PyTorch 설치

Pytorch 설치

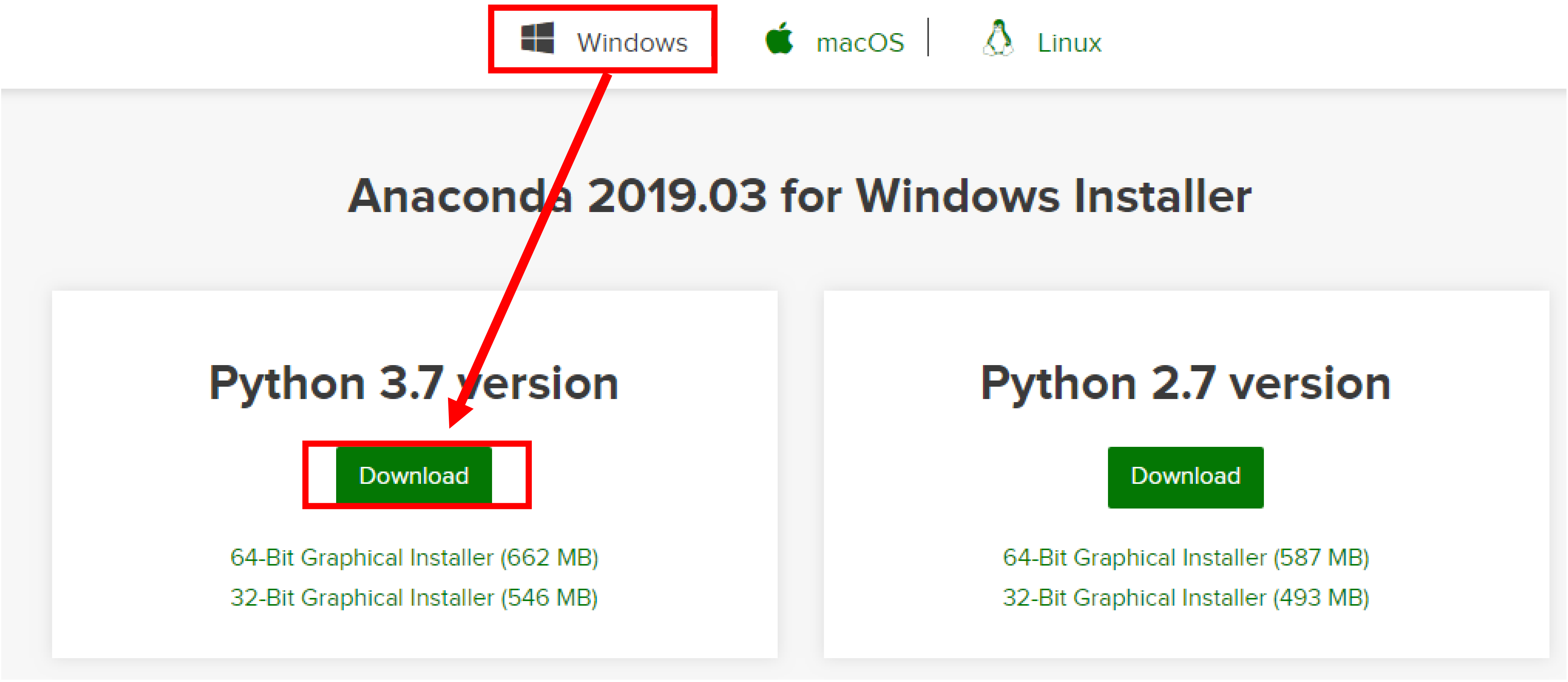
- Anaconda 설치

- Pycharm 설치 및 세팅

- Anaconda 가상환경에 PyTorch 설치하기

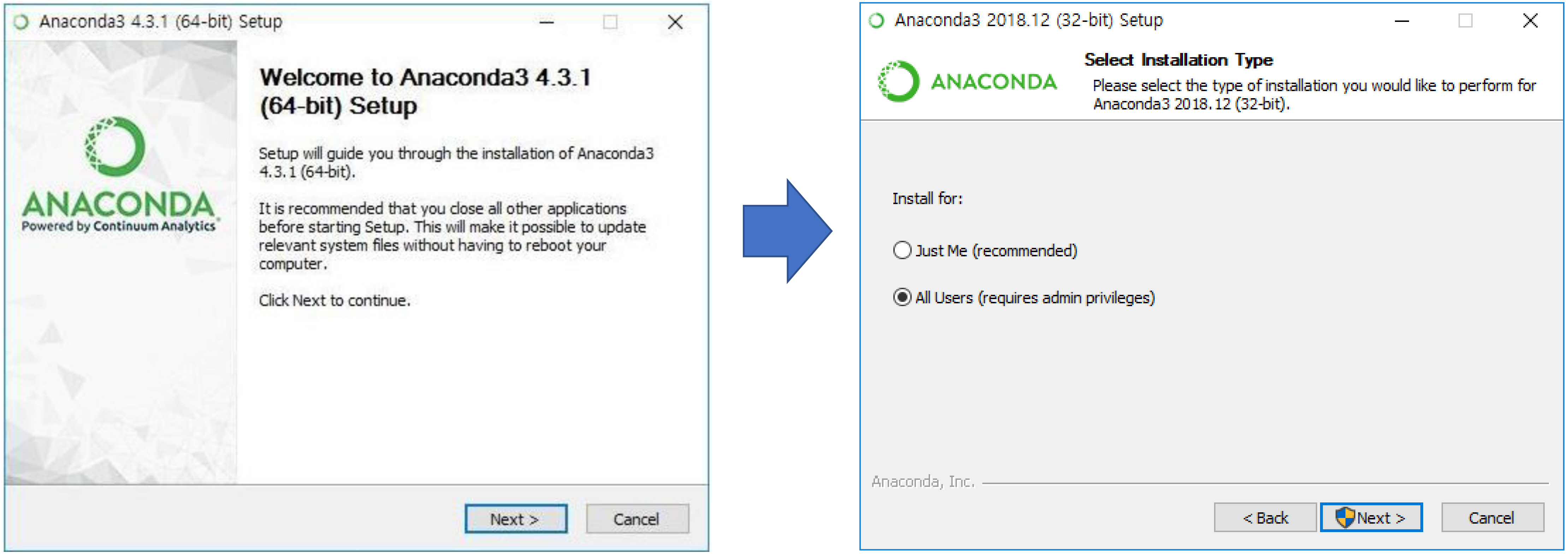
Anaconda 설치

- https://www.anaconda.com/distribution/ 접속하기
- 아래 그림과 같이 다운로드



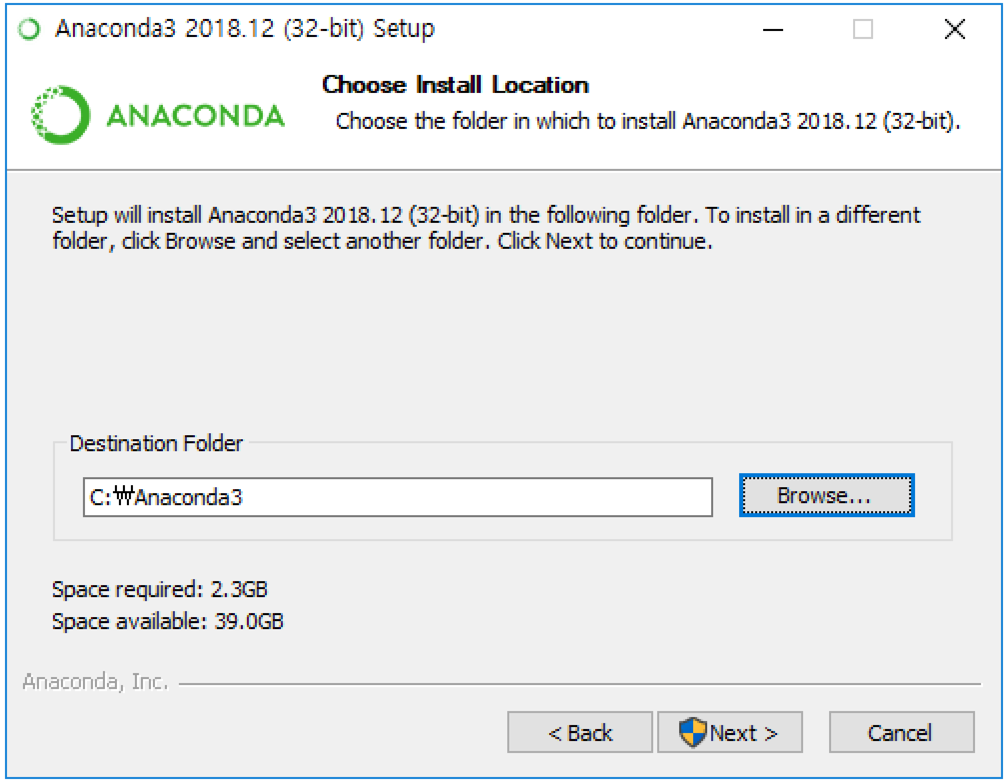
Anaconda 설치

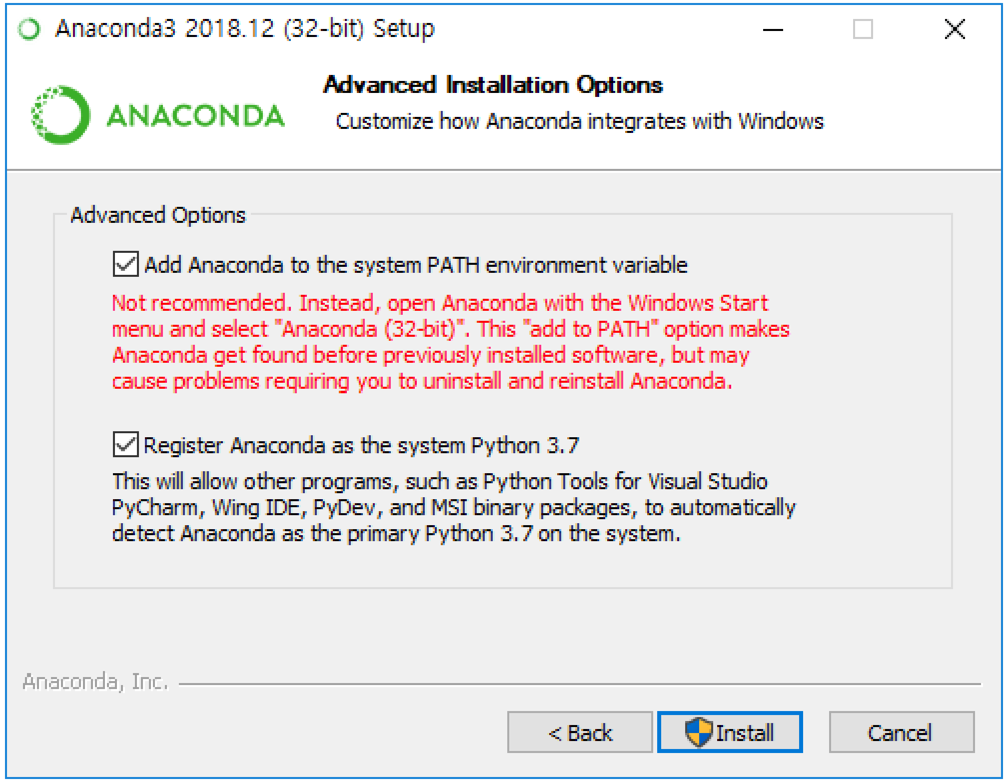
- 설치 파일 실행



Anaconda 설치

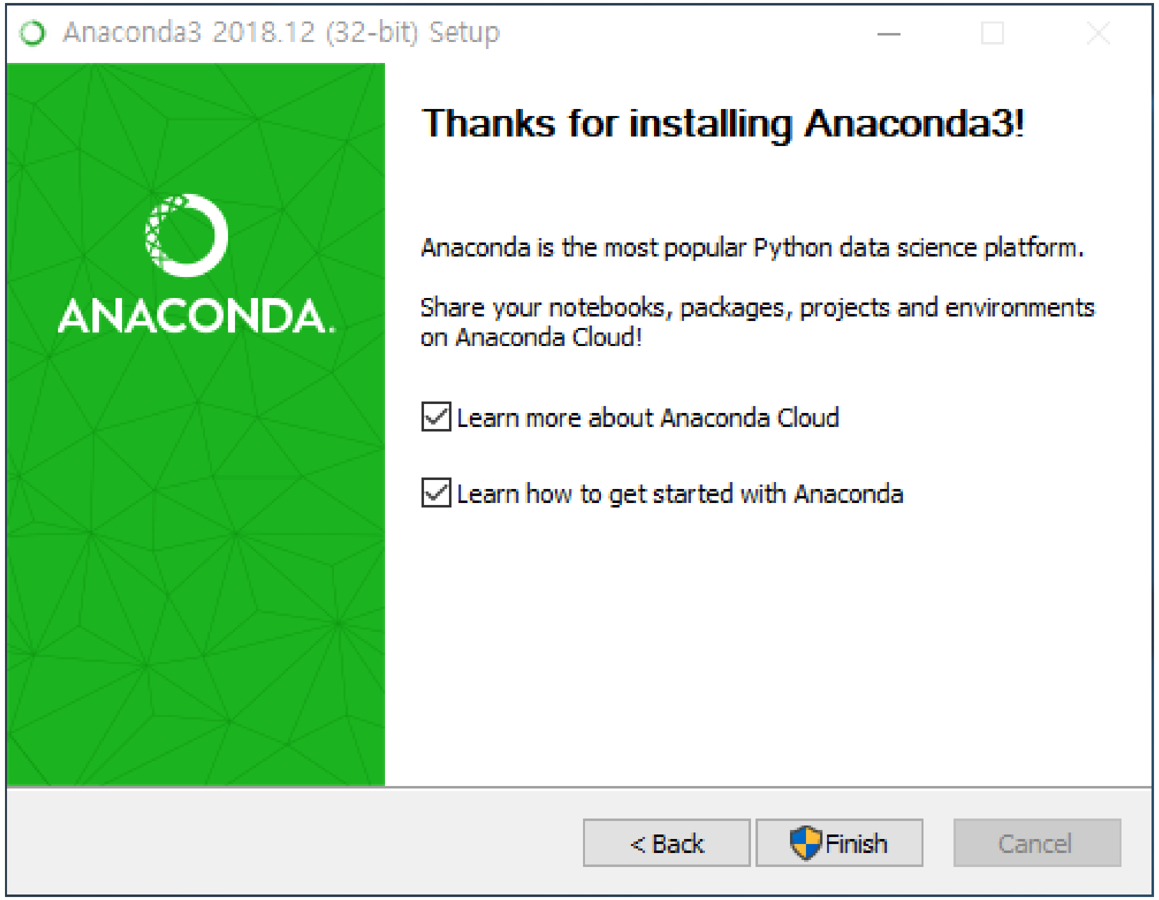
- 설치 파일 실행





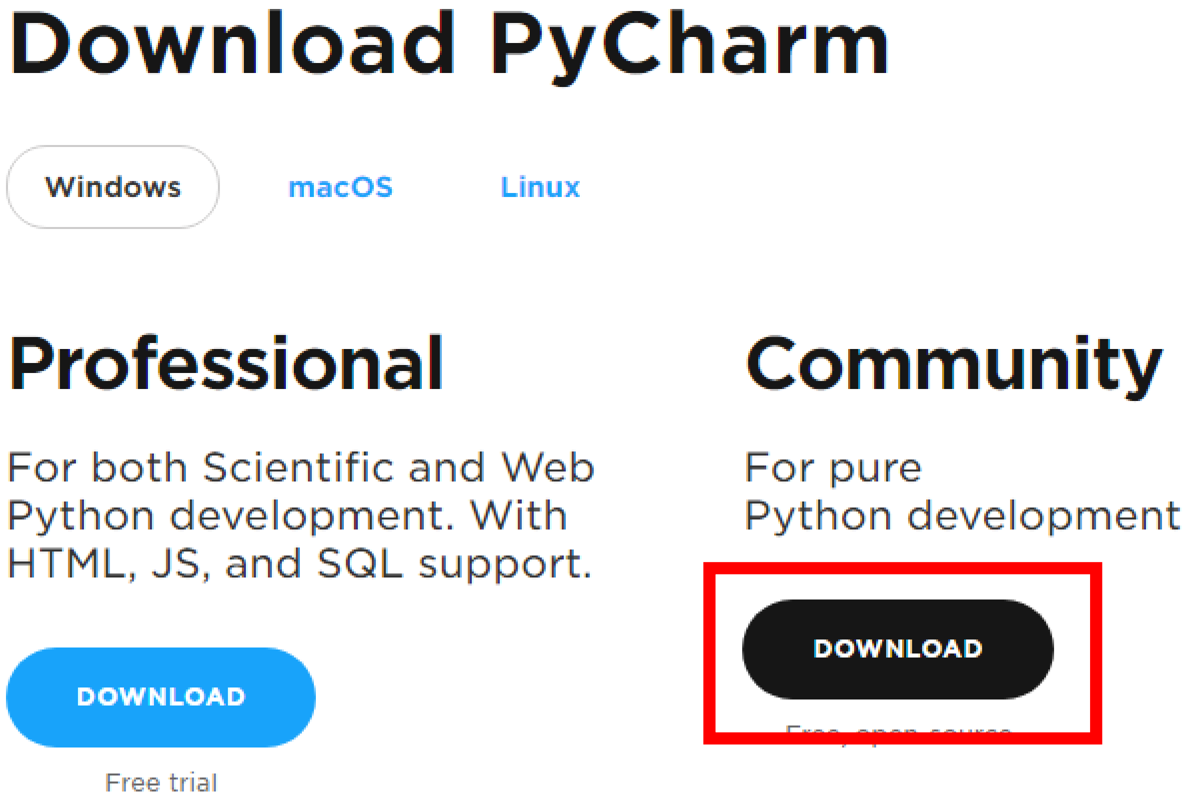
Anaconda 설치

- 설치 파일 실행



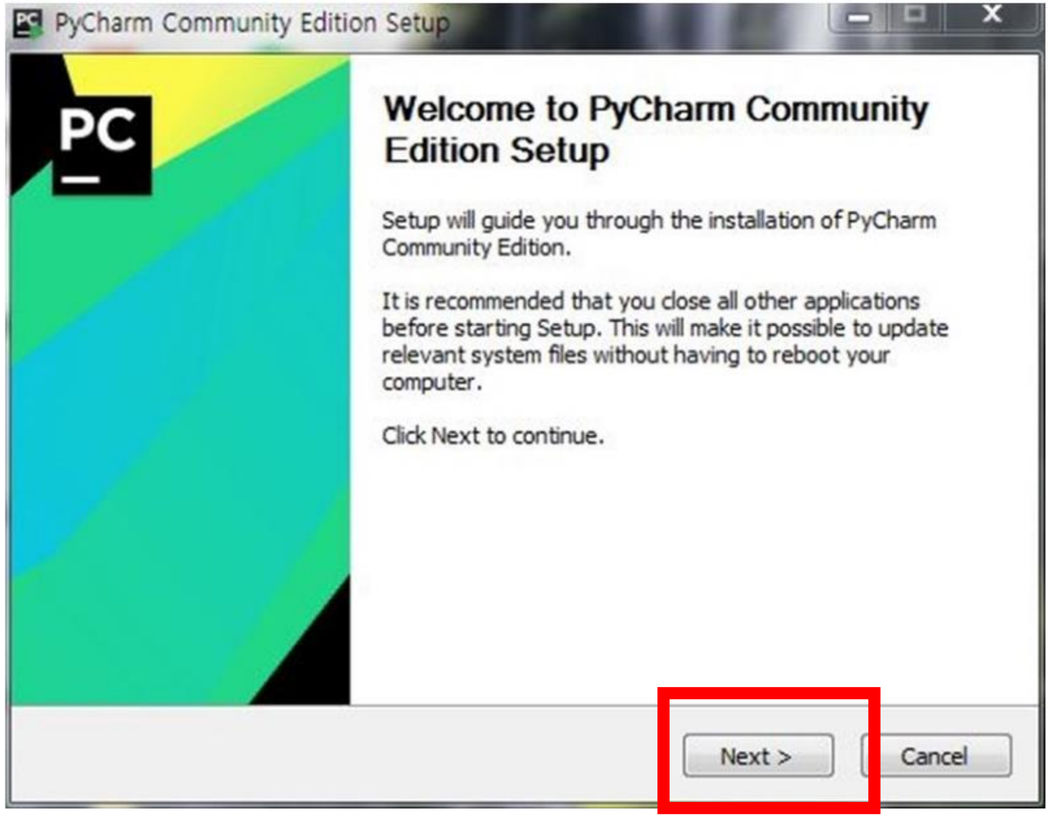
PyCharm 설치 및 세팅

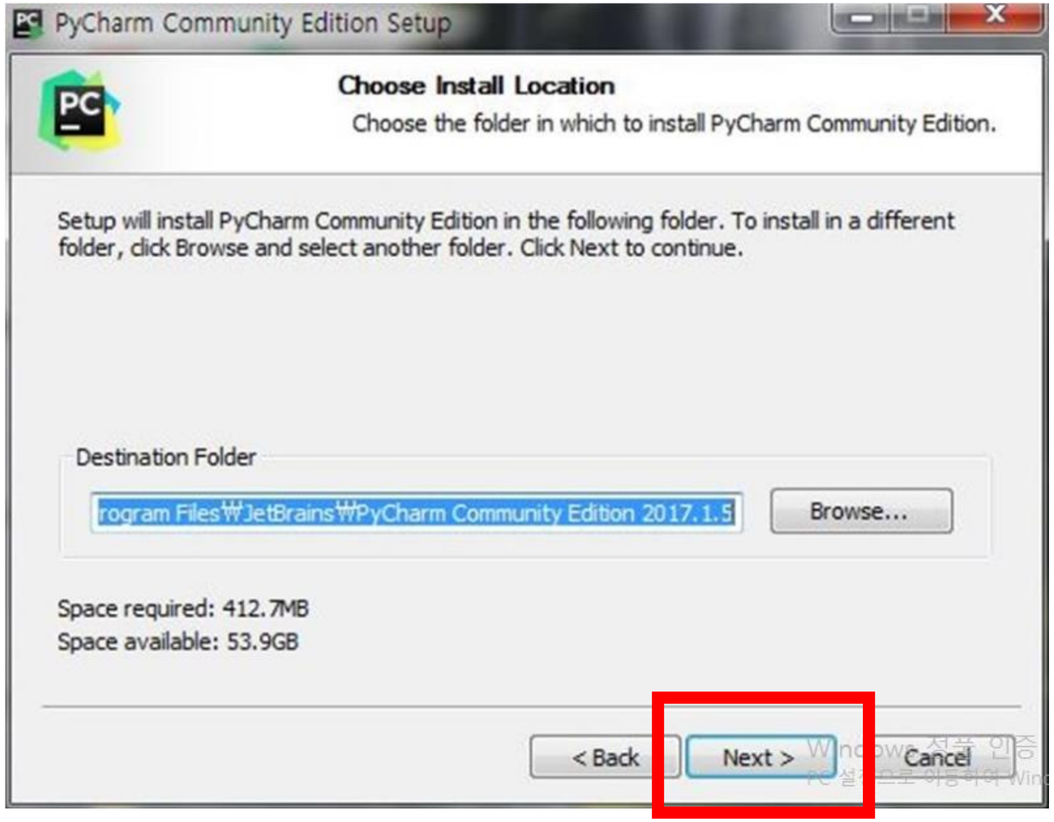
- https://www.jetbrains.com/pycharm/download/#section=windows 접속
- 아래 그림과 같이 다운로드



