

# Lecture 6 | Training Neural Networks I

이때까지 우리가 배운 것

1. 데이터 뚫음 수집 및 학습
2. 앞쪽 방향으로 계산하다보니, loss가 생김
3. Backprop을 이용하여 gradient를 계산
4. gradient를 이용하여 각종 변수들을 변화시켜나감

이번 강의부터는 어떻게 Neural Network를 학습시키는지에 대해 보다 더 구체적으로 배워볼 것이다.

1. 활성 함수, 데이터 전처리, weight 초기값 설정, regularization, gradient checking
2. 학습하고 있는 프로세스 관리, 매개변수 업데이트, hyperparameter 최적화
3. 평가

## Activation Function

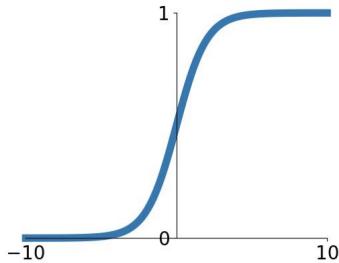
### Sigmoid

우선적으로 볼 활성함수인 Sigmoid는  $x$ 값에 따라 0과 1 사이의 함수값을 내보낸다. 값이 커질수록 1에 가깝고, 값이 작을수록 0에 가까운 형태이다. 주로 뉴런의 firing rate를 다루는데 많이 쓰인다.

# Activation Functions

$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron



## Sigmoid

하지만 이 함수에는 3가지 단점이 있다.

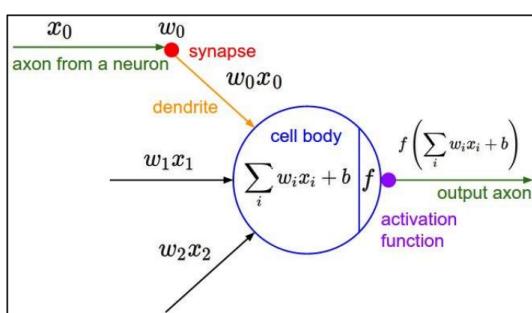
첫째, gradient 값이 상당히 작다.  $x = -10$ 인 경우를 예로 들면 그 때의 gradient는 거의 0에 가까울 정도로 작다. 뿐만 아니라  $x = 10$ 인 경우도 그렇다. Sigmoid에서는 x가 0에 가까운 값을 가질 때에만 의미 있는 gradient 값을 얻을 수 있다.

둘째, 함숫값의 중심이 0이 아니다.

만약 뉴런의 input이 항상 양수가 들어오면 어떻게 될까? 그러면 local gradient도 항상 양수가 되게 됩니다.

왜 그런지 살펴보자.

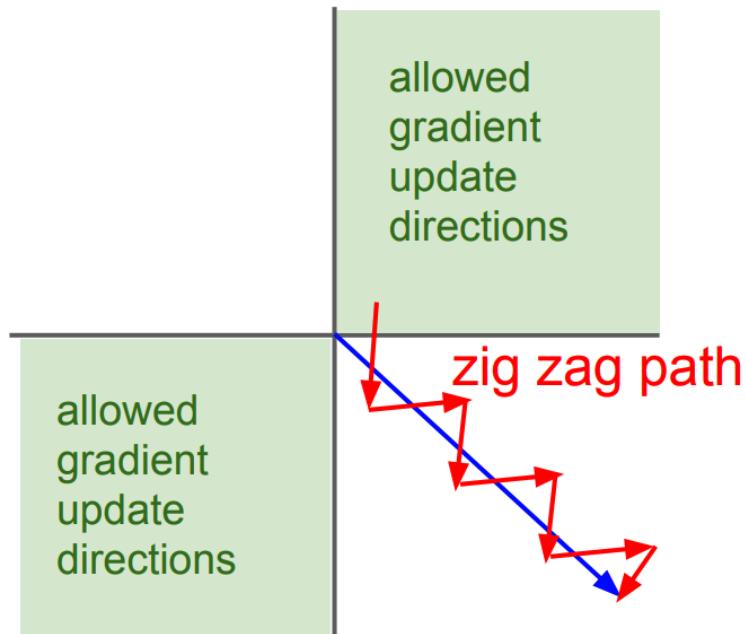
앞에서 배운 것을 통해  $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial W}$ 로 표현할 수 있다. 그런데  $f$ 가 아래의 함수이기 때문에  $\frac{\partial f}{\partial W_i} = x_i$ 가 된다. 그런데  $x_i$ 는 시그모이드 함수를 거쳐서 들어오기 때문에 항상 양수값이 된다. 그 말은 항상  $\frac{\partial L}{\partial W_i}$ 은 항상  $\frac{\partial L}{\partial f}$ 의 부호를 따르게 되고, 이는 항상 양수이거나, 음수라는 의미이다.



$$f \left( \sum_i w_i x_i + b \right)$$

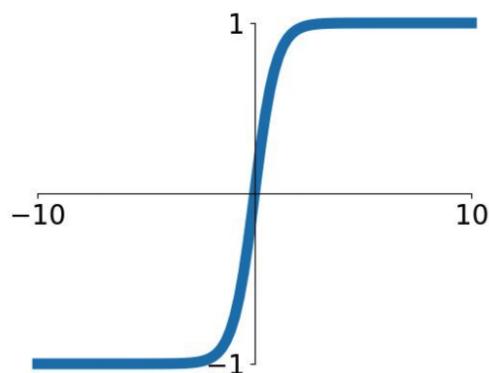
이렇게 되면  $w$  벡터를 찾는 것이 비효율적이어진다. 2차원 벡터를 생각해보면 gradient의 부호가 모두 양수이거나, 모두 음수이기 때문에 기울기도 모두 양수이거나 모두 음수이게 된다. 아래의 그림에서 파란색 벡터를 최적화된  $w$  벡터, 즉 우리가 찾고자 하는  $w$  벡터

라고 할 때 우리는 이를 찾는 과정에서 빨간색 path처럼 지그재그로 이동하면서 이를 탐색해야 한다. 즉,  $w$  벡터를 찾는 것이 상당히 비효율적이라는 것을 알 수 있다.



