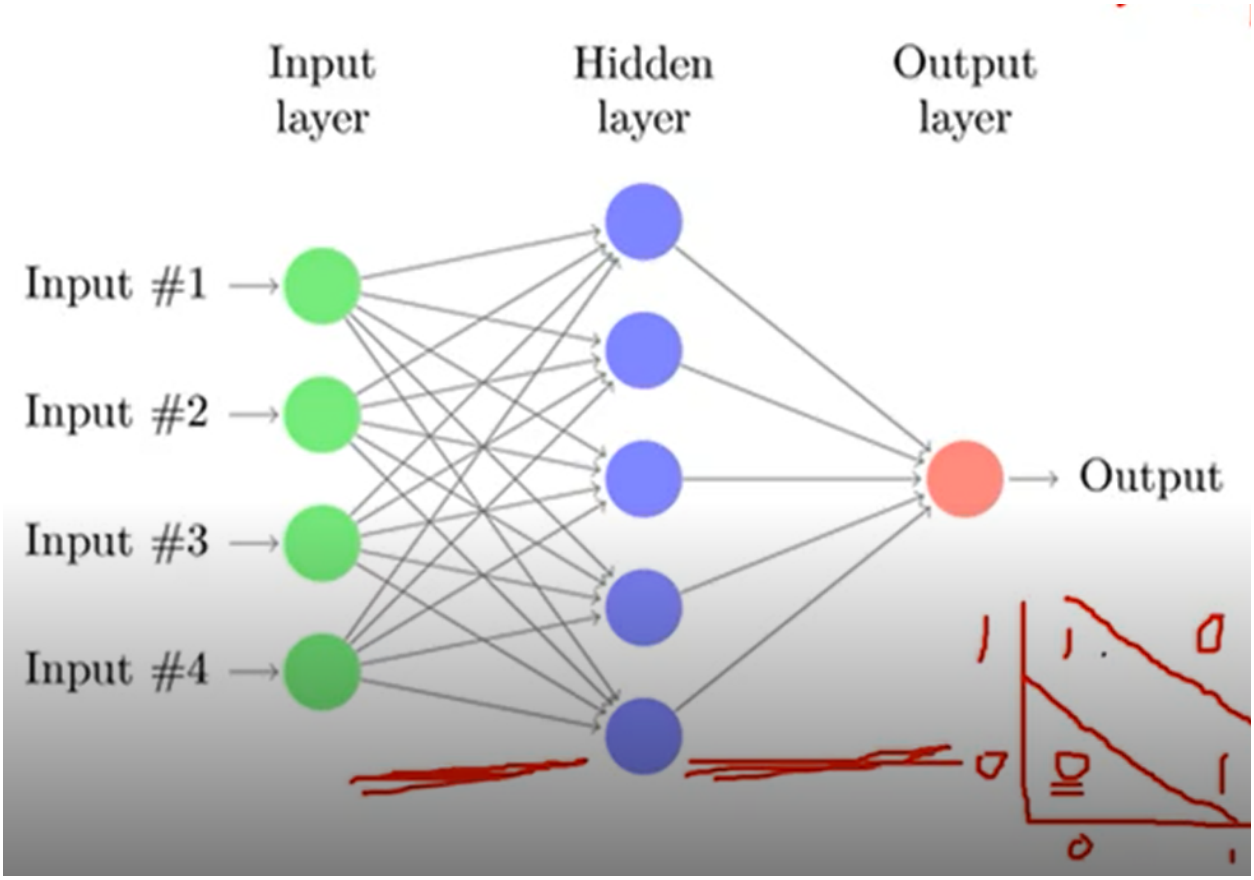
Perceptron

상태	DNN
담당자	

Introduction

> XOR 문제는 단층 퍼셉트론으로 해결할 수 없음 → multi-layer perceptron의 도입

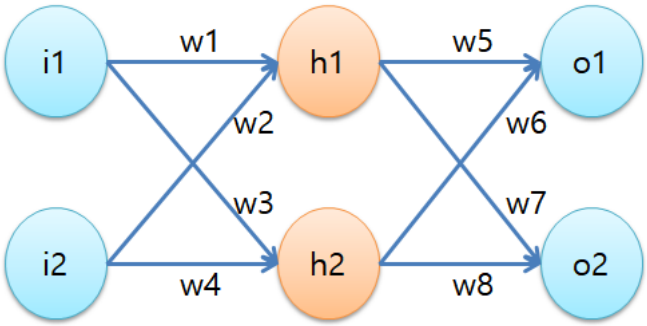
Multi-layer perceptron이 기존 단층 퍼셉트론과 다른 점은 무엇인가? 하나의 선을 더 그려서 기존에 해결하지 못했던 XOR 문제를 해결할 수 있다.



여기서의 문제점 : 그러면 multi-layer perceptron 구조를 어떻게 학습시킬 수 있는가?

Backpropagation

> 뒷단에서부터 Loss를 minimize할 수 있도록 weight을 업데이트하는 방법



i_1, i_2를 input layer, h_1, h_2를 hidden layer, o_1, o_2를 output layer라 하자. 그리고 다음과 같이 정의하자.

- net : input에 대한 weighted sum
- out : net을 activation function으로 활성화한 것

오차 계산 단계

예를 들어 $net_{h_1} = i_1 w_1 + i_2 w_2$, $out_{h_1} = f(net_{h_1}) = \frac{1}{1+e^{-net_{h_1}}}$ (이때의 activation function은 sigmoid라 하자)이다. net_{h_2}, out_{h_2}도 같은 방식으로 계산할 수 있다.

이 out_{h_1}, out_{h_2}는 다시 다음 층의 input이 된다는 점이 특징인데, 예를 들어
$net_{o_1} = out_{h_1} w_5 + out_{h_2} w_6$, $out_{o_1} = f(net_{o_1}) = \frac{1}{1+e^{-net_{o_1}}}$ 이 된다. net_{o_2}, out_{o_2}도 마찬가지.

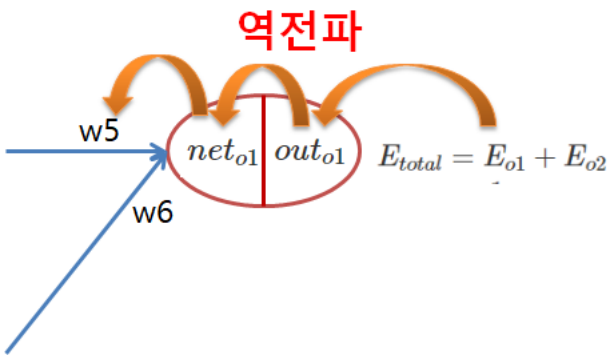
이렇게 구한 out_{o_1}, out_{o_2}는 최종 output이 되며 이를 다시 $target_{o_1}, target_{o_2}$와 비교해 오차를 계산한다.

$E_{total} = E_{o_1} + E_{o_2}$
$= \frac{1}{2}(total_{o_1} - out_{o_1})^2 + \frac{1}{2}(total_{o_2} - out_{o_2})^2$ 로 계산할 수 있다.

가중치 계산 단계

이제 가중치를 갱신하는데, 출력단에서 가까운 것부터 갱신한다. 편의상 w_5^+를 가중치가 갱신된 w_5라 한다.

$w_5^+ = w_5 + \alpha \frac{\partial E_{total}}{\partial w_5}$ 인데, $\frac{\partial E_{total}}{\partial w_5}$ 계산을 직접적으로 하기에는 E_{total}에 w_5 term이 직접적으로 들어있지 않다. 물론, E_{total}을 w_5에 관한 식으로 바꾸어 줄 수도 있지만 이는 번거롭기 때문에 chain rule을 이용해 식을 변형하자.



$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} \frac{\partial out_{o_1}}{\partial net_{o_1}} \frac{\partial net_{o_1}}{\partial w_5}$처럼 (위의 figure 참고) 오차를 w_5까지 역전파 시키는 것이다. Input layer와 output layer 사이의 weight을 갱신하는 것도 위의 방식대로 할 수 있다.

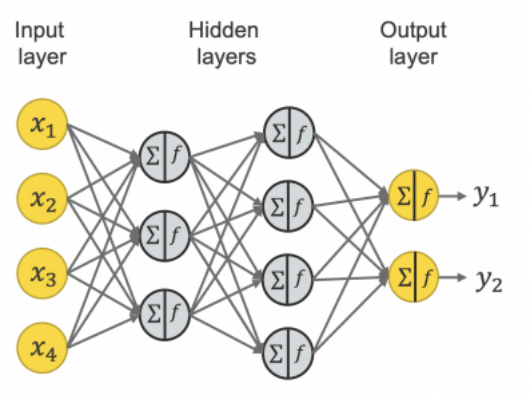
XOR with PyTorch

```
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
model = torch.nn.Sequential(linear1, sigmoid, linear2, sigmoid).to(device)
```

`model`이 실행될 동안 input이 `linear1` → `sigmoid` → `linear2` → `sigmoid` 순으로 통과




```
# define cost/loss & optimizer
criterion = torch.nn.BCELoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=1)  # modified learning rate from 0.1 to 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())
```

```
# Accuracy computation
# True if hypothesis>0.5 else False
with torch.no_grad():
    hypothesis = model(X)
    predicted = (hypothesis > 0.5).float()
    accuracy = (predicted == Y).float().mean()
    print('\nHypothesis: ', hypothesis.detach().cpu().numpy(), '\nCorrect: ', predicted.detach().cpu().numpy(), '\nAccuracy: ', accuracy.it
```

Multi-layer perceptron을 사용하면 XOR 문제도 잘 해결할 수 있다.