퍼셉트론 (Perceptron) & MLP (Multi-Layer Perceptron)

퍼셉트론이란?

- Perceive (지각하다, 감지하다. 알아채다….) + -tron(장치, 기계, 도구…)
  - ➔ 지각하는 장치, 감지하는 도구 …
  - ➔ 뉴런을 단순화한 모델

- 1957년에 로젠블라트(Frank Rosenblatt)가 고안

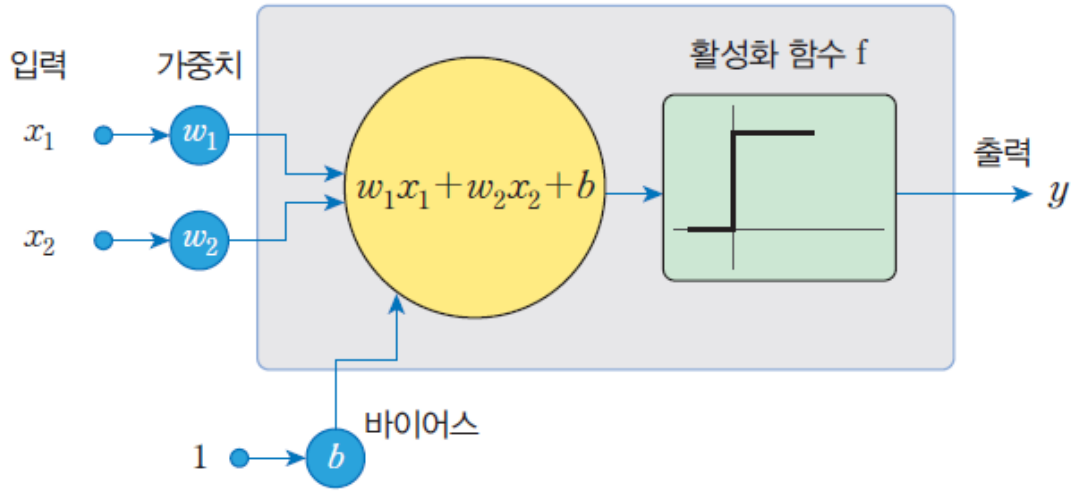
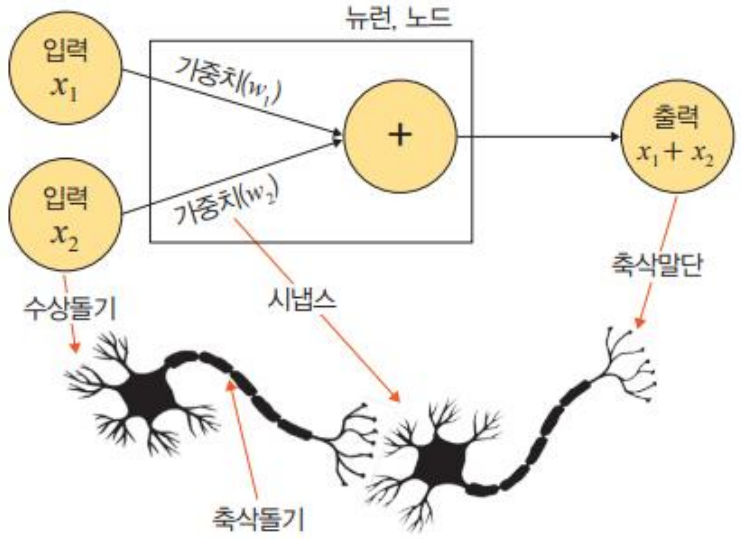


그림 5-5 퍼셉트론에서의 뉴런의 모델

퍼셉트론이란?

- 수상돌기: 주변이나 다른 뉴런에서 자극을 받아들이고, 이 자극들을 전기적 신호 형태로 세포체와 축삭돌기로 보내는 역할
- 축삭돌기: 다른 뉴런(수상돌기)에 신호를 전달하는 기능을 하는 뉴런의 한 부분
- 축삭말단: 전달된 전기 신호를 받아 신경 전달 물질을 시냅스 틈새로 방출
- 시냅스: 신경 세포들이 이루는 연결 부위로, 한 뉴런의 축삭돌기와 다음 뉴런의 수상돌기가 만나는 부분



