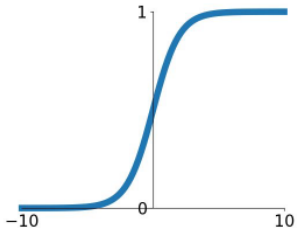
Last time

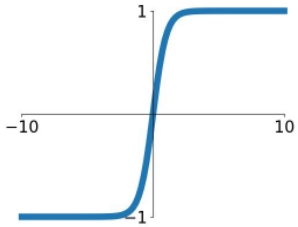
Sigmoid

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



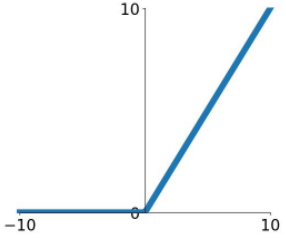
tanh

$$\tanh(x)$$



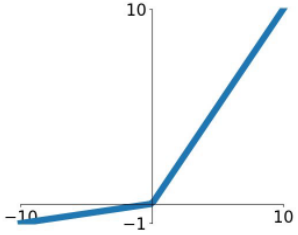
ReLU

$$\max(0, x)$$



Leaky ReLU

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

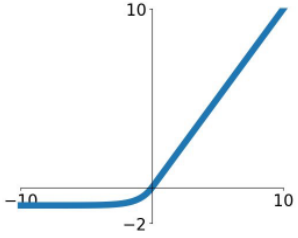


Maxout

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

ELU

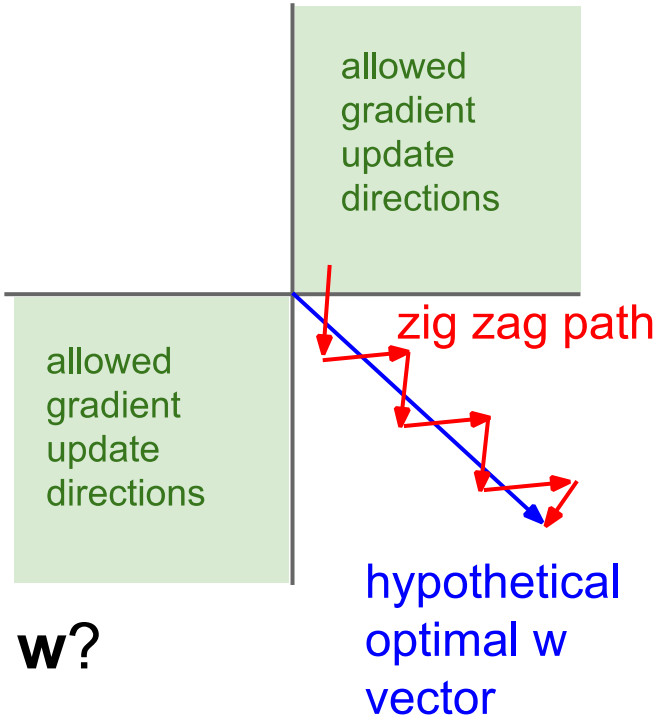
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Last time

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

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

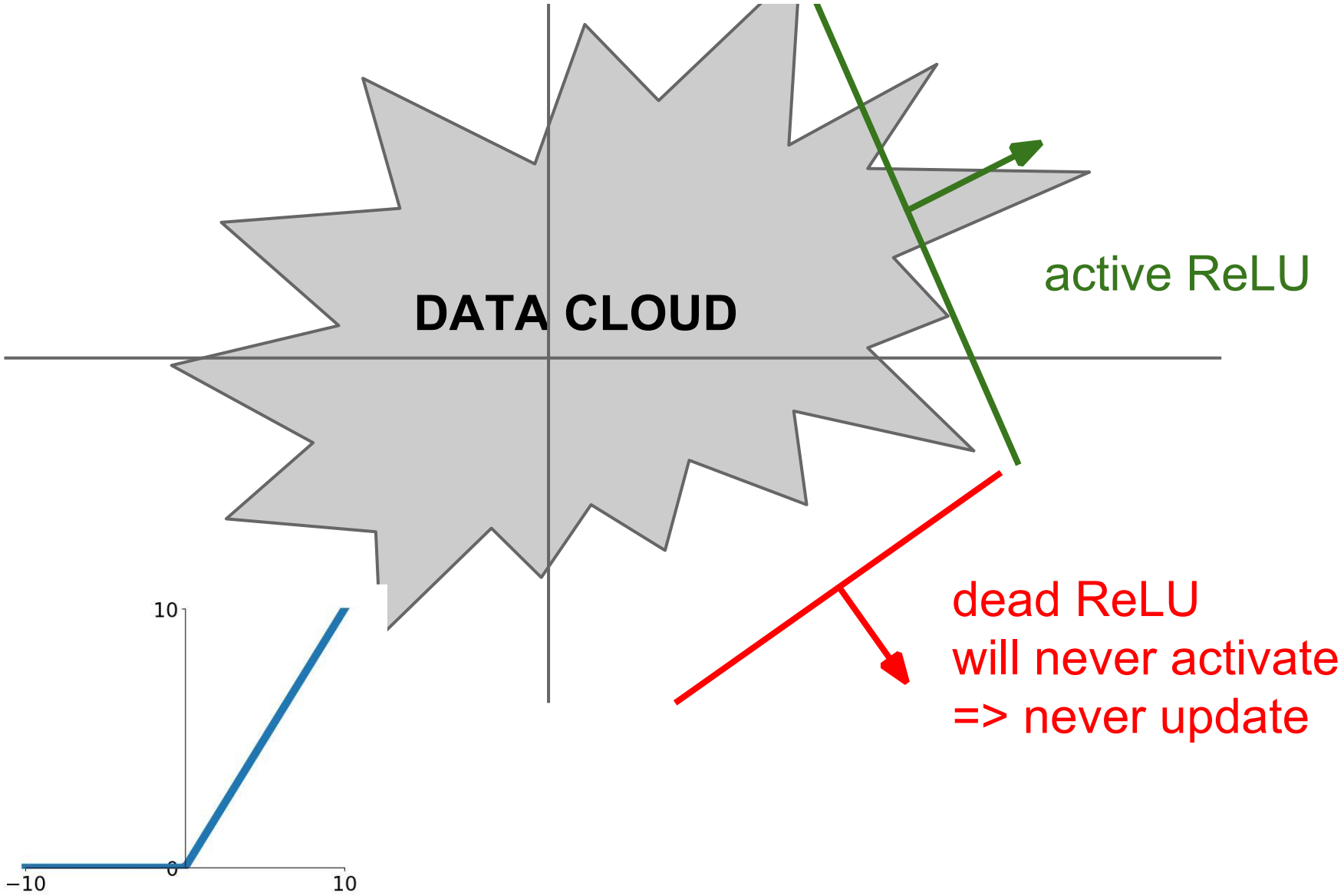


What can we say about the gradients on **w**?

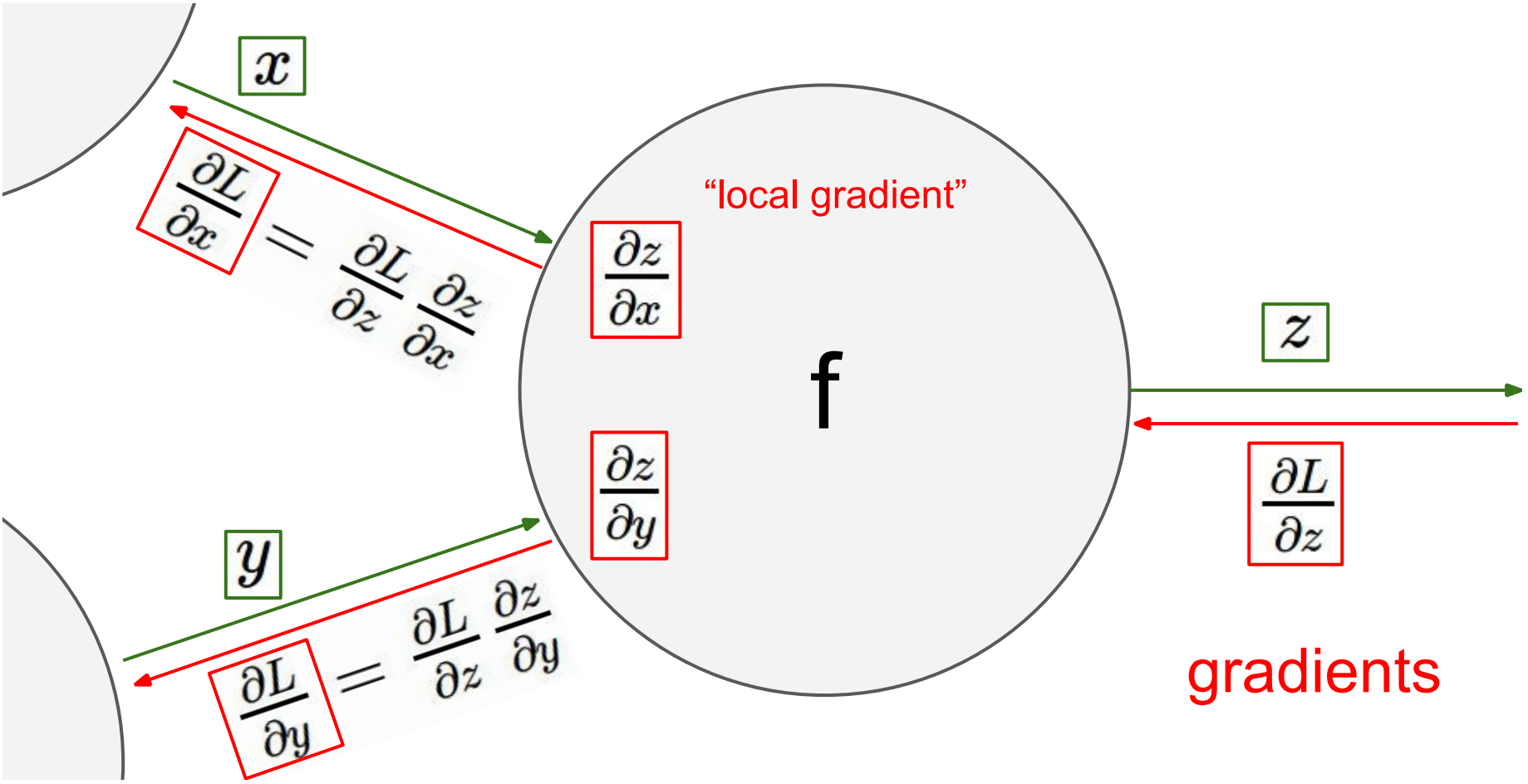
Always all positive or all negative :(

(this is also why you want zero-mean data!)

Last time


DATA CLOUD
active ReLU
dead ReLU
will never activate
=> never update
10
0
−10
10

Last time


x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
z
$\frac{\partial L}{\partial z}$
gradients

Lecture6
Training Neural Networks

Overview

1. **One time setup**
   activation functions, preprocessing, weight initialization, regularization, gradient checking
2. **Training dynamics**
   babysitting the learning process, parameter updates, hyperparameter optimization
3. **Evaluation**
   model ensembles

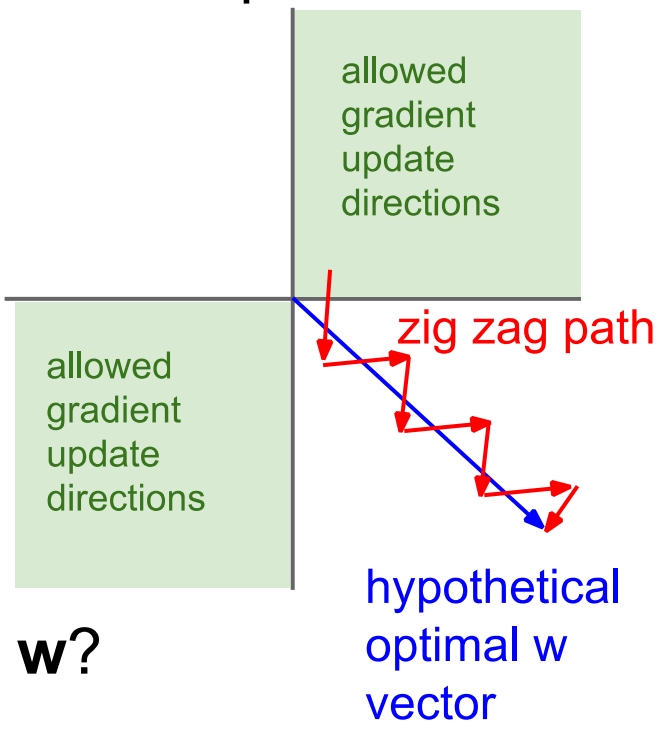
- Data preprocessing
- Data augmentation
- Weight initialization
- Weight decay
- Dropout
- Batch Normalization
- Optimization
- Babysitting the learning process
- Hyper-parameter optimization
- Model Ensembles

Data Preprocessing

Data preprocessing

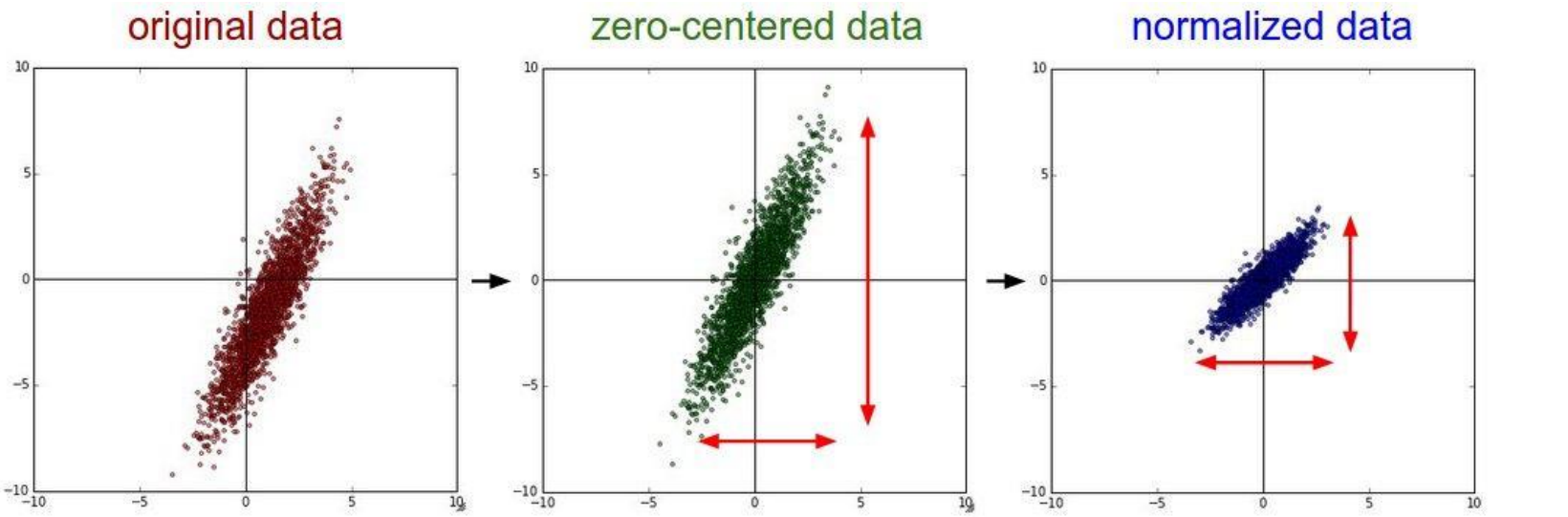
Remember: Consider what happens when the input to a neuron is always positive...

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



What can we say about the gradients on **w**?
Always all positive or all negative :(
(this is also why you want zero-mean data!)

