functional API

Wide and Deep

- sequential NN의 경우 데이터가 모든 층을 거쳐야 하기 때문에 간단판 패턴이기에 간단한 변환으로 풀 수 있는 문제가 연속된 변환으로 왜곡됨
- 이 때 입력의 일부 또는 전체를 출력층에 바로 연결하면 복잡한 패턴과 간단한 패턴 모두 학습할 수 있게 됨

데이터 준비

In [1]:

```
import tensorflow as tf
from tensorflow import keras
```

In [2]:

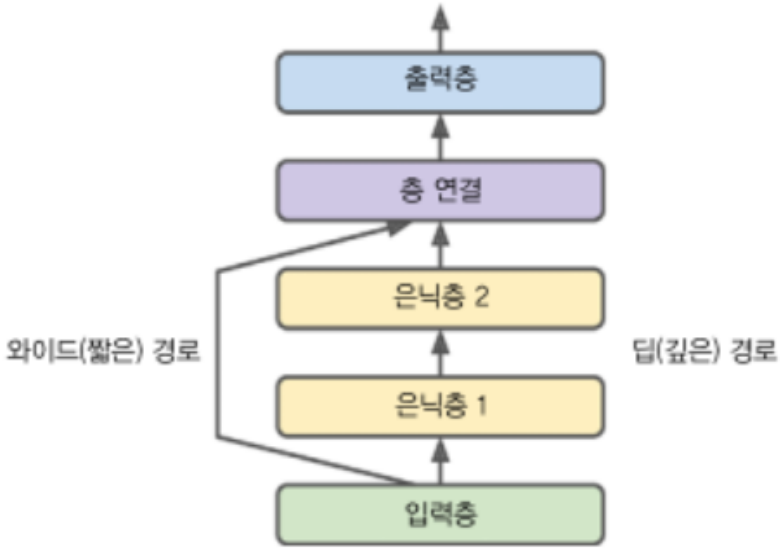
```
import numpy as np
import pandas as pd
```

In [3]:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(housing.data, housing.target)
X_train,X_valid,y_train,y_valid = train_test_split(X_train_full, y_train_full)

# 훈련집합에 맞게 fit 해준 후 fit한 scaler으로 valid와 test set에 대해서 normalization 진행
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

Wide & Deep 구현

모든 feature을 짧은(Wide) 경로로 전달



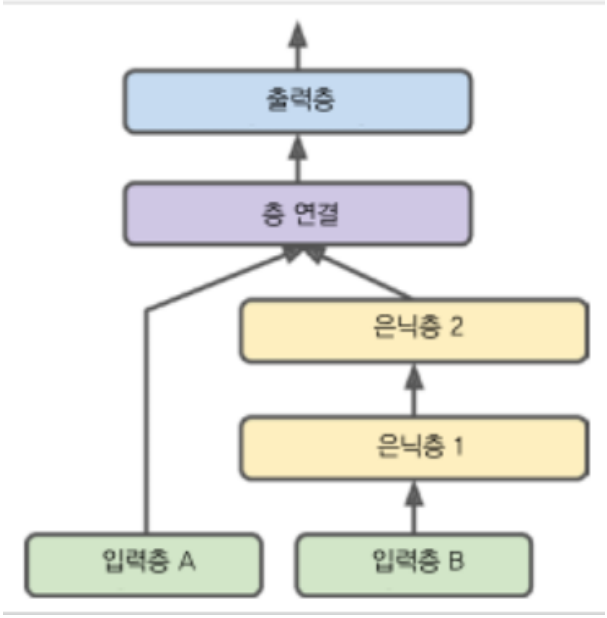
In [6]:

```
# layer 생성
input_ = keras.layers.Input(shape = X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu", name = "hidden1")(input_)
hidden2 = keras.layers.Dense(30, activation="relu", name = "hidden2")(hidden1)
concat = keras.layers.Concatenate()([input_,hidden2])
output = keras.layers.Dense(1)(concat)

# 모델 생성
model = keras.Model(inputs=[input_], outputs = [output])
```

- Input 객체 생성(한 모델은 여러개의 Input을 가질 수 있음)
- 층 뒤에 ()로 어떤 layer와 이어질지 정해줘야 함
  - sequential같은 경우 당연히 바로 앞 layer의 output을 전달받을 것이기 때문에 필요 없었음
- Dense층
- concatenate의 경우 axis로 합칠 차원을 정할 수 있음
  - default : -1(맨 마지막 차원)

일부 feature을 짧은(Wide) 경로로 전달

!



In [24]:

```
input_A = keras.layers.Input(shape=[5,], name = "wide_input")
input_B = keras.layers.Input(shape=[6,], name = "deep_input")
hidden1 = keras.layers.Dense(30, activation = "relu")(input_B)
hidden2 = keras.layers.Dense(30, activation = "relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name = "output")(concat)
model = keras.Model(inputs = [input_A, input_B], outputs = [output])
```

- 모델이 복잡해지면 적어도 가장 중요한 층(헷갈리기 쉬운 층에는) 이름을 붙이는 것이 실수 방지에 좋음

In [14]:

```
model.compile(loss="mse",optimizer = keras.optimizers.SGD(lr=1e-3))

# 모든 train, valid, test에 대해서 input data에 대한 동일한 분할이 이루어져야 함
X_train_A, X_train_B = X_train[:,:5], X_train[:,2:]
X_valid_A, X_valid_B = X_valid[:,:5], X_valid[:,2:]
X_test_A, X_test_B = X_test[:,:5], X_test[:,2:]
```

In [18]:

```
history = model.fit((X_train_A,X_train_B), y_train, epochs=20,
                    validation_data = ((X_valid_A,X_valid_B),y_valid))
```

```
Epoch 1/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4858 - val_loss: 0.
4888
Epoch 2/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4839 - val_loss: 0.
4889
Epoch 3/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4811 - val_loss: 0.
4817
Epoch 4/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4797 - val_loss: 0.
4802
Epoch 5/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4778 - val_loss: 0.
4787
Epoch 6/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4745 - val_loss: 0.
4779
Epoch 7/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4721 - val_loss: 0.
4744
Epoch 8/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4716 - val_loss: 0.
4717
Epoch 9/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4687 - val_loss: 0.
4739
Epoch 10/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4676 - val_loss: 0.
4688
Epoch 11/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4651 - val_loss: 0.
4688
Epoch 12/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4634 - val_loss: 0.
4656
Epoch 13/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4614 - val_loss: 0.
4635
Epoch 14/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4611 - val_loss: 0.
4621
Epoch 15/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4597 - val_loss: 0.
4603
Epoch 16/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4573 - val_loss: 0.
4587
Epoch 17/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4561 - val_loss: 0.
4587
Epoch 18/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4548 - val_loss: 0.
4567
Epoch 19/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4522 - val_loss: 0.
```

```
4552
Epoch 20/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4520 - val_loss: 0.
4535
```

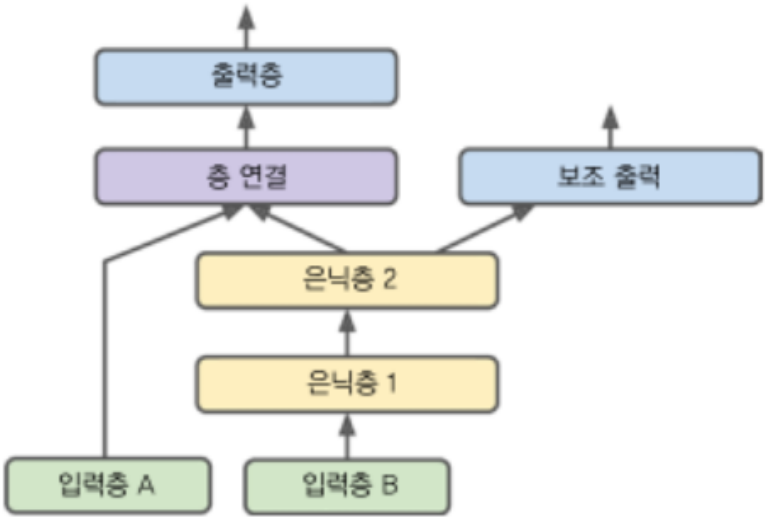
딕셔너리를 통한 인자 전달

In [ ]:

```
history = model.fit({"wide_input":X_train_A, "deep_input":X_train_B},y_train, epochs=20,
                    validation_data = ({"wide_input":X_valid_A, "deep_input":X_valid_B},y_valid))
```

