BE201 AI Basic HW1 202011047 김승태 report

2 Spam Mail Classification by Naive Bayes

2.(e)
먼저, spam_prob()를 구현하여서, 전체 중에서 spam이 얼마나 있는지를 계산한다. 이는 전체에서 spam이 나올 확률이다. 이 implementation에서는 word_exists()를 사용하여서, 각각 스팸과 햄 메일에 특정 단어들이 들어가 있는 조건부 확률을 계산한다. 모델은 위의 가능성, 즉 확률에 기반하여서 이메일이 스팸인지 햄인지 분류한다.
결과값은 다음과 같다.

Accuracy 0.9491626794258373 (1587 / 1672)

Accuracy (scikit-learn) 0.9796650717703349 (1638 / 1672)

2.(f)
각기 실행을 시켜 보았다. 결과값은 다음과 같다.
Accuracy 0.9491626794258373 (1587 / 1672) (셋 다 동일)

MultinomialNB - Accuracy (scikit-learn) 0.9796650717703349 (1638 / 1672)

BernoulliNB - Accuracy (scikit-learn) 0.9766746411483254 (1633 / 1672)

ComplementNB - Accuracy (scikit-learn) 0.9617224880382775 (1608 / 1672)

결론적으로, 대부분 비슷하였지만, ComplementNB가 다른 방식보다 정확도가 떨어졌고, MultinomialNB가 우리가 사용하는 코드에서는 가장 정확도가 높은 것으로 드러났다.

2.(g)
정확도를 향상시키는 방법 중, Laplace smoothing 기법을 사용하였다. 이는 매개변수 alpha를 도입하여서, train set에 없는 단어에 대한 확률을 제공하고, 0으로 나누어지는 것을 피할 수 있다.
바꾼 코드는 다음과 같다.

```
def spam_cond_prob(self, word, alpha=1.0):

    num_spam_with_word = sum(1 for content, label in self.train_dataset if label == 1 and word in
tokenize(content))

    num_spam = len(self.spam_mail_list)

    return (num_spam_with_word + alpha) / (num_spam + alpha * len(self.all_words_list))



def ham_cond_prob(self, word, alpha=1.0):
```

```
    num_ham_with_word = sum(1 for content, label in self.train_dataset if label == 0 and word in
tokenize(content))

    num_ham = len(self.ham_mail_list)

    return (num_ham_with_word + alpha) / (num_ham + alpha * len(self.all_words_list))
```

이것의 결과값은 다음과 같다.

Accuracy 0.9671052631578947 (1617 / 1672)

Accuracy (scikit-learn) 0.9796650717703349 (1638 / 1672)

즉, 이 smoothing 기법을 이용하였더니,

Accuracy 0.9491626794258373 (1587 / 1672)에서
Accuracy 0.9671052631578947 (1617 / 1672)으로 증가된 것을 알 수 있다.(약 0.02 증가)

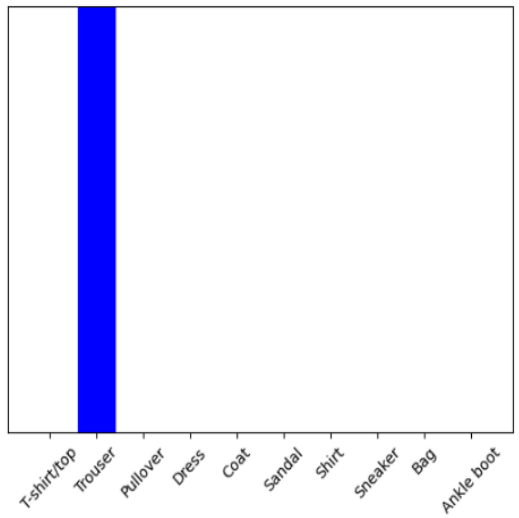
3 Classification by deep neural networks

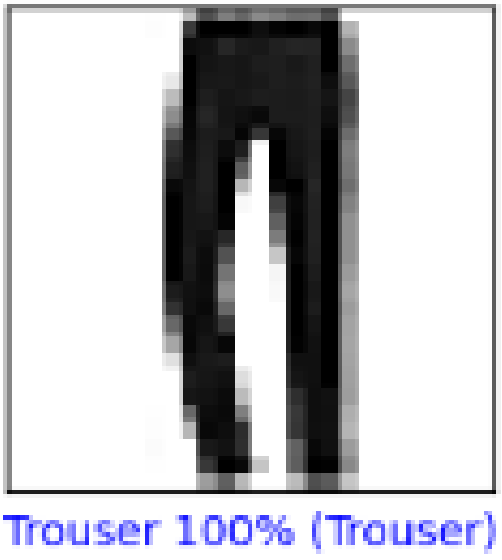
3.(a) After training is done, plot output values of the network for an arbitrary image from test dataset.

```
image = test_images[2]
image = (np.expand_dims(image,0))
predictions_single = probability_model.predict(image)

plot_value_array(2, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()
```

```
1/1 [==============================] - 0s 139ms/step
```





test_images 에서 2번째 이미지를 불러와서 output을 불러오도록 하였다. 여기서 output은 막대그래프 형식으로 나타난다.

