지능형 컴퓨팅과정 포트폴리오 경진대회

11 월 20 일

컴퓨터정보공학과 PE 반

작성자: 20161886 강연승

인공지능(AI : Artificial Intelligence)

- 컴퓨터가 인간처럼 지적 능력을 갖게 하거나 행동하도록 하는 모든 기술
- 머신러닝(machine learning)
  - 머신러닝은 기계가 스스로 학습할 수 있도록 하는 인공지능의 한 연구 분야
  - SVM(Support Vector Machine) : 수학적인 방식의 학습 알고리즘
  - 딥러닝
    - 다중 계층의 신경망 모델을 사용하는 머신러닝의 일종
    - 특징과 데이터가 많을수록 딥러닝에 적합

머신러닝

- 주어진 데이터를 기반으로 기계가 스스로 학습하여 성능을 향상시키거나 최적의 해답을 찾기 위한 학습 지능 방법
- 스스로 데이터를 반복적으로 학습하여 기술을 터득하는 방식
  - 명시적으로 프로그래밍을 하지 않아도 컴퓨터가 학습을 할 수 있도록 해주는 인공지능의 한 형태
  - 더 많은 데이터가 유입되면, 컴퓨터는 더 많이 학습을 하고, 시간이 흐르면서 더 스마트해져서 작업을 수행하는 능력과 정확도가 향상

머신러닝 분류 개요

- **지도학습**
  - 정답이 있는 예측
  - 올바른 입력과 출력의 쌍으로 구성된 정답의 훈련 데이터로부터 입출력 간의 함수를 학습시키는 방법
    - K-최근접 이웃, 선형 회귀, 로지스틱 회귀, 서포트 벡터 머신, 결정 트리와 랜덤 포레스트
- **비지도(자율)학습**
  - 군집화(클러스터링) 알고리즘
  - 정답이 없는 훈련 데이터를 사용하여 데이터 내에 숨어있는 어떤 관계를 찾아내는 방법
    - Clustering
- **강화학습**
  - 잘한 행동에 대해 보상을 주고 잘못한 행동에 대해 벌을 주는 경험을 통해 지식을 학습하는 방법
    - 딥마인의 알파고
    - 자동 게임분야

● 머신 러닝과 딥 러닝의 차이점

	머신 러닝	딥 러닝
데이터 의존성	중소형 데이터 세트에서 탁월한 성능	큰 데이터 세트에서 뛰어난 성능
하드웨어 의존성	저가형 머신에서 작업 가능	GPU 가 있는 강력한 기계가 필요 DL 은 상당한 양의 행렬 곱셈을 수행
기능 공학	데이터를 나타내는 기능을 이해해야 함	데이터를 나타내는 최고의 기능을 이해할 필요가 없음
실행 시간	몇 분에서 몇 시간	최대 몇 주. 신경망은 상당한 수의 가중치를 계산해야 함

● 퍼셉트론

- 세계 최초의 인공신경망을 제안
- 신경망에서는 방대한 양의 데이터를 신경망으로 유입
  - 데이터를 정확하게 구분하도록 시스템을 학습시켜 원하는 결과를 얻어냄
- 현재, 발전해 여러 분야에서 활용
  - 항공기나 드론의 자율비행, 자동차의 자율 주행, 필기체 인식, 음성인식에 이용, 언어 번역

● 인공신경망(ANN)

- 인간의 신경세포인 뉴런을 모방하여 만든 가상의 신경
- 뇌와 유사한 방식으로 입력되는 정보를 학습하고 판별하는 신경 모델

● MLP

- 입력층과 출력층
  - 다수의 신호를 입력 받아서 하나의 신호를 출력
- 중간의 은닉층
  - 여러 개의 층으로 연결하여 하나의 신경망을 구성

● 심층신경망(DNN)

- 다중 계층인 심층신경망을 사용
  - 학습 성능을 높이는 고유 특징들만 스스로 추출하여 학습하는 알고리즘
  - 입력 값에 대해 여러 단계의 심층신경망을 거쳐 자율적으로 사고 및 결론 도출

- **조건 연산 tf.cond()**
  - tf.cond(pred, true_fn=None, false_fn=None, name=None)
  - pred 를 검사해 참이면 true_fc 반환, 거짓이면 false_fc 반환

tf.cond

```
[6]  x = tf.constant(1.)
     bool = tf.constant(True)
     res = tf.cond(bool, lambda: tf.add(x, 1.), lambda: tf.add(x, 10.))

     print(res)
     print(res.numpy())
```

```
tf.Tensor(2.0, shape=(), dtype=float32)
2.0
```

```
[7]  x = tf.constant(2)
     y = tf.constant(5)
     def f1(): return tf.multiply(x, 17)
     def f2(): return tf.add(y, 23)
     r = tf.cond(tf.less(x, y), f1, f2)

     r.numpy()
```

```
34
```

- **1 차원 배열 텐서**

```
[14] # 1차원 배열 텐서
     t = tf.constant([1, 2, 3])
     t
```

```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([1, 2, 3], dtype=int32)>
```

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

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

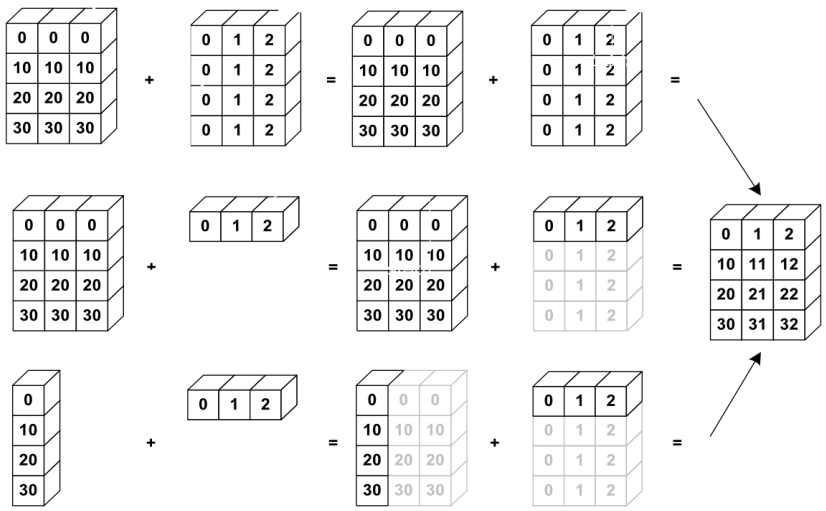
```
[ 6  8 10]
```

```
[16] 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 텐서 연산

- Shape 이 다르더라도 연산이 가능하도록 가지고 있는 값을 이용하여 shape 을 맞춤



● Numpy

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

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

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

