1조 다음은 2조

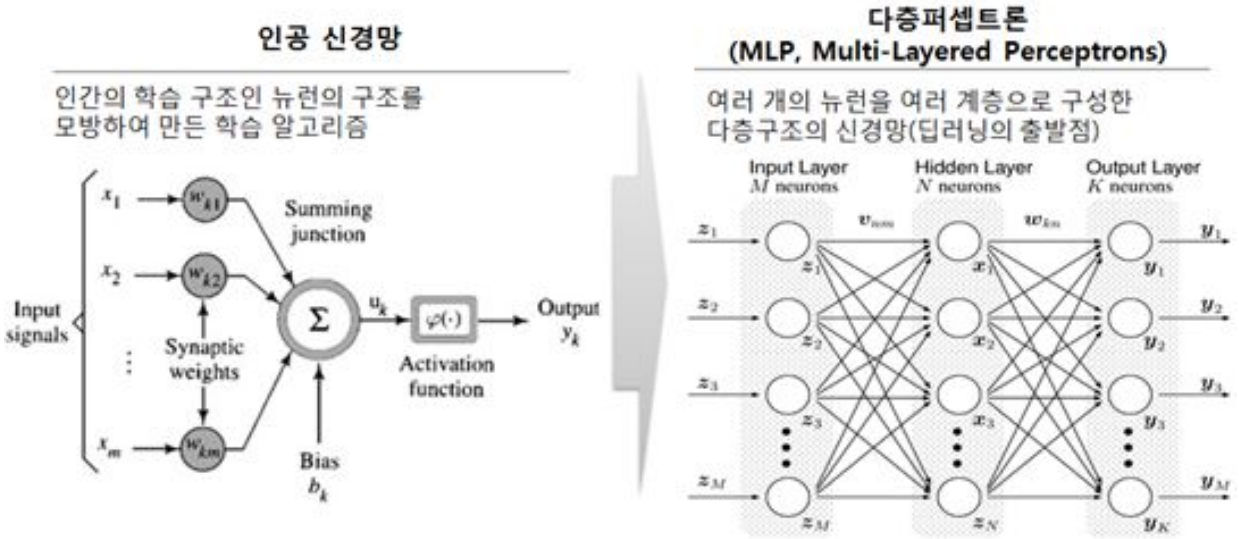
목차

3-1 신경망의 구조

1.1 층

1.2 모델(=네트워크)

1.3 손실 함수 와 옵티마이저



3-1 신경망의 구조

3.1.1 층

: 데이터 처리 <- 가중치 포함

1) 데이터 처리방법

- 2차원 : 보통 밀집 연결 층(dense connected layer) 으로 처리 (ex.케라스 에서는 DENSE)
- 3차원 : 순환층(recurrent layer) (ex. LSTM)
- 4차원 : 2D 합성곱(convolution layer) (ex. Conv2D)

3.1.2 모델(=네트워크)

1) 주요 네트워크 구조

- 가지(branch) 가 2개
- 출력이 여러 개
- 인셉션 블록(inception)

3-1 신경망의 구조

3.1.3 손실 함수 와 옵티마이저

1) 손실함수(목적 함수) : 정확도를 나타낸다.

2) 옵티마이저 : 네트워크의 파라미터의 업데이트 방법(한 종류의 SGD(확률적 경사 하강법) 시행)

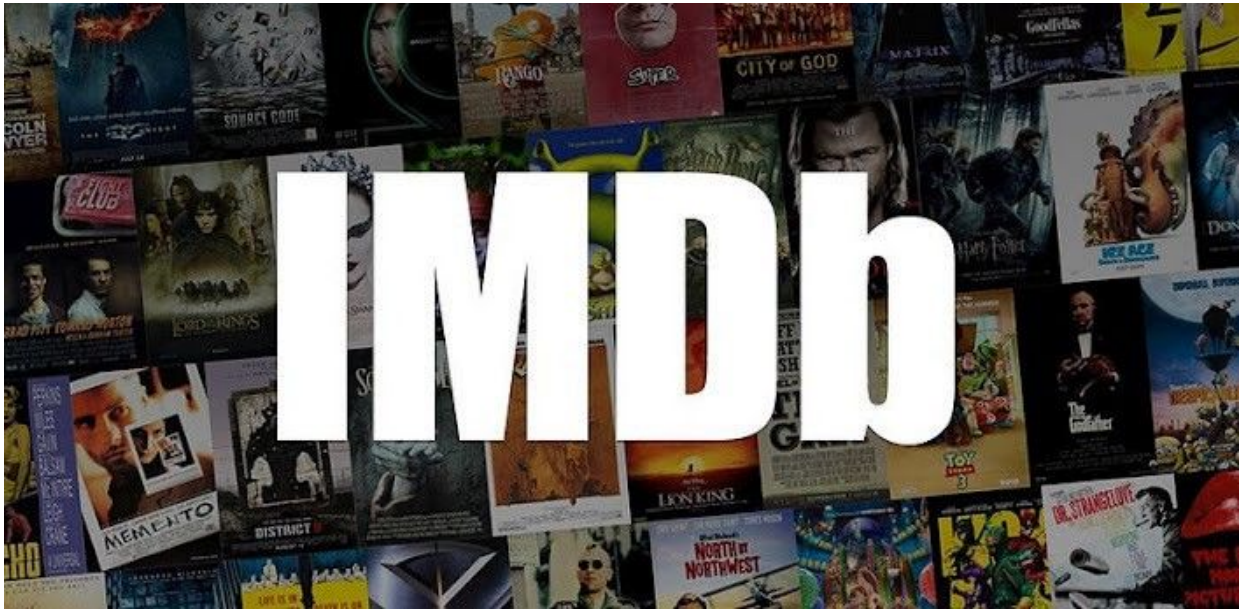
3) 손실함수 선택 방법

- 분류(2 클래스) : 이진 크로스엔트로피(binary crossentropy)
- 분류(3 클래스) : 범주형 크로스 엔트로피
- 회귀 : 평균제곱 오차
- 시퀀스 학습 : CTC(connection temporal Classification)

