활성화 함수 그리기

실습 파일

- **파일 생성**
  - reg_basic.ipynb

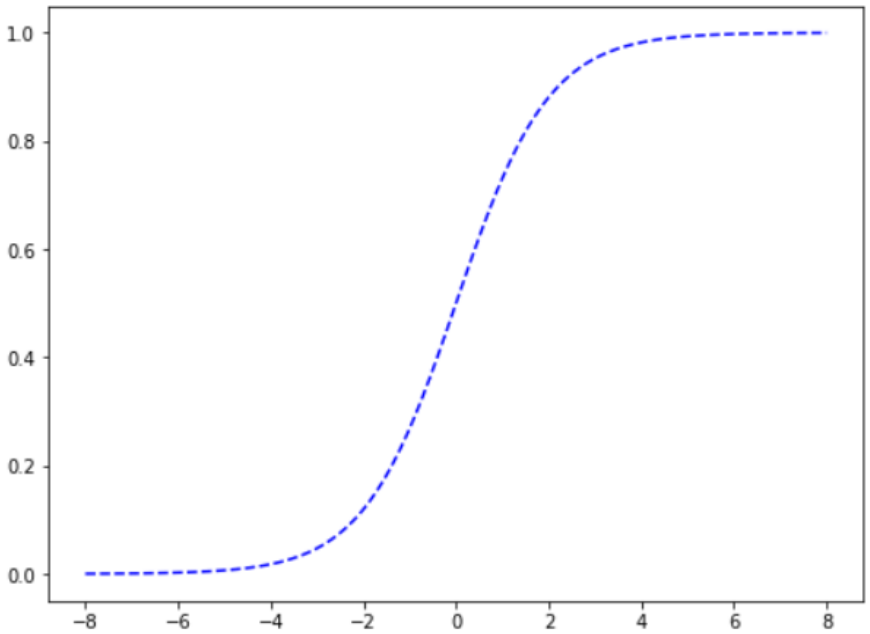
시그모이드 함수

- **S자 곡선**
  - (0, 1) 사이의 값

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

```
[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>]
```



