CS 2770: Computer Vision

Classification and Tools (CNN, SVM)

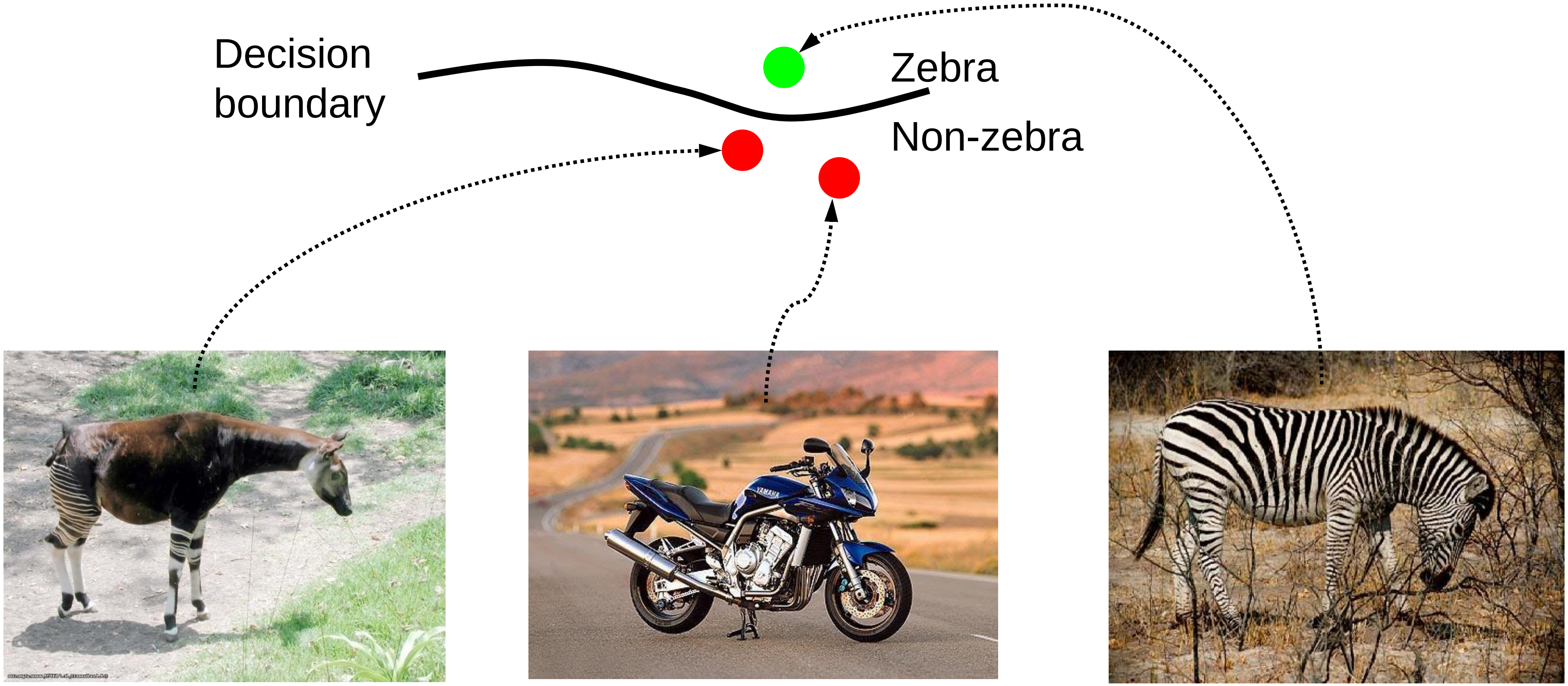
Prof. Adriana Kovashka
University of Pittsburgh
February 7, 2019

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?



Decision boundary

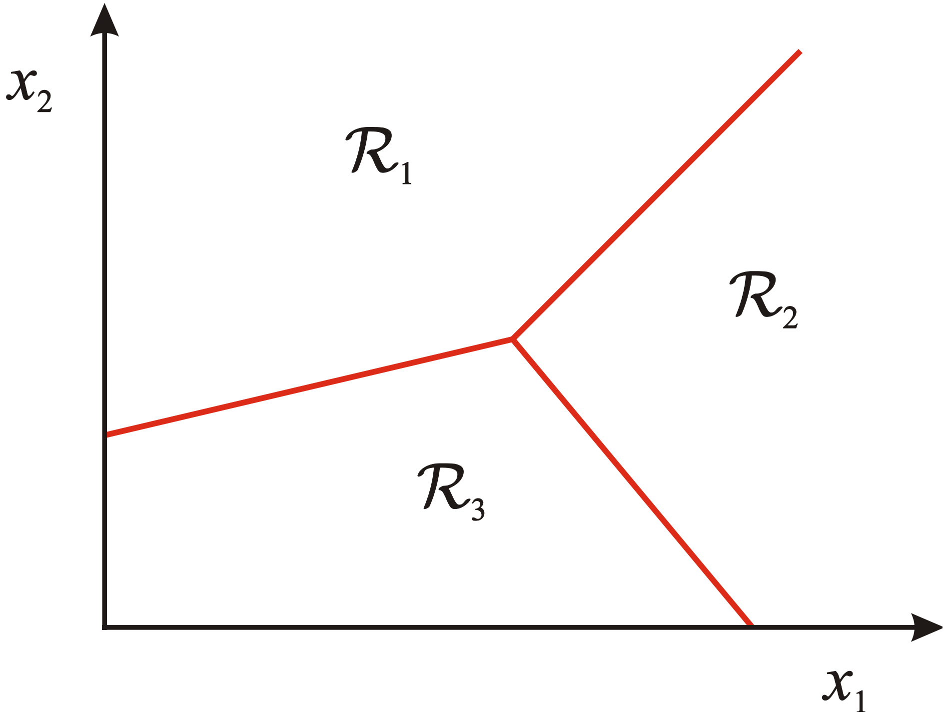
Zebra

Non-zebra

Slide credit: L. Lazebnik

Classification

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



Slide credit: L. Lazebnik

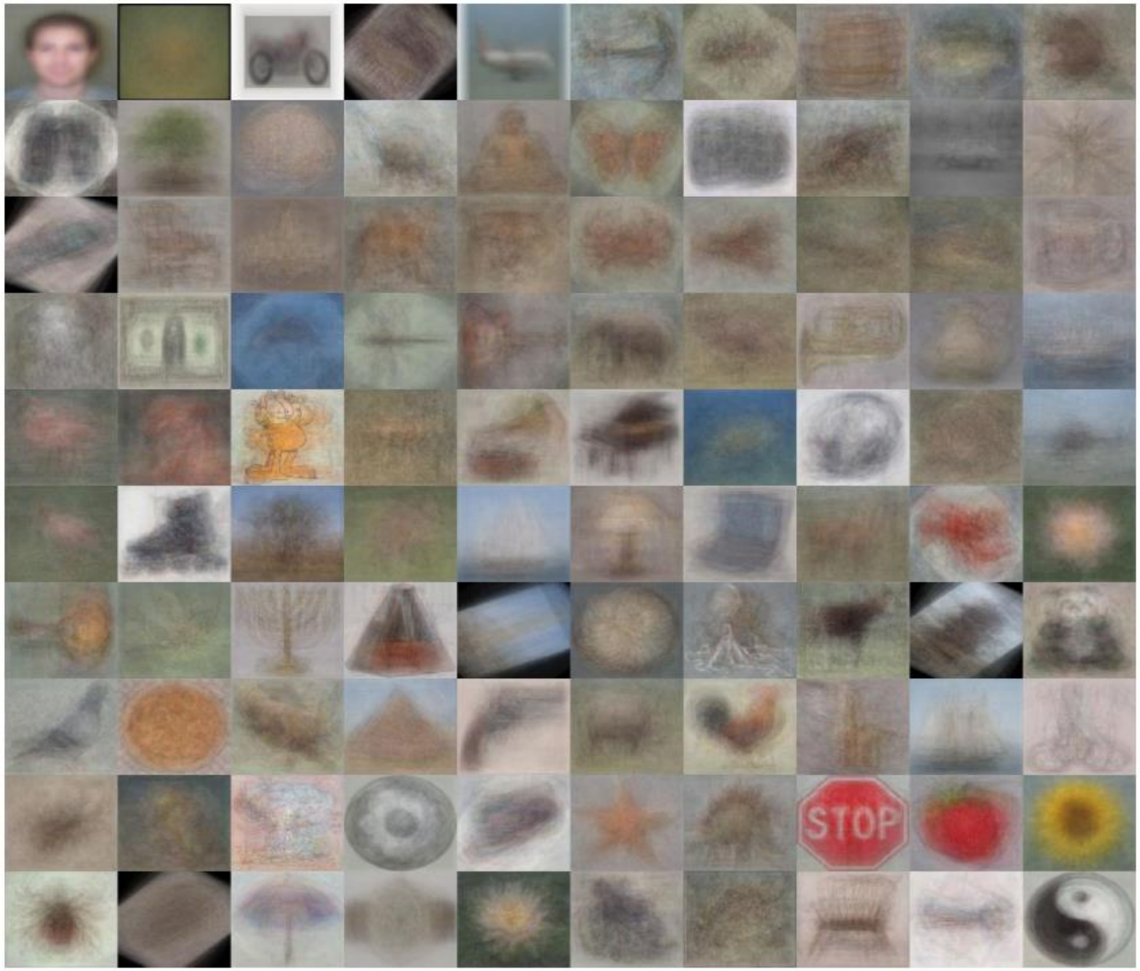
Examples of image classification

- Two-class (binary): Cat vs Dog

Adapted from D. Hoiem

Examples of image classification

- Multi-class (often): Object recognition



Caltech 101 Average Object Images

Adapted from D. Hoiem

Examples of image classification

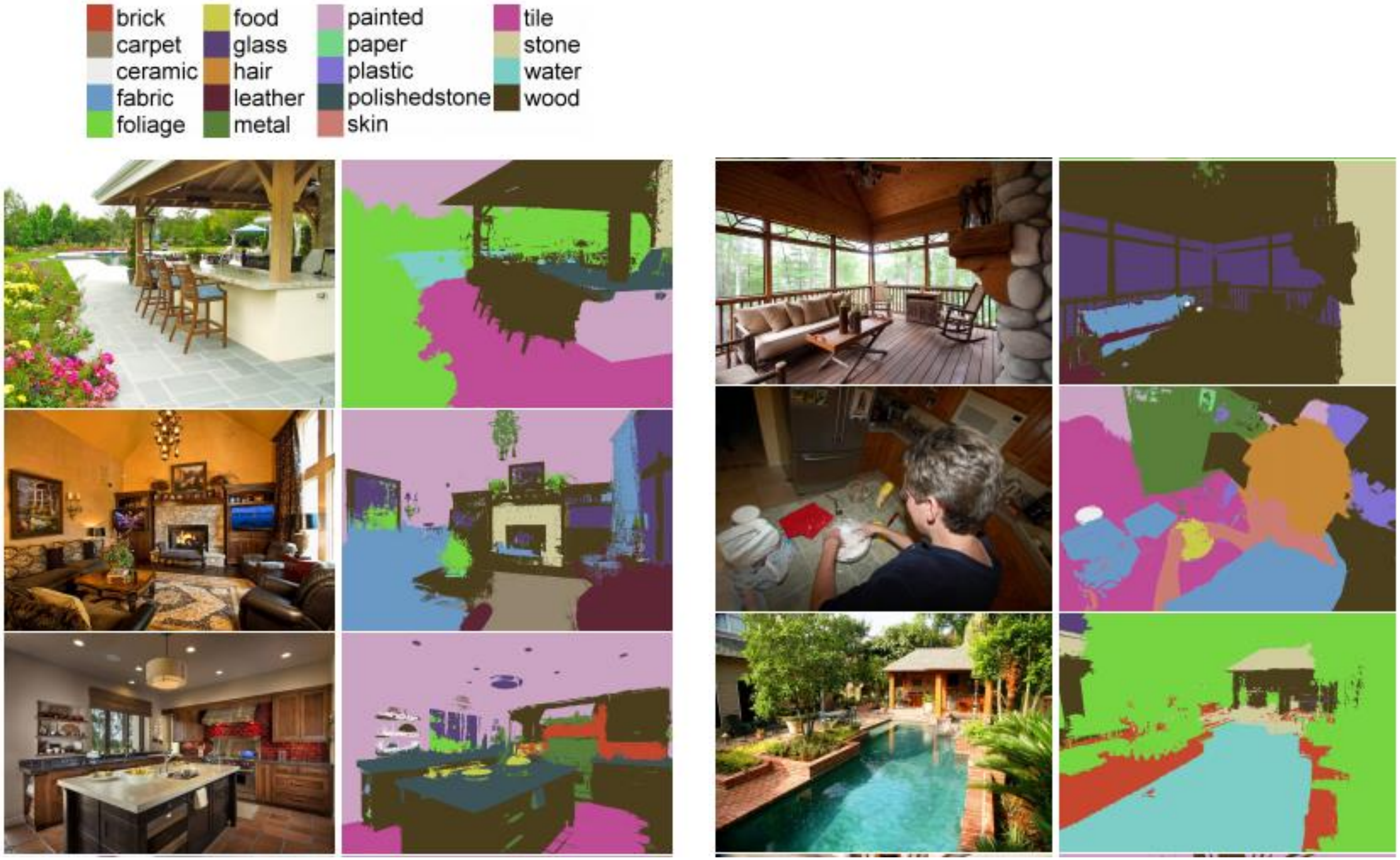
- Place recognition

Places Database [Zhou et al. NIPS 2014]

Slide credit: D. Hoiem

Examples of image classification

- Material recognition

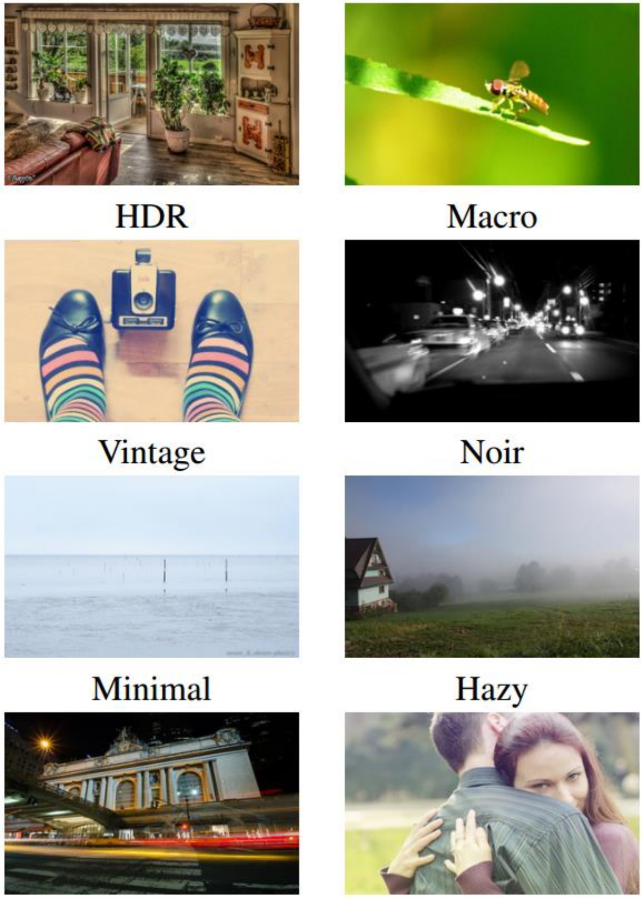


[Bell et al. CVPR 2015]

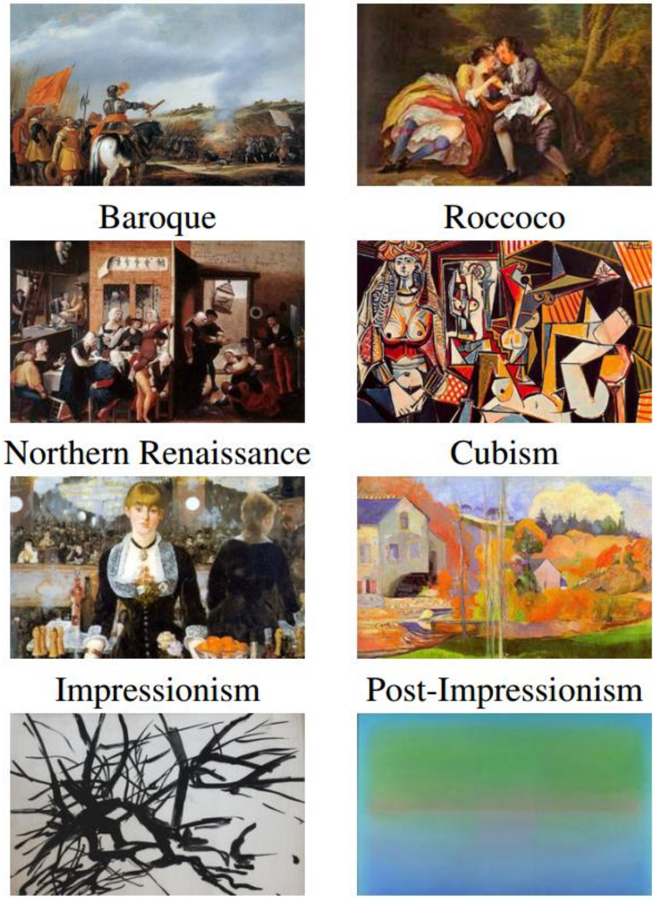
Slide credit: D. Hoiem

Examples of image classification

- Image style recognition


HDR, Macro, Vintage, Noir, Minimal, Hazy, Long Exposure, Romantic

Flickr Style: 80K images covering 20 styles.


Baroque, Roccoco, Northern Renaissance, Cubism, Impressionism, Post-Impressionism, Abs. Expressionism, Color Field Painting

Wikipaintings: 85K images for 25 art genres.

[Karayev et al. BMVC 2014]

Slide credit: D. Hoiem

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() = “apple”

f() = “tomato”

f() = “cow”

Slide credit: L. Lazebnik

The machine learning framework

$$y = f(\mathbf{x})$$

output

prediction function

image / image feature

- **Training:** given a *training set* of labeled examples $\{(\mathbf{x}_1,y_1), \ldots, (\mathbf{x}_N,y_N)\}$, 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 $y = f(\mathbf{x})$

Slide credit: L. Lazebnik

The old-school way

Training

Training Labels

Training Images

Image Features

Training

Learned model

Testing

Test Image

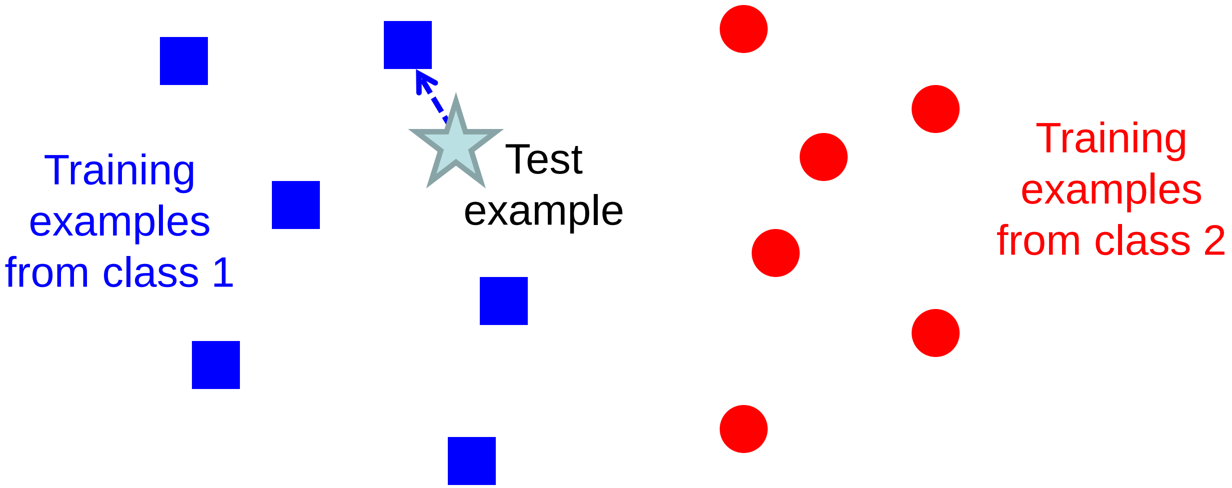
Image Features

Learned model

Prediction

Slide credit: D. Hoiem and L. Lazebnik

The simplest classifier



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

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

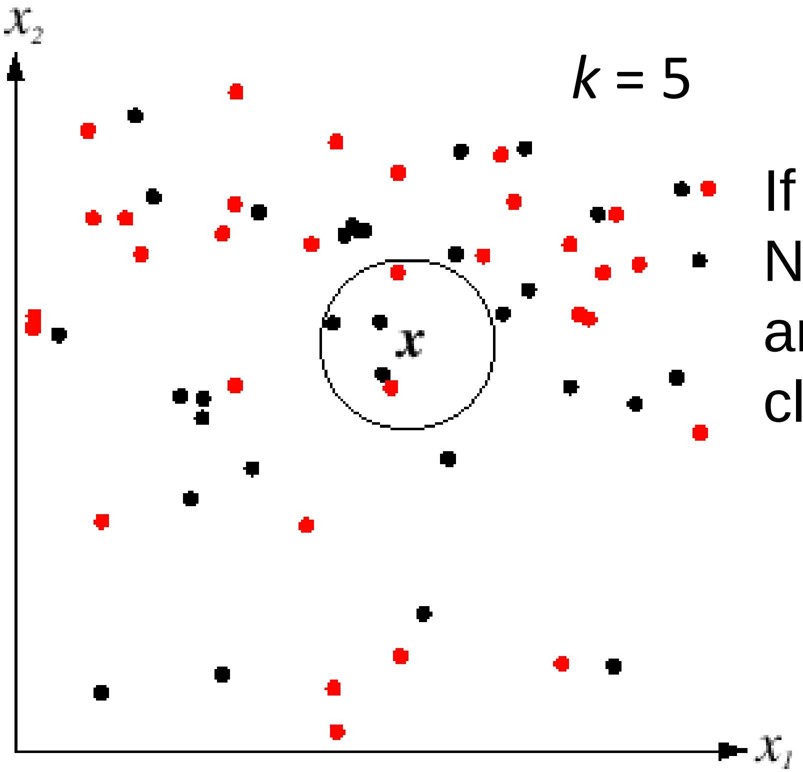
Slide credit: L. Lazebnik

K-Nearest Neighbors classification

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

Black = negative

Red = positive



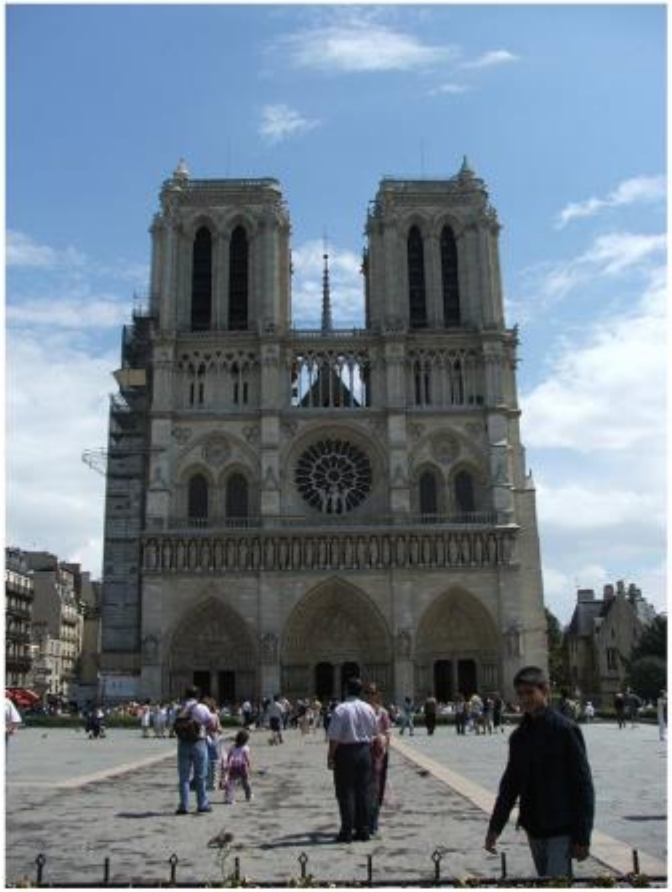
If query lands here, the 5 NN consist of 3 negatives and 2 positives, so we classify it as negative.

Slide credit: D. Lowe

im2gps: **Estimating Geographic Information from a Single Image**

James Hays and Alexei Efros, CVPR 2008

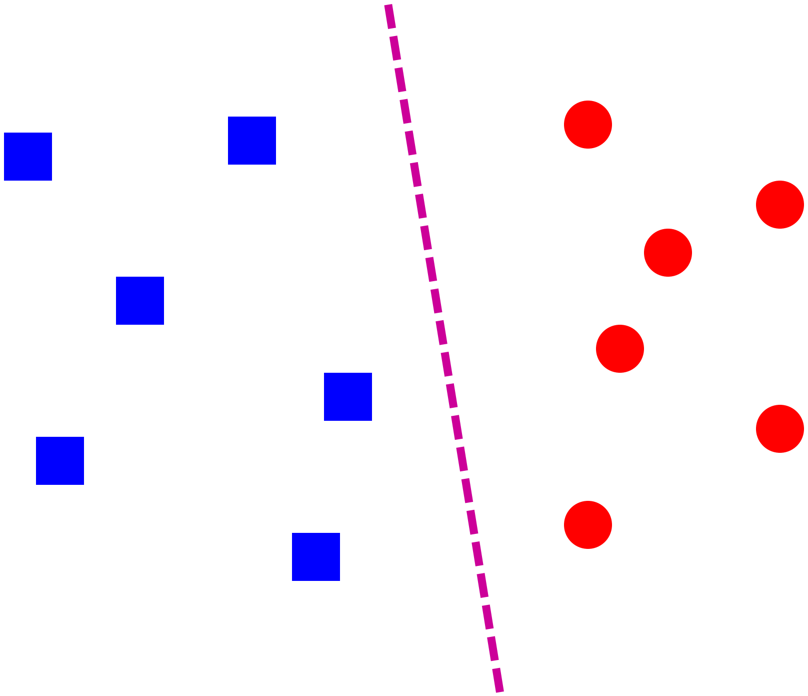
Where was this image taken?



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

Slide credit: James Hays

Linear classifier



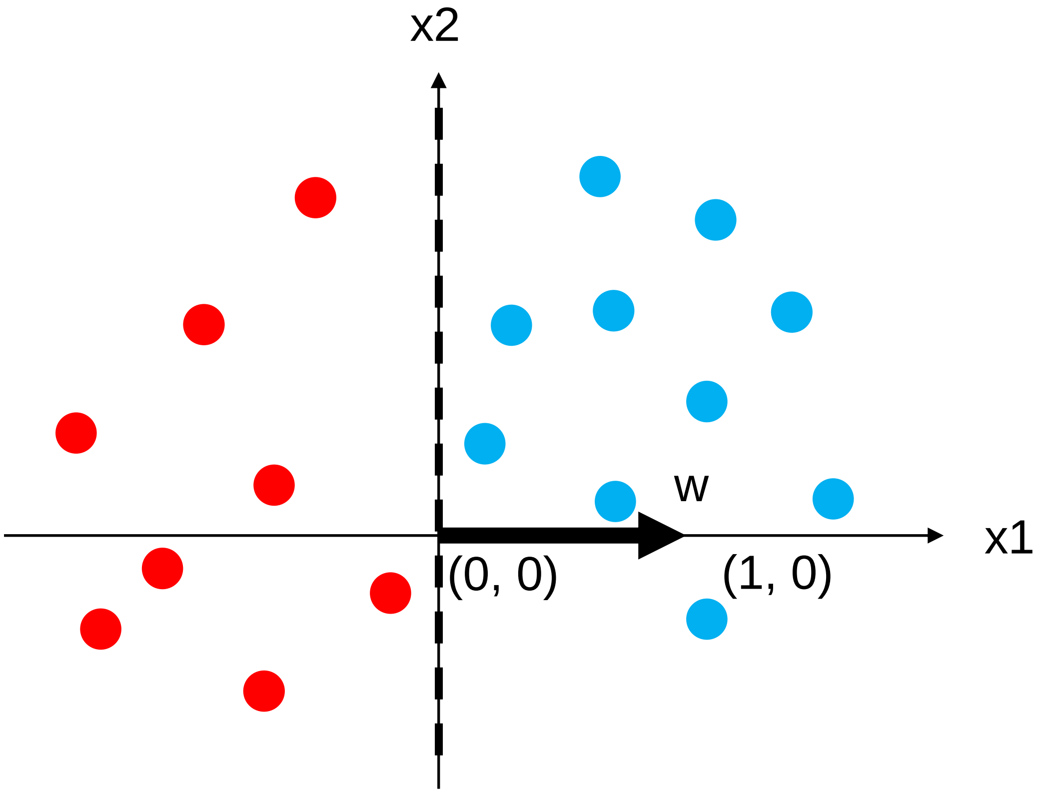
- Find a *linear function* to separate the classes

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

Slide credit: L. Lazebnik

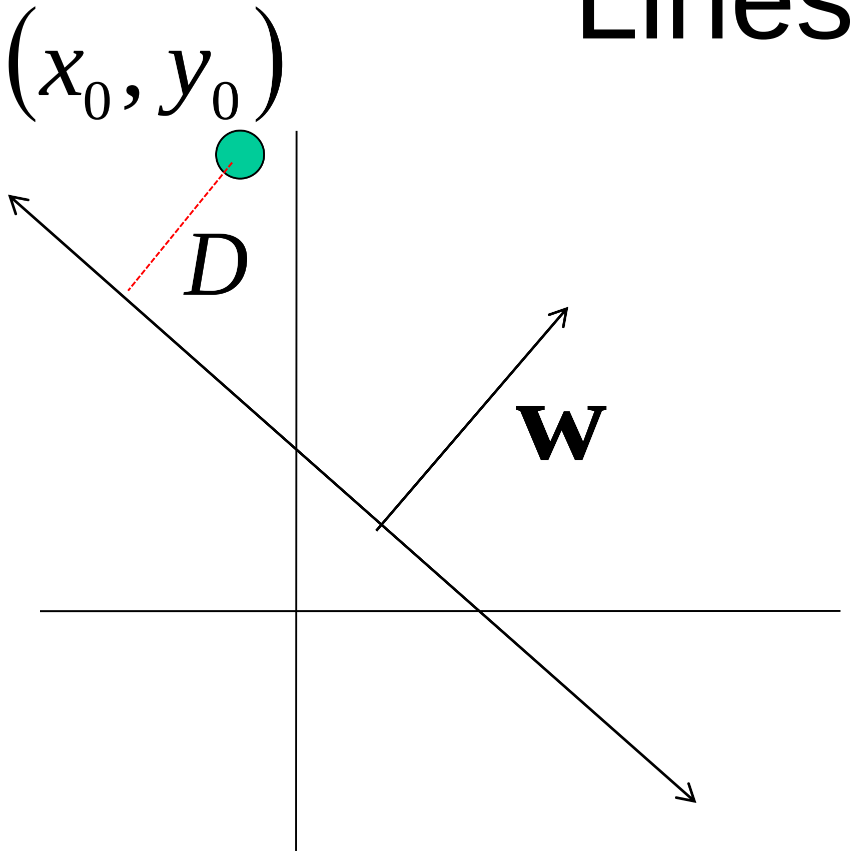
Linear classifier

- Decision = sign($w^T x$) = sign(w1*x1 + w2*x2)



- What should the weights be?

