17기 정규세션

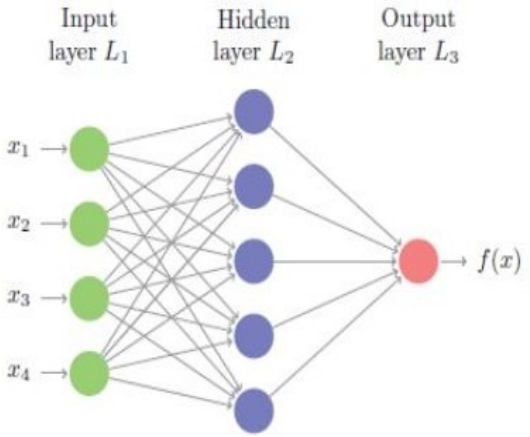
ToBig's 16기 김종우

NN 심화

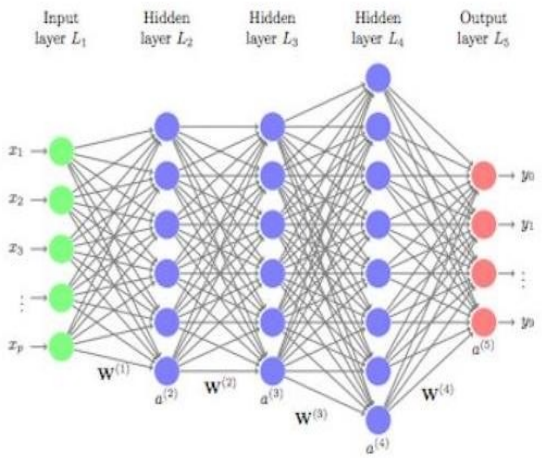
Contents

Unit 01 | Introduction

Neural Network



<Simple Neural Net>



<Deep Neural Net>

Neural Net은 Layer을 깊게 쌓을수록 성능이 올라갈 가능성이 높아진다!

Unit 01 | Introduction

Neural Network

Neural Network Layer가 너무 deep 하면 발생하는 문제들

- **Underfitting** : 학습이 잘 안돼
- **Too Slow** : 학습이 너무 느려 (학습해야 하는 Layer의 증가)
- **Overfitting** : 일반화가 힘들어 (train data에만 맞춤)

따라서 효율적, 효과적으로 Layer를 쌓아야 한다!

Unit 01 | Introduction

Neural Network

Neural Network Layer가 너무 deep 하면 발생하는 문제 해결 방법

Activation Function : Sigmoid, Tanh ,ReLU ….

Weigh Initialization : Xavier, He …

Batch Normalization : Internal Covariate Shift 해결

Optimization : SGD, RMSprop, Adam … 동조현상 방지

Dropout

Unit 01 | Introduction

Neural Network

다양한 해결 방법을 통해서 TEST(VALIDATION) ERR를 줄여 나가 보자

```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 Layer로 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 %
```

Unit 02 | Activation Function

Neural Network

Activation Function의 필요성

중간에 Activation Function X

```
1 class MLP(nn.Module):
2   def __init__(self):
3     super().__init__()
4     self.linear = nn.Sequential(
5       nn.Linear(2, 10),
6       nn.Linear(10, 1), # 10개의 노드를 가지는 1개의 은닉층 생성
7       nn.Sigmoid()
8     )
9
10  def forward(self, x):
11    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

Neural Network

Activation Function의 필요성

중간에 Activation Function O

```
1 class MLP(nn.Module):
2   def __init__(self):
3     super().__init__()
4     self.linear = nn.Sequential(
5         nn.Linear(2, 10),
6         nn.Sigmoid(),
7         nn.Linear(10, 1), # 10개의 노드를 가지는 1개의 은닉층 생성
8         nn.Sigmoid()
9     )
10
11  def forward(self, x):
12    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: 0.02195087820291519
EPOCH: 9000, LOSS: 0.016813486814489
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!

