



Type something...

## 01) Training Neural Networks 1

### 01\_Activation Functions

1. Sigmoid functions
2. tanh(x)
3. ReLU = max(0, x)
4. Leaky ReLU = max(0.01x, x)
5. Exponential Linear Units(ELU)
6. Maxout Neuron

### 02\_Data Preprocess

### 03\_Weight Initialization

1. Xavier Initialization
2. He Initialization

### 04\_Batch Normalization

### 05\_Babysitting the learning process

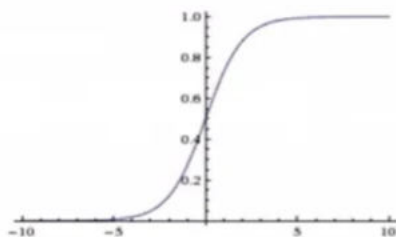
### 06\_Hyperparameter Optimization

## 01) Training Neural Networks 1

### 01\_Activation Functions

#### 1. Sigmoid functions

## Activation Functions



**Sigmoid**

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

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

3 problems:

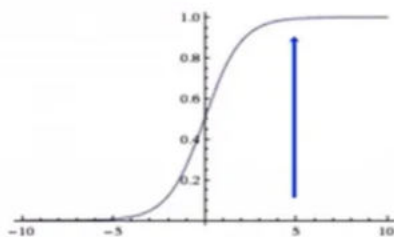
1. Saturated neurons "kill" the gradients

- Sigmoid function은 지금은 잘 사용되지 않는 함수이다.

1. 그 이유는 gradient vanish 문제가 있기 때문이다.

- x값이 꽤 크거나 꽤 작으면 local gradient가 0에 가깝게 된다.

## Activation Functions



**Sigmoid**

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

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

3 problems:

1. Saturated neurons "kill" the gradients

2. Sigmoid outputs are not zero.

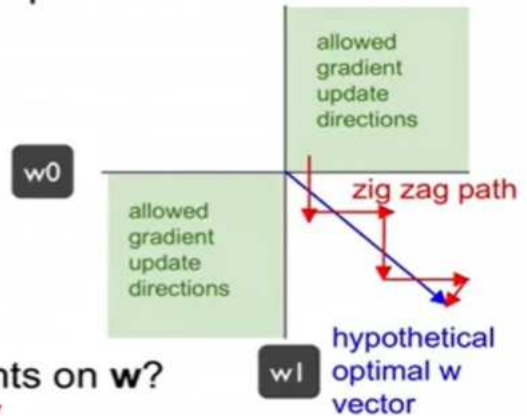
## 2. Sigmoid outputs are not zero centered

2. 두번째 문제는 output이 0이 중심이지 않아 수렴하는데 오래 걸린다.

- 만약  $x$ 가 양수라면  $w$ 에 대한 gradient가 모두  $dl/df$ 의 부호가 같아지기 때문에 모두 양수이거나 모두 음수이다.

Consider what happens when the input to a neuron is always positive...

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



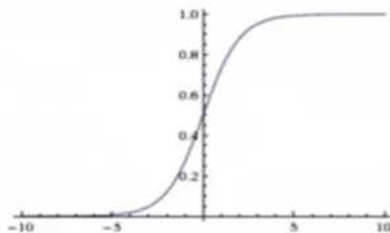
What can we say about the gradients on  $w$ ?

Always all positive or all negative :( (this is also why you want zero-mean data!)

- zero-centered가 아니기 때문에 부호가 하나로 정해져서 수렴 속도가 느려지게 된다.

## Activation Functions

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



Sigmoid

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

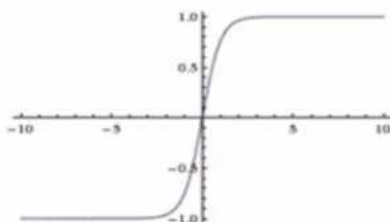
3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3.  $\exp()$  is a bit compute expensive

3. 마지막 문제는  $\exp()$  연산이 매우 비싼 연산이라는 점이다.

2.  $\tanh(x)$

## Activation Functions



$\tanh(x)$

- Squashes numbers to range  $[-1, 1]$
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

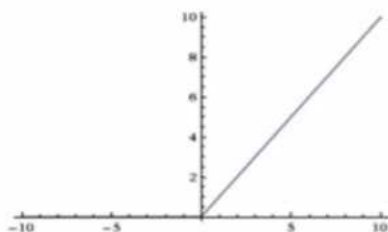
- sigmoid의 문제점을 해결하기 위해 zero-centered된 함수를 만들었다.



- 하지만 이것 또한 gradient vanish 문제를 피할 수 없었다.

