Generating Text Using a Character RNN and LSTM

Lecture 11 (eng)

Contents

- Understang of LSTM
- Text Generation Using a character RNN
- Generating Shakespearean Text Using a LSTM, GRU

PANDARUS:

Alas, I think he shall be come approached and the day

When little srain would be attain'd into being never fed,

And who is but a chain and subjects of his death,

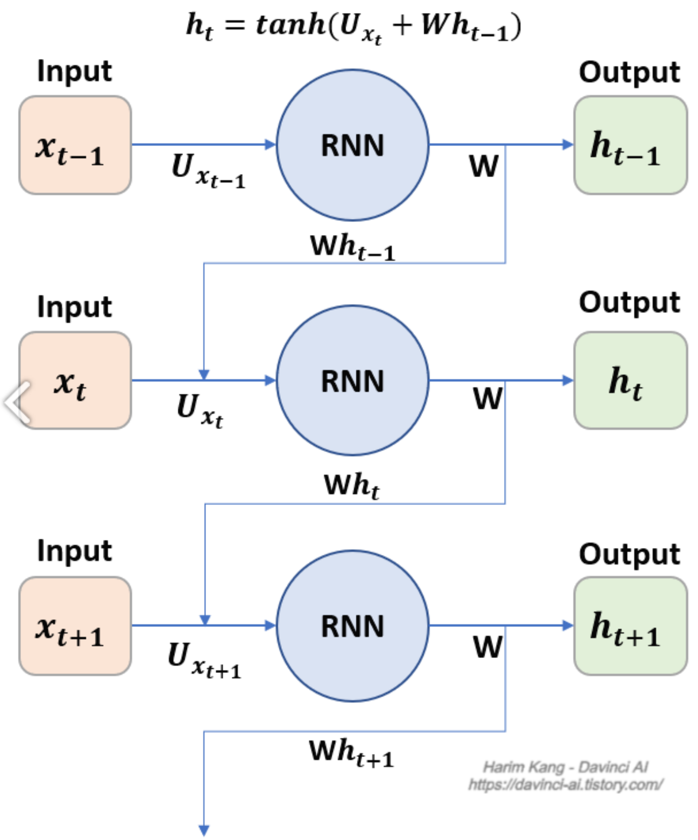
I should not sleep.

RNN-NLP for UST AI

- download ppt and codes
- https://github.com/hongsukyi/rnn-nlp-lectures

Review of Recurrent Neural Networks

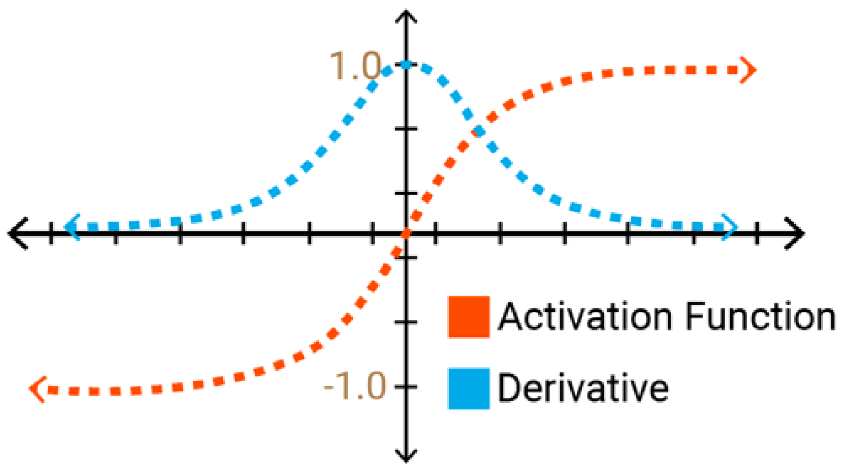
➢ In standard RNNs, this repeating module will have a single tanh layer.



$$h_t = tanh(U_{x_t} + Wh_{t-1})$$

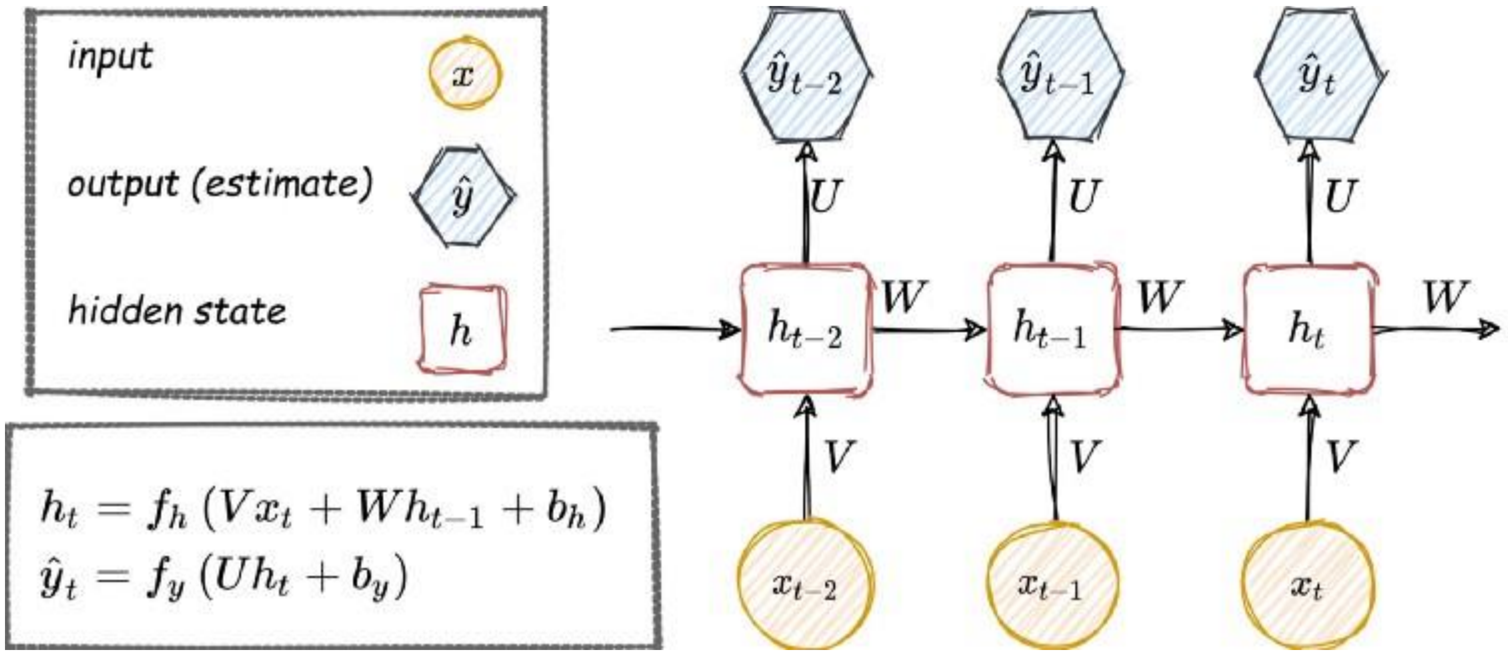
$$f(x) = \frac{2}{1+e^{-2x}} - 1 = \tanh(x)$$

$$f'(x) = 1 - \tanh^2(x)$$



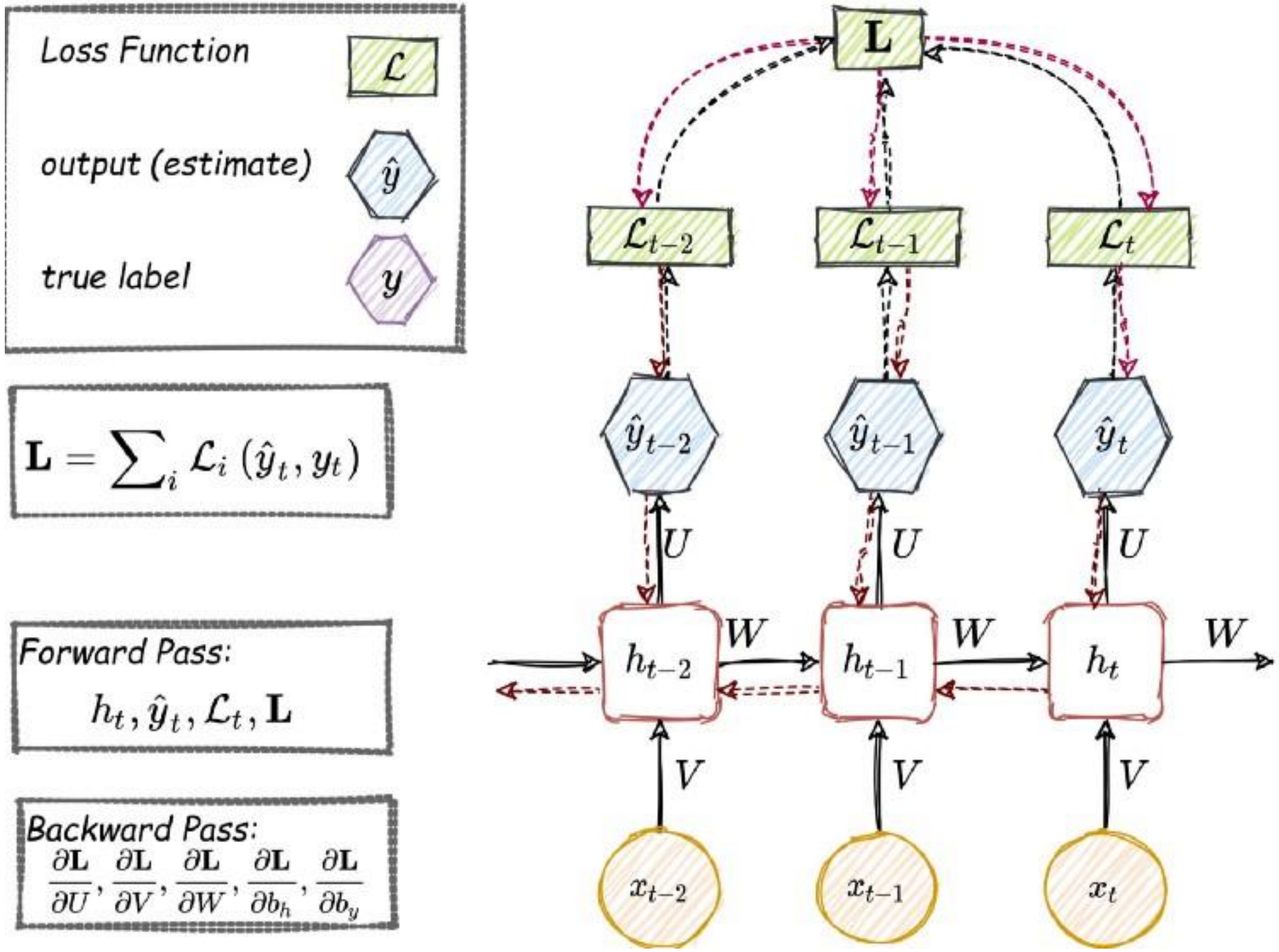
Review of Recurrent Neural Networks

- A simple RNN architecture is shown below, where V, W, and U are the weights matrices, and b is the bias vector.