셋째,  $\exp()$  함수는 연산하는데 시간이 오래걸린다.

### $\tanh(x)$



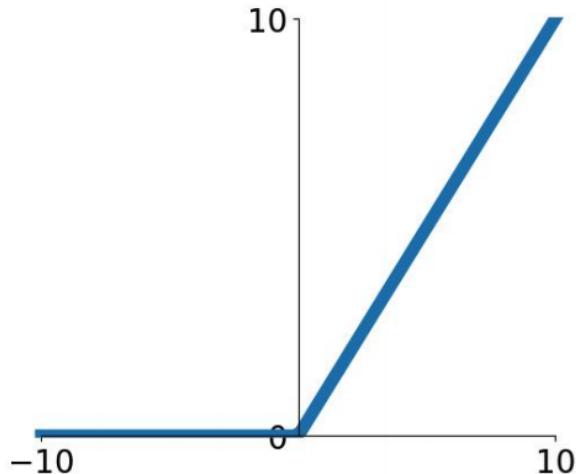
**$\tanh(x)$**

sigmoid와 비슷하지만, 함수값의 범위가 -1부터 1까지이다.

또, 이는 zero centered로 위에서 언급했던 sigmoid의 단점을 해결하였다.

하지만 여전히 0부근이 아니라면 gradient가 굉장히 작음을 알 수 있다.

## ReLU(Rectified Linear Unit)



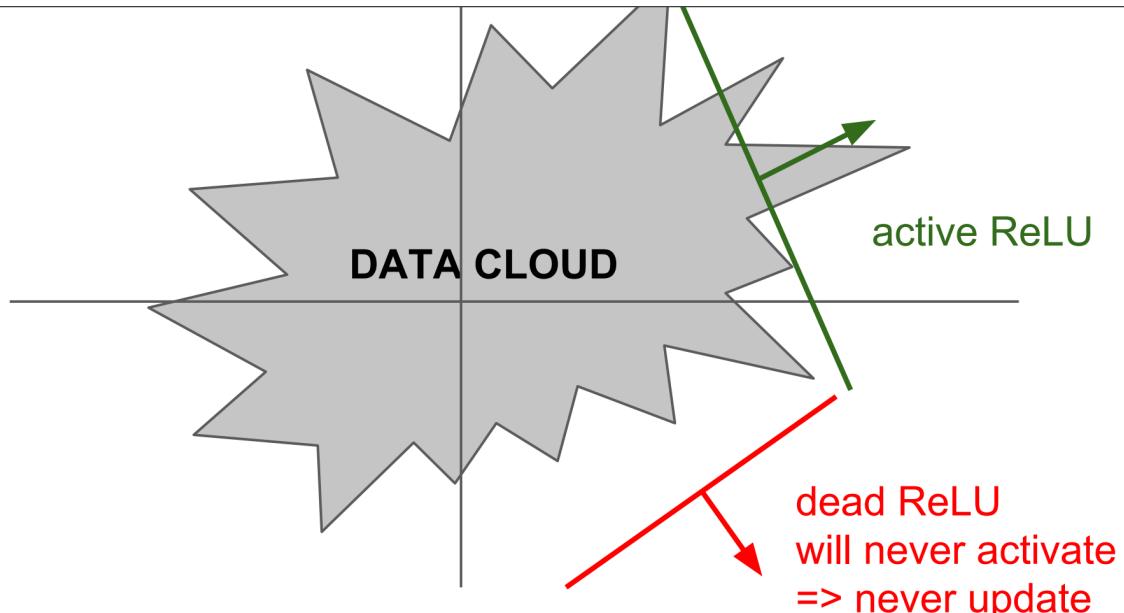
$$f(x) = \max(0, x)$$
로 연산

이는 상당히 많이 쓰이는 활성함수 중 하나로, 앞에서 보았던 기울기 손실 문제를  $x > 0$  인 부분에서는 해결하였다. 또, 연산이 상당히 간단하기 때문에 앞의 두 함수들(sigmoid , tanh)보다 대략 6배정도 빠르게 연산이 가능하다.

하지만 zero-centered 문제가 또다시 발생하였음을 알 수 있다. 또한 음수 부분에서는 기울기가 항상 0이기 때문에 기울기 손실 문제가 아직 남아있다.

아예 기울기가 0이 되어버리면, 더이상 weight가 변화하지 못할 것이고 이렇게 하는 부분을 우리는 dead ReLU라고 한다. 즉 아래의 그림처럼 ReLU에는 active ReLU와 dead ReLU가 존재한다. 이런 상황에서, 만약 초기값을 잘못 설정하여 dead ReLU에 빠지게 된다면 gradient를 구해도 항상 0이 나오기 때문에 weight를 업데이트 하는 것이 불가능해진다.

일반적으로는 training 과정이 상당히 긴 상황에서 정상적인 초기값에서 시작하여도 학습 과정, 즉 gradient를 이용하여 weight를 변화시키는 과정에서 dead ReLU에 빠지게 되는 경우가 많다. 실제로 이를 활용해보면, 10~20%정도가 dead ReLU에 빠진다고 한다.

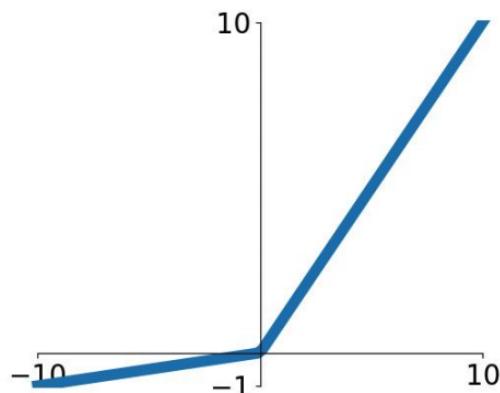


## leaky ReLU

ReLU를 조금 수정한 것인데, 차이점은 음수 부분에서 음의 기울기를 가지고 있다는 것이다.

ReLU와 다르게 dead ReLU에 빠지지 않는다는 특징이 있다.

$f(x) = \max(0.01x, x)$ 로 표현된다.

