인공지능프로그래밍

Lab5

Sequence Model of GRU

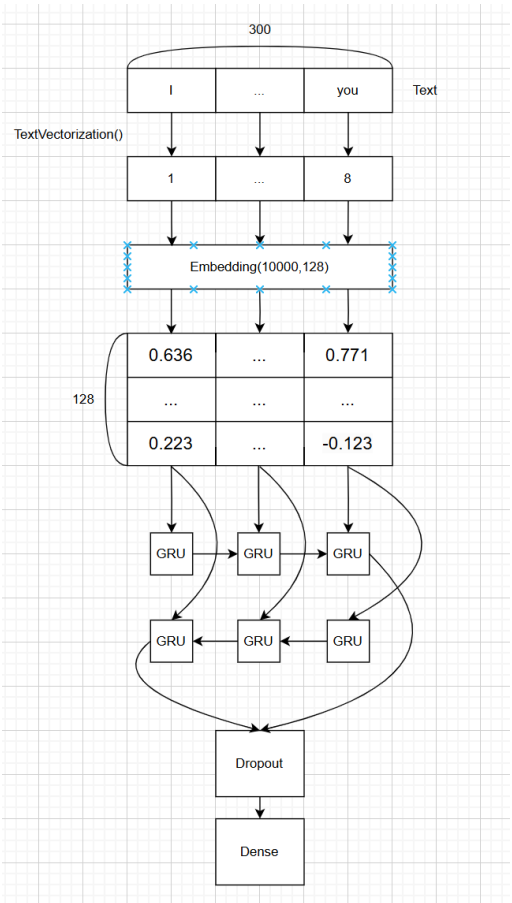
2024/10/31

2019202050 이강현

Lab Objective

LSTM의 간단한 구조인 GRU를 keras를 통해 구현해보며 그 구조와 동작방식을 이해한다. Keras.datasets에서 제공되는 imdb_reviews라는 긍정과 부정으로 클래스가 나뉘는 데이터를 tokenize하고 embedding하며 문장 속 단어가 LLM에서 어떻게 처리되는지 이해한다. 이때 처리된 단어들을 bidirectional GRU를 통해 문장이 학습되는 과정을 이해한다.

Program Flow



문장 속 단어들의 수는 300으로 제한되어 있고 그림은 처음 단어가 I, 끝 단어가 you인 하나의 문장을 보여준다. 문장은 단어를 기준으로 tokenize되고 embedding되어 bidirectional GRU에 입력되어 처리되는 전체적인 흐름을 볼 수 있다. 이후 dropout과 dense를 통과하여 클래스를 예측하게 된다. 추가적인 각 레이어에서의 설명은 이후 코드부분에서 자세히 언급하겠다.

Result

```
import os
os.environ["KERAS_BACKEND"] = "tensorflow"

import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds
import keras
import matplotlib.pyplot as plt
```

먼저 필요한 라이브러리를 import한다. 이번 과제의 핵심 라이브러리는 tensorflow_datasets로 긍정, 부정으로 나누어지는 리뷰에 대한 데이터셋이 있다.

```
#from keras.datasets import imdb
(ds_train, ds_test), ds_info = tfds.load('imdb_reviews',
                                         split=['train', 'test'],  # + 'unsupervised'
                                         shuffle_files=True,
                                         as_supervised=True,
                                         with_info=True)

print(ds_info.features)
```

```
FeaturesDict({
    'label': ClassLabel(shape=(), dtype=int64, num_classes=2),
    'text': Text(shape=(), dtype=string),
})
```

데이터셋을 불러오고 train과 test로 나누는 과정이다. 클래스는 긍정,부정으로 2가지가 있는 것을 확인한다. 데이터는 섞을 것이기에 shuffle_files가 true이다.

```
X_train = []
y_train = []

X_test_str = []
y_test = []

#preparing datasets
for sentense, label in ds_train:
    X_train.append(sentense.numpy().decode('utf8'))
    y_train.append(label.numpy())

for sentense, label in ds_test:
    X_test_str.append(sentense.numpy().decode('utf8'))  # X_test_str is used at the test stage
    y_test.append(label.numpy())

y_train = np.array(y_train)
y_test = np.array(y_test)
```

```
print(X_train[0])
print('The review is', 'Positive' if y_train[0]==1 else 'Negative')
```

```
This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside.
The review is Negative
```

