PyTorch

Pytorch로 딥러닝

인공신경망 (DNN) 기초

임낙준

PyTorch

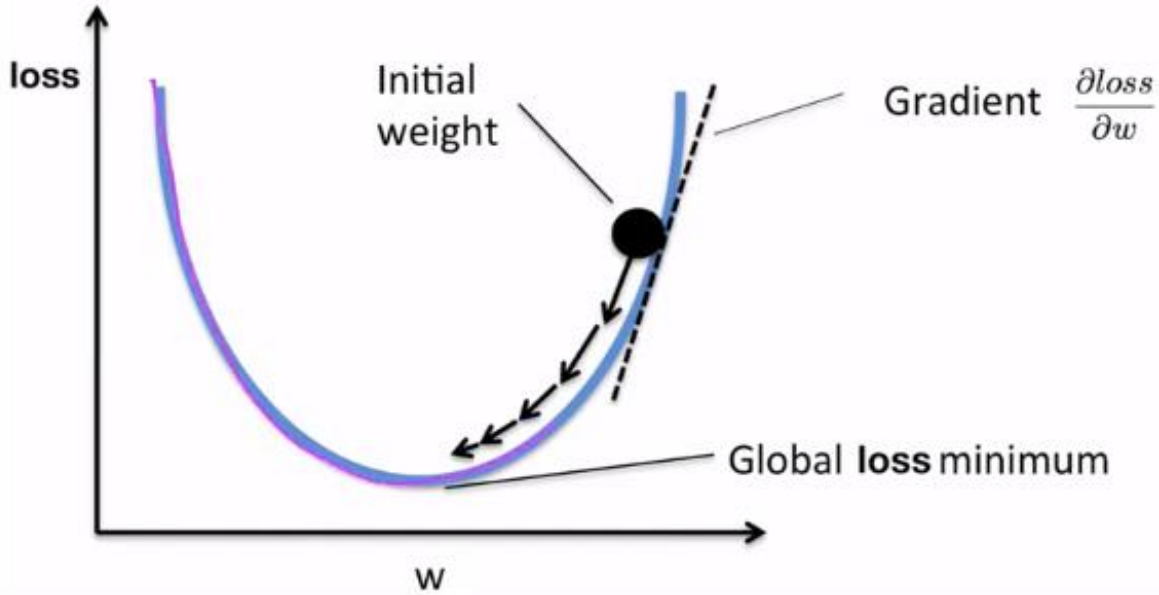
Contents

0. Gradient Descent Method

$$\hat{y} = XW + b$$

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \hat{y}\right)}^{2}$$

The square of the difference between actual and predicted



Loss(손실함수)를 MSE로 정의 하면,
이차함수 모양이기 때문에 오른쪽과 같이 불룩한 모양

통계적 회귀 모형은 기울기가 0인 지점(미분계수가 0)
을 한번에 찾아서 loss를 최소화 하는 w를 찾자는 것.

Gradient Descent는?

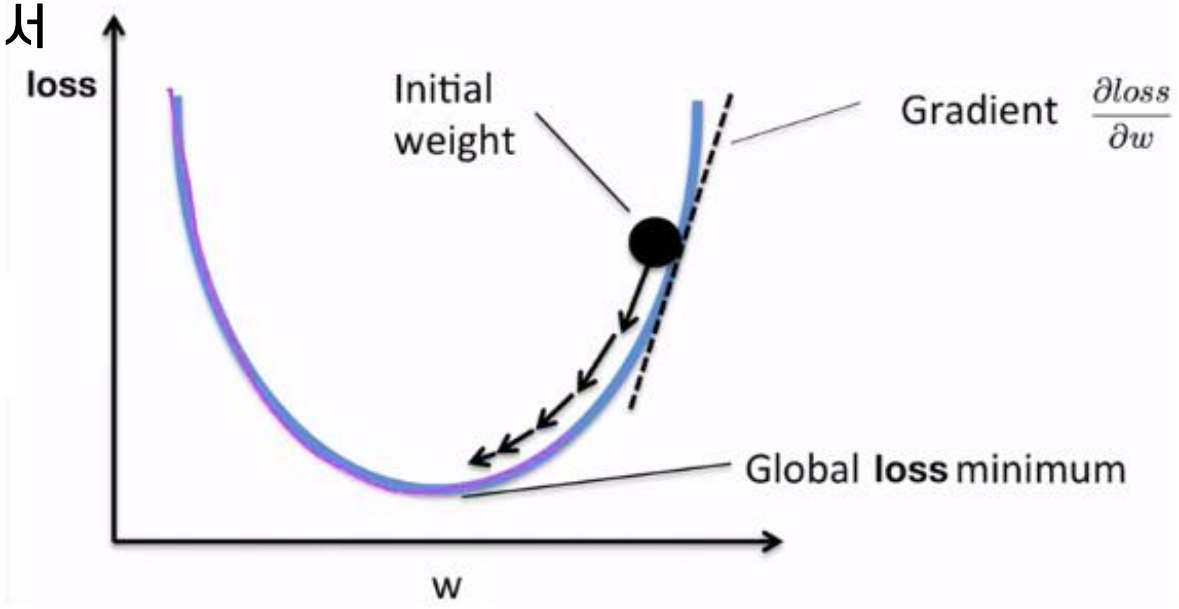
PyTorch

0. Gradient Descent Method

Gradient Descent Method(경사하강법)

손실함수를 미분하여 이 함수의 기울기(gradient)를 구해서
손실을 최소화되는 방향을 찾아 반복적으로 모델의
파라미터를 업데이트 하는 알고리즘

$$w_{n+1} = w_n - \gamma \nabla J(w_n),\ \gamma \text{ is a learning rate}$$



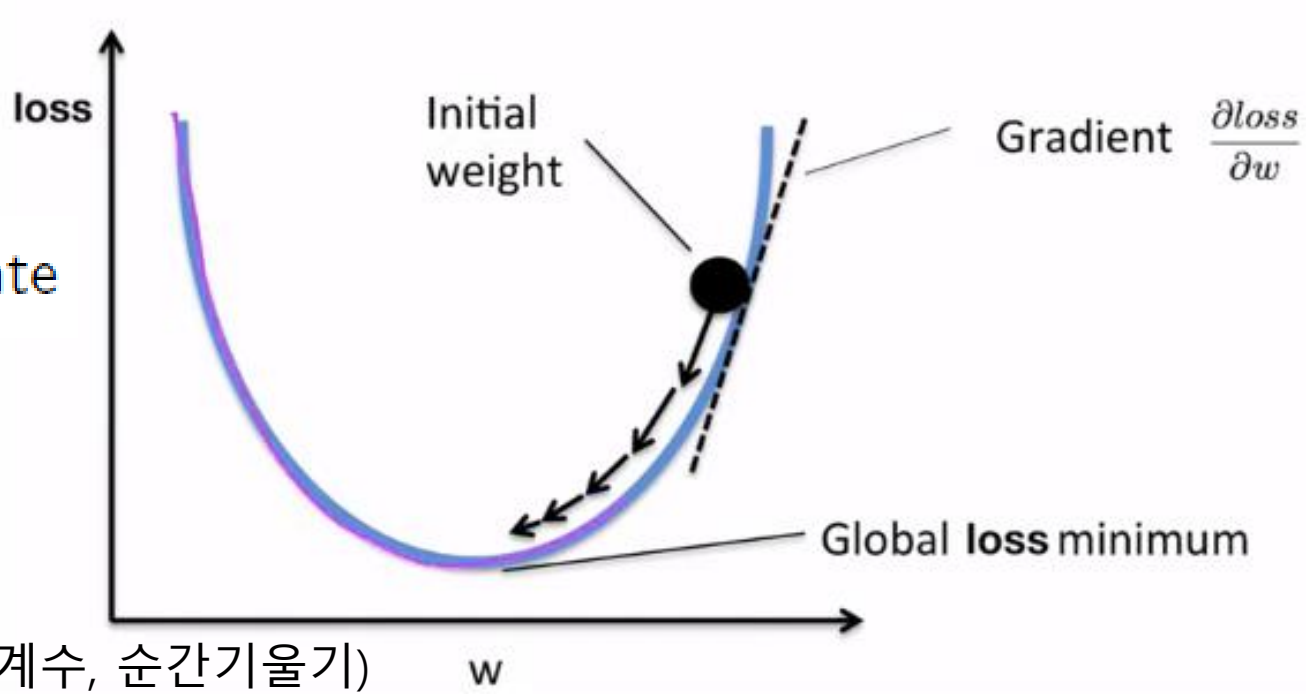
0. Gradient Descent Method

Gradient Descent Method(경사하강법) 수식과 그 의미

$$\hat{y} = XW + b$$

$w_{n+1} = w_n - \gamma \nabla J(w_n)$, γ is a learning rate

$$w_{n+1} = w_n - \gamma \frac{\delta loss}{\delta w}$$



learning rate or step size
얼마나 움직일 것인지 결정

gradient (미분계수, 순간기울기)
어느 방향으로 움직일지 결정

PyTorch

0. Gradient Descent Method

Gradient Descent Method(경사하강법) 으로 다항회귀 구현

자동 미분 (autograd)의 이해 먼저!

