인공지능 응용 프로그래밍

학번	20191783
이름	임현아

1. 기초 문법

1-1. List

1-2. Tuple

1-3. Dictionary

1-4. List, Tuple, Dictionary 비교

2. Tensorflow

2-1. Tensor

2-2. Session

1. 기초 문법

AI 관련 내용을 알기 위해서는 파이썬에 대한 기본 지식이 필요하다. 이 포트폴리오는 AI 를 설명하기 위해 작성하는 것으로 기본적인 자료형, for문 또는 if 문법은 생략한다. 파이썬에서 주로 볼 수 있는 List, Tuple, Dictionary 의 개념만 설명할 것이다.

1-1. List

리스트는 [...] 로 콤마로 구분된 항목(또는 원소)들을 표현한 것이다. 항목은 정수, 실수, 문자열, 리스트 등이 모두 가능하다. 항목의 순서는 의미가 있으며 항목 자료값은 중복되어도 무관하다.

리스트 구조
리스트 이름 = [인자, 인자, 인자, 인자]
int = [1, 3, 5, 7, 9] float = [1.4, 2.7, 3.9] string = [고양이, 코끼리, 돼지] list = [[1, 3, 5, 7, 9], [1.4, 2.7, 3.9]]

1-2. Tuple

튜플은 (...) 을 사용하여 콤마로 항목을 구분하여 표현한다. 항목의 순서나 내용의 수정이 불가능하다.

튜플 구조
튜플 이름 = (인자, 인자, 인자, 인자)
int = (1, 3, 5, 7, 9) float = (1.4, 2.7, 3.9) string = (고양이, 코끼리, 돼지) tuple = ((1, 3, 5, 7, 9), (1.4, 2.7, 3.9))

1-3. Dictionary

딕셔너리는 키와 값이 쌍인 항목을 나열한 시퀀스로 항목은 키:값, 전체는 중괄호 {...} 를 사용한다. 콤마로 구분된 항목(또는 요소, 원소)들의 리스트로 표현된다.

딕셔너리 구조
딕셔너리 이름 = {키:값, 키:값, 키:값, 키:값}
int = {'일': 1, '삼': 3, '오' : 5} float = {'일점사' : 1.4, '이점칠': 2.7} string = {'고양이': 4, '코끼리':4, '돼지' : 4}

1-4. List, Tuple, Dictionary 비교

	List	Tuple	Dictionary
전체	[...]	(...)	{ ... }
항목	전체	전체	전체
순서	O	X	X
중복	O	O	O
형태	항목, 항목	항목, 항목	키:값
수정	O	X	O

2. Tensorflow

텐서플로란 구글에서 만든 연구 및 프로덕션용 오픈소스 딥러닝 라이브러리이다. 딥러닝 프로그램을 쉽게 구현할 수 있도록 다양한 기능을 제공하며 테스크톱, 모바일, 웹, 클라우드 개발용 API를 제공하여 라이브러리로써 뛰어난 기능을 제공해준다. 지원하는 언어는 Python, Java, Go 등을 지원하며 파이썬을 최우선으로 지원하여 파이썬으로 개발할 경우 편리하게 사용할 수 있게 도와준다. 필자가 사용할 도구이기도 하여 간단히 설명할 것이다.

2-1. Tensor

텐서, 딥러닝에서 데이터를 표현하는 방식이다.

0-D 텐서 : 스칼라 (차원이 없는 텐서)
1-D 텐서 : 벡터 (1차원 텐서)
2-D 텐서 : 행렬 (2차원 텐서)

행렬: 하나의 채널에 2차원 행렬(배열)로 표현하며 회색조 이미지로 표현된다.
형식: [[1, 2, 3], [4, 5, 6]]

n차원 행렬(배열): 텐서의 차원을 텐서의 rank (순위)라고 부른다. R, G, B 각 3개의 채널마다 2차원 행렬(배열)로 표현하여 RGB 이미지로 보여준다. 텐서는 행렬로 표현할 수 있는 n 차원 형태의 배열을 높은 차원으로 확장할 수 있다.
형식: [[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]]

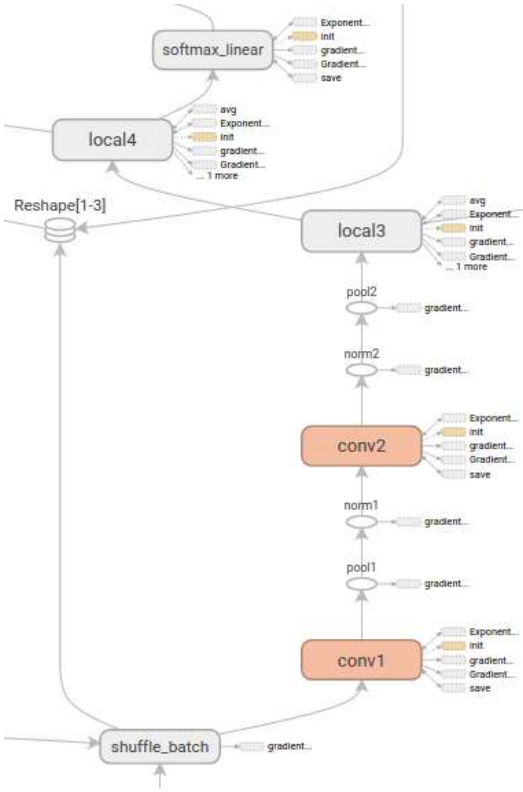
2-2. Session

텐서플로에서 텐서 계산 과정은 모두 그래프라고 부르는 객체 내에 저장되어 실행된다. 그래프를 계산하려면 외부 컴퓨터에 이 그래프 정보를 전달하고 결과 값을 받아야 한다.

Session은 이런 계산 과정을 하기 위해 통신을 담당하는 객체이다. Session은 생성, 사용, 종류 세 과정으로 이루어져 있다.

```
x = tf.constant(3)
y = x**2

session = tf.Session()          //Session 생성
print(session.run(x))           //Session 사용
print(session.run(y))           //Session 사용
session.close()                 //Session 종료
```

Session 생성은 tf.Session()을 사용하여 생성한다.
Session 사용은 run 매서드에 그래프를 입력하면 출력 값을 계산하여 반환한다.
Session 종료는 close 메서드를 사용하여 with 문을 사용하면 명시적으로 호출이 불필요해진다.



2-3. 데이터 흐름 그래프

텐서 형태의 데이터들이 딥러닝 모델을 구성하는 연산들의 그래프를 따라 흐르면서 연산이 일어나게 된다.

3. Colaboratory(Colab) 기본 문법

필자는 코랩을 사용하여 프로그래밍할 것이다. 앞으로 사용하게 될 코랩의 기본 문법을 설명할 것이다.

3-1. 버전 사용방법

프로그래밍하기 전, import 하기 전에 텐서플로 버전을 선택해준다. 버전을 선택하는 명령어는 아래와 같다.

```
[1]  %tensorflow_version 1.x #텐서플로 1버전

     TensorFlow 1.x selected.
```

```
[2]  %tensorflow_version 2.x #텐서플로 2버전

     TensorFlow 2.x selected.
```

```
[3]  try:
      %tensorflow_version 1.x #텐서플로 1버전
     except Except:
       pass

     TensorFlow 1.x selected.
```

```
[4]  #텐서플로 사용버전 확인

     import tensorflow as tf
     tf.__version__

     '1.15.2'
```

%tensorflow_version은 텐서플로 버전을 선택한다는 것이며 그 뒤 숫자는 텐서플로 몇 버전을 사용할 것인지 선택하는 것이다. 2버전 중 1버전으로 변경할 때엔 명령어 없이 ctrl + M 을 누르면 바로 1버전이 실행된다.

3-2. Shape, Type

```
[3]  a = tf.constant([1, 2, 3])
     a.shape

     TensorShape([3])
```

```
[4]  a = tf.constant([[1, 2, 3], [4, 5, 6]])
     a.shape

     TensorShape([2, 3])
```

```
[5]  a = tf.constant([[[1, 2, 3], [4, 5, 6]],[[1, 2, 3], [4, 5, 6]]])
     a.shape

     TensorShape([2, 2, 3])
```

Shape는 행렬의 개수를 의미한다. tf.constant로 설정해두었던 행렬의 개수가 알고 싶다면 사용하는 명령어이다. n 차원인 행렬은 TensorShape([...]) 안에 n개의 숫자가 들어가게 된다.

```
[6]  a

     <tf.Tensor: shape=(2, 2, 3), dtype=int32, numpy=
     array([[[1, 2, 3],
             [4, 5, 6]],

            [[1, 2, 3],
             [4, 5, 6]]], dtype=int32)>
```

변수만 입력하고 실행하게 된다면 어떻게 될까? 변수만 입력한 후 실행하게 된다면 Type 형식으로 변수에 입력하였던 정보가 출력된다. 변수에 입력되어있는 shape, dtype, numpy, 행렬 값들을 알아낼 수 있다.

3-3. 배열 텐서

```
[9]  x = tf.constant([1, 2, 3])
     y = tf.constant([5, 6, 7])

     print((x+y).numpy())
```

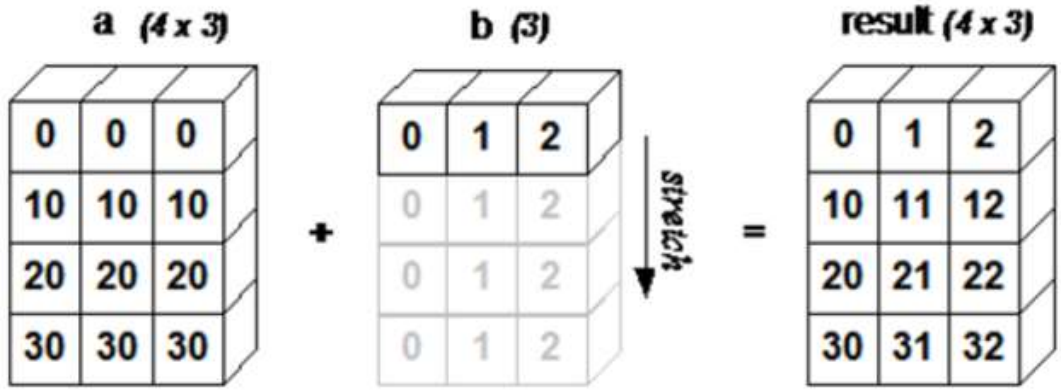
```
[ 6  8 10]
```

```
[10] a = tf.constant([5], dtype=tf.float32)
     b = tf.constant([10], dtype=tf.float32)
     c = tf.constant([2], dtype=tf.float32)
     print(a.numpy())

     d = a * b + c

     print(d)
     print(d.numpy())
```

```
[5.]
tf.Tensor([52.], shape=(1,), dtype=float32)
[52.]
```

배열 텐서는 C 나 Java를 사용하였을 때처럼 동일하게 방식으로 사용된다. 참고로 numpy() 라는 함수는 값만 출력하고 싶을 때 사용하는 명령어이다.

텐서의 브로드캐스팅은 shape이 다르더라도 연산이 가능하도록 한 것이다. 즉 가지고 있는 값을 이용하여 shape을 맞춘다.



[브로드캐스팅 -1] 동일한 행 개수를 가진 열이 다른 행렬 연산

[브로드캐스팅 -1]처럼 a 는 4x3 배열로 2차원 배열, b 는 1x3 배열로 1차원 배열을 이루고 있다. 이렇게 차원은 다르지만 배열의 행이나 열의 개수가 동일하다면 차원이 더 작은 배열이 행이나 열의 개수를 동일하게 하여 연산을 가능케 한다. [브로드캐스팅 -1] 은 행을 동일하기에 동일한 행 여러 개를 붙여 열을 동일하게 만들어준다. b 배열에 희미하게 그려져 있는 것이 행을 여러 개를 붙여 열을 동일하게 만들었다는 것을 가이드로 보여주는 것이다. a 와 b 배열이 연상이 가능하게 하였다.

