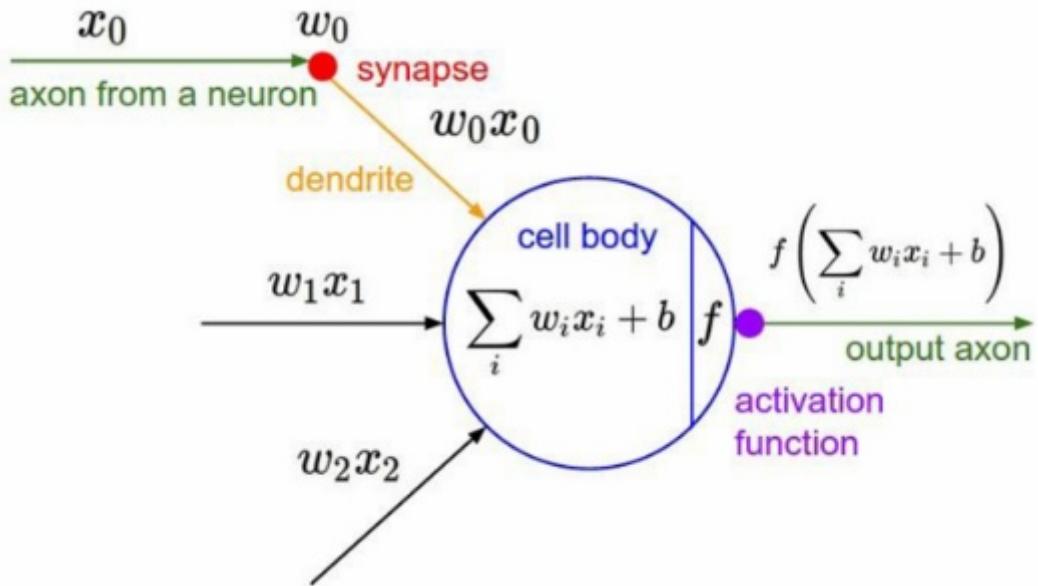


# Ch 6

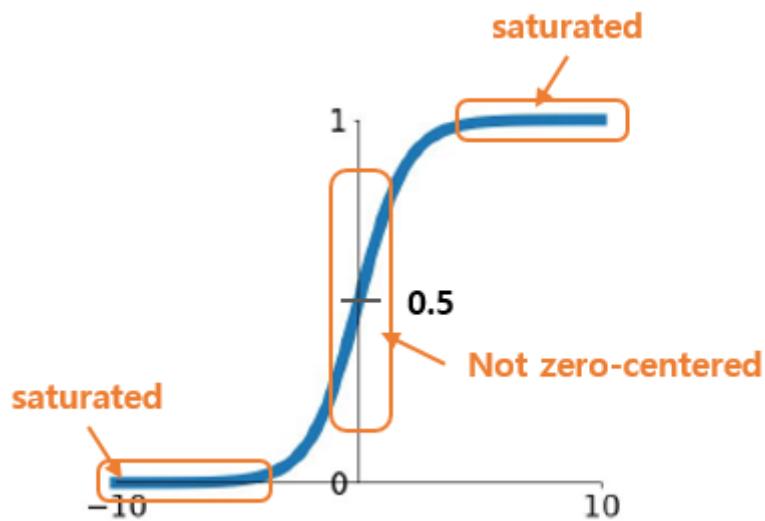
## Activation Functions : 활성 함수 선택



데이터가 들어오면 가중치 연산 후 활성 함수를 거치게 됨

앞에서 배운 Sigmoid 함수 포함 6가지의 활성 함수 소개 → 어떤 함수를 써야 할까

### Sigmoid



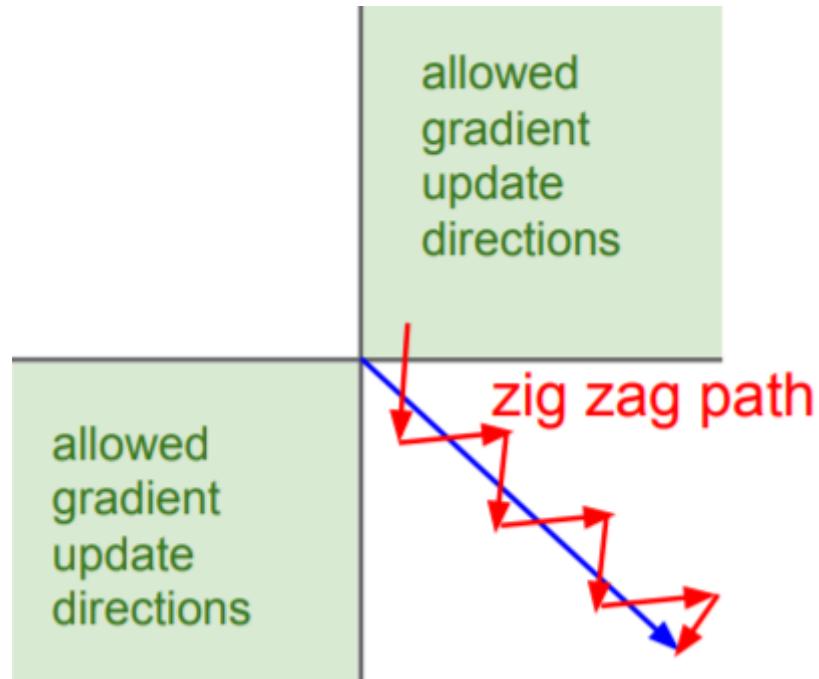
입력 신호의 총합을 0에서 1 사이의 숫자로 수축

입력 신호의 값이 극단적일 경우 뉴런의 활성화율(firing rate)이 수렴(saturate)한다.

- Large Positive  $\approx 1$
- Very Negative  $\approx 0$

3가지 문제점

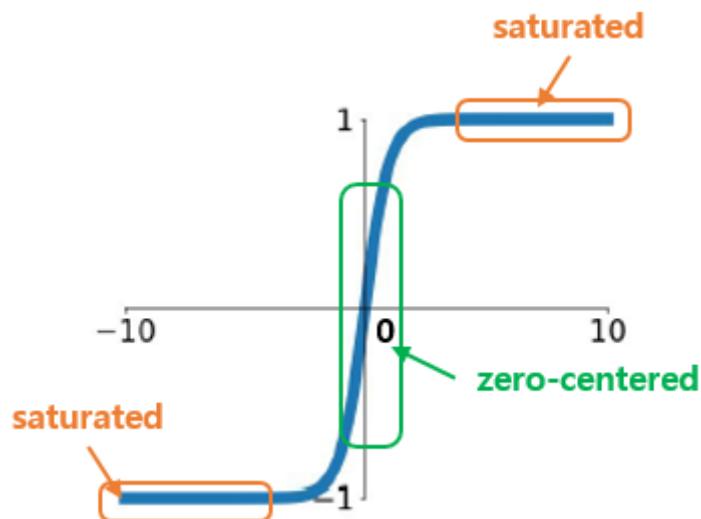
- 0 또는 1로 수렴된 뉴런들이 Gradient를 소멸시킨다.  
역전파 과정에서 0이 곱해지고 이로 인해 아래 층에는 아무것도 전달되지 않음
- 출력값이 원점 중심이 아니다. (Not zero-centered)  
 $dF/dw = X$  : Gradient의 부호가 입력 받은 gradient의 부호와 계속 동일  
→  $w$ 의 Gradient가 같은 방향으로만 움직임 ⇒ 비효율적



$w$ 가 2차원인 예시  
⇒ 1, 3 사분면의 방향으로만 update 가능

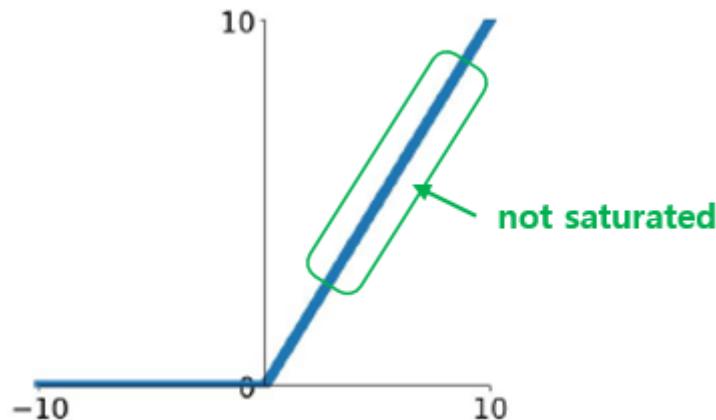
- 지수함수가 포함되어 계산이 복잡하다. (minor problem)

## tanh (hyperbolic tangent function)



입력 신호의 총합을 -1과 1 사이의 숫자로 수축  
원점 중심이다(zero-centered) ⇒ Sigmoid 함수와 가장 큰 차이점  
여전히 Gradient를 소멸시킨다. (saturated regime)

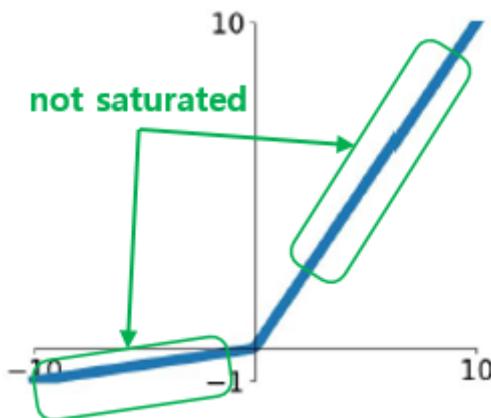
## ReLU Function (Rectified Linear Unit)



$$f(x) = \max(0, x)$$

- 음수일 경우 0 출력, 양수일 경우 입력값 그대로 출력
  - 적어도 입력 공간의 절반은 Saturation 발생 X
  - max 함수  $\Rightarrow$  계산 효율적
  - 수렴속도가 sigmoid와 tanh에 비해 매우 빠름 (Stochastic Gradient Descent의 경우 약 6배)
  - 생물학적 타당성도 sigmoid보다 나음 (firing rate 연상)
  - tanh와 달리 원점 중심이 아님
  - 가중치 합이 음수가 되는 순간 0만 출력  $\Rightarrow$  gradient = 0 (dead ReLU)
  - Dead ReLU
1. Initialization 잘못한 경우  
 $\Rightarrow$  Never activate / Never update / Back Propagation X
  2. Learning rate가 지나치게 높은 경우  
 $\rightarrow$  updated를 큰 범위로  $\rightarrow$  가중치 날뜀  
 $\rightarrow$  학습 과정에서 ReLU가 데이터의 manifold를 벗어나게 됨  
 $\rightarrow$  처음엔 학습이 잘 되다가 갑자기 죽어버림  
실제로 학습이 끝난 네트워크의 10~20%가량이 dead ReLU가 됨

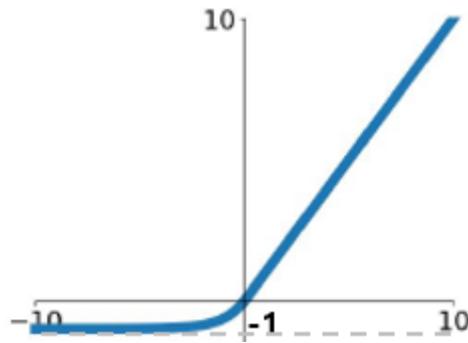
## Leaky ReLU



$$f(x) = \max(\alpha x, x)$$

- 일반적으로  $\alpha = 0.01$  (fixed)
- Dead ReLU 해결 (does not saturate)
- PReLU (Parametric ReLU)
  - : Leaky ReLU와 식 동일
  - : PReLU에서는  $\alpha$ 의 값도 학습되도록 하여 역전파에 의해  $\alpha$ 의 값이 변경된다.
  - : 소규모 데이터셋에서는 과적합될 위험 존재

## ELU



$$\text{ELU}_\alpha = \begin{cases} \alpha (\exp(x) - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

- Zero-mean에 가까운 출력을 보임
- 음의 Saturation regime  
⇒ 이런 deactivation이 ELU가 Leaky ReLU보다 noise에 강하게 만듦
- $\alpha$  :  $x$ 가 음수일 때 ELU가 수렴할 값을 정해준다. (보통  $\alpha=1$ )

## Maxout Neuron

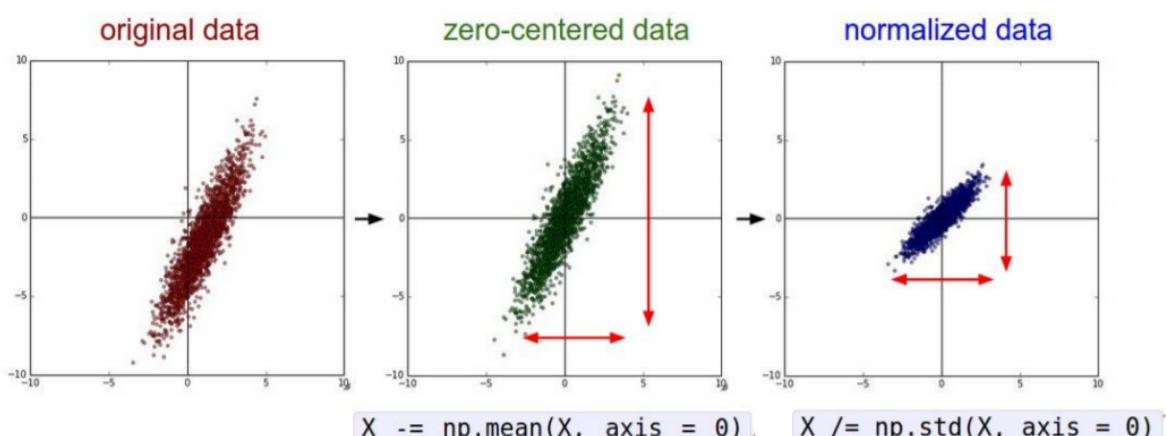
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