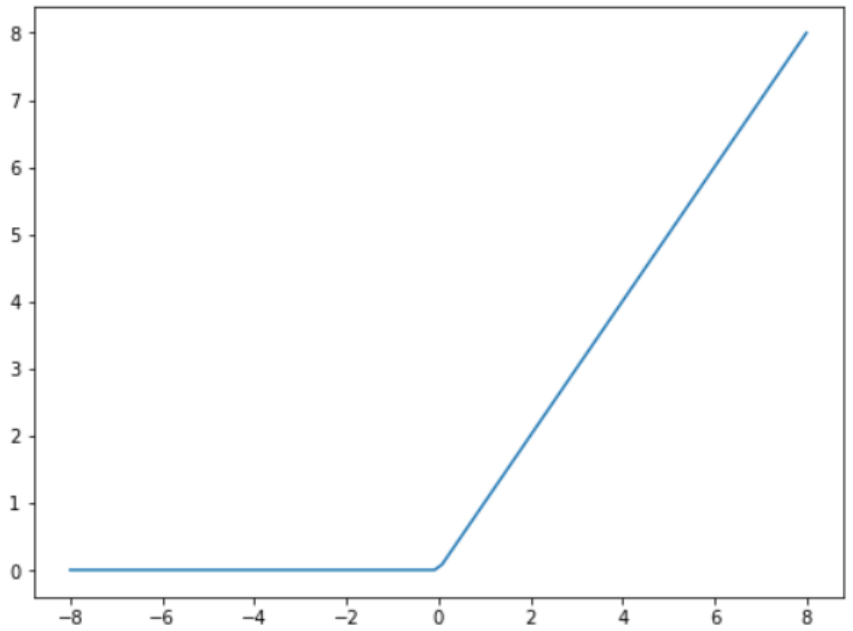
ReLU 함수

- x
  - 0, 음수면 0
  - 양수면 x

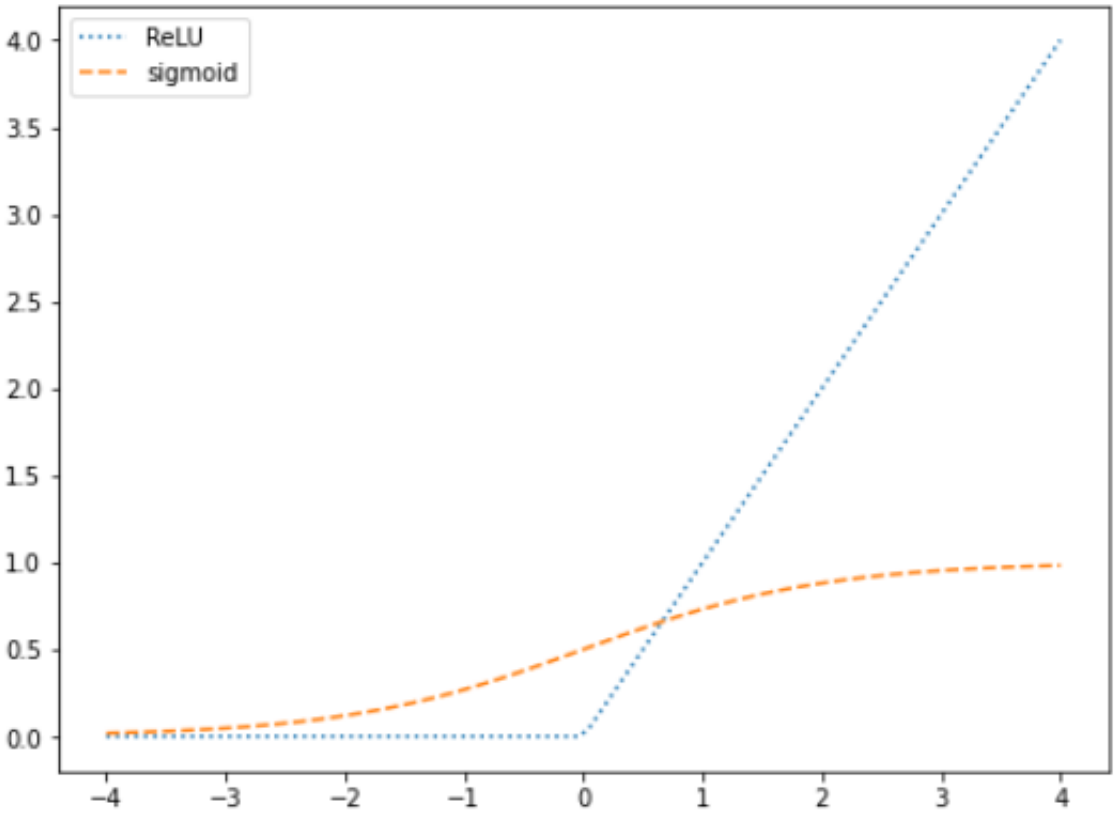
$$h(x) = \begin{cases} 0 \ (x \leq 0) \\ x(x > 0) \end{cases}$$

```
[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>]
```



