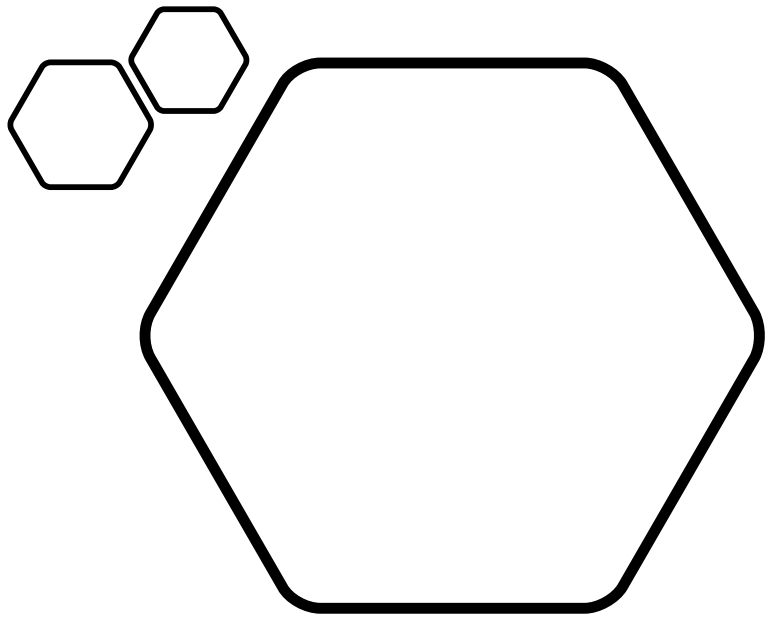
Lecture 7

Cs231n

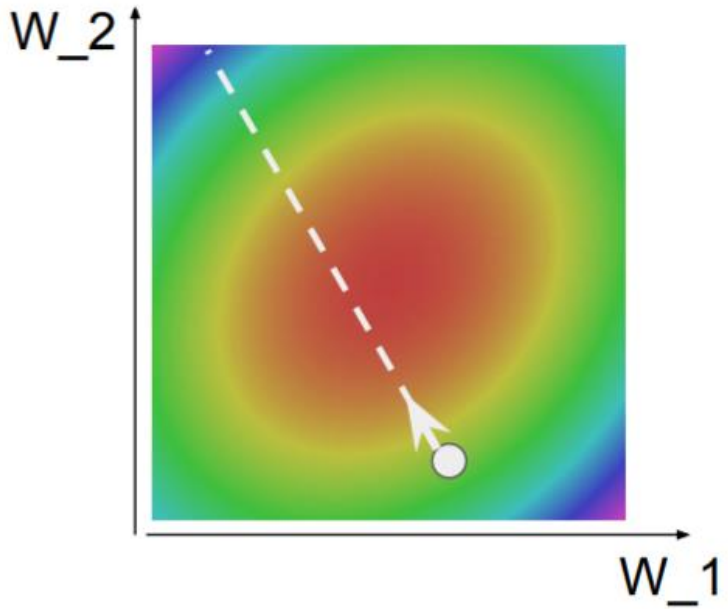


SGD

Optimization

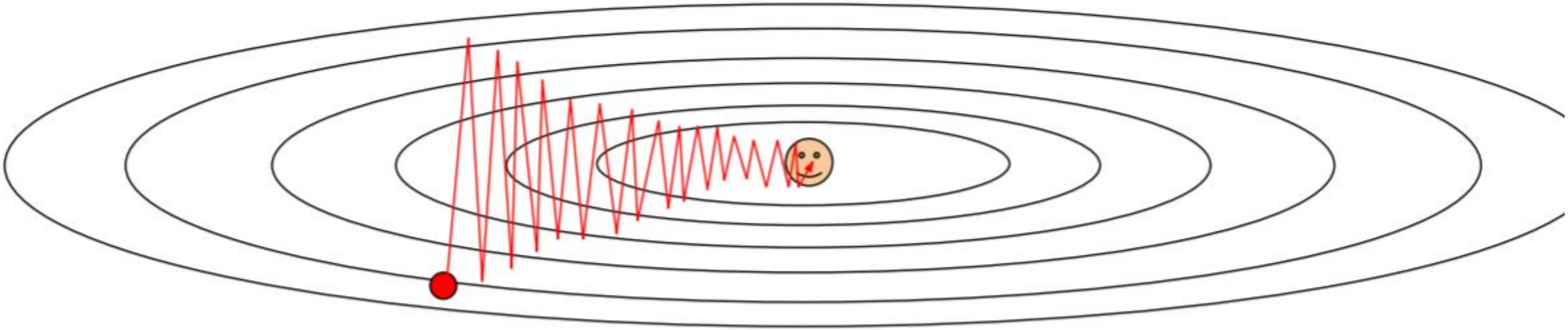
```
# Vanilla Gradient Descent

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



Problem with SGD – loss function affects a lot

- IF) 한 방향의 가중치에 대해서만 잘 영향받는 descent?

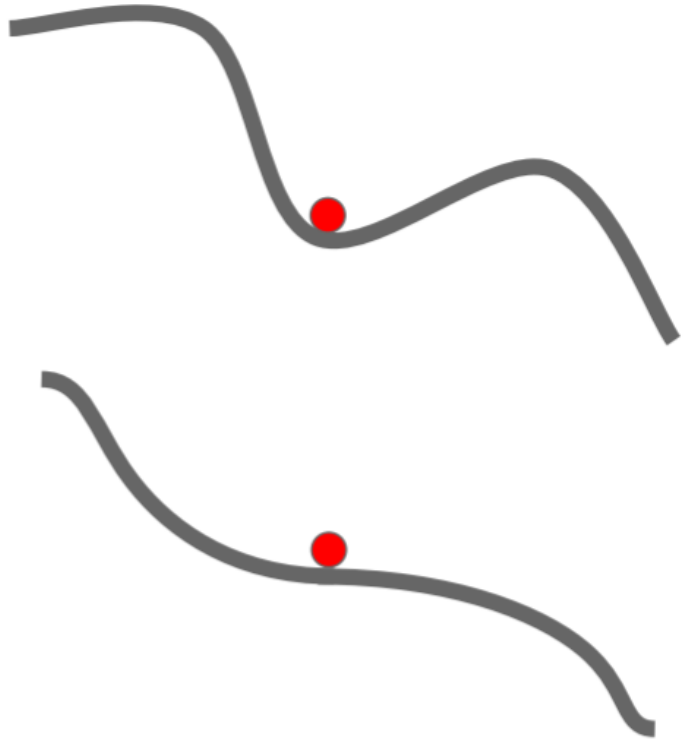


- 지그재그로 움직이는 문제 발생

Problem with SGD

- IF) local minima 나 saddle point(안장점) 에 안착할 경우 더 좋은 값을 안찾으러감.

- 더 큰 문제는 high dimension 일수록 더 많은 local minima,saddle point 가 있음.

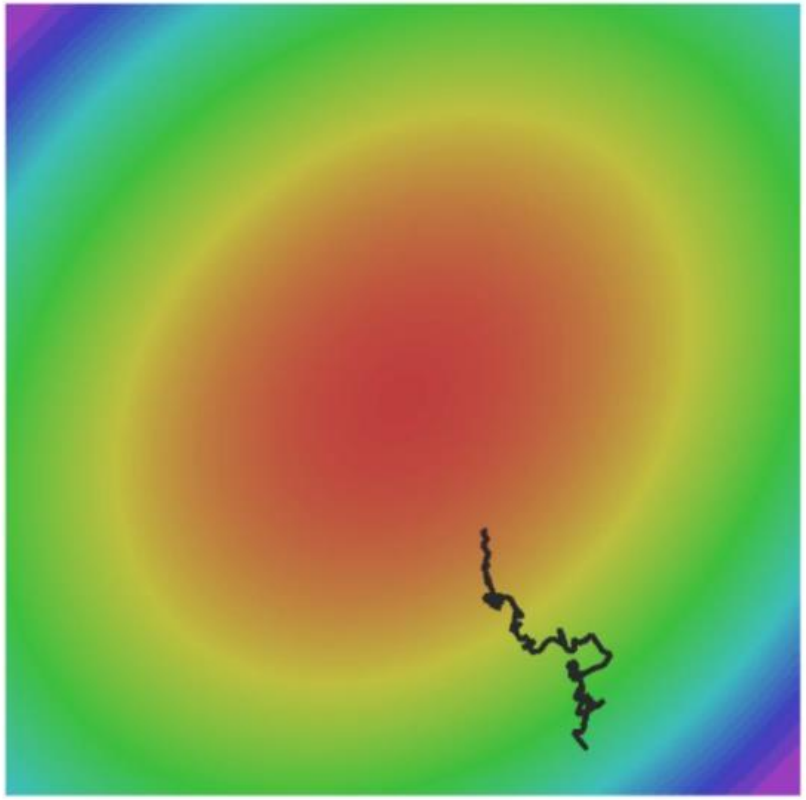


Problem with SGD

- SGD 는 배치단위 → 노이즈가 발생할 확률이 올라감. → 시간이 많이걸리게됨.

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