- ReLU와 Leaky ReLU의 일반화된 형태  
: 두 개의 선형 함수를 취함 (linear regime)
- Does not Saturate ⇒ Does not Die
- 문제점: 뉴런 당 모수의 개수 2배 ( $w_1$  and  $w_2$ )

## 결론

: learning rate에 주의하여 ReLU를 먼저 적용해보고  
→ Leaky ReLU / ELU와 같은 ReLU family 혹은 Maxout 시도  
(→ tanh 시도, sigmoid는 일반적으로 사용하지 않음)

## Data Preprocessing : 데이터 전처리



- 일반적으로, zero-mean으로 만들고 정규화 진행

- if not zero-mean  
⇒ Suboptimal한 최적화를 하게 됨
- Normalization 목적  
: 모든 feature가 동등하게 contribute하도록
- 실제로 이미지 데이터에선 Zero-centering만 진행  
: 이미 각 차원 간에 스케일이 어느정도 맞춰져 있기 때문
- 평균 값을 빼서 Zero-centering
  1. Image value 전체의 평균을 빼는 경우  
: e.g. subtract [32, 32, 3] array ⇒ AlexNet
  2. Channel 별 평균을 빼는 경우  
: e.g. 3개의 평균값 (RGB) ⇒ VGGNet (더 편리)
- Test Set에서도 Train Set으로 계산한 평균값을 뺀다.  
: mini-batch가 아닌 전체 Train data로 계산한 평균  
(이상적으로 전체 Train의 평균과 batch의 평균이 같은 함)
- Zero-centering으로 Sigmoid 함수의 non-zero mean 문제를 해결할 수 있을까?  
: No. 첫번째 layer에서만 해결되고 다음 layer부터는 같은 문제가 반복

## Weight Initialization : 가중치 초기화

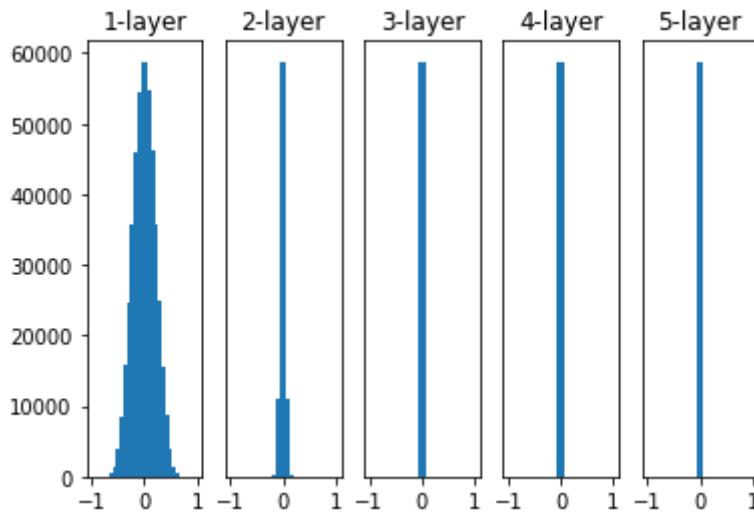
### 1. Weight 초기값이 0인 경우

- 모든 뉴런이 같은 연산을 수행
- 출력도 같고 gradient도 같으므로 모든 가중치가 똑 같은 값으로 update
- 뉴런이 하나뿐인 것처럼 작동 ⇒ 학습이 제대로 이뤄지지 않음

### 2. 작은 난수인 경우 (small random numbers)

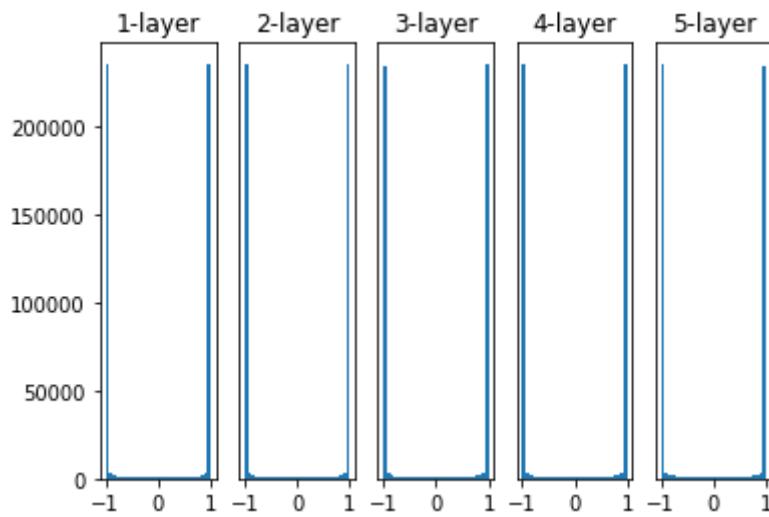
평균이 0이고 표준편차가 0.01인 정규분포를 따르는 값으로 랜덤하게 초기화 (일반적)

- 신경망이 얇은 경우 제대로 작동



- 하지만 신경망이 깊어질수록 문제 발생
- 모든 활성함수 결과가 0이 되고,  
gradient도 매우 작은 값을 형성하여 update가 잘 이뤄지지 않음

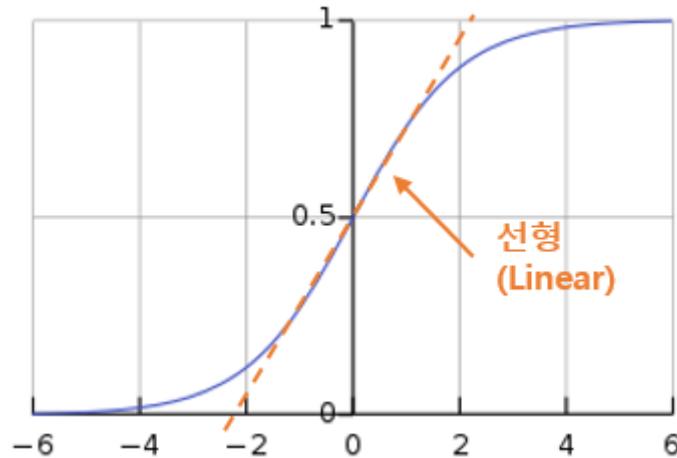
평균이 0이고 표준편차가 1인 정규분포를 따르는 값으로 초기화할 경우



- $\tanh(x)$ 의 출력이 saturated (-1 또는 1의 출력값을 가짐)
- gradient가 0이 되어 가중치가 update되지 않음

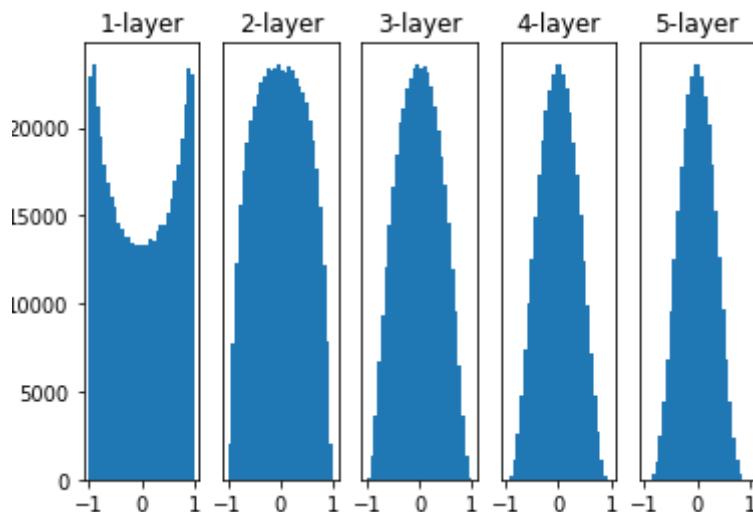
### 3. Xavier Initialization

- 입/출력 값의 분산을 맞춰준다.  
( $\Rightarrow$  즉, Back Propagation 과정에서 layer 통과 전/후의 Gradient 분산이 동일해야 함)
- 활성함수가 선형이라고 가정



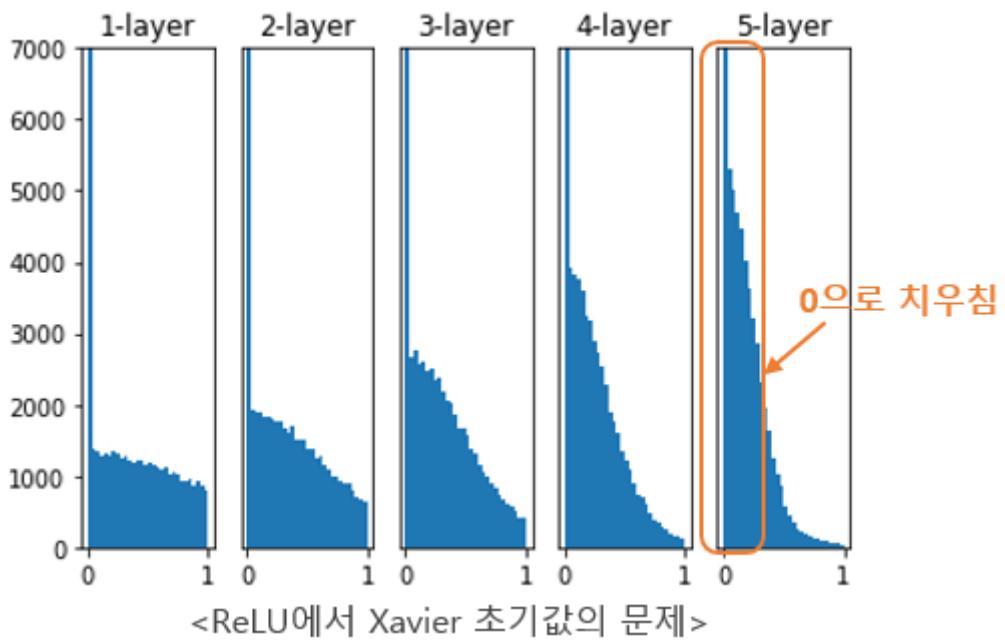
- $W$ : 표준 정규분포에서 뽑은 난수를 '입력의 수'로 스케일링해준 값을 초기값으로

image.png

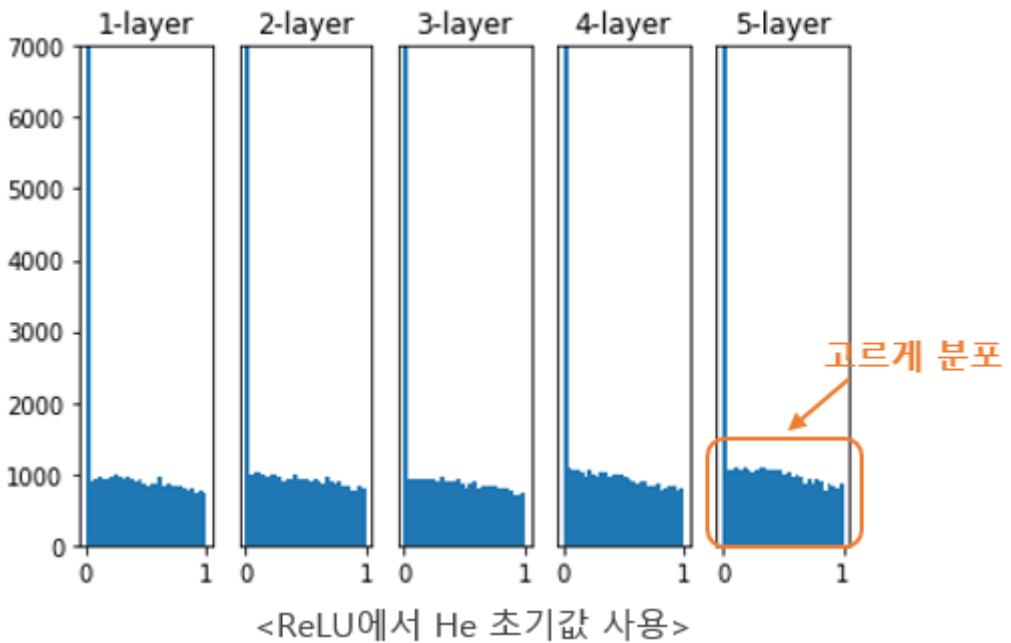


- $\tanh$  함수의 active region 안에 있다고 가정 (선형 가정)
- $\approx$  표준 정규 분포

## ReLU + Xavier Initialization



## He Initialization



- 간단히 말하면 Xavier 초기값에  $\sqrt{2}$  를 곱한 값

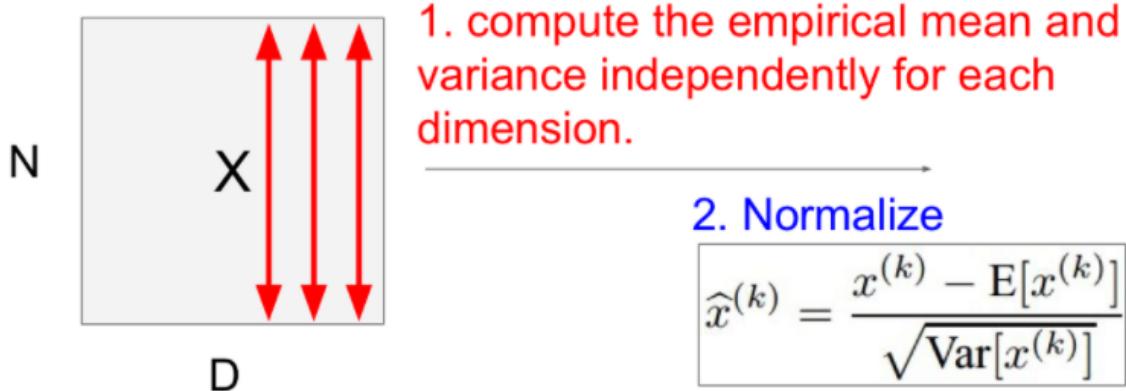


# Batch Normalization

Batch Normalization은 어떤 레이어로부터 나온 Batch 단위만큼의 activation이 있다고 했을 때, 이 값들이 Unit gaussian이기를 원한다는 아이디어에서 시작 되었습니다.

