인공지능 프로그래밍 포트폴리오

20160760 박윤석

텐서플로를 이용한 함수

```
[ ]  import tensorflow as tf
     tf.__version__
```

```
'2.3.0'
```

```
[ ]  a = [2, 3, 4] #함수지정
     a             #값을 출력
```

```
[2, 3, 4]
```

```
[ ]  s = 'hello, tenserflow'   #s에 hello, tenserflow를 저장
     s                         #s를 출력
```

```
'hello, tenserflow'
```

```
[3]  a = tf.constant(5)
     b = tf.constant(3)
     print(a+b)
```

```
tf.Tensor(8, shape=(), dtype=int32)
```

브로드 캐스팅을 이용한 행렬 덧셈

```
[7]  a = tf.constant([[1, 2],
                      [3, 4]])
     b = tf.add(a, 1)
     print(b)
     #모든 행과 열에 1씩 + 해줌

     tf.Tensor(
     [[2 3]
      [4 5]], shape=(2, 2), dtype=int32)
```

```
[8]  x = tf.constant([[0], [10], [20], [30]])
     y = tf.constant([0, 1, 2])

     print((x+y).numpy())
     # 0+0 0+1 0+2
     # 10+0 10+1 10+2
     # 20+0 20+1 20+2
     # 30+0 30+1 30+2

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

텐서플로를 통한 행렬 생성

```
[9]  import numpy as np

     print(np.arange(3))     #0부터 2까지 생성
     print(np.ones((3, 3)))  #1로만 3행 3열 생성
     print()

     x = tf.constant((np.arange(3)))
     y = tf.constant([5], dtype=tf. int64)
     print(x)
     print(y)
     print(x+y)

     [0 1 2]
     [[1. 1. 1.]
      [1. 1. 1.]
      [1. 1. 1.]]

     tf.Tensor([0 1 2], shape=(3,), dtype=int64)
     tf.Tensor([5], shape=(1,), dtype=int64)
     tf.Tensor([5 6 7], shape=(3,), dtype=int64)
```

arange() : 0부터 괄호안에 적힌 숫자 만큼 행렬을 생성

once() : 괄호안에 행과 열을 적고 그 적힌 숫자만큼 1로 채움

x : 0부터 2까지 행열생성
y : 5라는 행열생성
x + y : [0, 1, 2]의 행열과 [5]를 더함

행열의 덧셈

```
[10]  x = tf.constant((np.arange(3)))
      y = tf.constant([5], dtype=tf.int64)
      print((x+y).numpy())

      x = tf.constant((np.ones((3, 3))))
      y = tf.constant(np.arange(3), dtype=tf.double)
      print((x+y).numpy())

      x = tf.constant((np.arange(3).reshape(3, 1)))
      y = tf.constant(np.arange(3))
      print((x+y).numpy())
```

```
[5 6 7]
[[1. 2. 3.]
 [1. 2. 3.]
 [1. 2. 3.]]
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```

x를 [0, 1, 2]라는 행열을 생성하고 y의 [5]행열과 더함

x를 1로만 만들어진 행열을 만들고 y의 [0, 1, 2]의 행열을 더함

x를 0부터 2까지 열로 만들고 y의 [0, 1, 2]를 더함

텐서플로의 연산

```
[12]  a = 2
      b = 3
      c = tf.add(a, b)
      print(c.numpy())
```

```
5
```

a : 2로 설정
b : 3으로 설정
c : a와 b를 더함

```
[13]  x = 2
      y = 3
      add_op = tf.add(x, y)   #5
      mul_op = tf.multiply(x, y)  #6
      pow_op = tf.pow(add_op, mul_op) #5의 6승(제곱)

      print(pow_op.numpy())
```

```
15625
```

x : 2로 설정
y : 3으로 설정
add_op : x와 y를 더함
add_op : x와 y를 곱함
pow_op : add_op의 add_op제곱

행열과 원소의 곱

```
[35]  a = tf.constant([[1, 2],
                       [3, 4]])
      b = tf.add(a, 1)
```

```
[36]  #연산자 오버로딩 지원
      print(a)
      #텐서로부터 numpy값 얻기:
      print(a.numpy())
      print(b)
      print("asdf", b.numpy())
      print(a * b)
```

```
tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)
[[1 2]
 [3 4]]
tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)
asdf [[2 3]
 [4 5]]
tf.Tensor(
[[ 2  6]
 [12 20]], shape=(2, 2), dtype=int32)
```

a : [1, 2]
[3, 4] 로 설정
b : a의 행열에 각각 1씩 더함

a.numpy() : a의 행열을 출력
b.numpy() : b의 행열을 출력

print(a + b) : a행열과 b 행열을 더함

텐서플로 rank

```
[19]  my_image = tf.zeros([2, 5, 5, 3])
      my_image.shape

      TensorShape([2, 5, 5, 3])

[20]  tf.rank(my_image)

      <tf.Tensor: shape=(), dtype=int32, numpy=4>
```

2, 5, 5, 3이라는 값을 저장 하고 출력

shape와 reshape

```
[21] rank_three_tensor = tf.ones([3, 4, 5])
     rank_three_tensor.shape
```

```
TensorShape([3, 4, 5])
```

```
[22] rank_three_tensor.numpy()
```

```
array([[[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]]], dtype=float32)
```

```
[23] #기존 내용을 6x10 행렬로 형태 변경
     matrix = tf.reshape(rank_three_tensor, [6, 10])
     matrix
```

```
<tf.Tensor: shape=(6, 10), dtype=float32, numpy=
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]], dtype=float32)>
```

```
[24] #기존 내용을 3x20 행렬로 형태 변경
     #-1은 차원 크기를 계산하여 자동으로 결정하라는 의미
     matrixB = tf.reshape(matrix, [3, -1])
     matrixB
```

```
<tf.Tensor: shape=(3, 20), dtype=float32, numpy=
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1.]], dtype=float32)>
```

```
[25] #기존 내용을 4x3x5 텐서로 형태 변경
     matrixAlt = tf.reshape(matrixB, [4, 3, -1])
     matrixAlt
```

```
<tf.Tensor: shape=(4, 3, 5), dtype=float32, numpy=
array([[[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]]], dtype=float32)>
```

다차원 배열을 1차원으로 만드는 flatten

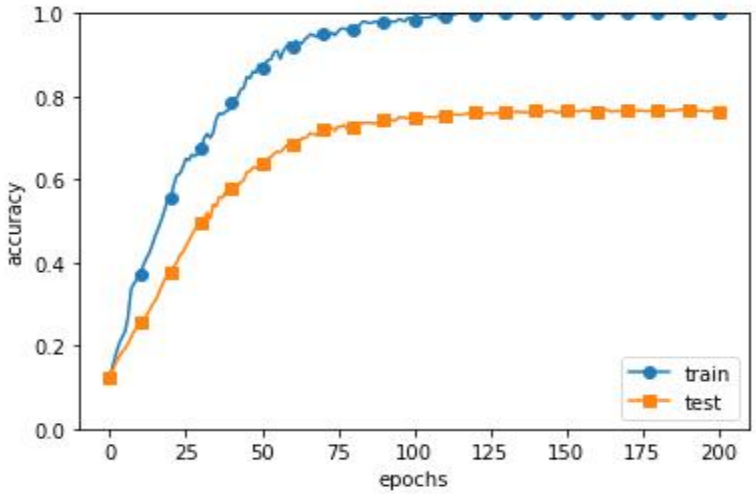
```
[1]  import numpy as np

     x = np.array([2, 3, 254, 5, 6, 3])
     x = x / 255.0
     print(x)

     x = x.reshape(2, 3)
     print(x)

     x = x.flatten()
     print(x)
```

```
[0.00784314 0.01176471 0.99607843 0.01960784 0.02352941 0.01176471]
[[0.00784314 0.01176471 0.99607843]
 [0.01960784 0.02352941 0.01176471]]
[0.00784314 0.01176471 0.99607843 0.01960784 0.02352941 0.01176471]
```

