```
# K2020008: 사이킷런(sklearn)이란?
# K2020008: 사이킷런은 파이썬에서 머신러닝 분석을 할 때 유용하게 사용할 수 있는 라이브러리 입니다

# K2020008: s이킷런에 내장되어있는 유방암 데이터 import
from sklearn.datasets import load_breast_cancer

# K2020008: sklearn에 내장된 원본 데이터 불러오기 변수 저장
cancer = load_breast_cancer()
# K2020008: 독립변수 데이터 모음(영향을 주는 변수)
data = cancer.data
# K2020008: 종속변수 데이터 모음(영향을 받는 변수)
labels = cancer.target

# print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
# print('0:2d 1:3d 2:4d' % (x, x*x, x*x*x))

print("독립변수 -> 암에 영향을 주는 변수₩n%s" % data)
print("종속변수 -> 암진단을 받은 경우 = 1, 암진단을 받지 않은 경우 = 0₩n%s" % labels)
print("행,   열    ₩n", data.shape)
```

```
독립변수 -> 암에 영향을 주는 변수
[[1.799e+01 1.038e+01 1.228e+02 ... 2.654e-01 4.601e-01 1.189e-01]
 [2.057e+01 1.777e+01 1.329e+02 ... 1.860e-01 2.750e-01 8.902e-02]
 [1.969e+01 2.125e+01 1.300e+02 ... 2.430e-01 3.613e-01 8.758e-02]
 ...
 [1.660e+01 2.808e+01 1.083e+02 ... 1.418e-01 2.218e-01 7.820e-02]
 [2.060e+01 2.933e+01 1.401e+02 ... 2.650e-01 4.087e-01 1.240e-01]
 [7.760e+00 2.454e+01 4.792e+01 ... 0.000e+00 2.871e-01 7.039e-02]]
종속변수 -> 암진단을 받은 경우 = 1, 암진단을 받지 않은 경우 = 0
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1 1 0 1 0 0
 1 0 1 0 0 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 1
 1 1 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1
 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1 0 0 1 1 1 1 0 1 1 0 0 0 1 0
 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 1 0 1 0 0 0 0 1 1 0 0 1 1
 1 0 1 1 1 1 1 0 0 1 1 0 1 1 0 0 1 0 1 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1
 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 0 0 0 1 1
 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0
 0 1 0 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1
 1 0 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 0 1 1 1 1 1 0 1 1
 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1
 1 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 0 1 0 1 0 1 1 1 1 1 0 1 1 0 1 0 1 0 0
 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 0 0 0 0 0 0 1]
행,   열
 (569, 30)
```

```
# K2020008: Split data
# K2020008: train, test 데이터 분할
from sklearn.model_selection import train_test_split
# K2020008: [klearn의 train_test_split() 사용법
    # 머신러닝 모델을 학습하고 그 결과를 검증하기 위해서는 원래의 데이터를 Training, Validation, Te
    # 그렇지 않고 Training에 사용한 데이터를 검증용으로 사용하면 시험문제를 알고 있는 상태에서 공부
    # 딥러닝을 제외하고도 다양한 기계학습과 데이터 분석 툴을 제공하는 scikit-learn 패키지 중 model_
```