[브로드캐스팅 -1] 을 코드로 만들면 하단 코드가 된다. a와 b의 행과 열의 개수가 다름에도 불구하고 연산이 가능하며 위 그림과 동일한 값이 나오는 것을 볼 수 있다.

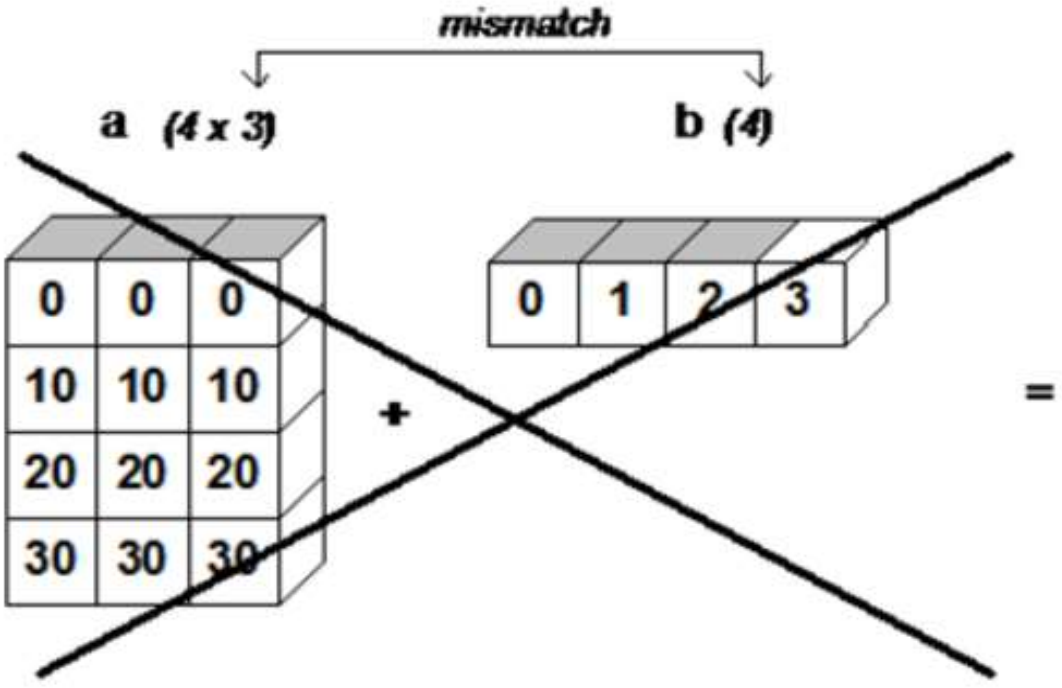
```
[9]  a = tf.constant([[0, 10, 20, 30], [0, 10, 20, 30], [0, 10, 20, 30]])
     b = tf.constant([[0], [1], [2]])


     print((a+b).numpy())
```

```
[[ 0 10 20 30]
 [ 1 11 21 31]
 [ 2 12 22 32]]
```

[브로드캐스팅 코드-1] 브로드캐스팅-1 코드

만약 행과 열 둘다 다른 값이라면 어떻게 될까?

[브로드캐스팅 -2]는 그 궁금증을 없애주는 그림이다. 브로드캐스팅 연산을 하기 위한 행렬의 조건은 동일한 개수의 열을 가지거나 동일한 개수의 행을 가져야만 한다. [브로드캐스팅-2] 에서 a는 4x3 배열, b는 1x4 배열이다. a 배열의 열과 b 배열의 행은 동일한 개수이지만 열의 개수나 행의 개수가 동일하지 않다.



[브로드캐스팅 -2] 행과 열 개수가 다른 행렬의 연산

이런 경우를 코드로 작성해보면 [브로드캐스팅 코드-2] 가 된다. InvalidArgumentEr

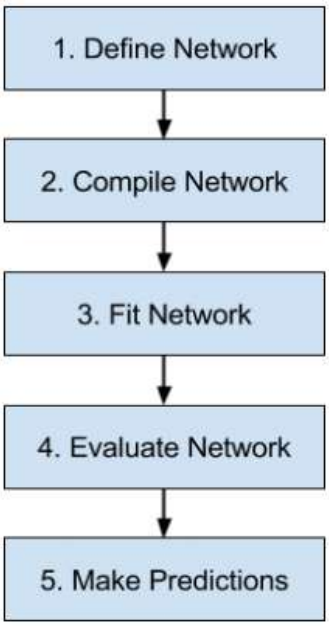
ror로 에러가 발생한 것을 확인할 수 있다. 이 에러는 열과 행이 동일하지 않으면 나타나는 에러로 연산을 불가능하다는 것을 확인할 수 있다.

```
[11]  a = tf.constant([[0, 10, 20, 30], [0, 10, 20, 30], [0, 10, 20, 30]])
      b = tf.constant([[0], [1], [2], [3]])


      print((a+b).numpy())
```

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-11-0effdf1b7d04> in <module>()
      3
      4
----> 5 print((a+b).numpy())

                                  5 frames
/usr/local/lib/python3.6/dist-packages/six.py in raise_from(value, from_value)

InvalidArgumentError: Incompatible shapes: [3,4] vs. [4,1] [Op:AddV2]

SEARCH STACK OVERFLOW
```

[브로드캐스팅 코드-2] 브로드캐스팅-2 코드

3-4. 딥러닝 구현 방법



딥러닝 구현을 하기 위해서는
1. 딥러닝 모델 설계 (Define)

2. 주요 훈련방법을 설정 (Complie)
- 최적화 방법
- 손실 함수
- 훈련 모니터링 지표

3. 훈련 (fit)

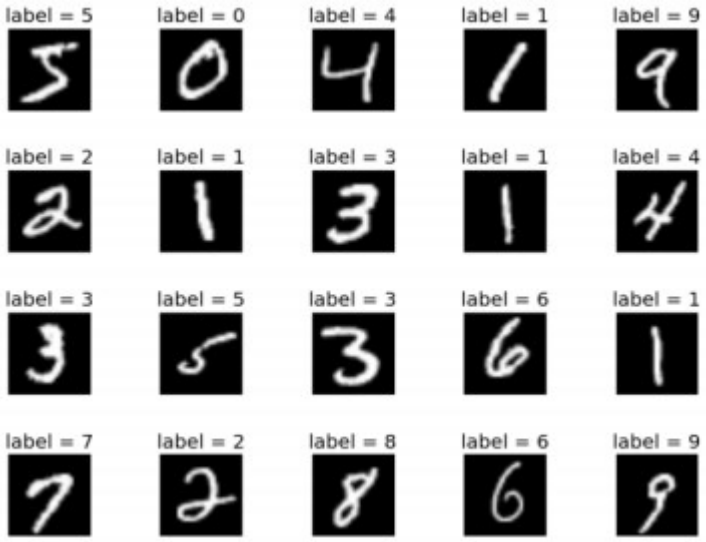
4. 테스트 데이터를 평가(evaluate)

5. 정답을 예측(predict)

[딥러닝구현] 구현방법

4. MNIST 데이터 셋

MNIST는 미국 국립 표준 기술원(NIST)에서 만든 딥러닝 손글씨 인식에 사용되는 데이터 셋이다. 28x28 로 784 픽셀에서 손글씨를 작성하게 된다. 내부 값이 0~255가 되는데 이 값을 0~1로 수정하여 사용한다. 숫자가 높으면 높을수록 더 진하게 출력된다. MNIST는 [MNIST 레이블] 사진처럼 필기 숫자 이미지와 필기 숫자 이미지의 정답인 레이블의 쌍으로 구성된다. 숫자의 범위는 0~9까지 총 10개의 패턴을 의미한다.



[MNIST 레이블] 숫자 이미지와 레이블의 관계

4-2. 딥러닝 과정

딥러닝 과정에서 중요한 것을 모델을 구성하고 훈련하고 예측하는 것이다. 모델은 딥러닝 핵심 신경망으로 완전연결층으로 1차원 배열로 평탄화하게 구성되어 있다. 모델 구성은 블랙박스로 해주고 모델 훈련은 모델이 문제를 해결하도록 훈련시키고 학습 방법 및 모니터링 지표를 설정해준다.

```
#MNIST 형태를 알아 봅시다. 데이터 수, 행렬 형태 등
print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
```

```
(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
```

MNIST에는 총 7000개의 데이터가 있다. 훈련 데이터 세트는 60000개, 테스트 데이터 세트는 10000개가 있다. 훈련 세트에만 학습에 사용되며 테스트 데이터로 학습 데이터가 들어온다.

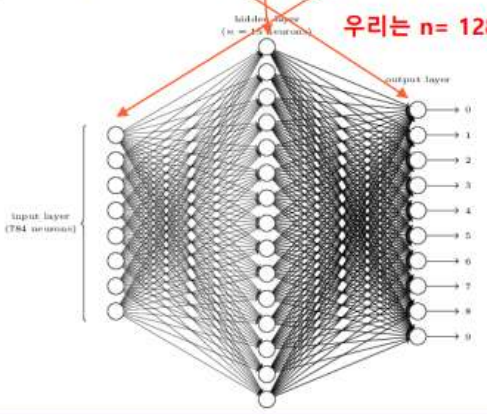
MNIST 딥러닝 과정은 필요 모듈 임포트, 훈련과 정답 데이터 지정, 모델 구성, 학습에 필요한 최적화 방법과 손실 함수 등을 설정하며 생성된 모델로 훈련 데이터를

학습시켜 테스트 데이터로 성능을 평가한다.

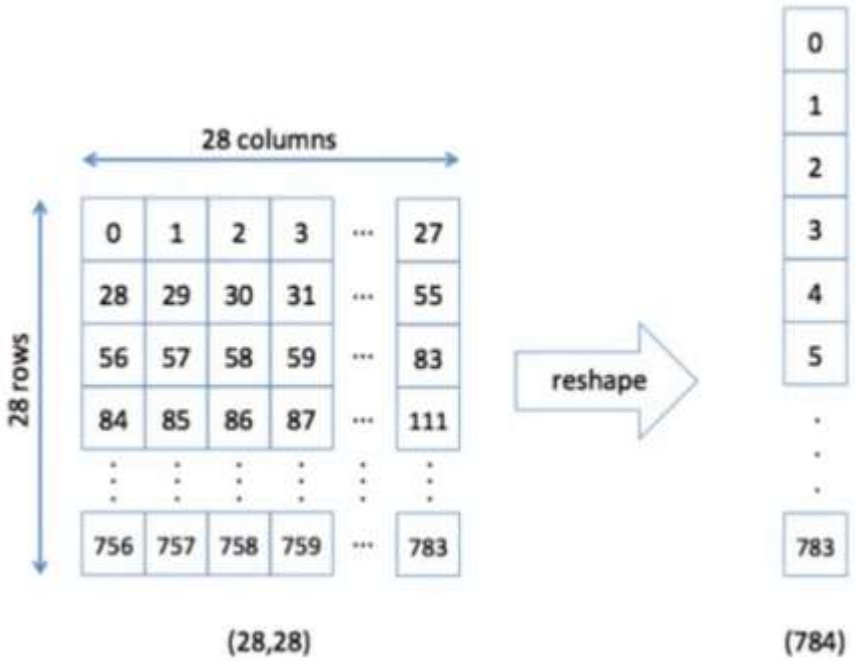
```
[3]  import tensorflow as tf

     mnist = tf.keras.datasets.mnist
     # MNIST 데이터셋을 훈련과 테스트 데이터로 로드하여 준비
     (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

① MNIST 데이터 셋 로드하여 준비하기, 전처리라고도 부르는 이 준비는 샘플 값을 정수에서 부동소수로 변환하여 한 비트의 값을 255로 나눈다. 정규화 결과, 픽셀 값은 0에서 1 사이의 값이다.

