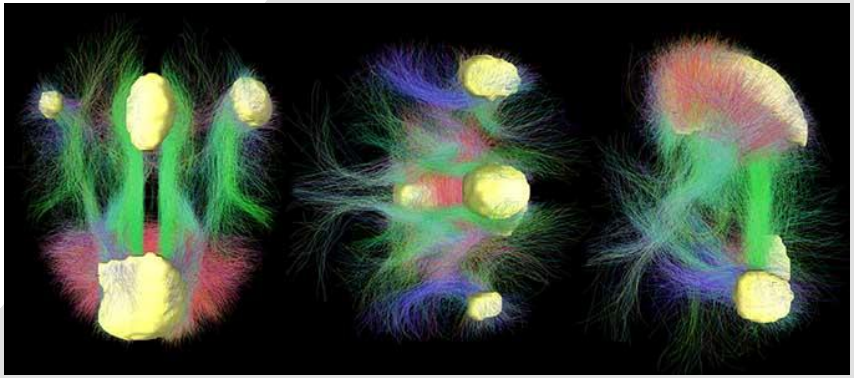
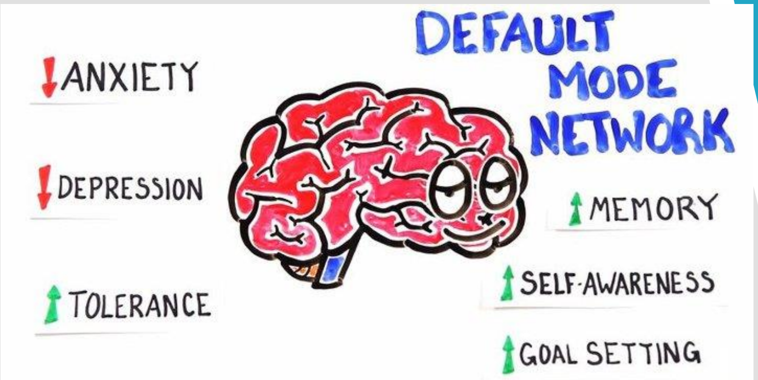
6장 *Deep learning for text and sequences*

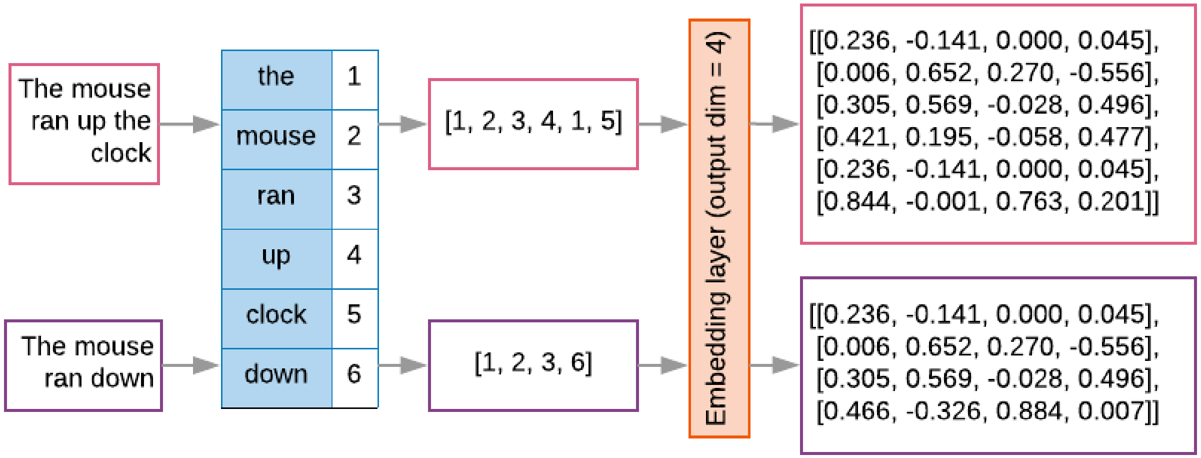
"default mode network"





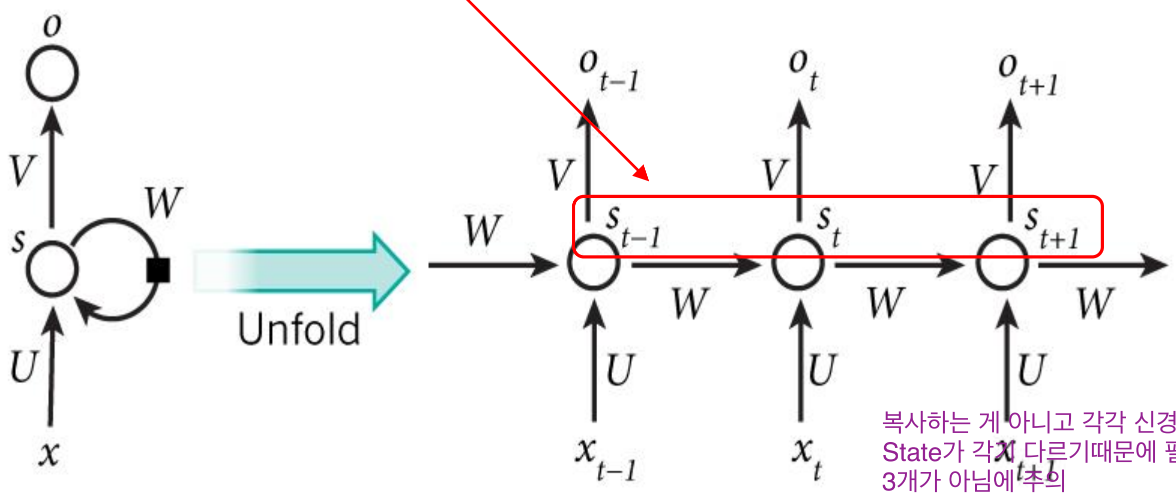
6.2Understanding recurrent neural networks

- A major characteristic of all neural networks is that they have no memory - *no state kept in between inputs*
- feedforward networks - IMDB example: an entire movie review was transformed into a single large vector and processed in one go.



6.2Understanding recurrent neural networks

- word by word (eye saccade by eye saccade) - from past information → constantly updated as new information
- 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.



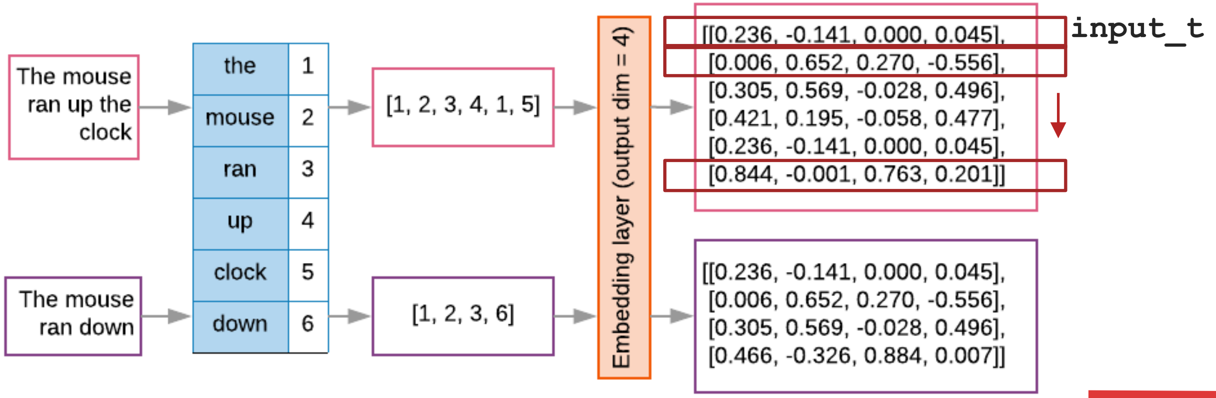
복사하는 게 아니고 각각 신경망?? 인데 (공용?)
State가 각각 다르기때문에 펼쳐놓고 보는 것(이해하기 쉽게?)
3개가 아님에 주의

6.2Understanding recurrent neural networks

- RNN takes as input a sequence of vectors - 2D tensor of size (timesteps, input_features)
- set the state for the next step to be this previous output.
- *initial state* - all-zero vector

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.
```

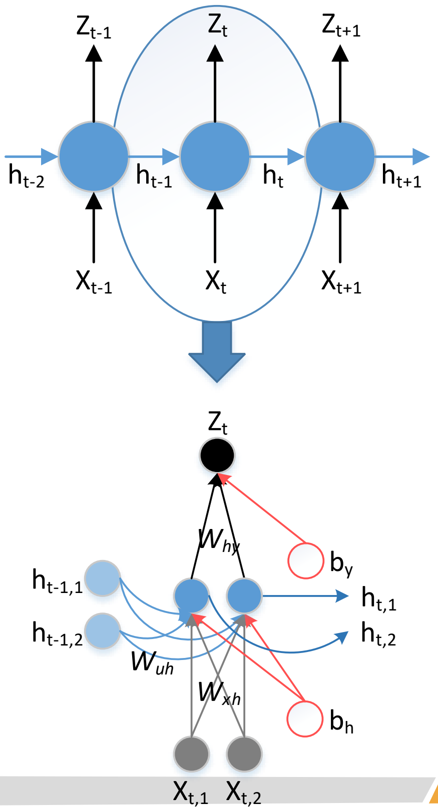


6.2 Understanding recurrent neural networks

- f: input and state → W and U, and a bias vector

Listing 6.20 More detailed pseudocode for the RNN

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

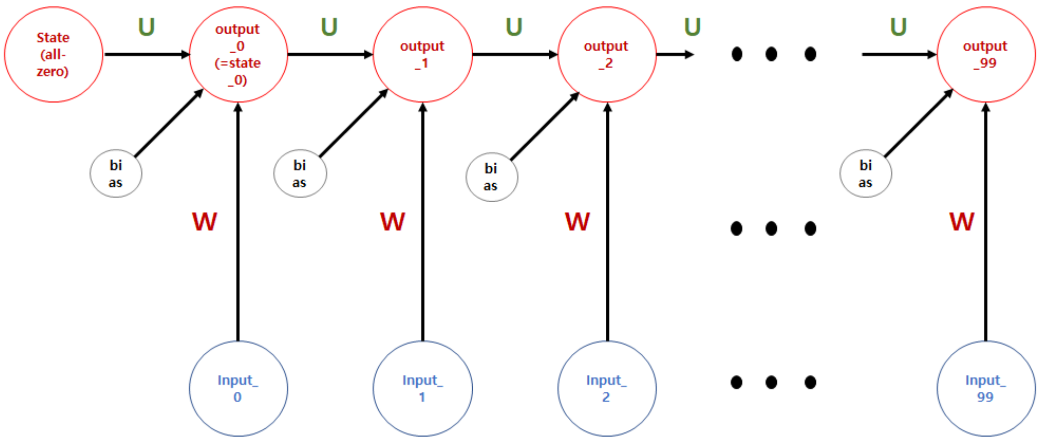


6.2Understanding recurrent neural networks

- naive Numpy implementation of the forward pass of the simple RNN.

Listing 6.21 Numpy implementation of a simple RNN

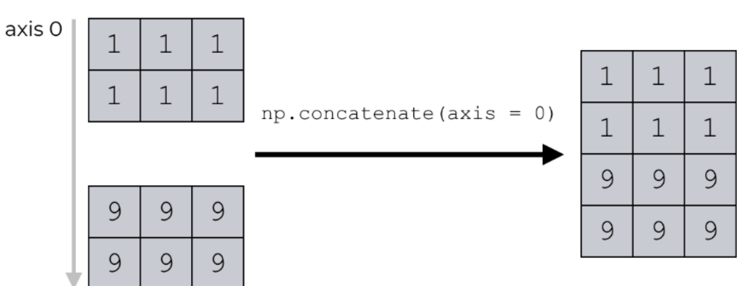
```
import numpy as np
timesteps = 100 # Number of timesteps in the input sequence
input_features = 32 # Dimensionality of the input feature space
output_features = 64 # Dimensionality of the output feature space
inputs = np.random.random((timesteps, input_features))
# Input data: random noise for the sake of the example
state_t = np.zeros((output_features,))
# Initial state: all-0 vector
W = np.random.random((output_features, input_features)) # (64,32)
U = np.random.random((output_features, output_features)) # (64,64)
b = np.random.random((output_features,)) # (64,)
# 1 random weight matrices
```

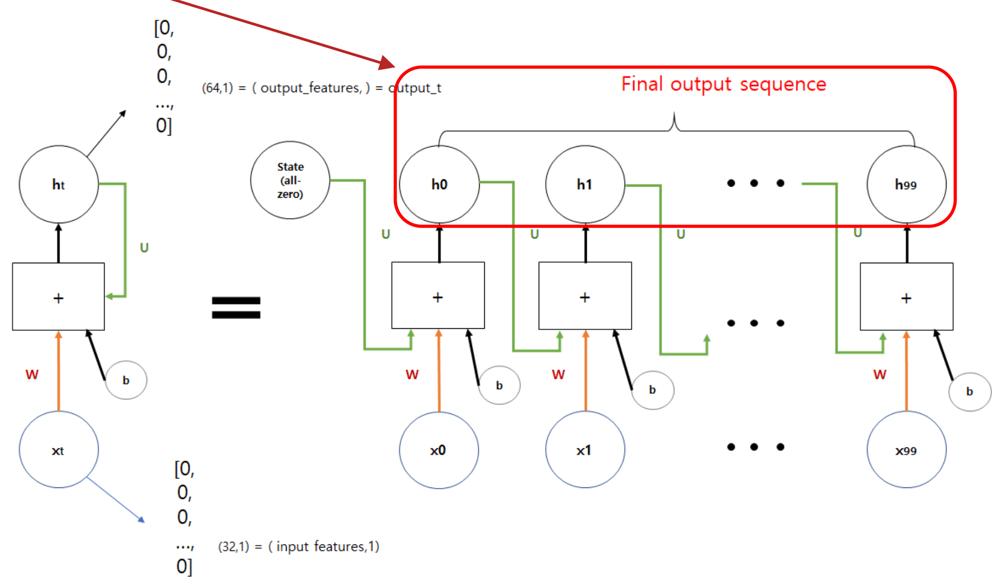


6.2 Understanding recurrent neural networks