요약하면 train을 시작할 때 network가 unit gaussian distribution을 취할 수 있도록 만들어 주는 방법입니다.

## Batch Normalization의 과정

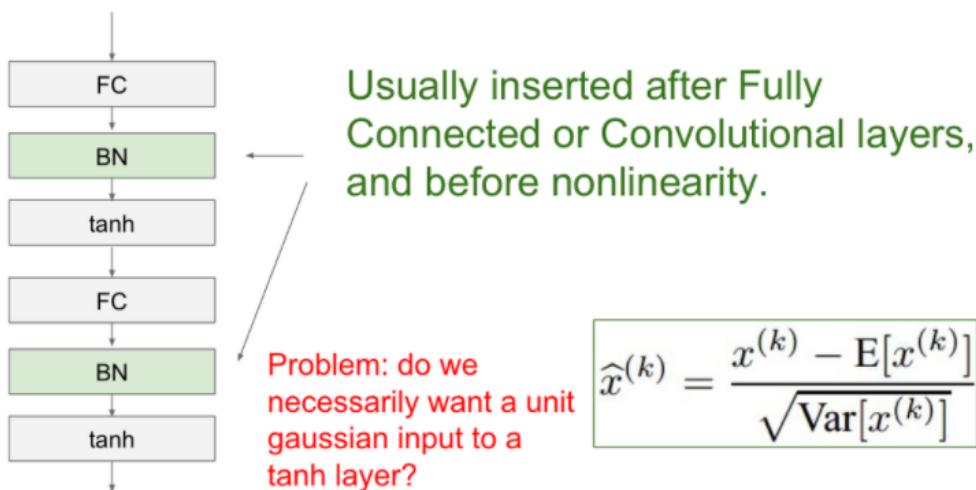


Batch 단위로 한 레이어에 입력으로 들어오는 모든 값을 이용해서 평균과 분산을 구하고, 그 값을 이용해 Normalization 해주면 됩니다

## Batch Normalization의 위치

### Batch Normalization

[Ioffe and Szegedy, 2015]



Batch-Norm은 입력의 스케일만 살짝 조정해주는 역할이기 때문에 FC(Fully-Connected Layer)와 Conv(Convolution Layer) 어디에든 적용할 수 있는데, Batch Norm 연산은 FC나 Conv 직후에 넣어줍니다.

그런데 한 가지 문제가 있는데 이렇게 Layer를 거칠 때마다 매번 normalization을 해주는 것이 맞는지에 대한 의문이 들 수 있습니다.

Batch Normalization에서는 normalization 연산이 있습니다. normalization 연산 이후에 Batch-Norm에서는 scaling 연산을 추가합니다.

그래서 Batch Normalization에서는 이런 판단도 학습에 의해 가능하다고 합니다.

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathbb{E}[x^{(k)}]$$

to recover the identity mapping.

여기서 감마는 scaling parameter, 베타는 shifting parameter로 학습가능한 parameter입니다. 학습의 예시로 normalize 한 값을 원래대로 복구하고 싶으면 감마에 분산 값을, 베타에 평균값을 넣어주면 됩니다.

최종적으로 Batch Normalization의 과정을 보면 아래와 같습니다.

## Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;  
Parameters to be learned:  $\gamma, \beta$   
**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Batch Normalization의 효과로는 gradient의 흐름을 원활하게 해 주어 결국 학습이 더 잘되게 만들어 준다는 것입니다.

또 하나는 각 레이어의 출력이 해당 데이터 하나 뿐만 아니라 batch안에 존재하는 모든 데이터들의 영향을 받기 때문에 regularization 역할도 한다는 것입니다.

마지막으로 test에서 Batch Normalization 인데 Test 할 때에는 전체의 mean/var 기준으로 구합니다.

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;  
Parameters to be learned:  $\gamma, \beta$   
**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

**Note: at test time BatchNorm layer functions differently:**

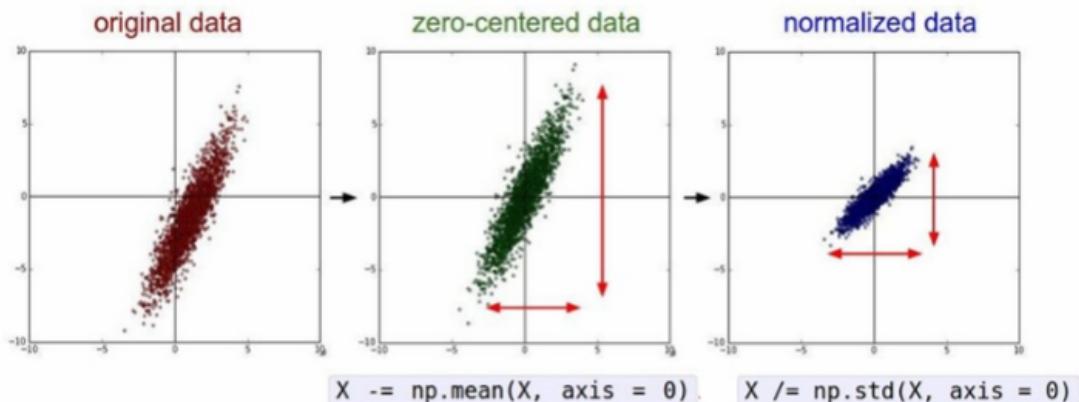
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

# Babysitting the Learning Process

다음으로 학습 과정에서 체크해야 하는 것입니다.

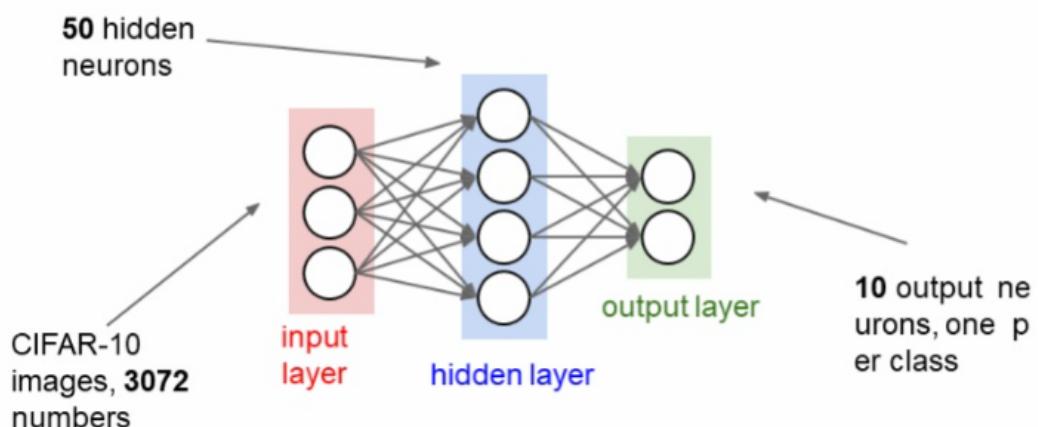
## Step 1: Preprocess the data



(Assume  $X$  [ $N \times D$ ] is data matrix,  
each example in a row)

첫번째는 전처리 과정입니다. 앞에서 말한 대로 보통 zero-centered로 바꿔줍니다.

## Step 2: Choose the architecture: say we start with one hidden layer of 50 neurons:



두 번째는 아키텍처 선택입니다. Hidden layer를 어떻게 구성할 것인지 등을 선택하는데 보통 대충의 사이즈를 설정합니다.

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0) # disable regularization
print loss
2.30261216167
loss ~2.3. " correct " for 10 classes
returns the loss and the gradient for all parameters
```

세번째로 레이어 구상이 끝났으면 loss값이 잘 나오는지 확인합니다.

3장에서 언급되었던 sanity check를 통해 loss 값이 잘 설정 되었는지 확인할 수 있습니다.

Lets try to train now...

**Tip:** Make sure that you can overfit very small portion of the training data

Very small loss, train accuracy 1.00, nice!

```

model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)

Finished epoch 1 / 200: cost: 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost: 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost: 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost: 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost: 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost: 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost: 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost: 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost: 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost: 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost: 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost: 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost: 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost: 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost: 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost: 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost: 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost: 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost: 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03