```
[44] import numpy as np

     print(np.arange(3))
     print(np.ones((3, 3)))
     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)
```

```
[45] 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]]
```

● 텐서플로 연산

- tf.add() 덧셈
- tf.multiply()
- tf.pow()
- tf.reduce_mean()
- tf.reduce_sum()

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

```
5
```

```
[47] x = 2
     y = 3
     add_op = tf.add(x, y)
     mul_op = tf.multiply(x, y)
     pow_op = tf.pow(add_op, mul_op)

     print(pow_op.numpy())
```

```
15625
```

```
[48] a = tf.constant(2.)
     b = tf.constant(3.)
     c = tf.constant(5.)

     # Some more operations.
     mean = tf.reduce_mean([a, b, c])
     sum = tf.reduce_sum([a, b, c])

     print("mean = ", mean.numpy())
     print("sum = ", sum.numpy())
```

```
mean =  3.3333333
sum =  10.0
```

● 브로드캐스팅

```
[11] # 브로드캐스팅(Broadcasting) 지원
     a = tf.constant([[1, 2],
                      [3, 4]])
     b = tf.add(a, 1)
     print(b)
```

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

● tf.matmul() 곱셈

```
[50] # Matrix multiplications 1
     matrix1 = tf.constant([[1., 2.], [3., 4.]])
     matrix2 = tf.constant([[2., 0.], [1., 2.]])

     gop = tf.matmul(matrix1, matrix2)
     print(gop.numpy())

     # Matrix multiplications 2
     gop = tf.matmul(matrix2, matrix1)
     print(gop.numpy())
```

```
[[ 4.  4.]
 [10.  8.]]
[[ 2.  4.]
 [ 7. 10.]]
```

● 행렬, 원소와의 곱

```
[15] # 연산자 오버로딩 지원
     print(a)
     # 텐서로부터 numpy 값 얻기:
     print(a.numpy())
     print(b)
     print(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)
[[2 3]
 [4 5]]
tf.Tensor(
[[ 2  6]
 [12 20]], shape=(2, 2), dtype=int32)
```

```
[14] # NumPy값 사용
     import numpy as np

     c = np.multiply(a, b)
     print(c)
```

```
[[ 2  6]
 [12 20]]
```

tf.rank 행렬의 차수반환

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

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

```
[19] tf.rank(my_image)
```

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

Shape 와 reshape

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

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

```
[29] 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)
```

```
# 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)>
```

reshape 에서 -1 사용

```
[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)>
```

tf.cast

- **tf.Tensor 의 자료형을 다른 것으로 변경**

```
[48] # 정수형 텐서를 실수형으로 변환.
     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)>
```

```
[49] float_tensor.dtype
```

```
tf.float32
```

변수 Variable

- 변수로 사용
  - 텐서플로 그래프에서 tf.Variable 의 값을 사용하려면 이를 단순히 tf.Tensor 로 취급
- 메소드 assign, assing_add
  - 값을 변수에 할당
- 메소드 read_value
  - 현재 변수 값 읽기

```
[39] v = tf.Variable(0.0)
     v
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.0>
```

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

```
<tf.Tensor: shape=(), dtype=float32, numpy=17.0>
```

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

```
10.0
```

```
[46] v = tf.Variable(2.0)
     v.assign_add(5)
     v
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=7.0>
```

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

```
<tf.Tensor: shape=(), dtype=float32, numpy=7.0>
```

● 균등분포 난수

- tf.random.uniform([1], 0, 1)
  - 배열, [시작, 끝]

```
[6]  1 # 3.7 랜덤한 수 얻기 (균일 분포)
     2 rand = tf.random.uniform([1],0,1)
     3 print(rand)
```

```
tf.Tensor([0.5543064], shape=(1,), dtype=float32)
```

```
[8]  1 rand = tf.random.uniform([5, 4],0,1)
     2 print(rand)
```

```
tf.Tensor(
[[0.43681145 0.84187937 0.9562702  0.7846168 ]
 [0.6079582  0.95665395 0.9038415  0.19482386]
 [0.51012075 0.8609252  0.9433547  0.9636986 ]
 [0.2134043  0.9559026  0.5170028  0.4017253 ]
 [0.0141474  0.15949261 0.23697984 0.7221806 ]], shape=(5, 4), dtype=float32)
```

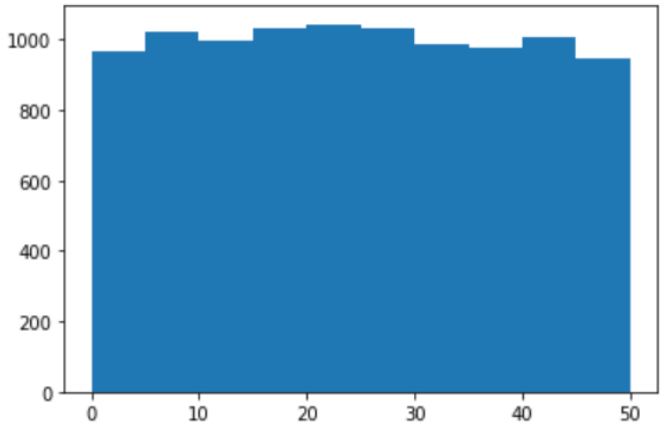
```
[11]  1 rand = tf.random.uniform([1000],0,10)
      2 print(rand[:10])
```