```
successive_outputs = []
for input_t in inputs: # a vector of shape (input_features,)
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
    # Combines the input with the current state (the previous output)
    successive_outputs.append(output_t) #Stores this output in a list
    state_t = output_t
    # Updates the state of the network for the next timestep
final_output_sequence = np.concatenate(successive_outputs, axis=0)
# The final output is a 2D tensor of
# shape (timesteps, output_features), shape = (100, 64)
```

Setting `axis=0` concatenates along the row axis



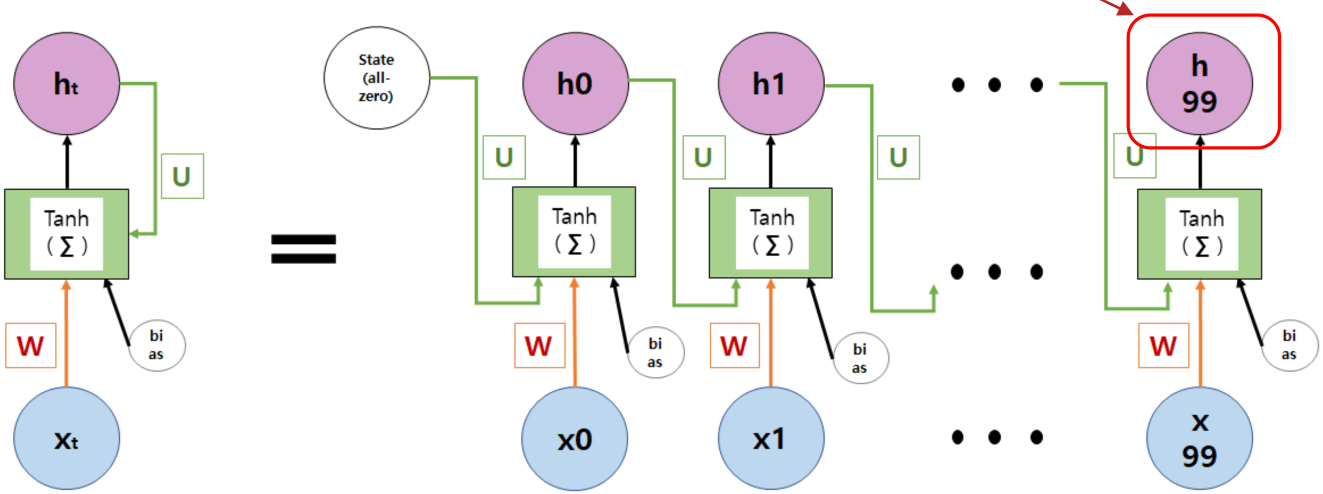


6.2Understanding recurrent neural networks

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```

NOTE The final output is a 2D tensor of shape `(timesteps, output_features)` at time `t`. Only the last output (`output_t`) at the end of the loop) is needed, because it already contains information about the entire sequence.

.



6.2 Understanding recurrent neural networks

6.2.1 A recurrent layer in Keras

- `SimpleRNN` layer:

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

- There is one minor difference: `SimpleRNN` processes batches of sequences, like all other Keras layers

`(timesteps, input_features)` →

`(batch_size, timesteps, input_features)`

- two different modes of return
  - (batch_size, timesteps, output_features) - the full sequences of successive outputs
  - (batch_size, output_features) - only the last output for each input sequence
- These two modes are controlled by the return_sequences constructor argument.

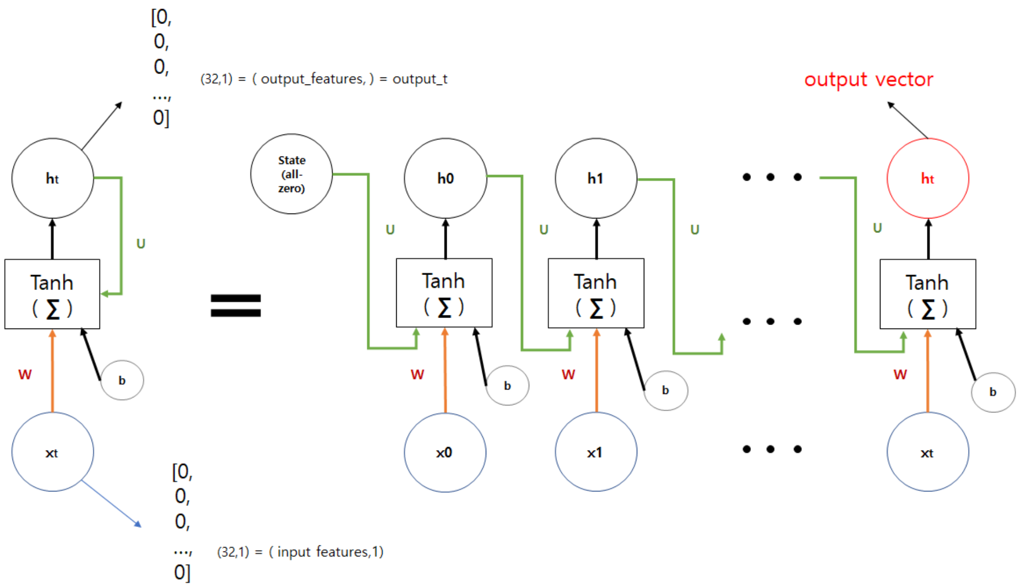
6.2 Understanding recurrent neural networks

- SimpleRNN and returns only the output at the last timestep:

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

10000은 가장 많이 사용되는 만 개
sparse는 10000개, 그걸 축약해서 32개로 embedding