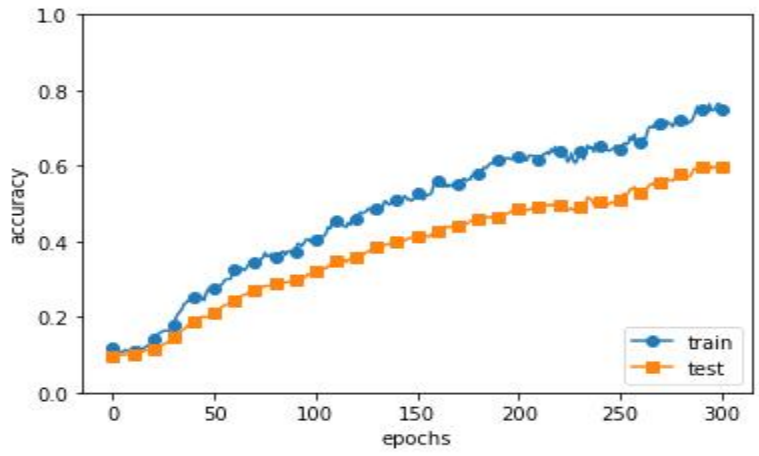
드롭아웃

- 신경망 모델이 복잡해질 때 가중치 감소만으로는 어려운데 드롭아웃 기법은 뉴런의 연결을 임의로 삭제하는 것이다. 훈련할 때 임의의 뉴런을 골라 삭제하여 신호를 전달하지 않게 한다. 테스트할 때는 모든 뉴런을 사용한다.

드롭아웃 적용 전



드롭아웃 적용 후



자료형 변환

정수형 텐서를 실수로 변경

```
[26]  float_tensor = tf.cast(tf.constant([1, 2, 3]), dtype=tf.float32)
      float_tensor
```

```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1., 2., 3.], dtype=float32)>
```

```
[27]  float_tensor.dtype
```

```
tf.float32
```

텐서플로 Variable

```
[28]  v = tf.Variable(0, 0)
      v
```

```
<tf.Variable 'Variable:0' shape=() dtype=int32, numpy=0>
```

```
[29]  w = v + 10
      w
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=10>
```

```
[30]  w.numpy()
```

```
10
```

```
[30]
```

```
[31]  v = tf.Variable(2, 0)
      v.assign_add(5)
      v
```

```
<tf.Variable 'Variable:0' shape=() dtype=int32, numpy=7>
```

```
[32]  v.read_value()
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=7>
```

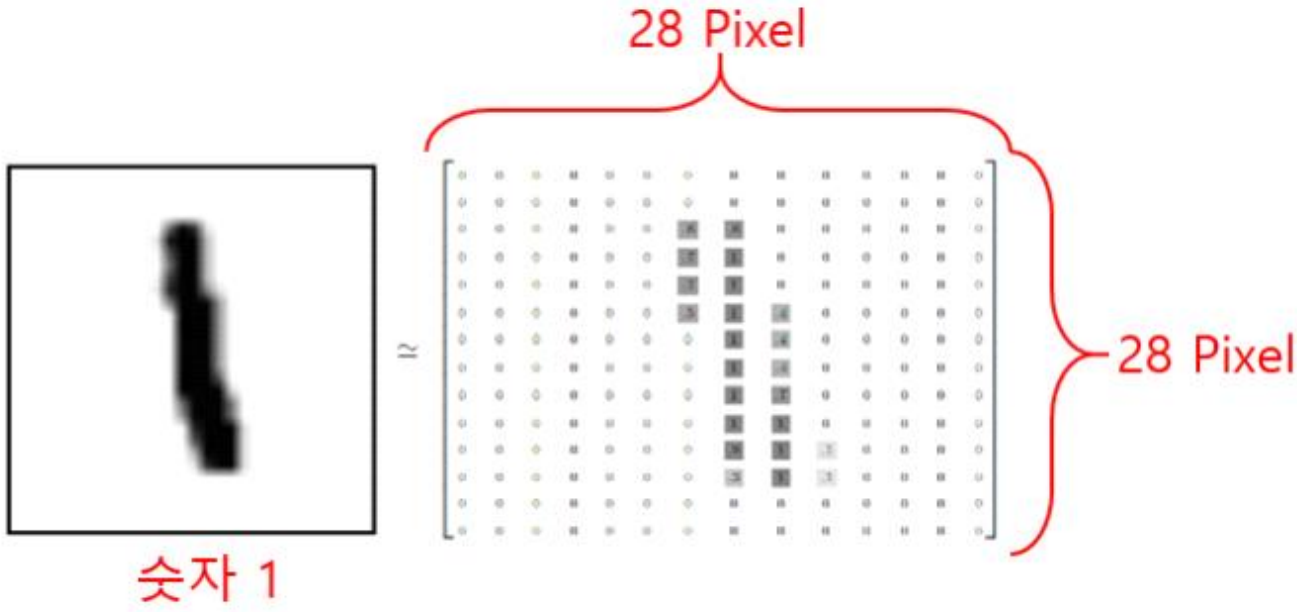
텐서플로 그래프에서
tf.Variable의값을 사용하려면 이를
단순히tf.Tensor로 취급

assign, assign_add : 값을
변수에 할당시켜줌

read_value : 현재의 변수값을 출력

mnist 데이터

- MNIST는 55,000개의 학습 데이터(mnist.train)와 10,000개의 테스트 데이터(mnist.test), 그리고 5,000개의 검증 데이터(mnist.validation)로 이루어져 있다.
- NIST의 각 이미지는 아래와 같이 28×28 픽셀로 이루어져 있다.



MNIST의 이미지

MNIST 데이터

```
In [ ]: import tensorflow as tf

        mnist = tf.keras.datasets.mnist
        (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

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

```
In [ ]: x_train.shape
```

```
Out[ ]: (60000, 28, 28)
```

```
In [ ]: y_train.shape
```

```
Out[ ]: (60000,)
```

MNIST데이터를 훈련값과, 테스트값을 불러오기

훈련값은 6만개로 되어있음

행렬의 내용을 직접 출력

```
import sys
for x in x_train[0]:
  for i in x:
    sys.stdout.write('%3d' % i)
  sys.stdout.write('₩n')
```

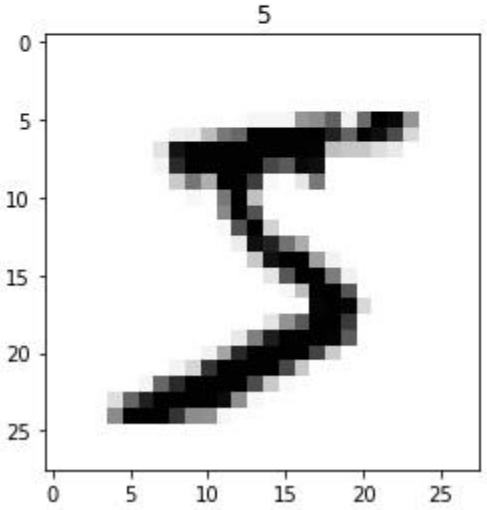
```
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  3 18 18 18126136175 26166255247127  0  0  0  0
  0  0  0  0  0  0  0  0 30 36 94154170253253253253253225172253242195 64  0  0  0  0
  0  0  0  0  0  0  0 49238253253253253253253253253251 93 82 82 56 39  0  0  0  0  0
  0  0  0  0  0  0  0 18219253253253253253198182247241  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0 80156107253253205 11  0 43154  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0 14  1154253 90  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0 0139253190  2  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0 11190253 70  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0 35241225160108  1  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0 81240253253119 25  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 45186253253150 27  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 16 93252253187  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0249253249 64  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 46130183253253207  2  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0 39148229253253253250182  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0 24114221253253253253201 78  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0 23 66213253253253253198 81  2  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0 18171219253253253253195 80  9  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0 55172226253253253253244133 11  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0 13625325325321213513216  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

