16기 정규세션

ToBig's 15기 강의자
이성범

Neural Network 심화

Contents

Unit 01 | Intro

Unit 02 | Activation Function

Unit 03 | Weight Initialization

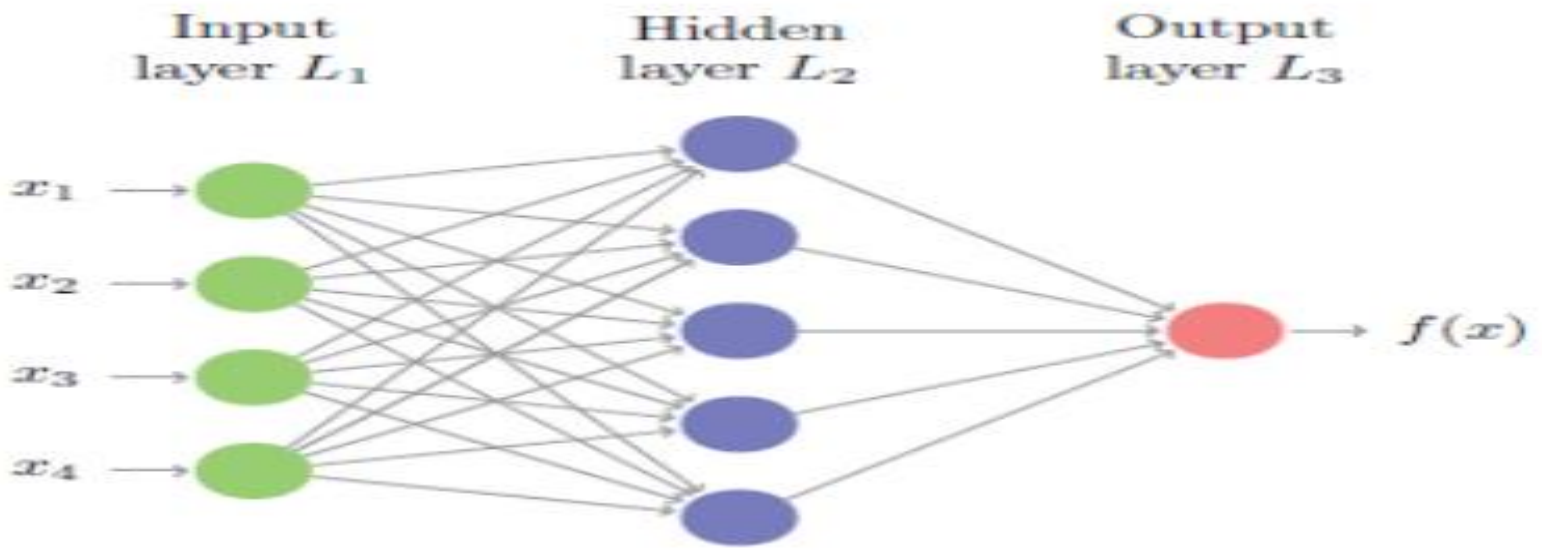
Unit 04 | Batch Normalization

Unit 05 | Optimization

Unit 06 | Dropout & Regularization

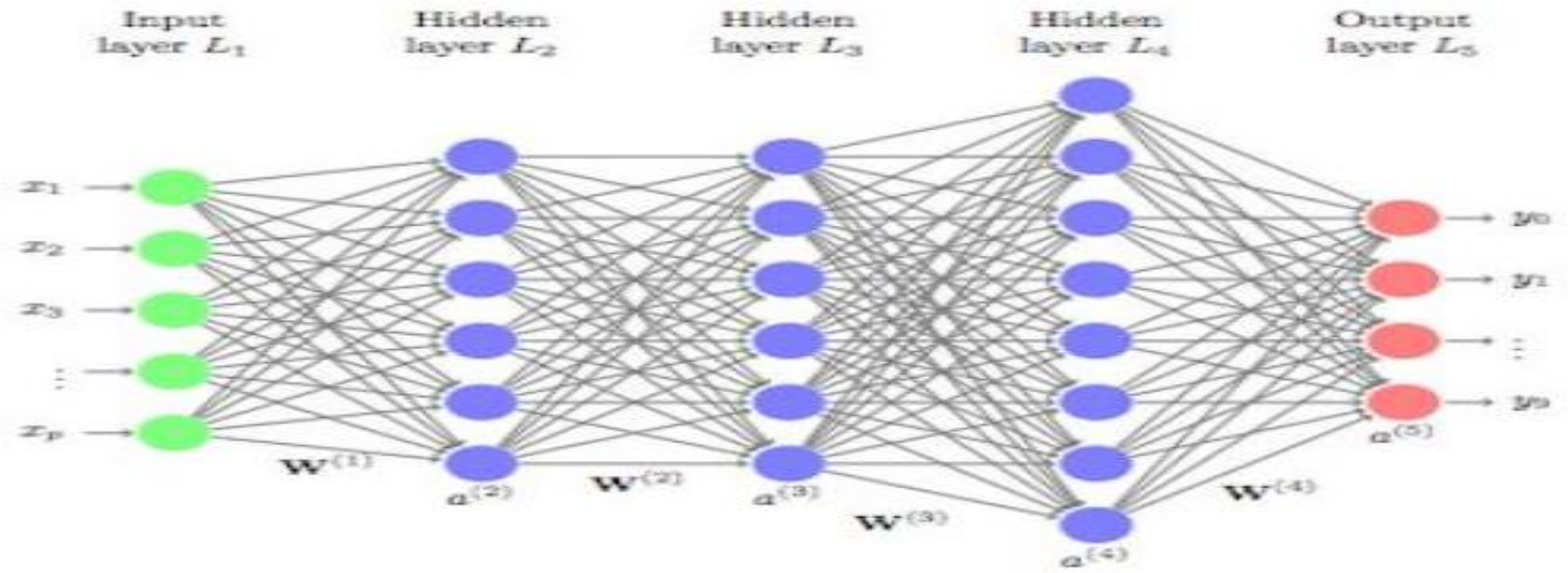
01. Intro

~얖게~



Simple Neural Net

~깊게~



Deep Neural Net

Neural Net은

Layer을 깊게 쌓을수록

성능이 올라갈 가능성이 높아진다!

하지만 아무 생각 없이 무작정 Layer을 깊게 쌓기만 한다면??

다음과 같은 문제점이 발생!

- **Underfitting** 학습이 잘 안돼!
- **Too Slow** 학습이 너무 느려!
- **Overfitting** 융통성이 없어!

따라서

효율적, 효과적으로

Layer를 쌓아야 한다!

다음과 같은 해결책이 존재!

- **Activation Function** — Sigmoid, Tanh ,ReLU ....
- **Weigh Initialization** — Xavier, He ...
- **Batch Normalization** — Internal Covariate Shift 해결
- **Optimization** — SGD, RMSprop, Adam ...
- **Dropout** — 동조현상 방지
- **Regularization** — L1, L2 ...

우리는 해결책을 배워나가며 아래 모델의 성능을 높여 나갈 것임!


```python
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.linear1 = nn.Sequential(
        nn.Linear(28 * 28, 512)
    )
    self.linear2 = nn.Sequential(
        nn.Linear(512, 256)
    )
    self.linear3 = nn.Sequential(
        nn.Linear(256, 128)
    )
    self.linear4 = nn.Sequential(
        nn.Linear(128, 10)
    )

  def forward(self, x):
    x = x.view(-1, 28 * 28)
    x = self.linear1(x)
    x = self.linear2(x)
    x = self.linear3(x)
    x = self.linear4(x)
    return x
```

단순히 Linear Layer로만 이루어진

간단한 NN

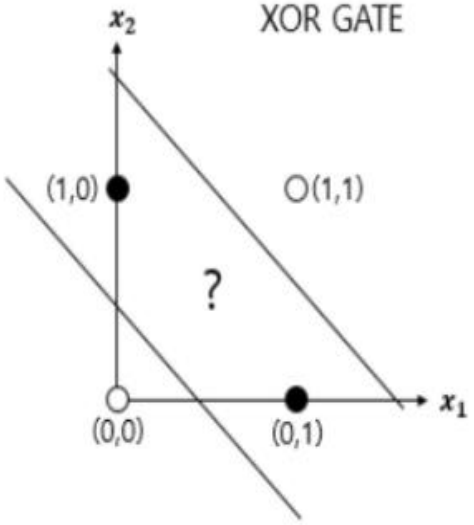
```python
model = Net().to(DEVICE)
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum = 0.5)
criterion = nn.CrossEntropyLoss()
```

```
[EPOCH: 1],     Train Loss: 1.3255,     Train Accuracy: 57.03 %,     Val Loss: 0.8147,     Val Accuracy: 69.37 %
[EPOCH: 2],     Train Loss: 0.7146,     Train Accuracy: 73.83 %,     Val Loss: 0.6427,     Val Accuracy: 77.01 %
[EPOCH: 3],     Train Loss: 0.6003,     Train Accuracy: 78.72 %,     Val Loss: 0.5542,     Val Accuracy: 80.32 %
[EPOCH: 4],     Train Loss: 0.5401,     Train Accuracy: 81.25 %,     Val Loss: 0.5189,     Val Accuracy: 81.71 %
[EPOCH: 5],     Train Loss: 0.5090,     Train Accuracy: 82.20 %,     Val Loss: 0.5009,     Val Accuracy: 82.68 %
[EPOCH: 6],     Train Loss: 0.4904,     Train Accuracy: 82.70 %,     Val Loss: 0.4808,     Val Accuracy: 82.95 %
[EPOCH: 7],     Train Loss: 0.4780,     Train Accuracy: 83.21 %,     Val Loss: 0.4940,     Val Accuracy: 82.13 %
[EPOCH: 8],     Train Loss: 0.4678,     Train Accuracy: 83.64 %,     Val Loss: 0.4552,     Val Accuracy: 83.87 %
[EPOCH: 9],     Train Loss: 0.4595,     Train Accuracy: 84.07 %,     Val Loss: 0.4516,     Val Accuracy: 84.06 %
[EPOCH: 10],    Train Loss: 0.4524,     Train Accuracy: 84.16 %,     Val Loss: 0.4476,     Val Accuracy: 84.27 %
[EPOCH: 11],    Train Loss: 0.4478,     Train Accuracy: 84.29 %,     Val Loss: 0.4432,     Val Accuracy: 84.44 %
[EPOCH: 12],    Train Loss: 0.4427,     Train Accuracy: 84.46 %,     Val Loss: 0.4597,     Val Accuracy: 84.26 %
[EPOCH: 13],    Train Loss: 0.4385,     Train Accuracy: 84.68 %,     Val Loss: 0.4386,     Val Accuracy: 84.68 %
[EPOCH: 14],    Train Loss: 0.4358,     Train Accuracy: 84.77 %,     Val Loss: 0.4332,     Val Accuracy: 84.77 %
[EPOCH: 15],    Train Loss: 0.4315,     Train Accuracy: 84.89 %,     Val Loss: 0.4398,     Val Accuracy: 84.66 %
```

02. Activation Function

Unit 02 | Activation Function

선형 분류 X



중간에 Activation Function X

```python
class MLP(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear = nn.Sequential(
        nn.Linear(2, 10),
        nn.Linear(10, 1), # 10개의 노드를 가지는 1개의 은닉층 생성
        nn.Sigmoid()
    )

  def forward(self, x):
    return self.linear(x)
```

Loss 감소 X

```
EPOCH: 0, LOSS: 0.7647101879119873
EPOCH: 1000, LOSS: 0.6931471824645996
EPOCH: 2000, LOSS: 0.6931471824645996
EPOCH: 3000, LOSS: 0.6931471824645996
EPOCH: 4000, LOSS: 0.6931471824645996
EPOCH: 5000, LOSS: 0.6931471824645996
EPOCH: 6000, LOSS: 0.6931471824645996
EPOCH: 7000, LOSS: 0.6931471824645996
EPOCH: 8000, LOSS: 0.6931471824645996
EPOCH: 9000, LOSS: 0.6931471824645996
EPOCH: 10000, LOSS: 0.6931471824645996
```

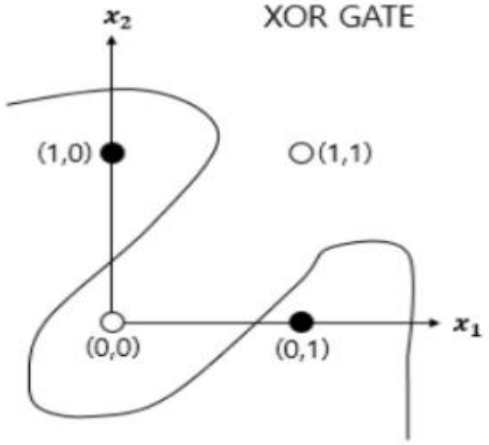
정확도 X

```
모델의 출력값(Hypothesis):
 [[0.50000006]
 [0.49999994]
 [0.5000001 ]
 [0.49999994]]
모델의 예측값(Predicted):
 [[1.]
 [0.]
 [1.]
 [0.]]
실제값(Y):
 [[0.]
 [1.]
 [1.]
 [0.]]
정확도(Accuracy):  0.5
```

중간에 Activation Function이 없는
Neural Net은 Linear Regression Model과 같음!

Unit 02 | Activation Function

비선형 분류 O



중간에 Activation Function O

```
class MLP(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear = nn.Sequential(
        nn.Linear(2, 10),
        nn.Sigmoid(),
        nn.Linear(10, 1), # 10개의 노드를 가지는 1개의 은닉층 생성
        nn.Sigmoid()
    )

  def forward(self, x):
    return self.linear(x)
```

Loss 감소 O

```
EPOCH: 0, LOSS: 0.6942969560623169
EPOCH: 1000, LOSS: 0.6921975612640381
EPOCH: 2000, LOSS: 0.6882604360580444
EPOCH: 3000, LOSS: 0.6495662331581116
EPOCH: 4000, LOSS: 0.400748074054718
EPOCH: 5000, LOSS: 0.11533121764659882
EPOCH: 6000, LOSS: 0.0511920303106308
EPOCH: 7000, LOSS: 0.031123246997594833
EPOCH: 8000, LOSS: 0.0219508782029151 9
EPOCH: 9000, LOSS: 0.016813486814498 9
EPOCH: 10000, LOSS: 0.013563135638833046
```

정확도 O

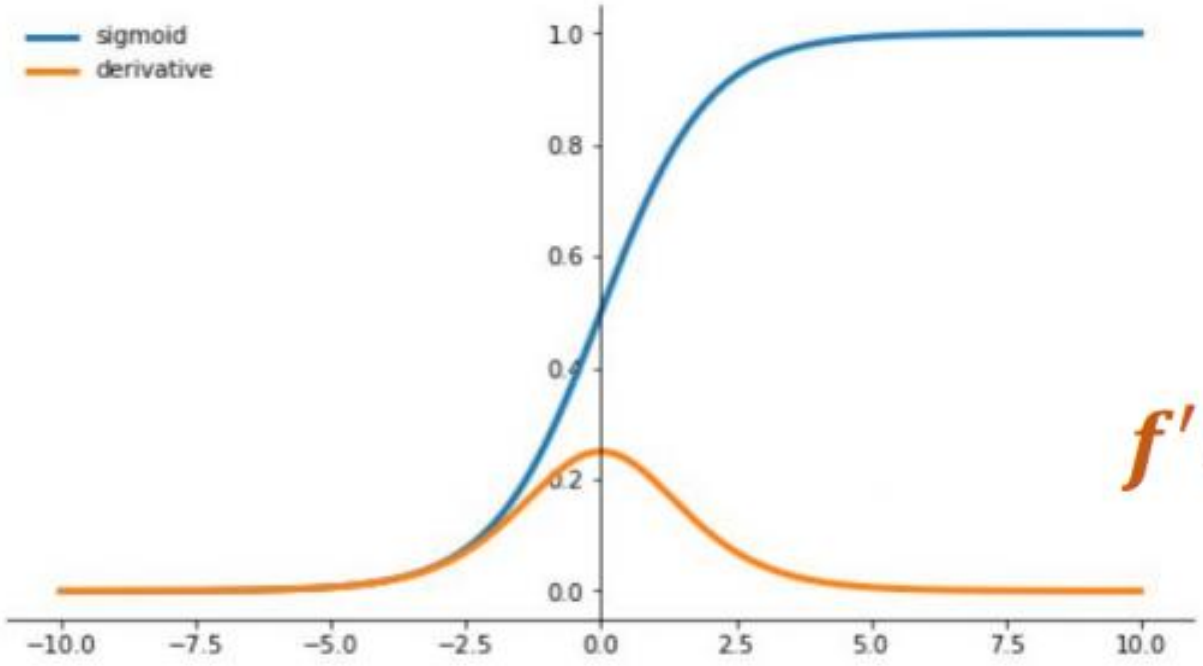
```
모델의 출력값(Hypothesis):
 [[0.01344184]
 [0.98397887]
 [0.9883984 ]
 [0.01280589]]
모델의 예측값(Predicted):
 [[0.]
 [1.]
 [1.]
 [0.]]
실제값(Y):
 [[0.]
 [1.]
 [1.]
 [0.]]
정확도(Accuracy):  1.0
```

비선형 분류 문제를 해결하기 위하여
Activation Function을 사용하는 것!