시그모이드 ReLU 함께 그리기

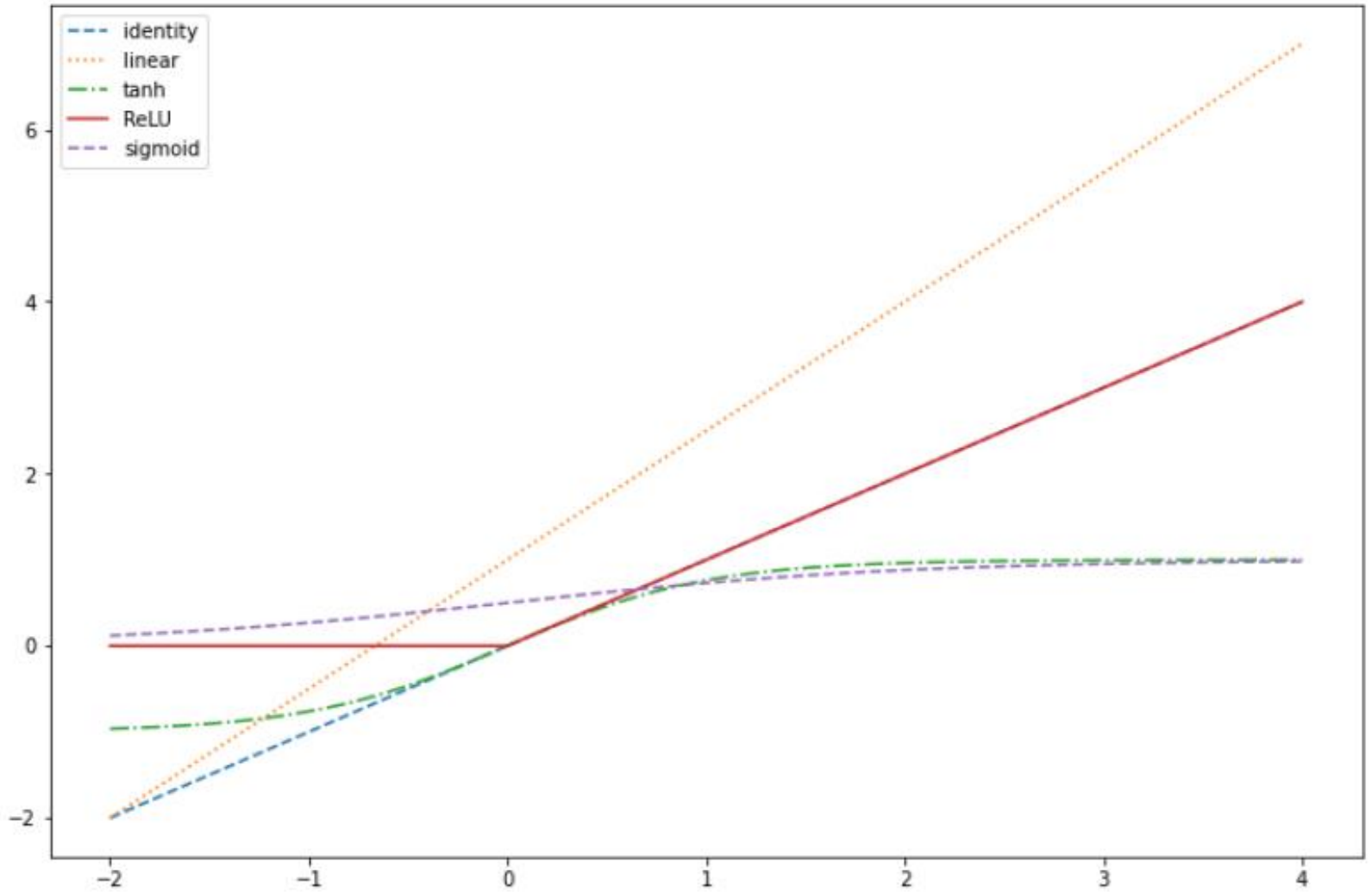



```
import numpy as np
import matplotlib.pylab as plt

# ReLU(Rectified Linear Unit
# (정류된 선형 유닛) 함수
def relu_func(x):
    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=(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')
```

