2, 3장

박채원, 윤예준, 최혜원

2장. 신경망 시작하기 전에

2장.신경망

1) 신경망과의 첫 만남

범주(category) = 클래스(class)
데이터포인트 = 샘플(sample)
특정샘플의 클래스 = 레이블(lable)

   a) 작업순서
      i) Train_images와 train_labels를 네트워크에 주입
      ii) 이미지와 레이블을 연관시킬 수 있도록 학습
      iii) Test_images에 대한 예측을 네트워크에 요청
      iv) 예측이 test_lables와 맞는지 확인
   b) 핵심구성요소
      i) Layer : 더 유용한 형태로 출력
      ii) 손실함수 : 훈련데이터에서 신경망이 성능을 측정하는 방법
      iii) 옵티마이저 : 입력데이터와 손실함수를 기반으로 네트워크를 업데이트
      iv) 훈련과 테스트 과정을 모니터링할 지표

2장.신경망

2) 신경망을 위한 데이터 표현
   a) 스칼라(0D 텐서)
      i) Float32, float64
      ii) Ndim = 0 (스칼라 텐서의 축 개수, rank = 0)
   b) 벡터(1D 텐서)
      i) 1개의 축
      ii) ex) [12, 3, 6, 14, 7] -> 5개의 원소, 5차원 벡터 (5D 텐서와 다름)
   c) 행렬(2D 텐서)
      i) 2개의 축(행, 열)
   d) 3D 텐서와 고차원 텐서
      i) 3개의 축 혹은 그 이상
      ii) 딥러닝에서는 보통 0D ~ 4D, 동영상은 5D

2장.신경망

2) 신경망을 위한 데이터 표현

e) 핵심속성

i) 축의 개수(rank) : ndim에 저장

ii) 크기(shape) : 텐서의 차원을 튜플형태로 반환

iii) 데이터 타입 : float32, uint8, float64

f) 넘파이로 텐서 조작하기

i) 슬라이싱 /ex) my_slice = train_images[10:100]

ii) 간격 선택 / ex) my_slice = train_images[:, 14:, 14:]

iii) 음수 인덱스 / ex) my_slice = train_images[:, 7:-7, 7:-7]

g) 배치 데이터

i) 샘플 축 : 모든 데이터 텐서의 첫 번째 축

ii) 배치 데이터 : batch = train_images[128 * n : 128 * (n+1)]

2장.신경망

2) 신경망을 위한 데이터 표현

h) 텐서의 실제 사례

i) 벡터 데이터 : (samples, features) 크기의 2D 텐서

ii) 시계열 데이터/시퀀스 데이터 : (samples, timesteps, features) 크기의 3D데이터

iii) 이미지: (samples, height, width, channels) 또는 (samples, channels, height, width)

iv) 동영상: (samples, frames, height, width, channels) 5D 텐서

i) 벡터 데이터

i) (samples, features)

ii) ex1) 사람의 나이, 우편번호, 소득으로 구성된 인구 통계 데이터. 각 사람당 3개의 특징을 갖고 사람들이 만 명일 때 -> (10000, 3)

iii) ex2) (공통단어 2만로 만든 사전) 각 단어가 등장한 횟수로 표현된 텍스트 문서 데이터셋이 500개-> (500, 20000) 크기의 텐서 (단어가 특성)

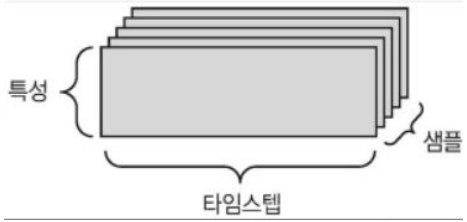
2장.신경망

2) 신경망을 위한 데이터 표현

j) 시계열 데이터 | 시퀀스 데이터

i) 데이터에서 시간이 중요할 때. 연속된 순서가 중요할 때 3D 텐서.

ii) 각 샘플은 벡터(2D)의 연속으로 인코딩되므로 배치 데이터는 3D 텐서로 인코딩



iii) 시간축은 두 번째 축(idx = 1)

iv) ex1) 주식 가격 데이터셋- 1분마다 (현재 주식가격, 지난1분동안 최고가격, 최소가격), 하루의 거래시간 390분, 250일치의 데이터 -> (250, 390, 3)

v) ex2) 트윗 데이터셋 - 128개의 알파벳으로 구성된 280개의 문자시퀀스 트윗 100만개 (해당문자의 인덱스만 1, 나머지 0)->(1000000, 280, 128)

2장.신경망

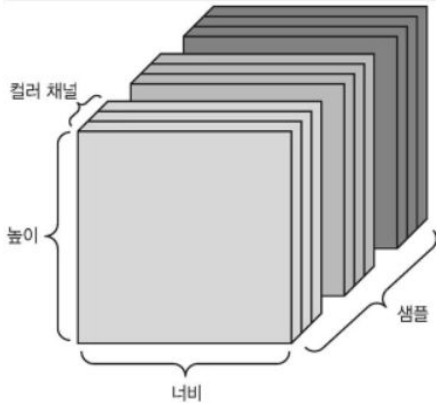
2) 신경망을 위한 데이터 표현

k) 이미지 데이터

i) 높이, 너비, 컬러(흑백 : (128,256,256,1)/컬러:(128, 256, 256,3))

ii) 채널 마지막 방식 (samples, height, width, color_depth) 구글텐서플로, 케라스 지원

iii) 채널 우선방식(samples, color_depth, height, width) 씨아노, 케라스 지원



2장.신경망

2) 신경망을 위한 데이터 표현

I) 비디오 데이터

i) 프레임의 연속(frames, height, width, color_depth), 각 프레임은 하나의 컬러 이미지 (height, width, color_depth)

ii) 여러 비디오의 배치(samples, frames, height, width, color_depth)

iii) ex) 60초짜리 144x256 유튜브 비디오 클립을 초당 4프레임으로 샘플링
->(4, 240, 144, 256, 3)

2장.신경망

3) 신경망의 톱니바퀴: 텐서 연산

a) 케라스 층 생성

keras.layers.Dense(512, activation='relu')

-> Output = relu(dot(W, input) + b)

3가지 텐서 연산

i) 텐서 W와 입력 텐서 사이 점곱
ii) 점곱의 결과인 2D텐서와 벡터 b 사이의 덧셈
iii) Relu 연산

relu 연산

```
In [1]: def naive_relu(x):
            assert len(x.shape)

            x = x.copy()
            for i in range(x.shape[0]):
                for j in range(x.shape[1]):
                    x[i, j] = max(x[i, j], 0)

            return x
```

```
In [2]: def naive_add(x,y):
            assert len(x.shape) == 2
            assert x.shape == y.shape

            x = x.copy()
            for i in range(x.shape[0]):
                for j in range(x.shape[1]):
                    x[i, j] += y[i, j]

            return x
```

덧셈 연산

```
In [ ]: import numpy as np
        z = x + y
        z = np.maximum(x, 0.)
```

넘파이 배열을 다룰때는 내장함수로
간단하게 연산을 처리할 수 있다.