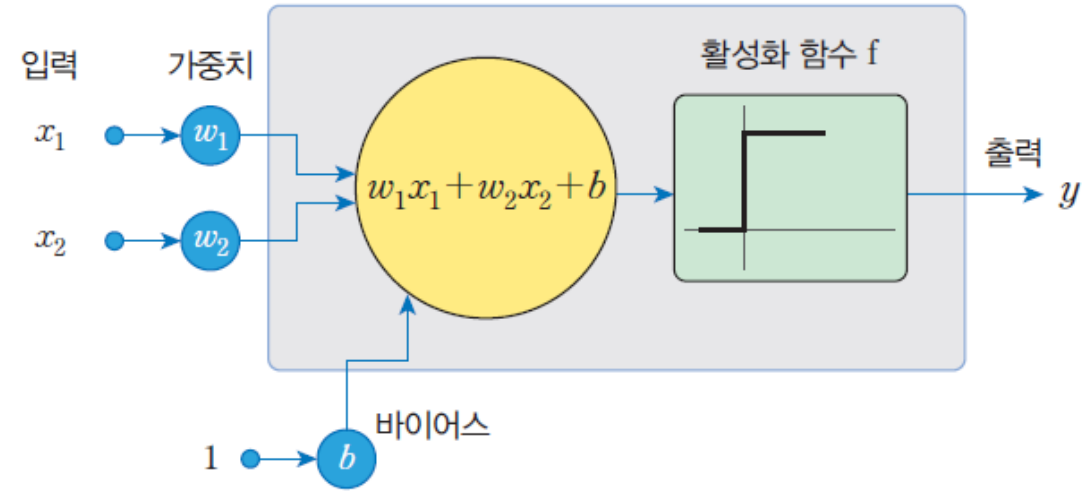
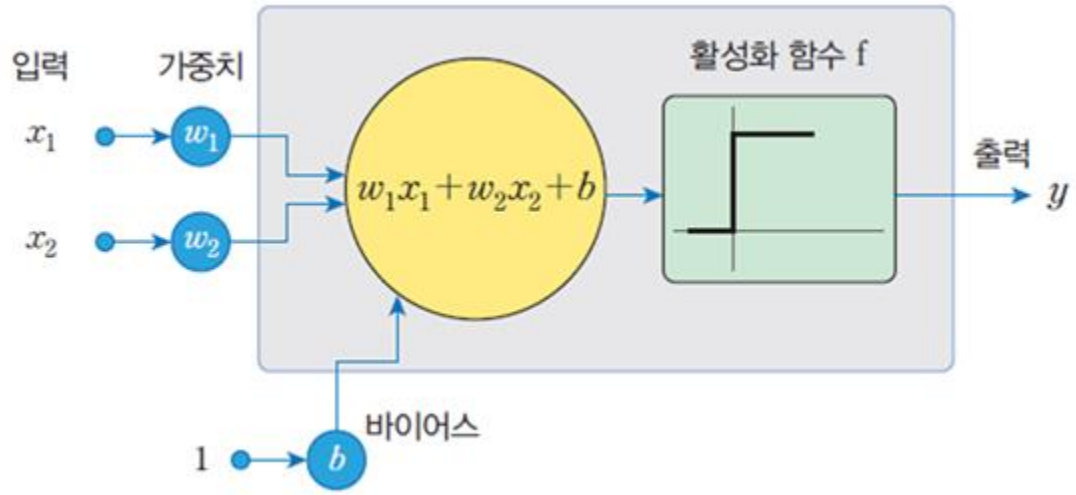


그림 5-5 퍼셉트론에서의 뉴런의 모델

실습 – AND 학습하기



입력 및 출력:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

활성화 함수: sigmoid

$$\frac{1}{1+e^{-x}}$$

학습:

w_1, w_2, b을 찾아라

w/o PyTorch

```
# 필요한 패키지, 모듈 읽어오기

# 활성화 함수 및 미분

# 학습에 필요한 데이터셋

# 가중치 초기화 (랜덤)

# 학습 루프
  # 순전파
  # 손실 계산
  # 역전파
  # 가중치 업데이트
  # 학습 확인 (로그 출력)

# 최종 예측 확인: 추론
```

w/o PyTorch

```
# 필요한 패키지, 모듈 읽어오기
import math
import random

# 활성화 함수 및 미분
def sigmoid(x):
    return 1 / (1 + math.exp(-x))

def sigmoid_deriv(x):
    return -x * (1 - x)

# 학습에 필요한 데이터셋
X = [[0,0], [0,1], [1,0], [1,1]]
y = [0, 0, 0, 1]

# 가중치 초기화 (랜덤)
random.seed(42)
W = [random.uniform(-1, 1), random.uniform(-1, 1)]  # random.uniform(-1, 1): -1에서 1사이의 실수를 uniform random 하게
b = random.uniform(-1, 1)
```

w/o PyTorch

```
# 학습 루프
for epoch in range(1000):
    total_loss = 0
    for i in range(len(X)):
        # Forward
        z = X[i][0]*W[0] + X[i][1]*W[1] + b
        a = sigmoid(z)

        # 손실 계산 (0.5 * MSE)
        loss = 0.5 * (y[i] - a) ** 2
        total_loss += loss

        # Backpropagation (출력층만 있으므로 단순)
        error = (y[i] - a) * sigmoid_deriv(a)

        # 가중치 업데이트
        W[0] -= 0.1 * error * X[i][0]
        W[1] -= 0.1 * error * X[i][1]
        b -= 0.1 * error

    # 로그 출력
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {total_loss/4:.4f}")
```

w/o PyTorch

```
# 최종 예측 확인
print("\nFinal Predictions:")
for i in range(len(X)):
    z = X[i][0]*W[0] + X[i][1]*W[1] + b
    a = sigmoid(z)
    out = 1 if a >= 0.5 else 0
    print(f"Input: {X[i]}, Predicted: {out}, Target: {y[i]}")
```

w/ PyTorch

```
# 0: 필요한 패키지, 모듈 읽어오기

# 1: 데이터 셋

# 2: 모델 정의

# 3: 손실함수 & 옵티마이저

# 4: 학습 루프
   # 순전파 gradient 초기화
   # 순전파 (forward propagation)
   # 손실 계산
   # 역전파 (backward propagation, gradient 계산)
   # 파라미터 업데이트
   # validation확인

# 5: 평가: 추론
```

w/ PyTorch

```
# 0: 필요한 패키지, 모듈 읽어오기
import torch
import torch.nn as nn
import torch.optim as optim

# 1: 데이터 셋 (입력과 정답)
X = torch.tensor([[0.,0.], [0.,1.], [1.,0.], [1.,1.]])

Y = torch.tensor([[0.], [0.], [0.], [1.]])
```

w/ PyTorch

```
# 2: 모델 정의
class Perceptron(nn.Module):
    def __init__(self):
        super(Perceptron, self).__init__()
        self.layer = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return self.sigmoid(self.layer(x))

model = Perceptron()
```

w/ PyTorch

```
# 3: 손실함수 & 옵티마이저
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

w/ PyTorch

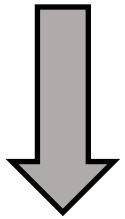
```
# 4: 학습
model.train()
epochs = 1000
for epoch in range(epochs):
    optimizer.zero_grad()                    # 순전파 gradient 초기화
    outputs = model(X)                       # 순전파 (forward propagation)
    loss = criterion(outputs, Y)             # 손실 계산
    loss.backward()                          # 역전파 (backward propagation, gradient 계산)
    optimizer.step()                         # 파라미터 업데이트

    if epoch % 100 == 0:                     # validation확인
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

w/ PyTorch

```
# 5: 최종 예측 확인
print("\nFinal Predictions:")
model.eval()
with torch.no_grad():
    preds = model(X)
    out = torch.where(preds >= 0.5, 1, 0)
    for i in range(len(X)):
        print(f"Input: {X[i].tolist()}, Predicted: {out[i].item():.3f}, Target: {Y[i].item()}")
```

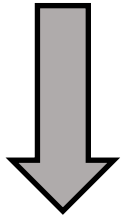
활성화 함수(Activation Function)

각 뉴런은 작은 자극에 대해서는 반응하지 않는다.
선형의 조합으로는 선형밖에 표현할 수 없다.