3.(b) Explain the meaning of the output values.

먼저, 데이터 전처리 과정에서, 우리가 이미지 픽셀 값을 0에서 1 사이로 정규화를 하였고, 그것이 막대그래프의 세로축에 적용되어서 확률의 합을 1로 만들어, 그것을

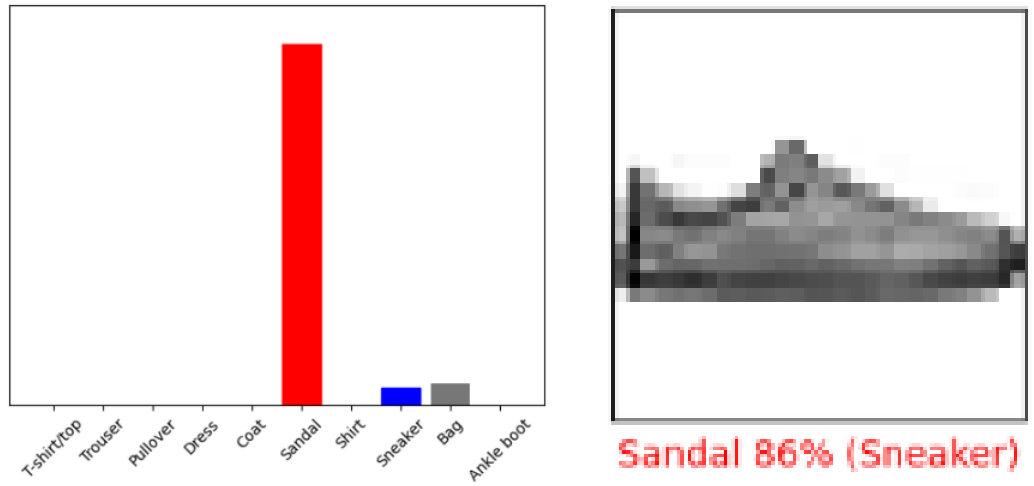
그대로 y축 값에 반영한 것이다. 즉, y축은 0부터 1까지의 확률값이 되는 것이다. 다시 말하자면, 각 클래스에 대한 모델의 확률 점수라고 할 수 있다. 이 중 가장 높은 확률을 가진 클래스가 예측 클래스가 된다.

3.(c) Find a wrong prediction from test set and report values. Again, explain the meaning of output values.

```
image = test_images[12]
image = (np.expand_dims(image, 0))
predictions_single = probability_model.predict(image)

plot_value_array(12, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()
```

```
1/1 [==============================] - 0s 53ms/step
```



이는 잘못 예측된 레이블이 있는 결과값이다.

```
def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

이 코드에서 볼 수 있듯, i는 true_label에서 i에 해당하는 실제 레이블을 가져오는데, 이 12번 이미지는 그것의 실제 레이블인 Sneaker와 다른 것이다. 그래서 Sneaker라고 예상된 부분이 파란색으로 떠 있는데, Sandal이라고 예측을 해 버린 것이다. 그래서 Sandal일 확률이 다른 것들보다 높기 때문에, Sandal이 빨간색 막대가 되어 있는 것이다. 왜냐하면, `thisplot[predicted_label].set_color('red')` 예측된 클래스에 해당하는 막대를 빨간색으로 설정하기 때문이다. 이것이 실제 레이블과 일치하면 파란 막대로 덮히게 되는 것이다.

3.(d) Please explain Flatten, Dense layer, Adam optimizer, and SparseCategoricalCrossentropy.

Flatten : Flatten은 2차원의 배열(신경망의 입력 데이터)을 1차원 배열로 변환하는 것이다. 여기에서는 28*28의 2차원 배열을 그 곱인 784 의 크기를 가진 1차원 배열로 펼쳐준다.
Dense layer : 각 입력 노드가 모든 출력 노드에 연결된 완전 연결층이며, 이 layer는 입력과 출력 사이의 매핑을 생성한다. 각 연결은 가중치를 가진다. 각각은 노드들을 가진다. 여기서 첫 번째 Dense layer는 128개의 유닛과 Relu 함수를 가진 은닉형 레이어로 사용된다. 두 번째 층은 10개의 확률을 반환하고, 그 합은 1이다. 이 노드들이 결국 10개 클래스 중 하나에 속할 확률인 것이다.

Adam optimizer(Adaptive Moment Estimation) : 각 매개변수마다 학습률을 조정하여서 전역 최적점과 지역 최적점 사이의 균형을 유지하는 학습 최적화 알고리즘이다. 이는 기울기 최적화 방법과 모멘텀 방법을 결합하여 최적화 성능을 향상하였다. 이 코드에서 신경망 학습 중, 손실 함수를 최소화하기 위해서 사용된다.

SparseCategoricalCrossentropy : 이는 Tensorflow에서 제공하는 classification task에서 사용할 수 있는 cross entropy loss 함수 중 하나이다.
SparseCategoricalCrossentropy는 Multi-class classification에서 주로 쓰이며, 클래스가 상호 배타적인 분류 문제에 사용되는 손실 함수이다. 각각의 인스턴스가 정확히 하나의 클래스에 속하는 경우에 적합하며, 훈련 데이터의 label(target)이 정수일 때 사용한다.

3.(e) Please replace MLP layers to convolutional layers of 3 × 3 support with the same channels, then retrain your network. How does the test accuracy change when the number of layers is equal to 1,2, and 3? Please analyze the result. Please save your model and submit it.

먼저, 기본 코드이다.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

fit 결과