3.  $\text{ReLU} = \max(0, x)$

## Activation Functions

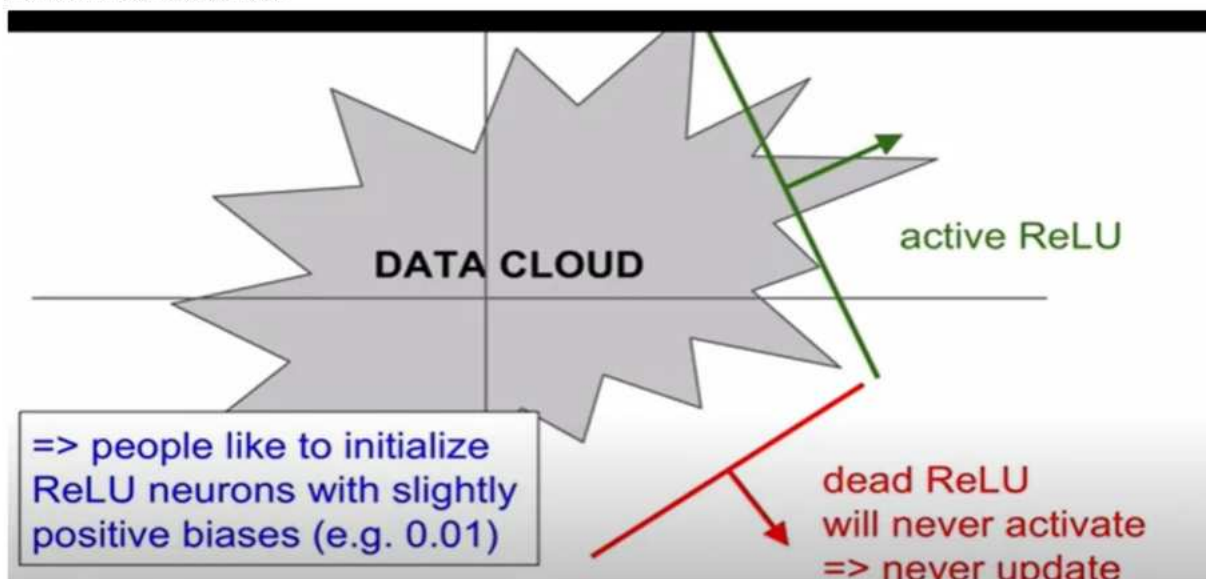


- Computes  $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

### ReLU (Rectified Linear Unit)

[Krizhevsky et al., 2012]

- 연산이 매우 효율적이고 수렴 속도가 빠르다.
- 현재 기본적인 activation function으로 사용되고 있다.

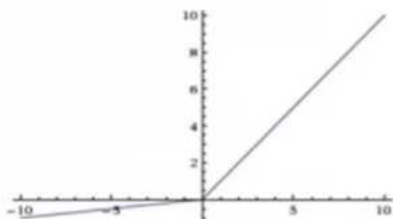


- 음수인 부분은 gradient가 0이기 때문에 update가 되지 않는 문제를 가지고 있다.
- learning\_rate가 너무 큰 경우에도 발생한다.
- 그래서 사람들이 약간 작은 양수로 초기화 하는 것을 좋아한다.

4.  $\text{Leaky ReLU} = \max(0.01x, x)$

## Activation Functions

[Mass et al., 2013]  
[He et al., 2015]



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not "die".

### Leaky ReLU

$$f(x) = \max(0.01x, x)$$

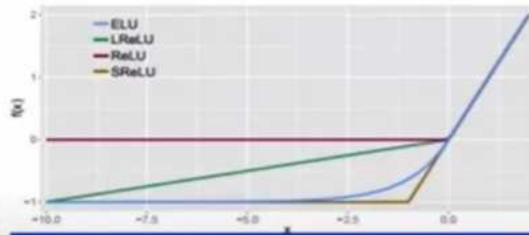
- 기울기가 죽지도 않는 좋은 함수로 보인다.
- 하지만 무조건 ReLU보다 낫다고는 말할 수 없다.

## 5. Exponential Linear Units(ELU)

### Activation Functions

[Clevert et al., 2015]

#### Exponential Linear Units (ELU)



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

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs
- Computation requires  $\exp()$

- ReLU의 장점을 가지면서 zero mean을 가지지만 exp 연산이 비싸다는 단점이 있다.

## 6. Maxout Neuron

### Maxout “Neuron”

[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters/neuron :(

- ReLU와 Leaky ReLU를 일반화한 함수라고 볼 수 있지만 parameter와 neuron이 2배로 든다는 단점이 있다.