뉴런과 같이 특정 자극 이상에서 반응하고,
비선형성을 도입하자!

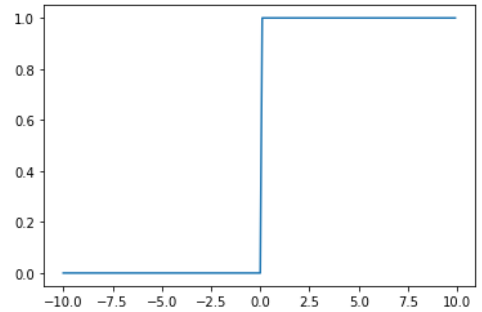
Step Function



미분이 안되면 학습을 어떻게 하지?

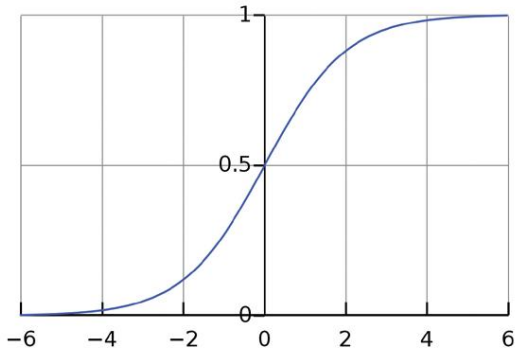
미분 가능한 다양한 활성화 함수 등장

활성화 함수(Activation Function)

- Step Function
  - $$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$
  - 

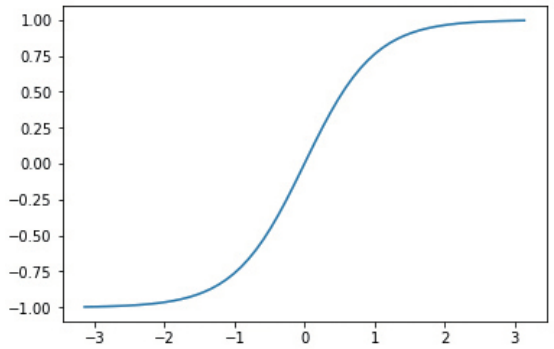
  - 생물학적 뉴런 모사 (역치, Threshold)
  - 미분 불가능
  - PyTorch 에서 지원 X

활성화 함수(Activation Function)

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

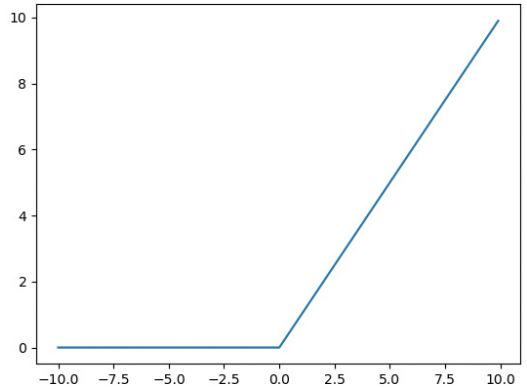
- Step Func.과 유사
- 미분 가능 => PyTorch 지원 (nn.Sigmoid())
- 기울기 소실 (Gradient Vanishing)
- 결과 값이 (0, 1) => 학습에 비효율적

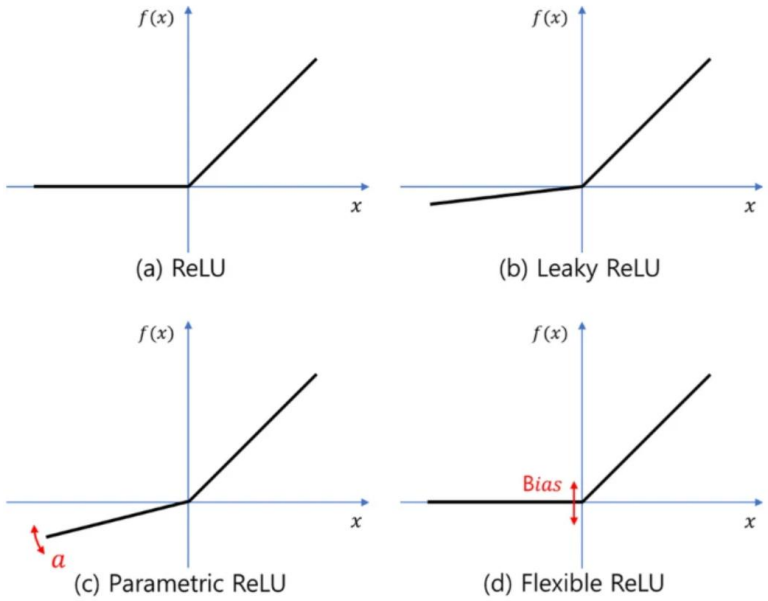
활성화 함수(Activation Function)

- Tanh
  - $$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1$$
  - 

- (-1, 1)범위, 0을 중심으로 출력
- 미분 가능 => PyTorch 지원 (nn.Tanh())
- 기울기 소실 (Gradient Vanishing)
- 계산이 복잡함

