

Lab: Deep Learning Model

Sungkyunkwan University (SKKU)
Chan Lim (임찬), Bonggeon Cha(차봉건)

PPT : bit.ly/SDS_day4_ppt_
실습 코드 : bit.ly/SDS_day4_code_

실습 자료 다운로드 및 Colab 설정

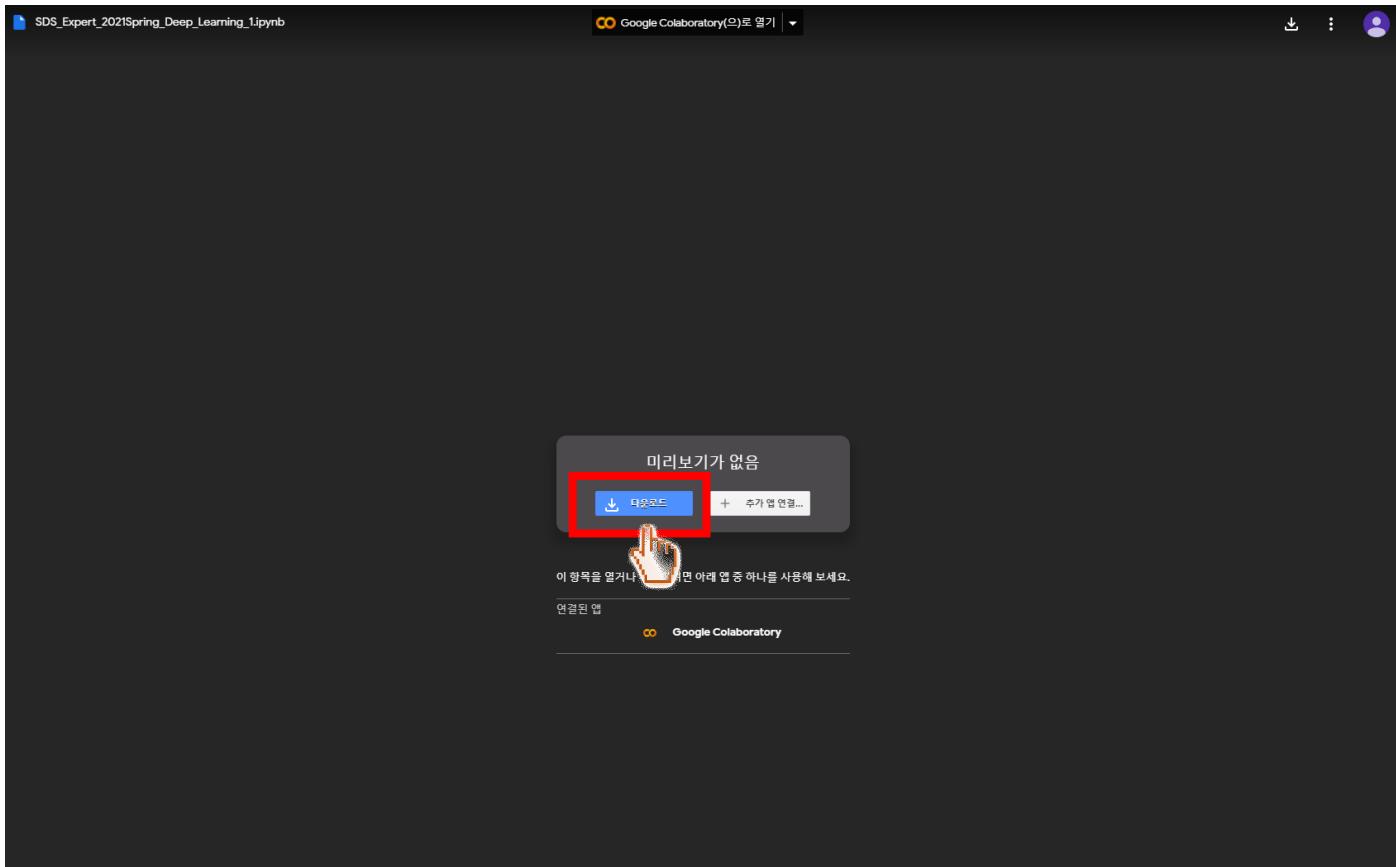
실습 자료 다운로드 및 Colab 설정

- 구글 Colab Notebook으로 실습하기
- Jupyter Notebook으로 실습하기

실습 자료 다운로드 및 기본 설정

- 실습 코드 URL 접속 후, 다운로드.

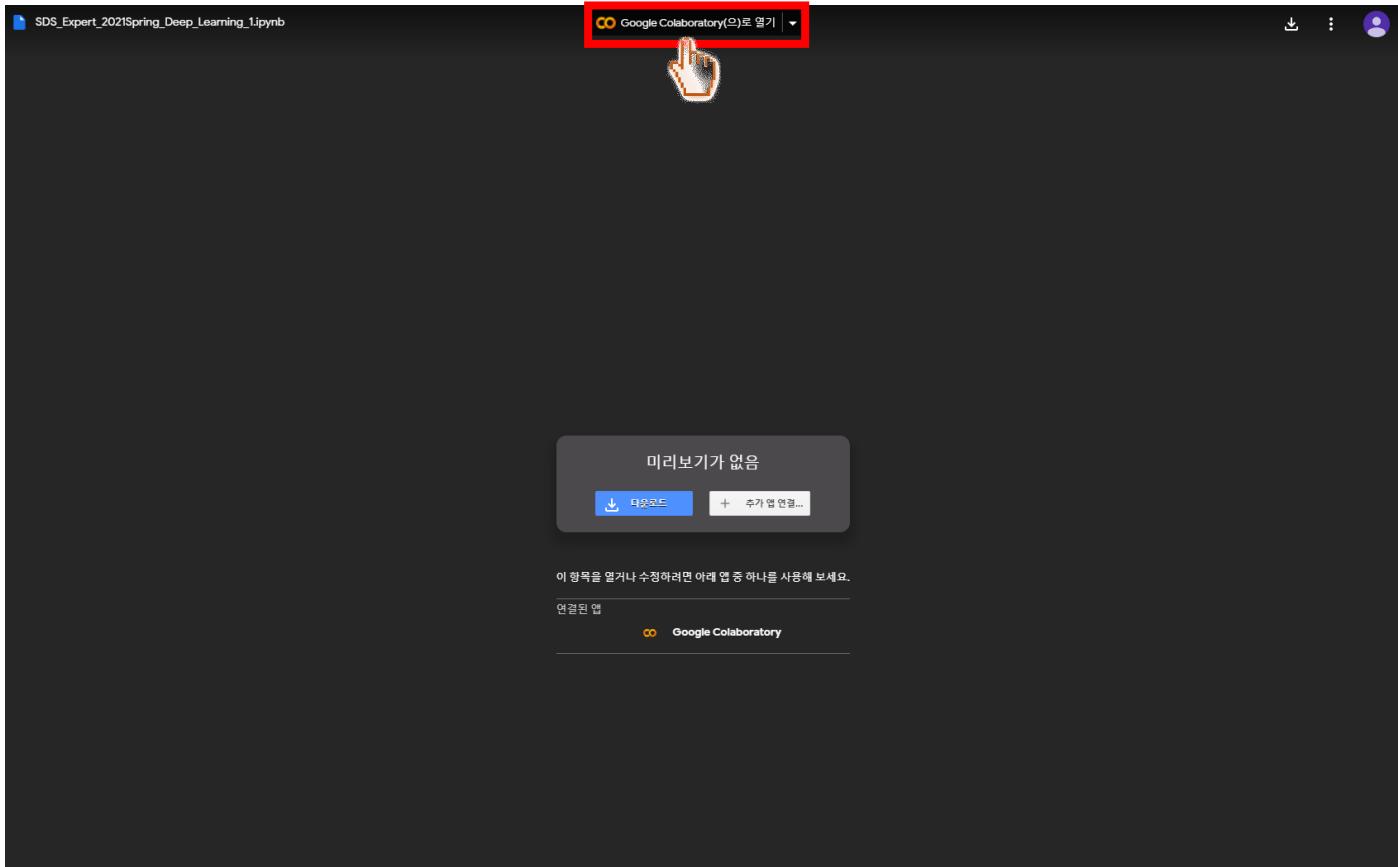
실습 코드 : http://bit.ly/SDS_day4_code



구글 colab으로 실습하기

- 혹은 Google Colaboratory로 열기 클릭. (구글 로그인 되어야 함)

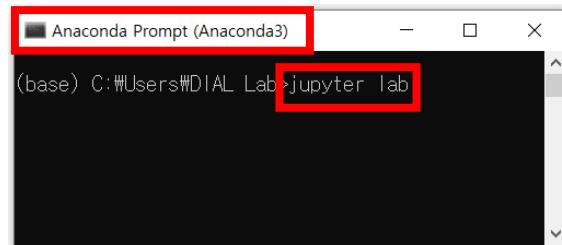
실습 코드 : http://bit.ly/SDS_day4_code



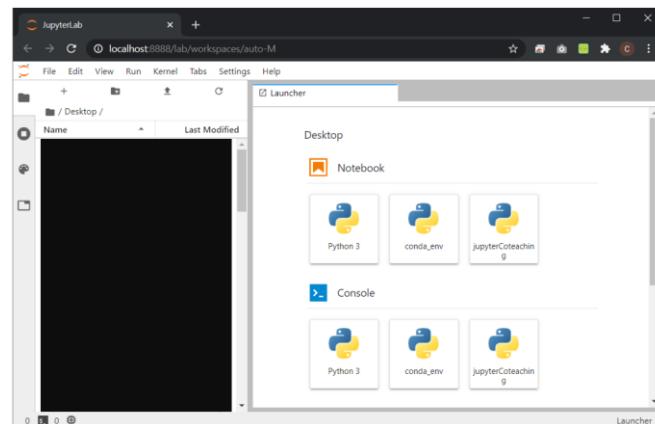
Jupyter Notebook으로 실습하기

➤ Jupyter notebook 열기

- 1) Windows 시작 버튼을 누르고 'Anaconda Prompt (Anaconda 3)' 를 실행합니다.
(윈도우 키를 누르고 바로 'Anaconda Prompt'를 입력하여 검색 OR Anaconda3 프로그램 폴더에서 찾아 실행)
- 2) Anaconda Prompt 창에 'jupyter lab'을 입력하고 Enter 키를 입력합니다.
잠시 기다리면 인터넷 브라우저(Chrome 등) 상에서 Jupyter Lab이 실행됩니다.
- 3) 실행된 Jupyter Lab 에서 실습 코드 (.ipynb)
파일을 찾아 실행합니다.(드래그 가능)



Anaconda Prompt 실행화면



Chrome에서 실행된 Jupyter Lab

Jupyter Notebook으로 실습하기

- Anaconda 사용한다고 가정.
- Jupyter Notebook으로 실습 시 필요한 package 정리
 - ◆ Pytorch
 - Cuda 설정이 안되어 있다면 CPU버전으로 설치
 - conda install pytorch torchvision cpuonly -c pytorch
 - Cuda 10.2 환경이라면
 - conda install pytorch torchvision cudatoolkit=10.2 -c pytorch

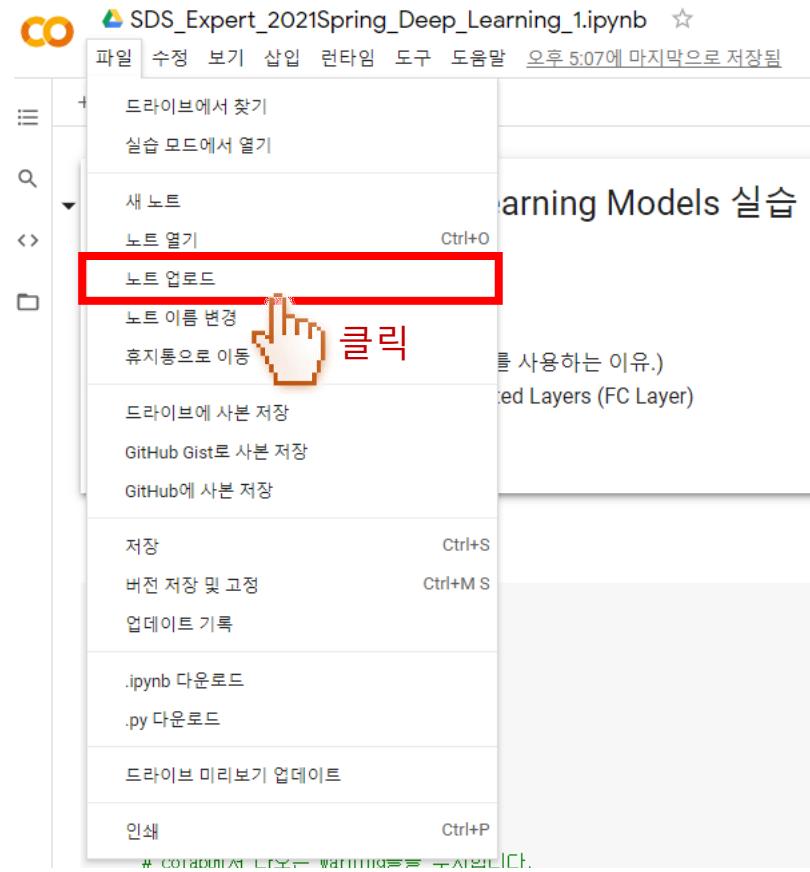
- ◆ Numpy: pip install numpy
- ◆ Sklearn: pip install sklearn
- ◆ Pandas: pip install pandas
- ◆ Matplotlib: pip install matplotlib
- ◆ PIL: pip install pillow

} Anaconda 설치 시 자동으로 설치되는 library들

구글 colab으로 실습하기

- 파일 - 노트 업로드 클릭.

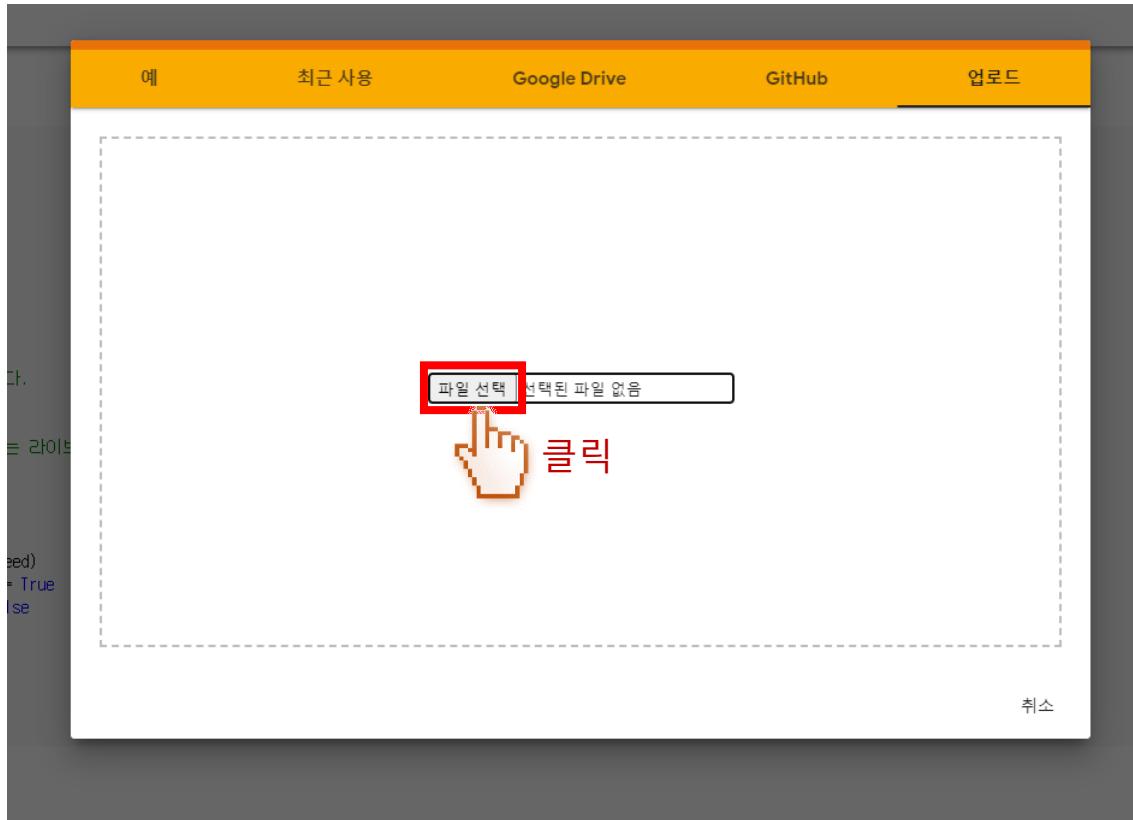
실습 코드 : http://bit.ly/SDS_day4_code



구글 colab으로 실습하기

- 파일 선택 클릭.