PyCharm 설치 및 세팅

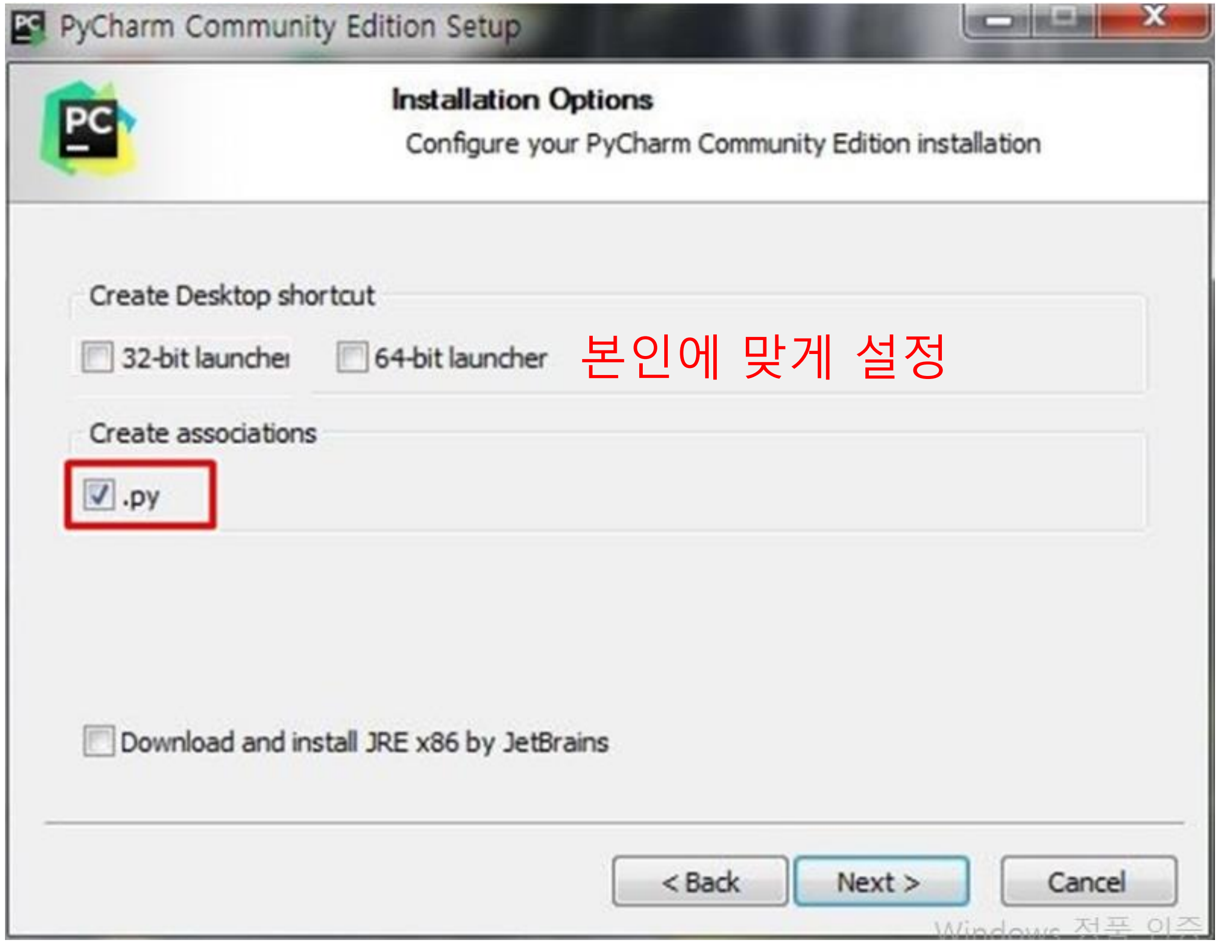
- 설치 파일 실행

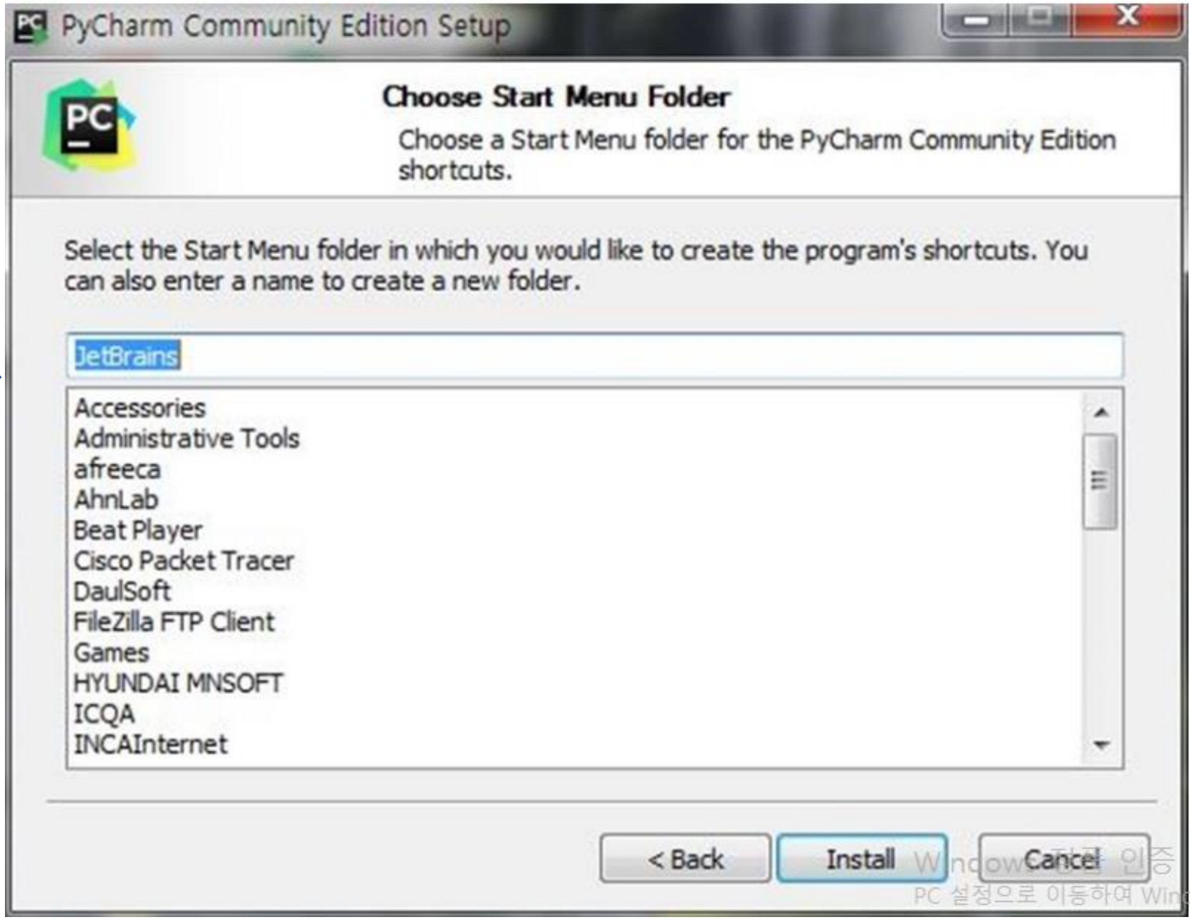




PyCharm 설치 및 세팅

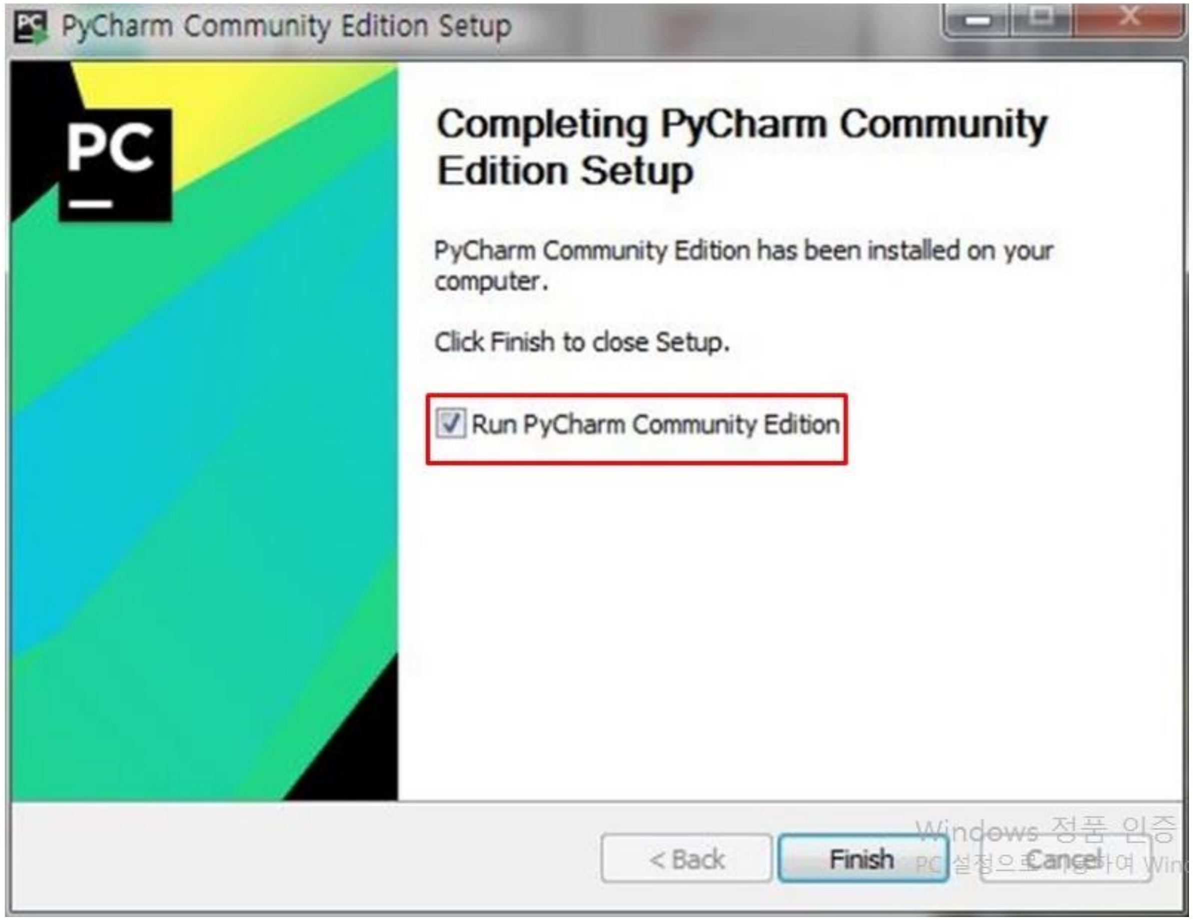
- 설치 파일 실행





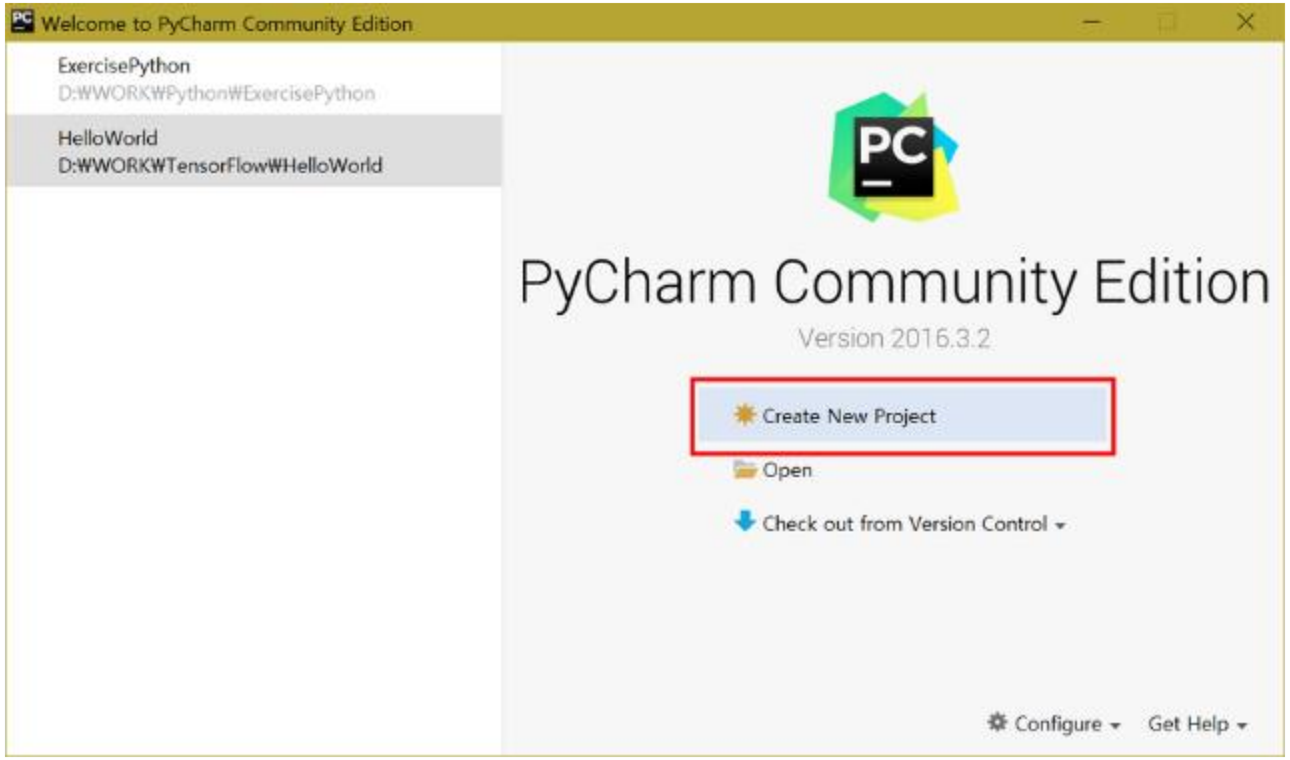
PyCharm 설치 및 세팅

- 설치 파일 실행



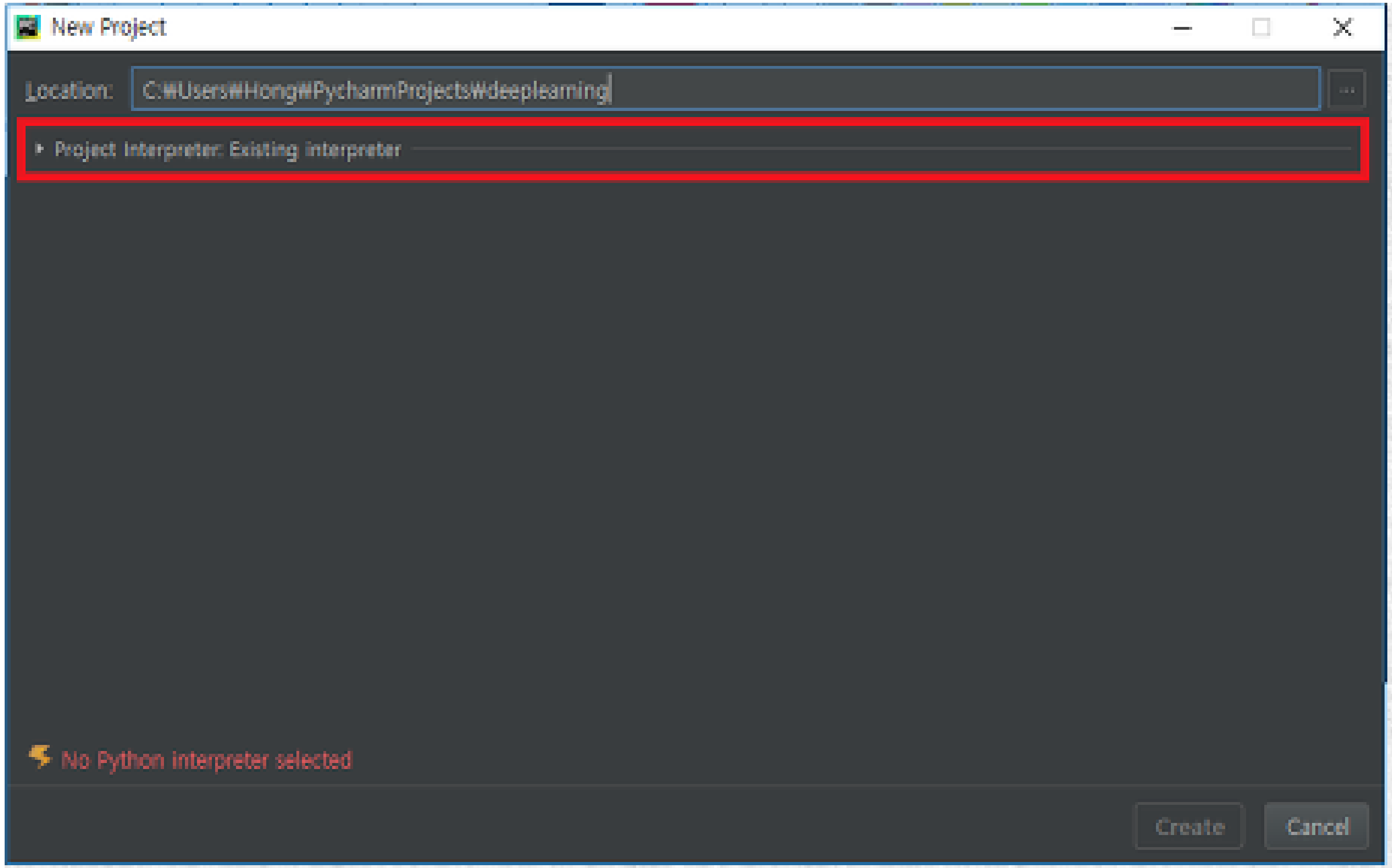
PyCharm 설치 및 세팅

- 세팅
  - 파일 실행 후 Create New Project 클릭



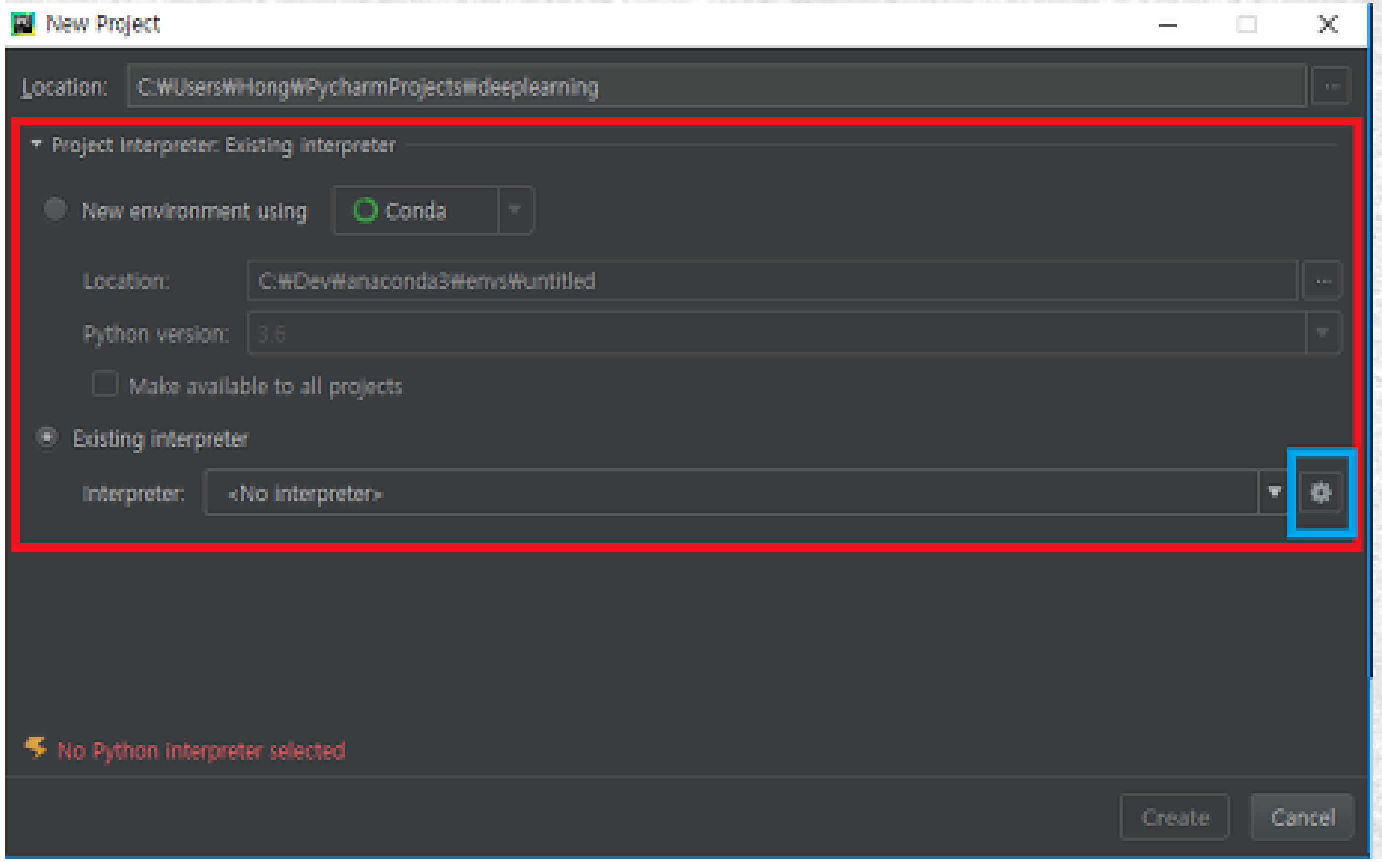
PyCharm 설치 및 세팅

- 세팅
