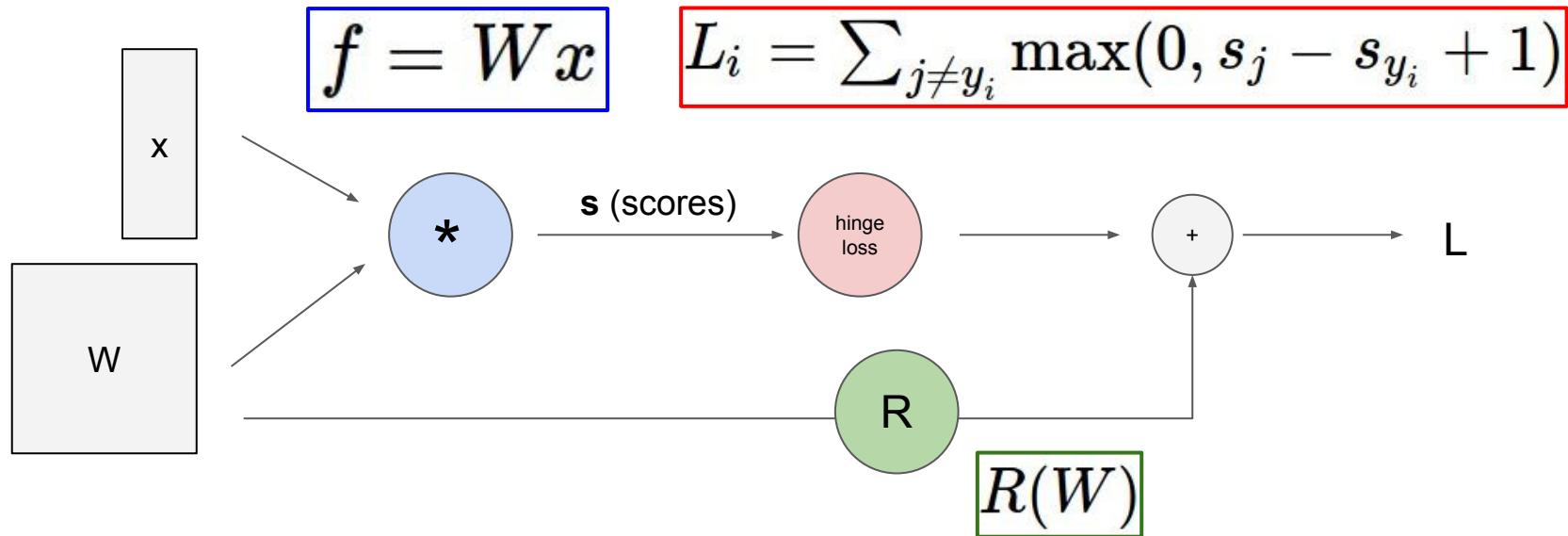


# Lecture 7: Training Neural Networks, Part I

Where we are now...

# Computational graphs



Where we are now...

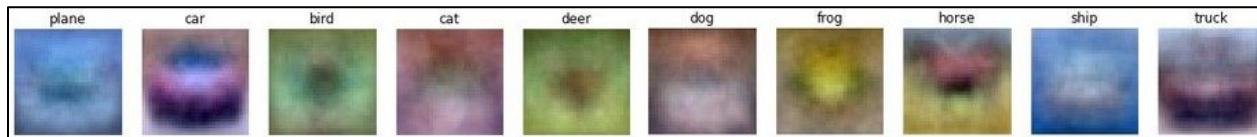
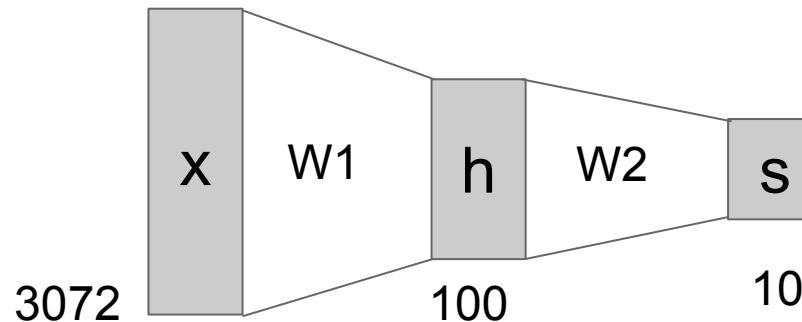
# Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$



Where we are now...

# Convolutional Neural Networks

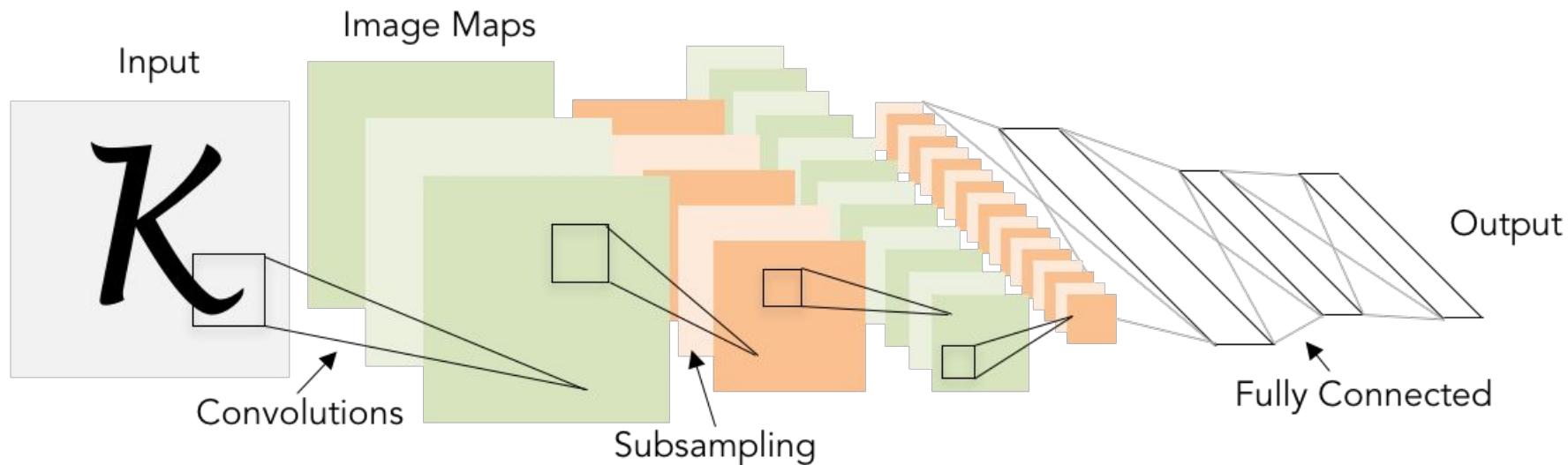
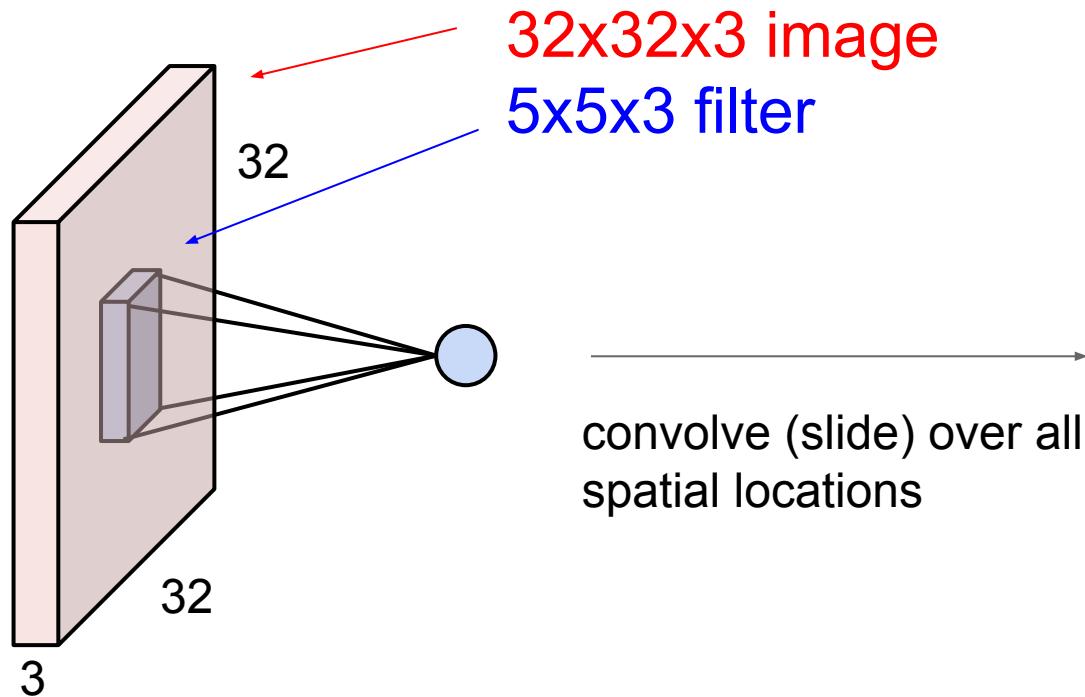


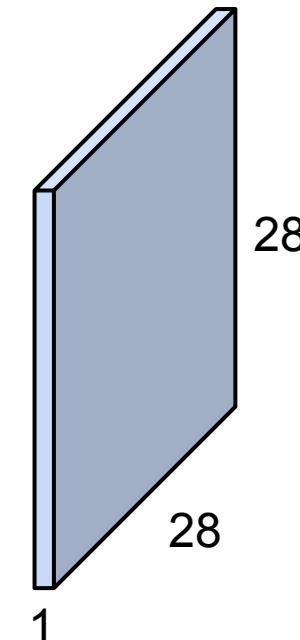
Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Where we are now...

## Convolutional Layer

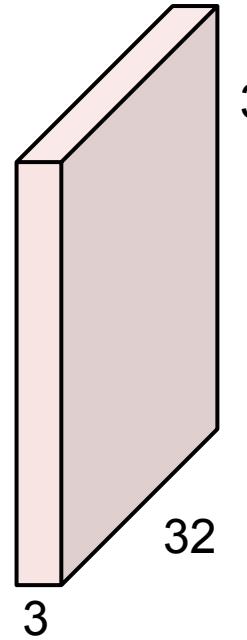


activation map



Where we are now...

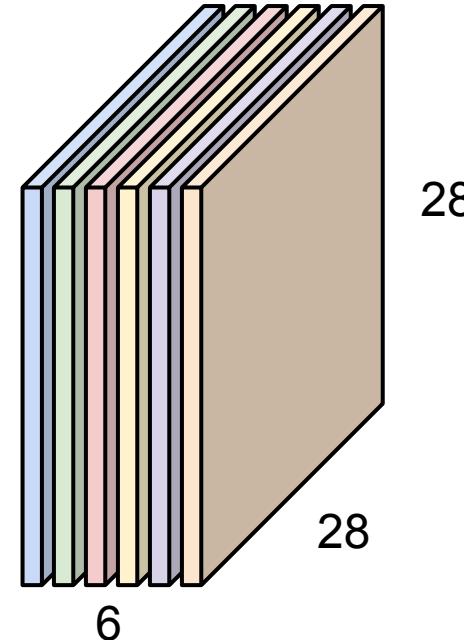
## Convolutional Layer



Convolution Layer

For example, if we had 6  $5 \times 5$  filters, we'll get 6 separate activation maps:

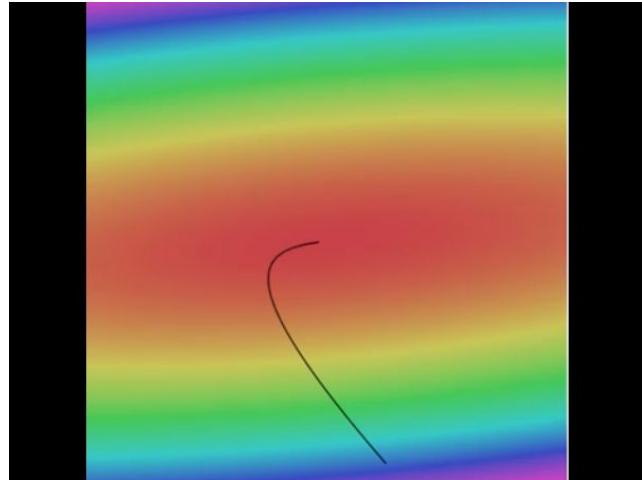
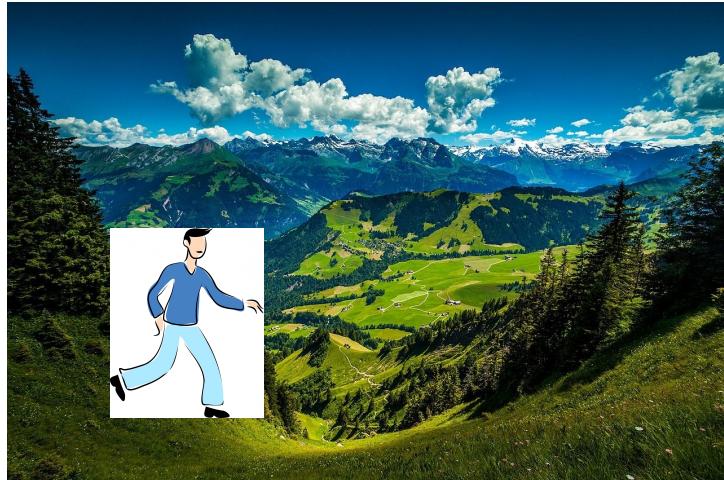
activation maps



We stack these up to get a “new image” of size  $28 \times 28 \times 6$ !

Where we are now...

# Learning network parameters through optimization



```
# Vanilla Gradient Descent  
  
while True:  
    weights_grad = evaluate_gradient(loss_fun, data, weights)  
    weights += - step_size * weights_grad # perform parameter update
```

Landscape image is CC0 1.0 public domain  
Walking man image is CC0 1.0 public domain

Where we are now...

# Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph  
(network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

# CPU vs GPU

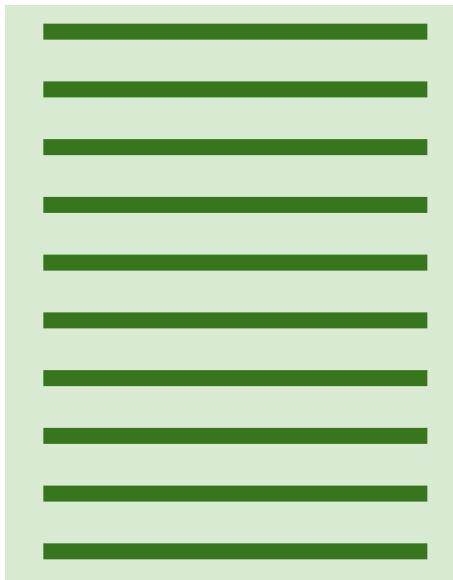
	<b>Cores</b>	<b>Clock Speed</b>	<b>Memory</b>	<b>Price</b>	<b>Speed</b>
<b>CPU</b> (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$385	~540 GFLOPs FP32
<b>GPU</b> (NVIDIA RTX 2080 Ti)	3584	1.6 GHz	11 GB GDDR6	\$1199	~13.4 TFLOPs FP32

**CPU:** Fewer cores, but each core is much faster and much more capable; great at sequential tasks

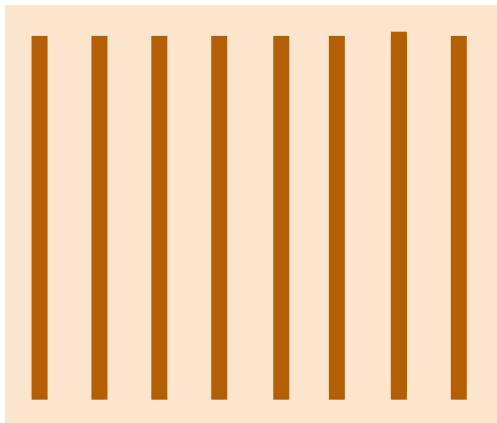
**GPU:** More cores, but each core is much slower and “dumber”; great for parallel tasks

# Example: Matrix Multiplication

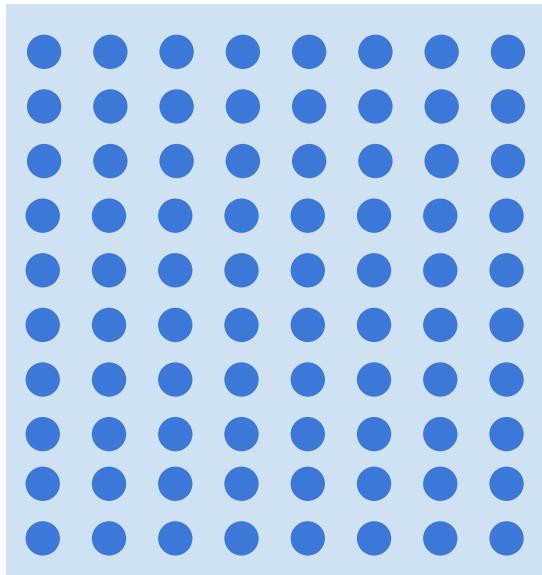
$A \times B$



$B \times C$

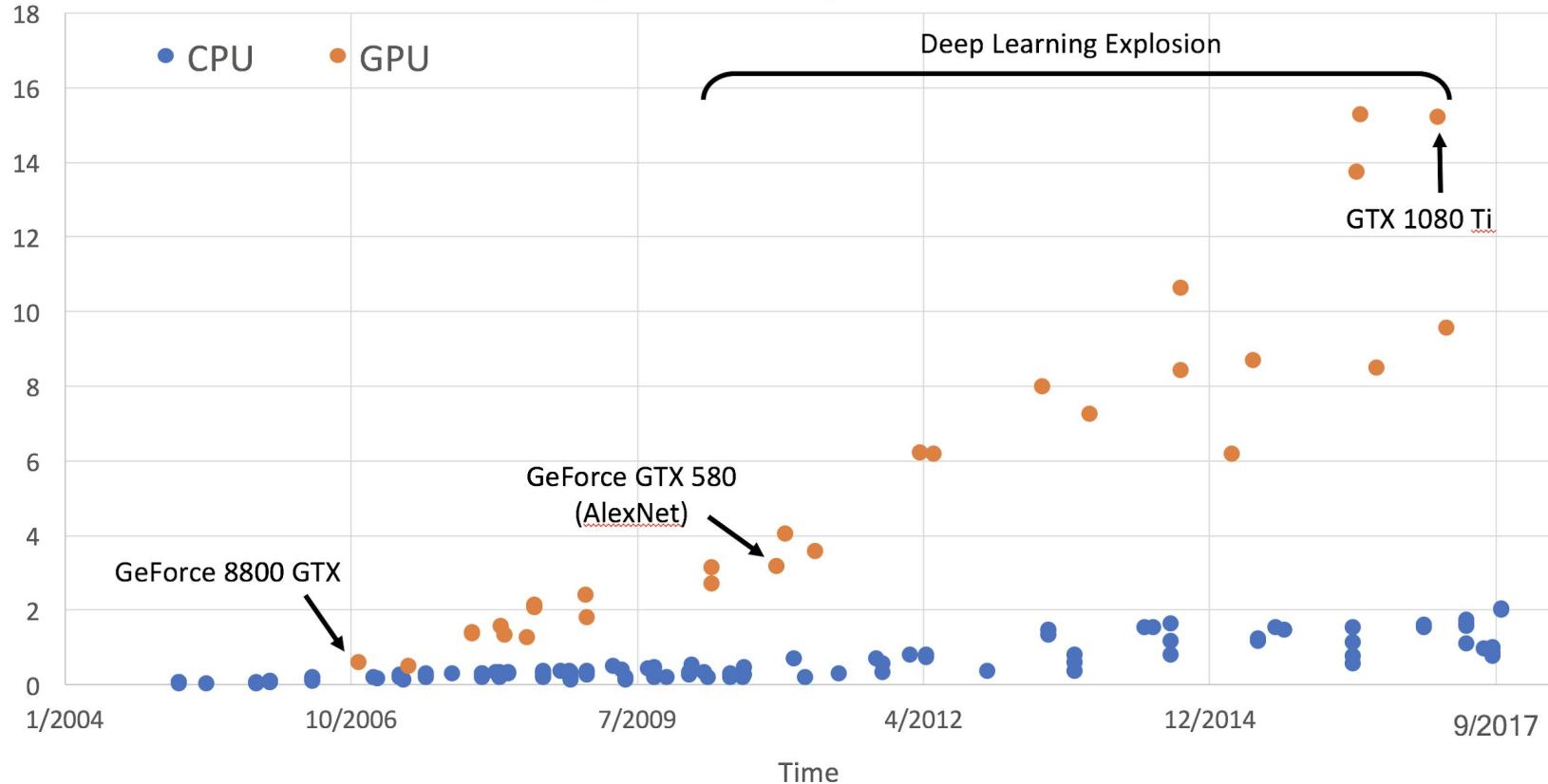


$A \times C$



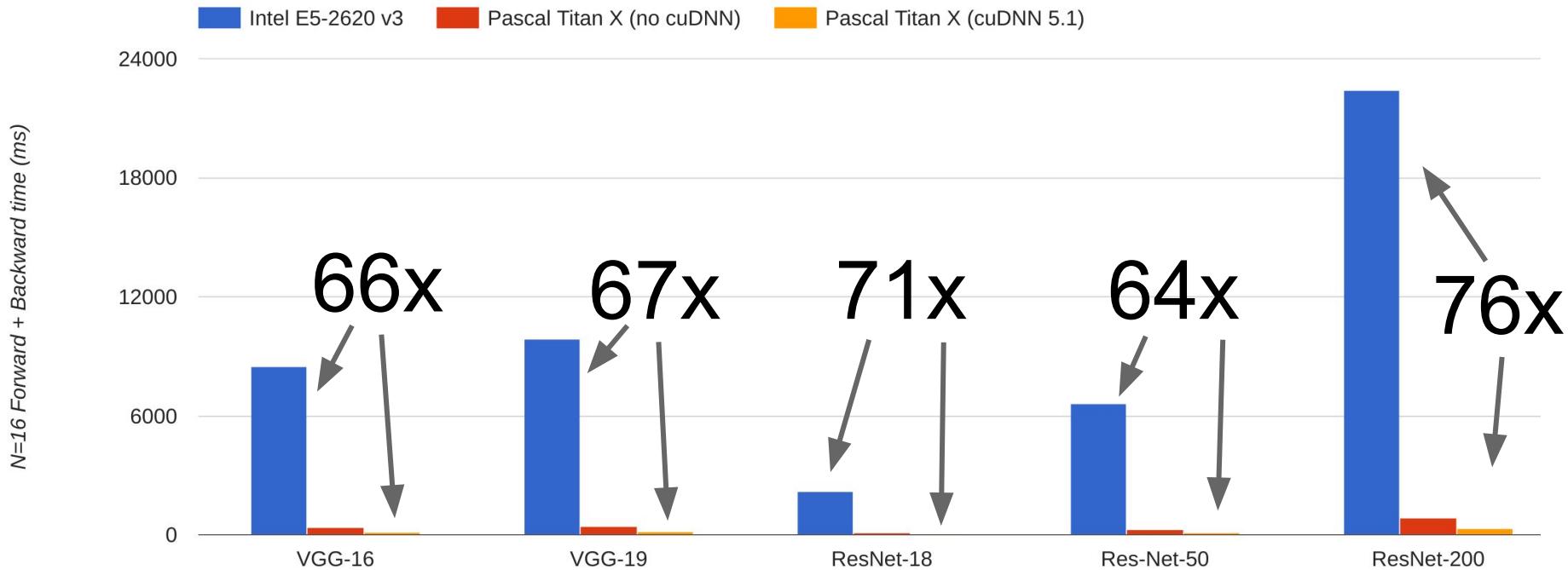
=

# GigaFLOPs per Dollar



# CPU vs GPU in practice

(CPU performance not well-optimized, a little unfair)



Data from <https://github.com/jcjohnson/cnn-benchmarks>

# Next: 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

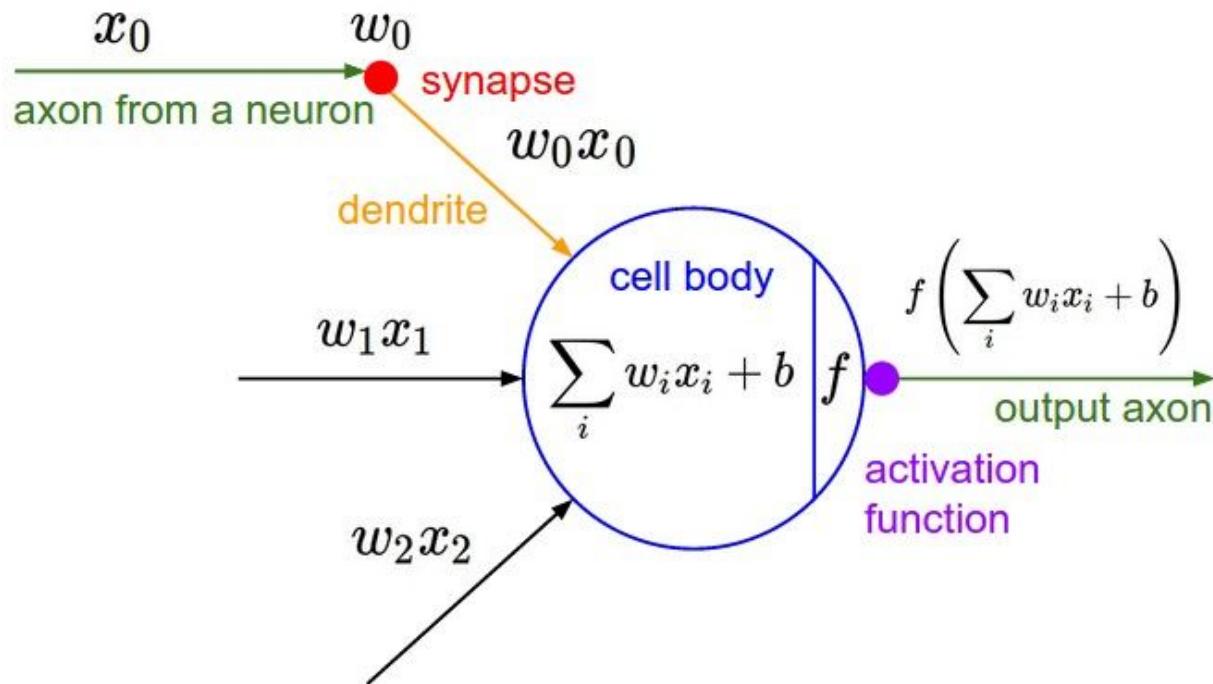
*model ensembles, test-time augmentation*

# Part 1

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization
- Babysitting the Learning Process
- Hyperparameter Optimization

# Activation Functions

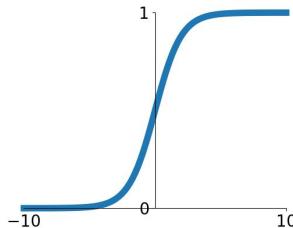
# Activation Functions



# Activation Functions

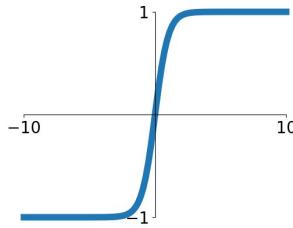
## Sigmoid

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



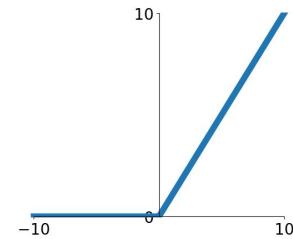
## tanh

$$\tanh(x)$$



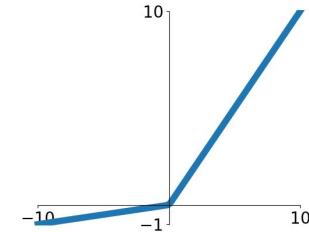
## ReLU

$$\max(0, x)$$



## Leaky ReLU

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

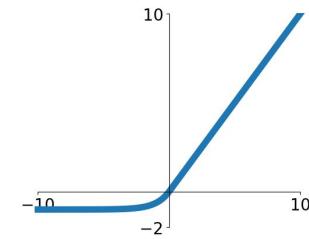


## Maxout

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

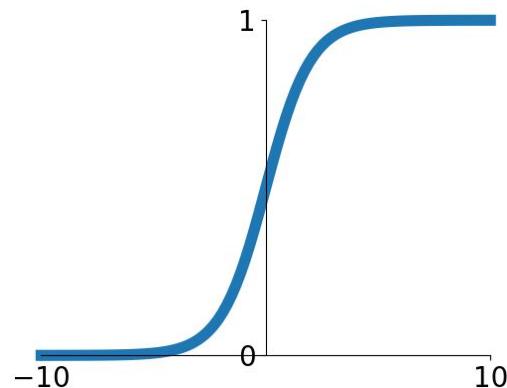
## ELU

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



# Activation Functions

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

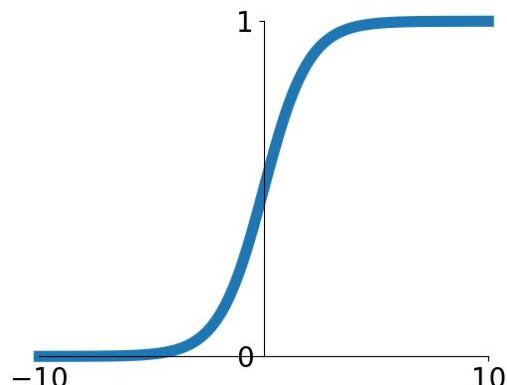


**Sigmoid**

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

# Activation Functions

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

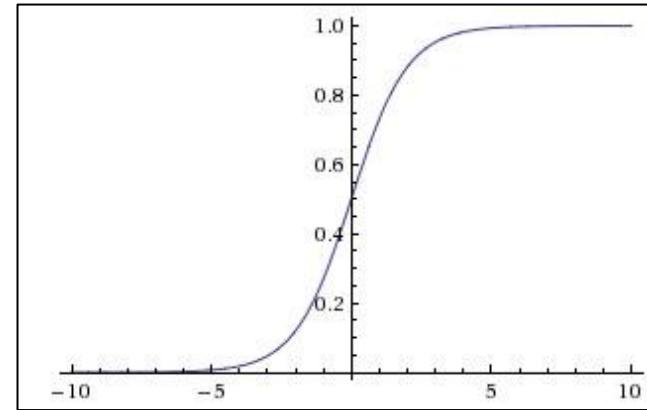
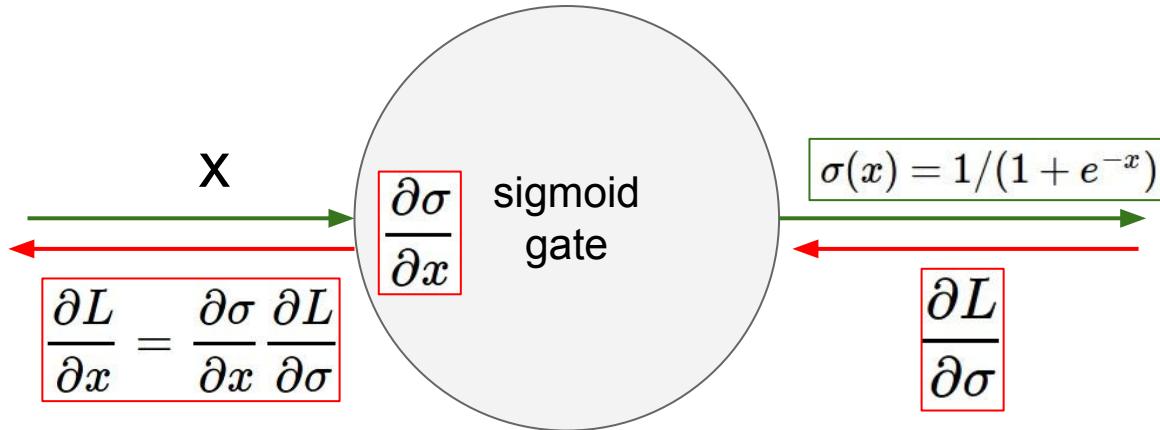


Sigmoid

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

3 problems:

1. Saturated neurons “kill” the gradients



What happens when  $x = -10$ ?

