Recurrent Neural Networks and Long Short Term Memory

Outlines

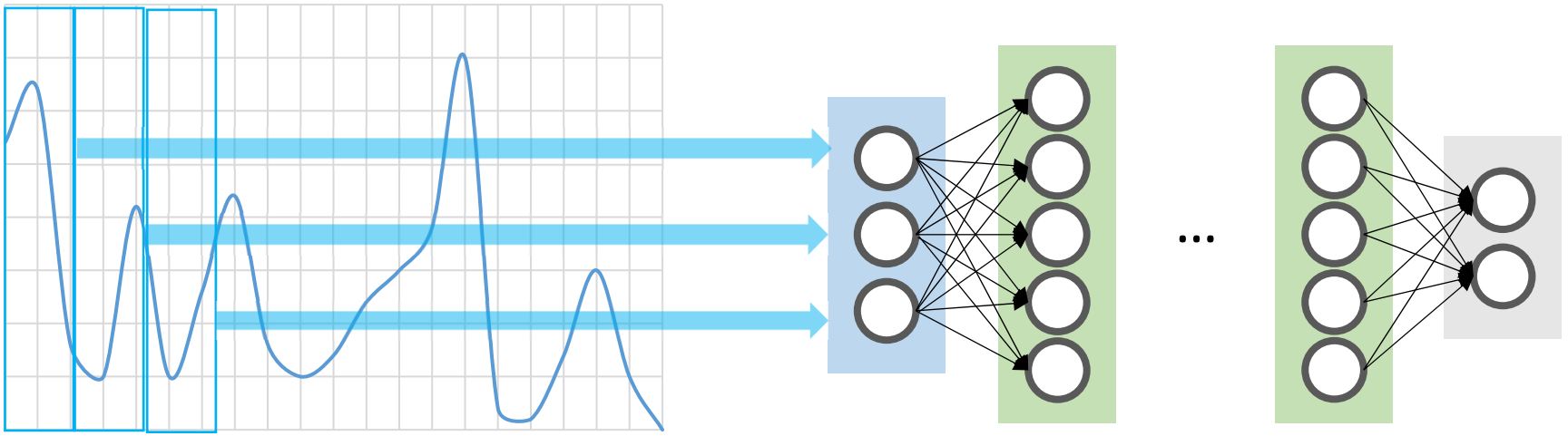
- **Recurrent neural networks (RNN)**
- **Long Short term Memory (LSTM)**
- **How to design a good LSTM**

Application

- **Document classification** and **timeseries** (時間序列) **classification**, **sequence data classification**, such as identifying the topic of an article or the author of a book
- **Timeseries comparisons**, such as estimating how closely **related** two documents or two stock tickers are **sequence-to-sequence learning**, such as decoding an English sentence into French.
- **Sentiment analysis (情感分析)**, such as classifying the sentiment of tweets or movie reviews as positive or negative
- **Timeseries forecasting**, such as **predicting** the future weather at a certain location, given recent weather data

Consider a Problem

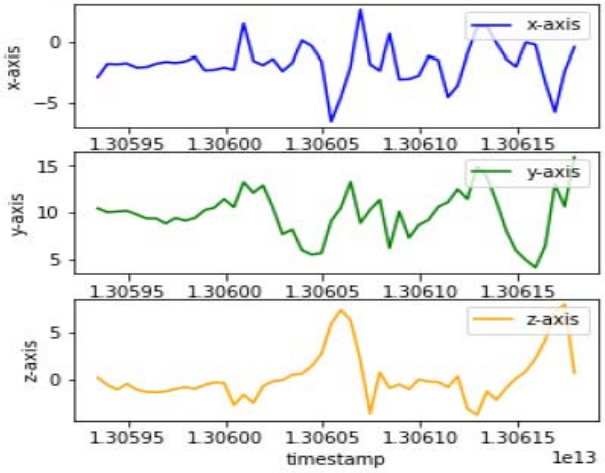
The relation between input ?



DNN

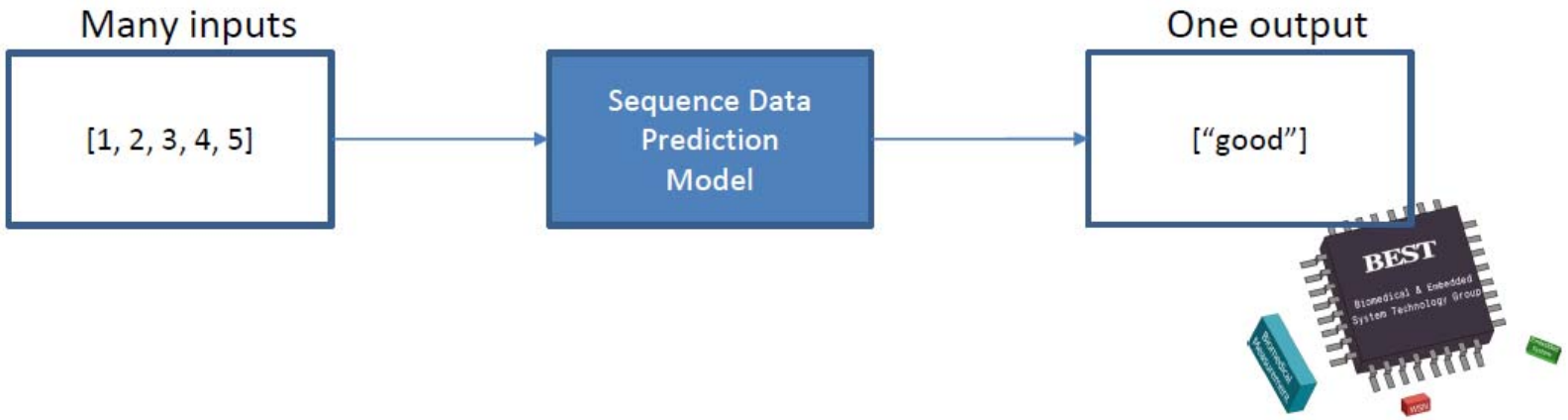
Sequence data classification

- Sequence data classification is different from other types of supervised learning problems.
  - Time information must be retained when training models and making predictions.
- Example: 3-axis data from accelerometer for activities recognition.



Sequence data classification

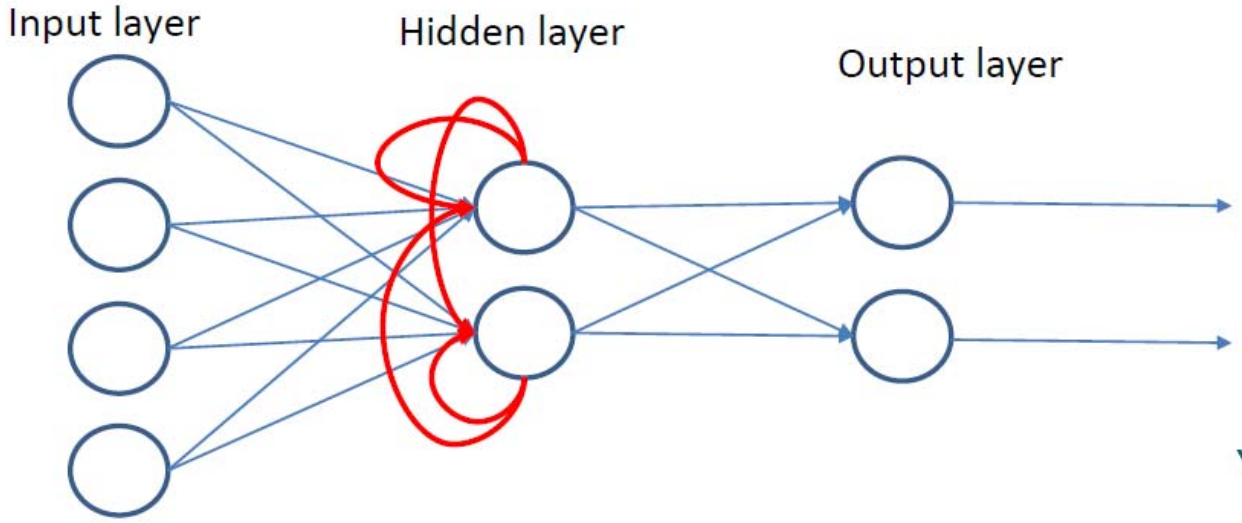
- The objective of sequence data classification is to build a classification model using a labeled dataset so that the model can be used to predict the class label of an unseen sequence.
- The problem we face is “Many to One” .

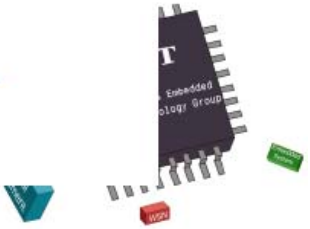


Understanding Recurrent Neural Networks

Recurrent neural network

- The outputs of the network depend not only on present inputs, but also PAST inputs.
- Through these connections the model can retain information about the past, enabling it to discover temporal correlations between events that are far away from each other in the data.





Recurrent neural network

- Recurrent neural network (RNN) can capture temporal dependences of input sequence.
  - Example: speech signal, text, hand written characters, etc.
- There are several RNN architectures based on the number of inputs and outputs.
  - One to Many : (Image captioning)
  - Many to One : (Human activity recognition)
  - Many to Many : (Emotional recognition)

Traditional V.S. Recurrent

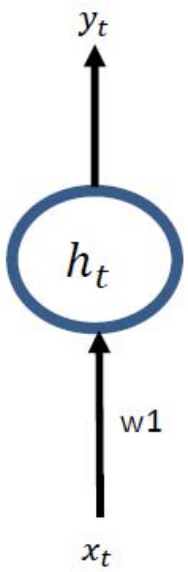
Traditional Network:
CNN、Fully Connect

- We called it “feed and forward” network.
- We have I/P and O/P, and then the same I/P always get same O/P.

Recurrent Network:
RNN、LSTM

- It takes the past I/P into account .
- We could get present O/P which is a function of present I/P and past O/P.

Cell of RNN



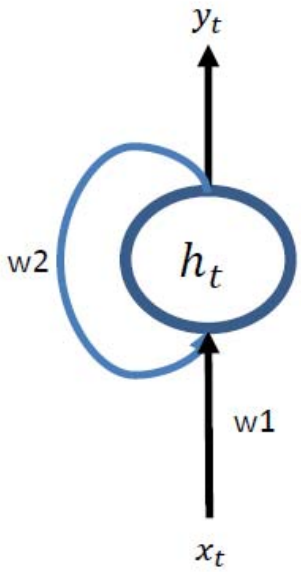
y_t=f(h_t) =$\frac{1}{1+e^{-h_t}}$

f(h_t): is activation (sigmoid)

$h_t = x_t w_1$

Normal cell (Perception)

Only 1 input



y_t=f(h_t) =$\frac{1}{1+e^{-h_t}}$

f(h_t): is activation (sigmoid)

$h_t = x_t w_1 + y_{(t-1)} w_2$

RNN cell

2 inputs

Recurrent neural network

- The network has the following equation.

$$h_t = x_t w_1 + y_{(t-1)} w_2$$

$$y_t = \mathrm{f}(h_t) = \frac{1}{1+e^{-h_t}} \text{ (sigmoid)}$$

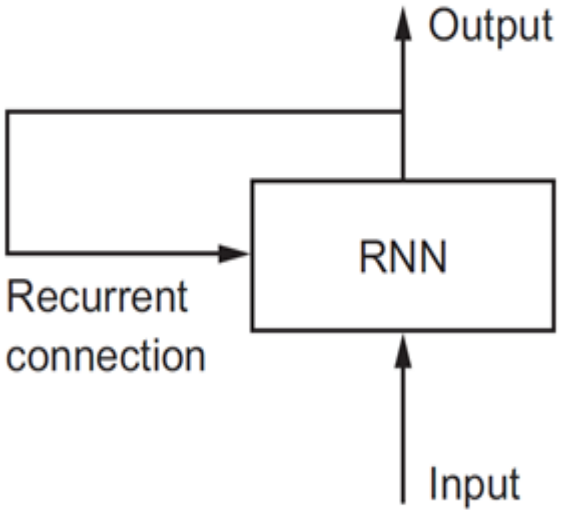
- Variable :
  - t is a discrete variable, i.e, t = 0, 1, 2…
  - We usually let $y_{-1} = 0$ as the initial condition.
  - Weights are not changed over variable t.
  - Weights only changed with training.