  - 밑의 Project Interpreter Existing interpreter 클릭



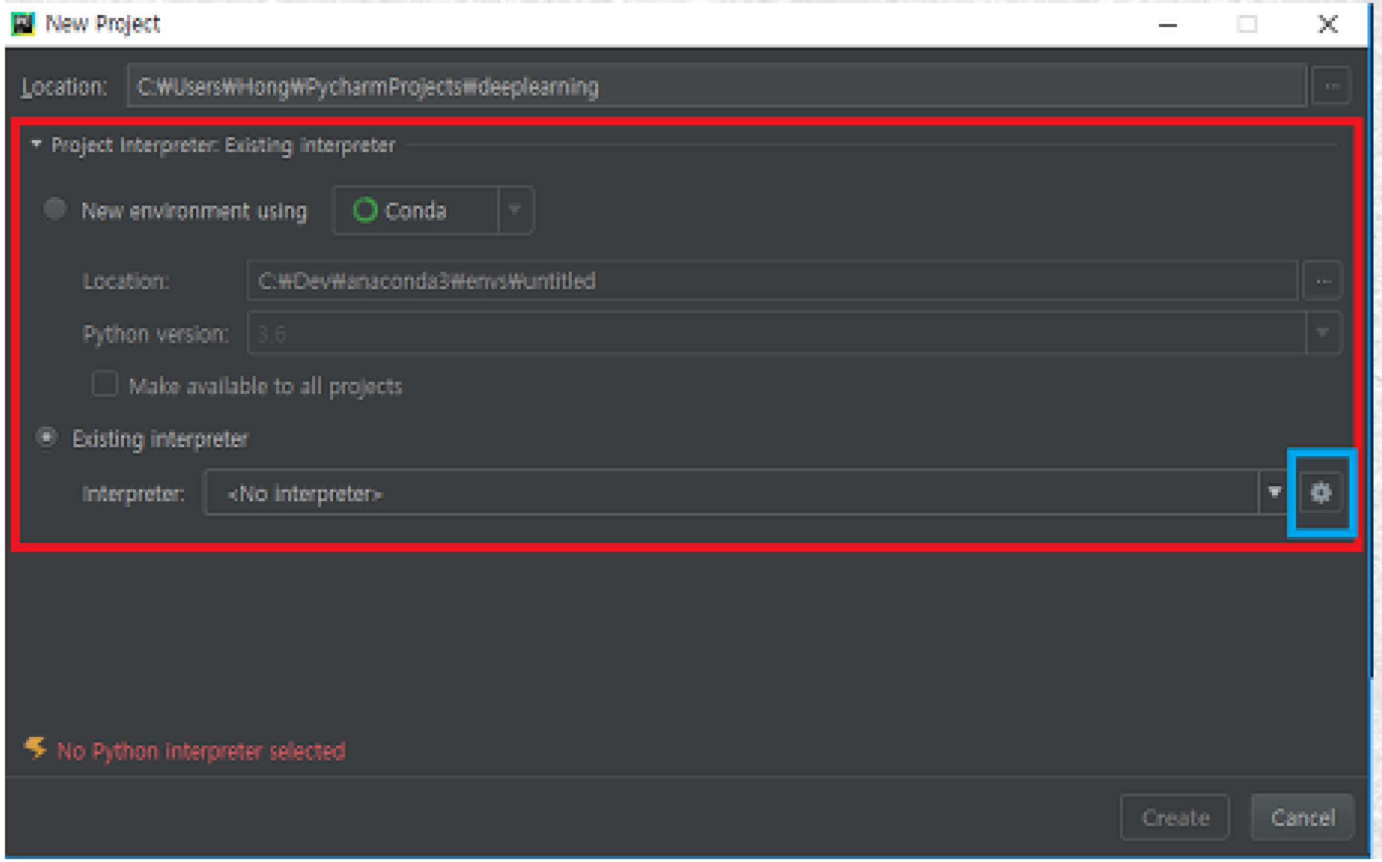
PyCharm 설치 및 세팅

- 세팅
  - Existing Interperter 클릭 후
  - Anaconda3 설치 폴더 안의 python.exe 선택 후 ok



PyCharm 설치 및 세팅

- 세팅
  - Existing Interperter 클릭 후
  - Anaconda3 설치 폴더 안의 python.exe 선택 후 ok



Anaconda 가상환경에 PyTorch 설치하기

- 윈도우=> 시작에서 Anaconda Prompt 실행
- 다음 명령어를 입력하여 새 가상 환경 만들기

```
$ conda create -y -n pytorch ipykernel
```