2장.신경망

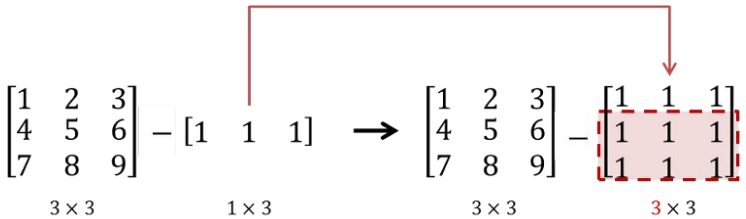
3) 신경망의 톱니바퀴: 텐서 연산

b) 브로드캐스팅

i) 큰 텐서의 ndim에 맞도록 작은 텐서에 축을 추가해준다.

ii) 작은 텐서가 새 축을 따라서 큰 텐서의 크기에 맞도록 반복한다.

예를 들어 (32, 10)의 크기를 갖는 **x**와 (10,)의 크기를 갖는 **y**가 있다고 할 때, y가 x의 ndim을 맞추기 위해 y에 비어있는 첫번째 축을 추가해줘서 y의 크기가 (1, 10)이 된다.
그 후 이 축에 y를 32번 반복하면 텐서 y의 크기는 (32, 10)이 되어 두 텐서를 더할 수 있게된다.



2장.신경망

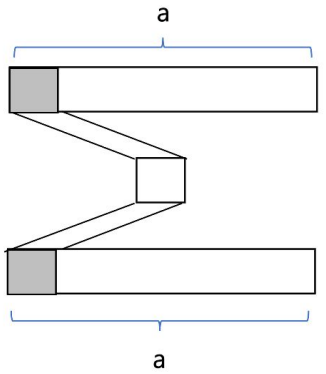
3) 신경망의 톱니바퀴: 텐서 연산

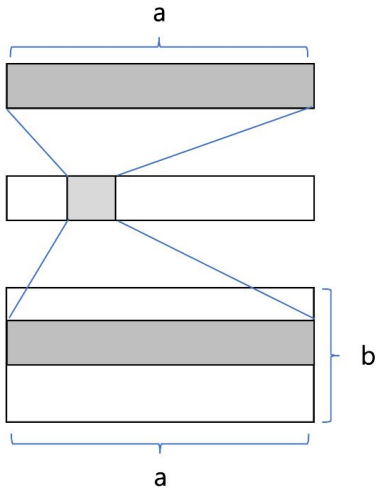
c) 텐서 점곱

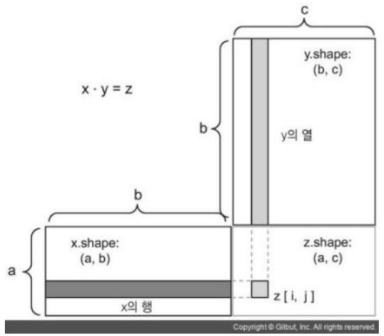
i) 두 벡터간 점곱 - 결과값이 스칼라 값, x.shape == y.shape

ii) 벡터와 행렬간의 점곱 - 결과값이 벡터, x.shape[1] == y.shape[0]

iii) 두 행렬간의 점곱 - 결과값이 행렬, x.shape[1] == y.shape[0]







2장.신경망

3) 신경망의 톱니바퀴: 텐서 연산
   d) 텐서 크기 변환
   e) 텐서 전치

```
>>> x = np.array([[0., 1.],
                  [2., 3.],
                  [4., 5.]])
>>> print(x.shape)
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
>>> x = x.reshape((2, 3))
>>> x
array([[ 0., 1., 2.],
       [ 3., 4., 5.]])
```

텐서 크기 변환 (reshape)
- 원소의 개수는 동일하고 단지 특정 크기에 맞게 열과 행을 재배열

```
>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> print(x.shape)
(20, 300)
```

텐서 전치(transpose)
- 행과 열을 바꾸는 것

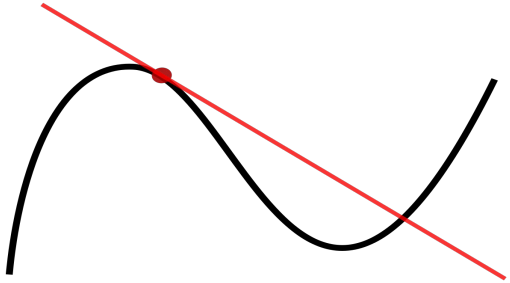
2장.신경망

4) 신경망의 엔진: 그래디언트 기반 최적화

a) 변화율이란?

i) 미분

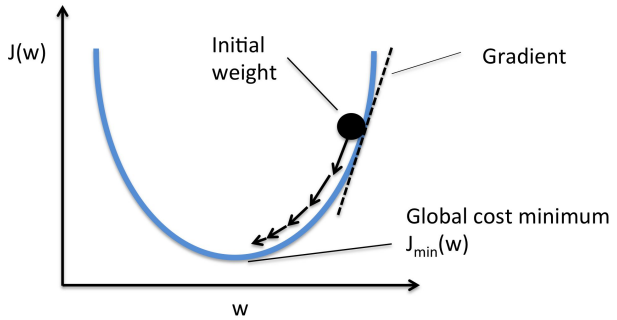
ii) 기울기 a가 음수인 경우와 양수인 경우 x가 변할 때 f(x)의 변화



b) 텐서 연산의 변화율: 그래디언트

i) 텐서 연산의 변화율

ii) W1 = W0 - step * gradient(f)(W0)



2장.신경망

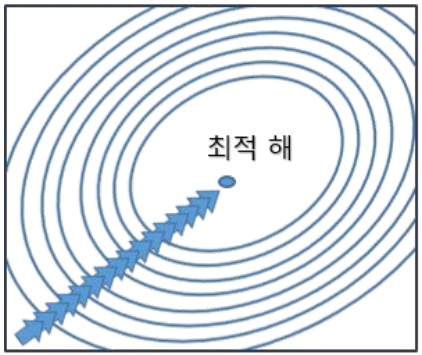
4) 신경망의 엔진: 그래디언트 기반 최적화

c) 확률적 경사 하강법

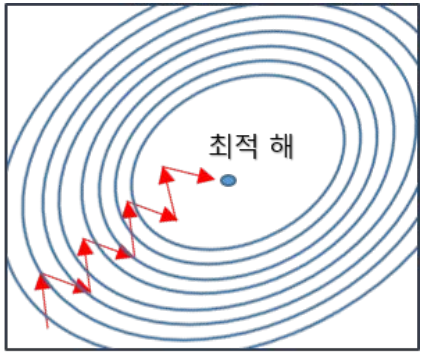
i) 추출된 데이터 한 개에 대해서 그래디언트를 계산하고, 경사 하강 알고리즘 적용하는 것

ii) 배치 경사 하강법에 비해 적은 데이터로 학습할 수 있고, 속도가 빠르다. 그러나 노이즈가 심하며 지역 최솟값을 가질 수 있음.

