파이썬 입문

한국폴리텍대학

2023.06.16

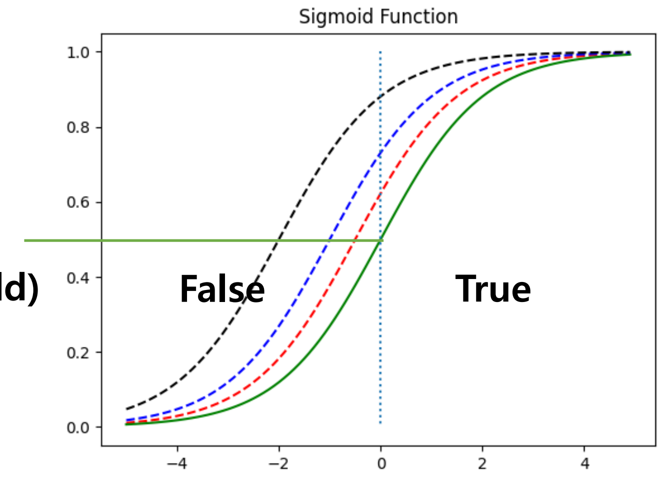
PyTorch – 로지스틱회귀 (Logistic Regression)

두 개의 선택지 중에서 정답을 고르는 문제: 이진 분류 (Binary Classification)

예) 제조 부품 테스트에서 점수 60점 이상이면 정상, 미만이면 불량 판정

H(x) = sigmoid(Wx + b)   # 선형회귀 : H(x) = Wx + b

임계값
(threshold)



w : 경사
b : 좌우이동

시그모이드 함수 (Sigmoid function)

$$H(x) = \frac{1}{1+e^{-(wx+b)}} = sigmoid(wx+b) = \sigma(wx+b)$$

e : 자연상수(exponential), 2.718281...
w : 가중치, weight
b : 편향, bias

- 선형회귀, 다중선형회귀
- 로지스틱회귀, 다중로지스틱회귀
- 소프트맥스회귀

```
import torch
import matplotlib.pyplot as plt

def sigmoid(x):
    return 1 / (1 + torch.exp(-x))

x = torch.arange(-5.0, 5.0, 0.1)
y1 = sigmoid(0.5 * x)
y2 = sigmoid(x)
y3 = sigmoid(2 * x)
y4 = sigmoid(10 * x)

plt.plot(x, y1, 'r', linestyle='--') # w = 0.5
plt.plot(x, y2, 'g') # w = 1
plt.plot(x, y3, 'b', linestyle='--') # w = 2
plt.plot(x, y4, 'black', linestyle='--') # w = 10
plt.plot([0,0],[1.0,0.0], ':') # 가운데 점선 추가
plt.title('Sigmoid Function')
plt.show()
```

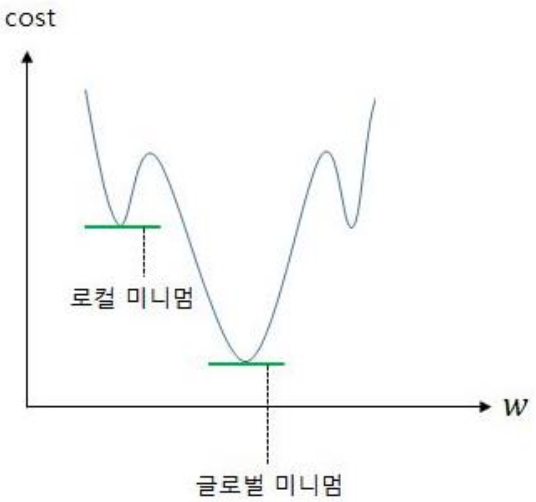
```
import torch
import matplotlib.pyplot as plt
def sigmoid(x):
    return 1 / (1 + torch.exp(-x))

x = torch.arange(-5.0, 5.0, 0.1)
y1 = sigmoid(x + 0.5)
y2 = sigmoid(x)
y3 = sigmoid(x + 1.0)
y4 = sigmoid(x + 2.0)

plt.plot(x, y1, 'r', linestyle='--')
plt.plot(x, y2, 'g')
plt.plot(x, y3, 'b', linestyle='--')
plt.plot(x, y4, 'black', linestyle='--')
plt.plot([0,0],[1.0,0.0], ':')
plt.title('Sigmoid Function')
plt.show()
```

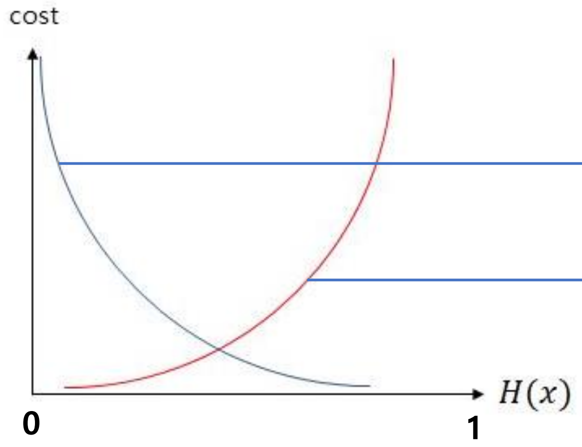
PyTorch – 로지스틱회귀 (Logistic Regression)

비용함수 (Cost Function), 또는 손실함수 (Loss Function)

이진 크로스 엔트로피 (Binary Cross Entropy)

- 선형회귀, 다중선형회귀 : MSE(Mean Squared Error)





실제값이 1인 경우
H(x)가 1에 가까울수록 cost는 0에 가깝다

실제값이 0인 경우
H(x)가 0에 가까울수록 cost는 0에 가깝다

$$cost(W) = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}logH(x^{(i)}) + (1-y^{(i)})log(1-H(x^{(i)}))]$$

$$W := W - \alpha\frac{\partial}{\partial W}cost(W)$$

PyTorch – 다중로지스틱회귀 (Multivariate Logistic Regression)

구현

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

torch.manual_seed(1)

x_data = [[1], [2], [3], [4], [5], [6]] # 로지스틱회귀
# x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]  #다중로지스틱회귀
y_data = [[0], [0], [0], [1], [1], [1]]
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)

# Model 가중치, 편향 초기화
W = torch.zeros((1, 1), requires_grad=True)
# W = torch.zeros((2, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)

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

```
epochs = 1000
for epoch in range(epochs + 1):

    # cost
    hypothesis = torch.sigmoid(W * x_data + b)
    # hypothesis = torch.sigmoid(x_data.matmul(W) + b)
    cost = -(y_train * torch.log(hypothesis) + (1 - y_train) *
        torch.log(1 - hypothesis)).mean()

    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch, epochs, cost.item() ))

prediction = hypothesis >= torch.FloatTensor([0.5])  # 임계값 (threshold)
print(prediction)

print(W)
print(b)
```

PyTorch – 다중로지스틱회귀 (Multivariate Logistic Regression)

numpy, torch dot product

TORCH.DOT

torch.dot(*input*, *other*, *, *out=None*) → Tensor

Computes the dot product of two 1D tensors.

• NOTE

Unlike NumPy's dot, torch.dot intentionally only supports computing the dot product of two 1D tensors with the same number of elements.

Parameters:

- **input** (*Tensor*) – first tensor in the dot product, must be 1D.
- **other** (*Tensor*) – second tensor in the dot product, must be 1D.

Keyword Arguments:

out (*Tensor, optional*) – the output tensor.

Example:

```
>>> torch.dot(torch.tensor([2, 3]), torch.tensor([2, 1]))
tensor(7)
```

https://pytorch.org/docs/stable/generated/torch.dot.html

numpy dot product는 다차원 x 다차원 가능

torch dot product는 1차원 x 1차원만 가능

해결책

1. torch.matmul 사용
2. numpy.ndarray로 계산 후 tensor로 변경

   - tensor_data.numpy() # numpy.ndarray로 변환
   - a = numpy.dot(x, w) # dot product계산
   - torch.Tensor(a) # tensor로 변환
   - requires_grad = True인 경우, False로 변경 -> 수행 -> True로 변경

PyTorch – 다중로지스틱회귀 (Multivariate Logistic Regression)

nn.Module로 구현 (nn.Linear, nn.Sigmoid) - refactoring

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

torch.manual_seed(1)

x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]  #다중로지스틱회귀
#x_data = [[1], [2], [3], [4], [5], [6]] # 로지스틱회귀
y_data = [[0], [0], [0], [1], [1], [1]]
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)

model = nn.Sequential(
    nn.Linear(2, 1),  # 다중로지스틱회귀
    # nn.Linear(1, 1),  # 로지스틱회귀
    # nn.Linear(1, 1, bias=False),  # 편향 사용 여부
    nn.Sigmoid()
)
model(x_train)
```

```
optimizer = optim.SGD(model.parameters(), lr=0.1)
epochs = 5000
for epoch in range(epochs + 1):
    hypothesis = model(x_train)
    cost = F.binary_cross_entropy(hypothesis, y_train)
    # cost = F.mse_loss(hypothesis, y_train)  # 선형회귀
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
    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, epochs, cost.item(), accuracy * 100, ))
model(x_train)
print(list(model.parameters()))
```

PyTorch – 다중로지스틱회귀 (Multivariate Logistic Regression)

class로 구현 - refactoring

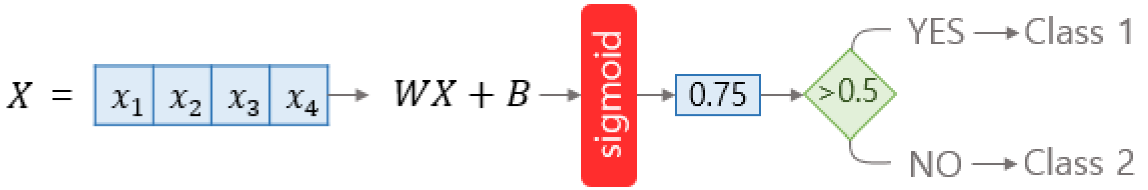
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)

x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]  #다중로지스틱회귀
#x_data = [[1], [2], [3], [4], [5], [6]] # 로지스틱회귀
y_data = [[0], [0], [0], [1], [1], [1]]
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)

class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(2, 1)     # 다중로지스틱회귀
        # self.linear = nn.Linear(1, 1)   # 로지스틱회귀
        self.sigmoid = nn.Sigmoid()
    def forward(self, x):       # forward propagation
        return self.sigmoid(self.linear(x))

model = BinaryClassifier()
```

```
optimizer = optim.SGD(model.parameters(), lr=0.1)
epochs = 5000
for epoch in range(epochs + 1):
    hypothesis = model(x_train)
    cost = F.binary_cross_entropy(hypothesis, y_train)
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
    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, epochs, cost.item(), accuracy * 100, ))
model(x_train)
print(list(model.parameters()))
```

- __init__()
- forward()

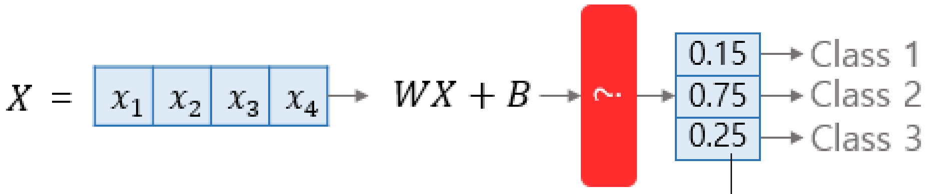
PyTorch – 소프트맥스회귀 (Softmax Regression)

다중 클래스 분류 (Multi-class Classification)



가설 : $H(X) = sigmoid(WX + B)$

→ 로지스틱회귀



가설 : $H(X) = softmax(WX + B)$

→ 확률의 합은 1

→ 소프트맥스회귀

- 2개 이상의 class 분류
- 활성화함수(activation function)가 다르다

다중 클래스 분류 (Multi-class Classification)

소프트맥스 함수

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad for\ i = 1, 2, \ldots k$$

입력: k차원의 벡터 z

비용 함수 : 크로스 엔트로피 (cross entropy, categorical cross entropy)

$$cost(W) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} y_j^{(i)}\ log(p_j^{(i)})$$

PyTorch – 소프트맥스회귀 (Softmax Regression)

다중 클래스 분류 (Multi-class Classification)

구현

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

torch.manual_seed(1)

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)

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

```
# 모델 초기화
W = torch.zeros((4, 3), requires_grad=True)
b = torch.zeros((1, 3), requires_grad=True)
# optimizer 설정
optimizer = optim.SGD([W, b], lr=0.1)
epochs = 1000
for epoch in range(epochs + 1):
    z = x_train.matmul(W) + b
    cost = F.cross_entropy(z, y_train)   # 비용함수에 소프트맥스함수가 포함
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format( epoch, epochs, cost.item() ))
```

```
print(z)
```

크로스 엔트로피
가장 큰 값이 예측결과 (cost가 0일 때, 모두 더하면 0)

```
tensor([[-4.3709, -0.3246,  4.5068],
        [-3.0665,  0.6256,  2.5122],
        [-7.2341,  3.7583,  4.7448],
        [-6.3326,  4.4897,  3.0981],
        [ 0.6266,  1.4420, -2.1420],
        [ 2.1208,  3.8136, -3.9867],
        [ 2.7828,  2.3761, -4.5870],
        [ 4.1330,  2.8725, -6.3362]], grad_fn=<AddmmBackward0>)
```

PyTorch – 소프트맥스회귀 (Softmax Regression)

nn.Module로 구현 - refactoring

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

torch.manual_seed(1)

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)

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

```
# 모델 초기화
model = nn.Linear(4, 3)

# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=0.1)

epochs = 1000
for epoch in range(epochs + 1):
    prediction = model(x_train)
    cost = F.cross_entropy(prediction, y_train)             # softmax
    # cost = F.binary_cross_entropy(prediction, y_train)   # logistic
    # cost = F.mse_loss(prediction, y_train)                # linear
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
    if epoch % 100 == 0: print('Epoch {:4d}/{} Cost: {:.6f}'.format(
        epoch, epochs, cost.item() ))

print(prediction)
```

```
tensor([[-4.3709, -0.3246,  4.5068],
        [-3.0665,  0.6256,  2.5122],
        [-7.2341,  3.7583,  4.7448],
        [-6.3326,  4.4897,  3.0981],
        [ 0.6266,  1.4420, -2.1420],
        [ 2.1208,  3.8136, -3.9867],
        [ 2.7828,  2.3761, -4.5870],
        [ 4.1330,  2.8725, -6.3362]], grad_fn=<AddmmBackward0>)
```

PyTorch – 소프트맥스회귀 (Softmax Regression)

class로 구현 - refactoring

```
class SoftmaxClassifierModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(4, 3) # Output이 3!
    def forward(self, x):
        return self.linear(x)
model = SoftmaxClassifierModel()
```

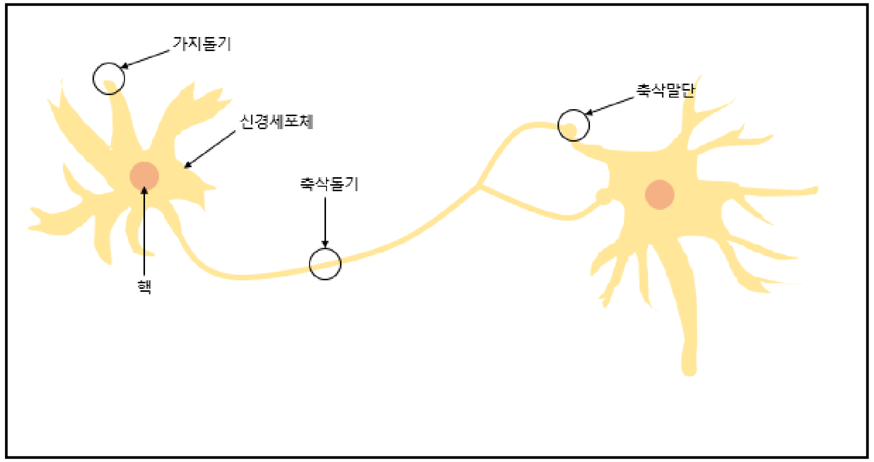
```
optimizer = optim.SGD(model.parameters(), lr=0.1)
epochs = 1000
for epoch in range(epochs + 1):
    prediction = model(x_train)
    cost = F.cross_entropy(prediction, y_train)
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(epoch, epochs, cost.item() ))
```

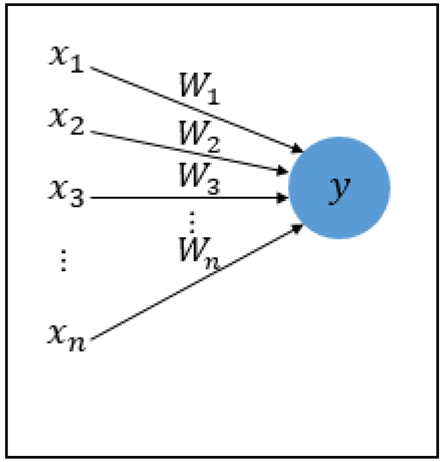
```
print(prediction)
```

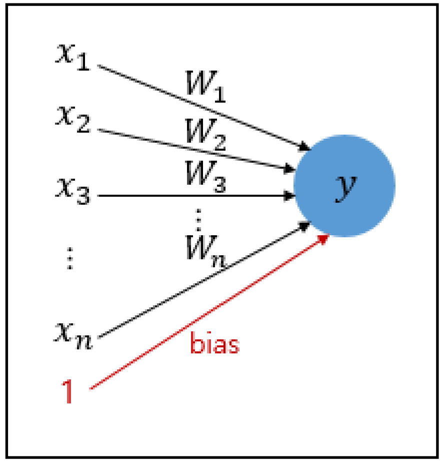
```
tensor([[-4.3709, -0.3246,  4.5068],
        [-3.0665,  0.6256,  2.5122],
        [-7.2341,  3.7583,  4.7448],
        [-6.3326,  4.4897,  3.0981],
        [ 0.6266,  1.4420, -2.1420],
        [ 2.1208,  3.8136, -3.9867],
        [ 2.7828,  2.3761, -4.5870],
        [ 4.1330,  2.8725, -6.3362]], grad_fn=<AddmmBackward0>)
```

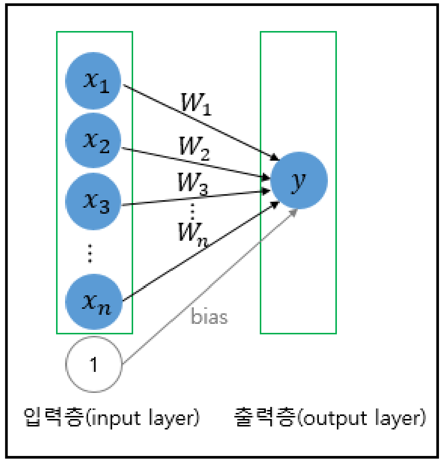
PyTorch – 단층 퍼셉트론 (Single-Layer Perceptron)

퍼셉트론(Perceptron): 초기 인공신경망 모델









인공신경망(퍼셉트론)

활성화함수(계단함수)

$$f(\sum_i^n W_i x_i + b)$$

PyTorch – 단층 퍼셉트론 (Single-Layer Perceptron)

퍼셉트론(Perceptron): 초기 인공신경망 모델

```
import torch
import torch.nn as nn
X = torch.FloatTensor([[0, 0], [0, 1], [1, 0], [1, 1]])
Y = torch.FloatTensor([[0], [1], [1], [0]])

linear = nn.Linear(2, 1, bias=True)
sigmoid = nn.Sigmoid()
model = nn.Sequential(linear, sigmoid)

criterion = torch.nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1)

for step in range(10001):
    optimizer.zero_grad()
    hypothesis = model(X)
    cost = criterion(hypothesis, Y)
    cost.backward()
    optimizer.step()
    if step % 100 == 0:
        print(step, cost.item())
```

```
hypothesis = model(X)
predicted = (hypothesis > 0.5).float()
accuracy = (predicted == Y).float().mean() * 100
print('모델의 출력값(Hypothesis): ', hypothesis)
print('모델의 예측값(Predicted): ', predicted)
print('실제값(Y): ', Y)
print('정확도(Accuracy): ', accuracy.item())
```

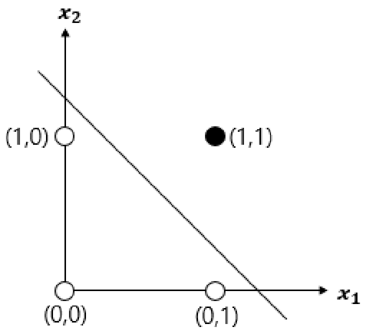
단층 퍼셉트론은 XOR문제를 풀 수 없다

XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

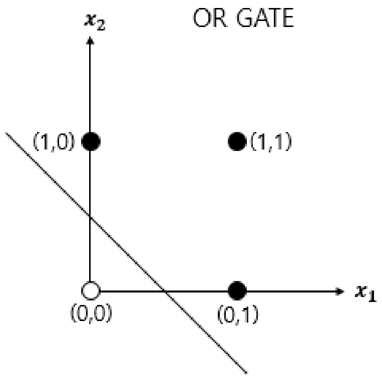
PyTorch – 단층 퍼셉트론 (Single-Layer Perceptron)

AND

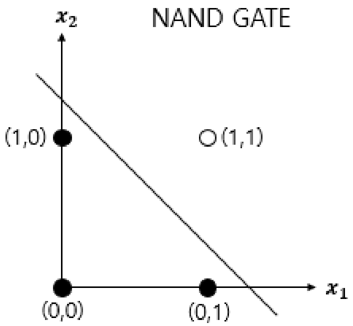


x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

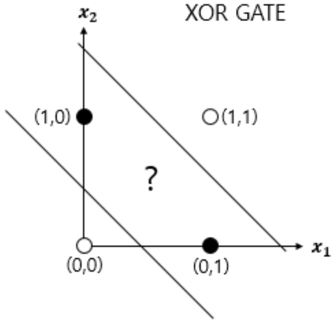
OR



NAND

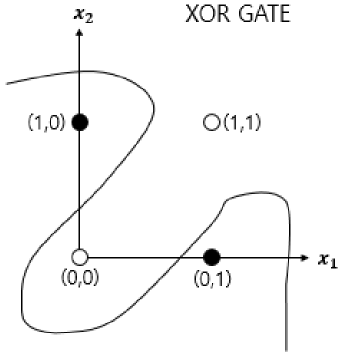


XOR



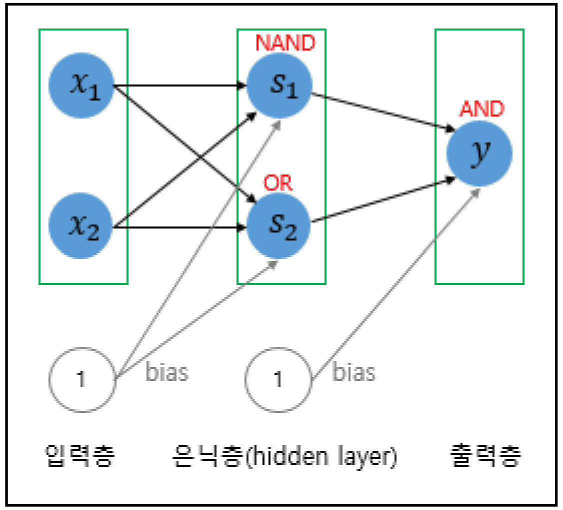
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

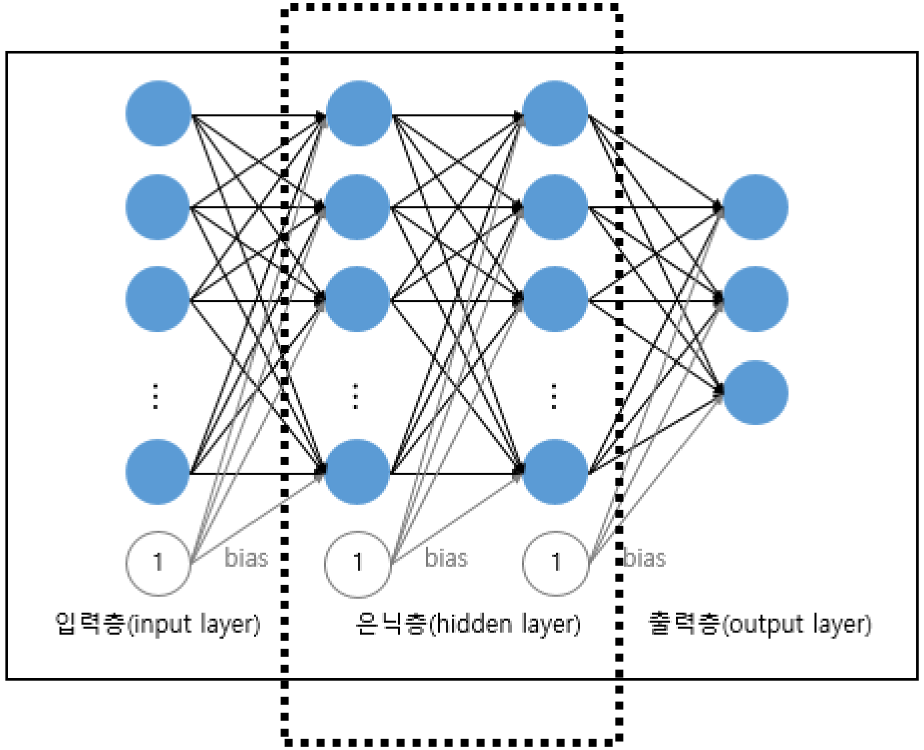
XOR



PyTorch – 다층 퍼셉트론 (MLP, Multi-Layer Perceptron)

다층 퍼셉트론(MLP, Multi-Layer Perceptron)





은닉층(Hidden Layer)이 2개 이상인 신경망:
심층 신경망(DNN, Deep Neural Network)

심층 신경망 학습: 딥러닝(Deep Learning)

PyTorch – 다층 퍼셉트론 (MLP, Multi-Layer Perceptron)

```
import torch
import torch.nn as nn
X = torch.FloatTensor([[0, 0], [0, 1], [1, 0], [1, 1]])
Y = torch.FloatTensor([[0], [1], [1], [0]])

model = nn.Sequential(
        nn.Linear(2, 10, bias=True), # input_layer=2, hidden_layer1=10
        nn.Sigmoid(),
        nn.Linear(10, 10, bias=True), # hidden_layer1=10, hidden_layer2=10
        nn.Sigmoid(),
        nn.Linear(10, 10, bias=True), # hidden_layer2=10, hidden_layer3=10
        nn.Sigmoid(),
        nn.Linear(10, 1, bias=True), # hidden_layer3=10, output_layer=1
        nn.Sigmoid()
)

criterion = torch.nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1)

for epoch in range(10001):
        optimizer.zero_grad() # forward 연산
        hypothesis = model(X) # 비용 함수
        cost = criterion(hypothesis, Y)
        cost.backward()
        optimizer.step()
        if epoch % 100 == 0:
                print(epoch, cost.item())
```

```
hypothesis = model(X)
predicted = (hypothesis > 0.5).float()      # threshold 0.5
accuracy = (predicted == Y).float().mean() * 100
print('모델의 출력값(Hypothesis): ', hypothesis)
print('모델의 예측값(Predicted): ', predicted)
print('실제값(Y): ', Y)
print('정확도(Accuracy): ', accuracy.item())
```

다층 퍼셉트론은 XOR문제를 풀 수 있다

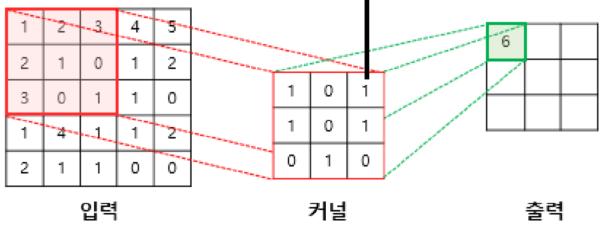
PyTorch – 합성곱 신경망 (CNN, Convolutional Neural Network)

kernel, window, filter, mask

가중치 (weight)

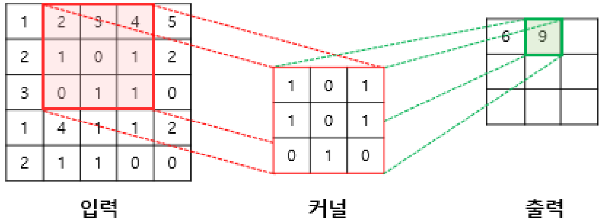
합성곱 (Convolution)

1. 첫번째 스텝



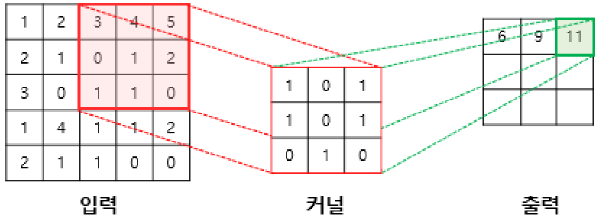
(1×1) + (2×0) + (3×1) + (2×1) + (1×0) + (0×1) + (3×0) + (0×1) + (1×0) = 6

2. 두번째 스텝



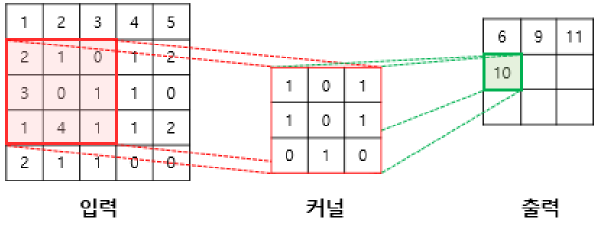
(2×1) + (3×0) + (4×1) + (1×1) + (0×0) + (1×1) + (0×0) + (1×1) + (1×0) = 9

3. 세번째 스텝



(3×1) + (4×0) + (5×1) + (0×1) + (1×0) + (2×1) + (1×0) + (1×1) + (0×0) = 11

4. 네번째 스텝



(2×1) + (1×0) + (0×1) + (3×1) + (0×0) + (1×1) + (1×0) + (4×1) + (1×0) = 10

6	9	11
10	4	4
7	7	4

특징맵 (Feature Map)

PyTorch – 합성곱 신경망 (CNN, Convolutional Neural Network)

합성곱 (Convolution)

```
import torch
import torch.nn as nn
# batch size × channel × height × width의 Tensor 선언
x = torch.full((1, 1, 5), 2.)
print(x)

# in_channels, out_channels, kernel_size, padding, stride
conv = nn.Conv1d(1, 1, 3, padding=0, stride=1)
nn.init.uniform_(conv.weight, 1,1)
# nn.init.constant_(conv.weight, 1,1)
nn.init.uniform_(conv.bias, 3,3)
print(conv.weight, conv.bias)

print(conv(x))
```

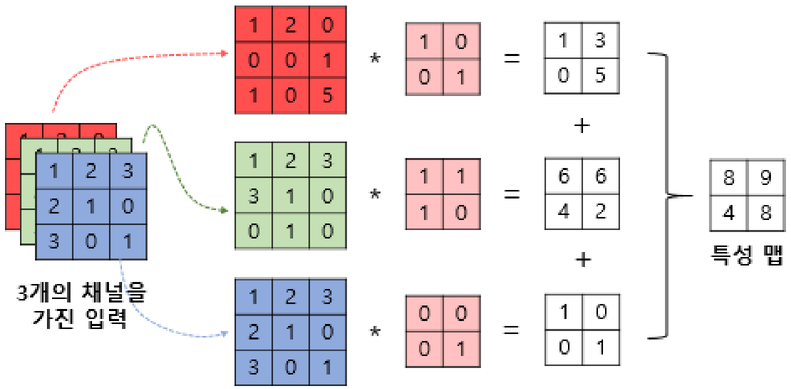
1D Convolution

```
import torch
import torch.nn as nn
# batch size × channel × height × width의 Tensor 선언
x = torch.full((1, 1, 5, 5), 2.)
print(x)

# in_channels, out_channels, kernel_size, padding, stride
conv = nn.Conv2d(1, 1, 3, padding=0, stride=1)
nn.init.uniform_(conv.weight, 1,1)
nn.init.uniform_(conv.bias, 3,3)
print(conv.weight, conv.bias)

print(conv(x))
```

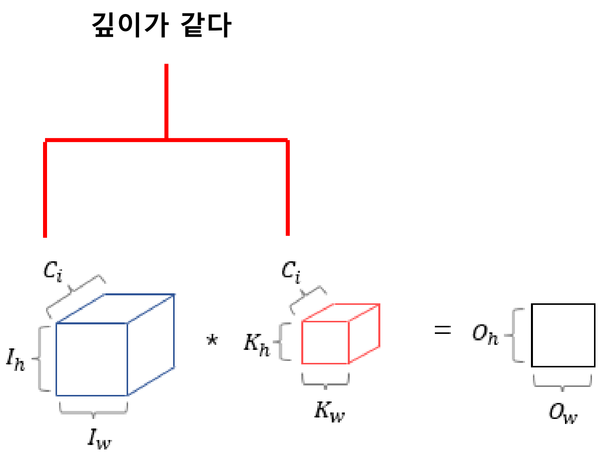
2D Convolution

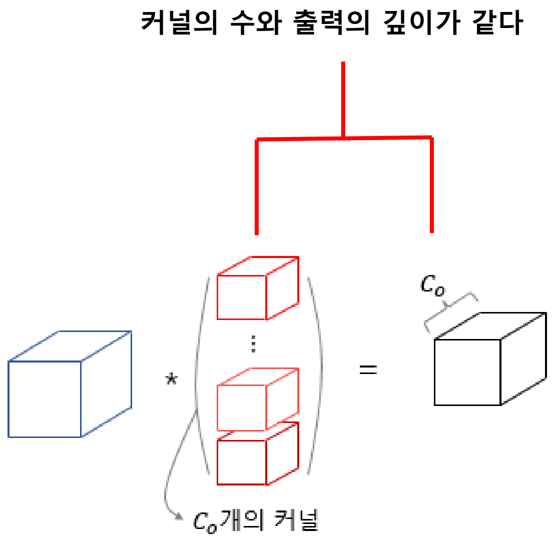
PyTorch – 합성곱 신경망 (CNN, Convolutional Neural Network)

합성곱 (Convolution)



채널 간 합성곱 연산





입력의 깊이(depth)와 필터의 깊이(depth)는 같아야 한다

입력: 3 x 3 x 3
필터: 3 x 2 x 2
출력: 1 x 2 x 2

예) 흑백 이미지: 256 x 256 x 1
컬러 이미지: 256 x 256 x 3

1	2	3	4	5
2	1	0	1	2
3	0	1	1	0
1	4	1	1	2
2	1	1	0	0

패딩 전

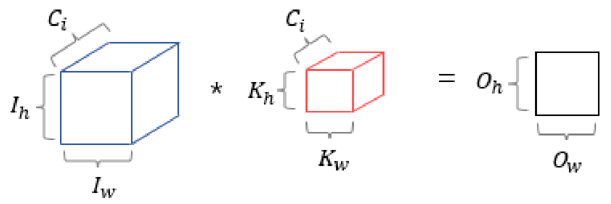
0	0	0	0	0	0	0
0	1	2	3	4	5	0
0	2	1	0	1	2	0
0	3	0	1	1	0	0
0	1	4	1	1	2	0
0	2	1	1	0	0	0
0	0	0	0	0	0	0

패딩 후

PyTorch – 합성곱 신경망 (CNN, Convolutional Neural Network)

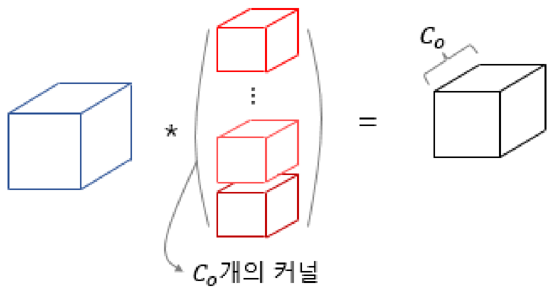
합성곱 (Convolution)

```
import torch
import torch.nn as nn
# batch size × channel × height × width의 Tensor 선언
x = torch.full((1, 3, 5, 5), 2.)
print(x)

# in_channels, out_channels, kernel_size, padding, stride
conv = nn.Conv2d(3, 1, 3, padding=0, stride=1)
nn.init.uniform_(conv.weight, 1,1)
nn.init.uniform_(conv.bias, 3,3)
print(conv.weight, conv.bias)

print(conv(x))
```



2D convolution

```
import torch
import torch.nn as nn
# batch size × channel × height × width의 Tensor 선언
x = torch.full((1, 3, 5, 5), 2.)
print(x)

# in_channels, out_channels, kernel_size, padding, stride
conv = nn.Conv2d(3, 3, 3, padding=0, stride=1)
nn.init.uniform_(conv.weight, 1,1)
nn.init.uniform_(conv.bias, 3,3)
print(conv.weight, conv.bias)

print(conv(x))
```

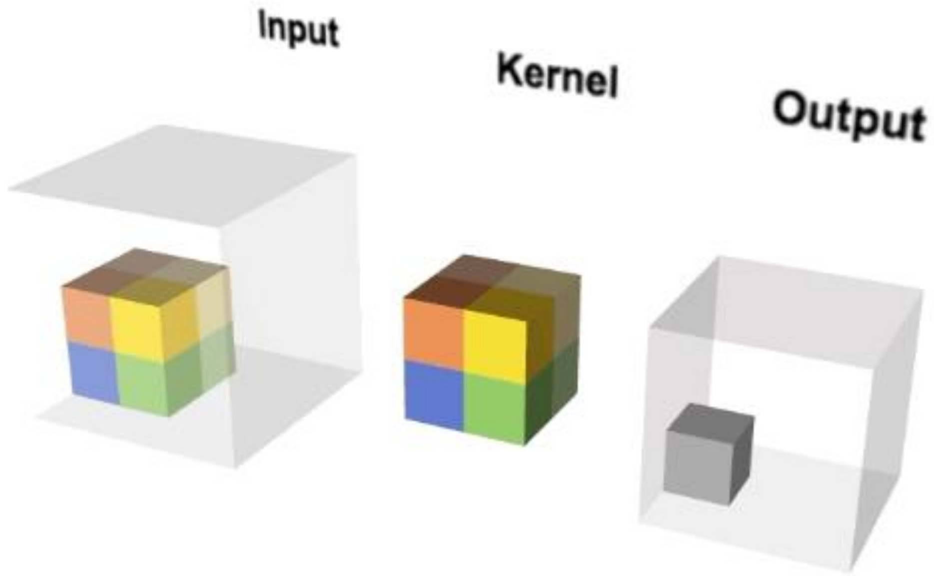


2D convolution

PyTorch – 합성곱 신경망 (CNN, Convolutional Neural Network)

합성곱 (Convolution)

```
import torch
import torch.nn as nn
# batch size × channel × height × width의 Tensor 선언
x = torch.full((1, 1, 5, 5, 5), 2.)
print(x)

# in_channels, out_channels, kernel_size, padding, stride
conv = nn.Conv3d(1, 1, 3, padding=0, stride=1)
nn.init.uniform_(conv.weight, 1,1)
nn.init.uniform_(conv.bias, 3,3)
print(conv.weight, conv.bias)

print(conv(x))
```



3D Convolution

PyTorch – 합성곱 신경망 (CNN, Convolutional Neural Network)

합성곱 (Convolution)

```
model = nn.Conv2D(1, 1, 5)    # in_channels, out_channels, kernel_size
print(list(model.parameters()))

# 가중치(weight), 편향(bias) 초기화
nn.init.uniform_(model.weight, 0, 5)
# nn.init.constant_(model.weight, 2)
# nn.init.ones_(model.weight)
# nn.init.zeros_(model.weight)
# nn.init.eye_(model.weight)
# nn.init.normal_(model.weight)
# nn.init.xavier_uniform_(model.weight, gain=1.0)
# nn.init.xavier_normal_(model.weight, gain=1.0)
# nn.init.kaiming_uniform_(model.weight, mode='fan_in', nonlinearity='relu')
# nn.init.kaiming_normal_(model.weight, mode='fan_out', nonlinearity='relu')
# model.weight.data = nn.Parameter(torch.Tensor([ [ [ [1,1,1,1,1],
                                                      [1,1,1,1,1],
                                                      [1,1,1,1,1],
                                                      [1,1,1,1,1],
                                                      [1,1,1,1,1] ] ] ]))
# model.bias.data = nn.Parameter(torch.Tensor([ [ [ [0.5] ] ] ]))
```

```
import torch
import matplotlib.pyplot as plt
import numpy as np

tensor = torch.empty(100)
torch.nn.init.uniform_(tensor, 1, 10)
# torch.nn.init.uniform_(tensor)

# graph
plt.plot(range(len(tensor)),tensor)
plt.show()

# histogram
plt.hist(np.sort(tensor))
plt.show()
print("uniform")
```

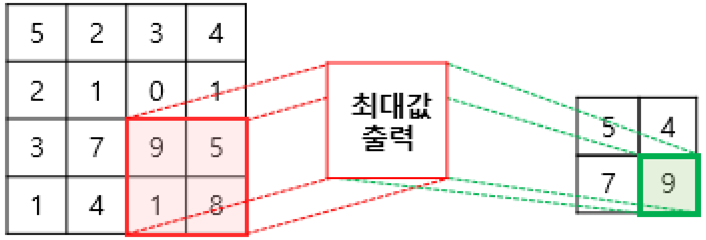
PyTorch – 합성곱 신경망 (CNN, Convolutional Neural Network)

풀링 (Pooling) : 다운샘플링 (Down Sampling)

노이즈를 제거하고, 효율적으로 특징추출

Parameter를 줄여, computation이 줄고, hardware resource를
절약하고, 학습속도를 높인다. 과적합(Overfitting) 억제 효과가 있다.

Max Pooling, Average Pooling, Global Average Pooling



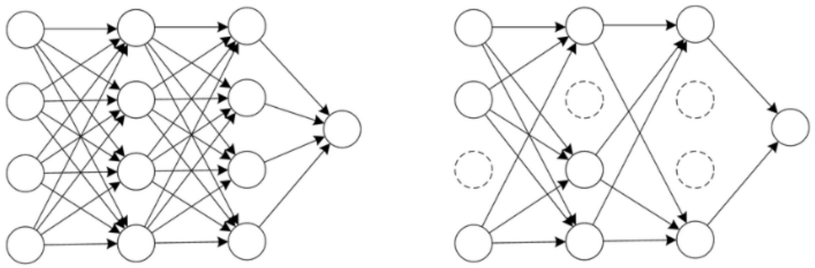
Max Pooling : [5, 4, 7, 9]
Average Pooling : [2.5, 2, 3.75, 5.75]
Global Average Pooling : [3.5]

PyTorch – 합성곱 신경망 (CNN, Convolutional Neural Network)

과(대)적합 (Overfitting) ←→ 과소적합(Underfitting)

1. 데이터 양 늘리기 (데이터 증강, Data Augmentation)
2. 모델의 복잡도 줄이기
3. 가중치 규제(Regularization) 적용하기
4. 드롭아웃(DropOut) 적용하기

Dropout = 0.5, 랜덤으로 절반의 뉴런만 사용하여 학습
(추론시에는 모든 뉴런 사용)



(a) Standard Neural Network　　(b) Network after Dropout

PyTorch – 합성곱 신경망 (CNN, Convolutional Neural Network)

Optimizer

1. SGD (Stochastic Gradient Descent)
2. Adam (Adaptive Moment Estimation)
3. AdaGrad (Adaptive Gradient)
4. AdaDelta (Adaptive Delta
5. Momentum
6. RMSProp
7. NAG (Nesterov Accelerated Gradient)

momentum

$$V_t = m \times V_{t-1} - \eta \nabla_\omega J(\omega_t)$$

$$\omega_{t+1} = \omega_t + V_t$$