김범수

뉴럴 네트워크

Neural Network

By POSCAT

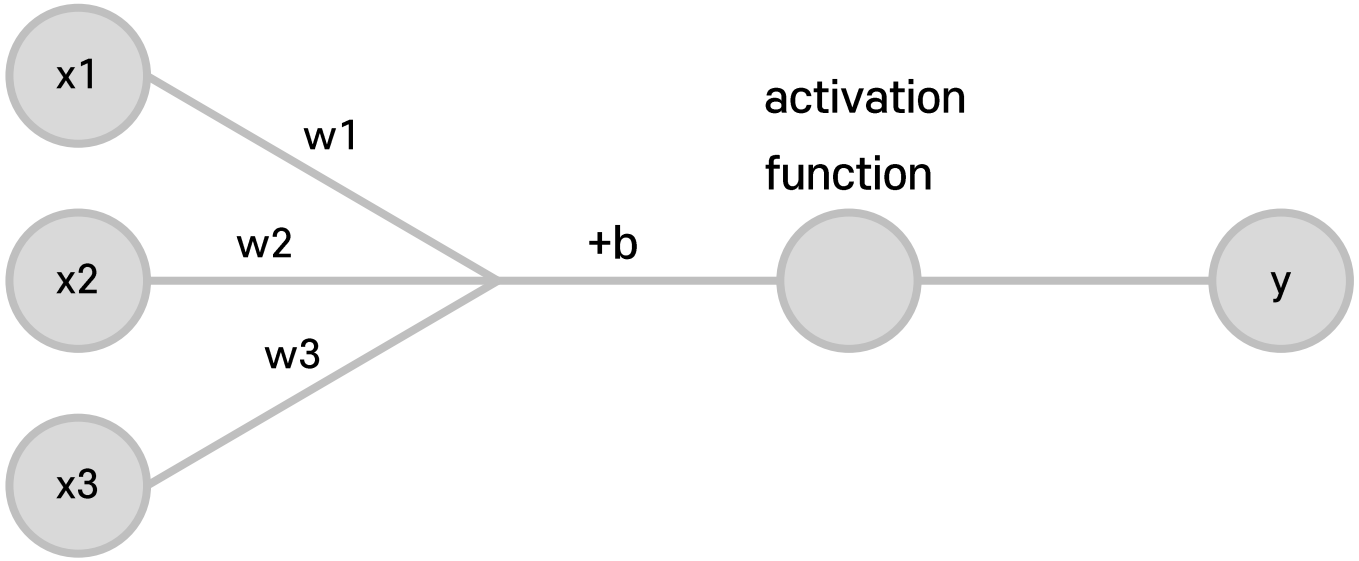
Contents

Perceptron

- 뉴럴 네트워크는 생각보다 오래 전부터 연구된 분야입니다. 하지만 초기의 뉴럴 네트워크는 매우 빈약했고, 여러 번의 발전 과정을 거쳐 현재까지 오게 된 것입니다. 지금부터 뉴럴 네트워크의 발전 과정을 따라가면서 뉴럴 네트워크를 이해해 보겠습니다.
- Perceptron은 사람의 신경인 뉴런을 모방한 가장 간단한 모델입니다. 사람의 뉴런은 역치 이상의 수치에서만 반응을 한다고 알려져 있습니다. 따라서 뉴런의 특징을 모방한 perceptron이 개발됩니다.

Perceptron

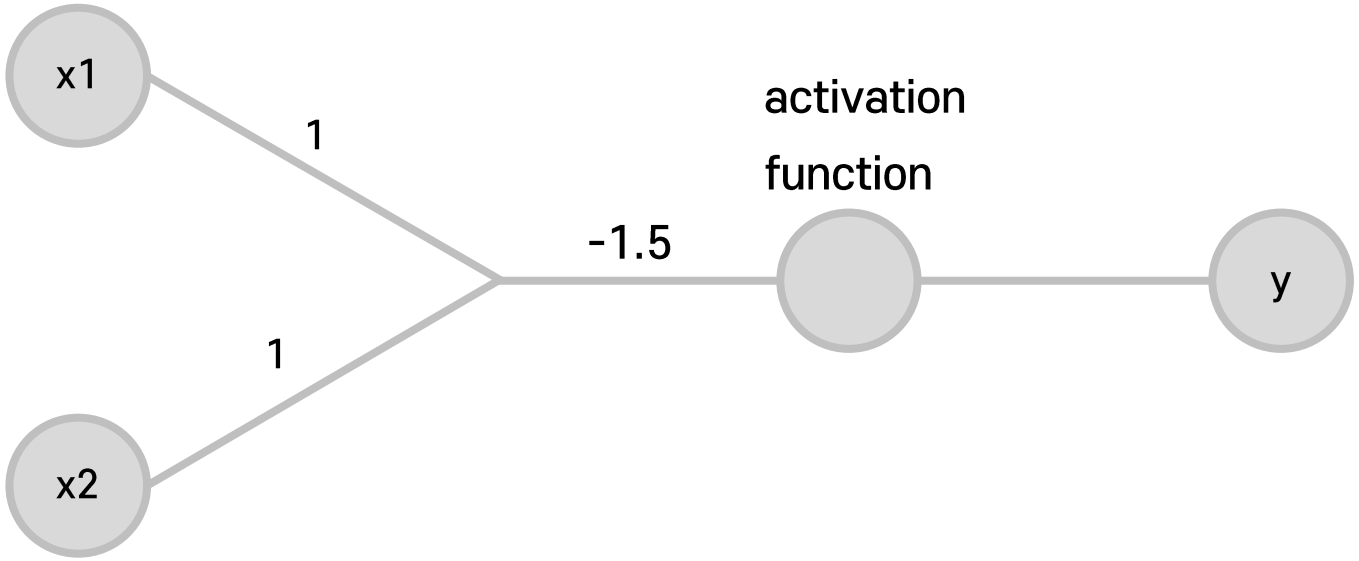
- Perceptron의 구조는 아래와 같습니다. 하나의 perceptron은 여러 개의 입력 x1, x2…을 받아 0 또는 1을 출력합니다.
- Perceptron은 w1, w2 등의 parameter로 구성되어 있습니다.
- 이 때 activation function은 input값이 일정 값보다 크면 1, 아니면 0을 출력하는 함수로, step function이라 불립니다.
- 초기에는 perceptron을 이용해 AND gate, OR gate 등의 문제를 해결하였습니다.

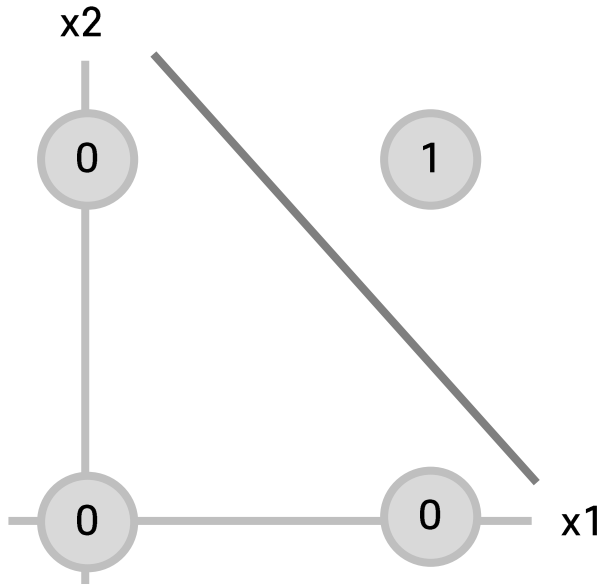


$$y = \begin{cases} 0 \; if \; x1w1 + x2w2 + x3w3 + b < 0 \\ 1 \; if \; x1w1 + x2w2 + x3w3 + b \geq 0 \end{cases}$$