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



이 모든 문제는 full batch에서도 발생함.

몇가지 해결방법

- Momentum
  - -Nesterov Momentum
- AdaGrad
- RMSProp
- Adam

Momentum

- 운동량의 개념을 도입. “속도” , “마찰력”의 개념을 생각
- 모든 문제가 조금은 완화됨.
- local minima,saddle point에서도 속도를 가지고있다고 생각하면 멈추지 않고 계속감
- 가속도의 개념으로 poor conditioning 을 해결가능(지그재그로 가지않는 지점은 가속)
- 모멘텀이 노이즈를 평균화하여 완화

SGD

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

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

SGD+Momentum

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

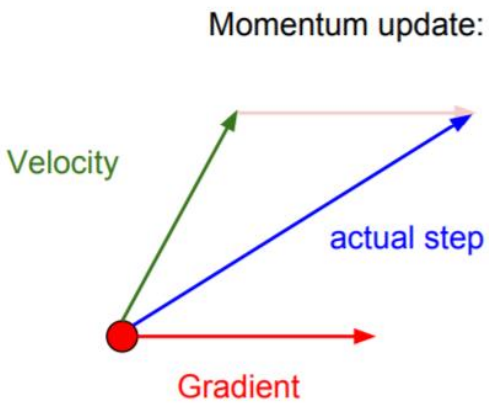
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

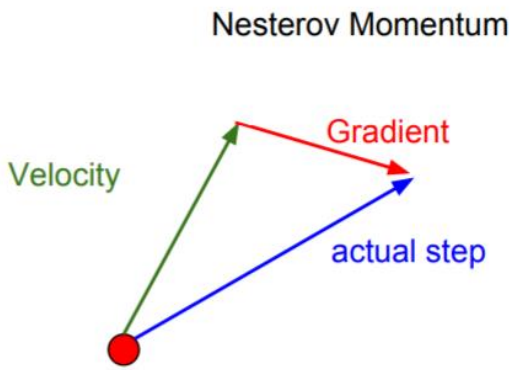
