

# CHAPTER 5

## Table of contents

### 1. Introduction to RNN

### 2. Recurrent neural network

- 1) Architecture – Neurons with recurrence
- 2) Application

### 3. LSTM / GRU

- 1) SimpleRNN (vanilla RNN)
- 2) LSTM(Long Short-Term Memory)
- 3) GRU(Gated Recurrent Unit)
- 4) Study Case for LSTM, GRU

### 4. Text Analysis in RNN

- 1) Tokenization
- 2) Word Embedding / Token Embedding
- 3) Study Case(IMDB Movie Reviews Sentiment Analysis)

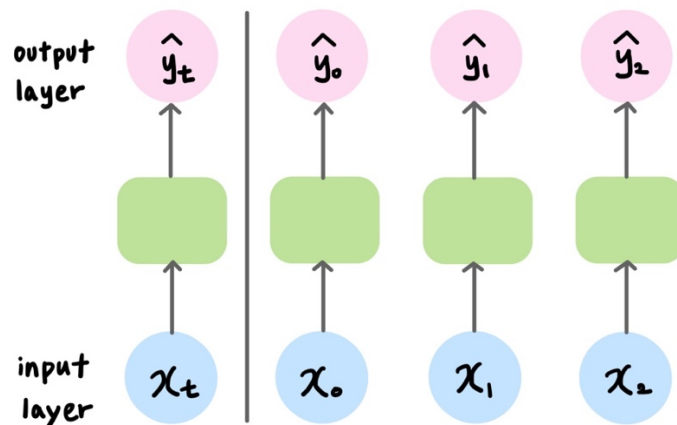
## 1. Introduction to RNN

RNN(Recurrent neural network)는 시퀀스 데이터(Sequential data)를 처리하는 모델이다. 예를 들어, 번역기에서 번역하고자 하는 문장들은 단어들의 시퀀스 형태이다. 이외에도, 주식이나 DNA 구조 등 시퀀스 데이터는 여러가지 형태로 존재하고 있다. 따라서, 현실세계에서 시퀀스 형태가 어떻게 분석할 수 있는지 알아보고, RNN model이 이런 상황을 어떻게 수행할 수 있으며, 그 예시를 보도록 한다.

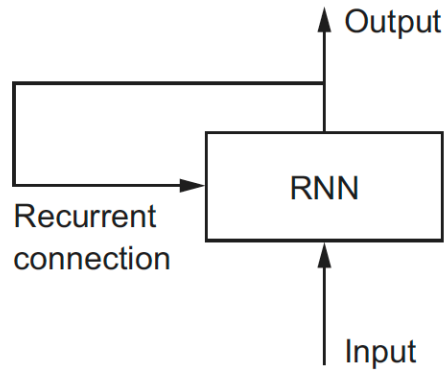
## 2. Recurrent Neural Network

### 1) Architecture - Neurons with recurrence

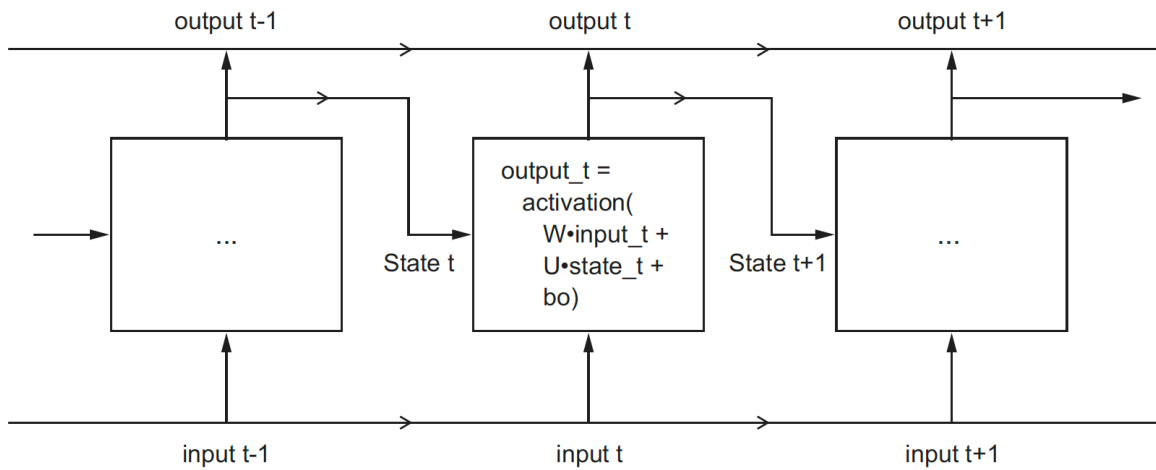
RNN model의 구조를 이해하기 위해선 recurrence concept에 대한 이해가 필요하다.



위의 그림은 신경망의 가장 기본적인 구조이다.  $x_t$ 가 입력되면  $\hat{y}_t$ 가 계산되는데 이런 구조에서는 각  $t$ 에 대한 output만 계산가능하며, input에서 output으로만 학습결과를 전달하는 전방향전달(feedforward) 방식을 사용한다. RNN model은  $t$ 와  $t+1$  사이의 관계를 분석하기 위해 고안된 모델이기 때문에 위의 구조로는 분석이 불가능하다. 우선,  $t$ 와  $t+1$ 의 관계에 대하여 이해해보도록 한다. 뒤에서 다루게 될 IMDB data는 영화 리뷰데이터로 관객의 영화감상평을 이용하여 영화에 대하여 긍정적인 평가를 했는지 부정적인 평가를 했는지 감정 분석한다. 관객이 남긴 문장을 처음부터 끝까지 읽는 과정에서 인간은 이전에 등장한 단어들의 내용을 기억하면서 문장의 의미를 해석하게 된다. 과거의 기억을 지속하면서 새로운 정보를 받아들임과 동시에 과거의 정보와 새로운 정보에 대한 새로운 관계까지 추론해 내는 것이다. 이런 관계가 recurrence concept이며, RNN은 이런 원리를 그대로 구현한 model이다.



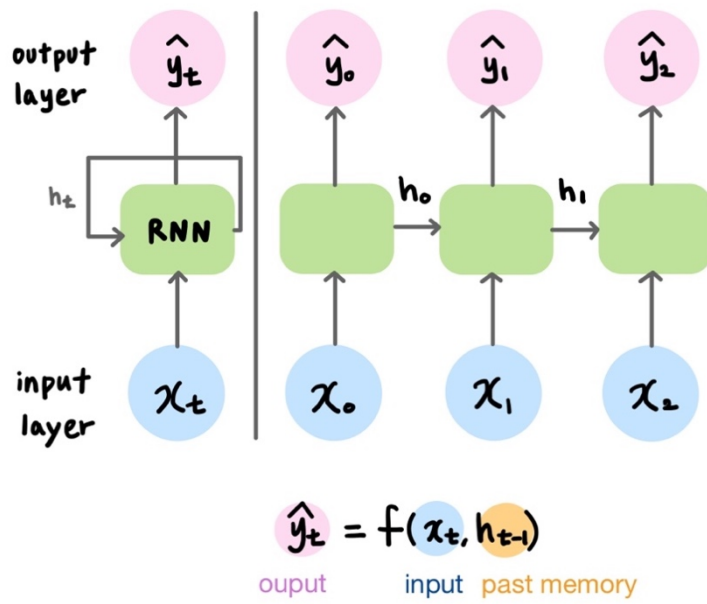
위의 그림은 RNN의 가장 간단한 구조이다. RNN model은 이전의 상태를 유지한 상태로 새로운 정보를 받아 들여야하기 때문에 위의 그림처럼 신경망 내부에서 루프가 생긴다. 루프를 병렬로 나열하면 아래의 그림과 같이 표현할 수 있다.



다음으로 RNN에서 일어나는 연산에 대해 정의한다.

타임 스텝  $t$ 에서 순환층(hidden layer)의 출력은 이전 타임 스텝의 모든 입력에 대한 함수이므로 이를 일종의 메모리 형태라고 할 수 있다. 타임 스텝에 걸쳐서 어떤 상태를 보존하는 신경망의 구성 요소를 메모리 셀(memory cell)이라고 한다.

일반적으로 타임 스텝  $t$ 에서의 셀의 상태  $h_t$  ( $h$ 는 hidden을 의미)는 그 타임 스텝의 입력과 이전 타임 스텝의 상태에 대한 함수이다. 즉,  $h_t = f(h_{t-1}, x_t)$ 이다. 타임 스텝  $t$ 에서의 출력  $y_t$ 도 이전 상태와 현재 입력에 대한 함수이다.



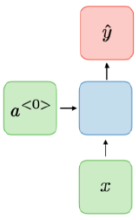
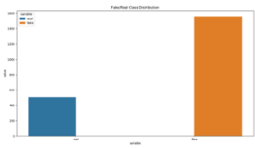
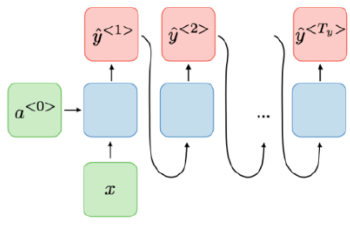
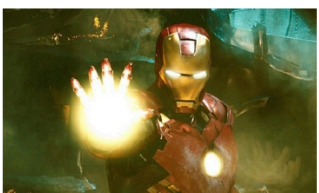
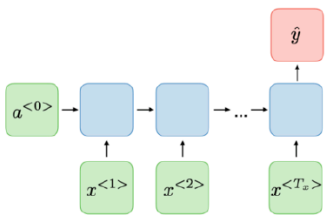
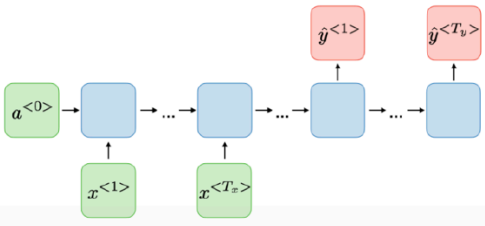
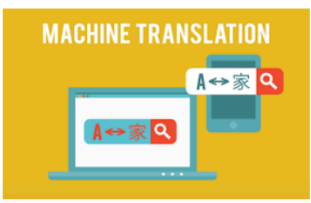
$h_t$ 는 현재 시점  $t$ 에서의 hidden layer를 거치며 계산된 값이다.  $h_t$ 를 계산하기 위해서는 두 개의 가중치가 필요하다. 하나는 입력값에 대한 가중치  $w_x$ 이고, 하나는 이전 시점인  $t-1$ 의 hidden layer에서 계산된  $h_{t-1}$ 을 위한 가중치  $w_h$ 이다. 이를 식으로 표현하면 다음과 같다.

$$\text{Hidden layer : } h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$$

정리하면, 입력값으로 들어온  $x_t$ 와 이전 시점에서 계산된  $h_{t-1}$ 을 이용하여 현재 시점  $t$ 의  $h_t$ 가 계산된다. 이 값은 두 번 쓰이게 되는데, 현재 시점  $t$ 의 출력값인  $\hat{y}_t$ 를 계산하는데 쓰이고, 다음 시점의  $h_{t+1}$ 을 구하는데 쓰인다.

## 2) Application

RNN 은 입력값과 출력값의 길이를 다르게 설계할 수 있어 모델을 여러가지 형태로 만들 수 있다.

