Problems & Solutions

2017.08.26

최건호

INDEX

01

Overfitting & Underfitting

- Why?
- Data Splitting
- Regularization
- Drop out
- Data Augmentation

02

Convergence

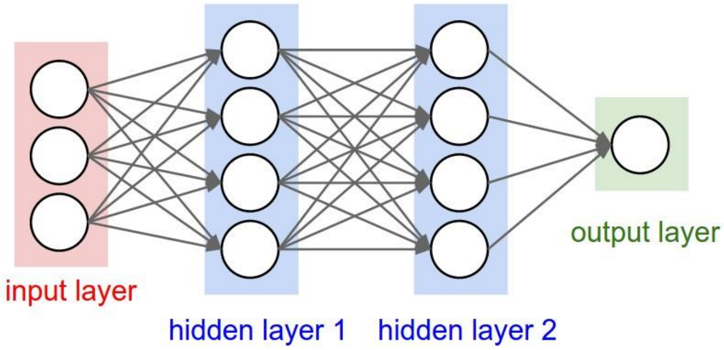
- Weight Initialization
- Data Normalization
- Batch Normalization

03

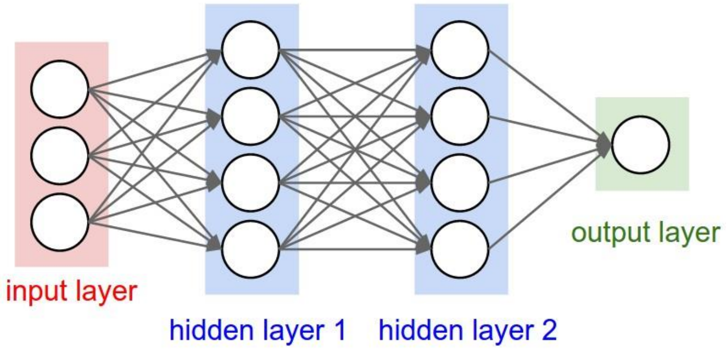
Optimization Algorithms

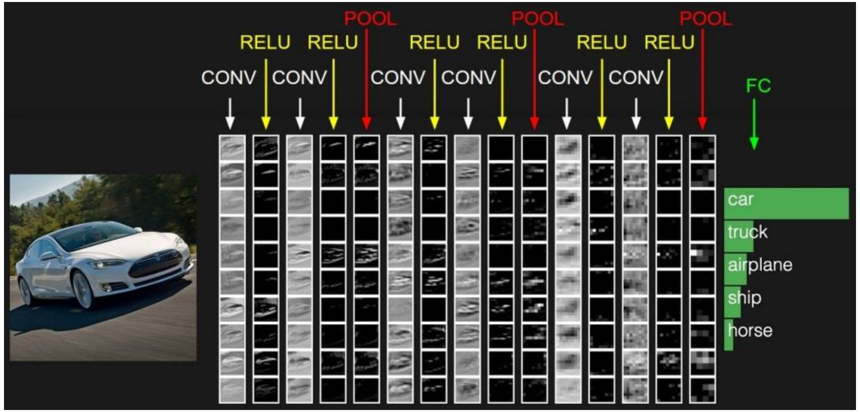
- SGD
- Momentum
- Nestrov
- Adagrad
- RMSProp
- Adam

Overfitting & Underfitting

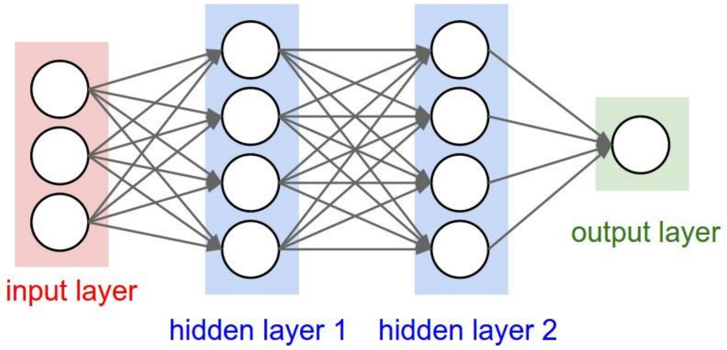


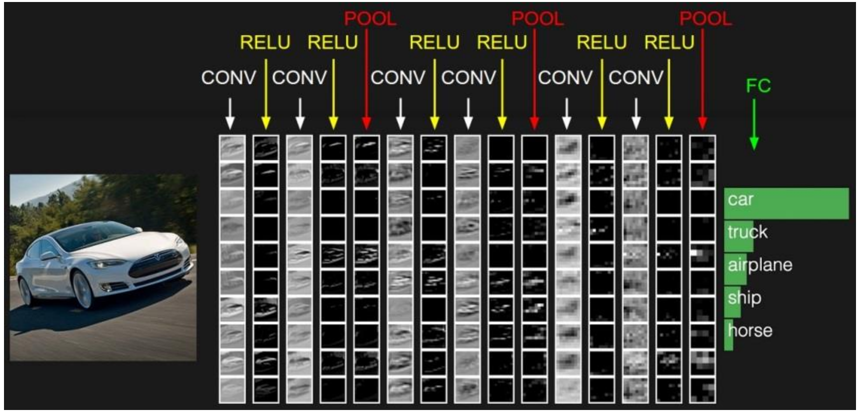
Overfitting & Underfitting

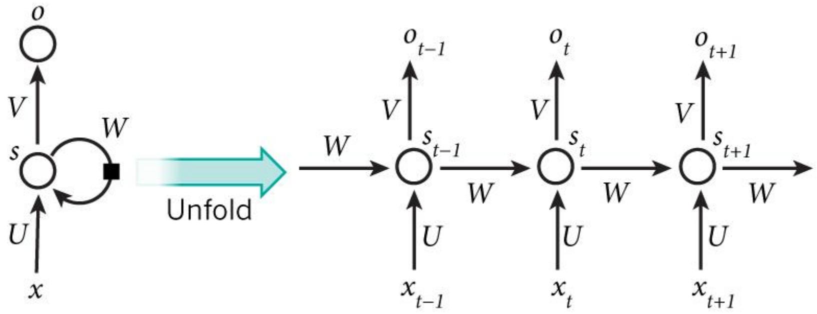




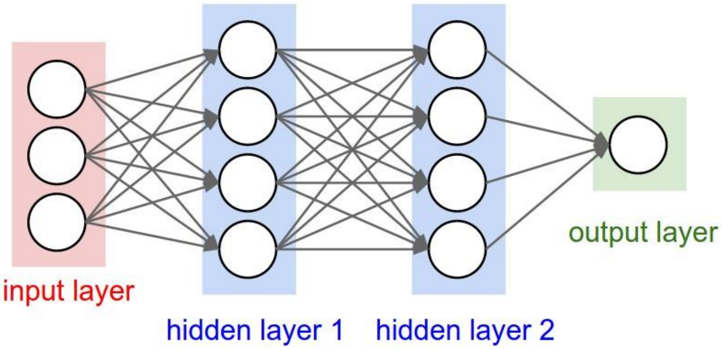
Overfitting & Underfitting

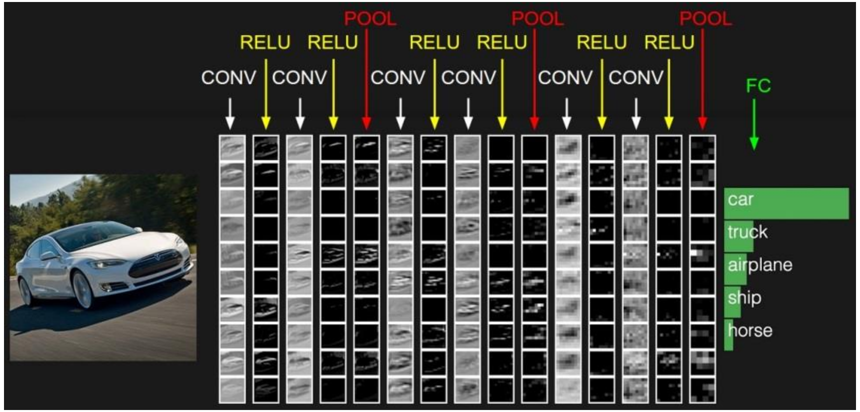


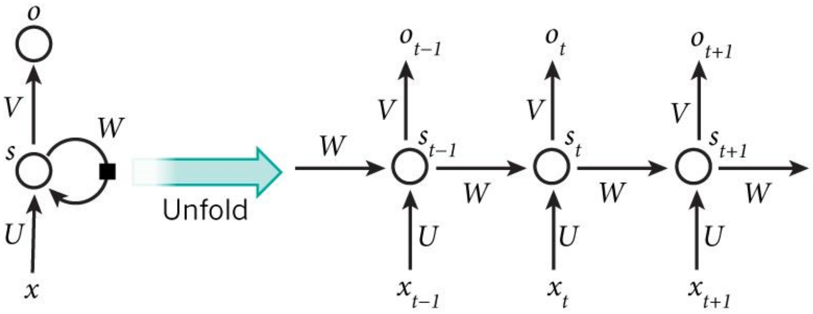




Overfitting & Underfitting

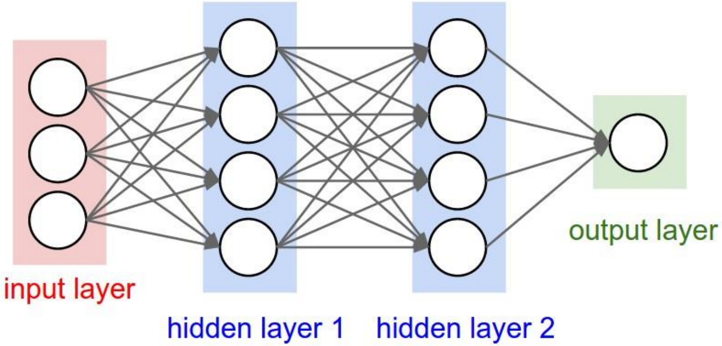


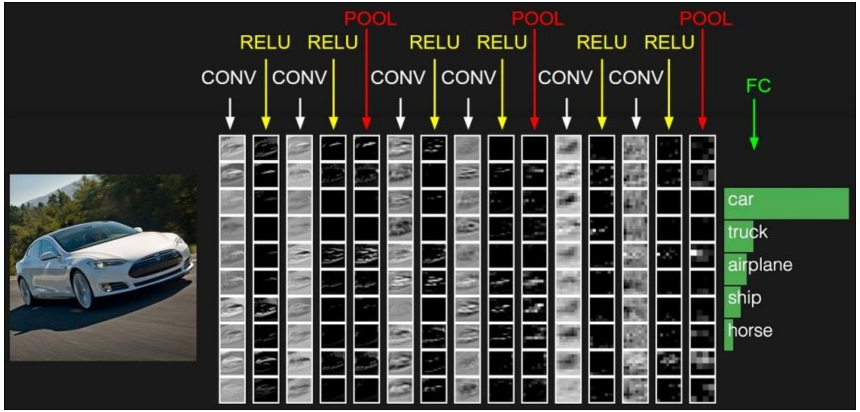


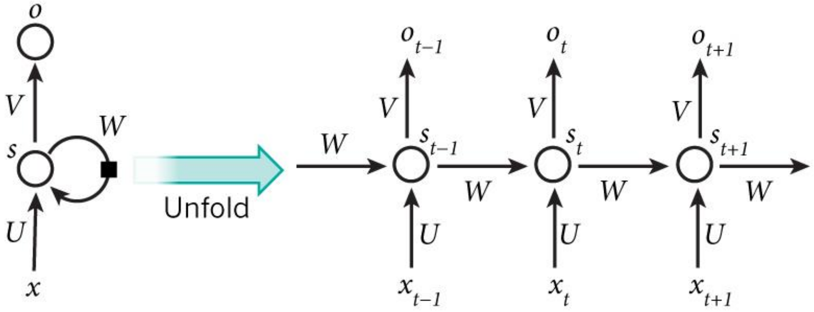


잘 된다니까 그냥
쓰면 되는 건가?

Overfitting & Underfitting







잘 된다니까 그냥
쓰면 되는 건가?

물론 NO!

Overfitting & Underfitting

딥러닝으로 결국 하고자 하는 것은 한번도 보지
못한 데이터에 대해서도 적절한 판단을 내리는 것

Overfitting & Underfitting

딥러닝으로 결국 하고자 하는 것은 한번도 보지
못한 데이터에 대해서도 적절한 판단을 내리는 것

⇩

하지만 학습 시 사용된 데이터에만 좋은 성능을 내거나
어떤 경우에는 학습 데이터에도 성능이 안 나오기도 함

Overfitting & Underfitting

딥러닝으로 결국 하고자 하는 것은 한번도 보지
못한 데이터에 대해서도 적절한 판단을 내리는 것

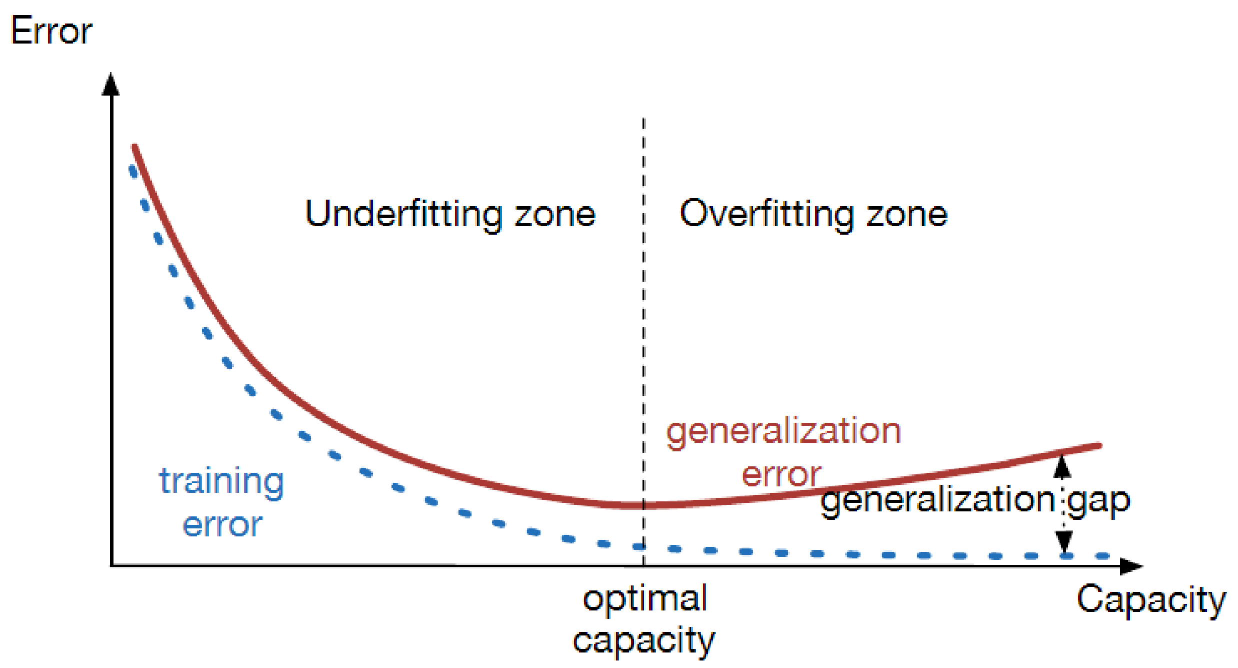
⇩

하지만 학습 시 사용된 데이터에만 좋은 성능을 내거나
어떤 경우에는 학습 데이터에도 성능이 안 나오기도 함

⇩

Overfitting & Underfitting

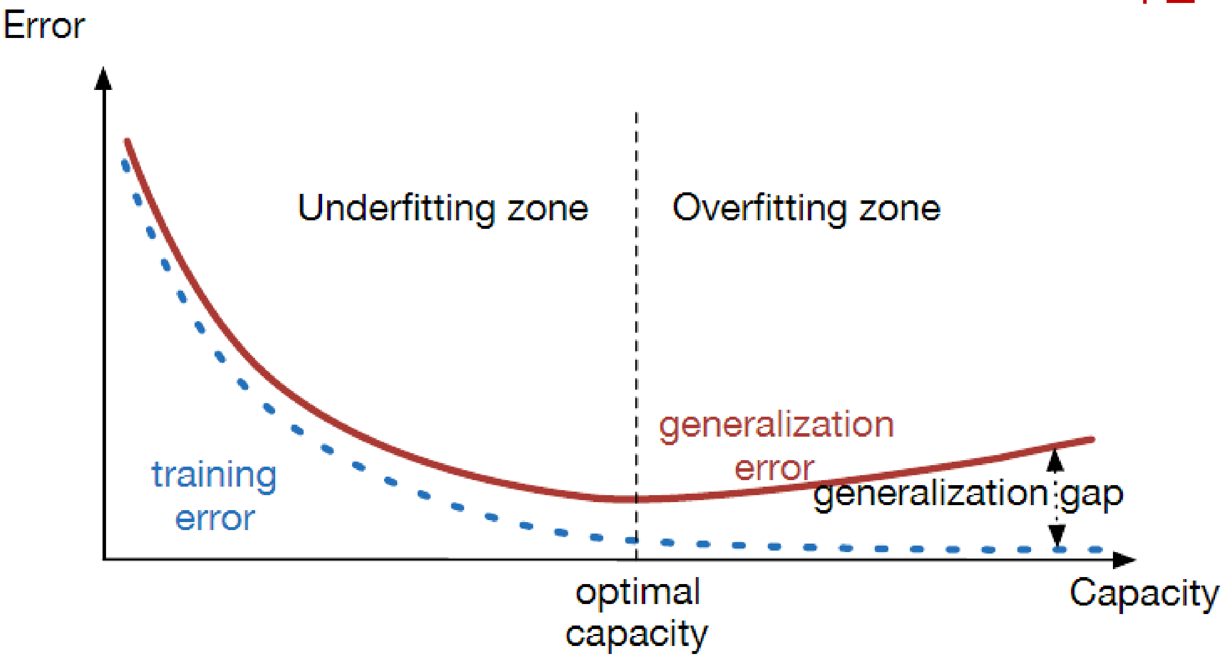
Overfitting & Underfitting



(출처: Andrew NG coursera 강의)

Overfitting & Underfitting

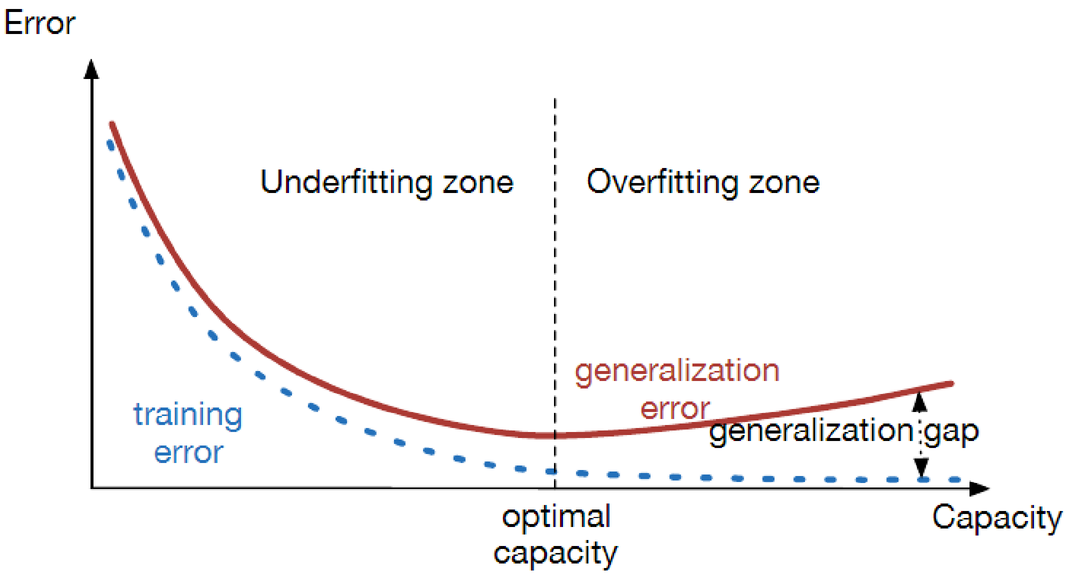
ex)
연습 문제 오답률
시험 문제 오답률



(출처: Andrew NG coursera 강의)

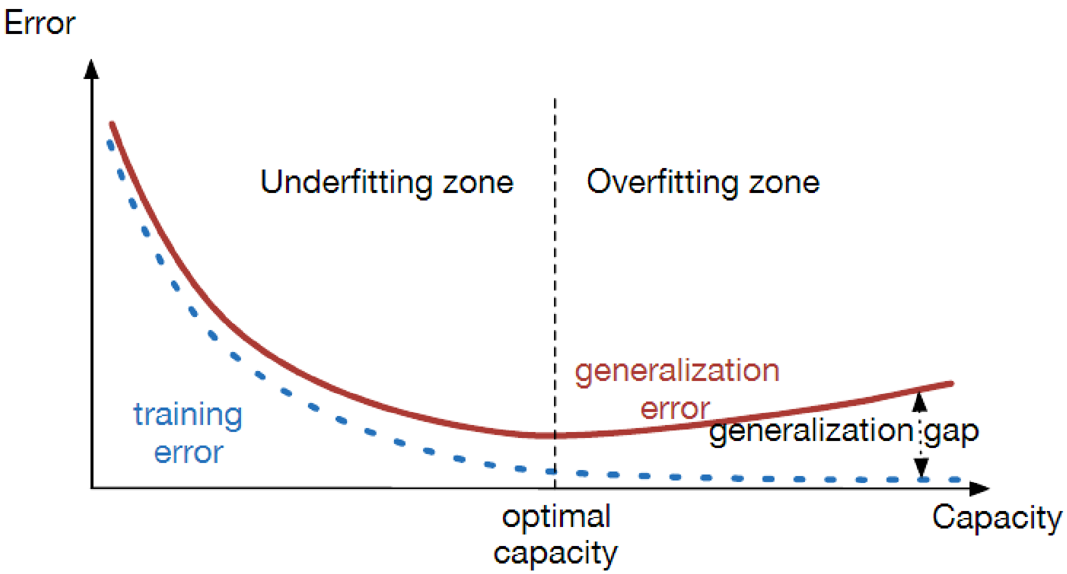
Overfitting & Underfitting

test error – train error
= generalization gap



(출처: Andrew NG coursera 강의)

Overfitting & Underfitting



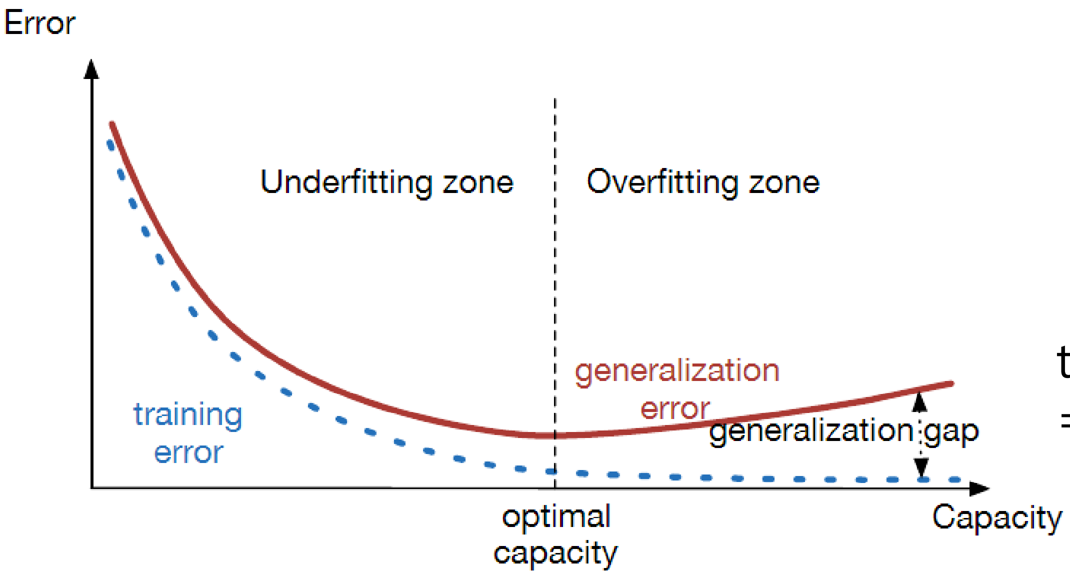
test error – train error
= generalization gap

⇩

목표는 test error의 최소화

(출처: Andrew NG coursera 강의)