종류

- Sigmoid
- Tanh
- ReLU
- Leaky ReLU
- etc. elu, PReLU ...

Sigmoid

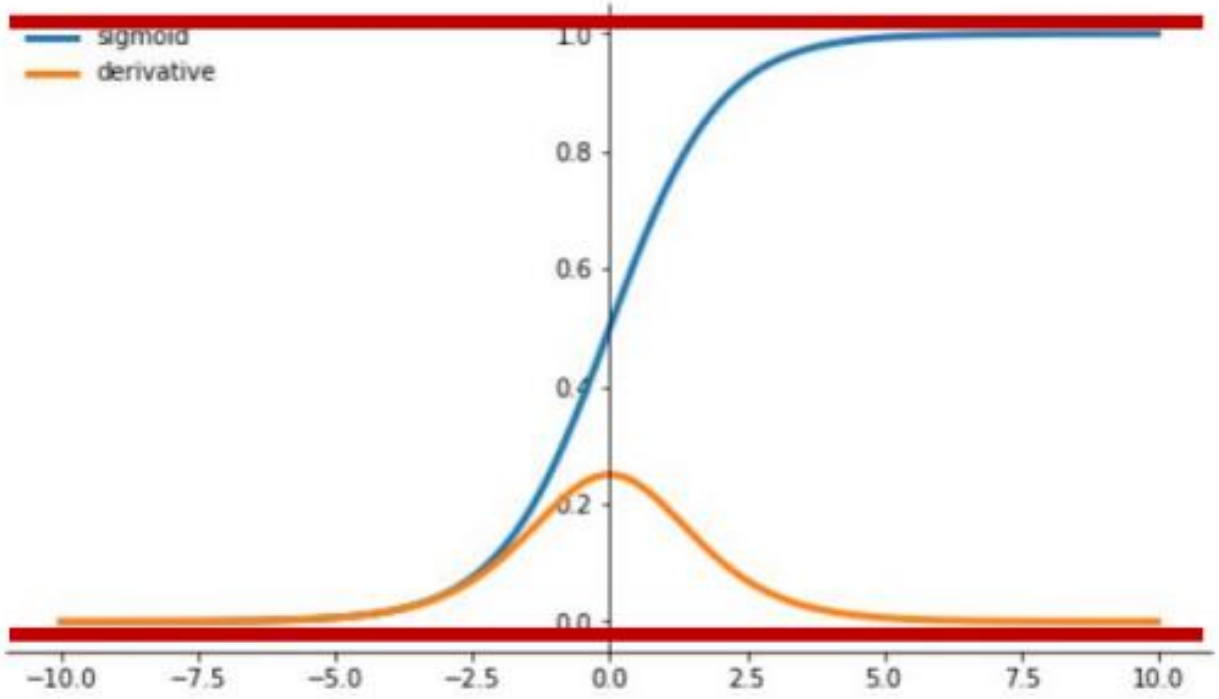


$$f(x) = \frac{1}{1 + e^{-x}}$$

$$f'(x) = f(x)(1 - f(x))$$

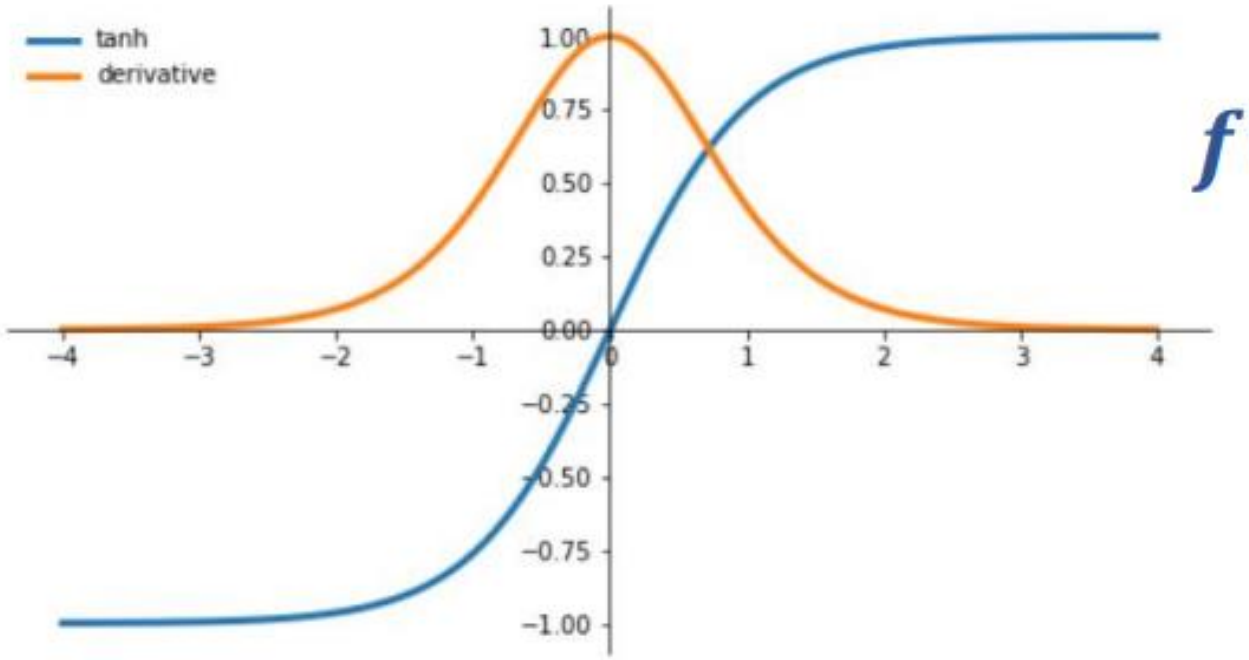
미분 가능!

Sigmoid



0~1 사이
↓
출력값 = 확률 형태

Tanh

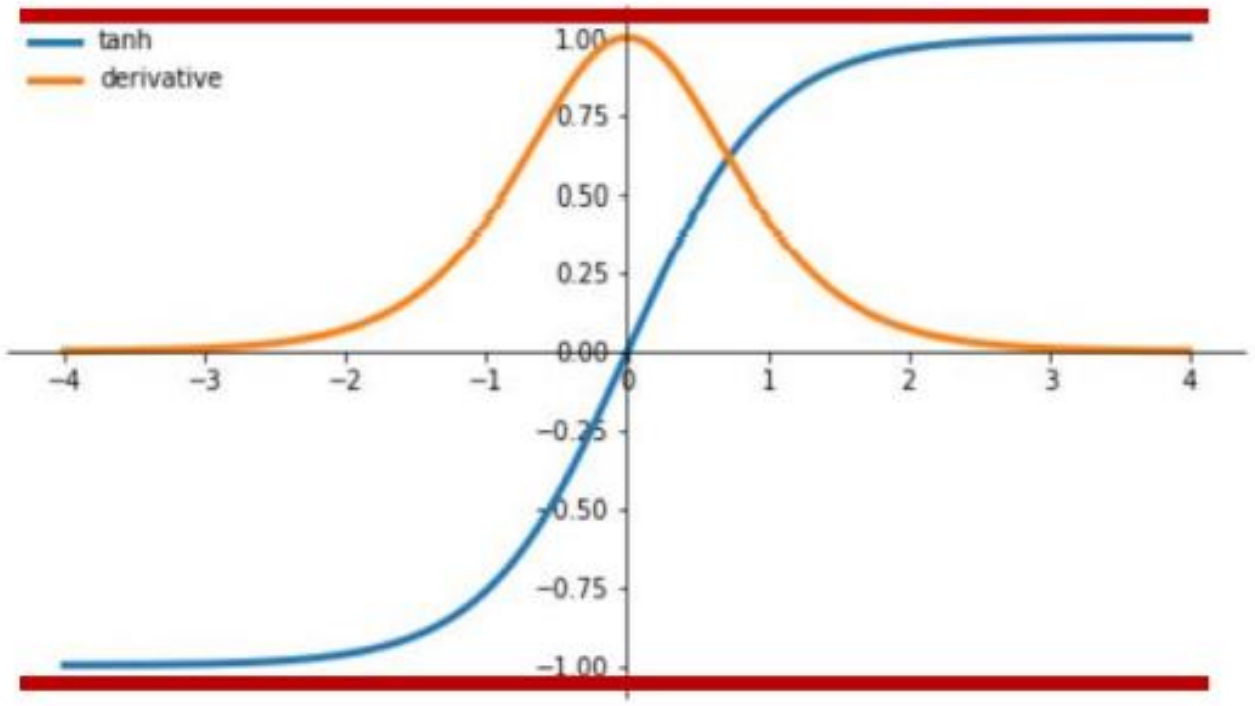


$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

$$f'(x) = 1 - f(x)^2$$

미분 가능!

Tanh



-1 ~ 1 사이

↓

출력 범위 더 넓음

경사면이 더 가파름

빠르게 수렴

그런데

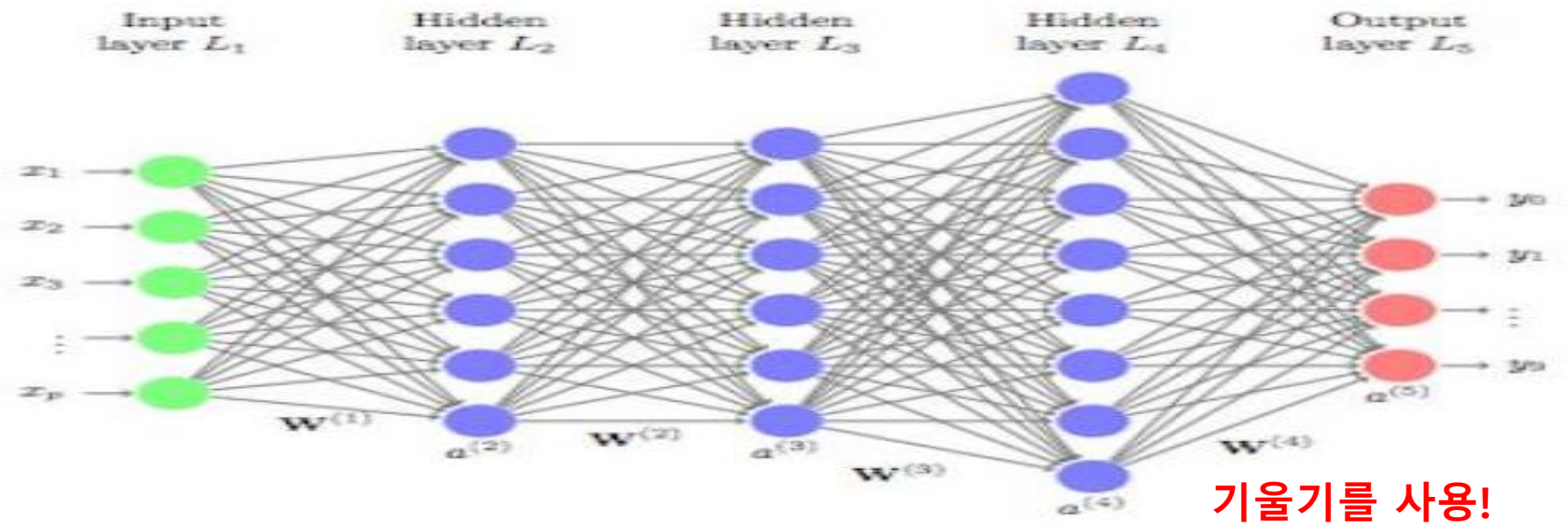
Sigmoid와 Tanh은

매우 큰 문제점이 존재!

Vanishing Gradient Problem

기울기 소실 문제

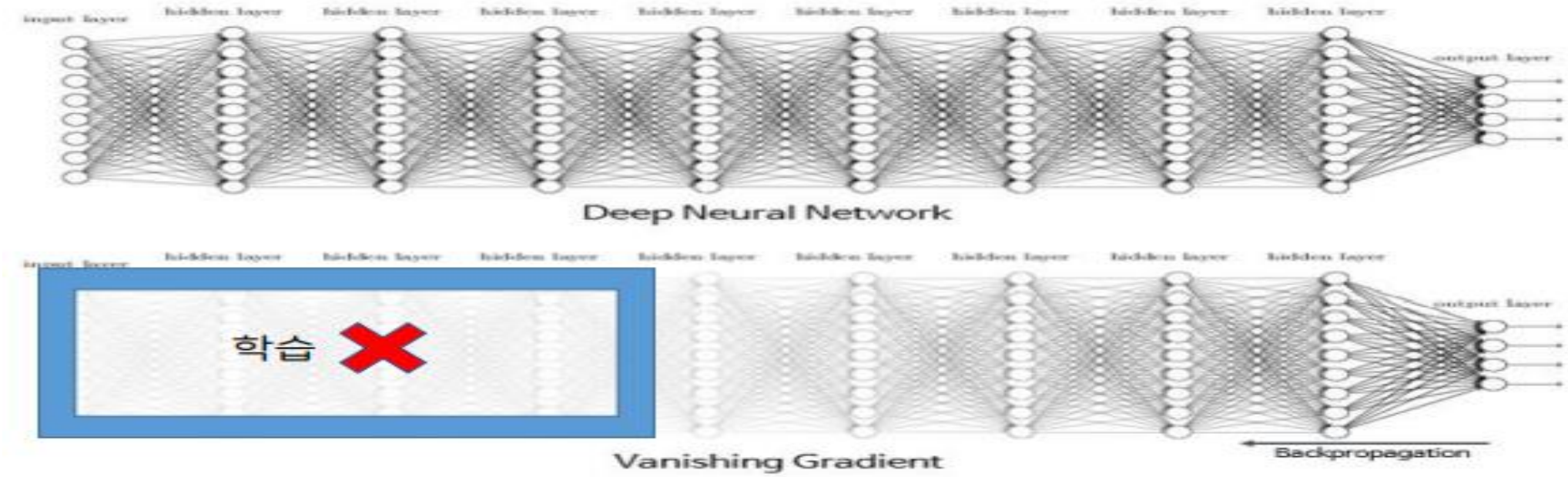
Vanishing Gradient Problem 이란?