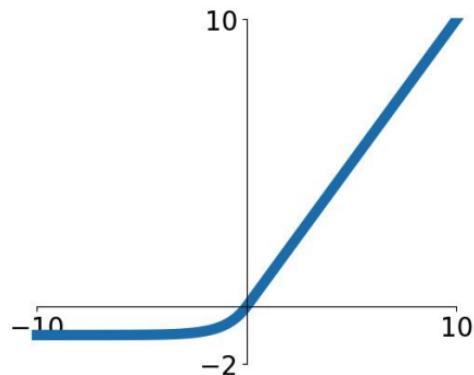


이와 유사하게, Parametric Rectifier (PReLU)가 있는데, 이는  $f(x) = \max(\alpha x, x)$ 로 표현된다.

## Exponential Linear Units(ELU)

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

ReLU의 변형으로 나온 또 다른 활성함수인 ELU인데, ReLU의 장점을 모두 가지고 있다. 또 평균이 0과 가까우며, 음수 부분에서 Leaky ReLU와 비교하였을 때, 노이즈를 보정하기 위해 더 단단하게 만들어져 있다. 하나 작은 음수값으로 갈 수록 기울기가 0과 유사해지면서 기울기 손실 문제가 발생한다. ReLU와 Leaky ReLU 사이의 단계라고 볼 수 있다. 하지만  $\exp()$ 함수가 있어서 연산 속도 느려진다는 단점도 가지고 있다.



## Maxout "Neuron"

이는 일반적인 형태를 가지고 있지는 않은데, ReLU와 Leaky ReLU의 장점만을 가져왔다고 할 수 있다.

절대 죽지 않고, 기울기 손실도 없는 장점이 있다.

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

하지만 매개변수와 뉴런의 개수가 2배가 된다는 단점이 있다.

결과적으로 요약하자면.

ReLU를 사용하는 것이 좋다.

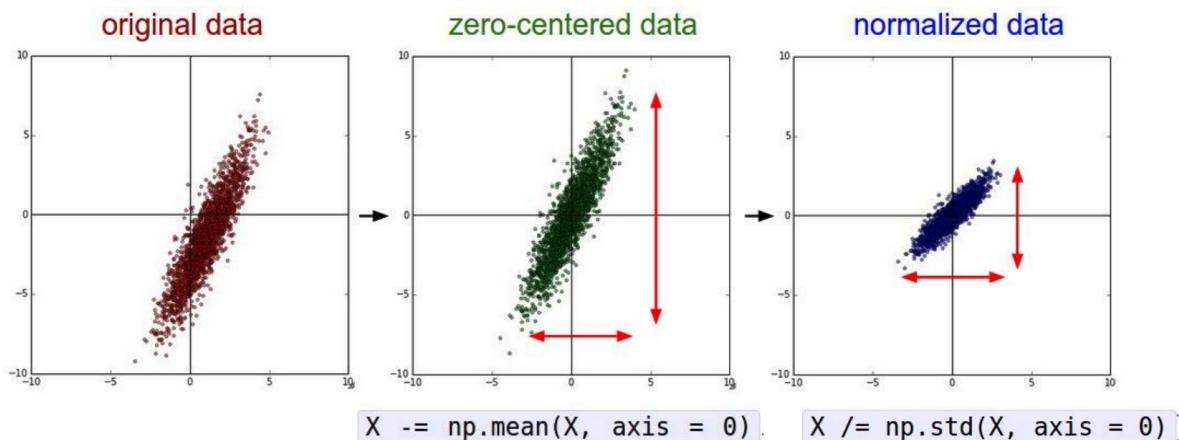
또, Leaky ReLU / Maxout / ELU 도 시도해보자

tanh는 시도해보되, 많이 기대하지는 말고, sigmoid는 쓰지 말자.

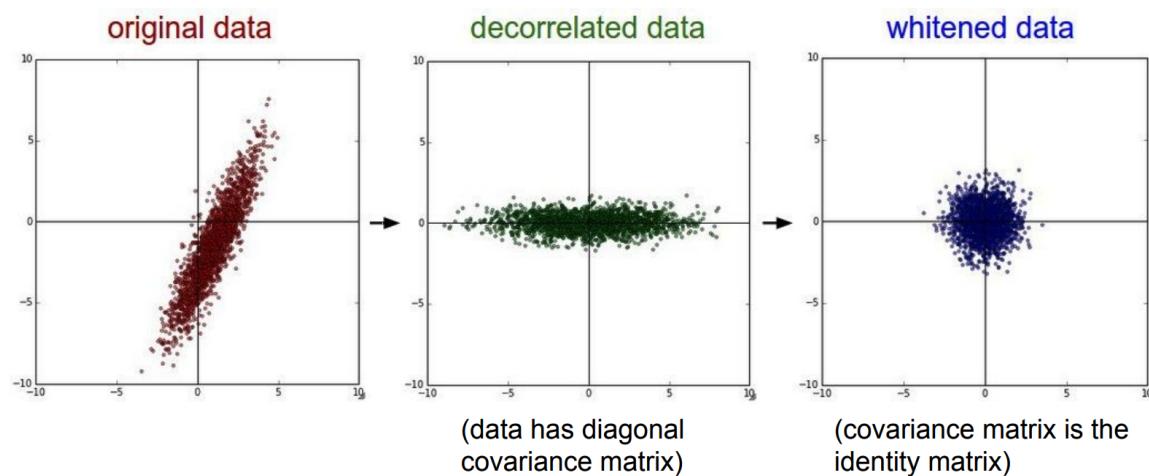
## Data Preprocessing

딥러닝이나 머신러닝에서는 전처리 과정이 상당히 중요하다.

데이터를 받으면 전처리 과정에서 zero-centered data로 변경시키고 너무 값의 편차가 클 수도 있으니 표준 편차로 나누어 주는 normalized data로 변경시키곤 한다. 그런데, image data는 픽셀의 RGB 값이 0~255 까지 범위가 정해져있기 때문에 normalization은 하지 않고, 기존의 데이터를 zero-centered 데이터로 변화시키기만 한다.



또는 아래처럼 PCA 또는 Whitening 과정을 전처리 과정을 거칠 수도 있다. 잘 사용하진 않는다.



