

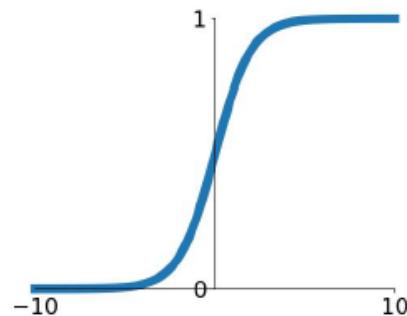
Training Neural Networks



Activation Functions

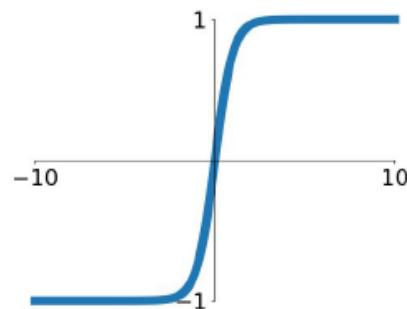
Sigmoid

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



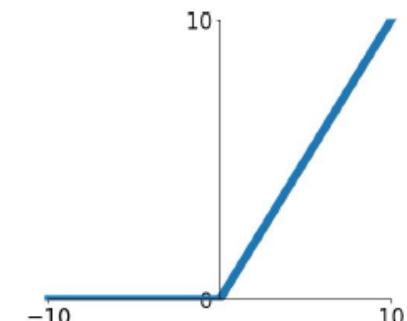
tanh

$$\tanh(x)$$



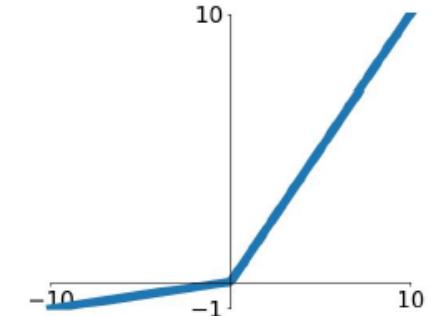
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

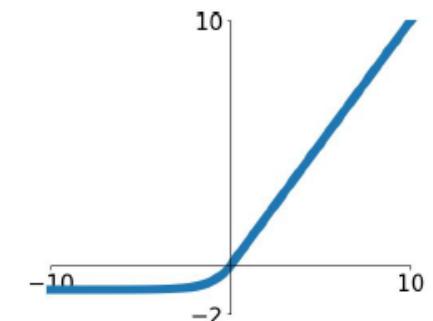


Maxout

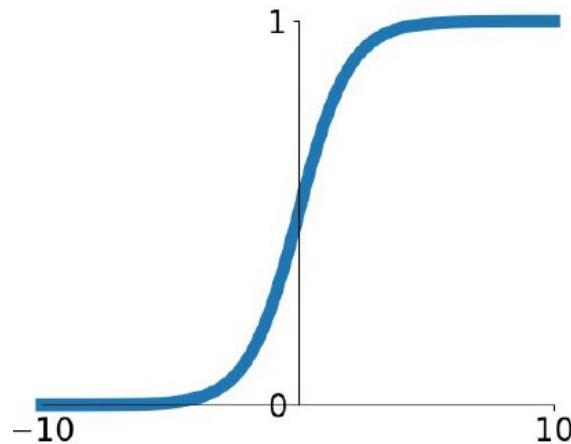
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid



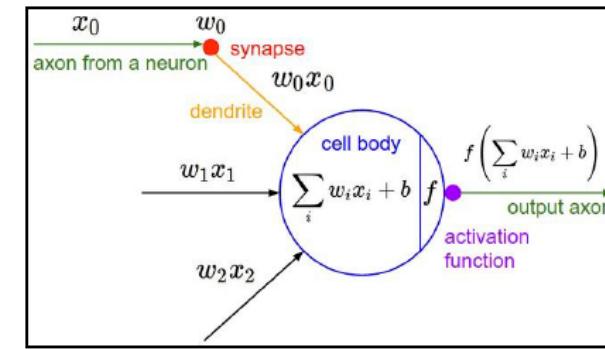
$$f(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron
- 3 Problems
 - Saturated neurons “kill” the gradients
 - Sigmoid outputs are not zero-centered
 - Exp() is a bit compute expensive

What happens when the input to a neuron is always positive

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 ?

We know that local gradient of sigmoid is always positive

We are assuming x is always positive

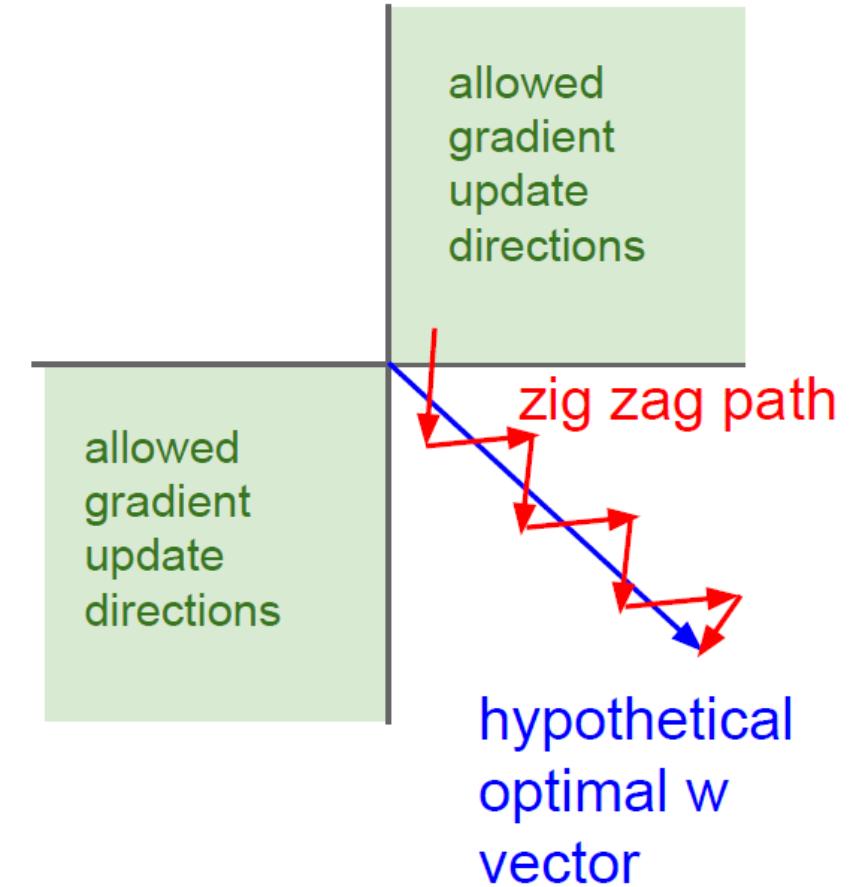
So!! Sign of gradient **for all w_i** is the same as the sign of upstream scalar gradient!

$$\frac{\partial L}{\partial w} = \sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x \times \text{upstream_gradient}$$

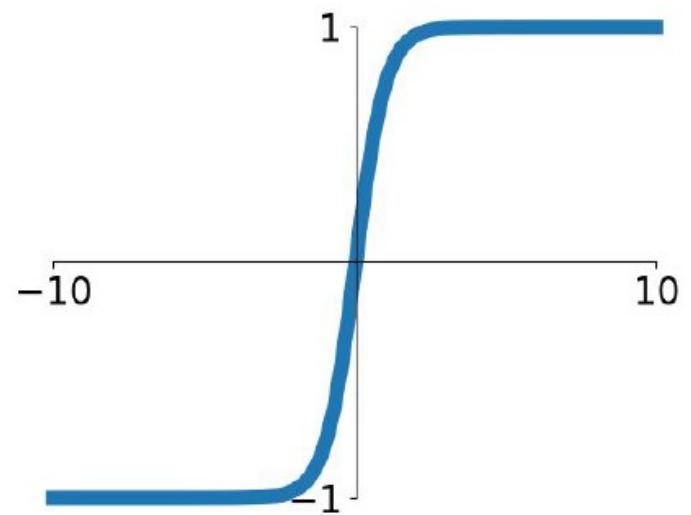
What happens when the input to a neuron is always positive

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

- What can we say about the gradients on w ?
→ Always all positive or all negative
→ This is also why you want zero-mean data



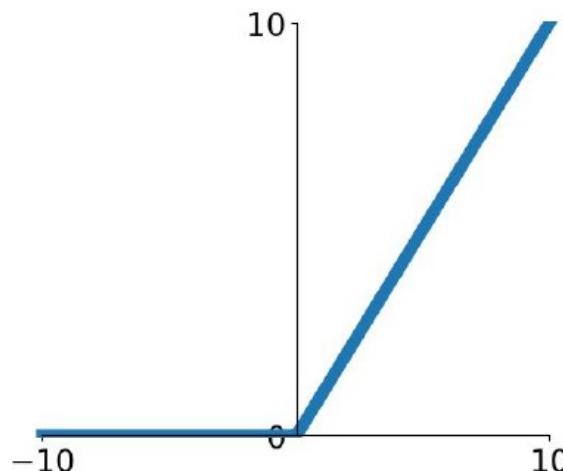
Tanh



$$f(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Squashes numbers to range $[-1, 1]$
- Zero centered
- Still kills gradient when saturated

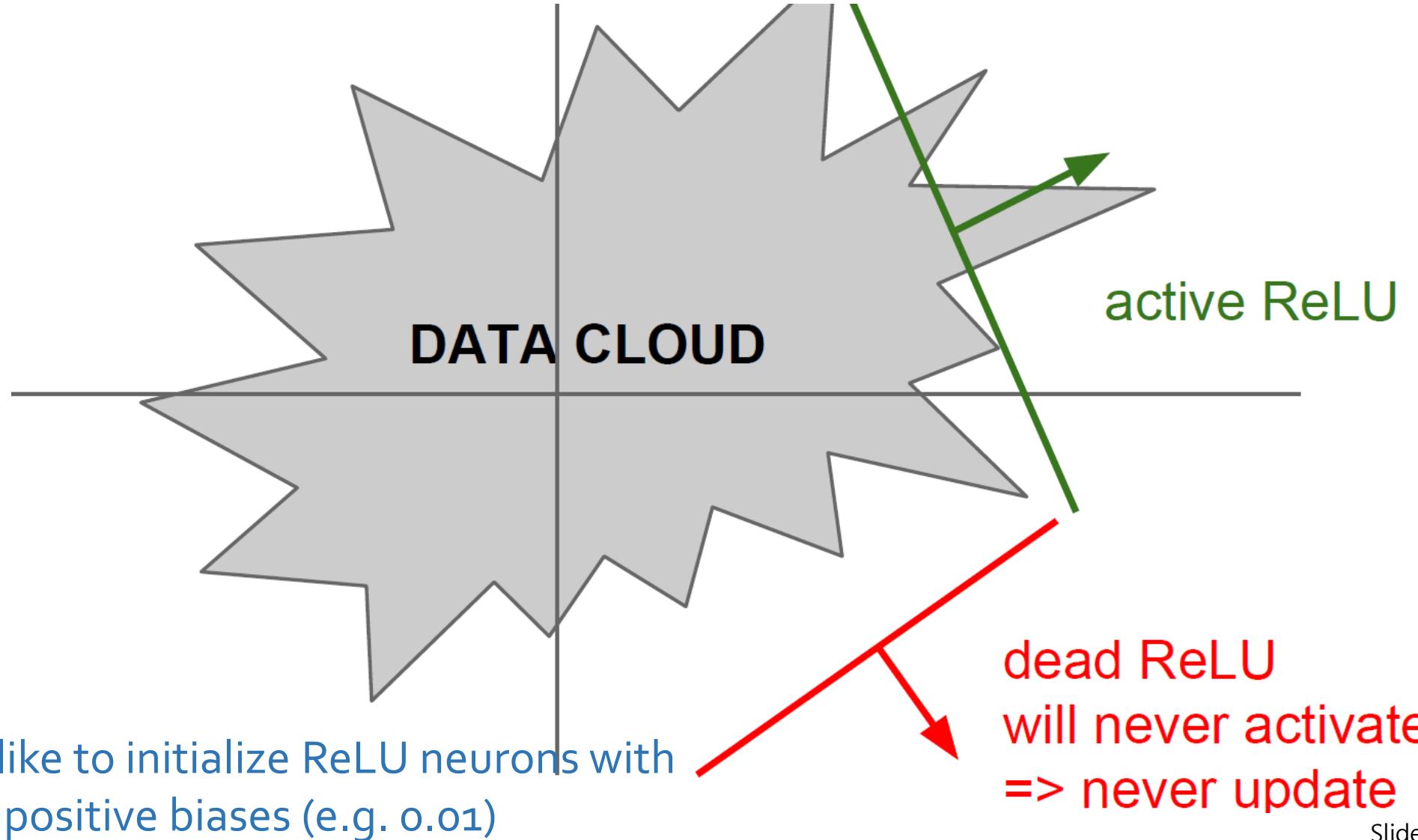
ReLU(Rectified Linear Unit)



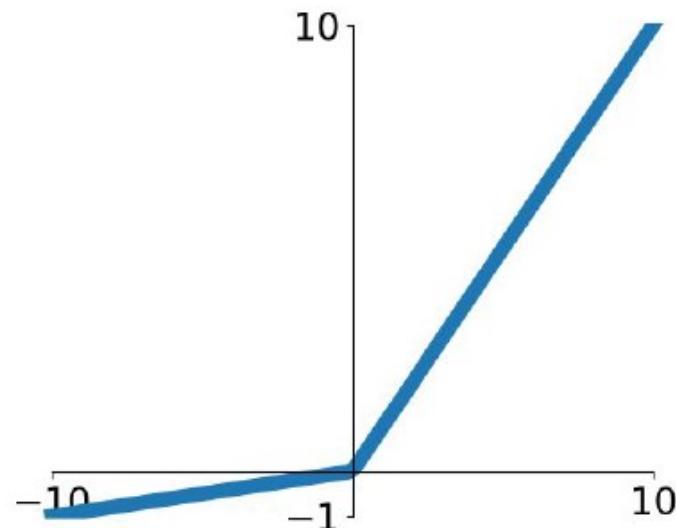
Computes $f(x) = \max(0, x)$

- Does not saturate
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice(e.g. 6x)
- Actually more biologically plausible than sigmoid
- Problems
 - Not zero-centered output
 - An annoyance – dead ReLU

Dead ReLU



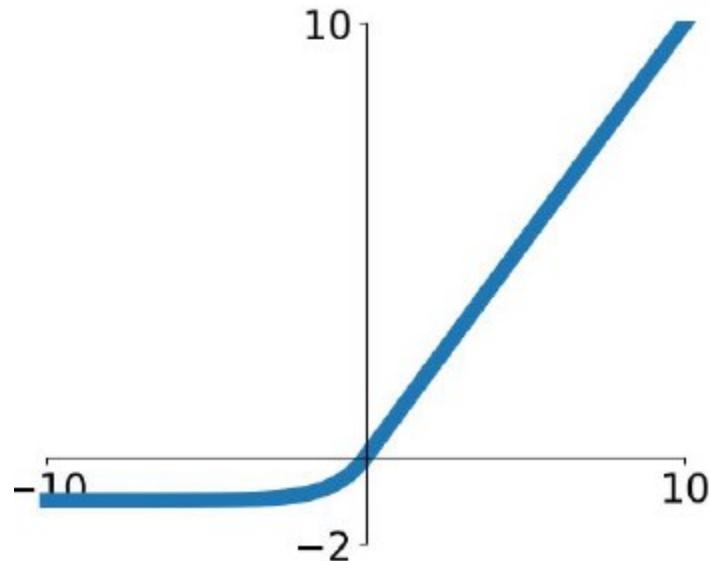
Leaky ReLU, PReLU(Parametric-)



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

- Does not saturate
- Computationally efficient
- Converge much faster than sigmoid/tanh
- Will not ‘die’

ELU(Exponential Linear Units)



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

- All benefits of ReLU
- Closer to zero mean output
- Computation requires $\exp()$

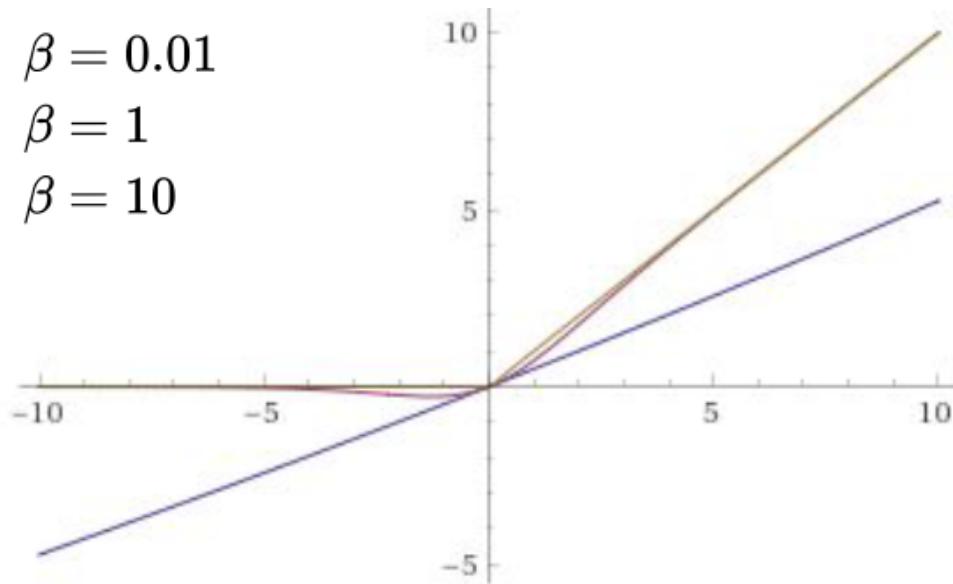
Maxout

- 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

Swish



$$f(x) = x\sigma(\beta x)$$

- They trained a neural network to generate and test out different non-linearities.
- Swish outperformed all other options for CIFAR-10 accuracy