```
    # (1) Parameter
    # arrays : 분할시킬 데이터를 입력 (Python list, Numpy array, Pandas dataframe 등..)
    # test_size : 테스트 데이터셋의 비율(float)이나 갯수(int) (default = 0.25)
    # train_size : 학습 데이터셋의 비율(float)이나 갯수(int) (default = test_size의 나머지)
    # random_state : 데이터 분할시 셔플이 이루어지는데 이를 위한 시드값 (int나 RandomState로 입력)
    # shuffle : 셔플여부설정 (default = True)
    # stratify : 지정한 Data의 비율을 유지한다. 예를 들어, Label Set인 Y가 25%의 0과 75%의 1로 이루

    # (2) Return
    # X_train, X_test, Y_train, Y_test : arrays에 데이터와 레이블을 둘 다 넣었을 경우의 반환이며, C
    # X_train, X_test : arrays에 레이블 없이 데이터만 넣었을 경우의 반환
    # [출처] [Python] sklearn의 train_test_split() 사용법|작성자 Paris Lee

# K2020008: train, test 데이터 분할하기
# K2020008: 오버피팅을 막기위해 데이터를 train, test로 분할 합시다
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.1)
print("데이터 갯수 : {0:d}, 테스트 데이터 갯수 : {1:d}, 데이터 유형 : {2}".format(len(x_train), len
# K2020008: print("학습데이터 갯수 : {0:d}, 테스트 데이터 갯수 : {1:d}, 데이터 유형 : {2}".format(l
print("학습 데이터[{1}]₩n{0}".format(x_train, len(x_train)))
print("학습 레이블[{1}]₩n{0}".format(y_train, len(y_train)))
print("테스트 데이터[{1}]₩n{0}".format(x_test,len(x_test)))
print("테스트 레이블[{1}]₩n{0}".format(y_test,len(y_test)))
```

```
데이터 갯수 : 512, 테스트 데이터 갯수 : 57, 데이터 유형 : <class 'numpy.ndarray'>
```

```
# Convert to tensor
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
from torch.autograd import Variable

# K2020008 : 자동 계산을 위해서 사용하는 변수는 torch.autograd에 있는 Variable 입니다.

# K2020008 : Variable은 아래와 같은 속성들을 갖고 있습니다.
    # .backward() 가 호출되면 미분이 시작되고 그 정보가 담기게 됩니다.

    # data
    # Tensor 형태의 데이터
    # grad
    # Data가 거쳐온 layer에 대한 미분 값
    # grad_fn
    # 미분 값을 계산한 함수에 대한 정보

    # 출처: https://dororongju.tistory.com/142 [웹 개발 메모장]

# K2020008 : 자동 계산을 위해 변수의 변환
x_train = Variable(torch.from_numpy(x_train).float())
y_train = Variable(torch.from_numpy(y_train).float())

x_test = Variable(torch.from_numpy(x_test).float())
y_test = Variable(torch.from_numpy(y_test).float())

print("="*100)
print("x_train.data:", x_train.data)
print("x_train.grad:", x_train.grad)
print("x_train.grad_fn:", x_train.grad_fn)
print("데이터 유형 :{0}".format(type(x_train)))
print("="*100)
```

```
====================================================================================================
x_train.data: tensor([[1.5040e+01, 1.6740e+01, 9.8730e+01,  ..., 1.0180e-01, 2.1770e-01,
         8.5490e-02],
        [1.3980e+01, 1.9620e+01, 9.1120e+01,  ..., 1.8270e-01, 3.1790e-01,
         1.0550e-01],
        [2.8110e+01, 1.8470e+01, 1.8850e+02,  ..., 1.5950e-01, 1.6480e-01,
         5.5250e-02],
        ...,
        [1.2030e+01, 1.7930e+01, 7.6090e+01,  ..., 2.7960e-02, 2.1710e-01,
         7.0370e-02],
        [1.3050e+01, 1.9310e+01, 8.2610e+01,  ..., 1.1110e-02, 2.4390e-01,
         6.2890e-02],
        [1.1080e+01, 1.4710e+01, 7.0210e+01,  ..., 4.3060e-02, 1.9020e-01,
         7.3130e-02]])
x_train.grad: None
x_train.grad_fn: None
데이터 유형 :<class 'torch.Tensor'>
====================================================================================================
```

```
# K2020008 : Generating dataset
```