### TLDR: In practice:

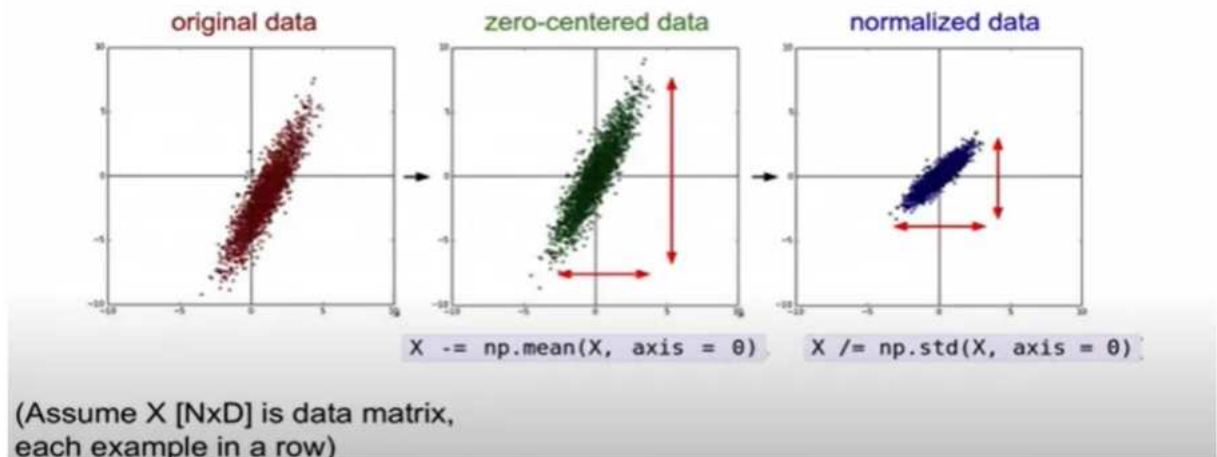
- Use **ReLU**. Be careful with your learning rates
- Try out **Leaky ReLU** / **Maxout** / **ELU**
- Try out **tanh** but don't expect much
- **Don't use sigmoid**



- 따라서 일반적으로는 ReLU를 사용하지만 Leaky ReLU/Maxout/ELU를 시도해 볼 수 있다.
- 하지만 sigmoid는 사용하지 마라.

## 02\_Data Preprocess

### Step 1: Preprocess the data



- zero-centered + normalized
- 이미지에서는 값이 [0~255] 사이의 값을 가지기 때문에 굳이 normalized를 하지 않는다.

### TLDR: In practice for Images: center only

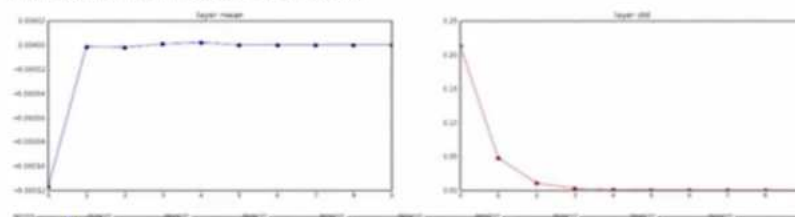
e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)  
(mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)  
(mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening

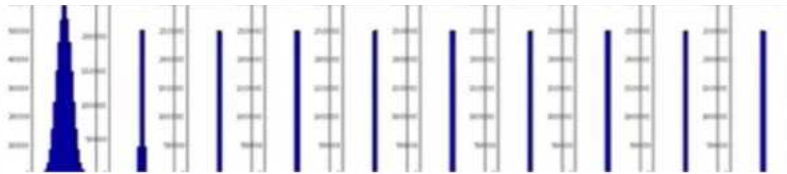
## 03\_Weight Initialization

Input layer had mean 0.000927 and std 0.998388  
hidden layer 1 had mean -0.000117 and std 0.213881  
hidden layer 2 had mean -0.000001 and std 0.047551  
hidden layer 3 had mean -0.000002 and std 0.016038  
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 0.000020  
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



All activations become zero!

Q: think about the backward pass. What do the gradients look like?



$$dW1 = X * dW2$$

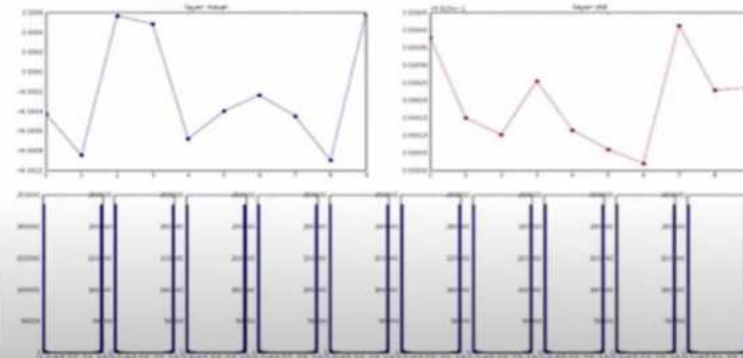
Hint: think about backward pass for a  $W \times X$  gate.

