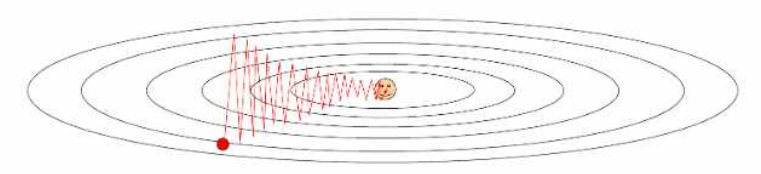
cs231n lecture 7

1. Optimization Algorithm

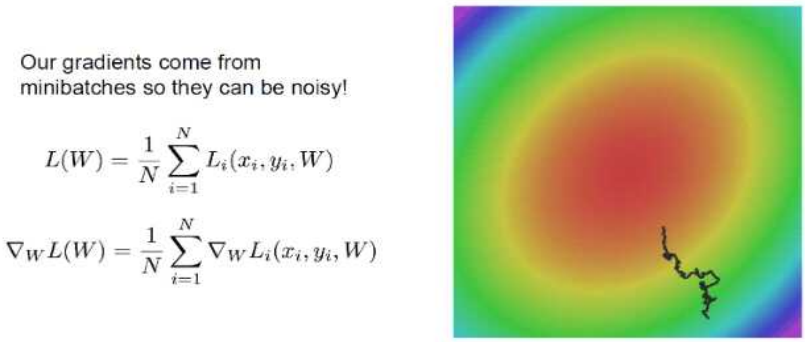
- SGD Algorithm의 문제점 :

1) Loss의 방향이 한 방향으로만 바뀌고 반대 방향으로는 느리게 바뀌는 경우, 즉 불균형한 방향이 존재한다면 잘 동작하지 않음



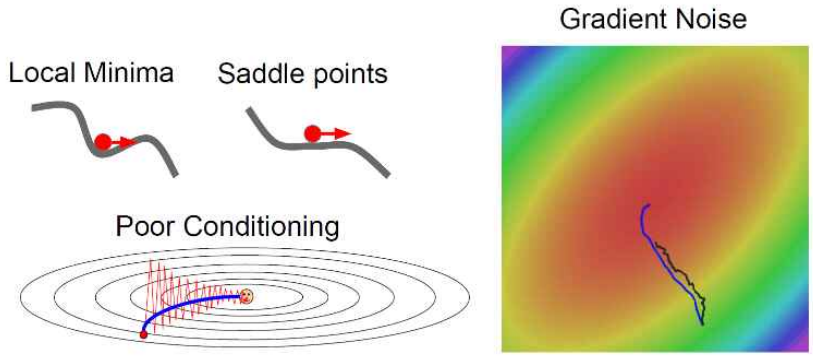
2) Local minima나 saddle point에 빠져서 나오지 못하거나 기울기가 완만한 구간에서 update가 잘 이뤄지지 않을 수 있음