여러 개의 출력

1. 회귀와 분류를 같이 하는 경우
   - ex) 물체를 분류하고 위치를 알아야 하는 경우
2. 동일한 데이터에서 독립적인 여러 작업을 수행하는 경우
   - 작업마다 다른 신경망을 훈련할 수도 있지만, 다중 출력을 가진 단일 신경망을 훈련하는 것이 보통 더 나은 결과
   - ex) 동일한 이미지에서 얼굴 표정을 분류하고 다른 출력에서는 안경을 썼는지를 출력
3. 규제 기법을 사용하는 경우
   - overfitting을 방지하고 일반화 성능을 높이도록 훈련에 제약을 가하는 경우
   - 신경망 구조 안에 보조 출력을 추가함으로써 하위 네트워크가 나머지 네트워크에 의존하지 않고 그 자체로 유용한 것을 학습하는지 확인하는 경우



- deep network가 wide network에 의존하지 않고, 그 자체로 유용한 것을 학습하도록 제약을 가한 경우에 해당됨

*input shape*의 경우 *[]/()*의 형태로 작성하여야 하고, *Dense*의 경우 단순히 출력 뉴런 개수를 정해주는 것이기 때문에 상수로 기입하면 됨

- input shape가 1차원이면 [2,]도 가능하고 [2]도 가능하지만 concat층이 있다면 이 표현을 동일하게 해주어야 함

In [44]:

```
input_A = keras.layers.Input(shape=[5], name = "wide_input")
input_B = keras.layers.Input(shape=[6], name = "depp_input")
hidden1 = keras.layers.Dense(30, activation="relu", name = "hidden1")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu", name = "hidden2")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1,name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)

model = keras.Model(inputs = [input_A,input_B], outputs = [output, aux_output])
```

- output이 2개이기 때문에 두 output에 대한 loss를 리스트로 전달해주어야 함.
  - 하나의 손실함수만 전달하면 모든 출력의 손실 함수가 동일하다고 가정
- output loss에 대한 가중치를 부여할 수 있음

In [45]:

```
# 주 출력에 더많은 가중치를 부여(보조출력은 단순 규제이기 때문)
model.compile(loss=["mse","mse"], loss_weights = [0.9,0.1], optimizer = keras.optimizers.SGD(lr=0.0
```

In [46]:

```
# 동일한 것을 예측하는 것이기 때문에 y데이터에 같은 데이터를 넣음
history = model.fit(
    [X_train_A, X_train_B], [y_train,y_train], epochs = 20,
    validation_data = ([X_valid_A,X_valid_B],[y_valid,y_valid]))
```

```
Epoch 1/20
363/363 [==============================] - 1s 2ms/step - loss: 0.9214 - main_output_
loss: 0.8131 - aux_output_loss: 1.8965 - val_loss: 0.6495 - val_main_output_loss: 0.
5836 - val_aux_output_loss: 1.2431
Epoch 2/20
363/363 [==============================] - 1s 1ms/step - loss: 0.7682 - main_output_
loss: 0.7263 - aux_output_loss: 1.1453 - val_loss: 0.5584 - val_main_output_loss: 0.
5007 - val_aux_output_loss: 1.0774
Epoch 3/20
363/363 [==============================] - 1s 1ms/step - loss: 0.5752 - main_output_
loss: 0.5270 - aux_output_loss: 1.0093 - val_loss: 0.5329 - val_main_output_loss: 0.
4860 - val_aux_output_loss: 0.9543
Epoch 4/20
363/363 [==============================] - 1s 1ms/step - loss: 0.5485 - main_output_
loss: 0.5086 - aux_output_loss: 0.9078 - val_loss: 0.5053 - val_main_output_loss: 0.
4652 - val_aux_output_loss: 0.8665
Epoch 5/20
363/363 [==============================] - 1s 1ms/step - loss: 0.5015 - main_output_
loss: 0.4682 - aux_output_loss: 0.8014 - val_loss: 0.4864 - val_main_output_loss: 0.
4532 - val_aux_output_loss: 0.7857
Epoch 6/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4854 - main_output_
loss: 0.4566 - aux_output_loss: 0.7450 - val_loss: 0.4800 - val_main_output_loss: 0.
4504 - val_aux_output_loss: 0.7463
Epoch 7/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4712 - main_output_
loss: 0.4457 - aux_output_loss: 0.7009 - val_loss: 0.4615 - val_main_output_loss: 0.
4344 - val_aux_output_loss: 0.7055
Epoch 8/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4654 - main_output_
loss: 0.4419 - aux_output_loss: 0.6770 - val_loss: 0.4506 - val_main_output_loss: 0.
4253 - val_aux_output_loss: 0.6784
Epoch 9/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4798 - main_output_
loss: 0.4580 - aux_output_loss: 0.6763 - val_loss: 0.4463 - val_main_output_loss: 0.
4235 - val_aux_output_loss: 0.6515
Epoch 10/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4513 - main_output_
loss: 0.4314 - aux_output_loss: 0.6307 - val_loss: 0.4406 - val_main_output_loss: 0.
4186 - val_aux_output_loss: 0.6379
Epoch 11/20
363/363 [==============================] - 1s 2ms/step - loss: 0.6429 - main_output_
loss: 0.6280 - aux_output_loss: 0.7776 - val_loss: 0.4697 - val_main_output_loss: 0.
4438 - val_aux_output_loss: 0.7024
Epoch 12/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4943 - main_output_
loss: 0.4765 - aux_output_loss: 0.6545 - val_loss: 0.4710 - val_main_output_loss: 0.
4526 - val_aux_output_loss: 0.6369
Epoch 13/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4830 - main_output_
loss: 0.4687 - aux_output_loss: 0.6122 - val_loss: 0.4339 - val_main_output_loss: 0.
4143 - val_aux_output_loss: 0.6102
Epoch 14/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4393 - main_output_
```

```
loss: 0.4228 - aux_output_loss: 0.5881 - val_loss: 0.4190 - val_main_output_loss: 0.
3999 - val_aux_output_loss: 0.5902
Epoch 15/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4233 - main_output_
loss: 0.4065 - aux_output_loss: 0.5744 - val_loss: 0.4146 - val_main_output_loss: 0.
3964 - val_aux_output_loss: 0.5788
Epoch 16/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4140 - main_output_
loss: 0.3976 - aux_output_loss: 0.5621 - val_loss: 0.4066 - val_main_output_loss: 0.
3882 - val_aux_output_loss: 0.5723
Epoch 17/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4155 - main_output_
loss: 0.3999 - aux_output_loss: 0.5563 - val_loss: 0.4042 - val_main_output_loss: 0.
3873 - val_aux_output_loss: 0.5567
Epoch 18/20
363/363 [==============================] - 1s 1ms/step - loss: 0.4076 - main_output_
loss: 0.3923 - aux_output_loss: 0.5454 - val_loss: 0.3975 - val_main_output_loss: 0.
3807 - val_aux_output_loss: 0.5488
Epoch 19/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4118 - main_output_
loss: 0.3967 - aux_output_loss: 0.5472 - val_loss: 0.3919 - val_main_output_loss: 0.
3755 - val_aux_output_loss: 0.5398
Epoch 20/20
363/363 [==============================] - 1s 1ms/step - loss: 0.3958 - main_output_
loss: 0.3814 - aux_output_loss: 0.5248 - val_loss: 0.3972 - val_main_output_loss: 0.
3820 - val_aux_output_loss: 0.5342
```

In [49]:

```
total_loss, main_loss, aux_loss = model.evaluate([X_test_A, X_test_B],[y_test,y_test])
```

```
162/162 [==============================] - 0s 1ms/step - loss: 0.3923 - main_output_
loss: 0.3789 - aux_output_loss: 0.5134
```

