[7주차]
Training
Neural Networks II

1기 장세영
1기 황시은

EURON

목차

1. REVIEW

2. Optimization

3. Regularization

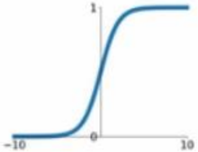
4. Transfer Learning

1. Review

Activation Functions

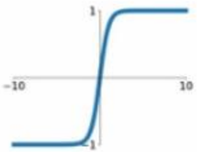
Sigmoid

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



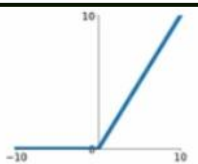
tanh

$\tanh(x)$



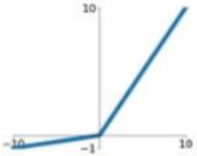
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

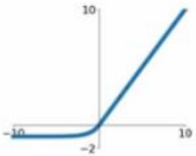


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

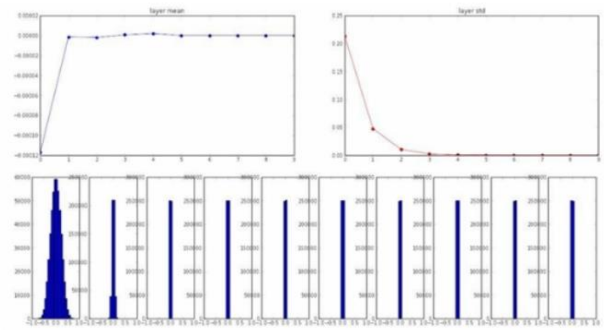
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



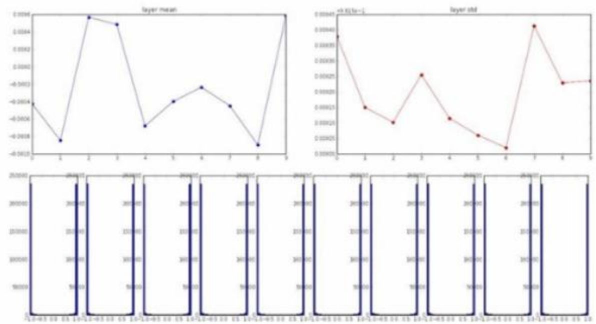
- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid

1. Review

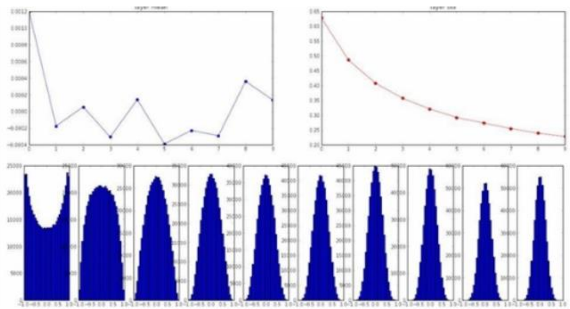
너무 작은 수



너무 큰 수



Xavier

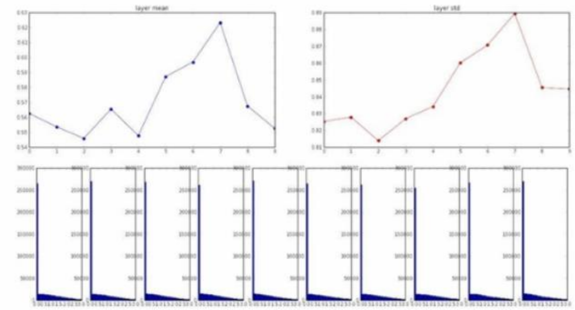


but when using the ReLU
nonlinearity it breaks.

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

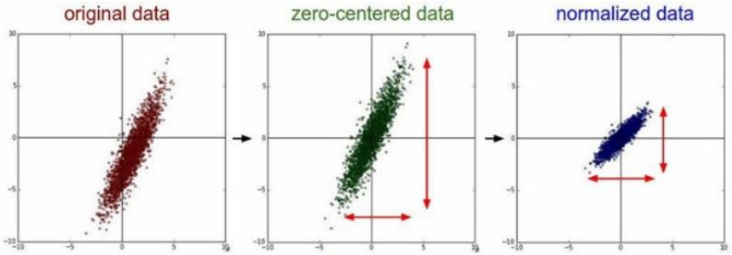
He et al., 2015
(note additional /2)



Xavier/2

1. Review

Step 1: Preprocess the data

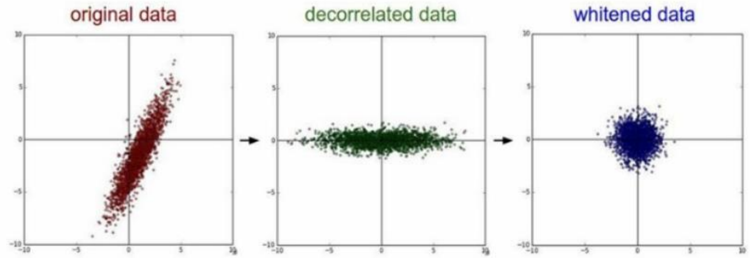


`X -= np.mean(X, axis = 0)`   `X /= np.std(X, axis = 0)`

(Assume X [NxD] is data matrix, each example in a row)

Step 1: Preprocess the data

In practice, you may also see **PCA** and **Whitening** of the data



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

1. Review

Last time: Batch Normalization

Input: $x : N \times D$

Learnable params:
$\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$
$\hat{x} : N \times D$

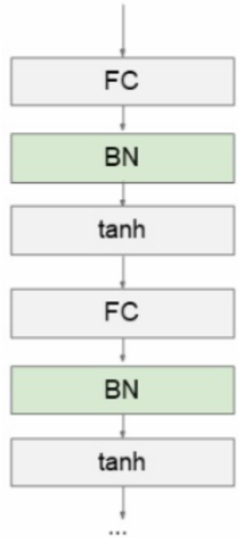
Output: $y : N \times D$

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$



output = g(WX + b) → output = g(BN(WX + b))

2. Optimization

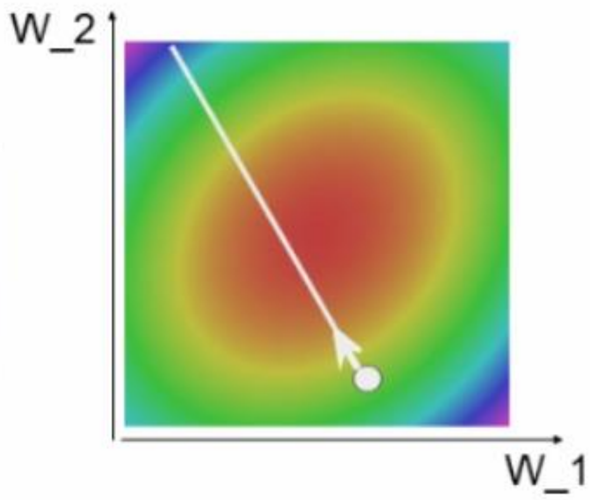
Loss를 줄이자!

한 번에 얼만큼 이동하는가? (보폭)
Learning rate

어디로 이동하는가? (방향)
gradient

2. Optimization

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

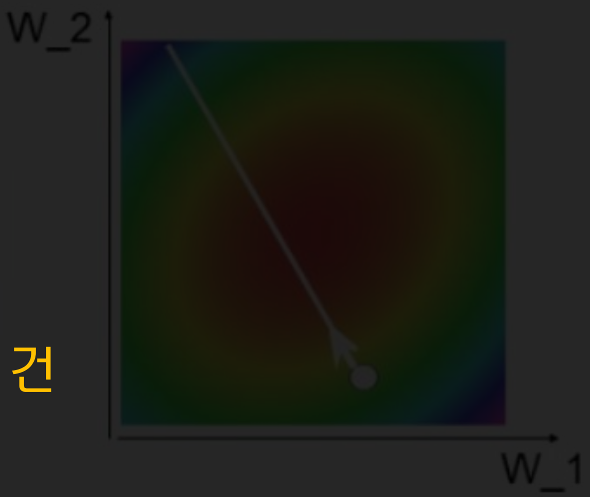


Weight의 업데이트 = Loss 줄이는 방향 (descent) x 보폭 (learning rate) x 현 지점의 기울기 (gradient)

2. Optimization

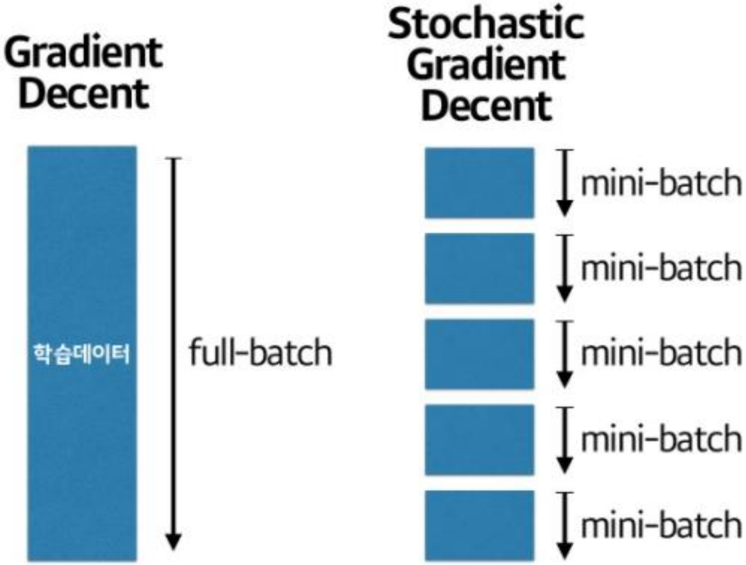
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

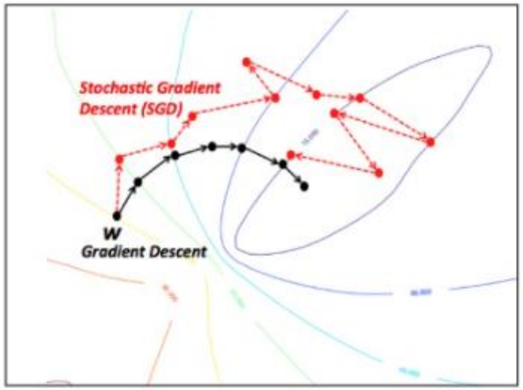
매번 전체 데이터를 사용해 계산하는 건
매우 비효율적!