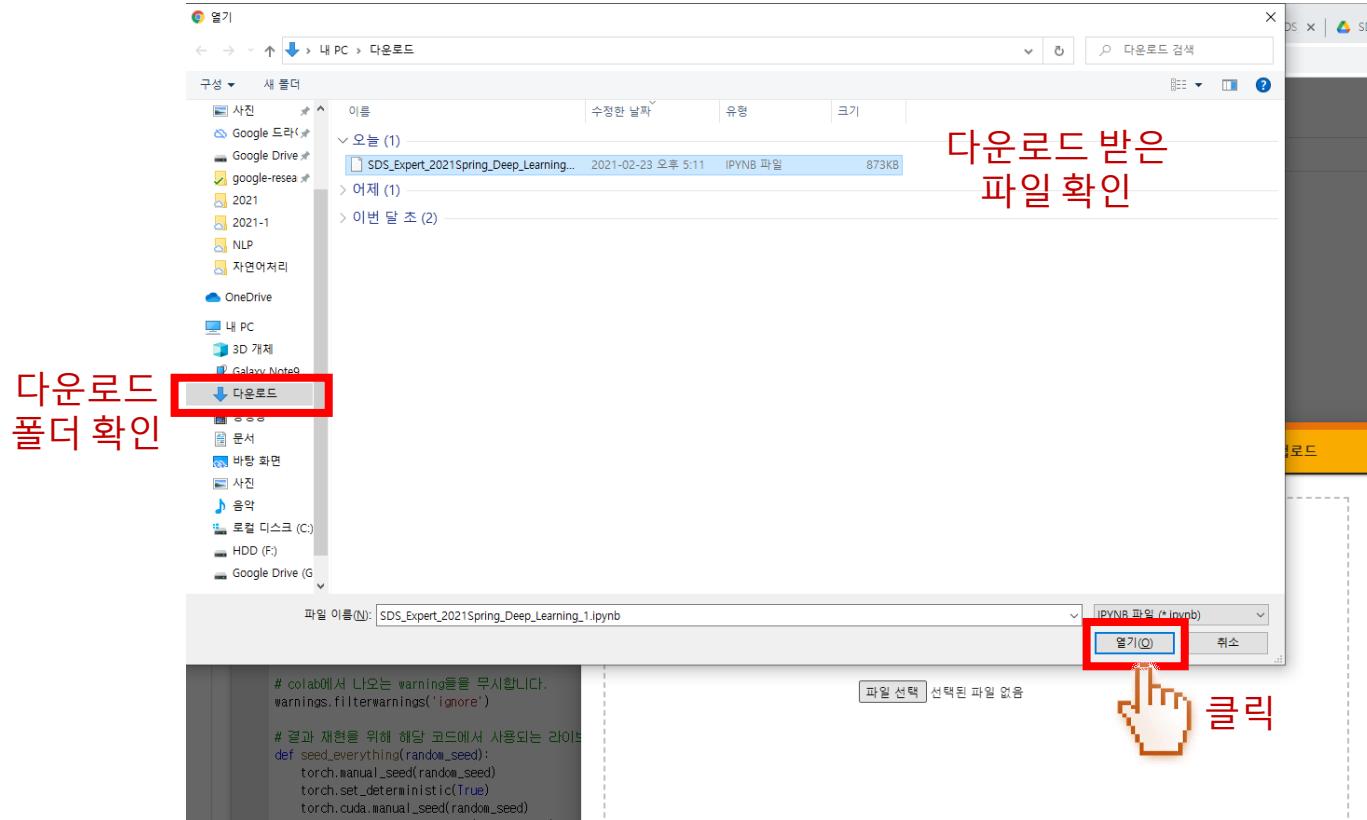
실습 코드 : http://bit.ly/SDS_day4_code



구글 colab으로 실습하기

- ▶ 다운로드 받은 파일을 고른 후 열기 클릭.

실습 코드 : http://bit.ly/SDS_day4_code

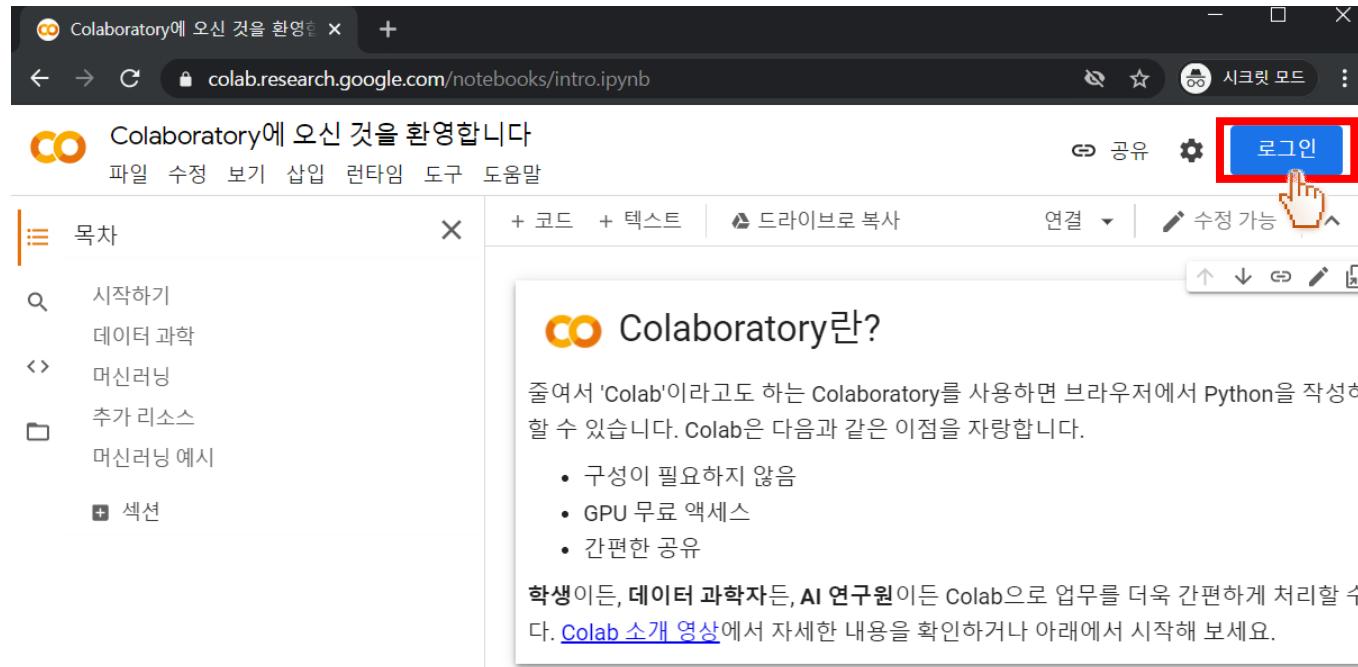


실습 자료 다운로드 및 기본 설정

- Chrome 브라우저를 이용해 Google Colaboratory 접속 후 본인 Google ID로 로그인

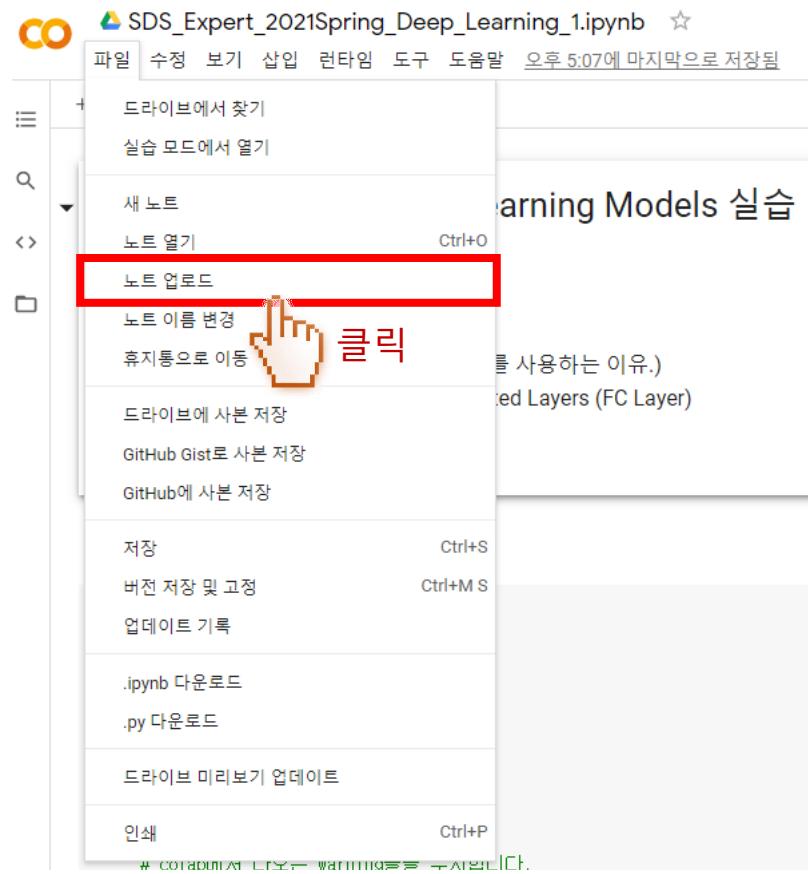
Google Colaboratory :

<https://colab.research.google.com>



실습 자료 다운로드 및 기본 설정

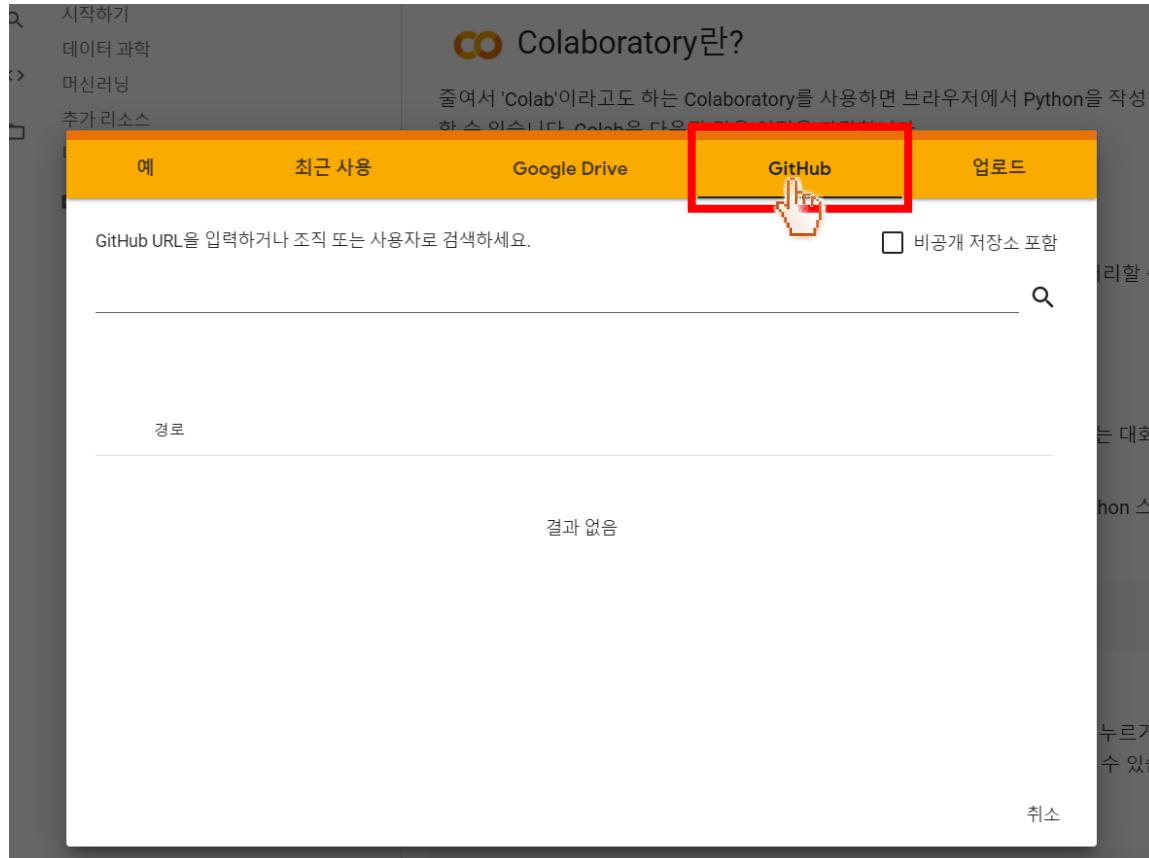
▶ 노트 업로드 클릭.



방법 2.

실습 자료 다운로드 및 기본 설정

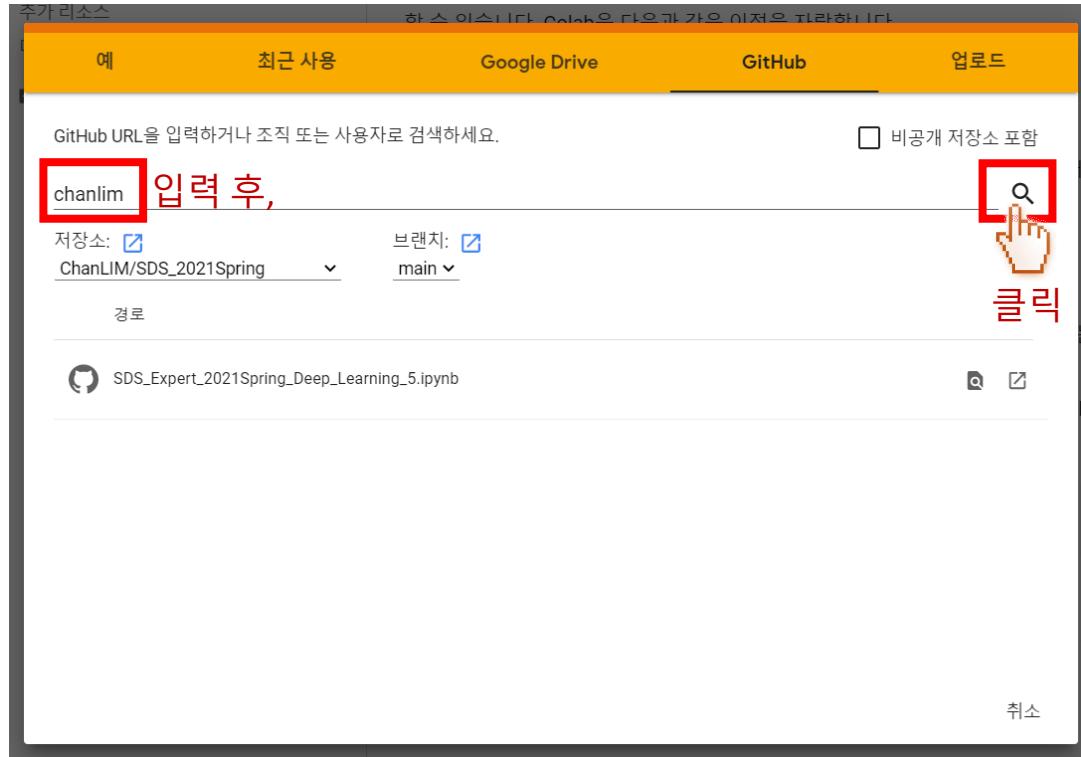
➤ GitHub 클릭.



