인공지능 MT Report

Classifying movie reviews:
a binary classification example

201632230 컴퓨터공학과 이다은

2020. 06. 01

0. 실습문제 전처리

주어진 train data 25,000개를 15,000개의 partial train data와 10,000개의 partial test data로 나누어 새로 만드시오.

python의 slicing을 이용해서 train data 25,000개를 15,000개의 partial train data와 10,000개의 partial test data로 나누었다. 해당 데이터로 1, 2번의 실습을 진행했다.

```
In [4]: # train data 25,000개 >>> partial train data 15,000개 + partial test data 10,000개

        # data
        partial_train_data = vector_train_data[:15000] # 15,000개
        partial_test_data = vector_train_data[15000:] # 10,000개

        # labels
        partial_train_labels = vector_train_labels[:15000] # 15,000개
        partial_test_labels = vector_train_labels[15000:] # 10,000개
```

[그림 0-1] 전처리 과정의 소스코드

1-1. 실습문제 정의

15,000개의 partial train data를 사용하여 3-fold cross validation 결과를 보이시오.

1-2. 문서화된 소스 코드

우선, 신경망 모델은 만들었다. 모델은 10,000개의 input을 가진 input layer, 16개의 units을 가진 2개의 hidden layers, 1개의 unit을 가진 1개의 output layer로 구성됐다.

```
In [5]: # 신경망 모델 만들기

        from keras import models
        from keras import layers

        def build_model():
            model = models.Sequential()

            model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
            model.add(layers.Dense(16, activation='relu'))
            model.add(layers.Dense(1, activation='sigmoid'))

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

            return model
```

[그림 1-2-1] 신경망 모델 소스코드

다음으로, 3-fold cross validation을 진행했다. 각 fold마다 fold_train_data, fold_train_labels로 훈련이, fold_val_data, fold_val_labels로 검증이 이뤄졌다. 매 fold 검증마다 산출되는 loss와 accuracy는 각각 loss_scores와 acc_scores 리스트에 추가됐다.

```
In [6]: # 3-fold cross validation

        k = 3
        n_val_samples = len(partial_train_data) // k

        n_epochs = 20

        loss_scores = []
        acc_scores = []

        for i in range(k): # i == 0, 1, 2
            print('processing fold #', i)

            # 폴드 검증 데이터
            fold_val_data = partial_train_data[i * n_val_samples:(i+1) * n_val_samples]
            fold_val_labels = partial_train_labels[i * n_val_samples:(i+1) * n_val_samples]

            # 폴드 훈련 데이터
            fold_train_data = np.concatenate([partial_train_data[:i * n_val_samples], partial_train_data[(i+1) * n_val_samples:]], axis=0)
            fold_train_labels = np.concatenate([partial_train_labels[:i * n_val_samples], partial_train_labels[(i+1) * n_val_samples:]], axis=0)

            model = build_model() # 모델 정의

            history = model.fit(fold_train_data, # 모델 폴드 훈련
                                fold_train_labels,
                                epochs=n_epochs,
                                batch_size=512,
                                validation_data=(fold_val_data, fold_val_labels),
                                verbose=0)

            fold_val_loss, fold_val_acc = model.evaluate(fold_val_data, fold_val_labels, verbose=0)

            loss_scores.append(fold_val_loss)
            acc_scores.append(fold_val_acc)

processing fold # 0
processing fold # 1
processing fold # 2
```

[그림 1-2-2] 3-fold cross validation 소스코드

3번의 fold를 거친 후의 loss_scores, acc_scores는 다음과 같다.

```
In [13]: loss_scores
Out[13]: [0.6521190624475479, 0.636427992117405, 0.6988423901319504]

In [14]: acc_scores
Out[14]: [0.8587999939918518, 0.8644000291824341, 0.8597999811172485]
```

[그림 1-2-3] loss_scores, acc_scores 리스트 확인

각 리스트의 평균 값을 구하면 다음과 같다. 해당 평균 값들은 최종 validation loss score, validation accuracy score이다.