Understanding RNN

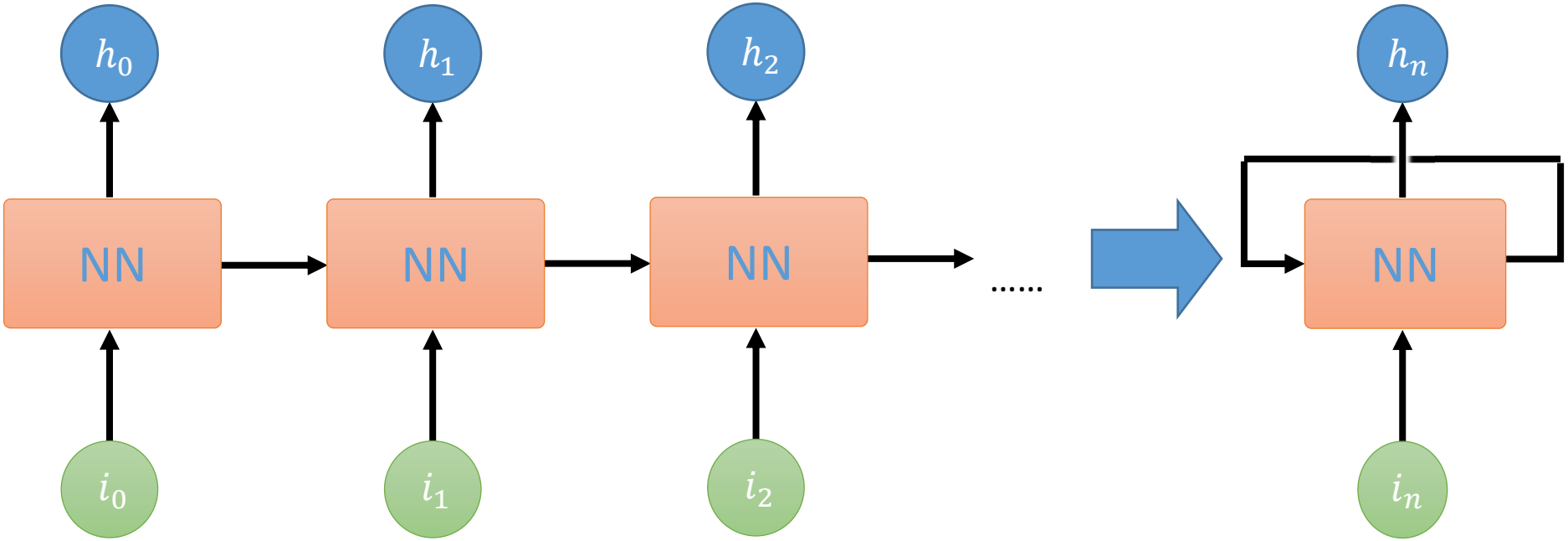
- A ***recurrent neural network*** **(RNN):** it processes **sequences** by iterating through the sequence elements and maintaining a ***state*** containing information relative to what it has seen so far.
- In effect, an RNN is a type of neural network that has an **internal loop**.

Listing 6.19 Pseudocode RNN

```
state_t = 0                          <-- The state at t
for input_t in input_sequence:       <-- Iterates over sequence elements
    output_t = f(input_t, state_t)
    state_t = output_t               <-- The previous output becomes the state for the next iteration.
```

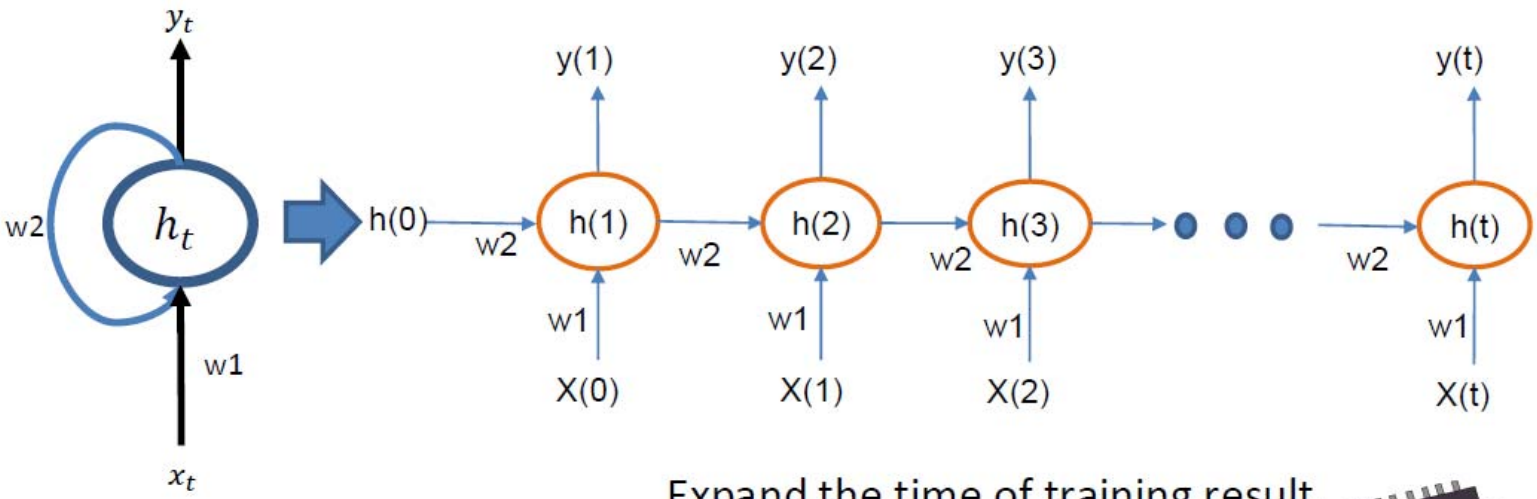


Recurrent Neural Networks (RNN)



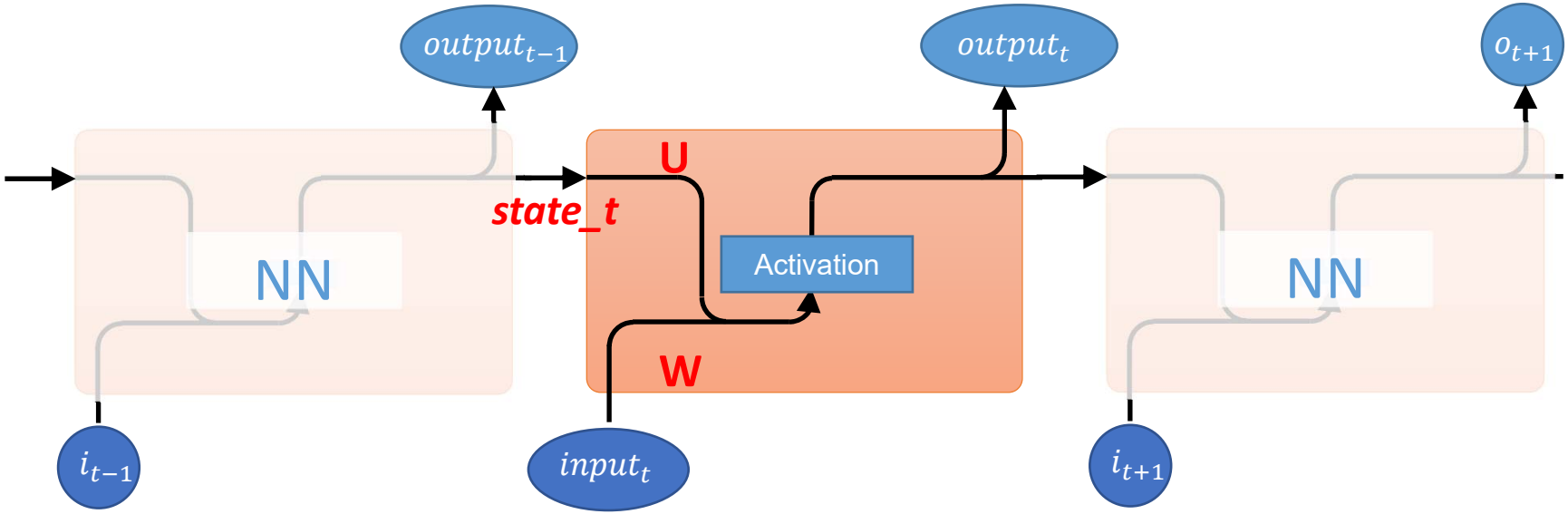
Recurrent neural network

- Imagine that we are training for this perceptron, and t is the training epoch.



Expand the time of training result.

Inside a standard RNN




```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

以 Numpy 實現簡單的 RNN

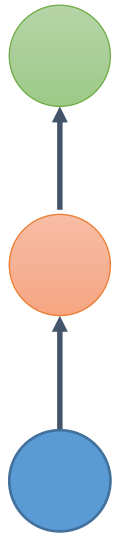
```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

```
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

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

successive_outputs = []
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網絡狀態

final_output_sequence = np.concatenate(successive_outputs, axis=0)
print(final_output_sequence.shape)
```

The Types of RNN

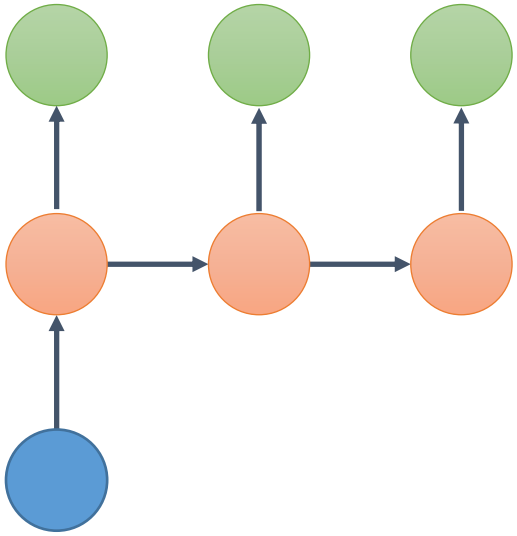
- One to One
- One to many
- Many to one
- Many to many
  - Synced
  - Unsynced

One to One



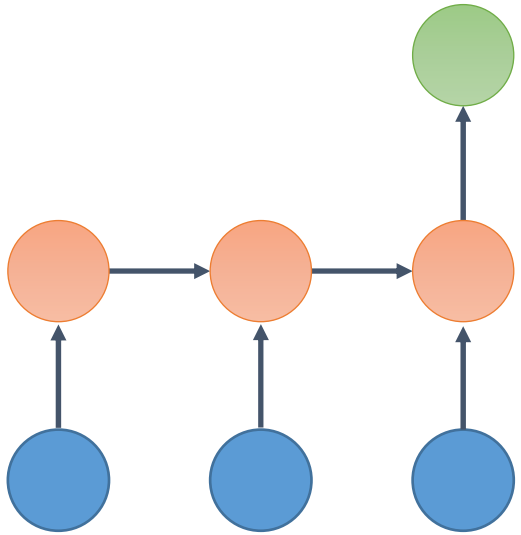
One to Many

- One input
- Many outputs
  - The output have time dependency
- E.g.) Image captioning (標題)