방법 2.

실습 자료 다운로드 및 기본 설정

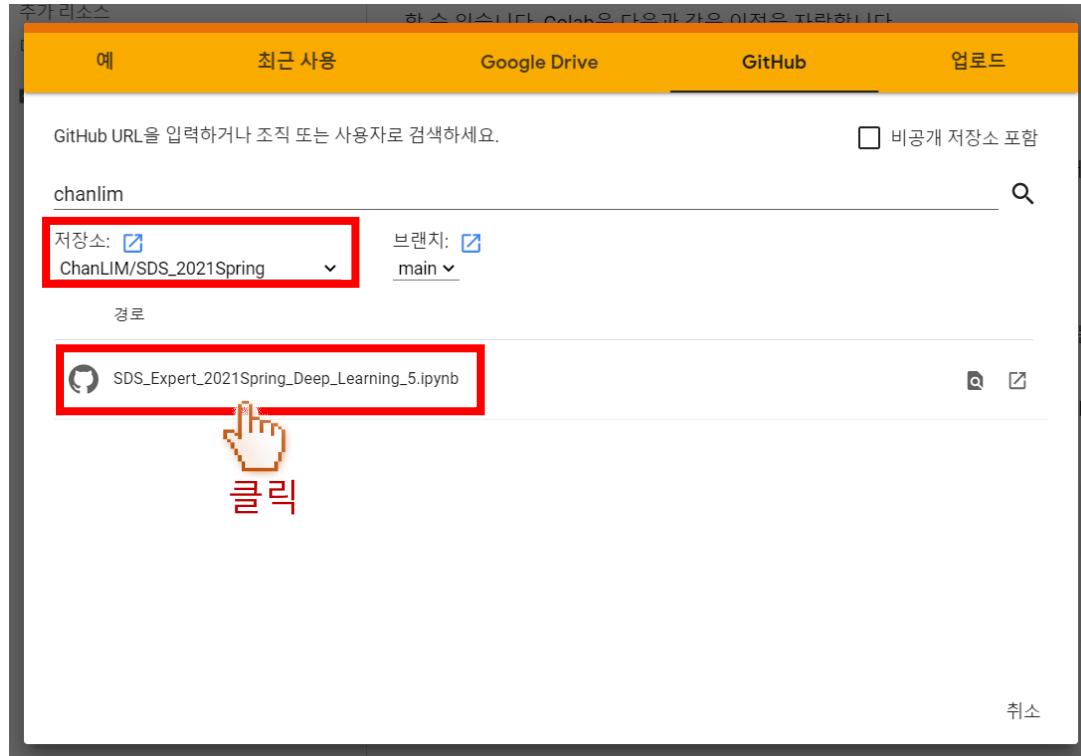
- 검색창에 'chanlim' 입력 후 검색 버튼 클릭.



방법 2.

실습 자료 다운로드 및 기본 설정

- ChanLIM/SDS_2021Spring 저장소에 있는 SDS_Expert_2021Spring_Deep_Learning_4.ipynb 클릭하여 실행



방법 2.

구글 colab으로 실습하기

- Shift + Enter로 코드 실행

The screenshot shows the Google Colab interface with a notebook titled "Untitled0.ipynb". The code cell contains Python code for building a Matrix Factorization (MF) model using TensorFlow. The code includes importing libraries, loading data, setting hyperparameters, defining placeholders, initializing variables, performing embeddings, calculating predictions, and defining a loss function. It also includes training steps and session initialization.

```
[1]: # 필요 라이브러리 및 함수 불러오기
import tensorflow as tf
import numpy as np
from utils.load_data import data_loading
from utils.load_data import get_titles
from utils.evaluation import evaluation

# 데이터 불러오기
train_matrix, test_matrix, num_users, num_items = data_loading()

# 하이퍼 파라미터 설정
num_factor = 30
learning_rate = 0.001
training_epochs = 4 # 총 4번 학습

# tensorflow를 이용하여, MF 모델 만들기

# 플레이스홀더 구성
user_id = tf.placeholder(dtype=tf.int32, shape=[None])
item_id = tf.placeholder(dtype=tf.int32, shape=[None])
Y = tf.placeholder("float", [None])

# U와 V 초기화 설정
U = tf.Variable(tf.random_normal([num_users, num_factor], stddev=0.01))
V = tf.Variable(tf.random_normal([num_items, num_factor], stddev=0.01))

# 사용자, 항목의 latent factor 가져오기
user_latent = tf.nn.embedding_lookup(U, user_id)
item_latent = tf.nn.embedding_lookup(V, item_id)

# 선호도 예측
Y_ = tf.reduce_sum(tf.multiply(user_latent, item_latent), axis=1)

# 예측 값과 실제 값의 차이 설정
loss = tf.reduce_sum(tf.square(Y - Y_))

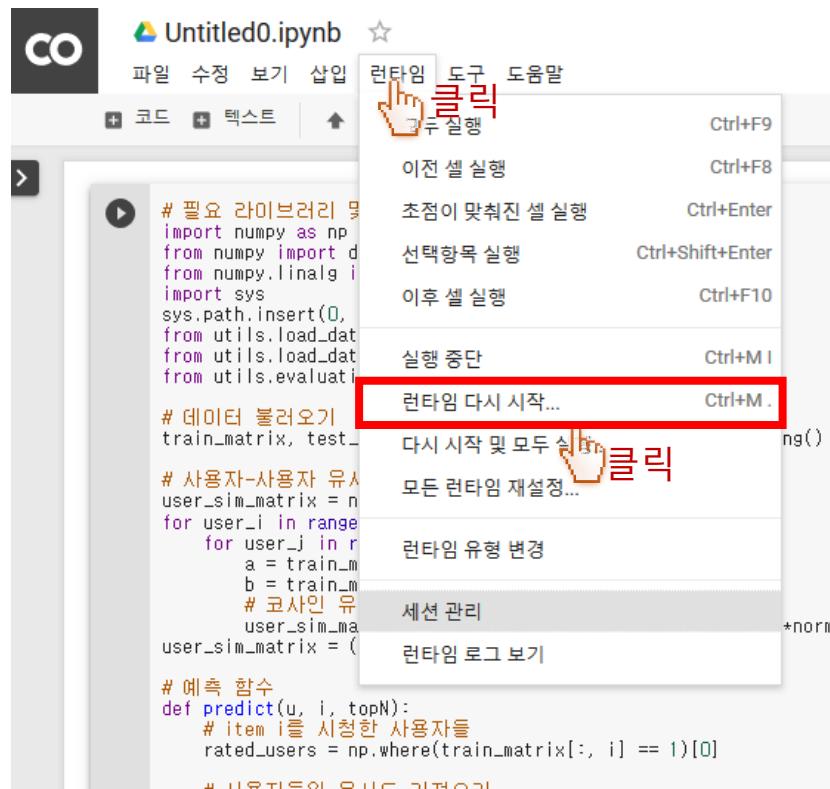
# 학습 방법 설정(gradient descent방법 설정)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)

# 세션 생성 및 파라미터 초기화
sess = tf.Session()
sess.run(tf.global_variables_initializer()) # 파라미터 U, V 초기화
```

구글 colab으로 실습하기

➤ 실행 환경 초기화

- ◆ 런타임 다시 시작



구글 colab으로 실습하기

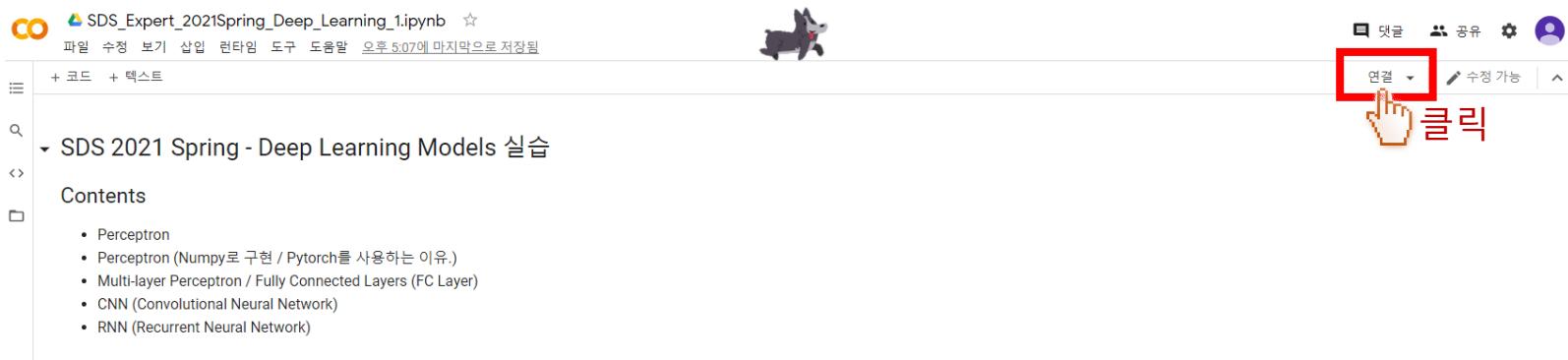
➤ 실행 환경 초기화

- ◆ 에러로 인하여 처음부터 다시 실행하고 싶을 때 사용



구글 colab으로 실습하기

➤ 실행 환경 연결/재연결



Numpy/Pytorch 튜토리얼

Numpy & Pytorch



- ◆ 메모리 효율적, 빠른 연산 속도
- ◆ 다차원 배열 표현



- ◆ 인기 있는 딥러닝 프레임워크
- ◆ 복잡한 Backpropagation 계산을 자동으로 수행
- ◆ GPU 가속 지원
- ◆ 직관적 문법과 쉬운 디버깅으로 연구자들 사이 인기
- ◆ Numpy와 닮은 부분이 많음.

Numpy의 유용함

➤ 두 행렬의 곱

```
py_mat_a = [[1, 2], [3, 4]]  
py_mat_b = [[2, 1], [4, 3]]  
  
result = []
```

Python List로 구현 시

```
for i in range(2):  
    row = []  
    for j in range(2):  
        n = 0  
        for k in range(2):  
            n += py_mat_a[i][k] * py_mat_b[k][j]  
        row.append(n)  
    result.append(row)
```

VS.

Numpy array로 구현 시

```
np_mat_a = np.array(py_mat_a)  
np_mat_b = np.array(py_mat_b)  
  
result = np.matmul(np_mat_a, np_mat_b)
```

Numpy Tutorial

➤ 배열 생성

```
# Python List로부터 배열 생성  
arr_a = np.array([1,2,3,4])
```

특정 값으로 채운 배열

```
arr_zeros = np.zeros((2,2)) # (2,2)크기의 0으로 채워진 배열  
arr_ones = np.ones((1,6)) # (1,6) 크기의 1로 채워진 배열  
arr_full = np.full((2,2), 10) # 10으로 채워진 (2,2)크기의 배열  
arr_eye = np.eye(2) # 2크기의 diagonal matrix (대각행렬)
```

```
array([[0., 0.],  
       [0., 0.]])    array([[1., 1., 1., 1., 1., 1.]])    array([[10, 10],  
                                              [10, 10]])    array([[1., 0.],  
                                              [0., 1.]])
```

np.zeros

np.ones

np.full

np.eye

Numpy Tutorial

➤ 배열 생성

```
bigger = np.arange(10) # [0, 1, 2, ..., 9]
```

```
smaller = np.flip(bigger) # [9, 8, 7, ..., 0]
```

9에서부터 0까지 1씩 작아지는 크기 10의 배열.

0~100 사이의 값으로 채워진 (4,4) 크기의 배열
array([[15., 9., 19., 35.],
 [40., 54., 42., 69.],
 [20., 88., 3., 67.],
 [42., 56., 14., 20.]])

무작위 값으로 채워진 배열

```
rand_between0_1 = np.random.random((2,2)) # 0~1 사이의 값으로 채워  
진 (2,2) 크기의 배열
```

```
rand_between0_100 = np.round(100 * np.random.random((4,4)))
```

0~100 사이의 값으로 채워진 (4,4) 크기의 배열

rand_between0_1 배열에 100을 곱한 후, 1의 자리까지 반올림.

위와 같이 배열 전체에 scalar의 곱도 가능하고 반올림도 가능.

```
print("0~100 사이의 값으로 채워진 (4,4) 크기의 배열")
```

```
rand_between0_100
```

Numpy Tutorial

➤ Numpy array의 모양 및 변형

```
print("bigger 배열 내 값들")
print(bigger)
print("bigger 배열의 모양")
print(bigger.shape)
```

```
print('*'*100) # 구분선
```

```
print("bigger 배열을 (5행, ?)의 크기로 변환
(=배열 길이/5)열의 모양으로 바꿨을 때")
print(bigger.reshape(5, -1))
print(bigger.reshape(5, -1).shape)
```

```
bigger 배열 내 값들
[0 1 2 3 4 5 6 7 8 9]
bigger 배열의 모양
(10,)
```

```
=====
bigger 배열을 5행 ? (=배열 길이/5)열의 모양으로 바꿨을 때
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
(5, 2)
```

Numpy Tutorial

➤ 배열 dimension 추가, 제거

배열 dimension 추가, 제거

```
print("bigger 배열의 모양을 (10, 1)로.")
```

```
bigger_dim = bigger.reshape(10,1)
```

```
print(bigger_dim.shape)
```

Tip : (10,)의 배열 형태로는 실행되던 코드가 input이 (10, 1)의 형태를 가졌을 때에는 실행이 안되는 경우가 있음

print(bigger_dim.squeeze().shape) # squeeze() : shape에서 크기가 1인 차원을 없앰.

```
print(bigger[0]) # 0 출력
```

```
print(bigger_dim.reshape(10, 1)[0]) # [0] 출력
```

```
print(bigger_dim.reshape(10, 1).squeeze()[0]) # 0 출력
```

bigger 배열의 모양을 (10, 1)로.
(10, 1)
(10,)
0
[0]
0

Numpy Tutorial

➤ Numpy array 인덱싱

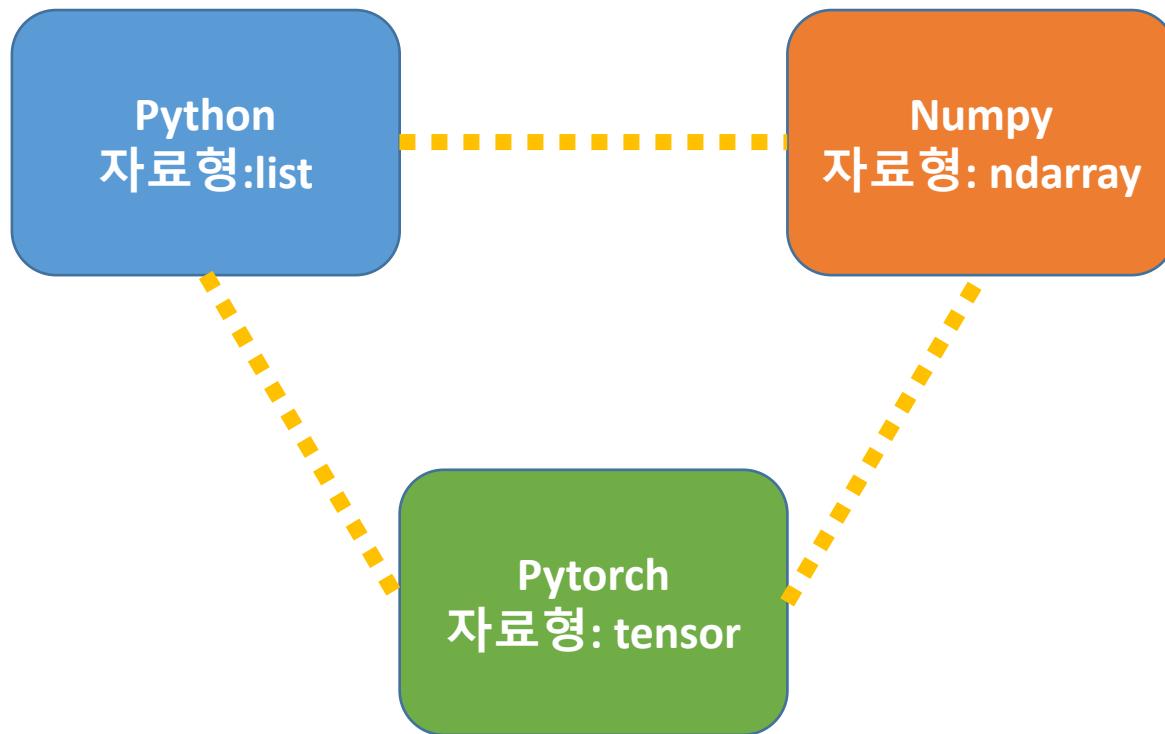
```
a = np.arange(16).reshape(4,4)  
print(a)  
print('*'*100) # 구분선
```

```
[[ 0  1  2  3]  
 [ 4  5  6  7]  
 [ 8  9 10 11]  
 [12 13 14 15]]
```

```
print(a[0,0]) # 0행 0열 값  
print(a[0:2, 0]) # 0열의 0이상 2미만 행들(0행과 1행)의 값  
print(a[:, 0].reshape(-1,1))  
# 0열의 모든 값 / 열의 값을 가져와도 값을 가장 낮은 차원으로 줄여  
줌.
```

```
print(a[0, :]) # 0행의 모든 값
```

Python, Numpy, Pytorch의 자료형과 변환



1. Python의 자료형 = **list** (리스트)
2. Numpy의 기본 자료형 = **array** (배열, ndarray)
3. Pytorch의 자료형 = **tensor** (텐서, ndarray와 유사한 자료형)

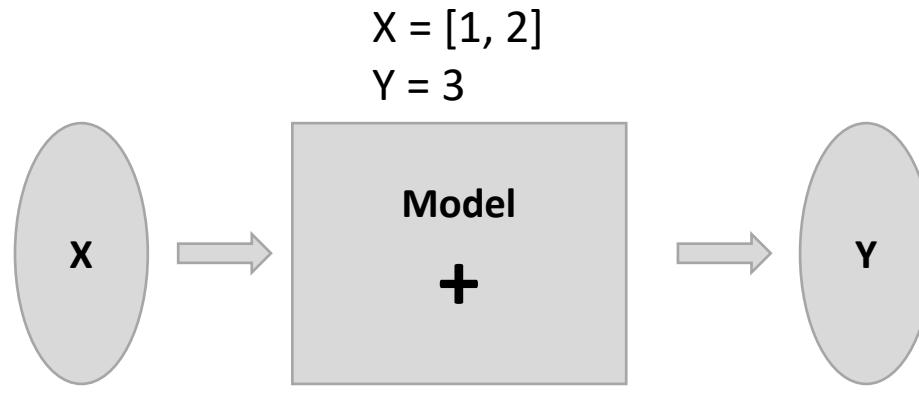
Python, Numpy, Pytorch의 자료형과 변환

```
# list -> ndarray, tensor
py_list = [1, 2, 3]
np_from_pylist = np.array(py_list)
torch_from_pylist = torch.FloatTensor(py_list)
torch_from_pylist = torch.tensor(py_list).float()
# torch tensor 변환 후 자료형 변경

# tensor -> list, ndarray
torch_tensor = torch.rand(4)
py_from_tensor = torch_tensor.tolist()
np_from_tensor = torch_tensor.numpy()

# ndarray -> list, tensor
numpy_array = np.array([[1,2],[3,4]])
py_from_ndarray = numpy_array.tolist()
torch_from_ndarray = torch.tensor(numpy_array).float()
```

X(input), Y(output), Model in ML_(machine learning)



e.g.) 더하기를 하는 Model

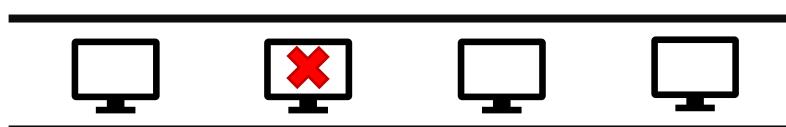
학습이 되지 않은 Model : 3 아냐? Feedback
Model([1,2])=5?

학습 후 Model : 잘했어.
Model([1,2])=3!

Next Step: 그러면 [4,6]도 해볼래?

Classification 이란?

- 분류(classification) 문제 (on categorical data) <->
- 회귀(regression) 문제 (on numerical data)
- What is Categorical Data?
 - 몇 가지 case로 분류가 되는 데이터
 - 생산 라인의 제품 불량 여부 (O와 X, 2가지로 분류)
 - 개 고양이 햄스터 사진 분류 (3가지로 분류)



dog

cat

hamster

Regression 이란?

- 분류(classification) 문제 (on categorical data) <->
 - 회귀(regression) 문제 (on numerical data)
-
- What is Numerical Data?
 - 숫자로 표현되는 데이터
 - 7시간 잔 다음 날, 달리기 기록이 몇 초가 나오는지. (second)
 - 주행 환경에 따라 속도가 100km/h까지 도달하는 데 걸리는 시간 (second)
 - 자동차 주행 습관을 바꿨을 때, 연비(리터당 주행거리)가 얼마나 오르고 내리는지. (km/l)



목차

Classification

- **Multi-layer Perceptron / Fully Connected Layer (FC Layer)**
 - ◆ 이미지 분류
- **CNN (Convolutional Neural Network)**
 - ◆ 이미지 분류 (MLP와 성능비교)

Regression

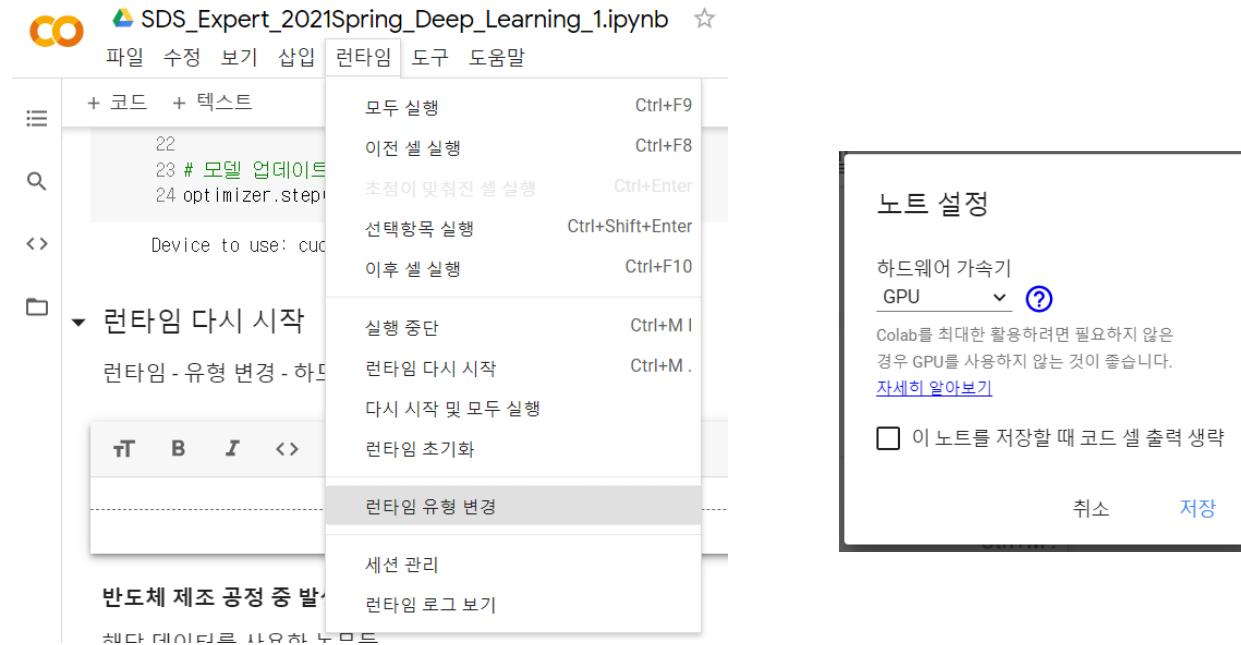
- **RNN (Recurrent Neural Network)**
 - ◆ 주가 예측

Multilayer Perceptron (MLP) 실습

Multilayer Perceptron (MLP)

For Colab!

런타임 유형 변경 – ‘GPU’ 선택 – 저장 – 런타임 다시 시작



이후에 각각 CPU로 실행했을 때와 GPU로 실행했을 때의 속도 차이를 비교해보세요

반도체 웨이퍼 데이터

```
mapping_type={'Center':0,'Donut':1,'Edge-Loc':2,'Edge-Ring':3,'Loc':4,'Random':5,'Scratch':6,'Near-full':7}
```

➤ 실습: MIR-WM811K 데이터셋

반도체 공정 중 나오는 불량 웨이퍼 분류

```
from google_drive_downloader import GoogleDriveDownloader as gdd
```

```
gdd.download_file_from_google_drive(  
    file_id='1NIQZLRCgt3yz6c6Yeb8IH9sguUF6ehCF',  
    dest_path='./preprocessed_LSWMD.pkl',  
)
```

```
data = df.waferMap.to_numpy()  
label=[df.failureNum[i] for i in range(df.shape[0])]  
label=np.array(label)
```

웨이퍼맵
이미지
데이터

Label
정답 레이블

X Y

Edge-Ring 3

Random 5

Loc 4

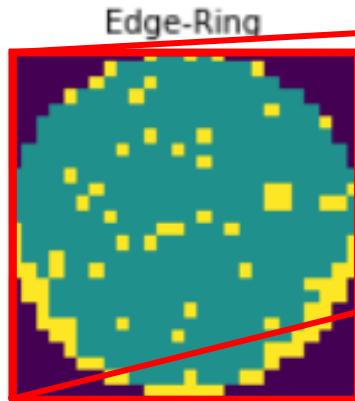
⋮ ⋮

➤ 데이터 불러오기

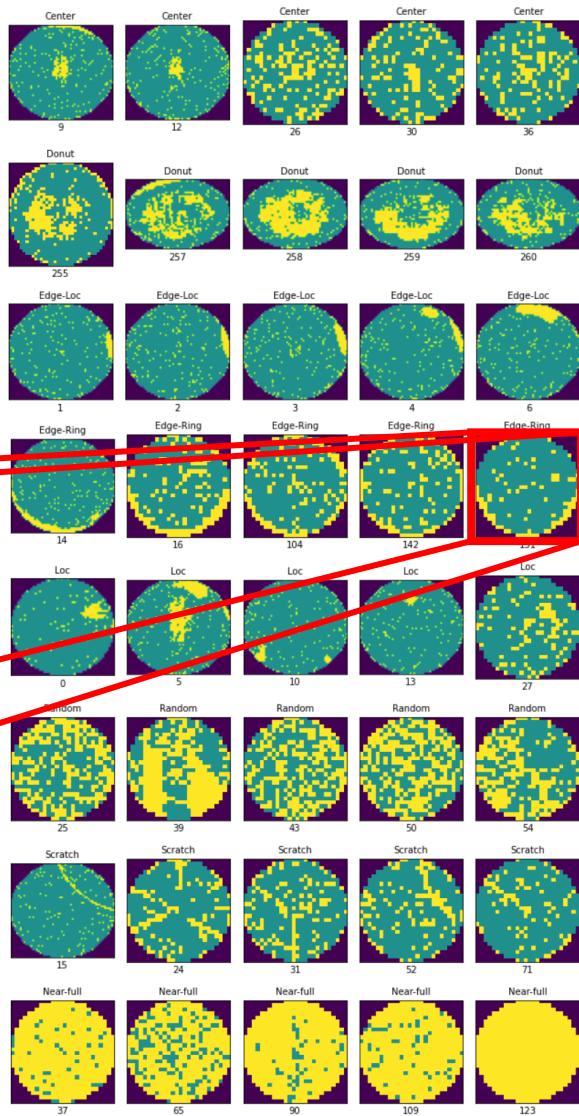
반도체 웨이퍼 데이터

➤ 실습: MIR-WM811K 데이터셋

- 반도체 공정 중 불량 웨이퍼 데이터
- 다양한 사이즈 존재.
- 64 X 64로 정규화 (4096개의 특징)

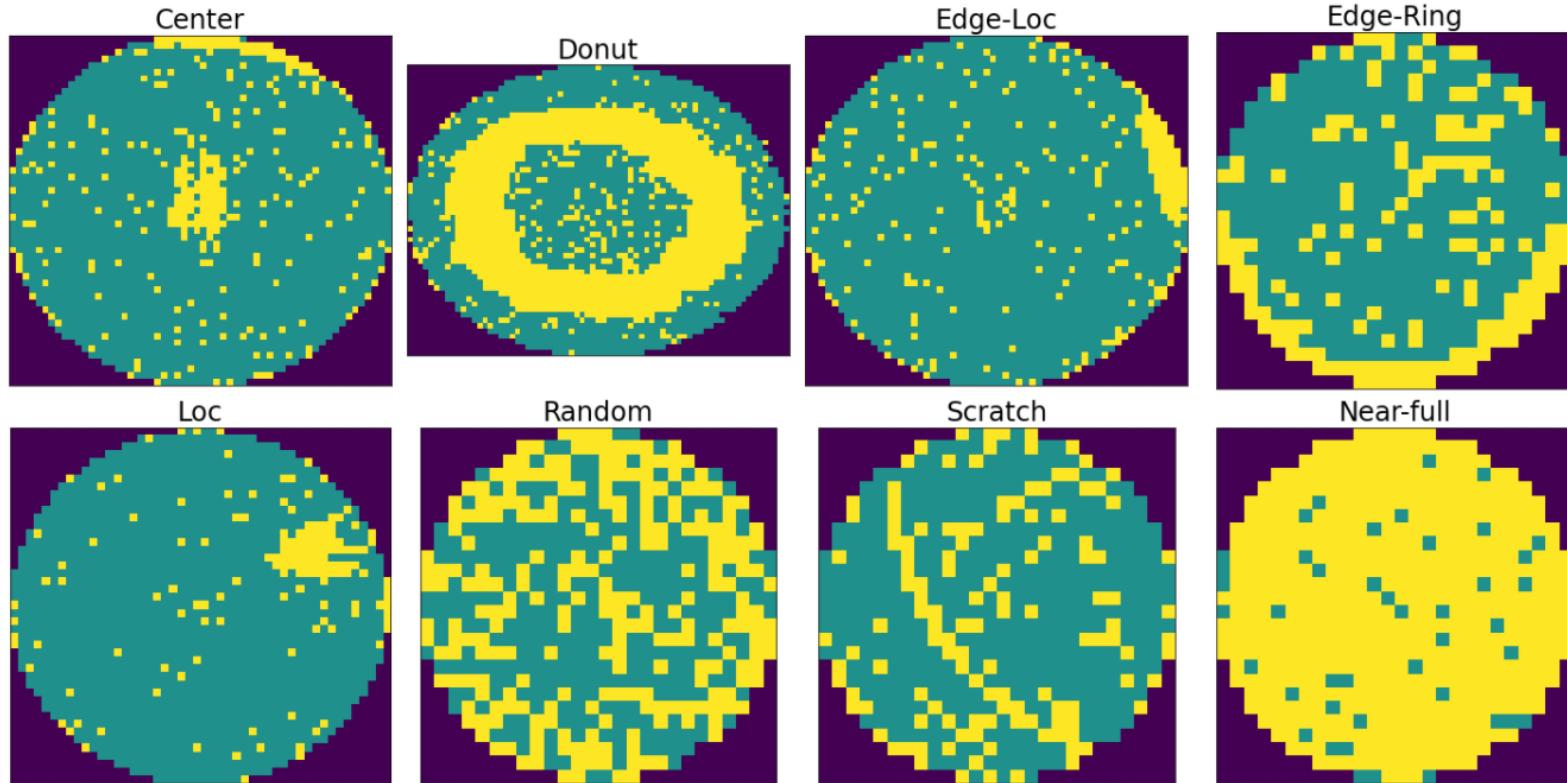


8가지 불량 case 존재



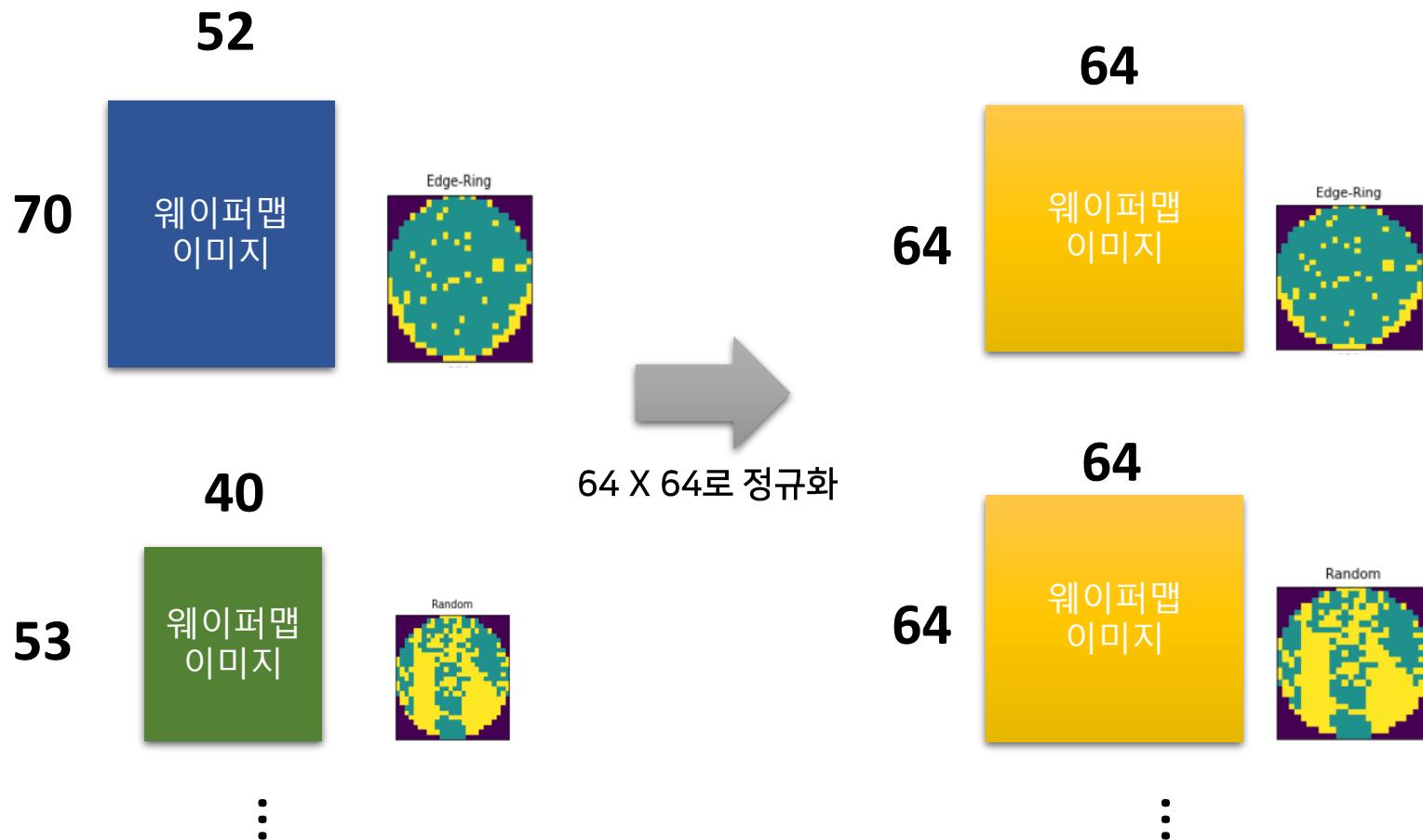
반도체 웨이퍼 데이터

➤ 8가지 불량 case



반도체 웨이퍼 데이터

- 실습: MIR-WM811K 데이터셋
- 같은 크기로 정규화



Dataset and Dataloader

```
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, x, y):
        self.x_data = x
        self.y_data = y
        self.transform = transforms.Compose([
            transforms.Resize((64,64)), # 이미지를 (64, 64) 크기로 변환
            transforms.ToTensor() # 이미지를 Tensor로 변환
        ])

    # 인덱스를 입력받아 해당 인덱스의 입출력 데이터를 리턴
    def __getitem__(self, idx):
        img = self.x_data[idx]
        img = Image.fromarray(img) # transform을 거치기 위해 python Image로 변환.
        img = self.transform(img).float()
        img = img.view(-1, 64*64).squeeze()
        target = self.y_data[idx]
        return img, target
```

- ◆ `__init__` : 데이터셋 정보
- ◆ `__getitem__` : idx를 input으로 받아 idx에 해당하는 이미지를 리턴

Dataset and Dataloader

```
train_dataset = CustomDataset(data, label)  
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=128,  
shuffle=True)
```

- ◆ DataLoader - 주어진 Dataset을 batch 단위로 나누어줌.
 - Batch_size : GPU의 메모리 크기에 따라 유동적으로 조정.
 - Shuffle : 주어진 dataset을 random하게 섞을지 결정.
 - Augmentation / Normalize : (선택 사항)

Mini-Batch Training

➤ Mini-batch 단위로 train해야 하는 이유

- ◆ Resource(메모리) 부족

예) 라면은 총 5개를 끓여야하는데 냄비에 2개씩
밖에 못 끓인다면?

Solution : 2개, 2개, 1개씩 끓임



➤ Dataloader

- ◆ Dataset을 정해진 mini-batch 크기로 나누어주는 모듈.
 - 데이터의 순서 섞어주기
 - 데이터 가공
 - 나머지 처리

반도체 웨이퍼 데이터

```
def display_augmented_images(aug_f):
    fig, subplots = plt.subplots(2, 5, figsize=(20, 8))
    for i, idx in enumerate([9, 340, 3, 16, 0]):# 임의로 고른 9, 340, 3, 16, 0 번째 이미지
        axi1 = subplots.flat[i] # '9'의 차례 때 i는 0 (enumerate)
        axi2 = subplots.flat[i+5]

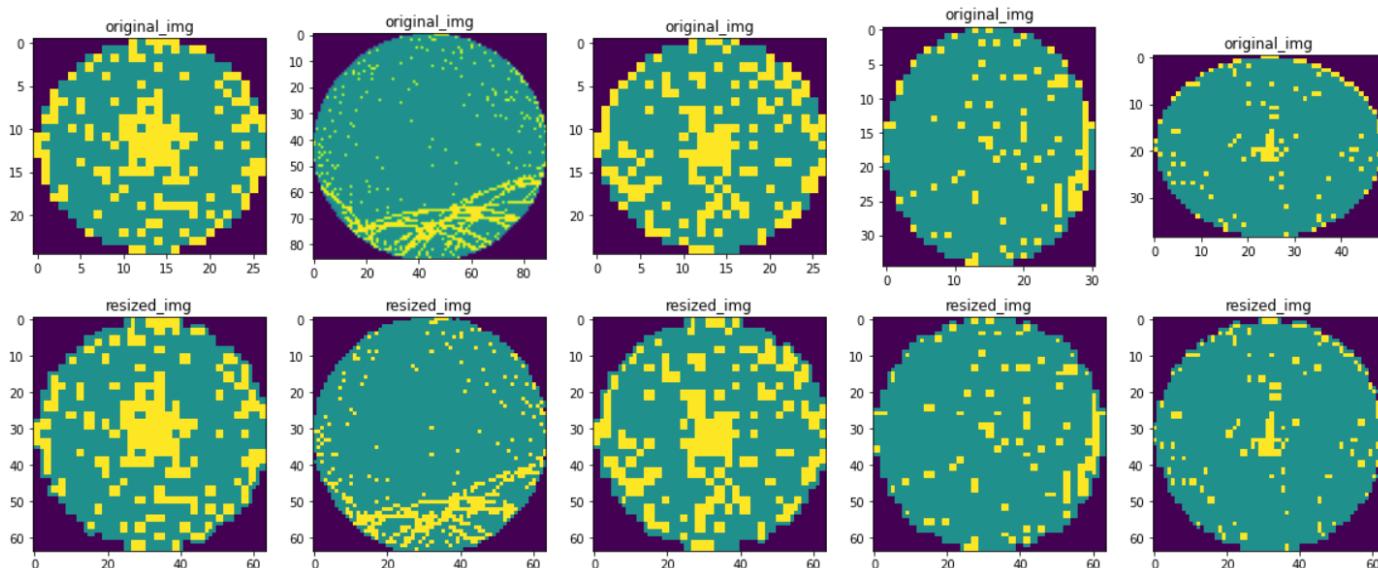
        original_img = Image.fromarray(train_dataset.x_data[idx])
        augmented_img = aug_f(original_img)

        axi1.imshow(original_img)
        axi2.imshow(augmented_img)
        axi1.set_title('original_img')
        axi2.set_title('resized_img')
```

- **display_augmented_images()** : 이미지에 aug_f를 적용하기 전과 후를 보여주는 그래프를 그리는 함수

반도체 웨이퍼 데이터

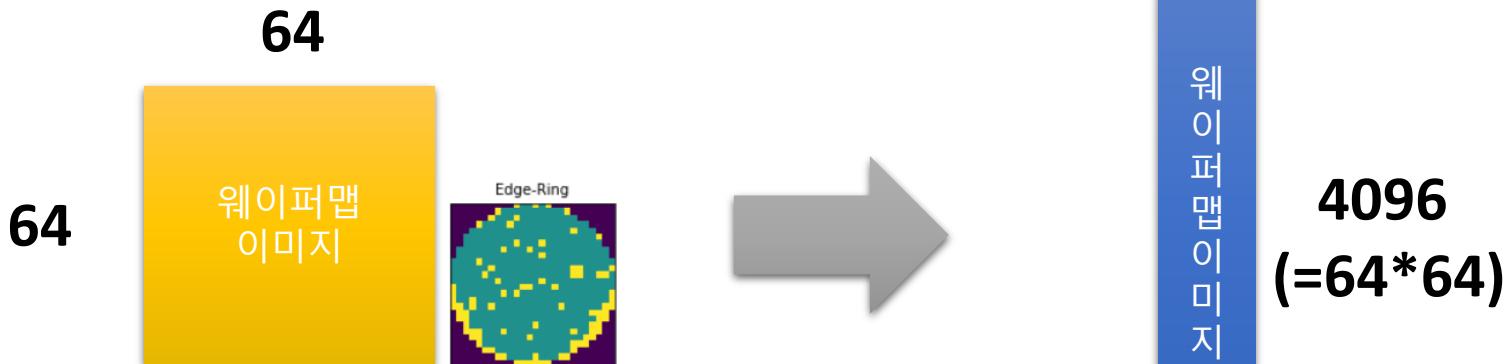
```
transform = transforms.Compose([
    # transforms.Pad(4),
    transforms.Resize((64,64)),
    # transforms.RandomHorizontalFlip(),
])
display_augmented_images(transform)
```



