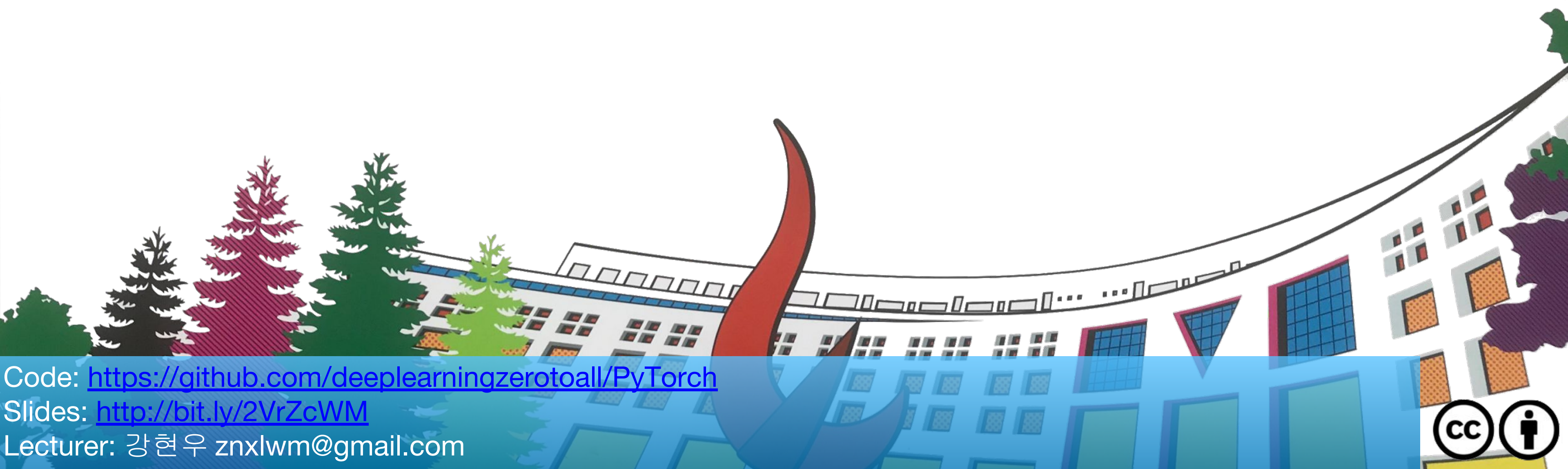
ML/DL for Everyone Season2

Multi Layer Perceptron

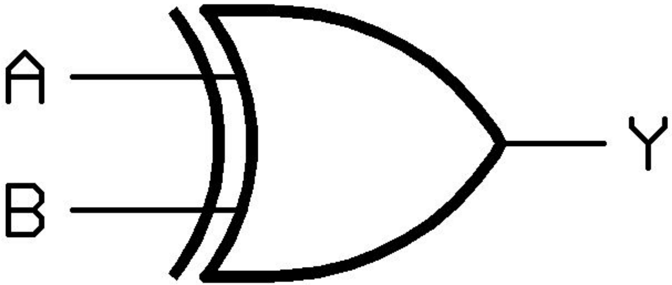


Code: https://github.com/deeplearningzerotoall/PyTorch
Slides: http://bit.ly/2VrZcWM
Lecturer: 강현우 znxlwm@gmail.com

Multi Layer Perceptron

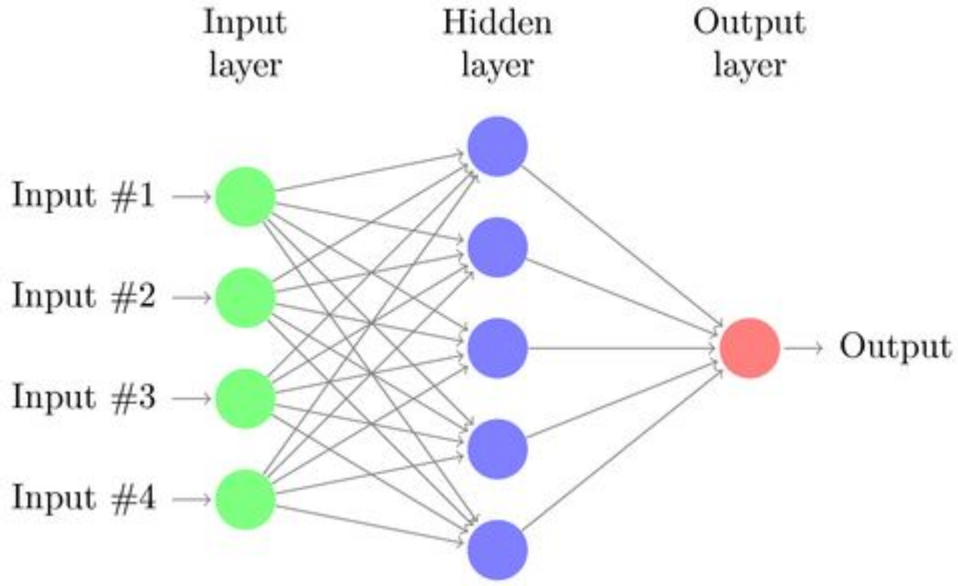
- Review: XOR
- Multi Layer Perceptron
- Backpropagation
- Code: xor-nn
- Code: xor-nn-wide-deep

Review: XOR



https://commons.wikimedia.org/wiki/File:Logic-gate-xor-us.png

Multi Layer Perceptron



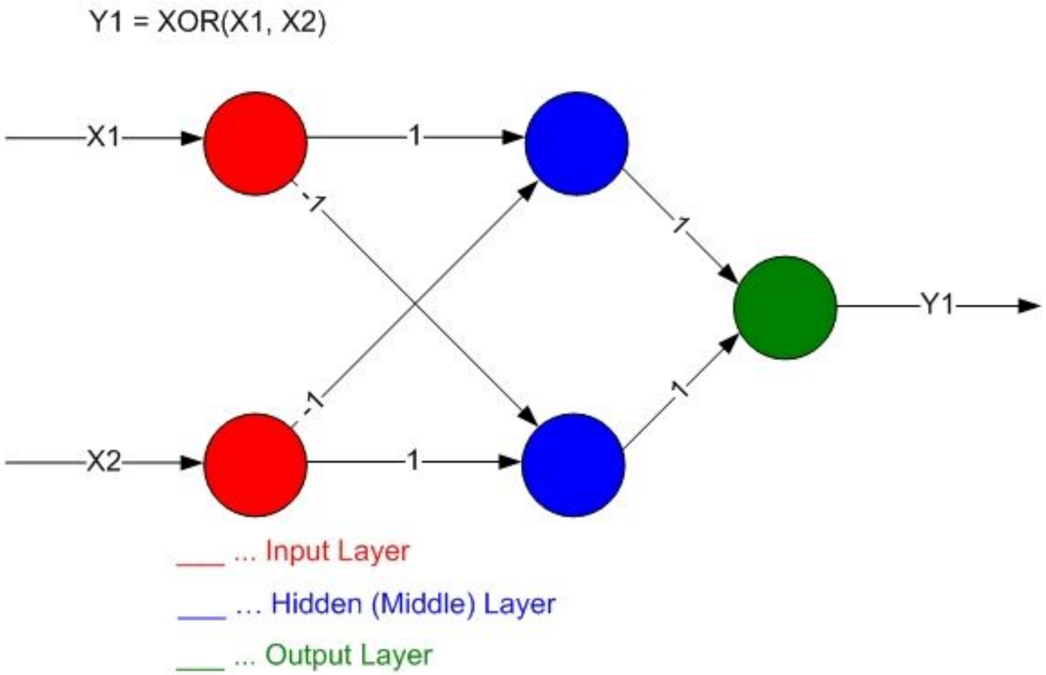
- We need to use MLP, multilayer perceptrons (multilayer neural nets)

- No one on earth had found a viable way to train MLPs good enough to learn such simple functions

(by Marvin Minsky, founder of the MIT AI Lab, 1969)

http://www.texample.net/tikz/examples/neural-network/

Backpropagation



https://commons.wikimedia.org/wiki/File:Perceptron_XOR.jpg

Backpropagation

https://www.youtube.com/watch?v=573EZkzfnZ0&list=PLlMkM4tgfjnLSOjrEJN31gZATbcj_MpUm&index=27

Backpropagation

```
X = torch.FloatTensor([[0, 0], [0, 1], [1, 0], [1, 1]]).to(device)
Y = torch.FloatTensor([[0], [1], [1], [0]]).to(device)

# nn layers
w1 = torch.Tensor(2, 2).to(device)
b1 = torch.Tensor(2).to(device)
w2 = torch.Tensor(2, 1).to(device)
b2 = torch.Tensor(1).to(device)

def sigmoid(x):
   # sigmoid function
   return 1.0 / (1.0 + torch.exp(-x))
   # return torch.div(torch.tensor(1), torch.add(torch.tensor(1.0), torch.exp(-x)))

def sigmoid_prime(x):
   # derivative of the sigmoid function
   return sigmoid(x) * (1 - sigmoid(x))
```

Backpropagation

```python
for step in range(10001):
    # forward
    l1 = torch.add(torch.matmul(X, w1), b1)
    a1 = sigmoid(l1)
    l2 = torch.add(torch.matmul(a1, w2), b2)
    Y_pred = sigmoid(l2)

    cost = -torch.mean(Y * torch.log(Y_pred) + (1 - Y) * torch.log(1 - Y_pred))

    # Back prop (chain rule)
    ...
```

Backpropagation

```
for step in range(10001):
    # forward
    ...

    # Back prop (chain rule)
    # Loss derivative
    d_Y_pred = (Y_pred - Y) / (Y_pred * (1.0 - Y_pred) + 1e-7)

    # Layer 2
    d_l2 = d_Y_pred * sigmoid_prime(l2)
    d_b2 = d_l2
    d_w2 = torch.matmul(torch.transpose(a1, 0, 1), d_b2)

    # Layer 1
    d_a1 = torch.matmul(d_b2, torch.transpose(w2, 0, 1))
    d_l1 = d_a1 * sigmoid_prime(l1)
    d_b1 = d_l1
    d_w1 = torch.matmul(torch.transpose(X, 0, 1), d_b1)
```

Backpropagation

```
for step in range(10001):
    # forward
    ...

    # Back prop (chain rule)
    ...

    # Weight update
    w1 = w1 - learning_rate * d_w1
    b1 = b1 - learning_rate * torch.mean(d_b1, 0)
    w2 = w2 - learning_rate * d_w2
    b2 = b2 - learning_rate * torch.mean(d_b2, 0)

    if step % 100 == 0:
        print(step, cost.item())
```

```
0 1.021416425704956
100 0.5204591155052185
200 0.2311055213212967
300 0.07911999523639679
...
9800 0.0011527219321657874
9900 0.0011408460559323430
10000 0.0011291791452094913
```

```
Hypothesis:  [[9.0581283e-04]
 [9.9857259e-01]
 [9.9856734e-01]
 [7.4765802e-04]]
Correct:  [[0.]
 [1.]
 [1.]
 [0.]]
Accuracy:  1.0
```

Code: xor-nn

```
X = torch.FloatTensor([[0, 0], [0, 1], [1, 0], [1, 1]]).to(device)
Y = torch.FloatTensor([[0], [1], [1], [0]]).to(device)
# nn layers
linear1 = torch.nn.Linear(2, 2, bias=True)
linear2 = torch.nn.Linear(2, 1, bias=True)
sigmoid = torch.nn.Sigmoid()
model = torch.nn.Sequential(linear1, sigmoid, linear2, sigmoid).to(device)
# define cost/loss & optimizer
criterion = torch.nn.BCELoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=1)
for step in range(10001):
   optimizer.zero_grad()
   hypothesis = model(X)
   # cost/loss function
   cost = criterion(hypothesis, Y)
   cost.backward()
   optimizer.step()
   if step % 100 == 0:
       print(step, cost.item())
```

```
0 0.7434073090553284
100 0.6931650638580322
200 0.6931577920913696
300 0.6931517124176025
...
9800 0.0012681199004873633
9900 0.0012511102249845862
10000 0.0012345188297331333
```

```
Hypothesis:  [[0.00106378]
 [0.9988938 ]
 [0.9988939 ]
 [0.00165883]]
Correct:  [[0.]
 [1.]
 [1.]
 [0.]]
Accuracy:  1.0
```

Code: xor-nn-wide-deep

```
X = torch.FloatTensor([[0, 0], [0, 1], [1, 0], [1, 1]]).to(device)
Y = torch.FloatTensor([[0], [1], [1], [0]]).to(device)

# nn layers
linear1 = torch.nn.Linear(2, 10, bias=True)
linear2 = torch.nn.Linear(10, 10, bias=True)
linear3 = torch.nn.Linear(10, 10, bias=True)
linear4 = torch.nn.Linear(10, 1, bias=True)
sigmoid = torch.nn.Sigmoid()


...
```

```
0 0.6948983669281006
100 0.6931558847427368
200 0.6931535005569458
300 0.6931513547897339
...
9800 0.00016415018762927502
9900 0.00016021561168599874
10000 0.00015650462592020631
```

```
Hypothesis:  [[1.1170952e-04]
 [9.9982882e-01]
 [9.9984229e-01]
 [1.8537771e-04]]
Correct:  [[0.]
 [1.]
 [1.]
 [0.]]
Accuracy:  1.0
```

What’s Next?

- ReLU