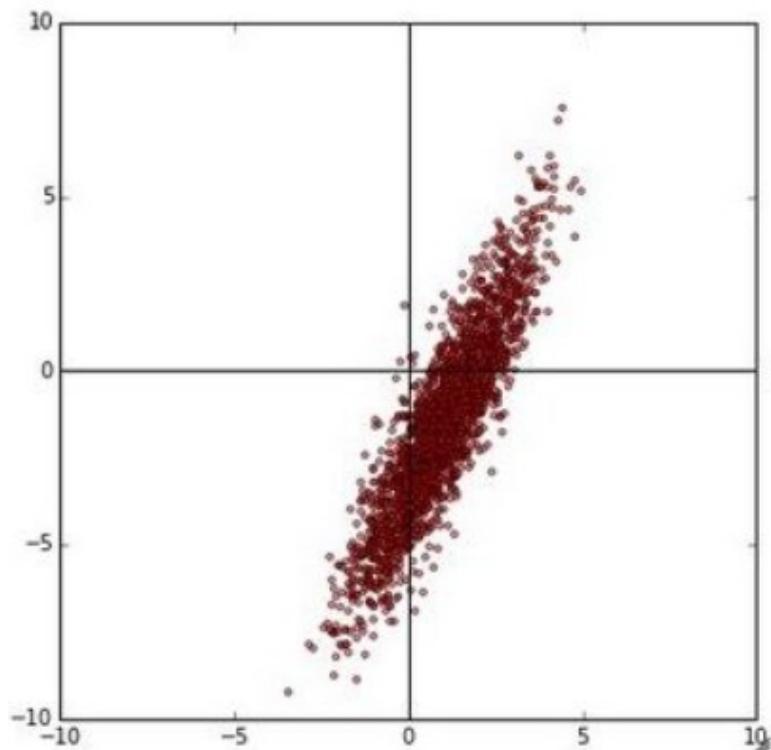
Other Activation Functions

- Softmax
- Softplus : $\ln(1 + e^x)$
- SELU: scaled ELU

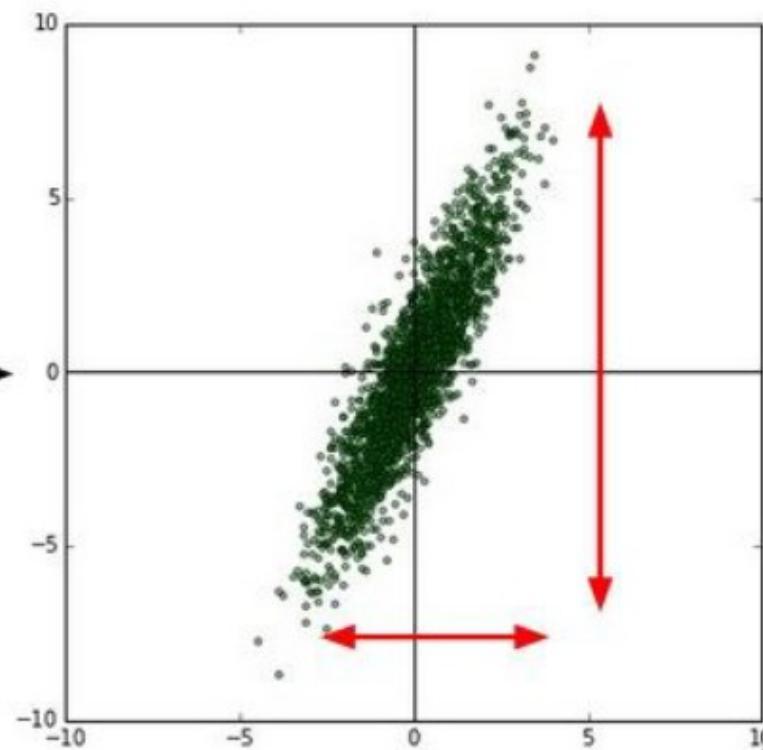
...

Data Preprocessing

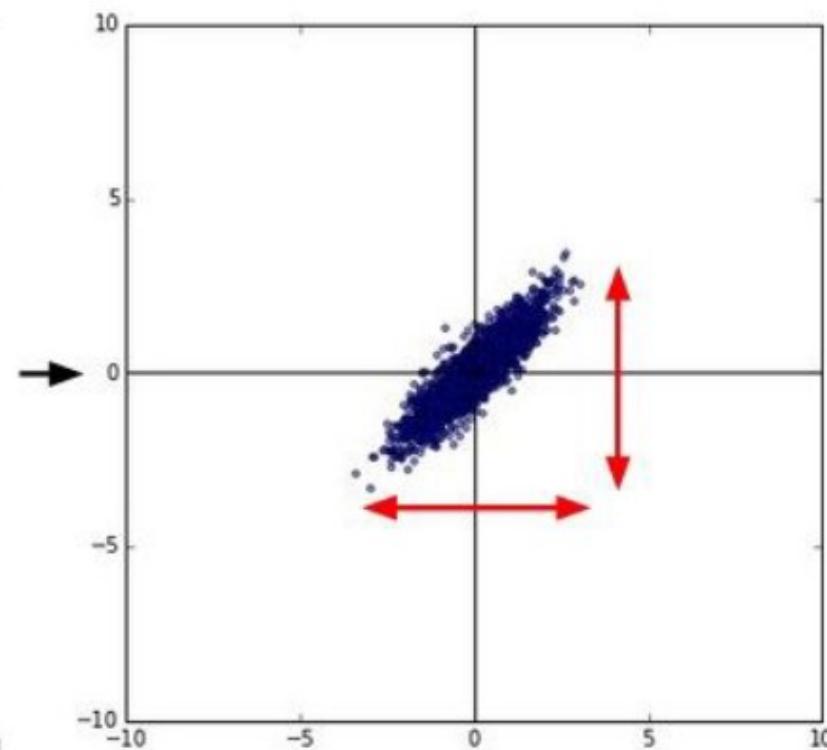
original data



zero-centered data



normalized data

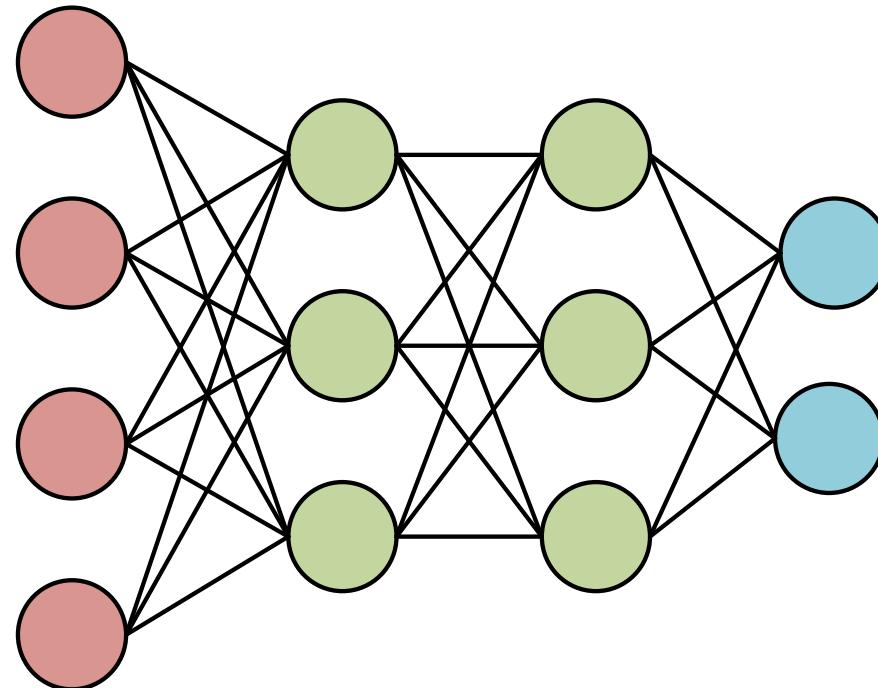


In Practice for Images

- 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

Weight Initialization

- What happens when $W=\text{constant}$ init is used?



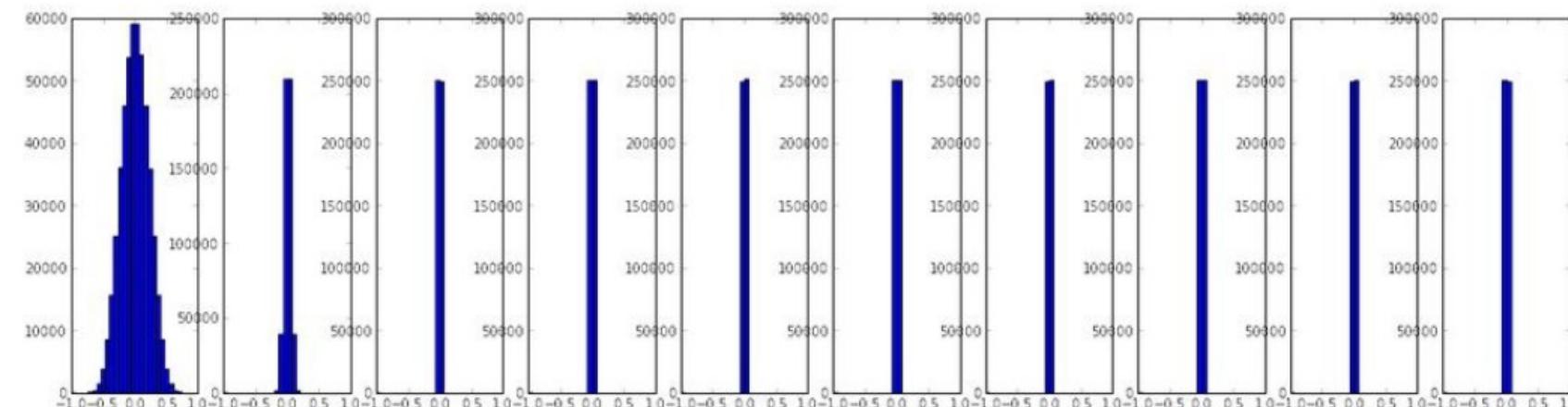
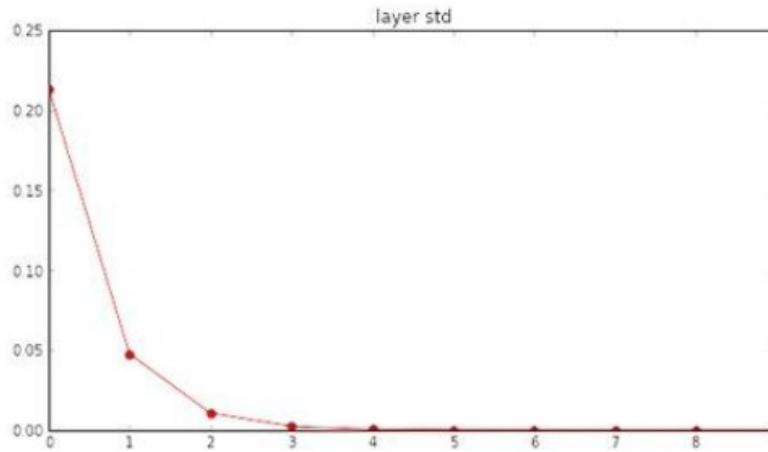
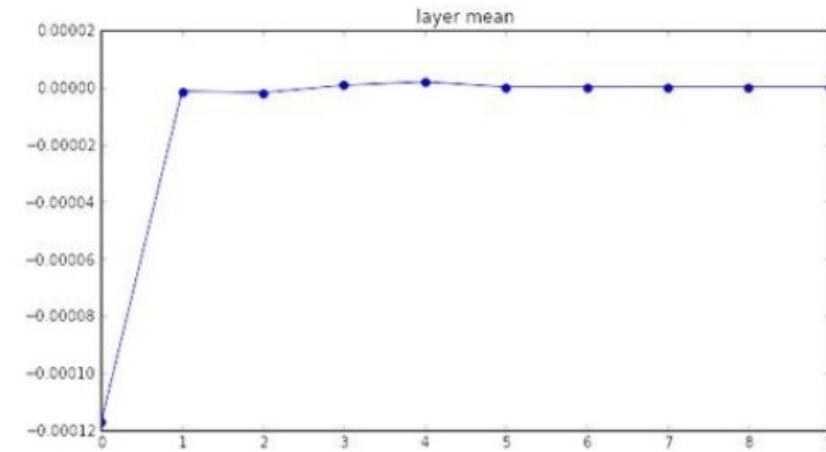
Weight Initialization

- First idea : Small random numbers
 - Gaussian with zero mean and $1e-2$ standard deviation

```
W = 0.01* np.random.randn(D,H)
```

- Let's look at some activation statistics
 - 10-layer network with 500 neurons on each layer
 - Using tanh activation function

```
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
```



All activations become zero!

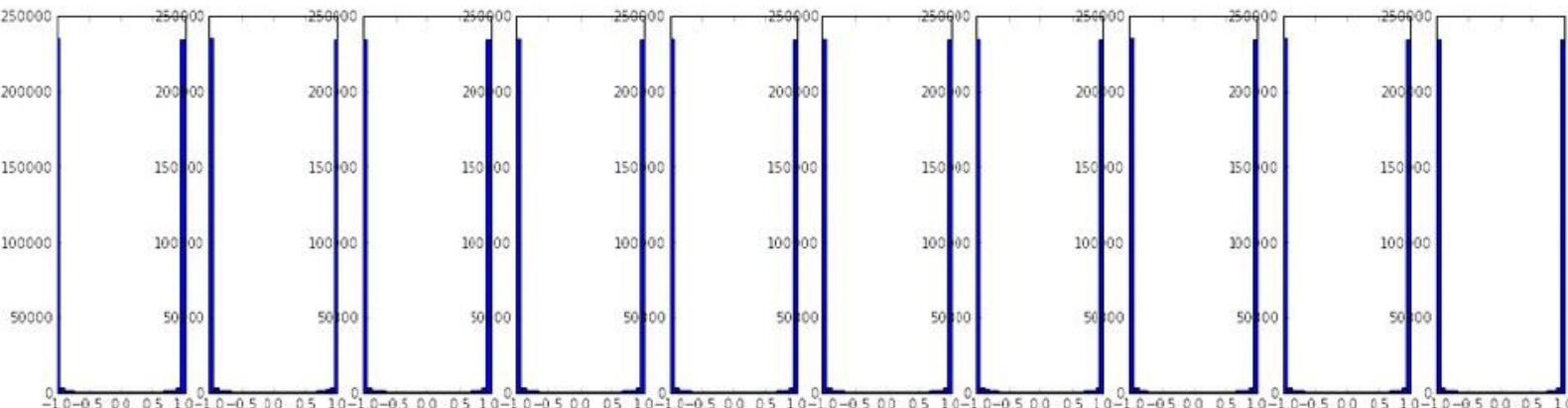
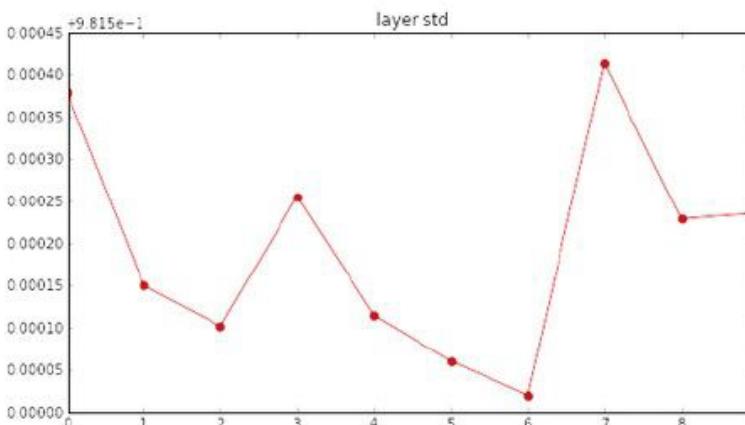
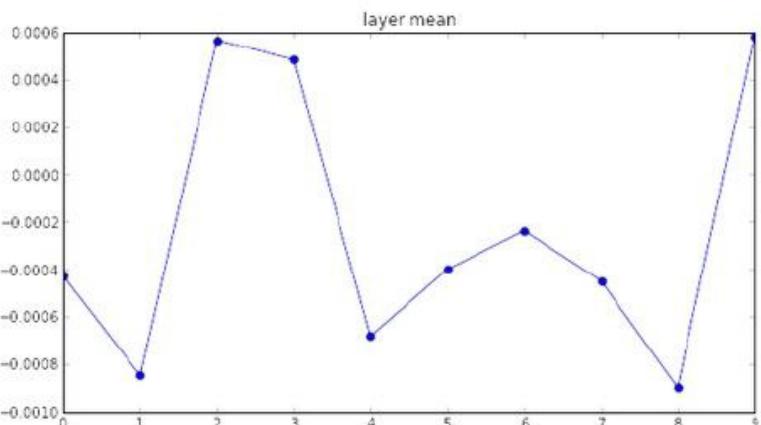
Q: think about the backward pass.
What do the gradients look like?