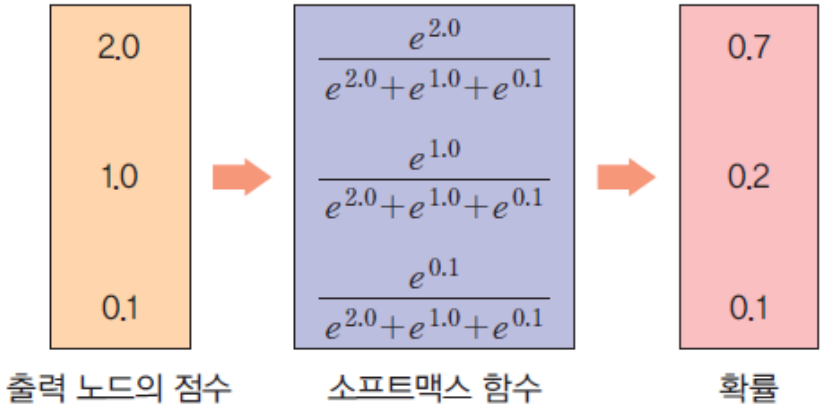
활성화 함수(Activation Function)

- ReLU (Rectified Linear Unit Function)
  - $$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$
  - 



- 계산 속도 빠름, 기울기 소실 문제 해결
- 희소성 촉진 (0인 출력 생성) => 복잡도, 효율성 증대
- 부분적 미분 가능 => PyTorch 지원 (nn.ReLU())
- 죽은 ReLU (음수의 경우 정보 손실) ➔ LeakyReLU (nn.LeakyReLU()), ParametricReLU (nn.PReLU())
- 출력이 모두 양수 => Batch Normalization (nn.BatchNorm1d(num_features=10))

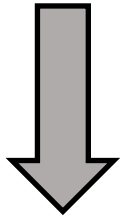
활성화 함수(Activation Function)

- Softmax
  - $$p_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad for\ i = 1, 2, \ldots k$$
  - 

- (0, 1): 출력 값을 양수화, 정규화, 확률화
- 가장 큰 값을 강조: 지수 함수
- 다중 클래스 분류 문제의 출력층(마지막 층)으로 주로 사용
- PyTorch에서 지원: nn.Softmax(dim=1)

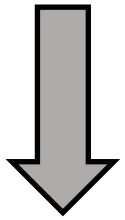
손실 함수(Loss Function)

컴퓨터는 알잘딱센을 못하는데?



수식으로 목표를 정해서 주자!

목적 함수 or 손실 함수 (MSE 기반)



세상에는 AI가 해야 하는 임무가 다양한데?

목적에 맞는 다양한 손실 함수 등장
But, 미분 가능해야 학습 가능함

손실 함수(Loss Function)

- Mean Sequared Error(MSE) Loss
  - $L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
  - 회귀 문제(연속적인 값 예측)에 주로 사용
  - 예측값과 정답의 차이를 줄이는 방향으로 학습

  - PyTorch 구현
    - 함수: nn.MSELoss()
    - 사용법
      - 예측값과 정답을 비교
    - 예시: criterion = nn.MSELoss()
      criterion(predicted, target)

손실 함수(Loss Function)

- Binary Cross Entropy Loss
  - $L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$
  - 두 개의 클래스(레이블) 중 하나를 예측하는 문제에서 사용
  - PyTorch 구현
    - 함수: nn.BCELoss()
    - 사용법
      - 예측값: 활성화 함수를 적용한 tensor의 값(클래스 1에 속할 확률)
      - 비교 정답: 정답 레이블
    - 예시: criterion = nn.BCELoss()
      criterion(predicted, label)

손실 함수(Loss Function)

- Binary Cross Entropy Loss (Logit 값을 그대로 받는 경우)

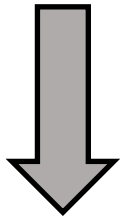
```
# 모델 정의
class Perceptron(nn.Module):
    def __init__(self):
        super(Perceptron, self).__init__()
        self.layer = nn.Linear(2, 1)

    def forward(self, x):
        return self.layer(x)

# 손실함수 & 옵티마이저
criterion = nn.BCEWithLogitsLoss() # BCEWithLogitsLoss 의 경우 sigmoid 함수를 포함
                                   # 활성화 함수 통과 전 (logit)을 그대로 넣어줘야 함
```

손실 함수(Loss Function)

- CrossEnytopyLoss
  - $H(p, q) = -\sum_{x} p(x) \log_n q(x)$
  - 다중(셋 이상) 클래스 중 하나를 예측하는 문제에서 사용
  - PyTorch 구현
    - 함수: nn.CrossEntropyLoss()
    - 기능: Softmax + cross entropy loss 계산
    - 사용법:
      - 예측값: 활성화 함수 적용(각 클래스의 확률 계산) 전 logits (클래스 개수 만큼의 값)
      - 비교 정답: 정답 레이블 (1개)

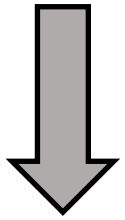
옵티마이저(Optimizer)

최종 파라미터 값에 어떻게 잘 도착할까?



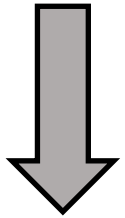
전체 학습 데이터를 다 계산하고, 평균적으로 가 볼까?

Gradient Descent



너무 느린데? 중간 중간 업데이트 하자!

Stochastic Gradient Descent



불안정한데? 크기나 변해가는 경향도 참고하자!

