exam 개인학습

Lec 7

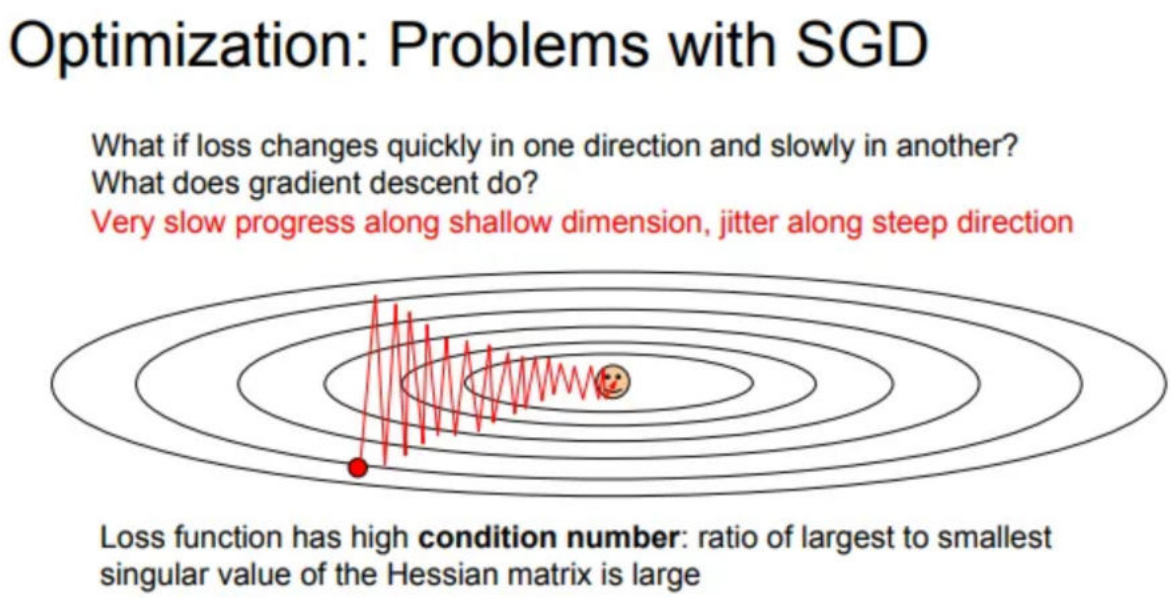
AI summary Try on this page

Type something...



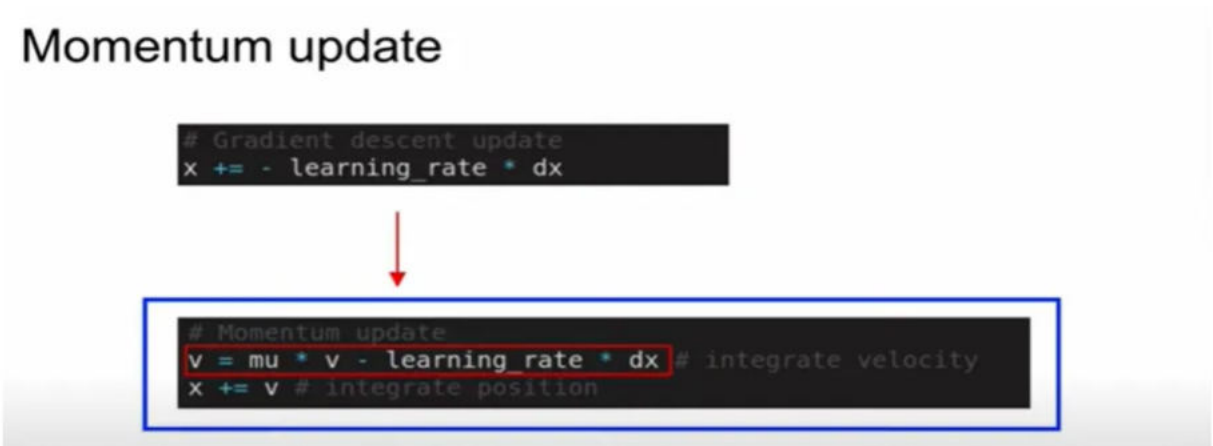
# 01) Training Neural Networks 2

## 01_Fancier optimization




- 매우 느린 수렴 속도를 보인다.