cf. : ctrl+? 단축키로 주석을 제거 혹은 #을 지워서 다른 전처리 방법도 확인해보세요

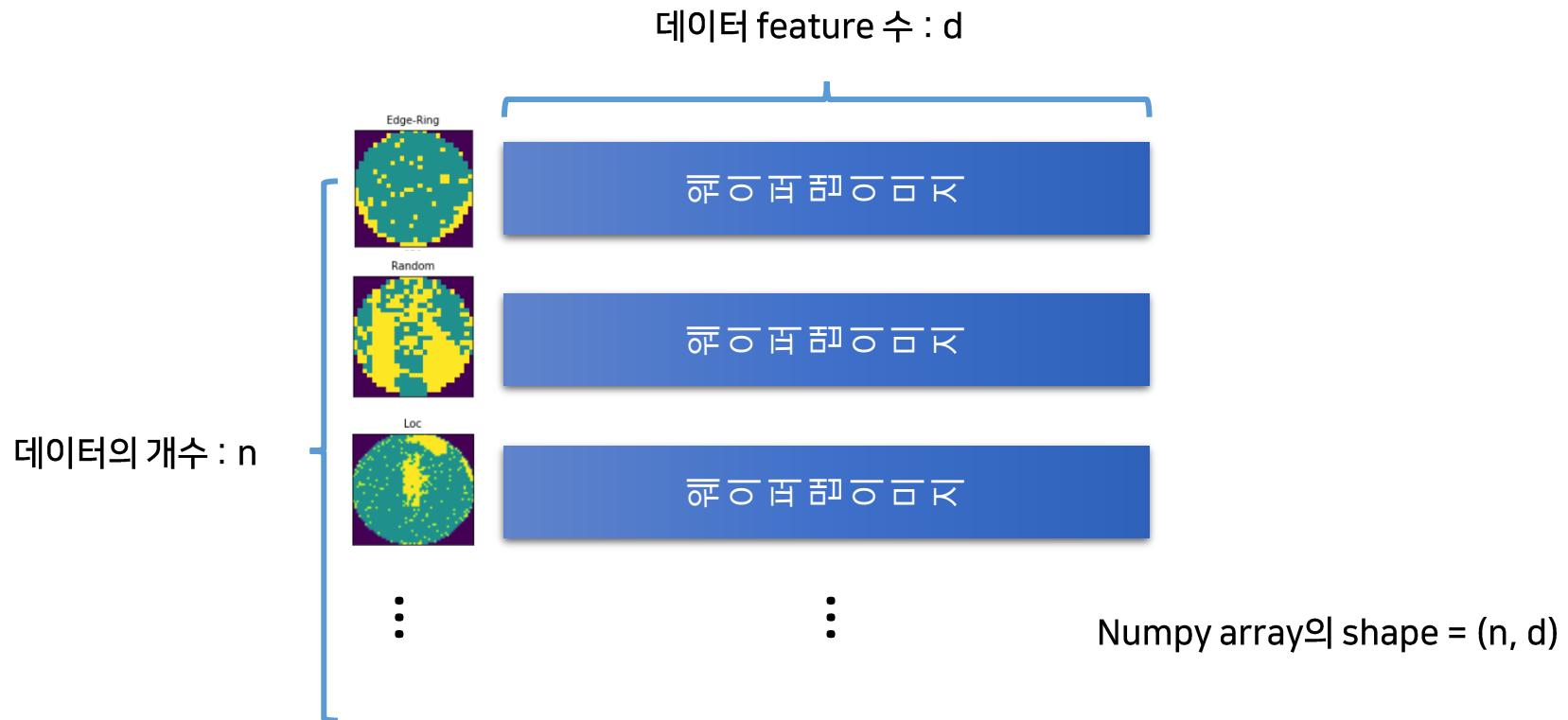
반도체 웨이퍼 데이터

- 실습: MIR-WM811K 데이터셋

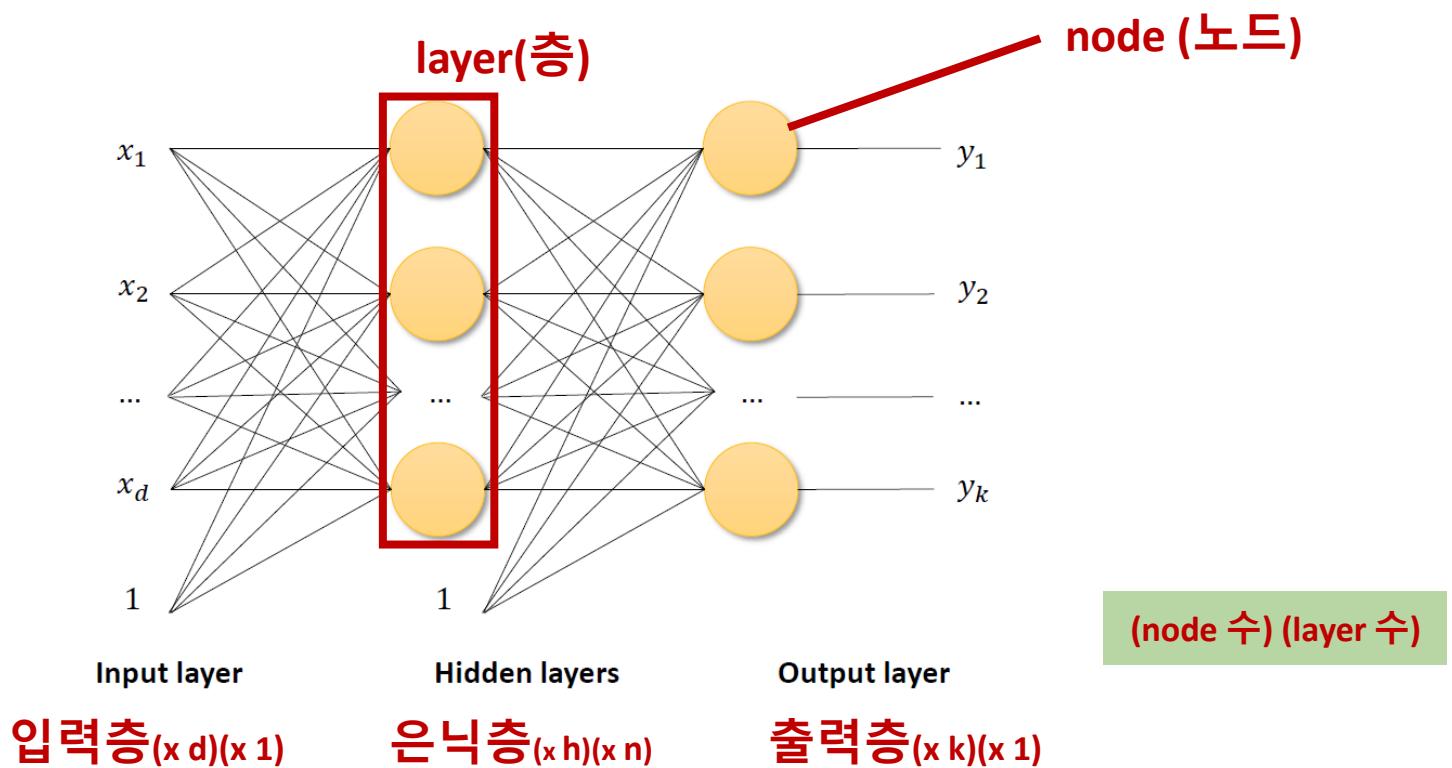


Multilayer Perceptron (MLP)

▶ 실습: MIR-WM811K 데이터셋

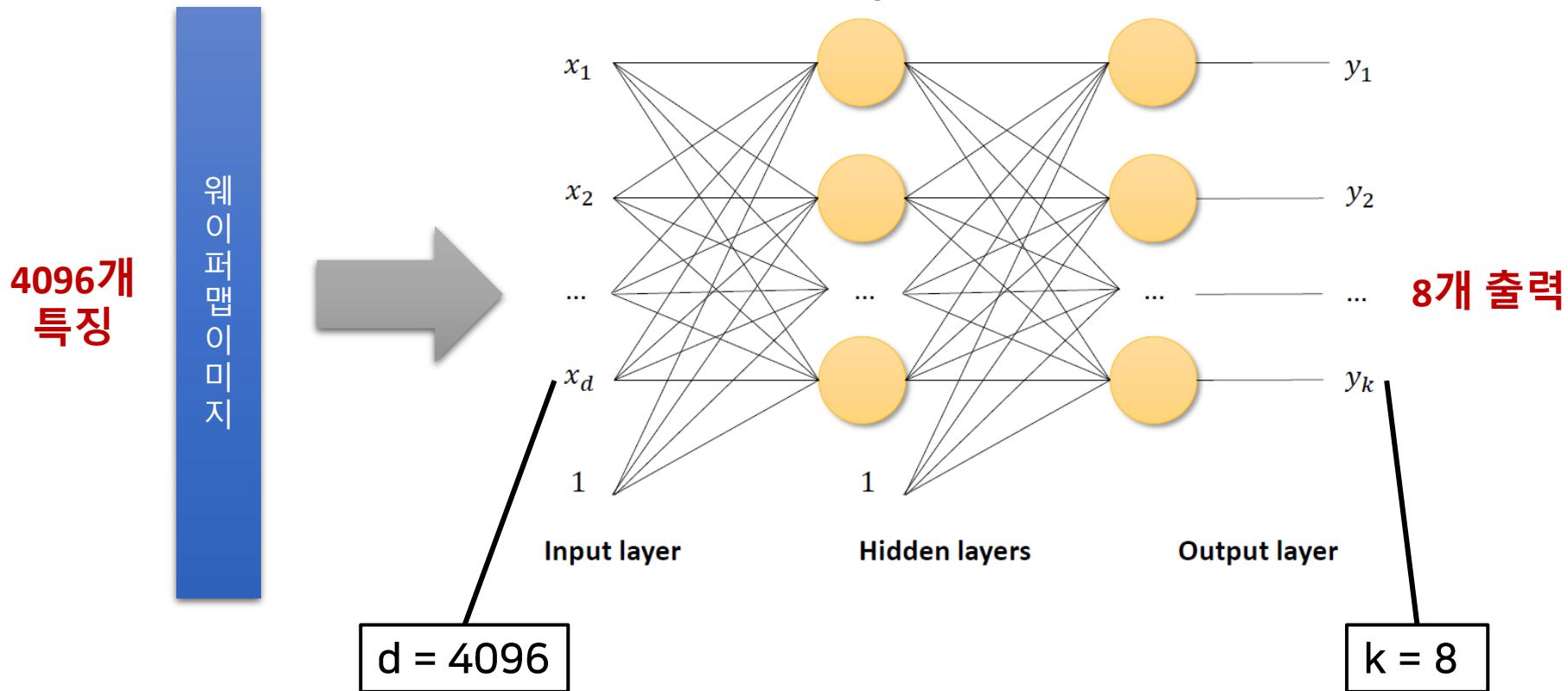


Multilayer Perceptron (MLP)



Multilayer Perceptron (MLP)

➤ 실습: MIR-WM811K 데이터셋



직접 Deep Learning Model 만들고 학습하기

```
class MLPClassifier_complex(nn.Module):
    def __init__(self):
        super(MLPClassifier, self).__init__()
        # self.linear1 = nn.Linear(in_features=16, out_features=32,bias=True)
        # self.linear2 = nn.Linear(in_features=32, out_features=8,bias=True)
        # self.relu = nn.ReLU()

    def forward(self, x):
        # x = self.relu(self.linear1(x))
        # x = self.linear2(x)
        return x
```

➤ 모델 설명

- ◆ 입력이 16차원, 출력이 32차원인 linear layer를 거친 후,
- ◆ ReLU activation을 거치고,
- ◆ 입력이 32차원, 출력이 8차원인 linear layer를 거친 후
- ◆ 최종 출력을 반환하는 모델.

직접 Deep Learning Model 만들고 학습하기

➤ 아래 설명을 만족하는 MLPClassifier_complex 모델을 완성하세요

[모듈]

- self.relu
 - 1차원에 적용
- self.linear1
 - 입력: 4096차원
 - 은닉: 512차원
- self.linear2
 - 입력: 512차원
 - 출력: 256차원
- self.linear3
 - 입력: 256차원
 - 출력: 9차원

```
1 class MLPClassifier_complex(nn.Module):  
2     def __init__(self):  
3         super(MLPClassifier_complex, self).__init__()  
4         # self.linear1 = nn.Linear(in_features=16, out_features=32,bias=True)  
5         # self.relu = nn.ReLU()  
6         #####  
7         #####  
8         #####  
9     def forward(self, x):  
10        # x = self.relu(self.linear1(x))  
11        #####  
12        #####  
13        #####  
14        #####  
15        return x
```

[동작]

- 입력을 받아 self.linear1 통과
- linear1에서 값을 받아 self.relu 통과
- self.linear2에 통과
- linear2에서 값을 받아 self.relu 통과
- self.linear3에 통과
- self.linear3 출력 반환

직접 Deep Learning Model 만들고 학습하기

```
class MLPClassifier_complex(nn.Module):
    def __init__(self):
        super(MLPClassifier_complex, self).__init__()
        self.linear1 = nn.Linear(in_features=4096, out_features=512,bias=True)
        self.linear2 = nn.Linear(in_features=512, out_features=256,bias=True)
        self.linear3 = nn.Linear(in_features=256, out_features=8,bias=True)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.linear1(x))
        x = self.relu(self.linear2(x))
        x = self.linear3(x)
        return x
```

➤ (코드 내 ‘정답’ 셀을 더블 클릭하면 코드가 보입니다.)

직접 Deep Learning Model 만들고 학습하기

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'  
print(f'Device to use: {device}')  
  
# 5 Random shape data with random label  
x = torch.rand(5, 4096).to(device)  
y = torch.LongTensor([0, 3, 1, 4, 7]).to(device)  
  
model = MLPClassifier_complex().to(device)  
  
learning_rate = 0.01  
criterion = nn.CrossEntropyLoss()  
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

- Pytorch는 CPU, GPU 모두 이용해 학습이 가능
주의사항 ! data, model이 같은 device에 존재해야 함.
- 임의의 Data 생성
- Model, 학습에 필요한 hyperparameter, loss, optimizer 모듈 선언

직접 Deep Learning Model 만들고 학습하기

```
# Gradient 기록 초기화  
optimizer.zero_grad()
```

```
# Model 출력  
out = model(x)
```

```
# Loss 계산  
loss = criterion(out, y)  
print(f'loss = {loss}')
```

```
# 모델 업데이트  
optimizer.step()
```

➤ 자동으로 model을 update. (총 3개의 과정을 거치게 됨)

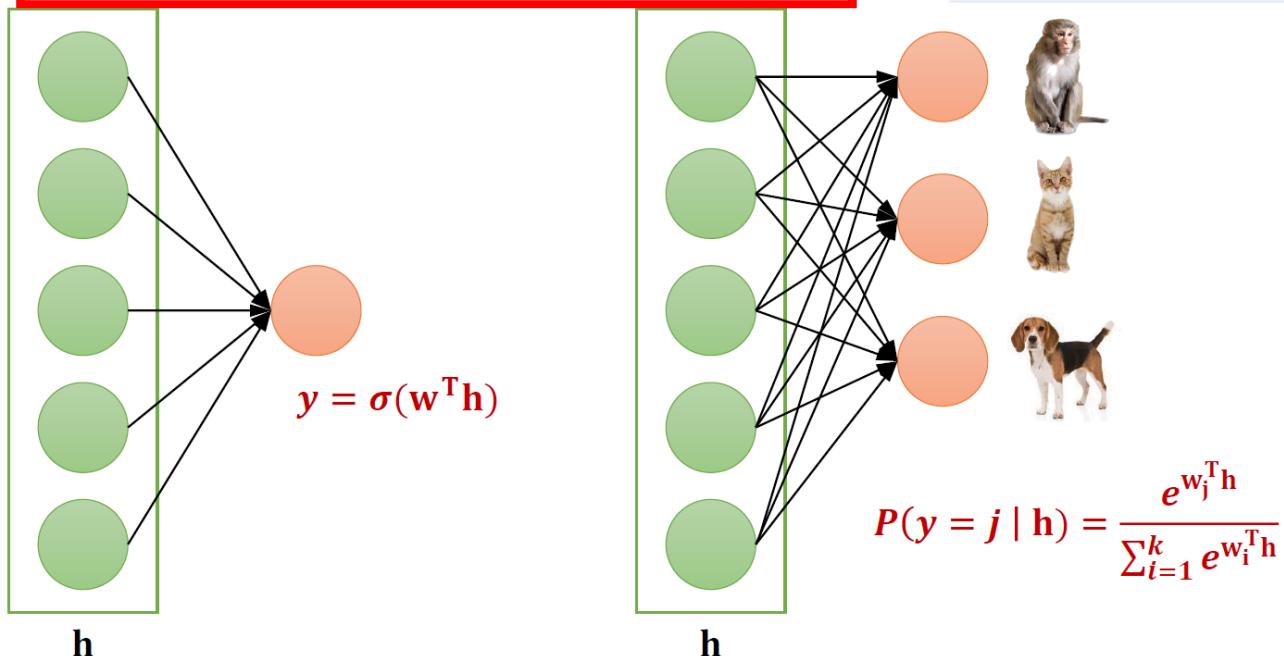
- ◆ optimizer.zero_grad()
- ◆ loss = criterion(out, y)
- ◆ optimizer.step()

How to Choose Loss Functions

모델이 수행하려는 Task에 따라 Loss를 계산하는 방법이 달라집니다.

➤ For output, there are three cases:

- ◆ Regression: $y = w^T h$
- ◆ Binary classification: $y = \sigma(w^T h)$
- ◆ Multi-class classification: $y = \text{softmax}(w^T h)$



Loss functions in torch.nn

MeanSquaredError == MSELoss

BinaryCrossEntropy == BCELoss

CrossEntropy == CrossEntropyLoss

loss가 보통 0에 가까울수록 모델이 주어진 데이터에 맞게 학습된 것

Multilayer Perceptron (MLP)

```
for step in range(100):
    pred = model(x) # 값 예측
    cost = criterion(pred.squeeze(), y) # Loss 계산
    optimizer.zero_grad()
    cost.backward()          # Backpropagation을 통해 모델 update
    optimizer.step()
    if step % 10 == 0: # 10번째 학습마다 Loss가 어떻게 줄어드는지 확인
        print(f'{step}th Epoch Loss {cost.item():.3f}')
```

- pred – 현재까지 학습된 model이 주어진 x에 대해 예측한 값
- cost – 예측한 값이 실제 값과 얼마나 차이가 나는지 계산 후 backpropagation을 통해 모델에 반영

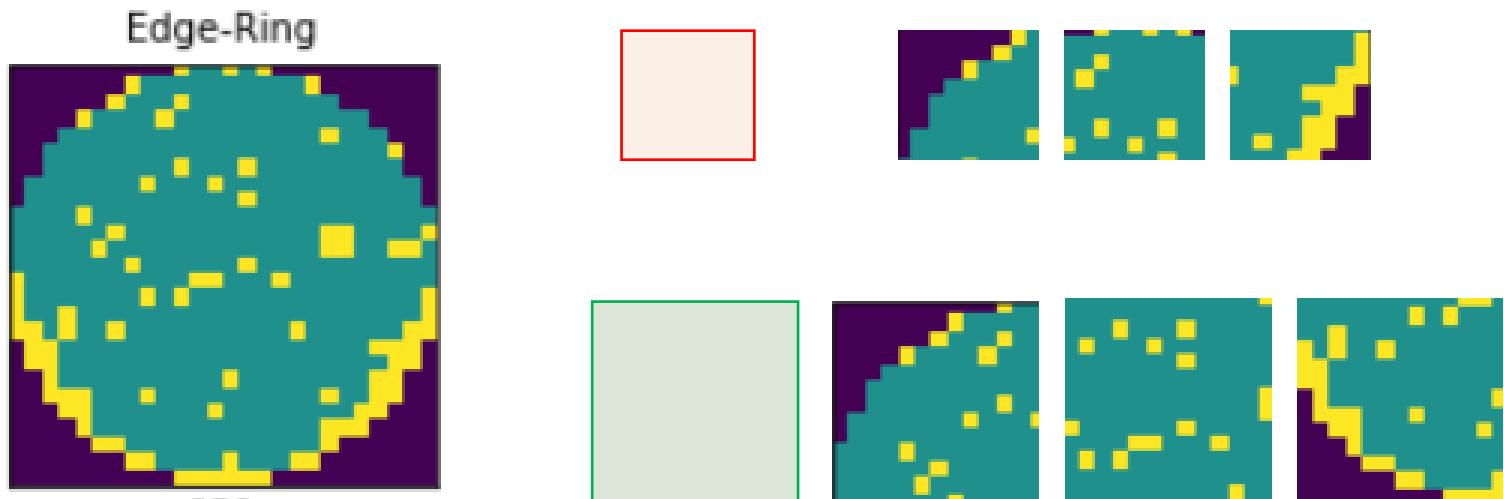
```
[Epoch: 18] cost = 0.2728
Train Accuracy : 89.08%
Test Accuracy : 84.40%
[Epoch: 19] cost = 0.2745
Train Accuracy : 88.85%
Test Accuracy : 83.97%
[Epoch: 20] cost = 0.2638
Train Accuracy : 88.99%
Test Accuracy : 83.72%
[Epoch: 21] cost = 0.2822
Train Accuracy : 88.47%
Test Accuracy : 84.76%
```

실행 결과 :

CNN 실습

Convolutional Neural Network

- 다양한 Kernel로 각기 다른 Local Feature를 얻을 수 있음.
- Neural Network를 통해 좋은 Kernel를 학습.
- Kernel이 곧 weight.