In [5]:
```
# 경사하강법 코드를 보고 있으면 requires_grad = True, backward() 등이 나온다.
# 이는 파이토치에서 제공하는 자동 미분 기능을 수행하고 있는 것이다.
```

In [6]:
```
import torch
# 임의의 스칼라 텐서 w 선언
# requires_grad = True는 이 텐서에 대한 기울기를 저장하겠다는 의미. 이렇게 하면 'grad'속성에 w에 대한 미분값이 저장됨
w = torch.tensor(2.0, requires_grad = True)
```

In [7]:
```
y = w**2
z = 2*y + 5
```

In [8]:
```
# backward()를 호출하면 해당 수식에서 requires_grad = True로 설정한 텐서들에 대한 기울기를 계산한다.
z.backward()
```

In [11]:
```
print(f'수식을 w로 미분한 값 : {w.grad}')
```

```
수식을 w로 미분한 값 : 8.0
```

0. Gradient Descent Method

Gradient Descent Method(경사하강법) 으로 다항회귀 구현

데이터 로드

```
import torch
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]])
```

PyTorch

0. Gradient Descent Method

Gradient Descent Method(경사하강법) 으로 다항회귀 구현

옵티마이저 설정 # 훈련 과정 (예측 -> 오차 계산 -> Gradient 계산 -> 미분계수로 파라미터 업데이트)

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

# optimizer 설정
optimizer = torch.optim.SGD([W,b],lr = 1e-5)     ## learning rate에 따라 발산할 수 있다. lr = 0.01만 돼도 발산한다.
nb_epochs = 100

for epoch in range(nb_epochs) :
    hypothesis = x_train.matmul(W) + b
    cost = torch.mean((hypothesis - y_train)**2)
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
```

0. Gradient Descent Method

Gradient Descent Method(경사하강법) 으로 다항회귀 구현

결과

```
Epoch    1/100 hypothesis: tensor([0., 0., 0., 0., 0.])  Cost : 29661.800781
Epoch    2/100 hypothesis: tensor([67.2578, 80.8397, 79.6523, 86.7394, 61.6605])  Cost : 9298.520508
Epoch    3/100 hypothesis: tensor([104.9128, 126.0990, 124.2466, 135.3015,  96.1821])  Cost : 2915.712891
Epoch    4/100 hypothesis: tensor([125.9942, 151.4381, 149.2133, 162.4896, 115.5097])  Cost : 915.040527
```

```
Epoch   97/100 hypothesis: tensor([152.7708, 183.6973, 180.9596, 197.0631, 140.1320])  Cost : 1.566340
Epoch   98/100 hypothesis: tensor([152.7704, 183.6976, 180.9594, 197.0630, 140.1324])  Cost : 1.565658
Epoch   99/100 hypothesis: tensor([152.7700, 183.6979, 180.9593, 197.0629, 140.1328])  Cost : 1.564987
Epoch  100/100 hypothesis: tensor([152.7695, 183.6982, 180.9592, 197.0628, 140.1332])  Cost : 1.564298
```

```
y_train = torch.FloatTensor([[152],[185],[180],[196],[142]])
```

0. Gradient Descent Method

PyTorch

Gradient Descent Method(경사하강법) 으로 다항회귀 구현
클래스로 파이토치 모델 구현하기

In [16]:
```
import torch.nn as nn
```

In [17]:
```
class MultivariateLinearRegressionModel(nn.Module) :
    def __init__(self) : # 여기서 모델의 구조를 정의하는 생성자를 정의한다.
        super().__init__()   # (above) 이는 객체가 갖는 속성값을 초기화 하는 역할로, 객체가 생성될때 자동으로 호출됨.
        self.linear = nn.Linear(3,1)  # 입력차원 3, 출력차원 1

    def forward(self, x) :      # forward 함수는 모델이 학습데이터를 입력받아서 forward연산 (예측)을 진행하는 함수
        return self.linear(x)       # 모델 객체를 데이터와 함께 호출하면 자동으로 실행됨.
```

In [18]:
```
model = MultivariateLinearRegressionModel()
```

In [21]:
```
# 모델에 저장되어 있는 파라미터 확인
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[-0.3791, -0.1642, -0.5551]], requires_grad=True), Parameter containing:
tensor([0.1892], requires_grad=True)]
```

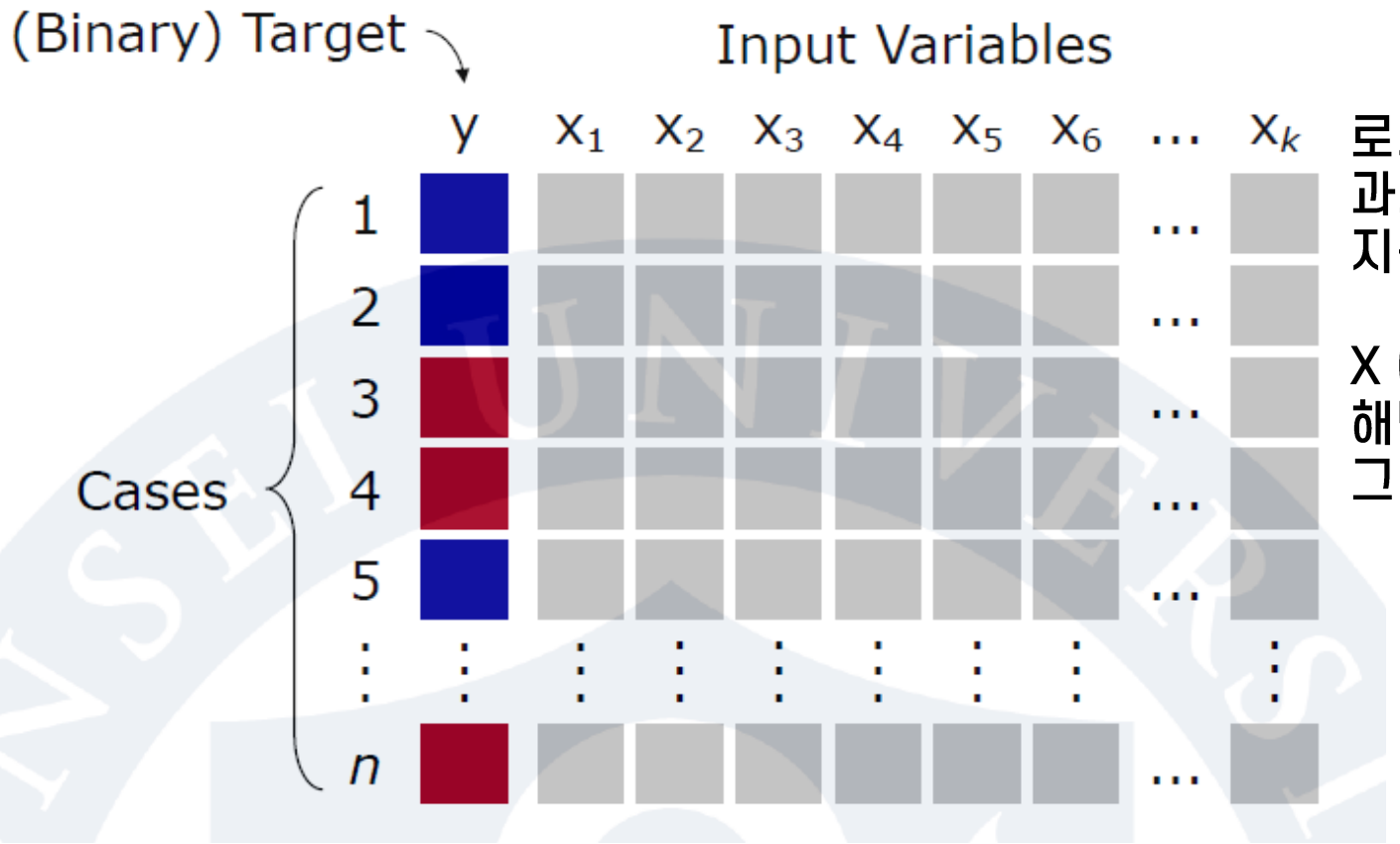
0. Gradient Descent Method

Gradient Descent Method(경사하강법) 으로 다항회귀 구현
클래스로 파이토치 모델 구현하기