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Nesterov Momentum

- Actural update는 gradient와 속도의 가중평균으로 구해짐
- 기본 아이디어는 같으나 순서를 조금 바꾼아이디어.
- Gradient 에 velocity 를 가중하여 계산하는것이 기존방법
- Nesterov는 현재지점에서 velocity를 먼저 계산한 후에 그 지점에서 gradient를 계산 후 원점으로 돌아가 합함. Velocity값을 그래디언트 값으로 보정함.

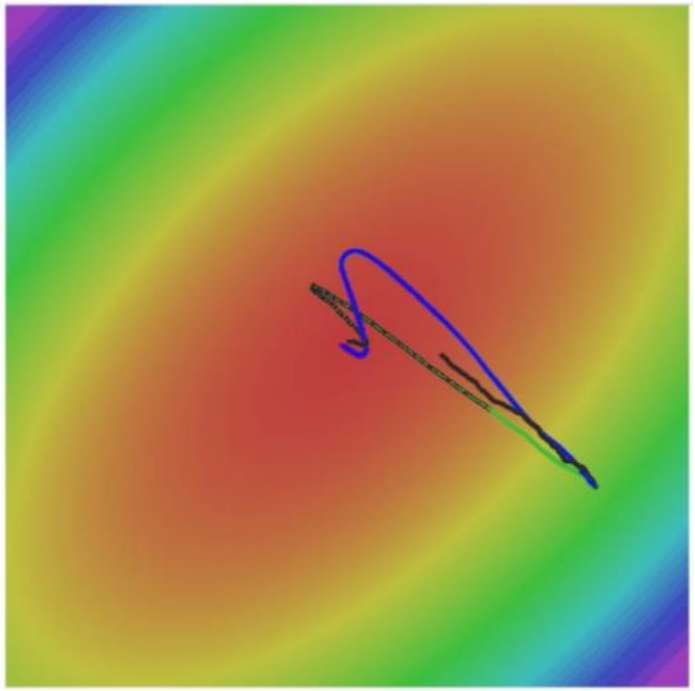
- Velocity는 초기값을 항상 0으로 함.(Not hyperparameter)





단점은 convex 하게 최적화된곳에서는 성능이 좋지만 Neural Network 처럼 non convex한 환경에서는 그다지 성능을 보여주지 못함.

Nesterov Momentum



SGD

SGD+Momentum

Nesterov

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

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

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