```
Layer (type)                 Output Shape        Param #
==============================================================
embedding_22(Embedding)  (None,None,32)   320000
simplernn_10(SimpleRNN)  (None,32)        2080 = (32*32*2+32)
```



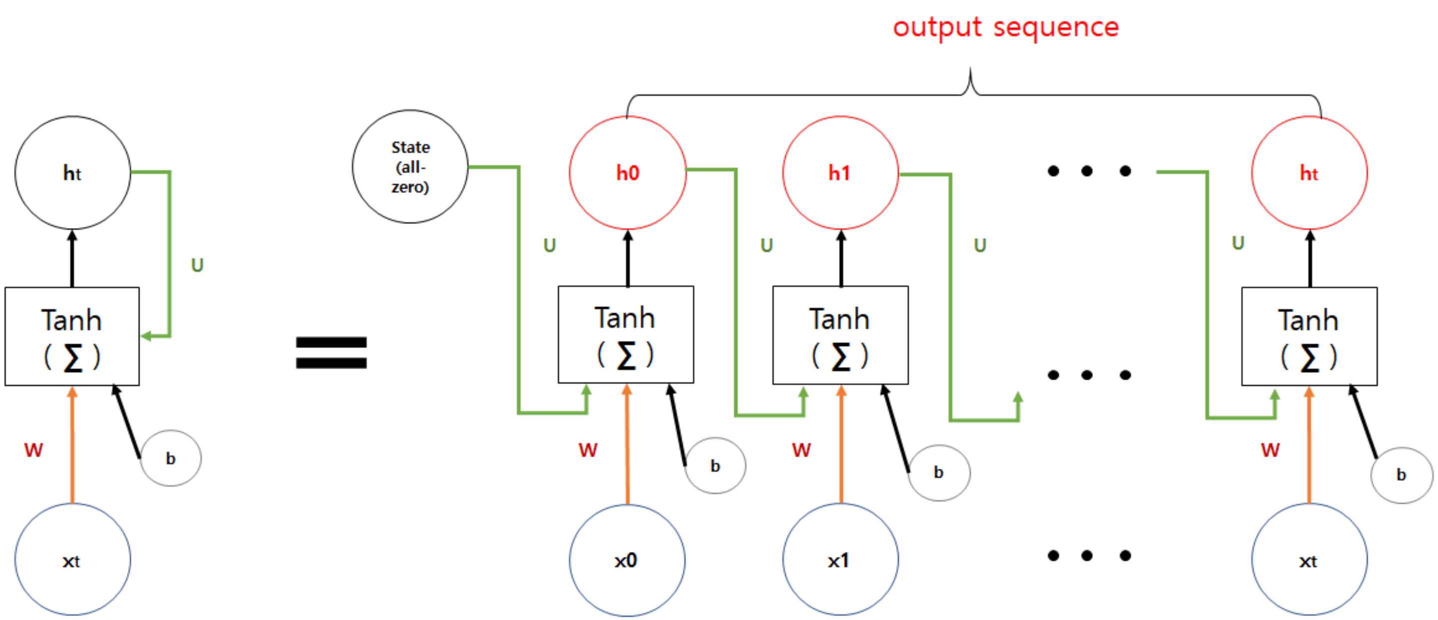
6.2 Understanding recurrent neural networks

- The following example returns the full state sequence:

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.summary()
```

32개를 각각 다 저장한다?

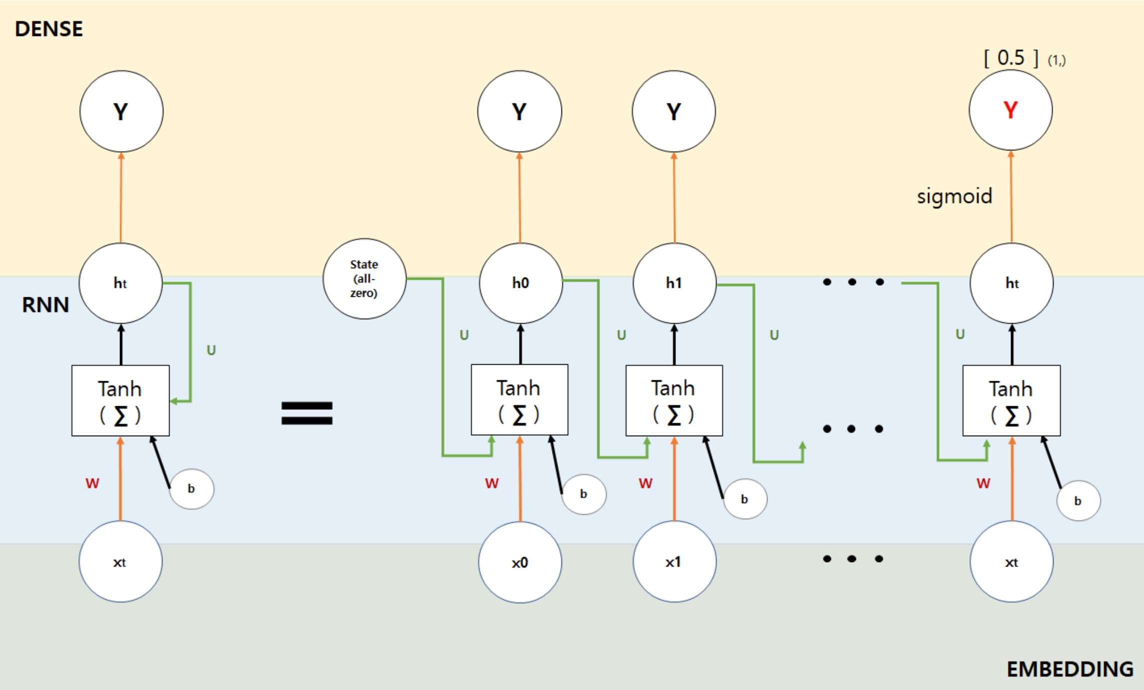
Layer (type)	Output Shape	Param #
embedding_23(Embedding)	(None,None,32)	320000
simplernn_11(SimpleRNN)	(None,None,32)	2080



6.2 Understanding recurrent neural networks

- stack several recurrent layers:

```
>>> 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)) # Last layer only returns the last output
>>> model.summary()
Layer (type)                    Output Shape            Param #
================================================================
embedding_3(Embedding)         (None, None, 32)         320000
simple_rnn_3(SimpleRNN)        (None, None, 32)
simple_rnn_4(SimpleRNN)        (None, None, 32)         2080
simple_rnn_5(SimpleRNN)        (None, None, 32)         2080
simple_rnn_6(SimpleRNN)        (None, 32)               2080
```



6.2 Understanding recurrent neural networks

- IMDB movie-review-classification problem - First, preprocess the data.

Listing 6.22 Preparing the IMDB data

```
from keras.datasets import imdb
from keras.preprocessing import sequence
max_features = 10000  # Number of words to consider as features
maxlen = 500  # Cuts off texts after this many words
batch_size = 32
(input_train, y_train), (input_test, y_test) =
            imdb.load_data(num_words=max_features)
print(len(input_train), 'train sequences') # 25000
print(len(input_test), 'test sequences') # 25000
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train shape:', input_train.shape) # (25000, 500)
print('input_test shape:', input_test.shape) # (25000, 500)
```

6.2 Understanding recurrent neural networks

- Train an Embedding layer and a SimpleRNN layer.

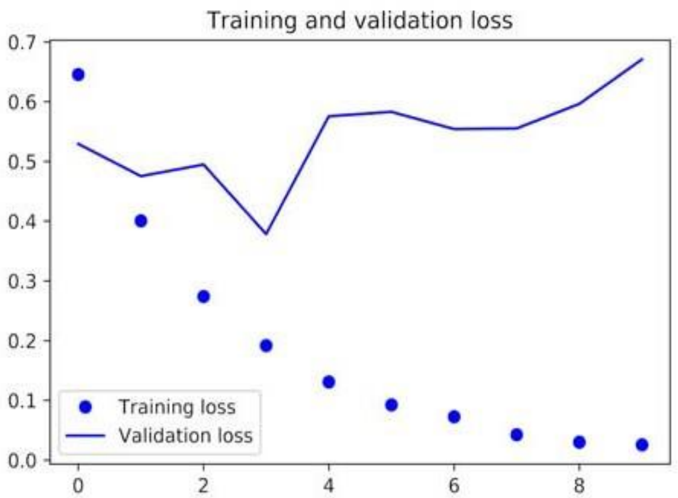
Listing 6.23 Training the model with `Embedding` and `SimpleRNN` layers

