03. Multi-Layer Perceptrons

AILAB
Hanyang Univ.

실습 내용

1. Linear regression 실습
2. Logistic regression 실습
3. Multi-Layer Perceptron 실습

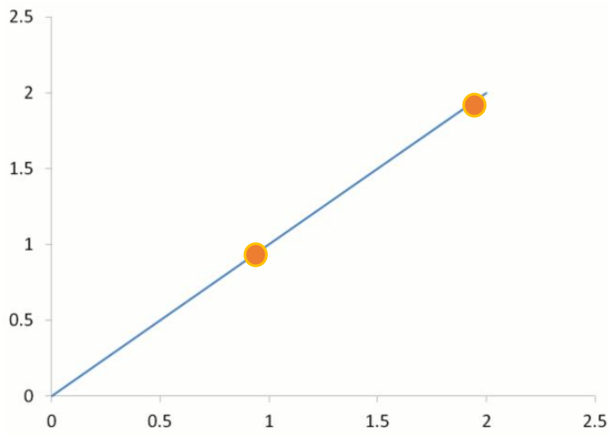
신경망모델 학습 프로세스

1. 데이터 processing
2. model 디자인
   - layer 종류, 개수 및 뉴런 개수 설정
   - 각 layer 마다의 activation function 설정
3. Loss function 설정
4. Optimizer 설정
5. 학습

1. Linear Regression 실습

Linear Regression 실습

- Linear Regression
- $\hat{y} = Wx + b$
  - 파라미터 W, b 를 주어진 데이터 3개로 학습하여 아래와 같은 그래프를 만드는 것이 목표



Input	Output
1	1
2	2
3	3

- Loss function $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

Linear Regression 실습

```
import torch
import torch.nn as nn
import torch.optim as optim
```

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

```
W = torch.rand(1, requires_grad=True) # set model
b = torch.rand(1, requires_grad=True)
```

```
def criterion(y_hat, y):  # set loss function
  return torch.mean((y_hat - y) ** 2)
```

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

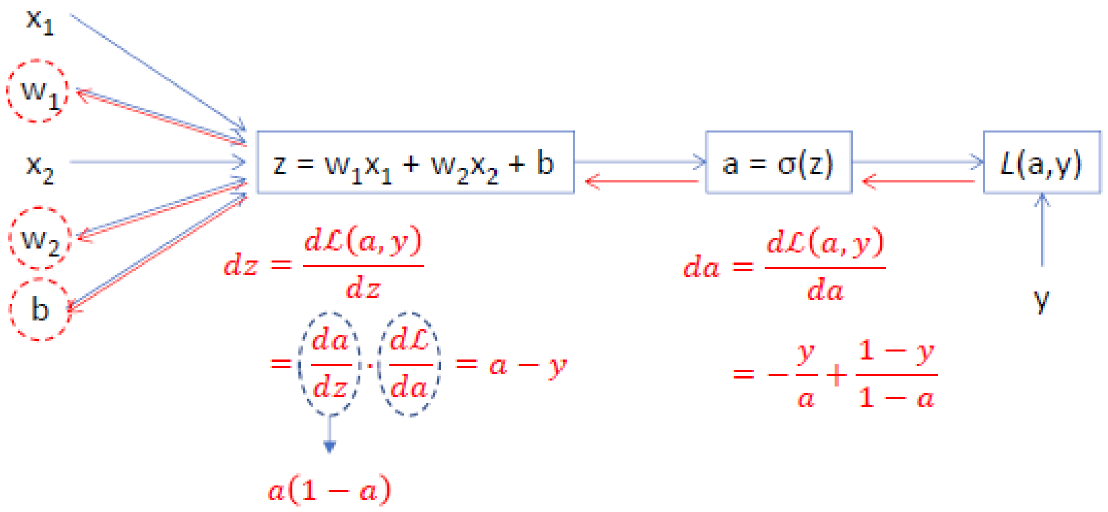
```
epochs = 30
for epoch in range(epochs):
  hypothesis = x_train * W + b  # foward propagation
  cost = criterion(hypothesis, y_train) # get cost

  optimizer.zero_grad()
  cost.backward() # backward propagation
  optimizer.step() # update parameters

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

Linear Regression 실습

1. 데이터 processing
2. model 디자인
3. Loss function 설정
4. Optimizer 설정
5. 학습

```python
import torch
import torch.nn as nn
import torch.optim as optim

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

W = torch.rand(1, requires_grad=True) # set model
b = torch.rand(1, requires_grad=True)

def criterion(y_hat, y):  # set loss function
  return torch.mean((y_hat - y) ** 2)

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

epochs = 30
for epoch in range(epochs):
  hypothesis = x_train * W + b  # foward propagation
  cost = criterion(hypothesis, y_train) # get cost

  optimizer.zero_grad()
  cost.backward() # backward propagation
  optimizer.step() # update parameters

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

Linear Regression 실습



$$dz = \frac{d\mathcal{L}(a,y)}{dz} = \frac{da}{dz} \cdot \frac{d\mathcal{L}}{da} = a - y$$

$$\frac{da}{dz} = a(1-a)$$

$$da = \frac{d\mathcal{L}(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$dw_1 = \frac{d\mathcal{L}(a,y)}{dw_1} = \frac{dz}{dw_1} \cdot \frac{d\mathcal{L}}{dz} = x_1 \cdot dz = x_1 \cdot (a-y)$$

$$dw_2 = \frac{d\mathcal{L}(a,y)}{dw_2} = \frac{dz}{dw_2} \cdot \frac{d\mathcal{L}}{dz} = x_2 \cdot dz = x_2 \cdot (a-y)$$

$$db = \frac{d\mathcal{L}(a,y)}{db} = \frac{dz}{db} \cdot \frac{d\mathcal{L}}{dz} = dz = (a-y)$$

Gradient Descent Algorithm

$$w_1 \leftarrow w_1 - \alpha \cdot dw_1$$

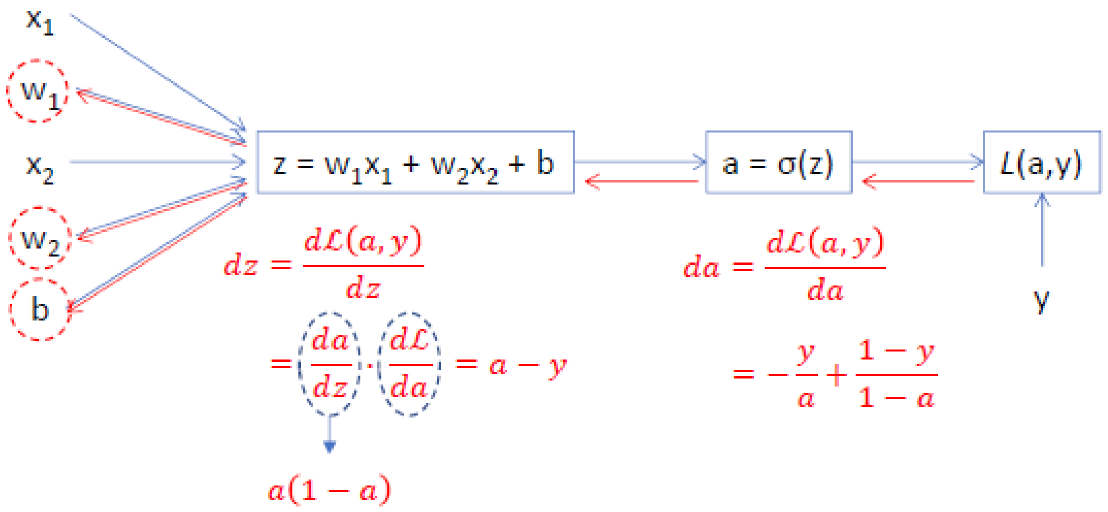
$$w_2 \leftarrow w_2 - \alpha \cdot dw_2$$

$$b \leftarrow b - \alpha \cdot db$$

```python
for epoch in range(epochs):
    hypothesis = x_train * W + b  # foward propagation
    cost = criterion(hypothesis, y_train) # get cost

    optimizer.zero_grad()
    cost.backward() # backward propagation
    optimizer.step() # update parameters

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

Backward propagation
Update parameter

Linear Regression 실습



$$dw_1 = \frac{d\mathcal{L}(a,y)}{dw_1} = \frac{dz}{dw_1}\cdot\frac{d\mathcal{L}}{dz} = x_1 \cdot dz = x_1 \cdot (a-y)$$

$$dw_2 = \frac{d\mathcal{L}(a,y)}{dw_2} = \frac{dz}{dw_2}\cdot\frac{d\mathcal{L}}{dz} = x_2 \cdot dz = x_2 \cdot (a-y)$$

$$db = \frac{d\mathcal{L}(a,y)}{db} = \frac{dz}{db}\cdot\frac{d\mathcal{L}}{dz} = dz = (a-y)$$

Gradient Descent Algorithm

$$w_1 \leftarrow w_1 - \alpha \cdot dw_1$$

$$w_2 \leftarrow w_2 - \alpha \cdot dw_2$$

$$b \leftarrow b - \alpha \cdot db$$

zero grad로
gradient 0으로 초기화

```
for epoch in range(epochs):
  hypothesis = x_train * W + b  # foward propagation
  cost = criterion(hypothesis, y_train) # get cost

  optimizer.zero_grad()
  cost.backward() # backward propagation
  optimizer.step() # update parameters

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

Linear Regression 실습

Module class

```
import torch
import torch.nn as nn
import torch.optim as optim

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

def criterion(y_hat, y):
  return torch.mean(torch.square(y_hat-y))

class LinearRegression(nn.Module):
  def __init__(self, x_in, x_out):
    super(LinearRegression, self).__init__()
    self.linear = nn.Linear(x_in, x_out)
  def forward(self, x):
    return self.linear(x)

model = LinearRegression(1, 1)

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

epochs = 30
for epoch in range(epochs):
  hypothesis = model(x_train)  # forward propagation
  cost = criterion(hypothesis, y_train) # get cost

  optimizer.zero_grad()
  cost.backward() # backward propagation
  optimizer.step() # update parameters

  print('Epoch {:4d}/{} Cost: {:.6f} W: {:.3f}, b: {:.3f}'.format(₩
                epoch, epochs, cost.item(), model.linear.weight.item(), model.linear.bias.item()))
```

Cost function

- import torch
- import torch.nn as nn
- # torch.mean 으로 평균 손실값을 구함
- # torch.square로 거리의 제곱을 손실함수로 적용

cost = torch.mean(torch.square(Y – model))

- # pytorch가 기본 제공하는 cross entropy 함수를 손실함수로 적용

criterion = nn.CrossEntropyLoss() (binary인 경우 BCELoss())

cost = criterion(input=logits, target=y)

Optimizer

- # StochasticGradientDescent(SGD)를 사용해서 손실값을 최소화하는 최적화 수행
- # 0.001 은 learning rate
- optimizer = optim.SGD(model.parameters(), lr=0.01)

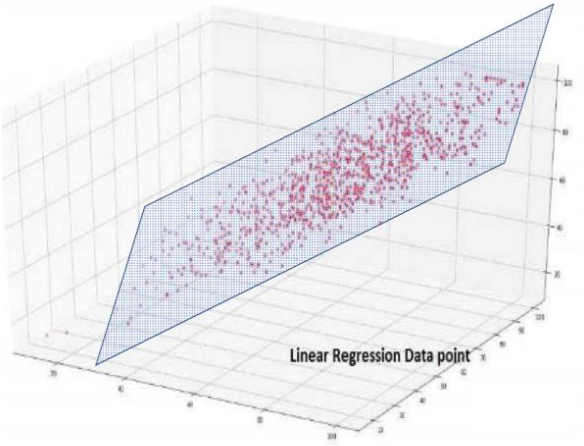
2. Logistic Regression 실습

Logistic Regression

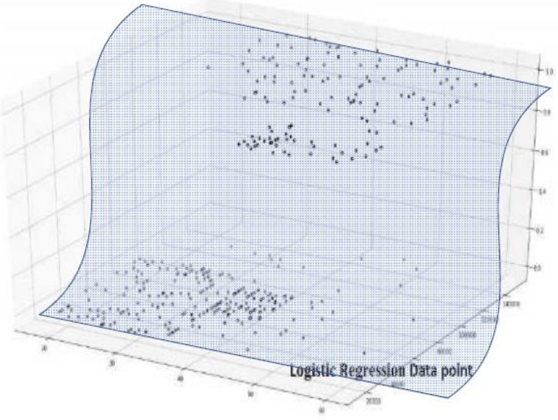
- Linear Regression
  - 예측 모델인 선형회귀는 단순한 linear combination

- Logistic regression
  - 데이터가 복잡하게 표현되면 linear regression으로 표현 불가능

$$y = w_1 x_1 + w_2 x_2 + b$$



$$y = \frac{1}{1+e^{-(w_1 x_1 + w_2 x_2 + b)}}$$



Logistic Regression

- Logistic Regression

$$\hat{y}^{(i)} = \sigma\left(w^T x^{(i)} + b\right)$$

- Activation function

$$\sigma\left(z^{(i)}\right) = \frac{1}{1+e^{-z^{(i)}}}$$

- Loss function

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

활성화 함수(Activation Function)

- 개념 : 입력 신호의 총합을 출력 신호로 변환하는 함수

- 종류
  - Sigmoid function
  - Tanh function
  - ReLU function

  등 이외에 많음

활성화 함수(Activation Function)

Sigmoid

$$g(z) = \frac{1}{1 + e^{-z}}$$

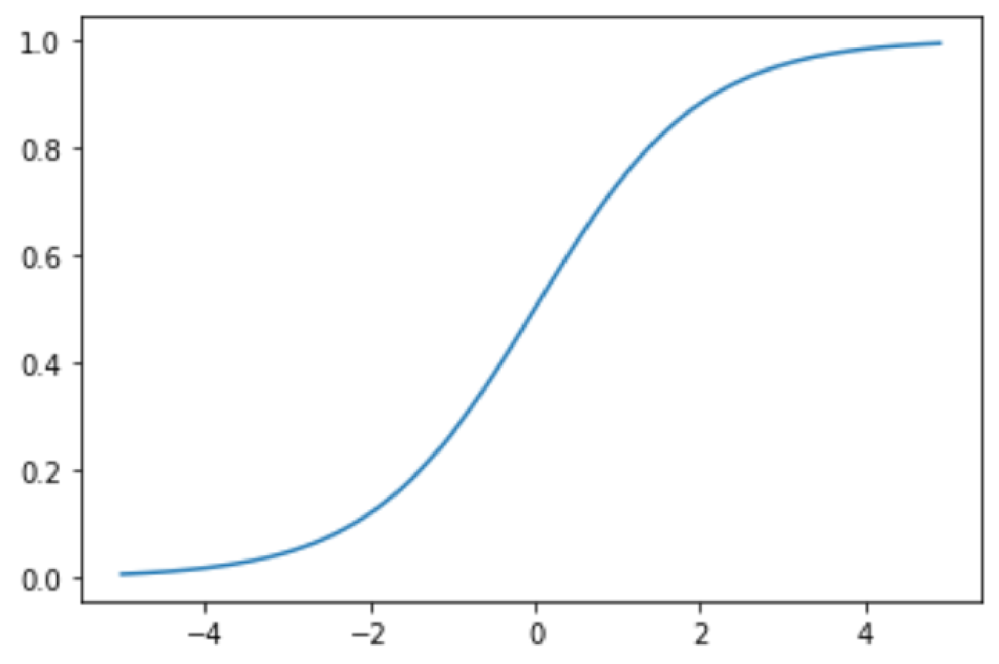
```
import torch
import torch.nn as nn
```

```
activation_function = nn.Sigmoid()
```

```
input = torch.arange(-5.,5.,0.1)
output = activation_function(input)
```

```
import matplotlib.pyplot as plt
plt.plot(input,output)
```

```
[<matplotlib.lines.Line2D at 0x7f5a83969790>]
```



활성화 함수(Activation Function)

Tanh

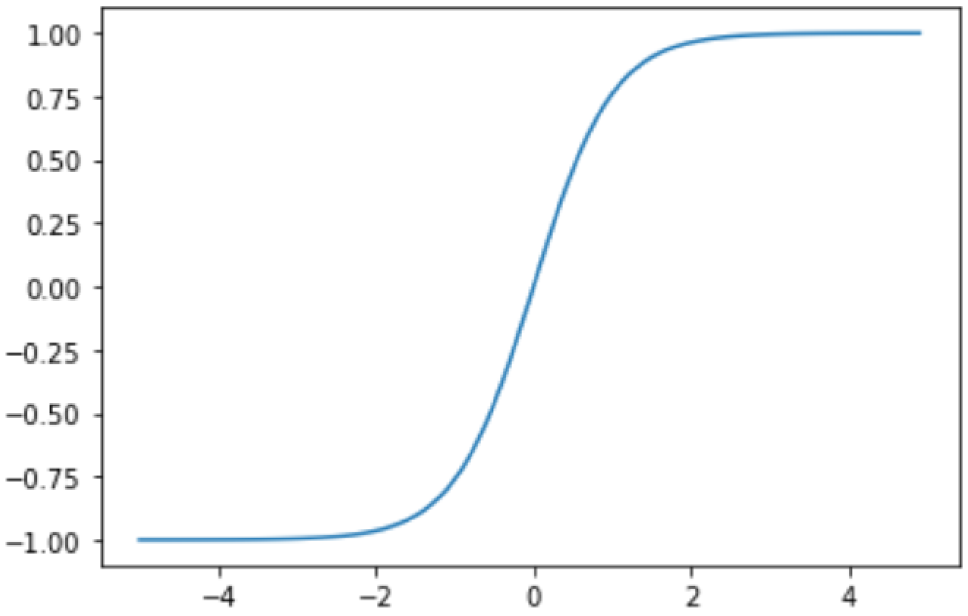
$$g(z) = \tanh(z)$$

$$= \frac{e^{+z} - e^{-z}}{e^{+z} + e^{-z}}$$

```
[ ]  import torch
     import torch.nn as nn
```

```
[ ]  activation_function = nn.Tanh()
```

```
[ ]  input = torch.arange(-5.,5.,0.1)
     output = activation_function(input)
```

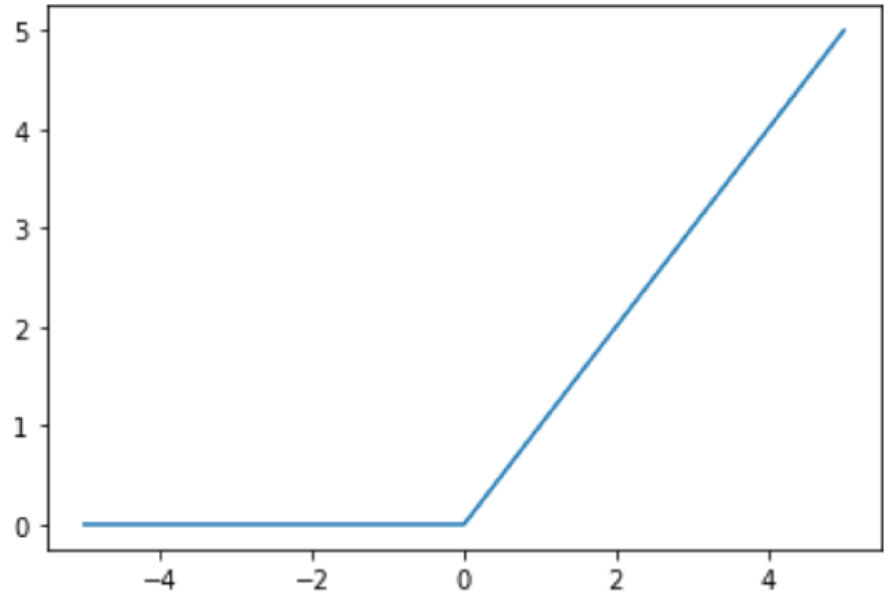


활성화 함수(Activation Function)

ReLU

$$g(z) = \max(0, z)$$

```
[1]  import torch
     import torch.nn as nn

[2]  activation_function = nn.ReLU()

[3]  input = torch.range(-5,5,0.1)
     output = activation_function(input)
```



Logistic Regression

- **AND 문제**

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Logistic Regression

```
import torch
import torch.nn as nn
import torch.optim as optim
```

```
x_train = torch.FloatTensor([[0,0], [0,1], [1,0],[1,1]])
y_train = torch.FloatTensor([[0], [0], [0], [1]])
```

```
class LogisticRegression(nn.Module):
  def __init__(self, x_in, x_out):
    super(LogisticRegression, self).__init__()
    self.linear = nn.Linear(x_in, x_out)
    self.activation = nn.Sigmoid()
  def forward(self, x):
    z = self.linear(x)
    a = self.activation(z)
    return a
```

```
model = LogisticRegression(2, 1).train()
```

```
optimizer = optim.SGD(model.parameters(), lr=0.01) # set optimizer
```

```
criterion = nn.BCELoss()
```

```
epochs = 1000
for epoch in range(epochs):
  model.train()
  hypothesis = model(x_train)  # forward propagation
  cost = criterion(hypothesis+1e-8, y_train) # get cost
  optimizer.zero_grad()
  cost.backward() # backward propagation
  optimizer.step() # update parameters

  if epoch != 0 and epoch % 100 == 0:
    model.eval()
    with torch.no_grad():
      predicts = (model(x_train))
      print('predict with model : {}'.format(predicts))
      print('real value y : {}'.format(y_train))
```