이 부분은 데이터셋을 numpy로 변환하고 데이터와 라벨을 분리해서 저장하기 위함이다.

```
# hyperparameter for word embeddings
vocab_size = 10000
embedding_size = 128
max_length = 300

### START CODE HERE ###

tokenizer = keras.layers.TextVectorization(max_tokens=vocab_size, output_sequence_length=max_length) # tokenizer for integer encoding

tokenizer.adapt(X_train) # adjust to train datasets

X_train = tokenizer(X_train) # tokenize train datasets
X_test = tokenizer(X_test_str) # tokenize test datasets

### END CODE HERE ###
```

이 부분은 문장을 구성하는 단어의 종류가 10000개에 해당하고 이 단어들을 정수형태로 변환해야 하므로 TextVectorization 함수를 사용한다. 문장내 단어의 수는 최대 300으로 설정하여 이보다 커지면 자르고 작다면 0으로 padding한다. 함수를 train 데이터를 기준으로 맞추고 각각 train, test 데이터셋을 벡터화한다.

```
hidden_states = 64
dropout_rate = 0.5

model = keras.Sequential() # initialize sequential model

### START CODE HERE ###

model.add(keras.layers.InputLayer(shape=(max_length,)))  # input layer
model.add(keras.layers.Embedding(vocab_size,embedding_size))  # embedding layer
model.add(keras.layers.Bidirectional(keras.layers.GRU(hidden_states)))  # bidirectional GRU
model.add(keras.layers.Dropout(dropout_rate))  # dropout
model.add(keras.layers.Dense(1,activation="sigmoid"))  # output layer

### END CODE HERE ###

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']) # compile model

model.summary()
```

```
Model: "sequential_7"
```

Layer (type)	Output Shape	Param #
embedding_5 (Embedding)	(None, 300, 128)	1,280,000
bidirectional_1 (Bidirectional)	(None, 128)	74,496
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 1)	129

```
Total params: 1,354,625 (5.17 MB)
Trainable params: 1,354,625 (5.17 MB)
Non-trainable params: 0 (0.00 B)
```

핵심 단계인 모델 레이어를 쌓는 부분이다. 먼저 300의 단어 수로 맞추어진 문장이 입력으로 전달되기 때문에 inputlayer의 입력 크기를 300으로 맞추어준다. Embedding은 10000개의 단어 종류들을 각각 128 크기의 벡터로 변환하기 위해 위와 같이 설정한다. 128의 크기를 가지게 된 문장들은 hidden state가 64인 bidirectional GRU에 입력되게 되고 이후 regularization을 위한 dropout을 통과한 후 feedforward인 dense를 거친다.

추가적으로 각각 layer들이 가지는 parameter의 수에 대한 설명이다.

먼저 embedding은 10000개의 단어종류들을 각각 128개의 벡터 크기로 변환하기 위해 10000 x 128의 matrix가 필요할 것이다. 따라서 1280000의 parameter가 필요하다.

다음은 bidirectional layer이다. GRU는 reset gate, update gate, 다음 hidden state를 만드는 연산을 진행한다. 이때 reset gate, update gate는 hidden state와 input을 이어붙이고 이에 가중치를 행렬곱하는 연산을 한다. 이때 결과는 hidden state와 element-wise 연산을 위해 크기가 동일해야하므로 연산에 필요한 가중치 행렬의 모양은 64 x (128 + 64)가 될 것이다. 이와 같은 연산을 총 3번 진행하고 bidirection이라는 것을 고려해 64 x 192 x 3 x 2를 진행하면 73728의 결과가 나온다. 추가로 bias도 고려해야 한다. 이는 가중치와 곱해진 후 생성된 64에 대한 bias이고 이는 이전 hidden state와 input 각각에 대한 bias가 존재한다. Bias 또한 가중치와 마찬가지로 3번 더해지고 bidirection이라는 것을 고려해 최종적으로 64 x 2 x 3 x 2를 진행하면 768이다. 두 parameter를 더하면 73728 + 768 = 74496이다. Dropout은 입력된 비율만큼 마스킹처리하는 하는 것과 마찬가지이므로 parameter는 0이다. Dense는 입력된 128 크기 벡터를 1의 크기로 만들기 위한 가중치 행렬 128개와 바이어스 1개로 이루어져 129의 parameter 수를 가진다.

```
n_batch = 64

es = keras.callbacks.EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=4) # prevent overfitting as too many epochs
mc = keras.callbacks.ModelCheckpoint('GRU_imdb.keras', monitor='val_accuracy', mode='max', verbose=1, save_best_only=True) #save best state

results = model.fit(X_train, y_train,
            batch_size=n_batch,
            epochs=10,
            callbacks=[es, mc],
            validation_split=0.2)

print(results.history['loss'])
print(results.history['accuracy'])
```