Perceptron

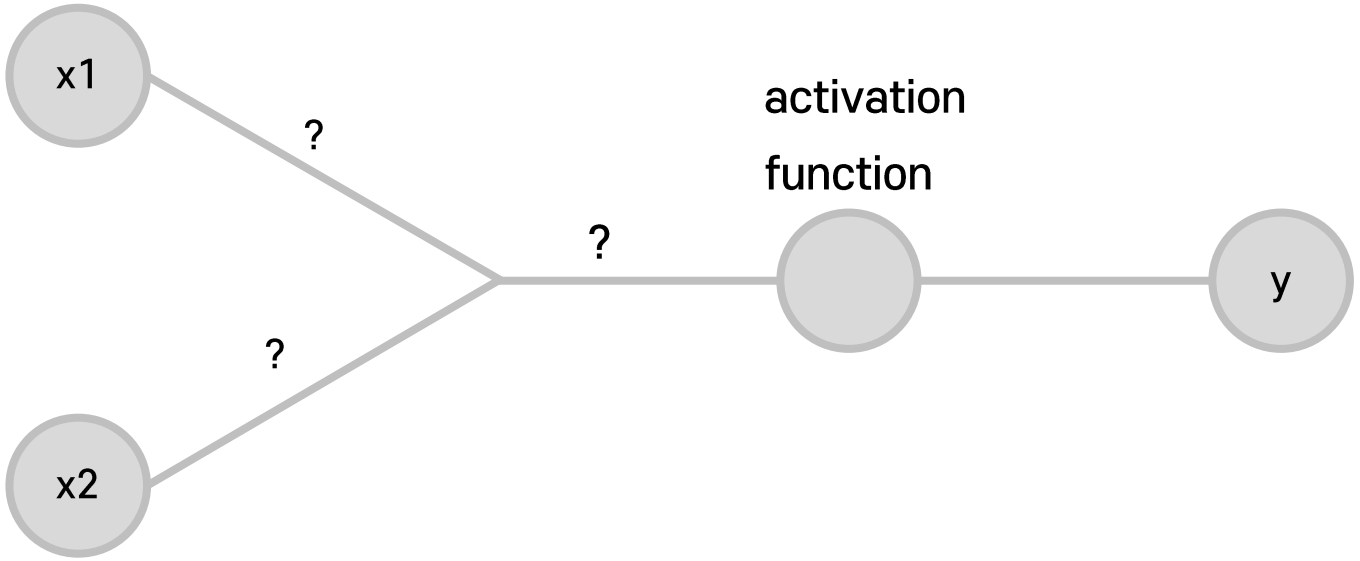
- AND gate는 x1, x2가 모두 1이면 1을 반환하고, 둘 중 하나라도 0이면 0을 반환하는 함수입니다. (x1, x2는 0 또는 1의 값만 가질 수 있습니다.)
- 아래의 perceptron을 보면, x1, x2의 합이 2일 때만 y값이 1이 되므로, AND gate와 동등한 기능을 함을 알 수 있습니다.
- 하지만, perceptron의 한계는 XOR gate에서 오게 됩니다.

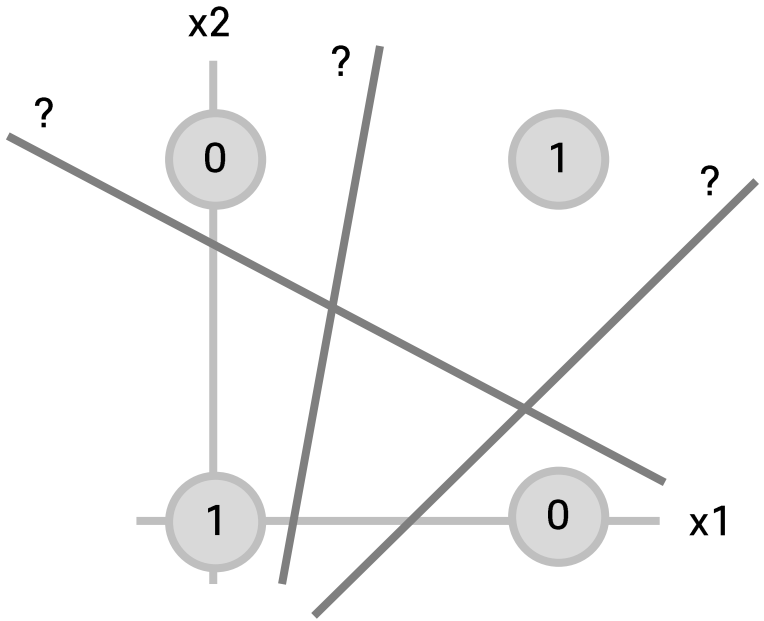




Perceptron

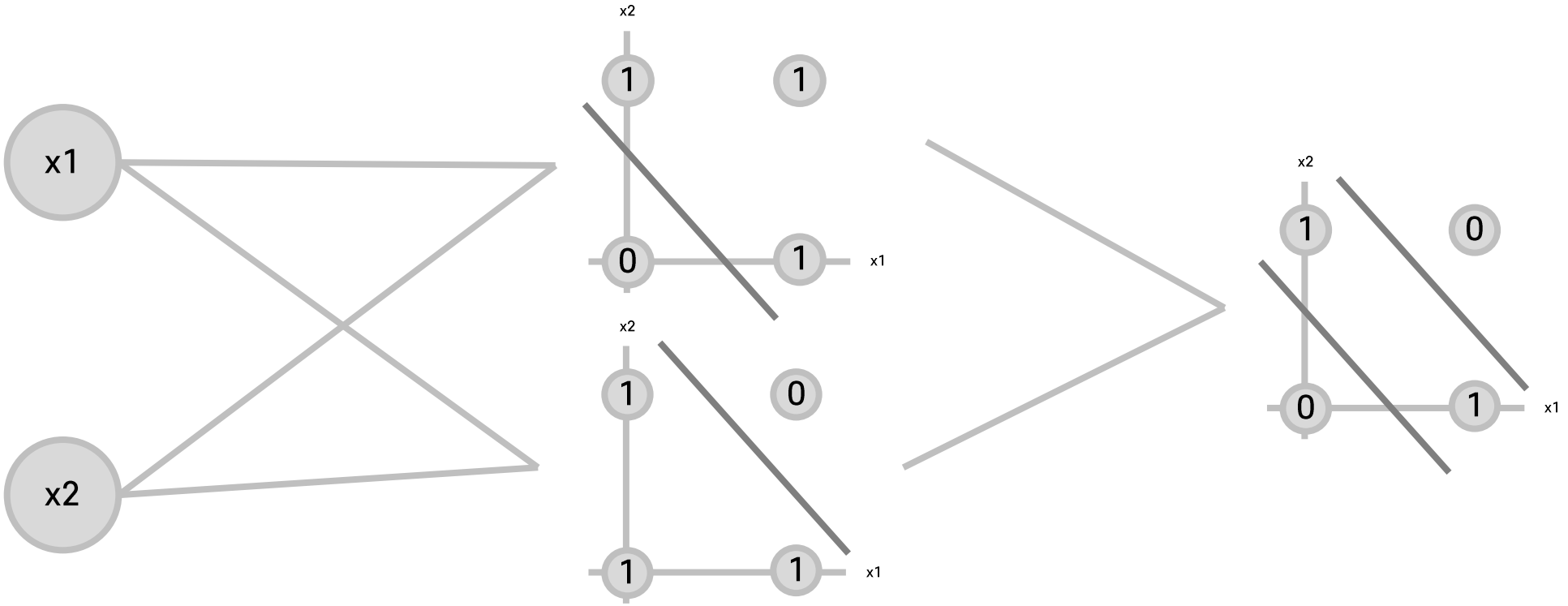
- XOR gate는 x1, x2가 같은 수라면 1, 다른 수라면 0을 반환하는 함수입니다. (x1, x2는 0 또는 1의 값만 가질 수 있습니다.)
- 하지만 역치를 기준으로 0, 1을 출력하는 perceptron은 XOR gate를 만들 수 없음을 알게 되고, 뉴럴 네트워크는 다시 침체기에 빠지게 됩니다.