Logistic Regression

```
[1]  import torch
     import torch.nn as nn
     import torch.optim as optim
```

```
[2]  x_train = torch.FloatTensor([[0,0], [0,1], [1,0],[1,1]])
     y_train = torch.FloatTensor([[0], [0], [0], [1]])
```

```
[3]  class LogisticRegression(nn.Module):
       def __init__(self, x_in, x_out):
         super(LogisticRegression, self).__init__()
         self.linear = nn.Linear(x_in, x_out)
         self.activation = nn.Sigmoid()
       def forward(self, x):
         z = self.linear(x)
         a = self.activation(z)
         return a
```

```
[4]  model = LogisticRegression(2, 1).train()
```

```
[5]  optimizer = optim.SGD(model.parameters(), lr=0.1) # set optimizer
```

```
[6]  criterion = nn.BCELoss()
```

```
epochs = 8000
for epoch in range(epochs):
  model.train()
  hypothesis = model(x_train)  # forward propagation
  cost = criterion(hypothesis+1e-8, y_train) # get cost
  optimizer.zero_grad()
  cost.backward() # backward propagation
  optimizer.step() # update parameters

  if epoch != 0 and epoch % 100 == 0:
    model.eval()
    with torch.no_grad():
      predicts = (model(x_train))
      print('predict with model : {}'.format(predicts))
      print('real value y : {}'.format(y_train))
```

Logistic Regression

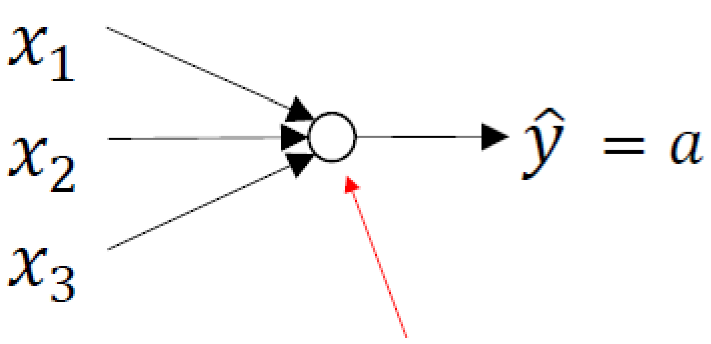
- **결과**

```
predict with model : tensor([[2.5810e-05],
        [2.5644e-02],
        [2.5644e-02],
        [9.6408e-01]])
real value y : tensor([[0.],
        [0.],
        [0.],
        [1.]])
```

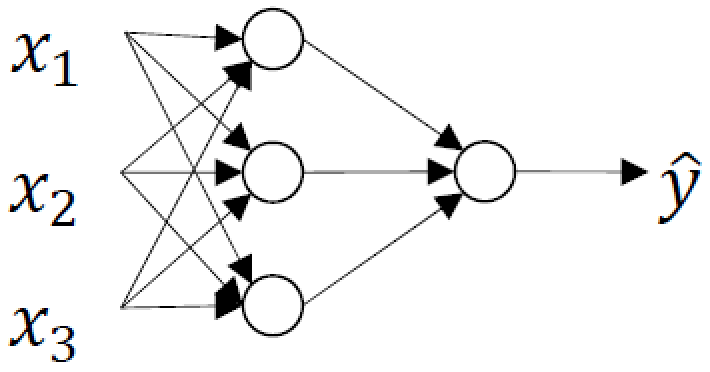
3. Single-Layer Perceptron

Single-Layer Perceptron

- Logistic Regression

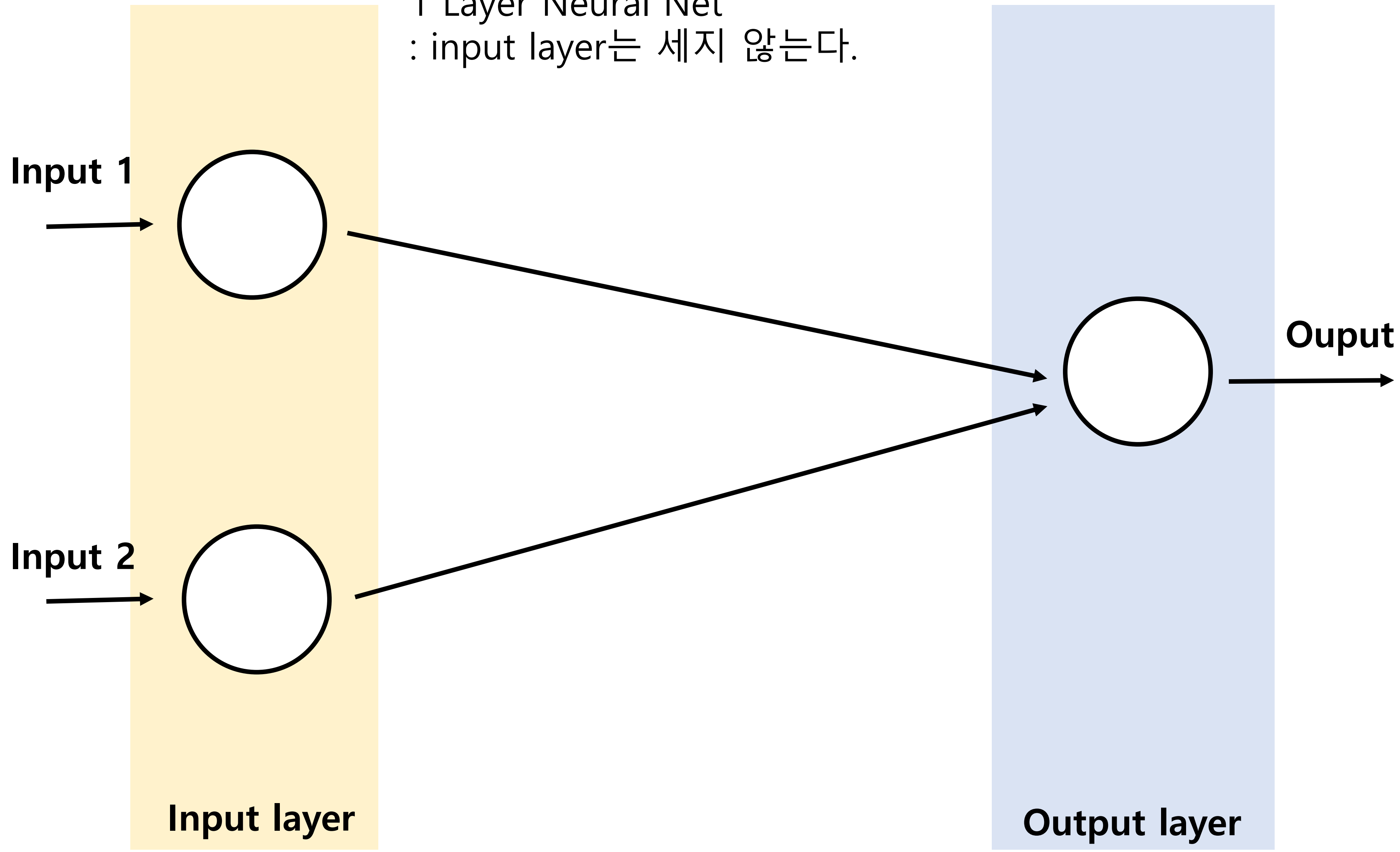


- Neural Network



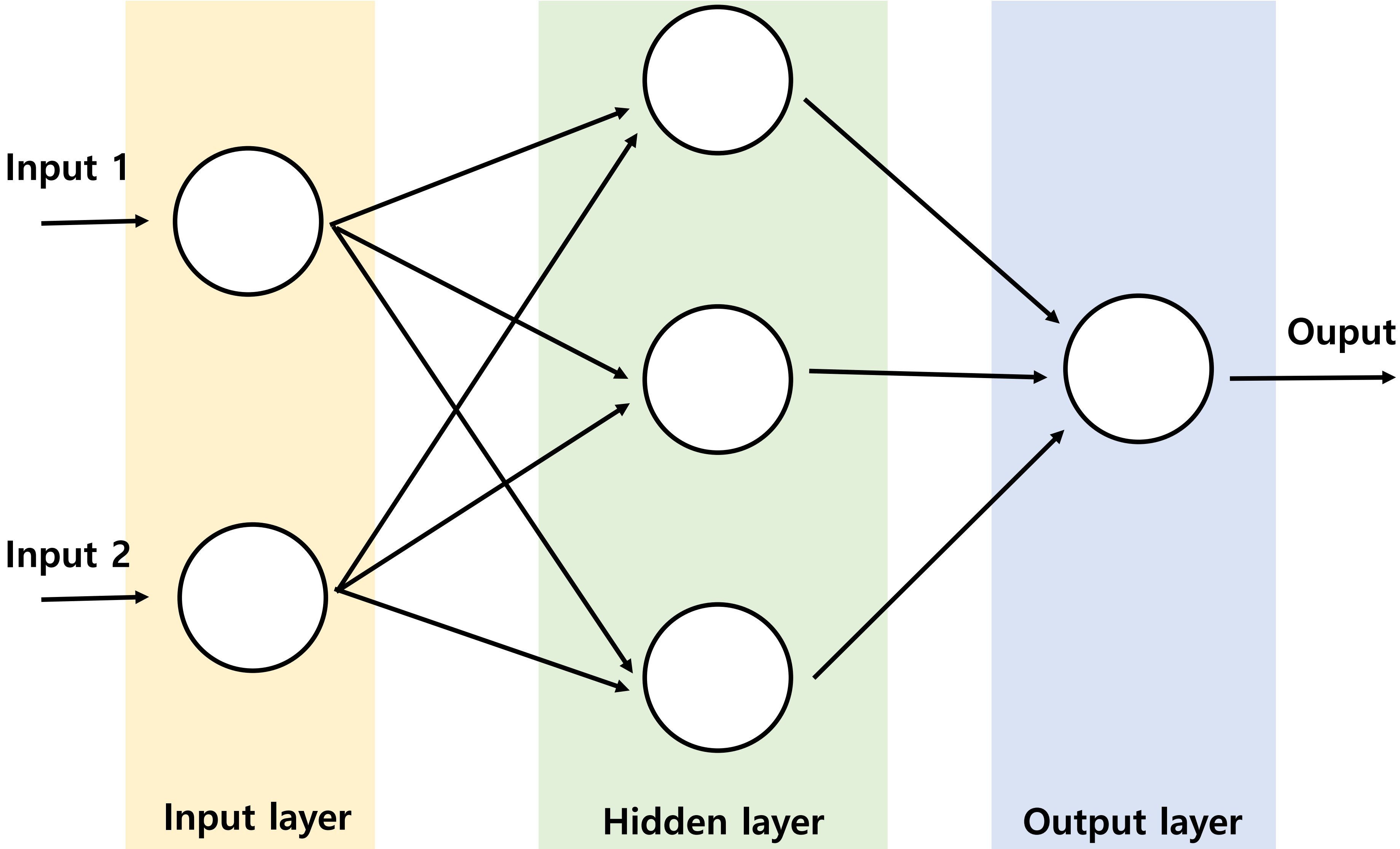
Single-Layer Perceptron 실습

1 Layer Neural Net
: input layer는 세지 않는다.



3. Multi-Layer Perceptron

Multi-Layer Perceptron

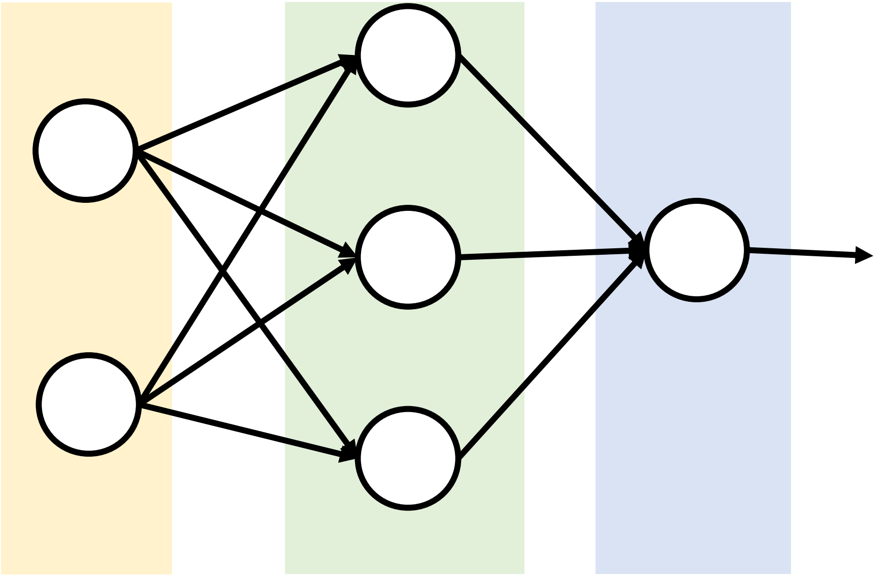


XOR 문제

- 단순한 Regression으로 풀리지 않음 → Neural Net으로 해결

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Multi-Layer Perceptron



```python
class MultiLayerPerceptron(nn.Module):
  def __init__(self):
    super(MultiLayerPerceptron, self).__init__()
    self.linear1 = nn.Linear(2, 3)
    self.activation = nn.Sigmoid()
    self.linear2 = nn.Linear(3, 1)

  def forward(self, x):
    z1 = self.linear1(x)
    a1 = self.activation(z1)

    z2 = self.linear2(a1)
    a2 = self.activation(z2)
    return a2
```

Multi-Layer Perceptron

- 결과

```
        [0.]])
predict with model : tensor([[1.8329e-04],
        [9.9974e-01],
        [9.9968e-01],
        [3.3466e-04]])
real value y : tensor([[0.],
        [1.],
        [1.],
        [0.]])
```

실습 내용

1. Linear regression 실습
2. Logistic regression 실습
3. Multi-Layer Perceptron 실습