$$h_t = f_h\left(Vx_t + Wh_{t-1} + b_h\right)$$
$$\hat{y}_t = f_y\left(Uh_t + b_y\right)$$

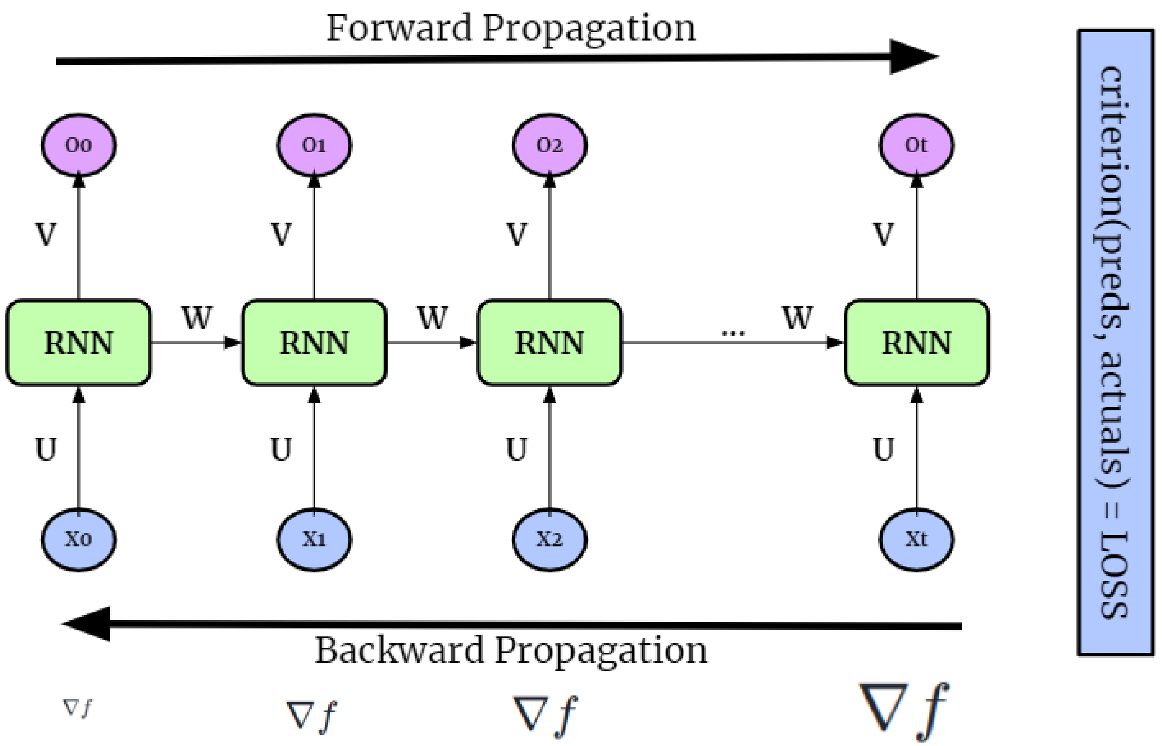
Backpropagation Through Time (BPTT)



$$\frac{\partial \mathbf{L}}{\partial W} \propto \sum_{i=0}^{T} \left(\prod_{i=k+1}^{y} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W}$$

Vanishing Gradient Problem

- it is due to the nature of backpropagation (during the optimization process)
  - ✓ if the adjustment in the previous layer is small, then that in the current layer will be smaller



RNNs suffer from the problem of VGP and EGP

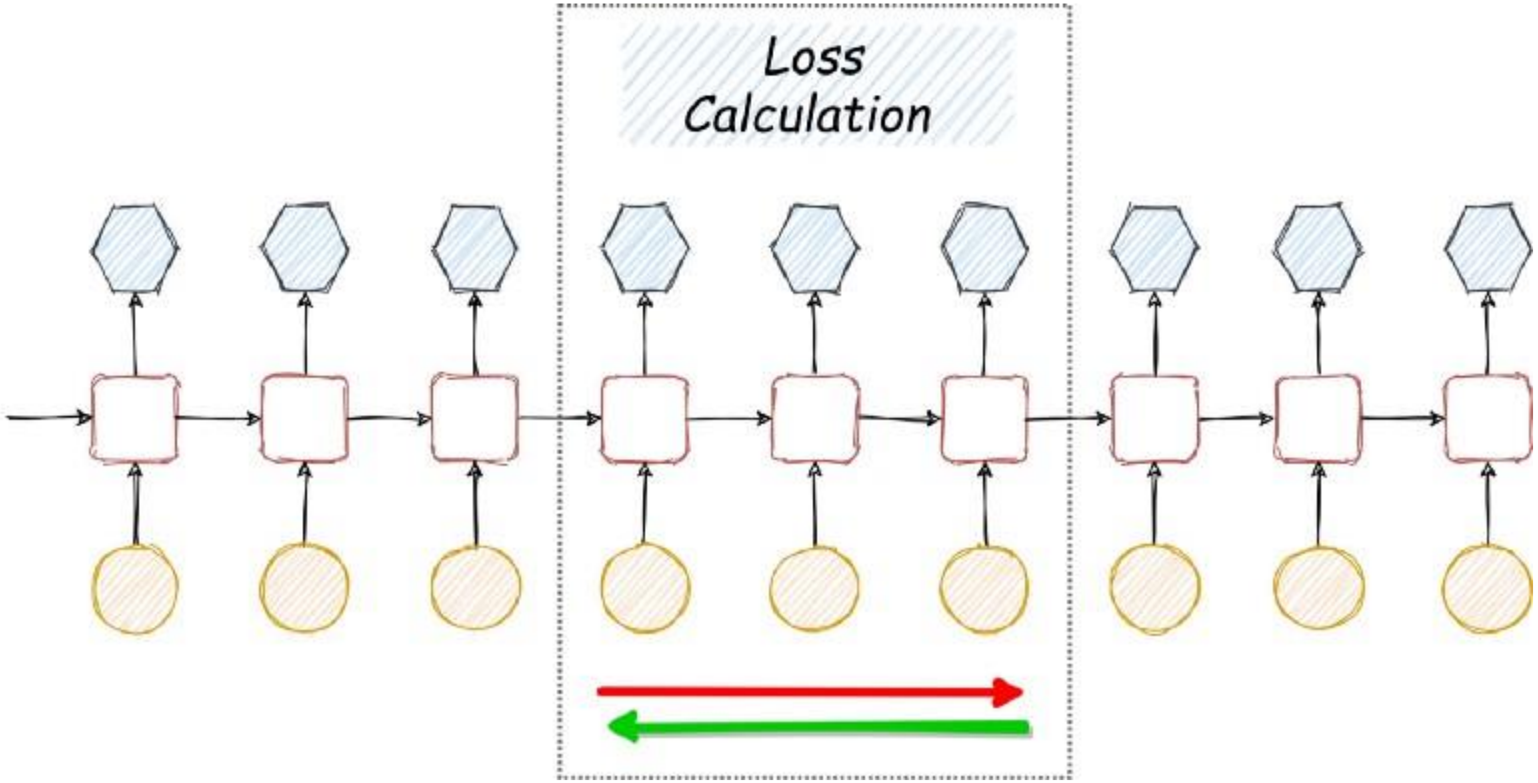
- VGP(Vanishing Gradient Problem)
  - ✓ when the gradient becomes too small, the parameter updates become insignificant. This makes the learning of long data sequences difficult.
- EGP(Exploding Gradient Problem)
  - ✓ If the slope tends to grow exponentially instead of decaying, when large error gradients accumulate during the training process.

1. *Vanishing gradient* $\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 < 1$

2. *Exploding gradient* $\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 > 1$

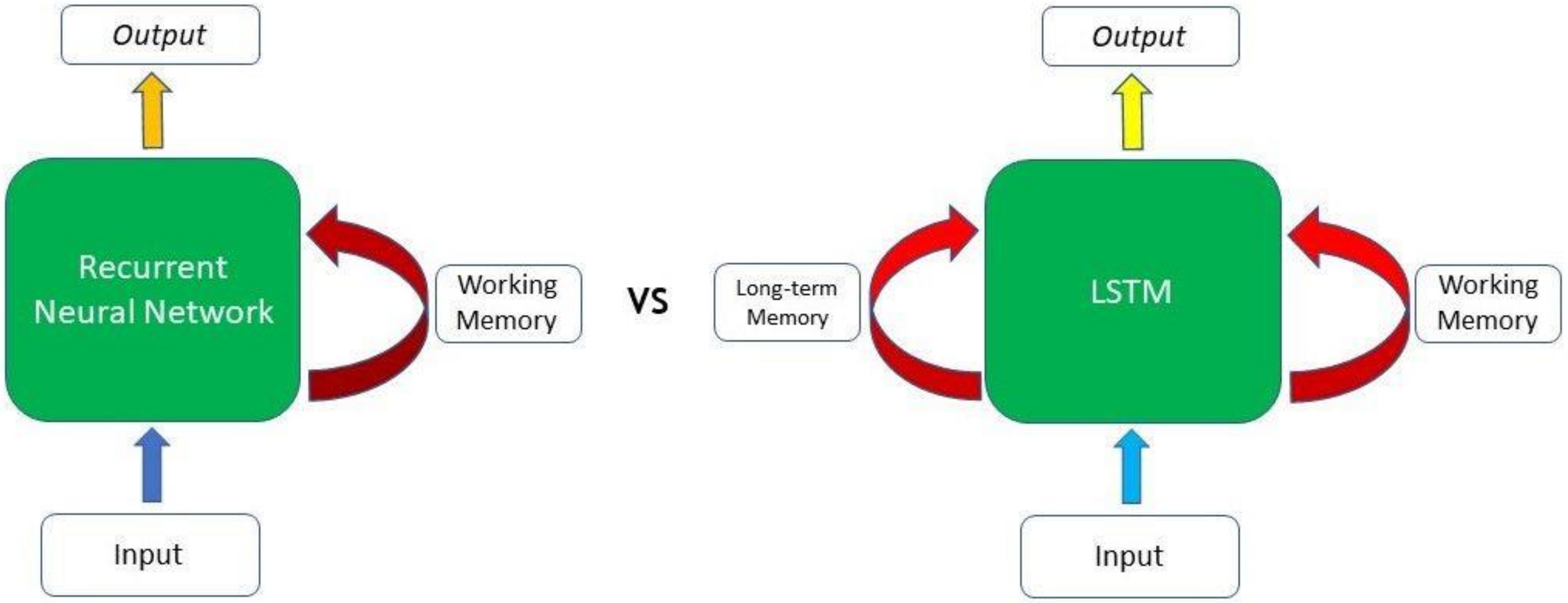
Truncated Backpropagation Through Time (Truncated BPTT)