Overfitting & Underfitting



test error – train error
= generalization gap

⇩

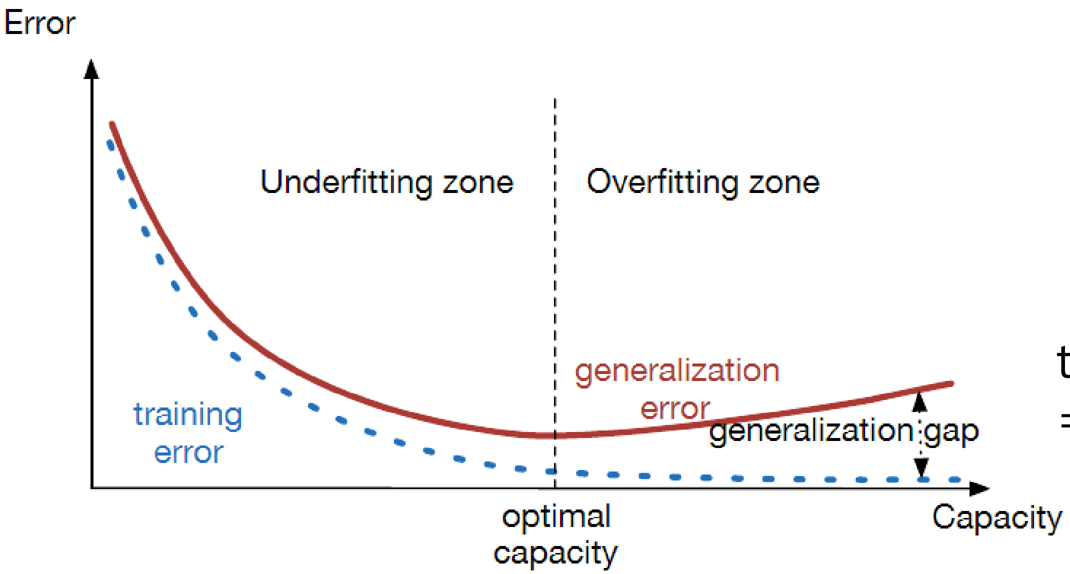
목표는 test error의 최소화

⇩

test error
= train error + generalization gap

(출처: Andrew NG coursera 강의)

Overfitting & Underfitting



test error – train error
= generalization gap

⇩

목표는 test error의 최소화

⇩

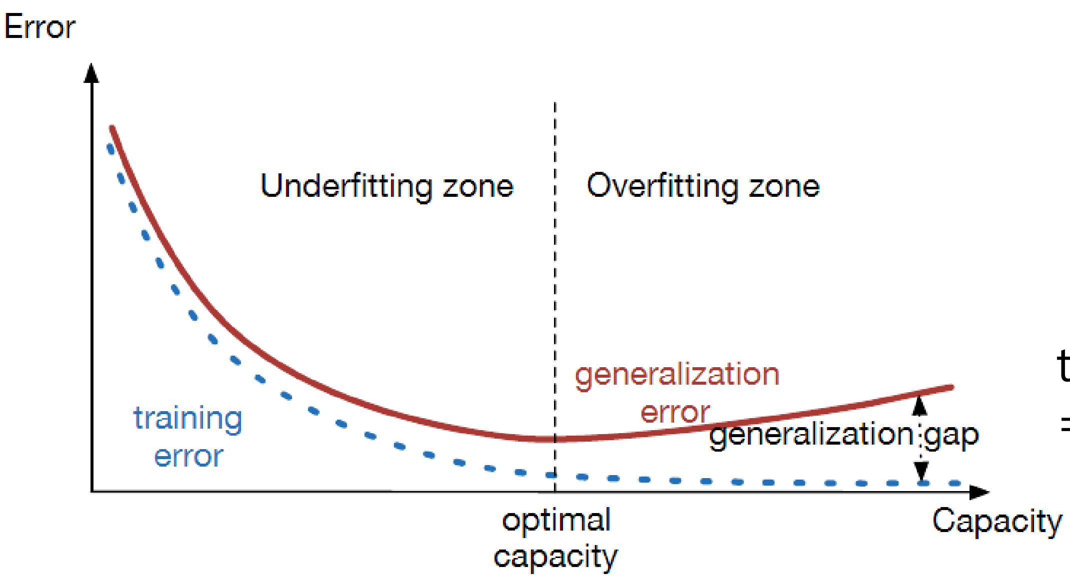
test error
= train error + generalization gap

⇩

즉, 둘 다 줄여야 함.

(출처: Andrew NG coursera 강의)

Overfitting & Underfitting



test error – train error
= generalization gap

⇩

목표는 test error의 최소화

⇩

test error
= train error + generalization gap

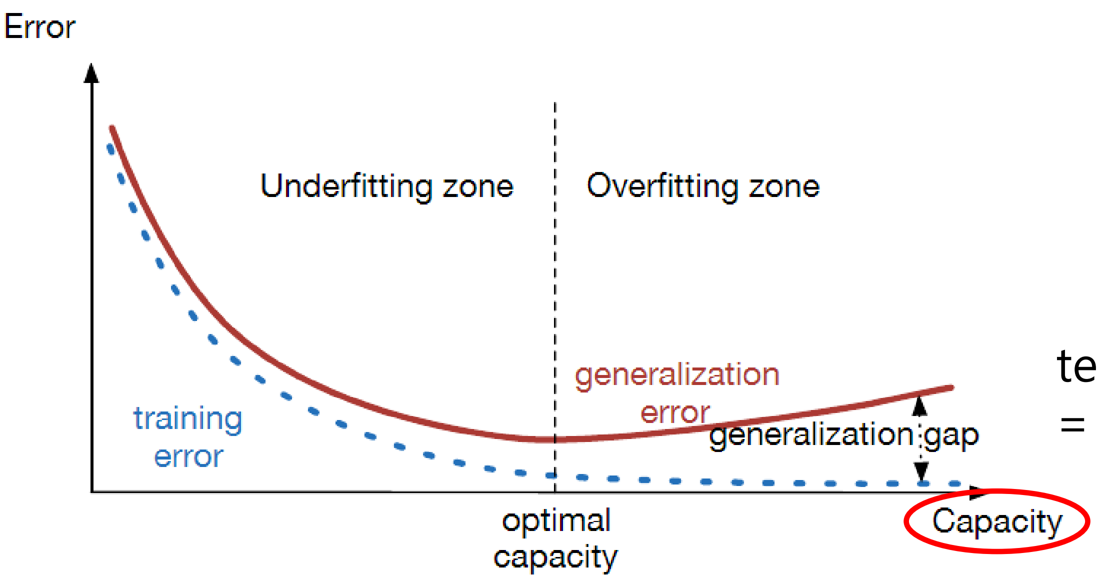
⇩

어떻게?

⇦ 즉, 둘 다 줄여야 함.

(출처: Andrew NG coursera 강의)

Overfitting & Underfitting



test error – train error
= generalization gap

⇩

목표는 test error의 최소화

⇩

test error
= train error + generalization gap

⇩

즉, 둘 다 줄여야 함.

Model Capacity를 통해서 줄여야 함 ⇦

Overfitting & Underfitting

Model Capacity?

Overfitting & Underfitting

Model Capacity?

모델이 표현 가능한 범위

Overfitting & Underfitting

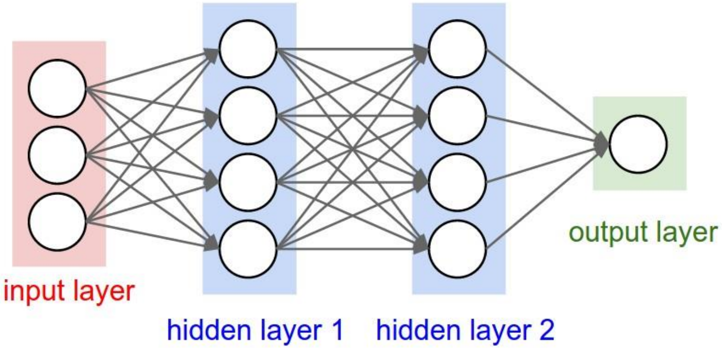
Model Capacity?

모델이 표현 가능한 범위 ex) $y = a_0 + a_1x1 + a_2x^2 + \cdots + a_nx^n$ 에서 n

Overfitting & Underfitting

Model Capacity?

모델이 표현 가능한 범위 ex) $y = a_0 + a_1 x1 + a_2 x^2 + \cdots + a_n x^n$ 에서 n

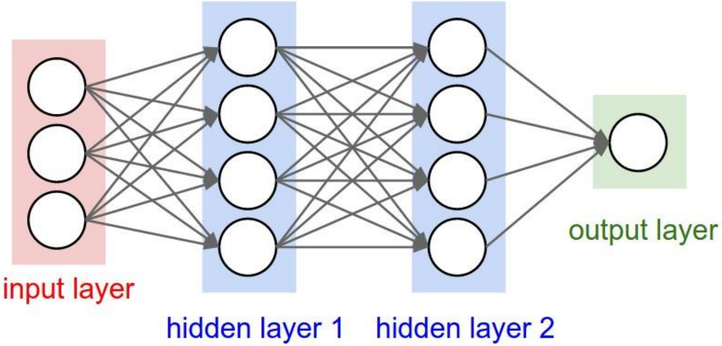


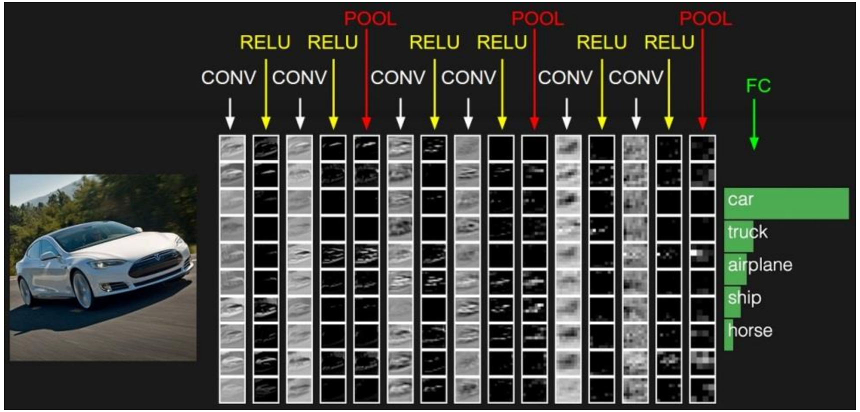
hidden layer 수

Overfitting & Underfitting

Model Capacity?

모델이 표현 가능한 범위 ex) $y = a_0 + a_1x1 + a_2x^2 + \cdots + a_nx^n$ 에서 n





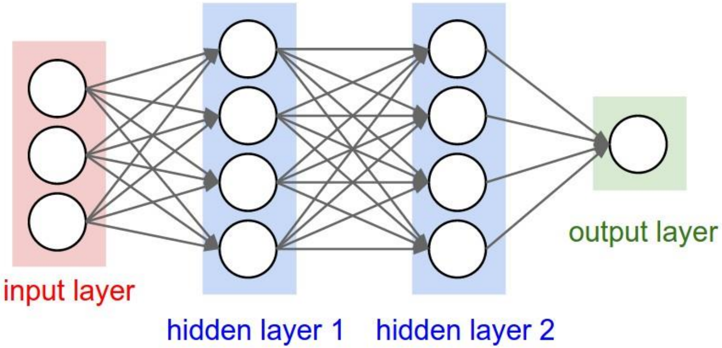
hidden layer 수

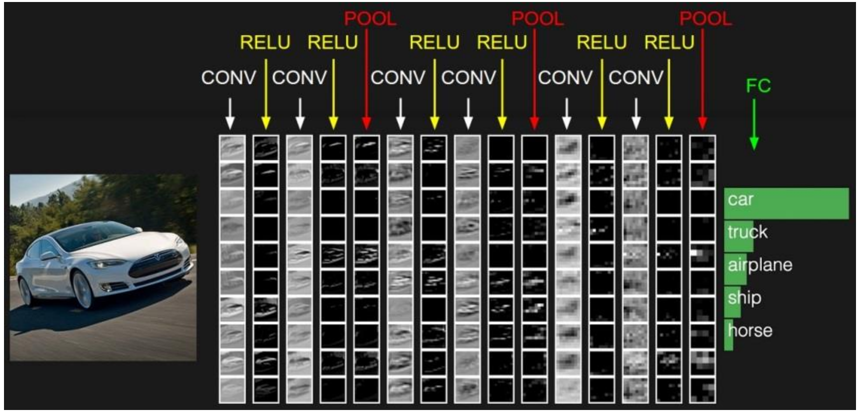
CNN filter 수

Overfitting & Underfitting

Model Capacity?

모델이 표현 가능한 범위 ex) $y = a_0 + a_1 x1 + a_2 x^2 + \cdots + a_n x^n$ 에서 n





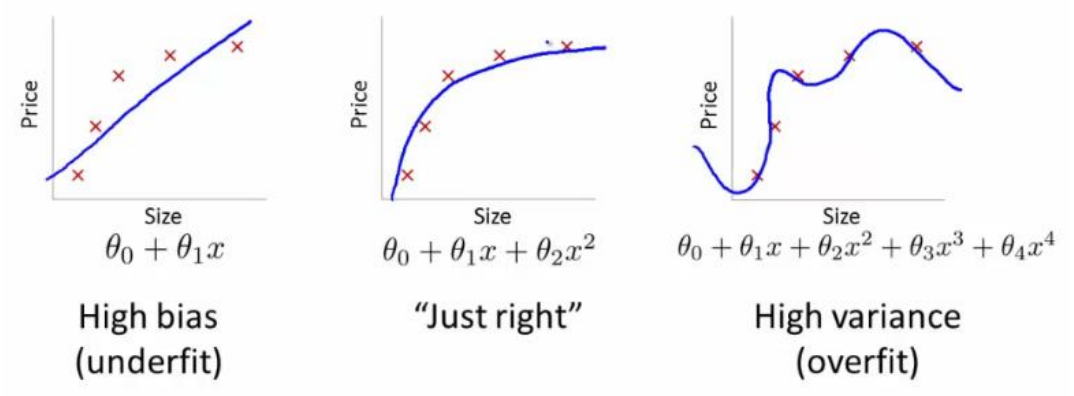
hidden layer 수

CNN filter 수

등등

Overfitting & Underfitting

Polynomial Regression

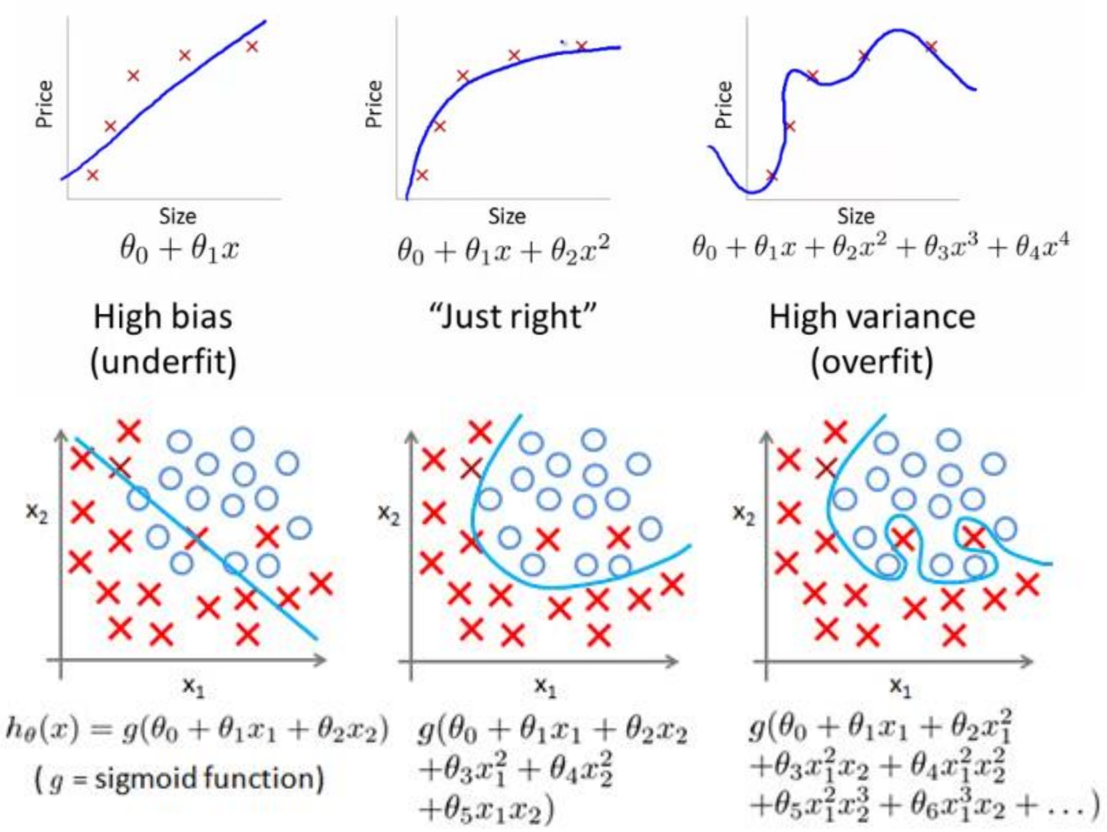


(출처: Andrew NG coursera 강의)

Overfitting & Underfitting

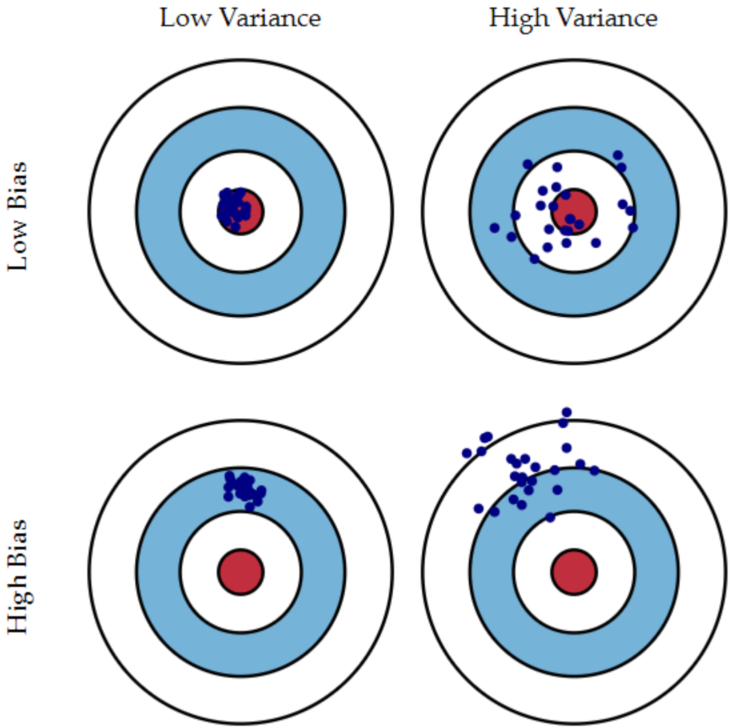
Polynomial Regression

Classification



(출처: Andrew NG coursera 강의)

Overfitting & Underfitting



Bias는 에러 값
Variance는 분산

(출처: http://scott.fortmann-roe.com/docs/BiasVariance.html)

Overfitting & Underfitting

Train error, Test error 를 구하기 위해서는 우선
데이터를 train set/test set으로 나눠야 함

Overfitting & Underfitting

Train error, Test error 를 구하기 위해서는 우선
데이터를 train set/test set으로 나눠야 함

⇩

train set만으로 학습 및 train error를 구하고
test set으로 test error를 구함

Overfitting & Underfitting

Train error, Test error 를 구하기 위해서는 우선
데이터를 train set/test set으로 나눠야 함

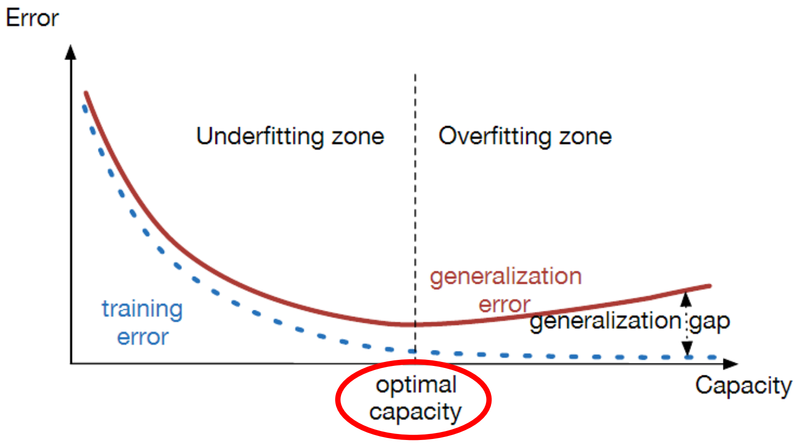
⇩

train set만으로 학습 및 train error를 구하고
test set으로 test error를 구함

⇩

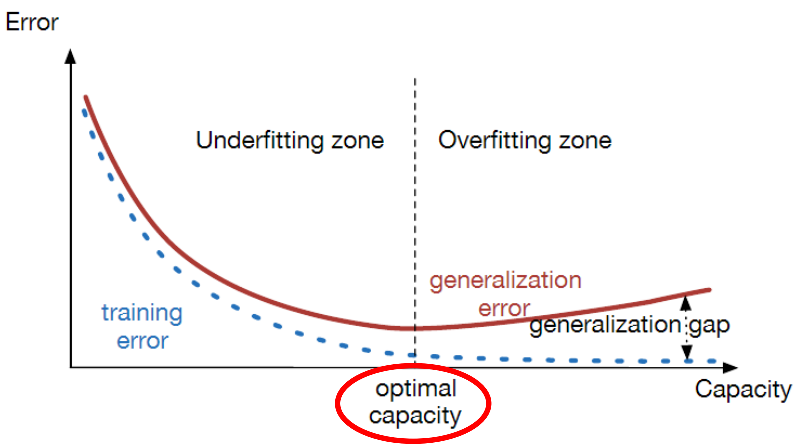
그런데 learning rate, model capacity 같은 hyperparameter도
찾아야 하기 때문에 사실 train/validation/test set으로 분할
ex) 5:3:2, 6:2:2

Overfitting & Underfitting



모델 capacity가 작으면 아예 optimal capacity에 도달할 수 없음

Overfitting & Underfitting

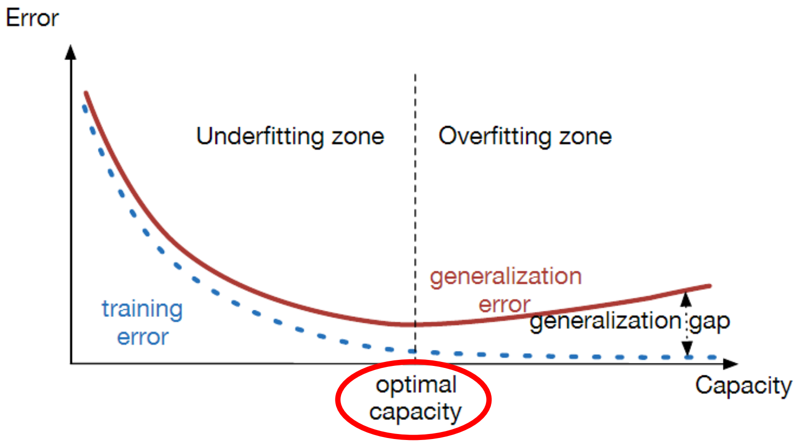


모델 capacity가 작으면 아예 optimal capacity에 도달할 수 없음

⇩

그렇다면 capacity를 좀 넉넉하게 주고 generalization gap을 줄이는 건 어떨까?

Overfitting & Underfitting



모델 capacity가 작으면 아예 optimal capacity에 도달할 수 없음

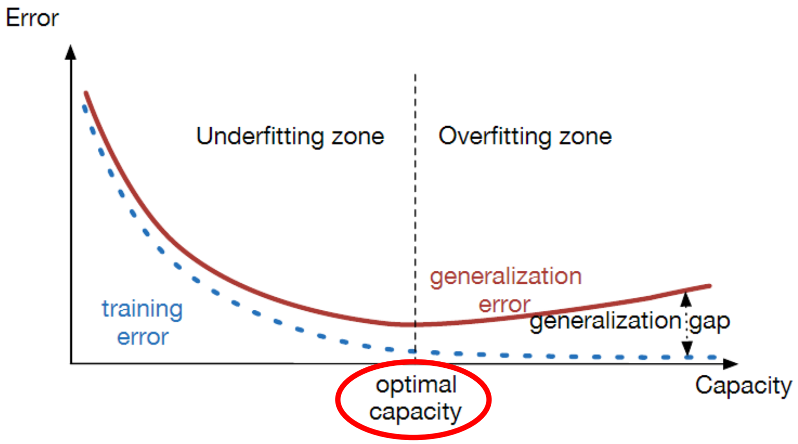
⇩

그렇다면 capacity를 좀 넉넉하게 주고 generalization gap을 줄이는 건 어떨까?