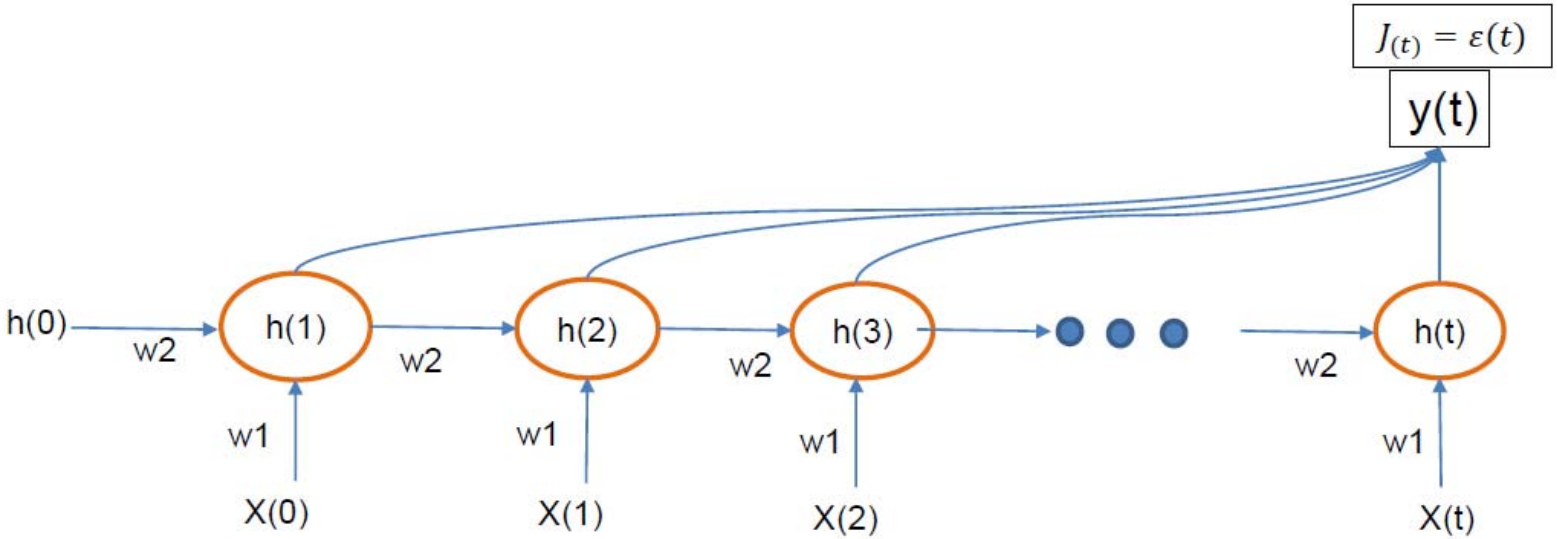
Many to One

- Many inputs
  - The inputs have time dependency
- One output
- E.g.) **Sentiment (情緒) analysis**



RNN many to one

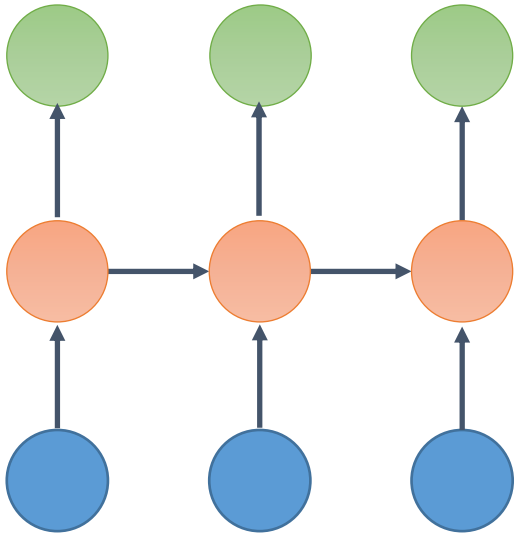
- Unroll RNN training (Many to One).



Many to Many (synced)

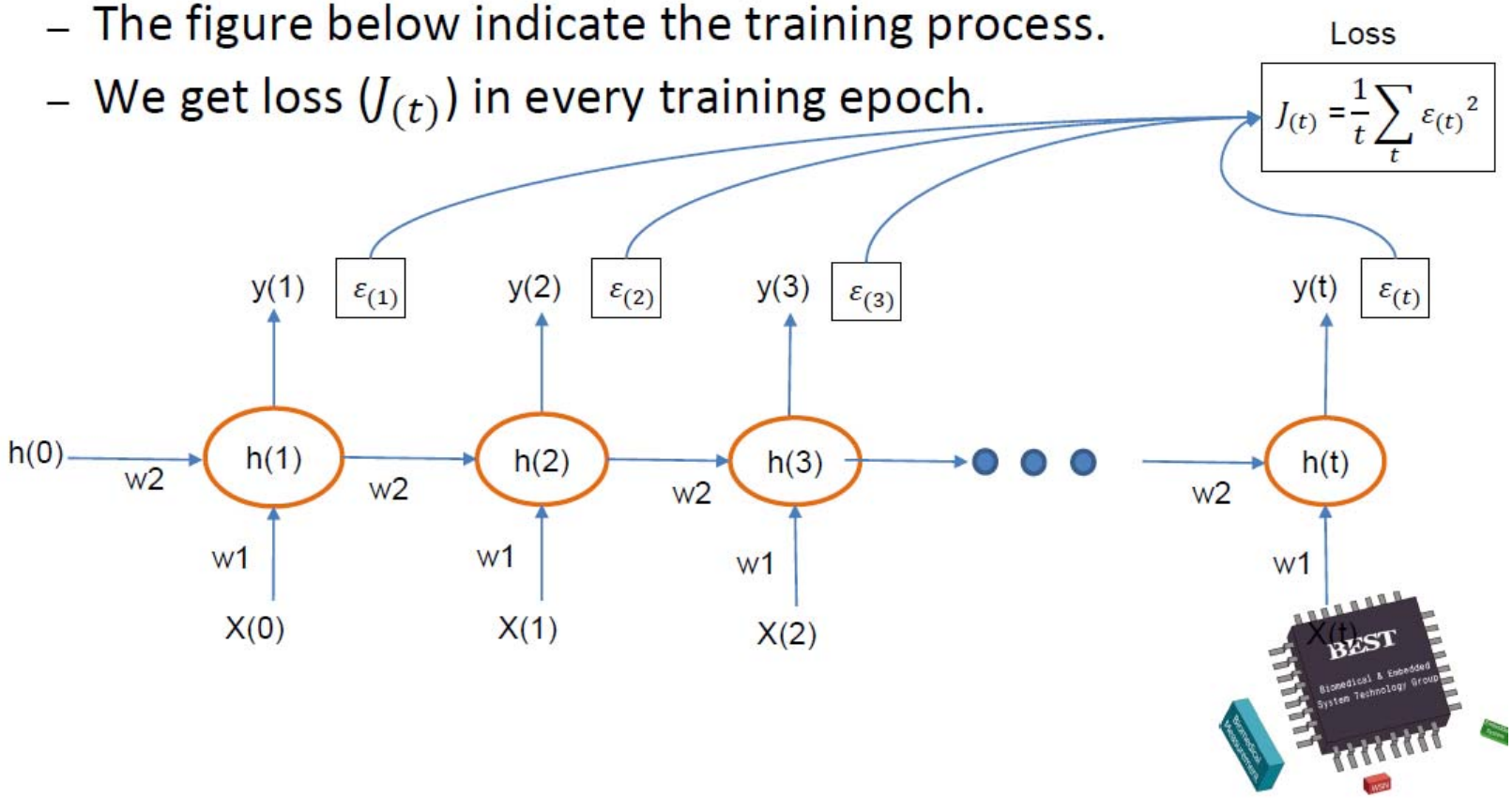
- Many inputs
- Many outputs
- All of them have **time dependency**
- Input **size** is the same as output size

- E.g.) Predicting the next word



RNN

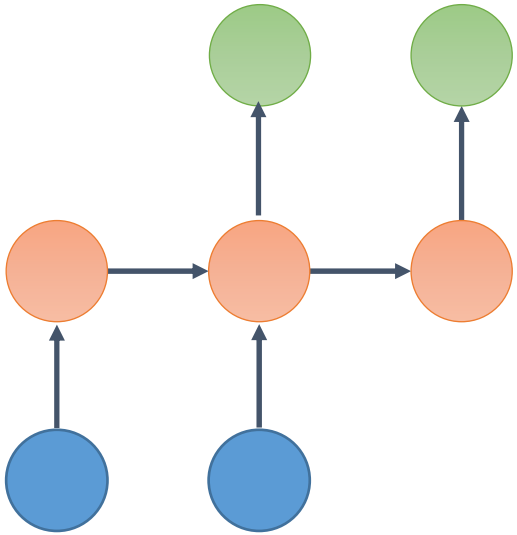
- Unroll RNN training (Many to Many).
  - The figure below indicate the training process.
  - We get loss ($J_{(t)}$) in every training epoch.



Many to Many (unsynced)

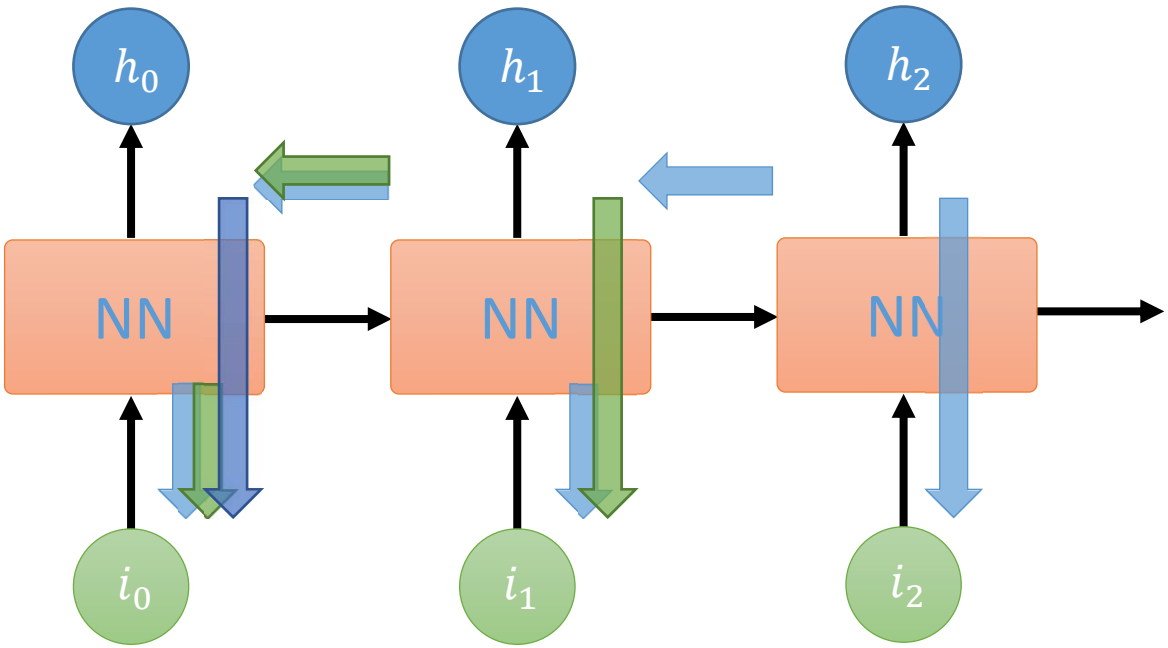
- Many inputs
- Many outputs
- All of them have time dependency
- The lengths could be different

- E.g.) **Translation**



Training on RNN

- **Back propagation** through time
  - The error should propagate to previous nodes



Loss function and cost function

- Loss function
  - It is simply the deviation of true value from predicted value, now this can be in form of squared difference (MSE) or absolute difference (MAE) etc.

- Cost function
  - While the loss function is for only one training example, the cost function accounts for entire data set.

Loss of RNN

- We define the cost function as the sum of MSE over time (if there are multiple output nodes, all outputs are also summed).

