

## 1. Neural Network Training

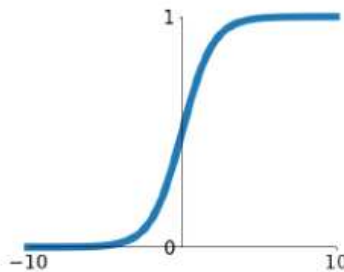
- 모든 data를 가지고 gradient descent Algorithm에 적용하면  
계산량이 많아 SGD(Stochastic Gradient Descent) Algorithm을 이용  
Sample을 뽑아내 Gradient Descent Algorithm을 사용하는 방법

## 2. Activation Function

- Sigmoid function

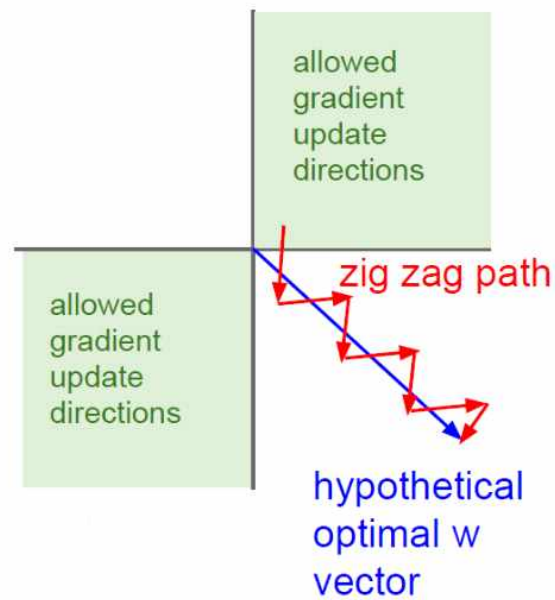
### Sigmoid

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



- \* 출력이 (0,1) 사이의 값이 나오도록 하는 선형 함수
- \* 단점:
  - Saturated neurons가 Gradient값을 0으로 만든다.
  - 원점 중심이 아니다.
  - 지수함수가 계산량이 많다.
- \* Gradient의 값이 0이 되는 것이 왜 단점?  
Chain Rule에서 Global gradient값, 즉 결과값이 0이 되면  
local gradient 값도 0이 돼서  
Input에 있는 gradient 값을 구할 수 없음

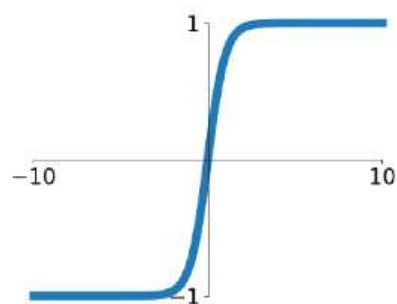
\* 원점 중심이 아닌 것은 왜 단점?



$w$ 의 경우 제 1사분면과 제 3사분면으로 update가 되기 때문에 원하는 방향으로 update 하기 힘들다.

-  $\tanh(x)$

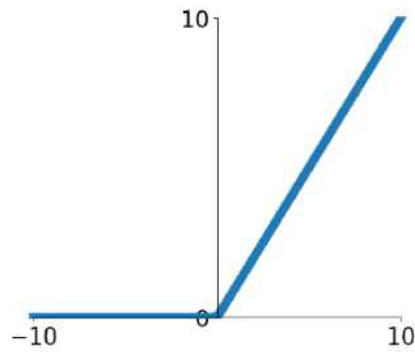
\* Sigmoid가 원점 중심이 아닌 것을 보완하기 위해 나온 function