Nesterov는 속도에 에러값을 보정하고 local minima를 지나 global minima(이 경우)로 수렴하게 해주고 overshooting 도 줄여줌.

AdaGrad

- Scalaing개념. Gradient의 제곱을 계속더한값으로 나눠줌.
- $step1)\ G = G_0^2\ , update\ \frac{1}{\sqrt{G}} G_0\ step2)\ G = G_0^2 + G_1^2, update\ \frac{1}{\sqrt{G}} G_1 .. over\ and\ over$
- Problem is G 값이 너무 커짐. 점점 업데이트 속도가 느려짐.

→ Convex에서는 괜찮지만 non convex의 경우 local minima, saddle point에 안착하게됨.

RMSProp

- AdaGrad의 대안
- 아이디어는 유지하면서

Local minima 등에 안빠지게

속도를 조절!

Decay rate 를 곱해주자!

Almost decay = 0.9 or 0.99

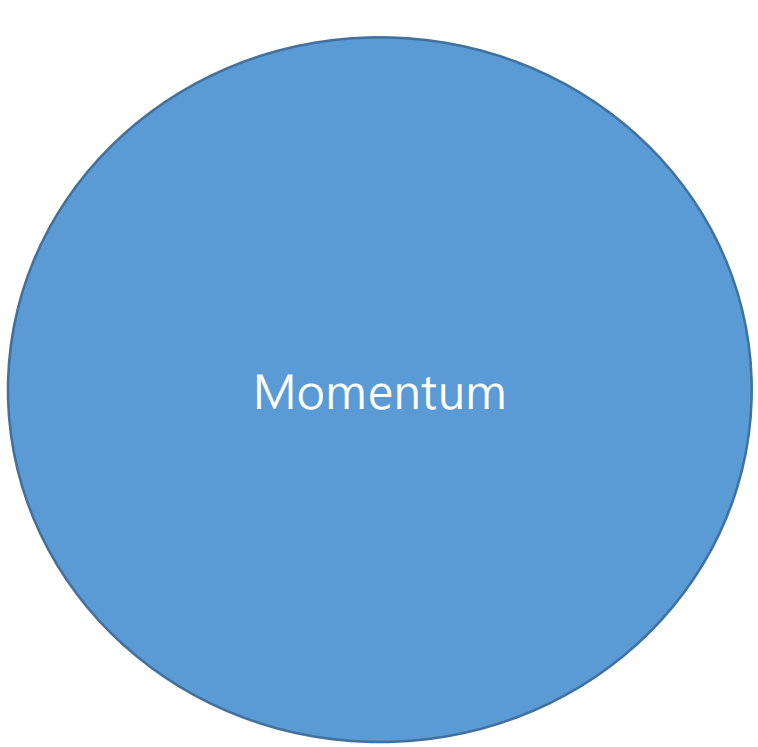
AdaGrad

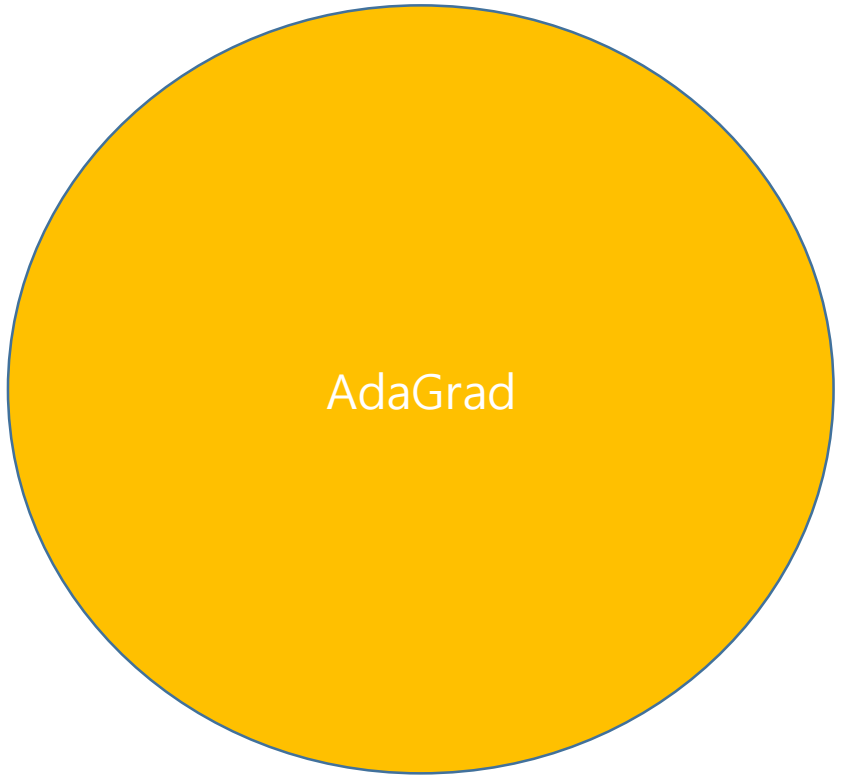
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

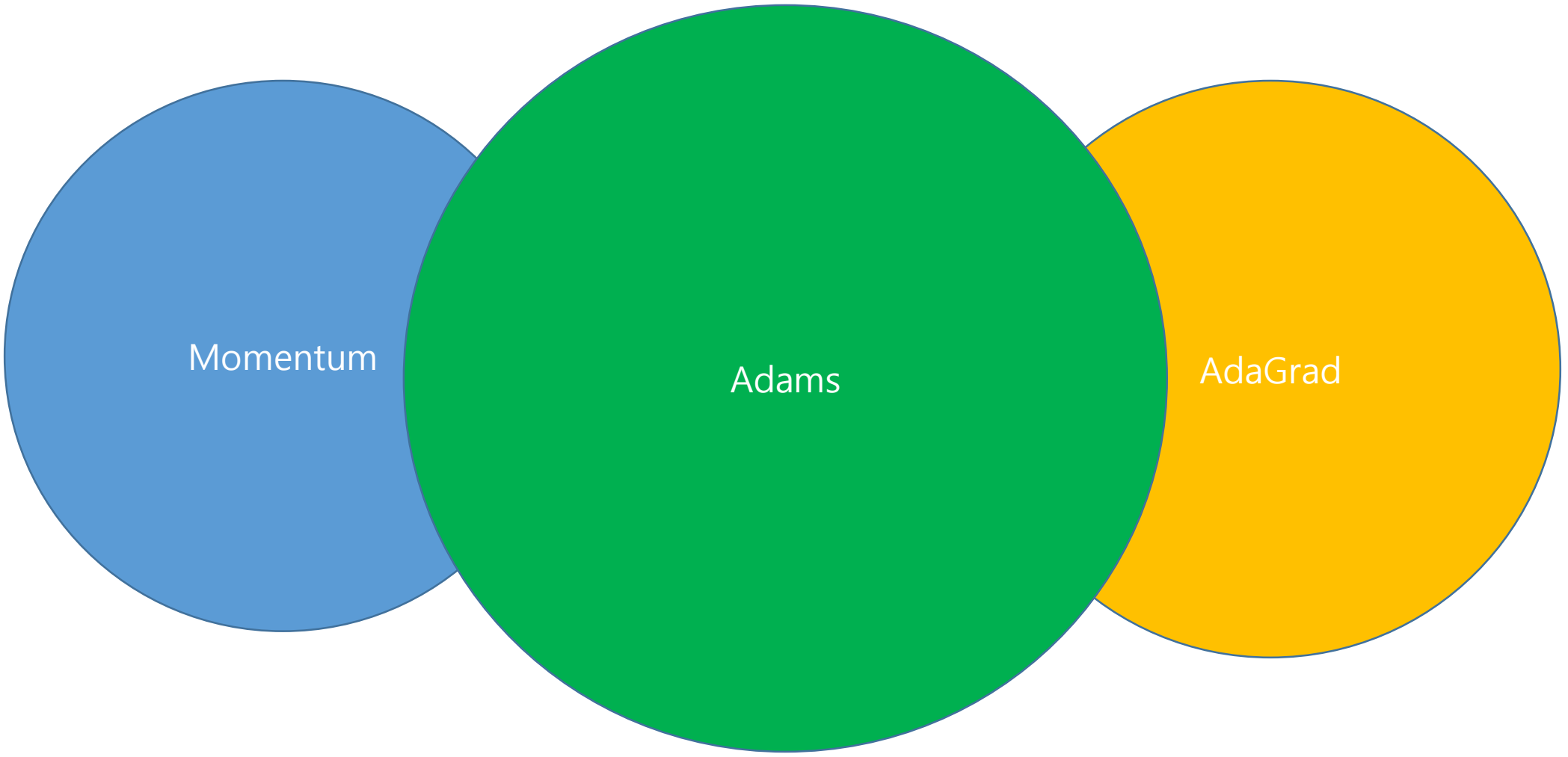
RMSProp