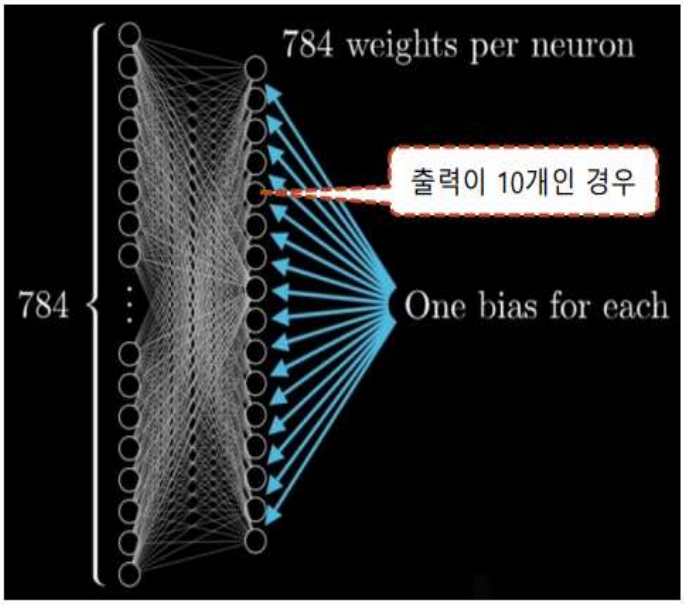
```
# 층을 차례대로 쌓아 tf.keras.models.Sequential 모델을 생성
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])
```



② 모델 구성, 층을 차례대로 쌓아 tf.keraas.models.Sequential 모델을 생성한다. 신경망을 구성하는데 층을 쌓아 만드는 것이다. 대략적인 설명을 위해 그림을 참고하자. 이 그림은 인공 신경망(Neural Networks)라고 한다. 입력층과 중간층(은닉층), 출력층으로 이루어져 있다. 입력층, 출력층은 각각 28개로, 중간층(은닉층)은 128로 데이터를 표현할 수 있는 무리를 만든다.



2차원 그림을 1차원으로 평탄화, [MNIST레이블]에서는 손글씨로 쓰여져 있으며 보기 쉽게 28x28 구조를 하고 있었다. 28x28 구조는 2차원 그림을 한줄로 늘어트린 것을 1차원이라고 한다.



중간 은닉층이 없는 구조도 존재한다. 단순 신경망 모델, Dense()층이라고 한다. 완전연결층이라고 하는 이 층은 층을 차례대로 쌓아 모델을 생성한다. 입력은 이미지의 각 픽셀값, 출력은 각 위치의 값이 될 크기로 써 중간층이 없어도 되는 구조를 이루고 있다.

③ 학습에 필요한 최적화 방법과 손실 함수 등을 설정, 우선 훈련에 사용할 옵티

마이저와 손실 함수등을 선택한다. model.summary() 로 각 층의 구조와 패라미터 수를 표시한다.

④ 모델을 훈련, model.fit() 명령어는 모델을 훈련할 수 있도록 하는 함수이다.

4-3. MNIST 기본 설명

```
[3]  import tensorflow as tf

     mnist = tf.keras.datasets.mnist
     # MNIST 데이터셋을 훈련과 테스트 데이터로 로드하여 준비
     (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

[MNIST모델설계] 훈련, 테스트 데이터 모델 설계

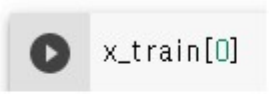
[MNIST모델설계] 그림에서 x와 y의 훈련과 테스트 데이터 셋을 만든다. 훈련 데이터를 통해 테스트 데이터가 쌓여 만들어진다.

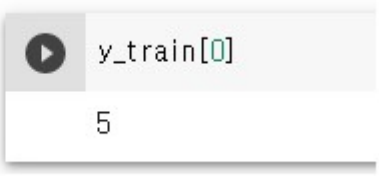
```
[4]  x_train.shape

     (60000, 28, 28)
```

```
[5]  y_train.shape

     (60000,)
```

[MNIST데이터] MNIST 데이터 확인

[MNIST데이터]에서 볼 수 있듯이 훈련 데이터가 60000개로 잘 만들어졌는지 확인해준다. x_train 옆에 있는 28이란 숫자는 숫자를 그릴 픽셀의 개수를 뜻한다.





훈련 데이터 내부도 살펴보자. x는 그림을, y는 x가 무슨 숫자인지 보여주는 역할을 한다.

```
#MNIST 데이터셋을 훈련과 테스트 데이터로 로드하여 준비
(x_train, y_train), (x_test, y_test) = mnist.load_data()

#MNIST 형태를 알아 봅시다. 데이터 수, 행렬 형태 등
print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)

#MNIST 훈련 데이터의 내부 첫 내용도 알아보자.
print(x_train[0])
print(y_train[0])

#MNIST 테스트 데이터의 내부 첫 내용도 알아보자.
print(x_test[0])
print(y_test[0])
```

[MNIST데이터확인] MNIST 데이터 확인

정리하자면 MNIST 데이터 셋을 훈련과 테스트 데이터로 로드하여 준비한 다음, MNIST 형태를 알아보는 명령문을 사용하여 데이터 수와 행렬의 형태를 알아본다.

```
#MNIST 형태를 알아 봅시다. 데이터 수, 행렬 형태 등
print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
```

```
(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
```

MNIST 형태는 훈련은 60000개의 데이터를 가지고 있으며 가로 28, 세로 28의 총 784개의 정수 값을 가진다. 테스트는 10000개의 데이터를 가지고 있으며 훈련 데이터와 동일하게 가로 28, 세로 28의 총 784개의 정수 값을 가진다.

```
#MNIST 훈련 데이터의 내부 첫 내용도 알아보자.
print(x_train[0])
print(y_train[0])
```

[x_train[0]] x_train[0] 그림

[y_train[0]] y_train[0]

훈련 데이터의 내부는 이렇게 생겼다. x는 숫자를 모아 5를 그린 것처럼 생겼으며 y는 그 숫자가 5라는 것을 알려준다.

```
#MNIST 테스트 데이터의 내부 첫 내용도 알아보자.
print(x_test[0])
print(y_test[0])
```

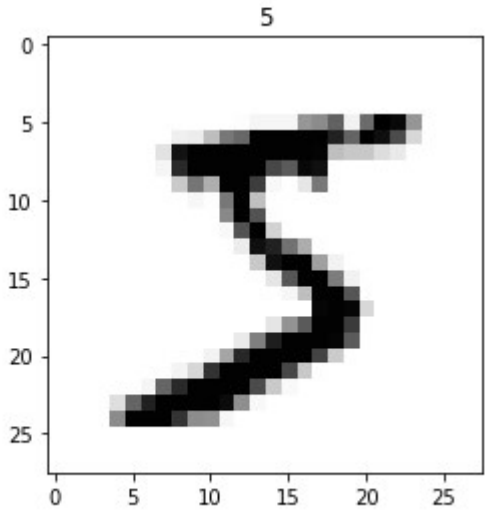
[x_test[0]] x_test[0] 그림

[y_test[0]] y_test[0]

테스트 데이터의 내부는 이렇게 생겼다. x는 숫자를 모아 7을 그린 것처럼 생겼으며 y는 그 숫자가 7이라는 것을 보여준다.

```
[12]  import matplotlib.pyplot as plt

      n = 0
      ttl = str(y_train[n])
      plt.figure(figsize=(6, 4))
      plt.title(ttl)
      plt.imshow(x_train[n], cmap='Greys')
```

```
<matplotlib.image.AxesImage at 0x7f7fd2276470>
```

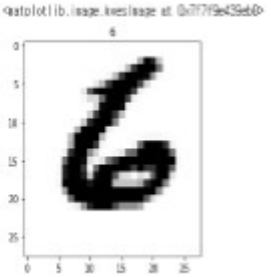


[MNIST 손글씨(1)] MNIST 손글씨(1)

MNIST는 숫자를 모아 멀리서 보면 숫자와 비슷하게 생기도록 그리는 것도 있지만 손글씨를 그리는 것도 있다. [MNIST 손글씨(1)]는 손글씨 위에 해당 숫자가 무슨 숫자인지 볼 수 있는 레이블이 써있으며 하단에는 28x28 픽셀 안에 그려진 숫자 그림을 확인할 수 있다. 위 그림은 y_train[0] 이기 때문에 5를 그린 것을 확인할 수 있다.

그렇다면 만약 str(y_train[n]) 이나 len(x_train) 에 사칙연산을 넣어보면 어떻게 될까? 답은 간단하다. 사칙연산을 넣은 그대로 레이블에 그릴 수 있으면 그려 보여준다.

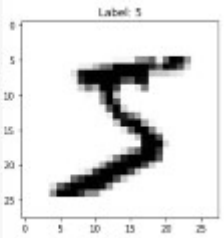
```
n = len(y_test)-1
ttl = str(y_test[n])
plt.figure(figsize=(6, 4))
plt.title(ttl)
plt.imshow(x_test[n], cmap='Greys')
```



4-4. MNIST 기본 정리

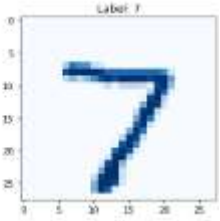
이제 기본적인 MNIST 설명을 끝마치며 기본에 대해 다시 정리해보자. 그림 하나마다 간략한 설명을 할 것이다.

```
import matplotlib.pyplot as plt

tmp = "Label: " + str(y_train[0])
plt.title(tmp)
plt.imshow(x_train[0], cmap="Greys")
plt.show()
```



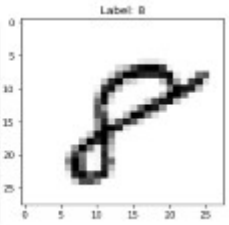
첫 그림은 y_train[0]인 것을 찾아 손글씨로 보여주는 것이다. str(y_train[0])은 5를 뜻한다. imshow 는 imshow(그림, camp="색상") 으로 정해진 색상으로 그림을 그리는 명령문이다. x_train[0]은 5를 뜻한다. 출력되는 것을 'Label: 5' 문구와 하단의 회색 색상으로 그린 5 숫자이다.

```
tmp = "Label: " + str(y_test[0])
plt.title(tmp)
plt.imshow(x_test[0], cmap="Blues")
plt.show()
```



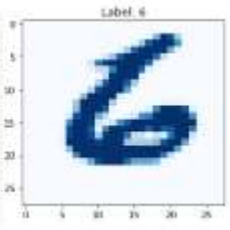
y_test[0]은 7을 의미하며 x_test[0]도 7을 뜻한다. 출력되는 것을 'Label: 7' 문구와 하단의 회색 색상으로 그린 7 숫자이다.이 코드에서 중요한 것을 camp의 색상이 회색이 아닌 파란색이라는 것이다.

```
idx = len(x_train)-1
tmp = "Label: " + str(y_train[idx])
plt.title(tmp)
plt.imshow(x_train[idx], cmap='Greys')
plt.show()
```



x_train 은 5이다. y_train[4]인 숫자와 숫자의 그림을 그리는 것이 이 코드의 목적이다. 옆에서 그려진 것을 보았을 때, x_train[4], y_train[4]는 숫자 8이라는 것을 알 수 있다.

```
idx = len(x_test)-1
tmp = "Label: " + str(y_test[idx])
plt.title(tmp)
plt.imshow(x_test[n], cmap='Blues')
plt.show()
```



x_test는 7을 의미한다. 고로 y_test[6] 을 의미한다. 이 코드의 목적은 x_test[6]과 y_test[6]의 숫자와 숫자 그림을 알아내는 것이다. 오른쪽에 그려진 그림을 보았을 때, x_train[6]과 y_test[6]은 숫자 6 이라는 것을 알 수 있다.

4-5. MNIST 심화 설명

이제 기본적인 MNIST 는 알게 되었을 것이다. 이젠 조금 더 심화하게 들어가 MNIST 에 대해 더 알아보도록 하자. 테스트 데이터의 첫 번째 손글씨 예측 결과를 확인해보도록 하자. model.predict(input)은 손글씨를 배열로 입력하여 그 숫자만 보여주는 명령어이다.