- 랜덤으로 값을 초기화 한 후 학습을 시켜보니 activation이 0으로 가는 문제가 발생한다.
- 그 결과 gradient vanishing이 발생한다.

```
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 0.901879  
 hidden layer 2 had mean -0.000449 and std 0.901649  
 hidden layer 3 had mean 0.000566 and std 0.901601  
 hidden layer 4 had mean 0.000483 and std 0.901755  
 hidden layer 5 had mean -0.000682 and std 0.901614  
 hidden layer 6 had mean -0.000401 and std 0.901500  
 hidden layer 7 had mean -0.000237 and std 0.901520  
 hidden layer 8 had mean -0.000440 and std 0.901913  
 hidden layer 9 had mean -0.000999 and std 0.901726  
 hidden layer 10 had mean 0.000504 and std 0.901736

\*1.0 instead of \*0.01



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

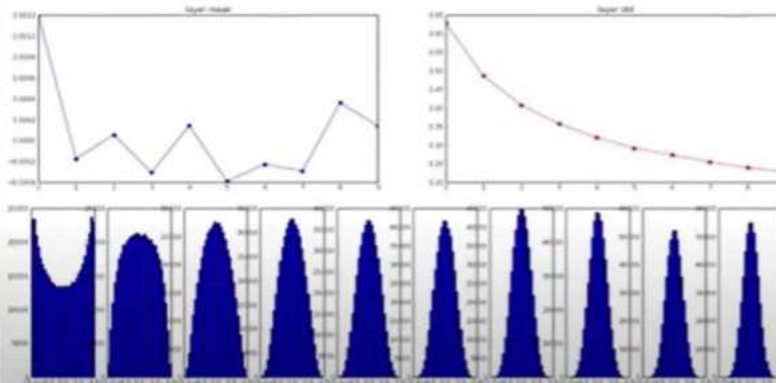
- 1로 초기화 하게 되면 값이 튀어서 -1 또는 1로 포화된다.

## 1. Xavier Initialization

Input layer had mean 0.001800 and std 1.001311  
 hidden layer 1 had mean 0.001198 and std 0.627953  
 hidden layer 2 had mean -0.000175 and std 0.406051  
 hidden layer 3 had mean 0.000055 and std 0.407723  
 hidden layer 4 had mean -0.000350 and std 0.357108  
 hidden layer 5 had mean 0.000142 and std 0.320917  
 hidden layer 6 had mean -0.000389 and std 0.292116  
 hidden layer 7 had mean -0.000228 and std 0.273307  
 hidden layer 8 had mean -0.000291 and std 0.254935  
 hidden layer 9 had mean 0.000301 and std 0.239266  
 hidden layer 10 had mean 0.000139 and std 0.220008

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

"Xavier initialization"  
 [Glorot et al., 2010]



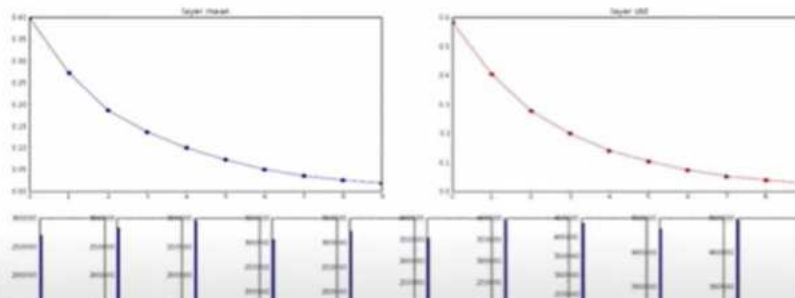
Reasonable initialization.  
 (Mathematical derivation assumes linear activations)

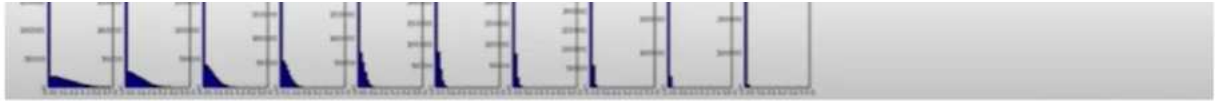
- Xavier initialization 방식은 매우 합리적인 방식으로 linear activation에 많이 쓰는 초기화 방식이다.

Input layer had mean 0.000501 and std 0.999444  
 hidden layer 1 had mean 0.390623 and std 0.502273  
 hidden layer 2 had mean 0.272352 and std 0.403795  
 hidden layer 3 had mean 0.166076 and std 0.276912  
 hidden layer 4 had mean 0.130442 and std 0.190085  