Finished epoch 195 / 200: cost: 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost: 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost: 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost: 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost: 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization, best validation accuracy: 1.000000, val 1.000000, lr 1.000000e-03

```

네번째로 train을 시켜봅니다.

그런데 바로 모든 데이터를 넣는 것이 아니라 작은 데이터 셋을 먼저 넣습니다.

이 경우에는 데이터 수가 작기 때문에 overfitting되어 train accuracy가 100%가 나오게 되는데 overfitting이 되면 모델이 잘 작동하고 있는 것을 확인할 수 있습니다.

마지막으로 regularization 값과 learning rate 같은 적절한 값을 넣어보면서 찾으면 됩니다.

# Hyperparameter Optimization

learning rate와 regularization 같은 parameter들을 어떻게 정하는지 더 자세히 보겠습니다.

먼저 소개할 방법은 값을 넓은 범위에서 좁은 범위로 좁혀나가는 방법인데 첫번째 단계에서는 어느 정도 hyperparameter에서 정상적으로 작동하는지 범위를 구합니다.

그리고 두번째 단계에서는 좀더 긴 training time에서 범위안의 hyperparameter중 어느 것이 좋은지 골라냅니다.

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6) ←
        trainer = ClassifierTrainer()
        model = init two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
        trainer = ClassifierTrainer()
        best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                                model, two_layer_net,
                                                num_epochs=5, reg=reg,
                                                update='momentum', learning_rate_decay=0.9,
                                                sample_batches = True, batch_size = 100,
                                                learning_rate=lr, verbose=False)

        val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
        val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
        val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
        val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
        val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
        val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
        val_acc: 0.441000, lr: 1.750259e-04, reg: 2.1108007e-04, (7 / 100)
        val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
        val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
        val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
        val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

note it's best to optimize  
in log space!

← nice

예시에서 위에서와 같이 random한 값을 이용했는데 lr이 1e-4에서 괜찮은 valid accuracy를 가진다는 것을 알 수 있습니다.

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6) ←
        adjustrange →
        max_count = 100
        for count in xrange(max_count):
            reg = 10**uniform(-4, 0)
            lr = 10**uniform(-3, -4)

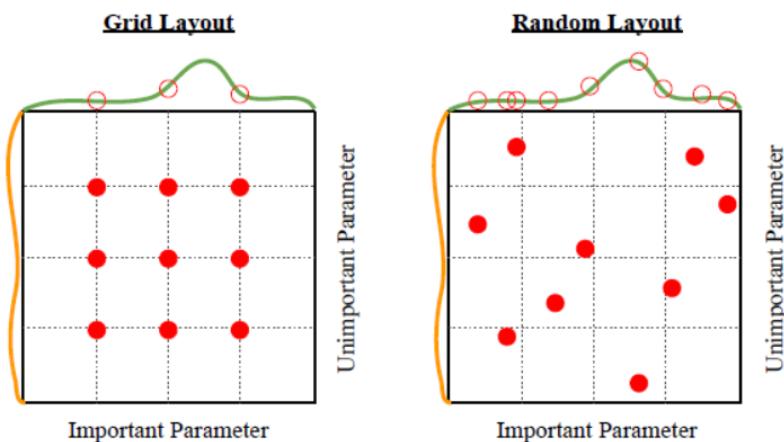
            val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
            val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
            val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
            val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
            val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
            val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
            val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
            val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
            val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
            val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
            val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
            val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
            val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
            val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562888e-02, (13 / 100)
            val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
            val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
            val_acc: 0.514000, lr: 6.438349e-04, reg: 3.633781e-01, (16 / 100)
            val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
            val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
            val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
            val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
            val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good f  
or a 2-layer neural net  
with 50 hidden neurons.

따라서 lr에 대해 1e-4, 1e-5 사이에서 finer search를 진행합니다. 근데 지금 보면 9.47 -04까지도 높은 accuracy를 보여주는데 1e-5에서도 높은 accuracy가 나올 가능성이 있는데 이런 경우는 다시 범위를 넓혀서 search를 다시 해보면 되겠습니다.

# Random Search vs. Grid Search

Random Search for  
Hyper-Parameter Optimization  
Bergstra and Bengio, 2012



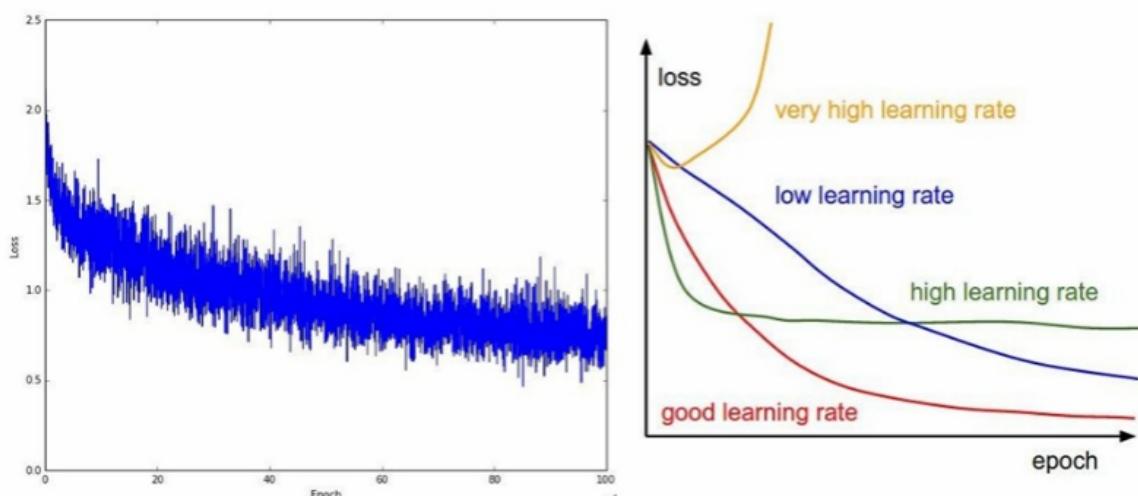
이렇게 찾는 방법에는 grid search와 random search 두가지 방법이 있습니다.  
이 경우에는 grid search가 더 좋은 성능을 내는 parameter 값을 찾지 못할 수 있기 때문에 random search가 더 좋습니다.

Cross-validation  
“command center”



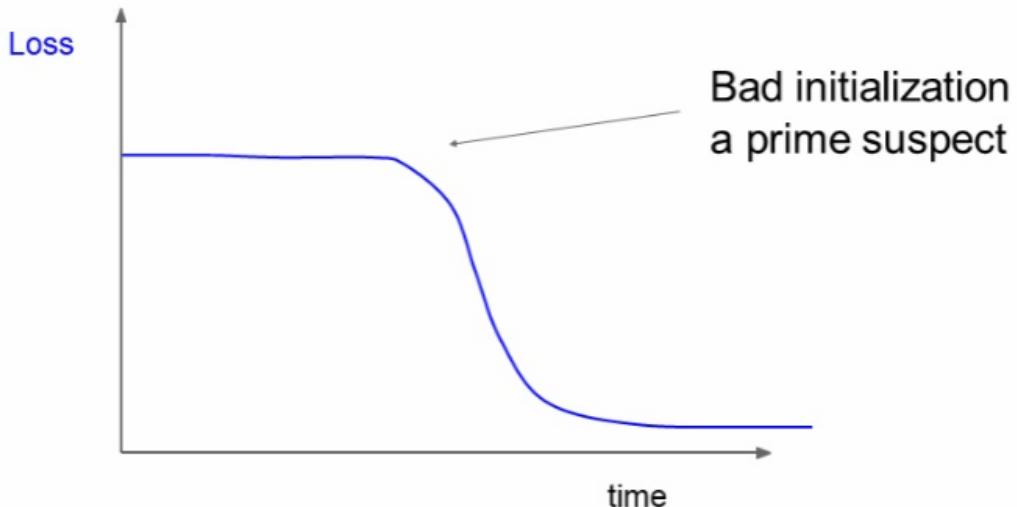
실제로 이렇게 많은 방법으로 최적의 값을 찾는다고 합니다.  
이처럼 시행착오를 통해 최적의 파라미터 값을 찾아가면 되겠습니다.

## Monitor and visualize the loss curve



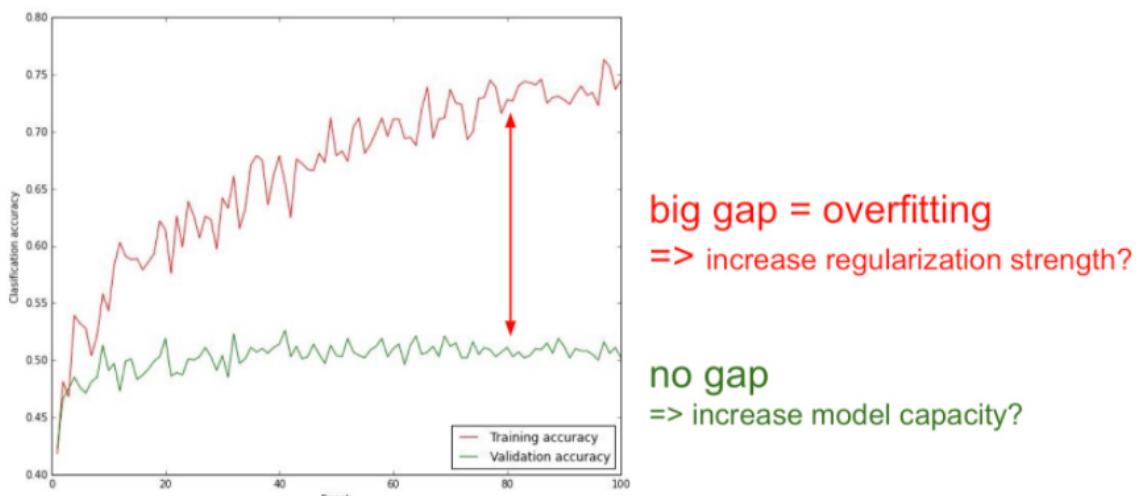
그리고 loss curve를 보고도 learning rate가 적당한지 판단 할 수 있습니다.

노란색 선은 learning rate가 너무 높아 발산할 때, 초록색 선은 learning rate가 '조금' 높을 때, 파란색 선은 learning rate가 낮을 때입니다. 이 경우들은 loss curve가 빨간색 선의 모양을 보이게 learning rate를 조절 할 필요가 있습니다.



loss curve가 위의 모양을 하고 있는 경우도 있는데, 이 경우는 initialization이 좋지 못해서 gradient의 backpropagation이 초반에는 잘 되지 않다가 학습이 진행되면서 회복된 경우로 해석할 수 있습니다.

### Monitor and visualize the accuracy:



다음은 accuracy curve입니다.

train accuracy와 validation accuracy와의 gap이 크면 overfitting을 의심해 볼 수 있습니다. 따라서 regularization의 값을 높여야 한다고 해석 할 수도 있겠습니다.

train accuracy와 validation accuracy와의 gap이 거의 없으면 아직 overfitting되지 않았다고 해석 할 수 있습니다. 따라서 model의 capacity를 높이기 위해 layer를 추가 할 수도 있겠습니다.

Track the ratio of weight updates / weight magnitudes:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

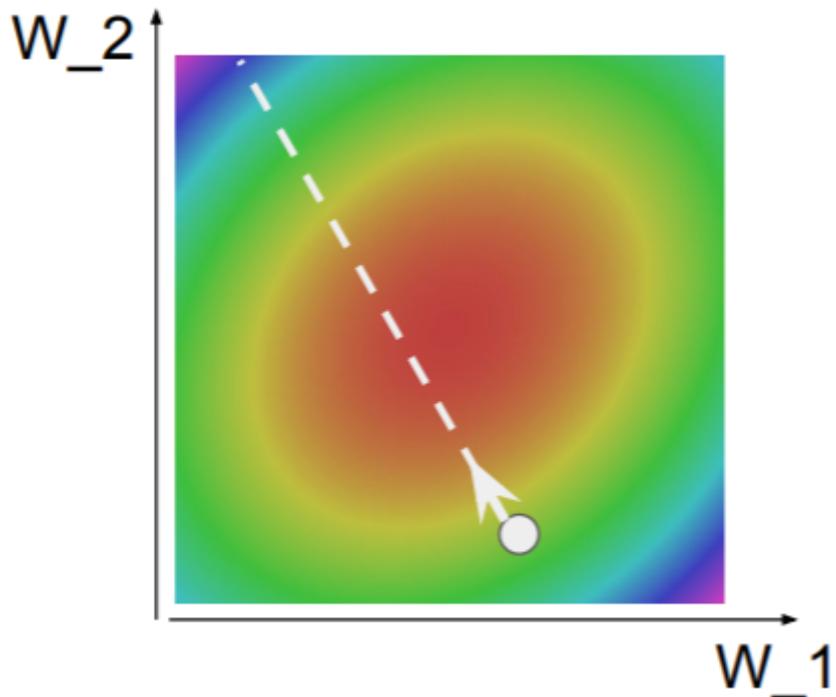
ratio between the updates and values:  $\sim 0.0002 / 0.02 = 0.01$  (about okay)  
**want this to be somewhere around 0.001 or so**

마지막으로 여기서 update/value rate의 비를 통해서도 학습이 진행되는지 판단해 볼 수 있는데 update/value rate가 0.001~이면 learning이 잘 되고 있는 것이라고 합니다.

## Ch 7

### Optimization

- SGD보다 강력한 최적화 알고리즘
- Optimization은 Neural Network에서 가장 핵심인 문제



가중치  $W_1, W_2$ 에 대한 loss function

=> 가장 붉은색인 지점 = 두 개의 가중치에 대해 최적화된(가장 loss가 낮은) 지점

# Stochastic Gradient Descent

```
# Vanilla Gradient Descent

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

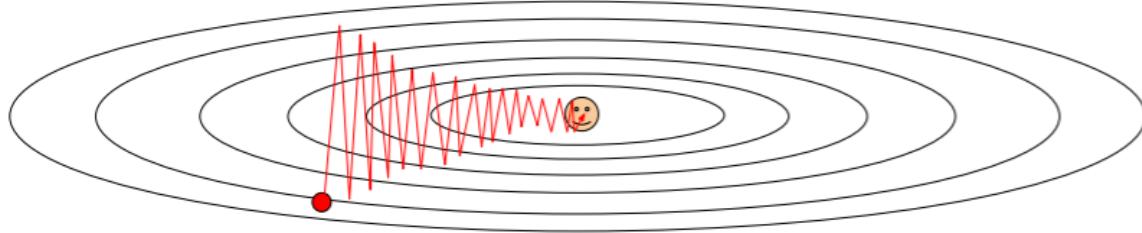
**negative** direction of the gradient(loss가 감소돼야 하기 때문)

parameter update 반복 => 붉은색 지점으로 수렴(loss 최소화)

## SGD의 문제점

i) **condition number**가 높은 경우

\* condition number: ratio of largest to smallest singular value of the Hessian matrix