Unit 02 | Activation Function

Neural Network

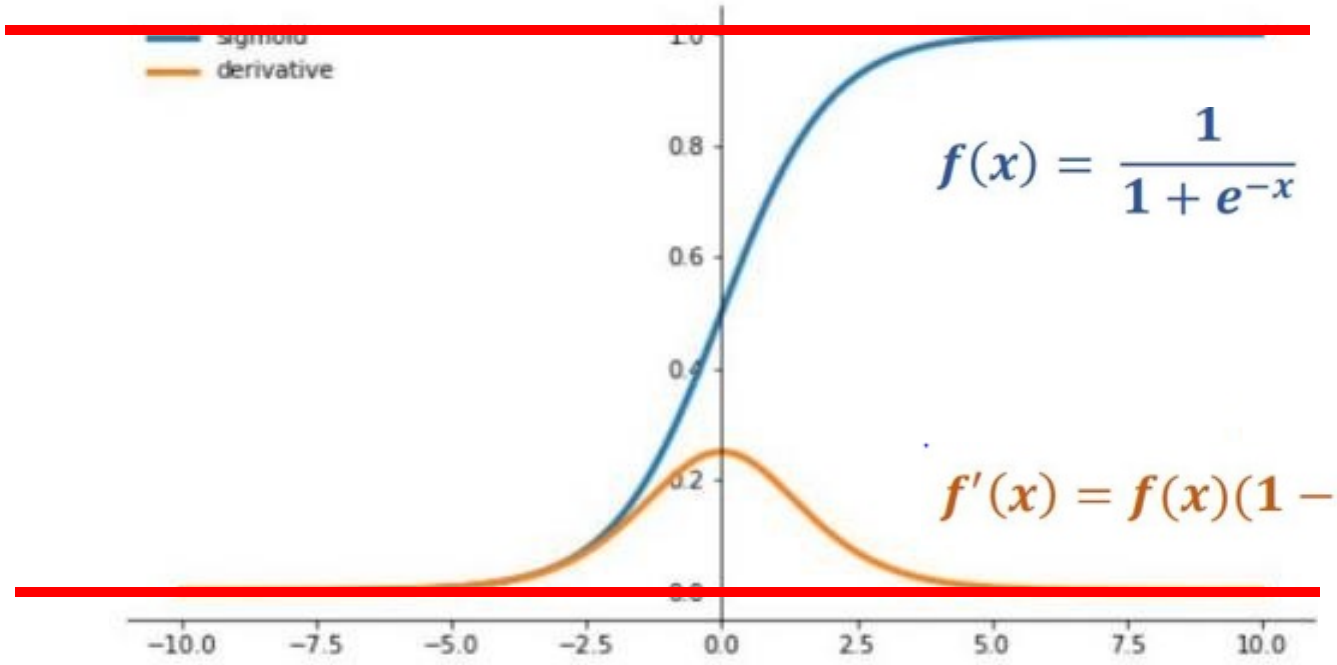
Activation Function의 종류

- **Sigmoid**
- **Tanh**
- **ReLU**
- **Leaky ReLU**

Unit 02 | Activation Function

Neural Network

Activation Function의 종류 : sigmoid



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

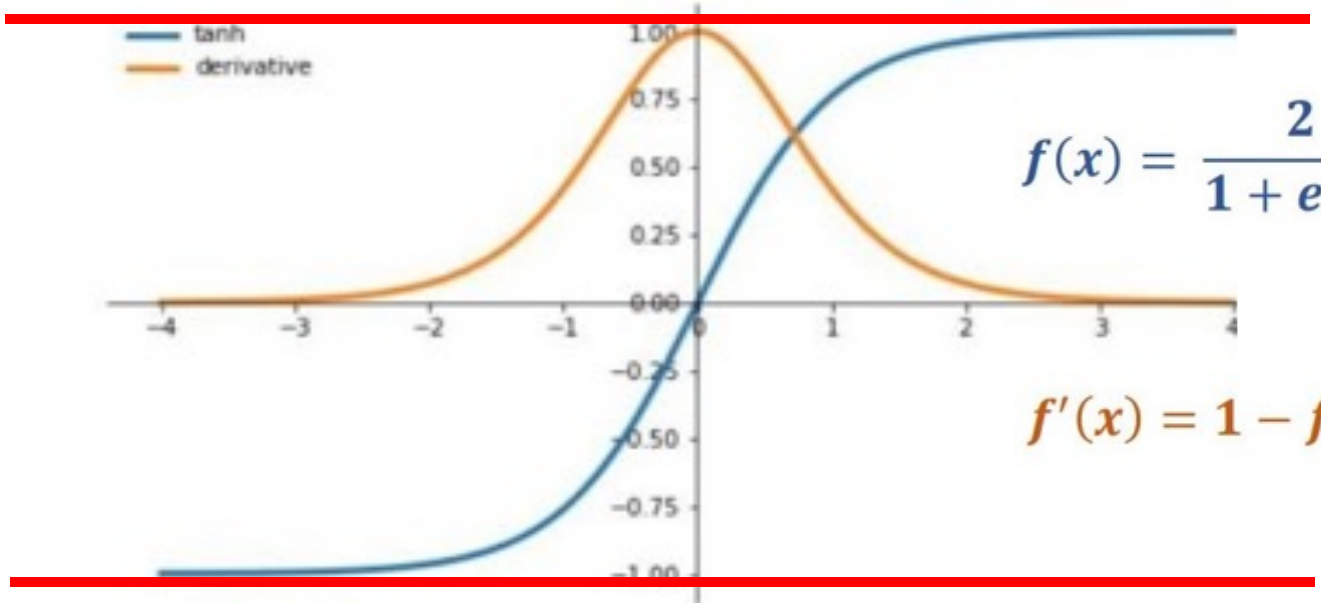
$$f'(x) = f(x)(1-f(x))$$

0~1 사이
↓
출력값 = 확률 형태

Unit 02 | Activation Function

Neural Network

Activation Function의 종류 : Tanh



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

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

-1 ~ 1 사이
↓
출력 범위 더 넓음
경사면이 더 가파름
빠르게 수렴

Unit 02 | Activation Function

Neural Network

Activation Function의 종류

그런데

Sigmoid와 Tanh은

매우 큰 문제점이 존재!