Weight의 업데이트 = Loss 줄이는 방향 (descent) x 보폭 (learning rate) x 현 지점의 기울기 (gradient)

2-1. SGD





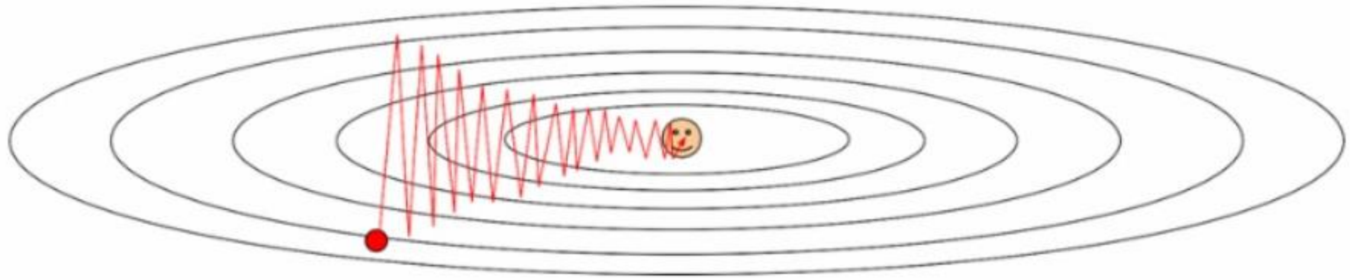
전부다 읽고나서
최적의 1스텝 간다.

작은 토막마다
일단 1스텝간다.

2-1. SGD

문제1) 학습 속도가 느리다.

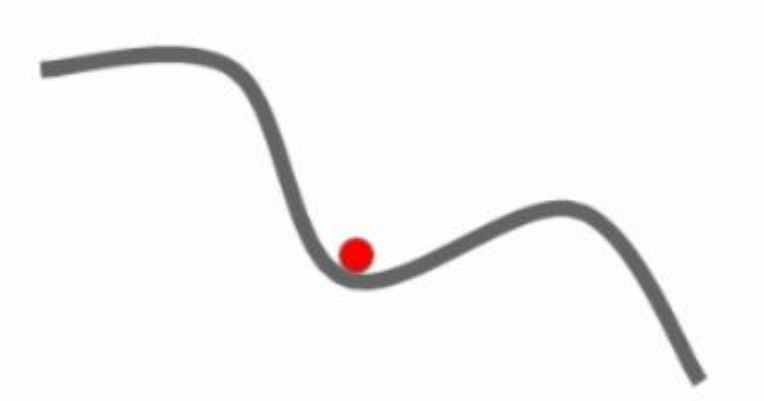
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



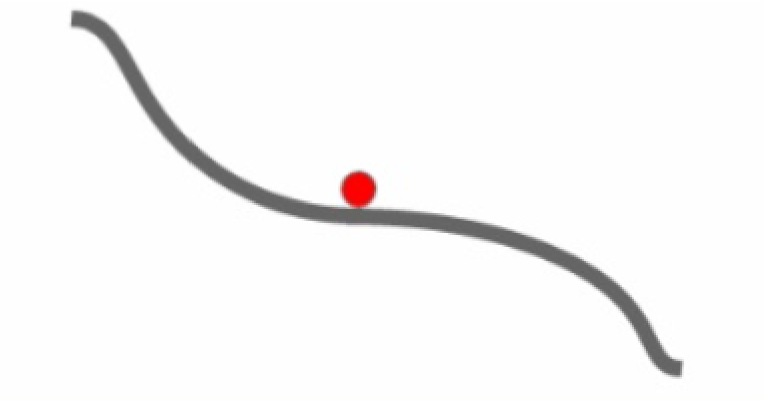
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

2-1. SGD

문제2) local minima와 saddle point에 빠질 수 있다. (gradient가 0)



local minima



saddle point
고차원일수록 더 자주 발생

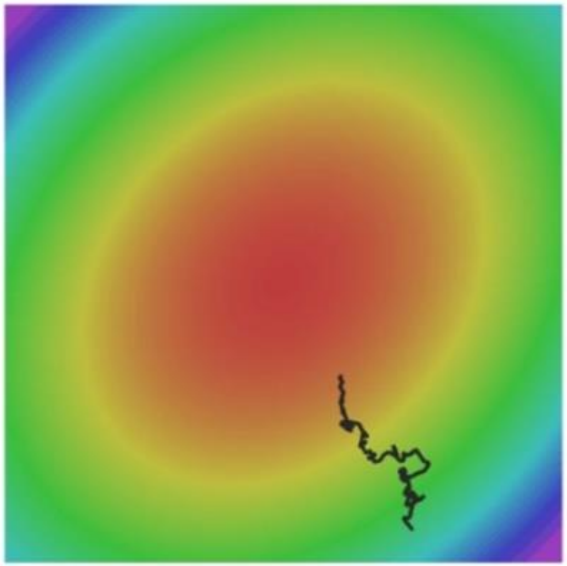
2-1. SGD

문제3) mini-batch 단위로 계산하기 때문에 noise에 취약하다.

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



2-1. SGD

Gradient가 0일 때도 멈추지 않도록 하는 방법이 필요하다!

2-2. SGD + Momentum

관성을 사용해 gradient가 0일 때도 멈추지 않도록 한다.

➔ mini-batch의 gradient 방향과 velocity를 함께 고려한다.

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

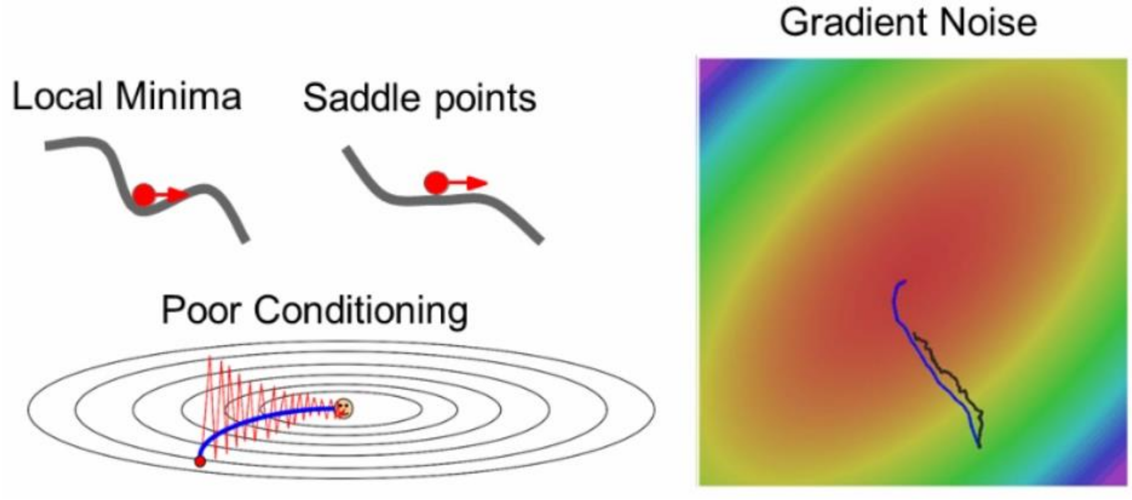
rho는 보통 0.9 또는 0.99

$$\underline{v_{t+1}} = \boxed{\rho} v_t + \boxed{\nabla f(x_t)}$$
$$x_{t+1} = x_t - \alpha \underline{v_{t+1}}$$

2-2. SGD + Momentum

관성을 사용해 gradient가 0일 때도 멈추지 않도록 한다.

➔ mini-batch의 gradient 방향과 velocity를 함께 고려한다.

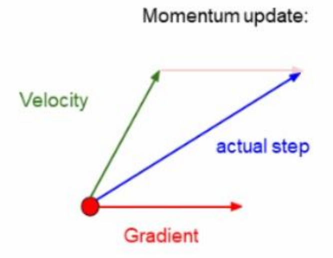


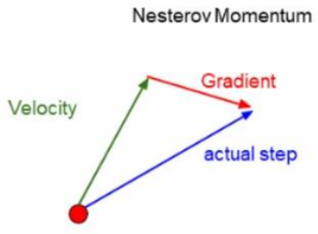
Gradient가 0이더라도 v값이 더해지면서 멈추지 않고 이동할 수 있다!

2-3. Nesterov Momentum (Nesterov Accelerated Gradient)

관성 방향으로 움직인 자리에서 gradient로 next step을 계산한다.

➔ 실제로 움직이는 것은 아니다. 관성 방향으로 이동한 자리에서 예측 후, 원래 지점에서 계산한다.





$$v_{t+1} = \rho v_t - \alpha \nabla f(\boxed{x_t + \rho v_t})$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\boxed{\tilde{x}_t})$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1 + \rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \boxed{\rho(v_{t+1} - v_t)}$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

Error-correcting term

(이전 velocity와 현재 velocity의 차 ➔ drastic shooting 방지)

2-4. AdaGrad

각각의 매개변수에 맞게 맞춤형으로 매개변수를 갱신한다.

➔ 기울기 제곱에 반비례하도록 learning rate를 조정한다.

$$h \leftarrow h + \boxed{\frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}}$$

$$W \leftarrow W - \eta \boxed{\frac{1}{\sqrt{h}}} \frac{\partial L}{\partial W}$$

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

기울기가 가파를수록(클수록) 조금씩 이동하도록

➔ 각 가중치마다 다른 learning rate를 적용해 변동을 줄이는 효과

2-4. AdaGrad

각각의 매개변수에 맞게 맞춤형으로 매개변수를 갱신한다.

➔ 기울기 제곱에 반비례하도록 learning rate를 조정한다.

$$h \leftarrow h + \boxed{\frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}}$$

➔

업데이트 1) $\frac{1}{\sqrt{K_0^2}} K_o$

업데이트 2) $\frac{1}{\sqrt{K_1^2+K_0^2}} K_1$

업데이트 3) $\frac{1}{\sqrt{K_3^2+K_1^2+K_0^2}} K_3$

업데이트 4) $\frac{1}{\sqrt{K_4^2+K_3^2+K_1^2+K_0^2}} K_4$

학습이 진행될수록 축적되는 h

➔

$$W \leftarrow W - \eta \boxed{\frac{1}{\sqrt{h}}} \frac{\partial L}{\partial W}$$

Learning rate 감소

➔ 기울기가 0인 부근에서는?
학습이 급격하게 느려진다!

2-5. RMSProp

AdaGrad처럼 보폭을 갈수록 줄이되, 이전 기울기의 맥락을 고려한다.

➔ decay rate를 사용해 이전 스텝의 기울기를 더 크게 반영하여 h값이 단순 누적되는 것을 방지한다.