- ➢ Truncated BPTT trick tries to overcome the VGP
  - ✓ by considering a moving window through the training process



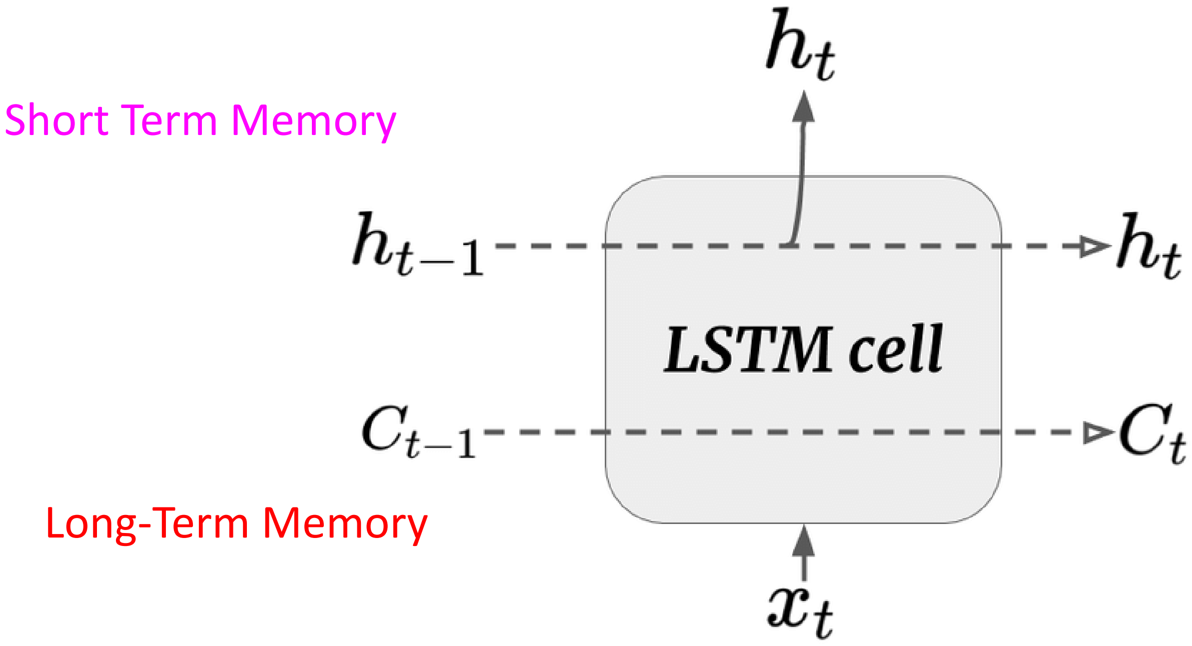
Long Short-Term Memory networks (LSTMs)

- Allows learning of long-term dependencies
  - ✓ A RNN addresses the vanishing/exploding gradient problem
  - ✓ Capable of learning long-term dependencies by remembering information



Long Short Term Memory Network

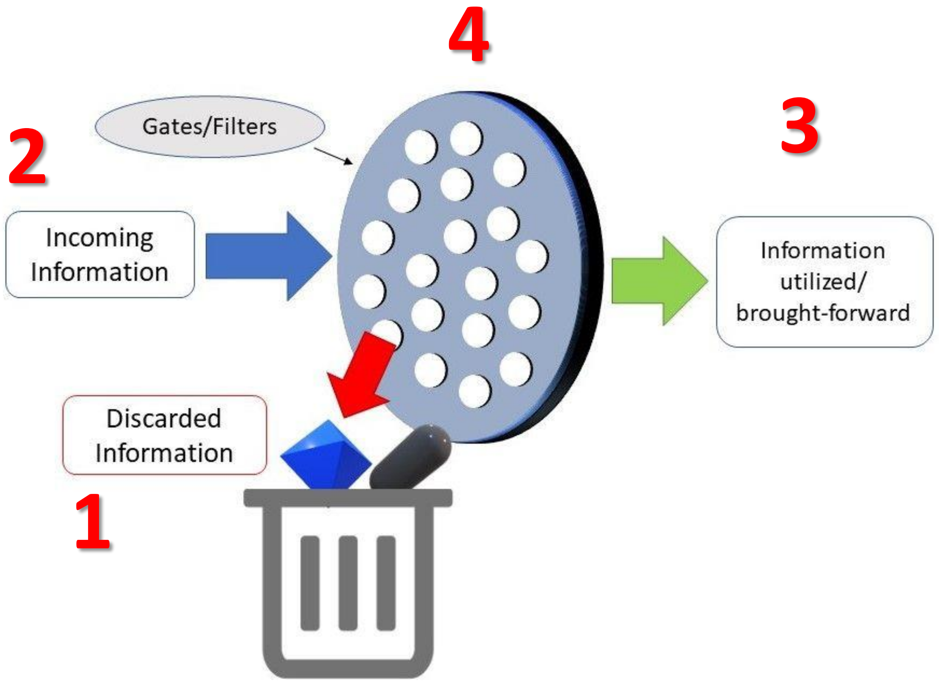
- LSTM (Long Short Term Memory Network)
  - ✓ LSTMs are explicitly designed to avoid long-term dependency problems
  - ✓ The observed state x_t is combined with previous memory and hidden states to output a hidden state h_t.

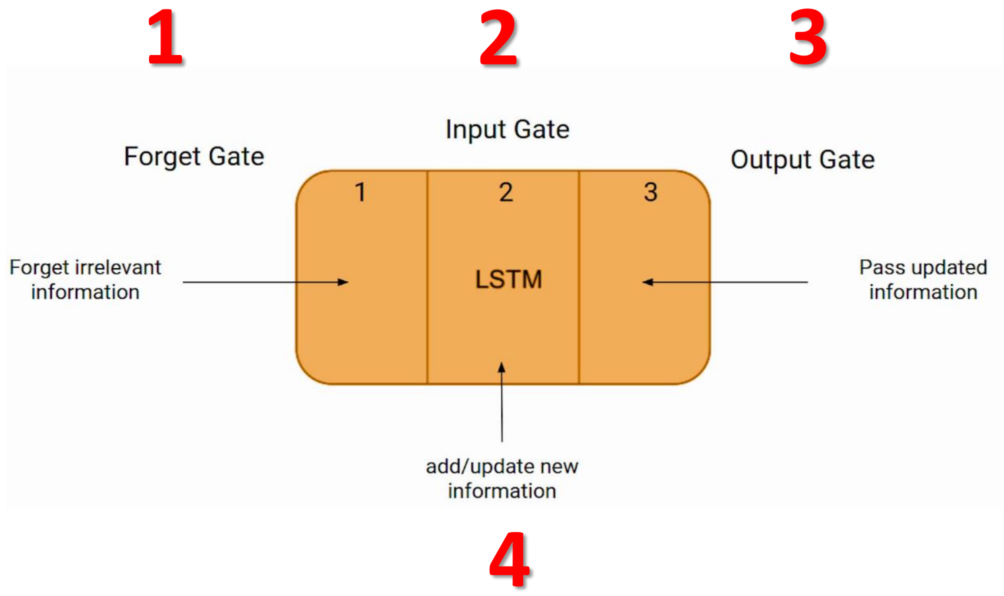


Gates

➢ Gates control the flow of information to/from the memory

✓ Gates are controlled by a concatenation of the output from the previous time step and the current input and optionally the cell state vector.

