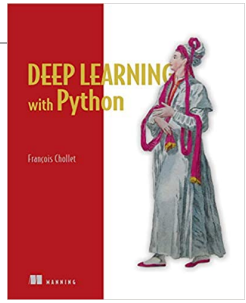
Deep Learning for Text

Neural Network, RNN, LSTM

CHIEN CHIN CHEN



Deep Learning with Python - Manning Publications
François Chollet, 2017

Preface

This subject is intended as a primer for those with no experience in neural networks.

- We will not go into the math of the underlying algorithm, *backpropagation*.

Instead, we first present basic components of neural networks.

- *Neurons*, *fully connected networks*.

And then talk about some advanced network structure.

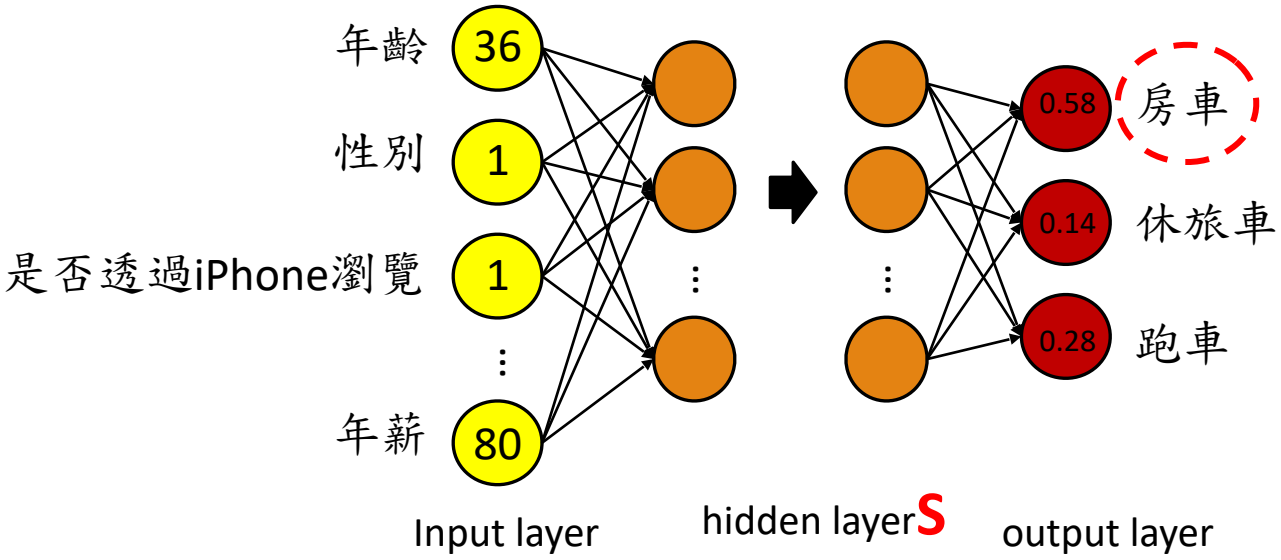
- *Embedding layers*, *recurrent neural networks*, *long short-term memory networks*.

Examples of IMDB review sentiment analysis will be provided to practice these well-known networks.

Anatomy of a Neural Network (1/4)

What is a neural network?

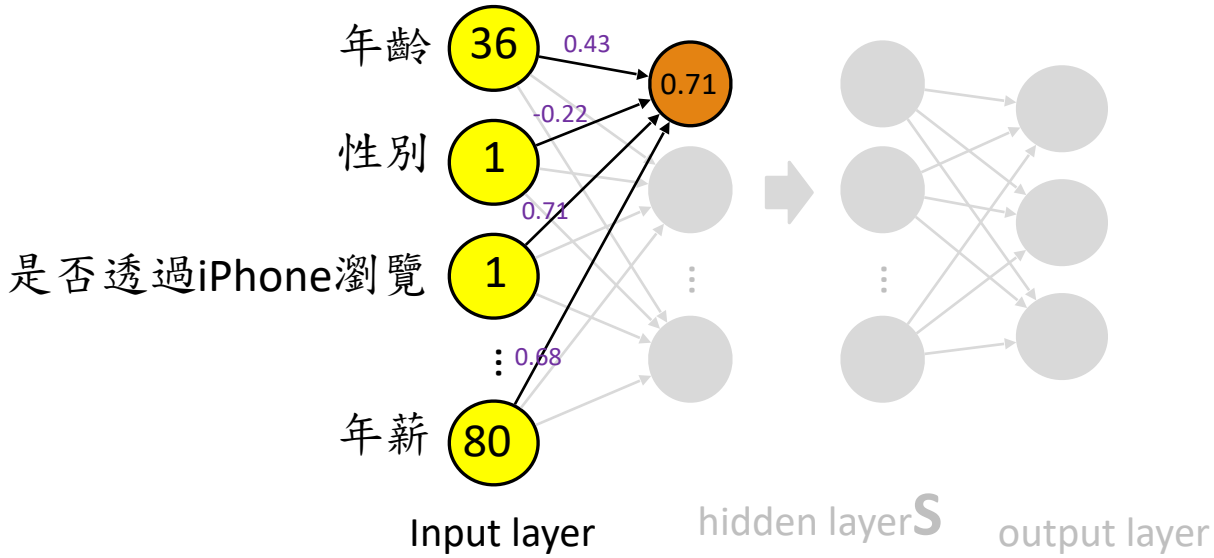
- A network normally consists of **layers** and **weights**.
- Analyze the input (data features) to predict or estimate something



Anatomy of a Neural Network (2/4)

The prediction is very similar to regression

- Aggregate **A LOT OF** regression functions to make predictions.
- Layers extract (transform) features iteratively.
- Is a kind of **feature engineering**, but features are learned (acquired) automatically.



Anatomy of a Neural Network (3/4)

One big thing is to determine the weights of a network.

- Fortunately, it is a **supervised learning** process.
- The weights are **learned automatically** from **training data** through some sophisticated math.