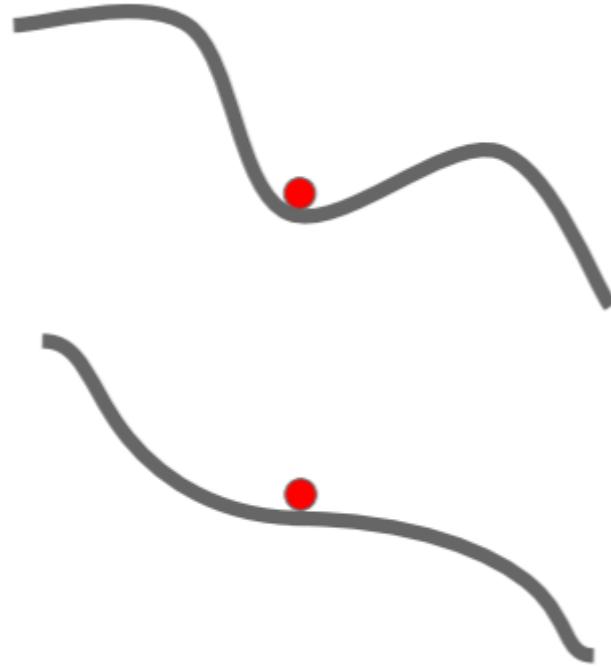


- 수평 축은 weight 변화에 둔감, 수직 축은 weight 변화에 민감

=> 수평/수직 방향 gradient update 속도 차이 때문에 지저분해진다

- 고차원일수록 condition number가 커질 가능성 ↑

ii) **local minima / saddle point**가 있는 경우

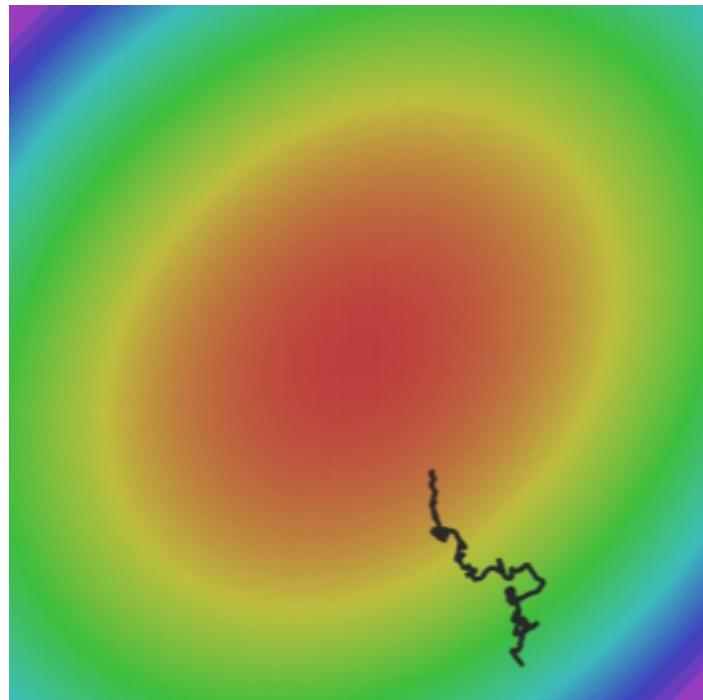


- local minima, saddle point에서는 gradient가 0

=> update 중단

- 고차원일수록 local minima는 찾기 힘들고 거의 모든 지점이 saddle point일 것
- 큰 규모의 neural network는 saddle point에 매우 취약
- saddle point는 근방도 기울기가 작기 때문에 그 근방은 gradient update 속도 ↓

iii) **mini batch**로 loss를 추정



- batch를 training set 전체로 설정하고 loss를 계산할 수는 없기 때문에 실제로는 mini batch 데이터로 loss를 추정한다

=> noise가 들어가서 gradient update 속도 ↓

그럼 full batch GD에서는 문제가 없는가?

=> 아니다

noise는 mini batch라서만 발생하는 건 아니다. explicit stochasticity에 의해서도 발생한다

## 해결책 1. SGD + Momentum

- Momentum term을 추가해주면 대다수의 문제 간단하게 해결 가능

### SGD

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

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

### SGD + Momentum

```

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

```

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

- mini batch의 gradient estimates에 **velocity**를 더해준다

- \* velocity  $\approx$  exponentially weighted moving average
- gradient의 방향이 아닌 velocity의 방향으로 step
- hyperparameter  $\rho$  : friction(momentum의 비율), 보통 0.9 or 0.99

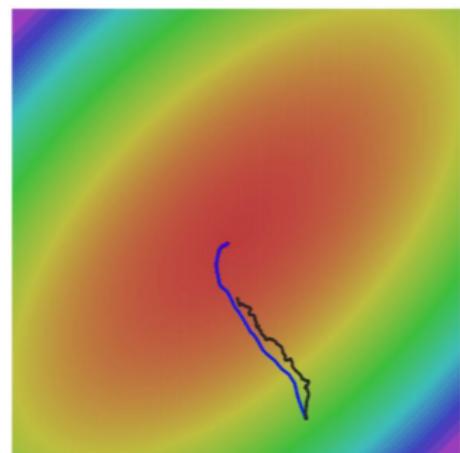
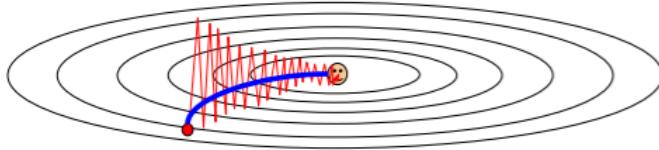
=>  $\rho$  비율만큼의 velocity  $v_t$ 를 gradient  $\nabla f(x_t)$ 에 더해주는 방식

### Gradient Noise

Local Minima      Saddle points



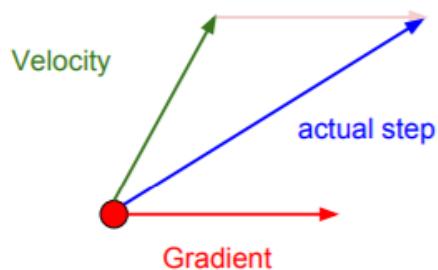
Poor Conditioning



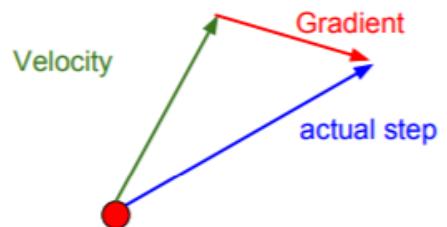
간단하게 해결!

**Nesterov Momentum** - 좀 더 섞어주는 방식, convex optimization에 탁월

Momentum update:



Nesterov Momentum



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

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

$$(\tilde{x}_t = x_t + \rho v_t)$$

↓

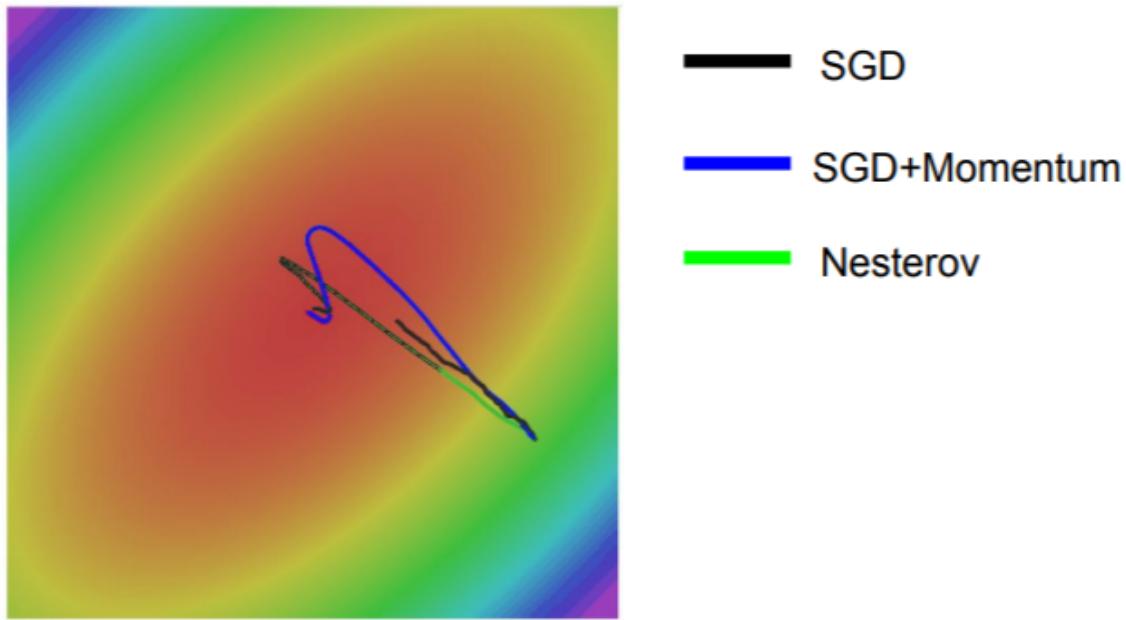
$$\begin{aligned}
 v_{t+1} &= \rho v_t - \alpha \nabla f(\tilde{x}_t) \\
 \tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1} \\
 &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)
 \end{aligned}$$

```

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

```

### SGD / SGD+Momentum / SGD+Nesterov Momentum



- Nesterov가 일반 momentum에 비해 overshooting이 덜하다

## 해결책 2. AdaGrad, RMSProp

- Velocity 대신 **squared gradient**의 합 사용

### AdaGrad

```

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

```

- Update term을 grad\_squared의 식으로 나눈다

=> small dimension에서는 속도 ↑, large dimension에서는 속도 ↓ (condition number problem 해결)

\* 1e-7을 더하는 이유: 분모에 0이 오는 것 방지

- AdaGrad의 문제: step을 진행할수록 step size가 점점 작아진다 (loss function이 convex라면 유용 하나 non-convex라면 곤란)

### RMSProp

- AdaGrad의 문제 개선

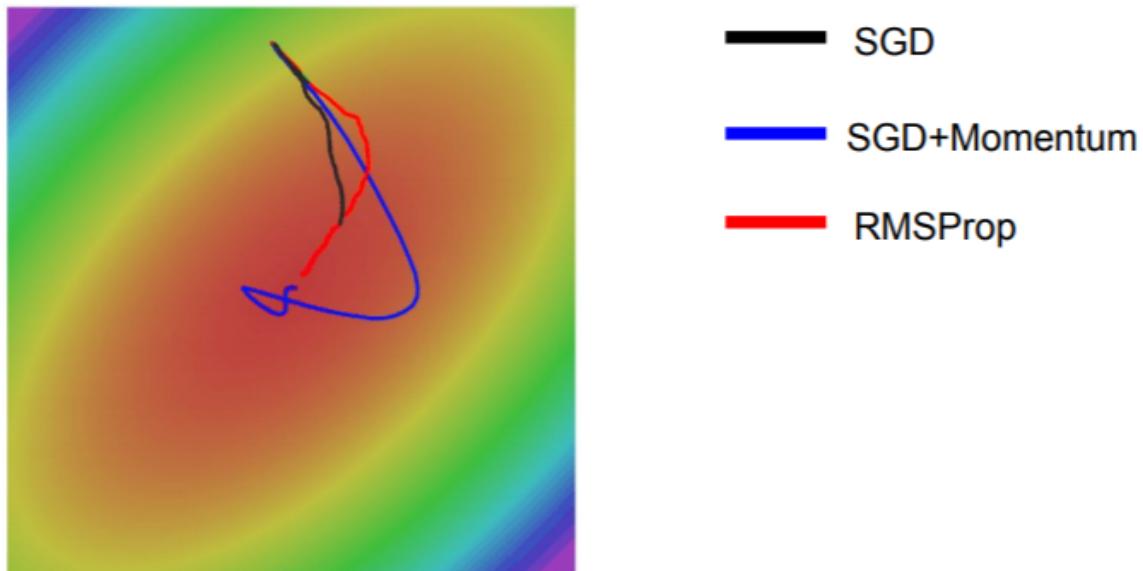
```

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)

```

- 여전히 squared gradient를 사용하지만 단순 누적이 아니라 decay rate 사용
- momentum term과 유사, decay rate는 momentum  $\rho$ 와 마찬가지로 보통 0.9, 0.99 사용
- step 부분은 AdaGrad와 동일
- decay rate 조절로 step size 작아지는 문제 보완 가능

### SGD / SGD+Momentum / RMSProp



- SGD+Momentum, RMSProp 모두 기본 SGD보다 훨씬 낫다
- SGD+Momentum; Overshoot하다가 복귀
- RMSProp; 모든 차원에서 균일하게 진행되도록 규칙 조절
- AdaGrad를 추가한다면 감소하는 learning rate 때문에 RMSProp에 가려 안보일 것

## 해결책3. Adam

- Momentum 계열과 Ada 계열 조합
- First moment, second moment 이용
- 거의 모든 architecture에서 잘 작동

### Adam(Amazing)

```

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

- first moment: gradient의 가중합(Momentum 방식, velocity 담당)
- second moment: squared gradient 사용(AdaGrad, RMSProp 방식)
- 문제: second moment를 0으로 initialize한 데다 update 후에도 second moment decay rate(beta2) 0.9 or 0.99를 사용하기 때문에 여전히 0에 가깝다

=> second moment가 분모로 들어가면 step이 너무 커진다(loss function의 기울기와 별개로)

### Adam(Full Form)

- Bias correction term 추가

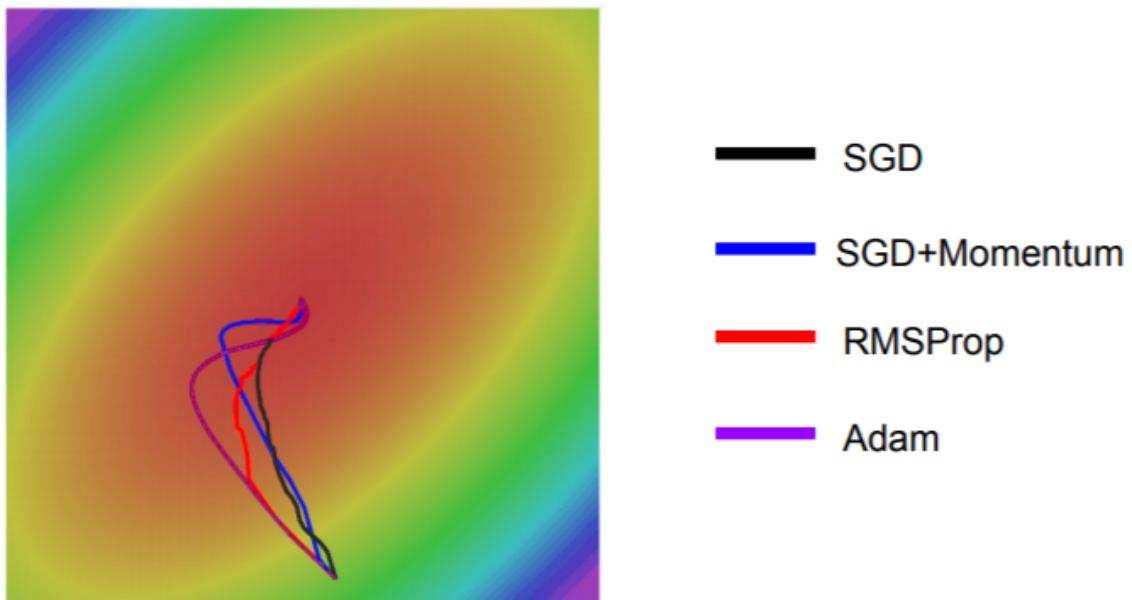
```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment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