```
Epoch 1/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.4996
- accuracy: 0.8252
Epoch 2/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3727
- accuracy: 0.8658
```

```
Epoch 3/10
1875/1875 [==============================] - 16s 8ms/step - loss:
0.3353 - accuracy: 0.8785
Epoch 4/10
1875/1875 [==============================] - 15s 8ms/step - loss:
0.3108 - accuracy: 0.8862
Epoch 5/10
1875/1875 [==============================] - 10s 5ms/step - loss:
0.2943 - accuracy: 0.8921
Epoch 6/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2812
- accuracy: 0.8962
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2682
- accuracy: 0.8996
Epoch 8/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2571
- accuracy: 0.9039
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2474
- accuracy: 0.9081
Epoch 10/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2379
- accuracy: 0.9107
<keras.src.callbacks.History at 0x7e153d3b3be0>

Accuracy 결과 :
313/313 - 1s - loss: 0.3286 - accuracy: 0.8817 - 642ms/epoch - 2ms/step

Test accuracy: 0.8816999793052673
```

conv2D 1층을 추가한 코드이다.

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(128, (3,3),
activation='relu',input_shape=(28, 28,1)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
]) 이렇게 change하였다.
```

fit 결과

```
Epoch 1/10
1875/1875 [==============================] - 12s 6ms/step - loss:
0.3634 - accuracy: 0.8703
Epoch 2/10
1875/1875 [==============================] - 11s 6ms/step - loss:
0.2290 - accuracy: 0.9143
Epoch 3/10
1875/1875 [==============================] - 11s 6ms/step - loss:
0.1685 - accuracy: 0.9380
```

```
Epoch 4/10
1875/1875 [==============================] - 11s 6ms/step - loss:
0.1229 - accuracy: 0.9542
Epoch 5/10
1875/1875 [==============================] - 12s 6ms/step - loss:
0.0878 - accuracy: 0.9674
Epoch 6/10
1875/1875 [==============================] - 11s 6ms/step - loss:
0.0655 - accuracy: 0.9768
Epoch 7/10
1875/1875 [==============================] - 11s 6ms/step - loss:
0.0459 - accuracy: 0.9834
Epoch 8/10
1875/1875 [==============================] - 11s 6ms/step - loss:
0.0355 - accuracy: 0.9875
Epoch 9/10
1875/1875 [==============================] - 11s 6ms/step - loss:
0.0288 - accuracy: 0.9899
Epoch 10/10
1875/1875 [==============================] - 11s 6ms/step - loss:
0.0216 - accuracy: 0.9926
<keras.src.callbacks.History at 0x7a918d835f00>

Accuracy 결과:

313/313 - 1s - loss: 0.5302 - accuracy: 0.9037 - 1s/epoch - 4ms/step

Test accuracy: 0.9036999940872192
```

이 상태로, 층을 하나 더 늘려보았다.

```
CODE CHANGE TO
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(128, (3,3),
activation='relu',input_shape=(28, 28,1)),
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

```
fit 결과
Epoch 1/10
1875/1875 [==============================] - 16s 8ms/step - loss:
0.3638 - accuracy: 0.8676
Epoch 2/10
1875/1875 [==============================] - 14s 8ms/step - loss:
0.2243 - accuracy: 0.9161
Epoch 3/10
```

```
1875/1875 [==============================] - 14s 8ms/step - loss:
0.1563 - accuracy: 0.9410
Epoch 4/10
1875/1875 [==============================] - 15s 8ms/step - loss:
0.0986 - accuracy: 0.9631
Epoch 5/10
1875/1875 [==============================] - 14s 8ms/step - loss:
0.0630 - accuracy: 0.9775
Epoch 6/10
1875/1875 [==============================] - 15s 8ms/step - loss:
0.0429 - accuracy: 0.9840
Epoch 7/10
1875/1875 [==============================] - 15s 8ms/step - loss:
0.0292 - accuracy: 0.9898
Epoch 8/10
1875/1875 [==============================] - 14s 8ms/step - loss:
0.0231 - accuracy: 0.9918
Epoch 9/10
1875/1875 [==============================] - 14s 8ms/step - loss:
0.0196 - accuracy: 0.9933
Epoch 10/10
1875/1875 [==============================] - 15s 8ms/step - loss:
0.0170 - accuracy: 0.9944
<keras.src.callbacks.History at 0x7a918d79d2a0>