In [52]:

```
y_pred_main, y_pred_aux = model.predict([X_test_A[:3], X_test_B[:3]])
```

In [54]:

```
print(y_pred_main)
print(y_pred_aux)
print(y_test[:3])
```

```
[[3.5884962]
 [1.2826921]
 [2.2489727]]
[[3.0051064]
 [0.9495542]
 [2.093636 ]]
[4.    0.847 2.659]
```

서브클래싱 API로 동적 모델 만들기

선언적 프로그래밍

- 시퀀셜 API와 함수형 API는 모두 선언적(declarative)

    - 사용할 layer와 연결 방식을 먼저 정한 후, 모델에 데이터를 주입하여 훈련, 추론을 시작
    - 장점
        - 모델을 저장, 복사, 공유하기 쉬움
        - 모델의 구조를 출력하거나 분석하기 좋음
        - 프레임워크가 크기를 짐작하고 타입을 확인하여 데이터 주입 전 에러를 일찍 발견할 수도 있음
        - 디버깅하기에 좋음

하지만, 어떤 모델은 반복문을 포함하고 다양한 크기를 다루어야 하며 조건문을 가지는 등 여러 가지 동적인 구조를 필요로 함.

-> 명령형 프로그래밍

- 서브클래싱 API
- **init** : layer 구성(Input layer은 안 만들어도 됨)
- call() : 정방향계산(layer을 이어줌) -> 직접계산으로 실행시 call이 실행됨(항상 이걸로 실행하려고 할 시, super시 dynamic=True)
    - 장점
        - input data의 feature수/hidden layer에서의 뉴런수 등을 미리 지정할 필요가 없어서 큰 유연성을 가짐
        - 조건문/반복문 등을 사용하여 개인화된 모델 생성이 가능
    - 단점
        - 모델 구조가 call()메서드 안에 숨겨져 있기 때문에 케라스가 쉽게 이를 분석하기 힘듦. 따라서 모델을 저장하거나 복사할 수 없음
        - summary()메서드를 호출하면 층의 목록만 나열되고 층 간의 연결 정보를 얻을 수 없음
        - 타입과 크기를 미리 확인할 수 없어 실수가 발생하기 쉬움

직접계산

- 기존 tensorflow 1.x 에서는 모델 그래프를 모두 선언한 후 계산이 이루어졌음.
- tensorflow2에서는 "직접계산"을 제공해줌. 즉, 모델 그래프를 선언하지 않아도 연산을 직접 실행하는 명령형 프로그래밍 환경

In [96]:

```
class WideAndDeepModel(keras.Model): # keras.Model class를 상속받음

    #kwargs : "hi"="hihi","hoho"="keke" 이런것들을 인자에 입력하면 하나의 딕셔너리로 만들어줌
    def __init__(self,units = 30, activation = "relu", **kwargs):
        # 부모 class의 내용을 불러오고 표준 매개변수 처리
        super().__init__(**kwargs) # model name 같은 걸 지정해 줄 수 있음
        self.hidden1 = keras.layers.Dense(units, activation=activation, name = "hidden1")
        self.hidden2 = keras.layers.Dense(units, activation=activation)

        # keras에는 output 메서드가 있기 때문에 main함수의 subclass에서 해당 이름을 사용할 수 없음
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    # Input 클래스를 미리 만들지 않음 -> 미리 input shape를 지정해줄 필요가 없음
    def call(self,inputs): # call의 input 매개변수 사용 : fit 할 때 input data넣어주면 알아서 layer
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output
```

In [107]:

```
model = WideAndDeepModel()
model.compile(loss=["mse","mse"],loss_weights = [0.9,0.1], optimizer = keras.optimizers.SGD(lr=1e-3
model.fit([X_train_A,X_train_B],(y_train,y_train),
          epochs = 30, batch_size = 20, validation_data = ((X_valid_A,X_valid_B),
                                                           (y_valid,y_valid)))
```

```
Epoch 1/30
581/581 [==============================] - 1s 2ms/step - loss: 1.9287 - output_1_los
s: 1.7325 - output_2_loss: 3.6953 - val_loss: 0.9581 - val_output_1_loss: 0.7916 - v
al_output_2_loss: 2.4564
Epoch 2/30
581/581 [==============================] - 1s 1ms/step - loss: 0.8786 - output_1_los
s: 0.7511 - output_2_loss: 2.0259 - val_loss: 0.7718 - val_output_1_loss: 0.6733 - v
al_output_2_loss: 1.6584
Epoch 3/30
581/581 [==============================] - 1s 1ms/step - loss: 0.7469 - output_1_los
s: 0.6527 - output_2_loss: 1.5948 - val_loss: 0.7161 - val_output_1_loss: 0.6328 - v
al_output_2_loss: 1.4660
Epoch 4/30
581/581 [==============================] - 1s 1ms/step - loss: 0.7014 - output_1_los
s: 0.6194 - output_2_loss: 1.4387 - val_loss: 0.6825 - val_output_1_loss: 0.6042 - v
al_output_2_loss: 1.3867
Epoch 5/30
581/581 [==============================] - 1s 2ms/step - loss: 0.6726 - output_1_los
s: 0.5974 - output_2_loss: 1.3490 - val_loss: 0.6586 - val_output_1_loss: 0.5841 - v
al_output_2_loss: 1.3289
Epoch 6/30
581/581 [==============================] - 1s 2ms/step - loss: 0.6491 - output_1_los
s: 0.5789 - output_2_loss: 1.2805 - val_loss: 0.6383 - val_output_1_loss: 0.5673 - v
al_output_2_loss: 1.2770
Epoch 7/30
581/581 [==============================] - 1s 2ms/step - loss: 0.6306 - output_1_los
s: 0.5645 - output_2_loss: 1.2258 - val_loss: 0.6205 - val_output_1_loss: 0.5528 - v
al_output_2_loss: 1.2295
Epoch 8/30
581/581 [==============================] - 1s 2ms/step - loss: 0.6132 - output_1_los
s: 0.5505 - output_2_loss: 1.1775 - val_loss: 0.6077 - val_output_1_loss: 0.5434 - v
al_output_2_loss: 1.1869
Epoch 9/30
581/581 [==============================] - 1s 2ms/step - loss: 0.6017 - output_1_los
s: 0.5424 - output_2_loss: 1.1361 - val_loss: 0.5963 - val_output_1_loss: 0.5349 - v
al_output_2_loss: 1.1489
Epoch 10/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5906 - output_1_los
s: 0.5339 - output_2_loss: 1.1001 - val_loss: 0.5893 - val_output_1_loss: 0.5307 - v
al_output_2_loss: 1.1167
Epoch 11/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5795 - output_1_los
s: 0.5253 - output_2_loss: 1.0667 - val_loss: 0.5766 - val_output_1_loss: 0.5203 - v
al_output_2_loss: 1.0840
Epoch 12/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5716 - output_1_los
s: 0.5199 - output_2_loss: 1.0367 - val_loss: 0.5689 - val_output_1_loss: 0.5150 - v
al_output_2_loss: 1.0541
Epoch 13/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5665 - output_1_los
s: 0.5173 - output_2_loss: 1.0091 - val_loss: 0.5619 - val_output_1_loss: 0.5102 - v
al_output_2_loss: 1.0273
Epoch 14/30
```

```
581/581 [==============================] - 1s 1ms/step - loss: 0.5566 - output_1_los
s: 0.5092 - output_2_loss: 0.9832 - val_loss: 0.5633 - val_output_1_loss: 0.5143 - v
al_output_2_loss: 1.0042
Epoch 15/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5534 - output_1_los
s: 0.5082 - output_2_loss: 0.9600 - val_loss: 0.5484 - val_output_1_loss: 0.5005 - v
al_output_2_loss: 0.9792
Epoch 16/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5439 - output_1_los
s: 0.5000 - output_2_loss: 0.9387 - val_loss: 0.5429 - val_output_1_loss: 0.4967 - v
al_output_2_loss: 0.9583
Epoch 17/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5381 - output_1_los
s: 0.4958 - output_2_loss: 0.9190 - val_loss: 0.5394 - val_output_1_loss: 0.4949 - v
al_output_2_loss: 0.9397
Epoch 18/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5315 - output_1_los
s: 0.4905 - output_2_loss: 0.9006 - val_loss: 0.5341 - val_output_1_loss: 0.4908 - v
al_output_2_loss: 0.9236
Epoch 19/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5283 - output_1_los
s: 0.4886 - output_2_loss: 0.8859 - val_loss: 0.5285 - val_output_1_loss: 0.4865 - v
al_output_2_loss: 0.9063
Epoch 20/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5227 - output_1_los
s: 0.4840 - output_2_loss: 0.8704 - val_loss: 0.5248 - val_output_1_loss: 0.4839 - v
al_output_2_loss: 0.8928
Epoch 21/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5183 - output_1_los
s: 0.4807 - output_2_loss: 0.8569 - val_loss: 0.5211 - val_output_1_loss: 0.4812 - v
al_output_2_loss: 0.8798
Epoch 22/30
581/581 [==============================] - 1s 2ms/step - loss: 0.5145 - output_1_los
s: 0.4779 - output_2_loss: 0.8444 - val_loss: 0.5160 - val_output_1_loss: 0.4771 - v
al_output_2_loss: 0.8659
Epoch 23/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5110 - output_1_los
s: 0.4753 - output_2_loss: 0.8324 - val_loss: 0.5128 - val_output_1_loss: 0.4748 - v
al_output_2_loss: 0.8546
Epoch 24/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5077 - output_1_los
s: 0.4729 - output_2_loss: 0.8213 - val_loss: 0.5093 - val_output_1_loss: 0.4722 - v
al_output_2_loss: 0.8431
Epoch 25/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5054 - output_1_los
s: 0.4715 - output_2_loss: 0.8102 - val_loss: 0.5067 - val_output_1_loss: 0.4705 - v
al_output_2_loss: 0.8324
Epoch 26/30
581/581 [==============================] - 1s 1ms/step - loss: 0.5017 - output_1_los
s: 0.4685 - output_2_loss: 0.7999 - val_loss: 0.5041 - val_output_1_loss: 0.4686 - v
al_output_2_loss: 0.8231
Epoch 27/30
581/581 [==============================] - 1s 1ms/step - loss: 0.4975 - output_1_los
s: 0.4649 - output_2_loss: 0.7907 - val_loss: 0.5035 - val_output_1_loss: 0.4689 - v
al_output_2_loss: 0.8150
Epoch 28/30
581/581 [==============================] - 1s 1ms/step - loss: 0.4950 - output_1_los
s: 0.4631 - output_2_loss: 0.7826 - val_loss: 0.4974 - val_output_1_loss: 0.4632 - v
al_output_2_loss: 0.8054
Epoch 29/30
581/581 [==============================] - 1s 1ms/step - loss: 0.4924 - output_1_los
```

```
s: 0.4611 - output_2_loss: 0.7742 - val_loss: 0.4933 - val_output_1_loss: 0.4596 - v
al_output_2_loss: 0.7964
Epoch 30/30
581/581 [==============================] - 1s 1ms/step - loss: 0.4896 - output_1_los
s: 0.4588 - output_2_loss: 0.7662 - val_loss: 0.4918 - val_output_1_loss: 0.4588 - v
al_output_2_loss: 0.7888
```