Momentum

Bias correction

AdaGrad / RMSProp

SGD / SGD+Momentum / RMSProp / Adam

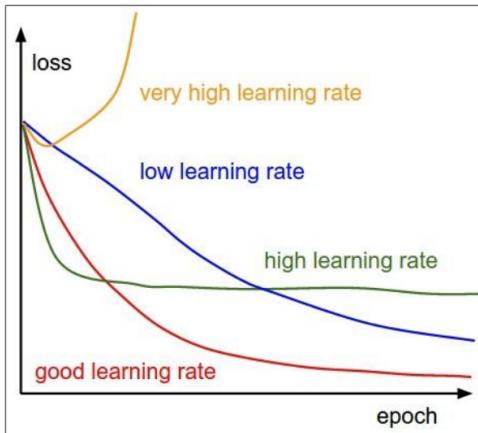


- Adam은 SGD+Momentum(살짝 overshoot)과 RMSProp(살짝 curve) 특성의 혼합인 것 관찰 가능

## Learning Rate

SGD, Adam 등 앞서 배운 optimizer들이 공통적으로 가지는 hyper parameter이다. learning rate 설정은 학습에 있어서 매우 중요하다. 강의에서 unfair comparison이라고 거듭 강조하듯이 알고리즘에 따라 적당한 수치가 다르다. 게다가 우리는 너무 큰 learning rate은 exploding, 너무 작은 값은 slow training 문제가 있음을 잘 알고 있다. 아래 그림에서 확인할 수 있다.

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

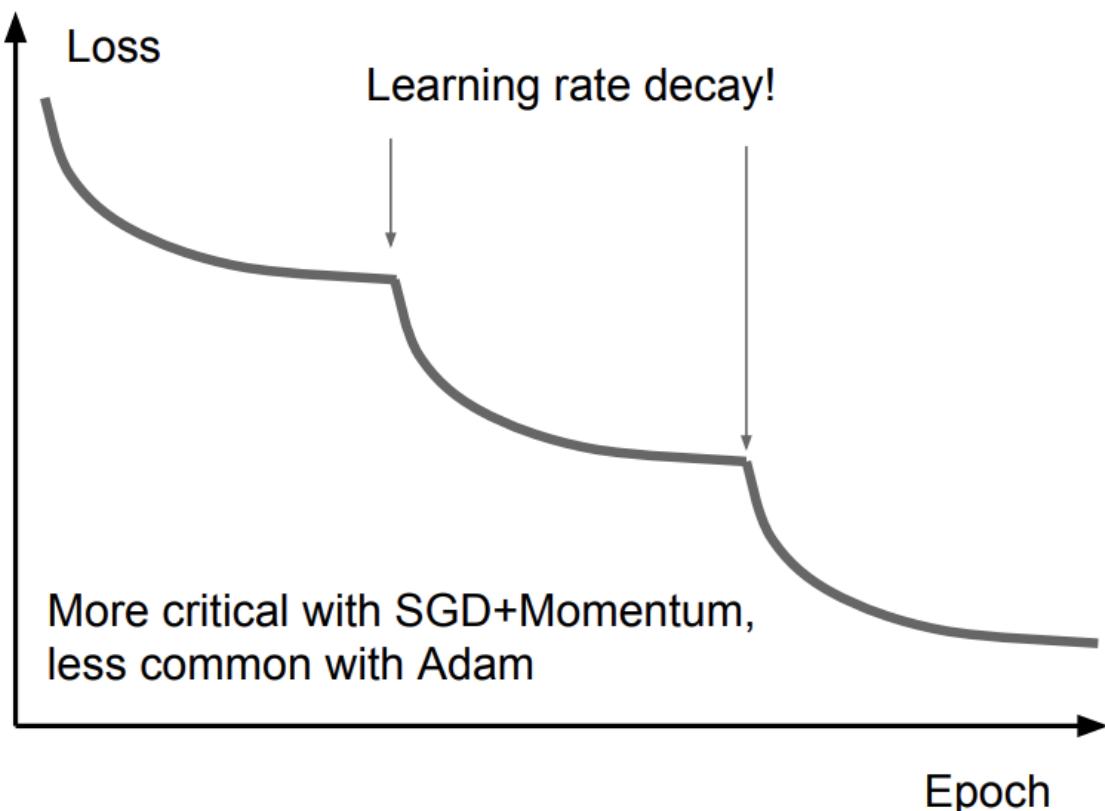


Q: Which one of these learning rates is best to use?

이에 학습을 진행하면서 고정된 값이 아니라 learning rate을 수정해가는 방법들이 제시된다.

### 1) step decay

- training에서 시간에 따라 훈련 속도를 조정하는 것이 도움이 될 수 있다. 다소 직관적인 이해를 위해서 방학 첫 세션에서 비유했듯이 높은 산에서 한 발, 한 발 내딛는 상황을 떠올려보자. 최적화 과정은 각 step(iteration 또는 epoch)마다 보폭을 줄여나가면서 최저점을 찾아간다고 생각하면 이해가 쉽다. Gradient가 점점 작아져서 loss landscape 안쪽으로 깊게 뛰어들어가고 있는 시점에 learning rate를 감소시켜주면 속도가 줄어 다시 들어갈 수 있다
- 5 epoch마다 learning rate를 반으로 줄이거나 20 epoch마다 1/10씩 줄이는 등 **몇 epoch마다 일정량만큼 학습 속도를 줄이는 방식**이다. 실전에서는 우선 no decay로 학습을 진행한다. validation error를 보고 성능 개선이 되지 않을 때 decay를 적용하여 학습 속도를 감소시킨다고 한다. 성능 개선이 되지 않는다는 것은 그림에서 기울기가 0에 가까워지는 상황이다.
- Adam보다는 **SGD+Momentum**에 쓰면 효과적.



다음의 2가지 방법은 간단히 소개한다. step decay가 해석이 용이해서 선호된다고 함.

### **exponential decay:**

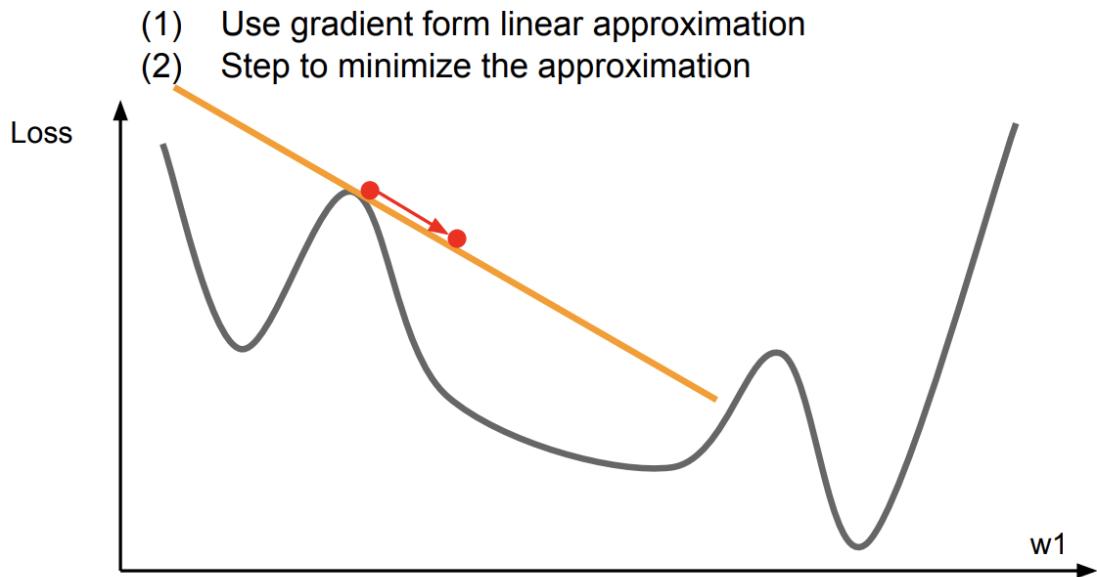
$$\alpha = \alpha_0 e^{-kt}$$

### **1/t decay:**

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

## **Second Order Optimization**

앞서 배운 알고리즘들이 First Order Optimization. First Order에서는 1차 미분 gradient를 이용해서 linear approximation, 이후 loss 최소화하는 step



Second Order는 gradient과 함께 Hessian matrix(2계 도함수) 이용해서 quadratic approximation을 한다. 이후 minima로 바로 이동.

직관적으로 Hessian matrix는 어떤 함수의 curvature(볼록/오목성)를 뜻하고, 이 정보를 사용해서 더 효율적인 update를 수행할 수 있다. 기울기의 변화율을 아는 것은 loss function의 형태에 대해서 1차 미분 접근에 비해서 더 많은 정보 포함.

second-order Taylor expansion:

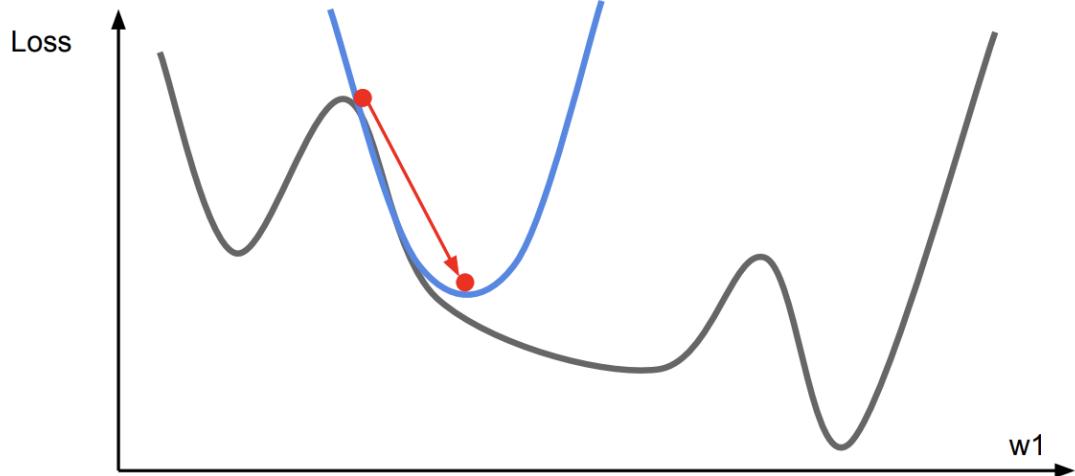
$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

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

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

Hessian has  $O(N^2)$  elements  
 Inverting takes  $O(N^3)$   
 $N = (\text{Tens or Hundreds of}) \text{ Millions}$

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



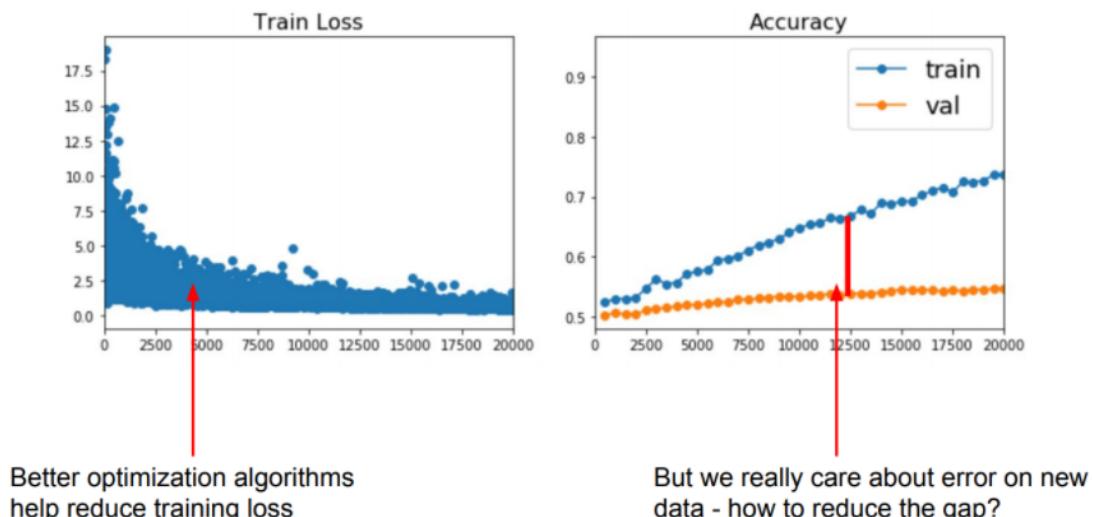
따라서 hyperparameter인 learning rate을 설정하지 않아도 된다. 하지만 Hessian Matrix를 구할 시간과 메모리 등을 고려하면 실제 적용하기에는 무리가 있다. 2차 근사도 완벽하지 않아서 결국엔 learning rate이 필요하지만 NN에서는 사용불가. matrix가  $n \times n$ 인데 parameter가 너무 많다. 그래서 Hessian을 근사시키는 BFGS 이용. Hessian의 역행렬 대신 rank를 1로 하는 approximate inverse Hessian을 update하는 방식이라고 한다.