Lines in R^2



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

$$ax + cy + b = 0$$

$$\updownarrow$$

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

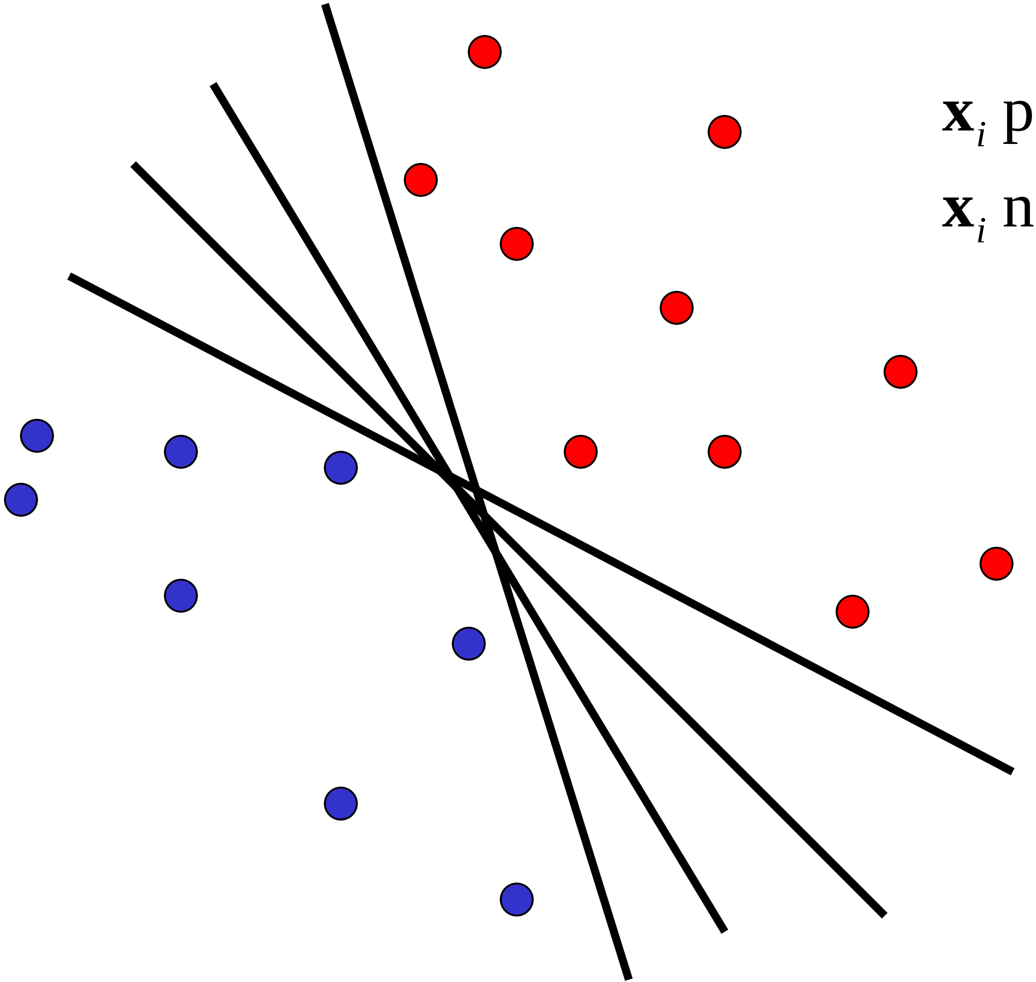
$$D = \frac{\left| a x_0 + c y_0 + b \right|}{\sqrt{a^2 + c^2}} = \frac{|\, \mathbf{w}^{\mathrm{T}} \mathbf{x} + b \,|}{\left\| \mathbf{w} \right\|} \Bigg\}$$

distance from point to line

Kristen Grauman

Linear classifiers

- Find linear function to separate positive and negative examples



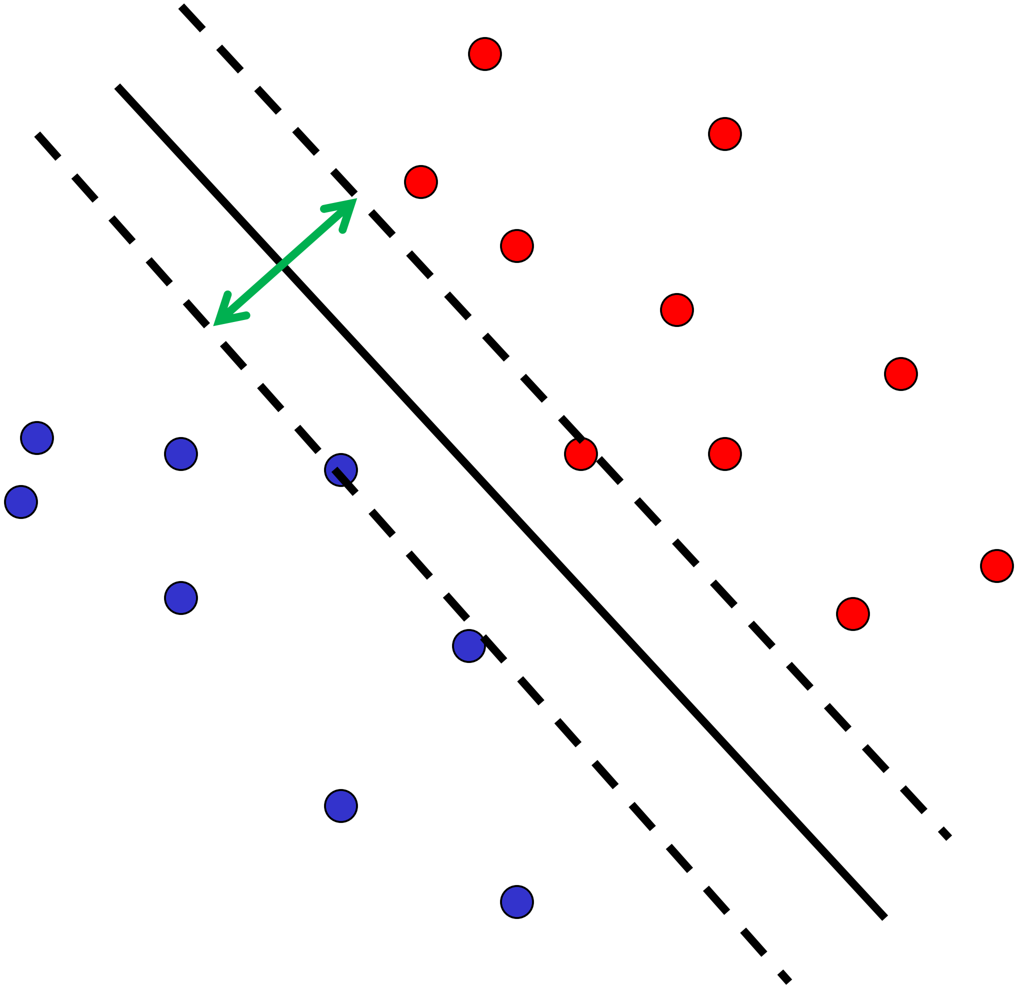
$\mathbf{x}_i$ positive: $\quad \mathbf{x}_i \cdot \mathbf{w} + b \geq 0$

$\mathbf{x}_i$ negative : $\quad \mathbf{x}_i \cdot \mathbf{w} + b < 0$

Which line
is best?

C. Burges, A Tutorial on Support Vector Machines for Pattern Recognition, Data Mining and Knowledge Discovery, 1998

Support vector machines

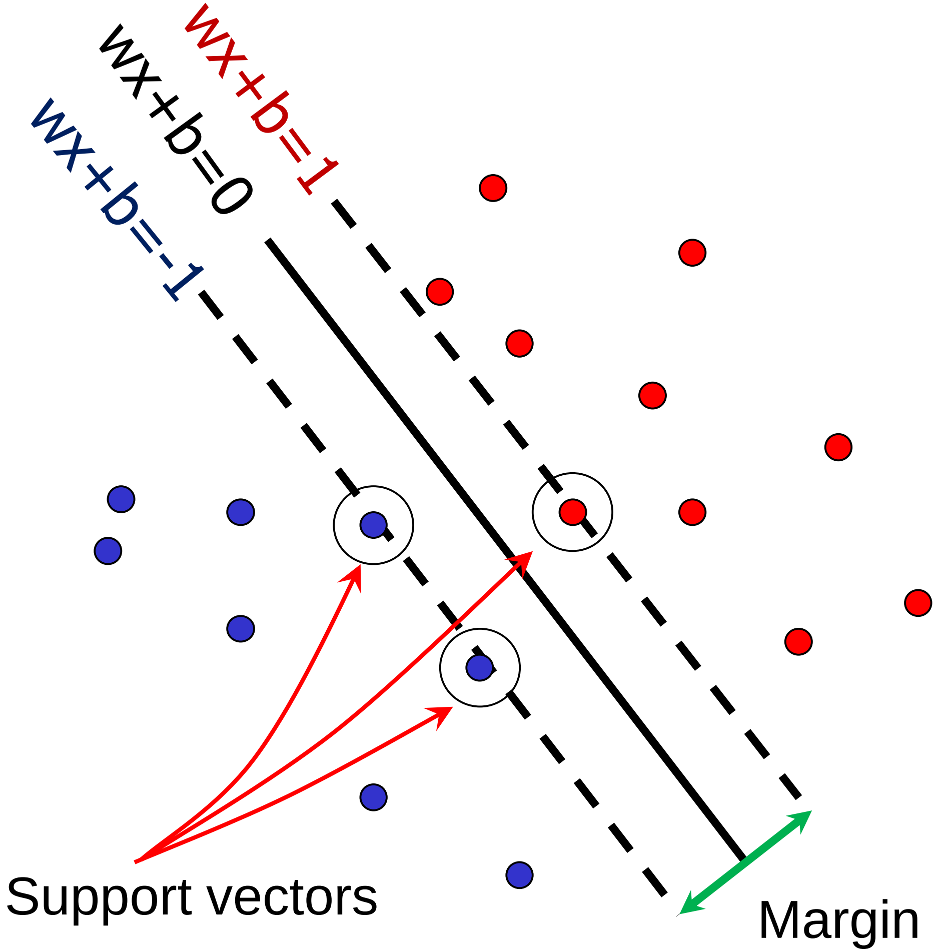


- Discriminative classifier based on *optimal separating line (for 2d case)*
- Maximize the *margin* between the positive and negative training examples

C. Burges, [A Tutorial on Support Vector Machines for Pattern Recognition](), Data Mining and Knowledge Discovery, 1998

Support vector machines

- Want line that maximizes the margin.



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

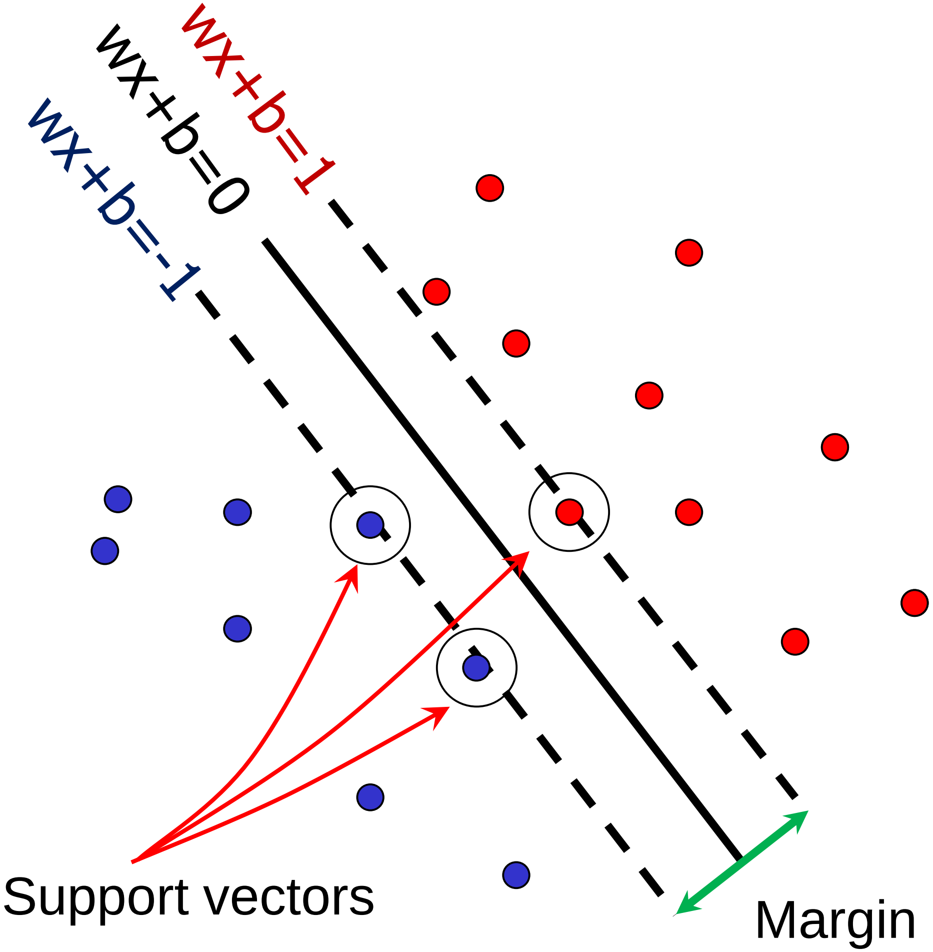
$\mathbf{x}_i$ negative $(y_i = -1)$: $\quad \mathbf{x}_i \cdot \mathbf{w} + b \leq -1$

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

C. Burges, A Tutorial on Support Vector Machines for Pattern Recognition, Data Mining and Knowledge Discovery, 1998

Support vector machines

- Want line that maximizes the margin.



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

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

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

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

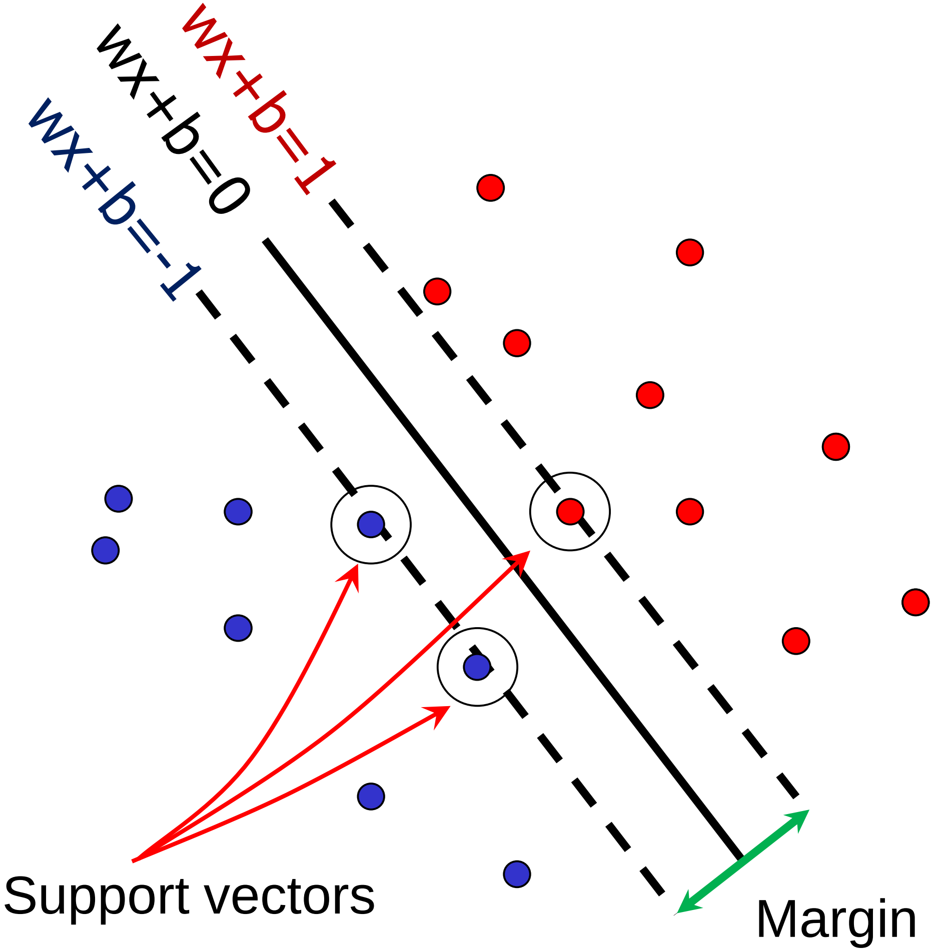
For support vectors:

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

C. Burges, A Tutorial on Support Vector Machines for Pattern Recognition, Data Mining and Knowledge Discovery, 1998

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 $2 / \|\mathbf{w}\|$

C. Burges, A Tutorial on Support Vector Machines for Pattern Recognition, Data Mining and Knowledge Discovery, 1998

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:

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

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

One constraint for each training point.

Note sign trick.

C. Burges, A Tutorial on Support Vector Machines for Pattern Recognition, Data Mining and Knowledge Discovery, 1998

Finding the maximum margin line

- Solution: $\mathbf{w} = \sum_i \underbrace{\alpha_i}_{\text{Learned weight}} y_i \underbrace{\mathbf{x}_i}_{\text{Support vector}}$

C. Burges, A Tutorial on Support Vector Machines for Pattern Recognition, Data Mining and Knowledge Discovery, 1998

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:

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

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

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

C. Burges, A Tutorial on Support Vector Machines for Pattern Recognition, Data Mining and Knowledge Discovery, 1998

Inner product

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

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

$$f(x) = \text{sign}\,(\mathbf{w}\cdot\mathbf{x} + \text{b})$$
$$= \text{sign}\left(\sum_{\text{i}} \alpha_{\text{i}} y_i \boxed{\mathbf{x}_i \cdot \mathbf{x}} + b\right)$$

- **The inner product is equal**

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

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

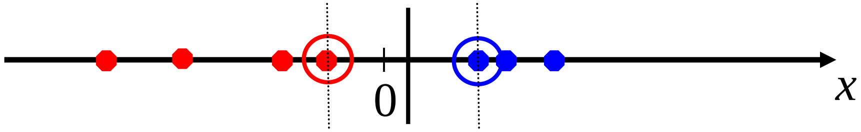
If the angle between them is 90 then: $(\mathbf{x}_i^T \mathbf{x}) = 0$

The inner product measures how similar the two vectors are

Adapted from Milos Hauskrecht

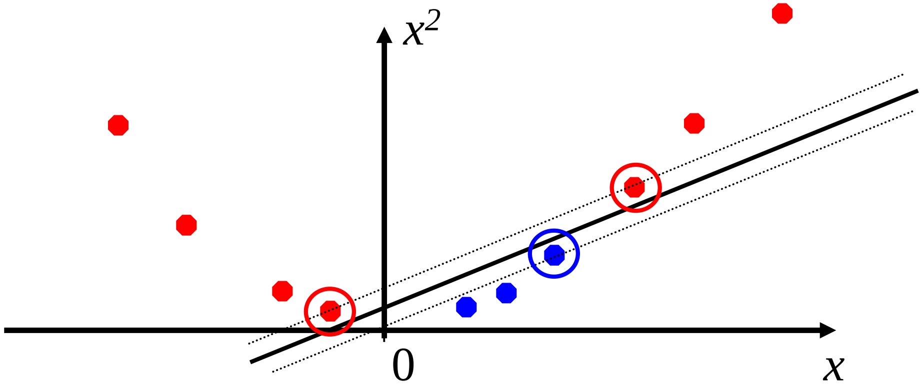
Nonlinear SVMs

- Datasets that are linearly separable work out great:



- But what if the dataset is just too hard?

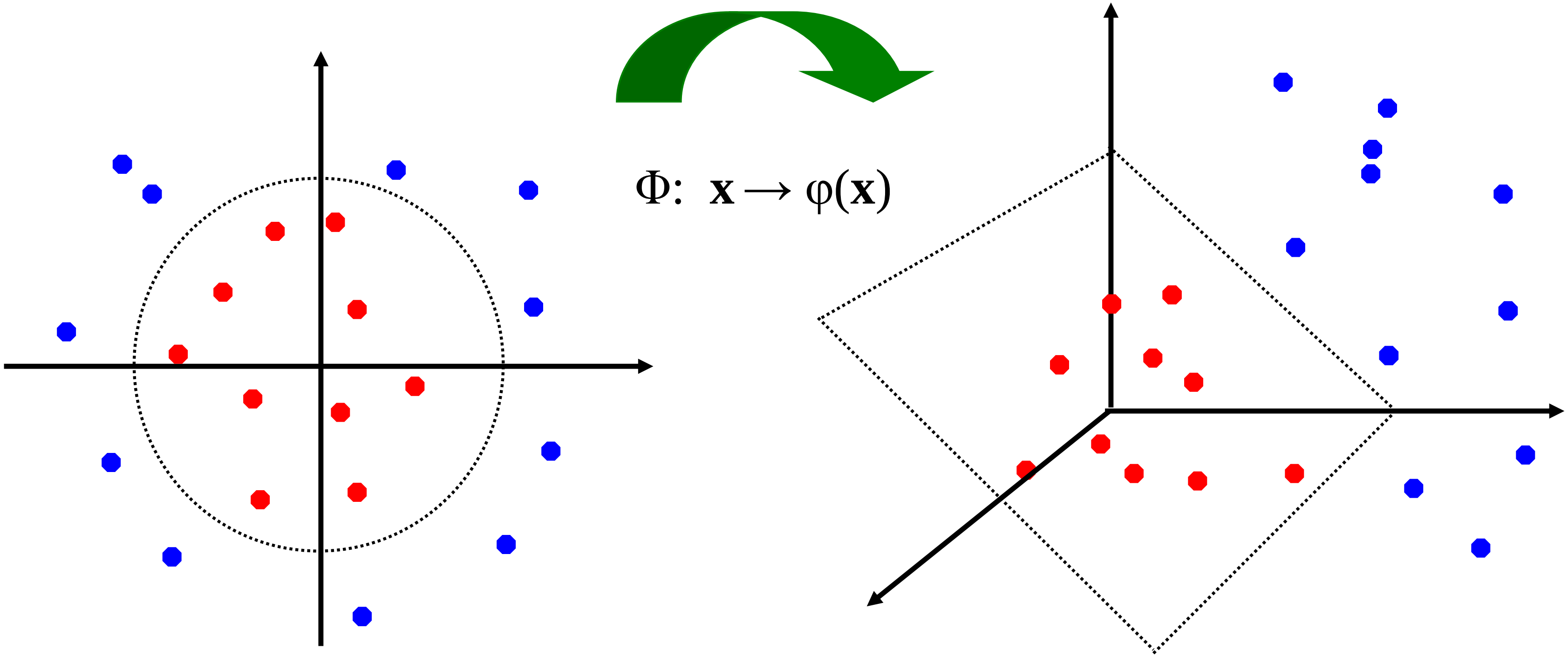
- We can map it to a higher-dimensional space:



Andrew Moore

Nonlinear SVMs

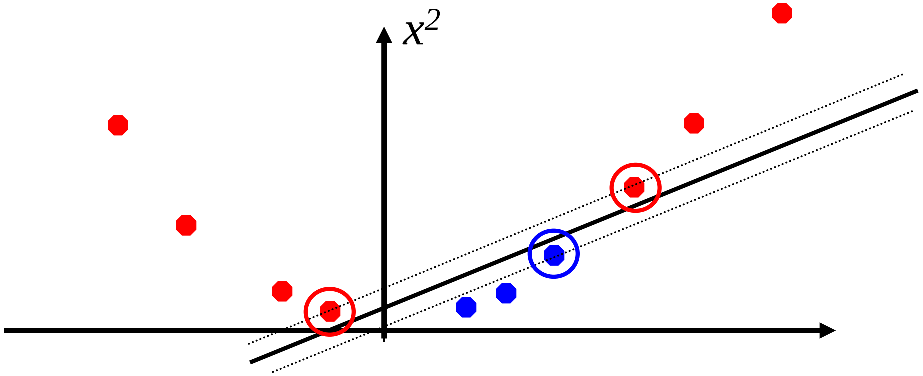
- General idea: the original input space can always be mapped to some higher-dimensional feature space where the training set is separable:



$\Phi\colon\ \mathbf{x} \rightarrow \varphi(\mathbf{x})$

Nonlinear kernel: Example

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



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

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

Svetlana Lazebnik

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 Φ: $\mathbf{x}_i \rightarrow \boldsymbol{\varphi}(\mathbf{x}_i)$, the dot product becomes: $K(\mathbf{x}_i,\mathbf{x}_j) = \boldsymbol{\varphi}(\mathbf{x}_i) \cdot \boldsymbol{\varphi}(\mathbf{x}_j)$
- 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 $\boldsymbol{\varphi}(\mathbf{x})$, define a kernel function K such that: $K(\mathbf{x}_i,\mathbf{x}_j) = \boldsymbol{\varphi}(\mathbf{x}_i) \cdot \boldsymbol{\varphi}(\mathbf{x}_j)$

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

$$\underset{\text{The } w \text{ that minimizes...}}{\boxed{\min_{\boldsymbol{w}}}} \quad \underbrace{\frac{1}{2}\|\boldsymbol{w}\|^2}_{\text{Maximize margin}}$$

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

Soft-margin SVMs

\# data samples

Misclassification cost

Slack variable

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

The *w* that minimizes…

$$\text{subject to} \quad y_i\boldsymbol{w}^T\boldsymbol{x}_i \geq 1 - \xi_i,$$

$$\xi_i \geq 0, \quad \forall i = 1, \ldots, N$$

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

Svetlana Lazebnik

Multi-class problems

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

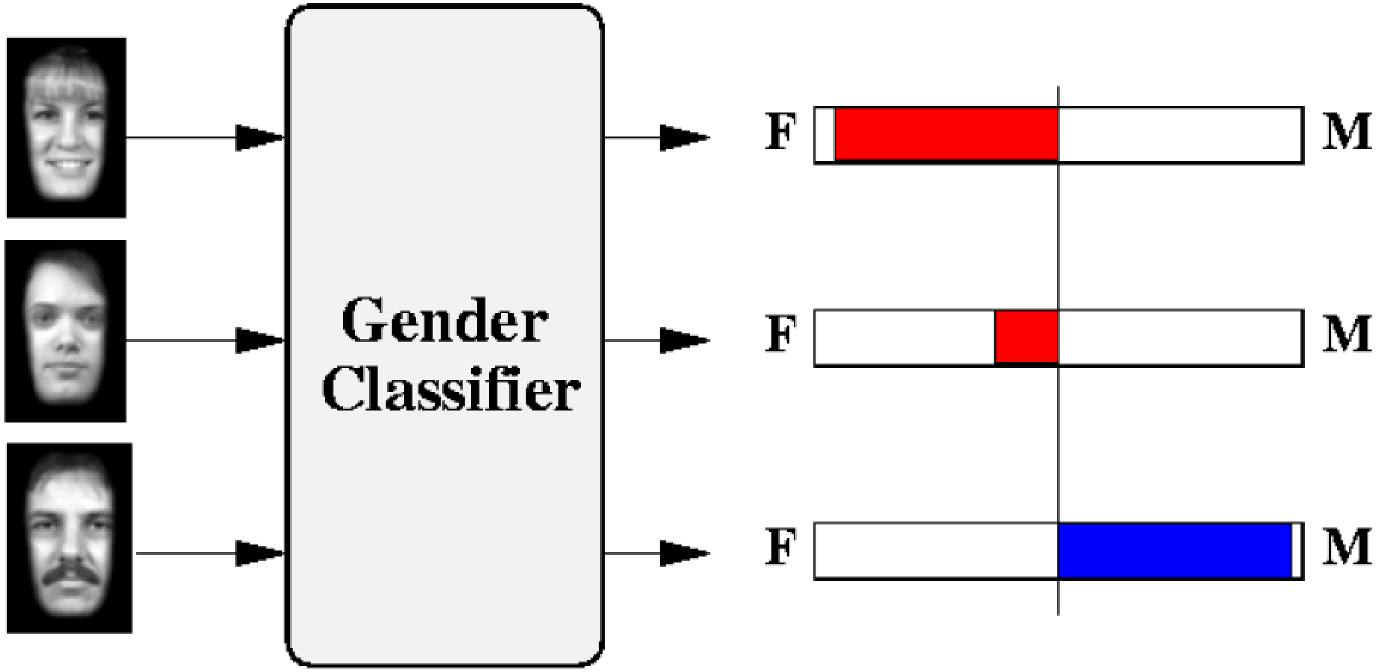
- Train K classifiers
- In each, pos = data from class *i*, neg = data from classes other than *i*
- The class with the most confident prediction wins
- Example:
  - You have 4 classes, train 4 classifiers
  - 1 vs others: score 3.5
  - 2 vs others: score 6.2
  - 3 vs others: score 1.4
  - 4 vs other: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

Example: Learning gender w/ SVMs



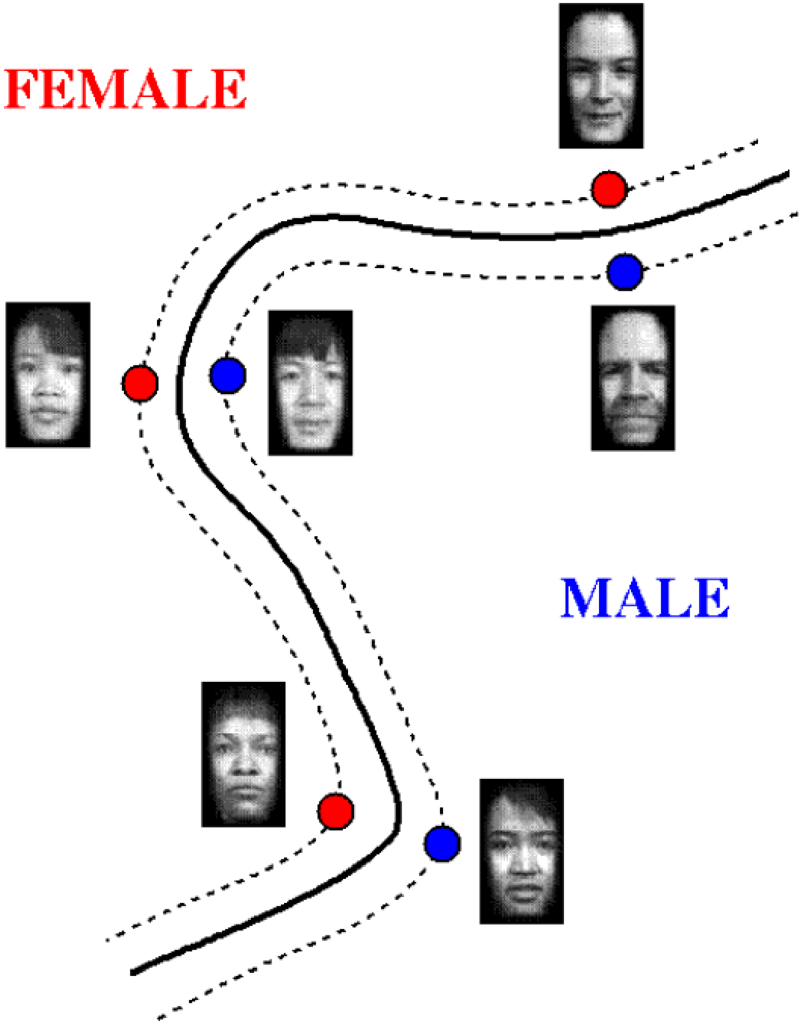
Moghaddam and Yang, Learning Gender with Support Faces, TPAMI 2002

Moghaddam and Yang, Face & Gesture 2000

Kristen Grauman

Example: Learning gender w/ SVMs

Support faces



Kristen Grauman

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)

Adapted from L. Lazebnik

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

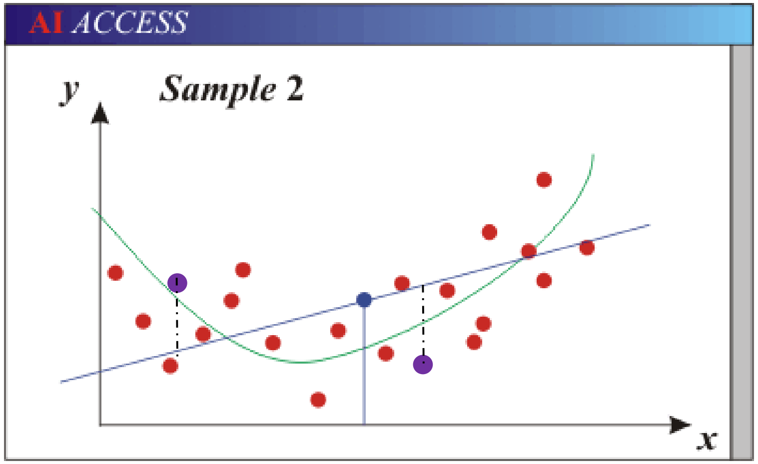
Training set (labels known)

Test set (labels unknown)

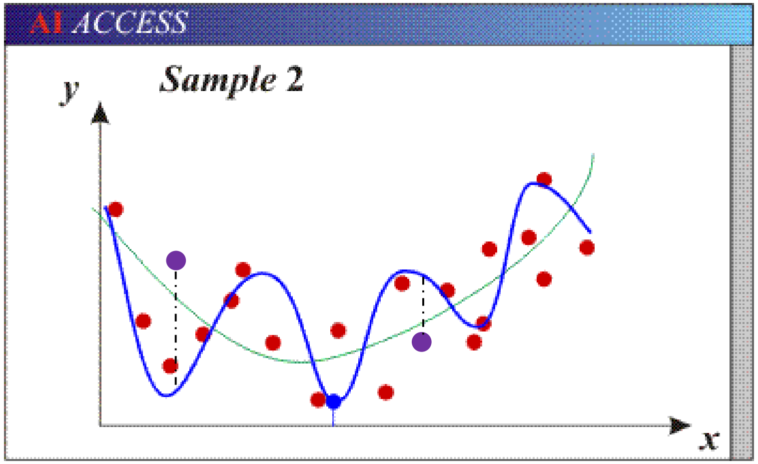
- How well does a learned model generalize from the data it was trained on to a new test set?