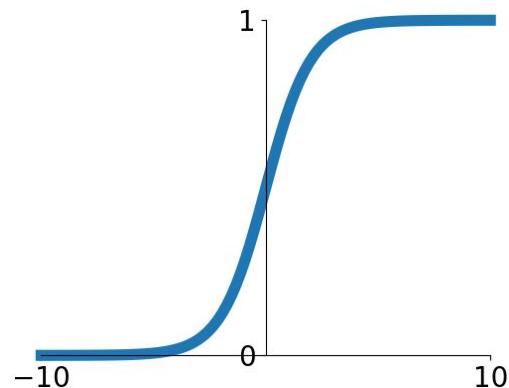
What happens when  $x = 0$ ?

What happens when  $x = 10$ ?

# Activation Functions

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

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



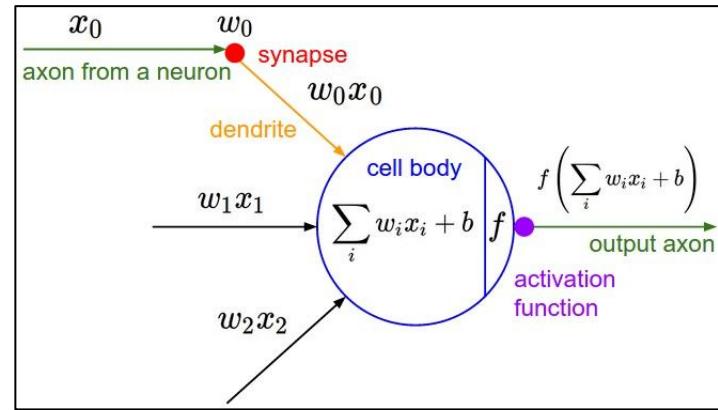
Sigmoid

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

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

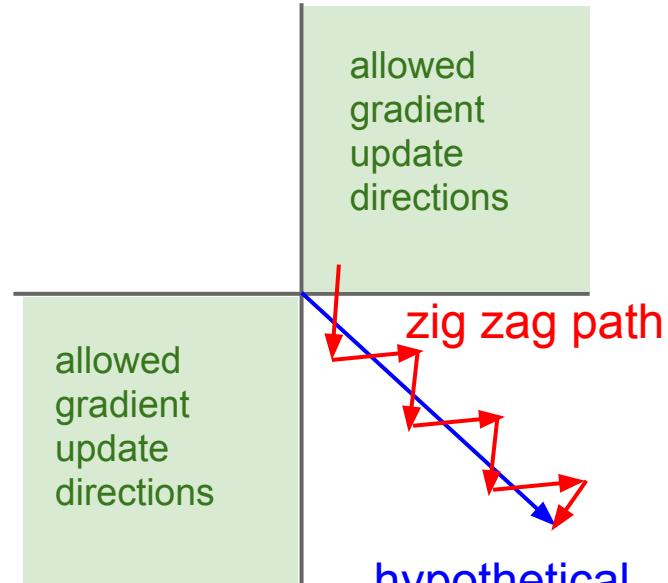
$$f \left( \sum_i w_i x_i + b \right)$$



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

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

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

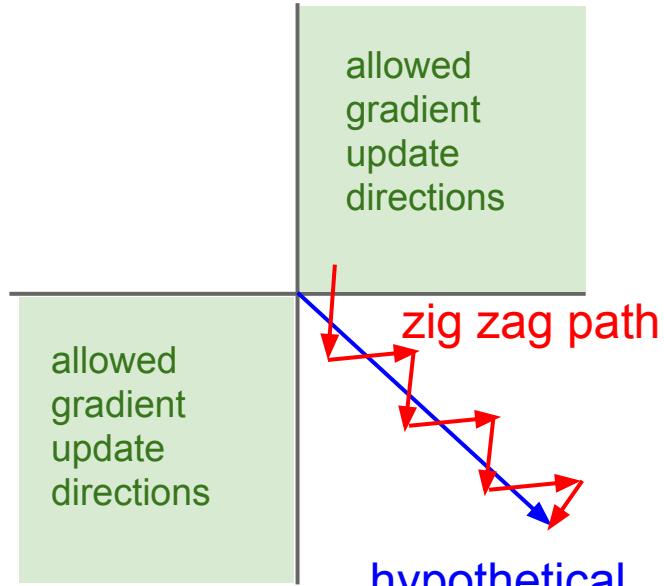


What can we say about the gradients on  $w$ ?

Always all positive or all negative :(

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  $w$ ?

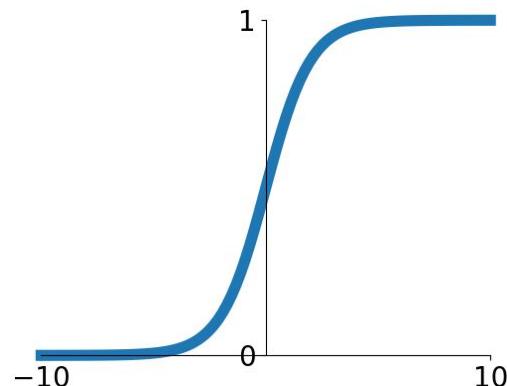
Always all positive or all negative :(

(For a single element! Minibatches help)

# Activation Functions

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

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

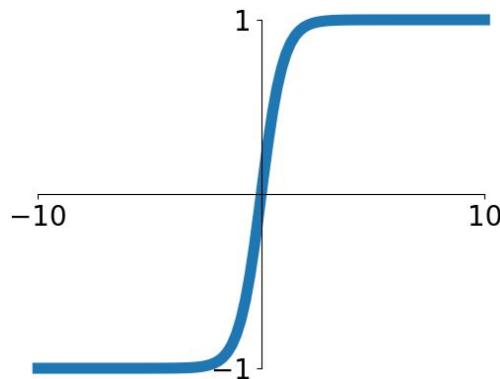


Sigmoid

3 problems:

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

# Activation Functions

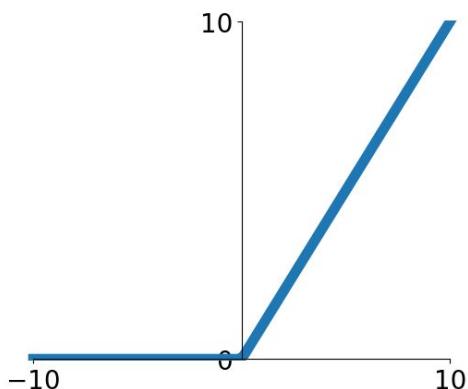


**tanh(x)**

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

[LeCun et al., 1991]

# Activation Functions

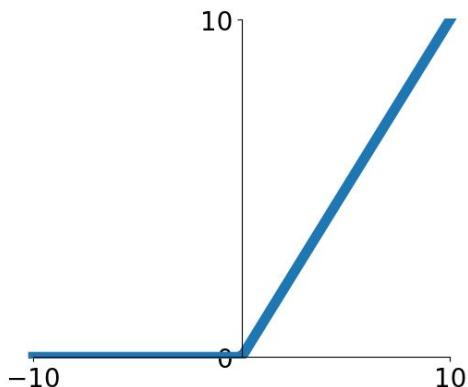


- Computes  $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

**ReLU**  
(Rectified Linear Unit)

[Krizhevsky et al., 2012]

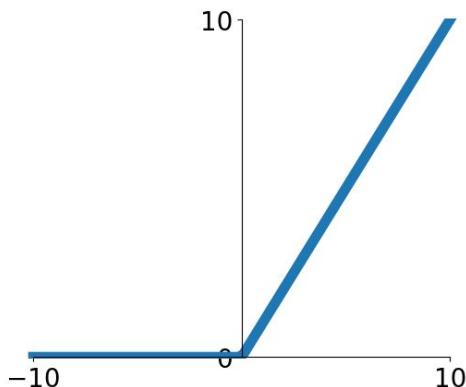
# Activation Functions



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

**ReLU**  
(Rectified Linear Unit)

# Activation Functions

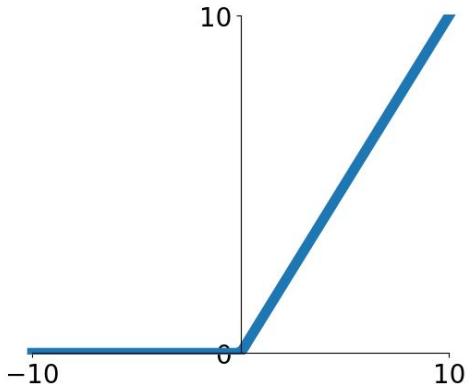
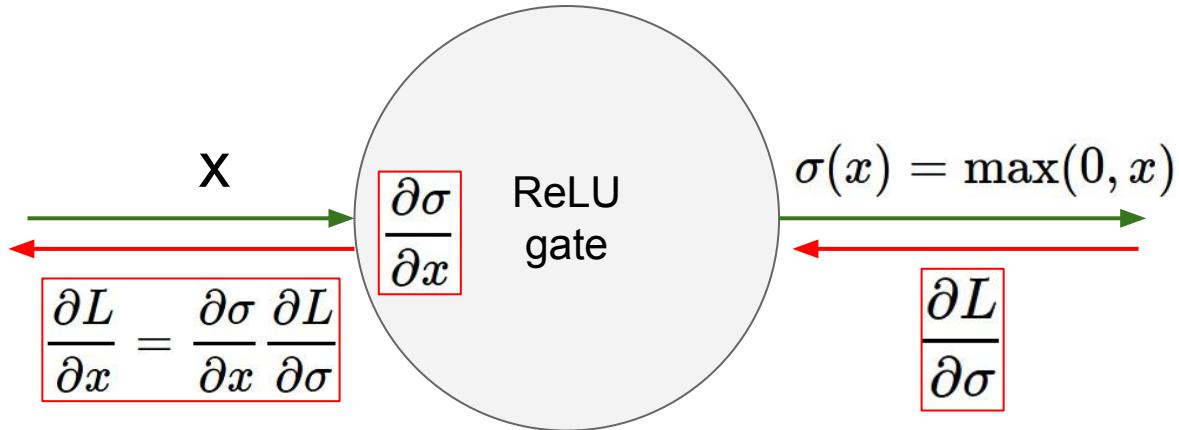


**ReLU**  
(Rectified Linear Unit)

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

- Not zero-centered output
- An annoyance:

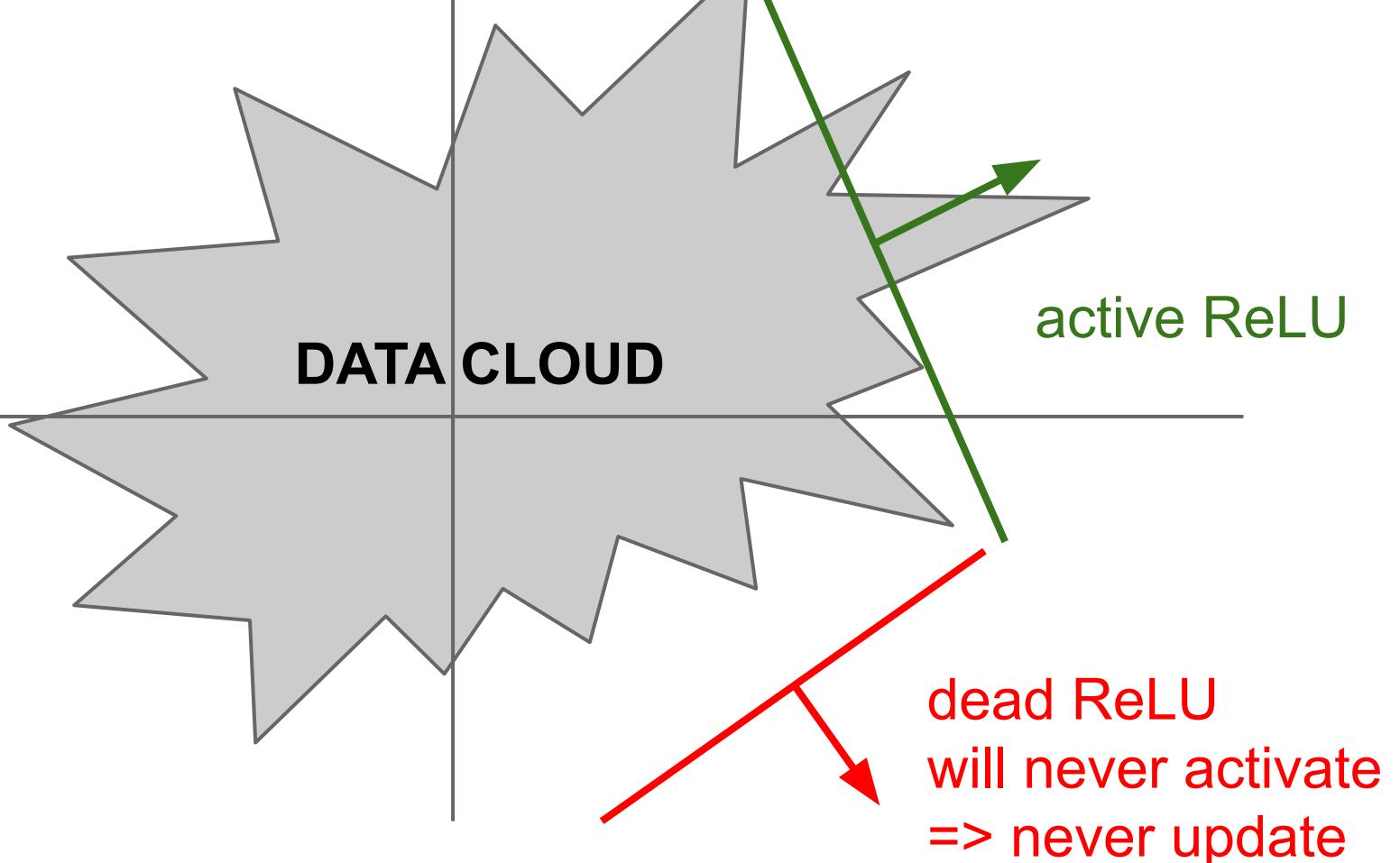
hint: what is the gradient when  $x < 0$ ?

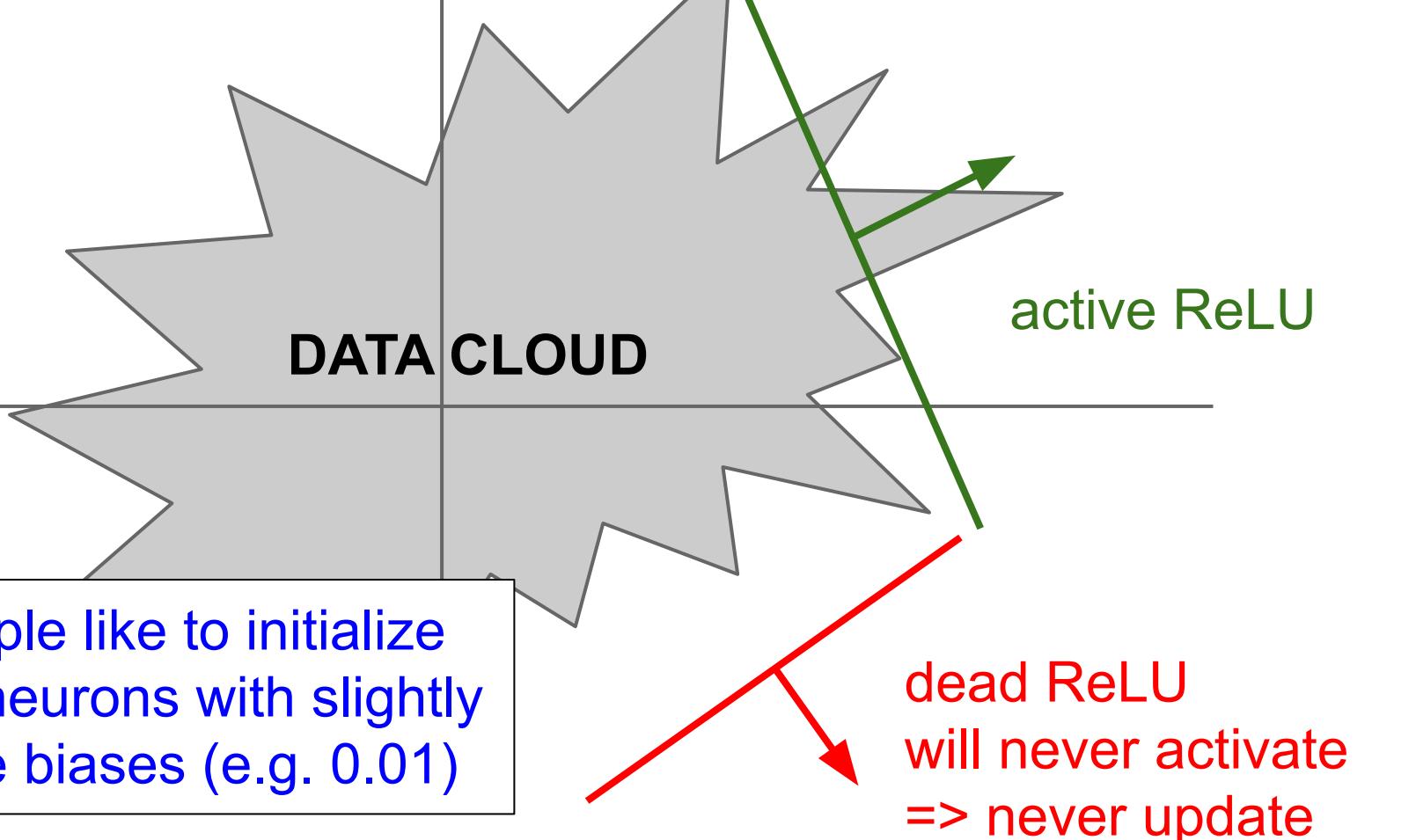


What happens when  $x = -10$ ?

What happens when  $x = 0$ ?

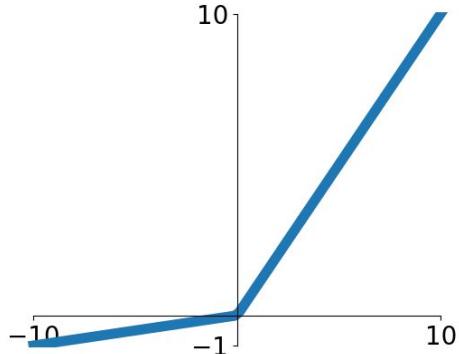
What happens when  $x = 10$ ?





# Activation Functions

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



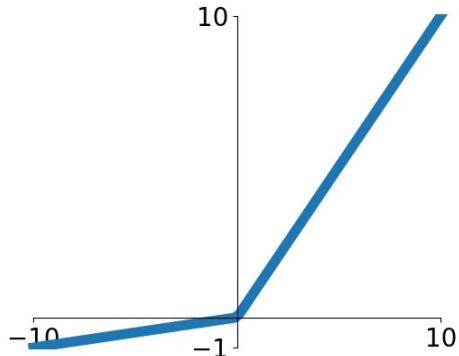
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

## Leaky ReLU

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

# Activation Functions

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



## Leaky ReLU

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

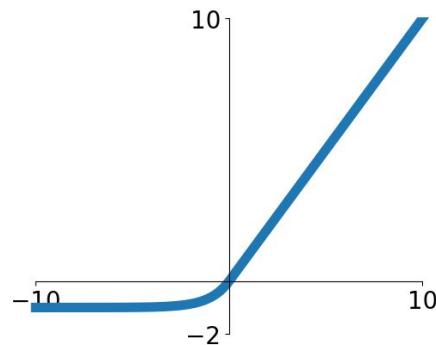
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

## Parametric Rectifier (PReLU)

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

backprop into  $\alpha$   
(parameter)

## Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

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

- Computation requires  $\exp()$

# Maxout “Neuron”

[Goodfellow et al., 2013]

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

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

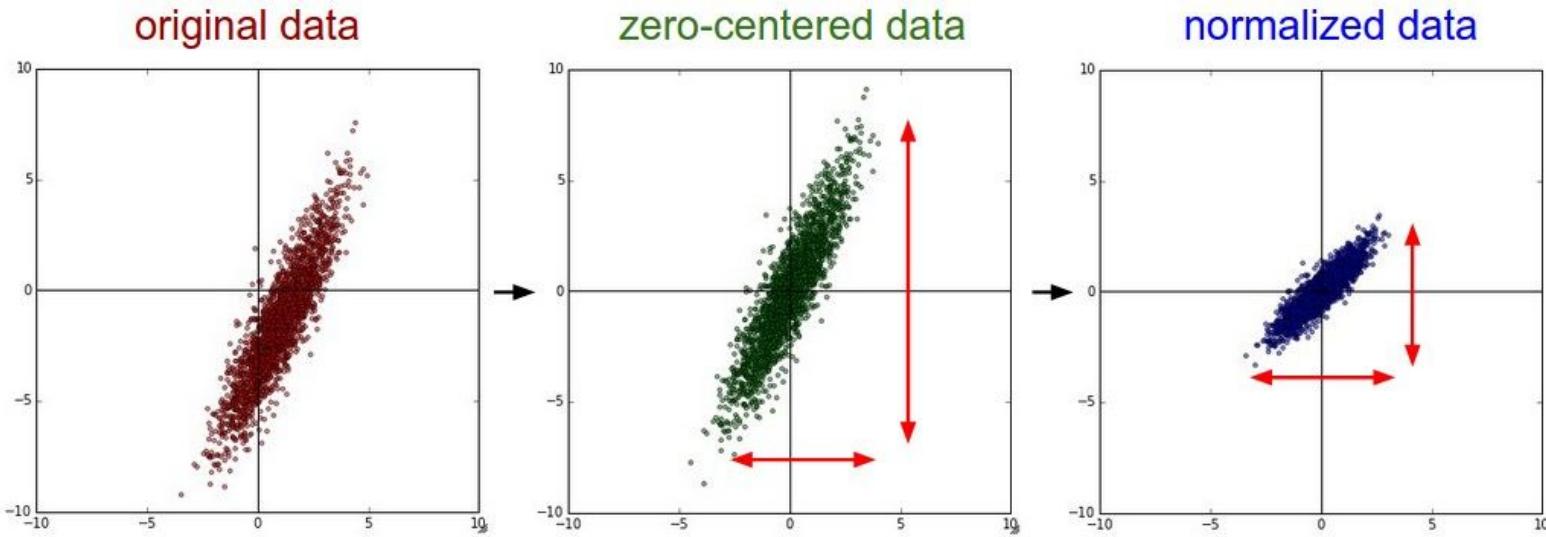
Problem: doubles the number of parameters/neuron :(