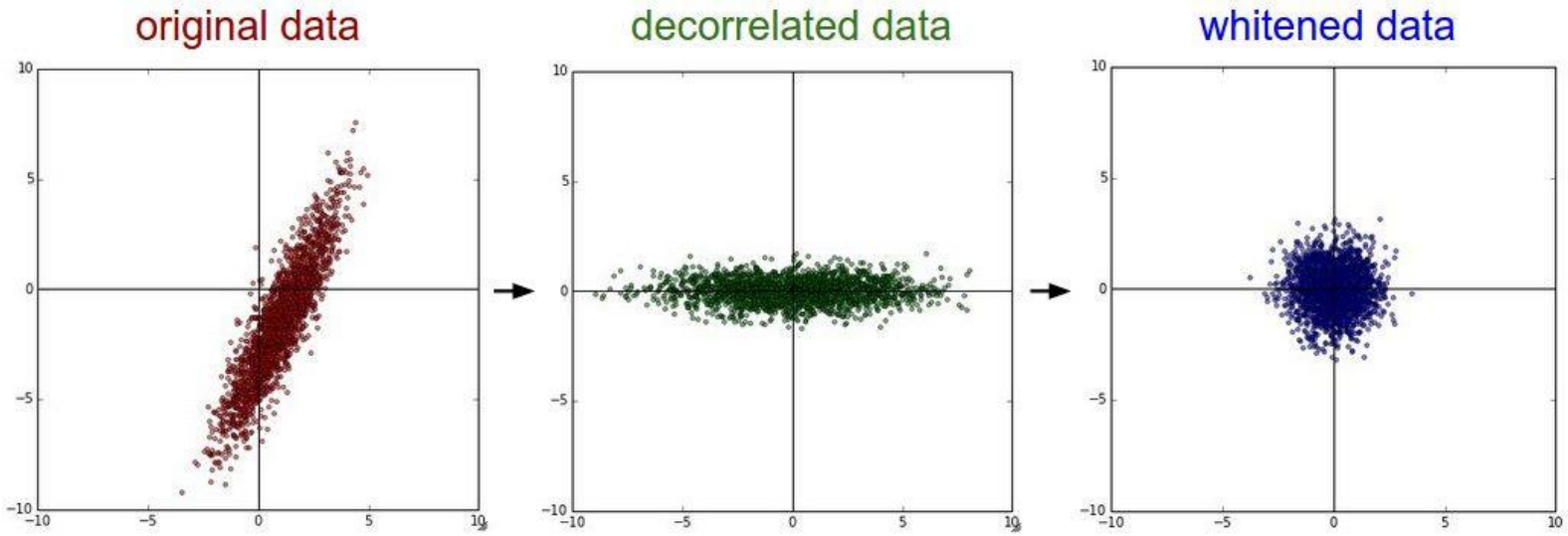
Data preprocessing



(Assume X [NxD] is data matrix,
each example in a row)

Data preprocessing

In practice, you may also see **PCA** and **Whitening** of the data

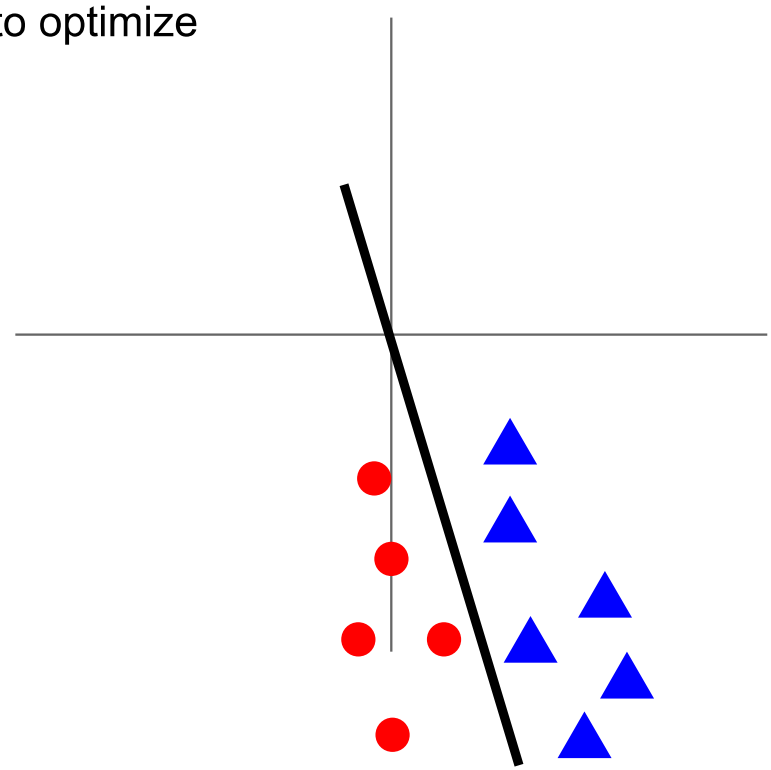


(data has diagonal covariance matrix)

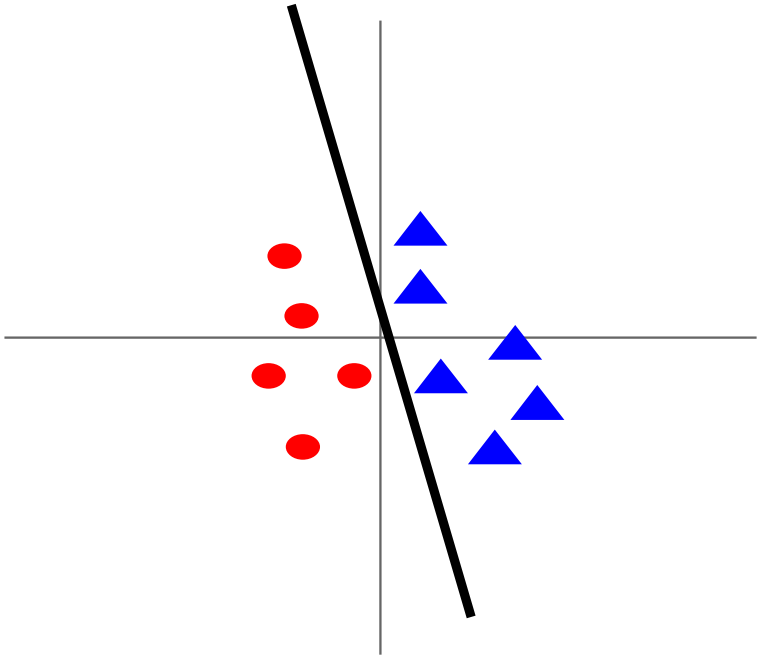
(covariance matrix is the identity matrix)

Data preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Data preprocessing

TLDR: In practice for Images: center only

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

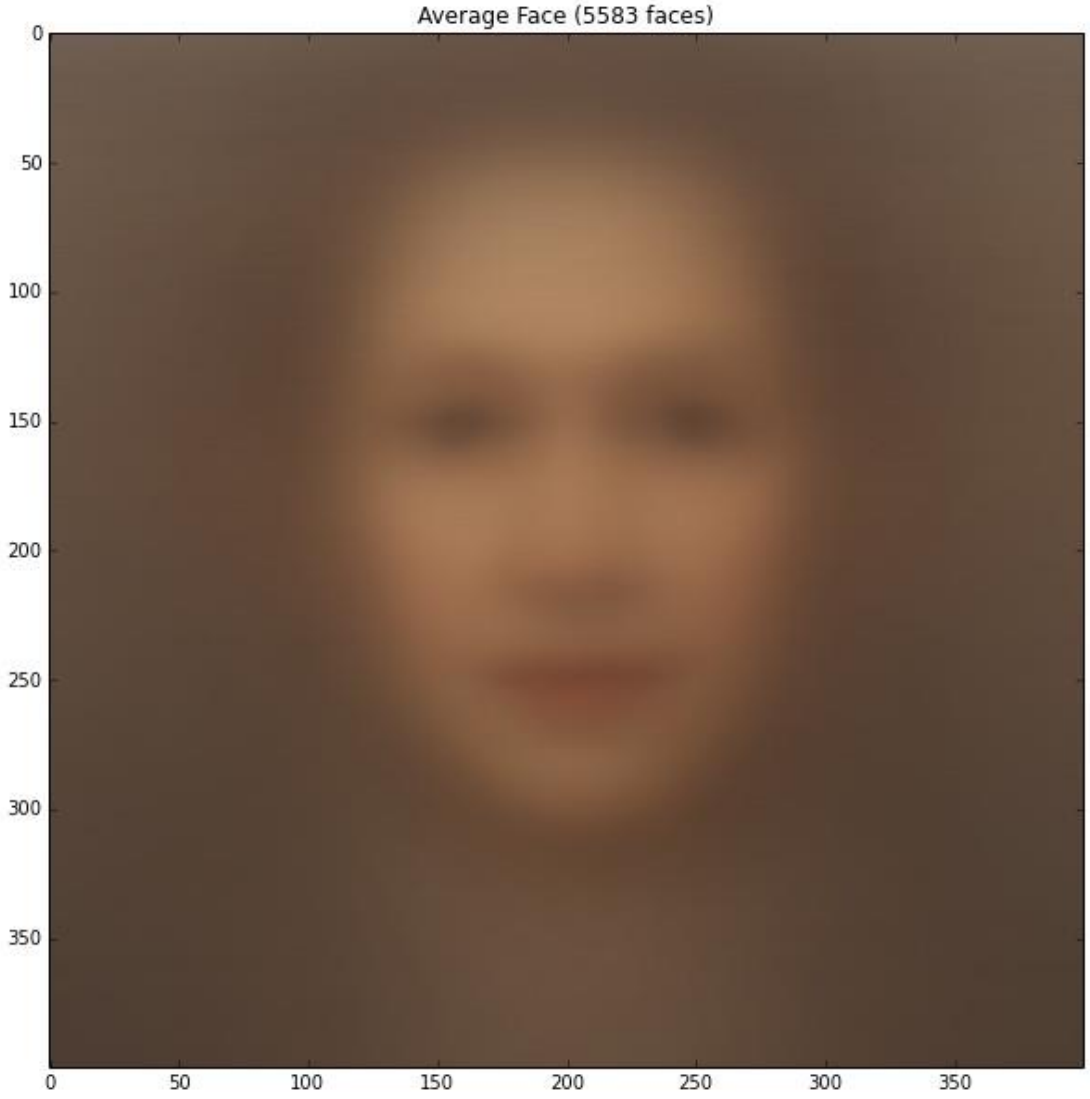
- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

- Data preprocessing should be done in both training and test phases.
- During test, we use the empirical mean estimated on the training data, to keep the mode evaluation as the same.

Not common to normalize variance, to do PCA or whitening

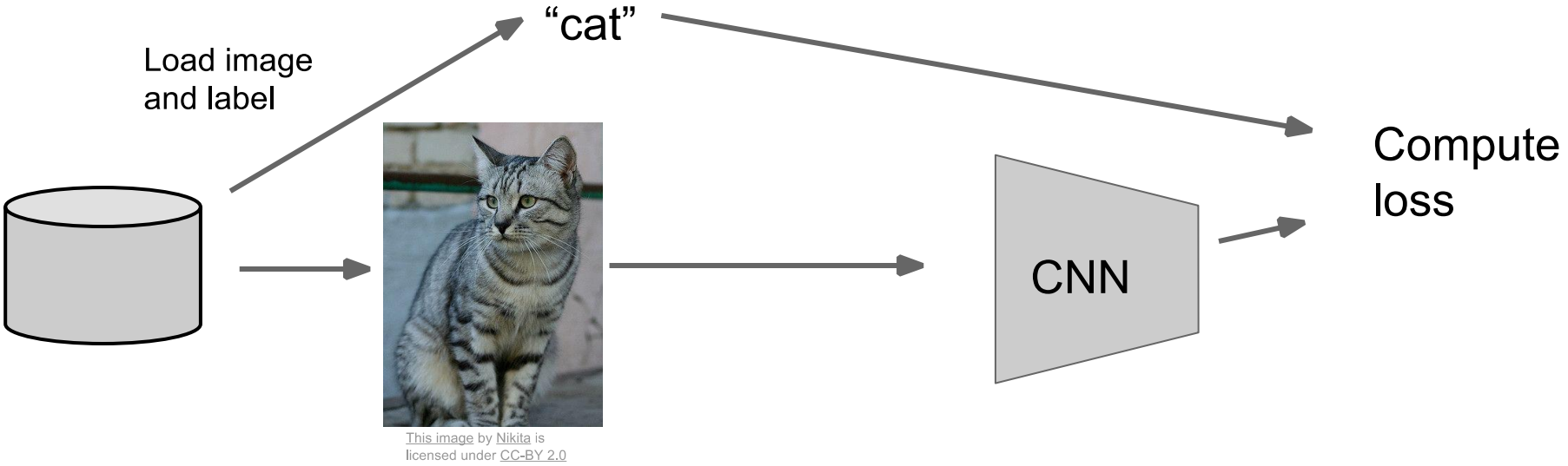
Q: Does data preprocessing solve the zero-mean problem caused by sigmoid?

Data preprocessing

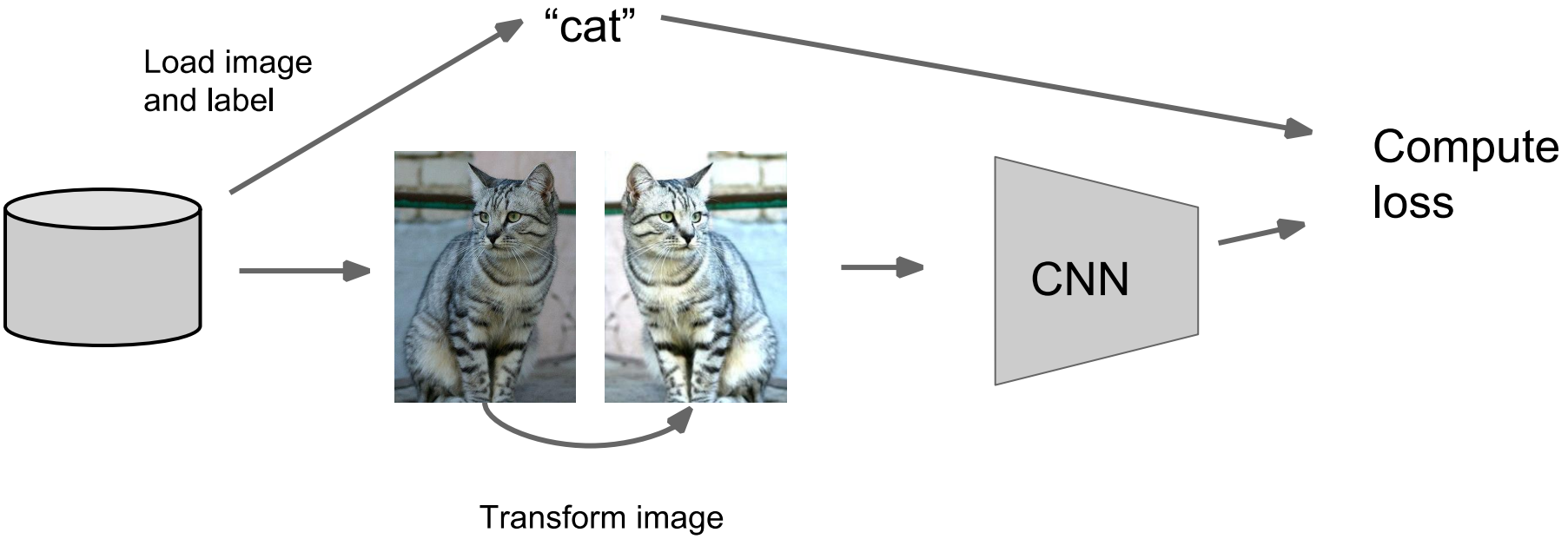


Data Augmentation

Data augmentation

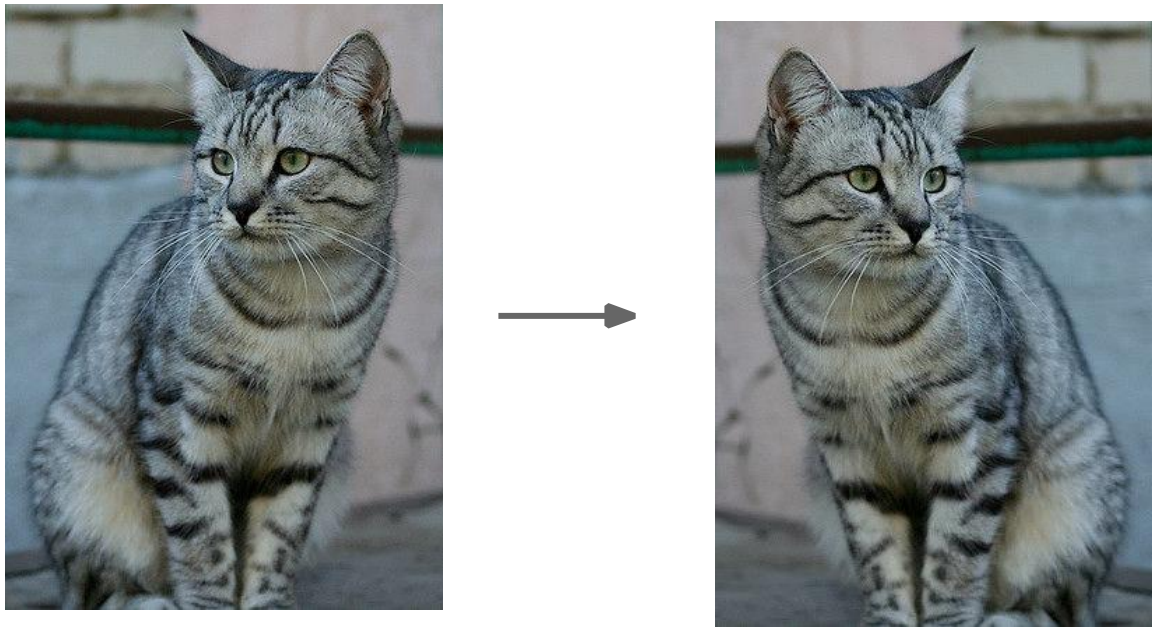


Data augmentation



Data augmentation

Horizontal Flips



How about vertical flip?