Out[107]:

```
<tensorflow.python.keras.callbacks.History at 0x1f85bcccd30>
```

In [109]:

```
loss,main_loss, aux_loss = model.evaluate([X_test_A,X_test_B],[y_test,y_test])
main_predicted, aux_predicted = model.predict([X_test_A[:3],X_test_B[:3]])
```

```
162/162 [==============================] - 0s 1ms/step - loss: 0.4750 - output_1_los
s: 0.4460 - output_2_loss: 0.7360
```

모델 저장과 복원

- 일반적으로 하나의 스크립트에서 모델을 훈련하고 저장한 다음 하나 이상의 스크립트(또는 웹 서비스)에서 모델을 로드하고 예측을 만드는 데 활용

시퀀셜, 함수형 API

- model.save()
  - HDF5 포맷을 사용(디폴트)
    - save()메서드의 save_format 매개변수를 tf로 지정하면 텐서플로의 SavedModel 포맷으로 저장할 수 있음
  - 모든 층의 하이퍼 파라미터, 모델 구조, 파라미터(가중치, 편향), 옵티마이저를 저장

In [161]:

```
model = keras.models.Sequential([
    keras.layers.Input(shape = [8]),
    keras.layers.Dense(30, activation = "relu"),
    keras.layers.Dense(30, activation = "relu"),
    keras.layers.Dense(1)
])
```

In [121]:

```
model.compile(loss = "mse", optimizer = keras.optimizers.SGD(lr=1e-3))
model.fit(X_train, y_train, epochs = 20, validation_data = (X_valid, y_valid))
```

```
Epoch 1/20
363/363 [==============================] - 1s 2ms/step - loss: 1.8545 - val_loss:
0.7743
Epoch 2/20
363/363 [==============================] - 1s 2ms/step - loss: 0.7423 - val_loss:
0.6883
Epoch 3/20
363/363 [==============================] - 1s 2ms/step - loss: 0.6808 - val_loss:
0.6433
Epoch 4/20
363/363 [==============================] - 1s 2ms/step - loss: 0.6371 - val_loss:
0.6069
Epoch 5/20
363/363 [==============================] - 1s 1ms/step - loss: 0.6021 - val_loss:
0.5769
Epoch 6/20
363/363 [==============================] - 1s 1ms/step - loss: 0.5724 - val_loss:
0.5507
Epoch 7/20
```

In [122]:

```
model.save("my_keras_model_sequential.h5")
```

In [123]:

```
model = keras.models.load_model("my_keras_model_sequential.h5")
```

서브클래스 API

- model.save_weights()를 통해 파라미터 저장 및 model.load_weights()를 통해 불러올 수 있음
- 나머지는 모두 따로 저장해주어야 함
- 피클(pickle)모듈을 사용하여 모델 객체를 직렬화할 수 있음

In [111]:

```
model.save_weights("my_keras_model.h5")
```

- 모델 구조 및 하이퍼 파라미터 등을 미리 설정을 완료한 모델을 생성한 뒤, model의 파라미터를 불러옴

In [115]:

```
model.load_weights("my_keras_model.h5")
```

In [113]:

```
loss,main_loss, aux_loss = model.evaluate([X_test_A,X_test_B],[y_test,y_test])
```

```
162/162 [==============================] - 0s 1ms/step - loss: 0.4750 - output_1_los
s: 0.4460 - output_2_loss: 0.7360
```

콜백(callback)

- fit()/predict()/evaluate() 메서드의 callbacks 매개변수를 사용하여 케라스가 훈련의 시작이나 끝에 호출할 객체 리스트를 지정할 수 있음

- 시퀀셜 API / 함수형 API 에서만 사용 가능

체크포인트 콜백(checkpoint callback)

- 훈련 마지막에 모델을 저장하는 것뿐만 아니라 훈련 도중 일정 간격으로 체크포인트로 모델을 저장
  - 디폴트 : 매 에포크의 끝에서 저장
- 디폴트 : 체크포인트 포맷
  - save_format 매개변수를 h5로 지정하거나 파일 경로가 .h5로 끝나는 경우에는 HDF5 포맷으로 저장

In [134]:

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5") # 여기에 모델을 저장하겠다
history = model.fit(X_train, y_train, epochs = 10, callbacks = [checkpoint_cb])
```