```
    # 파이토치에서는 데이터를 좀 더 쉽게 다룰 수 있도록 유용한 도구로서 데이터셋(Dataset)과 데이터로
    # 기본적인 사용 방법은 Dataset을 정의하고, 이를 DataLoader에 전달하는 것입니다.
    # TensorDataset은 기본적으로 텐서를 입력으로 받습니다. 텐서 형태로 데이터를 정의합니다(파라메터

# K2020008 : TensorDataset의 입력으로 사용하고 train_set/test_set에 저장합니다
train_set = TensorDataset(x_train, y_train)
test_set = TensorDataset(x_test, y_test)

# K2020008 : DataLoader
    # 파이토치의 데이터셋을 만들었다면 데이터로더를 사용 가능합니다.
    # 데이터로더는 기본적으로 2개의 인자를 입력받는다. 하나는 데이터셋, 미니 배치의 크기입니다.
    # 이때 미니 배치의 크기는 8의 배수를 사용합니다. 그리고 추가적으로 많이 사용되는 인자로 shuffle
    # shuffle=True를 선택하면 Epoch마다 데이터셋을 섞어서 데이터가 학습되는 순서를 바꿉니다.


train_loader = DataLoader(train_set, batch_size = 8, shuffle=True)



# K2020008 : 모델과 설계, Construct model
class Model(nn.Module):
  def __init__(self):
    super().__init__()
    # K2020008 :  입력층-은닉층-은닉층-출력층의 5층 구조
    self.layer1 = nn.Linear(30, 128)
    self.layer2 = nn.Linear(128, 64)
    self.layer3 = nn.Linear(64, 32)
    self.layer4 = nn.Linear(32, 16)
    self.layer5 = nn.Linear(16, 1)
    # self.layer6 = nn.Linear(16, 1)
    # K2020008 : 깊은 신경망은 ReLU 적용
    self.act = nn.ReLU()

  def forward(self,x):
    x = self.act(self.layer1(x))
    x = self.act(self.layer2(x))
    x = self.act(self.layer3(x))
    x = self.act(self.layer4(x))
    # x = self.act(self.layer5(x))
    x = self.layer5(x)
    # K2020008 : 로지스틱 시그모이드
    x = torch.sigmoid(x)

    return x

model = Model()
print(model)
```

```
# K2020008 : 옵티마이저 설계, Configure optimizer
# optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
# optimizer = torch.optim.Adam(model.parameters(), lr=0.00001)
optimizer = torch.optim.Adam(model.parameters(), lr=0.00001)
# optimizer = torch.optim.Adam(model.parameters(), lr=0.0001, weight_decay=0.001)



# 00000000000000000000000000000000000000000000000000# Training

epochs = 2001
losses = list()
accuracies = list()

# K2020008 : 학습 진행 (200회)
for epoch in range(epochs):
  epoch_loss = 0
  epoch_accuracy = 0

  # K2020008 : 8개씩 훈련
  for x, y in train_loader:
    # print(len(x))
    # print(len(y))
    optimizer.zero_grad()

    # K2020008 : 학습 모델 적용 -> H(x) 계산
    output = model(x)

    # K2020008 : cost 계산
    loss = F.binary_cross_entropy(output, y)

    # K2020008 : cost로 H(x) 개선
    # K2020008 : loss를 x로 미분
    # K2020008 : 경사하강법(Gradient descent 구현)
    # optimizer.zero_grad()
    loss.backward()
    optimizer.step()


    # K2020008 : 데이터의 정규화 (0, 1)
    # K2020008 : output>=0.5 -> 1, output < 0.5 -> 0