Slide credit: L. Lazebnik

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

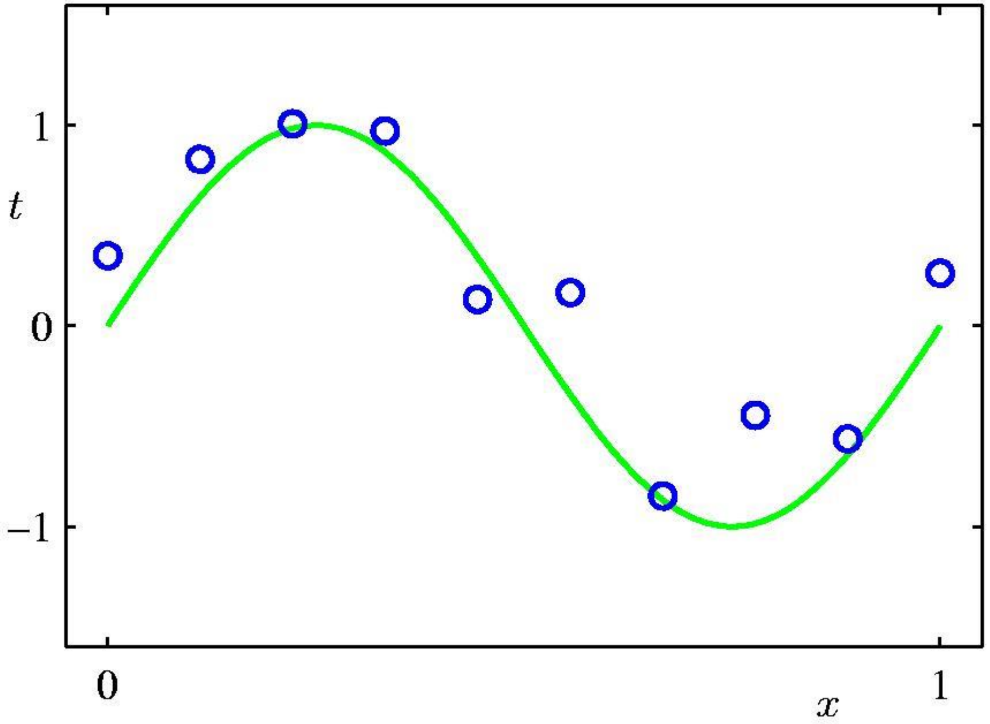
Adapted from D. Hoiem

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

Slide credit: L. Lazebnik

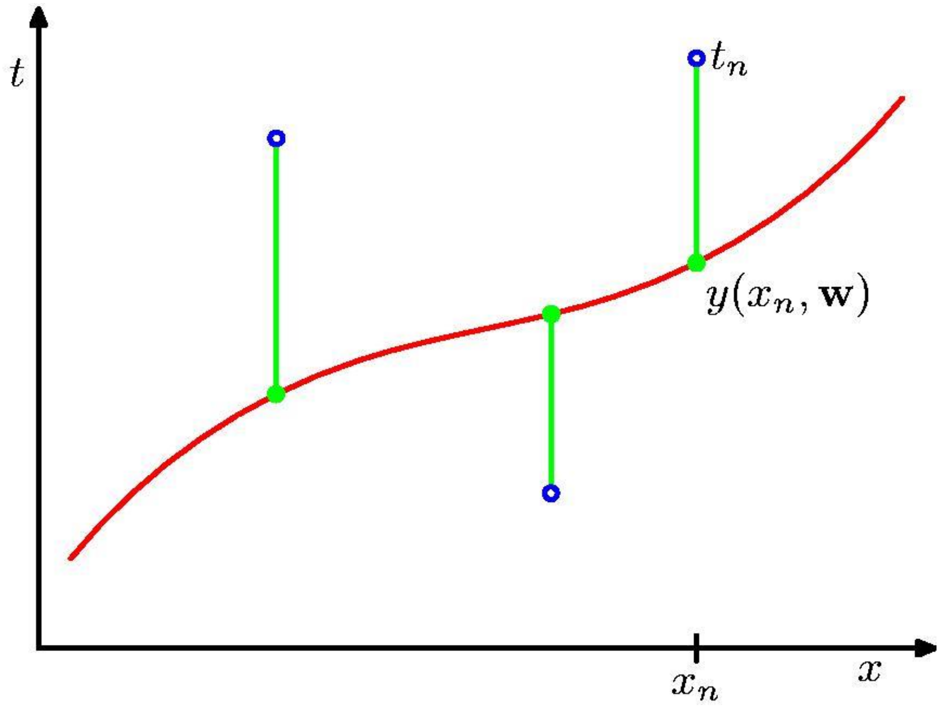
Polynomial Curve Fitting



$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

Slide credit: Chris Bishop

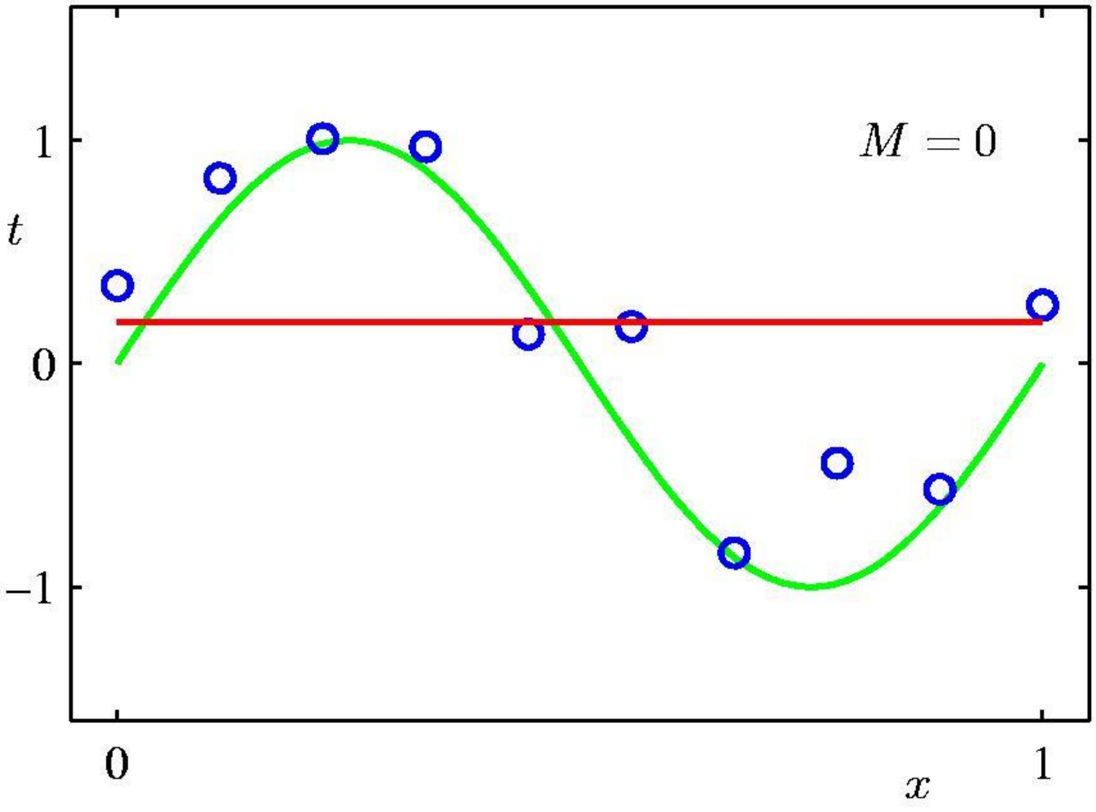
Sum-of-Squares Error Function



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

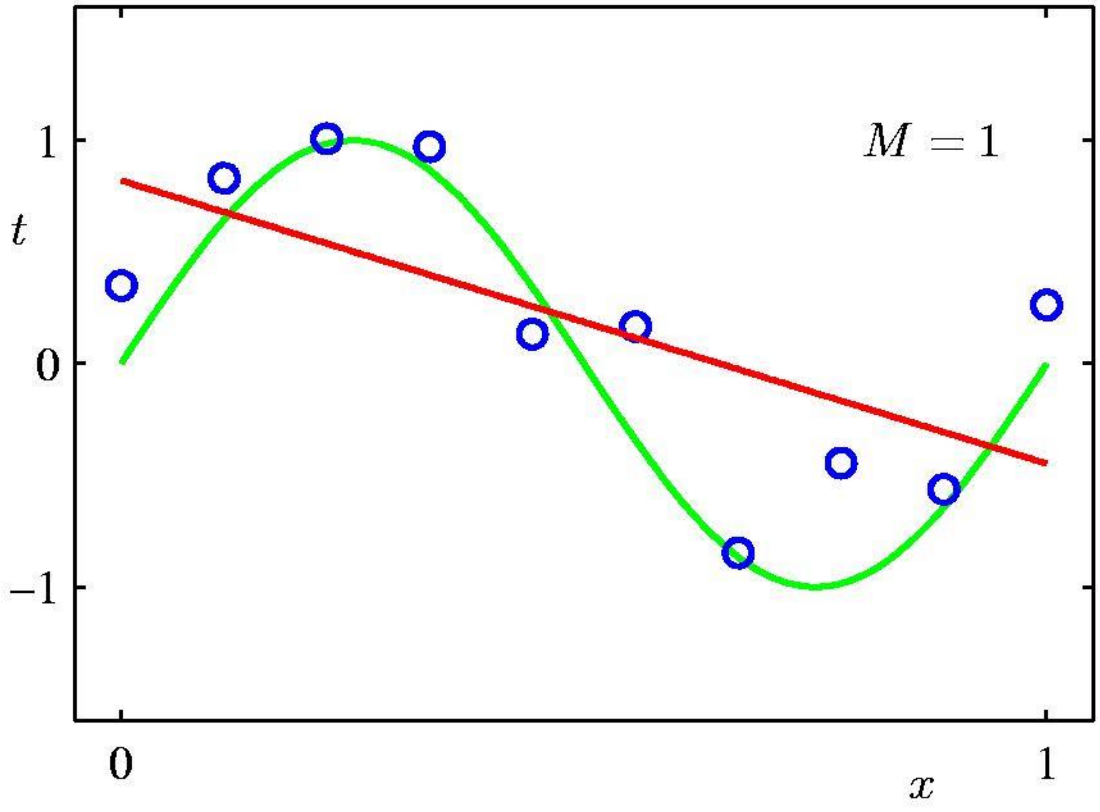
Slide credit: Chris Bishop

0^{th} Order Polynomial



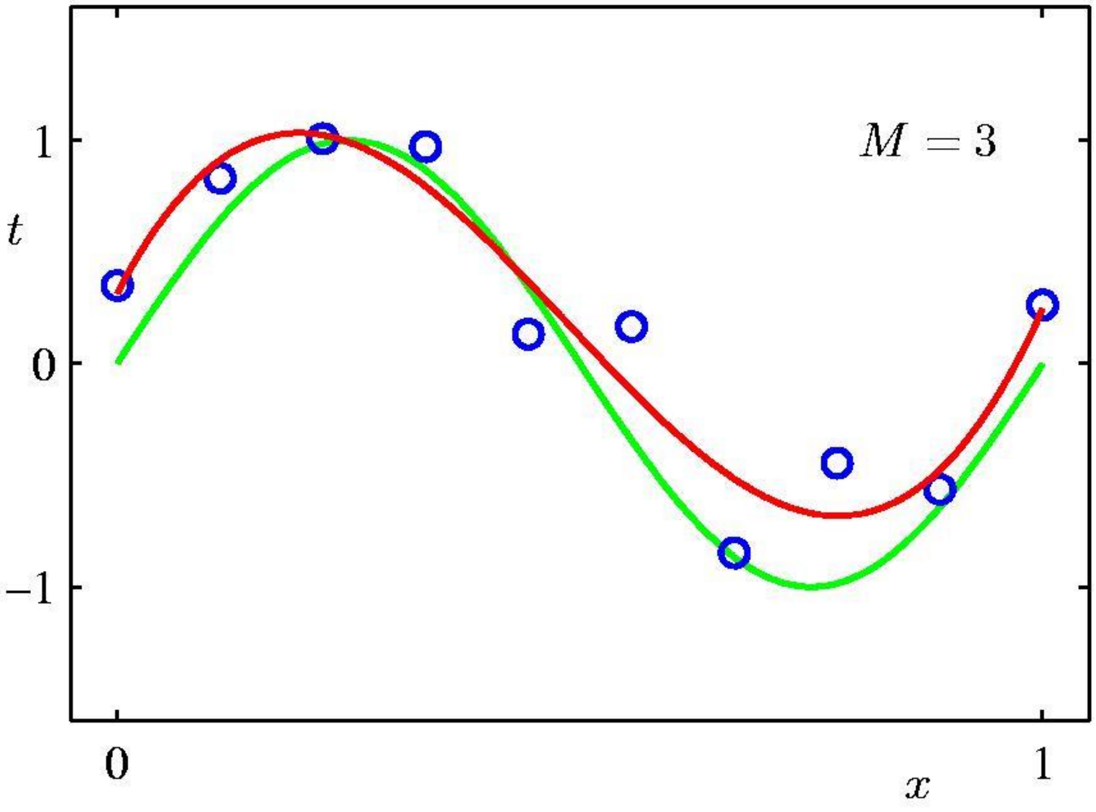
Slide credit: Chris Bishop

1^{st} Order Polynomial



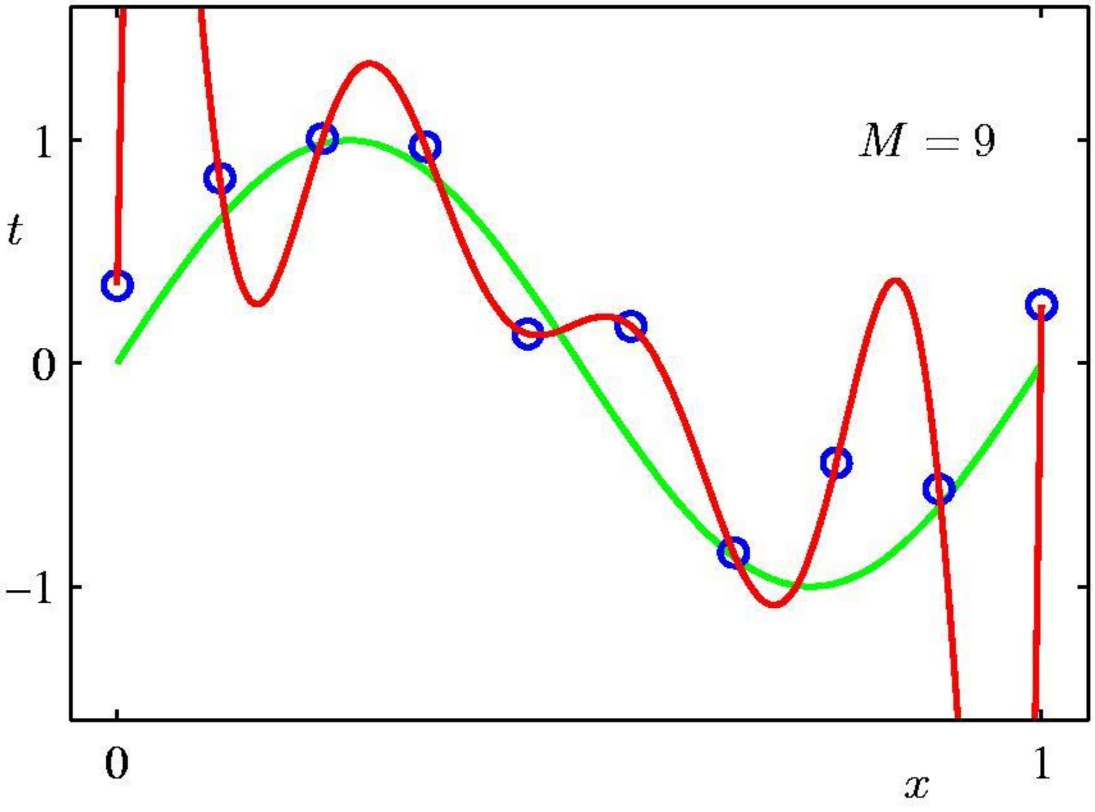
Slide credit: Chris Bishop

3^{rd} Order Polynomial



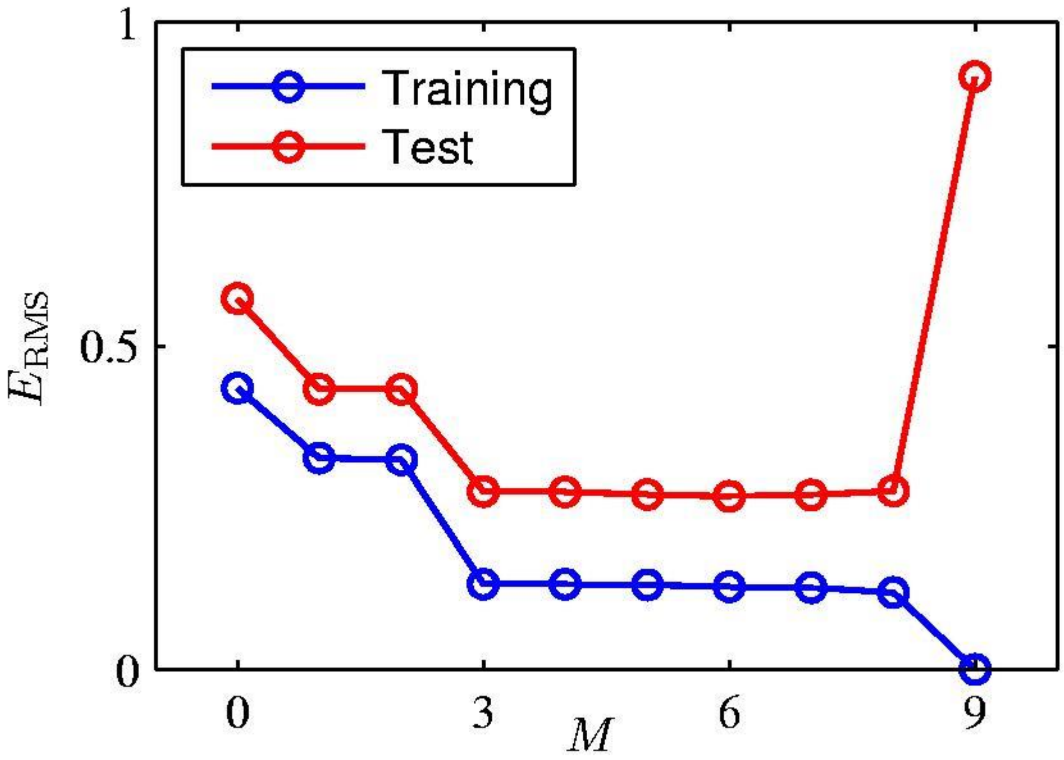
Slide credit: Chris Bishop

9th Order Polynomial



Slide credit: Chris Bishop

Over-fitting

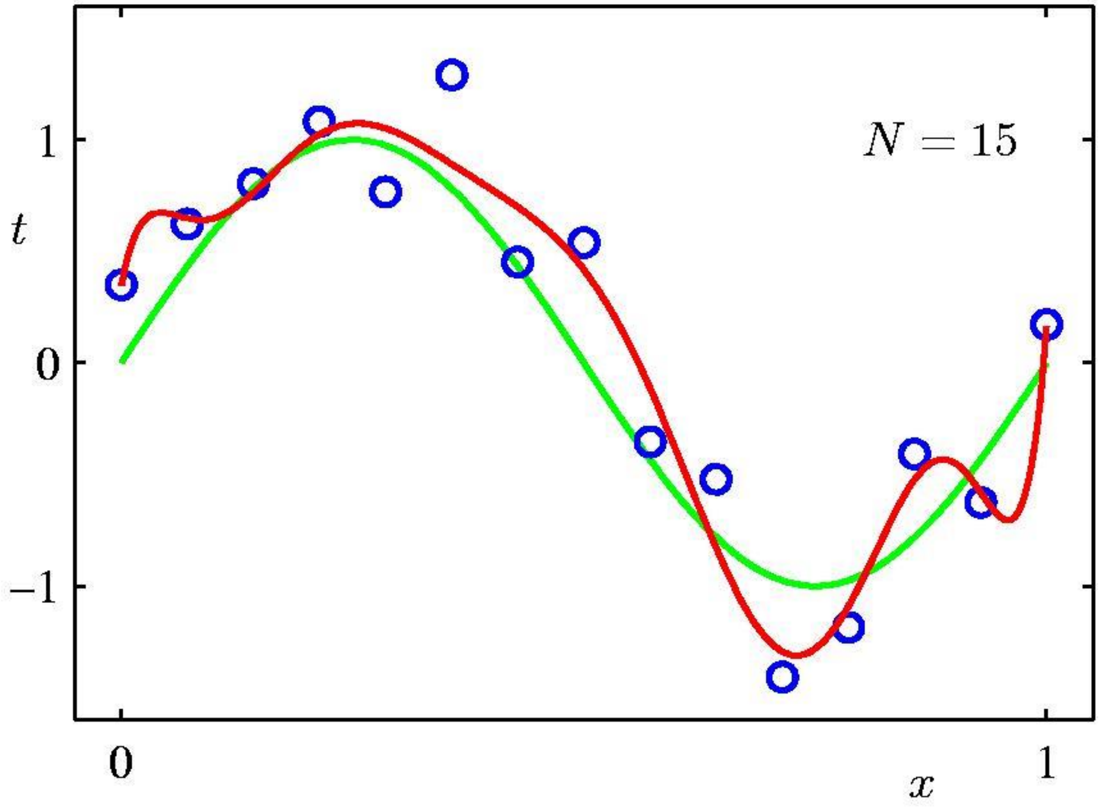


Root-Mean-Square (RMS) Error: $E_{\mathrm{RMS}} = \sqrt{2E(\mathbf{w}^\star)/N}$

Slide credit: Chris Bishop

Data Set Size: $N = 15$

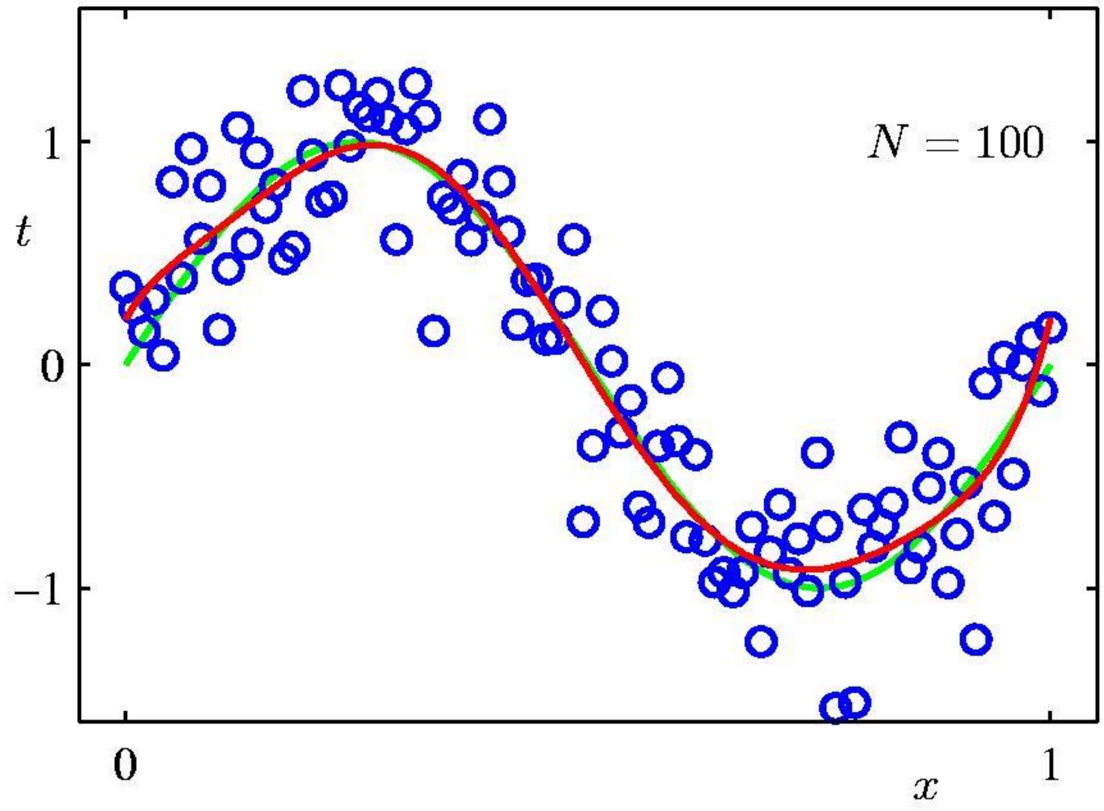
9th Order Polynomial



Slide credit: Chris Bishop

Data Set Size: $N = 100$

9th Order Polynomial



Slide credit: Chris Bishop

Regularization

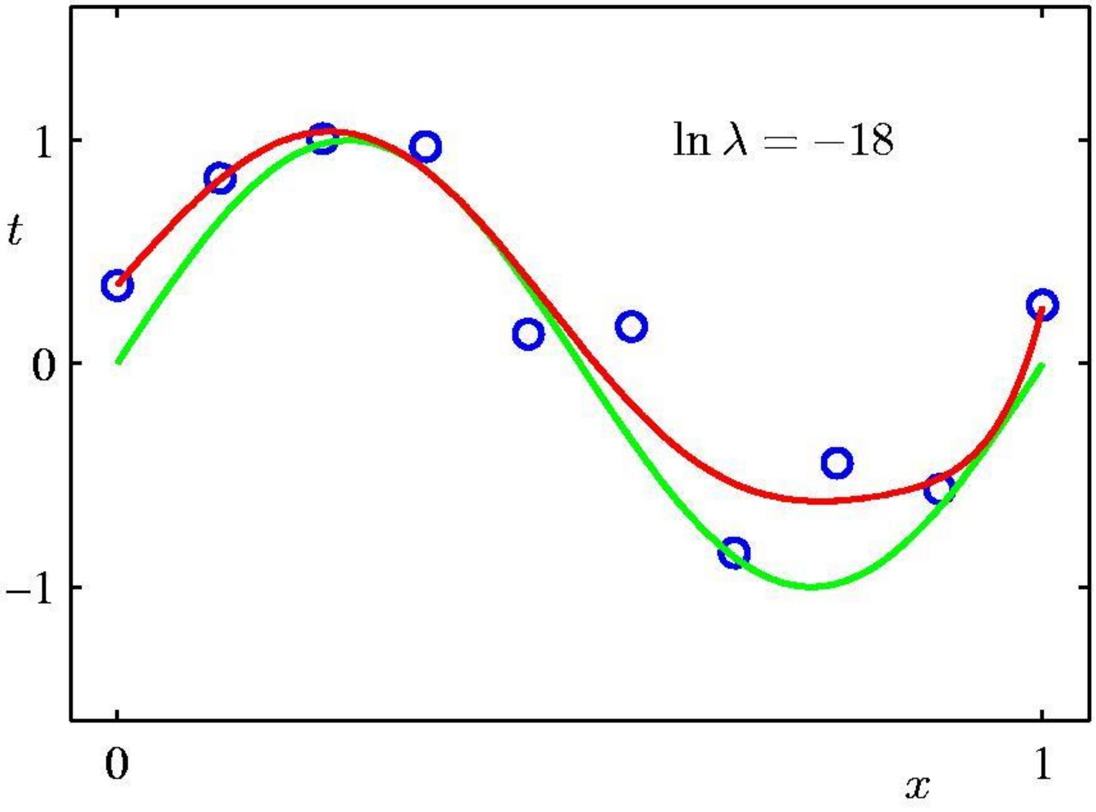
Penalize large coefficient values

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

(Remember: We want to minimize this expression.)