```
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)
```

Optimization….
Two types idea, and still both are good idea






Momentum
Adams
AdaGrad

Adams(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

작은 문제가 있음. second momen가 처음에 너무 작은값이라 step 이 너무 큼.

조정이 필요함

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

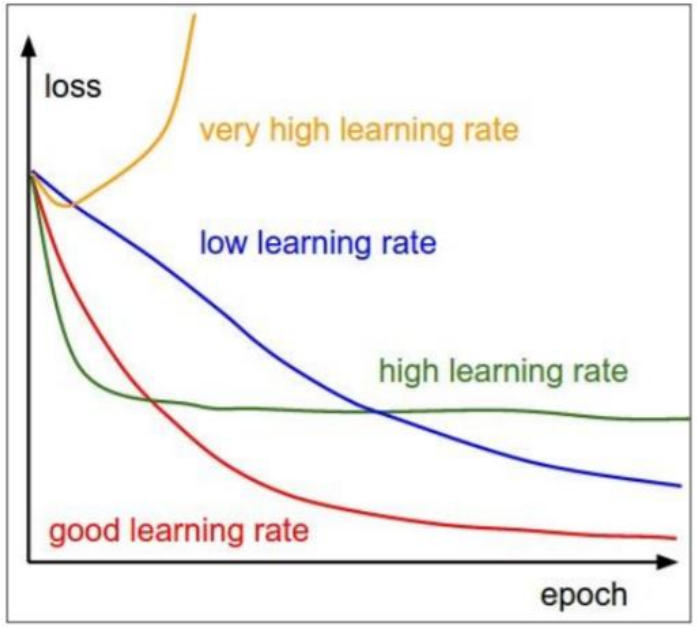
AdaGrad / RMSProp

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!

Learning rate..

- SGD, AdaGrad, Momentum, Adam,, all have learning rate term.
- Learning rate is hyperparameter. 하나로 고정하지 않음.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