3-4 영화 리뷰 분류

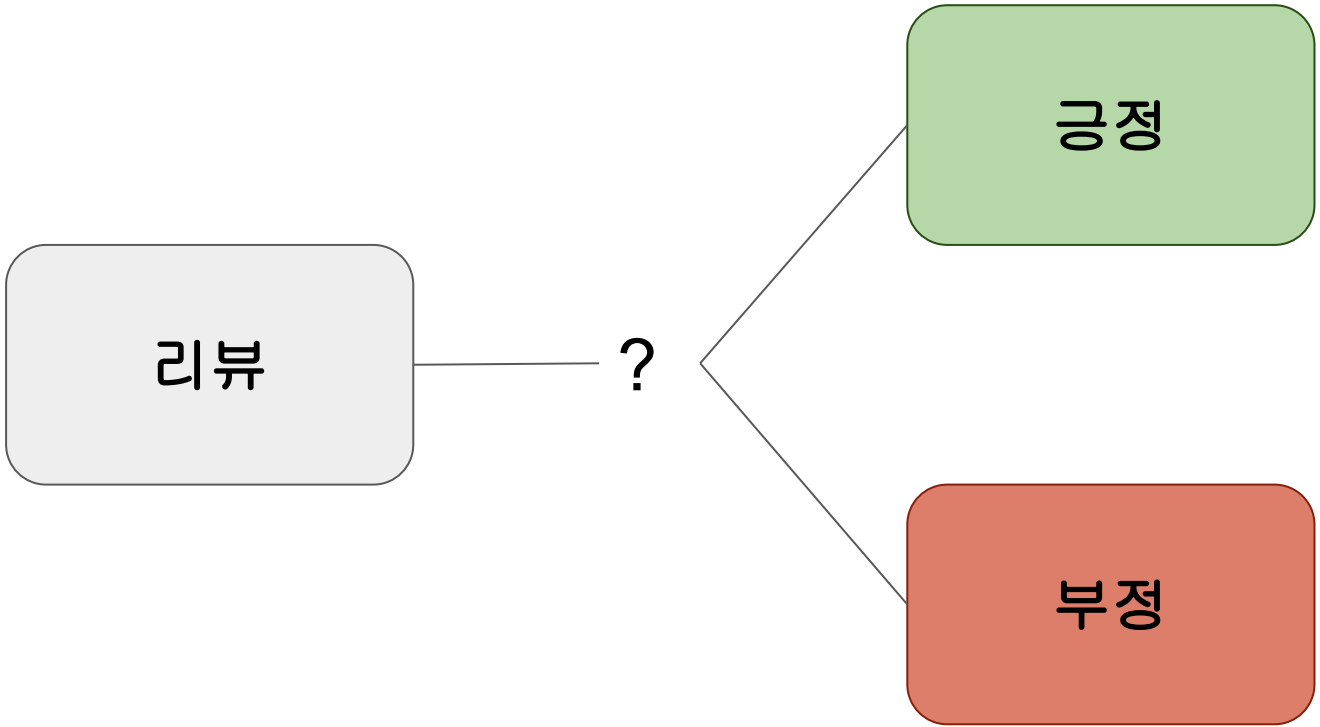
이진 분류 문제

IMDB 데이터셋 소개



- 부정 50%, 긍정 50%
- 훈련 데이터, 테스트 데이터
- 각 리뷰가 숫자 시퀀스로 변환

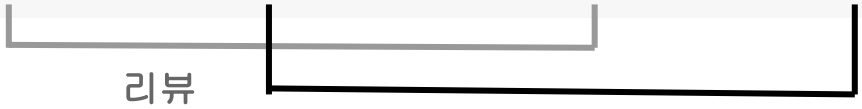
IMDB 데이터셋 소개



데이터 로드

[코드]

```
#imdb 불러오기
from keras.datasets import imdb

#자주 나타나는 단어 1만개만 사용
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
```



라벨(긍정 1, 부정 0)

[결과]

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17465344/17464789 [==============================] - 1s 0us/step
```

데이터 로드

this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert ? is an amazing actor and now the same being director ? father came from the same scottish island as myself so i loved the fact there was a real connection with this film the witty remarks throughout the film were great it was just brilliant so much that i bought the film as soon as it was released for ? and would recommend it to everyone to watch and the fly fishing was amazing really cried at the end it was so sad and you know what they say if you cry at a film it must have been good and this definitely was also ? to the two little boy's that played the ? of norman and paul they were just brilliant children are often left out of the ? list i think because the stars that play them all grown up are such a big profile for the whole film but these children are amazing and should be praised for what they have done don't you think the whole story was so lovely because it was true and was someone's life after all that was shared with us all

네.. 그렇대요...
다 읽으셨죠? 하하하ㅏ하

데이터 준비

[코드]

```
import numpy as np

#one-hot encoding
def vectorize_sequences(sequences, dimension=10000):
    #모든 원소 0으로 초기화
    results = np.zeros((len(sequences),dimension))
    for i, sequence in enumerate(sequences):#enumerate -> sequences 열거
        #특정 단어 1로 만들기
        results[i, sequence] = 1 #랜덤으로 i번째를 1로 지정
    return results

#벡터 변환
x_train = vectorize_sequences(train_data)
y_test = vectorize_sequences(test_data)