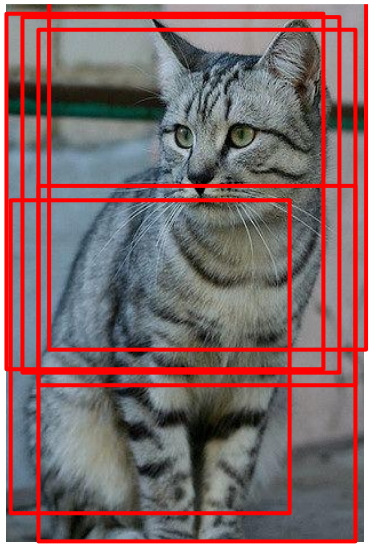
Data augmentation

Random crops and scales

Training: sample random crops / scales

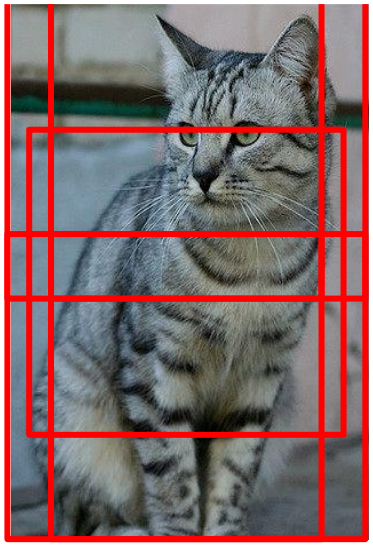
ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Data augmentation

Random crops and scales



Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

Testing: average a fixed set of crops

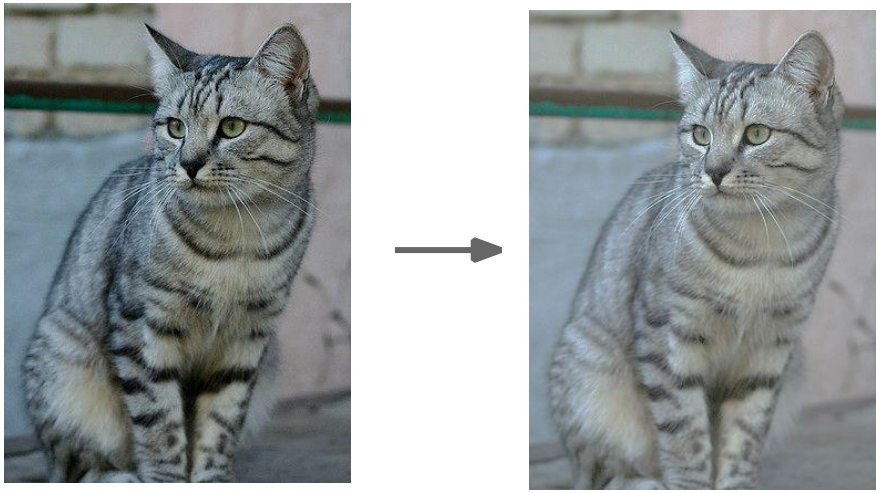
ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data augmentation

Color Jitter

Simple: Randomize
contrast and brightness



Data augmentation

Get creative for your problem!

(a) Original

(b) Crop and resize

(c) Crop, resize (and flip)

(d) Color distort. (drop)

(e) Color distort. (jitter)

(f) Rotate $\{90°, 180°, 270°\}$

(g) Cutout

(h) Gaussian noise

(i) Gaussian blur

(j) Sobel filtering

Figures from Simclr

Data augmentation

Data augmentation is more important for self-supervised contrastive learning

Figure 5. Linear evaluation (ImageNet top-1 accuracy) under individual or composition of data augmentations, applied only to one branch. For all columns but the last, diagonal entries correspond to single transformation, and off-diagonals correspond to composition of two transformations (applied sequentially). The last column reflects the average over the row.

case	unsup. pre-train MLP	aug+	cos	epochs	ImageNet acc.	VOC detection AP_{50}	AP	AP_{75}
supervised					76.5	81.3	53.5	58.8
MoCo v1				200	60.6	81.5	55.9	62.6
(a)	✓			200	66.2	82.0	56.4	62.6
(b)		✓		200	63.4	82.2	56.8	63.2
(c)	✓	✓		200	67.3	**82.5**	57.2	63.9
(d)	✓	✓	✓	200	67.5	82.4	57.0	63.6
(e)	✓	✓	✓	**800**	**71.1**	**82.5**	**57.4**	**64.0**

Table 1. **Ablation of MoCo baselines**, evaluated by ResNet-50 for (i) ImageNet linear classification, and (ii) fine-tuning VOC object detection (mean of 5 trials). "**MLP**": with an MLP head; "**aug+**": with extra blur augmentation; "**cos**": cosine learning rate schedule.

SIMCLR

Composition of multiple data augmentation operations is crucial in defining the contrastive prediction tasks that yield effective representations. In addition, unsupervised contrastive learning benefits from stronger data augmentation than supervised learning

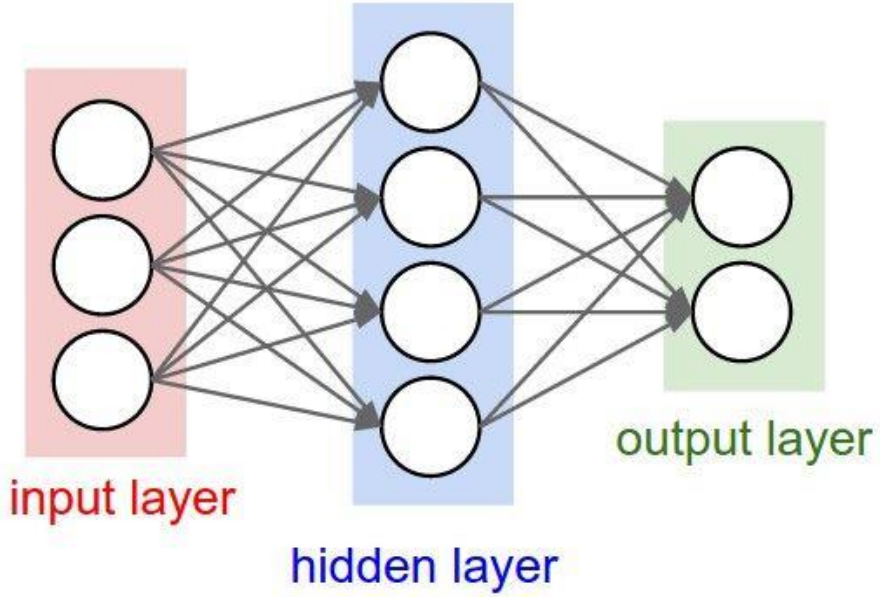
MOCO-V2

With simple modifications to MoCo—namely, using an MLP projection head and more data augmentation—we establish stronger baselines that outperform SimCLR and do not require large training batches.

Weight Initialization

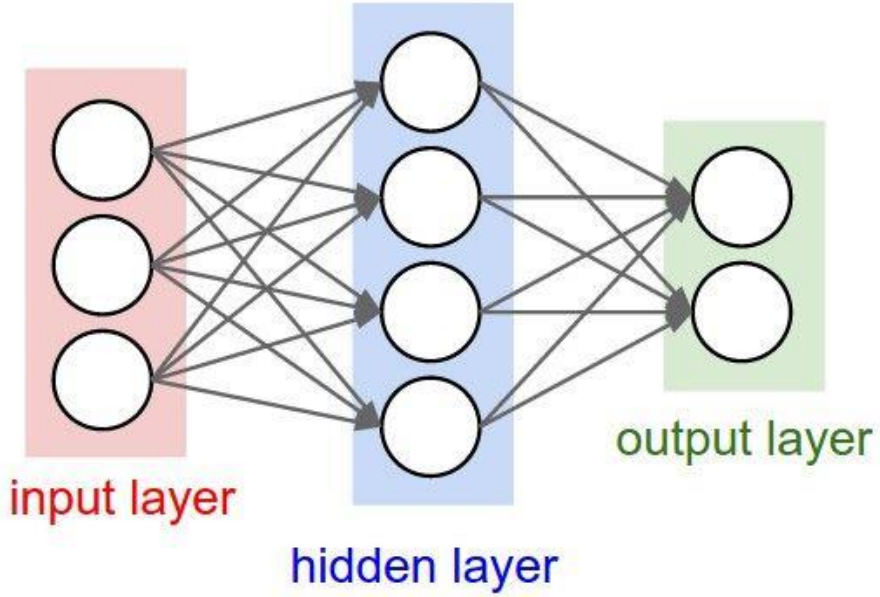
Weight initialization

Q: what happens when W is uniformly initialized?



Weight initialization

Q: what happens when W is uniformly initialized?
A: All the neurons with the same input will get the same gradient and update in the same way.



Weight initialization

- First idea: **Small random numbers**

(gaussian with zero mean and 1e-2 standard deviation)

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

Weight initialization

- First idea: **Small random numbers**

(gaussian with zero mean and 1e-2 standard deviation)

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

Works ~okay for small networks, but problems with deeper networks.

Weight initialization

Lets look at some activation statistics

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

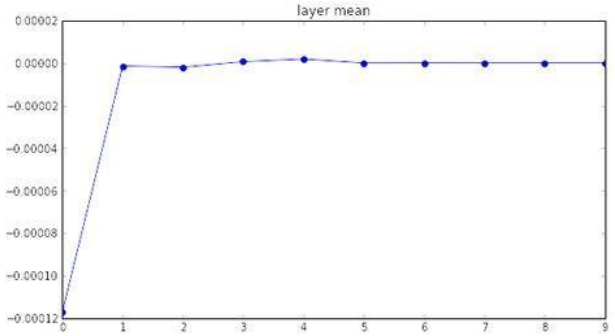
```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

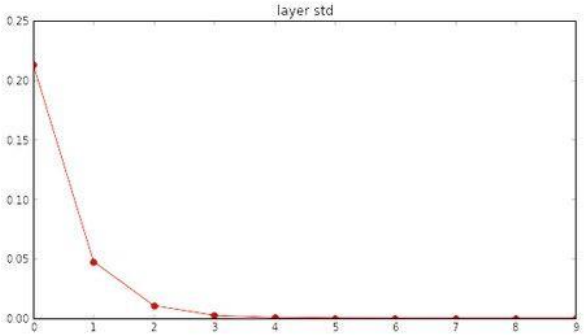
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

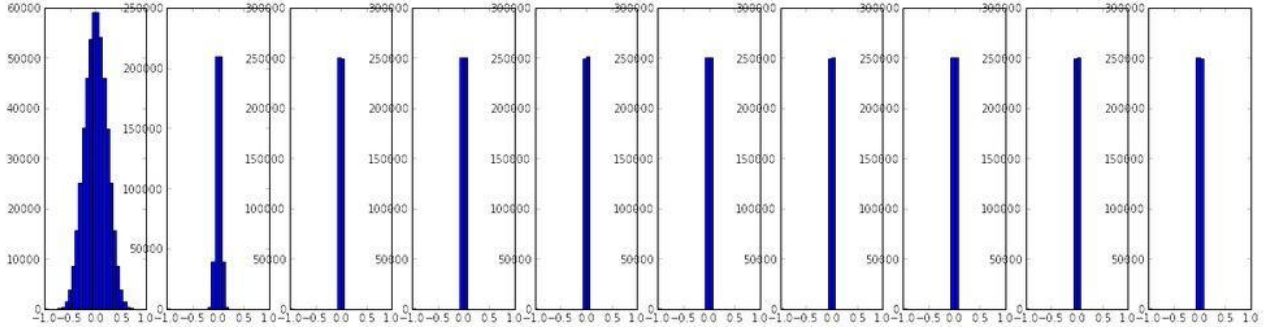
```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

Weight initialization

```
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
```

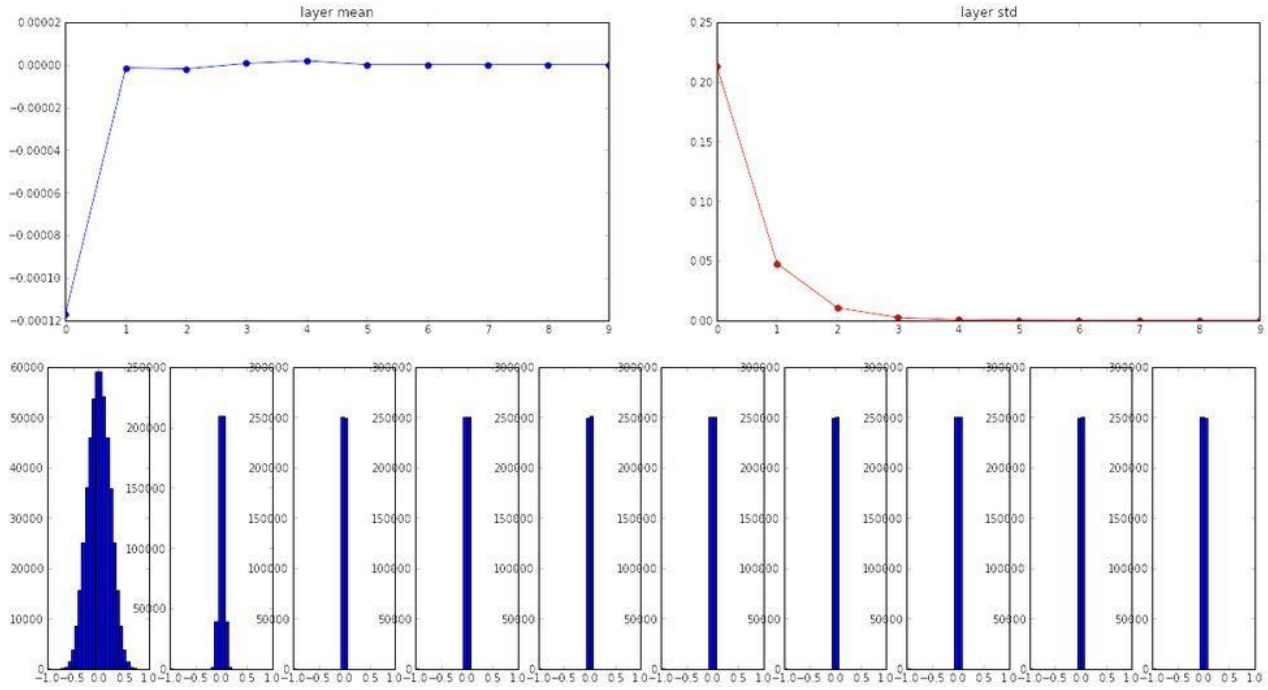






Weight initialization

```
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?

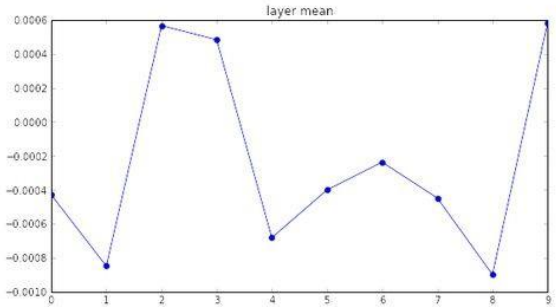
Hint: think about backward pass for a W*X gate.

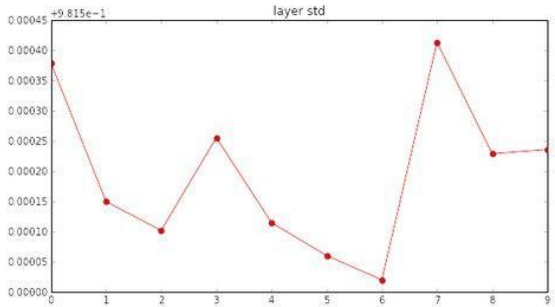
Weight initialization

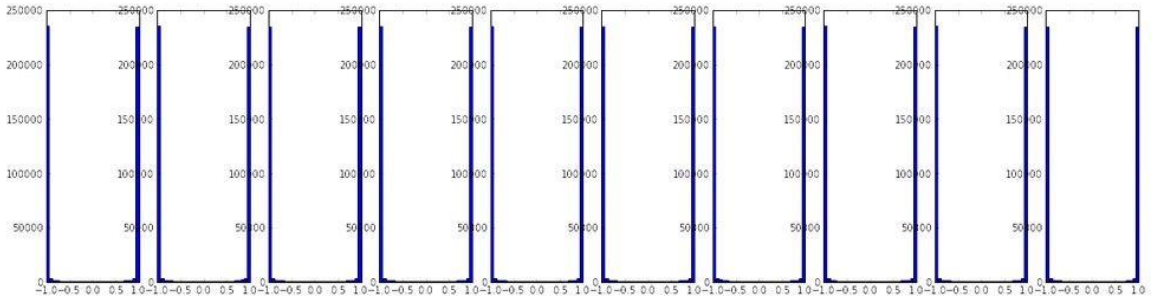
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
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.

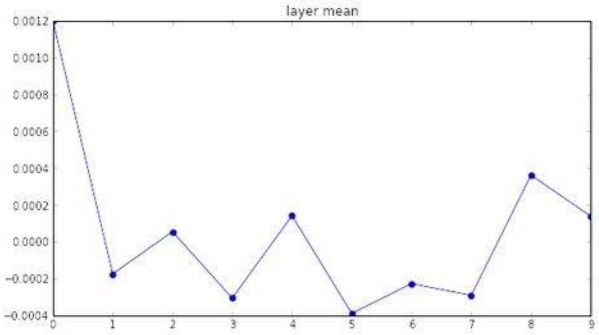
Weight initialization

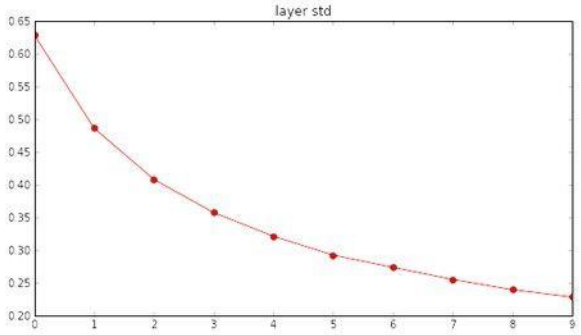
Divided by the number of inputs

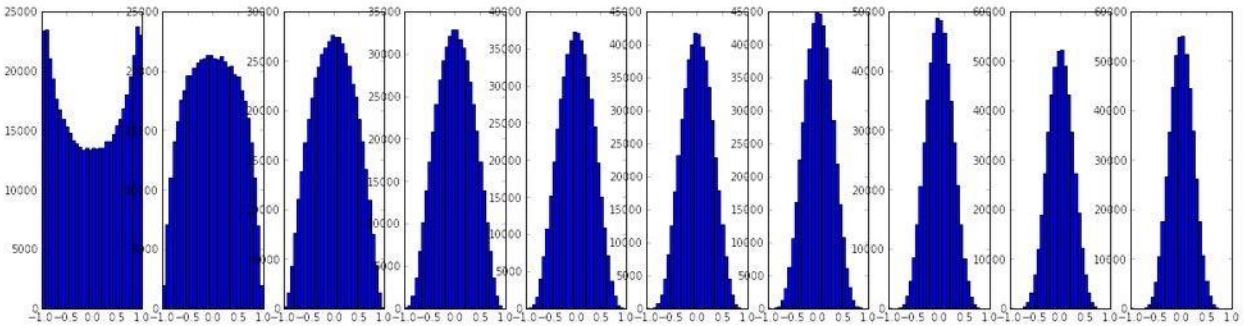
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
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
```

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







Reasonable initialization.
(Mathematical derivation assumes linear activations)

$$\begin{aligned}\mathrm{Var}[a^l] &= \mathrm{Var}[\sum_{i=1}^{n^{(l-1)}} w_i^l a_i^{(l-1)}] \\ &= \sum_{i=1}^{n^{(l-1)}} \mathrm{Var}[w_i^l]\,\mathrm{Var}[a_i^{(l-1)}] \\ &= n^{(l-1)}\,\mathrm{Var}[w_i^l]\,\mathrm{Var}[a_i^{(l-1)}].\end{aligned}$$

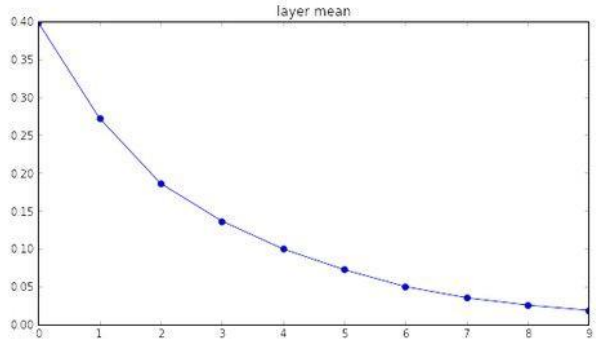
Should equal to 1 if we want the variance of inputs equals to the variance of outputs

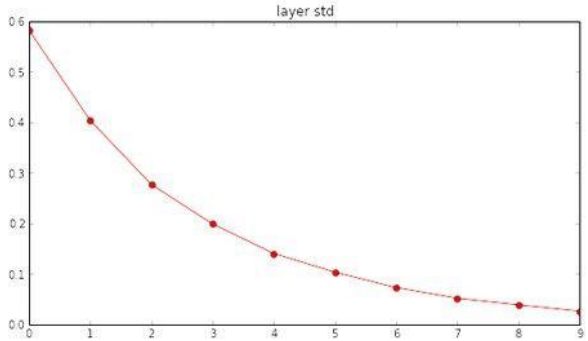
Weight initialization

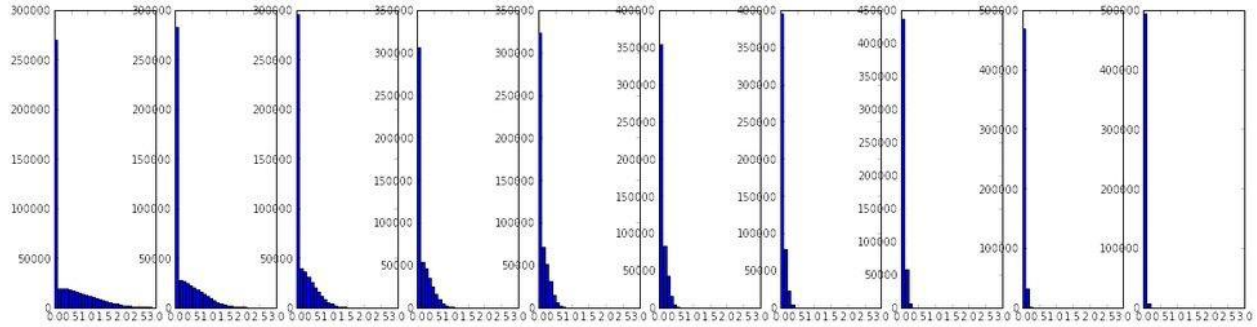
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
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
```

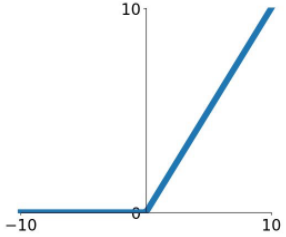
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.









Assume half of your linear neuron outputs are smaller than zero, then half of the corresponding activation outputs will be zero

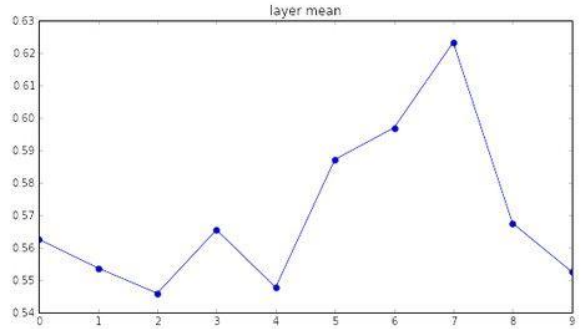
Weight initialization

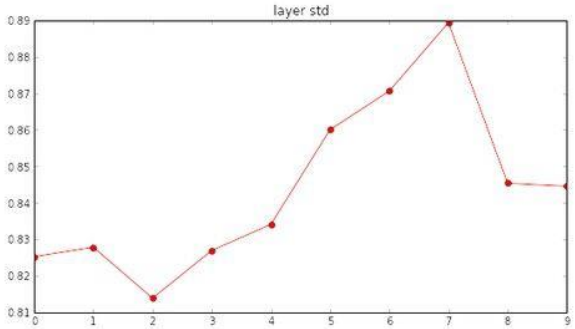
A simple solution:
Multiply the output variance by 2

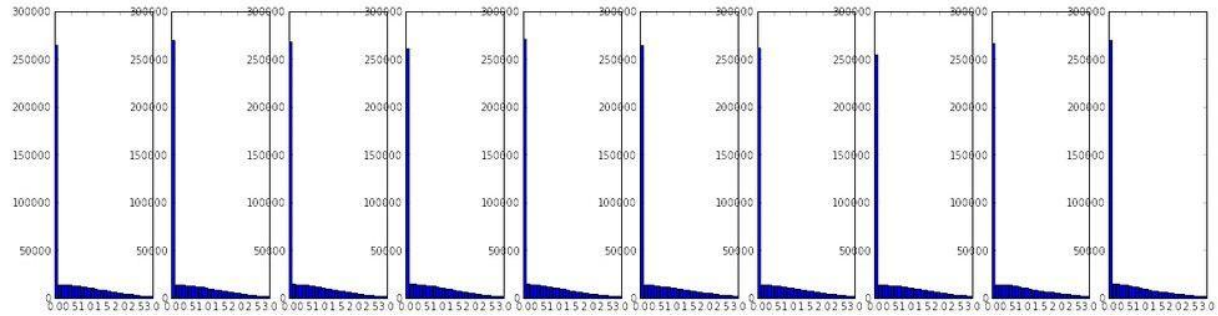
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
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(fan_in/2) # layer initialization
```

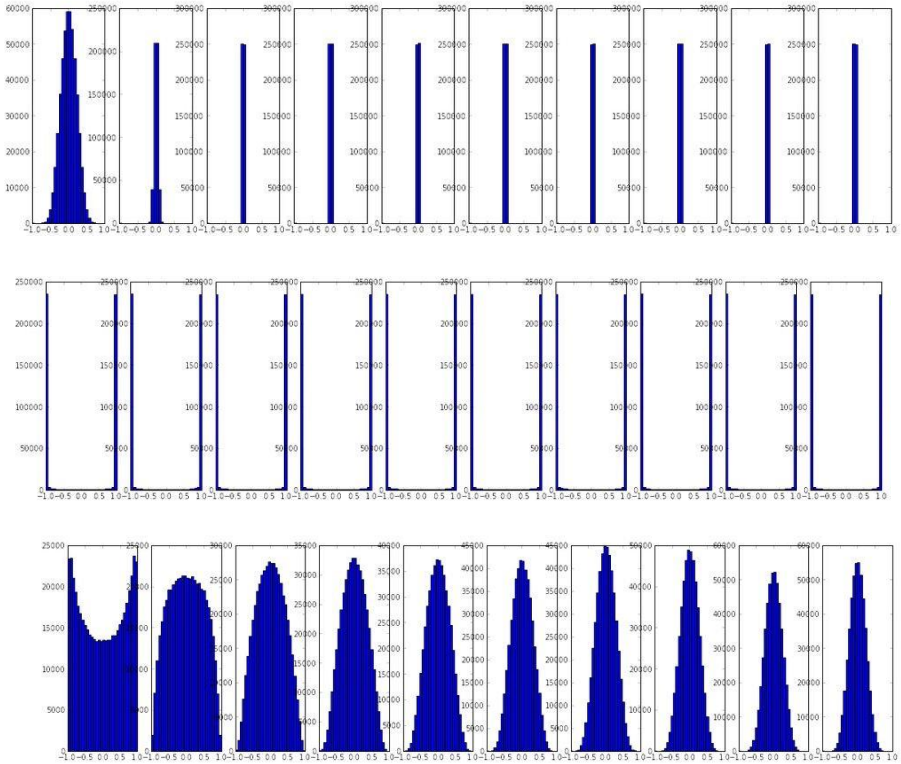
He et al., 2015
(note additional /2)







Weight initialization

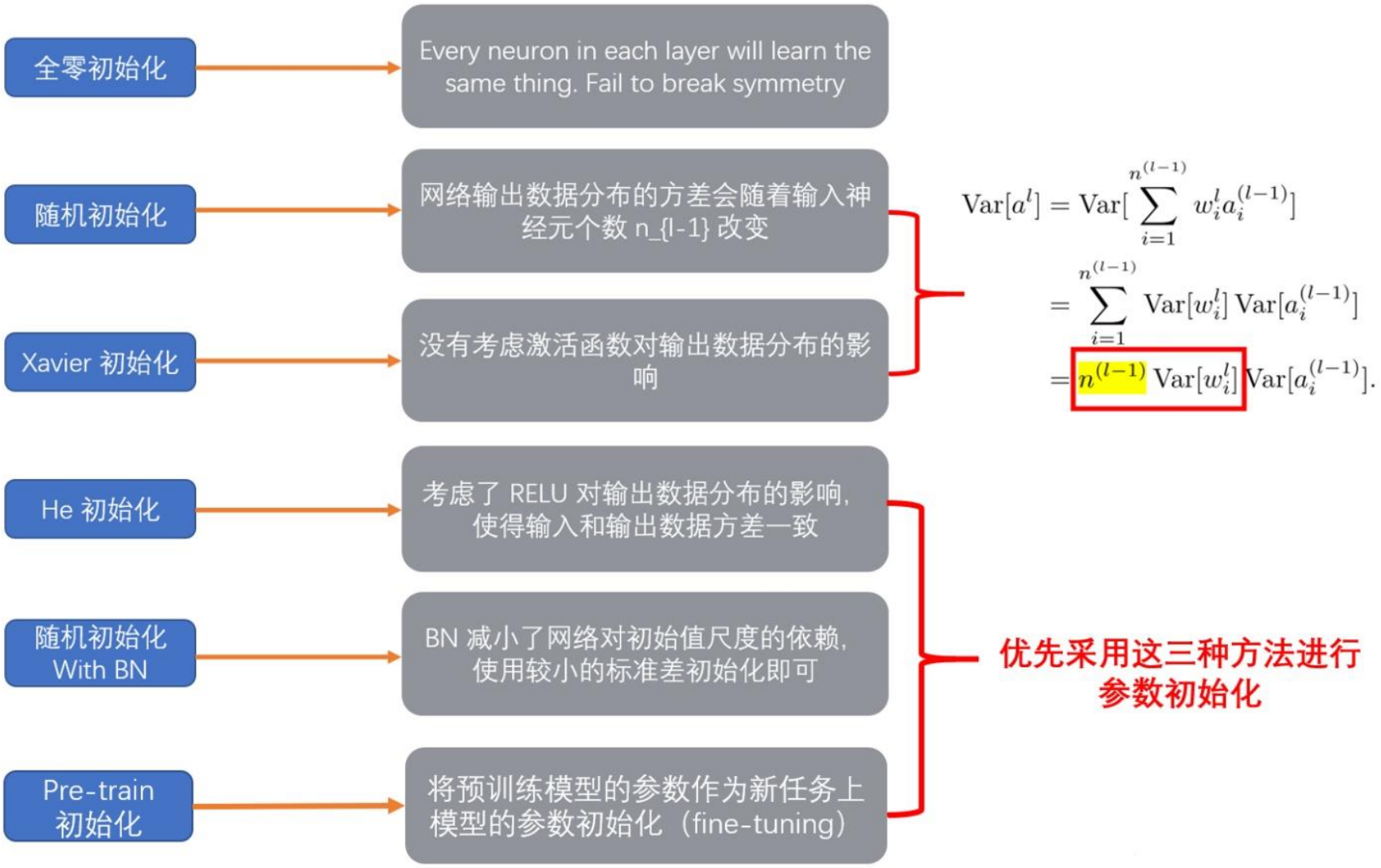


Initialization too small:
Activations go to zero, gradients also zero,
No learning

Initialization too big:
Activations saturate (for tanh),
Gradients zero, no learning

Initialization just right:
Nice distribution of activations at all layers,
Learning proceeds nicely

Weight initialization



Weight Decay

Recall: regularization in a narrow sense

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Model should be “simple”, so it works on test data

Weight decay [A. Krogh and J. A. Hertz, NIPS, 1992]

Consider the following objective function:

$$L(\mathbf{W}) = L_0(\mathbf{W}) + \frac{1}{2}\lambda||\mathbf{W}||_2^2$$

where E0 is one's favorite error measure, and lambda is the parameter governing how strongly large weights are penalized. The corresponding weight update rule is

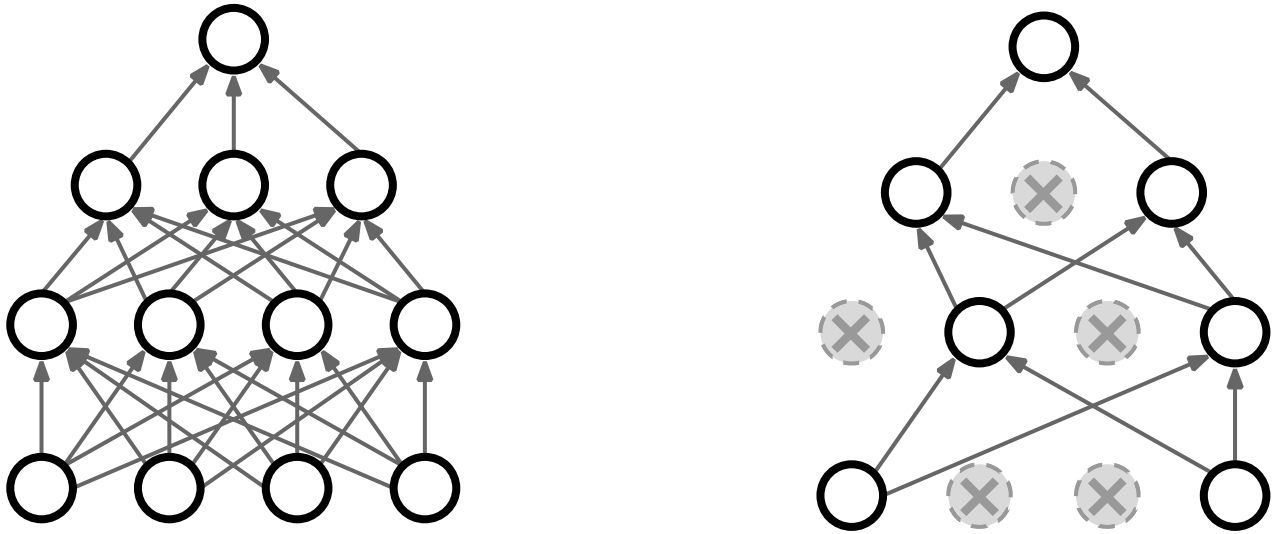
$$\mathbf{W} \leftarrow \mathbf{W} - \eta\frac{\partial L_0}{\partial \mathbf{W}} - \eta\lambda\mathbf{W}$$

Cause the weights to decay in proportion to its size

Dropout

Dropout

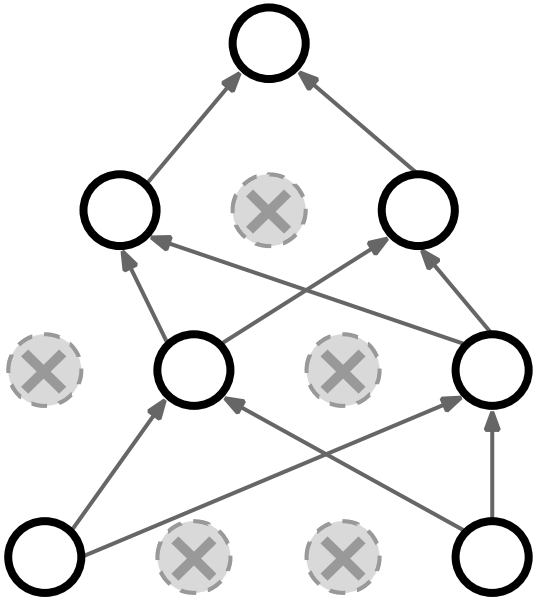
In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Dropout

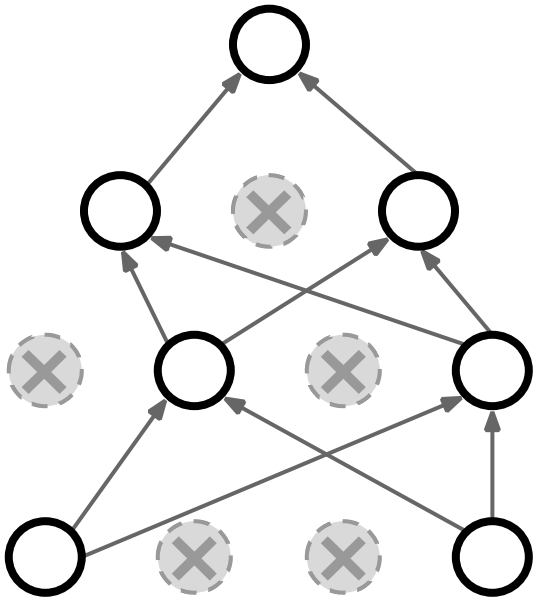
How can this possibly be a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features

Dropout

How can this possibly be a good idea?



Another interpretation:

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

.

Output (label) Input (image)

Dropout makes our output random! $y = f_W(x, z)$ Random mask

Want to "average out" the randomness at test-time

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

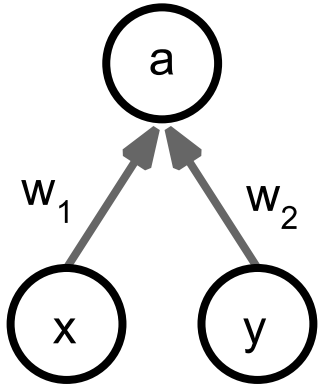
But this integral seems hard …

Dropout

Want to approximate the integral

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

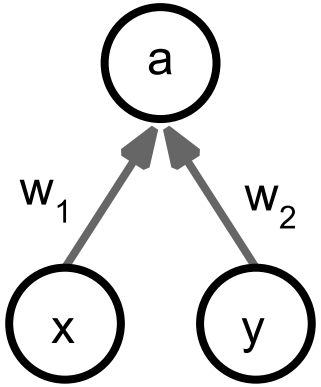
Consider a single neuron.



Dropout

Want to approximate the integral

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



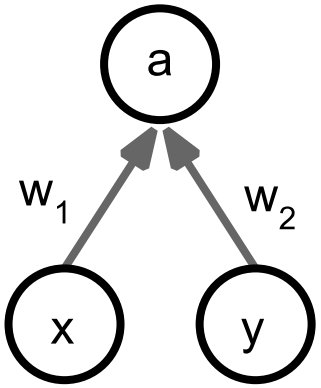
Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

Dropout

Want to approximate the integral

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



Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

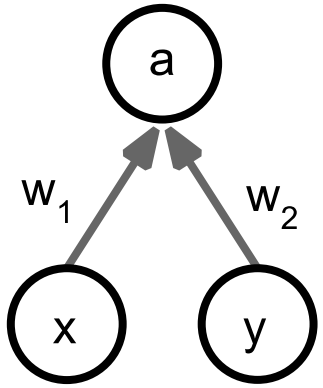
During training we have:

$$\begin{aligned} E\big[a\big] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

Dropout

Want to approximate the integral

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



Consider a single neuron.

At test time we have: $E[a] = w_1 x + w_2 y$

During training we have:

$$\begin{aligned} E[a] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

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

Dropout

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time

Dropout

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

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

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Dropout Summary

drop in forward pass

scale at test time

Dropout

More common: “Inverted dropout”

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

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

Normalization

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

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

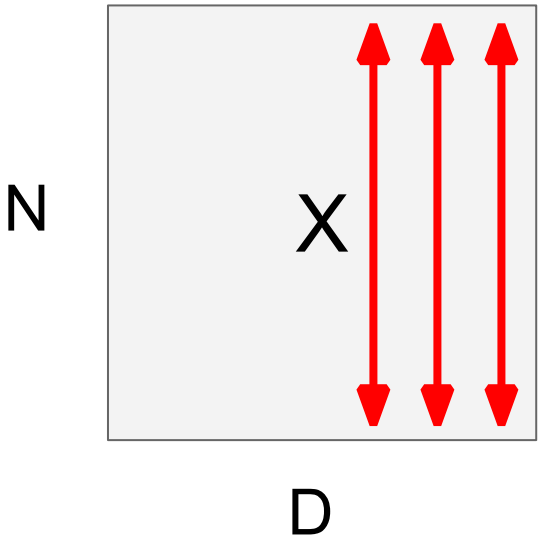
this is a vanilla differentiable function...

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.

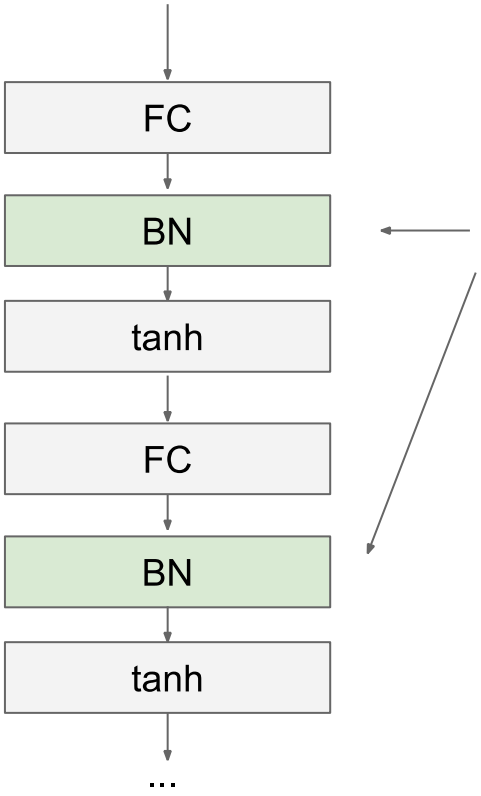
2. Normalize

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

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]



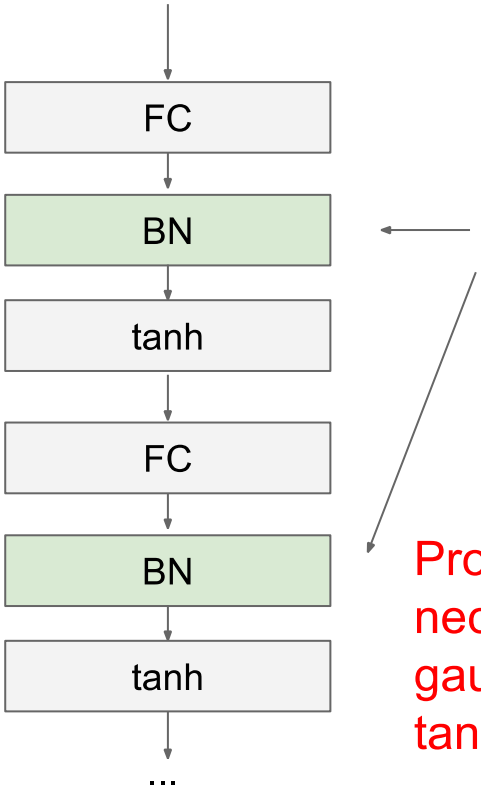
Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

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

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

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

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

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

Note, the network can learn:

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

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

to recover the identity mapping.

Maybe some degree of saturation is good. Let's give the learning algorithm the flexibility to control it, or even recover the identity function.

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

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

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

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)

- The forward and backward computation wrt an example, will be affected by other examples in the same mini-batch.
- The running mean and std are not parameters to be updated by gradients, but they need to be saved along with your model parameters.

Dropout vs. Batch Normalization

Ian Goodfellow, AI research scientist
Answered Aug 12, 2016 · Upvoted by Zeeshan Zia, PhD in Computer Vision and Machine Learning and Naran Bayanbat, MSCS with focus in machine learning

Dropout is mostly a technique for regularization. It introduces noise into a neural network to force the neural network to learn to generalize well enough to deal with noise. (This is a big oversimplification, and dropout is really about a lot more than just robustness to noise)

Batch normalization is mostly a technique for improving optimization.

As a side effect, batch normalization happens to introduce some noise into the network, so it can regularize the model a little bit.

When you have a large dataset, it's important to optimize well, and not as important to regularize well, so batch normalization is more important for large datasets. You can of course use both batch normalization and dropout at the same time—-I do this for some of my GANs in this paper: [1606.03498] Improved Techniques for Training GANs

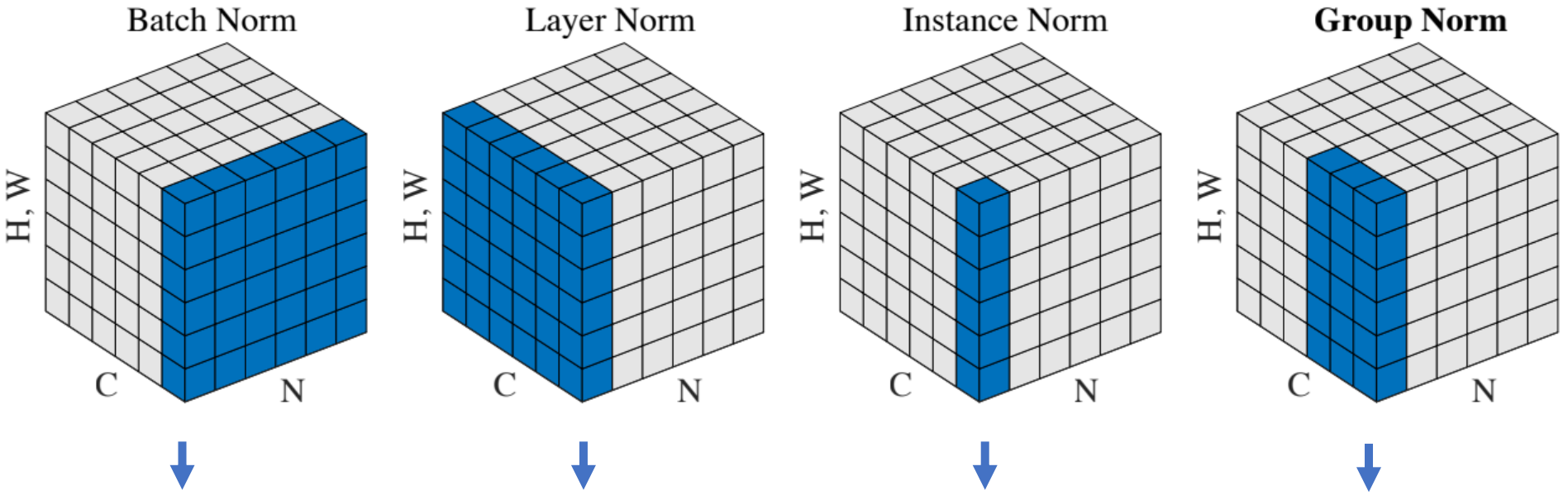
Dropout vs. Batch Normalization

- Dropout and BN can be used together, typically with the order as follows:

-> CONV/FC -> BatchNorm -> ReLu(or other activation) -> Dropout -> CONV/FC ->

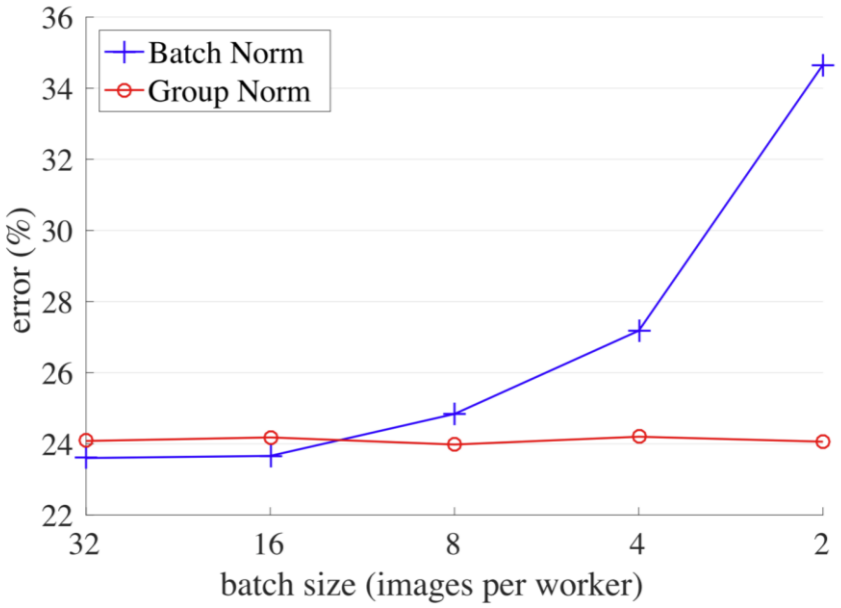
- See the following paper for more detailed analysis: “Identity Mappings in Deep Residual Networks”, https://arxiv.org/abs/1603.05027

Beyond Batch Normalization



Batch Norm	Layer Norm	Instance Norm	Group Norm
沿C方向对N,H,W维度进行归一化	沿N方向对C,H,W维度进行归一化	将归一化作用域某个实例，对H,W维度归一化	将C个通道分成G组，对每组通道进行归一化
Dependent of batch	Independent of batch, LN and IN are special cases of GN		
适用于classification等任务，但由于object detection、segmentation等任务的batch size较小，并不适合（syncbn is hence proposed to perform BN across GPUs）	Effective for training sequential models, such as RNN/LSTM/Transformer (一个batch里的序列长度不一，在后段时刻，有效batch size往往很小，因此要避免对batch维度进行归一化)		Stronger representation power than LN/IN (an extra parameter G). A good alternative of BN/LN/IN for a variety of tasks (but be careful with hyper-parameter tuning, as they may be already tuned to fit BN).

Beyond Batch Normalization



GN是针对BN在batch size较小时错误率较高而提出的改进算法，因为BN层的计算结果依赖当前batch的数据，当batch size较小时（比如2、4这样）,该batch数据的均值和方差的代表性较差，因此对最后的结果影响也较大。如图Figure1所示，随着batch size越来越小，BN层所计算的统计信息的可靠性越来越差，这样就容易导致最后错误率的上升；而在batch size较大时则没有明显的差别。虽然在分类算法中一般的GPU显存都能cover住较大的batch设置，但是在目标检测、分割以及视频相关的算法中，由于输入图像较大、维度多样以及算法本身原因等，batch size一般都设置比较小，所以GN对于这种类型算法的改进应该比较明显。
原文链接：
https://blog.csdn.net/u014380165/article/details/79810040

An intuitive motivation of why group channels:

The channels of visual representations are not entirely independent. Classical features of SIFT, HOG, and GIST are group-wise representations by design, where each group of channels is constructed by some kind of histogram. These features are often processed by groupwise normalization over each histogram or each orientation. Higher-level features such as VLAD and Fisher Vectors (FV) are also group-wise features where a group can be thought of as the sub-vector computed with respect to a cluster.

Optimization

基本框架

定义当前时刻待优化参数为$\theta_t \in \mathbb{R}^d$，损失函数为$J(\theta)$，学习率为$\eta$，参数更新框架为：

1. 计算损失函数关于当前参数的梯度：$g_t = \nabla J(\theta_t)$
2. 根据历史梯度计算一阶动量和二阶动量：

$$m_t = \phi(g_1, g_2, ..., g_t), V_t = \psi(g_1, g_2, ..., g_t)$$

3. 计算当前时刻的下降梯度：

$$\Delta\theta_t = -\eta \cdot \frac{m_t}{\sqrt{V_t}}$$

4. 根据下降梯度更新参数：$\theta_{t+1} = \theta_t + \Delta\theta_t$

其中，一阶动量和二阶动量分别是历史梯度的一阶函数和二阶函数。

SGD（Stochastic Gradient Descent）：由于SGD没有动量的概念，也即没有考虑历史梯度，所以当前时刻的一阶动量即为当前时刻的梯度$m_t = g_t$，且二阶动量$V_t = E$，所以SGD的参数更新公式为

$$\Delta\theta_t = -\eta \cdot \frac{g_t}{\sqrt{E}} = -\eta \cdot g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t = \theta_t - \eta \cdot g_t$$

缺点：容易陷入局部最优。由于SGD只考虑当前时刻的梯度，在局部最优点的当前梯度为0。由计算公式可知，此时参数不再进行更新，故陷入局部最优的状态。

改进策略及算法

- 引入历史梯度的一阶动量，代表算法有：Momentum、NAG
- 引入历史梯度的二阶动量，代表算法有：AdaGrad、RMSProp、AdaDelta
- 同时引入历史梯度的一阶动量及二阶动量，代表算法有：Adam、Nadam

Momentum

Momentum（SGD with Momentum）：为了抑制SGD的震荡，Momentum认为梯度下降过程可以加入**惯性**，也就是在SGD基础上引入了一阶动量。而所谓的一阶动量就是该时刻梯度的指数加权移动平均值：$\eta \cdot m_t := \beta \cdot m_{t-1} + \eta \cdot g_t$（其中$g_t$并不严格按照指数加权移动平均值的定义采用权重$1-\beta$，而是使用我们自定义的学习率$\eta$）。由于此时仍然没有用到二阶动量，所以$V_t = E$，那么Momentum的参数更新公式为

$$\Delta\theta_t = -\eta \cdot \frac{m_t}{\sqrt{E}} = -\eta \cdot m_t = -\left(\beta m_{t-1} + \eta g_t\right)$$

$$\theta_{t+1} = \theta_t - \left(\beta m_{t-1} + \eta g_t\right)$$

NAG

NAG（Nesterov Accelerated Gradient）：除了利用惯性跳出局部沟壑以外，我们还可以尝试往前看一步。想象一下你走到一个盆地，四周都是略高的小山，你觉得没有下坡的方向，那就只能待在这里了。可是如果你爬上高地，就会发现外面的世界还很广阔。因此，我们不能停留在当前位置去观察未来的方向，而要向前多看一步。我们知道Momentum在时刻t的主要下降方向是由历史梯度（惯性）决定的，当前时刻的梯度权重较小，那不如先看看如果跟着惯性走了一步，那个时候外面的世界是怎样的。也即在Momentum的基础上将当前时刻的梯度g_t换成下一时刻的梯度$\nabla J(\theta_t - \beta m_{t-1})$，由于此时仍然没有用到二阶动量，所以$V_t = E$，NAG的参数更新公式为

$$\Delta\theta_t = -\eta \cdot \frac{m_t}{\sqrt{E}} = -\eta \cdot m_t = -\left(\beta m_{t-1} + \eta \nabla J(\theta_t - \beta m_{t-1})\right)$$

$$\theta_{t+1} = \theta_t - \left(\beta m_{t-1} + \eta \nabla J(\theta_t - \beta m_{t-1})\right)$$

上式仅仅是第t时刻第i维度参数的更新公式，对于第t时刻的所有维度参数的整体更新公式为

$$V_t = \mathrm{diag}\left(v_{t,1}, v_{t,2}, ..., v_{t,d}\right) \in \mathbb{R}^{d \times d}$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{V_t + \epsilon}} \cdot g_t$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{V_t + \epsilon}} \cdot g_t$$

注意，由于V_t是对角矩阵，所以上式中的ϵ只用来平滑V_t对角线上的元素。

缺点：随着时间步的拉长，历史累计梯度平方和$v_{t,i}$会越来越大，这样会使得所有维度参数的学习率都不断减小（单调递减），无论更新幅度如何。

改进思想：从数学的角度来看，更新幅度很大的参数，通常历史累计梯度的平方和会很大；相反的，更新幅度很小的参数，通常其累计历史梯度的平方和会很小。

缺点：随着时间步的拉长，历史累计梯度平方和会越来越大，这样会使得所有维度参数的学习率都不断减小（单调递减），无论更新幅度如何。

RMSProp/AdaDelta

Datawhale

由于AdaGrad单调递减的学习率变化过于激进，我们考虑一个改变二阶动量计算方法的策略：不累积全部历史梯度，而只关注过去一段时间窗口的下降梯度，采用Momentum中的指数加权移动平均值的思路。这也就是AdaDelta名称中Delta的来历。首先看最简单直接版的RMSProp，RMSProp就是在AdaGrad的基础上将普通的历史累计梯度平方和换成历史累计梯度平方和的指数加权移动平均值，所以只需将AdaGrad中的$v_{t,i}$的公式改成指数加权移动平均值的形式即可，也即

$$v_{t,i} = \beta v_{t-1,i} + (1-\beta) g_{t,i}^2$$

$$V_t = \mathrm{diag}\left(v_{t,1}, v_{t,2}, ..., v_{t,d}\right) \in R^{d \times d}$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{V_t + \epsilon}} \cdot g_t$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{V_t + \epsilon}} \cdot g_t$$

改进思想：不累积全部历史梯度，而只关注过去一段时间窗口的下降梯度，采用Momentum中的指数加权移动平均值的思路。

Adam

Datawhale

具体地，首先计算一阶动量：$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$

然后类似RMSProp/AdaDelta计算二阶动量

$$v_{t,i} = \beta_2 v_{t-1,i} + (1-\beta_2) g_{t,i}^2$$

$$V_t = \mathrm{diag}\left(v_{t,1}, v_{t,2}, ..., v_{t,d}\right) \in R^{d \times d}$$

然后分别加上指数加权移动平均值的修正因子

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_{t,i} = \frac{v_{t,i}}{1-\beta_2^t}$$

$$\hat{V}_t = \mathrm{diag}\left(\hat{v}_{t,1}, \hat{v}_{t,2}, ..., \hat{v}_{t,d}\right) \in R^{d \times d}$$

改进思想：加入Momentum的一阶动量计算方法及AdaGrad的二阶动量计算方法。

Nadam

Datawhale

具体地，首先NAdam在Adam的基础上将$\hat{m}_t$展开

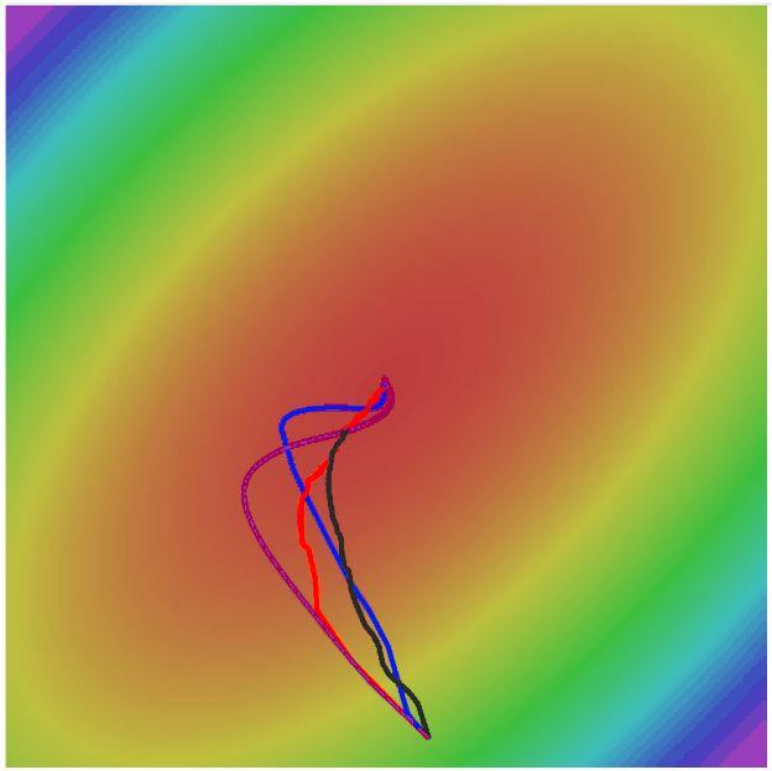
$$\begin{aligned}\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{V}_t + \epsilon}} \cdot \hat{m}_t \\ &= \theta_t - \frac{\eta}{\sqrt{\hat{V}_t + \epsilon}} \cdot \left(\frac{\beta_1 m_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1) g_t}{1-\beta_1^t}\right)\end{aligned}$$

此时，如果我们将第$t-1$时刻的动量m_{t-1}用第t时刻的动量m_t近似代替的话，那么我们就引入了「未来因素」，所以将m_{t-1}替换成m_t即可得到Nadam的表达式

$$\begin{aligned}\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{V}_t + \epsilon}} \cdot \left(\frac{\beta_1 m_t}{1-\beta_1^t} + \frac{(1-\beta_1) g_t}{1-\beta_1^t}\right) \\ &= \theta_t - \frac{\eta}{\sqrt{\hat{V}_t + \epsilon}} \cdot \left(\beta_1 \hat{m}_t + \frac{(1-\beta_1) g_t}{1-\beta_1^t}\right)\end{aligned}$$

改进思想：Nadam = Nesterov + Adam。核心在于计算当前时刻的梯度时使用了未来梯度。

Comparison



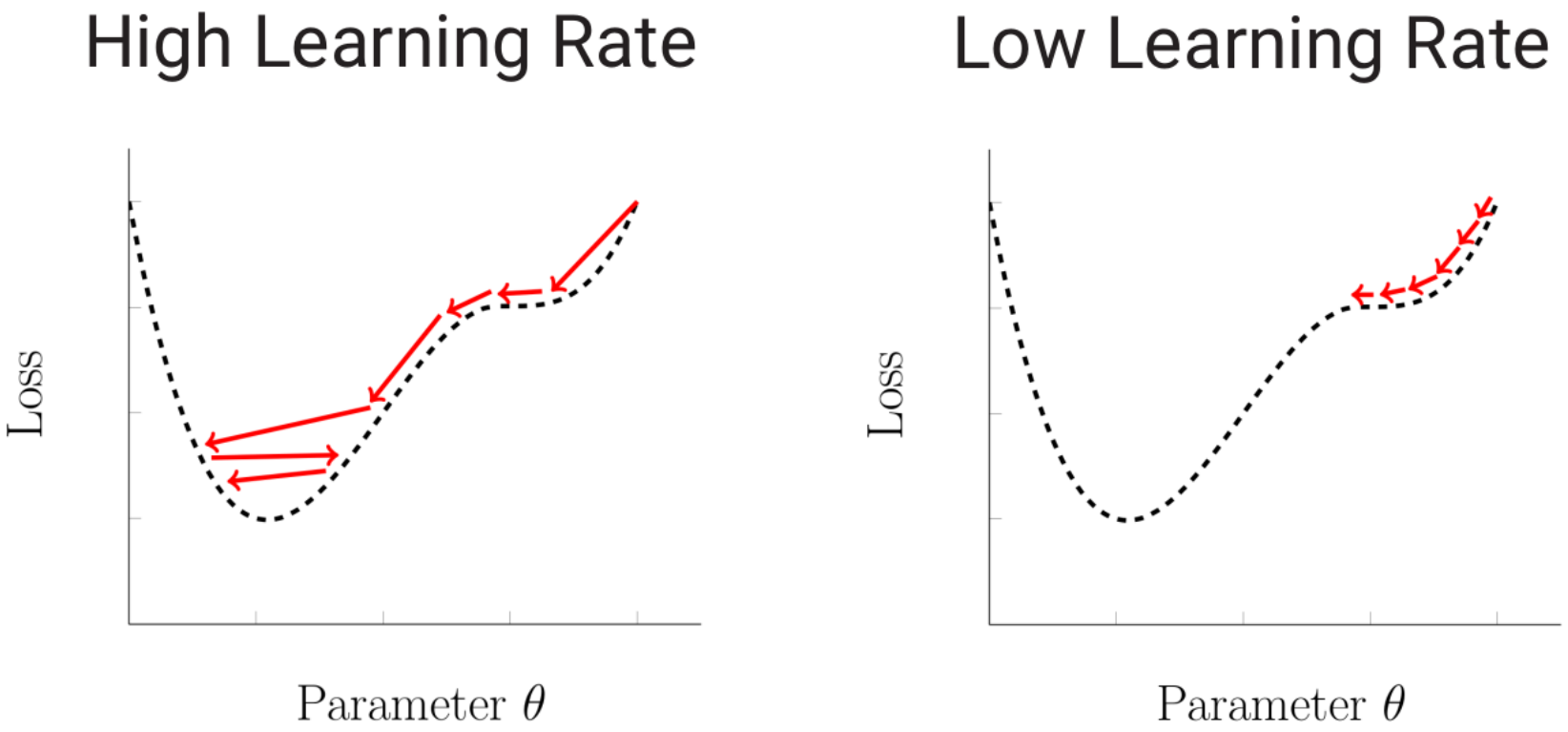
SGD

SGD+Momentum

RMSProp

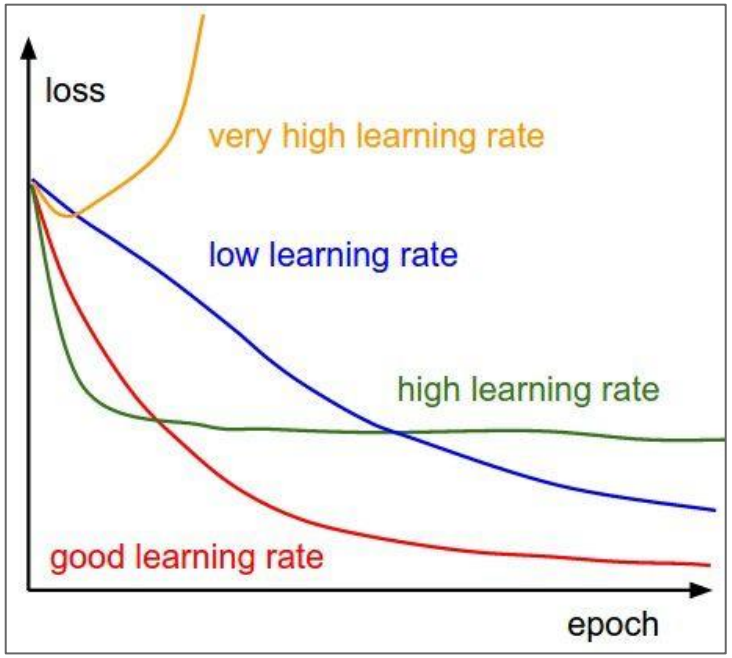
Adam

Learning rate



Learning rate

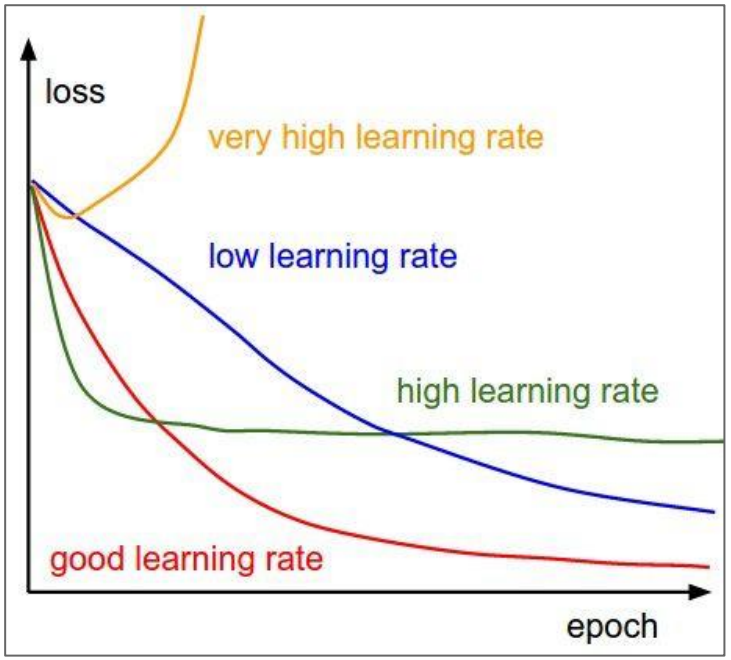
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



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

Learning rate

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



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

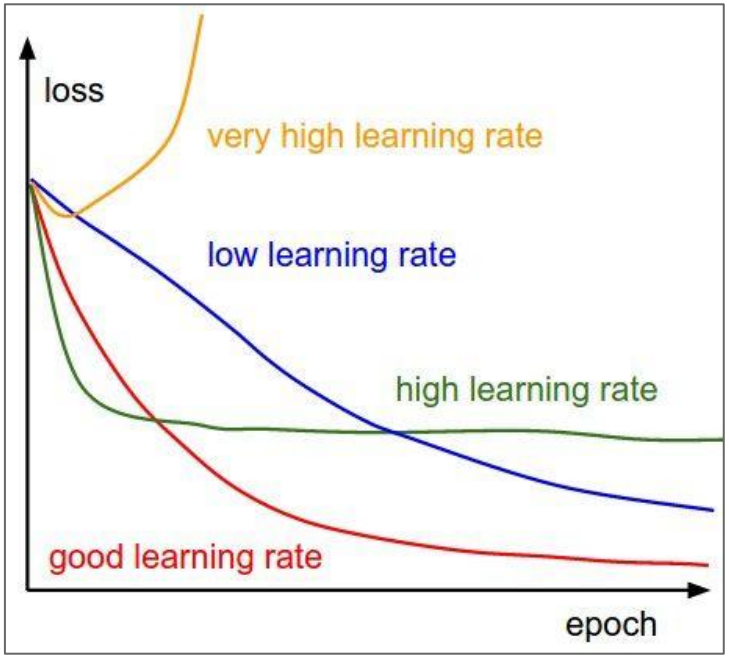
$$\alpha = \alpha_0 e^{-kt}$$

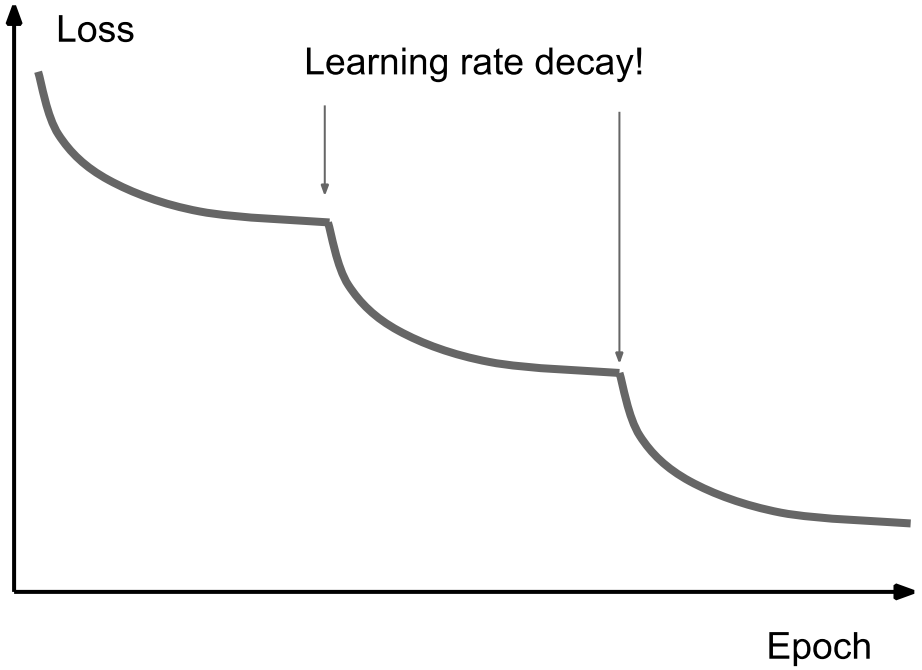
1/t decay:

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

Learning rate

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





Reparameterization
(重参数化)

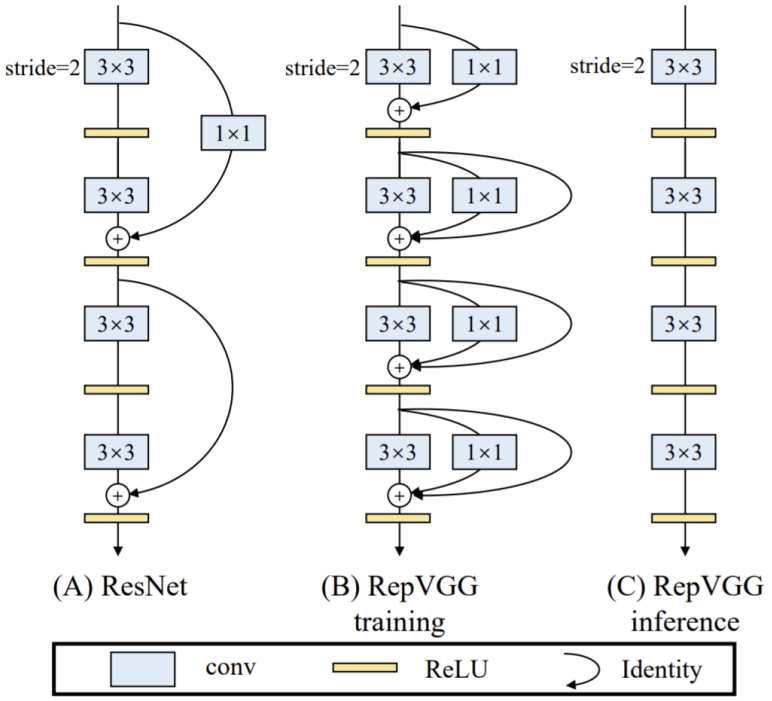
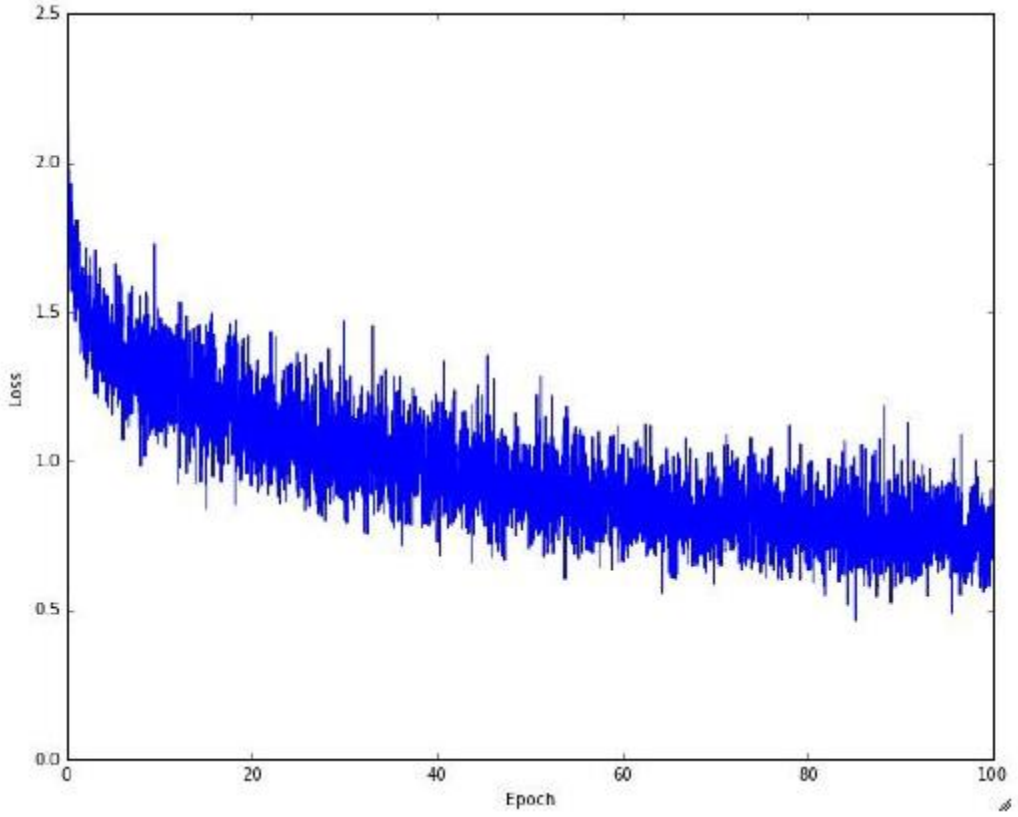


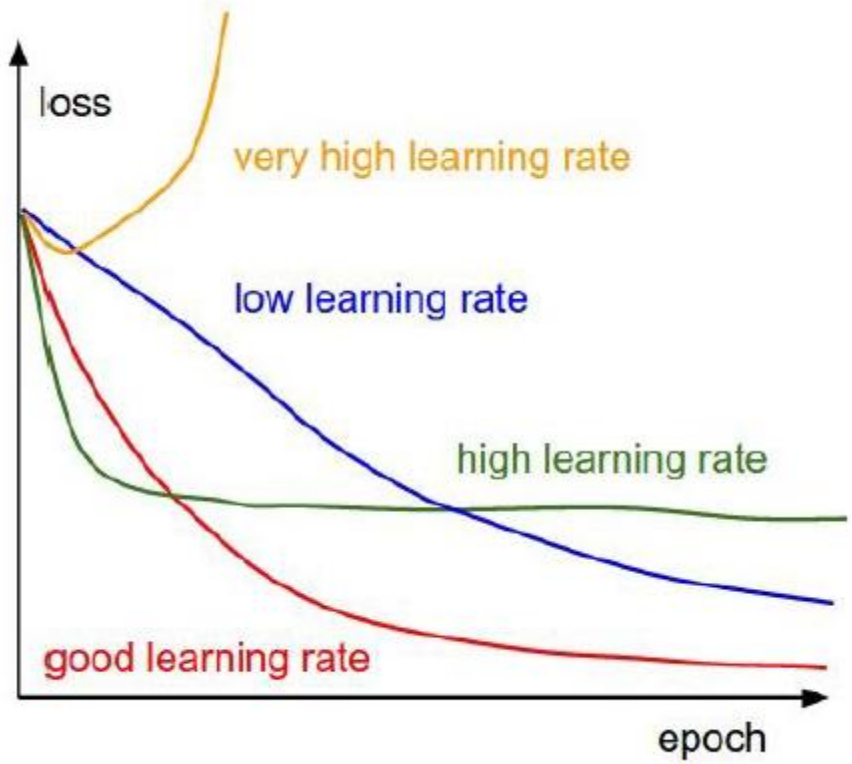
Figure 2: Sketch of RepVGG architecture. RepVGG has 5 stages and conducts down-sampling via stride-2 convolution at the beginning of a stage. Here we only show the first 4 layers of a specific stage. As inspired by ResNet [10], we also use identity and 1×1 branches, but only for training.

We present a simple but powerful architecture of convolutional neural network, which has a VGG-like inference-time body composed of nothing but a stack of 3×3 convolution and ReLU, while the training-time model has a multi-branch topology. Such decoupling of the training-time and inference-time architecture is realized by a structural re-parameterization technique so that the model is named RepVGG. On ImageNet, RepVGG reaches over 80% top-1 accuracy, which is the first time for a plain model, to the best of our knowledge. On NVIDIA 1080Ti GPU, RepVGG models run 83% faster than ResNet-50 or 101% faster than ResNet-101 with higher accuracy and show favorable accuracy-speed trade-off compared to the state-of-the-art models like EfficientNet and RegNet. The code and trained models are available at https://github.com/megvii-model/RepVGG.

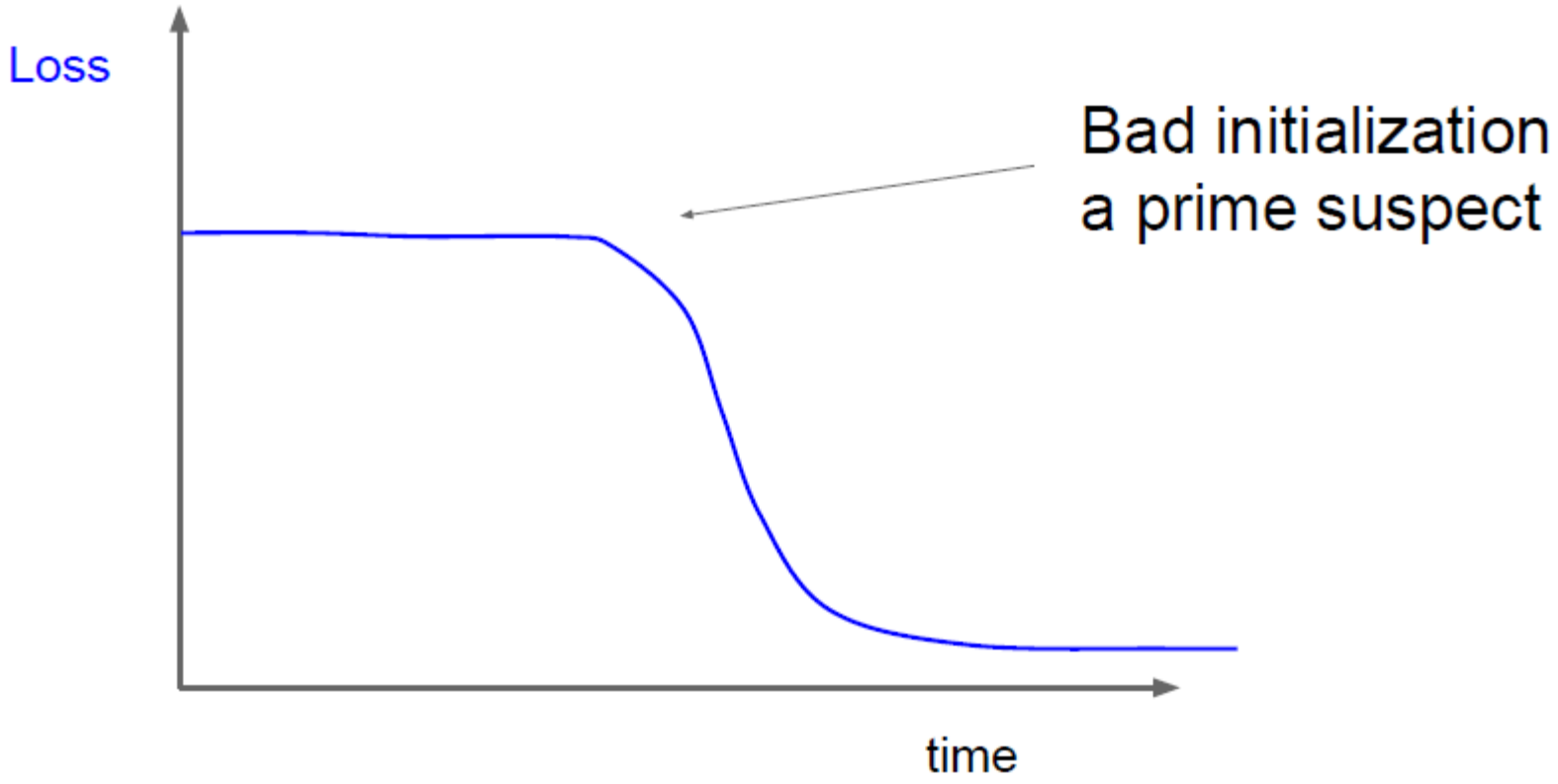
Babysitting the Learning Process

Monitor and visualize the Loss curve

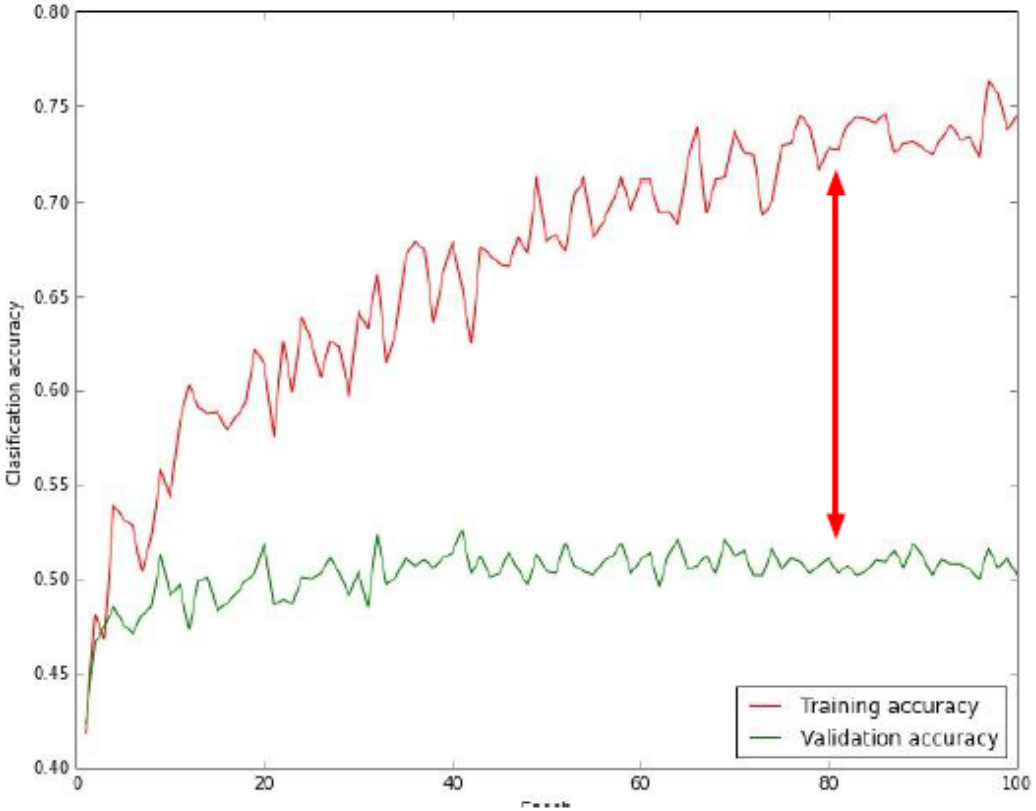




Monitor and visualize the Loss curve



Monitor and visualize the Accuracy curve



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Track the ratio of weight updates / weight magnitudes

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

want this to be somewhere around 0.001 or so

Hyper-parameter Optimization

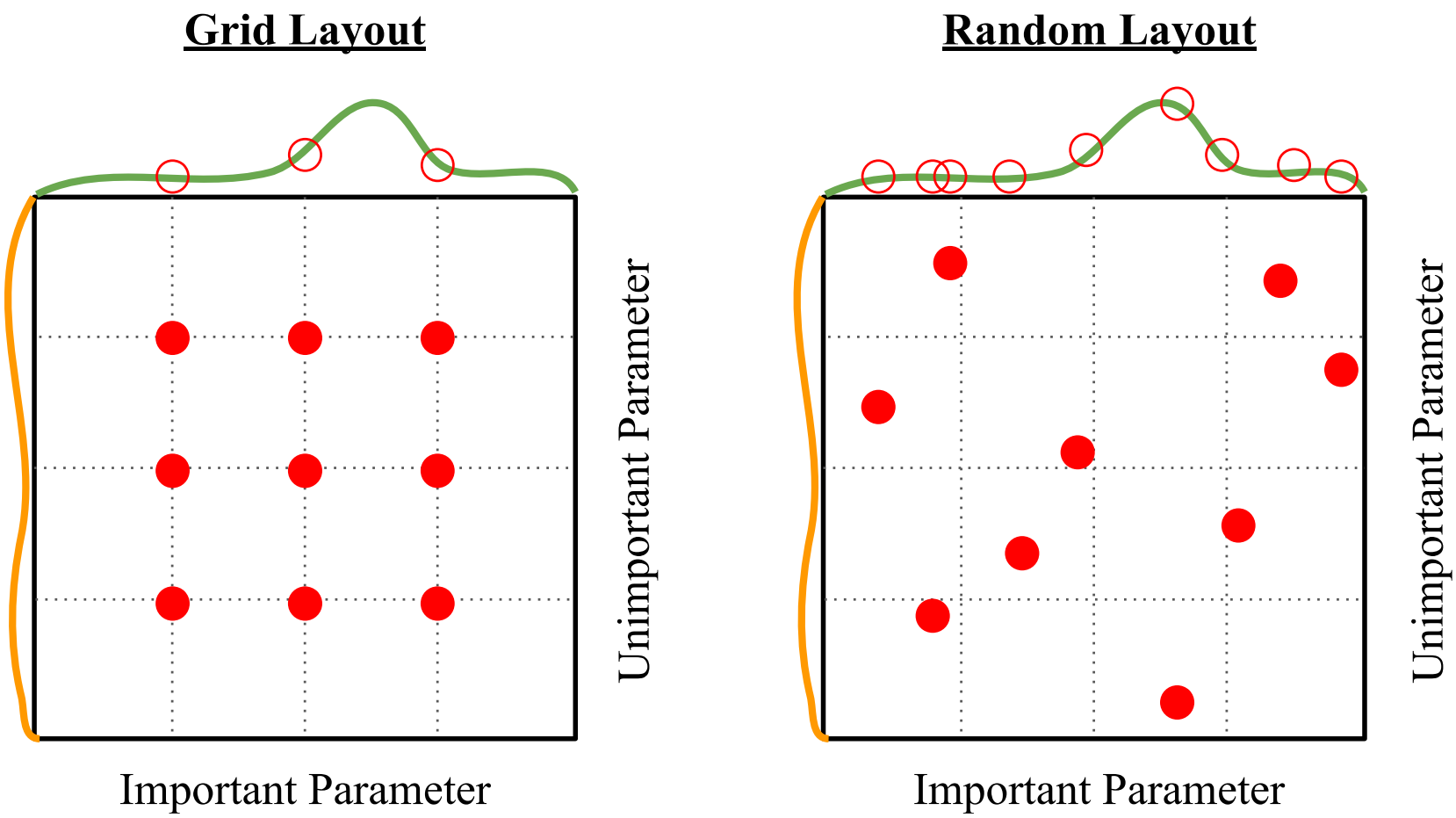
Coarse-to-fine search

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

It is best to search in Log space

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

Grid search vs. Random search



Model Ensembles

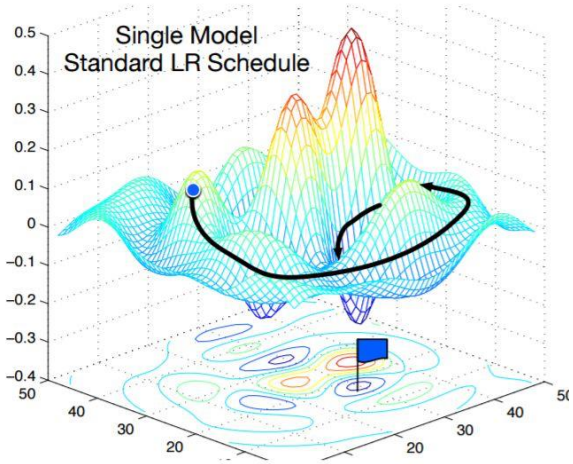
Model Ensembles

1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance

Model Ensembles

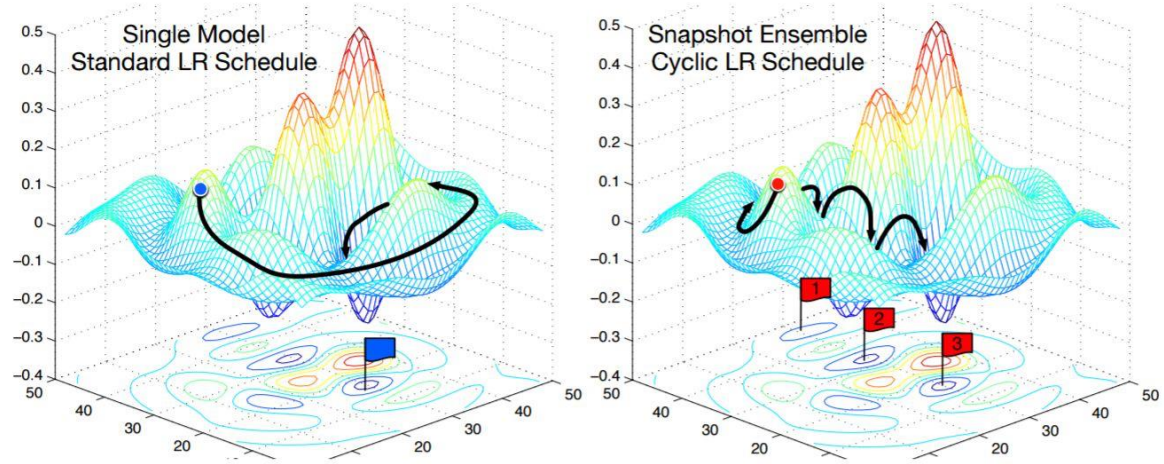
Instead of training independent models, use multiple snapshots of a single model during training!

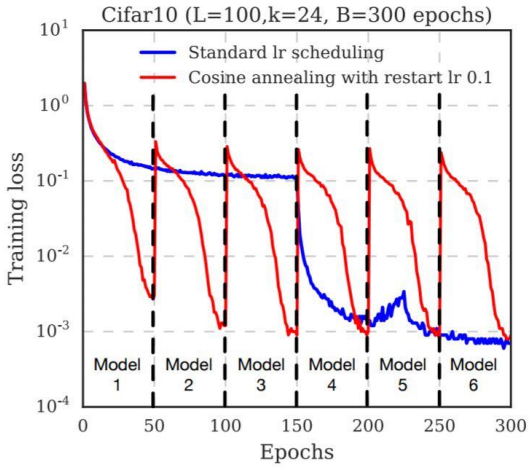


Loshchilov and Hutter, “SGDR: Stochastic gradient descent with restarts”, arXiv 2016
Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017

Model Ensembles

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





Cyclic learning rate schedules can make this work even better!

Loshchilov and Hutter, “SGDR: Stochastic gradient descent with restarts”, arXiv 2016
Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Summary

Summary

Regularization (to minimize generalization gap)	**Optimization (to minimize training error)**
• Data augmentation • Weight decay • Dropout • Batch Normalization (a little) • Model Ensembles	• Data preprocessing (zero mean, …) • Weight initialization • Batch Normalization (mainly) • Optimization (momentum, RMSProp, Adam, …)