위에서 말한 것처럼 이미지를 다룰 때에는 centered 과정만 거치게 된다. 이 방법에도 2 가지가 있는데 전체 평균을 계산하여 빼는 방법이 있고, 각각의 채널별로 평균을 계산하여 빼는 방법이 있다. 채널이란, RGB를 의미한다. 즉 Red 계열별로 평균을 내서 뺀고, Green, Blue도 각각의 계열별로 평균을 내서 뺀다는 의미이다.

# Weight Initialization

Weight를 gradient를 이용한 update를 진행한다고 해도 분명 초기값이 필요하다. 우리는 어떻게 이 초기값을 설정할까?

우선, 만약  $W=0$ 을 초기값으로 사용하면 어떻게 될까?

모든 뉴런이 동일한 행동을 할 것이다. 모두 값을 가지고, 모두 같은 gradient를 가지게 되면 계속 모두 동일하게 변화할 것이다.

## First Idea : small random number

여기서는 정규 분포 함수에서 얻은 값들에 0.01을 곱하여 그것들을  $W$ 의 초기값으로 사용할 것이다.

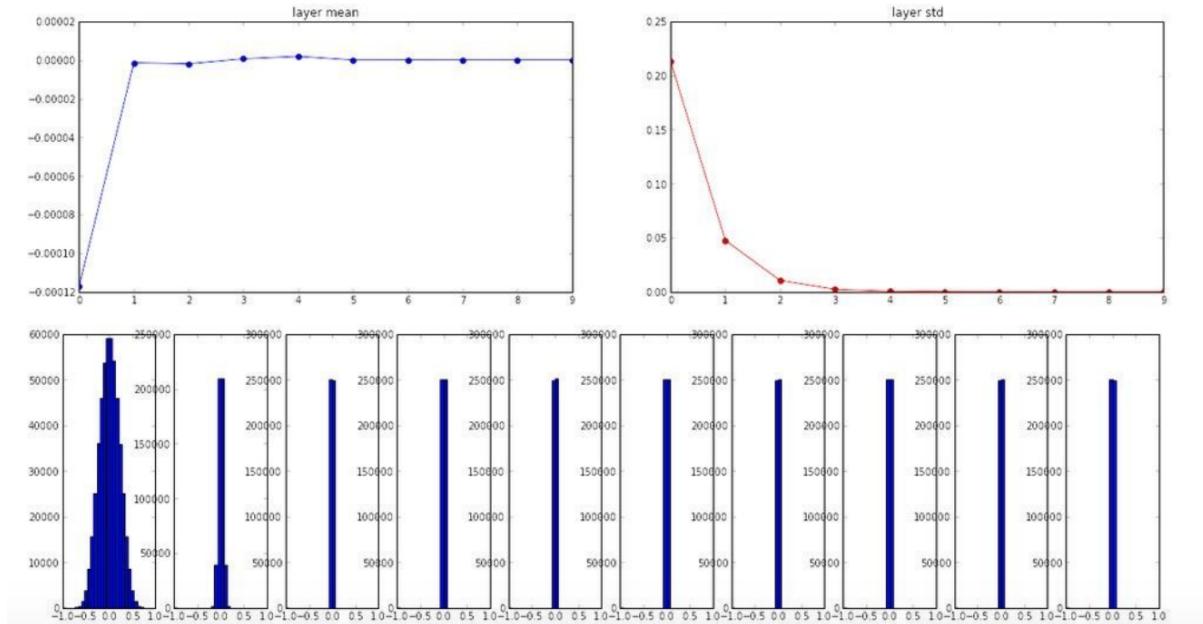
활성함수는 tanh를 사용하면서 위에서처럼  $W$ 를 초기값으로 설정했다. tanh는 결과값이 -1하고 1 사이이기 때문에 매번 layer에서 평균은 0에 가까웠다. 그런데 매 layer를 지날 수록, 표준편차도 0에 가까워 졌다. 아래의 히스토그램을 보면 점점 값들의 분포가 줄어드는 것을 관찰할 수 있고 결국 10개의 layer가 지나고 나서는 모든 값들이 0이 되어 우리가 원하지 않는 결과를 초래하였다.

만약 이때 backward pass를 한다면 gradient는 어떻게 변할까?  $dW_1 = x_i * dW_2$ 라고 했을 때,  $x$ 가 점점 0으로 수렴하기 때문에 gradient도 0으로 수렴한다는 사실을 알 수 있다. 여기서, gradient 손실이 발생하게 된다.

```

input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000

```



그렇다면 조금 큰 값을 사용하면 어떻게 될까?

그러면 대부분의 값이 -1과 1 근처가 될 것이며 gradient가 0이 되어버릴 것이다. 이런 문제 때문에 적당한 초기값을 설정하는 것은 상당히 어렵다. 아래의 히스토그램을 보면 대부분의 값이 -1 또는 1에 밀집되어 있음을 알 수 있다.

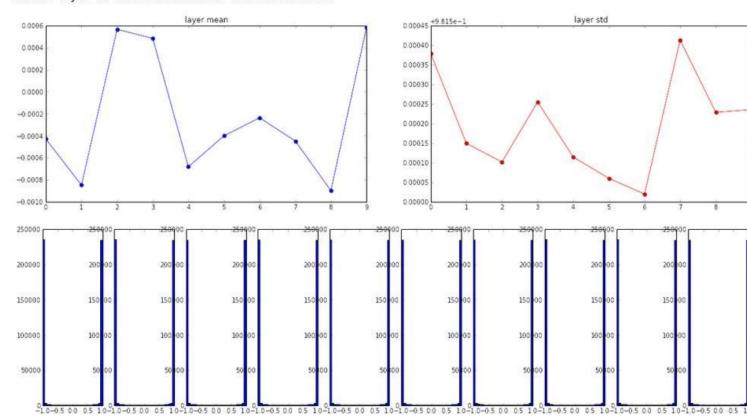
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```

input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736

```

\*1.0 instead of \*0.01

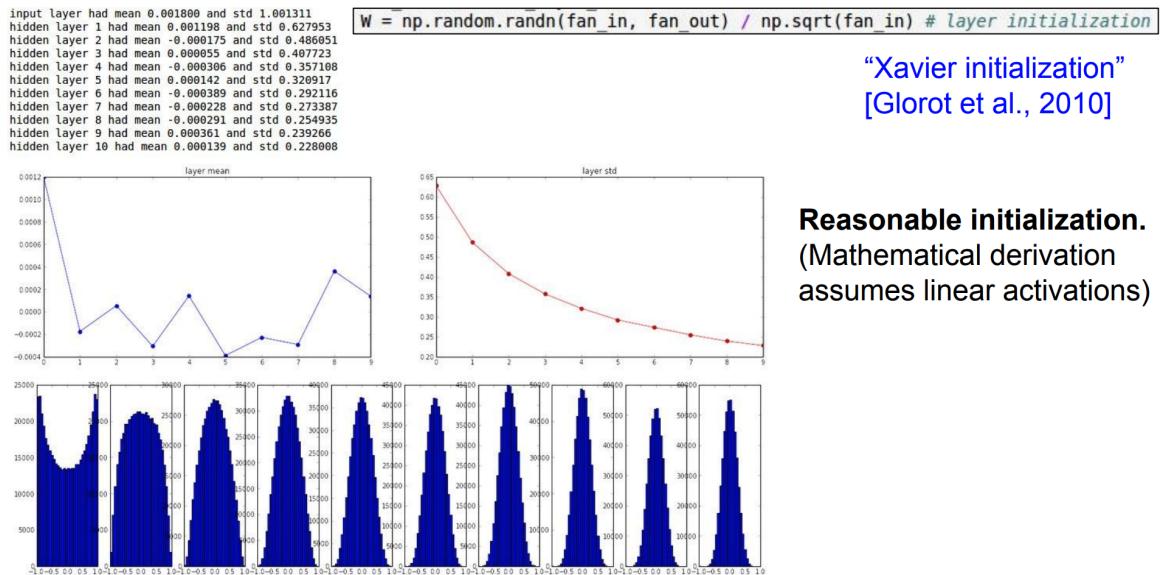


Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

여기서 우리가 사용할 방법은 Xavier initialization이다.

여기서는  $W = np.random.rand(fan\_in, fan\_out)/np.wqrt(fan\_in)$  이라는 식을 사용한다.

즉 랜덤 값들을 정규 분포를 통해 구하되, 그것을 그 개수로 나누는 방법이다. 우리가 많은 input을 가지고 있다면, 값들이 커지는 것을 방지하여 큰 값으로 나누는 것이고, 적은 input을 가지고 있다면 값들이 작아지는 것을 방지하여 작은 값으로 나누는 것이다. 이는 상당히 합리적인 방법이라고 할 수 있다. 아래의 히스토그램을 보면 tanh에 상당히 잘 적용되었음을 알 수 있다.



하지만 이를 ReLU에서 사용한다면, 문제가 발생한다. ReLU에서는 절반의 값을 0으로 만들기 때문에 효과적으로 동작하지 않았다.

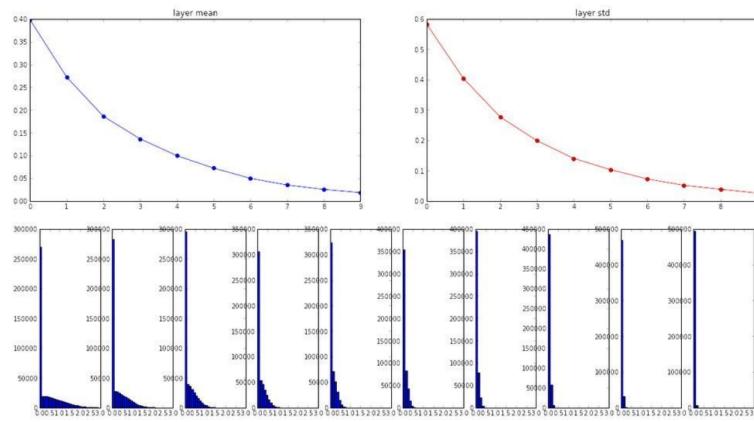
```

input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186976 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099444 and std 0.149585
hidden layer 6 had mean 0.072234 and std 0.125389
hidden layer 7 had mean 0.049775 and std 0.077748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076

```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.



그런데 ReLU에서는 개수를 나눌 때, 2를 나누어준 후에 사용하면 된다는 연구 결과를 발표하였고 그래서 ReLU에서는 2를 나누어 사용한다고 한다.

여기서는  $W = np.random.rand(fan\_in, fan\_out)/np.wqr(fan\_in/2)$ 을 사용하였다.

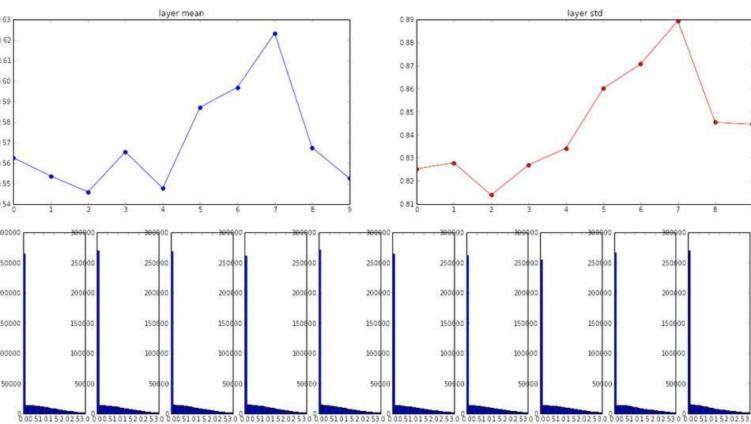
```