# TLDR: In practice:

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

# Data Preprocessing

# Data Preprocessing



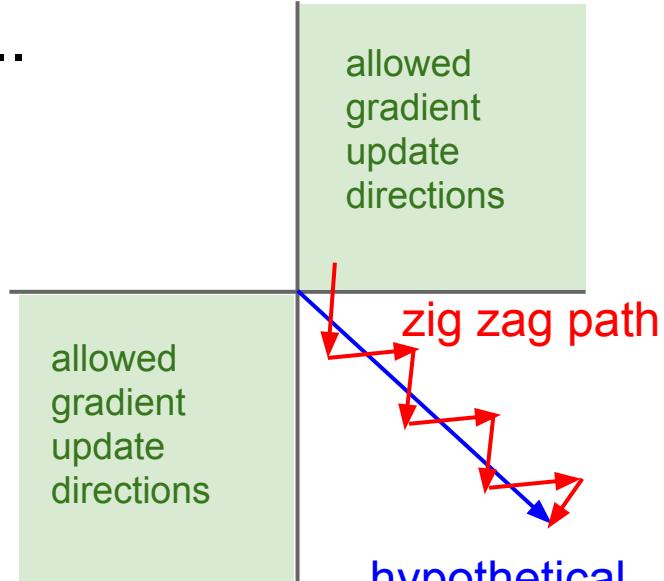
```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

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

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

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

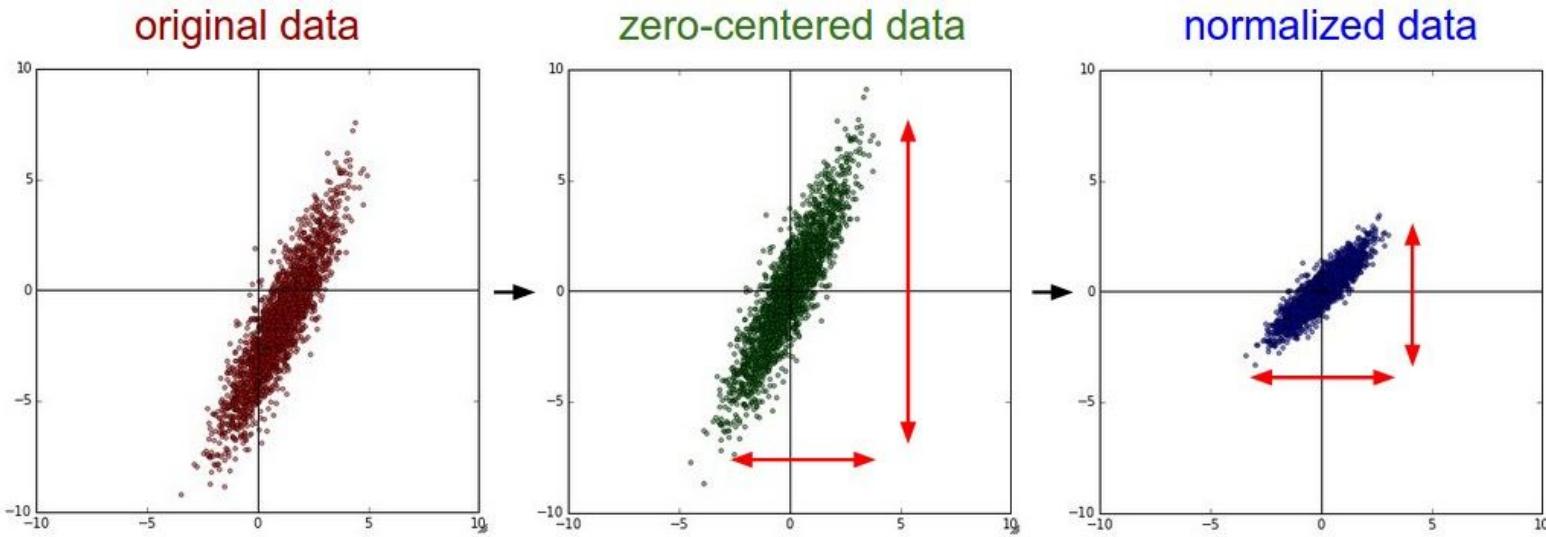


What can we say about the gradients on  $w$ ?

Always all positive or all negative :(

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

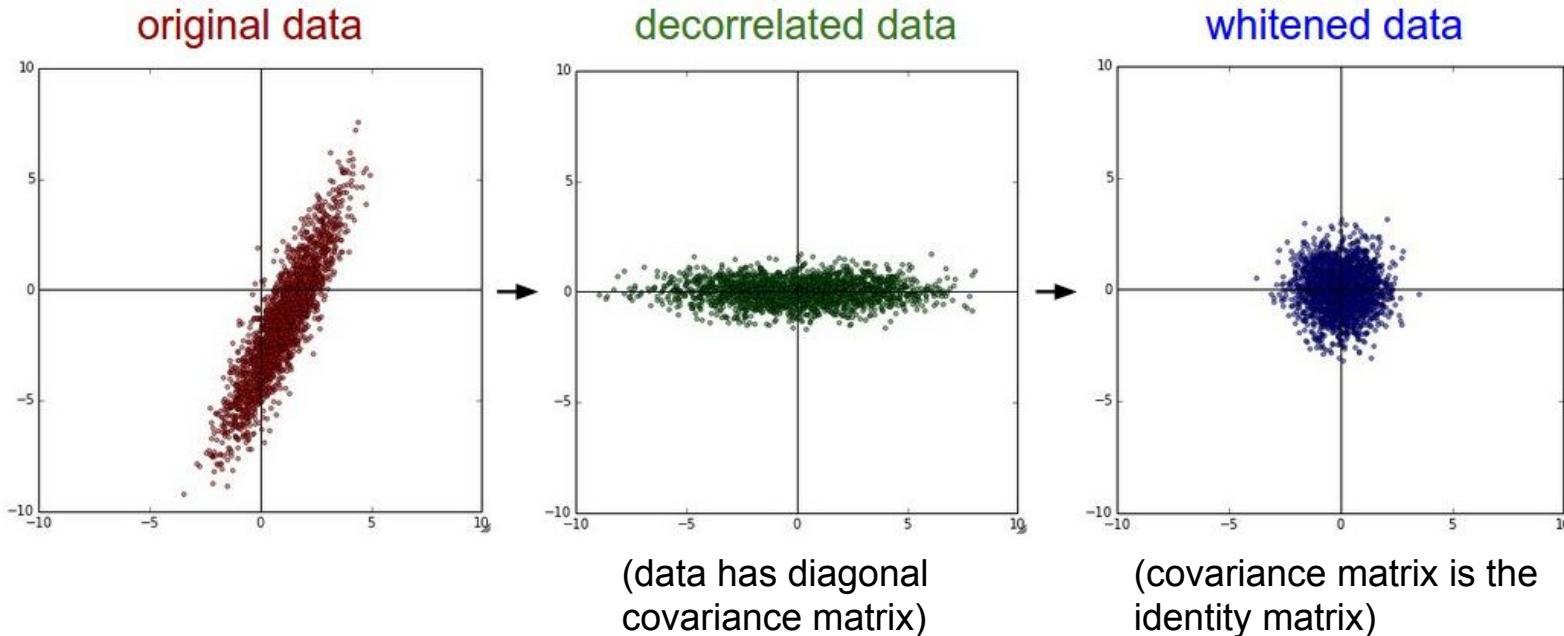
# Data Preprocessing



(Assume  $X$  [ $N \times D$ ] is data matrix, each example in a row)

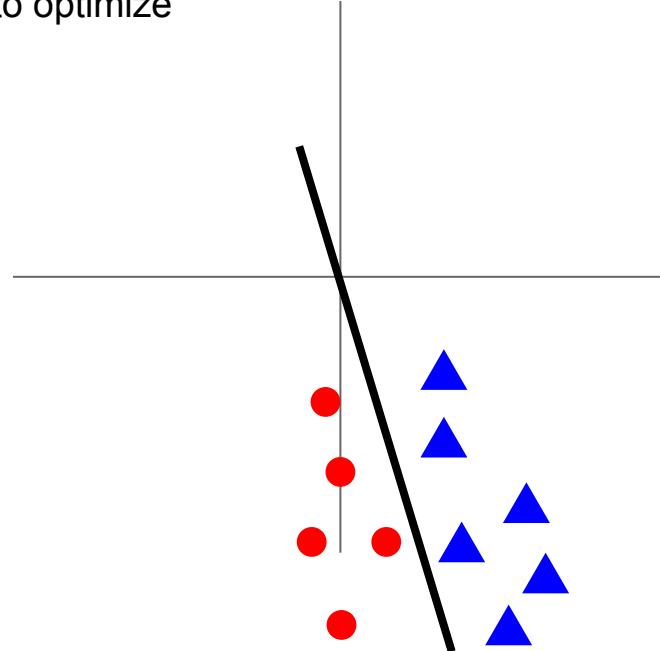
# Data Preprocessing

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

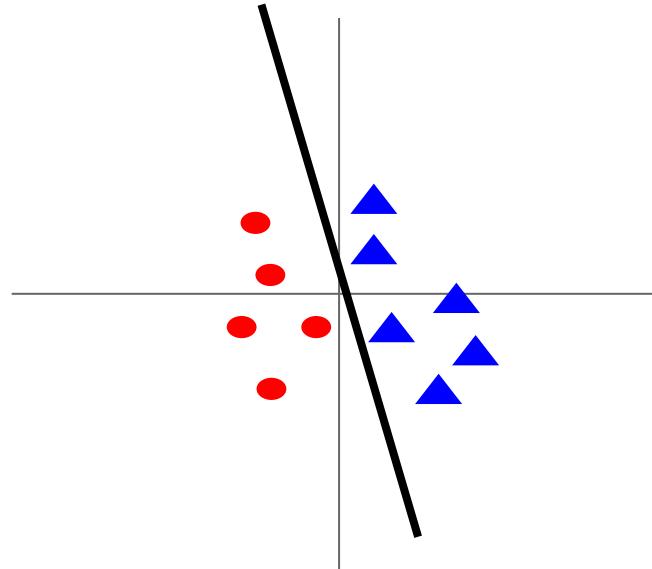


# Data Preprocessing

**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



# TLDR: In practice for Images: center only

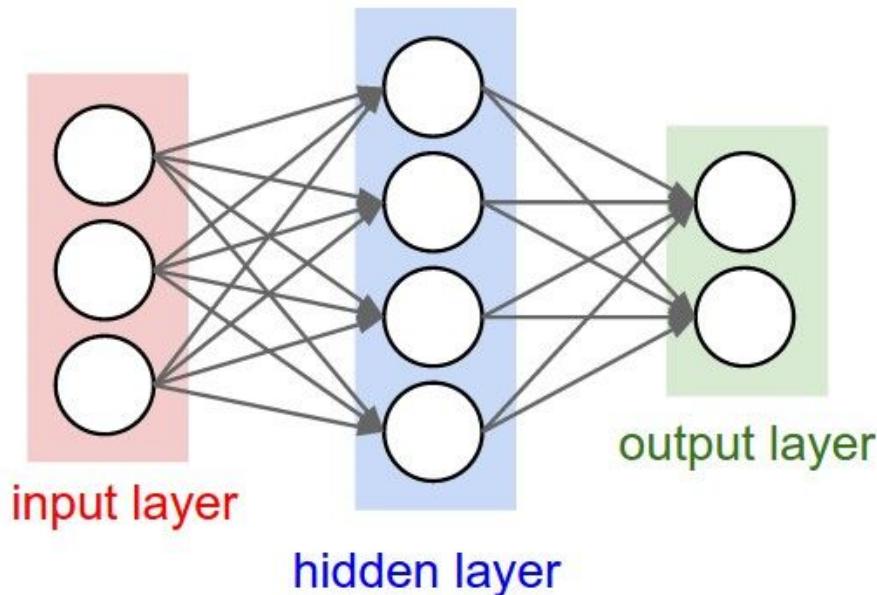
e.g. consider CIFAR-10 example with [32,32,3] images

- 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)
- Subtract per-channel mean and  
Divide by per-channel std (e.g. ResNet)  
(mean along each channel = 3 numbers)

Not common  
to do PCA or  
whitening

# Weight Initialization

- Q: what happens when  $W=\text{constant init}$  is used?



- First idea: **Small random numbers**  
(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01 * np.random.randn(Din, Dout)
```

- First idea: **Small random numbers**  
(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01 * np.random.randn(Din, Dout)
```

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

# Weight Initialization: Activation statistics

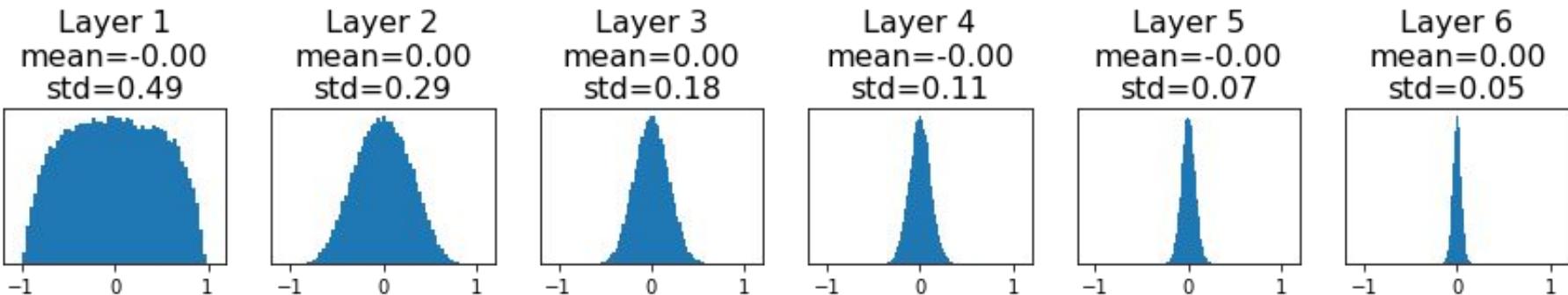
```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                  net with hidden size 4096  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.01 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

# Weight Initialization: Activation statistics

```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                  net with hidden size 4096  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.01 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

All activations tend to zero for deeper network layers

**Q:** What do the gradients  $dL/dW$  look like?



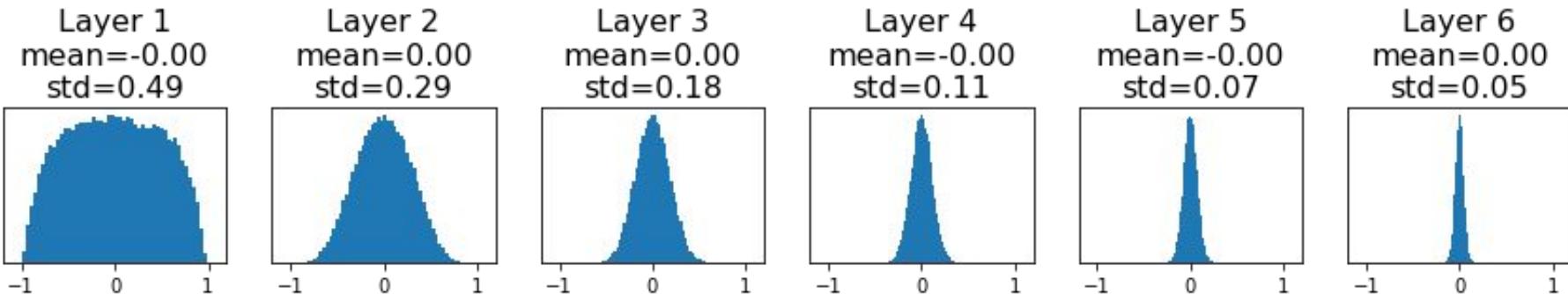
# Weight Initialization: Activation statistics

```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                  net with hidden size 4096  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.01 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

All activations tend to zero for deeper network layers

**Q:** What do the gradients  $dL/dW$  look like?

**A:** All zero, no learning =(



# Weight Initialization: Activation statistics

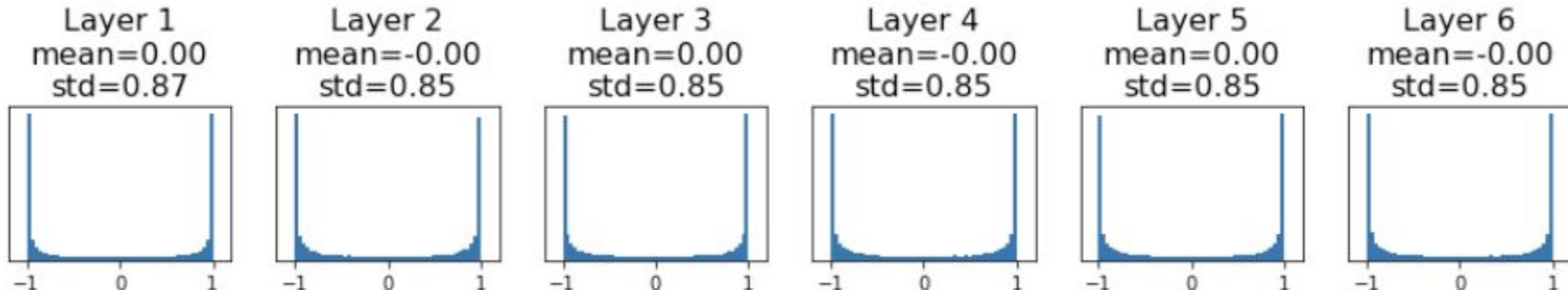
```
dims = [4096] * 7    Increase std of initial  
hs = []              weights from 0.01 to 0.05  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.05 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

# Weight Initialization: Activation statistics

```
dims = [4096] * 7    Increase std of initial  
hs = []                  weights from 0.01 to 0.05  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.05 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

All activations saturate

**Q:** What do the gradients look like?



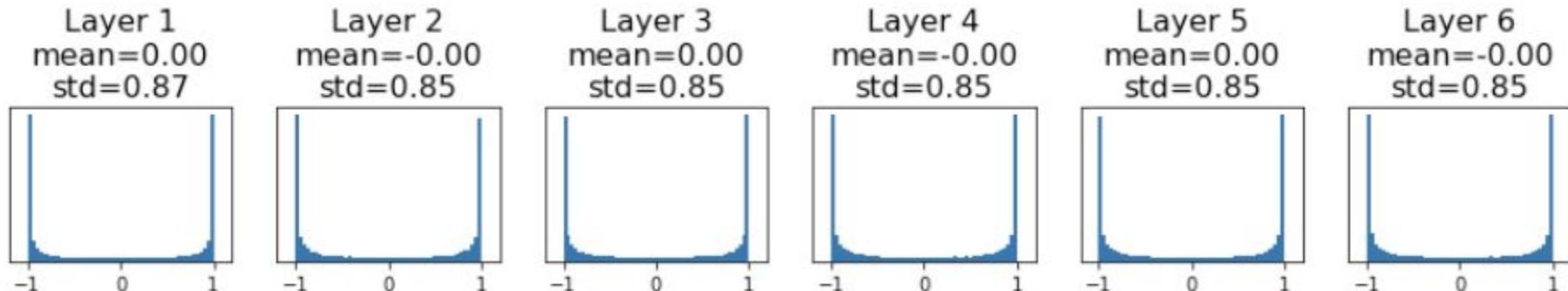
# Weight Initialization: Activation statistics

```
dims = [4096] * 7    Increase std of initial  
hs = []                  weights from 0.01 to 0.05  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.05 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

All activations saturate

**Q:** What do the gradients look like?

**A:** Local gradients all zero, no learning =(



# Weight Initialization: “Xavier” Initialization

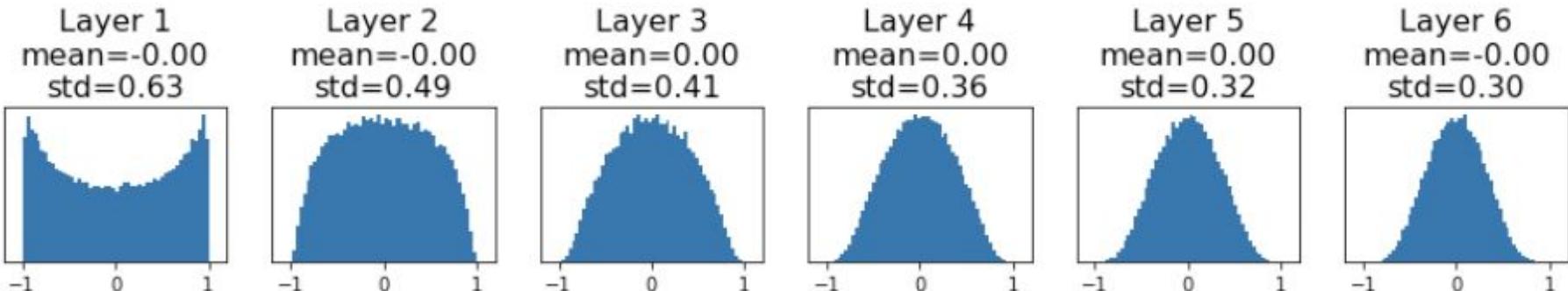
```
dims = [4096] * 7           "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

# Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!



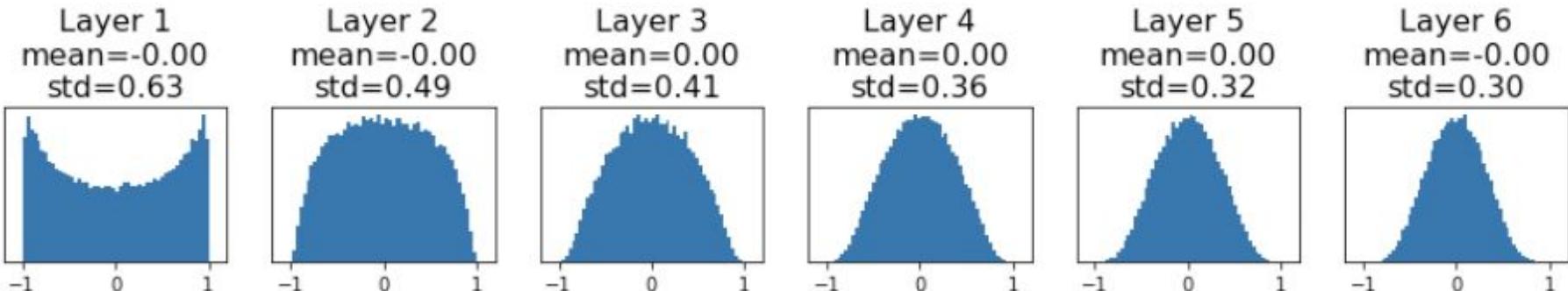
Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

# Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers,  $Din$  is  $\text{kernel\_size}^2 * \text{input\_channels}$



Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

# Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7           "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers,  $Din$  is  $\text{kernel\_size}^2 * \text{input\_channels}$

## Derivation:

$$y = Wx$$
$$h = f(y)$$

$$\begin{aligned}\text{Var}(y_i) &= \text{Din} * \text{Var}(x_i w_i) && [\text{Assume } x, w \text{ are iid}] \\ &= \text{Din} * (\mathbb{E}[x_i^2]\mathbb{E}[w_i^2] - \mathbb{E}[x_i]^2 \mathbb{E}[w_i]^2) && [\text{Assume } x, w \text{ independant}] \\ &= \text{Din} * \text{Var}(x_i) * \text{Var}(w_i) && [\text{Assume } x, w \text{ are zero-mean}]\end{aligned}$$

If  $\text{Var}(w_i) = 1/\text{Din}$  then  $\text{Var}(y_i) = \text{Var}(x_i)$

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

# Weight Initialization: What about ReLU?

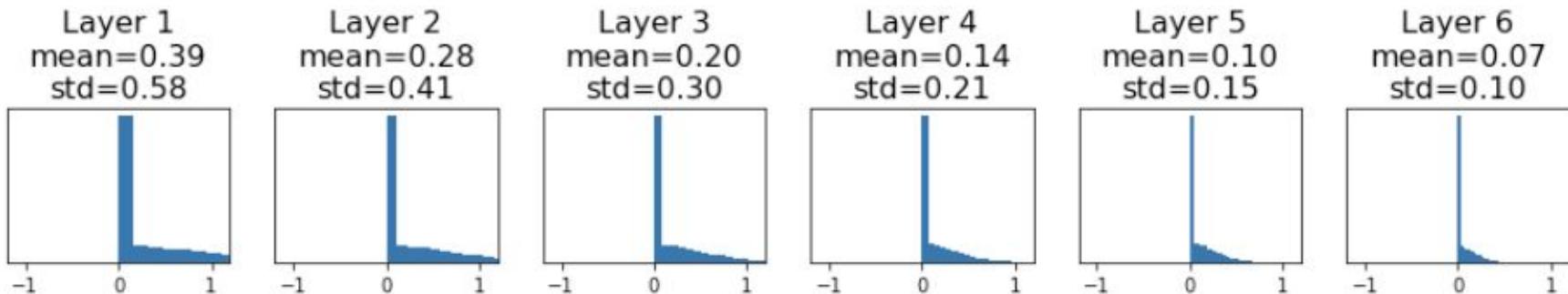
```
dims = [4096] * 7      Change from tanh to ReLU
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

# Weight Initialization: What about ReLU?

```
dims = [4096] * 7      Change from tanh to ReLU
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Xavier assumes zero centered activation function

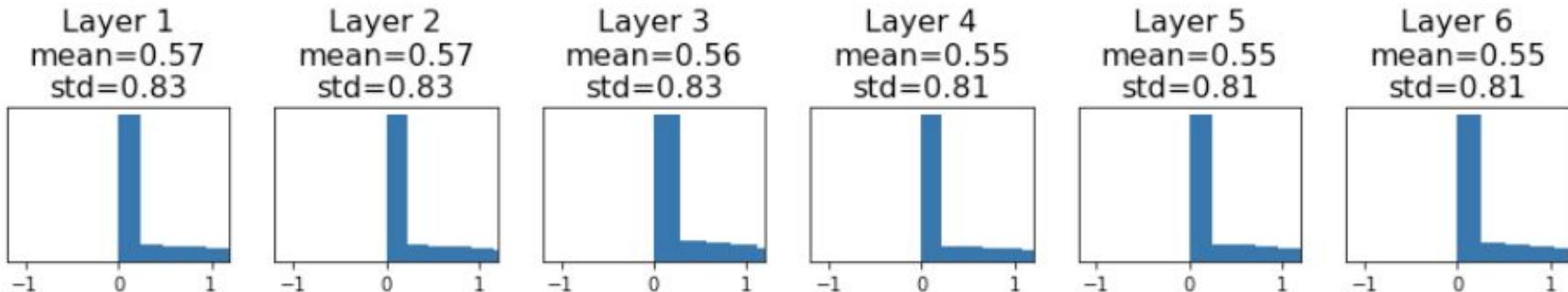
Activations collapse to zero again, no learning =(