$y_{(t)}$: predicted label

$d_{(t)}$: desired label

(deviation) $\varepsilon_{(t)} = d_{(t)} - y_{(t)}$, t = 0,1,2…

(Loss function - MSE) = $(d_{(t)} - y_{(t)})^2$

(Cost function - MSE) = $\frac{\sum_t (d_{(t)} - y_{(t)})^2}{t}$

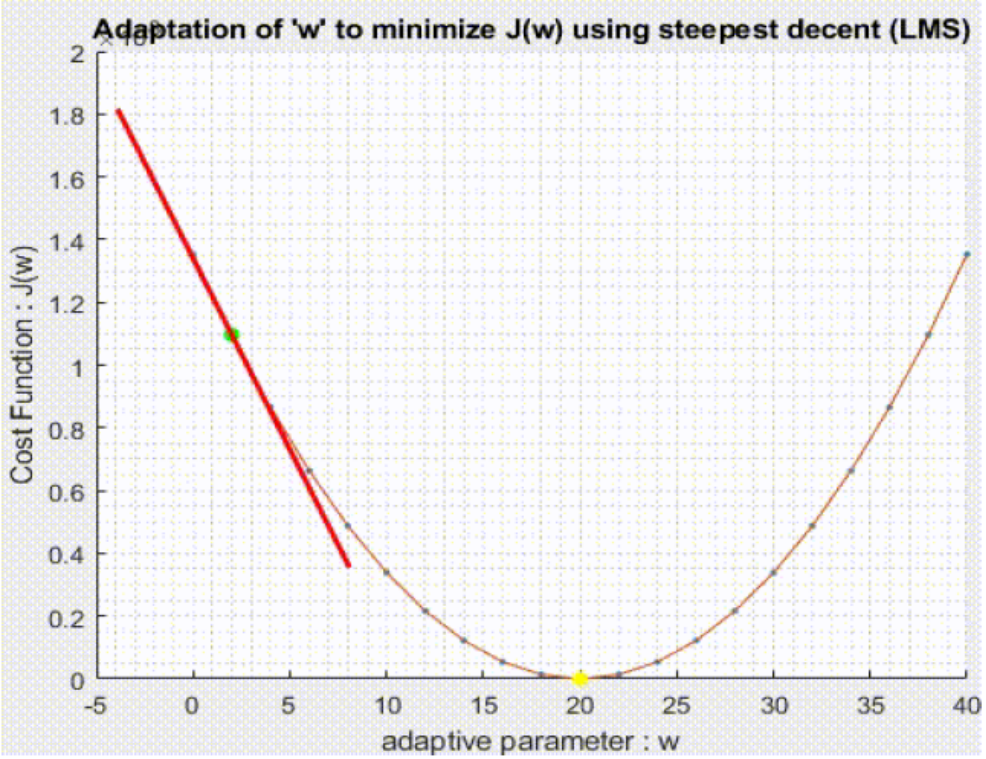
(Cost function of RNN perception) $J_{(t)} = \frac{1}{t}\sum_t \varepsilon_{(t)}{}^2 = \frac{1}{t}\sum_t (d_{(t)} - y_{(t)})^2$

Gradient

- Gradient:
  - During model training, after we feed training data into model, and we get the prediction error to make loss function.
  - Gradient can be obtained by partial differentiation of loss function with respect to the weights .
  - Gradient is a kind of vector, and the direction of gradient is the direction of the weights update.

Gradient

- MSE gradient descent to update weight to get smaller loss.



Recurrent neural network

- It can be difficult to train standard RNNs to solve problems that require learning long-term temporal dependencies.

- When you use RNN as the training model, gradient explosion and gradient vanish may happen quite often.

Recurrent neural network

- Gradient explosion
  - The large gradients accumulate and result in very large updates to weight in model during training.
- Gradient vanish
  - The gradient will get smaller until the gradient disappears completely, which will stop us from training the model.

A recurrent layer in Keras

- The SimpleRNN layer for implementing RNN:

```
from keras.layers import SimpleRNN
```

- **SimpleRNN processes batches of sequences**, not a single sequence as in the Numpy example.
- It takes inputs of shape **(batch_size, timesteps, input_features)**, rather than **(timesteps, input_features).**
- **SimpleRNN** can be run in two different modes (controlled by parameter **return_sequence**):
  - It can return either the **full sequences** of successive outputs for each timestep (a 3D tensor of shape (batch_size, timesteps, output_features)).
  - Only the **last output** for each input sequence (a 2D tensor of shape (batch_size, output_features)).

Keras Embedding Layer

- Keras提供了一個**嵌入層(Embedding Layer)**，適用於文本(text)資料的神經網路。
- 它要求輸入資料是整數編碼的，所以每個字都用一個唯一的整數表示。這個資料準備步驟可以使用Keras提供的Tokenizer API來執行。
- 嵌入層(Embedding Layer)用**隨機權重**進行初始化，並將學習訓練資料集中所有單詞的嵌入。
- 它是一個靈活的圖層，可以以多種方式使用，例如：
  - 它可以單獨使用來學習**一個單詞嵌入**，以後可以保存並在另一個模型中使用。
  - 它可以用作深度學習模型的一部分，其中嵌入與模型本身一起學習。

Embedding Layer

- 嵌入層被定義為網路的第一個隱藏層。它必須指定3個參數：
  - Input_dim：這是文本資料中詞彙的取值可能數。例如，如果您的資料是整數編碼為0-9之間的值，那麼詞彙的大小就是10個單詞；
  - output_dim：這是嵌入單詞的向量空間的大小。它為每個單詞定義了這個層的輸出向量的大小。例如，它可能是32或100甚至更大，可以視為具體問題的超參數；
  - input_length：這是輸入序列的長度，就像您為Keras模型的任何輸入層所定義的一樣，也就是一次輸入帶有的詞彙個數。例如，如果您的所有輸入文檔都由1000個字組成，那麼input_length就是1000。
- 定義一個詞彙表為200的嵌入層（例如從0到199的整數編碼的字，包括0到199），一個32維的向量空間，其中將嵌入單詞，以及輸入文檔，每個單句有50個單詞。
- **e = Embedding(input_dim=200, output_dim=32, input_length=50)**

以嵌入向量 Embedding 層和 SimpleRNN 層訓練模型

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_22 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
simplernn_10 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

RNN Example 1

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_23 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
simplernn_11 (SimpleRNN)     (None, None, 32)          2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

- **return_sequences**:默認為false。
  - 當為false時，返回最後一層最後一個步長的hidden state;
  - 當為true時，返回最後一層的所有hidden state。
- **return_state**:默認false.
  - 當為true時，返回最後一層的最後一個步長的輸出hidden state和輸入cell state。

RNN Example 2

```
>>> 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))
>>> model.summary()
```

Last layer only returns the last output

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_24 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
simplernn_12 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simplernn_13 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simplernn_14 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simplernn_15 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
```

IMDB sentiment analysis (情緒分析)

- 利用循環神經網路RNN實作IMDb數據資料的訓練，
- 通常CNN是應用於圖像辨識，而RNN用於文字處理分析，為什麼要用RNN呢?
- 因為在文字的世界會因為時間循序漸進，所以我們要把模型模擬跟人一樣要有過往的記憶，利用過往的歷史資訊來預測未來。
- **情緒分析** sentiment analysis 又稱為意見探勘 ，是以自然語言處理、文字分析的方法，找出作者在某些話題的態度、情感、評價或情緒。
- IMDb: Internet Movie Database是線上電影資料庫，始於 1990 年，1998 年起，成為 amazon 旗下的網站，收錄了 4百多萬筆作品資料。
- IMDb 資料及共有 50000 筆影評文字，分訓練與測試資料各 25000 筆，每一筆資料都被標記為「**正面評價**」或「**負面評價**」

準備 IMDB 資料 example 6-18

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

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') # 25000 筆訓練用序列資料 (評論)
print(len(input_test), 'test sequences')    # 25000 筆測試用序列資料

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

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)
```

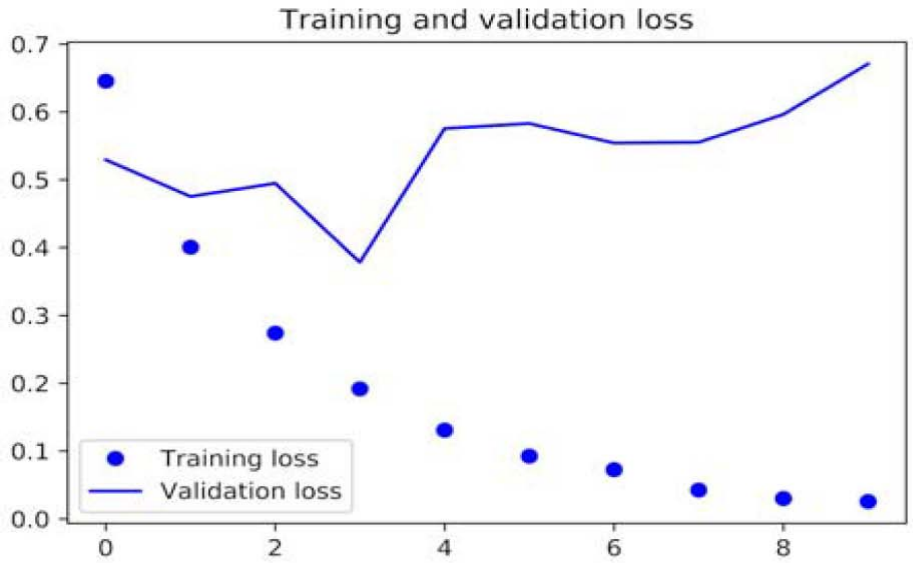


Figure 6.11 Training and validation loss on IMDB with `SimpleRNN`

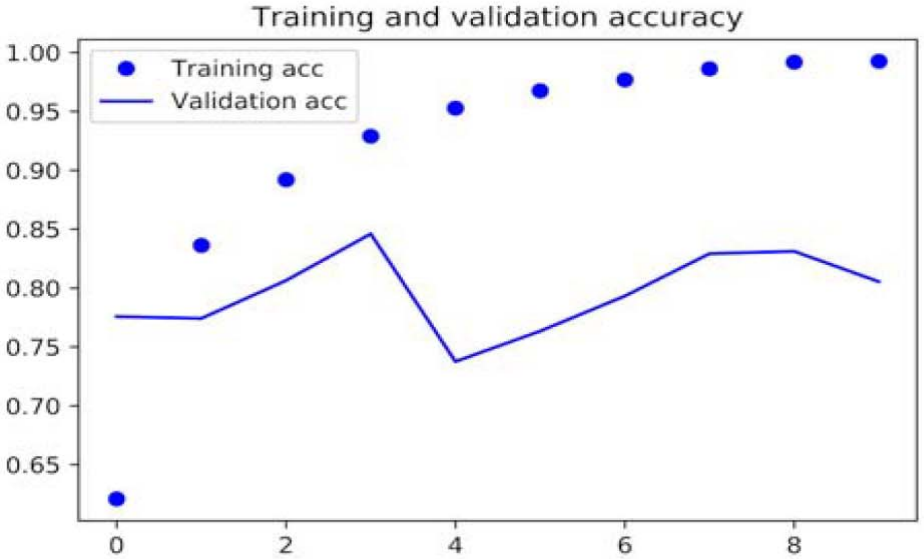
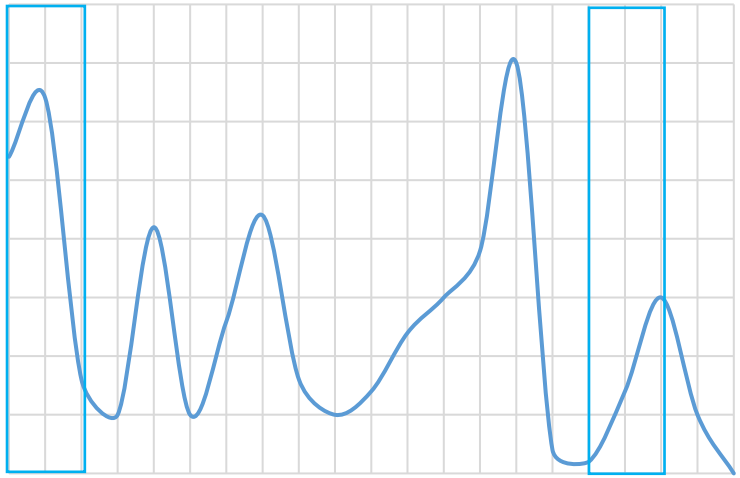


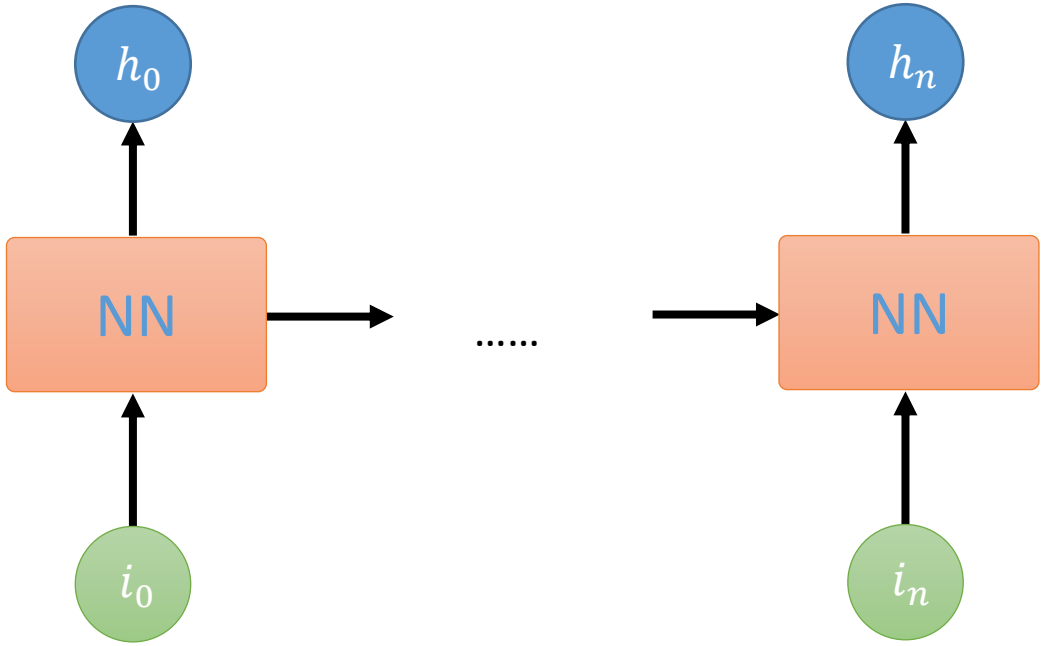
Figure 6.12 Training and validation accuracy on IMDB with `SimpleRNN`

Understanding the LSTM

Another Problem of RNN



The relation between problems ?