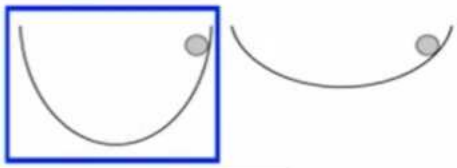
### 1. Momentum



- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
- mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

## Momentum update

```
# Gradient descent update
x += - learning_rate * dx
```



```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- Allows a velocity to "build up" along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

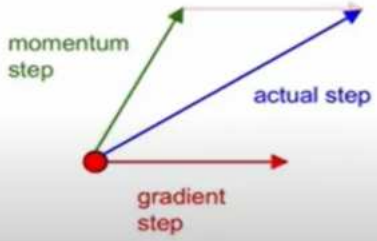
- momentum을 통해서 빠르게 수렴한다.

2. Nesterov Momentum update

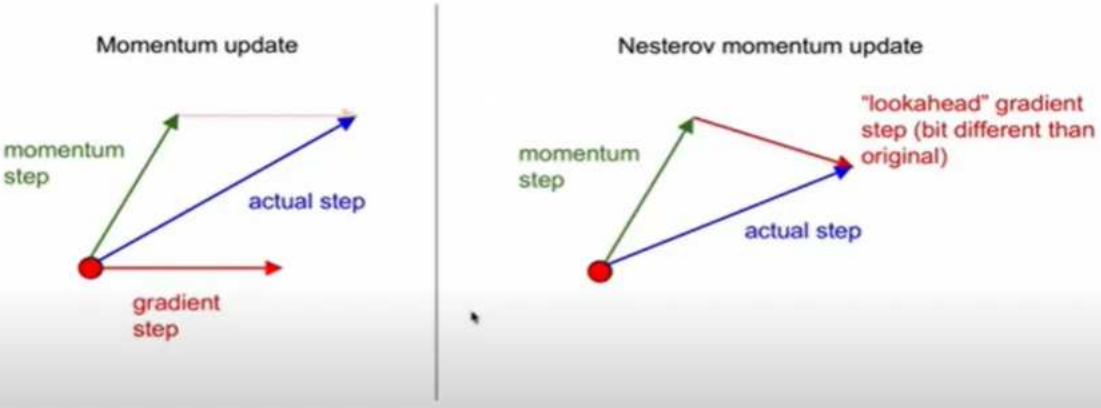
## Nesterov Momentum update

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Ordinary momentum update:




## Nesterov Momentum update




- 시작점을 shift한 후 계산한다.

## Nesterov Momentum update

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$
$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

- 하지만 계산을 하기 쉽지가 않다.

## Nesterov Momentum update

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$
$$\theta_t = \theta_{t-1} + v_t$$

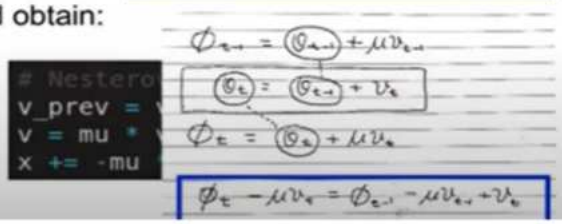
Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Variable transform and rearranging saves the day: $\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}$

Replace all thetas with phis, rearrange and obtain:

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\phi_{t-1})$$
$$\phi_t = \phi_{t-1} - \mu v_{t-1} + (1+\mu) v_t$$




- 그래서 새로운 변수를 도입해서 계산을 해준다.

$$v_{prev} = v$$

$$v = mu * v - learning\_rate * dx$$

$$x += mu * v_{prev} + (1 + mu) * v$$

### 3. AdaGrad update

## AdaGrad update

[Duchi et al., 2011]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

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

- per-parameter adaptive learning rate method

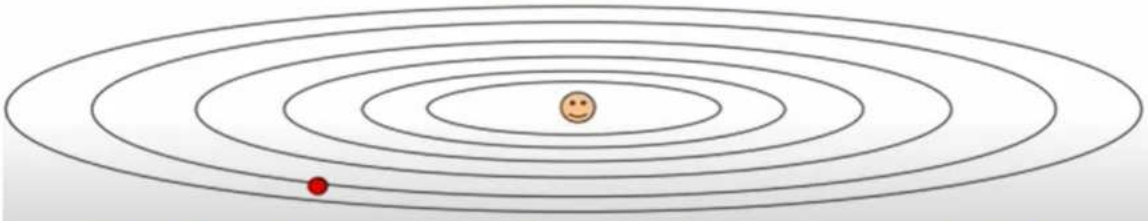
$$cache += dx^2$$

$$x+ = -learning_rate * dx/(np.sqrt(cache) + 1e^{-7})$$

- gradient가 커지면 x의 업데이트 속도가 낮아지고 gradient가 작으면 x의 업데이트 속도가 빨라진다.

AdaGrad update

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Q2: What happens to the step size over long time?

- step size가 시간이 지남에 따라 learning_rate가 0에 매우 가까워질 것이다. 따라서 학습이 종료가 될 것이다.

**4. RMSProp**

- AdaGrad의 문제점을 보완하기 위해 나온 방식이다.

RMSProp update [Tieleman and Hinton, 2012]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

$$cache = decay\_rate * cache + (1 - decay\_rate) * dx^2$$

- decay_rate값을 적용해서 cache값이 서서히 줄어든다.

**5. Adam**

Adam update [Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

- rms + momentum

$$m = beta1 * m + (1 - beta1) * dx$$

$$v = beta2 * v + (1 - beta2) * dx^2$$

$$x+ = -learning_rate * m/(np.sqrt(v) + 1e^-7)$$

## Adam update

[Kingma and Ba, 2014]

```
# Adam
m,v = #... initialize caches to zeros
for t in xrange(1, big_number):
  dx = # ... evaluate gradient
  m = beta1*m + (1-beta1)*dx # update first moment
  v = beta2*v + (1-beta2)*(dx**2) # update second moment
  mb = m/(1-beta1**t) # correct bias
  vb = v/(1-beta2**t) # correct bias
  x += - learning_rate * mb / (np.sqrt(vb) + 1e-7)
```

momentum

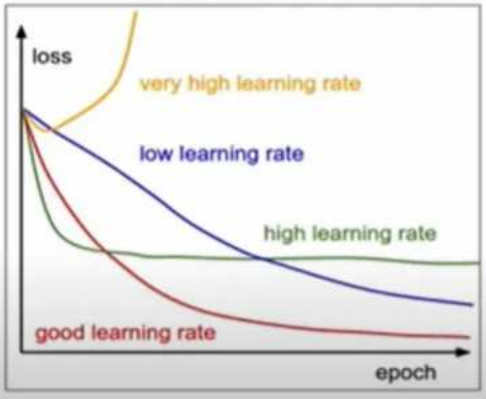
bias correction
(only relevant in first few iterations when t is small)

RMSProp-like

The bias correction compensates for the fact that m,v are initialized at zero and need some time to "warm up".

- 최종형태는 correct bias부분을 추가해준 것이다.
- 초기화할 때 0근처이면 scaling up을 해주는 과정이다.

## SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.




**=> 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:**

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

- 처음에는 빠르게 수렴하다가 나중에 갈수록 전전히 수렴하는 방식을 사용하는 게 좋다. 그래서 learning_rate를 갈수록 점차 줄여가는 방식을 사용한다.
- ex) step decay, **exponentail decay**, 1/t decay
- 기본값으로는 Adam을 많이 사용한다.

### 6. Second order optimization methods

## Second order optimization methods

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:

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

Q: what is nice about this update?

- converge가 매우 빠르고 learning_rate가 필요없다.

## Second order optimization methods

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:

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

notice:
no hyperparameters! (e.g. learning rate)

Q2: why is this impractical for training Deep Neural Nets?

- 현실적으로 매우 큰 행렬의 역행렬을 구하는 것은 매우 비싸기 때문에 불가능하다.

## In practice:

- **Adam** is a good default choice in most cases
- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

- 정리하면 Adam이 일반적으로 좋은 선택이다.
- full batch를 사용할 여력이 된다면 L-BFGS도 고려해볼만 하다.

**02_Model Ensembling**

1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance

- 독립적은 모델을 여러개를 써서 평균을 내주면 성능이 개선되는 효과가 있다.

Fun Tips/Tricks:

- can also get a small boost from averaging multiple model checkpoints of a single model.
- keep track of (and use at test time) a running average parameter vector:

```
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
```

- 기존 값들을 반영해서 average하게 되면 성능 개선에 효과가 있다.

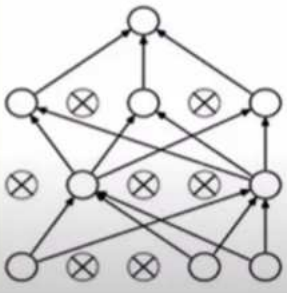
### 03_Regularization

BN < - > Dropout

#### 1. Dropout

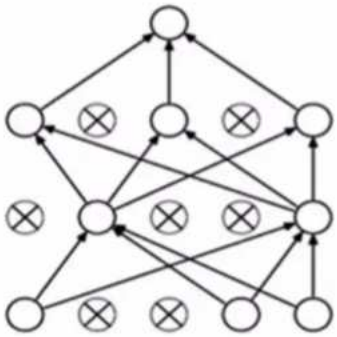
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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)
```

Example forward pass with a 3-layer network using dropout



- 특정 노드들만 활성화하고 나머지 노드들을 죽이는 방식이다.

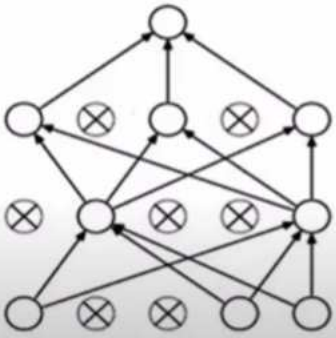
## Waaaait a second…
## How could this possibly be a good idea?



Forces the network to have a redundant representation.


- 하나의 노드가 다른 것들도 보는, 중복을 가지게 된다.

## Waaaait a second…
## How could this possibly be a good idea?



Another interpretation:

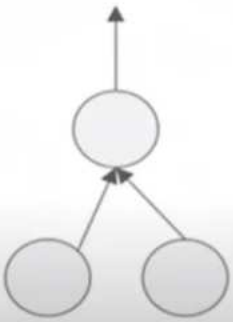
Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.

- ensemble을 주는 효과도 나온다.

## At test time….
Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



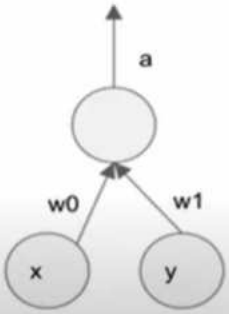
(this can be shown to be an approximation to evaluating the whole ensemble)

- test time 때는 dropout을 사용하지 않고 모든 노드들을 사용한다.

## At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).




during test: **a = w0*x + w1*y**

during train:

E[a] = ¼ * (w0*0 + w1*0

w0*0 + w1*y

w0*x + w1*0

w0*x + w1*y)

= ¼ * (2 w0*x + 2 w1*y)

= ½ * **(w0*x + w1*y)**

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training! => Have to compensate by scaling the activations back down by ½

- 그런데 여기서 주의해야할 점은 dropout을 통해서 output의 값이 줄어들었다는 점이다. 따라서 train 때 p로 값을 나눠주어서 scaling을 유지해줘야 한다.

---

## More common: "Inverted dropout"

```
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
```

test time is unchanged!