```
tf.Tensor(
[5.1413307 1.548909  8.911686  9.880335  5.5388713 5.6710424 6.80269
 1.9444573 7.549943  6.573516 ], shape=(10,), dtype=float32)
```

● 균등 분포 1000 개 그리기

```
[14]  1 import matplotlib.pyplot as plt
      2 rand = tf.random.uniform([10000],0,50)
      3 plt.hist(rand, bins=10)
```

```
(array([ 965., 1020.,  994., 1032., 1043., 1030.,  987.,  976., 1008.,
         945.]),
 array([6.0796738e-04, 4.9998469e+00, 9.9990854e+00, 1.4998324e+01,
        1.9997562e+01, 2.4996801e+01, 2.9996040e+01, 3.4995281e+01,
        3.9994518e+01, 4.4993759e+01, 4.9992996e+01], dtype=float32),
 <a list of 10 Patch objects>)
```



정규 분포 난수

- tf.random.normal([4],0,1)
  - 크기, 평균, 표준편차

```
[53]  1 # 3.9 랜덤한 수 여러 개 얻기 (정규 분포)
      2 rand = tf.random.normal([4],0,1)
      3 print(rand)
```

```
tf.Tensor([-0.5962639   0.47093895  1.9455601  -0.42773333], shape=(4,), dtype=float32)
```

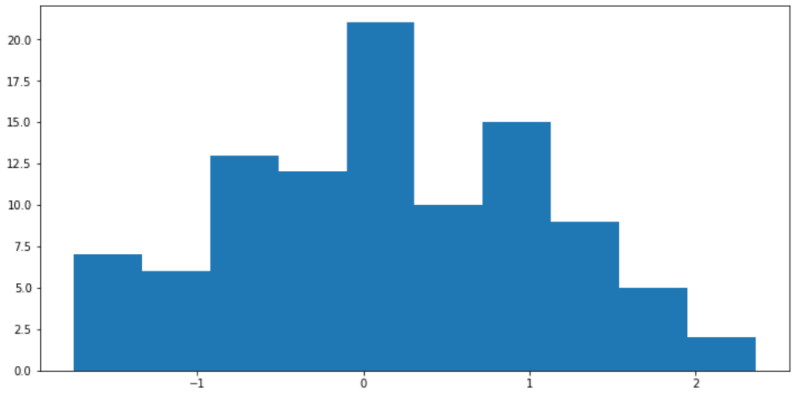
```
[54]  1 # 3.9 랜덤한 수 여러 개 얻기 (정규 분포)
      2 rand = tf.random.normal([2, 4],0,2)
      3 print(rand)
```

```
tf.Tensor(
[[-2.145662    0.64699423  2.0760484  -1.4640687 ]
 [ 1.3588632  -0.9740333   1.4347676  -1.3747462 ]], shape=(2, 4), dtype=float32)
```

정규 분포 100 개 그리기

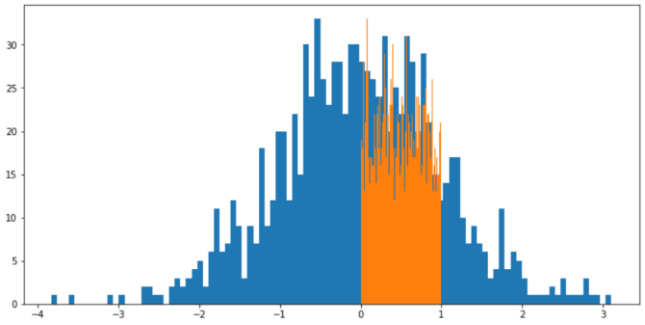
```
[52]  1 import matplotlib.pyplot as plt
      2 rand = tf.random.normal([100], 0, 1)
      3 plt.hist(rand, bins=10)
```

```
(array([ 7.,  6., 13., 12., 21., 10., 15.,  9.,  5.,  2.]),
 array([-1.7424849 , -1.332153  , -0.921821  , -0.511489  , -0.10115702,
         0.30917495,  0.7195069 ,  1.129839  ,  1.5401709 ,  1.9505029 ,
         2.3608348 ], dtype=float32),
 <a list of 10 Patch objects>)
```



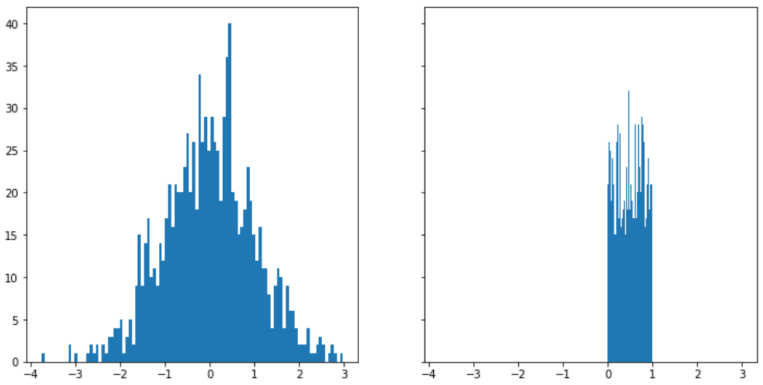
균등분포와 정규분포의 비교

```
[57]  1 import matplotlib.pyplot as plt
      2 rand1 = tf.random.normal([1000],0, 1)
      3 rand2 = tf.random.uniform([2000], 0, 1)
      4 plt.hist(rand1, bins=100)
      5 plt.hist(rand2, bins=100)
```



● 균등분포와 정규분포를 부분으로 그리기

```
import matplotlib.pyplot as plt
rand1 = tf.random.normal([1000],0, 1)
rand2 = tf.random.uniform([2000], 0, 1)

plt.rcParams["figure.figsize"] = (12,6)
fig, axes = plt.subplots(1, 2, sharex=True, sharey=True)
axes[0].hist(rand1, bins=100)
axes[1].hist(rand2, bins=100)
```



● Shuffle

- tf.random.shuffle(a)