Understanding *the LSTM*

- SimpleRNN isn't the only recurrent layer available in Keras. There are two others: LSTM and GRU.
- In practice, you'll always use one of these, because SimpleRNN is generally **too simplistic** to be of real use.
- SimpleRNN has a **major issue**:
  - Although it should theoretically be able to retain at time *t* information about inputs seen many timesteps before, in practice, such **long-term dependencies** are **impossible to learn**.
  - This is due to the ***vanishing gradient problem(梯度消失問題)***, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers **deep**: as you keep adding layers to a network, the network eventually becomes **untrainable**.

Long Short-Term Memory (LSTM)

- The underlying **Long Short-Term Memory (LSTM)** algorithm was developed by Hochreiter and Schmidhuber in 1997; it was the culmination of their research on the vanishing gradient problem.
- It adds a way to **carry information across many timesteps**.
- Imagine a **conveyor belt** running parallel to the sequence you're processing. Information from the sequence can **jump onto** the conveyor belt **at any point**, be transported to a later timestep, and **jump off**, intact, when you need it.
- This is essentially what LSTM does: **it saves information for later, thus preventing older signals from gradually vanishing during processing**.

The Memories in LSTM

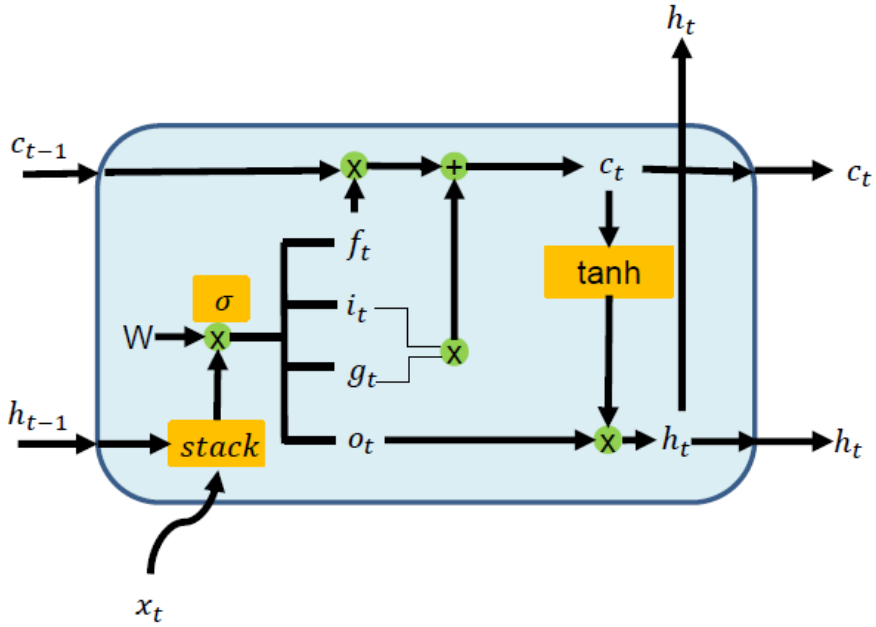
- **Short term memory**
  - Record the information from the neighboring input
  - Traditional RNN
- **Long term memory**
  - Record the input far far away

- LSTM可以通過 **gate (閘門)** 決定網絡需要記住和遺忘多長時間之前的記憶，以此聯合之前的狀態、記憶和輸入。

LSTM (Long Short-Term Memory)

- One improvement to RNN is LSTM, standing for long short-term memory.
- Meaning that their model can have long-term or short-term memory.
- A memory cell in LSTM unit has a feedback with weight of one.

Cell of LSTM



One LSTM cell

i_t, f_t, O_t Input, forget and output **gate** from 0 to 1

C_t memory

x_t Input, y_t output

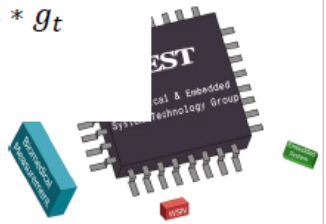
σ sigmoid function

$$\begin{pmatrix} i_t \\ f_t \\ o_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

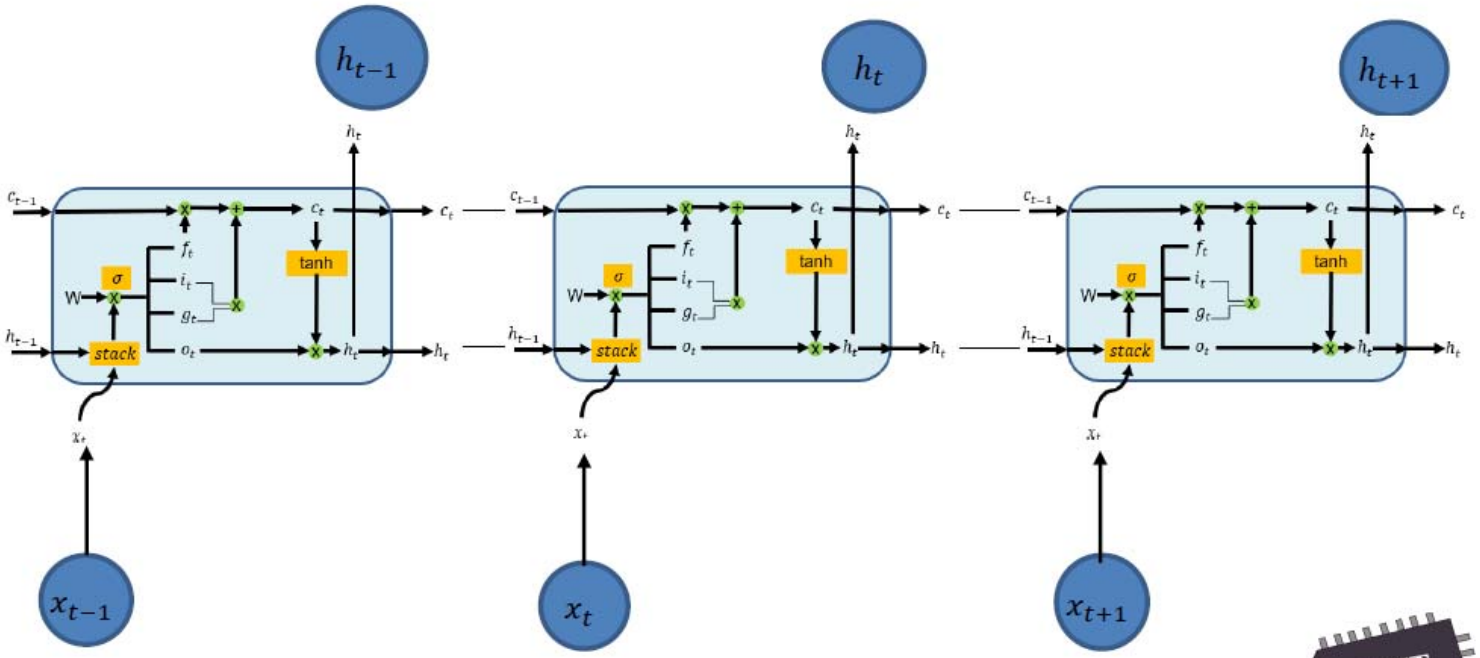
$$c_t = f_t * c_{t-1} + i_t * g_t$$

$$h_t = o * \tanh(c_t)$$

All inputs and outputs connect to all gates and weight.

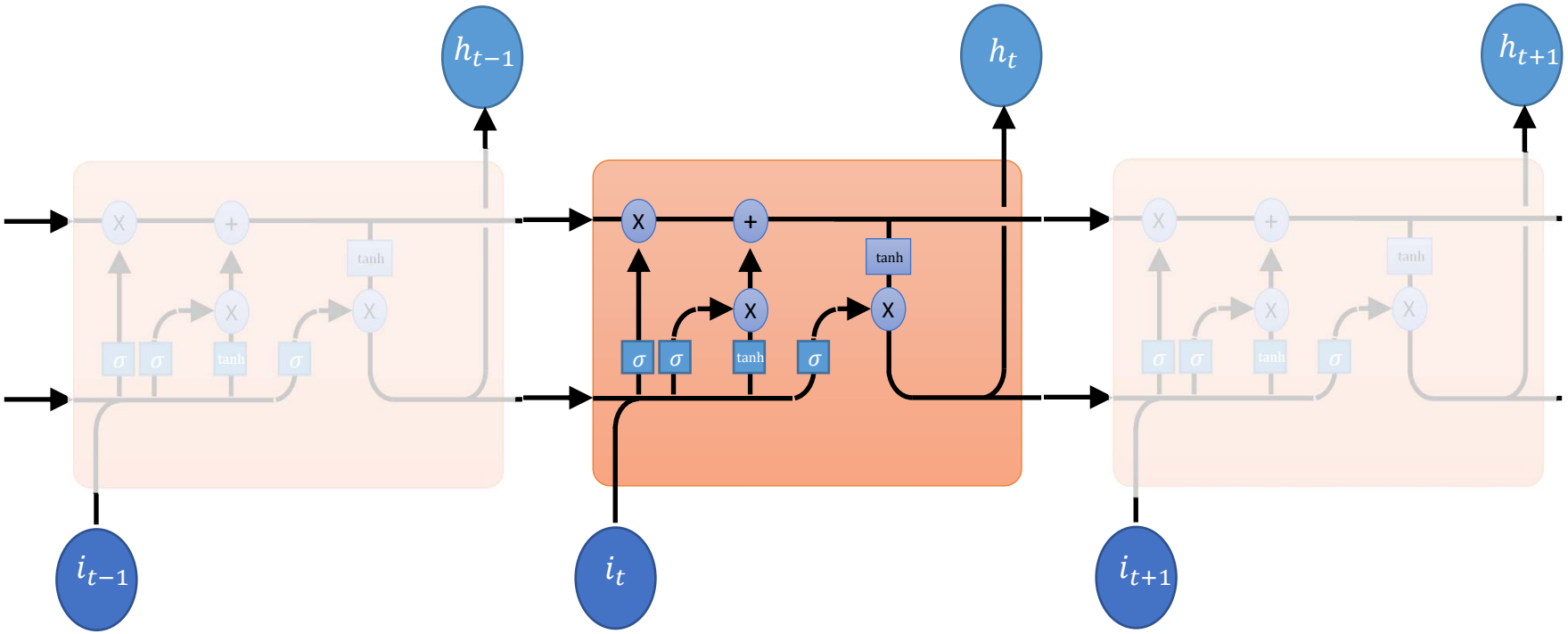


LSTM (Long Short-Term Memory)