---

decay rate
주로 0.9 또는 0.99 사용

$$h_i \leftarrow \boxed{\rho} h_{i-1} + \boxed{(1-\rho)} \frac{\partial L_i}{\partial W} \odot \frac{\partial L_i}{\partial W}$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$$

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

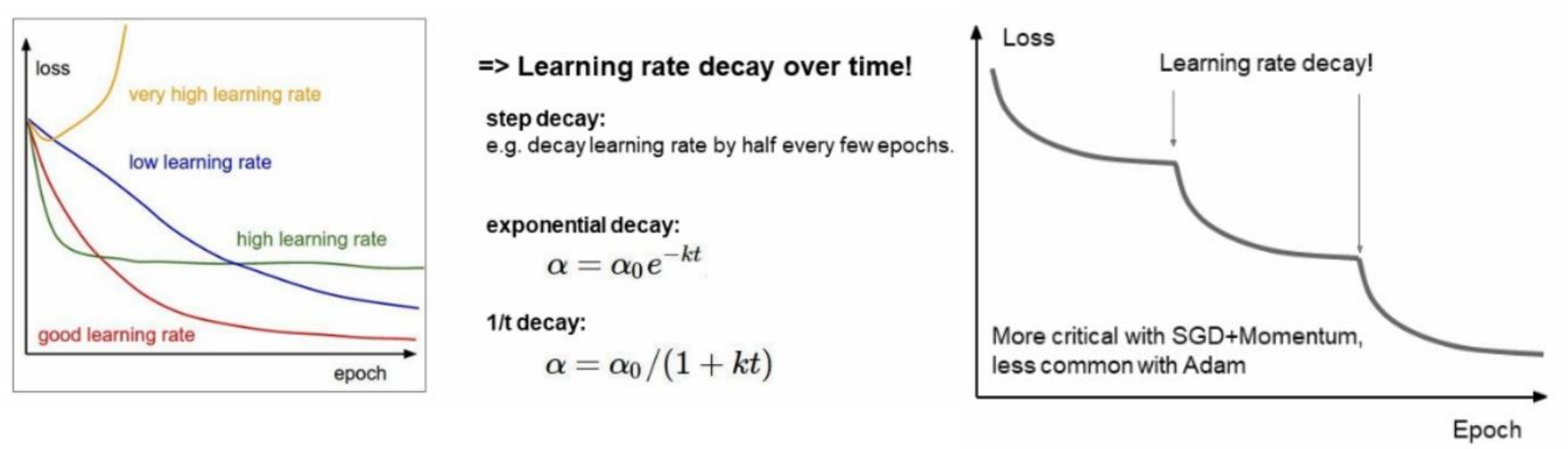
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

h 가 무한히 커지지 않으면서 ρ 가 작을 수록 가장 최신의 기울기를 더 크게 반영해
무조건적인 slow down을 방지한다!

2-5. RMSProp

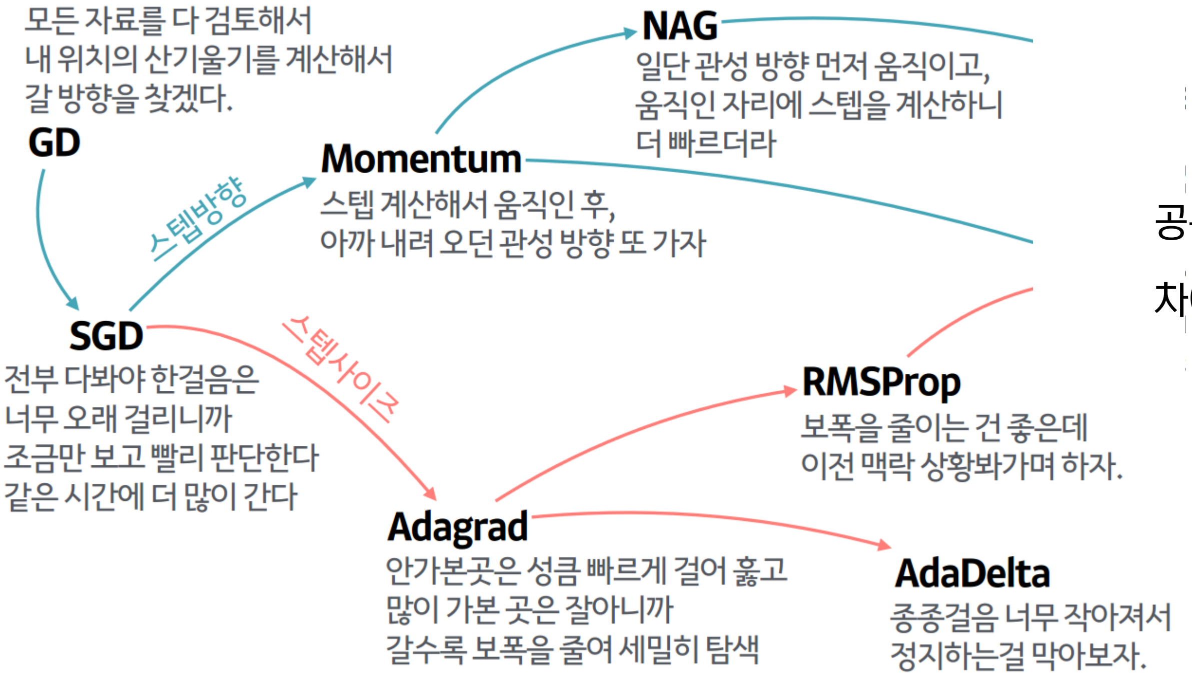
Learning rate decay?

➔ learning rate 값을 크게 준 후 일정 epoch마다 값을 감소시켜 최적의 학습까지 빨리 도달할 수 있게 하는 방법



더 다양한 decay rate 방법이 궁금하다면? https://www.youtube.com/watch?v=WUaz0tIti0g

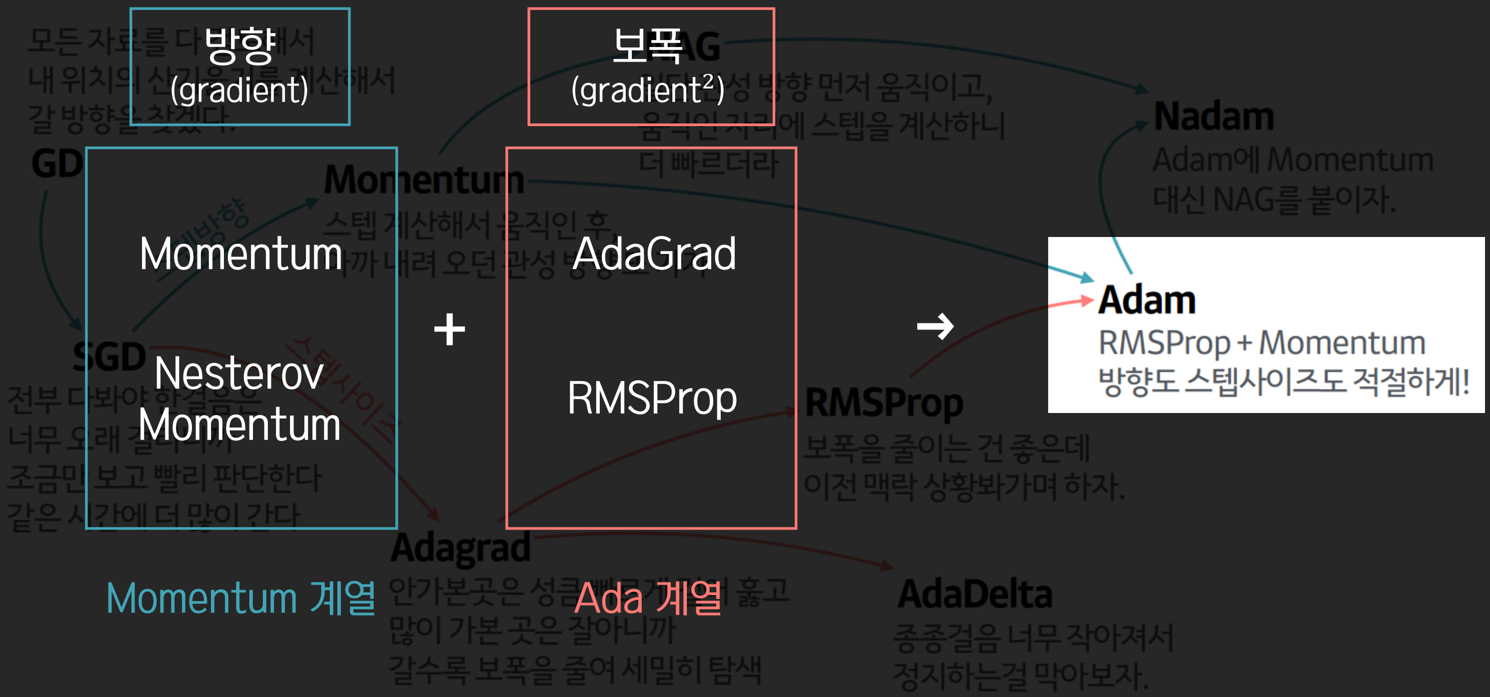
Weight의 업데이트 = Loss 줄이는 방향 (descent) x 보폭 (learning rate) x 방향 (gradient)



공통점 이전 step의 gradient 활용

차이점 gradient / gradient^2

Weight의 업데이트 = Loss 줄이는 방향 (descent) X 보폭 (learning rate) X 방향 (gradient)



2-6. Adam

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Beta2는 decay rate이기 때문에 0.9 혹은 0.99
1회 업데이트 후 여전히 0에 가까운 second_moment로 나누면…

2-6. Adam

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

현재 step에 맞는 적절한 bias를 넣어 값이 튀지 않게 한다!

2-6. Adam

```
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

↓

Gradient의 1차 moment에 대한 추정치 $$\boxed{m_t} = \beta_1 m_{t-1} + (1-\beta_1)\boxed{g_t}$$

Gradient의 2차 moment에 대한 추정치 $$\boxed{v_t} = \beta_2 v_{t-1} + (1-\beta_2)\boxed{g_t^2}$$

2-6. Adam

확률변수 X의 n차 moment(적률) : $E[X^n]$

1차 moment(적률) : $E[\mathrm{X}]$ 모평균

2차 moment(적률) : $E[X^2]$

$Var[\mathrm{X}] = E[X^2] - E[\mathrm{X}]^2$

표본평균과 표본제곱평균을 통해 모수인 $E[\mathrm{X}]$와 $E[X^2]$를 추정

Gradient의 1차 moment

$E[Gradient]$

Gradient의 2차 moment

$E[Gradient^2]$

2-6. Adam

$\beta_1 = 0.9$

Gradient의 1차 moment에 대한 추정치 $\boxed{m_t} = \beta_1 m_{t-1} + (1-\beta_1)\boxed{g_t}$

Gradient의 2차 moment에 대한 추정치 $\boxed{v_t} = \beta_2 v_{t-1} + (1-\beta_2)\boxed{g_t^2}$

$\beta_2 = 0.99$

---

bias-corrected

$$\boxed{\hat{m}_t} = \frac{m_t}{1-\beta_1^t} \qquad (\mathrm{E}[\widehat{m_t}] = \mathrm{E}[\mathrm{Gradient}])$$

$$\boxed{\hat{v}_t} = \frac{v_t}{1-\beta_2^t} \qquad (\mathrm{E}[\widehat{v_t}] = \mathrm{E}[Gradient^2])$$

최종

best learning rate = 1e−3 or 5e−4

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

2-6. Adam

Adam (full form)

```python
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

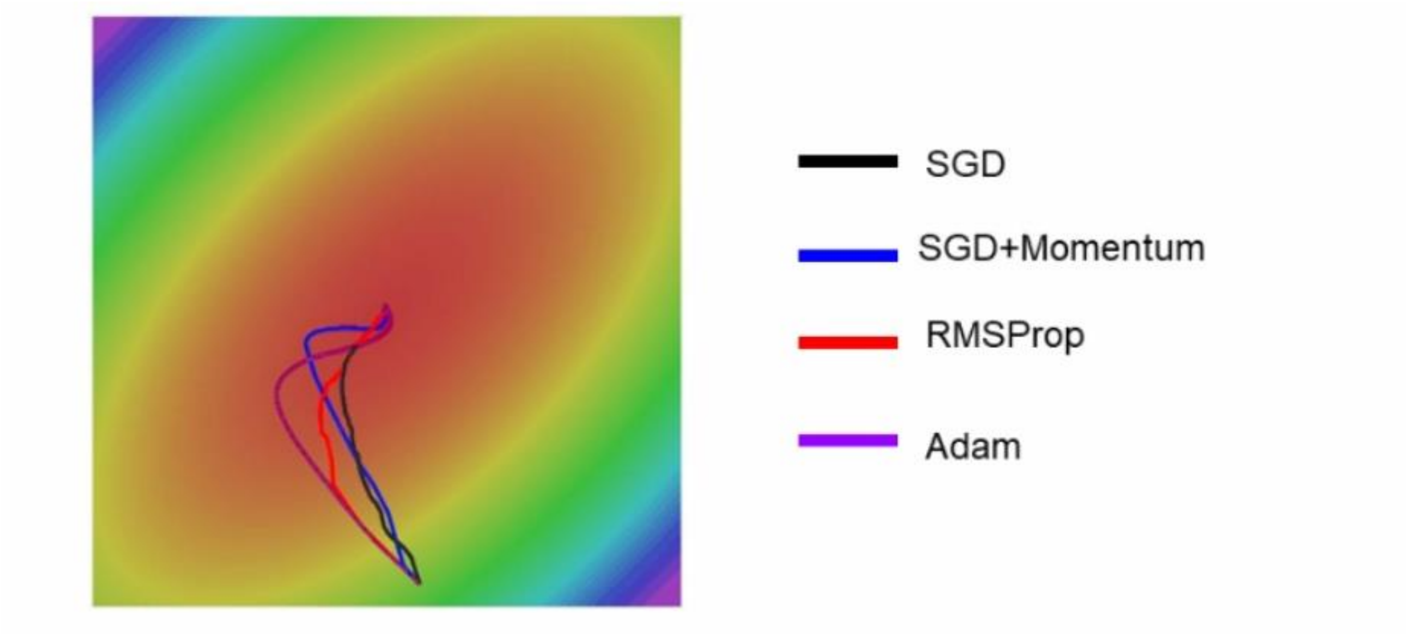
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

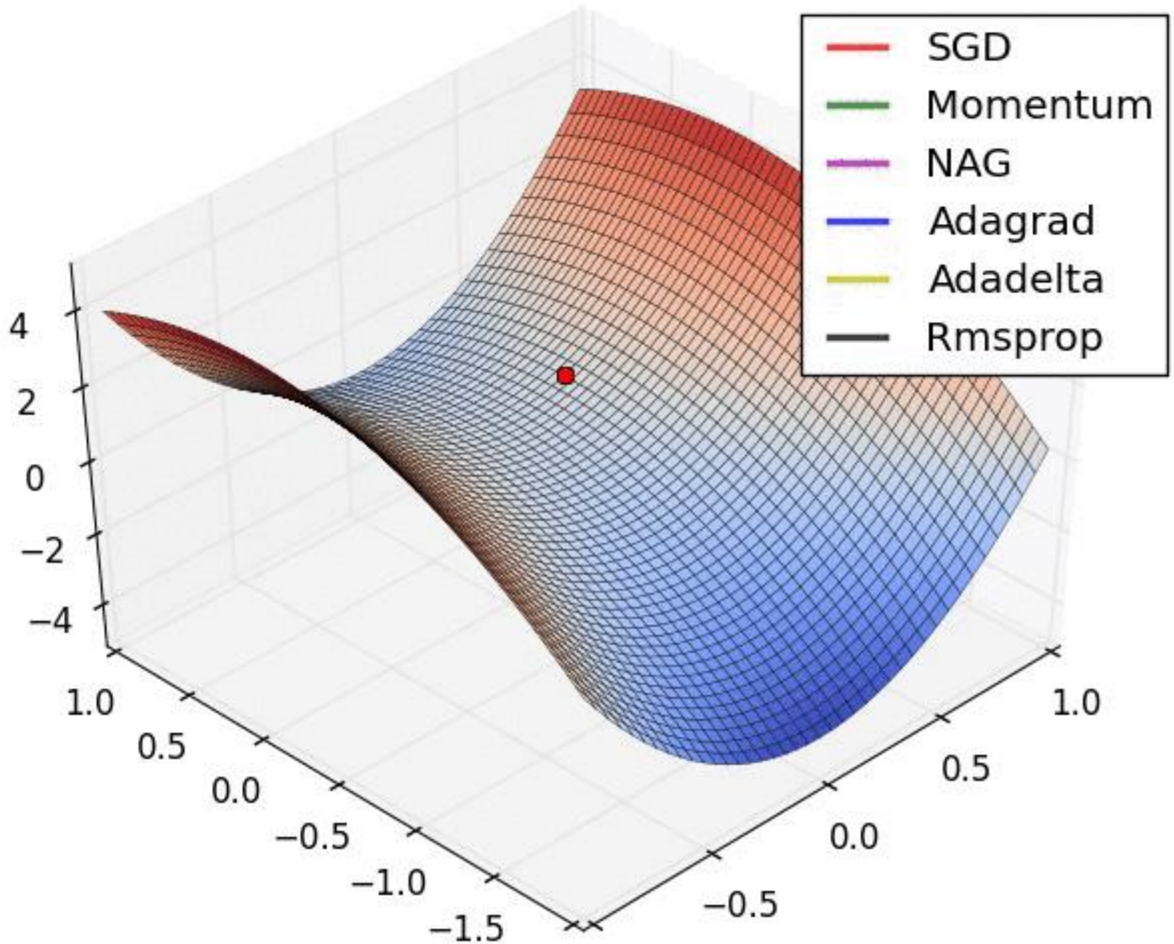
Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

특히 decay rate 가 작으면, 즉 β_1 과 β_2 가 1에 가까우면 편향이 더 심해진다.
편향을 잡아주기 위해 bias-correction을 계산한다.

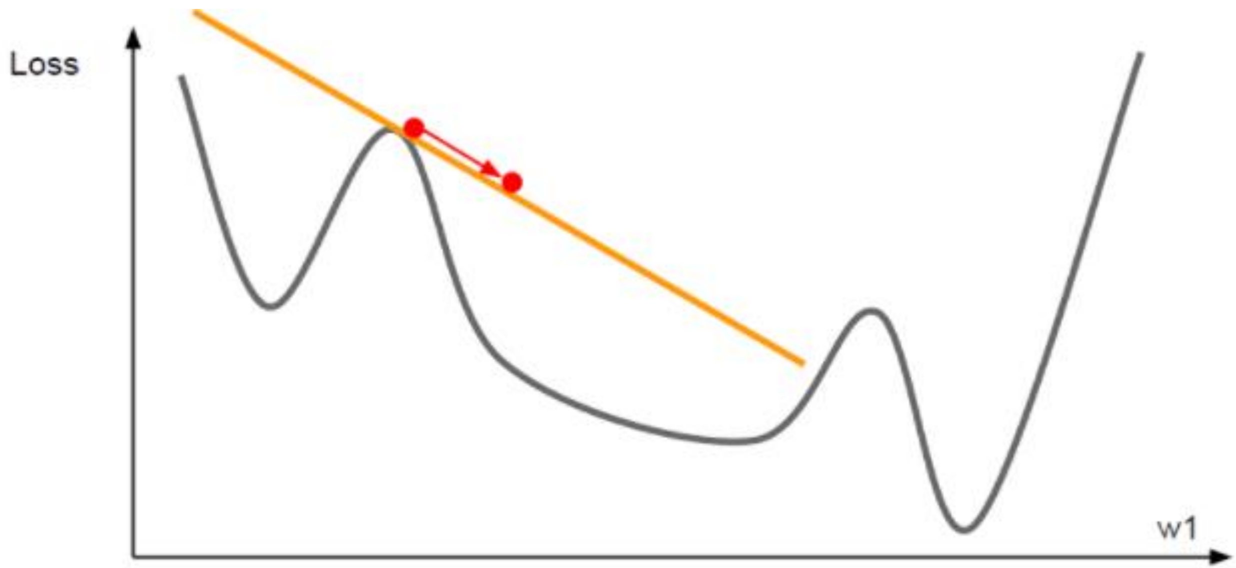
2-6. Adam



Optimizer overview 논문 : An overview of gradient descent optimization algorithms

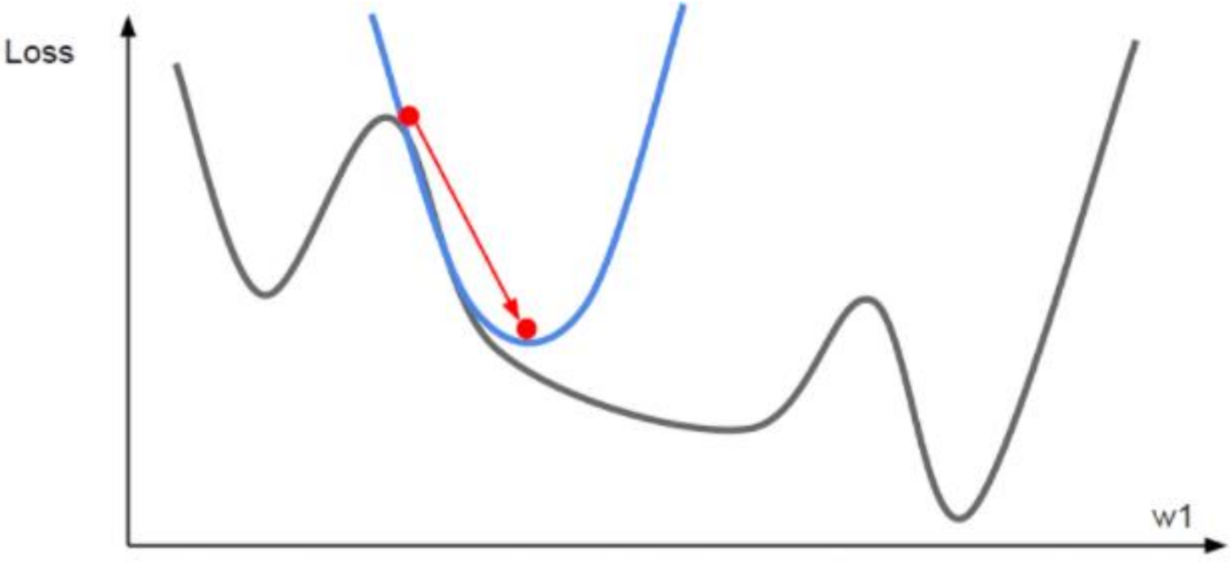

SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

First-Order Optimization



1) Use gradient form linear approximation
2) Step to minimize the approximation

Second-Order Optimization



1) Use gradient and **Hessian** to form **quadratic** approximation
2) Step to the **minima** of the approximation