```
[29]  import numpy as np
      a = np.arange(10)
      print(a)
      tf.random.shuffle(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([7, 9, 1, 4, 3, 5, 8, 6, 2, 0])>
```

```
[26]  import numpy as np
      a = np.arange(20).reshape(4, 5)
      a
```

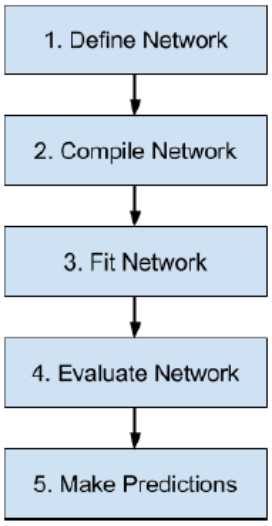
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```
[27]  tf.random.shuffle(a)
```

```
<tf.Tensor: shape=(4, 5), dtype=int64, numpy=
array([[ 0,  1,  2,  3,  4],
       [15, 16, 17, 18, 19],
       [10, 11, 12, 13, 14],
       [ 5,  6,  7,  8,  9]])>
```

● 케라스 딥러닝 구현

- Define : 딥러닝 모델을 생성
- Compile : 주요 훈련방법 설정
  - 최적화 방법
  - 손실 함수
  - 훈련 모니터링 지표
- Fit : 훈련
- Evaluate : 테스트 데이터 평가
- Predict : 정답 예측



● MNIST 손 글씨 데이터 로드

- 훈련 데이터 손 글씨와 정답 : x_train, y_train (6 만개)
- 테스트 데이터 손 글씨와 정답 : x_test, y_test (1 만개)

```
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()
```

● MNIST 손 글씨 데이터 구조

① X_train[0], y_train[0]

```
[62]  1 x_train.shape

(60000, 28, 28)

[68]  1 y_train.shape

(60000,)

[70]  1 x_train[0]

array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
         18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170,
        253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253,
        253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,
          0,   0],

[71]  1 y_train[0]

5
```

② MNIST 데이터셋을 훈련과 테스트 데이터로 로드하여 준비

(x_train, y_train), (x_test, y_test) = mnist.load_data()

③ MNIST 형태 - 데이터 수, 행렬 형태 등

print(x_train.shape, y_train.shape) -> (60000, 28, 28) (60000,)

print(x_test.shape, y_test.shape) -> (10000, 28, 28) (10000,)

④ MNIST 훈련 데이터의 내부 첫 내용

print(x_train[0]) -> [[...]]

print(y_train[0]) -> 5

⑤ MNIST 테스트 데이터의 내부 첫 내용

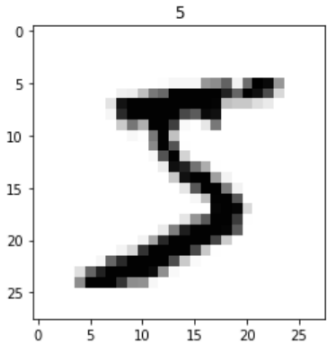
print(x_test[0]) -> [[...]]

print(y_test[0]) -> 7

⑥ 훈련데이터 첫 손 글씨

```
[67]  1 n = 0
      2 ttl = str(y_train[n])
      3 plt.figure(figsize=(6, 4))
      4 plt.title(ttl)
      5 plt.imshow(x_train[n], cmap='Greys')
```

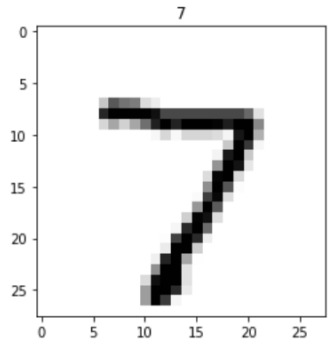
```
<matplotlib.image.AxesImage at 0x7faf8ba0bda0>
```



- 테스트 데이터 첫 손 글씨

```
[74]  1 n = 0
      2 ttl = str(y_test[n])
      3 plt.figure(figsize=(6, 4))
      4 plt.title(ttl)
      5 plt.imshow(x_test[n], cmap='Greys')
```

```
<matplotlib.image.AxesImage at 0x7faf8c402be0>
```



● 딥러닝 구현 순서

① 훈련과 정답 데이터 지정

- 데이터 전처리(옵션) : 샘플 값을 정수에서 부동 소수로 변환 (한 비트의 값을 255 로 나눔)

```
import tensoflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

# 샘플 값을 정수(0~255)에서 부동소수(0~1)로 변환
x_train, x_test = x_train / 255.0, x_test / 255.0
```

② 모델 구성

- 층을 차례대로 쌓아 tf.keras.models.Sequential 모델을 생성
- 신경망 구성

```
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')
])
```

- Neural Networks
  - 입력층
  - 중간층(은닉층)
  - 출력층
- Flatten(input_shape=(28,28)), - 모델에서 2 차원 그림을 1 차원으로 평탄화
  - 60000 개의 (28,28) 크기를 가진 배열
- Dense() – 완전연결층

③ 학습에 필요한 최적화 방법과 손실 함수 등 설정 / 구성된 모델 요약(옵션)

- 훈련에 사용할 옵티마이저와 손실 함수 등을 선택
- 옵티마이저 : 입력된 데이터와 손실함수를 기반으로 모델을 업데이트하는 메커니즘
- 손실 함수 : 훈련 데이터에서 신경망의 성능을 측정하는 방법
  모델이 옳은 방향으로 학습될 수 있도록 도와주는 기준 값
- 훈련과 테스트 과정을 모니터링할 지표 – 정확도만 고려
- 모델요약 : compile 전에도 summary() 가능

```
# 훈련에 사용할 옵티마이저(optimizer)와 손실 함수, 출력정보를 모델에 설정
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
              # metrics=['accuracy', 'mse'])

# 모델 요약 표시
model.summary()
```