```
[33]  1 # 테스트 데이터의 첫 번째 손글씨 예측 결과를 확인
      2 print(x_test[:1].shape)
      3
      4 pred_result = model.predict(x_test[:1])
      5 print(pred_result.shape)
      6 print(pred_result)
      7 print(pred_result[0])
```

```
(1, 28, 28)
(1, 10)
[[8.7629097e-12 4.7056760e-14 2.5735870e-12 1.3529770e-07 1.9923079e-21
  1.6554103e-12 2.3112234e-21 9.9999988e-01 2.5956004e-10 3.6446388e-10]]
[8.7629097e-12 4.7056760e-14 2.5735870e-12 1.3529770e-07 1.9923079e-21
 1.6554103e-12 2.3112234e-21 9.9999988e-01 2.5956004e-10 3.6446388e-10]
```

[MNIST심화] MNIST 심화

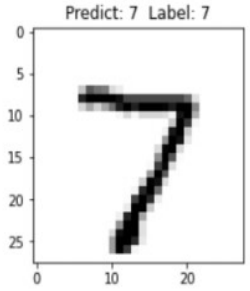
손글씨로 써주기 위해 모양을 28x28 로 잡아주었다. 결과는 1~10의 이차원 배열로 결과는 10개의 0~1의 실수이다. [MNIST심화]에서 나온 배열의 값은 0~1의 값으로 10개의 합이 1이다.

```
import numpy as np

# 10 개의 수를 더하면?
one_pred = pred_result[0]
print(one_pred.sum())

# 혹시 가장 큰 수가 있는 첨자가 결과
one = np.argmax(one_pred)
print(one)

import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(5, 3))
tmp = "Predict: " + str(one) + "  Label: " + str(y_test[0])
plt.title(tmp)
plt.imshow(x_test[0], cmap='Greys')
```

10개의 합은 확률은 언제나 1이니 1로 출력되어야 한다. 가장 큰 수가 있는 첨자의 결과를 알려는 명령어를 통해 이 코드의 가장 큰 수가 있는 첨자의 결과를 살펴본다. y_test[0] 이 7을 가르켰으니 7을 확인한다. 하단의 그림과 동일한지 확인해보는 것이 제일 중요하다.

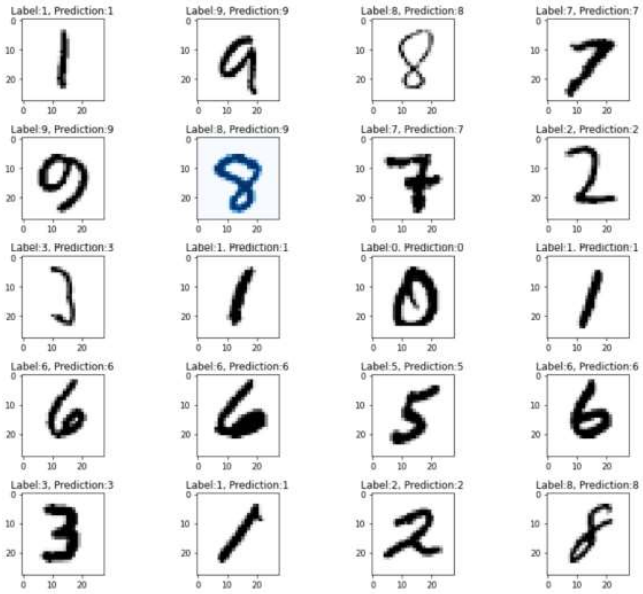
```
1.0
7
<matplotlib.image.AxesImage at 0x7f4a7890f2b0>
```



4-5. MNIST 심화 정리

```
# 임의의 20개 그리기
count = 0
plt.figure(figsize=(12,10))
for n in samples:
    count += 1
    plt.subplot(nrows, ncols, count)
    # 예측이 틀린 것은 파란색으로 그리기
    cmap = 'Greys' if ( pred_labels[n] == y_test[n]) else 'Blues'
    plt.imshow(x_test[n].reshape(28, 28), cmap=cmap, interpolation='nearest')
    tmp = "Label:" + str(y_test[n]) + ", Prediction:" + str(pred_labels[n])
    plt.title(tmp)