⇩

Regularization!

Overfitting & Underfitting



$$Total\ Loss = Loss + Regularization$$

모델 capacity가 작으면 아예 optimal capacity에 도달할 수 없음

⇩

그렇다면 capacity를 좀 넉넉하게 주고 generalization gap을 줄이는 건 어떨까?

⇩

Regularization!

Overfitting & Underfitting

ex) **Linear regression with regularization**

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2$$

(출처: http://www.holehouse.org/mlclass/10_Advice_for_applying_machine_learning.html)

Overfitting & Underfitting

ex) **Linear regression with regularization**

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \frac{1}{2m} \underbrace{\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2}_{\text{기존의 loss function}} + \frac{\lambda}{2m} \sum_{j=1}^{m} \theta_j^2$$

Overfitting & Underfitting

ex) **Linear regression with regularization**

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \underbrace{\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2}_{\text{기존의 loss function}} + \underbrace{\boxed{\frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2}}_{\text{weight들의 제곱의 합}}$$

(출처: http://www.holehouse.org/mlclass/10_Advice_for_applying_machine_learning.html)

Overfitting & Underfitting

ex) **Linear regression with regularization**

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \boxed{\frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2}$$

기존의 loss function

weight들의 제곱의 합

기본적으로 weight 값들이 작아지고
loss를 줄이는데 꼭 필요한 값들만 남음

(출처: http://www.holehouse.org/mlclass/10_Advice_for_applying_machine_learning.html)

Overfitting & Underfitting

ex) **Linear regression with regularization**

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \underbrace{\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2}_{\text{기존의 loss function}} + \boxed{\frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2}$$

λ에 의해 비율이 결정됨

기존의 loss function

weight들의 제곱의 합

기본적으로 weight 값들이 작아지고
loss를 줄이는데 꼭 필요한 값들만 남음

(출처: http://www.holehouse.org/mlclass/10_Advice_for_applying_machine_learning.html)

Overfitting & Underfitting

large λ proper λ small λ

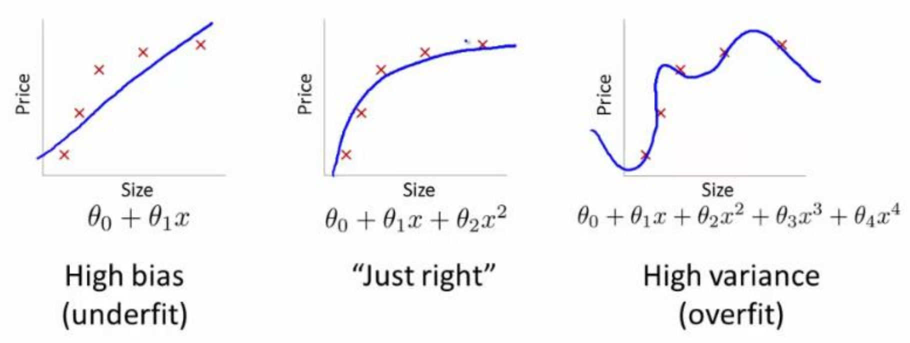
Polynomial Regression

Classification

Overfitting & Underfitting

large λ proper λ small λ

Polynomial Regression



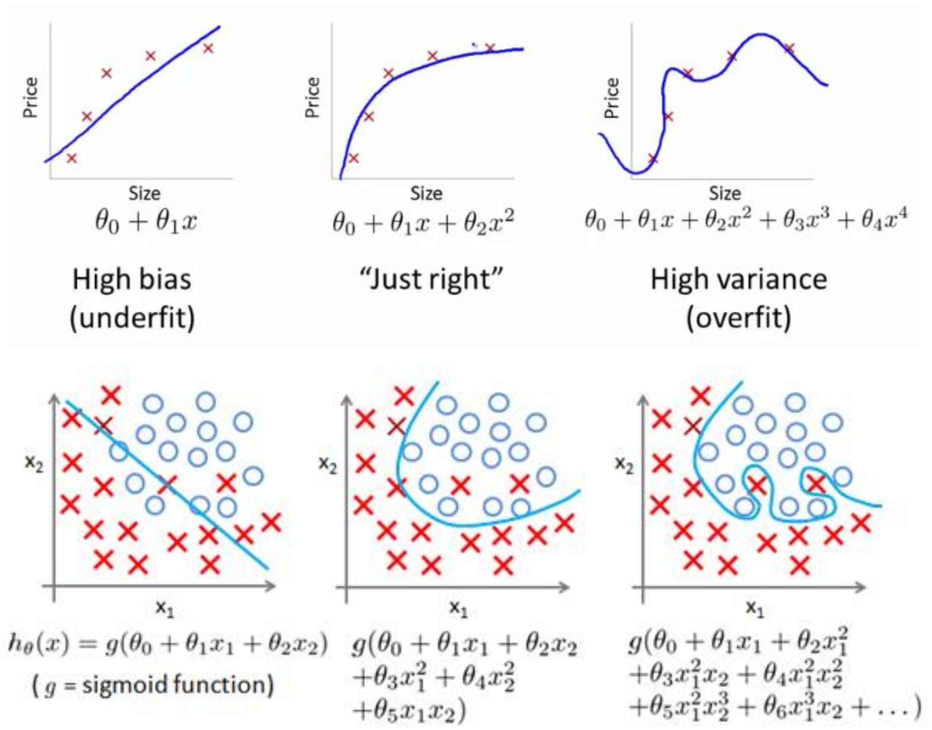
Classification

Overfitting & Underfitting

large λ | proper λ | small λ

Polynomial Regression

Classification



Overfitting & Underfitting

Linear regression with regularization

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2$$

Overfitting & Underfitting

Linear regression with regularization

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{m}\boxed{\theta_j^2}$$

In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Overfitting & Underfitting

Linear regression with regularization

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{m}\boxed{\theta_j^2}$$

In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

구현은 어떻게?

Overfitting & Underfitting

class `torch.optim.SGD`(*params*, *lr=<object object>*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*) [source]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float*, *optional*) – momentum factor (default: 0)
- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float*, *optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool*, *optional*) – enables Nesterov momentum (default: False)

Overfitting & Underfitting

class `torch.optim.SGD`(*params*, *lr=<object object>*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*) [source]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

Overfitting & Underfitting

class `torch.optim.SGD`(*params, lr=<object object>, momentum=0, dampening=0, weight_decay=0, nesterov=False*) [source]

Implements stochastic gradient descent (optionally with momentum).

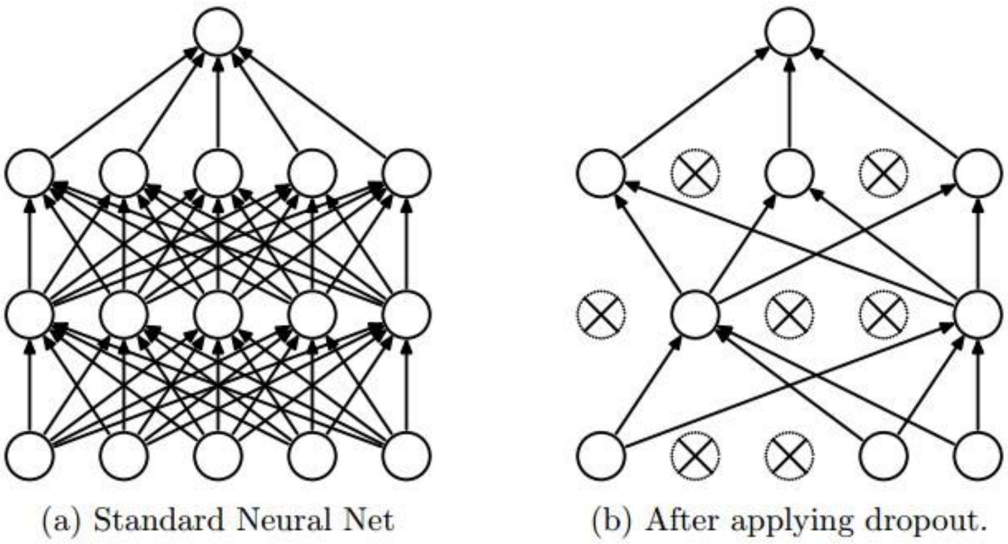
Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

Weight Decay -> https://stats.stackexchange.com/questions/29130/difference-between-neural-net-weight-decay-and-learning-rate

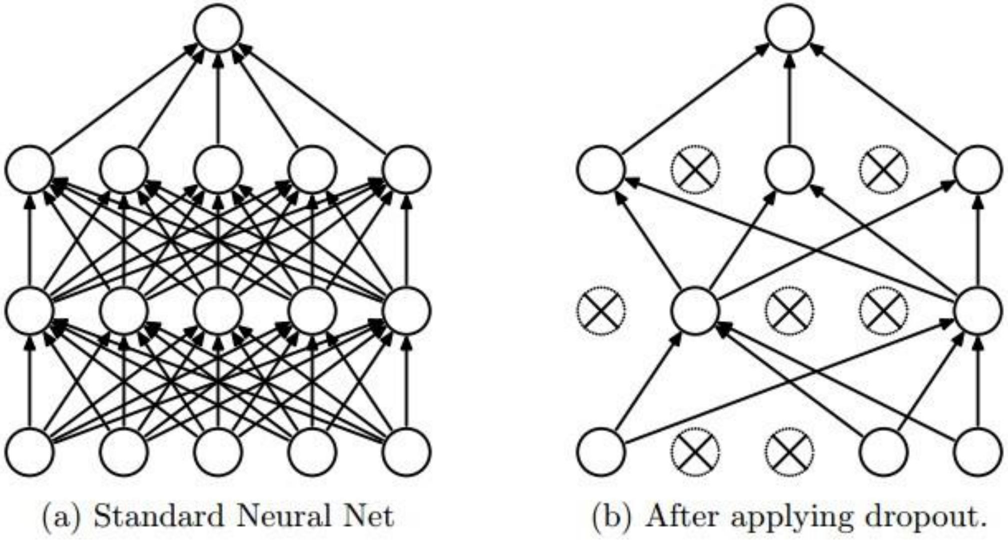
Overfitting & Underfitting



Dropout

(출처: http://cs231n.github.io/assets/nn2/dropout.jpeg)

Overfitting & Underfitting

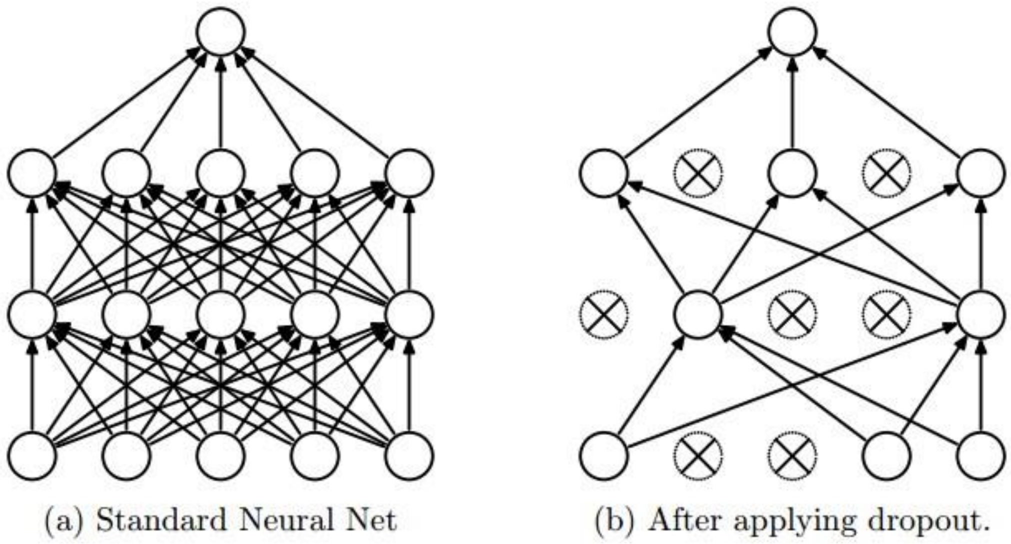


일정 확률로 전달 값을 0으로 바꿔 값이 전달 안되고 drop out 되게 하는 방법

Dropout

(출처: http://cs231n.github.io/assets/nn2/dropout.jpeg)

Overfitting & Underfitting



Dropout

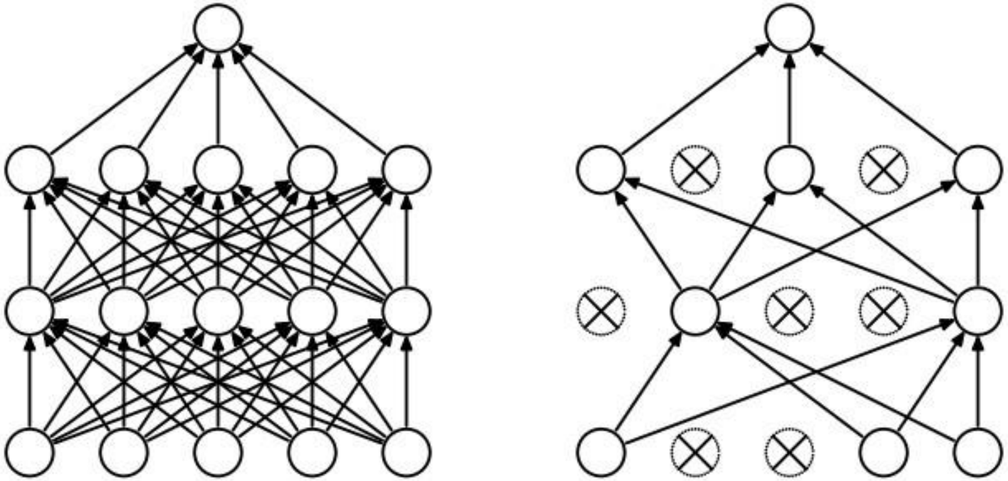
일정 확률로 전달 값을 0으로 바꿔 값이 전달 안되고 drop out 되게 하는 방법

⇩

어떻게 overfitting을 방지 하는 걸까?

(출처: http://cs231n.github.io/assets/nn2/dropout.jpeg)

Overfitting & Underfitting



(a) Standard Neural Net

(b) After applying dropout.

Dropout

일정 확률로 전달 값을 0으로 바꿔 값이 전달 안되고 drop out 되게 하는 방법

⇩

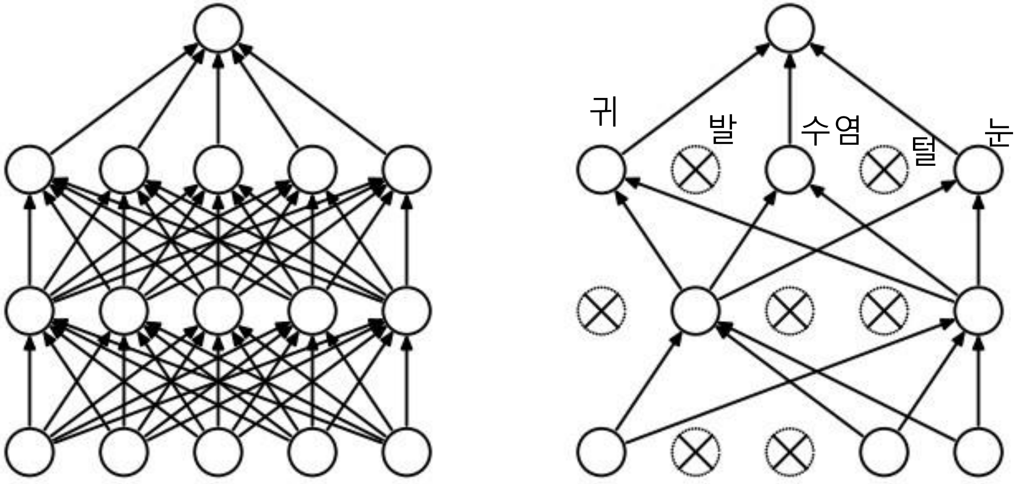
어떻게 overfitting을 방지 하는 걸까?

⇩

전달을 drop한다는 것은 해당 weight를 0으로 만들어 없애기 때문에 model capacity를 낮추는 것과 같다.

(출처: http://cs231n.github.io/assets/nn2/dropout.jpeg)

Overfitting & Underfitting



(a) Standard Neural Net (b) After applying dropout.

Dropout

일정 확률로 전달 값을 0으로 바꿔 값이 전달 안되고 drop out 되게 하는 방법

⇩

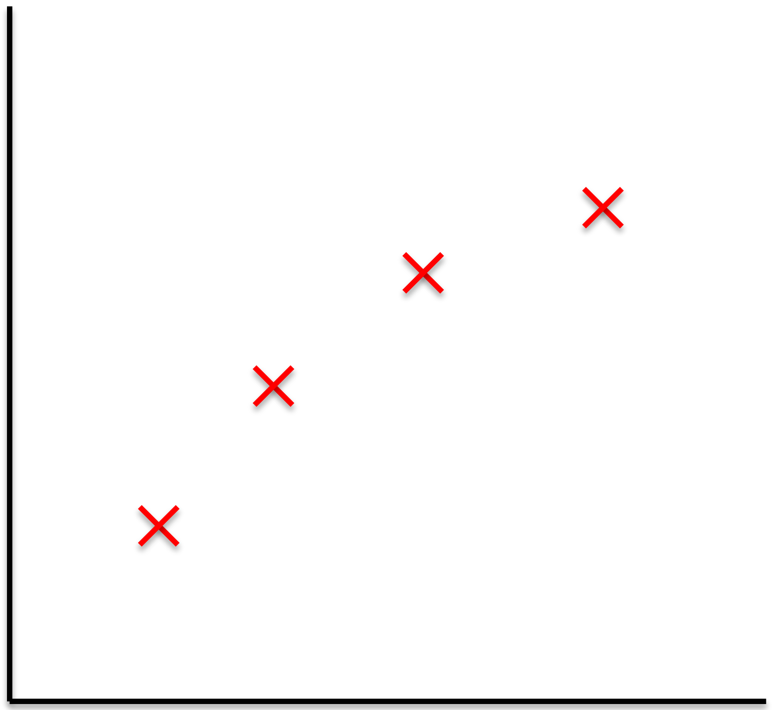
어떻게 overfitting을 방지 하는 걸까?

⇩

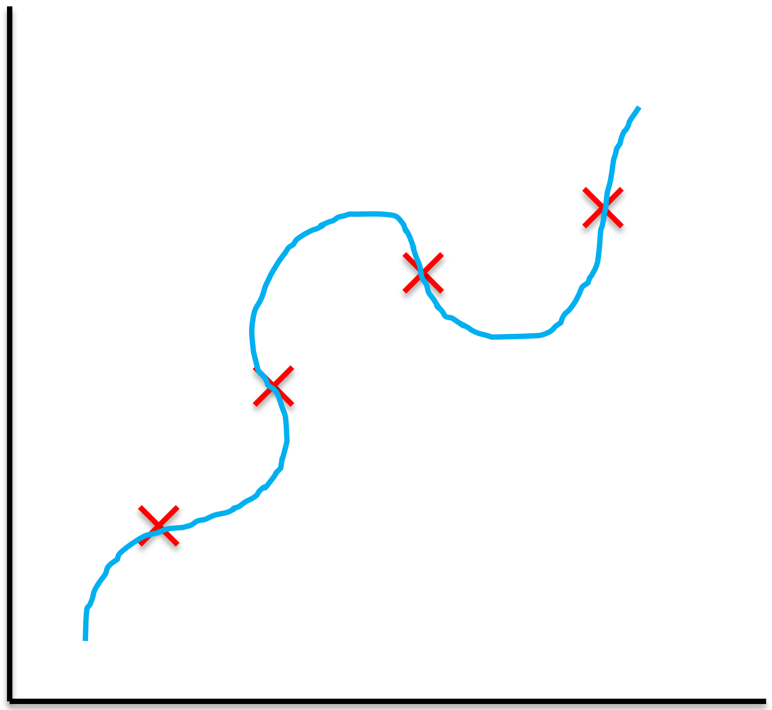
전달을 drop한다는 것은 해당 weight를 0으로 만들어 없애기 때문에 model capacity를 낮추는 것과 같다.

(출처: http://cs231n.github.io/assets/nn2/dropout.jpeg)