**$\tanh(x)$**

\* 여전히 saturated한 뉴런일 때, gradient값이 0

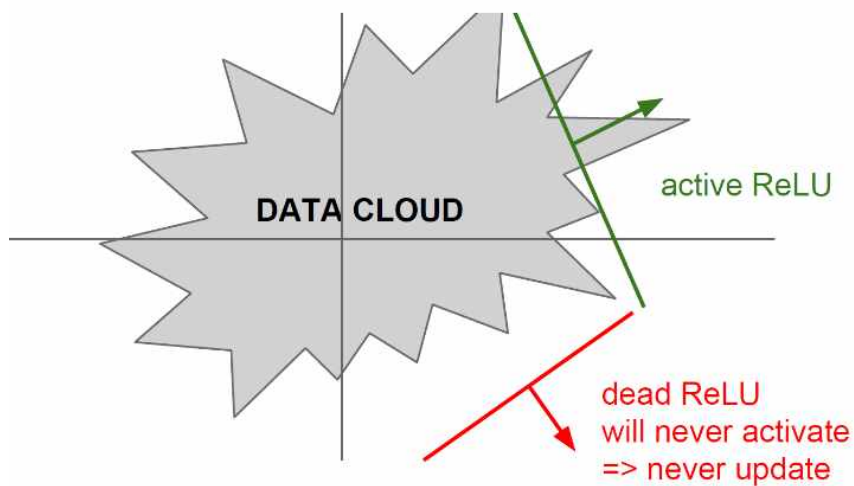
- ReLU



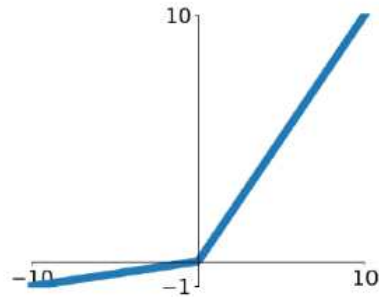
## ReLU (Rectified Linear Unit)

- \* (+) 영역에서 saturate하지 않고,  
계산 속도도 element-wise 연산이어서  
sigmoid/tanh보다 훨씬 빠름

- \* 단점 :  
(-)의 값은 0으로 만들어 버려 Data의 절반만 activate하게 만들



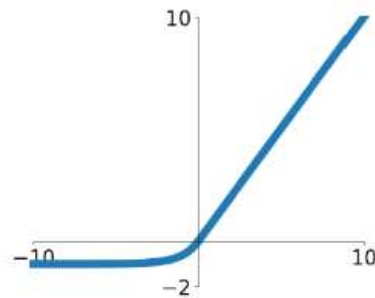
- Leaky ReLU



## Leaky ReLU

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

- Exponential Linear Unit



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

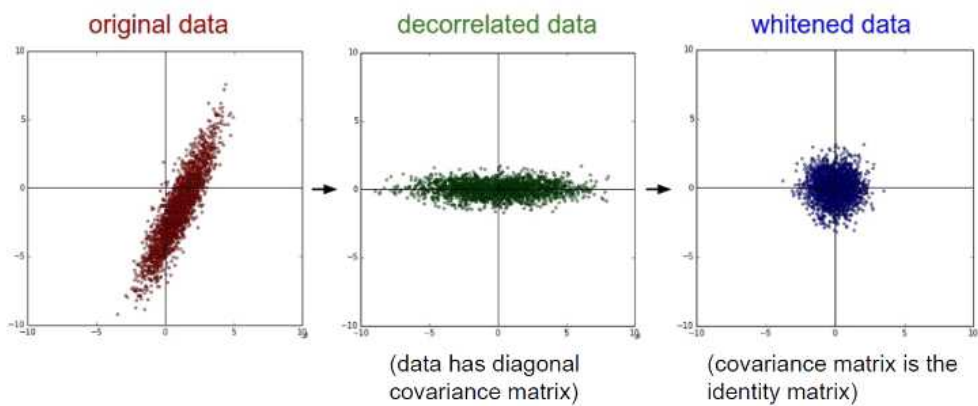
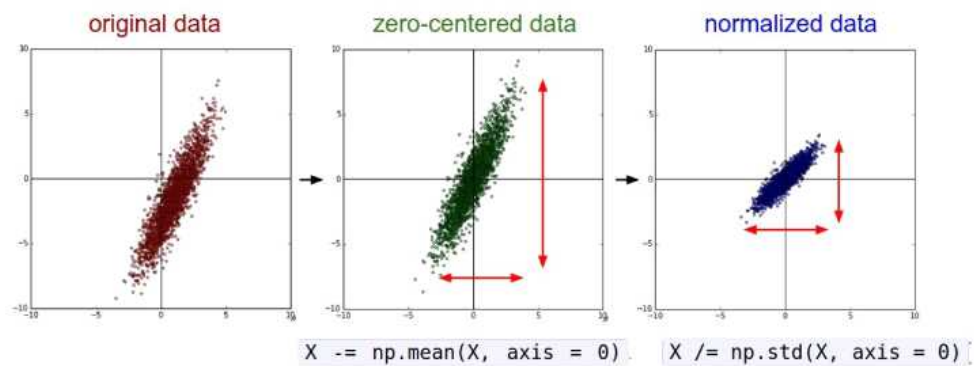
- Maxout

