaxPooling1D)	(None, None, 32)	0
conv1d_9 (Conv1D)	(None, None, 32)	5152
gru_1 (GRU)	(None, 32)	6336
dense_7 (Dense)	(None, 1)	33
Total params: 13,793		
Trainable params: 13,793		
Non-trainable params: 0		

說明：此方法對對序列資料看得更細

1. 資料產生器回溯參數lookback
2. 減少產生器步驟參數step(檢視高解析率的時間序列)

## 6-4-4 結合CNN與RNN處理長序列資料



✓ 繪製結果

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

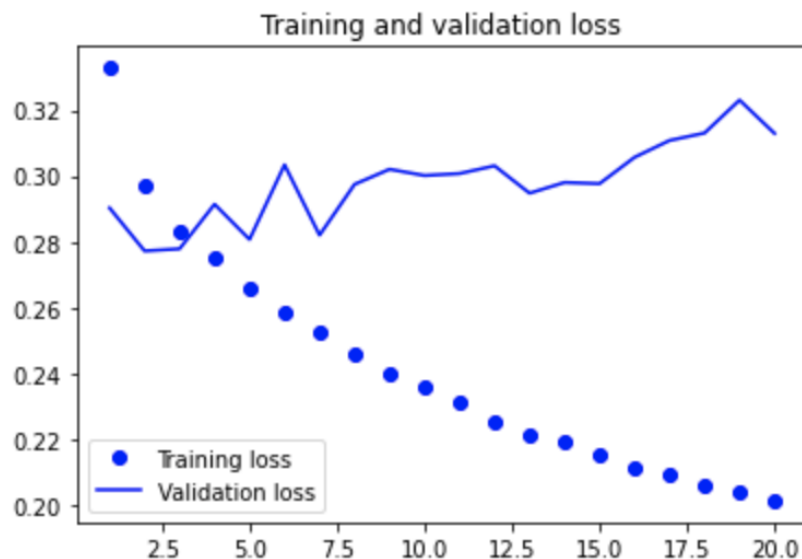
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

## 6-4-4 結合CNN與RNN處理長序列資料



✓ (續)繪製結果

Out[] :



總結：

從驗證的損失來看，此方法不如單獨GRU好，但速度明顯更快，且可檢視兩倍的資料。雖然對這個例子沒有太大的幫助，但也許對其他類型的資料集會有幫助。



目前為止，我應該知道的内容：

- ✓ 文件資料的預處理(preprocessing)
  - 如何對文件資料進行分詞
  - 什麼是嵌入向量以及如何使用
- ✓ 使用循環神經網路
  - 如何堆疊RNN層並使用雙向RNN來建構更強大的序列資料處理模型
- ✓ 使用1D convnet進行序列性資料的處理
  - 如何使用1D卷積神經網路進行序列資料處理
  - 如何結合1D卷積神經網路和RNN來處理長時間序列資料