기울기를 사용!

Forward Propagation -> Loss 계산 -> Back Propagation -> Update Parameters

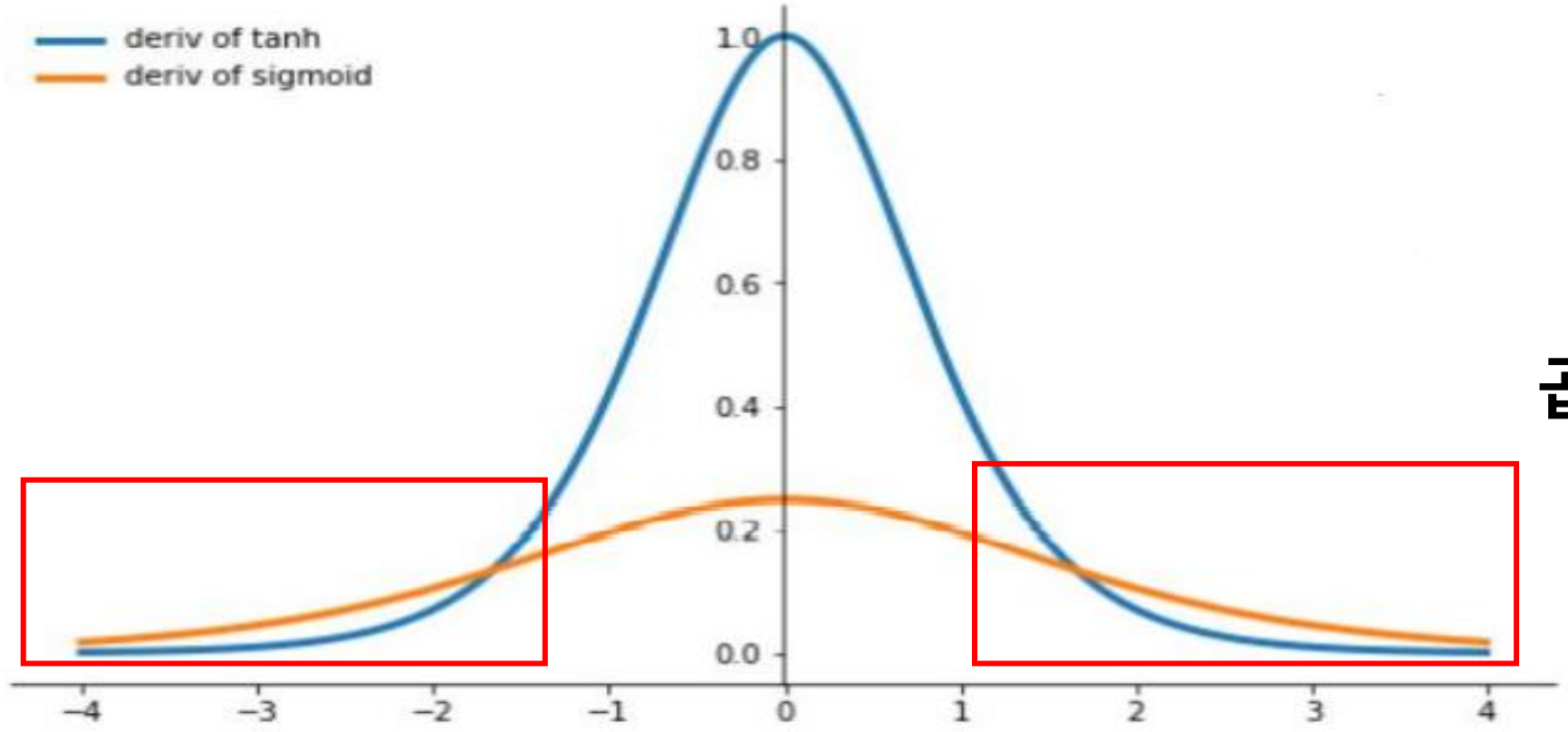
Vanishing Gradient Problem 이란?



역전파 과정에서 오자 그래디언트를 계산하면서(미분) 파라미터를 수정하는데,

하위층으로 진행됨에 따라 그래디언트가 점점 작아져 좋은 솔루션을 내지 못하는 것!

왜?



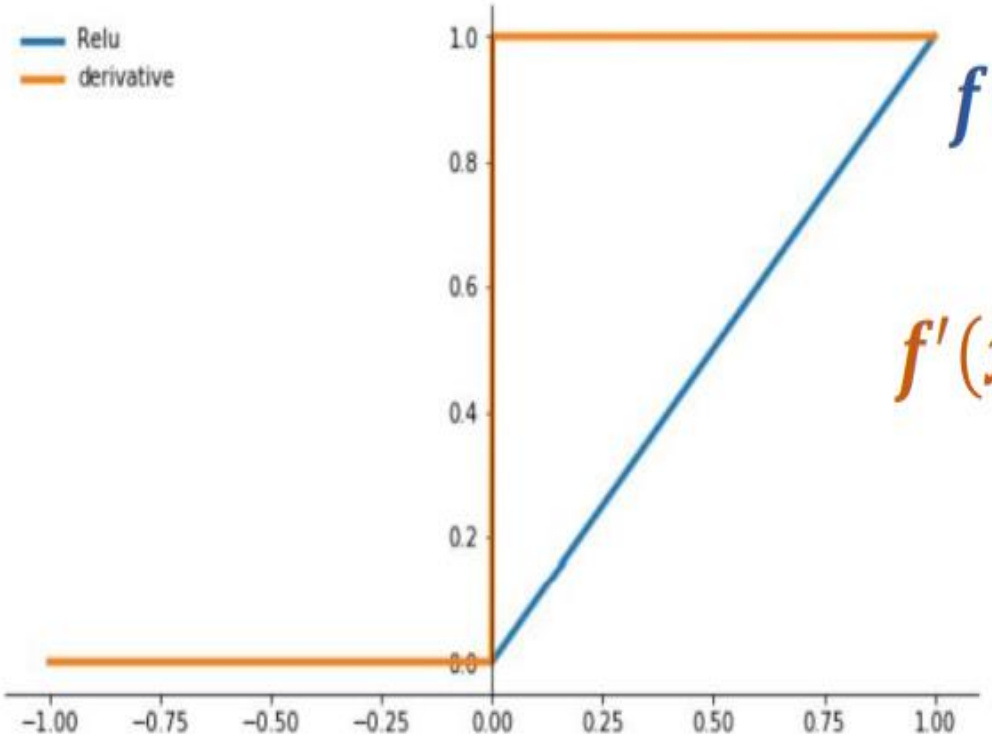
미분 값이 1이하

곱할수록 점점 작아짐

Rectified Linear Unit(ReLU)

그래서 나온 해결책!

ReLU



$$f(x) = max(0, x)$$

$$f'(x) = 0 \; when \; x < 0$$
$$otherwise \; 1$$

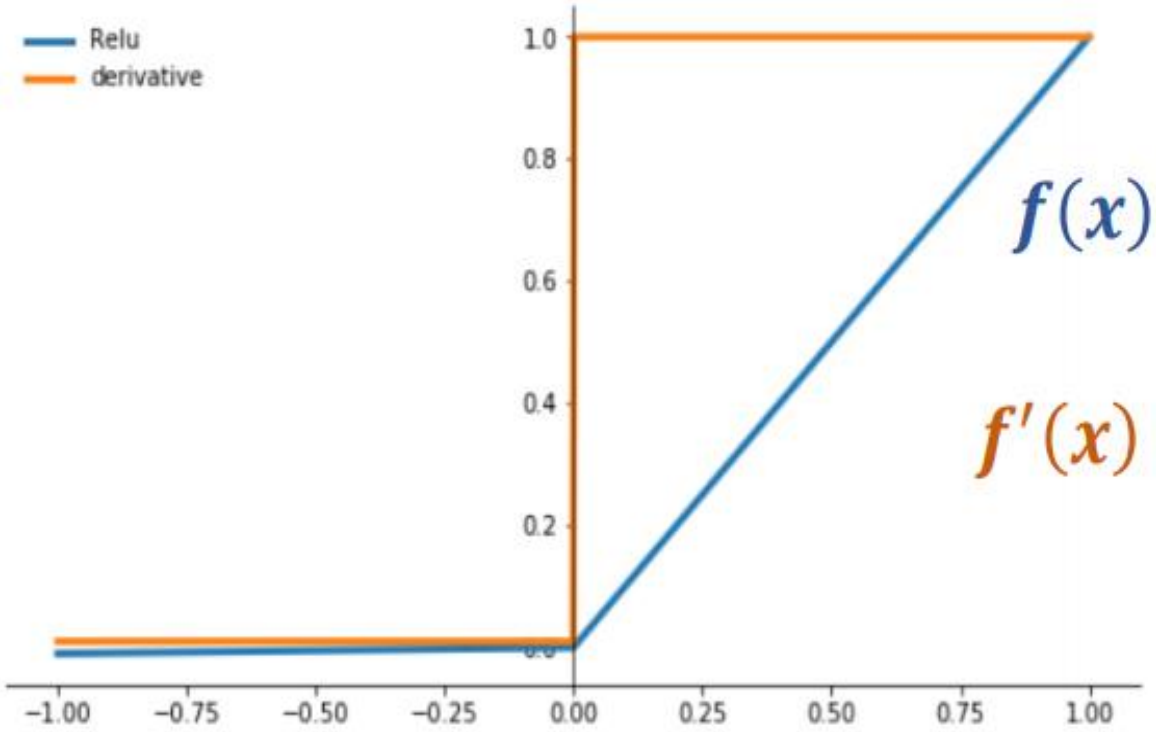
미분값이 양수인 경우 1

Vanishing Gradient 해결

But 음수 값 무시

dying ReLU

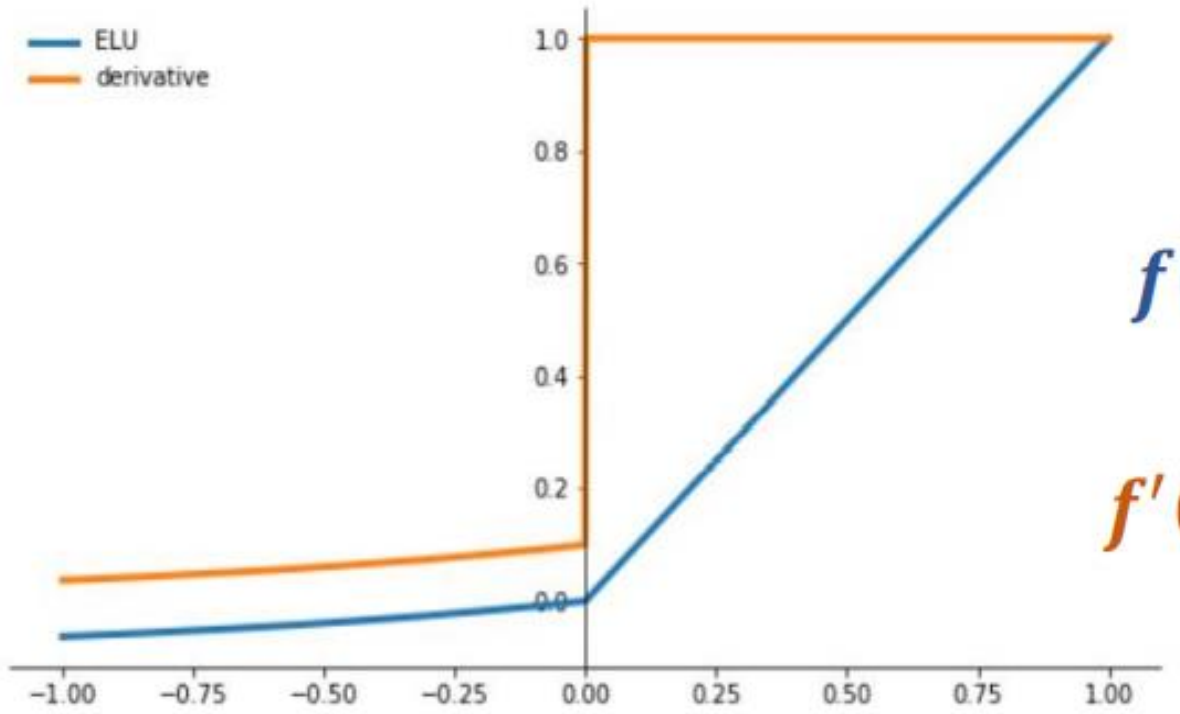
Leaky ReLU



$$f(x) = max(0.01x, x)$$

$$f'(x) = 0.01 \; when \; x < 0$$
$$otherwise \; 1$$

ELU



$$f(x) = x \quad if\ x > 0$$
$$f(x) = \alpha(e^x - 1)\ o.w.$$

$$f'(x) = \alpha e^x\ when\ x \leq 0$$
$$otherwise\ 1$$

Activation Function 추가!

```python
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.linear1 = nn.Sequential(
        nn.Linear(28 * 28, 512),
        nn.ReLU()
    )
    self.linear2 = nn.Sequential(
        nn.Linear(512, 256),
        nn.ReLU()
    )
    self.linear3 = nn.Sequential(
        nn.Linear(256, 128),
        nn.ReLU()
    )
    self.linear4 = nn.Sequential(
        nn.Linear(128, 10)
    )

  def forward(self, x):
    x = x.view(-1, 28 * 28)
    x = self.linear1(x)
    x = self.linear2(x)
    x = self.linear3(x)
    x = self.linear4(x)
    return x
```

중간에 ReLU 추가!