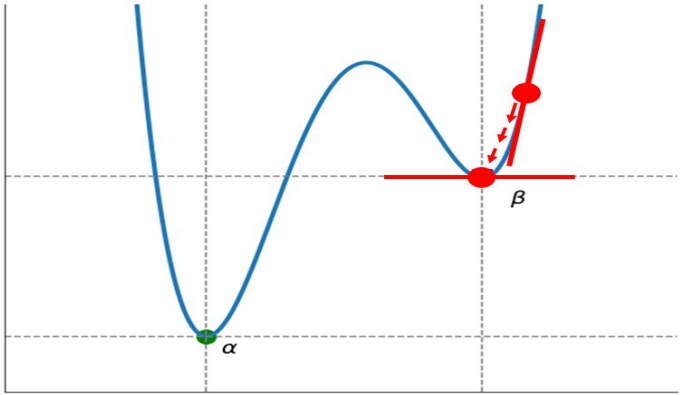
iii) SGD, MSGD



경사 하강법



확률적 경사 하강법



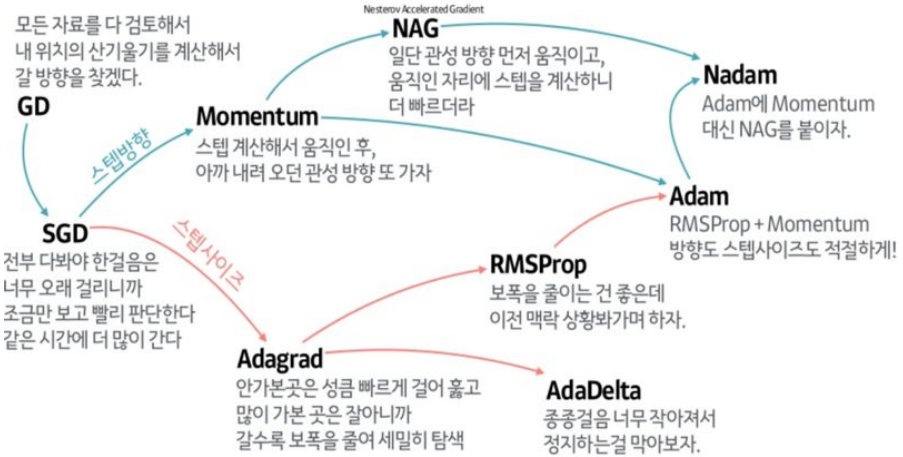
2장.신경망

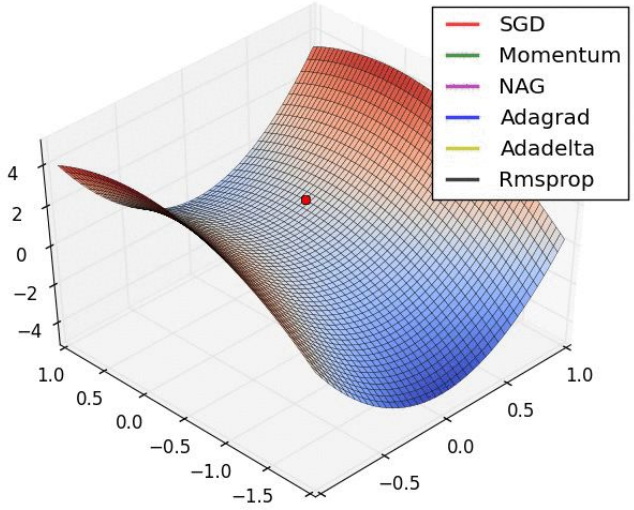
4) 신경망의 엔진: 그래디언트 기반 최적화

d) 경사 하강법 최적화 방법

i) SGD의 문제점인 수렴 속도와 지역 최솟값을 해결

ii) Momentum, NAG, Adagrad, Adadelta, RMSProp





2장.신경망

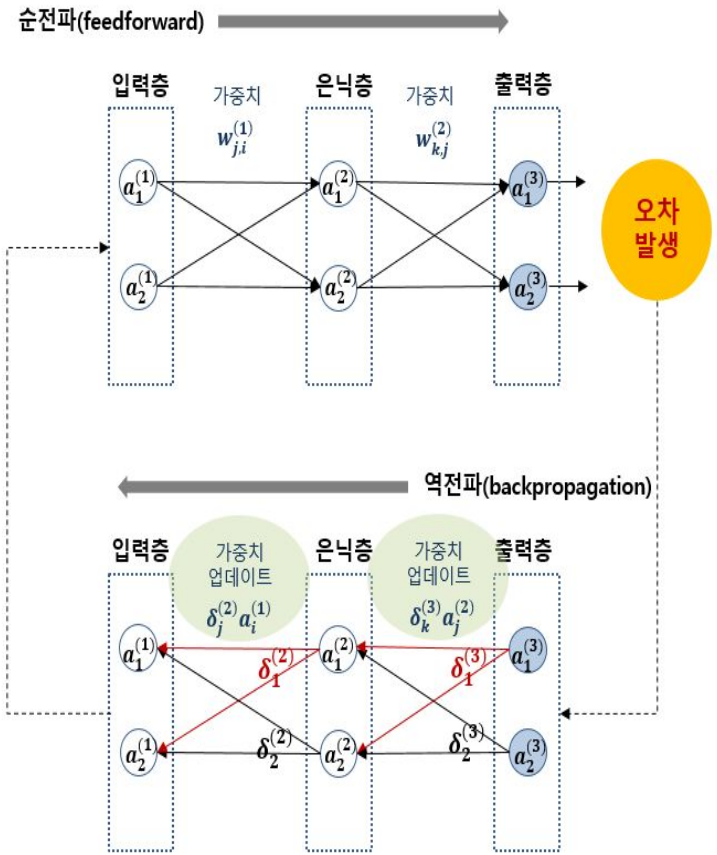
4) 신경망의 엔진: 그래디언트 기반 최적화

e) 변화율 연결: 역전파 알고리즘

i) 순전파와 반대로 출력층에서 입력층 방향으로 계산하면서 가중치 업데이트

ii) 동일 입력층에 대해 원하는 값이 출력되도록 개개의 가중치를 조정하는 방법으로 사용되며, 느리지만 안정적인 결과를 얻을 수 있다.

iii) 단순히 기울기가 작아지는 방향으로 움직이기 때문에, 시작점에 따라 결과가 달라질 수 있다.



2장.신경망

5) 첫 번째 예제 다시 살펴보기

a) Data Load

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

b) Data Type

```
train_images = train_images.reshape((60000, 28*28))
train_images = train_images.astype('float32')/255

test_images = test_images.reshape((10000, 28*28))
test_images = test_images.astype('float32')/255
```

2장.신경망

5) 첫 번째 예제 다시 살펴보기

c) 신경망 layer 추가

```
[8]  1  from keras import models
     2  from keras import layers
     3
     4  network = models.Sequential()
     5  network.add(layers.Dense(512, activation='relu', input_shape=(28*28,)))
     6  network.add(layers.Dense(10, activation='softmax'))
```

d) 네트워크 컴파일

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

- categorical_crossentropy : 손실함수 -> 미니배치 확률적 경사 하강법으로 손실감소
- rmsprop 옵티마이저 : 경사 하강법을 적용하는 구체적 방식 결정 (수렴속도, 지역최솟값 해결)