- 만든 가상환경으로 들어가기

```
$ activate pytorch
```

- PyTorch 설치하기

```
(pytorch)$ conda install -y -c peterjc123 pytorch
```

Anaconda 가상환경에 PyTorch 설치하기

- Anaconda Prompt 에서 “activate pytorch” 입력

```
(base) C:₩Users₩sungju₩PycharmProjects₩untitled>activate pytorch
```

바뀌는지 확인

```
(pytorch) C:₩Users₩sungju₩PycharmProjects₩untitled>
```

- “python” 입력

```
(pytorch) C:₩Users₩sungju₩PycharmProjects₩untitled>python
```

- “import torch” 입력 후 오류가 뜨지 않으면 설치 성공

```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)] :: An
conda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
```

Anaconda 가상환경에 PyTorch 설치하기

- “x=torch.rand(5)”를 입력하고 “print(x)”를 입력해서 x가 출력되면 성공

```
>>> import torch
>>> x=torch.rand(5)
>>> print(x)
tensor([0.7948, 0.1786, 0.9381, 0.3884, 0.4286])
```

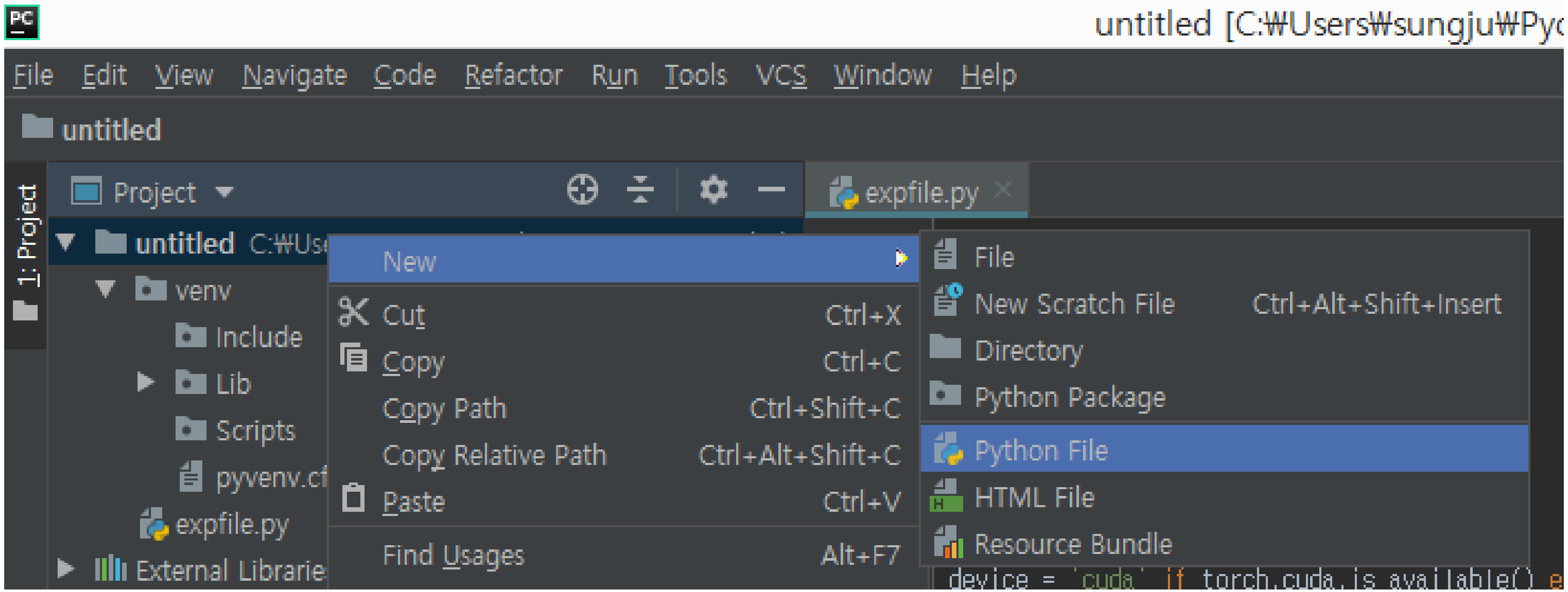
Pytorch 실행

- Anaconda 가상환경

- Pycharm => 편집기로 사용하기 위해

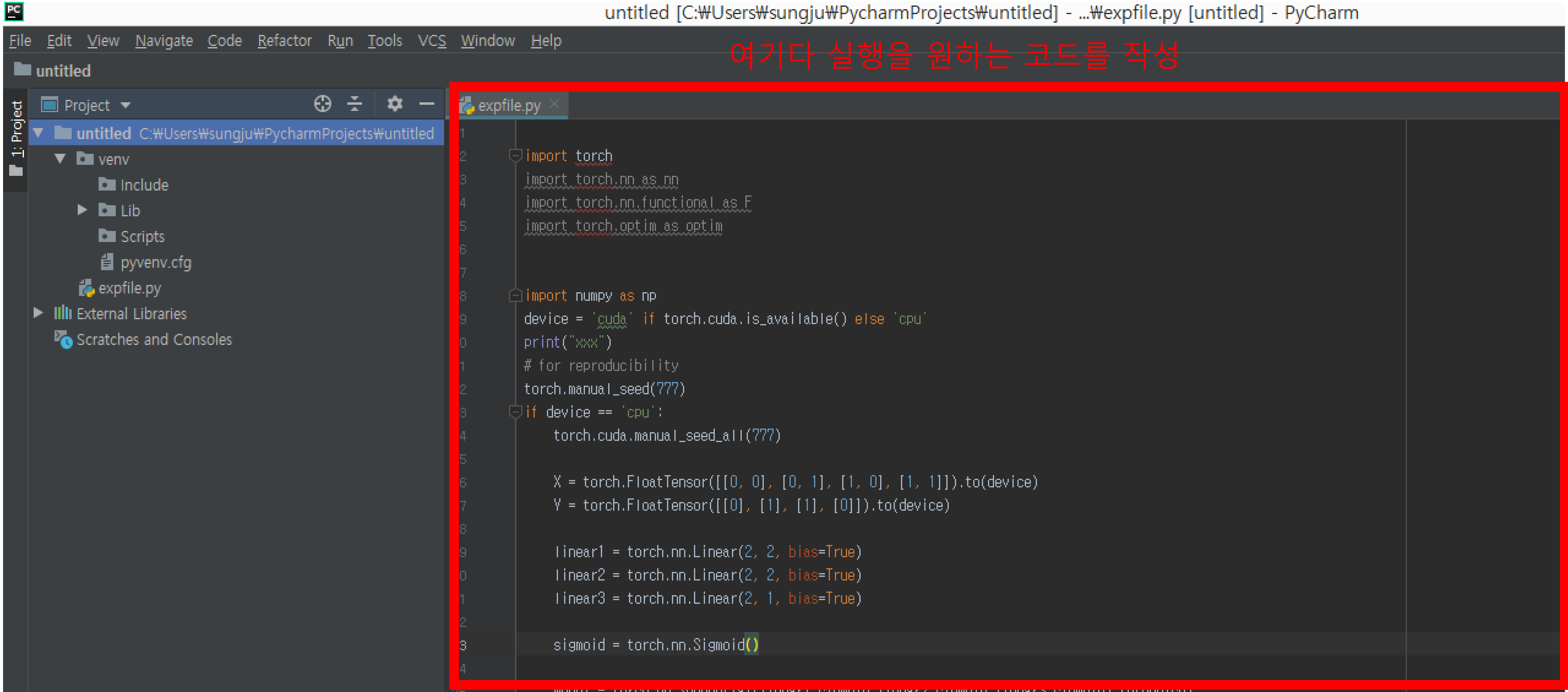
Pytorch 실행

- Pycharm 실행 후 Project에 오른쪽 버튼 눌러서 new=>Python File 클릭하여 .py 파일 생성



Pytorch 실행

코드 작성 하고 저장



Pytorch 실행

Anaconda Prompt 에서 cd 명령어로 방금 저장한 .py 파일이 있는 폴더 경로에 가서 명령어로 “python (작성한 파일명).py”를 입력하면 해당 .py의 파일이 실행이 됨

```
(pytorch) C:₩Users₩sungju₩PycharmProjects₩untitled>python expfile.py
```

현재 가상환경 폴더 경로 명령어

Pytorch 실행

Pycharm에 다음과 같이 작성한 후 앞의 방법을 이용하여 파일을 실행시켜 보자

expfile.py

```python

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


x=torch.rand(5)
print(x)

```

다음과 같이 뜨면 성공

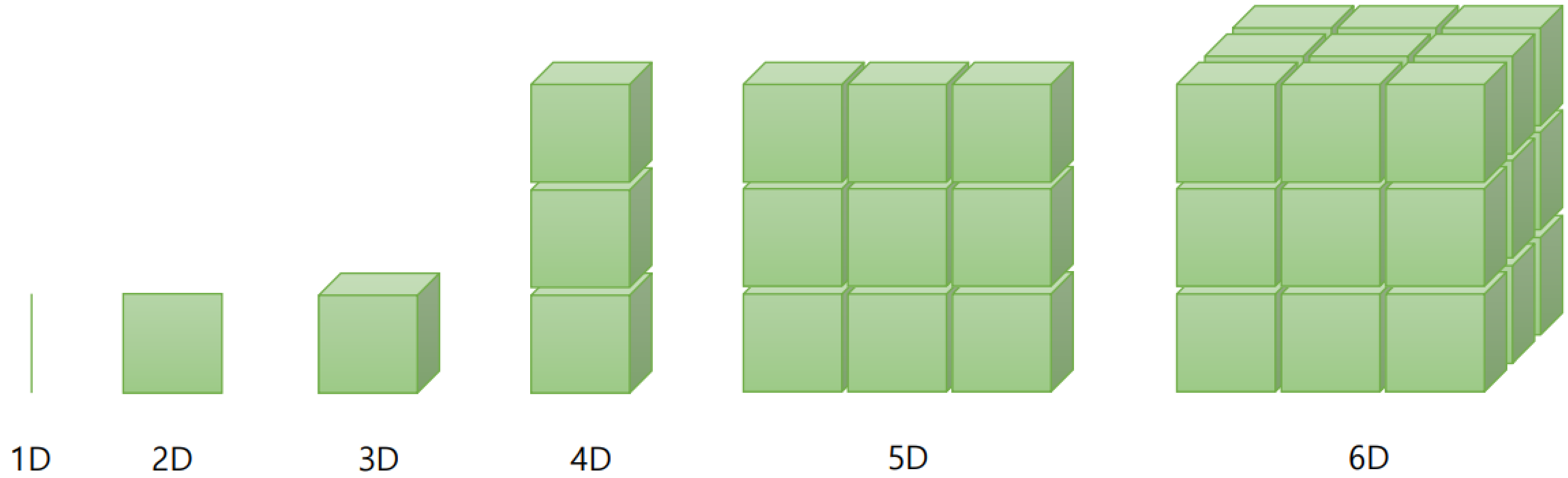
```
(pytorch) C:\Users\sungju\PycharmProjects\untitled>python expfile.py
tensor([0.1970, 0.5931, 0.5114, 0.1615, 0.5637])
```

Tensor Manipulation

Pytorch Basic Tensor

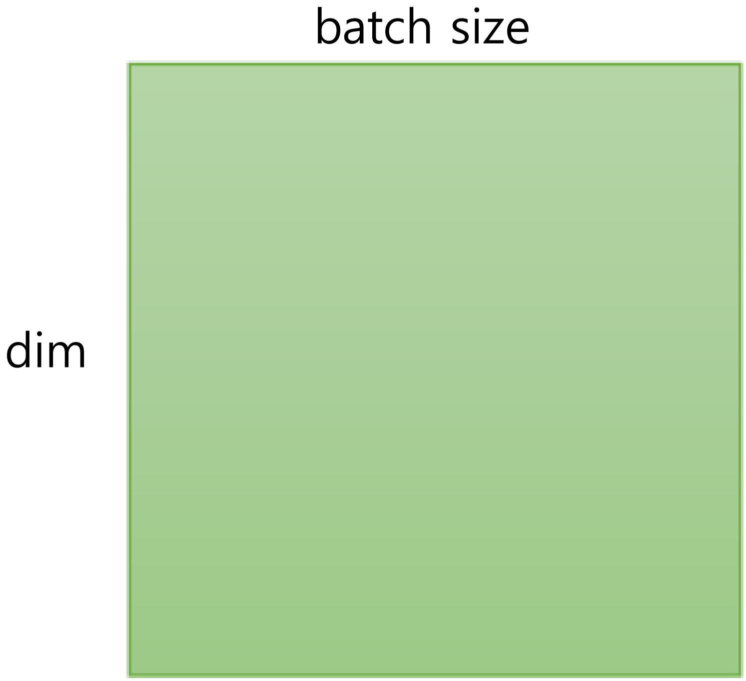
- Vector, Matrix and Tensor
- NumPy
- Pytorch Tensor Allocation
- Matrix Multiplication
- Other Basic Ops

Vector, Matrix and Tensor



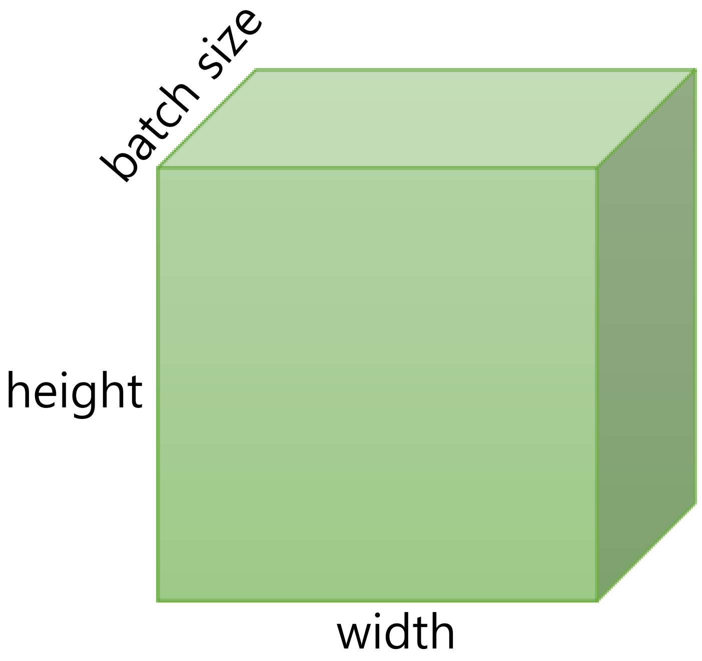
PyTorch Tensor Shape Convention

- 2D Tensor (Typical Simple Setting)
  - $|t| = (batch\ size, dim)$



PyTorch Tensor Shape Convention

- 3D Tensor (Typical Computer Vision)
  - $|t| = (batch\ size, width, height)$



Import

Run `pip install -r requirements.txt` in terminal to install all required Python packages.

```
import numpy as np
import torch
```