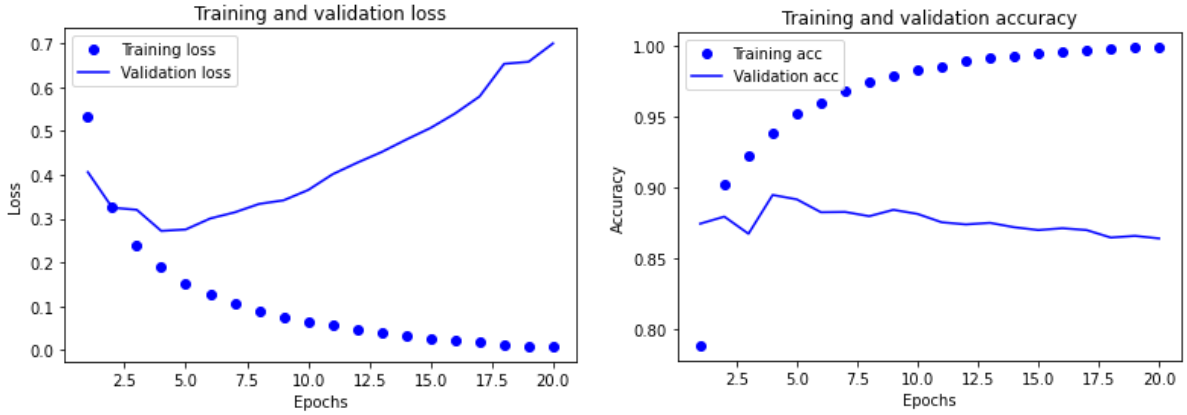
```
In [15]: np.mean(loss_scores) # 최종 validation loss
Out[15]: 0.662463148232301

In [16]: np.mean(acc_scores) # 최종 validation accuracy
Out[16]: 0.8610000014305115
```

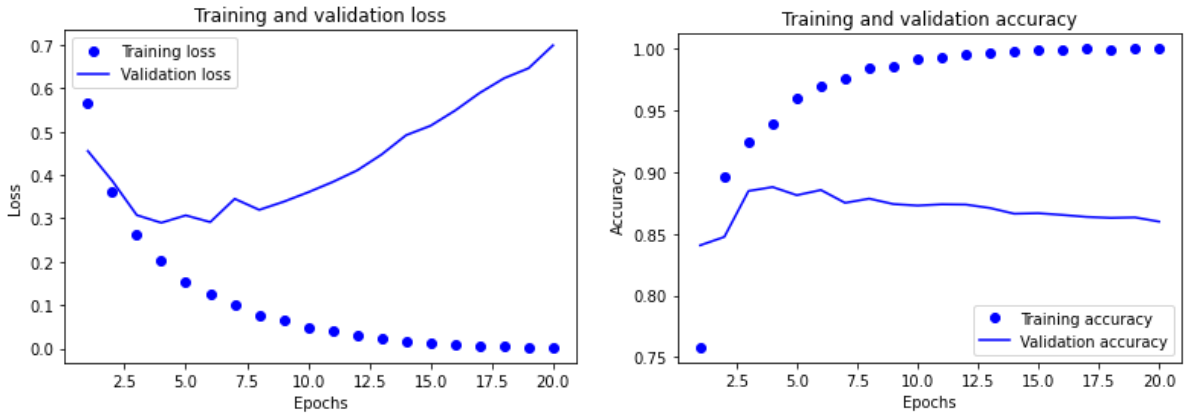
[그림 1-2-4] 최종 validation loss score, validation accuracy score

1-3. 실행 결과, 실행 결과에 대한 설명 및 검토

3-fold cross validation의 효과를 알아보기 위해 3-fold cross validation 적용 전과 적용 후의 그래프를 각각 첨부했다. 왼쪽은 학습 및 검증 loss, 오른쪽은 학습 및 검증 accuracy 그래프이다.



[그림 1-3-1] 3-fold cross validation 적용 전, 학습 및 검증 loss 및 accuracy 그래프



[그림 1-3-2] 3-fold cross validation 적용 후, 학습 및 검증 loss 및 accuracy 그래프

3-fold cross validation 적용 전이나 적용 후는 비슷한 양상을 보인다. 4번째 epoch 뒤에 overfitting이 시작되는 걸 알 수 있다.

1-4. 실습을 통한 이해 및 개선 방안

검증 dataset이 아주 적을 때에 k-fold cross validation을 이용해 훈련과 검증을 진행하면 효과적이다. 그러나, 해당 데이터의 훈련 및 검증 dataset은 각각 15,000개와 10,000개로 충분히 많다. 이런 경우에 k-fold cross validation을 사용해도 훈련 및 검증 과정에서 크게 변화가 없다. [그림 1-3-1], [그림 1-3-2]를 통해서도 알 수 있듯이, 3-fold cross validation 적용 전과 적용 후의 결과 그래프는 거의 유사하게 도출되었다.

1-5. 결론

검증 dataset이 적은 경우 k-fold cross validation을 이용하면 효과적인 검증 결과를 얻게 된다. 한편, 현재 데이터의 검증 dataset은 적지 않아 3-fold cross validation을 진행해도 충분히 좋은 결과를 얻을 수 있다.

2-1. 실습문제 정의

15,000개의 partial train data와 10,000개의 partial test data를 사용하여 overfitting을 최소로 하는 신경망 모델을 케라스로 프로그램하고, 그래프 등을 사용하여 결과를 보여주고, 그에 대해 분석 설명하시오. 여기서 사용되는 test dataset은 원본 25,000개의 test dataset으로 test 하시오.

2-2. 문서화된 소스 코드

신경망에서 overfitting을 방지하기 위해 가장 널리 사용하는 방법은 총 네 가지가 있다. ①훈련 데이터를 더 모은다. ②네트워크의 용량을 감소시킨다. ③가중치 규제를 추가한다. ④드롭아웃을 추가한다.

현 데이터에서 훈련 데이터를 더 수집하는 건 불가능하므로, 방법 ②, ③, ④를 사용해서 실습을 진행했다.

각 방법별로 신경망 모델을 함수로 만들고, 모델을 변수에 정의하고, 훈련과 검증을 한다. 모델 훈련 시의 epochs와 batch_size는 강의 내용과 동일한 값을 적용했다.

In [89]:
```
# 네트워크 크기 축소

# 신경망 모델 만들기
def small_size_model():
    model = models.Sequential()

    model.add(layers.Dense(6, activation='relu', input_shape=(10000,)))
    model.add(layers.Dense(6, activation='relu'))
    model.add(layers.Dense(1, activation='sigmoid'))

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

    return model
```

In [90]:
```
small_size_model = small_size_model() # 모델 정의

history = small_size_model.fit(partial_train_data, # 모델 훈련
                               partial_train_labels,
                               epochs=20,
                               batch_size=512,
                               validation_data=(partial_test_data, partial_test_labels), # 모델 검증
                               verbose=0)
```

[그림 2-2-1] 네트워크 크기 축소 (방법 ②) 소스코드

In [95]:
```
# 가중치 규제 추가 (L2 규제)

# 신경망 모델 만들기
def l2_regularization_model():
    model = models.Sequential()

    model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001), activation='relu', input_shape=(10000,)))
    model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001), activation='relu'))
    model.add(layers.Dense(1, activation='sigmoid'))

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

    return model
```

In [96]:
```
from keras import regularizers

l2_regularization_model = l2_regularization_model() # 모델 정의

history = l2_regularization_model.fit(partial_train_data, # 모델 훈련
                                       partial_train_labels,
                                       epochs=20,
                                       batch_size=512,
                                       validation_data=(partial_test_data, partial_test_labels), # 모델 검증
                                       verbose=0)
```

[그림 2-2-2] 가중치 규제 추가 (L2 규제, 방법 ③) 소스코드

In [100]:
```
# 드롭아웃 추가

# 신경망 모델 만들기
def dropout_model():
    model = models.Sequential()

    model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(16, activation='relu'))
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(1, activation='sigmoid'))

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

    return model
```

In [101]:
```
dropout_model = dropout_model() # 모델 정의

history = dropout_model.fit(partial_train_data, # 모델 훈련
                            partial_train_labels,
                            epochs=20,
                            batch_size=512,
                            validation_data=(partial_test_data, partial_test_labels), # 모델 검증
                            verbose=0)
```

[그림 2-2-3] 드롭아웃 추가 (방법 ④) 소스코드

2-3. 실행 결과, 실행 결과에 대한 설명 및 검토

각 방법별로 overfitting 방지가 잘 됐는지 알아보기 위해 기존 신경망 모델 실행 결과와 비교해 보았다. 학습 및 검증 loss 및 accuracy 그래프 시각화를 진행하고, 최종 테스트 loss 및 accuracy를 확인했다.

[그림 1-3-1]에 기존 신경망 모델 학습 및 검증 loss 및 accuracy 그래프가 이미 존재해 [그림 2-3-1]에 기존 신경망 모델 최종 테스트 loss 및 accuracy 결과만 추가 첨부했다.