    output[output>=0.5] = 1
    output[output<0.5] = 0

    # K2020008 : 예측값이(output.data.T) 같으면 True
    accuracy = sum(sum(y.data.numpy() == output.data.T.numpy()))

    # K2020008 : Cost 계산 / accuracy 계산
    epoch_loss += loss.item()
    epoch_accuracy += accuracy


# K2020008 : 학습이 잘 되는지 확인하기 위한 내용출력
```

```
  # K2020008 : 최종 (Cost 계산 / accuracy 계산)
  # K2020008 : loss은 0에 가깝고, accuracy는 1에 근접해야 좋은 학습 결과임
    epoch_loss /= len(train_loader)
    epoch_accuracy /= len(x_train)
    if epoch % 10 == 0:
      print(str(epoch).zfill(3), "loss :", round(epoch_loss,4),"accuracy :", round(epoch_accuracy,4))

    losses.append(epoch_loss)
    accuracies.append(epoch_accuracy)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:21: UserWarning: Using a target s
000 loss : 0.0183 accuracy : 0.9961
010 loss : 0.0189 accuracy : 0.9961
020 loss : 0.0176 accuracy : 0.9961
030 loss : 0.0173 accuracy : 0.9961
040 loss : 0.0169 accuracy : 0.9941
050 loss : 0.0183 accuracy : 0.9961
060 loss : 0.0168 accuracy : 0.9961
070 loss : 0.0162 accuracy : 0.9961
080 loss : 0.0169 accuracy : 0.9941
090 loss : 0.0159 accuracy : 0.9941
100 loss : 0.0169 accuracy : 0.9941
110 loss : 0.0153 accuracy : 0.9961
120 loss : 0.0153 accuracy : 0.9961
130 loss : 0.0149 accuracy : 0.9961
140 loss : 0.0147 accuracy : 0.9941
150 loss : 0.015 accuracy : 0.9961
160 loss : 0.0146 accuracy : 0.9941
170 loss : 0.0136 accuracy : 0.9961
180 loss : 0.0137 accuracy : 0.9961
190 loss : 0.0136 accuracy : 0.998
200 loss : 0.0134 accuracy : 1.0
210 loss : 0.0137 accuracy : 0.9961
220 loss : 0.0132 accuracy : 0.9961
230 loss : 0.013 accuracy : 0.9961
240 loss : 0.0134 accuracy : 0.9961
250 loss : 0.0131 accuracy : 0.9961
260 loss : 0.0126 accuracy : 0.998
270 loss : 0.0136 accuracy : 0.9961
280 loss : 0.0125 accuracy : 0.9961
290 loss : 0.0116 accuracy : 0.9961
300 loss : 0.013 accuracy : 1.0
310 loss : 0.0114 accuracy : 0.9961
320 loss : 0.0123 accuracy : 0.9961
330 loss : 0.011 accuracy : 0.9961
340 loss : 0.0112 accuracy : 0.998
350 loss : 0.0107 accuracy : 0.9961
360 loss : 0.0106 accuracy : 0.998
370 loss : 0.0104 accuracy : 0.9961
380 loss : 0.0103 accuracy : 0.9961
390 loss : 0.0101 accuracy : 0.9961
400 loss : 0.0099 accuracy : 0.998
410 loss : 0.011 accuracy : 0.998
420 loss : 0.0101 accuracy : 0.998
430 loss : 0.0098 accuracy : 0.9961
440 loss : 0.0094 accuracy : 0.9961
450 loss : 0.0097 accuracy : 0.9961
460 loss : 0.0094 accuracy : 1.0
