순환 신경망 이해하기 - LSTM

학습 내용

- LSTM에 대해 알아보기
- IMDB 데이터 셋을 불러와 모델을 구축하고 학습을 수행시켜본다.

목차

01. LSTM 기본 이해

목차로 이동하기

- RNN의 한 종류입니다. RNN의 장기 의존성 문제(long-term dependencies)를 해결하기 위해 나온 모델입니다.
- 시계열 처리, 자연어 처리에 많이 사용되며, 광범위하게 사용되고 있습니다.

LSTM 층으로 모델을 구성하고 IMDB 데이터에서 훈련

- SimpleRNN외에 다른 RNN 알고리즘도 있습니다. LSTM과 GRU 2개입니다.
- LSTM 층은 출력 차원만 지정하고 다른 (많은) 매개변수는 기본값을 사용합니다.
- 좋은 기본값을 가지고 있어, 직접 매개변수를 튜닝하는 데 시간을 쓰지 않고도 거의 항상 어느정도 작동하는 모델 획득 가능

라이브러리 불러오기

In [1]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN
from tensorflow.keras.layers import LSTM

from tensorflow.keras.datasets import imdb
from tensorflow.keras.layers import Dense
from tensorflow.keras.preprocessing import sequence
import matplotlib.pyplot as plt
```

데이터 셋

- IMDB(IMDB Movie Review Sentiment Analysis)

- 감성 분류를 연습하기 위해 자주 사용하는 영화 리뷰 데이터 셋.
- 리뷰에 대한 텍스트, 긍정의 경우 1, 부정의 경우 0의 레이브로 구성.
- imdb.data_load()를 통해 영화 리뷰 데이터 로드 가능.
    - num_words를 사용하여 데이터의 등장 빈도 순위로 몇번째 단어까지 사용할 것인가?

In [2]:

```
max_features = 10000  # 특성으로 사용할 단어의 수
maxlen = 500    # 텍스트의 길이(가장 빈번한 max_features 개의 단어만 사용합니다)
batch_size = 32 # 배치 데이터 사이즈

(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)
print(len(input_train), '훈련 시퀀스')
print(len(input_test), '테스트 시퀀스')
print(input_train[0], y_train[0])
```

```
25000 훈련 시퀀스
25000 테스트 시퀀스
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256,
5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2,
336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 202
5, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1
247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2
223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135,
48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117,
5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4,
2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18,
4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 10
4, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 534
5, 19, 178, 32] 1
```

In [3]:

```
# 리스트를 (samples, maxlen)크기의 2D 정수 텐서로 변환
print('시퀀스 패딩 (samples x time)')
print('input_train 크기:', input_train.shape)
print('input_test 크기:', input_test.shape)
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train 크기:', input_train.shape)
print('input_test 크기:', input_test.shape)
```

```
시퀀스 패딩 (samples x time)
input_train 크기: (25000,)
input_test 크기: (25000,)
input_train 크기: (25000, 500)
input_test 크기: (25000, 500)
```

LSTM과 GRU층 이해하기

- simpleRNN은 이론적으로 시간 t에서 이전의 모든 타임스텝의 정보를 유지할 수 있다.
- 실제로 긴 시간에 걸친 의존성은 학습할 수 없는 것이 문제.
    - 층이 많은 일반 네트워크(피드포워드 네트워크)에서 나타나는 것과 비슷한 현상인 그래디언트 소실 문제(vanishing gradient problem)때문.
    - 1990년대 초 호크라이터(Hochreiter), 슈미트후버(Schmidhuber), 벤지오(Bengio)가 이런 현상에 대한 원인을 연구.

    - 이 문제를 해결하기 위해 고안된 것은 LSTM과 GRU층이다.
- 이것은 SimpleRNN의 기능 개선한 변종이다. 정보를 여러 타임스텝에 걸쳐 나르는 방법이 추가.

02. 모델 구축

목차로 이동하기

- 임베딩(embedding)은 자연어 처리(NLP)분야에서 사람이 사용하는 언어를 기계가 이해할 수 있도록 벡터로 바꾼 결과를 의미
- 워드 임베딩은 텍스트 내의 단어들을 밀집 벡터(dense vector)로 만드는 것을 의미.
- Embedding() : Embedding()은 단어를 밀집 벡터로 만드는 역할.

- Embedding()은 (samples, input_length)인 2D 정수 텐서 입력. (3D텐서도 있음)
    - samples은 정수 인코딩된 결과, 정수의 시퀀스. 가변 길이의 경우, 제로패딩 등으로 맞춘다.
- Embedding층은 크기가 3D텐서를 리턴
    - (samples, input_length, embedding_dimensionality)인 3D 텐서를 반환
- Embedding()
    - 첫번째 인자 : 단어집합의 크기. 총 단어의 수
    - 두번째 인자 : 임베딩 벡터의 출력 차원. 결과로 나오는 임베딩 벡터의 크기
    - input_length : 입력 시퀀스의 길이

- LSTM(num_units)
    - num_units는 hidden state(output)의 차원.

In [4]:

```
model = Sequential()
model.add(Embedding(max_features, 32))

model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))
```

In [5]:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 32)          320000

 lstm (LSTM)                 (None, 32)                8320

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 328,353
Trainable params: 328,353
Non-trainable params: 0
_________________________________________________________________
```

In [6]:

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
```

03. 모델 학습 및 평가결과 확인

목차로 이동하기

In [7]:

```
%%time

history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

```
Epoch 1/10
157/157 [==============================] - 38s 225ms/step - loss: 0.5068 - acc: 0.75
78 - val_loss: 0.3977 - val_acc: 0.8394
Epoch 2/10
157/157 [==============================] - 32s 202ms/step - loss: 0.2926 - acc: 0.88
78 - val_loss: 0.4592 - val_acc: 0.8304
Epoch 3/10
157/157 [==============================] - 33s 209ms/step - loss: 0.2368 - acc: 0.91
17 - val_loss: 0.3514 - val_acc: 0.8500
Epoch 4/10
157/157 [==============================] - 32s 203ms/step - loss: 0.2019 - acc: 0.92
52 - val_loss: 0.3669 - val_acc: 0.8478
Epoch 5/10
157/157 [==============================] - 30s 192ms/step - loss: 0.1784 - acc: 0.93
62 - val_loss: 0.3402 - val_acc: 0.8872
Epoch 6/10
157/157 [==============================] - 30s 193ms/step - loss: 0.1582 - acc: 0.94
34 - val_loss: 0.7199 - val_acc: 0.7838
Epoch 7/10
157/157 [==============================] - 30s 194ms/step - loss: 0.1447 - acc: 0.94
78 - val_loss: 0.3439 - val_acc: 0.8840
Epoch 8/10
157/157 [==============================] - 30s 193ms/step - loss: 0.1295 - acc: 0.95
54 - val_loss: 0.3455 - val_acc: 0.8498
Epoch 9/10
157/157 [==============================] - 31s 200ms/step - loss: 0.1184 - acc: 0.95
89 - val_loss: 0.3737 - val_acc: 0.8794
Epoch 10/10
157/157 [==============================] - 34s 215ms/step - loss: 0.1106 - acc: 0.96
14 - val_loss: 0.3655 - val_acc: 0.8840
CPU times: total: 19min 51s
Wall time: 5min 20s
```

In [10]:

```
model.evaluate(input_test, y_test)
```

```
782/782 [==============================] - 25s 32ms/step - loss: 0.4142 - acc: 0.865
8
```

Out[10]:

```
[0.4142196476459503, 0.8658000230789185]
```

검증의 손실과 정확도

In [8]:

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

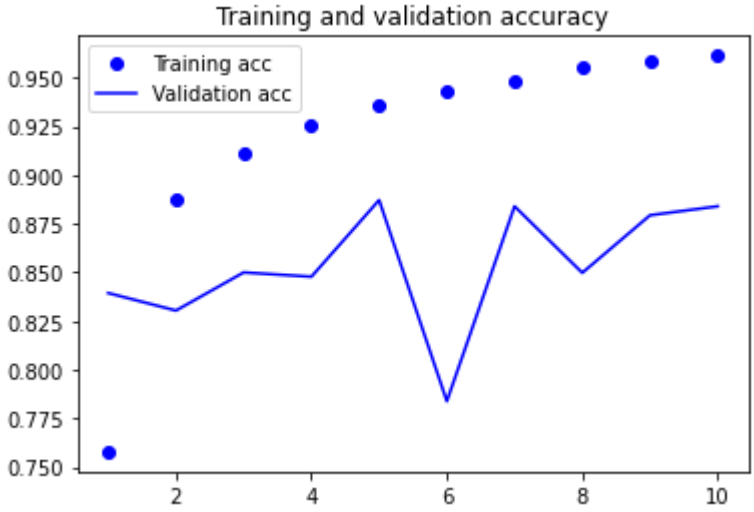
In [9]:

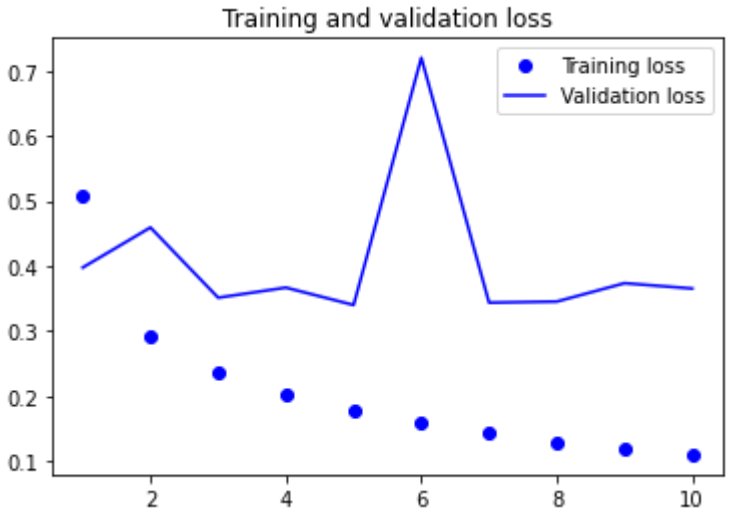
```
epochs = range(1, len(acc) + 1)

# 정확도
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()
```





결과 확인

- 좀 더 높은 검증 정확도 달성.
- SimpleRNN 네트워크보다 성능이 더 나은 결과를 보임. LSTM이 그래디언트 소실 문제로부터 덜 영향을 받기 때문.
- 대부분 LSTM이 낫지만, 모든 분야에 나은 것은 아닐수 있다. case by case
    - LSTM이 더 높은 성능을 보이는 곳은 시계열 처리와 자연어 처리문제에 많이 사용됨.
        - 질문-응답(question-answering)과 기계 번역(machine translation)