 hidden layer 5 had mean 0.099560 and std 0.140299  
 hidden layer 6 had mean 0.072234 and std 0.103200  
 hidden layer 7 had mean 0.049775 and std 0.072748  
 hidden layer 8 had mean 0.035130 and std 0.051372  
 hidden layer 9 had mean 0.025404 and std 0.030583  
 hidden layer 10 had mean 0.018408 and std 0.020076

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

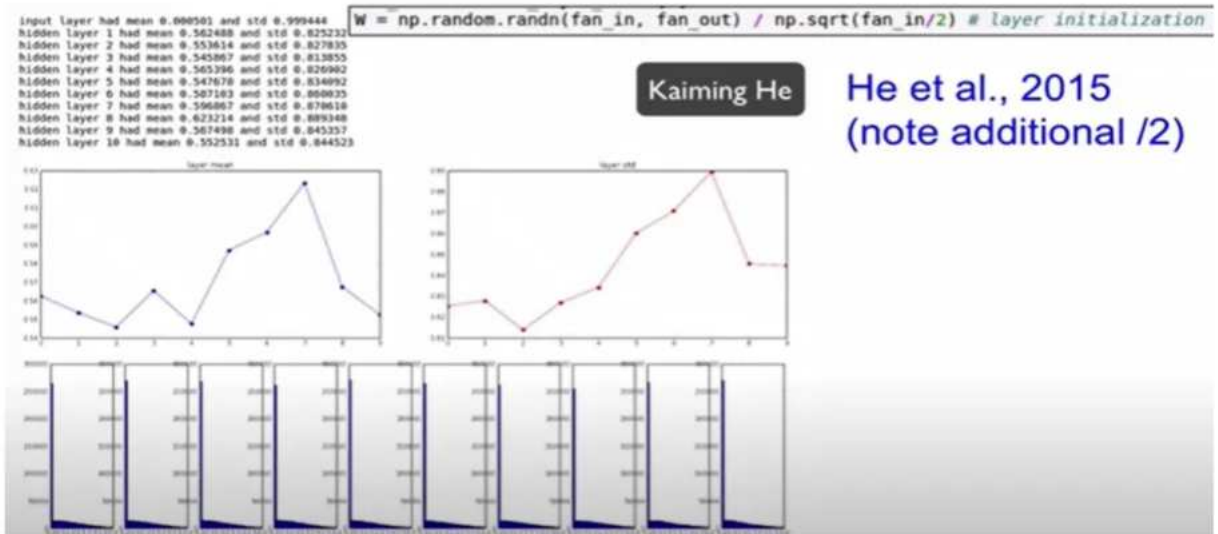
but when using the ReLU nonlinearity it breaks.





- 하지만 ReLU같은 nonlinearity 함수에는 잘 적용되지 못했다.

## 2. He Initialization



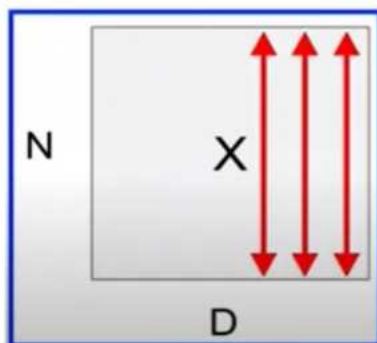
- ReLU에는 He initialization을 적용하면 잘 동작하는 것을 확인할 수 있다.

## 04\_Batch Normalization

### Batch Normalization

[Ioffe and Szegedy, 2015]

“you want unit gaussian activations?  
just make them so.”



1. compute the empirical mean and variance independently for each dimension.

2. Normalize

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

### Batch Normalization

[Ioffe and Szegedy, 2015]

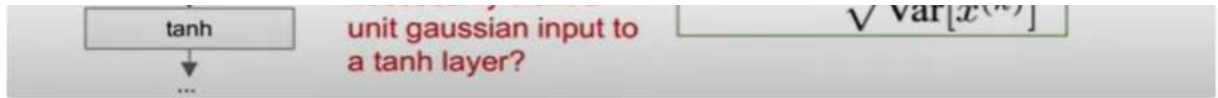


Usually inserted after Fully Connected / (or Convolutional, as we'll see soon) layers, and before nonlinearity.

Problem: do we necessarily want a

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





- FC - BN - activation 순서로 적용한다.

## Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots x_m\}$ ;  
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

- 자세한 과정은 과제를 참고하도록 하자.

## Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots x_m\}$ ;  
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:**

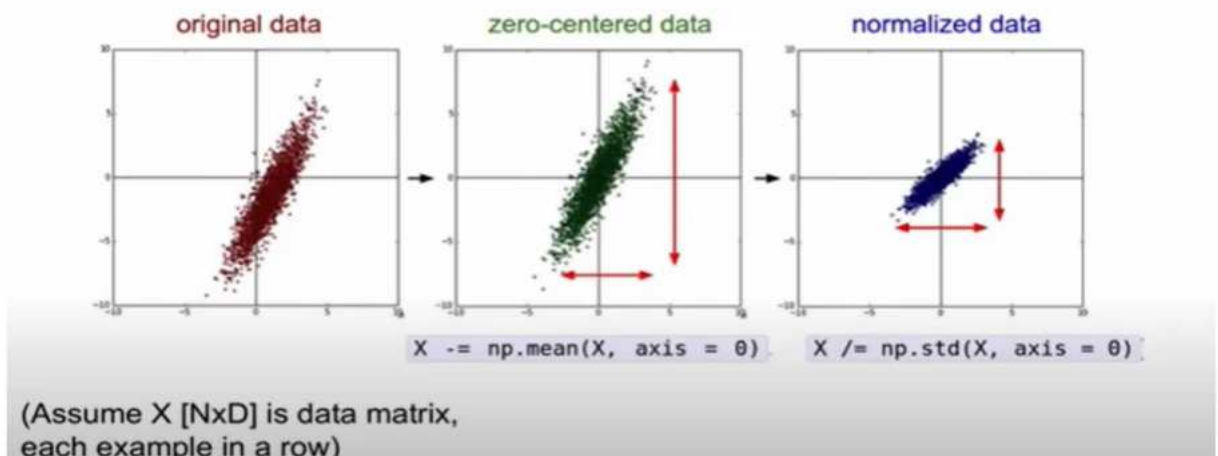
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

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

- test를 할 때는 미리 mean과 variance를 저장해둔 것을 활용한다.

### 05\_Babysitting the learning process

## Step 1: Preprocess the data

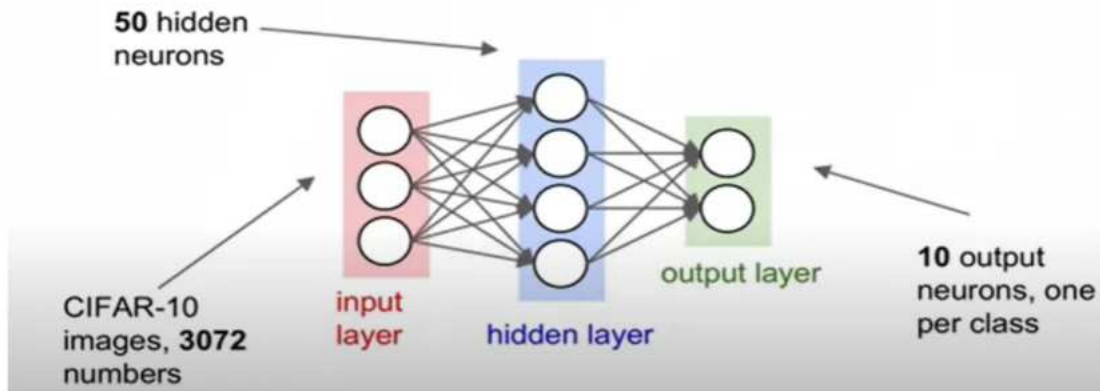


1. zero centered



## Step 2: Choose the architecture:

say we start with one hidden layer of 50 neurons:



2. Choose the architecture

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)
```