```
In [22]: model = MultivariateLinearRegressionModel()
         optimizer = torch.optim.SGD(model.parameters(), lr = 0.00001)
         ## lr은 중요한 하이퍼 파라미터. 0.01, 0.001, 0.00001을 넣어보고 비교해봐라

         for epoch in range(nb_epochs) :
             # hypothesis 계산
             hypothesis = model(x_train)

             # cost 계산
             cost = torch.nn.functional.mse_loss(hypothesis, y_train)   # 파이토치에서 제공하는 평균 제곱 오차 함수


             # cost로 hypothesis 개선
             optimizer.zero_grad()
             cost.backward()
             optimizer.step()

             print('Epoch {:4d}/{} hypothesis: {}  Cost : {:.6f}'.format(epoch+1, nb_epochs, hypothesis.squeeze().detach(),
                                                                         cost.item()
                                                                         ))

Epoch    1/100 hypothesis: tensor([-8.2666, -2.7712, -6.4352, -6.7662, -0.5484])  Cost : 31427.130859
Epoch    2/100 hypothesis: tensor([60.9637, 80.4374, 75.5522, 82.5160, 62.9186])  Cost : 9852.556641
Epoch    3/100 hypothesis: tensor([ 99.7235, 127.0226, 121.4540, 132.5018,  98.4512])  Cost : 3090.071289
Epoch    4/100 hypothesis: tensor([121.4240, 153.1038, 147.1528, 160.4871, 118.3443])  Cost : 970.390503
```

PyTorch

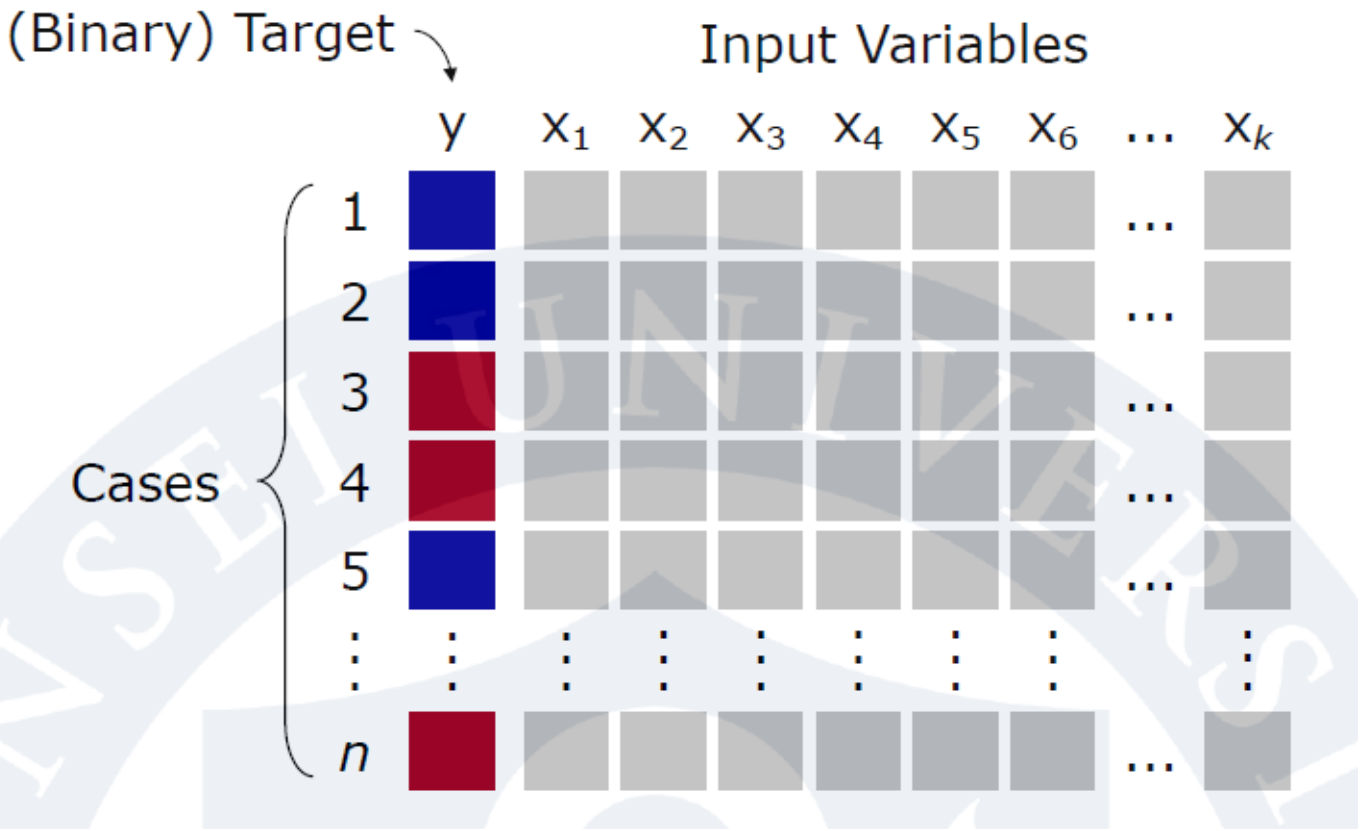
2. Logistic Regression



로지스틱 회귀는 이진 분류(즉, 나올 수 있는 결과가 2개밖에 없는 경우, 무엇이 더 발생 가능한지를 예측하기 위한)를 위한 회귀 모형!

X (input 데이터, features)를 가지고
해당 데이터가 0으로 분류될지, 1로 분류될지
그 확률을 계산하는 모형

2. Logistic Regression

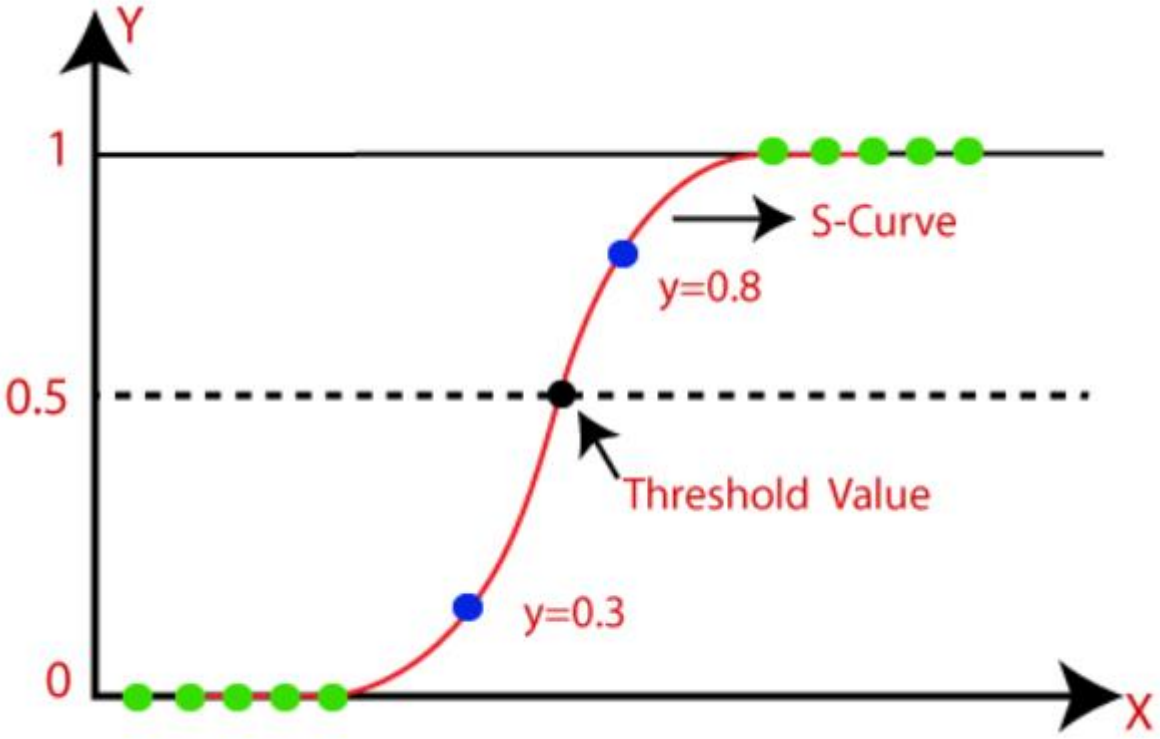


예를 들어, 합격을 1로 불합격을 0으로 둘때
X(국어성적) 이 커질수록
그 합격 확률을 높아진다고 할 수 있을 것.

그렇다면 X(국어 성적)과 y(합격 1,불합격 0)
과의 관계를 구체적으로 어떻게 모델링 할 수 있을까?

-> Next Page

2. Logistic Regression



Sigmoid Function(logistic function) 으로
X와 y의 관계를 모형화.

$$p(x) = \frac{e^{b_1 + b_1 x_1}}{1 + e^{b_0 + b_1 x_1}} = \frac{1}{1 + e^{-(b_0 + b_1 x_1)}}$$

0 < p(x) < 1

국어 성적에 더해 수학, 영어 성적까지 포함해서
합격여부를 예측할 때 식이 어떻게 바뀔까?

2. Logistic Regression

Prediction (예측확률) :

$$p(X) = \frac{e^{XW}}{1+e^{XW}} = \frac{1}{1+e^{-(XW)}}$$

Cost (오차) :

$$\text{cost} = -\frac{1}{m}\sum ylog(p(X)) + (1-y)(\log(1-p(X))$$

Cross Entropy로 계산

if $y \simeq p(X)$, cost gets near to 0

if $y \neq p(X)$, cost gets high.

2. Logistic Regression 코드 구현

데이터 생성

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

In [2]:
```
# For reproducivility
torch.manual_seed(1)
```