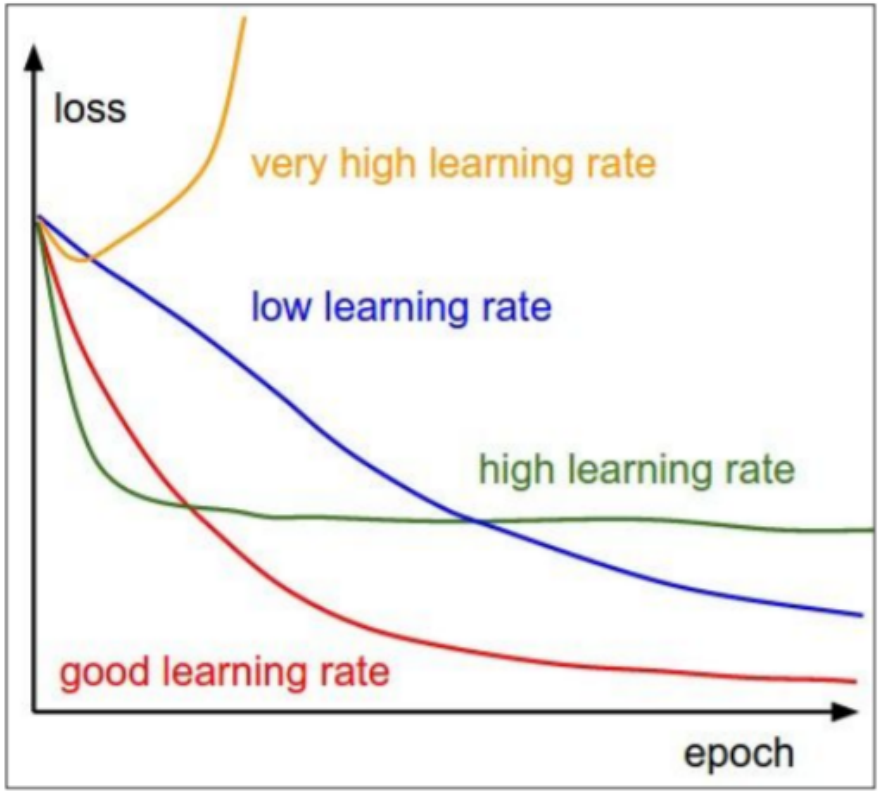
exponential decay:

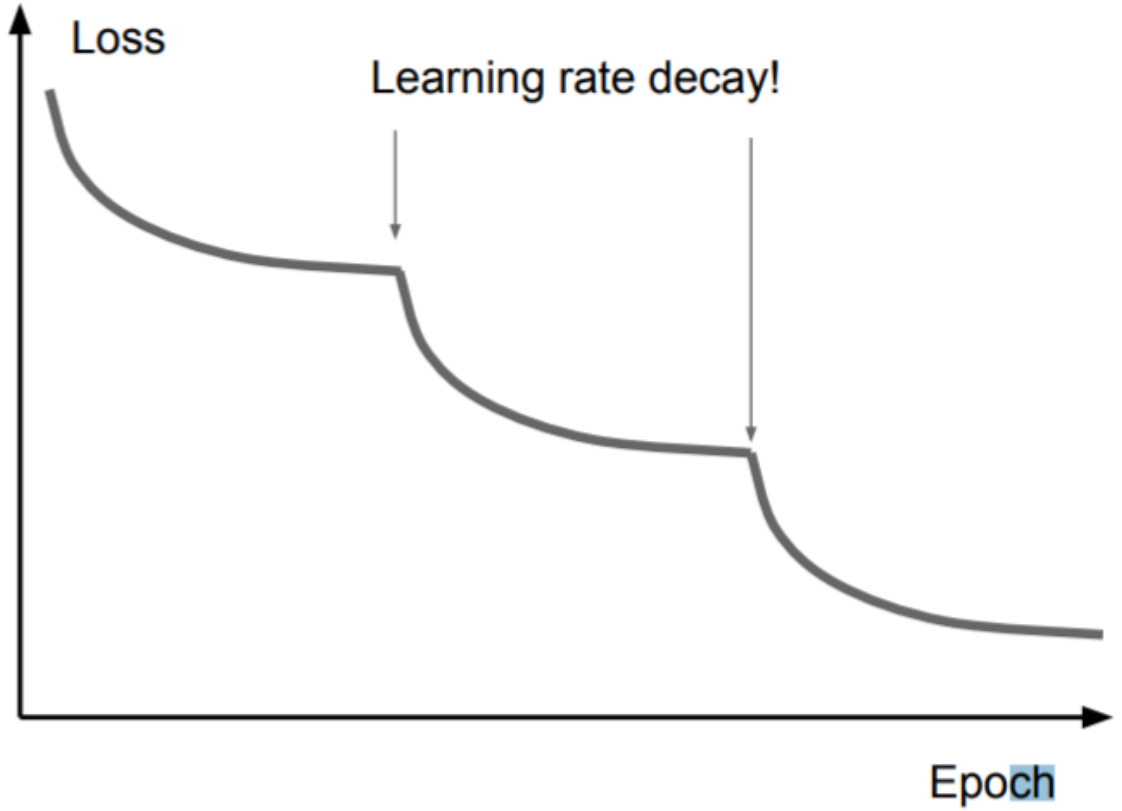
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

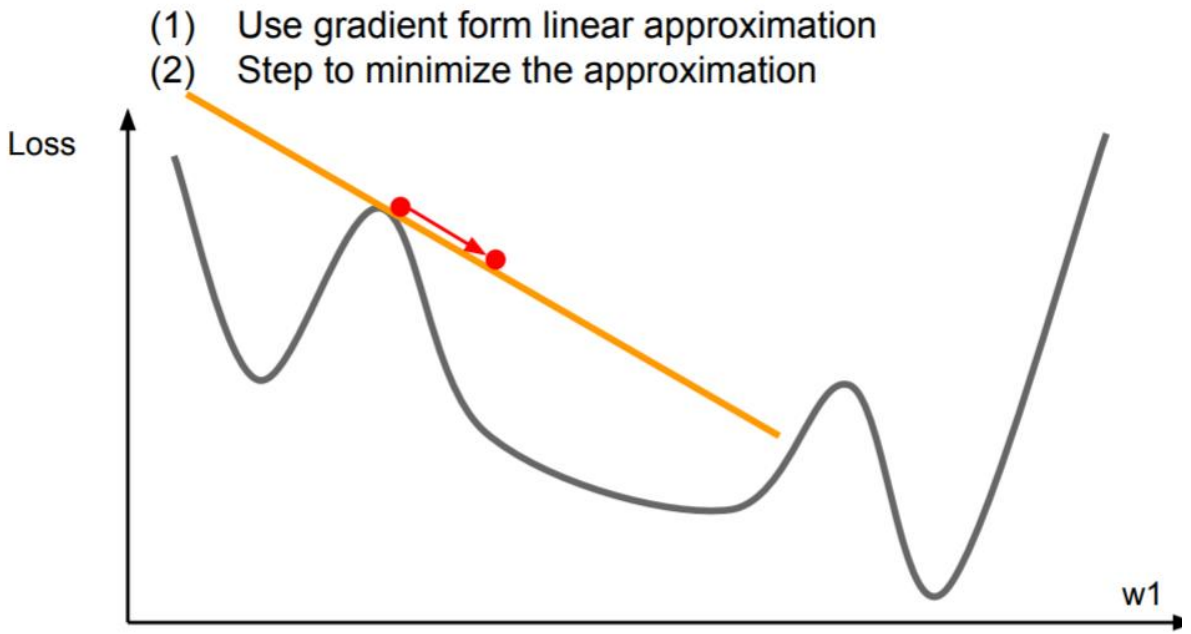
From ResNet

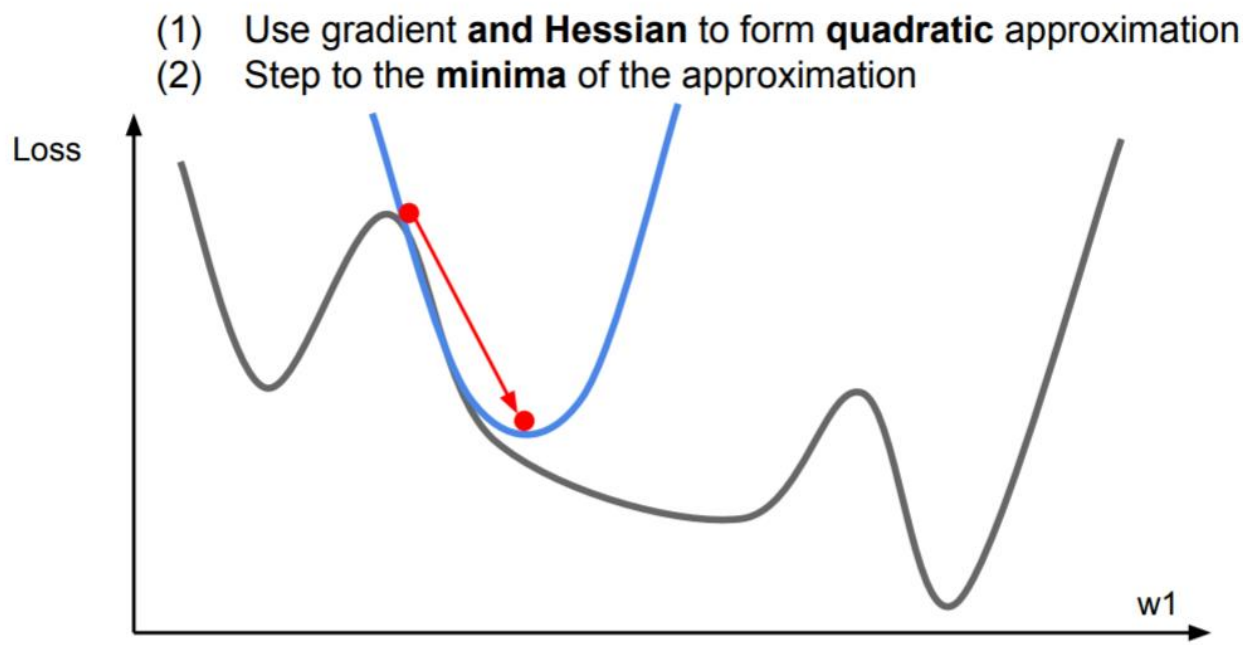




Decay가있다고 하더라도 초기화는 중요하다. 처음에 잘선택하자.

First-Order and Second-Order optimizer





최종적으로는 learning rate 가 필요하지만 지역적으로는 필요없음. 항상 로컬 미니멈을 찾음. 하지만 hessian matrix는 계산 비용이 너무 큼(inverse $O(N^3)$)

second-order Taylor expansion:

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

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

L-BFGS

- Hessian 을 근사시켜 계산비용을 조금 줄임.

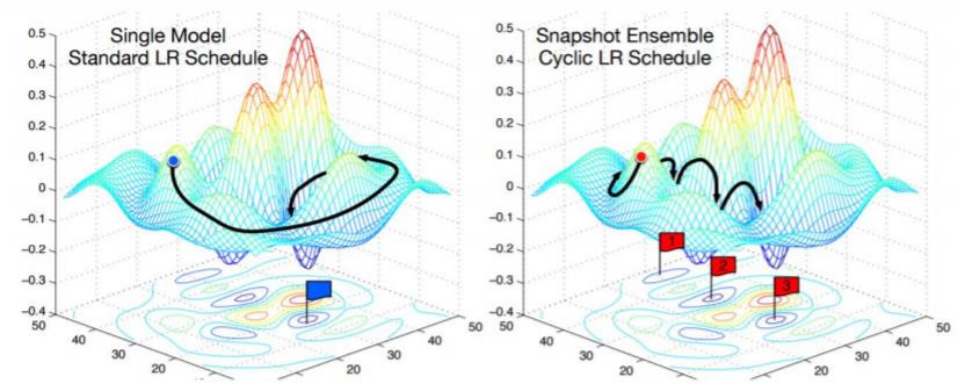