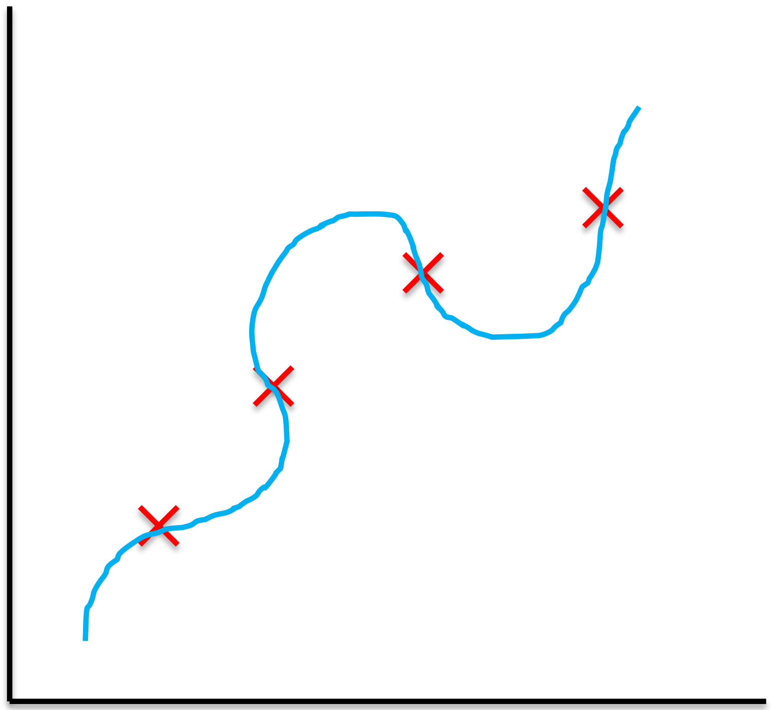
Overfitting & Underfitting



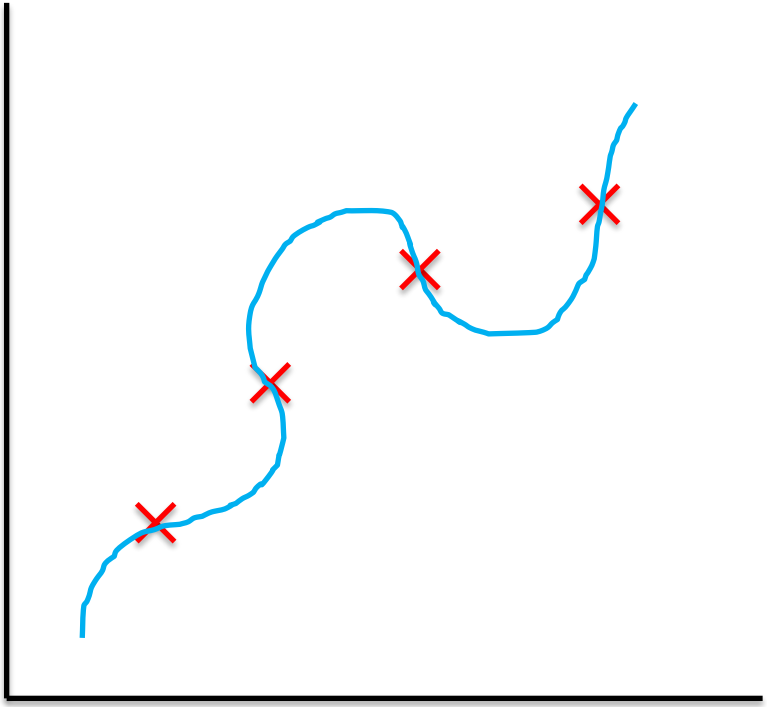
Overfitting & Underfitting

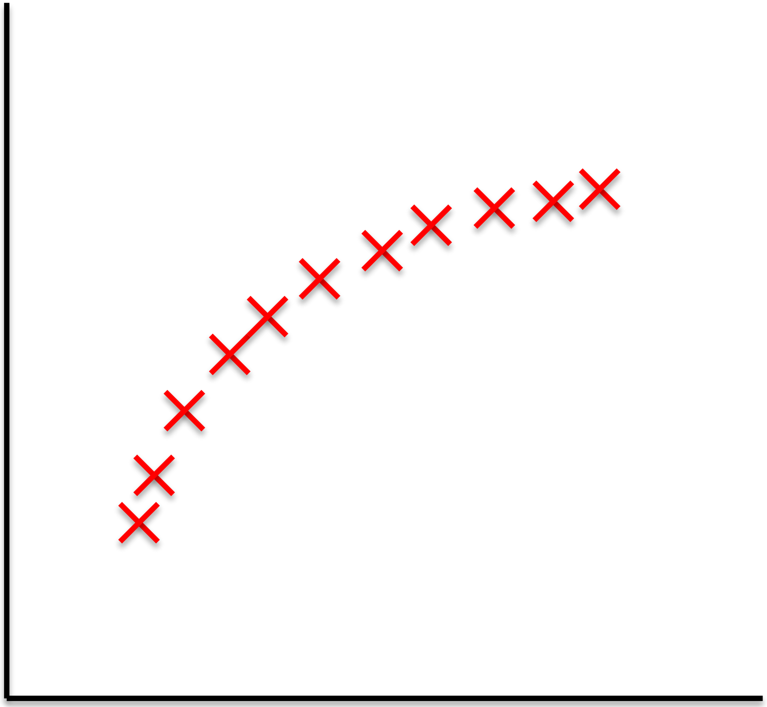


Overfitting & Underfitting

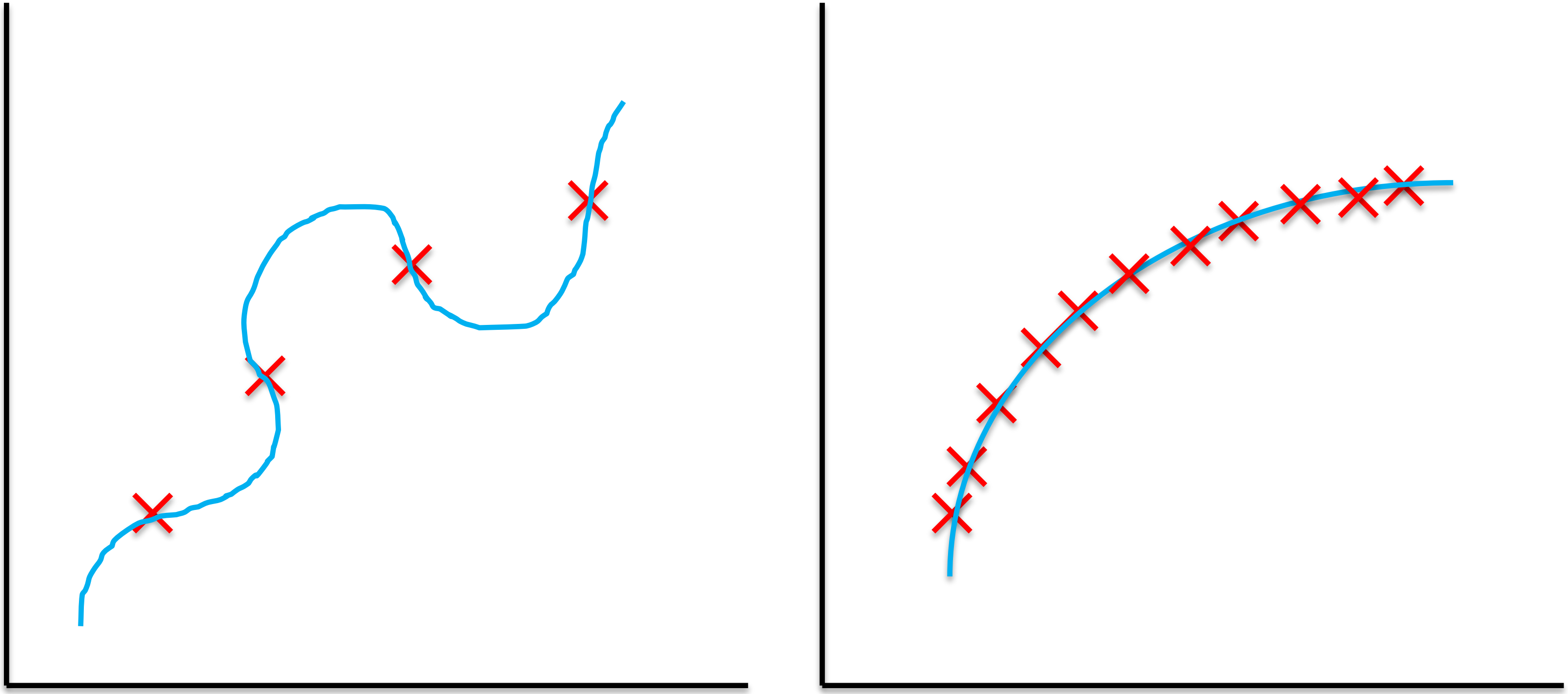


Overfitting & Underfitting



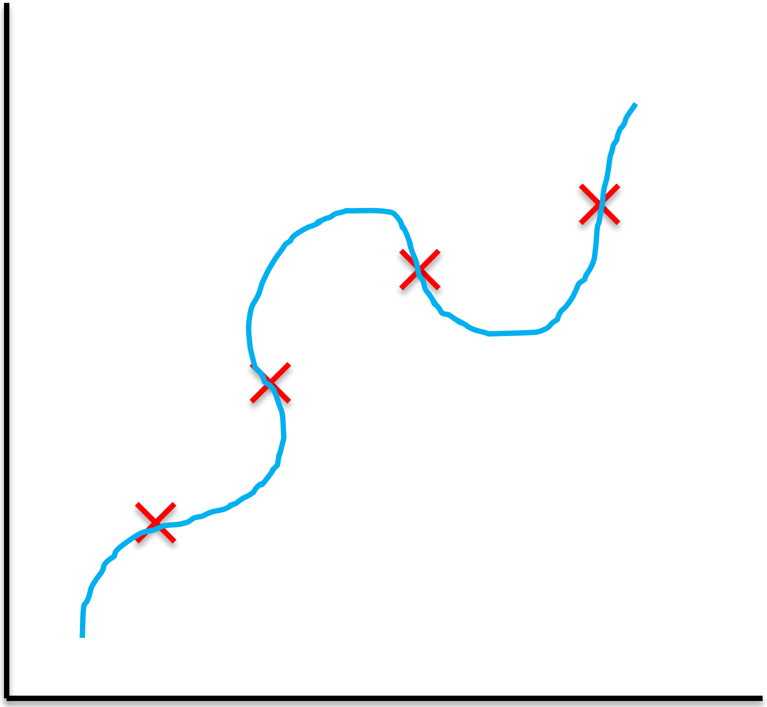


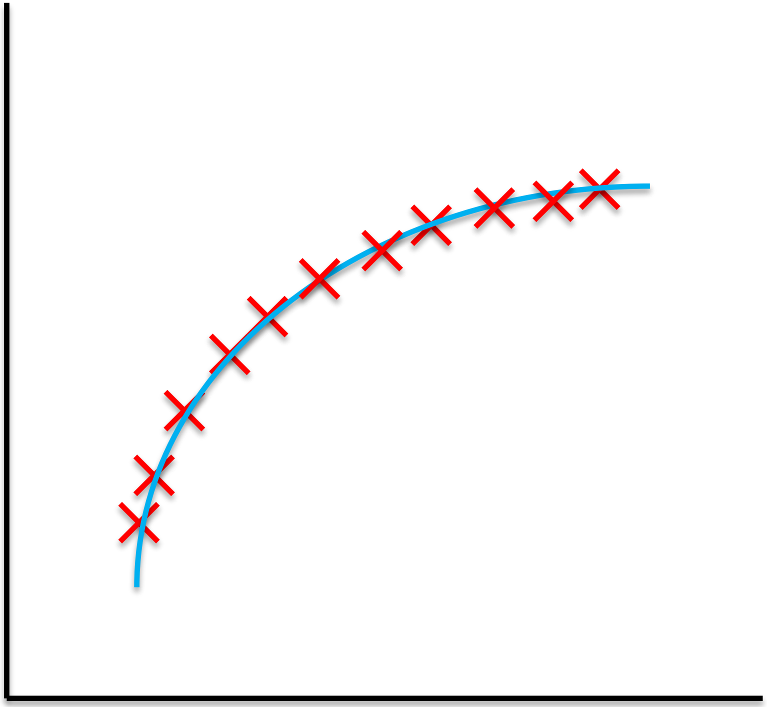
Overfitting & Underfitting



Overfitting & Underfitting

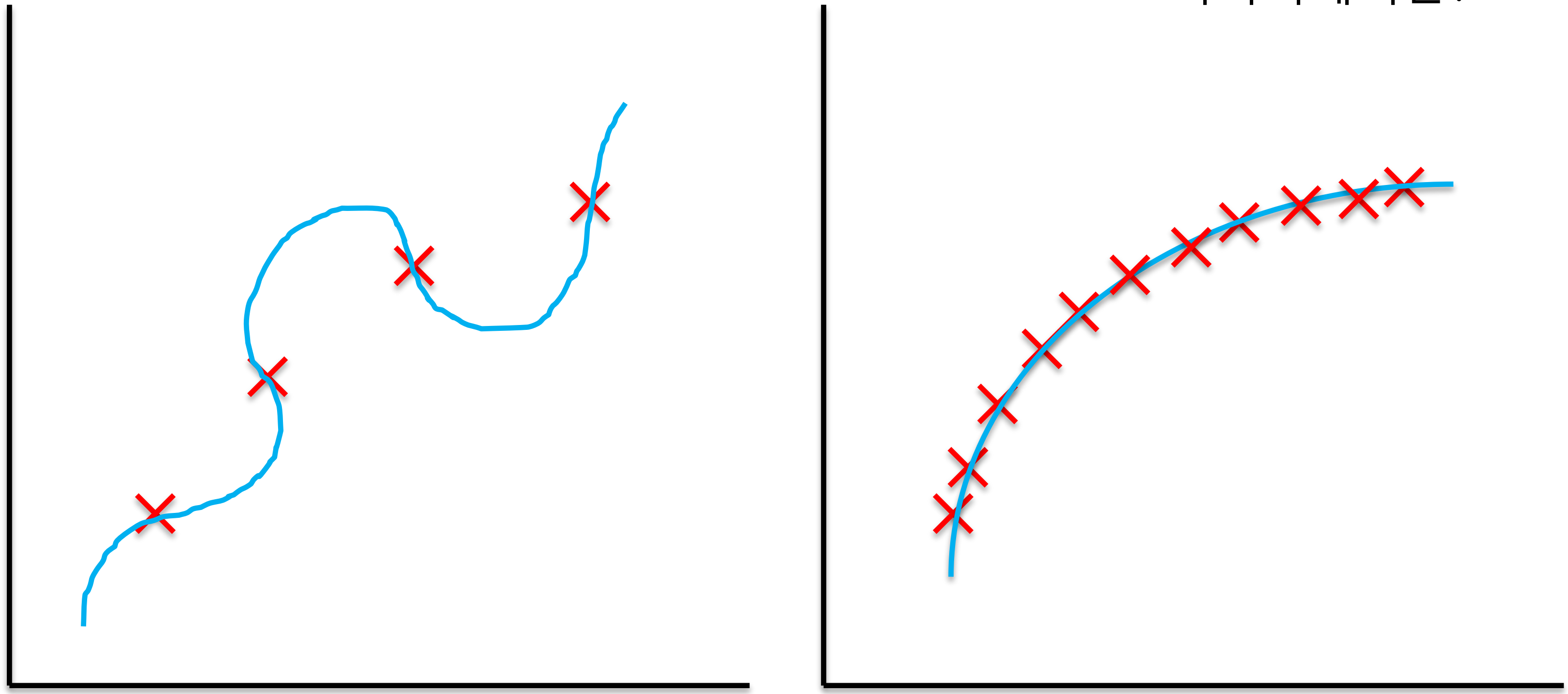
Data Augmentation





Overfitting & Underfitting

Data Augmentation
-> 이미지에서는?



Overfitting & Underfitting

원본

Overfitting & Underfitting

원본

Flip(LR)

Overfitting & Underfitting


원본


Flip(LR)


Flip(UD)

Overfitting & Underfitting

원본

Flip(LR)

Flip(UD)

Translation

Overfitting & Underfitting

원본

Flip(LR)

Flip(UD)

Translation

Rotate

Overfitting & Underfitting

원본

Flip(LR)

Flip(UD)

Translation

Rotate

CROP

Overfitting & Underfitting

원본

Flip(LR)

Flip(UD)

Translation

Rotate

CROP

$+\alpha$

Overfitting & Underfitting

원본

Overfitting & Underfitting

원본

Overfitting & Underfitting

원본

CROP

Overfitting & Underfitting

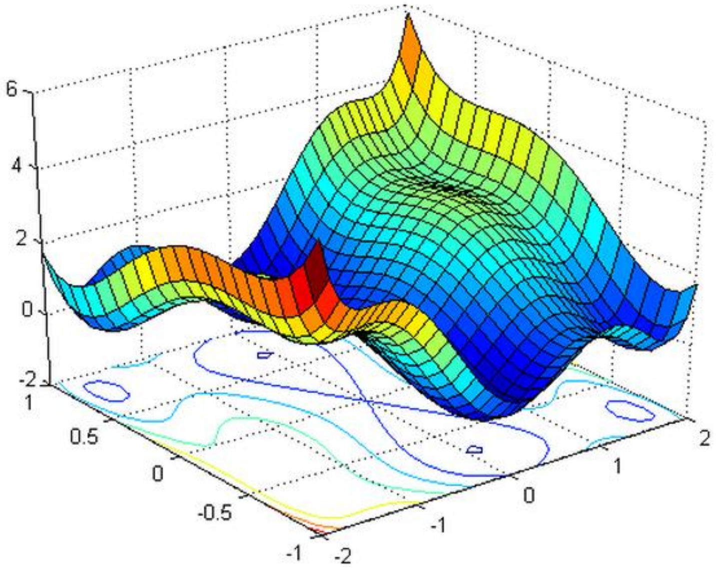
원본

CROP

```
img_dir = "./maps/"
img_data = dset.ImageFolder(root=img_dir, transform = transforms.Compose([
                                            transforms.Scale(size=img_size),
                                            transforms.CenterCrop(size=(img_size,img_size*2)),
                                            transforms.ToTensor(),
                                            ]))
img_batch = data.DataLoader(img_data, batch_size=batch_size,
                            shuffle=True, num_workers=2)
```

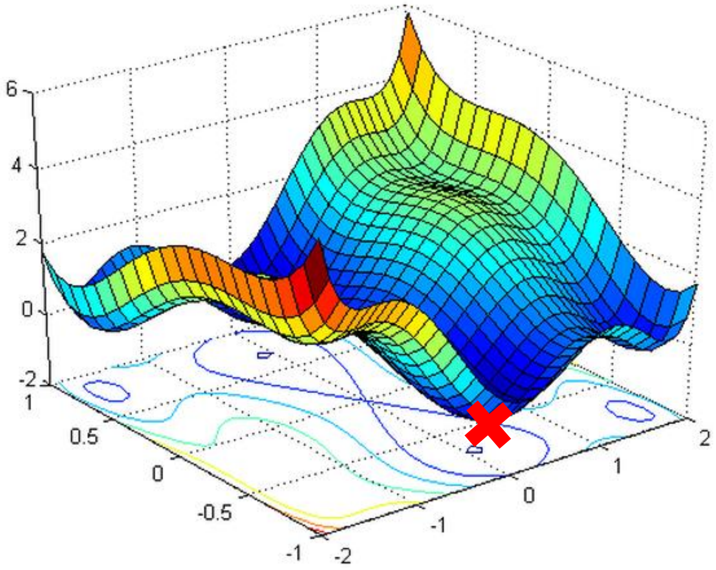
Convergence

학습 시 목표는 Global Minimum에 수렴하는 것

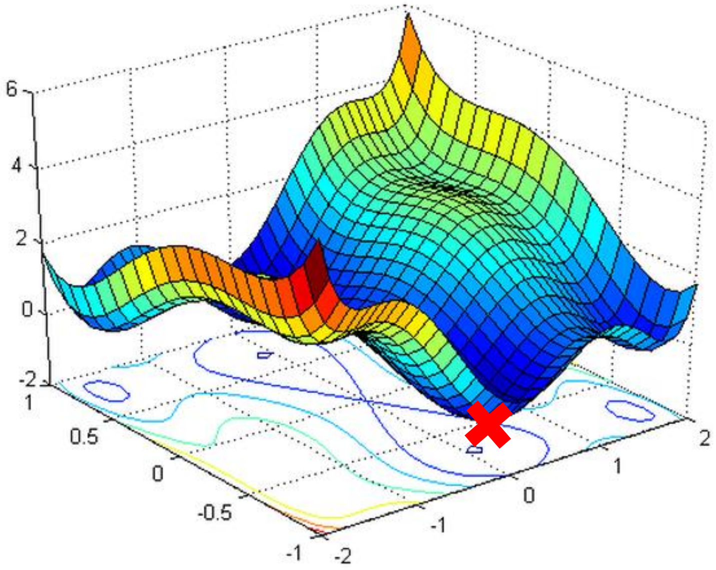


Convergence

학습 시 목표는 Global Minimum에 수렴하는 것



Convergence

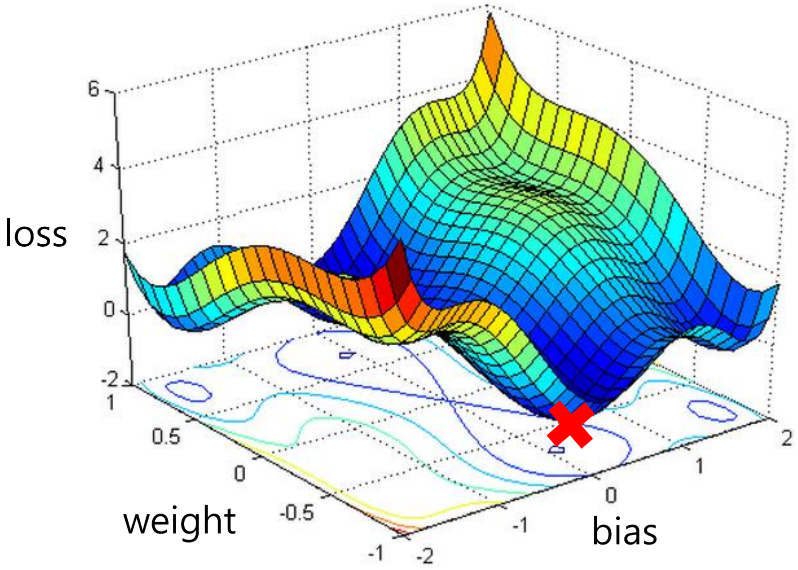


학습 시 목표는 Global Minimum에 수렴하는 것

⇩

어떻게 하면 잘 수렴할 수 있을까?

Convergence



학습 시 목표는 Global Minimum에 수렴하는 것

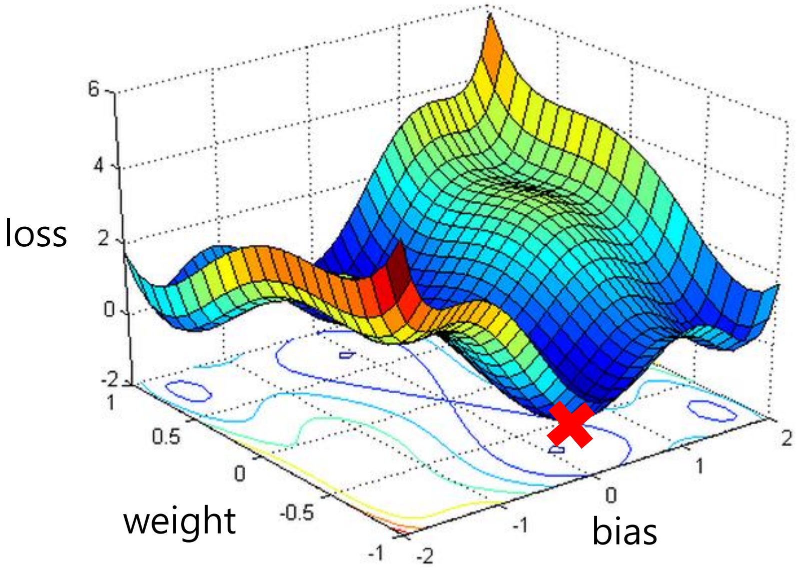
⇩

어떻게 하면 잘 수렴할 수 있을까?

⇩

왼쪽 그림은 weight와 bias에 대한 loss값을 나타낸 것

Convergence



학습 시 목표는 Global Minimum에 수렴하는 것

⇩

어떻게 하면 잘 수렴할 수 있을까?

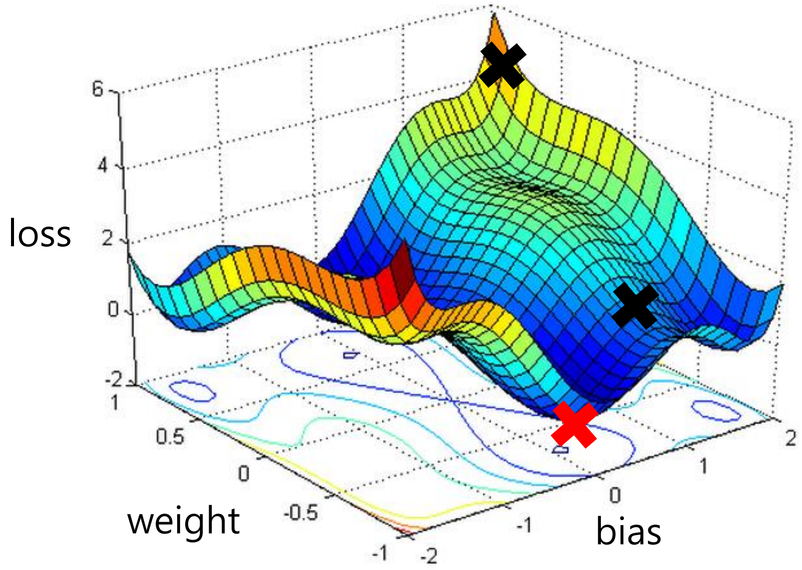
⇩

왼쪽 그림은 weight와 bias에 대한 loss값을 나타낸 것

⇩

Gradient descent를 사용하기 때문에 시작지점이 중요하다.

Convergence



학습 시 목표는 Global Minimum에 수렴하는 것

⇩

어떻게 하면 잘 수렴할 수 있을까?

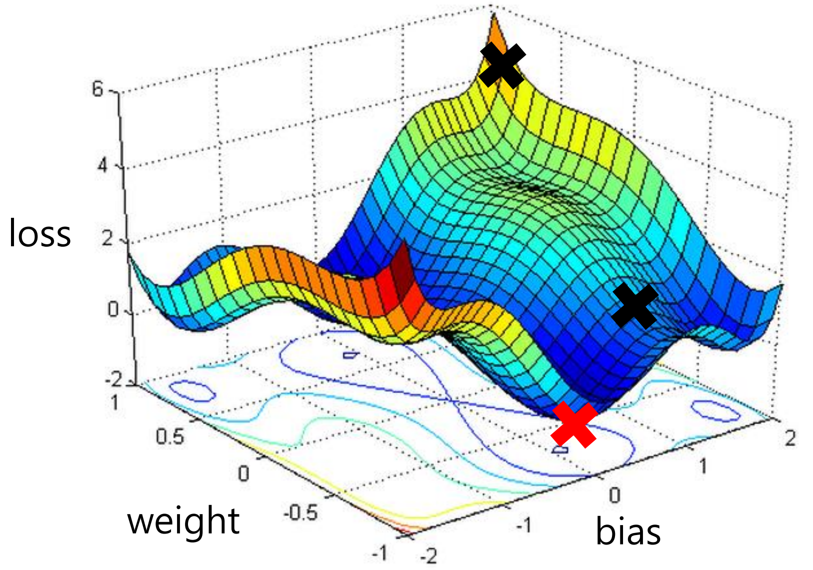
⇩

왼쪽 그림은 weight와 bias에 대한 loss값을 나타낸 것

⇩

Gradient descent를 사용하기 때문에 시작지점이 중요하다.

Convergence



학습 시 목표는 Global Minimum에 수렴하는 것

⇩

어떻게 하면 잘 수렴할 수 있을까?