- model.summary()
  - 각 층의 구조와 파라미터 수 표시 – 가중치와 편향
  - 총 파라미터 수 – 모델이 구해야 할 수의 개수

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

④ 생성된 모델로 훈련 데이터 학습

- 모델을 훈련 : model.fit() – 훈련 횟수 epochs 에 지정

```
# 모델을 훈련 데이터로 총 5번 훈련
model.fit(x_train, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0248 - accuracy: 0.9913
Epoch 2/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0234 - accuracy: 0.9920
Epoch 3/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0236 - accuracy: 0.9918
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0232 - accuracy: 0.9920
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0229 - accuracy: 0.9922
<tensorflow.python.keras.callbacks.History at 0x7f4a78e6cd30>
```

⑤ 테스트 데이터로 성능 평가

- 테스트 세트에서도 모델이 잘 작동하는지 확인
- model.evaluate() – 손실 값과 예측 정확도 반환

```
313/313 [==============================] - 0s 1ms/step - loss: 0.0868 - accuracy: 0.9805
[0.08675415068864822, 0.9804999828338623]
```

98%의 정확도로 손글씨를 맞춤

● model.predict(input)

- input 값
  - 모델의 fit(),evaluate()에 입력과 같은 형태가 필요
- 첫번째 손 글씨만 알아보더라도 3 차원 배열로 입력
  - 슬라이스해서 사용, x_test[:1]
  - Pred_result = model.predict(x_test[:1])

```
[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]
```

● Tensorflow 메소드

- **tf.reduce_sum(), tf.argmax()**

```
import numpy as np

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

# 혹시 가장 큰 수가 있는 첨자가 결과
print(tf.argmax(one_pred).numpy())
```

```
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)
```

```
1.0
7
```

```
[8]  import numpy as np

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

     # 혹시 가장 큰 수가 있는 첨자가 결과
     print(tf.argmax(one_pred).numpy())
```

```
tf.Tensor(1.0, shape=(), dtype=float32)
1.0
7
```

● 메소드 np.argmax()

- 2 차원에서 내부 행의 argmax 를 구하려면

```
[46]  1 import numpy as np
      2
      3 ##############################################
      4 # 원핫 인코딩과 argmax 학습
      5 print(np.argmax([5, 4, 10, 1, 2]))
      6 print(np.argmax([3, 1, 4, 9, 6, 7, 2]))
      7 print(np.argmax([[0.1, 0.8, 0.1], [0.7, 0.2, 0.1], [0.2, 0.1, 0.7]], axis=1))
```

```
2
3
[1 0 2]
```

● 메소드 tf.argmax()

```
[11] import numpy as np

     ##############################################
     # 원핫 인코딩과 argmax 학습
     print(tf.argmax([5, 4, 10, 1, 2]))
     print(tf.argmax([3, 1, 4, 9, 6, 7, 2]))
     print(tf.argmax([[0.1, 0.8, 0.1], [0.7, 0.2, 0.1], [0.2, 0.1, 0.7]], axis=1))
```

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

● 드롭아웃

- 층에서 결과 값을 일정 비율로 제거하는 방법
- 오버피팅 문제를 해결하는 정규화 목적을 위해서 필요
- tf.keras.layers.Dropout(0.2) – 확률 값은 0.2~0.5 를 주로 사용
- 훈련 단계보다 더 많은 유닛이 활성화되기 때문에 균형을 맞추기 위해 층의 출력 값을 드롭아웃 비율만큼 줄이는 방법
- 일반적으로 훈련단계에서 적용 – 드롭아웃을 층에 적용하면 훈련하는 동안 층의 출력 특성을 랜덤하게 끔
- 테스트 단계에서는 어떤 유닛도 드롭아웃하지 않음
- Tf.keras 에서는 Dropout 층을 이용해 네트워크에 드롭아웃을 추가
  - 이 층은 바로 이전 층의 출력에 드롭아웃을 적용

```
[94] data = np.arange(1, 11).reshape(5, 2).astype(np.float32)
     print(data)
     np.sum(data)
```

```
[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]
 [ 9. 10.]]
55.0
```

```
[100] tf.random.set_seed(0)
      #layer = tf.keras.layers.Dropout(.2, input_shape=(2,))
      layer = tf.keras.layers.Dropout(.3, input_shape=(2,))
      outputs = layer(data, training=True)
      #outputs = layer(data, training=False)
      print(outputs)
      np.sum(outputs)
```

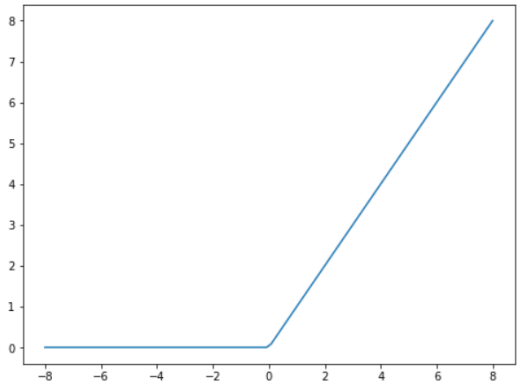
```
tf.Tensor(
[[ 0.          0.        ]
 [ 4.285714    5.714286  ]
 [ 7.1428576   8.571428  ]
 [10.         11.428572  ]
 [12.857143    0.        ]], shape=(5, 2), dtype=float32)
60.0
```

● 활성화 함수 ReLU

- Rectified(정류된) Linear Unit(선형 함수, y=x 를 의미)
  - 선형 함수를 정류하여 0 이하는 모두 0 으로 한 함수
  - max(x,0) – 양수만 사용
- 2010 년 이후 층이 깊어질수록 많이 활용
  - 양수를 그대로 반환하므로 값의 왜곡이 적어지는 효과

```
[45]  1 import numpy as np
      2 import matplotlib.pylab as plt
      3
      4 def relu_func(x): # ReLU(Rectified Linear Unit, 정류된 선형 유닛) 함수
      5     return np.maximum(0, x)
      6     #return (x>0)*x # same
      7
      8 # ReLU 함수 그리기
      9 plt.figure(figsize=(8, 6))
     10 x = np.linspace(-8, 8, 100)
     11 plt.plot(x, relu_func(x))
```

```
[<matplotlib.lines.Line2D at 0x7f93b409b748>]
```