```
Epoch 1/10
363/363 [==============================] - 0s 1ms/step - loss: 1.7266
Epoch 2/10
363/363 [==============================] - 0s 1ms/step - loss: 0.7691
Epoch 3/10
363/363 [==============================] - 0s 1ms/step - loss: 0.6862
Epoch 4/10
363/363 [==============================] - 0s 1ms/step - loss: 0.6424
Epoch 5/10
363/363 [==============================] - 0s 1ms/step - loss: 0.6102
Epoch 6/10
363/363 [==============================] - 0s 1ms/step - loss: 0.5843
Epoch 7/10
363/363 [==============================] - 0s 1ms/step - loss: 0.5637
Epoch 8/10
363/363 [==============================] - 0s 1ms/step - loss: 0.5462
Epoch 9/10
363/363 [==============================] - 0s 1ms/step - loss: 0.5326
Epoch 10/10
363/363 [==============================] - 0s 1ms/step - loss: 0.5209
```

- 검증 세트에서 최상의 점수를 낸 모델만을 저장

In [135]:

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only = True)
history = model.fit(X_train, y_train, epochs = 10,
                    validation_data = (X_valid, y_valid),
                    callbacks = [checkpoint_cb])

# model은 검증세트에서 최상의 점수를 낸 모델이 아니기 때문에 최상의 모델로 복원
model = keras.models.load_model("my_keras_model.h5")
```

```
Epoch 1/10
  1/363 [..............................] - ETA: 0s - loss: 0.3908WARNING:tensorflow:
Callbacks method `on_train_batch_end` is slow compared to the batch time (batch tim
e: 0.0000s vs `on_train_batch_end` time: 0.0010s). Check your callbacks.
363/363 [==============================] - 1s 2ms/step - loss: 0.5109 - val_loss: 0.
5118
Epoch 2/10
363/363 [==============================] - 1s 1ms/step - loss: 0.5026 - val_loss: 0.
5030
Epoch 3/10
363/363 [==============================] - 1s 1ms/step - loss: 0.4950 - val_loss: 0.
4959
Epoch 4/10
363/363 [==============================] - 1s 2ms/step - loss: 0.4879 - val_loss: 0.
4905
Epoch 5/10
363/363 [==============================] - 1s 1ms/step - loss: 0.4818 - val_loss: 0.
4842
Epoch 6/10
363/363 [==============================] - 1s 1ms/step - loss: 0.4759 - val_loss: 0.
4784
Epoch 7/10
363/363 [==============================] - 1s 2ms/step - loss: 0.4707 - val_loss: 0.
4721
Epoch 8/10
363/363 [==============================] - 1s 2ms/step - loss: 0.4656 - val_loss: 0.
4679
Epoch 9/10
363/363 [==============================] - 1s 2ms/step - loss: 0.4610 - val_loss: 0.
4630
Epoch 10/10
363/363 [==============================] - 1s 2ms/step - loss: 0.4567 - val_loss: 0.
4585
```

조기종료 콜백(Early stopping callback)

- 일정 에포크(patience 매개변수로 지정) 동안 검증 세트에 대한 점수가 향상되지 않으면 훈련을 멈춤
- 최상의 모델을 복원할 수도 있음(restore_best_weights로 지정)

- (컴퓨터가 문제를 일으키는 경우를 대비해서) 체크포인트 저장 콜백과 (시간과 컴퓨팅 자원을 낭비하지 않기 위해) 진전이 없는 경우 조기종료 콜백을 함께 사용할 수 있음

In [136]:

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience = 10,
                                                  restore_best_weights = True)
history = model.fit(X_train, y_train, epochs = 100,
                    validation_data = (X_valid, y_valid),
                    callbacks = [checkpoint_cb, early_stopping_cb])
```

```
Epoch 1/100
363/363 [==============================] - 1s 2ms/step - loss: 0.4530 - val_loss:
0.4550
Epoch 2/100
363/363 [==============================] - 0s 1ms/step - loss: 0.4489 - val_loss:
0.4520
Epoch 3/100
363/363 [==============================] - 1s 1ms/step - loss: 0.4459 - val_loss:
0.4489
Epoch 4/100
363/363 [==============================] - 0s 1ms/step - loss: 0.4426 - val_loss:
0.4481
Epoch 5/100
363/363 [==============================] - 1s 1ms/step - loss: 0.4400 - val_loss:
0.4442
Epoch 6/100
363/363 [==============================] - 1s 2ms/step - loss: 0.4372 - val_loss:
0.4415
Epoch 7/100
363/363 [==============================] - 1s 1ms/step - loss: 0.4347 - val_loss:
```

사용자 정의 callback

- 훈련 시(train)
  - on_epoch_end(self, epoch, logs)
  - on_train_begin()
  - on_train_end()
  - on_epoch_begin()
  - on_epoch_end()
  - on_batch_begin()
  - on_batch_end()
- 검증 시(evaluate)
  - on_test_begin()
  - on_test_end()
  - on_test_batch_begin()
  - on_test_batch_end()
- 예측 시(predict)
  - on_predict_begin()
  - on_predict_end()
  - on_predict_batch_begin()
  - on_predict_batch_end()

In [ ]:

```
model = keras.models.Sequential([
    keras.layers.Input(shape = [8]),
    keras.layers.Dense(30, activation = "relu"),
    keras.layers.Dense(30, activation = "relu"),
    keras.layers.Dense(1)
])
model.compile(loss = "mse", optimizer = keras.optimizers.SGD(lr=1e-3))
```

In [216]:

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("₩nval/train : {:.2f}".format(logs["val_loss"]/logs["loss"]))
    def on_train_begin(self, epoch):
        print("훈련이당")
    def on_batch_begin(self,epoch,logs):
        pass
    def on_predict_begin(self,logs):
        print("예측이당")
    def on_test_begin(self,logs):
        print("검증이당")
```

In [217]:

```
printcallback = PrintValTrainRatioCallback()
```

In [218]:

```
history = model.fit(X_train, y_train, epochs = 10,
                    validation_data = (X_valid, y_valid),
                    callbacks = [checkpoint_cb, early_stopping_cb, printcallback])
```

```
훈련이당
Epoch 1/10
338/363 [==========================>...] - ETA: 0s - loss: 0.3716검증이당

val/train : 1.02
363/363 [==============================] - 1s 1ms/step - loss: 0.3713 - val_loss: 0.
3783
Epoch 2/10
332/363 [==========================>...] - ETA: 0s - loss: 0.3656검증이당

val/train : 1.02
363/363 [==============================] - 1s 1ms/step - loss: 0.3705 - val_loss: 0.
3780
Epoch 3/10
334/363 [==========================>...] - ETA: 0s - loss: 0.3678검증이당

val/train : 1.02
363/363 [==============================] - 1s 1ms/step - loss: 0.3701 - val_loss: 0.
3779
Epoch 4/10
331/363 [==========================>...] - ETA: 0s - loss: 0.3642검증이당

val/train : 1.02
363/363 [==============================] - 0s 1ms/step - loss: 0.3694 - val_loss: 0.
3780
Epoch 5/10
323/363 [=========================>....] - ETA: 0s - loss: 0.3753검증이당

val/train : 1.03
363/363 [==============================] - 1s 1ms/step - loss: 0.3688 - val_loss: 0.
3786
Epoch 6/10
341/363 [===========================>..] - ETA: 0s - loss: 0.3699검증이당

val/train : 1.02
363/363 [==============================] - 1s 2ms/step - loss: 0.3687 - val_loss: 0.
3758
Epoch 7/10
331/363 [==========================>...] - ETA: 0s - loss: 0.3687검증이당

val/train : 1.02
363/363 [==============================] - 0s 1ms/step - loss: 0.3684 - val_loss: 0.
3762
Epoch 8/10
323/363 [=========================>....] - ETA: 0s - loss: 0.3716검증이당

val/train : 1.02
363/363 [==============================] - 1s 1ms/step - loss: 0.3676 - val_loss: 0.
3764
Epoch 9/10
324/363 [=========================>....] - ETA: 0s - loss: 0.3732검증이당

val/train : 1.02
363/363 [==============================] - 1s 1ms/step - loss: 0.3674 - val_loss: 0.
3751
```

```
Epoch 10/10
315/363 [=======================>....] - ETA: 0s - loss: 0.3682검증이당

val/train : 1.02
363/363 [==============================] - 1s 1ms/step - loss: 0.3669 - val_loss: 0.
3744
```

In [215]:

```
model.evaluate(X_test, y_test, callbacks = [printcallback])
```

```
검증이당
162/162 [==============================] - 0s 949us/step - loss: 0.3709
```

