

Lecture 7 | Training Neural Networks II

지난 시간에는 neural network를 학습하는 여러 방법들에 대해 배웠다. 이어서 이번 강에서도 다른 여러 방법들에 대해 알아 볼 것이다.

- Fancier optimization
- Regularization
- Transfer Learning

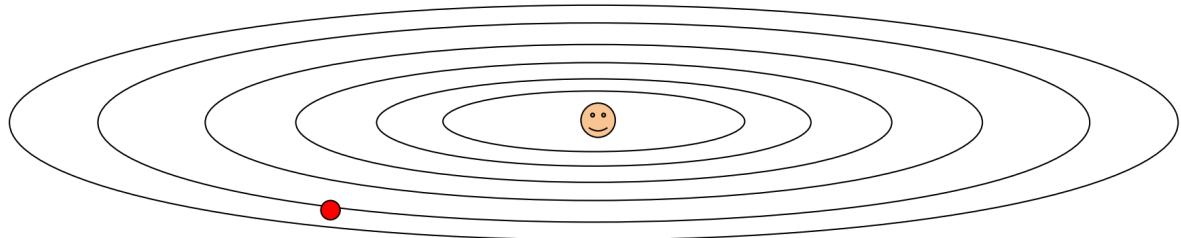
Optimization

우리가 gradient descent를 사용하여 loss가 가장 낮은 위치를 찾으려 할 때, 아래와 같이 하였다. 각 점에서 gradient를 구하고, 기울기가 감소하는 방향으로 weight를 더하는 식으로 반복하였다.