input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.550667 and std 0.826555
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587183 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523

```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

He et al., 2015  
(note additional /2)



Weight Initialization은 현재도 활발하게 연구되고 있는 분야이다.

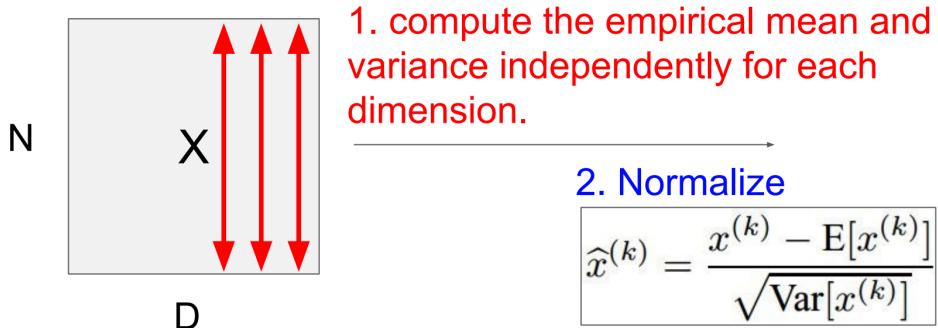
## Batch Normalization

이는 우선 vanishing gradient가 발생하지 않도록 하는 방법 중 하나이다.

앞에서 여러 방법들을 통해 개선하였지만, Batch Normalization은 학습하는 과정에서 속도도 빠르게 하고, 안정화 하는 방법에 대한 것이다.

학습 과정에서 불안정화가 발생하는 이유를, 데이터들의 편차가 크기 때문이라고 생각했고, 각 layer를 지나칠 때 Normalization을 거치게끔 하였다. 매번 정규분포를 통해 normalization을 시행하였다.

N개의 training data가 주어지고 각각의 차원이 D라면, 같은 차원끼리 평균과 분산을 구해 정규분포를 통한 normalization을 해준다. 주로 이 batch normalization은 FC(or Conv) layer와 Activation layer 사이에서 시행해준다.



그런데, 매 layer마다 Batch normalization이 필요한 것은 아니다. 이 또한, 학습에 의해 어느 정도의 Batch normalization을 할지 직접 선택할 수 있게끔 구현하였다.

아래에서 볼 수 있는 것처럼 우선 Normalization을 진행하고 학습에 의해 얻어진 값  $\gamma, \beta$ 를 사용하여 두 번째 식에 적용시킨다. 만약  $\gamma$ 가 표준편차와 동일하고,  $\beta$ 가 평균과 동일하다면 Batch Normalization은 상쇄되어 원래 값이 그대로 Activation layer로 넘어가게 될 것이다.

Normalize:

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

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:

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

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

to recover the identity mapping.

위에서 설명한 전체 진행 과정은 아래와 같다.

<b>Input:</b> Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$ ;	
Parameters to be learned: $\gamma, \beta$	
<b>Output:</b> $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$	
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$	// mini-batch mean
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$	// mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$	// normalize
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$	// 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

## Babysitting the Learning Process

이제 process를 학습하는 과정에 대해 알아보자.

우선, 첫번째는 전처리 과정을 거친다. 앞에서 말했던 것처럼 이미지에서는 zero-centered만 시행한다.

두번째로는 architecture을 선택한다.(ex) 50 neuron with 1 hidden layer)

시작과 동시에 loss를 구한다. sanity check를 하는 것인데, 우선 regularization을 0으로 하고, 앞에서 배운것처럼  $-\ln 1/10 = 2.3$  이 나오는것을 보아 제대로 작동하는 것을 알 수 있다. 또, 이 regularization을 증가시켰을 때, Loss가 올라가는지 재확인 함으로서 정상작동하는지 체크할 수 있다.

## 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

이후, regularization을 0으로 한 채, test data중 일부만을 가지고 training을 시작한다. training을 계속 하면서 training set과 validation set의 정확도가 1.0으로 가는 것을 보고, 우리가 설정한 것들이 정상적으로 작동한다는 것을 알 수 있다.

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=0.0,
                                    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 epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

learning rate를 상당히 작은 값으로 하고, training을 시키는 상황을 가정해보자.

learning rate가 작기 때문에 loss의 변화는 거의 없을 것이다. 그런데 실제로 해보면 정확도는 상승한다. 왜그럴까?

```

model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                    model, two_layer_net,
                                    num_epochs=10, reg=0.000001,
                                    update='sgd', learning_rate_decay=1,
                                    sample_batches=True,
                                    learning_rate=1e-6, verbose=True)

Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000

```

## Loss barely changing

간단하게 말하자면, loss가 변하지 않더라도 training이 계속 되고 있기 때문에 정확도가 어느 정도까지는 올라가는 현상이 발생하는 것이다.

만약 learning rate가 크면 어떻게 될까? 그러면 loss의 값이 NaN으로 출력되거나, infinity로 출력될 것이다. 만약 이런 출력을 보게 된다면 거의 대부분 learning rate의 값이 크기 때문이고 우리는 적절한 learning rate가 그 값보다는 작다는 사실을 알 수 있게 된다.

# Hyperparameter Optimization

## Cross-validation 전략

처음에는 적은 epoch을 통해 적당한 아이디어와, 어떤 매개변수가 잘 동작하는지에 대한 대략적인 데이터를 뽑아낸다. (epoch : 전체 데이터셋에 대해 forward/backward 을 거친 상태, 즉 한번 학습을 완료한 상태를 의미함)

아래의 예시를 보면 5 epoch으로 시행을 하면서 대략적으로 정확도가 높은 learning rate에 대한 값을 추출해 낸다. 코드를 보면  $10^{**\text{uniform}(-5,5)}$ 와 같은 형태로 만들어 놓았는데, 이렇게 하면 log space에서 효과적이라고 한다.

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)
```

note it's best to optimize in log space!

```
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: 2.110807e-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)
```

nice →

이후에 점점 시간을 늘려나가면서 세부적인 탐색을 시작하는 것이다.

아래의 예시를 보면 처음의 범위보다 범위가 많이 줄어든 것을 볼 수 있다. 또 validation set 정확도도 이전보다 높은 0.53..의 값이 나오는 것을 확인할 수 있다. 정확도가 늘어난다는 것은 제대로 탐색을 하고 있다는 것에 대한 증거라고도 할 수 있다.