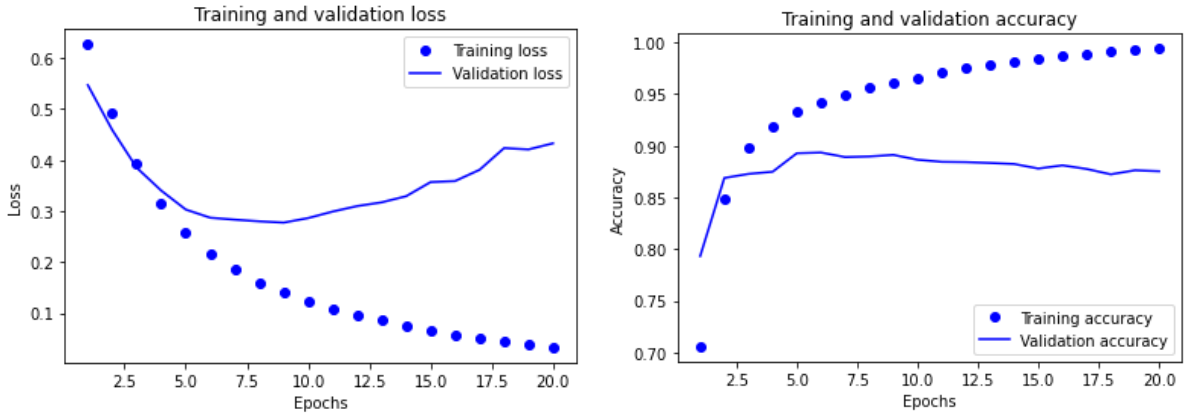
In [88]:
```
# 최종 결과 확인

results = build_model.evaluate(vector_test_data, vector_test_labels)
results
```

```
25000/25000 [==============================] - 3s 127us/step
```

Out[88]:
```
[0.7740436159527302, 0.8474000096321106]
```

[그림 2-3-1] 기존 신경망 모델, 최종 테스트 loss 및 accuracy 결과



[그림 2-3-2] 네트워크 크기 축소 적용 후, 학습 및 검증 loss 및 accuracy 그래프

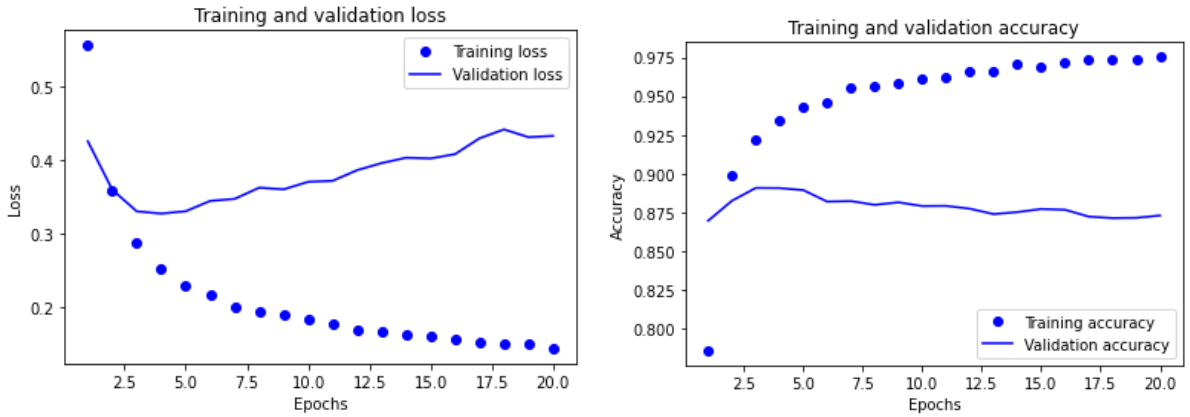
```
In [94]: small_size_result = small_size_model.evaluate(vector_test_data, vector_test_labels) # 모델 테스트
         small_size_result

         25000/25000 [==============================] - 3s 128us/step

Out[94]: [0.4775712836718559, 0.8592000007629395]
```

[그림 2-3-3] 네트워크 크기 축소 적용 후, 최종 테스트 loss 및 accuracy 결과

네트워크 크기 축소를 적용하면 기존 신경망 모델보다 overfitting이 완화된 걸 확인할 수 있다. 또한, 최종 테스트의 loss는 감소하고 accuracy는 증가했다.