2장.신경망

5) 첫 번째 예제 다시 살펴보기

e) 훈련 반복

```
[13]  1  network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

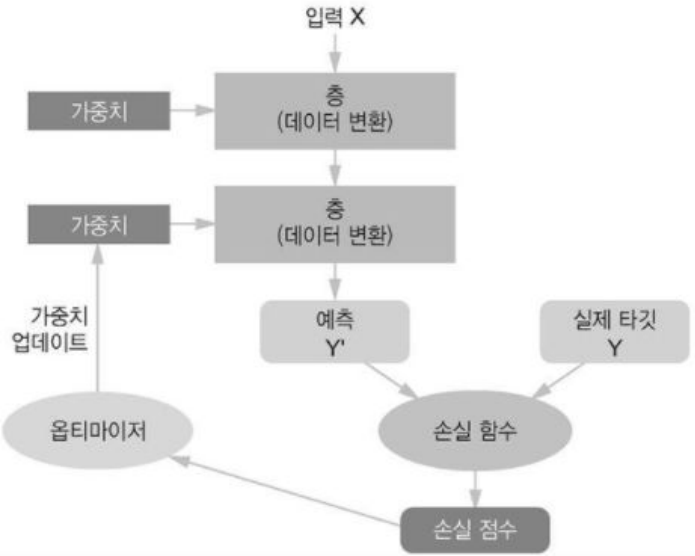
```
Epoch 1/5
469/469 [==============================] - 15s 3ms/step - loss: 0.4388 - accuracy: 0.8716
Epoch 2/5
469/469 [==============================] - 1s 3ms/step - loss: 0.1143 - accuracy: 0.9666
Epoch 3/5
469/469 [==============================] - 1s 3ms/step - loss: 0.0721 - accuracy: 0.9788
Epoch 4/5
469/469 [==============================] - 1s 3ms/step - loss: 0.0495 - accuracy: 0.9856
Epoch 5/5
469/469 [==============================] - 1s 3ms/step - loss: 0.0366 - accuracy: 0.9892
<keras.callbacks.History at 0x7f796006d2d0>
```

- 128개 샘플씩 미니 배치로 훈련데이터를 epoch만큼 반복
- 각 반복마다 batch에서 loss에 대한 가중치의 gradient계산하고 가중치 업데이트

3장. 신경망 시작하기

3.1 신경망의 구조

- **네트워크(또는 모델)**를 구성하는 **층(layer)**
- **입력 데이터**와 그에 상응하는 **타깃**
- 학습에 사용할 피드백 신호를 정의하는 **손실함수**
- 학습 진행방식을 결정하는 **옵티마이저**
- 옵티마이저 : 손실값을 사용하여 네트워크 가중치 업데이트



3.1 신경망의 구조

1) **층** : 딥러닝의 구성 단위
   a) 층: 하나 이상의 텐서를 입력으로 받아 하나 이상의 텐서를 출력하는 데이터 처리 모듈
   b) 대부분 가중치(확률적 경사 하강법에 의해 학습되는 하나 이상의 텐서)라는 층의 상태를 가짐
   c) - 2D 벡터데이터 = 밀집 연결층에 의해 처리됨(Dense)
      - 3D 시퀀스데이터 = LSTM같은 순환층에 의해 처리됨
      - 4D 이미지데이터 = 2D합성곱 층에 의해 처리됨(Conv2D)

```
from keras import layers
from keras import models

model= models.Sequential()
model.add(layers.Dense(32, input_shape=(784,))  #32개의 유닛으로 된 밀집층, 차원이 784인 2D 텐서 입력받음
model.add(layers.Dense(10))  #자동으로 input이 32로 맞춰짐
```

3.1 신경망의 구조

2) **모델** : 층의 네트워크
   a) 층으로 만든 비 순환 유향 그래프
   b) 하나의 입력을 하나의 출력으로 매핑하는 층을 순서대로 쌓는 것.
   c) 다양한 네트워크 구조 존재.
      - 가지(branch)가 2개인 네트워크
      - 출력이 여러 개인 네트워크
      - 인셉션(Inception) 블록
   d) 위 네트워크 구조는 가설공간(hypothesis space)을 정의
   e) 네트워크 구조를 선택함으로써 가능성 있는 공간을 입력 데이터에서 출력 데이터로 매핑하는 일련의 특정 센서 연산으로 제한.

3.1 신경망의 구조

3) **손실함수**와 **옵티마이저**: 학습과정을 조절하는 열쇠
   a) 네트워크 구조 정의 후, 선택해야하는 항목- 손실함수& 옵티마이저
   b) 손실함수 : 훈련하는 동안 최소화될 값. 주어진 문제에 대한 성공 지표.
      - 출력 당 하나씩 손실함수.
      - but, 경사하강법은 하나의 스칼라 소실 값을 기준
      - 모든 손실을 평균내서 하나의 스칼라로 합침
   c) 문제에 맞는 올바른 손실함수 선택해야함.
      - 2개 클래스(분류) : 이진 크로스엔트로피(binary crossentropy)
      - 여러개 클래스(분류) : 범주형 크로스엔트로피(category crossentropy)
      - 회귀 : 평균 제곱 오차
      - 시퀀스 학습 : CTC(Connection Temporal Classification)

3.2 케라스 소개

1) 케라스 특징
   - 동일 코드로 CPU 와 GPU 에서 실행할 수 있다.
   - 사용하기 쉬운 API -> 딥러닝 모델의 프로토타입을 빠르게 만들 수 있다.
   - 합성곱 신경망, 순환 신경망을 지원. 자유롭게 조합 가능
   - 다중 입력, 다중 출력 모델, 층의 공유, 모델 공유 등 어떤 네트워크 구조도 생성가능 -> GAN부터 뉴럴 튜링 머신까지 어떤 딥러닝 모델에도 적합하다는 의미.

3.2 케라스 소개

1) 케라스, 텐서플로 씨아노, CNTK

여러 백엔드 엔진 연동 가능

고수준 모델 제공 라이브러리,모듈구조

딥러닝 백엔드 엔진

심층 신경망 라이브러리, 저수준 텐서 연산 라이브러리

3.2 케라스 소개

1) 케라스를 사용한 개발: 빠르게 둘러보기
   a) 입력 텐서와 타깃 텐서로 이루어진 훈련 데이터를 정의
   b) 입력과 타깃을 매핑하는 층으로 이루어진 네트워크(모델)을 정의
   c) 손실함수, 옵티마이저, 모니터링하기 위한 측정 지표를 선택하여 학습과정 설명
   d) 훈련 데이터에 대해 모델의 fit() 메서드를 반복적 호출

3.2 케라스 소개

1) 케라스를 사용한 개발: 빠르게 둘러보기

a) , b) 모델정의하는 두 가지 방법 - 7장에 더 자세히 나온다

- Sequential 클래스 이용 : 자주 사용하는 구조인 층을 순서대로 쌓아올림

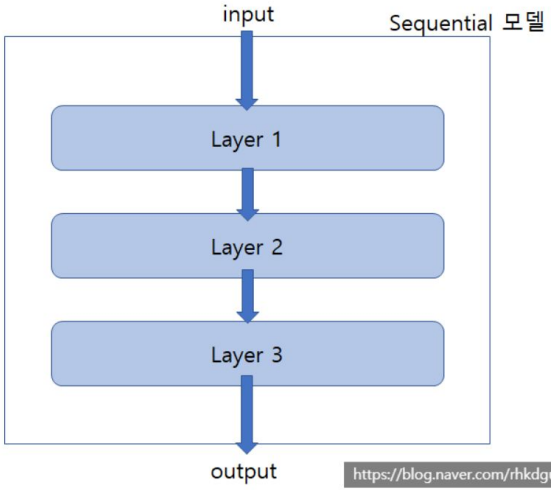
```
from keras import layers
from keras import models

model= models.Sequential() #모델 자동 생성
model.add(layers.Dense(32, input_shape=(784,))
model.add(layers.Dense(10))
```

- 함수형 API(임의의 구조를 만들 수 있는 비순환 유향 그래프)