L-BFGS가 가장 대중적이긴 하지만 역시 전체 훈련 세트를 대상으로 계산해야 한다는 단점 등의 이유들로 지금까지 이차 근사(Second order) 방법들은 잘 사용되지 않는다고 한다. mini-batch, stochastic case에는 잘 맞지 않음(full batch, deterministic)

## Training/Test error

우리의 궁극적인 목표는 unseen data에 대해 예측을 높이는 것. training error가 중요한 게 아님. 오히려 overfitting.

여기서부터는 overfitting 줄이는 여러 방법들을 살펴볼 것이다.



Neural Network의 성능을 끌어올릴 수 있는 좋은 방법은 여러 모형을 만들고 그 모형들의 평균값으로 예측하는 것이다.

여러 개의 독립적인 모형을 만들고 test 때 예측값을 averaging 하는 것이다. ensemble에 관여하는 model이 많아지면, 보통 성능은 단조적으로 개선된다. 게다가 ensemble 내에서 모형의 다양함이 늘어날 수록 성능의 개선은 더 극적이다.

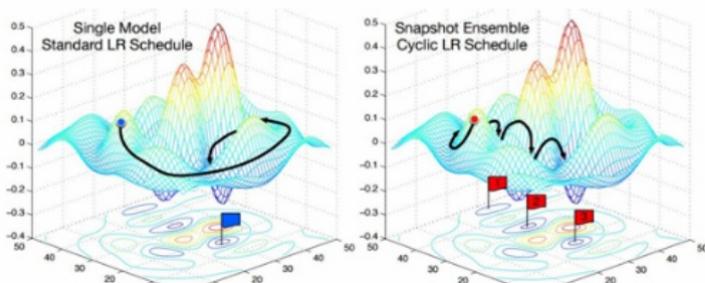
이렇듯 여러 모델들을 종합해서 performance 향상시키는 방법으로 model ensemble은 dramatic change는 기대하기 힘들지만 contest 등에서 많이들 이용하는 방법이다.

여러 독립적인 모델을 training해서 결과치를 averaging하는 양상을

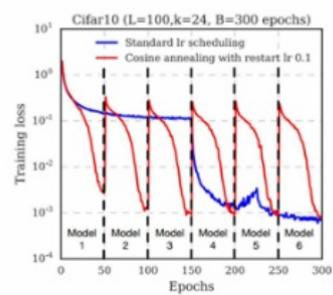
snapshot 찍듯이 single model을 training하면서 중간중간 모델을 저장하고 중간 예측값을 평균내는 방법. 모델을 한번 training하는 것만으로도 꽤 좋은 성능을 보일 수 있다.

## Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!



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!

Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 7 - 56 April 25, 2017

양상을에서 여러 모델들을 혼합해서 performance를 향상시키는 방법을 간단히 살펴봤다면 이번에는 Regularization을 통해 single model의 성능을 높이기 위한 방법들이 소개된다.

train할 때는 randomness(noise, stochasticity)를 발생시키고 test 때 randomness를 averaging-out해서 overfitting을 줄이는 전략

1) penalty term to Loss

L1, L2 regularization term을 추가해서 overfitting을 피할 수 있다 lambda 값이 모형을 단순하게 만드는 hyper parameter 역할을 한다. 이 값이 0이 되면 penalty term이 없는 것이 되어 기존 loss function과 동일하게 된다.

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

In common use:

**L2 regularization**  $R(W) = \sum_k \sum_l W_{k,l}^2$  (Weight decay)

**L1 regularization**  $R(W) = \sum_k \sum_l |W_{k,l}|$

**Elastic net (L1 + L2)**  $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

2) Dropout

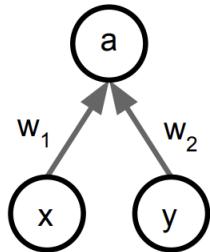
같은 network의 smaller version 또는 subset. forward passing에서 노드들을 랜덤하게 작동시킨다. training 때마다 random하게 노드들이 선택되기에 다른 subset이 채택된다. 주의할 것은 test time에는 complete network를 이용한다는 점이다. features 간 상호작용을 방지하고 특정 feature에 대한 의존도를 줄여서 결과적으로 과적합을 방지하게 된다.

single model 내에서 모델 양상을 해석할 수도 있다. random mask에 의해서 아예 다른 model인 것처럼 보일 수 있기 때문이다. dropout 역시 training 과정에서 randomness를 발생시키고(random mask), train time 때 randomness를 averaging을 통해 제거한다. averaging에서 적분 과정이 까다롭기에 근사하는 방식을 채택한다.

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

test 때 dropout 확률을 곱해서 적분을 대신해서 근사한다는 것인데 inverted dropout이라고 이 과정을 test가 아니라 training에 반영하는 방식이 있다. 상대적으로 training 과정이 길어질 수 있으나 test 과정을 간단하고 효율적으로 만들기 위해서다. 추가로, training 때의 stochasticity를 test 때 averaging 하는 방법은 batch normalization에서도 사용된다. 특정 노드가 어떤 mini-batch에 속할지는 랜덤하고 test 때는 global 관점에서 averaging 효과를 얻기 때문. 다만 dropout 확률처럼 tuning할 hyper parameter 가 없다.