plt.tight_layout()
plt.show()
```

상단의 그림은 MNIST를 20개 그리는 코드이다. 예측과 레이블이 동일하다면 회색으로, 틀리다면 파란색으로 그리게 만드는 코드이다. 하단 그림은 코드를 돌린 이후의 모습이다. 파란색으로 되어있는 것은 예측과 레이블이 다르기 때문에 그려진 것이다. 예측은 9로 레이블은 8로 나와있기 때문에 틀렸다고 판단하고 파란색으로 그려준 것이다.

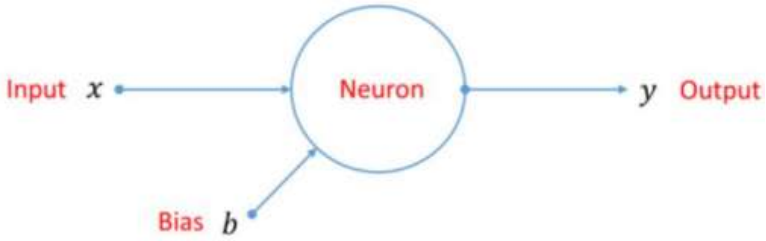


5. 인공 신경망 퍼셉트론

인공 신경 세포는 입력과 편향을 할 수 있는 뉴런과 이러한 뉴런이 연결되어 있는 신경망이 존재한다. 입력과 출력의 편향은 편향을 조정하여 출력을 맞춘다.

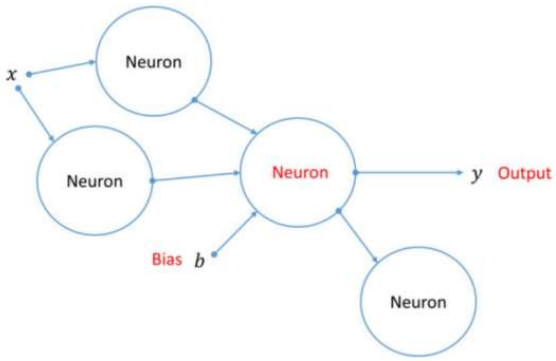
5-1. 인공 신경 세포

뉴런
- 입력
- 편향(bias)

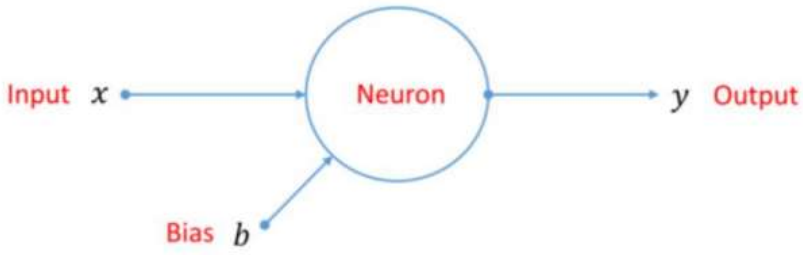


뉴런은 입력과 편향이 하나로 결과도 하나로 나온다.

신경망(network)
- 뉴런의 연결



신경망은 뉴런을 연결한 것으로 그물망처럼 생겼다는 것에서 따온 이름이다.



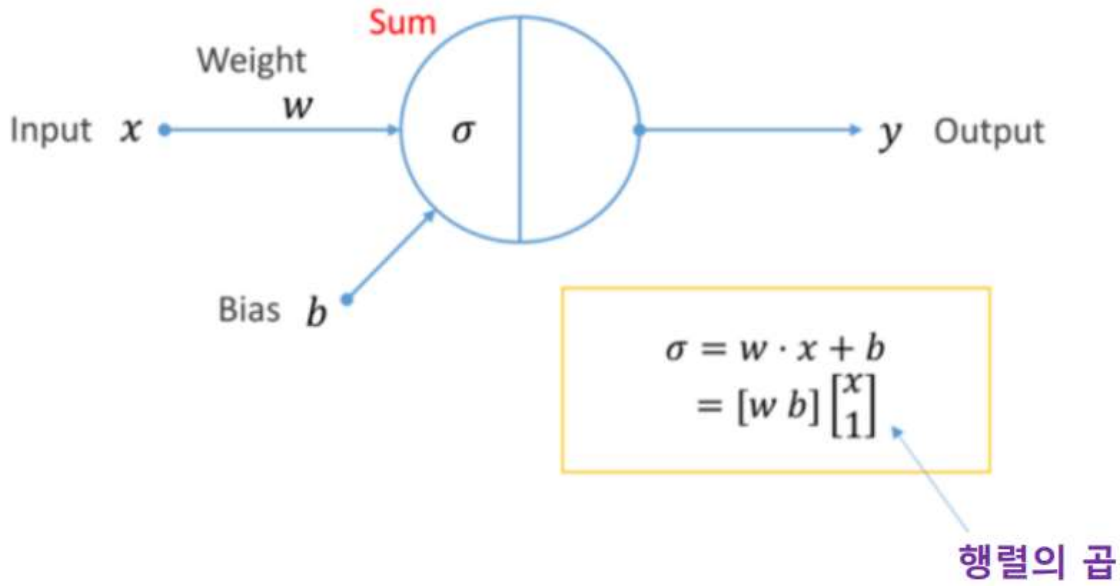
Input x	Output y
Size of house	Price
Time spent for studying	Score in exam

위 사진은 편향을 가르키는 것으로 편향을 조정하면 출력을 맞출 수 있다.

뉴런에는 여러 가지 함수도 존재한다. 뉴런 식, 가중치와 편향, 행렬 곱 연산, 활성화 함수 등이 있다. 대부분의 함수는 가중치와 입력, 편향으로 이루어져 있다.

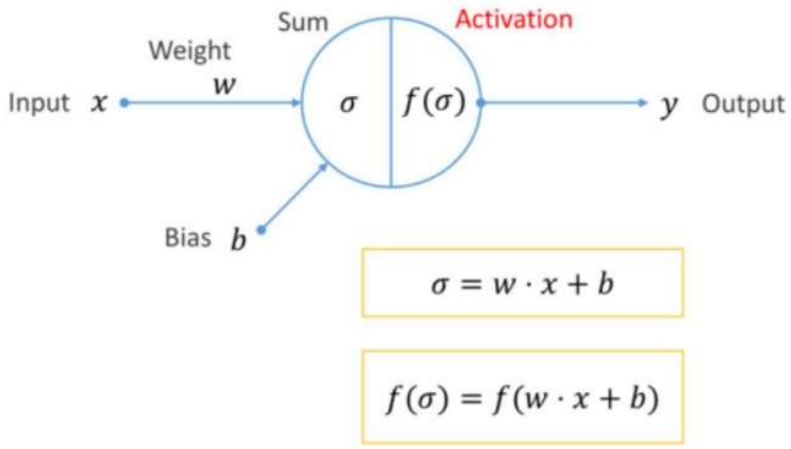
뉴런의 식 = 가중치 x 입력 x 편향
가중치와 편향의 식 = 가중치 x 입력 + 편향
행렬 곱 연산식 = 가중치 x 입력 + 편향을 가르킨다.

행렬 곱 연산은 말로 설명하기 어려우니 하단에 사진을 첨부한다.



5-2. 활성화 함수

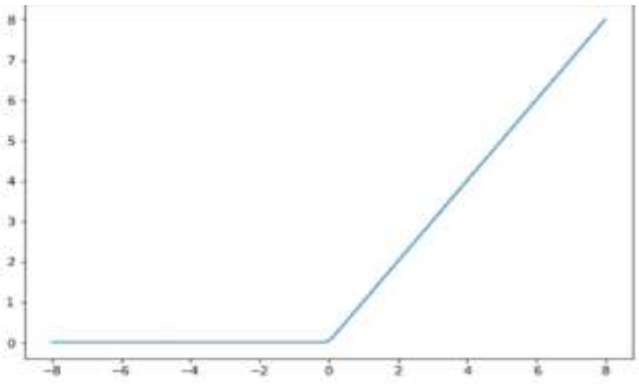
활성화 함수는 뉴런의 출력 값을 정하는 함수이다.



활성화 함수에는 크게 두가지 함수가 있다. ReLU 함수와 Sigmoid 함수이다.

ReLU: 정류된 선형 함수는 선형 함수를 정류하여 0 이하는 모두 0으로 한 함수이다. max(x, 0)으로 양수만 사용한다. 2010년 이후에 층이 깊어질수록 많이 이용하기 시작한 함수이다.

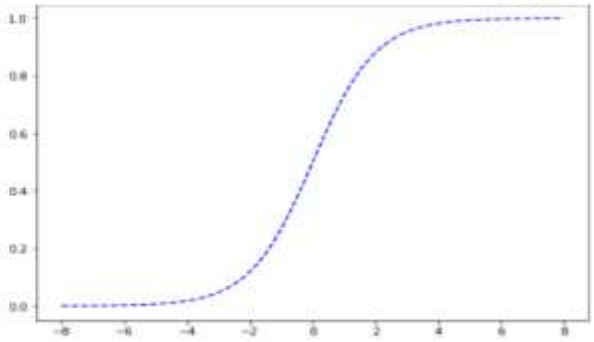
$$h(x) = \begin{cases} 0\ (x \le 0) \\ x (x > 0) \end{cases}$$



[ReLU함수] ReLU 함수 모양

Sigmoid: s자 형태의 곡선이라는 의미가 있는 함수는 예전에 많이 사용하였지만 최근에는 많이 쓰이진 않는 함수이다.

$$h(x) = \frac{1}{1 + e^{-x}}$$



[Sigmoid함수] Sigmoid 함수 모양

활성화 함수와 편향은 결과 값이 임계 값 역할을 하여 이상이면 활성화되고 미만이면 비활성화하는 편향을 지니고 있다.

이 외에도 활성화 함수는 다양하게 존재한다.

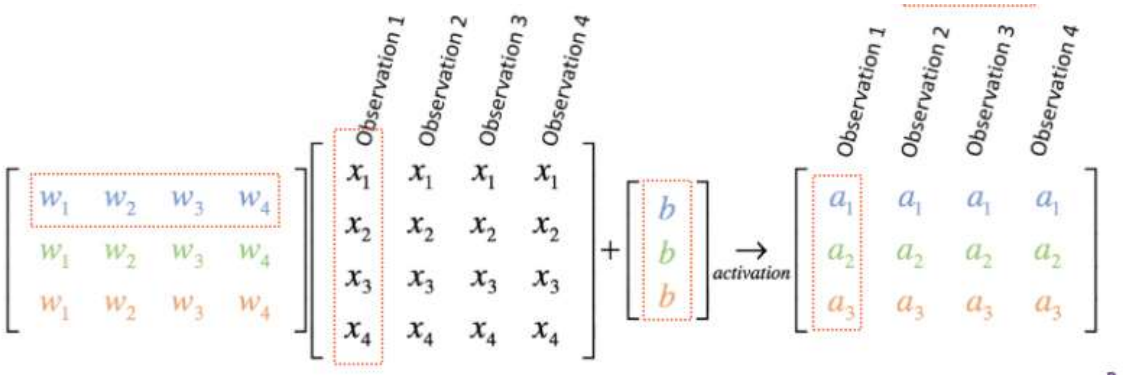
5-3. 행렬 함수

행렬은 입력을 오른쪽 행렬에 배치하여 가중치는 왼쪽 행렬에 배치하는 것이다. 곱의 순서도 변환된다. 가중치와 입력 값의 순서도 수정하여 행렬의 순서를 바꾼

계산이다.

5-3. 행렬 연산

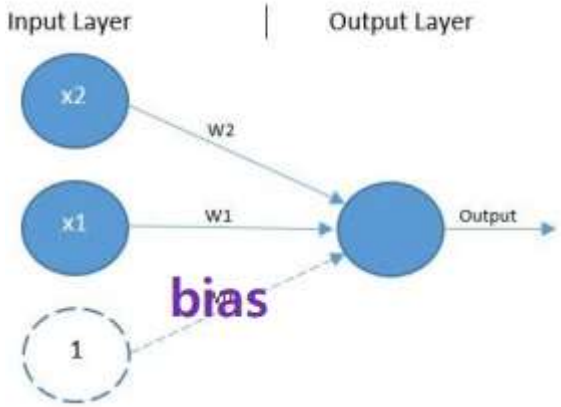
행렬은 다양한 표현을 가지고 있다. 그중 하나를 살펴보자면 입력을 오른쪽 행렬에 배치하며 가중치는 왼쪽 행렬에 배치, 곱의 순서도 변환한다.



6. 논리 게이트 AND OR XOR 신경망 구현

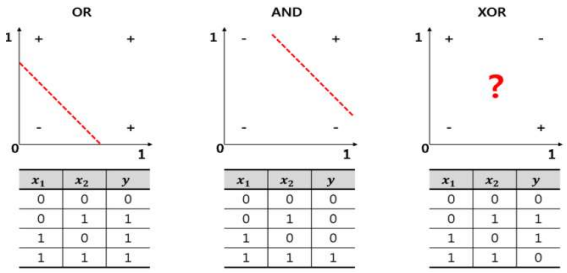
6-1. AND 게이트 구현

AND 게이트의 구조는 입력 2개, 편향, 출력 1 로 이루어져 있다. 가중치 2개와 편향 1개의 값을 구한다.



6-2. XOR 게이트 구현

XOR 게이트는 마빈 민스키와 시모어 페퍼트가 증명한 것이다. XOR의 구조는 하나의 퍼셉트론으로는 XOR 게이트는 불가능하다. 뉴런은 3개의 2층으로 가능하다. 모델이 구해야 할 총 매개변수는 가중치와 편향이 포함되어 있다.



x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

이러한 XOR 을 해결할 수 있는 것은 뉴런 3개의 2층으로 가능하다. 모델이 구해야 할 총 매개변수는 (가중치와 편향)이다.

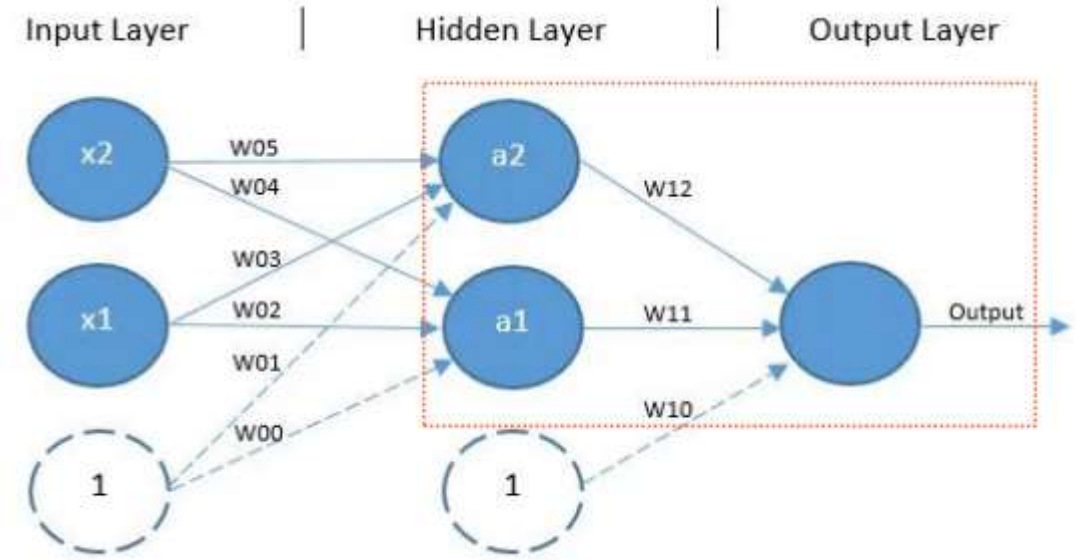


Figure 4: Multilayer Pereceptron Architecture for XOr

6-3. Sequential 모델

Dense 층을 통해 모델을 만든다. Dense 층은 가장 기본적인 층으로 인자 units, activation으로 뉴런 수와 활성화 함수를 가르키며 인자는 input_shape로 표현된다. 첫 번째 층에서만 정의되며 입력의 차원을 명시해야만 한다.

Sequential 모델과 딥러닝 구조는 입력, 은닉, 출력 층 세가지로 이루어져 있다.

7. 회귀와 분류

7-1. 회귀 모델

연속적인 값을 예측하는 회귀 모델은 예를 들어, 캘리포니아의 주택 가격이 얼마인가요? 사용자가 이 광고를 클릭할 확률이 얼마인가요? 등의 질문을 던지는 것과 동일하다.

회귀 분석은 관찰된 연속형 변수들에 대해 두 변수 사이의 모형을 구한 뒤 적합도를 측정해 내는 분석 방법이다. 시간에 따라 변화하는 데이터나 어떤 영향, 가설적 실험, 인과 관계의 모델링 등의 통계적 예측에 이용된다.

7-2. 분류 모델

불연속적인 값을 예측하는 회귀 모델은 예를 들어, 주어진 이메일 메시지가 스팸인가요, 스팸이 아닌가요?, 이 이미지가 강아지, 고양이 또는 햄스터의 이미지인가요? 등의 질문을 던지는 것과 동일하다.

7-3. 기본 개념 익히기

가설: 가중치와 편향, 기울기와 절편
손실 함수: MSE (평균제곱오차), Categorical crossentropy, Spare Categorical crossentropy
경사 하강법: 내리막 경사 따라가기
학습률: 대표적인 하이퍼패러미터

7-4. 선형 회귀

데이터의 경향성을 가장 잘 설명하는 하나의 직선을 예측하는 방법을 뜻한다. 단순 선형 회귀 분석은 입력하고 출력하는 형식이다. 다중 선형 회귀 분석은 특징이 여러 개인데 출력이 하나의 값이라는 것이다. 로지스틱 회귀는 이진 분류와 입력이 하나 또는 여러 개, 출력은 0 아니면 1을 나타낸다.

인공지능이란 단순 선형 회귀 분석 식에서 가중치와 편향을 구하는 것이다.

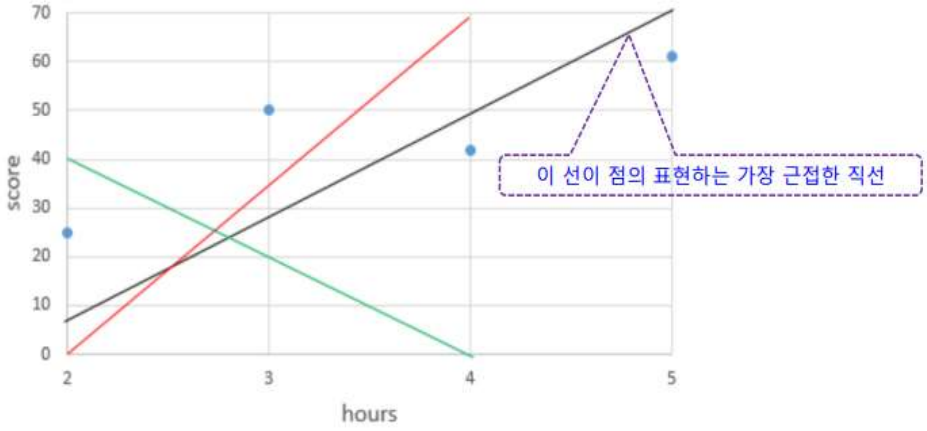
손실 함수는 실제 값과 가설로부터 얻은 예측 값의 오차를 계산하는 식이다. 손실 함수는 목적 함수, 비용 함수라고도 부르며 실제 값과 예측 값에 대한 오차에 대해 최적화되어 있다.

가설은 머신 러닝으로 y와 x간의 관계를 유추한 식을 가설이라고 부른다. 선형 회귀에서는 적절한 기울기, 가중치와 편향을 구하는 것이다.

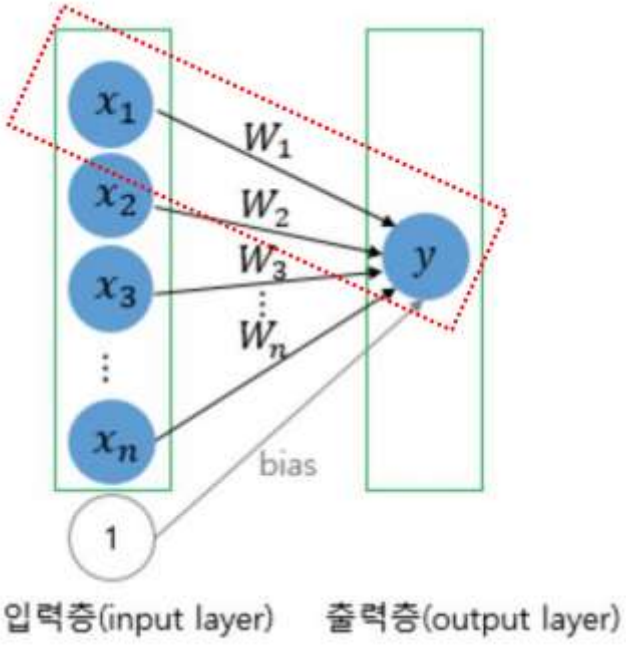
H(x)에서 H는 Hypothesis를 의미

$$H(x) = Wx + b$$

W:기울기, 가중치
b: 절편, 편향



선형 회귀를 Dense 층에서 입력이 1차원이며 출력도 1차원인 곳에서 하게 된다면 어떻게 될까? 활성화 함수 linear 도 디폴트 값, 입력 뉴런과 가중치로 계산된 결과값이 그대로 출력으로 나오게 된다.



확률적 경사하강법은 경사하강법의 계산량을 줄이기 위해 확률적 방법으로 경사하강법을 사용한다. 전체를 계산하지 않고 확률적으로 일부 샘플로 계산하는 것이다.

mae는 평균 절대 오차로 모든 예측과 정답과의 오차 합의 평균을 의미한다.

$$\mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

선형 회귀 모델이 학습(훈련)하는 것은 히스토리 객체를 훈련시키는 것이다. 매 에포크 마다의 훈련 손실 값(loss), 정확도(accuracy), 손실값(val_loss), 정확도(val_acc)는 훈련과정 정보를 history 객체에 저장한다.

