

*CS 2770: Computer Vision*

# **Classification**

# **(Convolutional Neural Networks, Support Vector Machines)**

Prof. Adriana Kovashka

University of Pittsburgh

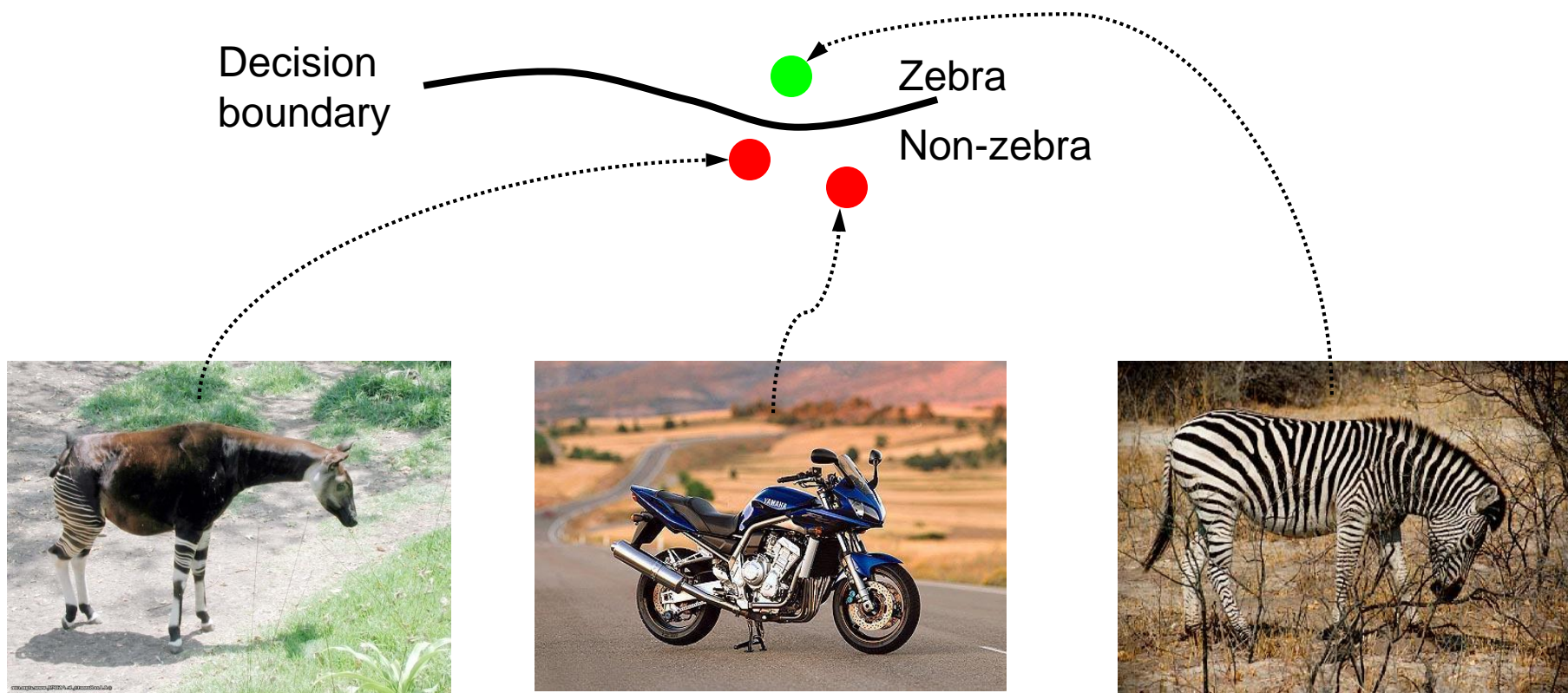
February 11, 2020

# Plan for this lecture

- What is classification?
- Support vector machines
  - Separable case / non-separable case
  - Linear / non-linear (kernels)
  - The importance of generalization
- Convolutional neural networks

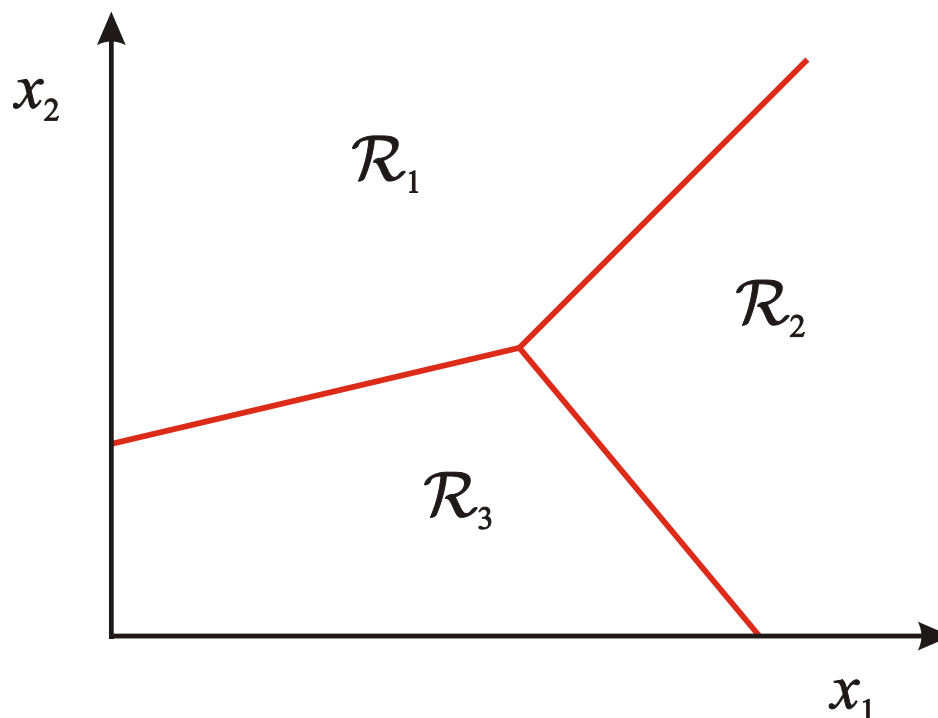
# Classification

- Given a feature representation for images, how do we learn a model for distinguishing features from different classes?



# Classification

- Assign input vector to one of two or more classes
- Input space divided into *decision regions* separated by *decision boundaries*



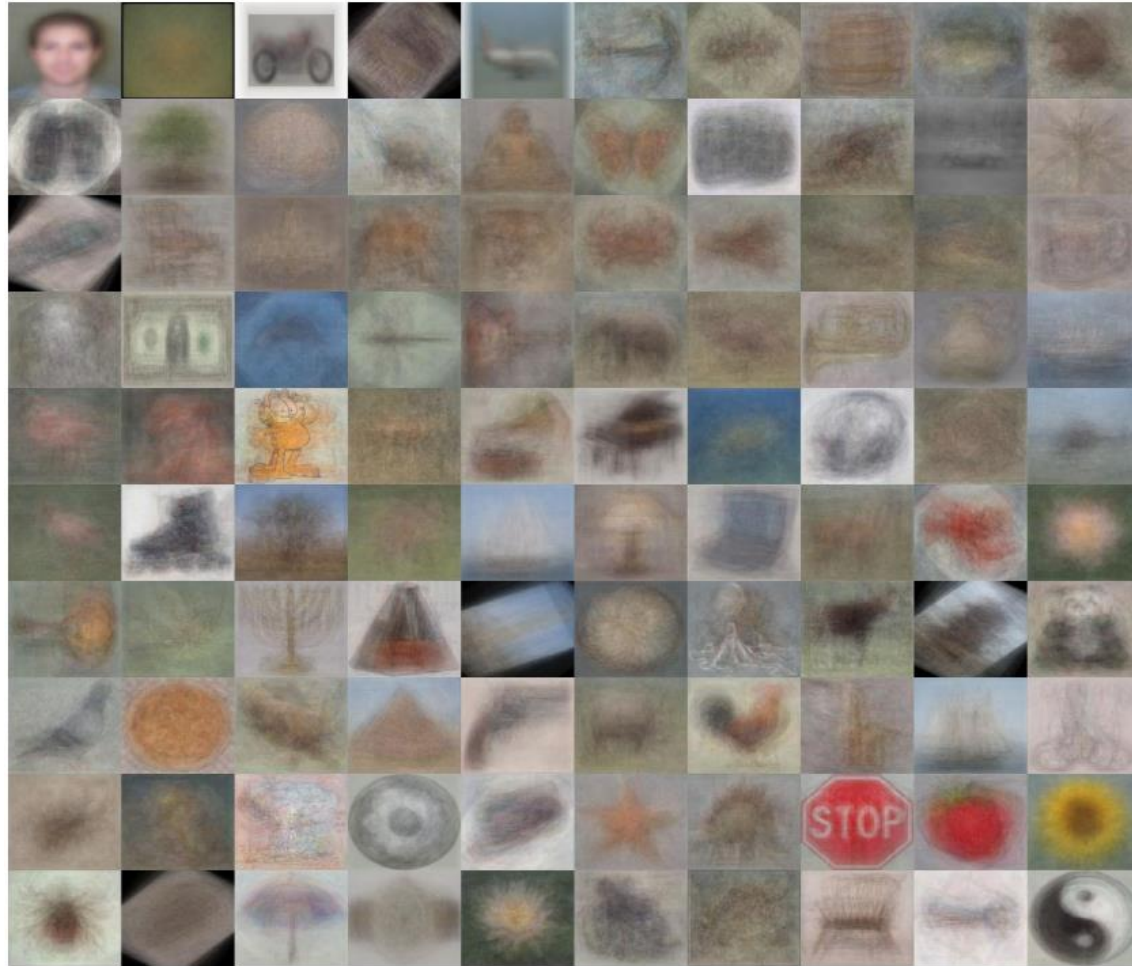


# Examples of image classification

- Two-class (binary): Cat vs Dog



- Multi-class (often): Object recognition



## Caltech 101 Average Object Images



# Examples of image classification

- Place recognition



Places Database [[Zhou et al. NIPS 2014](#)]

# Examples of image classification

- Material recognition



[[Bell et al. CVPR 2015](#)]



# Examples of image classification

- Image style recognition



HDR



Macro



Baroque



Rococo



Vintage



Noir



Northern Renaissance



Cubism



Minimal



Hazy



Impressionism



Post-Impressionism



Long Exposure



Romantic



Abs. Expressionism



Color Field Painting

Flickr Style: 80K images covering 20 styles.

Wikipaintings: 85K images for 25 art genres.

[[Karayev et al. BMVC 2014](#)]

# Recognition: A machine learning approach



# The machine learning framework

- Apply a prediction function to a feature representation of the image to get the desired output:

$$f(\text{apple image}) = \text{"apple"}$$

$$f(\text{tomato image}) = \text{"tomato"}$$

$$f(\text{cow image}) = \text{"cow"}$$

# The machine learning framework

$$y = f(x)$$

output      prediction function      image / image feature

- **Training:** given a *training set* of labeled examples  $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$ , estimate the prediction function  $f$  by minimizing the prediction error on the training set
- **Testing:** apply  $f$  to a never before seen *test example*  $\mathbf{x}$  and output the predicted value  $y = f(\mathbf{x})$



# The old-school way

## Training

Training  
Images



Image  
Features

Training  
Labels

Training

Learned  
model

## Testing



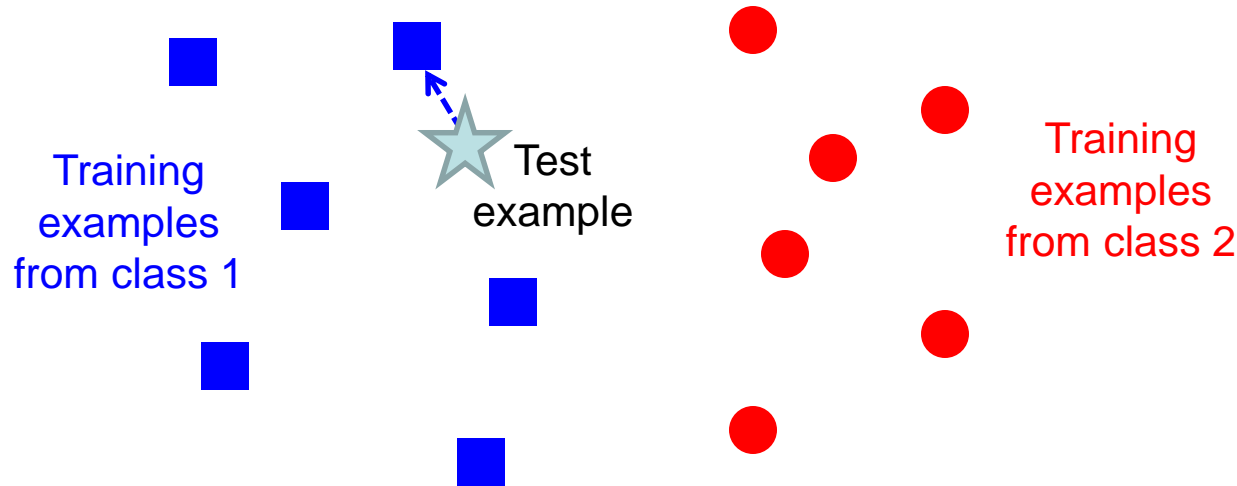
Test Image

Image  
Features

Learned  
model

Prediction

# The simplest classifier

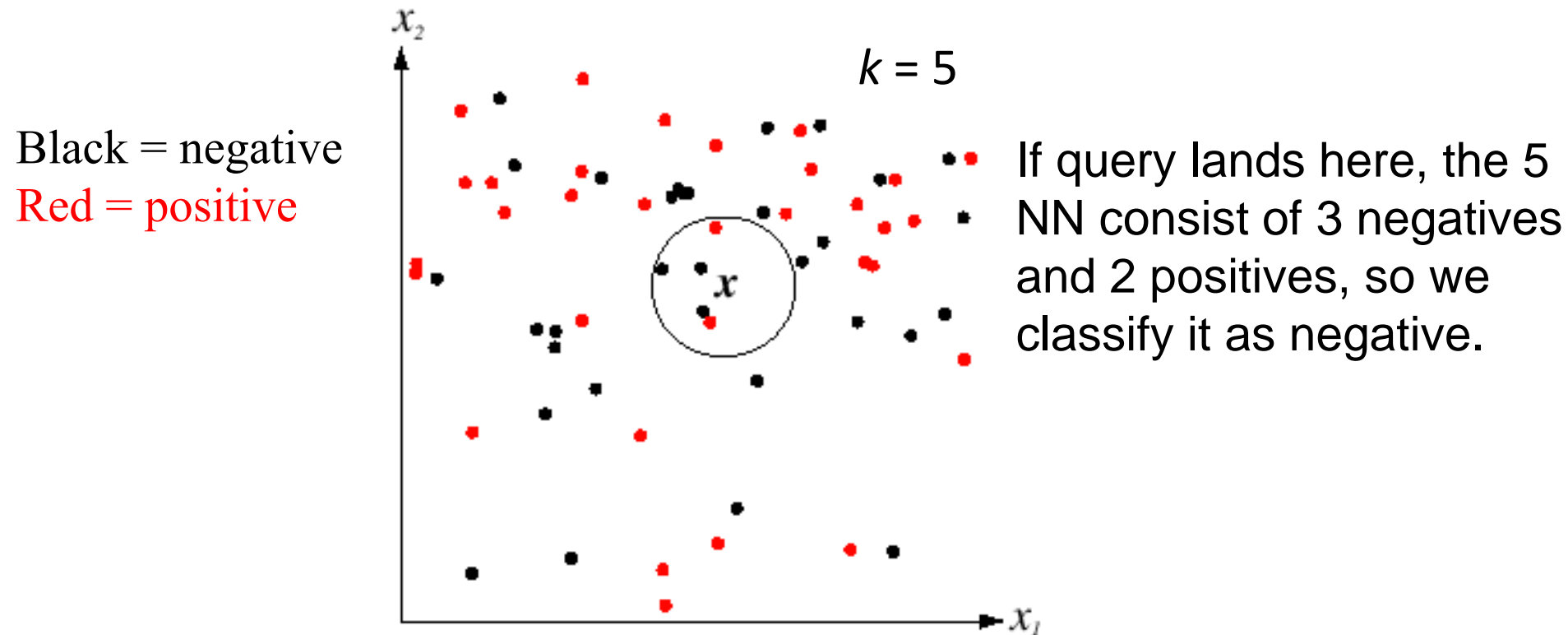


$f(\mathbf{x}) = \text{label of the training example nearest to } \mathbf{x}$

- All we need is a distance function for our inputs
- No training required!

# K-Nearest Neighbors classification

- For a new point, find the  $k$  closest points from training data
- Labels of the  $k$  points “vote” to classify



# im2gps: Estimating Geographic Information from a Single Image

James Hays and Alexei Efros, CVPR 2008

Where was this image taken?



Paris



Paris



Paris



Paris



Paris



Paris



Paris



Madrid



Rome



Paris



Cuba



Paris



Paris



Poland



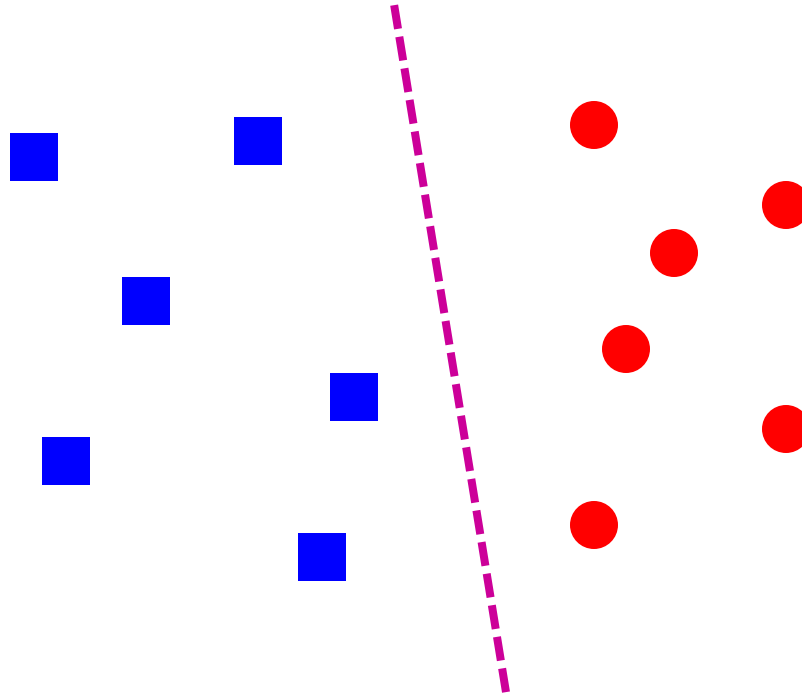
Paris



Paris

Nearest Neighbors according to bag of SIFT + color histogram + a few others

# Linear classifier

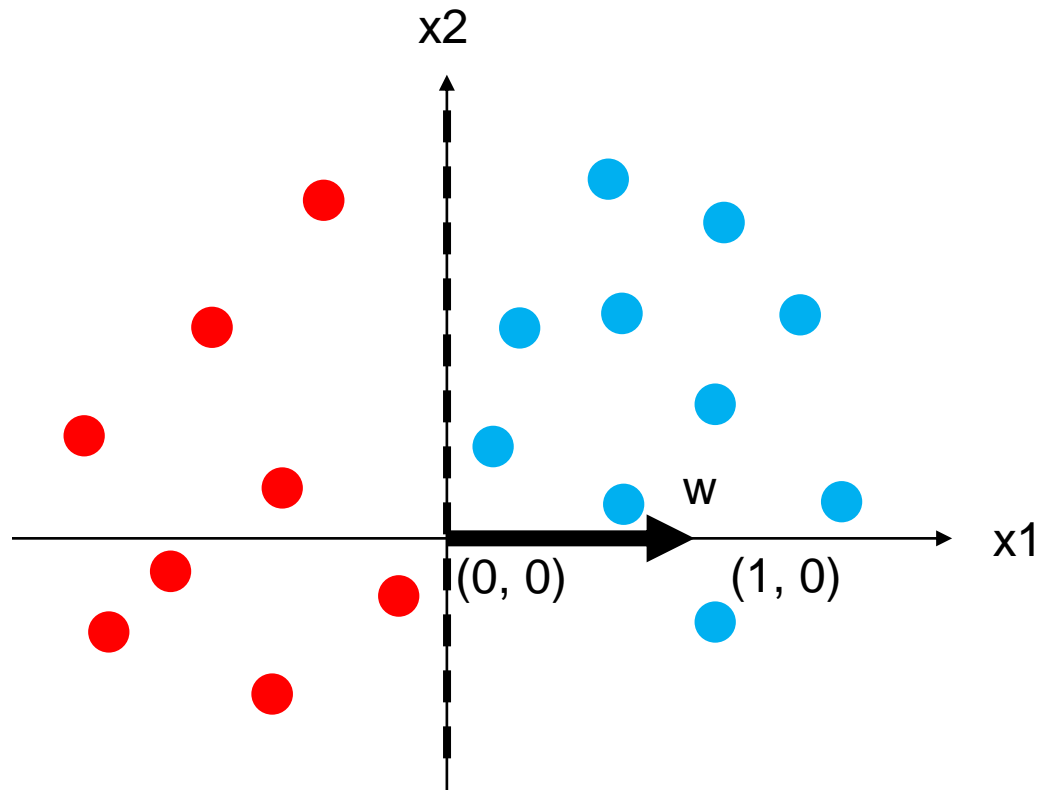


- Find a *linear function* to separate the classes

$$f(\mathbf{x}) = \text{sgn}(w_1x_1 + w_2x_2 + \dots + w_Dx_D) = \text{sgn}(\mathbf{w} \cdot \mathbf{x})$$

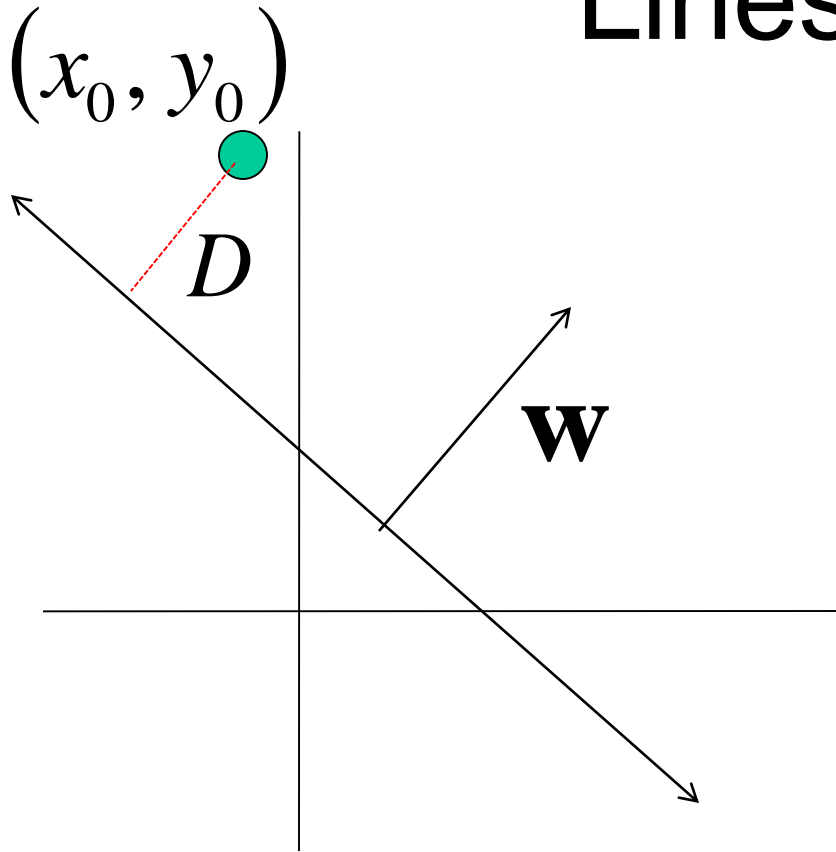
# Linear classifier

- Decision =  $\text{sign}(w^T x) = \text{sign}(w_1 * x_1 + w_2 * x_2)$



- What should the weights be?

# Lines in $\mathbb{R}^2$



Let  $\mathbf{w} = \begin{bmatrix} a \\ c \end{bmatrix}$   $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$

$$ax + cy + b = 0$$



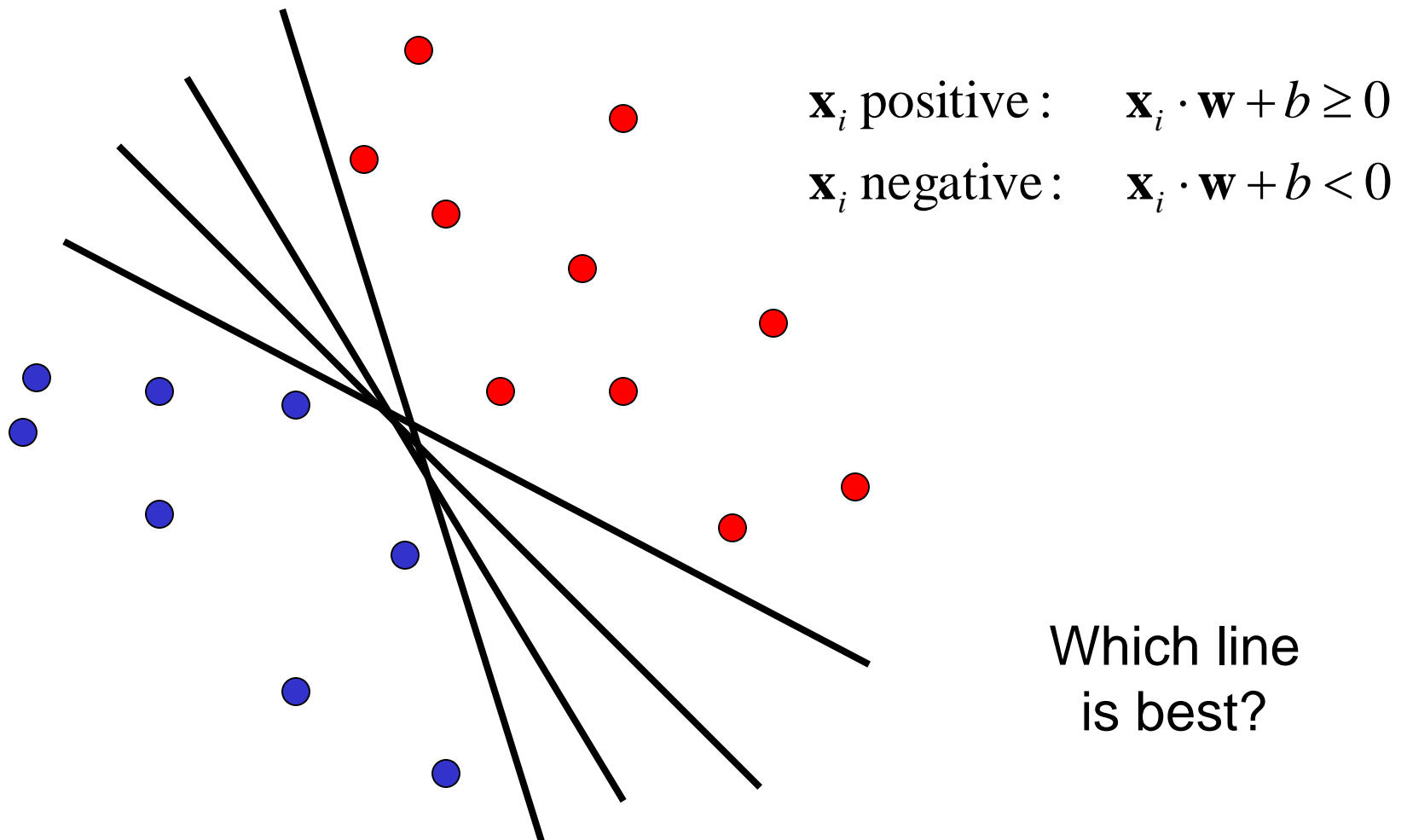
$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

$$D = \frac{|ax_0 + cy_0 + b|}{\sqrt{a^2 + c^2}} = \frac{|\mathbf{w}^T \mathbf{x} + b|}{\|\mathbf{w}\|} \quad \left. \vphantom{\frac{|\mathbf{w}^T \mathbf{x} + b|}{\|\mathbf{w}\|}} \right\} \begin{array}{l} \text{distance from} \\ \text{point to line} \end{array}$$

# Linear classifiers

---

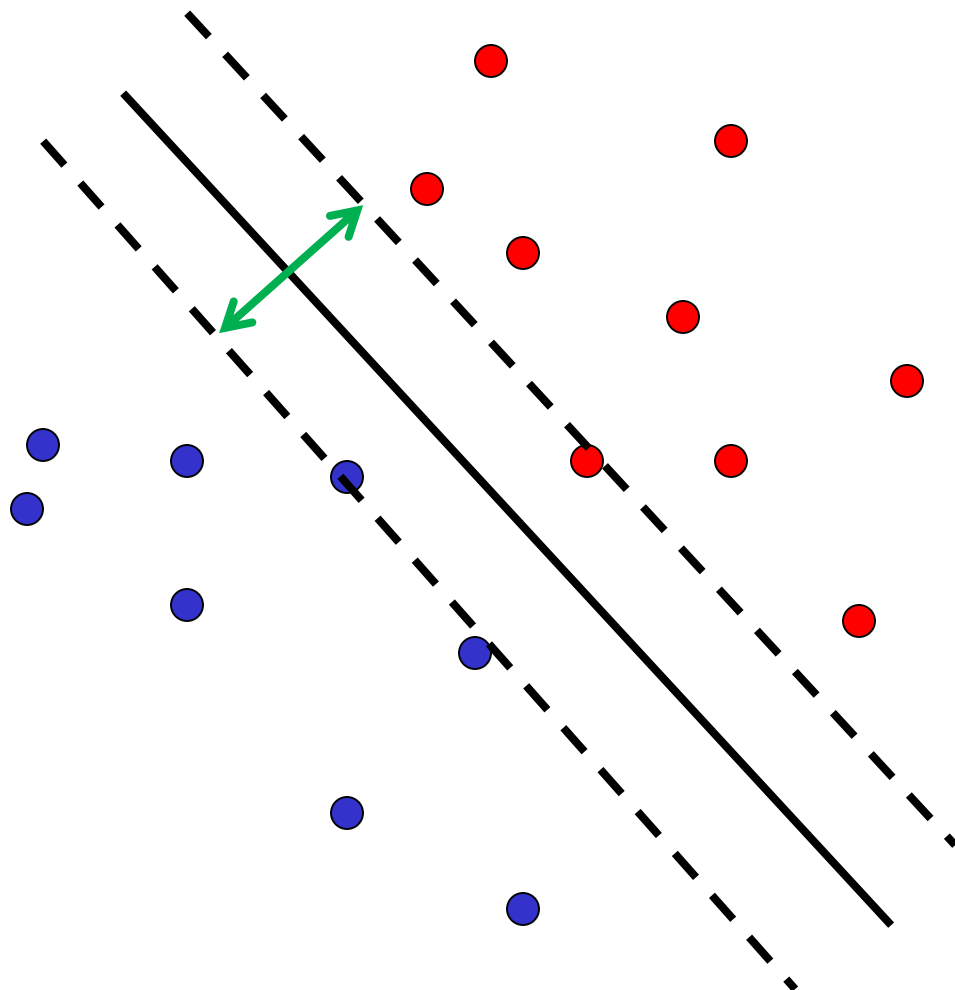
- Find linear function to separate positive and negative examples





# Support vector machines

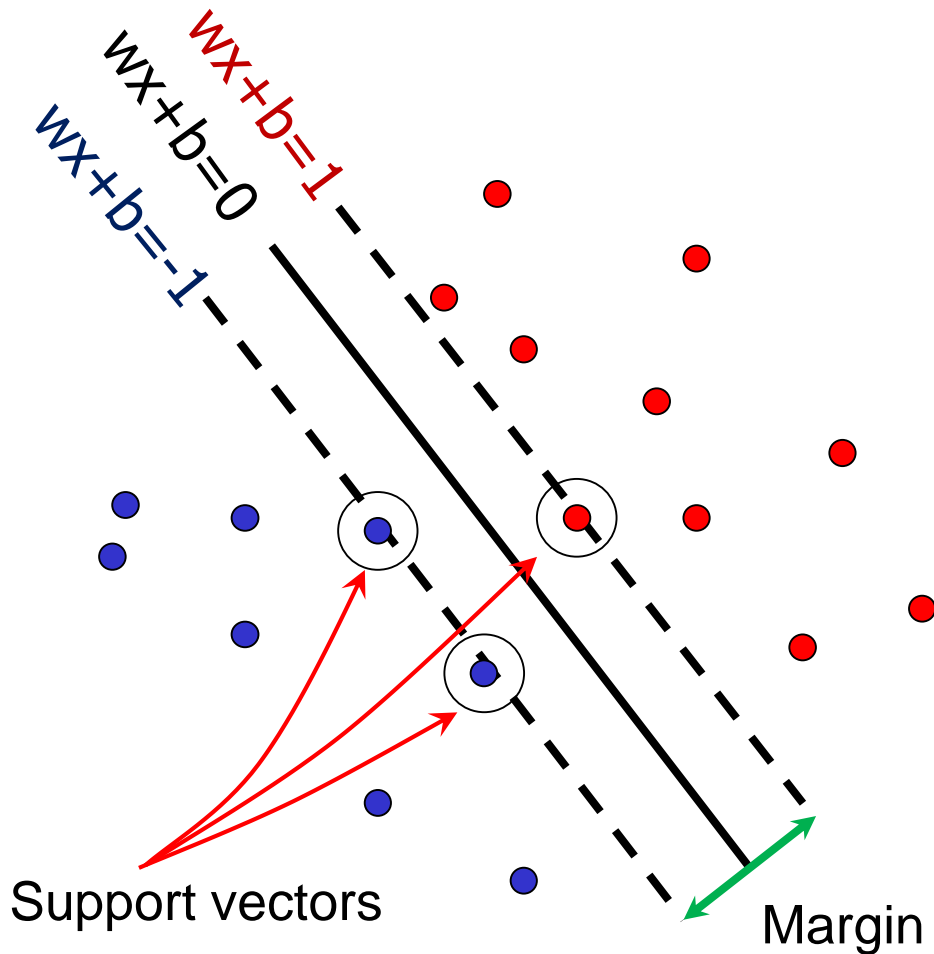
---



- Discriminative classifier based on *optimal separating line* (for 2d case)
- Maximize the *margin* between the positive and negative training examples

# Support vector machines

- Want line that maximizes the margin.



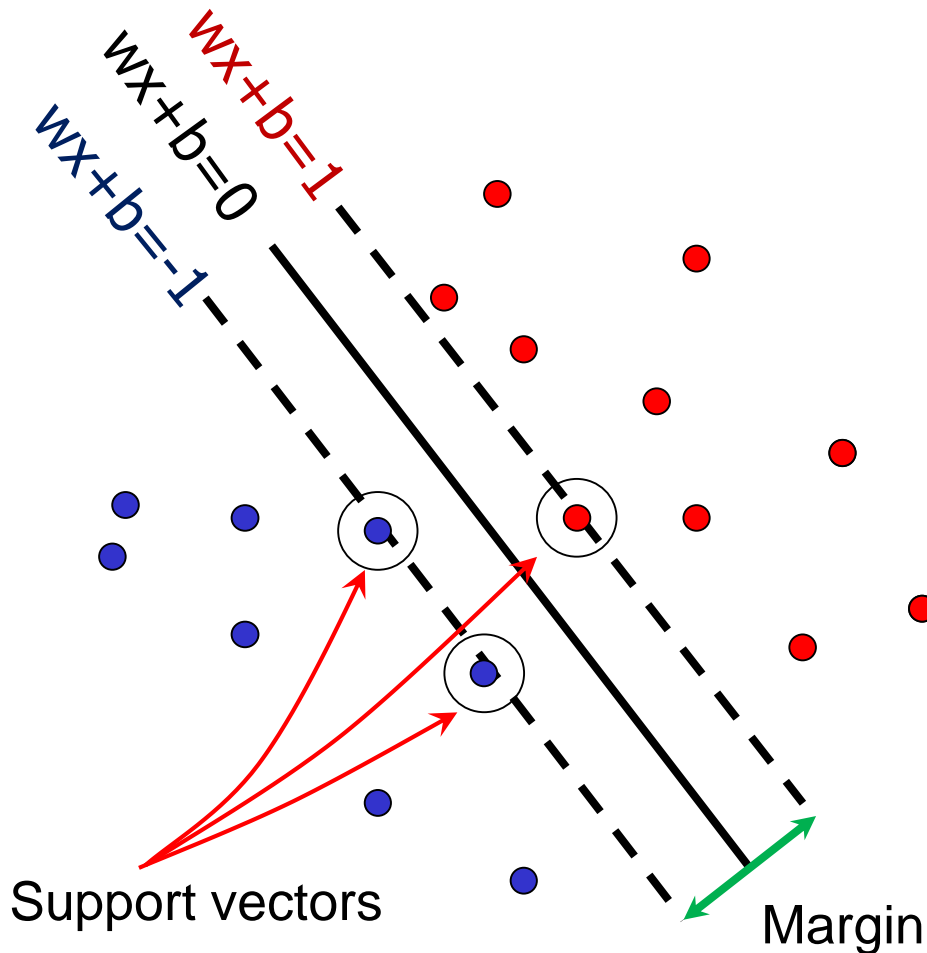
$$\mathbf{x}_i \text{ positive } (y_i = 1): \quad \mathbf{x}_i \cdot \mathbf{w} + b \geq 1$$

$$\mathbf{x}_i \text{ negative } (y_i = -1): \quad \mathbf{x}_i \cdot \mathbf{w} + b \leq -1$$

$$\text{For support, vectors,} \quad \mathbf{x}_i \cdot \mathbf{w} + b = \pm 1$$

# Support vector machines

- Want line that maximizes the margin.



$$\mathbf{x}_i \text{ positive } (y_i = 1): \quad \mathbf{x}_i \cdot \mathbf{w} + b \geq 1$$

$$\mathbf{x}_i \text{ negative } (y_i = -1): \quad \mathbf{x}_i \cdot \mathbf{w} + b \leq -1$$

$$\text{For support, vectors, } \mathbf{x}_i \cdot \mathbf{w} + b = \pm 1$$

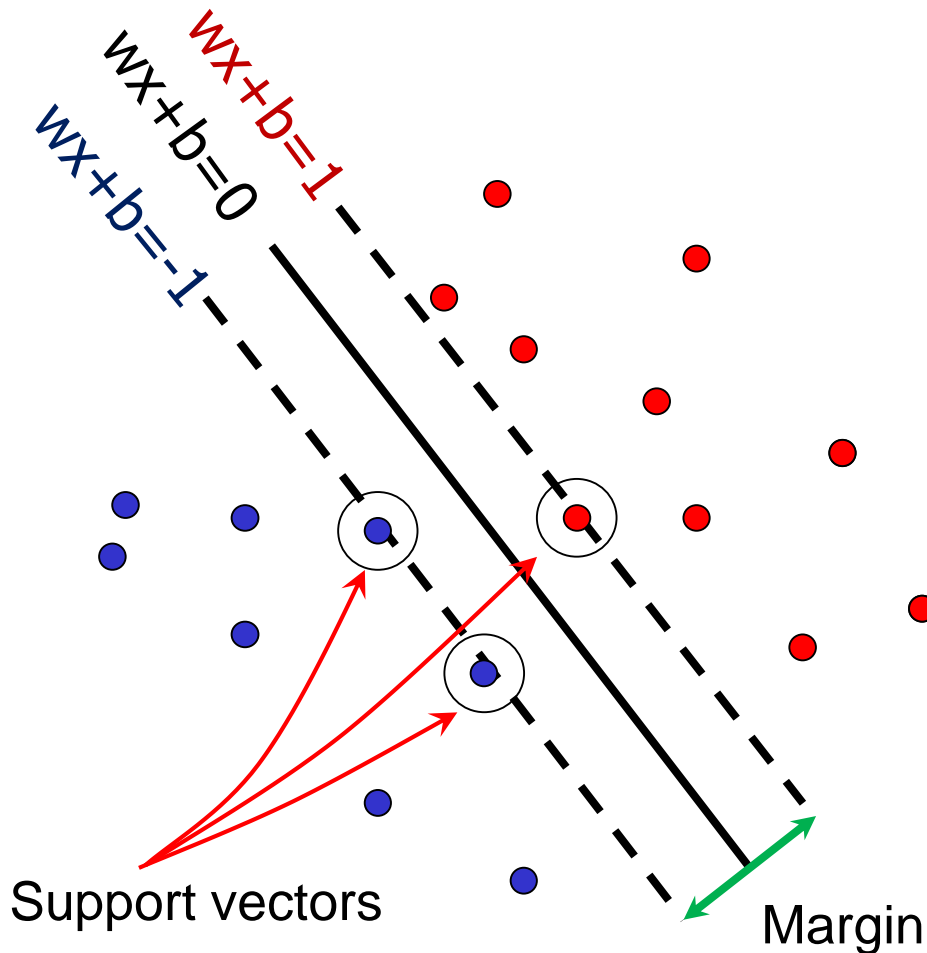
$$\text{Distance between point and line: } \frac{|\mathbf{x}_i \cdot \mathbf{w} + b|}{\|\mathbf{w}\|}$$

For support vectors:

$$\frac{\mathbf{w}^T \mathbf{x} + b}{\|\mathbf{w}\|} = \frac{\pm 1}{\|\mathbf{w}\|} \quad M = \left| \frac{1}{\|\mathbf{w}\|} - \frac{-1}{\|\mathbf{w}\|} \right| = \frac{2}{\|\mathbf{w}\|}$$

# Support vector machines

- Want line that maximizes the margin.



$\mathbf{x}_i$  positive ( $y_i = 1$ ):  $\mathbf{x}_i \cdot \mathbf{w} + b \geq 1$

$\mathbf{x}_i$  negative ( $y_i = -1$ ):  $\mathbf{x}_i \cdot \mathbf{w} + b \leq -1$

For support, vectors,  $\mathbf{x}_i \cdot \mathbf{w} + b = \pm 1$

Distance between point and line:  $\frac{|\mathbf{x}_i \cdot \mathbf{w} + b|}{\|\mathbf{w}\|}$

Therefore, the margin is  $2 / \|\mathbf{w}\|$

# Finding the maximum margin line

---

1. Maximize margin  $2/\|\mathbf{w}\|$
2. Correctly classify all training data points:

$$\mathbf{x}_i \text{ positive } (y_i = 1): \quad \mathbf{x}_i \cdot \mathbf{w} + b \geq 1$$

$$\mathbf{x}_i \text{ negative } (y_i = -1): \quad \mathbf{x}_i \cdot \mathbf{w} + b \leq -1$$

*Quadratic optimization problem:*

$$\text{Minimize } \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

$$\text{Subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$$

One constraint for each training point.

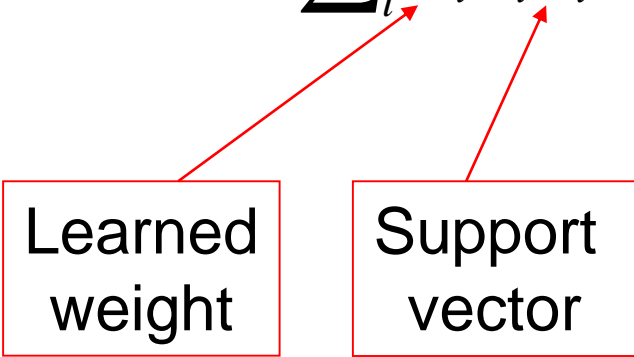
Note sign trick.

# Finding the maximum margin line

---

- Solution:  $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$

Learned  
weight



The diagram illustrates the components of the weight vector  $\mathbf{w}$  in the equation  $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$ . Two red arrows originate from the boxes below. The first arrow points from the 'Learned weight' box to the coefficient  $\alpha_i$  in the summation. The second arrow points from the 'Support vector' box to the vector  $\mathbf{x}_i$  in the summation.

Support  
vector

# Finding the maximum margin line

---

- Solution:  $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$   
 $b = y_i - \mathbf{w} \cdot \mathbf{x}_i$  (for any support vector)

- Classification function:

$$\begin{aligned} f(x) &= \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) \\ &= \text{sign}\left(\sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} + b\right) \end{aligned}$$

*If  $f(x) < 0$ , classify as negative, otherwise classify as positive.*

- Notice that it relies on an *inner product* between the test point  $\mathbf{x}$  and the support vectors  $\mathbf{x}_i$
- (Solving the optimization problem also involves computing the inner products  $\mathbf{x}_i \cdot \mathbf{x}_j$  between all pairs of training points)

# Inner product

---

- The decision boundary for the SVM and its optimization depend on the inner product of two data points (vectors):

$$\mathbf{x}_i^T \mathbf{x}_j$$

$$\begin{aligned} f(x) &= \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) \\ &= \text{sign}\left(\sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} + b\right) \end{aligned}$$

- The inner product is equal

$$(\mathbf{x}_i^T \mathbf{x}) = \|\mathbf{x}_i\| * \|\mathbf{x}\| \cos \theta$$

If the angle in between them is 0 then:  $(\mathbf{x}_i^T \mathbf{x}) = \|\mathbf{x}_i\| * \|\mathbf{x}\|$

If the angle between them is 90 then:  $(\mathbf{x}_i^T \mathbf{x}) = 0$

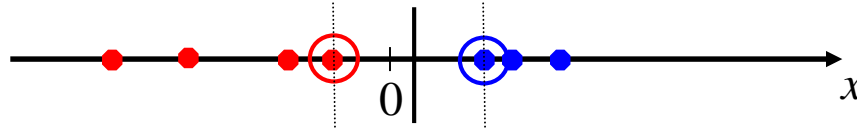
**The inner product measures how similar the two vectors are**



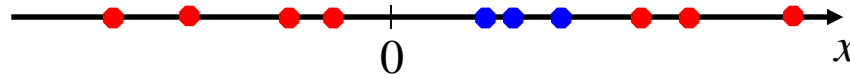
# Nonlinear SVMs

---

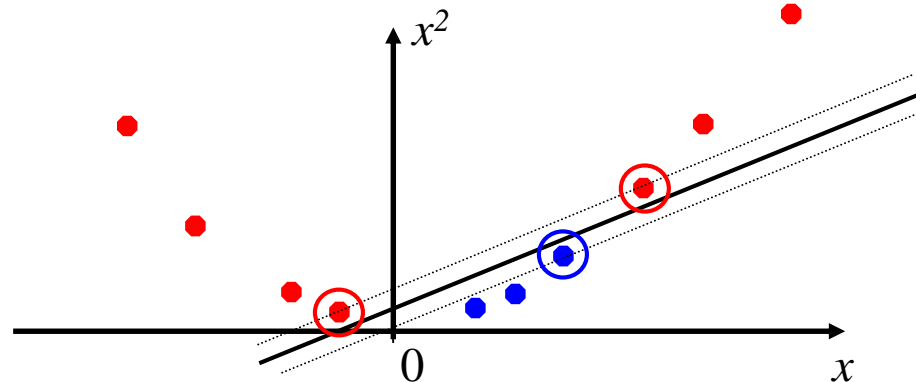
- Datasets that are linearly separable work out great:



- But what if the dataset is just too hard?



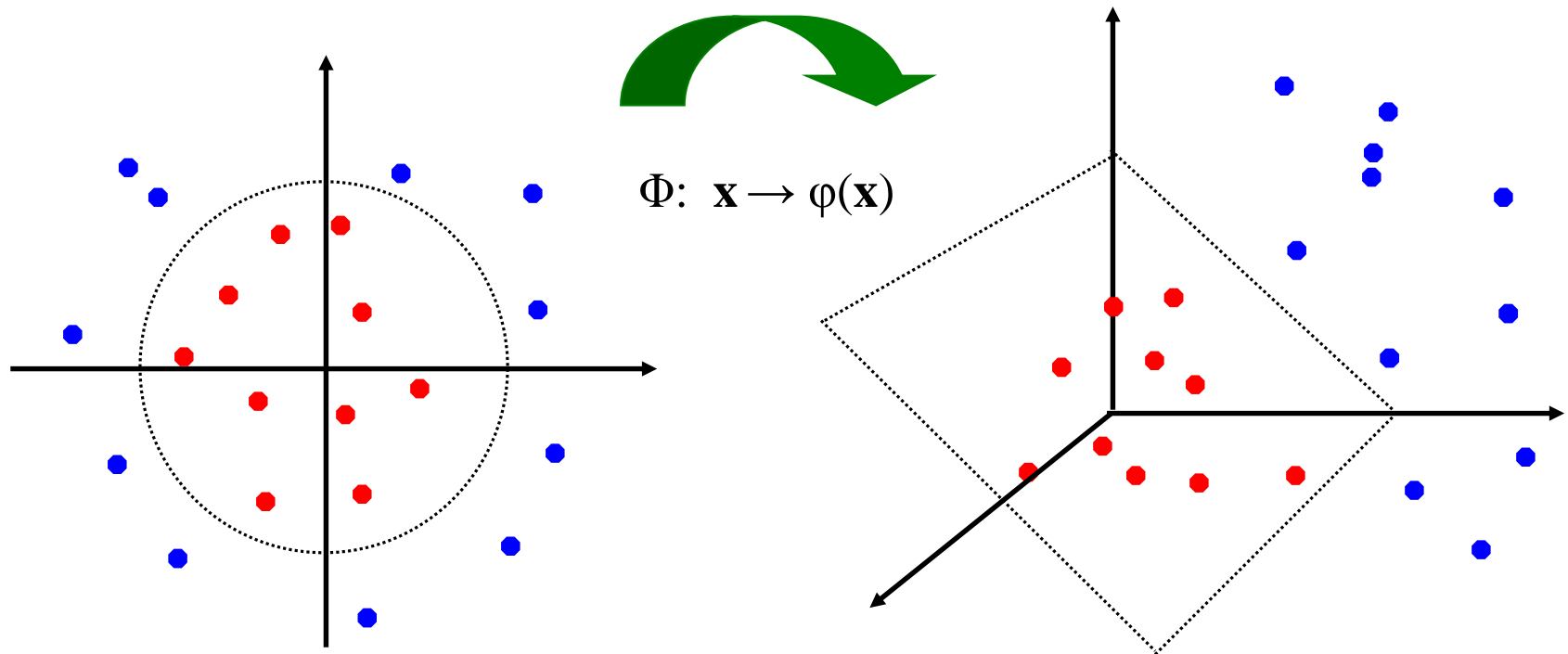
- We can map it to a higher-dimensional space:



# Nonlinear SVMs

---

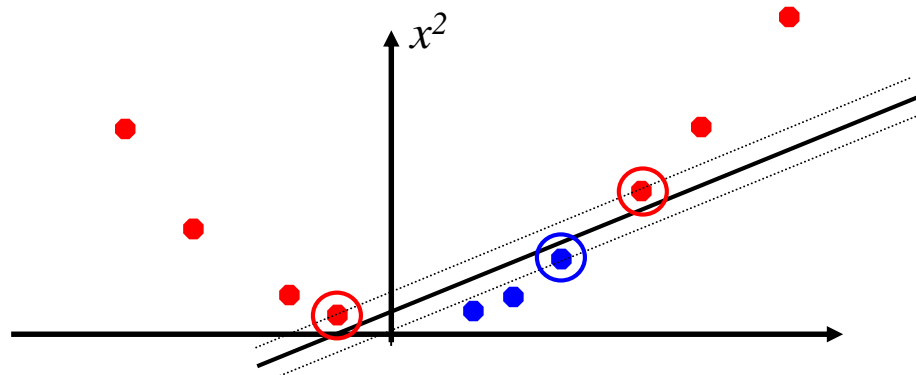
- General idea: the original input space can always be mapped to some higher-dimensional feature space where the training set is separable:



# Nonlinear kernel: Example

---

- Consider the mapping  $\varphi(x) = (x, x^2)$



$$\varphi(x) \cdot \varphi(y) = (x, x^2) \cdot (y, y^2) = xy + x^2 y^2$$

$$K(x, y) = xy + x^2 y^2$$

# The “Kernel Trick”

---

- The linear classifier relies on dot product between vectors  $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$
- If every data point is mapped into high-dimensional space via some transformation  $\Phi: \mathbf{x}_i \rightarrow \phi(\mathbf{x}_i)$ , the dot product becomes:  $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$
- A *kernel function* is similarity function that corresponds to an inner product in some expanded feature space
- *The kernel trick*: instead of explicitly computing the lifting transformation  $\phi(\mathbf{x})$ , define a kernel function  $K$  such that:  $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$

# Examples of kernel functions

---

- Linear:  $K(x_i, x_j) = x_i^T x_j$

- Polynomials of degree up to  $d$ :

$$K(x_i, x_j) = (x_i^T x_j + 1)^d$$

- Gaussian RBF:

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

- Histogram intersection:

$$K(x_i, x_j) = \sum_k \min(x_i(k), x_j(k))$$

# Hard-margin SVMs

---

$$\min_{\mathbf{w}} \quad \underbrace{\frac{1}{2} \|\mathbf{w}\|^2}_{\text{Maximize margin}}$$

The  $\mathbf{w}$  that minimizes...