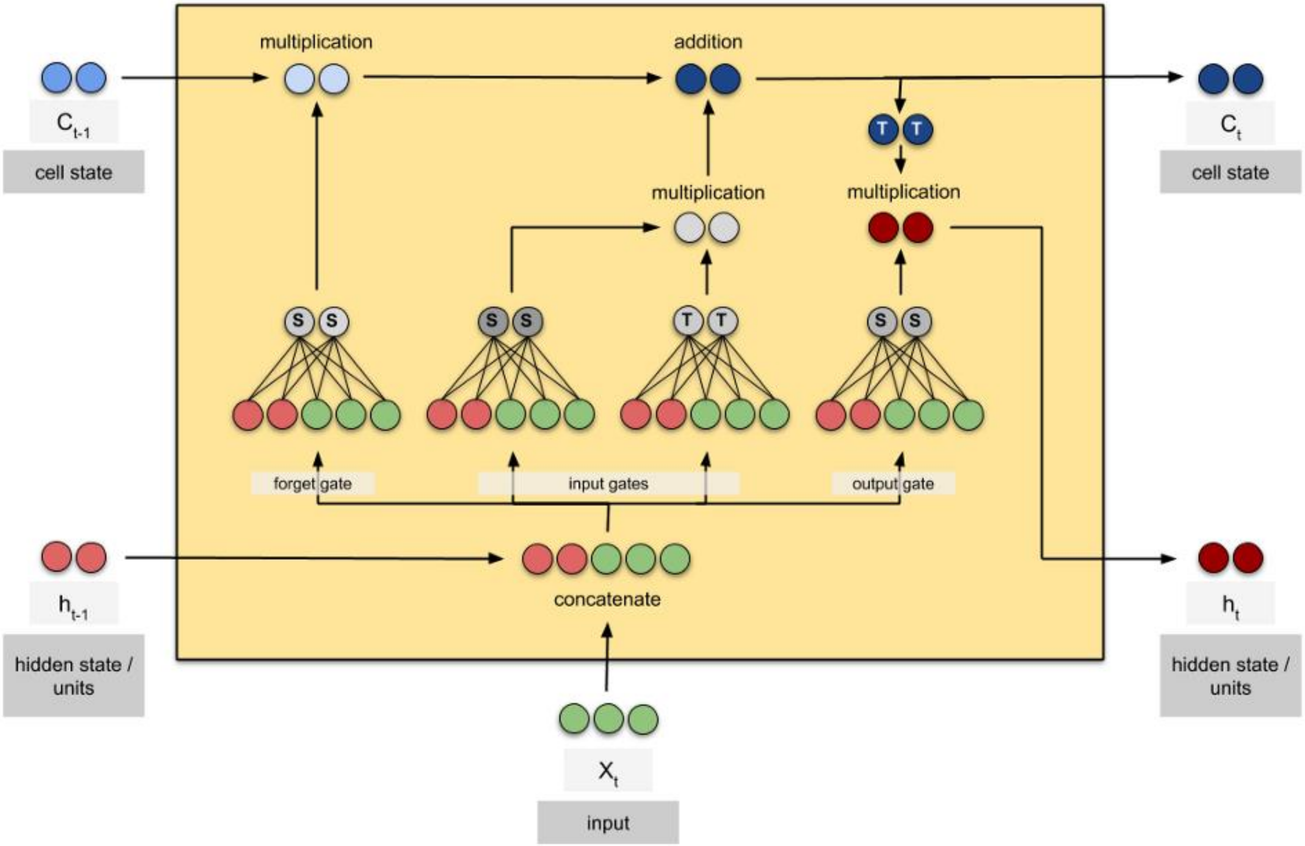




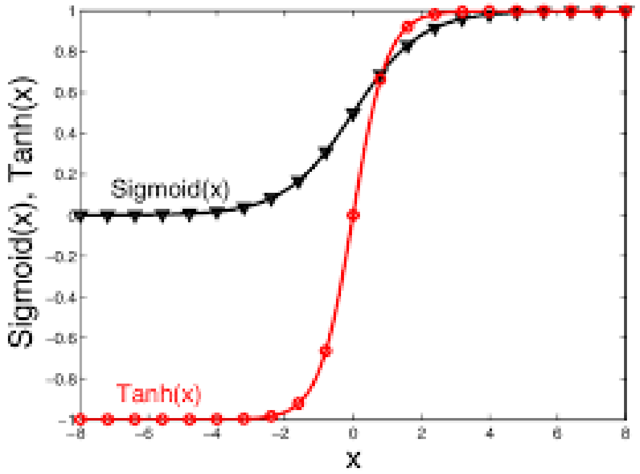
Understanding the roles played by gates in LSTM architecture

- Forget Gate
  - ✓ whether we should keep the information from the previous timestamp or forget it
- Input Gate
  - ✓ Decide how much this unit adds to the current state
- New information: Memory Upgate
  - ✓ The cell state vector aggregates the two components (old memory via the forget gate and new memory via the input gate)
- Output Gate
  - ✓ Decide what part of the current cell state makes it to the output

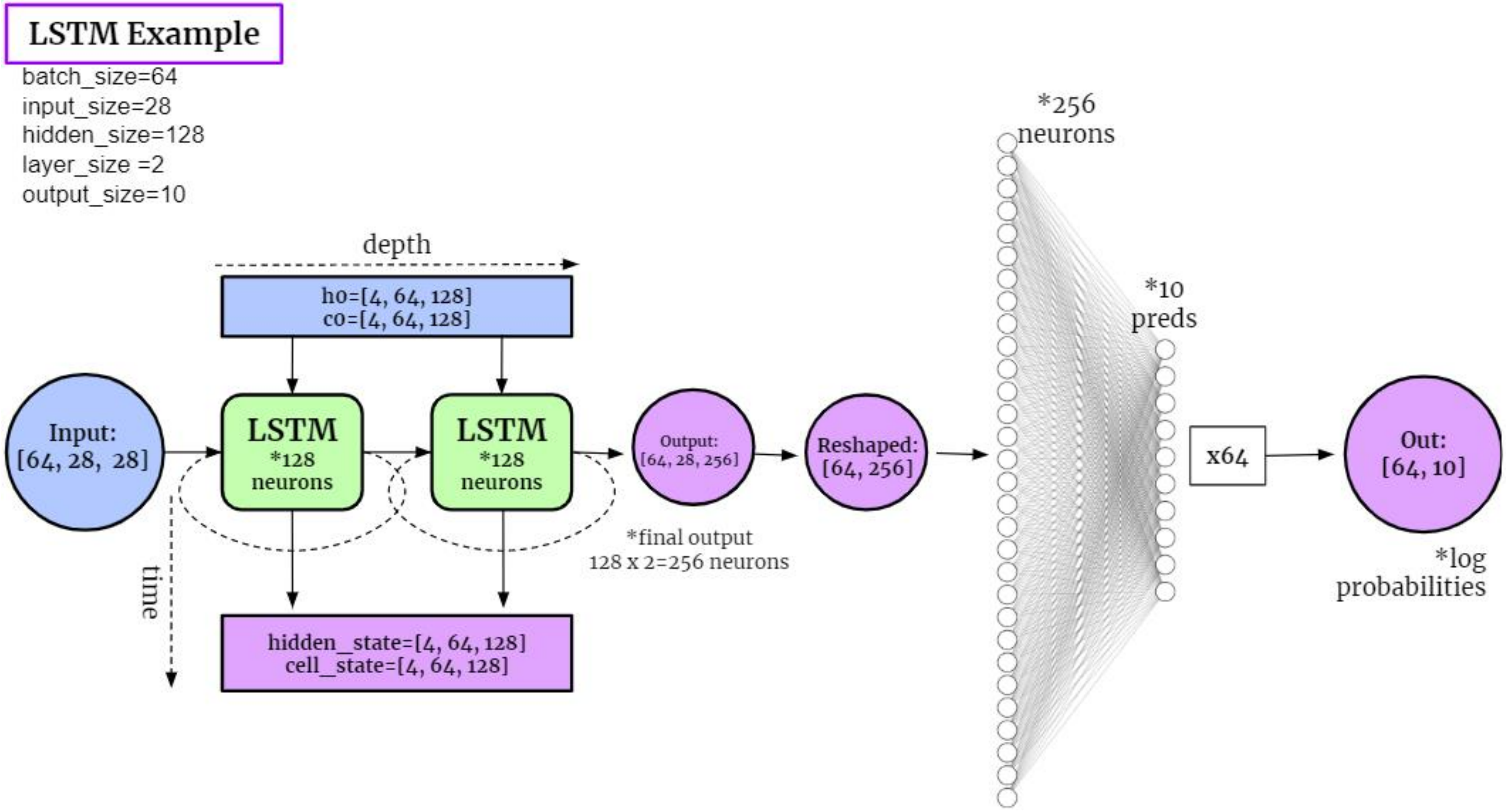
LSTM Process



Activation functions



LSTM Example



Lab 11-1
Text Tockenizer

Text Generation

Understanding the principles

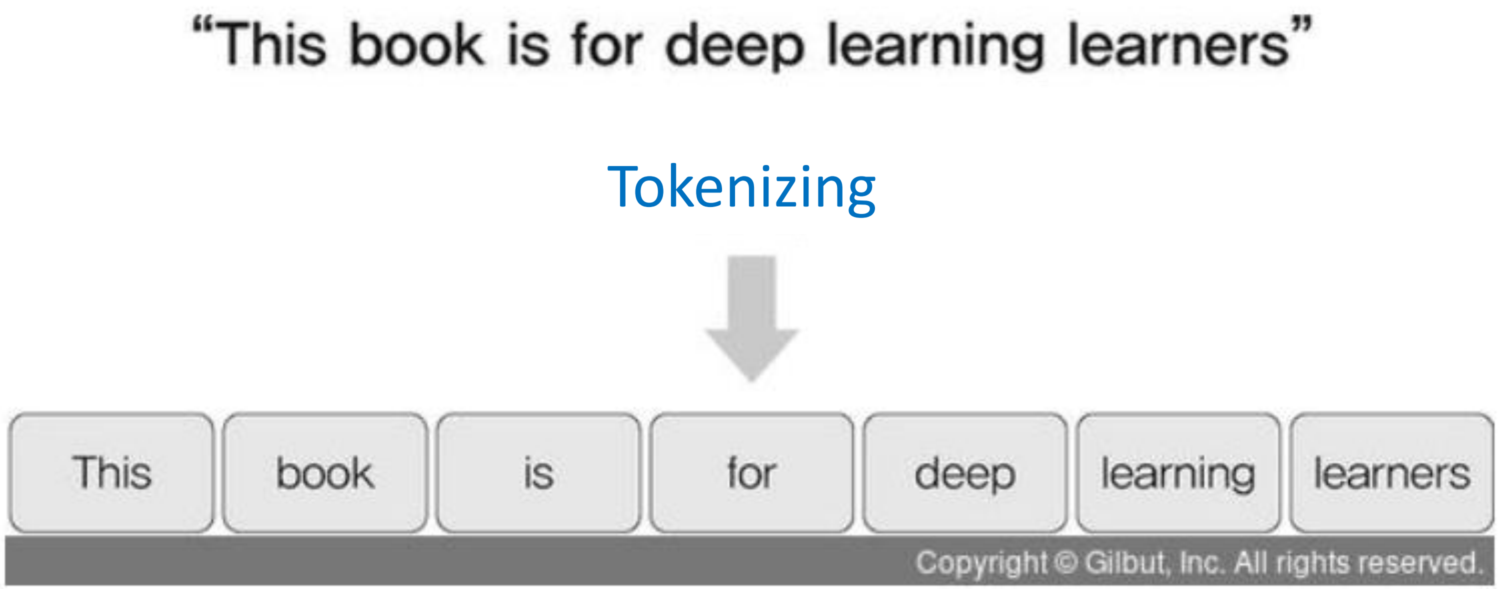
- Token : Language elements that we can't share anymore
- Tokenizer
  - ✓ work to input text data into the neural network.
  - ✓ The preprocessing process that converts it into an appropriate form through encoding
- One-hot encoding
  - ✓ In the case of text data, an embedding layer is basically used.

Text
"The cat sat on the mat."