⇩

왼쪽 그림은 weight와 bias에 대한 loss값을 나타낸 것

⇩

Gradient descent를 사용하기 때문에 시작지점이 중요하다.

Weight Initialization!!

Convergence

일단 global minimum은 모르기 때문에
좋은 시작점은 모름

Convergence

일단 global minimum은 모르기 때문에
좋은 시작점은 모름

⇩

최소 학습하다가 업데이트 값이 0이 되거나
엄청 큰 값이 되는 것만 피하자
(Vanishing Gradient & Exploding Gradient)

Convergence

일단 global minimum은 모르기 때문에
좋은 시작점은 모름

⇩

최소 학습하다가 업데이트 값이 0이 되거나
엄청 큰 값이 되는 것만 피하자
(Vanishing Gradient & Exploding Gradient)

⇩

Xavier Initialization/He Initialization

Convergence

Xavier Initialization

sigmoid나 tanh를 사용할 때 gradient가 적절히 전달되도록 해주는 초기값.

He Initialization

ReLU를 사용할 때 gradient 값이 절반 정도는 0으로 바뀌는걸 감안하여 적용한 초기값

Convergence

Xavier Initialization

sigmoid나 tanh를 사용할 때 gradient가 적절히 전달되도록 해주는 초기값.

```
# Xavier initialization
# Glorot et al. 2010
W = np.random.randn(fan_in, fan_out)/np.sqrt(fan_in)
```

He Initialization

ReLU를 사용할 때 gradient 값이 절반 정도는 0으로 바뀌는걸 감안하여 적용한 초기값

```
# He et al. 2015
W = np.random.randn(fan_in, fan_out)/np.sqrt(fan_in/2)
```

Convergence

Xavier Initialization

sigmoid나 tanh를 사용할 때
gradient가 적절히 전달되도록
해주는 초기값.

```
# Xavier initialization
# Glorot et al. 2010
W = np.random.randn(fan_in, fan_out)/np.sqrt(fan_in)
```

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.xavier_normal(w)
```

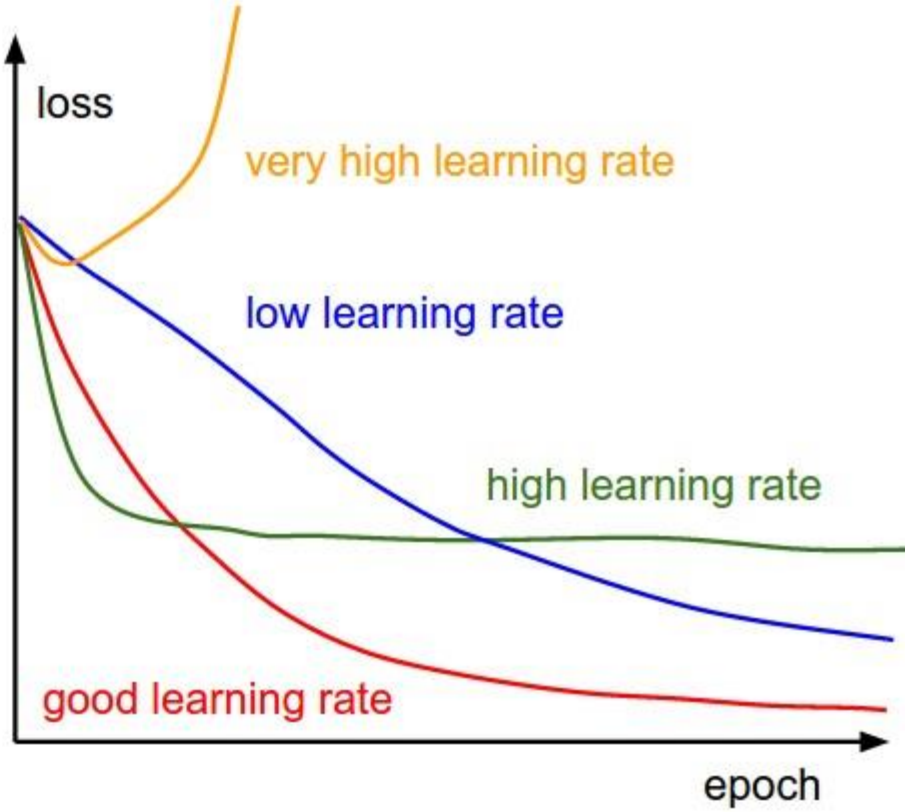
He Initialization

ReLU를 사용할 때 gradient 값이
절반 정도는 0으로 바뀌는걸 감안
하여 적용한 초기값

```
# He et al. 2015
W = np.random.randn(fan_in, fan_out)/np.sqrt(fan_in/2)
```

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.kaiming_normal(w, mode='fan_out')
```

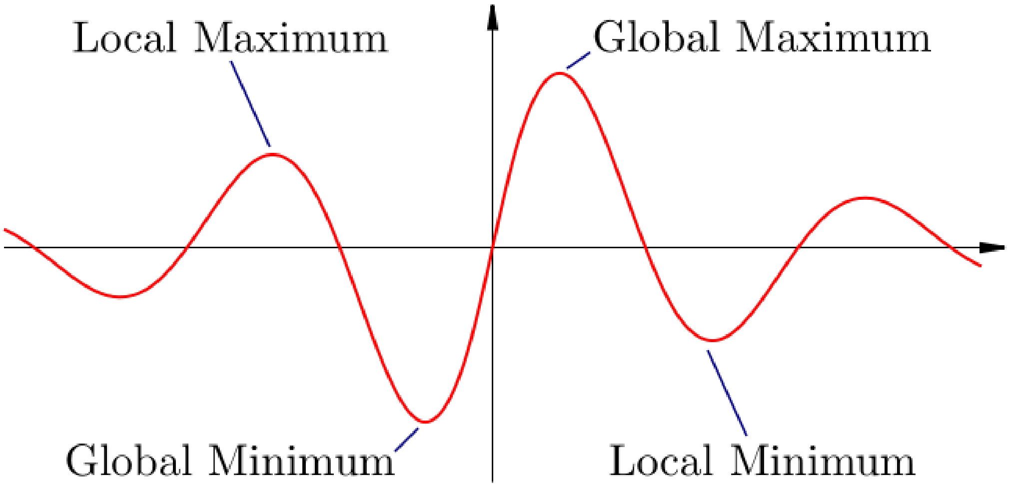
Convergence



초기값을 적절히 잡았다고
해도 learning rate가 문제

(출처:https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html)

Convergence



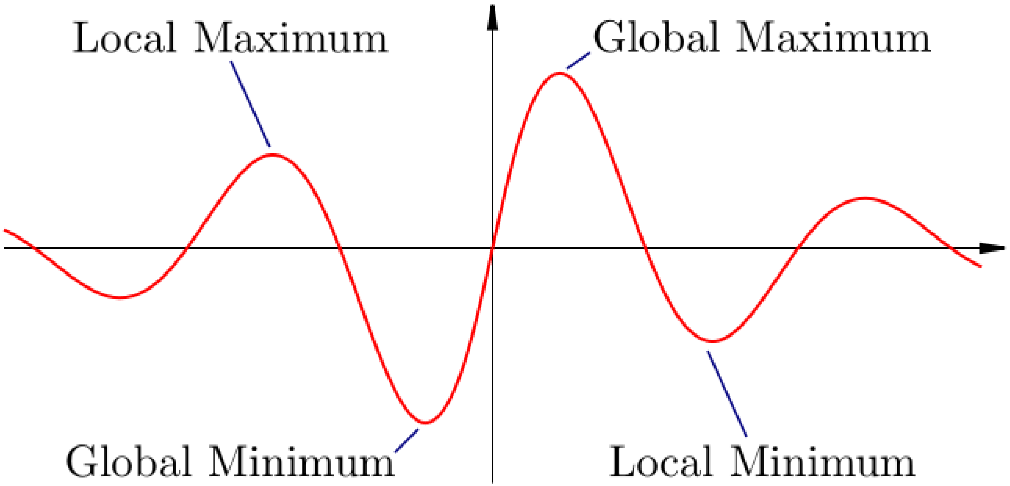
초기값을 적절히 잡았다고 해도 learning rate가 문제

⇩

너무 크면 수렴을 못하고 너무 작으면 local minima에 빠져서 최선의 결과를 못 냄

(출처:https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html)

Convergence



초기값을 적절히 잡았다고 해도 learning rate가 문제

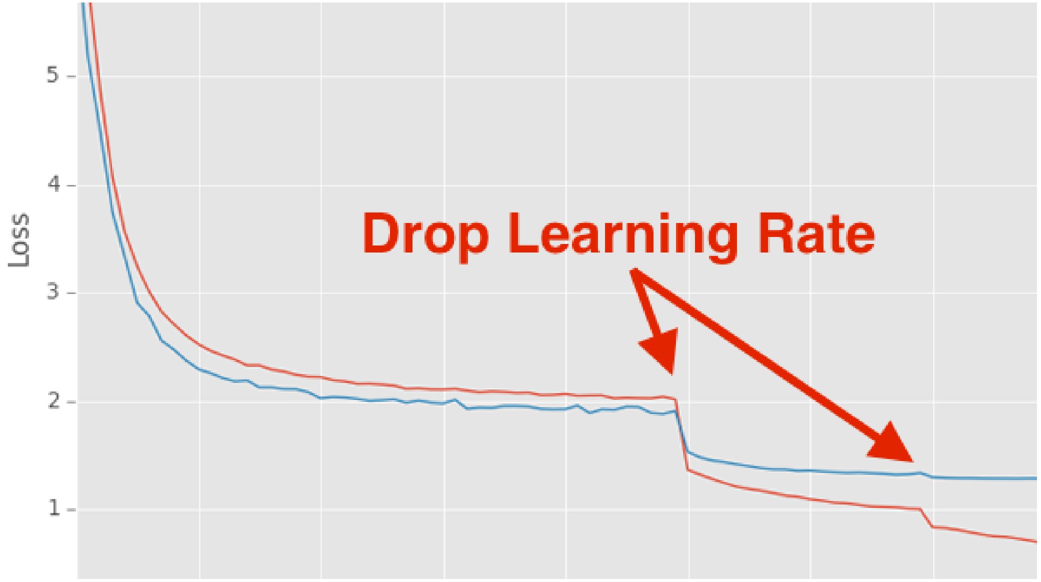
⇩

너무 크면 수렴을 못하고 너무 작으면 local minima에 빠져서 최선의 결과를 못 냄

⇩

Learning Rate Decay

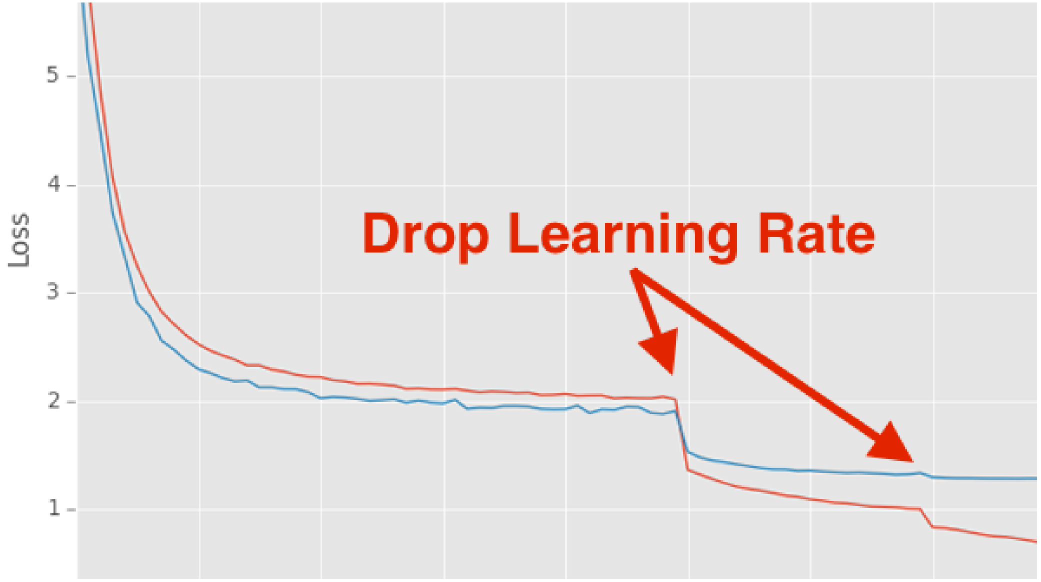
Convergence



Learning Rate Decay

(출처:https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html)

Convergence

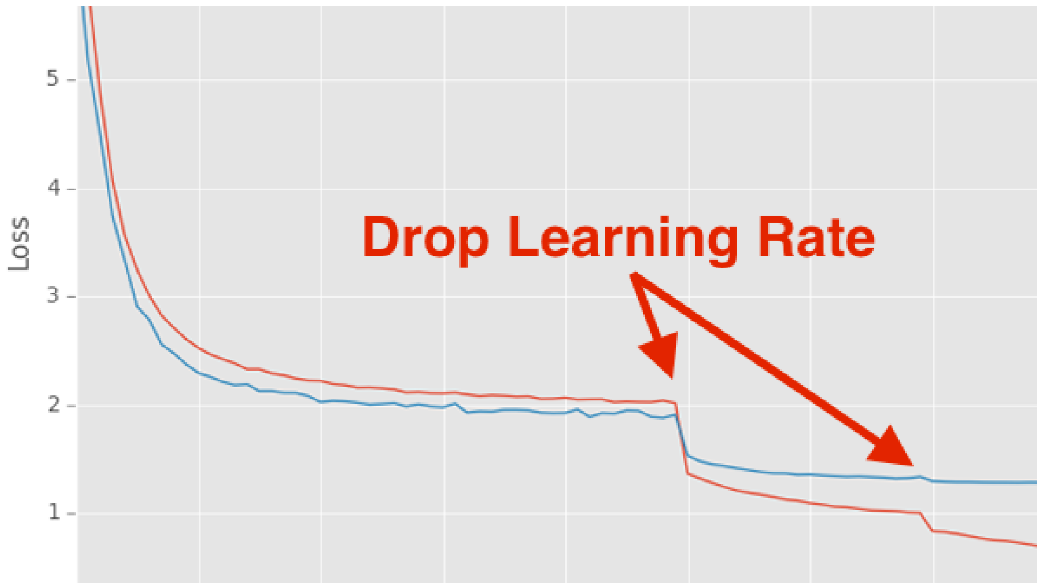


Learning Rate Decay

learning rate를 점차
떨어뜨리는 방법

(출처:https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html)

Convergence



Learning Rate Decay

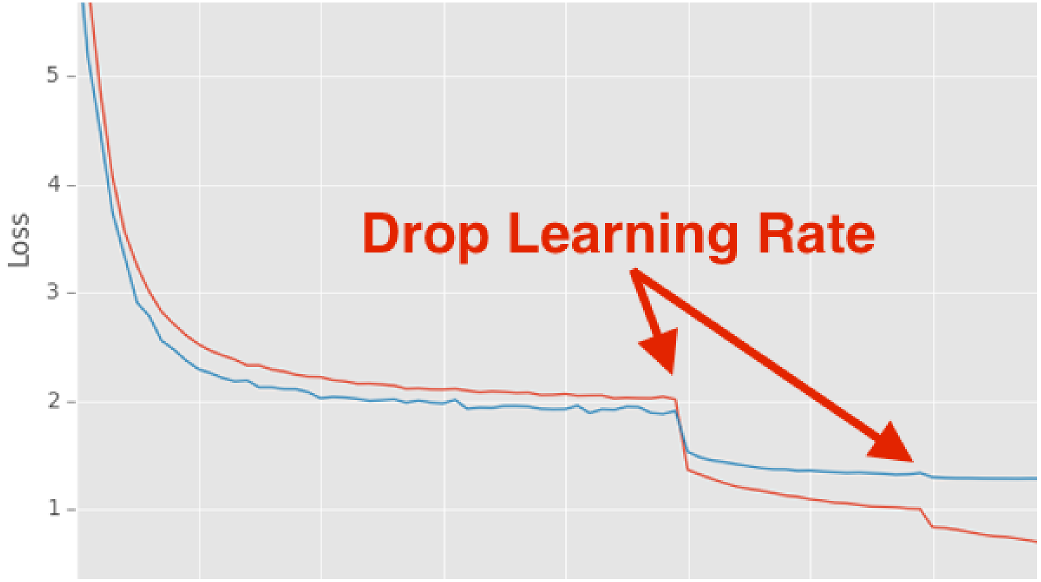
learning rate를 점차
떨어뜨리는 방법

⇩

처음에는 크게 크게 업데이트
하다가 점차 global minimum에
가까워질수록 learning rate를
낮춰서 수렴하도록 함

(출처:https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html)

Convergence



`class torch.optim.lr_scheduler.StepLR(optimizer, step_size, gamma=0.1, last_epoch=-1)` [source]

`class torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma, last_epoch=-1)` [source]

Learning Rate Decay

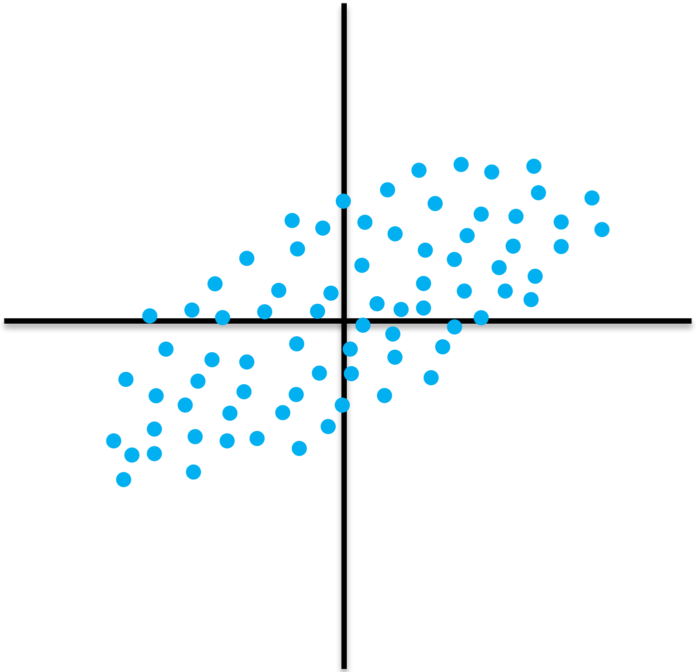
learning rate를 점차 떨어뜨리는 방법

⇩

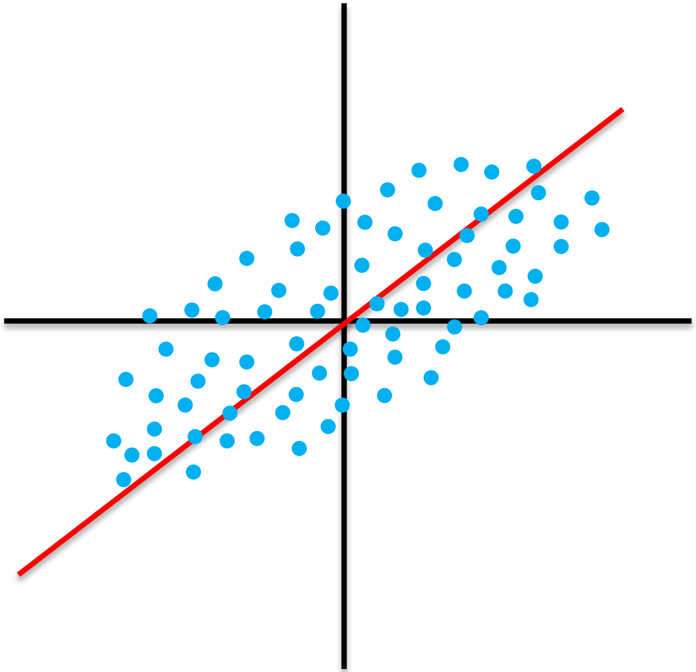
처음에는 크게 크게 업데이트 하다가 점차 global minimum에 가까워질수록 learning rate를 낮춰서 수렴하도록 함

(출처:https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html)