Type of RNN	Illustration	Example
일 대 일(one-to-one)		Traditional neural network/ Binary Classification 
일 대 다(one-to-many)		Image Captioning  "There is an ironman"
다 대 일(many-to-one)		Sentiment Classification "I am happy with this water bottle." Positive "This is a bad investment." Negative "I am going to walk today." Neutral
다 대 다(many-to-many)		Machine Translation 

예를 들어 하나의 입력에 대해서 여러개의 출력을 의미하는 일 대 다(one-to-many) 구조의 모델은 하나의 이미지 입력에 대해서 사진의 제목을 출력하는 image captioning 작업에 사용할 수 있다. 사진의 제목은 단어들의 나열이므로 시퀀스를 출력한다고 할 수 있다. 또한, 단어의 시퀀스에 대해서 하나의 출력을 하는 다 대 일(many-to-one) 구조의 모델은 입력값이 긍정인지 부정인지 판별하는 감정 분류(Sentiment Classification)이나 메일이 정상메일인지 스팸메일인지 분류하는 스팸 메일 분류(spam detection) 등에 사용할 수 있다. 마지막으로 다 대 다(many-to-many) 구조의 모델은 입력문장으로부터 다른 나라의 언어로 번역된 문장을 출력하는 번역기같은 작업이 속한다.

## 3. LSTM/GRU

가장 기본적인 모델인 SimpleRNN 모델은 매우 단순해 실생활에 적용시키는데 있어 좋은 성능을 보이지 못한다. 또한, 긴 시퀀스로 RNN 을 훈련하기 위해선 많은 타임 스텝에 걸쳐 실행해야 하므로 펼쳐진 RNN 은 매우 깊은 신경망이 되어 vanishing gradient problem 이 발생한다. DNN(chapter 3)에서 이 문제를 완화하기 위해 사용한 기법으로 활성화 함수, 가중치 초기화, learning rate 설정, dropout, batch-normalization 등이 있었다. 활성화 함수의 경우 수렴하지 않는 함수를 사용하게 되면 큰 도움이 되지 않는다. 예를 들어, ReLU 를 이용하는 경우 모든 타임 스텝에서 가중치를 사용하기 때문에 ReLU 를 통과하면서 매 단계마다 조금씩 출력이 증가한다. 이런식으로 결과값이 폭주하는 문제가 발생할 수 있으므로 일반적으로 sigmoid 나 tanh 가 이용된다. 뒤에서 다루게 될 LSTM 이나 GRU 에서도 모두 두개의 활성화 함수만을 이용한다. 다음으로 SimpleRNN에서 발생하는 문제는 단기기억상실이다. RNN에서 매 타임 스텝을 거치면서 일부 정보가 사라지게 되는데, 어느 정도의 시간이 지나면 RNN 에서 첫번째 입력의 흔적은 거의 남아있지 않는다. 이런 문제를 해결하기 위해 장기 메모리를 가진 여러 종류의 셀이 연구되었다. 장기 메모리를 가진 셀에서 가장 인기 있는 모델이 LSTM(Long Short-term memory)과 GRU(Gated recurrent units)이다.

## 1) SimpleRNN (Vanilla RNN)

타임 스텝의 길이가  $T$  인 시퀀스를 RNN 에 훈련한다고 가정해보자. 하나의 메모리 셀에서  $h_t$  를 계산하는 식은 다음과 같다.

$$h_t = \tanh(x_t \cdot W_{xh} + h_{t-1} \cdot W_{hh} + b_h)$$

$x_t$  와  $h_{t-1}$  이 입력되면 각각 가중치  $W_{xh}, W_{hh}$  가 곱해진다.  $W_{xh}$  는 입력값( $x_t$ )으로부터 현재 셀상태( $h_t$ )를 계산하며  $W_{hh}$  는 이전의 셀 상태( $h_{t-1}$ )값으로 부터 현재의 셀 상태( $h_t$ ) 값을 계산한다. SimpleRNN 은 시퀀스의 길이가 길어지게 되면 vanishing gradient 문제로 학습이 어려워진다.

keras 를 이용하여 간단한 RNN model 을 만들어 보도록한다.

```
# import Library
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import SimpleRNN, LSTM, Dense
from tensorflow.keras import Sequential
```

다음으로 RNN model 에 넣기 위한 데이터를 생성한다. 데이터는 1000 개의 시계열 데이터이며 함수  $y = \sin(2x) + \cos(\frac{x}{2})$ 로 정의한다.

```
# data 생성

x = np.arange(0, 100, 0.1)
y = np.sin(2*x) + np.cos(x/2)

seq_data = y.reshape(-1,1)

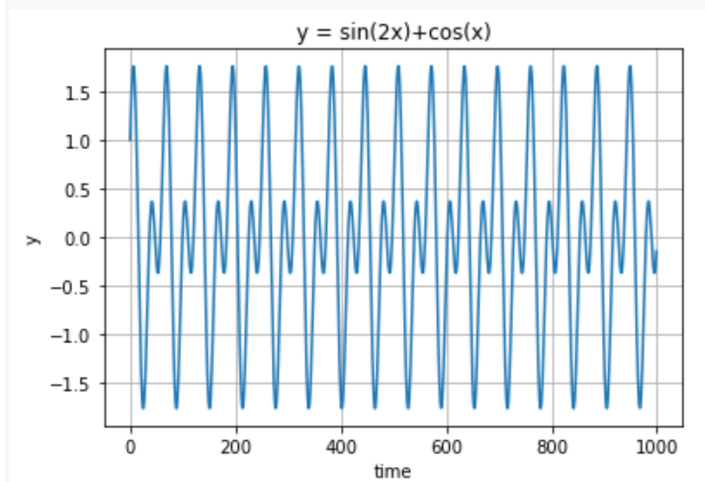
print(seq_data.shape)
print(seq_data[:5])
```

위의 코드에서 정의된 `seq_data` 는 RNN 에 데이터를 입력하기 위해 필수적인 단계이다. RNN 에 입력되는 데이터는 (batch size, time steps, input\_dim) 3 차원 텐서의 형태를 띄어야 한다.

생성한 데이터를 시각화한다.

```
plt.grid()
plt.title('y = sin(2x)+cos(x)')
plt.xlabel('time')
plt.ylabel('y')
plt.plot(seq_data)

plt.show()
```



시계열 데이터이므로 주기성이 관찰되고 있다. 다음으로 데이터를 `x, y` 로 분리한 후 train data 와 test data 로 나눈다.

```
# function for 3D tensor

def seq2dataset(seq, time_steps, input_dim):

    X = []
    Y = []

    for i in range(len(seq)-(time_steps+input_dim)+1):

        x = seq[i:(i+time_steps)]
        y = (seq[i+time_steps+input_dim-1])

        X.append(x)
        Y.append(y)

    return np.array(X), np.array(Y)
```

위에서 정의된 함수는 데이터를 `x, y` 로 분리하여 RNN 입력에 적합한 데이터의 형태로 변환한다. `np.array()`는 2 차원 행렬인 `x.shape` 을 (batch size, time steps, input dims) 형태인 3D tensor 의 형태로 만들어준다.

```
t = 20 # time steps
h = 1 # input_dim
```

```
X, Y = seq2dataset(seq_data, t, h)
```

```
print(X.shape, Y.shape)
```

```
(980, 20, 1) (980, 1)
```

$t = 20$  으로 설정하였으므로 time step 이 20 만큼 학습하면  $h_t$ 가 출력되는 것을 의미한다. 또한, 20 개의 데이터가 하나의  $x$  로 취급되면서 데이터의 길이가 20 만큼 줄어든 것을 확인할 수 있다. 다음으로 데이터를 train data 와 test data 로 분리한다.

```
# split train/test data
```

```
split_ratio = 0.8
```

```
split = int(split_ratio*len(X))
```

```
x_train = X[0:split]
```

```
y_train = Y[0:split]
```

```
x_test = X[split:]
```

```
y_test = Y[split:]
```

```
print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

```
(784, 20, 1) (784, 1) (196, 20, 1) (196, 1)
```

980 개의 데이터를 8:2 의 비율로 train data 와 test data 로 분리하였다. 데이터가 완성되었으므로 모델을 정의한다.

```
model = Sequential()
```

```
#model.add(SimpleRNN(units=128, activation='tanh', input_shape=x_train[0].shape))
```

```
model.add(SimpleRNN(units=128, activation='tanh', input_shape=(20,1)))
```

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

```
model.summary()
```

```
Model: "sequential_11"
```

Layer (type)	Output Shape	Param #
simple_rnn_16 (SimpleRNN)	(None, 128)	16640
dense_5 (Dense)	(None, 1)	129

=====  
 Total params: 16,769  
 Trainable params: 16,769  
 Non-trainable params: 0

NOTE)

기본적으로 keras의 순환층은 최종 출력만 반환한다. 타임 스텝마다 출력을 반환하려면 `return_sequence = True`



로 지정해야 한다.

NOTE)

위의 모델에서 simpleRNN 층을 추가하기 위해선 `return_sequences = T` 옵션을 추가해야 한다.

SimpleRNN 층의 노드는 128 개로 설정하였으며, 각 노드의 활성화 함수는 `tanh` 이다. 출력되는 `y` 의 값은 1 개이므로 dense 층에서의 노드는 1로 설정한다.

```
model.compile(loss='mse', optimizer='adam', metrics=['mae'])
```

```
from datetime import datetime
```

```
start_time = datetime.now() # calculate training time
```

```
hist = model.fit(x_train, y_train, epochs=100, validation_data=(x_test, y_test))
```

```
end_time = datetime.now()
```

```
print('Elapsed Time => ', end_time-start_time)
```

```
plt.title('Loss')
```

```
plt.plot(hist.history['loss'], label='loss')
```

```
plt.plot(hist.history['val_loss'], label='val_loss')
```

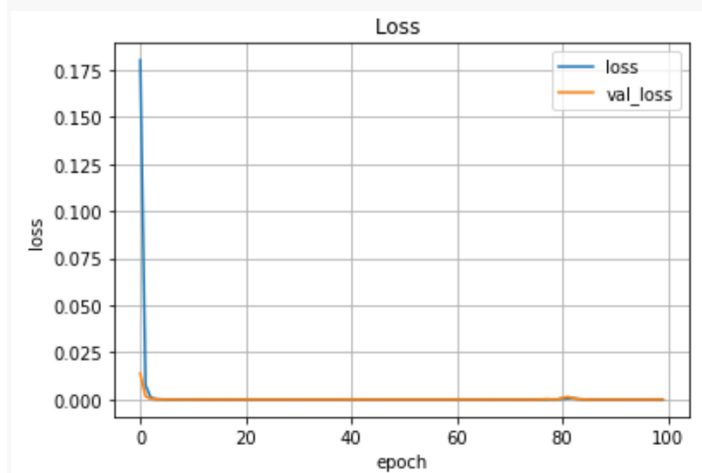
```
plt.xlabel('epoch')
```

```
plt.ylabel('loss')
```

```
plt.grid()
```

```
plt.legend(loc='best')
```

```
plt.show()
```



loss 값으로 설정한 mse 가 0 에 거의 수렴하고 있다. 따라서, 예측값이 실제값을 잘 추정하고 있음을 의미한다.

```
pred = model.predict(x_test)
```

```
(196, 1)
```

예측값과 실제값을 임의로 5 개 추출하여 오차를 관찰하도록 한다.

```
# setting random seed
np.random.seed(1886)

rand_idx = np.random.randint(0, len(y_test), size=5)

print('random idx = ', rand_idx, '\n')

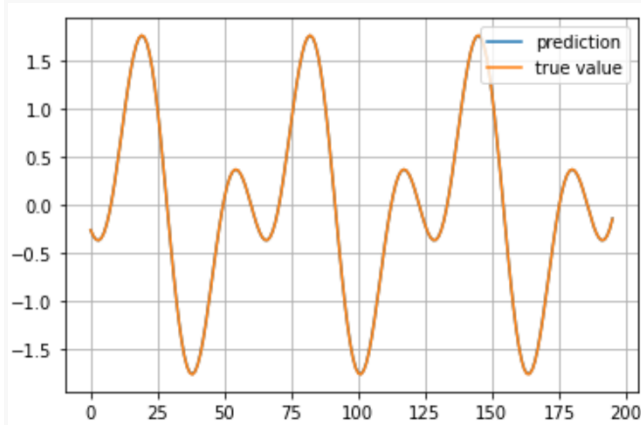
print('pred = ', pred.flatten()[rand_idx])
print('label = ', y_test.flatten()[rand_idx])

random_idx = [166 62 33 92 105]

pred = [-1.6262805 -0.1958181 -1.2127525 -0.19390522 -1.3780198 ]
label = [-1.62753387 -0.19606911 -1.21355631 -0.19342387 -1.37960403]

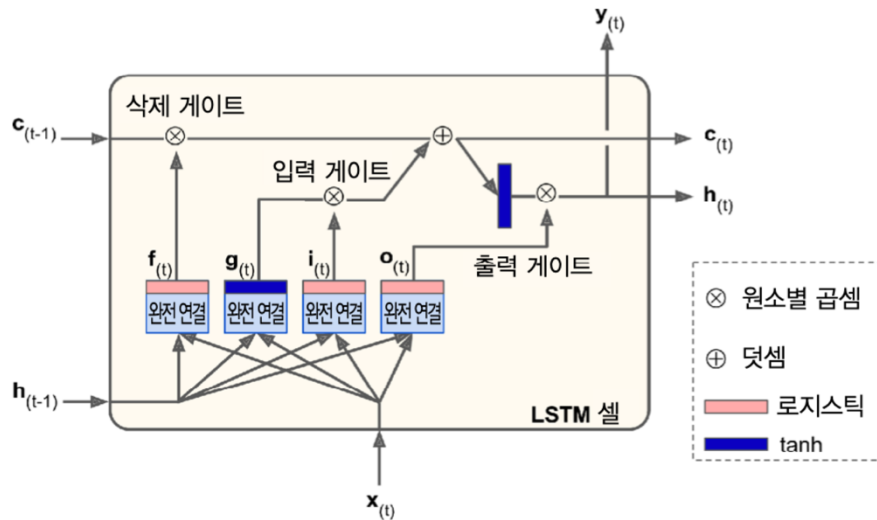
plt.plot(pred, label='prediction')
plt.plot(y_test, label='true value')
plt.grid()
plt.legend(loc='best')

plt.show()
```



예측값이 실제값을 잘 추정하고 있다.

## 2) LSTM(Long Short-Term Memory)



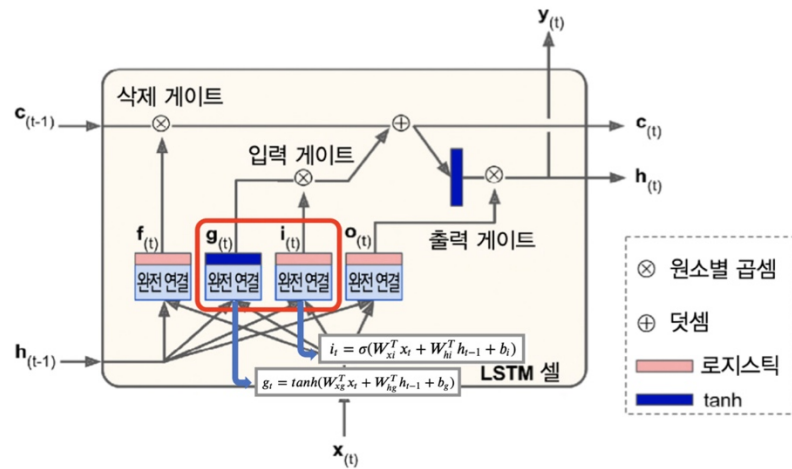
위의 그림은 LSTM의 구조를 나타낸 그림이다. SimpleRNN은 장기 기억을 보존하지 못한다는 문제가 있었는데, 이를 보완한 RNN model이 LSTM(Long Short-Term Memory)이다. LSTM은 은닉층의 메모리 셀에 입력 게이트, 삭제 게이트, 출력 게이트를 추가하여 불필요한 정보를 지우고 장기 상태에 저장할 정보를 정한다. 결론적으로 LSTM은 은닉 상태(hidden state)를 계산하는 식이 좀 더 복잡해지며 장기 기억을 저장하기 위해 셀 상태(cell state,  $c_t$ )라는 값을 추가하게 된다. 장기 기억  $c_{t-1}$ 은 네트워크를 왼쪽에서 오른쪽으로 메모리 셀을 왼쪽에서 오른쪽으로 관통하면서, 삭제 게이트를 지나 일부 기억을 읽고, 그 다음 덧셈 연산으로 새로운 기억(입력 게이트에서 선택한 기억)을 일부 추가한다. 만들어진  $c_t$ 는 다른 추가 변환 없이 바로 출력으로 보내진다. 이 과정이 각 셀을 통과할 때마다 반복되면서 일부 기억이 삭제되거나 추가된다. 또한, 덧셈 연산 이후 장기 기억상태는 복사되어 tanh 함수로 전달된다. 그 다음 출력 게이트에 의해 걸러진 정보는 단기 상태  $h_t$  정보로 만들어진다. 다음으로 새로운 기억이 들어오는 곳과 각 게이트가 어떻게 작동하는지 보도록 한다.

새로 추가된 3개의 게이트(삭제 게이트, 입력 게이트, 출력 게이트)는 은닉 상태(hidden state,  $h_t$ )의 값과 셀 상태( $c_t$ )의 값을 구하기 위해 사용된다. 이 3개의 게이트는 공통적으로 시그모이드 함수를 사용하는데, 시그모이드 함수를 지나면 0과 1 사이의 값이 나와 이 값들을 가지고 게이트를 조절한다. 다음으로 각 게이트에 들어가기전에 4개의 완전 연결층에서 일어나는 과정에 대해 본다.

먼저 현재 입력 벡터  $x_t$ 와 이전의 단기 상태  $h_{t-1}$ 은 네 개의 다른 완전 연결층에 주입된다. 가장 중요한 역할을 하는 층은  $g_t$ 를 출력하는 층이다. 이 층은 현재 입력  $x_t$ 와 이전의 (단기)상태  $h_{t-1}$ 을 분석하는 역할을 담당한다. SimpleRNN에서는 이 층 이외에 다른 층은 존재하지 않아 바로  $y_t$ 와  $h_t$ 를 출력한다. 그러나, LSTM에서는 이 층의 출력이 곧바로 나가지 않고, 장기 기억에서 가장 중요한 부분이 저장된다(나머지는 버린다).

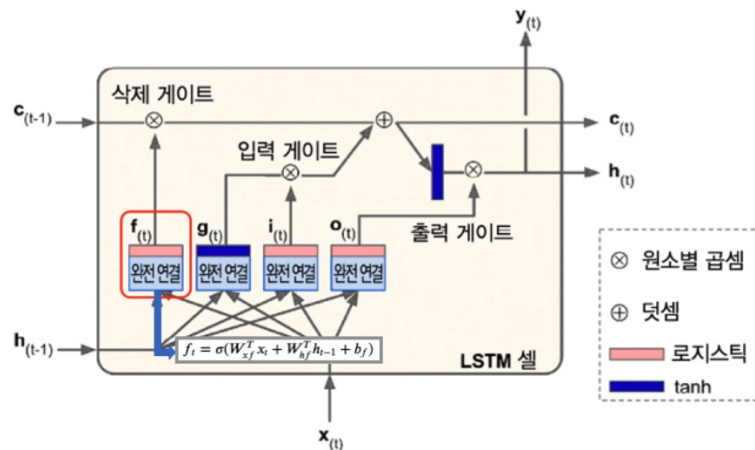
세 개의 다른 층은 게이트 제어기(gate controller)이다. 이 층들은 로지스틱 활성화 함수를 사용하기 때문에 출력의 범위가 0에서 1 사이이다. 이들의 출력은 원소별 곱셈 연산으로 주입되어, 0을 출력하면 게이트를 닫고 1을 출력하면 게이트를 연다. 이제 각 층들을 지나 출력된 값들이 게이트에 입력되어 어떤 작동이 일어나는지 알아보도록 한다.

## a) 입력 게이트 (input gate)



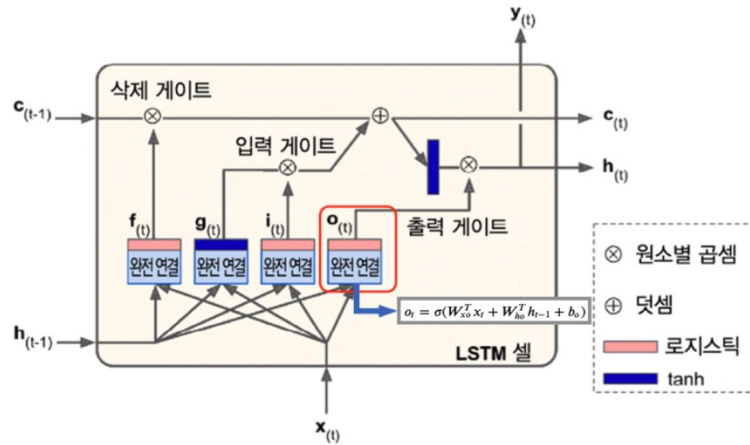
입력 게이트는 현재의 정보를 기억하는 게이트이다. 우선, 현재 시점  $t$ 의  $x_t$  값과 입력 게이트로 이어지는 가중치  $W_{xi}$ 를 곱한 값과 이전 시점  $t-1$ 의 은닉 상태가 입력 게이트로 이어지는 가중치  $W_{hi}$ 를 곱한 값을 더하여 sigmoid 함수를 지난다. 이를  $i_t$ 라고 한다. 그리고 현재 시점  $t$ 의  $x$  값과 입력 게이트로 이어지는 가중치  $W_{xg}$ 를 곱한 값과 이전 시점  $t-1$ 의 은닉 상태가 입력 게이트로 이어지는 가중치  $W_{hg}$ 를 곱한 값을 더하여 tanh 함수를 지난다. 이를  $g_t$ 라고 한다. sigmoid 함수를 지나면  $i_t$ 는 0 과 1 사이의 값을 가지게 되고, tanh 함수를 지나면  $g_t$ 는 -1 과 1 사이의 값을 가진다. 이때, 입력 게이트를 제어하는 값은  $i_t$ 로  $g_t$ 의 어느 부분이 장기 상태에 더해져야 하는지를 제어한다.

## b) 삭제 게이트(forget gate)



삭제 게이트는 장기 상태의 어느 부분이 삭제되어야 하는지 제어한다. 현재 시점  $t$ 의  $x_t$  값에 삭제 게이트로 연결되는 가중치인  $W_{xf}$ 를 곱한 후 이전 시점  $t-1$ 의 은닉 상태인  $h_{t-1}$ 에 가중치  $W_{hf}$ 를 곱한 값에 편향하여 sigmoid 함수를 지나게 된다. sigmoid 함수를 지나면 0 과 1 사이의 값이 나오게 되는데, 이 값은 삭제 과정을 거친 정보의 양을 의미한다. 0 에 가까울수록 정보가 많이 삭제된 것이고 1 에 가까울수록 정보를 기억한 것이다.

## c) 출력 게이트(output gate)



출력 게이트는  $x_t$ 의 값에 가중치  $W_{xo}$ 를 곱하고 이전 시점의 은닉 상태  $h_{t-1}$ 에 가중치  $W_{ho}$ 가 곱해진 후 편향을 더하여 sigmoid 함수를 지난다. 여기에 셀 상태의 값인  $c_t$ 가  $\tanh$  함수를 지나면서 -1 과 1 사이의 값이 되고, 이 값은 출력 게이트의 값과 연산되면서 현재 시점  $t$ 의 은닉 상태값인  $h_t$ 가 된다. 정리하면,  $o_t$ 값은 장기 상태의 어느 부분을 읽어서 현재 시점  $t$ 의  $h_t$ 와  $y_t$ 를 출력할지 제어하는 역할을 한다.

## d) 셀 상태(cell state)

셀 상태  $c_t$ 를 구하기 위한 식은 다음과 같다.

$$c_t = f_t \otimes C_{t-1} + i_t \otimes g_t$$

입력 게이트에서 계산된  $i_t, g_t$ 에 대하여 원소별 곱(entrywise product)을 진행한다. 그 다음, 입력 게이트에서 선택된 기억을 삭제 게이트의 결과값과 더한다. 이 값이 현재 시점  $t$ 의 셀 상태  $c_t$ 라고 하며, 이 값은 다음 시점인  $t+1$  시점의 LSTM 셀로 넘어간다.

$c_t$ 를 결정하는 삭제 게이트와 입력 게이트의 역할에 대해 알아보자. 만약 삭제 게이트의 출력값인  $f_t$ 가 0 이 된다면 삭제 게이트가 열리지 않기 때문에 이전 시점의 셀 상태 값인  $c_{t-1}$ 이 삭제 게이트를 통과하지 못하면서 현재 시점의 셀 상태 값을 결정하는데 영향을 미치지 않는다. 따라서, 오직 입력 게이트의 결과만이 현재 시점의 셀 상태 값  $c_t$ 을 결정할 수 있다. 다시말해, 삭제 게이트가 완전히 닫히고 입력 게이트만 연 상태를 의미한다. 반대로 입력 게이트의  $i_t$  값이 0 이면 현재 시점의 셀 상태의 값  $c_t$ 는 오직 이전 시점의 셀 상태의 값  $c_{t-1}$ 의 값에만 의존한다. 이는 입력 게이트를 완전히 닫고 삭제 게이트만을 연 상태를 의미한다. 결과적으로 삭제 게이트는 이전 시점의 입력을 얼마나 반영할지를 의미하고, 입력 게이트는 현재 시점의 입력을 얼마나 반영할지를 결정한다.

하나의 타임 스텝에 대해 셀의 장기 상태와 단기 상태 그리고 출력을 계산하는 식을 정리하면 다음과 같다.

$$\begin{aligned} i_t &= \sigma(W_{xi}^T x_t + W_{hi}^T h_{t-1} + b_i) \\ f_t &= \sigma(W_{xf}^T x_t + W_{hf}^T h_{t-1} + b_f) \\ o_t &= \sigma(W_{xo}^T x_t + W_{ho}^T h_{t-1} + b_o) \\ g_t &= \tanh(W_{xg}^T x_t + W_{hg}^T h_{t-1} + b_g) \\ c_t &= f_t \otimes C_{t-1} + i_t \otimes g_t \\ y_t &= h_t = o_t \otimes \tanh(c_t) \end{aligned}$$

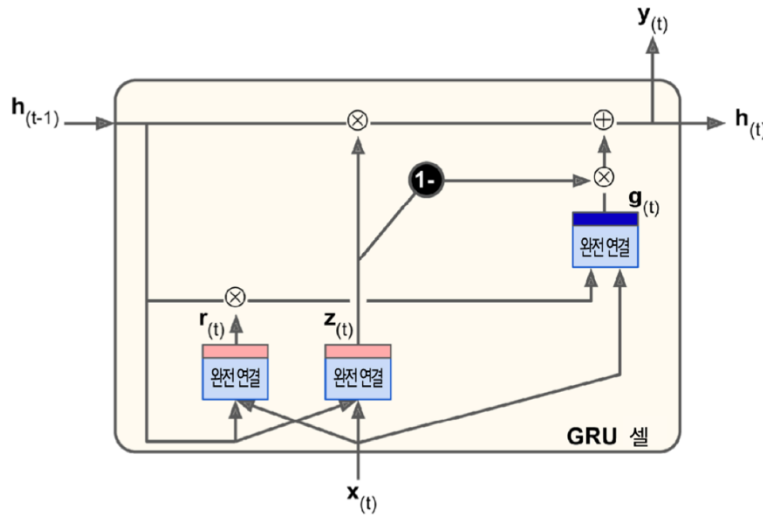
NOTE)

- $W_{xi}, W_{xf}, W_{xo}, W_{xg}$  는 입력 벡터  $x_t$ 에 각각 연결된 네 개 층의 가중치 행렬이다.
- $W_{hi}, W_{hf}, W_{ho}, W_{hg}$  는 이전 단계 상태  $h_{t-1}$ 에 각각 연결된 네 개 층의 가중치 행렬이다
- $b_i, b_f, b_o, b_g$  는 네 개 층 각각에 대한 편향이다.

정리하면, LSTM 셀은 중요한 입력을 인식하고(입력 게이트의 역할), 장기 상태에 저장하고, 필요한 기간 동안 이를 보존하며(삭제 게이트의 역할), 필요할 때마다 이를 추출하기 위해 학습한다. 따라서, LSTM 셀은 시계열, 긴 텍스트, 오디오 녹음 등에서 장기 패턴을 잡아내는데 좋은 성과를 낸다.

### 3) GRU(Gated Recurrent Unit)

LSTM 에서는 출력, 입력, 삭제 게이트 3 개가 존재했다. GRU 는 LSTM 의 간소화된 버전으로 업데이트 게이트와 리셋 게이트 두 가지 게이트만이 존재한다.



그림에서 알 수 있듯 셀 상태  $c_t$  와 은닉 상태  $h_t$  는 하나로 합쳐져  $h_t$  만이 존재한다. 또한,  $z_t$  가 삭제 게이트와 입력 게이트를 모두 제어한다.  $z_t$  가 1 을 출력하면 삭제 게이트( $=1$ )가 열리고 입력 게이트( $1-1 = 0$ )가 닫힌다. 반면,  $z_t$  가 0 을 출력하면 삭제 게이트( $=0$ )가 닫히고 입력 게이트( $1-0=1$ )가 닫힌다. 다시말해 기억이 저장될 때마다 저장될 위치가 먼저 삭제된다.

GRU 에는 출력 게이트가 존재하지 않기 때문에 매번 전체 상태를 그대로 출력하게 된다. 그러나 이전 상태인  $h_{t-1}$  의 어느 부분이  $g_t$  에 노출될지 제어하는 게이트인  $r_t$  가 존재한다.

하나의 타임 스텝에 대해 셀에서 일어나는 계산을 정리하면 다음과 같다.

$$z_t = \sigma(W_{xz}^T x_t + W_{hz}^T h_{t-1} + b_z)$$

$$r_t = \sigma(W_{xr}^T x_t + W_{hr}^T h_{t-1} + b_r)$$

$$g_t = \tanh(W_{xg}^T x_t + W_{hg}^T (r_t \otimes h_{t-1}) + b_g)$$

$$h_t = z_t \otimes h_{t-1} + (1 - z_t) \otimes g_t$$

#### 4) Study Case for LSTM, GRU

삼성전자 주가 데이터를 이용하여 LSTM 과 GRU 의 성능을 비교해보자. 주가 예측 분석의 경우 feature 로 가격이평선이 자주 사용되므로 데이터에 3 일(3MA), 5 일(5MA) 가격이평선을 추가한다.

```
# import library

import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, GRU, Dense, Dropout
from sklearn.metrics import r2_score

import plotly.graph_objects as go
```

```
# Load dataset

raw_df = pd.read_csv('./samsung_electronics_.csv')

print(raw_df.shape)

raw_df.head()

(5528, 7)
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2000-09-01	5540.0	5600.0	5420.0	5540.0	4192.145020	49955000
1	2000-09-04	5440.0	5550.0	5310.0	5340.0	4040.806396	28625000
2	2000-09-05	5260.0	5340.0	5030.0	5150.0	3897.030029	79765000
3	2000-09-06	5200.0	5280.0	5090.0	5150.0	3897.030029	53120000
4	2000-09-07	4940.0	4980.0	4760.0	4790.0	3624.616699	83905000

데이터의 개수는 총 5528 개이며 변수는 총 7 개이다. 2000 년 9 월부터 2022 년 9 월까지 22 년치 정보를 가지고 있다.

```
# add columns

raw_df['3MA'] = raw_df['Close'].rolling(window = 3).mean()
raw_df['5MA'] = raw_df['Close'].rolling(window = 5).mean()
raw_df.head()
```

	Date	Open	High	Low	Close	Adj Close	Volume	3MA	5MA
0	2000-09-01	5540.0	5600.0	5420.0	5540.0	4192.145020	49955000	NaN	NaN
1	2000-09-04	5440.0	5550.0	5310.0	5340.0	4040.806396	28625000	NaN	NaN
2	2000-09-05	5260.0	5340.0	5030.0	5150.0	3897.030029	79765000	5343.333333	NaN
3	2000-09-06	5200.0	5280.0	5090.0	5150.0	3897.030029	53120000	5213.333333	NaN
4	2000-09-07	4940.0	4980.0	4760.0	4790.0	3624.616699	83905000	5030.000000	5194.0

위의 데이터에서 새로 생성된 feature 인 3MA 과 5MA 의 데이터가 각각 3 번째와 5 번째 observation 부터 시작되는 것을 볼 수 있다. 구해진 3MA 과 5MA 의 index 를 하나씩 뒤로 미룰 필요가 있다. 예를들어, 2000-09-05 에서 구해진 3MA 의 경우 2000-09-01, 2000-09-04, 2000-09-05 3 일간의 close 의 평균이다. 이때, 예측하고자 하는 값은 미래의 값인 2000-09-06 일의 close 이므로 3MA, 5MA 을 미래의 값을 예측하기 위해 이용하기 위해선 각 값을 하루씩 뒤로 미뤄야 한다.

```
import numpy as np
raw_df['3MA_new'] = np.NaN
raw_df['5MA_new'] = np.NaN
```

```
for i in range(len(raw_df)-1):
    raw_df.iloc[i+1,9] = raw_df.iloc[i,7]
    raw_df.iloc[i+1,10] = raw_df.iloc[i,8]
```

```
df2 = raw_df.copy()
raw_df.head(6)
```

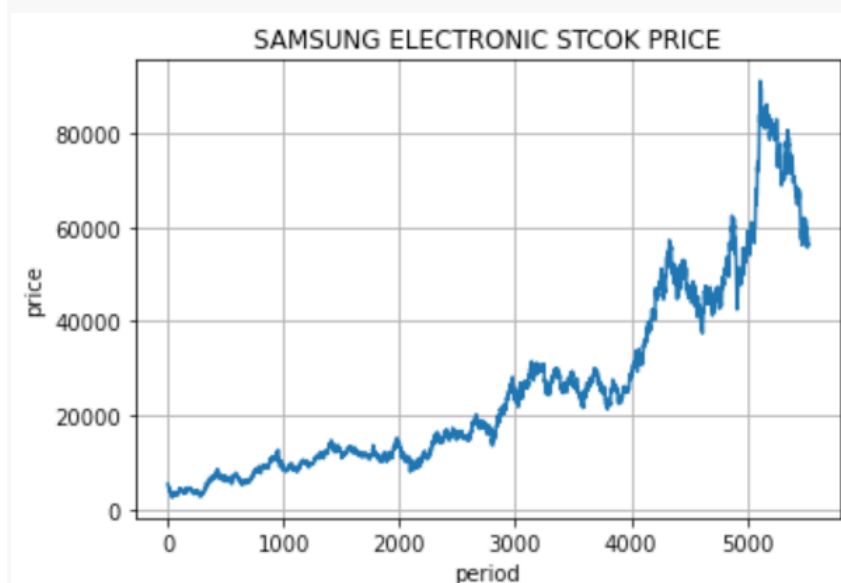
	Date	Open	High	Low	Close	Adj Close	Volume	3MA	5MA	3MA_new	5MA_new
0	2000-09-01	5540.0	5600.0	5420.0	5540.0	4192.145020	49955000	NaN	NaN	NaN	NaN
1	2000-09-04	5440.0	5550.0	5310.0	5340.0	4040.806396	28625000	NaN	NaN	NaN	NaN
2	2000-09-05	5260.0	5340.0	5030.0	5150.0	3897.030029	79765000	5343.333333	NaN	NaN	NaN
3	2000-09-06	5200.0	5280.0	5090.0	5150.0	3897.030029	53120000	5213.333333	NaN	5343.333333	NaN
4	2000-09-07	4940.0	4980.0	4760.0	4790.0	3624.616699	83905000	5030.000000	5194.0	5213.333333	NaN
5	2000-09-08	4900.0	4930.0	4790.0	4880.0	3692.720703	57170000	4940.000000	5062.0	5030.000000	5194.0

22 년동안 주가 변화를 시각화한다.

```
plt.title('SAMSUNG ELECTRONIC STCOK PRICE')
plt.ylabel('price')
plt.xlabel('period')
plt.grid()
```

```
plt.plot(raw_df['Close'], label='Close')
```

```
plt.show()
```





```
fig = go.Figure()
fig.add_trace(go.Scatter(x = raw_df['Date'], y = raw_df['Close']))
fig.show()
```



위의 그림은 마우스를 그래프 위에 올려두면 각 Adj Close 의 발생 날짜와 수치를 볼 수 있다.