3) Minibatch에서 gradient값이 노이즈 값에 의해 많이 변해 꼬불꼬불한 형태로 gradient값이 update 될 수 있음

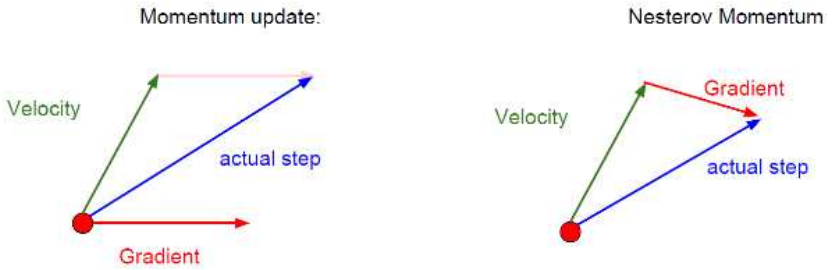


-Momentum: 위의 SGD 문제점을 해결하기 위해 도입

: 가고자 하는 방향의 속도를 유지하면서 gradient update 진행



- Nestrov Momentum : 기존의 momentum과 다르게 순서를 바꾸어 update 시킴



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

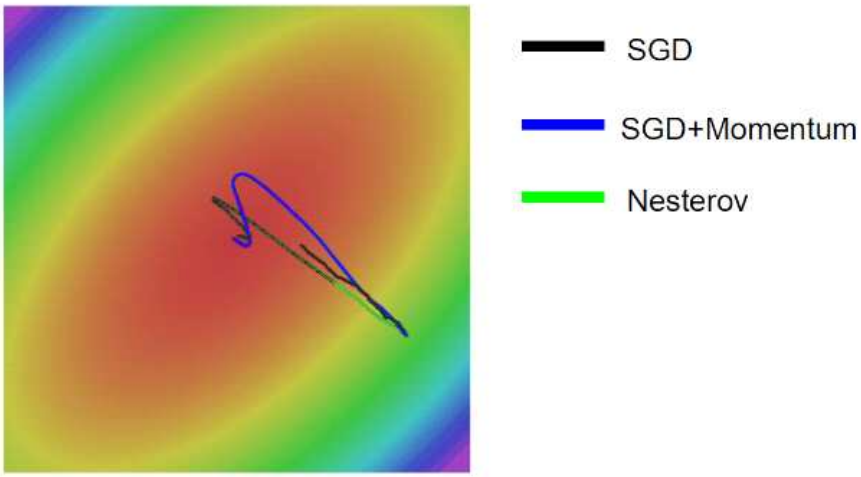
Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

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

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

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

- SGD, SGD+Momentum, Nesterov 결과값 비교



- AdaGrad : Velocity term 대신 grad squared term 이용해 grad update
  : 학습률을 효과적으로 정하기 위해 제안
  : update를 계속 진행하게 되면
  small dimension에서는 가속도가 늘어나고,
  large dimension에서는 가속도가 줄어듬.
  시간이 지나면 지날수록 step size는 점점 줄어듬

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

- RMSProp : AdaGrad 단점 보완한 방법
  : decay_rate 변수를 통해 step의 속도를 감속 가능
  : 과거의 모든 기울기를 균일하게 반영해주는 AdaGrad와 달리,
  RMSProp은 새로운 기울기 정보에 대하여
  더 크게 반영하여 update를 진행

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)
```

- Adam : momentum + AdaGrad

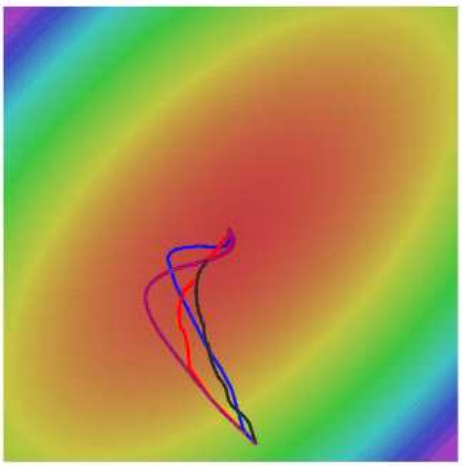
Adam (full form)