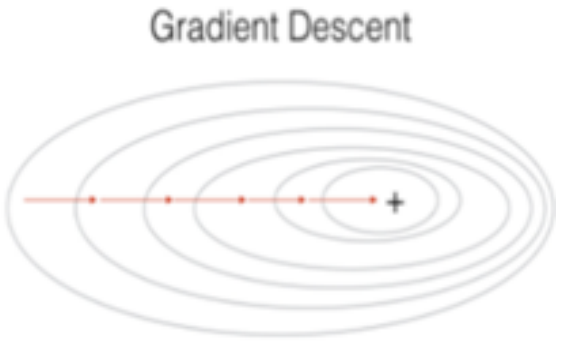
속도와 안정성을 고려하는 다양한 옵티마이저 등장

Optimizer: Stochastic Gradient Descent

경사하강법(GD)

전체 데이터의 평균으로
매개변수 업데이트

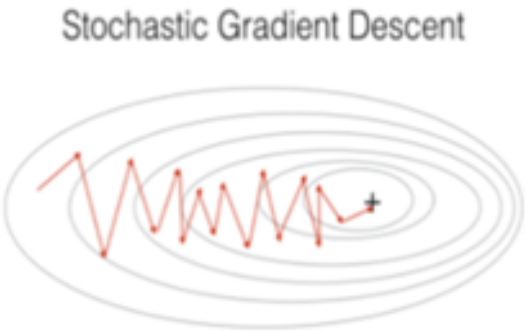
$$Loss(W,b) = \frac{1}{n}\sum_{i=1}^{n}((Wx_i+b)-y_i)^2$$



확률적 경사하강법(SGD)

무작위로 선택된 하나의 샘플로
매개변수 업데이트

$$Loss(W,b) =((Wx_i+b)-y_i)^2$$



최적의 방향으로 안정적으로 수렴

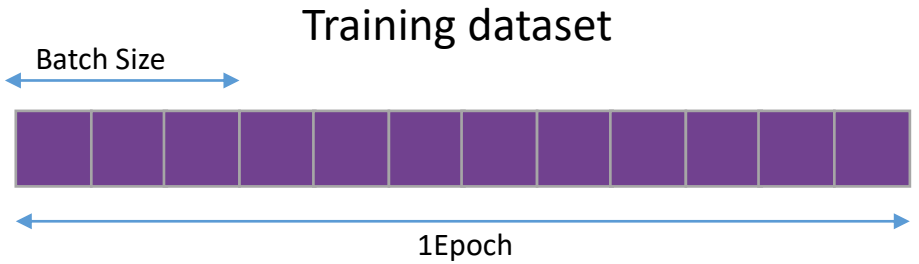
한 번의 업데이트에 오래 걸림

한 번의 업데이트에 굉장히 빠름

불규칙하게 수렴

미니 배치 경사 하강법 (Mini-Batch GD)

Optimizer: Stochastic Gradient Descent



1 Epoch = 데이터 셋의 모든 데이터를 이용하여 학습했음을 의미
1 iteration = 1회 학습 (= weights(가중치) 1회 업데이트)

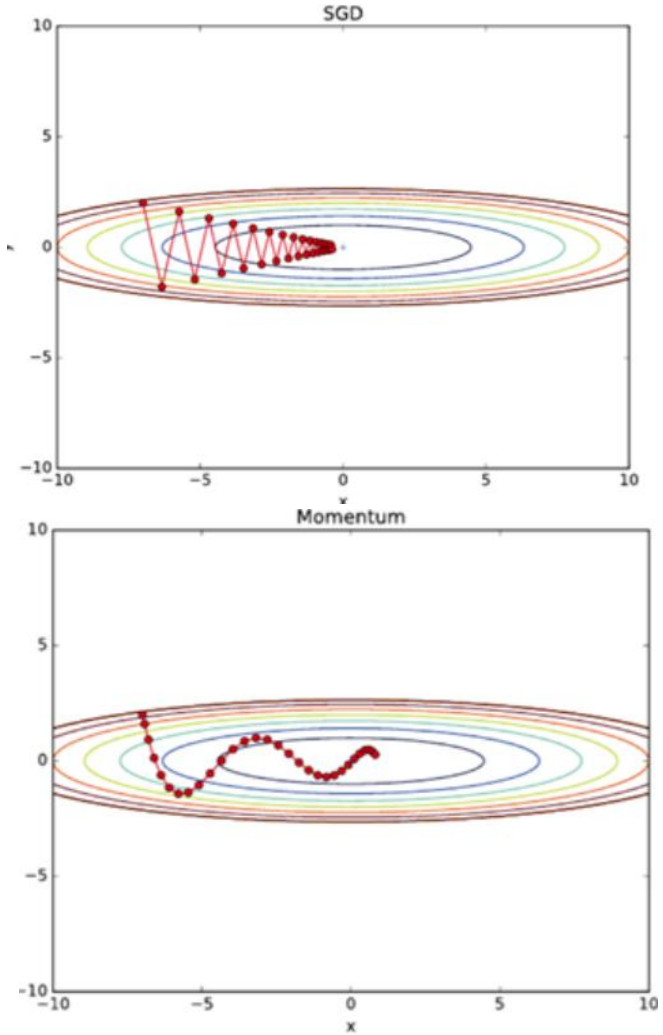
Minibatch (그냥 batch라고도 함) = 데이터 셋을 batch size 크기로 쪼개서 학습함
(쪼개지 않은 것을 full-batch라고 하기도 함)

Ex> 데이터 개수가 100, Batch size = 10 일 때
GD: 100개의 데이터로 한 번 모델 업데이트 (1 Epoch => 1 iteration)
SGD: 1개의 데이터로 한 번 모델 업데이트 (1 Epoch => 100 iterations)
Mini-batch GD: 10개의 데이터로 한 번 모델 업데이트 (1 Epoch => 10 iterations)

Mini-batch GD를 가장 많이 사용 ➔ 일반적으로 SGD로 부르기도 함

Pytorch: optim.SGD(model.parameters(), lr=0.1)로 동일 ➔ 모델 업데이트 주기에 따라 구분

Optimizer: SGD with Momentum



불규칙(불안정)하게 수렴
➔ Learning rate, batch size로 조절하는 것은 한계가 있음
➔ 너무 커지면 더 불안정
➔ 너무 작아지면 수렴의 문제

관성 (Momentum) 을 이용하자.
: 이전의 이동을(속도를) 어느 정도 유지하자!

