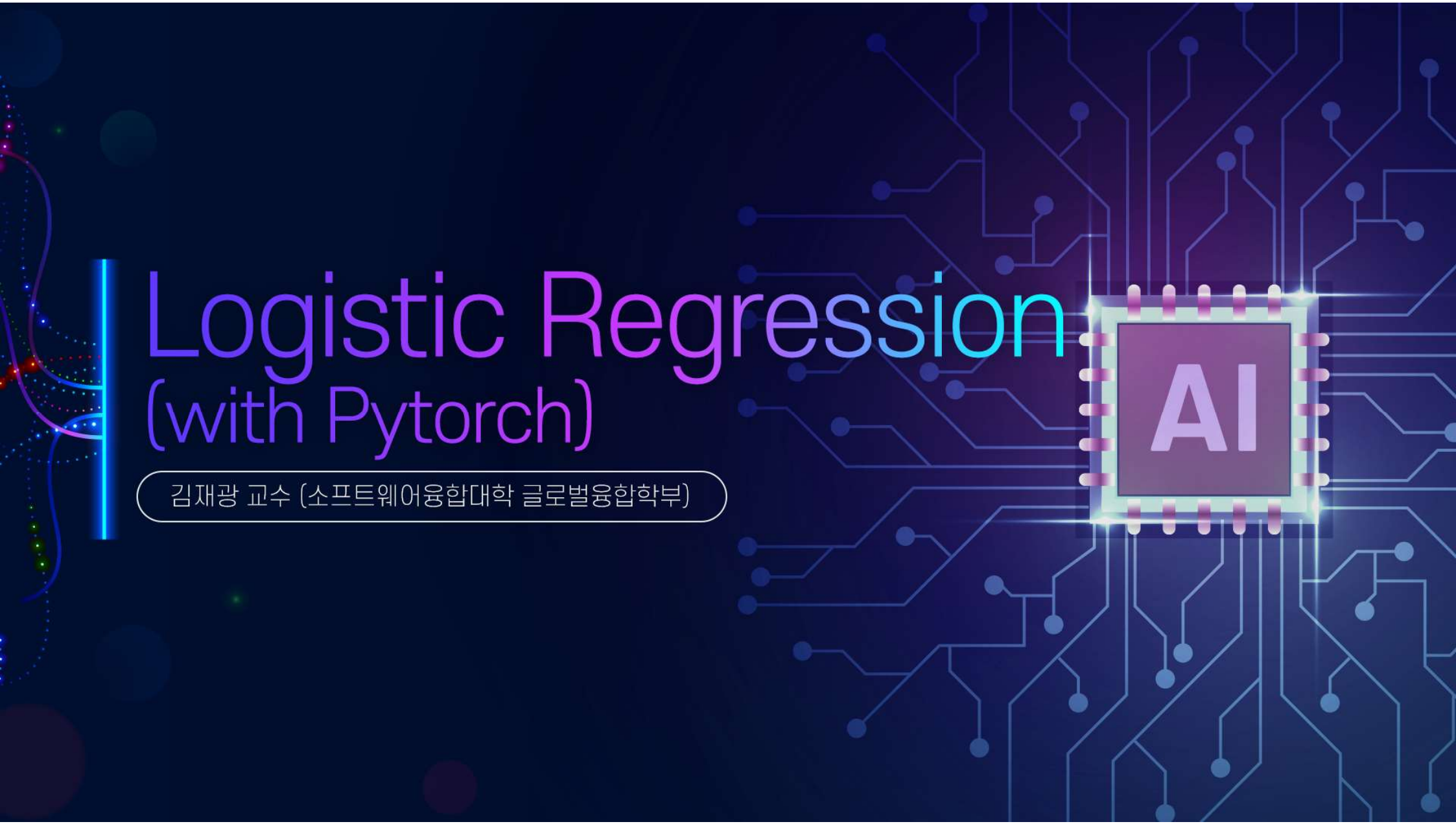
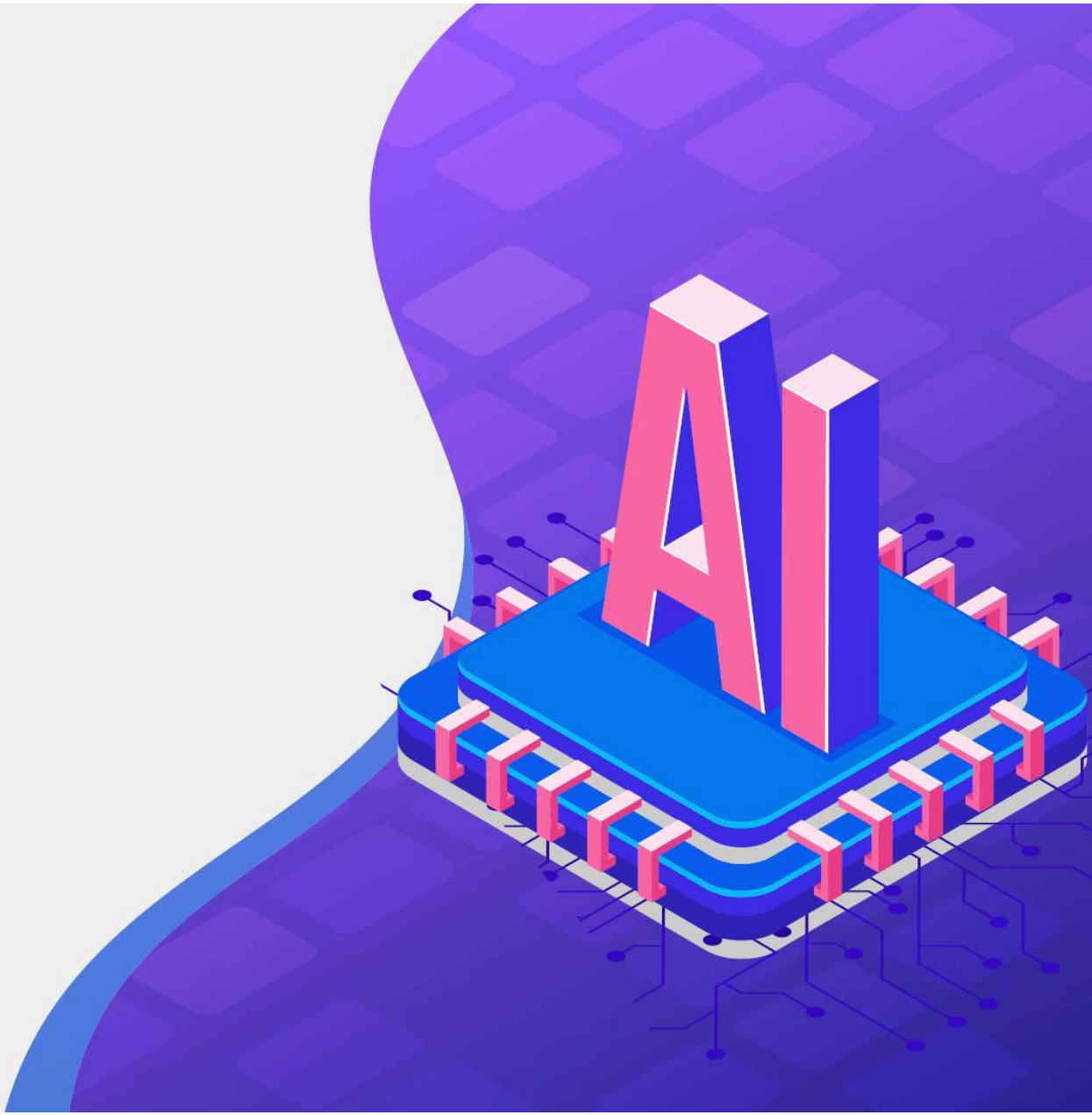

Logistic Regression
(with Pytorch)
김재광 교수 (소프트웨어융합대학 글로벌융합학부)
AI

05 Logistic Regression - with Pytorch

Contents

- Reminder
- Imports
- Training Data
- Computing Hypothesis
- Computing Cost Function
- Evaluation
- Higher Implementation



Reminder

- **Hypothesis**

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

- **Cost**

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

  - If y ≅ H(x), cost is near 0
  - If y ≠ H(x), cost is high

Reminder (Cont'd)

- **Weight update via Gradient Descent**

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

Imports

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

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

Training Data

```
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)
```

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

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

Computing Hypothesis

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

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

```
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=<MulBackward>)
torch.Size([6, 1])
```

Computing Hypothesis (Cont'd)

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

```
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 Cost Function

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

```
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 Cost Function (Cont'd)

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

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

```
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>)
```

Computing Cost Function (Cont'd)

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

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

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

Computing Cost Function (Cont'd)

- with F.binary_cross_entropy

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

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

Computing Cost Function (Cont'd)

```
# 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)
```

Computing Cost Function (Cont'd)

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

    # Cost 계산
    hypothesis = torch.sigmoid(x_train.matmul(W) + b) # or .mm or @
    cost = -(y_train * torch.log(hypothesis) +
              (1 - y_train) * torch.log(1 - hypothesis)).mean()

    # 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()
        ))
```

```
Epoch    0/1000 Cost: 0.693147
Epoch  100/1000 Cost: 0.134722
Epoch  200/1000 Cost: 0.080643
Epoch  300/1000 Cost: 0.057900
Epoch  400/1000 Cost: 0.045300
Epoch  500/1000 Cost: 0.037261
Epoch  600/1000 Cost: 0.031673
Epoch  700/1000 Cost: 0.027556
Epoch  800/1000 Cost: 0.024394
Epoch  900/1000 Cost: 0.021888
Epoch 1000/1000 Cost: 0.019852
```

Computing Cost Function (Cont'd)

```
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()
        ))
```

```
Epoch    0/1000 Cost: 0.693147
Epoch  100/1000 Cost: 0.134722
Epoch  200/1000 Cost: 0.080643
Epoch  300/1000 Cost: 0.057900
Epoch  400/1000 Cost: 0.045300
Epoch  500/1000 Cost: 0.037261
Epoch  600/1000 Cost: 0.031673
Epoch  700/1000 Cost: 0.027556
Epoch  800/1000 Cost: 0.024394
Epoch  900/1000 Cost: 0.021888
Epoch 1000/1000 Cost: 0.019852
```

Evaluation

```
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>)
```

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

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

Evaluation (Cont'd)

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

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

Evaluation (Cont'd)

```
correct_prediction = prediction.float() == y_train
print(correct_prediction[:5])
```

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

```
accuracy = correct_prediction.sum().item() / len(correct_prediction)
print('The accuracy of {:2.2f}% for the training set.'.format(accuracy * 100))
```

```
The accuracy of 76.68% for the training set.
```

Higher Implementation

```
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()

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

nb_epochs = 100
```

Higher Implementation (Cont'd)

```
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()

    # 10번마다 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,
        ))
```

```
Epoch    0/100 Cost: 0.704829 Accuracy 45.72%
Epoch   10/100 Cost: 0.572391 Accuracy 67.59%
Epoch   20/100 Cost: 0.539563 Accuracy 73.25%
Epoch   30/100 Cost: 0.520042 Accuracy 75.89%
Epoch   40/100 Cost: 0.507561 Accuracy 76.15%
Epoch   50/100 Cost: 0.499125 Accuracy 76.42%
Epoch   60/100 Cost: 0.493177 Accuracy 77.21%
Epoch   70/100 Cost: 0.488846 Accuracy 76.81%
Epoch   80/100 Cost: 0.485612 Accuracy 76.28%
Epoch   90/100 Cost: 0.483146 Accuracy 76.55%
Epoch  100/100 Cost: 0.481234 Accuracy 76.81%
```