```
Epoch 374/500
1/1 [==============================] - 0s 1ms/step - loss: 4.2576e-04 - mae: 0.0172 - mse: 4.2576e-04
Epoch 375/500
1/1 [==============================] - 0s 1ms/step - loss: 4.2321e-04 - mae: 0.0171 - mse: 4.2321e-04
Epoch 376/500
1/1 [==============================] - 0s 2ms/step - loss: 4.2068e-04 - mae: 0.0171 - mse: 4.2068e-04
Epoch 377/500
1/1 [==============================] - 0s 1ms/step - loss: 4.1817e-04 - mae: 0.0170 - mse: 4.1817e-04
Epoch 378/500
1/1 [==============================] - 0s 1ms/step - loss: 4.1566e-04 - mae: 0.0170 - mse: 4.1566e-04
Epoch 379/500
1/1 [==============================] - 0s 1ms/step - loss: 4.1318e-04 - mae: 0.0169 - mse: 4.1318e-04
```

```
# ⑤ 테스트 데이터로 성능 평가
x_test = [1.2, 2.3, 3.4, 4.5]
y_test = [2.4, 4.6, 6.8, 9.0]

print('손실', model.evaluate(x_test, y_test))
```

```
1/1 [==============================] - 0s 1ms/step - loss: 0.0012 - mae: 0.0313 - mse: 0.0012
손실: [0.0012317538494244218, 0.031307220458984375, 0.0012317538494244218]
```

```
# x = [3.5, 5, 5.5, 6]의 예측
print(model.predict([3.5, 5, 5.5, 6]))

pred = model.predict([3.5, 5, 5.5, 6])
# 예측 값만 1차원으로
print(pred.flatten())
print(pred.squeeze())
```

```
[[ 6.9934297]
 [ 9.975829 ]
 [10.969961 ]
 [11.964094 ]]
[ 6.9934297  9.975829  10.969961  11.964094 ]
[ 6.9934297  9.975829  10.969961  11.964094 ]
```

```
# ⑤ 테스트 데이터로 성능 평가
x_test = [1.2, 2.3, 3.4, 4.5]
y_test = [2.4, 4.6, 6.8, 9.0]

print('손실', model.evaluate(x_test, y_test))
```

```
1/1 [==============================] - 0s 1ms/step - loss: 0.0012 - mae: 0.0313 - mse: 0.0012
손실: [0.0012317538494244218, 0.031307220458984375, 0.0012317538494244218]
```

```
# x = [3.5, 5, 5.5, 6]의 예측
print(model.predict([3.5, 5, 5.5, 6]))

pred = model.predict([3.5, 5, 5.5, 6])
# 예측 값만 1차원으로
print(pred.flatten())
print(pred.squeeze())
```