- 그래도 deep neural network에서는 잘 안씀. 조금 변형에서 간간히 쓰일곳이있음. → 확률적인 모델을 잘 다룰수가없음. Non-convex에서는 성능이 별로임.

Beyond Training Error

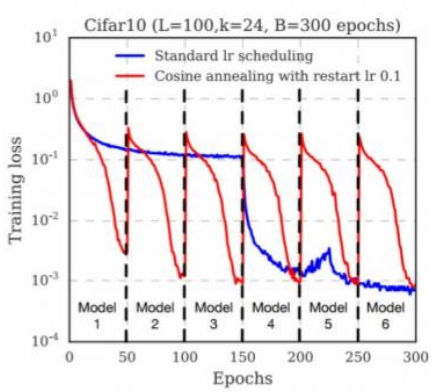
- Training error를 줄이기 위해 노력했음. 하지만 결국 중요한건 한번도 보지않은 데이터에 대한 결과임.(unseen data)
- Train Data와 real data(unseen data)의 갭을 어떻게 줄일까?

- Model Ensembles
  - Train Model independently and ensemble.

Tricks and tips



Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.



Cyclic learning rate schedules can make this work even better!

- 여러 개 모델을 돌리는 대신 하나의 모델을 여러 스냅샷으로 트레이닝.

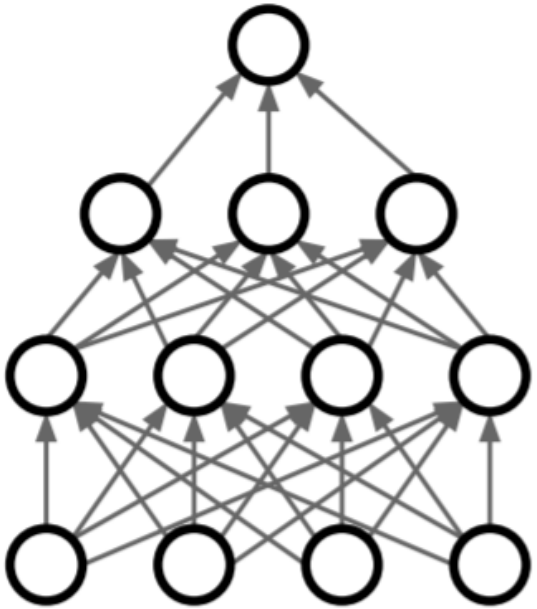
```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