Neural Network

Activation Function의 종류

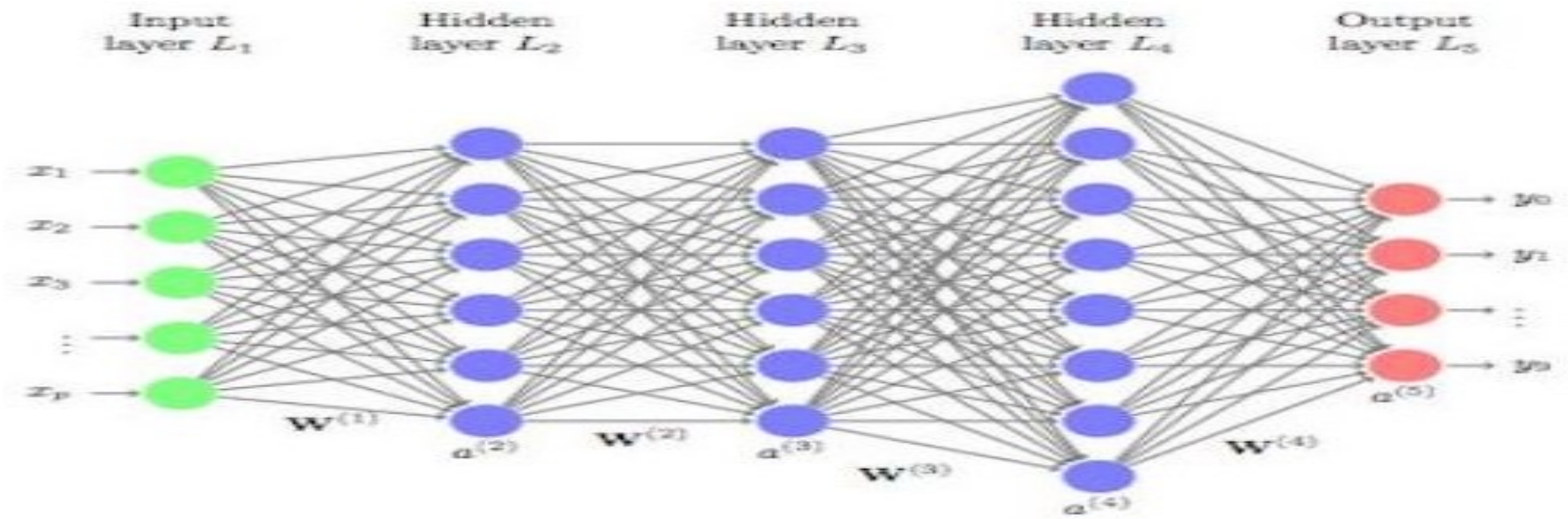
Vanishing Gradient Problem

기울기 소실 문제

Unit 02 | Activation Function

Neural Network

Activation Function의 종류 : Vanishing Gradient Problem

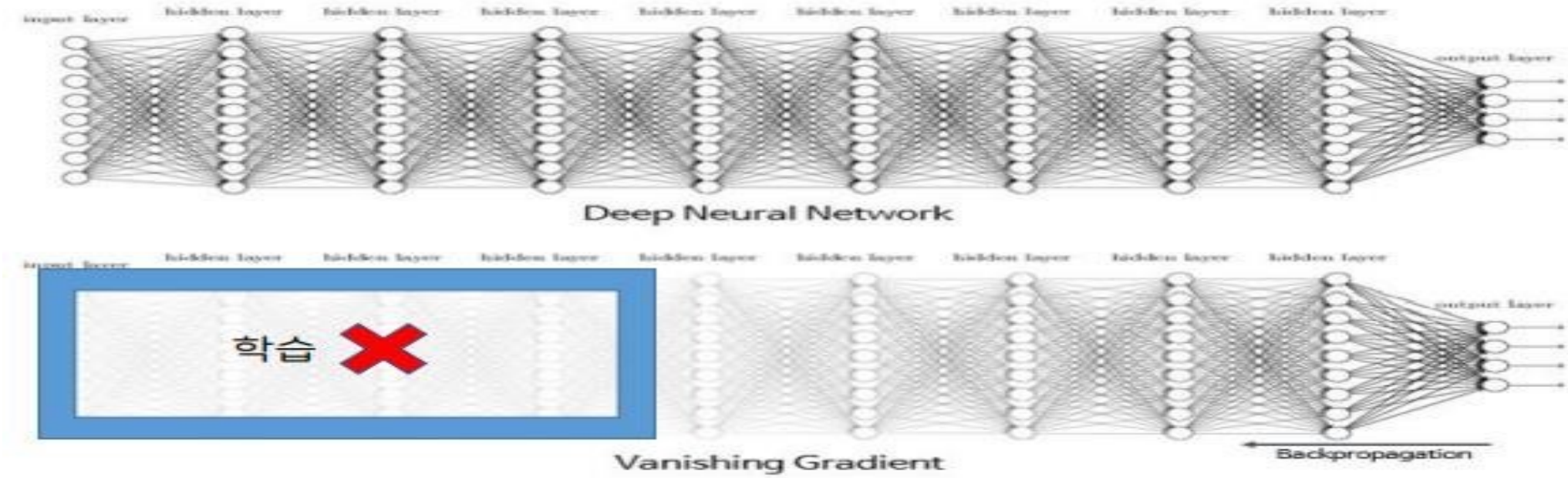


Forward Propagation -> Loss 계산 -> Back Propagation -> Update Parameters

Unit 02 | Activation Function

Neural Network

Activation Function의 종류 : Vanishing Gradient Problem



역전파 과정에서 그래디언트를 계산하면서(미분) 파라미터를 수정하는데,

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

Neural Network

Activation Function의 종류 : Vanishing Gradient Problem

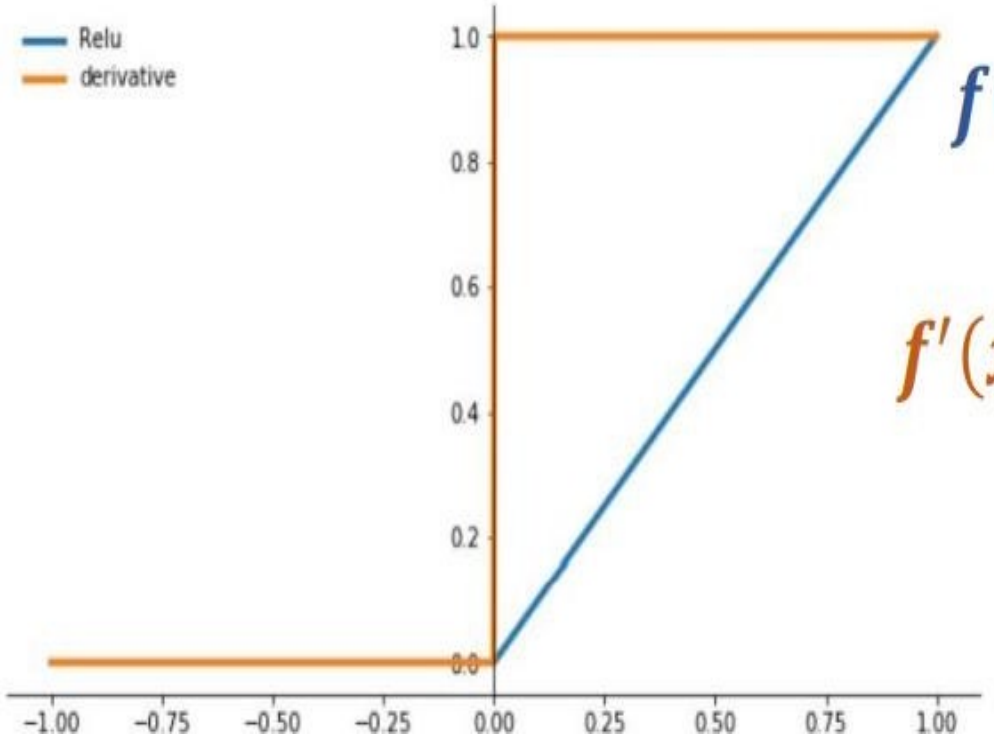
Rectified Linear Unit(ReLU)

그래서 나온 해결책!

Unit 02 | Activation Function

Neural Network

Activation Function의 종류 : ReLU



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

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

미분값이 양수인 경우 1

Vanishing Gradient 해결

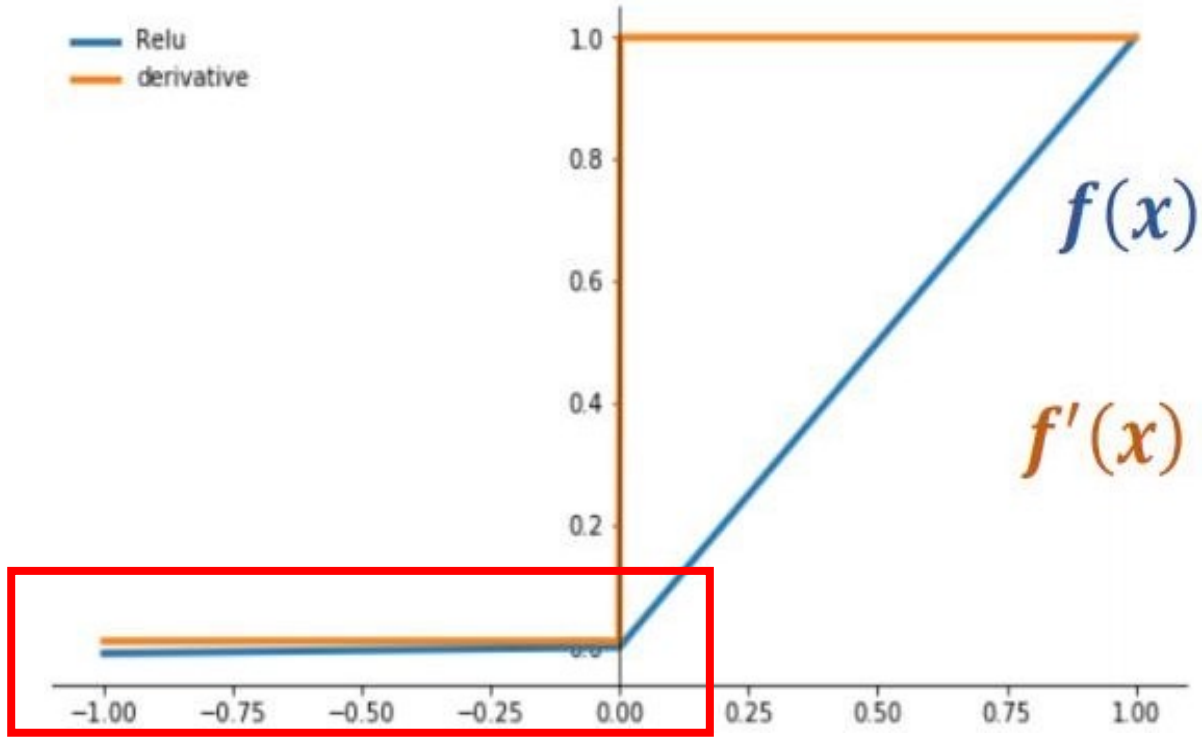
But 음수 값 무시

dying ReLU

Unit 02 | Activation Function

Neural Network

Activation Function의 종류 : Leaky ReLU



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

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

Unit 02 | Activation Function

Neural Network

Activation Function의 종류

```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()
```

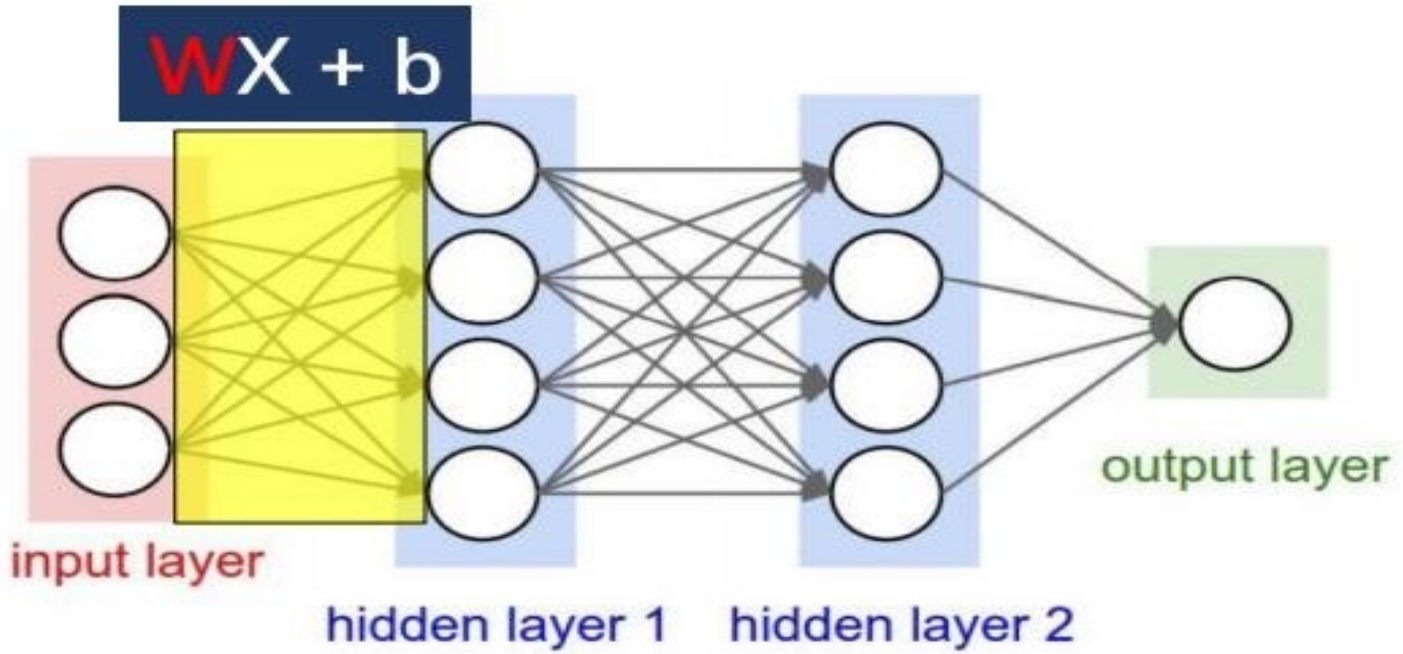
```
[EPOCH: 1],    Train Loss: 1.9380,    Train Accuracy: 33.59 %,    Val Loss: 1.2035,    Val Accuracy: 59.36 %
[EPOCH: 2],    Train Loss: 0.9337,    Train Accuracy: 64.72 %,    Val Loss: 0.7758,    Val Accuracy: 70.08 %
[EPOCH: 3],    Train Loss: 0.7257,    Train Accuracy: 73.16 %,    Val Loss: 0.6562,    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: 85.48 %
[EPOCH: 15],   Train Loss: 0.4106,    Train Accuracy: 85.55 %,    Val Loss: 0.4050,    Val Accuracy: 85.59 %
```

성능 향상!

Unit 03 | Weight initialization

Neural Network

Weight initialization



NN을 학습시킬 때 W의 초깃값을 임의로 설정해주는 것!