Accuracy 결과:
313/313 - 1s - loss: 0.5711 - accuracy: 0.9169 - 1s/epoch - 3ms/step

Test accuracy: 0.9168999791145325
```

이 상태로, 층을 하나 더 늘려보아 3-layer를 만들었다.

code:

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(128, (3,3),
activation='relu',input_shape=(28, 28,1)),
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

fit 결과

```
Epoch 1/10
1875/1875 [==============================] - 19s 9ms/step - loss:
0.3739 - accuracy: 0.8647
Epoch 2/10
```

```
1875/1875 [==============================] - 18s 9ms/step - loss:
0.2270 - accuracy: 0.9156
Epoch 3/10
1875/1875 [==============================] - 18s 10ms/step - loss:
0.1633 - accuracy: 0.9399
Epoch 4/10
1875/1875 [==============================] - 18s 9ms/step - loss:
0.1138 - accuracy: 0.9575
Epoch 5/10
1875/1875 [==============================] - 19s 10ms/step - loss:
0.0745 - accuracy: 0.9718
Epoch 6/10
1875/1875 [==============================] - 18s 10ms/step - loss:
0.0489 - accuracy: 0.9821
Epoch 7/10
1875/1875 [==============================] - 18s 9ms/step - loss:
0.0382 - accuracy: 0.9862
Epoch 8/10
1875/1875 [==============================] - 18s 9ms/step - loss:
0.0276 - accuracy: 0.9898
Epoch 9/10
1875/1875 [==============================] - 18s 10ms/step - loss:
0.0291 - accuracy: 0.9904
Epoch 10/10
1875/1875 [==============================] - 18s 9ms/step - loss:
0.0189 - accuracy: 0.9936
<keras.src.callbacks.History at 0x7a9160320d60>

Accuracy 결과:
313/313 - 1s - loss: 0.6093 - accuracy: 0.9133 - 1s/epoch - 4ms/step

Test accuracy: 0.9132999777793884
```

결과 분석
기본 layer의 Test accuracy 값은 다음과 같다.
```
Test accuracy: 0.8816999793052673
```

각각의 conv layer의 개수마다, Test accuracy값을 불러오면, 다음과 같다.
```
layer 1 Test accuracy: 0.9036999940872192
layer 2 Test accuracy: 0.9168999791145325
layer 3 Test accuracy: 0.9132999777793884
```

각각의 layer마다 loss는 다음과 같다.
```
Base layer loss - 0.3286
layer 1 loss - 0.5302
layer 2 loss - 0.5711
layer 3 loss - 0.6093
```

먼저, conv를 적용하지 않았을 때보다, test accuracy가 전부 높다는 것은 확실히 볼 수 있다. layer가 증가할수록, Test accuracy가 증가하는 것이 일반적인 생각이다. 물론, layer가 무한히 증가한다고 accuracy가 계속 증가하는 것은 아니지만,

2개까지의 test 결과에서는 증가하는 것을 볼 수 있다. 너무 과다한 layer를 만들게 되면 계산 복잡성이 증가하거나, gradient 소실이 일어나는 등의 문제가 있지만, 2개 정도의 layer에서는 layer가 증가하게 되면, 모델의 용량이 증가하여서 더욱 복잡한 패턴이나 표현을 학습하는 데 있어서 사용할 수 있는 용량이 증가하므로, 일반적으로 Test accuracy 가 증가할 수밖에 없다. 그리고 이는, 위의 layer 증가에 따른 Test accuracy 증가 결과가 그 증거로 볼 수 있다는 것이다. 또한, 깊은 네트워크를 구성할 수 있게 되며, 더 많은 데이터에 대해 더 잘 일반화될 수 있기 때문도 있다. 그러나, 3개의 layer에서는 Test accuracy가 줄어들었다. 그 이유는,과도한 레이어들이 있게 되면, 과적합이 일어날 수도 있기 때문이다. 더욱 깊은 모델은 더 많은 매개변수를 사용하고, 훈련 데이터에 민감하게 반응한다. 레이어의 개수에 따라 훈련 데이터가 충분하지 않다면, 오히려 과적합되어서 accuracy가 감소할 수 있다. 그리고, 이전에 서술하였던, 계산 복잡성이나, gradient 소실 및 폭발이 영향을 미칠 수 있다.

결론적으로, layer의 개수를 증가시키면 accuracy가 증가하기는 하지만, 너무 과도한 layer의 개수는 test accuracy를 감소시킨다는 결론을 내릴 수 있다.

또한, 각각의 fit 결과를 살펴보면, epoch 가 증가할 때마다, 선형적으로 loss 값이 줄어든다는 것을 알 수 있고, loss 값은 layer 추가된 3가지 중에서는 1개 추가한 것이 가장 낮은 것을 알 수 있고, base와 많이 차이가 난다는 것을 알 수 있었다.

3.(f) Please retrain your 3 convolution layers model with different optimizers: adadelta, sgd, and rmsprop. Please analyze the results.

이 질문은, optimizer를 변경하였을 때, 결과값을 분석하는 것이다.