Hint: think about backward pass for a W^*X gate.

```
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.

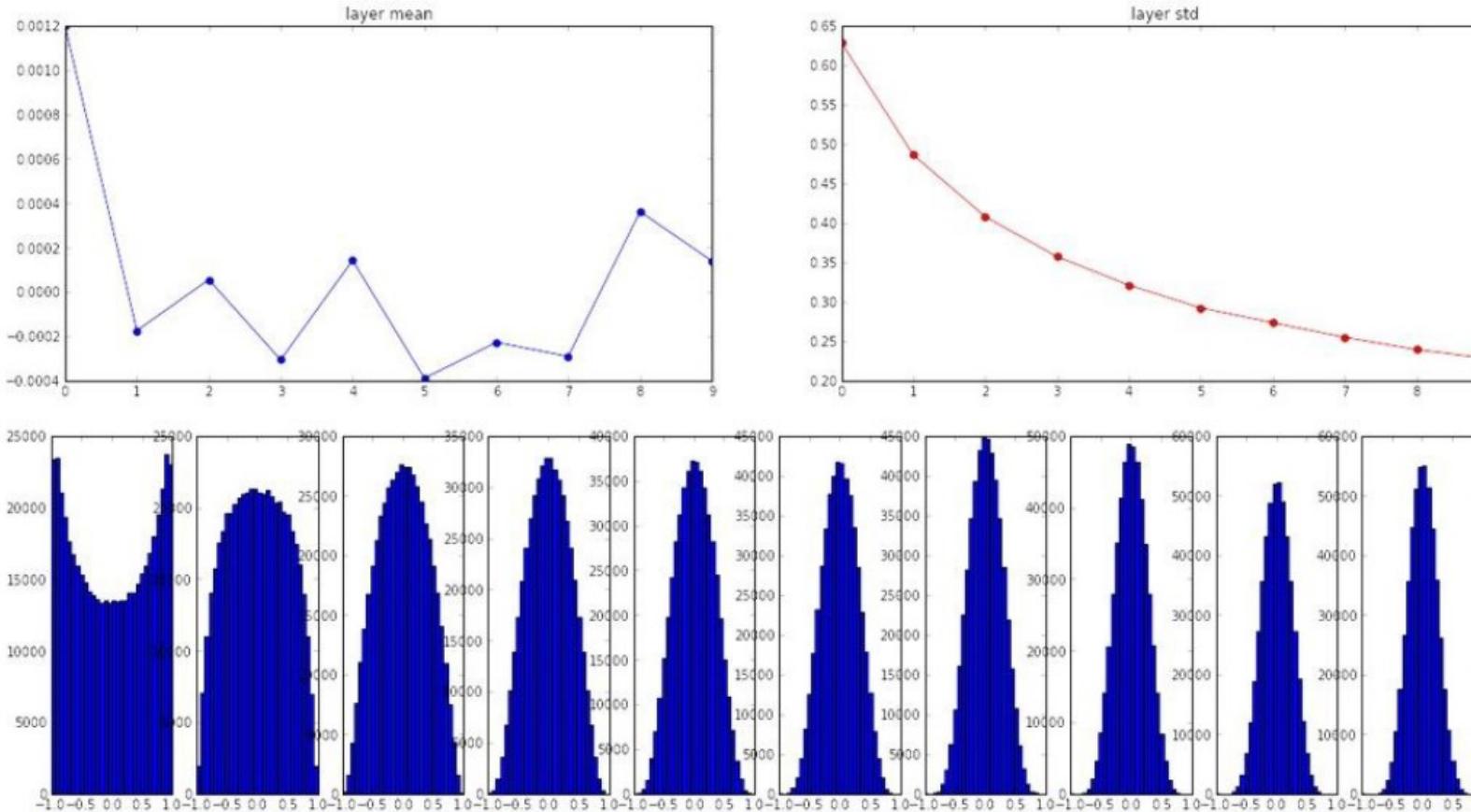
```
input layer had mean 0.001800 and std 1.001311  
hidden layer 1 had mean 0.001198 and std 0.627953  
hidden layer 2 had mean -0.000175 and std 0.486051  
hidden layer 3 had mean 0.000055 and std 0.407723  
hidden layer 4 had mean -0.000306 and std 0.357108  
hidden layer 5 had mean 0.000142 and std 0.320917  
hidden layer 6 had mean -0.000389 and std 0.292116  
hidden layer 7 had mean -0.000228 and std 0.273387  
hidden layer 8 had mean -0.000291 and std 0.254935  
hidden layer 9 had mean 0.000361 and std 0.239266  
hidden layer 10 had mean 0.000139 and std 0.228008
```

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

Input neuron 수가 다른데 같은 initialization
값을 사용하는 것은 문제가 있음
→ input 이 많으면 더 작은 weight 값으로
시작해야함

“Xavier initialization”
[Glorot et al., 2010]

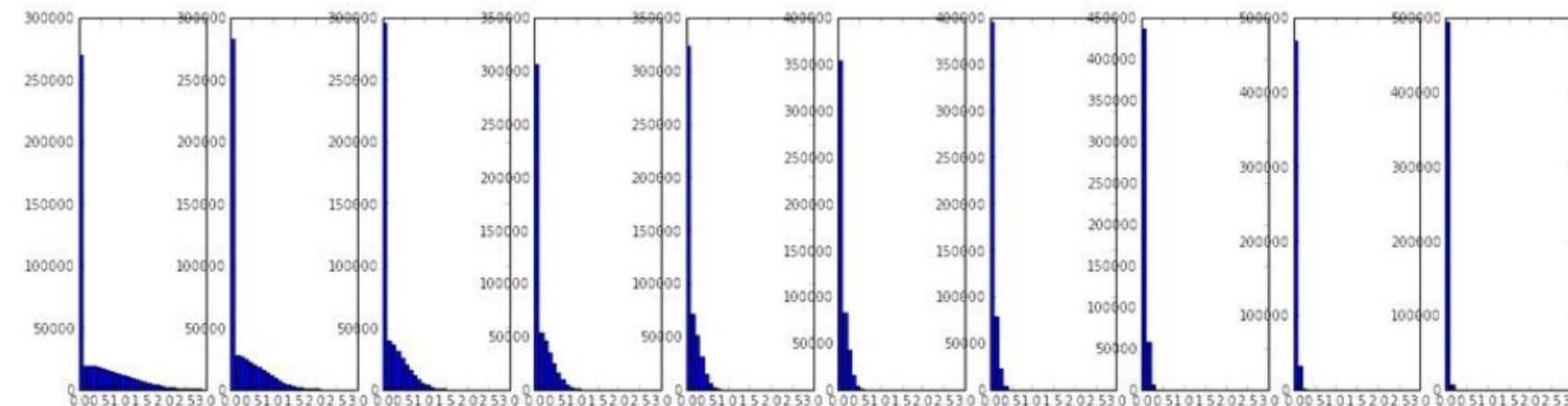
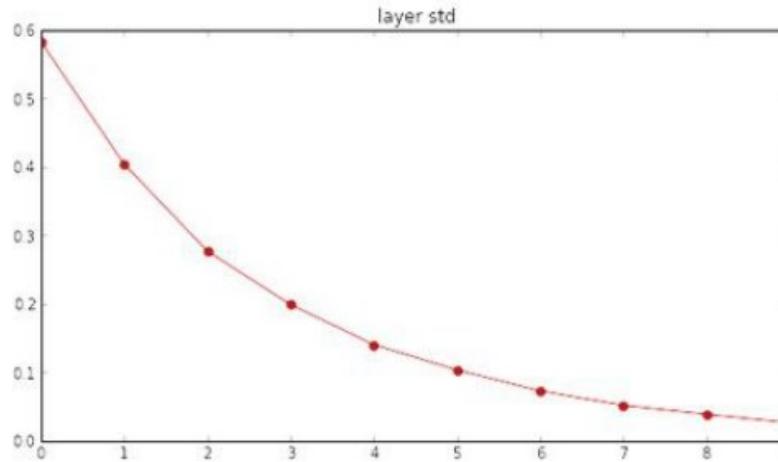
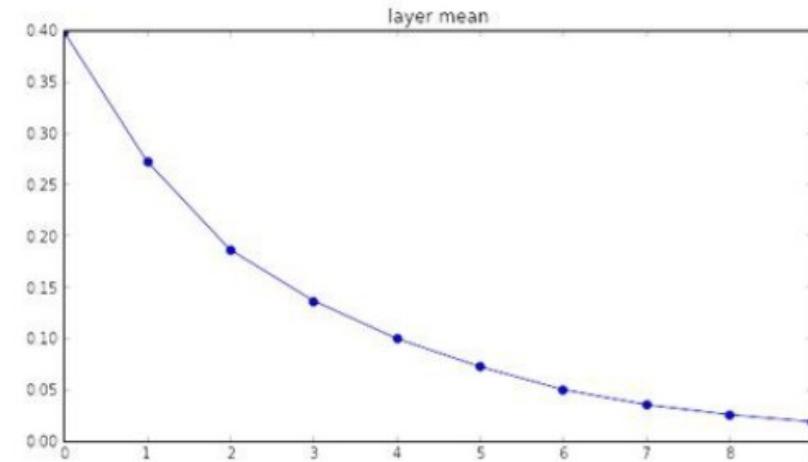
Reasonable initialization.
(Mathematical derivation
assumes linear activations)



```
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 0.186076 and std 0.276912  
hidden layer 4 had mean 0.136442 and std 0.198685  
hidden layer 5 had mean 0.099568 and std 0.140299  
hidden layer 6 had mean 0.072234 and std 0.103280  
hidden layer 7 had mean 0.049775 and std 0.072748  
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.



```

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 0.545867 and std 0.813855
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 0.587103 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

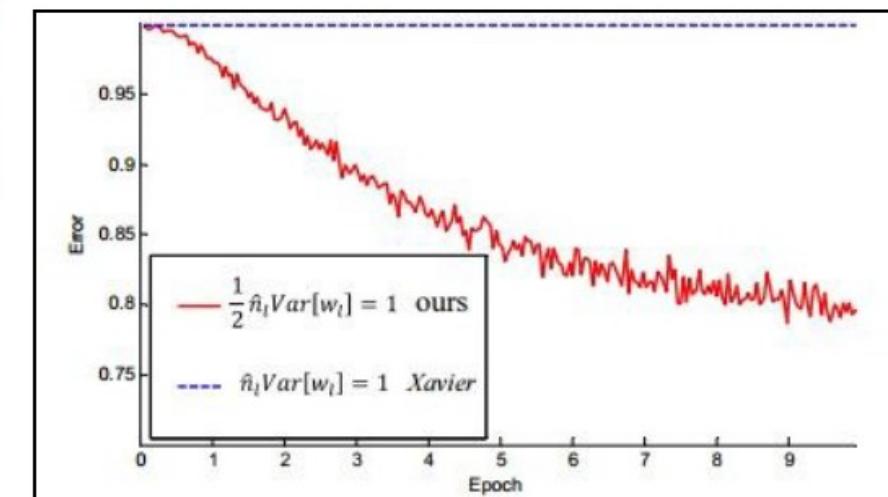
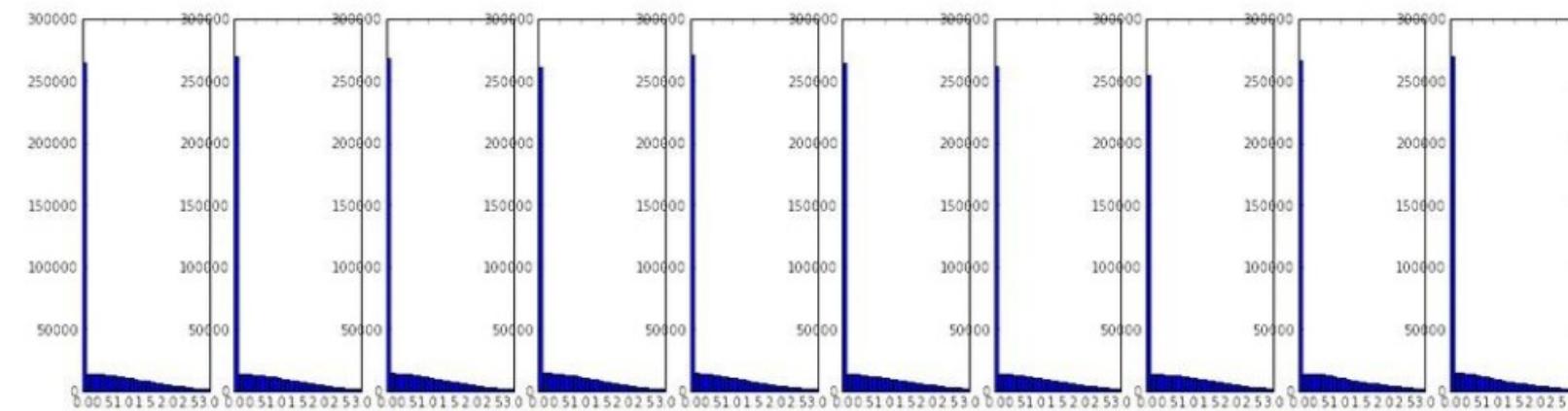
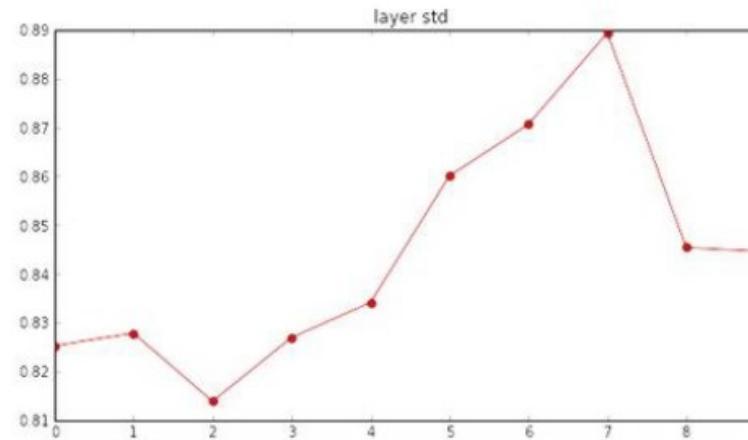
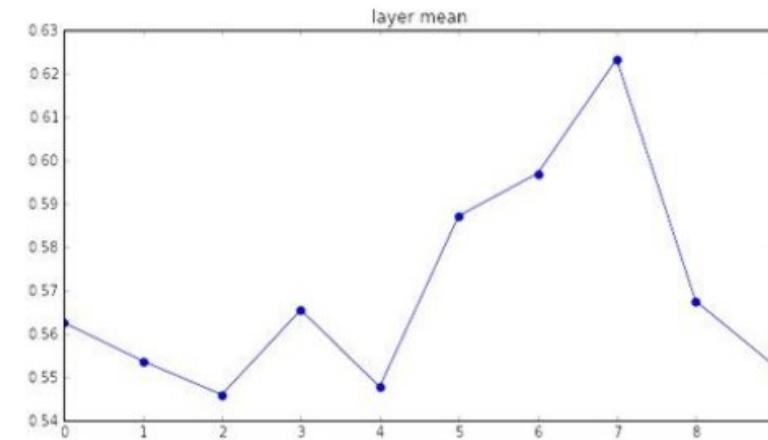
```

```

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

```

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



Weight Initialization

- Xavier Initialization
 - Activation function은 linear라고 가정하고, in/out의 variance를 같게 해보자

forward: $\text{Var}(W_i) = \frac{1}{n} = \frac{1}{n_{\text{in}}}$

backward: $\text{Var}(W_i) = \frac{1}{n_{\text{out}}}$

**Xavier
Initialization:** $\text{Var}(W_i) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$

Weight Initialization

- He Initialization
 - Activation function을 ReLU나 PReLU로 하고, variance를 같게 해보자

ReLU

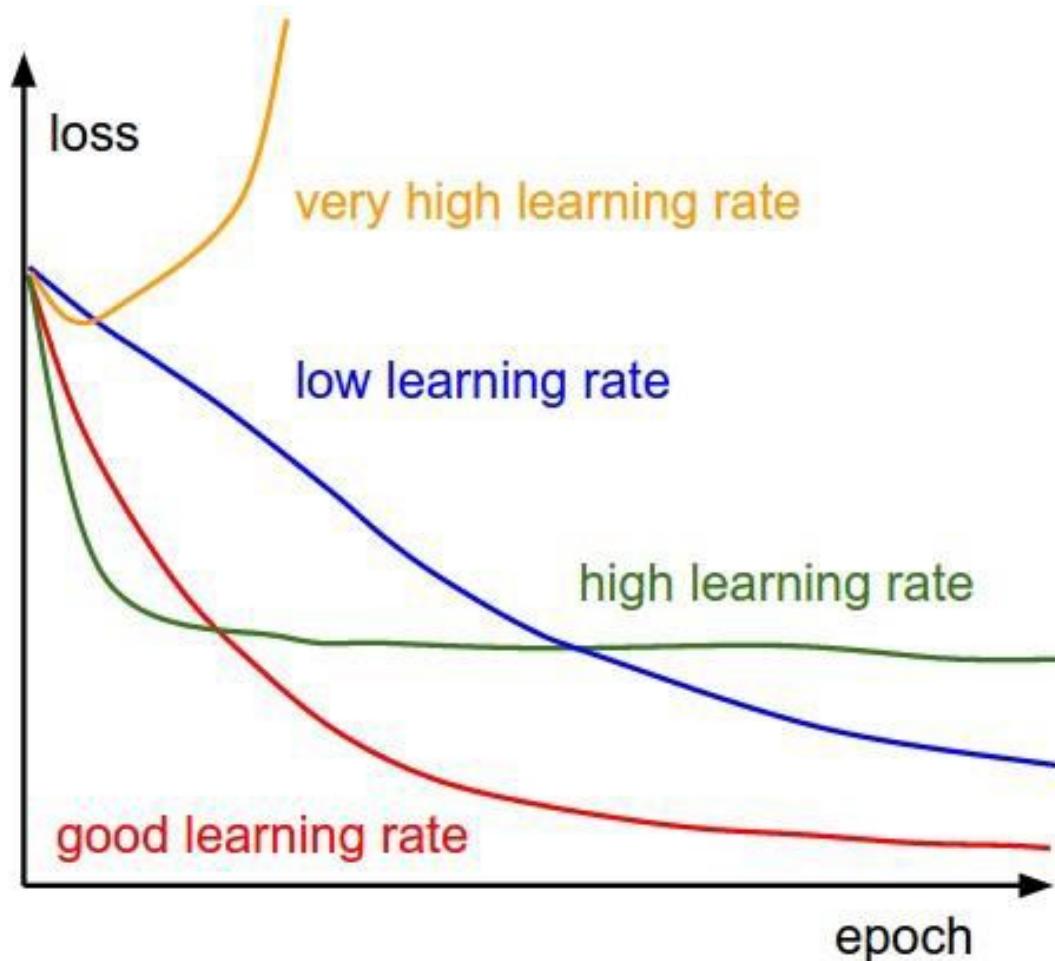
$$\left\{ \begin{array}{l} Var[w_l] = \frac{2}{n_l} \implies \text{standard deviation (std)} = \sqrt{\frac{2}{n_l}} \\ w_l \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_l}}\right) \text{ and } \mathbf{b} = 0 \end{array} \right.$$