다음으로, RNN 에 입력하기 위한 데이터를 만들기 위해 전처리 과정이 필요하다. 전처리 과정에서는 우선 outlier 와 missing data 를 확인 후 데이터를 보강하거나 삭제한다. 다음으로, 정규화 작업 후 RNN model 에 넣기 위한 형태로 정의한다.

```
raw_df.describe()
```

	Open	High	Low	Close	Adj Close	Volume	3MA	5MA	3MA_new	5MA_new
count	5528.000000	5528.000000	5528.000000	5528.000000	5528.000000	5.528000e+03	5526.000000	5524.000000	5525.000000	5523.000000
mean	25981.065485	26239.493488	25717.206946	25975.984081	22684.264165	2.094515e+07	25974.200748	25972.468139	25968.730015	25966.904581
std	20430.668835	20599.558283	20254.349994	20414.717698	20074.980642	1.440244e+07	20409.319297	20404.506985	20407.114019	20402.163765
min	2540.000000	2760.000000	2420.000000	2730.000000	2065.804688	0.000000e+00	2790.000000	2810.000000	2790.000000	2810.000000
25%	10820.000000	10960.000000	10680.000000	10820.000000	8268.144287	1.170836e+07	10815.000000	10840.000000	10813.333333	10840.000000
50%	18050.000000	18300.000000	17810.000000	18030.000000	14442.782715	1.726500e+07	18043.333333	18036.000000	18040.000000	18024.000000
75%	37755.000000	38170.000000	37427.500000	37885.000000	32633.070801	2.599000e+07	37916.666667	37938.000000	37866.666667	37878.000000
max	90300.000000	96800.000000	89500.000000	91000.000000	88367.835938	1.642150e+08	90433.333333	89960.000000	90433.333333	89960.000000

주식 거래량을 나타내는 변수인 Volume 의 최솟값이 0 임을 알 수 있다. 주식과 같은 금융데이터에서는 Volume 이 0 으로 나타나는 경우 일반적으로 missing value 로 취급한다.

```
# Missing Data
raw_df.isnull().sum()
```

```
Date      0
Open      0
```

```

High      0
Low       0
Close     0
Adj Close 0
Volume    0
3MA       2
5MA       4
3MA_new   3
5MA_new   5
dtype: int64

```

missing data 는 3MA 와 5MA 에서 발생하고 있다. 3MA 와 5MA 컬럼의 생성과정을 고려해보았을때 missing data 가 나오는 것은 필연적이다. 따라서, missing data 를 삭제한다.

```
raw_df.loc[raw_df['Volume']==0]
```

	Date	Open	High	Low	Close	Adj Close	Volume	3MA	5MA	3MA_new	5MA_new
6	2000-09-11	4880.0	4880.0	4880.0	4880.0	3692.720703	0	4850.000000	4970.0	4940.000000	5062.0
7	2000-09-12	4880.0	4880.0	4880.0	4880.0	3692.720703	0	4880.000000	4916.0	4850.000000	4970.0
8	2000-09-13	4880.0	4880.0	4880.0	4880.0	3692.720703	0	4880.000000	4862.0	4880.000000	4916.0
22	2000-10-03	3810.0	3810.0	3810.0	3810.0	2883.045898	0	3886.666667	3950.0	3970.000000	3988.0
31	2000-10-16	3030.0	3030.0	3030.0	3030.0	2292.816406	0	3066.666667	3216.0	3130.000000	3348.0
...	...	...	...	...	...	...	...	...	...	...	...
5391	2022-02-28	71900.0	71900.0	71900.0	71900.0	71095.539063	0	71766.666667	72780.0	72566.666667	73260.0
5394	2022-03-04	72900.0	72900.0	72900.0	72900.0	72084.343750	0	72500.000000	72260.0	72166.666667	71980.0
5397	2022-03-10	69500.0	69500.0	69500.0	69500.0	68722.390625	0	69700.000000	70980.0	70833.333333	71420.0
5400	2022-03-15	70200.0	70200.0	70200.0	70200.0	69414.554688	0	70133.333333	69880.0	69900.000000	69860.0
5402	2022-03-17	70400.0	70400.0	70400.0	70400.0	69612.320313	0	70333.333333	70240.0	70266.666667	70060.0

115 rows x 11 columns

Volume 의 값이 0 인 데이터가 총 115 개이다. 이는 missing data 가 115 개임을 의미한다.

```

for col in raw_df.columns:

    missing_rows = raw_df.loc[raw_df[col]==0].shape[0]
    print(col + ': ' + str(missing_rows))

```

```

Date: 0
Open: 0
High: 0
Low: 0
Close: 0
Adj Close: 0
Volume: 115
3MA: 0
5MA: 0
3MA_new: 0
5MA_new: 0

```

0 이 존재하는 column 은 Volume 이 유일하다. Volume 의 0 값을 NaN 으로 바꾼 후 Missing data 로 처리하여 raw 를 삭제한다.

```
raw_df['Volume'] = raw_df['Volume'].replace(0, np.nan)
```

```
for col in raw_df.columns:

    missing_rows = raw_df.loc[raw_df[col]==0].shape[0]
    print(col + ': ' + str(missing_rows))
```

```
Date: 0
Open: 0
High: 0
Low: 0
Close: 0
Adj Close: 0
Volume: 0
3MA: 0
5MA: 0
3MA_new: 0
5MA_new: 0
```

```
raw_df.isnull().sum()
```

```
Date          0
Open          0
High          0
Low           0
Close         0
Adj Close     0
Volume       115
3MA           2
5MA           4
3MA_new       3
5MA_new       5
dtype: int64
```

```
raw_df = raw_df.dropna()
```

```
raw_df.isnull().sum()
```

```
Date          0
Open          0
High          0
Low           0
Close         0
Adj Close     0
Volume        0
3MA           0
5MA           0
3MA_new       0
5MA_new       0
dtype: int64
```

데이터에 missing data 가 존재하지 않으므로 정규화 단계로 넘어간다.

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler()
```

```
scale_cols = ['Open', 'High', 'Low', 'Close', 'Adj Close', '3MA', '5MA', 'Volume', '3MA_new',
              '5MA_new']
```

```
scaled_df = scaler.fit_transform(raw_df[scale_cols])

scaled_df = pd.DataFrame(scaled_df, columns=scale_cols)

print(scaled_df)
```

Open	High	Low	Close	Adj Close	3MA	5MA	\
0	0.026892	0.023075	0.027216	0.024357	0.018851	0.024531	0.025617
1	0.023473	0.020736	0.023197	0.022431	0.017361	0.023200	0.023138
2	0.022334	0.019779	0.021589	0.018353	0.014204	0.021184	0.021921
3	0.016750	0.014462	0.014699	0.014048	0.010872	0.017723	0.019832
4	0.016522	0.017120	0.016996	0.016200	0.012538	0.015632	0.018180
...	...	...	...	...	...	...	...
5403	0.620556	0.592726	0.624483	0.627280	0.649280	0.613585	0.619190
5404	0.611440	0.578903	0.616445	0.612552	0.634217	0.616628	0.618501
5405	0.620556	0.577839	0.615296	0.603489	0.624947	0.618149	0.615976
5406	0.604603	0.570396	0.609554	0.605755	0.627264	0.610923	0.616435
5407	0.612580	0.576776	0.615296	0.608021	0.629582	0.609402	0.618272

	Volume	3MA_new	5MA_new
0	0.348031	0.025558	0.027355
1	0.646227	0.023847	0.023546
2	0.503494	0.023200	0.023362
3	0.890917	0.021184	0.022146
4	0.612089	0.017723	0.020057
...	...	...	...
5403	0.104386	0.609782	0.617900
5404	0.076116	0.613585	0.619277
5405	0.070877	0.616628	0.618589
5406	0.081790	0.618149	0.616064
5407	0.073628	0.610923	0.616523

[5408 rows x 10 columns]

다음으로 RNN 에 입력하기 위한 데이터의 형태를 만들기 위한 과정이다. 우선 time step 에 따라 feature 와 label data 를 생성하는 함수를 정의한다.

```
def make_sequene_dataset(feature, label, window_size):

    feature_list = []
    label_list = []

    for i in range(len(feature)-window_size):

        feature_list.append(feature[i:i+window_size])
        label_list.append(label[i:i+window_size])

    return np.array(feature_list), np.array(label_list)
```

주가 예측을 위해서 3MA, 5MA, Adj Close 를 feature 로, true value 인 label 은 Close 로 설정한다.

```
feature_cols = ['3MA_new', '5MA_new']
label_cols = ['Close']

feature_df = pd.DataFrame(scaled_df, columns=feature_cols)
label_df = pd.DataFrame(scaled_df, columns=label_cols)
```

생성된 데이터는 데이터 프레임으로 `make_sequene_dataset` 에 입력하기 위해 `numpy` 형태로 만들어준다.

```
feature_np = feature_df.to_numpy()
label_np = label_df.to_numpy()

print(feature_np.shape, label_np.shape)

(5408, 2) (5408, 1)
```

```
feature_np
array([[0.02555813, 0.02735513],
       [0.02384665, 0.02354561],
       [0.02320009, 0.02336202],
       ...,
       [0.616628 , 0.61858864],
       [0.61814932, 0.61606426],
       [0.61092306, 0.61652324]])
```

`time step` 을 40 으로 설정하여 시계열 데이터를 생성한다.

```
window_size = 40

X, Y = make_sequene_dataset(feature_np, label_np, window_size)

print(X.shape, Y.shape)

(5368, 40, 2) (5368, 1)
```

다음으로 `train data` 와 `test data` 의 비율을 95:5 로 나눈다.

```
split = int(len(X)*0.95)

x_train = X[0:split]
y_train = Y[0:split]

x_test = X[split:]
y_test = Y[split:]

print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)

(5099, 40, 2) (5099, 1)
(269, 40, 2) (269, 1)
```

데이터가 완성되었으므로 모델을 정의한다.