Out[215]:

```
0.3709146976470947
```

In [206]:

```
model.predict(X_train[:3], callbacks = [printcallback])
```

```
예측이당
```

Out[206]:

```
array([[3.4922676],
       [2.1402497],
       [5.5691657]], dtype=float32)
```

텐서보드를 이용한 시각화

- 훈련하는 동안의 학습 곡선, 여러 실행 간의 학습 곡선 비교, 계산 그래프 시각화와 훈련 통계 분석 가능
- 모델이 생성한 이미지 확인, 3D에 투영된 복잡한 다차원 데이터 시각화, 클러스터링 기능

- **텐서보드를 이용하는 가장 좋은 방법**(결과를 시각화하고 쉽게 비교 가능)
  - 루트(root) 로그 디렉토리 생성(텐서보드 서버가 가리키는)
  - 프로그램을 실행할 때마다 다른 서브디렉터리에 이진 로그 파일(이벤트 파일) 생성
    - 이진 데이터 레코드 : 서머리

In [227]:

```
import os
root_logdir = os.path.join(os.curdir, "my_logs")

# 루트 로그 디렉터리와 현재 날짜와 시간을 이용한 서브 디렉터리 생성하는 함수
def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id) # 디렉토리 주소를 반환

run_logdir = get_run_logdir() # 새로 그래프를 생성하려고 할 때마다 이 것을 실행해 주어야 함
```

In [228]:

```
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                    validation_data = (X_valid, y_valid),
                    callbacks = [tensorboard_cb])
```

```
Epoch 1/30
  1/363 [..............................] - ETA: 0s - loss: 0.1585WARNING:tensorflow:
From C:₩ProgramData₩Anaconda3₩lib₩site-packages₩tensorflow₩python₩ops₩summary_ops_v
2.py:1277: stop (from tensorflow.python.eager.profiler) is deprecated and will be re
moved after 2020-07-01.
Instructions for updating:
use `tf.profiler.experimental.stop` instead.
  2/363 [..............................] - ETA: 27s - loss: 0.2521WARNING:tensorflo
w:Callbacks method `on_train_batch_end` is slow compared to the batch time (batch ti
me: 0.0070s vs `on_train_batch_end` time: 0.1419s). Check your callbacks.
363/363 [==============================] - 1s 2ms/step - loss: 0.3666 - val_loss: 0.
3740
Epoch 2/30
363/363 [==============================] - 1s 2ms/step - loss: 0.3662 - val_loss: 0.
3738
Epoch 3/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3660 - val_loss: 0.
3733
Epoch 4/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3655 - val_loss: 0.
3726
Epoch 5/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3651 - val_loss: 0.
3726
Epoch 6/30
363/363 [==============================] - 1s 2ms/step - loss: 0.3643 - val_loss: 0.
3716
Epoch 7/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3644 - val_loss: 0.
3723
Epoch 8/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3642 - val_loss: 0.
3711
Epoch 9/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3632 - val_loss: 0.
3708
Epoch 10/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3635 - val_loss: 0.
3714
Epoch 11/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3626 - val_loss: 0.
3706
Epoch 12/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3624 - val_loss: 0.
3694
Epoch 13/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3619 - val_loss: 0.
3697
Epoch 14/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3619 - val_loss: 0.
3699
Epoch 15/30
363/363 [==============================] - 1s 2ms/step - loss: 0.3612 - val_loss: 0.
```

```
3690
Epoch 16/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3611 - val_loss: 0.
3685
Epoch 17/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3604 - val_loss: 0.
3686
Epoch 18/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3606 - val_loss: 0.
3676
Epoch 19/30
363/363 [==============================] - 1s 2ms/step - loss: 0.3598 - val_loss: 0.
3682
Epoch 20/30
363/363 [==============================] - 1s 2ms/step - loss: 0.3591 - val_loss: 0.
3675
Epoch 21/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3592 - val_loss: 0.
3669
Epoch 22/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3585 - val_loss: 0.
3665
Epoch 23/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3588 - val_loss: 0.
3668
Epoch 24/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3584 - val_loss: 0.
3664
Epoch 25/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3578 - val_loss: 0.
3665
Epoch 26/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3576 - val_loss: 0.
3655
Epoch 27/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3572 - val_loss: 0.
3697
Epoch 28/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3573 - val_loss: 0.
3646
Epoch 29/30
363/363 [==============================] - 1s 1ms/step - loss: 0.3570 - val_loss: 0.
3643
Epoch 30/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3561 - val_loss: 0.
3636
```

In [229]:

```
model = keras.models.Sequential([
    keras.layers.Input(shape = [8]),
    keras.layers.Dense(30, activation = "relu"),
    keras.layers.Dense(30, activation = "relu"),
    keras.layers.Dense(1)
])
model.compile(loss = "mse", optimizer = keras.optimizers.SGD(lr=1e-2))
```

In [235]:

```
run_logdir = get_run_logdir()
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data = (X_valid, y_valid),
                    callbacks = [tensorboard_cb])
```

```
Epoch 1/10
  2/363 [..............................] - ETA: 21s - loss: 0.5095WARNING:tensorflo
w:Callbacks method `on_train_batch_begin` is slow compared to the batch time (batch
time: 0.0010s vs `on_train_batch_begin` time: 0.0120s). Check your callbacks.
WARNING:tensorflow:Callbacks method `on_train_batch_end` is slow compared to the bat
ch time (batch time: 0.0010s vs `on_train_batch_end` time: 0.1063s). Check your call
backs.
363/363 [==============================] - 1s 2ms/step - loss: 0.3185 - val_loss: 0.
3335
Epoch 2/10
363/363 [==============================] - 1s 2ms/step - loss: 0.3171 - val_loss: 0.
3150
Epoch 3/10
363/363 [==============================] - 1s 2ms/step - loss: 0.3153 - val_loss: 0.
3113
Epoch 4/10
363/363 [==============================] - 1s 2ms/step - loss: 0.3138 - val_loss: 0.
3992
Epoch 5/10
363/363 [==============================] - 1s 1ms/step - loss: 0.3138 - val_loss: 0.
3145
Epoch 6/10
363/363 [==============================] - 1s 1ms/step - loss: 0.3119 - val_loss: 0.
3147
Epoch 7/10
363/363 [==============================] - 1s 2ms/step - loss: 0.3112 - val_loss: 0.
3193
Epoch 8/10
363/363 [==============================] - 1s 2ms/step - loss: 0.3092 - val_loss: 0.
3116
Epoch 9/10
363/363 [==============================] - 1s 1ms/step - loss: 0.3102 - val_loss: 0.
3288
Epoch 10/10
363/363 [==============================] - 1s 2ms/step - loss: 0.3102 - val_loss: 0.
3086
```

텐서보드 실행

- 주피터 노트북에서 바로 실행
- 아나콘다에서 실행(가상환경에서 실행했다면 해당 환경으로 접속해주어야 함)

In [240]:

```
%load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

In [247]:

```
%tensorboard --logdir=./my_logs --port=6006
```

```
Reusing TensorBoard on port 6006 (pid 25548), started 3 days, 2:48:25 ago. (Use '!ki
ll 25548' to kill it.)
```

텐서보드를 이용한 시각화할 수 있는 스칼라, 히스토그램, 이미지, 오미지, 텍스트 기록

In [246]:

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(test_logdir)
with writer.as_default():
    for step in range(1,1000+1):
        tf.summary.scalar("my_scalar", np.sin(step/10), step=step)
        data = (np.random.randn(100)+2)*step/100 # 몇몇 랜덤 데이터
        tf.summary.histogram("my_hist", data, buckets = 50, step=step)
        images = np.random.rand(2,32,32,3)
        tf.summary.image("my_images", images*step/1000, step=step)
        texts = ["The step is" + str(step), "Its square is "+str(step**2)]
        tf.summary.text("my_text", texts, step=step)
        sine_wave = tf.math.sin(tf.range(12000)/48000*2*np.pi*step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1,-1,1])
        tf.summary.audio("my_audio", audio, sample_rate = 48000, step=step)
```

신경망 하이퍼파라미터 튜닝하기

- 신경망은 많은 하이퍼파라미터를 가지고 있음
- 적절한 하이퍼파라미터를 찾기 위해 sklearn의 GridSearchCV / RandomizedSearchCV를 사용
  - keras모델을 sklearn의 추정기 처럼 보이도록 바꾸어야 함

1. keras model build

In [252]:

```
def build_model(n_hidden = 1, n_neurons = [30], learning_rate = 3e-3, input_shape = [8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape = input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons[layer], activation = "relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer = optimizer)
    return model
```

- keras.wrappers.scikit_learn.KerasRegressor
  - build_model()로 만들어진 케라스 모델을 sklearn model처럼 보이도록 감싸는 간단한 wrapper.

