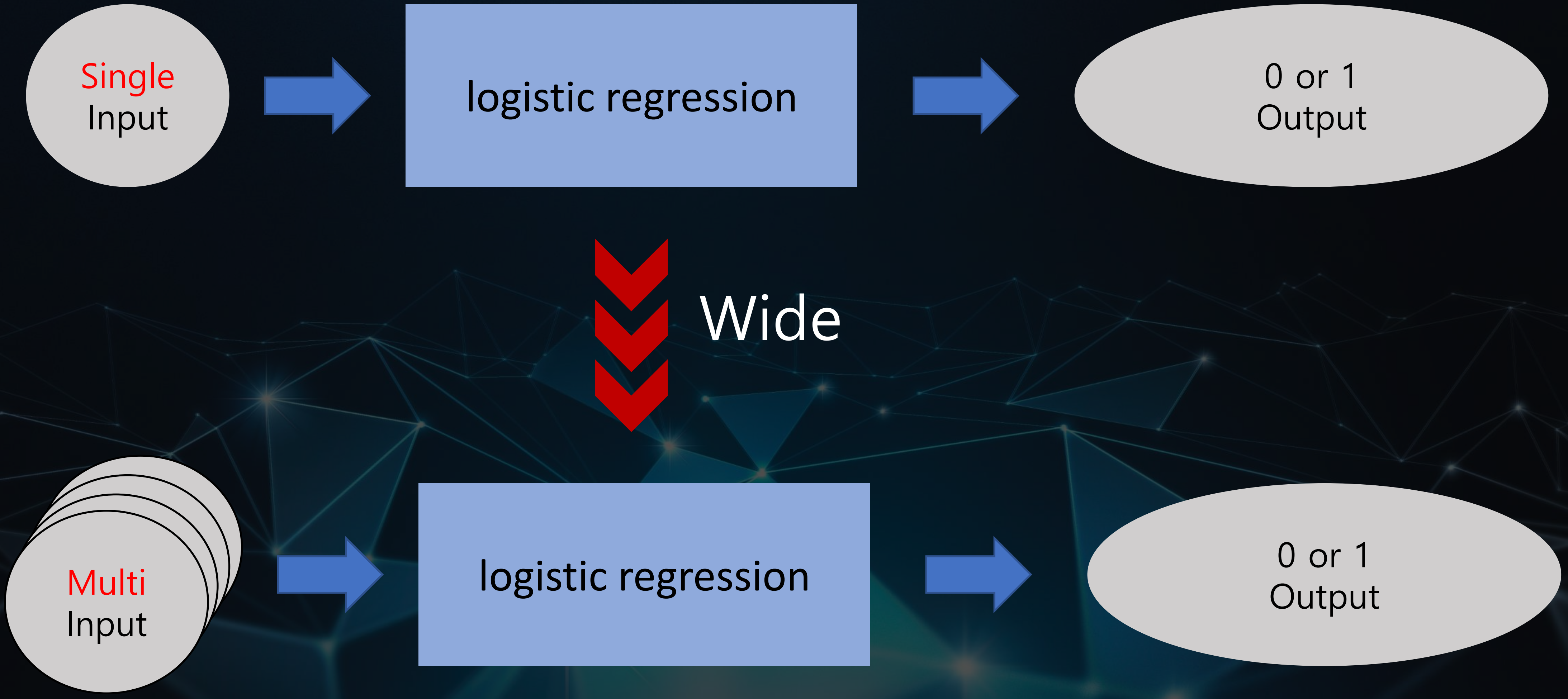
PyTorch Zero To All 7-8

Wide and Deep & PyTorch DataLoader

박종헌

Lecture 07: Wide and Deep

Lecture 07: Wide and Deep



Lecture 07: Wide and Deep

기획
공방

Single Input Logistic Regression

```
[3]
x_data = tensor([[2.1], [4.2], [3.1], [3.3]])
y_data = tensor([[0.], [1.], [0.], [1.]])
```

```
[4]
class Model1(nn.Module):
    def __init__(self):
        super(Model1, self).__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        y_pred = sigmoid(self.linear(x))
        return y_pred

model1 = Model1()
criterion1 = nn.BCELoss(reduction='mean')
optimizer1 = optim.SGD(model1.parameters(), lr=0.1)
```

```
for epoch in range(1000):
    y_pred = model1(x_data)

    loss = criterion1(y_pred, y_data)
    if epoch % 100 == 99:
        print(f'Epoch {epoch + 1}/1000 | Loss: {loss.item():.4f}')

    optimizer1.zero_grad()
    loss.backward()
    optimizer1.step()
```

```
Epoch 100/1000 | Loss: 0.6544
Epoch 200/1000 | Loss: 0.5989
Epoch 300/1000 | Loss: 0.5548
Epoch 400/1000 | Loss: 0.5194
Epoch 500/1000 | Loss: 0.4908
Epoch 600/1000 | Loss: 0.4672
Epoch 700/1000 | Loss: 0.4477
Epoch 800/1000 | Loss: 0.4312
Epoch 900/1000 | Loss: 0.4172
Epoch 1000/1000 | Loss: 0.4051
```

Wide

Multi Input Logistic Regression

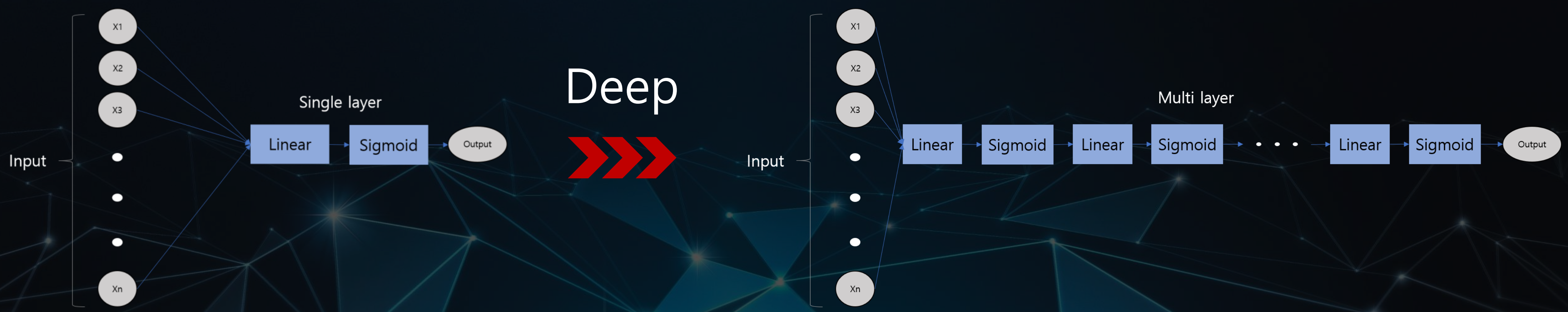
```
[8]
x_data = tensor([[2.1, 0.1], [4.2, 0.8], [3.1, 0.9], [3.3, 0.2]])
y_data = tensor([[0.], [1.], [0.], [1.]])
```

```
[9]
class Model2(nn.Module):
    def __init__(self):
        super(Model2, self).__init__()
        self.linear = nn.Linear(2, 1)

    def forward(self, x):
        y_pred = sigmoid(self.linear(x))
        return y_pred

model2 = Model2()
criterion2 = nn.BCELoss(reduction='mean')
optimizer2 = optim.SGD(model2.parameters(), lr=0.1)
```

```
[10]
for epoch in range(1000):
    y_pred = model2(x_data)

    loss = criterion2(y_pred, y_data)
    if epoch % 100 == 99:
        print(f'Epoch {epoch + 1}/1000 | Loss: {loss.item():.4f}')

    optimizer2.zero_grad()
    loss.backward()
    optimizer2.step()
```

```
Epoch 100/1000 | Loss: 0.5524
Epoch 200/1000 | Loss: 0.4931
Epoch 300/1000 | Loss: 0.4431
Epoch 400/1000 | Loss: 0.4007
Epoch 500/1000 | Loss: 0.3645
Epoch 600/1000 | Loss: 0.3335
Epoch 700/1000 | Loss: 0.3066
Epoch 800/1000 | Loss: 0.2833
Epoch 900/1000 | Loss: 0.2628
Epoch 1000/1000 | Loss: 0.2449
```

Lecture 07: Wide and Deep



기획 공방

Lecture 07: Wide and Deep

Single Layer Logistic Regression

```
[8]
x_data = tensor([[2.1, 0.1], [4.2, 0.8], [3.1, 0.9], [3.3, 0.2]])
y_data = tensor([[0.], [1.], [0.], [1.]])
```

```
[9]
class Model2(nn.Module):
    def __init__(self):
        super(Model2, self).__init__()
        self.linear = nn.Linear(2, 1)

    def forward(self, x):
        y_pred = sigmoid(self.linear(x))
        return y_pred

model2 = Model2()
criterion2 = nn.BCELoss(reduction='mean')
optimizer2 = optim.SGD(model2.parameters(), lr=0.1)
```

```
[10]
for epoch in range(1000):
    y_pred = model2(x_data)

    loss = criterion2(y_pred, y_data)
    if epoch % 100 == 99:
        print(f'Epoch {epoch + 1}/1000 | Loss: {loss.item():.4f}')

    optimizer2.zero_grad()
    loss.backward()
    optimizer2.step()
```

```
Epoch 100/1000 | Loss: 0.5524
Epoch 200/1000 | Loss: 0.4931
Epoch 300/1000 | Loss: 0.4431
Epoch 400/1000 | Loss: 0.4007
Epoch 500/1000 | Loss: 0.3645
Epoch 600/1000 | Loss: 0.3335
Epoch 700/1000 | Loss: 0.3066
Epoch 800/1000 | Loss: 0.2833
Epoch 900/1000 | Loss: 0.2628
Epoch 1000/1000 | Loss: 0.2449
```

Deep

Multi Layer Logistic Regression

```
class Model3(nn.Module):
    def __init__(self):
        super(Model3, self).__init__()
        self.linear1 = nn.Linear(2, 4)
        self.linear2 = nn.Linear(4, 3)
        self.linear3 = nn.Linear(3, 1)

    def forward(self, x):
        x = sigmoid(self.linear1(x))
        x = sigmoid(self.linear2(x))
        y_pred = sigmoid(self.linear3(x))
        return y_pred

model3 = Model3()
criterion3 = nn.BCELoss(reduction='mean')
optimizer3 = optim.SGD(model3.parameters(), lr=0.1)
```

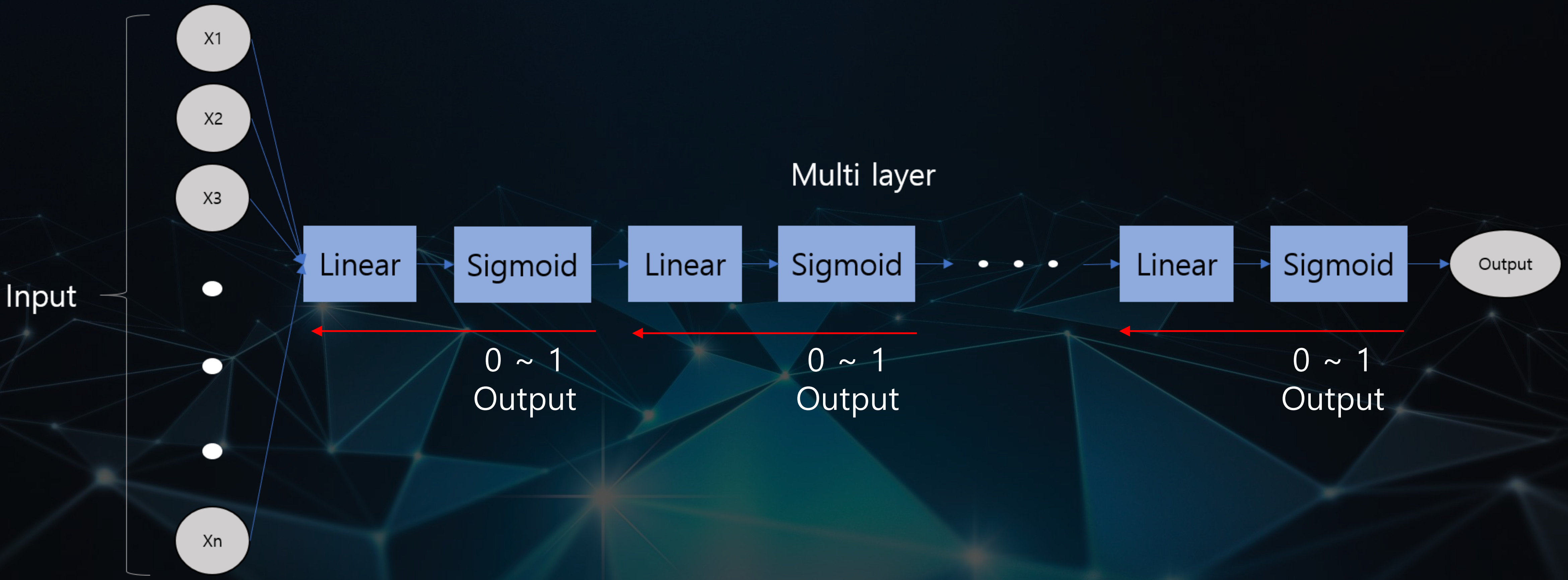
```
[25]
for epoch in range(1000):
    y_pred = model3(x_data)

    loss = criterion3(y_pred, y_data)
    if epoch % 100 == 99:
        print(f'Epoch {epoch + 1}/1000 | Loss: {loss.item():.4f}')

    optimizer3.zero_grad()
    loss.backward()
    optimizer3.step()
```

```
Epoch 100/1000 | Loss: 0.6955
Epoch 200/1000 | Loss: 0.6948
Epoch 300/1000 | Loss: 0.6942
Epoch 400/1000 | Loss: 0.6936
Epoch 500/1000 | Loss: 0.6931
Epoch 600/1000 | Loss: 0.6926
Epoch 700/1000 | Loss: 0.6921
Epoch 800/1000 | Loss: 0.6916
Epoch 900/1000 | Loss: 0.6911
Epoch 1000/1000 | Loss: 0.6905
```

Sigmoid: Vanishing Gradient Problem

Lecture 07: Wide and Deep

Sigmoid: Vanishing Gradient Problem 원인



Lecture 07: Wide and Deep

기획 공방

Sigmoid: Vanishing Gradient Problem 원인

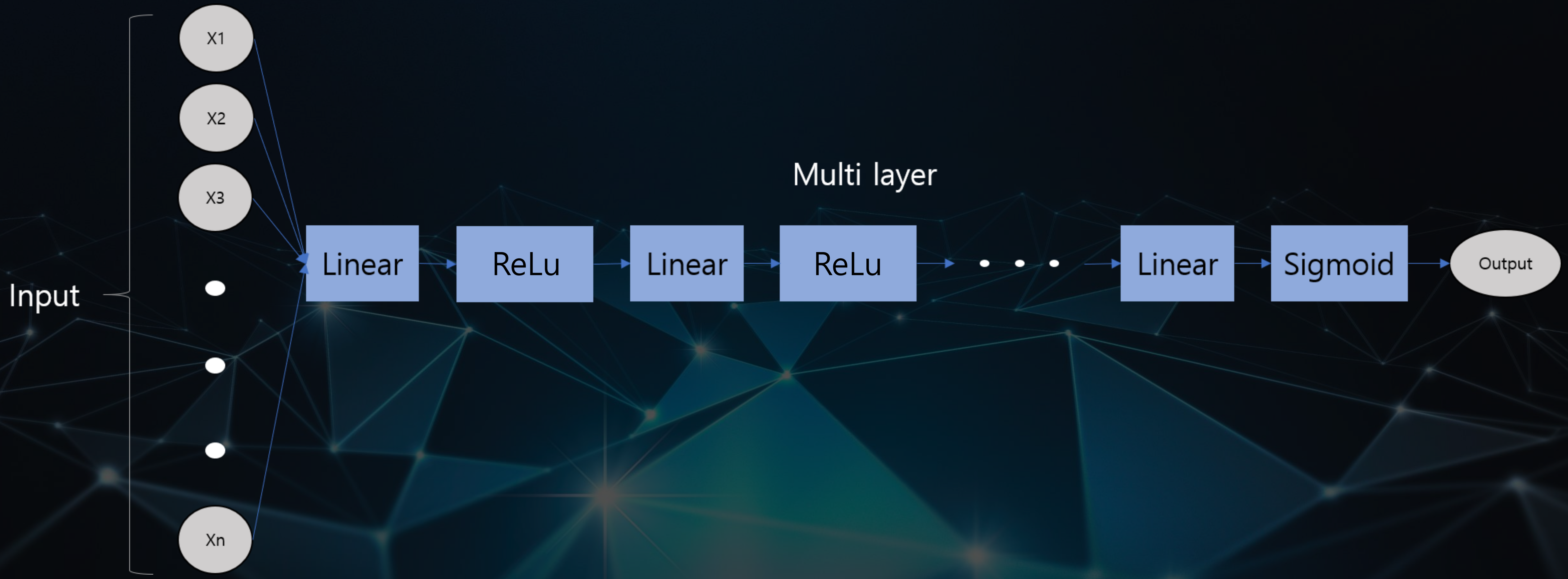
```
Epoch 100/1000 | Loss: 0.6955
        layer 1
        weight : tensor([   [-0.4401, -0.3759],
                            [ 0.6436, -0.3482],
                            [ 0.5769,  0.0342],
                            [ 0.6950,  0.2353]])
        weight gradient: tensor([   [-3.1907e-04,  1.6195e-04],
                                    [-4.5139e-04, -2.1703e-04],
                                    [-1.2960e-04, -2.3442e-04],
                                    [-3.2091e-05, -7.7177e-06]])

        bias: tensor([ 0.1691, -0.0483, -0.3824,  0.6112])
        bias gradient: tensor([ 4.2501e-04, -4.8234e-04, -5.4707e-04, -2.4267e-05])
****************************************************
        layer 2
        weight : tensor([   [-0.1597, -0.0220,  0.2512,  0.0954],
                            [ 0.2478, -0.3817,  0.4408, -0.0572],
                            [-0.1254, -0.3464, -0.4318,  0.4407]])
        weight gradient: tensor([   [-0.0036,  0.0035,  0.0041,  0.0014],
                                    [ 0.0011, -0.0011, -0.0012, -0.0004],
                                    [ 0.0008, -0.0008, -0.0009, -0.0003]])

        bias: tensor([-0.2947, -0.5061,  0.2944])
        bias gradient: tensor([-3.9862e-04,  1.7900e-04,  9.1265e-05])
****************************************************
        layer 3
        weight : tensor([[-0.6831,  0.2270,  0.1522]])
        weight gradient: tensor([[-0.0013,  0.0017,  0.0043]])

        bias: tensor([0.1785])
        bias gradient: tensor([0.0026])
****************************************************
```

Training

```
Epoch 1000/1000 | Loss: 0.6905
        layer 1
        weight : tensor([   [-0.4281, -0.3804],
                            [ 0.6319, -0.3540],
                            [ 0.5743,  0.0271],
                            [ 0.6971,  0.2358]])
        weight gradient: tensor([   [ 6.5484e-05, -4.4609e-05],
                                    [ 6.3232e-04,  3.2091e-04],
                                    [ 1.3046e-04,  3.8950e-04],
                                    [-2.1879e-05, -6.1979e-06]])

        bias: tensor([ 0.1572, -0.0610, -0.3987,  0.6129])
        bias gradient: tensor([-1.1816e-04,  7.0723e-04,  8.9169e-04, -1.8849e-05])
****************************************************
        layer 2
        weight : tensor([   [ 0.1322, -0.3310, -0.1043, -0.0438],
                            [ 0.1521, -0.2751,  0.5626, -0.0064],
                            [-0.1388, -0.3323, -0.4156,  0.4472]])
        weight gradient: tensor([   [-0.0034,  0.0032,  0.0037,  0.0011],
                                    [ 0.0012, -0.0014, -0.0016, -0.0007],
                                    [-0.0004,  0.0005,  0.0005,  0.0002]])

        bias: tensor([-0.2817, -0.5056,  0.2939])
        bias gradient: tensor([-6.0014e-04, -7.2208e-05,  1.8089e-05])
****************************************************
        layer 3
        weight : tensor([[-0.7194,  0.2483, -0.0806]])
        weight gradient: tensor([[ 0.0028, -0.0015,  0.0024]])

        bias: tensor([0.1771])
        bias gradient: tensor([-0.0005])
****************************************************
```

Lecture 07: Wide and Deep

Sigmoid: Vanishing Gradient Problem 해결책



기획 공방

Lecture 07: Wide and Deep

1

```
Epoch 100/1000 | Loss: 0.5415
        layer 1
        weight : tensor([  [-0.1960, -0.3894],
                           [ 0.7286, -0.6200],
                           [ 0.3893, -0.6543],
                           [-0.4494, -0.0221]])
        weight gradient: tensor([  [ 0.0000,  0.0000],
                                   [-0.0088,  0.0267],
                                   [-0.0044,  0.0134],
                                   [ 0.0000,  0.0000]])

        bias: tensor([-0.0897, -0.0414,  0.2696, -0.4879])
        bias gradient: tensor([0.0000, 0.0457, 0.0229, 0.0000])
*****************************************************
        layer 2
        weight : tensor([  [ 0.4601,  0.6845,  0.4134,  0.0360],
                           [ 0.3999,  0.4208, -0.0537,  0.1145],
                           [-0.1630,  0.1551, -0.1244,  0.0670]])
        weight gradient: tensor([  [ 0.0000, -0.0294, -0.0102,  0.0000],
                                   [ 0.0000, -0.0145, -0.0050,  0.0000],
                                   [ 0.0000,  0.0088,  0.0030,  0.0000]])

        bias: tensor([-0.5975, -0.5398,  0.5555])
        bias gradient: tensor([ 0.0541,  0.0266, -0.0161])
*****************************************************
        layer 3
        weight : tensor([[ 0.7916,  0.3900, -0.2363]])
        weight gradient: tensor([[-0.0716, -0.0518,  0.0338]])

        bias: tensor([-0.6099])
        bias gradient: tensor([0.0683])
*****************************************************
```

3

2

Training

```
Epoch 1000/1000 | Loss: 0.0041
        layer 1
        weight : tensor([  [-0.1960, -0.3894],
                           [ 1.2359, -1.3885],
                           [ 0.4704, -0.9732],
                           [-0.4494, -0.0221]])
        weight gradient: tensor([  [ 0.0000,  0.0000],
                                   [ 0.0027,  0.0009],
                                   [-0.0012, -0.0005],
                                   [ 0.0000,  0.0000]])

        bias: tensor([-0.0897, -1.7094, -0.5833, -0.4879])
        bias gradient: tensor([0.0000, 0.0044, 0.0014, 0.0000])
*****************************************************
        layer 2
        weight : tensor([  [ 0.4601,  1.9783,  0.8435,  0.0360],
                           [ 0.3999,  1.2394,  0.4119,  0.1145],
                           [-0.1630,  0.2167, -0.2153,  0.0670]])
        weight gradient: tensor([  [ 0.0000, -0.0019, -0.0004,  0.0000],
                                   [ 0.0000, -0.0011, -0.0011,  0.0000],
                                   [ 0.0000, -0.0009,  0.0003,  0.0000]])

        bias: tensor([-1.7361, -1.0740,  1.6963])
        bias gradient: tensor([ 0.0021,  0.0009, -0.0037])
*****************************************************
        layer 3
        weight : tensor([[ 2.6854,  1.5869, -1.6359]])
        weight gradient: tensor([[-0.0029, -0.0017,  0.0039]])

        bias: tensor([-2.0598])
        bias gradient: tensor([0.0022])
*****************************************************
```

기획
공방

Lecture 07: Wide and Deep

Classifying Diabetes

-0.411765	0.165829	0.213115	0	0	-0.23696	-0.894962	-0.7	1
-0.647059	-0.21608	-0.180328	-0.353535	-0.791962	-0.0760059	-0.854825	-0.833333	0
0.176471	0.155779	0	0	0	0.052161	-0.952178	-0.733333	1
-0.764706	0.979899	0.147541	-0.0909091	0.283688	-0.0909091	-0.931682	0.0666667	0
-0.0588235	0.256281	0.57377	0	0	0	-0.868488	0.1	0
-0.529412	0.105528	0.508197	0	0	0.120715	-0.903501	-0.7	1
0.176471	0.688442	0.213115	0	0	0.132638	-0.608027	-0.566667	0
0.176471	0.396985	0.311475	0	0	-0.19225	0.163962	0.2	1

```
xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)

x_data = Variable(torch.from_numpy(xy[:, 0:-1]))
y_data = Variable(torch.from_numpy(xy[:, [-1]]))

print(x_data.data.shape) # torch.Size([759, 8])
print(y_data.data.shape) # torch.Size([759, 1])
```

```python
from torch import nn, optim, from_numpy
import numpy as np

xy = np.loadtxt('./data/diabetes.csv.gz', delimiter=',', dtype=np.float32)
x_data = from_numpy(xy[:, 0:-1])
y_data = from_numpy(xy[:, [-1]])
print(f'X\'s shape: {x_data.shape} | Y\'s shape: {y_data.shape}')


class Model(nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.l1 = nn.Linear(8, 6)
        self.l2 = nn.Linear(6, 4)
        self.l3 = nn.Linear(4, 1)

        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        out1 = self.sigmoid(self.l1(x))
        out2 = self.sigmoid(self.l2(out1))
        y_pred = self.sigmoid(self.l3(out2))
        return y_pred


# our model
model = Model()

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = nn.BCELoss(reduction='mean')
optimizer = optim.SGD(model.parameters(), lr=0.1)

# Training loop
for epoch in range(100):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(f'Epoch: {epoch + 1}/100 | Loss: {loss.item():.4f}')

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Lecture 08: PyTorch DataLoader

Classifying Diabetes

```
[46]
import numpy as np
from torch import from_numpy, tensor

xy = np.loadtxt("./gdrive/MyDrive/Colab Notebooks/study_pytorch/data/diabetes.csv.gz", delimiter=',', dtype=np.float32)
x_data = from_numpy(xy[:, 0:-1])
y_data = from_numpy(xy[:, [-1]])
print(f'X\'s shape: {x_data.shape} | Y\'s shape: {y_data.shape}')

X's shape: torch.Size([759, 8]) | Y's shape: torch.Size([759, 1])
```

759개의 data

```
# Training loop
for epoch in range(100):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(f'Epoch: {epoch + 1}/100 | Loss: {loss.item():.4f}')

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

759개의 데이터가 전부 모델을 통과한 후
Back-propagation 실행

대용량의 데이터를 처리하기에 비효율적

Lecture 08: PyTorch DataLoader

- 1 epoch : 모든 dataset에 대해 forward pass와 backward pass를 한번씩 진행한 경우
- batch size : forward pass와 backward pass를 1번 진행할 때의 dataset 수
  -> batch size가 클수록 더 많은 메모리 공간 필요!!
- batch : batch size만큼의 dataset
- 1 pass : 1번의 forward pass와 backward pass
- iterations : 1 epoch 동안 일어나는 pass의 수
  -> 1 epoch 동안 batch size만큼의 dataset이 몇 번 통과하는지

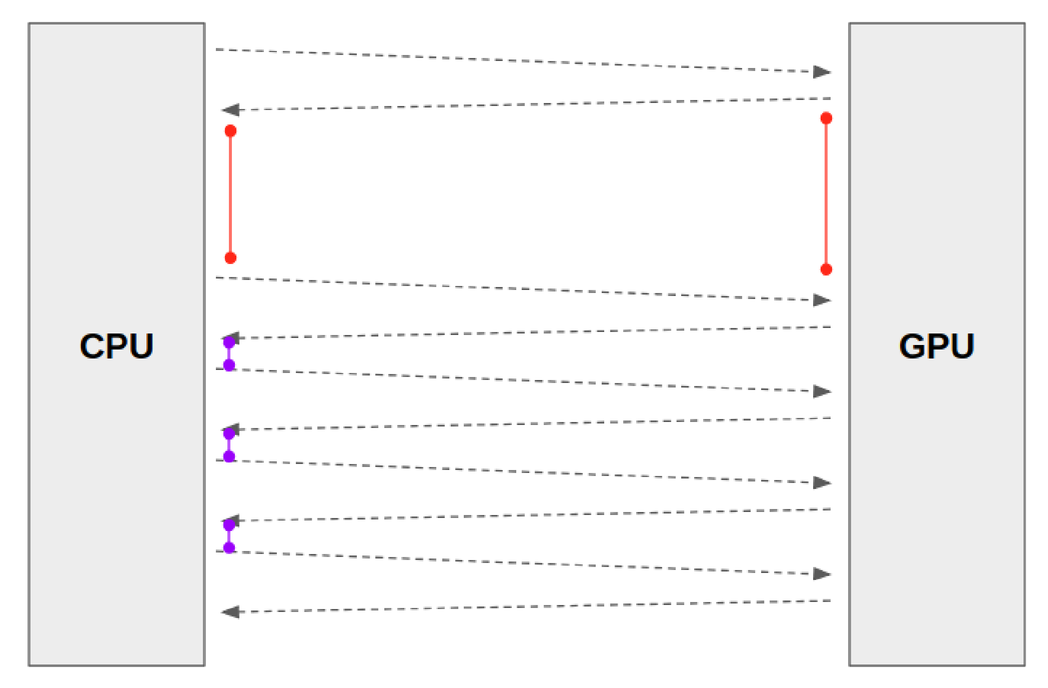
전체 데이터 수가 1000개이고 배치 크기(batch size)가 500개이면
1 epoch 동안 2번의 iteration 반복

Lecture 08: PyTorch DataLoader

```python
from torch.utils.data import Dataset, DataLoader
from torch import from_numpy, tensor
import numpy as np

class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):
        xy = np.loadtxt('./data/diabetes.csv.gz',
                        delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = from_numpy(xy[:, 0:-1])
        self.y_data = from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

shuffle : dataset의 순서가 model의 학습을 방해하지 않도록 해준다

num_workers : 사용할 CPU 코어 개수



Lecture 08: PyTorch DataLoader

Exercise 8-1: CIFAR10

```
trainset
Dataset CIFAR10
    Number of datapoints: 50000
```

```
[1,     1] loss: 2.304
[2,     1] loss: 2.304
Finished Training
1 epoch 동안 걸린 시간 11.298972606658936
```

```
trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=5,
                                          shuffle=True,
                                          num_workers=2)
```

```
[1,  2000] loss: 2.175
[1,  4000] loss: 1.796
[1,  6000] loss: 1.620
[1,  8000] loss: 1.515
[1, 10000] loss: 1.454
[2,  2000] loss: 1.399
[2,  4000] loss: 1.360
[2,  6000] loss: 1.325
[2,  8000] loss: 1.310
[2, 10000] loss: 1.271
Finished Training
1 epoch 동안 걸린 시간 48.29654943943024
```

기획
공방

D-ai-ving

Thank You!!