```
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

이 코드에서, optimizer를 'adam'에서, 각각 adadelta, sgd, rmsprop로 바꾸고 결과를 받는다.

(1) adadelta

```
Epoch 1/10
1875/1875 [==============================] - 21s 11ms/step - loss: 1.4553 - accuracy:
0.5279
Epoch 2/10
1875/1875 [==============================] - 18s 10ms/step - loss: 0.7398 - accuracy:
0.7339
Epoch 3/10
1875/1875 [==============================] - 18s 9ms/step - loss: 0.6562 - accuracy: 0.7639
Epoch 4/10
1875/1875 [==============================] - 18s 10ms/step - loss: 0.6135 - accuracy:
0.7810
Epoch 5/10
1875/1875 [==============================] - 18s 9ms/step - loss: 0.5840 - accuracy: 0.7930
Epoch 6/10
1875/1875 [==============================] - 18s 9ms/step - loss: 0.5616 - accuracy: 0.8012
Epoch 7/10
1875/1875 [==============================] - 18s 9ms/step - loss: 0.5430 - accuracy: 0.8095
Epoch 8/10
```

```
1875/1875 [==============================] - 18s 10ms/step - loss: 0.5273 - accuracy:
0.8160
Epoch 9/10
1875/1875 [==============================] - 18s 10ms/step - loss: 0.5145 - accuracy:
0.8223
Epoch 10/10
1875/1875 [==============================] - 18s 9ms/step - loss: 0.5025 - accuracy: 0.8256
<keras.src.callbacks.History at 0x7a91602ce8f0>

313/313 - 1s - loss: 0.5238 - accuracy: 0.8156 - 1s/epoch - 4ms/step

Test accuracy: 0.8155999779701233
```

(2) sgd

```
Epoch 1/10
1875/1875 [==============================] - 17s 9ms/step - loss: 0.6020 - accuracy: 0.7907
Epoch 2/10
1875/1875 [==============================] - 18s 9ms/step - loss: 0.4029 - accuracy: 0.8564
Epoch 3/10
1875/1875 [==============================] - 17s 9ms/step - loss: 0.3457 - accuracy: 0.8751
Epoch 4/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.3091 - accuracy: 0.8874
Epoch 5/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.2813 - accuracy: 0.8975
Epoch 6/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.2594 - accuracy: 0.9040
Epoch 7/10
1875/1875 [==============================] - 17s 9ms/step - loss: 0.2397 - accuracy: 0.9120
Epoch 8/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.2224 - accuracy: 0.9180
Epoch 9/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.2075 - accuracy: 0.9236
Epoch 10/10
1875/1875 [==============================] - 19s 10ms/step - loss: 0.1924 - accuracy:
0.9284
<keras.src.callbacks.History at 0x7a91602731f0>

313/313 - 1s - loss: 0.2789 - accuracy: 0.8998 - 1s/epoch - 4ms/step

Test accuracy: 0.8998000025749207
```

(3) rmsprop

```
Epoch 1/10
1875/1875 [==============================] - 18s 9ms/step - loss: 0.3762 - accuracy: 0.8636
Epoch 2/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.2279 - accuracy: 0.9179
Epoch 3/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.1742 - accuracy: 0.9367
Epoch 4/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.1298 - accuracy: 0.9537
Epoch 5/10
1875/1875 [==============================] - 17s 9ms/step - loss: 0.0965 - accuracy: 0.9665
Epoch 6/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.0702 - accuracy: 0.9758
Epoch 7/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.0534 - accuracy: 0.9822
Epoch 8/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.0431 - accuracy: 0.9866
Epoch 9/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.0361 - accuracy: 0.9891
Epoch 10/10
1875/1875 [==============================] - 17s 9ms/step - loss: 0.0306 - accuracy: 0.9913
```

```
<keras.src.callbacks.History at 0x7a914c6c50f0>
```

```
313/313 - 1s - loss: 0.7874 - accuracy: 0.9213 - 1s/epoch - 4ms/step

Test accuracy: 0.9212999939918518
```

각각의 Test accuracy 를 다시 써 보자.

```
adam - Test accuracy:     0.8816999793052673
adadelta - Test accuracy: 0.8155999779701233
sgd - Test accuracy:      0.8998000025749207
rmsprop - Test accuracy:  0.9212999939918518
```