#레이블 벡터 변환
x_label = np.asarray(train_labels).astype('float32')
y_label = np.asarray(test_labels).astype('float32')
```

모델 정의 및 컴파일

[코드]

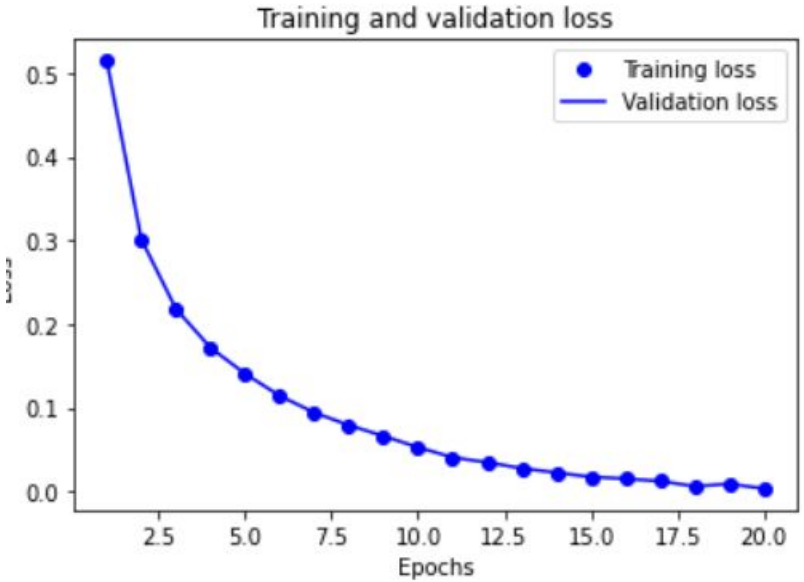
```
#모델 정의
from keras import models
from keras import layers

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'))

#모델 컴파일
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy']
```

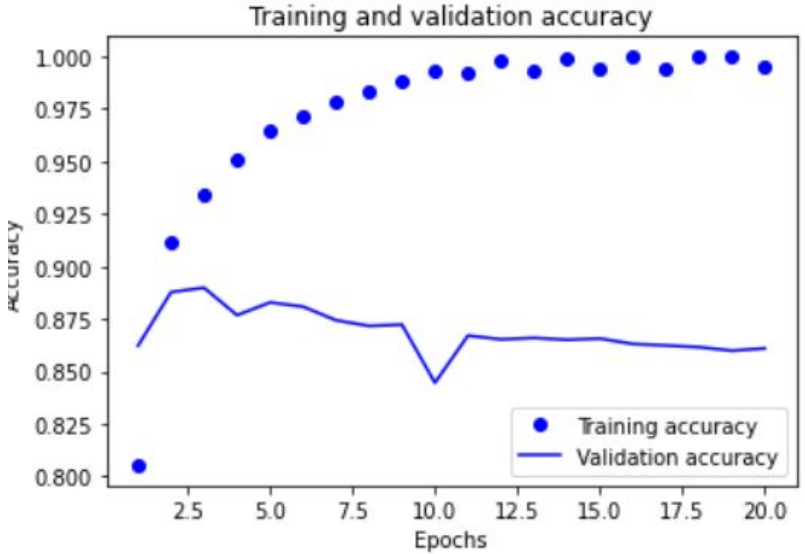
모델 훈련

[코드]

```
#데이터 뽑아오기
partial_x_train = x_train[10000:]
partial_y_train = x_label[10000:]
x_val = x_train[:10000]
y_val = x_label[:10000]
#모델 컴파일
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

#모델 훈련
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

모델 훈련 - 정확도 약 86%


Training and validation loss
Training loss
Validation loss
0.5
0.4
0.3
0.2
0.1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs


Training and validation accuracy
Training accuracy
Validation accuracy
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs

추가 실험

1. **은닉층 1개 또는 3개 사용해보기**
    - 1개 - 정확도 약 87%
    - 3개 - 정확도 86%
2. **mse함수 사용해보기**
    - 정확도 약 54%
3. **relu대신 tanh써보기**
    - 정확도 약 85%

3-5 뉴스 기사 분류

다중 분류 문제

개요

앞선, 영화 리뷰 문제(이진 분류 문제)

=> 데이터를 벡터형태로 바꾼 입력을 2개의 클래스(긍정 or 부정) 로 분류할 것인가 ?

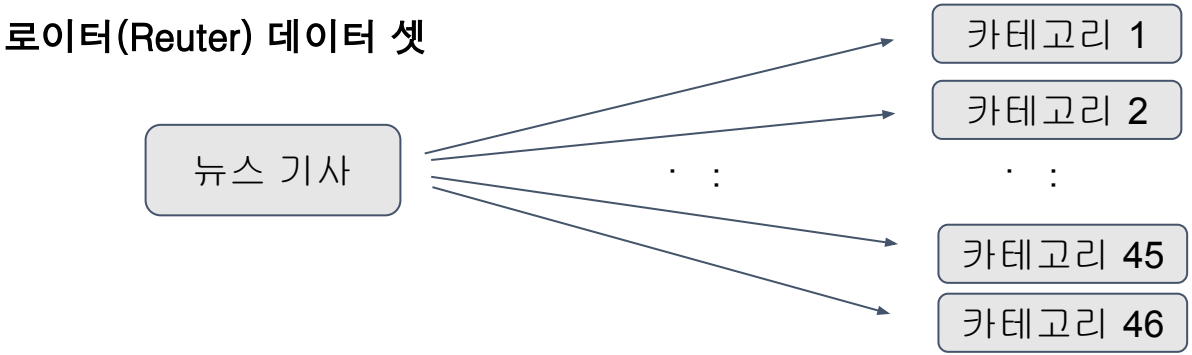
이번엔, 로이터(Reuter) 뉴스 토픽 데이터 => 46개의 클래스(= 즉, 마지막 출력 Layer = 46개)

분류 해야할 클래스가 많다. => 다중분류(Multi-Class Classification), 범주형 데이터

입력 데이터 포인트 → 하나의 범주 (이번 예제에서의 '하나의 뉴스 토픽') = 단일 레이블(Single-label) 다중 분류

입력 데이터 포인트 → 여러 개의 범주 = 다중 레이블(multi-label) 다중 분류

데이터 셋 소개



앞선, 영화 리뷰 데이터 IMDB와 숫자이미지 데이터 MNIST와 더불어,
로이터 데이터 또한 => 케라스에 포함.