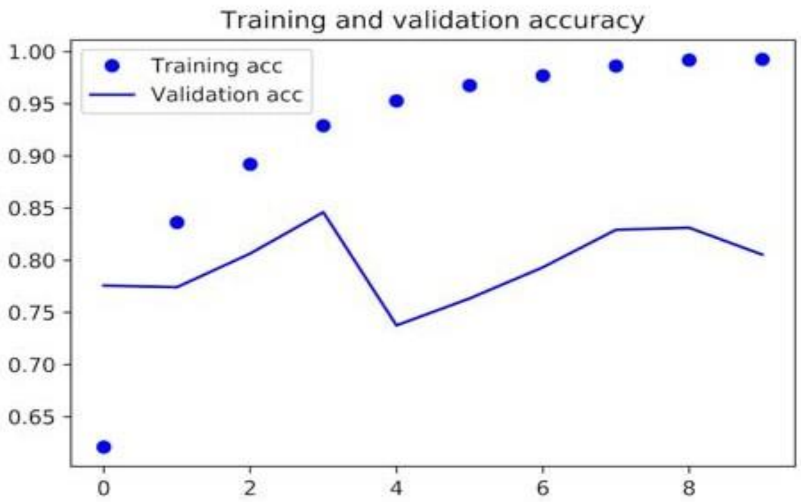
```
from keras.layers import Dense
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
        metrics=['acc'])
history = model.fit(input_train, y_train,
        epochs=10, batch_size=128, validation_split=0.2) #20%
```

6.2 Understanding recurrent neural networks

Listing 6.24 Plotting results

```
import matplotlib.pyplot as pltacc = 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()
```





6.2Understanding recurrent neural networks

- In chapter 3, test accuracy - 88%
- recurrent network - 85% validation accuracy
- Inputs only the first 500 words - less information than the earlier baseline model.
- `SimpleRNN` - No good at processing long sequences, such as text (vanishing information).
- Other types of recurrent layers perform much better.

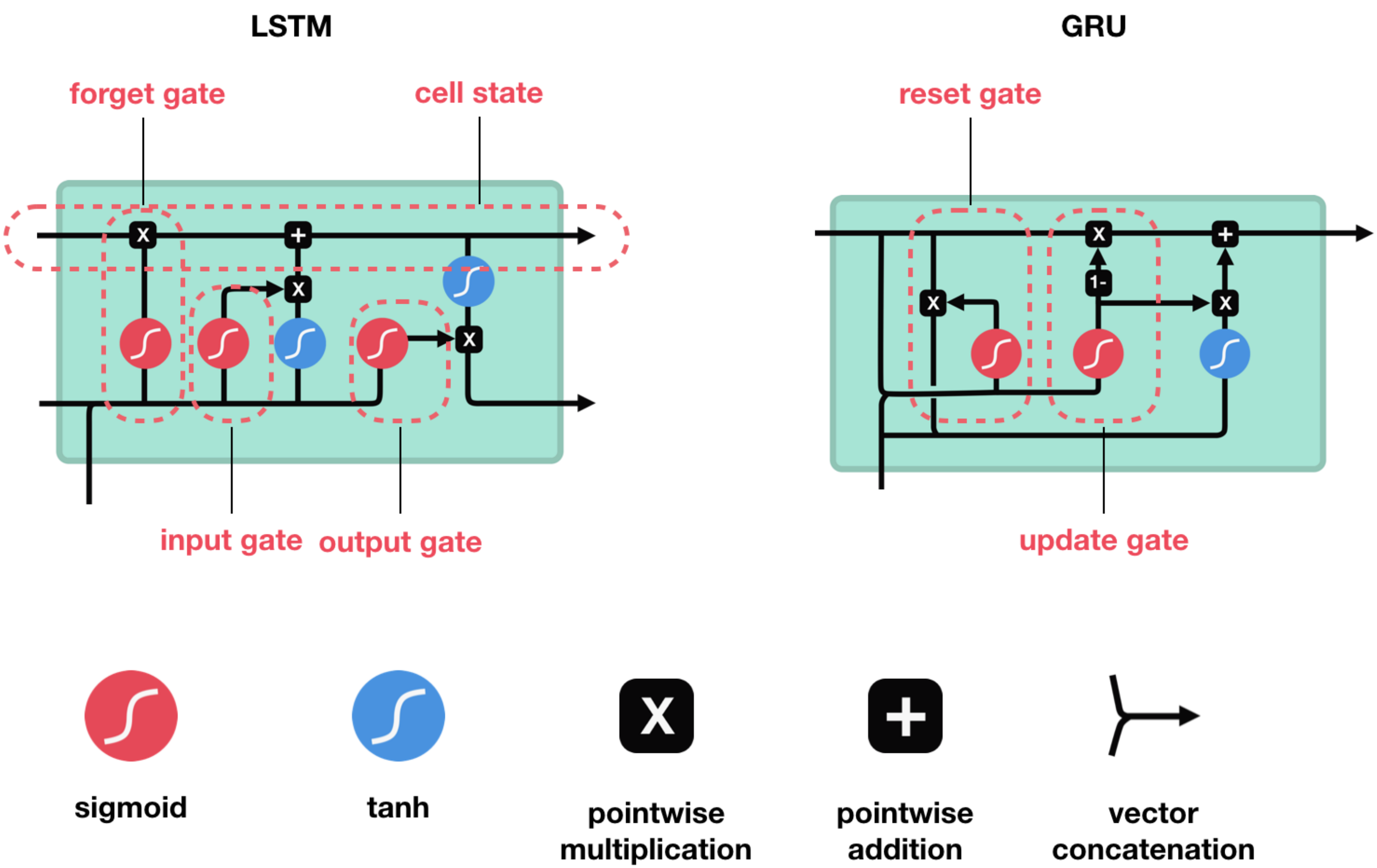
6.2 Understanding recurrent neural networks