```
Input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation = 'relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)
mode = models.Model(inputs = input_tensor, outputs = output_tensor)
```

3.2 케라스 소개

1) 케라스를 사용한 개발: 빠르게 둘러보기
   - Sequential 클래스 이용 :




```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu',
input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

```
seq_model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_9 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_10 (Dense)             (None, 10)                330
=================================================================
Total params: 3,466
Trainable params: 3,466
Non-trainable params: 0
_________________________________________________________________
```

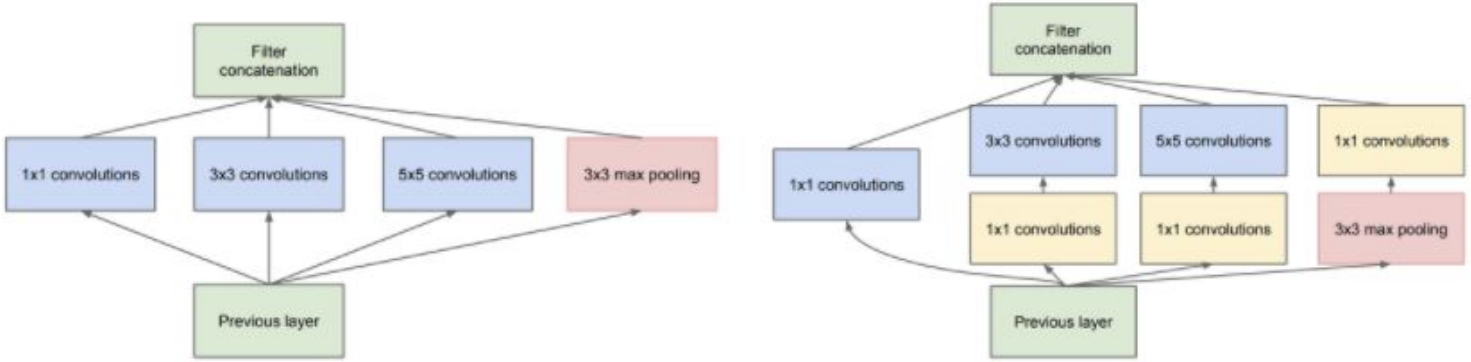
3.2 케라스 소개

1) 케라스를 사용한 개발: 빠르게 둘러보기

- Sequential 클래스 이용 :

한계! ⇨ - 다중입력모델, 다중 출력 모델
- 인셉션 모듈 : 나란히 놓인 합성곱 층으로 구성
- 잔차 연결 : 하위층의 출력을 상위 층의 특성맵에 더함

⇨ 함수형 API 사용



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

3.2 케라스 소개

1) 케라스를 사용한 개발: 빠르게 둘러보기
- 함수형 API 이용:

```
input_tensor = Input(shape=(64,))
x1 = layers.Dense(32,
activation='relu')(input_tensor)
x2 = layers.Dense(32, activation='relu')(x1)
output_tensor = layers.Dense(10,
activation='softmax')(x2)

model = Model(input_tensor,output_tensor)
model.summary()
```

```
Model: "functional_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 64)]              0
_________________________________________________________________
dense_11 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_12 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_13 (Dense)             (None, 10)                330
=================================================================
Total params: 3,466
Trainable params: 3,466
Non-trainable params: 0
_________________________________________________________________
```

3.2 케라스 소개

1) 케라스를 사용한 개발: 빠르게 둘러보기

c) 손실함수, 옵티마이저, 모니터링하기 위한 측정 지표를 선택하여 학습과정 설명

```
from keras import optimizers

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

적절한 손실함수 적용

- 2개 클래스(분류) : 이진 크로스엔트로피(binary crossentropy)
- 여러개 클래스(분류) : 범주형 크로스엔트로피(category crossentropy)
- 회귀 : 평균 제곱 오차
- 시퀀스 학습 : CTC(Connection Temporal Classification)

3.2 케라스 소개

1) 케라스를 사용한 개발: 빠르게 둘러보기

d) 훈련 데이터에 대해 모델의 fit() 메서드를 반복적 호출 : 학습

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

3.4 영화 리뷰 분류: 이진 분류 예제

1) IMDB 데이터셋 구성/로드

```
In [2]: import tensorflow as tf
        import keras

In [3]: from keras.datasets import imdb

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

양극단의 영화 리뷰 5만개로 이루어진
데이터 셋인 IDMB을 이용해 이진 분류

훈련 데이터, 테스트 데이터 각 2만 5천개

```
In [45]: train_data[0]

Out[45]: [1,
          14,
          22,
          16,
          43,
          530,
          973,
          1622,
          1385,
          65,
          458,
          4468,
          66,
          3941,
          4,
          173,
          36,
          256,
          5,
```

train_data는
각 리뷰가 숫자 시퀀스로
변환 되어있음.
여기서 숫자
시퀀스는 사전에 있는
고유한
단어를 가리킴

Ex) 14->'this'
22_>film'

```
In [39]: train_labels[0]

Out[39]: 1
```

라벨은 ndarray로 표현되어있다.
0은 부정을 나타내는 데이터,
1은 긍정을 나타내는 데이터를 의미

이 리스트들을 신경망에
넣어줄 수 있는
텐서 형태로 변환하기 위해
vectorize_squences 함수 정의

3.4 영화 리뷰 분류: 이진 분류 예제

2) 텐서 형태로 변경(벡터, 스칼라)

In [4]:
```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results=np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

vectorize_squences 에서는 enumerate 함수를 이용해 train_data 원소를 인덱스와 함께 요소를 반환해 인덱스에 해당하는 특정 부분을 1로 채우는 원-핫 인코딩을 하였다.
train_data와 test_data를 이 함수를 통해 신경망에 넣어줄 수 있는 0과 1만을 갖는 벡터로 변환한다.

그 결과 x_train[0]이 다음과 같은 벡터를 갖는다.

In [6]:
```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

label은 ndarray였으므로 간단하게 asarray를 이용해 변경해줬다.

In [5]:
```
x_train[0]
```

Out[5]:
```
array([0., 1., 1., ..., 0., 0., 0.])
```

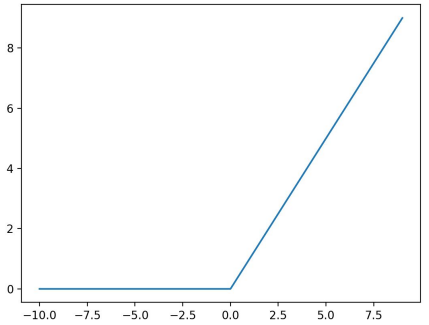
3.4 영화 리뷰 분류: 이진 분류 예제

3) 모델 생성

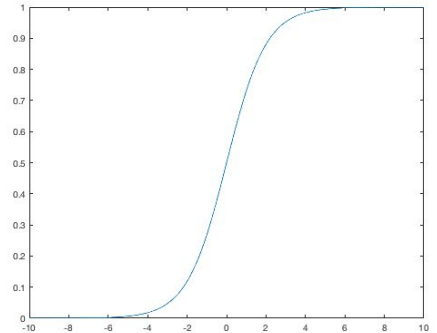
```
In [7]: from keras import models
        from keras import layers

        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'))
```

이제 신경망을 구성하는데, 책에서 추천하는 구조인
은닉 층 2개, 예측을 출력하는 층 하나, 총 3개의 층을 구성했다.