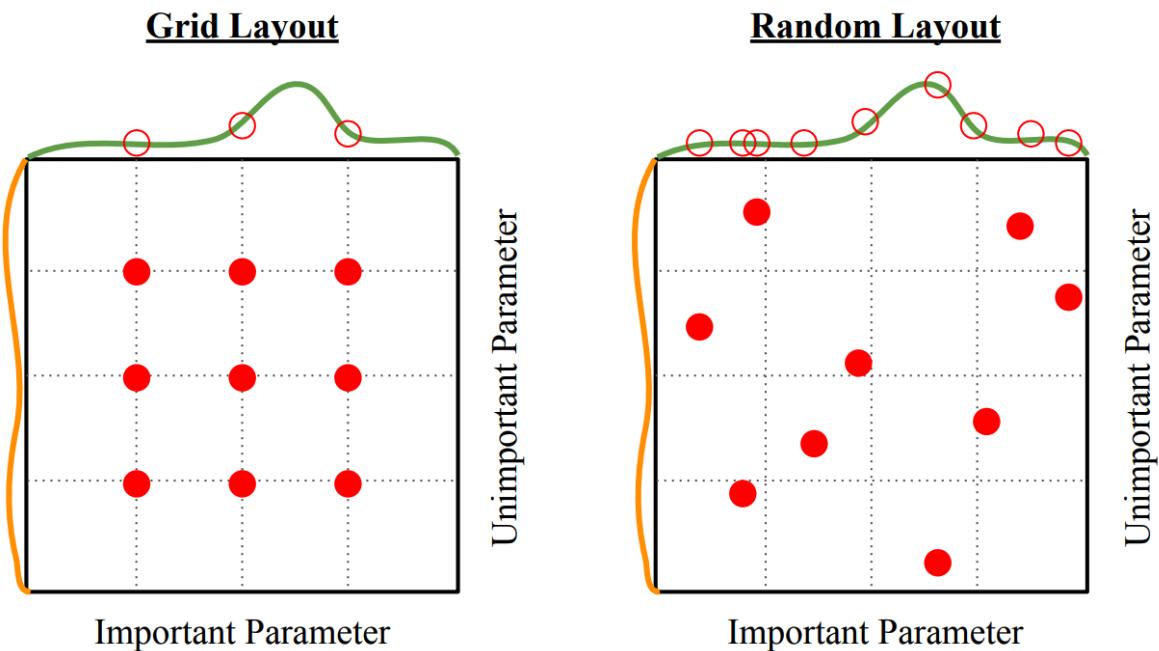
그런데 아래를 보면 높은 정확도를 보이는 learning rate 중 9.471...e-04라는 값이 있는데 이는 우리가 범위로 설정한  $10^{-3}$ 에 상당히 가까운 값이다. 즉, 우리가 범위를 잘못 설정했을 수도 있겠다는 가능성을 열어주는 지표이다. 이럴때는 그 범위를 바꾸어 나가며 다시 진행해볼 필요도 있다.