현재 속도 이전 속도 역전파 결과

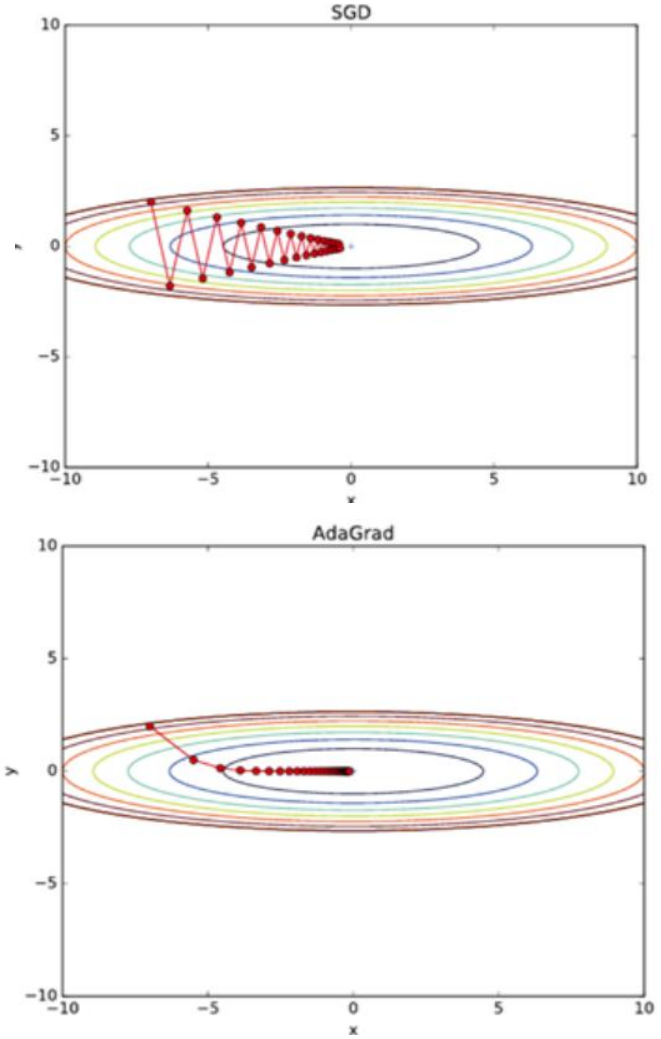
$$v_t = \beta v_{t-1} + (1-\beta)\nabla W_t$$

$$W_{t+1} = W_t - \eta v_t$$

업데이트 파라미터: 역전파 결과를 바로 반영하지 말고
이전 이동을 적당히(=β) 반영하자.

Pytorch: optim.SGD(model.parameters(), lr=0.1, momentum=0.9): momentum = β, 안 쓰면 SGD (β=0)

Optimizer: AdaGrad(Adaptive Gradient)



불규칙(불안정)하게 수렴
➔ Learning rate, batch size로 조절하는 것은 한계가 있음
➔ 너무 커지면 더 불안정
➔ 너무 작아지면 수렴의 문제

크게 변동하는 파라미터는 조금만:
업데이트되는 크기를 누적하면서, 많이 움직이는 파라미터는 적게 업데이트 하자!

역전파 결과의 제곱값

$$G_t = G_{t-1} + g_t^2$$

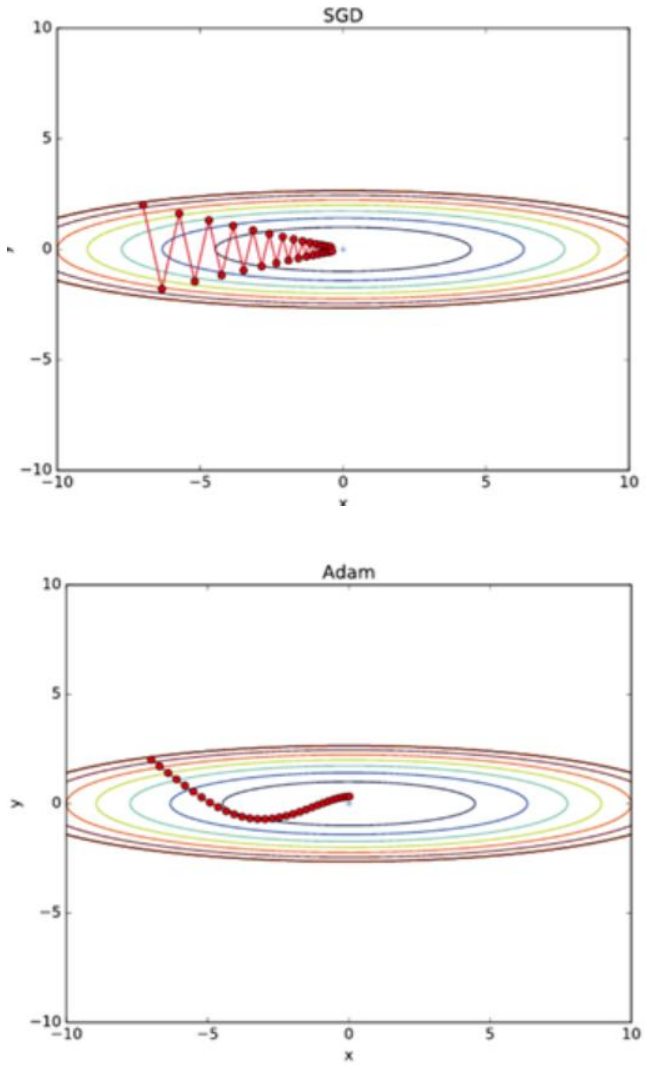
$$W_{t+1} = W_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

제곱의 합이 분모에...