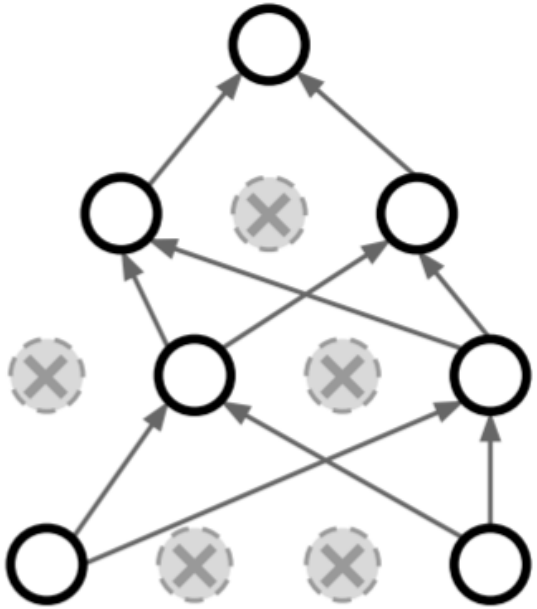
- 실제 파라미터를 사용하지 않고 파라미터의 평균을 계속 갱신해 사용. 조금 성능 향상. 잘 안씀.

No ensemble, single model

- Multiple model ensemble 은 효율이 떨어짐. 어떻게 성능을 올릴까.? →Regularization!
- 기존에 배운 L2,L1은 NN에서는 별로 좋진않음.
- Drop out 이라는 방법을 사용

Drop out





Forward pass 에서 임의의 뉴런을 0으로 만들자.

1. Overfitting 을 막아줌.

2. 매번 다른 뉴런을 drop out 하기 때문에 여러 모델을 앙상블한 효과.

Drop out

Test Time에서 랜덤이있는건 별로 좋지않다. 고양이를 어제는 고양이로 오늘은 개로 인식하면 매우 이상할것.

기댓값을 average out 시켜줘야함.

Dropout: Test time

Dropout makes our output random!

Output (label) Input (image) Random mask

$$y = f_W(x, z)$$

Want to "average out" the randomness at test-time

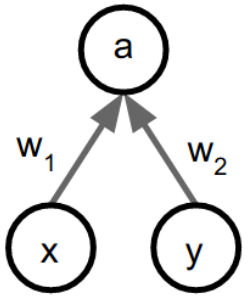
$$y = f(x) = E_z\left[f(x,z)\right] = \int p(z) f(x,z) dz$$

But this integral seems hard …

Drop out

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E[a] = w_1 x + w_2 y$

During training we have:

$$\begin{aligned} E[a] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

At test time, **multiply** by dropout probability

Drop out

```python
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Dropout Summary

drop in forward pass

scale at test time

Test time은 성능을 빠르게 하고 싶다.

Drop out

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Train 에서 미리 반영. →
Drop out을 사용하면
Train시간이 늘어남.

test time is unchanged!

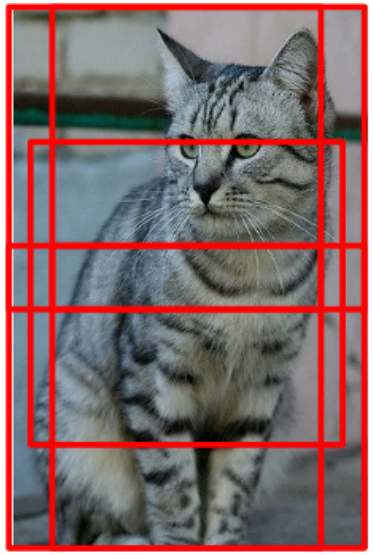
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Testing: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Regularization 효과.

Common Pattern

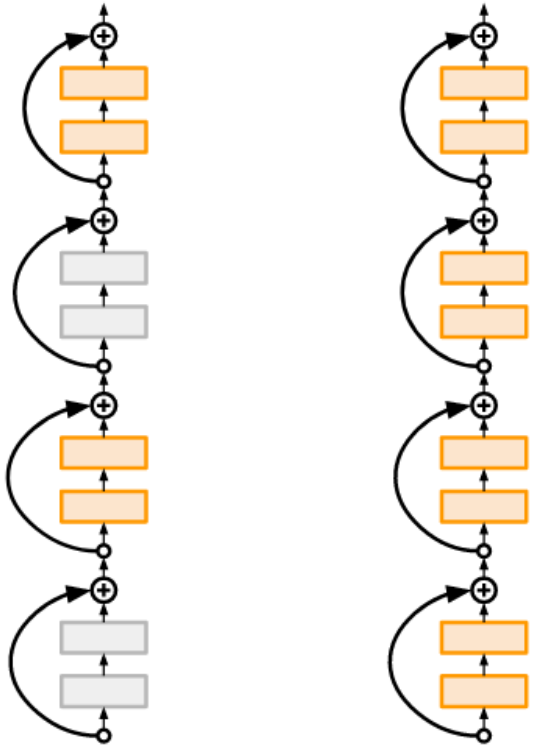
Batch norm이 가장일반적임. 이걸로 regularization이 모자르다면 drop out 추가. 그 이상은 잘없음.

Regularization: A common pattern

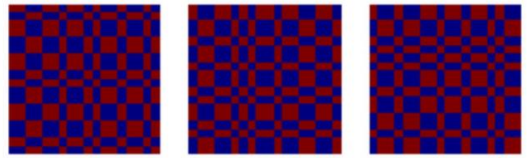
Training: Add random noise
Testing: Marginalize over the noise

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth

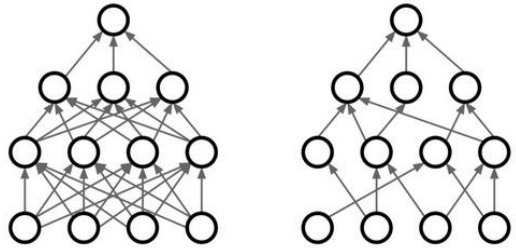
Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016



Stochastic Depth



Fractional Max pooling



DropConnection

Thank you

Q&A