그 결과, optimizer가 달라지면, Test accuracy도 달라진다는 것을 알 수 있다. 각각은 신경망의 학습 중 가중치를 업데이트하는 방식이 다르다. Adam은 적응형 학습률을 사용하여 학습률을 동적으로 조절하면서 가중치를 업데이트 하는 optimizer이고, adadelta는 이동 평균을 활용하고, sgd는 가장 기본적인 옵티마이저로, 기울기의 반대 방향으로 가중치를 업데이트하고, rmsprop는 각 매개변수에 대한 학습률을 조정하여 가중치를 업데이트 한다. 이 모델을 구현하는 데 있어서, 가장 효과적인 Test accuracy를 보여주는 것은 rmsprop이다. 가장 저조한 optimizer는 adadelta이다.

4 Sentiment analysis by recurrent neural networks (RNN)

4.(a) Explain the meaning of the output values.

먼저, 첫 줄부터 imdb를 import하는데, keras.datasets 안에 있는 imdb는 영화를 본 사람들의 리뷰가 들어있다. 그것을 불러오는데, nice 한 것이냐, 그렇지 않냐에 대해서 라벨을 가지고 있는 데이터셋이기 때문에, 이를 활용한 결과값은, Sentiment Analysis(감정 분석)에서의 0~1 사이의 확률값으로 나오고, 이는, nice한 것이냐, 그렇지 않냐의 확률을 나타내는 것이다.

뒤에 있는 4.(d)~4.(h)를 비교설명하기 위해 fit 결과와 test dataset의 model accuracy를 base로 돌려 보자.

model summary

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 500, 32)	160000
simple_rnn (SimpleRNN)	(None, 100)	13300
dense (Dense)	(None, 1)	101

```
=================================================================
Total params: 173401 (677.35 KB)
Trainable params: 173401 (677.35 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

fit 결과

```
Epoch 1/3
391/391 [==============================] - 83s 204ms/step - loss:
0.6780 - accuracy: 0.5610
Epoch 2/3
391/391 [==============================] - 81s 208ms/step - loss:
0.5348 - accuracy: 0.7356
Epoch 3/3
391/391 [==============================] - 82s 209ms/step - loss:
0.6018 - accuracy: 0.6800
<keras.callbacks.History at 0x7f773dbba430>
```

model accuracy:

```
Model accuracy on the test dataset: 64.70%
```

4.(b) Please explain the meaning of max words, embedding vector length, and input length in 'Embedding(max words, embedding vector length, input length=max review length)'.

max_words = 5000 이 쓰이는데, 여기서 max_words는 모델이 처리할 최대 단어 수를 제한하기 위해 쓰인다. 즉, 5000개의 단어까지가 embedding을 실시하게 된다. 즉, 최대 단어를 뜻한다. embedding_vector_length 는 각 단어를 vector로 임베딩하는데, 그것의 길이를 정하는 것이다. 이 코드에서의 값은 32를 부여받는데, 각 단어를 32 길이의 vector로 표현하게 되는 것이다. input_length는 이 코드에서 500으로 받게 되는데, 이 뜻은, 입력 시퀀스의 길이를 전부 동일하게 500으로 맞추겠다는 것이다. hey 라는 단어가 들어와도, 499개의 값이 비어있는 것들로 채워 넣어서, 입력 시퀀스를 500으로 맞추게 되는 것이다. 500이 넘어가게 되면, 500까지로 줄어들게 만들 것이다.

4.(c) Please explain the meaning of in units, activation, and return sequences in SimpleRNN(units=100, activation='tanh', return sequences=True)

먼저, units은 RNN 레이어에 그만큼의 뉴런 or 유닛이 있다는 것인데, 이는 dimension을 바꾸어 주는 것이다. input data가 RNN을 통과하게 될 때, embedding의 dimension에서 RNN 유닛만큼의 dimension으로 바뀌게 된다. 먼저, return sequences를 설명하자면, dimension으로 바뀌고 나서도 RNN 레이어가 출력으로 전체 시퀀스를 반환할지 정하는 것이다. 이를 True로 해 놓으면, 전체 시퀀스가 반환되고, false이면, 마지막 뉴런만 반환하게 된다. activation은 , RNN 뉴런 각각에 적용된 활성화 함수이다. y = $Wy \times f(WxXt + WhHt-1 + bias)$ 이 연산에서의 f에 해당한다. 위의 코드에서는 이것을 tanh로 지정하겠다는 것이다.

4.(d)Please change epoch of 3 to the epoch of 4, then retrain the model with one RNN(100) layer. How does the test accuracy change? and Why does the test accuracy increase (or decrease)? Please analyze the result. Please save your model and submit it.

RNN(100) 하나 그대로로, epochs를 4로만 바꾸어서 다시 돌린다.