Maximize margin

$$\text{subject to} \quad y_i \mathbf{w}^T \mathbf{x}_i \geq 1, \quad \forall i = 1, \dots, N$$

# Soft-margin SVMs

---

The  $w$  that minimizes...

$$\min_w \underbrace{\frac{1}{2} \|\mathbf{w}\|^2}_{\text{Maximize margin}} + \underbrace{C \sum_{i=1}^N \xi_i}_{\text{Minimize misclassification}}$$

Misclassification cost

# data samples

Slack variable

subject to

$$y_i \mathbf{w}^T \mathbf{x}_i \geq 1 - \xi_i,$$
$$\xi_i \geq 0, \quad \forall i = 1, \dots, N$$

# What about multi-class SVMs?

---

- Unfortunately, there is no “definitive” multi-class SVM formulation
- In practice, we have to obtain a multi-class SVM by combining multiple two-class SVMs
- One vs. others
  - Training: learn an SVM for each class vs. the others
  - Testing: apply each SVM to the test example, and assign it to the class of the SVM that returns the highest decision value
- One vs. one
  - Training: learn an SVM for each pair of classes
  - Testing: each learned SVM “votes” for a class to assign to the test example



# Multi-class problems

---

## One-vs-all (a.k.a. one-vs-others)

- Train  $K$  classifiers
- In each, pos = data from class  $i$ , neg = data from classes other than  $i$
- The class with the most confident prediction wins
- Example:
  - You have 4 classes, train 4 classifiers
  - 1 vs others: score 3.5
  - 2 vs others: score 6.2
  - 3 vs others: score 1.4
  - 4 vs other: score 5.5
  - Final prediction: class 2

# Multi-class problems

---

## One-vs-one (a.k.a. all-vs-all)

- Train  $K(K-1)/2$  binary classifiers (all pairs of classes)
- They all vote for the label
- Example:
  - You have 4 classes, then train 6 classifiers
  - 1 vs 2, 1 vs 3, 1 vs 4, 2 vs 3, 2 vs 4, 3 vs 4
  - Votes: 1, 1, 4, 2, 4, 4
  - Final prediction is class 4

# Some SVM packages

---

- LIBSVM <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>
- LIBLINEAR  
<https://www.csie.ntu.edu.tw/~cjlin/liblinear/>
- SVM Light <http://svmlight.joachims.org/>

# Linear classifiers vs nearest neighbors

- Linear pros:
  - + Low-dimensional *parametric* representation
  - + Very fast at test time
- Linear cons:
  - Can be tricky to select best kernel function for a problem
  - Learning can take a very long time for large-scale problem
- NN pros:
  - + Works for any number of classes
  - + Decision boundaries not necessarily linear
  - + *Nonparametric* method
  - + Simple to implement
- NN cons:
  - Slow at test time (large search problem to find neighbors)
  - Storage of data
  - Especially need good distance function (but true for all classifiers)

# Training vs Testing

- What do we want?
  - High accuracy on training data?
  - No, high accuracy on *unseen/new/test data*!
  - Why is this tricky?
- Training data
  - Features (x) and labels (y) used to learn mapping f
- Test data
  - Features (x) used to make a prediction
  - Labels (y) only used to see how well we've learned f!!!
- Validation data
  - Held-out set of the *training data*
  - Can use both features (x) and labels (y) to tune parameters of the model we're learning

# Generalization



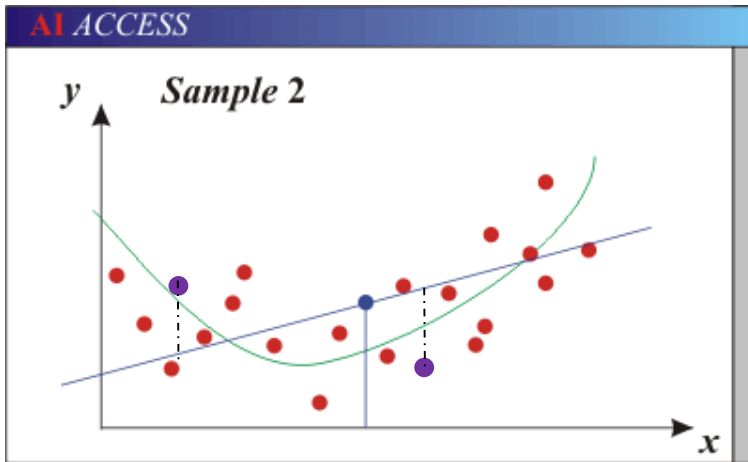
Training set (labels known)



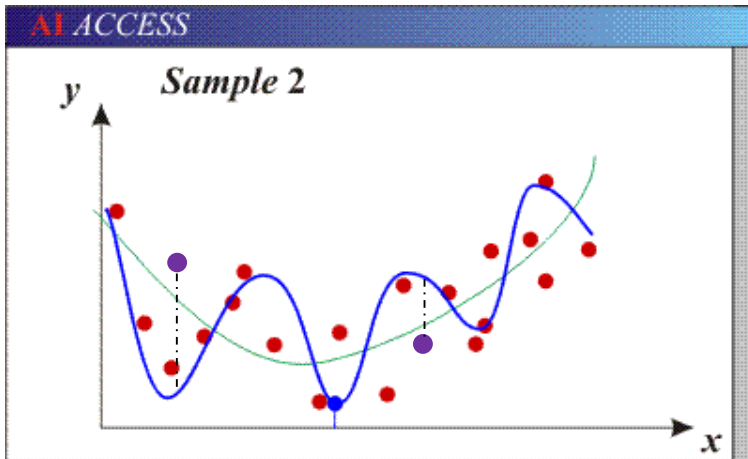
Test set (labels unknown)

- How well does a learned model generalize from the data it was trained on to a new test set?

# Generalization



- Underfitting: Models with too few parameters are inaccurate because of a large bias (not enough flexibility).



- Overfitting: Models with too many parameters are inaccurate because of a large variance (too much sensitivity to the sample).

Purple dots = possible test points

Red dots = training data (all that we see before we ship off our model!)

Green curve = true underlying model

Blue curve = our predicted model/fit

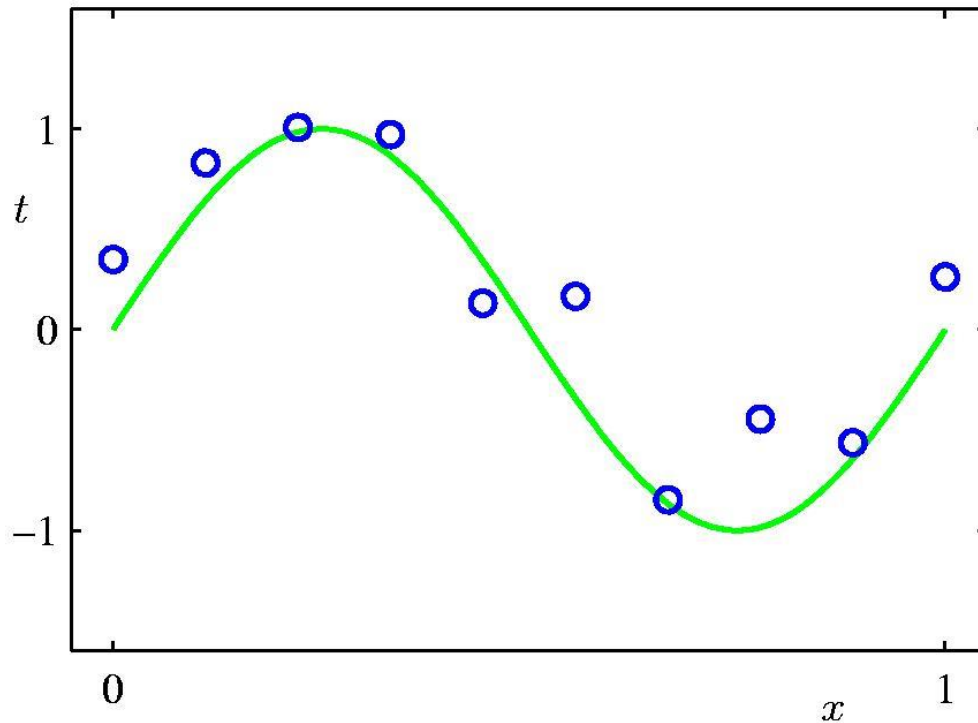
# Generalization

- Components of generalization error
  - **Noise** in our observations: unavoidable
  - **Bias**: how much the average model over all training sets differs from the true model
    - Inaccurate assumptions/simplifications made by the model
  - **Variance**: how much models estimated from different training sets differ from each other
- **Underfitting**: model is too “simple” to represent all the relevant class characteristics
  - High bias and low variance
  - High training error and high test error
- **Overfitting**: model is too “complex” and fits irrelevant characteristics (noise) in the data
  - Low bias and high variance
  - Low training error and high test error



# Polynomial Curve Fitting

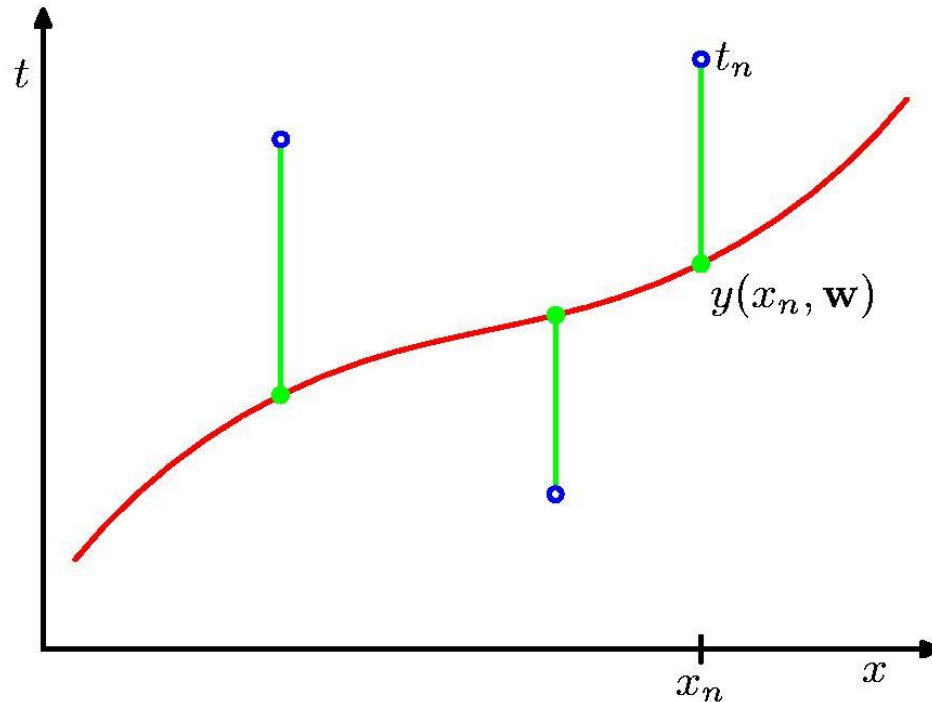
---



$$y(x, \mathbf{w}) = w_0 + w_1x + w_2x^2 + \dots + w_Mx^M = \sum_{j=0}^M w_jx^j$$

# Sum-of-Squares Error Function

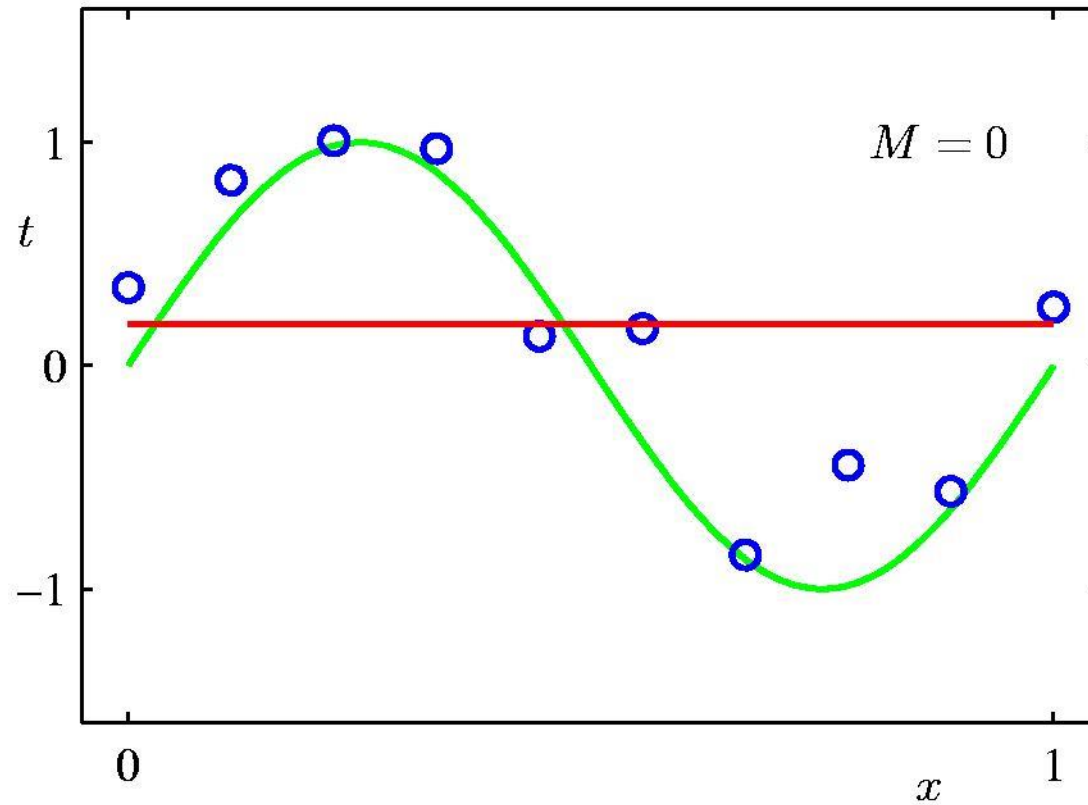
---



$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, \mathbf{w}) - t_n\}^2$$

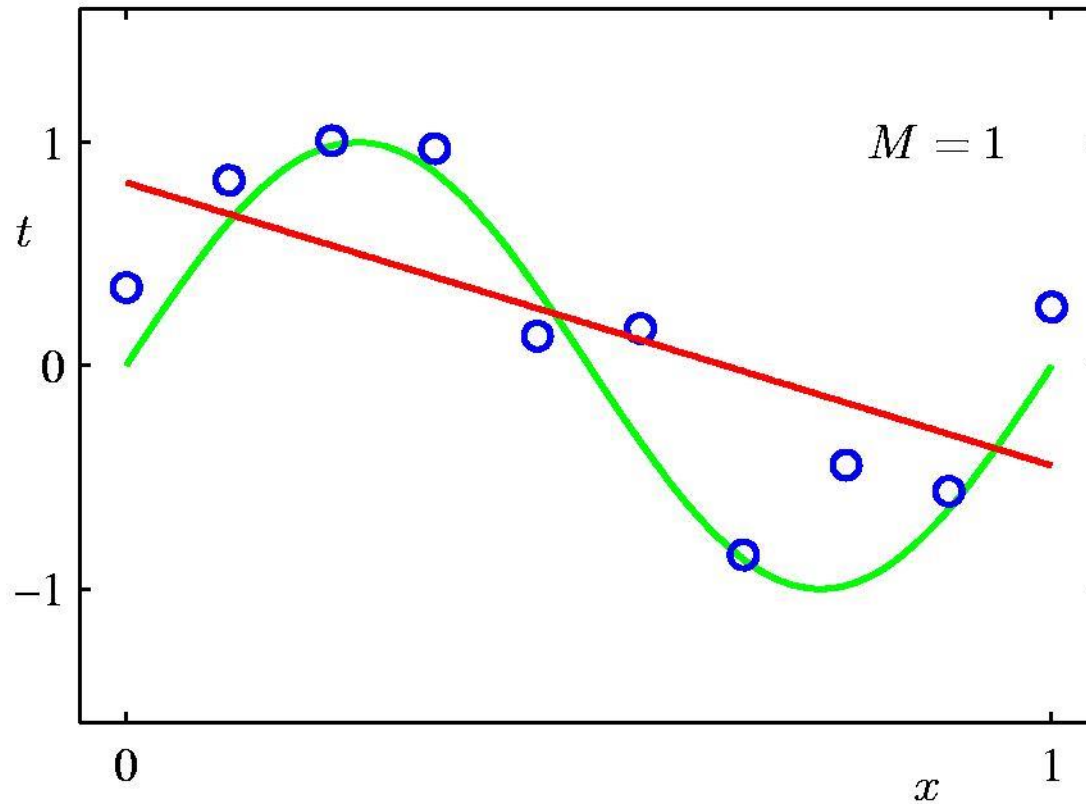
# 0<sup>th</sup> Order Polynomial

---



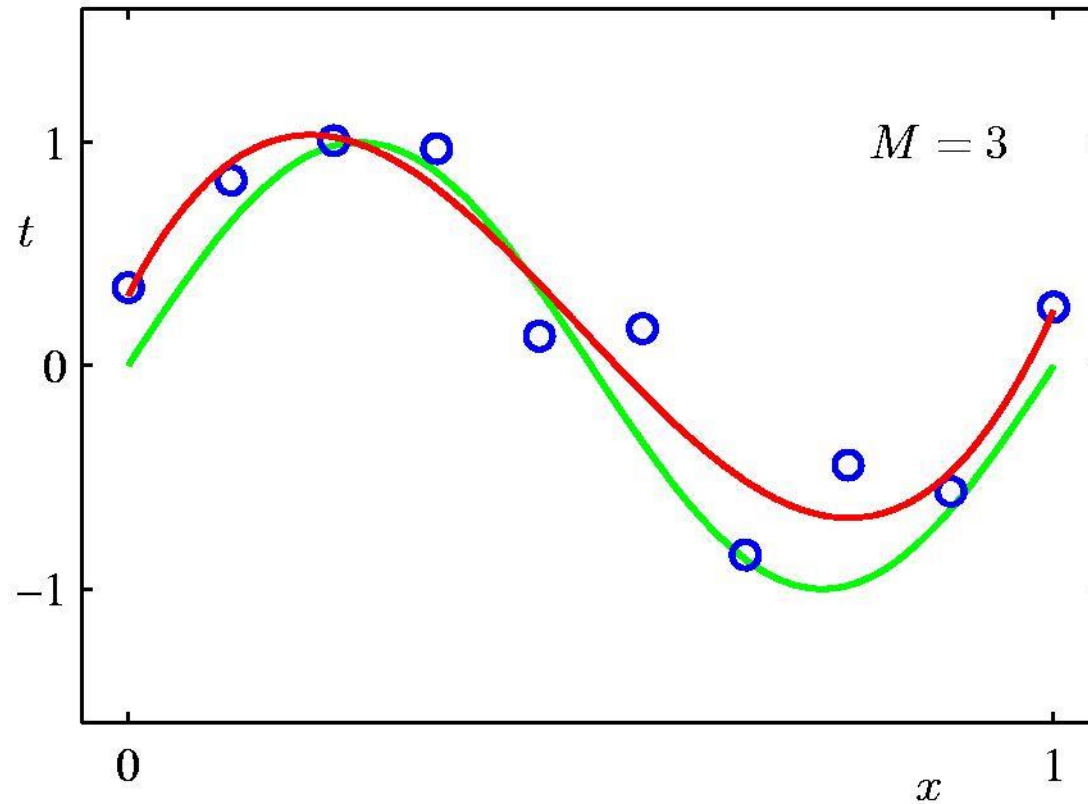
# 1<sup>st</sup> Order Polynomial

---



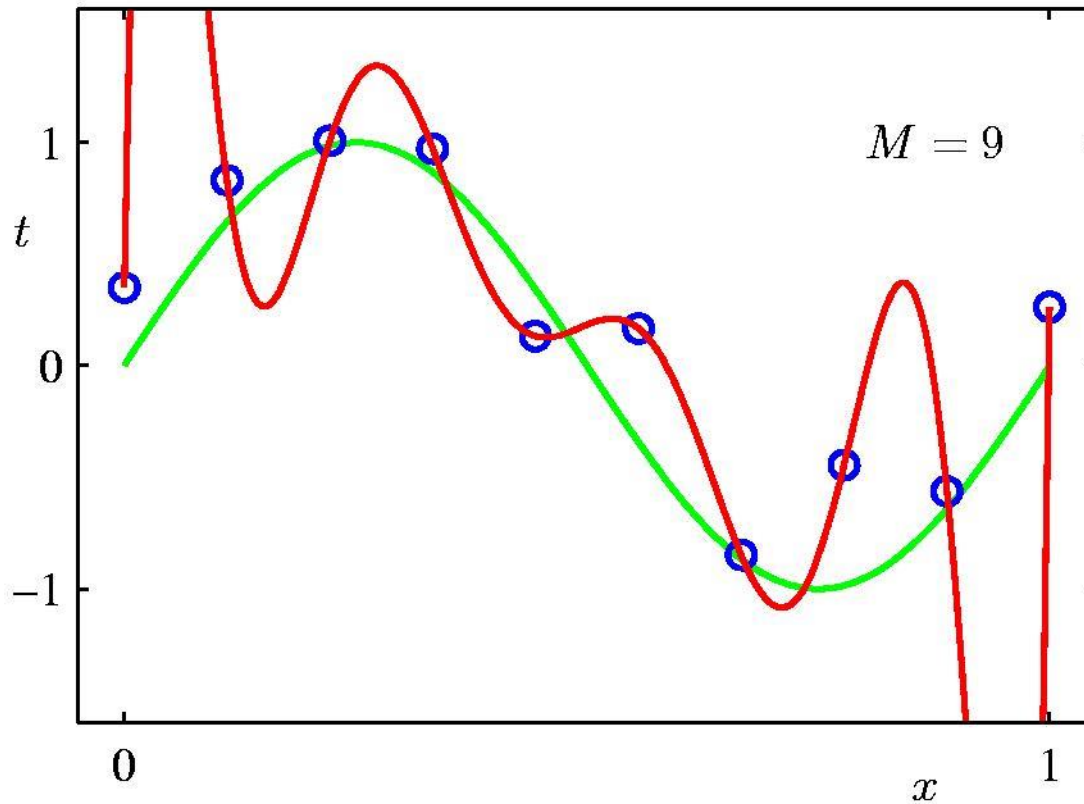
# 3<sup>rd</sup> Order Polynomial

---



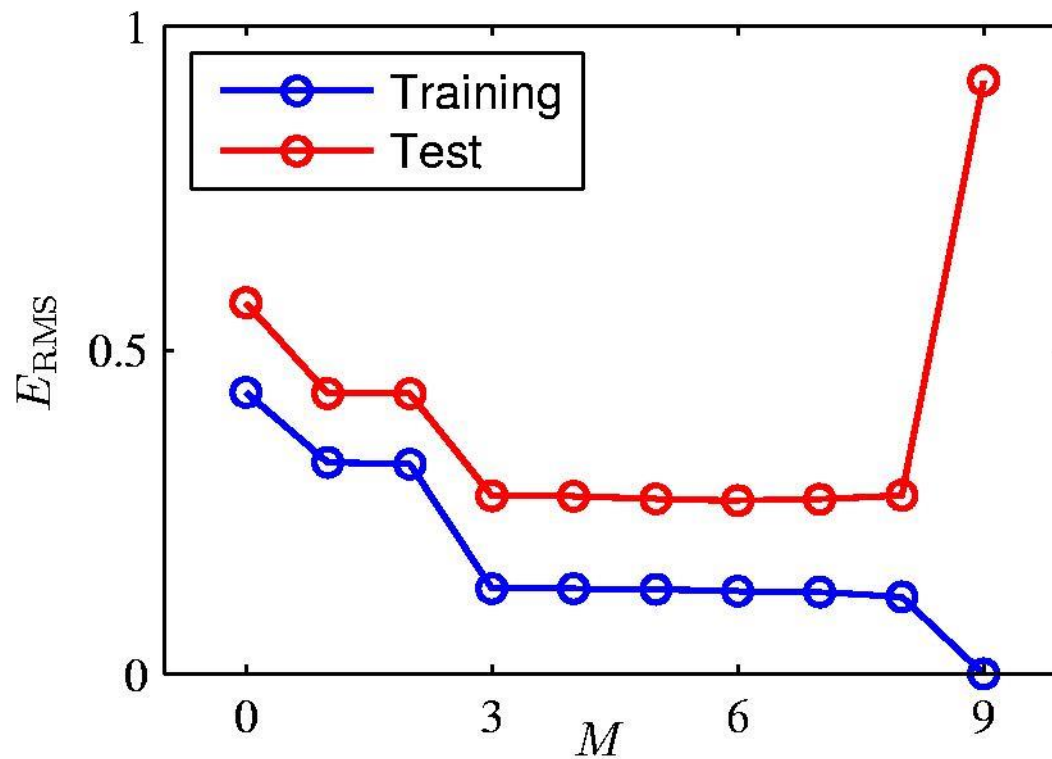
# 9<sup>th</sup> Order Polynomial

---



# Over-fitting

---

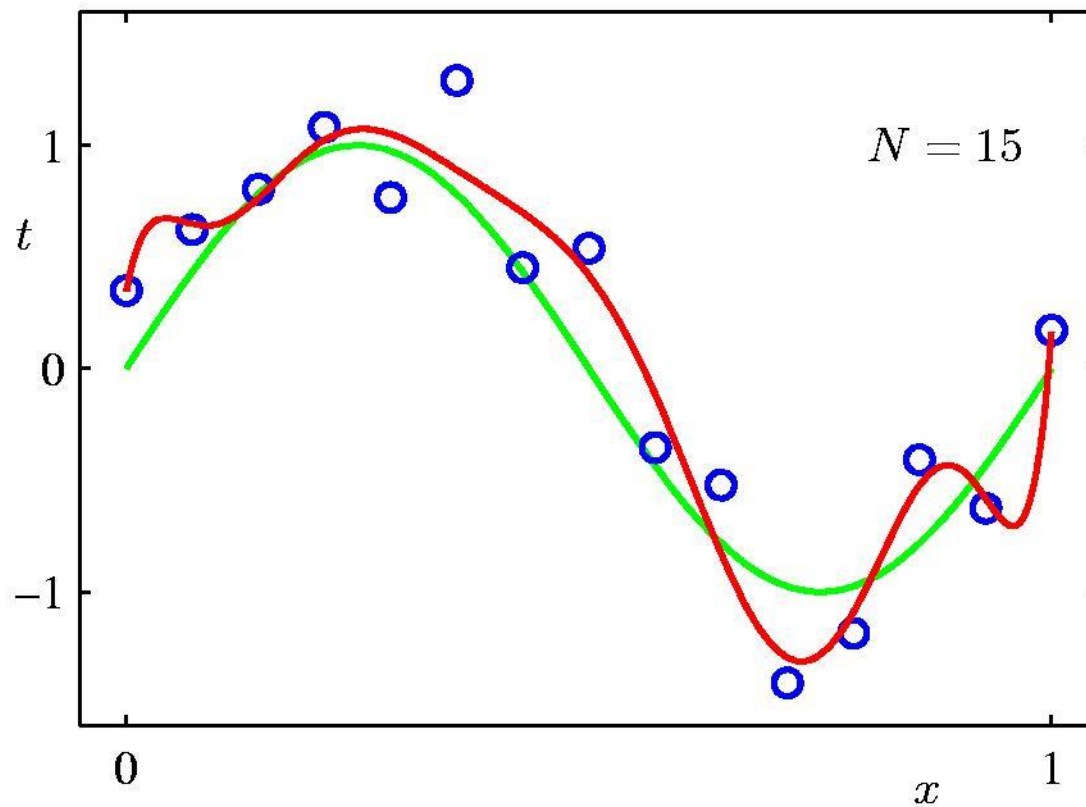


Root-Mean-Square (RMS) Error:  $E_{\text{RMS}} = \sqrt{2E(\mathbf{w}^*)/N}$

# Data Set Size: $N = 15$

---

9<sup>th</sup> Order Polynomial

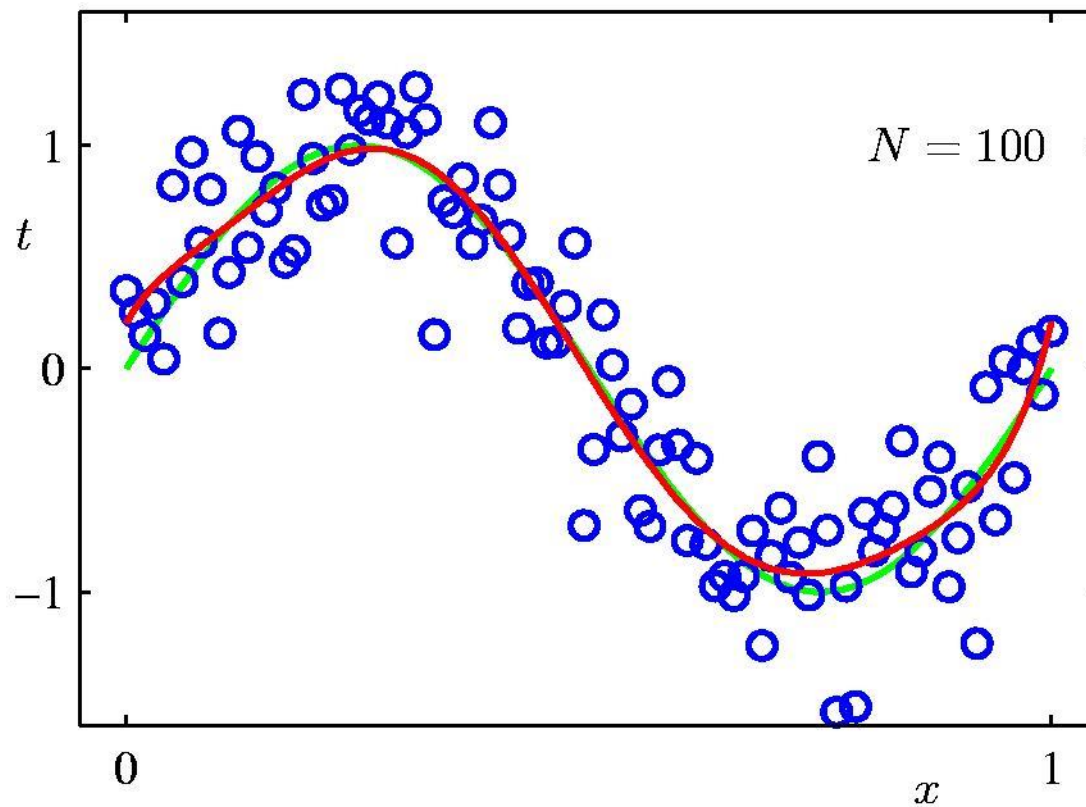




# Data Set Size: $N = 100$

---

9<sup>th</sup> Order Polynomial



# Regularization

---

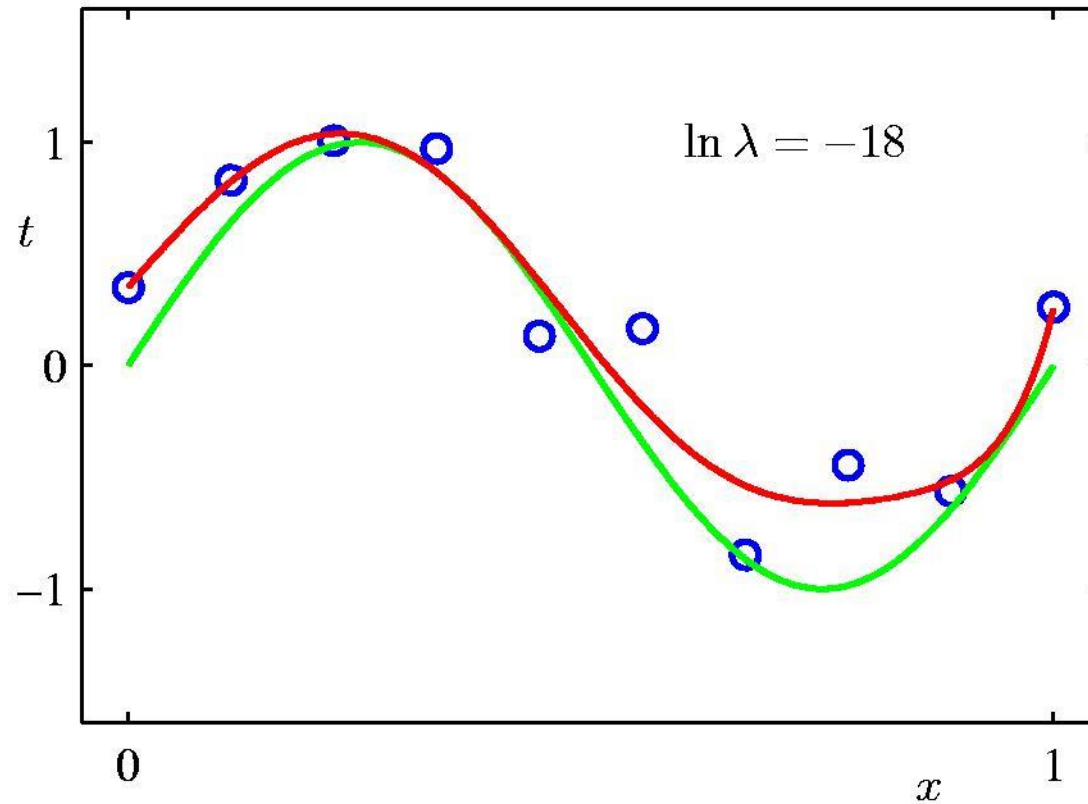
Penalize large coefficient values

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, \mathbf{w}) - t_n\}^2$$

(Remember: We want to minimize this expression.)

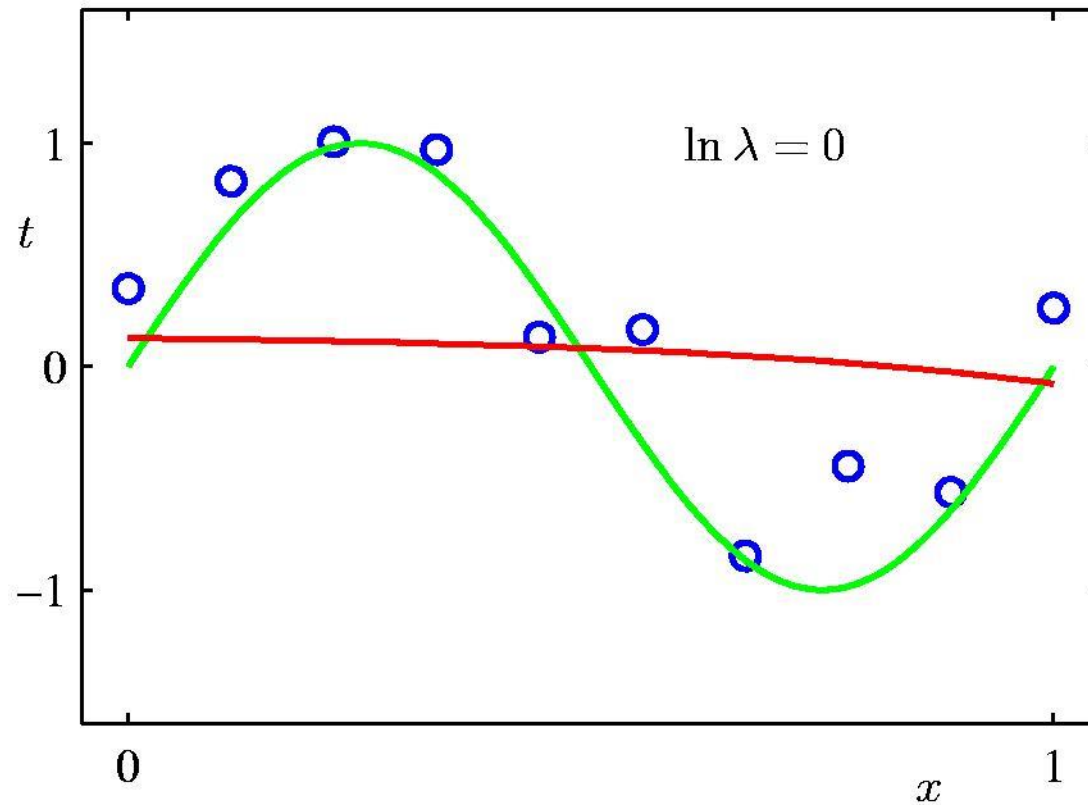
# Regularization: $\ln \lambda = -18$

---



# Regularization: $\ln \lambda = 0$

---



# Polynomial Coefficients

---

	$M = 0$	$M = 1$	$M = 3$	$M = 9$
$w_0^*$	0.19	0.82	0.31	0.35
$w_1^*$		-1.27	7.99	232.37
$w_2^*$			-25.43	-5321.83
$w_3^*$			17.37	48568.31
$w_4^*$				-231639.30
$w_5^*$				640042.26
$w_6^*$				-1061800.52
$w_7^*$				1042400.18
$w_8^*$				-557682.99
$w_9^*$				125201.43

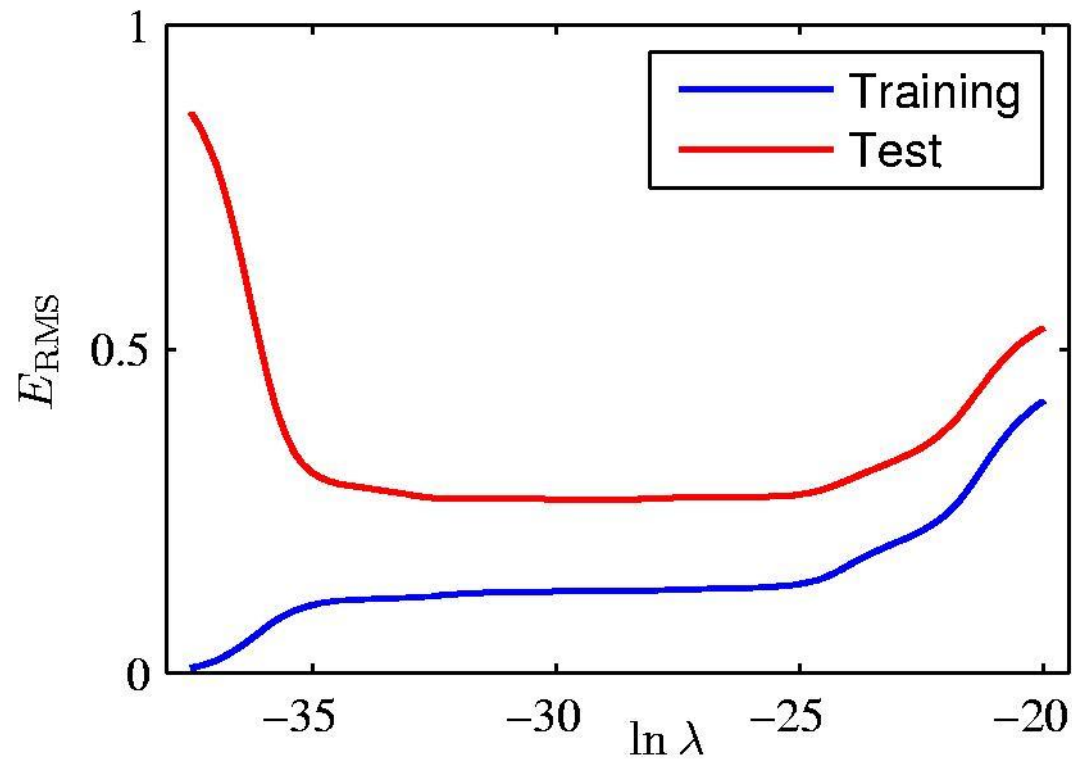
# Polynomial Coefficients

---

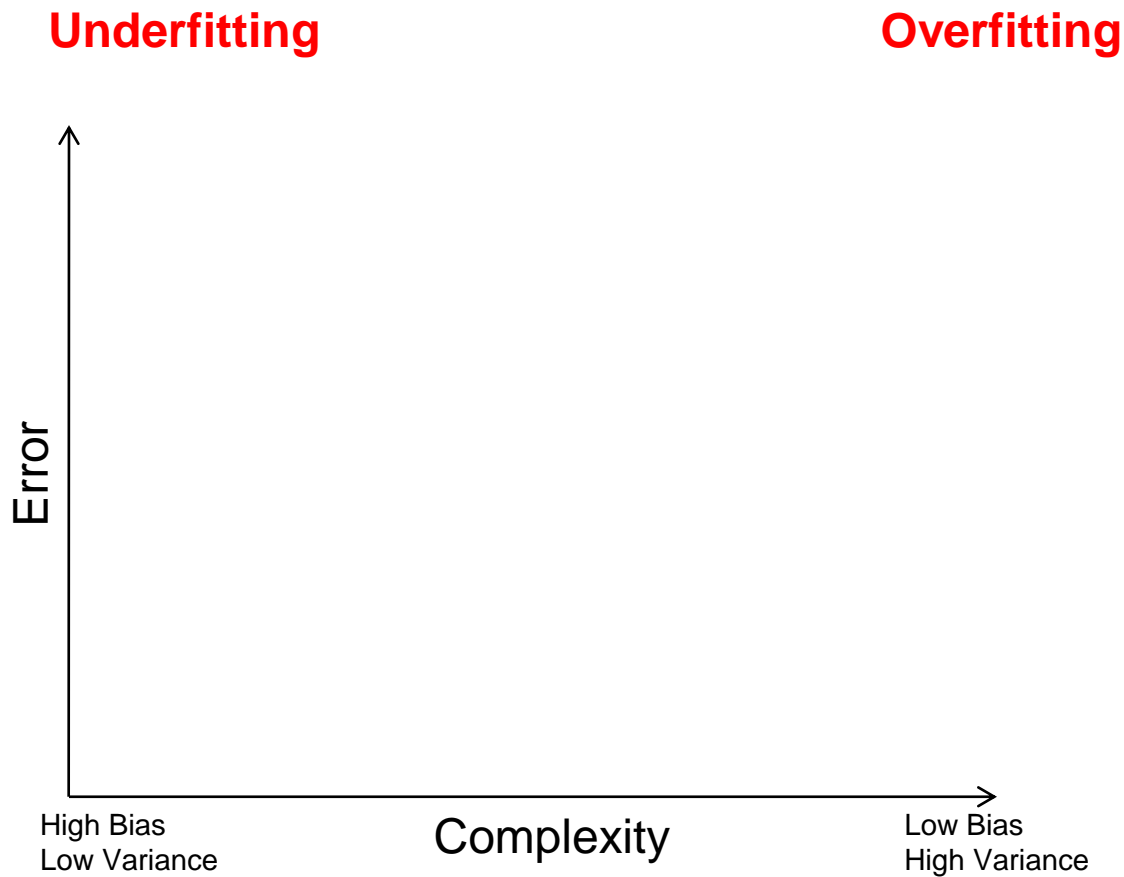
	No regularization		Huge regularization
	$\ln \lambda = -\infty$	$\ln \lambda = -18$	$\ln \lambda = 0$
$w_0^*$	0.35	0.35	0.13
$w_1^*$	232.37	4.74	-0.05
$w_2^*$	-5321.83	-0.77	-0.06
$w_3^*$	48568.31	-31.97	-0.05
$w_4^*$	-231639.30	-3.89	-0.03
$w_5^*$	640042.26	55.28	-0.02
$w_6^*$	-1061800.52	41.32	-0.01
$w_7^*$	1042400.18	-45.95	-0.00
$w_8^*$	-557682.99	-91.53	0.00
$w_9^*$	125201.43	72.68	0.01

# Regularization: $E_{\text{RMS}}$ vs. $\ln \lambda$

---

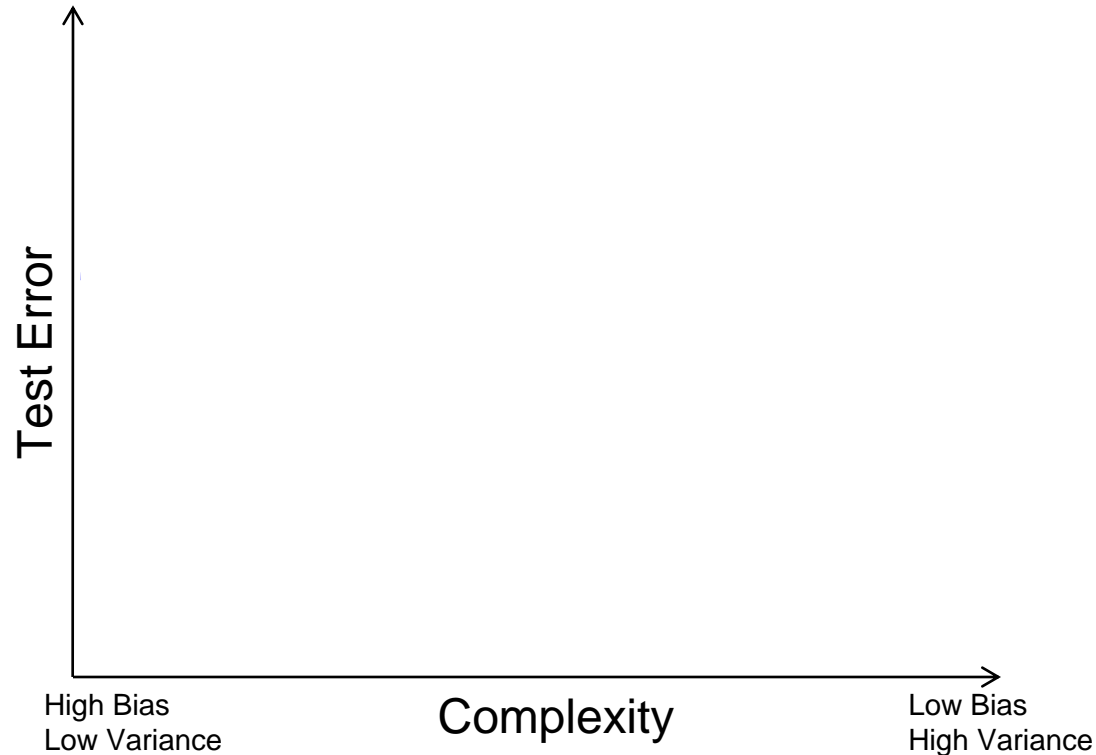


# Training vs test error



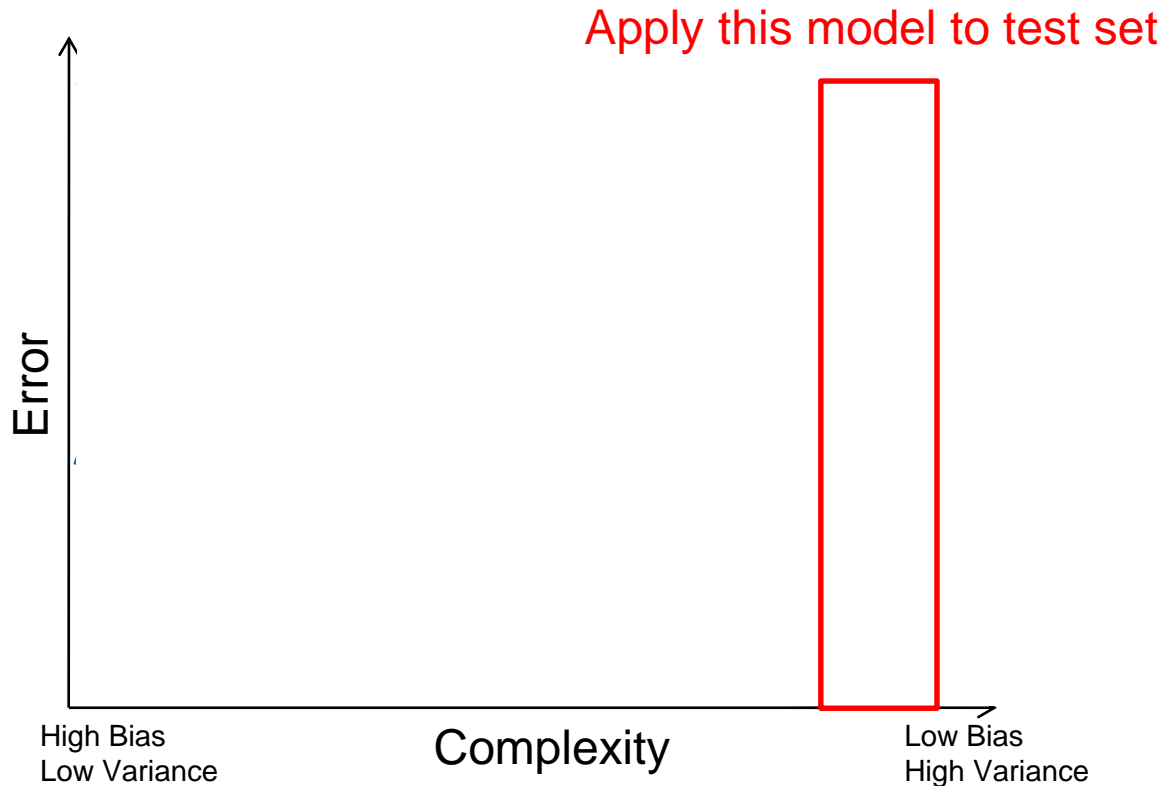


# The effect of training set size



# Choosing the trade-off between bias and variance

- Need validation set (separate from the test set)



# Summary of generalization

- Try simple classifiers first
- Better to have smart features and simple classifiers than simple features and smart classifiers
- Use increasingly powerful classifiers with more training data
- As an additional technique for reducing variance, try regularizing the parameters

# Plan for the rest of the lecture

---

## Neural network basics

- Definition
- Loss functions
- Optimization w/ gradient descent and backpropagation

## Convolutional neural networks (CNNs)

- Special operations
- Common architectures

## Practical matters

- Getting started: Preprocessing, initialization, optimization, normalization
- Improving performance: regularization, augmentation, transfer
- Hardware and software

## Understanding CNNs

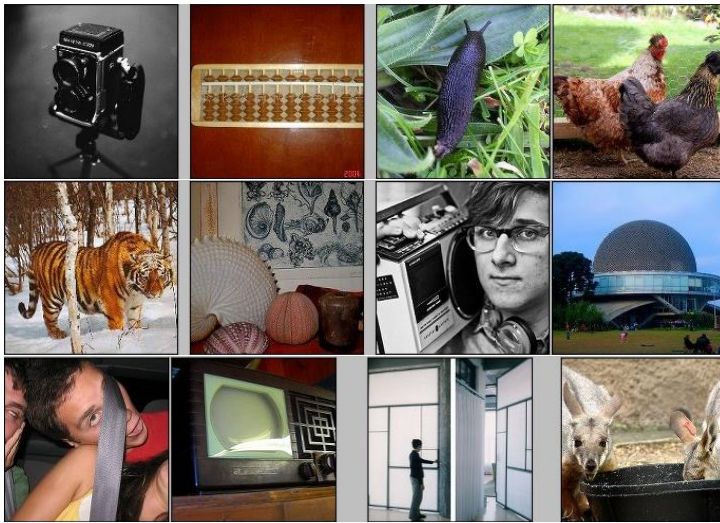
- Visualization
- Breaking CNNs

# Neural network basics

# ImageNet Challenge 2012

---

IMAGENET



[Deng et al. CVPR 2009]

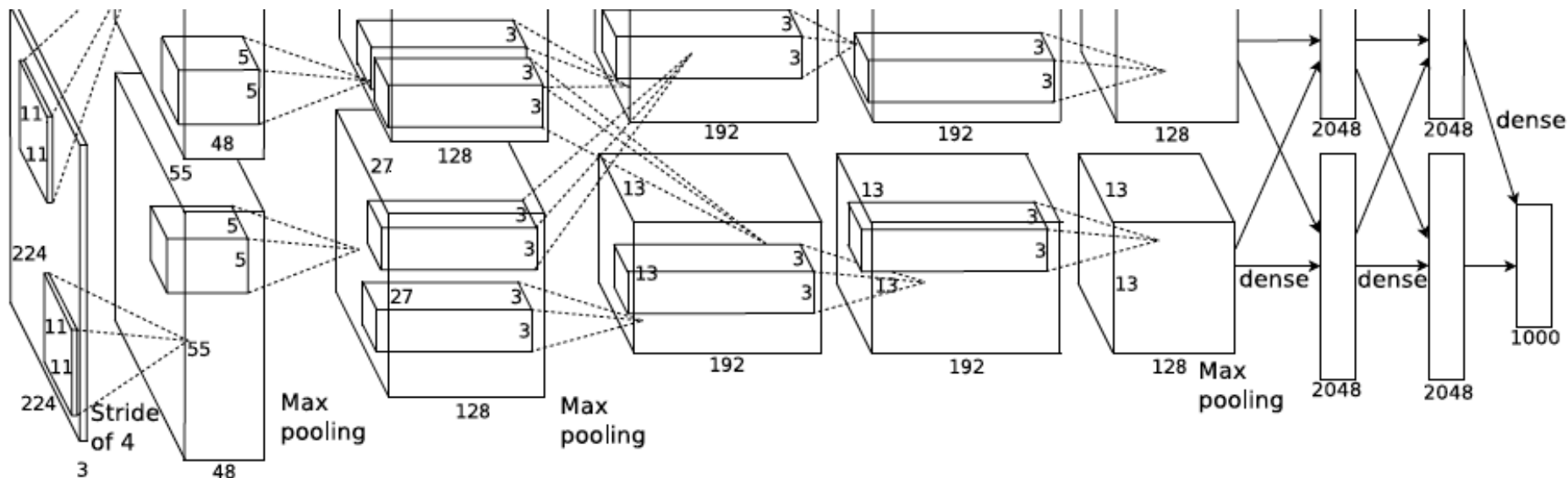
- ~14 million labeled images, 20k classes
- Images gathered from Internet
- Human labels via Amazon Turk
- Challenge: 1.2 million training images, 1000 classes