위 코드는 모델이 학습될 때 batch 크기를 설정하고 validation loss를 기준으로 early stopping 여부와 loss를 epoch마다 저장해둘 수 있도록 하는 부분이다. 추가로 모델을 학습시키고 결과를 출력하는 코드이다.

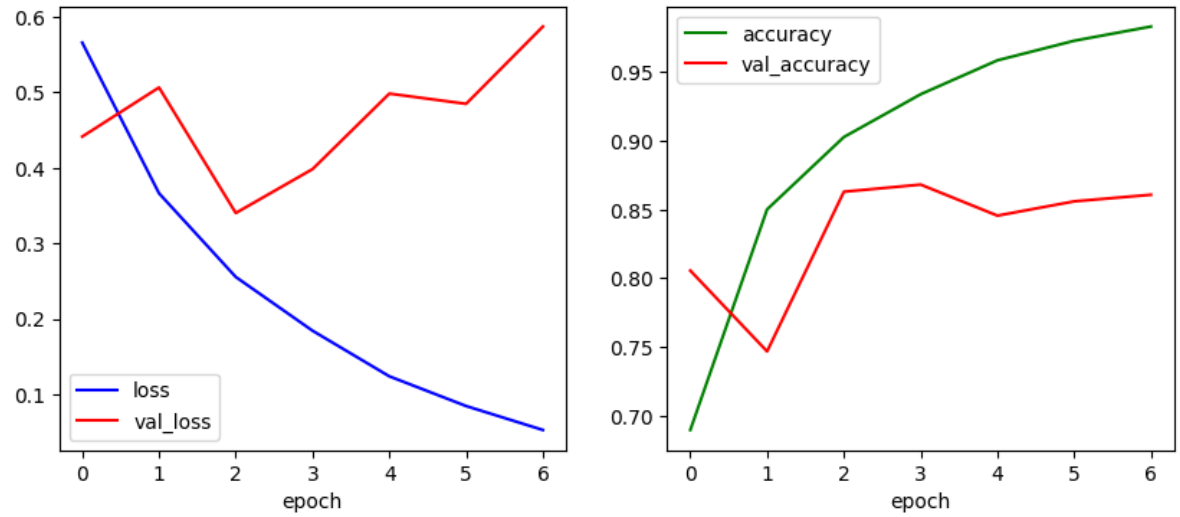
```
Epoch 1/10
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.5977 - loss: 0.6416
Epoch 1: val_accuracy improved from -inf to 0.80560, saving model to GRU_imdb.keras
313/313 ━━━━━━━━━━━━━━━━━━━━ 11s 29ms/step - accuracy: 0.5980 - loss: 0.6413 - val_accuracy: 0.8056 - val_loss: 0.4413
Epoch 2/10
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.8518 - loss: 0.3672
Epoch 2: val_accuracy did not improve from 0.80560
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 27ms/step - accuracy: 0.8518 - loss: 0.3672 - val_accuracy: 0.7468 - val_loss: 0.5063
Epoch 3/10
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - accuracy: 0.8973 - loss: 0.2705
Epoch 3: val_accuracy improved from 0.80560 to 0.86300, saving model to GRU_imdb.keras
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 27ms/step - accuracy: 0.8973 - loss: 0.2704 - val_accuracy: 0.8630 - val_loss: 0.3401
Epoch 4/10
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.9429 - loss: 0.1673
Epoch 4: val_accuracy improved from 0.86300 to 0.86820, saving model to GRU_imdb.keras
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 31ms/step - accuracy: 0.9429 - loss: 0.1674 - val_accuracy: 0.8682 - val_loss: 0.3982
Epoch 5/10
312/313 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.9611 - loss: 0.1169
Epoch 5: val_accuracy did not improve from 0.86820
313/313 ━━━━━━━━━━━━━━━━━━━━ 9s 27ms/step - accuracy: 0.9611 - loss: 0.1170 - val_accuracy: 0.8456 - val_loss: 0.4982
Epoch 6/10
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.9720 - loss: 0.0871
Epoch 6: val_accuracy did not improve from 0.86820
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 27ms/step - accuracy: 0.9720 - loss: 0.0870 - val_accuracy: 0.8560 - val_loss: 0.4847
Epoch 7/10
312/313 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.9858 - loss: 0.0475
Epoch 7: val_accuracy did not improve from 0.86820
313/313 ━━━━━━━━━━━━━━━━━━━━ 8s 25ms/step - accuracy: 0.9858 - loss: 0.0476 - val_accuracy: 0.8608 - val_loss: 0.5870
Epoch 7: early stopping
[0.5657184720039368, 0.3662283122539520, 0.2554053068161011, 0.1842335015535354, 0.1236588988780975, 0.0845220685005188, 0.0528282672166824]
[0.6895999908447266, 0.8501499891281128, 0.9028000235557556, 0.9339500069618225, 0.9586499929428101, 0.9728999733924866, 0.9832500219345093]
```

결과는 위와 같다. Epoch을 10으로 설정했으나 설정해둔 early stopping을 통해 7에서 멈추었고 epoch마다 저장해둔 loss와 accuracy가 출력되었다.

```
# plot loss and accuracy
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(results.history['loss'], 'b-', label='loss')
plt.plot(results.history['val_loss'], 'r-', label='val_loss')
plt.xlabel('epoch')
plt.legend()

plt.subplot(1,2,2)
plt.plot(results.history['accuracy'], 'g-', label='accuracy')
plt.plot(results.history['val_accuracy'], 'r-', label='val_accuracy')
plt.xlabel('epoch')
plt.legend()

plt.show()
```



그래프를 통해 epoch당 train과 validation 데이터에 따른 loss와 accuracy를 시각화하였다. Epoch이 증가하면서 validation loss가 높아지는 것을 보고 미리 멈춘 것으로 그래프를 통해 예상할 수 있다.

```
# model = keras.models.load_model('GRU_imdb.keras')
model.evaluate(X_test, y_test)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - accuracy: 0.8365 - loss: 0.6809
[0.6736342906951904, 0.8377599716186523]
```

테스트 결과를 확인한다.

```
idx = np.random.randint(X_test.shape[0]) #create random index

X_input = tf.reshape(X_test[idx], shape=(1,-1)) #pick one random sampling in test datasets

score = float(tf.squeeze(model.predict(X_input))) #save model's predict score

decision = 1 if score>0.5 else 0
rate = score if decision==1 else (1-score) # Above 0.5, use the score as is, below use 1-score.

print(X_test_str[idx])

print('The review is', 'Positive' if decision==1 else 'Negative', 'in {:.2f}% '.format(rate*100))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
The bearings of western-style Feminism on the various subcultures of India have hitherto remained largely non-existent,
The review is Negative in 99.76%
```

위는 무작위로 하나의 문장을 뽑아서 확인해본 결과 해당 문장이 99.76%의 확률로 부정적인 댓글이라고 예측하는 것을 보여주는 코드이다.

Discussion

제공되는 라이브러리들을 사용하여 데이터셋부터 tokenize, embedding의 연산이 어떻게 이루어지는지 keras에서 어떻게 사용하는지 알 수 있었다. 추가로 각 레이어를 쌓아보면서 함수 인자에 어떤 것이 들어가야 할지 고민하고 parameter의 수를 직접 강의자료를 보고 계산해보면서 정확히 맞아 떨어지는 것이 신기했다. 앞에서부터 한번 뒤에서부터 한번 state를 뽑아내는 Bidirectional GRU를 보고 문장이 만약 더 길어져서 중간에서 필요한 정보가 있는 경우에는 어떻게 state를 계산하면 좋을까라는 의문점이 생겼다. 또 중앙에서 시작해서 서로 반대방향으로 state를 계산해 뽑아내면 더 길이가 짧고 효율적이지 않을까란 생각을 하게되었다.