업데이트 파라미터: 누적한 역전파 제곱값이 클수록 작게 업데이트

Pytorch: optim.Adagrad(model.parameters(), lr=0.01)

Optimizer: ADAM(Adaptive Moment Estimation)

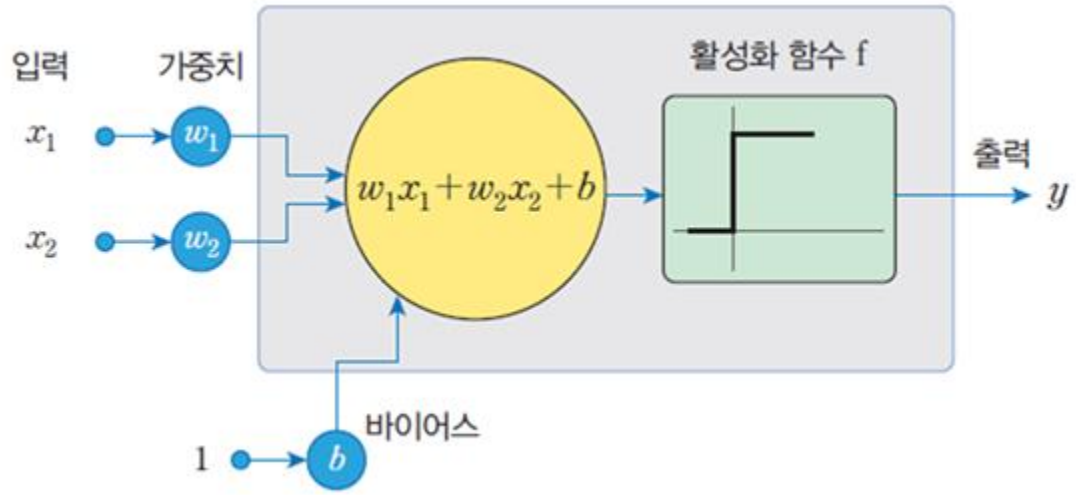


불규칙(불안정)하게 수렴

앞의 두 개를 합치면 더 좋아지겠다:
SGD with momentum + RMSProp (AdaGrad 개선)
➔ ADAM

Pytorch:
optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-8)

실습 – XOR 학습하기



입력 및 출력:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

활성화 함수: sigmoid

$$\frac{1}{1+e^{-x}}$$

학습:

w_1, w_2, b을 찾아라

실습 – XOR 학습하기

```
# 필요한 패키지, 모듈 읽어오기
import torch
import torch.nn as nn
import torch.optim as optim

# 데이터 셋 (입력과 정답)
X = torch.tensor([[0.,0.], [0.,1.], [1.,0.], [1.,1.]])

Y = torch.tensor([[0.], [1.], [1.], [0.]])
```

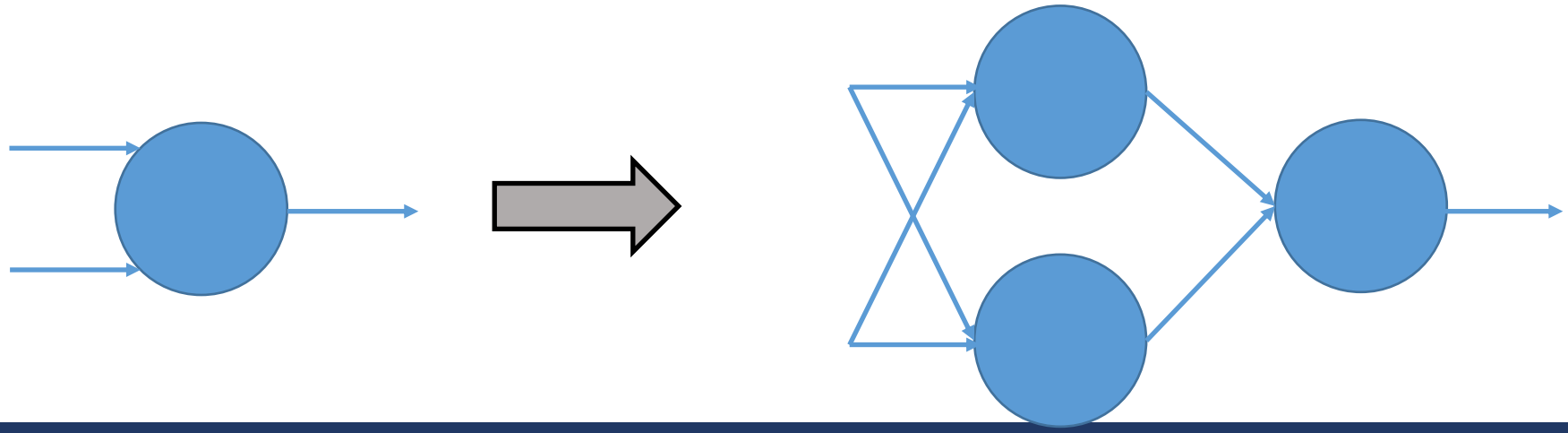
MLP(Multi-Layer Perceptron)

Perceptron 개념만으로는 XOR도 풀지 못함

1차 AI 겨울

여러 개 쌓으면 되지! But, 학습은?

Chain Rule을 이용한 Back Propagation으로!!