NumPy

1D Array with NumPy

```
t = np.array([0., 1., 2., 3., 4., 5., 6.])
print(t)
```

```
[ 0.  1.  2.  3.  4.  5.  6.]
```

```
print('Rank  of t: ', t.ndim)
print('Shape of t: ', t.shape)
```

```
Rank  of t:  1
Shape of t:  (7,)
```

```
print('t[0] t[1] t[-1] = ', t[0], t[1], t[-1]) # Element
print('t[2:5] t[4:-1]  = ', t[2:5], t[4:-1])   # Slicing
print('t[:2] t[3:]     = ', t[:2], t[3:])      # Slicing
```

```
t[0] t[1] t[-1] =  0.0 1.0 6.0
t[2:5] t[4:-1]  =  [ 2.  3.  4.] [ 4.  5.]
t[:2] t[3:]     =  [ 0.  1.] [ 3.  4.  5.  6.]
```

NumPy

2D Array with NumPy

```
t = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.], [10., 11., 12.]])
print(t)
```

```
[[  1.   2.   3.]
 [  4.   5.   6.]
 [  7.   8.   9.]
 [ 10.  11.  12.]]
```

```
print('Rank  of t: ', t.ndim)
print('Shape of t: ', t.shape)
```

```
Rank  of t:  2
Shape of t:  (4, 3)
```

PyTorch Tensor

1D Array with PyTorch

```
t = torch.FloatTensor([0., 1., 2., 3., 4., 5., 6.])
print(t)
```

```
tensor([0., 1., 2., 3., 4., 5., 6.])
```

```
print(t.dim())  # rank
print(t.shape)  # shape
print(t.size()) # shape
print(t[0], t[1], t[-1])  # Element
print(t[2:5], t[4:-1])    # Slicing
print(t[:2], t[3:])       # Slicing
```

```
1
torch.Size([7])
torch.Size([7])
tensor(0.) tensor(1.) tensor(6.)
tensor([2., 3., 4.]) tensor([4., 5.])
tensor([0., 1.]) tensor([3., 4., 5., 6.])
```

PyTorch Tensor

2D Array with PyTorch

```
t = torch.FloatTensor([[1., 2., 3.],
                       [4., 5., 6.],
                       [7., 8., 9.],
                       [10., 11., 12.]
                      ])
print(t)
```

```
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.],
        [10., 11., 12.]])
```

```
print(t.dim())  # rank
print(t.size()) # shape
print(t[:, 1])
print(t[:, 1].size())
print(t[:, :-1])
```

```
2
torch.Size([4, 3])
tensor([ 2.,  5.,  8., 11.])
torch.Size([4])
tensor([[ 1.,  2.],
        [ 4.,  5.],
        [ 7.,  8.],
        [10., 11.]])
```

PyTorch Tensor Operation (addition)

```
# Same shape
m1 = torch.FloatTensor([[3, 3]])
m2 = torch.FloatTensor([[2, 2]])
print(m1 + m2)
```

백터 + 백터 일 경우

```
tensor([[5., 5.]])
```

```
# Vector + scalar
m1 = torch.FloatTensor([[1, 2]])
m2 = torch.FloatTensor([3]) # 3 -> [[3, 3]]
print(m1 + m2)
```

백터 + 상수 일 경우
백터에 상수 값이 동일하게 더 해짐

```
tensor([[4., 5.]])
```

```
# 2 x 1 Vector + 1 x 2 Vector
m1 = torch.FloatTensor([[1, 2]])
m2 = torch.FloatTensor([[3], [4]])
print(m1 + m2)
```

서로 차원을 늘려서 차원을 맞춤

$$\begin{bmatrix}1\\2\end{bmatrix}+[3,4]=\begin{bmatrix}1&1\\2&2\end{bmatrix}+\begin{bmatrix}3&4\\3&4\end{bmatrix}=\begin{bmatrix}4&5\\5&6\end{bmatrix}$$

```
tensor([[4., 5.],
        [5., 6.]])
```

Multiplication vs Matrix Multiplication

```
print()
print('-------------')
print('Mul vs Matmul')
print('-------------')
m1 = torch.FloatTensor([[1, 2], [3, 4]])
m2 = torch.FloatTensor([[1], [2]])
print('Shape of Matrix 1: ', m1.shape) # 2 x 2
print('Shape of Matrix 2: ', m2.shape) # 2 x 1
print(m1.matmul(m2)) # 2 x 1

m1 = torch.FloatTensor([[1, 2], [3, 4]])
m2 = torch.FloatTensor([[1], [2]])
print('Shape of Matrix 1: ', m1.shape) # 2 x 2
print('Shape of Matrix 2: ', m2.shape) # 2 x 1
print(m1 * m2) # 2 x 2
print(m1.mul(m2))
```

```
-------------
Mul vs Matmul
-------------
Shape of Matrix 1:  torch.Size([2, 2])
Shape of Matrix 2:  torch.Size([2, 1])
tensor([[ 5.],
        [11.]])
Shape of Matrix 1:  torch.Size([2, 2])
Shape of Matrix 2:  torch.Size([2, 1])
tensor([[1., 2.],
        [6., 8.]])
tensor([[1., 2.],
        [6., 8.]])
```

Mean

```
t = torch.FloatTensor([1, 2])
print(t.mean())
```

```
tensor(1.5000)
```

```
# Can't use mean() on integers
t = torch.LongTensor([1, 2])
try:
    print(t.mean())
except Exception as exc:
    print(exc)
```

```
Can only calculate the mean of floating types. Got Long instead.
```

You can also use `t.mean` for higher rank tensors to get mean of all elements, or mean by particular dimension.

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
```

```
tensor([[1., 2.],
        [3., 4.]])
```

```
print(t.mean())
print(t.mean(dim=0))
print(t.mean(dim=1))
print(t.mean(dim=-1))
```

```
tensor(2.5000)
tensor([2., 3.])
tensor([1.5000, 3.5000])
tensor([1.5000, 3.5000])
```

Sum

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
```

```
tensor([[1., 2.],
        [3., 4.]])
```

```
print(t.sum())
print(t.sum(dim=0))
print(t.sum(dim=1))
print(t.sum(dim=-1))
```

```
tensor(10.)
tensor([4., 6.])
tensor([3., 7.])
tensor([3., 7.])
```

Max and Argmax

```python
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
```

```
tensor([[1., 2.],
        [3., 4.]])
```

The `max` operator returns one value if it is called without an argument.

```python
print(t.max()) # Returns one value: max
```

```
tensor(4.)
```

The `max` operator returns 2 values when called with dimension specified. The first value is the maximum value, and the second value is the argmax: the index of the element with maximum value.

```python
print(t.max(dim=0)) # Returns two values: max and argmax
print('Max: ', t.max(dim=0)[0])
print('Argmax: ', t.max(dim=0)[1])
```

```
(tensor([3., 4.]), tensor([1, 1]))
Max:  tensor([3., 4.])
Argmax:  tensor([1, 1])
```

```python
print(t.max(dim=1))
print(t.max(dim=-1))
```

```
(tensor([2., 4.]), tensor([1, 1]))
(tensor([2., 4.]), tensor([1, 1]))
```

View (Reshape)

```
t = np.array([[[0, 1, 2],
               [3, 4, 5]],

              [[6, 7, 8],
               [9, 10, 11]]])
ft = torch.FloatTensor(t)
print(ft.shape)
```

```
torch.Size([2, 2, 3])
```

```
print(ft.view([-1, 3]))
print(ft.view([-1, 3]).shape)
```

```
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 6.,  7.,  8.],
        [ 9., 10., 11.]])
torch.Size([4, 3])
```

```
print(ft.view([-1, 1, 3]))
print(ft.view([-1, 1, 3]).shape)
```

```
tensor([[[ 0.,  1.,  2.]],

        [[ 3.,  4.,  5.]],

        [[ 6.,  7.,  8.]],

        [[ 9., 10., 11.]]])
torch.Size([4, 1, 3])
```

Squeeze

```
ft = torch.FloatTensor([[0], [1], [2]])
print(ft)
print(ft.shape)
```

```
tensor([[0.],
        [1.],
        [2.]])
torch.Size([3, 1])
```

```
print(ft.squeeze())
print(ft.squeeze().shape)
```

```
tensor([0., 1., 2.])
torch.Size([3])
```

Unsqueeze

```
ft = torch.Tensor([0, 1, 2])
print(ft.shape)
```

```
torch.Size([3])
```

```
print(ft.unsqueeze(0))
print(ft.unsqueeze(0).shape)
```

```
tensor([[0., 1., 2.]])
torch.Size([1, 3])
```

```
print(ft.view(1, -1))
print(ft.view(1, -1).shape)
```

```
tensor([[0., 1., 2.]])
torch.Size([1, 3])
```

```
print(ft.unsqueeze(1))
print(ft.unsqueeze(1).shape)
```

```
tensor([[0.],
        [1.],
        [2.]])
torch.Size([3, 1])
```

```
print(ft.unsqueeze(-1))
print(ft.unsqueeze(-1).shape)
```

```
tensor([[0.],
        [1.],
        [2.]])
torch.Size([3, 1])
```

Type Casting

```
lt = torch.LongTensor([1, 2, 3, 4])
print(lt)
```

```
tensor([1, 2, 3, 4])
```

```
print(lt.float())
```

```
tensor([1., 2., 3., 4.])
```

```
bt = torch.ByteTensor([True, False, False, True])
print(bt)
```

```
tensor([1, 0, 0, 1], dtype=torch.uint8)
```

```
print(bt.long())
print(bt.float())
```

```
tensor([1, 0, 0, 1])
tensor([1., 0., 0., 1.])
```

Concatenate

```
x = torch.FloatTensor([[1, 2], [3, 4]])
y = torch.FloatTensor([[5, 6], [7, 8]])
```

```
print(torch.cat([x, y], dim=0))
print(torch.cat([x, y], dim=1))
```

```
tensor([[1., 2.],
        [3., 4.],
        [5., 6.],
        [7., 8.]])
tensor([[1., 2., 5., 6.],
        [3., 4., 7., 8.]])
```

Stacking

```
x = torch.FloatTensor([1, 4])
y = torch.FloatTensor([2, 5])
z = torch.FloatTensor([3, 6])
```

```
print(torch.stack([x, y, z]))
print(torch.stack([x, y, z], dim=1))
```

```
tensor([[1., 4.],
        [2., 5.],
        [3., 6.]])
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

```
print(torch.cat([x.unsqueeze(0), y.unsqueeze(0), z.unsqueeze(0)], dim=0))
```

```
tensor([[1., 4.],
        [2., 5.],
        [3., 6.]])
```

Ones and Zeros

```
x = torch.FloatTensor([[0, 1, 2], [2, 1, 0]])
print(x)
```

```
tensor([[0., 1., 2.],
        [2., 1., 0.]])
```

```
print(torch.ones_like(x))
print(torch.zeros_like(x))
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

In-place Operation

```python
x = torch.FloatTensor([[1, 2], [3, 4]])
```

```python
print(x.mul(2.))
print(x)
print(x.mul_(2.))
print(x)
```

mul은 단순 출력

mul_은 x의 값도 변경시킴

```
tensor([[2., 4.],
        [6., 8.]])
tensor([[1., 2.],
        [3., 4.]])
tensor([[2., 4.],
        [6., 8.]])
tensor([[2., 4.],
        [6., 8.]])
```

Linear Regression

Pytorch Basic Tensor

- Date definition
- Hypothesis
- Compute loss
- Gradient descent

Data definition

- What would be the grade if I study 4 hours?

Hours (x)	Points (y)
1	2
2	4
3	6
4	?

학습 (Training data)

테스트 (Test data)

Data definition

```
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])
```

$$X_{\text{train}} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \qquad Y_{\text{train}} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}$$

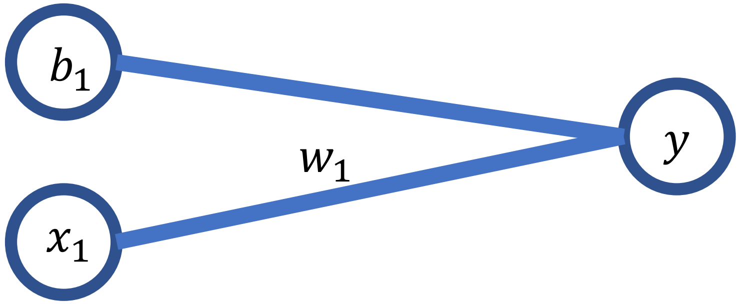
Hours (x)	Points (y)
1	2
2	4
3	6

학습 (Training data)

- 데이터 : torch.tensor로 할당
- 입력 : x_train
- 출력 : y_train
- 입출력은 x, y로 구분

Hypothesis = 가설 = 모델 설계

$$y = \underbrace{W}_{\text{Weight}} x + \underbrace{b}_{\text{Bias}}$$



$w = [w_1]$ b= $[b_1]$

y= $[y]$ $x = [x_1]$

Hypothesis = 가설 = 모델 설게

```python
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])
```

```python
W = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
hypothesis = x_train * W + b
```

- Weight 와 Bias 0 으로 초기화
  - 항상 출력 0이 되기 때문에 0으로 초기화 하면 안됨

- requires_grad = True
  - 학습이 될 변수라는 것을 명시

$$y = \underbrace{W}_{\text{Weight}} x + \underbrace{b}_{\text{Bias}}$$

Compute loss

- Loss = Mean Squared Error (MSE)

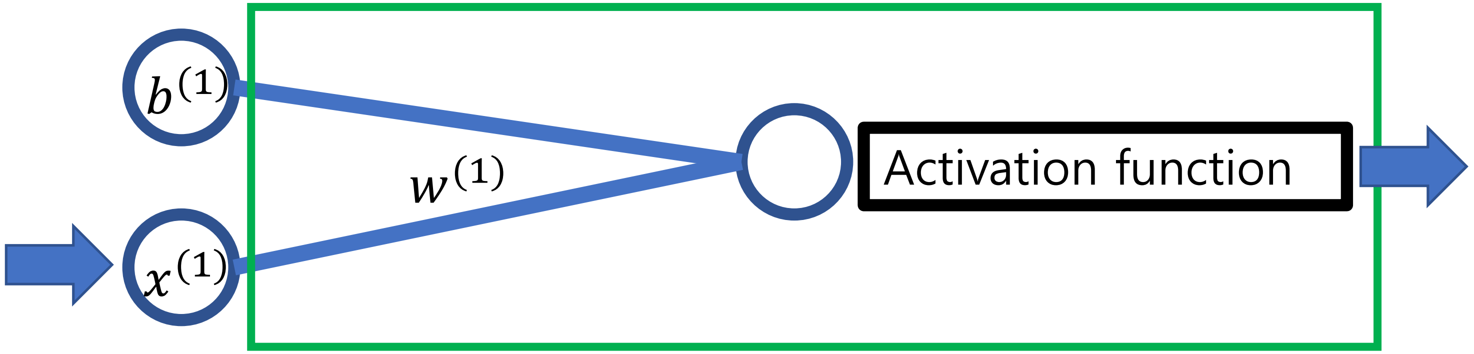
$$cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}\left(H(x^{(i)}) - y^{(i)}\right)^2$$

Mean

Prediction

Target

H( )



$H(x^{(i)})$

Gradient Descent = 학습 과정 단계

```
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])
```

```
W = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
hypothesis = x_train * W + b
```

```
cost = torch.mean((hypothesis - y_train) ** 2)
```

```
optimizer = optim.SGD([W, b], lr=0.01)


optimizer.zero_grad()
cost.backward()
optimizer.step()
```

- torch.optim 라이브러리 사용
  - [w, b] 는 학습할 tensor
  - lr=0.01은 learning ra

- 항상 같이 쓰는 3줄
  - zero_grad() 로 gradient 초기화
  - backward() 로 gradient 계산
  - step() 으로 학습 tensor들 개선

Full training code

데이터 정의
```python
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])
```
Hypothesis 초기화
```python
W = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```
Optimizer 정의
```python
optimizer = optim.SGD([W, b], lr=0.01)

nb_epochs = 1000
for epoch in range(1, nb_epochs + 1):
    hypothesis = x_train * W + b
    cost = torch.mean((hypothesis - y_train) ** 2)
```
Hypothesis 예측

Cost 계산

Optimizer 학습
```python
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
```

한번만

1. 데이터 정의
2. Hypothesis 초기화
3. Optimizer 정의

반복!

1. Hypothesis 예측
2. Cost 계산
3. Optimizer 로 학습

Deeper Look at Gradient Descent

- Hypothesis function review
- 사용할 모의 data 확인
- Cost function 이해
- Gradient descent 이론
- Gradient descent 구현
- Gradient descent 구현 (nn.optim)

Hypothesis (Linear regression)

$$H(x) = \underbrace{W}_{\text{Weight}}x + \underbrace{b}_{\text{Bias}}$$

```
W = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
hypothesis = x_train * W + b
```

Simpler Hypothesis Function

$$H(x) = Wx$$

Weight

No Bias!