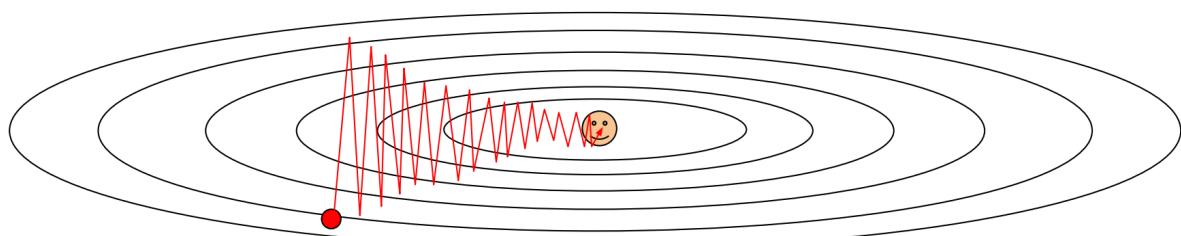
```
# Vanilla Gradient Descent

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

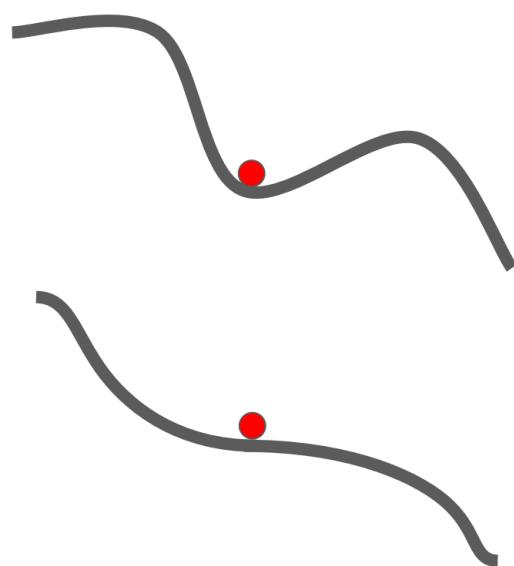
하지만 이렇게 하면 단점이 있었는데, 아래와 같은 상황이라고 가정해보자. 빨간 점이 현재의 값이고 정 중앙이 최적의 loss라고 하자. 이 상황에서 우리가 SGD를 사용하면 어떻게 빨간 점이 변화하면서 최적의 상태를 찾아갈까?



아래처럼 지그재그로 이동하며, 상당히 비효율적으로 최적의 loss값을 찾아나갈 것이다. 실제로 높은 차원에서는 이것이 더더욱 잘 들어난다고 한다. 정말 높은 차원에서라면 SGD가 정확한 값을 탐지하지 못할 경우도 있다고 한다.



뿐만 아니라, 아래 그림의 두 경우처럼 안정점에 존재하거나, 기울기가 0이 되는 경우에는 SGD에서 더이상의 변화가 생기지 않는다. 여기서 아래의 그림은 2차원에 불과하지만, 실제로 높은 차원에서는 안정점이 정말 많이 존재하기 때문에 SGD가 정상적인 결과를 낳지 못하는 경우가 많다. 뿐만 아니라, 기울기가 0이 아니더라도 기울기가 상당히 작으면 변화가 엄청 적을것이고, 속도가 상당히 느려지는 결과를 낳을 것이다.



이를 해결하는 많은 방법이 알려져 있는데, 그 중 하나가 Momentum을 추가하는 것이다.

SGD + Momentum

이는 속도를 구하고 그 속도를 이용하여 현재 상태를 계속 변화시켜 나가는 것이다. 이 때는 ρ 라는 hyperparameter가 존재하는데 이는 마찰계수라고 생각하면 된다.

SGD

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

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

SGD+Momentum

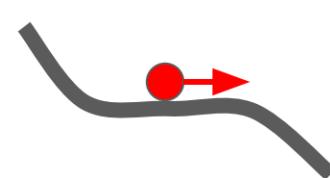
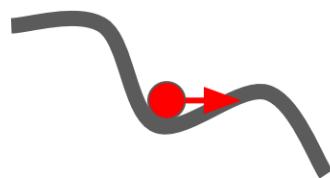
$$\begin{aligned} v_{t+1} &= \rho v_t + \nabla f(x_t) \\ x_{t+1} &= x_t - \alpha v_{t+1} \end{aligned}$$

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

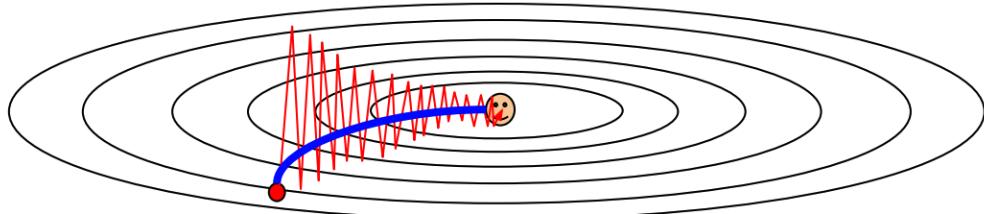
이렇게 해버리면 아래와 같이 안정점들에서도 멈추지 않고, 꾸준히 이동할 수 있다. 또 지그재그처럼 이동하지 않고 곧바로 최적의 위치를 찾아 이동하는 경향을 볼 수 있다.

Local Minima

Saddle points



Poor Conditioning



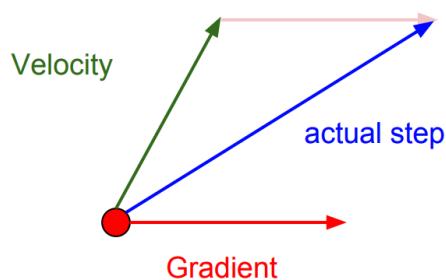
Nesterov Momentum

앞에서, SGD + Momentum의 경우 기존의 속도에 gradient를 더해서 이동할 위치를 결정했다.

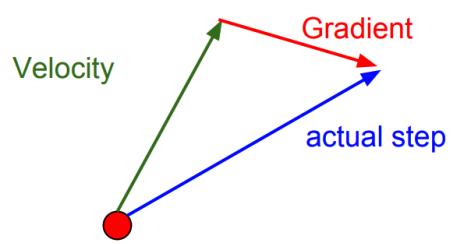
이것과 조금 다르게, Nesterov Momentum은 기존의 속도만큼은 어차피 이동할 예정이니 이동했다고 가정하고, 그 위치에서의 gradient를 구하여 최종적으로 이동할 위치를 결정하는 방법이다.

아래의 그림에서 보면 좌측의 그림은 SGD+Momentum 을 나타낸 것이고, 우측은 Nesterov Momentum을 나타낸 것이다.

Momentum update:



Nesterov Momentum



그런데 그렇게 되면 아래처럼 f 안에 $x_t + \rho v_t$ 가 들어가게 되고 이렇게 되면 다른 API들과 연동이 어렵다고 한다. 그래서 이를 조금 변형시켜서 아래처럼 식을 변형해서 사용한다고 한다.

Nesterov Momentum

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

$$x_{t+1} = x_t + v_{t+1}$$

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:

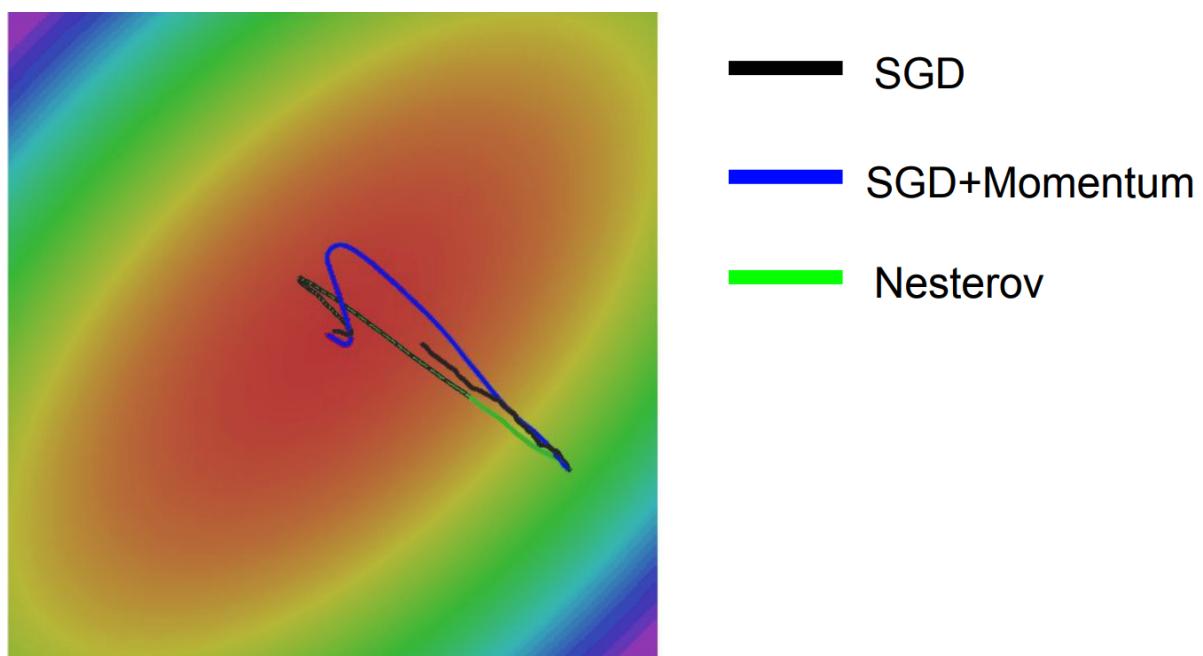
$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$

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

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

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

앞에서 설명한 3개를 비교해보면 SGD는 상당히 느리고, SGD + Momentum과 Nesterov는 훨씬 빠르다고 한다. 대부분 Nesterov가 더 빠르게 최적의 loss를 찾을 수 있다.



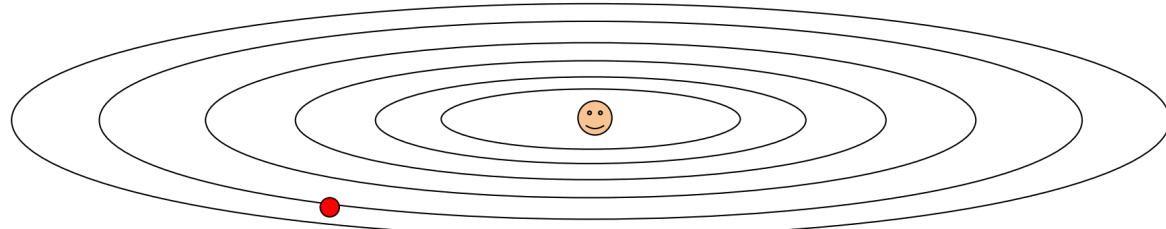
AdaGrad

다음은 AdaGrad가 제안한 방법으로, gradient의 제곱을 매번 더하면서 모으면서, 그 축적된 값의 제곱근을 나눔으로서 x 의 위치를 정하는 것이다.

이렇게 되면 어떤 일이 일어날까? 아래의 그림처럼 2차원의 형태가 있을 때, 세로 방향으로는 기울기가 꽤 있고, 가로방향으로는 기울기가 완만하다. 이러한 상황에서 기울기가 완만한 방향으로는 작은 값으로 나눠 많이 더하고, 기울기가 급한 방향으로는 큰 값으로 나눠 위치를 조금씩 변화시켜 나가는 것이다.

하지만 AdaGrad에도 하나의 단점이 있는데, 많은 시간동안 training을 하게 되면 축적된 값이 상당히 커지게 되고, 나중에는 계속 큰값으로 나누어져서 변화가 거의 없게 된다. 즉, 멈추게 되는 상황이 발생한다고 할 수 있다.

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



참고로, 나누는 값에 더해지는 굉장히 작은 값은 0으로 나누는 것을 방지하기 위한 값이라고 보면 된다.

RMSProp

AdaGrad의 단점을 개선한 방법이 바로 RMSProp이다.

계속 값을 축적해나가기보다는, 기존의 값을 어느 정도(decay_rate) 저장하면서, 새로운 데이터에 대한 정보도 포함시키는 방법이다. 이렇게 하면 시간이 지남에따라 값이 계속 커지게 되는 AdaGrad의 문제점은 해결할 수 있다.

AdaGrad

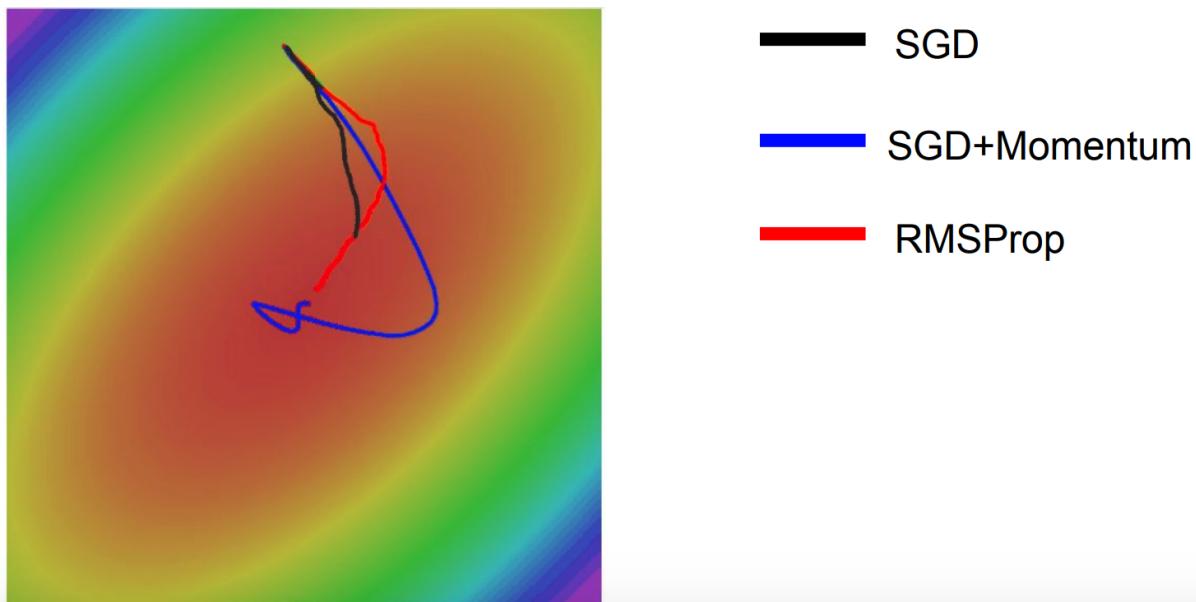
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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)
```

위에서 배운 것들을 직접 테스트해보면 그래프는 아래와 같다. 여기서, SGD+Momentum은 뒤로 한번에 꺾였다가 다시 되돌아오면서 최적의 위치를 찾는 형태를 보였다. 이에 비해 RMSProp은 처음부터 최적의 위치를 향해 다가가는 것을 볼 수 있었다. 하지만 초기에는 Momentum이 RMSProp에 비해 훨씬 빠르게 최적의 위치 근처로 다가가는 결과를 관찰할 수 있다. 그렇다면 이 둘을 합치면 보다 더 좋은 결과를 얻을 수 있지 않을까?



Adam

Momentum과 RMSProp의 장점을 모두 가지고 있는 것이 바로 Adam이다. 아래와 같이 속도를 나타내는 first_moment와 RMSProp에서 사용하던 second moment를 두어서 속도는 곱해주고, 제곱의 합은 나누어주는 식으로 진행하였다.

하지만 이렇게 하면 문제가 하나 발생한다. 완전 초기의 상황을 보면 first_moment와 second_moment의 값이 0일 것이다. 그리고, 이 값 둘다 기존의 데이터의 값을 어느정도 보존하면서, 새로운 변화에 대해 값을 변화시켜 나가는데 기존의 값이 0이기 때문에 초기에는 상당히 작은 값이 된다. 즉, 초기에 변화하는 속도가 굉장히 느리다는 것이다.

```
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

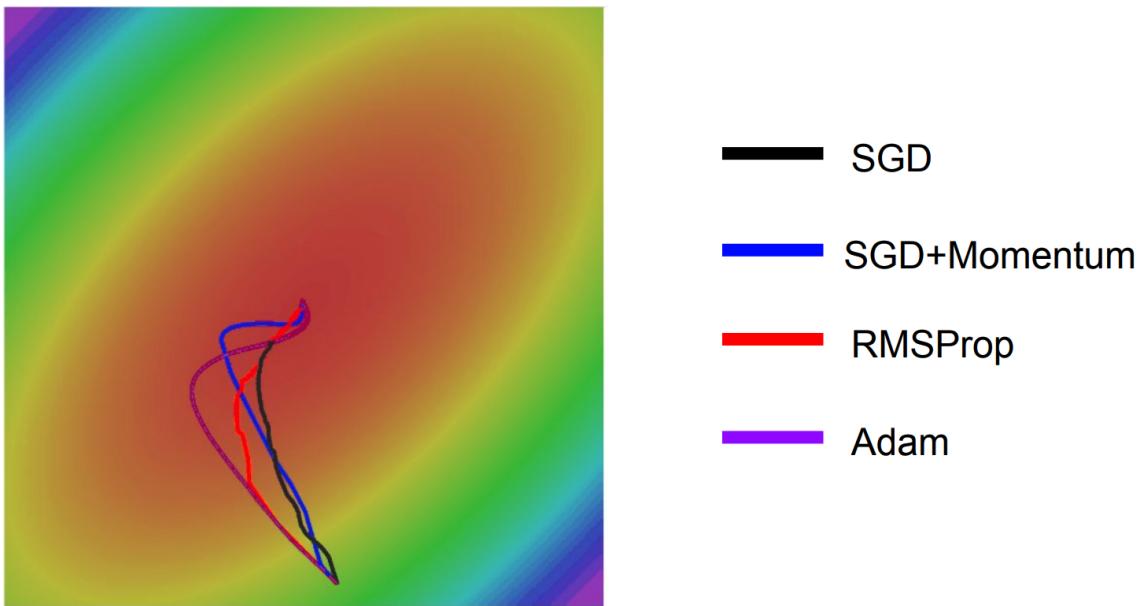
그런 단점을 해결하기 위해 Bias correction 코드를 넣었다. 시행 횟수에 대한 변수를 만들어서 초기에 굉장히 값이 작아지는 그런 문제를 해결해주는 것이다. 아래의 코드가 최종적으로 완성된 Adam 알고리즘에 대한 코드이다.

또 직접 실험해본 결과, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\text{learning_rate} = 0.001$ or 0.0005 정도로 하여 시작하면 좋다고 한다.

```
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

테스트해본 결과, Momentum보다는 덜 꺾이면서 빠르게 최적의 위치를 찾아나가는 모습을 관찰할 수 있었다.

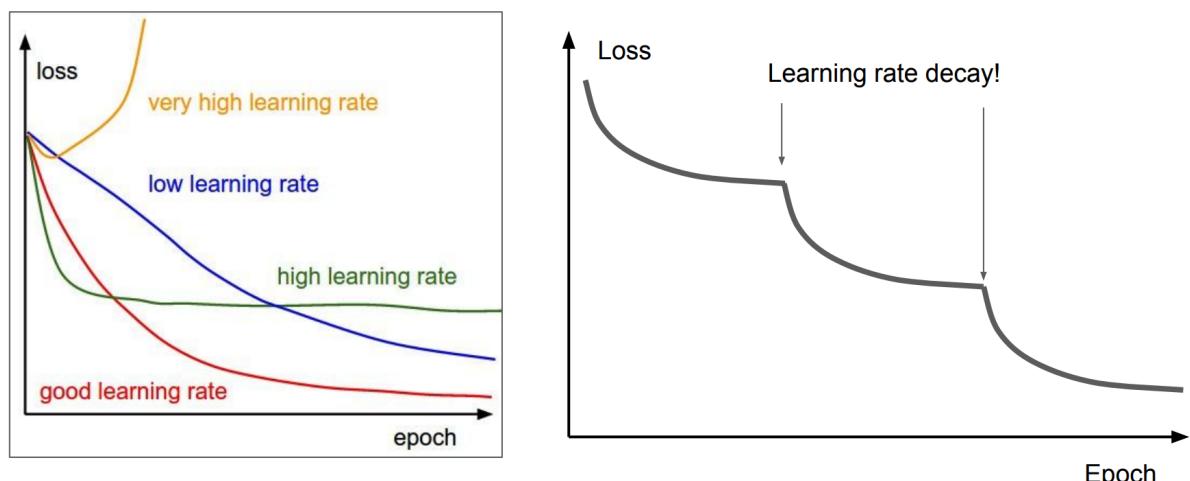


위에서 배운 모든 알고리즘에서는 hyperparameter로 learning rate를 사용한다. 그러면 어떻게 learning rate를 설정하여 사용하는 것이 좋을까?

아래의 그래프를 보면 learning rate가 크면 초기에 너무 많은 변화를 보이고, 작으면 적은 변화를 보인다는 것을 알 수 있다. 그렇다면, 많은 변화가 필요한 초기에는 learning rate를 크게 하되, 점점 그 값을 줄여나가면 빠르게 원하는 값을 찾을 수 있을 것이다. 본 강의에서는 총 3가지 방법이 제안되어 있다.

- step decay : 몇몇의 epoch마다 learning rate를 절반으로 감소
- exponential decay : $\alpha = \alpha_0 e^{-kt}$
- $1/t$ decay : $\alpha = \alpha_0 / (1 + kt)$

이처럼 decay를 사용하면 우측 그림과 같은 그래프를 보인다고 한다.



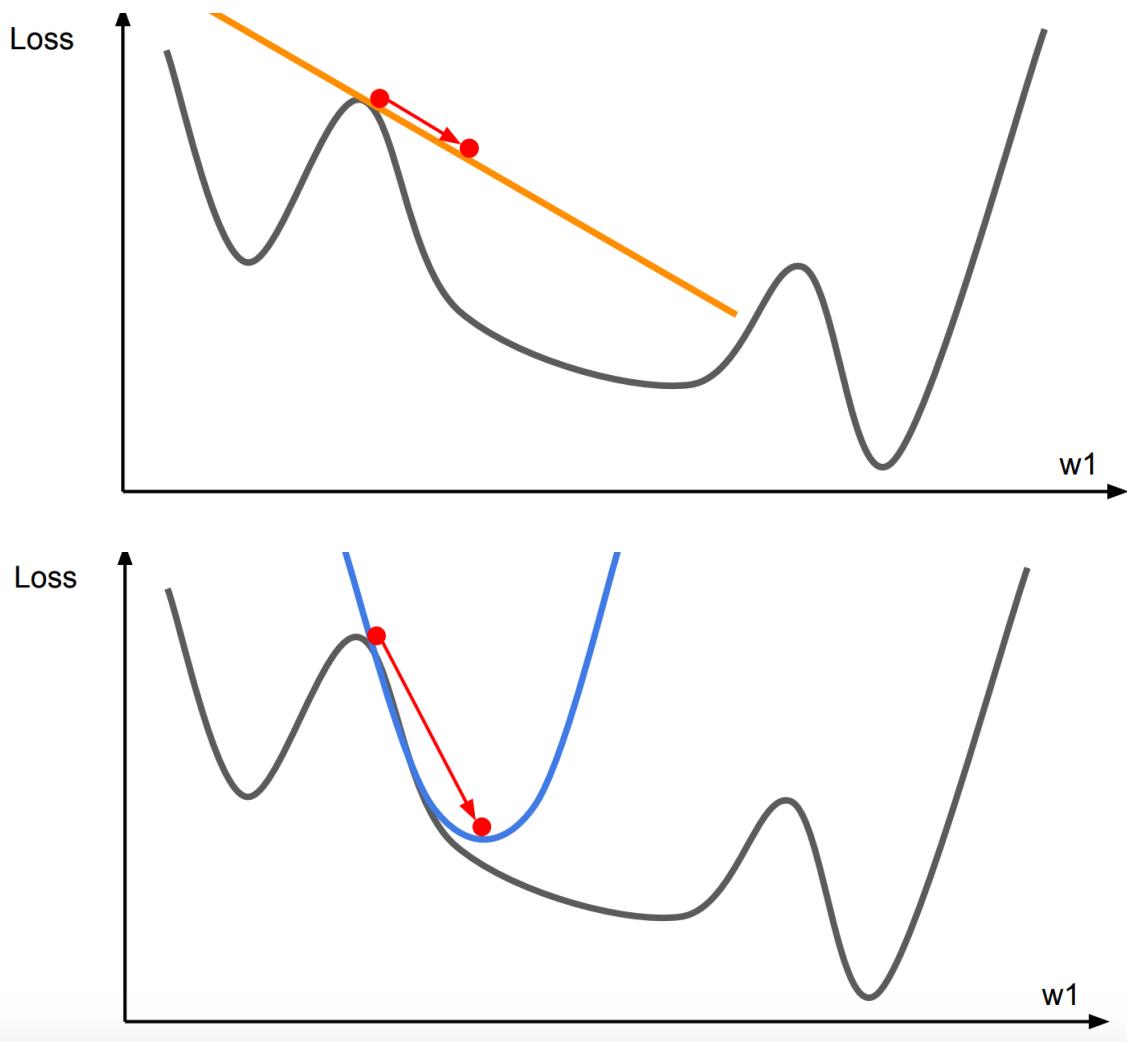
Second-Order Optimization

앞에서 우리는 gradient의 정보만을 사용하여 다음 상태를 예측하였다. 이런 최적화를 First-Order Optimization이라고 한다.

이제는 우리는 곡면의 형태도 이용하는 Second-Order Optimization을 설명하고자 한다. 곡면의 형태를 이용하게 되면 바로 최저점으로 이동할 수 있다고 한다.

그림으로 보면, 첫 번째 그래프가 First-Order Optimization, 두 번째 그래프가 Second-Order Optimization이다. First-Order Optimization에서는 기울기를 구하고, x 값을 조금씩 이동하고, 이를 반복하면서 Loss가 가장 작은 위치를 찾으려 했다.

이에 비해 Second-Optimization은 그래프처럼 곡면의 형태를 계산하고, 그 곡면에서 최저점을 찾아서 바로 그 지점으로 이동하는 방법을 사용하여 빠르게 loss가 작은 위치를 찾을 수 있다.



식은 아래와 같다.

이것의 특징은, learning rate가 필요 없다는 것이다. 바로 최저점으로 이동하기 때문이다. 하지만 이는 딥러닝에서 많이 사용되지는 않는다. Hessian은 주로 $n \times n$ 행렬이고, 이 것의 역행렬을 구하는 연산은 $O(N^3)$ 이므로 상당히 많은 시간이 걸리기 때문에 비효율적이다.

second-order Taylor expansion:

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

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

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

이를 개선하는 방법으로는 2가지가 제안된다.

1. Quasi-Newton Method (BFGS)

Hessian의 역을 구하는 것이 아니라, 대략적으로 그 값을 구함으로서 시간복잡도를 $O(N^2)$ 까지 줄인다고는 하는데, 어차피 큰 데이터를 메모리에 저장해야하기 때문에 자주 쓰이지는 않는다고 한다.