```
[[ 6.9934297]
 [ 9.975829 ]
 [10.969961 ]
 [11.964094 ]]
[ 6.9934297  9.975829  10.969961  11.964094 ]
[ 6.9934297  9.975829  10.969961  11.964094 ]
```

케라스로 예측도 할 수 있다. 케라스와 numpy 사용한다. 학습에 3개 데이터가 필요하며 예측으로 뒤 2개의 데이터를 알아낼 수도 있다.

케라스로 예측하는 순서는 이렇다.

케라스 패키지 임포트	Import tensorflow as tf Import numpy as np
데이터 지정	X = numpy.array([0,1,2,3,4]) Y = numpy.array([1,3,5,7,9])
인공신경망 모델 구성	Model = tf.keras.models.Sequential() Model.add(tf.keras.layers.Dense(출력수, input_shape=(입력수))
최적화 방법 & 손실 함수 인공신경망 모델 생성	Model.compile('SGD', 'mse')
생성된 모델로 훈련 데이터 학습	Model.fit(…)
성능 평가	Model.evaluate(…)
테스트 데이터 결과 예측	Model.predict(…)

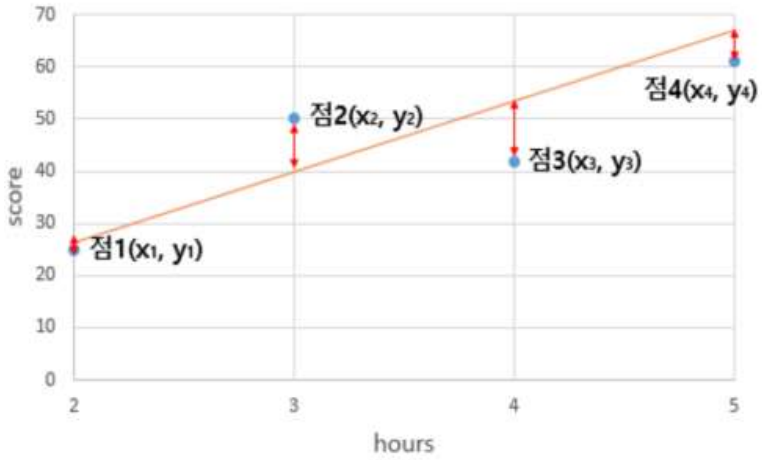
7-4. 손실 함수

목적 함수, 비용 함수라고도 부르는 손실함수는 머신 러닝은 가중치와 편향을 찾기 위해서 한다. 손실 함수란 실제 값과 가설로부터 얻은 예측 값의 오차를 계산하는 식이다. 손실 함수 값을 최소화하는 최적의 가중치와 편향을 찾아내려고 노력하는 것이 손실함수이다. 손실함수로 보통 평균 제곱 오차(MSE) 등을 사용한다.

평균 제곱 오차(MSE)는 가중치와 편향 값을 찾아내기 위해 오차의 크기를 측정할 방법으로 사용한다.

$$\frac{1}{n}\sum_{i}^{n}[y_i - H(x_i)]^2$$

실제 값 / 예측 값

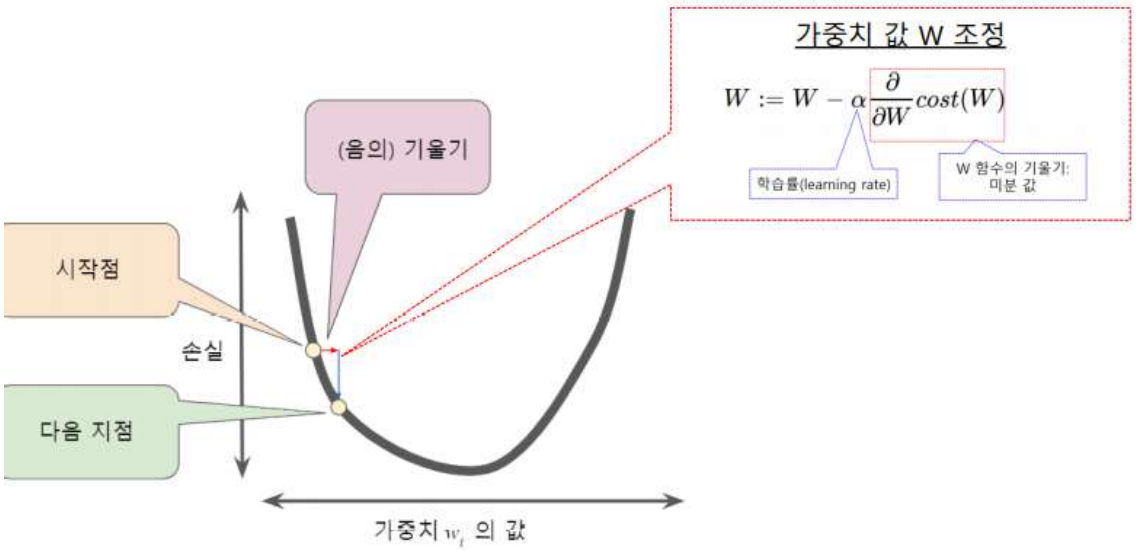


7-5. 최적화 과정

옵티마이저는 머신 러닝에서 학습하는 최적화 알고리즘이다. 적절한 가중치와 편향을 찾아내는 과정이기도 한다. 경사 하강법으로 경사를 따라 내려오면서 값을 나타내기도 한다.

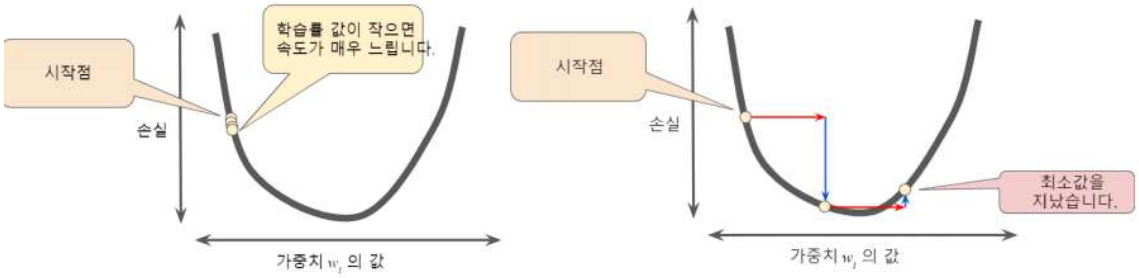
손실과 가중치는 항상 볼록 함수 모양을 지닌다. 볼록 문제에는 기울기가 정확하게 0인 지점인 최소값이 하나만 존재한다. 이 최소값에서 손실 함수가 수렴되어 결국엔 기울기를 구해야 편향 값이 나온다.

경사 하강법 첫 번째는 시작 값을 선택하는 것이다. 시작점은 별로 중요하지 않아 많은 알고리즘에서 0으로 설정하거나 임의의 값을 선택한다. 시작점에서 손실 곡선의 기울기를 계산하며 단일 가중치에 대한 손실의 기울기는 미분 값과 동일하다. 가중치의 조정을 하며 기울기가 0인 지점을 찾기 위해 기울기의 반대 방향으로 이동한다. 현재 기울기가 음수라면 다음 가중치 값은 현재의 값보다 크게 조정한다.



학습률은 다음 가중치 값에 따라 결정된다. 기울기에 학습률을 곱하여 다음 지점

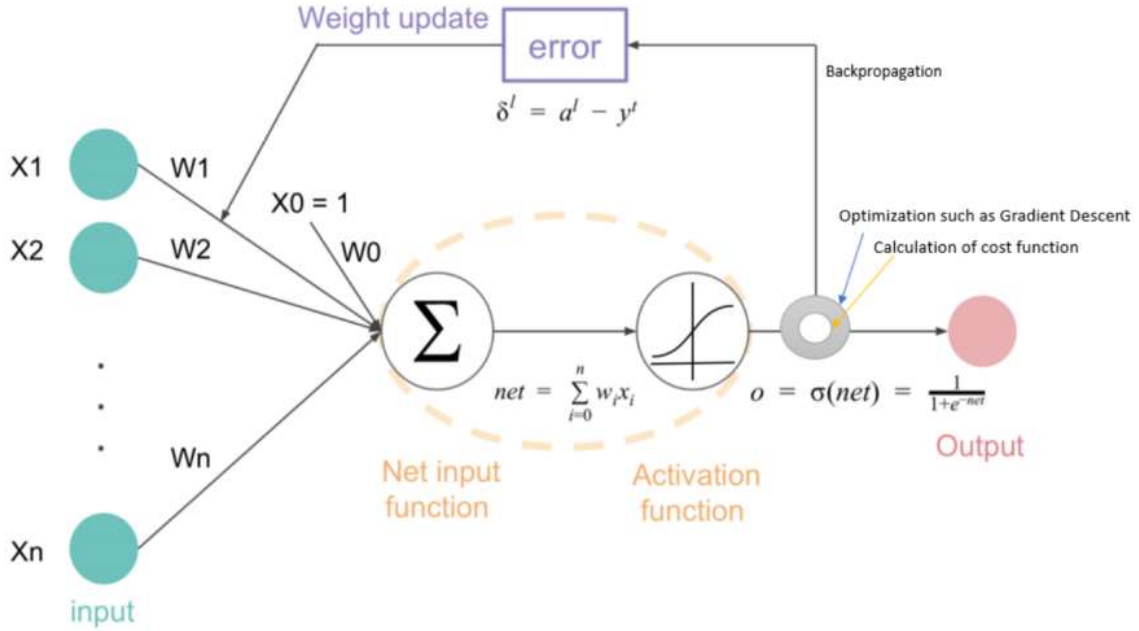
을 결정한다. 학습률의 값을 너무 작게 설정하면 학습 시간이 매우 오래 걸리며 크게 설정하면 다음 지점이 곡선의 최저점을 무질서하게 이탈할 위험이 있다.



고로 적절한 학습률을 설정해야 한다.

초매개변수는 매개변수와 대비되는 개념이다. 딥러닝에서 우리가 설정하는 값은 모델 학습을 연속적으로 실행하는 중에 개발자 본인에 의해 조작되는 손잡이이다. 예를 들어 학습률은 초매개변수 중 하나이다.

다양한 학습률 실험이 있는데 가중치 값 변경, 폭 이동 결정 등이 있다.

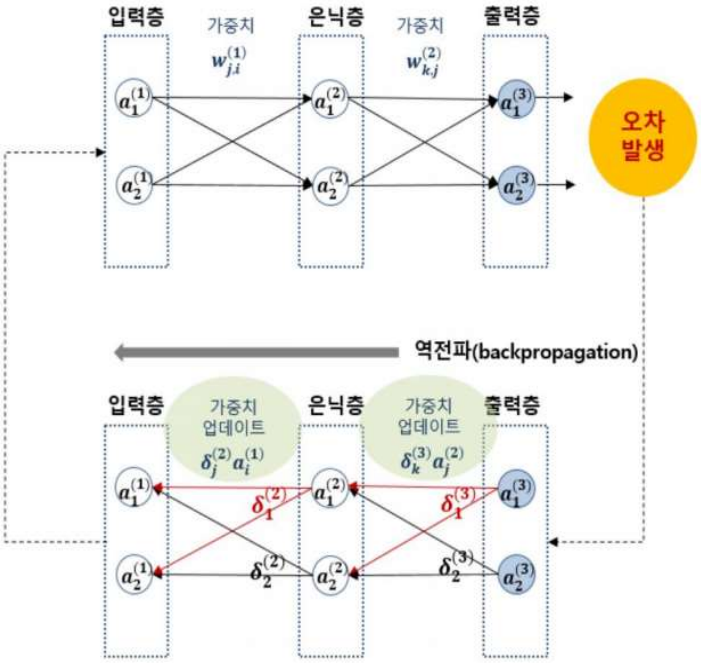


7-6. 오차역전파

순전파: 입력층에서 출력층으로 계산해 최종 오차를 계산하는 방법

역전파: 1986년 제프리 힌튼이 적용하여 엄청난 처리 속도의 증가를 불러 일으켰다. 오차 결과 값을 통해서 다시 역으로 input 방향으로 오차가 적어지도록 다시 보내

며 가중치를 다시 수정하는 방법

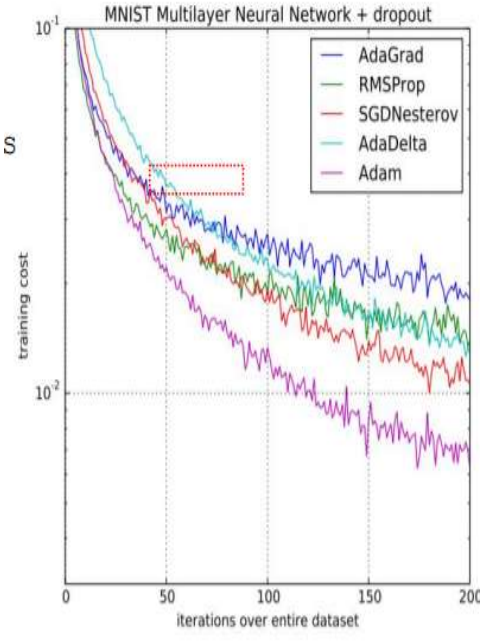


8. 케라스 모델 미사용 텐서플로 프로그래밍

8-1. 회귀 분석

Optimizer : 최적화 과정(복잡한 미분 계산 및 가중치 수정)을 자동으로 진행한다. 주로 SGD, adam 이 있다.

학습률은 보통 0.1~0.0001을 지닌다.



8-2. 변수 Variables

딥러닝 학습에서 최적화 과정은 모델의 매개변수 즉 가중치 및 편향을 조정하는 것이다. 변수 tf.Variable 는 프로그램에 의해 변화하는 공유된 지속 상태를 표현하는 가장 좋은 방법으로 텐서를 표현하거나 텐서에 연산을 수행하여 변경이 가능하다. 모델 피라미터를 저장하는데 tf.Variable을 사용한다. 변수 생성을 하기 위해서는 단순하게 초기값을 설정한다.

```
# a와 b를 랜덤한 값으로 초기화합니다.
# a = tf.Variable(random.random())
# b = tf.Variable(random.random())
a = tf.Variable(tf.random.uniform([1], 0, 1))
b = tf.Variable(tf.random.uniform([1], 0, 1))
```

8-2. 변수 Variables

최소화할 손실 함수를 정하고 var_list로 학습시킬 변수 리스트, 가중치와 편향을 정한다. 1000번의 학습을 거쳐 잔차의 제곱 평균을 최소화 하는 적절한 값 a, b에 도달한다.

```
for i in range(1000):
    # 잔차의 제곱의 평균을 최소화(minimize)합니다.
    optimizer.minimize(compute_loss, var_list=[a,b])
```

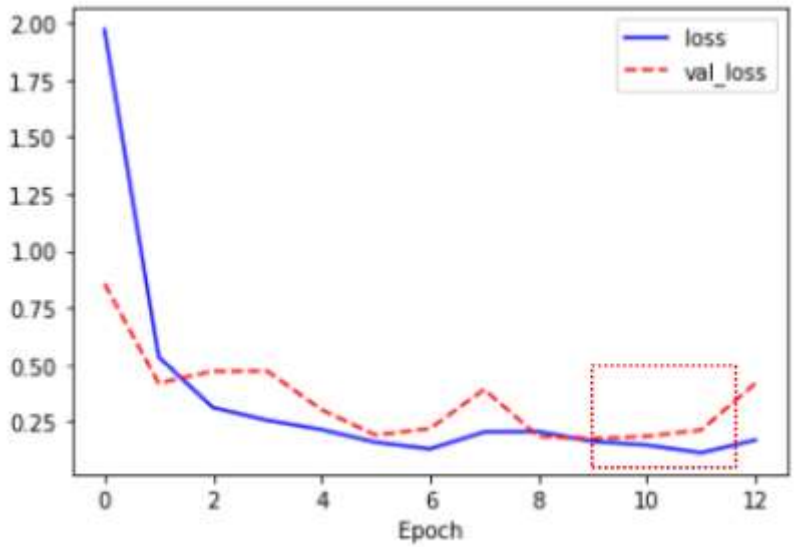
8-3. 자동 학습 중단

검증 손실(val_loss)가 적을수록 테스트 평가의 손실도 적다. 검증 데이터에 대한 성적이 좋도록 유도해야한다. 일찍 멈춤이 가능하여 좋다.

```
history = model.fit(train_X, train_Y, epochs=25, batch_size=32, validation_split=0.25,
          callbacks=[tf.keras.callbacks.EarlyStopping(patience=3, monitor='val_loss')])
```

이 외에도 자동 중단 시각화가 존재한다.

```
[73]  1 # 4.19 회귀 모델 학습 결과 시각화
      2 import matplotlib.pyplot as plt
      3 plt.plot(history.history['loss'], 'b-', label='loss')
      4 plt.plot(history.history['val_loss'], 'r--', label='val_loss')
      5 plt.xlabel('Epoch')
      6 plt.legend()
      7 plt.show()
```



8-4. 간단한 명령어 모음

Describe(): 전반적인 통계를 확인할 수 있는 명령어이다.

```
# 전반적인 통계도 확인
train_stats = train_dataset.describe()
print(train_stats)
```

	MPG	Cylinders	Displacement	Horsepower	Weight	W
count	314.000000	314.000000	314.000000	314.000000	314.000000	
mean	23.310510	5.477707	195.318471	104.869427	2990.251592	
std	7.728652	1.699788	104.331589	38.096214	843.898596	
min	10.000000	3.000000	68.000000	46.000000	1649.000000	
25%	17.000000	4.000000	105.500000	76.250000	2256.500000	
50%	22.000000	4.000000	151.000000	94.500000	2822.500000	
75%	28.950000	8.000000	265.750000	128.000000	3608.000000	
max	46.600000	8.000000	455.000000	225.000000	5140.000000	

	Acceleration	Model Year	USA	Europe	Japan
count	314.000000	314.000000	314.000000	314.000000	314.000000
mean	15.559236	75.898089	0.624204	0.178344	0.197452
std	2.789230	3.675642	0.485101	0.383413	0.398712
min	8.000000	70.000000	0.000000	0.000000	0.000000
25%	13.800000	73.000000	0.000000	0.000000	0.000000
50%	15.500000	76.000000	1.000000	0.000000	0.000000
75%	17.200000	79.000000	1.000000	0.000000	0.000000
max	24.800000	82.000000	1.000000	1.000000	1.000000

MPG: 레이블을 만들고 원 데이터집합에서 제거한다.