A. Krizhevsky, I. Sutskever, and G. Hinton, [ImageNet Classification with Deep Convolutional Neural Networks](#), NIPS 2012

# ImageNet Challenge 2012



- AlexNet: Similar framework to LeCun'98 but:
  - Bigger model (7 hidden layers, 650,000 units, 60,000,000 params)
  - More data ( $10^6$  vs.  $10^3$  images)
  - GPU implementation (50x speedup over CPU)
    - Trained on two GPUs for a week
  - Better regularization for training (DropOut)



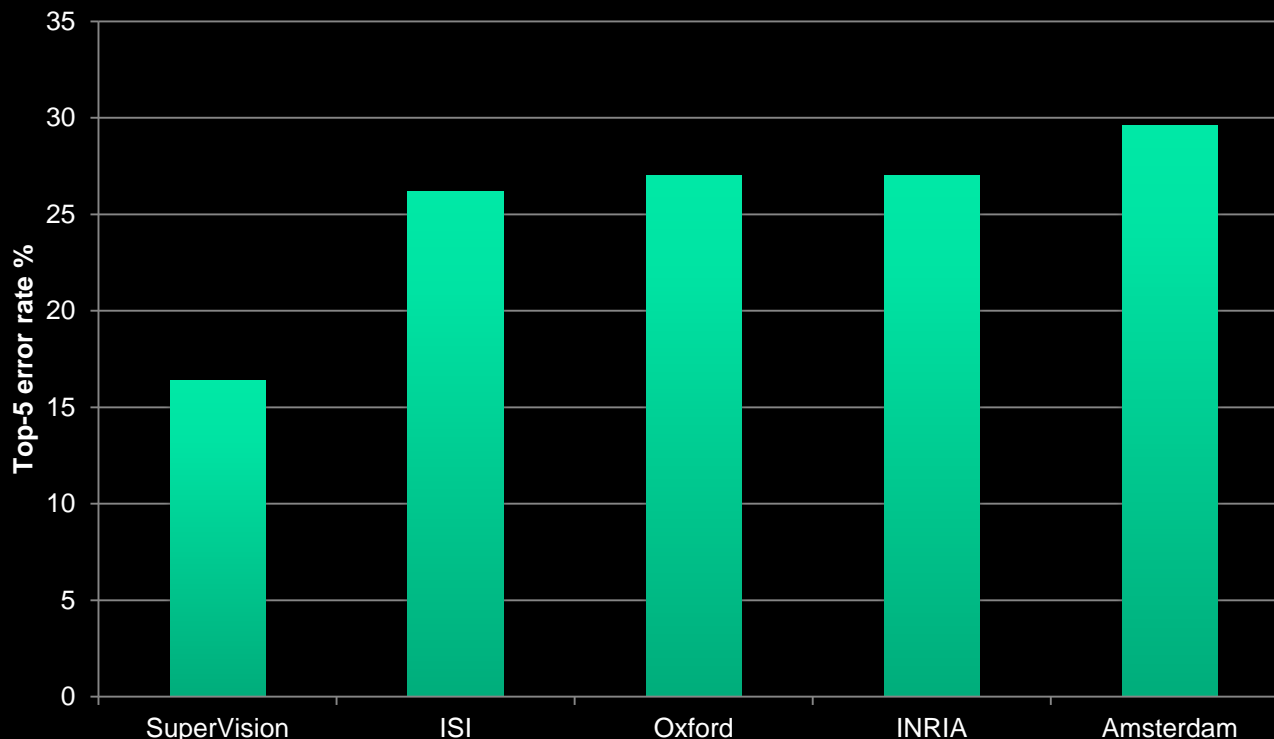
A. Krizhevsky, I. Sutskever, and G. Hinton, [ImageNet Classification with Deep Convolutional Neural Networks](#), NIPS 2012

# ImageNet Challenge 2012

---

Krizhevsky et al. -- **16.4% error** (top-5)

Next best (non-convnet) – **26.2% error**

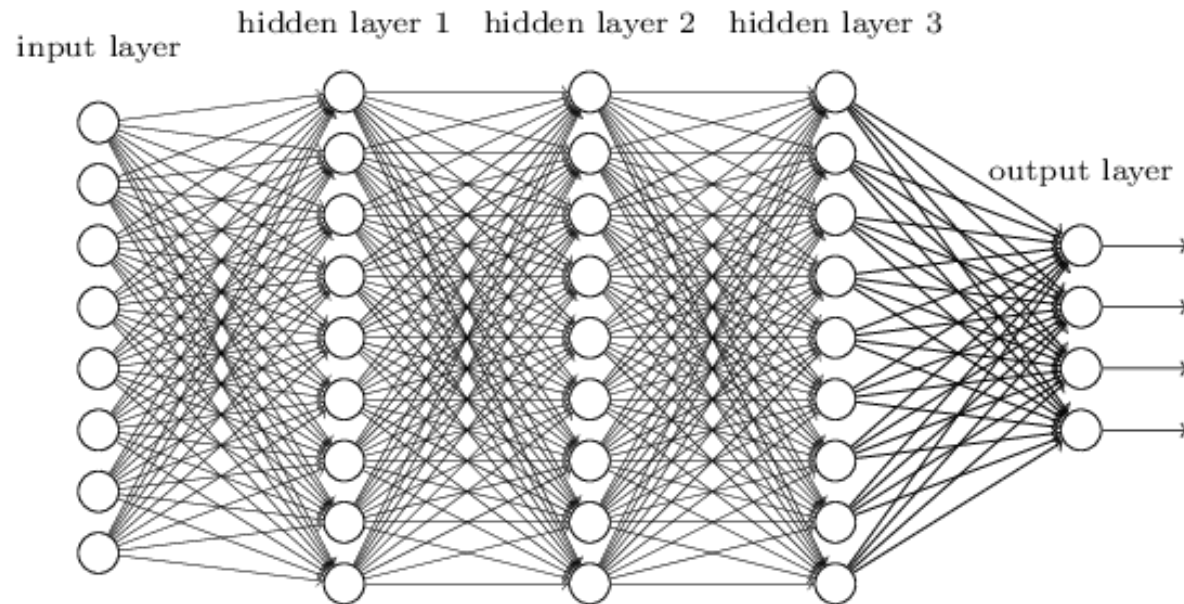




# What are CNNs?

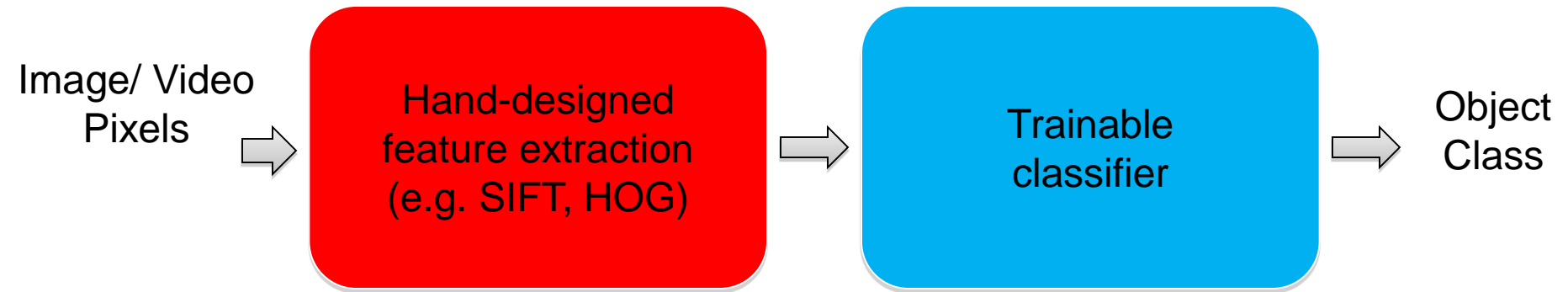
---

- Convolutional neural networks are a type of *neural network* with layers that perform special operations
- Used in vision but also in NLP, biomedical etc.
- Often they are *deep*



# Traditional Recognition Approach

---



- Features are key to recent progress in recognition, but research shows they're flawed...
- Where next?

# What about learning the features?

---

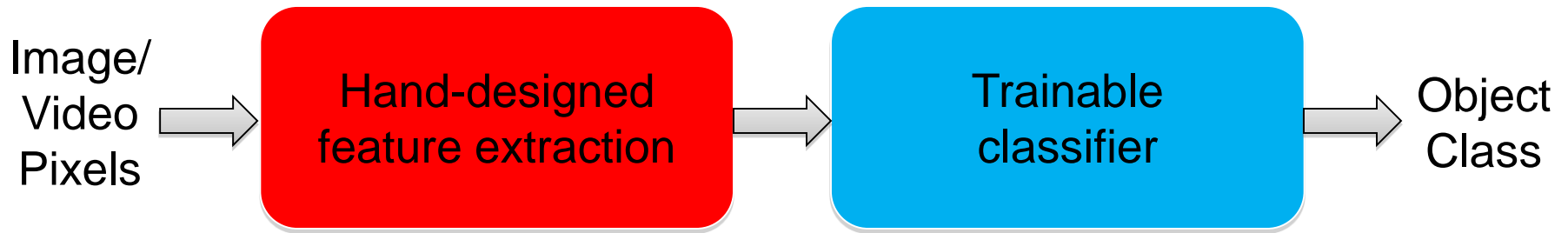
- Learn a *feature hierarchy* all the way from pixels to classifier
- Each layer extracts features from the output of previous layer
- Train all layers jointly



# “Shallow” vs. “deep” architectures

---

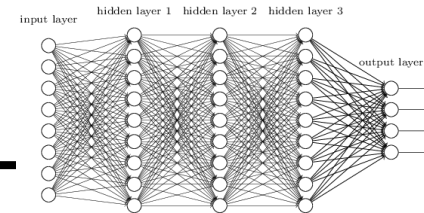
Traditional recognition: “Shallow” architecture



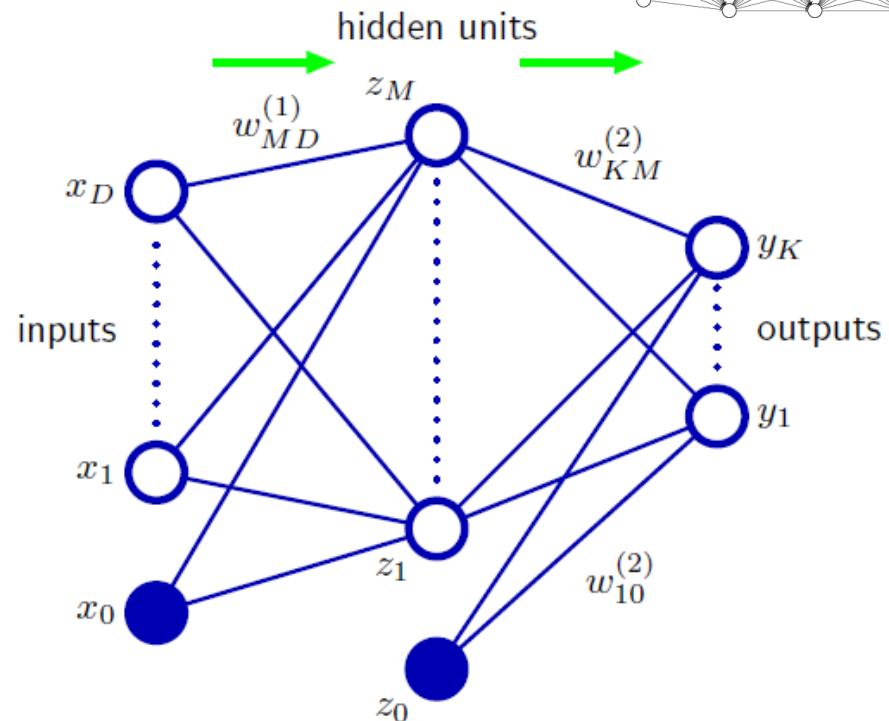
Deep learning: “Deep” architecture



# Neural network definition



**Figure 5.1** Network diagram for the two-layer neural network corresponding to (5.7). The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes, in which the bias parameters are denoted by links coming from additional input and hidden variables  $x_0$  and  $z_0$ . Arrows denote the direction of information flow through the network during forward propagation.



- Activations: 
$$a_j = \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$
- Nonlinear activation function  $h$  (e.g. sigmoid, RELU): 
$$z_j = h(a_j)$$

Recall SVM:  
 $w^T x + b$

# Neural network definition

- Layer 2

$$a_j = \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

- Layer 3 (final)

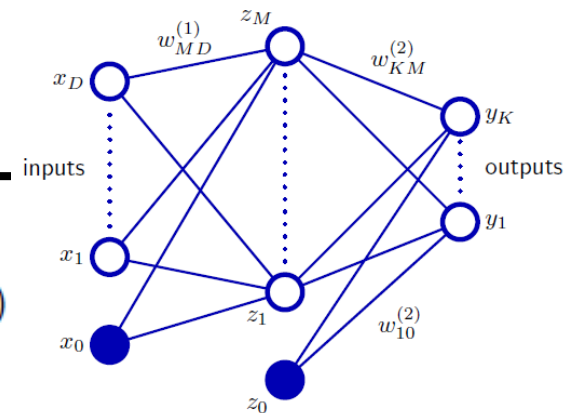
$$a_k =$$

- Outputs (e.g. sigmoid/softmax)

$$\begin{array}{ll} \text{(binary)} & y_k = \sigma(a_k) = \frac{1}{1 + \exp(-a_k)} \\ & \text{(multiclass)} & y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \end{array}$$

- Finally:

$$\text{(binary)} \quad y_k(\mathbf{X}, \mathbf{W}) = \sigma \left( \sum_{j=1}^M w_{kj}^{(2)} h \left( \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right)$$

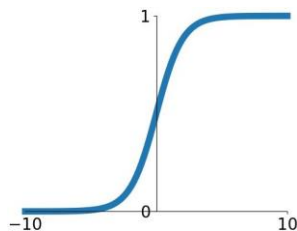


# Activation functions

---

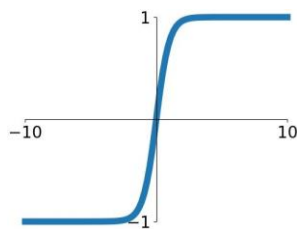
## Sigmoid

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



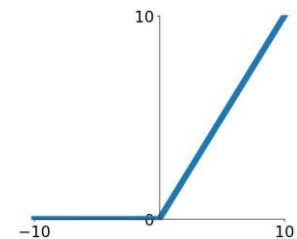
## tanh

$$\tanh(x)$$



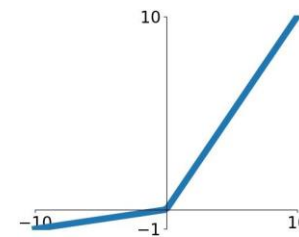
## ReLU

$$\max(0, x)$$



## Leaky ReLU

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

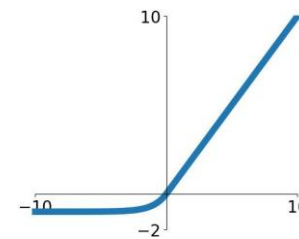


## Maxout

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

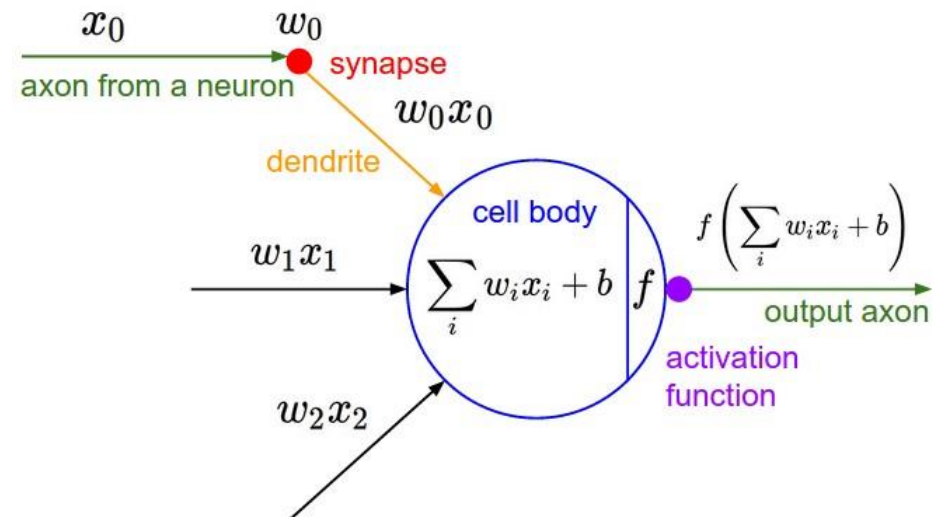
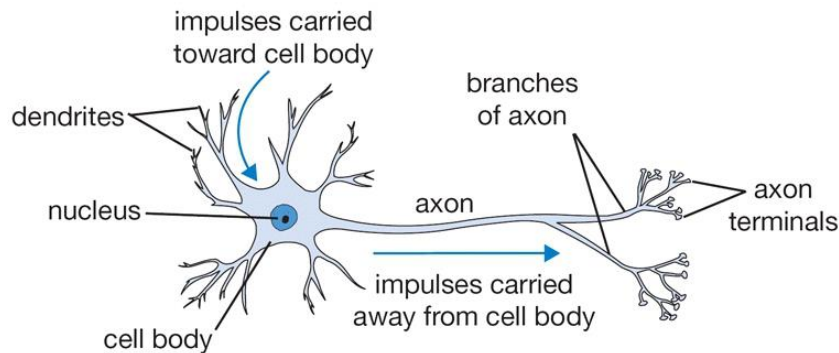
## ELU

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



# Inspiration: Neuron cells

- Neurons
  - accept information from multiple inputs,
  - transmit information to other neurons.
- Multiply inputs by weights along edges
- Apply some function to the set of inputs at each node
- If output of function over threshold, neuron “fires”

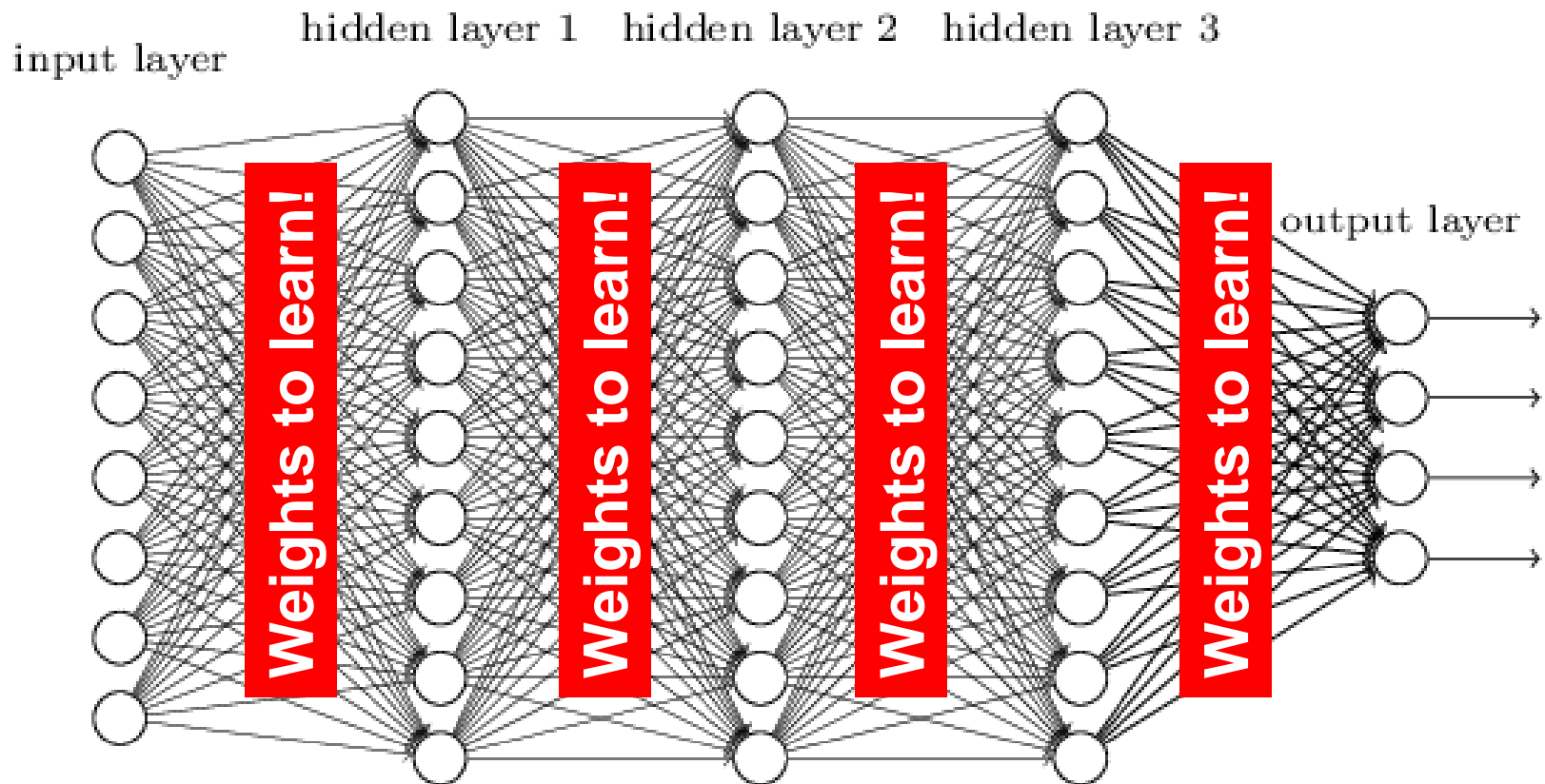




# Deep neural networks

---

- Lots of hidden layers
- Depth = power (usually)



# How do we train them?

---

- The goal is to iteratively find such a set of weights that allow the activations/outputs to match the desired output
- We want to *minimize a **loss function***
- The loss function is a function of the weights in the network
- For now let's simplify and assume there's a single layer of weights in the network

# Classification goal

---

airplane



automobile



bird



cat



deer



dog



frog



horse



ship



truck



Example dataset: **CIFAR-10**

**10** labels

**50,000** training images

each image is **32x32x3**

**10,000** test images.

# Classification scores

---

$$f(x, W) = Wx$$



$$f(\mathbf{x}, \mathbf{W})$$

→

**10** numbers,  
indicating class  
scores

**[32x32x3]**

array of numbers 0...1  
(3072 numbers total)

# Linear classifier

---



**[32x32x3]**

array of numbers 0...1

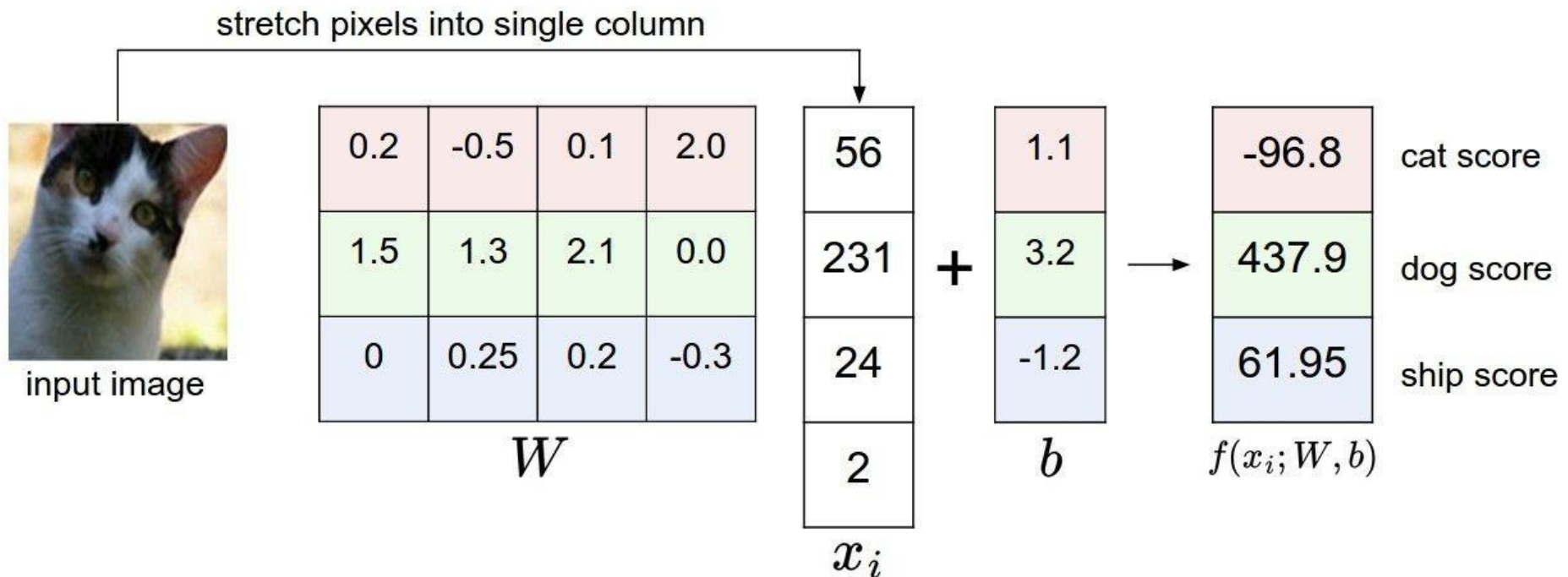
$$\boxed{f(x, W)}_{10 \times 1} = \boxed{W}_{10 \times 3072} \boxed{x}_{3072 \times 1} \boxed{(+b)}_{10 \times 1}$$

**10** numbers,  
indicating class  
scores

parameters, or “weights”

# Linear classifier

Example with an image with 4 pixels, and 3 classes (cat/dog/ship)



# Linear classifier

---

Going forward: Loss function/Optimization



cat	<b>3.2</b>	1.3	2.2
car	5.1	<b>4.9</b>	2.5
frog	-1.7	2.0	<b>-3.1</b>

## TODO:

1. Define a **loss function** that quantifies our unhappiness with the scores across the training data.
2. Come up with a way of efficiently finding the parameters that minimize the loss function.  
**(optimization)**

# Linear classifier

---

Suppose: 3 training examples, 3 classes.

With some  $W$  the scores  $f(x, W) = Wx$  are:



cat	<b>3.2</b>	1.3	2.2
car	5.1	<b>4.9</b>	2.5
frog	-1.7	2.0	<b>-3.1</b>



# Linear classifier: Hinge loss

Suppose: 3 training examples, 3 classes.  
With some  $W$  the scores  $f(x, W) = Wx$  are:



cat	<b>3.2</b>	1.3	2.2
car	5.1	<b>4.9</b>	2.5
frog	-1.7	2.0	<b>-3.1</b>

## Hinge loss:

Given an example  $(x_i, y_i)$   
where  $x_i$  is the image and  
where  $y_i$  is the (integer) label,

and using the shorthand for the  
scores vector:  $s = f(x_i, W)$

the loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Want:  $s_{y_i} \geq s_j + 1$   
i.e.  $s_j - s_{y_i} + 1 \leq 0$

If true, loss is 0

If false, loss is magnitude of violation

# Linear classifier: Hinge loss

Suppose: 3 training examples, 3 classes.  
With some  $W$  the scores  $f(x, W) = Wx$  are:



cat	<b>3.2</b>	1.3	2.2
car	5.1	<b>4.9</b>	2.5
frog	-1.7	2.0	<b>-3.1</b>
Losses:	<b>2.9</b>		

## Hinge loss:

Given an example  $(x_i, y_i)$   
where  $x_i$  is the image and  
where  $y_i$  is the (integer) label,

and using the shorthand for the  
scores vector:  $s = f(x_i, W)$

the loss has the form:

$$\begin{aligned} L_i &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \\ &= \max(0, 5.1 - 3.2 + 1) \\ &\quad + \max(0, -1.7 - 3.2 + 1) \\ &= \max(0, 2.9) + \max(0, -3.9) \\ &= 2.9 + 0 \\ &= 2.9 \end{aligned}$$

# Linear classifier: Hinge loss

Suppose: 3 training examples, 3 classes.  
With some  $W$  the scores  $f(x, W) = Wx$  are:



cat	<b>3.2</b>	1.3	2.2
car	5.1	<b>4.9</b>	2.5
frog	-1.7	2.0	<b>-3.1</b>
Losses:	2.9	0	

## Hinge loss:

Given an example  $(x_i, y_i)$   
where  $x_i$  is the image and  
where  $y_i$  is the (integer) label,

and using the shorthand for the  
scores vector:  $s = f(x_i, W)$

the loss has the form:

$$\begin{aligned} L_i &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \\ &= \max(0, 1.3 - 4.9 + 1) \\ &\quad + \max(0, 2.0 - 4.9 + 1) \\ &= \max(0, -2.6) + \max(0, -1.9) \\ &= 0 + 0 \\ &= 0 \end{aligned}$$

# Linear classifier: Hinge loss

Suppose: 3 training examples, 3 classes.  
With some  $W$  the scores  $f(x, W) = Wx$  are:



cat	<b>3.2</b>	1.3	2.2
car	5.1	<b>4.9</b>	2.5
frog	-1.7	2.0	<b>-3.1</b>
Losses:	2.9	0	12.9

## Hinge loss:

Given an example  $(x_i, y_i)$   
where  $x_i$  is the image and  
where  $y_i$  is the (integer) label,

and using the shorthand for the  
scores vector:  $s = f(x_i, W)$

the loss has the form:

$$\begin{aligned} L_i &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \\ &= \max(0, 2.2 - (-3.1) + 1) \\ &\quad + \max(0, 2.5 - (-3.1) + 1) \\ &= \max(0, 5.3 + 1) \\ &\quad + \max(0, 5.6 + 1) \\ &= 6.3 + 6.6 \\ &= 12.9 \end{aligned}$$

# Linear classifier: Hinge loss

Suppose: 3 training examples, 3 classes.  
With some  $W$  the scores  $f(x, W) = Wx$  are:



cat	<b>3.2</b>	1.3	2.2
car	5.1	<b>4.9</b>	2.5
frog	-1.7	2.0	<b>-3.1</b>
Losses:	<b>2.9</b>	<b>0</b>	<b>12.9</b>

## Hinge loss:

Given an example  $(x_i, y_i)$   
where  $x_i$  is the image and  
where  $y_i$  is the (integer) label,

and using the shorthand for the  
scores vector:  $s = f(x_i, W)$

the loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

and the full training loss is the mean  
over all examples in the training data:

$$L = \frac{1}{N} \sum_{i=1}^N L_i$$

$$L = (2.9 + 0 + 12.9) / 3 \\ = 15.8 / 3 = \mathbf{5.3}$$

# Linear classifier: Hinge loss

---

$$f(x, W) = Wx$$

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

# Linear classifier: Hinge loss

---

## Weight Regularization

$\lambda$  = regularization strength  
(hyperparameter)

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

In common use:

**L2 regularization**

L1 regularization

Dropout (will see later)

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

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

# Another loss: Softmax (cross-entropy)

---



cat	<b>3.2</b>
car	<b>5.1</b>
frog	<b>-1.7</b>

**scores = unnormalized log probabilities of the classes.**

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad \text{where} \quad s = f(x_i; W)$$

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

$$L_i = -\log P(Y = y_i|X = x_i)$$



# Another loss: Cross-entropy



$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

cat  
car  
frog

3.2  
5.1  
-1.7

exp

Probabilities  
must be  $\geq 0$

24.5  
164.0  
0.18

normalize

Probabilities  
must sum to 1

0.13  
0.87  
0.00

$$L_i = -\log(0.13) = 0.89$$

unnormalized log probabilities

probabilities

unnormalized probabilities

**Aside:**

- This is multinomial logistic regression
- Choose weights to maximize the likelihood of the observed x/y data (Maximum Likelihood Estimation)

# Another loss: Cross-entropy



$$s = f(x_i; W)$$

Probabilities  
must be  $\geq 0$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Probabilities  
must sum to 1

$$L_i = -\log P(Y = y_i|X = x_i)$$

cat  
car  
frog

**3.2**  
**5.1**  
**-1.7**

unnormalized  
log-probabilities / logits

exp

**24.5**  
**164.0**  
**0.18**

unnormalized  
probabilities

normalize

**0.13**  
**0.87**  
**0.00**

probabilities

compare

Kullback–Leibler  
divergence

$$D_{KL}(P||Q) = \sum_y P(y) \log \frac{P(y)}{Q(y)}$$

**1.00**  
**0.00**  
**0.00**

correct  
probs

# Other losses

- Triplet loss (Schroff, FaceNet, CVPR 2015)

$$\sum_i^N \left[ \|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + \alpha \right]_+$$

a denotes anchor  
p denotes positive  
n denotes negative

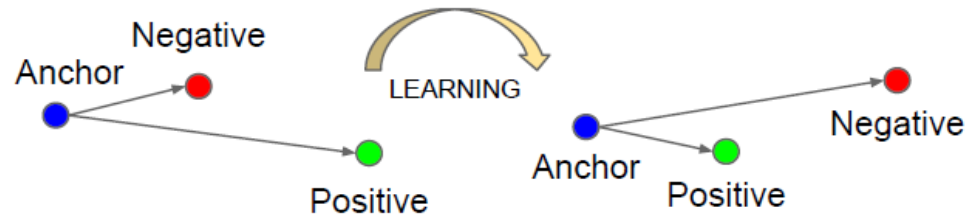


Figure 3. The **Triplet Loss** minimizes the distance between an *anchor* and a *positive*, both of which have the same identity, and maximizes the distance between the *anchor* and a *negative* of a different identity.

- Anything you want!

# How to minimize the loss function?

---



# How to minimize the loss function?

---

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives).

**current W:**

[0.34,  
-1.11,  
0.78,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

**loss 1.25347**

$dW = \dots$   
(some function  
data and W)



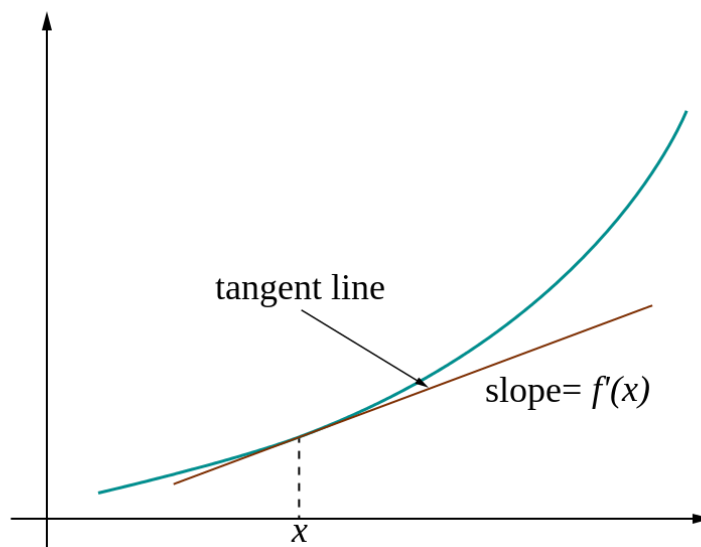
**gradient dW:**

[-2.5,  
0.6,  
0,  
0.2,  
0.7,  
-0.5,  
1.1,  
1.3,  
-2.1,...]

# Loss gradients

---

- Denoted as (diff notations):  $\frac{\partial E}{\partial w_{ji}^{(1)}} \quad \nabla_w L$
- i.e. how does the loss change as a function of the weights
- We want to change the weights in such a way that makes the loss decrease as fast as possible

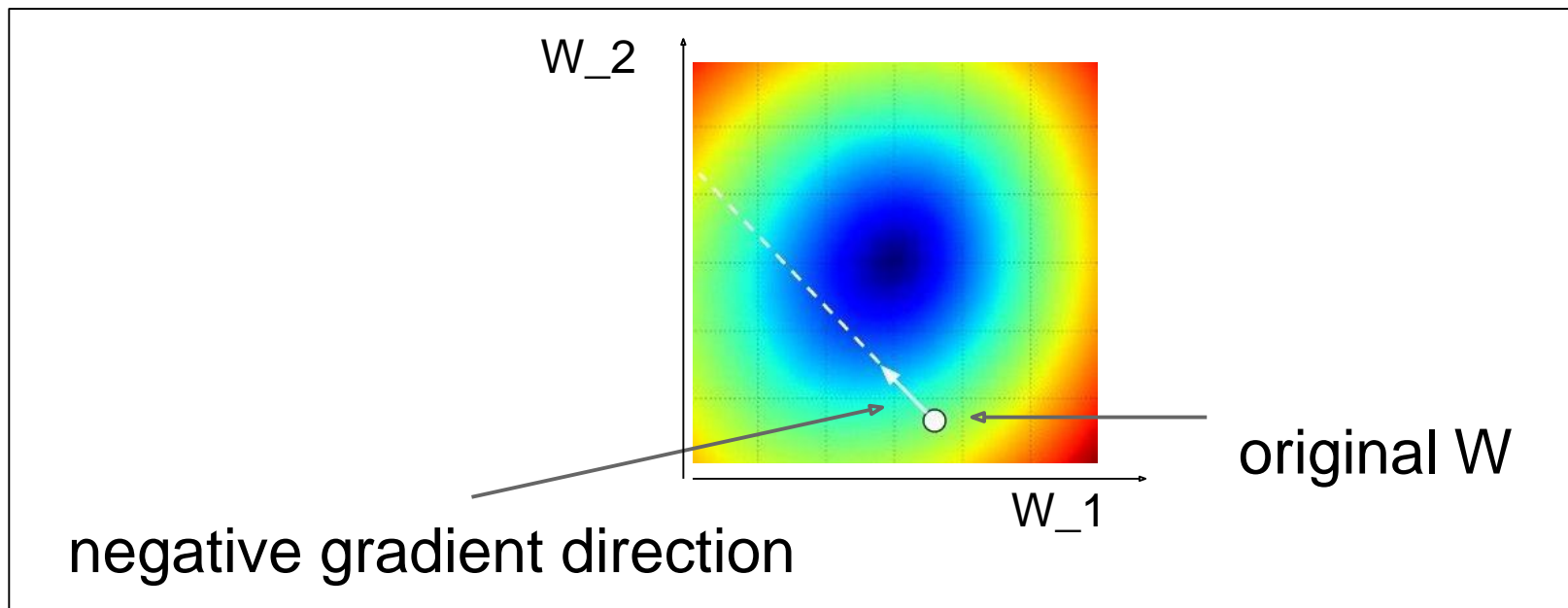


# Gradient descent

---

- We'll update weights
- Move in direction opposite to gradient:

$$\underset{\substack{\uparrow \\ \text{Time}}}{\mathbf{w}^{(\tau+1)}} = \mathbf{w}^{(\tau)} - \underset{\substack{\uparrow \\ \text{Learning rate}}}{\eta} \overbrace{\nabla E(\mathbf{w}^{(\tau)})}$$





# Gradient descent

---

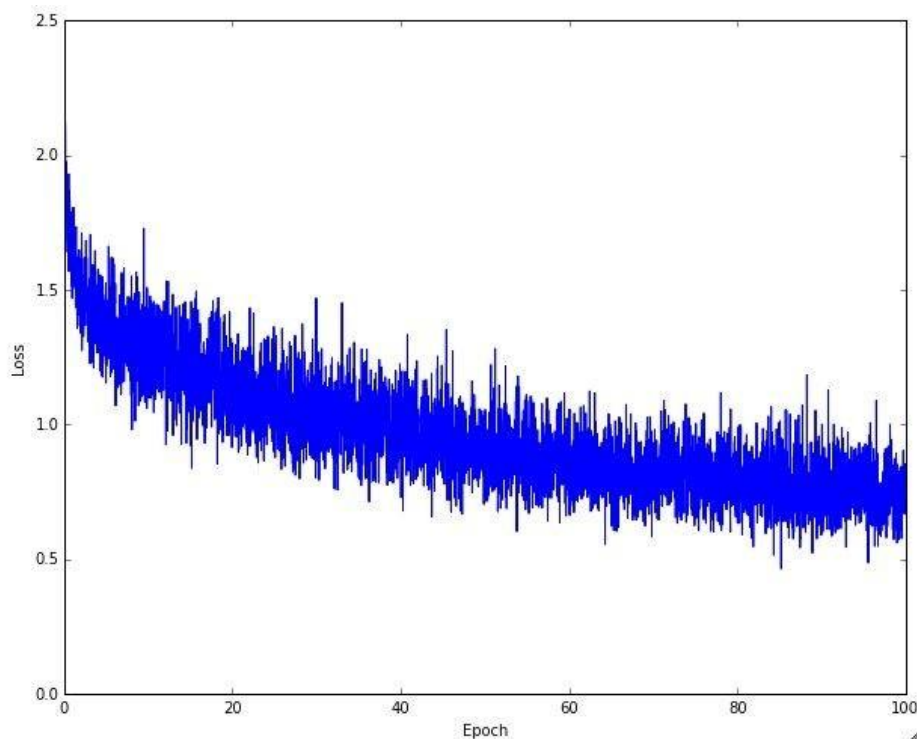
- Iteratively *subtract* the gradient with respect to the model parameters ( $w$ )
- I.e. we're moving in a direction opposite to the gradient of the loss
- I.e. we're moving towards *smaller* loss

# Mini-batch gradient descent

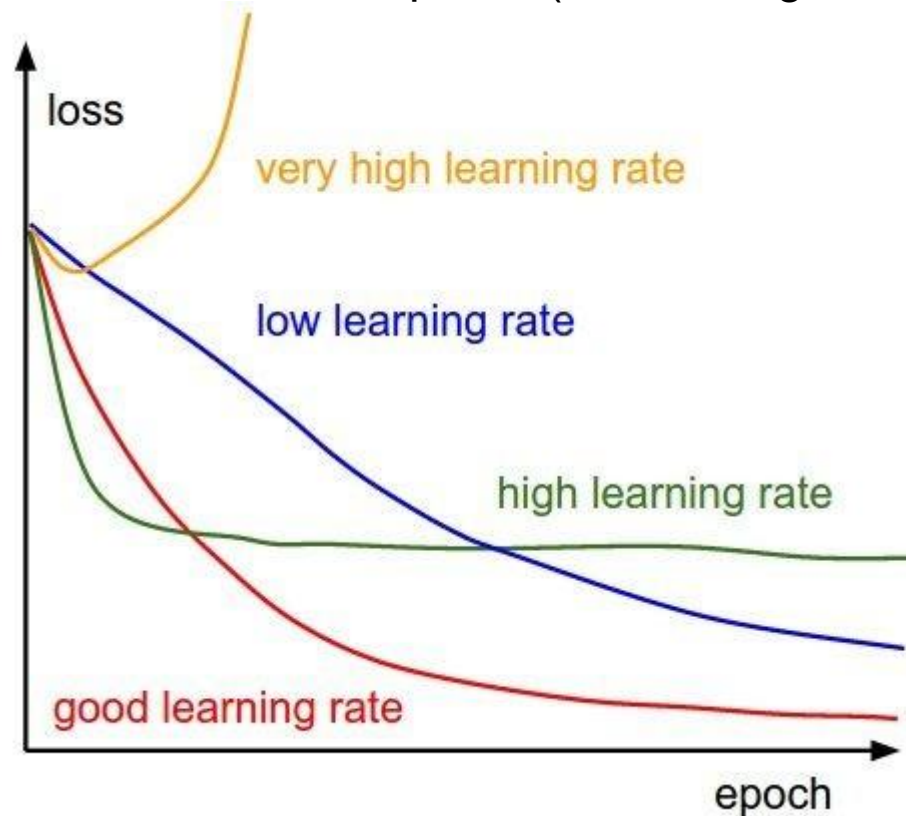
---

- In classic gradient descent, we compute the gradient from the loss for all training examples
- Could also only use *some* of the data for each gradient update
- We cycle through all the training examples multiple times
- Each time we've cycled through all of them once is called an 'epoch'
- Allows faster training (e.g. on GPUs), parallelization

# Learning rate selection



The effects of step size (or “learning rate”)



# Gradient descent in multi-layer nets

---

- We'll update weights
- Move in direction opposite to gradient:

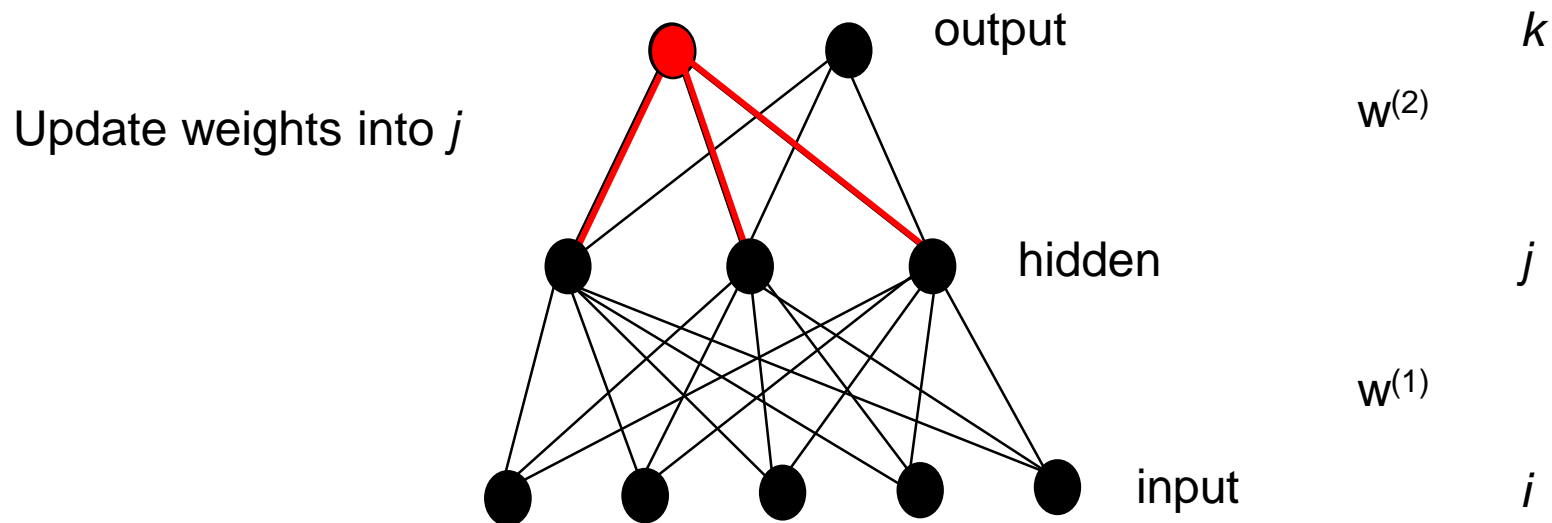
$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

- How to update the weights at all layers?
- Answer: backpropagation of error from higher layers to lower layers

# Backpropagation: Graphic example

---

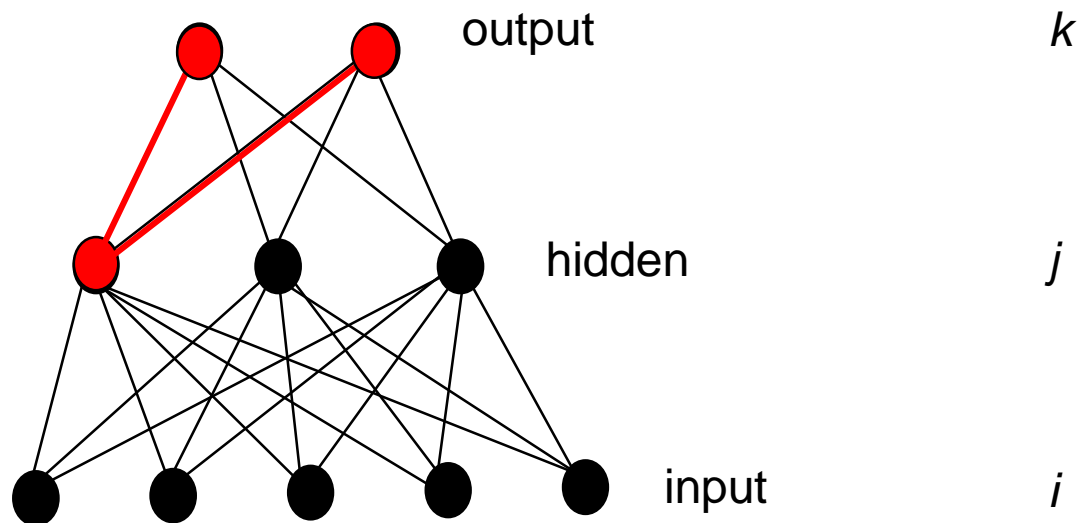
First calculate error of output units and use this to change the top layer of weights.



# Backpropagation: Graphic example

---

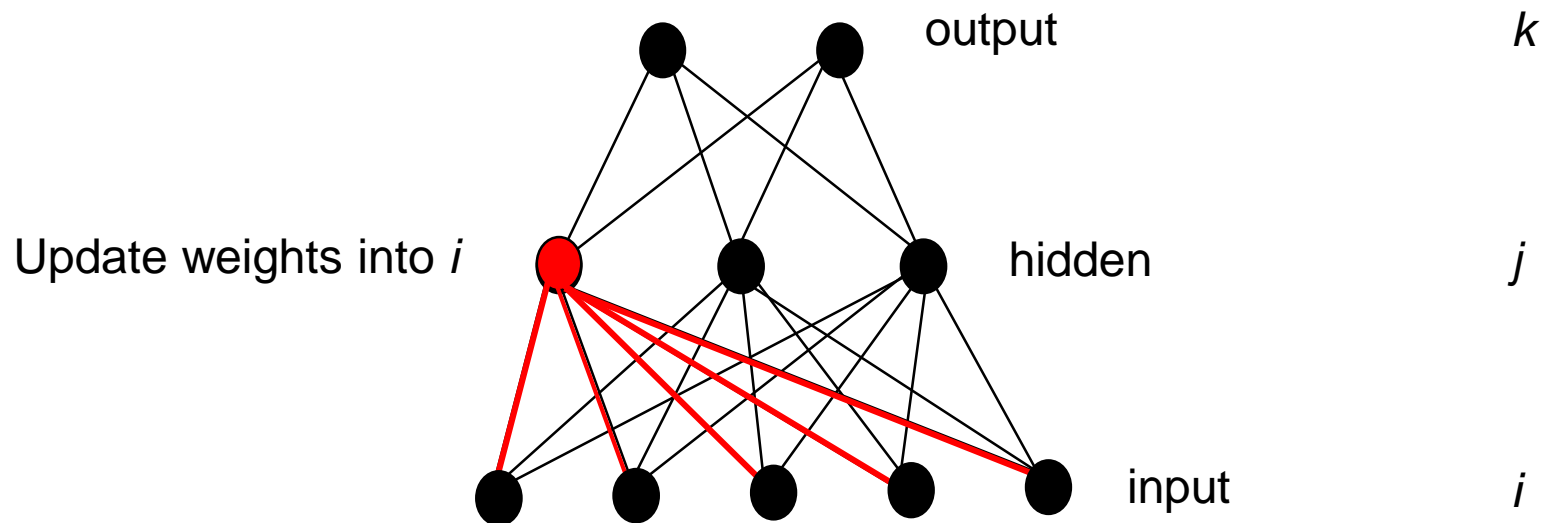
Next calculate error for hidden units based on errors on the output units it feeds into.



# Backpropagation: Graphic example

---

Finally update bottom layer of weights based on errors calculated for hidden units.



# Computing gradient for each weight

---

- We need to move weights in direction opposite to gradient of loss wrt that weight:

$$w_{kj} = w_{kj} - \eta \, dE/dw_{kj} \quad (\text{output layer})$$

$$w_{ji} = w_{ji} - \eta \, dE/dw_{ji} \quad (\text{hidden layer})$$

- Loss depends on weights in an indirect way, so we'll use the chain rule and compute:

$$dE/dw_{kj} = dE/dy_k \, dy_k/da_k \, da_k/dw_{kj}$$

$$dE/dw_{ji} = dE/dz_j \, dz_j/da_j \, da_j/dw_{ji}$$



# Gradient for output layer weights

---

- Loss depends on weights in an indirect way, so we'll use the chain rule and compute:

$$dE/dw_{kj} = dE/dy_k \quad dy_k/da_k \quad da_k/dw_{kj}$$

- How to compute each of these?
- $dE/dy_k$  : need to know form of error function
  - Example: if  $E = (y_k - y_k')^2$ , where  $y_k'$  is the ground-truth label, then  $dE/dy_k = 2(y_k - y_k')$
- $dy_k/da_k$  : need to know output layer activation
  - If  $h(a_k) = \sigma(a_k)$ , then  $dh(a_k)/da_k = \sigma(a_k)(1 - \sigma(a_k))$
- $da_k/dw_{kj}$  :  $z_j$  since  $a_k$  is a linear combination
  - $a_k = w_{k:}^T z = \sum_j w_{kj} z_j$

# Gradient for hidden layer weights

---

- We'll use the chain rule again and compute:

$$dE / dw_{ji} = dE / dz_j \quad dz_j / da_j \quad da_j / dw_{ji}$$

- Unlike the previous case (weights for output layer), the error ( $dE / dz_j$ ) is hard to compute (indirect, need chain rule again)
- We'll simplify the computation by doing it step by step via *backpropagation* of error
- You could directly compute this term— you will get the same result as with backprop (do as an exercise!)

# Gradients – slightly different notation

---

- *The following is a framework, slightly imprecise*
- Let us denote the inputs at a layer  $i$  by  $in_i$ , the linear combination of inputs computed at that layer as  $raw_i$ , the activation as  $act_i$
- We define a new quantity that will roughly correspond to accumulated error,  $err_i$
- Then we can write the updates as

$$w = w - \eta * err_i * in_i$$

- We can compute error as:

$$err_i = \frac{dE}{dact_i} * \frac{dact_i}{d raw_i}$$

# Gradients – slightly different approach

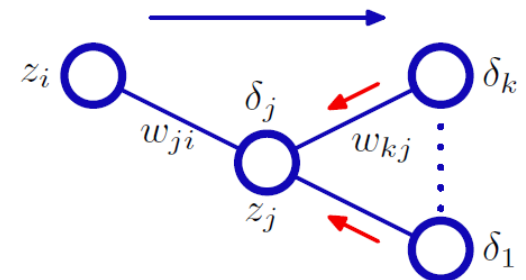
- We'll write the weight updates as follows

➤  $w_{kj} = w_{kj} - \eta \delta_k z_j$  for output units

➤  $w_{ji} = w_{ji} - \eta \delta_j x_i$  for hidden units

- What are  $\delta_k, \delta_j$ ?

- They store error, gradient wrt raw activations (i.e.  $dE/da$ )
- They're of the form  $dE/dz_j dz_j/da_j$
- The latter is easy to compute – just use derivative of activation function
- The former is easy for output – e.g.  $(y_k - y_k')$
- It is harder to compute for hidden layers
- $dE/dz_j = \sum_k w_{kj} \delta_k$  (see Bishop book Eq. 5.56)



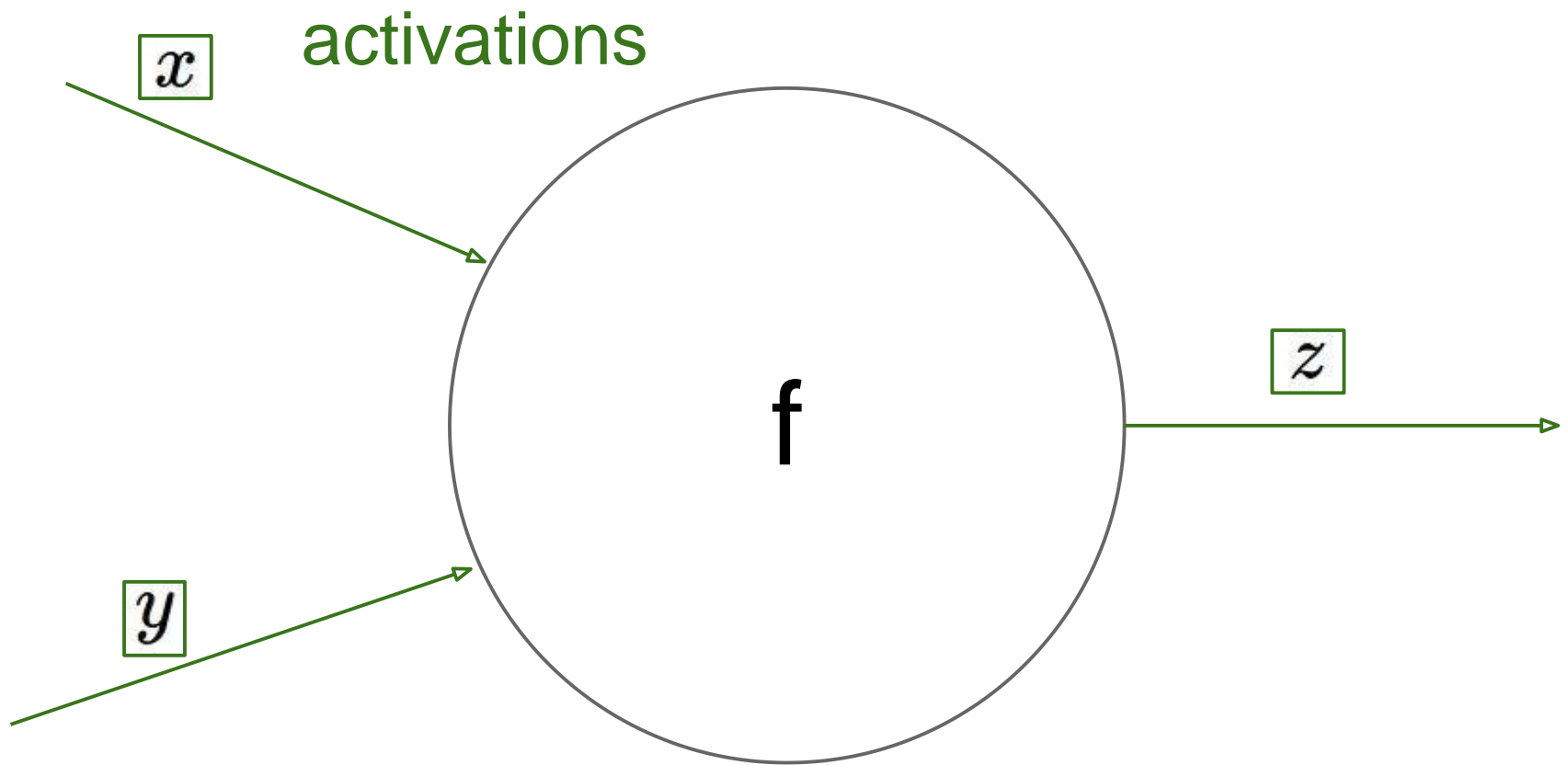
# Example algorithm for sigmoid, L2 error

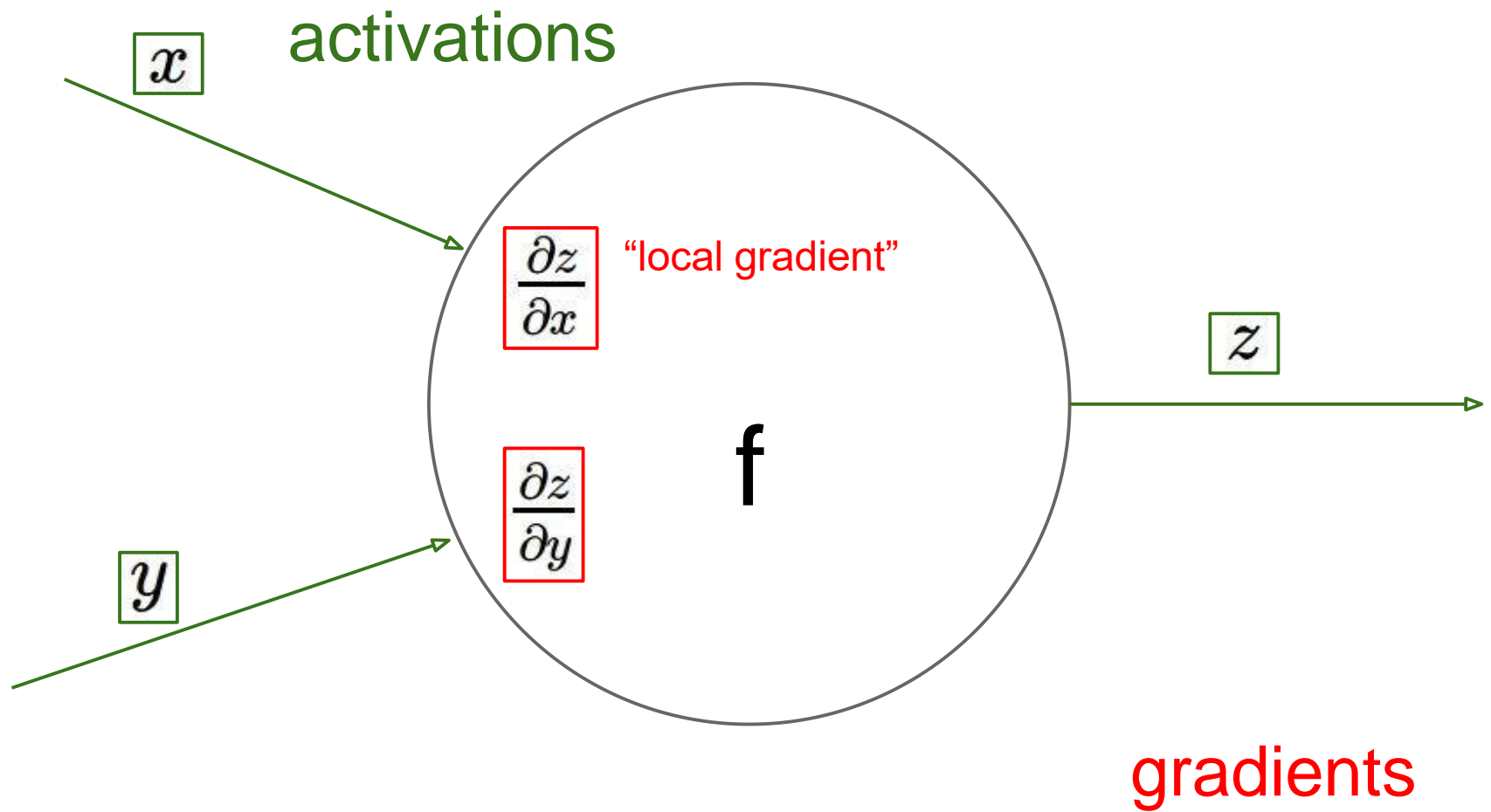
---

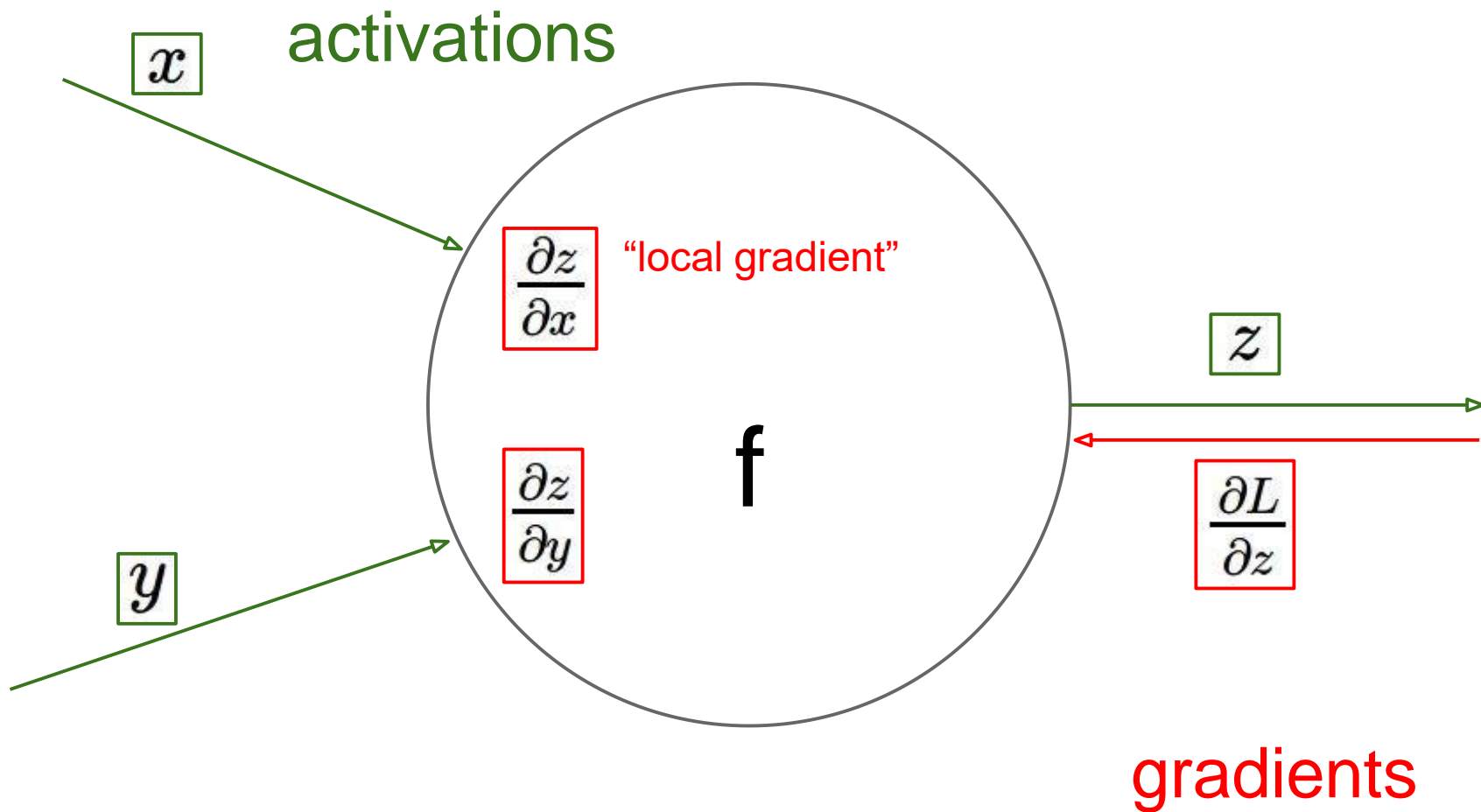
- Initialize all weights to small random values
- Until convergence (e.g. all training examples' error small, or error stops decreasing) repeat:
  - For each  $(\mathbf{x}, \mathbf{y}' = \text{class}(\mathbf{x}))$  in training set:
    - Calculate network outputs:  $y_k$
    - Compute errors (gradients wrt activations) for each unit:
      - »  $\delta_k = y_k (1 - y_k) (y_k - y'_k)$  for output units
      - »  $\delta_j = z_j (1 - z_j) \sum_k w_{kj} \delta_k$  for hidden units
    - Update weights:
      - »  $w_{kj} = w_{kj} - \eta \delta_k z_j$  for output units
      - »  $w_{ji} = w_{ji} - \eta \delta_j x_i$  for hidden units

Recall:  $w_{ji} = w_{ji} - \eta \frac{dE}{dz_j} \frac{dz_j}{da_j} \frac{da_j}{dw_{ji}}$

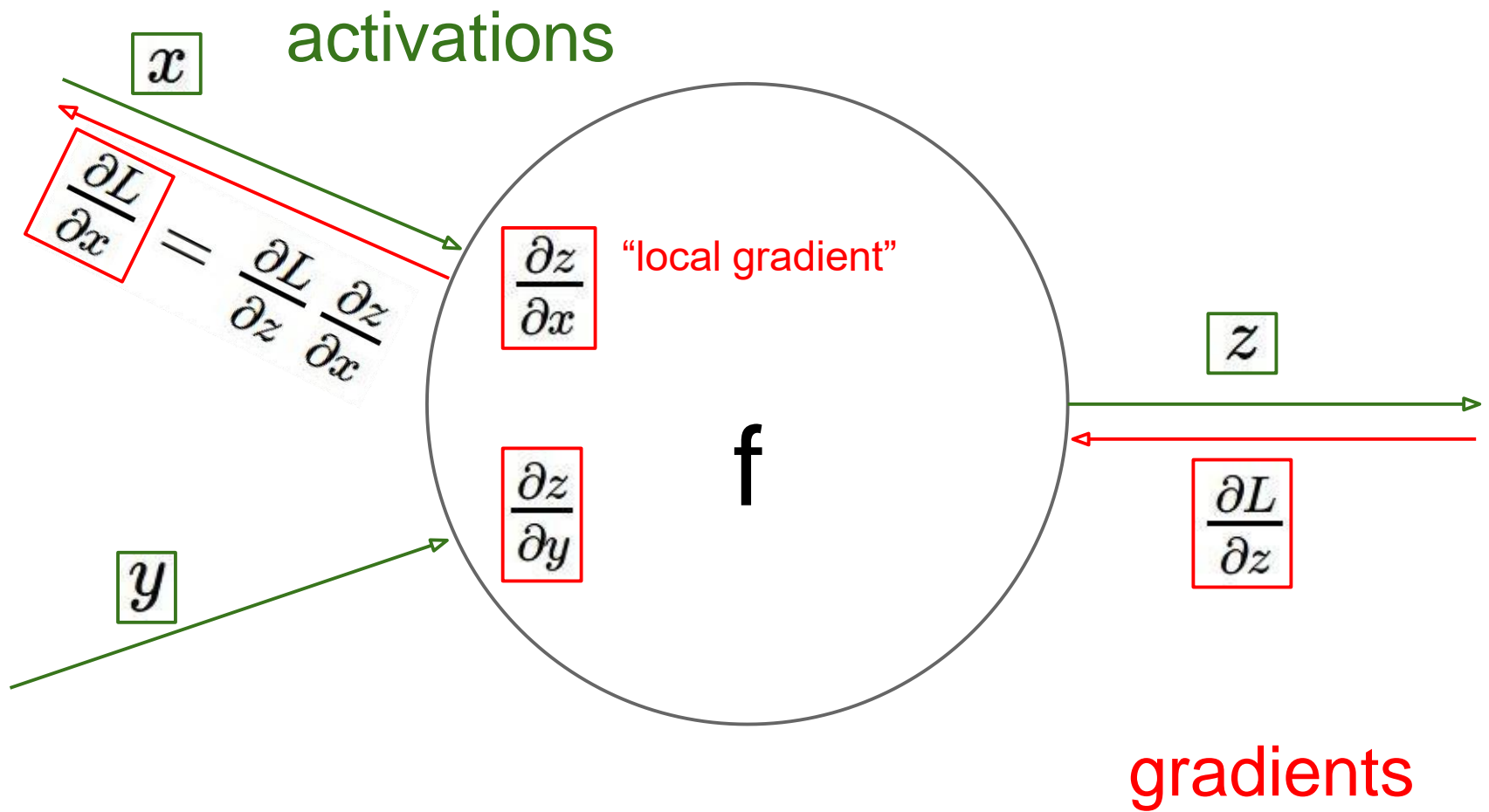
$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

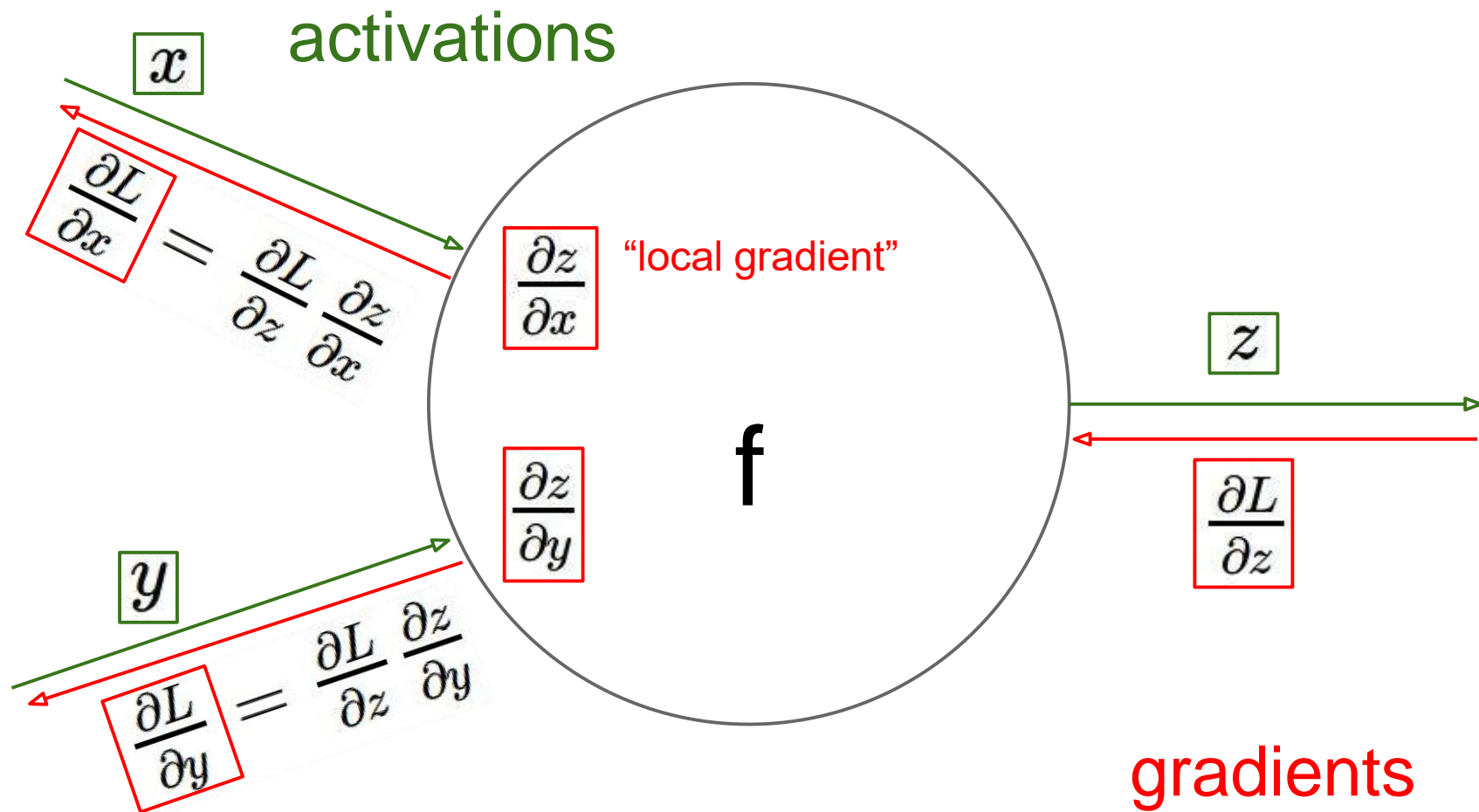








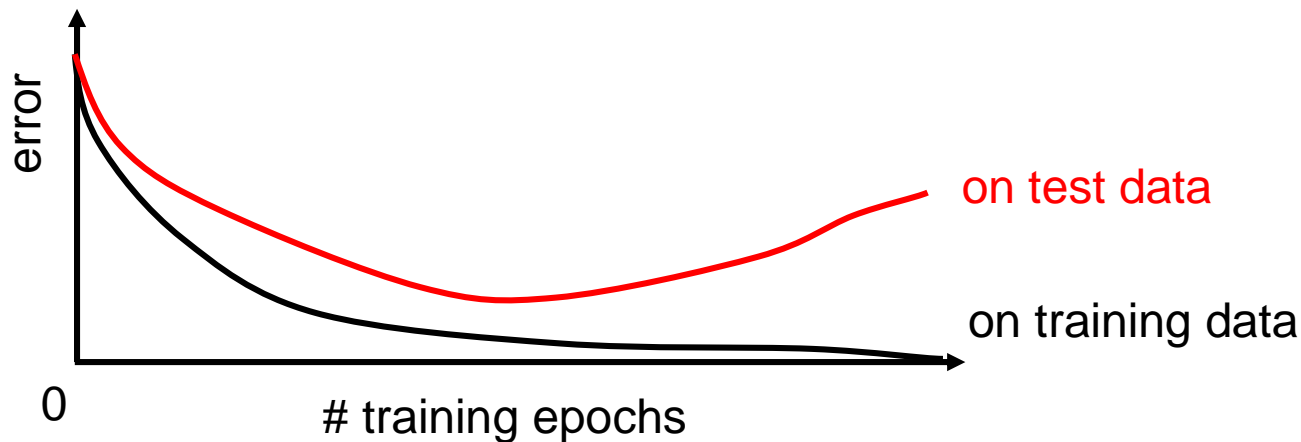




# Over-training prevention

---

- Running too many epochs can result in over-fitting.



- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase validation error.

# Comments on training algorithm

---

- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
- However, in practice, does converge to low error for many large networks on real data.
- Local minima – not a huge problem in practice for deep networks (but saddle points are).
- Thousands of epochs (epoch = network sees all training data once) may be required, hours or days to train.
- May be hard to set learning rate and to select number of hidden units and layers.
- When in doubt, use validation set to decide on design/hyperparameters.
- Neural networks had fallen out of fashion in 90s, early 2000s; now significantly improved performance (deep networks trained with dropout and lots of data).

# Plan for the rest of the lecture

---

## Neural network basics

- Definition
- Loss functions
- Optimization w/ gradient descent and backpropagation

## Convolutional neural networks (CNNs)

- Special operations
- Common architectures

## Practical matters

- Getting started: Preprocessing, initialization, optimization, normalization
- Improving performance: regularization, augmentation, transfer
- Hardware and software

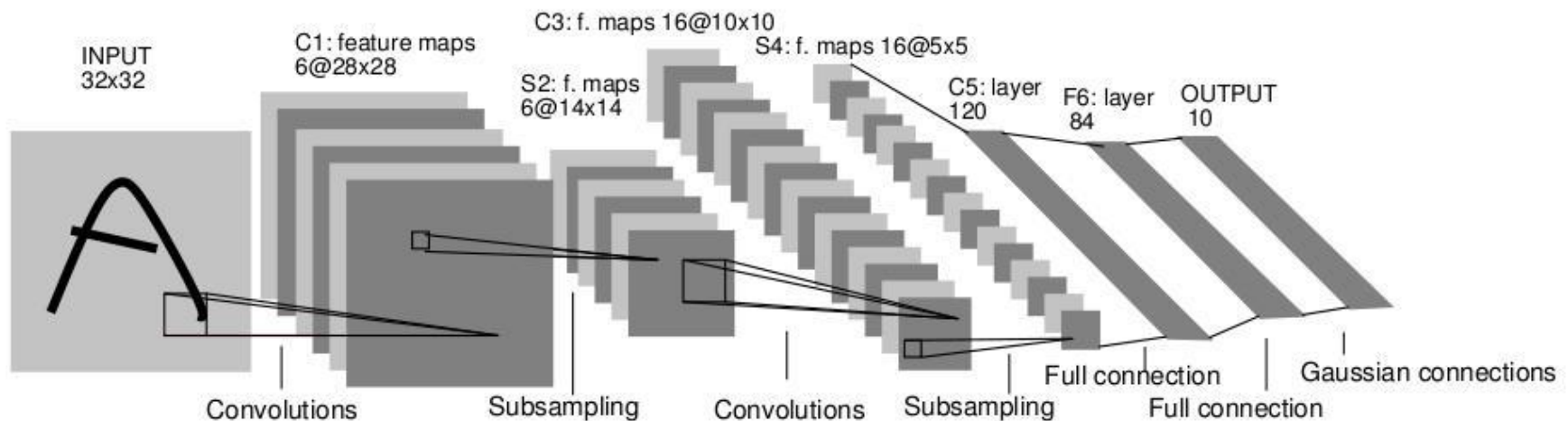
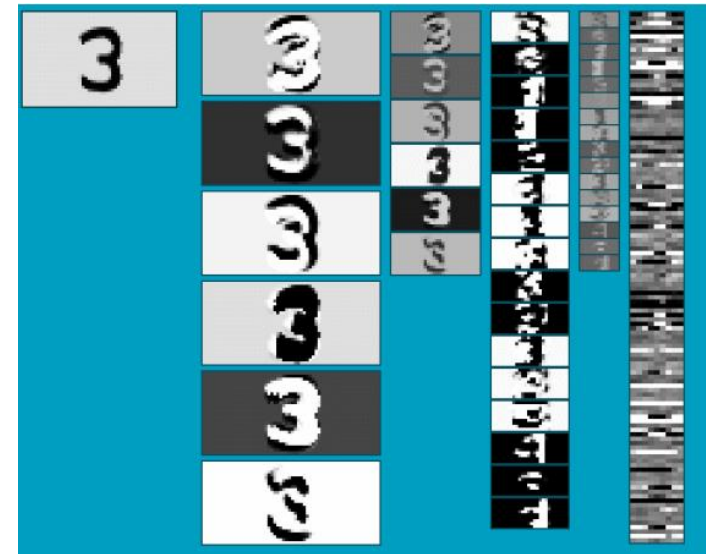
## Understanding CNNs

- Visualization
- Breaking CNNs

# Convolutional neural networks

# Convolutional Neural Networks (CNN)

- Neural network with specialized connectivity structure
- Stack multiple stages of feature extractors
- Higher stages compute more global, more invariant, *more abstract* features
- Classification layer at the end



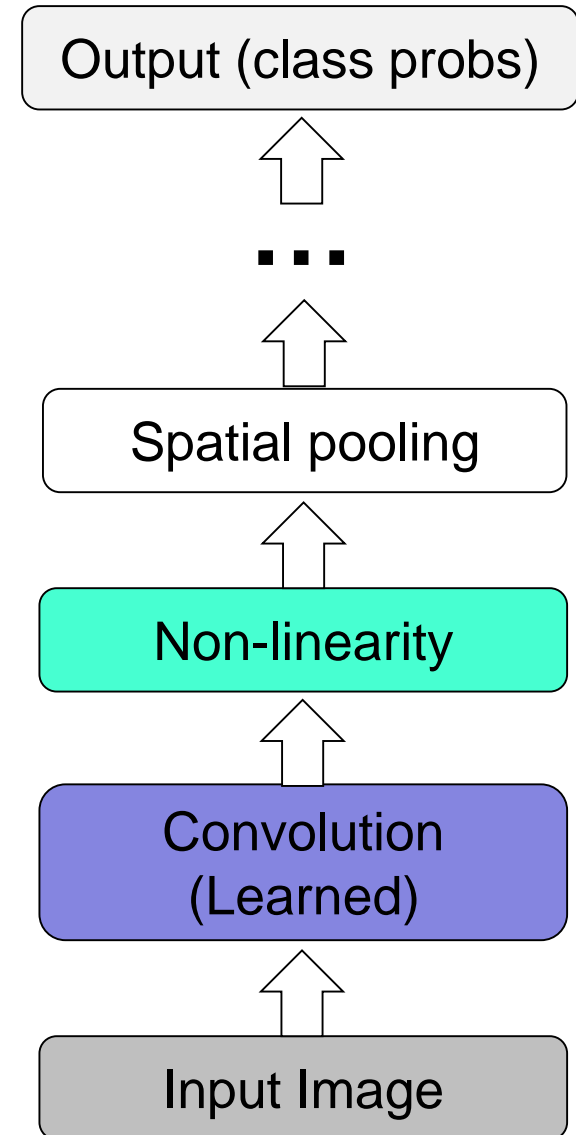
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, [Gradient-based learning applied to document recognition](#), Proceedings of the IEEE 86(11): 2278–2324, 1998.

Adapted from Rob Fergus

# Convolutional Neural Networks (CNN)

---

- Feed-forward feature extraction:
  1. Convolve input with learned filters
  2. Apply non-linearity
  3. **Spatial pooling (downsample)**
- Recent architectures have additional operations (to be discussed)
- Trained with some loss, backprop

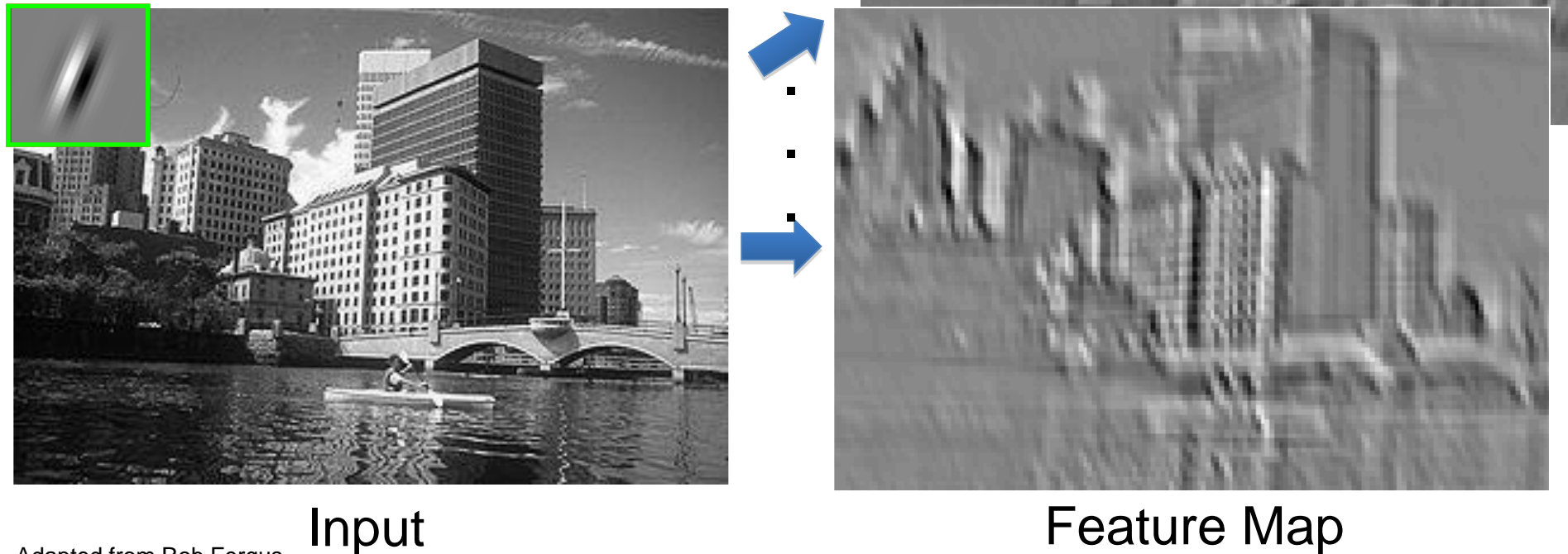




# 1. Convolution

---

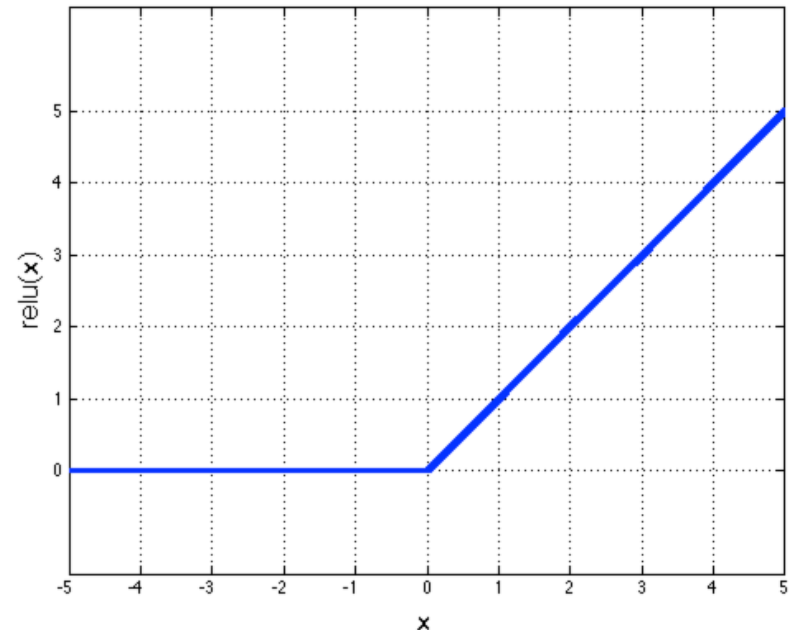
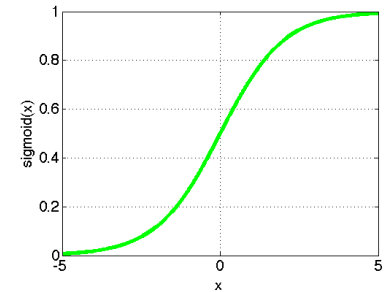
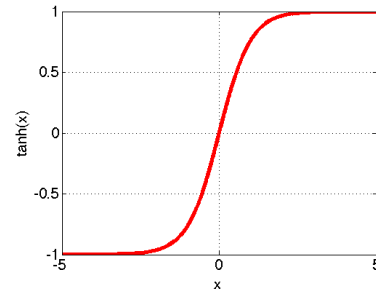
- Apply learned filter weights
- One feature map per filter
- Stride can be greater than 1 (faster, less memory)



## 2. Non-Linearity

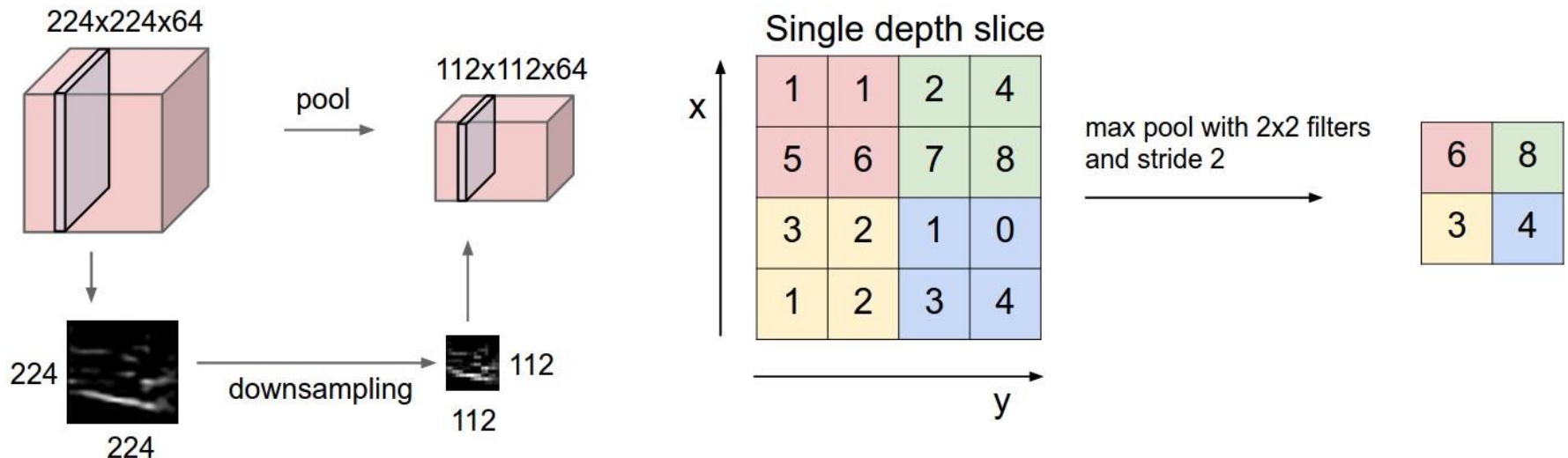
---

- Per-element (independent)
- Some options:
  - **Tanh**
  - **Sigmoid**:  $1/(1+\exp(-x))$
  - **Rectified linear unit (ReLU)**
    - Avoids saturation issues



# 3. Spatial Pooling

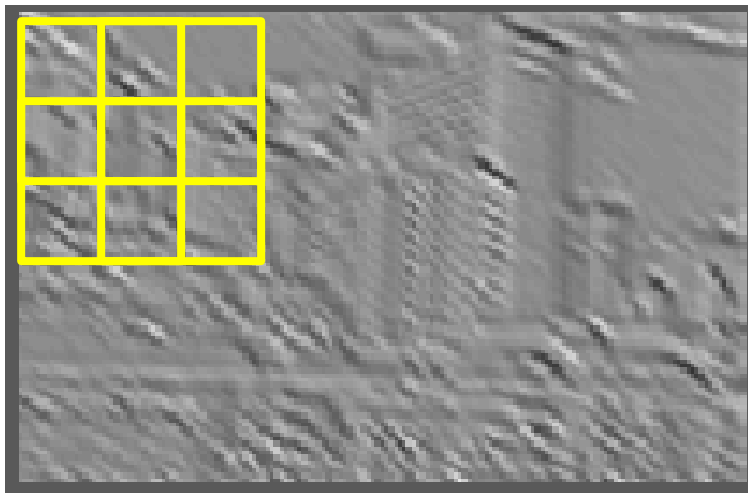
- Sum or max over non-overlapping / overlapping regions



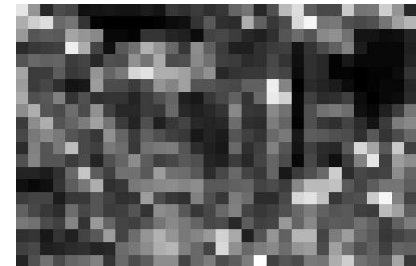
# 3. Spatial Pooling

---

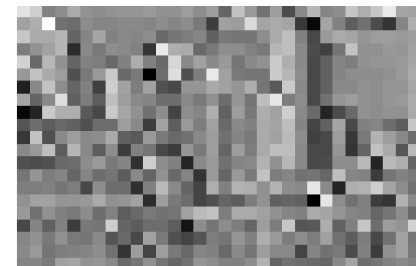
- Sum or max over non-overlapping / overlapping regions
- Role of pooling:
  - Invariance to small transformations
  - Larger receptive fields (neurons see more of input)



**Max**

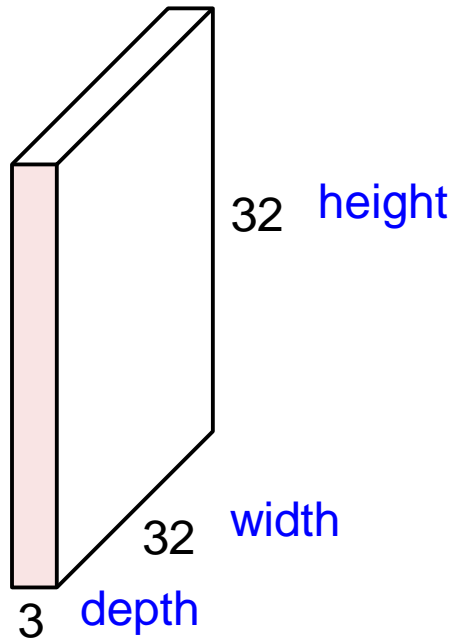


**Sum**



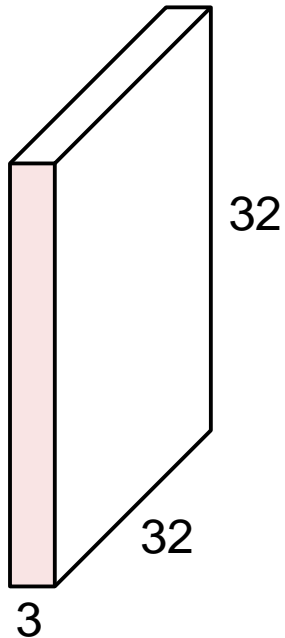
# Convolutions: More detail

32x32x3 image



# Convolutions: More detail

32x32x3 image



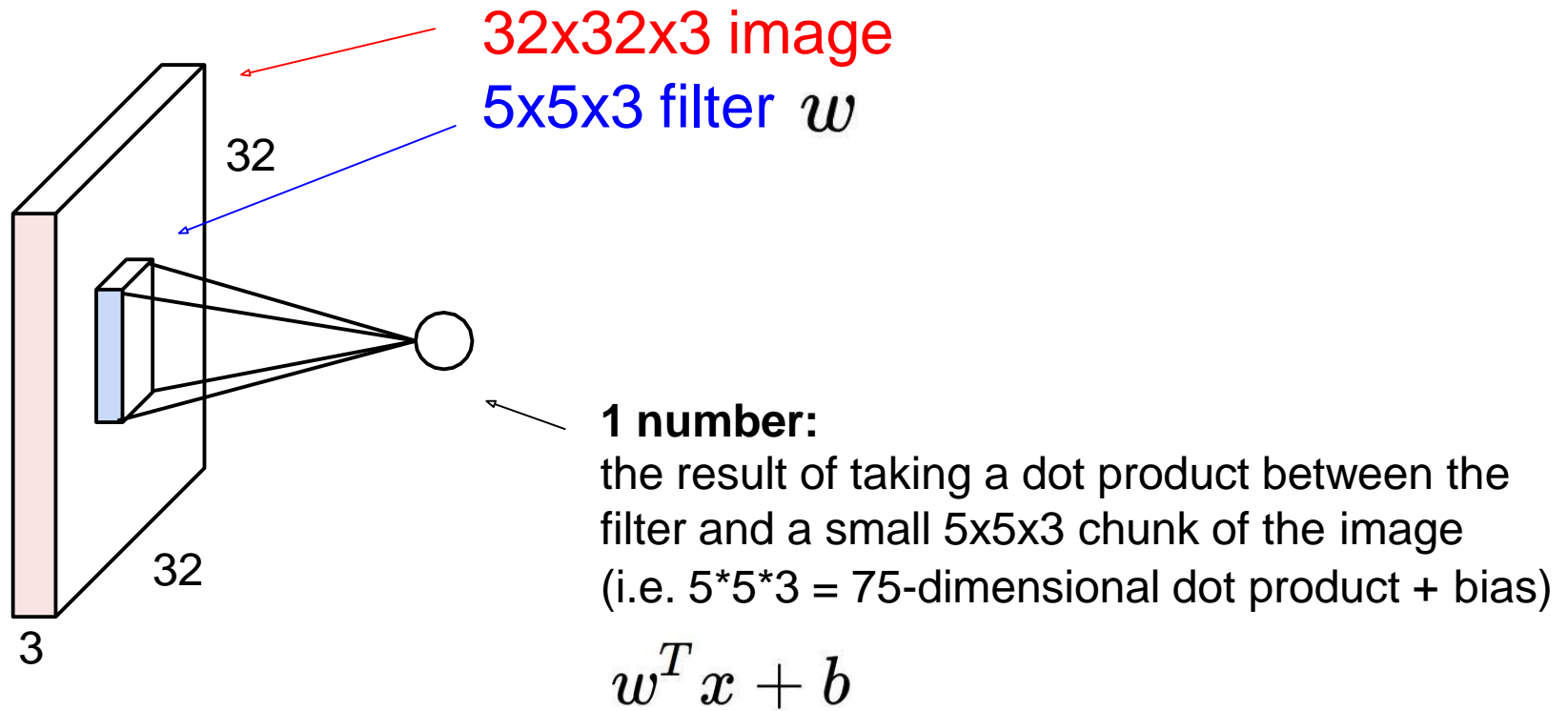
5x5x3 filter



**Convolve** the filter with the image  
i.e. “slide over the image spatially,  
computing dot products”

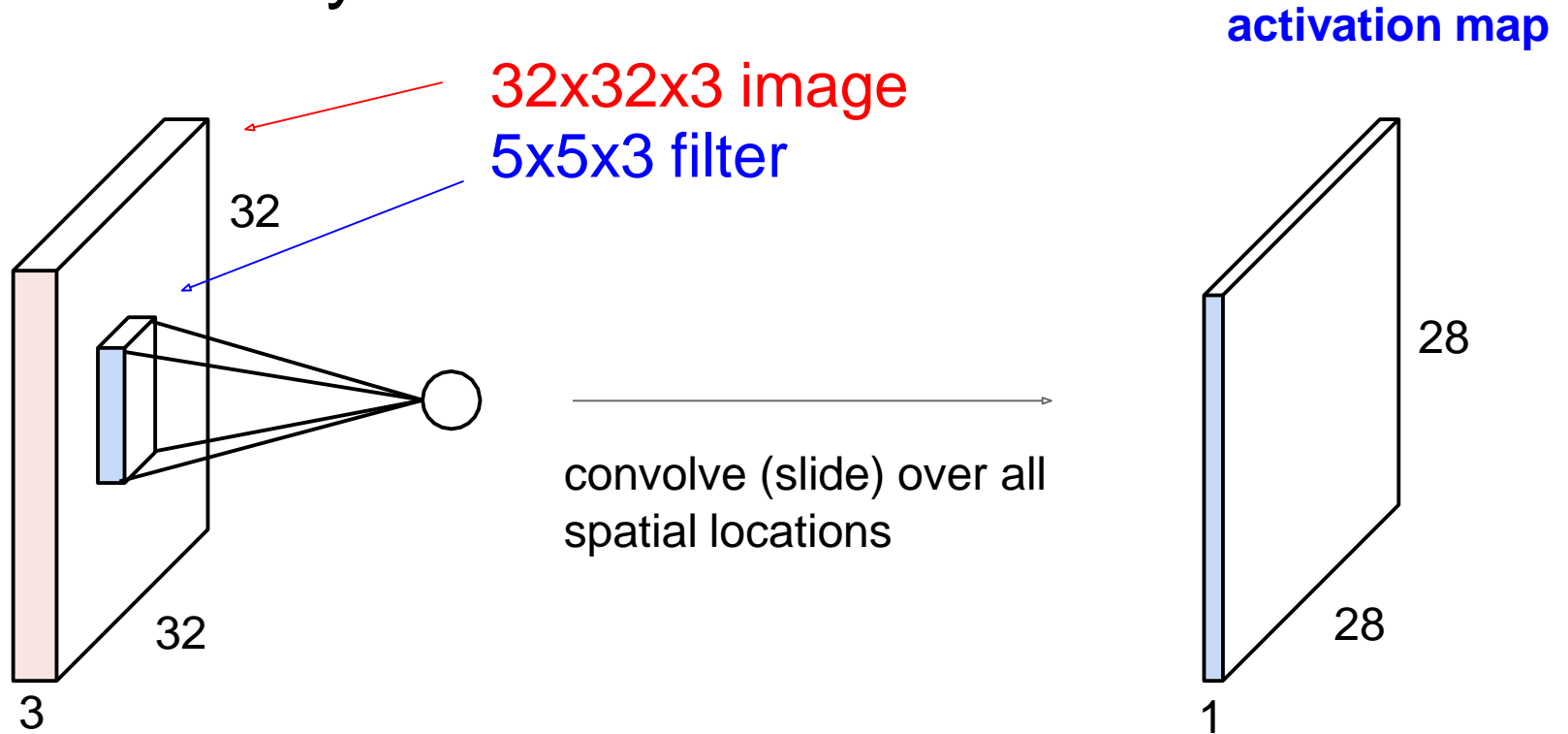
# Convolutions: More detail

## Convolution Layer



# Convolutions: More detail

## Convolution Layer

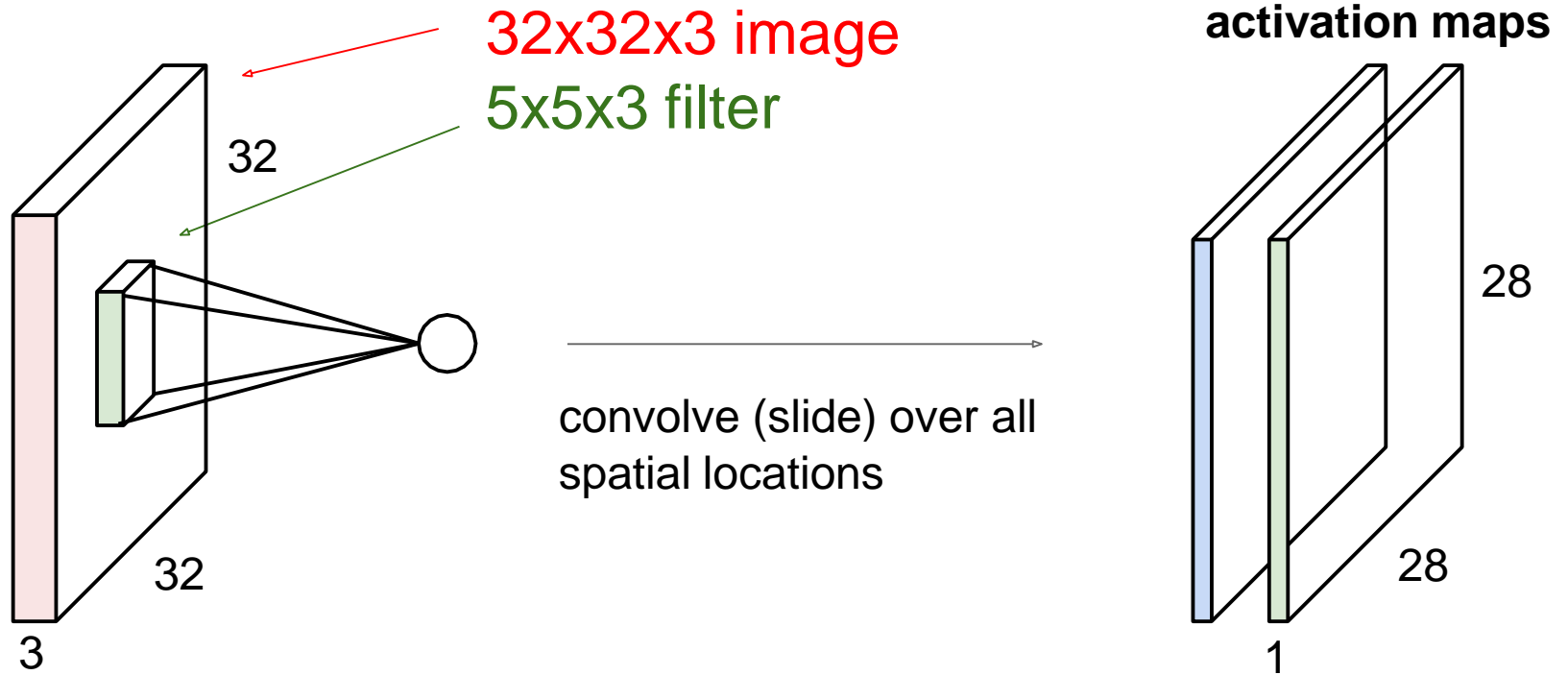




# Convolutions: More detail

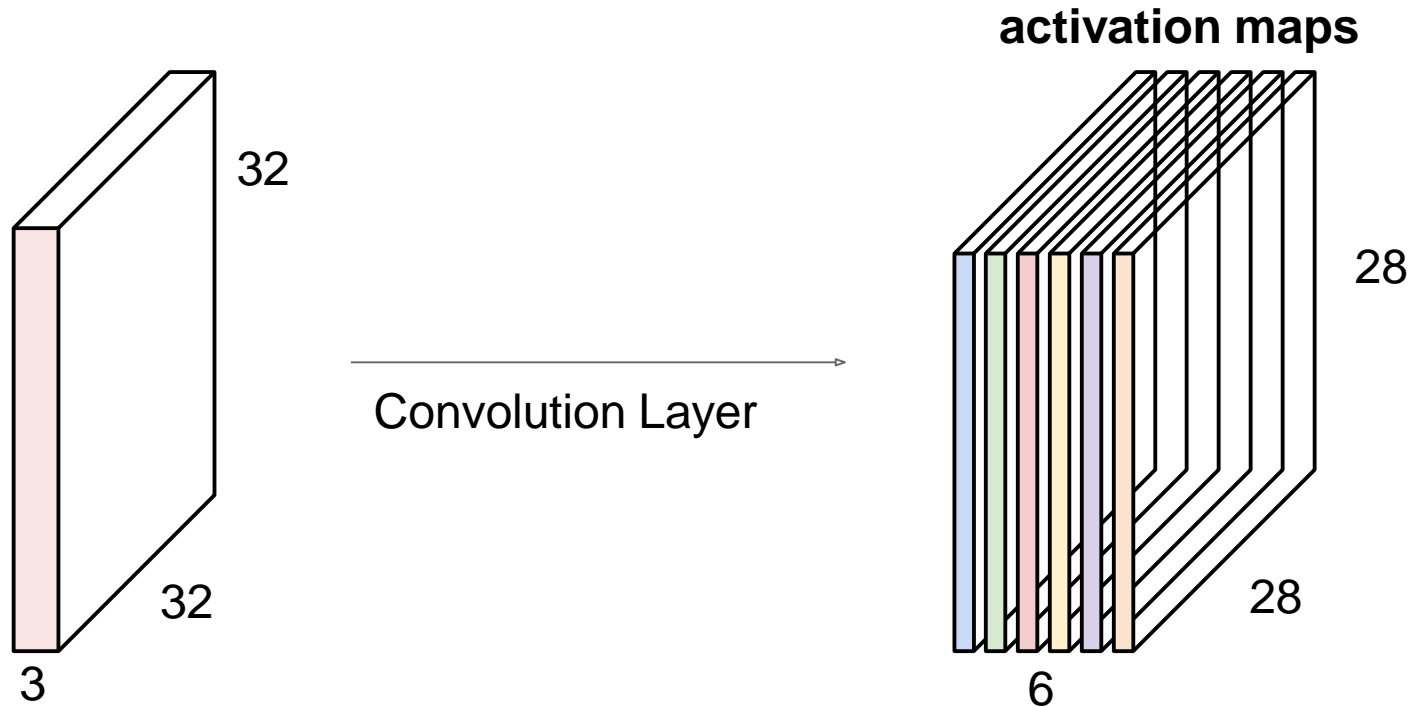
## Convolution Layer

consider a second, **green** filter



# Convolutions: More detail

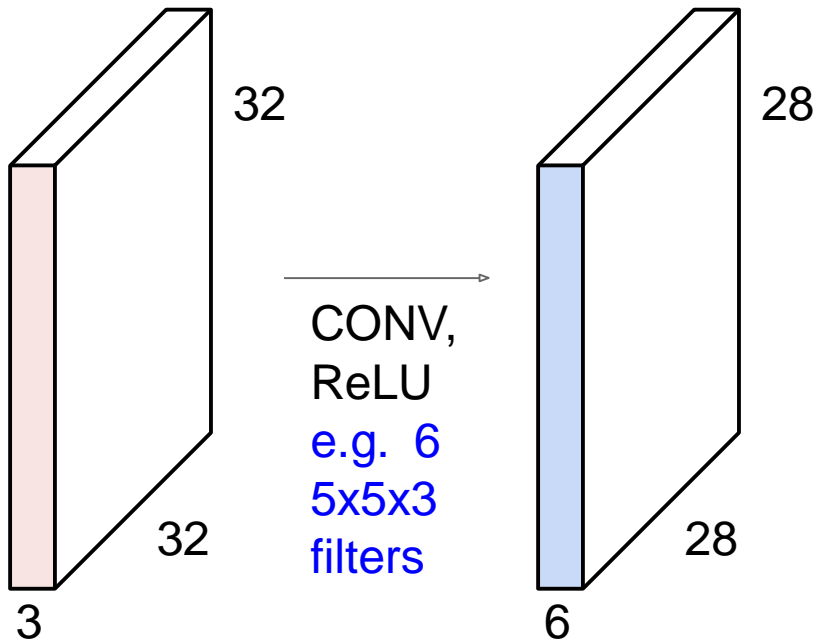
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a “new image” of size 28x28x6!