| Epoch | Cost     | Train Accuracy | Val Accuracy | Learning Rate |
|-------|----------|----------------|--------------|---------------|
| 1     | 2.382603 | 0.400000       | 0.400000     | 1.000000e-03  |
| 2     | 2.382258 | 0.450000       | 0.450000     | 1.000000e-03  |
| 3     | 2.381849 | 0.600000       | 0.600000     | 1.000000e-03  |
| 4     | 2.381196 | 0.650000       | 0.650000     | 1.000000e-03  |
| 5     | 2.380044 | 0.650000       | 0.650000     | 1.000000e-03  |
| 6     | 2.297864 | 0.550000       | 0.550000     | 1.000000e-03  |
| 7     | 2.293595 | 0.600000       | 0.600000     | 1.000000e-03  |
| 8     | 2.285096 | 0.550000       | 0.550000     | 1.000000e-03  |
| 9     | 2.268094 | 0.550000       | 0.550000     | 1.000000e-03  |
| 10    | 2.234787 | 0.500000       | 0.500000     | 1.000000e-03  |
| 11    | 2.173187 | 0.500000       | 0.500000     | 1.000000e-03  |
| 12    | 2.076862 | 0.500000       | 0.500000     | 1.000000e-03  |
| 13    | 1.974090 | 0.400000       | 0.400000     | 1.000000e-03  |
| 14    | 1.895885 | 0.400000       | 0.400000     | 1.000000e-03  |
| 15    | 1.820876 | 0.450000       | 0.450000     | 1.000000e-03  |
| 16    | 1.737438 | 0.450000       | 0.450000     | 1.000000e-03  |
| 17    | 1.642356 | 0.500000       | 0.500000     | 1.000000e-03  |
| 18    | 1.535239 | 0.600000       | 0.600000     | 1.000000e-03  |
| 19    | 1.421527 | 0.600000       | 0.600000     | 1.000000e-03  |
| 195   | 0.002694 | 1.000000       | 1.000000     | 1.000000e-03  |
| 196   | 0.002674 | 1.000000       | 1.000000     | 1.000000e-03  |
| 197   | 0.002655 | 1.000000       | 1.000000     | 1.000000e-03  |
| 198   | 0.002635 | 1.000000       | 1.000000     | 1.000000e-03  |
| 199   | 0.002617 | 1.000000       | 1.000000     | 1.000000e-03  |
| 200   | 0.002597 | 1.000000       | 1.000000     | 1.000000e-03  |