6.2.2 A Understanding the LSTM and GRU layers

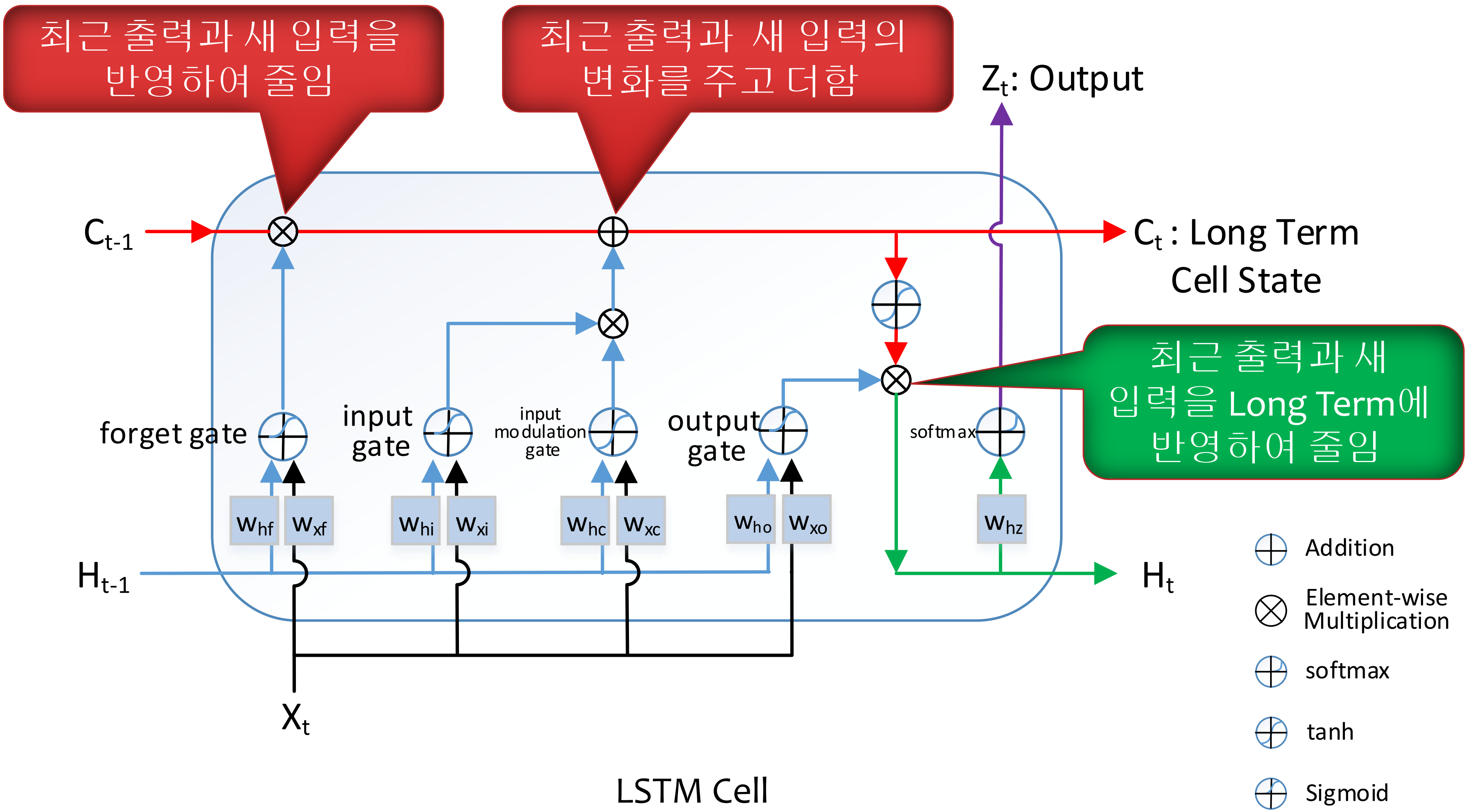
- LSTM and GRU - SimpleRNN has a major issue: long-term dependencies are impossible to learn.
- This is due to the *vanishing gradient problem*, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) studied by Hochreiter, Schmidhuber, and Bengio in the early 1990s.
- Long Short-Term Memory (LSTM) algorithm was developed by Hochreiter and Schmidhuber in 1997.
- Carry Track (C) information across many timesteps to save information for later, thus preventing older signals from gradually vanishing during processing.

LSTM-GRU Architecture - overview

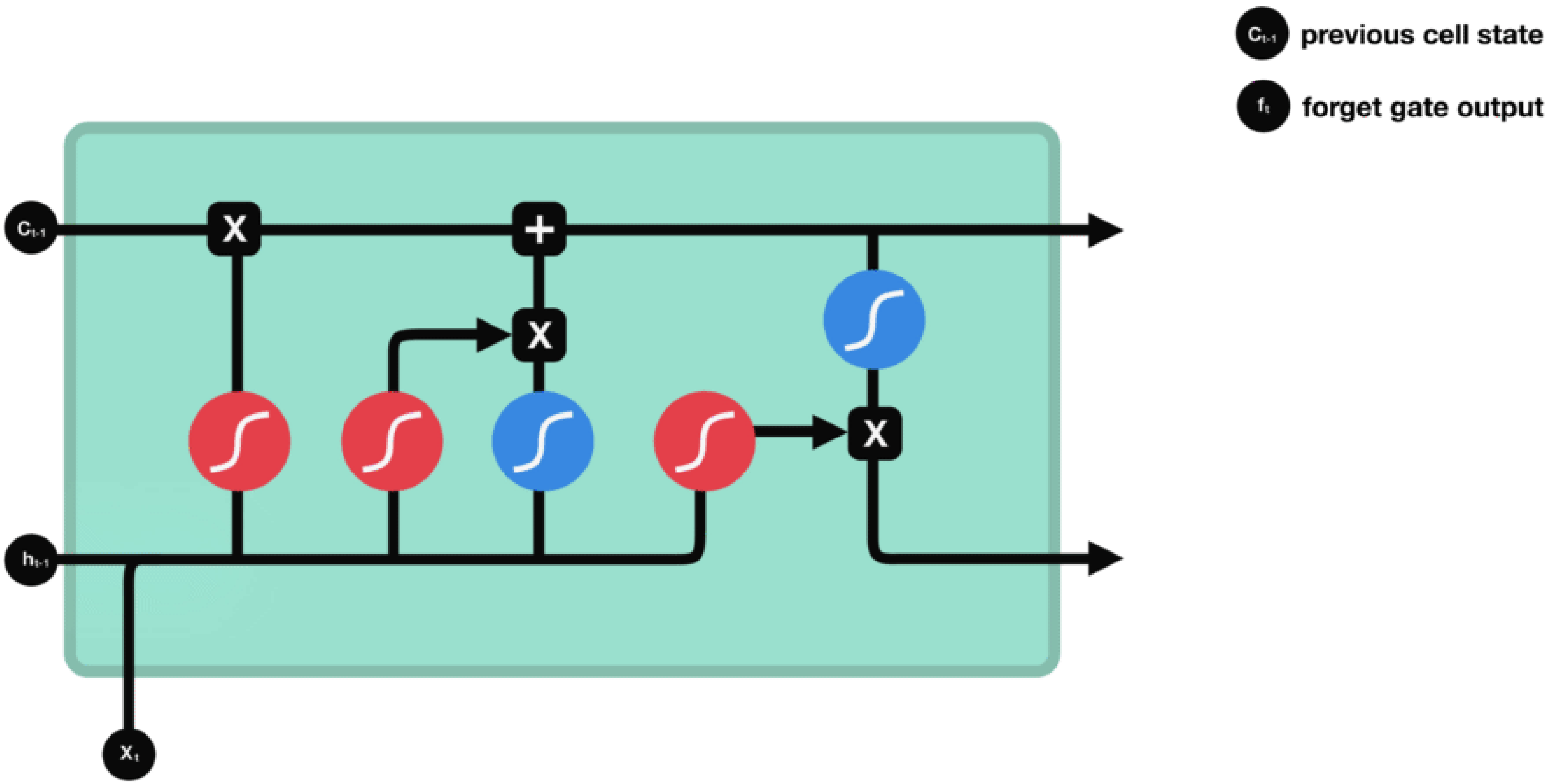
https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21



LSTM Architecture - overview

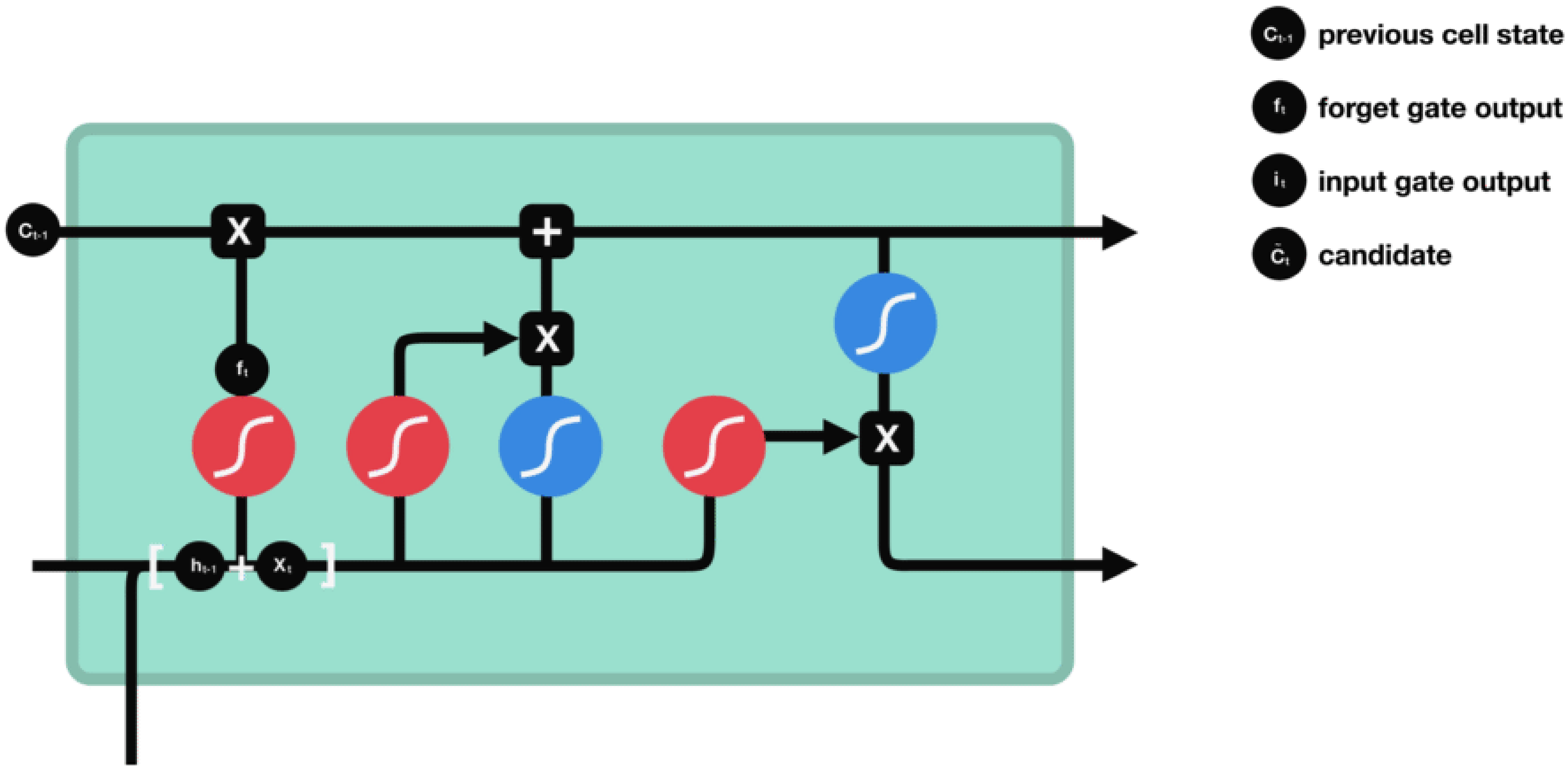


LSTM Architecture - overview



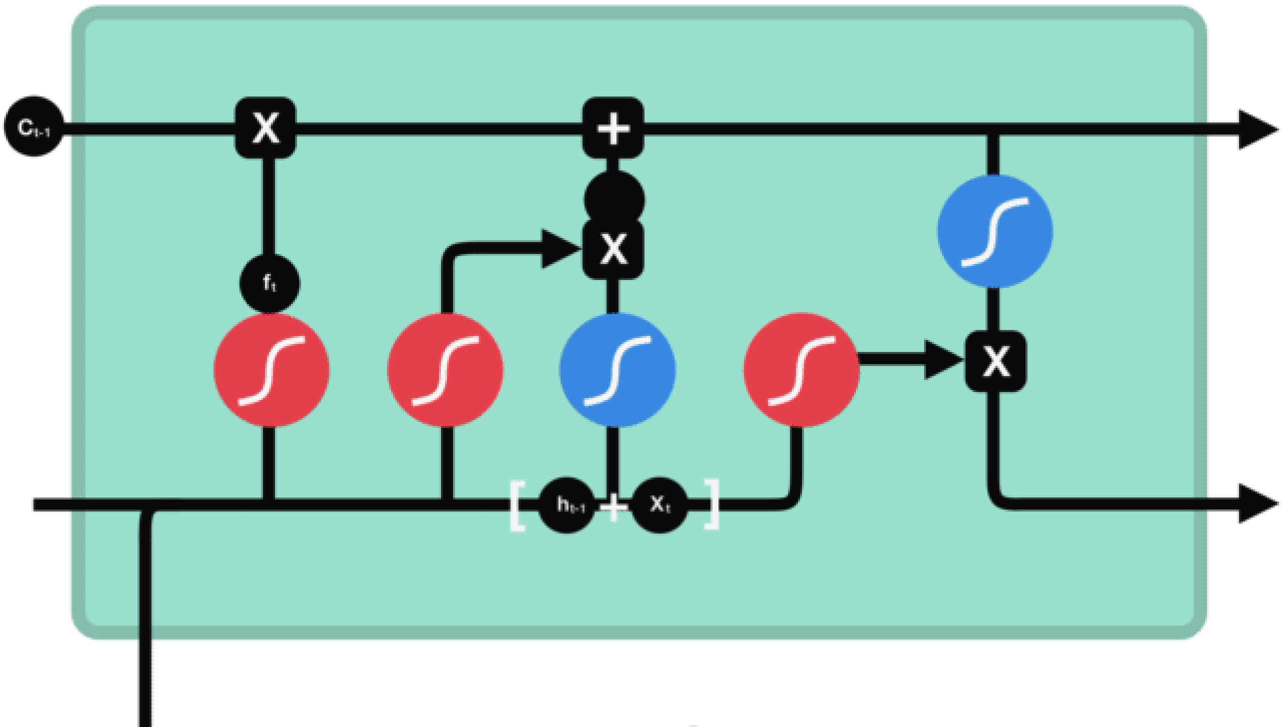
Forget gate operations

LSTM Architecture - overview



Input gate operations

LSTM Architecture - overview

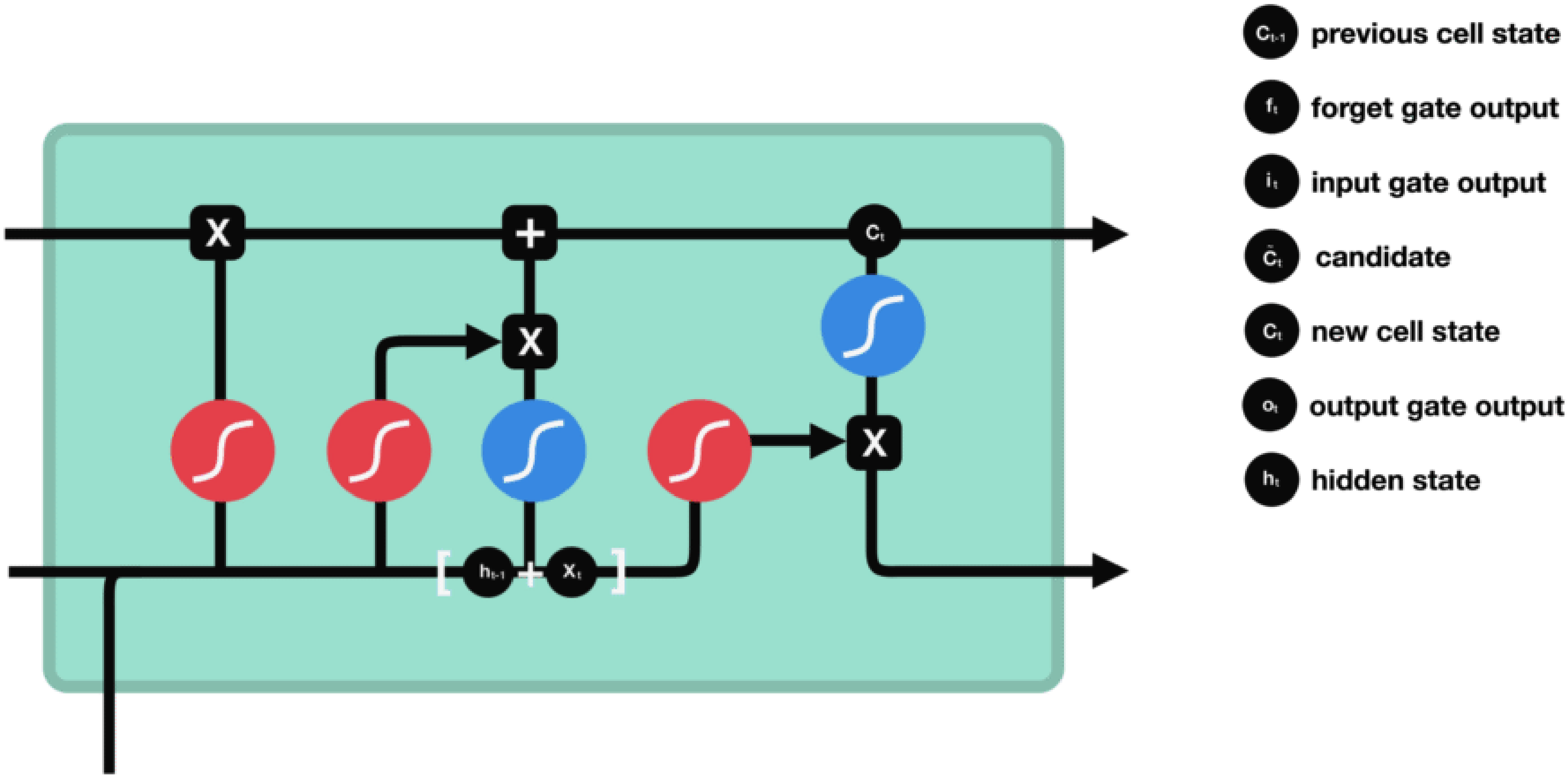


- C_{t-1} previous cell state
- f_t forget gate output
- i_t input gate output
- $\tilde{C}_t$ candidate
- C_t new cell state

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Calculating cell state

LSTM Architecture - overview



output gate operations

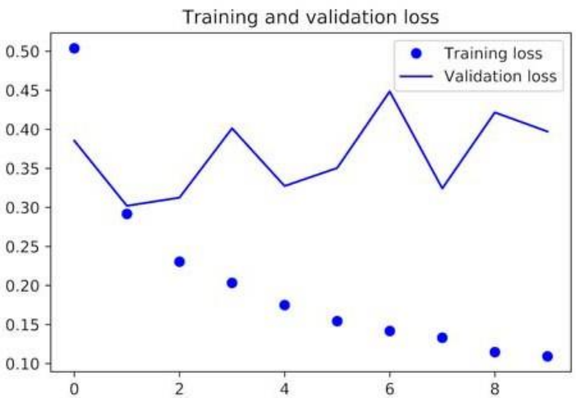
6.2 Understanding recurrent neural networks

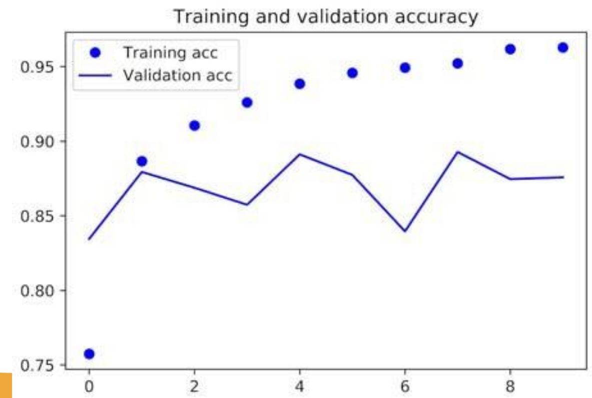
6.2.3 A concrete LSTM example in Keras

- set up a model using an LSTM layer and train it on the IMDB data (see figures 6.16 and 6.17).
- similar to the one with SimpleRNN - specify the output dimensionality of the LSTM layer; leave every other argument (there are many) at the Keras defaults.

Listing 6.27 Using the LSTM layer in Keras

```
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 Understanding recurrent neural networks

6.2.3 A concrete LSTM example in Keras

- achieve up to 89% validation accuracy with less vanishing-gradient problem—and slightly better than the fully connected approach from chapter 3
- less data than you were in chapter 3 by truncating sequences after 500 timesteps, whereas in chapter 3, you were considering full sequences.
- Why isn't LSTM performing better?
  - no effort to tune hyperparameters such as the embeddings dimensionality or the LSTM output dimensionality.
  - lack of regularization
  - analyzing the global, long-term structure of the reviews (what LSTM is good at) isn't helpful for a sentiment-analysis problem.
  - well solved by looking at what words occur in each review, and at what frequency in FCN
  - the strength of LSTM will become apparent: in particular, question-answering and machine translation

6.2.4 Wrapping up

- Now you understand the following:
  - What RNNs are and how they work
  - What LSTM is, and why it works better on long sequences than a naive RNN
  - How to use Keras RNN layers to process sequence data
- Next, advanced features of RNNs

6.3 Advanced use of recurrent neural networks

- three advanced techniques for improving the performance and generalization power of recurrent neural networks
- temperature-forecasting problem - timeseries of data points coming from sensors-temperature, air pressure, and humidity-to predict the temperature 24 hours after the last data point
- We’ll cover the following techniques:
  - Recurrent dropout —to fight overfitting in recurrent layers
  - Stacking recurrent layers—This increases the representational power of the network (at the cost of higher computational loads).
  - Bidirectional recurrent layers—These present the same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues.