CHATGPT에 대한 tweets sentiment를 분류해내는 딥러닝 model들의 성능 비교

2022711778 윤태호

CONTENTS

1. 서론

2023년 3월 14일 OpenAI의 최신 언어 모델인 GPT-4가 출시되었다. chatgpt(챗GPT)는 GPT3.5와 GPT-4를 기반으로 하는 대화형 인공지능 서비스로 프로포트라는 입력하는 곳에 대화를 입력하면 AI가 이에 대한 답변을 생성해낸다. 이에 따라 twitter에는 chatgpt에 대한 다양한 의견이 현재까지도 계속 생겨나고 있다. 이번 분석에 사용되는 데이터는 한달 동안에 chatgpt에 대한 tweets들을 모아놓은 데이터이다. 데이터의 개수는 총 219294개로 한달 동안의 chatgpt에 대한 tweets들이다. 이 데이터에 대한 종속변수로는 sentiment로 chatgptd에 대한 tweets 안에 있는 의견들의 sentiment가 positive 인지, negative 인지 neural 인지 나타낸다. 그래서 본 분석에서 이 tweets들인 text 데이터들을 수치형 자료로 변환하고 tweets 안에 들어 있는 주소라던지 분석에 용이하지 않은 특수 문자들을 제거하는 전처리 과정을 거쳐서 tweets들의 sentiment가 3가지 어떤 범주인지 판별해 내는 모델을 만드는 것을 목표로 한다.
분석에 사용될 모델의 종류로는 주로 텍스트 데이터나 시계열 데이터에 사용되는 RNN,LSTM,GRU이다. 이 세 모델을 tweets의 training data에 fitting한 다음에 test data의 성능을 측정해 볼 것이다. 그 후 세 가지 모델 중 가장 성능이 좋은 모델이 어떤 것인지 확인할 것이다.

https://www.kaggle.com/datasets/charunisa/chatgpt-sentiment-analysis/code?datasetId=2896084 (데이터 출처)

2. 데이터 전처리

```
df = pd.read_csv('/content/drive/MyDrive/file.csv')
df=df.head(12000)
df = df[['tweets', 'labels']]

# Remove all the tweet links since they all begin with https:
df['tweet_list'] = df['tweets'].str.split('https:')
# Select the text part of the list
text = [i[0] for i in df.tweet_list]
df['text'] = text
df = df[['text', 'labels']]
```

데이터를 pd.read_csv함수를 통해 불러오고 모델을 fitting 하는데 21만개의 데이터는 너무 오래 걸리기 때문에 12000개의 데이터만 따로 추출한다. 그리고 데이터의 tweets을 살펴보면 tweets가 나온뒤에 https:라는 주소가 나온다. 이 주소를 빼버리기 위하여 str.split 함수를 사용하여 https:를 기점으로 test를 분리하고 https:이후 부분을 빼버린 부분만 text에 따로 다시 저장한다.

```
# Import re for string processing
import re

# Remove all non-alphanumeric characters from the text list
string = r'[A-Za-z0-9 ]'
trim_list=[]
for row in text:
    s=''
    for letter in row:
        if bool(re.match(string, letter)):
            s+=letter
    trim_list.append(s)

# Remove the non-printing characters from text
rep_list = ['\U0001fae1', '\\n', '@', '#', '\xa0', '***']
for i in trim_list:
    for j in rep_list:
        if j in i:
            i.replace(j,'')

# Map the labels to integers
# 1 for good tweet
# 0 for neutral tweet
# -1 for bad tweet

df['lab_int'] = np.where(df['labels']=='good', 1, np.where(df['labels']=='bad', -1, 0))
df.lab_int
```

위 코드를 살펴보면 모든 문자열들을 추출하고 이 중에 @이나 #과 같은 특수문자들은 제거한다. 그 다음 종속변수인 sentiment 중에 good tweet는 1, neutral tweet는 0, bat tweet는 -1로 전처리 한다.

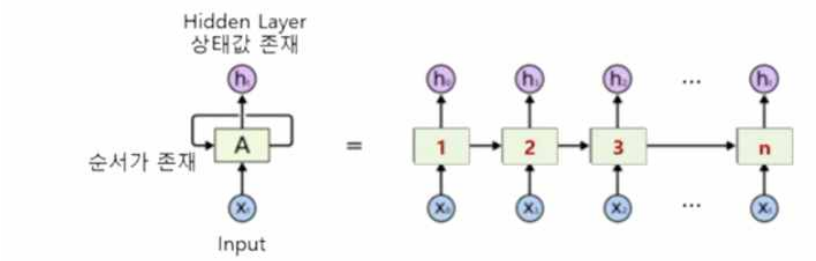
```
from tensorflow.keras.preprocessing.text import Tokenizer
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from sklearn.metrics import classification_report




# 텍스트 데이터를 정수로 인코딩
tokenizer = Tokenizer()
tokenizer.fit_on_texts(df.text)
encoded_texts = tokenizer.texts_to_sequences(df.text)
# 시퀀스 패딩
max_length = max([len(text.split()) for text in df.text])  # 시퀀스의 최대 길이
```

```
padded_texts =pad_sequences(encoded_texts, maxlen=max_length, padding='post')

# 삼진 분류를 위해 레이블을 One-hot 인코딩
num_classes = len(set(df.lab_int))
one_hot_labels = tf.keras.utils.to_categorical(df.lab_int, num_classes)

# 데이터를 train과 test로 나누기
split_ratio = 0.8  # train 데이터의 비율
split_index = int(len(padded_texts) * split_ratio)
train_texts = padded_texts[:split_index]
train_labels = one_hot_labels[:split_index]
test_texts = padded_texts[split_index:]
test_labels = one_hot_labels[split_index:]
```

그 다음으로 tokenizerlfit_on_texts라는 함수를 사용해서 현재 텍스트 데이터를 기준으로 사전을 만든다. 그 후 text data를 tokenizer.texts_to_sequences 함수를 이용하여 정수형으로 인코딩한다. 그 후 시퀀스의 최대 길이를 구한 다음에 모든 인코딩된 텍스트들이 똑같은 길이를 갖추게 하기 위해 pad_sequences 함수를 사용하여 시퀀스의 길이를 모두 똑같이 맞춘다. 그 다음으로 1(긍정),0(중립),-1(부정)으로 처리된 종속변수를 tf.keras.utils.to_categorical 함수를 사용해서 One-hot 인코딩 처리를 한다.
그 후 데이터를 8:2 비율로 train data와 test data로 나눈다. 이제 이 train data로 모델을 fitting할 준비가 완료되었다.

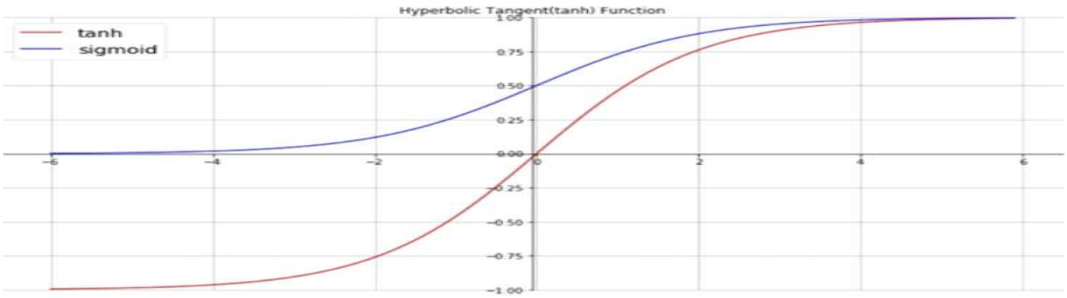
3. Model Fitting(RNN,LSTM,GRU)

-RNN(Recurrent neural network)



RNN의 구조는 위와 같다. RNN은 순환 신경망이라 부르며 데이터가 순환되는 한 단계를 time step이라 부르고 순환층을 cell이라고 부른다. 이러한 cell 출력의 결과는 hidden state 라고 한다.
만약에 새로운 input인 x값이 입력되면 cell을 통과해 hidden state가 된다. 이러한 hidden state는 다른 시점이나 다음 단어인 x와 결합되어 새로운 hidden state를 만들어 낸다. 이런 식의 과정이 반복되고 hidden state는 다음의 x값을 위해 계속 재사용 된다. 이처럼 hidden state 가 계속해서 순환하는 신경망을 순환 신경망이라 부른다.

이 순환 신경망에 주로 쓰이는 activation function은 하이퍼볼릭 탄젠트(tanh)이다.



위 사진과 같이 하이퍼볼릭 탄젠트 함수의 y값의 범위는 -1에서 1이다. 이 함수의 특징은 시그모이드 함수보다 학습 효율성이 뛰어나고 기울시 소실 문제가 적은 장점을 가지고 있다.

```
# RNN 모델 구축
model = Sequential()
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=100, input_length=max_length))
model.add(SimpleRNN(64))
model.add(Dense(num_classes, activation='softmax'))

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

기본적인 RNN모형을 위와 같이 구축한다.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 60, 100)           2058600

simple_rnn_1 (SimpleRNN)     (None, 64)                10560

dense_1 (Dense)              (None, 3)                 195

=================================================================
Total params: 2,069,355
Trainable params: 2,069,355
Non-trainable params: 0
_________________________________________________________________
```

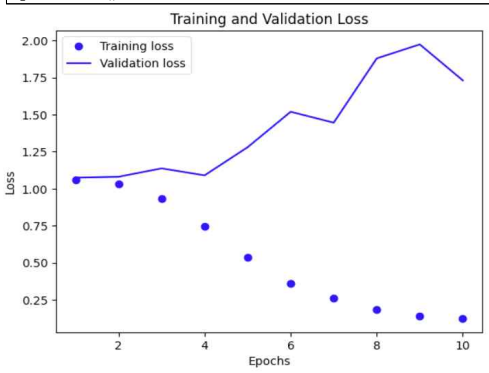
모델의 구조는 위와 같다.

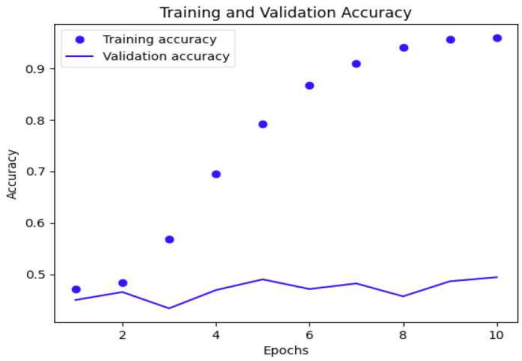
```
# 모델 훈련
history = model.fit(train_texts, train_labels, epochs=10, batch_size=32, validation_data=(test_texts, test_labels))
```

이제 epochs는 10, batch_size는 32로하여 train text data로 RNN 모형을 fitting한다.

```
Epoch 1/10
300/300 [==============================] - 18s 54ms/step - loss: 1.0621 - accuracy: 0.4710 - val_loss: 1.0741 - val_accuracy: 0.4500
Epoch 2/10
300/300 [==============================] - 14s 48ms/step - loss: 1.0336 - accuracy: 0.4835 - val_loss: 1.0802 - val_accuracy: 0.4654
Epoch 3/10
300/300 [==============================] - 17s 56ms/step - loss: 0.9334 - accuracy: 0.5686 - val_loss: 1.1369 - val_accuracy: 0.4338
Epoch 4/10
300/300 [==============================] - 14s 48ms/step - loss: 0.7433 - accuracy: 0.6953 - val_loss: 1.0899 - val_accuracy: 0.4692
Epoch 5/10
300/300 [==============================] - 15s 48ms/step - loss: 0.5365 - accuracy: 0.7919 - val_loss: 1.2796 - val_accuracy: 0.4900
Epoch 6/10
300/300 [==============================] - 15s 50ms/step - loss: 0.3582 - accuracy: 0.8664 - val_loss: 1.5188 - val_accuracy: 0.4712
Epoch 7/10
300/300 [==============================] - 14s 48ms/step - loss: 0.2629 - accuracy: 0.9094 - val_loss: 1.4455 - val_accuracy: 0.4821
Epoch 8/10
300/300 [==============================] - 14s 48ms/step - loss: 0.1823 - accuracy: 0.9400 - val_loss: 1.8789 - val_accuracy: 0.4571
Epoch 9/10
300/300 [==============================] - 14s 47ms/step - loss: 0.1392 - accuracy: 0.9567 - val_loss: 1.9735 - val_accuracy: 0.4863
Epoch 10/10
300/300 [==============================] - 14s 48ms/step - loss: 0.1243 - accuracy: 0.9595 - val_loss: 1.7305 - val_accuracy: 0.4942
```

```
import matplotlib.pyplot as plt
train_loss = history.history['loss']
val_loss = history.history['val_loss']
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
epochs = range(1, len(train_loss) + 1)
plt.plot(epochs, train_loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.plot(epochs, train_accuracy, 'bo', label='Training accuracy')
plt.plot(epochs, val_accuracy, 'b', label='Validation accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

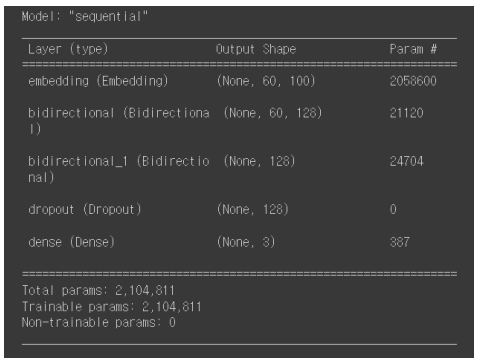




위의 두 그래프를 보면 loss 값은 train data는 epoch가 커질수록 점점 작아지지만 validation data는 점점 커지는 것을 확인할 수 있다. 또한 accuracy 데이터를 보면 training data는 accuracy 점점 높아지지만 validation의 데이터는 그닥 높아지지 않는 부분을 확인할 수 있다.

이에 따라서 RNN의 성능을 높이기 위해 rnn 모델의 구조를 변경하여 다음과 같이 입력하였다.

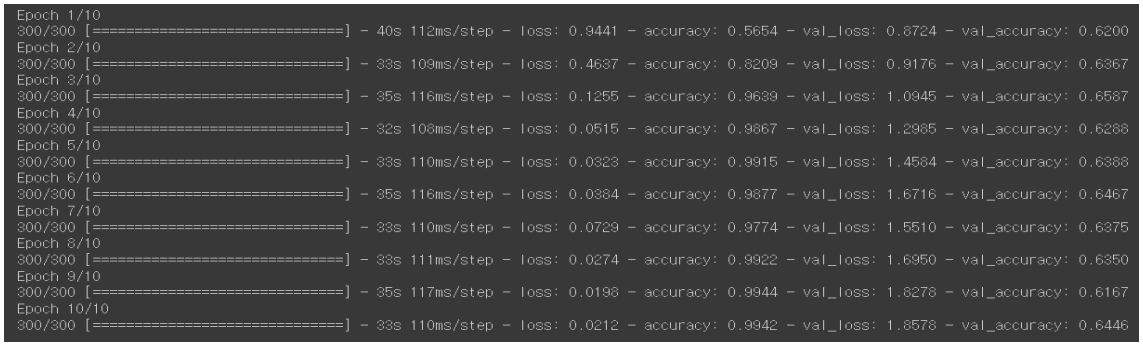
```
# RNN 모델 구축
model = Sequential()
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=100, input_length=max_length))
model.add(Bidirectional(SimpleRNN(64, return_sequences=True)))
model.add(Bidirectional(SimpleRNN(64)))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))

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



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 60, 100)           2058600

 bidirectional (Bidirectiona  (None, 60, 128)          21120
 l)

 bidirectional_1 (Bidirectio  (None, 128)              24704
 nal)

 dropout (Dropout)           (None, 128)               0

 dense (Dense)               (None, 3)                 387

=================================================================
Total params: 2,104,811
Trainable params: 2,104,811
Non-trainable params: 0
_________________________________________________________________
```

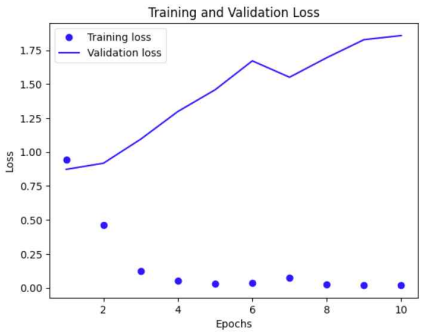
모델 표현력을 향상시키기 위해 추가적인 RNN 층을 쌓을 수 있다. 그래서 다음과 같이 model.add 함수를 통해 RNN을 추가하였다. 또한 단순히 그냥 추가하는 것이 아니라 return_sequences=True 옵션과 Bidirectional 함수를 넣었다. return_sequences=True 옵션은 각 time step의 출력을 반환하여 다음 층으로 전달하는 역할을 한다. 또한 Bidirectional 함수는 입력된 시퀀스를 순방향과 역방향으로 처리하여 정보를 더 잘 캡처하는데 도움을 준다. 그래서 위와 같이 Bidirectional 항목을 2번 넣어 모델의 성능을 높이자고 하였다. 마지막으로 추가적으로 Drop out이라는 Regularization을 적용하였다. 이는 과적합을 완화하기 위해 모델에 랜덤하게 뉴런을 비활성하여 일반화 성능을 향상시키는 방법이다. 그래서 model.add(Dropout(rate=0.2))와 같이 20% 비율의 dropout을 추가하였다.

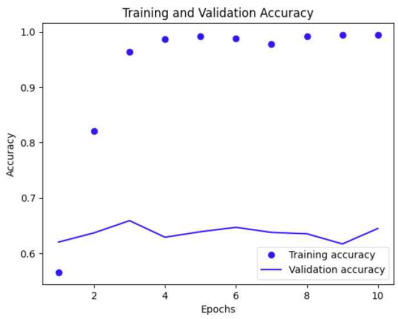
```
# 모델 훈련
history = model.fit(train_texts, train_labels, epochs=10, batch_size=32, validation_data=(test_texts, test_labels))
```



```
Epoch 1/10
300/300 [==============================] - 40s 112ms/step - loss: 0.9441 - accuracy: 0.5654 - val_loss: 0.8724 - val_accuracy: 0.6200
Epoch 2/10
300/300 [==============================] - 33s 109ms/step - loss: 0.4637 - accuracy: 0.8209 - val_loss: 0.9176 - val_accuracy: 0.6367
Epoch 3/10
300/300 [==============================] - 35s 116ms/step - loss: 0.1255 - accuracy: 0.9639 - val_loss: 1.0945 - val_accuracy: 0.6587
Epoch 4/10
300/300 [==============================] - 32s 108ms/step - loss: 0.0515 - accuracy: 0.9867 - val_loss: 1.2985 - val_accuracy: 0.6288
Epoch 5/10
300/300 [==============================] - 33s 110ms/step - loss: 0.0323 - accuracy: 0.9915 - val_loss: 1.4584 - val_accuracy: 0.6388
Epoch 6/10
300/300 [==============================] - 35s 116ms/step - loss: 0.0384 - accuracy: 0.9877 - val_loss: 1.6716 - val_accuracy: 0.6467
Epoch 7/10
300/300 [==============================] - 33s 110ms/step - loss: 0.0729 - accuracy: 0.9774 - val_loss: 1.5510 - val_accuracy: 0.6375
Epoch 8/10
300/300 [==============================] - 33s 111ms/step - loss: 0.0274 - accuracy: 0.9922 - val_loss: 1.6950 - val_accuracy: 0.6350
Epoch 9/10
300/300 [==============================] - 35s 117ms/step - loss: 0.0198 - accuracy: 0.9944 - val_loss: 1.8278 - val_accuracy: 0.6167
Epoch 10/10
300/300 [==============================] - 33s 110ms/step - loss: 0.0212 - accuracy: 0.9942 - val_loss: 1.8578 - val_accuracy: 0.6446
```

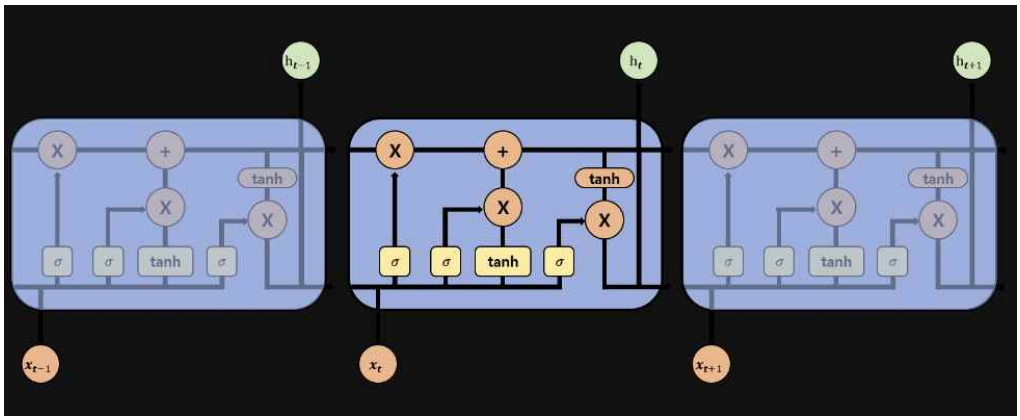
확실히 여러 option을 추가한 RNN모형의 test data의 accuracy를 살펴보았을 때 epoch 10 기준으로 64.46%로 향상된 것을 확인할 수 있다.
그래프를 보기 위해 코드를 RNN의 plot을 보기 위해 똑같이 입력해서 확인한 결과는 아래와 같다.





확실히 test data의 epoch별 loss와 accuracy는 train data에 비해 낮지만 일반적인 RNN에 비해 성능은 좋아진 것을 확인할 수 있다.

-LSTM-
LSTM은 RNN이 오래된 것을 기억하지 못하는, 즉 기울기 소실 문제를 해결하기 위해 탄생했다. LSTM 알고리즘은 기억할 것과 잊을 것을 선택해 중요한 정보를 기억하는 것이 핵심이다. LSTM도 RNN과 동일하게 입력과 가중치를 곱하고 절편을 더해 activation function을 통과시키는 층을 여러개 갖고 있다. RNN과의 차이점은 순환되는 층이 2개라는 점이다. hidden state와 cell state에서 과거 정보가 순환한다. 여기서 cell state는 hidden state와 달리 다음 층으로 전달되지 않는다. 또한 LSTM은 gate를 통해서 정보를 통제한다. 이 gate를 통해 원하는 정보만 통과시킨다.
gate는 input gate forget gate와 output gate 총 3가지가 있다. 우선 input gate는 현재 정보를 얼마나 기억할지 결정하는 gate이다. 이전 층의 hidden state에는 tahn를 곱하고 새로운 정보에는 sigmoid activation function을 곱한다. 이때 sigmoid를 곱하기 때문에 0~1의 값이 나오므로 이 함수를 통해 새로운 정보를 얼마나 사용할지 결정한다.
다음 gate는 forget gate로 정보를 얼마나 잊어버릴지 결정하는 gate이다. 여기에서는 hidden state를 서로 다른 가중치를 곱한 다음 sigmoid 활성 함수를 통과시킨다. 0~1 사이의 결과값을 얻기 때문에 얼마나 정보를 잊을 것인지 결정할 수 있다. 이전 타임 스텝의 cell state와 곱하여 새로운 cell state를 만들어 낸다.
마지막으로 output gate는 다음 층으로 전달한 hidden state를 만드는 gate이다. 이전 hidden state값과 입력 값에 각각 다른 가중치를 곱하고 sigmoid 활성 함수를 통과시킨다. 그 출력 값과 현재 Cell state 값을 tanh 활성 함수에 통과시킨 값을 서로 곱해서 hidden state를 만든다.




```
from tensorflow.keras.layers import LSTM
# LSTM 모델 구축
model = Sequential()
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=100, input_length=max_length))
model.add(Bidirectional(LSTM(128, return_sequences=True)))
model.add(LSTM(128))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

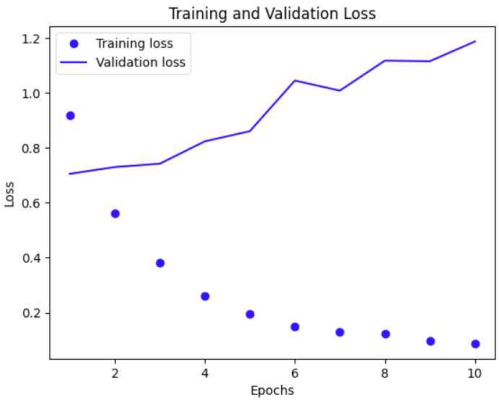
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 60, 100)           2058600

bidirectional_2 (Bidirectio  (None, 60, 256)           234496
nal)

lstm_1 (LSTM)                (None, 128)               197120

dropout_1 (Dropout)          (None, 128)               0

dense_1 (Dense)              (None, 3)                 387

=================================================================
Total params: 2,490,603
Trainable params: 2,490,603
Non-trainable params: 0
_________________________________________________________________
```

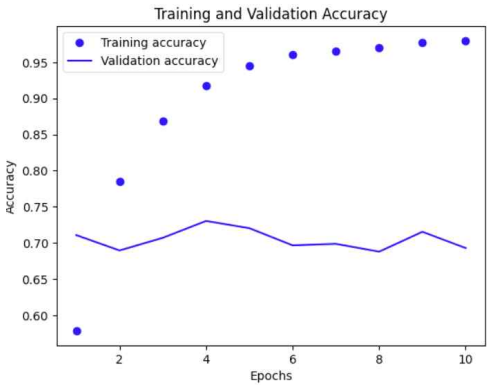
코드를 통해 구축된 모델은 위와 같다.
Bidirectional(LSTM(128, return_sequences=True))을 통해 양방향 LSTM을 사용하고, LSTM 층의 유닛 수를 128로 설정하며, 시퀀스 반환 옵션을 적용하여 다음 LSTM 층에 시퀀스 정보를 전달하도록 설정하였다. 또한 LSTM(128)을 추가하여 하나의 단방향 LSTM 층을 더 쌓았고 이 층은 마지막 시점의 출력만 반환하도록 하였다. 마지막으로 Dropout(0.2)를 추가하여 0.2의 드롭아웃 비율을 적용하여 과적합을 방지하도록 하였다.

```
# 모델 훈련
history = model.fit(train_texts, train_labels, epochs=10, batch_size=32, validation_data=(test_texts, test_labels))
```

```
Epoch 1/10
300/300 [==============================] - 128s 403ms/step - loss: 0.9177 - accuracy: 0.5785 - val_loss: 0.7052 - val_accuracy: 0.7108
Epoch 2/10
300/300 [==============================] - 119s 396ms/step - loss: 0.5621 - accuracy: 0.7845 - val_loss: 0.7298 - val_accuracy: 0.6896
Epoch 3/10
300/300 [==============================] - 117s 390ms/step - loss: 0.3805 - accuracy: 0.8682 - val_loss: 0.7421 - val_accuracy: 0.7071
Epoch 4/10
300/300 [==============================] - 121s 403ms/step - loss: 0.2593 - accuracy: 0.9176 - val_loss: 0.8234 - val_accuracy: 0.7304
Epoch 5/10
300/300 [==============================] - 119s 395ms/step - loss: 0.1965 - accuracy: 0.9449 - val_loss: 0.8605 - val_accuracy: 0.7204
Epoch 6/10
300/300 [==============================] - 125s 417ms/step - loss: 0.1485 - accuracy: 0.9602 - val_loss: 1.0448 - val_accuracy: 0.6967
Epoch 7/10
300/300 [==============================] - 128s 428ms/step - loss: 0.1284 - accuracy: 0.9659 - val_loss: 1.0081 - val_accuracy: 0.6988
Epoch 8/10
300/300 [==============================] - 121s 401ms/step - loss: 0.1230 - accuracy: 0.9697 - val_loss: 1.1173 - val_accuracy: 0.6879
Epoch 9/10
300/300 [==============================] - 123s 410ms/step - loss: 0.0982 - accuracy: 0.9769 - val_loss: 1.1152 - val_accuracy: 0.7154
Epoch 10/10
300/300 [==============================] - 118s 394ms/step - loss: 0.0874 - accuracy: 0.9798 - val_loss: 1.1871 - val_accuracy: 0.6929
```

model fitting 결과 epoch10 기준으로 test data의 accuracy는 69,29% 가 나온 것을 확인할 수 있다.



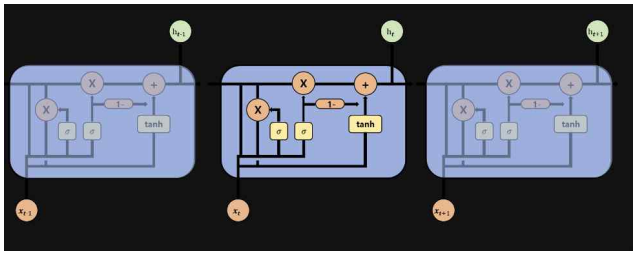


plot의 결과를 살펴보면 test data의 epoch별 loss는 train data의 loss와는 달리 점점 커지는 것을 확인할 수 있고 accuracy는 train data보다 낮은 것을 확인 할 수 있다. 그렇지만 앞에서의 여러 옵션이 추가된 RNN 모형보다는 성능이 더 좋다는 것을 확인할 수 있다.

-GRU-

다음으로 고려해 볼 모형은 GRU 모형이다. GRU 모델은 LSTM을 개선한 모델로 LSTM 보다 parameter의 개수가 적어 연산 비용도 적고, 모델도 상대적으로 간단해서 학습속도도 더 빠른 상태에서 비슷한 성능을 내는 모델이다. GRU 모델에서는 forget gate와 input gate를 합쳐 update gate를 만들고 reset gate를 추가하였다.

우선 update gate는 이전의 정보를 얼마나 통과시킬지 결정하는 gate이다. 입력값과 hidden state 값을 각각 가중치에 곱하고 sigmoid 함수에 통과시켜 forget gate로 사용한다. 이 값을 1에서 빼서 input gate로 사용한다. 다음으로 reset gate는 이전의 hidden state의 정보를 얼마나 잊을지를 결정하는 gate이다. sigmoid activation function을 활용하여 0과 1 사이의 범위로 잊을 정보의 양을 결정한다. 모델의 구조는 아래와 같다.



```
from tensorflow.keras.layers import GRU

# GRU 모델 구축
model = Sequential()
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=100, input_length=max_length))
model.add(Bidirectional(GRU(128, return_sequences=True, recurrent_dropout=0.2)))
model.add(Bidirectional(GRU(128, recurrent_dropout=0.2)))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))

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

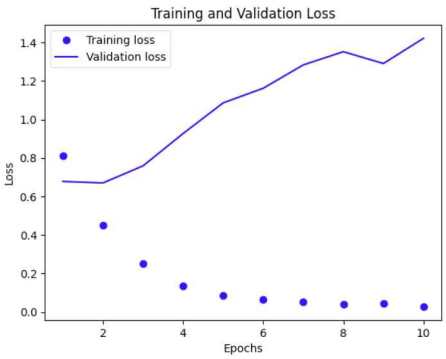
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 60, 100)           2058600

 bidirectional_3 (Bidirectio  (None, 60, 256)          176640
 nal)

 bidirectional_4 (Bidirectio  (None, 256)              296448
 nal)

 dropout_2 (Dropout)         (None, 256)               0

 dense_2 (Dense)             (None, 3)                 771

=================================================================
Total params: 2,532,459
Trainable params: 2,532,459
Non-trainable params: 0
_________________________________________________________________
```

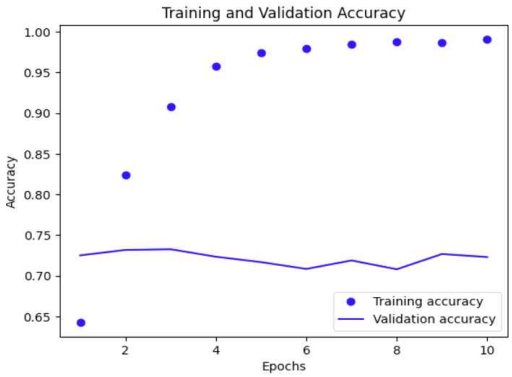
모델은 위와 같이 구축하였고 Bidirectional(GRU(128, return_sequences=True, recurrent_dropout=0.2)) 함수를 사용하여 양방향 GRU를 사용하고, GRU 층의 유닛 수를 128로 설정하였다. 또한 return_sequences=True로 설정하여 다음 GRU 층에 시퀀스 정보를 전달하고, recurrent_dropout=0.2로 순환 드롭아웃을 적용하였다. 또한 Bidirectional(GRU(128, recurrent_dropout=0.2))를 추가하여 하나의 단방향 GRU 층을 더 쌓았다. 이 층은 마지막 시점의 출력만 반환하며, recurrent_dropout=0.2로 순환 드롭아웃을 적용하였다. model fitting의 결과는 아래와 같다.

```
# 모델 훈련
history = model.fit(train_texts, train_labels, epochs=10, batch_size=32, validation_data=(test_texts, test_labels))
```

```
Epoch 1/10
300/300 [==============================] - 230s 741ms/step - loss: 0.8134 - accuracy: 0.6427 - val_loss: 0.6779 - val_accuracy: 0.7250
Epoch 2/10
300/300 [==============================] - 221s 736ms/step - loss: 0.4523 - accuracy: 0.8240 - val_loss: 0.6703 - val_accuracy: 0.7317
Epoch 3/10
300/300 [==============================] - 219s 729ms/step - loss: 0.2523 - accuracy: 0.9082 - val_loss: 0.7589 - val_accuracy: 0.7325
Epoch 4/10
300/300 [==============================] - 217s 723ms/step - loss: 0.1356 - accuracy: 0.9575 - val_loss: 0.9266 - val_accuracy: 0.7233
Epoch 5/10
300/300 [==============================] - 219s 732ms/step - loss: 0.0842 - accuracy: 0.9744 - val_loss: 1.0860 - val_accuracy: 0.7167
Epoch 6/10
300/300 [==============================] - 218s 727ms/step - loss: 0.0658 - accuracy: 0.9789 - val_loss: 1.1620 - val_accuracy: 0.7083
Epoch 7/10
300/300 [==============================] - 219s 732ms/step - loss: 0.0511 - accuracy: 0.9848 - val_loss: 1.2831 - val_accuracy: 0.7188
Epoch 8/10
300/300 [==============================] - 221s 739ms/step - loss: 0.0401 - accuracy: 0.9877 - val_loss: 1.3517 - val_accuracy: 0.7079
Epoch 9/10
300/300 [==============================] - 220s 735ms/step - loss: 0.0429 - accuracy: 0.9868 - val_loss: 1.2904 - val_accuracy: 0.7267
Epoch 10/10
300/300 [==============================] - 221s 736ms/step - loss: 0.0282 - accuracy: 0.9907 - val_loss: 1.4212 - val_accuracy: 0.7229
```

결과를 보면 epoch 10 기준으로 test data의 accuracy는 72.29%로 앞선 RNN 모형과 LSTM모델보다 성능이 더 좋게 나온 것을 확인할 수 있다.





plot을 살펴보면 이번 모델도 마찬가지로 test data의 loss는 epoch가 지날수록 점점 더 커지는 것을 확인할 수 있고 test data의 accuracy는 train data의 accuracy보다 낮은 것을 확인 할 수 있다. 이는 overfitting의 문제를 완전히 해결된 것으로 볼 수 없지만 다양한 모델을 fitting 시켜 성능을 비교해 보았을 때 test data의 accuracy 기준으로는 GRU 모델이 가장 좋은 것으로 확인 된다.

4. 결론

chatgpt가 국내에 출시된 이후 tweeter에는 chatgpt에 대한 다양한 tweets들이 많이 생겨났고 데이터는 kaggle에 있는 데이터를 이용하였고, 데이터는 chatgpt에 대한 다양한 의견이 들어있는 tweets들과 이 tweets에 대한 sentment로 이루어져 있고 sentiment 범주는 positive, neural, negative로 총 3가지의 범주로 구성되어 있다.

데이터 전처리는 tweets 안에 있는 https:를 기준으로 분리하여 주소가 들어간 부분을 제거하였고 text 데이터에 용이하지 않은 특수문자들은 다 제거하였다. 그 후 text 데이터를 정수로 인코딩화 하였고 pad_sequence 과정을 거쳤다. 다음으로 sentiment 변수를 분석에 용이하게 하기 위하여 one-hot encoding 처리하였다. 전체 데이터 중에 12000개만 따로 추출하여 train data와 test data를 8:2 비율로 분리하였다.

모든 전처리가 끝난 후에 simple RNN모형을 적용한 결과 epoch 10 기준으로 test data의 accuracy는 49.42%로 성능이 절반 정도만 맞출 정도로 좋지 않았다. 그래서 simple RNN 모형에 다양한 옵션을 추가하였는데 RNN 모형을 더 추가하여 RNN 모형의 층을 더 쌓았고 단순히 추가하는 것만이 아닌 retun_sequences=True 옵션과 Bidirectional 함수까지 적용하였다. 또한 overfitting을 막기 위해 dropout(rate=0.2)까지 추가적으로 적용하였다. 그 결과 test data의 accuracy는 epoch 10기준으로 64.46%로 모델의 성능이 더 향상되었다.

다음으로 LSTM 모델을 구축하였다. 추가적인 옵션으로 Bidirectional 함수와 return_sequences=True 옵션을 적용하였고 추가적인 층인 LSTM(128)과 Dropout(0.2)을 추가하였다. 그 결과 test data의 accuracy는 epoch 10기준 69.29%로 앞에 RNN 모형들보다 성능이 좋은 것을 확인 할 수 있다.

마지막으로 GRU 모델을 구축하였다. 마찬가지로 Bidirectional 함수와 recurrent_dropout=0.2 옵션을 적용하였고 model을 fitting 하였고 test data의 accuracy는 epoch 10 기준으로 72.29%로 fitting 된 모델 중에 가장 성능이 좋은 것을 확인할 수 있

었다. 모델의 plot들을 살펴본 결과 가장 성능이 좋은 GRU 모델임에도 불구하고 여전히 training data의 accuracy는 90%대가 나온 반면에 train data의 accuracy는 70% 초반대로 train data만큼 높은 accuracy를 보여주기에는 많이 부족한 것으로 보인다. 그래서 dropout을 추가한다던지 층을 더 늘린다던지 이것 이외에도 다른 옵션들을 더 추가적으로 적용해 보는 것을 고려해야 된다고 생각된다. 이러한 다양한 시도를 통해 모델의 성능을 test data의 accuracy 측면에서 더 높힐 수 있을 것이라고 기대한다.