2. L-BFGS (Limited memory BFGS)

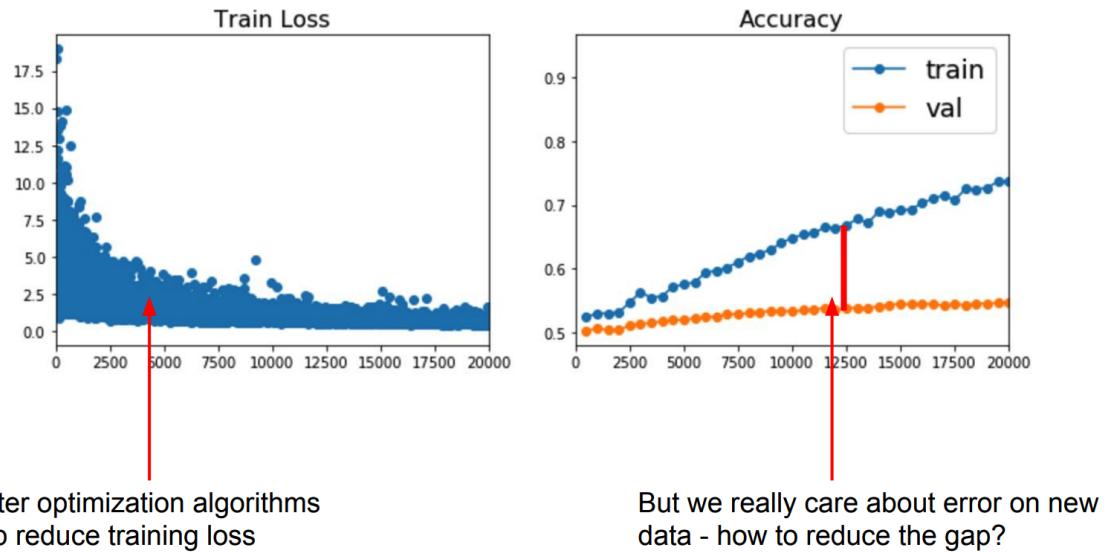
Hessian의 역행렬의 전체를 메모리에 저장하지 않는 방법이다.

그래도 상당히 복잡하기 때문에 random 요소나, 노이즈를 최대한 제거하고 사용해야 한다. 주로 full batch상황에서는 잘 동작하지만, mini-batch 상황에서는 좋은 결과를 낳지 못한다고 한다.

정리하자면, 대부분 Adam을 사용하고, 만약 Full-batch update가 가능하고 노이즈를 최대한 제거할 수 있다면, L-BFGS를 사용할 수도 있다.

Ensemble

이때까지 우리는 training data를 이용하여 loss를 줄여나가고, training data에 대한 정확도를 늘리는 작업들을 행해왔다. 하지만 실제로 우리는 training data의 정확도는 별로 중요하게 여기지 않는다. 새로 들어오는, 즉 unseen data에서 우리의 Machine이 얼마나 높은 정확도를 보이는지가 중요하다. 아래의 그림처럼 train data와 validation data의 정확도 간에는 gap이 발생하는데, 어떻게 하면 이 gap을 줄일 수 있을까?



간단한 방법 중 하나가 ensemble이다.

이는, 여러 개의 독립된 모델들을 학습시켜 결과를 얻은 후 각각에 대해 평균을 내서 test time에서 사용하는 방법이다. 이렇게 하면 2% 정도의 정확도가 증가하게 되는 장점이 있다.

Ensemble을 할 때 사용하는 여러 트릭들이 있다.

우선, 여러 개의 독립된 모델에 대해서 학습하지 않고, 하나의 모델을 학습시키는 과정에서 중간중간의 결과를 저장해두고 이를 이용하여 ensemble을 사용하는 것도 좋은 결과를 보이곤 한다.

또, 결과 뿐 아니라, parameter vector에 대해서도 ensemble을 적용하면, 정확도를 향상시킬 수 있다고 한다.

```

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

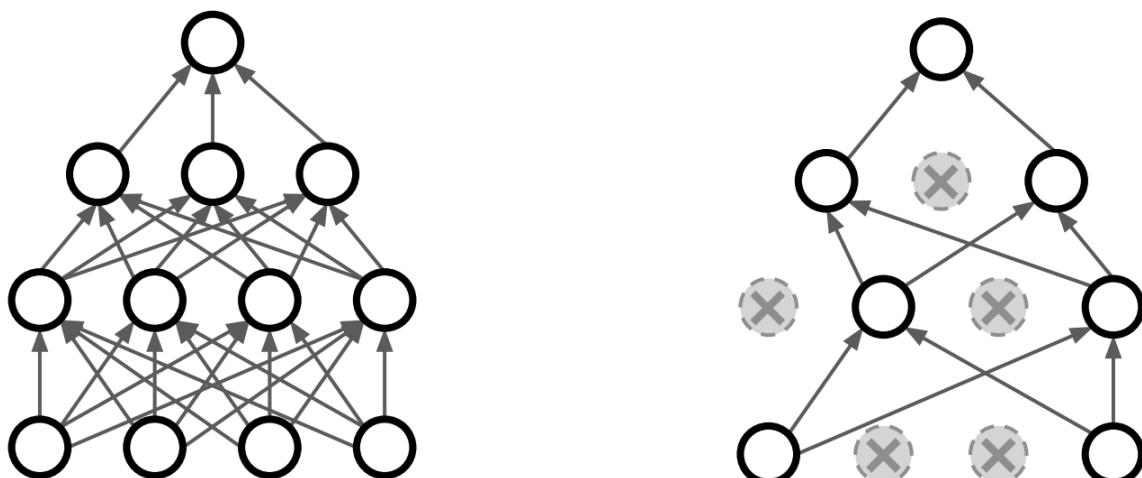
```

Regularization

Ensemble은 한 번에 여러개의 모델에 대해 학습하고 그 결과를 이용해야한다. 이렇게 하지 말고, 하나의 모델을 이용하여 성능을 향상하는 방법에는 어떤 것이 있을까?

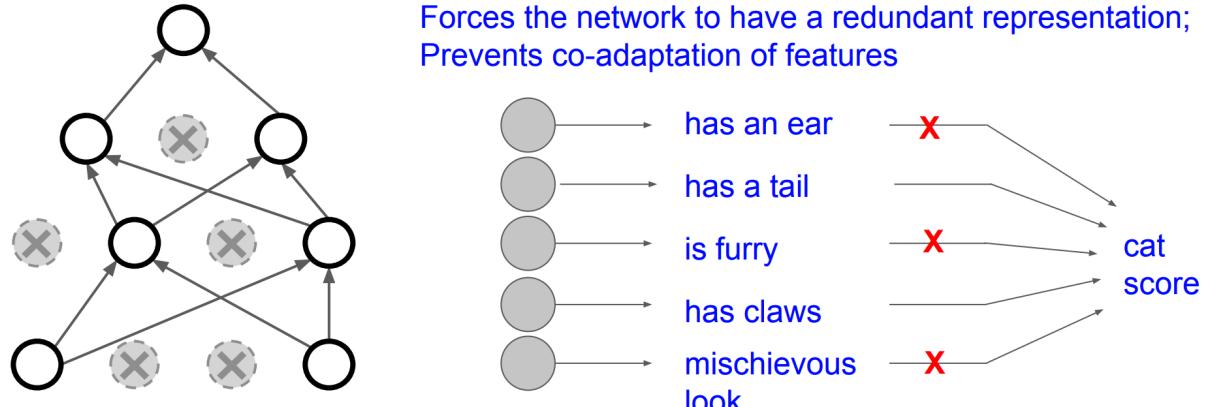
Dropout

많이 쓰이는 것 중 하나는 바로 dropout이다. 학습시키는 과정에서 무작위로 일부 뉴런들을 0으로 만들어서 사용하지 않는 방법이다. dropping하는 비율도 hyperparameter이며, 주로 0.5를 사용한다.



그렇다면 이 방법이 왜 성능 향상에 도움이 될까?

첫 번째 해석은 뉴런들이 자신이 담당하는 특징 뿐 아니라 다른 특징들에 대해서도 학습할 수 있다는 것이다. 아래의 그림에서처럼 귀랑 꼬리에 대한 뉴런이 각각 있는데, 귀에 대한 뉴런을 사용하지 않게 되면 꼬리에 대한 정보를 학습하는 뉴런이 꼬리 뿐 아니라 귀에 대해서도 어느정도 학습할 수 있다고 한다.



또 다른 해석 방법은 이 또한 양상불이라고 한다. 매 학습마다 사용되는 뉴런이 다른데, 그 것들을 다 다른 모델으로 생각한다면 여러개의 모델에 대해 학습하고 그 결과를 합치는 양상불과 같은 효과를 낼 수 있다고 한다.

Dropout을 이용하여 학습했다면 test하는 과정에서는 어떻게 해야하는지 살펴보자.

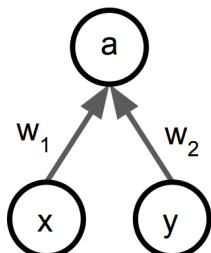
이를 그대로 test에도 적용하려면, 각각의 랜덤 상황에 대해 결과를 다 구하고, 그것들을 조합하여 최종 결과를 내야하는데 이는 상당히 까다롭다. 아래의 식을 살펴보면 각각의 랜덤인 상황을 다 구하고 평균을 내는 것이나, 그냥 random이 없다고 생각하고 모든 neuron들에 대해 연산하는 것이나 결과가 동일하다.

결과적으로, test할 때에는 Dropout 없이, 학습된 모든 뉴런들을 사용하면 된다.

Want to approximate the integral

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

Consider a single neuron.



At test time we have: $E[a] = w_1x + w_2y$

$$\begin{aligned} \text{During training we have: } E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

At test time, multiply by dropout probability

```

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!

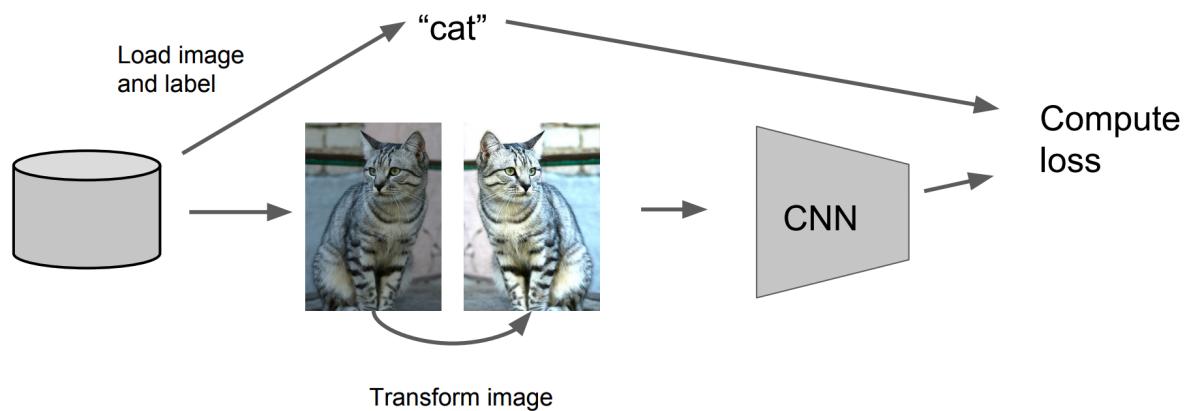
dropout뿐 아니라 batch normalization도 좋은 regularization중 하나이다. 매 train 때 각각의 mini-batch로 데이터가 샘플링될 때, 다른 데이터들과 만나게 된다. dropout과 유사하게 test 할 때에는 mini batch가 아닌 전체를 사용한다. 이 또한 하나의 regularization이다.

주로 Batch Normalization을 사용하지만 dropout도 나쁜 방법은 아니라고 한다.

Data Augmentation

또다른 방법은 data augmentation이다. 기존의 데이터가 하나 존재할 때, 그 데이터를 약간 변형시키더라도 데이터에 대한 label은 변하지 않는다. 예를 들어 고양이 사진이 있을 때, 그 사진을 좌우 반전하더라도 그것은 그저 고양이일 뿐이다. 하지만 컴퓨터가 바라보기에는 다른 데이터 셋이 된다.

Data augmentation은 학습시키는 과정에서 우리가 가지고 있는 training data를 그대로 학습시키는 것이 아니라, 무작위로 이를 변형한 후 학습시키는 것이다. 좌우반전 시킬 수도 있고, 일부 영역만을 잘라서 학습시킬 수도 있다. 색을 약간 변경시키는 color jitter 도 있고, 조금 복잡하게 color offset을 변경하는 방법도 있다고 하는데 이는 자주 쓰이는 방법은 아니라고 한다.



이렇게 우리는 regularization으로 dropout, batch-normalization, data augmentation에 대해 알아 보았다. 그 외에도 여러가지 regularization이 있다.

dropconnect 라고해서, dropout과는 조금 다르게 임의의 weight matrix를 0으로 만드는 것이다.

또, fractional max pooling이라고 해서 max pooling과 유사한데, 임의의 칸을 골라 max pooling을 한다고 생각하면 된다.

또, Stochastic Depth라고 있다. 이는 임의로 layer를 drop시키는 방법이다. 이렇게 하면 dropout과 성능이 유사하다고 한다.

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

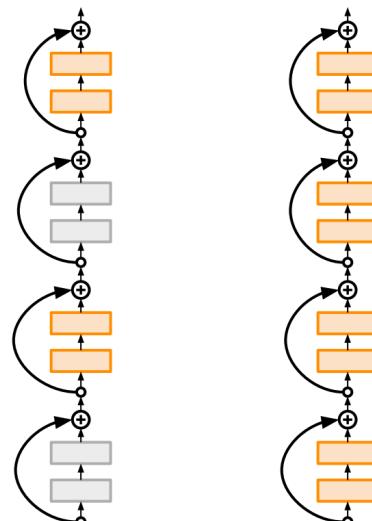
Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth



이처럼 여러 regularization이 있는데, 이 중 하나만을 사용한다고 한다. 주로 Batch Normalization이 사용된다.

Transfer Learning

이는 CNN을 학습하는데에 많은 데이터가 필요하다는 상식을 깨부순 그런 방법이다.

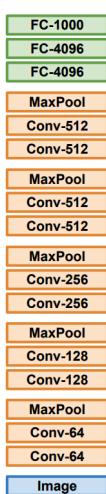
Transfer Learning

“You need a lot of data if you want to train/use CNNs”

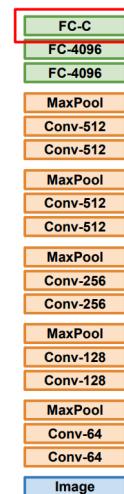
BUSTED

우선 잘 학습된 Imagenet이 하나 있다고 하자. 그리고 우리가 적은 데이터를 가지고 학습을 시도하고 있다고 하자. 만약 100 label 뿐이라면 마지막 layer를 FC-100으로 하고 그부분만 사용하여 학습을 한다는 것이다. 만약 조금 더 많으느 데이터가 있다면 추가적인 layer를 사용하여 학습을 할 수 있다.

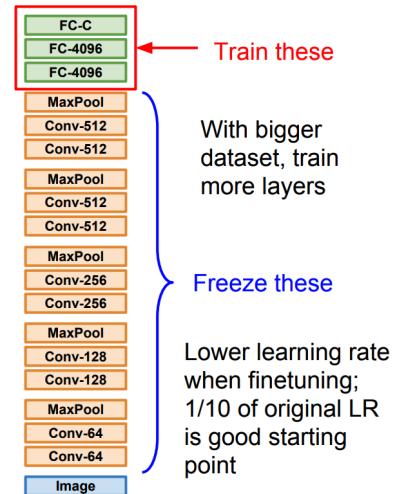
1. Train on Imagenet



2. Small Dataset (C classes)

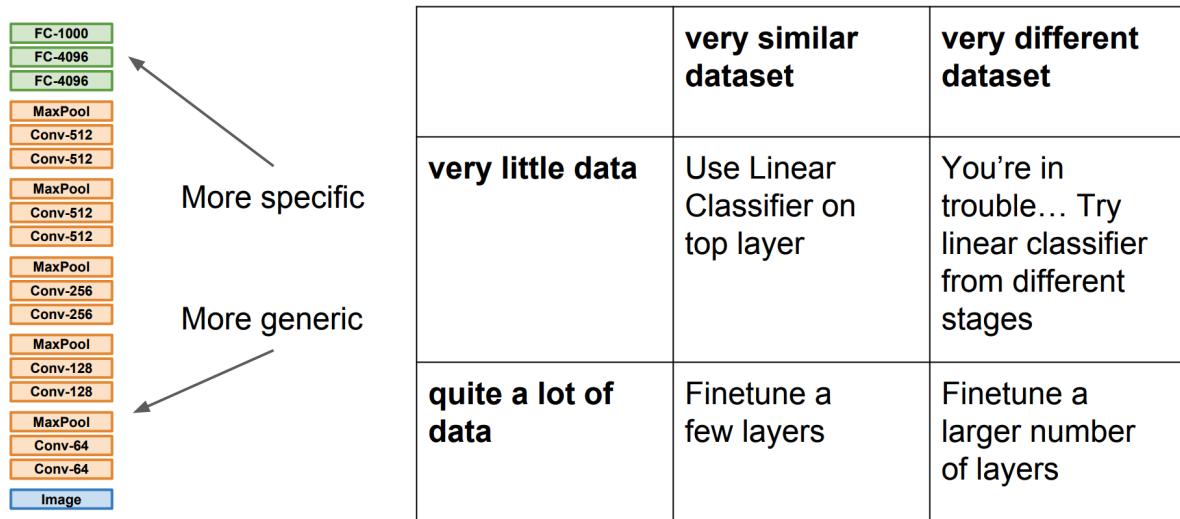


3. Bigger dataset



우리가 고려해야 할 사항은 데이터가 적은지, 많은지와 가지고 있는 데이터셋이 현재 우리가 학습시킬 것과 유사한지, 다른지이다.

유사한 데이터가 있다면 위에서 설명한 방법을 이용하여, 상위 몇개의 layer만으로 학습시킬 수 있다. 데이터가 조금 많다면 layer를 개수를 늘려주면 된다. 만약 데이터의 개수가 적은데 비슷한 데이터셋이 없다면, 여기서는 다른 방법을 시도해보아야 할 것이다. 데이터의 개수가 많다면 여러 layer를 Finetune하여 사용하는 것이 가능하다고 한다.



최근에는 이런 trasnfer learning을 상당히 많이 사용한다. 가장 밑바닥의 layer부터 training하기 보다는 다른 정보를 들고와서 중간의 layer부터 우리의 목적에 맞게 Finetune하여 사용하는 경우가 대다수라고 한다.