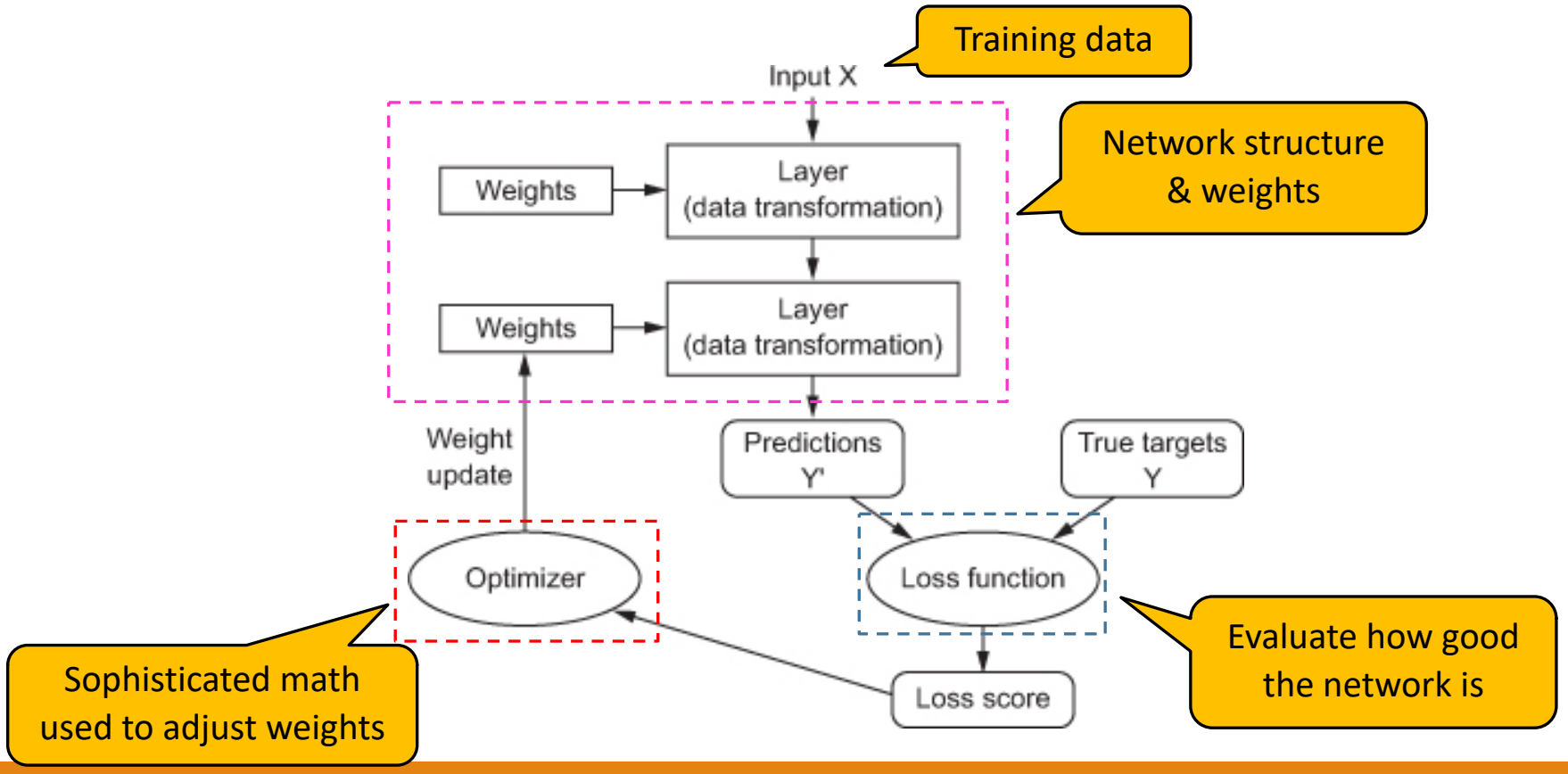
Another big thing is to specify the **structure of your network**.

- Fully connected, recurrent, encoder-decoder, etc.
- Picking the right structure is more an art than a science…

年齡	性別	iPhone	…	年薪	車款
36	1	1	…	80	1
44	0	1	…	135	2
…					
23	1	0	…	48	2

Anatomy of a Neural Network (4/4)

The overview of neural network **learning**:



IMDB Sentiment Dataset

The dataset comes packaged with `Keras`.

- Has been preprocessed – the reviews (sequences of words) have been turned into sequence of integers (unique word ids).

50,000 polarized reviews.

- 25,000 for training and 25,000 for testing.
- Each set consists of 50% negative and 50% positive reviews.

The sentiment detection task is simply a binary classification.

- Positive/negative reviews.

Neuon (1/4)

In the 1950s by Frank Rosenblatt, the **perceptron** offered a novel algorithm for finding patterns in data.

- To mimic the operation of a living **neuron cell**.
- The cell **weights** incoming signals when deciding when to fire.
- When the cell reaches a certain **level** of charge, it fires, sending an electrical signal out.

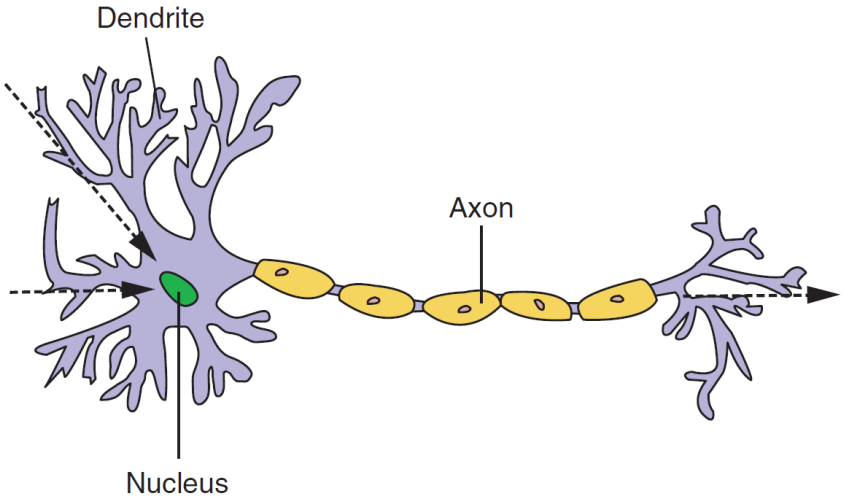


Figure 5.1 Neuron cell

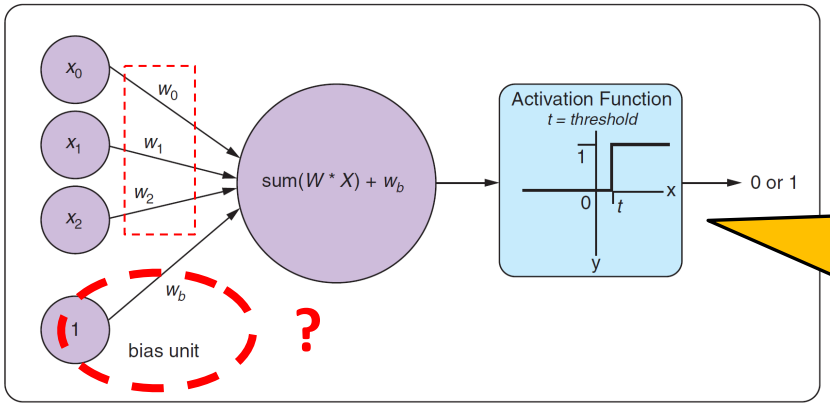
Neuon (2/4)

Rosenblatt's research was to teach a machine to recognize images.