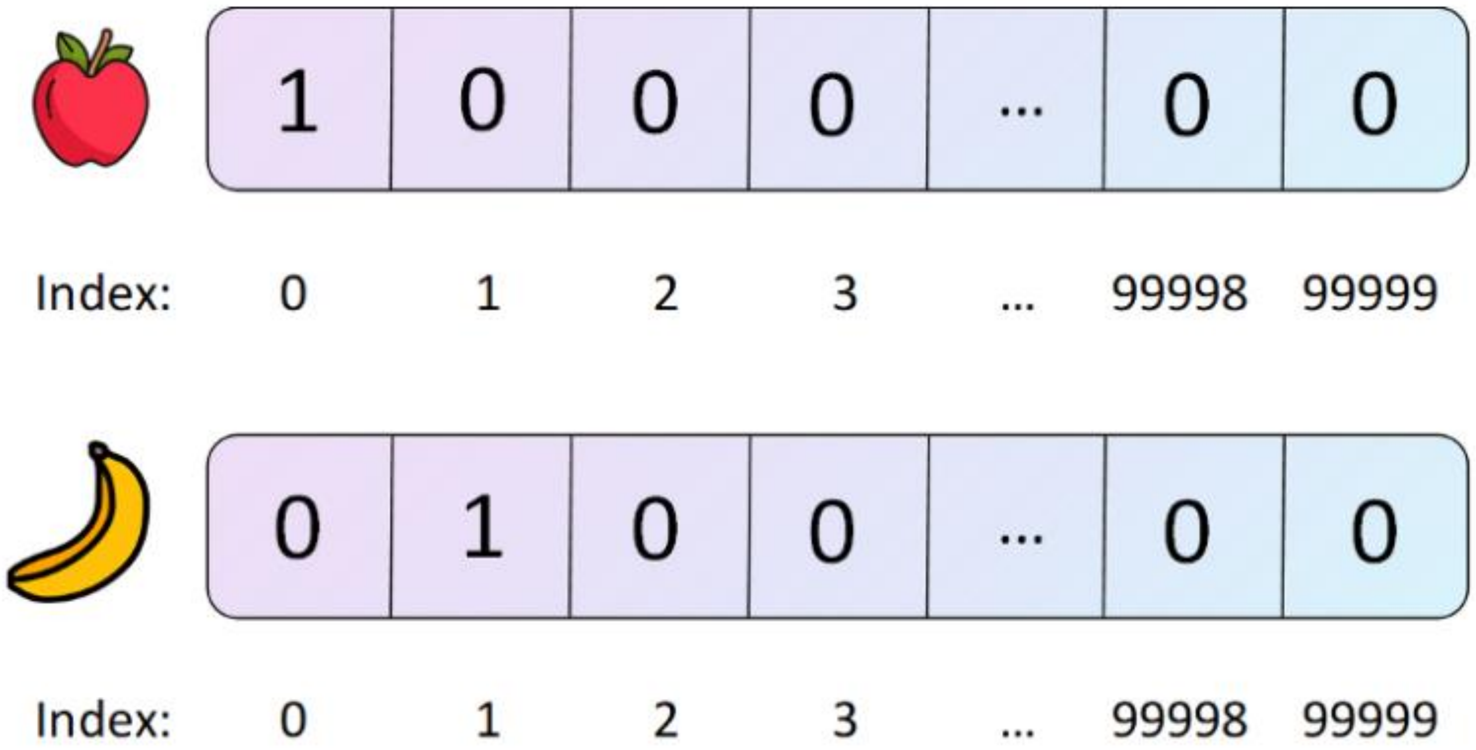
Tokens
"the", "cat", "sat", "on", "the", "mat", "."

Tokenizing words

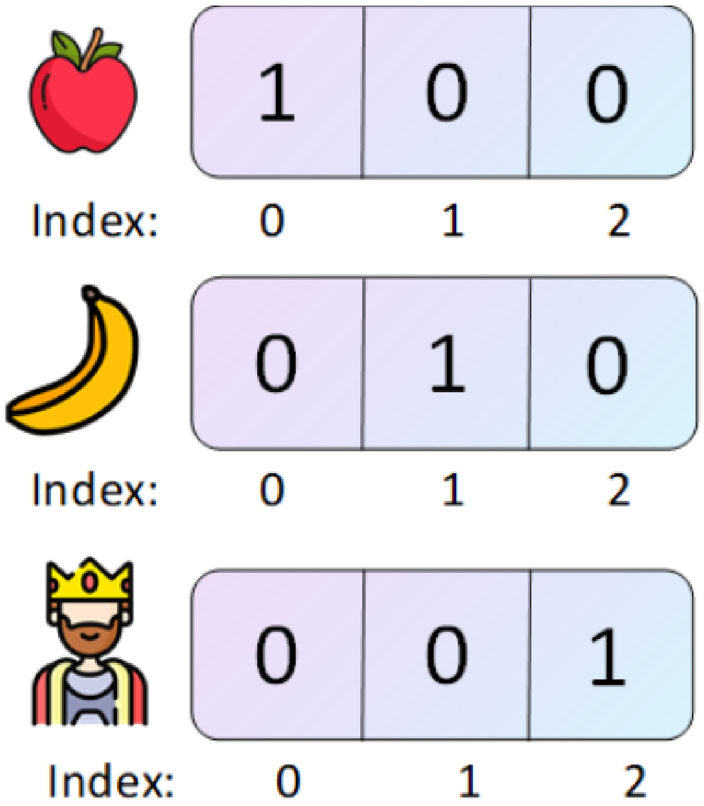
➢ Word tokenization divides sentences based on spacing as follows.

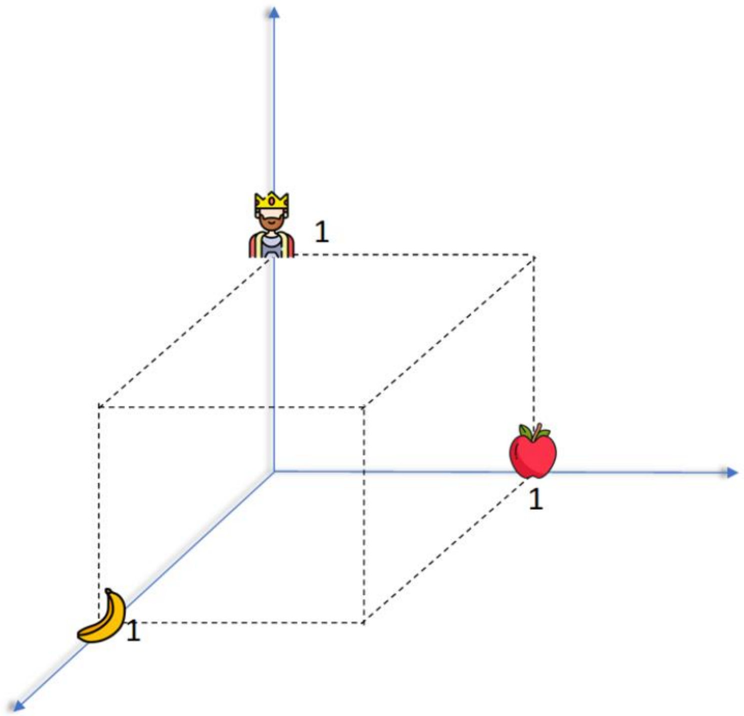


One-Hot Encoding



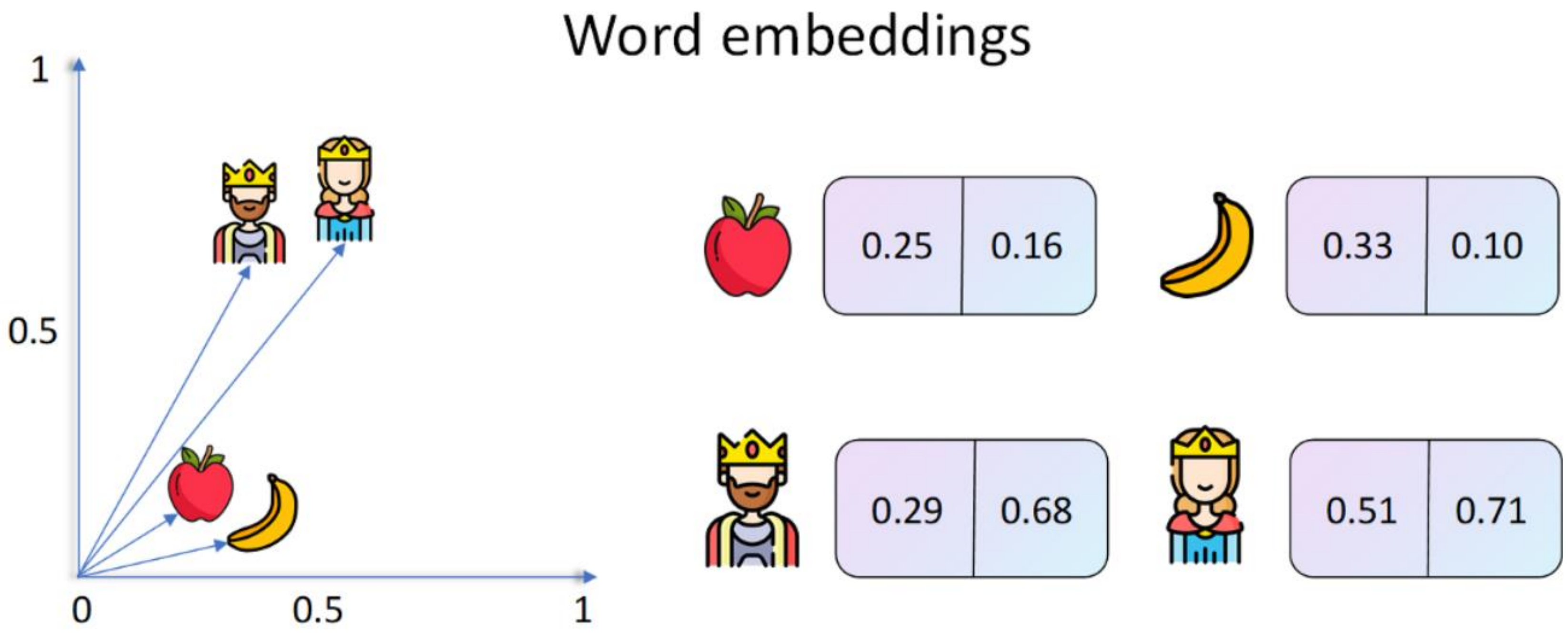
Word Embedding





Word Embedding

- 2 dimensional word embedding representation of our example words



1. Word-based encoding

- ➢ The example below shows how to encode two sentences
  - ✓ 'You are the Best', and 'You are the Nice' based on words using TensorFlow.

1. Word-based encoding

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical
```

```
sentences = [
  'You are the Best',
  'You are the Nice'
]
```

1. Word-based encoding

➢ fit_on_texts() 메서드는 문자 데이터를 입력받아서 리스트의 형태로 변환합니다.