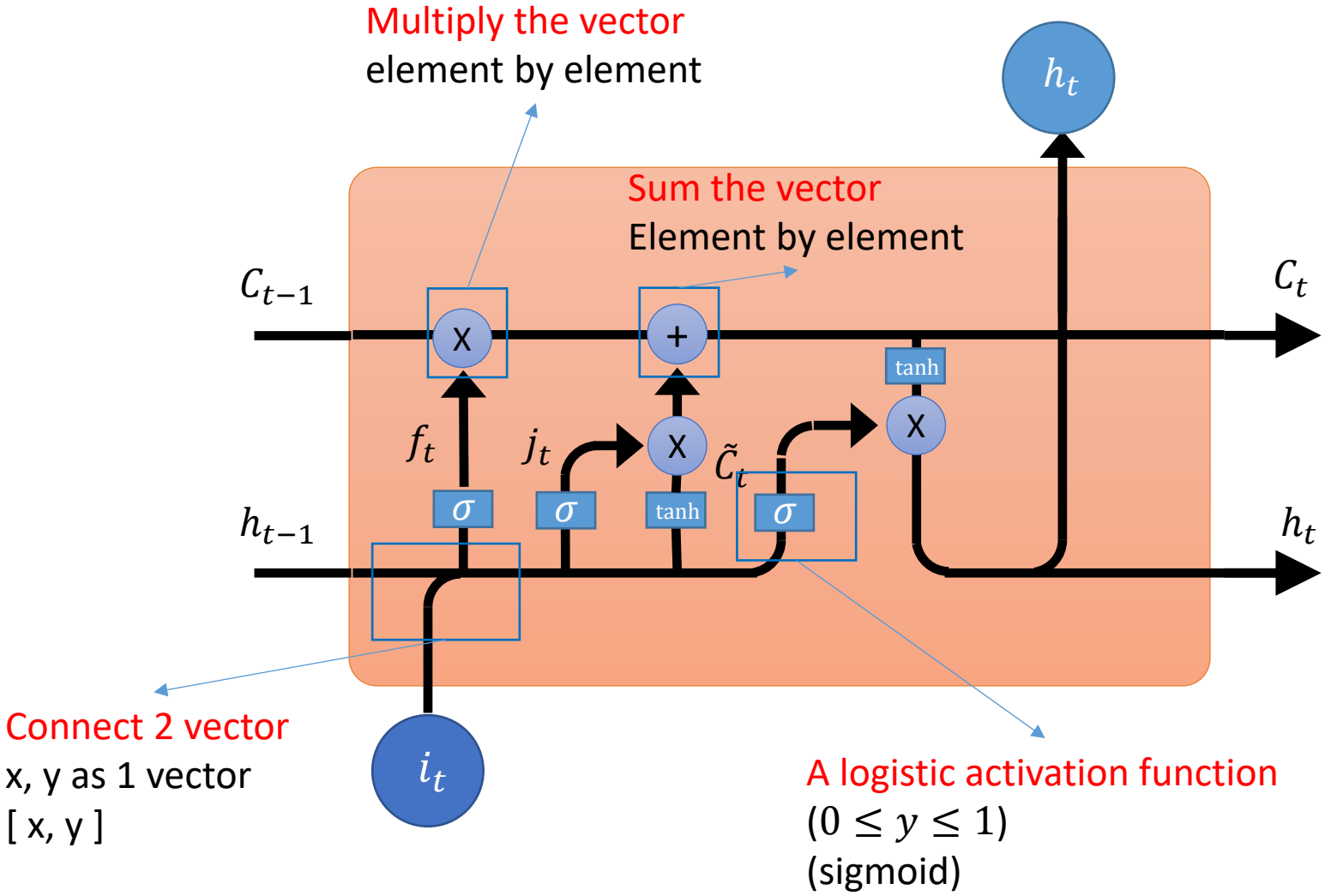


LSTM (另一版本)

The Structure of LSTM

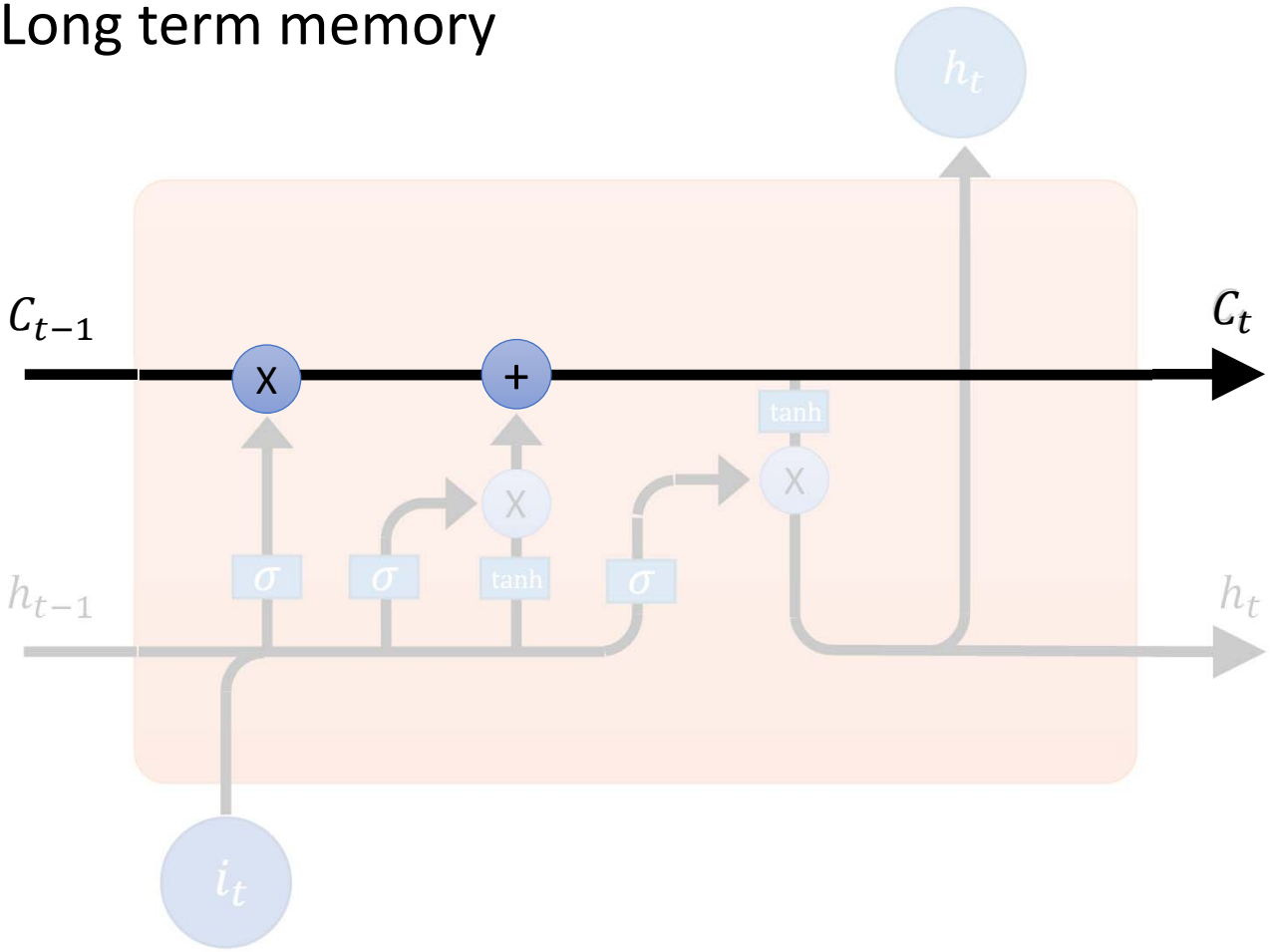


The meaning in each block



The Memory in LSTM

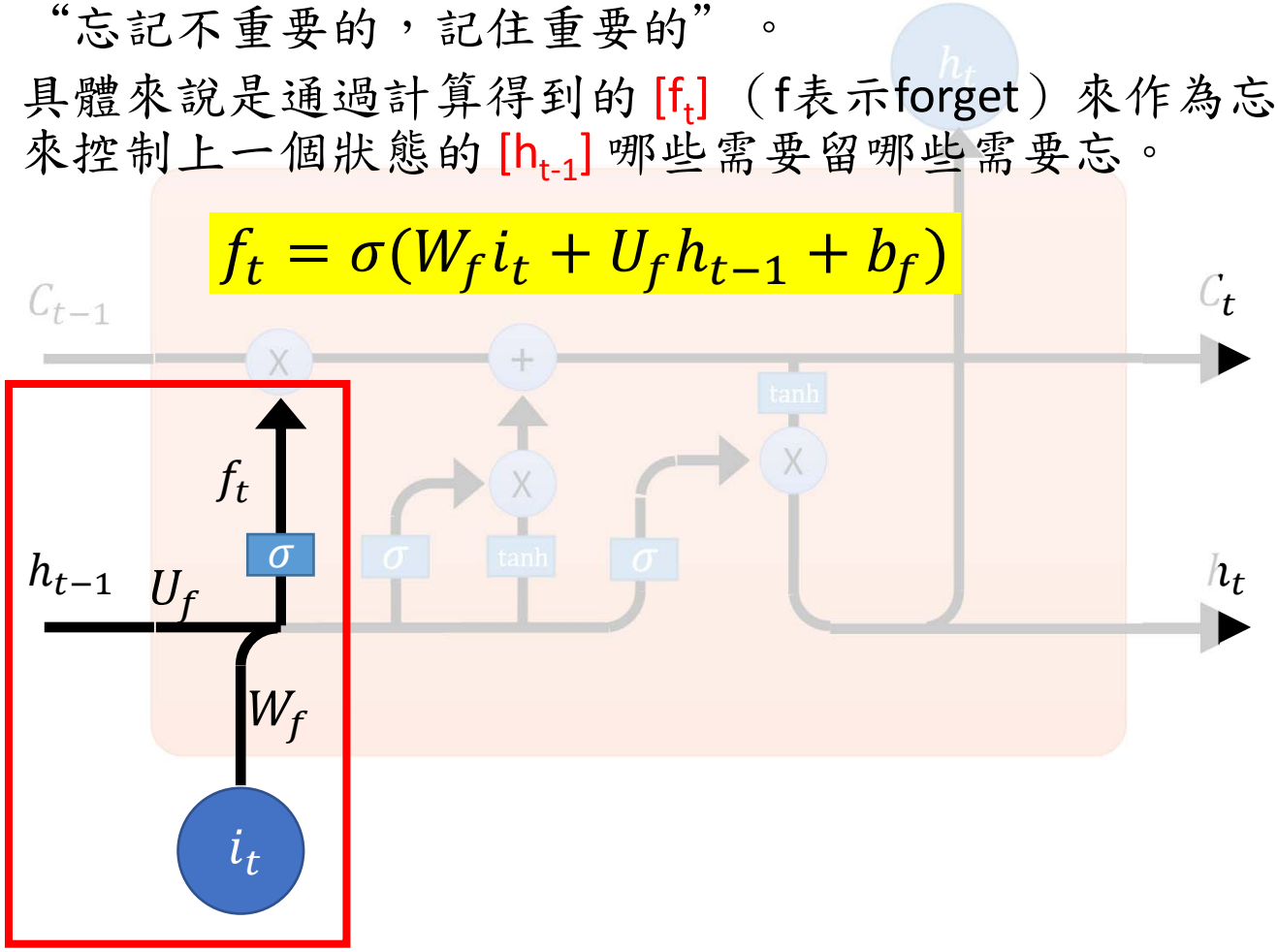
- Record the previous outputs
  - Long term memory