470 loss : 0.0092 accuracy : 0.9961
480 loss : 0.0094 accuracy : 0.998
490 loss : 0.0091 accuracy : 0.998
500 loss : 0.0097 accuracy : 0.9961
510 loss : 0.0095 accuracy : 1.0
520 loss : 0.0083 accuracy : 1.0
530 loss : 0.0085 accuracy : 1.0
540 loss : 0.0084 accuracy : 0.9961
550 loss : 0.0098 accuracy : 1.0
560 loss : 0.0083 accuracy : 0.9961
570 loss : 0.008 accuracy : 1.0
580 loss : 0.0085 accuracy : 1.0
590 loss : 0.0077 accuracy : 0.998
```

```
600 loss : 0.008 accuracy : 0.998
610 loss : 0.0081 accuracy : 0.9961
620 loss : 0.0083 accuracy : 0.998
630 loss : 0.0076 accuracy : 1.0
640 loss : 0.0081 accuracy : 0.998
650 loss : 0.0071 accuracy : 0.998
660 loss : 0.0086 accuracy : 0.998
670 loss : 0.0072 accuracy : 1.0
680 loss : 0.007 accuracy : 1.0
690 loss : 0.007 accuracy : 0.998
700 loss : 0.0069 accuracy : 1.0
710 loss : 0.0066 accuracy : 1.0
720 loss : 0.0065 accuracy : 1.0
730 loss : 0.0079 accuracy : 0.998
740 loss : 0.0074 accuracy : 1.0
750 loss : 0.0063 accuracy : 1.0
760 loss : 0.0068 accuracy : 1.0
770 loss : 0.0066 accuracy : 1.0
780 loss : 0.0061 accuracy : 1.0
790 loss : 0.0059 accuracy : 1.0
800 loss : 0.0064 accuracy : 0.998
810 loss : 0.0062 accuracy : 0.998
820 loss : 0.0059 accuracy : 1.0
830 loss : 0.0064 accuracy : 1.0
840 loss : 0.0057 accuracy : 1.0
850 loss : 0.0056 accuracy : 1.0
860 loss : 0.0056 accuracy : 1.0
870 loss : 0.0059 accuracy : 1.0
880 loss : 0.0061 accuracy : 1.0
890 loss : 0.0058 accuracy : 1.0
900 loss : 0.0051 accuracy : 1.0
910 loss : 0.0058 accuracy : 0.998
920 loss : 0.005 accuracy : 1.0
930 loss : 0.0051 accuracy : 1.0
940 loss : 0.0054 accuracy : 1.0
950 loss : 0.0049 accuracy : 0.998
960 loss : 0.0049 accuracy : 1.0
970 loss : 0.0058 accuracy : 1.0
980 loss : 0.0049 accuracy : 1.0
990 loss : 0.0049 accuracy : 1.0
1000 loss : 0.0054 accuracy : 0.998
1010 loss : 0.0044 accuracy : 1.0
1020 loss : 0.0047 accuracy : 1.0
1030 loss : 0.0045 accuracy : 1.0
1040 loss : 0.0045 accuracy : 1.0
1050 loss : 0.0049 accuracy : 1.0
1060 loss : 0.0055 accuracy : 1.0
1070 loss : 0.0049 accuracy : 1.0
1080 loss : 0.0049 accuracy : 1.0
1090 loss : 0.0045 accuracy : 1.0
1100 loss : 0.0046 accuracy : 1.0
1110 loss : 0.0043 accuracy : 1.0
1120 loss : 0.0053 accuracy : 1.0
1130 loss : 0.0049 accuracy : 1.0
1140 loss : 0.0046 accuracy : 1.0
1150 loss : 0.0055 accuracy : 1.0
1160 loss : 0.0042 accuracy : 1.0
1170 loss : 0.0041 accuracy : 1.0
1180 loss : 0.0042 accuracy : 1.0
1190 loss : 0.0046 accuracy : 1.0
1200 loss : 0.0039 accuracy : 1.0
1210 loss : 0.004 accuracy : 1.0
```

```
1220 loss : 0.0038 accuracy : 1.0
1230 loss : 0.0042 accuracy : 1.0
1240 loss : 0.0038 accuracy : 1.0
1250 loss : 0.0038 accuracy : 1.0
1260 loss : 0.0036 accuracy : 1.0
1270 loss : 0.0036 accuracy : 1.0
1280 loss : 0.0038 accuracy : 1.0
1290 loss : 0.0038 accuracy : 1.0
1300 loss : 0.0036 accuracy : 1.0
1310 loss : 0.0037 accuracy : 1.0
1320 loss : 0.0034 accuracy : 1.0
1330 loss : 0.0044 accuracy : 1.0
1340 loss : 0.0035 accuracy : 1.0
1350 loss : 0.0038 accuracy : 1.0
1360 loss : 0.0034 accuracy : 1.0
1370 loss : 0.0033 accuracy : 1.0
1380 loss : 0.0033 accuracy : 1.0
1390 loss : 0.0033 accuracy : 1.0
1400 loss : 0.0032 accuracy : 1.0
1410 loss : 0.0034 accuracy : 1.0
1420 loss : 0.0031 accuracy : 1.0
1430 loss : 0.0031 accuracy : 1.0
1440 loss : 0.0032 accuracy : 1.0
1450 loss : 0.003 accuracy : 1.0
1460 loss : 0.0048 accuracy : 0.998
1470 loss : 0.0033 accuracy : 1.0
1480 loss : 0.0032 accuracy : 1.0
1490 loss : 0.003 accuracy : 1.0
1500 loss : 0.0031 accuracy : 1.0
1510 loss : 0.0028 accuracy : 1.0
1520 loss : 0.0027 accuracy : 1.0
1530 loss : 0.0029 accuracy : 1.0
1540 loss : 0.0032 accuracy : 1.0
1550 loss : 0.0028 accuracy : 1.0
1560 loss : 0.0038 accuracy : 1.0
1570 loss : 0.0034 accuracy : 1.0
1580 loss : 0.0027 accuracy : 1.0
1590 loss : 0.0026 accuracy : 1.0
1600 loss : 0.0024 accuracy : 1.0
1610 loss : 0.0027 accuracy : 1.0
1620 loss : 0.0029 accuracy : 1.0
1630 loss : 0.0024 accuracy : 1.0
1640 loss : 0.0025 accuracy : 1.0
1650 loss : 0.0024 accuracy : 1.0
1660 loss : 0.0025 accuracy : 1.0
1670 loss : 0.0025 accuracy : 1.0
1680 loss : 0.0023 accuracy : 1.0
1690 loss : 0.0026 accuracy : 1.0
1700 loss : 0.0024 accuracy : 1.0
1710 loss : 0.0022 accuracy : 1.0
1720 loss : 0.0023 accuracy : 1.0
1730 loss : 0.0023 accuracy : 1.0
1740 loss : 0.0025 accuracy : 1.0
1750 loss : 0.0024 accuracy : 1.0
1760 loss : 0.0021 accuracy : 1.0
1770 loss : 0.0021 accuracy : 1.0
1780 loss : 0.0021 accuracy : 1.0
1790 loss : 0.002 accuracy : 1.0
1800 loss : 0.002 accuracy : 1.0
1810 loss : 0.002 accuracy : 1.0
1820 loss : 0.002 accuracy : 1.0
```

```
1830 loss : 0.002 accuracy : 1.0
1840 loss : 0.0021 accuracy : 1.0
1850 loss : 0.0019 accuracy : 1.0
1860 loss : 0.0018 accuracy : 1.0
1870 loss : 0.0018 accuracy : 1.0
1880 loss : 0.0019 accuracy : 1.0
1890 loss : 0.0018 accuracy : 1.0