28x28의 크기로 0부터
255의 정수값을 계산하여
손글씨로 출력함

28x28의 이미지 손글씨 보기

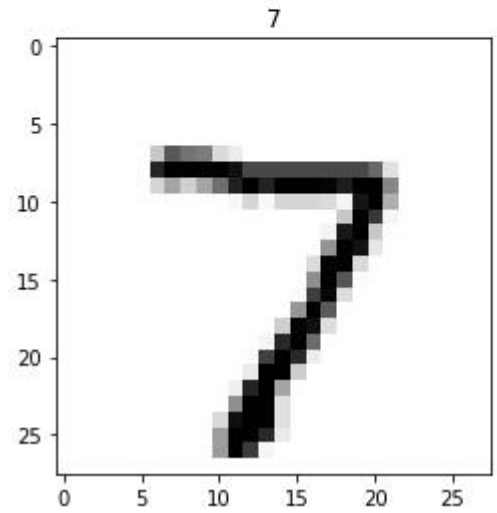
```
import matplotlib.pyplot as plt
n = 0
ttl = str(y_train[n])
plt.figure(figsize=(6, 4))
plt.title(ttl)
plt.imshow(x_train[n], cmap='Greys')
```

```
<matplotlib.image.AxesImage at 0x7f2c04e71240>
```




```
import matplotlib.pyplot as plt
n = 0
ttl = str(y_test[n])
plt.figure(figsize=(6,4))
plt.title(ttl)
plt.imshow(x_test[n], cmap='Greys')
```

```
<matplotlib.image.AxesImage at 0x7f2c04e1fd30>
```



28x28의 크기로 각각 훈련데이터와 테스트 데이터를 이용하여 손글씨를 출력

랜덤한 20개의 손글씨를 출력

```
from random import sample
norws, ncols = 4, 5
idx = sorted(sample(range(len(x_train)) , nrows * ncols))
#print(idx)

count = 0
plt.figure(figsize=(12, 10))

for n in idx:
    count += 1
    plt.subplot(nrows, ncols, count)
    tmp = "Index: " + str(n) + " Label: " +str(y_train[n])
    plt.title(tmp)
    plt.imshow(x_train[n], cmap='Greys')

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

norws : 출력할 가로의 수
ncols : 출력할 세로의 수
sorted(sample(range(len(x_train)) , nrows * ncols)) : 출력할 첨자를 선정

for문을 이용하여 같은 동작으로 하나씩 출력시킴

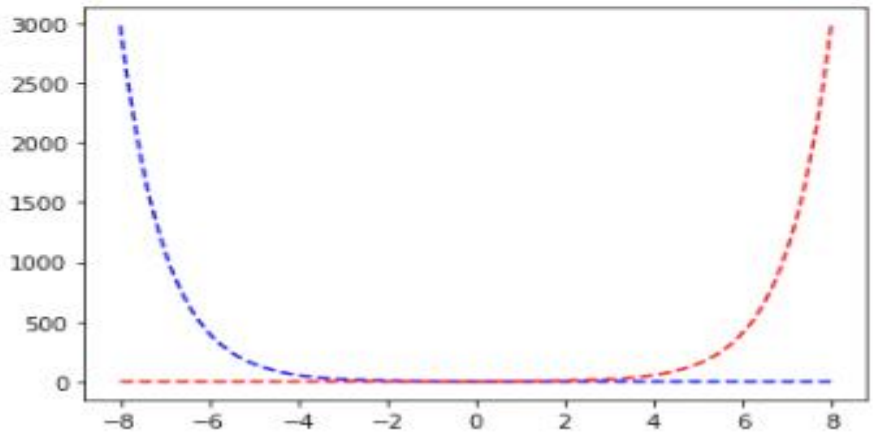
훈련과 정답 데이터 지정하기

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

한 비트의 값을 255로 나누어 0~255의 정수를 0~1로 변환

자연수와 자연수의 지수

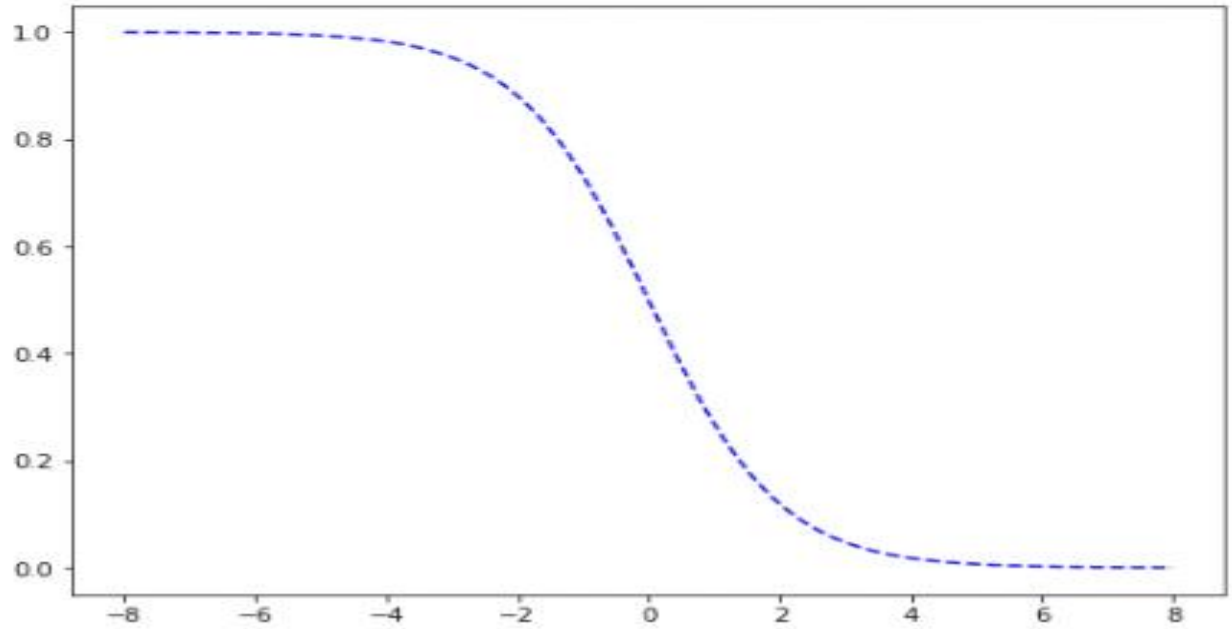
```
import numpy as np
import matplotlib.pylab as plt

plt.figure(figsize=(6, 4))
x = np.linspace(-8, 8, 100) #-8에서 8까지 100등분
plt.plot(x, np.exp(-x), 'b--')
_=plt.plot(x, np.exp(x), 'r--')
```



시그모이드 S자 곡선

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

def sigm_func(x): #sigmod 함수
    return 1 / (1 + np.exp(-x))

#시그모이드 함수 그리기
plt.figure(figsize=(8, 6))
x = np.linspace(-8, 8, 100) #-8에서 8까지 100등분
plt.plot(x, sigm_func(-x), 'b--')
```

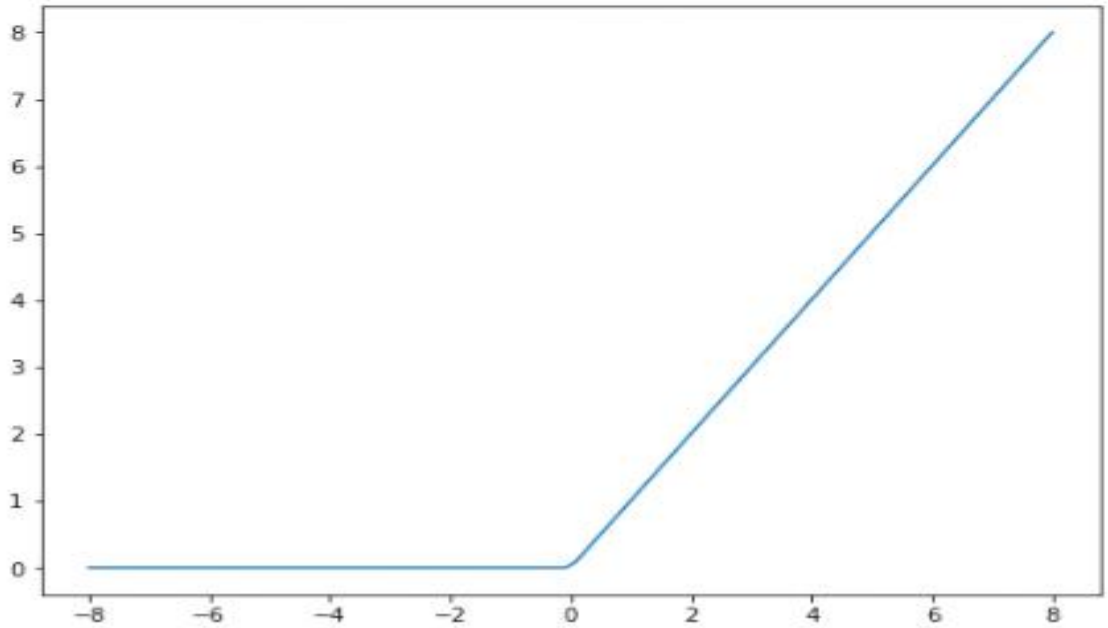
```
[<matplotlib.lines.Line2D at 0x7fdb89d8d8d0>]
```



RELU 함수를 이용한 그래프

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

def relu_func(x): #ReLU (정류화된 선형 유닛) 함수
    return np.maximum(0,x)

#ReLU함수 그리기
plt.figure(figsize=(8, 6))
x = np.linspace(-8, 8, 100)
plt.plot(x, relu_func(x))
```

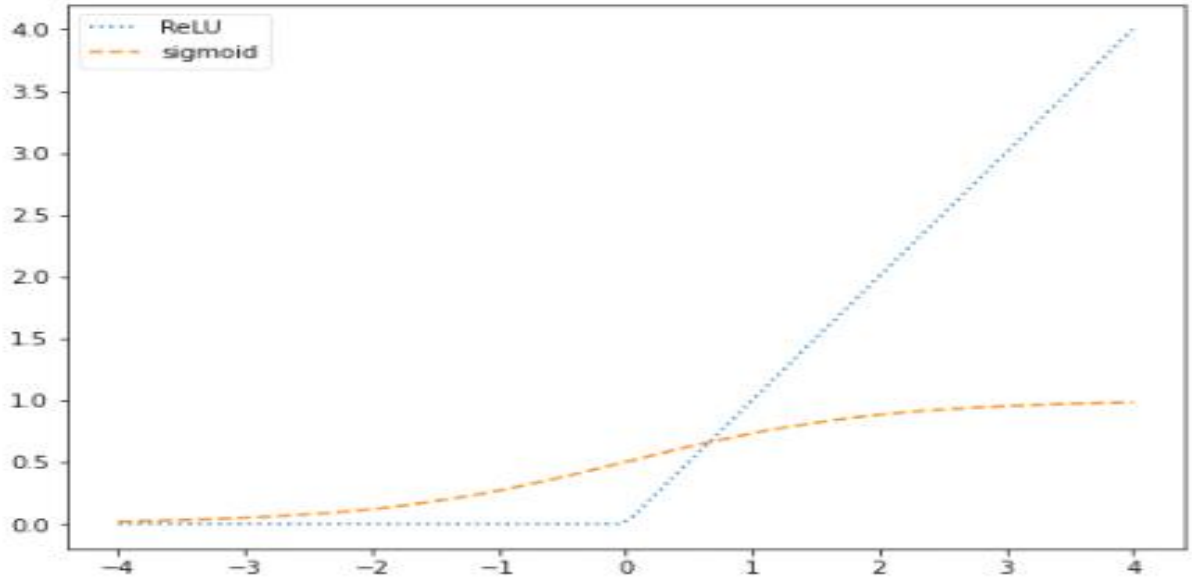
```
[<matplotlib.lines.Line2D at 0x7f462a4b3c88>]
```



시그모이드와 RELU함수

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

def relu_func(x): #ReLU (정류화된 선형 유닛) 함수
    return np.maximum(0,x)

def sigm_func(x): #sigmod 함수
    return 1 / (1 + np.exp(-x))

plt.figure(figsize=(8, 6))
x = np.linspace(-4, 4, 100)
y = np.linspace(-0.2, 2, 100)

plt.plot(x, relu_func(x), linestyle=':', label="ReLU")
plt.plot(x, sigm_func(x), linestyle='--', label="sigmoid")
plt.legend(loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7f462a23e940>
```



tf.keras 를 이용한 AND 네트워크 계산

```
[ ]  # tf.keras 를 이용한 AND 네트워크 계산
     import numpy as np
     x = np.array([[1,1],[1,0],[0,1],[0,0]])
     y = np.array([[1], [0], [0], [0]])

     model = tf.keras.Sequential([
         tf.keras.layers.Dense(units=1, activation='sigmoid', input_shape=(2,)),
     ])

     model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.3), loss='mse')
     model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 1)                 3
=================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
_________________________________________________________________
```