# Weight Initialization: Kaiming / MSRA Initialization

```
dims = [4096] * 7  ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!



He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

# Proper initialization is an active area of research...

***Understanding the difficulty of training deep feedforward neural networks***

by Glorot and Bengio, 2010

***Exact solutions to the nonlinear dynamics of learning in deep linear neural networks*** by Saxe et al, 2013

***Random walk initialization for training very deep feedforward networks*** by Sussillo and Abbott, 2014

***Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification*** by He et al., 2015

***Data-dependent Initializations of Convolutional Neural Networks*** by Krähenbühl et al., 2015

***All you need is a good init***, Mishkin and Matas, 2015

***Fixup Initialization: Residual Learning Without Normalization***, Zhang et al, 2019

***The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks***, Frankle and Carbin, 2019

# Batch Normalization

# Batch Normalization

[Ioffe and Szegedy, 2015]

“you want zero-mean unit-variance activations? just make them so.”

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

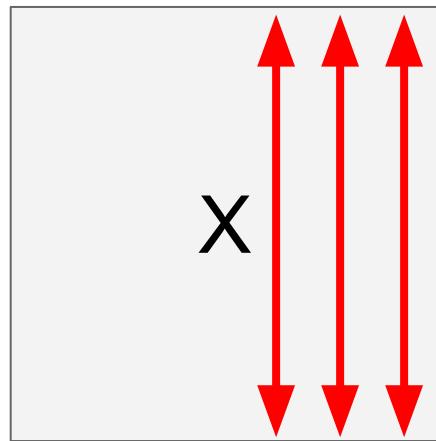
$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla  
differentiable function...

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:**  $x : N \times D$



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,  
shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel var,  
shape is D

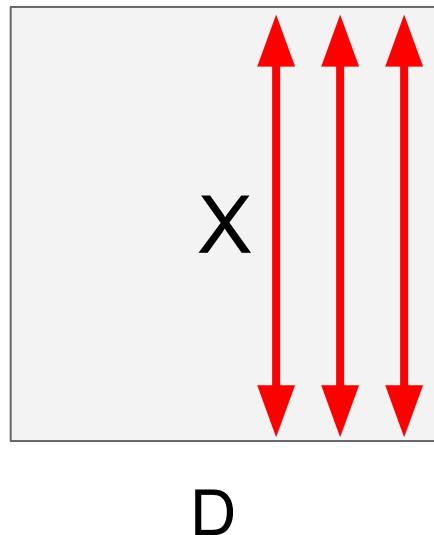
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x,  
Shape is  $N \times D$

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:**  $x : N \times D$



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,  
shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel var,  
shape is D

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

Normalized x,  
Shape is  $N \times D$

Problem: What if zero-mean, unit variance is too hard of a constraint?

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:**  $x : N \times D$

**Learnable scale and shift parameters:**

$\gamma, \beta : D$

Learning  $\gamma = \sigma$ ,  
 $\beta = \mu$  will recover the identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,  
shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel var,  
shape is D

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

Normalized x,  
Shape is  $N \times D$

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

Output,  
Shape is  $N \times D$

# Batch Normalization: Test-Time

Estimates depend on minibatch;  
can't do this at test-time!

**Input:**  $x : N \times D$

**Learnable scale and shift parameters:**

$\gamma, \beta : D$

Learning  $\gamma = \sigma$ ,  
 $\beta = \mu$  will recover the identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,  
shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel var,  
shape is D

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

Normalized x,  
Shape is  $N \times D$

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

Output,  
Shape is  $N \times D$

# Batch Normalization: Test-Time

**Input:**  $x : N \times D$

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

Per-channel mean,  
shape is D

**Learnable scale and  
shift parameters:**

$\gamma, \beta : D$

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

Per-channel var,  
shape is D

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

Normalized x,  
Shape is N x D

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

Output,  
Shape is N x D

During testing batchnorm  
becomes a linear operator!  
Can be fused with the previous  
fully-connected or conv layer

# Batch Normalization: Test-Time

**Input:**  $x : N \times D$

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

Per-channel mean,  
shape is D

**Learnable scale and  
shift parameters:**

$\gamma, \beta : D$

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

Per-channel var,  
shape is D

Learning  $\gamma = \sigma$ ,  
 $\beta = \mu$  will recover the  
identity function!

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

Normalized x,  
Shape is N x D

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

Output,  
Shape is N x D

# Batch Normalization for ConvNets

Batch Normalization for  
**fully-connected** networks

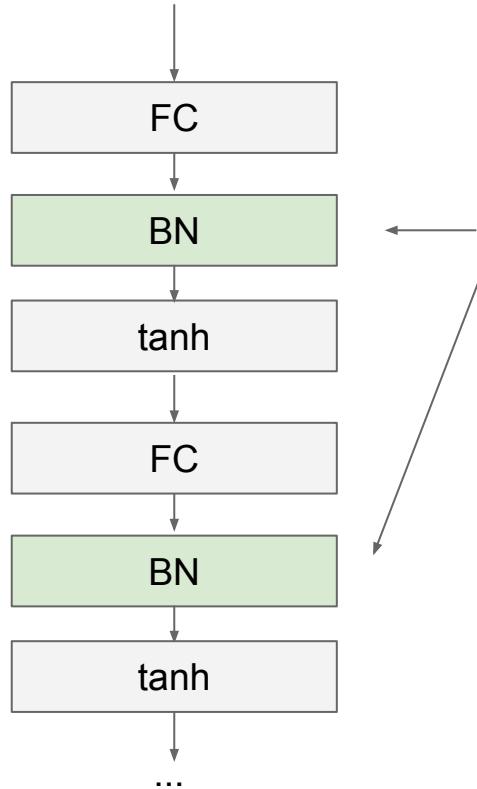
$$\begin{aligned} \mathbf{x} &: N \times D \\ \text{Normalize} & \quad \downarrow \\ \boldsymbol{\mu}, \sigma &: 1 \times D \\ \boldsymbol{\gamma}, \beta &: 1 \times D \\ \mathbf{y} &= \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \beta \end{aligned}$$

Batch Normalization for  
**convolutional** networks  
(Spatial Batchnorm, BatchNorm2D)

$$\begin{aligned} \mathbf{x} &: N \times C \times H \times W \\ \text{Normalize} & \quad \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \sigma &: 1 \times C \times 1 \times 1 \\ \boldsymbol{\gamma}, \beta &: 1 \times C \times 1 \times 1 \\ \mathbf{y} &= \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \beta \end{aligned}$$

# Batch Normalization

[Ioffe and Szegedy, 2015]

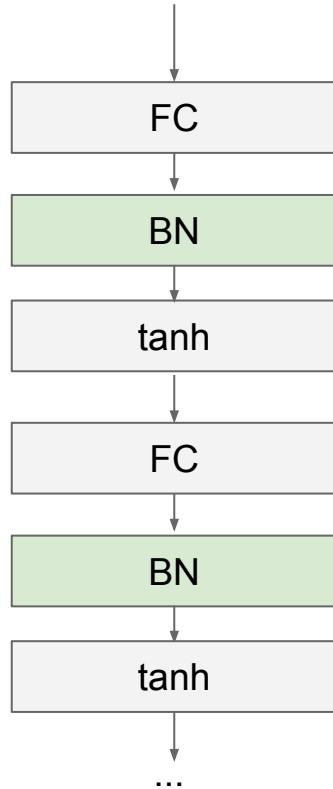


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

# Batch Normalization

[Ioffe and Szegedy, 2015]



- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

# Layer Normalization

**Batch Normalization** for  
fully-connected networks

**Layer Normalization** for  
fully-connected networks  
Same behavior at train and test!  
Can be used in recurrent networks

$\mathbf{x}: N \times D$

Normalize



$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$$\mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \boldsymbol{\beta}$$

$\mathbf{x}: N \times D$

Normalize



$\mu, \sigma: N \times 1$

$\gamma, \beta: 1 \times D$

$$\mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \boldsymbol{\beta}$$

Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

# Instance Normalization

**Batch Normalization** for convolutional networks

$\mathbf{x}$ :  $N \times C \times H \times W$

Normalize



$\mu, \sigma$ :  $1 \times C \times 1 \times 1$

$\gamma, \beta$ :  $1 \times C \times 1 \times 1$

$$\mathbf{y} = \gamma(\mathbf{x} - \mu) / \sigma + \beta$$

**Instance Normalization** for convolutional networks  
Same behavior at train / test!

$\mathbf{x}$ :  $N \times C \times H \times W$

Normalize



$\mu, \sigma$ :  $N \times C \times 1 \times 1$

$\gamma, \beta$ :  $1 \times C \times 1 \times 1$

$$\mathbf{y} = \gamma(\mathbf{x} - \mu) / \sigma + \beta$$

Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

# Summary

TLDRs

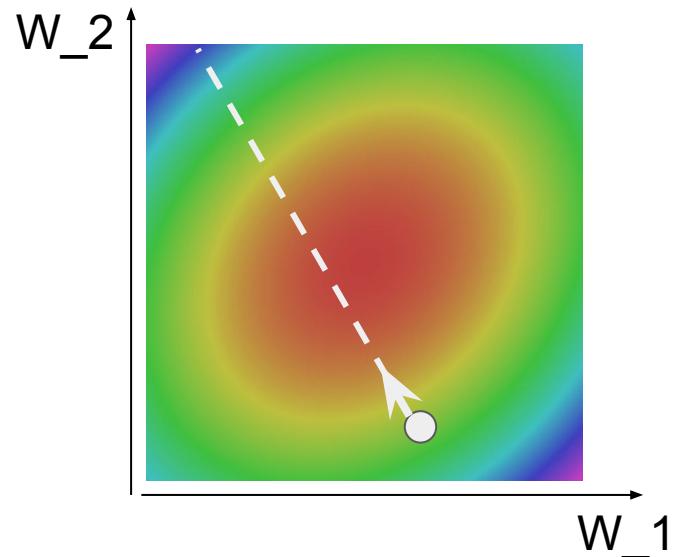
We looked in detail at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/He init)
- Batch Normalization (use)

# Optimization

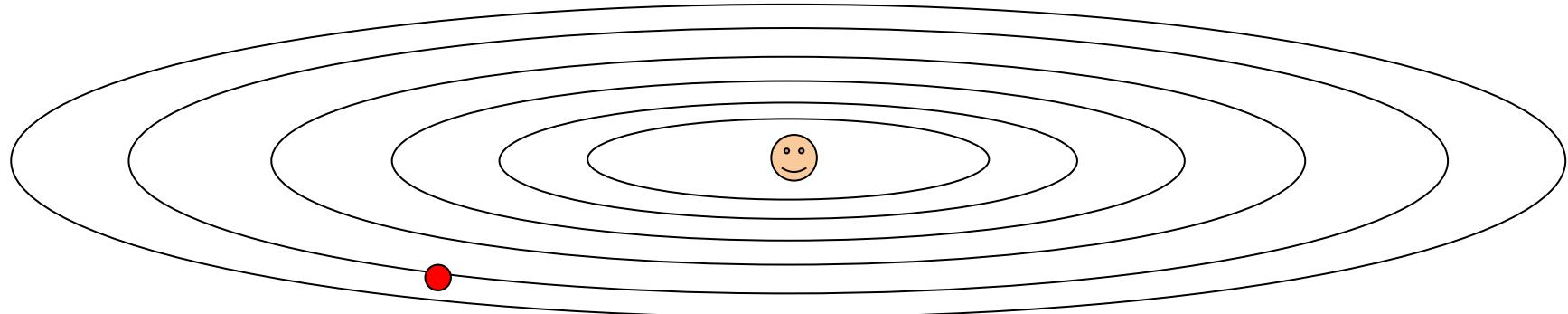
```
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad # perform parameter update
```



# Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?  
What does gradient descent do?



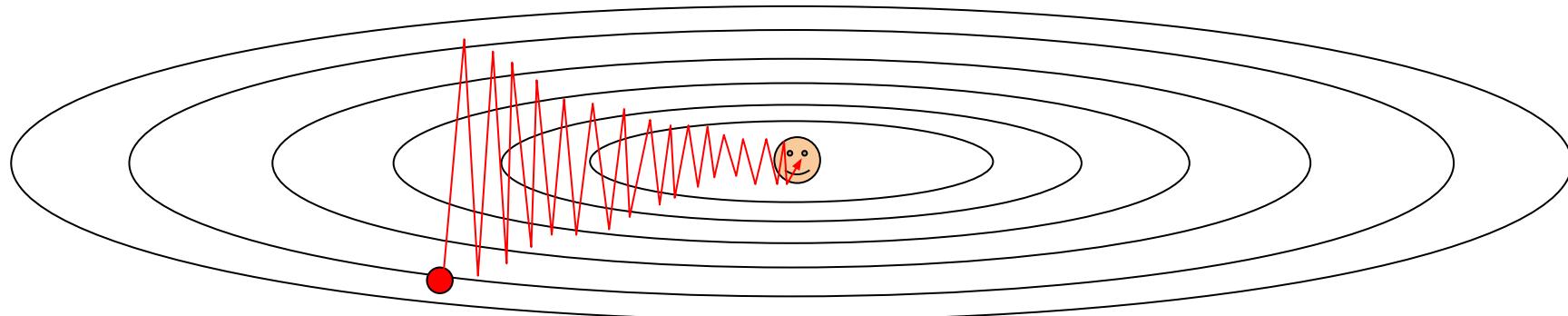
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

# Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?

What does gradient descent do?

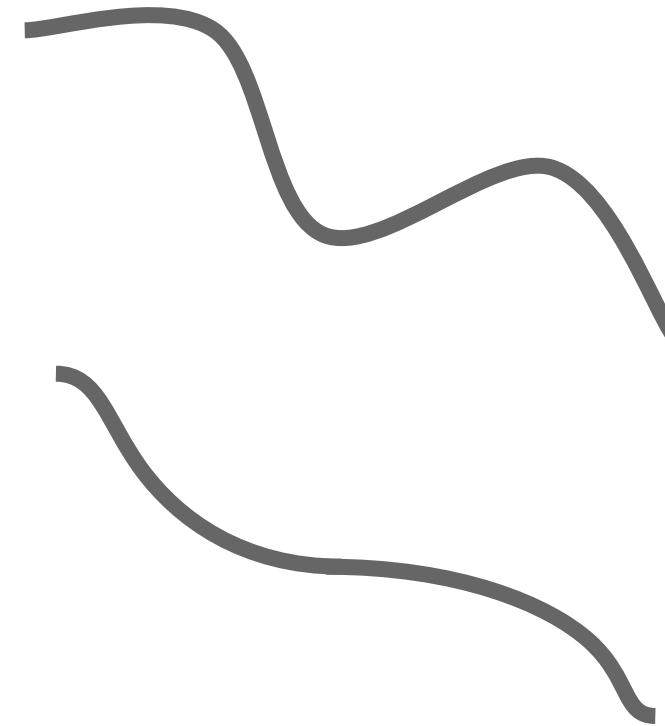
Very slow progress along shallow dimension, jitter along steep direction



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

# Optimization: Problems with SGD

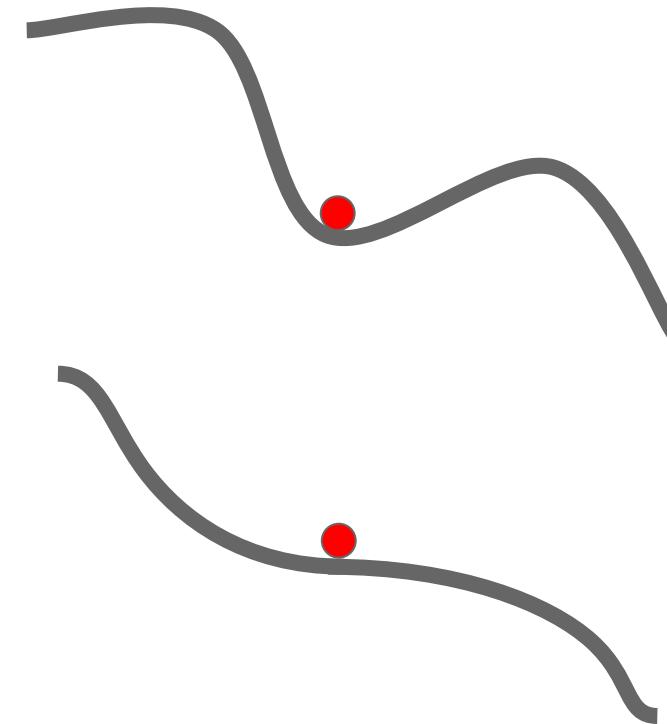
What if the loss  
function has a  
**local minima** or  
**saddle point**?



# Optimization: Problems with SGD

What if the loss  
function has a  
**local minima** or  
**saddle point**?

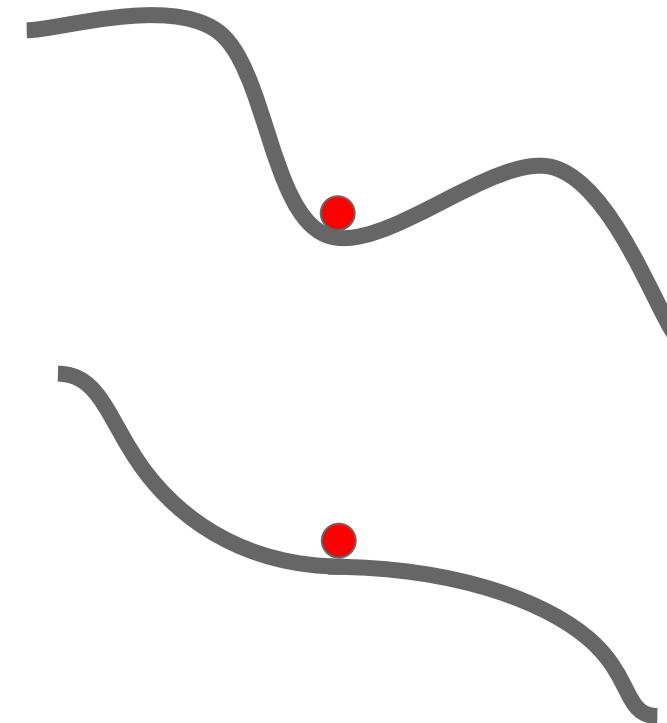
Zero gradient,  
gradient descent  
gets stuck



# Optimization: Problems with SGD

What if the loss  
function has a  
**local minima** or  
**saddle point**?

Saddle points much  
more common in  
high dimension



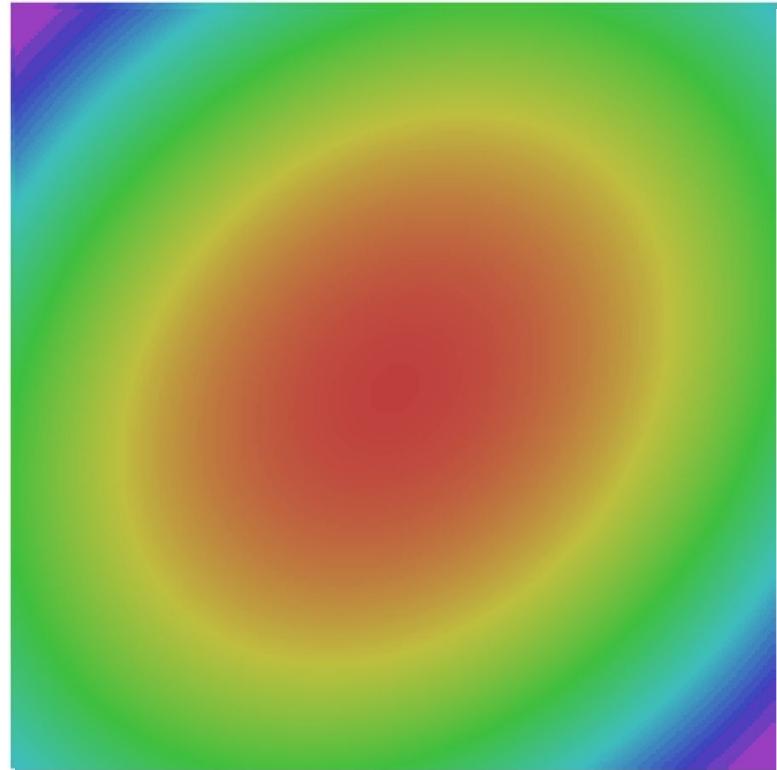
Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

# Optimization: Problems with SGD

Our gradients come from  
minibatches so they can be noisy!

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

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W)$$



# SGD + Momentum

## SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:  
    dx = compute_gradient(x)  
    x -= learning_rate * dx
```

## SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0  
while True:  
    dx = compute_gradient(x)  
    vx = rho * vx + dx  
    x -= learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

# SGD + Momentum

## SGD+Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx - learning_rate * dx
    x += vx
```

## SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

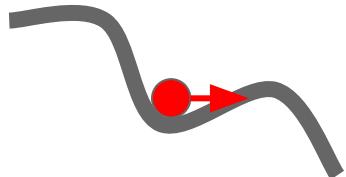
```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```

You may see SGD+Momentum formulated different ways,  
but they are equivalent - give same sequence of  $x$

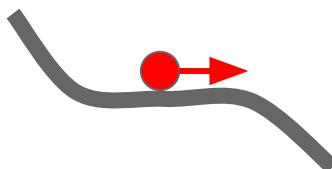
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

# SGD + Momentum

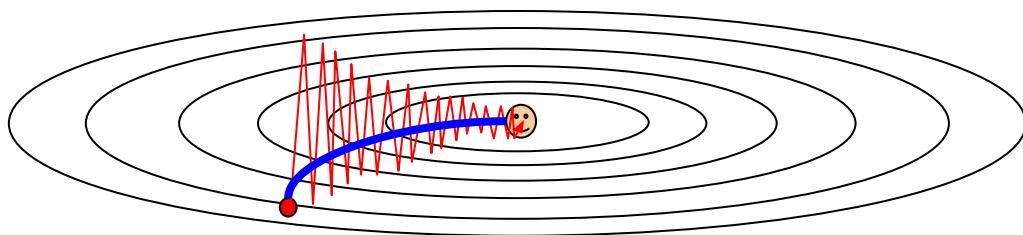
Local Minima



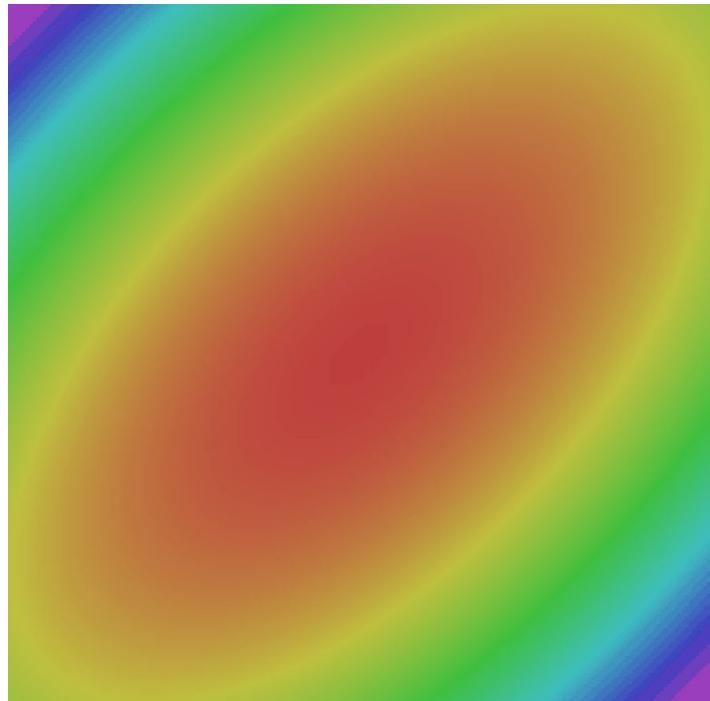
Saddle points



Poor Conditioning



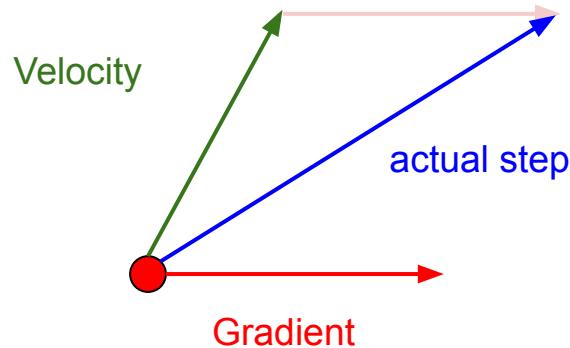
# Gradient Noise



— SGD      — SGD+Momentum

# SGD+Momentum

Momentum update:



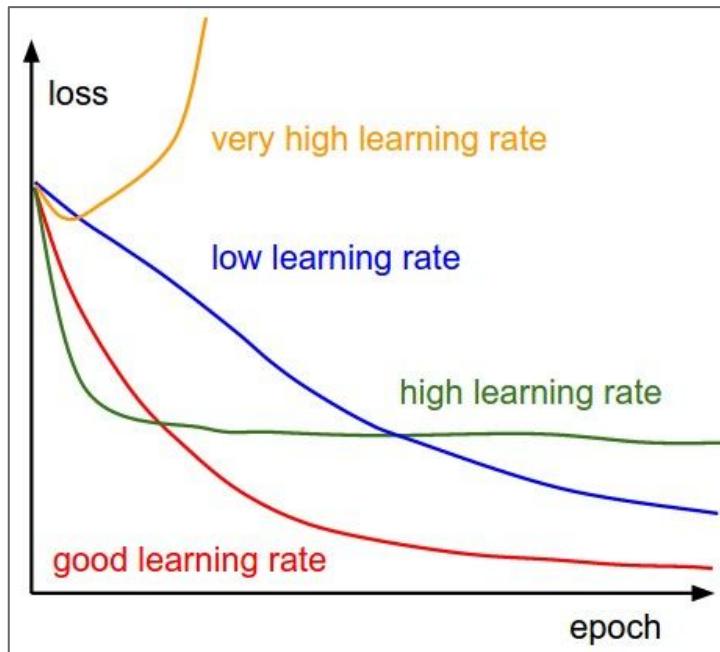
Combine gradient at current point with velocity to get step used to update weights

Nesterov, "A method of solving a convex programming problem with convergence rate  $O(1/k^2)$ ", 1983

Nesterov, "Introductory lectures on convex optimization: a basic course", 2004

Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

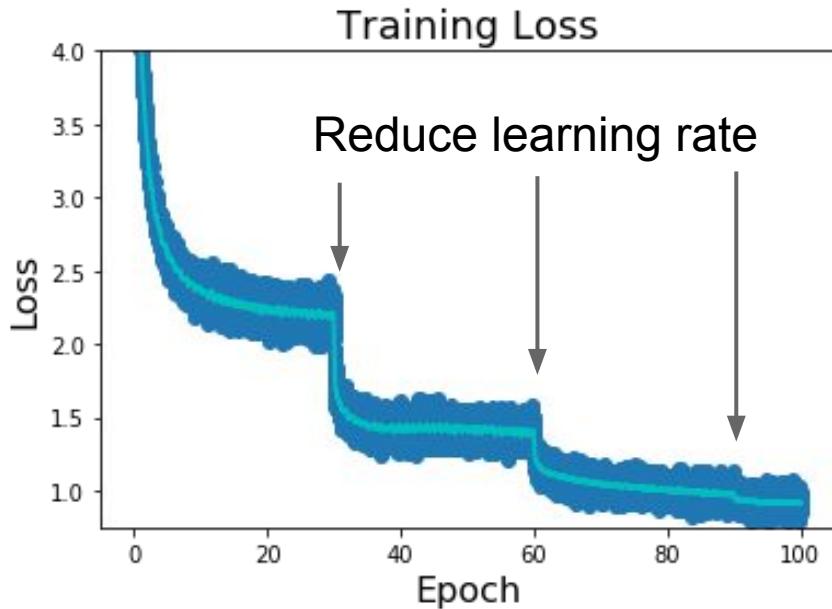
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



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

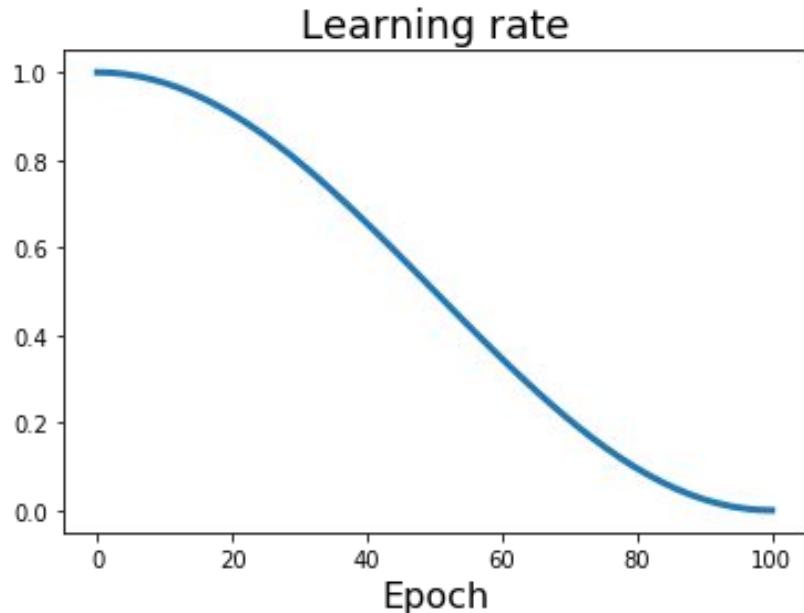
A: All of them! Start with large learning rate and decay over time

# Learning Rate Decay



**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

# Learning Rate Decay



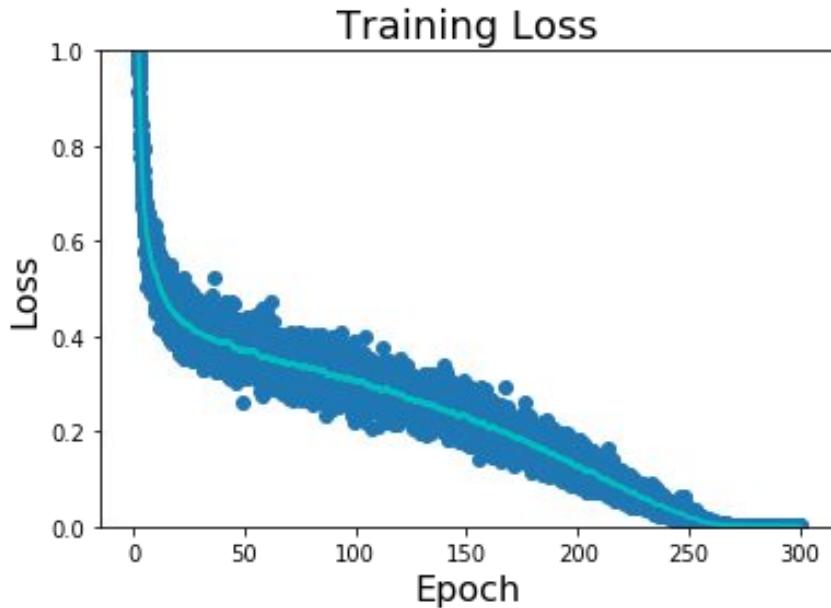
**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**  $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

$\alpha_0$  : Initial learning rate  
 $\alpha_t$  : Learning rate at epoch t  
 $T$  : Total number of epochs

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017  
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018  
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018  
Child et al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

# Learning Rate Decay



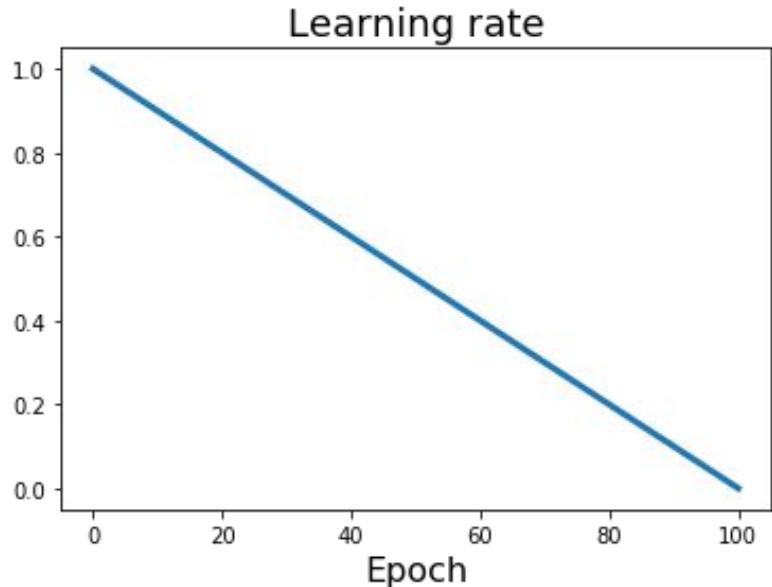
**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**  $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

$\alpha_0$  : Initial learning rate  
 $\alpha_t$  : Learning rate at epoch t  
 $T$  : Total number of epochs

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017  
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018  
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018  
Child et al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

# Learning Rate Decay



**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**  $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

**Linear:**  $\alpha_t = \alpha_0(1 - t/T)$

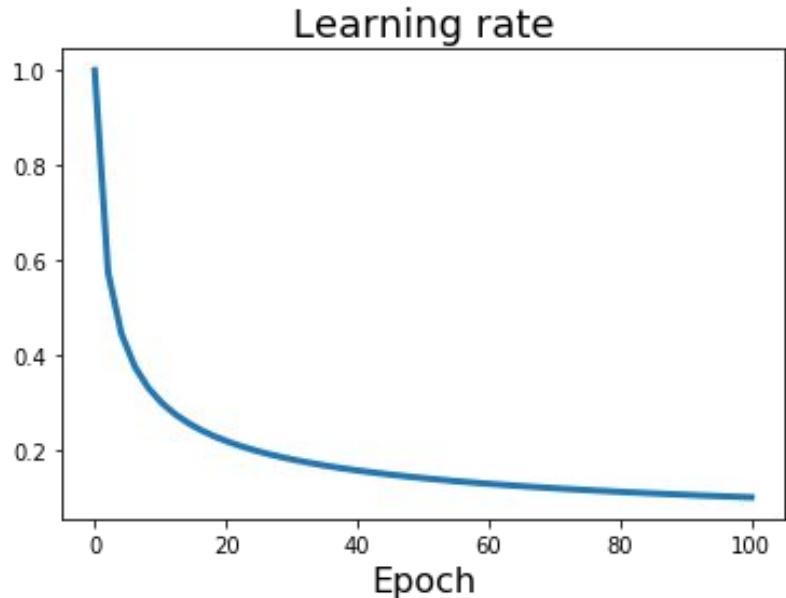
$\alpha_0$  : Initial learning rate

$\alpha_t$  : Learning rate at epoch  $t$

$T$  : Total number of epochs

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018

# Learning Rate Decay



**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**  $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

**Linear:**  $\alpha_t = \alpha_0(1 - t/T)$

**Inverse sqrt:**  $\alpha_t = \alpha_0/\sqrt{t}$

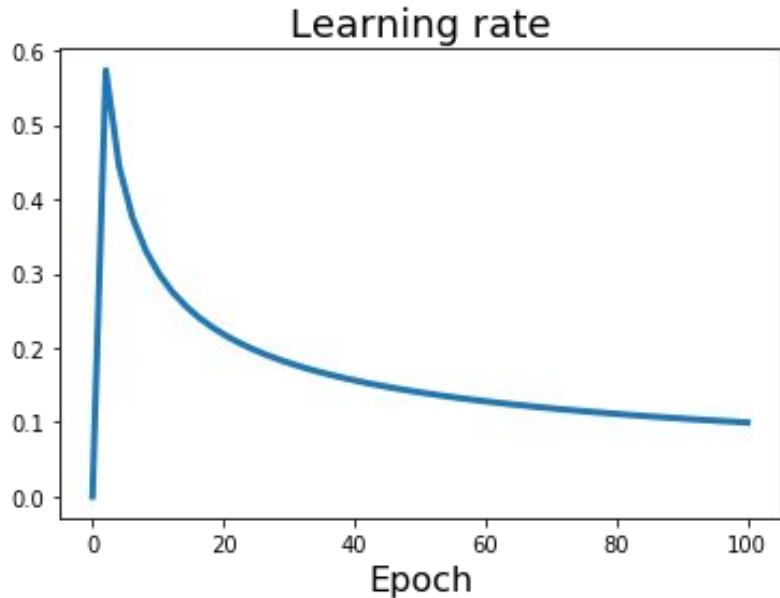
$\alpha_0$  : Initial learning rate

$\alpha_t$  : Learning rate at epoch  $t$

$T$  : Total number of epochs

Vaswani et al, "Attention is all you need", NIPS 2017

# Learning Rate Decay: Linear Warmup

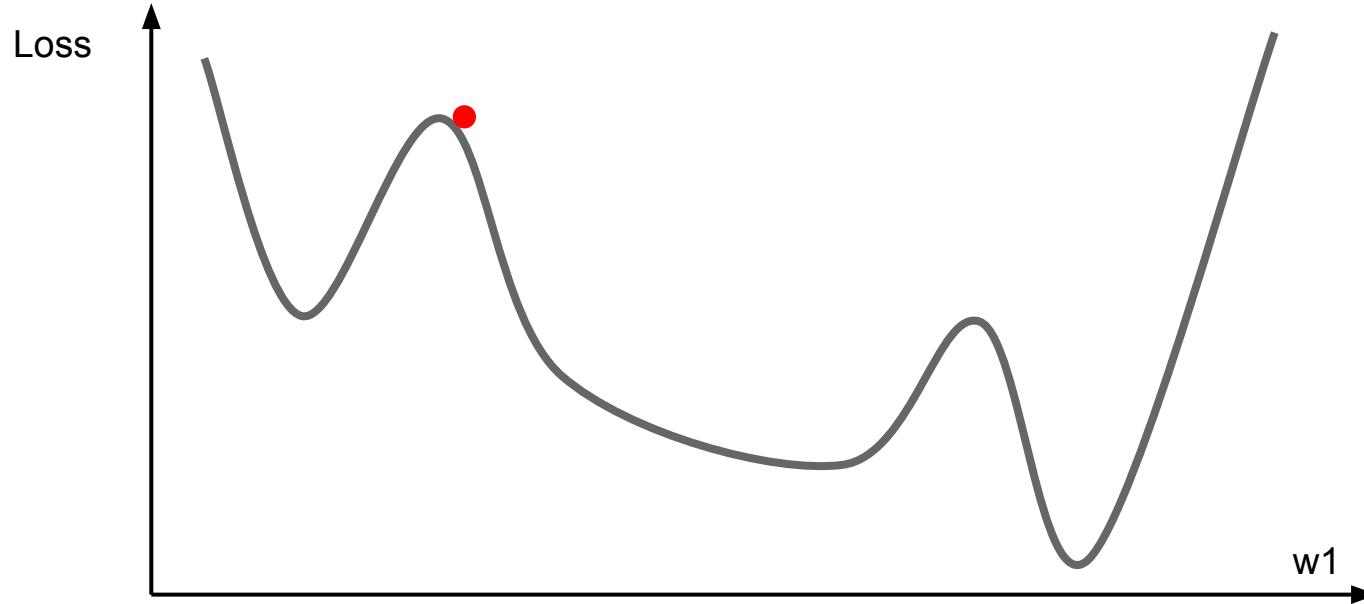


High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this

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

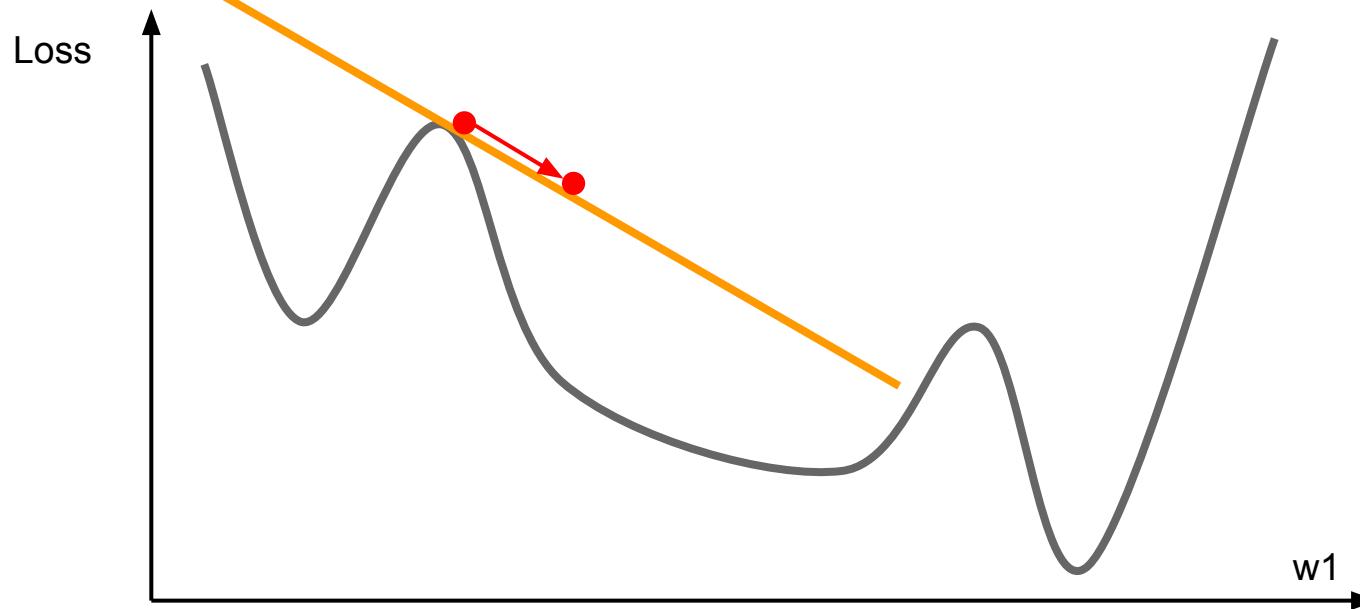
Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

# First-Order Optimization



# First-Order Optimization

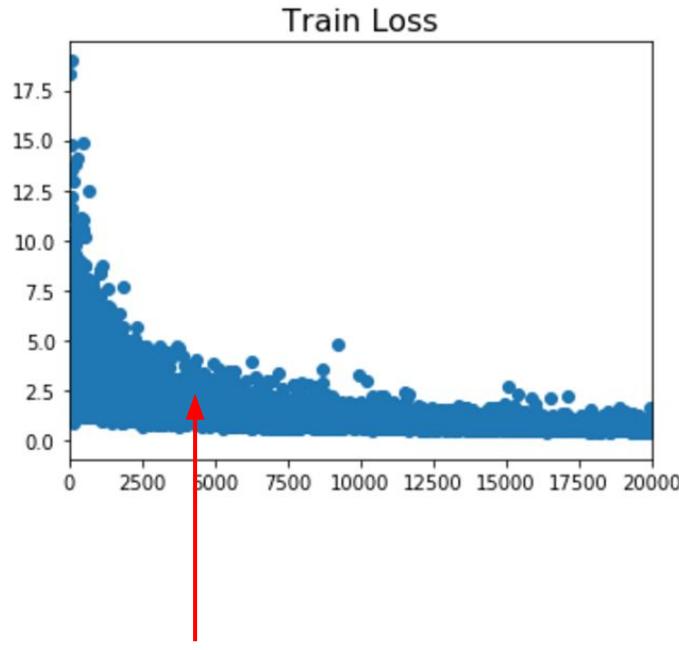
- (1) Use gradient form linear approximation
- (2) Step to minimize the approximation



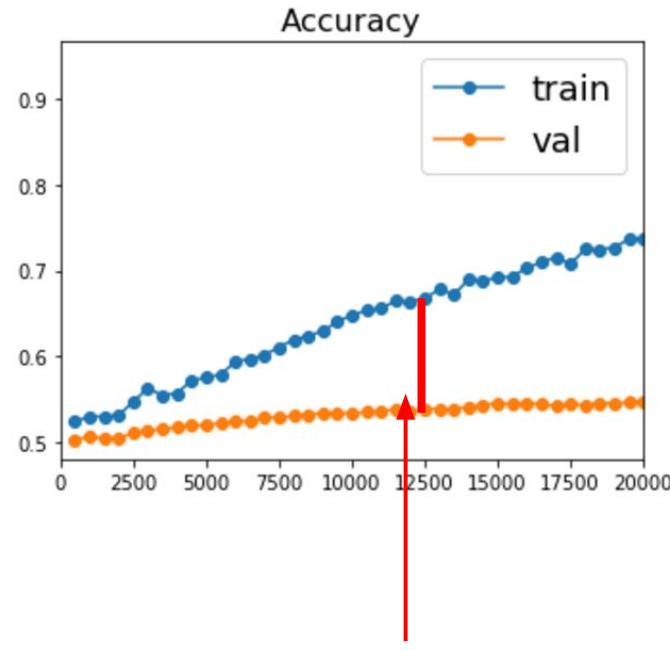
# In practice:

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule, very few hyperparameters!
- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

# Beyond Training Error

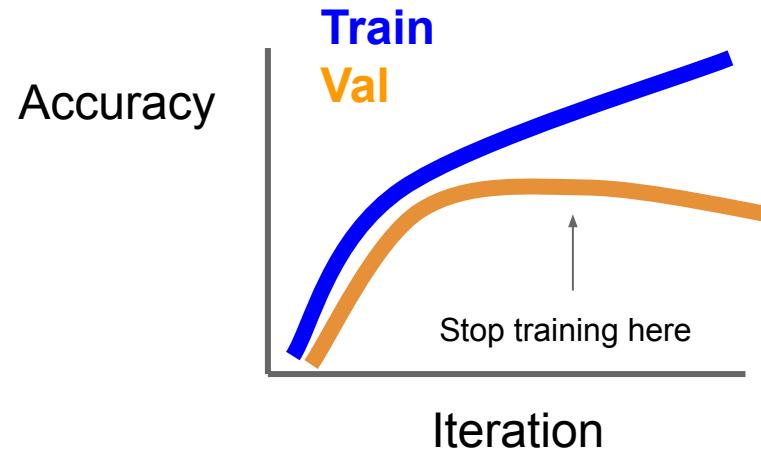
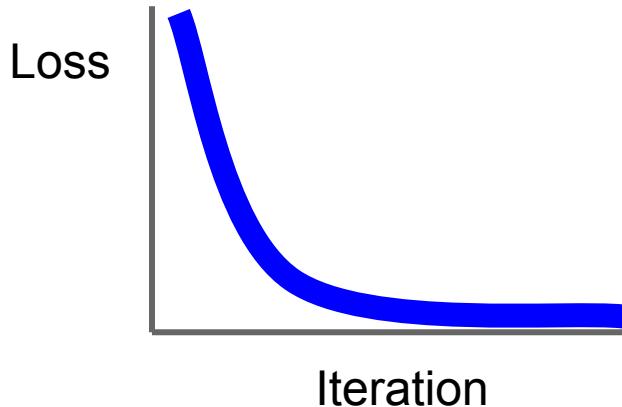


Better optimization algorithms help reduce training loss



But we really care about error on new data - how to reduce the gap?

# Early Stopping: Always do this



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

# Model Ensembles

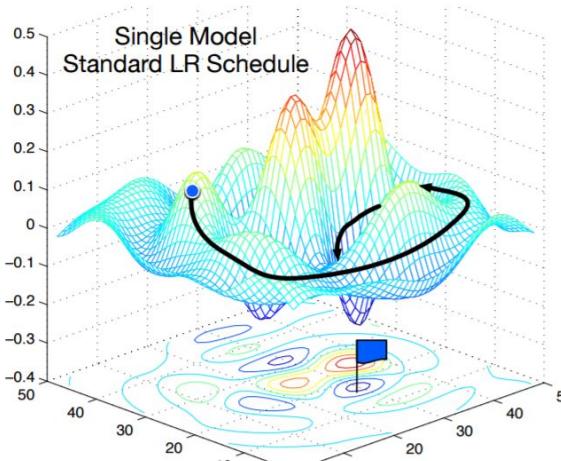
1. Train multiple independent models
2. At test time average their results

(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!



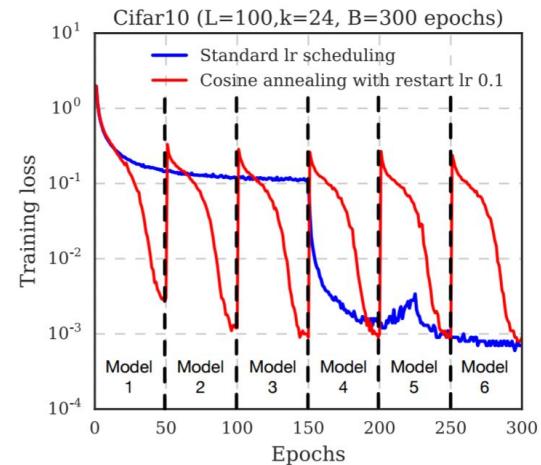
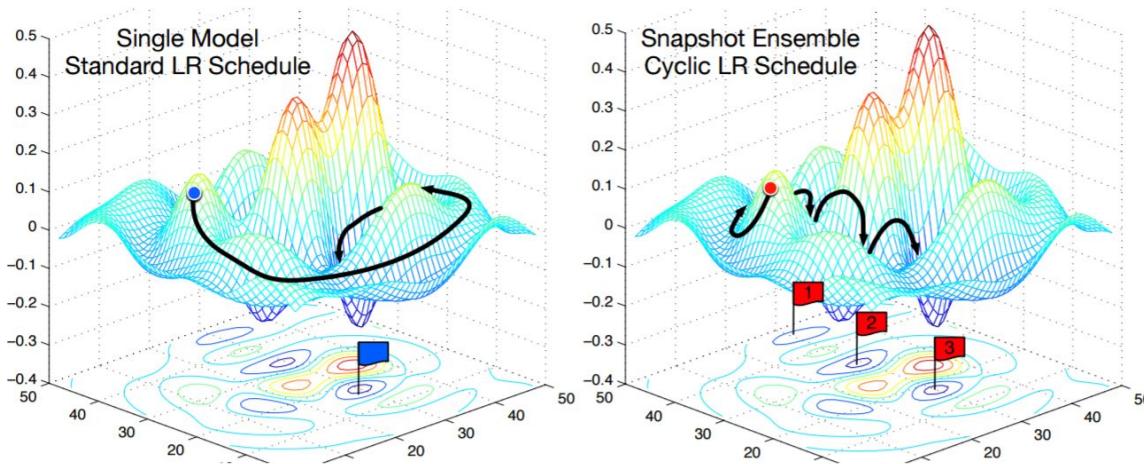
Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016

Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

# Model Ensembles: Tips and Tricks

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

Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Cyclic learning rate schedules can make this work even better!

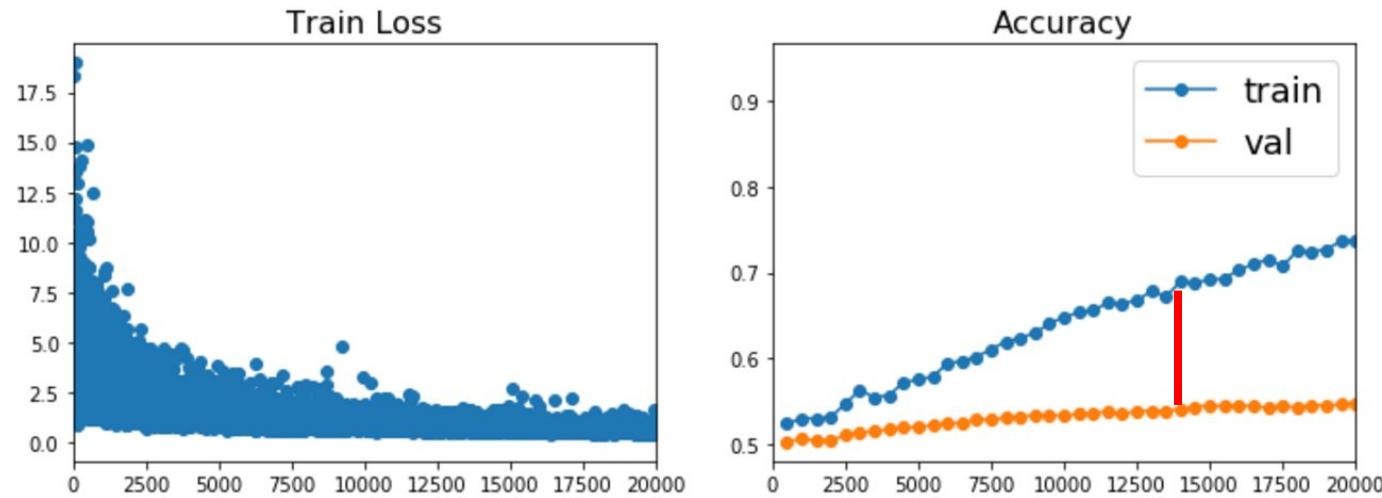
# Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```
while True:  
    data_batch = dataset.sample_data_batch()  
    loss = network.forward(data_batch)  
    dx = network.backward()  
    x += - learning_rate * dx  
    x_test = 0.995*x_test + 0.005*x # use for test set
```

Polyak and Juditsky, "Acceleration of stochastic approximation by averaging", SIAM Journal on Control and Optimization, 1992.

# How to improve single-model performance?



## Regularization

# Regularization: Add term to loss

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

**L2 regularization**

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

L1 regularization

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

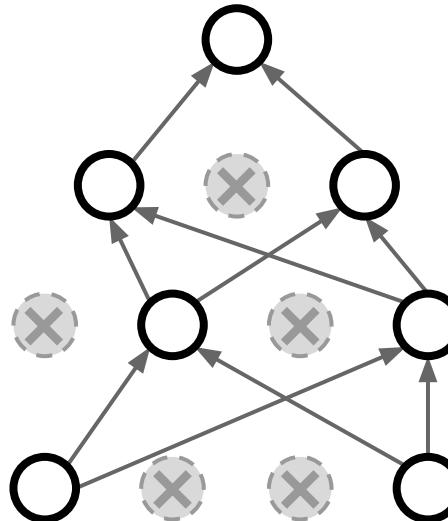
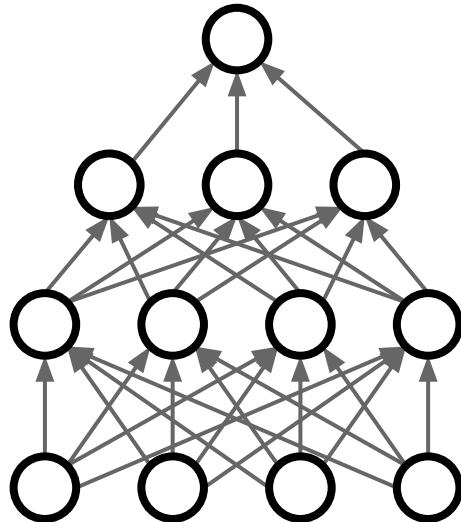
Elastic net (L1 + L2)

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

# Regularization: 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

# Regularization: Dropout

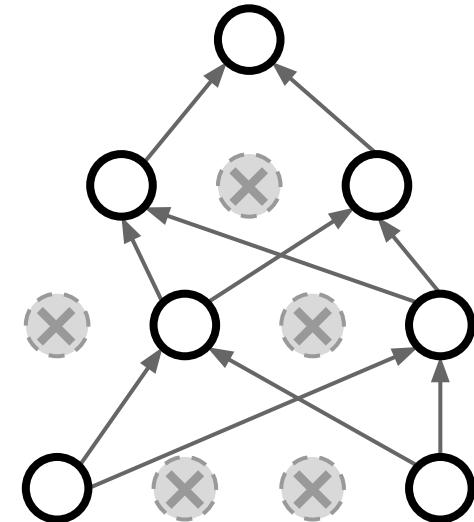
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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

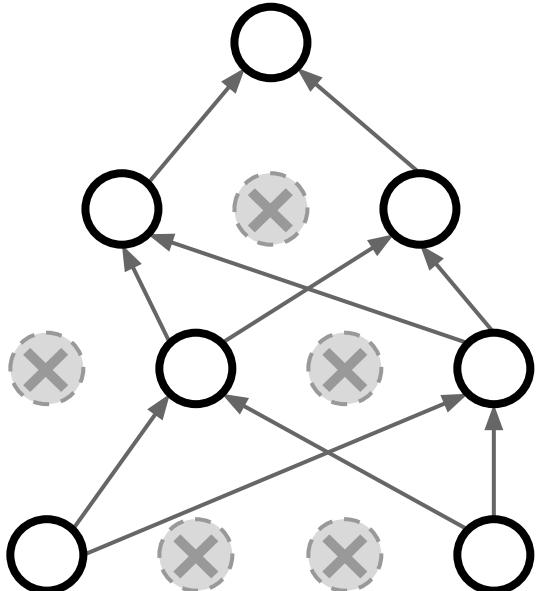
    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout

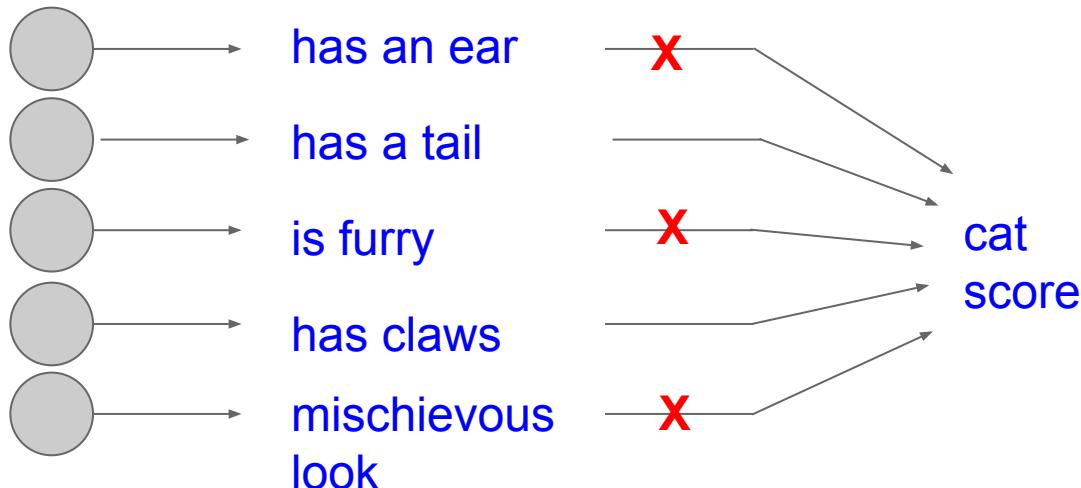


# Regularization: Dropout

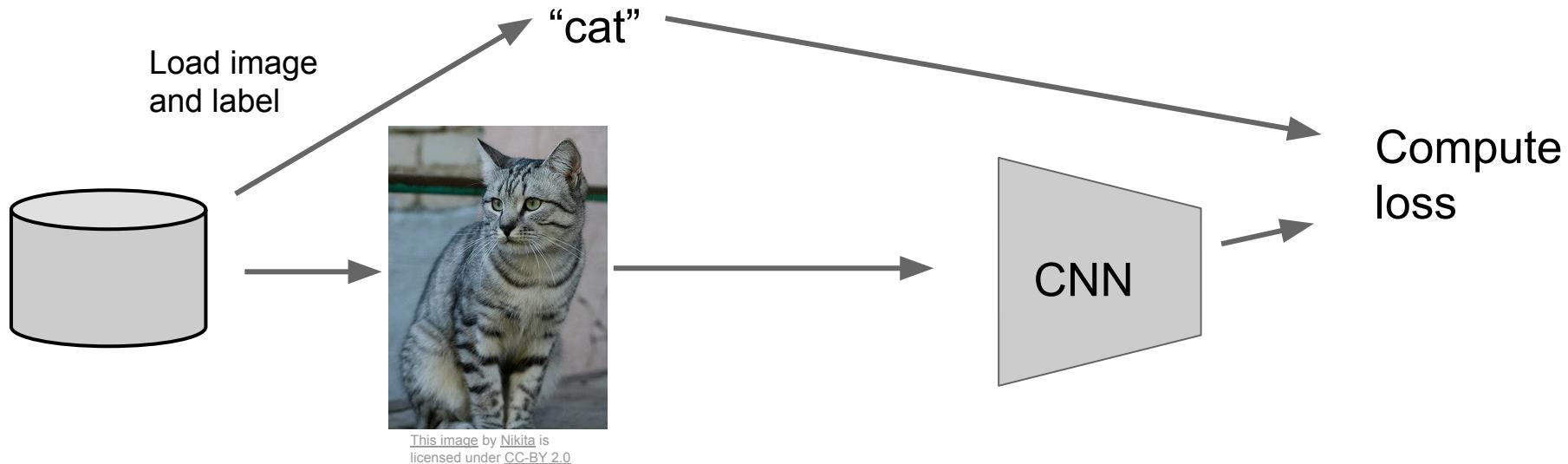
How can this possibly be a good idea?



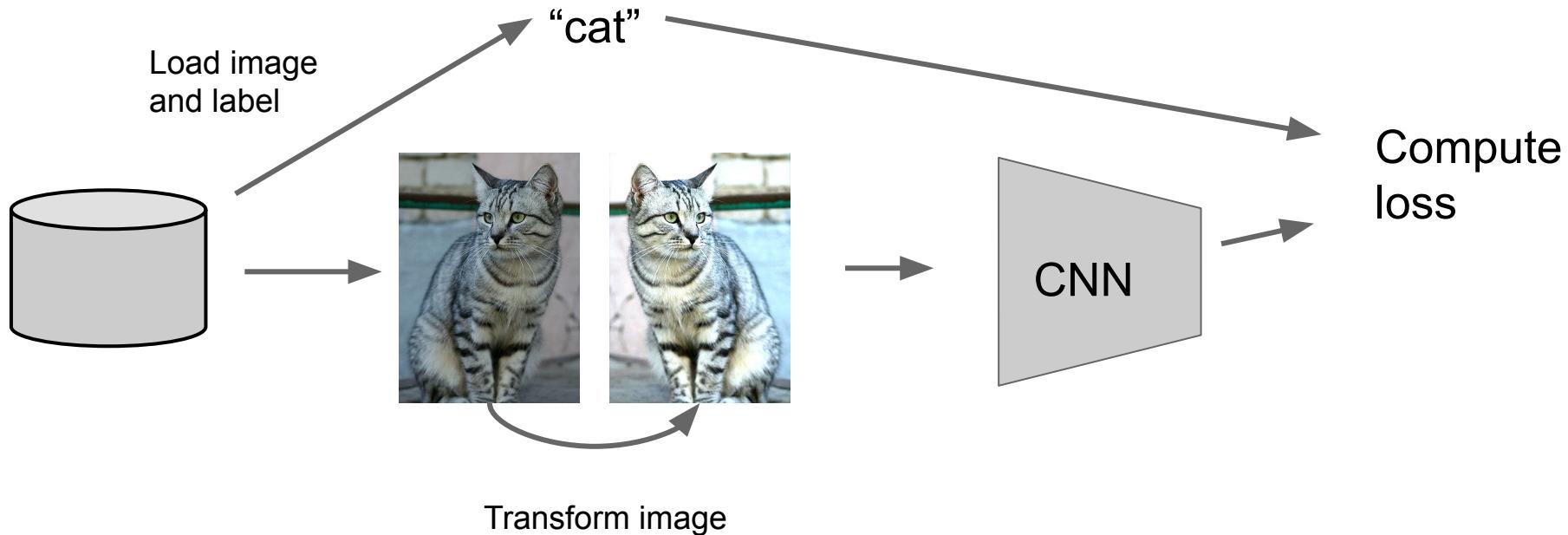
Forces the network to have a redundant representation;  
Prevents co-adaptation of features



# Regularization: Data Augmentation



# Regularization: Data Augmentation



# Data Augmentation

## Horizontal Flips



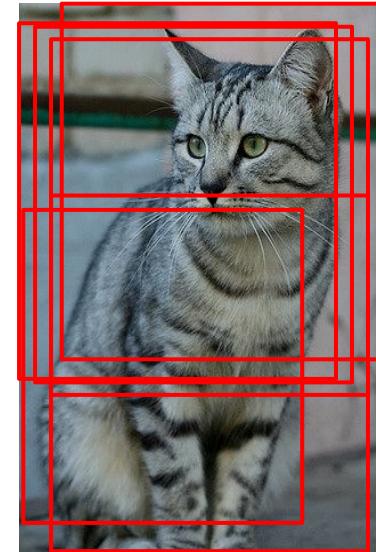
# Data Augmentation

## 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  $224 \times 224$  patch



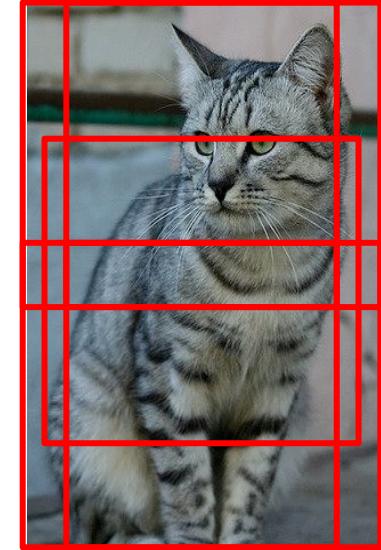
# Data Augmentation

## 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  $224 \times 224$  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  $224 \times 224$  crops: 4 corners + center, + flips

# Data Augmentation

## Color Jitter

Simple: Randomize  
contrast and brightness



# Data Augmentation

## Color Jitter

Simple: Randomize  
contrast and brightness



## More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

# Data Augmentation

Get creative for your problem!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)

# Regularization: A common pattern

**Training:** Add random noise

**Testing:** Marginalize over the noise

## **Examples:**

Dropout

Batch Normalization

Data Augmentation

# Regularization: Cutout

**Training:** Set random image regions to zero

**Testing:** Use full image

## Examples:

Dropout

Batch Normalization

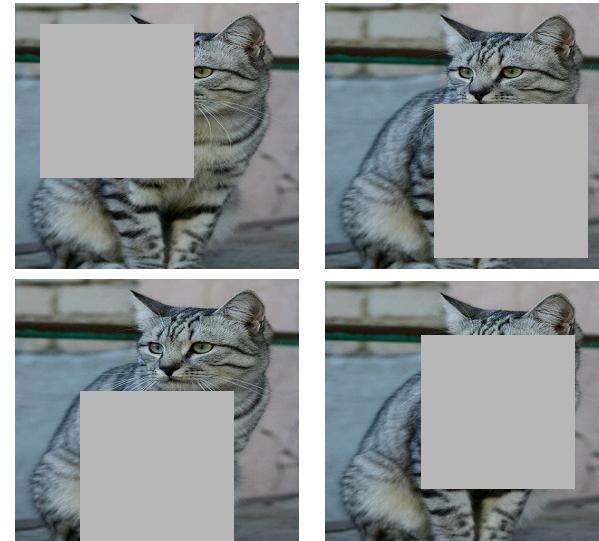
Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout



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

DeVries and Taylor, "Improved Regularization of  
Convolutional Neural Networks with Cutout", arXiv 2017

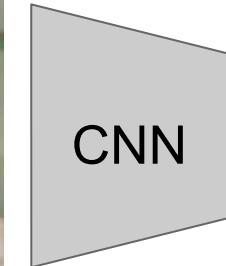
# Regularization: Mixup

**Training:** Train on random blends of images

**Testing:** Use original images

## Examples:

- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- 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

Zhang et al, "mixup: Beyond Empirical Risk Minimization", ICLR 2018

# Choosing Hyperparameters

(without tons of GPUs)

# Choosing Hyperparameters

## Step 1: Check initial loss

Turn off weight decay, sanity check loss at initialization  
e.g.  $\log(C)$  for softmax with  $C$  classes

# Choosing Hyperparameters

**Step 1:** Check initial loss

**Step 2:** Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization

Loss not going down? LR too low, bad initialization

Loss explodes to Inf or NaN? LR too high, bad initialization

# Choosing Hyperparameters

**Step 1:** Check initial loss

**Step 2:** Overfit a small sample

**Step 3:** Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

# Choosing Hyperparameters

**Step 1:** Check initial loss

**Step 2:** Overfit a small sample

**Step 3:** Find LR that makes loss go down

**Step 4:** Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: 1e-4, 1e-5, 0

# Choosing Hyperparameters

**Step 1:** Check initial loss

**Step 2:** Overfit a small sample

**Step 3:** Find LR that makes loss go down

**Step 4:** Coarse grid, train for ~1-5 epochs

**Step 5:** Refine grid, train longer

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

# Choosing Hyperparameters

**Step 1:** Check initial loss

**Step 2:** Overfit a small sample

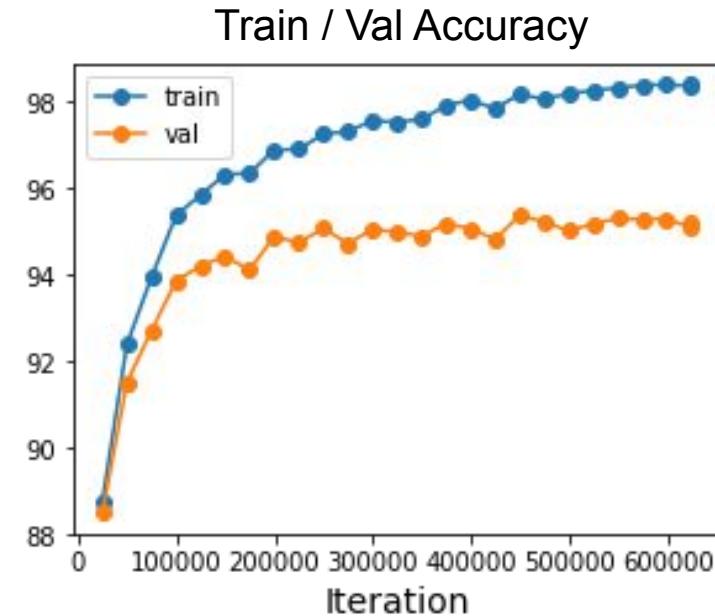
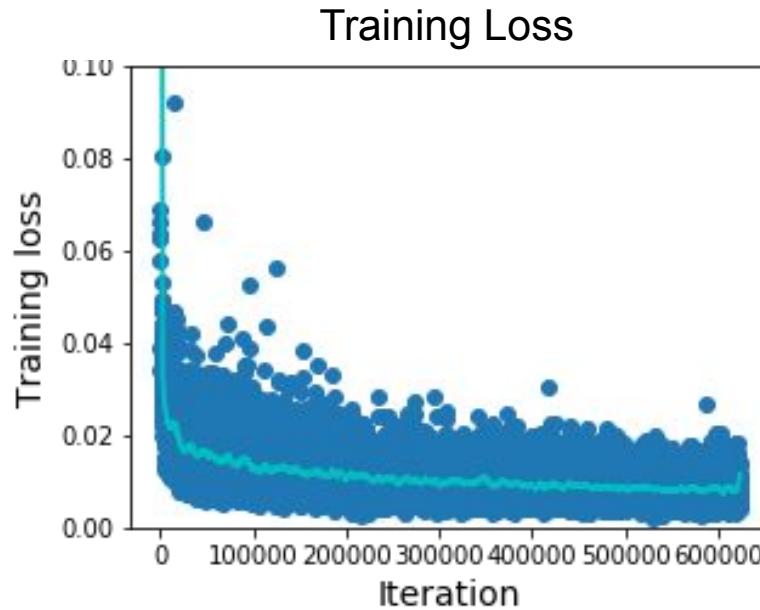
**Step 3:** Find LR that makes loss go down

**Step 4:** Coarse grid, train for ~1-5 epochs

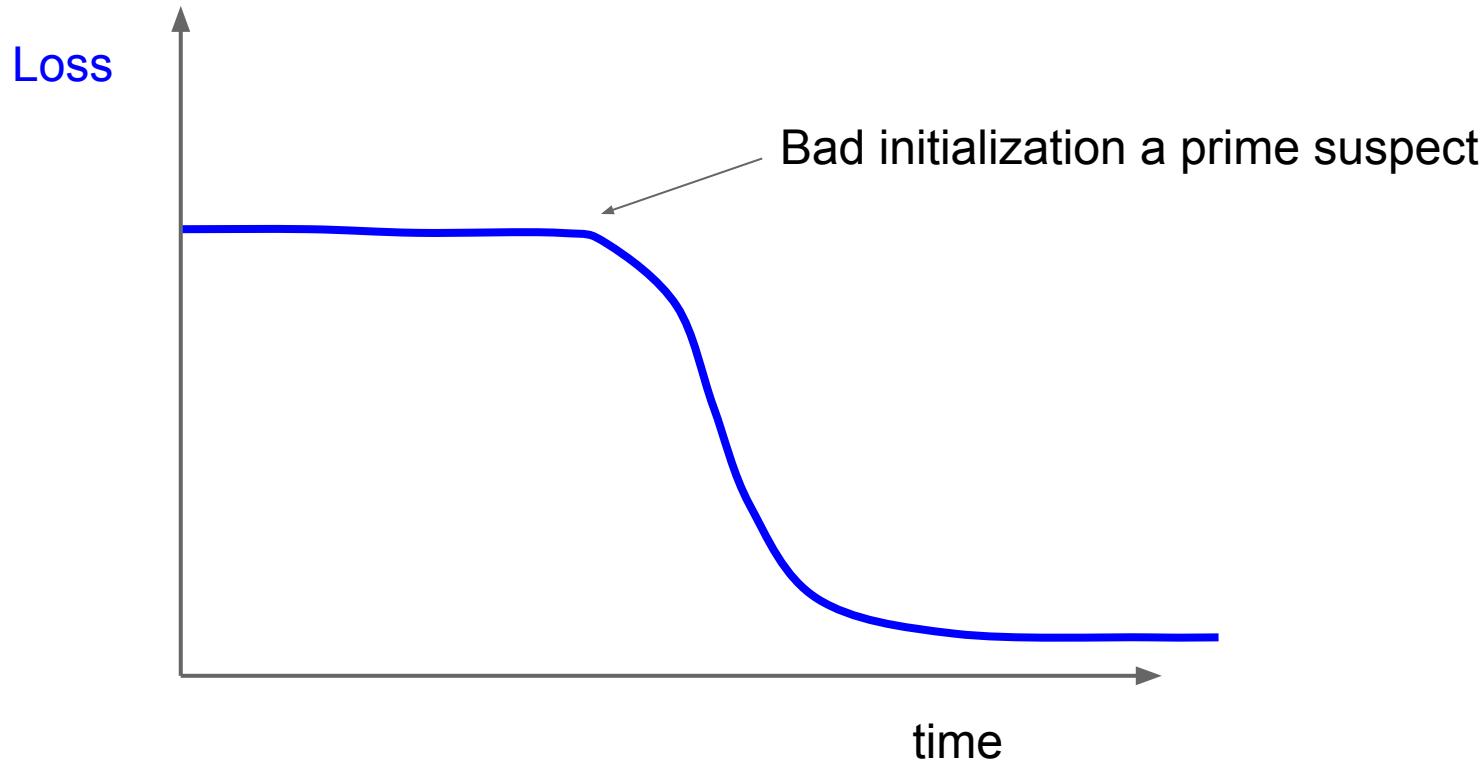
**Step 5:** Refine grid, train longer

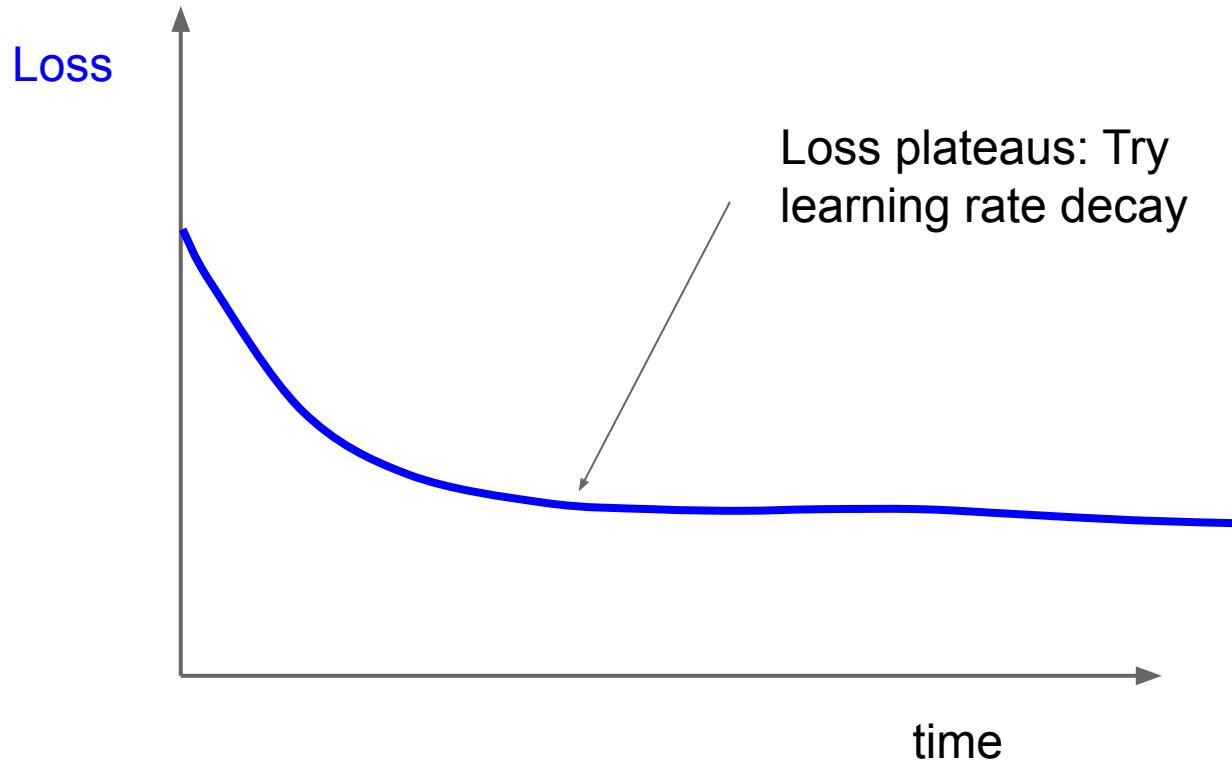
**Step 6:** Look at loss curves

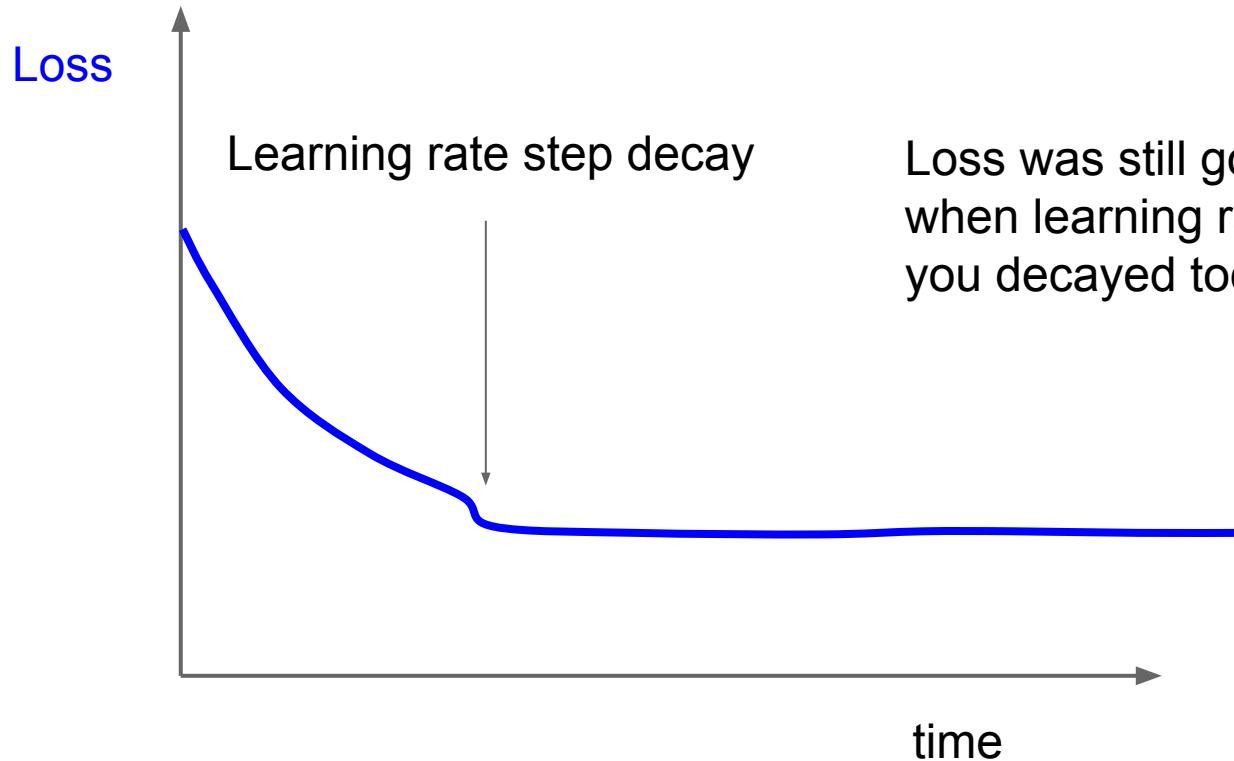
# Look at learning curves!



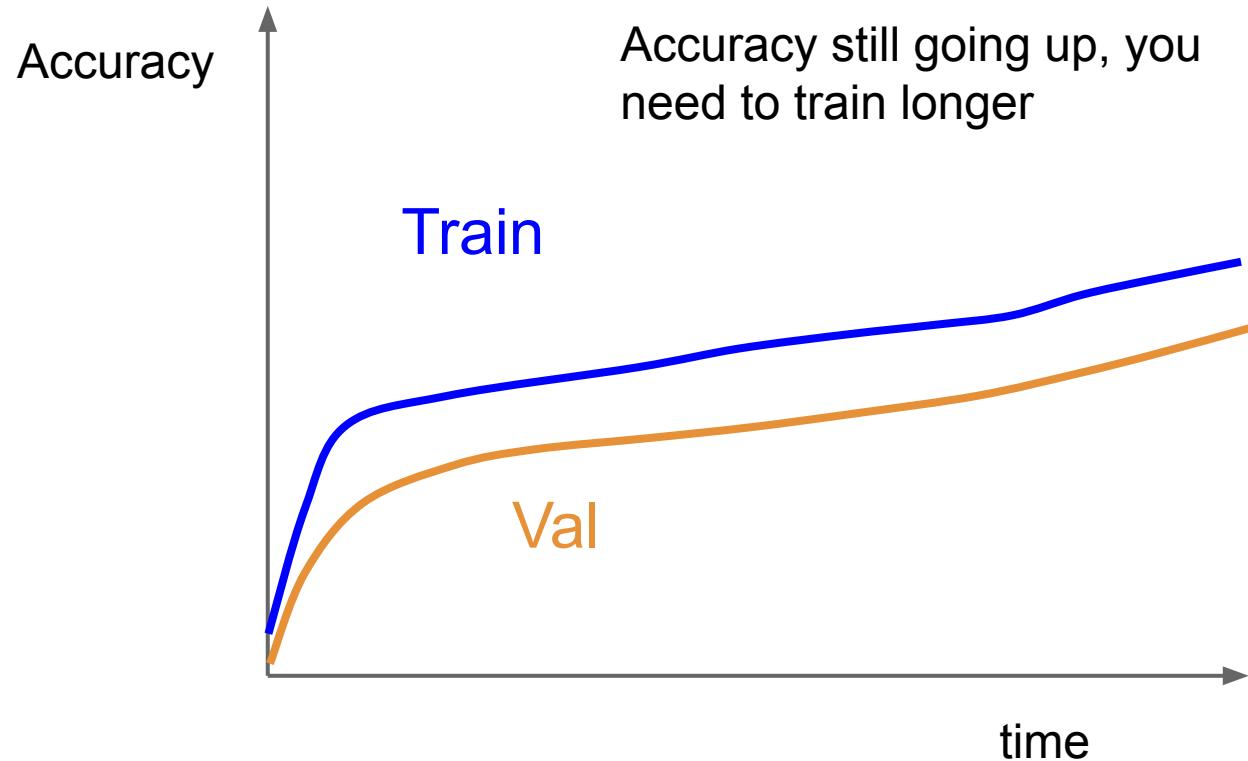
Losses may be noisy, use a scatter plot and also plot moving average to see trends better

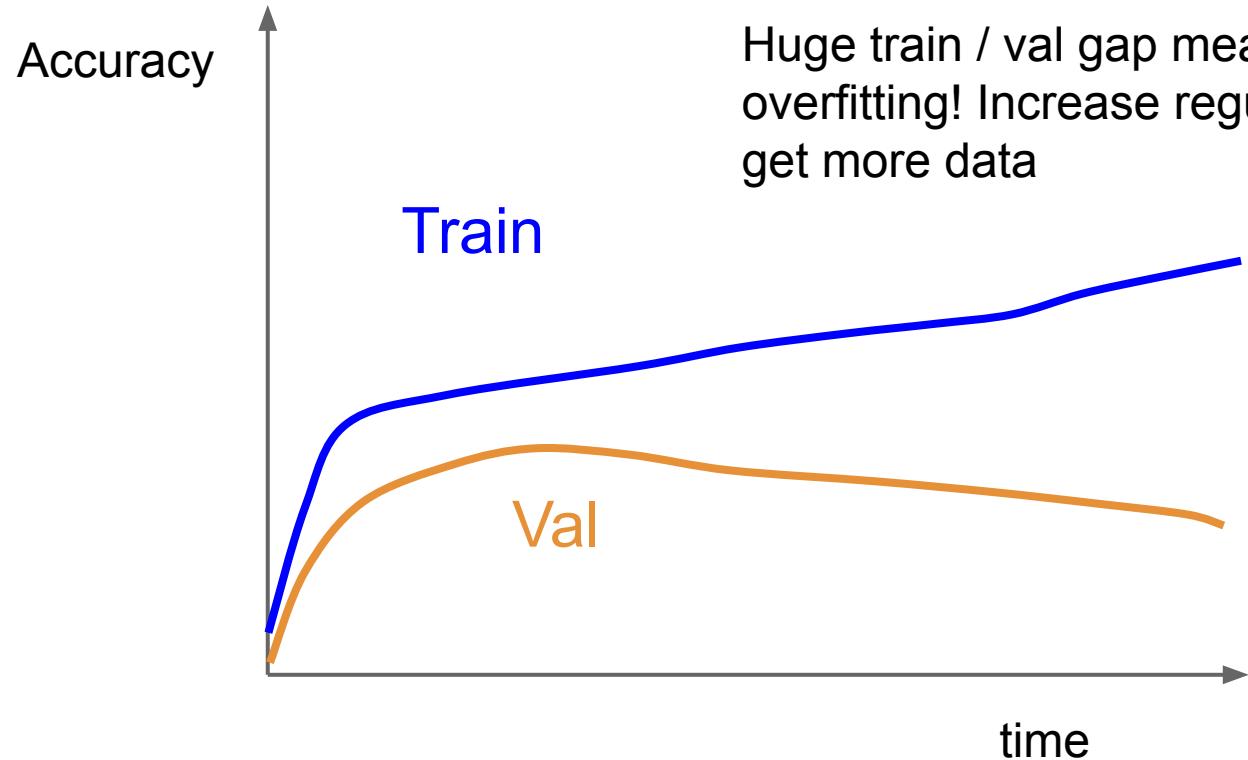




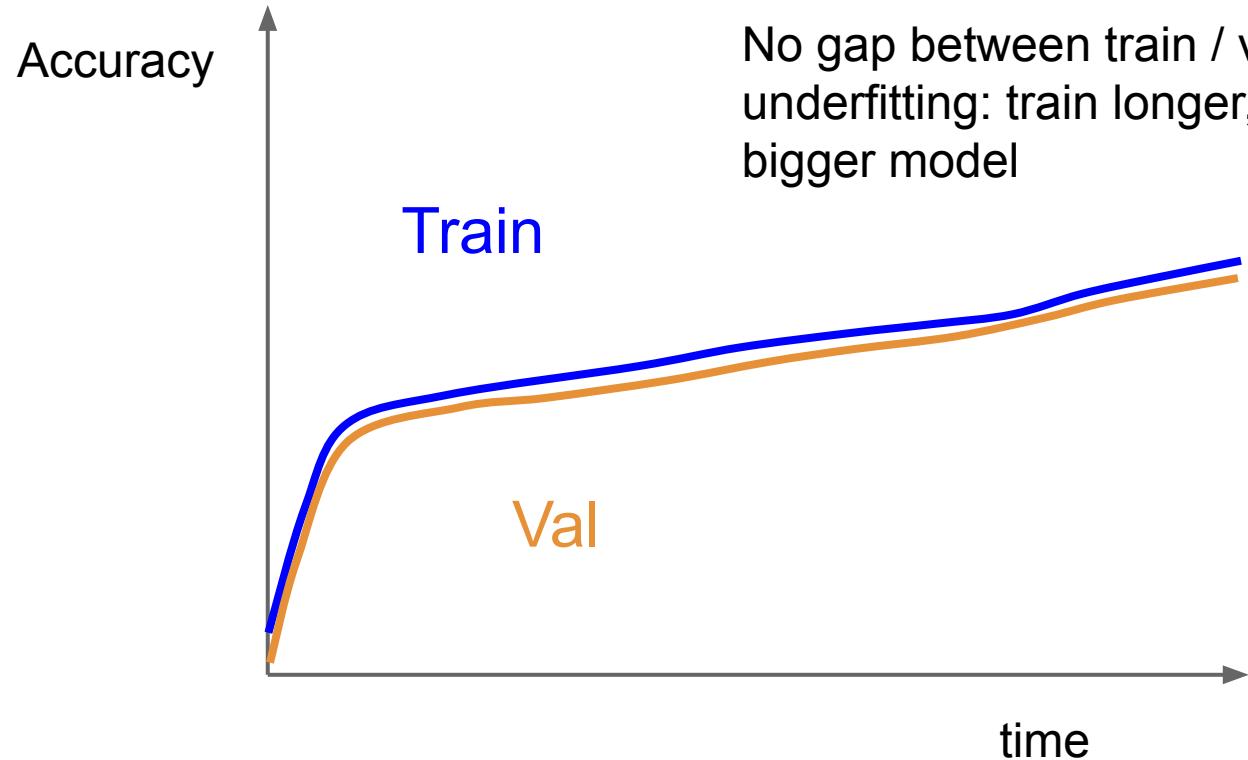


Loss was still going down  
when learning rate dropped,  
you decayed too early!





Huge train / val gap means overfitting! Increase regularization, get more data



No gap between train / val means underfitting: train longer, use a bigger model

# Choosing Hyperparameters

**Step 1:** Check initial loss

**Step 2:** Overfit a small sample

**Step 3:** Find LR that makes loss go down

**Step 4:** Coarse grid, train for ~1-5 epochs

**Step 5:** Refine grid, train longer

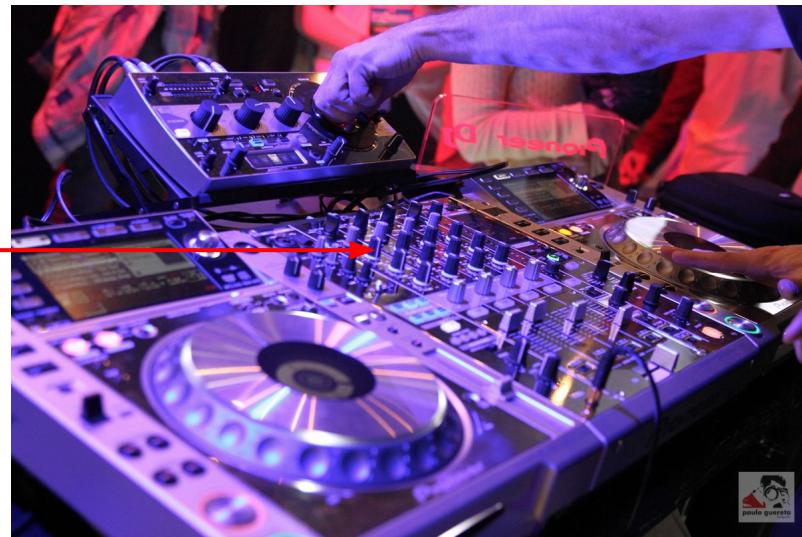
**Step 6:** Look at loss curves

**Step 7:** GOTO step 5

# Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner  
music = loss function



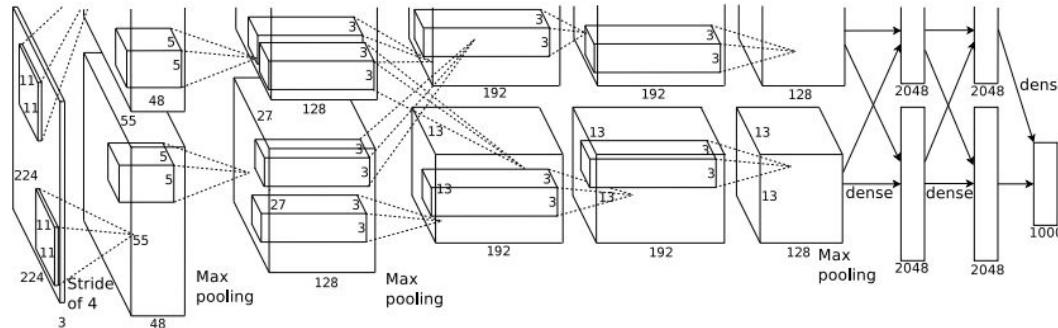
# Summary

- Improve your training error:
  - Optimizers
  - Learning rate schedules
- Improve your test error:
  - Regularization
  - Choosing Hyperparameters

# Lecture 13: Visualizing and Understanding

# What's going on inside ConvNets?

This image is CC0 public domain



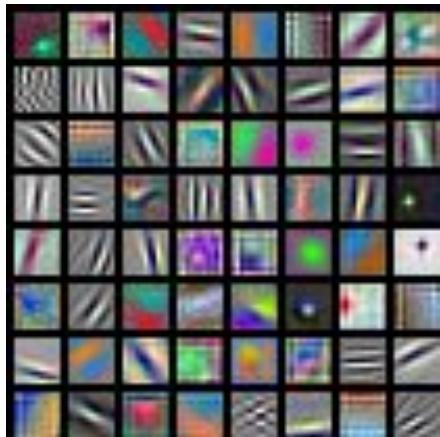
Input Image:  
3 x 224 x 224

What are the intermediate features looking for?

Class Scores:  
1000 numbers

Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012.  
Figure reproduced with permission.

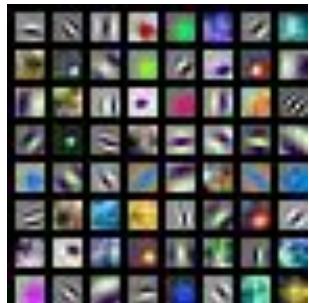
# First Layer: Visualize Filters



AlexNet:  
 $64 \times 3 \times 11 \times 11$



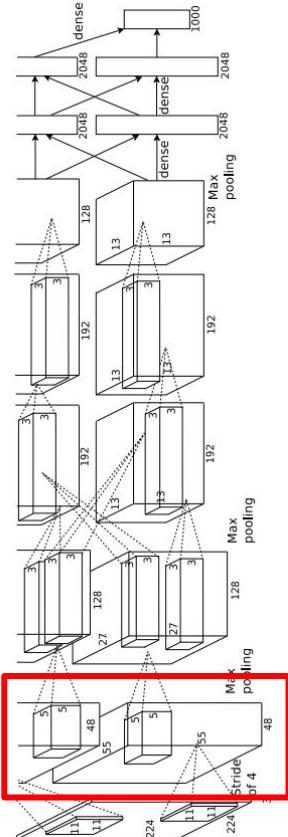
ResNet-18:  
 $64 \times 3 \times 7 \times 7$



ResNet-101:  
 $64 \times 3 \times 7 \times 7$



DenseNet-121:  
 $64 \times 3 \times 7 \times 7$



Krizhevsky, "One weird trick for parallelizing convolutional neural networks", arXiv 2014

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Huang et al, "Densely Connected Convolutional Networks", CVPR 2017

# Visualize the filters/kernels (raw weights)

We can visualize filters at higher layers, but not that interesting

(these are taken from ConvNetJS CIFAR-10 demo)

Weights:  

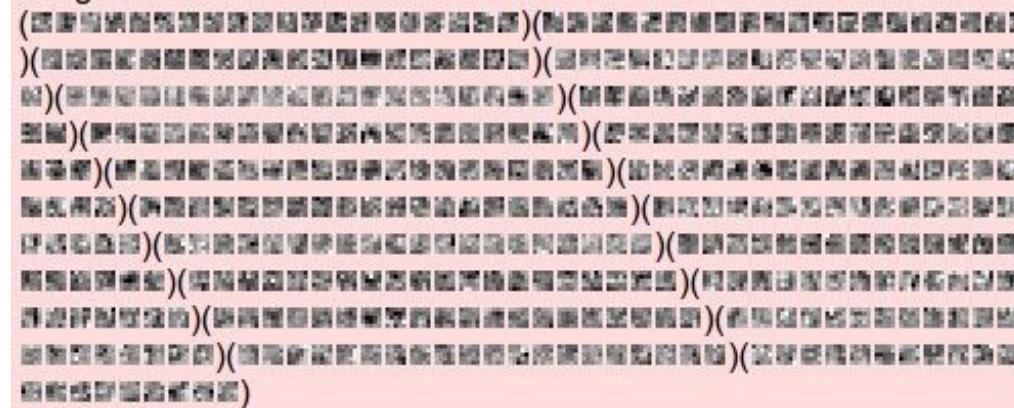

layer 1 weights

$16 \times 3 \times 7 \times 7$

Weights:  


layer 2 weights

$20 \times 16 \times 7 \times 7$

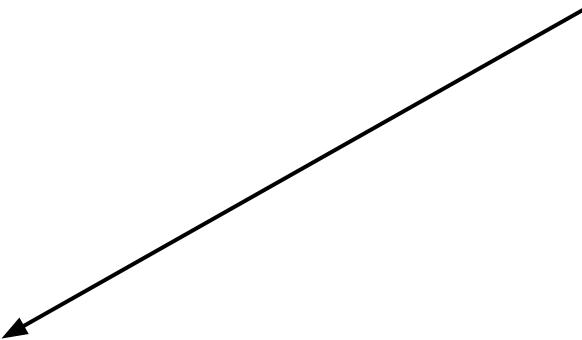
Weights:  


layer 3 weights

$20 \times 20 \times 7 \times 7$

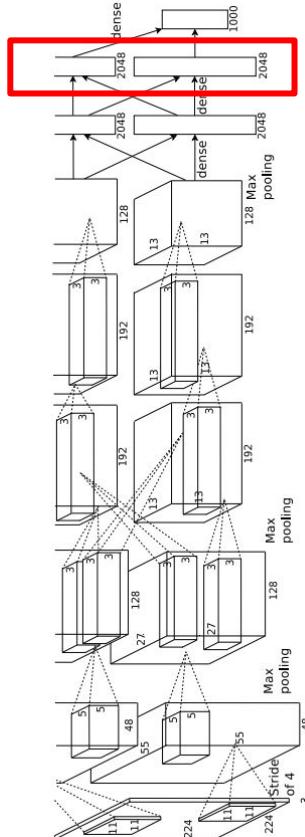
# Last Layer

FC7 layer



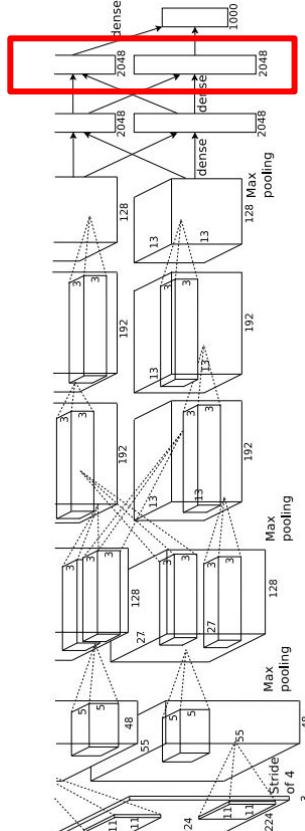
4096-dimensional feature vector for an image  
(layer immediately before the classifier)

Run the network on many images, collect the  
feature vectors



# Last Layer: Nearest Neighbors

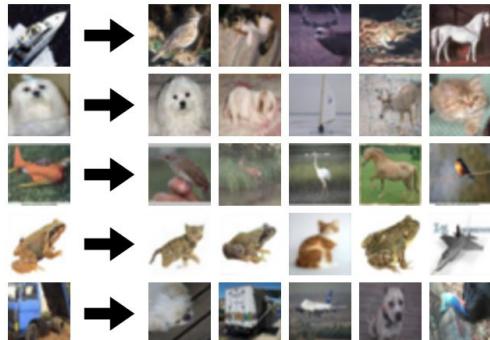
4096-dim vector



Test image    L2 Nearest neighbors in feature space



**Recall:** Nearest neighbors  
in pixel space



Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012.

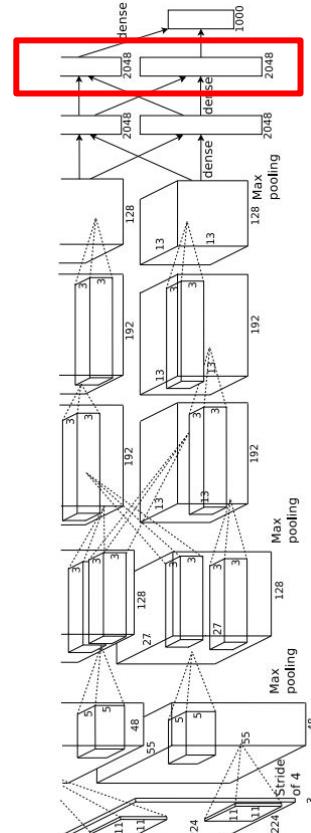
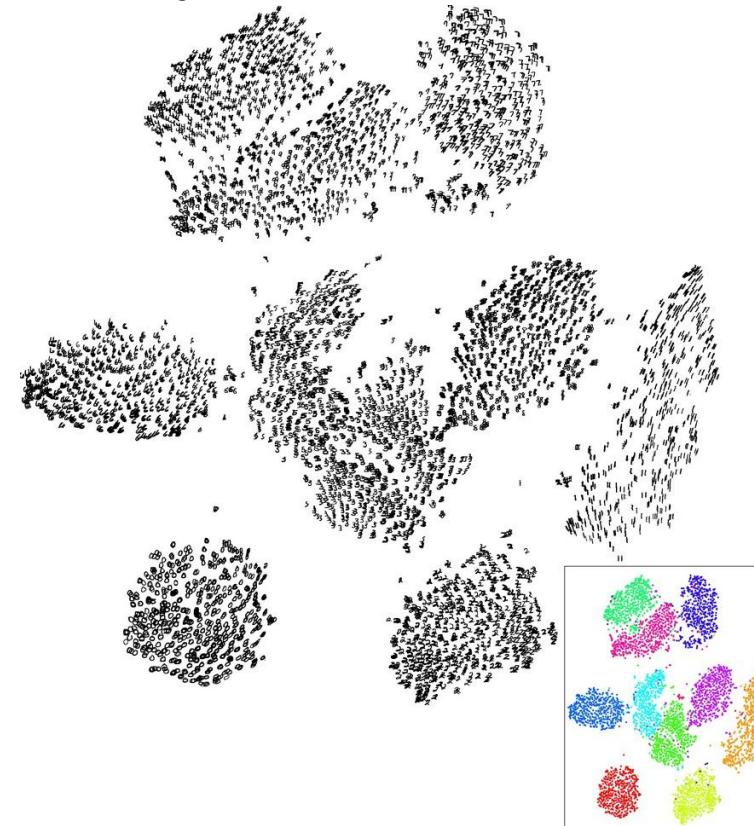
Figures reproduced with permission.

# Last Layer: Dimensionality Reduction

Visualize the “space” of FC7 feature vectors by reducing dimensionality of vectors from 4096 to 2 dimensions

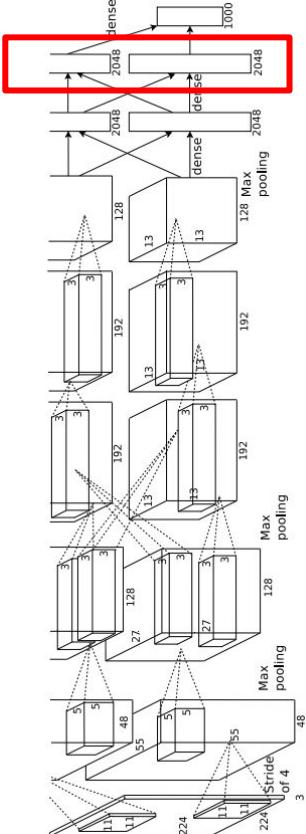
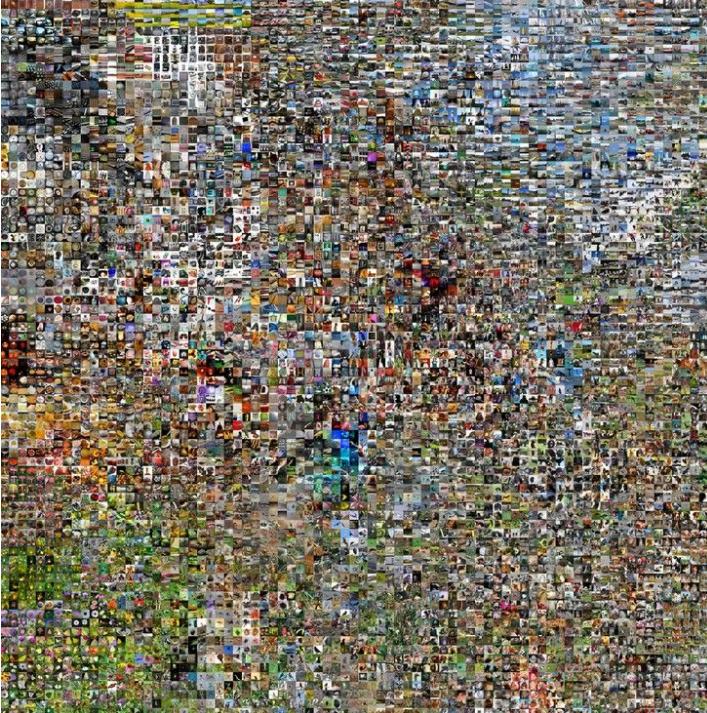
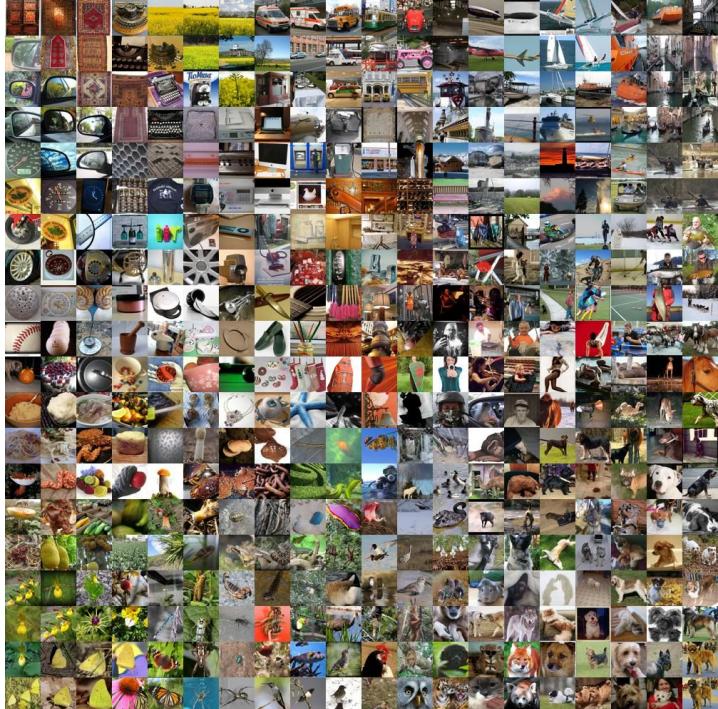
Simple algorithm: Principal Component Analysis (PCA)

More complex: t-SNE



Van der Maaten and Hinton, "Visualizing Data using t-SNE", JMLR 2008  
Figure copyright Laurens van der Maaten and Geoff Hinton, 2008. Reproduced with permission.

# Last Layer: Dimensionality Reduction

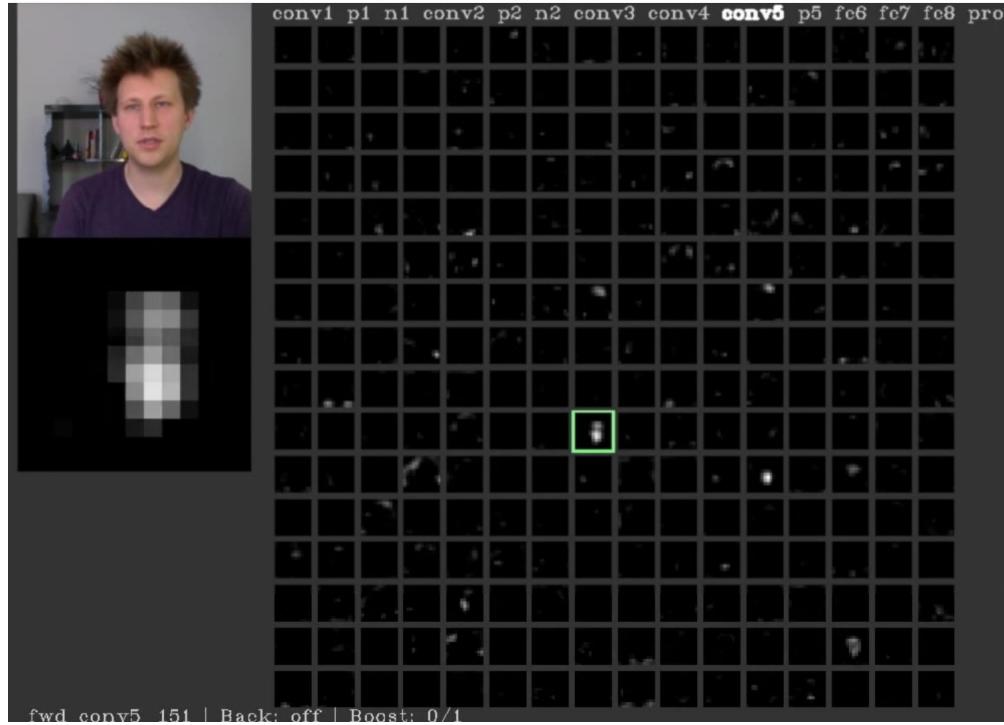


Van der Maaten and Hinton, "Visualizing Data using t-SNE", JMLR 2008  
Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012.  
Figure reproduced with permission.

See high-resolution versions at  
<http://cs.stanford.edu/people/karpathy/cnnembed/>

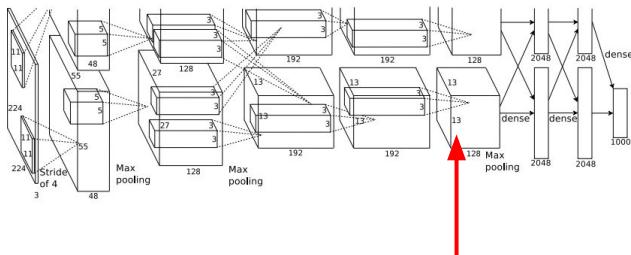
# Visualizing Activations

conv5 feature map is  
128x13x13; visualize  
as 128 13x13  
grayscale images



Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.  
Figure copyright Jason Yosinski, 2014. Reproduced with permission.

# Maximally Activating Patches



Pick a layer and a channel; e.g. conv5 is  
128 x 13 x 13, pick channel 17/128

Run many images through the network,  
record values of chosen channel

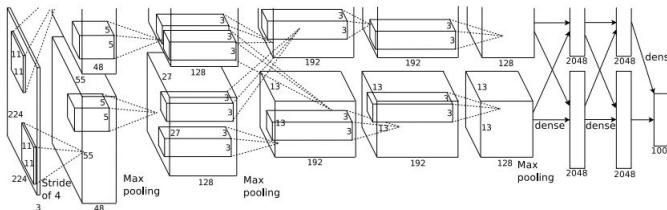
Visualize image patches that correspond  
to maximal activations



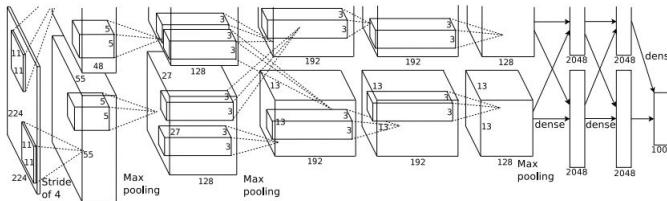
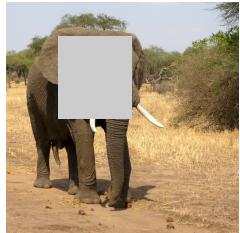
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015  
Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015;  
reproduced with permission.

# Which pixels matter: Saliency via Occlusion

Mask part of the image before feeding to CNN,  
check how much predicted probabilities change



$$P(\text{elephant}) = 0.95$$



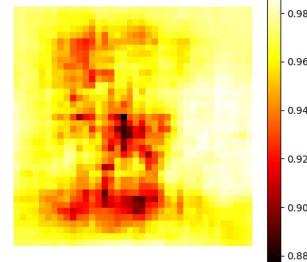
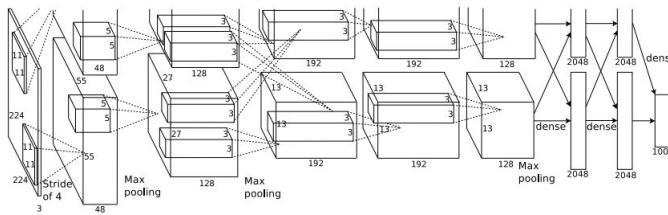
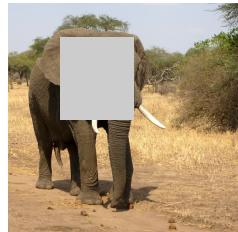
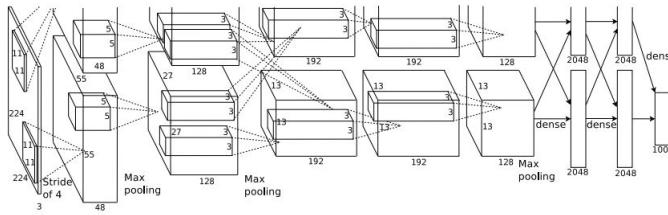
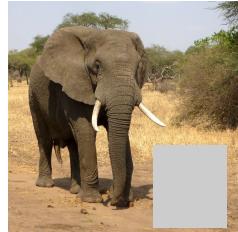
$$P(\text{elephant}) = 0.75$$

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014

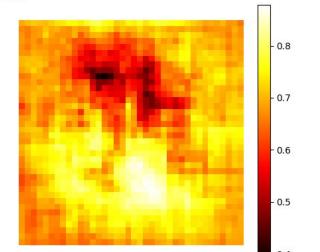
Boat image is CC0 public domain  
Elephant image is CC0 public domain  
Go-Karts image is CC0 public domain

# Which pixels matter: Saliency via Occlusion

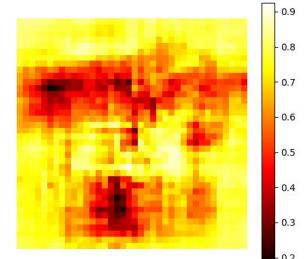
Mask part of the image before feeding to CNN,  
check how much predicted probabilities change



schooner



African elephant, *Loxodonta africana*



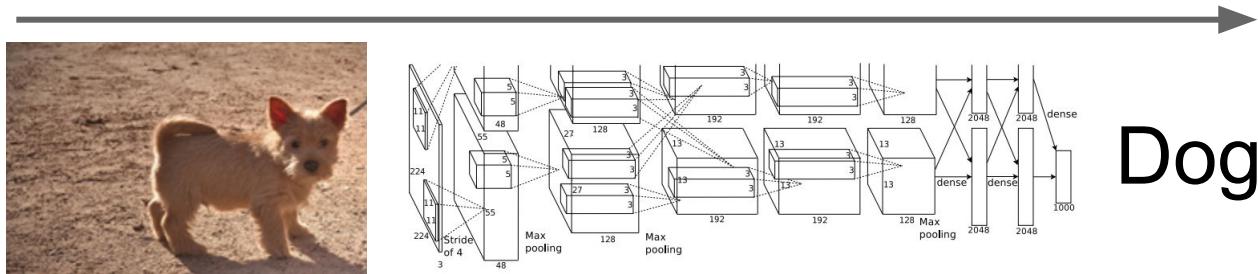
go-kart

Boat image is CC0 public domain  
Elephant image is CC0 public domain  
Go-Karts image is CC0 public domain

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014

# Which pixels matter: Saliency via Backprop

Forward pass: Compute probabilities

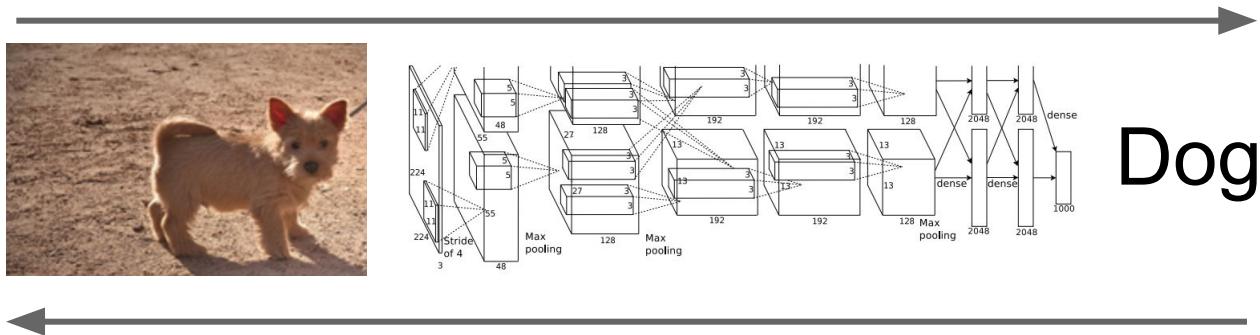


Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

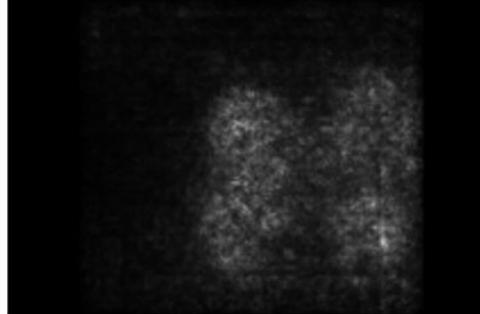
Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

# Which pixels matter: Saliency via Backprop

Forward pass: Compute probabilities



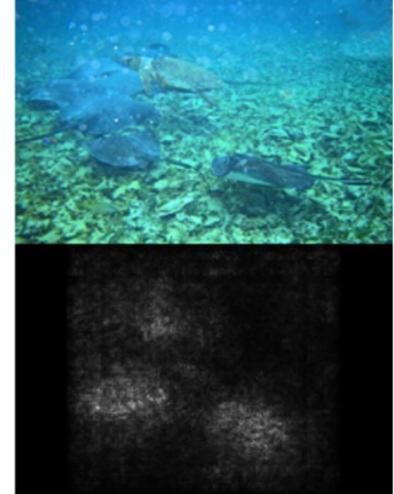
Compute gradient of (unnormalized) class score with respect to image pixels, take absolute value and max over RGB channels



Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

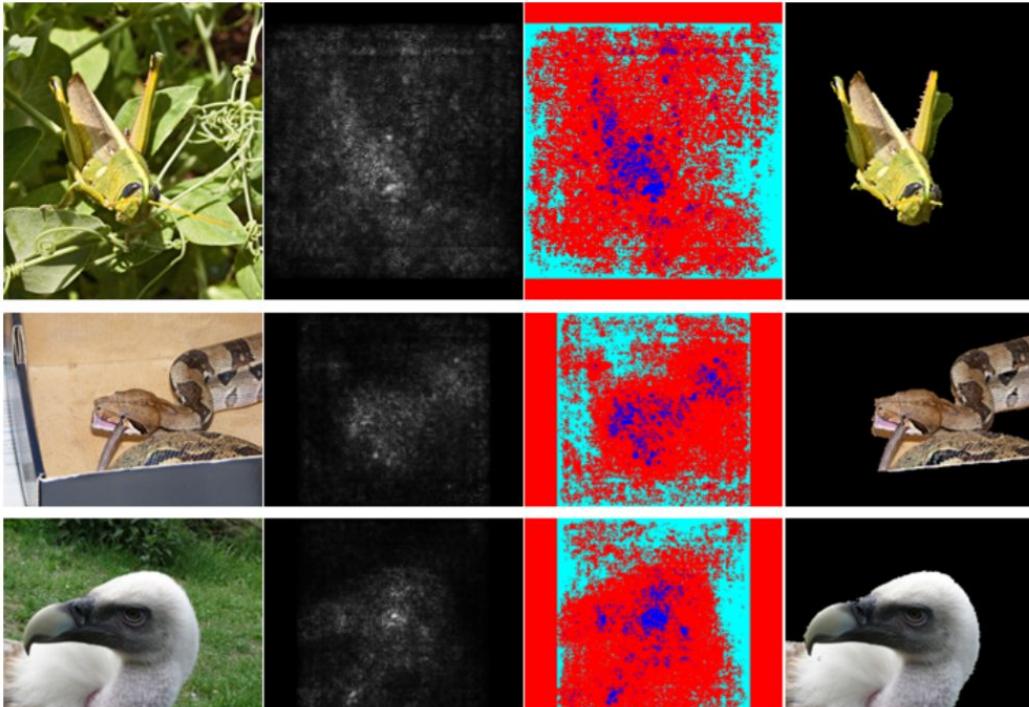
# Saliency Maps



Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

# Saliency Maps: Segmentation without supervision



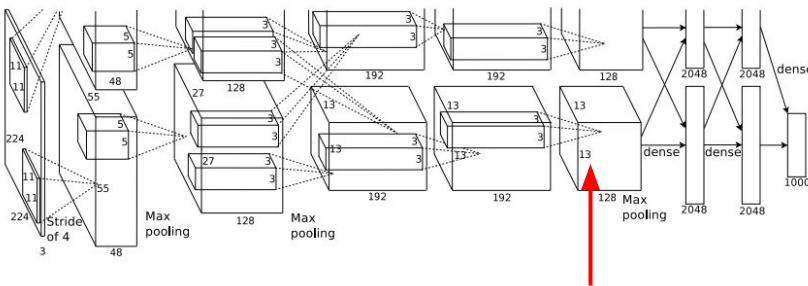
Use GrabCut on  
saliency map

Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

Rother et al, "Grabcut: Interactive foreground extraction using iterated graph cuts", ACM TOG 2004

# Intermediate Features via (guided) backprop

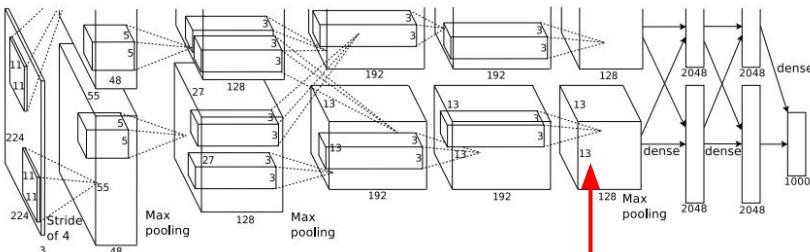


Pick a single intermediate neuron, e.g. one value in  $128 \times 13 \times 13$  conv5 feature map

Compute gradient of neuron value with respect to image pixels

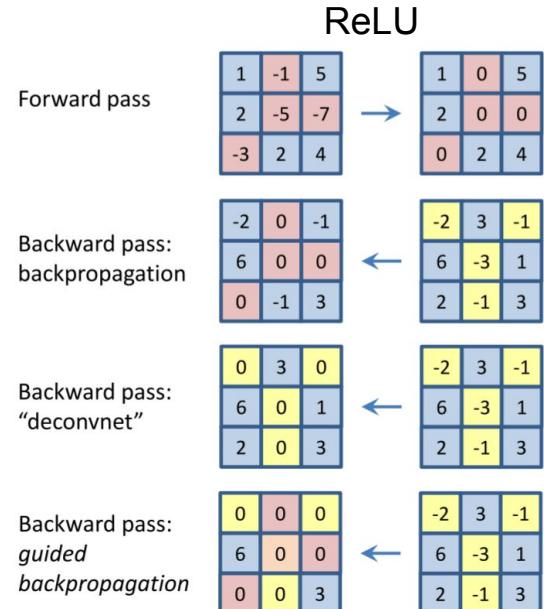
Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014  
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015

# Intermediate Features via (guided) backprop



Pick a single intermediate neuron, e.g. one value in  $128 \times 13 \times 13$  conv5 feature map

Compute gradient of neuron value with respect to image pixels



Images come out nicer if you only backprop positive gradients through each ReLU (guided backprop)

Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015; reproduced with permission.

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014  
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015

# Intermediate features via (guided) backprop



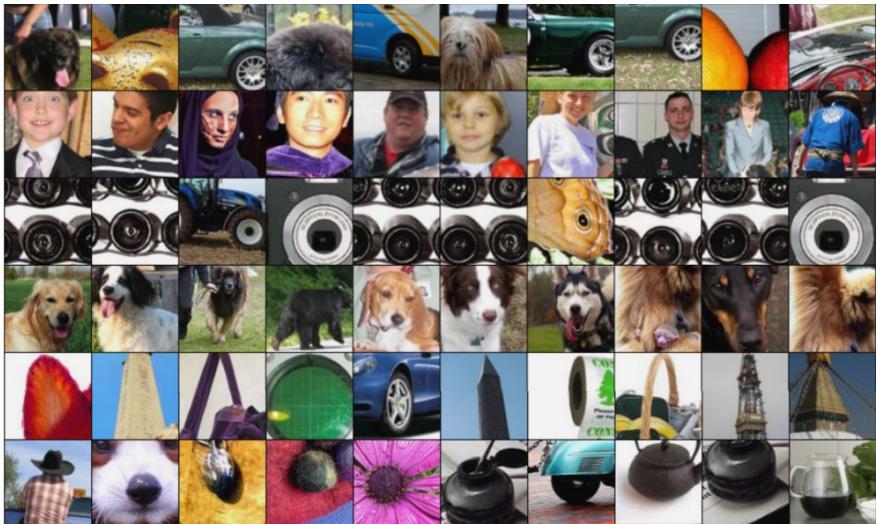
Maximally activating patches  
(Each row is a different neuron)



Guided Backprop

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014  
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015  
Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015; reproduced with permission.

# Intermediate features via (guided) backprop



Maximally activating patches  
(Each row is a different neuron)



Guided Backprop

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014  
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015  
Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015; reproduced with permission.

# Visualizing CNN features: Gradient Ascent

**(Guided) backprop:**

Find the part of an image that a neuron responds to

**Gradient ascent:**

Generate a synthetic image that maximally activates a neuron

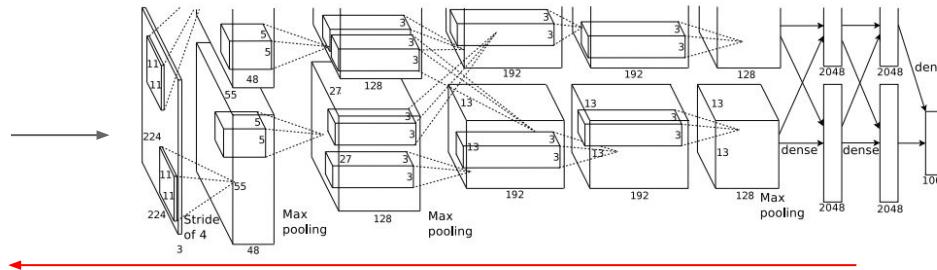
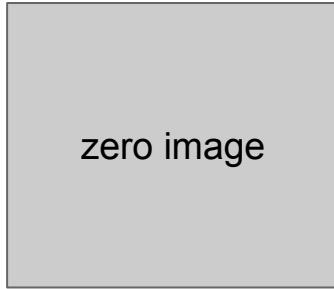
$$I^* = \arg \max_I f(I) + R(I)$$

Neuron value

Natural image regularizer

# Visualizing CNN features: Gradient Ascent

1. Initialize image to zeros



Repeat:

2. Forward image to compute current scores
3. Backprop to get gradient of neuron value with respect to image pixels
4. Make a small update to the image

$$\arg \max_I S_c(I) - \lambda \|I\|_2^2$$

score for class c (before Softmax)

# Visualizing CNN features: Gradient Ascent

Adding “multi-faceted” visualization gives even nicer results:  
(Plus more careful regularization, center-bias)

Reconstructions of multiple feature types (facets) recognized by the same “grocery store” neuron



Corresponding example training set images recognized by the same neuron as in the “grocery store” class



Nguyen et al., “Multifaceted Feature Visualization: Uncovering the Different Types of Features Learned By Each Neuron in Deep Neural Networks”, ICML Visualization for Deep Learning Workshop 2016.  
Figures copyright Anh Nguyen, Jason Yosinski, and Jeff Clune, 2016; reproduced with permission.

# Visualizing CNN features: Gradient Ascent



Nguyen et al., "Multifaceted Feature Visualization: Uncovering the Different Types of Features Learned By Each Neuron in Deep Neural Networks", ICML Visualization for Deep Learning Workshop 2016.

Figures copyright Anh Nguyen, Jason Yosinski, and Jeff Clune, 2016; reproduced with permission.

# Visualizing CNN features: Gradient Ascent

Optimize in FC6 latent space instead of pixel space:



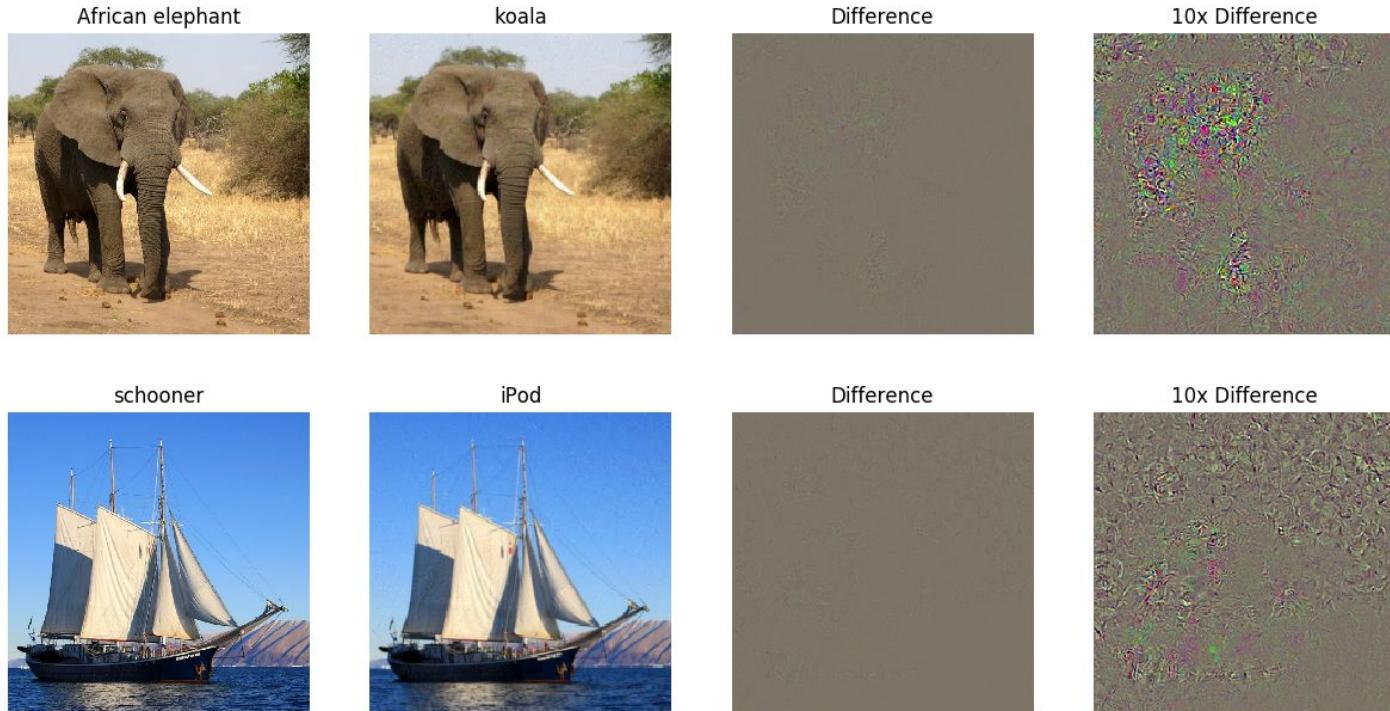
Nguyen et al., "Synthesizing the preferred inputs for neurons in neural networks via deep generator networks," NIPS 2016

Figure copyright Nguyen et al, 2016; reproduced with permission.

# Fooling Images / Adversarial Examples

- (1) Start from an arbitrary image
- (2) Pick an arbitrary class
- (3) Modify the image to maximize the class
- (4) Repeat until network is fooled

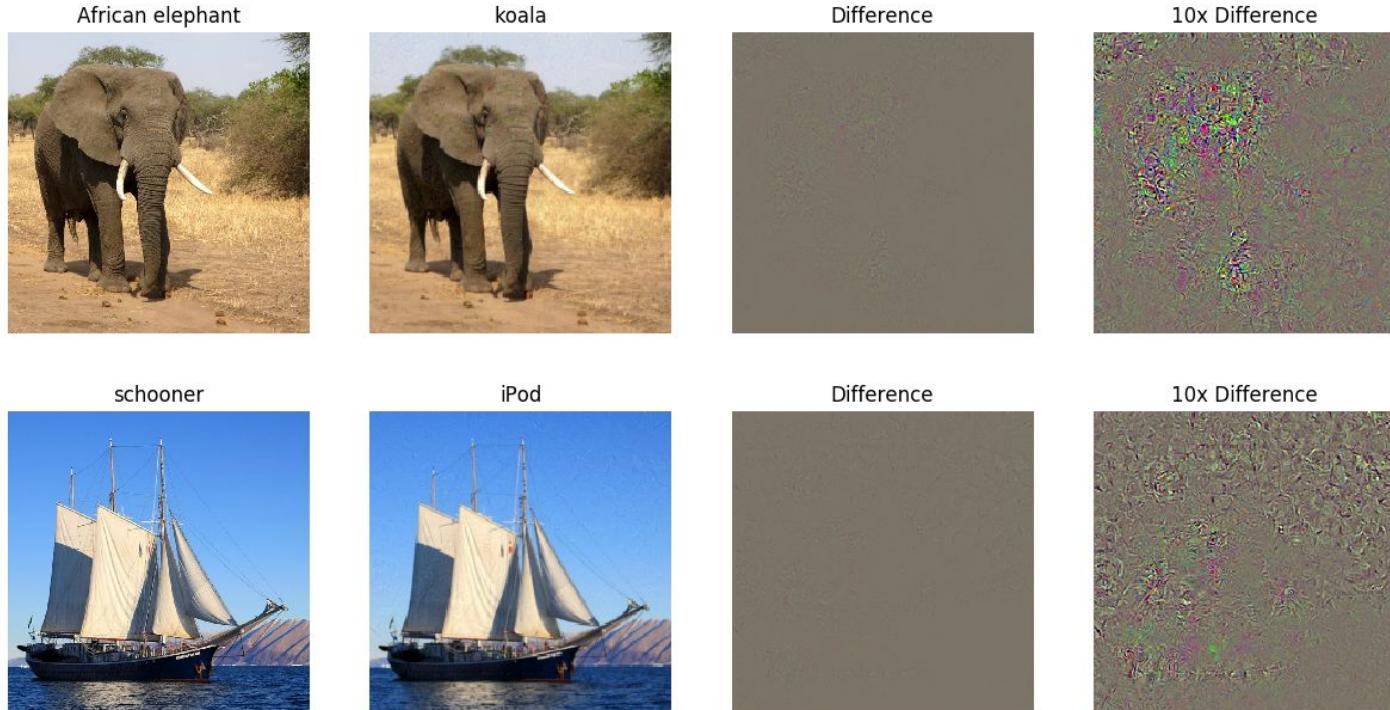
# Fooling Images / Adversarial Examples



Boat image is CC0 public domain

Elephant image is CC0 public domain

# Fooling Images / Adversarial Examples



Check out [Ian Goodfellow's lecture](#) from 2017

Boat image is CC0 public domain  
Elephant image is CC0 public domain

# Neural Style Transfer

Content Image



[This image is licensed under CC-BY 3.0](#)

+

Style Image



[Starry Night](#) by Van Gogh is in the public domain

Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015

# Neural Style Transfer

Content Image



[This image](#) is licensed under CC-BY 3.0

+

Style Image



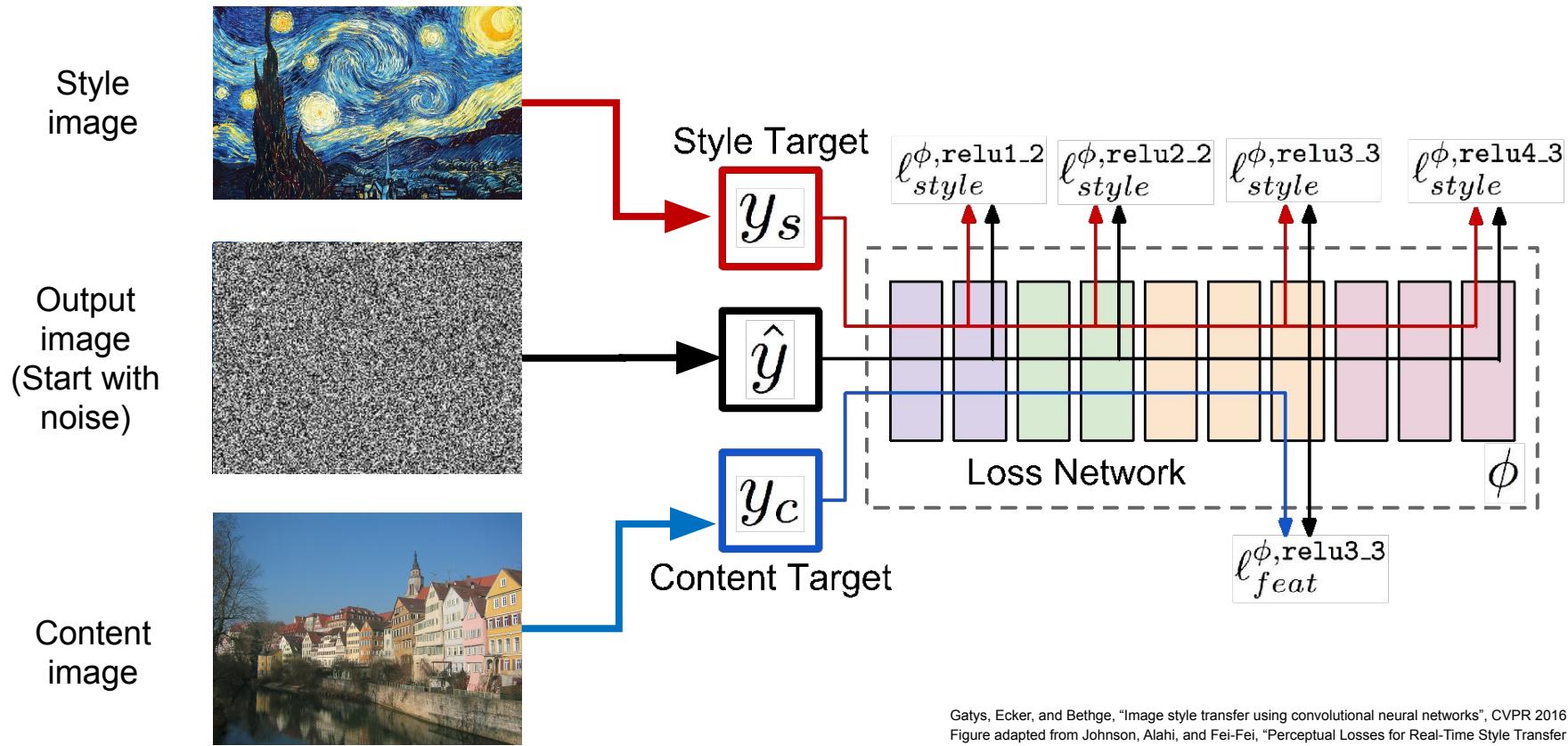
[Starry Night](#) by Van Gogh is in the public domain

=

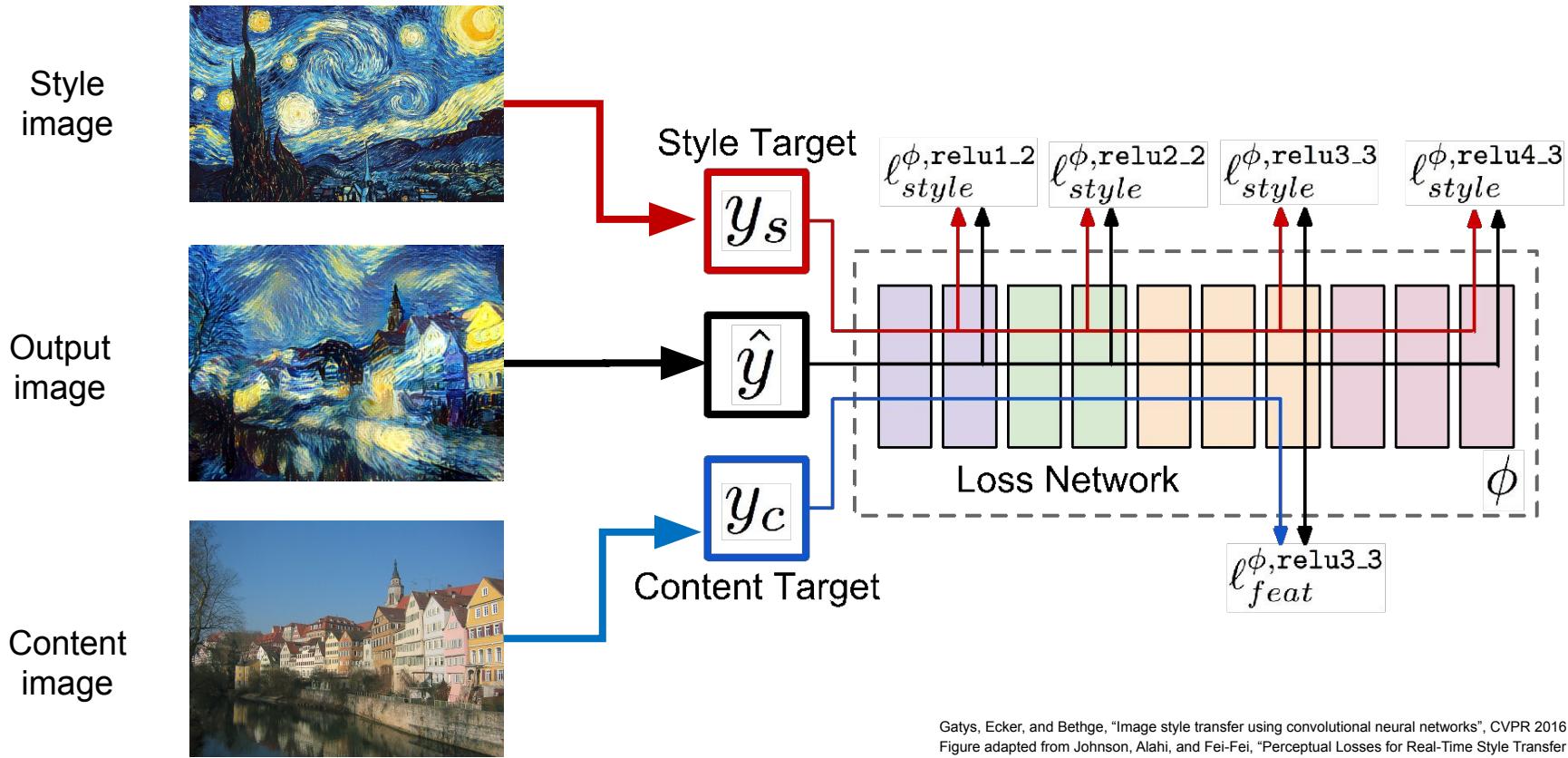


[This image](#) copyright Justin Johnson, 2015. Reproduced with permission.

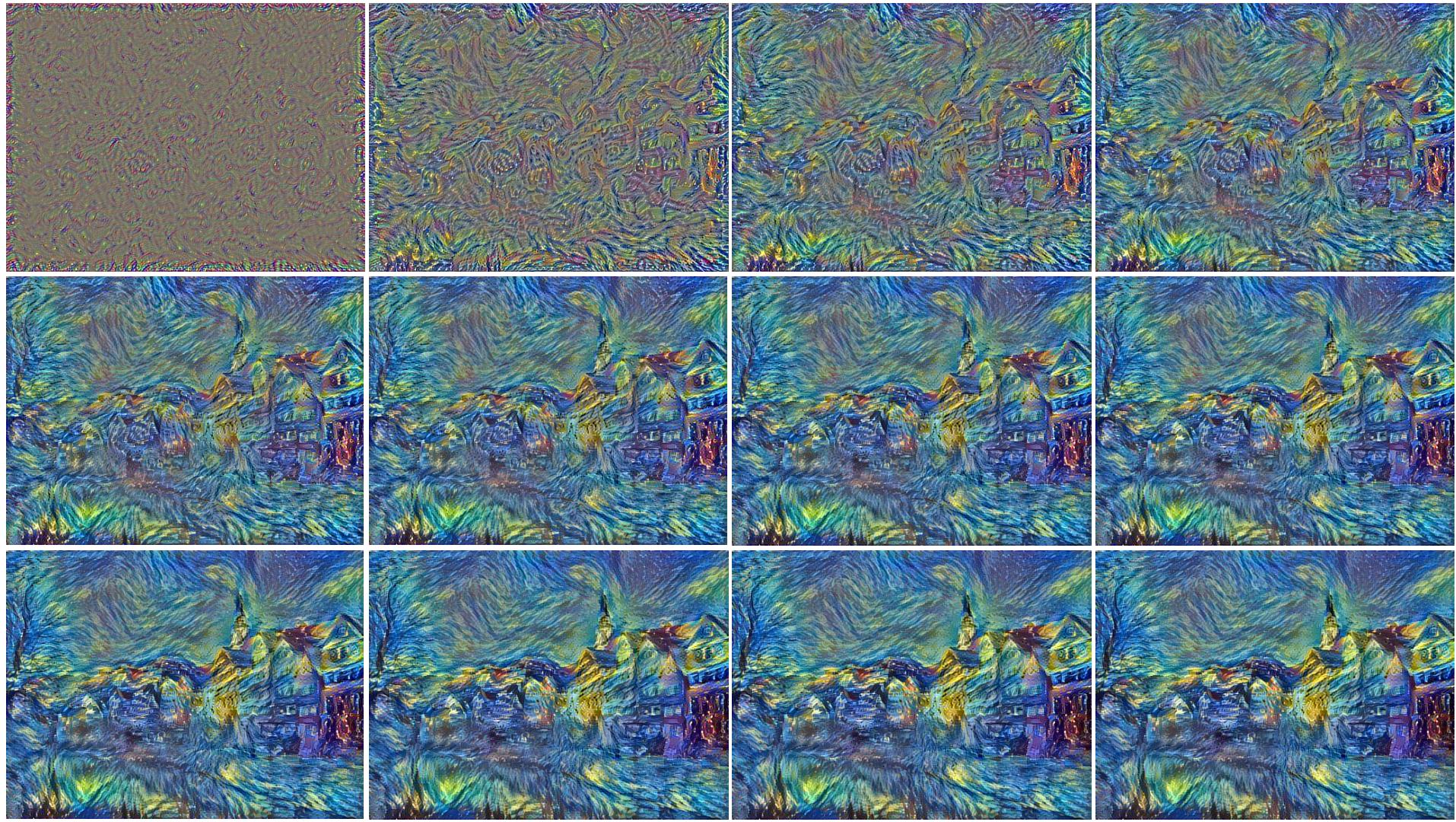
Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016  
 Figure adapted from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016  
 Figure adapted from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.



# Neural Style Transfer

Example outputs from  
[my implementation](#)  
(in Torch)



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016  
Figure copyright Justin Johnson, 2015.

# Neural Style Transfer



More weight to  
content loss



More weight to  
style loss

# Neural Style Transfer

Resizing style image before running style transfer algorithm can transfer different types of features



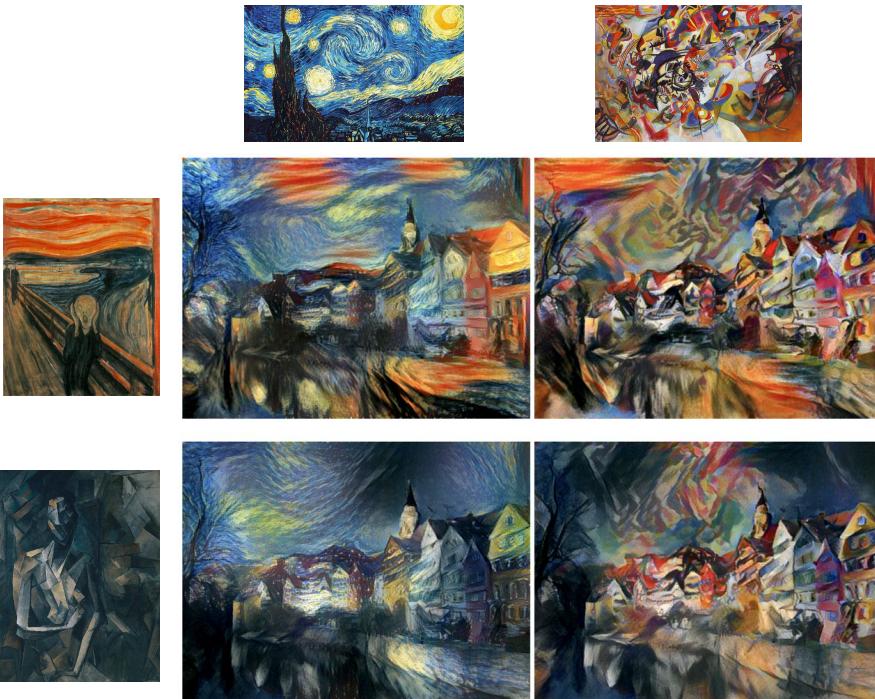
Larger style  
image

Smaller style  
image

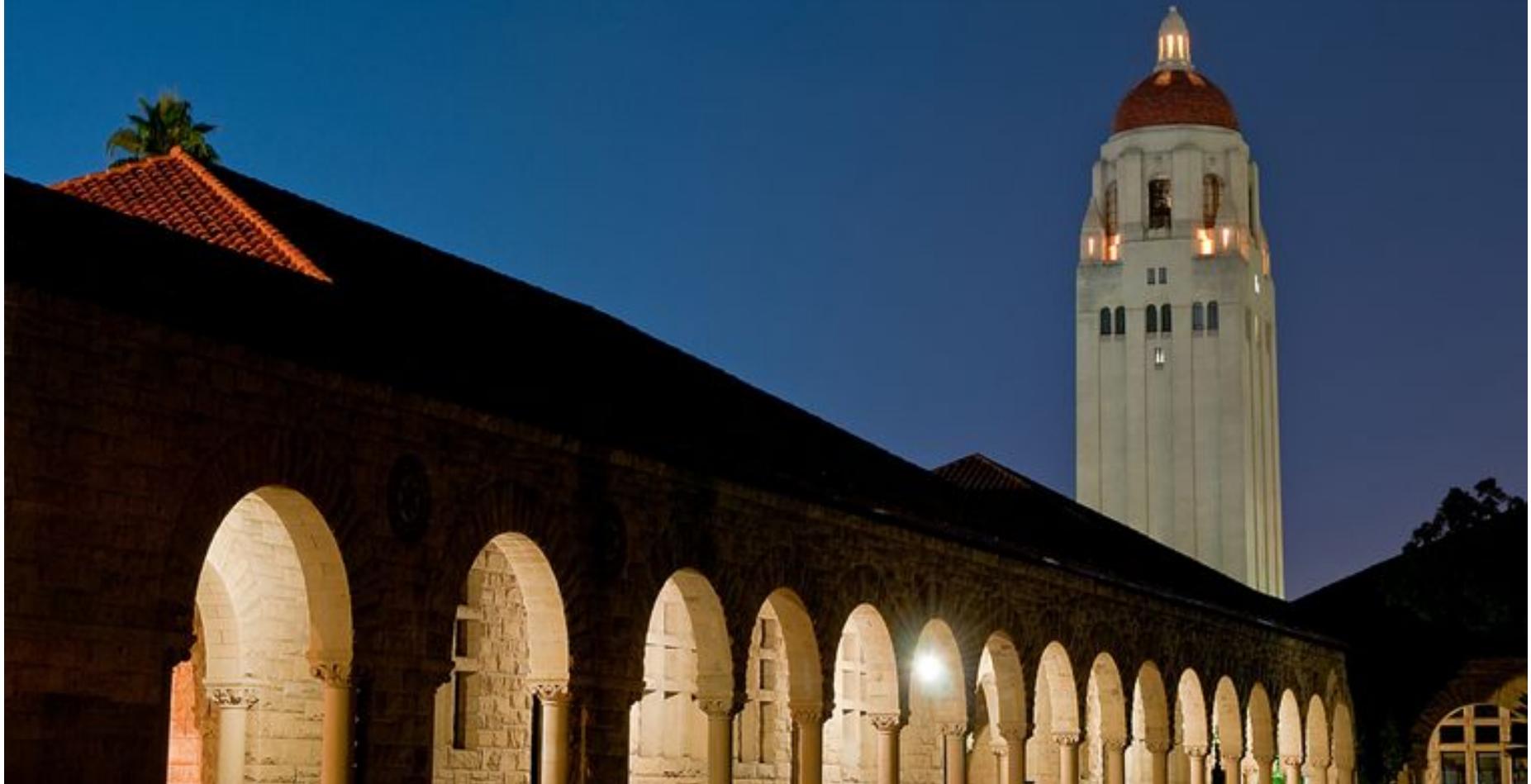
Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016  
Figure copyright Justin Johnson, 2015.

# Neural Style Transfer: Multiple Style Images

Mix style from multiple images by taking a weighted average of Gram matrices



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016  
Figure copyright Justin Johnson, 2015.



Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 13 - 74 May 16, 2019



Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 13 - 75 May 16, 2019



Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 13 - 76 May 16, 2019



# Fast Style Transfer



Replacing batch normalization with Instance Normalization improves results

Ulyanov et al. "Texture Networks: Feed-forward Synthesis of Textures and Stylized Images", ICML 2016

Ulyanov et al. "Instance Normalization: The Missing Ingredient for Fast Stylization", arXiv 2016

Figures copyright Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor Lempitsky, 2016. Reproduced with permission.

# Summary

Many methods for understanding CNN representations

**Activations:** Nearest neighbors, Dimensionality reduction, maximal patches, occlusion

**Gradients:** Saliency maps, class visualization, fooling images, feature inversion

**Fun:** DeepDream, Style Transfer.