[참고] 원본 로이터 데이터 셋 = 135개 토픽(카테고리)
이번 예제에선, 그 중 샘플이 많은 것을 뽑은 것 => “금융 관련 카테고리”

데이터 준비

1. 데이터 셋 로드

```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) = reuters.load_data(num_words=10000)
```

- num_words=10000 매개변수 => 가장 자주 등장하는 단어 1만 개로 제한

2. 데이터 인코딩

```
# Ver 1. 레이블 리스트 => 정수 텐서로 변환
import numpy as np

def vectorize_sequences(sequences, dimension=10000):  # 기존 데이터 -> 벡터 데이터로 변환
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data) # 훈련 데이터 벡터 변환
x_test = vectorize_sequences(test_data) # 테스트 데이터 벡터 변환
```

```
# Ver 2. 원-핫 인코딩(One-Hot encoding)
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

one_hot_train_labels = to_one_hot(train_labels) # 훈련 레이블 벡터 변환
one_hot_test_labels = to_one_hot(test_labels) # 테스트 레이블 벡터 변환
```

원-핫 인코딩 = 범주형 데이터에 사용
(자세한 개념은 6장 학습 시 소개)
[참고] 케라스에서 내장 함수로 제공

```
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

to_categorical 메소드

[참고]
load_data() 함수에서 test_split 매개변수로 테스트 데이터의 크기를 조절 할 수 있음
⇒ 기본값 0.2(전체 데이터 중 20% 테스트 데이터)

```
len(train_data)
```

```
8982
```

```
len(test_data)
```

```
2246
```

[번외] 자료(데이터)의 형태

[참고]
데이터 형태

1. **범주형 데이터(Categorical data)** : 몇 개의 범주로 나누어 진 자료를 말함
   - 명목형 자료(nominal data) = 아무런 순서가 없고, 단순히 '분류 ' 를 목적
   Ex) 성별(남/여), 성공여부(성공/실패), 혈액형(A/B/O/AB) 느낌
   - 순서형 자료(Ordinal data) : 명목형 자료이면서, '순서 ' 에 의미가 있는 자료를 말함
   Ex) 효과(없음(0)/조금있음(1~5~8)/매우있음(9)) 느낌
2. **수치형 자료(Numerical data)**
   - 이산형 자료(Discrete data) : 이산적인 값을 갖는 자료를 말함
   Ex) 일정기간 동안의 발생횟수, 출산횟수 느낌
   - 연속형 자료(Continuous data) : 연속적인 값을 갖는 자료를 말함
   Ex) 신장, 체중, 혈압 데이터 느낌

- 자료의 형태에 따라, 분석 방법이 달라짐(우리 입장에선, 모델 구성의 영향을 준다 생각함)

모델 구성

앞선, 영화 리뷰 분류(이진 분류)문제와 비슷하게 => 짧은 텍스트 분류
단, 출력 클래스가 2개 => 46개로 늘어남 (제약 사항)

모델에 Dense 층을 쌓을 것임. (단, 각 층은 이전 층의 출력에서 제공한 정보만 사용 가능하다고 했다.)
=> 이전 층에서 분류 문제에 필요한 일부 정보를 누락하면 -> 그 다음 층에서 이를 복원할 방법 X
=> 정보의 병목(Information Bottleneck) 현상

이런 병목현상을 방지하기 위해, 이 문제로 돌아와서 다시 생각해봤을 때, 앞선 이진 분류 문제에선
각 Layer에 16개의 은닉유닛(=16차원 공간)을 형성했던 걸,
이번 예제(뉴스 기사 토픽분류)에선, 46개(46개의 토픽)의 클래스를 구분하기 위해서
이보다 많은, 은닉유닛을 설정하는 것이 정보의 병목현상을 예방할 수 있음 (이 책에선 64개 유닛사용)

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))  # 각 층마다, 64개의 은닉유닛
model.add(layers.Dense(64, activation='relu')) # 64개 은닉유닛
model.add(layers.Dense(46, activation='softmax')) # 출력 층에서 46개의 클래스(토픽) 분류
```

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

[참고]
- 마지막 출력 층에, 활성화함수 = softmax를 사용 (앞선, 이진분류에선 -> simoid 활성화함수 사용했음)
  -> 각 입력 샘플마다 46개의 출력 클래스에 대한 확률 분포 출력
  = 46차원 출력 벡터 생성, output[i] = 어떤 샘플이 클래스 i에 속할 확률, 46개 전체 더하면 1 (전체 확률)

훈련 검증

훈련 데이터에서 1000개의 샘플 추출

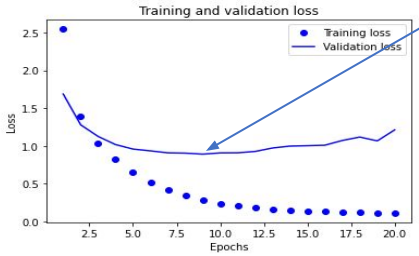
```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

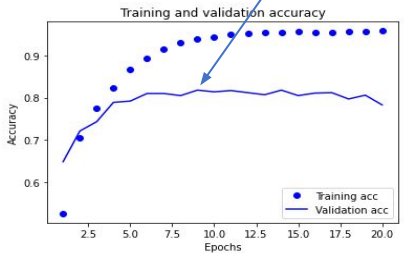
20번 epoch, 모델 훈련

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

훈련,검증 손실,정확도 시각화
(matplotlib사용, 코드 생략)