Convolution Layer의 동작

-1	-1	1
-1	1	-1
1	-1	-1

$$(I * K)_{xy} = \sum_{i=1}^w \sum_{j=1}^w K_{ij} \cdot I_{x+i-1, y+j-1}$$

0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	1	0	1	1	0	0	0
0	1	0	0	0	0	1	0	0
0	1	0	0	0	0	0	1	0
0	0	1	0	1	1	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0

Convolution Layer의 동작

-1	-1	1
-1	1	-1
1	-1	-1

$$(I * K)_{xy} = \sum_{i=1}^w \sum_{j=1}^w K_{ij} \cdot I_{x+i-1, y+j-1}$$

0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	1	0	1	1	0	0	0
0	1	0	0	0	0	1	0	0
0	1	0	0	0	0	1	0	0
0	0	1	0	1	1	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0

	-1							
	-2							

Convolution Layer의 동작

-1	-1	1
-1	1	-1
1	-1	-1

$$(I * K)_{xy} = \sum_{i=1}^w \sum_{j=1}^w K_{ij} \cdot I_{x+i-1, y+j-1}$$

0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	1	0	1	1	0	0
0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	0
0	0	1	0	1	1	0	0
0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0

Convolution Layer의 동작

- Input shape \neq Output shape
- Height, Width 측면
- Channel 측면

Convolution Layer의 동작 (Height, Width)

$$\text{Output size} = \frac{\text{input size} - \text{filter size} + (2 * \text{padding})}{\text{Stride}} + 1$$

- Input size = x,
- Filter size = 3, padding=1, stride=1,
- Output size = x.

Convolution Layer의 동작 (Height, Width)

- Kernel size = 3 & Padding = 1 & Stride = 1로 설정했을 때,
- Always Input H, W = Output H, W

1	2	3
4	5	6
7	8	9

Input (HxW)

0	0	0	0	0
0	1	2	3	0
0	4	5	6	0
0	7	8	9	0
0	0	0	0	0

Input with zero-padding = 1

stride = 1



kernel A (3x3)

output A

Convolution Layer의 동작 (Height, Width)

- Kernel size = 3 & Padding = 1 & Stride = 1로 설정했을 때,
- Always Input H, W = Output H, W

1	2	3	4
5	6	7	8
9	1	2	3
4	5	6	7

Input (HxW)

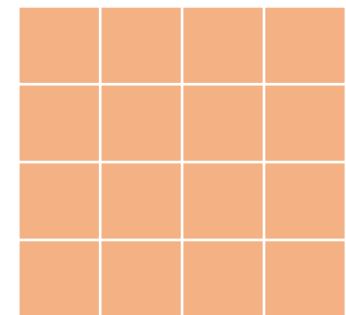
0	0	0	0	0	0	0
0	1	2	3	4	0	0
0	5	6	7	8	0	0
0	9	1	2	3	0	0
0	4	5	6	7	0	0
0	0	0	0	0	0	0

Input with zero-padding = 1

stride = 1



kernel A (3x3)



output A

Convolution Layer의 동작 (channel 수)

Input channel에서의 관점:

`input_dim`이 1보다 큰 경우 -> kernel의 두께가 두꺼워짐

`Output_dim = 1, kernel 수 = 1 (output dim)`

예) 컬러 RGB 이미지

1	2	3		
4	5	6		
7	8	9		

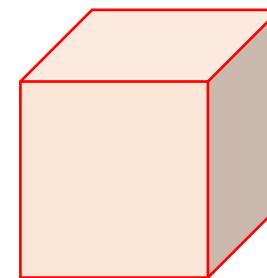
Input ($H \times W \times C$)

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	1	2	3	0	0	0
0	4	5	6	0	0	0
0	7	8	9	0	0	0
0	0	0	0	0	0	0

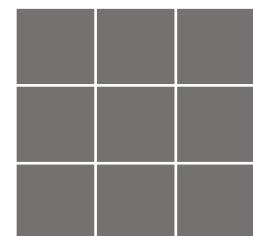
Input with zero-padding = 1

$C = \text{in_channels}$

RGB 3차원 컬러이미지를 흑백 이미지(feature map을 하나의 이미지로 간주)로 바꾼 것



kernel A (3x3x3)



Output

Convolution Layer의 동작 (channel 수)

Output channel에서의 관점:

input_dim은 = 1

Output_dim이 1보다 큰 경우
-> kernel의 개수가 많아짐

stride = 1

1	2	3
4	5	6
7	8	9

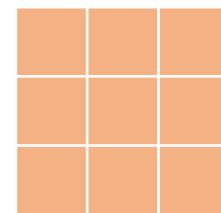
Input (HxW)

0	0	0	0	0
0	1	2	3	0
0	4	5	6	0
0	7	8	9	0
0	0	0	0	0

Input with zero-padding = 1



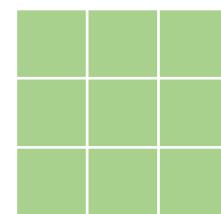
kernel A (3x3)



output A



kernel B (3x3)



output B

⋮

kernel N (3x3)

output N

N = out_channels

feature map도 하나의 이미지로 본다면 흑백이미지를 N차원 컬러이미지로 바꾼 것

연속적인 Convolution Layer

- Image size = $32 \times 32 \times 1$
- (Kernel size=3, padding=1, stride=1로 height와 width가 변하지 않을 때,)
- Convolution Layer1 : Input dim=1, Output dim=16
- Conv1(Image) = $32 \times 32 \times 16$
- Convolution Layer2 : Input dim=16, Output dim =64
- Conv2(Conv1(Image)) = $32 \times 32 \times 64$

중요!

Input의 dimension과 Output의 dimension을 맞춰주어야 합니다.

CNN 코드

```
class CNN_classifier(torch.nn.Module):
    def __init__(self):
        super(CNN_classifier, self).__init__()
        self.relu = torch.nn.ReLU()
        self.layer1 = torch.nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.layer2 = torch.nn.Conv2d(32, 16, kernel_size=3, stride=2, padding=1)
        self.layer3 = torch.nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1)
        self.fc1 = torch.nn.Linear(32*32*16, 512, bias=True)
        self.fc2 = torch.nn.Linear(512, 8, bias=True)

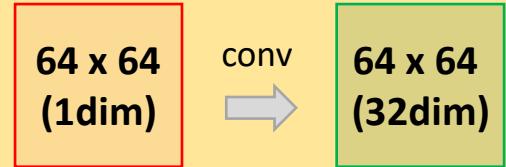
    def forward(self, x):
        out = self.relu(self.layer1(x))
        out = self.relu(self.layer2(out))
        out = self.relu(self.layer3(out))
        out = out.view(out.size(0), -1) # FC의 input으로 넣기 위해 Flatten하는 과정.
        out = self.relu(self.fc1(out))
        out = self.fc2(out)
        return out
```

CNN 코드

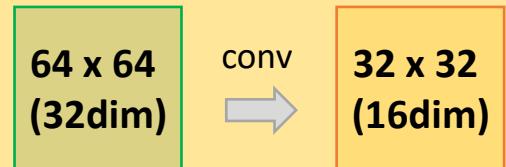
```
def __init__(self):
    super(CNN_classifier, self).__init__()
    self.relu = torch.nn.ReLU()

    # ?는 동시에 학습되는 이미지의 개수를 의미합니다. (= batch_size)
    # L1 ImgIn shape=(?, 64, 64, 1)
    # Conv  -> (?, 64, 64, 32)
    self.layer1 = torch.nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)

    # # L2 ImgIn shape=(?, 64, 64, 32)
    # # Conv  -> (?, 32, 32, 16)
    self.layer2 = torch.nn.Conv2d(32, 16, kernel_size=3, stride=2, padding=1)
```



- ReLU는 인자를 받지 않습니다.
- Conv2d는 총 5개의 인자를 받습니다.
in_channels, out_channels, kernel_size, stride, padding



CNN 코드

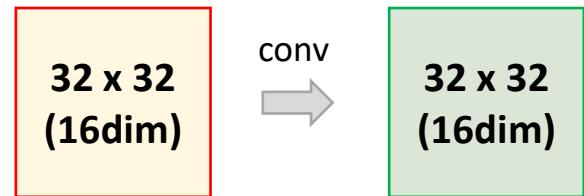
```
self.layer3 = torch.nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1)
```

```
self.fc1 = torch.nn.Linear(32*32*16, 512, bias=True)
```

L3 ImgIn shape=(?, 32, 32, 16)

Conv -> (?, 32, 32, 16)

L4 FC 32x32x16 inputs -> 512 outputs



- Convolution layer를 거친 뒤 항상 마지막에는 linear layer를 거쳐야합니다.
- Linear layer는 Neural Network에서 fully connected layer(fc layer)라고도 부릅니다.
- 마지막 Conv Layer의 output feature map의 크기와 (32*32*16)와 같도록 FC layer의 input dimension을 설정.

CNN 코드

```
# L4 FC 32x32x16 inputs -> 1024 outputs  
self.fc1 = torch.nn.Linear(32*32*16, 512, bias=True)
```

```
# L5 FC 1024 inputs -> 8 outputs  
self.fc2 = torch.nn.Linear(512, 8, bias=True)
```

L4 FC 32x32x16 inputs -> 512 outputs

L5 FC 512 inputs -> 8 outputs

- Neural Network의 task가 classification이라면,
마지막 FC Layer의 out_channel은 해당 classification의 class 수와 일치하게 만듭니다.
- (Task가 regression이라면 out_channel은 보통 1로 설정)
- WaferMap 불량 case가 8가지이기 때문에, 마지막 FC layer의 out_channel을 8로 설정.

Convolutional Neural Network

```
[Epoch: 16] loss = 0.3641  
Train Accuracy : 87.28%  
Test Accuracy : 85.33%  
[Epoch: 17] loss = 0.3464  
Train Accuracy : 87.04%  
Test Accuracy : 84.64%  
[Epoch: 18] loss = 0.3254  
Train Accuracy : 87.23%  
Test Accuracy : 85.60%  
[Epoch: 19] loss = 0.3103  
Train Accuracy : 88.95%  
Test Accuracy : 86.30%  
[Epoch: 20] loss = 0.3056  
Train Accuracy : 87.36%  
Test Accuracy : 84.01%
```

CNN

```
[Epoch: 18] cost = 0.2728  
Train Accuracy : 89.08%  
Test Accuracy : 84.40%  
[Epoch: 19] cost = 0.2745  
Train Accuracy : 88.85%  
Test Accuracy : 83.97%  
[Epoch: 20] cost = 0.2638  
Train Accuracy : 88.99%  
Test Accuracy : 83.72%  
[Epoch: 21] cost = 0.2822  
Train Accuracy : 88.47%  
Test Accuracy : 84.76%
```

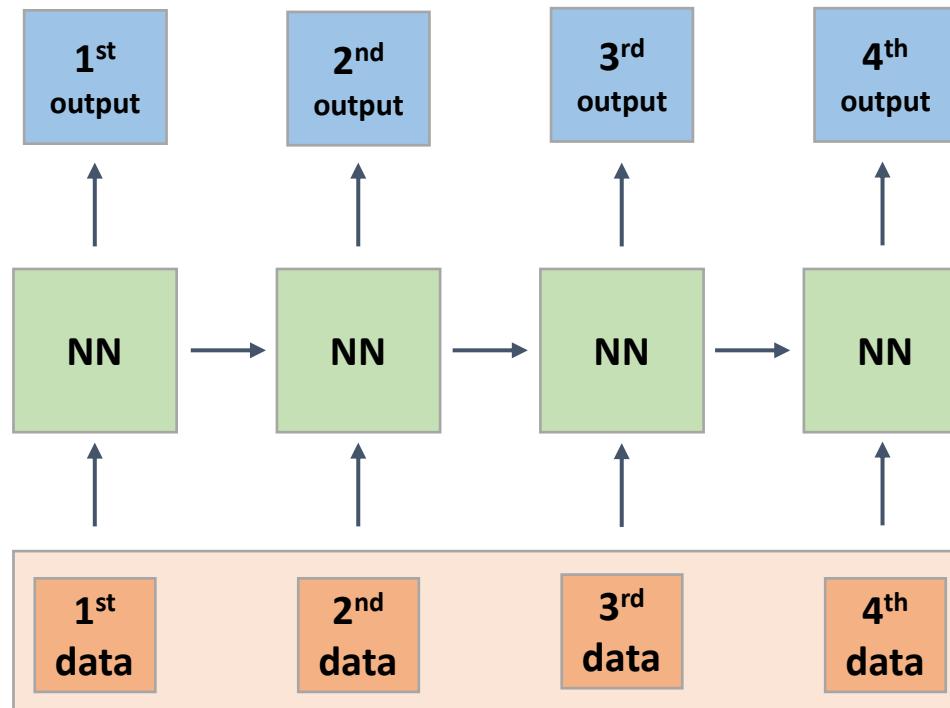
MLP

CNN 모델과 MLP 모델의 불량 웨이퍼 분류 성능 비교

- 높은 해상도의 이미지, 컬러이미지에서 성능, 시간 측면 큰 차이

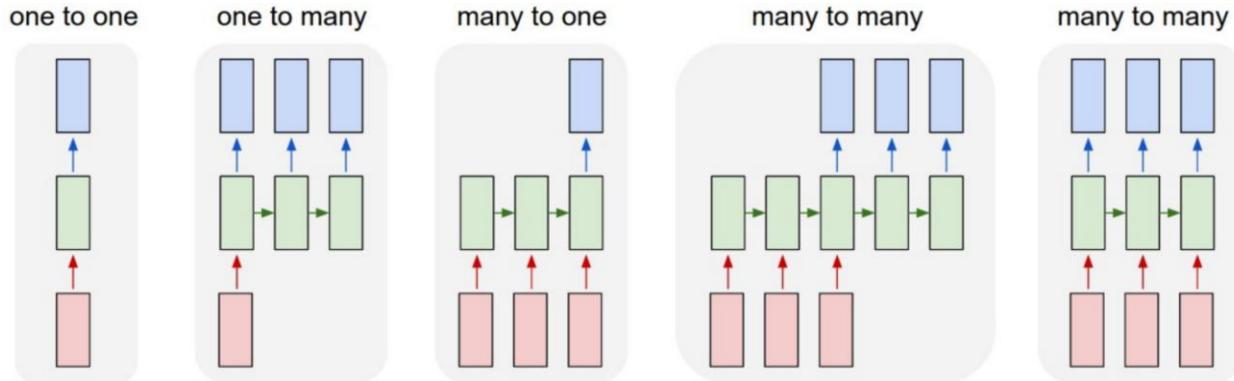
RNN 실습

RNN의 구조



Sequential data
1번째 – 2번째 – 3번째 – 4번째 데이터가
서로 관련이 있는 연속적인 데이터

RNN의 구조 활용방식



- ◆ one to one : 일반 NN
- ◆ one to many : image captioning (이미지 자막 넣기)
- ◆ Many to one : 여러 단어(문장)이 들어오면 하나의 출력 (감정분석)
- ◆ Many to many(1) : 번역기 (한 문장이 전부 들어온 뒤에야 번역)
- ◆ Many to many(2) : 비디오 매 frame별 물체 인식

RNN을 활용한 주가 예측

➤ RNN을 활용해 테슬라(TSLA)의 주가를 예측해보겠습니다.
(Just for fun)

- ◆ 주식의 가격 데이터는 시간적으로 연속적인 데이터.