XOR 구현

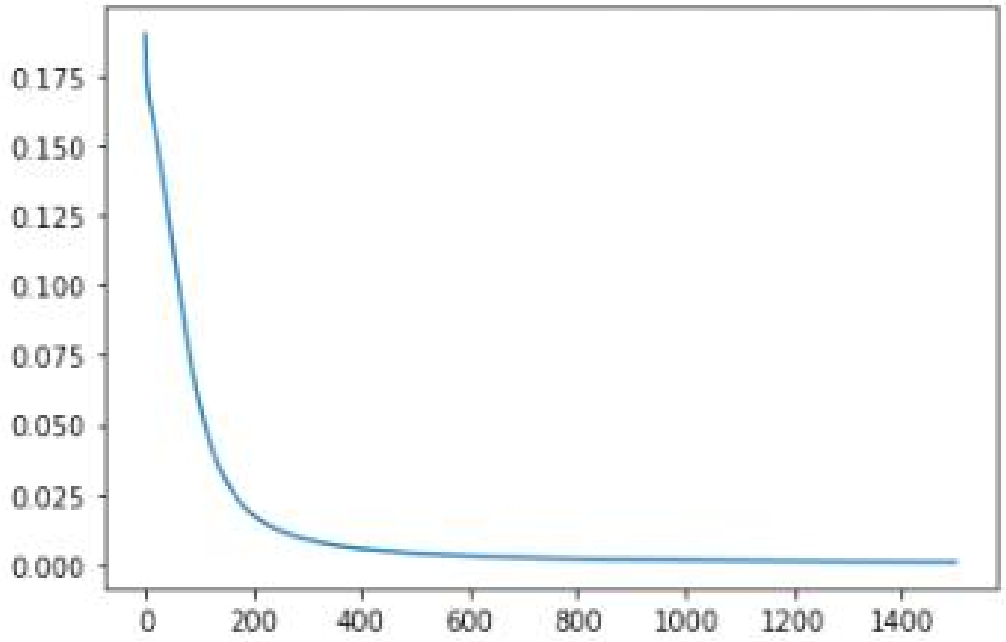
```
[ ]  import numpy as np
     x = np.array([[1,1],[1,0],[0,1],[0,0]])
     y = np.array([[1],[1],[1],[0]])

     model = tf.keras.Sequential([
         tf.keras.layers.Dense(units=2, activation='sigmoid', input_shape=(2,)),
         tf.keras.layers.Dense(units=1, activation='sigmoid')
     ])

     model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.3), loss='mse')

     model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 2)                 6
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 3
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```

XOR을 시각적으로 구현

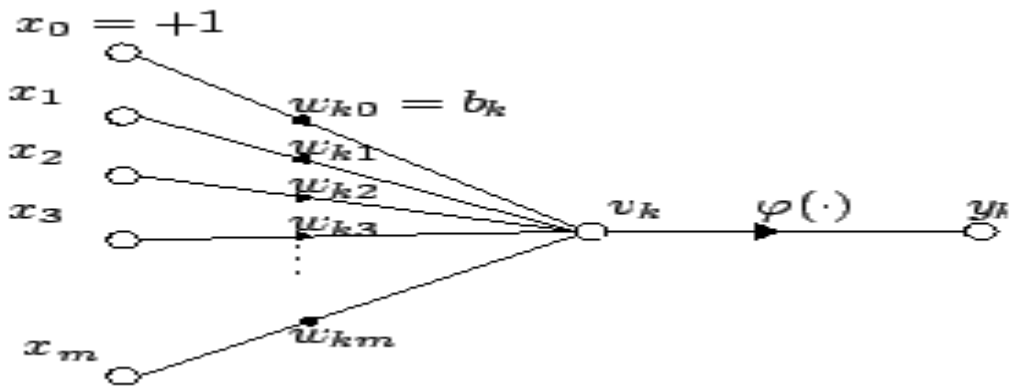
```
[ ]  import matplotlib.pyplot as plt
     plt.plot(history.history['loss'])
```

```
[<matplotlib.lines.Line2D at 0x7f45e301d940>]
```

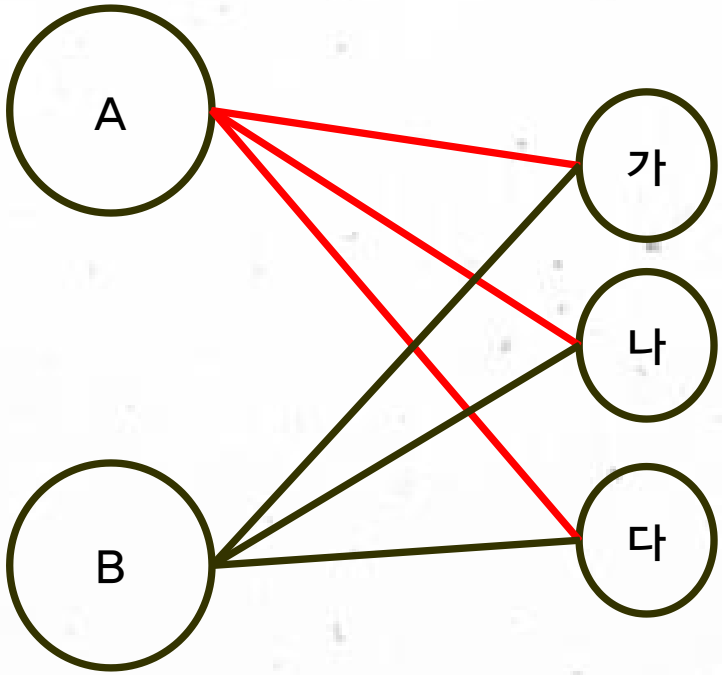


뉴런

- 인간의 사고와 인지에 관심이 있던 인지과학자와 새로운 계산모델에 관심을 갖고 있던 학자들은 신경해부학적 사실을 토대로 하여 간단한 연산기능만을 갖는 처리기(인공 뉴런)를 고안하였다. 그리고 이러한 처리기들을 가중치(weight)를 갖는 채널(데이터 통로)로 연결한 망(network) 형태의 계산모델을 제안하였다. 이렇게 제안된 모델을 인공신경망(artificial neural network)이라 한다. 인공신경망 기본구성요소인 인공뉴런은 그림과 같은 구조를 갖는 처리기이다. 우리가 보통 사용하는 컴퓨터는 한 개 또는 몇 개의 복잡한 기능을 가지고 있는 프로세스가 모든 연산 및 제어를 하고 있는 것과는 달리 인공신경망에서는 단순한 연산기능을 가지고 있는 수많은 인공뉴런이 서로 연결되어 정보를 저장하고 처리한다는 것이 기본 개념이라 할 수 있다.



뉴런의 계산방법



A와B를 하나의 행열 가나다를 다른 행열로 비교하여

(A B) * (A가 A나 A다)
(B가 B나 B다)

Dense 층

- 입력과 출력을 모두 연결해주며, 입력과 출력을 각각 연결해주는 가중치를 포함하고 있다. 즉, 입력이 4개, 출력이 8개라면 가중치는 총 32개가 존재한다. 그리고 이 Dense 레이어는 가장 머신러닝에 기본적인 층으로 영상이나 서로 연속적으로 상관관계가 있는 데이터가 아니라면 이 층을 통해 학습 시킬 수 있는 데이터가 많다.

```
import numpy as np
x = np.array([[1,1],[1,0],[0,1],[0,0]])
y = np.array([[1],[1],[1],[0]])

model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=2, activation='sigmoid', input_shape=(2,)),
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])

model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.3), loss='mse')