finished optimization. Best validation accuracy: 1.000000

- 작은 데이터를 위한 후에 학습을 시키면 무조건 overfitting이 나와야 한다.
- 이것이 모델이 잘 동작한다는 의미이다.

Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down:  
learning rate too low

```
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)
```

| Epoch | Cost     | Train Accuracy | Val Accuracy | Learning Rate |
|-------|----------|----------------|--------------|---------------|
| 1     | 2.382576 | 0.000000       | 0.103000     | 1.000000e-06  |
| 2     | 2.382582 | 0.121000       | 0.124000     | 1.000000e-06  |
| 3     | 2.382558 | 0.119000       | 0.138000     | 1.000000e-06  |
| 4     | 2.382519 | 0.127000       | 0.151000     | 1.000000e-06  |
| 5     | 2.382517 | 0.158000       | 0.171000     | 1.000000e-06  |
| 6     | 2.382518 | 0.179000       | 0.172000     | 1.000000e-06  |
| 7     | 2.382466 | 0.180000       | 0.176000     | 1.000000e-06  |
| 8     | 2.382452 | 0.175000       | 0.185000     | 1.000000e-06  |
| 9     | 2.382459 | 0.206000       | 0.192000     | 1.000000e-06  |
| 10    | 2.382420 | 0.190000       | 0.192000     | 1.000000e-06  |

finished optimization. Best validation accuracy: 0.192000

Loss barely changing: Learning rate is probably too low

- Learning\_rate가 너무 작으면 loss가 줄어들지 않는다.

Lets try to train now...

I like to start with small

```
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-5, verbose=True)
```

regularization and find learning rate that makes the loss go down.

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero encountered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value encountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

**loss not going down:**  
learning rate too low  
**loss exploding:**  
learning rate too high

cost: NaN almost always means high learning rate...

- Learning\_rate가 너무 크면 loss가 exploding 해버린다.

## 06\_Hyperparameter Optimization

### Cross-validation strategy

I like to do **coarse** -> **fine** cross-validation in stages

**First stage:** only a few epochs to get rough idea of what params work

**Second stage:** longer running time, finer search

... (repeat as necessary)

Tip for detecting explosions in the solver:

If the cost is ever  $> 3 \times$  original cost, break out early

- epoch을 작게 한 후 나중에 세부적으로 조정한다.

Now run finer search...

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

adjust range

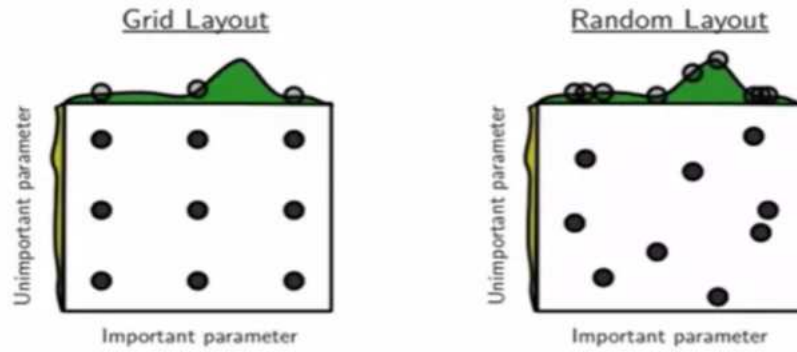
```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

```
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: 0.490000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484366e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.506261e-04, reg: 2.312605e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.252954e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979160e-04, reg: 1.010609e-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: 2.267807e-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: 1.433095e-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: 3.033781e-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 for a 2-layer neural net with 50 hidden neurons.