In [253]:

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

In [260]:

```
# sklearn model의 fit할 때의 매개변수라고 생각하면 됨
keras_reg.fit(X_train, y_train, epochs = 30,
              validation_data = (X_valid,y_valid),
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

```
Epoch 1/30
363/363 [==============================] - 1s 3ms/step - loss: 1.3377 - val_loss: 0.
6464
Epoch 2/30
363/363 [==============================] - 1s 2ms/step - loss: 0.6669 - val_loss: 0.
5957
Epoch 3/30
363/363 [==============================] - 1s 2ms/step - loss: 0.5818 - val_loss: 0.
5493
Epoch 4/30
363/363 [==============================] - 1s 2ms/step - loss: 0.5452 - val_loss: 0.
5272
Epoch 5/30
363/363 [==============================] - 1s 2ms/step - loss: 0.5217 - val_loss: 0.
5048
Epoch 6/30
363/363 [==============================] - 1s 2ms/step - loss: 0.5050 - val_loss: 0.
4949
Epoch 7/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4929 - val_loss: 0.
4847
Epoch 8/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4835 - val_loss: 0.
4782
Epoch 9/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4762 - val_loss: 0.
4730
Epoch 10/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4703 - val_loss: 0.
4683
Epoch 11/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4653 - val_loss: 0.
4625
Epoch 12/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4601 - val_loss: 0.
4609
Epoch 13/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4572 - val_loss: 0.
4539
Epoch 14/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4528 - val_loss: 0.
4517
Epoch 15/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4494 - val_loss: 0.
4481
Epoch 16/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4467 - val_loss: 0.
4460
Epoch 17/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4432 - val_loss: 0.
4432
Epoch 18/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4418 - val_loss: 0.
4404
```

```
Epoch 19/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4385 - val_loss: 0.
4395
Epoch 20/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4367 - val_loss: 0.
4376
Epoch 21/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4342 - val_loss: 0.
4346
Epoch 22/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4324 - val_loss: 0.
4326
Epoch 23/30
363/363 [==============================] - 1s 2ms/step - loss: 0.4313 - val_loss: 0.
4303
Epoch 24/30
363/363 [==============================] - 1s 4ms/step - loss: 0.4284 - val_loss: 0.
4338
Epoch 25/30
363/363 [==============================] - 1s 3ms/step - loss: 0.4254 - val_loss: 0.
4271
Epoch 26/30
363/363 [==============================] - 1s 4ms/step - loss: 0.4242 - val_loss: 0.
4253
Epoch 27/30
363/363 [==============================] - 1s 3ms/step - loss: 0.4221 - val_loss: 0.
4249
Epoch 28/30
363/363 [==============================] - 1s 3ms/step - loss: 0.4226 - val_loss: 0.
4226
Epoch 29/30
363/363 [==============================] - 1s 3ms/step - loss: 0.4175 - val_loss: 0.
4224
Epoch 30/30
363/363 [==============================] - 1s 4ms/step - loss: 0.4168 - val_loss: 0.
4171
```

Out[260]:

```
<tensorflow.python.keras.callbacks.History at 0x1f85fe64a00>
```

In [261]:

```
# sklearn model이기 때문에 score 존재
mse_test = keras_reg.score(X_test, y_test) # 음의 mse
print(mse_test) # 작을수록 좋음
```

```
162/162 [==============================] - 0s 1ms/step - loss: 0.4145
-0.41452452540397644
```

In [262]:

```
# 예측
y_pred = keras_reg.predict(X_test[:3])
print(y_pred)
```

```
WARNING:tensorflow:6 out of the last 9 calls to <function Model.make_predict_functio
n.<locals>.predict_function at 0x000001F86928B550> triggered tf.function retracing.
Tracing is expensive and the excessive number of tracings could be due to (1) creati
ng @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3)
passing Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has experimental_relax_shapes=True option
that relaxes argument shapes that can avoid unnecessary retracing. For (3), please r
efer to https://www.tensorflow.org/tutorials/customization/performance#python_or_ten
sor_args (https://www.tensorflow.org/tutorials/customization/performance#python_or_t
ensor_args) and https://www.tensorflow.org/api_docs/python/tf/function (https://www.
tensorflow.org/api_docs/python/tf/function) for  more details.
[3.9419143  0.89612746 2.6900332 ]
```

GridSearchCV/RandomizedSearchCV

- 하이퍼 파라미터의 개수가 많을 경우 그리드 서치는 너무 많은 시간이 걸리기 때문에 랜덤 서치

In [304]:

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "n_hidden" : [0,1,2,3],
    "n_neurons" : [i for i in range(1,100)],
    # 3e-4부터 3e-2까지 연속 랜덤 변수를 만듦
    "learning_rate" : reciprocal(3e-4, 3e-2)
}

# 랜덤으로 하이퍼파라미터를 선택, n_iter : 하이퍼파라미터의 세트 개수(랜덤으로 선택)
rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs, n_iter = 10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=10,
                  validation_data = (X_valid, y_valid),
                   callbacks = [keras.callbacks.EarlyStopping(patience=10)])
```

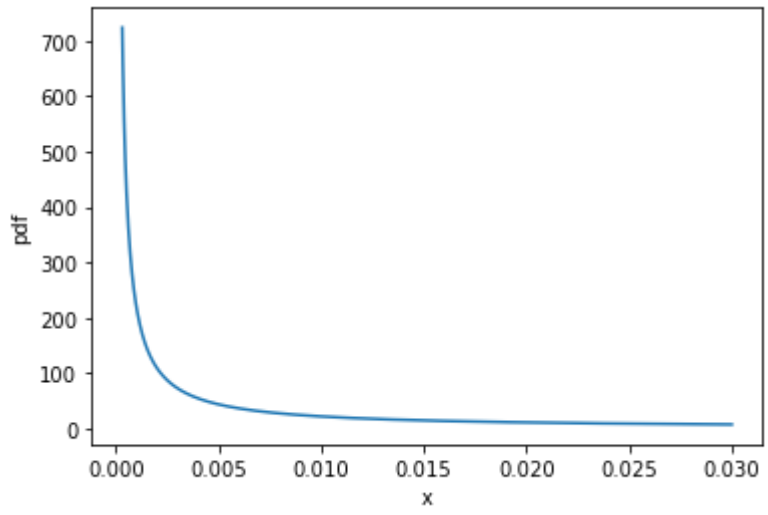
```
C:₩ProgramData₩Anaconda3₩lib₩site-packages₩sklearn₩model_selection₩_validation.p
y:548: FitFailedWarning: Estimator fit failed. The score on this train-test parti
tion for these parameters will be set to nan. Details:
Traceback (most recent call last):
  File "C:₩ProgramData₩Anaconda3₩lib₩site-packages₩sklearn₩model_selection₩_valid
ation.py", line 531, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "C:₩ProgramData₩Anaconda3₩lib₩site-packages₩tensorflow₩python₩keras₩wrappe
rs₩scikit_learn.py", line 157, in fit
    self.model = self.build_fn(**self.filter_sk_params(self.build_fn))
  File "<ipython-input-252-9a2d498ff86e>", line 5, in build_model
    model.add(keras.layers.Dense(n_neurons[layer], activation = "relu"))
TypeError: 'int' object is not subscriptable

  warnings.warn("Estimator fit failed. The score on this train-test"
```