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')
```

활성화 함수 결과



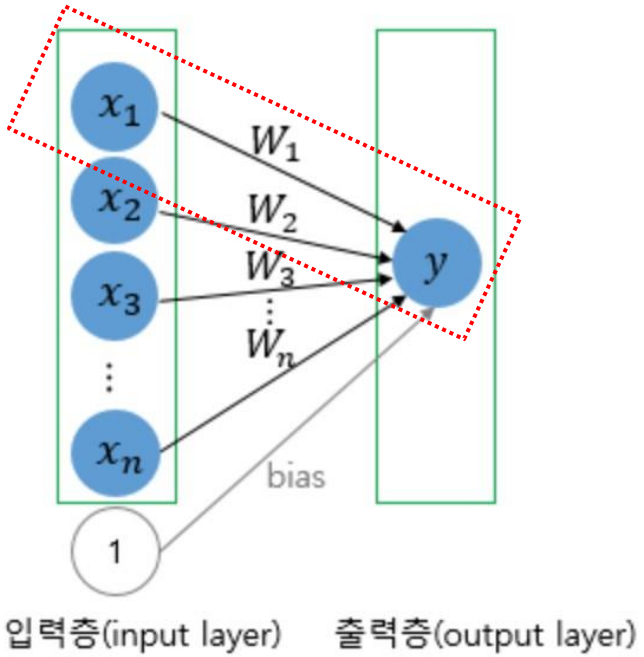
선형 회귀
y = 2x 예측

선형 회귀 문제

- **y = 2x 에 해당하는 값을 예측**
  - 훈련(학습) 데이터
    - **x_train = [1, 2, 3, 4]**
      y_train = [2, 4, 6, 8]
  - 테스트 데이터
    - **x_test = [1.2, 2.3, 3.4, 4.5]**
      y_test = [2.4, 4.6, 6.8, 9.0]
  - 예측, 다음 x에 대해 예측되는 y를 출력
    - **[3.5, 5, 5.5, 6]**

선형 회귀 케라스 구현(1)

- **하나의 Dense 층**
  - 입력은 1차원, 출력도 1차원
- **활성화 함수 linear**
  - 디폴트 값, 입력 뉴런과 가중치로 계산된 결과값이 그대로 출력으로




```
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)
])
```

선형 회귀 케라스 구현(2)

- **확률적 경사하강법(Stochastic Gradient Descent)**
  - optimizer='SGD'
    - **경사하강법의 계산량을 줄이기 위해 확률적 방법으로 경사하강법을 사용**
      - **전체를 계산하지 않고 확률적으로 일부 샘플로 계산**
- **mae**
  - 평균 절대 오차(MAE)
    - **모든 예측과 정답과의 오차 합의 평균**
    - **n = 오차의 갯수**
    - **∑ = 합을 나타내는 기호**

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

- **mse**
  - 오차 평균 제곱합(Mean Squared Error, MSE)
    - **모든 예측과 정답과의 오차 제곱 합의 평균**

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

```
# 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'])
```

선형 회귀 모델 정보

```
# 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(훈련)

- **히스토리 객체**
  - 매 에포크 마다의 훈련 손실값 (loss)
  - 매 에포크 마다의 훈련 정확도 (accuracy)
  - 매 에포크 마다의 검증 손실값 (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
```

선형 회귀 모델 성능 평가 및 예측

- **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]
```

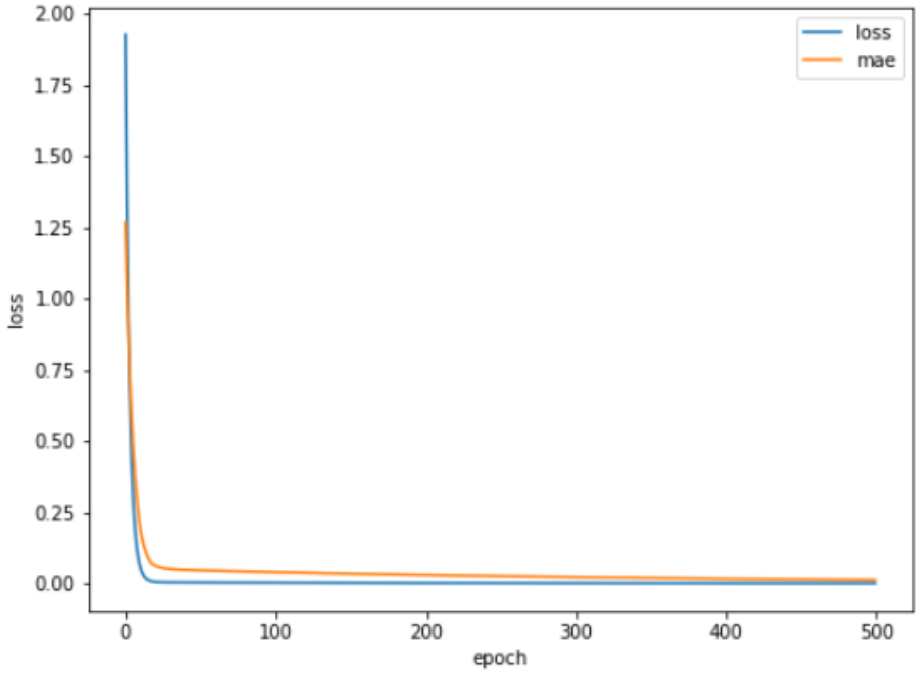
- **예측**

```
# 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())
```