```
Epoch 1/20
16/16 [==============================] - 1s 53ms/step - loss: 2.5471 - accuracy: 0.5248 - val_loss: 1.6873 - val_accuracy: 0.6480
Epoch 2/20
16/16 [==============================] - 1s 43ms/step - loss: 1.3878 - accuracy: 0.7050 - val_loss: 1.2796 - val_accuracy: 0.7210
Epoch 3/20
16/16 [==============================] - 1s 41ms/step - loss: 1.0369 - accuracy: 0.7760 - val_loss: 1.1257 - val_accuracy: 0.7430
Epoch 4/20
16/16 [==============================] - 1s 41ms/step - loss: 0.8209 - accuracy: 0.8236 - val_loss: 1.0177 - val_accuracy: 0.7890
Epoch 5/20
16/16 [==============================] - 1s 41ms/step - loss: 0.6477 - accuracy: 0.8661 - val_loss: 0.9586 - val_accuracy: 0.7920
Epoch 6/20
16/16 [==============================] - 1s 41ms/step - loss: 0.5202 - accuracy: 0.8931 - val_loss: 0.9350 - val_accuracy: 0.8100
Epoch 7/20
16/16 [==============================] - 1s 41ms/step - loss: 0.4152 - accuracy: 0.9146 - val_loss: 0.9086 - val_accuracy: 0.8100
Epoch 8/20
16/16 [==============================] - 1s 41ms/step - loss: 0.3380 - accuracy: 0.9297 - val_loss: 0.9049 - val_accuracy: 0.8050
Epoch 9/20
16/16 [==============================] - 1s 41ms/step - loss: 0.2798 - accuracy: 0.9386 - val_loss: 0.8913 - val_accuracy: 0.8180
Epoch 10/20
16/16 [==============================] - 1s 41ms/step - loss: 0.2349 - accuracy: 0.9430 - val_loss: 0.9079 - val_accuracy: 0.8140
Epoch 11/20
16/16 [==============================] - 1s 43ms/step - loss: 0.2043 - accuracy: 0.9504 - val_loss: 0.9093 - val_accuracy: 0.8170
Epoch 12/20
16/16 [==============================] - 1s 41ms/step - loss: 0.1808 - accuracy: 0.9536 - val_loss: 0.9272 - val_accuracy: 0.8120
Epoch 13/20
16/16 [==============================] - 1s 41ms/step - loss: 0.1606 - accuracy: 0.9553 - val_loss: 0.9729 - val_accuracy: 0.8070
Epoch 14/20
16/16 [==============================] - 1s 42ms/step - loss: 0.1517 - accuracy: 0.9550 - val_loss: 0.9980 - val_accuracy: 0.8180
Epoch 15/20
16/16 [==============================] - 1s 42ms/step - loss: 0.1327 - accuracy: 0.9574 - val_loss: 1.0033 - val_accuracy: 0.8050
Epoch 16/20
16/16 [==============================] - 1s 41ms/step - loss: 0.1360 - accuracy: 0.9551 - val_loss: 1.0096 - val_accuracy: 0.8110
Epoch 17/20
16/16 [==============================] - 1s 43ms/step - loss: 0.1209 - accuracy: 0.9553 - val_loss: 1.0727 - val_accuracy: 0.8120
Epoch 18/20
16/16 [==============================] - 1s 42ms/step - loss: 0.1184 - accuracy: 0.9579 - val_loss: 1.1179 - val_accuracy: 0.7970
Epoch 19/20
16/16 [==============================] - 1s 41ms/step - loss: 0.1156 - accuracy: 0.9570 - val_loss: 1.0667 - val_accuracy: 0.8060
Epoch 20/20
16/16 [==============================] - 1s 41ms/step - loss: 0.1110 - accuracy: 0.9588 - val_loss: 1.2127 - val_accuracy: 0.7830
```





9번째 epoch이후 오버피팅 시작

새로운 모델 훈련,평가(9번 epoch훈련)

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=9,          # epoch 9번만 시행
          batch_size=512,
          validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)    # 모델 평가
```

```
Epoch 1/9
16/16 [==============================] - 1s 52ms/step - loss: 2.6428 - accuracy: 0.5063 - val_loss: 1.7546 - val_accuracy: 0.6230
Epoch 2/9
16/16 [==============================] - 1s 43ms/step - loss: 1.4542 - accuracy: 0.6973 - val_loss: 1.3369 - val_accuracy: 0.7090
Epoch 3/9
16/16 [==============================] - 1s 42ms/step - loss: 1.0825 - accuracy: 0.7695 - val_loss: 1.1493 - val_accuracy: 0.7450
Epoch 4/9
16/16 [==============================] - 1s 42ms/step - loss: 0.8445 - accuracy: 0.8220 - val_loss: 1.0315 - val_accuracy: 0.7830
Epoch 5/9
16/16 [==============================] - 1s 42ms/step - loss: 0.6676 - accuracy: 0.8569 - val_loss: 0.9534 - val_accuracy: 0.8040
Epoch 6/9
16/16 [==============================] - 1s 41ms/step - loss: 0.5355 - accuracy: 0.8890 - val_loss: 0.9233 - val_accuracy: 0.8140
Epoch 7/9
16/16 [==============================] - 1s 42ms/step - loss: 0.4288 - accuracy: 0.9092 - val_loss: 0.8891 - val_accuracy: 0.8230
Epoch 8/9
16/16 [==============================] - 1s 41ms/step - loss: 0.3465 - accuracy: 0.9265 - val_loss: 0.9005 - val_accuracy: 0.8160
Epoch 9/9
16/16 [==============================] - 1s 41ms/step - loss: 0.2882 - accuracy: 0.9367 - val_loss: 0.8986 - val_accuracy: 0.8160
```

```
results  # 모델 평가
```

```
[0.9978088140487671, 0.7934104800224304]
```

대략 79% 정확도

새로운 데이터 예측

```
predictions = model.predict(x_test)
# predict(x_test) = load_data에서 0.2의 비율로 나뉜 2246개의 test데이터, 46개 토픽에 대한 확률 분포 반환
```

```
predictions[0].shape # 각 항목의 길이는 46(46개의 토픽)
```

```
(46,)
```

```
predictions[:].shape # 2246개의 test데이터, 각 항목의 길이는 46(46개의 토픽)
```

```
(2246, 46)
```

```
np.sum(predictions[0]) # 각 항목에 대해 원소의 합은 1
```

```
1.0000001
```

```
np.argmax(predictions[0]) # 가장 큰 값이 예측클래스, 즉, 1번째 테스트데이터는 3번 클래스(토픽)이라 생각
```

```
3
```