PReLU

$$\frac{1}{2}(1 + a^2)n_l \underline{Var[w_l]} = 1$$

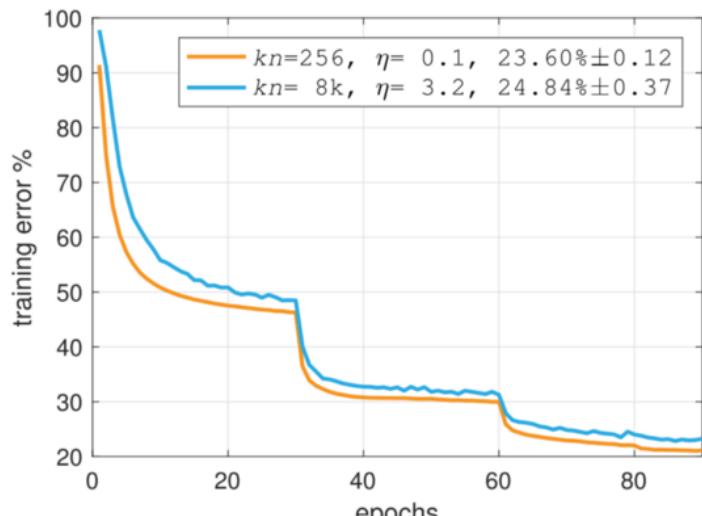
Learning Rate

- learning rate에 따라서 최종 결과가 달라짐

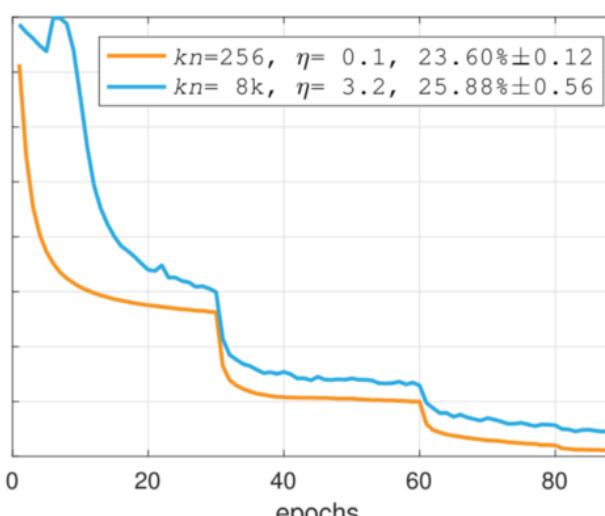


Learning Rate Decay

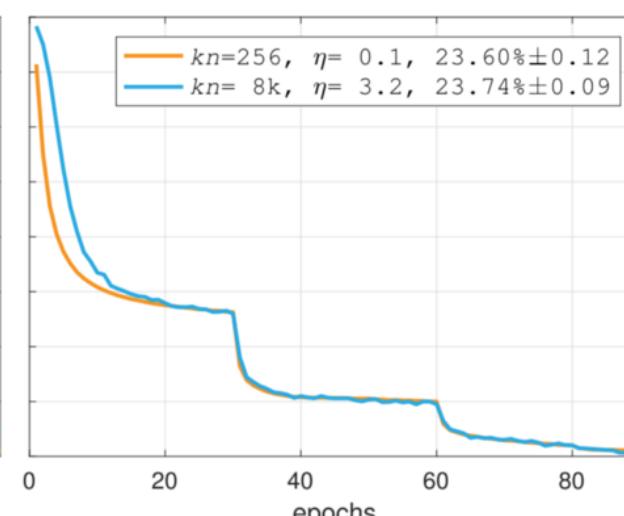
- Learning rate이 너무 크면 수렴을 못할 가능성이 있고, 너무 작으면 local minima 혹은 saddle point에서 못빠져나옴
- Learning Rate Decay
 - 처음에는 크게 움직이다가 일정 조건이 되면 learning rate을 낮춰서 점점 작게 움직이는 방법



(a) no warmup



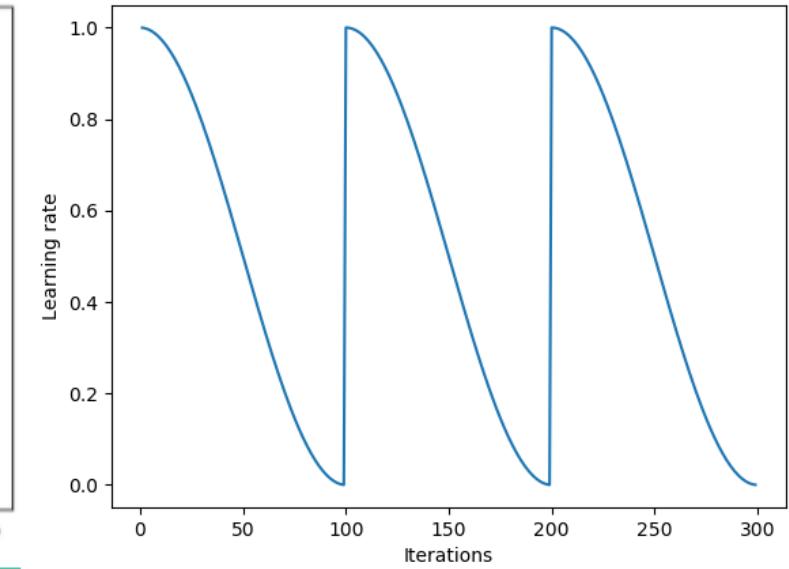
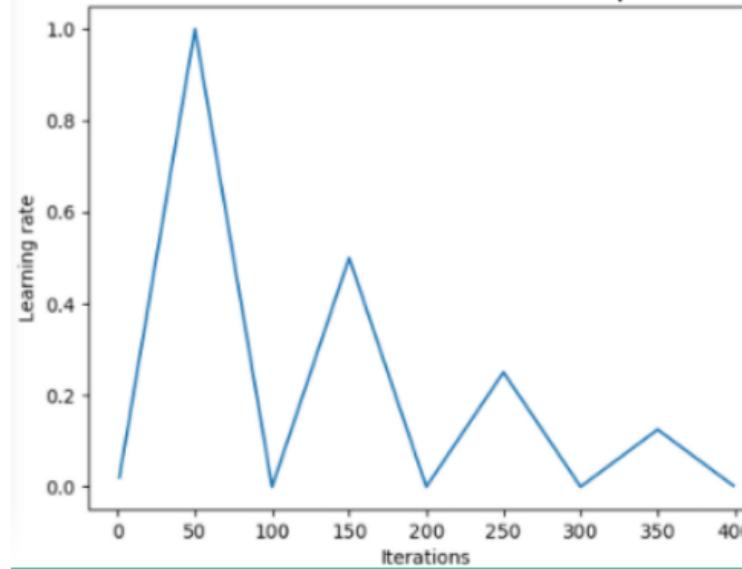
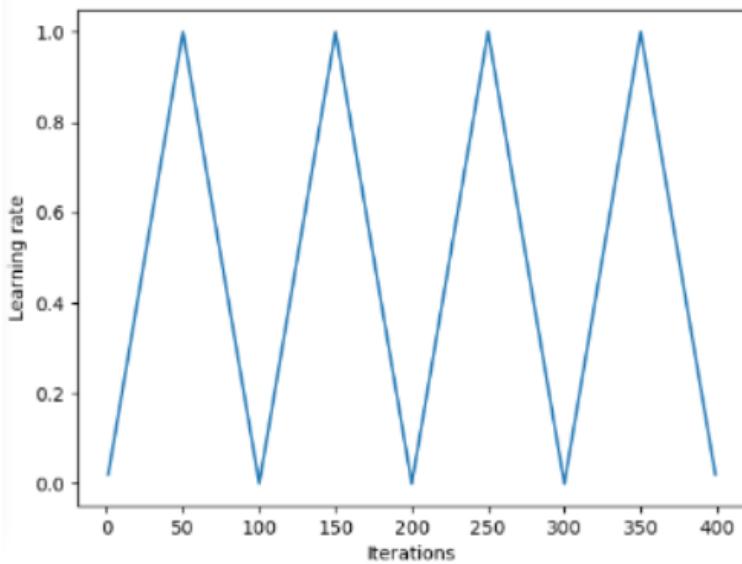
(b) constant warmup



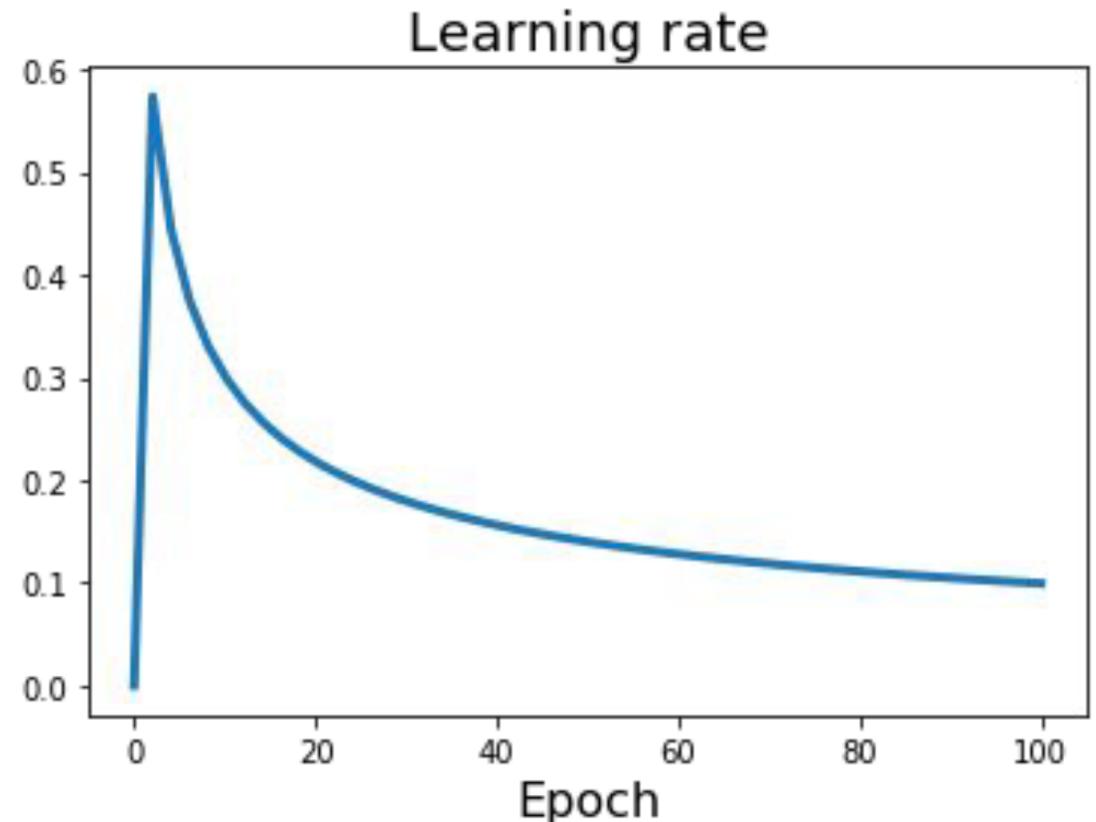
(c) gradual warmup

Cyclic Learning Rate

- Learning rate decay로 saddle point를 빠져나갈 수 있을까?
- Saddle point에서는 learning rate을 키워서 빠져나가는 것이 효과적일 수 있음 → Learning rate을 주기적으로 변경해보자



Linear Warmup



High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this

Empirical rule of thumb: If you increase the batch size by N , also scale the initial learning rate by N

Underfitting vs Overfitting

$$\text{test error} = \text{training error} + \text{generalization gap}$$

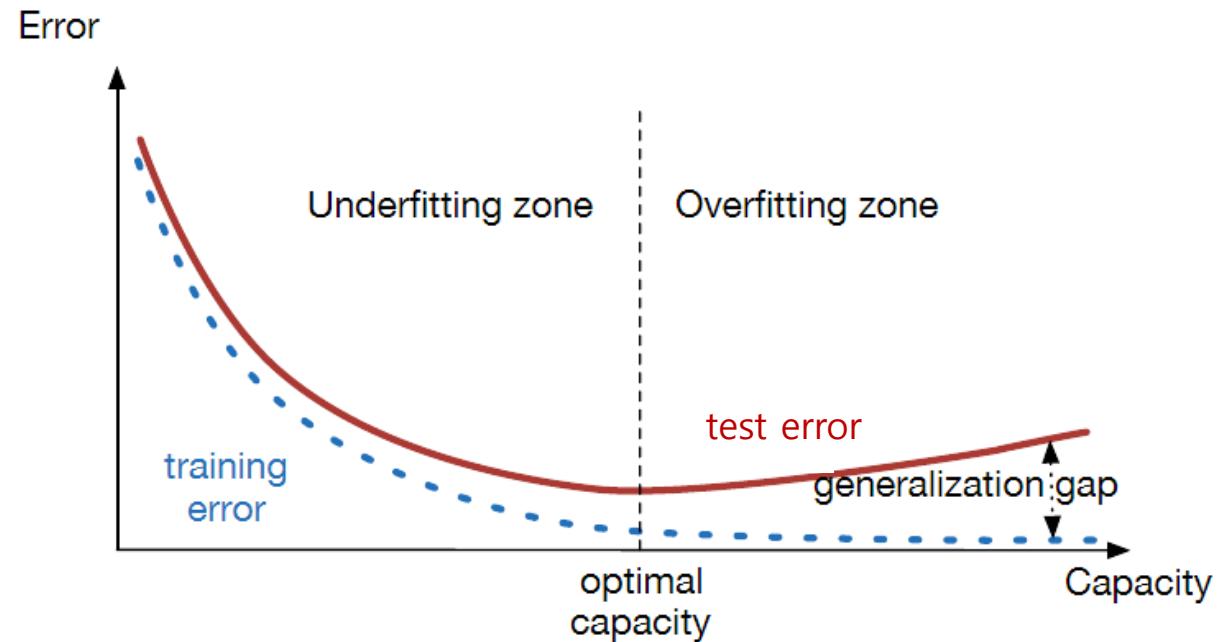
Model capacity가 너무 작으면,
- 아무리 학습을 해도 성능이 안나옴

→underfitting

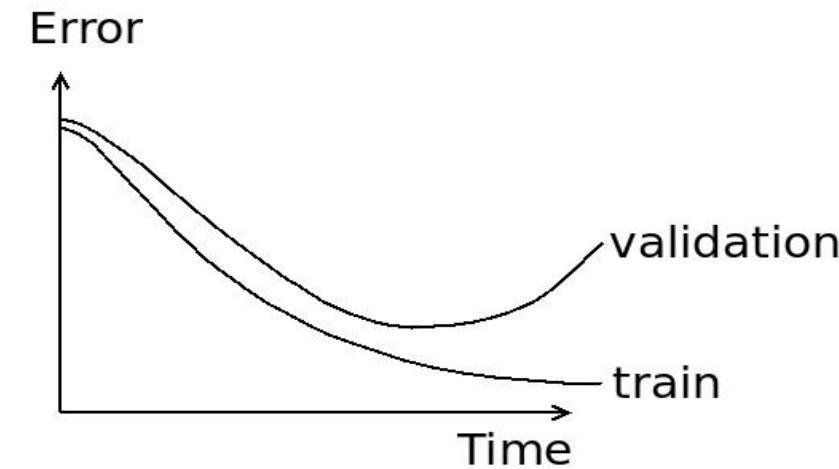
Model capacity가 너무 크면,
- Generalization gap이 커짐

→overfitting

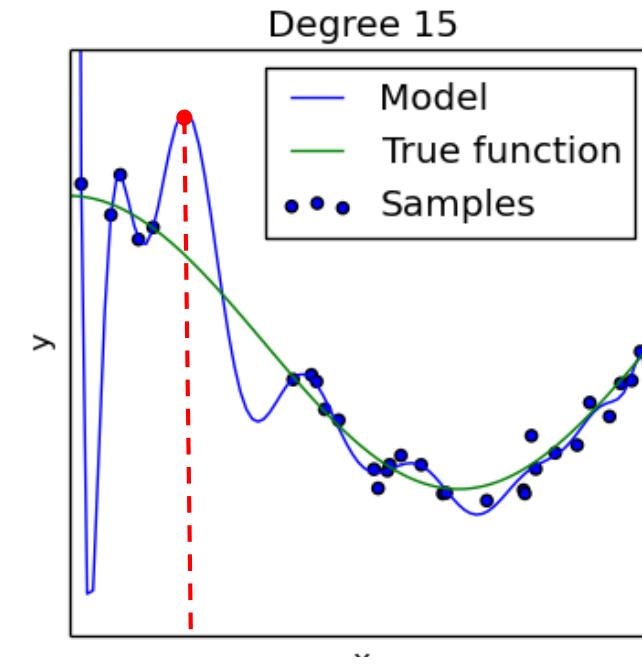
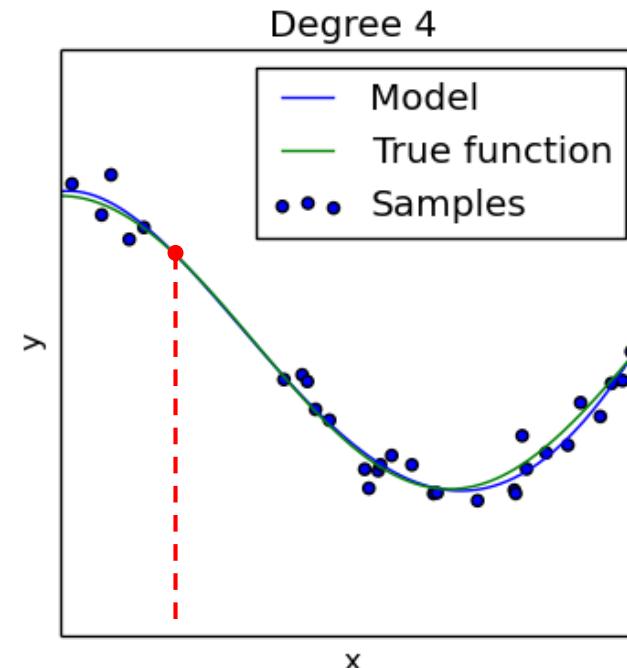
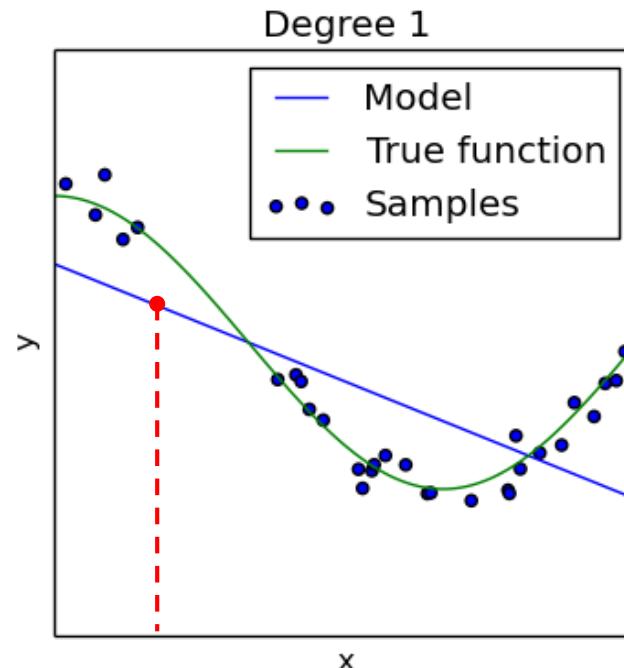
Deep Learning에 사용하는 model은 capacity가 대부분 크기
때문에 overfitting에 취약함