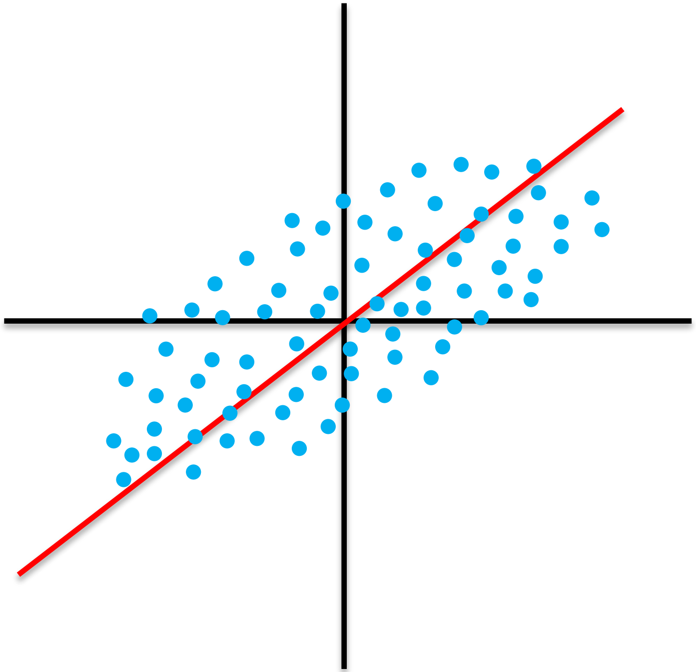
Convergence



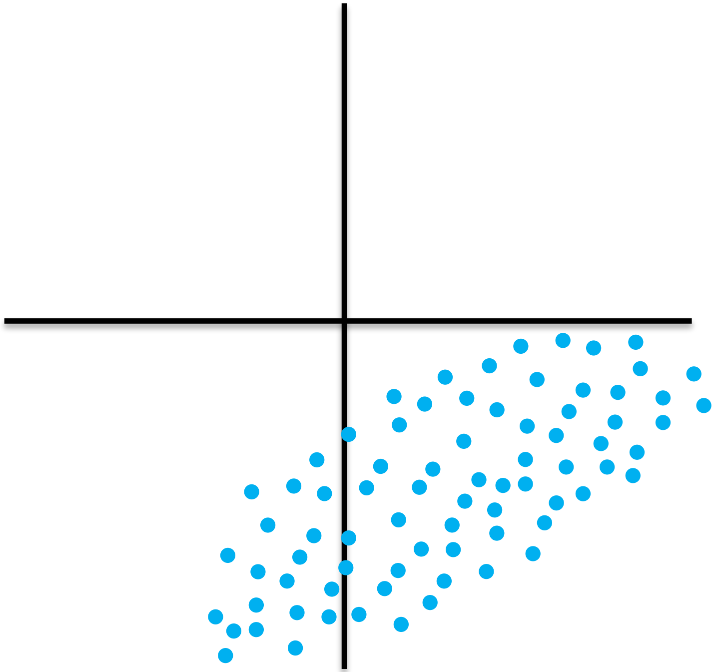
Convergence



Convergence

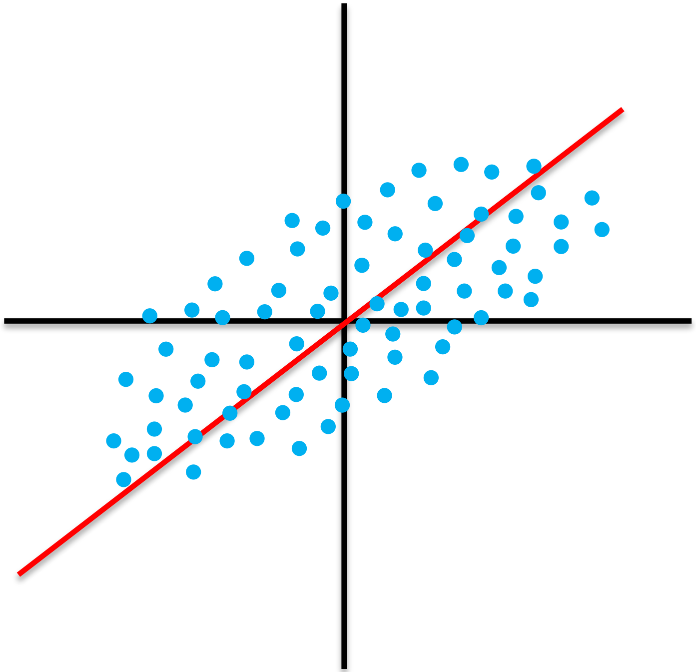


학습 데이터1

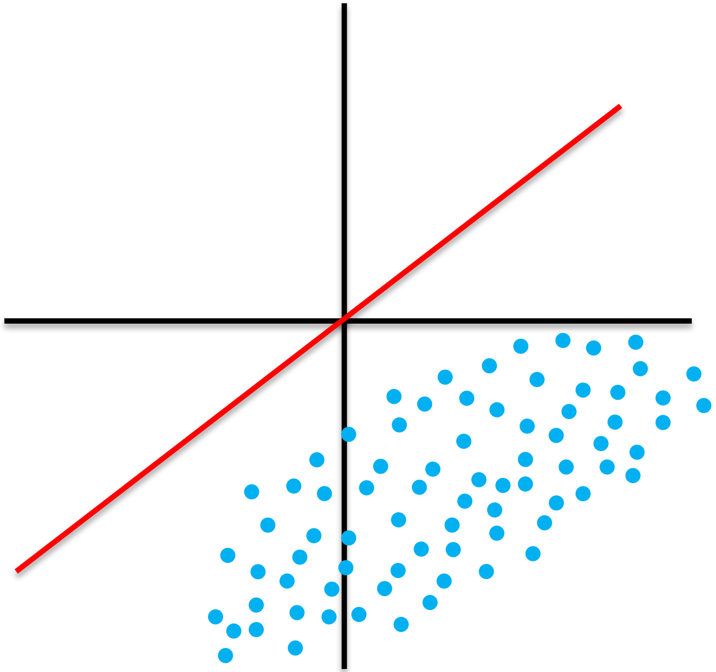


학습 데이터2

Convergence

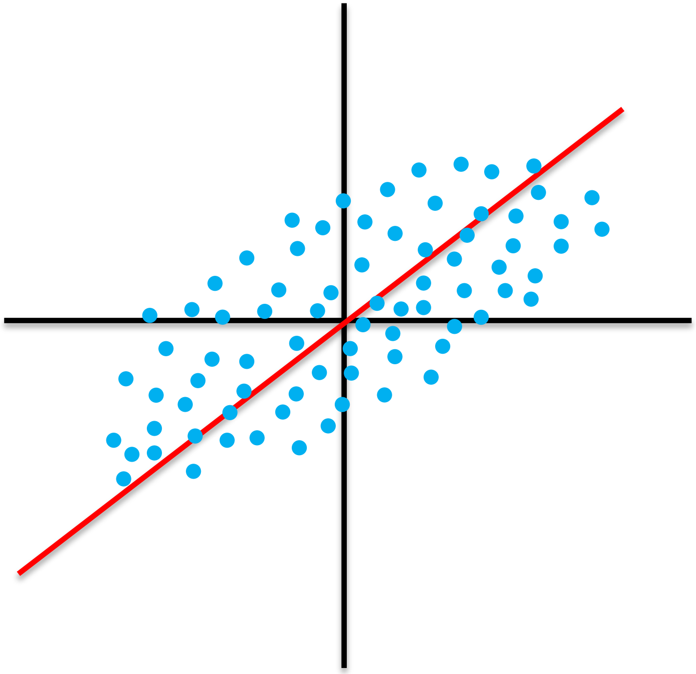


학습 데이터1

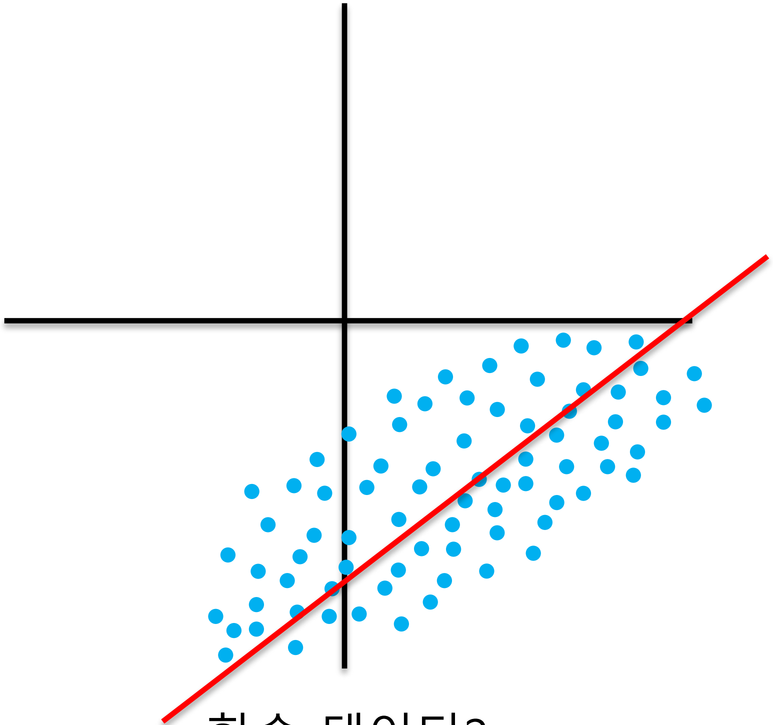


학습 데이터2

Convergence



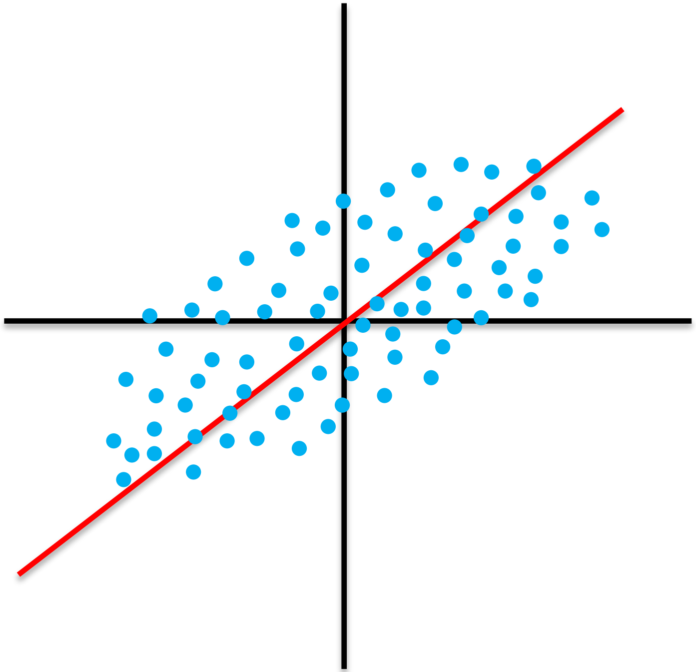
학습 데이터1



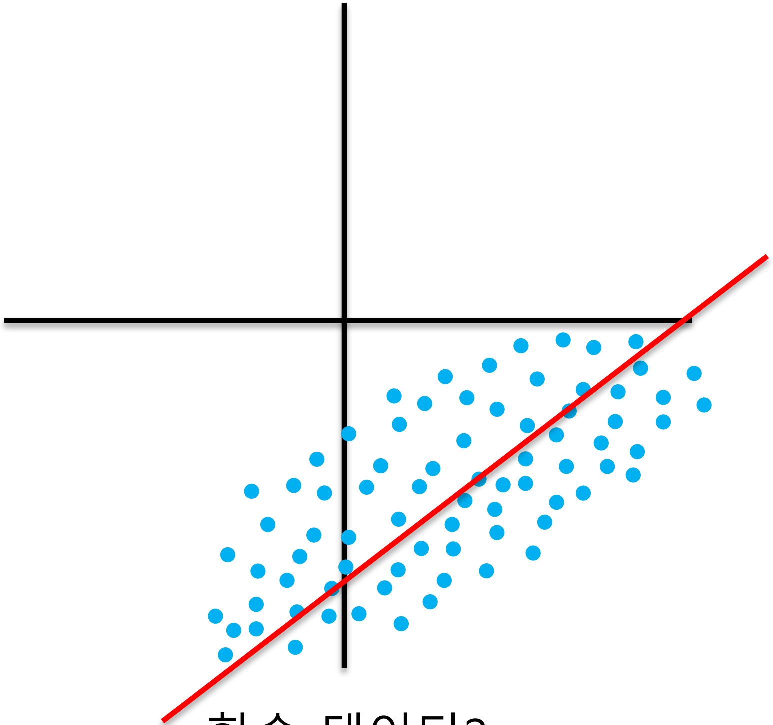
학습 데이터2

Convergence

학습이 제대로
이루어지지 못함

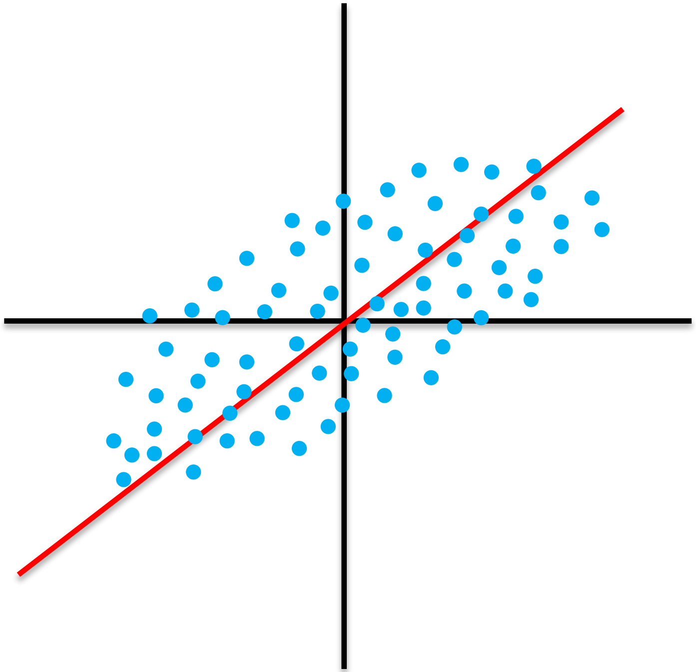


학습 데이터1

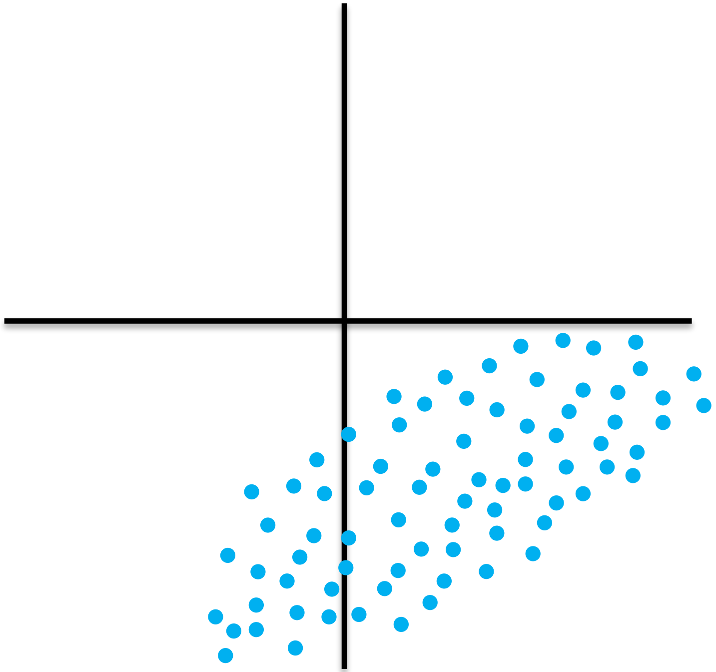


학습 데이터2

Convergence

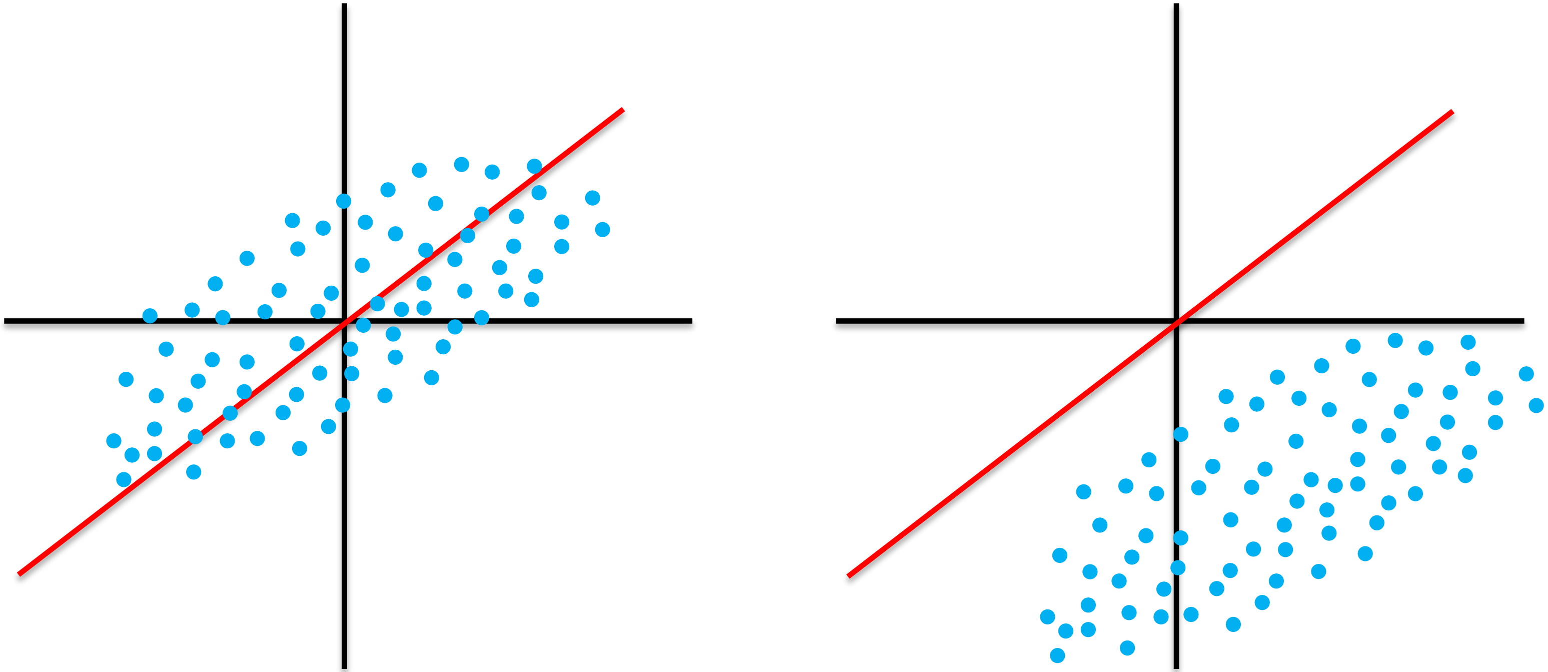


학습 데이터



테스트 데이터

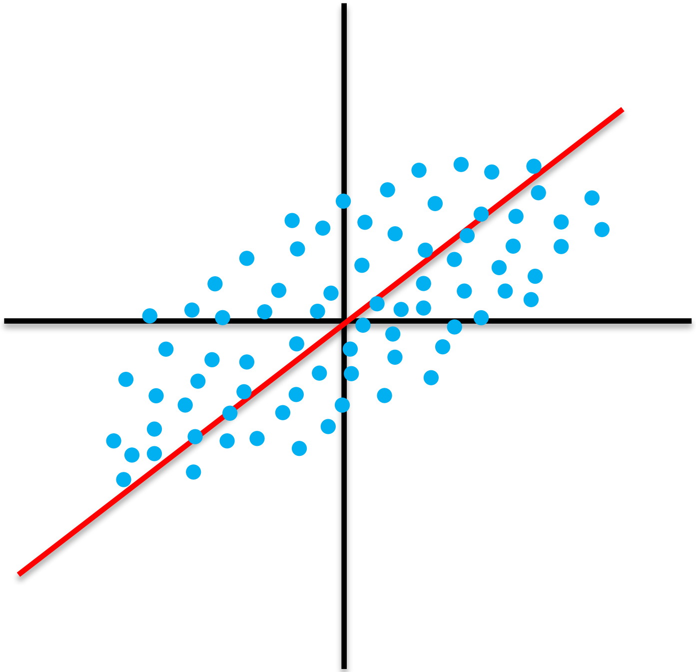
Convergence



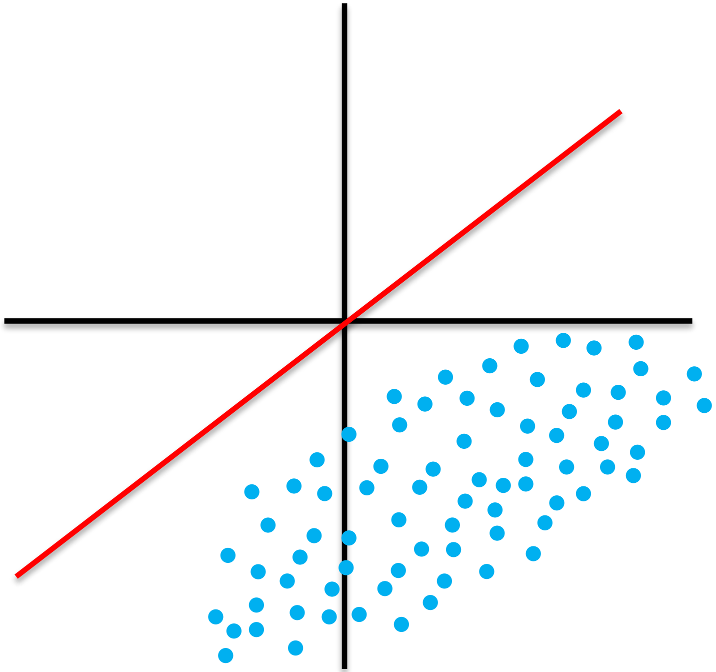
학습 데이터 테스트 데이터

Convergence

테스트에서도
정확도가 떨어짐



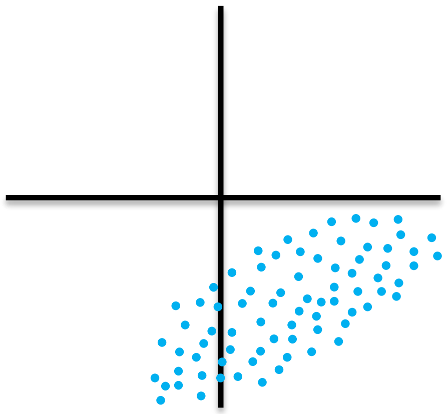
학습 데이터



테스트 데이터

Convergence

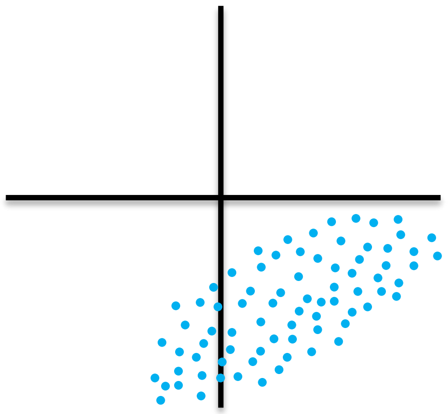
Normalize Data



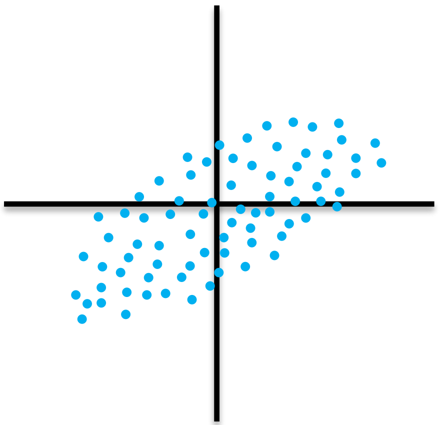
Unnormalized

Convergence

Normalize Data



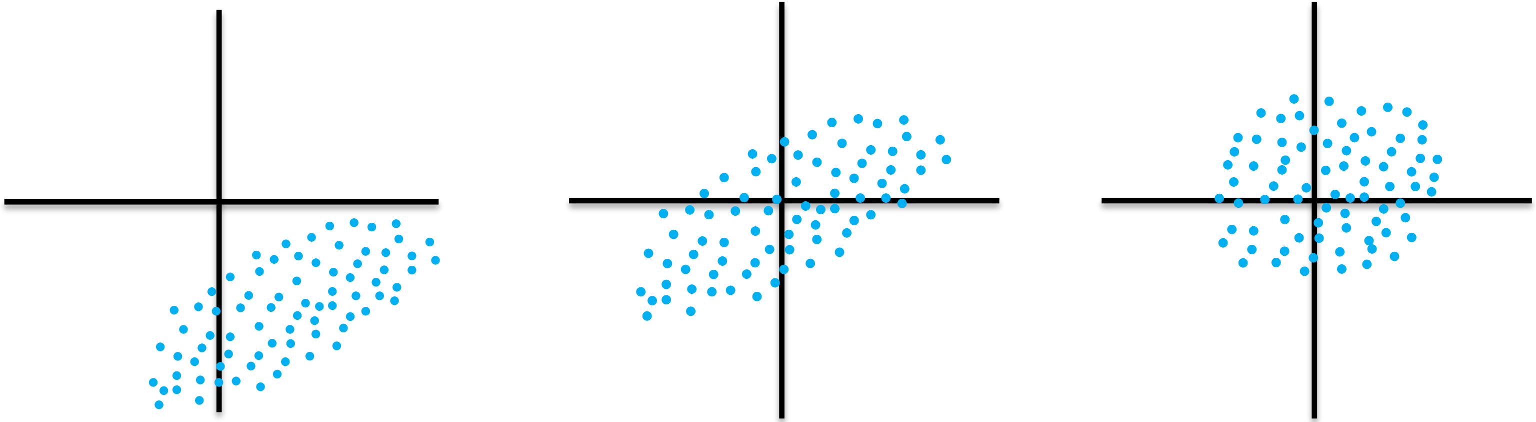
Unnormalized



$x' = x - \mu$

Convergence

Normalize Data



Unnormalized

$$x' = x - \mu$$

$$x' = \frac{x - \mu}{\sigma}$$

Convergence

```python
def image_preprocess(img_dir):
        img = Image.open(img_dir)
        transform = transforms.Compose([
                                        transforms.Scale(image_size),
                                        transforms.CenterCrop(image_size),
                                        transforms.ToTensor(),
                                        transforms.Normalize(mean=[0.40760392, 0.45795686, 0.48501961],
                                         std=[1,1,1]),
                                    ])
        img = transform(img).view((-1,3,image_size,image_size))
        return img
```

Convergence

ImageNet data
RGB mean

```
def image_preprocess(img_dir):
        img = Image.open(img_dir)
        transform = transforms.Compose([
                                        transforms.Scale(image_size),
                                        transforms.CenterCrop(image_size),
                                        transforms.ToTensor(),
                                        transforms.Normalize(mean=[0.40760392, 0.45795686, 0.48501961],
                                         std=[1,1,1]),
                                ])
        img = transform(img).view((-1,3,image_size,image_size))
        return img
```

Convergence

Input이 normalize되었어도 layer를 거치
는 과정에서 또 shift가 일어남
ex) $y = ReLU(Wx + b)$

Convergence

Input이 normalize되었어도 layer를 거치는 과정에서 또 shift가 일어남

ex) $y = ReLU(Wx + b)$

$\Downarrow$

그렇다면 activation값들도 normalize 해주면 되지 않을까?

Convergence

Input이 normalize되었어도 layer를 거치는 과정에서 또 shift가 일어남

ex) $y = ReLU(Wx + b)$

⇩

그렇다면 activation값들도 normalize 해주면 되지 않을까?

⇩

Batch Normalization!!

Convergence

Input이 normalize되었어도 layer를 거치는 과정에서 또 shift가 일어남

ex) $y = ReLU(Wx + b)$

$\Downarrow$

그렇다면 activation값들도 normalize 해주면 되지 않을까?

$\Downarrow$

Batch Normalization!!

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Convergence

Input이 normalize되었어도 layer를 거치는 과정에서 또 shift가 일어남

ex) $y = ReLU(Wx + b)$

⇩

그렇다면 activation값들도 normalize 해주면 되지 않을까?

⇩

Batch Normalization!!