```
W = torch.zeros(1, requires_grad=True)
# b = torch.zeros(1, requires_grad=True)
hypothesis = x_train * W
```

Data

- Input = Output!

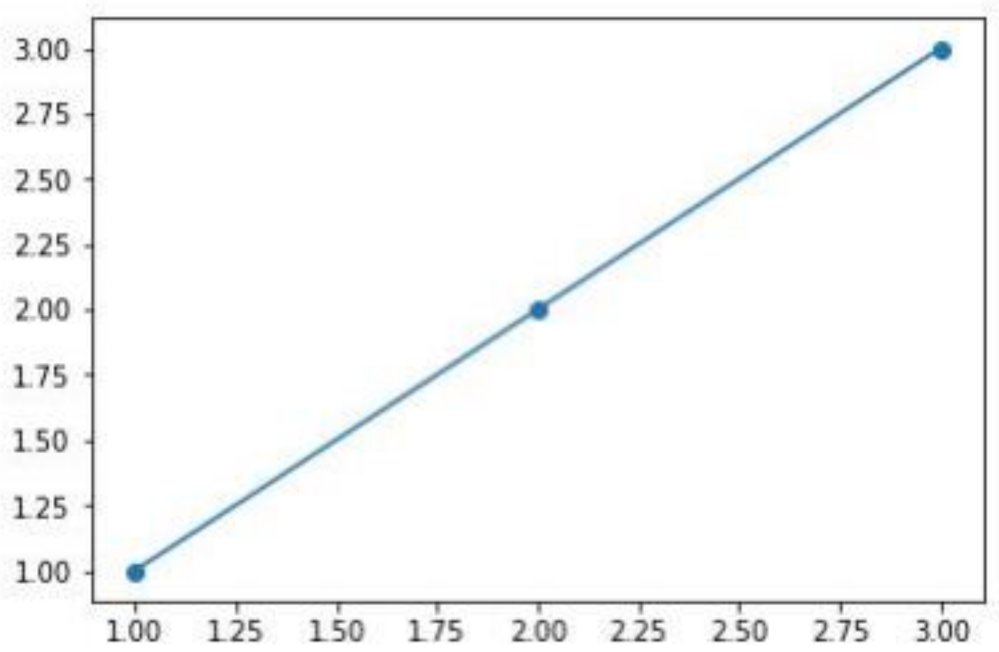
Hours (x)	Points (y)
1	1
2	2
3	3

학습 (Training data)

```
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[1], [2], [3]])
```

What is the best model

- $H(x) = x$ 가 정확한 모델
- $w = 1$이 최적값

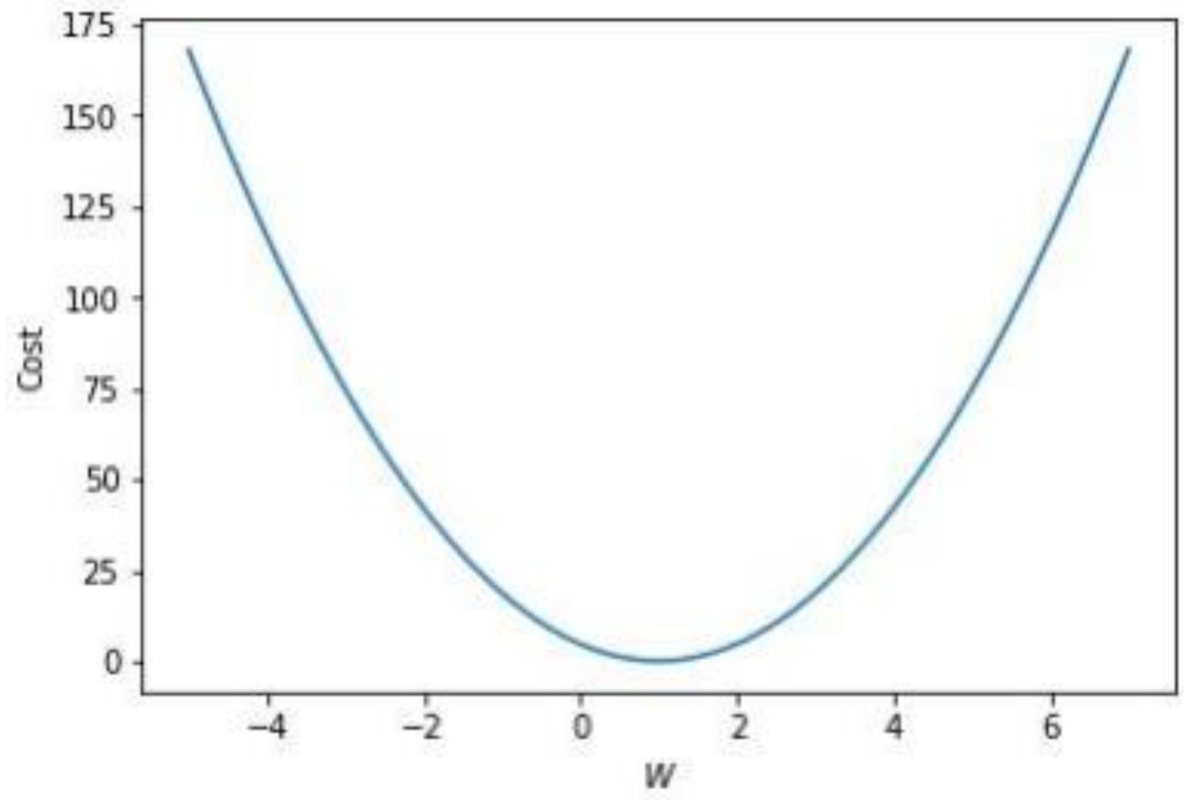


모델의 좋고 나쁨을 평가하는 방법?

Hours (x)	Points (y)
1	1
2	2
3	3

Cost function : Intuition

- $w = 1$ 일 때 $cost = 0$
- $w = 1$에서 멀어질수록 높아진다.



Cost function : MSE

- *Mean Squared Error (MSE)*

$$cost(W) = \frac{1}{m} \sum_{i=1}^{m} \left(H(x^{(i)}) - y^{(i)} \right)^2$$

Mean

Prediction

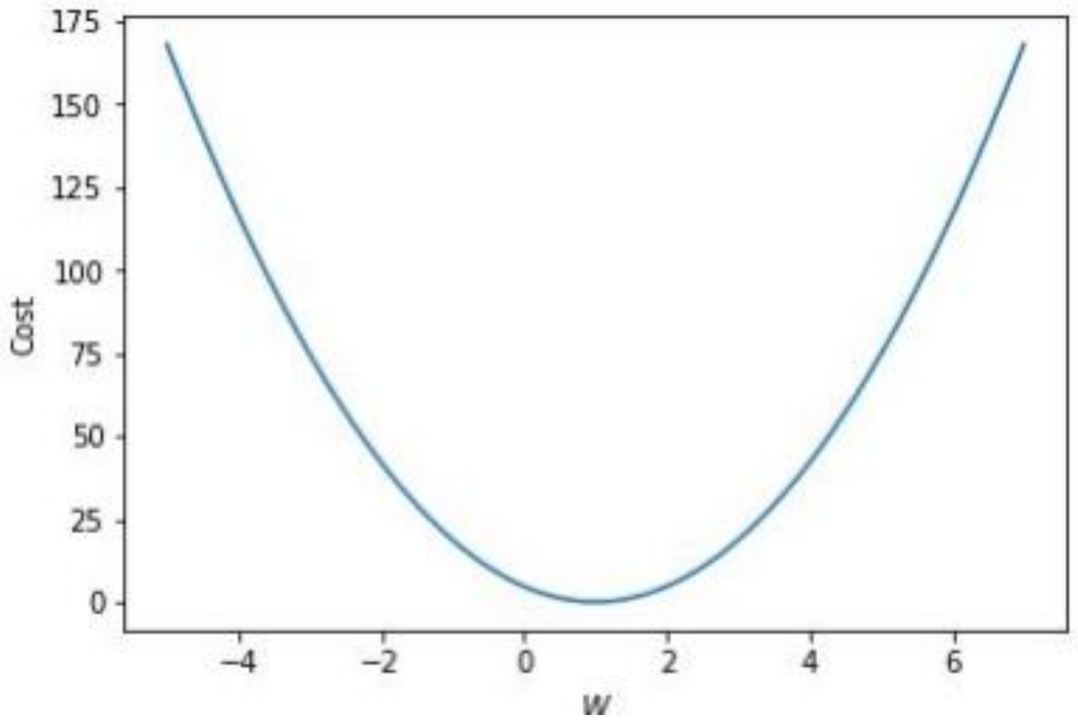
Target

```
cost = torch.mean((hypothesis - y_train) ** 2)
```

Gradient Descent : Intution

- 곡선을 내려가자
- 기울기가 클수록 더 멀리 갈 수 있음!
- “Gradient”를 계산하자

$$\frac{\partial cost}{\partial W} = \nabla W$$



Gradient Descent : The Math

$$cost(W) = \frac{1}{m}\sum_{i=1}^{m}\left(Wx^{(i)} - y^{(i)}\right)^2$$

$$\nabla W = \frac{\partial cost}{\partial W} = \frac{2}{m}\sum_{i=1}^{m}\left(Wx^{(i)} - y^{(i)}\right)x^{(i)}$$

$$W := W - \underbrace{\alpha}_{\text{Learning rate}} \underbrace{\nabla W}_{\text{Gradient}}$$

Gradient Descent : Code

$$\nabla W = \frac{\partial cost}{\partial W} = \frac{2}{m}\sum_{i=1}^{m}\left(Wx^{(i)} - y^{(i)}\right)x^{(i)}$$

$$W := W - \alpha \nabla W$$

```
gradient = 2 * torch.mean((W * x_train - y_train) * x_train)
lr = 0.1
W -= lr * gradient
```

Full Code

```
# 데이터
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[1], [2], [3]])
# 모델 초기화
W = torch.zeros(1)
# learning rate 설정
lr = 0.1

nb_epochs = 10
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = x_train * W

    # cost gradient 계산
    cost = torch.mean((hypothesis - y_train) ** 2)
    gradient = torch.sum((W * x_train - y_train) * x_train)

    print('Epoch {:4d}/{} W: {:.3f}, Cost: {:.6f}'.format(
        epoch, nb_epochs, W.item(), cost.item()
    ))

    # cost gradient로 H(x) 개선
    W -= lr * gradient
```

- Epoch : 데이터로 학습한 횟수
- 학습하면서 점점
  - 1에 수렴하는 W
  - 줄어드는 Cost

- 결과로 직접 확인하기!

Full Code

```python
# 데이터
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[1], [2], [3]])
# 모델 초기화
W = torch.zeros(1, requires_grad=True)
# optimizer 설정
optimizer = optim.SGD([W], lr=0.15)

nb_epochs = 10
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = x_train * W

    # cost 계산
    cost = torch.mean((hypothesis - y_train) ** 2)

    print('Epoch {:4d}/{} W: {:.3f} Cost: {:.6f}'.format(
        epoch, nb_epochs, W.item(), cost.item()
    ))

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
```

- Epoch : 데이터로 학습한 횟수
- 학습하면서 점점
  - 1에 수렴하는 W
  - 줄어드는 Cost

- 앞의 경우와 같은 지 결과로 직접 확인하기!

Gradient Descent with torch.optim

- torch.optim 으로도 gradient descent 를 할 수 있음
  - 시작할 때 Optimizer 정의
  - Optimizer.zero_grad()로 gradient를 0으로 초기화
  - cost.backward()로 gradient 계산
  - optimizer.step() 으로 gradient descent (w 수정 단계)

```
# optimizer 설정
optimizer = optim.SGD([W], lr=0.15)

# cost로 H(x) 개선
optimizer.zero_grad()
cost.backward()
optimizer.step()
```

Multivariate Linear Regression

(입력이 여러 개일 때)

Multivariate Linear Regression

- Simple Linear Regression 복습
- Multivariate Linear Regression 이론
- Naïve Data Representation
- Matirx Data Representation
- Multivariate Linear Regression
- nn.Module 소개
- F.mse_lose 소개

Simple Linear Regression

입력이 1개

$$H(x) = Wx + b$$

Multivariate Linear Regression

입력이 여러개

$$H(x) = Wx + b$$

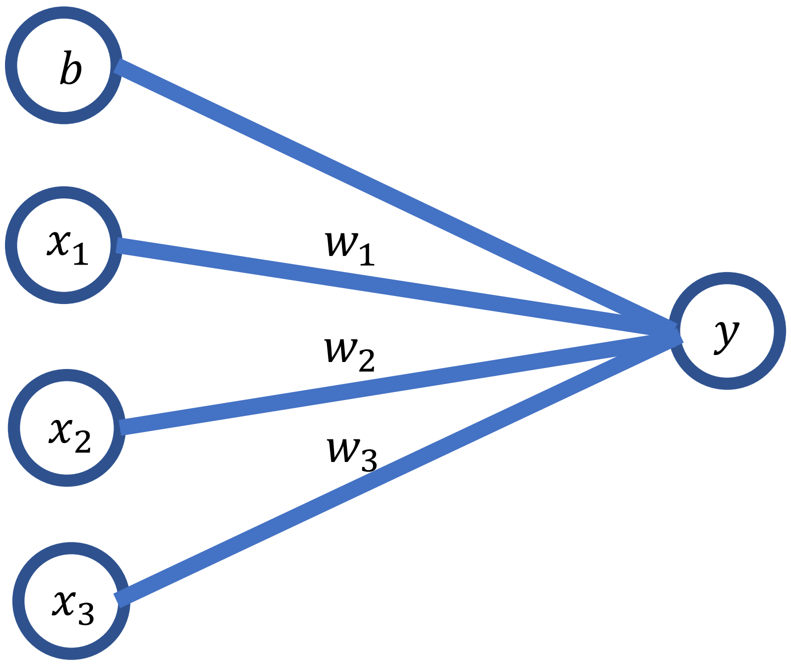
Data

Quiz 1 (x1)	Quiz 2 (x2)	Quiz 3 (x3)	Final (y)
73	80	75	152
93	88	93	185
89	91	80	180
96	98	100	196
73	66	70	142

```
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])
```

Hypothesis Function

$$w = [w_1, w_2, w_3] \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



$$H(x) = Wx + b$$

x 라는 vector 와 W 라는 matrix의 곱

$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$H(x) = w_1x_1 + w_2x_2 + w_3x_3 + b$$

입력변수가 3개라면 weight 도 3개!

Hypothesis Function : Naive

- 단순한 hypothesis 정의
- X 길이가 1000의 vector 경우

```
# H(x) 계산
hypothesis = x1_train * w1 + x2_train * w2 + x3_train * w3 + b
```

$$H(x) = w_1x_1 + w_2x_2 + w_3x_3 + b$$

Hypothesis Function : Matrix

```python
# H(x) 계산
hypothesis = x_train.matmul(W) + b # or .mm or @
```

- matmul() 로 한번에 계산
- 간결하며
- X의 길이가 바뀌어도 코드는 변함없음
- 속도도 더 빠름

$$H(x) = Wx + b$$

Cost function : MSE

- 기존 Simple Linear Regression 과 동일한 공식

$$\nabla W = \frac{\partial cost}{\partial W} = \frac{2}{m}\sum_{i=1}^{m}\left(Wx^{(i)} - y^{(i)}\right)x^{(i)}$$

$$W := W - \alpha \nabla W$$