```
# LSTM model

model_lstm = Sequential()

model_lstm.add(LSTM(128, activation='tanh', input_shape=x_train[0].shape))

model_lstm.add(Dense(1, activation='linear'))

model_lstm.compile(loss='mse', optimizer='adam', metrics=['mse'])
```

```
model_lstm.summary()
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
lstm_1 (LSTM)	(None, 128)	67072
dense_1 (Dense)	(None, 1)	129

---

Total params: 67,201  
Trainable params: 67,201  
Non-trainable params: 0

---

EarlyStopping 을 적용하여 모델을 학습한다.

```
from tensorflow.keras.callbacks import EarlyStopping
```

```
early_stop = EarlyStopping(monitor='val_loss', patience=10)
```

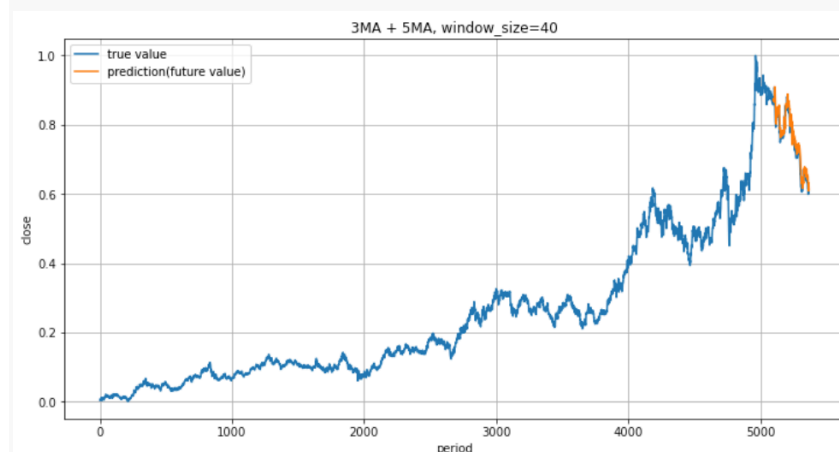
```
model_lstm.fit(x_train, y_train,
               validation_data=(x_test, y_test),
               epochs=100, batch_size=16,
               callbacks=[early_stop])
```

```
pred_lstm = model_lstm.predict(x_test)
```

```
x_split = range(5100, 5369)
```

```
plt.figure(figsize=(12, 6))
plt.title('3MA + 5MA, window_size=40')
plt.ylabel('close')
plt.xlabel('period')
plt.plot(Y, label='true value')
plt.plot(x_split, pred_lstm, label='prediction(future value)')
plt.grid()
plt.legend(loc='best')
```

```
plt.show()
```



LSTM model 을 이용하여 구한 예측값의 시기는 2021 년 6 월 1 일부터이다. 다만, LSTM 입력되는 데이터의 window size 가 40 으로 추정된 데이터는 40 일치 Close 에 대한 값이다.

```
print(r2_score(y_test, pred_lstm))
```

```
0.9180336839485008
```

$R^2$  값이 0.91 로 이는 모델의 모든 예측값과 실제값이 거의 일치한다는 것이다. 모델의 훈련 데이터에서 target 인 Close 의 설정때문인데, Close 를 정의하는 과정에서  $\hat{y}_t$  을  $y_{t-1}$  로 설정하였기 때문이다. 전날의 값을 오늘의 예측값으로 설정하는 것은 correlation 을 매우 높이기 때문에  $R^2$  이 높게 나올 수 밖에 없으므로 유의미한 분석이 아니다. 따라서, correlation 을 피하기 위해서 target 을  $R_t = \frac{y_t - y_{t-1}}{y_{t-1}}$  로 정의해야 한다. 새로운 target 인  $R_t$  를 생성한 후 동일한 과정을 거쳐 모델을 훈련하도록 한다.

```
# 새로운 target column 생성
```

```
df2['R_t'] = np.NaN
```

```
for i in range(len(df2)-1):
    y0 = df2.iloc[i, 4]
    y1 = df2.iloc[i+1, 4]
    df2.iloc[i+1, 11] = (y1 - y0)/y0
```

```
df2.head()
```

	Date	Open	High	Low	Close	Adj Close	Volume	3MA	5MA	3MA_new	5MA_new	R_t
0	2000-09-01	5540.0	5600.0	5420.0	5540.0	4192.145020	49955000	NaN	NaN	NaN	NaN	NaN
1	2000-09-04	5440.0	5550.0	5310.0	5340.0	4040.806396	28625000	NaN	NaN	NaN	NaN	-0.036101
2	2000-09-05	5260.0	5340.0	5030.0	5150.0	3897.030029	79765000	5343.333333	NaN	NaN	NaN	-0.035581
3	2000-09-06	5200.0	5280.0	5090.0	5150.0	3897.030029	53120000	5213.333333	NaN	5343.333333	NaN	0.000000
4	2000-09-07	4940.0	4980.0	4760.0	4790.0	3624.616699	83905000	5030.000000	5194.0	5213.333333	NaN	-0.069903

```
# missing data 처리
```

```
df2['Volume'] = df2['Volume'].replace(0, np.nan)
```

```
df2 = df2.dropna()
```

```
df2.isnull().sum()
```

```
Date      0
Open      0
High      0
Low       0
Close     0
Adj Close 0
Volume    0
3MA       0
5MA       0
3MA_new   0
5MA_new   0
R_t       0
dtype: int64
```

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler()
```

```

scale_cols = ['3MA_new', '5MA_new', 'R_t']

scaled_df2 = scaler.fit_transform(df2[scale_cols])

scaled_df2 = pd.DataFrame(scaled_df2, columns=scale_cols)

print(scaled_df2)

```

```

3MA_new    5MA_new    R_t
0      0.025558  0.027355  0.561470
1      0.023847  0.023546  0.382246
2      0.023200  0.023362  0.243221
3      0.021184  0.022146  0.206714
4      0.017723  0.020057  0.658627
...      ...      ...      ...
5403    0.609782  0.617900  0.648951
5404    0.613585  0.619277  0.423892
5405    0.616628  0.618589  0.451601
5406    0.618149  0.616064  0.510610
5407    0.610923  0.616523  0.510568

```

```
[5408 rows x 3 columns]
```

```

def make_sequene_dataset(feature, label, window_size):

    feature_list = []
    label_list = []

    for i in range(len(feature)-window_size):

        feature_list.append(feature[i:i+window_size])
        label_list.append(label[i+window_size])

    return np.array(feature_list), np.array(label_list)

```

```

feature_cols = ['3MA_new', '5MA_new']
label_cols = ['R_t']

feature_df = pd.DataFrame(scaled_df2, columns=feature_cols)
label_df = pd.DataFrame(df2, columns=label_cols)

```

```

feature_np = feature_df.to_numpy()
label_np = label_df.to_numpy()

print(feature_np.shape, label_np.shape)

```

```
(5408, 2) (5408, 1)
```

```
window_size = 40
```

```
X, Y = make_sequene_dataset(feature_np, label_np, window_size)
```

```
print(X.shape, Y.shape)
```

```
(5368, 40, 2) (5368, 1)
```

```
split = int(len(X)*0.95)
```



```
x_train = X[0:split]
y_train = Y[0:split]
```

```
x_test = X[split:]
y_test = Y[split:]
```

```
print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
```

```
(5099, 40, 2) (5099, 1)
(269, 40, 2) (269, 1)
```

```
# LSTM model
```

```
model_lstm2 = Sequential()
```

```
model_lstm2.add(LSTM(128, activation='tanh', input_shape=x_train[0].shape))
```

```
model_lstm2.add(Dense(1, activation='linear'))
```

```
model_lstm2.compile(loss='mse', optimizer='adam', metrics=['mse'])
```

```
model_lstm2.summary()
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
lstm_2 (LSTM)	(None, 128)	67072
dense_2 (Dense)	(None, 1)	129

```
=====
```

```
Total params: 67,201
```

```
Trainable params: 67,201
```

```
Non-trainable params: 0
```

```
from tensorflow.keras.callbacks import EarlyStopping
```

```
early_stop = EarlyStopping(monitor='val_loss', patience=10)
```

```
model_lstm2.fit(x_train, y_train,
                validation_data=(x_test, y_test),
                epochs=100, batch_size=16,
                callbacks=[early_stop])
```

```
pred_lstm2 = model_lstm2.predict(x_test)
```

```
x_split = range(5100, 5369)
```

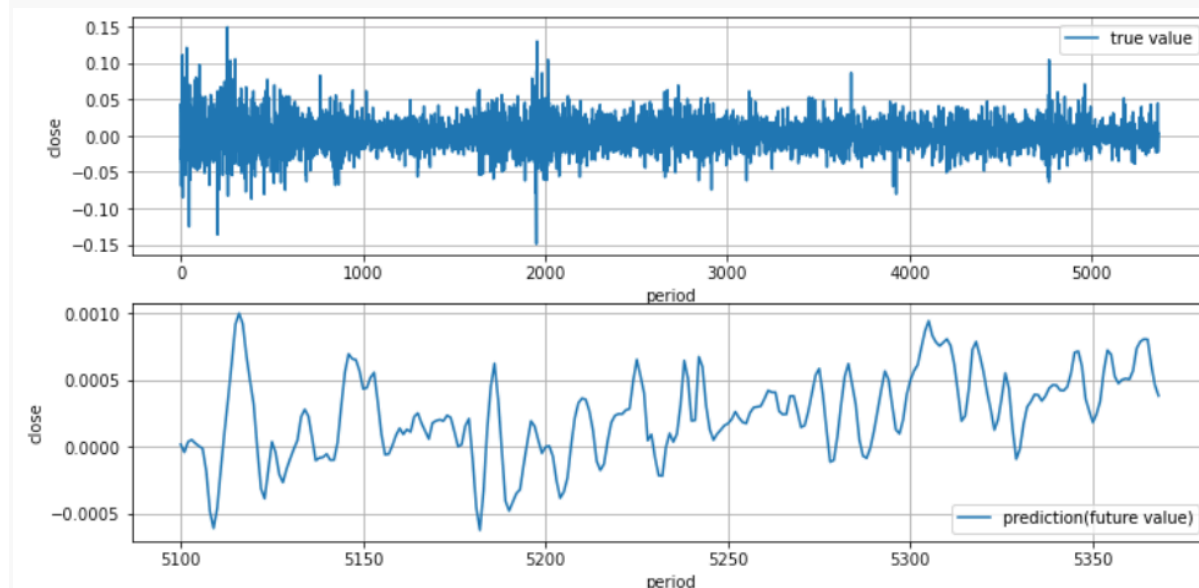
```
plt.figure(figsize=(12, 6))
plt.title('3MA + 5MA, window_size=40')
```

```
plt.subplot(211)
plt.plot(Y, label = 'true value')
plt.ylabel('close')
plt.xlabel('period')
```

```
plt.grid()
plt.legend(loc='best')

plt.subplot(212)
plt.plot(x_split, pred_lstm2, label = 'prediction(future value)')
plt.ylabel('close')
plt.xlabel('period')
plt.grid()
plt.legend(loc='best')

plt.show()
```



```
print(r2_score(y_test, pred_lstm2))
```

```
-0.008012695856493446
```

$R^2$ 값이 음수이다. 주식 데이터의 예측은 매우 어려운 작업으로 모델의 예측력이 매우 떨어진다.