```
<torch._C.Generator at 0x257571d3f50>
```

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

> 주어진 분류 문제를 생각해보자 : 학생들에 공부에 투자한 시간이 정보로 주어졌을때, 학생들의 시험 통과 여부를 예측하는 분류 문제. 강의를 한시간 듣고, 자습에 두시간 들었다면 x_data = [1,2] , 시험에 통과하지못했다면 y_data = [0]

In [4]:
```
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)
```

PyTorch

In [75]:
```
print(x_train.shape)
print(y_train.shape)
```

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

Learning parameters

In [76]:
```
# learning parameters
W = torch.zeros([2,1], requires_grad= True)
b = torch.zeros([1], requires_grad= True)
```

로지스틱 회귀모형 생성과 훈련

In [77]:
```
optimizer = torch.optim.SGD([W,b], lr=0.05)

for epoch in range(nb_epochs) :
    # 직접 이렇게 sigmoid를 구현해도 됨
    # torch.sigmoid() 대신   직접 시그모이드 함수를 구현해도 됨 ->  1/(1+torch.exp(- x_train.matmul(W)+b))
    hypothesis = torch.sigmoid(x_train.matmul(W)+b)
    cost = F.binary_cross_entropy(hypothesis, y_train)  # hypothesis, y_train 순서를 바꾸면 에러가 남

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

    print(f'Epochs : {epoch+1}/{nb_epochs}   Cost : {cost}')
```

```
Epochs : 375/1000   Cost : 0.3568093776702881
Epochs : 376/1000   Cost : 0.3565260469913482
Epochs : 377/1000   Cost : 0.356243044137954
Epochs : 378/1000   Cost : 0.35596051812171936
```

2. Logistic Regression 코드 구현

```
In [78]: # 훈련된 모델의 예측값
         hypothesis

Out[78]: tensor([[0.0725],
                 [0.1928],
                 [0.4653],
                 [0.7150],
                 [0.8905],
                 [0.9654]], grad_fn=<SigmoidBackward>)

In [79]: # 실제 참값
         y_train

Out[79]: tensor([[0.],
                 [0.],
                 [0.],
                 [1.],
                 [1.],
                 [1.]])
```

2. Logistic Regression 코드 구현

nn.Module을 이용하여 간결하게 모델 구조화

pytorch가 제공하는 nn.Module을 사용하여 모델 구조화

In [50]:
```
class BinaryClassification(nn.Module) :
    def __init__(self) :
        super().__init__()
        self.logit = nn.Linear(2,1)

    def forward(self, x) :
        logit = self.logit(x)
        return torch.sigmoid(logit)
```

In [51]:
```
model = BinaryClassification()
```

In [52]:
```
nb_epochs = 1000

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

for epoch in range(nb_epochs) :
    hypothesis = model(x_train)

    cost = F.binary_cross_entropy(hypothesis, y_train)   # hypothesis, y_train 순서를 바꾸면 에러가 남