Forget Gate

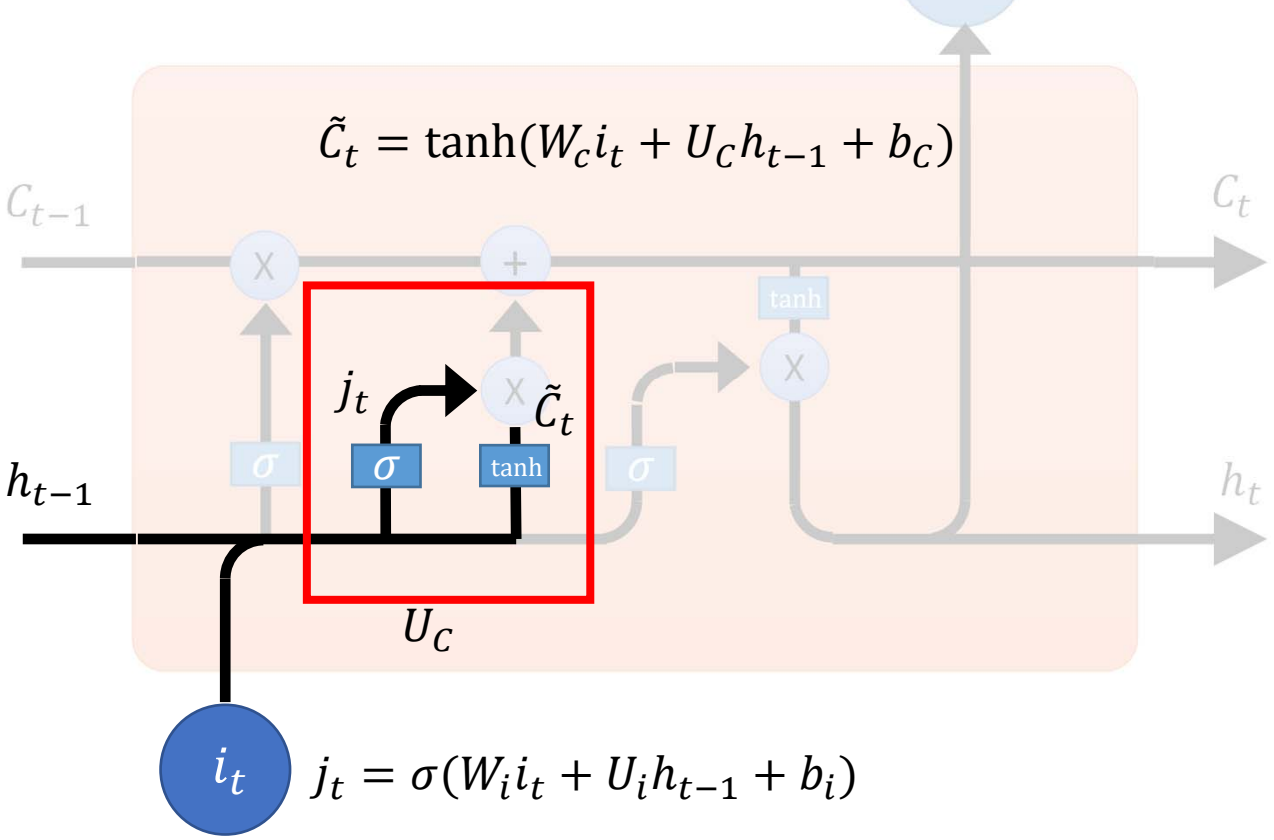
- Determine if we need to use the data from long term memory
  - 這個階段主要是對上一個節點傳進來的輸入進行選擇性忘記。“忘記不重要的，記住重要的”。
  - 具體來說是通過計算得到的 [f_t]（f表示forget）來作為忘記門控，來控制上一個狀態的 [h_{t-1}] 哪些需要留哪些需要忘。

$$f_t = \sigma(W_f i_t + U_f h_{t-1} + b_f)$$



Input Gate

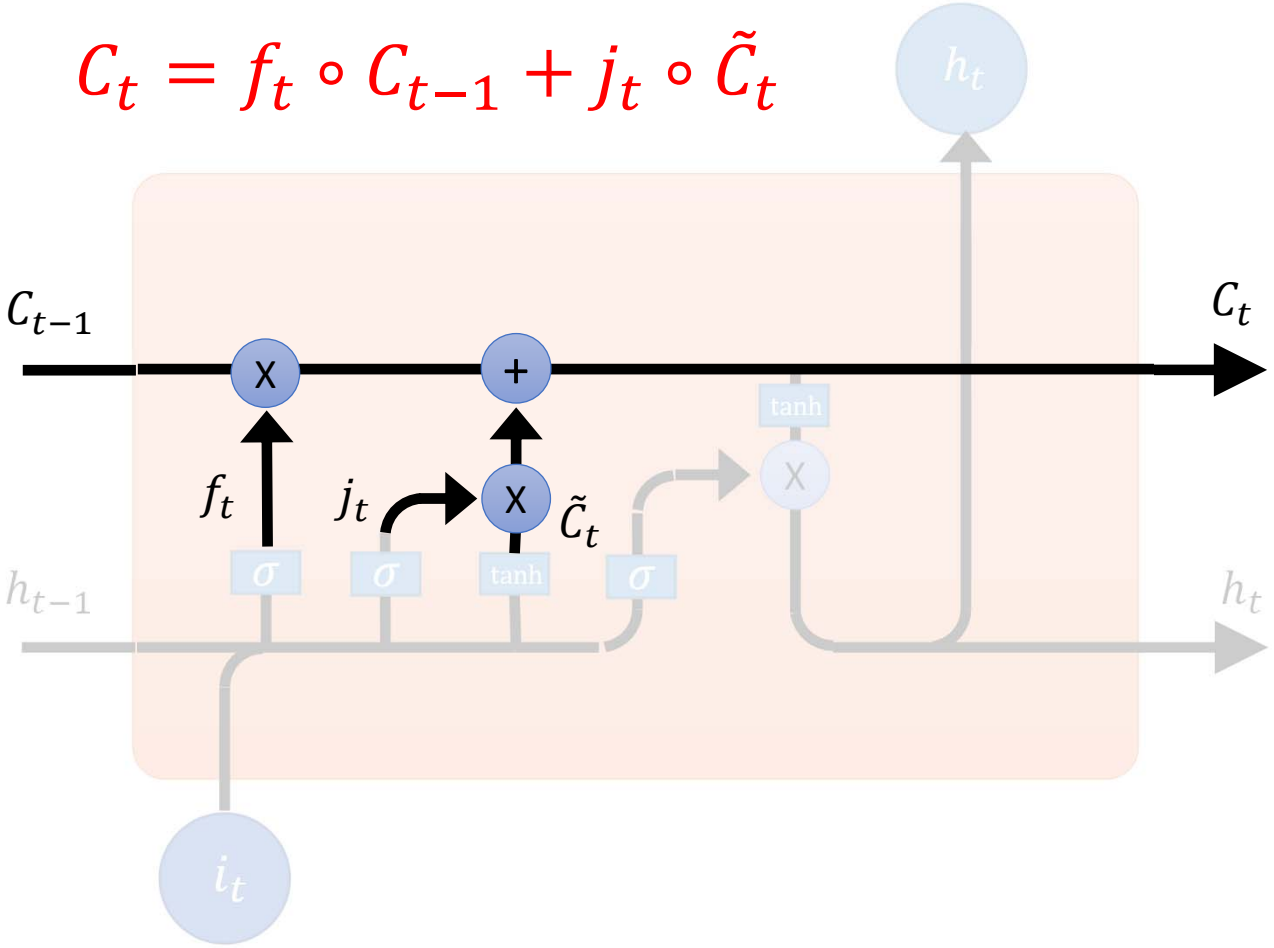
- Determine if we need to consider data from short term memory
- 決定有多少新的資訊要被記錄下來，然後 tanh 激勵函數算出一個向量，決定有多少資訊 C_t 要用來更新主要單元。



Update Cell State

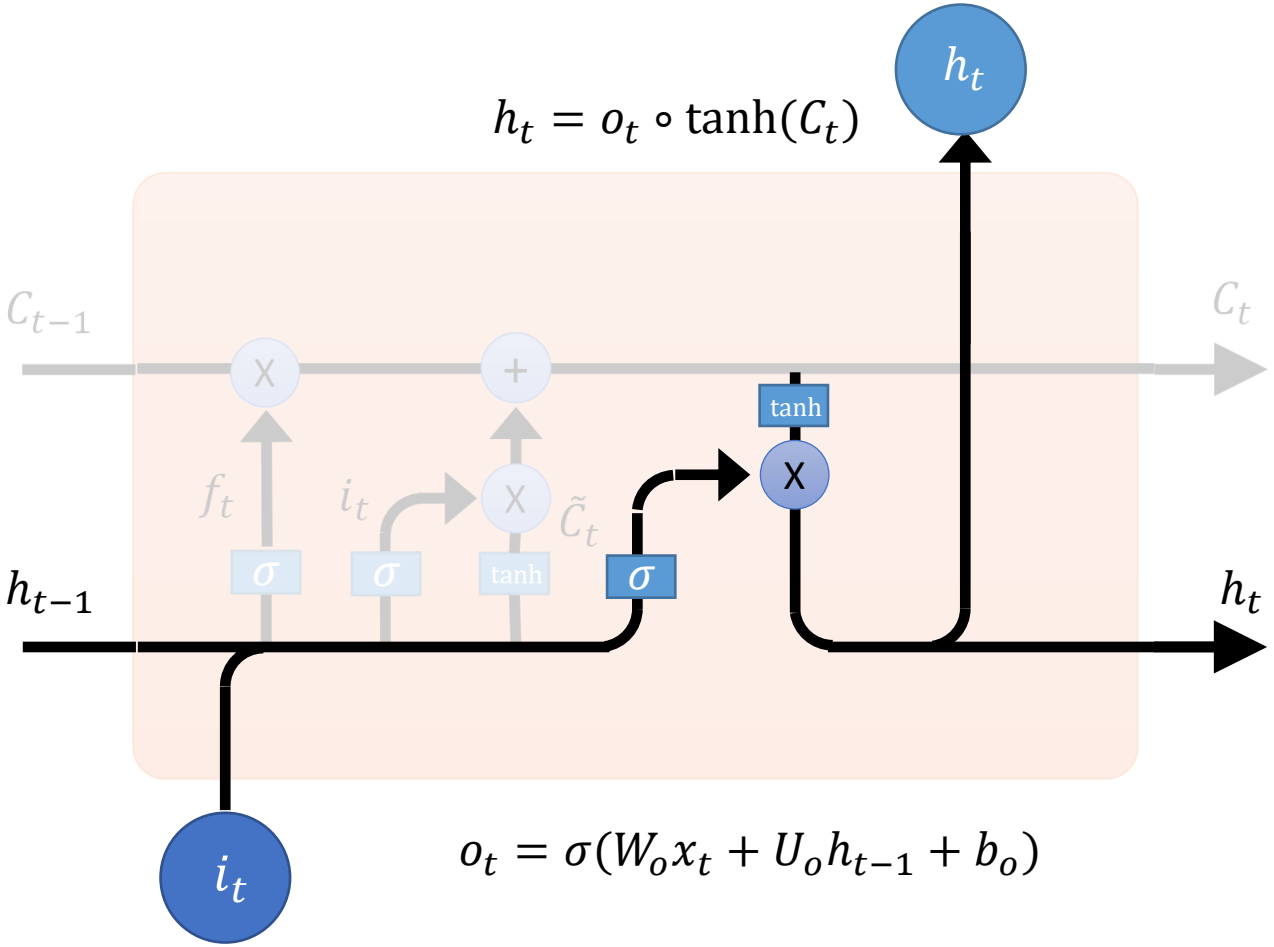
- Update the data on long term memory

$$C_t = f_t \circ C_{t-1} + j_t \circ \tilde{C}_t$$



Output Gate

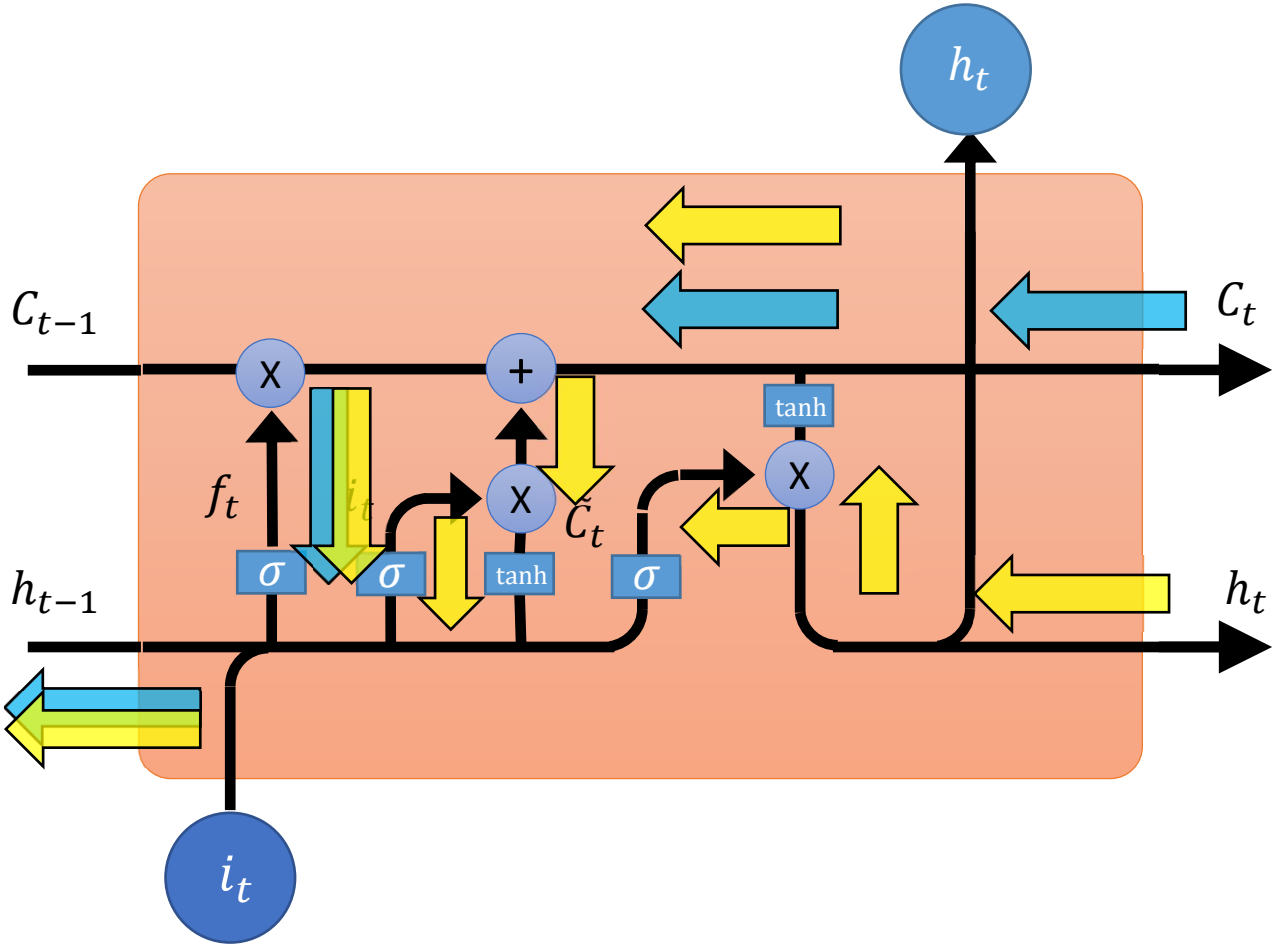
- Determine if we need previous data in next time



RNN 與 LSTM 的強大效果以及應用方向

- 遞歸神經網路強大的地方在於它允許輸入與輸出的資料不只是單一組向量，而是多組向量組成的**序列**。
- 當只能一組固定大小的向量輸入，且只允許一組向量輸出時時，可以處理分類問題。
- 但一組輸入多組輸出時，就可以處理將圖片自動標上文字的問題。
- 多組輸入一組輸出則可以處理文本的情感理解，比如說從一段文章中知道是正面或反面的結果。
- 而當可以處理多組輸入，多組輸出時，就可以處理自動翻譯，或是影片分類的問題。

Training LSTM



A concrete LSTM example in Keras

Code example 6-21 with IMDB

```
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)

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')
```

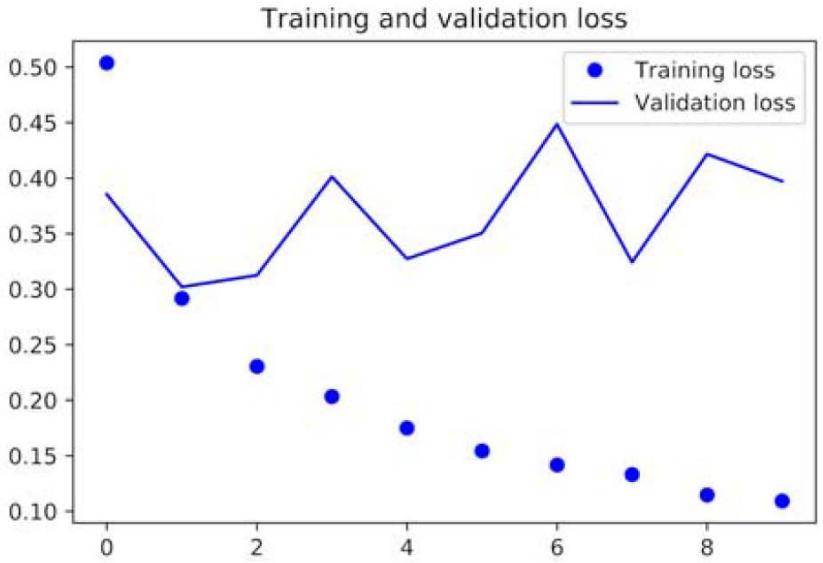


Figure 6.16 Training and validation loss on IMDB with LSTM

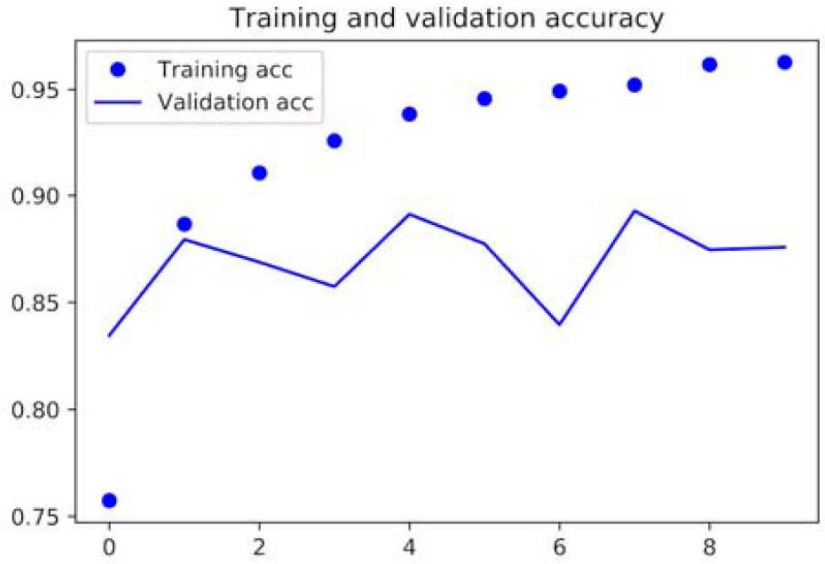
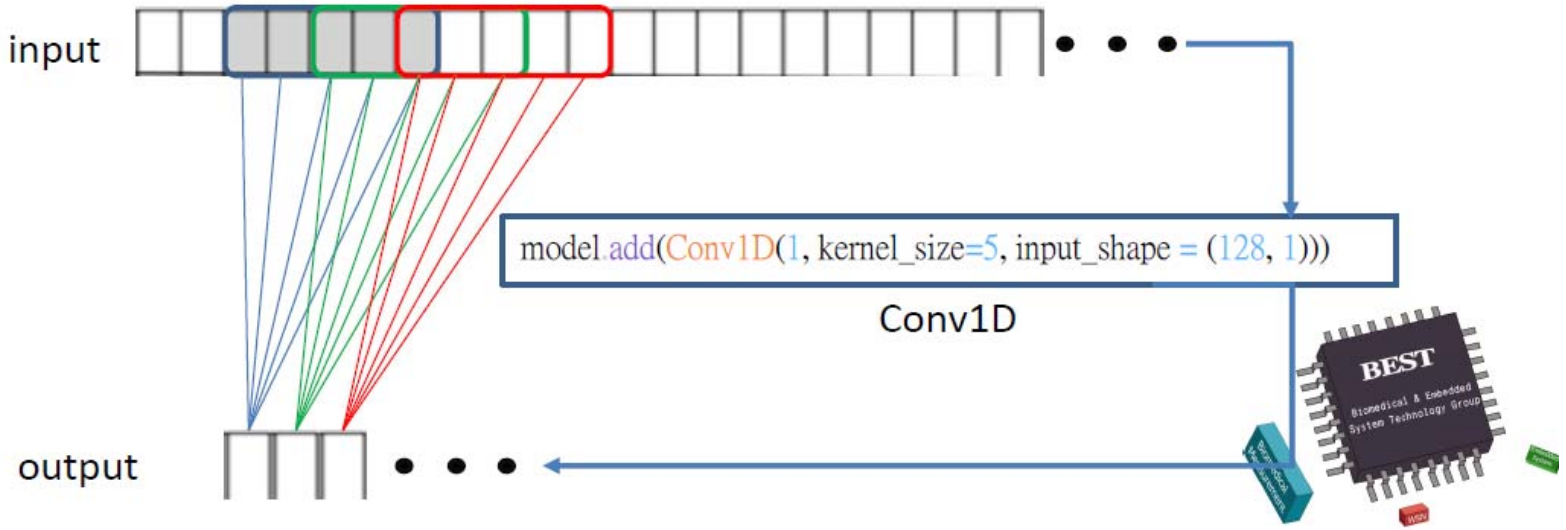


Figure 6.17 Training and validation accuracy on IMDB with LSTM

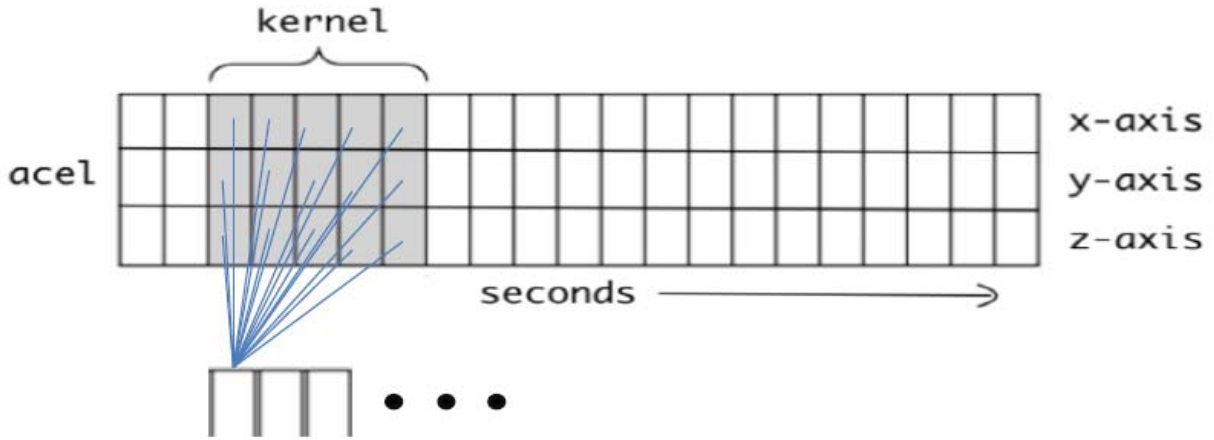
1D-CNN

- Time series data have a strong 1D structure, this means the temporally nearby variables exhibit a strong correlation.
- The output of 1D-CNN is also 1-dimensional vector.



1D-CNN

- The kernel in 1D-CNN can catch the short-term feature in sequence data.




```
model = keras.models.Sequential()

model.add(Conv1D(1, kernel_size = 5, input_shape = (128, 3)))
#-------------------------------------------------------------------------------
#Implement Conv2D as Conv1D
model.add(Conv2D(1, kernel_size = (5, 3), input_shape = (128, 3, 1), padding = 'valid' ))
```

How to design a good LSTM

The Hyper Parameters

- Input dimensions
- Output dimensions
- Activation functions

The Random Variables

- We need to train:
  - Forget gate
  - Input gate
  - Output gate

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

$$\tilde{C}_t = \tanh(W_c x_t + U_C h_{t-1} + b_C)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

- Dimension of each weight matrix:
  - W: $R^{dim_h \times dim_x}$
  - U: $R^{dim_h \times dim_h}$
  - b: $R^{dim_h \times dim_t}$

dim_x: dimension of the input

dim_h: dimension of the output

dim_t: length of the input in each block

The Number of variables

- For Forget gate $f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$
  - $dim_h \times dim_x + dim_h \times dim_h + dim_h \times dim_t$
    $= dim_h \times (dim_x + dim_h + dim_t)$

W: $R^{dim_h \times dim_x}$ dim_x: dimension of the input

U: $R^{dim_h \times dim_h}$ dim_h: dimension of the output

b: $R^{dim_h \times dim_t}$ dim_t: length of the input in each block

- We have to train f_t, i_t, $\tilde{C}_t$, o_t

$$4 \times dim_h \times (dim_x + dim_h + dim_t)$$

The Number of Variables

- Suppose that
  - The input dimension is 30
  - Data length in each block is 10
  - The output dimension is 20

- The total number of variables in each block is

$$4*20*(30+20+10)=4800$$

- The total number of variables is

$$N*4800, N = \text{number of blocks}$$

Which Data is Good for LSTM

- The data have long-term time dependency
  - Weather prediction
  - Time series prediction
  - Sentence analysis
  - Text

Human activity recognition model training experiment

Lab. 2.