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

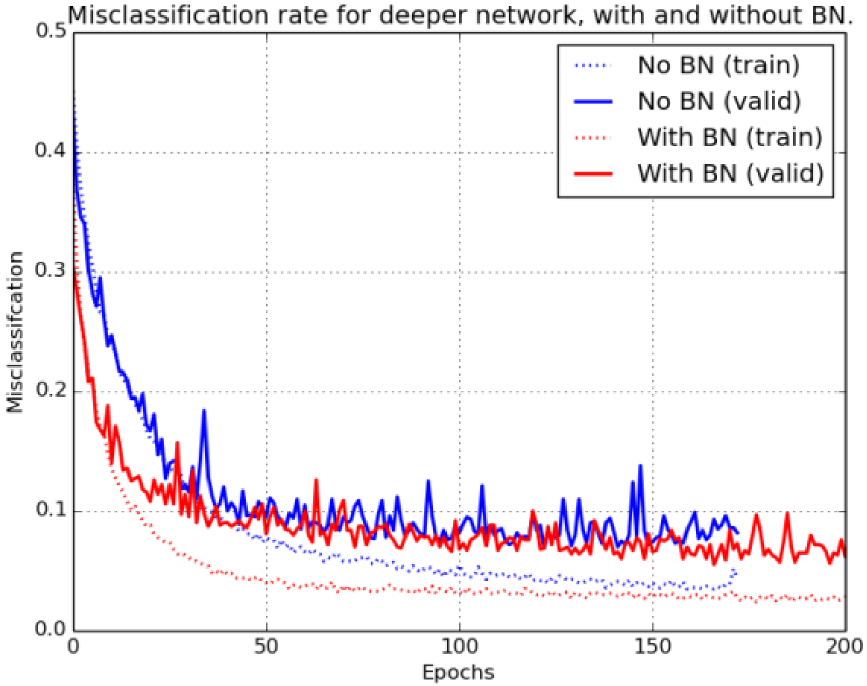
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

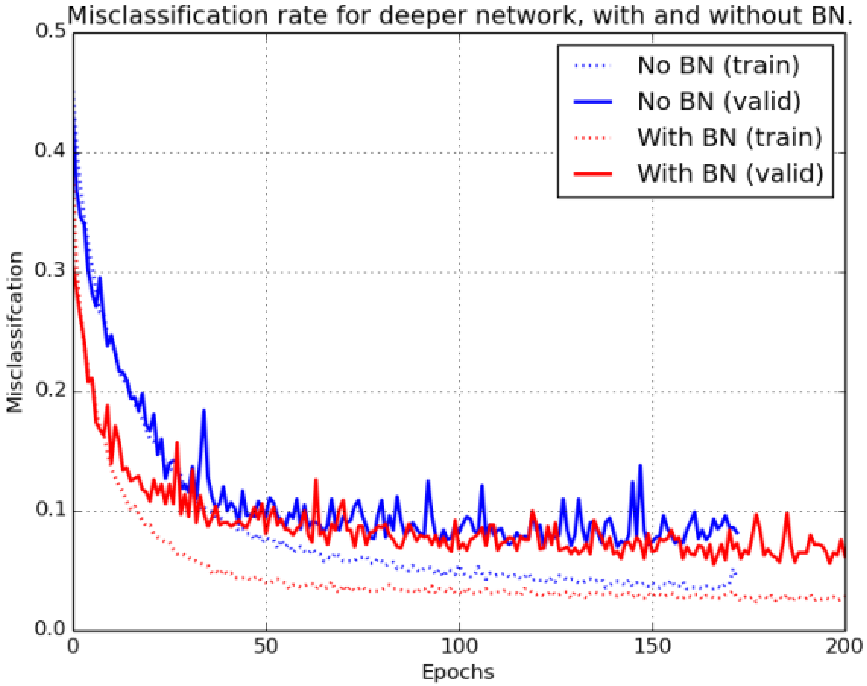
batch 단위로 mean, std 계산
normalize 후 scale, shift 적용

Convergence



(출처:https://cesarlaurent.wordpress.com/)

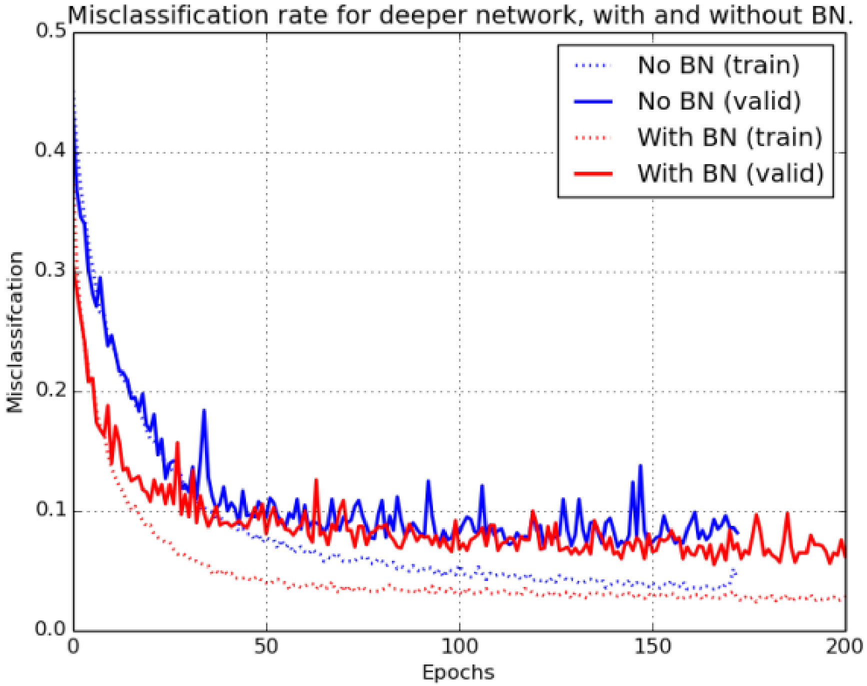
Convergence



속도, 정확도면에서 향상이 있음

(출처:https://cesarlaurent.wordpress.com/)

Convergence



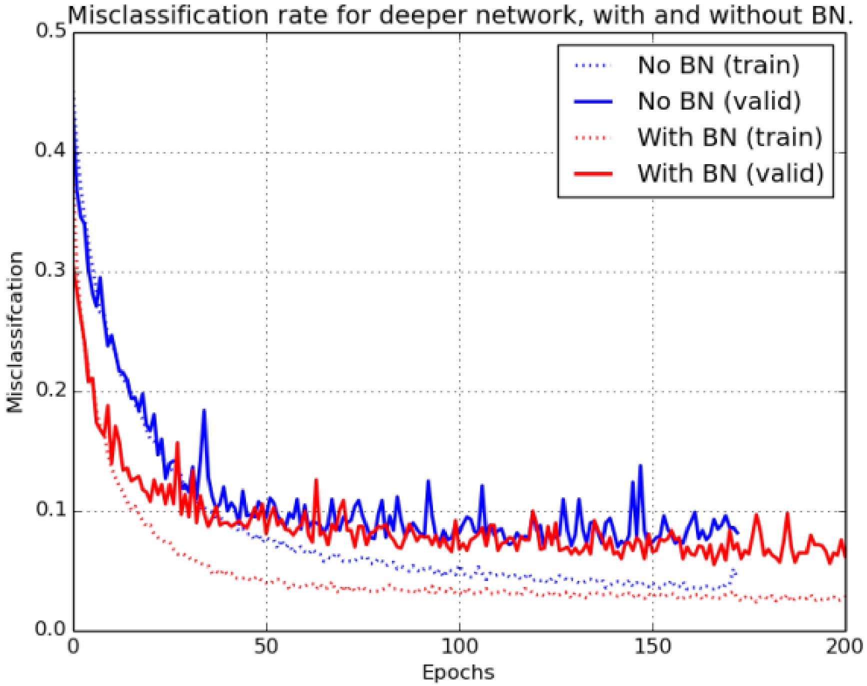
속도, 정확도면에서 향상이 있음

⇩

기본적으로 BN을 사용함

(출처:https://cesarlaurent.wordpress.com/)

Convergence



속도, 정확도면에서 향상이 있음

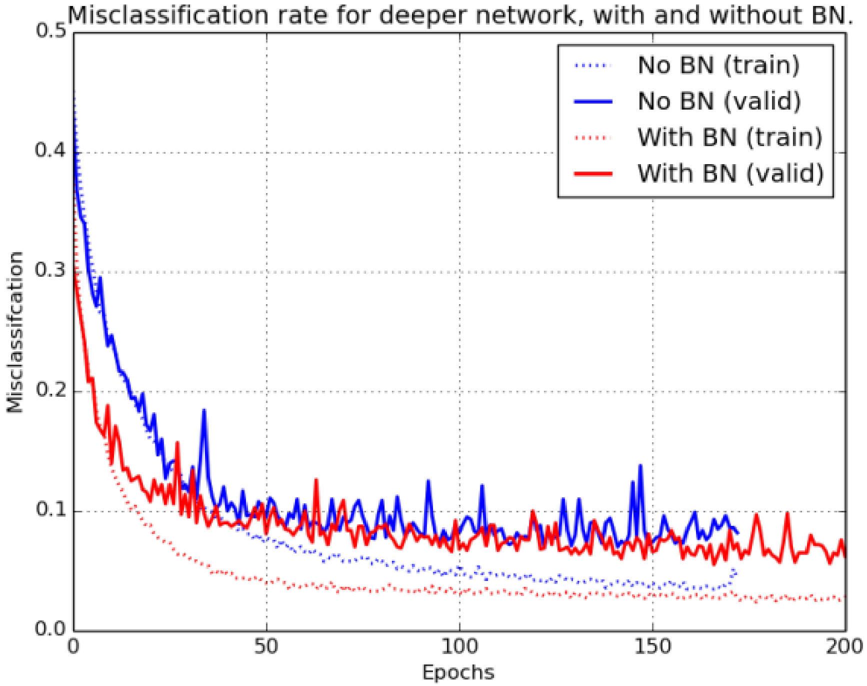
⇩

기본적으로 BN을 사용함

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm1d(100)
```

(출처:https://cesarlaurent.wordpress.com/)

Convergence



속도, 정확도면에서 향상이 있음

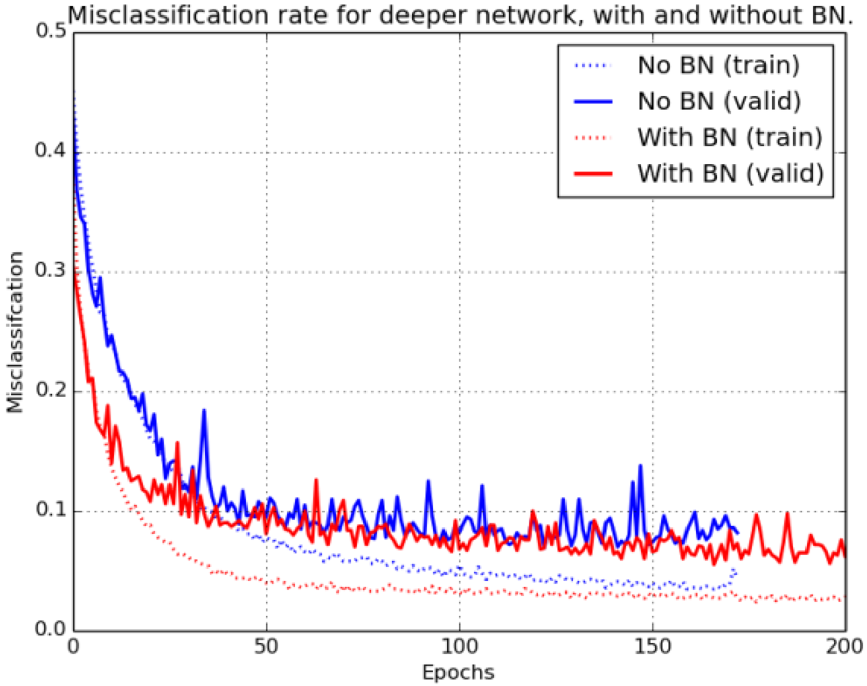
⇩

기본적으로 BN을 사용함

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm2d(100)
```

(출처:https://cesarlaurent.wordpress.com/)