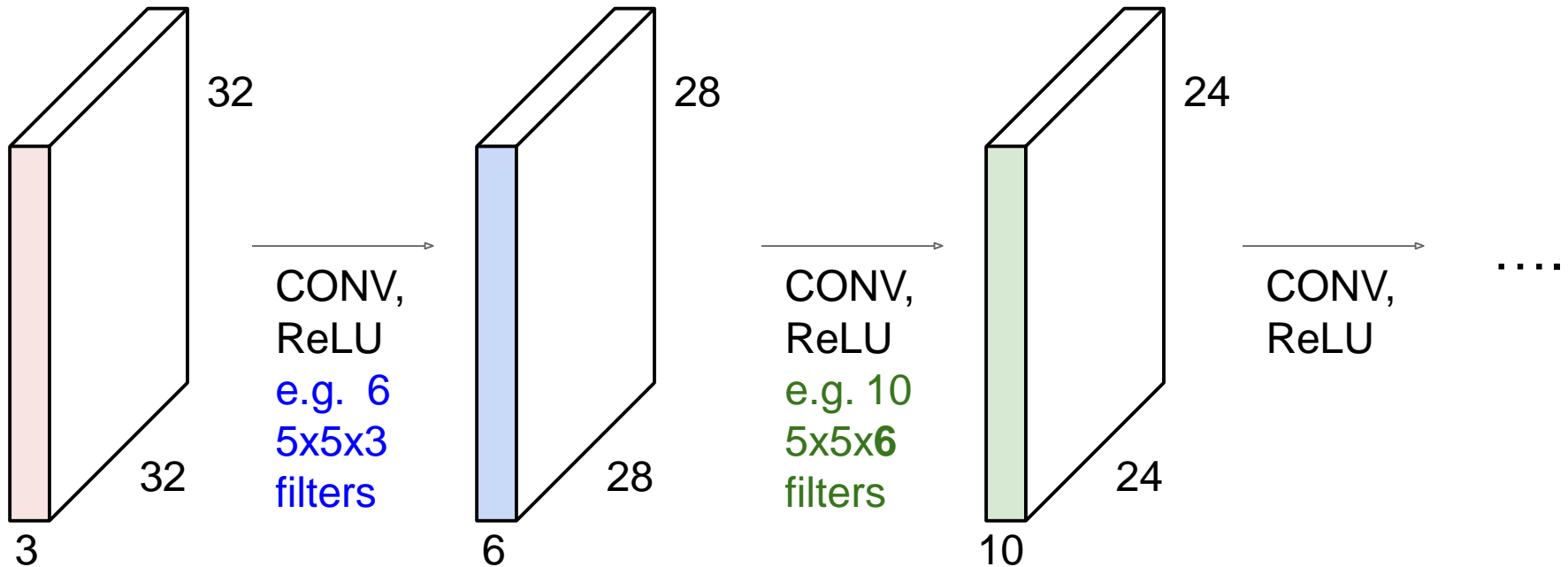
# Convolutions: More detail

**Preview:** ConvNet is a sequence of Convolution Layers, interspersed with activation functions



# Convolutions: More detail

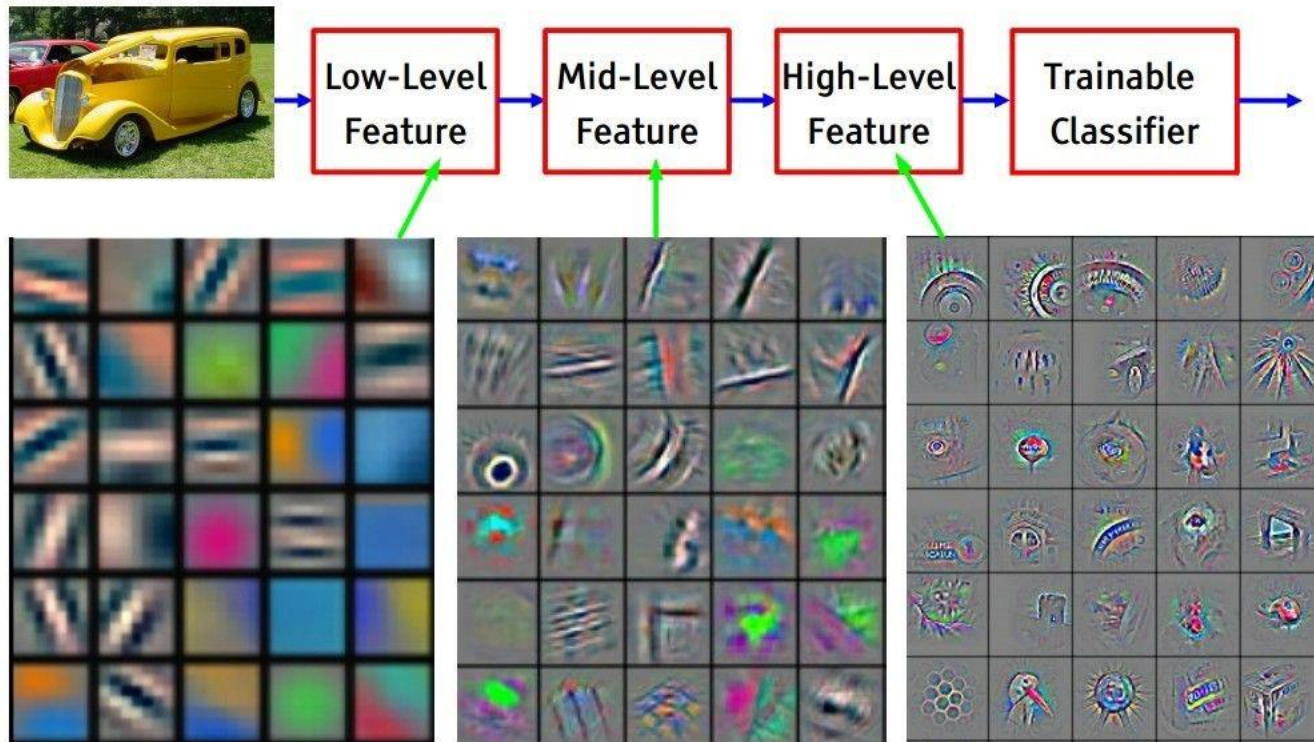
**Preview:** ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



# Convolutions: More detail

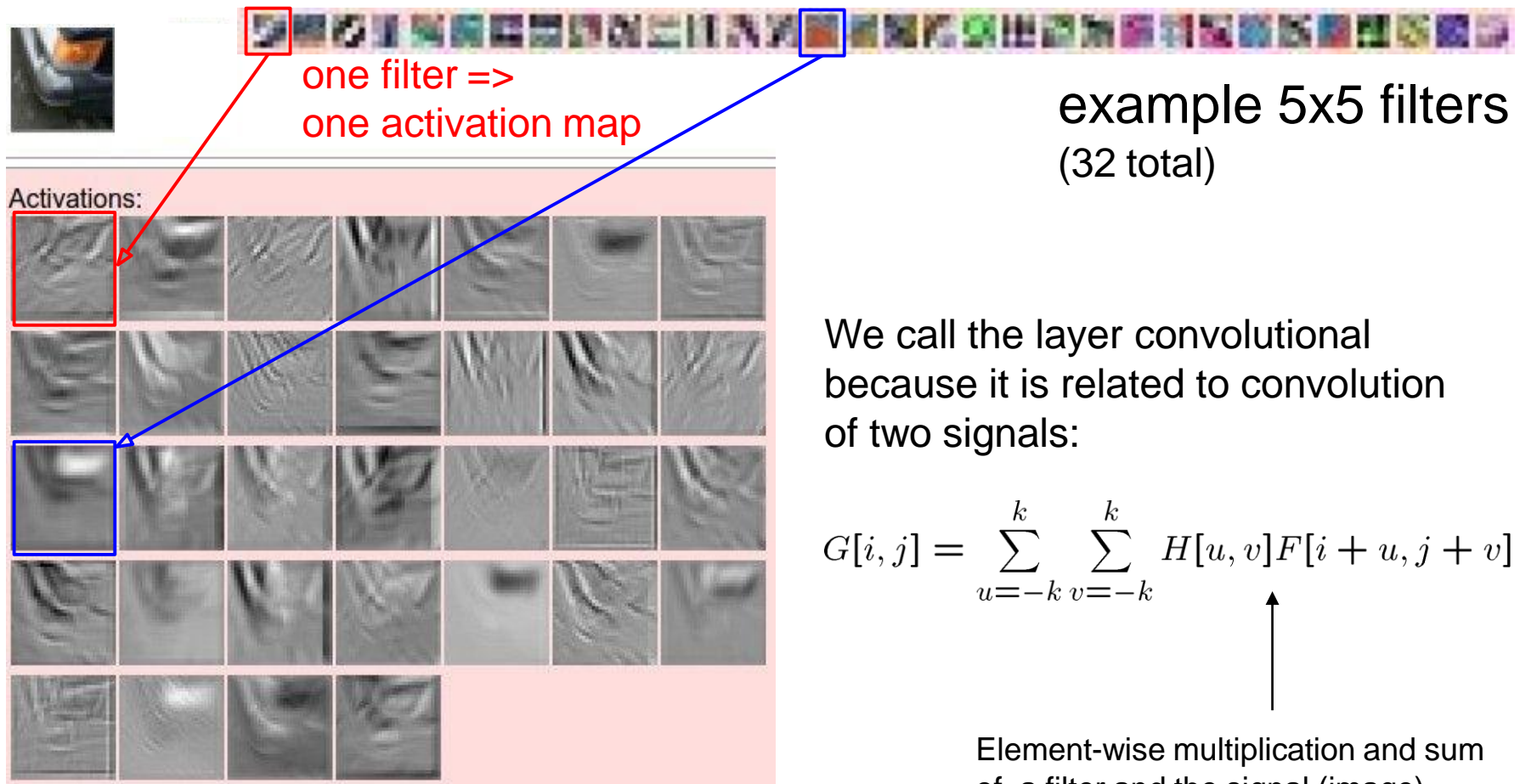
## Preview

*[From recent Yann LeCun slides]*



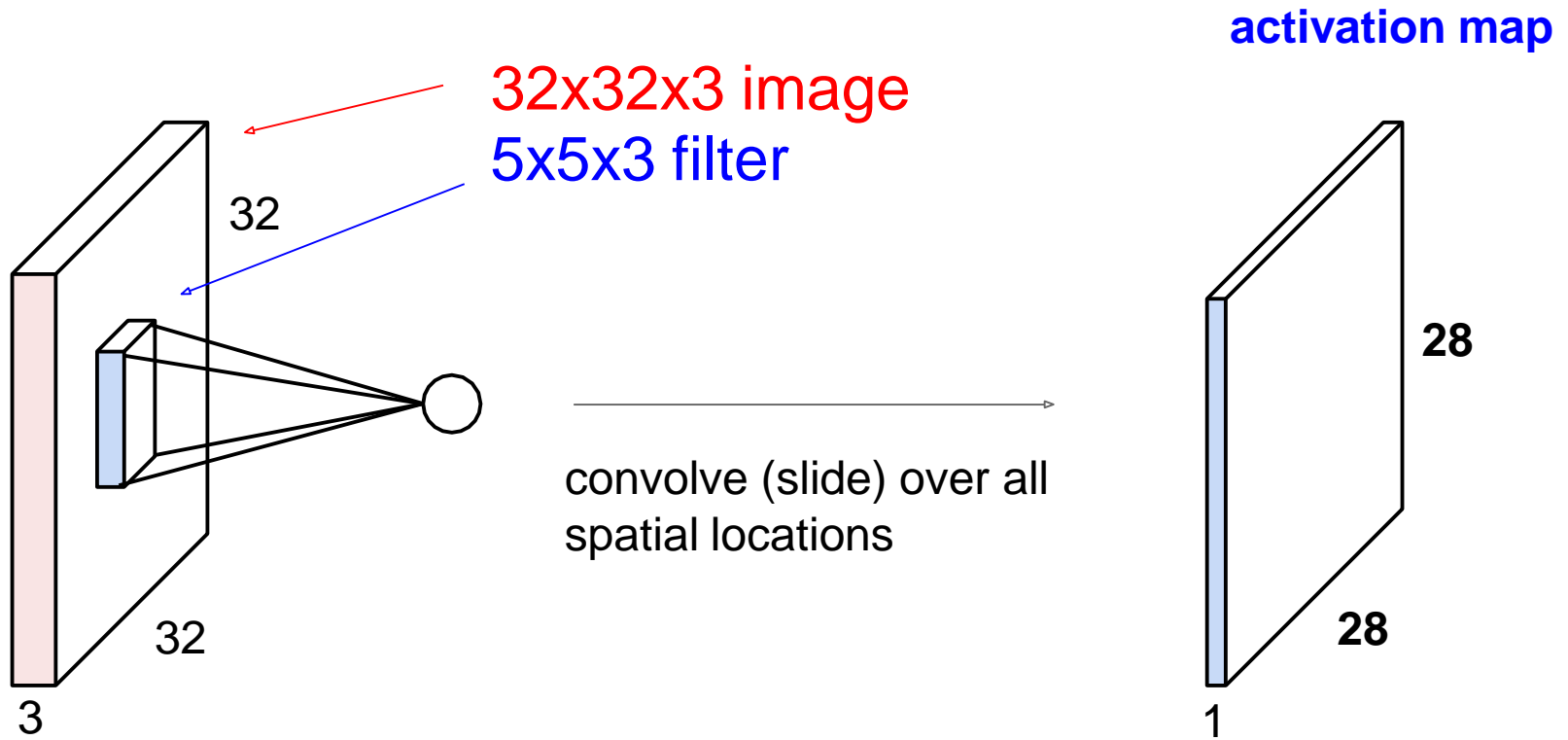
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

# Convolutions: More detail



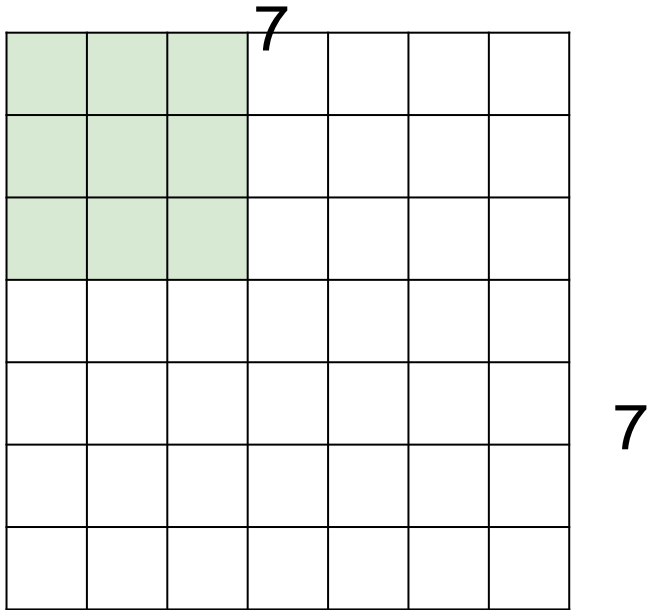
# Convolutions: More detail

A closer look at spatial dimensions:



# Convolutions: More detail

A closer look at spatial dimensions:

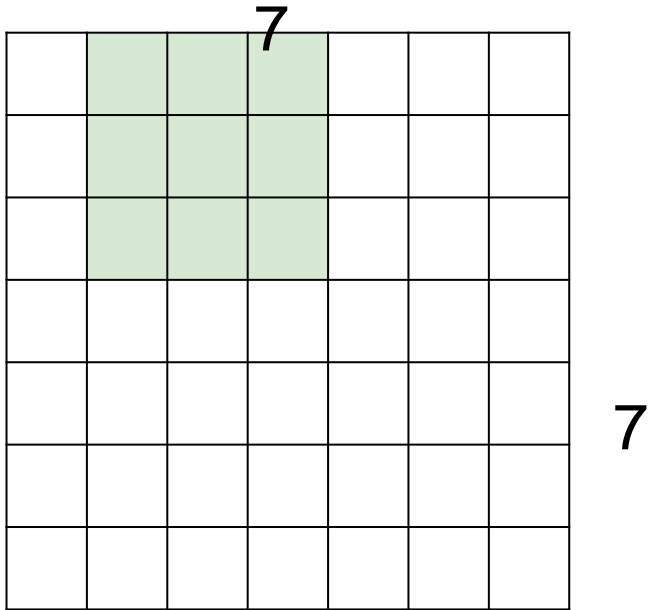


7x7 input (spatially)  
assume 3x3 filter



# Convolutions: More detail

A closer look at spatial dimensions:

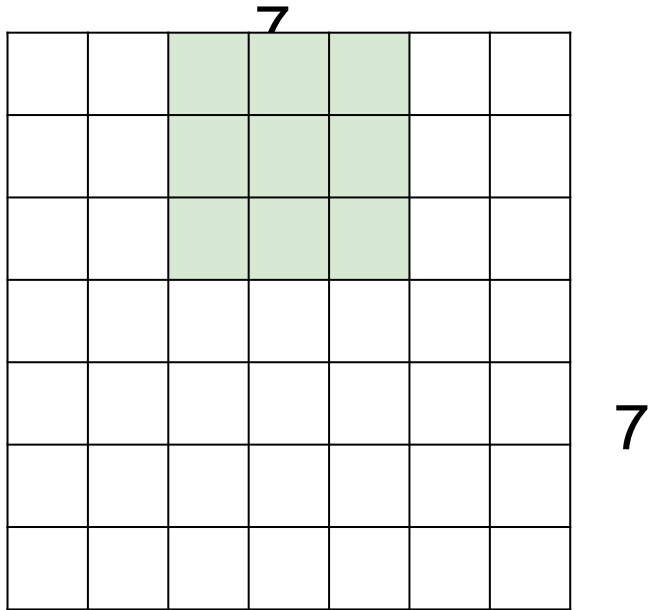


7x7 input (spatially)  
assume 3x3 filter

7

# Convolutions: More detail

A closer look at spatial dimensions:

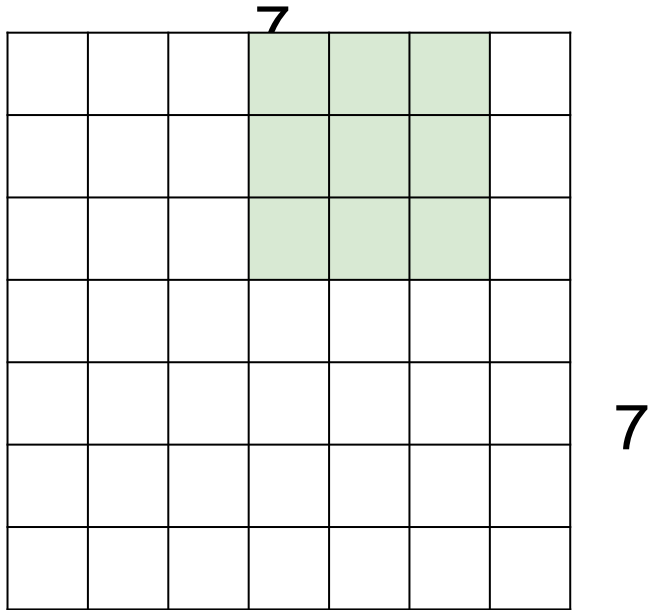


7x7 input (spatially)  
assume 3x3 filter

7

# Convolutions: More detail

A closer look at spatial dimensions:

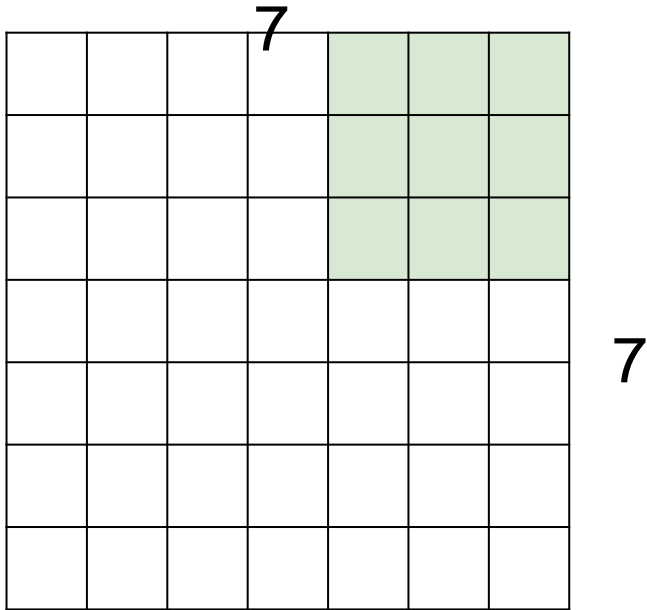


7x7 input (spatially)  
assume 3x3 filter

7

# Convolutions: More detail

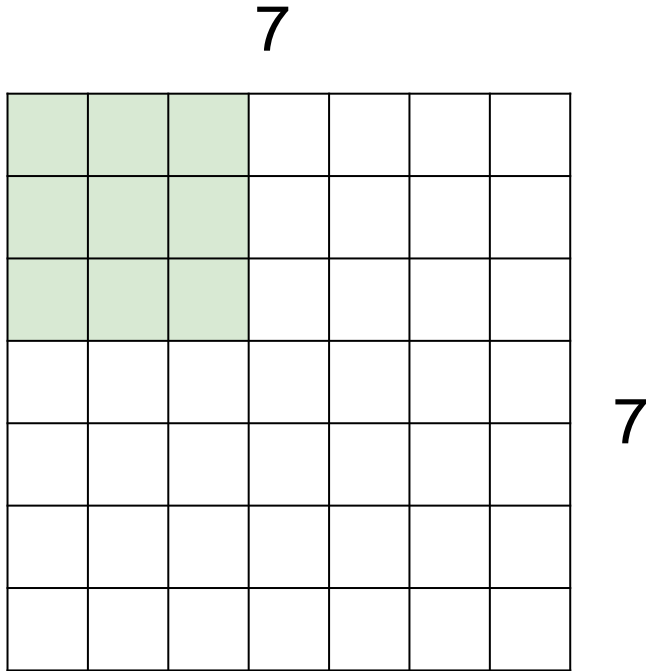
A closer look at spatial dimensions:



7x7 input (spatially)  
assume 3x3 filter  
**=> 5x5 output**

# Convolutions: More detail

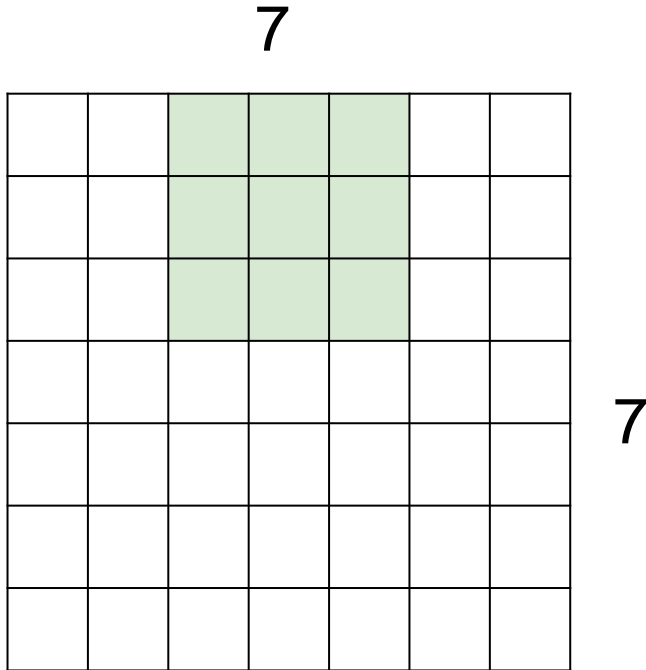
A closer look at spatial dimensions:



7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**

# Convolutions: More detail

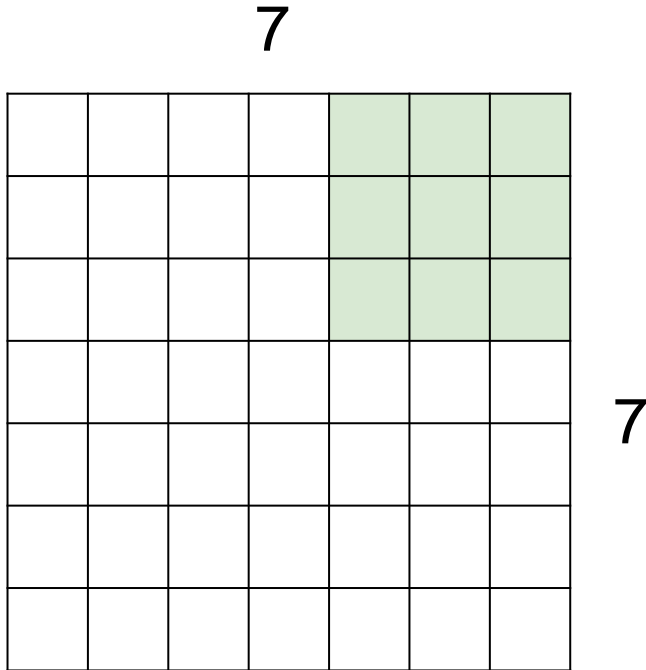
A closer look at spatial dimensions:



7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**

# Convolutions: More detail

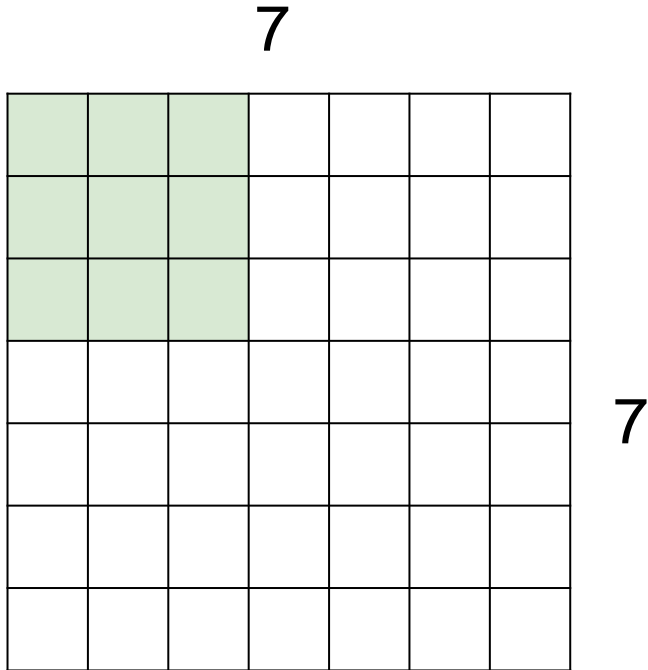
A closer look at spatial dimensions:



7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**  
**=> 3x3 output!**

# Convolutions: More detail

A closer look at spatial dimensions:

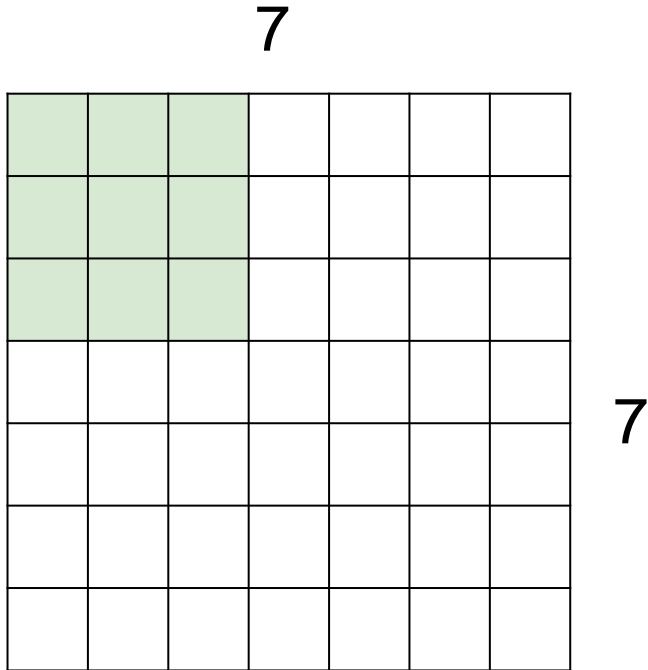


7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 3?**



# Convolutions: More detail

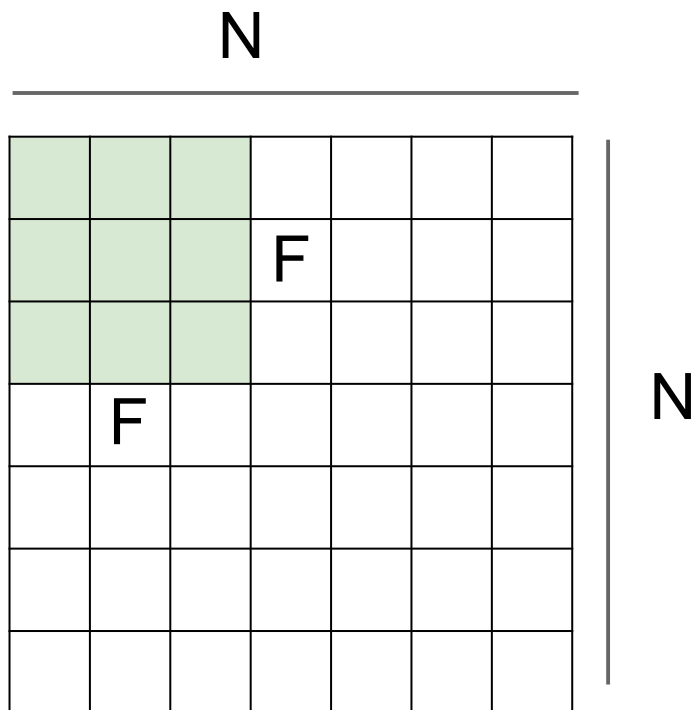
A closer look at spatial dimensions:



7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 3?**

**doesn't fit!**  
cannot apply 3x3 filter on  
7x7 input with stride 3.

# Convolutions: More detail



Output size:  
 **$(N - F) / \text{stride} + 1$**

e.g.  $N = 7, F = 3$ :

stride 1  $\Rightarrow (7 - 3) / 1 + 1 = 5$

stride 2  $\Rightarrow (7 - 3) / 2 + 1 = 3$

stride 3  $\Rightarrow (7 - 3) / 3 + 1 = 2.33 \therefore \backslash$

# Convolutions: More detail

In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

**3x3** filter, applied with **stride 1**

**pad with 1 pixel** border => what is the output?

(recall:)

$$(N - F) / \text{stride} + 1$$

# Convolutions: More detail

In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

**3x3** filter, applied with **stride 1**

**pad with 1 pixel** border => what is the output?

**7x7 output!**

# Convolutions: More detail

In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

**3x3** filter, applied with **stride 1**

**pad with 1 pixel** border => what is the output?

**7x7 output!**

in general, common to see CONV layers with stride 1, filters of size  $F \times F$ , and zero-padding with  $(F-1)/2$ . (will preserve size spatially)

e.g.  $F = 3 \Rightarrow$  zero pad with 1

$F = 5 \Rightarrow$  zero pad with 2

$F = 7 \Rightarrow$  zero pad with 3

$$(N + 2 * \text{padding} - F) / \text{stride} + 1$$

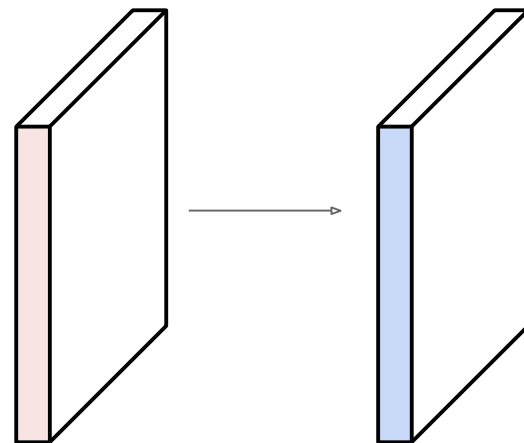
# Convolutions: More detail

Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Output volume size: ?



# Convolutions: More detail

Examples time:

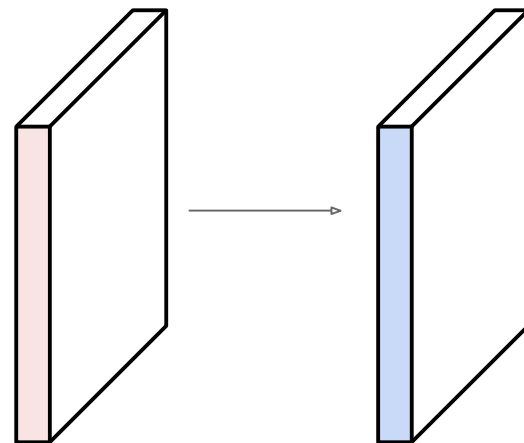
Input volume: **32x32x3**

**10** **5x5**x3 filters with stride **1**, pad **2**

Output volume size:

$(32 + 2 * 2 - 5) / 1 + 1 = 32$  spatially, so

**32x32x10**

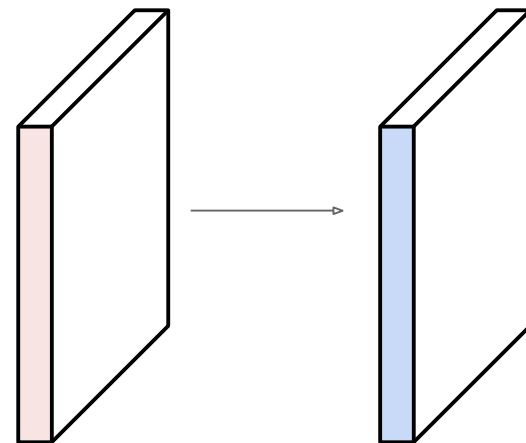


# Convolutions: More detail

Examples time:

Input volume: **32x32x3**

10 5x5x3 filters with stride 1, pad 2



Number of parameters in this layer?

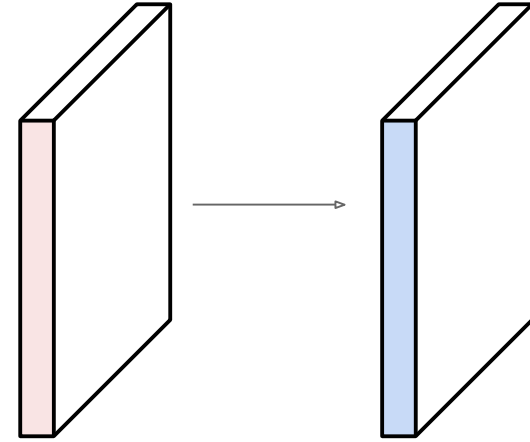


# Convolutions: More detail

Examples time:

Input volume: **32x32x3**

**10** **5x5x3** filters with stride 1, pad 2

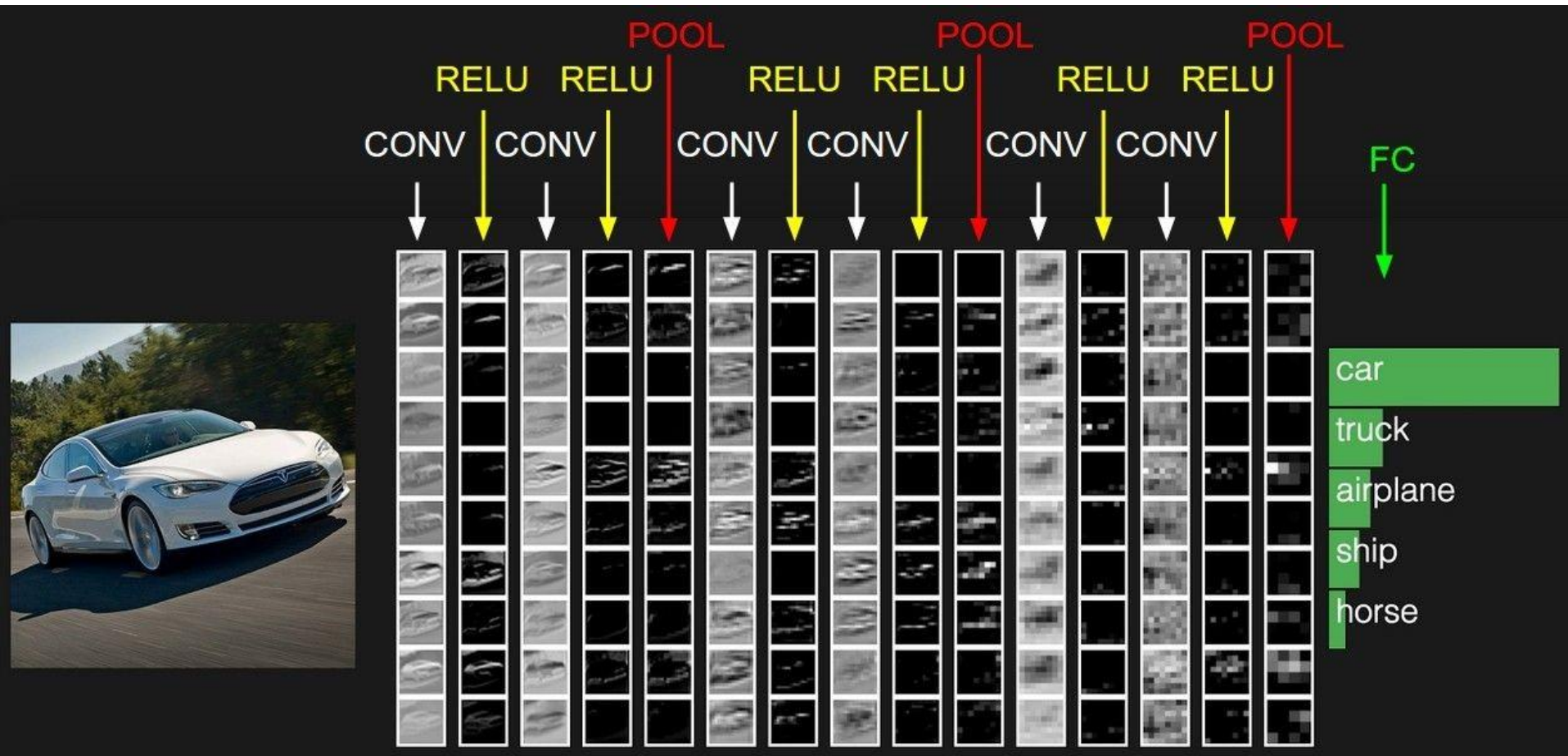


Number of parameters in this layer?

each filter has  $5*5*3 + 1 = 76$  params (+1 for bias)

=>  $76*10 = 760$

# Putting it all together



# Case Study: AlexNet

*[Krizhevsky et al. 2012]*

## Architecture:

CONV1

MAX POOL1

NORM1

CONV2

MAX POOL2

NORM2

CONV3

CONV4

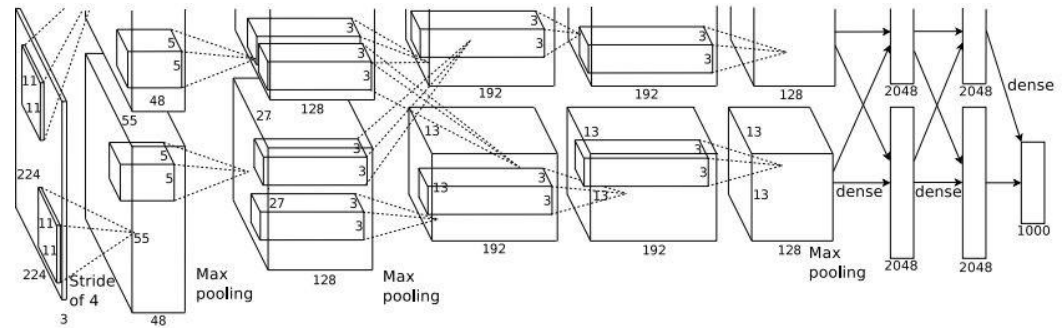
CONV5

Max POOL3

FC6

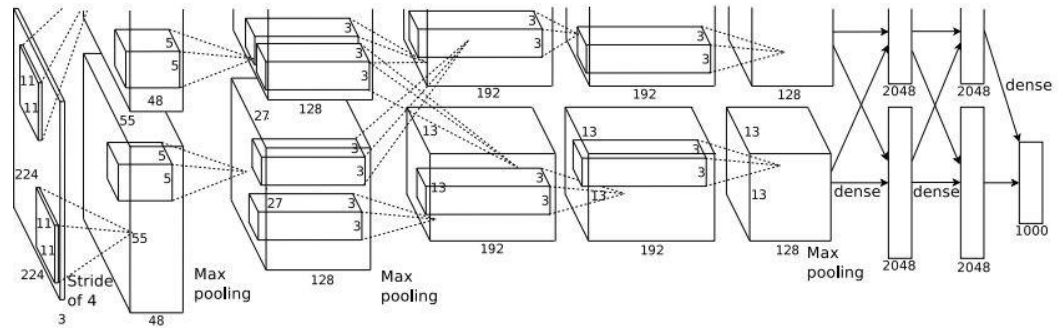
FC7

FC8



# Case Study: AlexNet

*[Krizhevsky et al. 2012]*



Input: 227x227x3 images

**First layer (CONV1):** 96 11x11 filters applied at stride 4

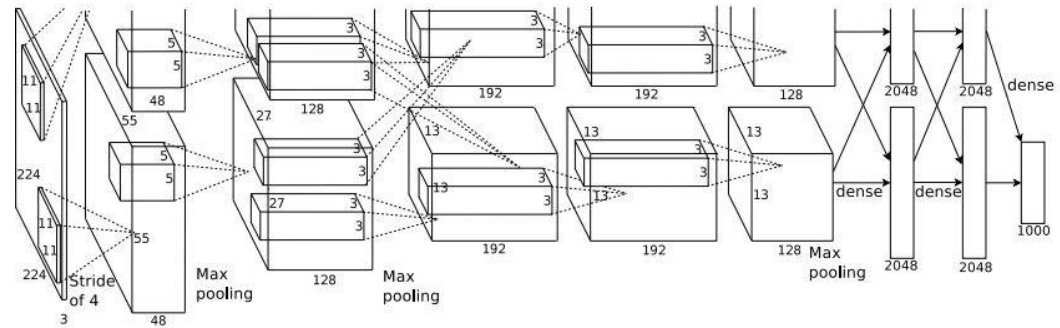
=>

Output volume **[55x55x96]**

Parameters:  $(11*11*3)*96 = 35K$

# Case Study: AlexNet

*[Krizhevsky et al. 2012]*



Input: 227x227x3 images

After CONV1: 55x55x96

**Second layer (POOL1):** 3x3 filters applied at stride 2

Output volume: 27x27x96

Q: what is the number of parameters in this layer?