## More common: “Inverted dropout”

```

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

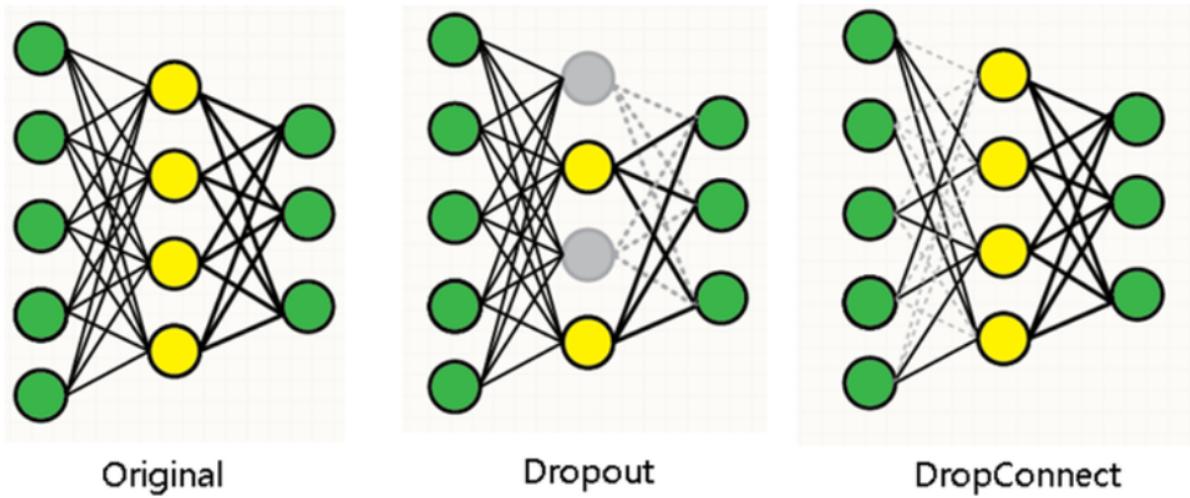
def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) /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



3) Drop Connect

dropout에 이어 이를 변형/개선 및 수학적으로 증명하는 많은 논문들이 발표되었는데 그 중 drop connect이 있다. 2013년 뉴욕대 팀이 발표한 논문 "Regularization of Neural Networks using drop connect by Li Wan, Matthew Zeiler, Yann LeCun, Rob Fergus" dropout이 노드/뉴런 자체를 생략하는 것이라면 drop connect는 노드 간 connection을 생략해서 결과적으로 weight를 생략하게 되는 방법이다. 즉 이때 노드는 남아있는 것이다. forward pass에서 dropout처럼 activation을 0으로 만들기보다 weight matrix의 값을 일부를 랜덤하게 0으로 만드는 과정인 셈이다. 자세한 수식은 "The Dropout Learning Algorithm"이라는 논문을 찾아보면 좋을 것 같다.

$$S_i(I) = \sum_{j=1}^n w_{ij} \delta_j I_j \quad \text{for } i = 1, \dots, k$$

[참고] 이해를 위해 간단한 single layer 상황에서 수식. dropout은 각 hidden activation에 independent Bernoulli random variable이 곱해진 형태로 표현된다. 아래 drop connect에서 첨자를

보면 weight matrix의 value값에 Bernoulli random variable이 1 또는 0으로 곱해진다는 차이를 볼 수 있다.

$$S_i(I) = \sum_{j=1}^n \delta_{ij} w_{ij} I_j \quad \text{for } i = 1, \dots, k$$

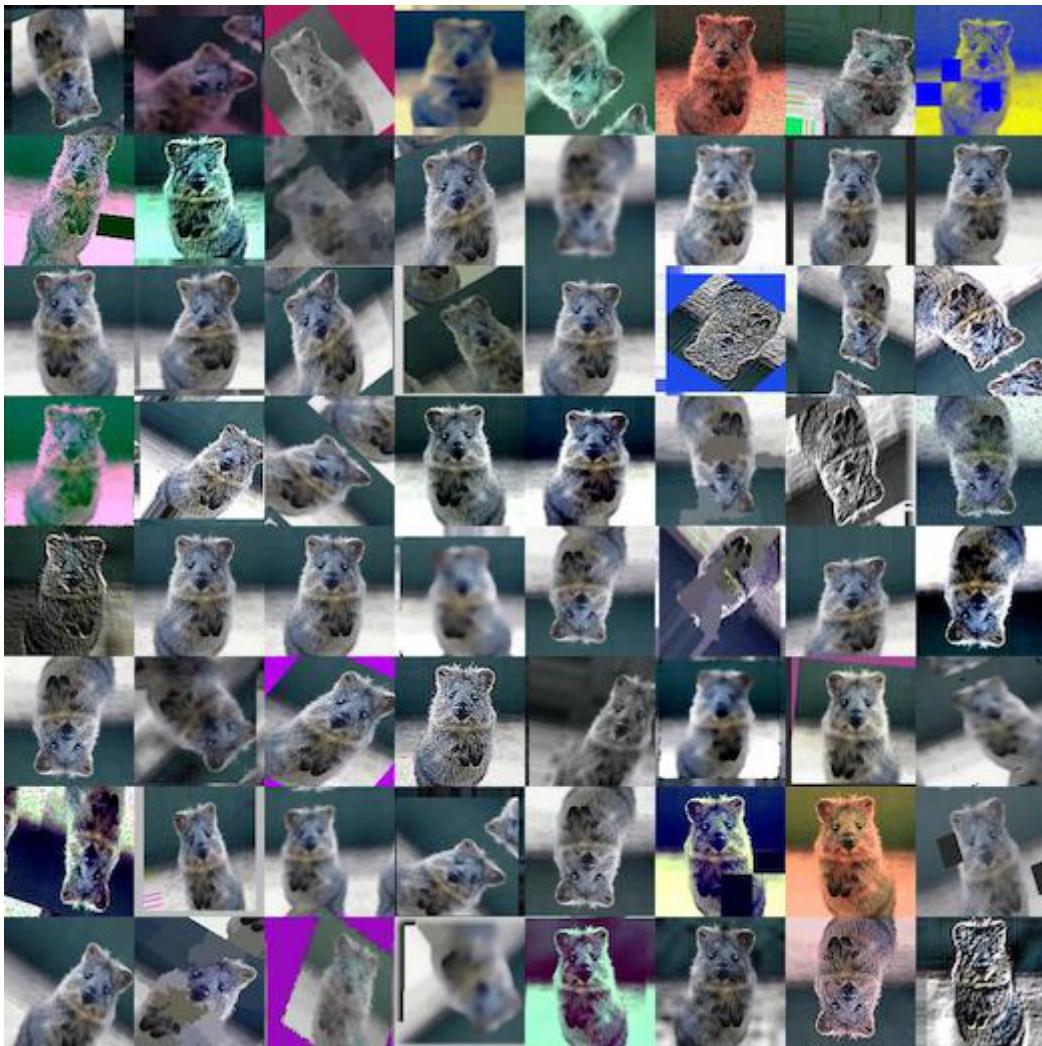
두 방식 모두 랜덤하게 특정 노드 혹은 connection을 생략함으로써 co-adaptation을 막는다는 것인데 drop connect가 dropout의 generalized version으로 이해하면 될 듯하다. nod보다 connection을 생략하는 과정이 possible model이 광범위하기 때문이다. 성능은 relu, sigmoid, tanh 등 사용한 activation function에 따라 상이한 결과를 얻었다고 한다.

drop connect 이외에도 dropout의 후속 연구로 MaxOut(2013 ICML), MaxDrop(2016 ACCV) 등이 있다 고 하니 참고.

#### 4) Data Augmentation

training data를 많이 확보할수록 overfitting을 줄일 수 있을 것이다. 하지만 양질의 데이터를 확보하는 데 시간/비용의 문제가 있다.

Augmentation은 적은 양의 원래 데이터를 변형하여 딥러닝 모델을 충분히 훈련하는 데 필요한 데이터를 확보한다. 이미지를 상하좌우로 뒤집거나(flipping) 자르기(cropping), 밝기 조절 등 많은 응용 기법들이 있다. 현실에서도 존재할 법한 데이터를 생성함으로써 좀 더 일반화된 모델을 얻는 걸 목표로 한다. 준비된 데이터보다 훨씬 많은 변형된 데이터를 학습하기에 augmentation 역시 overfitting을 방지하는 효과를 거둘 수 있다. 하지만 현실과 너무 동떨어지거나 기존 특징을 과도하게 왜곡한다면(low quality) 되려 역효과를 거둘 수 있음을 인지해야 한다. 일반적인 이미지 문제와 달리 글자 인식 문제에서 글자 이미지를 뒤집는 기법은 적절하지 않다. 글자의 의미가 왜곡될 수 있다. 심장을 촬영한 영상 등도 마찬가지다.



### 5) Fractional Max Pooling

CNN 공부할 때 Max Pooling 혹은 Average Pooling은 다들 익숙할 것이다.

pooling layer가 보통 representation의 크기를 심하게 줄이기 때문에(max pooling의 경우 dimension이 절반으로 축소) 최근 추세는 점점 pooling layer를 사용하지 않는 쪽으로 발전하고 있다고 한다.

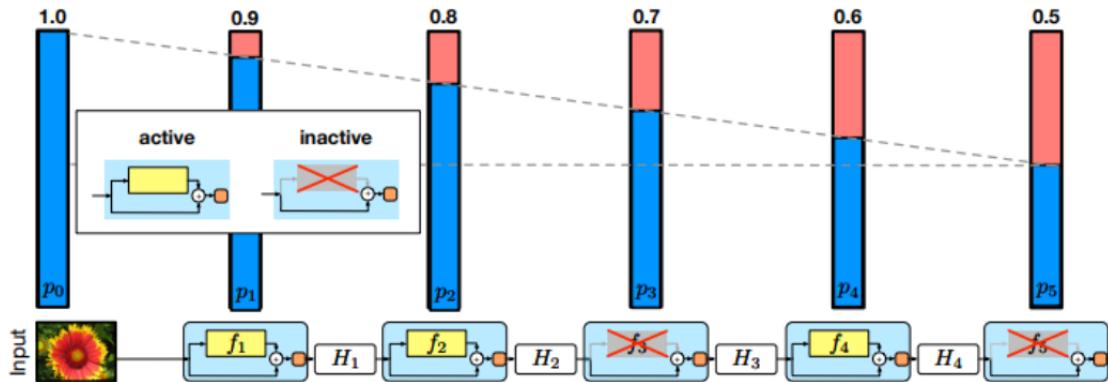
Fractional Max-Pooling은 2x2보다 더 작은 필터들로 pooling하는 방식으로 1x1, 1x2, 2x1, 2x2 크기의 필터들을 임의로(randomness) 조합해 pooling한다. 매 forward pass마다 grid들이 랜덤하게 생성되고, test 때에는 여러 grid들의 예측 점수들의 평균치를 사용하게 된다. 2x2 pooling 할 때는 size가 fixed인데 반해, fractional max pooling에서는 어느 지역에 pooling을 적용할지 random하게 작동한다. test 때 결과들을 평균내서 noise를 잡아준다. DL에서는 잘 사용되지는 않는다고 한다.

### 6) Stochastic Depth

CNN이 점점 deep해지면서 새롭게 연구할 문제가 발생했다. vanishing gradient, slow training time 등. 이를 해결하기 위한 연구가 최근에 많았는데 stochastic depth도 이 중 하나. stochastic depth는 다소 모순적인 setting 하에 실시된다; short networks에서 train하고 deep networks를 이용하여 test.

Stochastic depth는 학습 중에 layer를 무작위로(randomness) drop 함으로써 network를 단축시키는 아이디어. 결과적으로 layer간 direct connection 생성. 왜냐면 중간에 dropped layers가 있으므로 선형 layer에서 후속 layer로 향하는 short path가 형성됨. dropped layers는 input과 output이 같아져서 아무런 연산도 수행하지 않는 block으로 처리하여 (Identity Function) network의 depth를 조절한다. test 때는 whole network에 적용. 따라서 stochastic depth는 다양한 layer를 통한 ensembling으로 해석될 수 있다.

아까 소개한 Dropout과 drop connect와 비슷한 아이디어. 위의 방법들은 weight나 hidden unit(feature map)에 집중했다면, Stochastic depth란 network의 depth를 학습 단계에 random하게 줄이는 것을 의미한다. 즉 dropout이 network를 thin하게 만들었다면 stochastic depth는 network를 shorter하게 만드는 차이가 있는 것이다.

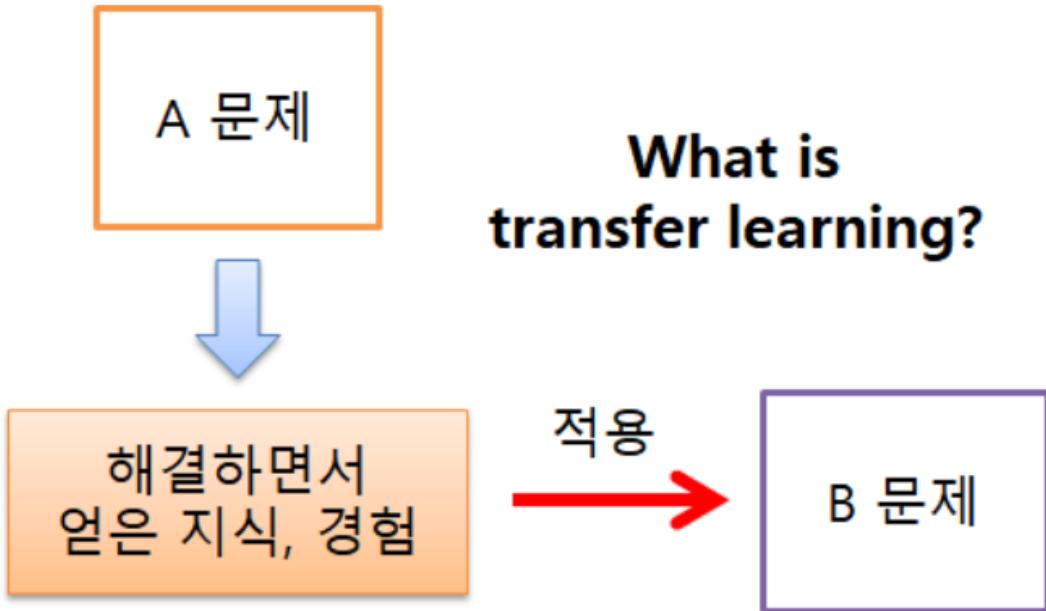


[참고] 그림에서  $p_l$ 는 survival probability로, 각 block이 drop되지 않고 '살아남을' 확률이다. training 과정에서의 new hyper-parameter이다. 1) 모든 block에 동일 확률값을 설정하는 방법이 있고 2) 그림처럼 initial layer에서 1(always active), layer를 거듭할수록 probability가 감소하는 함수를 이용하는 방법이 있는데 후자가 선호된다고 함. 해당 함수는  $p_l = 1 - \frac{l}{L} * (1 - p_{L-1})$ 로 simple linear decay rule 적용.

"the earlier layers extract low-level features that will be used by later layers and should therefore be more reliably present" -> 때문에 초기 layer에 더 높은 확률값이 할당된다. 더 자세한 사항은 "Deep Networks with Stochastic Depth"

## Transfer Training(전이 학습)

그림으로 먼저 살펴보면 다음과 같다.

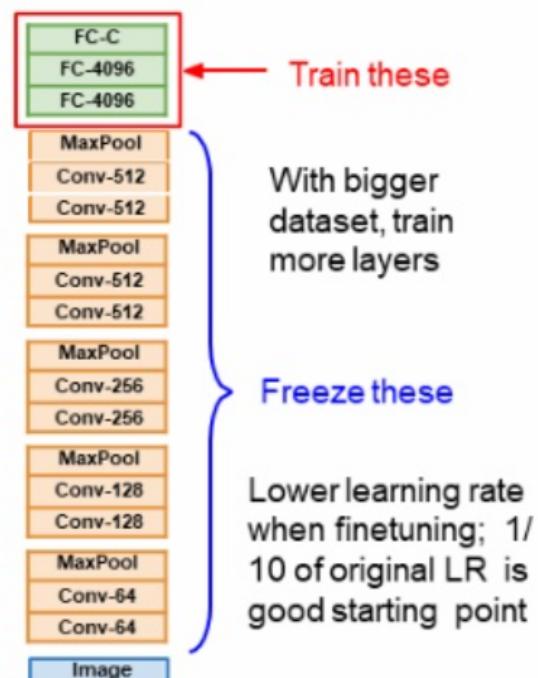


나한테 주어진 데이터가 적다. ImageNet 같이 아주 큰 dataset에 이미 훈련된 model이 있다. 이때 학습되어 있는 model의 가중치를(weight) 그대로 가져와서 parameter fine tuning만 해주면 시간도 적게 걸리고 비슷한 task 실행하니까 결과도 좋지 않을까?? 이게 전이 학습의 기본 아이디어. 처음부터 training하고 hyper parameter 찾고 시간이 오래 걸리니까 성능이 좋은 pretrained model을 차용해서 내가 가진 데이터에 적용하는 것.

## 2. Small Dataset (C classes)



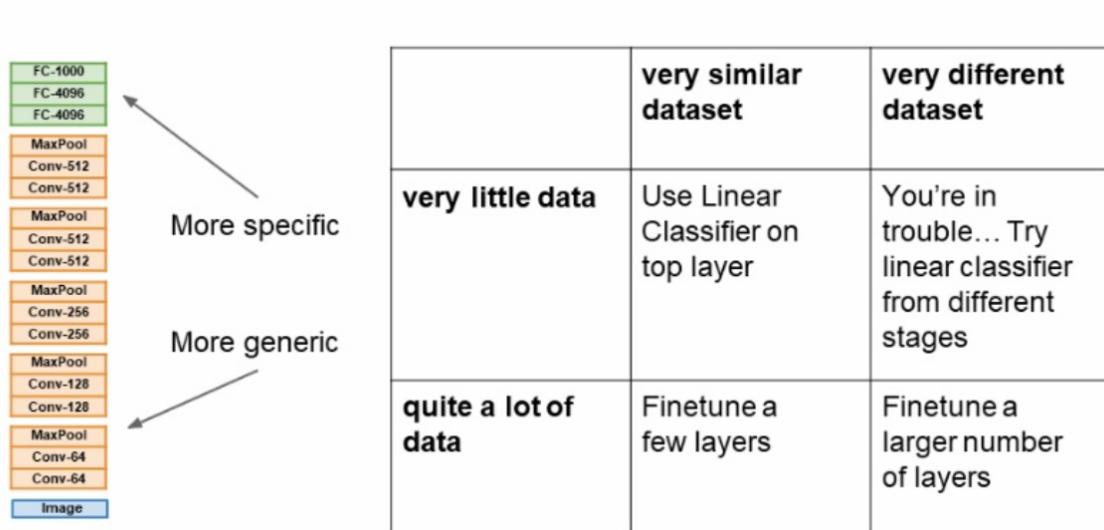
## 3. Bigger dataset



결과적으로 비교적 적은 수의 데이터를 가지고도 딥러닝 모델을 훈련시킬 수 있게 되었다고 한다. pretrained 모델을 바탕으로 우리의 분석 대상 데이터에 맞도록 파라미터 등을 미세조정(fine-tuning)해서 사용하면서 학습시간을 줄일 수 있다. 실제 CNN 구축하는 경우 대부분이 처음부터 학습하지 않는다고 한다. 일반적으로 VGG, ResNet, GoogleNet 등 이미 학습이 완료된 모델(Pre-Training Model)의 weight을 가지고 우리가 원하는 학습에 미세 조정하여 학습하고 이것이 전이학습이다. learning rate을 기준의

1/10 수준으로 설정하는 등 큰 데이터셋에 대해 알맞게 학습이 되어 있기에 learning rate를 작게 설정해서 미세 조정 과정만 거친다고 이해하면 될 것 같다.

우측 layers에서 볼 수 있듯이 data가 어느 정도 확보되어 있다면 여러 층에 대해서 training을 실행하면 될 것이다. freeze는 pretrained model의 결과치를 받아들인다는 의미.



data를 얼마나 보유하고 있는지, 또 내가 가진 data와 참고하려는 big dataset이 얼마나 유사한지에 따라 학습에 대한 가이드라인이 제시되어 있다. 참고.