1900 loss : 0.0018 accuracy : 1.0
1910 loss : 0.0019 accuracy : 1.0
1920 loss : 0.0026 accuracy : 1.0
1930 loss : 0.0018 accuracy : 1.0
1940 loss : 0.0018 accuracy : 1.0
1950 loss : 0.0017 accuracy : 1.0
1960 loss : 0.0018 accuracy : 1.0
1970 loss : 0.0018 accuracy : 1.0
1980 loss : 0.0017 accuracy : 1.0
1990 loss : 0.0016 accuracy : 1.0
2000 loss : 0.0015 accuracy : 1.0
```

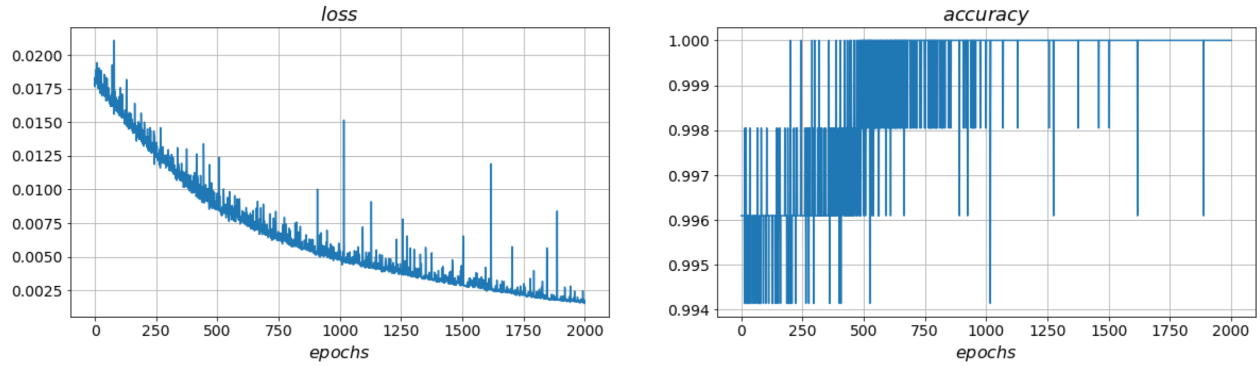
```
# K2020008 : Matplotlib으로 결과 시각화
import matplotlib.pyplot as plt

# K2020008 : Matplotlib는 파이썬에서 데이타를 차트나 플롯(Plot)으로 그려주는 라이브러리
# K2020008 : 최초 창의 크기 -> 가로20 세로 5인치로 설정, wspace의 경우는 subplot간의 간격 0.2
plt.figure(figsize=(20,5))
plt.subplots_adjust(wspace=0.2)

# K2020008 : plt.subplot(nrow,ncol,pos) _> 여러개의 그래프를 그리고 싶을때
# K2020008 : 손실률 그래프 추이
# K2020008 : 타이틀,라벨 달기 및 폰트 크기 설정
plt.subplot(1,2,1)
plt.title("$loss$",fontsize = 18)
plt.plot(losses)
plt.grid()
plt.xlabel("$epochs$", fontsize = 16)
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)

# K2020008 : 정확도 그래프 추이
# K2020008 : 타이틀,라벨 달기 및 폰트 크기 설정
plt.subplot(1,2,2)
plt.title("$accuracy$", fontsize = 18)
plt.plot(accuracies)
plt.grid()
plt.xlabel("$epochs$", fontsize = 16)
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)

# K2020008 : 그래프 출력
plt.show()
```




```
# K2020008 : x_test를 입력 했을때 output 결과 정확도를 확인 해 본다

output = model(x_test)
output[output>=0.5] = 1
output[output<0.5] = 0

accuracy = sum(sum(y_test.data.numpy() == output.data.T.numpy())) /len(y_test)

print("test_set accuracy :", round(accuracy,4))
```

```
test_set accuracy : 0.9825
```

[학습 성능을 향상시킬 수 있는 방법을 2가지 개선]

> 딥러닝 학습 향상을 위한 고려 사항 (http://www.gisdeveloper.co.kr/?p=8443)

- 다양한 경사하강법(Gradient Descent Variants) 최소의 손실값 찾기 위해 손실함수의 미분으로는 방식에 대한 선택에 대한 것입니다.

  > 1. SGD 방식에서 Adam 방식으로 변경
  > 2. lr=0.001 -> 0.00001과 학습 반복횟수 200 -> 2000으로 증가 시킴

- [비 교]

1. ptimizer = torch.optim.SGD(model.parameters(), lr=0.001), 반복 : 200 회

```
160 loss : 0.195 accuracy : 0.9336
170 loss : 0.1925 accuracy : 0.918
180 loss : 0.1974 accuracy : 0.9102
190 loss : 0.1942 accuracy : 0.9199
```

```
200 loss : 0.2015 accuracy : 0.9277

test_set accuracy : 0.9298
```

2. optimizer = torch.optim.Adam(model.parameters(), lr=0.000001), 반복 : 2000 회(지속적 학습

```
1960 loss : 0.0018 accuracy : 1.0
1970 loss : 0.0018 accuracy : 1.0
1980 loss : 0.0017 accuracy : 1.0
1990 loss : 0.0016 accuracy : 1.0
2000 loss : 0.0015 accuracy : 1.0

test_set accuracy : 0.9825
```