Unit 03 | Weight initialization

Neural Network

Weight initialization

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!

Unit 03 | Weight initialization

Neural Network

Weight initialization

초기 값을 무작위로 설정하자!

가중치가 고르게 되어버리는 상황을 막아줘,

모델이 다양성을 가지게 됨!(가중치가 다양함)

Unit 03 | Weight initialization

Neural Network

Weight initialization

초기 값을 단순히 무작위로 설정하니....

가중치가 너무 크거나 작아서 학습이 잘 안돼!

Unit 03 | Weight initialization

Neural Network

Weight initialization

어떠한 분포에서 가중치를 랜덤하게 뽑자!

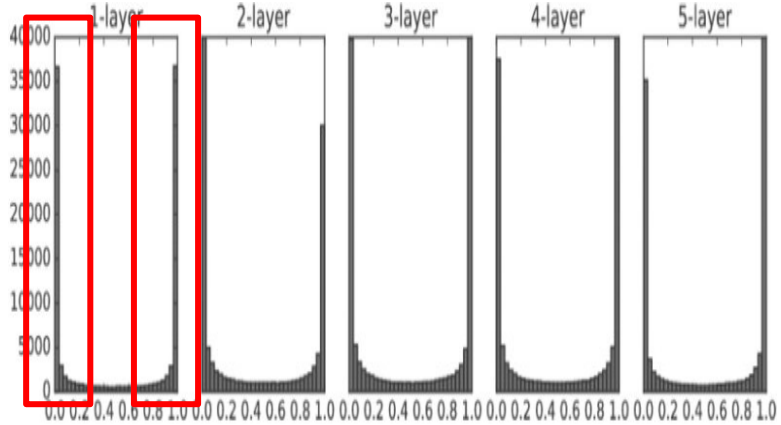
Unit 03 | Weight initialization

Neural Network

Weight initialization : 정규분포

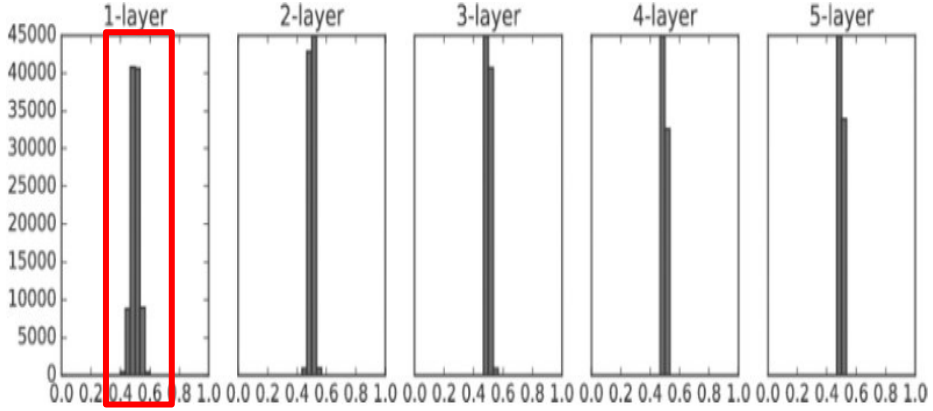
사진은 활성화 값(sigmoid)의 분포를 히스토그램으로 표현한 것

가중치의 표준편차 1



기울기 소실

가중치의 표준편차0.01



표현력 제한

Unit 03 | Weight initialization

Neural Network

Weight initialization

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

결과가 바로!
Xavier(자비에) 초깃값, He 초깃값

Unit 03 | Weight initialization

Neural Network

Weight initialization : Xavier 초깃값

Xavier Normal Initialization

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

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

Xavier Uniform Initialization

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

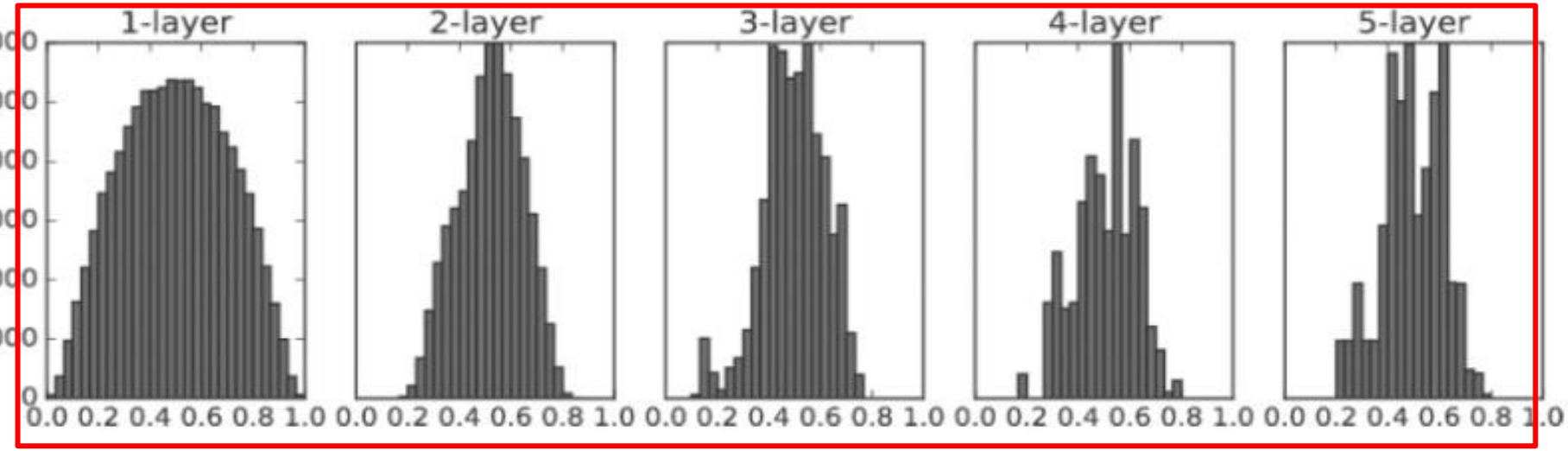
n_in = 들어오는 노드의 수 / n_out = 나가는 노드의 수

Unit 03 | Weight initialization

Neural Network

Weight initialization : Xavier 초깃값

고른 분포! == 가중치가 다양하다! == 표현력이 좋다!



Sigmoid, Tanh 와 함께 사용할 때 성능이 좋음!

Unit 03 | Weight initialization

Neural Network

Weight initialization : He(kaiming) 초깃값

He Normal Initialization

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

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

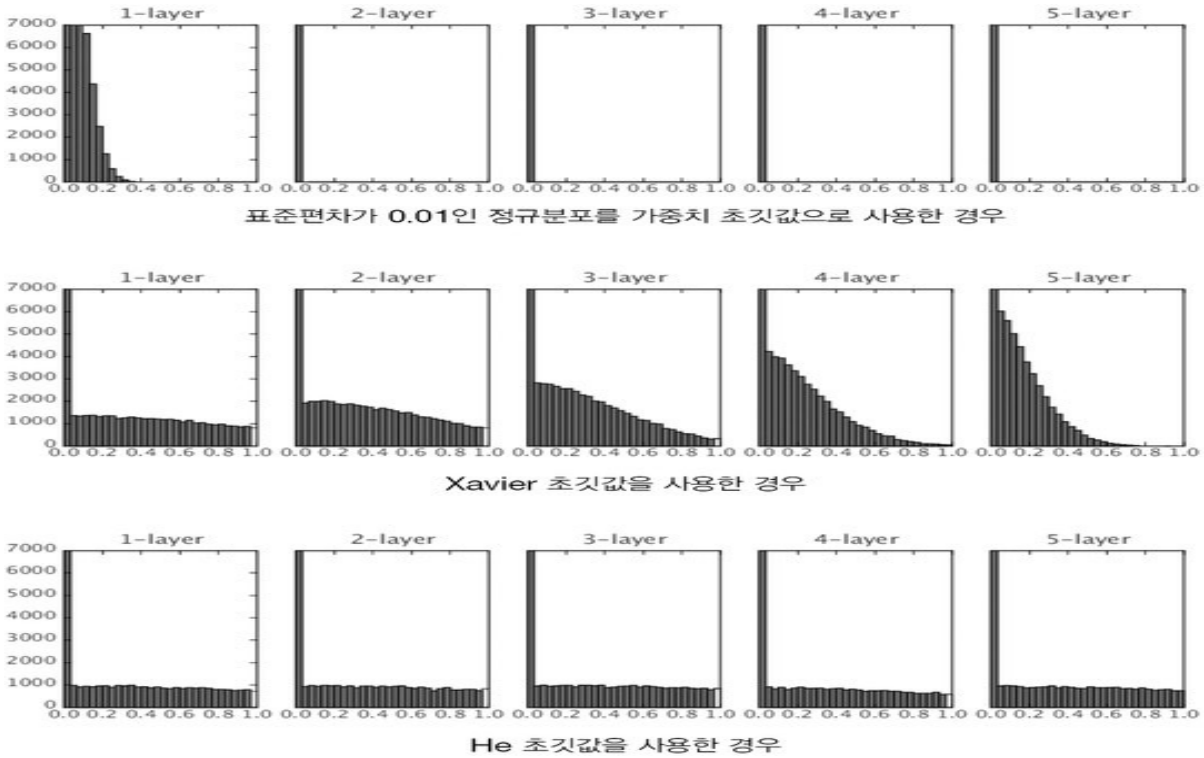
He Uniform Initialization

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

Unit 03 | Weight initialization

Neural Network

Weight initialization : He 초깃값



ReLU + 정규분포 = 기울기 소실

ReLU + Xavier = 기울기 소실

BEST!