[참고]

상기합니다

- 모델 컴파일시, 손실함수 = categorical_crossentropy 사용
=> 두 확률 분포 사이의 거리를 측정
(즉, 여기선 네트워크가 출력한 확률 분포와 진짜 레이블의 분포 사이의 거리,
이 거리를 최소화해야 진짜 레이블에 가장 가까운 출력을 하도록 모델 훈련하게 됨.)
(앞선 장에서, 분류문제문제에도, 분류해야할 클래스 수에따라, 적합한 손실함수도 다름)

일단, 지금까지 배운것만 봤을 때
-> 이진분류문제 = binary_crossentropy
-> 다중분류문제 = categorical_crossentropy
-> 회귀문제문제 = Connection_Temporal_classification(CTC)
를 사용한다고 알고 있자.

[참고]

레이블과 손실을 다루는 다른 방법

앞서, 레이블 인코딩하는 법에는
1. 원-핫 인코딩 = 범주형 인코딩 (이번 예제에서 한 것)
2. 레이블 리스트 -> 정수 텐서로 변환하는 것

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

이 방식을 사용할 경우, 변경 사항은 하나.
= 손실함수를 범주형 인코딩했을 경우 -> categorical_crossentropy 사용한 것을 sparse_categorical_crossentropy 로 변경하면 됨.
-> 이 손실함수는 인터페이스만 다를 뿐, categorical_crossentropy와 수학적으로 동일

[참고]

충분히 큰 중간 층을 두어야하는 이유

If) 마지막 출력층이 46차원인데, 중간 층의 히든 유닛(차원)이 훨씬 작은 차원의 층으로 구성된다면 ?

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu')) # 중간 층 4개의 은닉 유닛(4차원)으로 모델 구성할 경우
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

```
Epoch 1/20
63/63 [==============================] - 1s 17ms/step - loss: 3.4111 - accuracy: 0.1238 - val_loss: 3.0171 - val_accuracy: 0.2370
Epoch 2/20
63/63 [==============================] - 1s 15ms/step - loss: 2.6398 - accuracy: 0.2756 - val_loss: 2.3291 - val_accuracy: 0.5370
Epoch 3/20
63/63 [==============================] - 1s 15ms/step - loss: 1.8865 - accuracy: 0.5665 - val_loss: 1.7347 - val_accuracy: 0.5630
Epoch 4/20
63/63 [==============================] - 1s 15ms/step - loss: 1.4960 - accuracy: 0.5975 - val_loss: 1.5912 - val_accuracy: 0.5930
Epoch 5/20
63/63 [==============================] - 1s 15ms/step - loss: 1.3508 - accuracy: 0.6417 - val_loss: 1.5371 - val_accuracy: 0.6170
Epoch 6/20
63/63 [==============================] - 1s 16ms/step - loss: 1.2519 - accuracy: 0.6595 - val_loss: 1.5010 - val_accuracy: 0.6190
Epoch 7/20
63/63 [==============================] - 1s 15ms/step - loss: 1.1715 - accuracy: 0.6721 - val_loss: 1.5109 - val_accuracy: 0.6160
Epoch 8/20
63/63 [==============================] - 1s 15ms/step - loss: 1.1003 - accuracy: 0.6914 - val_loss: 1.4882 - val_accuracy: 0.6300
Epoch 9/20
63/63 [==============================] - 1s 15ms/step - loss: 1.0400 - accuracy: 0.7108 - val_loss: 1.5167 - val_accuracy: 0.6400
```

위에, 오버피팅을 고려해서, 9 epoch까지만 학습한 모델이라 생각했을 때,
모델 정확도는 0.71로, 중간 층이 64개 은닉 유닛이었을 때 모델의 정확도 약 0.79와 비교해서 약 0.08(즉, 8%)정도의 정확도 차이가 났다.
=> 이런, 손실의 대부분 원인은 많은 정보를 중간층의 저차원 표현 공간으로 압축할려 했기 때문

3-5장 요약

이것만이라도..기억합시다

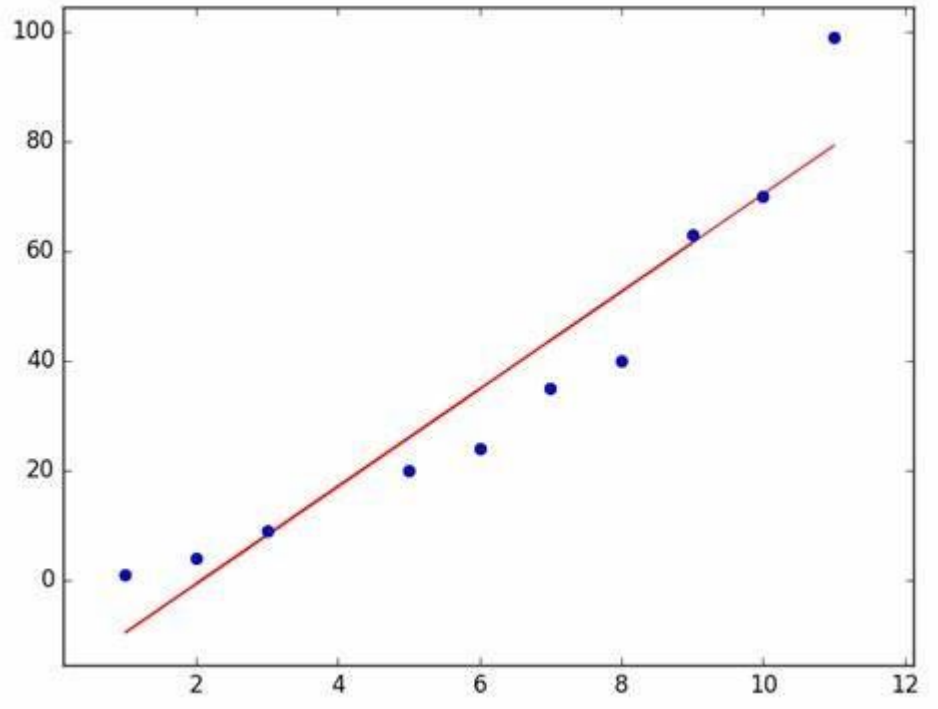
- N개의 클래스로 데이터를 분류 = 모델 네트워크 마지막 층의 크기도 N이어야 함.
- 단일 레이블, 다중 분류 문제에는 N개 클래스에 대한 확률 분포 출력이 목적 = “ softmax 활성화함수 “
  (앞선, 이진 분류 문제에선 => “ sigmoid 활성화 함수 “)
- 이런, 다중 분류 문제는 항상 “범주형 크로스엔트로피” 사용 해야 함.
  이는, 모델이 출력한 확률 분포와 타깃 분포 사이의 거리를 최소화 함.
  (앞선, 이진 분류 문제같은건 => “이진 크로스엔트로피 “
- 다중 분류에서 레이블을 다루는 두 가지 방법
  - 범주형 인코딩(원-핫 인코딩) => categorical_crossentropy 손실 함수 사용
  - 레이블을 정수로 인코딩 => sparse_categorical_crossentropy 손실 함수 사용
- 많은 수의 클래스(범주)를 분류할 때는, 중간 층의 크기가 너무 작아 네트워크에 정보의 병목현상이 생기지 않도록 합시다.