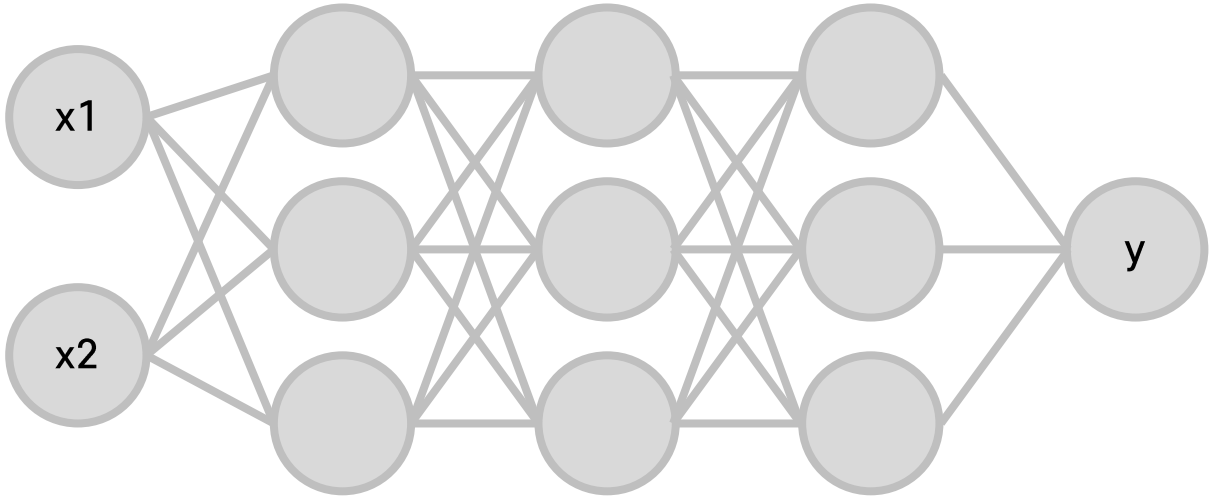
Perceptron

- 이 때 perceptron의 한계를 해결하기 위해 Multi layer perceptron이 고안됩니다. Perceptron 하나로는 XOR gate 문제를 해결하지 못한다면, 여러 개의 perceptron을 사용하면 괜찮지 않을까 하는 생각에서 출발합니다.
- 아래 그림처럼 perceptron에서 나온 결과를 또 다른 perceptron의 input으로 주는 방식을 고안합니다.



Perceptron

- 이처럼 여러 개의 perceptron 층(layer) 만드는 아이디어를 고안하였고, 이를 multi layer perceptron(MLP)이라고 칭하게 됩니다.
- 하지만, 일일이 parameter를 설정해서 함수를 만드는 것은 매우 어려웠습니다. 따라서 뉴럴 네트워크는 다시 정체기에 빠지게 됩니다.



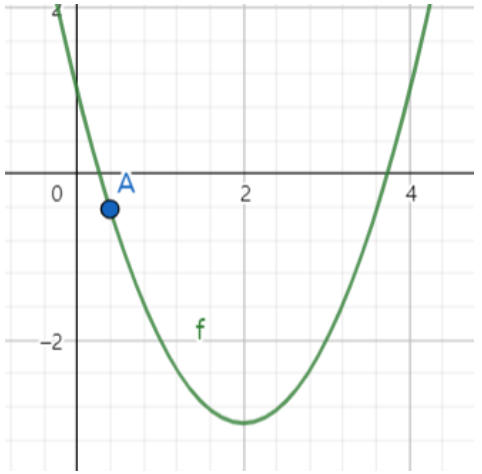
Gradient Descent

- 아주 간단한 문제 하나를 생각해 봅시다. y = ax + b라는 그래프와 (x, y) 데이터가 있을 때, 이 데이터를 잘 만족하는 a, b를 구하고 싶다고 합시다.
- 예를 들어 (x, y) = (3, 5), (4, 6), (5, 7)이라는 세 데이터가 있을 때, y = x + 3이라는 그래프는 이 데이터를 정확하게 설명합니다.
- 하지만, (x, y) = (3, 5), (4, 6.1), (5, 6.8)와 같은 데이터가 있으면, 이 데이터를 잘 표현하는 a, b값을 정의하는 것이 애매합니다.
- 따라서, 여기서는 cost function을 정의합니다. cost =$\sum(\hat{y}_i - y_i)^2$ 으로 정의합니다. 여기서 $\hat{y}_i$ 는 우리가 예상한 y값, y_i는 실제 y값으로 $\hat{y}_i = ax_i + b$가 됩니다. 이 때, 우리의 목표는 a, b값을 잘 조절해서 cost가 최소가 되도록 하는 것입니다.

Gradient Descent

- (x, y) = (2, 4), (4, 7), (8, 8) 이라는 3개의 데이터가 있을 때, cost를 최소화하는 parameter a, b를 구해봅시다.
- cost = $(2a + b - 4)^2 + (4a + b - 7)^2 + (8a + b - 8)^2$가 되고, 이를 최소화 해야 합니다.
- 미적분학의 지식을 이용해서 문제를 풀면, $\frac{\partial cost}{\partial a} = \frac{\partial cost}{\partial b} = 0$의 방정식을 풀어서 a, b값을 구하면 됩니다. 즉, cost가 최소가 되도록 하는 a, b값을 구하는 것이 되죠. 하지만 cost function이 복잡해지면 사람의 힘으로 이를 푸는 것은 불가능에 가까워집니다. 따라서 컴퓨터도 문제를 풀 수 있게 해주는 기법인 gradient descent가 도입됩니다.

Gradient Descent



- 오른쪽과 같은 함수에서 현재 A의 좌표에 해당하는 x값을 가지고 있다고 가정하겠습니다. 그리고, 우리는 x값을 바꿔가면서 y값이 점점 줄어들게 하고 싶습니다. 그렇다면, 우리는 x값을 조금씩 키워가면서 y값이 줄어들게 할 것입니다.
- 컴퓨터는 x값이 줄어들어야 할 지, 커져야 할 지 어떻게 판단할 수 있을까요? 컴퓨터는 기울기를 통해 이를 판단합니다. 오른쪽 그림에서 A점에 해당하는 좌표는 기울기가 음수이므로, x가 커지는 방향으로 이동하면 y값이 작아진다는 것을 알 수 있습니다.
- 따라서 이러한 아이디어를 적용해서, x로 미분한 값이 양수라면 x를 줄이고, x로 미분한 값이 음수라면 x를 증가시키는 알고리즘이 gradient descent algorithm입니다.
- 만약 변수가 여러 개라면, 편미분을 이용해서 기울기를 계산합니다. 다음 슬라이드에서 a, b값을 구하는 알고리즘의 수도 코드를 소개합니다.