Second-Order Optimization

Second-order Taylor expansion

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

장점

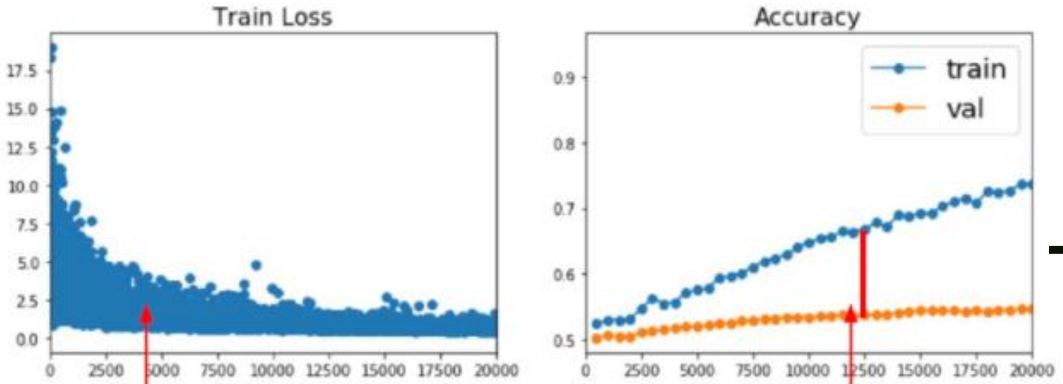
- Hyperparameters X
- Learning rate X

단점

- Hessian은 O(N^2)개의 elements

→ 연산 량이 너무 많다

Beyond Training Error



→ Overfitting

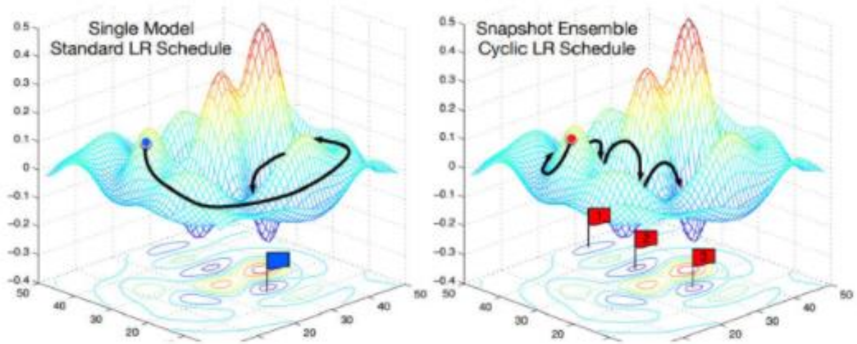
Better optimization algorithms help reduce training loss

But we really care about error on new data - how to reduce the gap?

Model Ensembles

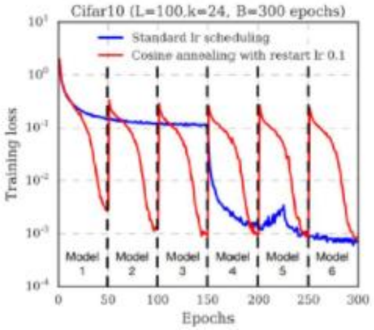
1. 여러 개의 모델을 학습시킨다
2. 테스트 할 때 결과를 평균 낸다

→ 1-2%의 최종 성능 향상이 가능하다



학습 도중에 모델들을 저장해서 평균을 낸다

→ 시간, 비용 부담 줄일 수 있음

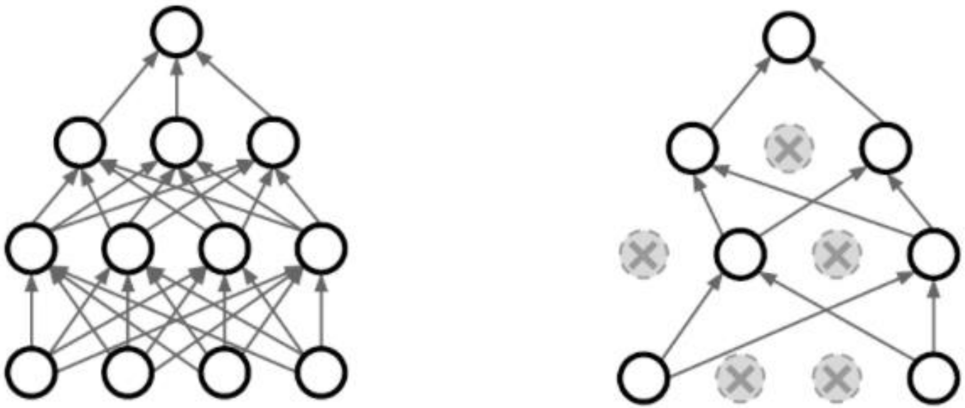


→ 여러 개의 learning rate를 사용하여 여러 지점에 수렴하도록 한다

Regularization: Dropout

Dropout: Forward pass 과정에서 일부 뉴런들의 activation 값을 랜덤하게 0으로 만든다.

Drop하는 확률은 hyperparameter인데, 보통 0.5로 설정한다.

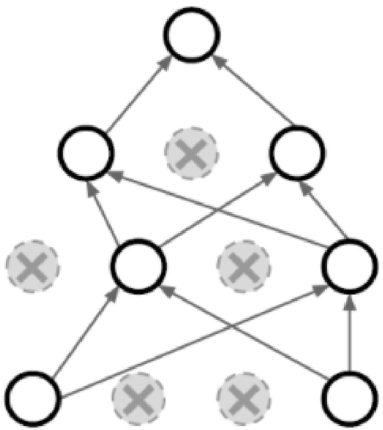


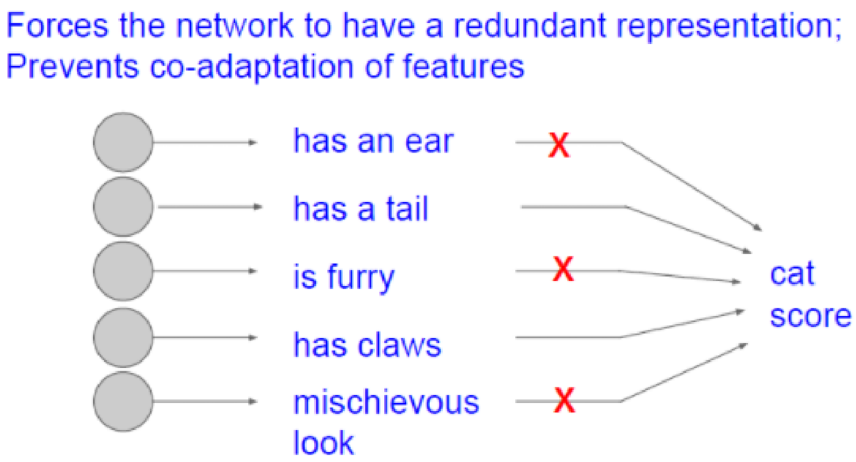
대부분 Fully Connected Layer에서 하지만, CNN Layer에서 할 때도 있음

Regularization: Dropout

Dropout: 하나의 feature를 담당하는 전문가의 수를 줄여 overfitting 방지

(전문가의 능력을 줄이는 것은 X)





하나의 노드

= 한 명의 전문가

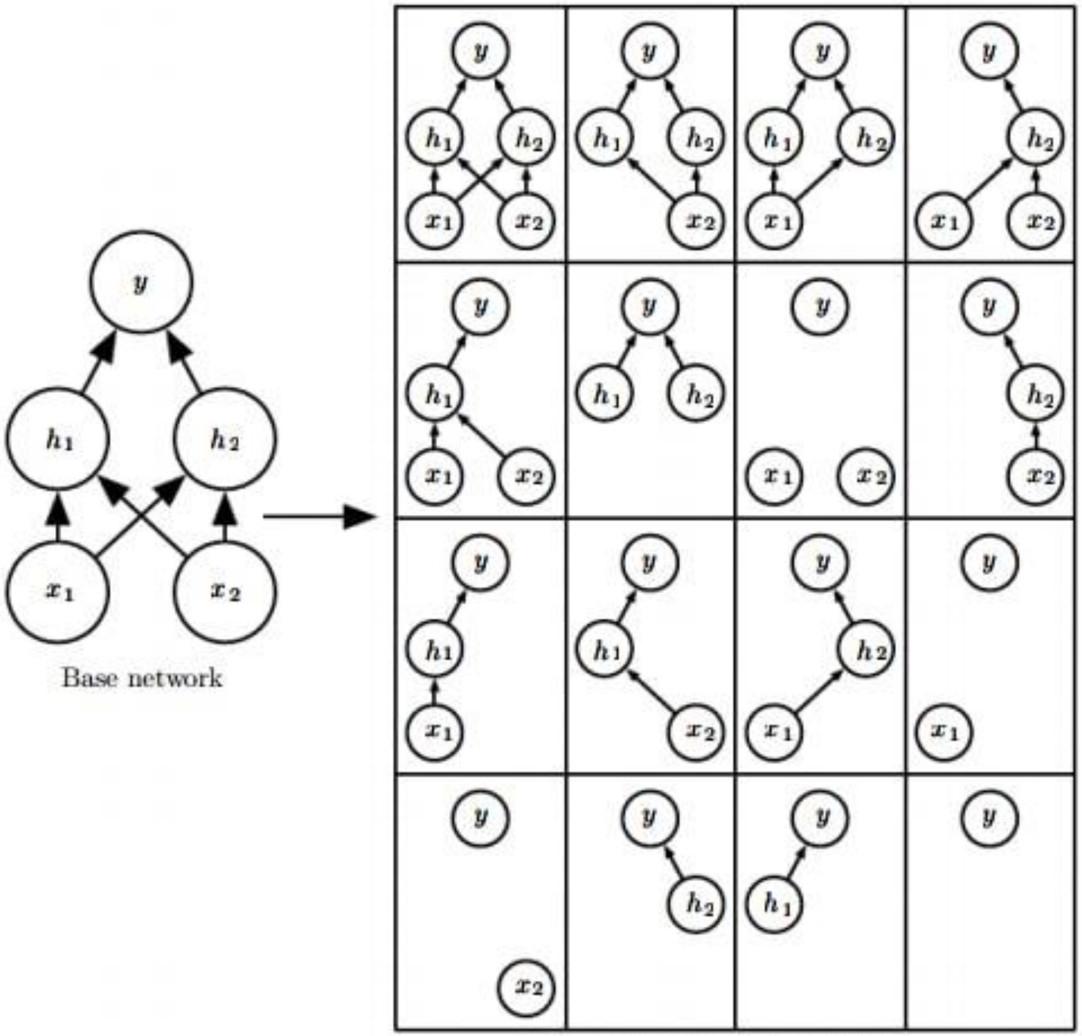
Dropout ⇔ Ensemble

: 전문가 노드 1개를 하나의 모델로 생각했을 때, dropout은 model 내에서의 Ensemble

→ 네트워크에서 하위 subnetwork들이 갈라지며 뉴런들이 증가하는 것을 dropout 으로 막아서 overfitting 방지

→ 같은 변수들을 공유하는 하나의 모델이지만 앙상블의 효과를 낸다

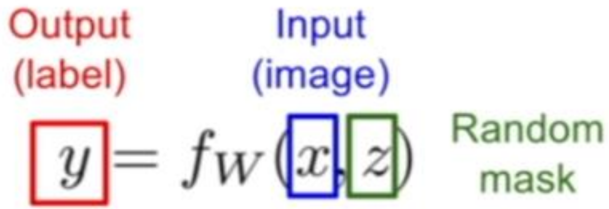
: 랜덤한 dropout → 랜덤한 subnetwork



Base network

Ensemble of Sub-Networks

Regularization: Dropout



Random mask가 추가되어
Random한 Output 나온다!

매번 random한 모델로 결과를 도출하면 안됨!

→ Randomness를 average-out 시킨다

Batch Normalization

Training: Mini batch를 사용하여 하나의 데이터가 샘플링 될 때 매번 서로

다른 데이터들과 만난다 → Randomness(Noise) 부여

Testing: 정규화를 train할 때 처럼 mini batch 단위가 아니라 global 단위로

수행하여 randomness를 average out

Data Augmentation

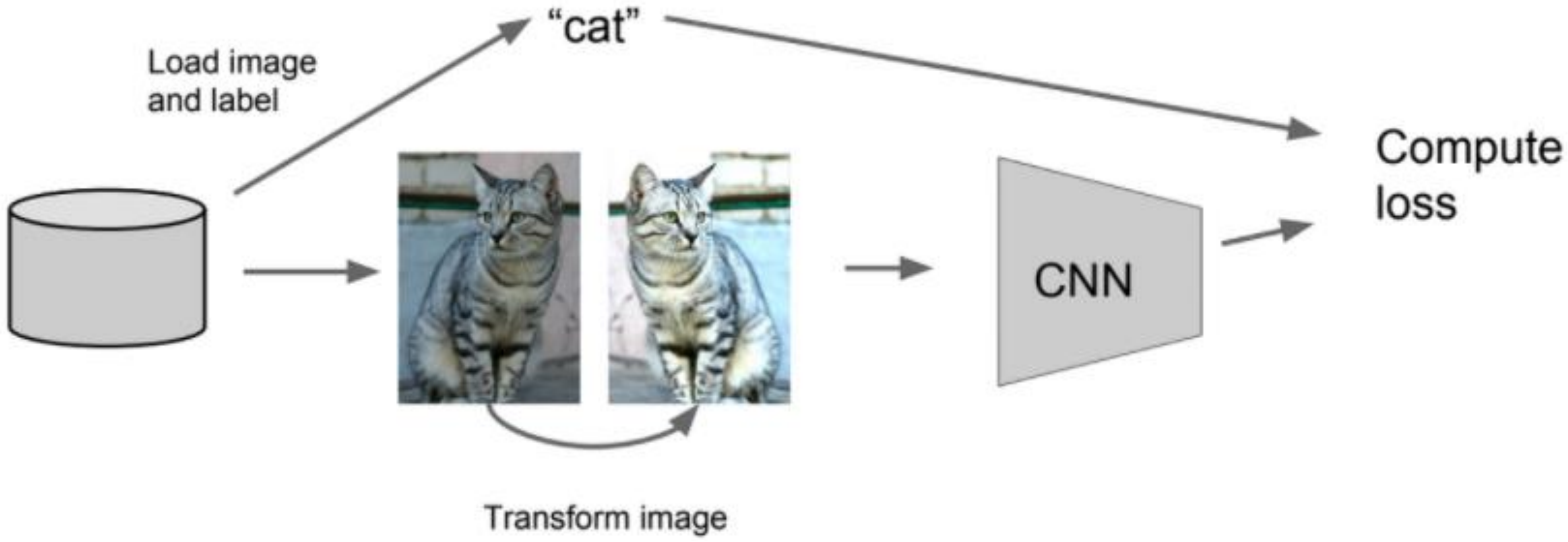
➔ Training set과 현실의 test set 사이의 괴리감 존재

➔ 임의의 잡음이나 translation을 training set에 가해서 괴리감을 줄이고 성능 향상

➔ 데이터 크기가 작은 경우 데이터셋을 늘리기 위해 사용

Regularization: Data Augmentation

Label은 보존! 이미지를 변형

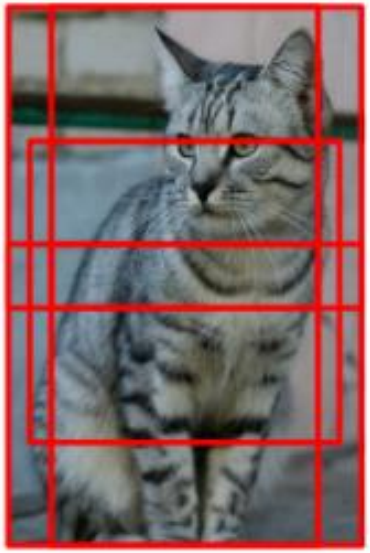


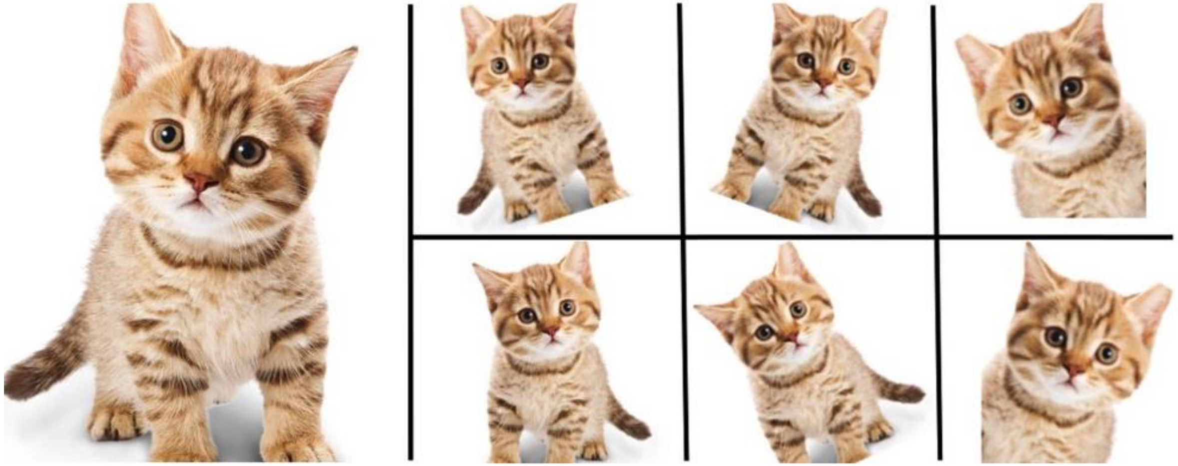
Regularization: Data Augmentation

Random crops and scales

Training: 이미지를 다른 사이즈의 sample crop들을 랜덤하게 수집한다

Testing: 모든 crop들을 average out한다

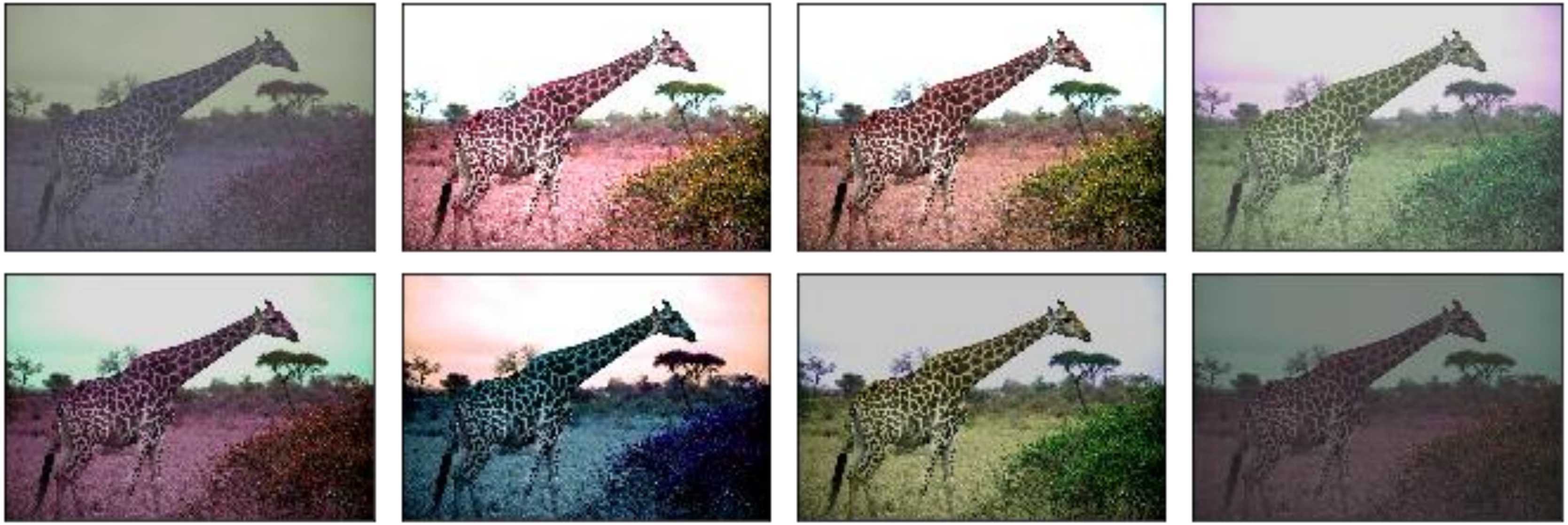




Regularization: Data Augmentation

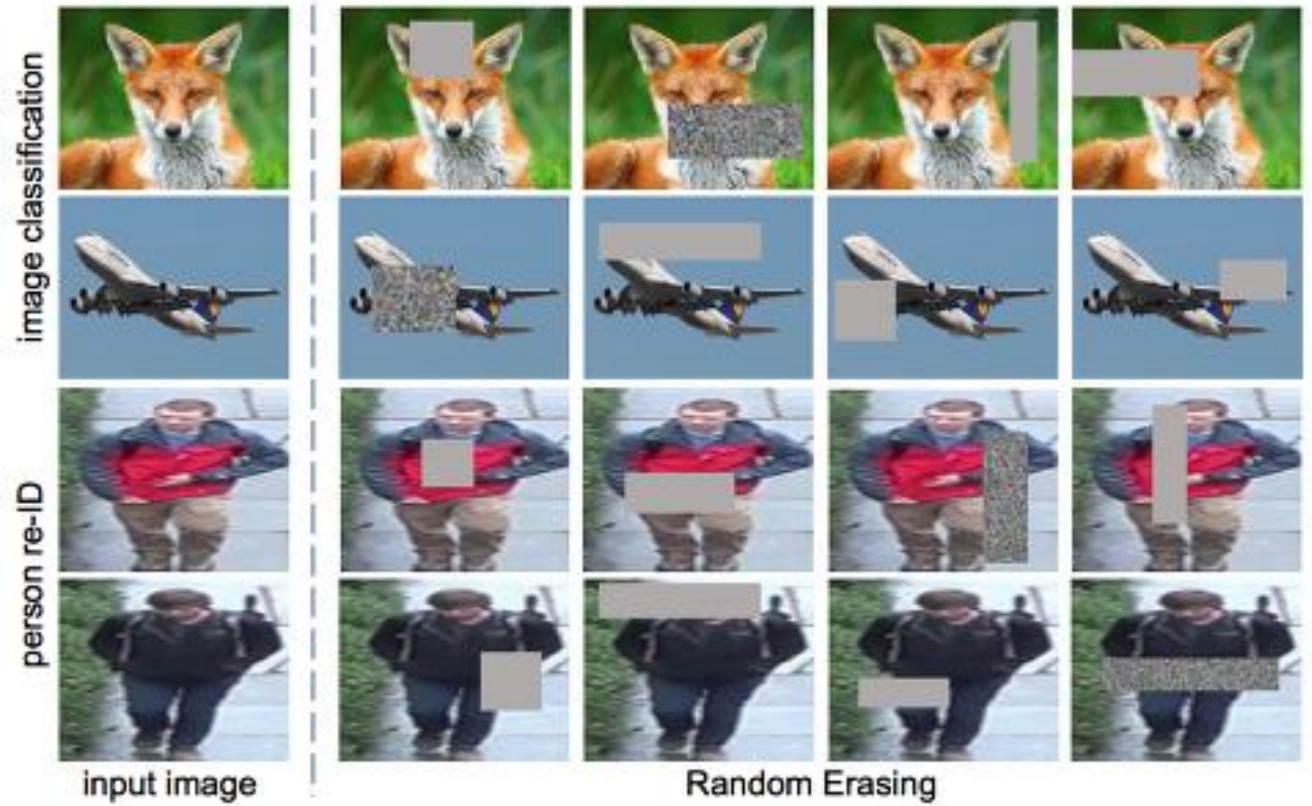
Color Jitter

: 학습 과정에서 랜덤으로 Lightness, Hue, Saturation 등 변화 시킴

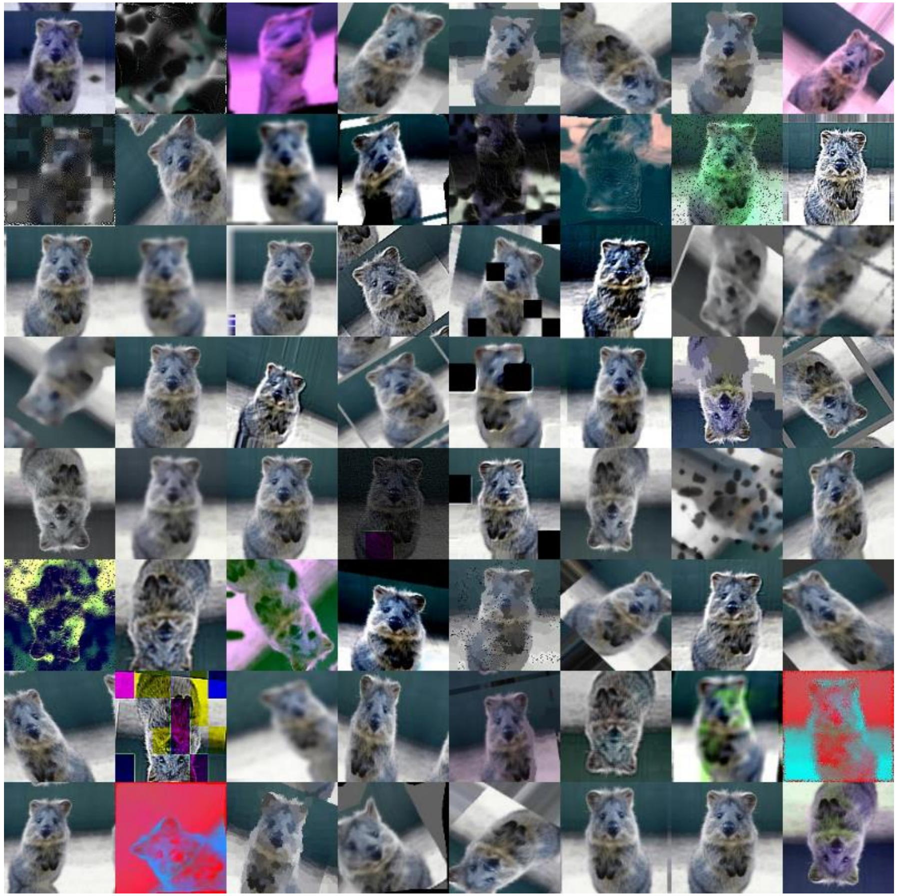


Regularization: Data Augmentation

Random Erasing



Regularization: Data Augmentation

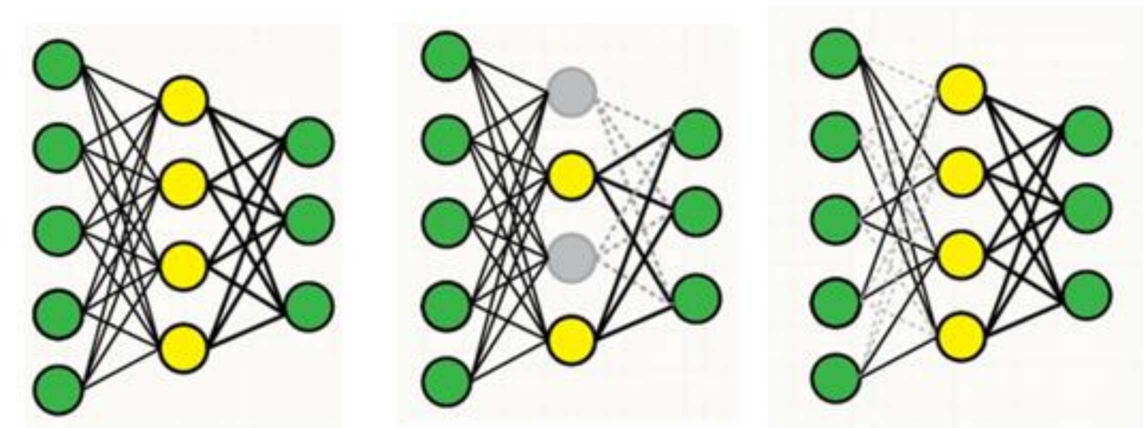


매우 다양한 방식의
Augmentation 존재!

Regularization: DropConnect

DropConnect: Weight matrix들만 임의로 0으로 제거

⇔ Dropout: 퍼셉트론을 끊어 연결된 가중치 제거



Original Dropout DropConnect

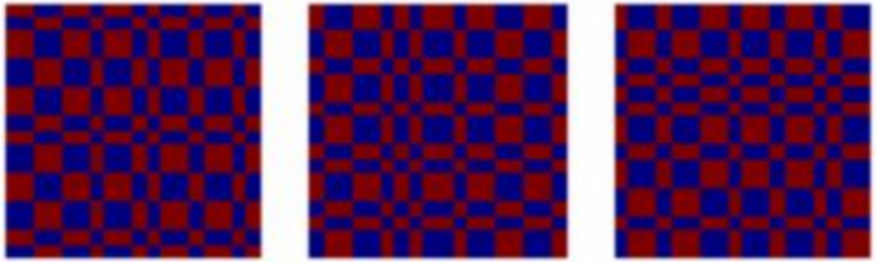
Fractional Max Pooling

Training: 보통의 max pooling 은 pooling 하는 영역이 정해져 있음

→ Pooling layer마다 pooling 하는 영역을 랜덤하게!

Testing: 고정된 pooling 영역을 사용

→ Average out stochasticity



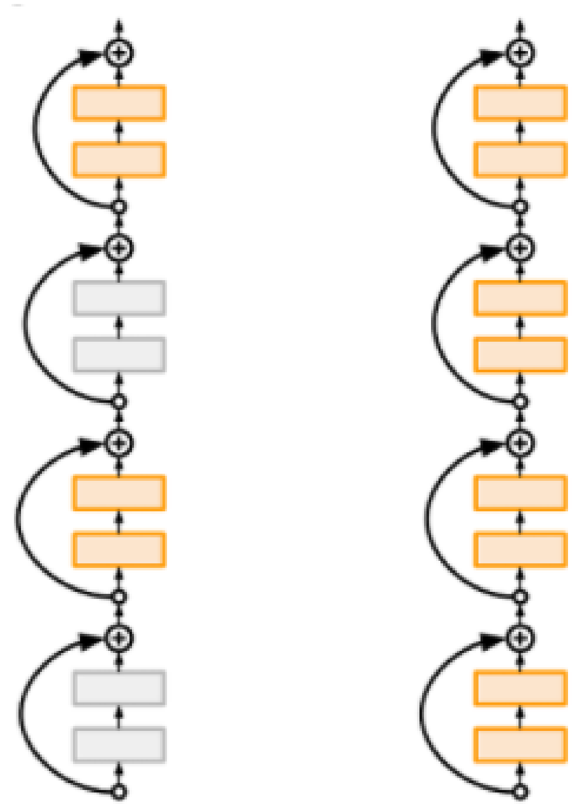
Regularization: Stochastic Depth

Stochastic Depth

Training: 네트워크의 depth를 랜덤하게 Drop한다

Testing: 전체 네트워크 사용

→ Dropout과 유사한 효과



Regularization

정규화의 패턴

Training: Add random noise

Testing: marginalize over the noise

< 예시 >

Dropout

Batch Normalization(BN)

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

CNN을 학습 / 사용하기 위해서는

매우 큰 데이터가 필요하다

CNN을 학습 / 사용하기 위해서는

매우 큰 데이터가 필요하다

Transfer Learning

: 사전학습 된 모델을 이용한다(pre-trained model)

→ 내가 풀고자 하는 문제와 비슷하고, 사이즈가 큰 데이터로 이미

학습된 모델

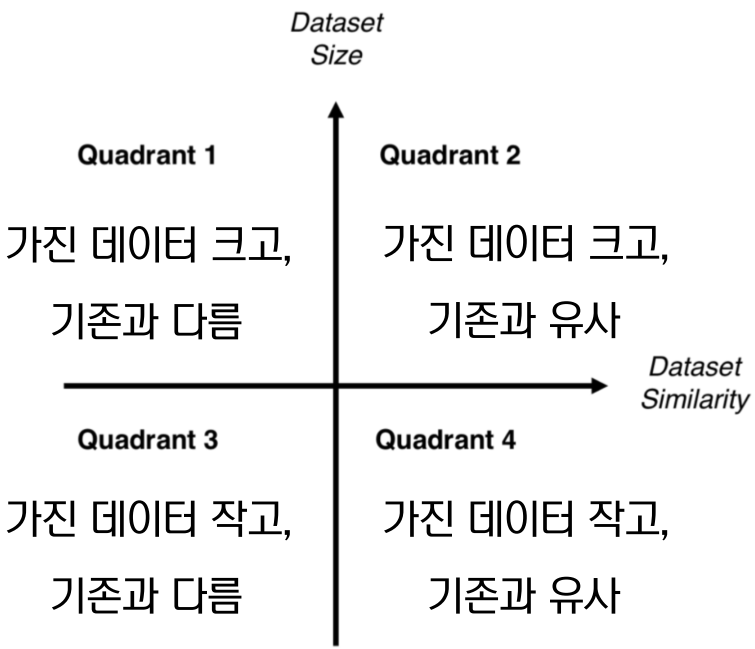
Transfer Learning

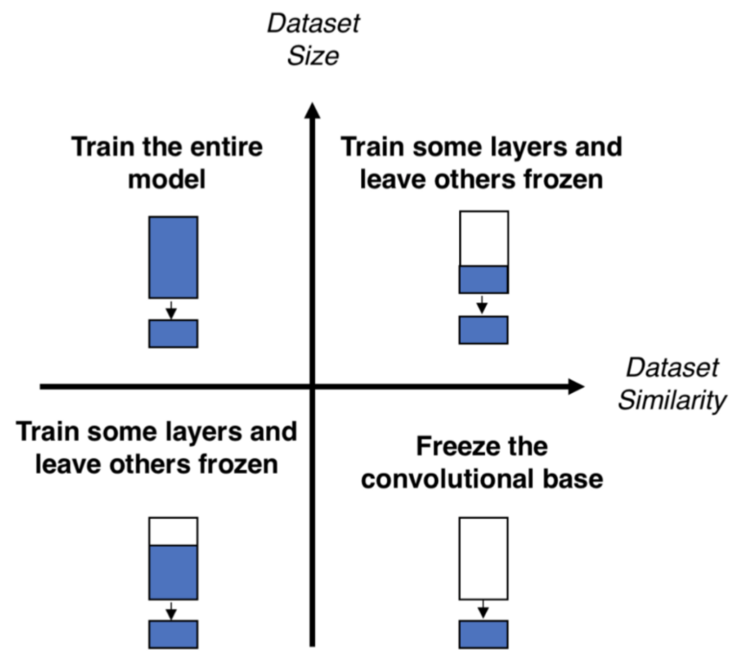
Transfer Learning

CNN의 Transfer Learning(Image Classification Model)

Convolutional base: 이미지로부터 특징 추출

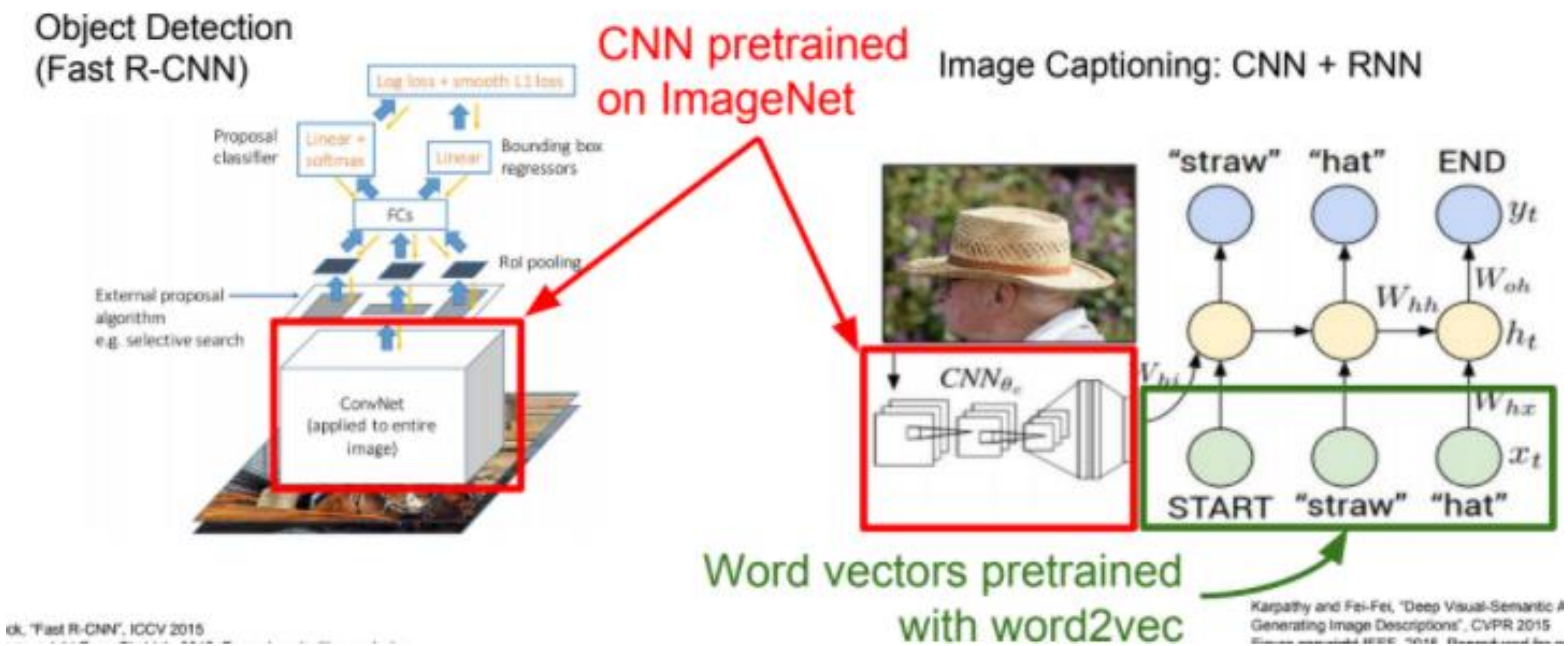
Classifier: 추출된 특징을 입력 받아 최종적으로 이미지의 카테고리 결정





Transfer Learning

높은 정확도, 빠른 시간

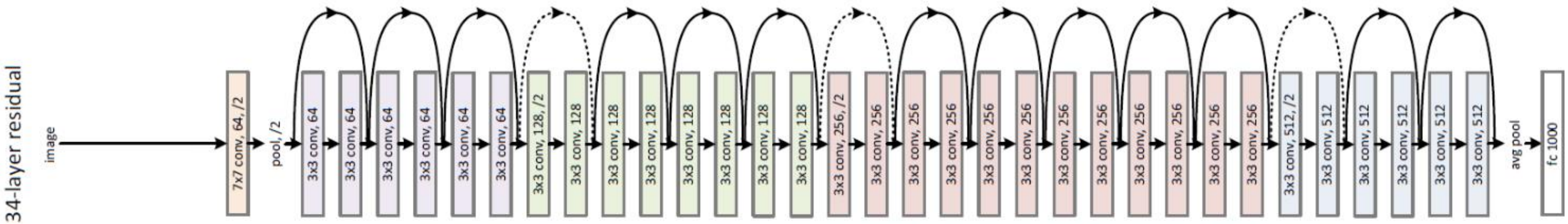


Transfer Learning

ResNet: residual blocks로 구성됨

Pytorch에서 ResNet-18, ResNet-34, ResNet-50,

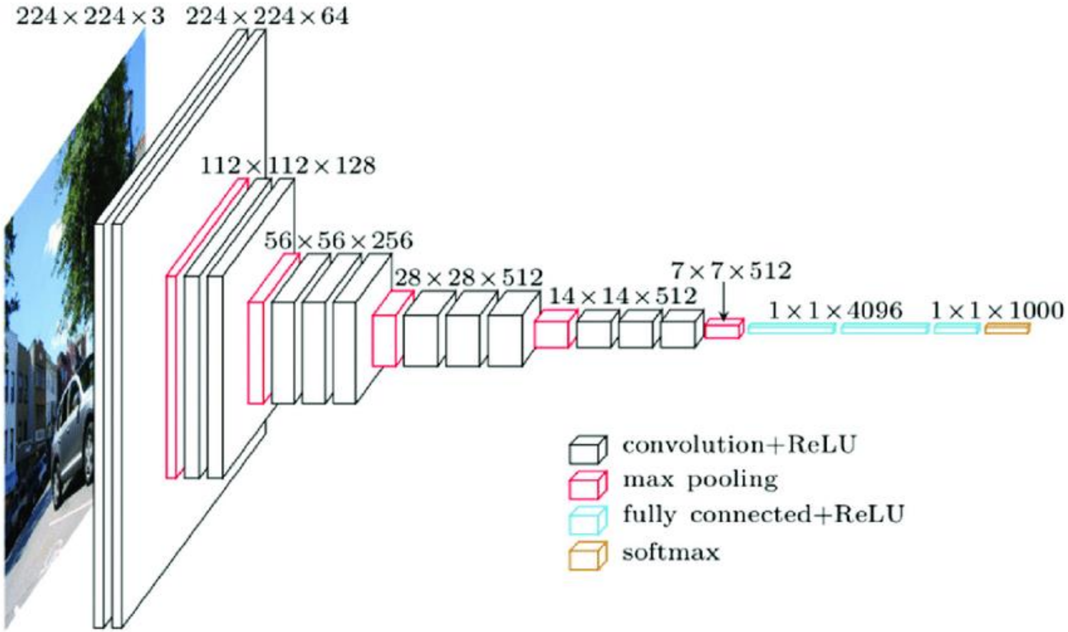
ResNet-101, and ResNet-152 등 제공



Transfer Learning

VGG(Visual Geometry Group at University of Oxford)

- VGG-N 모델들은 각각 N개의 층들이 있다.
- Pytorch에서 VGG-11, VGG-13, VGG-16, VGG-19 등이 제공됨



Transfer Learning

GoogLeNet

AlexNet

SqueezeNet

DenseNet

ShuffleNetv2

.

.

.

다양한 모델 활용 가능

감사합니다

Q&A