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

\* parameter가 기존 function보다 2배 있어야 함

### 3. Data processing

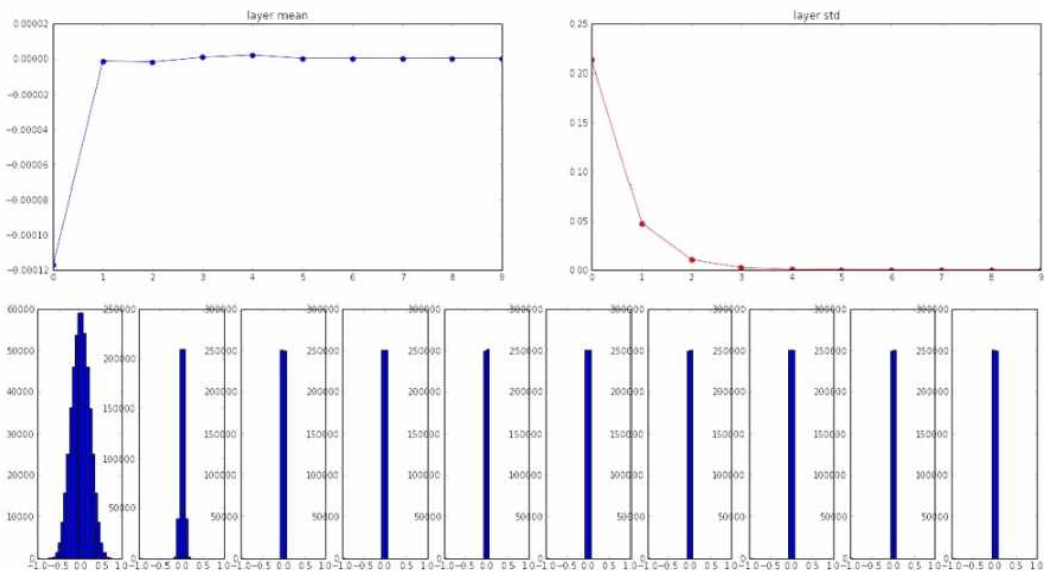
- 종류 : Zero-centered, Normalized, PCA, Whitening
- Zero-centered, Normalized :  
모든 차원이 동일한 범위에서 전부 동등한 기여를 할 수 있도록 하는 것  
이미지는 Zero-centered 과정만 거침
- PCA, Whitening:  
더 낮은 차원으로 projection하는 느낌  
이미지 처리에서는 이 전처리 과정은 거치지 않음



## 4. Weight Initialization

- 초기값을 0으로 하면 모든 뉴런은 동일한 일을 하게 돼  
모든 gradient의 값이 같게 될 것 -> 의미 x
- 작은 random한 수로 초기화:  
초기 Weight는 표준정규분포에서 sampling  
얕은 network에서는 잘 작동하지만 network가 깊어질 경우 문제가 생김  
network가 깊으면 깊을수록 weight의 값이 너무 작아 0으로 수렴하기 때문

```
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
```

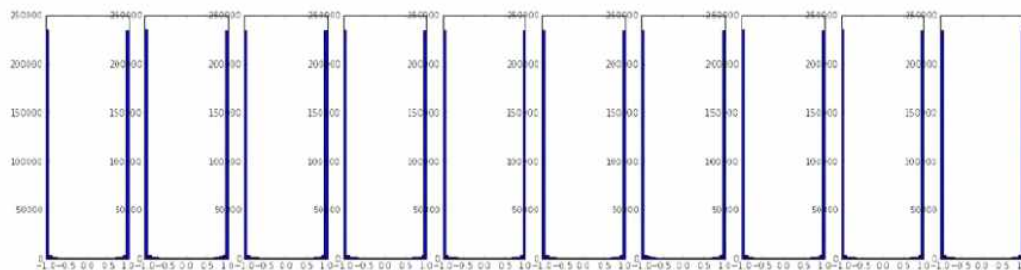
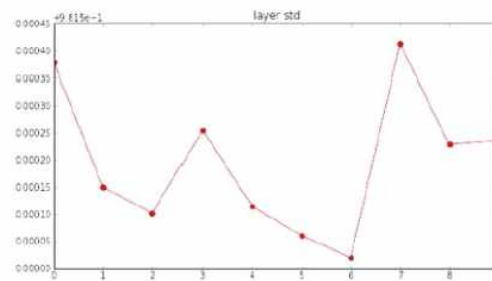
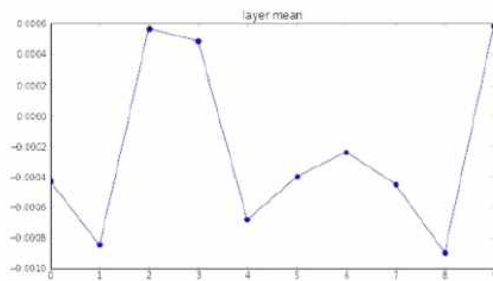