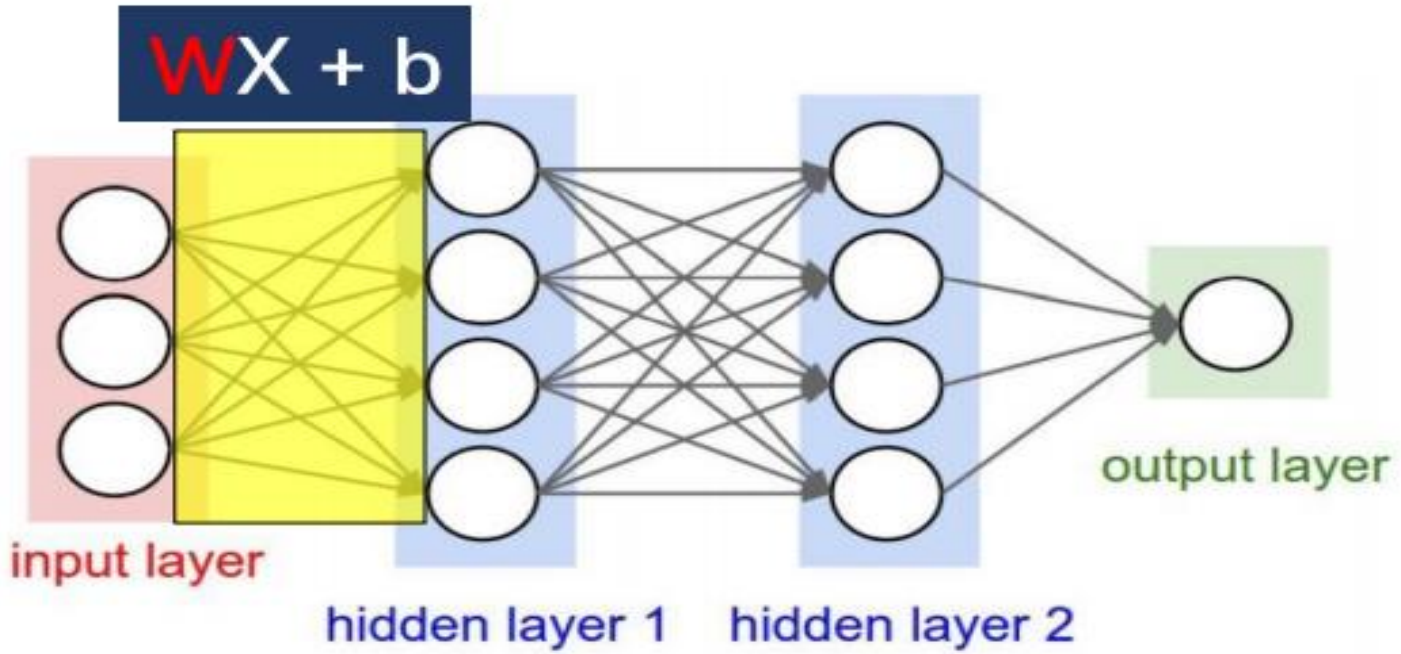
```python
model = Net().to(DEVICE)
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum = 0.5)
criterion = nn.CrossEntropyLoss()
```

```
[EPOCH: 1],     Train Loss: 1.9380,     Train Accuracy: 33.59 %,     Val Loss: 1.2035,     Val Accuracy: 59.36 %
[EPOCH: 2],     Train Loss: 0.9337,     Train Accuracy: 64.72 %,     Val Loss: 0.7758,     Val Accuracy: 70.08 %
[EPOCH: 3],     Train Loss: 0.7257,     Train Accuracy: 73.16 %,     Val Loss: 0.6562,     Val Accuracy: 76.92 %
[EPOCH: 4],     Train Loss: 0.6343,     Train Accuracy: 77.55 %,     Val Loss: 0.5887,     Val Accuracy: 78.45 %
[EPOCH: 5],     Train Loss: 0.5727,     Train Accuracy: 79.90 %,     Val Loss: 0.5600,     Val Accuracy: 80.05 %
[EPOCH: 6],     Train Loss: 0.5369,     Train Accuracy: 81.03 %,     Val Loss: 0.5068,     Val Accuracy: 81.94 %
[EPOCH: 7],     Train Loss: 0.5082,     Train Accuracy: 82.05 %,     Val Loss: 0.4916,     Val Accuracy: 82.18 %
[EPOCH: 8],     Train Loss: 0.4890,     Train Accuracy: 82.71 %,     Val Loss: 0.4704,     Val Accuracy: 83.12 %
[EPOCH: 9],     Train Loss: 0.4719,     Train Accuracy: 83.43 %,     Val Loss: 0.4673,     Val Accuracy: 82.94 %
[EPOCH: 10],    Train Loss: 0.4605,     Train Accuracy: 83.70 %,     Val Loss: 0.4497,     Val Accuracy: 84.27 %
[EPOCH: 11],    Train Loss: 0.4460,     Train Accuracy: 84.32 %,     Val Loss: 0.4362,     Val Accuracy: 84.24 %
[EPOCH: 12],    Train Loss: 0.4374,     Train Accuracy: 84.49 %,     Val Loss: 0.4301,     Val Accuracy: 84.54 %
[EPOCH: 13],    Train Loss: 0.4291,     Train Accuracy: 84.84 %,     Val Loss: 0.4254,     Val Accuracy: 84.64 %
[EPOCH: 14],    Train Loss: 0.4205,     Train Accuracy: 85.08 %,     Val Loss: 0.4084,     Val Accuracy: 85.44 %
[EPOCH: 15],    Train Loss: 0.4106,     Train Accuracy: 85.55 %,     Val Loss: 0.4050,     Val Accuracy: 85.59 %
```

성능 향상!

03. Weight Initialization

Weight Initialization 이란?



NN을 학습시킬 때 W의 초깃값을 임의로 설정해주는 것!

왜?

초기 값을 모두 같은 값으로 설정!(ex. 0 or 1)

역전파 과정에서 모든 가중치의 값이 똑같이 갱신됨! (학습 X)

이는 가중치를 여러 개 갖는 의미를 사라지게 함!

어떻게 해결?

초기 값을 **무작위로 설정**하자!

가중치가 고르게 되어버리는 상황을 막아줘,

모델이 다양성을 가지게 됨!(가중치가 다양함)

그런데?

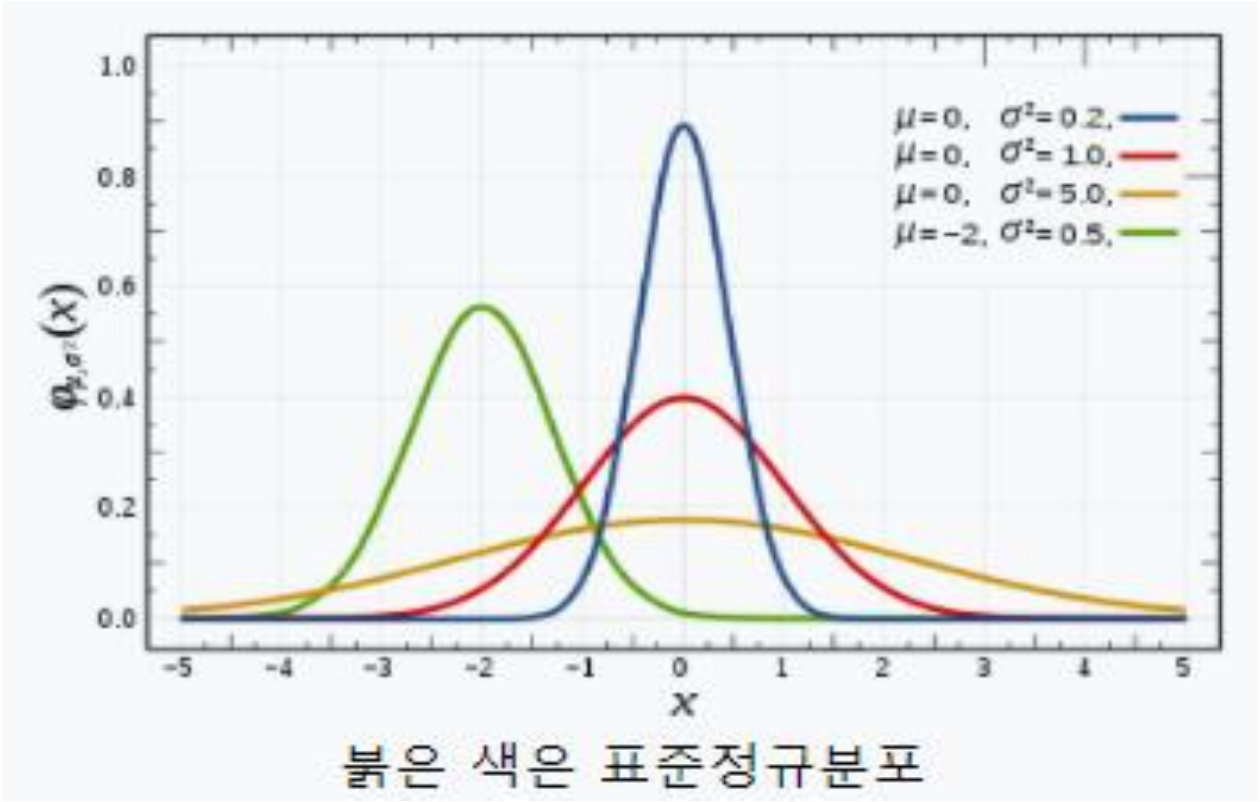
초기 값을 단순히 무작위로 설정하니....

가중치가 너무 크거나 작아서 학습이 잘 안돼!

그래서?

어떠한 분포에서 가중치를 랜덤하게 뽑자!

Normal Distribution(정규 분포)



붉은 색은 표준정규분포

표준정규분포:

평균 = 0, 분산 = 1

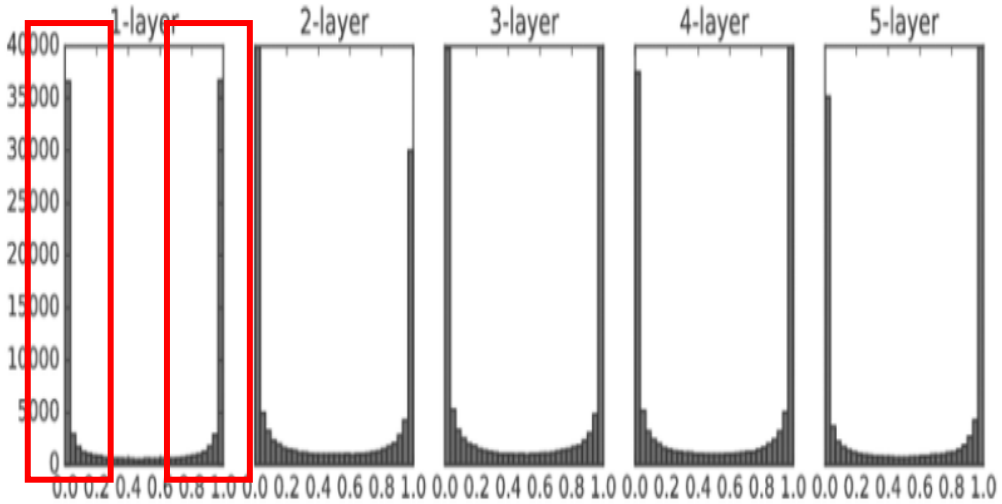
랜덤하게 뽑으니 학습이 잘되는 느낌!

하지만! 문제점이 존재....

문제점

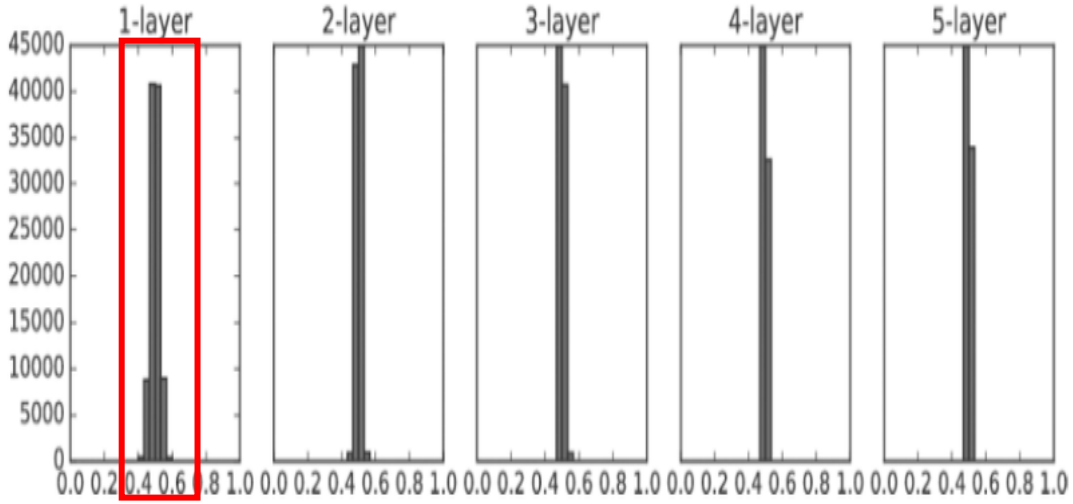
사진은 활성화 값(sigmoid)의 분포를 히스토그램으로 표현한 것

가중치의 표준편차1



기울기 소실

가중치의 표준편차0.01



표현력 제한

그래서 학자들이
최적의 학습을 위한 초기 가중치 설정에 대하여 연구를 함!

결과가 바로!
Xavier 초깃값, He 초깃값

Xavier 초깃값

Xavier Normal Initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

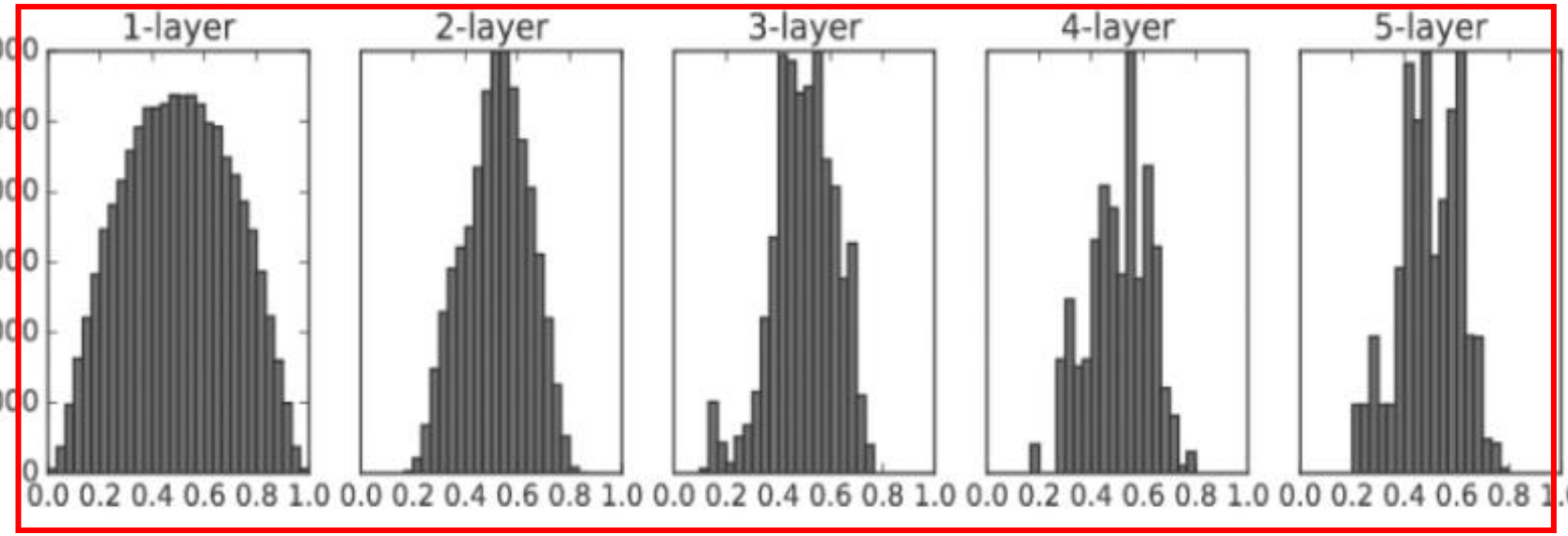
Xavier Uniform Initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in} + n_{out}}}, +\sqrt{\frac{6}{n_{in} + n_{out}}})$$

n_in = 들어오는 노드의 수 / n_out = 나가는 노드의 수

Xavier 초깃값

고른 분포! == 가중치가 다양하다! == 표현력이 좋다!



Sigmoid, Tanh 와 함께 사용할 때 성능이 좋음!