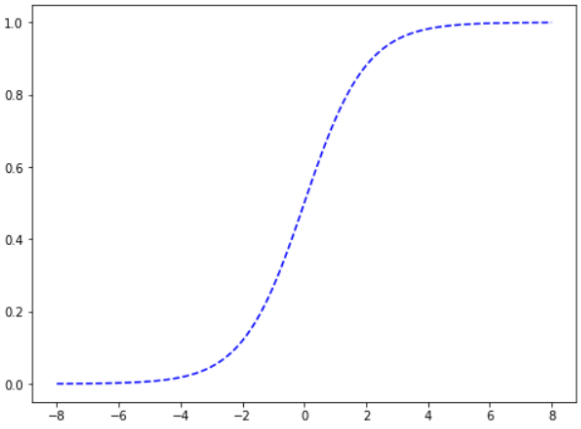


● Sigmoid

- S 자 형태의 곡선이라는 의미 (예전에 많이 사용)

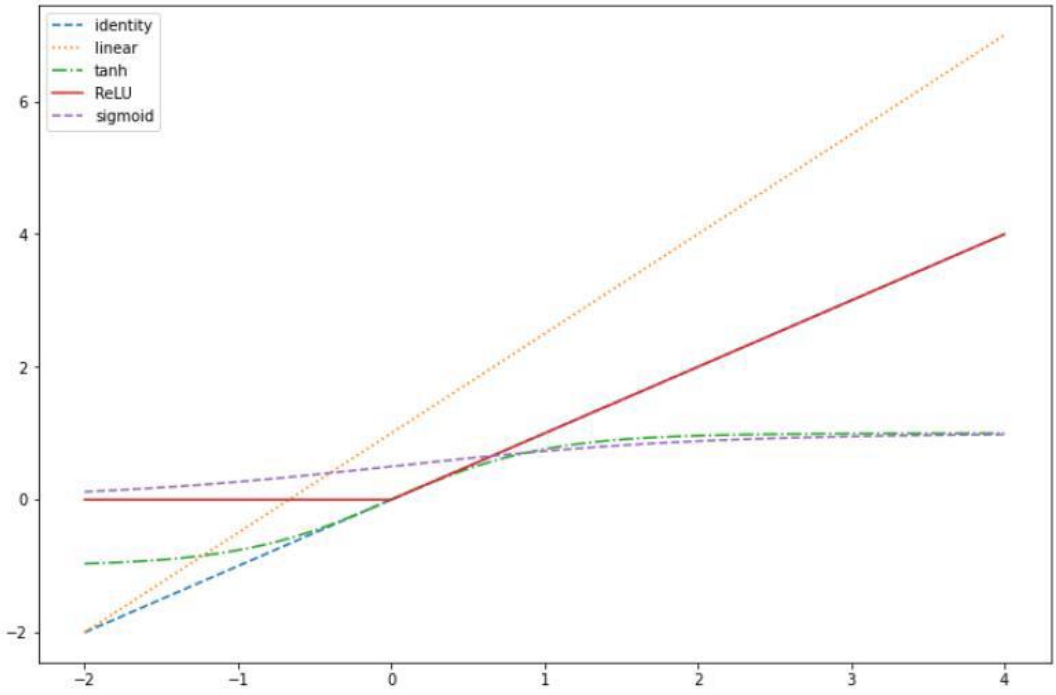
```
[44]  1 import numpy as np
      2 import matplotlib.pylab as plt
      3
      4 def sigm_func(x): # sigmoid 함수
      5     return 1 / (1 + np.exp(-x))
      6
      7 # 시그모이드 함수 그리기
      8 plt.figure(figsize=(8, 6))
      9 x = np.linspace(-8, 8, 100)
     10 plt.plot(x, sigm_func(x), 'b--')
```

```
[<matplotlib.lines.Line2D at 0x7f93b4130cc0>]
```



● 다양한 활성화 함수

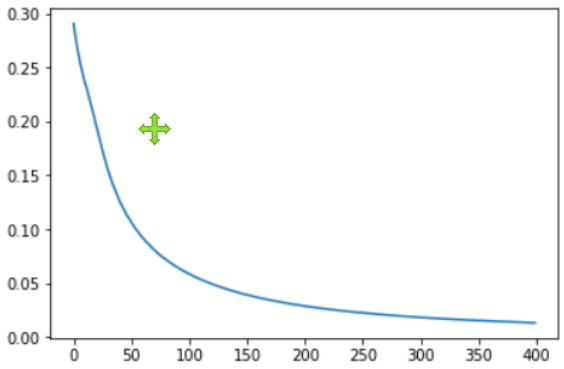
```
import numpy as np
import matplotlib.pylab as plt

def identity_func(x): # 항등함수
    return x

def linear_func(x): # 1차함수
    return 1.5 * x + 1 # a기울기(1.5), Y절편b(1) 조정가능

def tanh_func(x): # TanH 함수
    return np.tanh(x)

def relu_func(x): # ReLU(Rectified Linear Unit, 정류된 선형 유닛) 함수
    return np.maximum(0, x)
    #return (x>0)*x # same

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

# 그래프 그리기
plt.figure(figsize=(12, 8))
x = np.linspace(-2, 4, 100)

plt.plot(x, identity_func(x), linestyle='--', label="identity")
plt.plot(x, linear_func(x), linestyle=':', label="linear")
plt.plot(x, tanh_func(x), linestyle='-.', label="tanh")
plt.plot(x, relu_func(x), linestyle='-', label="ReLU")
plt.plot(x, sigm_func(x), linestyle='--', label="sigmoid")
plt.legend(loc='upper left')
```