- 표준 편차를 키운다면?  
activation value의 값이 극단적인 값을 가지게 되고,  
gradient의 값이 모두 0으로 수렴

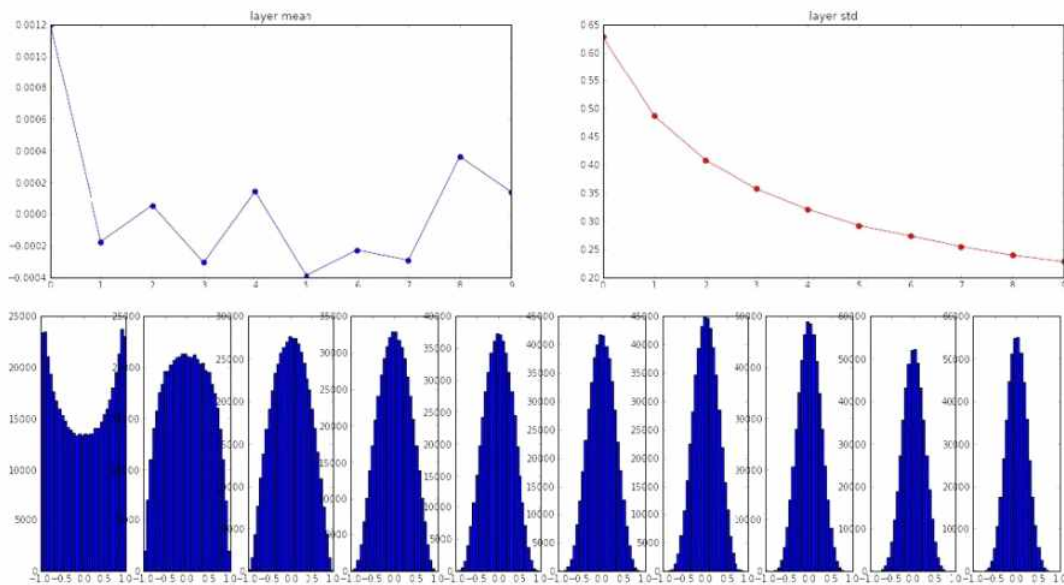
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000436 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 -0.000491 and std 0.981566
hidden layer 7 had mean -0.000237 and std 0.981526
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



- 이런 초기값 문제에 대해서 'Xavier initialization'이라는 논문이 제시
- activation function이 linear하다는 가정하에 다음과 같은 식을 사용하여 weight의 값을 초기화하고, 입/출력의 분산을 맞춰줄 수 있게 됨



- activation function이 ReLU인 경우, 출력의 분산이 반토막 나기 때문에 이 식이 성립하지 않음
- activation function이 ReLU인 경우에는 He Initialization을 사용



## 5. Weight Initialization

- Batch에서 계산한 mean과 variance를 이용하여 정규화를 해주는 과정을 Model에 추가해주는 것

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

- 각 layer에서 Weight가 지속적으로 곱해져서 생기는 Bad Scaling의 효과를 상쇄
- unit gaussian으로 바꿔주고 분산과 평균을 이용해 Normalized를 좀 더 유연하게 할 수 있음

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

<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

- Regularization의 역할 (Overfitting 방지)
- weight의 초기화 의존성에 대한 문제 줄임
- Test할 때 minibatch의 평균과 표준편차를 구할 수 없어 Training하면서 구한 평균의 이동평균을 이용해 고정된 Mean과 Std를 사용
- 학습 속도 개선 가능

## 6. Babysitting the Learning Process

- 전처리 과정 : 이미지의 경우 zero-centered
- 사용할 neural architecture를 정해줌
- loss 가 적절하게 나오는지 확인

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

- 규제 값을 올렸을 때 loss가 증가, network가 잘 동작하는지 확인

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, 1e3) # crank up regularization  
print loss  
3.06859716482
```

loss went up, good. (sanity check)

- 작은 데이터 셋을 이용해서 훈련
- Data 수가 작아 overfitting이 발생, train acc가 100% overfitting이 나온다는 것은 모델이 제대로 동작하고 있다는 의미

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_train, y_train,
                                  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 2.302250, 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 2.300644, 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 1.974096, 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 1.820076, 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 199 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03  
 Finished epoch 200 / 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

- hyper parameter 값들(regularization, learning rate)을 설정

Lets try to train now...

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)
```

Finished epoch 1 / 10: cost 2.302576, train: 0.800000, 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: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

- $lr = 1e6$ 로 설정하니 값이 너무 커서 cost 값이 Nan

Lets try to train now...

