20.04.13

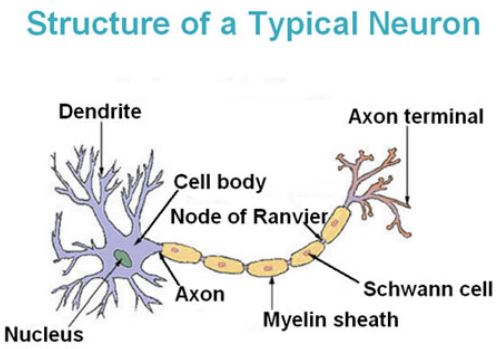
주제	**Perceptron / Multilayer Perceptron (MLP)**

Perceptron

모델

Perceptron은 Neural Network의 모델 중 하나이다.

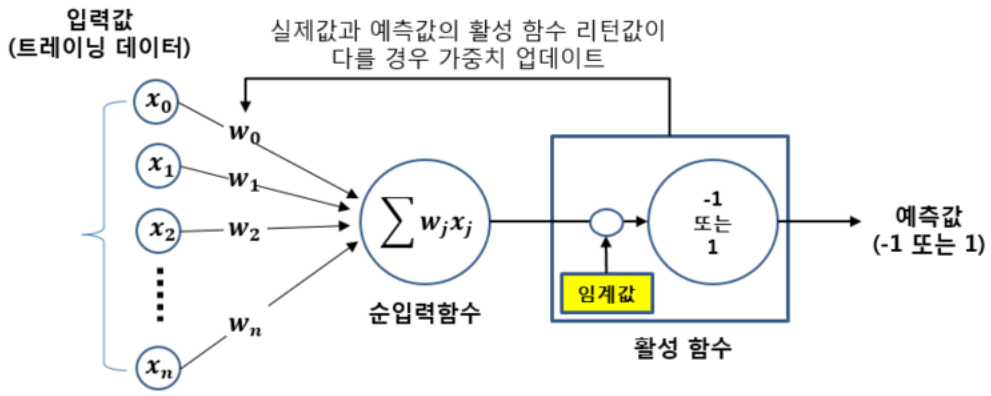
Neural Network는 인간의 두뇌가 작동하는 방식에서 착안한 개념이다.



인간의 신경 세포인 뉴런은 Dendrite로 부터 신호를 입력받아 신호량이 임계점을 넘기면 다음 뉴런으로 신호를 전달한다.

구현

Perceptron도 이와 마찬가지로 여러 신호를 입력받고 신호의 총합이 임계값을 넘었을 때 1을 출력한다. 임계값을 넘지 못했다면 0을 출력한다.



입력값은 일반적으로 데이터의 특성(feature)을 나태내는 값으로 이루어져 있다.
이 값에 가중치를 모두 곱하여 하나의 값으로 만드는 함수를 net input 함수라고 한다.
net input 함숫값과 임계값을 비교하여 출력 여부를 결정하는 함수가 Activation function이다.

Singlelayer Perceptron에서의 학습은 데이터에 포함된 정답값과 활성함수의 결과값을 비교,
오차(loss / cost)가 최소가 되도록 w를 업데이트 하는 방식으로 진행된다.

bias

퍼셉트론의 활성함수를 풀어쓰면 다음과 같다.

Output

$$w_1x_1 + w_2x_2 + \ldots + w_nx_n > \theta \Rightarrow 1$$

$$w_1x_1 + w_2x_2 + \ldots + w_nx_n \leq \theta \Rightarrow 0$$

SLP의 경우 활성함수는 net input 함수의 출력값을 임계점과 비교할 뿐이다.

좌변이 net input 함수의 출력값이고 이고 우변이 임계값 θ이다.

여기서 임계값 θ를 를 -b로 치환하여 좌변으로 넘기면

b + w1×1 + … + wnxn <0 ⇒ 0

b + w1×1 + … + wnxn >=0 ⇒ 1

입력값 x 가중치를 제외한 상수항 b를 bias라고 정의하는데

이에따르면 Perceptron에서 bias는 결국 임계값을 의미한다는 것을 알 수 있다.

node 에서 언제 1을 출력할 것인지를 조절하는 parameter인 셈이다.

bias 값이 높으면 높을 수록 그만큼 분류의 기준이 엄격하다는 것을 의미한다.

웬만해서는 1을 출력하지 않는다는 뜻이므로.

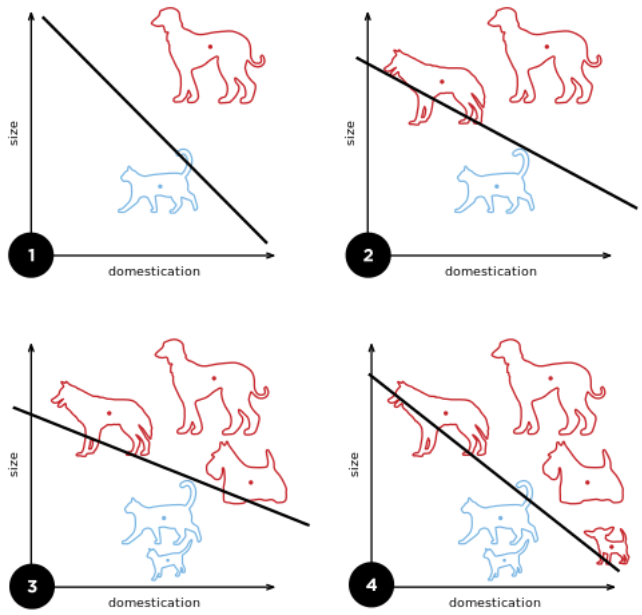
따라서 bias가 높을 수록 모델이 간단해지는 경향이 있다. (변수가 적고 더 일반화 될 것이다.)
그러므로 bias값이 지나치게 크면 underfitting의 위험이 있다.

반대로 bias 값이 작아지면 허용범위가 넓어지는 만큼 필요 없는 노이즈가 포함될 가능성도 높다.
이 경우 반대로 overfitting의 위험이 있는 것.

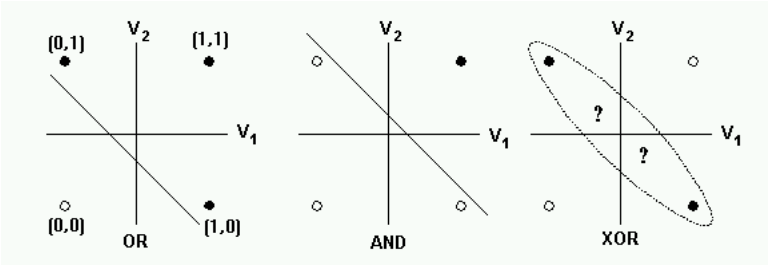
Single Layer

Singlelayer Perceptron의 한계

퍼셉트론은 모든 학습 데이터를 정확히 분류시킬 때까지 학습이 진행되기 때문에 학습 데이터가
다음과 같이 binary classification이 가능하여 선형적으로 분리될 수 있을때에는 적합하다.



그러나 XOR 과 같이 비선형적으로 분리되는 데이터 분포에 적용하기에는 한계가 있다.



▼ XOR을 singlelayer로 구현한 코드

```
# nn Layers
linear = torch.nn.Linear(2, 1, bias=True)# perceptron은 1개의 layer를 갖는 구조이므로
sigmoid = torch.nn.Sigmoid()

# model
model = torch.nn.Sequential(linear, sigmoid).to(device)

# define cost/Loss & optimizer
criterion = torch.nn.BCELoss().to(device)# binary cross_entropy loss
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())

# 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: ',
```

▼ 그 결과 문제점

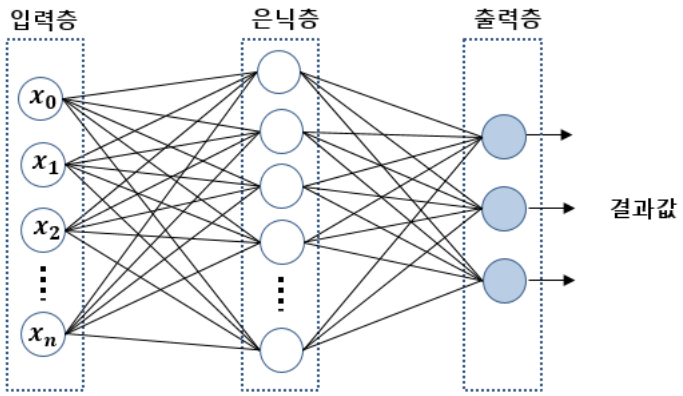
```
0 0.7666423320770264
100 0.6931473016738892
200 0.6931471824645996
300 0.6931471824645996
400 0.6931471824645996
500 0.6931471824645996
600 0.6931471824645996
700 0.6931471824645996
800 0.6931471824645996
900 0.6931471824645996
1000 0.6931471824645996
1100 0.6931471824645996
1200 0.6931471824645996
1300 0.6931471824645996
1400 0.6931471824645996
.
.
.
(생략)

Hypothesis:  [[0.5]
 [0.5]
 [0.5]
 [0.5]]
Correct:  [[0.]
 [0.]
 [0.]
 [0.]]
Accuracy:  0.5
```

200번 이후부터 loss 값이 줄어들지 않는데, 이는 학습이 제대로 되지 않는다는 것을 의미한다. 만번의 학습이후 각 X네가지에 대한 결괏값을 보면 perceptron이 0.5만 output으로 내게 된다.

Multilayer Perceptron (MLP)

정의



Singlelayer Perceptron의 한계를 극복하기 위해 입력층과 출력층 사이에 하나 이상의 중간층을 두어 비선형적으로 분리되는 데이터에 대해서도 학습이 가능하도록 한 것이 Multilayer Perceptron이다.

구조

입력층과 출력층 사이 존재하는 중간층을 숨어 있는 층이라 하여 은닉층이라고 부른다.
은닉층이 여러개 있는 Neural Network를 Deep Neural Network(DNN) 이라고 부르며
DNN을 학습하기 위해 고안된 알고리즘을 바로 딥러닝 이라고 하는 것이였다...!

MLP에서는 입력층에서 전달되는 모든 값이 은닉층의 모든 노드로 전달된다.
은닉층의 출력값 역시 마찬가지.

이런 형식으로 값이 전달되는 것이 순전파(feedforward)이다.

입력층, 은닉층, 출력층의 노드수가 각 n,m,h 인 MLP를 n-m-h MLP라고 부른다.

Back propagation

- Back propagation의 목적

우리가 지금까지 배웠던 간단한 선형 회귀(linear regression)모델(wx+b)에서 가중치(w)를 데이터에 맞게 조절하기 위해서, w에 대한 전체 모델의 gradient를 구해 이와 비례하는 값을 w에 빼는 방법을 취했습니다.

gradient는 다변수 함수 f(x1, x2, x3, ...)의 특정 지점에서 가장 가파르게 증가하는 방향을 가리키는 벡터를 구하는 데 주로 쓰는 연산입니다.(수학적인 정의는 아니지만, 가장 유용한 성질입니다.) 저희는 거기에 -를 붙여서 가장 급격히 내려가는 방향을 구하도록 만들어서 최종적으로는 local minimum이 도달하는 방식으로서 쓰는 것이고요.

아무튼 간에 여러 feature를 input으로 갖는 모델의 cost function 은 w1, w2, ... 라는 변수들을 가진 다변수 함수와도 동일합니다. cost function을 대충 C라고 한다면, C의 Gradient vector는 이런 수식으로 표현됩니다.

$$grad(C) = \nabla C(w_1, w_2, w_3) = \frac{\partial C}{\partial w_1}\frac{\vec{w_1}}{|\vec{w_1}|} + \frac{\partial C}{\partial w_2}\frac{\vec{w_2}}{|\vec{w_2}|} + \frac{\partial C}{\partial w_3}\frac{\vec{w_3}}{|\vec{w_3}|}$$

여기에서 w1 방향 성분인 편미분 항은 w1에 대한 gradient에 해당하는 것입니다. 여러 layer로 겹겹이 쌓여 입력과 출력을 layer 간에 교환하는 구조에서 이 항을 구하기 위해서 back propagation이라는 개념이 등장합니다.

Chain rule

multi layer 구조를 생각해보면, 전 단계 layer의 결과를 입력으로 받으려고 하는 것이, 우리가 알고 있는 합성함수의 개념과도 동일합니다. 실제로 수학적으로도 그렇게 나타나는 것 같습니다.

$$C(J(w_1, w_2), w_3)$$

수학적으로 나타낸다면 이런 느낌이 아닐까 싶습니다. 이런 상황에서

$$\frac{\partial C}{\partial w_2}$$

같은, 앞쪽 레이어의 가중치에 대한 gradient 값을 구하기 위해선 합성함수의 미분 방법이 필요하겠죠. 그 것과 일맥상통인 게 Chain rule 입니다.

- Chain Rule의 수학적 표현

$$\frac{\partial f}{\partial w_0} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial w_0}$$

▼ 구현코드

```
# Back prop (chain rule)
    # binary_cross_entropy loss를 미분한 식
    d_Y_pred = (Y_pred - Y) / (Y_pred * (1.0 - Y_pred) + 1e-7)
    # 마지막 항 1e-7은 0으로 나누어지는 경우를 막아주기 위한 term

    # 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) # transpose(x, y, z) : y 와 z 차원을 서로 swap시켜라
    # 여기서 a1은 (4, 2)였는데 (2, 4)로 바껴서 d_b2(= d_l2)즉 (4, 1)과 행렬곱이 가능해짐

    # 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)

    # Weight update
    # gradient descent를 minimize 시키는 원리
    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)
```