```
# Define how long the embedding vector will be
embedding_vector_length = 32

# Define the layers in the model
model = Sequential()
model.add(Embedding(max_words, embedding_vector_length,
input_length=max_review_length))
#model.add(SimpleRNN(100, return_sequences=True))
model.add(SimpleRNN(100))
#model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))

print("Model created.")
```

```
model.fit(X_train, y_train, epochs=4, batch_size=64)
```

결과값은 다음과 같다.

model summary(변화없음)

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 500, 32)           160000

 simple_rnn (SimpleRNN)      (None, 100)               13300

 dense (Dense)               (None, 1)                 101

=================================================================
Total params: 173401 (677.35 KB)
Trainable params: 173401 (677.35 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

fit 결과:

```
Epoch 1/4
391/391 [==============================] - 75s 187ms/step - loss: 0.6637 - accuracy: 0.5760
Epoch 2/4
391/391 [==============================] - 68s 173ms/step - loss: 0.6057 - accuracy: 0.6624
Epoch 3/4
391/391 [==============================] - 67s 172ms/step - loss: 0.5730 - accuracy: 0.6956
Epoch 4/4
391/391 [==============================] - 67s 172ms/step - loss: 0.5167 - accuracy: 0.7482
```

```
<keras.src.callbacks.History at 0x7f170125cd60>
```

Model accuracy on the test dataset

```
Model accuracy on the test dataset: 67.37%
```

Analyze

일반적으로 Epoch가 늘어나면 모델이 훈련 데이터에 더 많은 학습을 할 수 있기 때문에, 더욱더 잘 파악하여서 accuracy가 증가한다. 이는 그 예시이다. 4로 증가시켰을 때, 기본 값인 64.70 보다 높아진 것을 볼 수 있다.

4.(e) Please change epoch of 3 to the epoch of 5, then retrain the model with two RNN(100) layers. How does the test accuracy change? and Why does the test accuracy increase (or decrease)? Please analyze the result. Please save your model and submit it.

RNN은 100짜리 2개를 사용하고, epochs를 5로 변경한다.

```
# Define how long the embedding vector will be
embedding_vector_length = 32

# Define the layers in the model
model = Sequential()
model.add(Embedding(max_words, embedding_vector_length,
input_length=max_review_length))
model.add(SimpleRNN(100, return_sequences=True))
model.add(SimpleRNN(100))
#model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))

print("Model created.")
# Fit the model to the training data
model.fit(X_train, y_train, epochs=5, batch_size=64)
```

결과

model summary

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 500, 32)           160000

 simple_rnn_3 (SimpleRNN)    (None, 500, 100)          13300

 simple_rnn_4 (SimpleRNN)    (None, 100)               20100

 dense_2 (Dense)             (None, 1)                 101

=================================================================
Total params: 193501 (755.86 KB)
```

```
Trainable params: 193501 (755.86 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

fit 결과

```
Epoch 1/5
391/391 [==============================] - 171s 423ms/step - loss: 0.7208 - accuracy:
0.5156
Epoch 2/5
391/391 [==============================] - 156s 398ms/step - loss: 0.6242 - accuracy:
0.6323
Epoch 3/5
391/391 [==============================] - 155s 397ms/step - loss: 0.5233 - accuracy:
0.7456
Epoch 4/5
391/391 [==============================] - 156s 399ms/step - loss: 0.5569 - accuracy:
0.7102
Epoch 5/5
391/391 [==============================] - 154s 393ms/step - loss: 0.6484 - accuracy:
0.6183
<keras.src.callbacks.History at 0x78f3b8b86c20>
```

Model accuracy on the test dataset

```
Model accuracy on the test dataset: 63.78%
```

Analyze

layer 와 epoch가 둘 다 늘었는데, accuracy는 오히려 줄어든 값을 보였다. 이러한 이유를 살펴보자. 먼저, loss 의 출발값이 이전 것들보다 높았다는 점이 있다. 이는 과적합이 발생하였다고 볼 수 있다. 에포크와 layer 를 증가시켰을 때, 훈련 데이터에 대해 더 잘 적응할 수 있지만, 훈련 데이터에 민감해지면서 과적합이 발생되며 일반화 능력이 떨어진 것으로 볼 수 있다는 것이다. 또한, Epoch가 3까지는 줄어들다가, 그 이후에는 증가하는 것으로 볼 수 있다. 이는, Epoch가 오히려 너무 적은 값이라 그럴 수 있다. loss 그래프는 어느 값으로 수렴하게 되는데, 아직 수렴하지 않거나, 과적합으로 인한 발산하는 부분일 수 있을 것으로 추정된다.

4.(f) Please replace the model with one RNN(100) layer to that with one RNN(50) layer, then retrain your network with the epoch of 3. How does the test accuracy change? and Why does the test accuracy increase (or decrease)? Please analyze the result. Please save your model and submit it.

RNN을 50 한 개로 하고, epoch를 3으로 지정한다. 위와 코드 짜는 방법이 같고, 숫자만 바뀐 방식이므로 코드는 생략하겠다.

model Summary

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, 500, 32)           160000

 simple_rnn_5 (SimpleRNN)    (None, 50)                4150

 dense_3 (Dense)             (None, 1)                 51

=================================================================
```

```
Total params: 164201 (641.41 KB)
Trainable params: 164201 (641.41 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

fit 결과