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

    print(f'Epochs : {epoch+1}/{nb_epochs}   Cost : {cost}')
```

3. Softmax

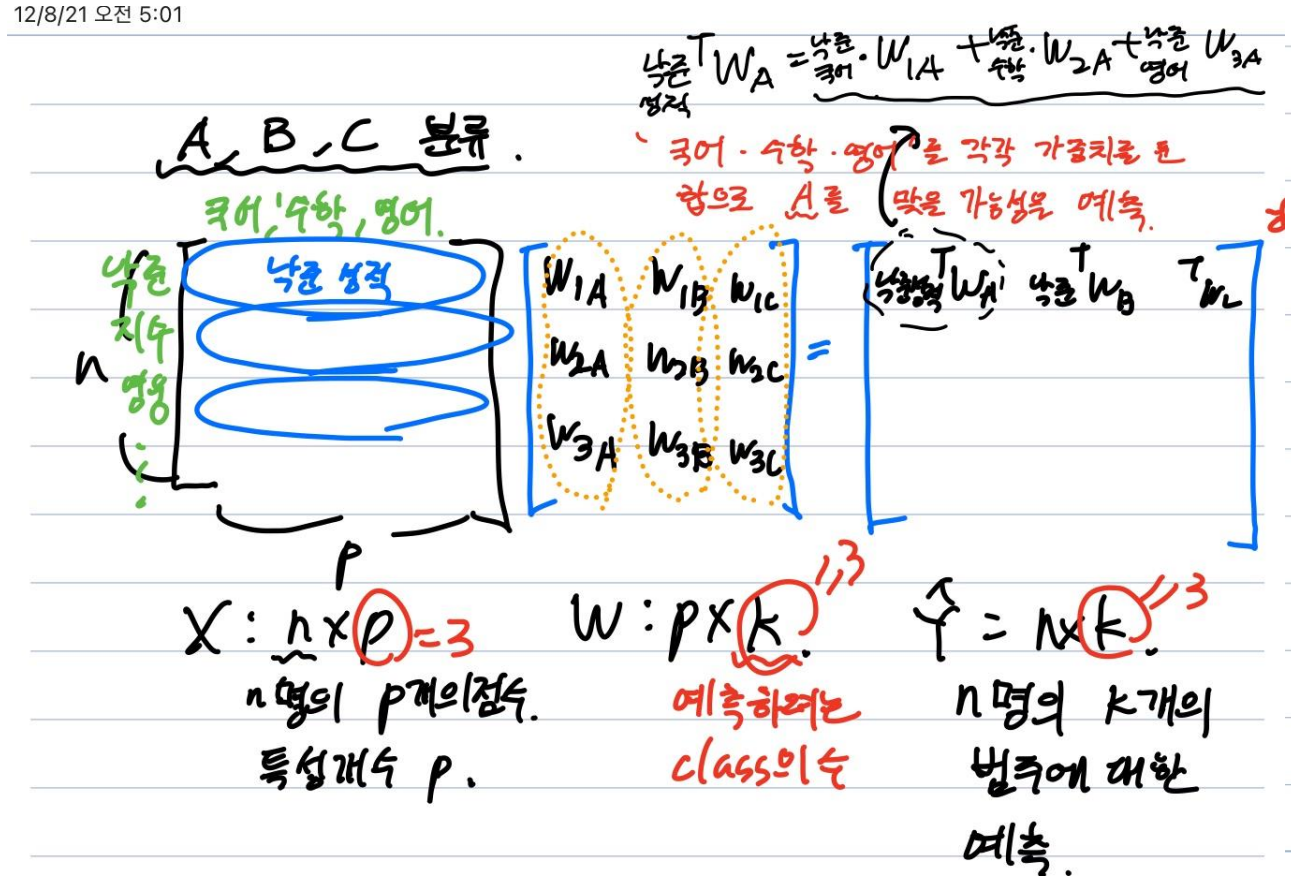
Logistic Regression : Binary Classification
(0,1) 만 분류

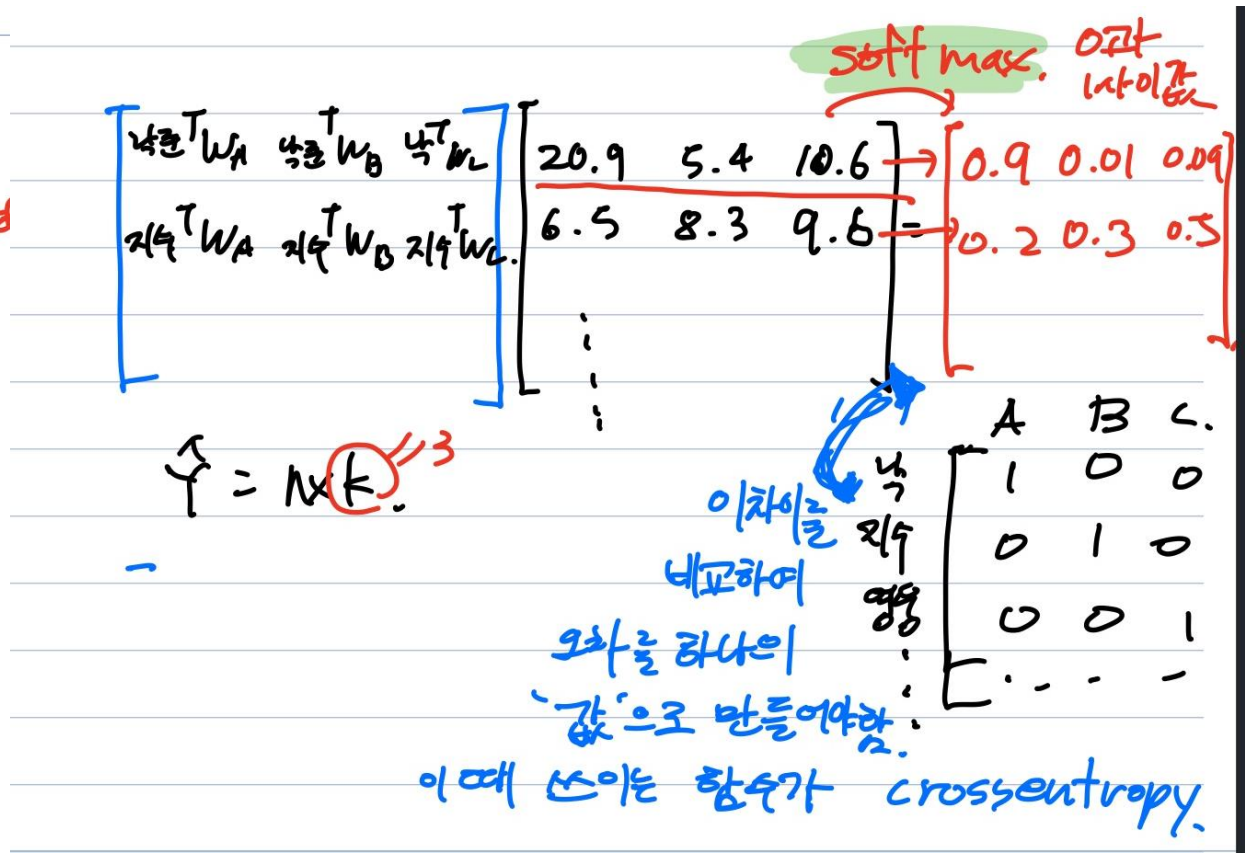
Softmax : Multinomial Classficiation
소프트맥스는 세 개 이상으로 분류하는 다중 클래스 분류에서 사용되는 함수다.
소프트맥스 함수는 분류될 클래스가 n개라 할 때, n차원의 벡터를 입력받아, 각 클래스에 속할 확률을 추정한다.

예제) 데싸랩 운영진들의 국어, 영어, 수학 성적과 국영수에 각각 어떠한 가중치를 매긴 것으로 A, B, C등급을 분류한 데이터가 있다.
주어진 데이터를 바탕으로 국,영,수 점수만을 가지고 A,B,C등급을 분류하는 모델을 만들고,
이를 바탕으로 데싸랩 회원들의 국,영,수 성적을 보고 A,B,C를 예측하는 모델을 만든다.
-> A,B,C 3개의 범주를 분류(또는 예측) 하는 Multinomial Classificaiton model!

3. Softmax

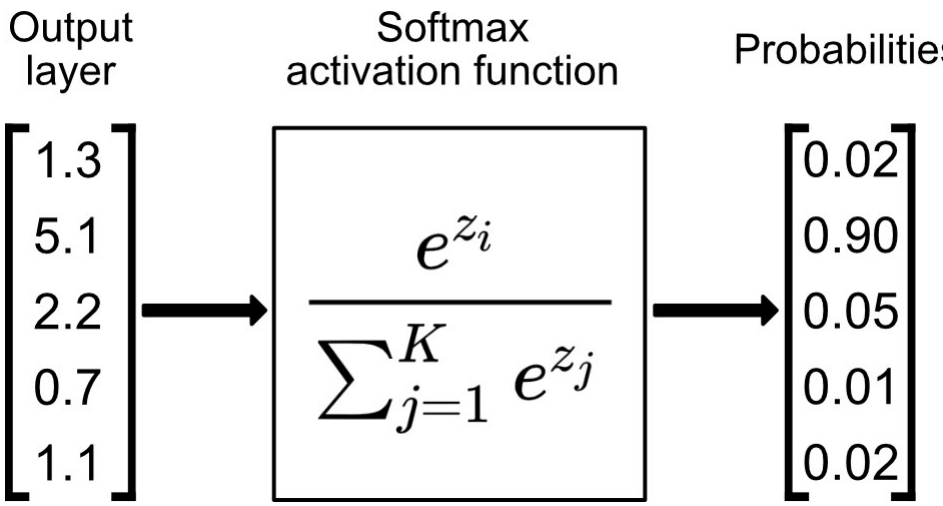
12/8/21 오전 5:01





3. Softmax

Softmax



Cross Entropy

$$\text{Loss} = -\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

(binary cross entropy의 확장)

In [26]:
```python
import torch
import torch.nn.functional as F
import torch.nn as nn
```

In [16]:
```python
class SoftmaxClassifierModel(nn.Module) :
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(4,3)

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

In [18]:
```python
# 모델 초기화
model = SoftmaxClassifierModel()
# optimizer 설정
optimizer = torch.optim.SGD(model.parameters(), lr=1)

nb_epochs = 1000
for epoch in range(nb_epochs) :
    # Cost 계산
    hypothesis = model(x_train)
    cost = F.cross_entropy(hypothesis, y_train)

    # prameter update
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

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

PyTorch

3. Softmax

```
In [9]:
x_train = [[1,2,1,1],
           [2,1,3,2],
           [3,1,3,4],
           [4,1,5,5],
           [1,7,5,5],
           [1,2,5,6],
           [1,6,6,6],
           [1,7,7,7]]
y_train = [2,2,2,1,1,1,0,0]
x_train = torch.FloatTensor(x_train)
y_train = torch.LongTensor(y_train)
```

Pytorch의
cross_entropy 함수는
자체적으로 softmax와
원-핫인코딩의 기능을
포함하고 있음

```
In [26]:
import torch
import torch.nn.functional as F
import torch.nn as nn
```

```
In [16]:
class SoftmaxClassifierModel(nn.Module) :
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(4,3)

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

```
In [18]:
# 모델 초기화
model = SoftmaxClassifierModel()
# optimizer 설정
optimizer = torch.optim.SGD(model.parameters(), lr=1)

nb_epochs = 1000
for epoch in range(nb_epochs) :
    # Cost 계산
    hypothesis = model(x_train)
    cost = F.cross_entropy(hypothesis, y_train)

    # prameter update
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

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

3. Softmax

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

```
Epoch 0/1000 , Cost : 1.6002955436706543
Epoch 100/1000 , Cost : 8.659186363220215
Epoch 200/1000 , Cost : 1.7698242664337158
Epoch 300/1000 , Cost : 1.6256897449493408
Epoch 400/1000 , Cost : 1.8773658275604248
Epoch 500/1000 , Cost : 2.0513205528259277
Epoch 600/1000 , Cost : 3.097203254699707
Epoch 700/1000 , Cost : 1.2486099004745483
Epoch 800/1000 , Cost : 0.8728685975074768
Epoch 900/1000 , Cost : 0.0035334948915988207
```

In [19]:
```python
hypothesis # 예측값
```

```
tensor([[-33.7473,   2.1603,  34.0113],
        [-19.8732,   6.8174,  14.5250],
        [-62.3554,  29.1691,  37.2332],
        [-50.4478,  32.5872,  22.0757],
        [  2.4817,   7.1825,  -0.5795],
        [  1.8612,  21.5579, -18.9747],
        [ 15.9636,  11.2257, -18.9673],
        [ 25.5124,  12.7910, -28.6972]], grad_fn=<AddmmBackward>)
```

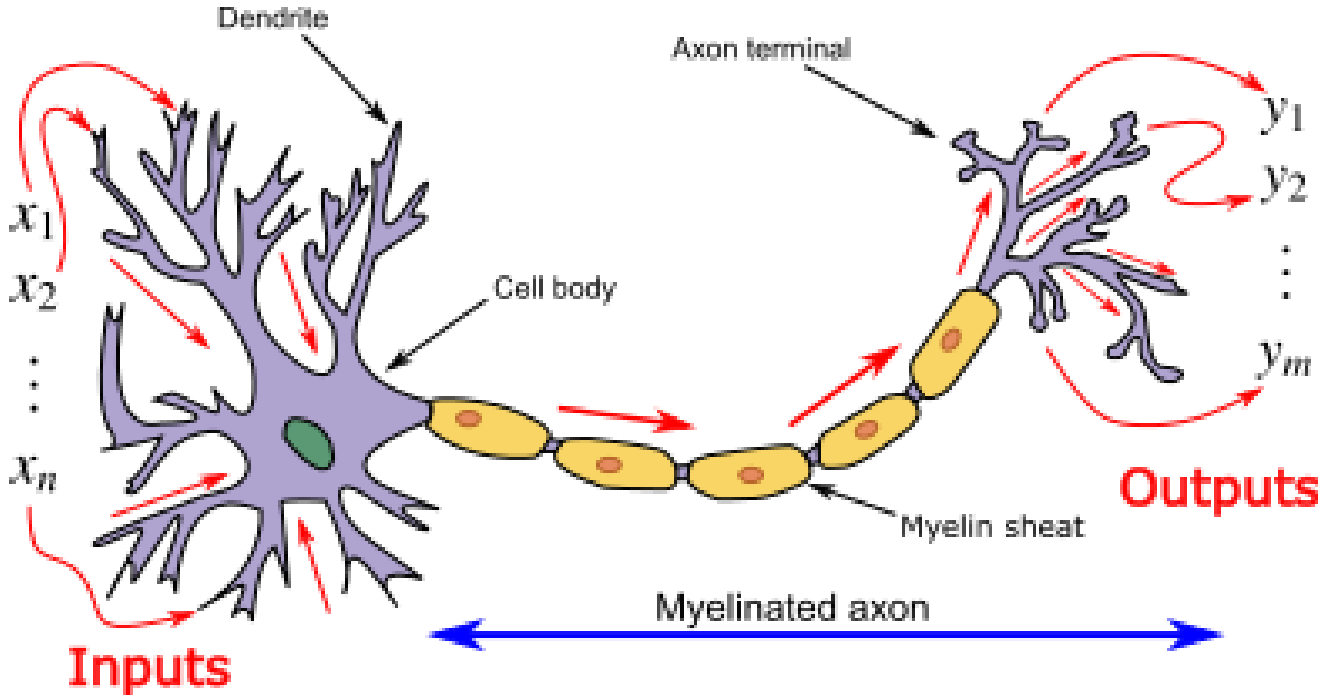
In [22]:
```python
F.softmax(hypothesis, dim=1)
```

```
tensor([[3.7395e-30, 1.4699e-14, 1.0000e+00],
        [1.1505e-15, 4.4923e-04, 9.9955e-01],
        [5.6052e-44, 3.1454e-04, 9.9969e-01],
        [8.6765e-37, 9.9997e-01, 2.7220e-05],
        [9.0027e-03, 9.9058e-01, 4.2158e-04],
        [2.7913e-09, 1.0000e+00, 2.4940e-18],
        [9.9132e-01, 8.6812e-03, 6.6979e-16],
        [1.0000e+00, 2.9863e-06, 2.8644e-24]], grad_fn=<SoftmaxBackward>)
```

In [27]:
```python
F.softmax(hypothesis, dim=1).sum(dim=1)
```

```
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
       grad_fn=<SumBackward1>)
```

4. 인공신경망



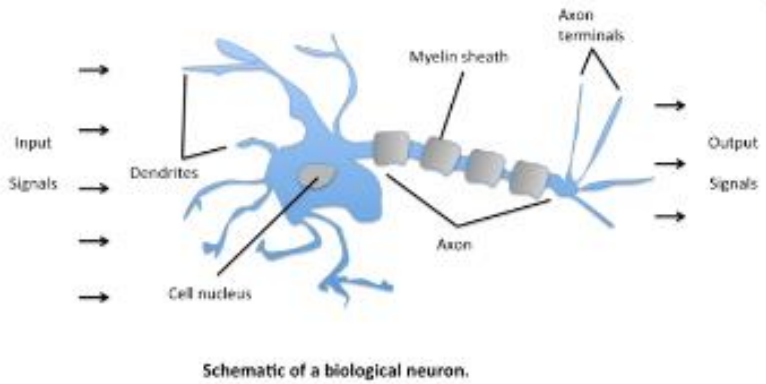
인간의 신경망 – 수많은 뉴런들이 모여 우리의 뇌를 구성

- 뉴런들이 전기적 신호를 주고 받으며 정보를 처리

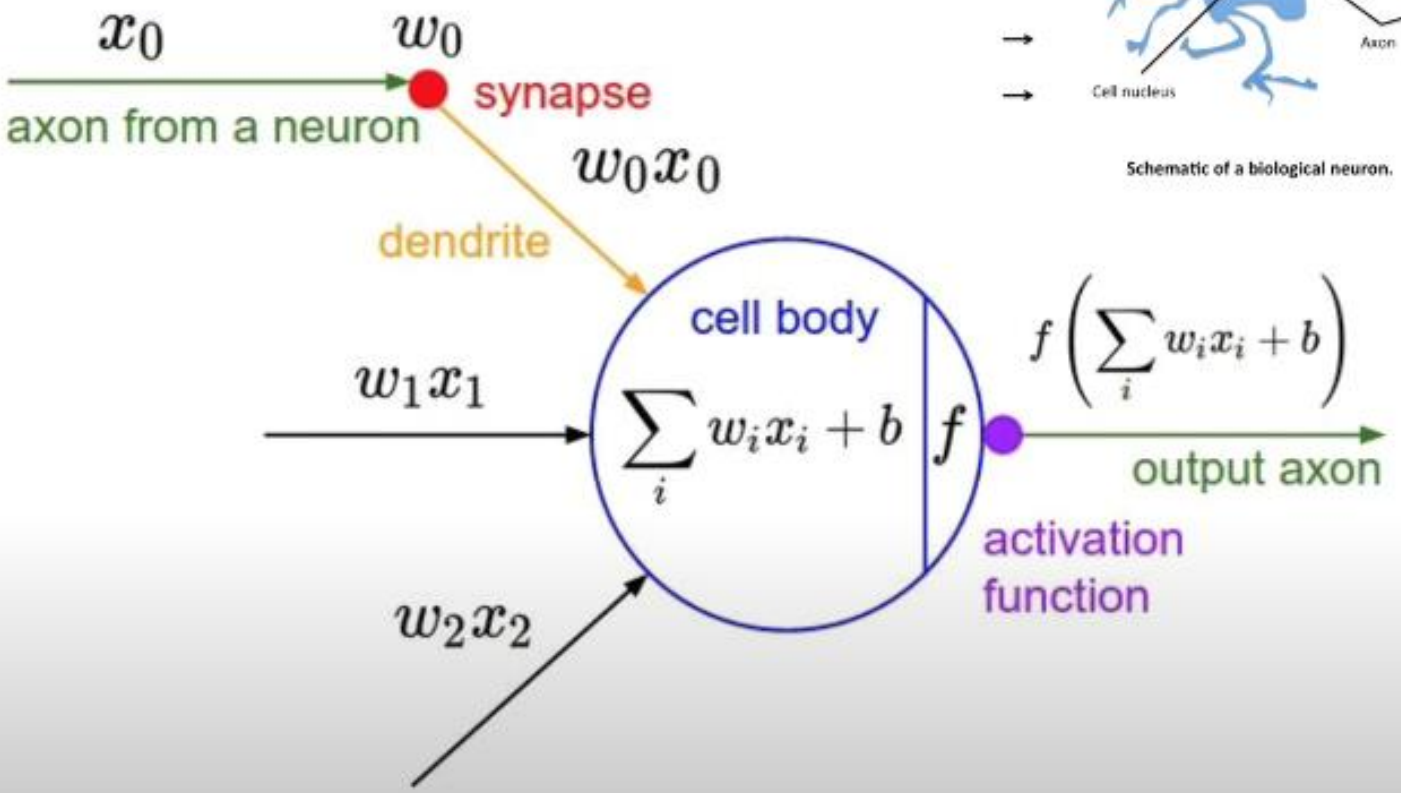
인공 신경망의 뉴런(Perceptron, node) – 인간의 뉴런을 모사

4. 인공신경망

Perceptron



Schematic of a biological neuron.



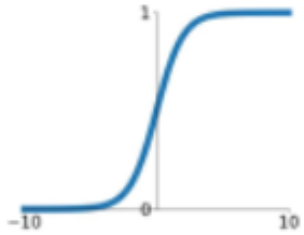
4. 인공신경망

Activation function (활성화 함수)

해당 노드를 활성화 시킬지, 말지를 결정하는 함수. 해당 노드의 값이 결과에 미치는 영향이 중요하다고 판단하면 큰값을 반환하여 '활성화' 시키고, 해당노드의 값이 중요하지 않으면 작은 값을 반환하여 'turn off' 시키는 함수. 현재 예제에서는 sigmoid를 쓰지만 sigmoid 말고도 Relu, tanh 등이 있다.
딥러닝의 비선형성을 제공

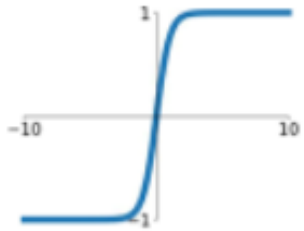
Sigmoid

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



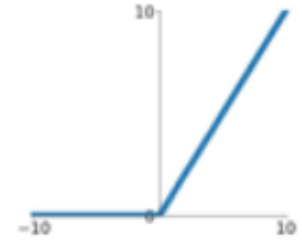
tanh

$$\tanh(x)$$

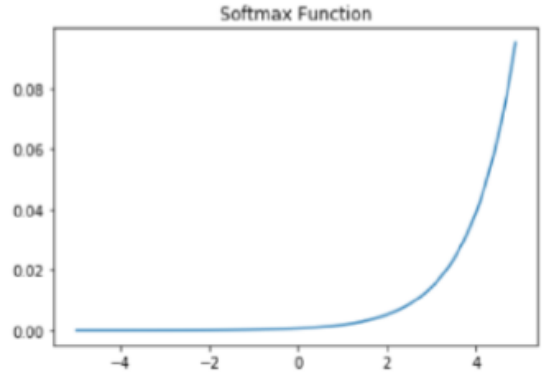


ReLU

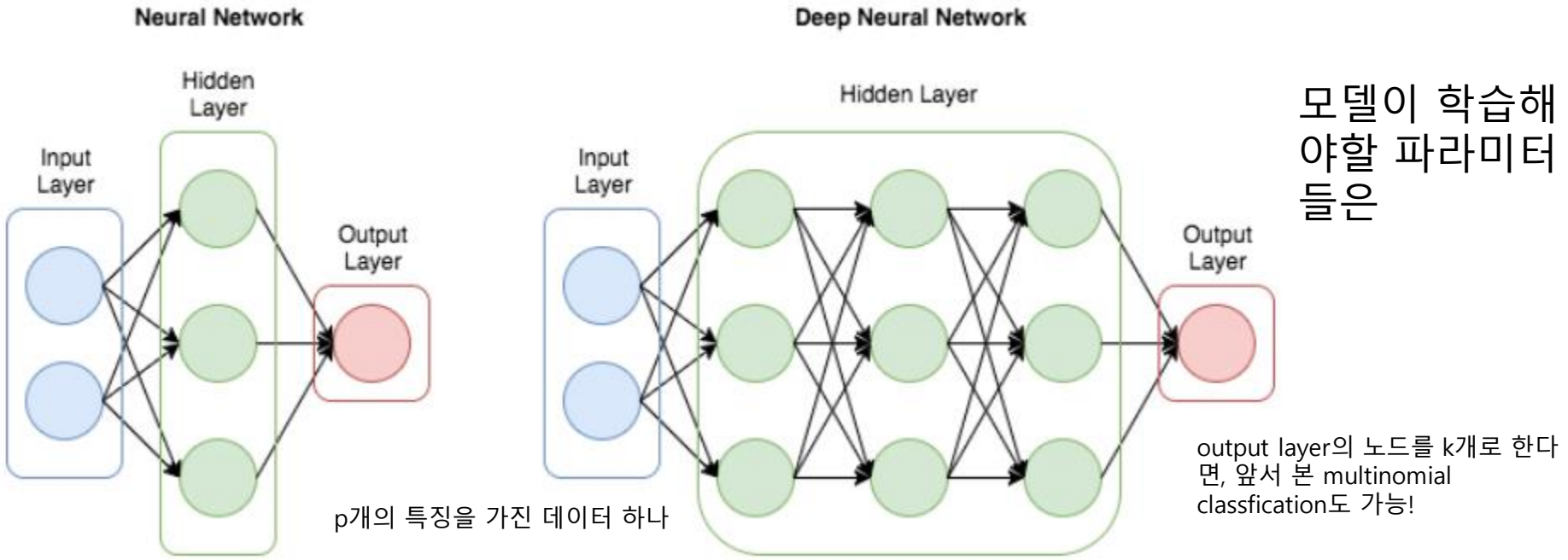
$$\max(0, x)$$



Softmax Function



4. 인공신경망(Deep Neural Network)



p개의 특징을 가진 데이터 하나

모델이 학습해야할 파라미터들은

output layer의 노드를 k개로 한다면, 앞서 본 multinomial classfication도 가능!

인공신경망 : 인간의 뇌가 뉴런이 얽히고 설켜 있듯, 퍼셉트론을 여러 개, 여러층으로 쌓아 올려 복잡한 문제를 풀 수 있도록 한 것

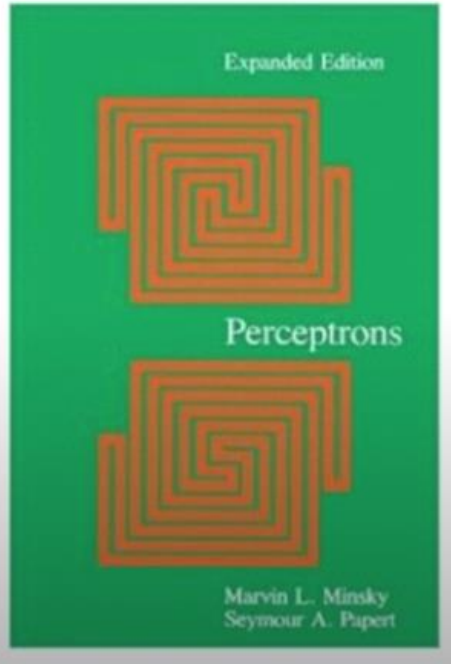
perceptron 를 여러층을 쌓아놓으면, 비 선형적인 모형을 만들 수 있다.

4. 인공신경망

모델 구조는 만들었는데… 학습은 어떻게?

Perceptrons (1969)

by Marvin Minsky, founder of the MIT AI Lab



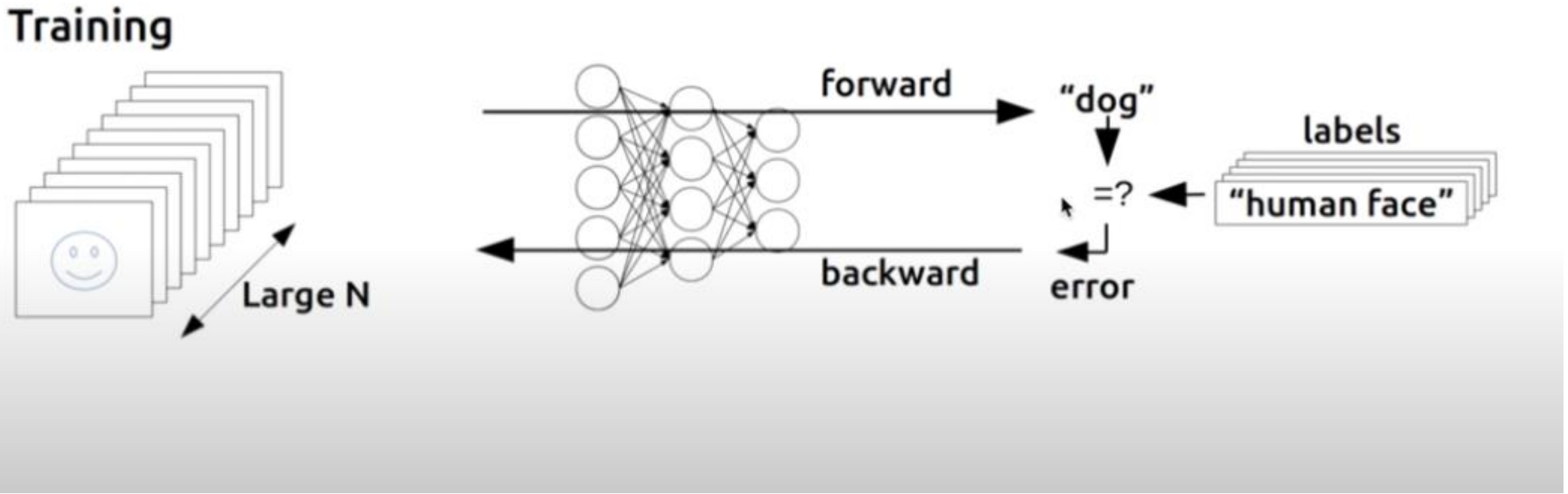
- We need to use MLP, multilayer perceptrons (multilayer neural nets)
- No one on earth had found a viable way to train MLPs good enough to learn such simple functions.

다층 퍼셉트론 구조는 일찌감치 개념화 되었지만, 이를 훈련시키는 것이 난제였음.

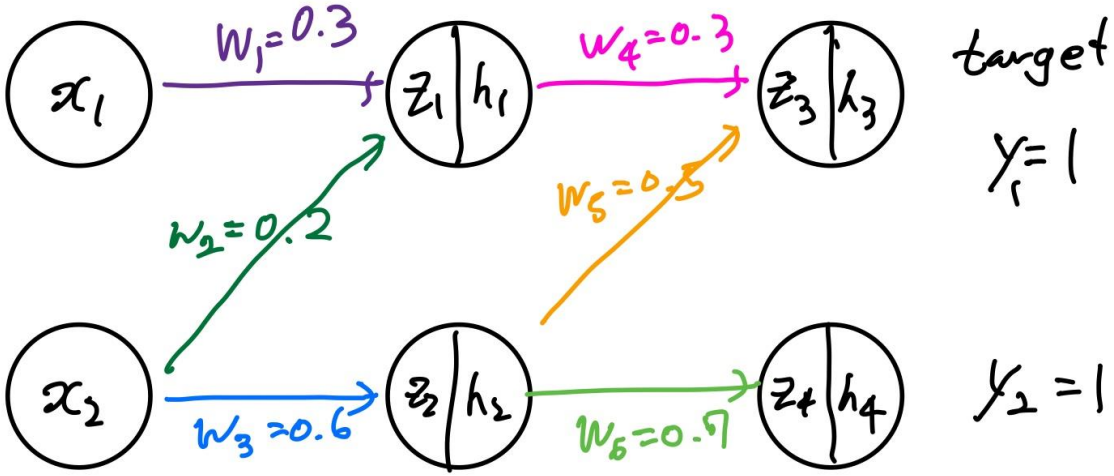
구조적 특성상, 미분계산량이 너무 많고 복잡했기 때문.

4. 인공신경망

Forward Propagation(순전파) : 모델로 예측값을 만든 후 오차를 계산하는 과정
Backward Propagation(역전파) : 예측값과 실제값으로부터 순전파의 반대방향으로 각 파라미터들에 대한 오차의 미분값을 구하는 과정. 이를 통해 경사하강법으로 파라미터를 업데이트 할 수 있다.

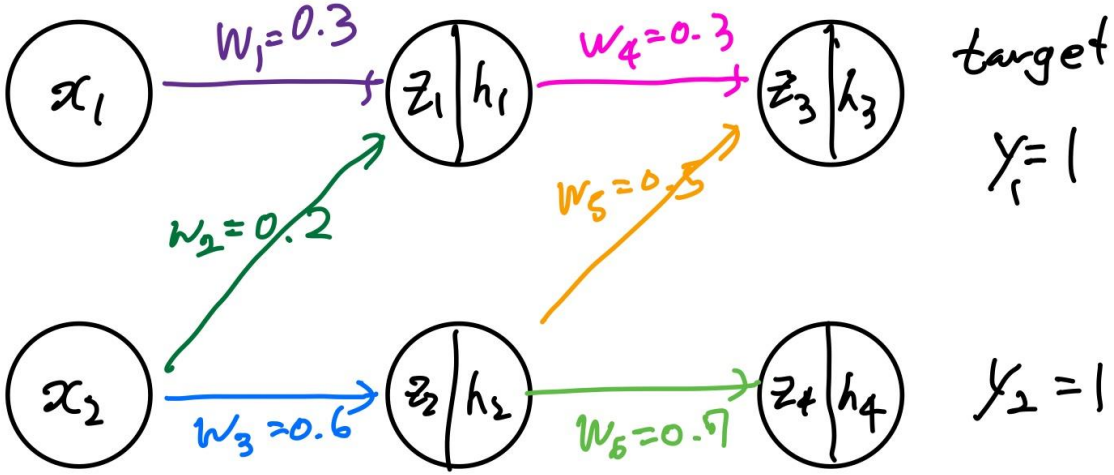


4. 인공신경망

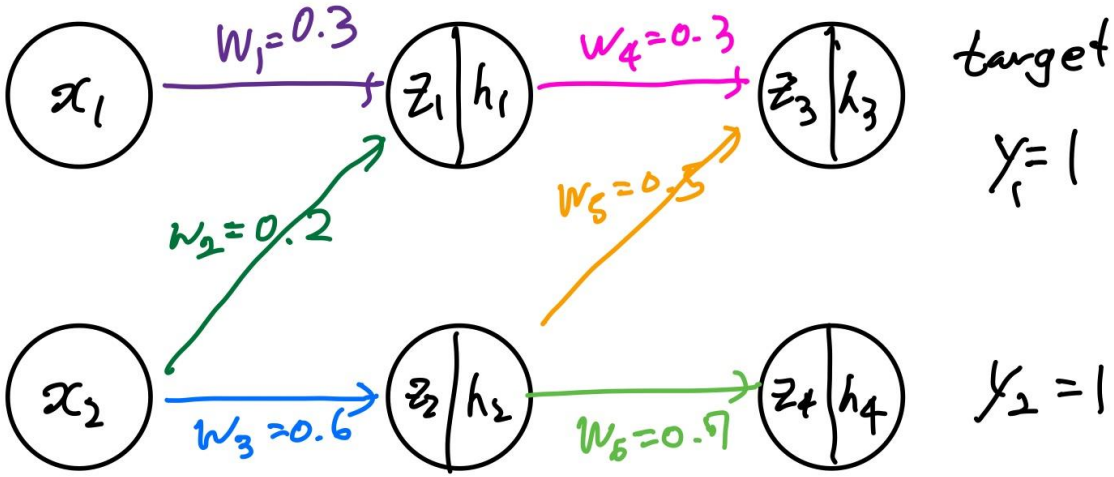

x_1
$w_1=0.3$
z_1 h_1
$w_4=0.3$
z_3 h_3
target
$y_1=1$
$w_2=0.2$
$w_5=0.5$
x_2
$w_3=0.6$
z_2 h_2
$w_6=0.7$
z_4 h_4
$y_2=1$

PyTorch

4. 인공신경망



4. 인공신경망


x_1
$w_1=0.3$
z_1 h_1
$w_4=0.3$
z_3 h_3
target
$y_1=1$
$w_2=0.2$
$w_5=0.5$
x_2
$w_3=0.6$
z_2 h_2
$w_6=0.7$
z_4 h_4
$y_2=1$

4. 인공신경망

PyTorch

In [36]:
```
X = torch.FloatTensor([[0,0],[0,1],[1,0],[1,1]])
Y = torch.FloatTensor([[5],[6],[8],[10]])
```

In [37]:
```
## hidden layers
class simpleNN(nn.Module) :
    def __init__(self) :
        super().__init__()
        self.linear1 = nn.Linear(2,2, bias = True)
        self.linear2 = nn.Linear(2,1, bias = True)
        self.sigmoid = nn.Sigmoid()