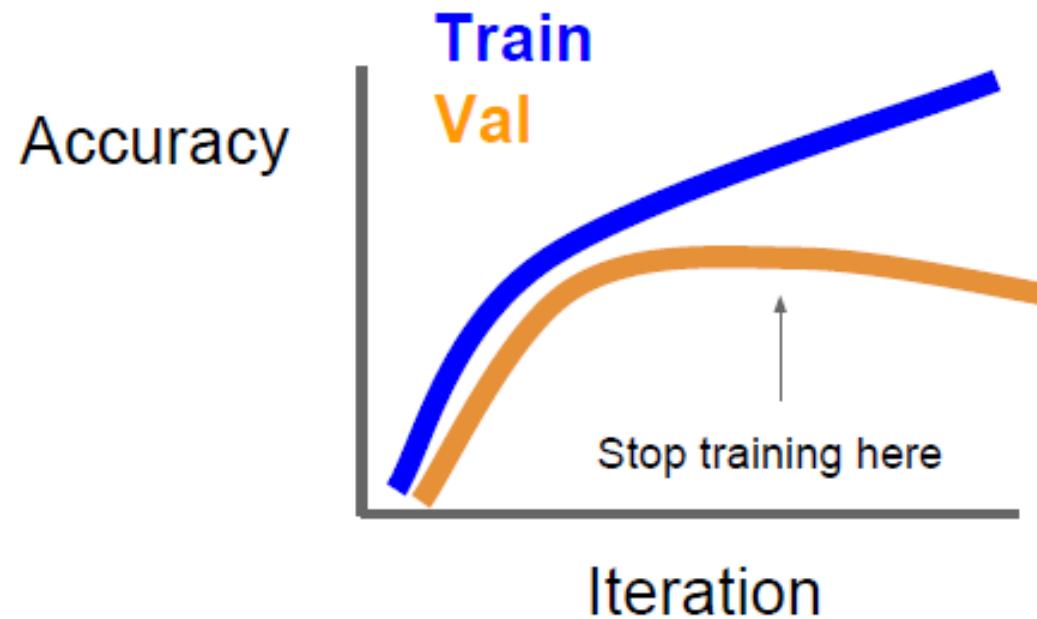
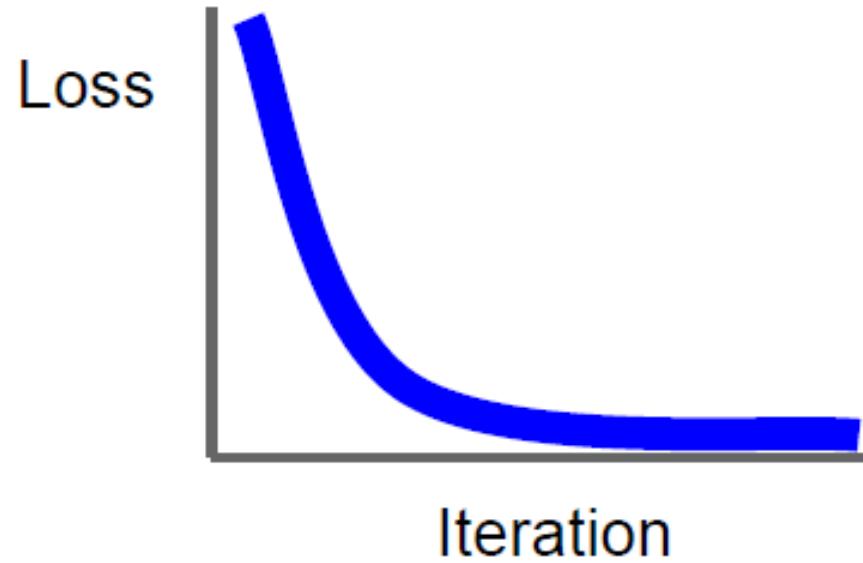
Underfitting vs Overfitting



Overfitting : Training data에 너무 최적화(fitting)를 함으로 인해서 일반화(generalization) 성능이 떨어지는 현상



Early Stopping



Stop training the model when accuracy on the validation set decreases

Or train for a long time, but always keep track of the model snapshot that worked best on val

Regularization Method

λ = regularization strength
(hyperparameter)

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i) + \lambda R(W)$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too well* on training data

Simple examples

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

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

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

More complex:

Dropout

Batch normalization

Stochastic depth, fractional pooling, etc

Add Term to Loss

L2 regularization

L1 regularization

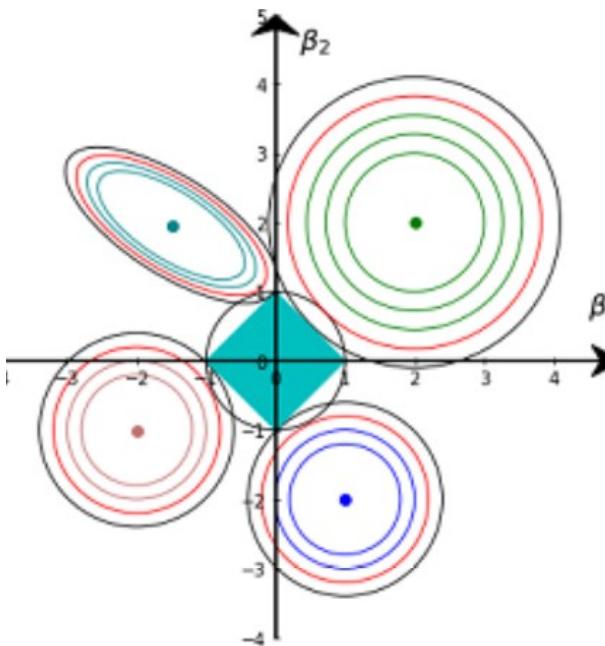
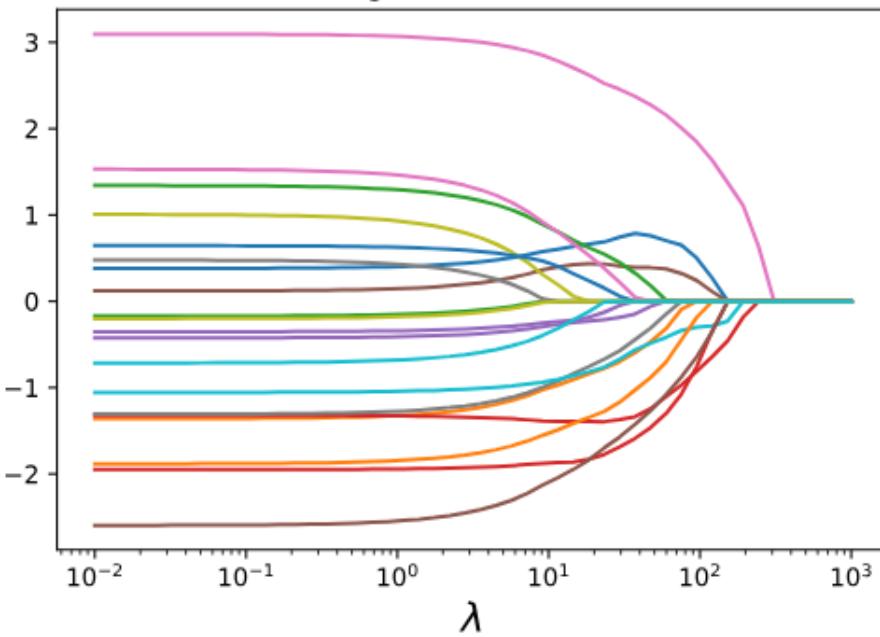
Elastic net (L1 + L2)

$$R(W) = \sum_k \sum_l W_{k,l}^2 \quad (\text{Weight decay})$$

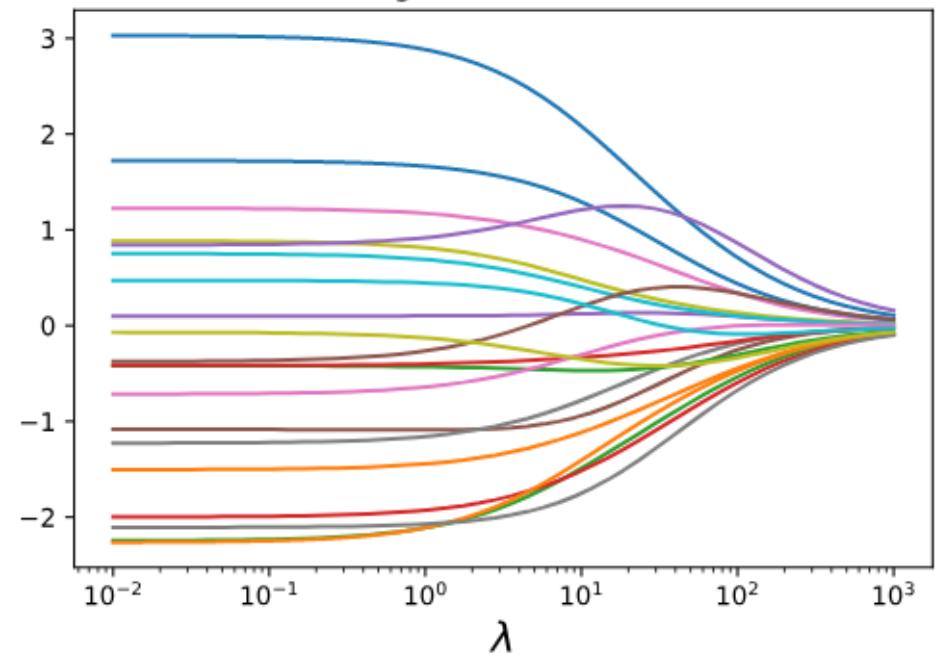
$$R(W) = \sum_k \sum_l |W_{k,l}|$$

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

L1 Regularization
Regularization Path

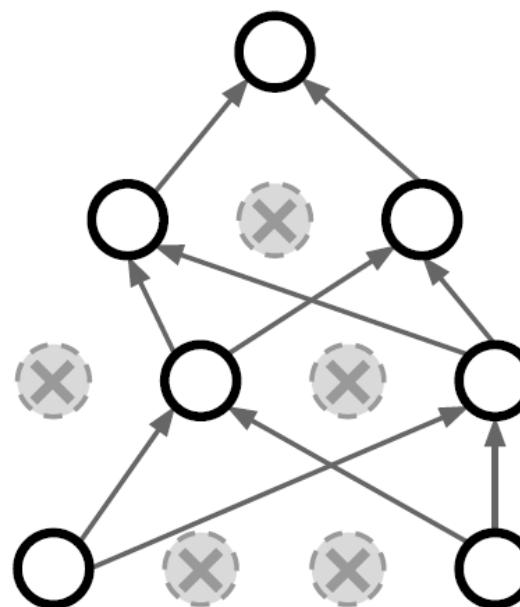
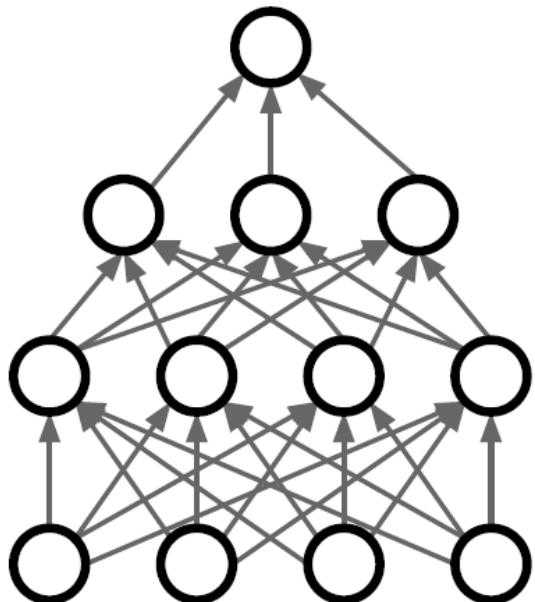


L2 Regularization
Regularization Path



Dropout

- In each forward pass, randomly set some neurons to zero
- Probability of dropping is a hyper-parameter; 0.5 is common



Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

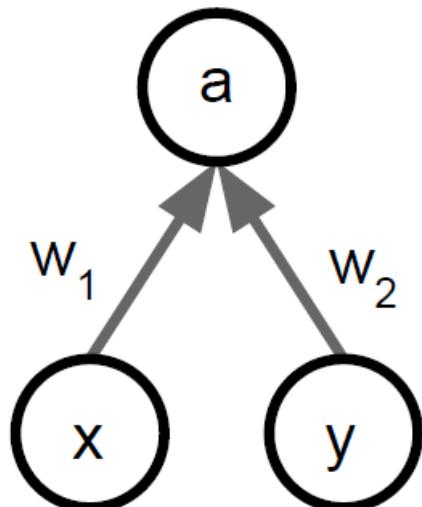
An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test Time

- Dropout makes the output random!
- Want to average out the randomness at test time

Consider a single neuron.



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

During training we have:

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

At test time, **multiply**
by dropout probability

Batch Normalization

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

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

After normalization, allow the network to squash the range if it wants to

Note, the network can learn:
 $\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$
 $\beta^{(k)} = \text{E}[x^{(k)}]$
to recover the identity mapping.

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization during training
- Zero overhead at test-time
- Behaves differently during training and testing: **this is a very common source of bugs**

Batch Normalization: Test Time

$$\mu_j = \text{(Running) average of values seen during training}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

Per-channel var, shape is D

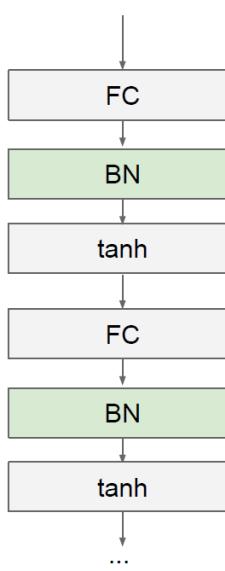
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

- The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.
- It can be estimated during training with running averages



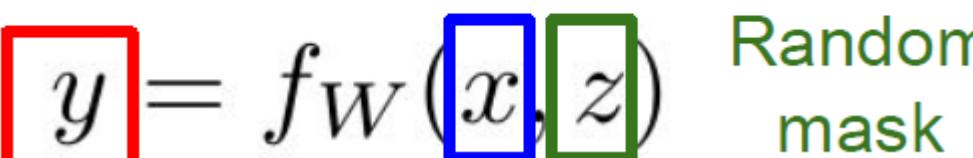
Regularization: A Common Pattern

- Training : Add some kind of randomness

Output Input
(label) (image)