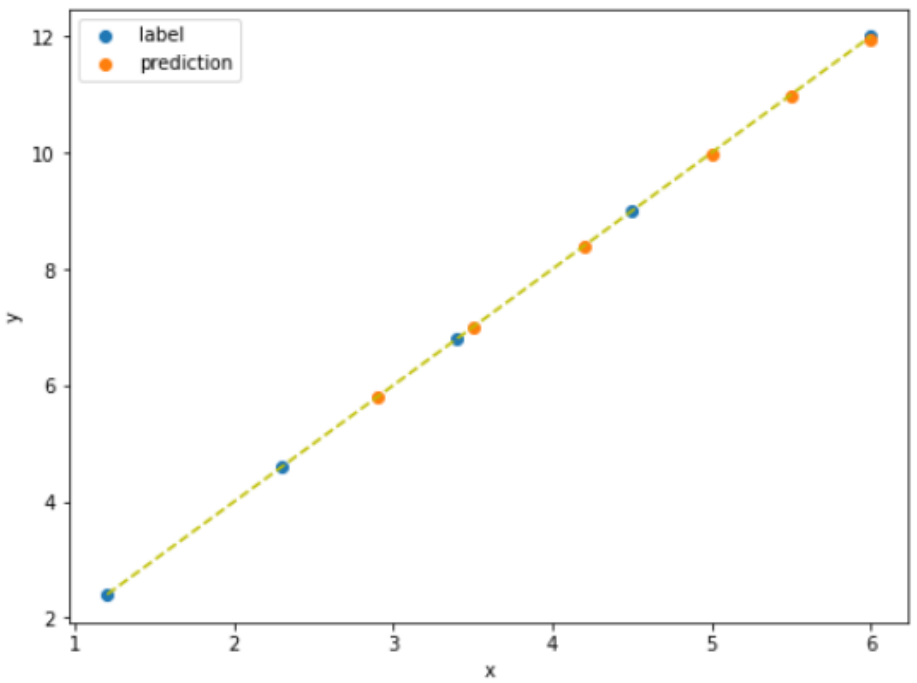
```
[[ 6.9934297]
 [ 9.975829 ]
 [10.969961 ]
 [11.964094 ]]
[ 6.9934297  9.975829  10.969961  11.964094 ]
[ 6.9934297  9.975829  10.969961  11.964094 ]
```

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')
```

전 코드

- 입출력 층만 존재

```
# 버전 1.x만 가능
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

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

# ② 모델 구성(생성)
model = Sequential([
    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, 출력정보를 선택
# Mean Absolute Error, Mean Squared Error
model.compile(optimizer='SGD', loss='mse',
              metrics=['mae', 'mse'])
# 모델을 표시(시각화)
model.summary()