Gradient Descent

Code Explanation

총 n개의 data {(x1, y1), (x2, y2)… (xn, yn)} 이 존재한다.

set proper α

initialize parameter a, b arbitrary

until cost is minimum:

a' = $\frac{\partial cost}{\partial a}$

$b' = \frac{\partial cost}{\partial b}$

$a = a - \alpha b'$

b = $b - \alpha b'$

Gradient Descent

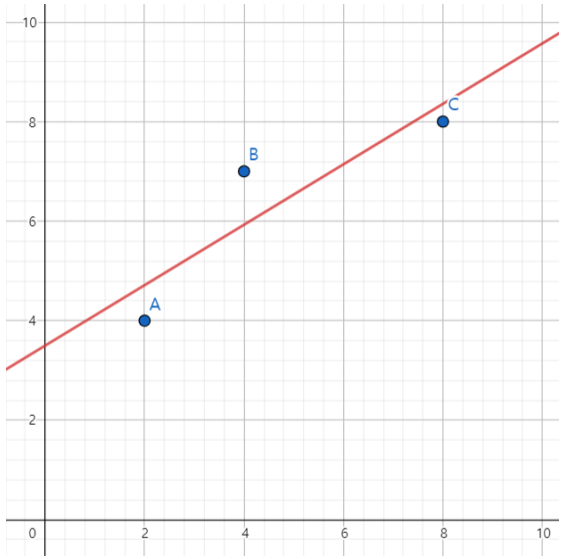
- (x, y) = (2, 4), (4, 7), (8, 8)라는 데이터가 주어지고, $\alpha = 0.001$, a = b = 0으로 초기화 되었다고 가정하고 gradient descent algorithm을 적용해서 a, b값을 계산해보겠습니다.
- 먼저 cost를 a, b로 미분한 식을 구해봅시다.
- $\frac{\partial cost}{\partial a} = \sum 2(ax_i + b - y_i)x_i = 2 * (0 * 2 + 0 - 4) * 2 + 2 * (0 * 2 + 0 - 7) * 4 + 2 * (0 * 2 + 0 - 8) * 8$
  $= -16 - 56 - 128 = -200$
- $\frac{\partial cost}{\partial b} = \sum 2(ax_i + b - y_i) = 2 * (0 * 2 + 0 - 4) + 2 * (0 * 2 + 0 - 7) + 2 * (0 * 2 + 0 - 8)$
  $= -8 - 14 - 16 = -38$
- 따라서 $\mathrm{a} = \mathrm{a} - \alpha \frac{\partial cost}{\partial a} = 0.2, b = b - \alpha \frac{\partial cost}{\partial b} = 0.038$로 update됩니다.
- 이 과정을 무수히 반복하면 a, b값에 가까워지게 됩니다.

Gradient Descent

- Gradient Descent를 수행하는 알고리즘을 직접 코드로 짜서 실제로도 잘 되는 지 확인해 봅시다. 실제로 오른쪽의 코드로 구한 a, b값으로 그래프를 그려보면, 꽤 정확한 a, b값이 나온다는 것을 알 수 있습니다.
- 이 때, alpha값은 기울기에 곱해지는 상수 값입니다. 만약 alpha값이 너무 크거나 너무 작다면 어떻게 될까요?

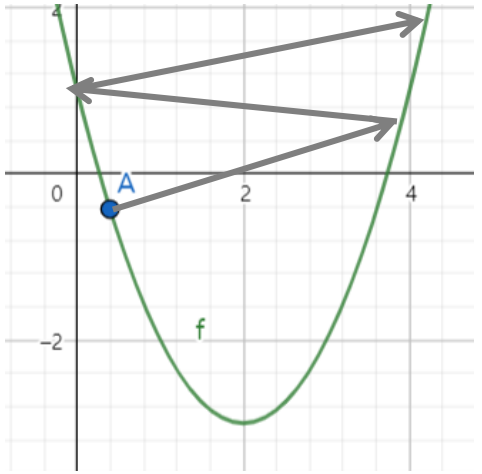
```python
data = [(2, 4), (4, 7), (8, 8)]
alpha = 0.001
a = b = 0
for rep in range(100000):
    da = db = 0
    for x, y in data:
        da += 2 * (a * x + b - y) * x
        db += 2 * (a * x + b - y)
    a = a - alpha * da
    b = b - alpha * db
print(a, b)
```

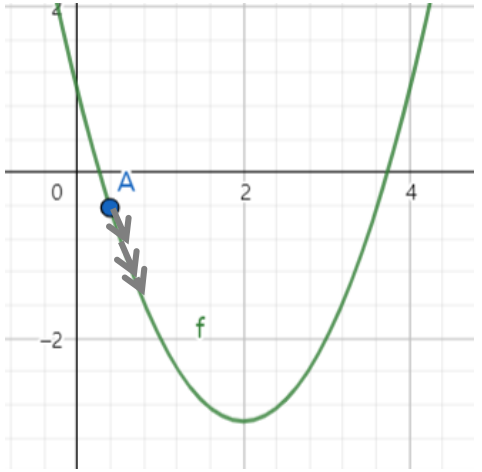
```
0.6071428571428886 3.4999999999998286

***Repl Closed***
```



Gradient Descent

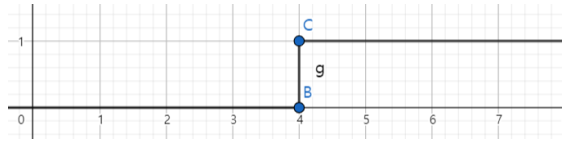
- alpha값이 너무 크다면, 변수가 우리가 원하는 것보다 훨씬 크게 바뀌게 됩니다. 따라서 오른쪽 위의 그림처럼 변수가 발산하게 됩니다.
- 반면에 alpha값이 너무 작다면, 변수가 너무 조금씩 바뀌게 됩니다. 따라서 오른쪽 아래의 그림처럼 y값이 최소가 되도록 하는 변수를 찾는 데 오래 걸리게 됩니다.
- 이처럼 alpha값이 클수록 변수가 학습되는 속도가 결정되므로, learning rate라 불립니다.





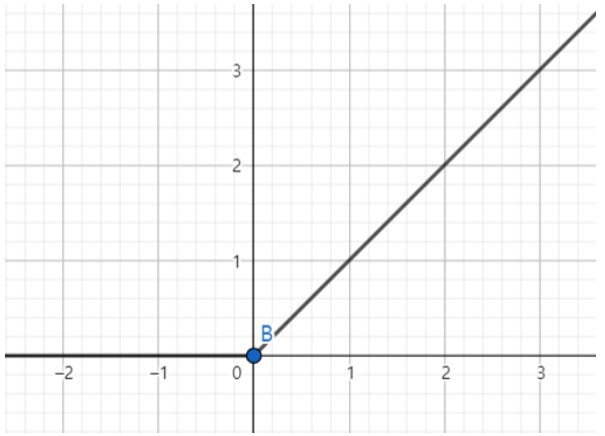
Gradient Descent

- 이처럼 Gradient Descent를 이용하면, cost가 최소가 되도록 하는 parameter를 컴퓨터가 알아서 학습하도록 할 수 있습니다. 그렇다면 아까 배운 MLP에 Gradient Descent를 적용할 수 있을까요? 결론부터 말하자면, activation function이 step function이면 적용할 수 없습니다.
- step function의 개형은 오른쪽과 같이 일정 값보다 작으면 0, 아니면 1을 가지는 함수입니다. 따라서 경곗값을 제외한 어느 점에서 미분해도, 미분값이 0이 되므로 gradient descent를 적용할 수 없습니다.