- 값을 점점 좁혀가면서 진행.
- boundary에 가까운 값이 최적이라면 범위를 다시 넓혀서 진행.

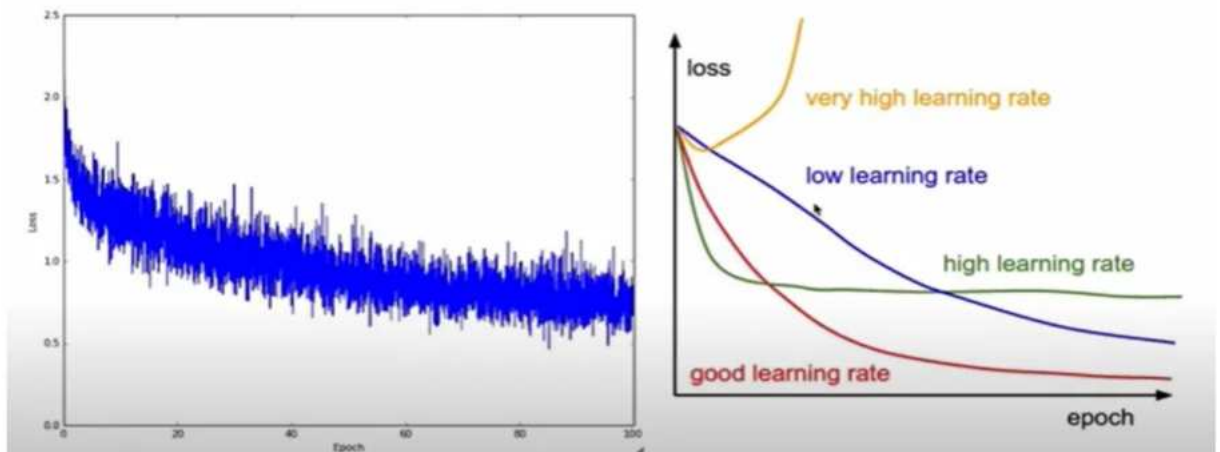
# Random Search vs. Grid Search



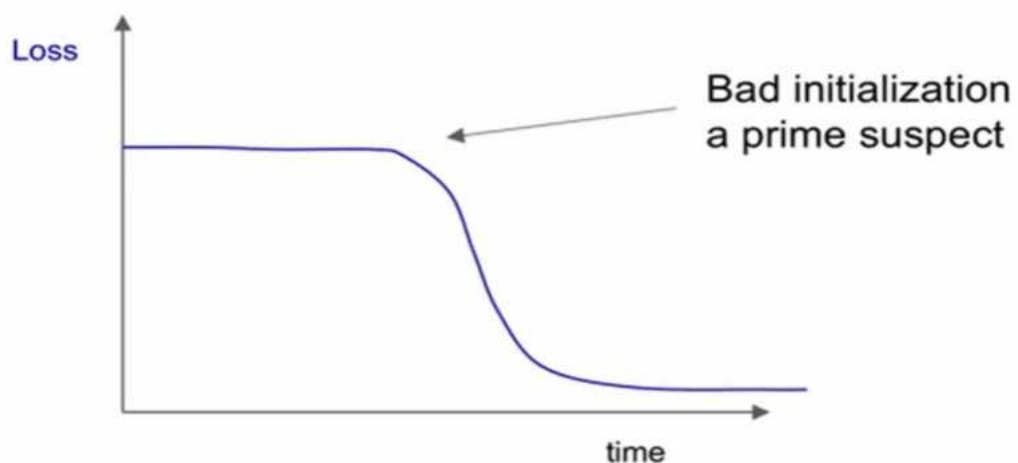
*Random Search for Hyper-Parameter Optimization*  
Bergstra and Bengio, 2012

- grid 등간격으로 진행하면 우리가 찾고 싶은 최적의 parameter를 찾지 못할 수도 있다.
- 결국 최적화를 하기 위해서는 random하게 접근해야 한다.

## Monitor and visualize the loss curve



- Loss curve를 통해서 좋은 learning\_rate를 찾는다.



- Loss가 정체되는 것은 initialization이 잘못된 것을 의미한다.