```
tokenizer = Tokenizer(num_words = 100, oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index
```

Words that have not been indexed in advance are indexed as "OOV"

```
print(word_index)
print('---------------------------------------')
total_words = len(tokenizer.word_index) + 1
print('total_words=',total_words)
```

```
{'<OOV>': 1, 'you': 2, 'are': 3, 'the': 4, 'best': 5, 'nice': 6}
---------------------------------------
total_words= 7
```

The word_index attribute of the tokenizer returns a dictionary containing a pair of key-values of words and numbers.

The output results show that the upper case 'I' has been converted to the lower case 'i'.

2) Converting text into a sequence

텍스트를 시퀀스로 변환하기

```
sequences = tokenizer.texts_to_sequences(sentences)

print(word_index)
print(sequences)
```

```
{'<OOV>': 1, 'you': 2, 'are': 3, 'the': 4, 'best': 5, 'nice': 6}
[[2, 3, 4, 5], [2, 3, 4, 6]]
```

3. Setting up padding

➢ You have to padding to make the sentence the same length.

✓ Padding uses the pad_sequences function.

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

Sequences are text sentences converted into sequences of integers

- Since the longest sequence is 7, it has all been converted into sequences of the same length

```
padded = pad_sequences(sequences)

print(word_index)
print(sequences)
print(padded)
```

```
{'<OOV>': 1, 'you': 2, 'are': 3, 'the': 4, 'best': 5, 'nice': 6}
[[2, 3, 4, 5], [2, 3, 4, 6]]
[[2 3 4 5]
 [2 3 4 6]]
```

3. Setting up padding

- padding parameter : 'pre', 'post’
  - ✓ If the padding parameter is specified as 'post', padding is filled after the sequence. The default is "pre"

```
padded = pad_sequences(sequences, padding='post')
print(padded)
```

```
[[2 3 4 5]
 [2 3 4 6]]
```

4) Encoding in binary form

```
# 이진 형태로 인코딩합니다.
binary_results = tokenizer.sequences_to_matrix(sequences, mode = 'binary')
print(f'binary_vectors:₩n {binary_results}₩n')
```

```
binary_vectors:
 [[0. 0. 1. 1. 1. 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. 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. 1. 1. 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. 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.]]
```

4) Encoding in binary form

```
print(f'One-Hot Encodeing:',to_categorical(sequences))
```

```
One-Hot Encodeing: [[[0. 0. 1. 0. 0. 0. 0.]
  [0. 0. 0. 1. 0. 0. 0.]
  [0. 0. 0. 0. 1. 0. 0.]
  [0. 0. 0. 0. 0. 1. 0.]]

 [[0. 0. 1. 0. 0. 0. 0.]
  [0. 0. 0. 1. 0. 0. 0.]
  [0. 0. 0. 0. 1. 0. 0.]
  [0. 0. 0. 0. 0. 0. 1.]]]
```

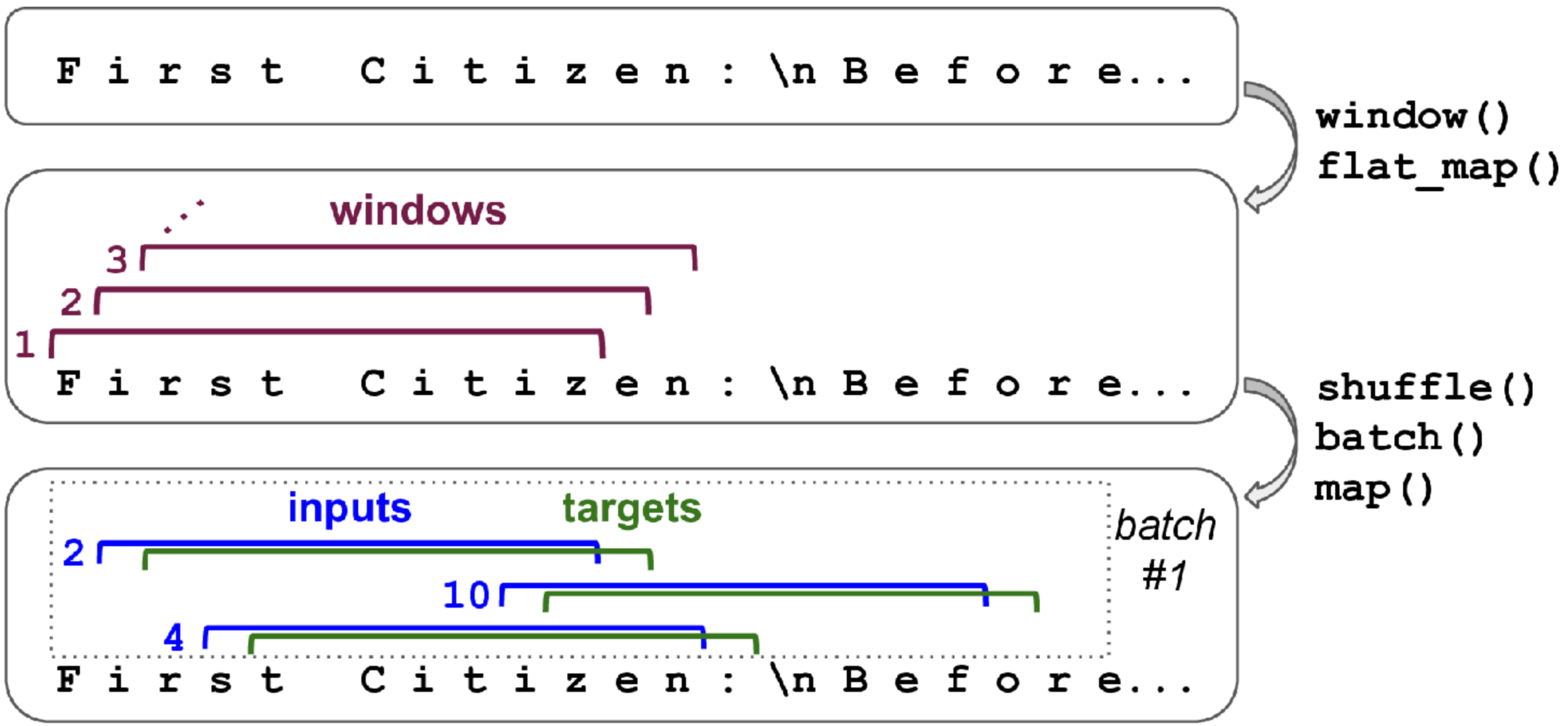
```
test_text = ['You are the One']
test_seq = tokenizer.texts_to_sequences(test_text)

print(f'test sequences: {test_seq}')
```

```
test sequences: [[2, 3, 4, 1]]
```

Lab 11-2
Generating Text Using a Character RNN

Text Generation

Preparing a dataset of shuffled windows

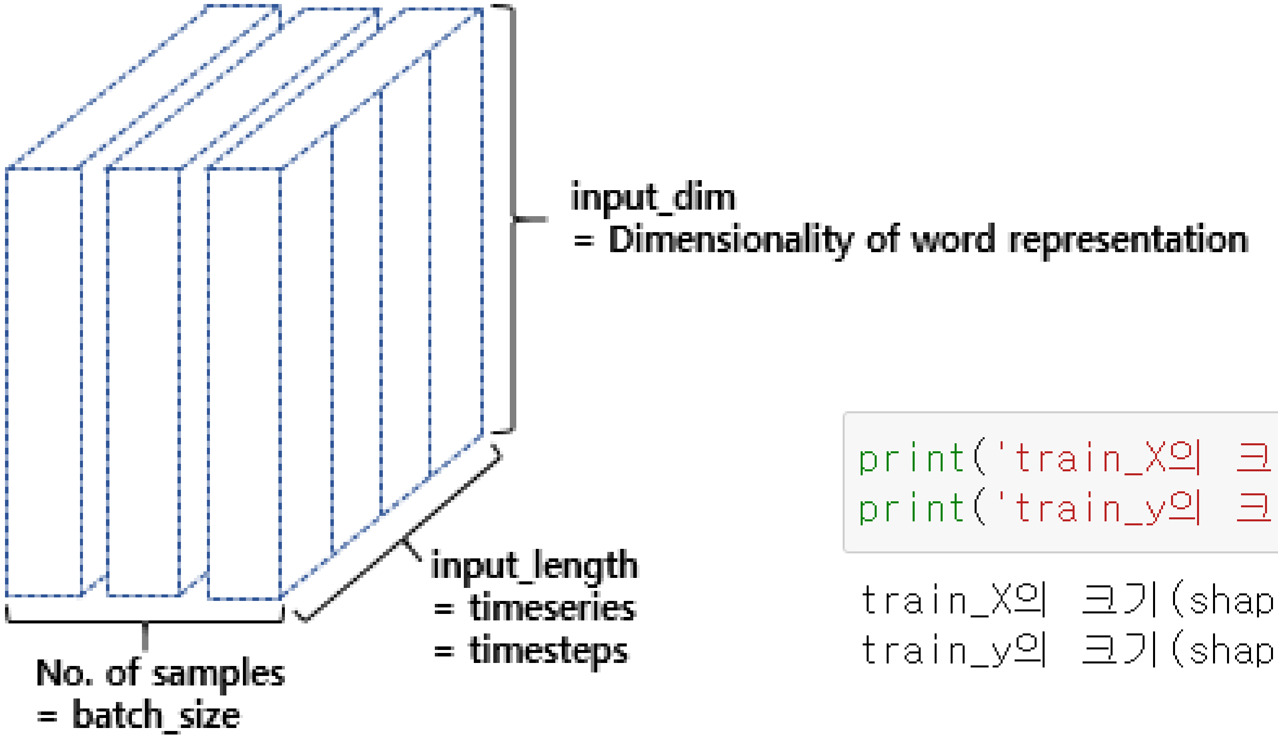


Text Generation through LSTM model

Lab11-2

1) 데이터에 대한 이해와 전처리

원-핫 벡터의 차원은 글자 집합의 크기인 56이어야 하므로 원-핫 인코딩이 수행되었다.

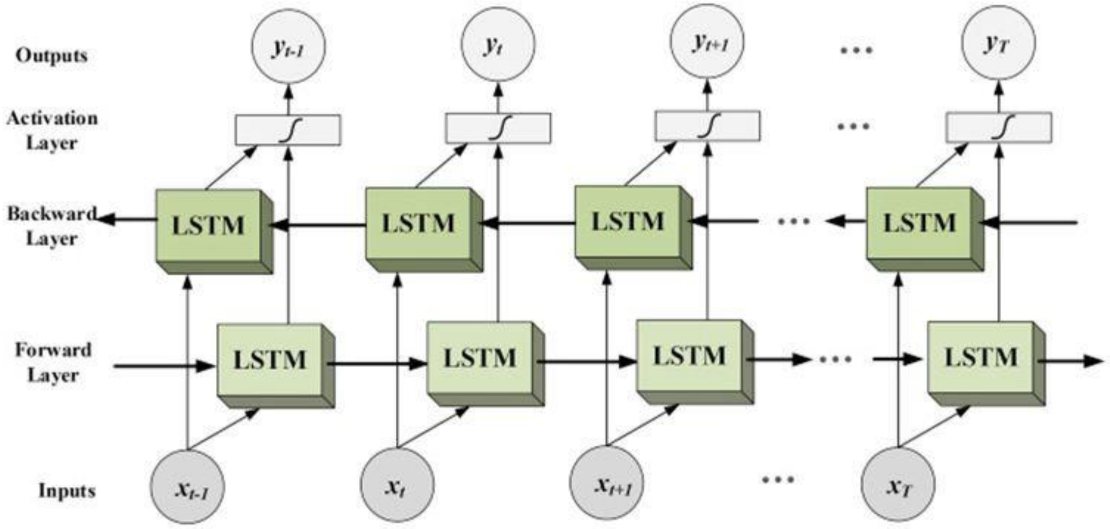