# Case Study: AlexNet

*[Krizhevsky et al. 2012]*

Full (simplified) AlexNet architecture:

[224x224x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

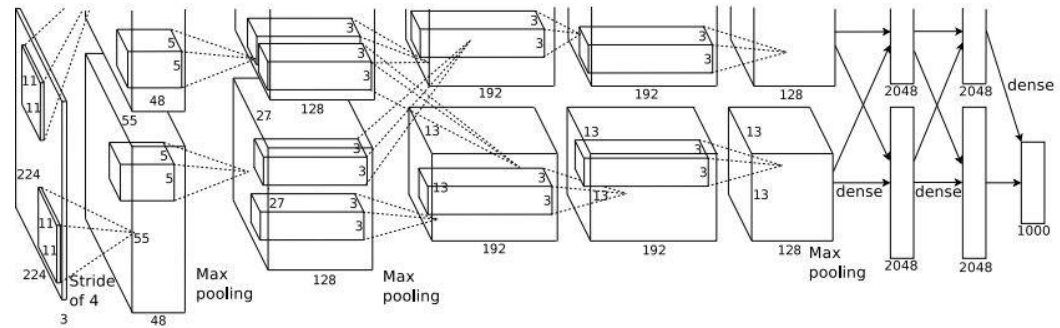
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

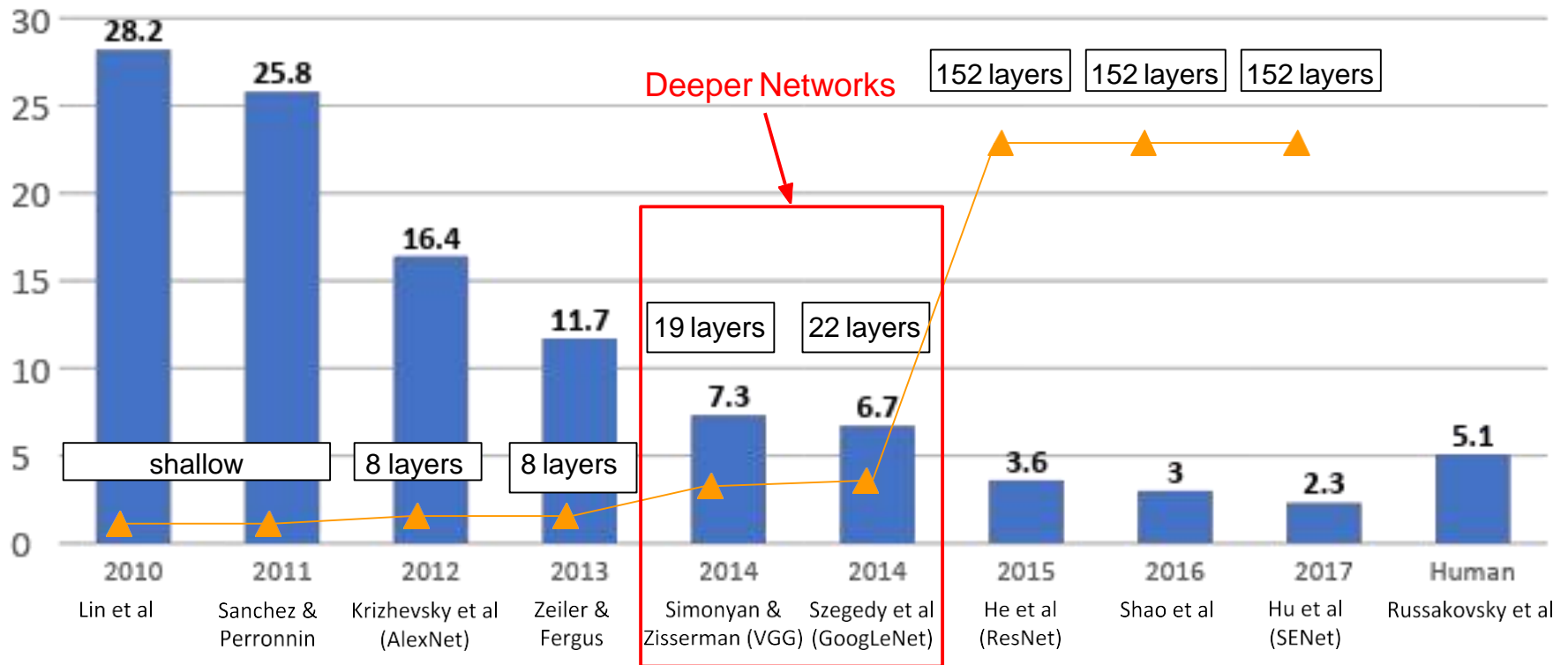
[1000] **FC8**: 1000 neurons (class scores)



## Details/Retrospectives:

- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

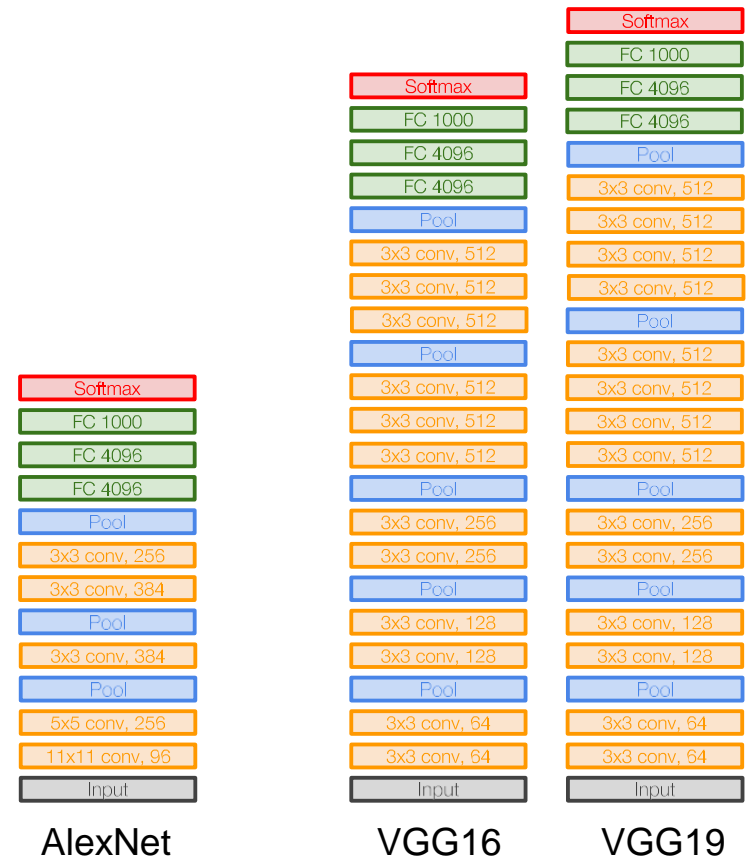
8 layers (AlexNet)

-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1  
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13  
(ZFNet)

-> 7.3% top 5 error in ILSVRC'14





# Case Study: VGGNet

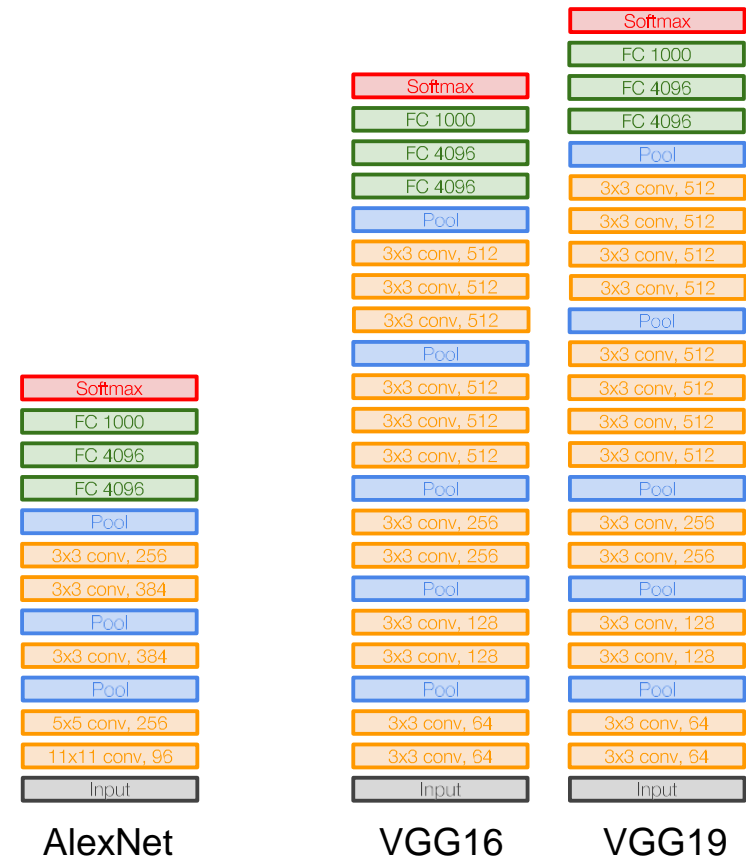
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters:  $3 * (3^2 C^2)$  vs.  $7^2 C^2$  for C channels per layer

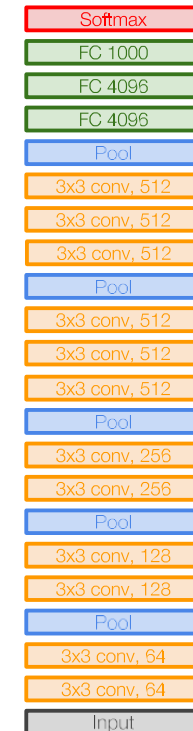


# Case Study: VGGNet

INPUT: [224x224x3]      memory:  $224*224*3=150K$     params: 0  
CONV3-64: [224x224x64]    memory:  $224*224*64=3.2M$     params:  $(3*3*3)*64 = 1,728$   
CONV3-64: [224x224x64]    memory:  $224*224*64=3.2M$     params:  $(3*3*64)*64 = 36,864$   
POOL2: [112x112x64]    memory:  $112*112*64=800K$     params: 0  
CONV3-128: [112x112x128]    memory:  $112*112*128=1.6M$     params:  $(3*3*64)*128 = 73,728$   
CONV3-128: [112x112x128]    memory:  $112*112*128=1.6M$     params:  $(3*3*128)*128 = 147,456$   
POOL2: [56x56x128]    memory:  $56*56*128=400K$     params: 0  
CONV3-256: [56x56x256]    memory:  $56*56*256=800K$     params:  $(3*3*128)*256 = 294,912$   
CONV3-256: [56x56x256]    memory:  $56*56*256=800K$     params:  $(3*3*256)*256 = 589,824$   
CONV3-256: [56x56x256]    memory:  $56*56*256=800K$     params:  $(3*3*256)*256 = 589,824$   
POOL2: [28x28x256]    memory:  $28*28*256=200K$     params: 0  
CONV3-512: [28x28x512]    memory:  $28*28*512=400K$     params:  $(3*3*256)*512 = 1,179,648$   
CONV3-512: [28x28x512]    memory:  $28*28*512=400K$     params:  $(3*3*512)*512 = 2,359,296$   
CONV3-512: [28x28x512]    memory:  $28*28*512=400K$     params:  $(3*3*512)*512 = 2,359,296$   
POOL2: [14x14x512]    memory:  $14*14*512=100K$     params: 0  
CONV3-512: [14x14x512]    memory:  $14*14*512=100K$     params:  $(3*3*512)*512 = 2,359,296$   
CONV3-512: [14x14x512]    memory:  $14*14*512=100K$     params:  $(3*3*512)*512 = 2,359,296$   
CONV3-512: [14x14x512]    memory:  $14*14*512=100K$     params:  $(3*3*512)*512 = 2,359,296$   
POOL2: [7x7x512]    memory:  $7*7*512=25K$     params: 0  
FC: [1x1x4096]    memory: 4096    params:  $7*7*512*4096 = 102,760,448$   
FC: [1x1x4096]    memory: 4096    params:  $4096*4096 = 16,777,216$   
FC: [1x1x1000]    memory: 1000    params:  $4096*1000 = 4,096,000$

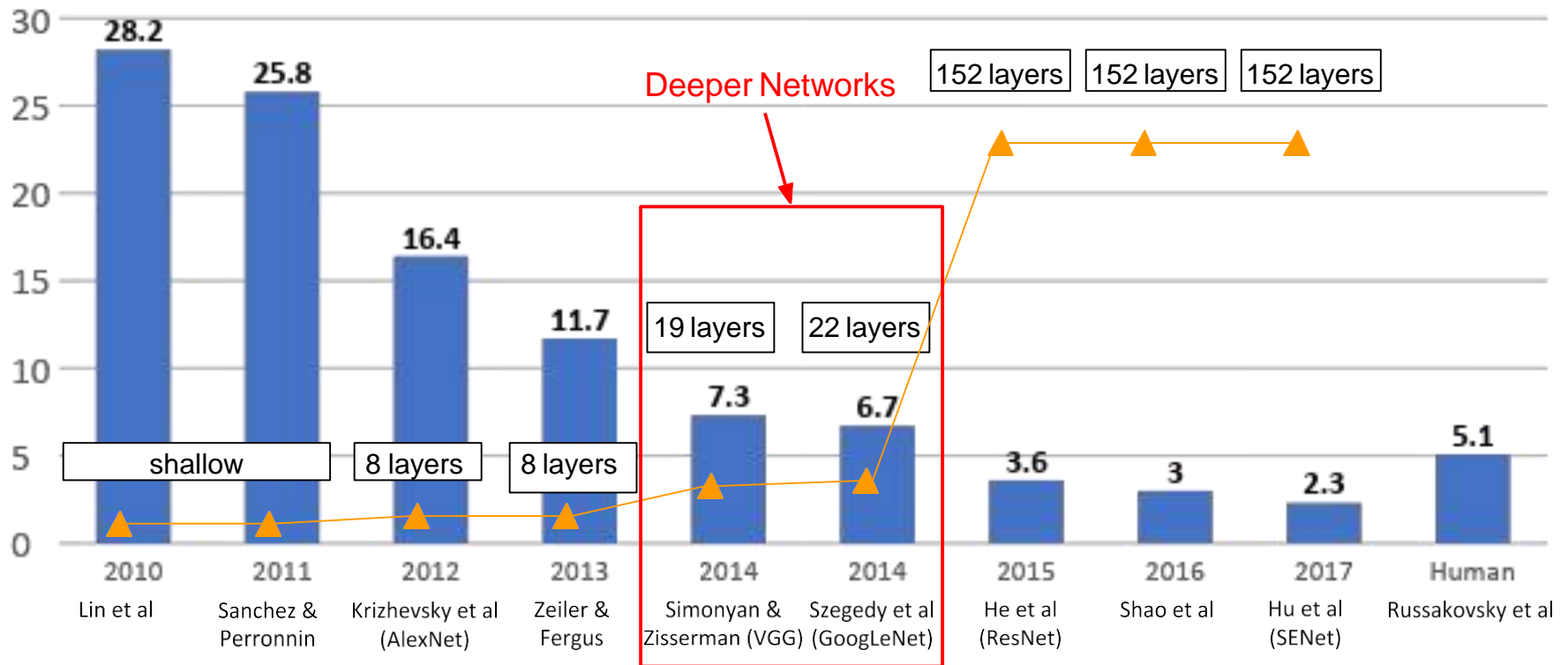
**TOTAL memory:**  $24M * 4 \text{ bytes} \approx 96MB / \text{image}$  (for a forward pass)

**TOTAL params:** 138M parameters



VGG16

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

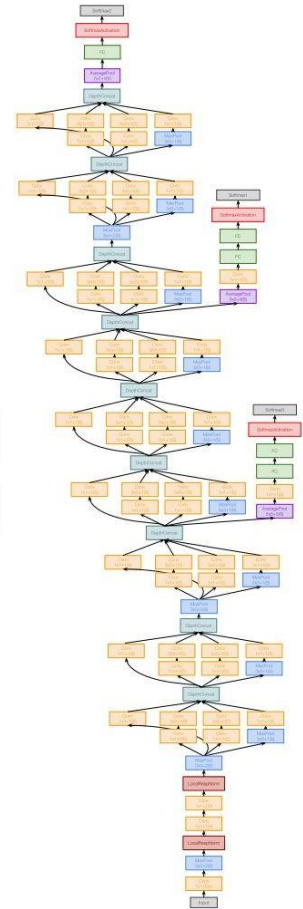
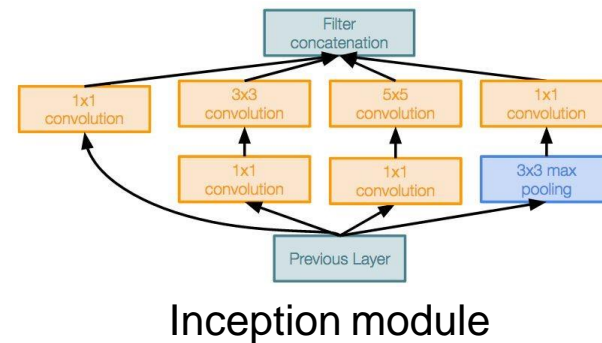


# Case Study: GoogLeNet

[Szegedy et al., 2014]

Deeper networks, with computational efficiency

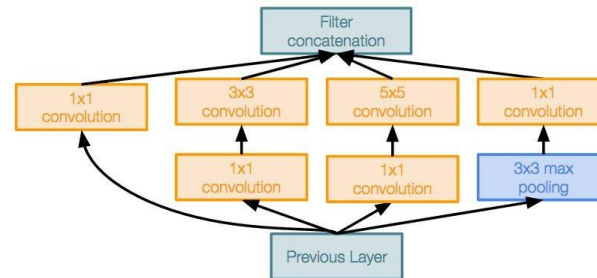
- 22 layers
- Efficient “Inception” module
- No FC layers
- Only 5 million parameters!  
12x less than AlexNet
- ILSVRC’14 classification winner  
(6.7% top 5 error)



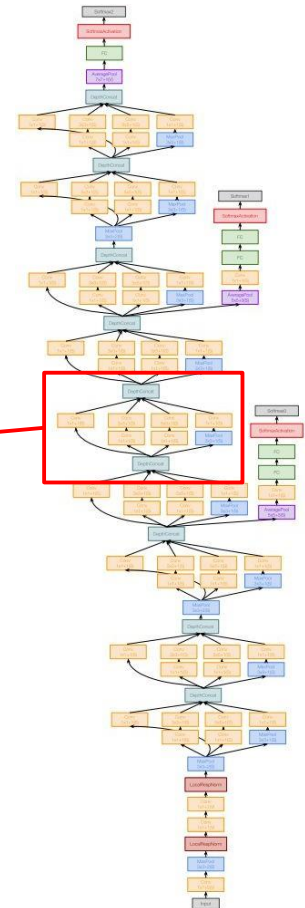
# Case Study: GoogLeNet

[Szegedy et al., 2014]

“Inception module”: design a good local network topology (network within a network) and then stack these modules on top of each other

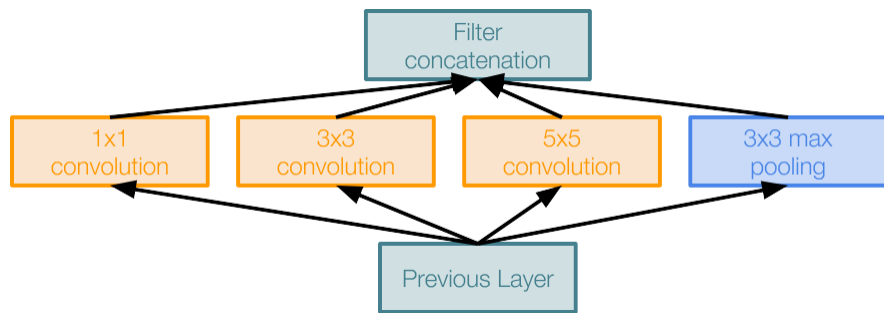


Inception module



# Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together depth-wise

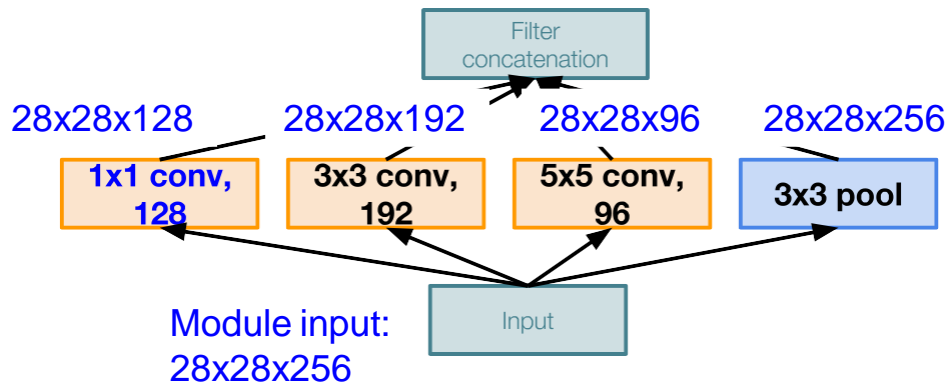
# Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3: What is output size after filter concatenation?

$$28 \times 28 \times (128 + 192 + 96 + 256) = 28 \times 28 \times 672$$



Naive Inception module

Q: What is the problem with this?  
[Hint: Computational complexity]

**Conv Ops:**

[ $1 \times 1$  conv, 128]  $28 \times 28 \times 128 \times 1 \times 1 \times 256$

[ $3 \times 3$  conv, 192]  $28 \times 28 \times 192 \times 3 \times 3 \times 256$

[ $5 \times 5$  conv, 96]  $28 \times 28 \times 96 \times 5 \times 5 \times 256$

**Total: 854M ops**

Very expensive compute

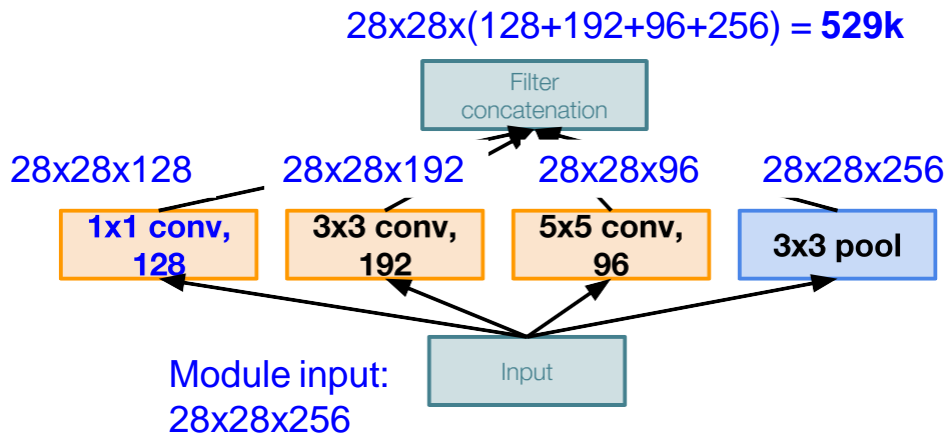
Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

# Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3: What is output size after filter concatenation?



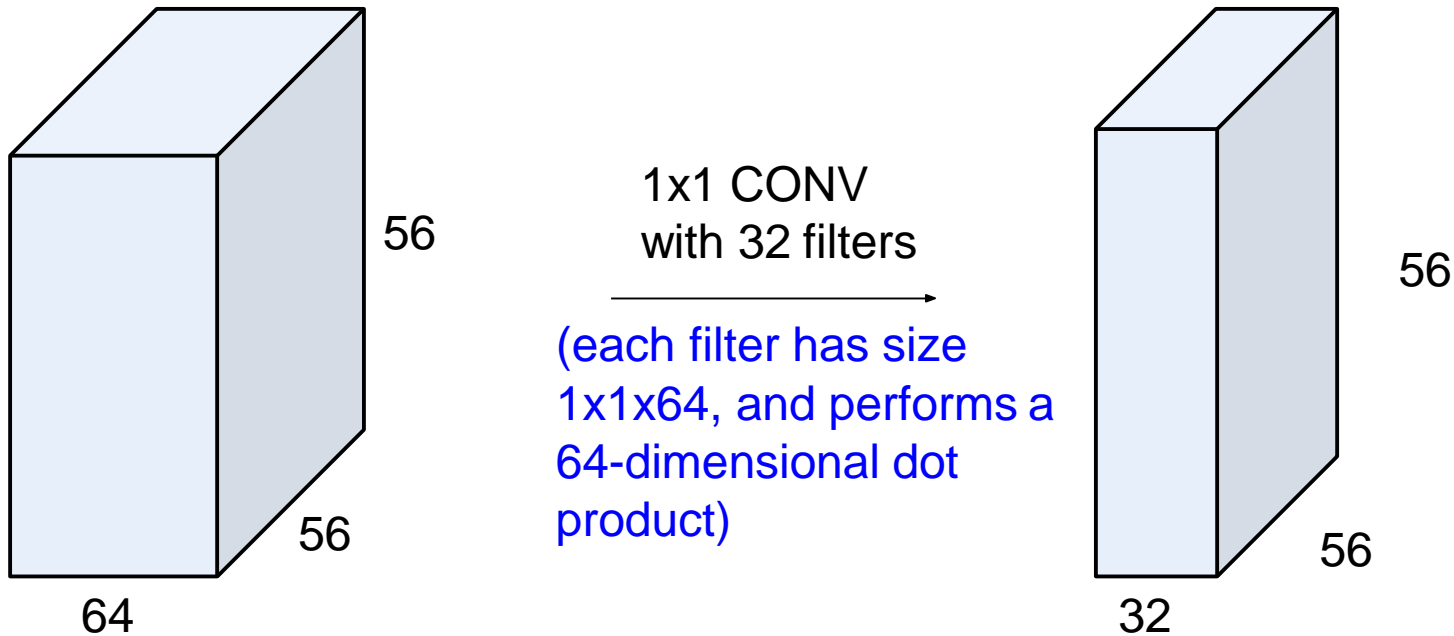
Naive Inception module

Q: What is the problem with this?  
[Hint: Computational complexity]

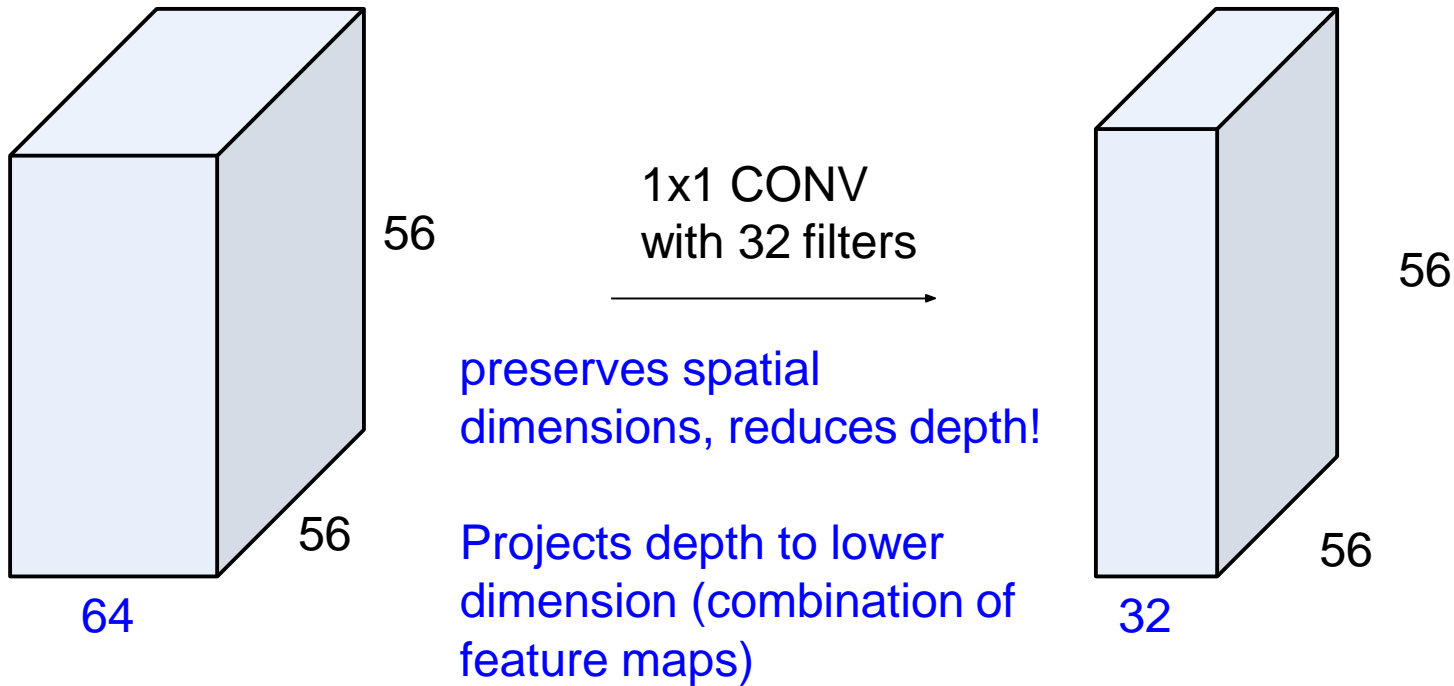
Solution: “bottleneck” layers that use 1x1 convolutions to reduce feature depth



# Reminder: 1x1 convolutions

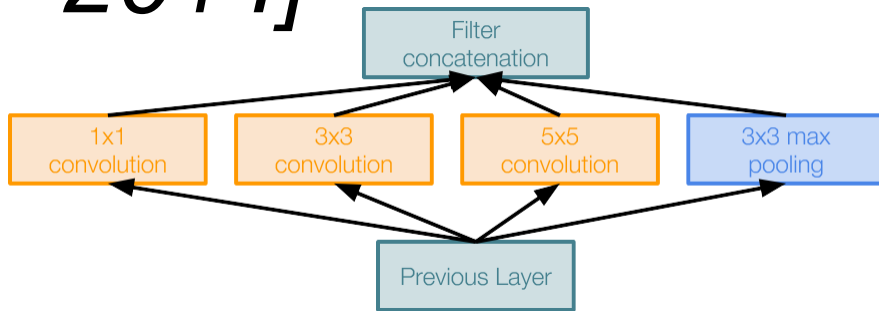


# Reminder: 1x1 convolutions

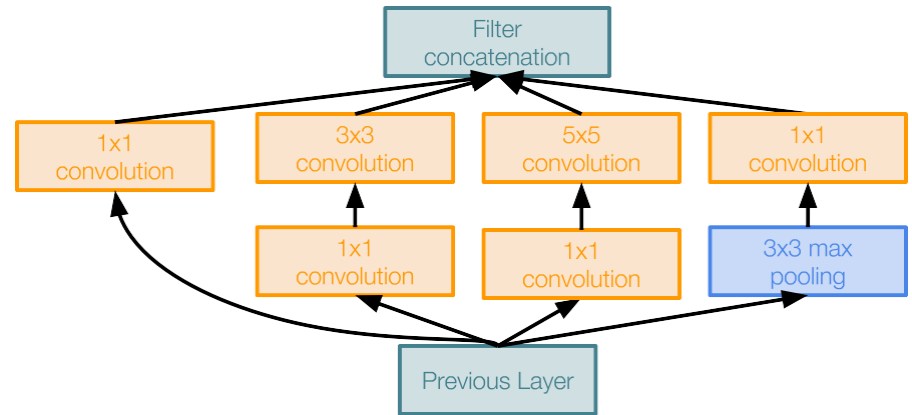


# Case Study: GoogLeNet

*[Szegedy et al., 2014]*



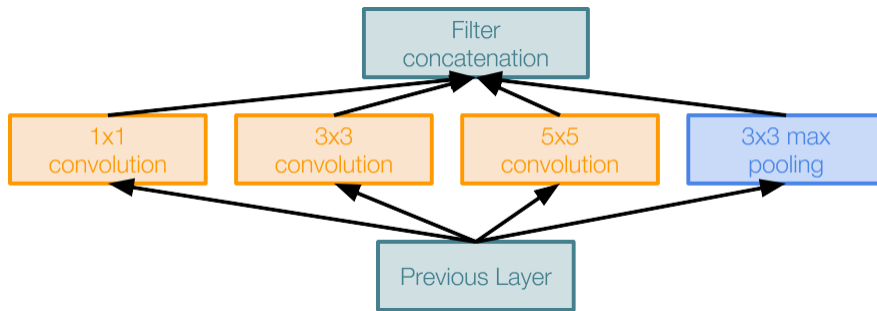
Naive Inception module



Inception module with dimension reduction

# Case Study: GoogLeNet

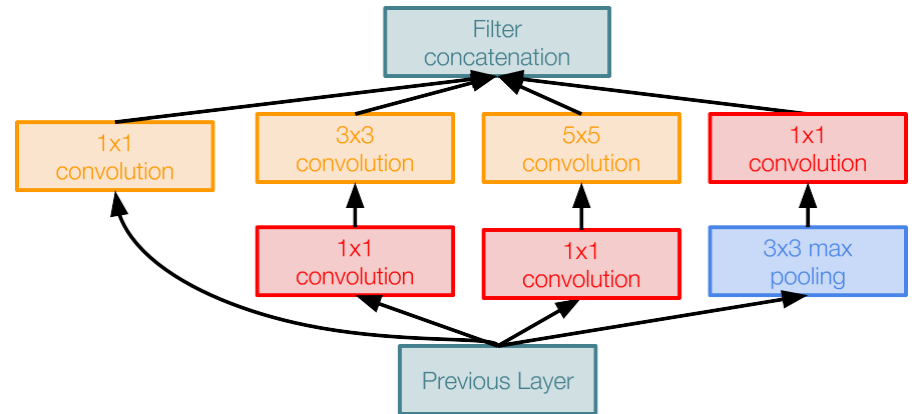
[Szegedy et al., 2014]



Naive Inception module

**Total: 854M ops**

1x1 conv “bottleneck”  
layers



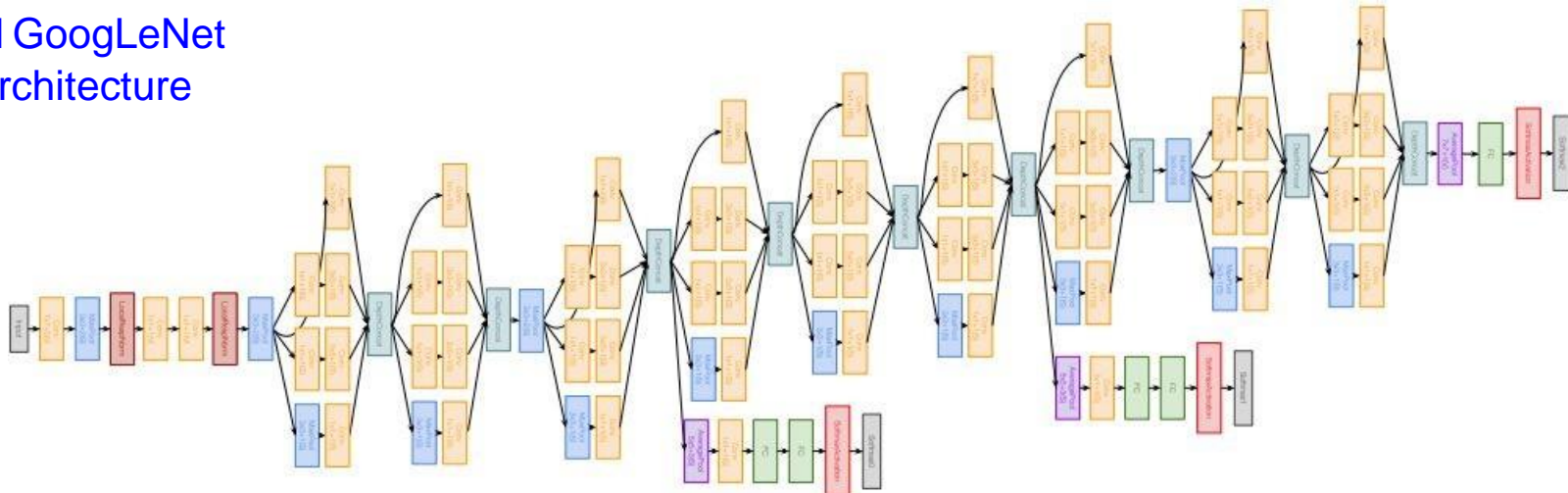
Inception module with dimension reduction

**Total: 358M ops**

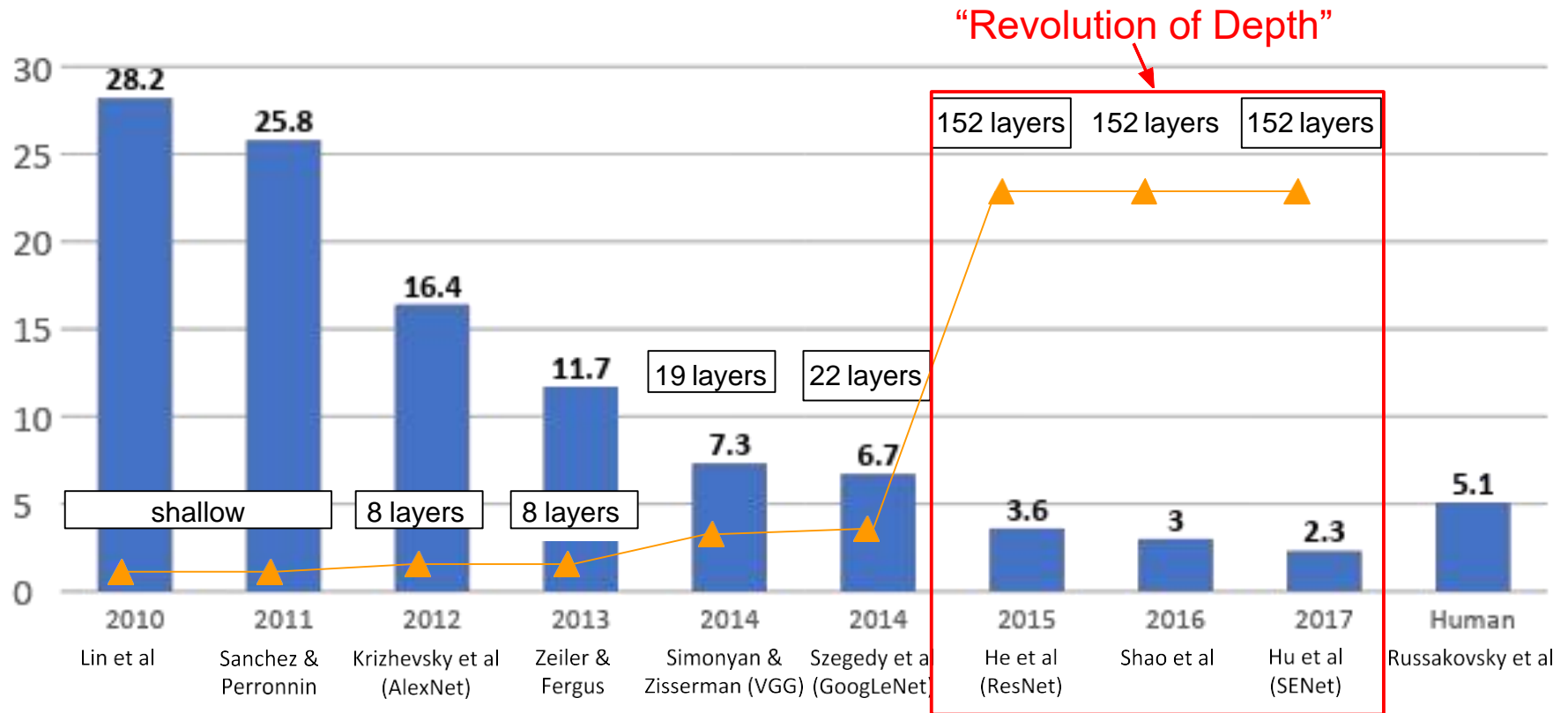
# Case Study: GoogLeNet

[Szegedy et al., 2014]

## Full GoogLeNet architecture



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

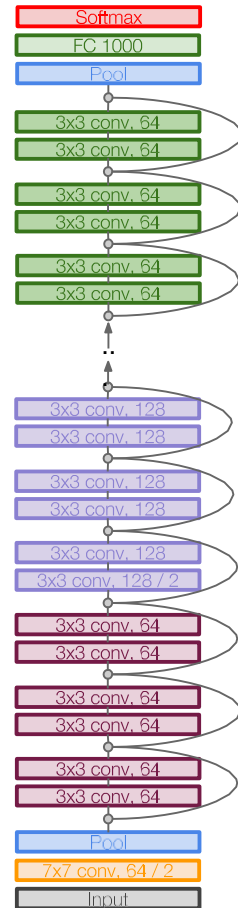
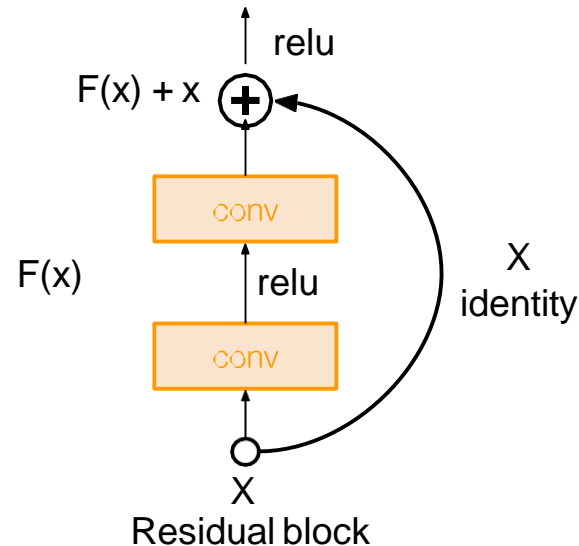


# Case Study: ResNet

[He et al., 2016]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!



# Case Study: ResNet

*[He et al., 2016]*

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



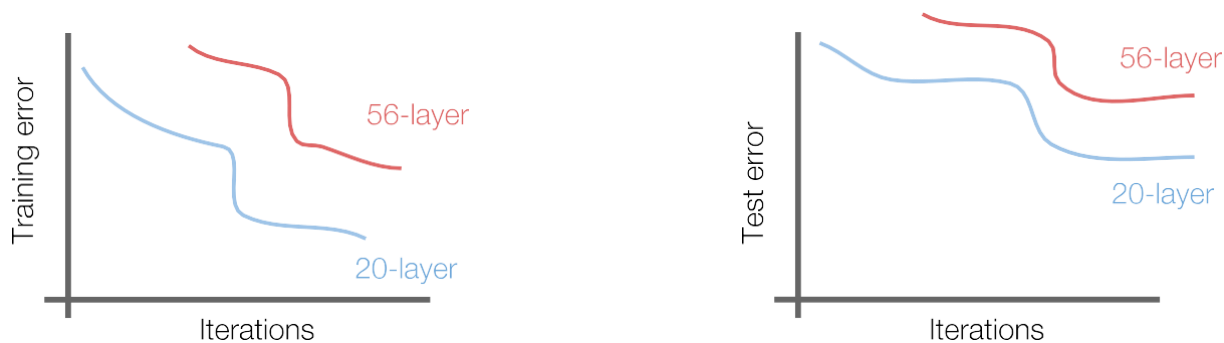
Q: What's strange about these training and test curves?  
[Hint: look at the order of the curves]



# Case Study: ResNet

*[He et al., 2016]*

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



56-layer model performs worse on both training and test error  
-> The deeper model performs worse, but it's not caused by overfitting!

# Case Study: ResNet

[He et al., 2016]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

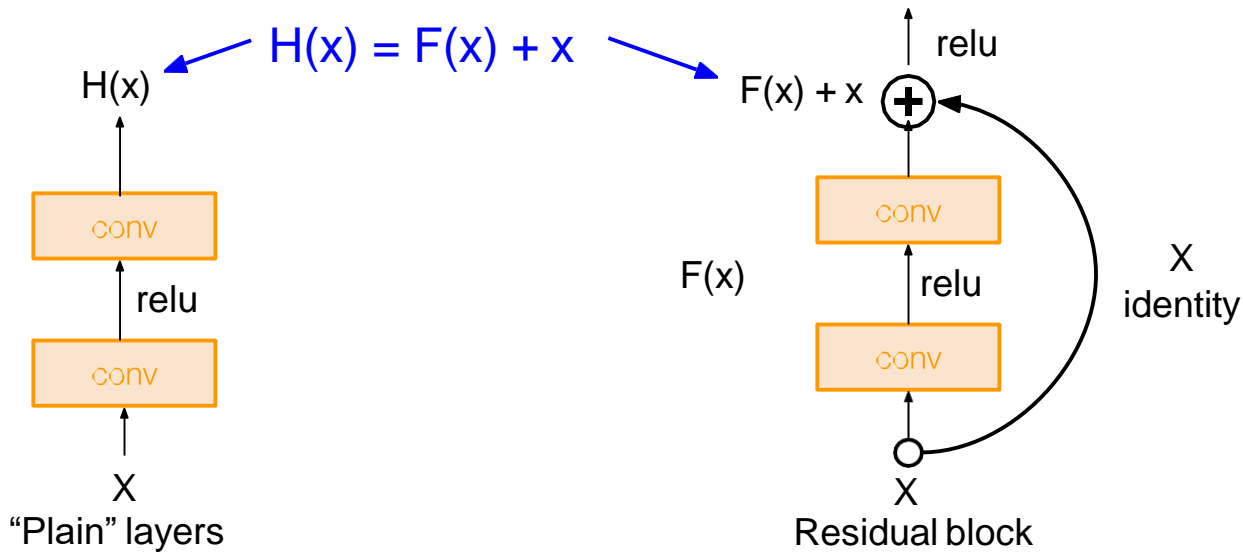
The deeper model should be able to perform at least as well as the shallower model.

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

# Case Study: ResNet

[He et al., 2016]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



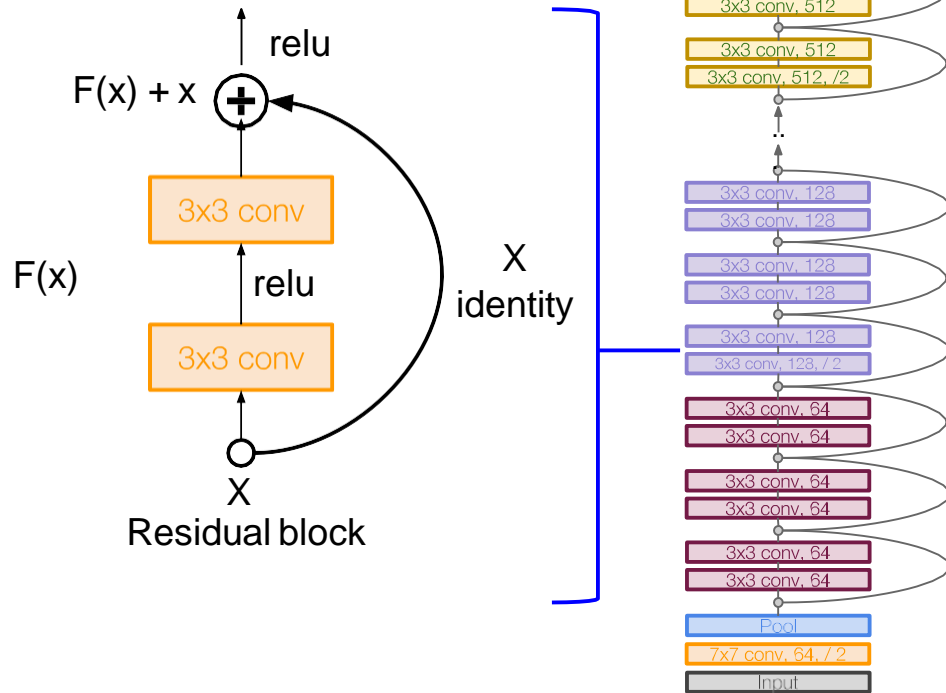
Use layers to  
fit residual  
 $F(x) = H(x) - x$   
instead of  
 $H(x)$  directly

# Case Study: ResNet

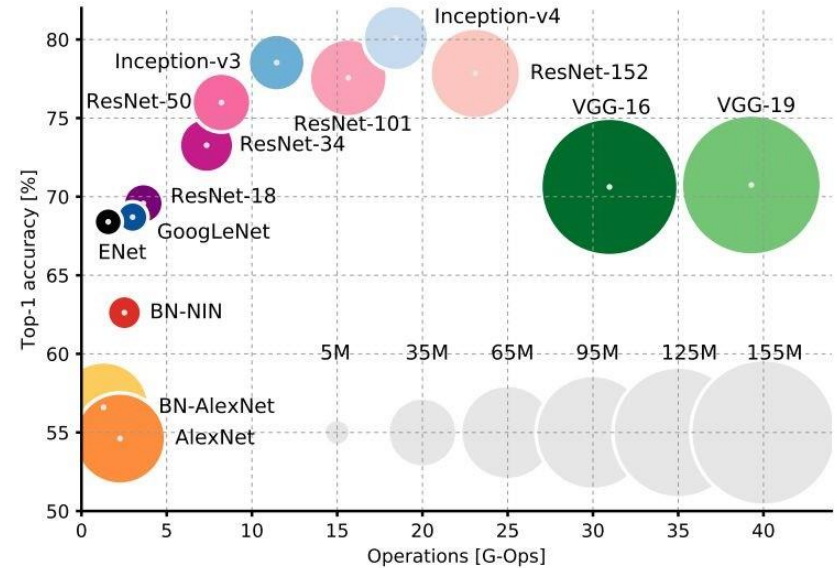
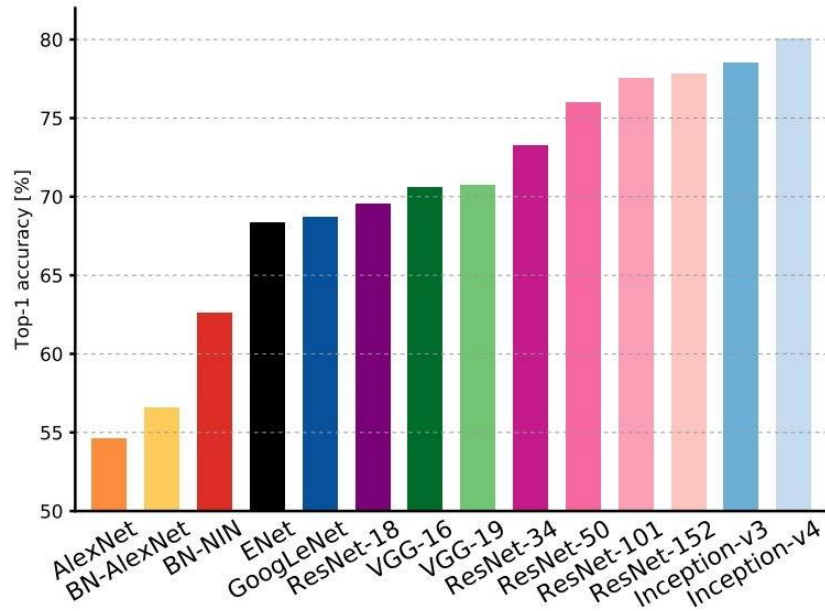
[He et al., 2016]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers



# Comparing complexity...



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

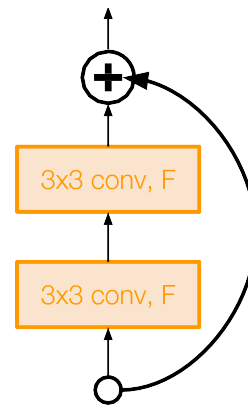
Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

## Improving ResNets...

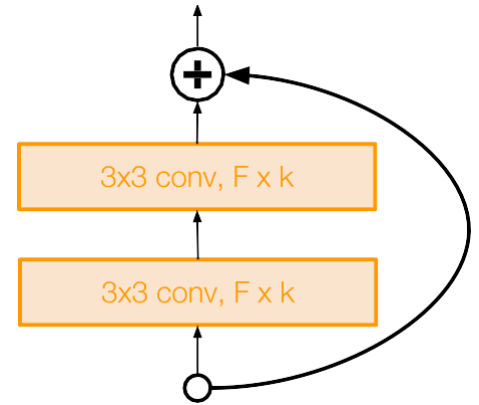
# Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- User wider residual blocks ( $F \times k$  filters instead of  $F$  filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)



Basic residual block



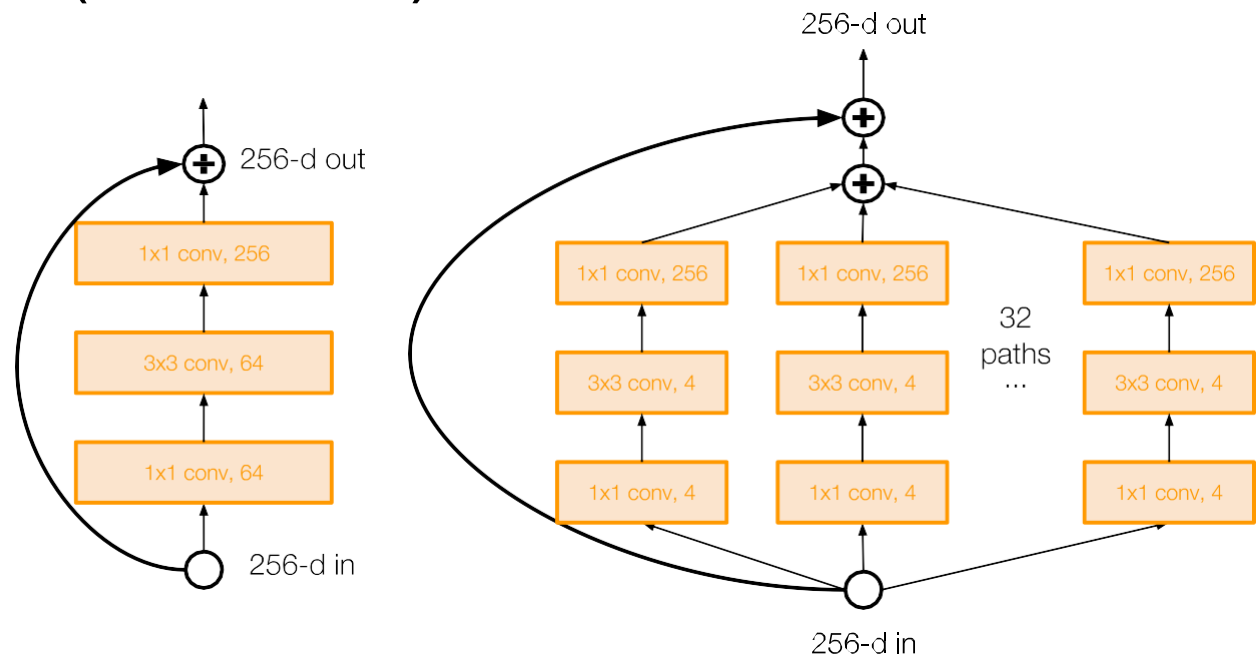
Wide residual block

# Improving ResNets...

## Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways (“cardinality”)
- Parallel pathways similar in spirit to Inception module

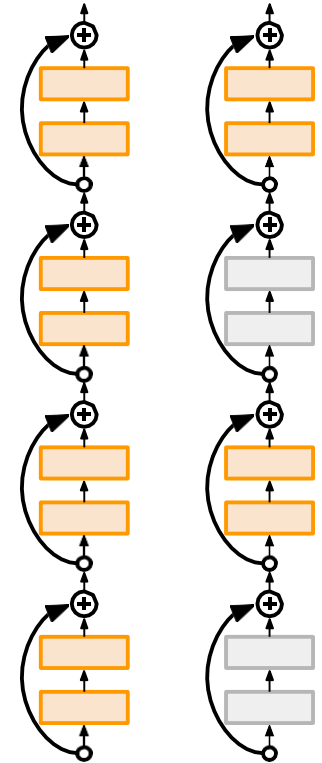


## Improving ResNets...

# Deep Networks with Stochastic Depth

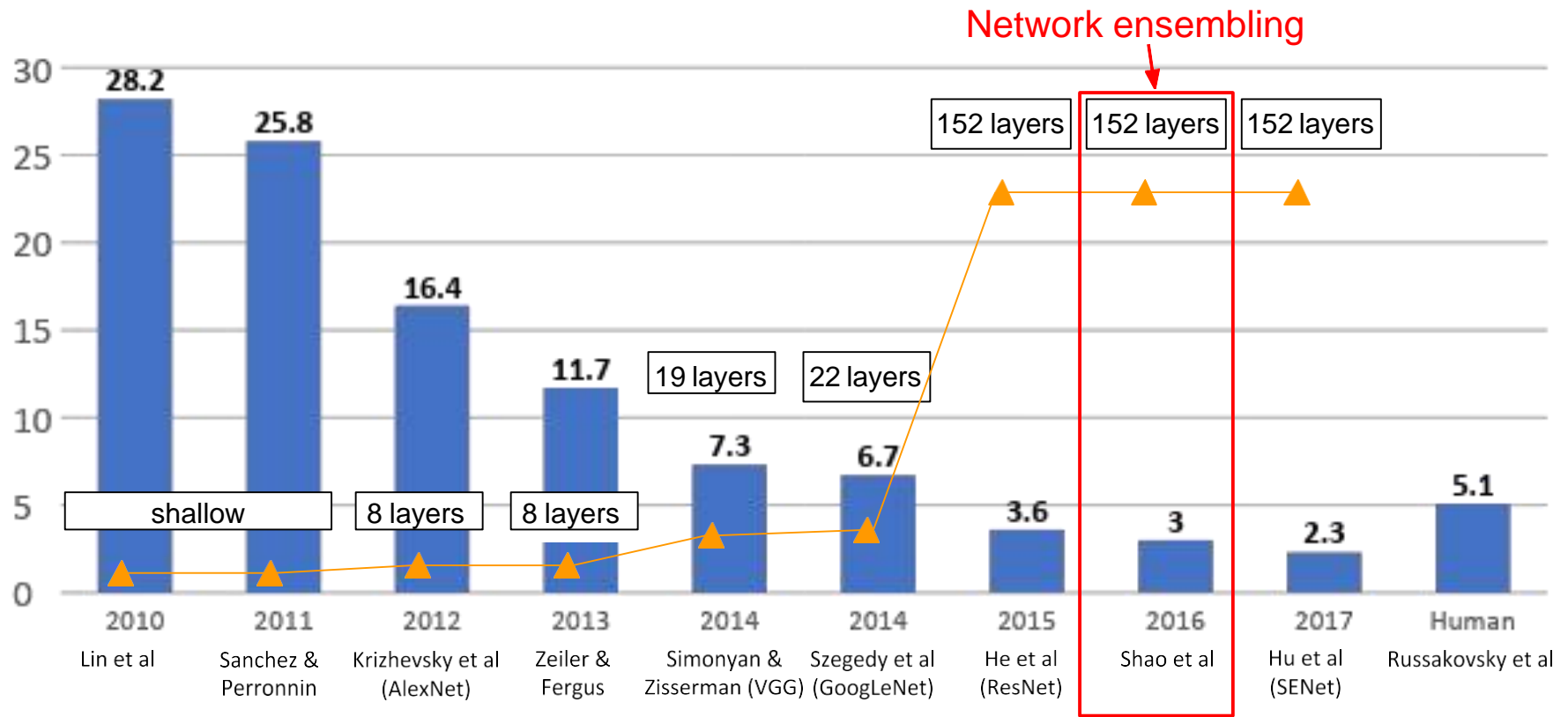
[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time