$$y = f_W(x, z)$$

Random mask



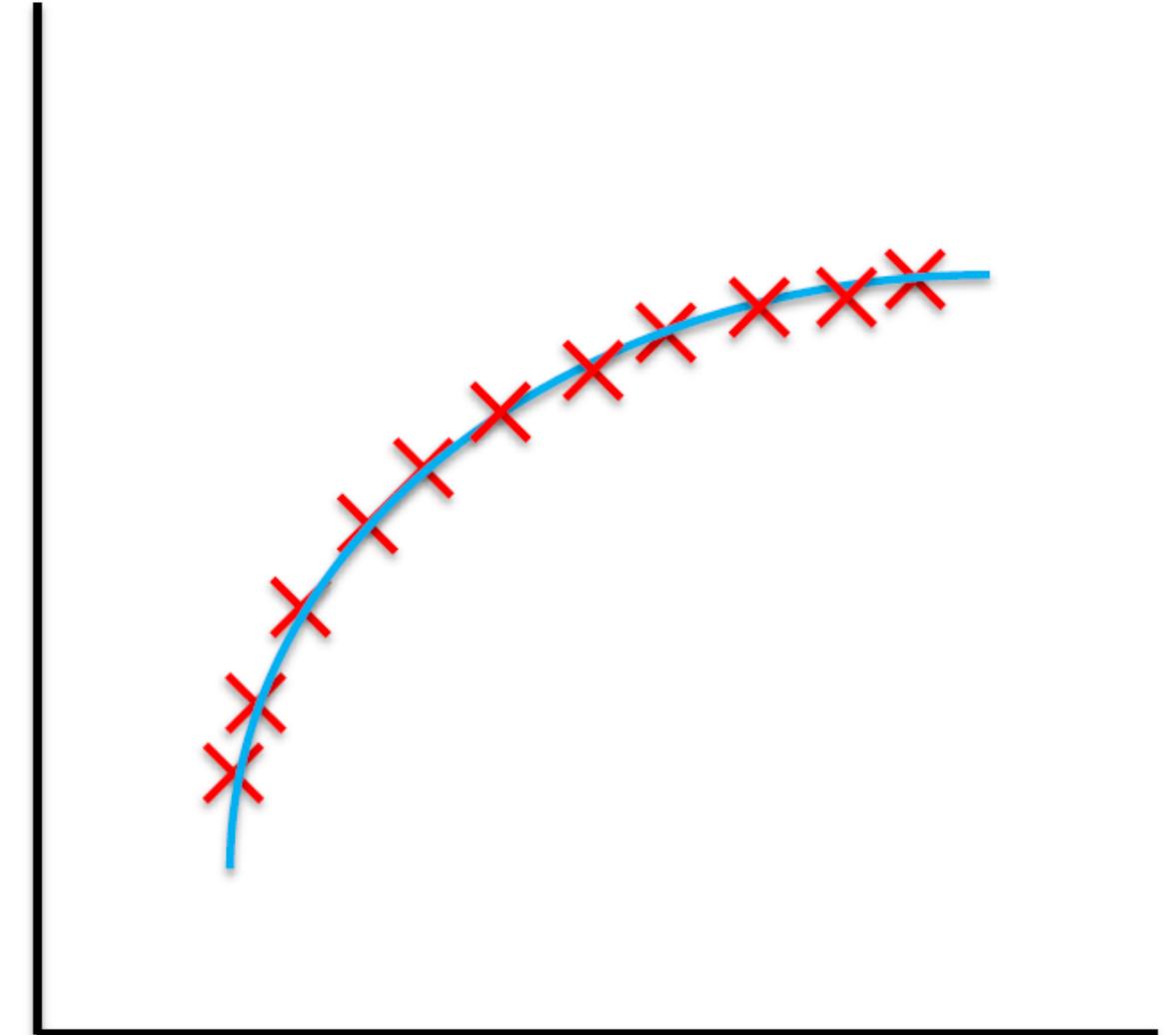
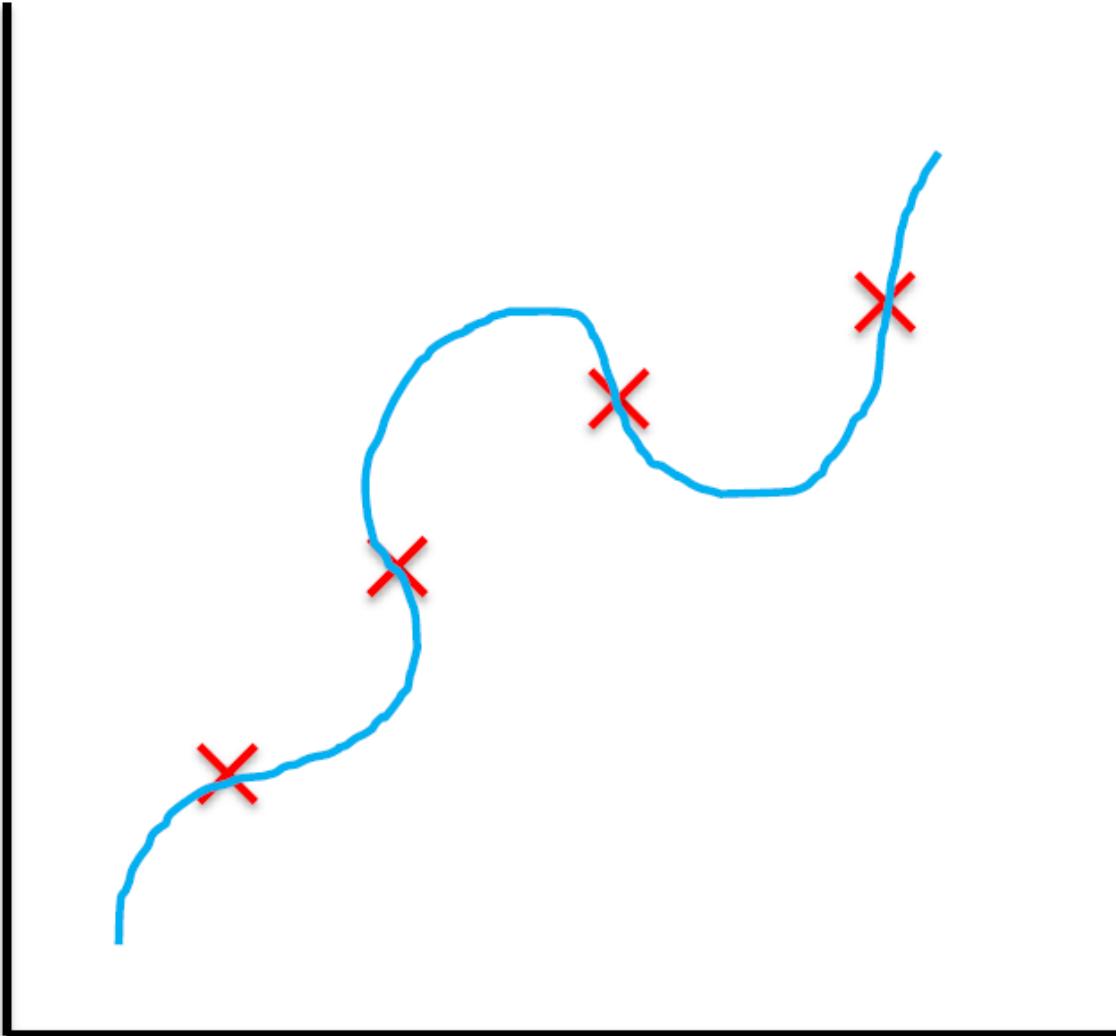
- Testing : Average out randomness

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

Regularization: A Common Pattern

- Dropout
 - Training : Randomly set some neurons to zero
 - Testing : Average out the randomness by multiplying dropout probability
- Batch Normalization
 - Training : Normalize using stats from random mini-batches
 - Testing : Use fixed stats to normalize

Data Augmentation

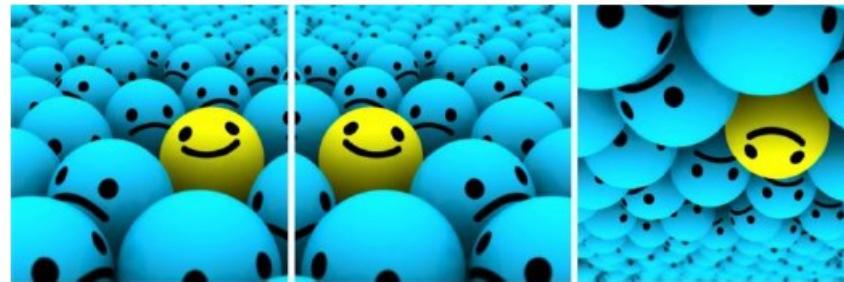


Data Augmentation

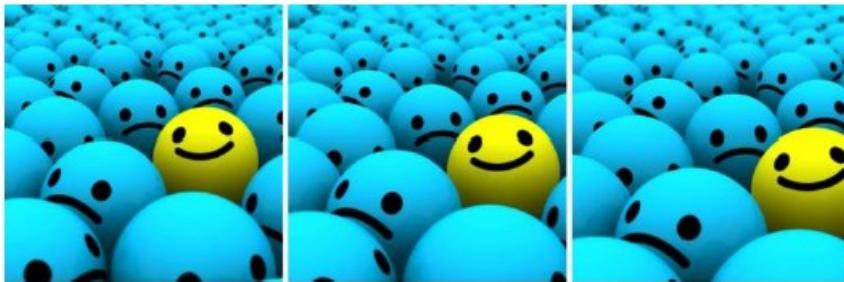
Crop:



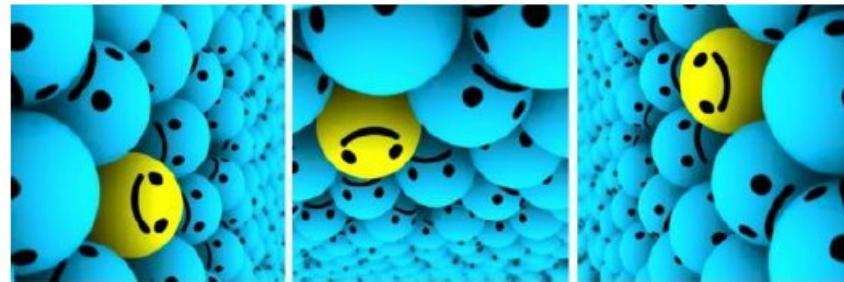
Flip:



Scale:



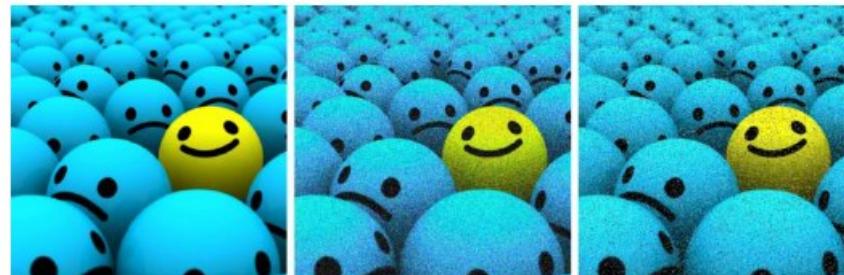
Rotate:



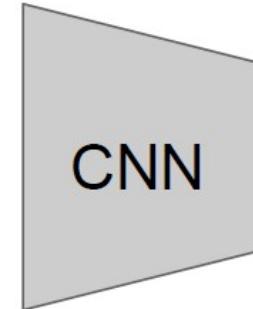
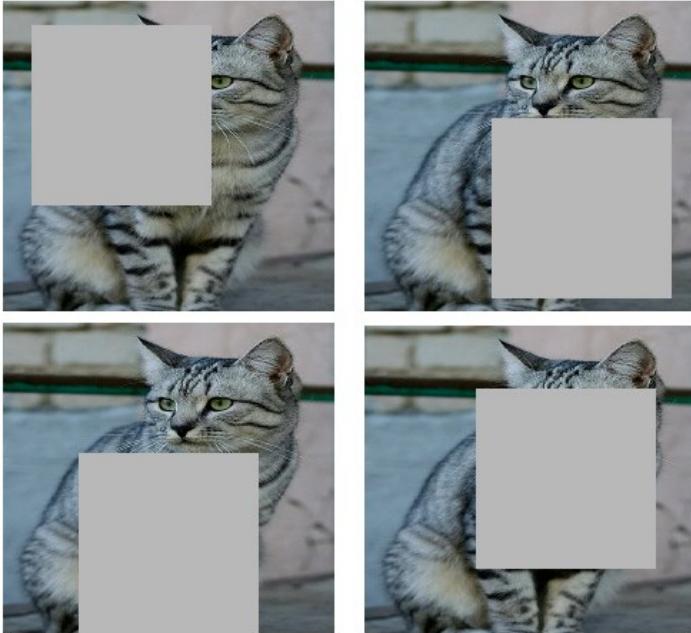
Translation:



Noise:



CutOut & MixUp



Target label:
cat: 0.4
dog: 0.6

Randomly blend the pixels
of pairs of training images,
e.g. 40% cat, 60% dog

Works very well for small datasets like CIFAR,
less common for large datasets like ImageNet

CutMix

	ResNet-50	Mixup [48]	Cutout [3]	CutMix
Image				
Label	Dog 1.0	Dog 0.5 Cat 0.5	Dog 1.0	Dog 0.6 Cat 0.4
ImageNet	76.3	77.4	77.1	78.6
Cls (%)	(+0.0)	(+1.1)	(+0.8)	(+2.3)
ImageNet	46.3	45.8	46.7	47.3
Loc (%)	(+0.0)	(-0.5)	(+0.4)	(+1.0)
Pascal VOC	75.6	73.9	75.1	76.7
Det (mAP)	(+0.0)	(-1.7)	(-0.5)	(+1.1)

$$\tilde{x} = \mathbf{M} \odot x_A + (1 - \mathbf{M}) \odot x_B$$

$$\tilde{y} = \lambda y_A + (1 - \lambda) y_B,$$

$$r_x \sim \text{Unif } (0, W), \quad r_w = W\sqrt{1 - \lambda},$$

$$r_y \sim \text{Unif } (0, H), \quad r_h = H\sqrt{1 - \lambda}$$

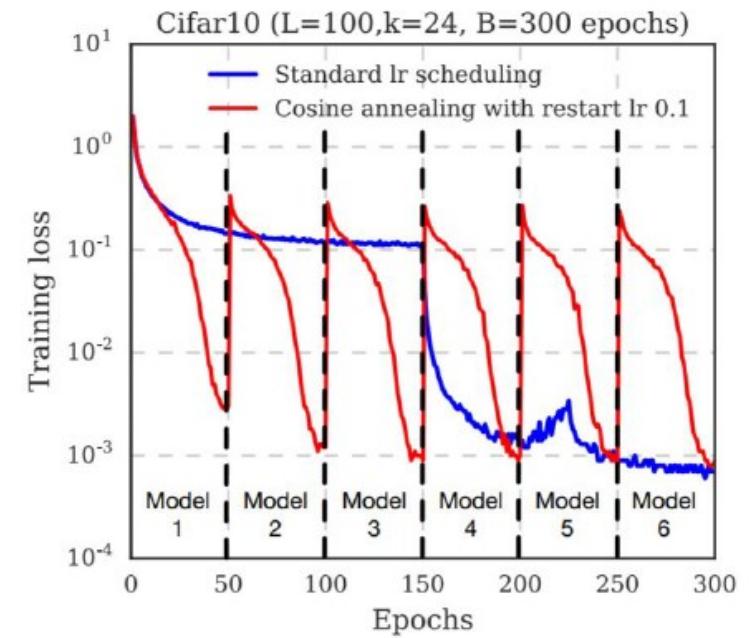
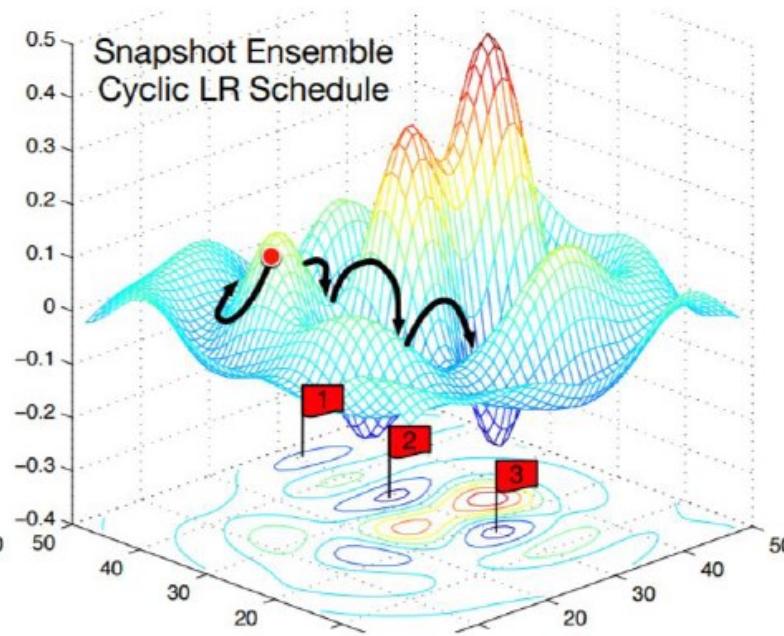
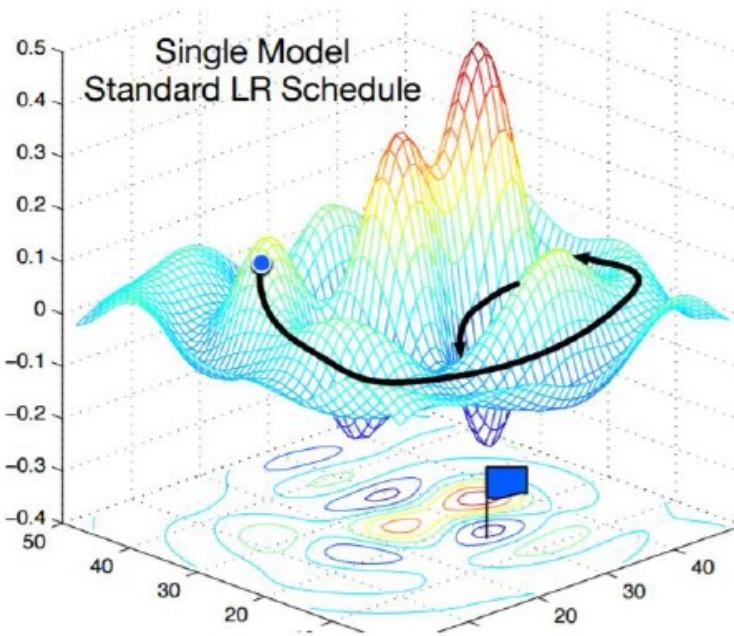
Model Ensembles

1. Train multiple independent models
 2. At test time average their results
 - Take average of predicted probability distributions, then choose argmax
- Enjoy 2% extra performance!

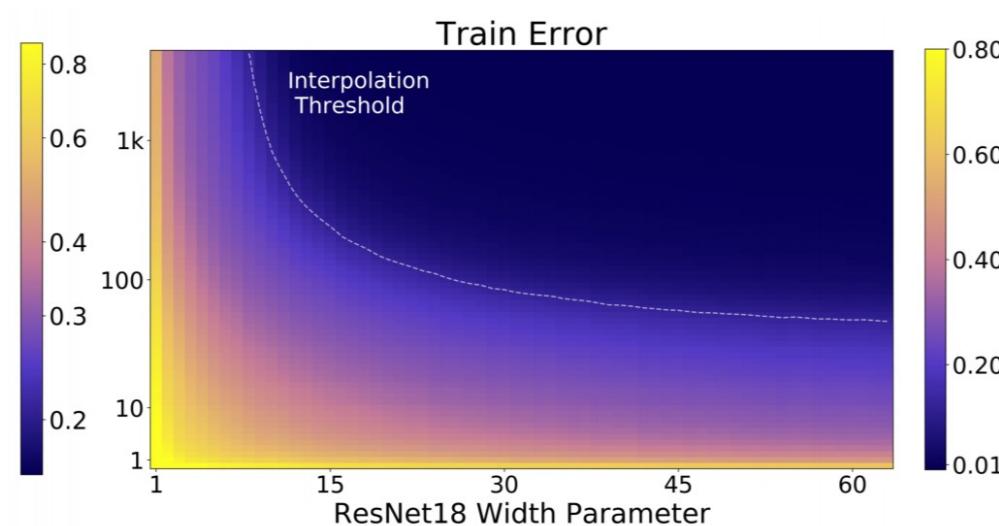
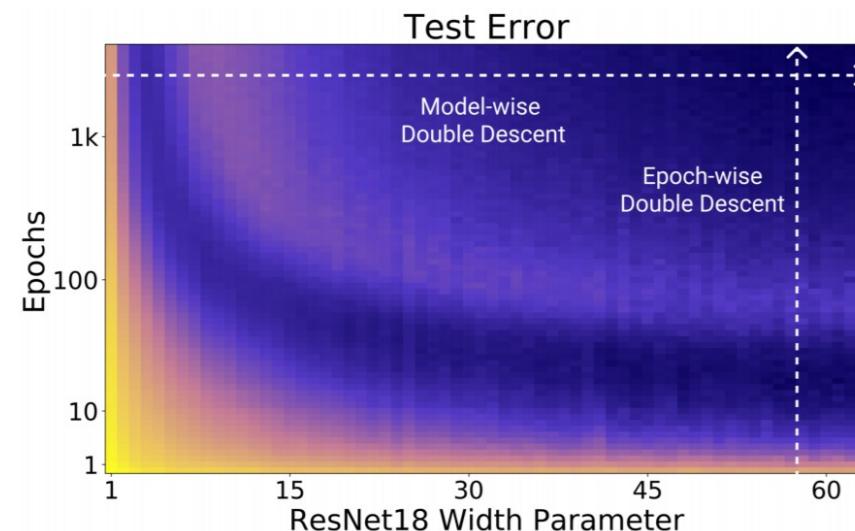
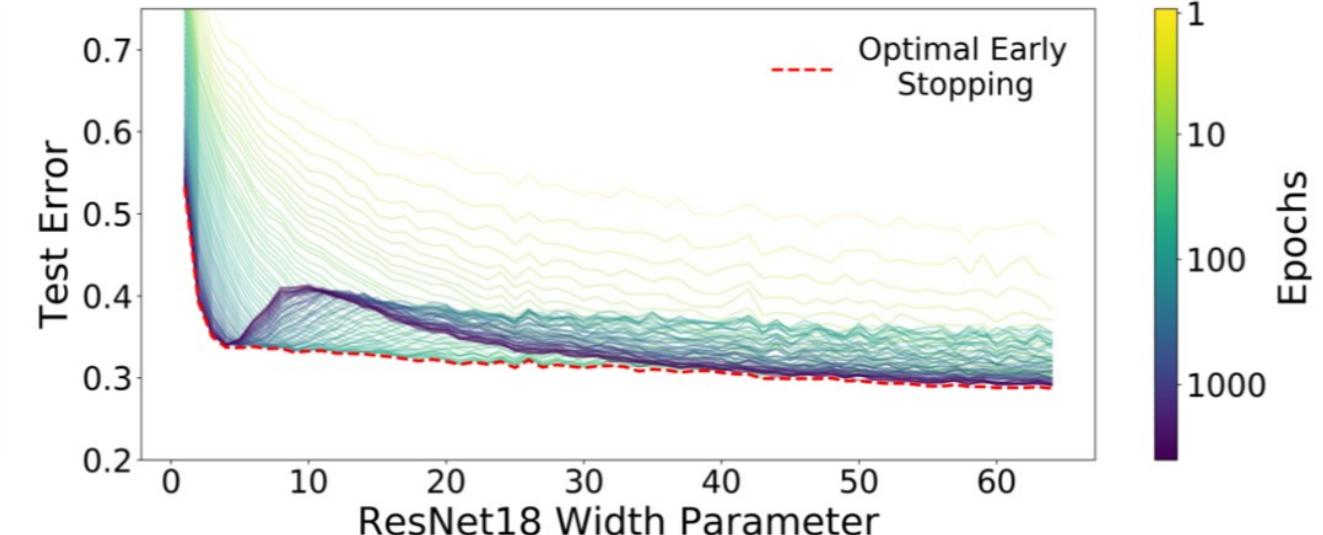
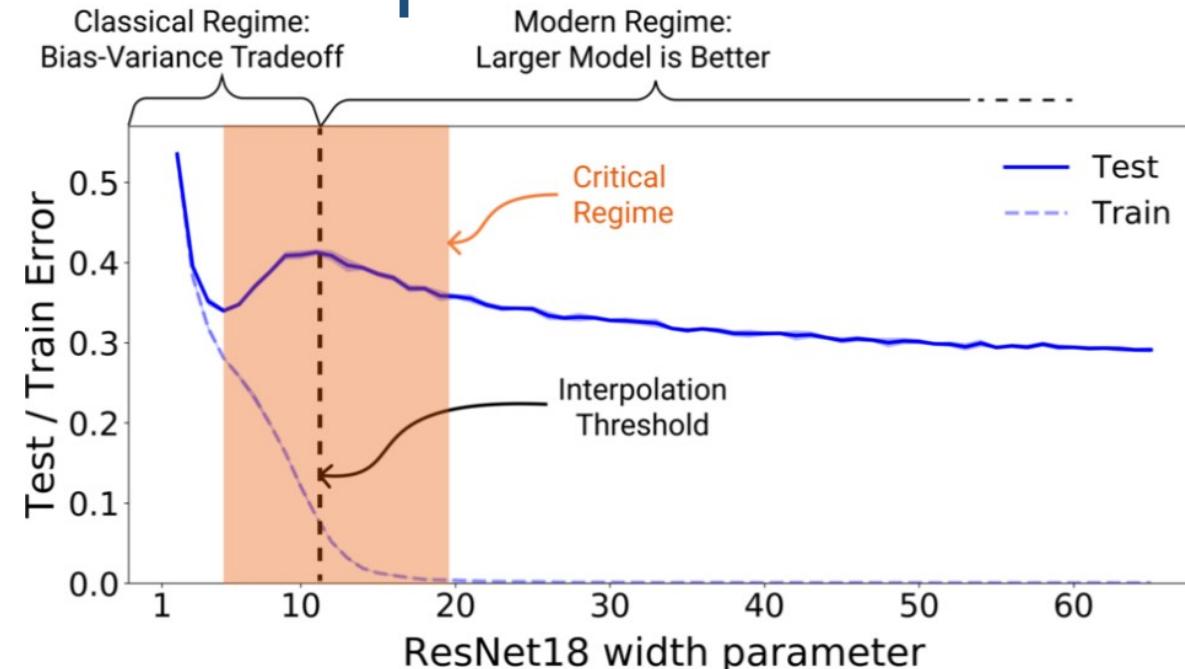


Model Ensembles: Tips and Tricks

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

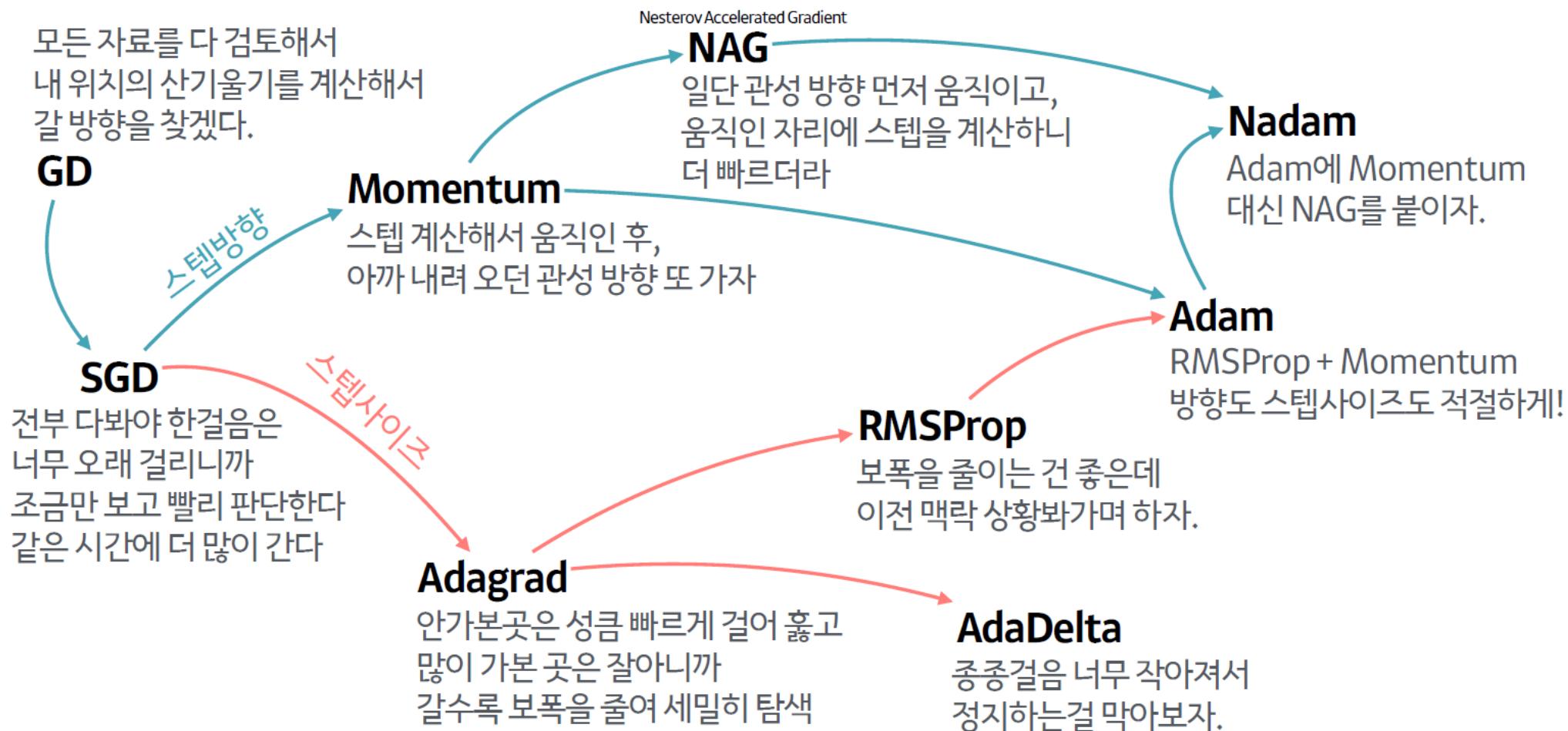


Deep Double Descent

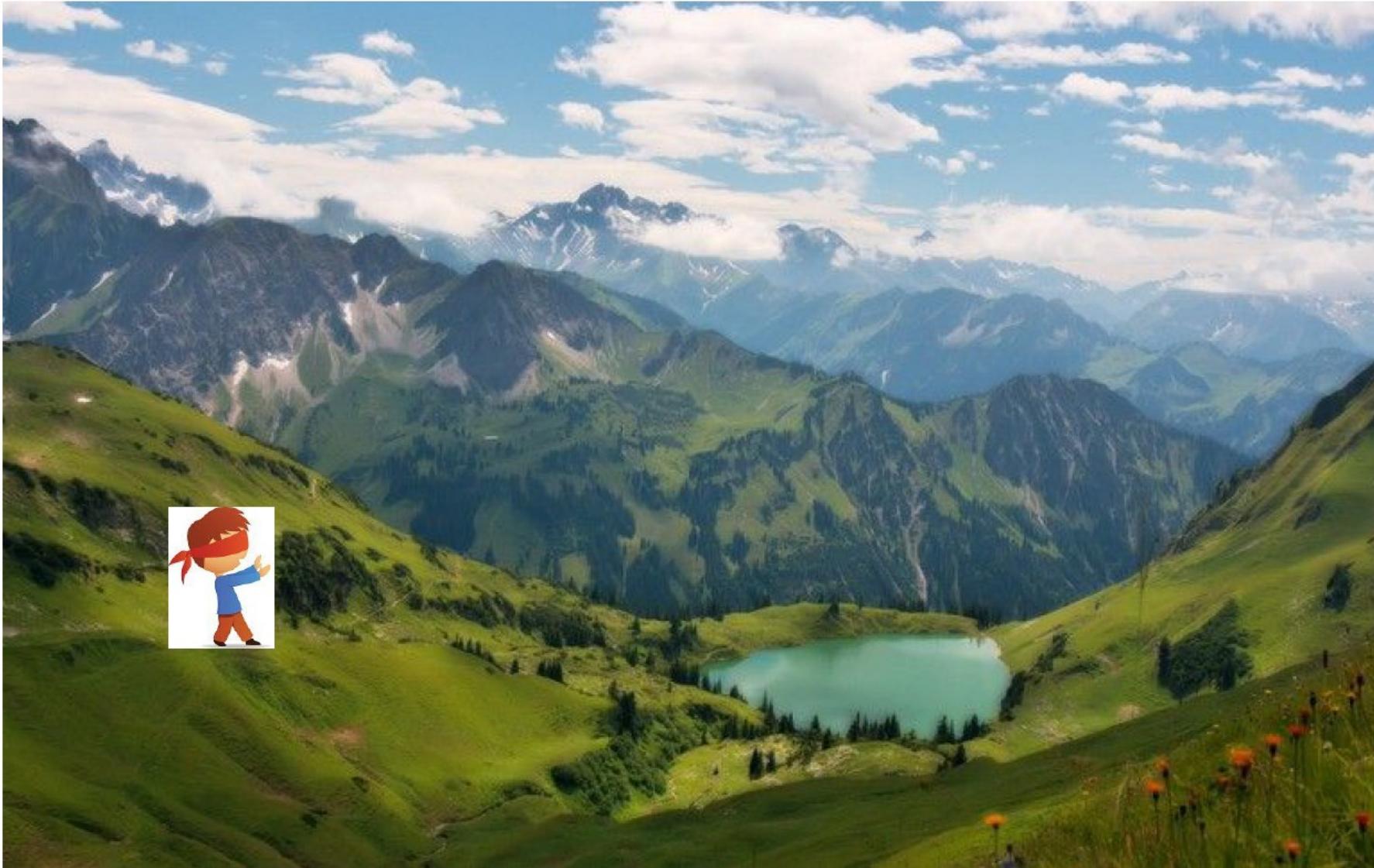


Now we will study

- Optimization Methods



Gradient Descent



Recap : Gradient Descent

- Batch gradient descent

$$\theta_t = \boxed{\theta_{t-1}} - \boxed{\eta} \cdot \boxed{\nabla_{\theta} J(\theta)}$$

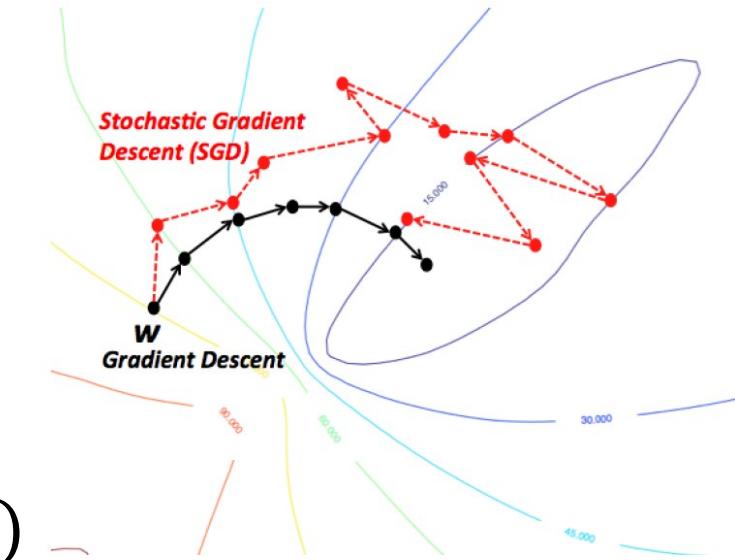
Previous parameter
Learning Rate

- Stochastic gradient descent

$$\theta_t = \theta_{t-1} - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)})$$

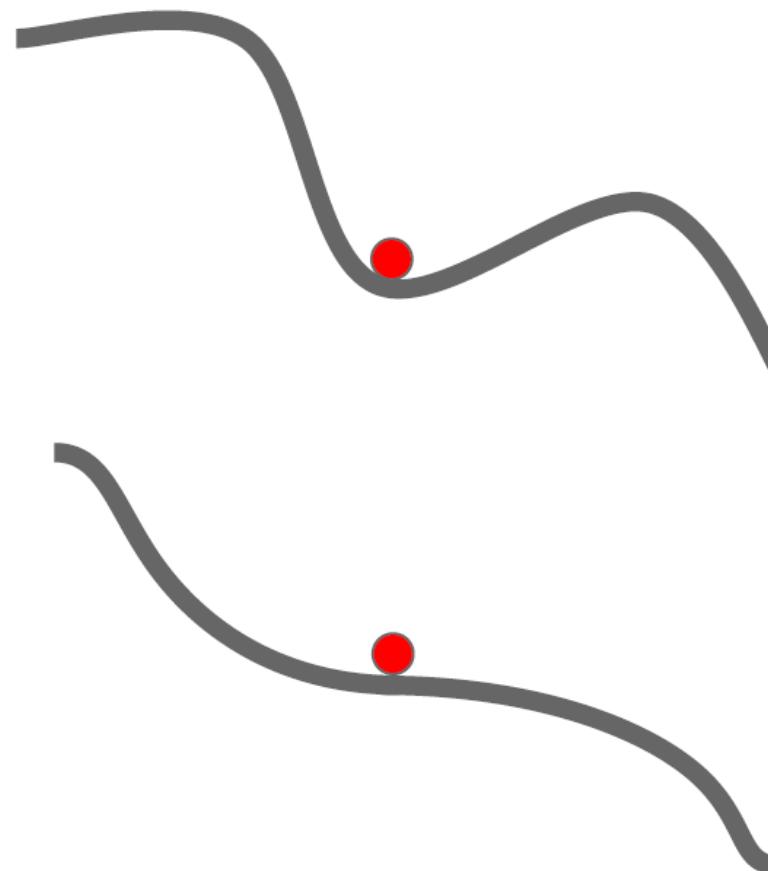
- Mini-batch gradient descent

$$\theta_t = \theta_{t-1} - \eta \cdot \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$



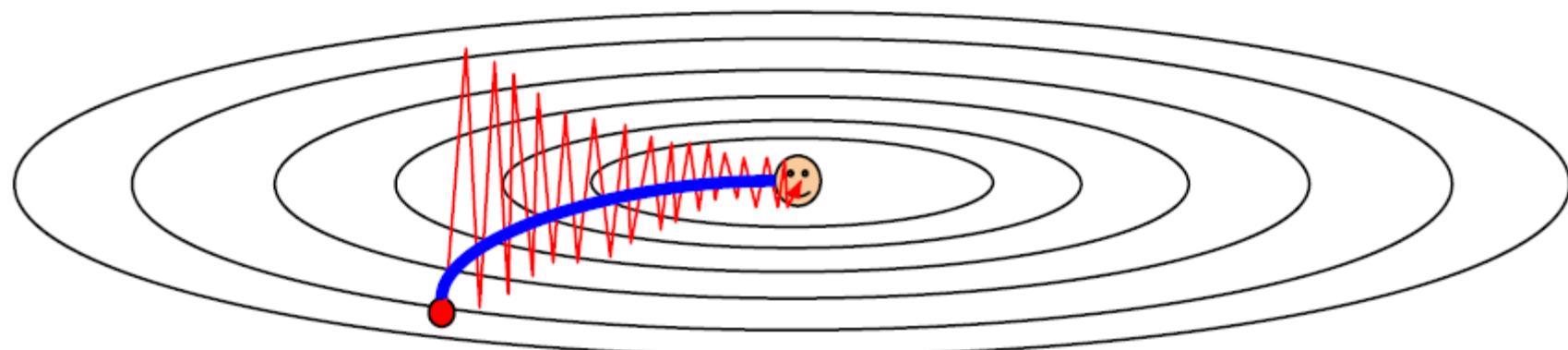
Problems of Gradient Descent

- (It can) stuck at local minima or saddle point(zero gradient)



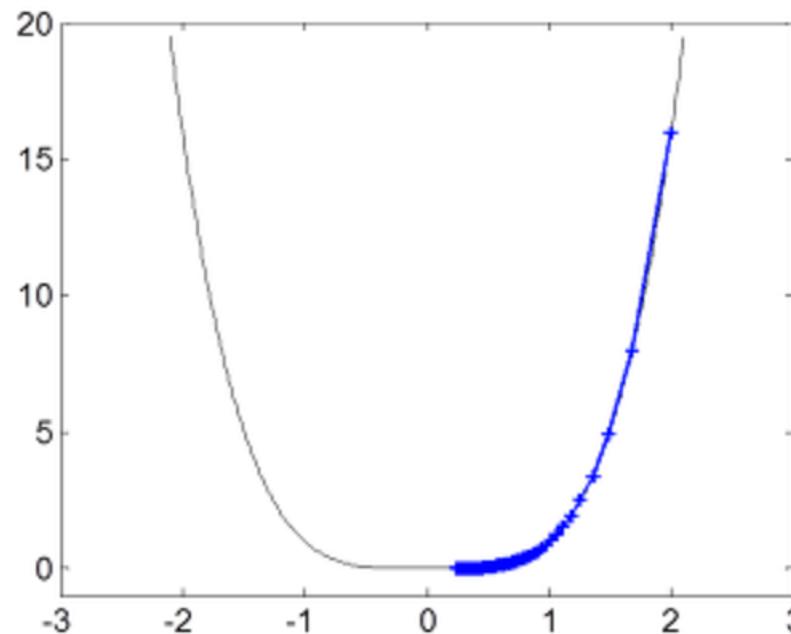
Problems of Gradient Descent

- Poor Conditioning



Problems of Gradient Descent

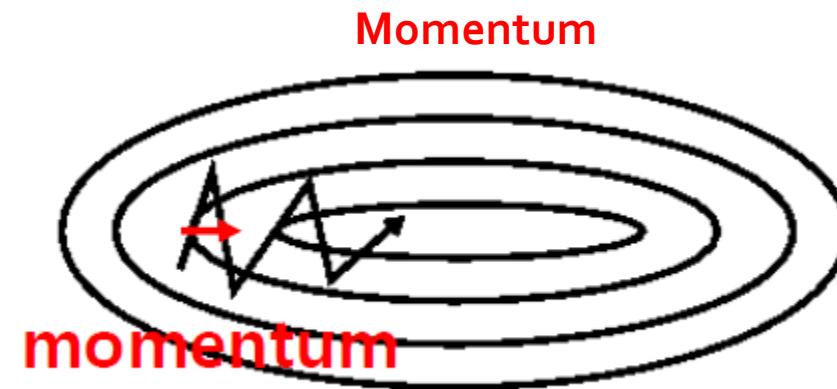
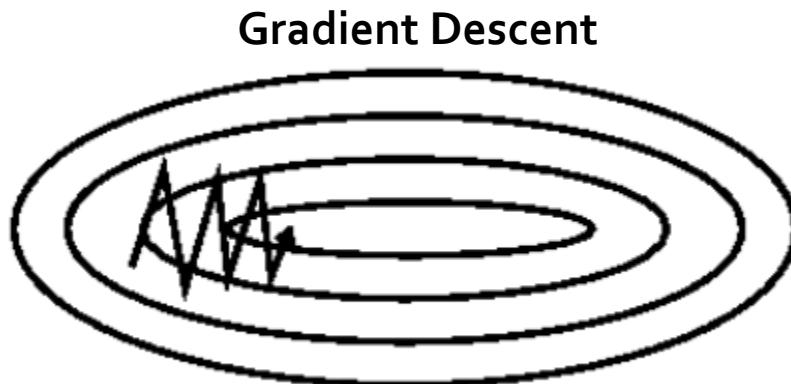
- Slow....
 - The closer to the optimal point, the smaller the gradient becomes.



Momentum

- Let's move with inertia in the direction that we moved earlier.

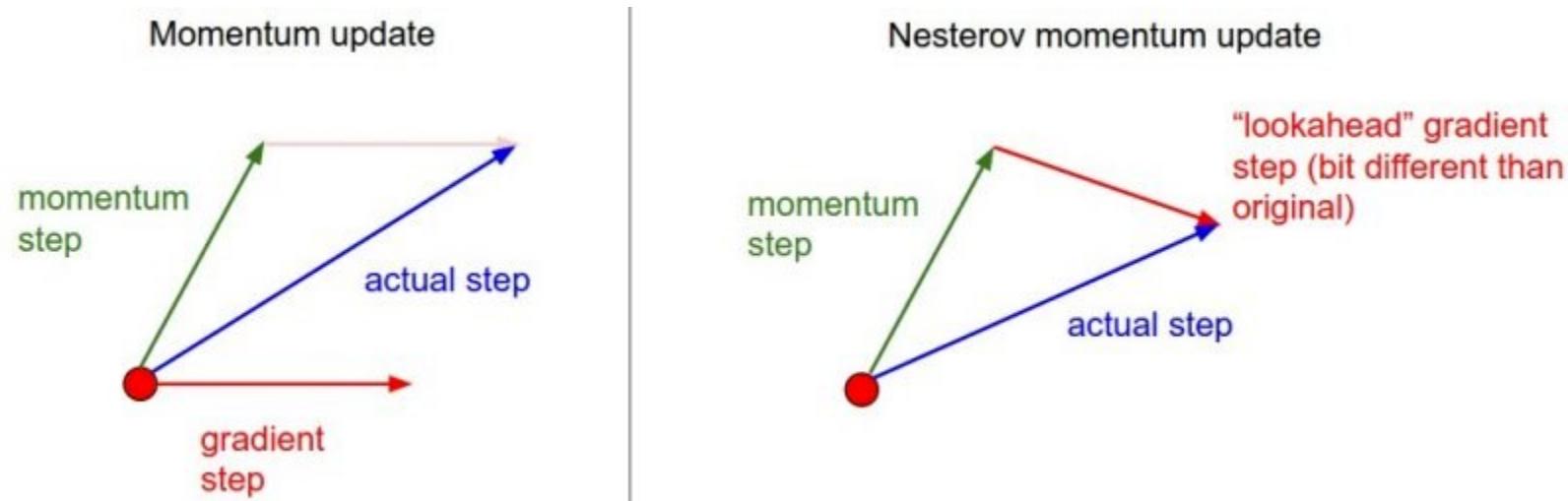
$$\begin{aligned}v_t &= \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta) \\ \theta_t &= \theta_{t-1} - v_t\end{aligned}$$



NAG(Nesterov Accelerated Gradient)

- Move in the direction we were previously moving, and then calculate the gradient from where we moved.

$$\begin{aligned}v_t &= \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1}) \\ \theta_t &= \theta_{t-1} - v_t\end{aligned}$$



Adagrad(Adaptive Gradient)

- It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.

$$G_t = G_{t-1} + \text{Element-wise Product} \\ (\nabla_{\theta} J(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_{\theta} J(\theta_t)$$

- As the learning continues, the G value continues to increase, so the step size becomes too small to learn

RMSprop

- Use exponentially decaying average of squared gradients

$$G_t = \gamma G_{t-1} + (1 - \gamma)(\nabla_{\theta} J(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_{\theta} J(\theta_t)$$

Adam(Adaptive Moment Estimation)

- Adaptive Moment Estimation (Adam) stores both exponentially decaying average of past gradients and squared gradients
- Combination of Momentum and RMSprop
- Compensate the initial momentum biased towards zero

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

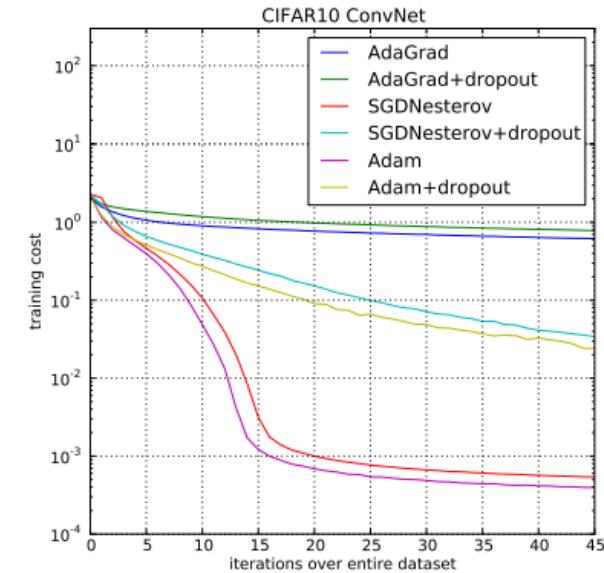
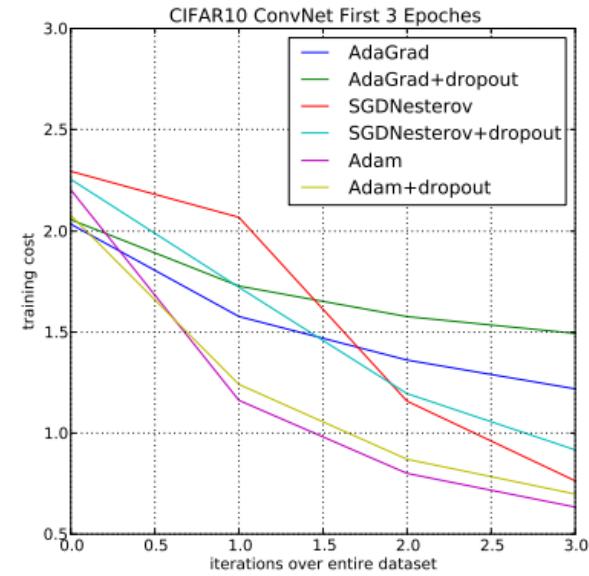
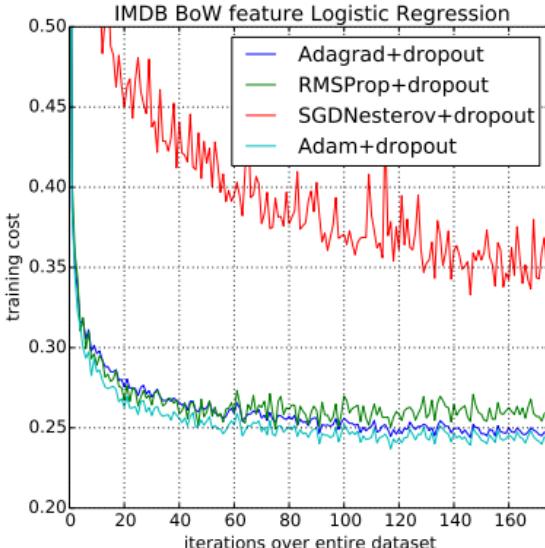
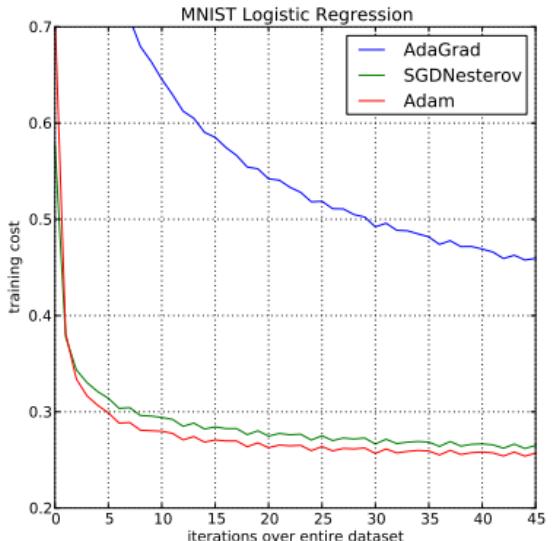
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\widehat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \widehat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

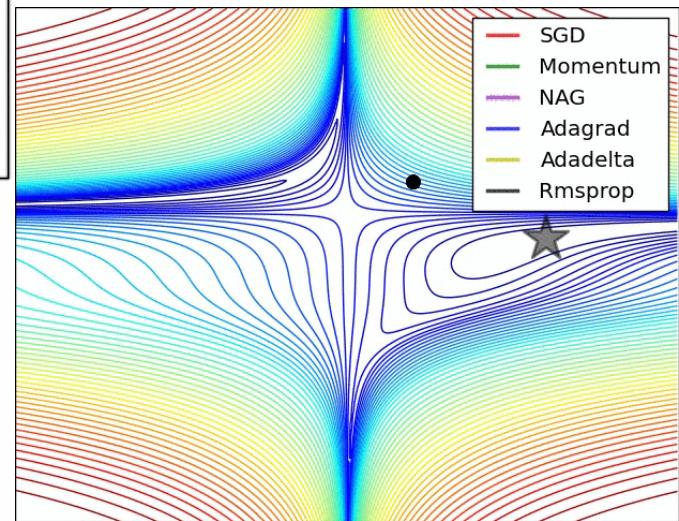
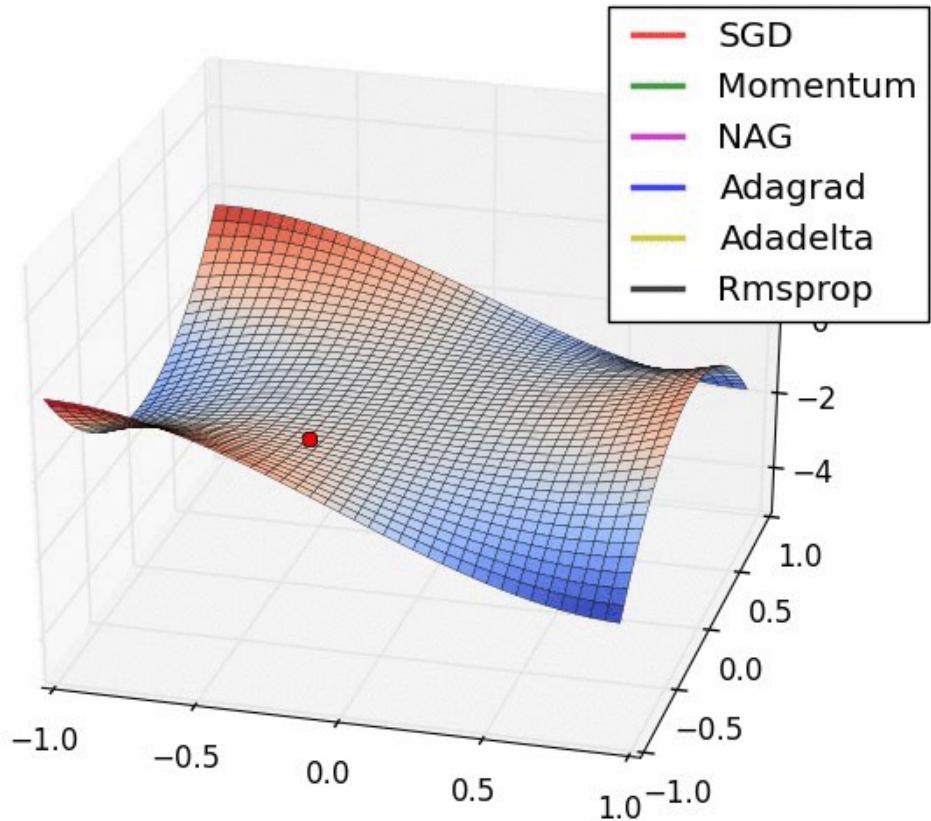
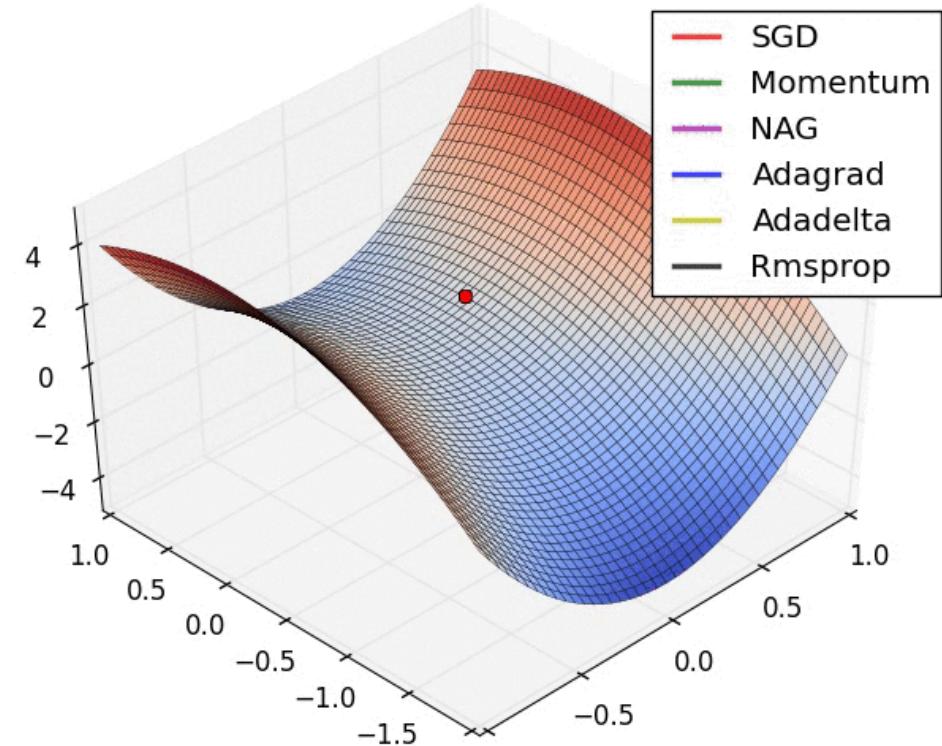
$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\widehat{v}_t} + \epsilon} \widehat{m}_t$$

Adam

- Adam is a very popular method



Optimization Methods



We Studied

- Optimization Methods