- **손실 값 그래프와 결과 예측**

```
[54] # 3.34 2-레이어 XOR 네트워크의 loss 변화를 선 그래프로 표시
     import matplotlib.pyplot as plt
     plt.plot(history.history['loss'])
```

```
[<matplotlib.lines.Line2D at 0x7efe712955f8>]
```




```
[59] model.predict(x)
```

```
array([[0.8535386 ],
       [0.12342612],
       [0.12364785],
       [0.00339741]], dtype=float32)
```

- **가중치와 편향 값 알아보기**

```
[60] for weight in model.weights:
         print(weight)
```

```
<tf.Variable 'dense_6/kernel:0' shape=(2, 1) dtype=float32, numpy=
array([[3.7209592],
       [3.723007 ]], dtype=float32)>
<tf.Variable 'dense_6/bias:0' shape=(1,) dtype=float32, numpy=array([-5.6813374], dtype=float32)>
```

```
[61] model.weights[0]
```

```
<tf.Variable 'dense_6/kernel:0' shape=(2, 1) dtype=float32, numpy=
array([[3.7209592],
       [3.723007 ]], dtype=float32)>
```

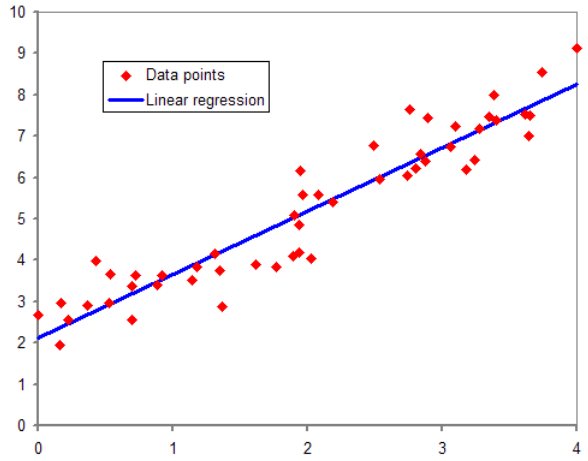
```
[62] model.weights[1]
```

```
<tf.Variable 'dense_6/bias:0' shape=(1,) dtype=float32, numpy=array([-5.6813374], dtype=float32)>
```

- **회귀 모델 – 연속적인 값을 예측 / 분류 모델 – 불 연속적인 값을 예측**
- **단순 선형 회귀 분석**
  - 입력 : 특징이 하나 / 출력 : 하나의 값
- **다중 선형 회귀 분석**
  - 입력 : 특징이 여러 개 / 출력 : 하나의 값
- **로지스틱 회귀(이진 분류)**
- 입력 : 하나 또는 여러 개 / 출력 : 0 아니면 1

선형회귀

- 데이터의 경향성을 가장 잘 설명하는 하나의 직선을 예측하는 방법
- Y=aX+b
- 딥러닝 분야에서는 Y=wX+b 가중치 w 와 편향인 b 를 구하는 것

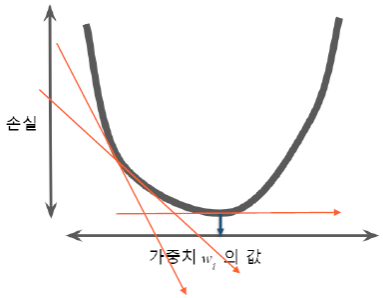


손실함수

- 실제 값과 예측 값에 대한 오차에 대한 식 (목적 함수, 비용 함수라고도 부름)
- 예측 값의 오차를 줄이는 일에 최적화 된 식
- 평균 제곱 오차 등을 사용

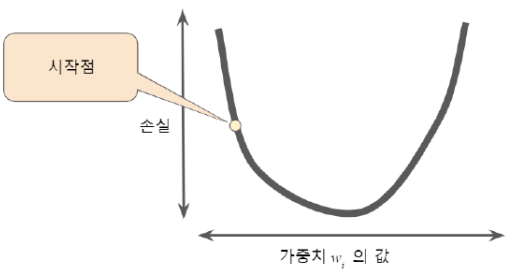
손실과 가중치

- 항상 볼록 함수 모양을 함
- 볼록 문제에는 기울기가 정확하게 0 인 지점인 최소값이 하나만 존재



경사하강법

- 시작 값을 선택, 시작점에서 손실 곡선의 기울기를 계산



- **학습률**
  - 다음 가중치 값 결정 방법
    - 기울기에 학습률을 곱하여 다음 지점을 결정
  - 학습률의 값
    - 너무 작게 설정하면 학습 시간이 매우 오래 걸림
    - 반대로 학습률을 너무 크게 설정하면 다음 지점이 곡선의 최저점을 무질서하게 이탈할 우려가 있음
  - 손실 함수의 기울기가 작다면 더 큰 학습률을 시도해 볼 수 있음
    - 작은 기울기를 보완하고 더 큰 보폭을 만들어 냄
- **오차역전파**
  - 순전파
    - 입력층에서 출력층으로 계산해 최종 오차를 계산하는 방법
  - 역전파
    - 오차 결과 값을 통해서 다시 역으로 input 방향으로 오차가 적어지도록 다시 보내며 가중치를 다시 수정하는 방법
    - 엄청난 처리 속도의 증가