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

4. 인공신경망

PyTorch

In [36]:
```
X = torch.FloatTensor([[0,0],[0,1],[1,0],[1,1]])
Y = torch.FloatTensor([[5],[6],[8],[10]])
```

In [37]:
```
## hidden layers
class simpleNN(nn.Module) :
    def __init__(self) :
        super().__init__()
        self.linear1 = nn.Linear(2,2, bias = True)
        self.linear2 = nn.Linear(2,1, bias = True)
        self.sigmoid = nn.Sigmoid()

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

4. 인공신경망

PyTorch

In [38]:
```
DNN = simpleNN()
```

In [39]:
```
## define cost/loss & optimizer
criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(DNN.parameters(), lr = 0.05)

for epoch in range(1000) :
    hypothesis = DNN(X)
    cost = criterion(hypothesis,Y)

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

    print(f'epoch : {epoch}/{1000} , cost : {cost.item()}')
```

4. 인공신경망

```
epoch : 998/1000 , cost : 0.0009952145628631115
epoch : 999/1000 , cost : 0.0009907563216984272
```

```
In [40]: hypothesis
```

```
tensor([[4.9626],
        [6.0347],
        [8.0226],
        [9.9708]], grad_fn=<AddmmBackward>)
```

```
In [42]: Y
```

```
tensor([[ 5.],
        [ 6.],
        [ 8.],
        [10.]])
```