Tesla, Inc. (TSLA)
NasdaqGS - NasdaqGS Real Time Price. Currency in USD

816.12 +4.46 (+0.55%)
At close: February 12 4:00PM EST

Add to watchlist

Summary Chart Conversations Statistics Historical Data Profile Financials Analysis Options Holders Sustainability

Historical Data

Time Period: Feb 15, 2020 - Feb 15, 2021 ▾ Show: Historical Prices ▾ Frequency: Daily ▾ Apply

Download

Date	Open	High	Low	Close*	Adj Close**	Volume
Feb 12, 2021	801.26	817.33	785.33	816.12	816.12	23,701,700
Feb 11, 2021	812.44	829.88	801.73	811.66	811.66	21,580,700
Feb 10, 2021	843.64	844.82	800.02	804.82	804.82	36,216,100
Feb 09, 2021	855.12	859.80	841.75	849.46	849.46	15,157,700
Feb 08, 2021	869.67	877.77	854.75	863.42	863.42	20,161,700

연속적
sequential

Dataset 구성

input_dim (number of features)

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	2020-02-18	2125.02	2166.07	2124.11	2155.67	2155.67	2945600
3	2020-02-19	2167.8	2185.1	2161.12	2170.22	2170.22	2561200
4	2020-02-20	2173.07	2176.79	2127.45	2153.1	2153.1	3131300
5	2020-02-21	2142.15	2144.55	2088	2095.97	2095.97	4646300
6	2020-02-24	2003.18	2039.3	1987.97	2009.29	2009.29	6547000
7	2020-02-25	2026.42	2034.6	1958.42	1972.74	1972.74	6219100
8	2020-02-26	1970.28	2014.67	1960.45	1979.59	1979.59	5224600
9	2020-02-27	1934.38	1975	1882.76	1884.3	1884.3	8144000
10	2020-02-28	1814.63	1889.76	1811.13	1883.75	1883.75	9493800
11	2020-03-02	1906.49	1954.51	1870	1953.95	1953.95	6761700
12	2020-03-03	1975.37	1996.33	1888.09	1908.99	1908.99	7534500
13	2020-03-04	1946.57	1978	1922	1975.83	1975.83	4772900
14	2020-03-05	1933	1960.72	1910	1924.03	1924.03	4748200
15	2020-03-06	1875	1910.87	1869.5	1901.09	1901.09	5273600

Sequence_length
며칠 간의 연속적인 데이터를 볼 것인지

맞히려는 값
다음 거래일의 주가

Dataset 구성

input_dim (number of features)

Stride

Sequence_length T

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	2020-02-18	2125.02	2166.07	2124.11	2155.67	2155.67	2945600
3	2020-02-19	2167.8	2185.1	2161.12	2170.22	2170.22	2561200
4	2020-02-20	2173.07	2176.79	2127.45	2153.1	2153.1	3131300
5	2020-02-21	2142.15	2144.55	2088	2095.97	2095.97	4646300
6	2020-02-24	2003.18	2039.3	1987.97	2009.29	2009.29	6547000
7	2020-02-25	2026.42	2034.6	1958.42	1972.74	1972.74	6219100
8	2020-02-26	1970.28	2014.67	1960.45	1979.59	1979.59	5224600
9	2020-02-27	1934.38	1975	1882.76	1884.3	1884.3	8144000
10	2020-02-28	1814.63	1889.76	1811.13	1883.75	1883.75	9493800
11	2020-03-02	1906.49	1954.51	1870	1953.95	1953.95	6761700
12	2020-03-03	1975.37	1996.33	1888.09	1908.99	1908.99	7534500
13	2020-03-04	1946.57	1978	1922	1975.83	1975.83	4772900
14	2020-03-05	1933	1960.72	1910	1924.03	1924.03	4748200
15	2020-03-06	1875	1910.87	1869.5	1901.09	1901.09	5273600

Task : 주가 예측

X					
2125.02	2166.07	2124.11	2155.67	2155.67	2945600
2167.8	2185.1	2161.12	2170.22	2170.22	2561200
2173.07	2176.79	2127.45	2153.1	2153.1	3131300
2142.15	2144.55	2088	2095.97	2095.97	4646300
2003.18	2039.3	1987.97	2009.29	2009.29	6547000
2026.42	2034.6	1958.42	1972.74	1972.74	6219100
1970.28	2014.67	1960.45	1979.59	1979.59	5224600
1934.38	1975	1882.76	1884.3	1884.3	8144000
1814.63	1889.76	1811.13	1883.75	1883.75	9493800
1906.49	1954.51	1870	1953.95	1953.95	6761700

이 주어졌을 때, Y

1908.99

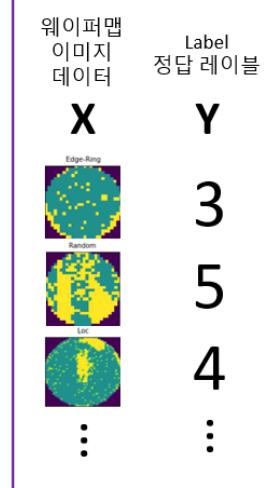
을 맞혀라.

2167.8	2185.1	2161.12	2170.22	2170.22	2561200
2173.07	2176.79	2127.45	2153.1	2153.1	3131300
2142.15	2144.55	2088	2095.97	2095.97	4646300
2003.18	2039.3	1987.97	2009.29	2009.29	6547000
2026.42	2034.6	1958.42	1972.74	1972.74	6219100
1970.28	2014.67	1960.45	1979.59	1979.59	5224600
1934.38	1975	1882.76	1884.3	1884.3	8144000
1814.63	1889.76	1811.13	1883.75	1883.75	9493800
1906.49	1954.51	1870	1953.95	1953.95	6761700
1975.37	1996.33	1888.09	1908.99	1908.99	7534500

이 주어졌을 때,

1975.83

을 맞혀라.



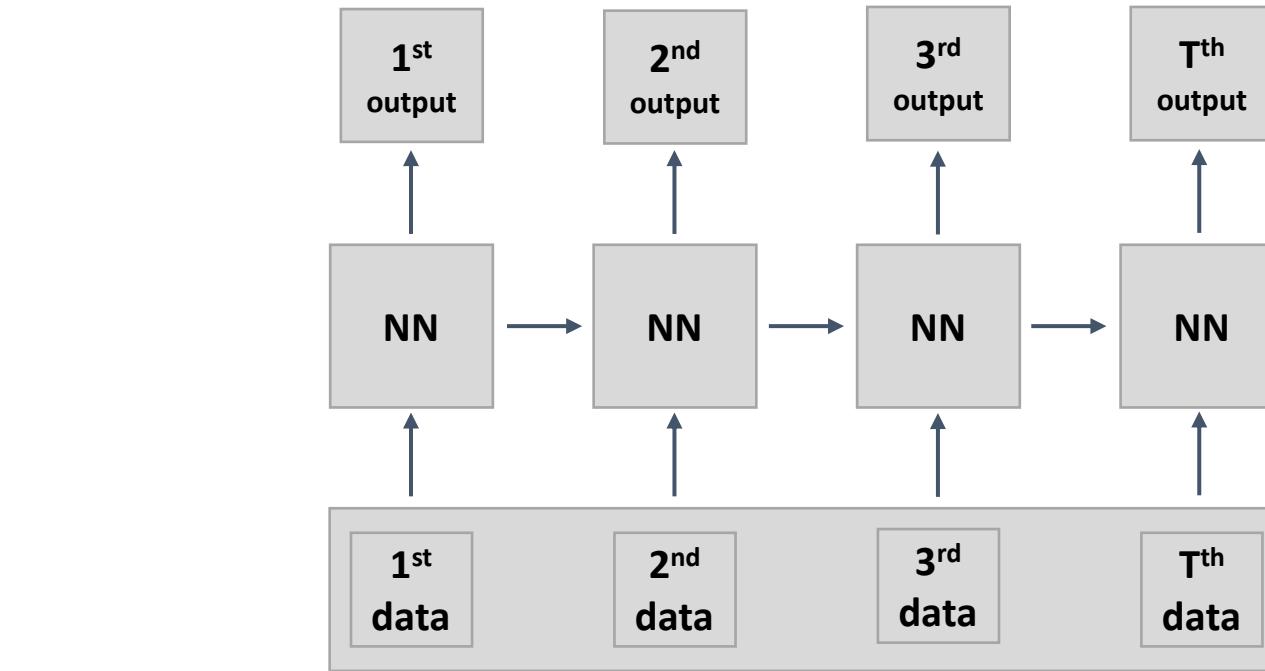
RNN의 구조

Y

1908.99

FC Layer

마지막 output만 사용



2125.02	2166.07	2124.11	2155.67	2155.67	2945600
2167.8	2185.1	2161.12	2170.22	2170.22	2561200
2173.07	2176.79	2127.45	2153.1	2153.1	3131300
2142.15	2144.55	2088	2095.97	2095.97	4646300
2003.18	2039.3	1987.97	2019.29	2009.29	6547000
2026.42	2043.6	1958.42	1972.74	1972.74	6219100
1970.28	2014.67	1960.45	1979.59	1979.59	5224600
1934.38	1975	1882.76	1884.3	1884.3	8144000
1814.63	1889.76	1811.13	1883.75	1883.75	9493800
1906.49	1954.51	1870	1953.95	1953.95	6761700

X input dim (number of features)

스케일링 (Scaling)

➤ 스케일링을 해야하는 이유?

- 변수의 스케일이 너무 작거나 큰 경우, 해당 변수를 학습할 때 모델이 크거나 작은 가중치를 가지게 될 수 있습니다.

Open	Close	Volume
1700	1800	2,600,000
1850	1740	1,800,000
1750	1780	2,500,000

- 주가에서의 1 차이와 거래량에서의 1 차이를 동일하게 봄에서는 안된다. → scaling 필요

스케일링 (Scaling)

➤ Min-Max Scaling

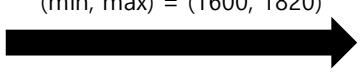
- 변수의 값의 범위가 0~1 사이로 변경됩니다.

$$X_i \rightarrow \frac{X_i - \min(X)}{\max(X) - \min(X)}$$

주가

1750	1680	1820	1800	1600
------	------	------	------	------

(min, max) = (1600, 1820)



0.68	0.36	1	0.91	0
------	------	---	------	---

거리량

2700000	2400000	2900000	3000000	2500000
---------	---------	---------	---------	---------

(min, max) = (2400000, 3000000)



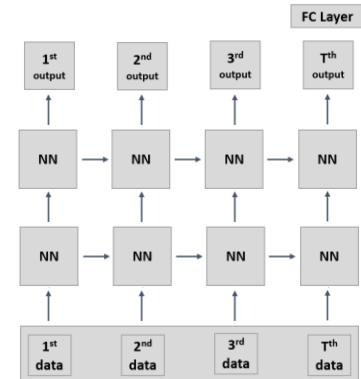
0.5	0	0.83	1	0.17
-----	---	------	---	------

RNN 코드

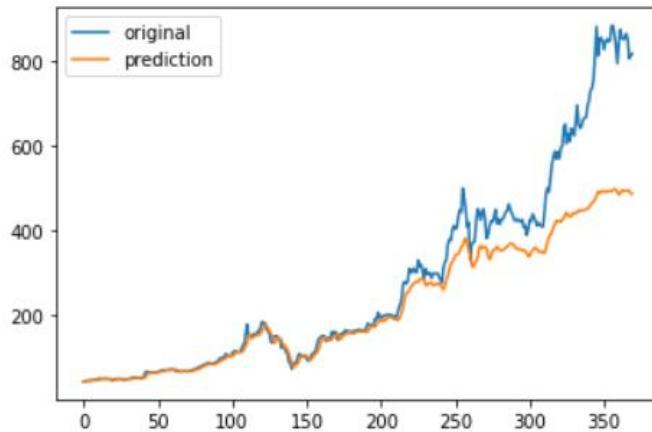
```
class RNNNet(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim=1, layers=2):
        super(RNNNet, self).__init__()
        self.rnn = torch.nn.RNN(input_dim, hidden_dim, num_layers=layers, batch_first=True)
        self.fc = torch.nn.Linear(hidden_dim, output_dim, bias=True)

    def forward(self, x):
        x, _status = self.rnn(x)
        x = self.fc(x[:, -1])
        return x
```

- Batch_first는 RNN이 받을 데이터의 모양을 결정.
- Layers = 2 : RNN 을 stack하여 쌓음.
- RNN의 마지막 출력을 FC layer에 넣음



RNN 학습 결과

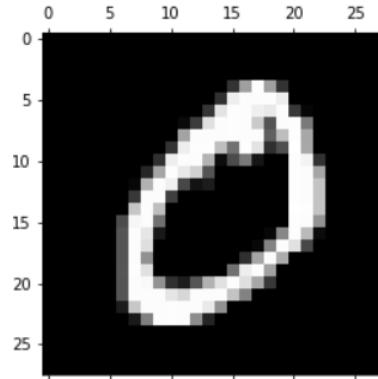


➤ 학습 후 실행 결과

- ◆ 단기적으로 급격하게 데이터가 바뀌는 경우 RNN이 오히려 불리한 측면이 있을 수도 있음.
- ◆ 과거 가격과 거래량만으로 미래 주가를 예측하는 것의 문제점.

평가: CNN으로 MNIST 분류하기

- ▶ 조건에 맞는 CNN 모델을 만들고, MNIST의 test data에 대해 **95% 이상의 정확도**를 가지도록 학습시켜보세요



평가: CNN으로 MNIST 분류하기

- ▶ 조건에 맞는 CNN 모델을 만들고, MNIST의 test data에 대해 95% 이상의 정확도를 가지도록 학습시켜보세요
- ▶ 9번 13번 17번째 줄에 있는 torch.nn.Conv2d 내부의 in_channel과 out_channel에 적절한 숫자를 채워주시고 학습을 진행해주세요.

조건

- self.layer1 : Conv2d - input_dim=1 / output_dim=32
- self.layer2 : Conv2d - input_dim=32 / output_dim=64
- self.layer3 : Conv2d - input_dim=64 / output_dim=32

Test Accuracy : 98.52%

Q&A