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

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

```
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=1e6, verbose=True)

/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: 0.891800, val 0.887000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.895000, val 0.887000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.887000, lr 1.000000e+06
```

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

- $lr = 3e-3$ 로 수정해도 inf, 몇 번의 실험을 통해  $1e-3 \sim 1e-5$  값이 적절한 값이라는 것을 추측

Lets try to train now...

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

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

```
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=3e-3, verbose=True)

Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

3e-3 is still too high. Cost explodes....

=> Rough range for learning rate we  
should be cross-validating is  
somewhere  $[1e-3 \dots 1e-5]$

## 7. Hyperparameter Optimization

- 종류 : Grid Search, Random Search
- Grid Search :
  - 탐색의 대상이 되는 특정 구간 내의 후보 hyperparameter 값들을 일정한 간격을 두고 선정
  - 각각에 대하여 측정한 성능 결과를 기록한 후
  - 가장 높은 성능을 발휘했던 hyperparameter 값을 선정하는 방법
- Random Search :
  - Grid Search와 큰 맥락은 유사
  - 탐색 대상 구간 내의 후보 hyperparameter 값들을 random sampling으로 선정하는 것이 차이
  - Grid Search에 비해 불필요한 반복 수행 횟수를 대폭 줄임
  - 정해진 간격(grid) 사이에 위치한 값들에 대해서도 확률적으로 탐색이 가능
  - 최적 hyperparameter 값을 더 빨리 찾을 수 있음
  - Random Search를 더 많이 사용

### Random Search vs. Grid Search

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

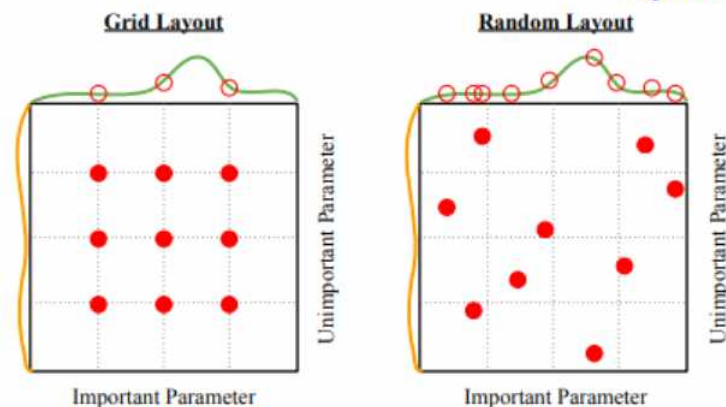


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017



- 과정 요약 :

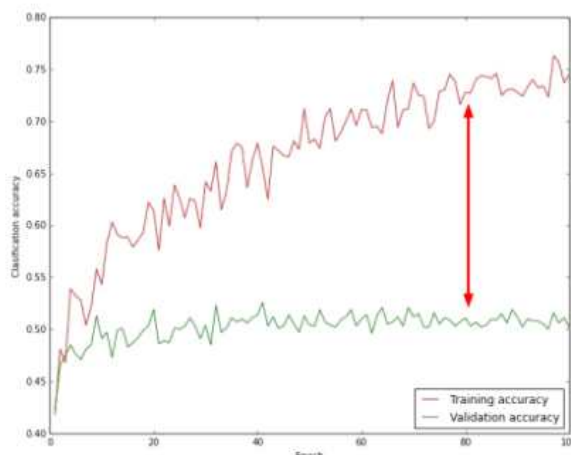
- \* Hyperparameter 값을 설정
- \* 위에서 정한 범위 내에서 파라미터 값을 무작위로 추출
- \* Validation Set을 이용하여 평가
- \* 특정 횟수를 반복하여 그 정확도를 보고 Hyperparameter 범위를 좁힘

- Hyperparameter를 정할 때 loss curve를 보고 이 hyperparameter가 적합한지 아닌지 평가를 하는 경우가 많음

- loss curve가 초기에 평평하다면 초기화가 잘못될 가능성이 큰 것

- training accuracy와 validation accuracy가 gap이 클 경우 overfitting이 된 가능성이 매우 높은 것

- gap이 없을 경우 model capacity를 늘리는 것을 고려  
training한 dataset이 너무 작은 경우일 수도 있음



big gap = overfitting  
=> increase regularization strength?

no gap  
=> increase model capacity?