[그림 2-3-4] 가중치 규제 추가 (L2 규제) 후, 학습 및 검증 loss 및 accuracy 그래프

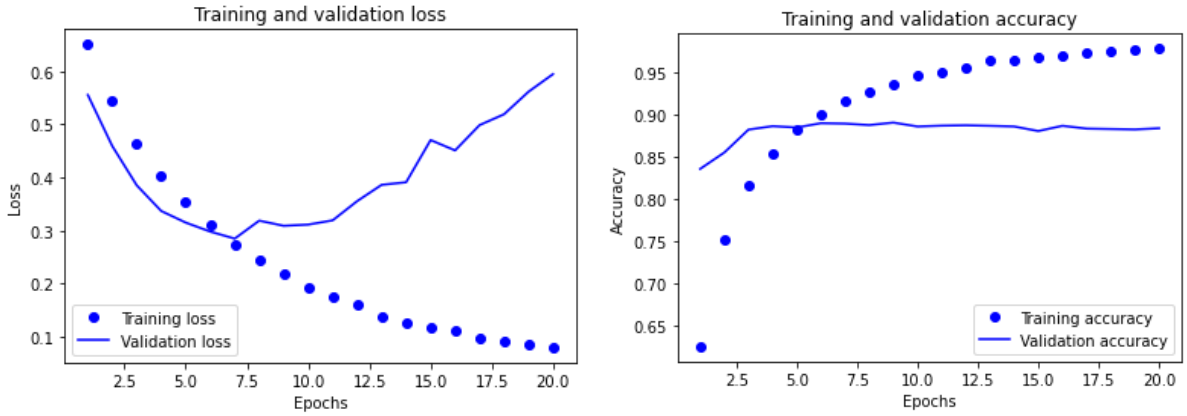
```
In [99]: l2_regularization_result = l2_regularization_model.evaluate(vector_test_data, vector_test_labels) # 모델 테스트
         l2_regularization_result

         25000/25000 [==============================] - 3s 131us/step

Out[99]: [0.4584274447917938, 0.8646799921989441]
```

[그림 2-3-5] 가중치 규제 추가 (L2 규제) 후, 최종 테스트 loss 및 accuracy 결과

가중치 규제 추가 (L2 규제)를 적용하면 기존 신경망 모델보다 overfitting이 완화되었고, 최종 테스트 loss가 감소하고 accuracy가 증가한 걸 확인할 수 있다.



[그림 2-3-6] 드롭아웃 추가 후, 학습 및 검증 loss 및 accuracy 그래프

```
In [104]: dropout_result = dropout_model.evaluate(vector_test_data, vector_test_labels) # 모델 테스트
          dropout_result
          25000/25000 [==============================] - 3s 137us/step
Out[104]: [0.6058519396340847, 0.8723999857902527]
```

[그림 2-3-7] 드롭아웃 추가 후, 최종 테스트 loss 및 accuracy 결과

드롭아웃을 적용한 경우에도 기존 신경망 모델보다 overfitting이 완화됐고, 최종 테스트 loss가 감소하고 accuracy가 증가한 걸 확인할 수 있다.

2-4. 실습을 통한 이해 및 개선 방안

신경망의 overfitting을 방지하기 위해 사용하는 방법 중 드롭아웃 추가를 적용한 경우에 신경망 모델의 overfitting이 최소화되었다. 최종 테스트 accuracy를 중점적으로 살펴보면, 기존 신경망 모델은 약 84.7%의 accuracy를 보인 반면, 드롭아웃 추가를 적용한 모델은 약 87.2%의 accuracy를 산출했다. 한편, 네트워크 크기 축소를 적용한 모델은 약 85.9%, 가중치 규제 추가 (L2 규제)를 적용한 모델은 약 86.4%의 accuracy를 나타냈다. 따라서, 기존 신경망 모델에서 overfitting을 최소화하려면 신경망 모델에 드롭아웃 추가를 적용하는 것이 가장 효과적이다.

2-5. 결론

기존 신경망 모델의 overfitting을 보완하기 위해 네트워크 크기 축소, 가중치 규제 추가 (L2 규제), 드롭아웃 추가의 방법들을 이용했다. 이 중에서 드롭아웃 추가를 적용했을 때, 신경망 모델의 overfitting이 최소화되었고 최종 테스트 accuracy도 가장 높게 나타났다.