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6) → adjust range → 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: 2.287897e-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: 1.562808e-02, (13 / 100)
val acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100) ← 53% - relatively good
for a 2-layer neural net
with 50 hidden neurons.
But this best
cross-validation result is
worrying. Why?
```

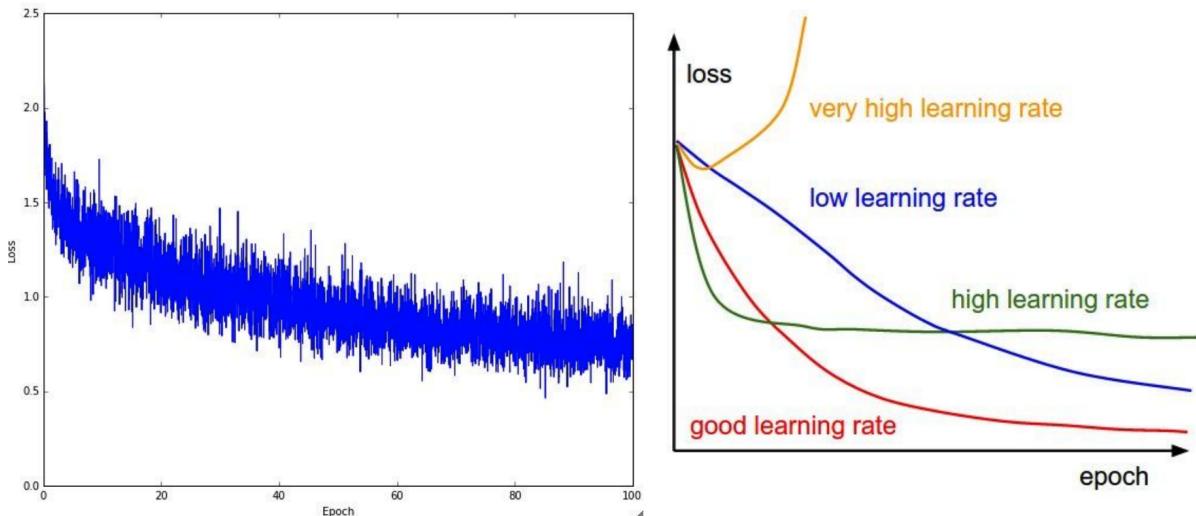
우리는 지금 Random Search를 하고 있지만, 이 뿐 아니라 Grid Search도 존재한다. Grid Search는 탐색하는데에 규칙성있게, 아래의 그림처럼 탐색을 하는 것이다. 하지만 이렇게 탐색을 하는 경우에는 최적의 값을 찾을 수 없을 확률이 높다. 아래의 그림과 같이 값들이 존재한다고 할 때, Grid Search를 아무리 많이 진행하더라도 저 최적값은 찾기 힘들다.

이처럼, Hyperparameter의 최적값을 찾을 때에는 반드시 Random Search를 사용해야 한다.



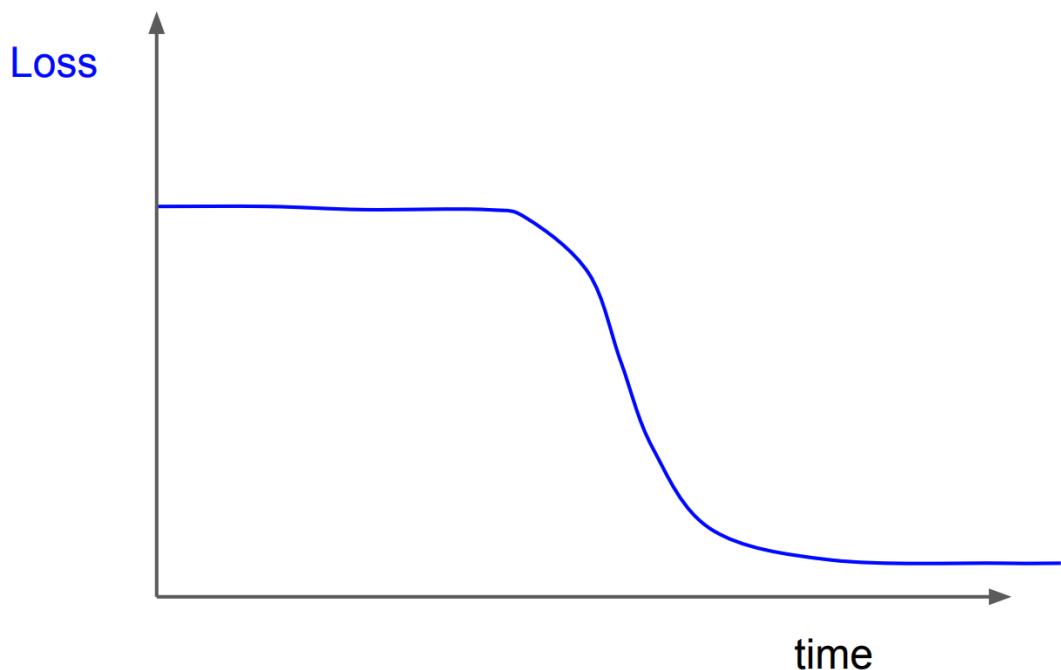
우리가 찾아야 할 Hyperparameters는 상당히 많다. 어떤 network architecture을 사용할 것인지, 어떤 learning rate를 사용할 것인지, 그 learning rate를 구하는 과정에서 어떻게 범위를 바꿔 나갈 것인지, 어떤 regularization을 사용할지 등이 있다. 이런 값들은 우리가 튜닝을 하는 것처럼 맞춰 나가며 최적의 Hyperparameter를 구하여 성능을 올려야한다.

아래와 같이 그래프가 나온다면 어떻게 해야할까?



그래프를 보면 감소하는 정도가 상당히 작다는 것을 알 수 있다. 이럴 때에는 learning rate를 증가시켜서 loss가 변화하는 정도를 더 키우는 방법이 있다. 오른쪽의 그래프의 개형들은 어떤 형태가 좋은 learning rate의 그래프인지, 어떤 형태가 나쁜 learning rate의 그래프인지에 대한 예시이다.

또, 만약 이런 경우는 어떤 문제때문에 발생하는 것일까?

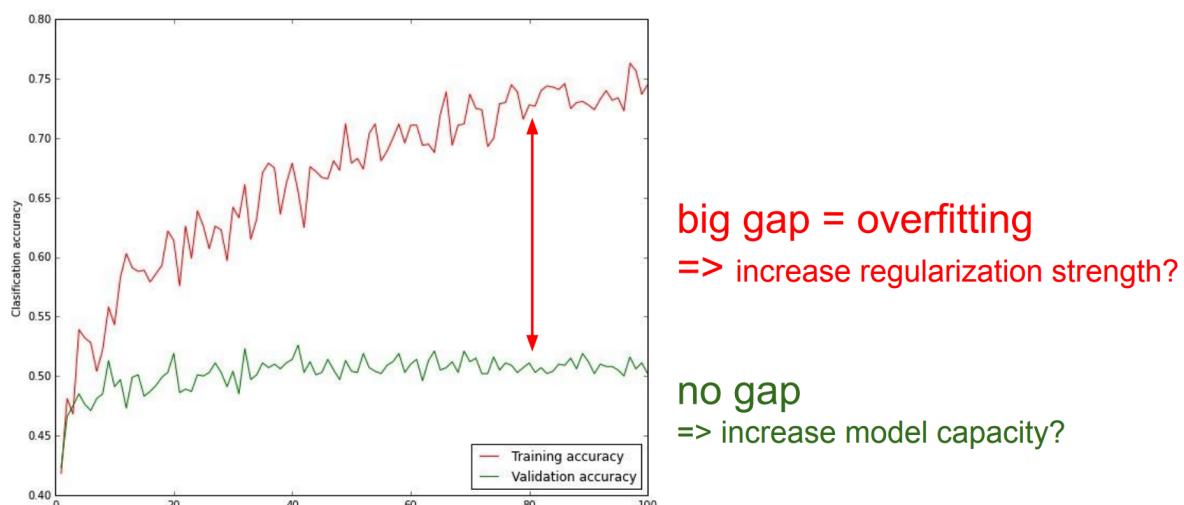


이는 아마  $W$ 의 초기값의 세팅이 잘못되었을 것이다. 초기값이 잘못 세팅되어, gradient의 변화가 거의 없기 때문에 loss의 변화가 없게 되고, 그것들이 계속 쌓여서 오랜 시간에 걸쳐 loss에 변화가 생긴 상황이라고 볼 수 있다.

loss 이외에도 우리가 바라보아야 하는 수치는 accuracy이다.

아래의 그래프를 보면 Training data와 validation data의 정확도를 보여주고 있다. 물론, 두 정확도에는 차이가 존재할 수 밖에 없다. 하지만 아래처럼 너무 많은 차이가 있다는 것은 regularization의 정도를 증가시켜주어야 한다. 또 만일 차이가 전혀 없다면 training data와 validation data에 차이가 없다는 것이며 이 때는 data의 capacity를 증가시켜 줄 필요가 있다.

Monitor and visualize the accuracy:



또 볼만한 값은 Weight의 변화율이다. update된 weight의 크기를 전체 weight의 크기로 나누어주는 것인데, 대체적으로 0.001 혹은 그 부근의 값인 경우가 좋다고 한다.