```
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

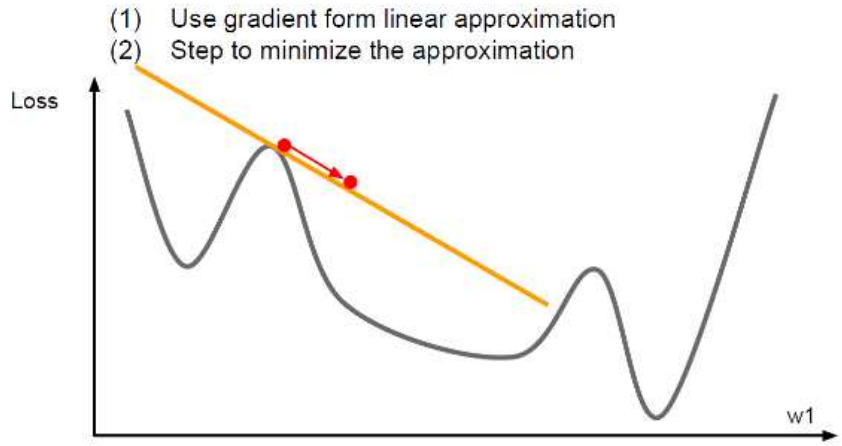
AdaGrad / RMSProp



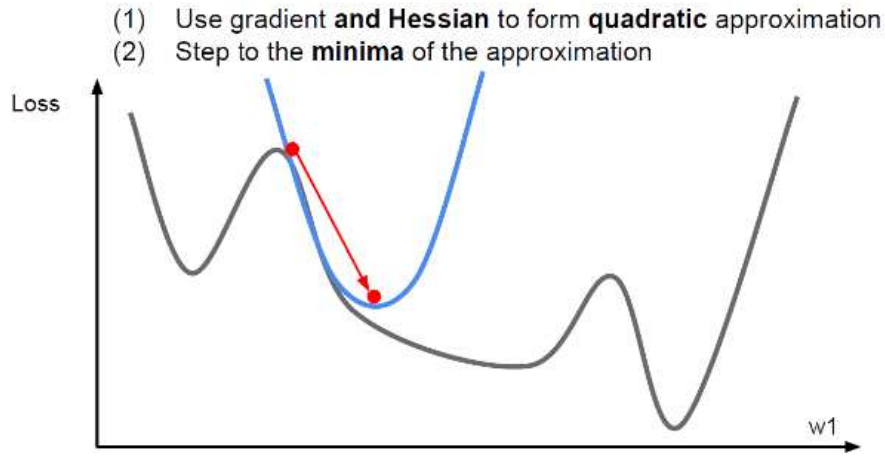
- 상황에 따라 최적의 optimization algorithm이 다름
  앞의 optimization algorithm은 모두 Learning rate를 hyperparameter로 가짐

2. First-Order & Second-Order Optimization

- 일차 함수로 근사화를 시켜 최적화를 시킬 때는 멀리 갈 수 없음



- 이차 함수로 근사화를 시킬때는 주로 테일러 급수를 이용
- 기본적으로 learning rate를 설정해 주지 않고 update 가능하다는 장점 (No Hyperparameters!)
- 복잡도가 너무 크다는 단점



- 이차 함수로 근사화 시키는 것은 Quasi-Newton 방법, non-linear한 최적화 방법 중 하나

- Newton methods보다 계산량이 적어 많이 쓰이고 있는 방법

- 그 중 가장 많이 쓰는 알고리즘은 BGFS와 L-BGFS이다.

- full-batch일 때는 좋은 성능을 보여서 Stochastic(확률론적) setting이 적을 경우 사용해 볼 수 있음

- 위의 방법들은 모두 Training 과정에서 error를 줄이기 위해 사용하는 방법들

3. Regularization

- loss function을 구현할 때 regularization에 대한 function을 추가

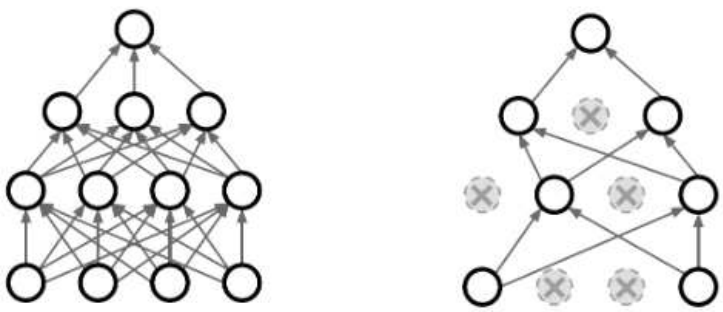
$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l}\rvert$
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}\rvert$

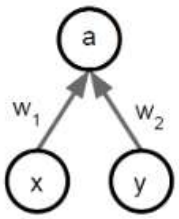
- dropout : 다양한 feature를 이용하여 예측을 하기 때문에
어떤 특정 feature에만 의존하는 경우를 방지
: test time 도 줄어들게 할 수 있음

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



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:

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

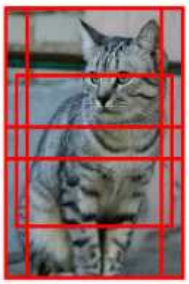
At test time, **multiply** by dropout probability

- Data Augmentation
  : Training을 시킬 때, 이미지의 patch를 random하게 잡아서 훈련을 시키는 경우
  : 이미지를 뒤집어서 train dataset에 추가해 훈련을 해주는 경우
  : 밝기값을 다르게 해서 train dataset에 추가하고 훈련을 해주는 경우