```
print('train_X의 크기(shape) : {}'.format(train_X.shape))
print('train_y의 크기(shape) : {}'.format(train_y.shape))
```

```
train_X의 크기(shape) : (2658, 60, 56)
train_y의 크기(shape) : (2658, 60, 56)
```

1) Bidirectional LSTMs

- Bidirectional LSTMs are an extension of traditional LSTMs that can improve model performance on sequence classification problems.
  - ✓ They train the model forward and backward on the same input (so for 1 layer LSTM we get 2 hidden and cell states)
  - ✓ First from left to right on the input sequence and the second in reversed order of the input sequence.



1) Bidirectional LSTMs

➢ implement this model in text generation

```
import numpy as np
import tensorflow as tf

from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Embedding, LSTM, Dense, GRU, Bidirectional
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import regularizers
from tensorflow.keras.utils import to_categorical, plot_model
```

2) Text pre-processing

- the whole text is cleaned and converted to lower case and the whole corpus of sentences are joined

```
tokenizer = Tokenizer()
data = open('./tiny-shakespeare.txt').read()
```

```
len(data)
```

```
3]: 1115394
```

```
data=data[:10000]
```

```
corpus = data.lower().split("₩n")
tokenizer.fit_on_texts(corpus)
print(tokenizer.word_index)
```


{'the': 1, 'you': 2, 'and': 3, 'to': 4, 'citizen': 5, 'first': 6, 'that': 7, '
enenius': 14, 'your': 15, 'a': 16, 'with': 17, 'in': 18, 'all': 19, 'are'
ut': 26, 'as': 27, 'this': 28, 'us': 29, 'have': 30, 'our': 31, 'me': 32
37, 'marcius': 38, 'him': 39, 'be': 40, 'one': 41, 'them': 42, 'his':
'speak': 49, 'at': 50, 'if': 51, 'did': 52, 'was': 53, 'must': 54, 'do':
e': 60, 'know': 61, 'more': 62, 'these': 63, 'other': 64, 'where': 6
70, 'let': 71, 'done': 72, 'poor': 73, 'on': 74, 'an': 75, 'gods': 76,
81, 'yourselves': 82, 'tell': 83, 'yet': 84, 'body': 85, 'then': 86, 'r
an': 91, 'corn': 92, 'own': 93, 'patricians': 94, 'were': 95, 'think':
'hath': 101, 'though': 102, 'can': 103, 'cannot': 104, 'way': 105,
110, 'care': 111, 'may': 112, 'by': 113, 'up': 114, 'or': 115, 'men
'should': 120, "'": 121, 'upon': 122, 'great': 123, 'toe': 124, 'nol
ius': 129, 'people': 130, "know't": 131, "we'll": 132, 'away': 133
137, 'revenge': 138, 'ere': 139, 'hunger': 140, "he's": 141, 'very
o': 146, 'soft': 147, 'men': 148, 'please': 149, 'virtue': 150, 'help
s': 155, 'worthy': 156, 'honest': 157, 'enough': 158, 'rest': 159,
ms': 164, 'dearth': 165, 'state': 166, 'whose': 167, 'ever': 168, 'v
ain': 173, 'daily': 174, 'rich': 175, 'eat': 176, "there's": 177, 'acc
'deliver': 183, 'only': 184, "i'": 185, 'see': 186, 'appetite': 187, 'c

2) Text pre-processing

```
print(corpus[:15])
```

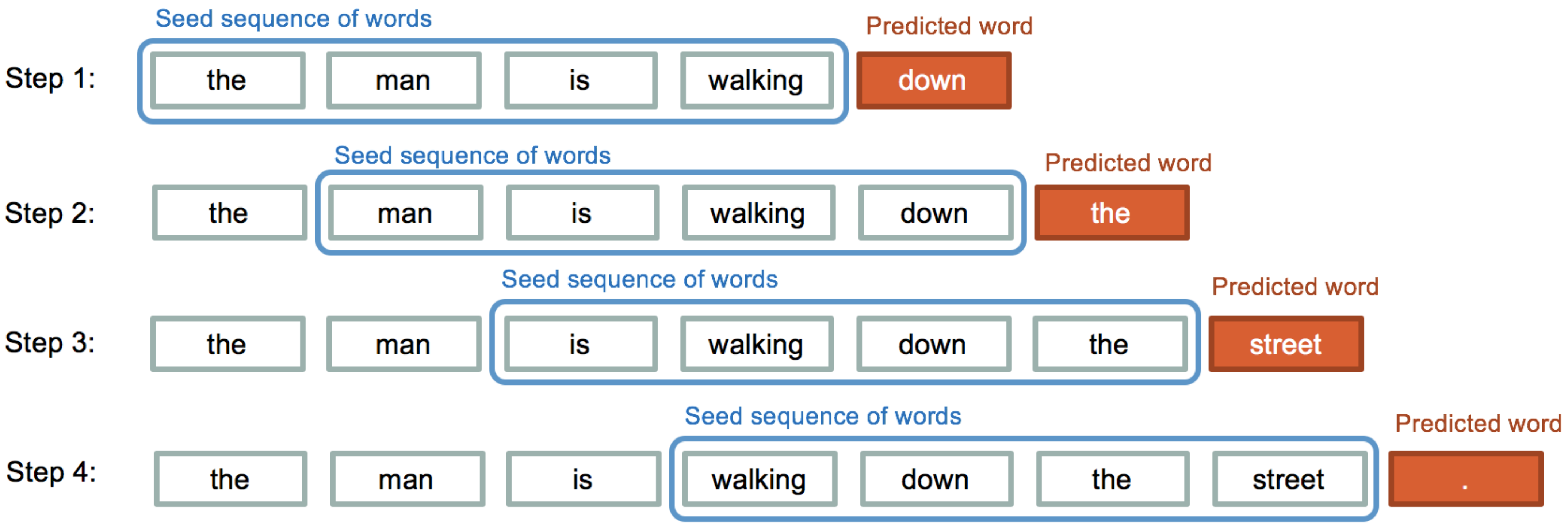
```
['first citizen:', 'before we proceed any further, hear me speak.', '', 'all:', 'speak, speak.', '', 'first citizen:', 'you are a
ll resolved rather to die than to famish?', '', 'all:', 'resolved. resolved.', '', 'first citizen:', 'first, you know caius marci
us is chief enemy to the people.', '']
```

```
total_words = len(tokenizer.word_index) + 1
print('total_words=',total_words)
```

```
total_words= 672
```

3) Creating Sequences

- For each word, an n-gram sequence is made and input sequences are updated.
  - ✓ It happens in the iteration for the next word and so on



3) Creating Sequences

- For example
  - ✓ in the sentence below first 'He' was extracted out then, 'He was ' was extracted, and then 'He was walking ' was extracted, and so on.

```python
# create input sequences using list of tokens
input_sequences = []
for line in corpus:
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)
```

Padding sequences

➤ The maximum length of the sentence is extracted and then the rest of the sentences are pre-padded as per the longest sentence.

```
# pad sequences
max_sequence_len = max([len(x) for x in input_sequences])
input_sequences = np.array(pad_sequences(input_sequences,
                                         maxlen=max_sequence_len,
                                         padding='pre'))
```

```
print('max_len', max_sequence_len)
print('total-words',total_words)
```

```
max_len 12
total-words 672
```

create features and Labels

➢ Extract the last word of sequence and convert it to categorical from numerical

```python
# create predictors and label
predictors, label = input_sequences[:,:-1],input_sequences[:,-1]

label = to_categorical(label, num_classes=total_words)
```

```python
print(input_sequences)
```

```
[[    0     0     0 ...     0   265  1628]
 [    0     0     0 ...   265  1628   101]
 [    0     0     0 ...  1628   101   235]
 ...
 [    0     0     0 ...  2018    36     1]
 [    0     0     0 ...    36     1  6745]
 [    0     0     0 ...     1  6745    66]]
```

X Y

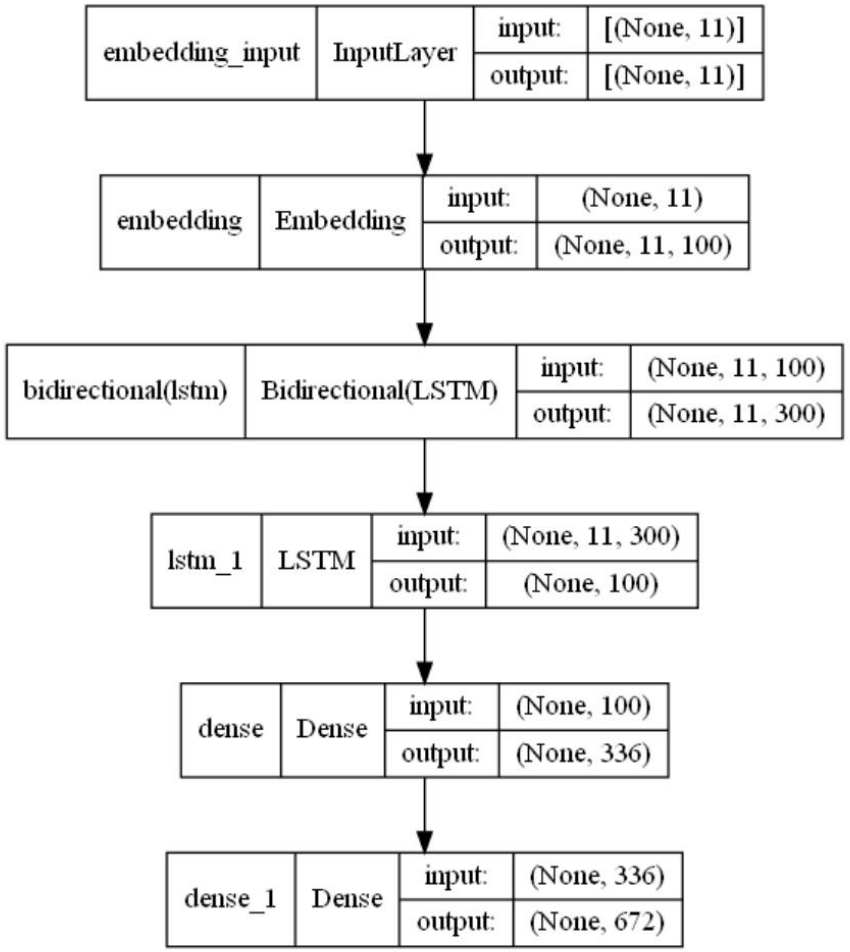
4) Bidirectional LSTM Model

- Let’s make a sequential model now with the first layer as the word embedding layer.
- 'return_sequence' is marked as ”True” so that the word generation keeps in consideration, previous and even the words coming ahead in the sequence.
  - ✓ The output layer has softmax to get the probability of the word to be predicted next.