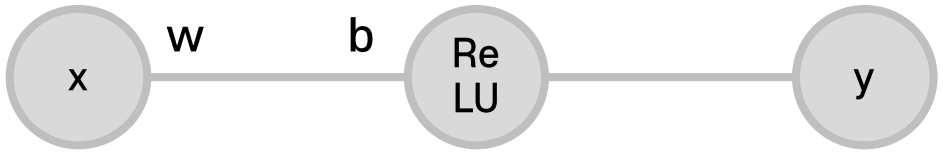


Gradient Descent

- 따라서 activation function을 ReLU라는 함수로 바꿉니다. ReLU는 간단한 함수로 오른쪽과 같이 음수면 0을 반환하고, 양수면 입력값을 반환하는 간단한 함수입니다. 따라서 입력값이 음수면 미분값이 0, 입력값이 양수면 미분값이 1이 되어 미분 계산도 간단하다는 장점이 있습니다.



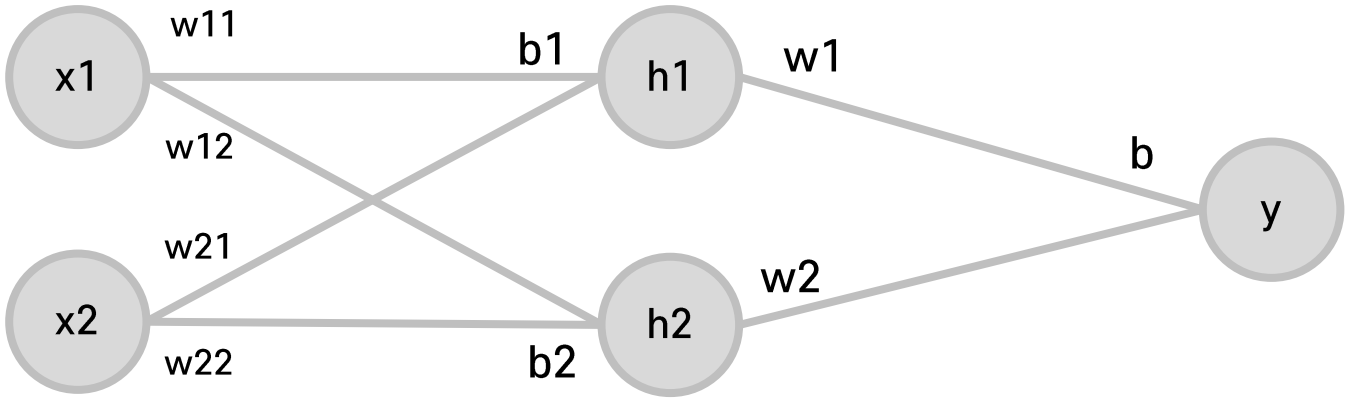
- 우리가 이전에 풀었던 문제는 아래의 perceptron을 학습시키는 것과 동등함을 알 수 있습니다. (데이터의 y가 모두 양수라고 가정합니다.)



Gradient Descent

- 그렇다면 MLP에서는 미분값을 어떻게 계산할까요? 예를 들어 아래의 MLP에서는 총 9개의 변수가 존재합니다. 마찬가지로 (x1, x2, y)의 n개의 데이터가 있고, cost = $\sum(\hat{y}_i - y_i)^2$라 정의한다면, 미분값을 어떻게 구할 수 있을까요?
- 먼저 cost를 b로 미분한 값은, cost=$\sum((h1w1 + h2w2 + b)) - y_i)^2$ 를 b에 대해서 단순히 미분하면 되므로,

  $\frac{\partial cost}{\partial b} = \sum 2((h1w1 + h2w2 + b)) - y_i)$ 로 구할 수 있습니다. 하지만, cost를 b1으로 미분한 값은 어떻게 구할 수 있을까요?



Gradient Descent

- MLB에서는 $\frac{\partial cost}{\partial b1} = \frac{\partial cost}{\partial h1}\frac{\partial h1}{\partial b1}$라는 사실을 이용해서 구합니다.

  $\frac{\partial cost}{\partial h1} = \sum 2\big((h1w1 + h2w2 + b)) - y_i\big) * w1$ 이 되고,

  $\frac{\partial h1}{\partial b1} = \frac{\partial}{\partial b1}\sum(w11x1 + w21x1 + b1)^2 = \sum 2(w11x1 + w21x1 + b1)$ 가 됩니다.

  따라서 두 결과를 곱해서 cost를 b1으로 미분한 값을 구할 수 있습니다.

- 이처럼 MLB에서도 미분값을 구할 수 있습니다.

Neural Network

- 지금까지 배운 MLP를 neural network (인공 신경망), 줄여서 nn이라고 합니다. nn은 여러 분야에서 사용됩니다. 분류 (이미지 분류, 글씨 분류), clustering(유사한 특징을 가진 데이터들끼리 묶기) 등에 사용됩니다. 이러한 nn의 핵심은 선형 함수 여러 개를 합쳐 우리가 원하는 비선형 함수를 만드는 것입니다.
- python에는 머신러닝을 지원하는 모듈이 여러가지 종류가 존재합니다. 이 중, 대표적인 모듈은 tensorflow와 pytorch입니다. tensorflow는 pytorch에 비해 사용에 제약이 있어 최초의 머신러닝 모듈임에도 불구하고 그 위상이 떨어진 상태입니다. 따라서 본 강의에서는 널리 사용되는 pytorch를 사용하여 강의를 진행합니다.
- 이후의 강의부터는 list로 저장할 수 없는 수많은 state에 대해 action value를 계산하기 위해 nn를 사용합니다.

Neural Network

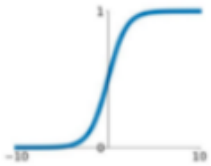
- 지금까지 배운 내용으로 neural network를 재정의해보겠습니다.
- Neural network는 MLP로, 수많은 perceptron들로 구성됩니다. 이 때, MLP의 첫 layer는 input layer, 마지막 layer는 output layer, 중간의 layer는 hidden layer라고 불립니다.
- 하나의 perceptron은 parameter와 activation function으로 구성됩니다. 이 때, activation function은 존재하지 않을 수도 있습니다.
- cost function을 정의하여, cost function의 gradient(미분값)를 계산하여 parameter에 반영함으로써 neural network를 학습시킵니다. 이 때, gradient는 계산 속도를 위해 output layer의 gradient를 먼저 계산하고, input layer를 마지막으로 계산하는 방식으로 계산하므로, 이 과정을 backpropagation(역전파)라고 합니다.
- 우리는 gradient값에 단순히 상수를 곱해서 parameter에 빼 줌으로서 parameter를 update하였습니다. 이와 같이 parameter를 update하는 것을 optimize라 합니다. 따라서 parameter를 update하는 식을 optimizer라고 하며, 우리가 배운 update식인 $x = x - \alpha \frac{\partial y}{\partial x}$은 가장 기초적인 optimizer라 할 수 있습니다.

Neural Network

- nn에 대한 추가적인 내용을 소개하겠습니다.
- 먼저 가장 많이 사용되는 두 activation function은 전에 소개한 ReLU와 sigmoid입니다. ReLU는 출력이 양수 값 전체일 때, sigmoid는 출력이 0이상 1이하의 실수로 하고 싶을 때 사용되며, 각 함수의 식과 개형은 아래와 같습니다. 보통 반환하는 값이 확률에 해당할 때, output layer의 activation function으로 sigmoid를 사용합니다.

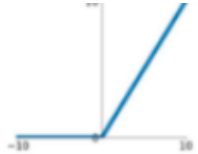
Sigmoid

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



ReLU

$$\max(0, x)$$



- Sigmoid는 미분값이 0.5를 넘을 수 없으므로, activation function으로 sigmoid만 사용하면, input layer에 가까운 쪽의 layer의 gradient가 매우 작아져서 거의 없어진다는 단점이 존재합니다.
- ReLU는 음수일 경우, 미분값이 0이므로, 음수인 perceptron의 parameter는 update되지 않아 죽어버린다는 단점이 있습니다.
- 우리가 가장 많이 사용할 optimizer는 Adam이라 불리는 optimizer입니다. 단순히 gradient가 아니라 관성 등 여러 개념을 추가한 optimzer입니다. 우리가 처음에 배운 opitimizer는 SGD라 불리는 optimizer입니다.

Pytorch

- 이제부터 본격적으로 pytorch를 사용해보겠습니다. 1강에서 말했듯이 파이썬의 가장 큰 장점은 외부 모듈을 쉽게 설치해서 사용할 수 있다는 것입니다. 지금부터 pytorch를 설치해보겠습니다.
- 먼저 cmd창을 실행합니다. (관리자 권한으로 실행하지 않으면 오류가 발생하니 “무조건” 관리자 권한으로 실행하세요.)
- 파이썬에서는 pip을 이용해 외부 모듈을 설치 및 관리합니다. pip을 최신 버전으로 업그레이드 합니다. pip install --upgrade pip 명령어로 할 수 업그레이드 할 수 있습니다.

- 만약 pip이라는 명령어를 cmd에서 인식하지 못한다면, python을 설치할 때, PATH가 제대로 설정되지 않은 것이므로, 1강을 참조해서 python을 삭제 후 재설치하시기 바랍니다.

Pytorch

- 이제 pip install torch 명령어로 pytorch를 설치해 줍니다. (저의 경우는 이미 설치가 되어있어 추가 설치가 이루어지지 않았습니다.)

```
C:₩WINDOWS₩system32>pip install torch
Requirement already satisfied: torch in c:₩u
Requirement already satisfied: typing-extens
ges (from torch) (3.7.4.3)
Requirement already satisfied: numpy in c:₩u
rch) (1.19.5)
```

- pip list 명령어를 통해 지금까지 설치한 외부모듈 목록을 볼 수 있습니다.
  오른쪽처럼 모듈에 torch와 numpy가 있다면 성공적으로 설치된 것입니다.

- pytorch에서는 nn, layer, optimizer 등 지금까지 배운 모든 내용을 지원합니다.
  지금부터 천천히 알아보도록 하겠습니다,

```
C:₩WINDOWS₩system32>pip list
Package            Version
------------------ -------
cloudpickle        1.6.0
future             0.18.2
gym                0.18.0
joblib             1.0.0
MouseInfo          0.1.3
numpy              1.19.5
Pillow             7.2.0
pip                21.0
PyAutoGUI          0.9.52
PyGetWindow        0.0.9
pyglet             1.5.0
PyMsgBox           1.0.9
pyperclip          1.8.1
PyRect             0.1.4
PyScreeze          0.1.26
PyTweening         1.0.3
scikit-learn       0.24.0
scipy              1.6.0
setuptools         49.2.1
sklearn            0.0
threadpoolctl      2.1.0
torch              1.7.1
typing-extensions  3.7.4.3
```

Pytorch

- pytorch에서는 list대신 tensor라는 자료형을 사용합니다. Tensor는 GPU 계산에 특화되었다는 특징이 있습니다. (본 강의에서는 GPU 사용법을 생략하였습니다.)
- 먼저 간단하게 pytorch로 1차원 tensor를 만들어 봅시다. dim, shape 등을 이용해 tensor의 정보를 얻을 수 있습니다.

```
1  import torch
2
3  t = torch.tensor([1, 2, 3, 4])
4
5  print(t.dim())
6  print(t.shape)
7  print(t.size())
```

```
1
torch.Size([4])
torch.Size([4])

***Repl Closed***
```

- 2차원 tensor도 유사합니다.

```
import torch

t = torch.tensor([[1, 2], [3, 4]])

print(t.dim())
print(t.shape)
print(t.size())
```

```
2
torch.Size([2, 2])
torch.Size([2, 2])
```

Pytorch

- tensor를 조작하는 함수는 너무 많은 종류가 존재하므로, 사용할 때마다 사용법을 알려드리겠습니다.
- 먼저 layer를 만들어봅시다. tensor.nn.Linear(a, b) 함수로 input이 a개, output이 b개인 layer를 만들 수 있습니다.
- 이 layer는 [[x1, x2, .. xa], [x1, x2, .. xa], …]]인 tensor를 input으로 받아 [[y1, y2, … ya], [y1, y2, … ya]]인 tensor를 output으로 출력합니다. 이 때 input은 float이어야 하므로 FloatTensor 함수로 tensor를 만들어줍니다.
- weight, bias 함수를 통해 weight과 bias 값을 알 수 있습니다.

```
import torch

layer = torch.nn.Linear(1, 1)
print(layer.weight)
print(layer.bias)

x = torch.FloatTensor([[1], [2], [3]])
y = layer(x)
print(y)
```

```
Parameter containing:
tensor([[-0.8241]], requires_grad=True)
Parameter containing:
tensor([0.2145], requires_grad=True)
tensor([[-0.6096],
        [-1.4338],
        [-2.2579]], grad_fn=<AddmmBackward>)
```

Pytorch

- 이제 optimizer를 사용해봅시다. optimizer는 parameter와 learning rate를 입력받아 optimizer를 생성합니다.
- gradient를 계산하고, gradient를 parameter에 적용하는 과정은 총 3개의 함수로 이뤄집니다.
  먼저 zero_grad 함수를 이용해 gradient 값을 초기화합니다.
  그 후 backward 함수를 이용해 gradient 값을 구합니다.
  마지막으로 step 함수를 이용해 gradient 값으로 parameter를 update합니다.
- 이를 이전의 예제를 다시 풀어보면서 어떻게 사용되는지 알아봅시다.
- 마찬가지로, (x, y) = (2, 4), (4, 7), (8, 8)의 데이터가 주어지고, cost를 최소화하는 y = ax + b를 구합니다.

Pytorch

- 먼저 data를 정의합니다.

```
1   import torch
2
3   x_train = torch.FloatTensor([[2], [4], [8]])
4   y_train = torch.FloatTensor([[4], [7], [8]])
```

- layer를 만들어 y = wx + b꼴의 함수를 정의합니다.

```
6   layer = torch.nn.Linear(1, 1)
```

- layer의 parameter과 learning rate를 optimizer에 전달하여 optimizer를 생성합니다. 이 때, optimizer는 SGD를 사용하고, learning rate는 0.001로 설정합니다. 이 때, layer의 parameter는 parameters() 함수로 가져올 수 있습니다.

```
8   optimizer = torch.optim.SGD(layer.parameters(), lr = 0.001)
```

- 그 후, 총 100000번 train을 진행하도록 합니다. 여기서 train은 cost를 계산하고, gradient를 계산하여 optimize하는 과정을 의미합니다.

```
10  for i in range(100000):
```

Pytorch

- 이제 train을 진행해봅시다. 먼저 cost를 계산하기 전에 tensor간의 사칙연산은 어떻게 하는 지 알아봅시다. 왼쪽처럼 사칙연산을 하면, 각 원소에 대해 계산이 진행됩니다.

```
print(x_train * 2)
```

```
tensor([[ 4.],
        [ 8.],
        [14.]])
```

- 물론 tensor간의 사칙연산도 잘 계산됩니다.

```
print(x_train + y_train)
```

```
tensor([[ 6.],
        [11.],
        [15.]])
```

- 따라서 cost는 아래와 같이 구할 수 있습니다.

```
cost = (layer(x_train) - y_train) ** 2
print(cost)
```

```
tensor([[11.6339],
        [28.9479],
        [23.3727]], grad_fn=<PowBackward0>)
```

Pytorch

- tensor의 sum 함수를 이용하면 tensor의 원소를 모두 더해줍니다.

```
cost = (layer(x_train) - y_train) ** 2
print(cost)
cost = cost.sum()
print(cost)
```

```
tensor([[11.6339],
        [28.9479],
        [23.3727]], grad_fn=<PowBackward0>)
tensor(63.9545, grad_fn=<SumBackward0>)
```

- 물론 한 줄로 구할 수도 있습니다.

```
cost = ((layer(x_train) - y_train) ** 2).sum()
print(cost)
```

```
tensor(63.9545, grad_fn=<SumBackward0>)
```

- 전에 배운 함수를 아래처럼 사용하면 train이 진행됩니다. zero_grad 함수로 gradient를 초기화하고, backward 함수로 cost의 gradient를 계산한 뒤, step으로 gradient를 parameter에 적용합니다.

```
11      cost = ((layer(x_train) - y_train) ** 2).sum()
12      optimizer.zero_grad()
13      cost.backward()
14      optimzer.step()
```

Pytorch

- 따라서 cost를 최소화하는 함수 y = wx + b를 학습하는 코드는 오른쪽과 같습니다.
- 학습 결과의 weight와 bias를 출력해보면,
  이전 예제의 결과와 거의 일치함을 알 수 있습니다.

```python
import torch

x_train = torch.FloatTensor([[2], [4], [8]])
y_train = torch.FloatTensor([[4], [7], [8]])

layer = torch.nn.Linear(1, 1)

optimizer = torch.optim.SGD(layer.parameters(), lr = 0.001)

for i in range(10000):
    cost = ((layer(x_train) - y_train) ** 2).sum()
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

print(layer.weight)
print(layer.bias)
```

```
Parameter containing:
tensor([[0.6072]], requires_grad=True)
Parameter containing:
tensor([3.4999], requires_grad=True)
```

연습문제1

- 지금까지 배운 내용을 사용해서 x = [-100, 100] 범위에서의 y = x^2 함수를 nn으로 생성해 봅시다. 이 연습문제를 통해 linear함수의 조합으로 non_linear 함수를 근사하여 만들 수 있음을 알아보겠습니다. 또한 model 학습 결과를 그래프로 출력해봅니다.
- 여기서는 여러 개의 layer로 nn를 구성할 것입니다. torch에서는 Sequential 함수를 이용해 여러 개의 layer를 연속적으로 구성할 수 있습니다. 여기서는 3개의 layer과 activation function으로 ReLU를 가진 nn를 정의합니다.
- 그 후 optimizer를 정의합니다. 여기서는 Adam optimizer를 사용해보겠습니다.

```
1   import torch
2   import random
3
4   model = torch.nn.Sequential(
5           torch.nn.Linear(1, 256),
6           torch.nn.ReLU(),
7           torch.nn.Linear(256, 256),
8           torch.nn.ReLU(),
9           torch.nn.Linear(256, 1)
10      )
11
12  optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
```

연습문제1

- random 함수를 이용해 -100 이하 100 이상의 랜덤한 수를 x에 저장하고, 이를 data set에 추가합니다. 한 번의 train마다 100개의 train set을 준비합니다.

```
14  for i in range(10000):
15      x_train = []
16      y_train = []
17      for i in range(100):
18          x = random.random() * 200 - 100
19          x_train.append([x])
20          y_train.append([x**2])
```

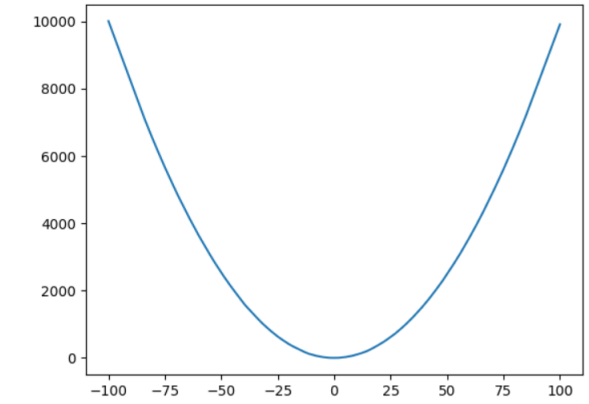
- 그 후, data set을 list에서 tensor로 바꾸고 train을 진행합니다.

```
22      x_train = torch.FloatTensor(x_train)
23      y_train = torch.FloatTensor(y_train)
24      cost = ((model(x_train) - y_train) ** 2).sum()
25      optimizer.zero_grad()
26      cost.backward()
27      optimizer.step()
```

연습문제1

- 그래프는 matplotlib 모듈로 출력할 수 있습니다. pip install matplotlib로 설치하시길 바랍니다.
- plt.plot(x, y)로 그래프를 그릴 수 있습니다. 예를 들어, x = [1, 2, 3], y = [3, 5, 7]이면, y = 2x + 1의 그래프가 x = [1, 3] 범위에 그려집니다.
- 그래프를 출력하기 위해서는 plt.show() 함수를 마지막에 사용해야 합니다.
- 그래프를 직접 출력해보면 model이 잘 학습되어 y=x^2과 거의 똑같은 함수가 만들어졌음을 알 수 있습니다.

```
30  from matplotlib import pyplot as plt
31
32  x = []
33  y = []
34  for i in range(-100, 101):
35      x.append(i)
36      y.append(model(torch.FloatTensor([[i]])).item())
37
38  plt.plot(x, y)
39  plt.show()
```



연습문제1

```
36          y.append(model(torch.FloatTensor([[i]])).item())
```

- 위 문장은 이해하기 어려워 보입니다. 하나하나 해석해 봅시다.
- torch.FloatTensor([[i]])로 i라는 원소 하나를 담은 2차원 tensor를 만듭니다.
- 이 tensor를 model에 넣어 결과를 구합니다. 결과는 [[y]]꼴의 tensor로 반환됩니다.
- item()함수는 tensor에 하나의 원소만 존재한다면, 해당 원소를 반환하는 함수입니다. 따라서 .item()함수로, [[y]] tensor의 유일한 원소 y가 반환됩니다. 즉 model(torch.FloatTensor([[i]]).item()은 숫자 i를 float tensor로 변환해 model에 전달할 수 있도록 하고, model에서 반환한 결과를 다시 숫자로 변환한 하는 과정입니다.
- 그 결과를 y list에 추가합니다.

연습문제2

- 이번에는 sin(x), 함수를 x = [-10, 10] 범위에서 설계해봅니다. 이번에는 class를 이용해서 model을 정의하는 법을 배웁니다. 또한, 이번에는 모델의 parameter를 파일로 저장하고 불러오는 방법도 배워봅니다.
- 그냥 model를 만들 때보다, class를 이용해서 model을 만들 때 더 구체적으로 model을 정의할 수 있습니다. 따라서 이번 이후에는 class를 이용하여 model을 만들게 됩니다.

연습문제2

- class로 모델을 만들 때는 torch.nn.Module을 상속받아 만들게 됩니다.
- 모델 class에서는 __init__와 forward 두 개의 메서드(함수)를 정의해야 합니다.
- __init__은 nn 모델을 정의해야 합니다. __init__의 맨 앞에 super().__init__()는 반드시 실행해야 하니, 그냥 받아들이면 됩니다.
- class로 모델을 정의하면 model(x)를 정의해야 합니다. model = nn.Sequential() 등으로 정의한 경우, model(x)를 하면, 자동으로 input x를 받아 output y를 반환했지만, class로 model을 만든 경우, model(x)를 직접 정의해야 합니다. 이는 forward(self, x)에서 정의할 수 있습니다. 즉 model(x) == model.forward(x)입니다.

```python
import torch

class sin_model(torch.nn.Module):

    def __init__(self):

        super().__init__()
        self.layer = torch.nn.Sequential(
                torch.nn.Linear(1, 256),
                torch.nn.ReLU(),
                torch.nn.Linear(256, 256),
                torch.nn.ReLU(),
                torch.nn.Linear(256, 1)
            )

    def forward(self, x):
        y = self.layer(x)
        return y
```

연습문제2

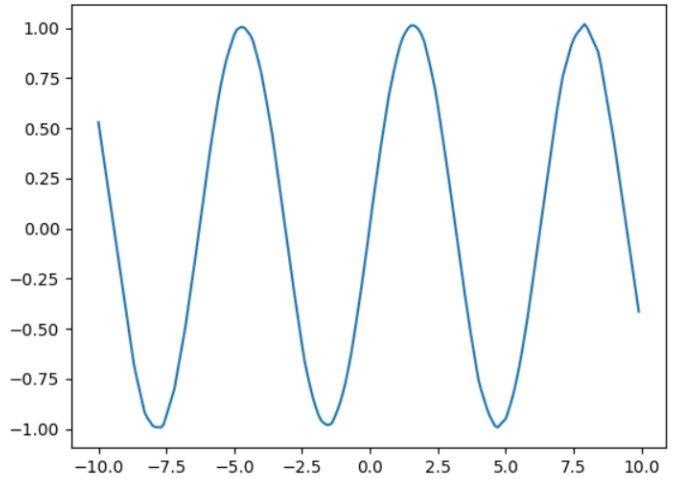
- 이제 이전과 똑같은 방법으로 학습을 진행하면 됩니다.
- sin 함수는 math 모듈에서 가져와 사용할 수 있습니다.

```
20  from math import sin
21  from random import random
22
23  model = sin_model()
24  optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
25
26  for i in range(10000):
27
28      x_train = []
29      y_train = []
30      for i in range(100):
31          x = random() * 20 - 10
32          x_train.append([x])
33          y_train.append([sin(x)])
34
35      x_train = torch.FloatTensor(x_train)
36      y_train = torch.FloatTensor(y_train)
37
38      cost = ((model(x_train) - y_train) ** 2).sum()
39      optimizer.zero_grad()
40      cost.backward()
41      optimizer.step()
```

연습문제2

- 같은 방식으로 그래프를 출력해보면, sin 그래프와 유사한 개형이 잘 나오는 것을 발견할 수 있습니다.

```
43  from matplotlib import pyplot as plt
44
45  x = []
46  y = []
47  for i in range(-100, 100):
48      x.append([i / 10])
49      y.append(model(torch.FloatTensor([[i / 10]])).item())
50  plt.plot(x, y)
51  plt.show()
52
53
```



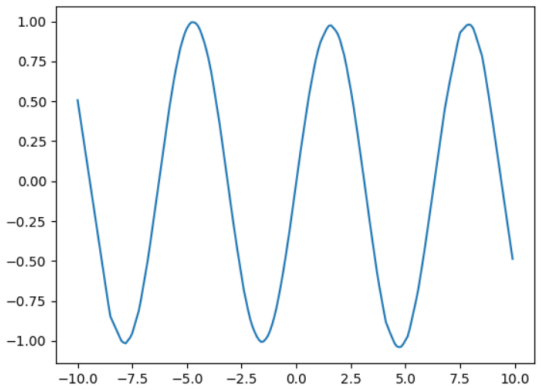
연습문제2

- 한 번 모델을 저장해 볼까요? 모델 저장은 torch.save(model.state_dict(), 경로) 로 저장하고, model.load_state_dict(torch.load(경로))로 불러올 수 있습니다. 확장자는 .pt를 주로 사용합니다.
- 아래처럼 model을 저장할 수 있습니다.

```
54    torch.save(model.state_dict(), "model.pt")
```

- 그 후 중간에 load하는 코드를 추가하고, 학습 과정을 생략해도 sin 그래프가 나오는 것을 알 수 있습니다.

```
23  model = sin_model()
24  model.load_state_dict(torch.load("model.pt"))
25
26  from matplotlib import pyplot as plt
27
28  x = []
29  y = []
30  for i in range(-100, 100):
31      x.append([i / 10])
32      y.append(model(torch.FloatTensor([[i / 10]])).item())
33  plt.plot(x, y)
34  plt.show()
35
36  torch.save(model.state_dict(), "model.pt")
```



연습문제3

- 마지막으로 max(x1,x2), min(x1, x2) 함수를 한 모델에 설계해봅시다. 이 때 모델은 3개의 큰 layer로 설계할 예정입니다.
- 첫 번째 layer은 2개의 input x1, x2을 받아 256개의 output을 반환합니다.
- 두 번째 layer는 256개의 input을 받아 max(x1, x2)를 반환합니다.
- 세 번째 layer는 256개의 input을 받아 min(x1, x2)를 반환합니다.
- x1, x2는 둘 다 0 이상 100 이하의 실수입니다.

연습문제3

- 먼저 3개의 layer를 적당히 만듭니다.
- self.layer는 2개의 input을 받고, 256개의 output을 반환합니다.
- self.max_layer는 256개의 input을 받고,
  max(x1, x2)값을 출력합니다.
- self.min_layer는 256개의 input을 받고,
  min(x1, x2)값을 출력합니다.

```python
import torch

class minmax_model(torch.nn.Module):

    def __init__(self):

        super().__init__()
        self.layer = torch.nn.Sequential(
                torch.nn.Linear(2, 256),
                torch.nn.ReLU(),
                torch.nn.Linear(256, 256),
                torch.nn.ReLU(),
                torch.nn.Linear(256, 256),
                torch.nn.ReLU()
            )

        self.max_layer = torch.nn.Sequential(
                torch.nn.Linear(256, 256),
                torch.nn.ReLU(),
                torch.nn.Linear(256, 1)
            )

        self.min_layer = torch.nn.Sequential(
                torch.nn.Linear(256, 256),
                torch.nn.ReLU(),
                torch.nn.Linear(256, 1)
            )
```

연습문제3

- forward 함수는 오른쪽처럼, layer를 거친 결과를 min_layer, max_layer에 전달하고, 두 layer에서의 결과를 반환하는 방식으로 함수를 정의할 수 있습니다.
- 이제 이후에는 똑같은 방식으로 train을 진행하면 됩니다.
- 단 이 때는 model에서 min과 max를 반환하므로, min에 대한 cost와 max에 대한 cost를 따로 계산 한 후 두 cost를 더하는 방식으로 계산해야 합니다.
- 이처럼 연습문제2와 유사한 방식으로 모델을 학습시킬 수 있습니다.

```
29      def forward(self, x):
30          out = self.layer(x)
31          Max = self.max_layer(out)
32          Min = self.min_layer(out)
33          return Max, Min
```

```
35  from random import random
36
37  model = minmax_model()
38  optimzer = torch.optim.Adam(model.parameters(), lr = 0.001)
39
40  for i in range(10000):
41
42      x_train = []
43      max_train = []
44      min_train = []
45
46      for i in range(100):
47          x1 = random() * 100
48          x2 = random() * 100
49          x_train.append([x1, x2])
50          max_train.append([max([x1, x2])])
51          min_train.append([min([x1, x2])])
52
53      x_train = torch.FloatTensor(x_train)
54      max_train = torch.FloatTensor(min_train)
55      min_train = torch.FloatTensor(max_train)
56      Max, Min = model(x_train)
57
58      cost = ((max_train - Max) ** 2).sum() + ((min_train - Min) ** 2).sum()
59
60      optimzer.zero_grad()
61      cost.backward()
62      optimzer.step()
```