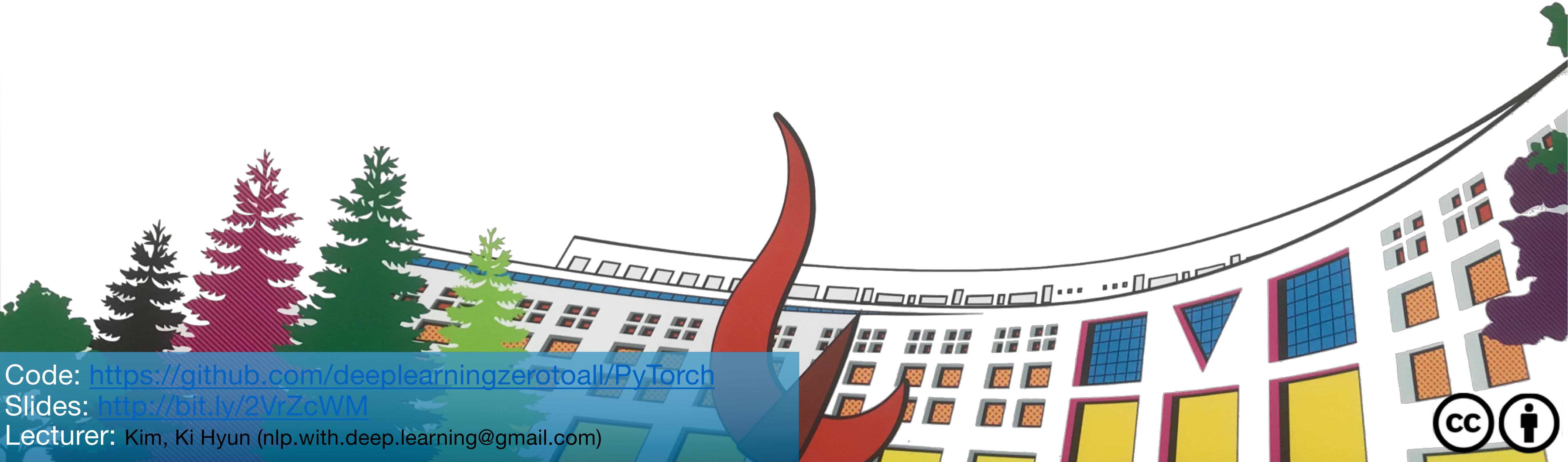
ML/DL for Everyone Season2

with PYTORCH

Logistic Regression



Code: https://github.com/deeplearningzerotoall/PyTorch
Slides: http://bit.ly/2VrZcWM
Lecturer: Kim, Ki Hyun (nlp.with.deep.learning@gmail.com)

Logistic Regression

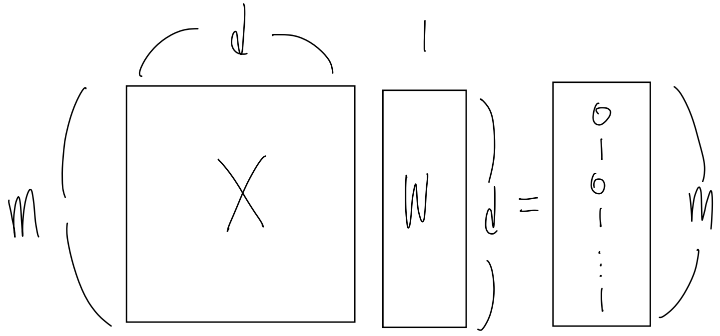
- Reminder
- Computing Hypothesis
- Computing Cost Function
- Evaluation
- Higher Implementation

Reminder: Logistic Regression

Hypothesis

sample의 1일 확률 $= H(X) = \dfrac{1}{1 + e^{-W^T X}} = P(x=1)$

$= \dfrac{}{1 + e^{-XW}}$



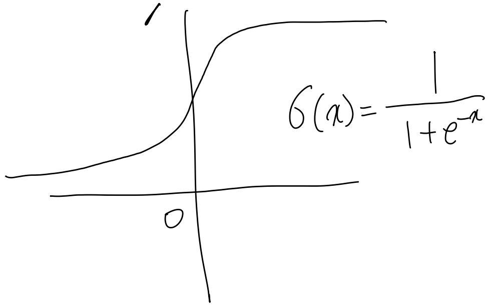
Cost

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

$y \log P(x=1) + (1-y) \log P(x=0)$

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

→ XW → sigmoid (0~1로 변환)



$H(x) = P(x=1 \mid W) = 1 - P(x=0 \mid W)$

→ $m \times 1$ 행렬

Reminder: Logistic Regression

Weight Update via Gradient Descent

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

- α: Learning rate

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)
```

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

Training Data

```python
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]).

```python
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.

```python
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 = 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

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

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

Whole Training Procedure

```
# 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()
        ))
```

```
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: 0.031672
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

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.]])
```

Evaluation

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

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

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

```
# optimizer 설정
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,
        ))
```

```
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%
```