```
# optimizer 설정
optimizer = optim.SGD([W, b], lr=1e-5)

# optimizer 사용법
optimizer.zero_grad()
cost.backward()
optimizer.step()
```

Full Code with torch.optim (1)

```
# 데이터
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])

# 모델 초기화
W = torch.zeros((3, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)

# optimizer 설정
optimizer = optim.SGD([W, b], lr=1e-5)
```

1. 데이터 정의

2. 모델 정의

3. optimizer 정의

Full Code with torch.optim (2)

```
nb_epochs = 20
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = x_train.matmul(W) + b # or .mm or @

    # cost 계산
    cost = torch.mean((hypothesis - y_train) ** 2)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    print('Epoch {:4d}/{} hypothesis: {} Cost: {:.6f}'.format(
        epoch, nb_epochs, hypothesis.squeeze().detach(),
        cost.item()
    ))
```

4. Hypothesis 계산

5. Cost 계산 (MSE)

6. Gradient descent

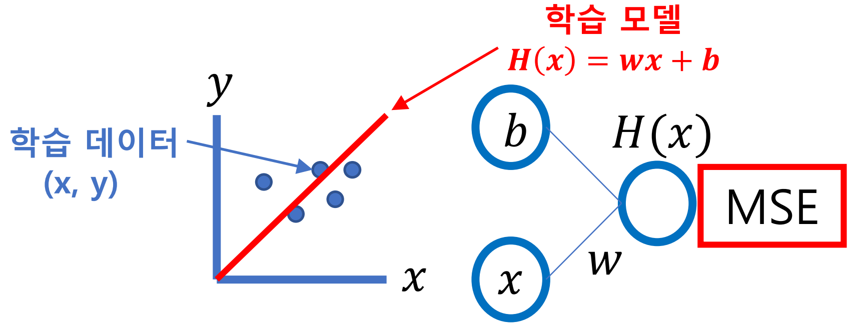
Results

- 점점 작아지는 Cost
- 점점 y 에 가까워지는 H(x)
- Learning rate에 따라 발산할 수 도 있음

Final (y)
152
185
180
196
142

Linear Regression => 값 추정!!

Simple Linear Regression



No sigmoid!

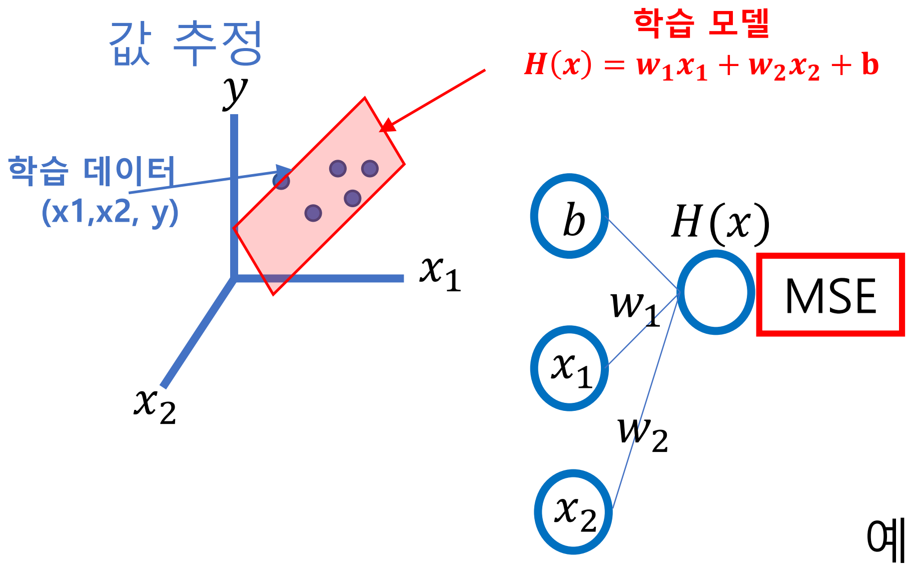
MSE (Mean Squared Error)

$$\frac{1}{m}\sum_{i=1}^{m}\left(H\left(x^{(i)}\right) - y\right)^2$$

예시) 입력 : 배달거리, 출력 : 배달 시간

1. 값 추정이므로 범위제한 x (no sigmoid!)
2. 값 추정이므로 MSE로 임의의 실수 값을 추정해야 한다. (MSE)

Multivariate Linear Regression



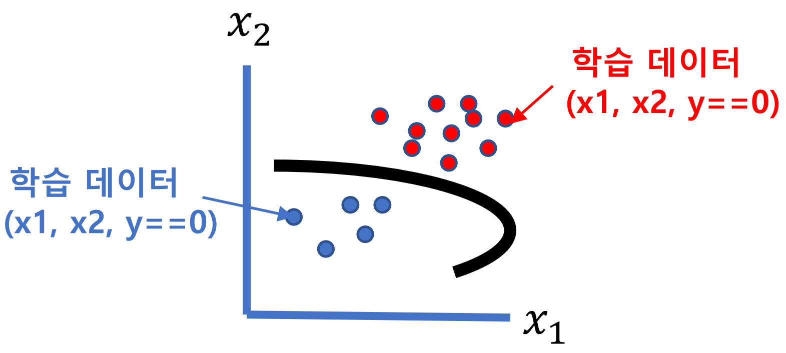
No sigmoid!

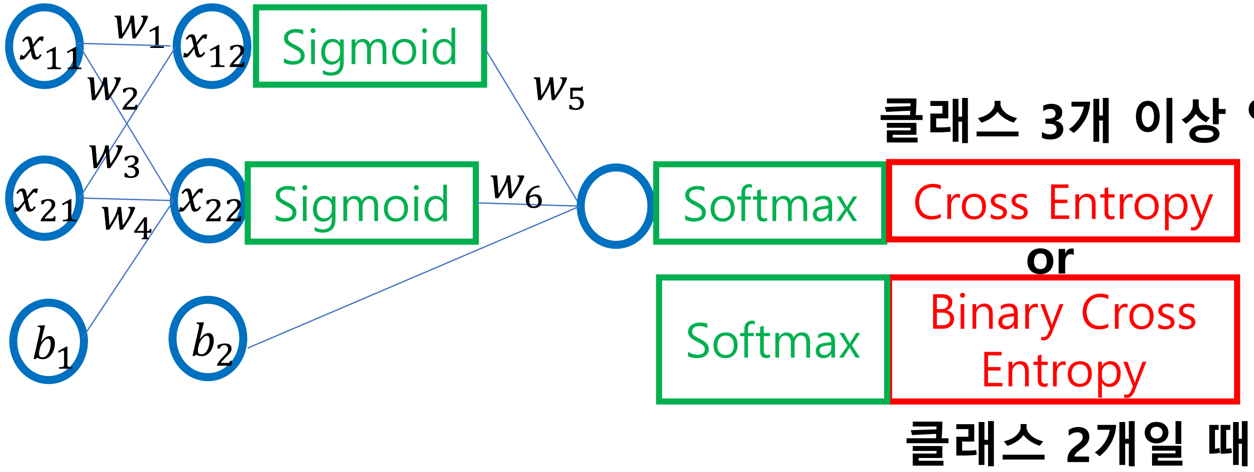
MSE (Mean Squared Error)

$$\frac{1}{m}\sum_{i=1}^{m}\left(H\left(x^{(i)}\right) - y\right)^2$$

예시) 입력 : Quiz1, Quiz2, Quiz3, 출력 :Final(합산 점수)

Logistic Regression=> 값 추정, 구분(Classification)!!
=> 값 추정은 여기서 다루지 않는다. 구분(Classification)위주로 다룸





F.cross_entropy
(추정, 정답)
Softmax와
Cross entropy
가 모두 들어있음

F.binary_cross_entropy(추정, 정답)
함수로 softmax와 binary cross
entropy가 모두 들어있음

중간은 Sigmoid!=> 다양한 모양

끝에는 Softmax=> 확률 형태

Cross_entropy => 구분(Classification)

$$L = \frac{1}{N}\sum -ylog(H(x)) \Leftarrow L = \frac{1}{m}\sum_{i=1}^{m}\left(H\left(x^{(i)}\right) - y\right)^2$$

1. Sigmoid를 통해 다양한 모양을 표현 할 수 있음
2. Softmax의 경우 sigmoid와 달리 모든 합이 1이며, 확률형태로 나타낼 수 있는 장점이 있음
3. MSE를 쓰지 않는 이유는 원래 그래프를 제곱하면 그래프가 복잡 해져서 Local Minima가 많이 생긴다. Cross Entropy의 log는 오히려 그래프를 단순하게 만든다.
4. 또한 0또는 1 형태의 정답인 classificatio의 경우 cross entropy가 학습이 더 잘된다.

Logistic Regression

(활성화 함수를 통해 곡선 표현)
(Sigmoid + Cross Entropy)

Logistic Regression

- Reminder
- Computing Hypothesis
- Computing Cost Function
- Evaluation
- Higher Implementation
- Experimental!

Reminder : Logistic Regression

Hypothesis

$$H(X) = \frac{1}{1 + e^{-W^T X}}$$

Cost

$$cost(W) = -\frac{1}{m} \sum y \log(H(x)) + (1 - y) (\log(1 - H(x))$$

- If $y \simeq H(x)$, cost is near 0.
- If $y \neq H(x)$, cost is high.

Reminder : Logistic Regression

Weight Update via Gradient Descent

$$W := W - \alpha \underbrace{\underbrace{\frac{\partial}{\partial W} cost(W)}_{\text{Gradient Descent}}}_{}$$

Weight Update

- α: Learning rate

Import

```
import torch    // Pytorch 라이브러리
import torch.nn as nn    // nn.module 라이브러리
import torch.nn.functional as F   // F.mse_loss 라이브러리
import torch.optim as optim   // Gradient descent 라이브러리
```

```
# For reproducibility
torch.manual_seed(1)
```

```
<torch._C.Generator at 0x7f247d342fb0>
```

Training Data

```
x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]
y_data = [[0], [0], [0], [1], [1], [1]]
```

Consider the following classification problem: given the number of hours each student spent watching the lecture and working in the code lab, predict whether the student passed or failed a course. For example, the first (index 0) student watched the lecture for 1 hour and spent 2 hours in the lab session ([1, 2]), and ended up failing the course ([0]).

```
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)
```

As always, we need these data to be in `torch.Tensor` format, so we convert them.

```
print(x_train.shape)
print(y_train.shape)
```

```
torch.Size([6, 2])
torch.Size([6, 1])
```

Computing the Hypothesis

$$H(X) = \frac{1}{1 + e^{-W^T X}}$$

PyTorch has a `torch.exp()` function that resembles the exponential function.

```
print('e^1 equals: ', torch.exp(torch.FloatTensor([1])))
```

```
e^1 equals:  tensor([2.7183])
```

We can use it to compute the hypothesis function conveniently.

```
W = torch.zeros((2, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```

```
hypothesis = 1 / (1 + torch.exp(-(x_train.matmul(W) + b)))
```

```
print(hypothesis)
print(hypothesis.shape)
```

```
tensor([[0.5000],
        [0.5000],
        [0.5000],
        [0.5000],
        [0.5000],
        [0.5000]], grad_fn=<MulBackward0>)
torch.Size([6, 1])
```

Computing the Hypothesis

Or, we could use `torch.sigmoid()` function! This resembles the sigmoid function:

```
print('1/(1+e^{-1}) equals: ', torch.sigmoid(torch.FloatTensor([1])))
```

```
1/(1+e^{-1}) equals:  tensor([0.7311])
```

Now, the code for hypothesis function is cleaner.

```
hypothesis = 1 / (1 + torch.exp(-(x_train.matmul(W) + b)))
```

```
hypothesis = torch.sigmoid(x_train.matmul(W) + b)
```

```
print(hypothesis)
print(hypothesis.shape)
```

```
tensor([[0.5000],
        [0.5000],
        [0.5000],
        [0.5000],
        [0.5000],
        [0.5000]], grad_fn=<SigmoidBackward>)
torch.Size([6, 1])
```

Computing the Cost Function

$$cost(W) = -\frac{1}{m} \sum y \log(H(x)) + (1-y)(\log(1-H(x))$$

We want to measure the difference between `hypothesis` and `y_train`.

```
print(hypothesis)
print(y_train)
```

```
tensor([[0.5000],
        [0.5000],
        [0.5000],
        [0.5000],
        [0.5000],
        [0.5000]], grad_fn=<SigmoidBackward>)
tensor([[0.],
        [0.],
        [0.],
        [1.],
        [1.],
        [1.]])
```

Computing the Cost Function

For one element, the loss can be computed as follows:

```
-(y_train[0] * torch.log(hypothesis[0]) +
  (1 - y_train[0]) * torch.log(1 - hypothesis[0]))
```

```
tensor([0.6931], grad_fn=<NegBackward>)
```

Computing the Cost Function

To compute the losses for the entire batch, we can simply input the entire vector.

```
losses = -(y_train * torch.log(hypothesis) +
           (1 - y_train) * torch.log(1 - hypothesis))
print(losses)
```

```
tensor([[0.6931],
        [0.6931],
        [0.6931],
        [0.6931],
        [0.6931],
        [0.6931]], grad_fn=<NegBackward>)
```

Then, we just `.mean()` to take the mean of these individual losses.

```
cost = losses.mean()
print(cost)
```

```
tensor(0.6931, grad_fn=<MeanBackward1>)
```

Computing the Cost Function

나중에 사실상 Sigmoid아닌 Softmax와 같이 쓰기 때문에 앞에서는
Sigmoid를 예로 들었지만, 앞으로는 Softmax + Cross entropy 조합

F.binary_cross_entropy()==softmax + cross entropy

```
F.binary_cross_entropy(hypothesis, y_train)
```

```
tensor(0.6931, grad_fn=<BinaryCrossEntropyBackward>)
```

Whole training process

```
x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]
y_data = [[0], [0], [0], [1], [1], [1]]
```

```
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)
```

```
# 모델 초기화
W = torch.zeros((2, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
# optimizer 설정
optimizer = optim.SGD([W, b], lr=1)

nb_epochs = 1000
for epoch in range(nb_epochs + 1):

    # Cost 계산
    hypothesis = torch.sigmoid(x_train.matmul(W) + b) # or .mm or @
    cost = F.binary_cross_entropy(hypothesis, y_train)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 100번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch, nb_epochs, cost.item()
        ))
```

층이 여러 개면 이렇게 표현하면 너무 복잡해짐=> nn.module 사용

결과 보기!!

nn.Module 활용

```python
# 모델 초기화
W = torch.zeros((3, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)

# H(x) 계산
hypothesis = x_train.matmul(W) + b # or .mm or @
```

```python
import torch.nn as nn

class MultivariateLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 1)

    def forward(self, x):
        return self.linear(x)

hypothesis = model(x_train)
```

- nn.Module 을 상속해서 모델 생성
- nn.Linear(3, 1)
    - 입력 차원: 3
    - 출력 차원: 1
- Hypothesis 계산은 forward() 에서!
- Gradient 계산은 PyTorch 가 알아서 해준다 backward()

F.mse_loss 활용

```python
# cost 계산
cost = torch.mean((hypothesis - y_train) ** 2)
```

공식은 매번 다시 써주어야 함

```python
import torch.nn.functional as F

# cost 계산
cost = F.mse_loss(prediction, y_train)
```

- torch.nn.functional 에서 제공하는 loss function 사용
- 쉽게 다른 loss와 교체 가능! (l1_loss, smooth_l1_loss 등...)