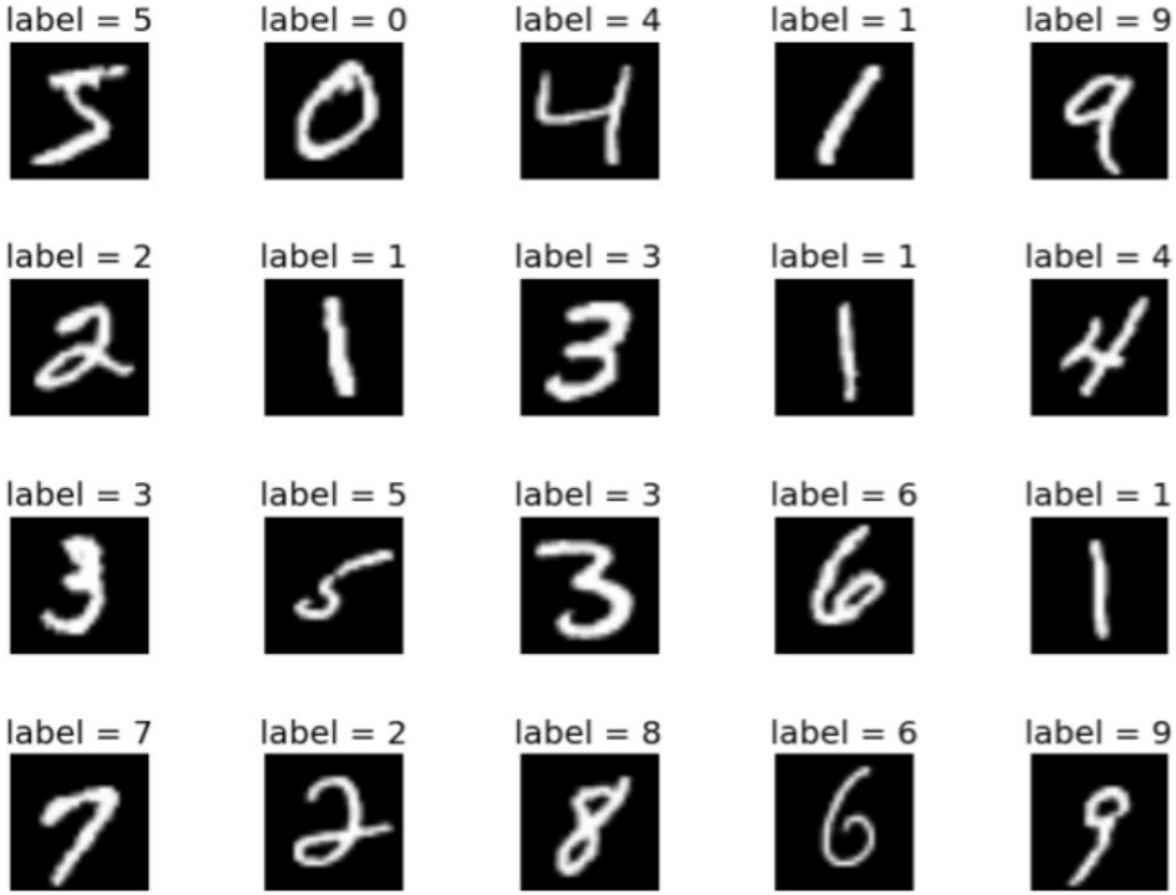
MLP: XOR

```
# 모델 정의
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layer1 = nn.Linear(2, 2)
        self.sigmoid1 = nn.Sigmoid()
        self.layer2 = nn.Linear(2, 1)
        self.sigmoid2 = nn.Sigmoid()

    def forward(self, x):
        x = self.layer1(x)
        x = self.sigmoid1(x)
        x = self.layer2(x)
        x = self. sigmoid2(x)

        return x

model = MLP()
```

Using PyTorch

```
# 0: 필요한 패키지, 모듈 읽어오기

# 1: 데이터 셋

# 2: 모델 정의

# 3: 손실함수 & 옵티마이저

# 4: 학습 루프
   # 순전파 gradient 초기화
   # 순전파 (forward propagation)
   # 손실 계산
   # 역전파 (backward propagation, gradient 계산)
   # 파라미터 업데이트
   # validation확인

# 5: 평가: 추론
```

MLP: 숫자 인식 (MNIST)



MLP: 숫자 인식 (MNIST)

```
# 0: 필요한 패키지, 모듈 읽어오기
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# 1: 데이터 셋
transform = transforms.Compose([
    transforms.ToTensor(),           # 이미지를 [0,1] 범위 tensor로
    transforms.Normalize((0.1307,), (0.3081,))  # MNIST 평균/표준편차 정규화
])

train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
test_dataset  = datasets.MNIST(root="./data", train=False, download=True, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

MLP: 숫자 인식 (MNIST)

```
# 2. 모델 정의 (MLP)
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = x.view(-1, 28*28)   # 이미지를 1차원 벡터로
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)        # CrossEntropyLoss는 softmax 자동 적용
        return x

model = MLP()

# 3. 손실 함수 & 옵티마이저
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

MLP: 숫자 인식 (MNIST)

```
# 4. 학습 루프
model.train()
epochs = 5
for epoch in range(epochs):
total_loss = 0
   for data, target in train_loader:
      optimizer.zero_grad()                    # 순전파 gradient 초기화
      output = model(data)                     # 순전파 (forward propagation)
      loss = criterion(output, target)         # 손실 계산
      loss.backward()                          # 역전파 (backward propagation, gradient 계산)
      optimizer.step()                         # 파라미터 업데이트
      total_loss += loss.item()                # loss 누적 (수정해야 함)
   print(f " Epoch {epoch+1}, Loss: {total_loss/len(train_loader):.4f} " )        # epoch별 누적된 loss 확인 (수정해야 함)
```

MLP: 숫자 인식 (MNIST)

```
# 5. 평가
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        preds = output.argmax(dim=1)              # 각 클래스 중 가장 높은 확률을 가진 클래스의 인덱스 번호 할당
        correct += (preds == target).sum().item()  # 각 샘플별로 같은 값은 True(1), 다른 값은 False(0) ➔ 모든 값 더하기(Tensor)
                                                   # 숫자로 값 얻어오기
        total += target.size(0)

print(f"\nTest Accuracy: {100.0 * correct / total:.2f}%")
```