ReLU + He = 고른 분포

Unit 03 | Weight initialization

Neural Network

Weight initialization

```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()
```

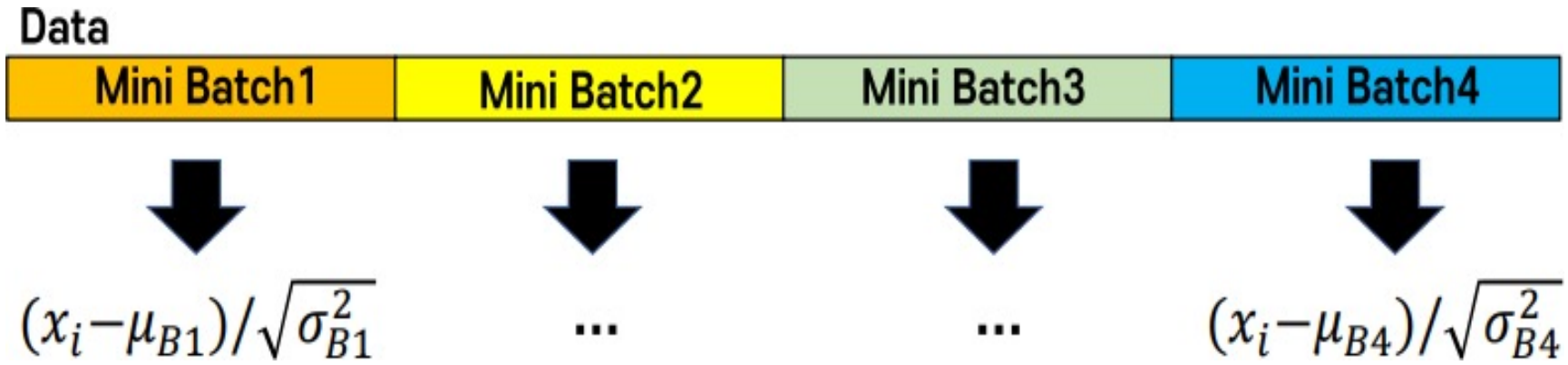
```
[EPOCH: 1],   Train Loss: 0.9834,   Train Accuracy: 67.64 %,   Val Loss: 0.6181,   Val Accuracy: 78.76 %
[EPOCH: 2],   Train Loss: 0.5687,   Train Accuracy: 80.35 %,   Val Loss: 0.5099,   Val Accuracy: 82.48 %
[EPOCH: 3],   Train Loss: 0.4964,   Train Accuracy: 82.70 %,   Val Loss: 0.4656,   Val 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: 86.17 %
[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!

Unit 04 | Batch Normalization

Neural Network

Batch Normalization

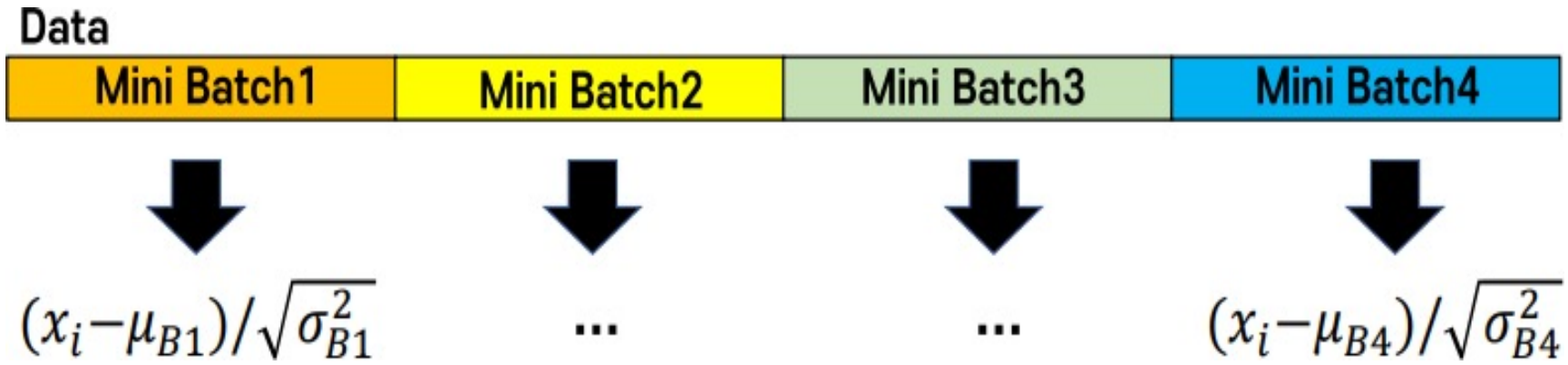


Mini Batch 단위를 Normalize 시키는 것!

Unit 04 | Batch Normalization

Neural Network

Batch Normalization



네트워크망 내부 데이터를 변경하여 안정적으로 학습 가능!

Unit 04 | Batch Normalization

Neural Network

Batch Normalization

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

- Internal Covariance Shift -

Network의 각 층이나 Activation 마다 Input의 Distribution이 달라지는 현상

Unit 04 | Batch Normalization

Neural Network

Batch Normalization

직관적으로 이야기하면

활성화 함수 통과 전에 배치 정규화를 수행함으로써 층마다 일정

하지 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!

Unit 04 | Batch Normalization

Neural Network

Batch Normalization

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!

Unit 04 | Batch Normalization

Neural Network

Batch Normalization

```
class Net
  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!

Unit 05 | Optimization

Neural Network

Optimization

기존 뉴럴넷이 가중치 parameter들을 최적화(optimize)하는 방법

Gradient Decent

Loss Function이 현 가중치에서의 기울기를 구해서

Loss를 줄이는 방향으로 업데이트

Neural Network

Optimization

현재 가진 Weight 세팅에서,

내가 가진 데이터를 다 넣으면

전체 에러가 계산된다!

Unit 05 | Optimization

Neural Network

Optimization

트레이닝 데이터가 몇 억 건이면 .....

어느 세월에 다하는가 .....

GD보다 빠른 옵티마이저는 없을까?