- **선형회귀 케라스 구현**

```
import tensorflow as tf

# ① 문제와 정답 데이터 지정
x_train = [1, 2, 3, 4]
y_train = [2, 4, 6, 8]

# ② 모델 구성(생성)
model = tf.keras.models.Sequential([
    #                   출력, 입력=여러 개 원소의 일차원 배열, 그대로 출력
    tf.keras.layers.Dense(1, input_shape=(1, ), activation='linear')
    #Dense(1, input_dim=1)
])

# ③ 학습에 필요한 최적화 방법과 손실 함수 등 지정
# 훈련에 사용할 옵티마이저(optimizer)와 손실 함수, 출력 정보를 지정
# Mean Absolute Error, Mean Squared Error
model.compile(optimizer='SGD', loss='mse',
              metrics=['mae', 'mse'])
```

```
# 모델을 표시(시각화)
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

● 선형 회귀 모델 학습

- 히스토리 객체
  - 매 에포크 마다의 훈련 손실 값 (loss)
  - 매 에포크 마다의 훈련 정확도 (accuracy)
  - 매 에포크 마다의 검증 손실 값 (val_loss)
  - 매 에포크 마다의 검증 정확도 (val_acc)

```
# ④ 생성된 모델로 훈련 데이터 학습
# 훈련과정 정보를 history 객체에 저장
history = model.fit(x_train, y_train, epochs=500)
```

```
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
```

● 성능 평가

```
# 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]
```

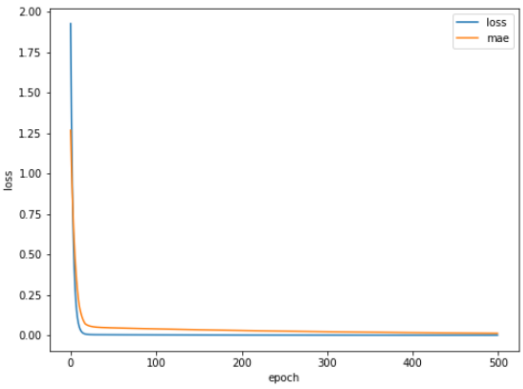
● 예측

```
# 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())
```

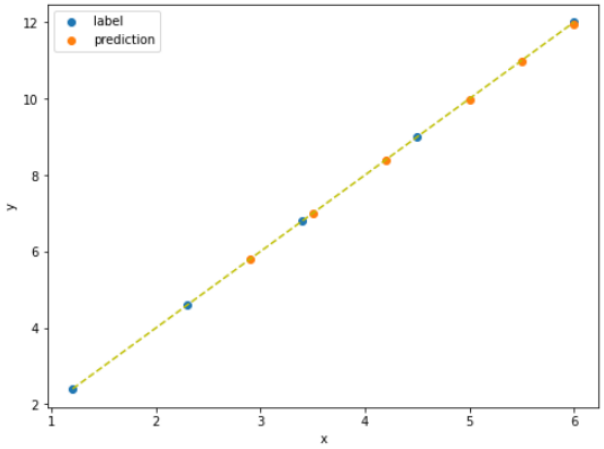
```
[[ 6.9934297]
 [ 9.975829 ]
 [10.969961 ]
 [11.964094 ]]
[ 6.9934297  9.975829  10.969961  11.964094 ]
[ 6.9934297  9.975829  10.969961  11.964094 ]
```

● 손실과 mae 시각화

```
import matplotlib.pylab as plt

# 그래프 그리기
fig = plt.figure(figsize=(8, 6))

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

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



- **예측 값 시각화**




```
import matplotlib.pylab 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]

# 그래프 그리기
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')
```

- **선형회귀 y=2x+1 예측**

  - x = [0, 1, 2, 3, 4]
  - y = [1, 3, 5, ?, ?]

```
import tensorflow as tf
import numpy as np

#훈련과 테스트 데이터
x = np.array([0, 1, 2, 3, 4])
y = np.array([1, 3, 5, 7, 9]) #y = x * 2 + 1

#인공신경망 모델 사용
model = tf.keras.models.Sequential()

#은닉계층 하나 추가
model.add(tf.keras.layers.Dense(1, input_shape=(1,)))

#모델의 패라미터를 지정하고 모델 구조를 생성
#최적화 알고리즘: 확률적 경사 하강법(SGD: Stochastic Gradient Descent)
#손실 함수(loss function): 평균제곱오차(MSE: Mean Square Error)
model.compile('SGD', 'mse')

#생성된 모델로 훈련 자료로 입력(x[:2])과 출력(y[:2])을 사용하여 학습
#키워드 매개변수 epoch(에폭): 훈련반복횟수
#키워드 매개변수 verbose: 학습진행사항 표시
model.fit(x[:3], y[:3], epochs=1000, verbose=0)

#테스트 자료의 결과를 출력
print('Targets(정답):', y[3:])

#학습된 모델로 테스트 자료로 결과를 예측(model.predict)하여 출력
print('Predictions(예측):', model.predict(x[3:]).flatten())
```

- **입력층과 출력층 구성**

```
import tensorflow as tf
import numpy as np

#훈련과 테스트 데이터
x = np.array([0, 1, 2, 3, 4])
y = np.array([1, 3, 5, 7, 9]) #y = x * 2 + 1

#인공신경망 모델 사용
model = tf.keras.models.Sequential()

#은닉계층 하나 추가
model.add(tf.keras.layers.Dense(1, input_shape=(1,)))

#모델의 패라미터를 지정한 후 학습
Model.compile('SGD', 'mse')
Model.fit(x[:3], y[:3], epochs=1000, verbose=0)

print('Targets(정답):', y[3:])
print('Predictions(예측):', model.predict(x[3:]).flatten())
```

- **케라스로 예측 순서**

① 케라스 패키지 임포트
- import tensorflow as tf
- import numpy as np

② 데이터 지정
- x = numpy.array([0, 1, 2, 3, 4])
- y = numpy.array([1, 3, 5, 7, 9]) #y = x * 2 + 1

③ 인공신경망 모델 구성
- model = tf.keras.models.Sequential()
- model.add(tf.keras.layers.Dense(출력수, input_shape=(입력수,)))

④ 최적화 방법과 손실 함수 지정해 인공신경망 모델 생성
- model.compile('SGD', 'mse')

⑤ 생성된 모델로 훈련 데이터 학습
- model.fit(...)

⑥ 성능 평가
- model.evaluate(...)

⑦ 테스트 데이터로 결과 예측
- model.predict(...)