He(kaiming) 초깃값

He Normal Initialization

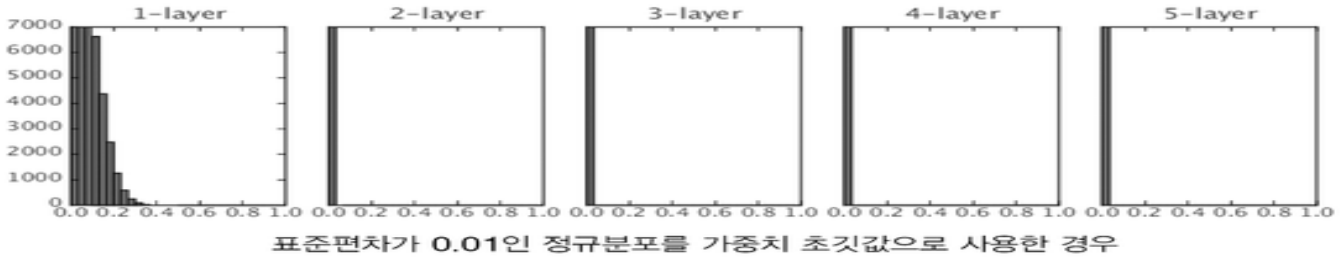
$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in}}}$$

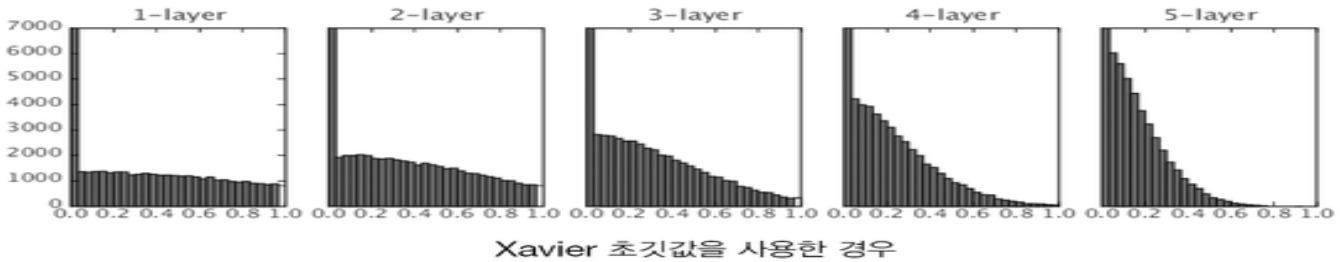
He Uniform Initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in}}}, +\sqrt{\frac{6}{n_{in}}})$$

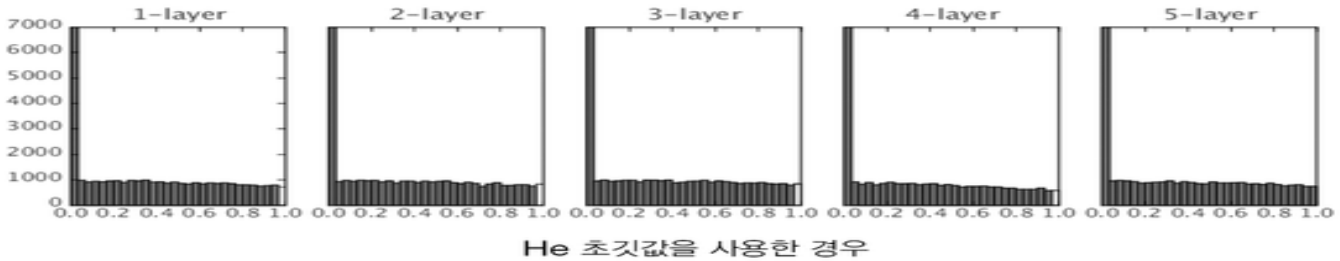
He 초깃값



표준편차가 0.01인 정규분포를 가중치 초깃값으로 사용한 경우



Xavier 초깃값을 사용한 경우



He 초깃값을 사용한 경우

ReLU + 정규분포 = 기울기 소실

ReLU + Xavier = 기울기 소실

BEST!

ReLU + He = 고른 분포

Unit 03 | Weight Initialization

가중치 초기화 추가!

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.linear1 = nn.Sequential(
            nn.Linear(28 * 28, 512),
            nn.ReLU()
        )
        self.linear2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU()
        )
        self.linear3 = nn.Sequential(
            nn.Linear(256, 128),
            nn.ReLU()
        )
        self.linear4 = nn.Sequential(
            nn.Linear(128, 10)
        )

        self._init_weight_()

    def _init_weight_(self):
        for m in self.linear1:
            if isinstance(m, nn.Linear):
                nn.init.kaiming_uniform_(m.weight)

        for m in self.linear2:
            if isinstance(m, nn.Linear):
                nn.init.kaiming_uniform_(m.weight)

        for m in self.linear3:
            if isinstance(m, nn.Linear):
                nn.init.kaiming_uniform_(m.weight)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = self.linear1(x)
        x = self.linear2(x)
        x = self.linear3(x)
        x = self.linear4(x)
        return x
```

He 초기화!

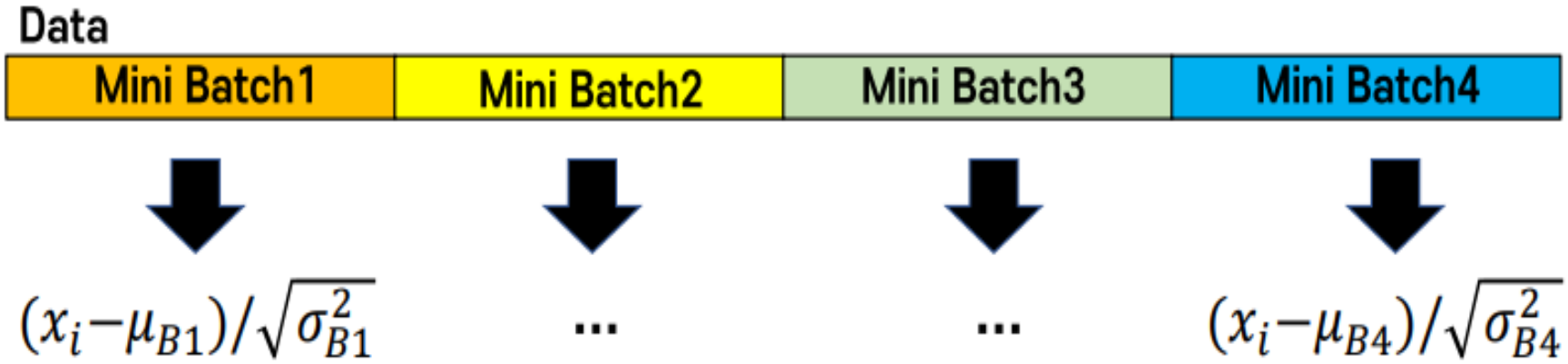
```python
model = Net().to(DEVICE)
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum = 0.5)
criterion = nn.CrossEntropyLoss()
```

```
[EPOCH: 1],    Train Loss: 0.9834,    Train Accuracy: 67.64 %,    Val Loss: 0.6181,    Val Accuracy: 78.76 %
[EPOCH: 2],    Train Loss: 0.5687,    Train Accuracy: 80.35 %,    Val Loss: 0.5099,    Val Accuracy: 82.48 %
[EPOCH: 3],    Train Loss: 0.4964,    Train Accuracy: 82.70 %,    Val Loss: 0.4656,    Val Accuracy: 83.98 %
[EPOCH: 4],    Train Loss: 0.4615,    Train Accuracy: 83.77 %,    Val Loss: 0.4841,    Val Accuracy: 82.45 %
[EPOCH: 5],    Train Loss: 0.4414,    Train Accuracy: 84.31 %,    Val Loss: 0.4273,    Val Accuracy: 84.82 %
[EPOCH: 6],    Train Loss: 0.4253,    Train Accuracy: 84.95 %,    Val Loss: 0.4496,    Val Accuracy: 84.25 %
[EPOCH: 7],    Train Loss: 0.4100,    Train Accuracy: 85.36 %,    Val Loss: 0.4010,    Val Accuracy: 85.97 %
[EPOCH: 8],    Train Loss: 0.3958,    Train Accuracy: 85.95 %,    Val Loss: 0.3845,    Val Accuracy: 86.66 %
[EPOCH: 9],    Train Loss: 0.3842,    Train Accuracy: 86.37 %,    Val Loss: 0.3895,    Val Accuracy: 86.28 %
[EPOCH: 10],    Train Loss: 0.3740,    Train Accuracy: 86.66 %,    Val Loss: 0.3932,    Val Accuracy: 85.74 %
[EPOCH: 11],    Train Loss: 0.3649,    Train Accuracy: 87.02 %,    Val Loss: 0.3689,    Val Accuracy: 87.17 %
[EPOCH: 12],    Train Loss: 0.3574,    Train Accuracy: 87.27 %,    Val Loss: 0.3871,    Val Accuracy: 86.45 %
[EPOCH: 13],    Train Loss: 0.3487,    Train Accuracy: 87.55 %,    Val Loss: 0.3627,    Val Accuracy: 87.38 %
[EPOCH: 14],    Train Loss: 0.3413,    Train Accuracy: 87.81 %,    Val Loss: 0.3714,    Val Accuracy: 86.68 %
[EPOCH: 15],    Train Loss: 0.3353,    Train Accuracy: 87.82 %,    Val Loss: 0.3594,    Val Accuracy: 87.23 %
```

성능 향상!

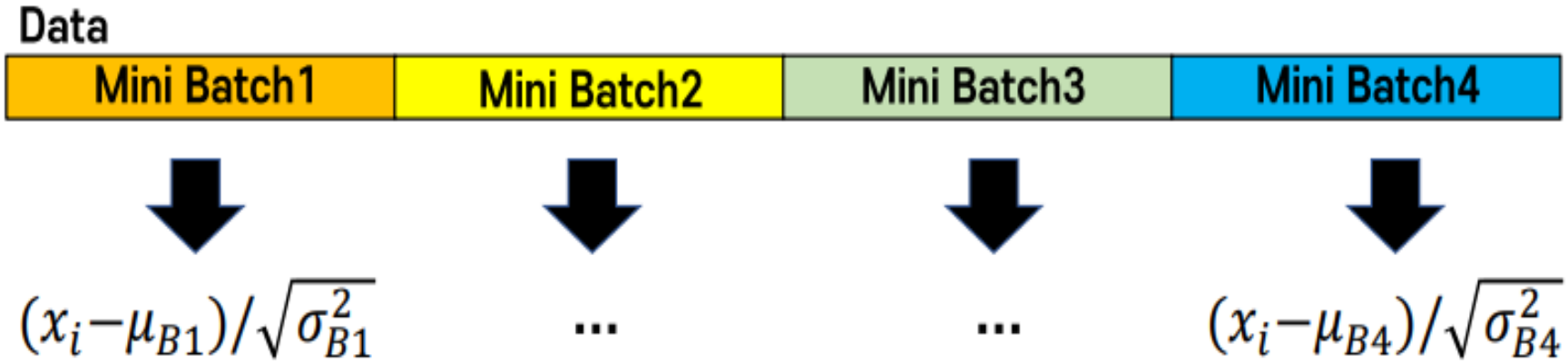
04. Batch Normalization

Batch Normalization 이란?



Mini Batch 단위를 Normalize 시키는 것!

Batch Normalization 이란?



네트워크망 내부 데이터를 변경하여 안정적으로 학습 가능!

왜?

평균 0, 분산 1로 Nornalize 하여 Internal Covariance Shift 해결

- Internal Covariance Shift -

Network의 각 층이나 Activation 마다 Input의 Distribution이 달라지는 현상

왜?

직관적으로 이야기하면

활성화 함수 통과 전에 배치 정규화를 수행함으로써

층마다 일정하지 않았던 Data의 분포를 일정하게 만드는 것!

이는 가중치를 고르게 분포 시키는 것과 비슷함!

가중치가 고르면? 학습이 잘 됨!(가중치 초기화를 하는 이유)

따라서 배치 정규화도 학습이 잘 되는 것!

효과

장점1. 기울기 소실 / 팽창 방지

장점2. 학습 시 가중치 초기값에 크게 의존하지 않게 해줌.

장점3. 자체적 Regularization 효과가 있음.

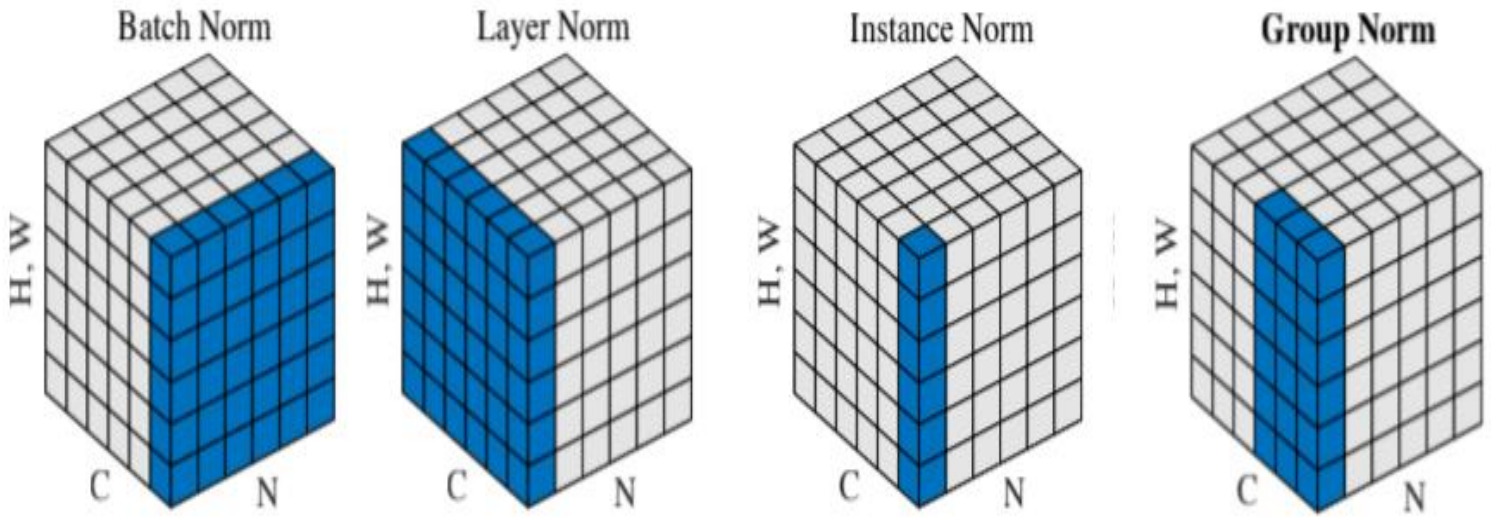
이는 Model 구축 시에 dropout 등을 제외할 수 있게 해 학습 속도 UP!

단점. 개별 유닛의 관계를 고려하지 못함

(decorrelated BN, Variance Consistency Loss을 통해 해결)

다른 Normalization

- Weight Normalization
- Layer Normalization
- Instance Normalization
- Group Normalization



채널 단위로 정규화

feature 차원 정규화

LN + IN

Unit 04 | Batch Normalization

배치 정규화 추가!

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.linear1 = nn.Sequential(
            nn.Linear(28 * 28, 512),
            nn.BatchNorm1d(512),
            nn.ReLU()
        )
        self.linear2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU()
        )
        self.linear3 = nn.Sequential(
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU()
        )
        self.linear4 = nn.Sequential(
            nn.Linear(128, 10)
        )

        self._init_weight_()

```

Linear 층과
활성화 함수 사이에
Batch Normalization 추가

```
[EPOCH: 1],    Train Loss: 0.8250,    Train Accuracy: 74.81 %,    Val Loss: 0.5111,    Val Accuracy: 82.68 %
[EPOCH: 2],    Train Loss: 0.4741,    Train Accuracy: 83.72 %,    Val Loss: 0.4178,    Val Accuracy: 85.72 %
[EPOCH: 3],    Train Loss: 0.4059,    Train Accuracy: 85.75 %,    Val Loss: 0.3770,    Val Accuracy: 86.99 %
[EPOCH: 4],    Train Loss: 0.3665,    Train Accuracy: 87.00 %,    Val Loss: 0.3589,    Val Accuracy: 87.29 %
[EPOCH: 5],    Train Loss: 0.3406,    Train Accuracy: 87.90 %,    Val Loss: 0.3522,    Val Accuracy: 87.34 %
[EPOCH: 6],    Train Loss: 0.3180,    Train Accuracy: 88.70 %,    Val Loss: 0.3452,    Val Accuracy: 87.63 %
[EPOCH: 7],    Train Loss: 0.2993,    Train Accuracy: 89.16 %,    Val Loss: 0.3228,    Val Accuracy: 88.47 %
[EPOCH: 8],    Train Loss: 0.2815,    Train Accuracy: 89.83 %,    Val Loss: 0.3331,    Val Accuracy: 87.88 %
[EPOCH: 9],    Train Loss: 0.2665,    Train Accuracy: 90.43 %,    Val Loss: 0.3187,    Val Accuracy: 88.35 %
[EPOCH: 10],   Train Loss: 0.2510,    Train Accuracy: 91.09 %,    Val Loss: 0.3183,    Val Accuracy: 88.27 %
[EPOCH: 11],   Train Loss: 0.2369,    Train Accuracy: 91.58 %,    Val Loss: 0.3185,    Val Accuracy: 88.76 %
[EPOCH: 12],   Train Loss: 0.2245,    Train Accuracy: 92.11 %,    Val Loss: 0.3144,    Val Accuracy: 88.67 %
[EPOCH: 13],   Train Loss: 0.2106,    Train Accuracy: 92.62 %,    Val Loss: 0.3232,    Val Accuracy: 88.45 %
[EPOCH: 14],   Train Loss: 0.1992,    Train Accuracy: 93.11 %,    Val Loss: 0.3274,    Val Accuracy: 88.55 %
[EPOCH: 15],   Train Loss: 0.1877,    Train Accuracy: 93.45 %,    Val Loss: 0.3173,    Val Accuracy: 88.76 %
```

성능 향상!