In [303]:

```
import matplotlib.pyplot as plt

x = np.linspace(3e-2,3e-4,1000)
y = 1/(x*np.log(3e-2/3e-4))

plt.plot(x,y)
plt.xlabel("x")
plt.ylabel("pdf")
```

Out[303]:

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



In [264]:

```
np.arange(1,100)
```

Out[264]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
       35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51,
       52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
       69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
       86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])
```

In [305]:

```
print(rnd_search_cv.best_params_)
print(rnd_search_cv.best_score_)
```

```
{'learning_rate': 0.0011678492013761225, 'n_hidden': 0, 'n_neurons': 24}
-0.5917686025301615
```

In [316]:

```
print(rnd_search_cv.best_estimator_)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-316-c821d1467406> in <module>
----> 1 print(rnd_search_cv.best_estimator_)

AttributeError: 'RandomizedSearchCV' object has no attribute 'best_estimator_'
```

하이퍼파라미터 탐색의 다른 방법

- 대규모 데이터셋의 경우 훈련에 시간이 굉장히 오래걸리기 때문에 탐색할 수 있는 하이퍼파라미터 공간에 제약이 생김
- 따라서, 하이퍼파라미터의 값을 크게하여 빠르게 첫번째 랜덤 탐색을 수행하고, 첫 번째 탐색에서 찾은 최상의 하이퍼파라미터 값을 중심으로 더 좁은 범위를 탐색하는 방법으로 계속 진행하면 좋은 하이퍼파라미터 집합을 좁혀나갈 수 있음

사용할 수 있는 몇 개의 파이썬 라이브러리

- Hyperopt
- Hyperas, kopt, Talos
  - 케라스 모델을 위한 라이브러리
- 케라스 튜너
  - 케라스 모델을 위한 라이브러리
- Scikit - Optimize(skopt)
  - 베이즈 최적화 라이브러리
- Spearmint
  - 베이즈 최적화 라이브러리
- Hyperband
- Sklearn-Depp
  - 진화알고리즘 기반의 하이퍼파라미터 최적화 라이브러리
- Arimo
- SigOpt
- CallDesk
- Oscar

진화 알고리즘

- 각 유전체를 통해 유전들을 만들고, 유전들을 선택, 교체, 변이시키면서 좋은 해를 찾아나가는 방법
- 변이의 방법 덕분에 local minimum에 빠질 위험성을 줄여줌(특히 복잡한 과제에서)

주요 하이퍼파라미터

1. 은닉층의 개수

- 이론적 측면
  - 은닉층이 하나이더라도 뉴런 개수가 충분하면 아주 복잡한 함수도 모델링할 수 있음
- 파라미터 효율성 측면
  - 깊은 층을 사용하고 얕은 신경망보다 적은 수의 뉴런을 사용하는 것이 좋음
  - Why?
  - 계층 구조는 아래쪽 은닉층은 저수준의 구조(방향/모양/선)를 모델링하고 중간 은닉층은 저수준의 구조를 연결해 중간 수준의 구조를 모델링(사각형/원), 그리고 가장 위쪽 은닉층과 출력층은 중간 수준의 구조를 연결해 고수준의 구조를 모델링(얼굴)
  - **이러한 과정을 통해 좋은 솔루션으로 빨리 수렴하게 도와줌**
  - **새로운 데이터에 일반화되는 능력도 향상시켜줌 : 전이학습을 가능하게 함**
  - 얼굴을 인식하는 모델의 훈련을 완료한 후, 헤어스타일을 인식하는 훈련을 만들 때, 새로운 신경망의 처음 몇개의 층의 가중치와 편향을 난수로 초기화하는 대신 첫번째 신경망의 층에 있는 가중치와 편향값으로 초기화할 수 있음
  - 이러한 전이학습은 복잡한 문제(대규모 데이터셋을 필요로 함)에서 훈련 속도와 필요한 데이터의 개수를 줄여줄 수 있음

2. 은닉층의 뉴런 개수

- 일반적으로 각 층의 뉴런을 점점 줄여서 깔때기처럼 구성
  - 저수준의 많은 특성이 고수준의 적은 특성으로 합쳐질 수 있기 때문
  - 요즈음의 경우, 모든 은닉층에 같은 크기를 사용해도 동일하거나 더 나은 성능
  - 첫번째 은닉층을 크게 하는 것이 도움이 됨
- 뉴런의 개수를 점차 늘려나가면서 성능을 보는 것보다 필요한 것보다 더 많은 층과 뉴런을 가진 모델을 선택하고, 과대적합되지 않도록 조기종료나 규제 기법을 사용하는 것이 더 간단하고 효과적
  - 일반적으로 층의 뉴런 수보다 층 수를 늘리는 쪽이 이득이 많음

3. 학습률

- 훈련 알고리즘이 발산하는 학습률의 절반 정도가 가장 좋음
- 매우 낮은 학습률(1e-5)에서 시작하여 점진적으로 매우 큰 학습률(10)까지 수백번 반복하여 모델을 훈련하는 것이 좋음
- 반복마다 일정한 값을 학습률에 곱함(ex 10^{-5} 부터 10^{1}까지 500번 반복 : $exp(log(10^{6})/500) \sim 1.012$을 곱함)
  - 학습률이 커짐에 따라 처음에는 손실이 줄어듦. 하지만 잠시 후 학습률이 더 커지면 손실이 커짐.
  - 최적의 학습률은 손실이 다시 상승하는 지점보다 조금 아래에 있을 것임.(일반적으로 상승점보다 약 10배 낮은 지점)

In [2]:

```
import numpy as np
```

In [5]:

```
y
```

Out[5]:

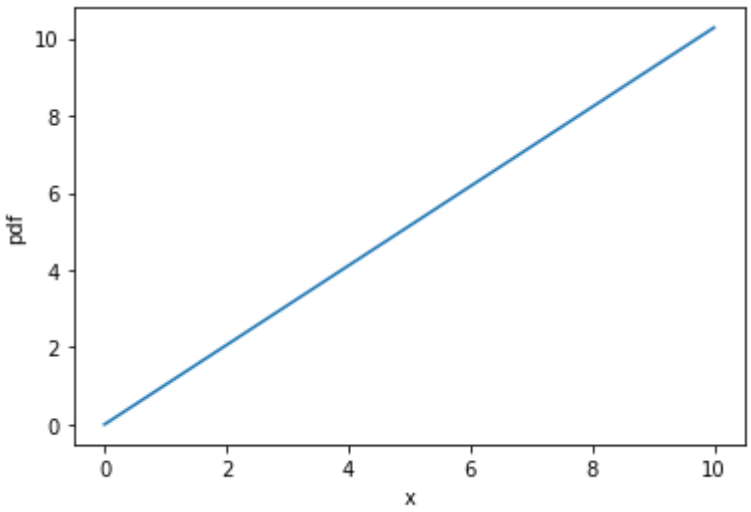
```
array([1.02801630e-05, 1.03839824e-03, 2.06651633e-03, ...,
       1.02781067e+01, 1.02791349e+01, 1.02801630e+01])
```

In [4]:

```
import matplotlib.pyplot as plt

x = np.linspace(1e-5,10,10000)
y = x*np.exp(np.log(10**6)/500)

plt.plot(x,y)
plt.xlabel("x")
plt.ylabel("pdf")
```

Out[4]:

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



4. 옵티마이저

- SGD외에도 다양한 옵티마이저 존재
- ex) Adam, momentum 등

5. 배치 크기

- 모델 성능 측면
  - 일반적으로 2~32정도가 가장 좋음
- 훈련 시간 측면
  - GPU 램에 맞는 가장 큰 배치 크기를 사용하면 GPU와 같은 하드웨어 가속기를 효율적으로 활용할 수 있음. 따라서 훈련 시간을 매우 단축할 수 있음
- 학습률 예열 방법과 같은 다양한 방법을 사용하면 큰 배치크기로도 좋은 성능을 얻을 가능성이 있음. 만약 그렇지 못하다면, 작은 배치크기를 사용하여야 함

6. 활성화함수

- 일반적으로 relu가 가장 좋음

7. 반복횟수

- overfitting등을 막기 위해 조기종료등을 사용하고, 튜닝할 필요 없음