model.summary()
```

Dense층을 이용한 xor 구현

회귀모델과 분류모델

회귀모델 : 어떤 연속형 데이터 Y와 Y의 원인이 되는 X간의 관계를 추정하기 위해 만든 관계식

Ex) $Y = f(X)$

분류모델 : 레이블이 달린 학습 데이터로 학습한 후에 새로 입력된 데이터가 학습했던 어느 그룹에 속하는 지를 찾아내는 방법

단순선형회귀

- 종속 변수 Y_i를 하나의 독립 변수 X_i로 설명한다. 두 개 이상의 독립 변수로 설명하는 경우는 중선형 회귀(또는 다중 선형 회귀라고도 함)라 한다.

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i$$

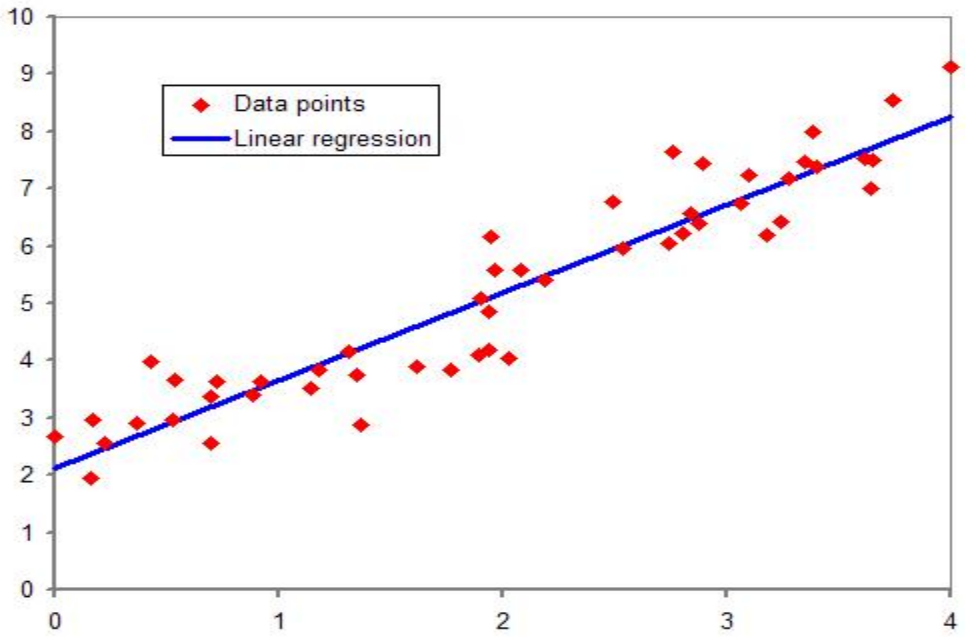
단순선형회귀 식

로지스틱 회귀

- 영국의 통계학자인 D. R. Cox가 1958년[1] 에 제안한 확률 모델로서 독립 변수의 선형 결합을 이용하여 사건의 발생 가능성을 예측하는데 사용되는 통계 기법
- 로지스틱 회귀의 목적은 일반적인 회귀 분석의 목표와 동일하게 종속 변수와 독립 변수간의 관계를 구체적인 함수로 나타내어 향후 예측 모델에 사용하는 것이다. 이는 독립 변수의 선형 결합으로 종속 변수를 설명한다는 관점에서는 선형 회귀 분석과 유사

선형회귀

- 종속 변수 y와 한 개 이상의 독립 변수 (또는 설명 변수) X와의 선형 상관 관계를 모델링하는 회귀분석 기법



선형회귀의 그레프

손실함수

- 신경망이 학습할 수 있도록 해주는 지표
- 머신러닝 모델의 출력값과 사용자가 원하는 출력값의 차이, 즉 오차를 말한다.

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2$$

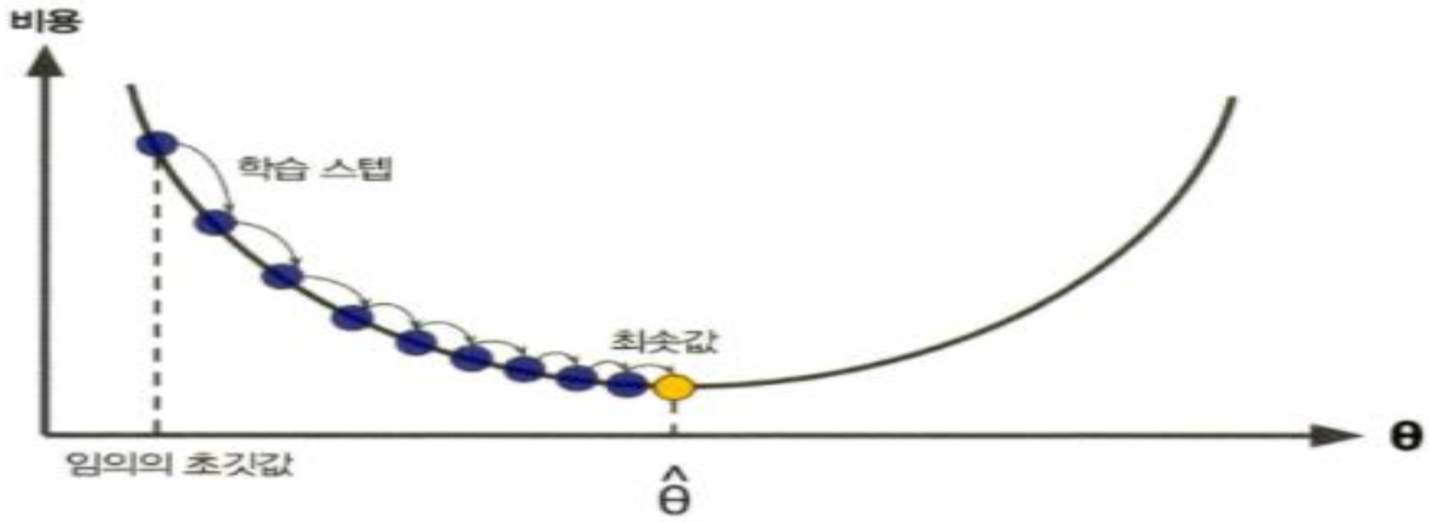
평균제곱 오차 식

$$E = -\sum_{k} t_k \log y_k$$

교차 엔트로피 오차 식

경사하강법

- 1차 근삿값 발견용 최적화 알고리즘이며 기본 개념은 함수의 기울기(경사)를 구하고 경사의 절댓값이 낮은 쪽으로 계속 이동시켜 극값에 이를 때까지 반복시키는 것



초매개변수

- 가중치(weight)같이 모델이 스스로 설정 및 갱신하는 매개변수가 아닌, 사람이 직접 설정해주어야 하는 매개변수를 말한다.신경망에서는 뉴런의 수, 배치(batch)의 크기, 학습률(learning rate), 가중치 감소시의 규제 강도(regularization strength) 등이 있다. 이러한 초매개변수 값에 따라 모델의 성능이 크게 좌우되기도 한다.

선형회귀 케라스 구현

```
import tensorflow as tf

x_train = [1, 2, 3, 4]
y_train = [2, 4, 6, 8]

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(1, input_shape=(1, ), activation='linear')
])

model.compile(optimizer='SGD', loss='mse',
              metrics=['mae', 'mse'])
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

문제와 정답 데이터 지정

모델 구성

학습에 필요한 최적화 방법과
손실함수 등 지정하고
훈련에 사용할 옵티마이저와
손실 함수, 출력 정보를 지정

모델 표시

선형 회귀 모델 학습

```
history = model.fit(x_train, y_train, epochs = 500)
```

생성된 모델로 훈련 데이터 학습
훈련과정 정보를 history 객체에 저장

```
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 3ms/step - loss: 0.0028 - mae: 0.0472 - mse: 0.0028
손실 [0.0027943099848926067, 0.04715406894683838, 0.0027943099848926067]
```

↓

테스트로 성능평가

선형회귀모델 성능평가 및 예측

```
print(model.predict([3.5, 5, 5.5, 6]))

pred = model.predict([3.5, 5, 5.5, 6])
print(pred.flatten())
print(pred.squeeze())
```

```
[[ 6.9759994]
 [ 9.911698 ]
 [10.8902645]
 [11.868832 ]]
[ 6.9759994  9.911698  10.8902645 11.868832 ]
[ 6.9759994  9.911698  10.8902645 11.868832 ]
```

x = [3.5, 5, 5.5, 6]예측

예측 값만 1차원으로

손실함수 mse

mse란?