05. Optimization

기존 뉴럴넷이 가중치 parameter들을 최적화(optimize)하는 방법

Gradient Decent

Loss Function이 현 가중치에서의 기울기를 구해서

Loss를 줄이는 방향으로 업데이트

현재 가진 Weight 세팅에서,

내가 가진 데이터를 다 넣으면

전체 에러가 계산된다!

트레이닝 데이터가 몇 억 건이면 .....

어느 세월에 다하는가 .....

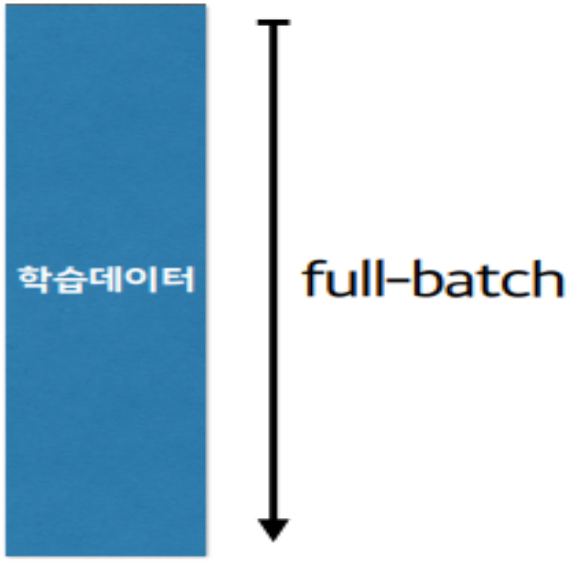
GD보다 빠른 옵티마이저는 없을까?

Stochastic Gradient Decent

컨셉: 느린 완벽보다 조금만 훑어보고 일단 빨리 가자!

Unit 05 | Optimization

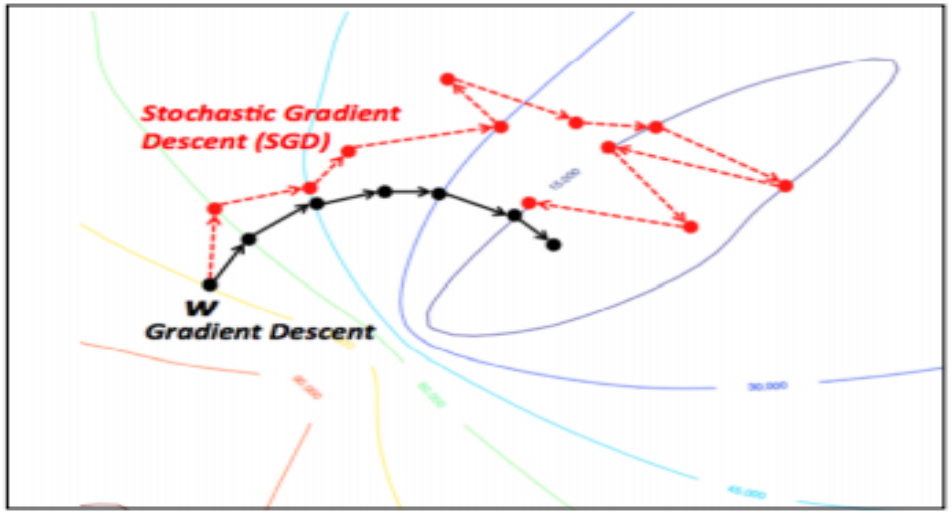
Gradient Decent



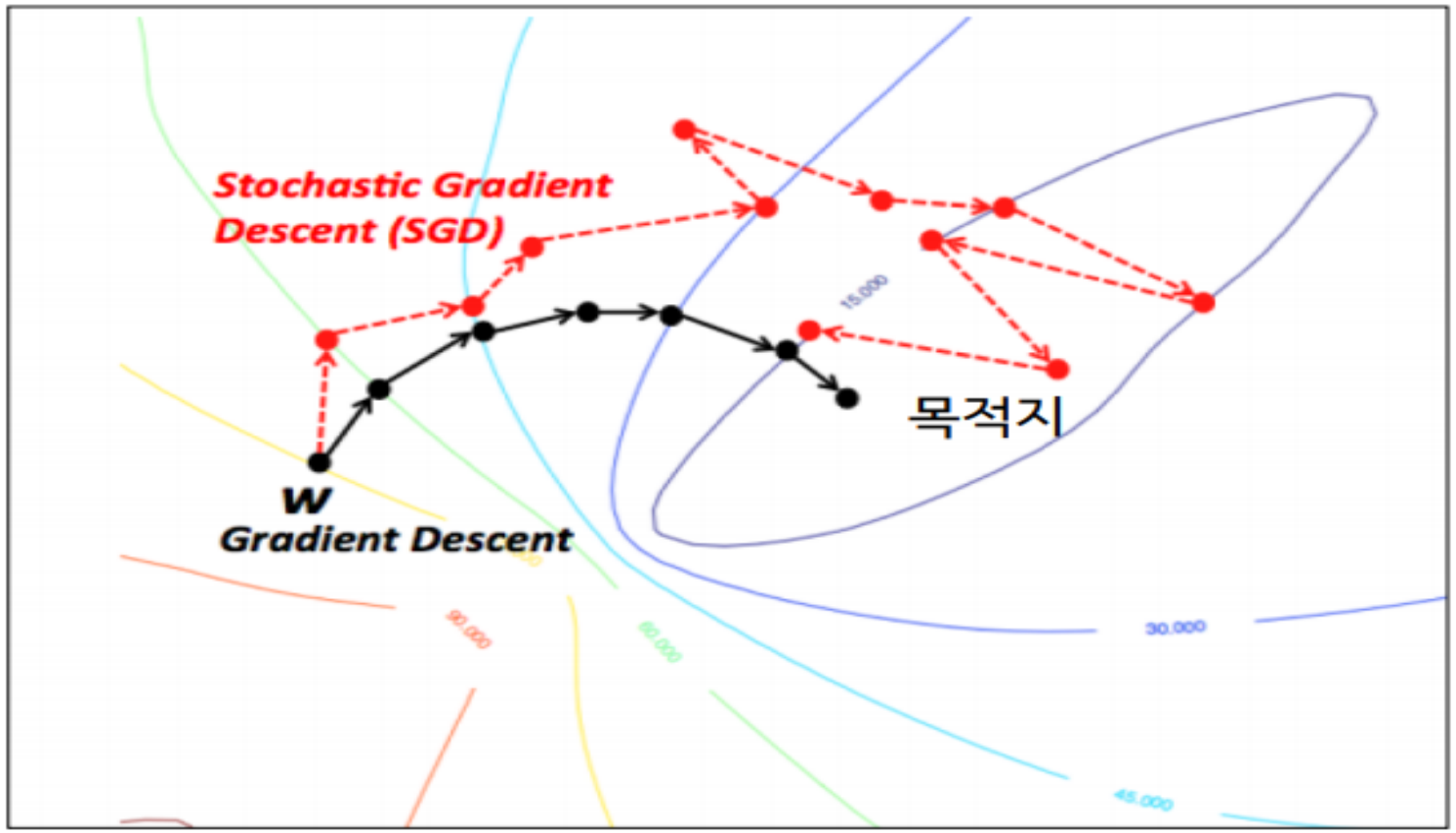
전부다 읽고나서
최적의 1스텝 간다.

Stochastic Gradient Decent

작은 토막마다
일단 1스텝간다.



GD vs SGD



Gradient Decent

모든 걸 계산(1시간)후
최적의 한스텝
6스텝 * 1시간 = 6시간

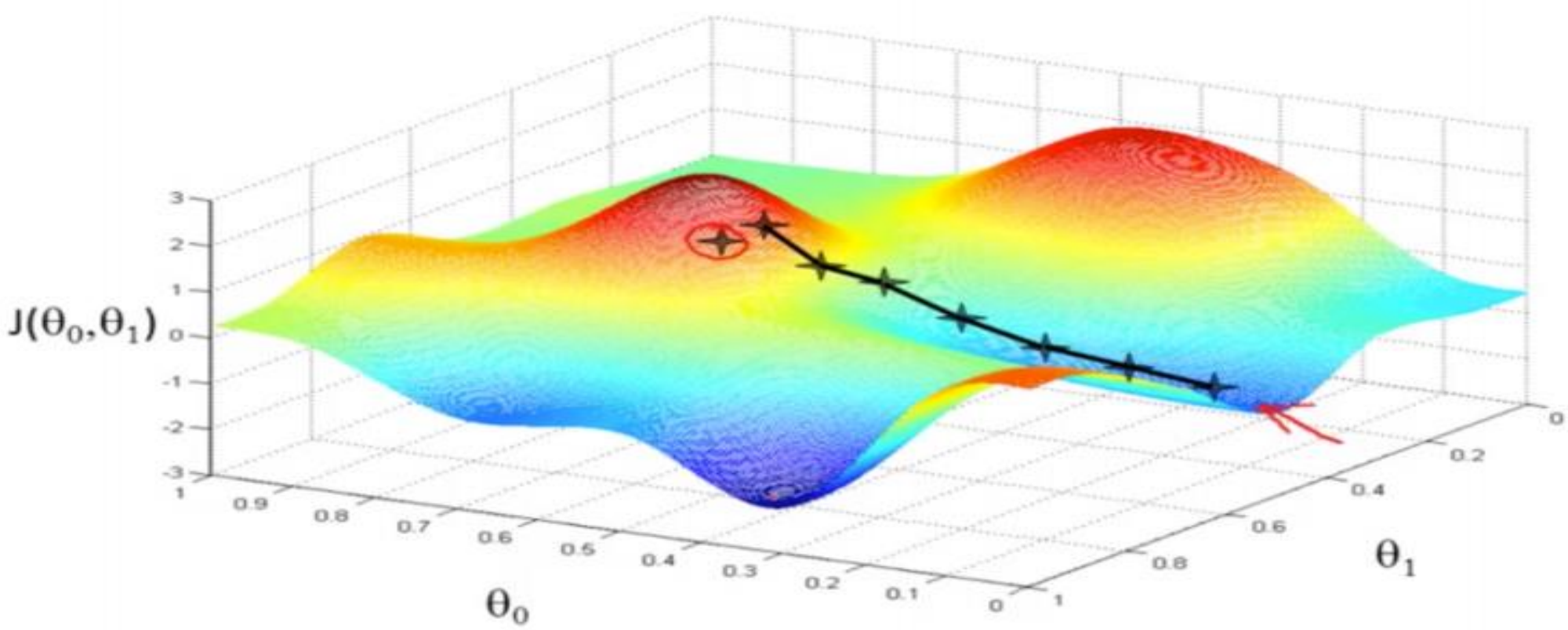
최적인데 너무 느리다!

Stochastic Gradient Descent

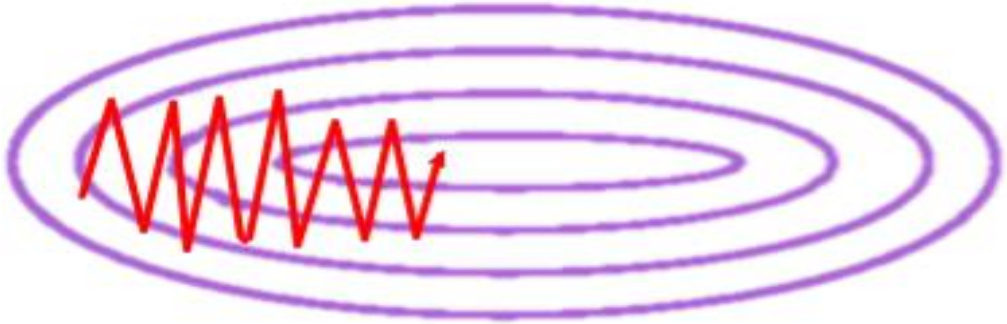
일부만 검토(5분)
틀려도 일단 간다! 빠른 스텝!
11스텝 * 5분 = 55분 < 1시간

조금 헤매도 어쨌든 인근에 아주 빨리 갔다!

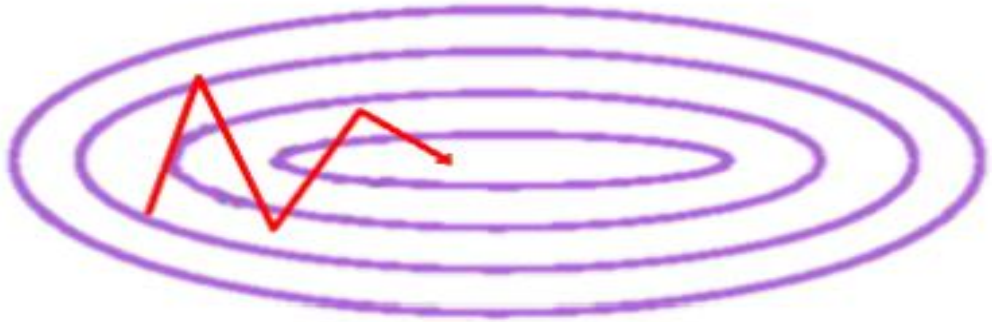
최적화는 좋은 오솔길을 찾아서 산을 내려오는 것과 매우 비슷!



근데 미니 배치를 하다 보니 방향 문제가 존재!