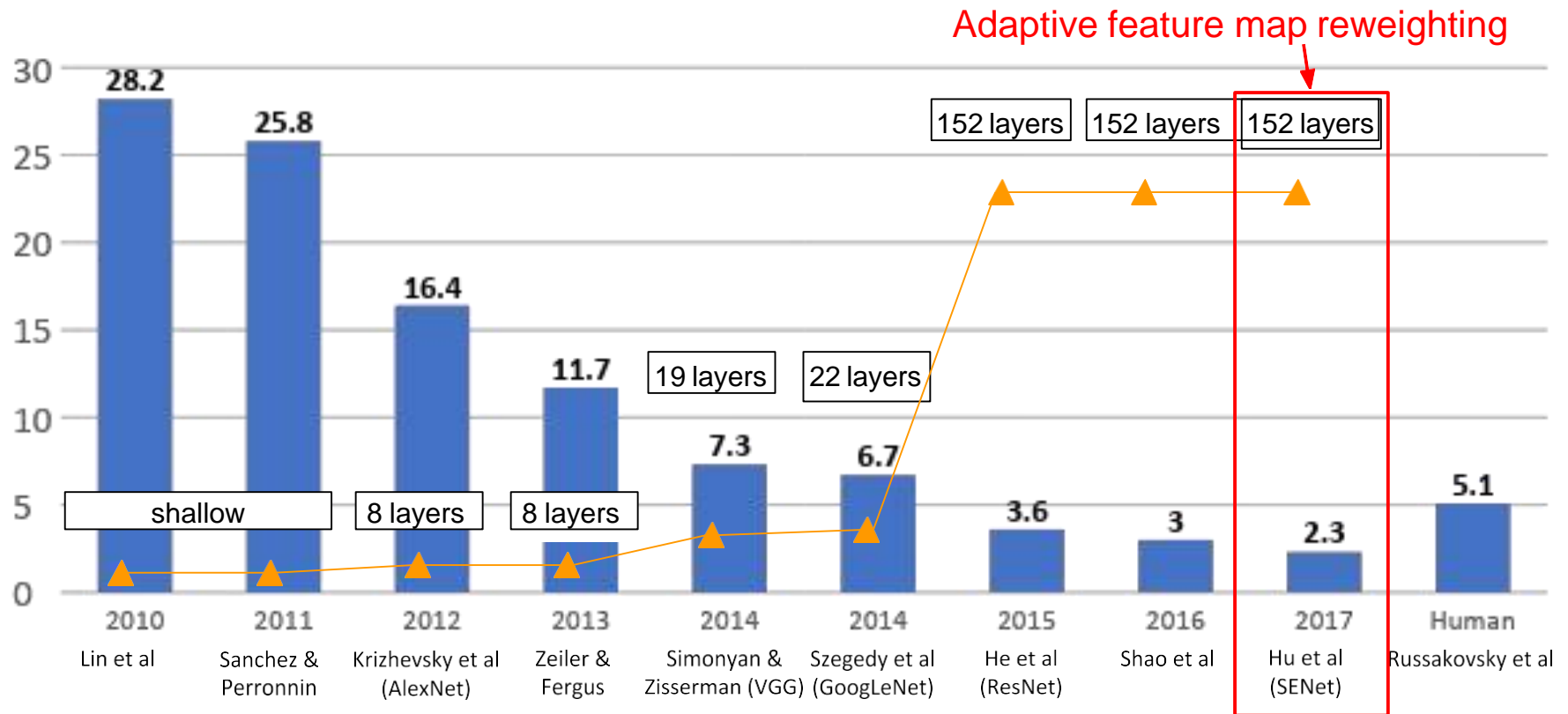




# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

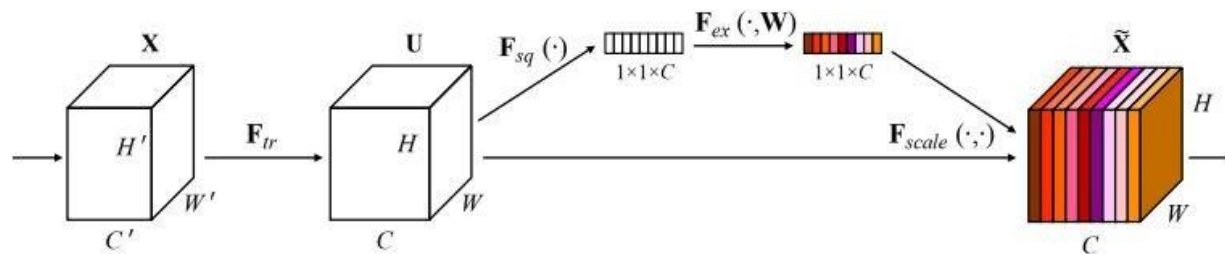
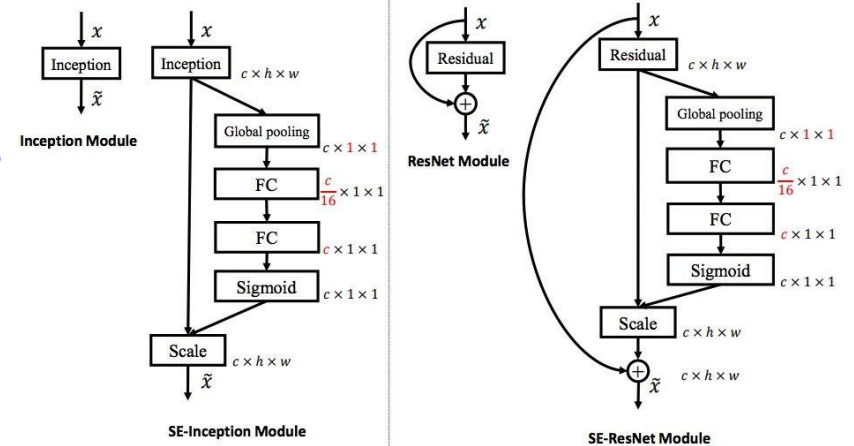


## Improving ResNets...

# Squeeze-and-Excitation Networks (SENet)

[Hu et al. 2017]

- Add a “feature recalibration” module that learns to adaptively reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC'17 classification winner (using ResNeXt-152 as a base architecture)

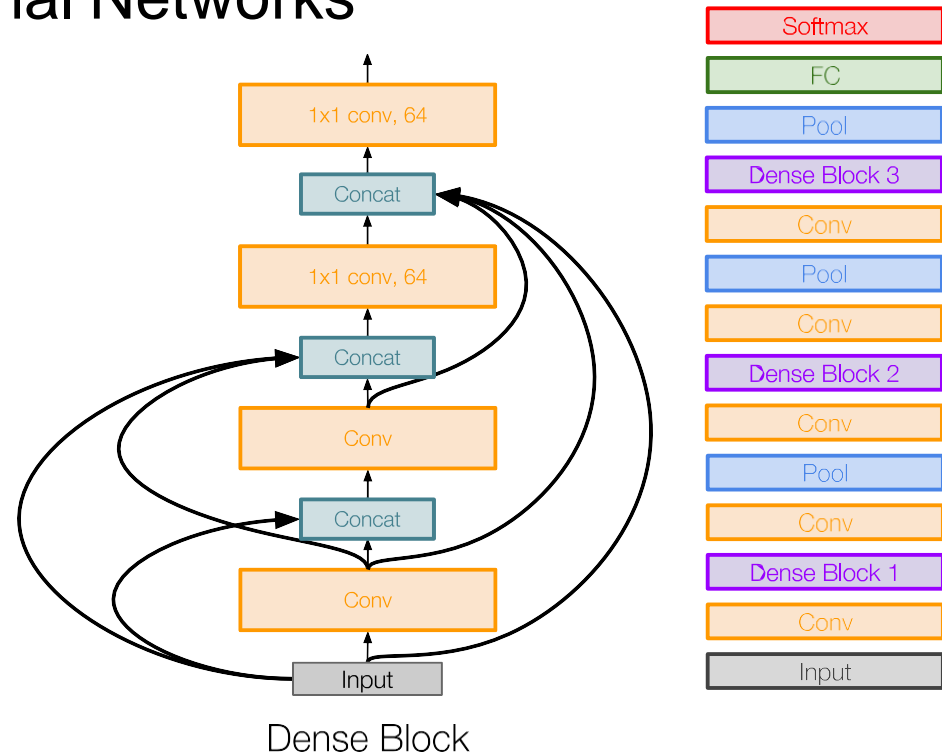


# Beyond ResNets...

## Densely Connected Convolutional Networks

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse



# Efficient networks...

## SqueezeNet: AlexNet-level Accuracy With 50x Fewer Parameters and <0.5Mb Model Size

[Iandola et al. 2017]

- Fire modules consisting of a 'squeeze' layer with 1x1 filters feeding an 'expand' layer with 1x1 and 3x3 filters
- AlexNet level accuracy on ImageNet with 50x fewer parameters
- Can compress to 510x smaller than AlexNet (0.5Mb)

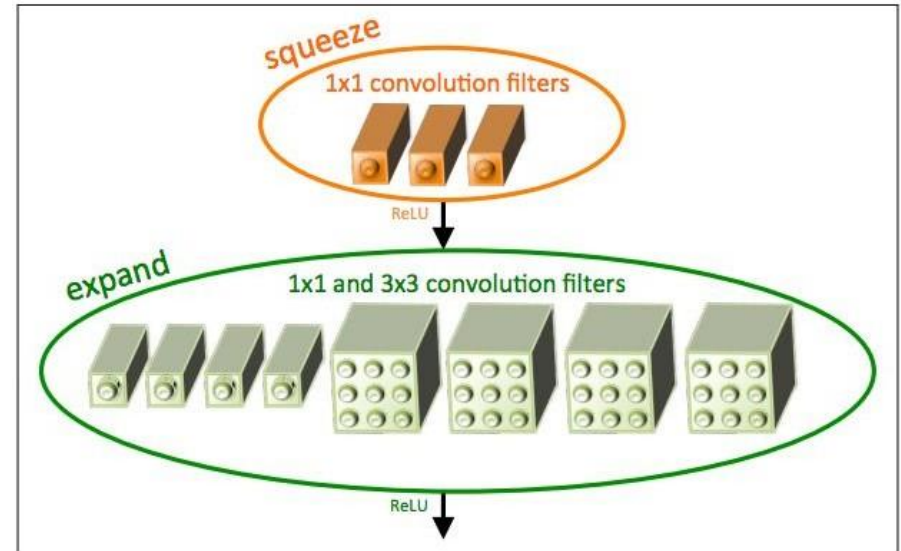


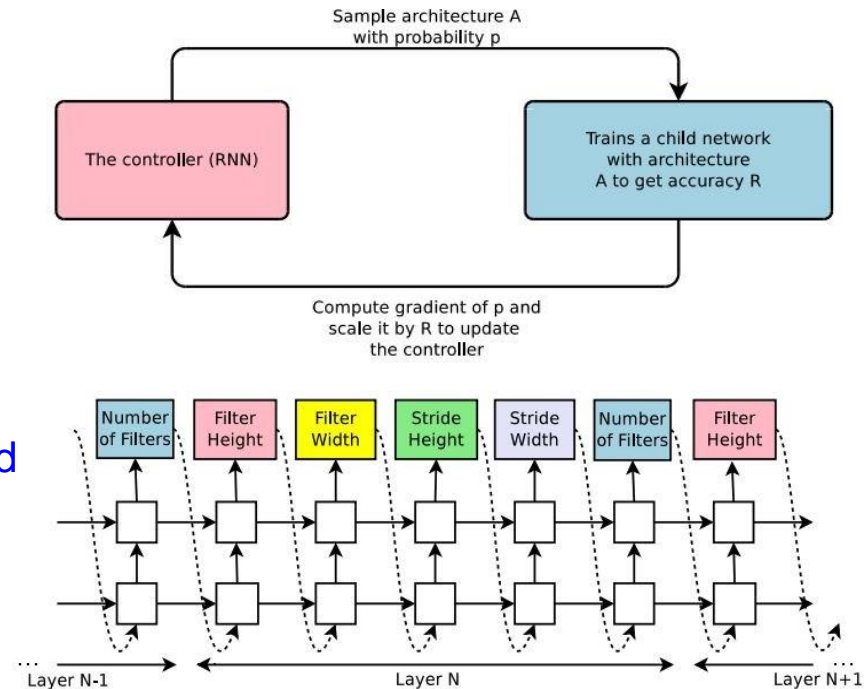
Figure copyright Iandola, Han, Moskewicz, Ashraf, Dally, Keutzer, 2017. Reproduced with permission.

# Meta-learning: Learning to learn network architectures...

## Neural Architecture Search with Reinforcement Learning (NAS)

[Zoph et al. 2016]

- “Controller” network that learns to design a good network architecture (output a string corresponding to network design)
- Iterate:
  - 1) Sample an architecture from search space
  - 2) Train the architecture to get a “reward”  $R$  corresponding to accuracy
  - 3) Compute gradient of sample probability, and scale by  $R$  to perform controller parameter update (i.e. increase likelihood of good architecture being sampled, decrease likelihood of bad architecture)



# Summary: CNN Architectures

## Case Studies

- AlexNet
- VGG
- GoogLeNet
- ResNet

## Also....

- Wide ResNet
- ResNeXT
- DenseNet
- Squeeze-and-Excitation Network

# Summary: CNN Architectures

- VGG, GoogLeNet, ResNet all in wide use, available in model zoos
- ResNet current best default, also consider SENet when available
- Trend towards extremely deep networks
- Significant research centers around design of layer / skip connections and improving gradient flow
- Efforts to investigate necessity of depth vs. width and residual connections
- Even more recent trend towards meta-learning



Practical matters

# Plan for the rest of the lecture

---

## Neural network basics

- Definition
- Loss functions
- Optimization w/ gradient descent and backpropagation

## Convolutional neural networks (CNNs)

- Special operations
- Common architectures

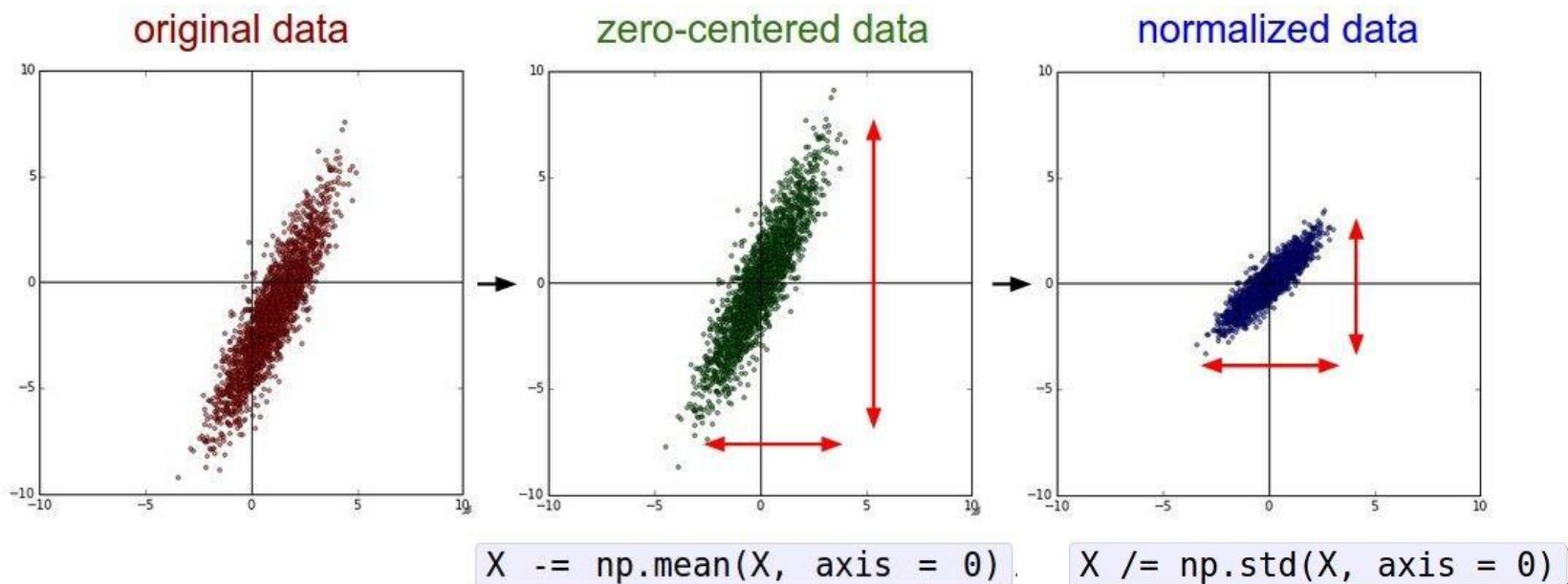
## Practical matters

- Getting started: Preprocessing, initialization, optimization, normalization
- Improving performance: regularization, augmentation, transfer
- Hardware and software

## Understanding CNNs

- Visualization
- Breaking CNNs

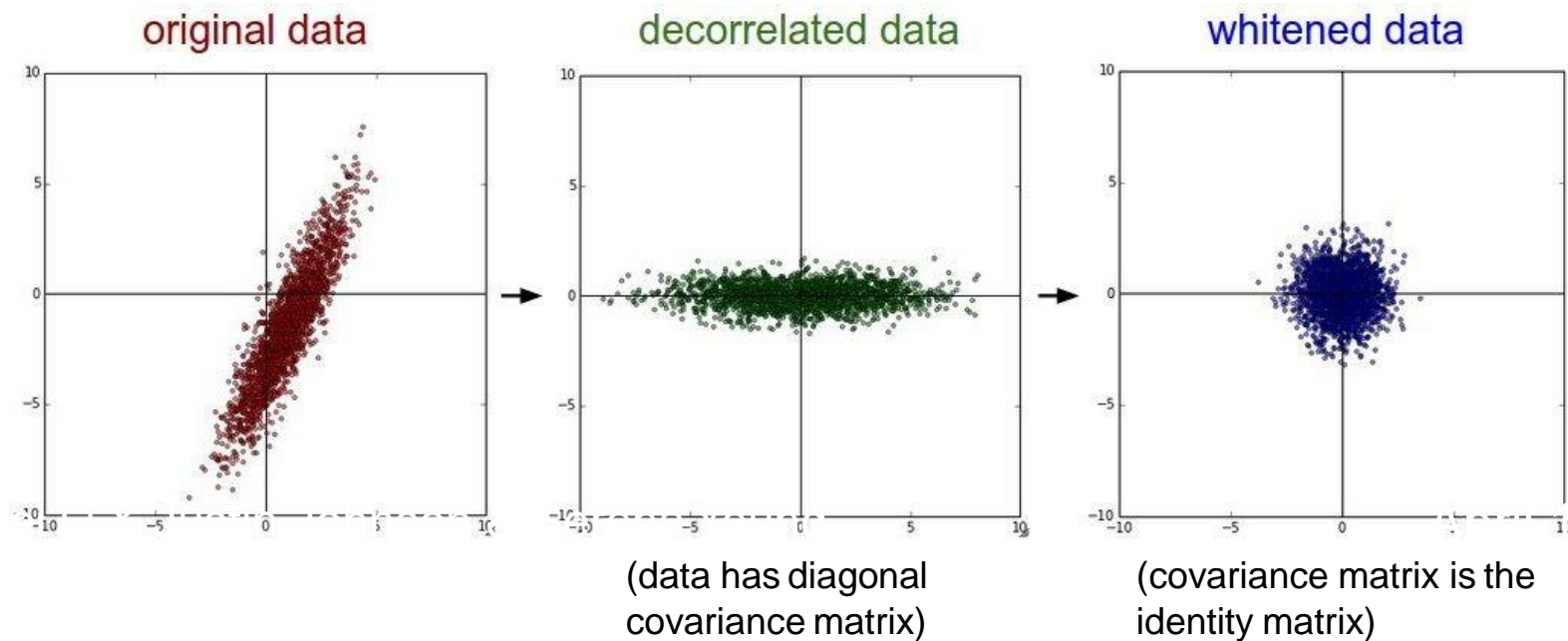
# Preprocessing the Data



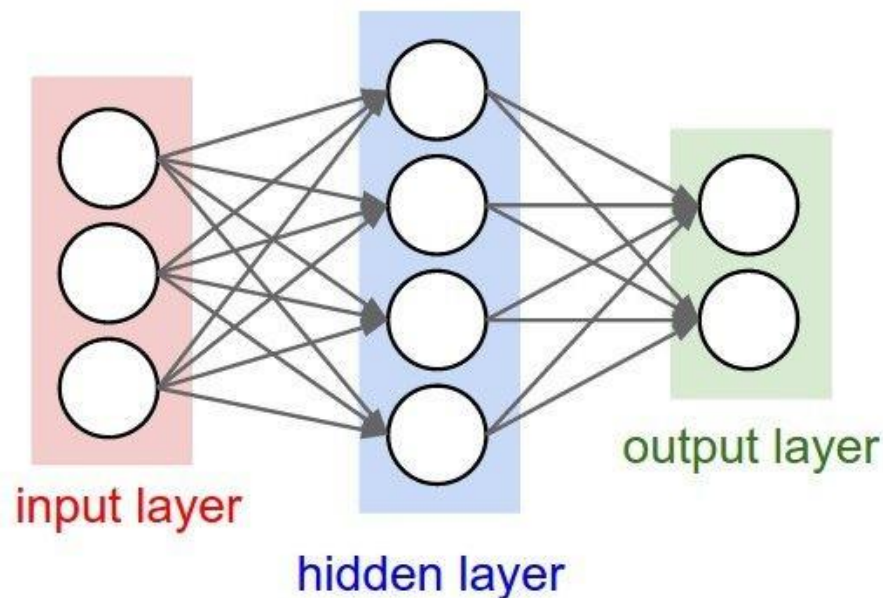
(Assume  $X$  [NxD] is data matrix,  
each example in a row)

# Preprocessing the Data

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



# Weight Initialization



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

# Weight Initialization

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

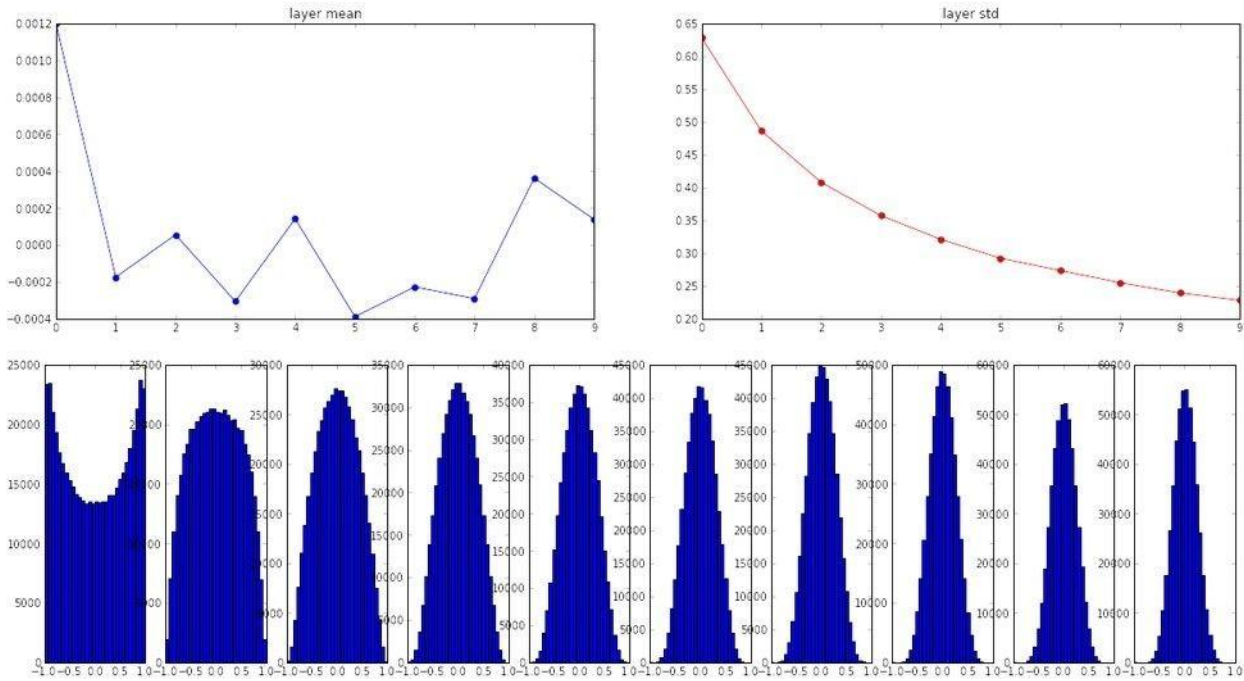
```
W = 0.01* np.random.randn(D,H)
```

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

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)

# 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:

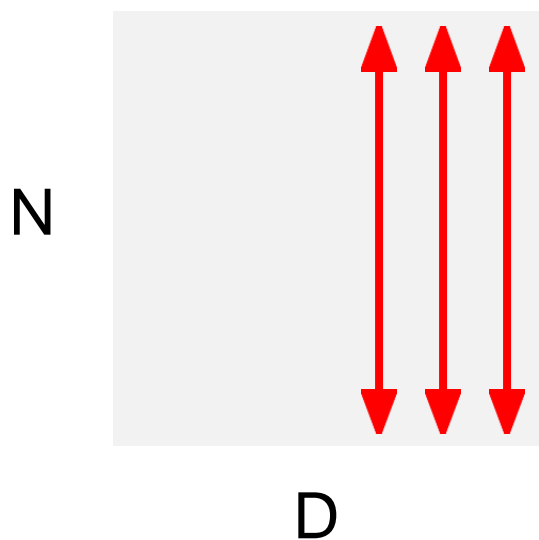
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$



# Batch Normalization

[Ioffe and Szegedy, 2015]

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



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

2. Normalize

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

# Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

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

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

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

Note, the network can learn:

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

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

to recover the identity mapping.

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;

Parameters to be learned:  $\gamma, \beta$

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

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

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

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

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

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;

Parameters to be learned:  $\gamma, \beta$

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

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

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

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

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

**Note:** at test time BatchNorm layer functions differently:

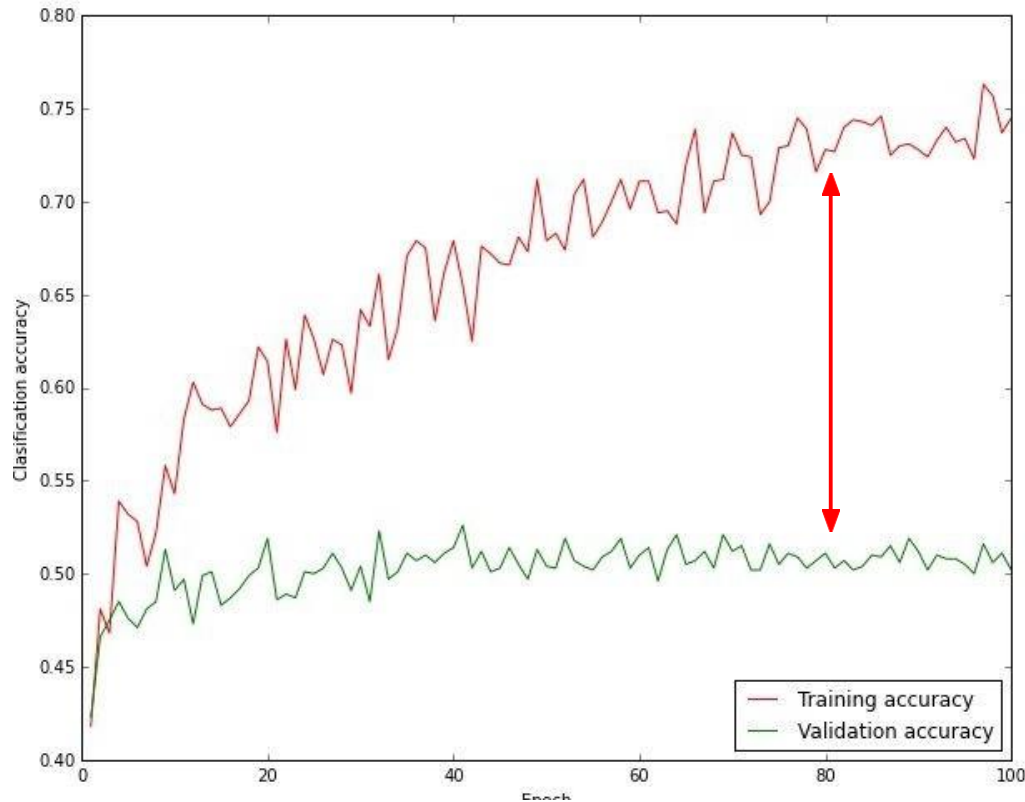
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

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

# Babysitting the Learning Process

- Preprocess data
- Choose architecture
- Initialize and check initial loss with no regularization
- Increase regularization, loss should increase
- Then train – try small portion of data, check you can overfit
- Add regularization, and find learning rate that can make the loss go down
- Check learning rates in range  $[1e-3 \dots 1e-5]$
- Coarse-to-fine search for hyperparameters (e.g. learning rate, regularization)

# Monitor and visualize accuracy



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

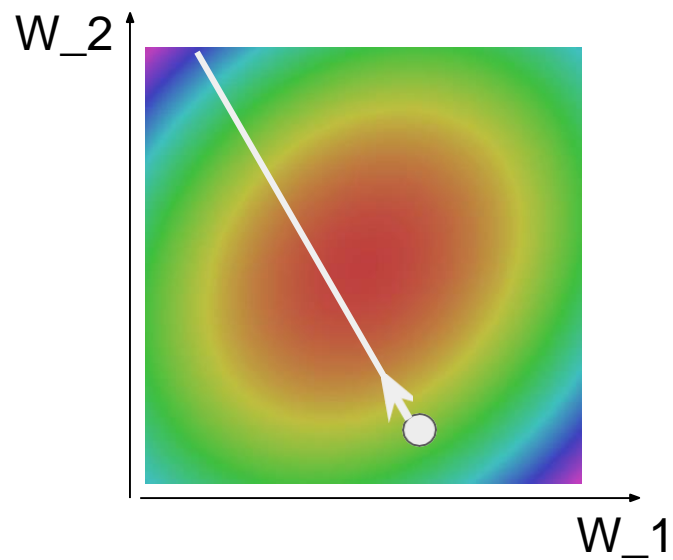
# Optimization

```
# Vanilla Gradient Descent
```

```
while True:
```