```
model = Sequential()
model.add(Embedding(total_words, 100, input_length=max_sequence_len-1))
model.add(Bidirectional(LSTM(150, return_sequences = True)))
model.add(LSTM(100))
model.add(Dense(total_words/2, activation='relu' ))
model.add(Dense(total_words, activation='softmax'))
```

4) Bidirectional LSTM Model

```
plot_model(model, './fig_lab10_eng2.png', show_shapes=True)
```




```
print(model.summary())
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, 11, 100)           67200

 bidirectional_1 (Bidirectio  (None, 11, 300)          226800
 nal)

 gru_3 (GRU)                 (None, 100)               120600

 dense_4 (Dense)             (None, 336)               33936

 dense_5 (Dense)             (None, 672)               226464

=================================================================
Total params: 675,000
Trainable params: 675,000
Non-trainable params: 0
_________________________________________________________________
None
```

Model compile

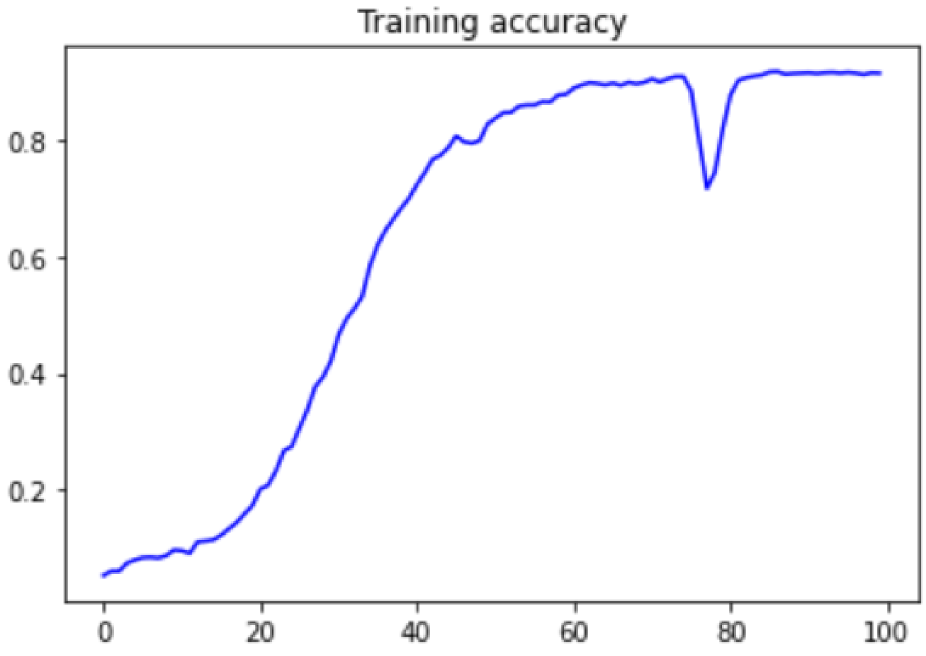
```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

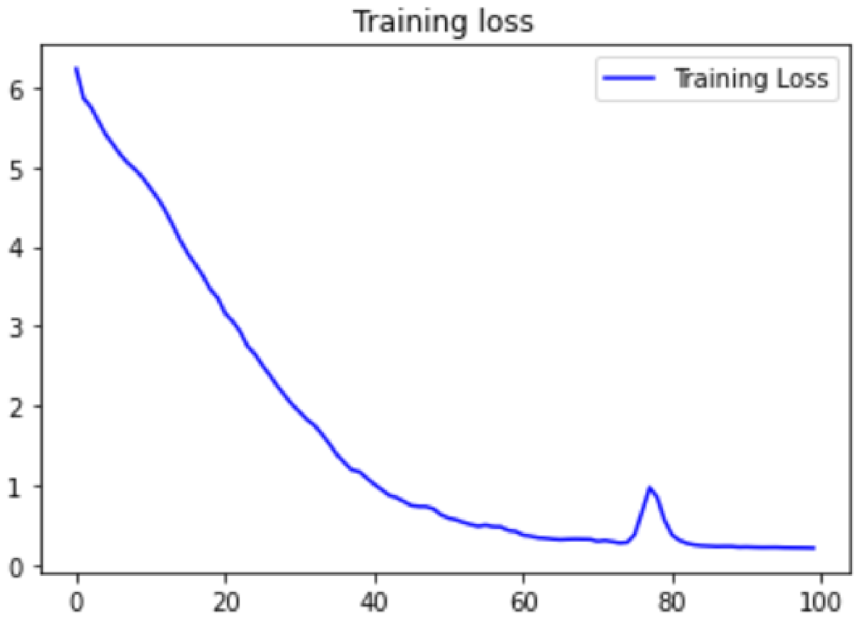
Model.fit and Elapsed Times

```
import time
start = time.perf_counter()
history = model.fit(predictors, label, epochs=100, verbose=1)
elapsed = time.perf_counter() - start
print('==============================₩n Elapsed %.3f seconds.' % elapsed)
```

```
Epoch 99/100
47/47 [==============================] - 2s 48ms/step - loss: 0.2143 - accuracy: 0.9176
Epoch 100/100
47/47 [==============================] - 2s 48ms/step - loss: 0.2117 - accuracy: 0.9169
==============================
 Elapsed 230.794 seconds.
```

Training Accuracy and Loss






```python
import matplotlib.pyplot as plt
acc = history.history['accuracy']
loss = history.history['loss']
epochs = range(len(acc))
plt.plot(epochs, acc, 'b', label='Training accuracy')
```

```python
plt.figure()
plt.plot(epochs, loss, 'b', label='Training Loss')
plt.title('Training loss')
plt.legend()
plt.show()
```

5) Test for the next generation

- 100 next words are generated this way
  - ✓ The seed will be taken at first and tokenized and padded on the token list.
  - ✓ Model is then used to predict with the token list as input.
  - ✓ Then most probable word is added to seed text and this happens for the next 100 words.

```
seed_text = "First Servingman: A strange one as ever I looked on: I cannot get him!"
next_words = 100
```

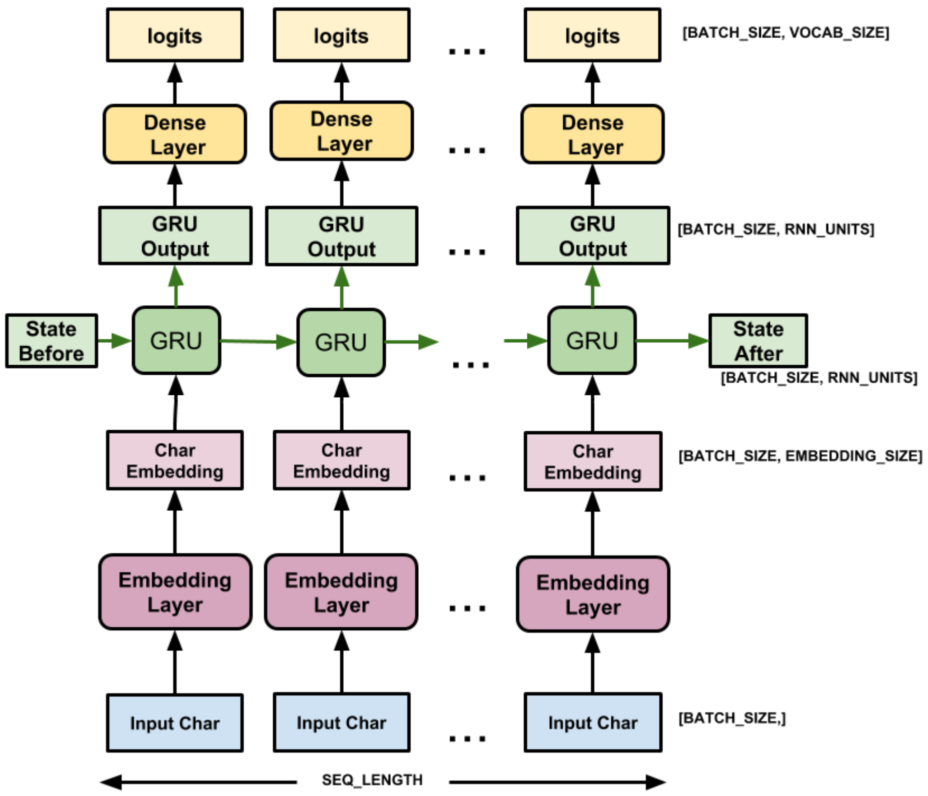
5) Test for the next generation

```python
for _ in range(next_words):
    token_list = tokenizer.texts_to_sequences([seed_text])[0]
    token_list = pad_sequences([token_list], maxlen=max_sequence_len-1, padding='pre')
    p_x = model.predict(token_list, verbose=0)
    predicted=np.argmax(p_x,axis=1)

    output_word = ""
    for word, index in tokenizer.word_index.items():
        if index == predicted:
            output_word = word
            break
    seed_text += " " + output_word
print(seed_text)
```

```
First Servingman: A strange one as ever I looked on: I cannot get him! in pray
you account remember you gods you you barren of all the rest were so mark
me say he fathers fathers fathers fathers fathers fathers did venture pray you s
ay there's which you to't own belly of man man which he remember remembe
r please please please can which our own price please ye caps please ye price
ye caps please ye caps please please ye trust ye 't please please ye 't please p
lease ye 't please please ye 't please please ye 't please please ye 't please ple
ase ye 't please please ye 't please please ye
```

HW : Optimized Model Architecture

- The output is not perfect as for training we took only a few lines of text.
  - ✓ Hence we can very well fine-tune it.



New GRU Model

```
model = Sequential()
model.add(Embedding(total_words, 100, input_length=max_sequence_len-1))
model.add(Bidirectional(GRU(150, return_sequences = True)))
model.add(GRU(100))
model.add(Dense(total_words/2, activation='relu' ))
model.add(Dense(total_words, activation='softmax'))
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
import time
start = time.perf_counter()
history = model.fit(predictors, label, epochs=100, verbose=1)
elapsed = time.perf_counter() - start
print('============================₩n Elapsed %.3f seconds.' % elapsed)
```

Code in Colab

Thanks