- E.g., hot dog/not hot dog.

Features of the input image were each a small subsection of the image.

A **weight** was assigned to each feature, to measure the importance of each feature.



Activation function: once the weighted sum is above a certain **threshold**, the perceptron outputs 1!!

Here, the activation function is a **step function**.

Neuon (3/4)

The reason for having the **bias** is that you need the perceptron to be resilient to inputs of all zeros.

- Sometime, you need your network not to output 0 when facing inputs of 0.
- With this bias, you have no problem.

Of course, the weights will be learned by training examples; rather than manually assign!!

Neuon (4/4)

The base unit of neural networks is the **neuron**!!

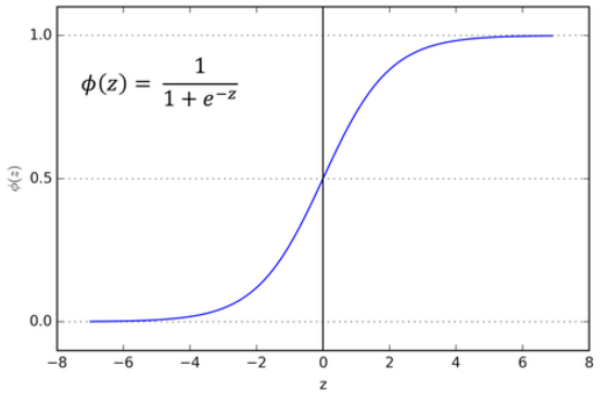
Some people regard perceptron is a special case of neuron that uses step function as the activation function.

Many prefer using continuously **differentiable nonlinear** function (e.g., **sigmoid**) as the activation function of neuron.

- To be able to model difficult problems.
- Allowing backpropagation training.

Sigmoid function:

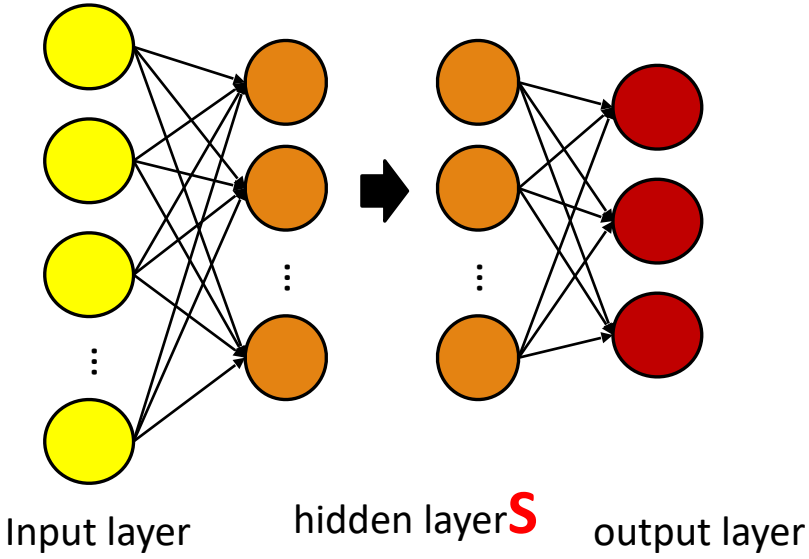
$$S(x) = \frac{1}{1 + e^{-x}}$$



Fully Connected Network (1/11)

The most popular neural network structure

- Each input element has a connection to every neuron in the next layer.
- And the outputs of layer *k* become the input of layer *k*+1, and so on.

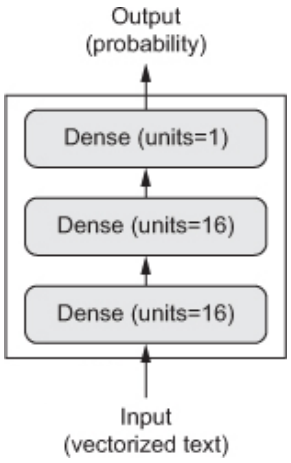


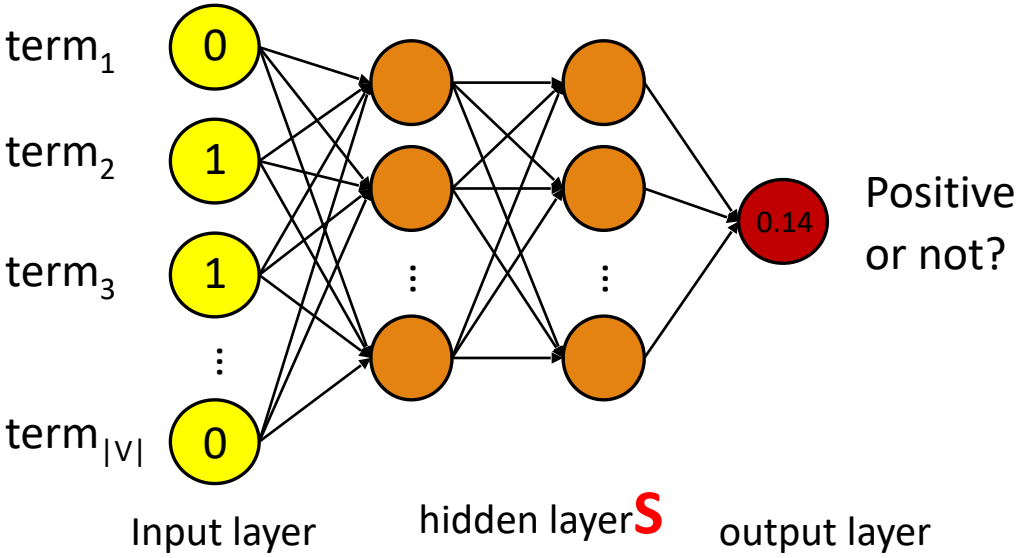
Fully Connected Network (2/11)

To process text information, we need to **vectorize** the input text.