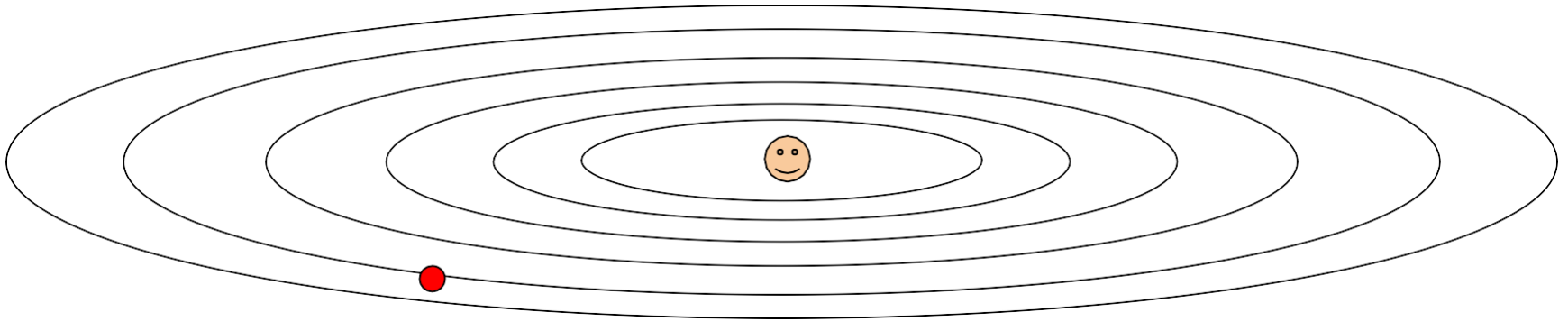
```
    weights_grad = evaluate_gradient(loss_fun, data, weights)
```

```
    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?



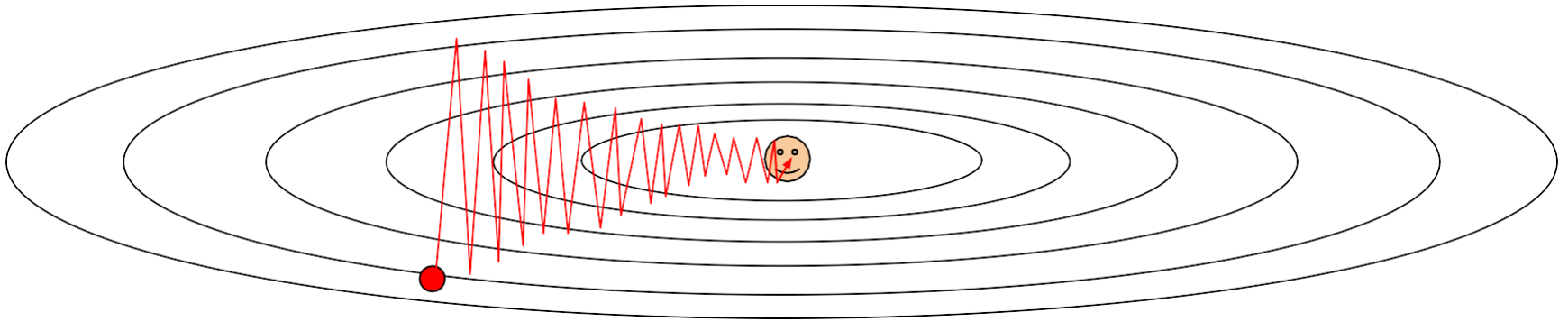
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large



# 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

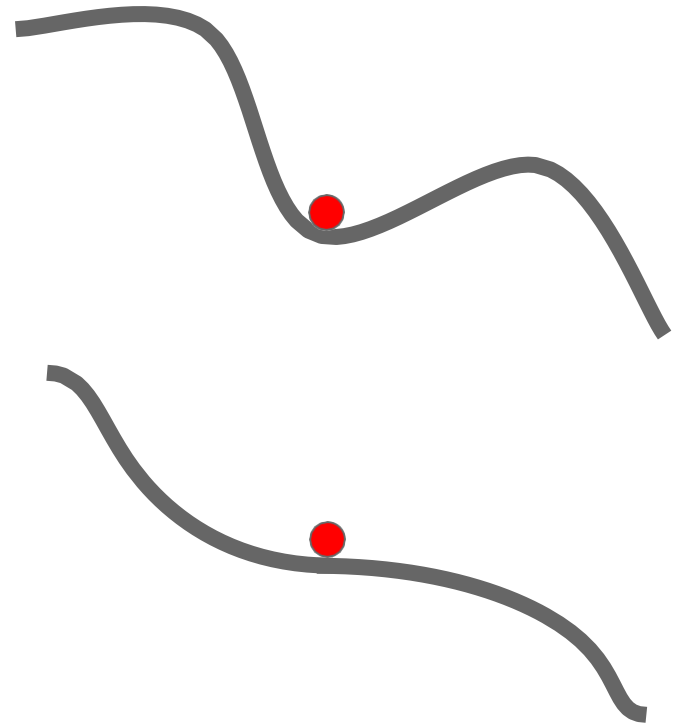


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

# Optimization: Problems with SGD

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

Zero gradient,  
gradient descent  
gets stuck

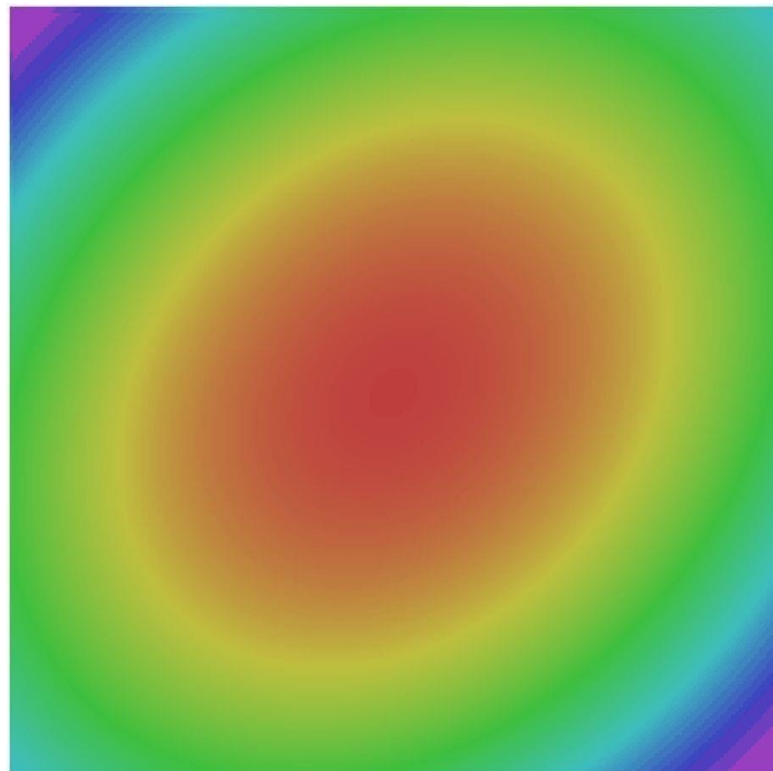


# Optimization: Problems with SGD

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

# AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

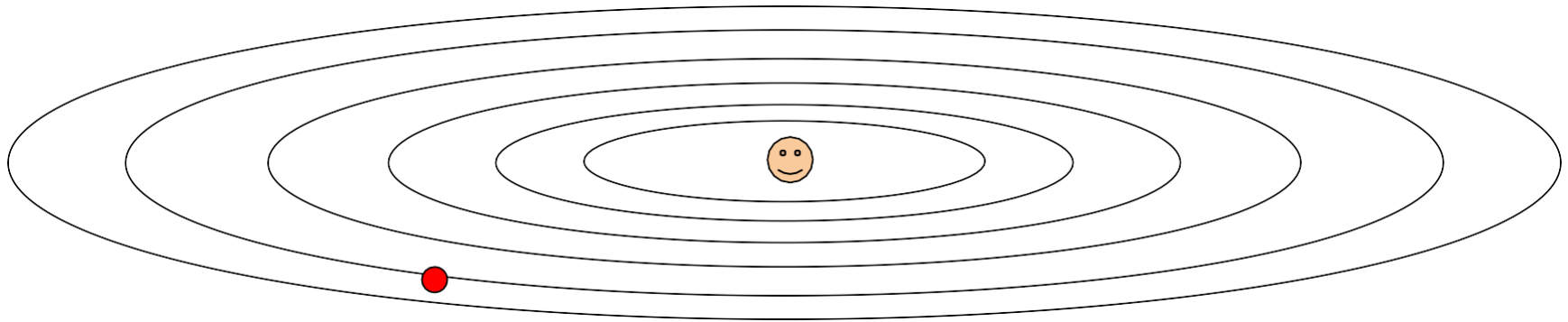
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

“Per-parameter learning rates”  
or “adaptive learning rates”

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

# AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

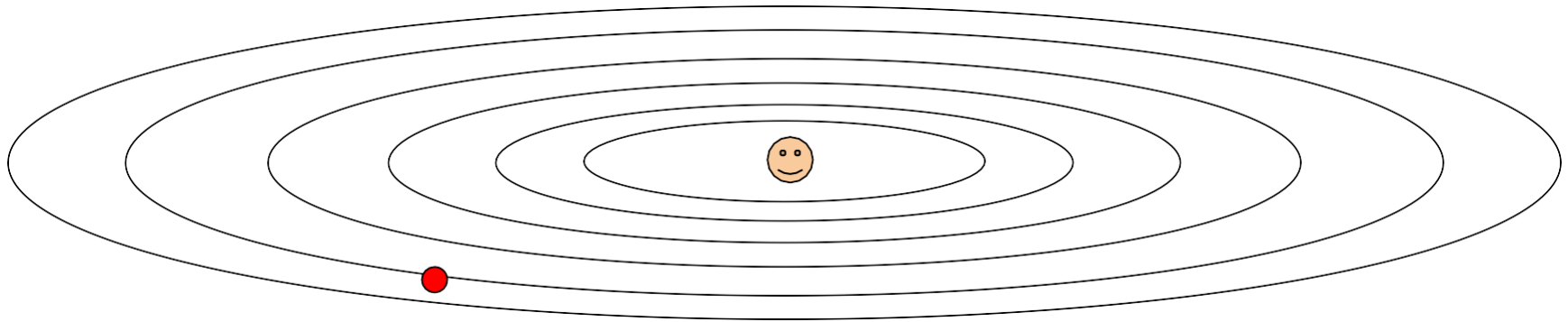


Q: What happens with AdaGrad?

Progress along “steep” directions is damped;  
progress along “flat” directions is accelerated

# AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q2: What happens to the step size over long time?

# RMSProp

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



RMSProp

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012



# Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

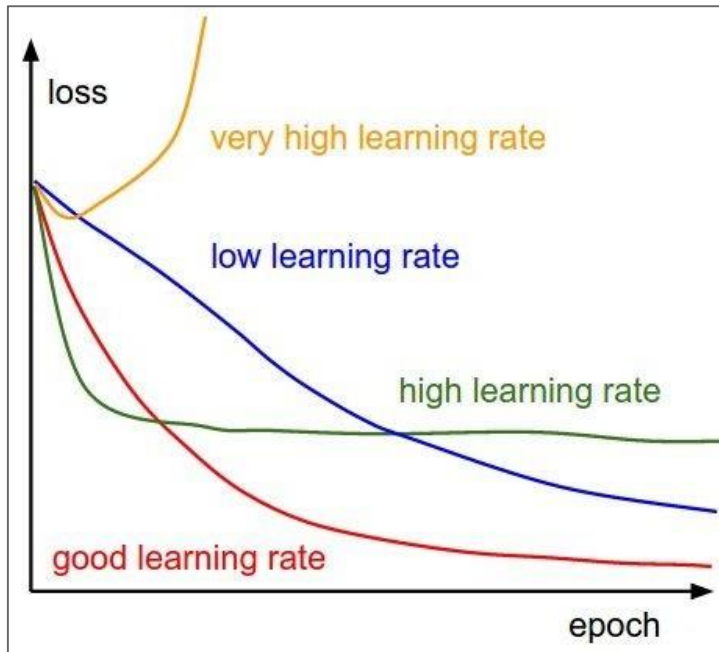
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ , and  $\text{learning\_rate} = 1\text{e-}3$  or  $5\text{e-}4$  is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

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.

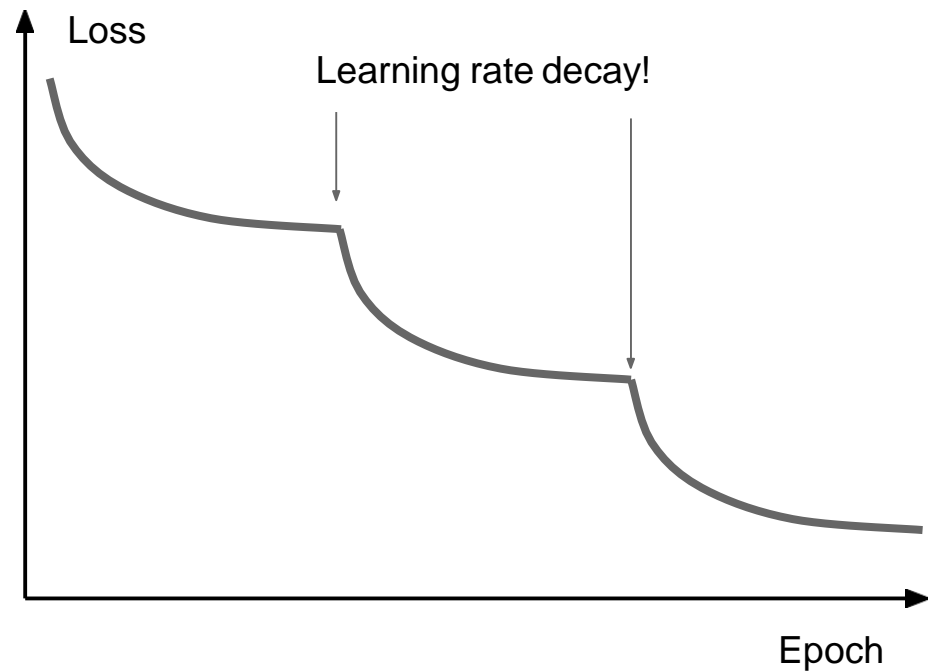
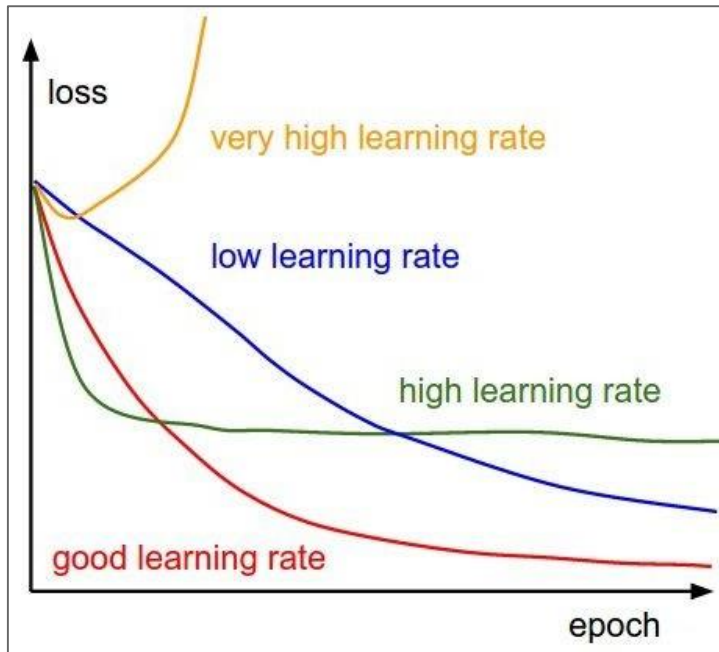
**exponential decay:**

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

**1/t decay:**

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

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



# Plan for the rest of the lecture

---

## Neural network basics

- Definition
- Loss functions
- Optimization w/ gradient descent and backpropagation

## Convolutional neural networks (CNNs)

- Special operations
- Common architectures

## Practical matters

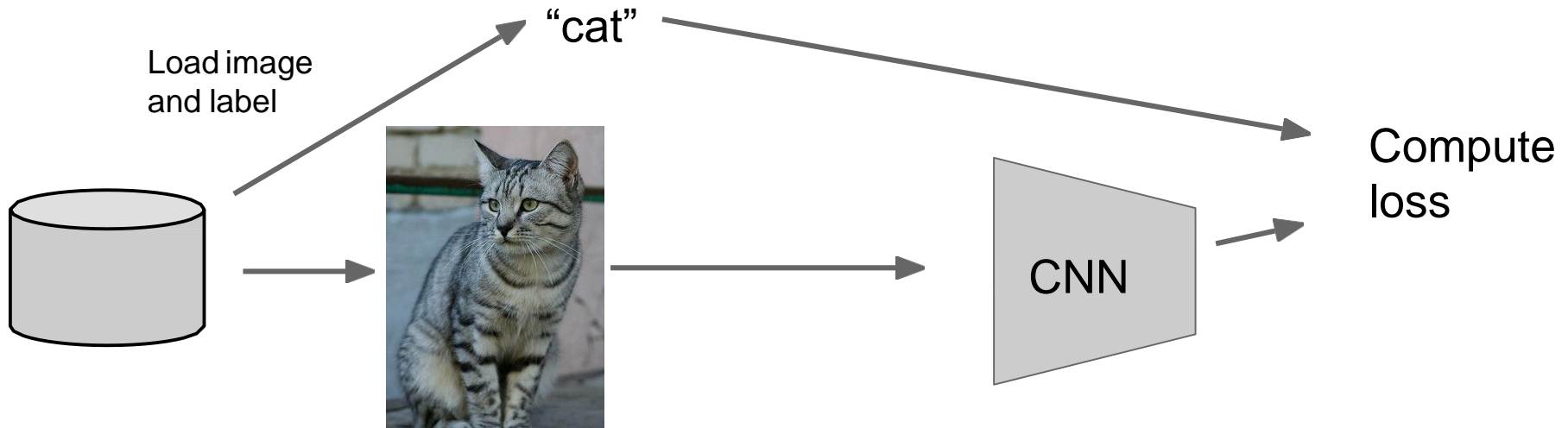
- Getting started: Preprocessing, initialization, optimization, normalization
- Improving performance: regularization, augmentation, transfer
- Hardware and software

## Understanding CNNs

- Visualization
- Breaking CNNs

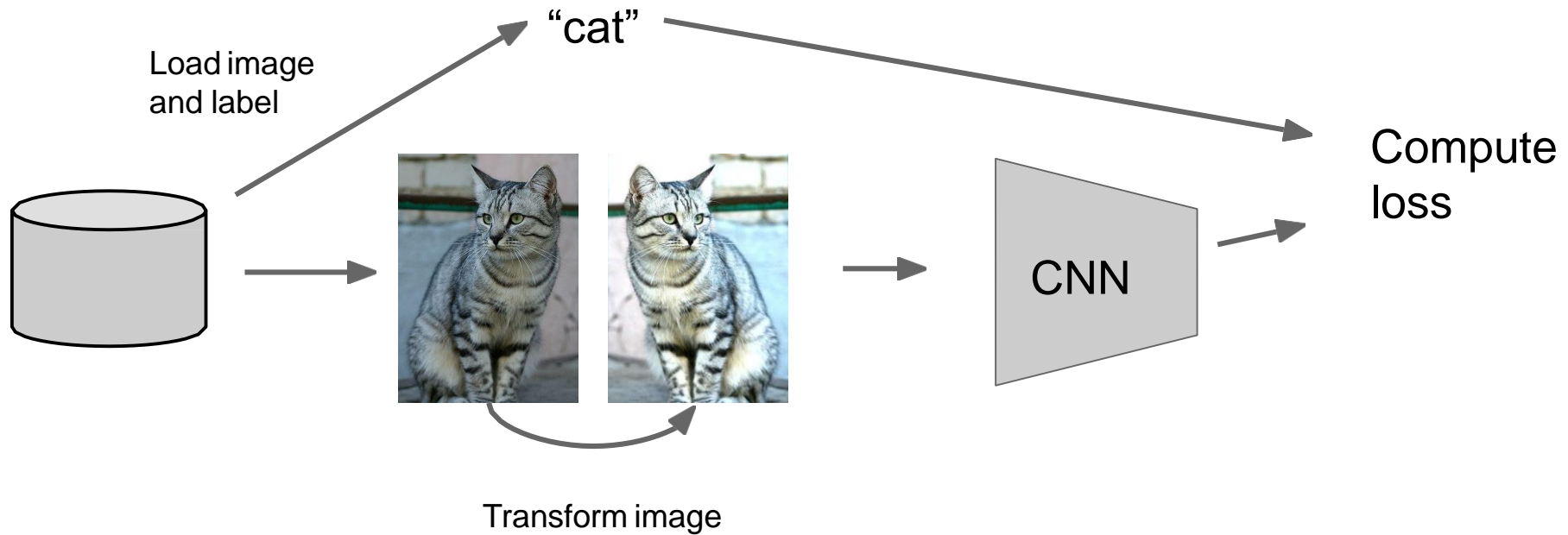
# Data Augmentation

---



# Data Augmentation

---



# Data Augmentation

---

## Horizontal Flips



# Data Augmentation

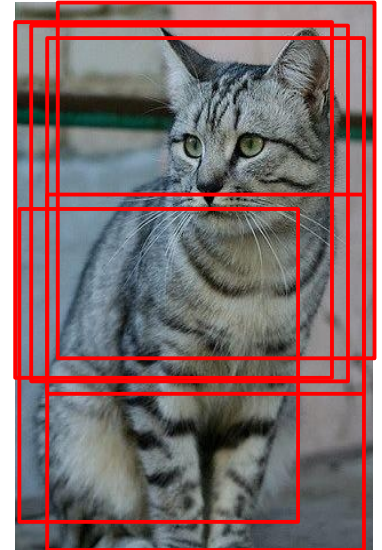
---

## 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

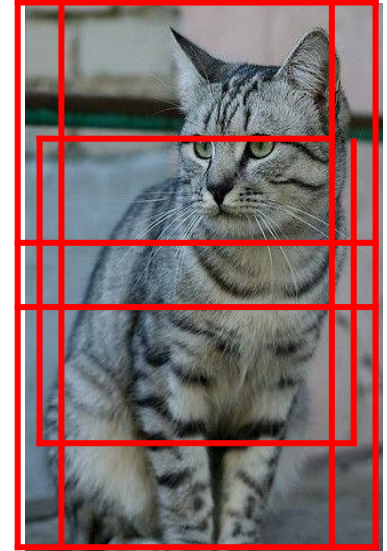
ResNet:

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



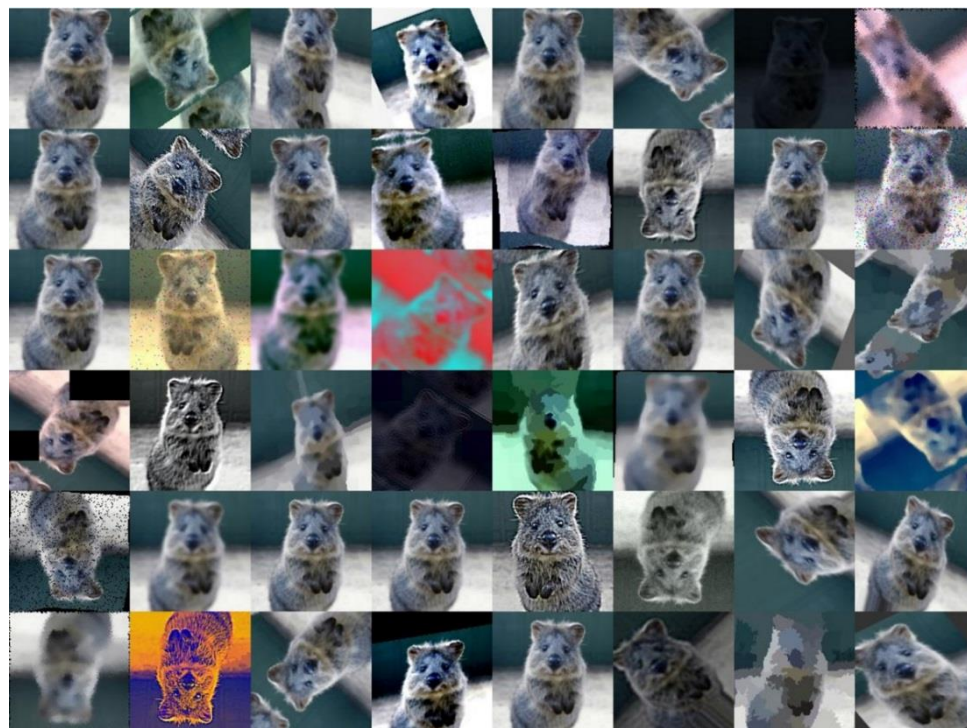
# Data Augmentation

---

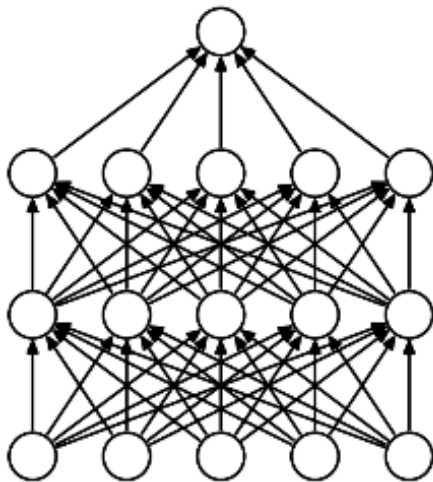
Get creative for your problem!

Random mix/combinations of :

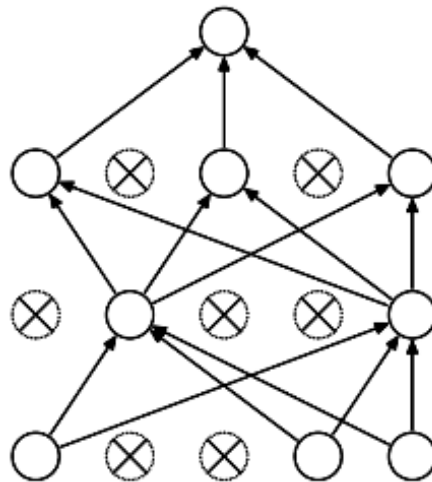
- translation
- rotation
- stretching
- shearing,
- lens distortions
- ...



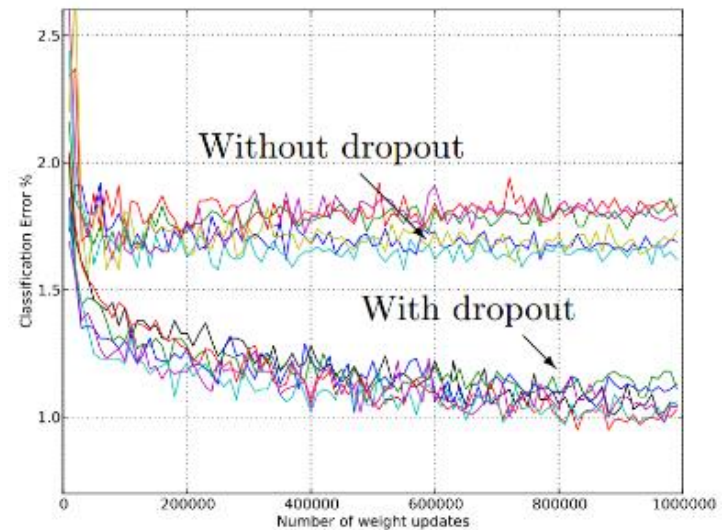
# Regularization: Dropout



(a) Standard Neural Net



(b) After applying dropout.



- Randomly turn off some neurons
- Allows individual neurons to independently be responsible for performance

Dropout: A simple way to prevent neural networks from overfitting [[Srivastava JMLR 2014](#)]

---

# Transfer Learning

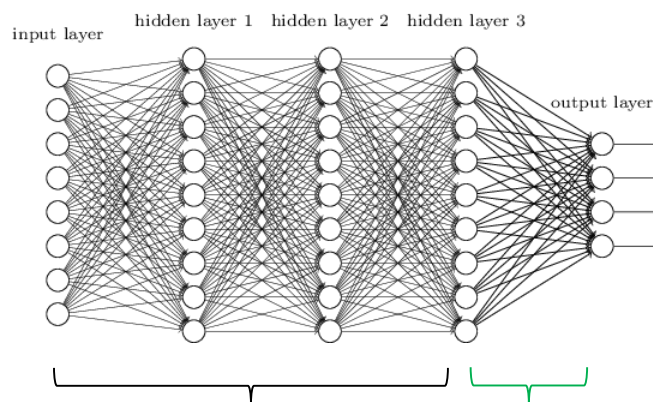
“You need a lot of data if you want to train/use CNNs”

**BUSTED**

# Transfer Learning with CNNs

---

- The more weights you need to learn, the more data you need
- That's why with a deeper network, you need more data for training than for a shallower network
- One possible solution:



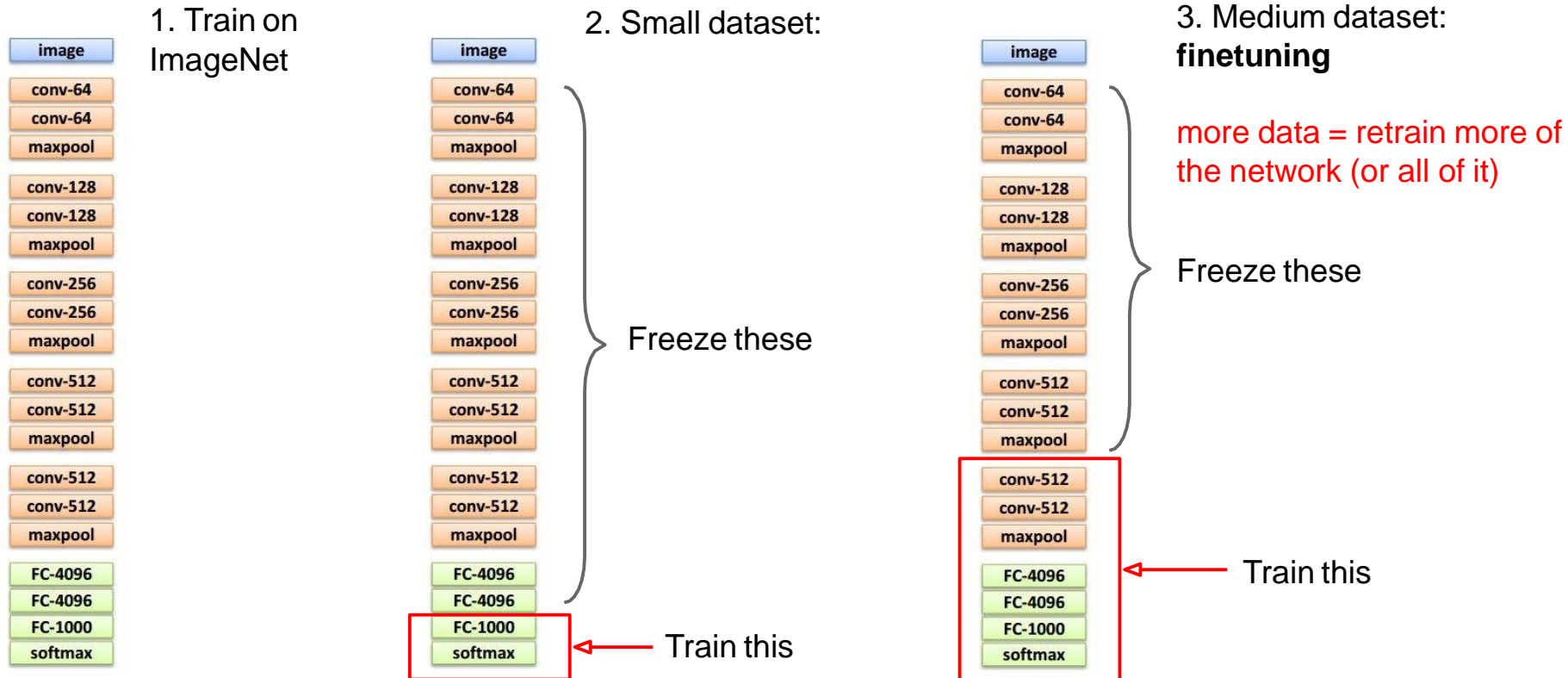
Set these to the already learned weights from another network

Learn these on your own task

# Transfer Learning with CNNs

Source: classification on ImageNet

Target: classification on Places



Another option: use network as feature extractor,  
train SVM on extracted features for target task

# Training: Best practices

---

- Center (subtract mean from) your data
- To initialize weights, use “Xavier initialization”
- Use RELU or leaky RELU or ELU, don’t use sigmoid
- Use mini-batch
- Use data augmentation
- Use regularization
- Use batch normalization
- Use cross-validation for your parameters
- Learning rate: too high? Too low?

# Plan for the rest of the lecture

---

## Neural network basics

- Definition
- Loss functions
- Optimization w/ gradient descent and backpropagation

## Convolutional neural networks (CNNs)

- Special operations
- Common architectures

## Practical matters

- Getting started: Preprocessing, initialization, optimization, normalization
- Improving performance: regularization, augmentation, transfer
- Hardware and software

## Understanding CNNs

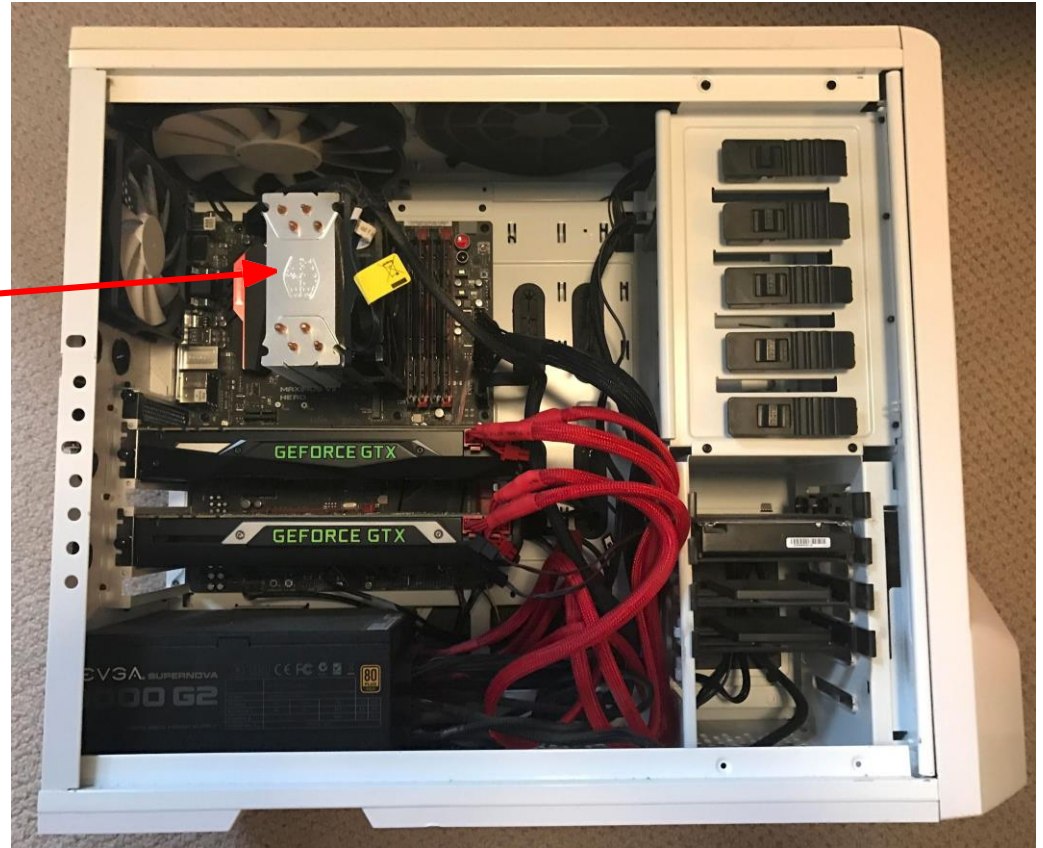
- Visualization
- Breaking CNNs



Hardware and software

# Spot the CPU! (central processing unit)

---



# Spot the GPUs! (graphics processing unit)

---



# CPU vs GPU

---

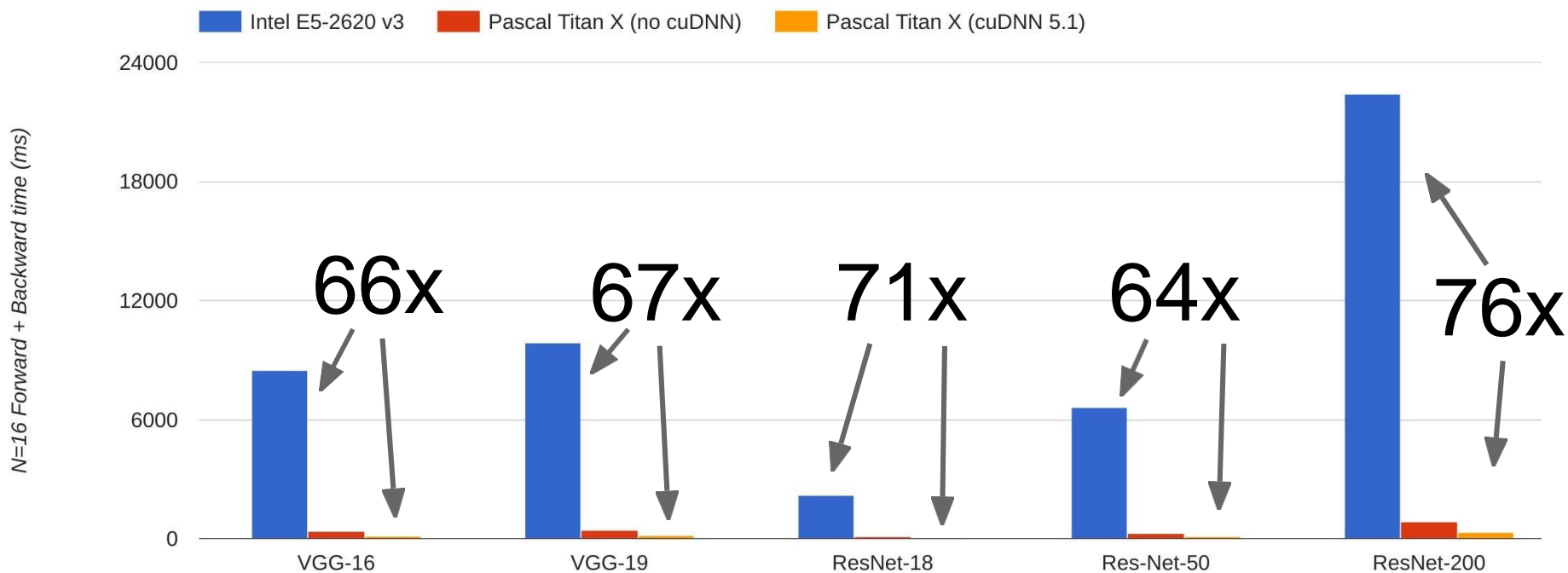
	Cores	Clock Speed	Memory	Price	Speed
<b>CPU</b> (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$339	~540 GFLOPs FP32
<b>GPU</b> (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	\$699	~11.4 TFLOPs FP32

**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

# CPU vs GPU in practice

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

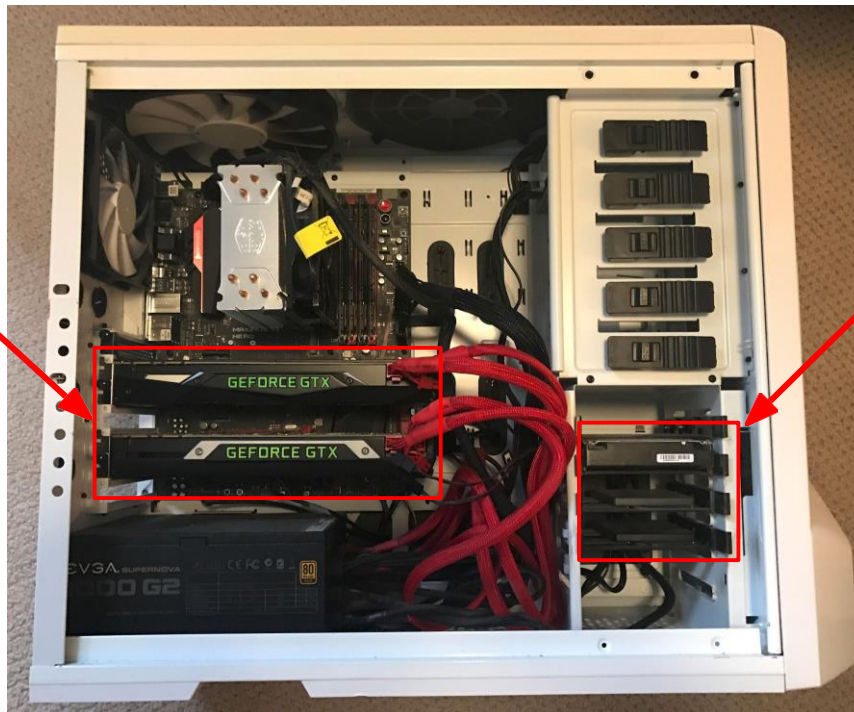


Data from <https://github.com/jcjohnson/cnn-benchmarks>

# CPU / GPU Communication

---

Model  
is here



Data is here

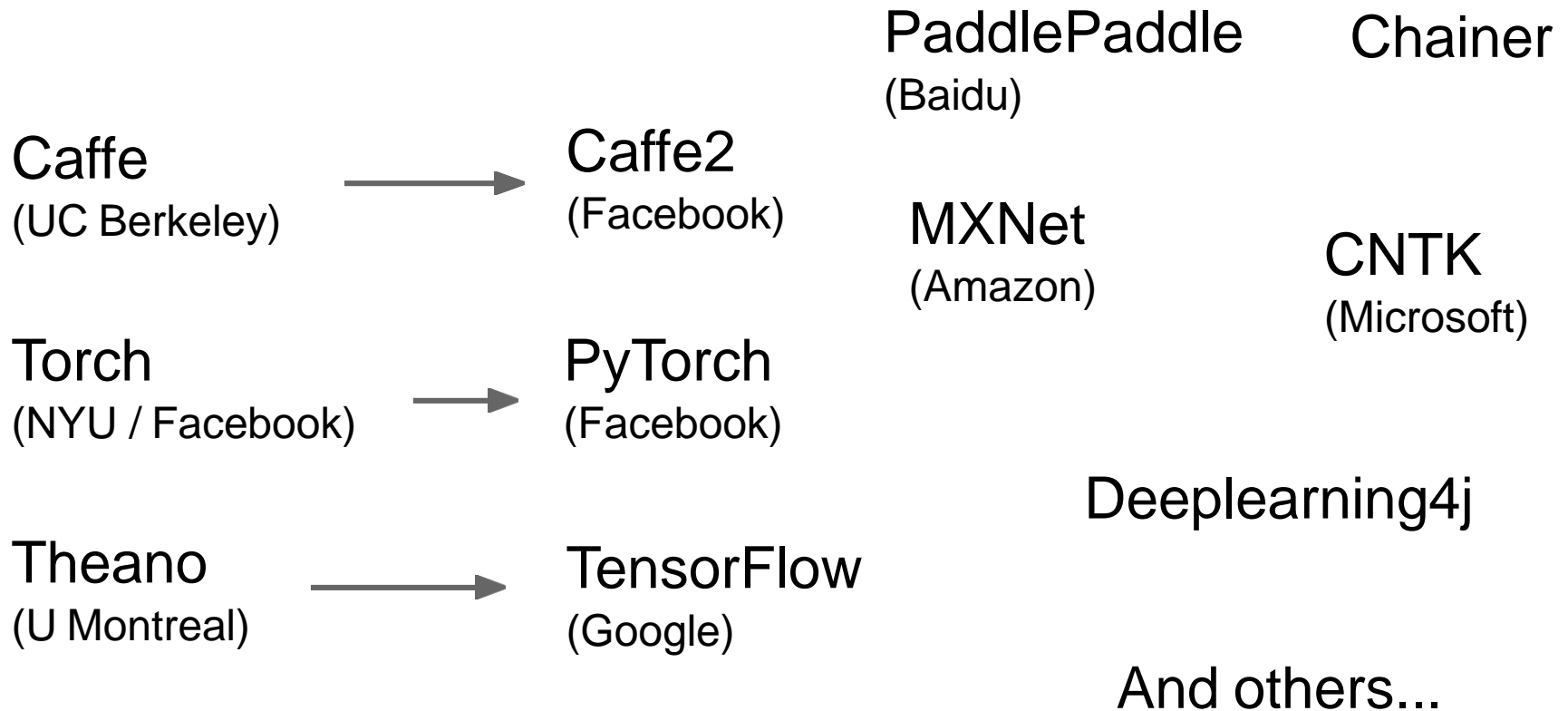
If you aren't careful, training can bottleneck on reading data and transferring to GPU!

## Solutions:

- Read all data into RAM
- Use SSD instead of HDD
- Use multiple CPU threads to prefetch data

# Software: A zoo of frameworks!

---



# Plan for the rest of the lecture

---

## Neural network basics

- Definition
- Loss functions
- Optimization w/ gradient descent and backpropagation

## Convolutional neural networks (CNNs)

- Special operations
- Common architectures

## Practical matters

- Getting started: Preprocessing, initialization, optimization, normalization
- Improving performance: regularization, augmentation, transfer
- Hardware and software

## Understanding CNNs

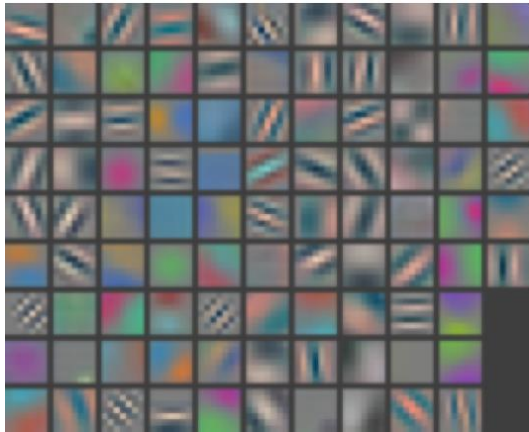
- Visualization
- Breaking CNNs



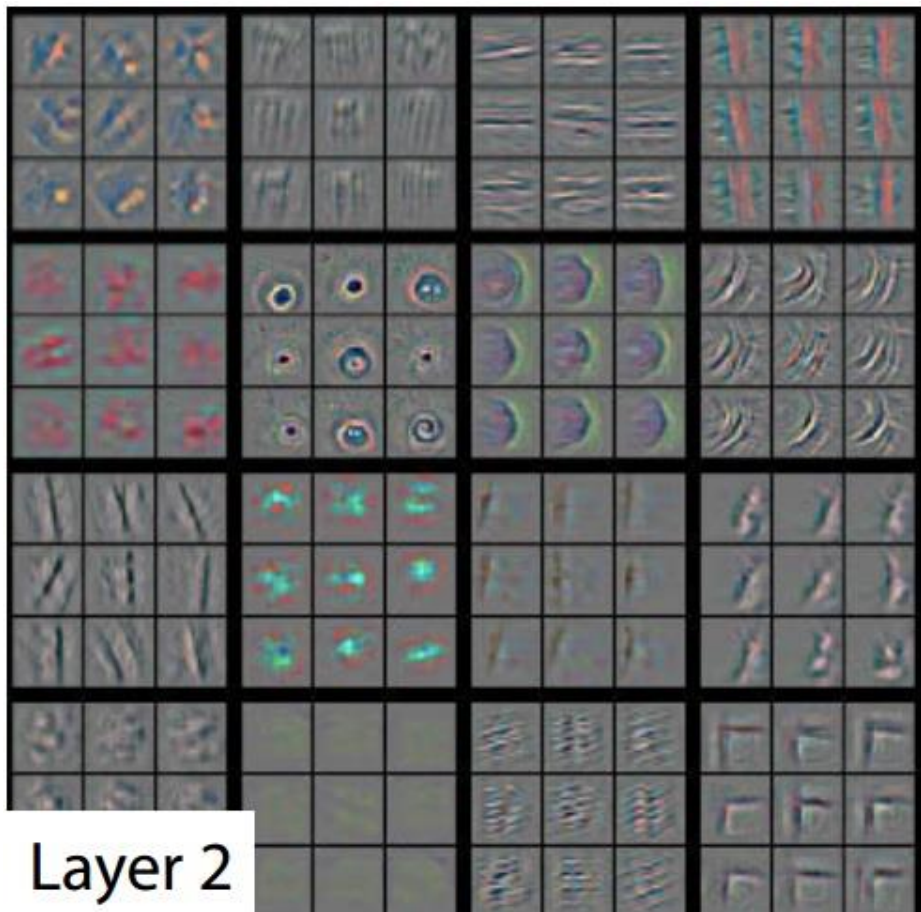
# Understanding CNNs

# Layer 1

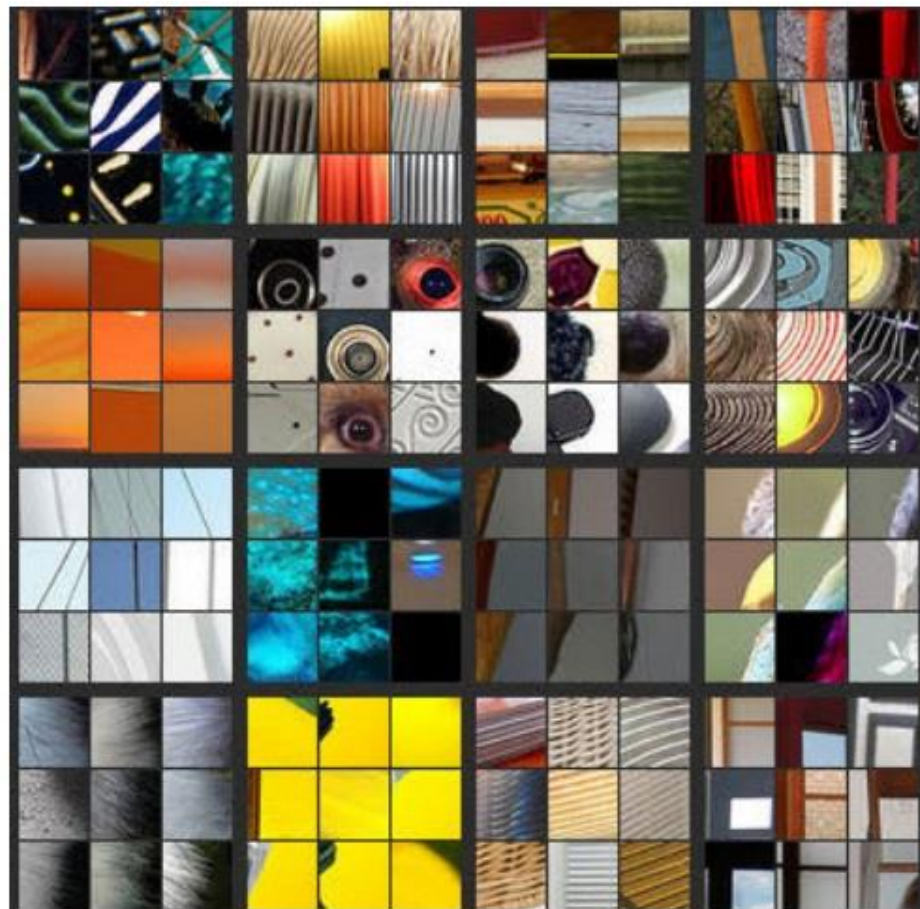
---



# Layer 2



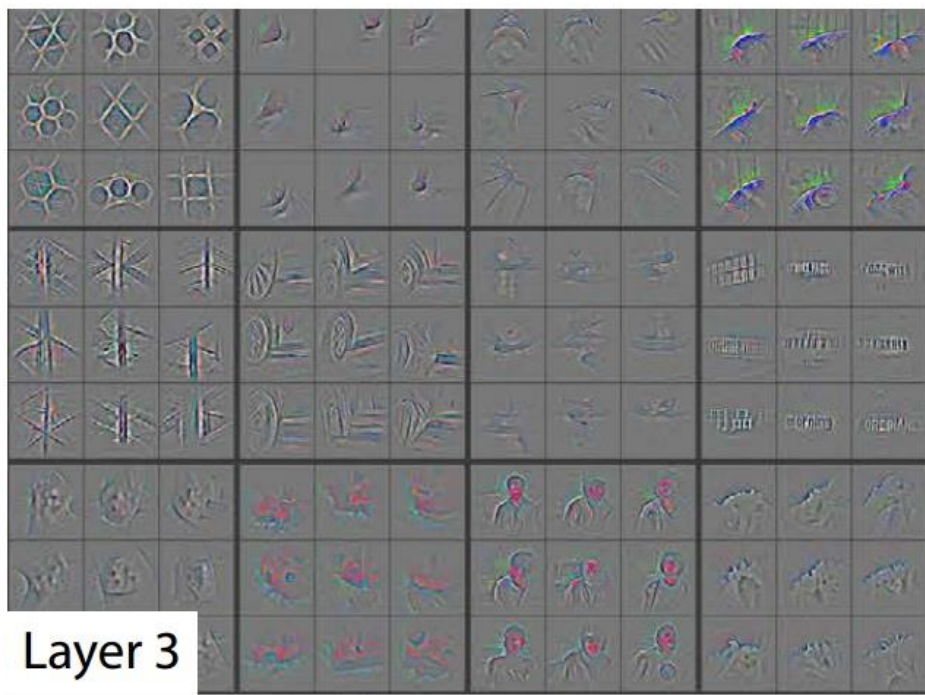
- Activations projected down to pixel level via deconvolution



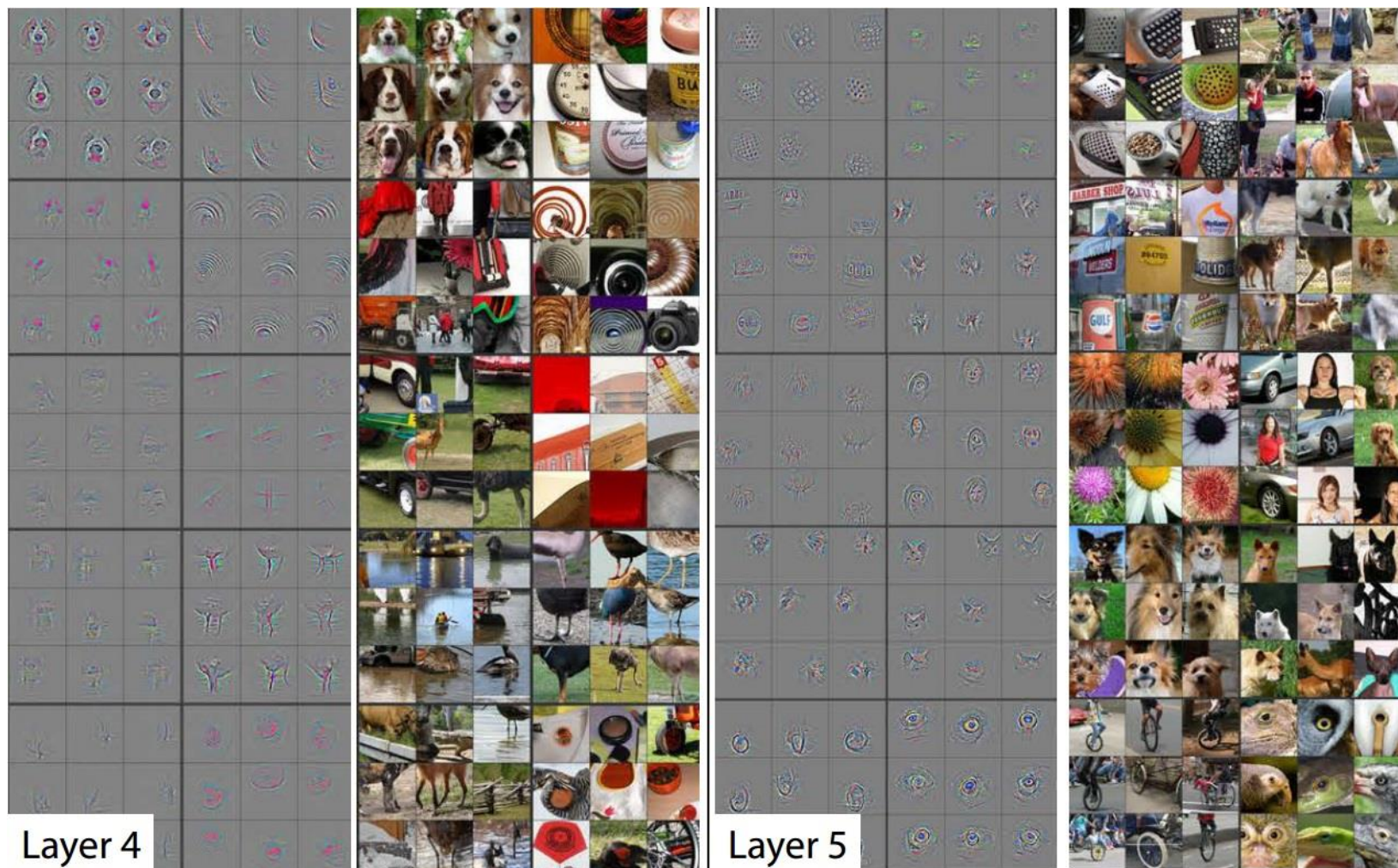
- Patches from validation images that give maximal activation of a given feature map



# Layer 3



# Layer 4 and 5





# Occlusion experiments

---

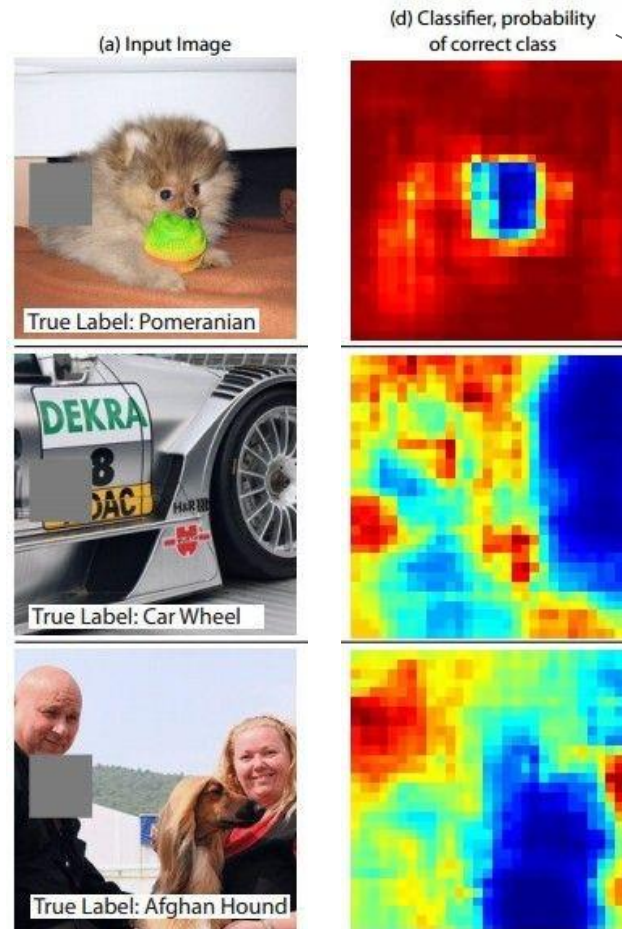


(d) Classifier, probability  
of correct class

(as a function of the  
position of the  
square of zeros in  
the original image)

[Zeiler & Fergus 2014]

# Occlusion experiments

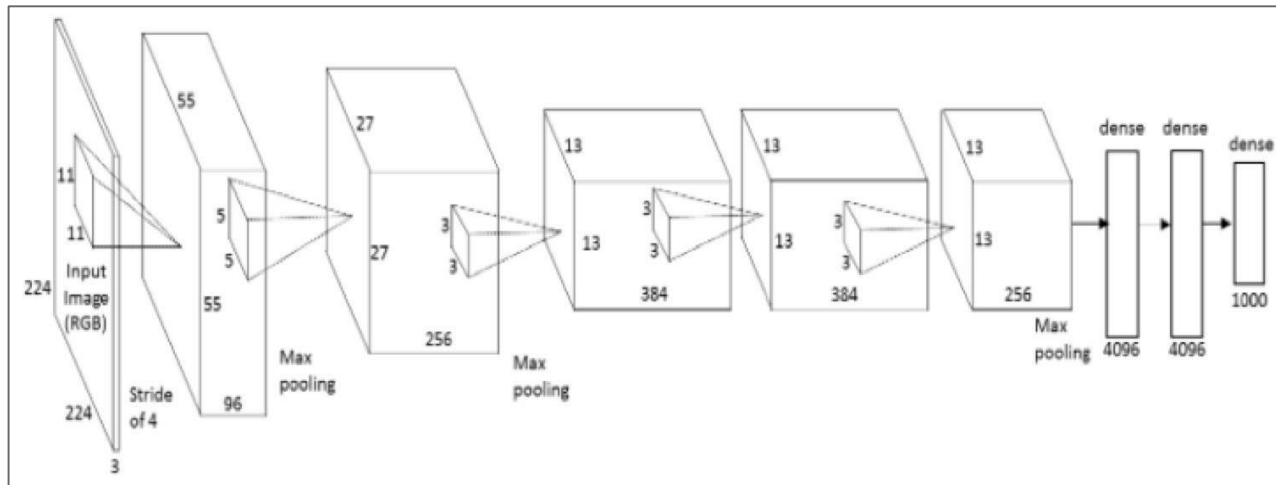


(as a function of the position of the square of zeros in the original image)

[Zeiler & Fergus 2014]

# What image maximizes a class score?

---



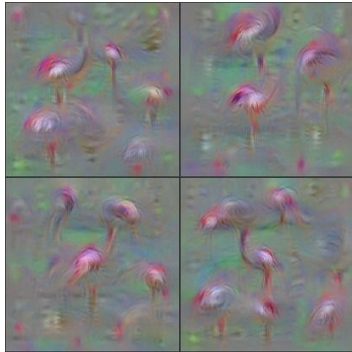
## Repeat:

1. Forward an image
2. Set activations in layer of interest to all zero, except for a 1.0 for a neuron of interest
3. Backprop to image
4. Do an "image update"

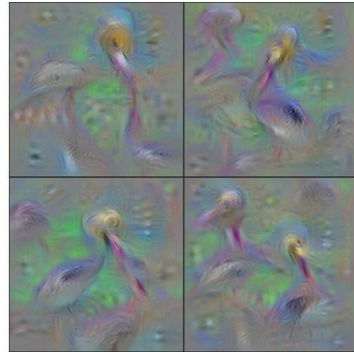


# What image maximizes a class score?

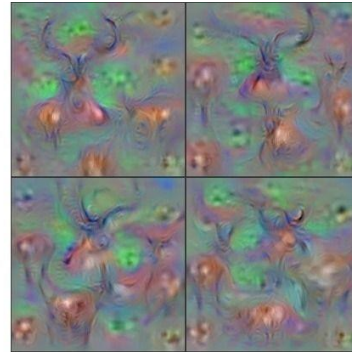
---



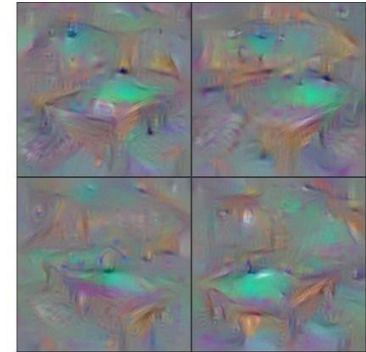
Flamingo



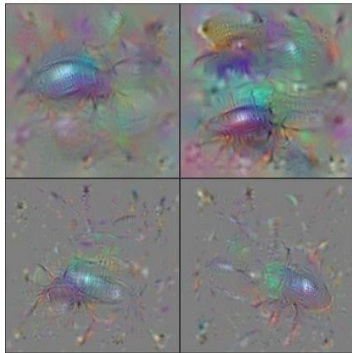
Pelican



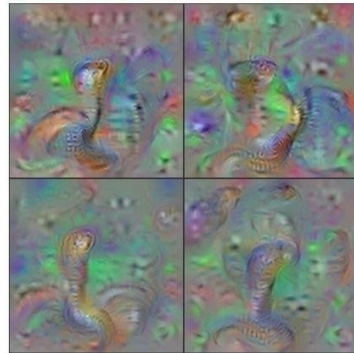
Hartebeest



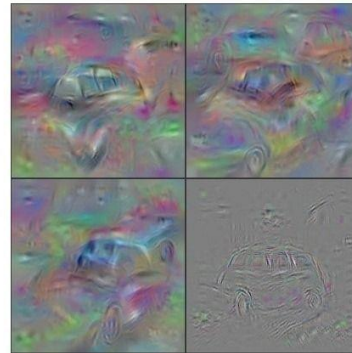
Billiard Table



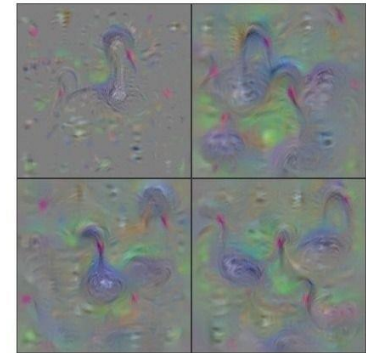
Ground Beetle



Indian Cobra



Station Wagon



Black Swan

*[Understanding Neural Networks Through Deep Visualization, Yosinski et al. , 2015]*

<http://yosinski.com/deepvis>

# What image maximizes a class score?

---

Layer 8



Pirate Ship

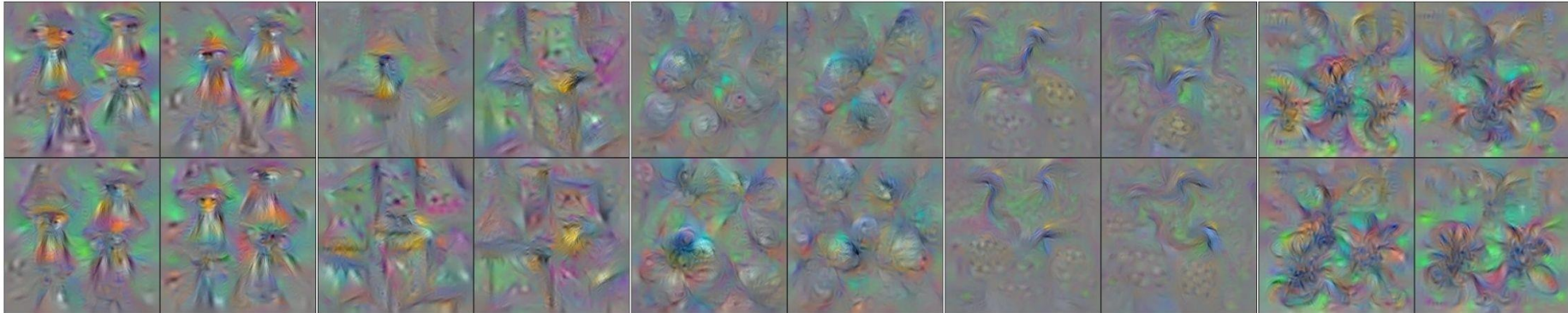
Rocking Chair

Teddy Bear

Windsor Tie

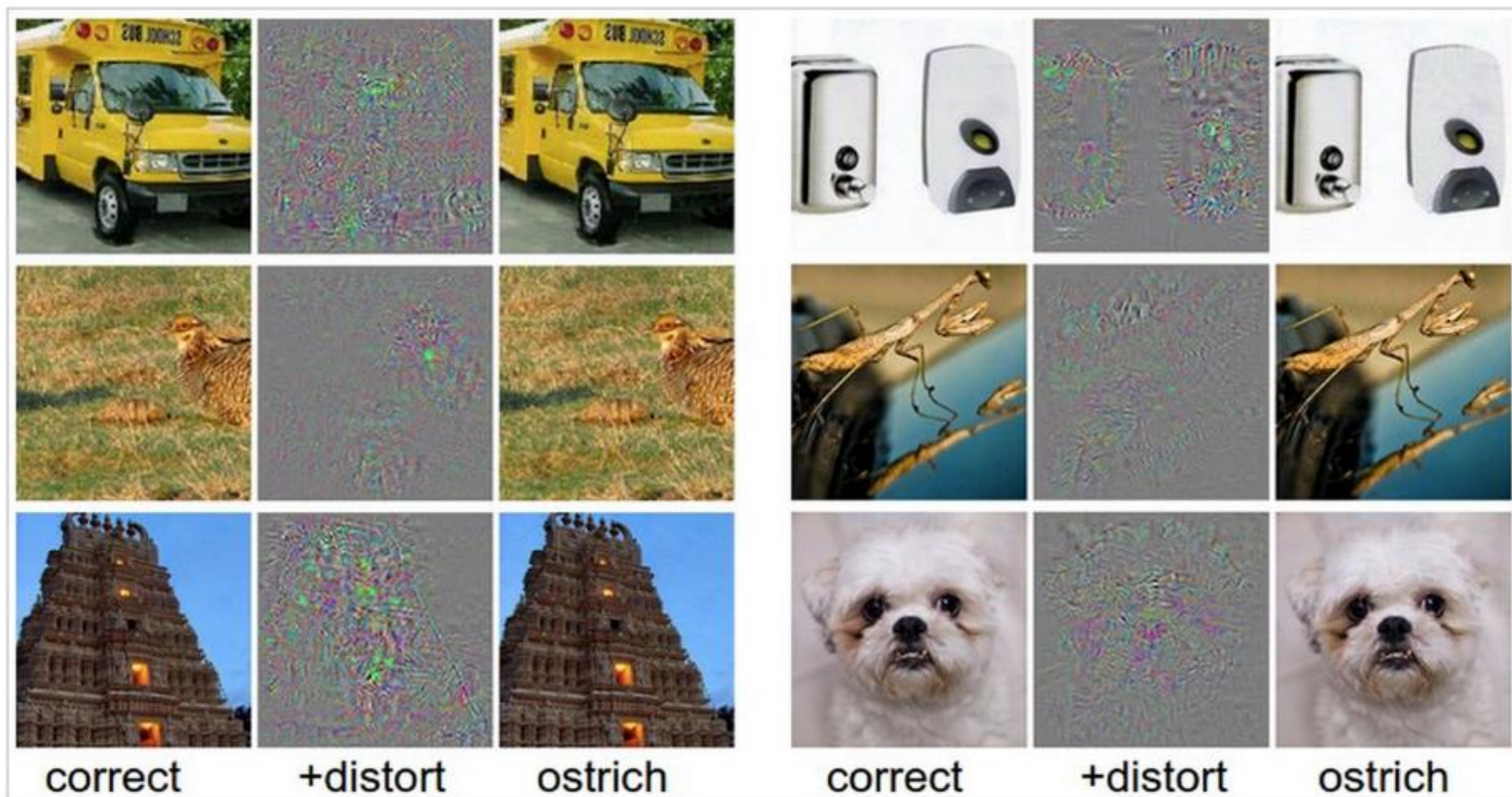
Pitcher

Layer 7





# Breaking CNNs

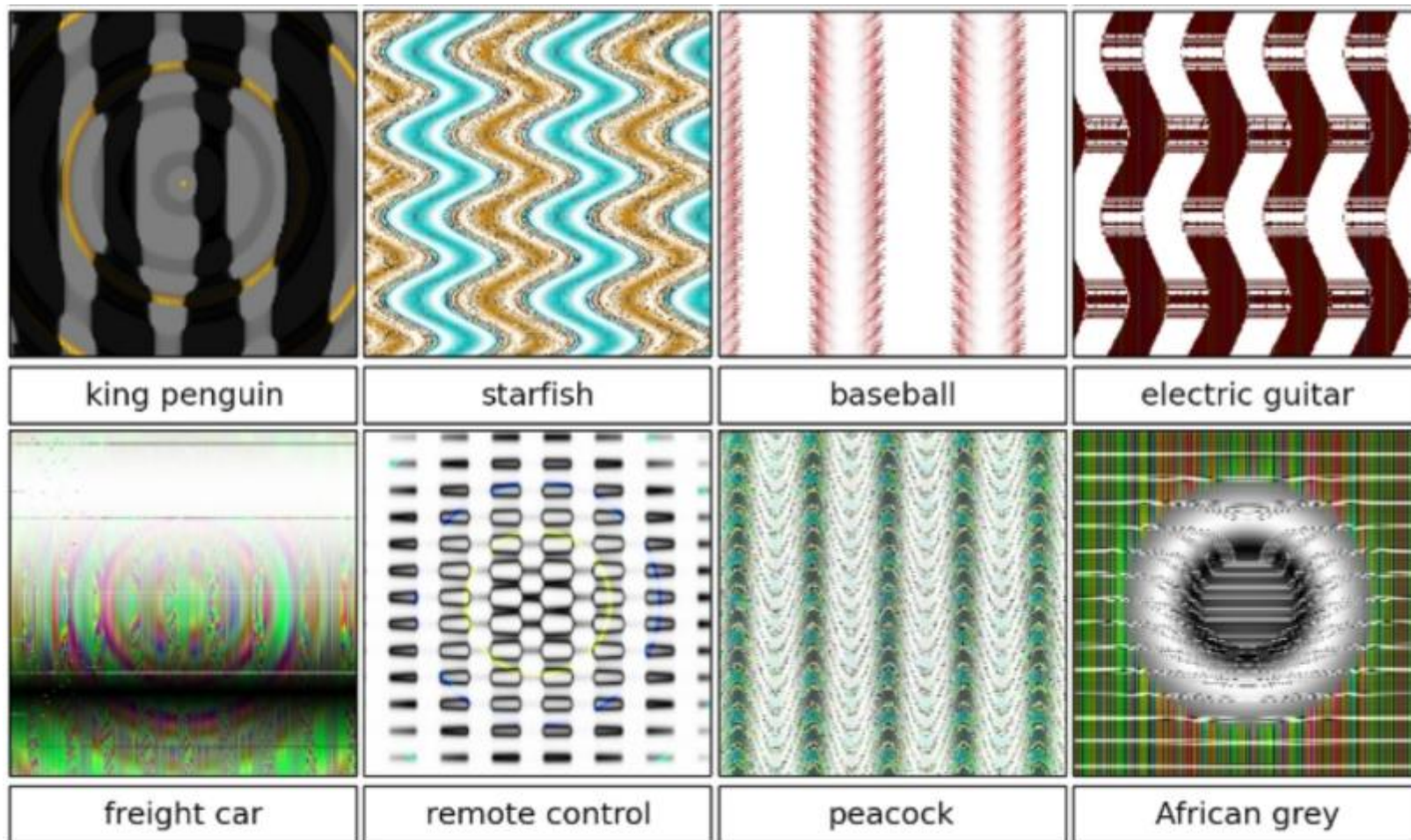


Take a correctly classified image (left image in both columns), and add a tiny distortion (middle) to fool the ConvNet with the resulting image (right).

Intriguing properties of neural networks [[Szegedy ICLR 2014](#)]

# Breaking CNNs

---



Deep Neural Networks are Easily Fooled: High Confidence Predictions for  
Unrecognizable Images [[Nguyen et al. CVPR 2015](#)]

# Summary of CNNs

---

- We use DNNs/CNNs due to performance
- Convolutional neural network (CNN)
  - Convolution, nonlinearity, max pooling
  - AlexNet, VGG, GoogleNet, ResNet, ...
- Training deep neural nets
  - We need an objective function that measures and guides us towards good performance
  - Backpropagate error towards all layers and change weights
  - Take steps to minimize the loss function: SGD, AdaGrad, RMSProp, Adam
- Practices for preventing overfitting
  - Dropout; data augmentation; transfer learning