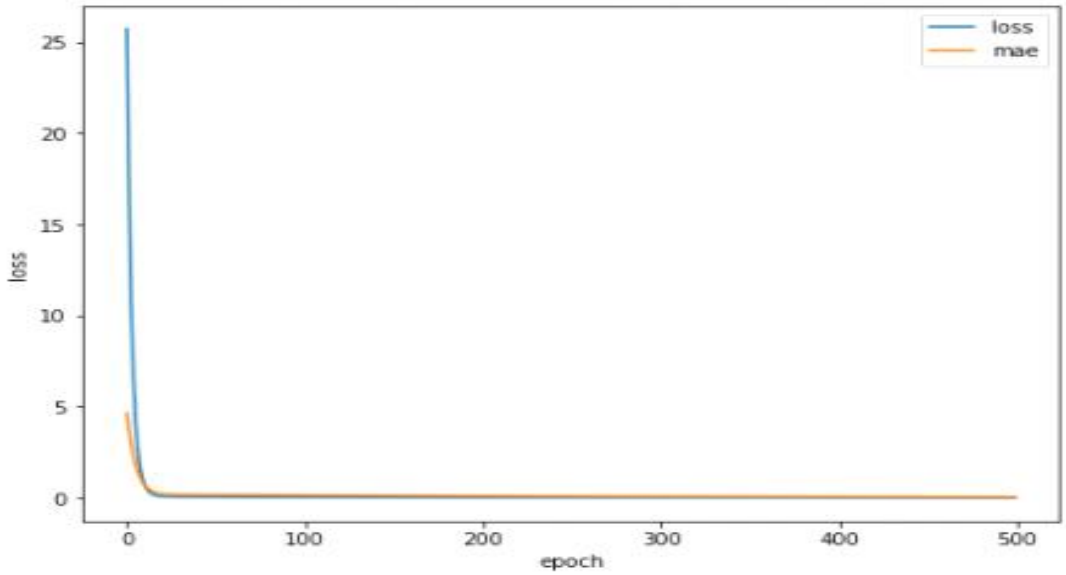
- 계산이 간편하여 가장 많이 사용되는 손실 함수이다.
- 기본적으로 모델의 출력 값과 사용자가 원하는 출력 값 사이의 거리 차이를 오차로 사용한다. 그러나 오차를 계산할 때 단순히 거리 차이를 합산하여 평균내게 되면, 거리가 음수로 나왔을 때 합산된 오차가 실제 오차보다 줄어드는 경우가 생기므로 각 거리 차이를 제곱하여 합산한 후에 평균을 낸다.

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2$$

손실과 mae 시각화로 나타내기

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8,6))

plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['mae'], label='mae')

plt.legend(loc='best')
plt.xlabel('epoch')
plt.ylabel('loss')
```

```
Text(0, 0.5, 'loss')
```



오차역전파

- 계산이 복잡하고 계속해서 계산 과정을 거듭할수록 출력된 결괏값은 생각하지 못한 이유로 정답과의 오차가 있을 확률이 높아진다. 그러므로 복잡한 계산과 오차가 있음을 가정하고, 오차를 줄이는 방법

역전파와 순전파

역전파

- 계산 결과와 정답의 오차를 구해 이 오차에 관여하는 값들의 가중치를 수정하여 오차가 작아지는 방향으로 일정 횟수를 반복해 수정하는 방법이다.
- 횟수가 커지면 그만큼 정확성이 높아지지만, 시간이 오래 걸린다는 단점이 있고, 횟수가 작아지면 정확성이 떨어지지만, 시간을 단축하는 장점이 있다.

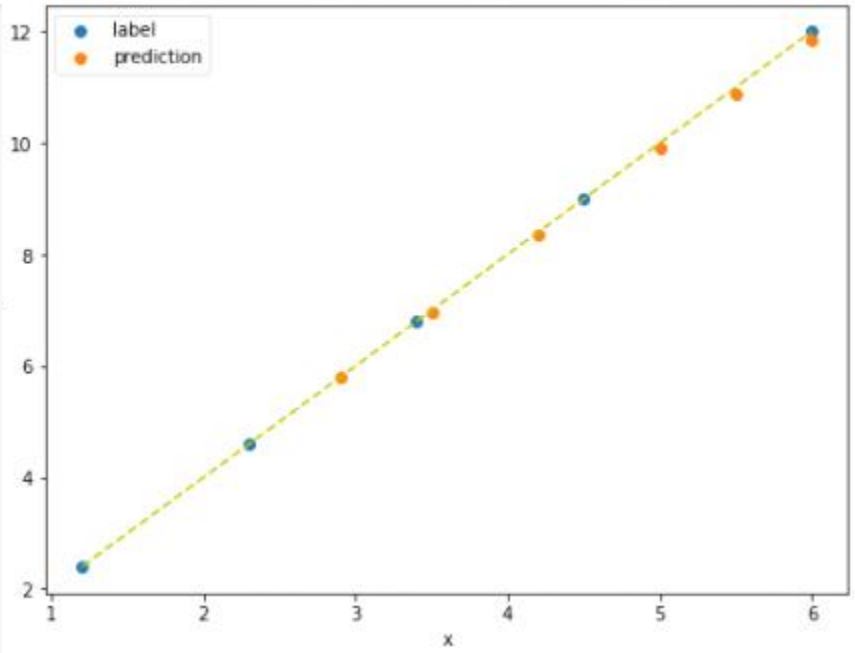
순전파

- 입력층에서 전달 되는 모든 값이 은닉층을 통해 출력층까지 전달되는 방식이다.

예측값 시각화

```
import matplotlib.pyplot as plt

x_test = [1.2, 2.3, 3.4, 4.5, 6.0]
y_test = [2.4, 4.6, 6.8, 9.0, 12.0]

fig = plt.figure(figsize = (8, 6))
plt.scatter(x_test, y_test, label='label')
plt.plot(x_test, y_test, 'y--')

x = [2.9, 3.5, 4.2, 5, 5.5, 6]
pred = model.predict(x)
plt.scatter(x, pred.flatten(), label='prediction')

plt.legend(loc='best')
plt.xlabel('x')
plt.ylabel('y')
```

```
Text(0, 0.5, 'y')
```



tanh 출력 코드 및 그래프

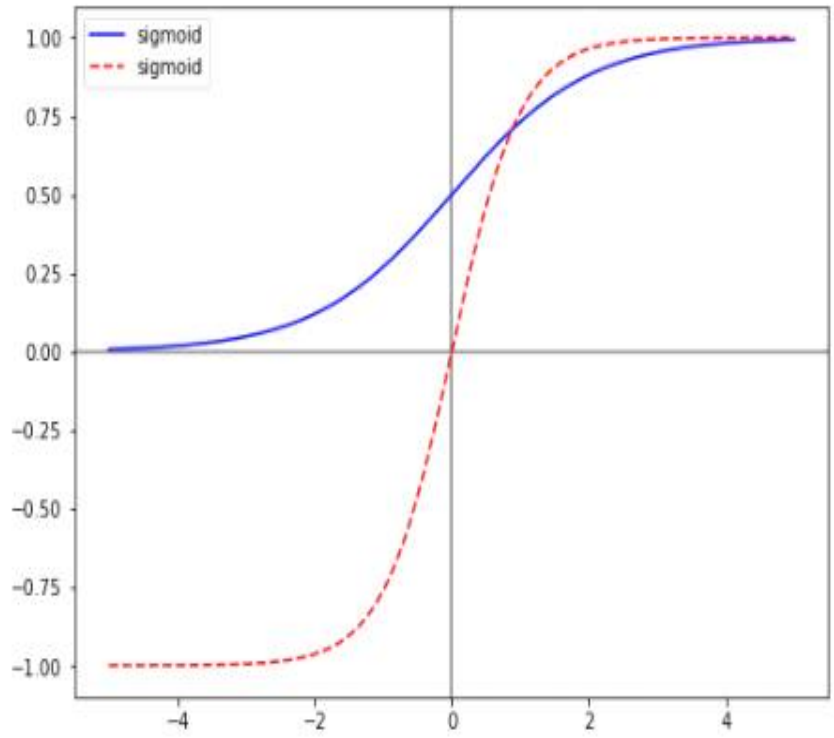
```
import numpy as np
```