앞의 2개의 층에선 활성함수로 relu를 이용했는데 이는 음수를
0으로 만들어주는 함수이며
마지막 층에서 활성함수로 사용된 sigmoid는 임의의 값을 [0, 1]
사이로 압축 해, 출력값을 확률처럼 해석할 수 있게 해준다.



relu 함수



Sigmoid 함수

3.4 영화 리뷰 분류: 이진 분류 예제

4) 모델 훈련

```
In [9]: x_val = x_train[:10000]
        partial_x_train = x_train[10000:]
        y_val = y_train[:10000]
        partial_y_train = y_train[10000:]
```

train_data에서 10000의 샘플의 떼어
검증셋을 만들어 준다.

이 검증셋은 학습된 모델이
과적합되지 않았는지 확인을
도와준다.

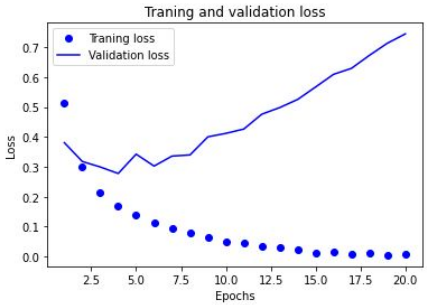
```
In [10]: history = model.fit(partial_x_train,
                             partial_y_train,
                             epochs=20,
                             batch_size=512,
                             validation_data=(x_val, y_val))
Epoch 1/20
30/30 [==============================] - 61s 311ms/step - loss: 0.5890 - accuracy: 0.7053 - val_loss: 0.3809 - val_accuracy: 0.8735
Epoch 2/20
30/30 [==============================] - 1s 30ms/step - loss: 0.3141 - accuracy: 0.9086 - val_loss: 0.3178 - val_accuracy: 0.8760
Epoch 3/20
30/30 [==============================] - 1s 27ms/step - loss: 0.2185 - accuracy: 0.9322 - val_loss: 0.2994 - val_accuracy: 0.8791
Epoch 4/20
30/30 [==============================] - 1s 28ms/step - loss: 0.1707 - accuracy: 0.9448 - val_loss: 0.2776 - val_accuracy: 0.8884
Epoch 5/20
30/30 [==============================] - 1s 28ms/step - loss: 0.1343 - accuracy: 0.9603 - val_loss: 0.3424 - val_accuracy: 0.8647
Epoch 6/20
30/30 [==============================] - 1s 27ms/step - loss: 0.1109 - accuracy: 0.9685 - val_loss: 0.3024 - val_accuracy: 0.8824
Epoch 7/20
30/30 [==============================] - 1s 28ms/step - loss: 0.0926 - accuracy: 0.9739 - val_loss: 0.3359 - val_accuracy: 0.8761
Epoch 8/20
30/30 [==============================] - 1s 26ms/step - loss: 0.0761 - accuracy: 0.9795 - val_loss: 0.3397 - val_accuracy: 0.8794
Epoch 9/20
30/30 [==============================] - 1s 25ms/step - loss: 0.0594 - accuracy: 0.9867 - val_loss: 0.4004 - val_accuracy: 0.8744
Epoch 10/20
30/30 [==============================] - 1s 28ms/step - loss: 0.0498 - accuracy: 0.9900 - val_loss: 0.4120 - val_accuracy: 0.8767
Epoch 11/20
30/30 [==============================] - 1s 25ms/step - loss: 0.0424 - accuracy: 0.9913 - val_loss: 0.4264 - val_accuracy: 0.8707
Epoch 12/20
30/30 [==============================] - 1s 28ms/step - loss: 0.0331 - accuracy: 0.9940 - val_loss: 0.4761 - val_accuracy: 0.8737
Epoch 13/20
30/30 [==============================] - 1s 26ms/step - loss: 0.0251 - accuracy: 0.9946 - val_loss: 0.4983 - val_accuracy: 0.8734
Epoch 14/20
30/30 [==============================] - 1s 27ms/step - loss: 0.0191 - accuracy: 0.9969 - val_loss: 0.5256 - val_accuracy: 0.8706
Epoch 15/20
30/30 [==============================] - 1s 28ms/step - loss: 0.0121 - accuracy: 0.9990 - val_loss: 0.5674 - val_accuracy: 0.8633
Epoch 16/20
30/30 [==============================] - 1s 26ms/step - loss: 0.0139 - accuracy: 0.9985 - val_loss: 0.6097 - val_accuracy: 0.8695
Epoch 17/20
30/30 [==============================] - 1s 27ms/step - loss: 0.0077 - accuracy: 0.9997 - val_loss: 0.6298 - val_accuracy: 0.8654
Epoch 18/20
30/30 [==============================] - 1s 27ms/step - loss: 0.0078 - accuracy: 0.9989 - val_loss: 0.6734 - val_accuracy: 0.8669
Epoch 19/20
30/30 [==============================] - 1s 29ms/step - loss: 0.0050 - accuracy: 0.9996 - val_loss: 0.7140 - val_accuracy: 0.8670
Epoch 20/20
30/30 [==============================] - 1s 28ms/step - loss: 0.0060 - accuracy: 0.9988 - val_loss: 0.7451 - val_accuracy: 0.8647
```

fit을 통해 모델 학습, 에포크는 20, batch는 512

3.4 영화 리뷰 분류: 이진 분류 예제

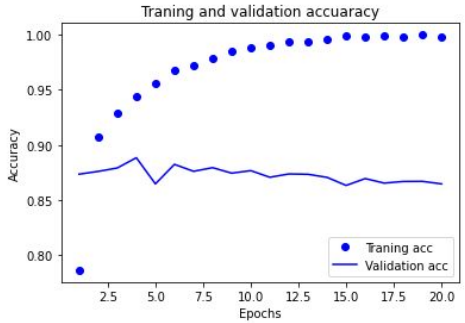
5) 검증셋을 통한 정확도, 손실 확인

```
In [27]: import matplotlib.pyplot as plt

history_dict = history.history
loss = history_dict['loss']
val_loss = history_dict['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Traning loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Traning and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



에포크에 따른 손실

```
In [34]: plt.clf()

acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']

plt.plot(epochs, acc, 'bo', label='Traning acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Traning and validation accuaracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



에포크에 따른 정확도

검증을 통해
학습시엔 에포크마다 눈에 띄게
손실이 감소하고 정확도는 증가하고
있지만 반대로 검증시 손실은
에포크마다 증가하고 검증셋
정확도는 에포크가 증가함에도 크게
증가하지 않는, 오히려 살짝 감소하는
경향을 볼 수 있다.

이로써 훈련데이터에 과도하게
최적화된 과적합이 일어났음을 알 수
있다.

과적합을 방지하기 위해 에포크를
줄여서 다시 모델 학습을
시도해본다.

3.4 영화 리뷰 분류: 이진 분류 예제

6) 새로운 모델 학습(에포크 변경 / 20->4)

```
In [38]: 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=['accuracy'])

         history = model.fit(x_train, y_train, epochs=4, batch_size=512)
         result = model.evaluate(x_test, y_test)

         Epoch 1/4
         49/49 [==============================] - 3s 16ms/step - loss: 0.5447 - accuracy: 0.7491
         Epoch 2/4
         49/49 [==============================] - 1s 16ms/step - loss: 0.2639 - accuracy: 0.9115
         Epoch 3/4
         49/49 [==============================] - 1s 14ms/step - loss: 0.1957 - accuracy: 0.9332
         Epoch 4/4
         49/49 [==============================] - 1s 15ms/step - loss: 0.1641 - accuracy: 0.9456
         782/782 [==============================] - 3s 3ms/step - loss: 0.3237 - accuracy: 0.8717

In [39]: result
Out[39]: [0.3236891329288827, 0.871680021286107]
```

같은 형식의 새로운 모델을 만들어 전체 **train** 데이터를 이용해 에포크가 **4**인 학습을 진행하고 테스트 데이터를 통해 평가하면 정확도가 **0.87**에 달성함을 확인할 수 있다.

```
In [40]: model.predict(x_test)
Out[40]: array([[0.13095096],
                [0.999897  ],
                [0.42740577],
                ...,
                [0.11287647],
                [0.03847513],
                [0.36533442]], dtype=float32)
```

학습된 모델로 **x_test**를 갖고 예측을 해보면 이와같은 확률들이 나온다.

3.5 뉴스 기사 분류: 다중 분류 문제

1) 로이터 데이터셋

```
from keras.datasets import reuters
(train_data, train_labels), (test_data, test_labels) = reuters.load_data(num_words=10000)
```

2) 데이터 준비

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

```
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)
one_hot_test_labels = to_one_hot(test_labels)
```

```
from keras.util.np_utils import to_categorical

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

```
to_categorical([0, 2])
[[1, 0, 0], [0, 0, 1]]
```

3.5 뉴스 기사 분류: 다중 분류 문제

3) 모델 구성

```
from keras import models
from keras import layers

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'])
```

4) 훈련 검증

a) 검증세트 준비 및 모델 훈련

```
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:]
```

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

loss : 훈련 손실값

acc : 훈련 정확도

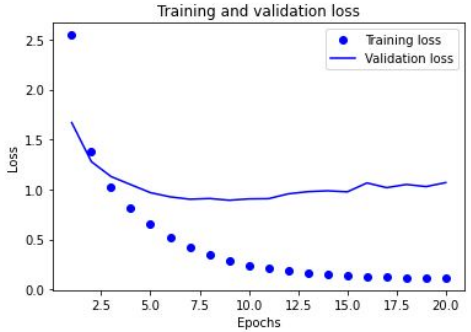
val_loss : 검증 손실값

val_acc : 검증 정확도

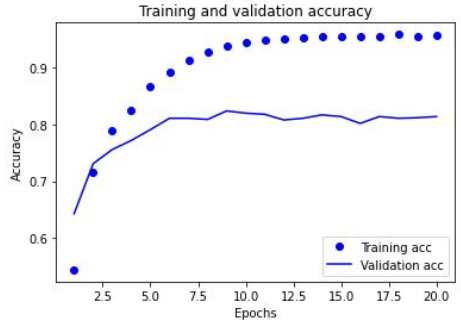
3.5 뉴스 기사 분류: 다중 분류 문제

5) 훈련과 검증 손실 및 정확도

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label = 'Training loss')
plt.plot(epochs, val_loss, 'b', label = 'Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```




```
plt.clf()

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']


plt.plot(epochs, acc, 'bo', label = 'Training acc')
plt.plot(epochs, val_acc, 'b', label = 'Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



3.5 뉴스 기사 분류: 다중 분류 문제

6) 에포크 수정 후 다시 훈련

```
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, batch_size =512, validation_data=(x_val, y_val))

result = model.evaluate(x_test, one_hot_test_labels)
print(result)
```

```
Train on 7982 samples, validate on 1000 samples
Epoch 1/9
7982/7982 [==============================] - 0s 61us/step - loss: 2.4849 - accuracy: 0.5277 - val_loss: 1.6664 - val_accuracy: 0.6400
Epoch 2/9
7982/7982 [==============================] - 0s 56us/step - loss: 1.3852 - accuracy: 0.7100 - val_loss: 1.2967 - val_accuracy: 0.7220
Epoch 3/9
7982/7982 [==============================] - 0s 58us/step - loss: 1.0540 - accuracy: 0.7705 - val_loss: 1.1400 - val_accuracy: 0.7580
Epoch 4/9
7982/7982 [==============================] - 0s 56us/step - loss: 0.8382 - accuracy: 0.8186 - val_loss: 1.0470 - val_accuracy: 0.7830
Epoch 5/9
7982/7982 [==============================] - 0s 58us/step - loss: 0.6690 - accuracy: 0.8576 - val_loss: 1.0310 - val_accuracy: 0.7760
Epoch 6/9
7982/7982 [==============================] - 0s 56us/step - loss: 0.5355 - accuracy: 0.8913 - val_loss: 0.9393 - val_accuracy: 0.8070
Epoch 7/9
7982/7982 [==============================] - 0s 56us/step - loss: 0.4330 - accuracy: 0.9112 - val_loss: 0.9258 - val_accuracy: 0.8060
Epoch 8/9
7982/7982 [==============================] - 0s 56us/step - loss: 0.3503 - accuracy: 0.9260 - val_loss: 0.9122 - val_accuracy: 0.8070
Epoch 9/9
7982/7982 [==============================] - 0s 57us/step - loss: 0.2879 - accuracy: 0.9374 - val_loss: 0.9014 - val_accuracy: 0.8190
2246/2246 [==============================] - 0s 41us/step
[0.987105168302677, 0.7871772050857544]
```

3.5 뉴스 기사 분류: 다중 분류 문제

7) 정리

a) N개의 클래스로 데이터 포인트를 분류하려면 네트워크의 마지막 층의 크기는 N이어야한다.

b) 단일 레이블, 다중 분류 문제에서는 N개의 클래스에 대한 확률 분포를 출력하기 위해서는 softmax 활성화 함수를 사용해야 한다.

c) 이런 문제는 항상 범주형 크로스엔트로피를 사용해야 한다. 이 함수는 모델이 출력한 확률 분포와 타깃 분포 사이의 거리를 최소화한다.

d) 다중 분류에서 레이블 다루는 방법 2가지

i) 원-핫 인코딩

ii) 정수 텐서로 변환하는 방법

e) 충분히 큰 중간층을 두어야 한다.

3.6 주택 가격 예측 : 회귀 문제

- 개별적인 레이블 대신(분류 x) 연속적인 값 예측
- ex) 기상데이터, 내일기온 예측
- ex) 소프트웨어 명세, 소프트웨어 프로젝트가 완료될 시간 예측
- 회귀와 로지스틱 회귀는 다른것! (로지스틱 회귀는 분류문제)

3.6 주택 가격 예측 : 회귀 문제

1) 데이터셋 load

```
from keras.datasets import boston_housing
(train_data, train_targets), (test_data, test_targets) = boston_housing.load_data()
```

2) 데이터 준비(numpy 이용)

   a) 데이터 특성별로 **정규화** 작업 필요

   b) 각 특성(열)에 대해서 특성의 **평균을 빼고 표준편차로 나눔**

   c) 특성의 중앙이 0근처에 맞춰지고 표준편차가 1이 됨

```
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) 모델 구성

   a) 데이터 크기 작음->64개의 유닛, 2개 은닉층

   b) 과대적합 일어나기 쉬움. 작은 모델 이용해야.

   c) 마지막 층 model.add(layers.Dense(1)) -> 선형 모델 출력 -> 스칼라 회귀
   활성함수를 적용하게 되면 0과 1사이의 값을 예측됨
   마지막층에 제한을 두지 않음으로써 자유롭게 값을 예측하도록 함.

   d) 손실함수 MSE , 평가 지표 MAE(평균절대오차, 예측과 타깃 사이 거리의 절댓값)

3.6 주택 가격 예측 : 회귀 문제

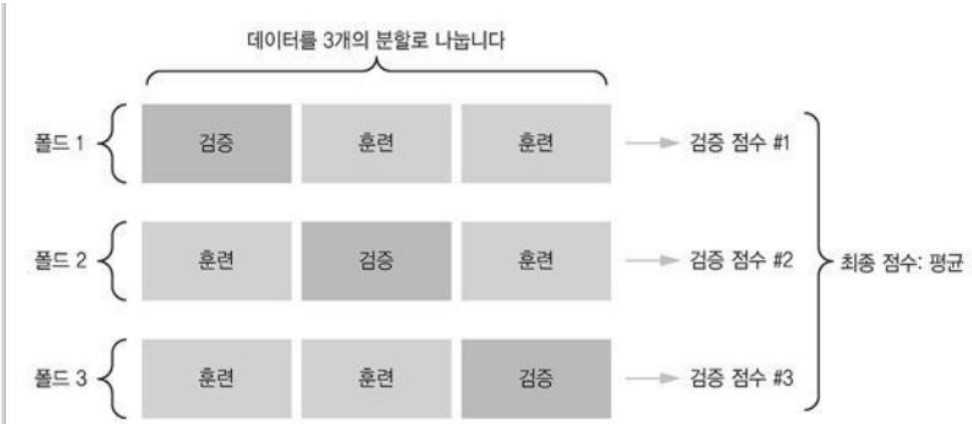
3) 모델 구성

```
from keras import models
from keras import layers

def build_model():
    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))  선형모델 출력
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])  손실함수 MSE , 평가 지표 MAE
    return model
```

3.6 주택 가격 예측 : 회귀 문제

4) K-겹 검증을 사용한 훈련 검증(K-fold cross-validation)

a)



b) 각 폴드 별로 검증 점수를 매긴 다음 구한 점수들의 평균을 내어 최종 점수를 도출.

c) 각 폴드 안에 k (보통 4, 5) 로 나누어 검증데이터를 그 중 하나로 결정

d) 검증데이터의 정확도를 위함

3.6 주택 가격 예측 : 회귀 문제

4) K-겹 검증을 사용한 훈련 검증(K-fold cross-validation)

```
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()
    print(partial_train_data.shape)
    print(partial_train_targets.shape)

    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)
```

검증데이터 i 즉 k번째

Val_mae : 검증 점수

검증세트로 모델 평가

```
1  all_scores
```

```
[1.954390287399292, 2.891843795776367, 2.943552255630493, 2.479199171066284]
```

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

```
2.567246377468109
```

결과 : 평균

3.6 주택 가격 예측 : 회귀 문제

4) K-겹 검증을 사용한 훈련 검증(K-fold cross-validation)

```
num_epochs = 500
all_mae_histories = []
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()
    history = model.fit(partial_train_data, partial_train_targets,validation_data=(val_data, val_targets),
                        epochs = num_epochs, batch_size = 1, verbose=0)
    print(history.history.keys())
    mae_history = history.history['val_mae']
    all_mae_histories.append(mae_history)
```

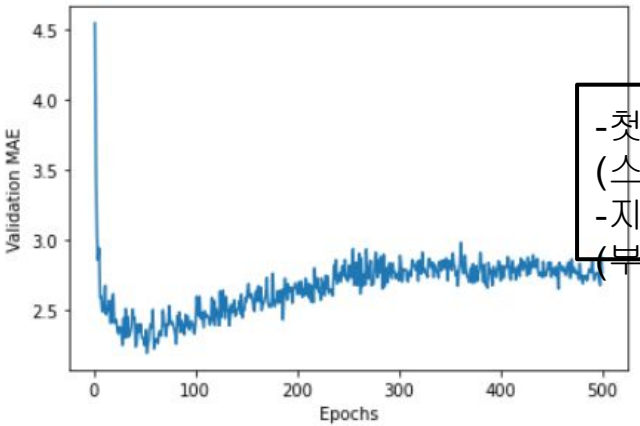
학습 이력
epoch 마다의 값들이 저장됨

3.6 주택 가격 예측 : 회귀 문제

4) K-겹 검증을 사용한 훈련 검증(K-fold cross-validation)

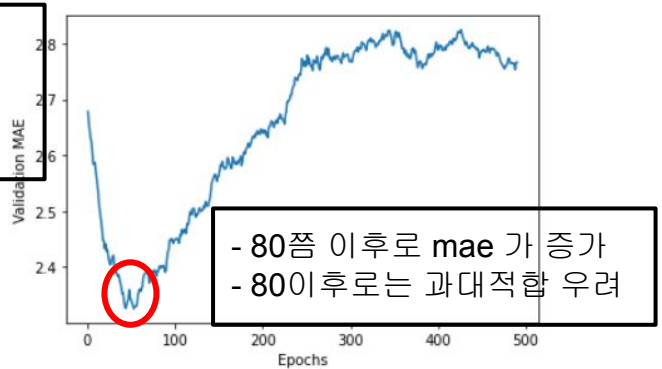
```
average_mae_history = [np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

```
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()
```




```
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()
```

-첫 10개 포인터 제외
(스케일차이 너무 큼)
-지수 이동 평균
(부드러운곡선)



- 80쯤 이후로 mae 가 증가
- 80이후로는 과대적합 우려

3.6 주택 가격 예측 : 회귀 문제

4) K-겹 검증을 사용한 훈련 검증(K-fold cross-validation)

최종 결과

```
model = build_model()
model.fit(train_data, train_targets, epochs=80, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)

test_mae_score
```

```
4/4 [==============================] - 0s 3ms/step - loss: 17.2296 - mae: 2.6256
2.625624895095825
```

```
model = build_model()
model.fit(train_data, train_targets, epochs=80, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)

test_mae_score
```

```
4/4 [==============================] - 0s 3ms/step - loss: 16.7698 - mae: 2.5836
2.5836360454559326
```

3.6 주택 가격 예측 : 회귀 문제

5) 정리
   a) 회귀는 분류와 달리 **평균제곱오차(MSE)**라는 손실함수를 사용
   b) 회귀는 분류와 달리 **평균절대오차(MAE)**라는 평가지표를 사용. **정확도 사용 x**
   c) 특성이 다른 범위를 가지면 전처리 단계에서 **스케일 조정**해야함
   d) 가용데이터가 적다면 **K겹 검증** 이용
   e) 가용데이터가 적다면 은닉층의 수를 줄여야 과대적합을 피할 수 있음(1 or 2개)