# ④ 생성된 모델로 훈련 데이터 학습
model.fit(x_train, y_train, epochs=1000)

# 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('정확도:', model.evaluate(x_test, y_test))

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

선형 회귀
y = 2x + 1 예측

다음을 예측해 보세요

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

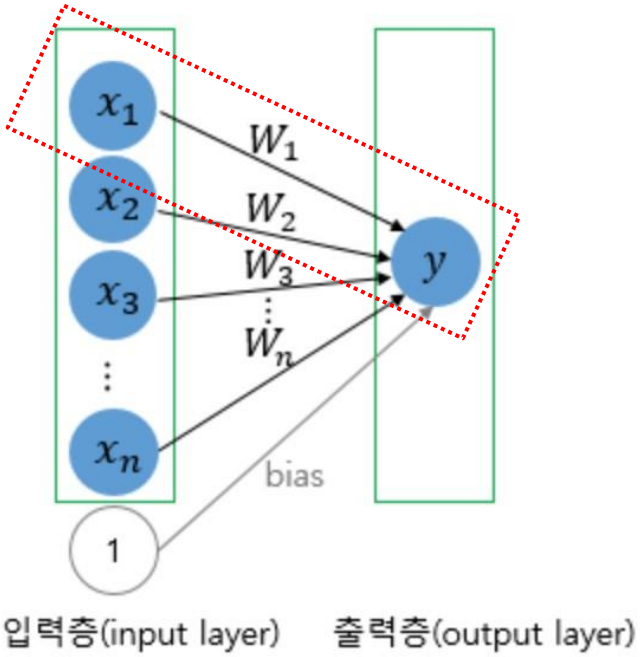
케라스로 예측

- 케라스와 numpy 사용
- 학습에 3개 데이터
  - x = [0, 1, 2, 3, 4]
    - **x[:3]**
  - y = [1, 3, 5, ?, ?]
    - **y[:3]**
- 예측
  - 뒤 2개 데이터 사용
  - x = [0, 1, 2, 3, 4]
    - **x[3:]**
  - y = [1, 3, 5, ?, ?]
    - **y[3:]**

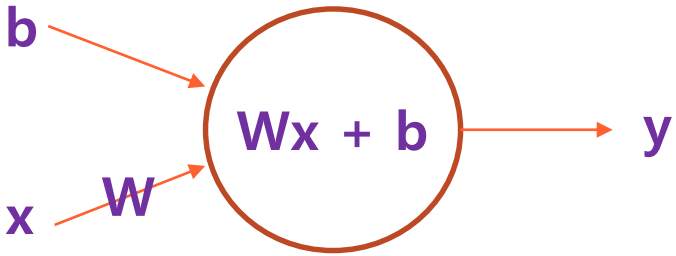
```
import tensorflow as tf
import numpy as np

#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())
```

가장 간단히 입력층과 출력층 구성

- **y[3:]의 2개 값을 맞추는 인공신경망**
  - 먼저 모델에서 W와 b를 구함
  - 완전연결계층
    - **fully connected or dense layer**
      - **입력 벡터에 가중치 벡터를 내적하고 편향값을 빼주는 연산**




```
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(...)

전 소스

```
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())
```

텐서플로로만 구현하는 선형 회귀 예제

변수 Variables

- **딥러닝 학습에서 최적화 과정**
  - 모델의 매개변수(parameters) 즉, 가중치(및 편향)를 조정하는 것
- **변수 tf.Variable**
  - 프로그램에 의해 변화하는 공유된 지속 상태를 표현하는 가장 좋은 방법
    - **하나의 텐서를 표현**
    - **텐서 값은 텐서에 연산을 수행하여 변경 가능**
  - 모델 파라미터를 저장하는데 tf.Variable을 사용
- **변수 생성**
  - 변수를 생성하려면 단순하게 초기값을 설정

```
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)
# 선형회귀 모델(Wx + b)을 위한 tf.Variable을 선언합니다.
W = tf.Variable(tf.random.normal(shape=[1]))
b = tf.Variable(tf.random.normal(shape=[1]))
```

텐서플로 함수로 회귀 구현

- **@tf.function**
  - 데코레이터
  - 다른 일반 함수들처럼 사용
    - **그래프 내에서 컴파일 되었을 때는 더 빠르게 실행하고, GPU나 TPU를 사용해서 작동**
    - **TF 2.0 버전은 즉시 실행 (eager execution)의 편리함과 TF 1.0의 성능을 합침**
    - **파이썬 문법의 일부를 이식 가능하고 높은 성능의 텐서플로 그래프 코드로 변환**
  - 데코레이터가 붙은 함수로 부터 호출된 모든 함수들은 그래프 모드에서 동작
    - **모든 함수에 데코레이터를 붙일 필요는 없음**