- Multi-hot encoding/Embedding layer

Let's start with the multi-hot encoding and a simple **three-layer** structure.





Fully Connected Network (3/11)

```
In [1]: from keras.datasets import imdb

In [28]: (train_data, train_labels),
         (test_data, test_labels) = imdb.load_data(num_words=10000)

In [twenty three]: train_data[0]
                   973,
                   1622,
                   1385,
                   65,
                   458,
                   4468,
                   66,
                   3941,
                   4,
                   173,
                   36,
                   256,
                   5
```

Loading the IMDB dataset, only returning the top 10,000 most frequently occurring words

```
In [twenty four]: train_labels

Out[24]: array([1, 0, 0, ..., 0, 1, 0])
```

Fully Connected Network (4/11)

```
In [5]: import numpy as np
```

```
In [6]: def vectorize_sequences(sequences, dimension=10000):
            results = np.zeros((len(sequences),dimension))
            for i, sequence in enumerate(sequences):
                results[i, sequence] = 1.
            return results
```

```
In []:
```

Define a function that turns each review into a multi-hot vector.

```
In [7]: x_train = vectorize_sequences(train_data)
        x_test = vectorize_sequences(test_data)
        x_train[0]
```

```
Out[7]: array([0., 1., 1., ..., 0., 0., 0.])
```

Fully Connected Network (5/11)

In [9]:
```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16,activation='relu', input_shape=(10000,)))
# need to be 10000, to let it be a tuple, otherwise, it is a scale 10000,
model.add(layers.Dense(16, activation = 'relu'))
model.add(layers.Dense(1,activation = 'sigmoid'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type) Output Shape Param #
================================================= ===============
dense (Dense) (None, 16) 160016
_________________________________________________________________
dense_1 (Dense) (None, 16) 272
_________________________________________________________________
dense_2 (Dense) (None, 1) 17
=================================================================
Total params: 160,305
Trainable params: 160,305
Non-trainable params: 0
```

In [10]:
```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
```

Fully Connected Network (6/11)

Loss function: a.k.a. ***objective function***, measures how well the learned model (weights) fits the training data.

- **Binary crossentropy** for binary classification problem

$$\text{Loss} = -\frac{1}{\text{output size}} \sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

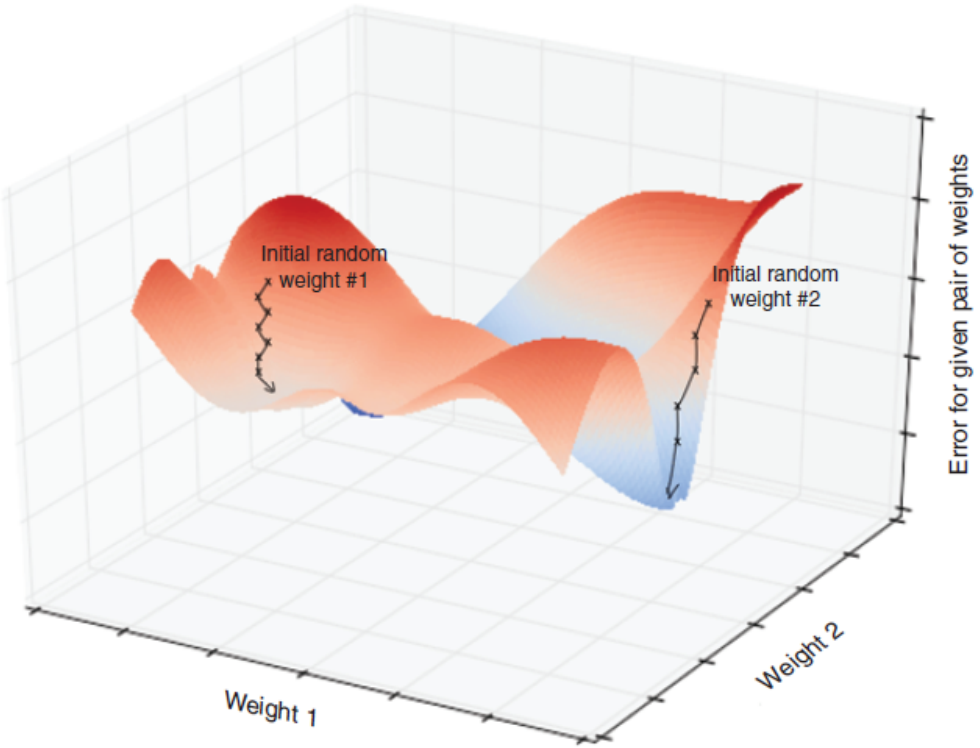
  - y_i: 1/0, the target value of *i*-th training example.
  - $\hat{y}_i$: the output of the model regarding *i*-th training example.

- **Categorical crossentropy** for multi-class classification problem.
- **MSE** for regression problem.

Fully Connected Network (7/11)

Optimizer: approach to adjust (update) network weights.

- SGD, **RMSprop**, Adam, … etc; most of them are based on **gradient descent**.



Fully Connected Network (8/11)

We split training data into training/validation sets.

- The validation set is used to search appropriate **hyper-parameters**.
  - E.g., activation function, number of hidden nodes, number of layers ….

```
In [11]: x_val = x_train[:10000]
         partial_x_train = x_train[10000:]

         y_val = train_labels[:10000]
         partial_y_train = train_labels[10000:]
```

```
In [12]: history = model.fit(partial_x_train, partial_y_train,
                            epochs=20,
                            batch_size = 512,
                            validation_data = (x_val,y_val))
```

```
Epoch 1/20
30/30 [==============================]-1s 26ms/step-loss: 0.5190-accuracy: 0.7760-val_loss : 0.3935-val_accuracy:
0.8715
Epoch 2/20
30/30 [==============================]-1s 23ms/step-loss: 0.3138-accuracy: 0.9026-val_loss : 0.3204-val_accuracy:
0.8778
Epoch 3/20
30/30 [==============================]-1s 21ms/step-loss: 0.2302-accuracy: 0.9289-val_loss : 0.3065-val_accuracy:
```

Fully Connected Network (9/11)