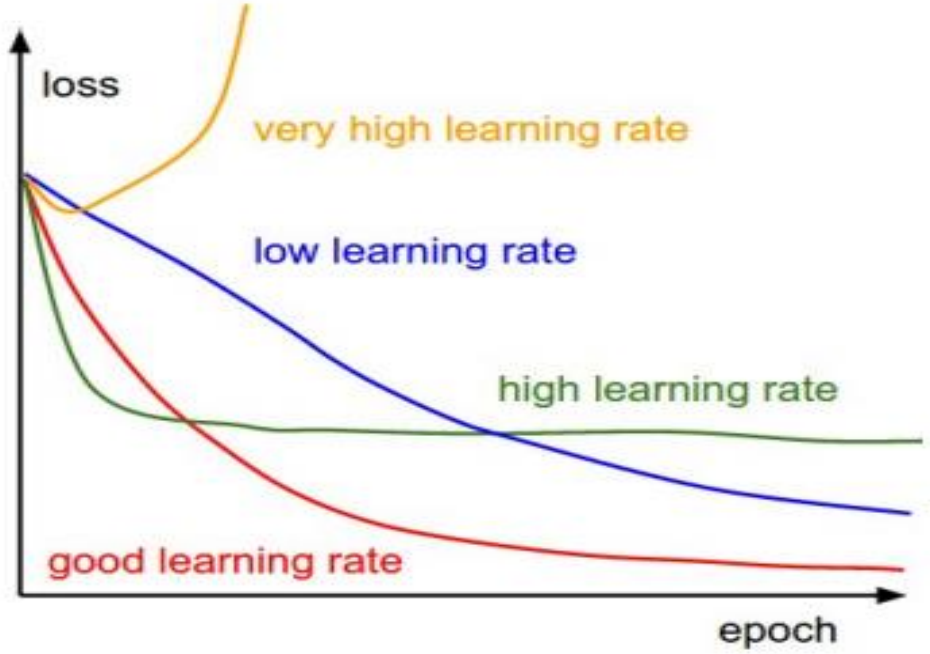


딱 봐도 더 잘 갈 수 있는데
훨씬 더 헤매면서 간다.



훑기도 잘 훑으면서,
좀 더 휙휙 **더 좋은 방향**으로 갈 순 없을까?

Learning rate도 문제가 된다!



보폭이 너무 작으면 오래 헤매고

보폭이 너무 크면, 최적해를 못 찾는다

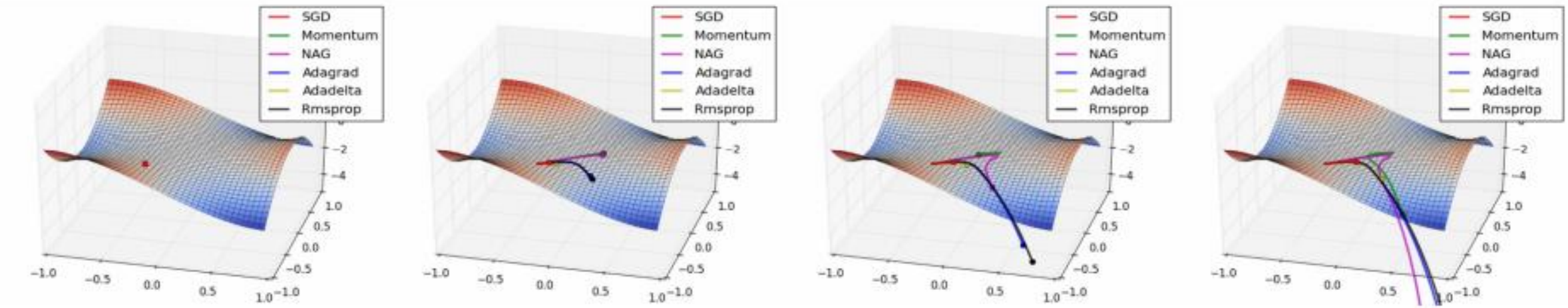
$-\gamma\nabla F(\mathbf{a^n})$ 산을 잘 타고 내려오는 것은

$\nabla F(\mathbf{a^n})$ 어느 방향으로 발을 디딜지

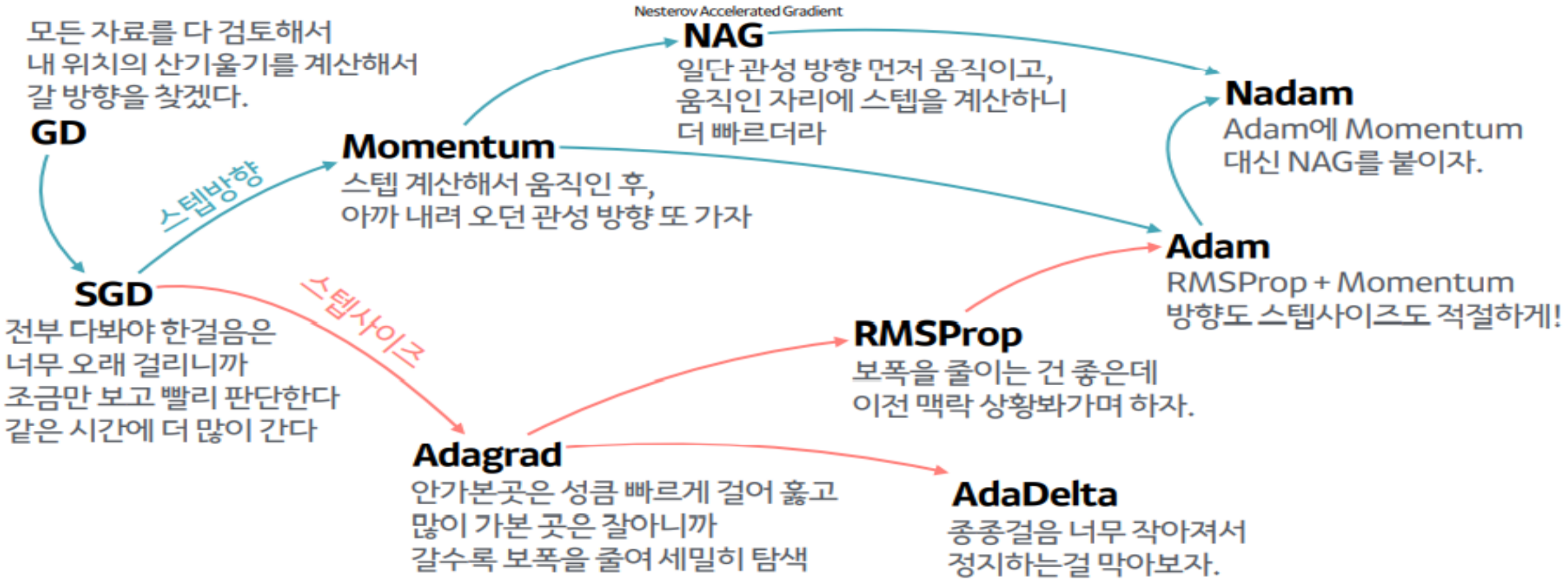
γ 얼마 보폭으로 발을 디딜지

두가지를 잘잡아야 빠르게 타고 내려온다.

그래서 SGD를 개선한 여러가지 방법이 존재!



산 내려오는 작은 오솔길 찾기(Optimizer)의 발달 계보



Optimization 변경!

```
model = Net().to(DEVICE)
optimizer = torch.optim.Adam(model.parameters(), lr = 0.01)
criterion = nn.CrossEntropyLoss()
```

SGD를

Adam으로 변경!

```
[EPOCH: 1],    Train Loss: 0.4864,    Train Accuracy: 82.29 %,    Val Loss: 0.3932,    Val Accuracy: 85.87 %
[EPOCH: 2],    Train Loss: 0.3715,    Train Accuracy: 86.52 %,    Val Loss: 0.4424,    Val Accuracy: 84.60 %
[EPOCH: 3],    Train Loss: 0.3308,    Train Accuracy: 87.68 %,    Val Loss: 0.3646,    Val Accuracy: 86.71 %
[EPOCH: 4],    Train Loss: 0.3033,    Train Accuracy: 88.66 %,    Val Loss: 0.3323,    Val Accuracy: 87.76 %
[EPOCH: 5],    Train Loss: 0.2774,    Train Accuracy: 89.58 %,    Val Loss: 0.3814,    Val Accuracy: 86.24 %
[EPOCH: 6],    Train Loss: 0.2629,    Train Accuracy: 90.11 %,    Val Loss: 0.3192,    Val Accuracy: 88.35 %
[EPOCH: 7],    Train Loss: 0.2456,    Train Accuracy: 90.72 %,    Val Loss: 0.3455,    Val Accuracy: 87.56 %
[EPOCH: 8],    Train Loss: 0.2307,    Train Accuracy: 91.37 %,    Val Loss: 0.3160,    Val Accuracy: 88.91 %
[EPOCH: 9],    Train Loss: 0.2179,    Train Accuracy: 91.83 %,    Val Loss: 0.3178,    Val Accuracy: 89.10 %
[EPOCH: 10],   Train Loss: 0.2088,    Train Accuracy: 92.11 %,    Val Loss: 0.3545,    Val Accuracy: 87.97 %
[EPOCH: 11],   Train Loss: 0.1932,    Train Accuracy: 92.59 %,    Val Loss: 0.3147,    Val Accuracy: 90.02 %
[EPOCH: 12],   Train Loss: 0.1854,    Train Accuracy: 92.98 %,    Val Loss: 0.3436,    Val Accuracy: 88.80 %
[EPOCH: 13],   Train Loss: 0.1710,    Train Accuracy: 93.43 %,    Val Loss: 0.3425,    Val Accuracy: 88.89 %
[EPOCH: 14],   Train Loss: 0.1656,    Train Accuracy: 93.77 %,    Val Loss: 0.3263,    Val Accuracy: 89.76 %
[EPOCH: 15],   Train Loss: 0.1547,    Train Accuracy: 94.03 %,    Val Loss: 0.3266,    Val Accuracy: 89.48 %
```

성능 향상!

06. Dropout & Regularization

Unit 06 | Dropout & Regularization

열심히 뉴럴넷에게 고양이  를 가르쳤더니..

뚱뚱하니까 고양이 아님

갈색이니까 고양이 아님

귀처졌으니까 고양이 아님

융통성이라곤 눈꼽 만큼도 없다! **Overfitting**

뉴럴넷의 융통성을 기르는 방법!

Dropout!

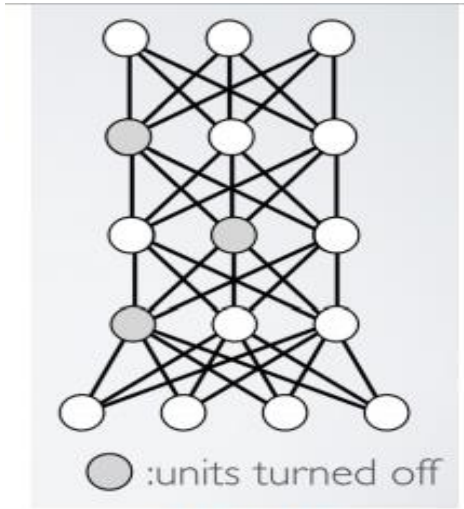
Dropout

학습 시킬 때,

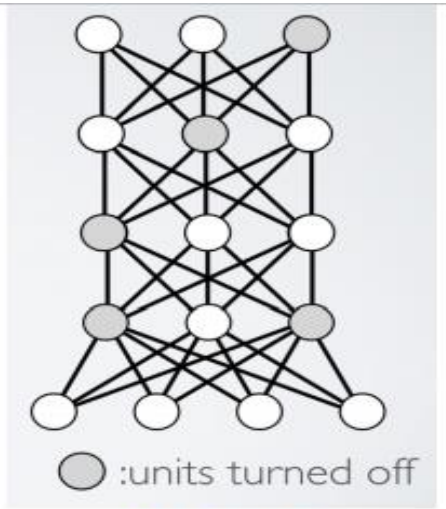
일부러 정보를 누락시키거나

중간 중간 노드를 끄는 것!

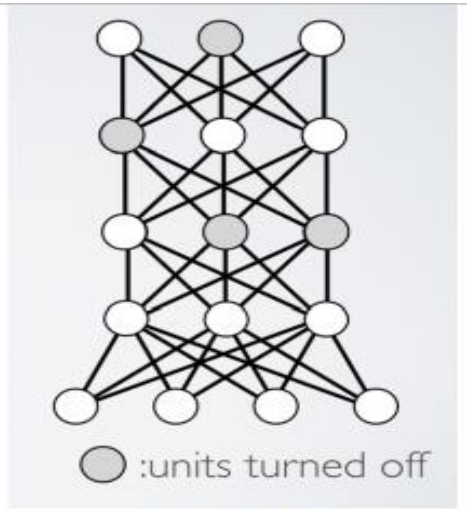
Dropout



얼굴위주

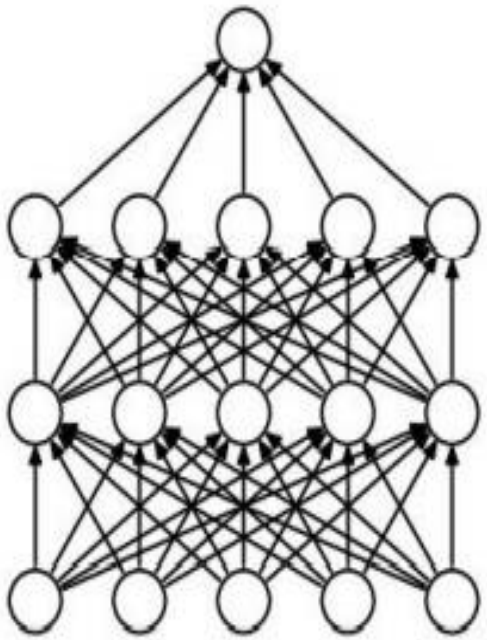


색지우고

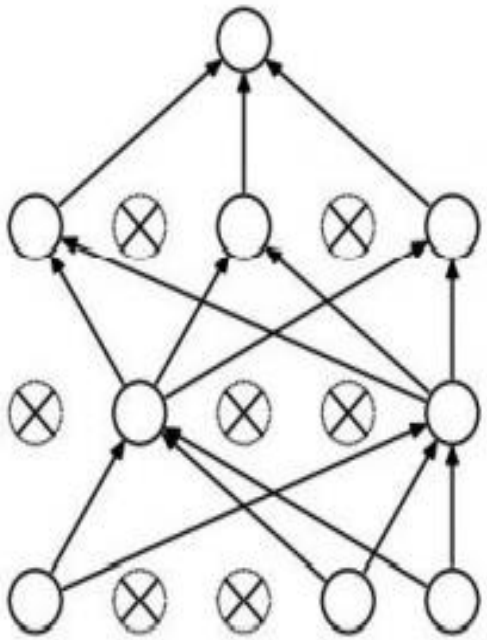


귀 빼고

Dropout



(a) Standard Neural Net



(b) After applying dropout.

Dropout으로 일부에 집착 X

중요한 요소를 스스로 학습

장점: 학습 시 weight 동조현상을 방지

단점: 매번 무작위로 선택하기 때문에 학습시간 증가

weight 동조현상 – weight가 서로 동일한 특징을 추출하는 것

Unit 06 | Dropout & Regularization

DropOut 추가!