```
[ ]  import math
     import matplotlib.pylab as plt
```

```
[ ]  def sigmoid(x):
       return 1/ (1 + math.exp(-x))
```

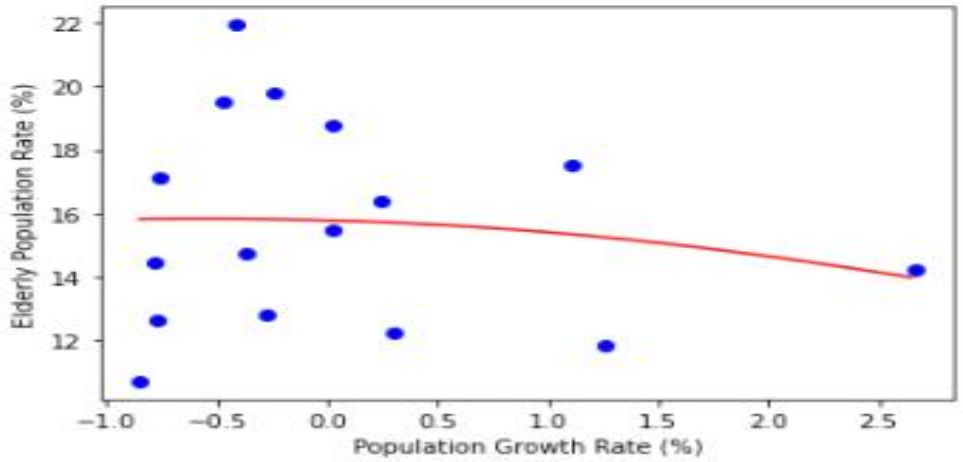
```
[ ]  x = np.arange(-5, 5, 0.01)
     sigmoid_x = [sigmoid(z) for z in x]
     tanh_x = [math.tanh(z) for z in x]
```

```
[ ]  plt.figure(figsize = (8, 6))

     plt.axhline(0, color='gray')
     plt.axvline(0, color='gray')
     plt.plot(x, sigmoid_x, 'b-', label='sigmoid')
     plt.plot(x, tanh_x, 'r--', label='sigmoid')
     plt.legend()
     plt.show()
```



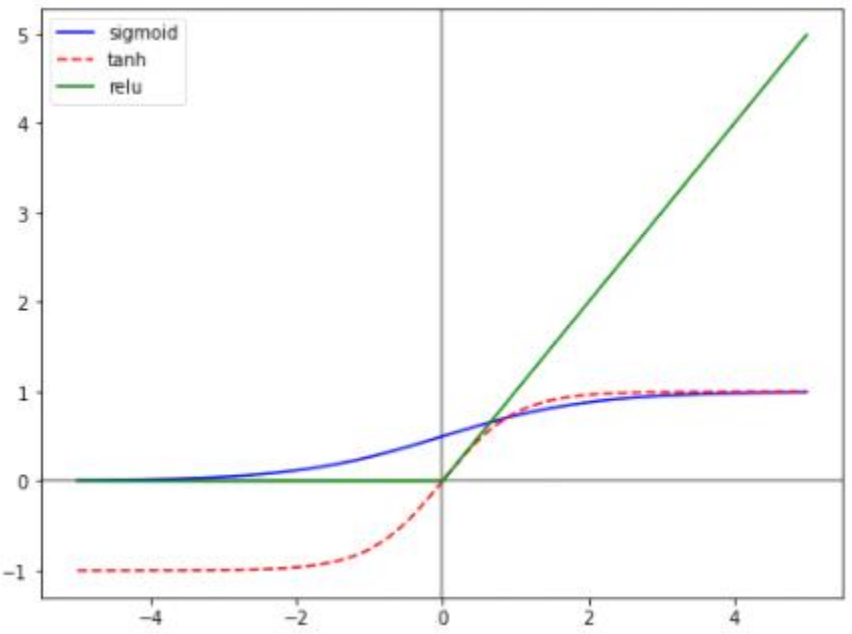
딥러닝 네트워크 회귀선 확인

```
import matplotlib.pyplot as plt

line_x = np.arange(min(x), max(x), 0.01)
line_y = model.predict(line_x)

plt.plot(line_x, line_y, 'r-')
plt.plot(x,y,'bo')

plt.xlabel('Population Growth Rate (%)')
plt.ylabel('Elderly Population Rate (%)')
plt.show()
```



활성화 함수

```
import numpy as np
import matplotlib.pyplot as plt
import math
def sigmoid(x):
  return 1 / (1 + math.exp(-x))

x = np.arange(-5, 5, 0.01)
sigmoid_x = [sigmoid(z) for z in x]
tanh_x = [math.tanh(z) for z in x]
relu = [0 if z < 0 else z for z in x]

plt.figure(figsize = (8, 6))

plt.axhline(0, color='gray')
plt.axvline(0, color='gray')
plt.plot(x, sigmoid_x, 'b-', label='sigmoid')
plt.plot(x, tanh_x, 'r--', label='tanh')
plt.plot(x, relu, 'g', label='relu')
plt.legend()
plt.show()
```



크로스 엔트로피 손실 함수

원핫 인코딩을 이용한 크로스 엔트로피 손실 함수

```
#크로스 엔트로피 손실 함수(원핫 인코딩)
import tensorflow as tf

y_true = [[0, 1, 0], [0, 0, 1]] #원핫 인코딩으로 정답이 있어야함 tf.keras.losses.categorical_crossentropy
y_pred = [[0.05, 0.95, 0], [0.1, 0.8, 0.1]] #첫번째 정답 두번째 오답
loss = tf.keras.losses.categorical_crossentropy(y_true, y_pred)
loss.numpy()
```

```
array([0.05129331, 2.3025851 ], dtype=float32)
```

일반 유형을 이용한 크로스 엔트로피 손실 함수

```
[3] #크로스 엔트로피 손실 함수(일반유형)
    import tensorflow as tf

    y_true = [[1], [2]] #일반유형 값들은 정답
    y_pred = [[0.05,0.95, 0], [0.1, 0.8, 0.1]]
    loss = tf.keras.losses.sparse_categorical_crossentropy(y_true, y_pred)
    loss.numpy()
```

```
array([0.05129344, 2.3025851 ], dtype=float32)
```

크로스 엔트로피 손실 함수

```python
#크로스 엔트로피 손실 함수(일반유형을 원핫인코딩으로 변환)
import tensorflow as tf

y_true = [[1], [2]]
y_true = tf.one_hot(y_true, depth=3)
print(y_true)
y_true = tf.reshape(y_true, [-1, 3])
print(y_true)
y_pred = [[0.05,0.95, 0], [0.1, 0.8, 0.1]]
loss = tf.keras.losses.categorical_crossentropy(y_true, y_pred)
loss.numpy()
```

```
tf.Tensor(
[[[0. 1. 0.]]

 [[0. 0. 1.]]], shape=(2, 1, 3), dtype=float32)
tf.Tensor(
[[0. 1. 0.]
 [0. 0. 1.]], shape=(2, 3), dtype=float32)
array([0.05129331, 2.3025851 ], dtype=float32)
```

크로스 엔트로피 손실 함수

```
[5]  #크로스 엔트로피 손실 값 직접 계산
     import tensorflow as tf
     import numpy as np

     y_true = tf.reshape(tf.one_hot([[1],[2]], depth=3), [-1, 3])
     y_pred = [[0.05, 0.95, 0],[0.1, 0.8, 0.1]]

     loss = tf.keras.losses.categorical_crossentropy(y_true, y_pred)
     print(loss.numpy())

     print(-np.log(0.95), -np.log(0.1))
```

```
[0.05129331 2.3025851 ]
0.05129329438755058 2.3025850929940455
```

소감

이 포트폴리오를 작성하면서 여전히 이해가 안갔던 부분을 알게 되고 또한 새롭게 복습을 하여 완전히 알게되는 계기가 되었던거 같다.
여러가지 모르고 있던 내용들도 다시 확인을 하니 이해가 더욱 잘 되어서 쉽게 외워지는 부분도 있었고 어려웠던 부분도 이제는 풀이를 할 수 있을거 같다.
무엇보다 mnist를 이용한 솔글씨가 가장 어려웠었는데 이 포트폴리오를 작성하면서 다시
한번 더 보니 이해가 되면서 어려웠던 부분도 이제는
풀 수 있을거같다.

감사합니다

Thank you