```
train_labels = train_dataset.pop('MPG')
test_labels = test_dataset.pop('MPG')
```

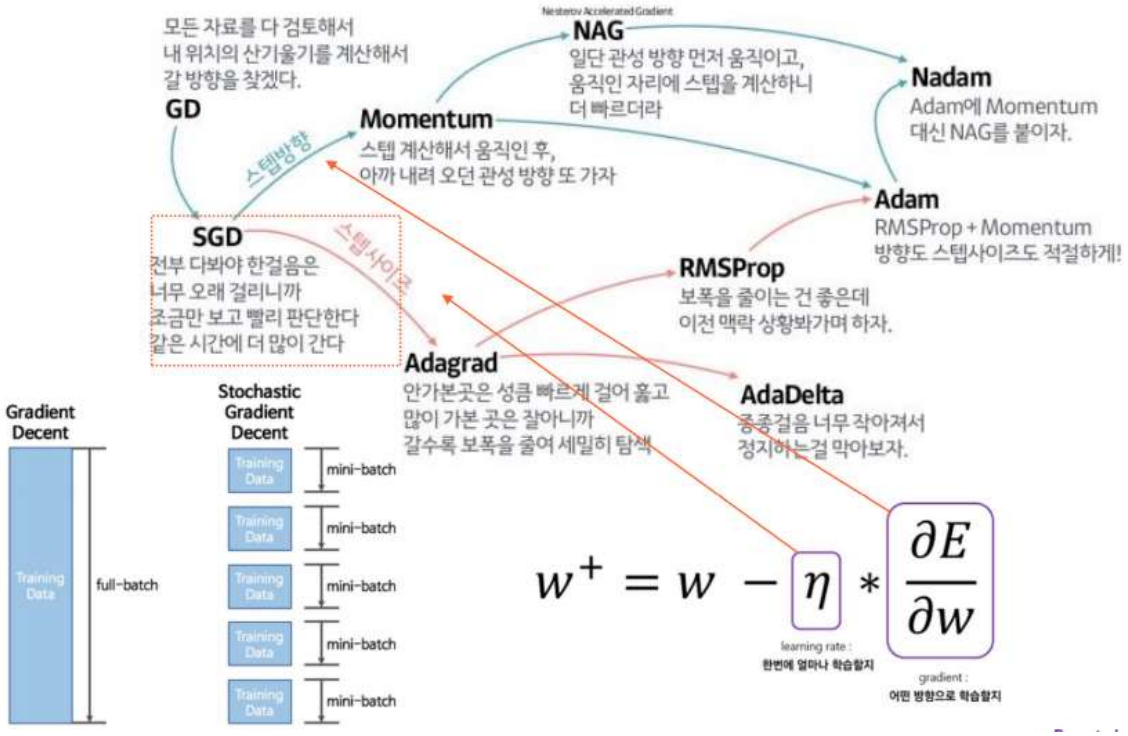
데이터 정규화: 특성의 스케일과 범위가 다르면 정규화 하는 것을 권장한다 의도적으로 훈련 세트만 사용하여 통계치를 생성하기도 한다. 테스트 세트를 정규화할 때에도 훈련 데이터의 평균과 표준변차를 사용한다. 테스트 세트를 모델이 훈련에 사용했던 것과 동일한 분포로 투영하기 위함이다.

콜백: 학습 과정의 한 에폭마다 적용할 함수의 세트이다. 학습의 각 단계에서 콜백의 적절한 메소드가 호출된다. 모델의 내적 상태와 통계자료를 확인한다. 키워드 인수로는 callbacks 가 있으며 Sequential이나 Model 클래스의 .fit() 매서드에 전달이 가능하다.

```
# 에포크가 끝날 때마다 점(.)을 출력, 100번마다 다음 줄로 이동해 훈련 진행 과정을 표시합니다
class PrintDot(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        if epoch % 100 == 0: print('')
        print('.', end='')

EPOCHS = 1000

history = model.fit(
    normed_train_data, train_labels,
    epochs=EPOCHS, validation_split = 0.2, verbose=0,
    callbacks=[PrintDot()])
```

훈련 과정을 시각화 하여 검증 손실이 계속 감소하는 것이 중요하다.

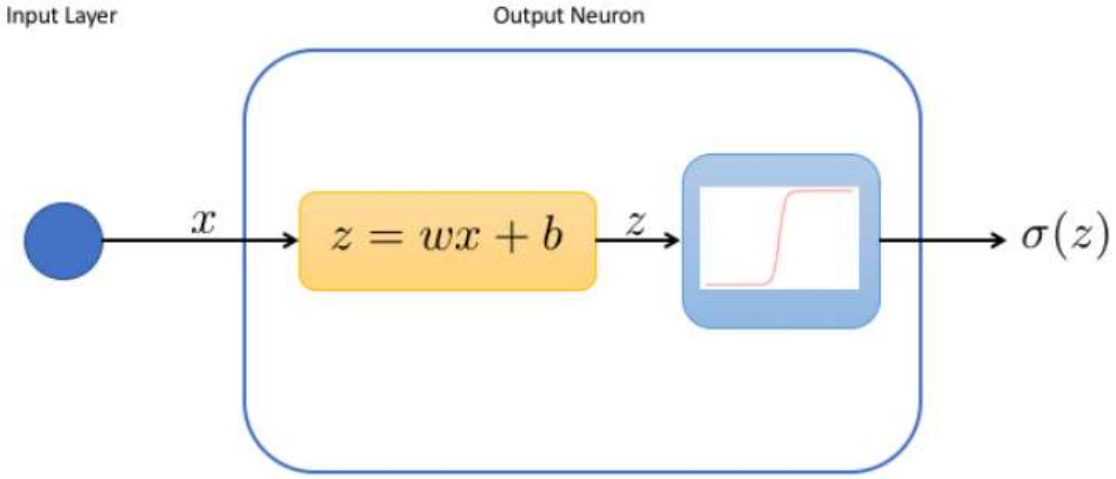
8-5. 옵티마이저 발전 과정



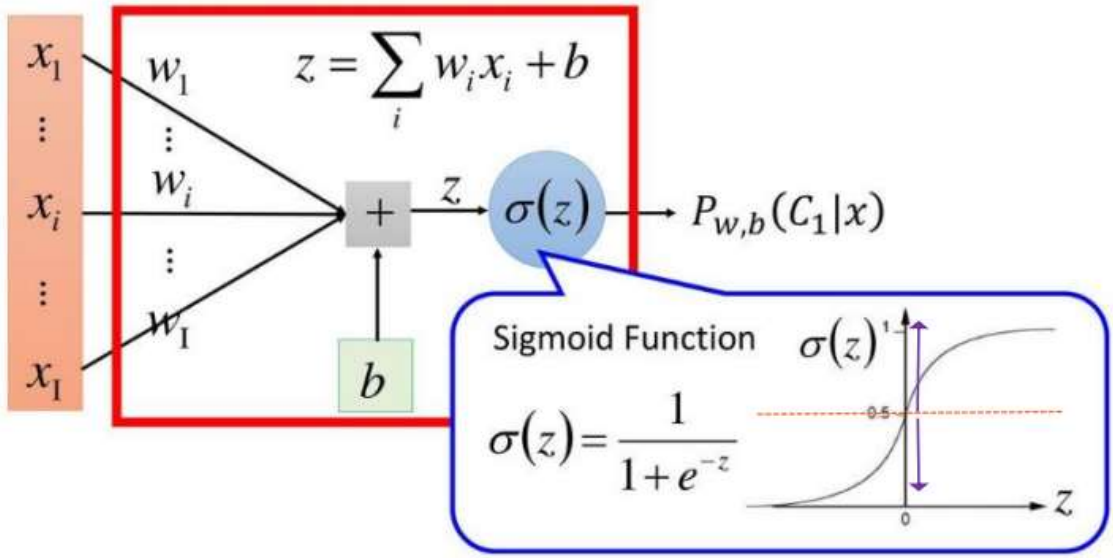
9. 분류

9-1. 이진(이항) 분류

두 가지로 분류하는 여러 가지 방법이 존재한다. 이것을 회귀로 표현하면 로지스틱 회귀라고 부른다. 결과 기술 방식이다. 4개의 결과가 나올 수 있으며 일반 레이블 방식과 OnehotEncoding 방식이 존재한다.



이진 분류 개념



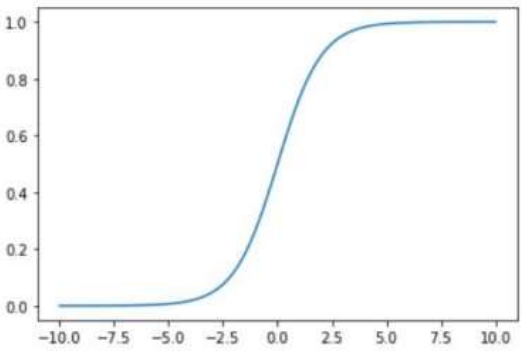
이진 분류 함수

9-1. 시그모이드 함수

이진분류 모델의 출력층에 주로 사용되는 활성화 함수이다. 0과 1 사이의 값으로 출력되며 출력 값이 특정 임계값 이상이면 양성, 이하이면 음성이라고 판별한다.

$$f(x) = \frac{1}{1 + e^{-x}}$$

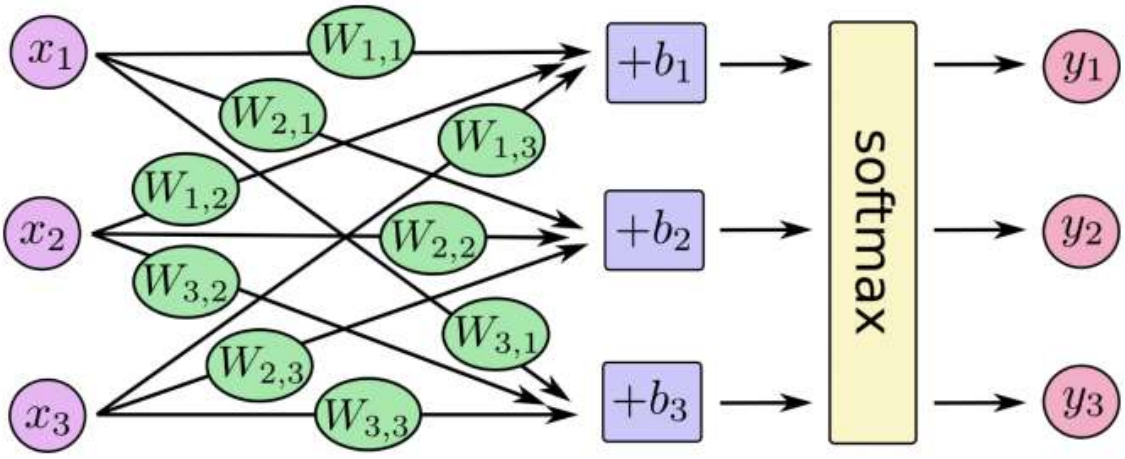
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 100)
y = 1 / ( 1 + np.exp(-x) )
plt.plot(x, y)
plt.show()
```



이진분류와 다중분류가 존재한다.
이진분류: 시그모이드 함수
다중분류: 소프트맥스 함수

9-1. 소프트맥스 함수

모든 y의 값은 1이다.



뉴런의 결과를 e의 지수승으로 하여 모든 합으로 나눈 결과는 이렇다.

exp(x)/tf.reduce_sum(exp(x))

대표적인 다중 분류로는 MNIST 손글씨가 존재한다.