`epochs=20`

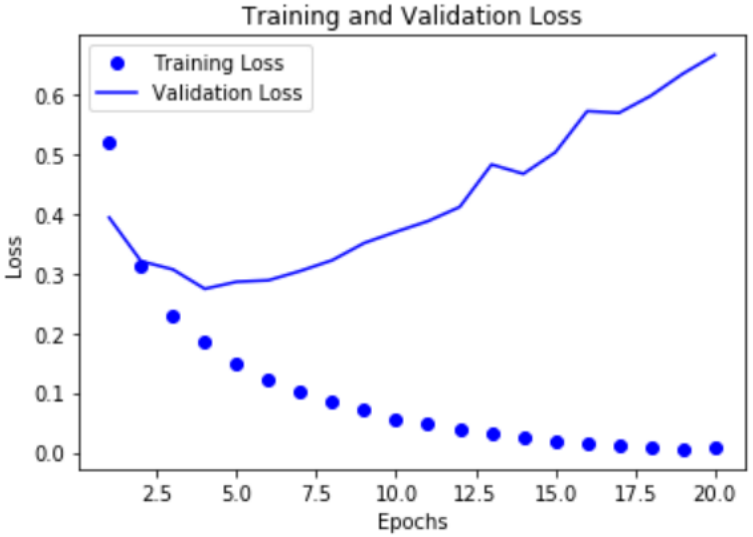
- Optimizers adjust network weights iteratively.
- After all training data has gone through the network **once**, and the errors are used by optimizers to update weights, we call this an **epoch** of the neural network training.
- Note that a large epochs could make your model **overfitting**!!

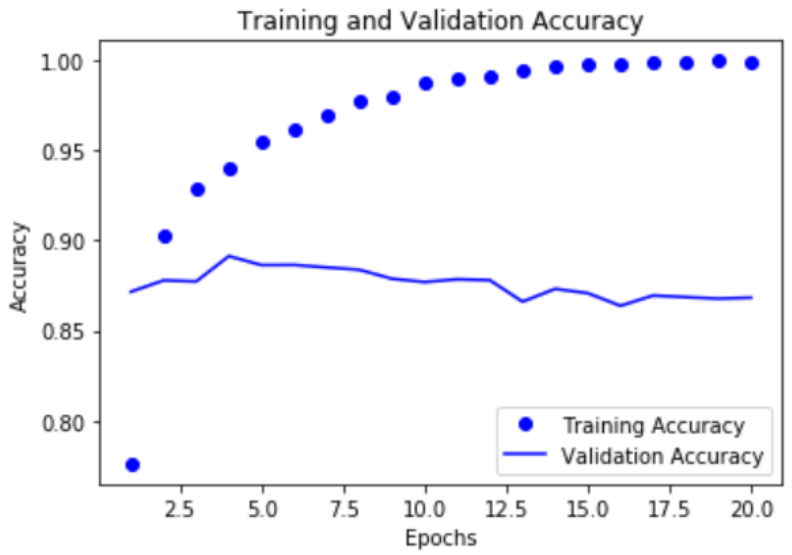
`batch_size=512`

- The size of a random subset of the training data.
- To speed up training and to avoid getting stuck in a local optima.
- Default is 32.

Fully Connected Network (10/11)

Example of **overfitting** the training dataset





Fully Connected Network (11/11)

In [15]:
```
final_model = models.Sequential()
final_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
final_model.add(layers.Dense(16, activation='relu'))
final_model.add(layers.Dense(1,activation='sigmoid'))

final_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
final_model.fit(x_train,train_labels, epochs=4, batch_size=512)
```

```
Epoch 1/4
49/49 [==============================]-0s 8ms/step-loss: 0.4435-accuracy: 0.8278
Epoch 2/4
49/49 [==============================]-0s 9ms/step-loss: 0.2574-accuracy: 0.9091
Epoch 3/4
49/49 [==============================]-0s 10ms/step-loss: 0.1984-accuracy: 0.9285
Epoch 4/4
49/49 [==============================]-0s 7ms/step-loss: 0.1657-accuracy: 0.9422
```

In [16]:
```
testing_result = final_model.evaluate(x_test, test_labels)
# or using final_model.predict(x_test) to predict the class of the testing data
```

```
782/782 [==============================]-1s 1ms/step-loss: 0.3020-accuracy: 0.8800
```

Embedding Layer (1/10)

The dimension of multi-hot vectors is too high, and too sparse.

- In the last example, vector dimension is 10,000!!

Word embeddings are low-dimensional, and dense vectors.

- The dimension of word embeddings is usually less than 1,000.
- Word embeddings are **learned** from data.
  - To make synonyms embedded into similar (close) vectors.

Two ways to obtain word embeddings:

- Learn word embeddings jointly with the task you care about (e.g., the IMDB sentiment analysis).
- Using pre-trained word embeddings (e.g., Word2Vec).

Note that it is good to learn task-specific embeddings **if you have plenty of task-specific data**; pre-trained embeddings sometimes would be too general to specific tasks.

Embedding Layer (2/10)

The following example shows how to add **embedding layers** into your neural network.

One difficulty – different review lengths; but the size of network inputs needs to be fixed!!

- Here, we simply examine the first 100 words of each review.

In [1]:
```
from keras.datasets import imdb
from keras import preprocessing

max_unique_tokens = 10000 # 取前10000個字來建立特徵維度
max_DocLen = 100 # 限定文件長度度為前100字

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words= max_unique_tokens)
```

Embedding Layer (3/10)

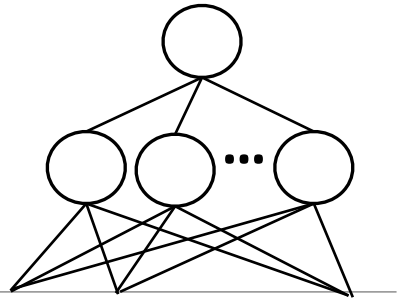
```
In [2]: print(len(train_data[0]))
        print(len(test_data[0]))

        218
        68
```

```
In [3]: train_data[0]

Out[3]: [1,
         14,
         22,
         16,
         43,
         530,
         973,
         1622,
         1385,
         65
```

```
In [7]: test_data[0]

Out[7]: array([   1,  591,  202,   14,   31,    6,  717,   10,   10,    2,    2,
                  5,    4,  360,    7,    4,  177, 5760,  394,  354,    4,  123,
                  9, 1035, 1035, 1035,   10,   10,   13,   92,  124,   89,  488,
               7944,  100,   28, 1668,   14,   31,   23,   27, 7479,   29,  220,
                468,    8,  124,   14,  286,  170,    8,  157,   46,    5,   27,
                239,   16,  179,    2,   38,   32,   25, 7944,  451,  202,   14,
                  6,  717,    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], dtype=int32)
```