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.linear1 = nn.Sequential(
            nn.Linear(28 * 28, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(0.2)
        )
        self.linear2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(0.2)
        )
        self.linear3 = nn.Sequential(
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(0.2)
        )
        self.linear4 = nn.Sequential(
            nn.Linear(128, 10)
        )
```

과적합 방지!

[EPOCH: 1],	Train Loss: 0.5282,	Train Accuracy: 80.74 %,	Val Loss: 0.3850,	Val Accuracy: 85.78 %
[EPOCH: 2],	Train Loss: 0.4143,	Train Accuracy: 85.00 %,	Val Loss: 0.3645,	Val Accuracy: 86.31 %
[EPOCH: 3],	Train Loss: 0.3743,	Train Accuracy: 86.35 %,	Val Loss: 0.3596,	Val Accuracy: 86.80 %
[EPOCH: 4],	Train Loss: 0.3480,	Train Accuracy: 87.31 %,	Val Loss: 0.3333,	Val Accuracy: 87.71 %
[EPOCH: 5],	Train Loss: 0.3228,	Train Accuracy: 88.03 %,	Val Loss: 0.3286,	Val Accuracy: 88.13 %
[EPOCH: 6],	Train Loss: 0.3105,	Train Accuracy: 88.60 %,	Val Loss: 0.3033,	Val Accuracy: 88.97 %
[EPOCH: 7],	Train Loss: 0.2946,	Train Accuracy: 89.04 %,	Val Loss: 0.3140,	Val Accuracy: 88.32 %
[EPOCH: 8],	Train Loss: 0.2873,	Train Accuracy: 89.38 %,	Val Loss: 0.2897,	Val Accuracy: 89.45 %
[EPOCH: 9],	Train Loss: 0.2716,	Train Accuracy: 89.72 %,	Val Loss: 0.2934,	Val Accuracy: 89.80 %
[EPOCH: 10],	Train Loss: 0.2616,	Train Accuracy: 90.18 %,	Val Loss: 0.3002,	Val Accuracy: 89.03 %
[EPOCH: 11],	Train Loss: 0.2516,	Train Accuracy: 90.66 %,	Val Loss: 0.2987,	Val Accuracy: 89.24 %
[EPOCH: 12],	Train Loss: 0.2451,	Train Accuracy: 90.86 %,	Val Loss: 0.2910,	Val Accuracy: 89.80 %
[EPOCH: 13],	Train Loss: 0.2354,	Train Accuracy: 91.22 %,	Val Loss: 0.2777,	Val Accuracy: 90.13 %
[EPOCH: 14],	Train Loss: 0.2319,	Train Accuracy: 91.30 %,	Val Loss: 0.3125,	Val Accuracy: 89.19 %
[EPOCH: 15],	Train Loss: 0.2217,	Train Accuracy: 91.76 %,	Val Loss: 0.3144,	Val Accuracy: 88.94 %

성능 향상!

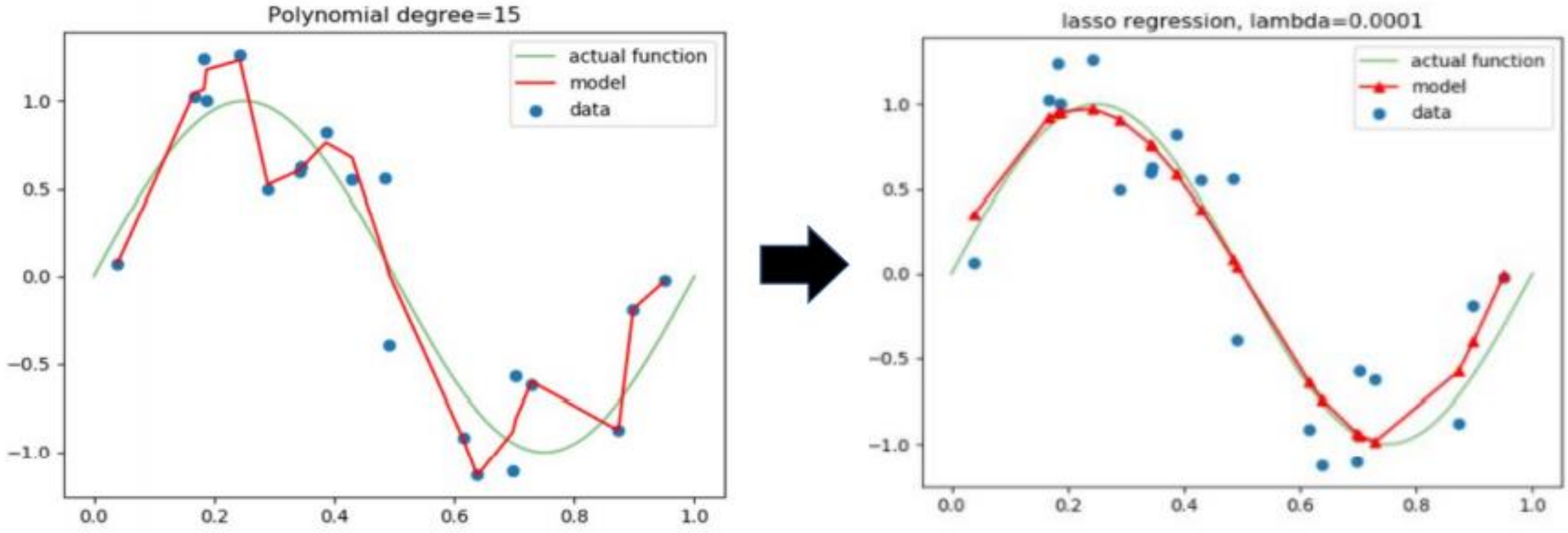
학습시에 특정 가중치 값들이 커지게 된다면??

Overfitting이 될 수 있다!

특정 가중치가 과도하게 커지지 않게 도와주는 것이!

Regularization!

Regularization



특정 가중치가 과도하게 커지지 않게 하여 잘 fitting 되게 도와주는 것!

L1 Regularization (Lasso)

$$Cost = \frac{1}{n}\sum_{i=1}^{n}\{\text{기존 } cost\ function + \frac{\lambda}{2}|w|\}$$

L2 Regularization (Ridge)

$$Cost = \frac{1}{n}\sum_{i=1}^{n}\{\text{기존 } cost\ function + \frac{\lambda}{2}|w|^2\}$$

L1 VS L2

$$RSS = minimize_{\beta}\left\{\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2\right\}$$

$$subject\ to \sum_{j=1}^{p}|\beta_j| \le s\ when\ L1, \sum_{j=1}^{p}\beta_j^2 \le s\ when\ L2$$

제약조건 하에 Residual Sum of Square을 최소화

L1 VS L2

$$subject\ to \sum_{j=1}^{p} |\beta_j| \leq s\ when\ L1, \sum_{j=1}^{p} \beta_j^2 \leq s\ when\ L2$$

변수가 2개인 경우 Lasso는 $|\beta_1| + |\beta_2| \leq s$

에서 RSS를 최소화하는 계수를 찾는다.

변수가 2개인 경우 Ridge는 $\beta_1^2 + \beta_2^2 \leq s$

에서 RSS를 최소화하는 계수를 찾는다.

L1 VS L2

유의미한 변수가 적을 때는 Lasso, 아닌 경우 Ridge가 좋음

Lasso는 중요하지 않은 변수의 계수를 0으로 만듬,

Ridge는 중요하지 않은 변수라도 어떻게든 사용함

따라서 Lasso는 Ridge보다 해석력이 더 좋다고 할 수 있지만,

정보손실과 정확도 하락의 문제가 존재!

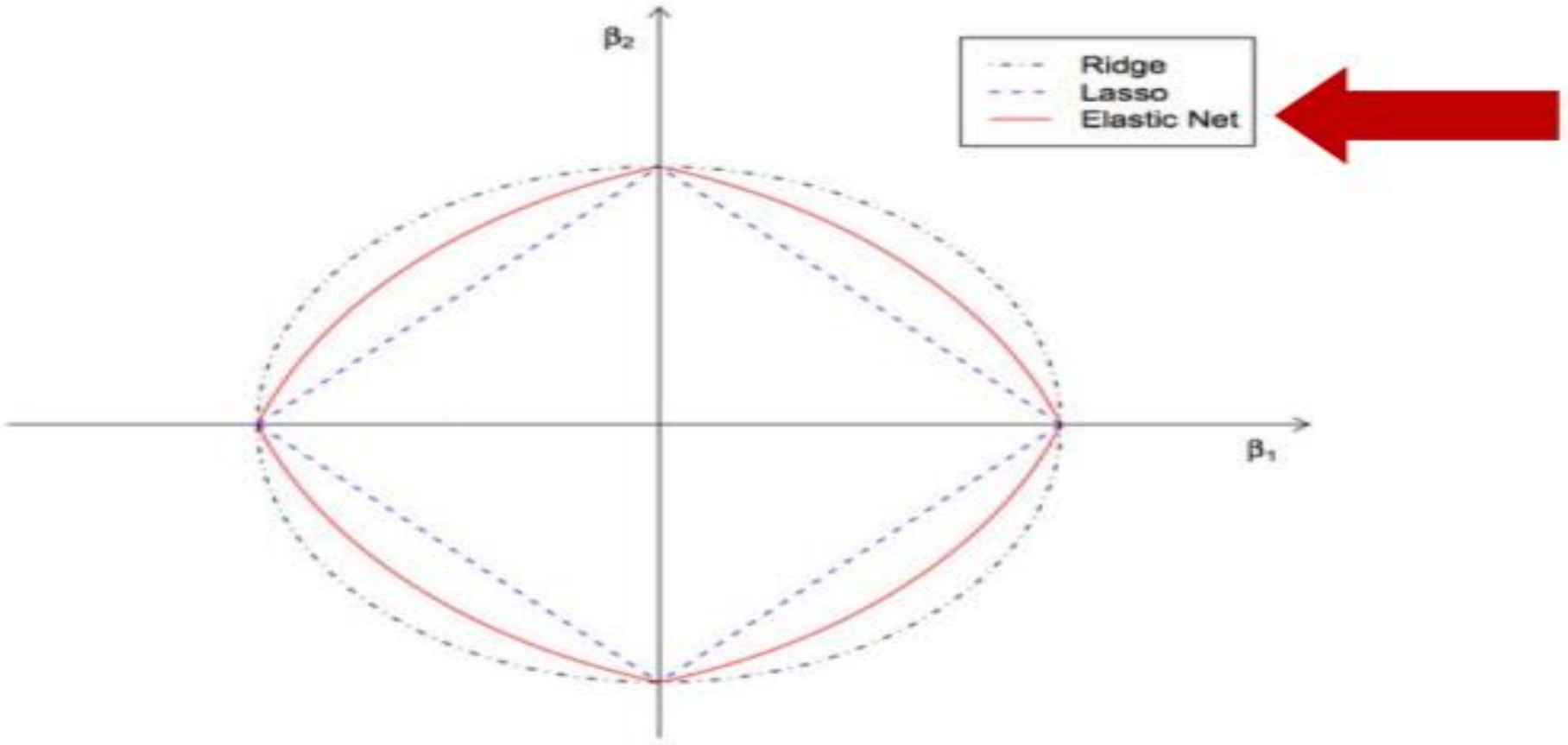
Elastic Net (L1 + L2)

$$RSS(\beta) + \lambda_1 \sum_{j=1}^{p} \beta_j^2 + \lambda_2 \sum_{j=1}^{p} |\beta_j|$$

큰 DataSet에서 Good!

변수 수도 줄이고 Variance도 줄이고 싶을 때 사용

Elastic Net (L1 + L2)



Unit 06 | Dropout & Regularization

L2 규제 추가!

```
model = Net().to(DEVICE)
optimizer = torch.optim.Adam(model.parameters(), lr = 0.01, weight_decay = 0.001)
criterion = nn.CrossEntropyLoss()
```

```
[EPOCH: 1],    Train Loss: 0.6341,    Train Accuracy: 77.47 %,    Val Loss: 0.5979,    Val Accuracy: 77.86 %
[EPOCH: 2],    Train Loss: 0.6045,    Train Accuracy: 78.64 %,    Val Loss: 0.6358,    Val Accuracy: 76.29 %
[EPOCH: 3],    Train Loss: 0.5973,    Train Accuracy: 78.95 %,    Val Loss: 0.5553,    Val Accuracy: 79.46 %
[EPOCH: 4],    Train Loss: 0.5959,    Train Accuracy: 79.01 %,    Val Loss: 0.6302,    Val Accuracy: 76.97 %
[EPOCH: 5],    Train Loss: 0.5900,    Train Accuracy: 79.09 %,    Val Loss: 0.5921,    Val Accuracy: 78.33 %
[EPOCH: 6],    Train Loss: 0.5957,    Train Accuracy: 79.11 %,    Val Loss: 0.5394,    Val Accuracy: 81.40 %
[EPOCH: 7],    Train Loss: 0.5903,    Train Accuracy: 79.29 %,    Val Loss: 0.5371,    Val Accuracy: 81.35 %
[EPOCH: 8],    Train Loss: 0.5881,    Train Accuracy: 79.51 %,    Val Loss: 0.6010,    Val Accuracy: 77.45 %
[EPOCH: 9],    Train Loss: 0.5941,    Train Accuracy: 79.13 %,    Val Loss: 0.5868,    Val Accuracy: 78.76 %
[EPOCH: 10],    Train Loss: 0.5847,    Train Accuracy: 79.25 %,    Val Loss: 0.5739,    Val Accuracy: 78.31 %
[EPOCH: 11],    Train Loss: 0.5837,    Train Accuracy: 79.60 %,    Val Loss: 0.5400,    Val Accuracy: 81.01 %
[EPOCH: 12],    Train Loss: 0.5864,    Train Accuracy: 79.44 %,    Val Loss: 0.6285,    Val Accuracy: 77.84 %
[EPOCH: 13],    Train Loss: 0.5836,    Train Accuracy: 79.73 %,    Val Loss: 0.5440,    Val Accuracy: 80.44 %
[EPOCH: 14],    Train Loss: 0.5817,    Train Accuracy: 79.44 %,    Val Loss: 0.5376,    Val Accuracy: 79.73 %
[EPOCH: 15],    Train Loss: 0.5820,    Train Accuracy: 79.54 %,    Val Loss: 0.5022,    Val Accuracy: 82.71 %
```

규제가 강해져 전체적인 성능하락
규제의 정도 또한 하이퍼 파라미터!

종합

- **Activation Function** — Sigmoid, Tanh ,ReLU ....
- **Weigh Initialization** — Xavier, He ...
- **Batch Normalization** — Internal Covariate Shift 해결
- **Optimization** — SGD, RMSprop, Adam ...
- **Dropout** — 동조현상 방지
- **Regularization** — L1, L2 ...

지금까지 배운 모든 내용 Code 실습

Week7_NN심화_BaseLine_Model.ipynb

코드를 Colab에서 실행해주세요

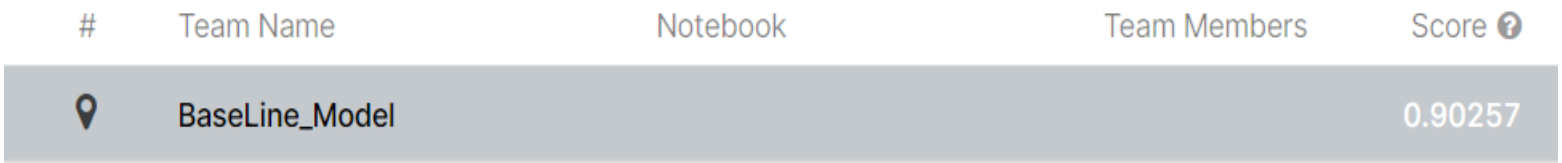
들어주셔서 감사합니다.

Assignment : 캐글 경진대회 참여!

캐글 경진 대회에 참여하여 가장 좋은 Model 을 만들어 보세요!

채점 기준은 리더보드 + 다양한 프레임 워크 함수 사용 + 모델의 결과에 대한 설명 입니다!

BaseLine Model은 모두 넘으셔야 합니다!!



#	Team Name	Notebook	Team Members	Score
	BaseLine_Model			0.90257

https://www.kaggle.com/t/d4342e94b4da4b1fbbd29c7968a309de

Unit | 참고 자료

- 밑바닥부터 시작하는 딥러닝 시즌1
- 백날 자습해도 이해 안 가던 딥러닝, 머리속에 인스톨 시켜드립니다. (하용호)
- 10기 박규리님 NN심화 강의 자료
- 13기 김현선님 NN심화 강의 자료
- 14기 한유진님 NN심화 강의 자료