Neural Network

Optimization

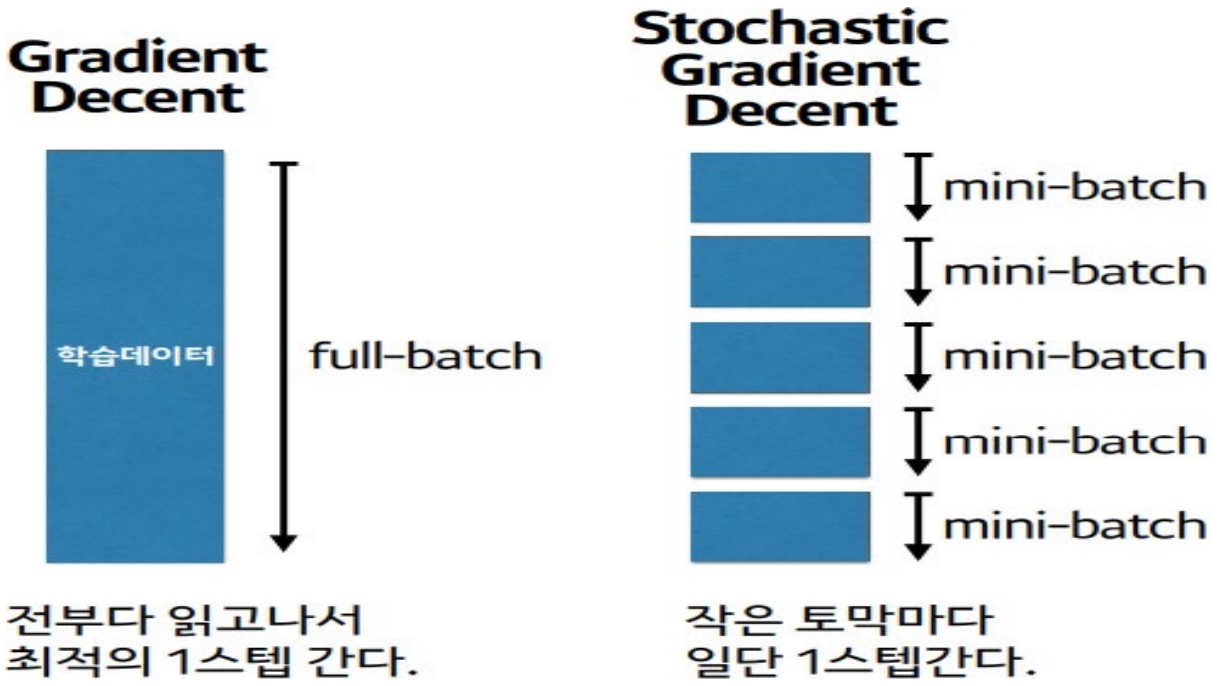
Stochastic Gradient Decent

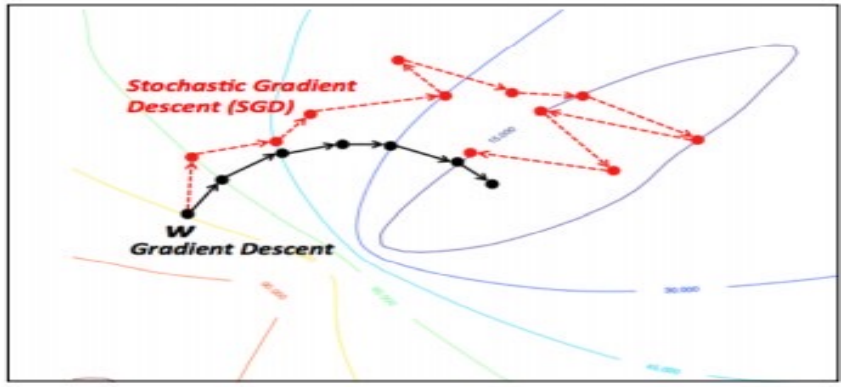
컨셉: 느린 완벽보다 조금만 훑어보고 일단 빨리 가자!

Unit 05 | Optimization

Neural Network

Optimization

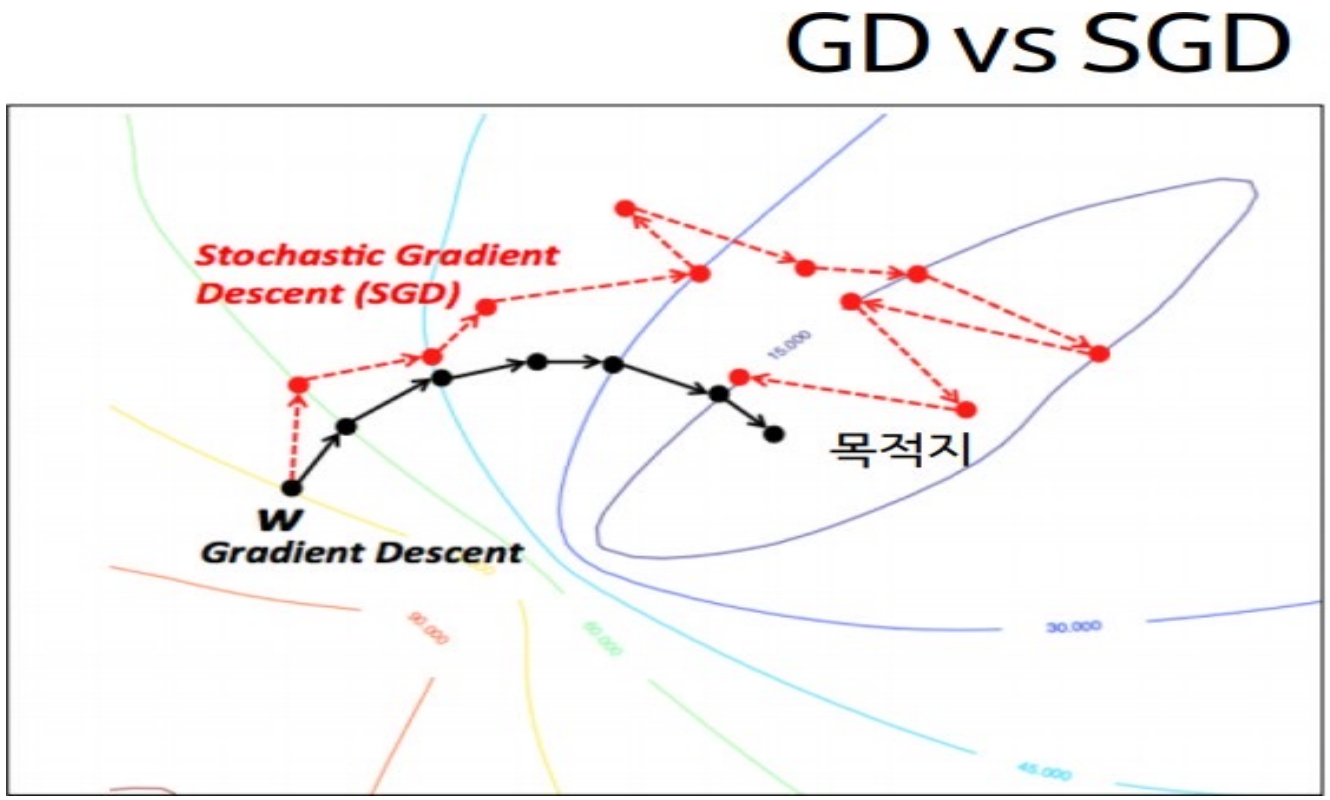




Unit 05 | Optimization

Neural Network

Optimization



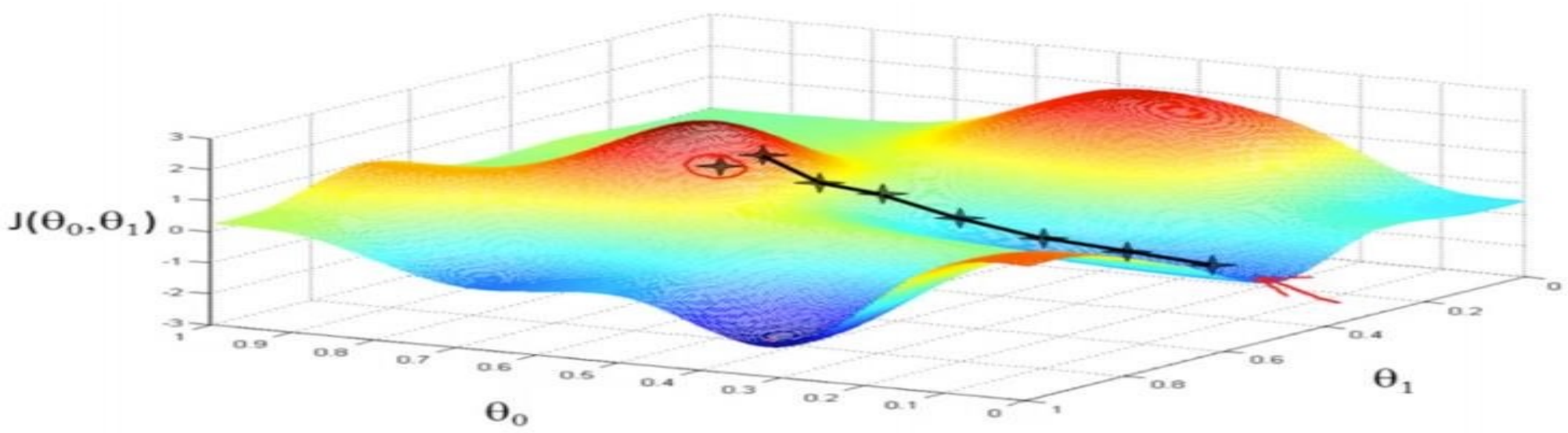
Gradient Decent
모든 걸 계산(1시간)후
최적의 한스텝
6스텝 * 1시간 = 6시간
최적인데 너무 느리다!

Stochastic Gradient Descent
일부만 검토(5분)
틀려도 일단 간다! 빠른 스텝!
11스텝 * 5분 = 55분 < 1시간
조금 헤매도 어쨌든 인근에 아주 빨리 갔다!

Neural Network

Optimization

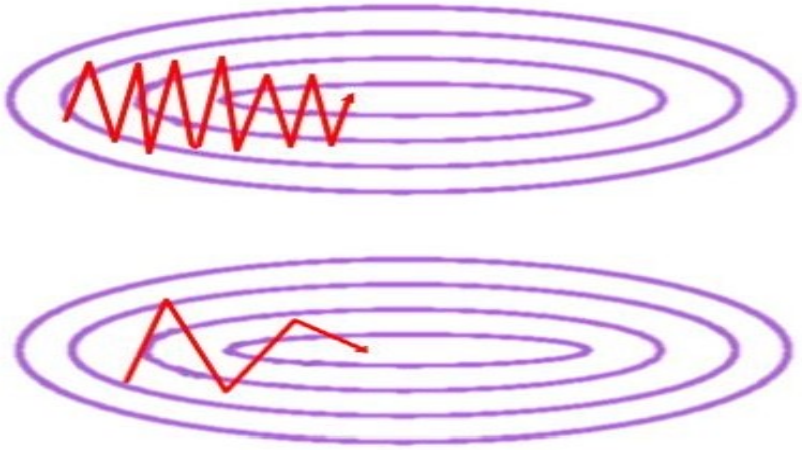
최적화는 좋은 오솔길을 찾아서 산을 내려오는 것과 매우 비슷!



Neural Network

Optimization

근데 미니 배치를 하다 보니 방향 문제가 존재!



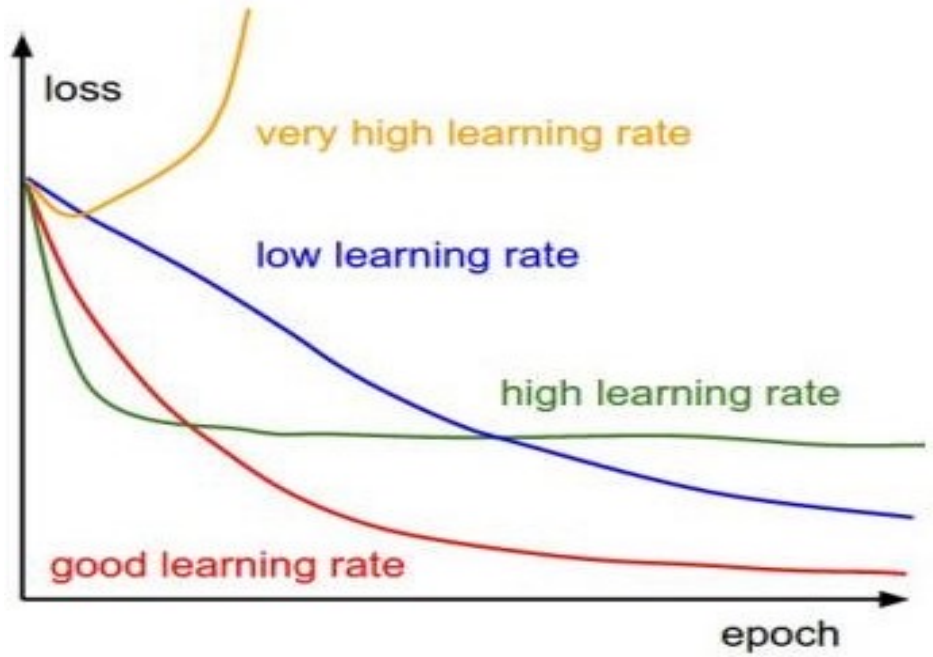
딱 봐도 더 잘 갈 수 있는데
훨씬 더 헤매면서 간다.

훑기도 잘 훑으면서,
좀 더 휙휙 더 좋은 방향으로 갈 순 없을까?

Neural Network

Optimization

Learning rate도 문제가 된다!



보폭이 너무 작으면 오래 헤매고

보폭이 너무 크면, 최적해를 못 찾는다

Neural Network

Optimization

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

$\nabla F(\mathbf{a^n})$ 어느 **방향**으로 발을 디딜지

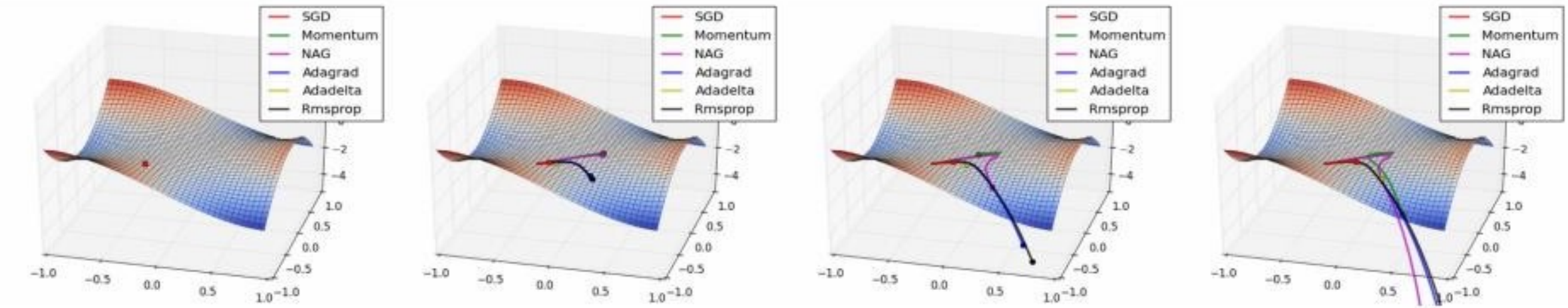
γ 얼마 **보폭**으로 발을 디딜지

두가지를 잘잡아야 빠르게 타고 내려온다.

Neural Network

Optimization

그래서 SGD를 개선한 여러가지 방법이 존재!



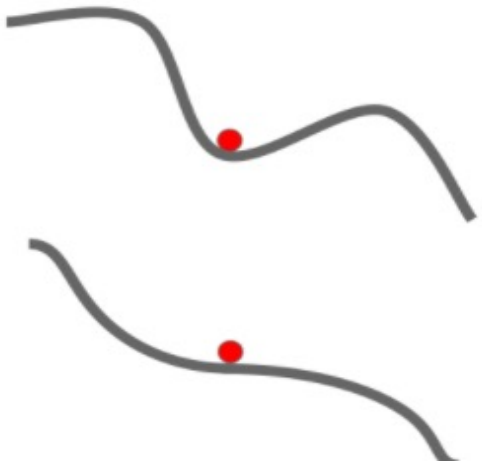
Unit 05 | Optimization

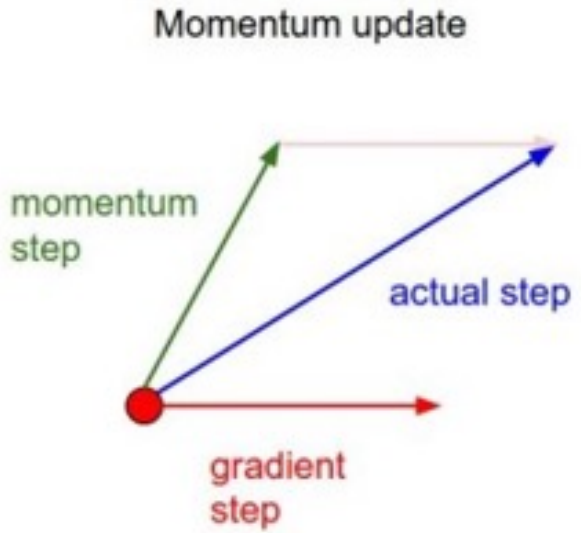
Neural Network

Optimization : momentum

Local minimum

Saddle point

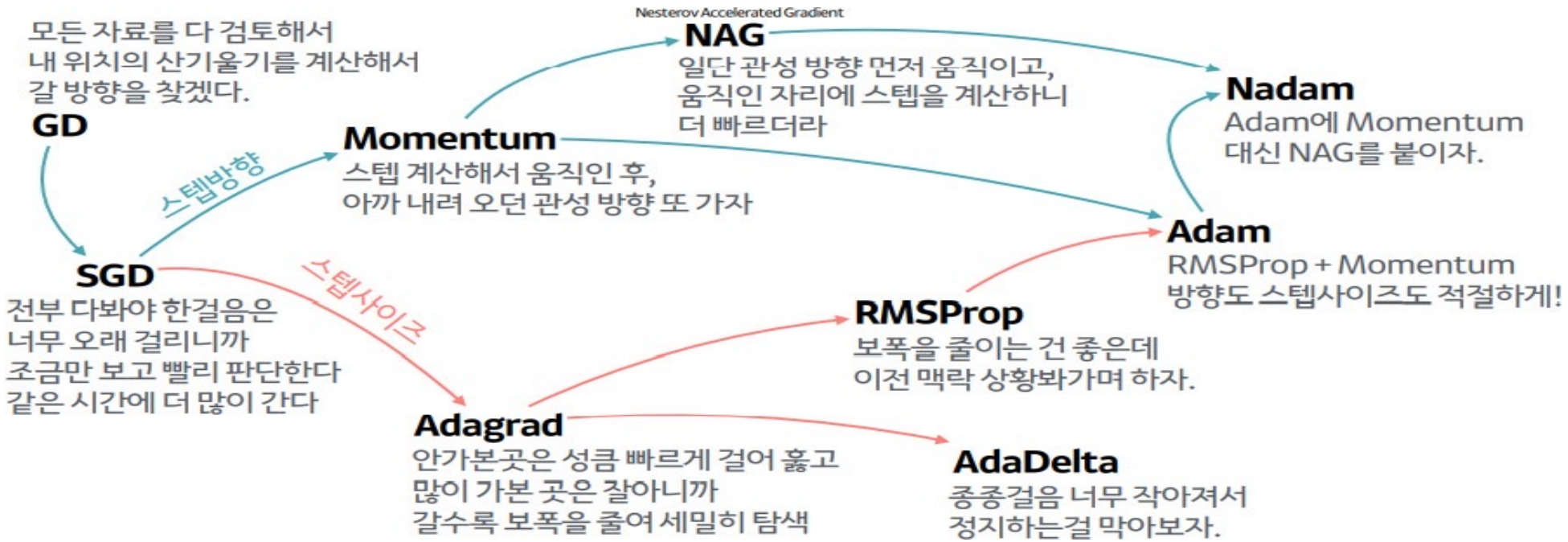




Unit 05 | Optimization

Neural Network

Optimization



Unit 05 | Optimization

Neural Network

Optimization

```
1 model = Net().to(DEVICE)
2 optimizer = torch.optim.Adam(model.parameters(), lr = 0.01)
3 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!

Unit 06 | Dropout

Neural Network

Dropout

뉴럴넷의 융통성을 기르는 방법!

Dropout!

Unit 06 | Dropout

Neural Network

Dropout

학습 시킬 때,

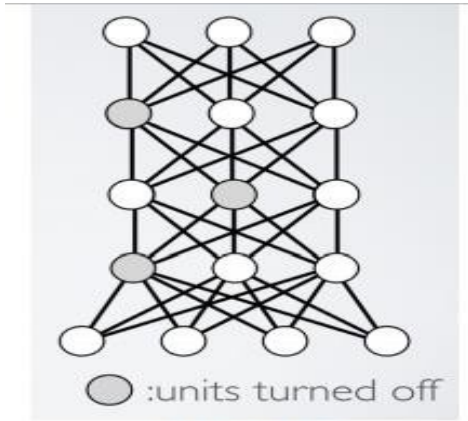
일부러 정보를 누락시키거나

중간 중간 노드를 끄는 것!

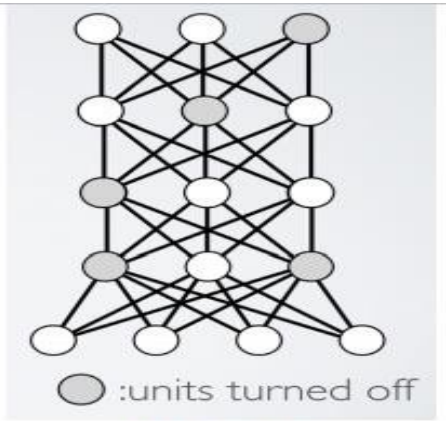
Unit 06 | Dropout

Neural Network

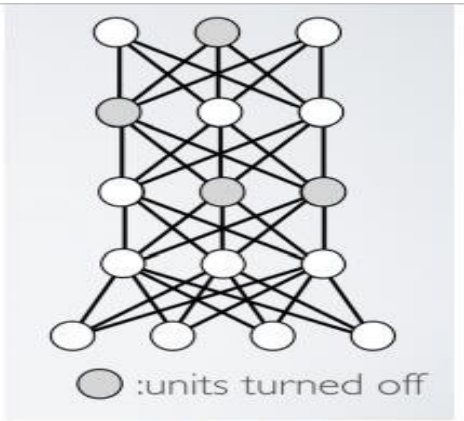
Dropout



얼굴위주



색지우고

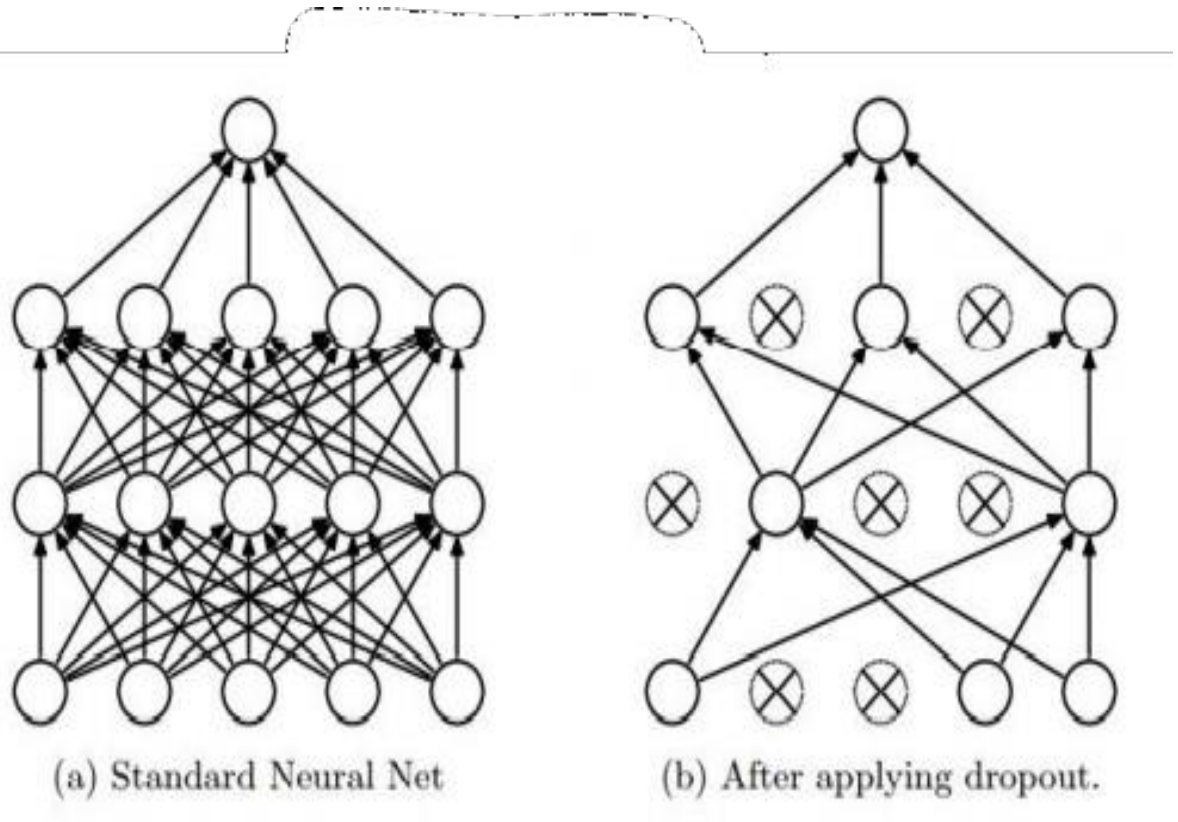


귀 빼고

Unit 06 | Dropout

Neural Network

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

Neural Network

Dropout

```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: 89.30 %
[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!

Neural Network

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/c/tobigs17-nnadv/leaderboard

들어주셔서 감사합니다.