다음으로 GRU 층을 이용한 모델을 정의한다.

```
model_gru = Sequential()

model_gru.add(GRU(256, activation='tanh', input_shape=x_train[0].shape))

model_gru.add(Dense(1, activation='linear'))

model_gru.compile(loss='mse', optimizer='adam', metrics=['mse'])
model_gru.summary()
```

```
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
=====		
gru (GRU)	(None, 256)	199680
dense_3 (Dense)	(None, 1)	257

```
=====
Total params: 199,937
Trainable params: 199,937
Non-trainable params: 0
=====
```

```
from tensorflow.keras.callbacks import EarlyStopping
```

```
early_stop = EarlyStopping(monitor='val_loss', patience=10)
```

```
model_gru.fit(x_train, y_train,
              validation_data=(x_test, y_test),
              epochs=100, batch_size=16,
              callbacks=[early_stop])
```

```
pred_gru = model_gru.predict(x_test)
```

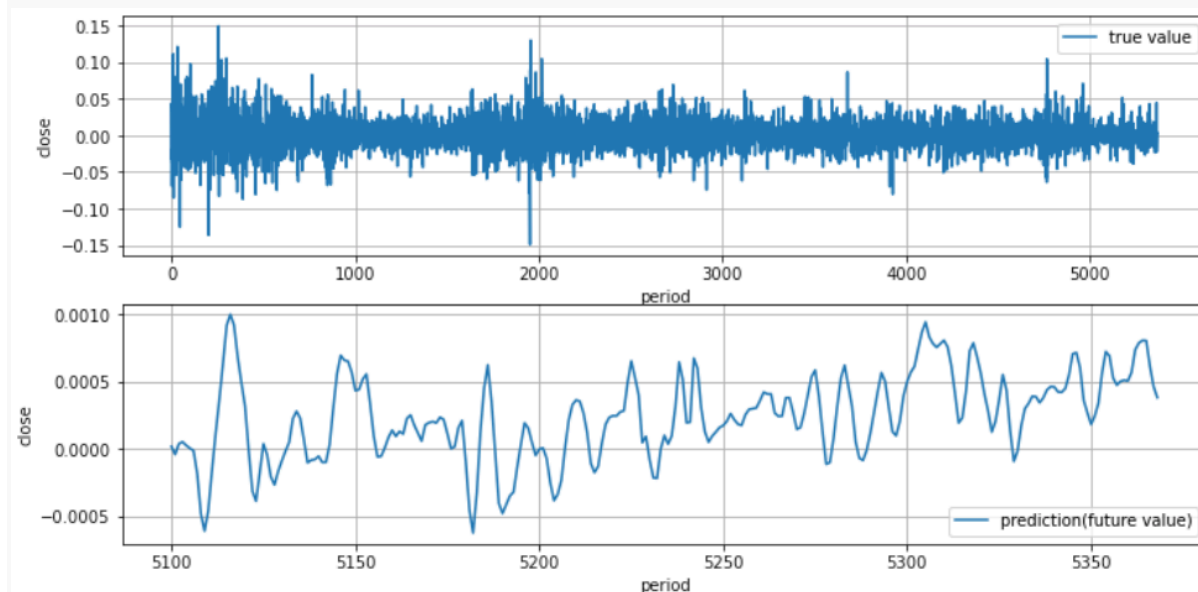
```
x_split = range(5100, 5369)
```

```
plt.figure(figsize=(12, 6))
plt.title('3MA + 5MA, window_size=40')
```

```
plt.subplot(211)
plt.plot(Y, label = 'true value')
plt.ylabel('close')
plt.xlabel('period')
plt.grid()
plt.legend(loc='best')
```

```
plt.subplot(212)
plt.plot(x_split, pred_lstm2, label = 'prediction(future value)')
plt.ylabel('close')
plt.xlabel('period')
plt.grid()
plt.legend(loc='best')
```

```
plt.show()
```



마찬가지로 GRU model 을 이용하여 구한 예측값의 시기는 2021 년 6 월 1 일부터이다. 다만, GRU 에 입력되는 데이터의 window size 가 40 으로 추정된 데이터는 40 일치 Close 에 대한 값이다.

```
print(r2_score(y_test, pred_gru))
```

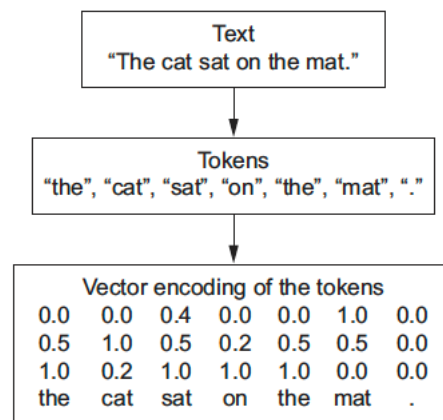
```
-0.09758368300630482
```

jmkim21@ewhain.net

## 4. Text Analysis in RNN

텍스트는 sequence data 에 속하는 데이터 형태 중 하나이다. 텍스트 데이터는 단어나 문장의 sequence 형태로 이해할 수 있는데, 이런 형태의 데이터를 분석하는 과정을 자연어 처리(natural-language processing)이라고 한다. 자연어 처리의 과정의 목적은 CNN 과 비슷하다. CNN 에서는 이미지를 픽셀 단위로 분리한 후 각 픽셀을 RGB 를 이용하여 숫자로 만들고 전처리가 완료된 데이터를 여러 종류의 필터가 지나가며 이미지의 특징을 추출하였다. 마찬가지로, 자연어 처리는 텍스트를 읽어가며 문맥을 분석하는데 이때 텍스트 데이터를 숫자로 변환해야 모델이 데이터를 분석할 수 있다.

텍스트 데이터를 숫자로 구성된 tensor 로 변환하는 과정에서 핵심은 숫자로 바꾸기 위한 단어의 개수나 범위를 어떻게 설정하느냐이다. 이를 토큰(token)이라는 단위로 명명하는데 토큰이란 상황에 따라 다르지만 일반적으로 의미있는 단위로 정의한다.



NOTE)

위의 문장에서 토큰의 단위는 1개의 단어이다.

## 1) Tokenization

단어는 의미를 가지는 가장 작은 단위로 여겨진다. 그러나, 단어 단위로 문장을 분리하게 되는 경우 문장의 문맥이 없어질 수 있으므로 문맥을 보존할 수 있는 단위로 문장을 분리하는 것이 필요하다. 이런 의미를 보존하는 단위를 토큰이라고 한다. 텍스트 데이터를 전처리하는 과정에서 문장을 토큰화하는 경우 토큰의 단위를 어떻게 설정할 것인가는 제일 중요한 문제이다. 토큰의 단위는 단어, 구, 문장 등이 될 수 있다. keras 에서 제공하는 토큰화 함수의 경우 문장에서 구두점을 제거한 후 문장에 존재하는 띄어쓰기를 기준으로 토큰화를 진행한다. 혹은 NLTK(Natural Language Toolkit) 패키지를 이용할 수도 있는데, 이는 파이썬에서 개발한 자연어 처리 패키지이다. 그러나 영어를 기준으로 만들어진 패키지이기 때문에 한국어에는 적용하는 것이 어렵다. 한국어는 영어와는 다르게 조사를 이용하기 때문에 의미를 가지는 최소단위가 형태소이다. 따라서, 한국어 처리를 위한 패키지의 경우 형태소를 기준으로 토큰화를 진행한다.

토큰화의 기본적인 방법 중 하나는 원-핫 인코딩(one-hot encoding)이다. 원-핫 인코딩을 이용하여 텍스트 데이터를 벡터로 만드는 경우 두 가지 과정을 거치게 된다. 하나의 단어를 토큰의 기준으로 잡을 경우 텍스트에 존재하는 각 단어에는 고유한 정수가 인덱스로 지정된다. 다음으로 표현하고 싶은 단어의 인덱스에 1을 부여하고, 다른 단어의 인덱스에는 0을 부여하여 벡터를 만든다. 그 결과 원-핫 인코딩을 거친 텍스트는 0과 1로 구성된 numeric vector가 된다.

아래의 예시는 토큰의 단위를 단어로 설정한 후 원-핫 인코딩을 수행하는 코드이다.

```
import numpy as np

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

token_index = {}
for sample in samples:
    for word in sample.split():
        if word not in token_index:
            token_index[word] = len(token_index) + 1

max_length = 10

results = np.zeros(shape = (len(samples), max_length, max(token_index.values())+1))

for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = token_index.get(word)
        results[i, j, index] = 1
```

results

```
array([[0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
[[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]])
```

다음은 keras 를 이용한 토큰화 과정이다.

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

tokenizer = Tokenizer(num_words = 1000)
tokenizer.fit_on_texts(samples)

sequences = tokenizer.texts_to_sequences(samples)

one_hot_results = tokenizer.texts_to_matrix(samples, mode = 'binary')

word_index = tokenizer.word_index
print(f'Found {len(word_index)} unique tokens.')

Found 9 unique tokens.

one_hot_results

array([[0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.]])
```

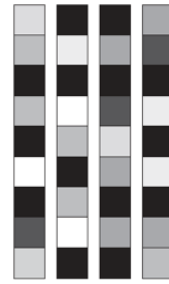
## 2) Word Embedding / Token Embedding

원-핫 인코딩에는 여러 단점들이 있다. 텍스트 데이터에 포함된 단어의 개수가 늘어날수록 벡터의 차원이 늘어나는데 이렇게 생성된 벡터는 대부분의 값이 0 으로 채워지게 된다. 데이터에 0 이 많은 경우 저장 공간 측면에서도 매우 비효율적인 방식이며, 단어의 유사도를 표현하지 못하는 문제점도 발생한다. 자연어 처리의 본연의 목적은 텍스트 데이터의 문맥을 파악하고자 함인데 긴 두개의 문장을 토큰화하는 경우 만들어진 numeric vector 는 두 문장의 의미나 관계를 파악하기가 어렵다. 특히, 단어 간 유사성을 파악하기 어렵다는 단점은 검색이나 추천 시스템에서 문제가 될 수 있다. 예를 들어, '이대 맛집'이라는 단어를 검색한 경우 '이화여대 맛집', '이대 주변 맛집', '이대 음식점' 등 유사 단어에 대한 결과도 보여줄 수 있어야 한다. 하지만, 단어간의 유사성을 계산할 수 없다면 '이화여대', '음식점' 등 연관된 단어의 결과를 함께 보여줄 수 없다. 이런 단점을 해결하기 위해 단어의 의미를 반영하여 텍스트를 토큰화 하는 기법이 '워드 임베딩(word embedding, token embedding)'이다.



One-hot word vectors:

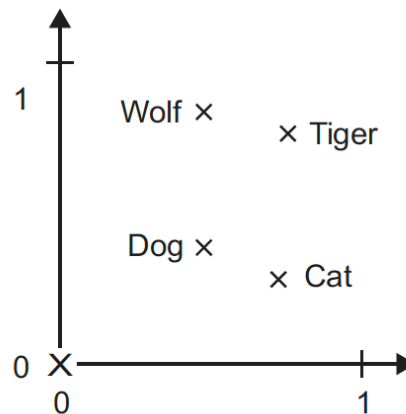
- Sparse
- High-dimensional
- Hardcoded



Word embeddings:

- Dense
- Lower-dimensional
- Learned from data

워드 임베딩은 인간이 사용하는 언어의 의미를 기계가 학습할 수 있도록 하기 위함이다. 예를 들어, 'exact'와 'accurate'는 대부분의 문장에서 비슷한 의미를 나타낸다. 워드 임베딩 과정에서 두 개의 단어가 동시에 등장하는 경우 두 단어의 의미가 비슷하다는 것을 신경망이 알 수 있도록 벡터가 만들어지는 것이다. 수리적으로 비슷하다는 것의 의미를 나타낼 수 있는 방법은 거리(distance)를 이용한 표현인데 변환된 두 단어의 벡터가 임베딩 공간(embedding space)에서 거리가 가까울수록 신경망은 데이터간의 유사성이 높다고 판단하게 된다. 또한, 임베딩 공간에서는 방향(direction)도 중요한 의미를 가진다. 예시를 통해 살펴보도록 한다.



위의 그림에서 'Wolf', 'Tiger', 'Cat', 'Dog' 4 개의 단어가 임베딩 공간에 있다. 이는 각 단어 사이의 의미가 기하학적으로 계산된 결과이다. 예를 들어, 'cat'에서 'tiger'를 연결한 벡터의 방향과 'dog'에서 'wolf'를 연결한 벡터의 방향이 동일하다는 것은 직관적으로 받아들여질 것이다. 만들어진 두 벡터의 의미는 'from pet to wild animal'으로 해석할 수 있다. 비슷한 방법으로 'dog'에서 'cat'을 연결한 벡터와 'wolf'에서 'tiger'를 연결한 벡터의 의미는 'from canine(개과) to feline(고양이과)'으로 해석할 수 있다. 실제 텍스트 데이터를 임베딩하는



과정에서는 일반적으로 'gender'나 'plural' 벡터가 많이 쓰인다. 예를 들어, 'female' 벡터를 'king' 벡터에 더하게 되면 'queen' 벡터를 얻을 수 있다.

RNN 에서 embedding 과정을 추가하기 위해선 keras 에서 제공하는 embedding layer 를 RNN model 에 쌓으면 된다. embedding layer 는 2D tensor (samples, sequence\_length)를 입력받으며 layer 를 통과 후 출력되는 값은 3D tensor (samples, sequence\_length, embedding\_dim) 이다.

### 3) Study Case (IMDB Movie Reviews Sentiment Analysis)

IMDB data 는 영화 리뷰 데이터로 리뷰가 긍정적인 경우 1, 부정인 경우 0 으로 표시한 label 로 구성된 데이터이다. 스탠포드 대학교에서 2011 년에 낸 논문에서 이 데이터를 소개하였으며, 논문에서는 이 데이터를 훈련 데이터와 테스트 데이터를 50:50 비율로 분할하여 88.89%의 정확도를 얻었다고 소개하고 있다. keras 에서 제공하는 IMDB data 의 경우 토큰화가 완료된 데이터이기 때문에 raw data 를 사용하도록 한다.

우선 데이터를 다운로드 후 아래의 코드를 이용하여 데이터를 feature 와 label(positive, negative)로 나눈다.

NOTE) DATA : <http://mng.bz/0tlo>

```
# Processing the labels of the raw IMDB data

import os

imdb_dir = '/Users/jiminpopo/Documents/lab/수업자료/ipynb/CHAPTER 5/IMDB'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

다음으로 텍스트 데이터에 대하여 토큰화를 실행한다. 우선, pretrained embedding 을 사용하지 않은 모델을 정의하도록 한다.

```
# Tokenizing the text of the raw IMDB data

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

maxlen = 100
training_samples = 20000
validation_samples = 5000
max_words = 10000

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print(f'Found {len(word_index)} unique tokens.')

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

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

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

Found 88582 unique tokens.
Shape of data tensor: (25000, 100)
Shape of label tensor: (25000,)
```

토큰화를 진행한 후 train data 와 validation data 로 분리하였다. train data 의 개수는 20000 개이며, validation data 의 개수는 5000 개이다. 모델의 성능을 평가하기 위한 test data 에 대해선 뒤에서 동일한 과정으로 토큰화를 진행하도록 한다.

```
# Training the same model without pretrained word embeddings

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

model = Sequential()
model.add(Embedding(max_words, 100, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs=20, batch_size=32, validation_data=(x_val, y_val))
```

Model: "sequential\_8"

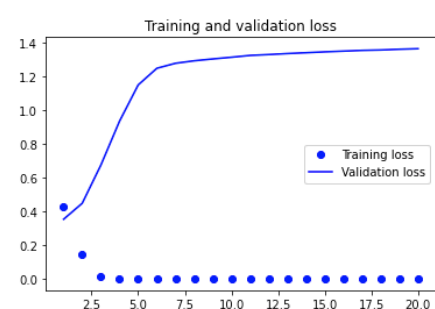
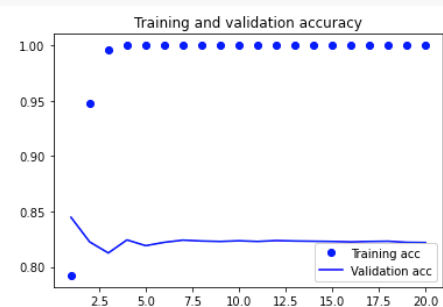
Layer (type)	Output Shape	Param #
embedding_5 (Embedding)	(None, 100, 100)	1000000
flatten_5 (Flatten)	(None, 10000)	0
dense_13 (Dense)	(None, 32)	320032
dense_14 (Dense)	(None, 1)	33

=====  
 Total params: 1,320,065  
 Trainable params: 1,320,065  
 Non-trainable params: 0  
 =====

# Plotting the results

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```



validation data 에 대해서 0.8 정도의 accuracy 를 보이고 있다. test data 를 이용하여 model 의 performance 를 판단하도록 한다.

```
# Tokenizing the data of the test set

test_dir = os.path.join(imdb_dir, 'test')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(test_dir, label_type)
    for fname in sorted(os.listdir(dir_name)):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)

print(len(x_test), len(y_test))
```

```
25000 25000
```

Test data 의 개수는 25000 개이다.

```
# Evaluating the model on the test set

model.evaluate(x_test, y_test)

[1.3728729486465454, 0.8187599778175354]
```

test data 를 이용한 accuracy 에서도 validation data 와 비슷한 성능을 보이고 있다.

다음으로 pretrained word embedding 을 이용해보도록 한다. RNN 에서 pretrained 를 이용하는 것은 이미지 분류에서 pretrained convnets 을 이용하는 것과 동일하다. 여기서 이용할 방법은 GloVe(global vectors)로 2014 년 스탠포드 대학교에서 개발되었다. 이 방법은 통계적인 방법을 이용하여 Co-occurrence Matrix 를 생성한다. Co-occurrence Matrix 의 생성과정에 대해 잠시 살펴해보도록 한다.

Co-occurrence Matrix 란 행과 열을 전체 단어 집합의 단어들로 구성하고,  $i$  단어의 윈도우 크기(window size) 내에서  $k$  단어가 등장한 횟수를  $(i,k)$ 에 기재한 행렬이다. 예를 들어, 다음과 같이 3문장으로 구성된 코퍼스가 있다고 가정하자.

NOTE)

Window size란 노드에서 1번의 time step 당 읽어들이는 데이터의 길이를 의미한다. Window size가 20인 경우 20개의 숫자가 1번의 time step에서 분석된다.

NOTE) 코퍼스란 분석의 대상이되는 텍스트 데이터 전체를 지칭하는 것으로 받아들인다.

- I like deep learning.
- I like NLP.
- I enjoy flying.

위의 코퍼스를 이용하여 Co-occurrence Matrix 를 만들면 다음과 같다.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

그러나 위의 행렬을 그대로 이용하게 될 경우 코퍼스의 단위가 커지게되면 sparse matrix 가 되어 원-핫 인코딩에서의 단점과 비슷한 문제가 발생할 수 있다. 이를 해결하기 위해서 GloVe 에서는 Co-occurrence Probability 을 이용한다. GloVe 의 과정에서 이용하는 함수를 전체적으로 이해하는 것은 어려움이 있으므로 이 함수에 들어가는 구성요소 중 가장 중요한 핵심 아이디어가 Co-occurrence Probability 라는 것만 알아두자.

Co-occurrence probability  $P(k|i)$ 란 Co-occurrence Matrix 로부터 특정 단어  $i$ 의 전체 등장 횟수를 카운트하고, 특정 단어  $i$ 가 등장했을 때 어떤 단어  $k$ 가 등장한 횟수를 카운트하여 계산한 조건부 확률이다.  $P(k|i)$ 에서  $i$ 를 중심 단어(center word),  $k$ 를 주변 단어(context word)라고 했을 때, Co-occurrence Matrix 에서 중심 단어  $i$ 의 행의 모든 값을 더한 값을 분모로 하고  $(i, k)$ 의 값을 분자로 한 값이다. 다음은 GloVe 논문에서 제안한 Co-occurrence probability 의 하나의 예이다.

	$x = \text{solid}$	$x = \text{gas}$	$x = \text{water}$	$x = \text{fashion}$
$P(x \text{ice})$	$1.9 \times 10^{-4}$	$6.6 \times 10^{-5}$	$3.0 \times 10^{-3}$	$1.7 \times 10^{-5}$
$P(x \text{steam})$	$2.2 \times 10^{-5}$	$7.8 \times 10^{-4}$	$2.2 \times 10^{-3}$	$1.8 \times 10^{-5}$
$\frac{P(x \text{ice})}{P(x \text{steam})}$	8.9	$8.5 \times 10^{-2}$	1.36	0.96

위의 표를 통해 ice가 등장했을 때 solid가 등장할 확률은 steam이 등장했을 때 solid가 등장할 확률보다 약 8.9 배 크다는 사실을 알 수 있다. 이 숫자가 의미하는 바는 solid 는 steam 보다 ice 와 더 자주 등장한다는 것이다. GloVe 의 아이디어를 요약하자면, 임베딩 된 중심 단어와 주변 단어 벡터의 내적이 전체 코퍼스에서 Co-occurrence probability 가 되도록 만드는 것이다.

$$w_i \cdot w_j = \log P(i|j)$$

$$J = \sum_{i,j=1}^V f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2$$

다시 코드로 돌아가 GloVe 를 다운받은 후 word embedding 을 진행한다.

NOTE)

Go to <https://nlp.stanford.edu/projects/glove> and download glove.6B.zip

```
# Parsing the GloVe word-embeddings file
```

```
glove_dir = './CHAPTER 5/glove'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs

f.close()

print(F'Found {len(embeddings_index)} word vectors.')
```

```
Found 400000 word vectors.
```

```
# Preparing the GloVe word-embeddings matrix
```

```
embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
```

```
# Model definition
```

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

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

```
Model: "sequential_10"
```

Layer (type)	Output Shape	Param #
=====		
embedding_7 (Embedding)	(None, 100, 100)	1000000
flatten_7 (Flatten)	(None, 10000)	0
dense_17 (Dense)	(None, 32)	320032
dense_18 (Dense)	(None, 1)	33
=====		

```
Total params: 1,320,065
Trainable params: 1,320,065
Non-trainable params: 0
```

---

```
# Loading pretrained word embeddings into the Embedding Layer
```

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

위의 코드는 model 에서 embedding layer 의 역할을 GloVe 의 embedding matrix 가 대신하도록 설정해준다.

```
# Training and evaluation
```

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs=30, batch_size=32, validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')
```

```
# Plotting the results
```

```
import matplotlib.pyplot as plt

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

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

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

plt.figure()

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

plt.show()
```



```
# Evaluating the model on the test set
```

```
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

```
[2.5379302501678467, 0.6824399828910828]
```

마지막으로 classification 문제이므로 confusion matrix 를 이용하여 오분류율을 계산한다. 아래는 CHAPTER 2 에서 경계확률에 따라 최소의 오분류율을 갖는 confusion matrix 를 구하는 코드이다.

```
# range()는 float 형이 들어갈 수 없으므로 float 을 삽입할 수 있는 range 함수를 정의
```

```
def range_with_floats(start, stop, step):
    while stop > start:
        yield start
        start += step
```

```
# class 1 에 속할 확률,[:,0]인 경우 class 0 에 속할 확률
prob = model.predict(x_test)
```

```
# 값들을 저장할 List 와 dictionary
```

```
mylist = []
res = {}
```

```
n = len(x_test)
```

```
# 최소의 오분류율을 만드는 경계확률을 찾기 위한 반복문
```

```
for i in range_with_floats(0.4, 0.801, 0.001):
    for j in range(0, n):

        # 경계확률보다 크면 class 1 로 분류, 아니면 0 으로 분류
        if prob[j] >= i:
            mylist.append(1)
        else:
            mylist.append(0)
```



```
# 오분류를 구하기 위해 필요한 confusion matrix
mymtx = confusion_matrix(mylist, y_test)

# 오분류율 계산
missrate = (mymtx[0,1] + mymtx[1,0])/(sum(sum(mymtx)))

# dictionary 에 경계확률을 key 로 오분류율을 value 로 저장
res[round(i,3)] = round(missrate,4)

# list 초기화
mylist = []
```

```
label = ["0(negative)", "1(positive)"]
```

```
import pandas as pd
table = pd.DataFrame(mymtx, columns = label, index = label)
table
```

	0(negative)	1(positive)
0(negative)	8397	3768
1(positive)	4103	8732

```
print(f'최소의 오분류율은 {min(res.values())} 이며 이때의 경계확률은 {min(res, key = res.get)} 이다.')
```

최소의 오분류율은 0.3143 이며 이때의 경계확률은 0.769 이다.