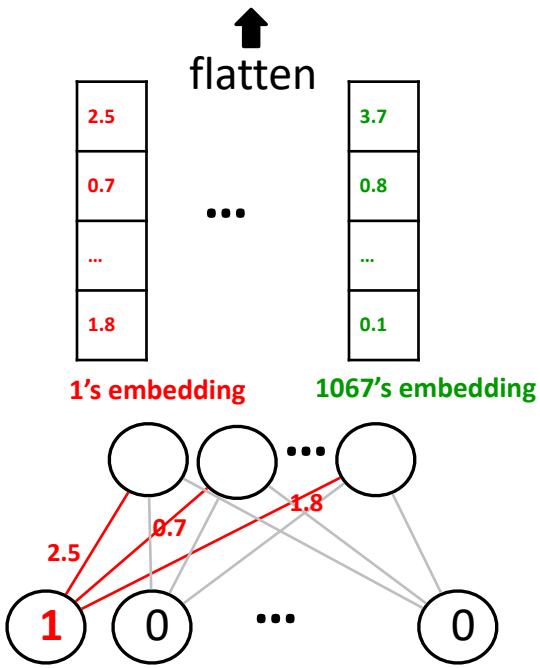
```
In [5]: train_data = preprocessing.sequence.pad_sequences(train_data, maxlen=max_DocLen, truncating='post', padding='post')
        test_data = preprocessing.sequence.pad_sequences(test_data, maxlen=max_DocLen, truncating='post', padding='post')

        print(train_data.shape)
        print(test_data.shape)

        (25000, 100)
        (25000, 100)
```

Embedding Layer (4/10)




```
In [8]: from keras.models import Sequential
        from keras.layers import Flatten, Dense, Embedding

        model = Sequential()
        model.add(Embedding(max_unique_tokens, 300, input_length=max_DocLen))
        model.add(Flatten())
        model.add(Dense(16, activation = 'relu'))
        model.add(Dense(1, activation='sigmoid'))

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

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 100, 300)          3000000
_________________________________________________________________
flatten (Flatten)            (None, 30000)             0
_________________________________________________________________
dense (Dense)                (None, 16)                480016
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 3,480,033
Trainable params: 3,480,033
Non-trainable params: 0
```




```
In [12]: testing_result = model.evaluate(test_data, test_labels)
         # or using final_model.predict(x_test) to predict the class of the testing data
```

```
782/782 [==============================] - 2s 2ms/step - loss: 0.4876 - acc: 0.8033
```

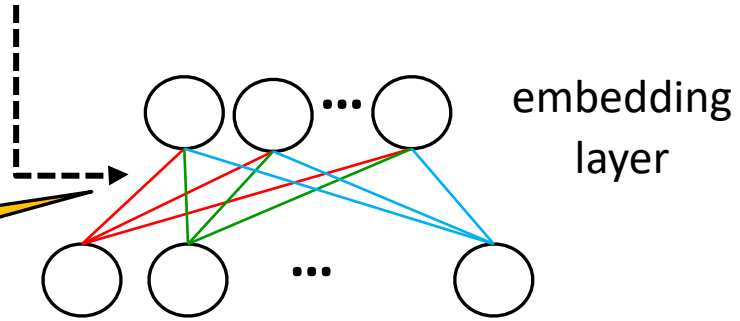
Embedding Layer (5/10)

The weights of the embedding layer are initially random!!

During training, the weights are gradually adjusted to meet the goal of the downstream task (e.g., sentiment analysis).

Once fully trained, the **embeddings** could be applied to other similar tasks.

- Pre-trained embeddings.
- E.g., product review analysis



The embeddings can be represented as a **matrix** of *#unique_token* x *embedding_dimension*, that is 10000x300

2.5	0.7	...	1.8
0.4	3.3	...	0.7
...			
1.2	1.7	...	2.2

Embedding Layer (6/10)

When training word embeddings, the embedding quality depends on the size of your training set.

- If you have **little** training data, you might not be able to learn appropriate task-specific embeddings.

An alternative is to **load embedding vectors from a precomputed embedding space** that captures generic language structures.

- Remember Word2Vec?

Below, we demonstrate how to load Word2Vec word embedding into a Keras `Embedding` layer.

Embedding Layer (7/10)

Let’s load Word2Vec embedding vectors first.

In [8]:
```
import gensim
import numpy as np

w2v_model = gensim.models.KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin.gz', binary=True)
w2v_dimension = 300

w2v_embedding_matrix = np.zeros((max_unique_tokens,w2v_dimension))
print(w2v_embedding_matrix.shape)
print(w2v_embedding_matrix)
```

```
(10000, 300)
[[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.]]
```

Embedding Layer (8/10)

Next, build up the **embedding matrix** that we will load into an `Embedding` layer for our neural network model.

```
In [9]: word_index = imdb.get_word_index()

        for word, wid in word_index.items():
            if wid < max_unique_tokens:
                try:
                    embedding_vector = w2v_model[word]
                except KeyError:
                    embedding_vector = np.zeros((w2v_dimension,))

                w2v_embedding_matrix[wid] = embedding_vector


        print(w2v_embedding_matrix)
```

```
[[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.08007812  0.10498047  0.04980469 ...  0.00366211  0.04760742
  -0.06884766]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 ...
 [ 0.31835938 -0.08056641 -0.11816406 ... -0.21875     0.02770996
   0.26171875]
 [ 0.13378906  0.1640625  -0.17382812 ... -0.25585938 -0.11669922
   0.08984375]
 [-0.12353516  0.21777344 -0.578125   ...  0.12695312  0.20507812
  -0.00543213]]
```

Not every word can be found in a pre-trained model!!

Zero embedding if a word is unknown to Word2Vec!!

Embedding Layer (9/10)

In [10]:
```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Embedding

model = Sequential()
model.add(Embedding(max_unique_tokens, w2v_dimension, input_length=max_DocLen))
model.add(Flatten())
model.add(Dense(16, activation = 'relu'))
model.add(Dense(1, activation='sigmoid'))

model.layers[0].set_weights([w2v_embedding_matrix])
model.layers[0].trainable = False

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


model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 100, 300)          3000000
_________________________________________________________________
flatten (Flatten)            (None, 30000)             0
_________________________________________________________________
dense (Dense)                (None, 16)                480016
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 3,480,033
Trainable params: 480,033
Non-trainable params: 3,000
```

In [12]:
```
testing_result = model.evaluate(test_data, test_labels)
# or using final_model.predict(x_test) to predict the class of the testing data
# 可以比較 with/without pretrain model的效能差異
```

```
782/782 [==============================] - 1s 2ms/step - loss: 0.4673 - acc: 0.7843
```

??

Embedding Layer (10/10)

When **lots** of training data is available, models with a task-specific embedding are generally more powerful (accurate) than those with pre-trained embeddings.

- In this example, we train models with 25,000 reviews.
- That is why the model with pre-trained embeddings is inferior to the first task-specific embedding model.

- *Self-practice: randomly select 200 reviews for training, and compare the two models.*

Recurrent Neural Networks (1/6)

A network structure useful for **sequential data**.

- E.g., time series and TEXT – token by token!!

Offering a kind of **memory mechanism** that helps process data sequentially.

- Like human being, enable the model to observe review tokens one by one, and gradually realize the emotion of the author.

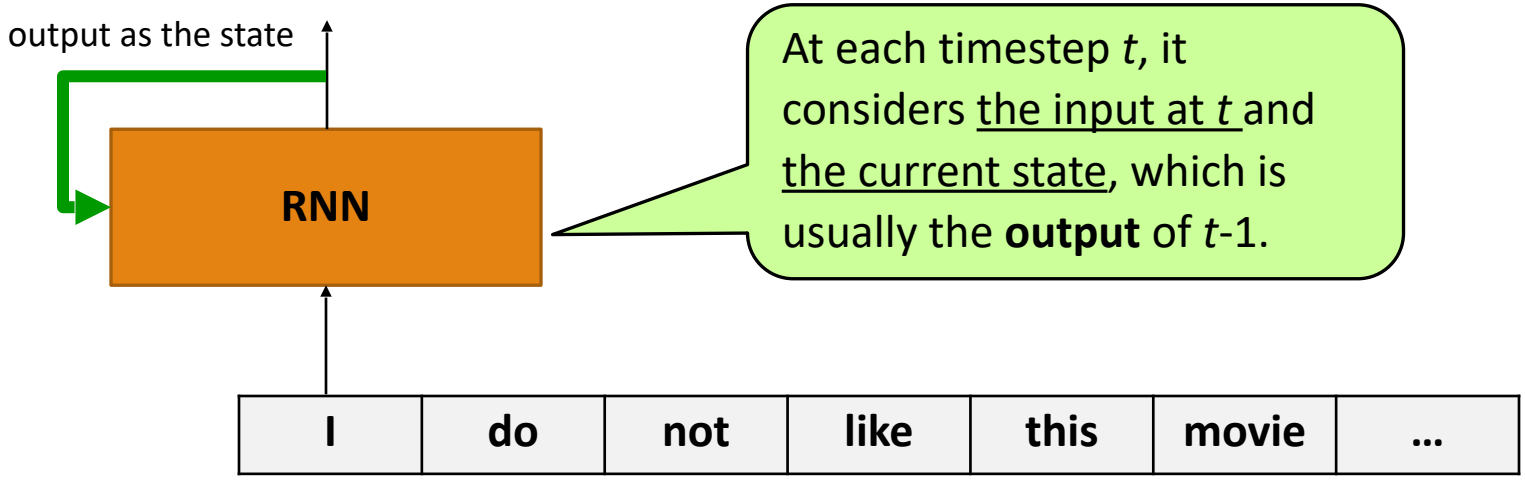
By contrast, the neural networks seen so far keep no **state** between inputs.

- Each input is processed independently.

Recurrent Neural Networks (2/6)

The simplest recurrent neural network (RNN) is a type of neural network that has an **internal loop**.

- It processes sequences by iterating through the sequence elements
- Maintaining a **state** containing (memorizing) information relative to what it has seen so far.
- There are many different RNNs (e.g., ***LSTM***) fitting this definition, but with more sophisticated structures.



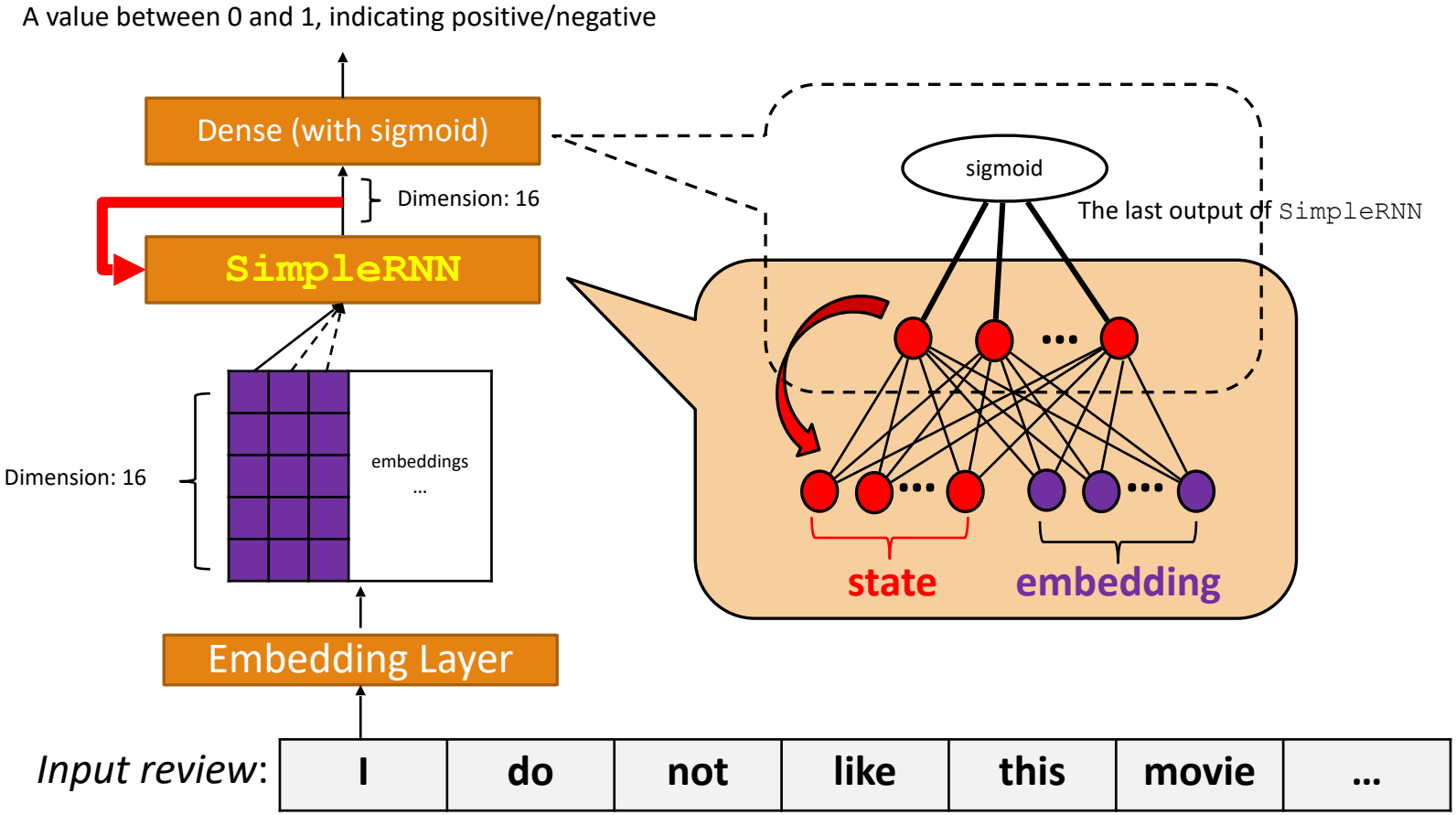
Recurrent Neural Networks (3/6)

`Keras` offers the `SimpleRNN` layer to help us construct a simple RNN.

In the following example, we sequentially process an IMDB review (i.e., token by token).

- Each token is first converted into an embedding (dim. 16).
- Then, the `SimpleRNN` layer processes all the embeddings one by one.
- Last, **the final output** of the RNN layer is fed into a dense layer to predict (classify) the sentiment of the review.

Recurrent Neural Networks (4/6)



Recurrent Neural Networks (5/6)

In [8]:
```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(max_unique_tokens, 16))
model.add(SimpleRNN(16))
model.add(Dense(1, activation='sigmoid'))

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

NO Flatten()!!

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 16)          160000
_________________________________________________________________
simple_rnn (SimpleRNN)       (None, 16)                528
_________________________________________________________________
dense (Dense)                (None, 1)                 17
=================================================================
Total params: 160,545
Trainable params: 160,545
Non-trainable params: 0
_________________________________________________________________
```

Recurrent Neural Networks (6/6)

In [9]:
```
history = model.fit(train_data, train_labels,
                    epochs=4,
                    batch_size = 512)
```

```
Epoch 1/4
49/49 [==============================] - 1s 24ms/step - loss: 0.6044 - acc: 0.6842
Epoch 2/4
49/49 [==============================] - 1s 24ms/step - loss: 0.4726 - acc: 0.8109
Epoch 3/4
49/49 [==============================] - 1s 27ms/step - loss: 0.4022 - acc: 0.8500
Epoch 4/4
49/49 [==============================] - 1s 26ms/step - loss: 0.3501 - acc: 0.8723
```

In [12]:
```
testing_result = model.evaluate(test_data, test_labels)
# or using final_model.predict(x_test) to predict the class of the testing data
```

```
782/782 [==============================] - 2s 3ms/step - loss: 0.4439 - acc: 0.8073
```

LSTM (1/4)

One major shortcoming of simple RNN:

- Unable to retain old information in the current state **after running many timesteps**.
- In other words, long-term dependencies are hard to keep.

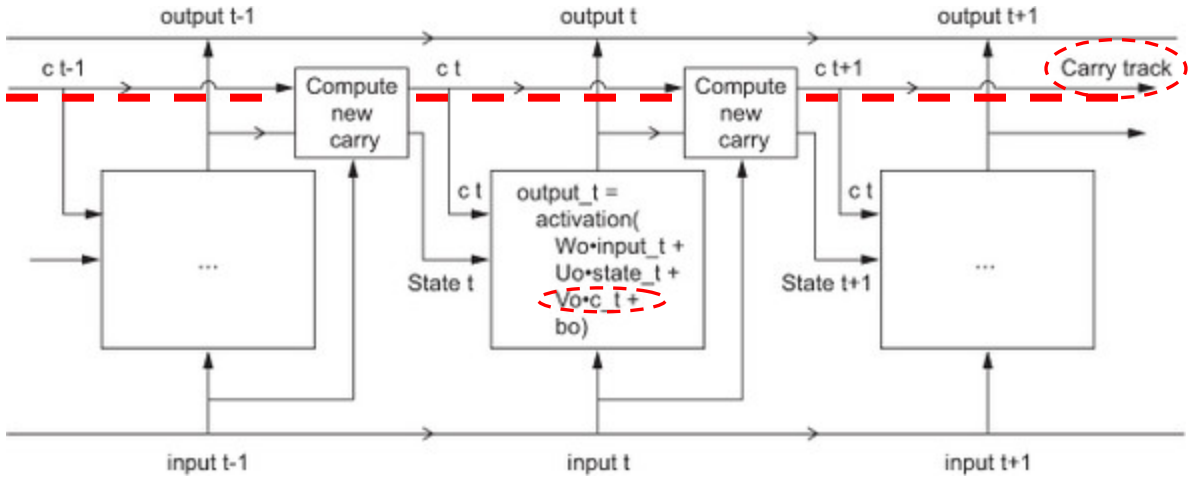
Long Short-Term Memory (**LSTM**) develops a way to carry information across many timesteps.

- A **conveyor belt (or carray track)** running parallel to the sequence you are processing.
- Information from sequence can jump onto or off the belt, and be transported to a later timestep.

LSTM (2/4)

Additional information flow – carry track:

- c_t the carry at time t.
- Connected with **the current input** and **state** to affect the state being sent to the next timestep.



LSTM (3/4)

Computing the new carry state (i.e., c_{t+1}) is a little bit complicated.

- Forget gate, input (candidate) gate, output (update) gate.
- We are not going to the detail of these gates.

Just keep in mind: the design of LSTM allows past information to be reinjected at a later time.

- To have a long term memory of what have happened.

LSTM (4/4)

In [8]:
```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Embedding, LSTM

model = Sequential()
model.add(Embedding(max_unique_tokens, 16))
model.add(LSTM(16))
model.add(Dense(1, activation='sigmoid'))

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

In [12]:
```
testing_result = model.evaluate(test_data, test_labels)
# or using final_model.predict(x_test) to predict the class of the testing data
```

```
782/782 [==============================] - 6s 7ms/step - loss: 0.4360 - acc: 0.8044
```