  추가로… 이번 예제하면서 **“ 과대적합(Overfitting) “**에 대해, 좀 더 알아봐야 할 것 같단 생각

3-6

보스턴 주택 가격 예측하기

3-6 주택 가격 예측



회귀 분석

: 변수간의 관계를 파악하여 모델을 설정한 후 그 모델이 적합한지 검사하는 과정.

왼쪽 그래프에서 파란 점을 통해서 빨간 직선관계가 있음을 알아내는것을 '추정' 이라하고,

새로운 데이터가 들어왔을 때, 우리가 알고싶은 데이터를 추정한 식을 통해 구해내는 것을 '예측' 이라고 정의(x0=5이면 y0=?)

해당 데이터셋을 통해서 다른 새로운 데이터가 들어왔을 때, 집값을 예측할 수 있는 모델을 만드는 것이 목표.

3-6 주택 가격 예측

해당 데이터 셋은 다른 예시와 달리 표본의 크기가 작음.

=> k-겹 교차 검증(K-f0ld cross validation)을 통해 검증 점수의 분산이 큰 문제점을 해결하고자 함.

3-6

3. 각 특성들이 크기가 서로 다름

ex) 범죄율은 0과 1사이의 비율로 나타나 있음

이를 해결하기 위해서 각 데이터 범주별로 데이터 정규화를 시킴.

이때, 테스트 데이터 셋또한 훈련 데이터셋의 평균과 표준편차를 이용해서 정규화를 시킴

=> 같은 변환을 시켜야 검증 결과가 올바르게 나오기 때문.

```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std

test_data -= mean
test_data /= std
```

3-6 모델 구성

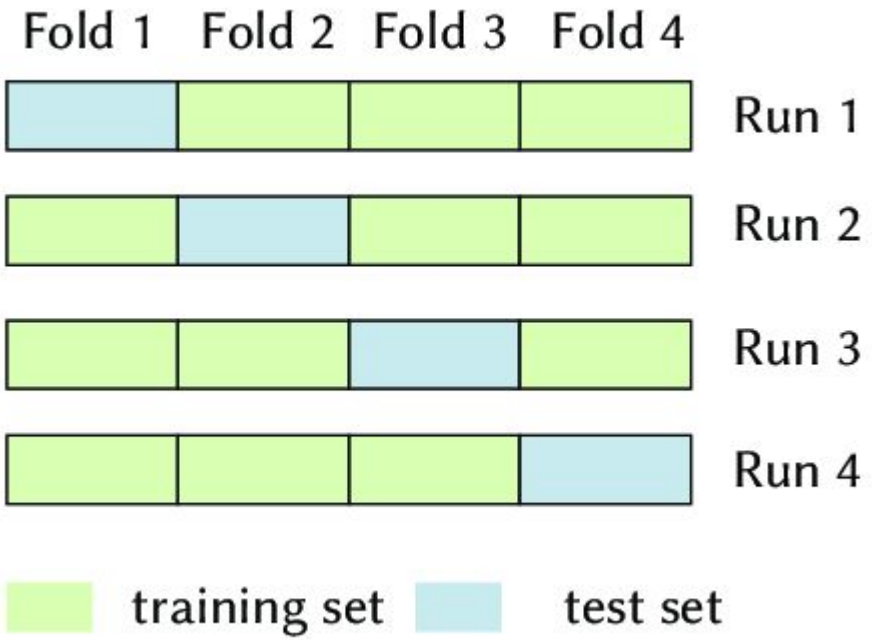
64개의 유닛을 가진 2개의 은닉층으로 네트워크 구성.

(train_data가 적으면 작은 모델을 써서 과대적합을 방지)

```
from keras import models
from keras import layers

def build_model():
    # 동일한 모델을 여러 번 생성할 것이므로 함수를 만들어 사용합니다
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

3-6 K-겹 교차 검증(K-fold cross-validation)

- 데이터 수가 많지 않기 때문에, K-겹 교차검증을 사용.
- 여기에선 k=4로 설정한 후, 3개의 분할로 훈련을 시키고 나머지 분할로 평가



3-6 평균 mae값 구하기

```
import numpy as np

k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print('처리중인 폴드 #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

검증 데이터 저장

np.concatenate를 이용해서
검증 데이터 셋을 기준으로 왼쪽
배열과 오른쪽 배열을 합쳐줌.

```
all_scores
```

```
[2.0591812133789062, 2.548744678497314 5, 2.966973066329956, 2.54862117767334]
```

```
np.mean(all_scores)
```

```
2.530880033969879
```

3-6 검증 점수 저장

- epoch를 500까지 늘린 후, mae_history에 epoch=1~500까지의 검증점수를 저장

```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=1, verbose=0)
    mae_history = history.history['val_mae']
    all_mae_histories.append(mae_history)
```

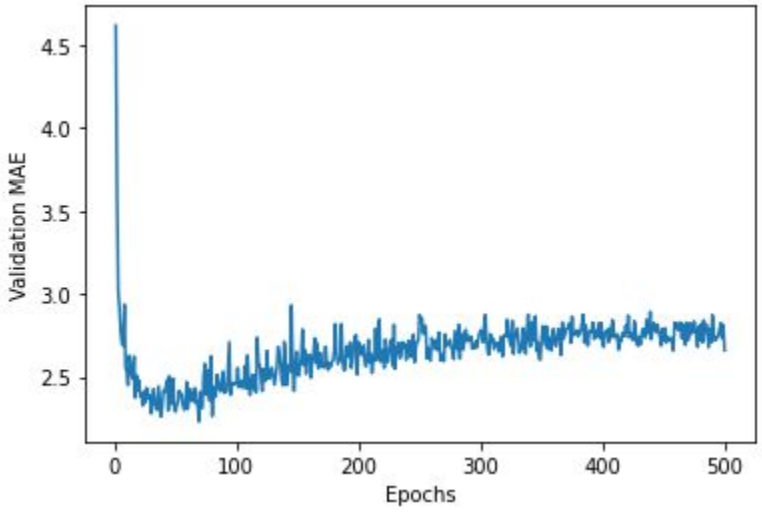
.history를 이용해서 학습시킨 데이터들을 **val_data**로 검증한 **mae** 값을 저장함.

3-6

```
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

각 분할별 mae 값을
평균으로 계산하여 저장

```
import matplotlib.pyplot as plt

plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```



epoch 별 mae값을 그래프로 출력

but, 해석에 어려움이 존재

3-6 그래프 변환

1. epoch가 0 근처일 때 mae 값이 비정상적으로 크므로 제외시킴
2. 완곡한 곡선을 얻기 위해서 지수 이동 평균(exponential moving average)을 이용(여기선 가중치(알파)를 0.1로 둠)

```
def smooth_curve(points, factor=0.9):
  smoothed_points = []
  for point in points:
    if smoothed_points:
      previous = smoothed_points[-1]
      smoothed_points.append(previous * factor + point * (1 - factor))
    else:
      smoothed_points.append(point)
  return smoothed_points

smooth_mae_history = smooth_curve(average_mae_history[10:])

plt.plot(range(1, len(smooth_mae_history) + 1), smooth_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

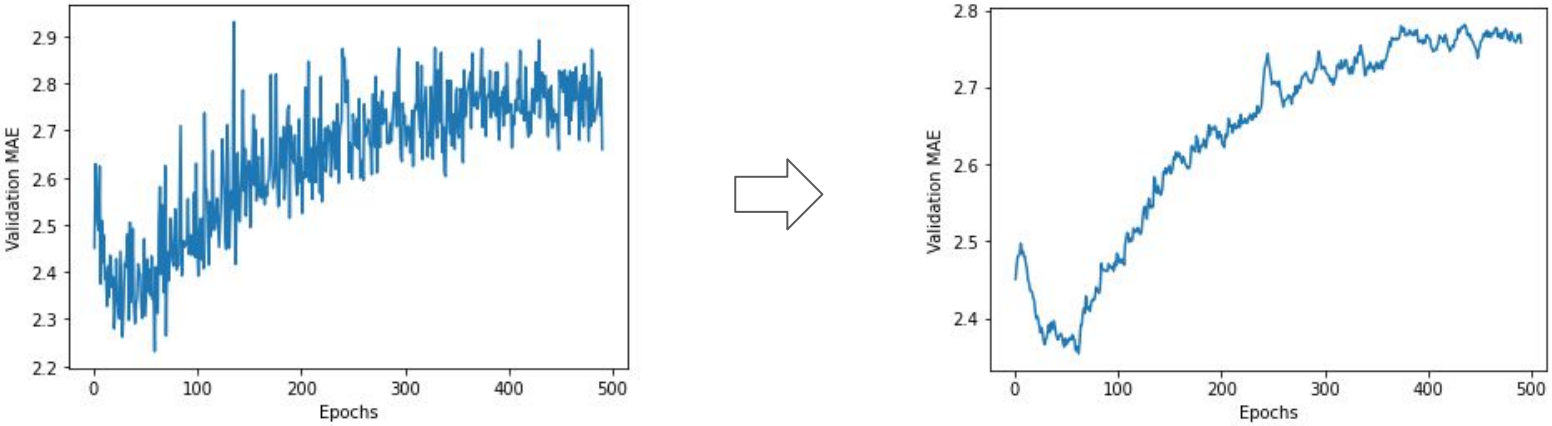
$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1-\alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

$$S_1 = 0.9 * Y_1$$

$$S_i = 0.9 * \left(Y_i + \sum_{k=1}^{i-1} (0.1)^k * Y_k \right)$$

$$= 0.9Y_i + 0.09Y_{i-1} + 0.009Y_{i-2} + \dots$$

3.6 그래프 변환



1. epoch가 80근처 일 때 mae가 최소화된 후 증가. (그 이후로 과대적합 발생)

   => epoch=90일 때 최소화 되는 것으로 추정(초반 10개 값을 뺐으므로)

3-6 Test data를 통한 성능 확인

```
model = build_model()
model.fit(train_data, train_targets,
          epochs=90, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
test_mae_score
```

```
4/4 [==============================] - 0s 2ms/step - loss: 16.6734 - mae:
2.66764235496521
```