```
Epoch 1/3
391/391 [==============================] - 211s 535ms/step - loss: 0.5864 - accuracy:
0.6683
Epoch 2/3
391/391 [==============================] - 194s 496ms/step - loss: 0.4470 - accuracy:
0.7885
Epoch 3/3
391/391 [==============================] - 191s 490ms/step - loss: 0.5463 - accuracy:
0.7234
<keras.src.callbacks.History at 0x7e6b0d6f5000>
```

Model accuracy on the test dataset

```
Model accuracy on the test dataset: 72.13%
```

Analyze

layer의 unit 수가 줄었지만, 기본 base의 코드보다, 정확도가 눈에 띄게 증가하였다. epoch가 4, 5 일 때의 accuracy보다도 높은 값이다. 이 이유로는, unit의 적합성을 들 수 있다. 이 dataset과 model에서 RNN 100 unit보다 50 unit이 더 좋은 결과를 내는 것이다.

4.(g) Please replace the model with one RNN(50) layer to that with two RNN(50) layers, then retrain your network. How does the test accuracy change? and Why does the test accuracy increase (or decrease)? Please analyze the result. Please save your model and submit it.

RNN 50을 두 개로 늘린다.

model Summary

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_4 (Embedding)     (None, 500, 32)           160000

 simple_rnn_6 (SimpleRNN)    (None, 500, 50)           4150

 simple_rnn_7 (SimpleRNN)    (None, 50)                5050

 dense_4 (Dense)             (None, 1)                 51

=================================================================
Total params: 169251 (661.14 KB)
Trainable params: 169251 (661.14 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

fit 결과

```
Epoch 1/3
```

```
391/391 [==============================] - 396s 1s/step - loss: 0.5808
- accuracy: 0.6627
Epoch 2/3
391/391 [==============================] - 379s 971ms/step - loss:
0.5783 - accuracy: 0.7011
Epoch 3/3
391/391 [==============================] - 373s 952ms/step - loss:
0.5139 - accuracy: 0.7457
<keras.src.callbacks.History at 0x7e6b152bbac0>
```

Model accuracy on the test dataset

```
Model accuracy on the test dataset: 75.54%
```

Analyze

이전 과정에서 layer를 하나 더 추가하였고, 그대로, accuracy도 조금 증가하였다. 다른 것이 같은 조건에서, layer를 추가하면, 각 층이 이전 층의 출력을 입력으로 받아 더욱더 깊은 학습을 할 수 있게 되고, 그에 따라 accuracy가 증가하는 것은 당연한 것이다.

4.(h) Please replace the model with one RNN(100) layer to that with one LSTM(100) layer with the epoch of 3, then retrain your network. How does the test accuracy change? and Why does the test accuracy increase (or decrease)? Please analyze the result. Please save your model and submit it.

기존 RNN을 LSTM(100)으로 지정한다.

model Summary

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_5 (Embedding)     (None, 500, 32)           160000

 lstm (LSTM)                 (None, 100)               53200

 dense_5 (Dense)             (None, 1)                 101

=================================================================
Total params: 213301 (833.21 KB)
Trainable params: 213301 (833.21 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

fit 결과

```
Epoch 1/3
391/391 [==============================] - 41s 96ms/step - loss: 0.5191
- accuracy: 0.7289
Epoch 2/3
```

```
391/391 [==============================] - 18s 45ms/step - loss: 0.3004
- accuracy: 0.8788
Epoch 3/3
391/391 [==============================] - 14s 35ms/step - loss: 0.2667
- accuracy: 0.8963
<keras.src.callbacks.History at 0x7e6b134f5390>
```

Model accuracy on the test dataset

```
Model accuracy on the test dataset: 86.78%
```

Analyze

이 결과는, 이 때까지 있었던 accuracy 중 가장 큰 값을 지니고 있다. 이는 기본 Single RNN 대신 LSTM 을 사용하였기 때문이다. 우리가 사용하는 데이터는 sequence 길이가 매우 길었다. 기존의 RNN은 길이가 긴 데이터들을 처리하기 어려워하는 장기 의존성의 문제가 있다. 그래서 손실되는 정보들이 있었고, 그 단점을 보완하는 LSTM을 사용하였기에 accuracy가 눈에 띄게 증가한 것을 알 수 있다. LSTM은 애초에 장기 의존성 문제를 해결하기 위해 Gate 구조를 사용하고 있기 때문에, 우리의 경우에 최적의 accuracy를 낼 수 있었던 것이다.