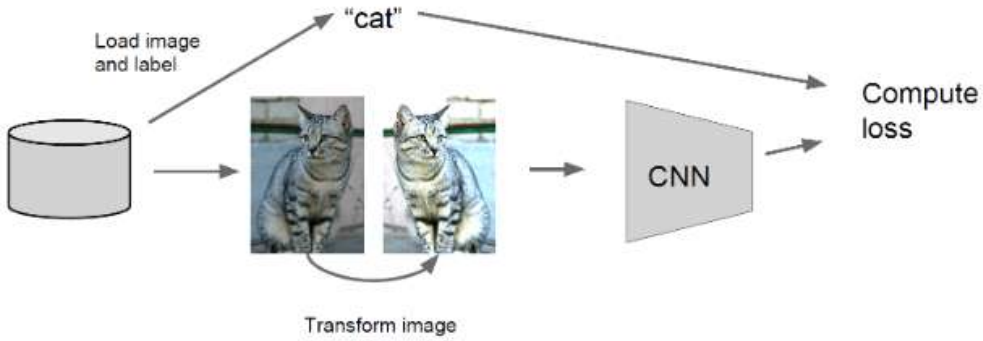
Data Augmentation

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



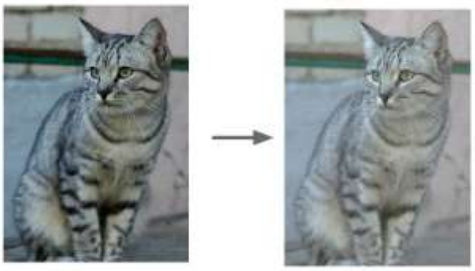
Testing: average a fixed set of crops
ResNet:
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips



Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a "color offset" along principal component directions
3. Add offset to all pixels of a training image

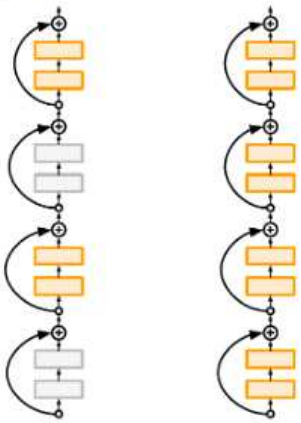
(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

- 이외에도 다양한 regularization 방법 존재

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

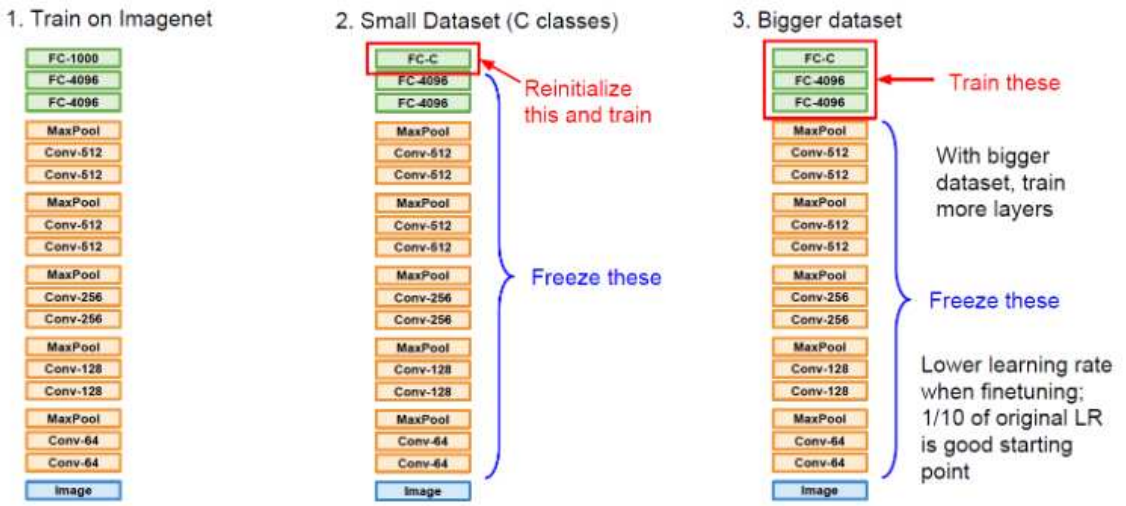


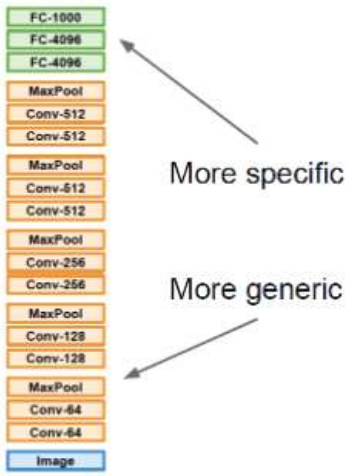
4. Transfer Learning

- 이미 pretrained된 모델을 이용하여 우리가 이용하는 목적에 맞게 fine tuning하는 방법

- Small Dataset으로 다시 training 시키는 경우 보통의 learning rate보다 낮춰서 다시 training

- Data set이 조금 클 경우, 좀 더 많은 layer들을 train





	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers