NeuralNet 101

6. Error back propagation

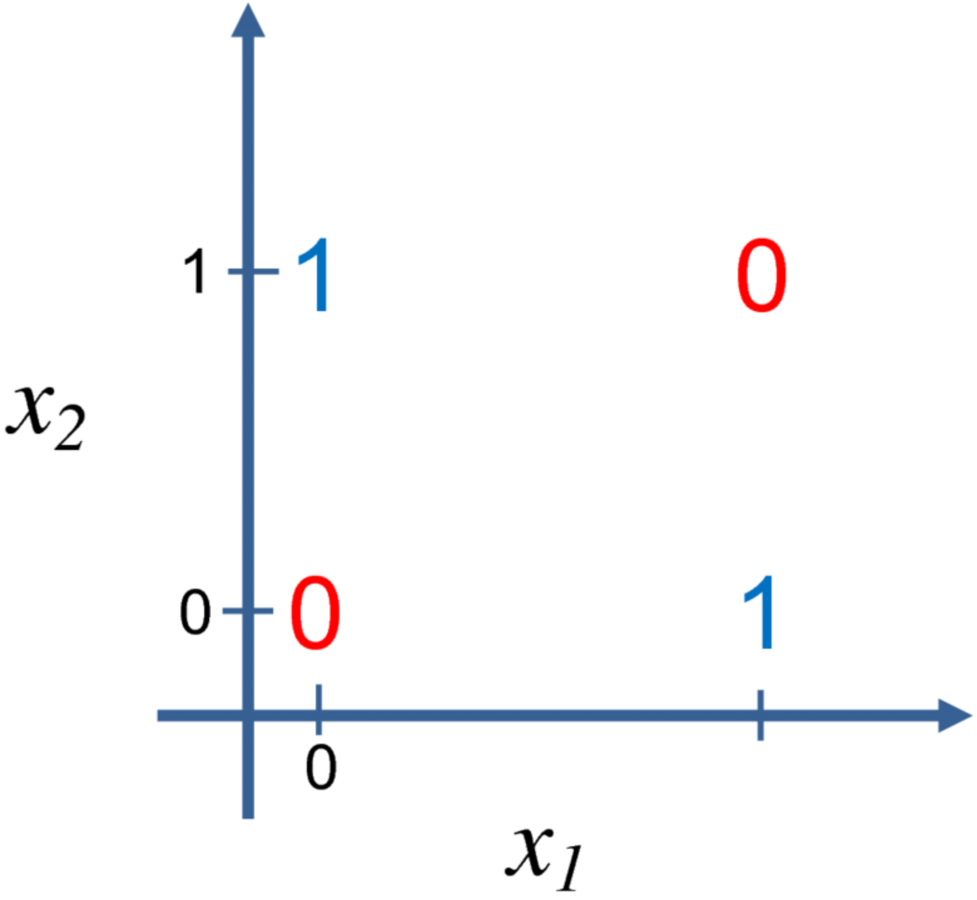
Based on Softmax, we can solve many things...

But we cannot solve the simple problem...

'XOR' gate

- When 0, 0 input -> returns 0
- When 0, 1 input -> returns 1
- When 1, 0 input -> returns 1
- When 1, 1 input -> returns 0

'XOR' gate



And How can we solve this problem?

Add more complexity!

Add complexity on neural network

- Let $F(x) = WX + b$
- Let $s(x) = \text{sigmoid}(x)$

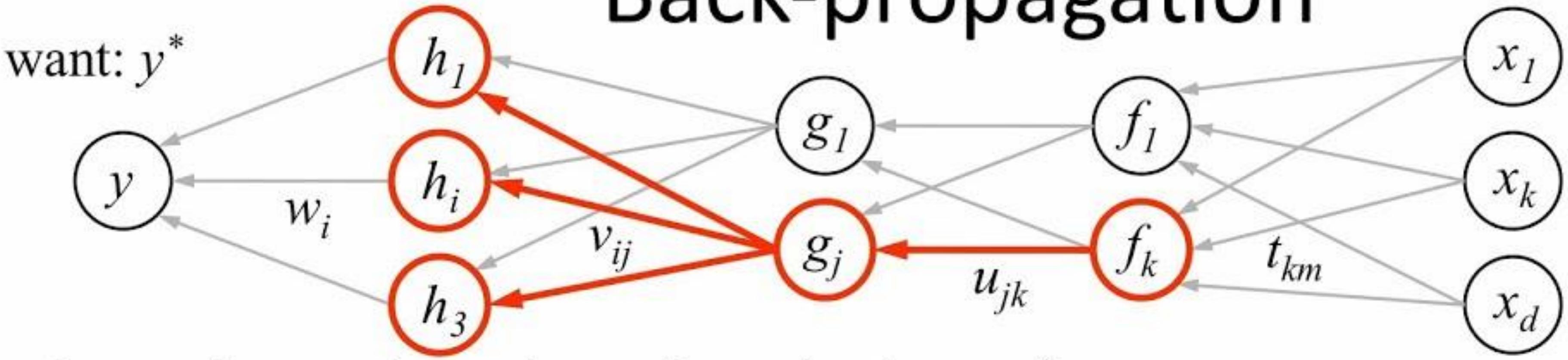
- Then, we can set new complex function as $G(x) = s(F(s(F(s(F(s(F(x))))))))$

- Then we might be able to solve the problem, but we cannot define the Loss function of it.

So, how to get loss function for each layer?

That is error backpropagation

Back-propagation



1. receive new observation $\boldsymbol{x} = [x_1 \ldots x_d]$ and target y^*
2. **feed forward:** for each unit g_j in each layer 1…L
   compute g_j based on units f_k from previous layer: $g_j = \sigma\left(u_{j0} + \sum_k u_{jk} f_k\right)$
3. get prediction y and error $(y\text{-}y^*)$
4. **back-propagate error:** for each unit g_j in each layer L…1

(a) compute error on g_j

$$\underbrace{\frac{\partial E}{\partial g_j}}_{\text{should } g_j \text{ be higher or lower?}} = \sum_i \underbrace{\sigma'(h_i) v_{ij}}_{\text{how } h_i \text{ will change as } g_j \text{ changes}} \underbrace{\frac{\partial E}{\partial h_i}}_{\text{was } h_i \text{ too high or too low?}}$$

(b) for each u_{jk} that affects g_j

(i) compute error on u_{jk}

$$\frac{\partial E}{\partial u_{jk}} = \underbrace{\frac{\partial E}{\partial g_j}}_{\text{do we want } g_j \text{ to be higher/lower}} \underbrace{\sigma'(g_j) f_k}_{\text{how } g_j \text{ will change if } u_{jk} \text{ is higher/lower}}$$

(ii) update the weight

$$u_{jk} \leftarrow u_{jk} - \eta \frac{\partial E}{\partial u_{jk}}$$

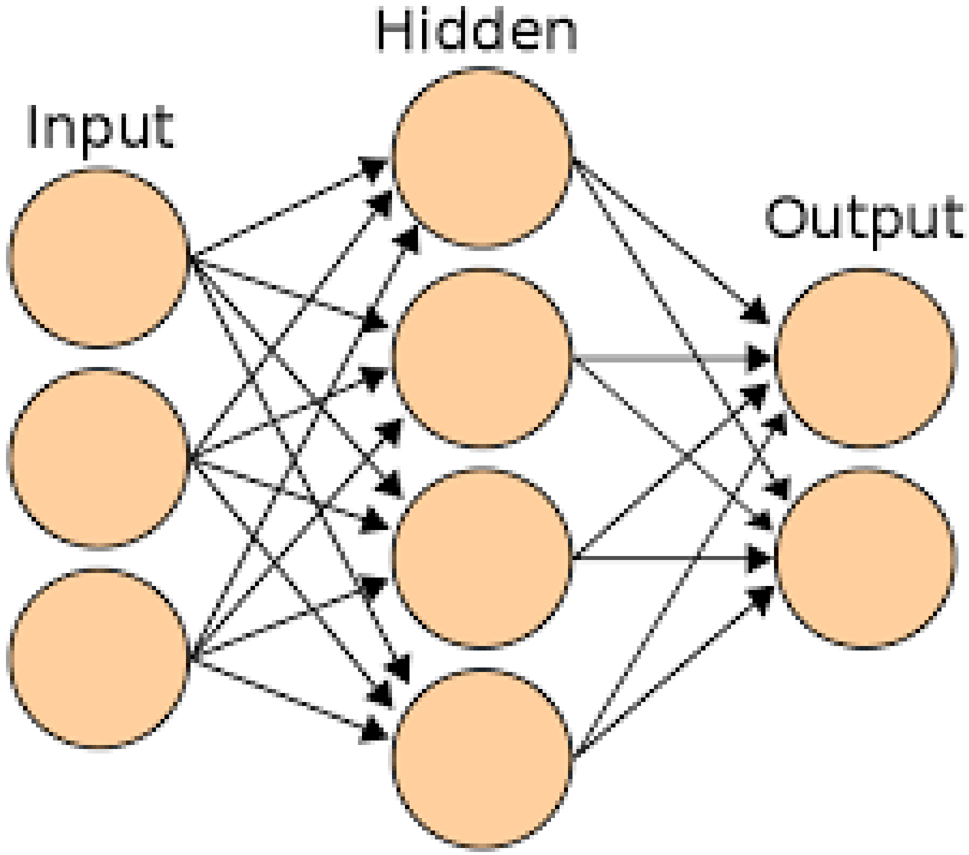
Lab session

In lab session…

- ANN (Artifical Neural Network)
- Making Affine, Sigmoid, Softmax layer with back propagation
- Making Model

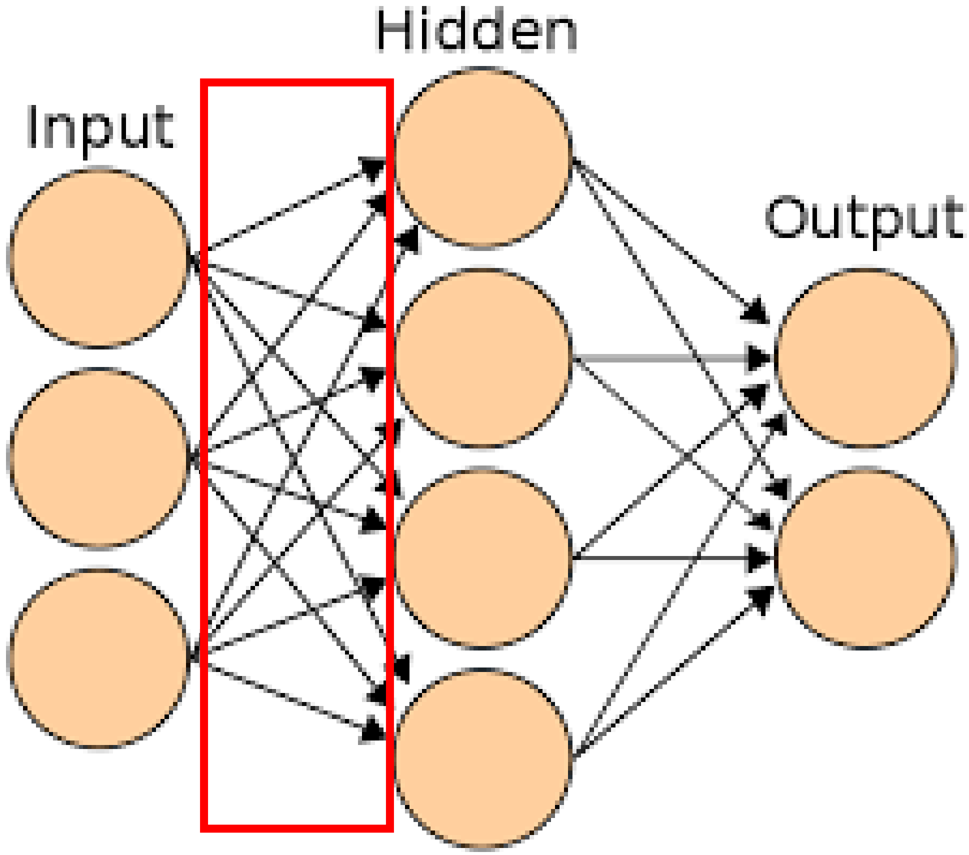
Add complexity on neural network -ANN (Artifical Neural Network)

- Adding layer
- Use non-linear function for non-linear classification



Making Affine layer

$$Y = X \cdot W + B$$



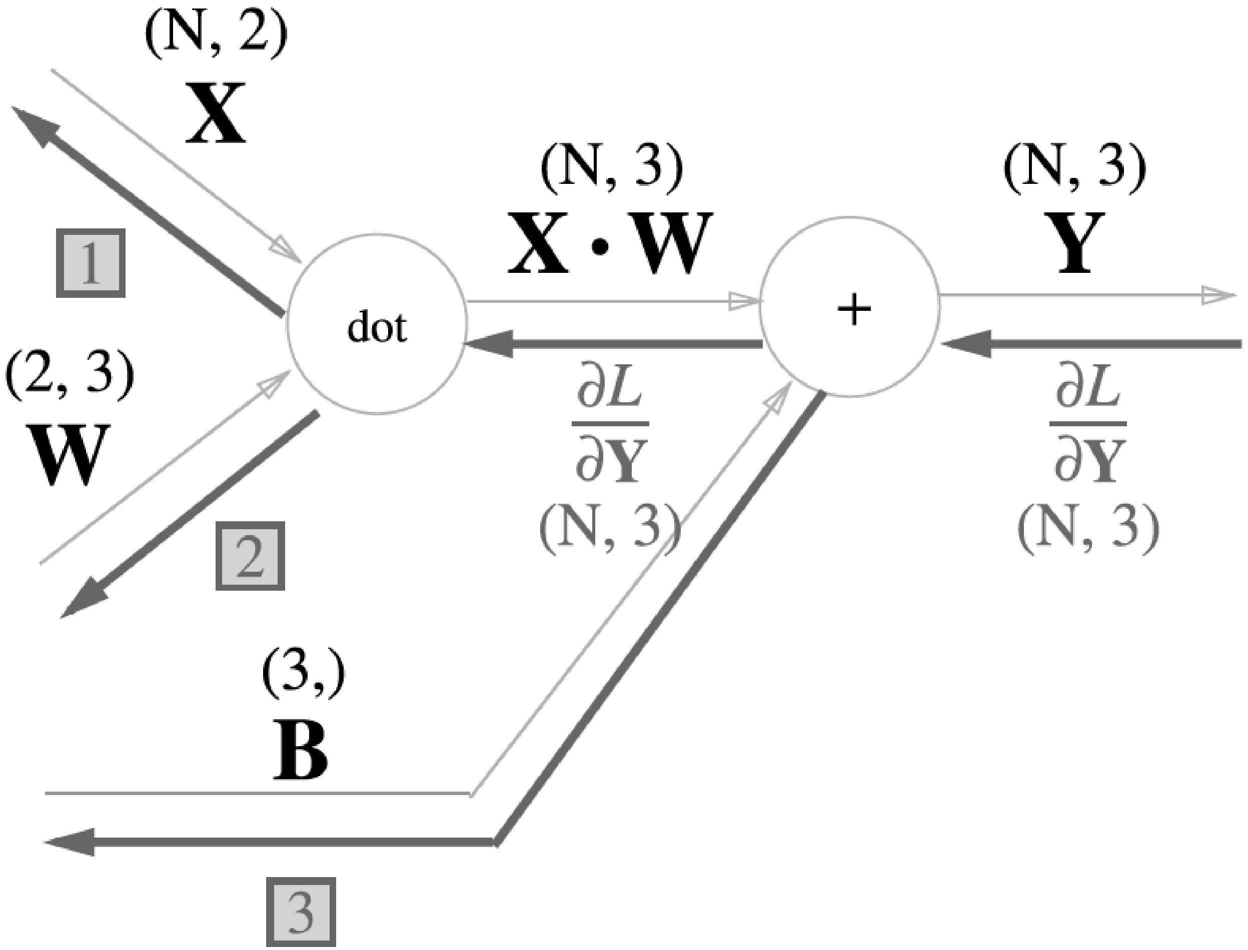
Making Affine layer

$$Y = X \cdot W + B$$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial X} = \frac{\partial L}{\partial Y} \cdot W^T$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial W} = X^T \cdot \frac{\partial L}{\partial Y}$$

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial B} = sum\left(\frac{\partial L}{\partial Y}\right) \text{ for axis 0}$$



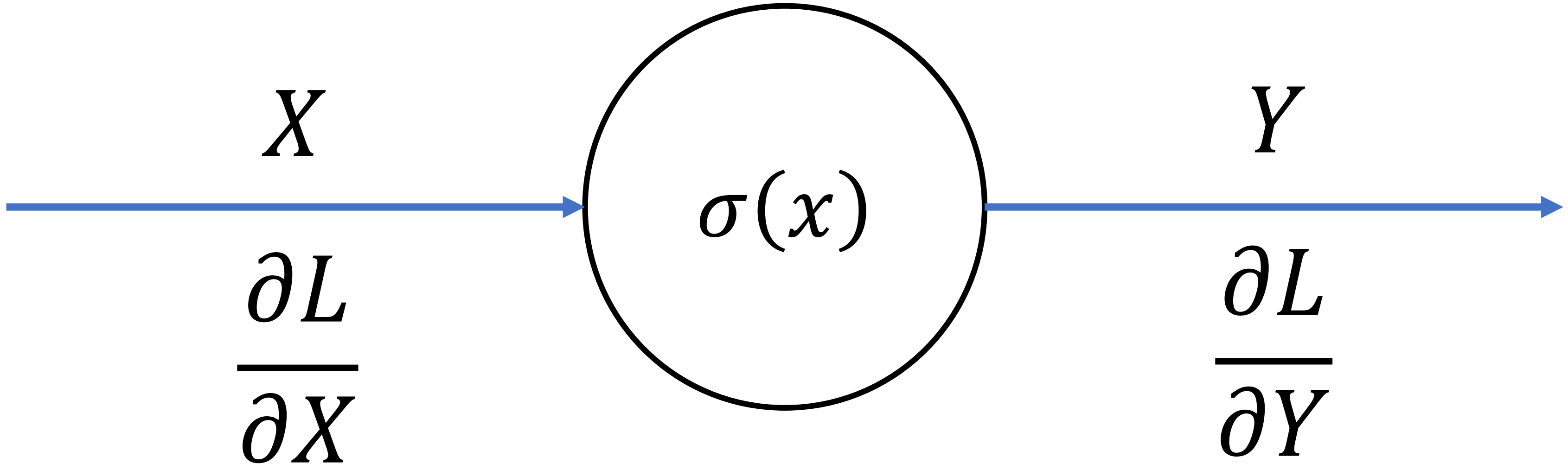
Making Affine layer

```
class Affine:
    def __init__(self, input_size, output_size):
        self.W = 1.0 * np.random.randn(input_size, output_size)
        self.b = 1.0 * np.zeros(output_size)
        self.x = None
        self.y = None
        self.dW = None # W gradient
        self.db = None # b gradient

    def forward(self, x):
        pass

    def backward(self, d_out, learning_rate):
        pass
```

Making Sigmoid Layer



$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \frac{\partial \sigma(x)}{\partial x} = (1 - \sigma(x))\sigma(x)$$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial X} = \frac{\partial L}{\partial Y}(1 - Y)Y$$

Making Sigmoid Layer

```python
class Sigmoid:
    def __init__(self):
        self.y = None

    def forward(self, x):
        pass

    def backward(self, d_out, learning_rate=None):
        pass
```

Making Softmax Layer

```python
class Softmax:
    def __init__(self):
        self.error = None
        self.y = None
        self.t = None

    def forward(self, x):
        pass

    def loss(self, t):
        pass #hint : cross_entropy_error

    def backward(self, d_out=1, learning_rate=None):
        pass
```

Backward hint :
Dx = mean of (hypothesis - answer)

Making Model

```python
class Model:
    def __init__(self):
        self.layer = []
        self.error = None

    def add(self, layer):
        self.layer.append(layer)

    def forward(self, x):
        pass #return results after calling all of layers

    def backward(self):
        pass #call all backward (in REVERSED order)
```

Problem 1

- Make XOR Model and train it using back propagation
- (Do NOT use PyTorch)

A	B	Value
0	0	0
0	1	1
1	0	1
1	1	0

Problem 2

- Make MNIST Model and train it using back propagation
- (Do NOT use PyTorch / only torchvision and numpy)
- Hint : one-hot vector / see Lab05 Iris skeleton code

```
import torchvision.datasets as dsets
import torchvision.transforms as transforms
```

```
encoding = np.eye(10)
mnist_train = dsets.MNIST(root="MNIST_data/", train=True, transform=transforms.ToTensor(), download=True)
x_train = np.array(mnist_train.data.view(-1, 28 * 28).float())/255
y_train = np.array([encoding[i] for i in mnist_train.targets])
```

References.

- https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=apr407&logNo=221237867979
- https://silver-g-0114.tistory.com/73
- https://velog.io/@kylebae1017/Backpropagation-in-Affine-Layer