```
@tf.function
def linear_model(x):
    return W*x + b

# ③ 학습에 필요한 최적화 방법과 손실 함수 등 지정
# 최적화를 위한 그라디언트 디센트 옵티마이저를 정의합니다.
optimizer = tf.optimizers.SGD(0.01)

# 손실 함수를 정의합니다. MSE 손실함수 \mean{(y' - y)^2}
@tf.function
def mse_loss(y_pred, y):
    return tf.reduce_mean(tf.square(y_pred - y))
```


https://www.tensorflow.org/tutorials/customization/autodiff?hl=ko

한번의 훈련 과정 처리 함수

- **클래스 GradientTape**
  - 자동 미분(주어진 입력 변수에 대한 연산의 그래디언트(gradient)를 계산하는 것) 수행
  - 실행된 모든 연산을 테이프(tape)에 "기록"
    - **테이프에 "기록된" 연산의 그래디언트를 계산**
      - **후진 방식 자동 미분(reverse mode differentiation)을 사용**
  - 입력 W와 b에 대한 loss의 미분 값 자동 계산
    - **tape.gradient(loss, model.trainable_variables)**
  - 예측 값과 손실을 계산하여, 손실에 대한 [w, b]의 미분 값인 gradients를 최적화 과정에 적용
    - **optimizer.apply_gradients(zip(gradients, model.trainable_variables))**

```
# ④ 생성된 모델로 훈련 데이터 학습
# 최적화를 위한 function을 정의합니다.
@tf.function
def train_step(x, y):
    with tf.GradientTape() as tape:
        y_pred = linear_model(x) # 모델에 위한 예측 값 계산
        loss = mse_loss(y_pred, y) # MSE 손실 계산
    gradients = tape.gradient(loss, [W, b]) # 미분 자동계산
    optimizer.apply_gradients(zip(gradients, [W, b])) # 최적화 과정에 적용
```

구현 전 소스

- **Version 2.x**

```
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)
# 선형회귀 모델(Wx + b)을 위한 tf.Variable을 선언합니다.
W = tf.Variable(tf.random.normal(shape=[1]))
b = tf.Variable(tf.random.normal(shape=[1]))

@tf.function
def linear_model(x):
    return W*x + b

# ③ 학습에 필요한 최적화 방법과 손실 함수 등 지정
# 최적화를 위한 그라디언트 디센트 옵티마이저를 정의합니다.
optimizer = tf.optimizers.SGD(0.01)

# 손실 함수를 정의합니다. MSE 손실함수 \mean{(y' - y)^2}
@tf.function
def mse_loss(y_pred, y):
    return tf.reduce_mean(tf.square(y_pred - y))

# ④ 생성된 모델로 훈련 데이터 학습
# 최적화를 위한 function을 정의합니다.
@tf.function
def train_step(x, y):
    with tf.GradientTape() as tape:
        y_pred = linear_model(x) # 모델에 위한 예측 값 계산
        loss = mse_loss(y_pred, y) # MSE 손실 계산
    gradients = tape.gradient(loss, [W, b]) # 미분 자동계산
    optimizer.apply_gradients(zip(gradients, [W, b])) # 최적화 과정에 적용

# 경사하강법을 1000번 수행합니다.
for i in range(1000):
    train_step(x_train, y_train)

# ⑤ 테스트 데이터로 성능 평가
x_test = [3.5, 5, 5.5, 6]

# 테스트 데이터를 이용해 학습된 선형회귀 모델이 데이터의 경향성(y=2x)을 잘 학습했는지 측정합니다.
# 예상되는 참값 : [7, 10, 11, 12]
print(linear_model(x_test).numpy())
```