Convergence



속도, 정확도면에서 향상이 있음

⇩

기본적으로 BN을 사용함

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm2d(100)
```

필터 수

(출처:https://cesarlaurent.wordpress.com/)

Convergence

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

BN은 Regularization 효과도 있다고 알려져 있는데 어떻게 가능한 걸까?

Convergence

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

BN은 Regularization 효과도 있다고 알려져 있는데 어떻게 가능한 걸까?

⇩

$x = \begin{bmatrix} 2 & 6 \\ 15 & 1 \end{bmatrix}$ 면 $\mu = 6, \sigma = 6.37704$

$$\hat{x} = x - \frac{6}{\sigma}$$

Convergence

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

BN은 Regularization 효과도 있다고 알려져 있는데 어떻게 가능한 걸까?

$$x = \begin{bmatrix} 2 & 6 \\ 15 & 1 \end{bmatrix} \text{ 면 } \mu = 6, \sigma = 6.37704$$

$$\hat{x} = x - \frac{6}{\sigma}$$

$$\hat{x} = \begin{bmatrix} -0.62725 & 0 \\ 1.31141 & -0.78406 \end{bmatrix}$$

Convergence

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

BN은 Regularization 효과도 있다고 알려져 있는데 어떻게 가능한 걸까?

⇩

$x = \begin{bmatrix} 2 & 6 \\ 15 & 1 \end{bmatrix}$ 면 $\mu = 6, \sigma = 6.37704$

$$\hat{x} = x - \frac{6}{\sigma}$$

$$\hat{x} = \begin{bmatrix} -0.62725 & 0 \\ 1.31141 & -0.78406 \end{bmatrix}$$

값들이 normalize되면서 전체적으로 weight값들이 작아졌음을 확인할 수 있음

Optimization

모델이 수렴을 좀 더 잘하게 하는 방법들

Optimization

모델이 수렴을 좀 더 잘하게 하는 방법들

⇩

그러면 모델이 더 빠르게 수렴하는 방법은 없을까?

Optimization

모델이 수렴을 좀 더 잘하게 하는 방법들

⇩

그러면 모델이 더 빠르게 수렴하는 방법은 없을까?

⇩

SGD, Momentum, Nestrov, Adagrad, RMSProp, Adam, ...

Optimization

Gradient Descent

Data 전체에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

Optimization

Gradient Descent

Data 전체에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

⇩

그런데 데이터가 백만개라면
백만번 돌리고 한번 업데이트됨

Optimization

Gradient Descent

Data 전체에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

⇩

그런데 데이터가 백만개라면
백만번 돌리고 한번 업데이트됨

정확하긴 하지만 너무 오래 걸림

Optimization

Gradient Descent

Data 전체에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

⇩

그런데 데이터가 백만개라면
백만번 돌리고 한번 업데이트됨

정확하긴 하지만 너무 오래 걸림

Stochastic Gradient Descent

데이터 하나당에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

Optimization

Gradient Descent	Stochastic Gradient Descent
Data 전체에 대한 Loss를 구하고 이에 대한 gradient로 업데이트	데이터 하나당에 대한 Loss를 구하고 이에 대한 gradient로 업데이트
⇩	⇩
그런데 데이터가 백만개라면 백만번 돌리고 한번 업데이트됨	데이터가 백만개면 한 epoch당 백만번 업데이트
정확하긴 하지만 너무 오래 걸림	

Optimization

Gradient Descent

Data 전체에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

⇩

그런데 데이터가 백만개라면
백만번 돌리고 한번 업데이트됨

정확하긴 하지만 너무 오래 걸림

Stochastic Gradient Descent

데이터 하나당에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

⇩

데이터가 백만개면 한 epoch당 백만번
업데이트

불필요한 특정 데이터 특징을 다 학습함

Optimization

Gradient Descent

Data 전체에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

⇩

그런데 데이터가 백만개라면
백만번 돌리고 한번 업데이트됨

정확하긴 하지만 너무 오래 걸림

batch_size = all

Stochastic Gradient Descent

데이터 하나당에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

⇩

데이터가 백만개면 한 epoch당 백만번
업데이트

불필요한 특정 데이터 특징을 다 학습함

batch_size = 1

Optimization

Mini-batch Gradient Descent

Gradient Descent

Data 전체에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

⇩

그런데 데이터가 백만개라면
백만번 돌리고 한번 업데이트됨

정확하긴 하지만 너무 오래 걸림

batch_size = all

Stochastic Gradient Descent

데이터 하나당에 대한 Loss를 구하고
이에 대한 gradient로 업데이트

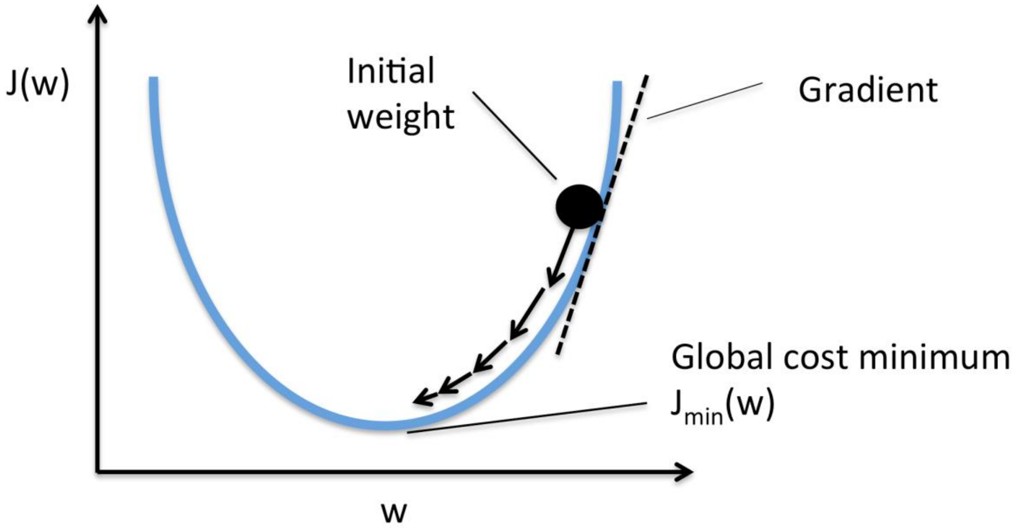
⇩

데이터가 백만개면 한 epoch당 백만번
업데이트

불필요한 특정 데이터 특징을 다 학습함

batch_size = 1

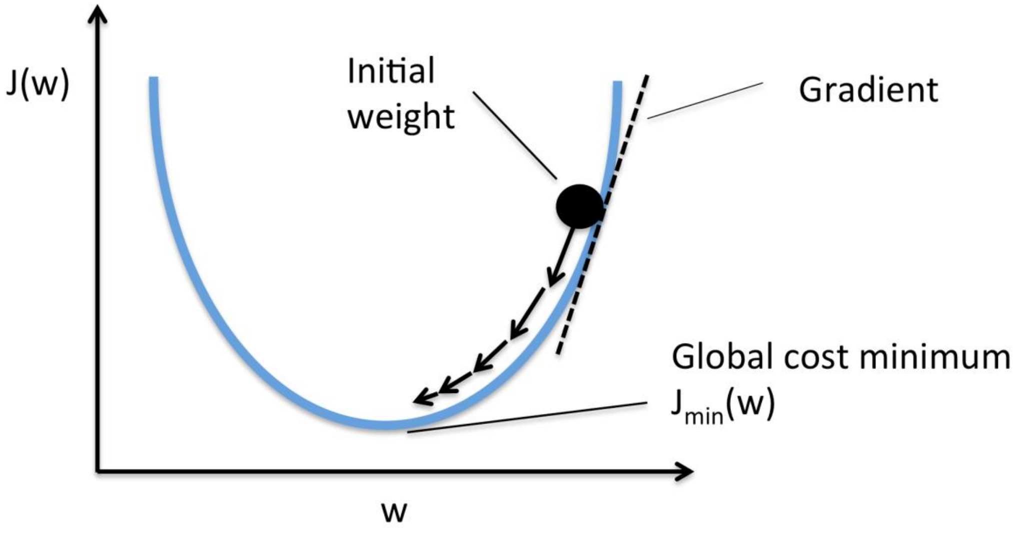
Optimization



Mini-batch Gradient Descent를
기준으로 잡고 해도 문제가 있음

(출처:https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html)

Optimization



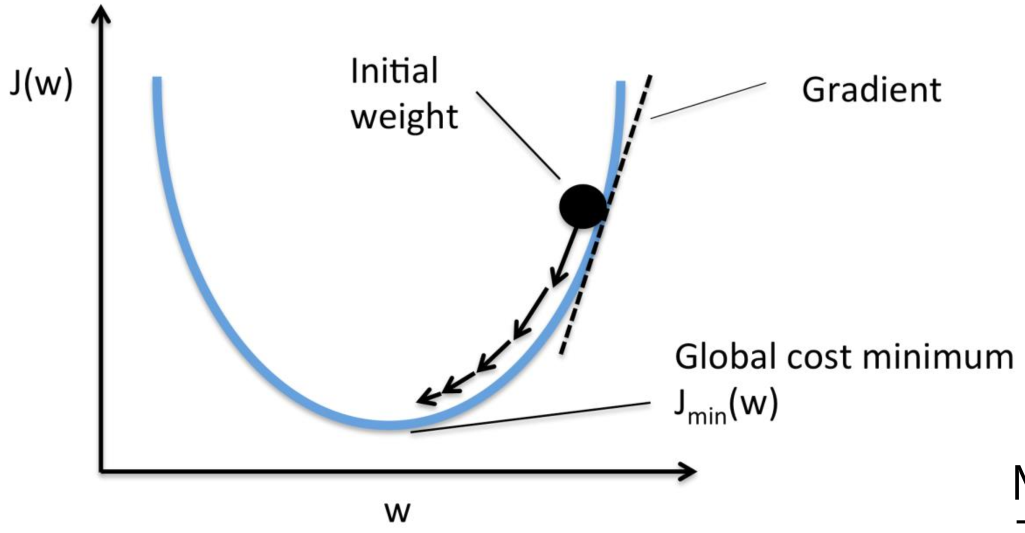
Mini-batch Gradient Descent를 기준으로 잡고 해도 문제가 있음

⇩

Convex 함수의 local minimum에 다가갈수록 gradient는 작아짐

(출처:https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html)

Optimization



Mini-batch Gradient Descent를
기준으로 잡고 해도 문제가 있음

⇩

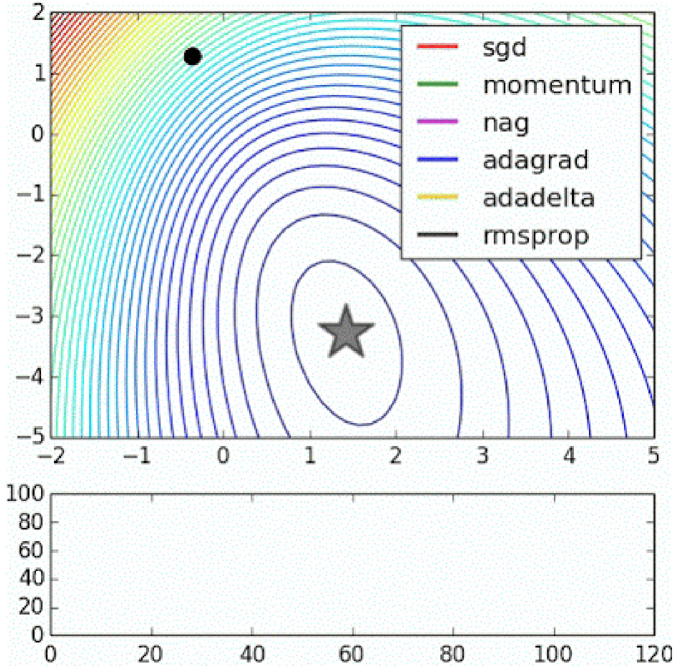
Convex 함수의 local minimum에
다가갈수록 gradient는 작아짐

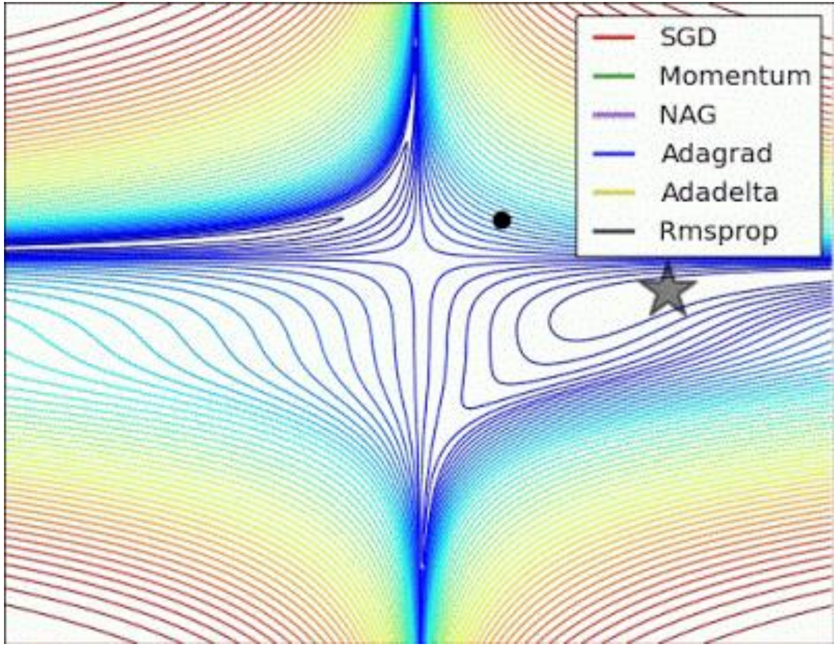
⇩

Minimum에 가까워질수록 업데이트
가 안됨. 그래서 느림

(출처:https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html)

Optimization





Optimization

Image 2: SGD without momentum

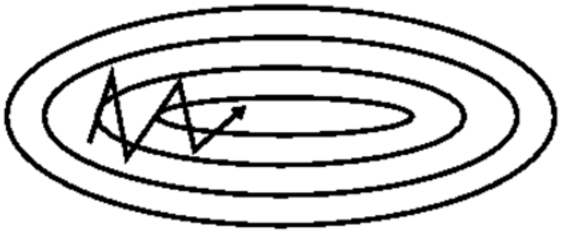


Image 3: SGD with momentum

Optimization

Momentum

Image 2: SGD without momentum

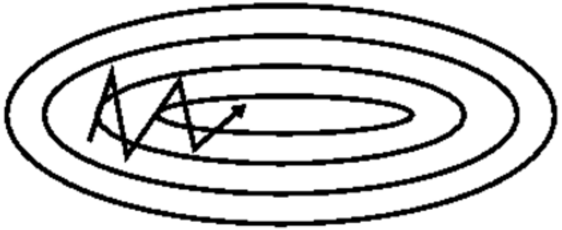


Image 3: SGD with momentum

$$\theta = \theta - lr * gradient$$

Optimization

Momentum

Image 2: SGD without momentum

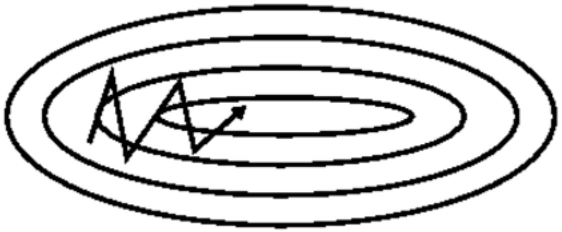


Image 3: SGD with momentum

$$\theta = \theta - lr * gradient$$

$$v_t = \gamma * v_{t-1} + lr * gradient$$

$$\theta = \theta - v_t$$

Optimization

Momentum

Image 2: SGD without momentum

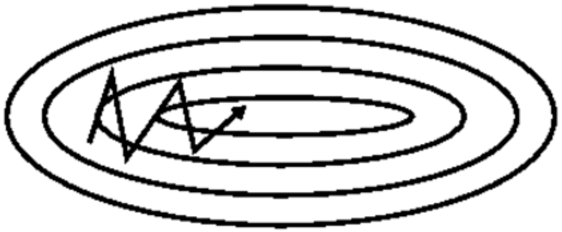


Image 3: SGD with momentum

$$\theta = \theta - lr * gradient$$

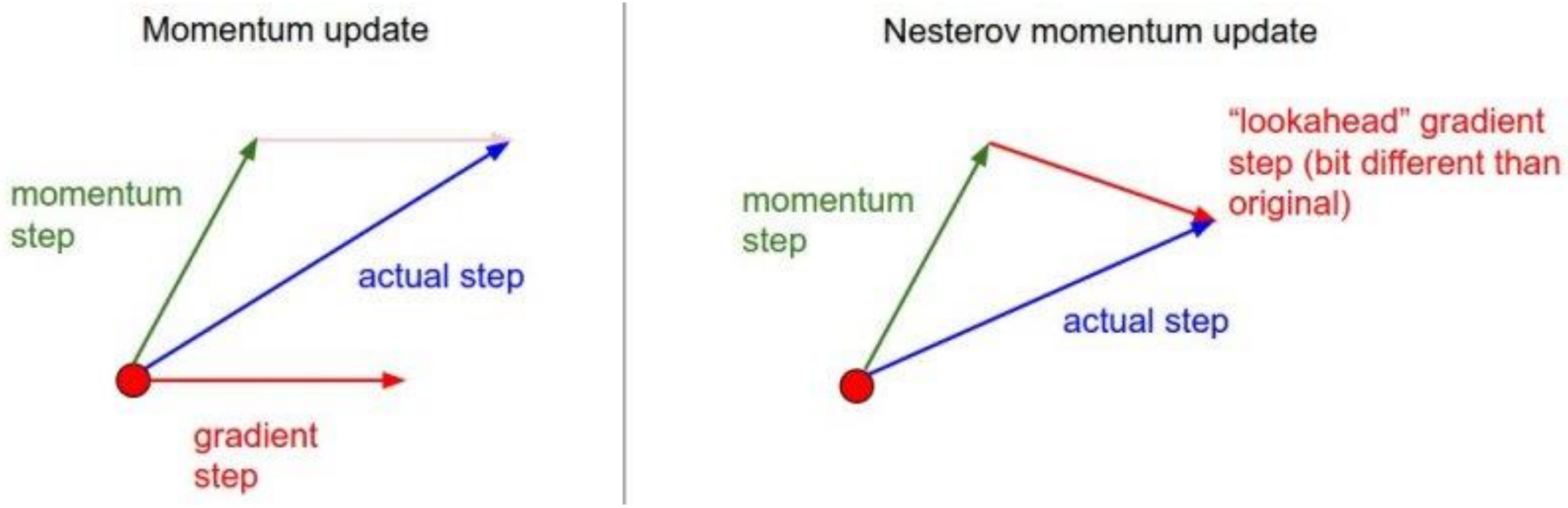
$$v_t = \underline{\gamma * v_{t-1}} + lr * gradient$$

$$\theta = \theta - v_t$$

기존의 업데이트를 일정 비율(γ)로 반영

Optimization

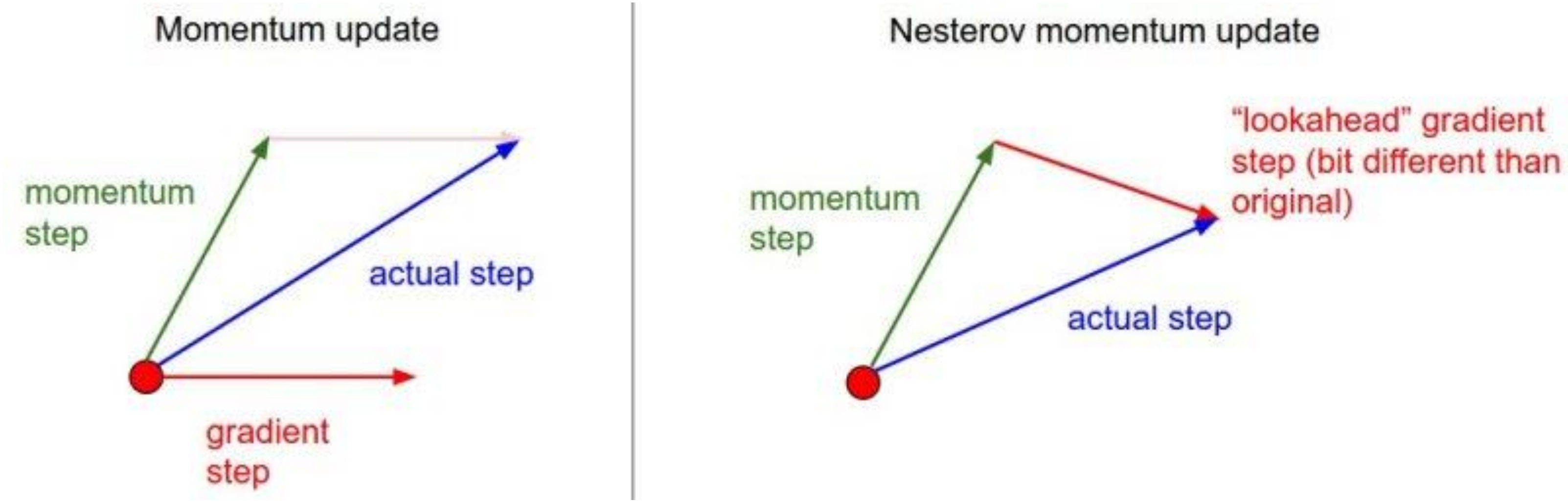
Nesterov



(출처: http://cs231n.github.io/neural-networks-3/)

Optimization

Nesterov



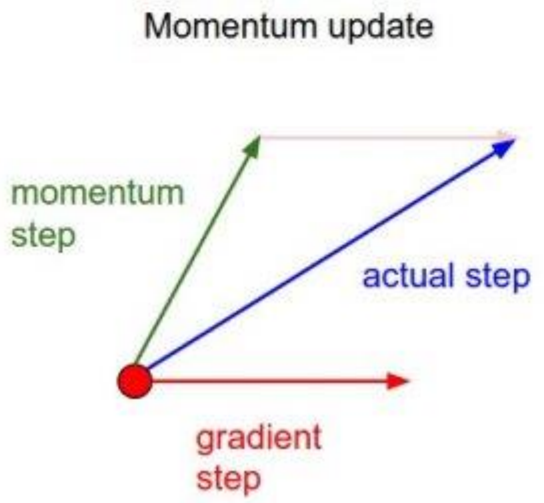
$$v_t = \gamma * v_{t-1} + lr * gradient$$

$$\theta = \theta - v_t$$

(출처: http://cs231n.github.io/neural-networks-3/)

Optimization

Nesterov



$$v_t = \gamma * v_{t-1} + lr * gradient$$

$$\theta = \theta - v_t$$

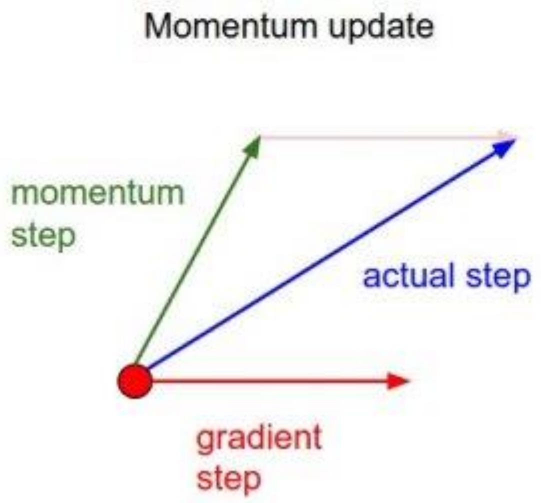
$$gradient = gradient(\theta - \gamma * v_{t-1})$$
$$v_t = \gamma * v_{t-1} + lr * gradient$$

$$\theta = \theta - v_t$$

(출처: http://cs231n.github.io/neural-networks-3/)

Optimization

Nesterov



momentum은 이미 확정
-이니까 미리 업데이트

$$v_t = \gamma * v_{t-1} + lr * gradient$$

$$\theta = \theta - v_t$$

$$gradient = gradient(\theta - \gamma * v_{t-1})$$

$$v_t = \gamma * v_{t-1} + lr * gradient$$

$$\theta = \theta - v_t$$

(출처: http://cs231n.github.io/neural-networks-3/)

Optimization

Algorithm 4 AdaGrad

Require: Global learning rate η

Require: Initial parameter θ

Initialize gradient accumulation variable $r = 0$

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

Apply interim update: $\theta \leftarrow \theta + \rho v$

Set $g = 0$

for $i = 1$ to m **do**

Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

end for

Accumulate gradient: $r \leftarrow r + g^2$ (square is applied element-wise)

Compute update: $\Delta\theta \leftarrow -\frac{\eta}{\sqrt{r}} g$ ($\frac{1}{\sqrt{r}}$ is applied element-wise)

Apply update: $\theta \leftarrow \theta + \Delta\theta_t$

end while

Adagrad

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 4 AdaGrad

Require: Global learning rate η

Require: Initial parameter θ

Initialize gradient accumulation variable $r = 0$

while Stopping criterion not met **do**

  Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$.

  Apply interim update: $\theta \leftarrow \theta + \rho v$

  Set $g = 0$

  for $i = 1$ to m **do**

    Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

  end for

  Accumulate gradient: $r \leftarrow r + g^2$ (square is applied element-wise)

  Compute update: $\Delta\theta \leftarrow -\frac{\eta}{\sqrt{r}} g$ ($\frac{1}{\sqrt{r}}$ is applied element-wise)

  Apply update: $\theta \leftarrow \theta + \Delta\theta_t$

end while

Adagrad

여태까지 gradient의 제곱을 누적하여 저장

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 4 AdaGrad

Require: Global learning rate η

Require: Initial parameter θ

Initialize gradient accumulation variable $r = 0$

while Stopping criterion not met **do**

  Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$.

  Apply interim update: $\theta \leftarrow \theta + \rho v$

  Set $g = 0$

  for $i = 1$ to m **do**

    Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

  end for

  Accumulate gradient: $r \leftarrow r + g^2$ (square is applied element-wise)

  Compute update: $\Delta\theta \leftarrow -\frac{\eta}{\sqrt{r}} g$ ($\frac{1}{\sqrt{r}}$ is applied element-wise)

  Apply update: $\theta \leftarrow \theta + \Delta\theta_t$

end while

Adagrad

여태까지 gradient의 제곱을 누적하여 저장

$$\Delta\theta = -\frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}} g_t$$

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 4 AdaGrad

Require: Global learning rate η

Require: Initial parameter θ

Initialize gradient accumulation variable $r = 0$

while Stopping criterion not met **do**

  Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

  Apply interim update: $\theta \leftarrow \theta + \rho v$

  Set $g = 0$

  for $i = 1$ to m **do**

    Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

  end for

  Accumulate gradient: $r \leftarrow r + g^2$ (square is applied element-wise)

  Compute update: $\Delta\theta \leftarrow -\frac{\eta}{\sqrt{r}} g$ ($\frac{1}{\sqrt{r}}$ is applied element-wise)

  Apply update: $\theta \leftarrow \theta + \Delta\theta_t$

end while

Adagrad

여태까지 gradient의 제곱을
누적하여 저장

$$\Delta\theta = -\frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}} g_t$$

여러 번 업데이트 된 파라미터는 조금만 업데이트,
별로 안된 파라미터는 더 많이 업데이트하는 방식

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 4 AdaGrad

Require: Global learning rate η

Require: Initial parameter θ

Initialize gradient accumulation variable $r = 0$

while Stopping criterion not met **do**

  Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

  Apply interim update: $\theta \leftarrow \theta + \rho v$

  Set $g = 0$

  for $i = 1$ to m **do**

    Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

  end for

  Accumulate gradient: $r \leftarrow r + g^2$ (square is applied element-wise)

  Compute update: $\Delta\theta \leftarrow -\frac{\eta}{\sqrt{r}} g$ ($\frac{1}{\sqrt{r}}$ is applied element-wise)

  Apply update: $\theta \leftarrow \theta + \Delta\theta_t$

end while

Adagrad

여태까지 gradient의 제곱을 누적하여 저장

$$\Delta\theta = -\frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}} g_t$$

그런데 실제로도 업데이트가 많이 필요하다면??

여러 번 업데이트 된 파라미터는 조금만 업데이트, 별로 안된 파라미터는 더 많이 업데이트하는 방식

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 8 AdaDelta

Require: Decay rate ρ, constant ϵ

Require: Initial parameter θ

Initialize accumulation variables $s = 0, r = 0$

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

Set $g = 0$

for $i = 1$ to m **do**

Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

end for

Accumulate gradient: $r \leftarrow \rho r + (1 - \rho) g^2$

Compute update: $\Delta\theta \leftarrow -\frac{\sqrt{s+\epsilon}}{\sqrt{r+\epsilon}} g$ (operation applied element-wise)

Accumulate update: $\theta \leftarrow \rho\theta + (1 - \rho)(\Delta\theta)^2$

Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

Adadelta

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 8 AdaDelta

Require: Decay rate ρ, constant ϵ

Require: Initial parameter θ

Initialize accumulation variables $s = 0, r = 0$

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

Set $g = 0$

for $i = 1$ to m **do**

Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

end for

Accumulate gradient: $r \leftarrow \rho r + (1 - \rho) g^2$

Compute update: $\Delta\theta \leftarrow -\frac{\sqrt{s+\epsilon}}{\sqrt{r+\epsilon}} g$ (operation applied element-wise)

Accumulate update: $\theta \leftarrow \rho\theta + (1 - \rho)(\Delta\theta)^2$

Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

Adadelta

누적된 gradient를 시간이 지날수록 decay

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 8 AdaDelta

Require: Decay rate ρ, constant ϵ

Require: Initial parameter θ

Initialize accumulation variables $s = 0, r = 0$

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

Set $g = 0$

for $i = 1$ to m **do**

Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

end for

Accumulate gradient: $r \leftarrow \rho r + (1 - \rho) g^2$

Compute update: $\Delta\theta \leftarrow -\frac{\sqrt{s+\epsilon}}{\sqrt{r+\epsilon}} g$ (operation applied element-wise)

Accumulate update: $\theta \leftarrow \rho\theta + (1 - \rho)(\Delta\theta)^2$

Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

Adadelta

누적된 gradient를 시간이 지날수록 decay

$$\mathrm{E}(g^2)_t = \rho \mathrm{E}(g^2)_{t-1} + (1-\rho) g_t^2$$

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 8 AdaDelta

Require: Decay rate ρ, constant ϵ

Require: Initial parameter θ

Initialize accumulation variables $s = 0, r = 0$

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

Set $g = 0$

for $i = 1$ to m **do**

Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

end for

Accumulate gradient: $r \leftarrow \rho r + (1 - \rho) g^2$

Compute update: $\Delta\theta \leftarrow -\frac{\sqrt{s+\epsilon}}{\sqrt{r+\epsilon}} g$ (operation applied element-wise)

Accumulate update: $\theta \leftarrow \rho\theta + (1 - \rho)(\Delta\theta)^2$

Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

Adadelta

누적된 gradient를 시간이 지날수록 decay

$$\mathrm{E}(g^2)_t = \rho \mathrm{E}(g^2)_{t-1} + (1 - \rho) g_t^2$$

예전에 많이 update된 para -meter라도 시간이 지나면 또 업데이트 되도록 함

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 8 AdaDelta

Require: Decay rate ρ, constant ϵ

Require: Initial parameter θ

Initialize accumulation variables $s = 0, r = 0$

while Stopping criterion not met **do**

  Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

  Set $g = 0$

  for $i = 1$ to m **do**

    Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

  end for

  Accumulate gradient: $r \leftarrow \rho r + (1 - \rho) g^2$

  Compute update: $\Delta\theta \leftarrow -\frac{\sqrt{s+\epsilon}}{\sqrt{r+\epsilon}} g$ (operation applied element-wise)

  Accumulate update: $\theta \leftarrow \rho\theta + (1 - \rho)(\Delta\theta)^2$

  Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

RMSProp

≈

Adadelta

누적된 gradient를 시간이 지날수록 decay

$$\mathrm{E}(g^2)_t = \rho \mathrm{E}(g^2)_{t-1} + (1 - \rho) g_t^2$$

예전에 많이 update된 para-meter라도 시간이 지나면 또 업데이트 되도록 함

(출처: http://prinsphield.github.io/image/optimization/AdaGrad.jpg)

Optimization

Algorithm 7 Adam

Require: Step size η
Require: Decay rates ρ_1 and ρ_2, constant ϵ
Require: Initial parameter θ
Initialize 1st and 2nd moment variables $s = 0, r = 0$.
Initialize timestep $t = 0$
while Stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.
  Set $g = 0$
  for $i = 1$ to m **do**
    Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

  end for
  $t \leftarrow t + 1$
  Get biased first moment: $s \leftarrow \rho_1 s + (1 - \rho_1) g$
  Get biased second moment: $r \leftarrow \rho_2 r + (1 - \rho_2) g^2$
  Compute biased-corrected first moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$
  Compute biased-corrected second moment: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
  Compute update: $\Delta\theta \leftarrow -\frac{\eta s}{\sqrt{r}+\epsilon} g$ (operation applied element-wise)
  Apply update: $\theta \leftarrow \theta + \Delta\theta$
end while

Adam

Optimization

Algorithm 7 Adam

Require: Step size η

Require: Decay rates ρ_1 and ρ_2, constant ϵ

Require: Initial parameter θ

Initialize 1st and 2nd moment variables $s = 0, r = 0$.

Initialize timestep $t = 0$

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

Set $g = 0$

for $i = 1$ to m **do**

Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

end for

$t \leftarrow t + 1$

Get biased first moment: $s \leftarrow \rho_1 s + (1 - \rho_1)g$

Get biased second moment: $r \leftarrow \rho_2 r + (1 - \rho_2)g^2$

Compute biased-corrected first moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$

Compute biased-corrected second moment: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$

Compute update: $\Delta\theta \leftarrow -\frac{\eta s}{\sqrt{r}+\epsilon} g$ (operation applied element-wise)

Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

Adam

= RMSProp + Momentum $+\alpha$

Optimization

Algorithm 7 Adam

Require: Step size η

Require: Decay rates ρ_1 and ρ_2, constant ϵ

Require: Initial parameter θ

Initialize 1st and 2nd moment variables $s = 0, r = 0$.

Initialize timestep $t = 0$

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

Set $g = 0$

for $i = 1$ to m **do**

Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

end for

$t \leftarrow t + 1$

Get biased first moment: $s \leftarrow \rho_1 s + (1 - \rho_1) g$

Get biased second moment: $r \leftarrow \rho_2 r + (1 - \rho_2) g^2$

Compute biased-corrected first moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$

Compute biased-corrected second moment: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$

Compute update: $\Delta\theta \leftarrow -\frac{\eta s}{\sqrt{r}+\epsilon} g$ (operation applied element-wise)

Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

Adam

= RMSProp + Momentum +α

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

gradient의 평균과 분산에 대한 정보 둘 다를 가지고 update

Optimization

Algorithm 7 Adam

Require: Step size η

Require: Decay rates ρ_1 and ρ_2, constant ϵ

Require: Initial parameter θ

Initialize 1st and 2nd moment variables $s = 0, r = 0$.

Initialize timestep $t = 0$

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$.

Set $g = 0$

for $i = 1$ to m **do**

Compute gradient:

$$g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta)), y^{(i)}; \theta).$$

end for

$t \leftarrow t + 1$

Get biased first moment: $s \leftarrow \rho_1 s + (1 - \rho_1) g$

Get biased second moment: $r \leftarrow \rho_2 r + (1 - \rho_2) g^2$

Compute biased-corrected first moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$

Compute biased-corrected second moment: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$

Compute update: $\Delta\theta \leftarrow -\frac{\eta s}{\sqrt{r}+\epsilon} g$ (operation applied element-wise)

Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

Adam

= RMSProp + Momentum $+\alpha$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

gradient의 평균과 분산에 대한 정보 둘 다를 가지고 update

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t.$$

Optimization

내용은 복잡하지만 구현은 쉬움

class `torch.optim.Adam`(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0*) [source]

Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

Optimization

내용은 복잡하지만 구현은 쉬움

class `torch.optim.Adam`(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0*) [source]

Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*, *optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float]*, *optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float*, *optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)

```
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
```

Optimization

내용은 복잡하지만 구현은 쉬움

optimizer 설정

class **torch.optim.Adam**(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0*) [source]

Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

```
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
```

Optimization

내용은 복잡하지만 구현은 쉬움

class torch.optim.Adam(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*) [source]

Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*, *optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float]*, *optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float*, *optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)

```
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
```

optimizer 설정

⇩

loss 계산

Optimization

내용은 복잡하지만 구현은 쉬움

class **torch.optim.Adam**(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0*) [source]

Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

```
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
```

optimizer 설정

⇩

loss 계산

⇩

gradient 계산

Optimization

내용은 복잡하지만 구현은 쉬움

class **torch.optim.Adam**(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0*) [source]

Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

```
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
```

optimizer 설정

⇩

loss 계산

⇩

gradient 계산

⇩

optimizer 설정대로 update

Optimization

내용은 복잡하지만 구현은 쉬움

class **torch.optim.Adam**(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0*) [source]

Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

```
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
```

optimizer 설정

⇩

loss 계산

⇩

gradient 계산

⇩

optimizer 설정대로 update

반복

Q&A