Full Code with torch.optim (1)

```python
# 데이터
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])

# 모델 초기화
W = torch.zeros((3, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
model = MultivariateLinearRegressionModel()


# optimizer 설정
optimizer = optim.SGD([W, b], lr=1e-5)
```

1. 데이터 정의

2. 모델 정의

3. optimizer 정의

```python
import torch.nn as nn

class MultivariateLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 1)

    def forward(self, x):
        return self.linear(x)

hypothesis = model(x_train)
```

Full Code with torch.optim (2)

```
nb_epochs = 20
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = x_train.matmul(W) + b # or .mm or @
    Hypothesis = model(x_train)

    # cost 계산
    cost = torch.mean((hypothesis - y_train) ** 2)
    cost = F.mse_loss(prediction, y_train)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    print('Epoch {:4d}/{} hypothesis: {} Cost: {:.6f}'.format(
        epoch, nb_epochs, hypothesis.squeeze().detach(),
        cost.item()
    ))
```

4. Hypothesis 계산

5. Cost 계산 (MSE)

6. Gradient descent

Higher Implementation with Class

```
class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(8, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return self.sigmoid(self.linear(x))
```

```
model = BinaryClassifier()
```

Higher Implementation with Class

```
x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]
y_data = [[0], [0], [0], [1], [1], [1]]
```

```
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)
```

```
optimizer = optim.SGD(model.parameters(), lr=1)

nb_epochs = 100
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = model(x_train)

    # cost 계산
    cost = F.binary_cross_entropy(hypothesis, y_train)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 20번마다 로그 출력
    if epoch % 10 == 0:
        prediction = hypothesis >= torch.FloatTensor([0.5])
        correct_prediction = prediction.float() == y_train
        accuracy = correct_prediction.sum().item() / len(correct_prediction
        print('Epoch {:4d}/{} Cost: {:.6f} Accuracy {:2.2f}%'.format(
            epoch, nb_epochs, cost.item(), accuracy * 100,
        ))
```

```
class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(8, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return self.sigmoid(self.linear(x))
```

```
model = BinaryClassifier()
```

위의 코드에서 그리는 모델 네트워크 그림 그려 보기!! (활성화 함수 및 Loss 함수 포함해서)

Higher Implementation with Class

```
class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(8, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return self.sigmoid(self.linear(x))
```

```
model = BinaryClassifier()
```

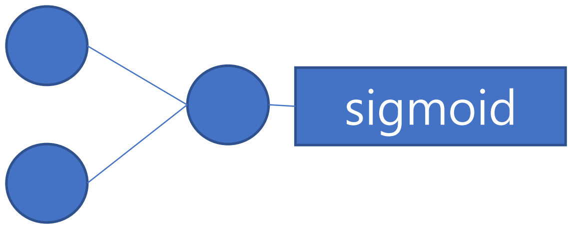
위의 코드를 간단히 대체

linear=torch.nn.Linear(8, 1, bias=True)
model=torch.nn.Sequential(linear, sigmoid).to(device)

Higher Implementation with Class

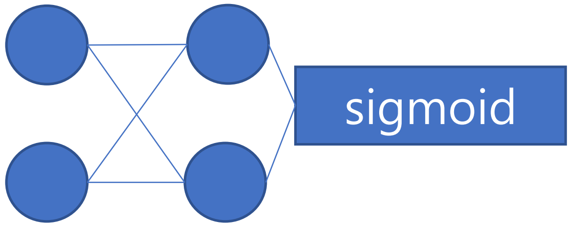
torch.nn.sequential 이란?

torch.nn.sequential(linear, sigmoid)



Linear1 = torch.nn.Linear(2, 2, bias=True)
Linear2 = torch.nn.Linear(2, 1, bias=True)
Sigmoid = torch.nn.Sigmoid()

torch.nn.sequential(linear1, sigmoid,linear2,sigmoid)

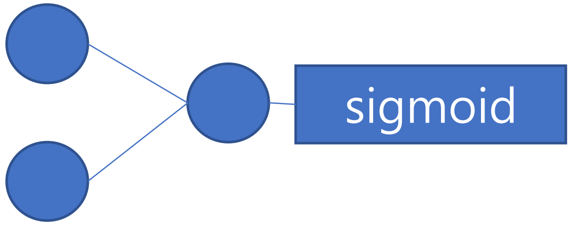


Experimental 1 – OR Network (1)

```python
import torch

device = 'cuda' if torch.cuda.is_available() else 'cpu'

# for reproducibility
torch.manual_seed(777)
if device == 'cuda':
    torch.cuda.manual_seed_all(777)

X = 
Y = 

# nn layers
linear = torch.nn.Linear(2, 1, bias=True)
sigmoid = torch.nn.Sigmoid()

# model
model = torch.nn.Sequential(linear, sigmoid).to(device)

# define cost/loss & optimizer
criterion = torch.nn.BCELoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=1)
```

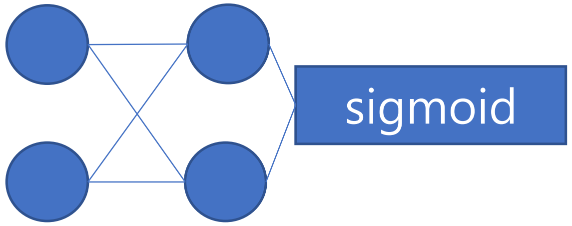
torch.nn.sequential 이란?

torch.nn.sequential(linear, sigmoid)



Linear1 = torch.nn.Linear(2, 2, bias=True)
Linear2 = torch.nn.Linear(2, 1, bias=True)
Sigmoid = torch.nn.Sigmoid()

torch.nn.sequential(linear1, sigmoid,linear2,sigmoid)



Experimental 1 – OR Network (2)

```
for step in range(10001):
    optimizer.zero_grad()
    hypothesis = model(X)

    # cost/loss function
    cost = criterion(hypothesis, Y)
    cost.backward()
    optimizer.step()

    if step % 100 == 0:
        print(step, cost.item())
```

```
if step % 10000 == 0:
    for param in model.parameters():
        print(param.data)
```

추가하면 weight값을 볼 수 있음

```
# Accuracy computation
# True if hypothesis>0.5 else False
with torch.no_grad():
    predicted = (model(X) > 0.5).float()
    accuracy = (predicted == Y).float().mean()
    print('\nHypothesis: ', hypothesis.detach().cpu().numpy(), '\nCorrect: ', predicted.detach().cpu().numpy(), '\nAccuracy: ', accuracy.item())
```

Evaluation

After we finish training the model, we want to check how well our model fits the training set.

```
hypothesis = torch.sigmoid(x_train.matmul(W) + b)
print(hypothesis[:5])
```

```
tensor([[0.4103],
        [0.9242],
        [0.2300],
        [0.9411],
        [0.1772]], grad_fn=<SliceBackward>)
```

Evaluation

We can change **hypothesis** (real number from 0 to 1) to **binary predictions** (either 0 or 1) by comparing them to 0.5.

```
prediction = hypothesis >= torch.FloatTensor([0.5])
print(prediction[:5])
```

```
tensor([[0],
        [1],
        [0],
        [1],
        [0]], dtype=torch.uint8)
```

Evaluation

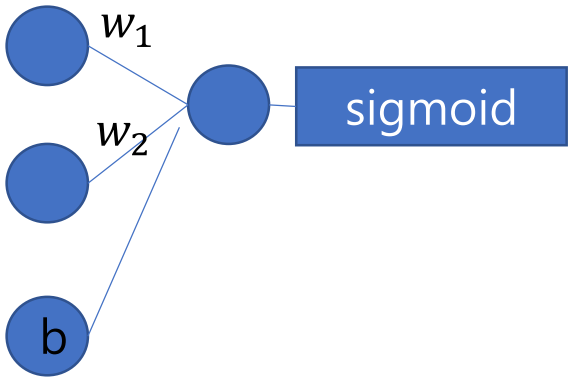
Then, we compare it with the correct labels `y_train`.

```
print(prediction[:5])
print(y_train[:5])
```

```
tensor([[0],
        [1],
        [0],
        [1],
        [0]], dtype=torch.uint8)
tensor([[0.],
        [1.],
        [0.],
        [1.],
        [0.]])
```

Experimental 1 – OR Network (2)

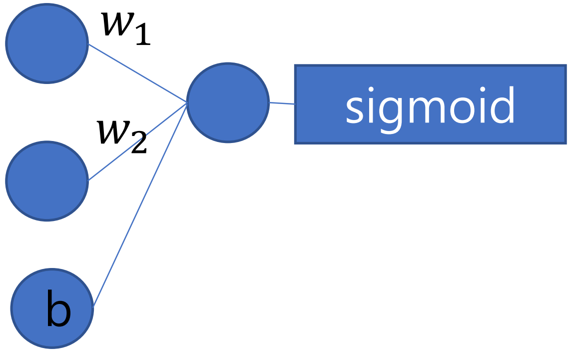
- **Question 1**
  - **W1, W2 구하기**
  - **Input=(0,0)일 때 출력 값?**
  - **Input=(1,0)일 때 출력 값?**
  - **Input=(0.5,0.5)일 때 출력 값?**



Input	Output
(0, 0)	0
(1, 0)	1
(0, 1)	1
(1, 1)	1

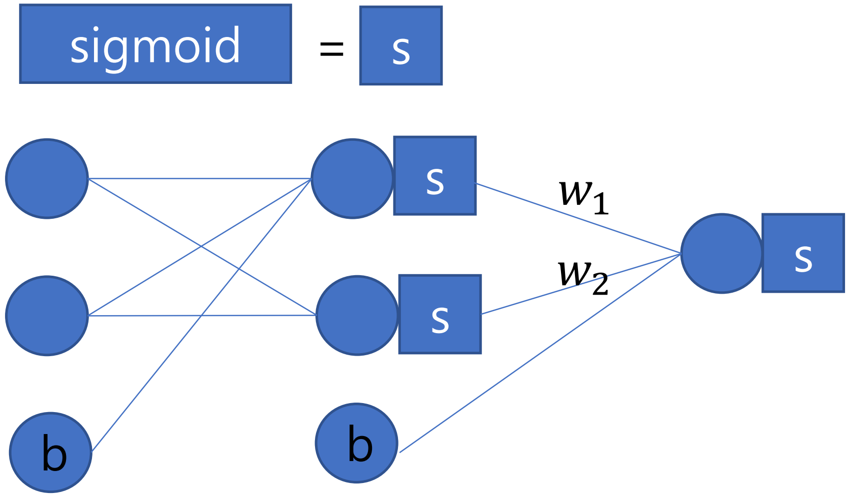
Experimental 1 – XOR Network (2)

- **Question 2 (layer-(2,1)일 때)**
  - **W1, W2 구하기**
  - **Input=(0,0)일 때 출력 값?**
  - **Input=(1,0)일 때 출력 값?**
  - **Input=(0.5,0.5)일 때 출력 값?**
  - **정확도 는?**
  - **정확도가 낮게 나오는 이유는?**



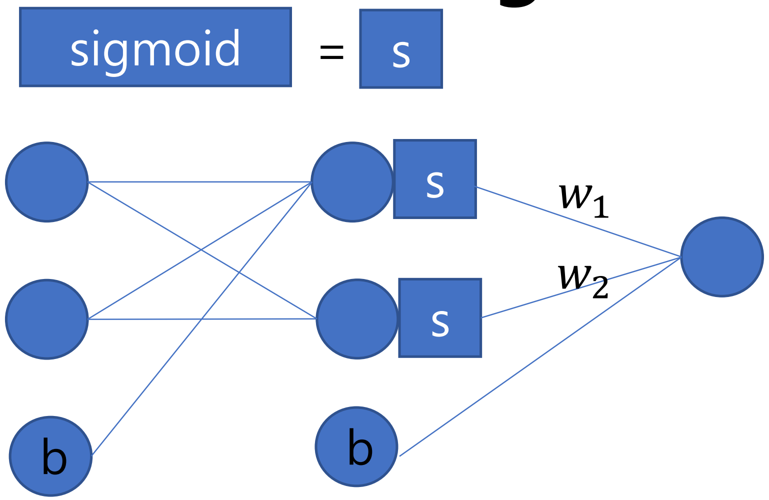
Input	Output
(0, 0)	0
(1, 0)	1
(0, 1)	1
(1, 1)	0

- **Question 3 (layer-(2,2,1)일 때)**
  - **W1, W2 구하기**
  - **Input=(0,0)일 때 출력 값?**
  - **Input=(1,0)일 때 출력 값?**
  - **Input=(0.5,0.5)일 때 출력 값?**

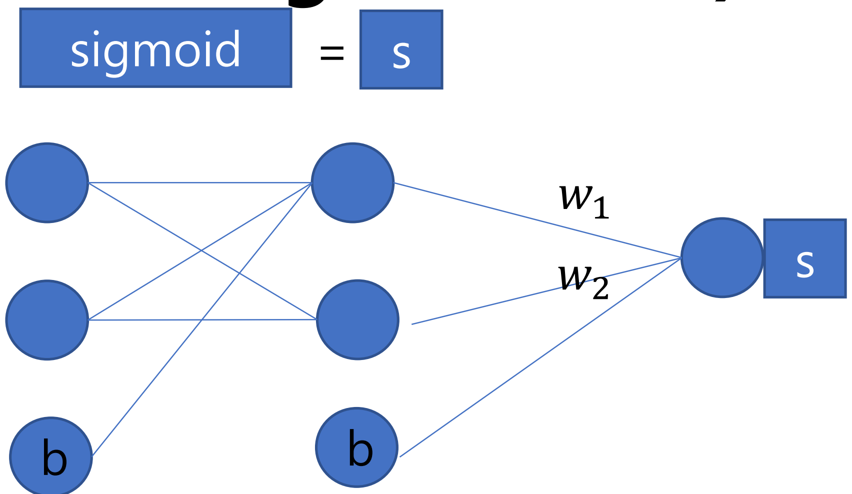


Experimental 1 – XOR Network (2)

- **Question 4 (layer-(2,2,1)일 때, 맨 마지막에 sigmoid빼기)**
  - **학습 시 문제가 있다면 그 이유는?**

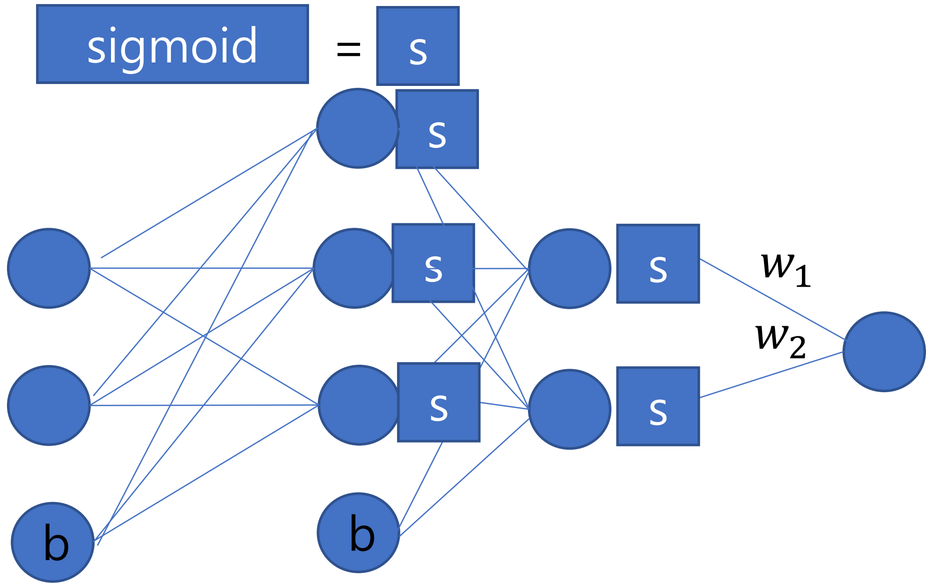


- **Question 5 (layer-(2,2,1)일 때, 중간에 sigmoid빼기)**
  - **W1, W2 구하기**
  - **Input=(0,0)일 때 출력 값?**
  - **Input=(1,0)일 때 출력 값?**
  - **Input=(0.5,0.5)일 때 출력 값?**
  - **정확도가 낮게 나오는 이유는?**



Experimental 1 – XOR Network (2)

- **Question 6 (layer-(2,3,2,1)일 때)**
  - **W1, W2 구하기**
  - **Input=(0,0)일 때 출력 값?**
  - **Input=(1,0)일 때 출력 값?**
  - **Input=(0.5,0.5)일 때 출력 값?**



- **Question 7 (layer-(2,2,2,2,2,1)일 때)**
  - **W1, W2 구하기**
  - **Input=(0,0)일 때 출력 값?**
  - **Input=(1,0)일 때 출력 값?**
  - **Input=(0.5,0.5)일 때 출력 값?**
  - **정확도가 낮게 나오는 이유는?**