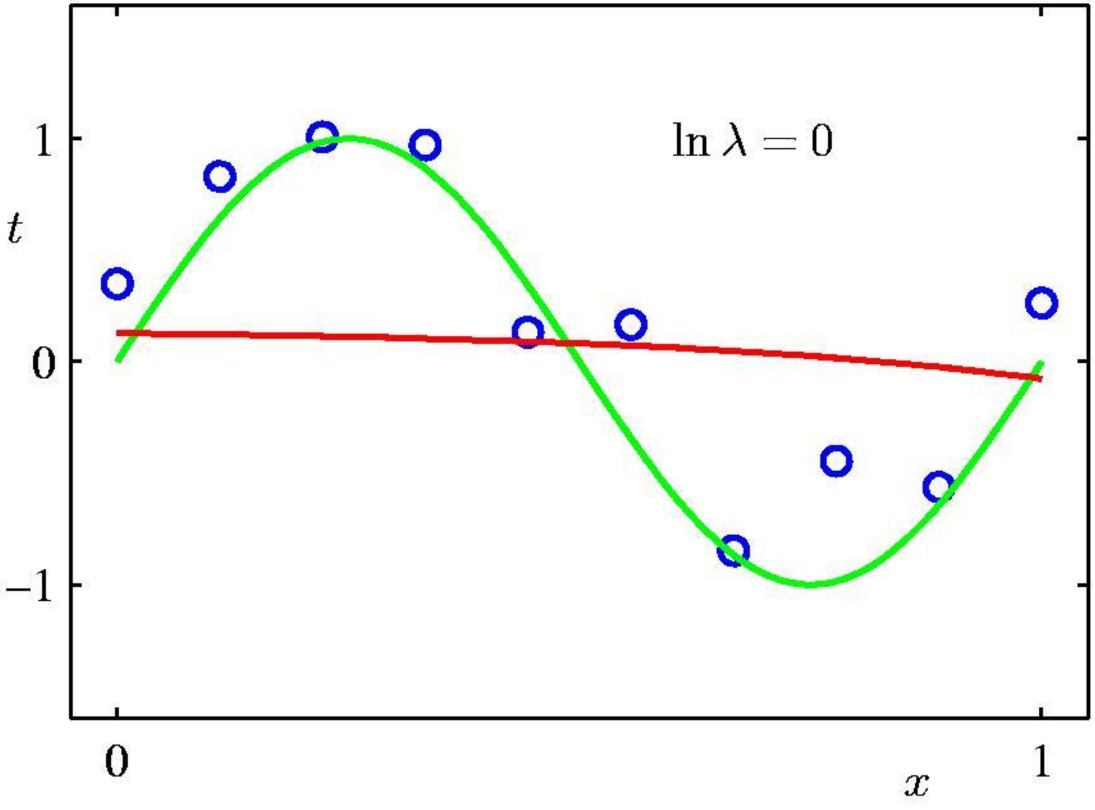
Adapted from Chris Bishop

Regularization: $\ln \lambda = -18$



Slide credit: Chris Bishop

Regularization: $\ln \lambda = 0$



Slide credit: Chris Bishop

Polynomial Coefficients

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

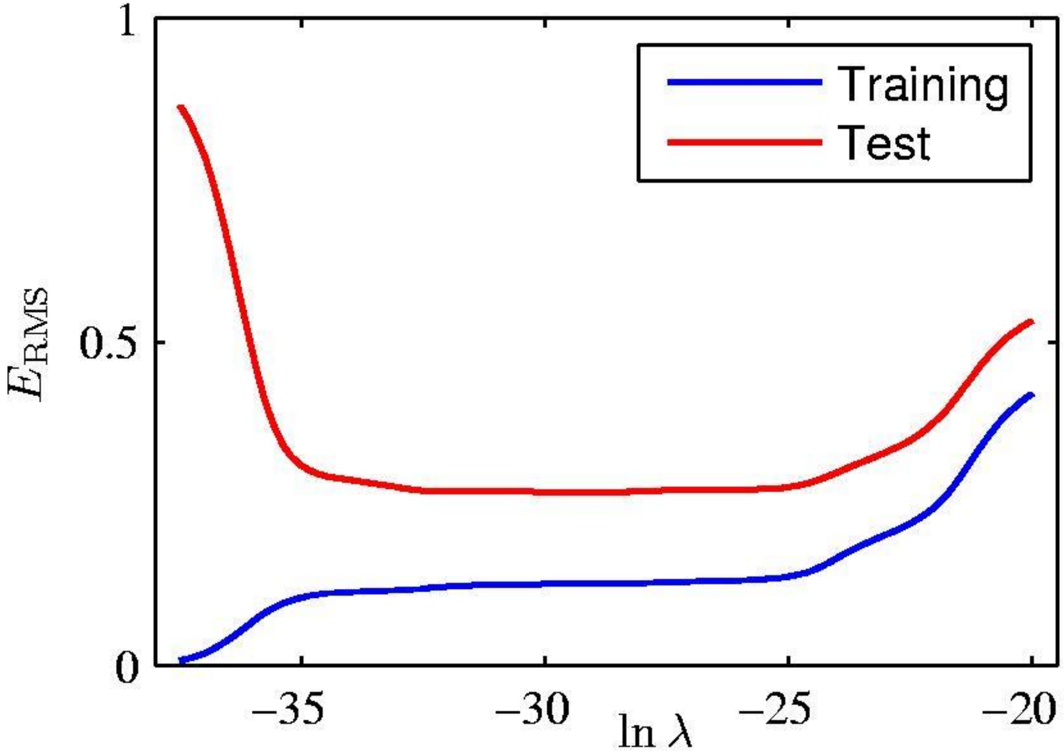
Slide credit: Chris Bishop

Polynomial Coefficients

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

Adapted from Chris Bishop

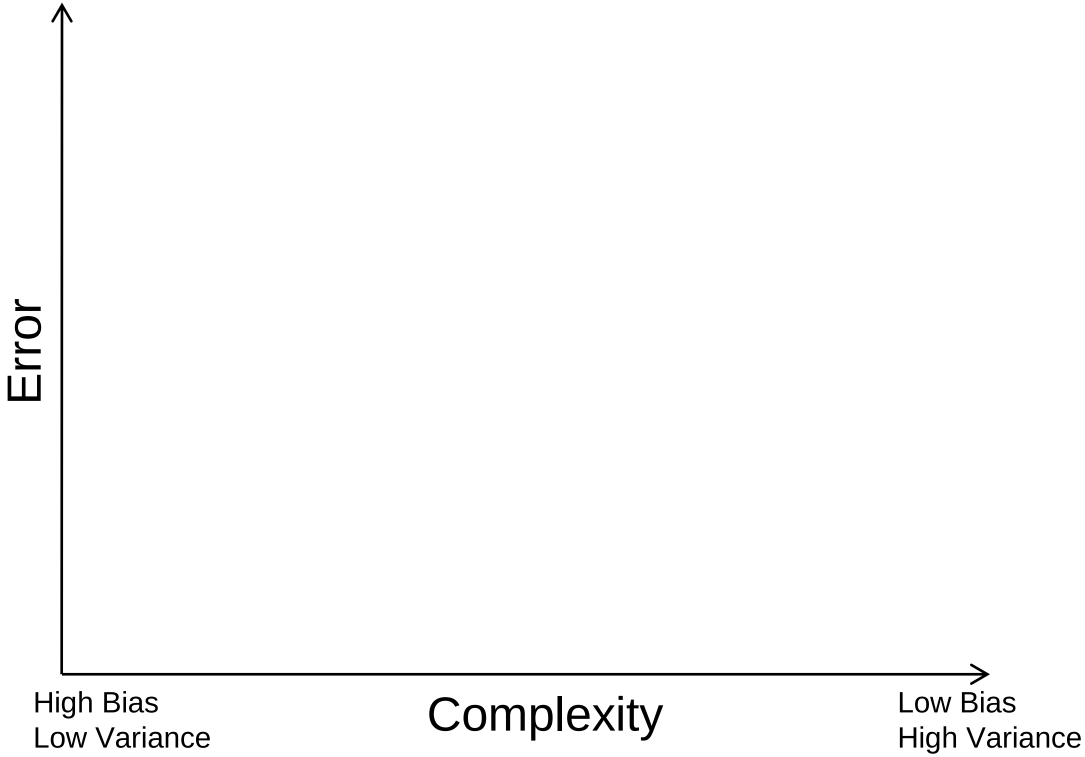
Regularization: E_{RMS} vs. $\ln \lambda$



Slide credit: Chris Bishop

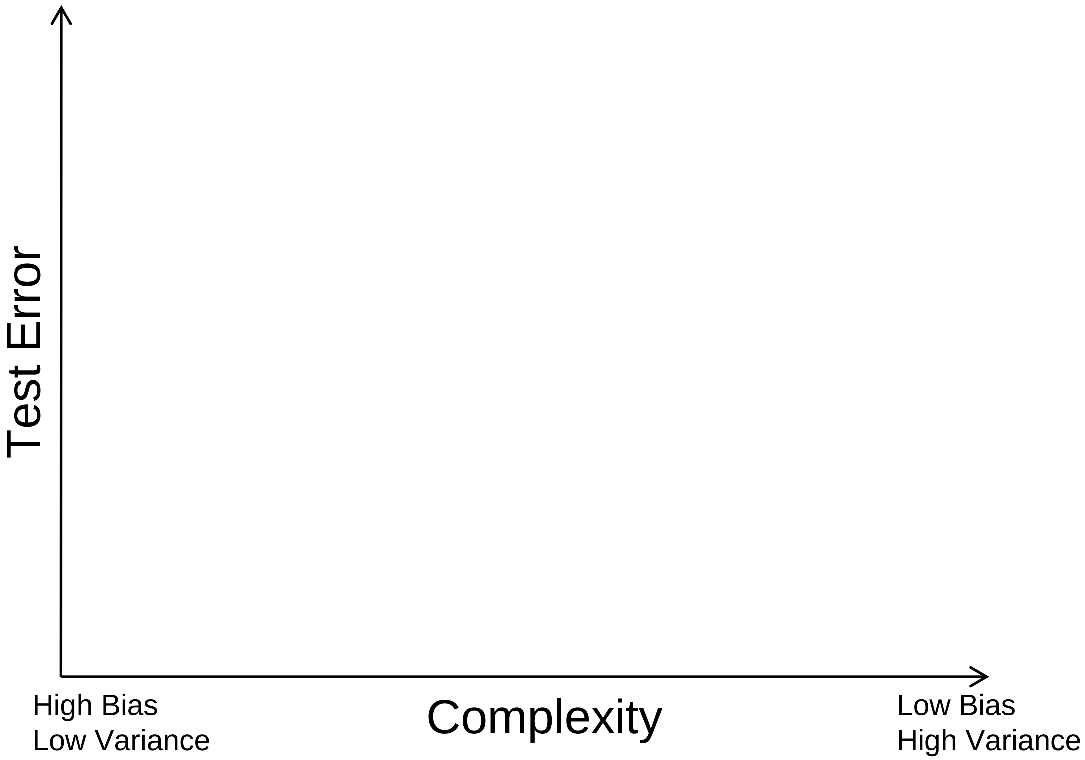
Training vs test error

Underfitting **Overfitting**



Slide credit: D. Hoiem

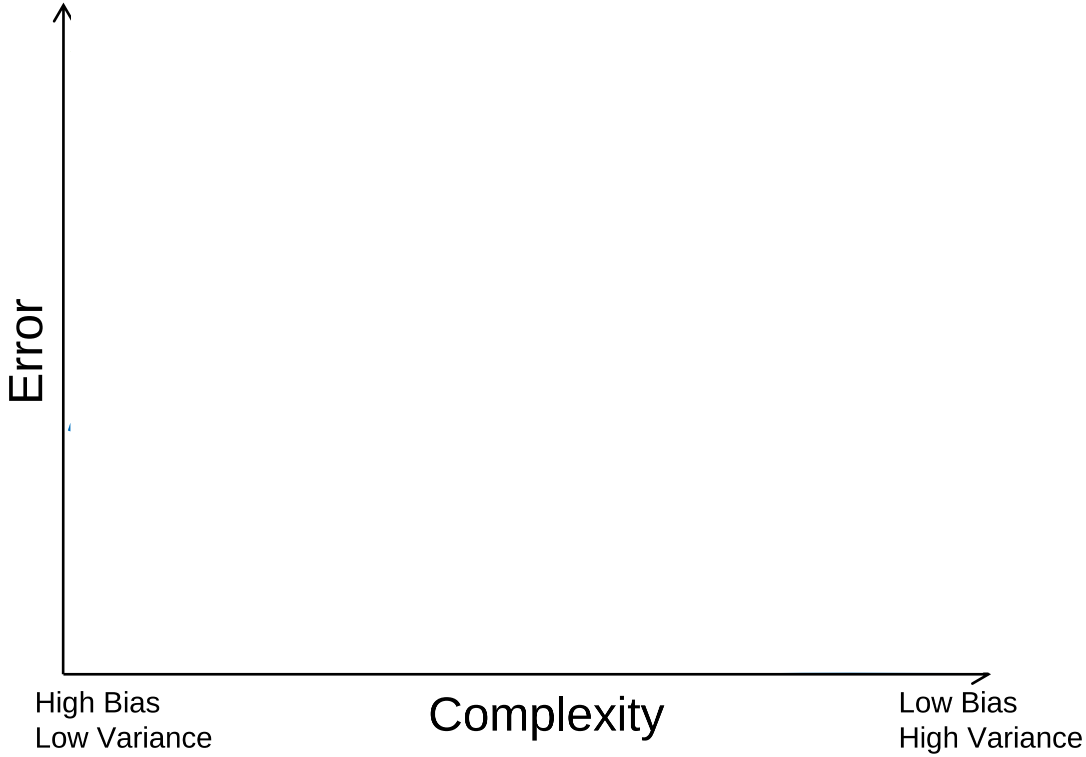
The effect of training set size



Slide credit: D. Hoiem

Choosing the trade-off between bias and variance

- Need validation set (separate from the test set)



Slide credit: D. Hoiem

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

Slide credit: D. Hoiem

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]

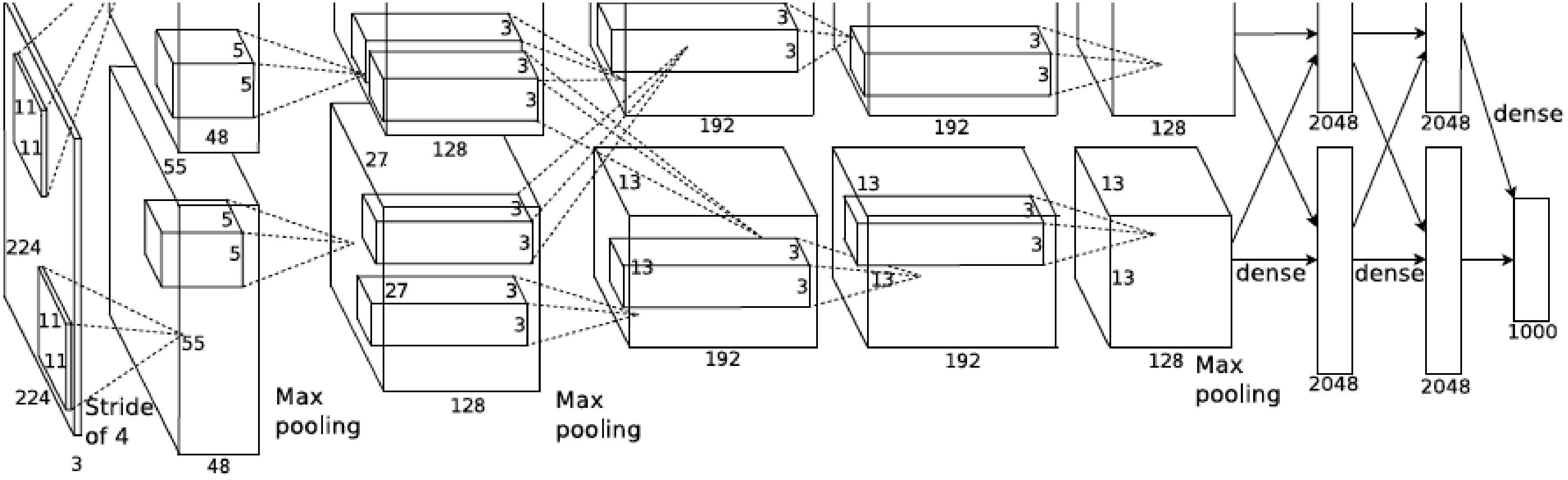
- ~14 million labeled images, 20k classes
- Images gathered from Internet
- Human labels via Amazon Turk
- Challenge: 1.2 million training images, 1000 classes

A. Krizhevsky, I. Sutskever, and G. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

Lana Lazebnik

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)



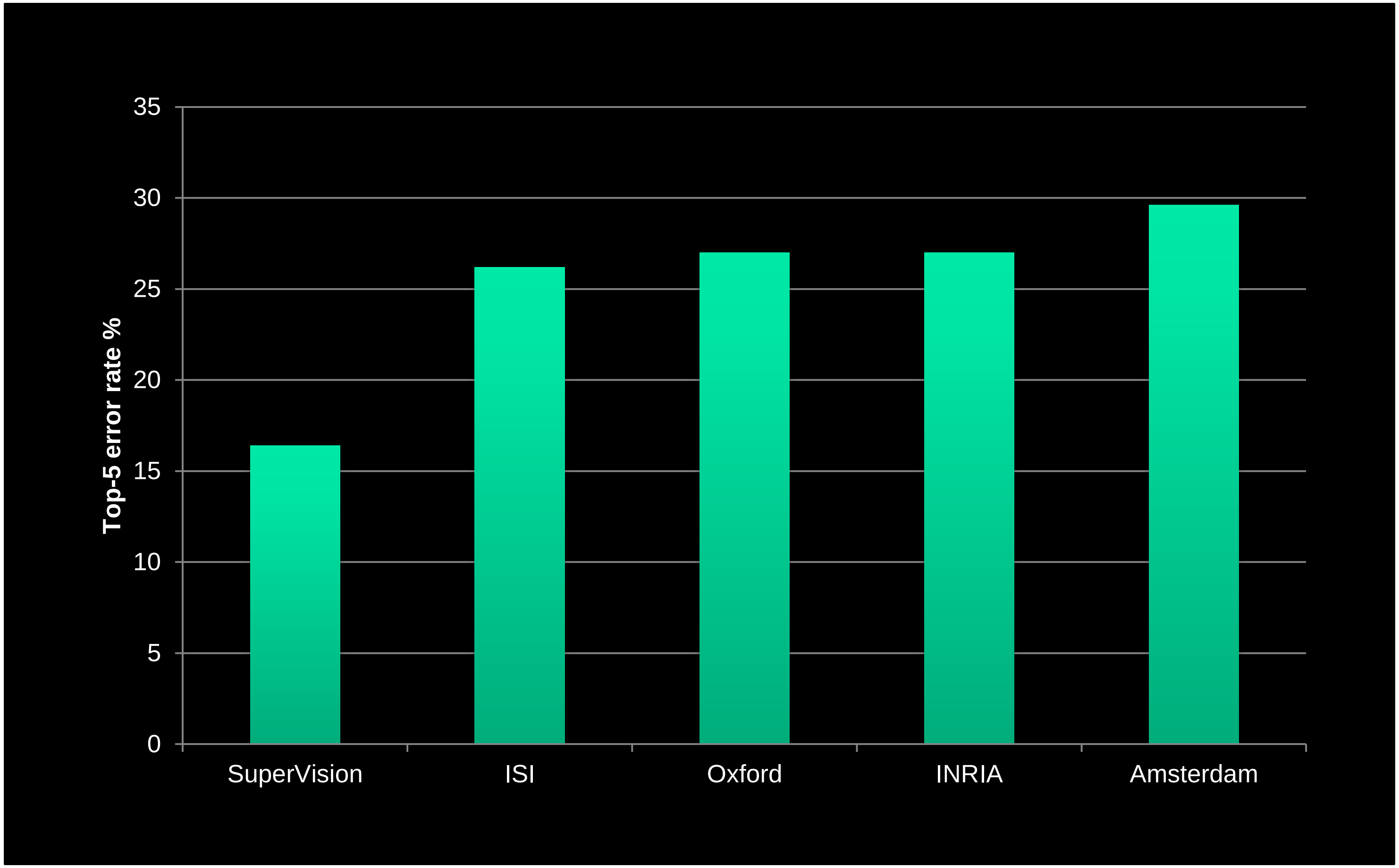
A. Krizhevsky, I. Sutskever, and G. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

Adapted from Lana Lazebnik

ImageNet Challenge 2012

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

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



Lana Lazebnik

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*

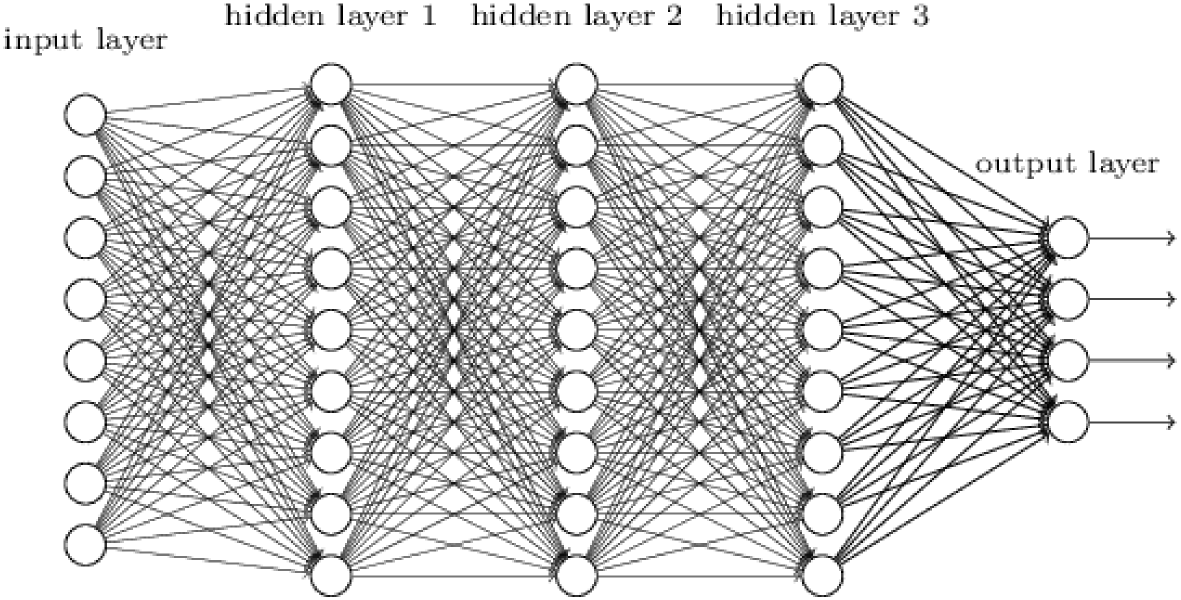
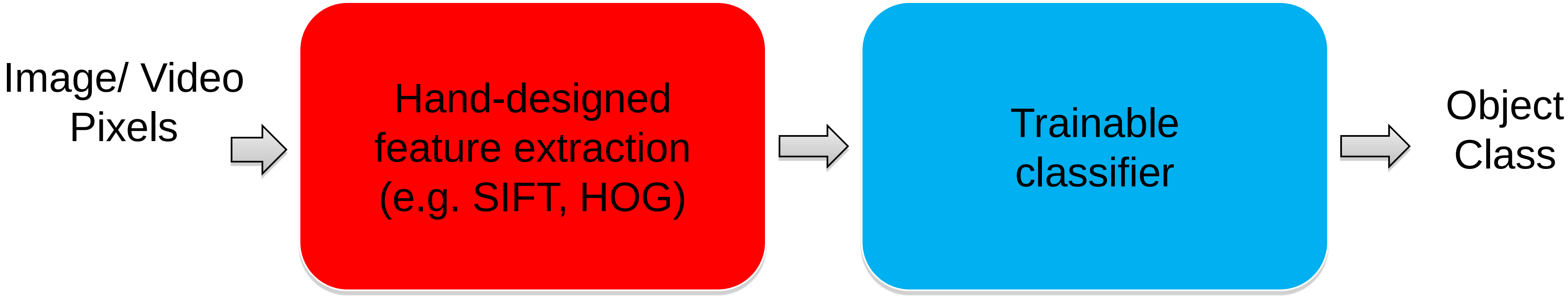


Figure from http://neuralnetworksanddeeplearning.com/chap5.html

Traditional Recognition Approach



Image/ Video Pixels

Hand-designed feature extraction (e.g. SIFT, HOG)

Trainable classifier

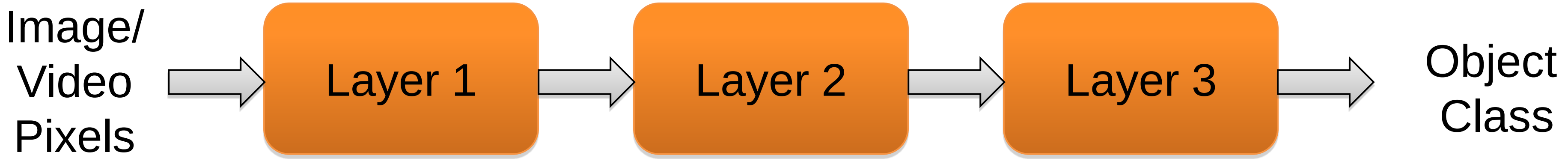
Object Class

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

Adapted from Lana Lazebnik

What about learning the features?

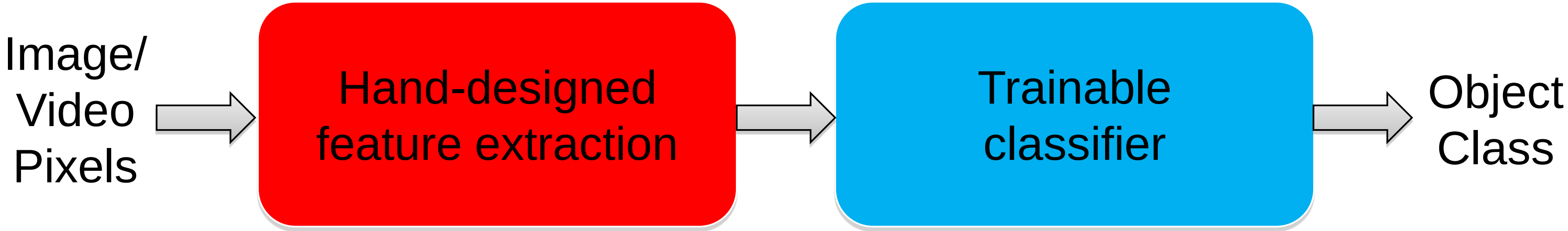
- Learn a *feature hierarchy* all the way from pixels to classifier
- Each layer extracts features from the output of previous layer
- Train all layers jointly



Lana Lazebnik

“Shallow” vs. “deep” architectures

Traditional recognition: “Shallow” architecture



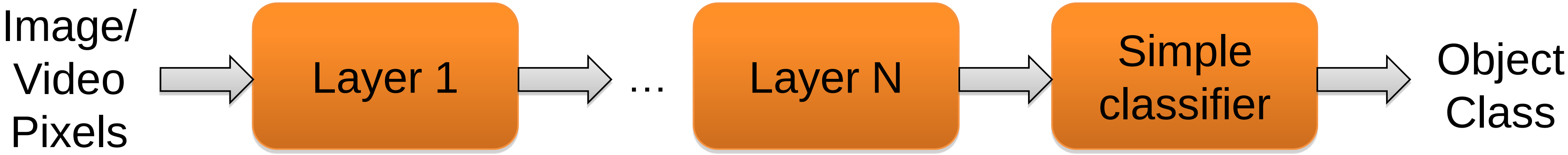
Image/ Video Pixels

Hand-designed feature extraction

Trainable classifier

Object Class

Deep learning: “Deep” architecture



Image/ Video Pixels

Layer 1

…

Layer N

Simple classifier

Object Class

Lana Lazebnik

Neural network definition

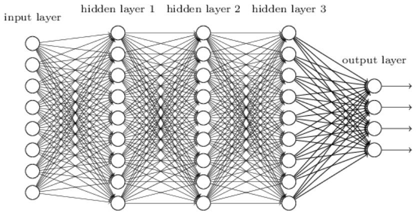
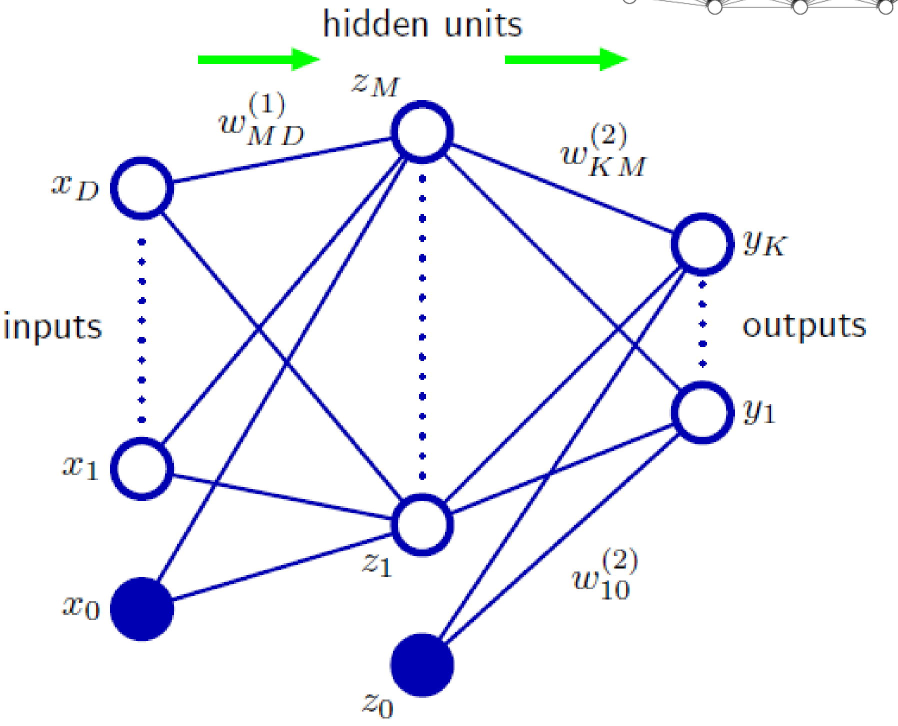


Figure 5.1 Network diagram for the two-layer neural network corresponding to (5.7). The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes, in which the bias parameters are denoted by links coming from additional input and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



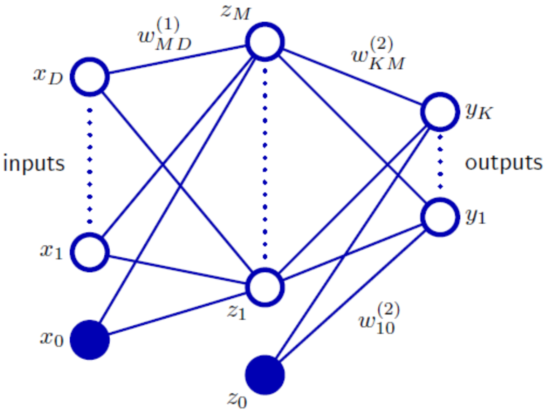
- Activations: $$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

Recall SVM:
$w^T x + b$

- Nonlinear activation function *h* (e.g. sigmoid, RELU): $$z_j = h(a_j)$$

Figure from Christopher Bishop

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)

(binary) $y_k = \sigma(a_k) = \dfrac{1}{1+\exp(-a_k)}$

(multiclass) $y_k = \dfrac{\exp(a_k)}{\sum_j \exp(a_j)}$

- Finally:

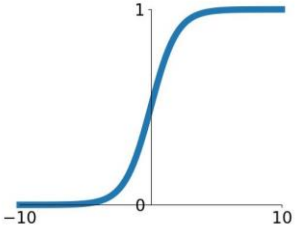
(binary)

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

Activation functions

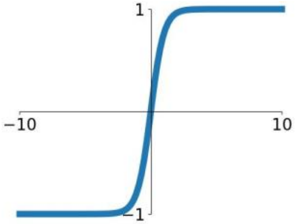
Sigmoid

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



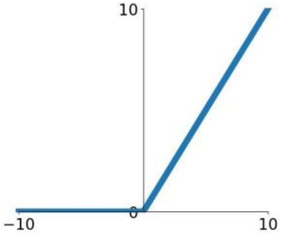
tanh

$$\tanh(x)$$



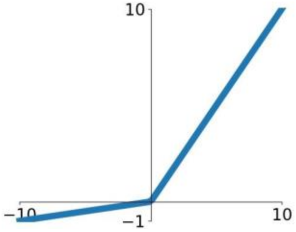
ReLU

$$\max(0, x)$$



Leaky ReLU

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

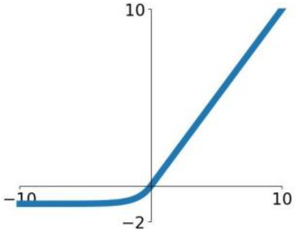


Maxout

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

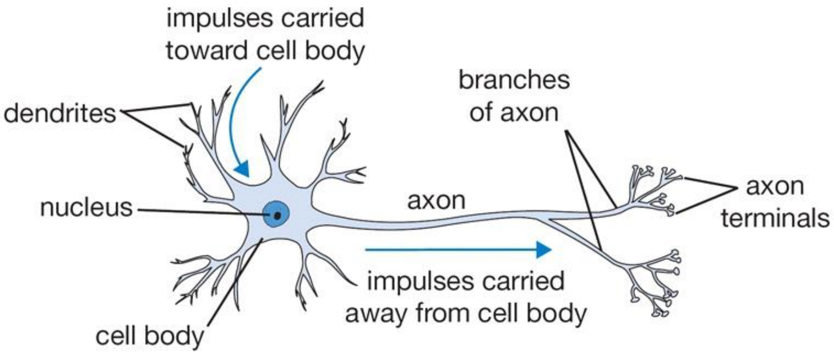
ELU

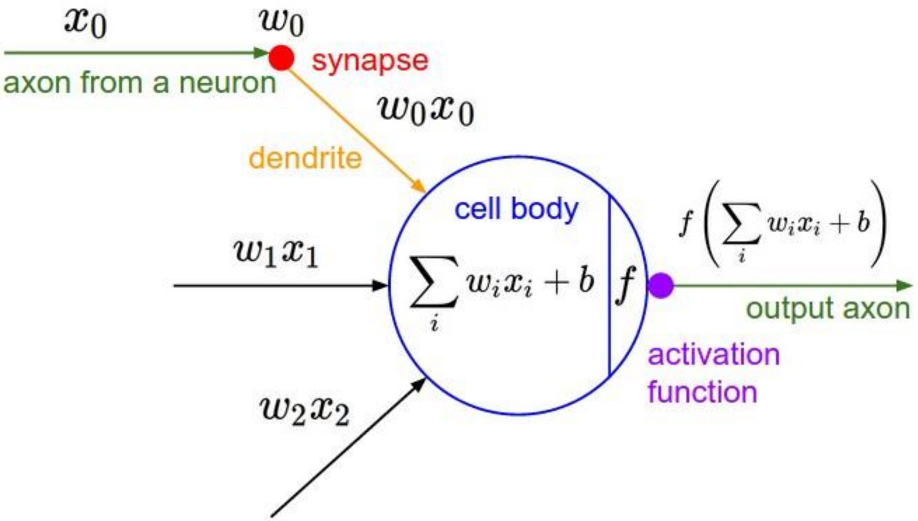
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



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”





Text: HKUST, figures: Andrej Karpathy

Deep neural networks

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

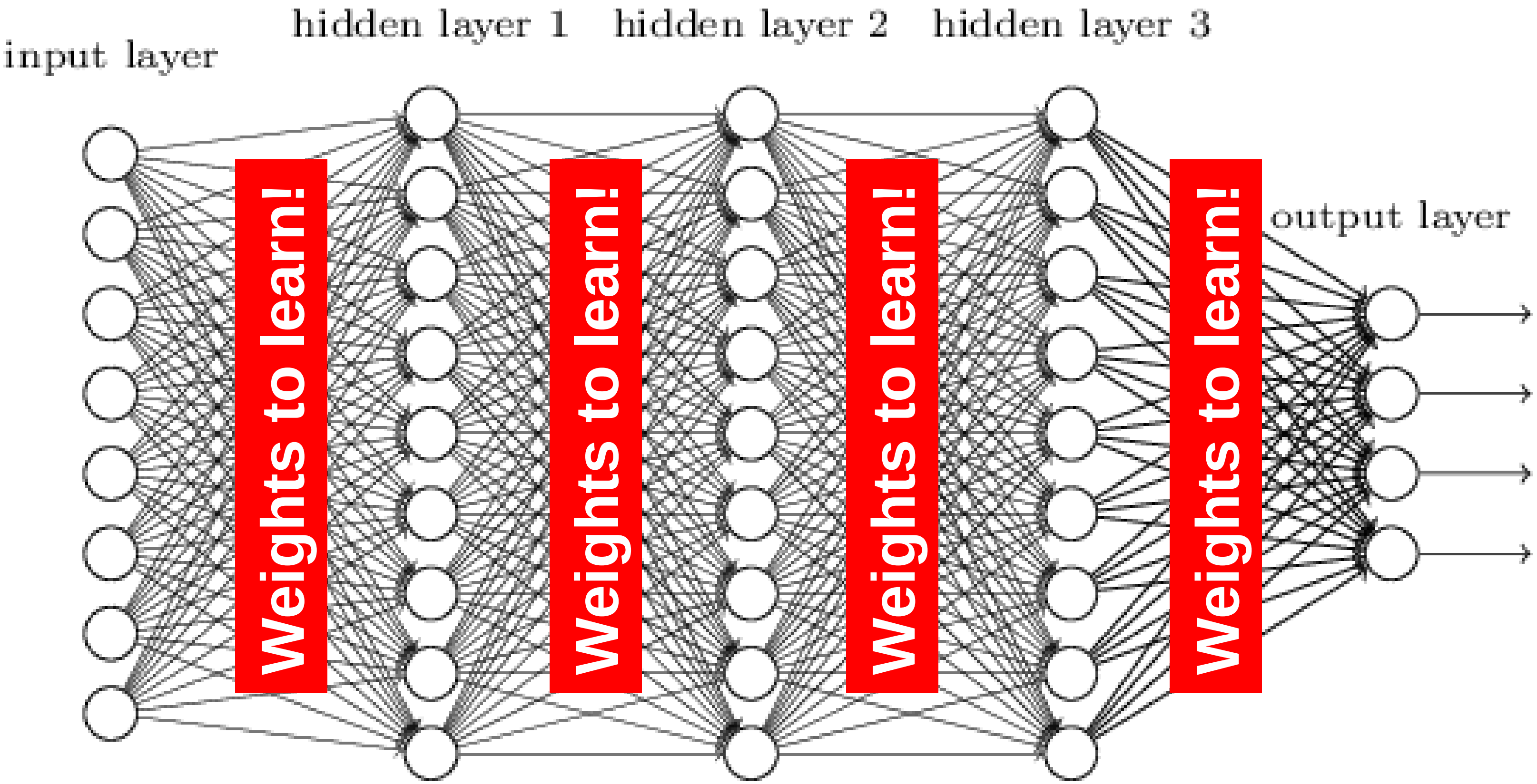
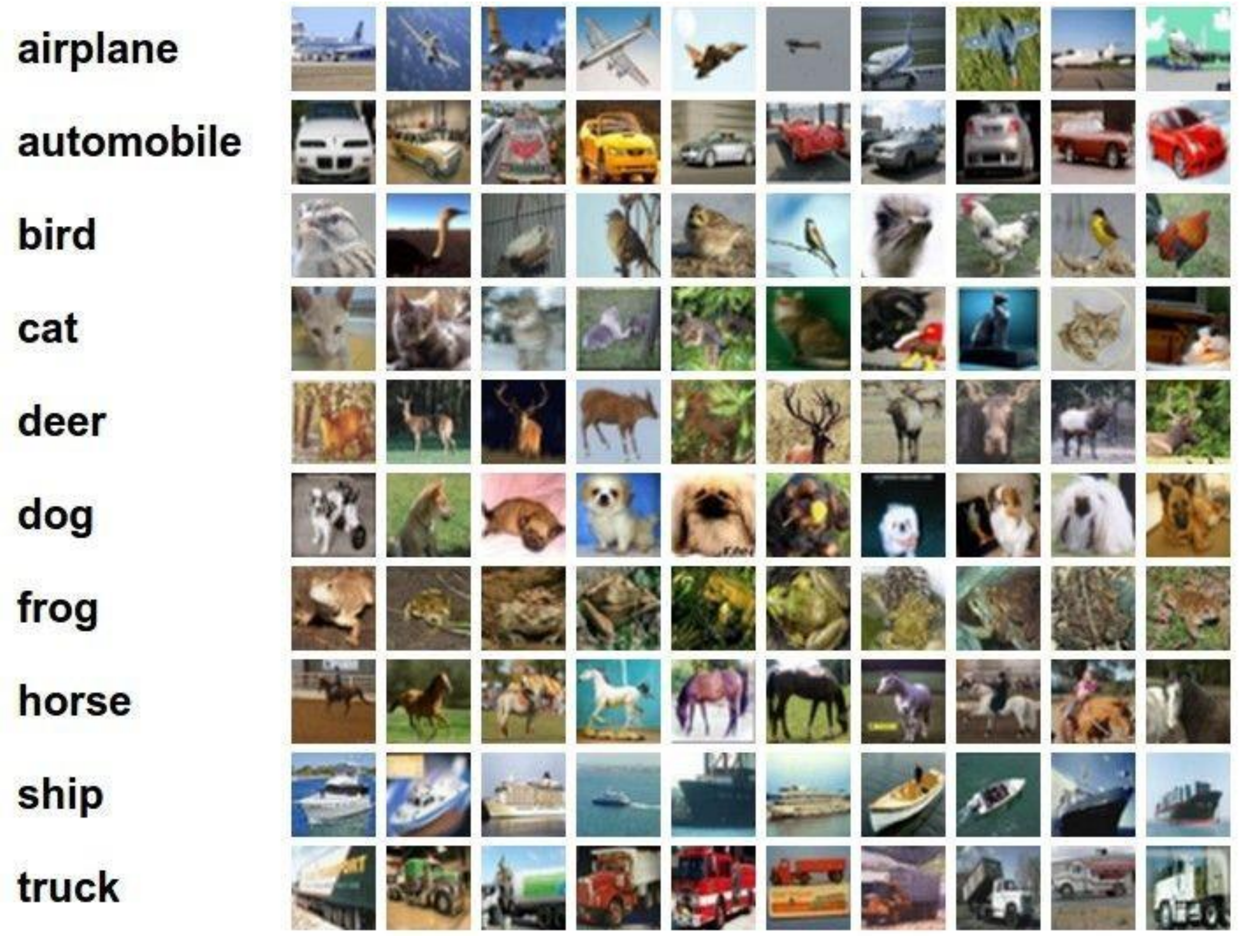


Figure from http://neuralnetworksanddeeplearning.com/chap5.html

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



Example dataset: **CIFAR-10**

10 labels

50,000 training images

each image is **32x32x3**

10,000 test images.

Andrej Karpathy

Classification scores

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

[32x32x3]
array of numbers 0...1
(3072 numbers total)

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

10 numbers,
indicating class
scores

Andrej Karpathy

Linear classifier

$$\underbrace{\boxed{f(x,W)}}_{\text{10x1}} = \underbrace{\boxed{W}}_{\text{10x3072}}\boxed{x}\ \text{3072x1} \quad \boxed{(+b)}\ \text{10x1}$$

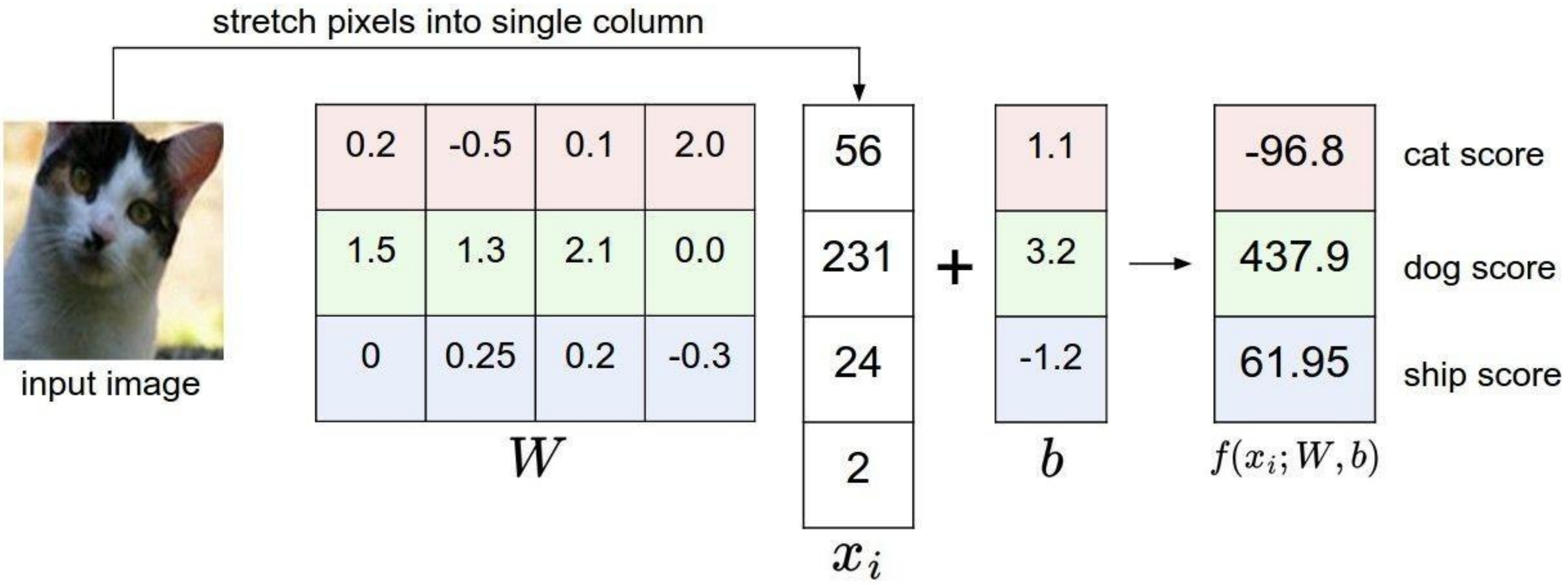
[32x32x3]
array of numbers 0...1

10 numbers, indicating class scores

parameters, or “weights”

Linear classifier

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



$$\begin{bmatrix}0.2 & -0.5 & 0.1 & 2.0\\ 1.5 & 1.3 & 2.1 & 0.0\\ 0 & 0.25 & 0.2 & -0.3\end{bmatrix}\begin{bmatrix}56\\ 231\\ 24\\ 2\end{bmatrix}+\begin{bmatrix}1.1\\ 3.2\\ -1.2\end{bmatrix}\longrightarrow\begin{bmatrix}-96.8\\ 437.9\\ 61.95\end{bmatrix}$$

Andrej Karpathy

Linear classifier

Going forward: Loss function/Optimization

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

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)**

Andrej Karpathy

Linear classifier

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

Adapted from Andrej Karpathy

Linear classifier: Hinge loss

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

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:

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

Want: $s_{y_i} >= s_j + 1$
i.e. $s_j - s_{y_i} + 1 <= 0$

If true, loss is 0
If false, loss is magnitude of violation

Adapted from Andrej Karpathy

Linear classifier: Hinge loss

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9		

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:

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

= max(0, 5.1 - 3.2 + 1)
+max(0, -1.7 - 3.2 + 1)
= max(0, 2.9) + max(0, -3.9)
= 2.9 + 0
= 2.9

Adapted from Andrej Karpathy

Linear classifier: Hinge loss

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
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:

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

$= \max(0, 1.3 - 4.9 + 1)$
$+ \max(0, 2.0 - 4.9 + 1)$
$= \max(0, -2.6) + \max(0, -1.9)$
$= 0 + 0$
$= 0$

Adapted from Andrej Karpathy

Linear classifier: Hinge loss

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
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:

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

= max(0, 2.2 - (-3.1) + 1)
+max(0, 2.5 - (-3.1) + 1)
= max(0, 5.3 + 1)
+ max(0, 5.6 + 1)
= 6.3 + 6.6
= 12.9

Adapted from Andrej Karpathy

Linear classifier: Hinge loss

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
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:

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

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 = **5.3**

Adapted from Andrej Karpathy

Linear classifier: Hinge loss

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

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

Adapted from Andrej Karpathy

Linear classifier: Hinge loss

Weight Regularization

λ = regularization strength (hyperparameter)

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

In common use:

L2 regularization

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

L1 regularization

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

Dropout (will see later)

Adapted from Andrej Karpathy

Another loss: Softmax (cross-entropy)

cat	**3.2**
car	5.1
frog	-1.7

scores = unnormalized log probabilities of the classes.

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

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: Softmax (cross-entropy)

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

	unnormalized log probabilities		unnormalized probabilities		probabilities
cat	**3.2**	exp →	**24.5**	normalize →	**0.13**
car	5.1		164.0		0.87
frog	-1.7		0.18		0.00

→ L_i = -log(0.13)
= **0.89**

Adapted from Andrej Karpathy

Other losses

- Triplet loss (Schroff, FaceNet)

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

a denotes anchor
p denotes positive
n denotes negative

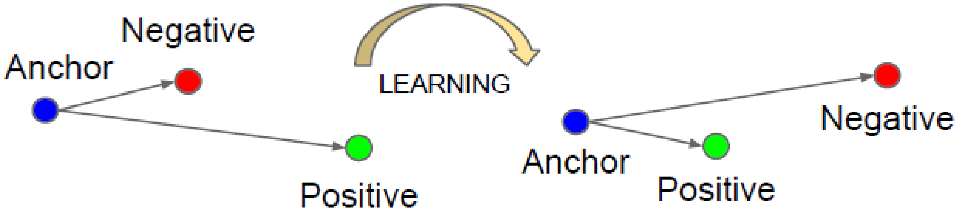


Figure 3. The **Triplet Loss** minimizes the distance between an *anchor* and a *positive*, both of which have the same identity, and maximizes the distance between the *anchor* and a *negative* of a different identity.

- Anything you want!

How to minimize the loss function?

How to minimize the loss function?

In 1-dimension, the derivative of a function:

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

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

Andrej Karpathy

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

dW = ...
(some function
data and W)

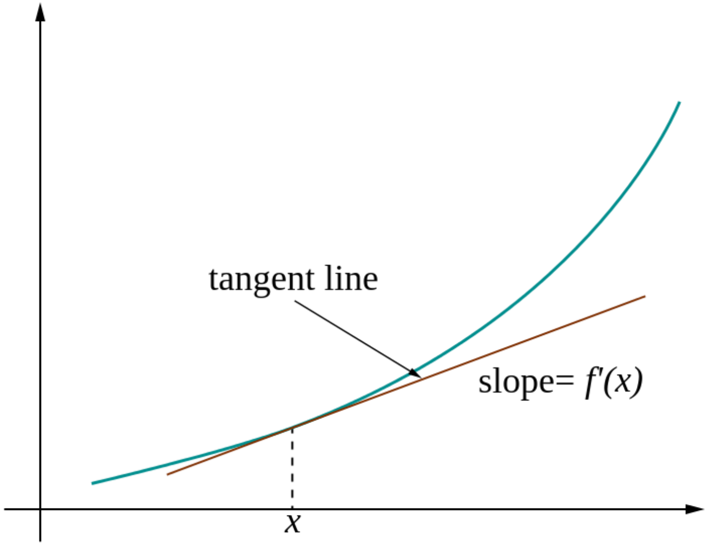
gradient dW:

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

Andrej Karpathy

Loss gradients

- Denoted as (diff notations): $\frac{\partial E}{\partial w_{ji}^{(1)}}$ $\nabla_W L$
- 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:

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

Time

Learning rate

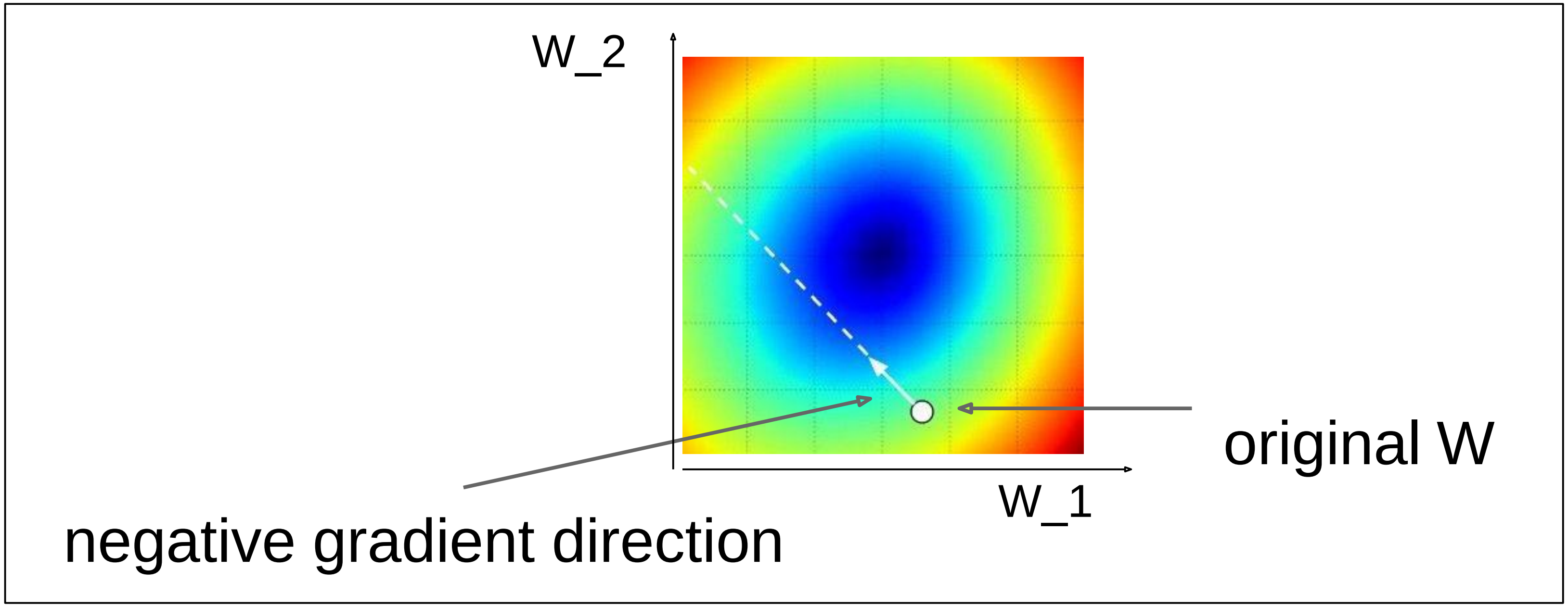


Figure from Andrej Karpathy

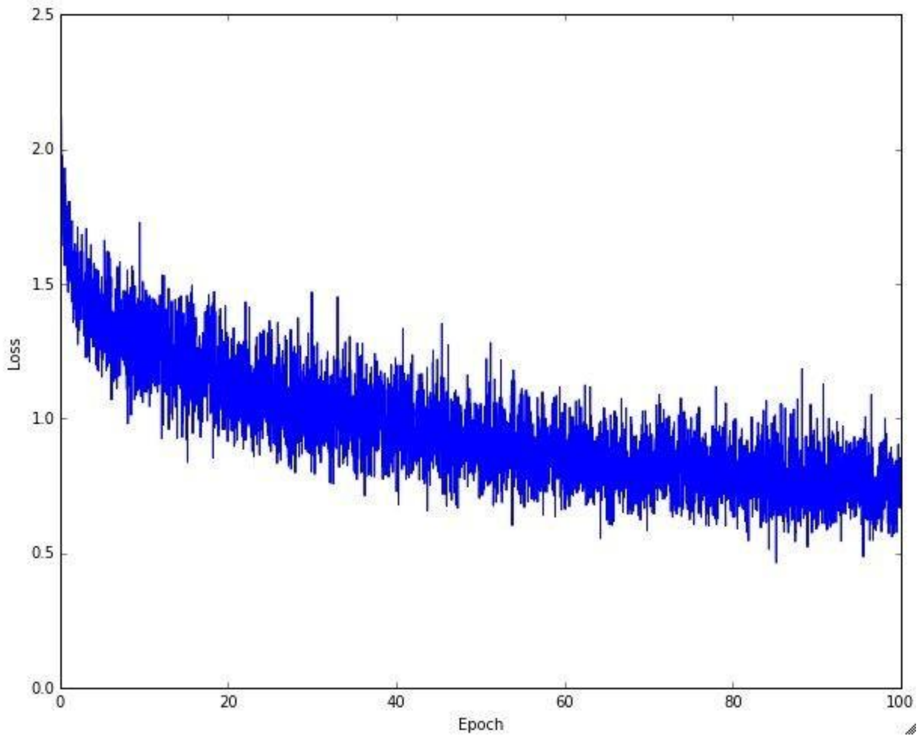
Gradient descent

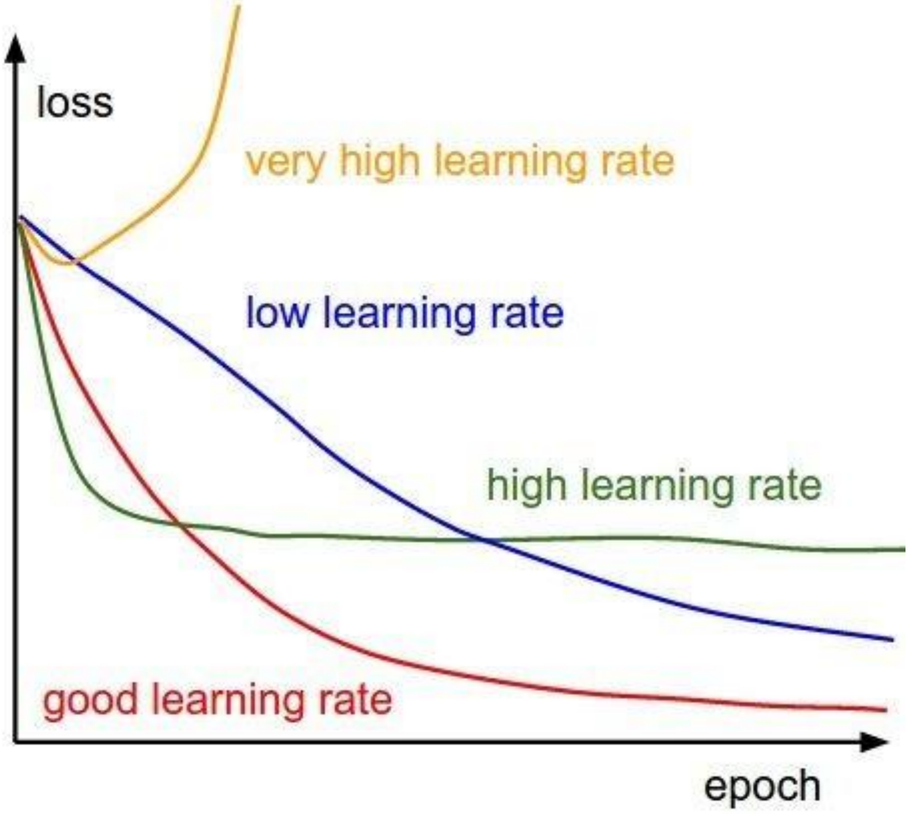
- Iteratively *subtract* the gradient with respect to the model parameters (w)
- I.e. we’re moving in a direction opposite to the gradient of the loss
- I.e. we’re moving towards *smaller* loss

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”)

Andrej Karpathy

Gradient descent in multi-layer nets

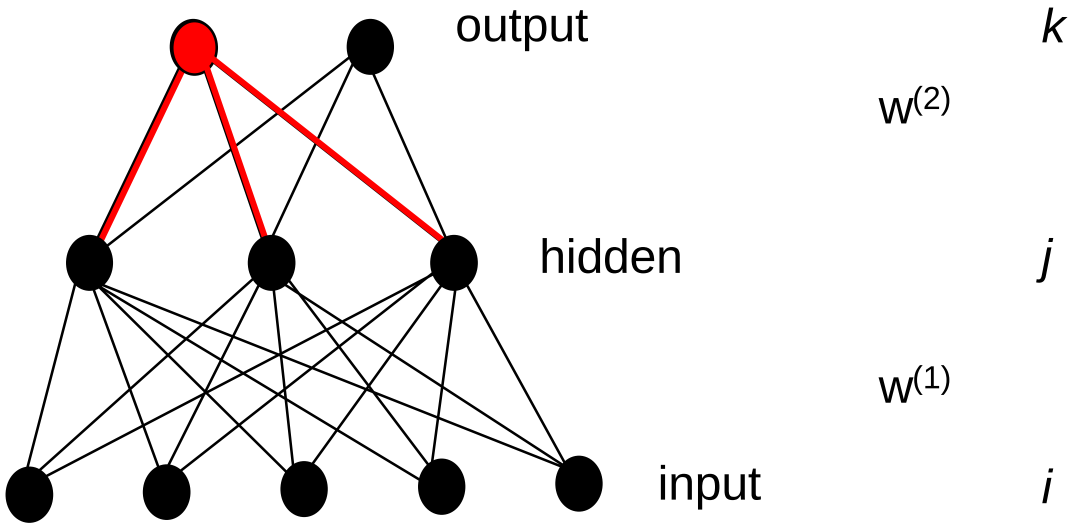
- We’ll update weights
- Move in direction opposite to gradient:

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

- 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.

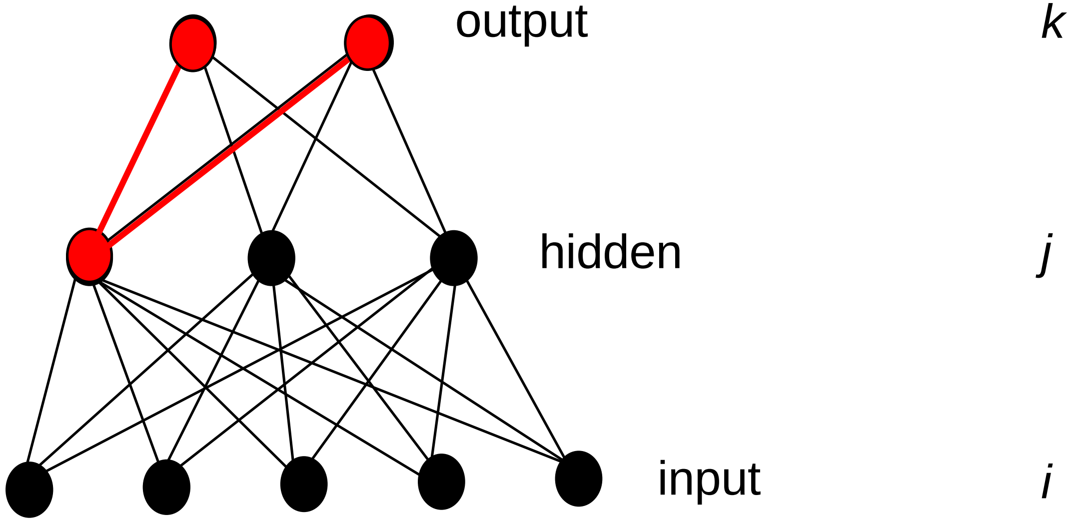


Update weights into *j*

Adapted from Ray Mooney, equations from Chris Bishop

Backpropagation: Graphic example

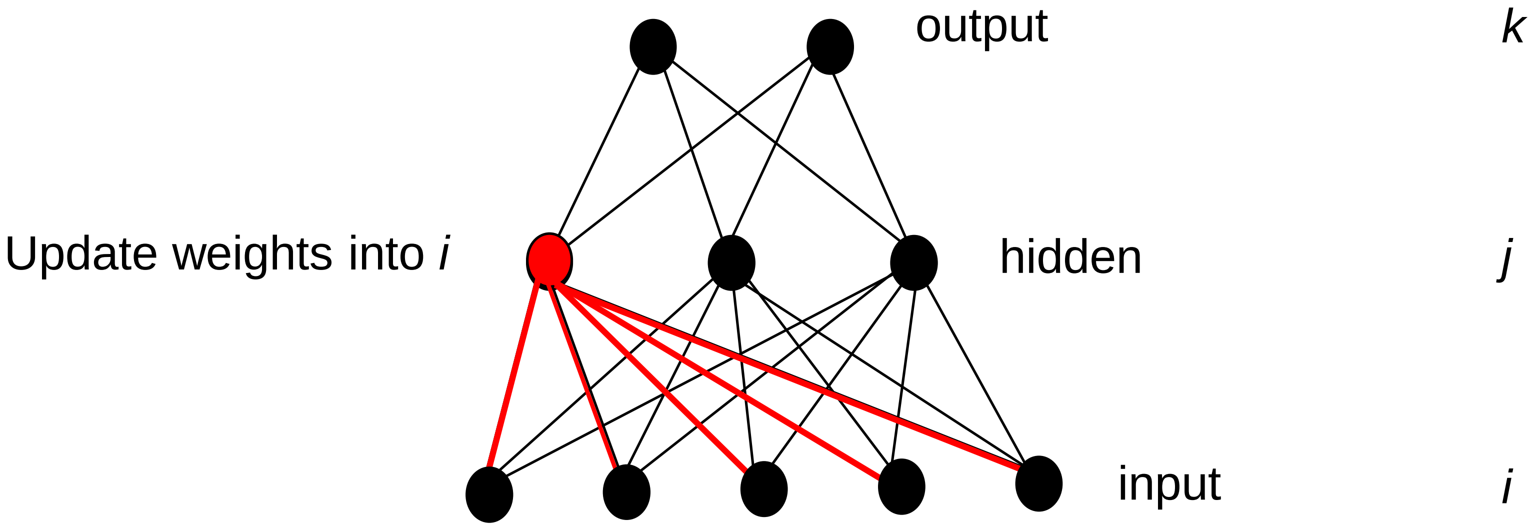
Next calculate error for hidden units based on errors on the output units it feeds into.



Adapted from Ray Mooney, equations from Chris Bishop

Backpropagation: Graphic example

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



Adapted from Ray Mooney, equations from Chris Bishop

Computing gradient for each weight

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

  $w_{ji} = w_{ji} - \eta\ dE/dw_{ji}$

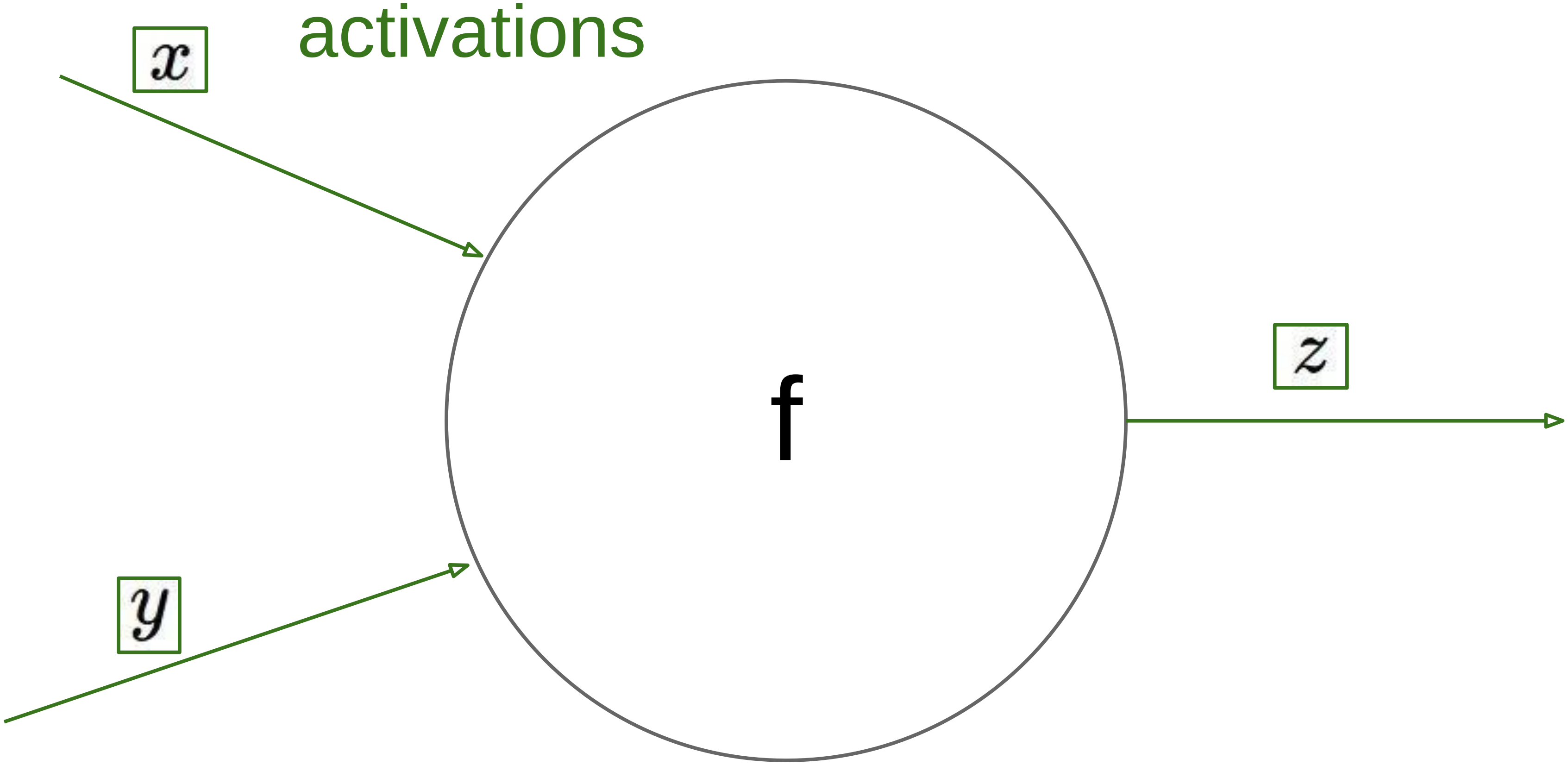
  $w_{kj} = w_{kj} - \eta\ dE/dw_{kj}$

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

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

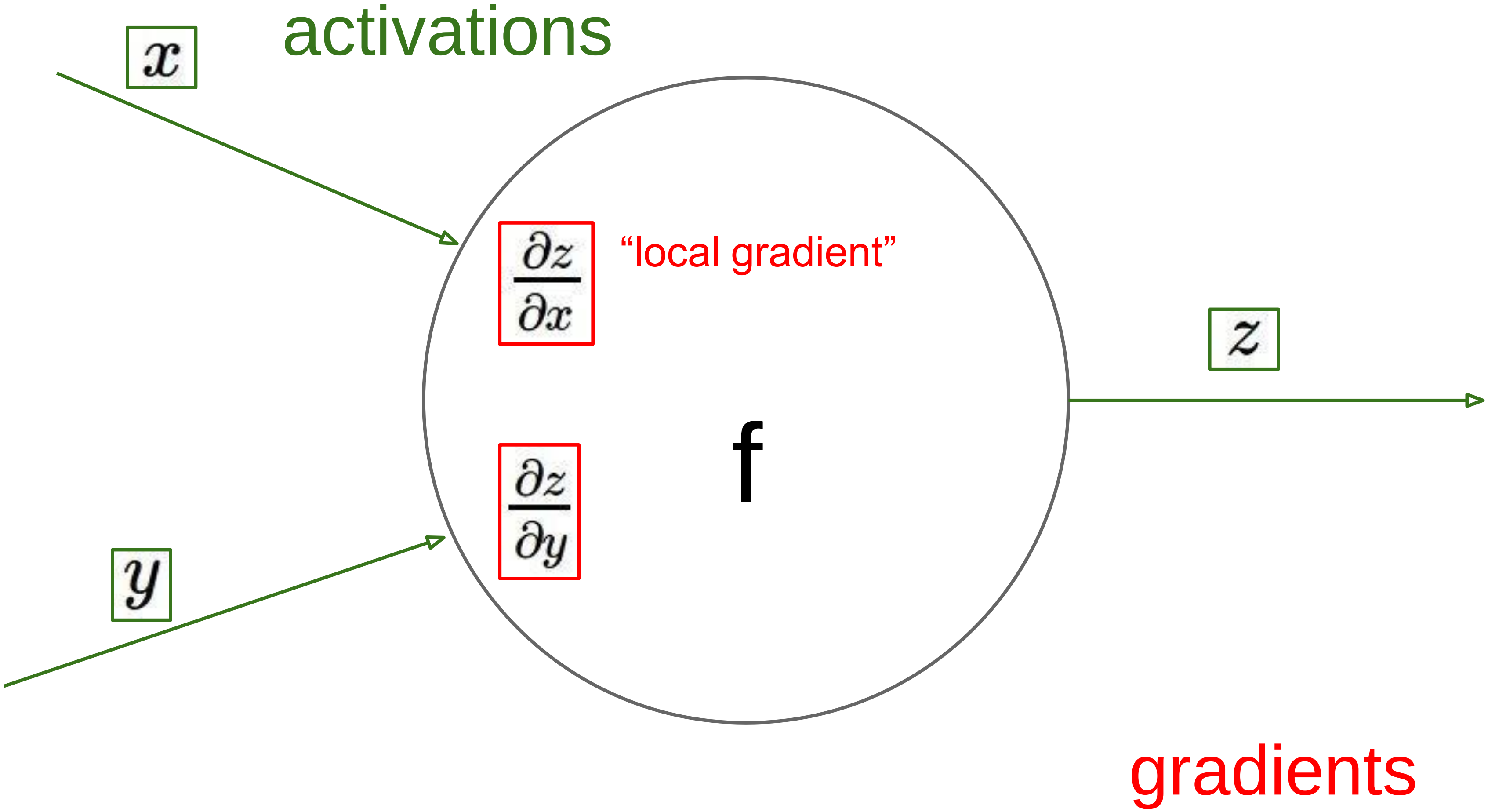
  (and similarly for dE/dw_{kj})

- 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



activations

Andrej Karpathy



activations

x

y

$\frac{\partial z}{\partial x}$ "local gradient"

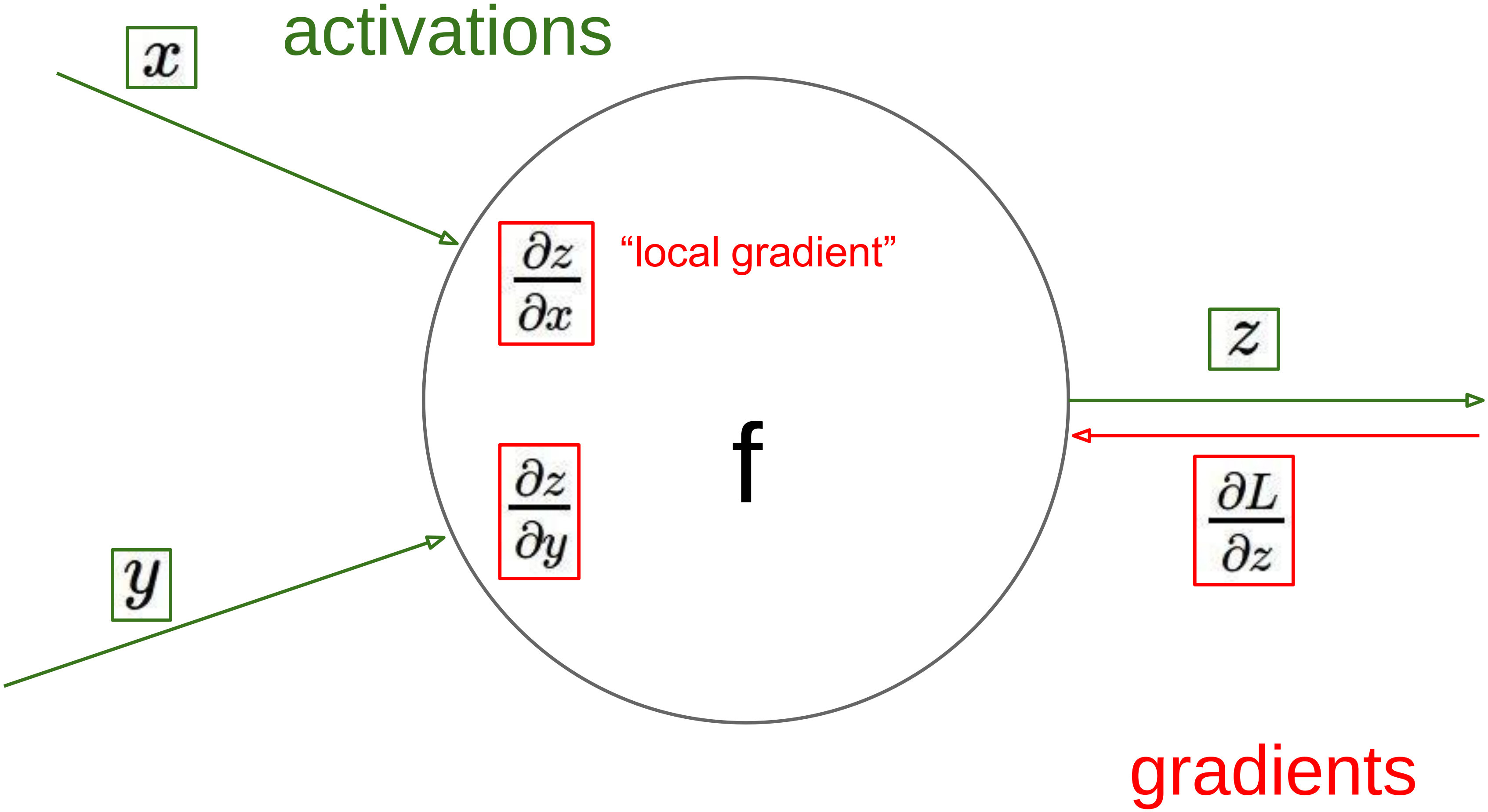
$\frac{\partial z}{\partial y}$

f

z

gradients

Andrej Karpathy



activations

x

$\frac{\partial z}{\partial x}$

"local gradient"

z

f

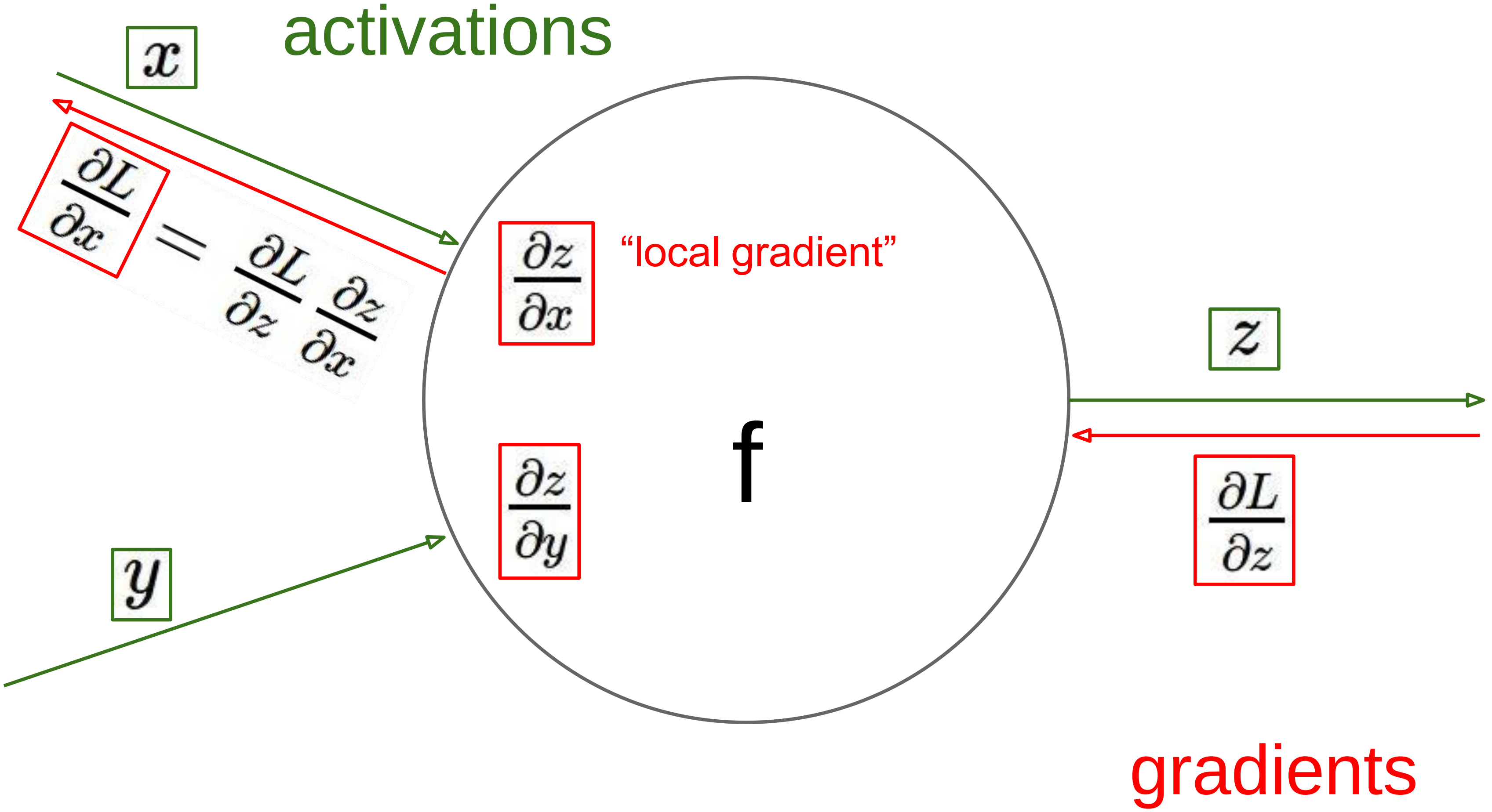
$\frac{\partial L}{\partial z}$

$\frac{\partial z}{\partial y}$

y

gradients

Andrej Karpathy



activations

x

y

$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$

$\frac{\partial z}{\partial x}$ "local gradient"

$\frac{\partial z}{\partial y}$

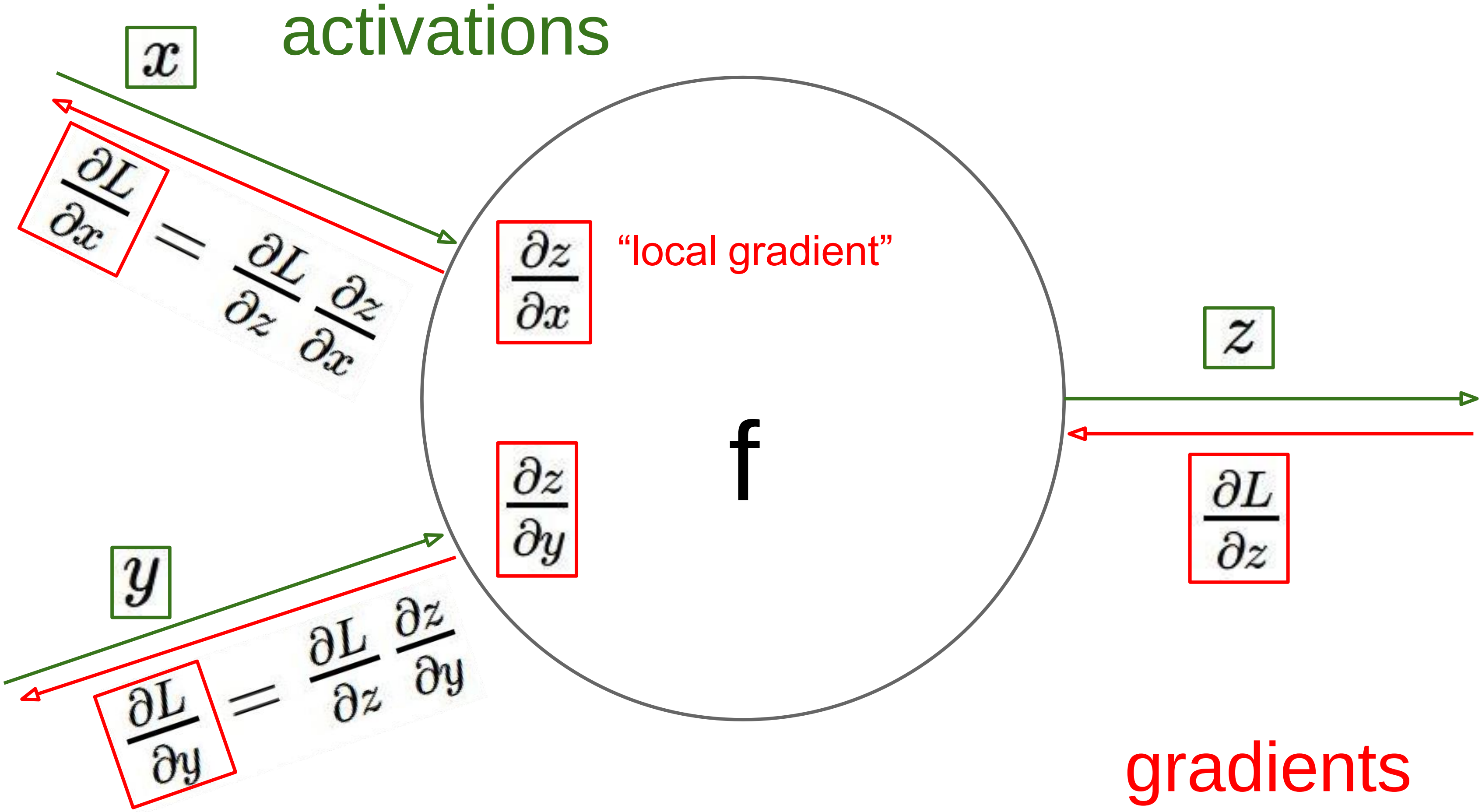
f

z

$\frac{\partial L}{\partial z}$

gradients

Andrej Karpathy



Andrej Karpathy

Example: algorithm for sigmoid, sqerror

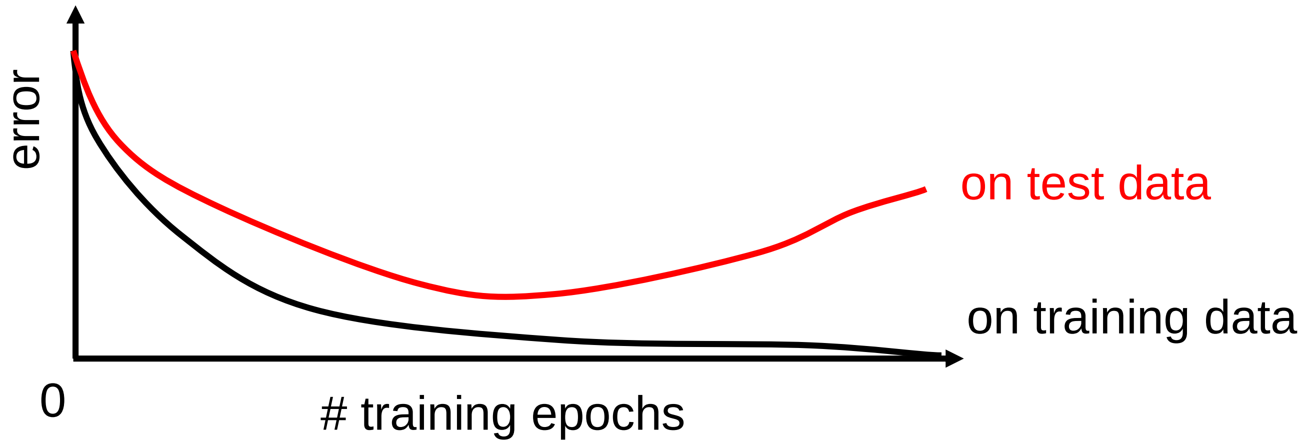
- Initialize all weights to small random values
- Until convergence (e.g. all training examples' error small, or error stops decreasing) repeat:
  - For each `(`**`x`**`, `**`t`**`=class(`**`x`**`))` in training set:
    - – Calculate network outputs: y_k
    - – Compute errors (gradients wrt activations) for each unit:
      - » $\delta_k = y_k\ (1-y_k)\ (y_k - t_k)$ 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: $w_{ji} = w_{ji} - \eta\ dE/dz_j\ dz_j/da_j\ da_j/dw_{ji}$

Adapted from R. Hwa, R. Mooney

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.

Adapted from Ray Mooney

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, with good choice of hyperparameters (e.g. learning rate).
- Thousands of epochs (epoch = network sees all training data once) may be required, hours or days to train.
- To avoid local-minima problems, run several trials starting with different random weights (*random restarts*), and take results of trial with lowest training set error.
- May be hard to set learning rate and to select number of hidden units and layers.
- Neural networks had fallen out of fashion in 90s, early 2000s; back with a new name and 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

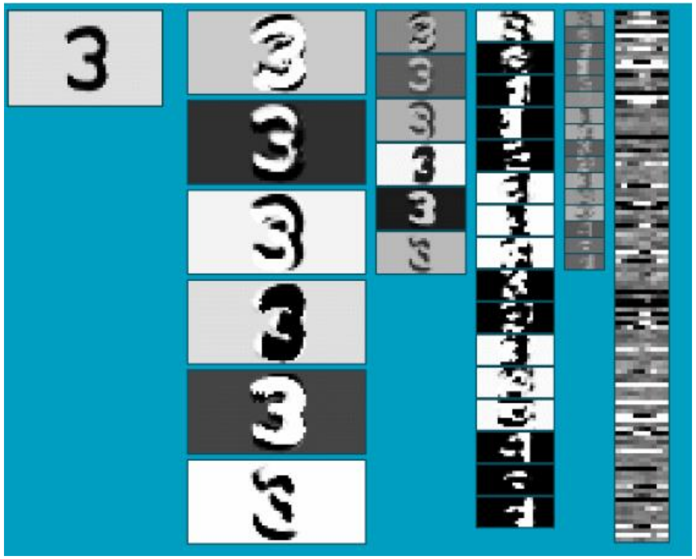
Understanding CNNs

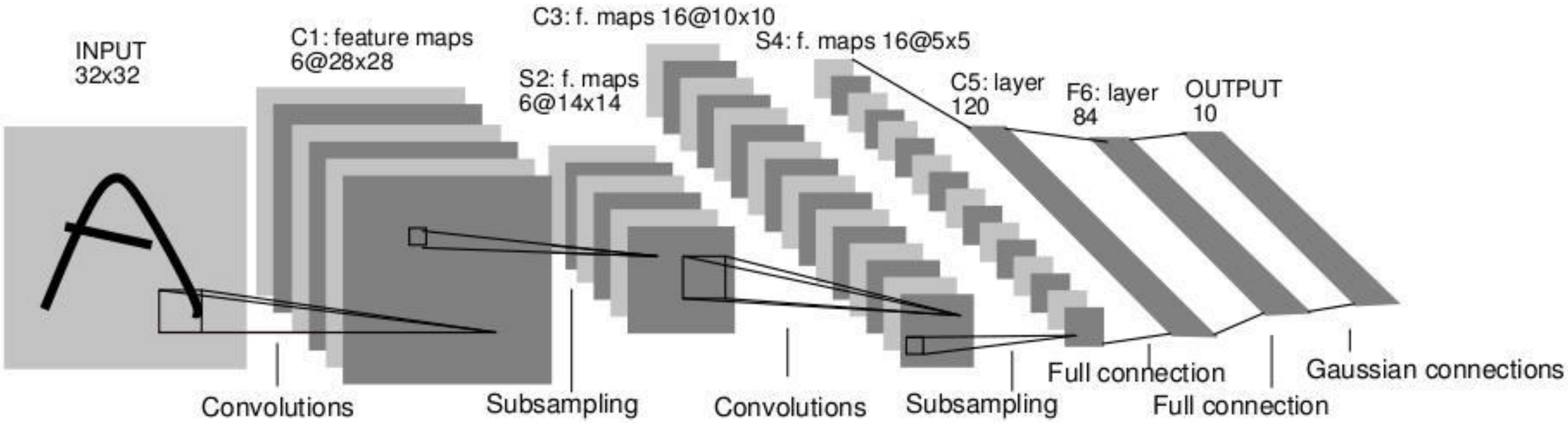
- Visualization
- Breaking CNNs

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



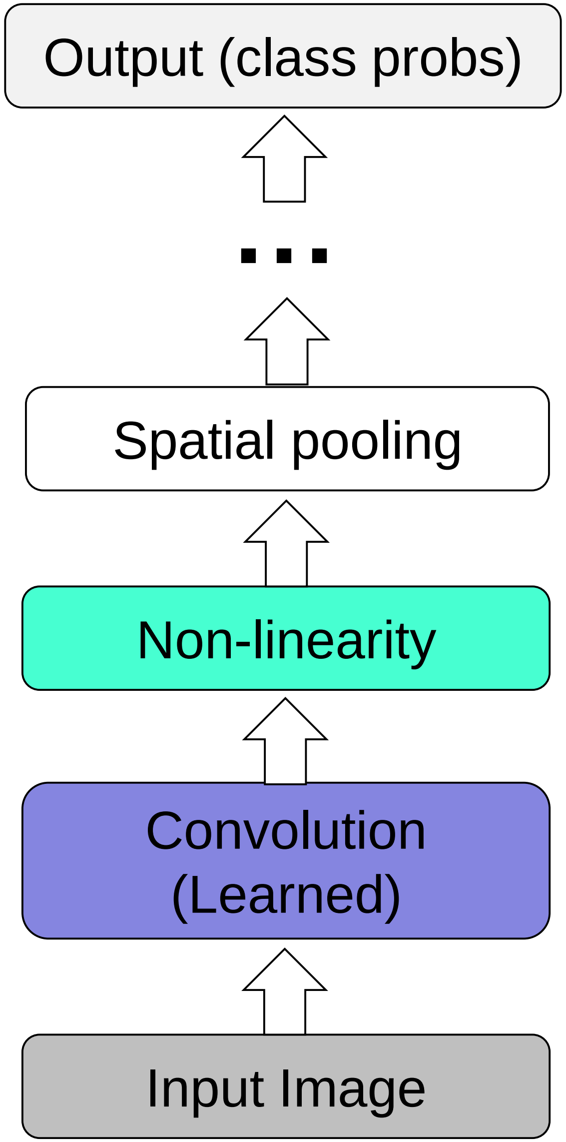


Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86(11): 2278–2324, 1998.

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



Output (class probs)

…

Spatial pooling

Non-linearity

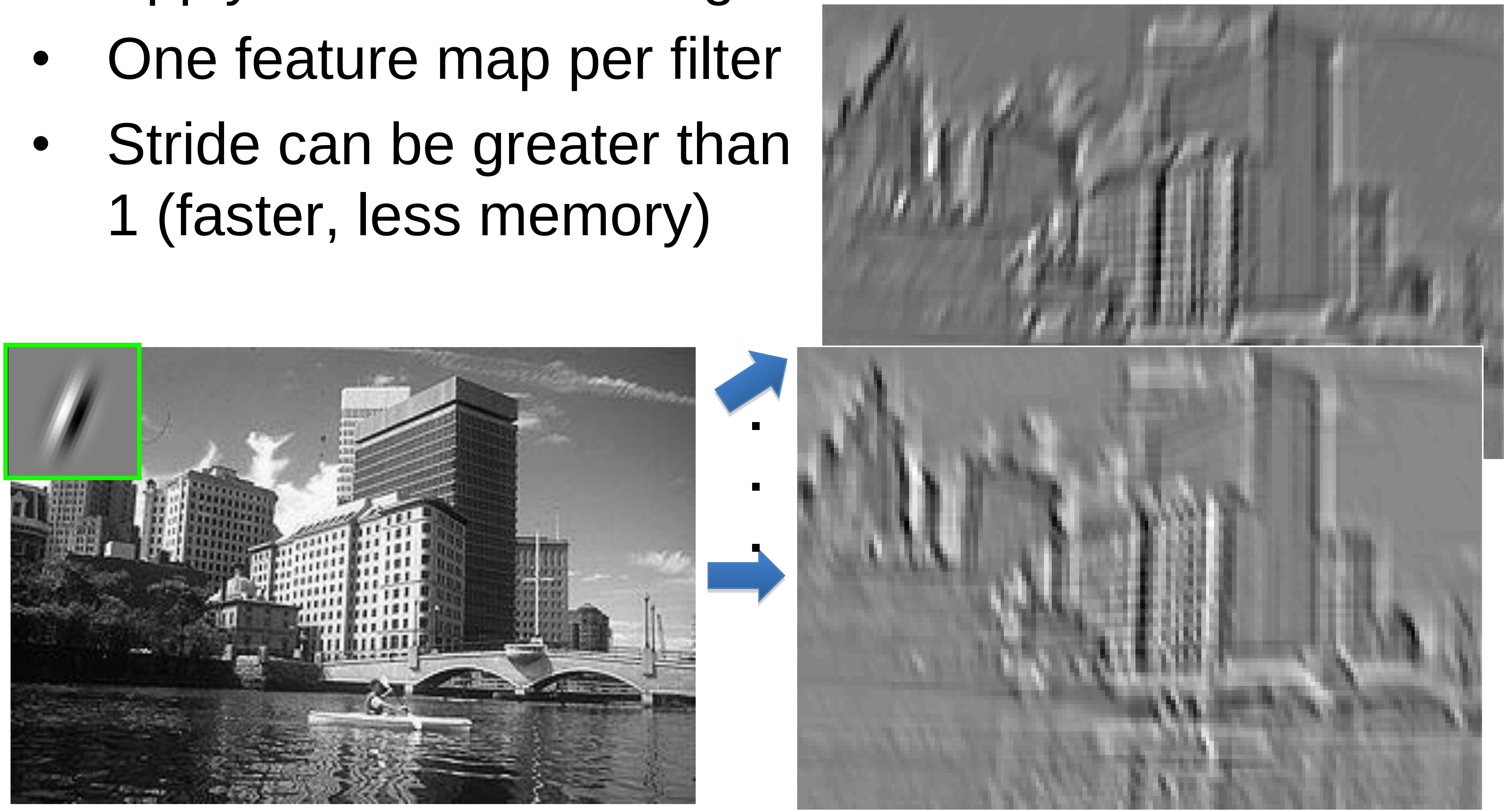
Convolution (Learned)

Input Image

Adapted from Lana Lazebnik

1. Convolution

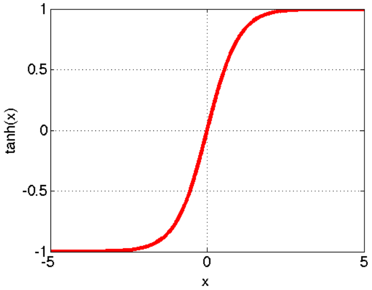
- Apply learned filter weights
- One feature map per filter
- Stride can be greater than 1 (faster, less memory)

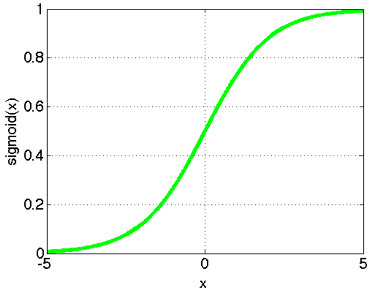


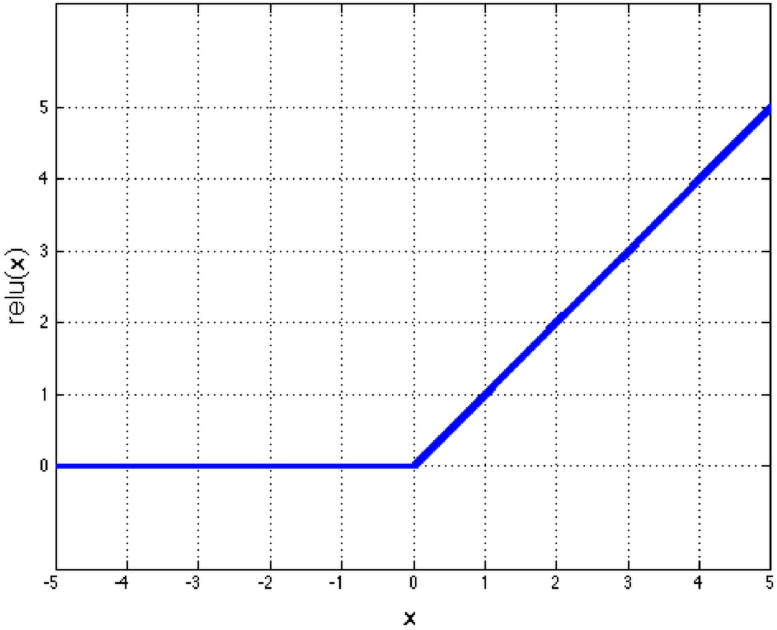
Adapted from Rob Fergus

2. Non-Linearity

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



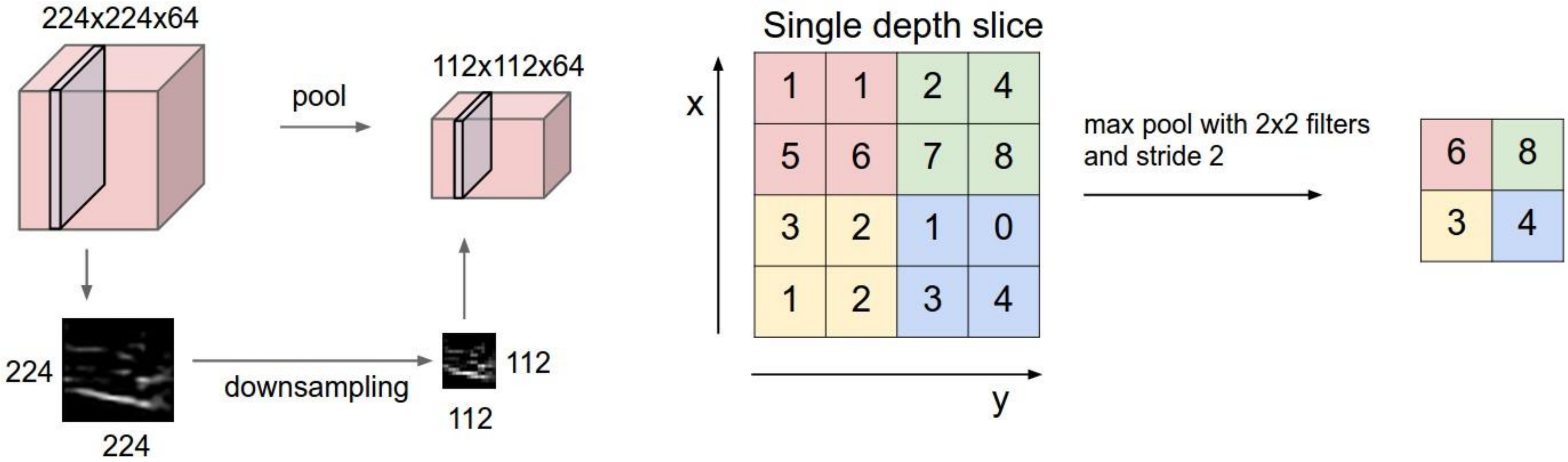




Adapted from Rob Fergus

3. Spatial Pooling

- Sum or max over non-overlapping / overlapping regions



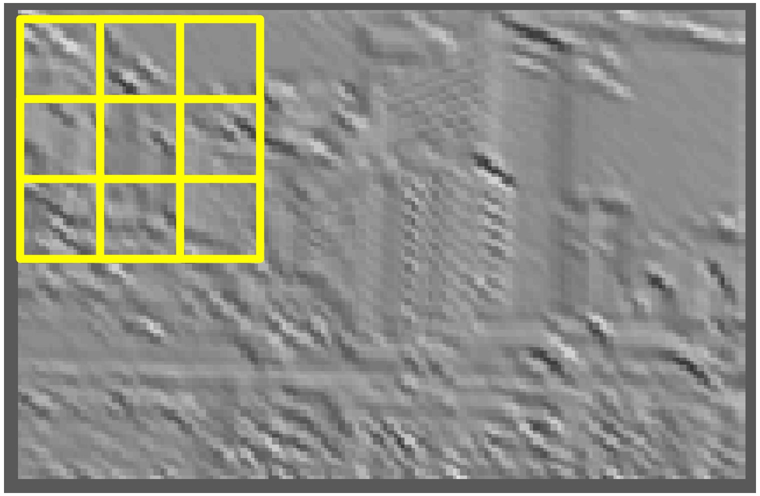
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

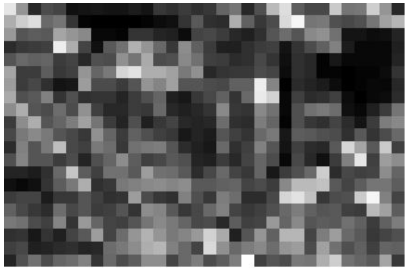
Rob Fergus, figure from Andrej Karpathy

3. Spatial Pooling

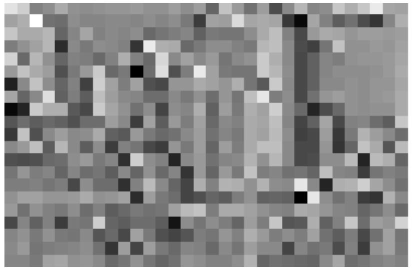
- Sum or max over non-overlapping / overlapping regions
- Role of pooling:
    - Invariance to small transformations
    - Larger receptive fields (neurons see more of input)



Max



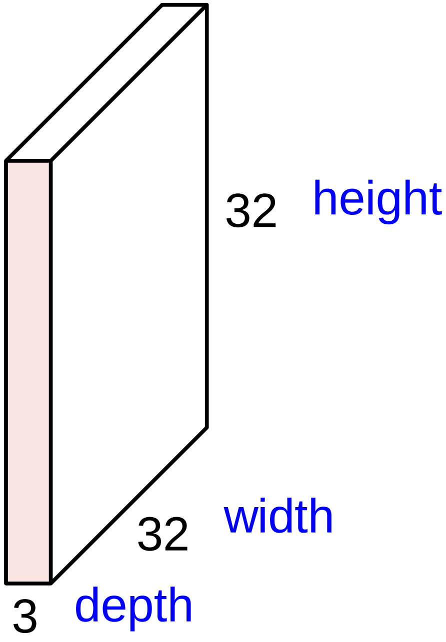
Sum



Adapted from Rob Fergus

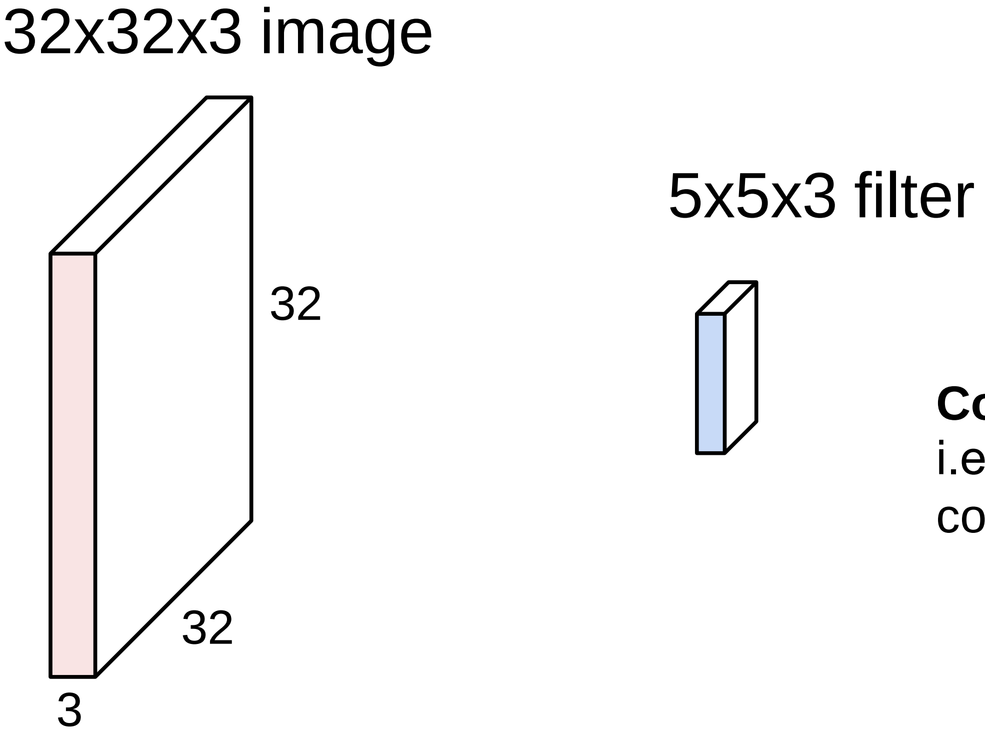
Convolutions: More detail

32x32x3 image



Andrej Karpathy

Convolutions: More detail

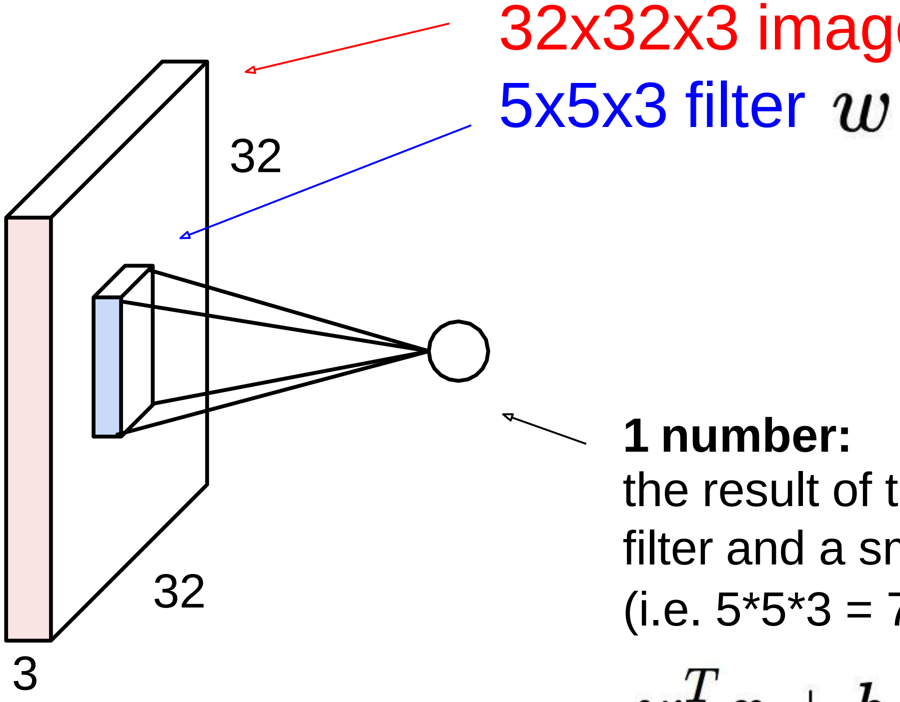


Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”

Andrej Karpathy

Convolutions: More detail

Convolution Layer



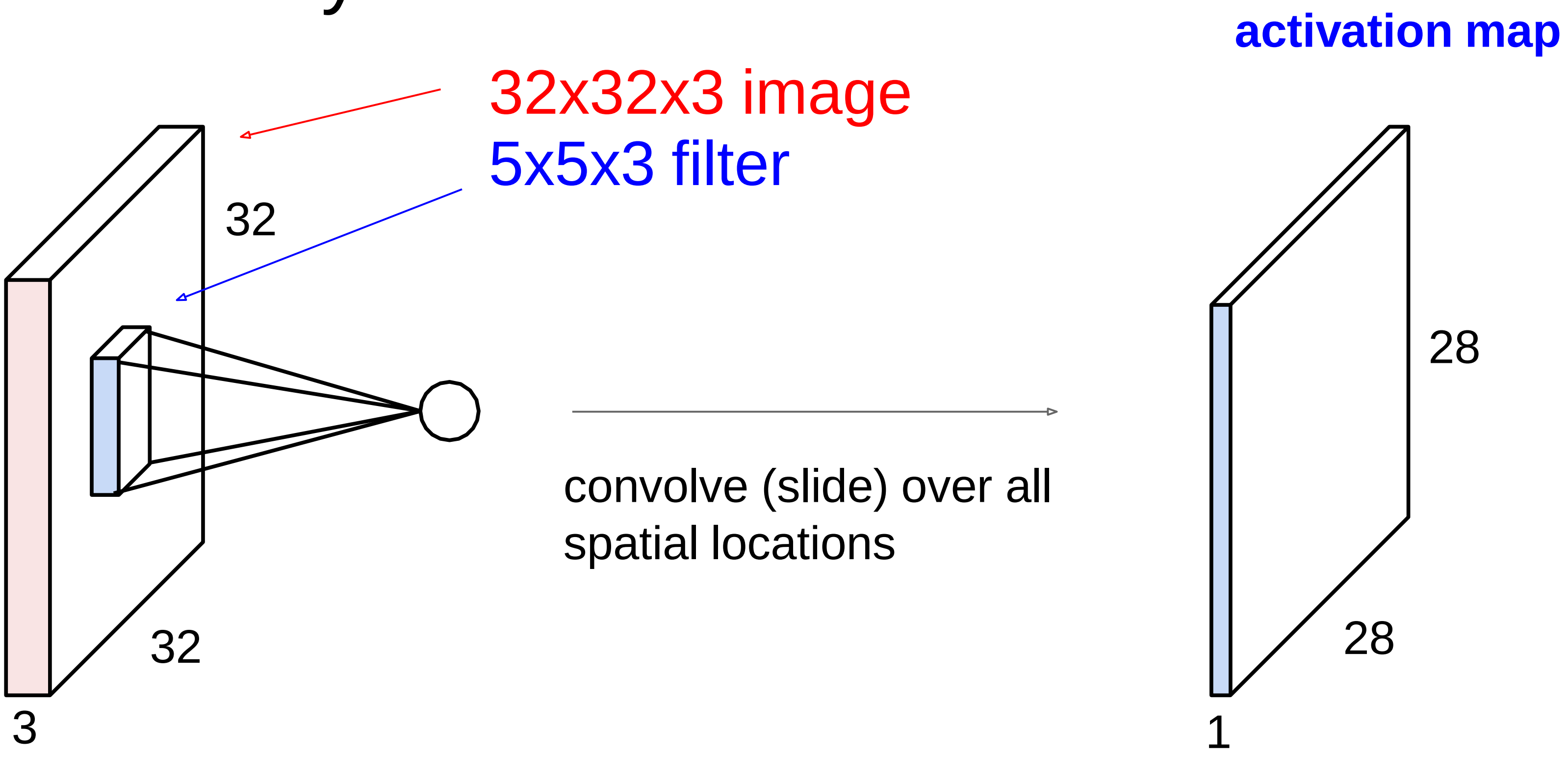
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Andrej Karpathy

Convolutions: More detail

Convolution Layer

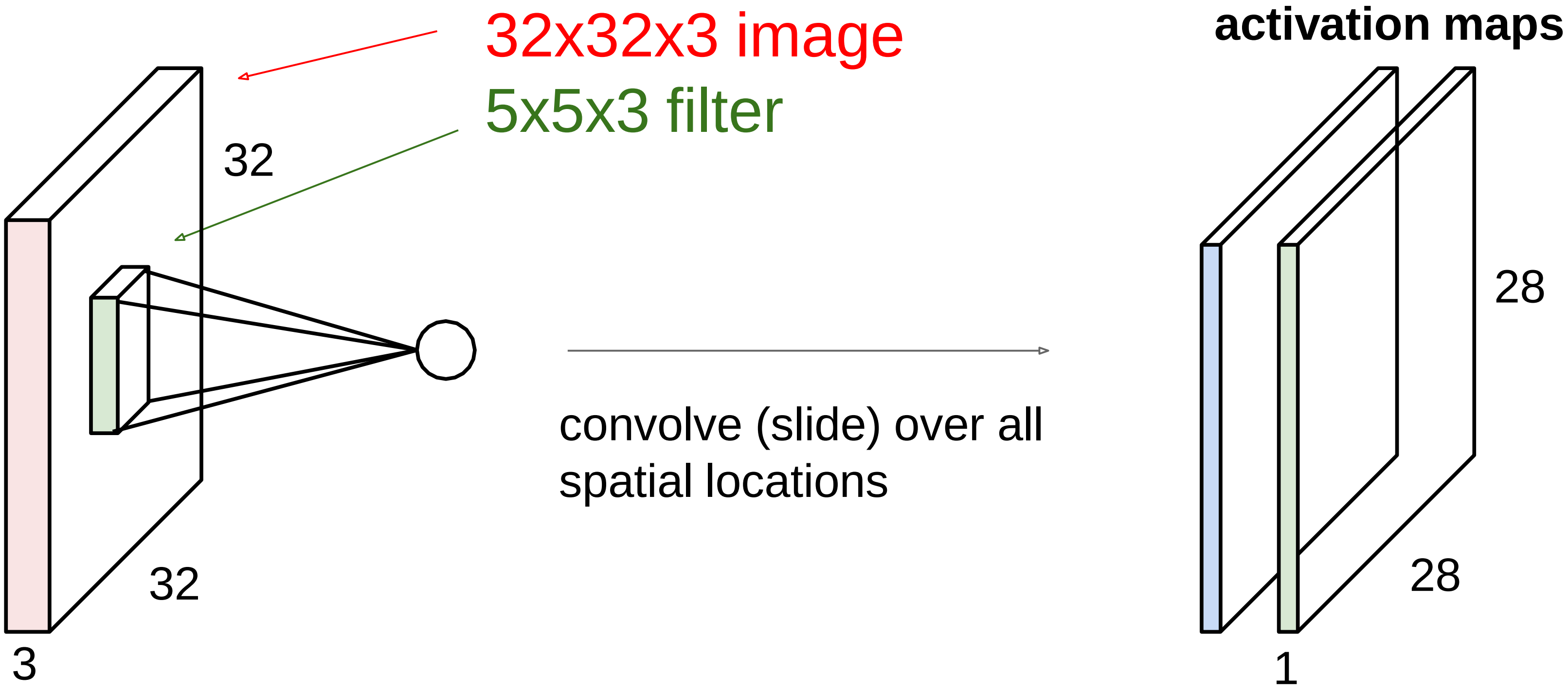


convolve (slide) over all spatial locations

Convolutions: More detail

Convolution Layer

consider a second, green filter



32x32x3 image

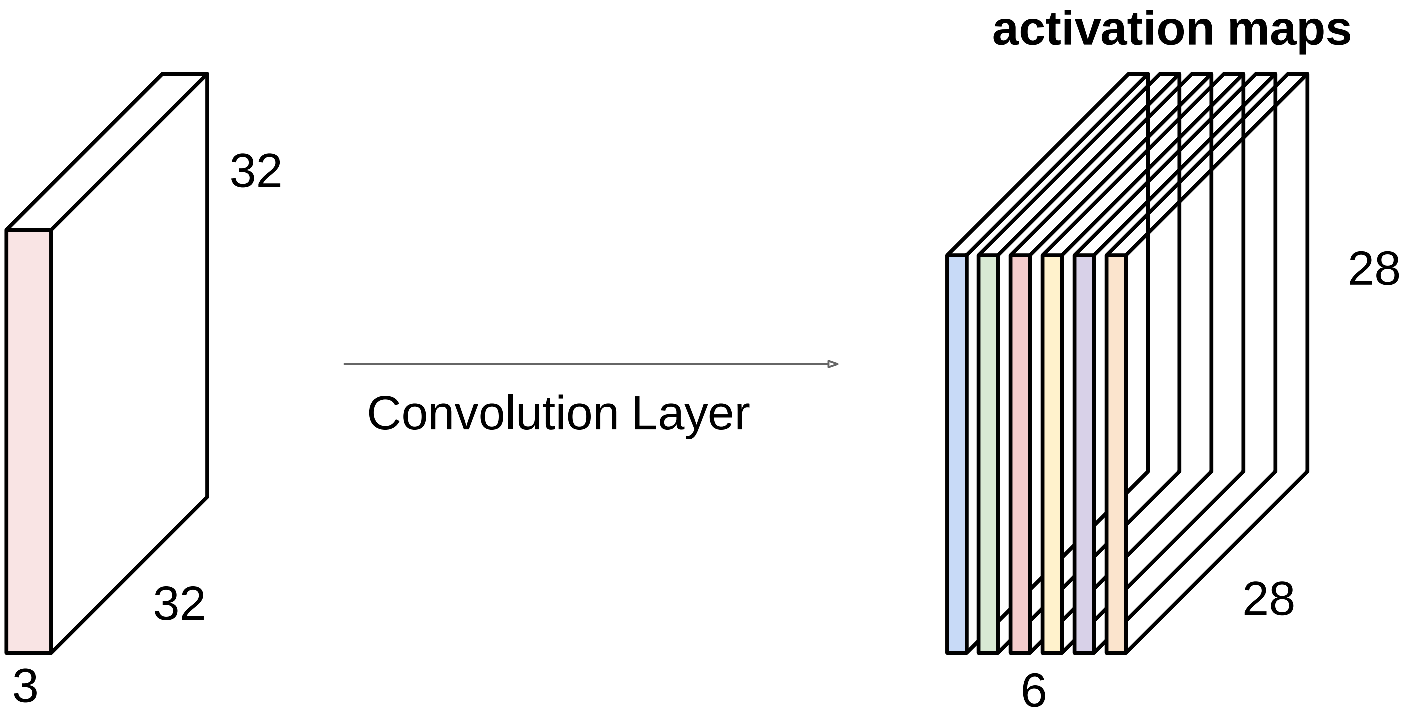
5x5x3 filter

convolve (slide) over all spatial locations

activation maps

Convolutions: More detail

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

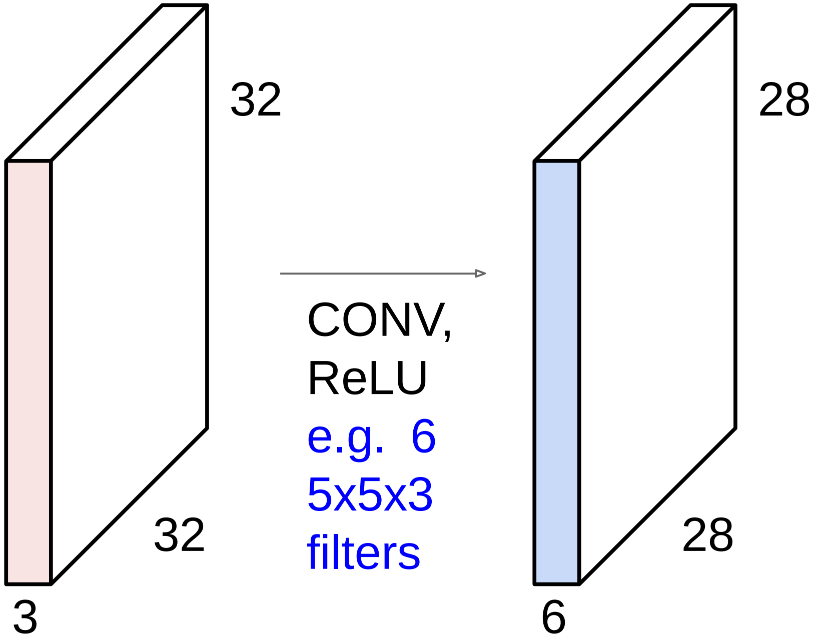


We stack these up to get a "new image" of size 28x28x6!

Andrej Karpathy

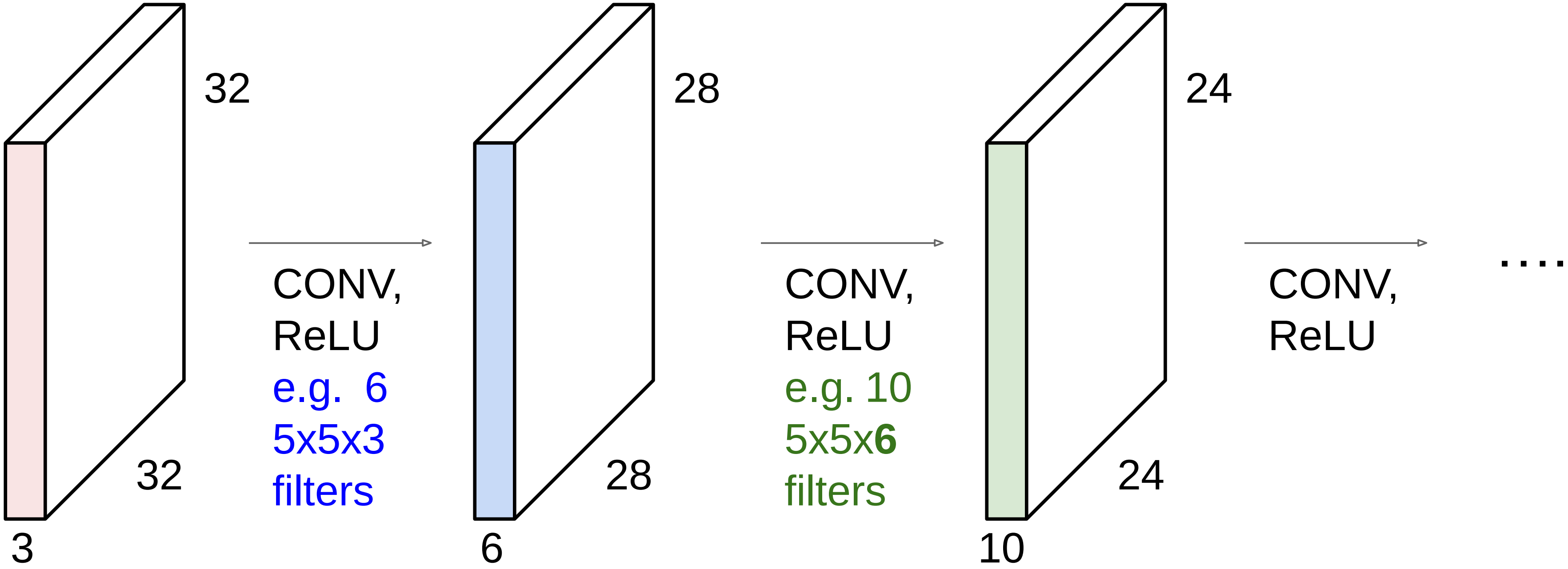
Convolutions: More detail

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



32

32

3

CONV,
ReLU
e.g. 6
5x5x3
filters

28

28

6

CONV,
ReLU
e.g. 10
5x5x**6**
filters

24

24

10

CONV,
ReLU

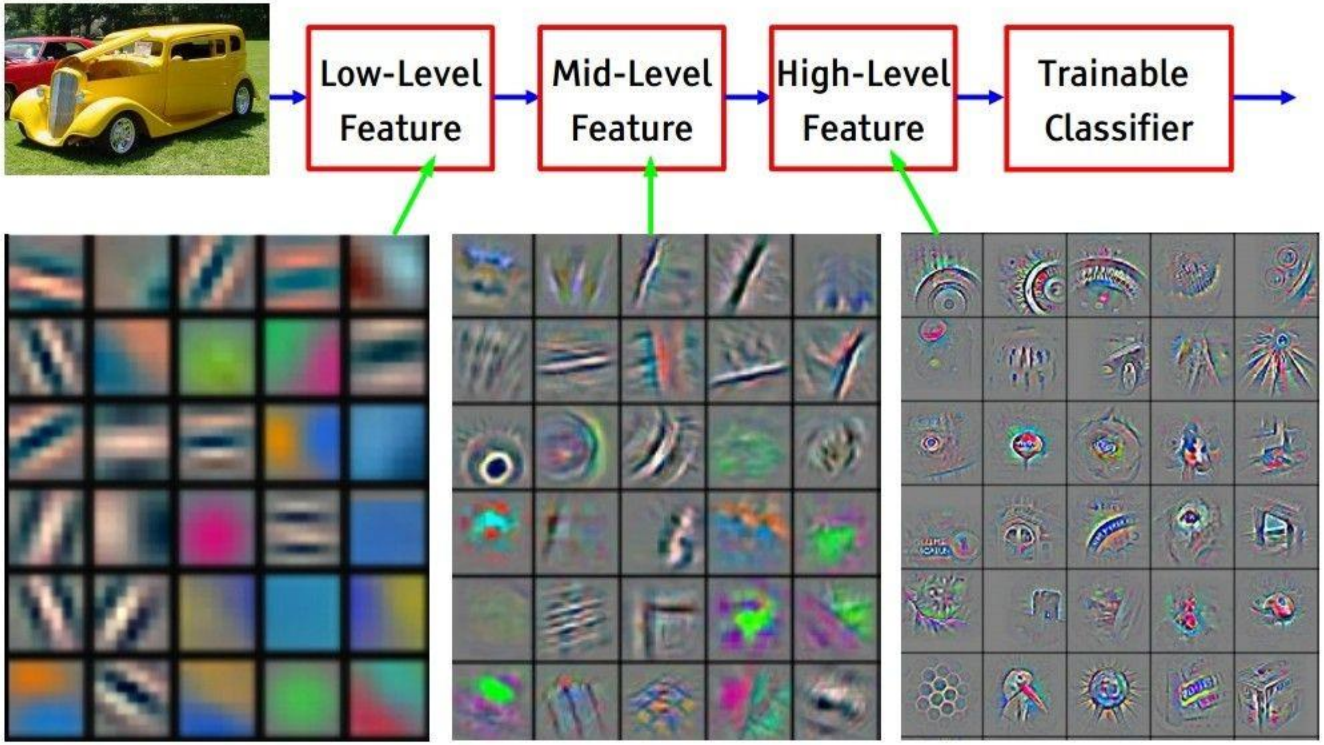
….

Andrej Karpathy

Convolutions: More detail

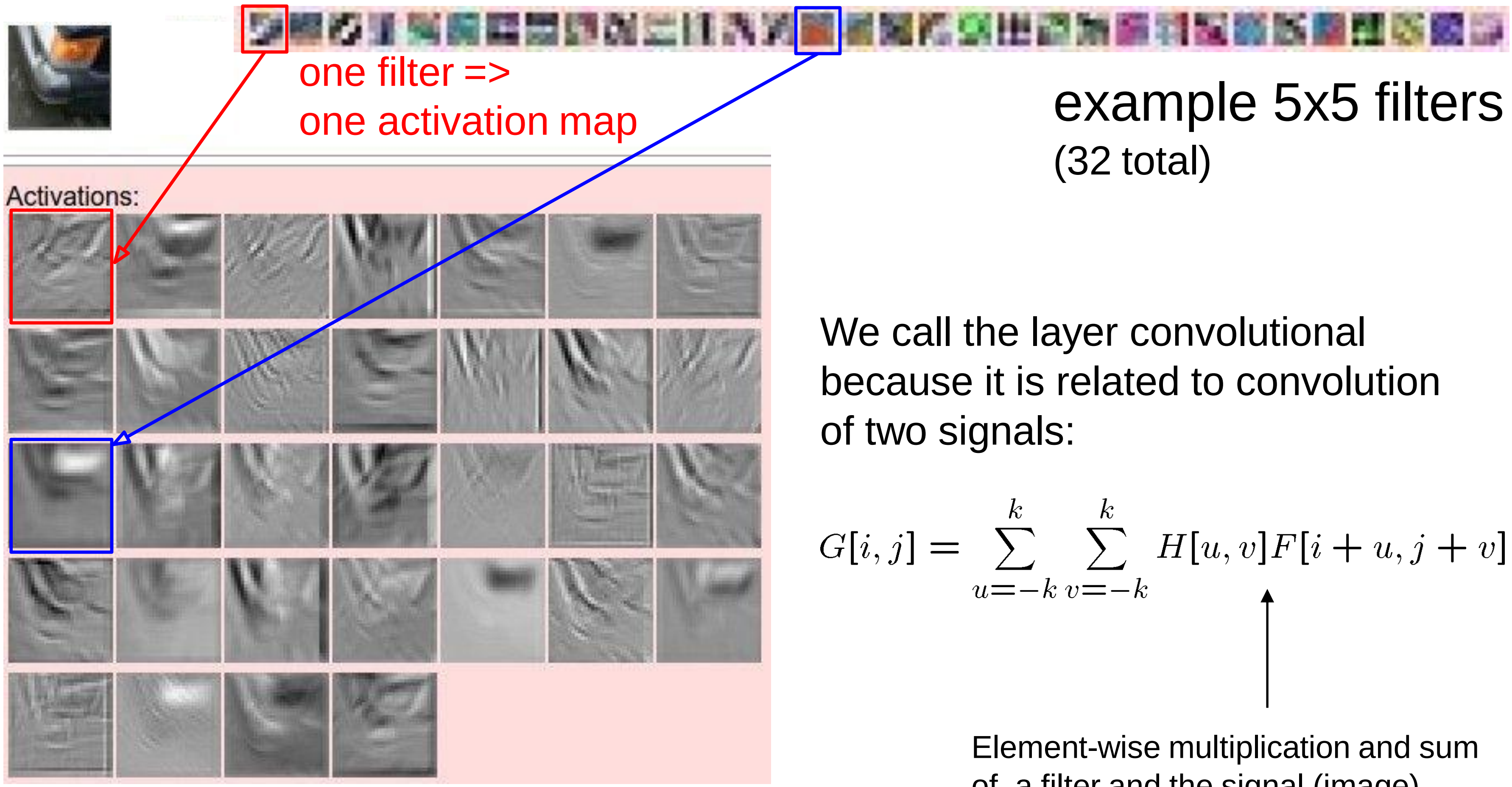
Preview

[From recent Yann LeCun slides]



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

Convolutions: More detail



one filter =>
one activation map

example 5x5 filters
(32 total)

We call the layer convolutional because it is related to convolution of two signals:

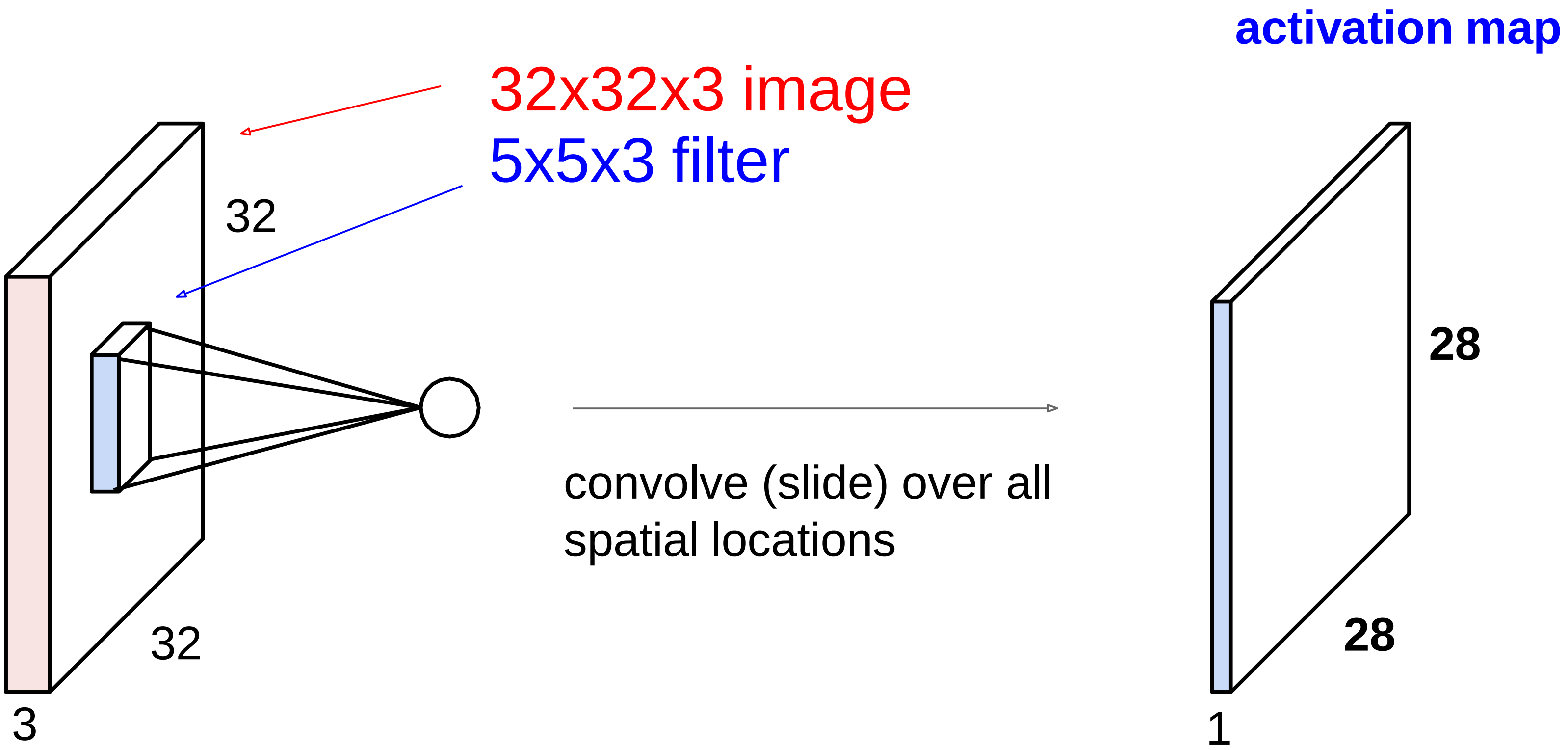
$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v] F[i+u, j+v]$$

Element-wise multiplication and sum of a filter and the signal (image)

Adapted from Andrej Karpathy, Kristen Grauman

Convolutions: More detail

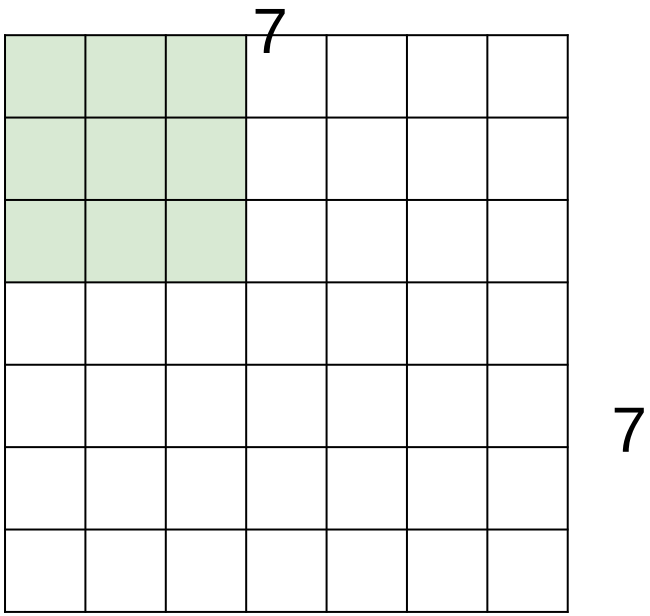
A closer look at spatial dimensions:



Andrej Karpathy

Convolutions: More detail

A closer look at spatial dimensions:

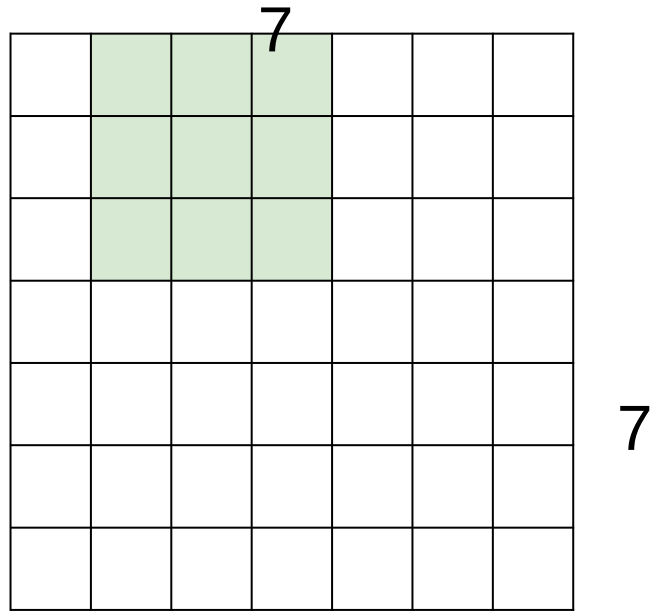


7x7 input (spatially)
assume 3x3 filter

Andrej Karpathy

Convolutions: More detail

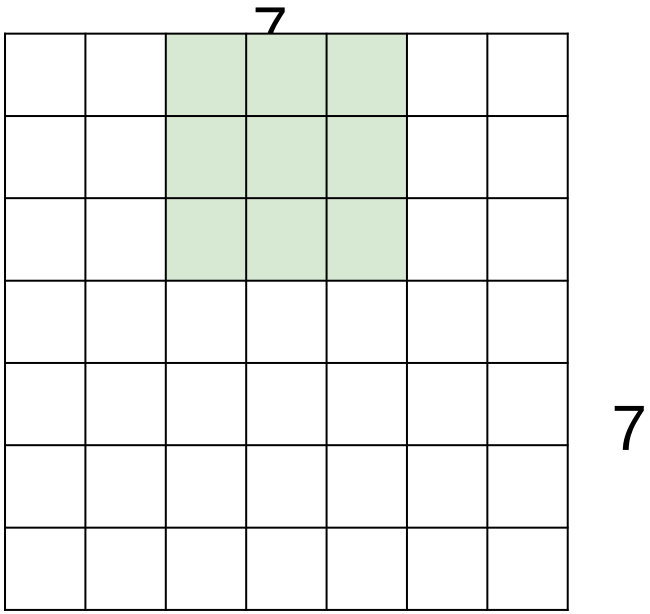
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

Convolutions: More detail

A closer look at spatial dimensions:

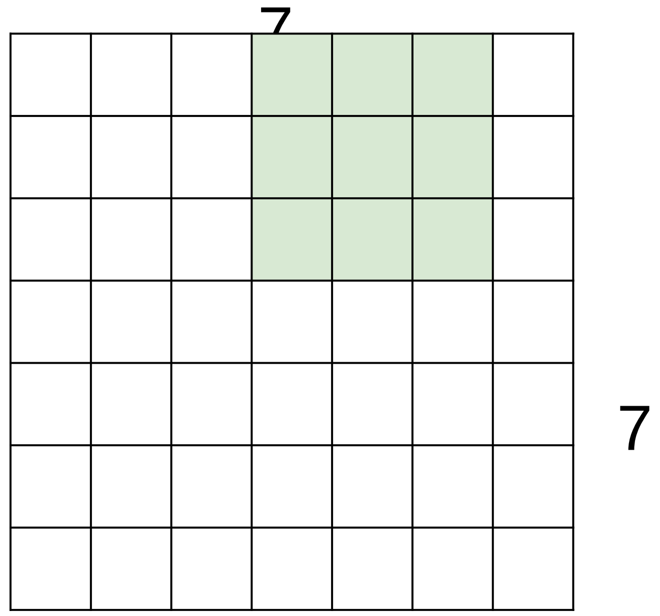


7x7 input (spatially)
assume 3x3 filter

Andrej Karpathy

Convolutions: More detail

A closer look at spatial dimensions:

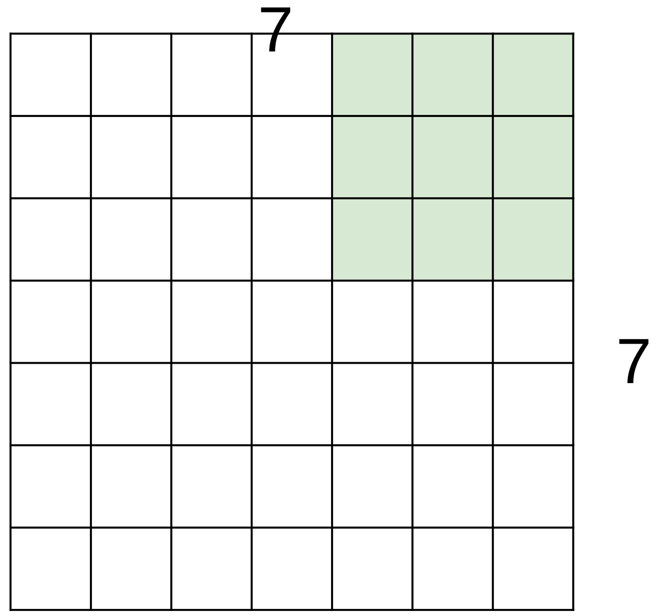


7x7 input (spatially)
assume 3x3 filter

Andrej Karpathy

Convolutions: More detail

A closer look at spatial dimensions:

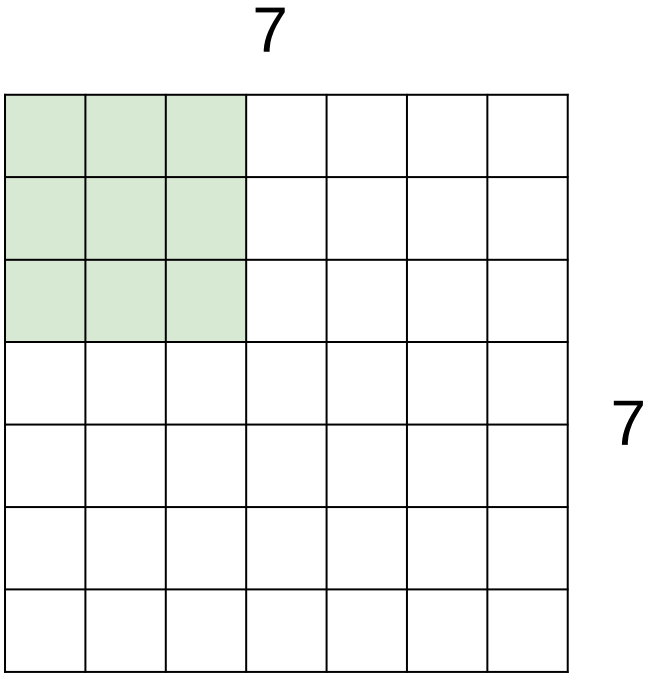


7x7 input (spatially)
assume 3x3 filter

=> 5x5 output

Convolutions: More detail

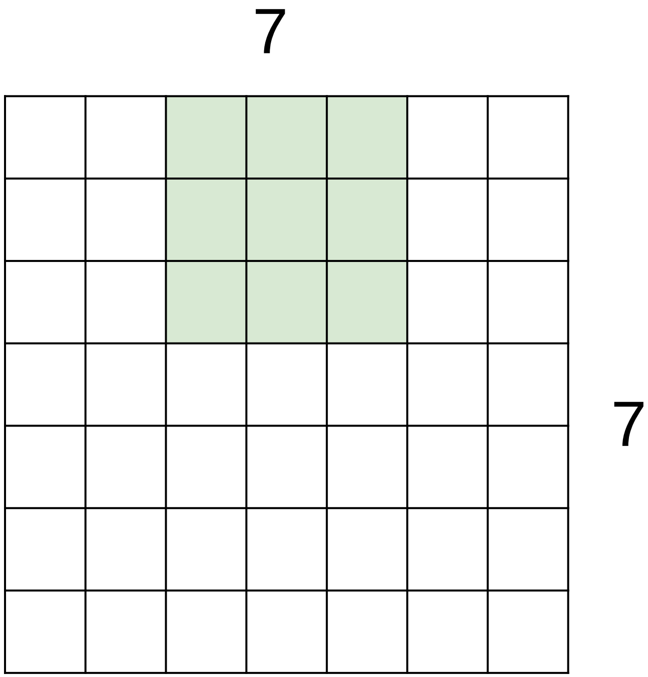
A closer look at spatial dimensions:



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

Convolutions: More detail

A closer look at spatial dimensions:

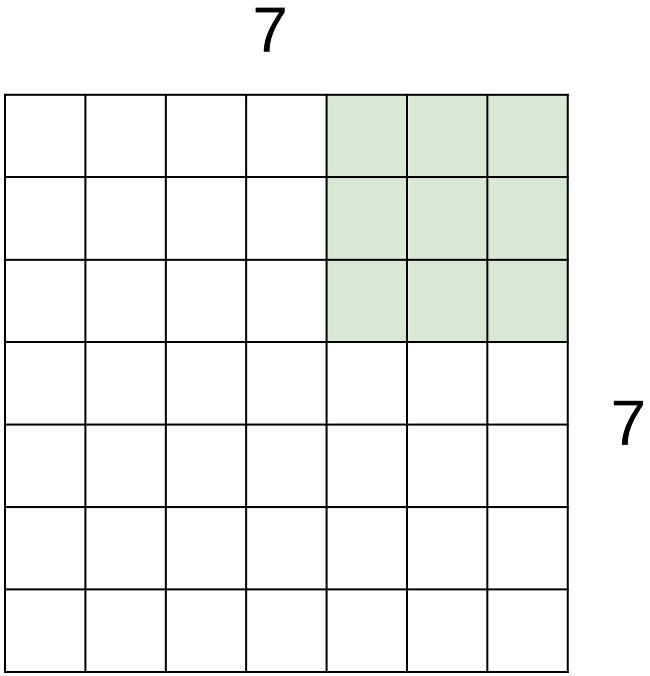


7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

Andrej Karpathy

Convolutions: More detail

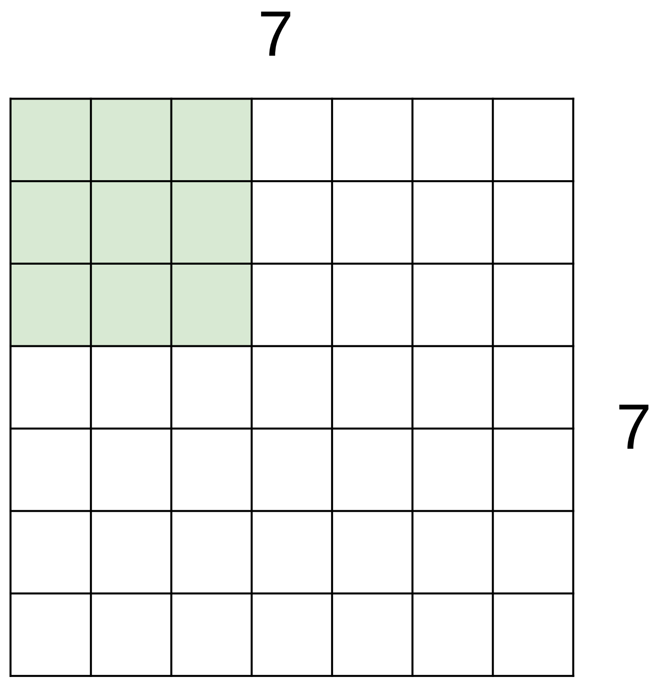
A closer look at spatial dimensions:



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

Convolutions: More detail

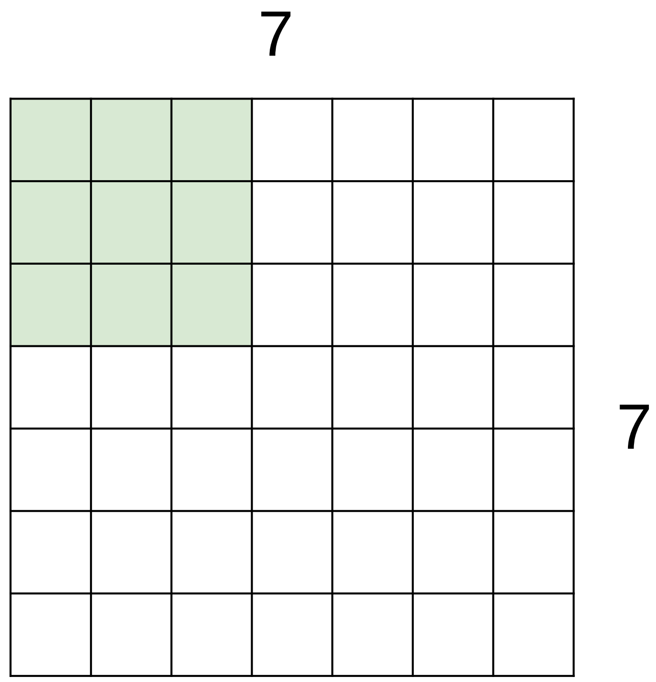
A closer look at spatial dimensions:



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

Convolutions: More detail

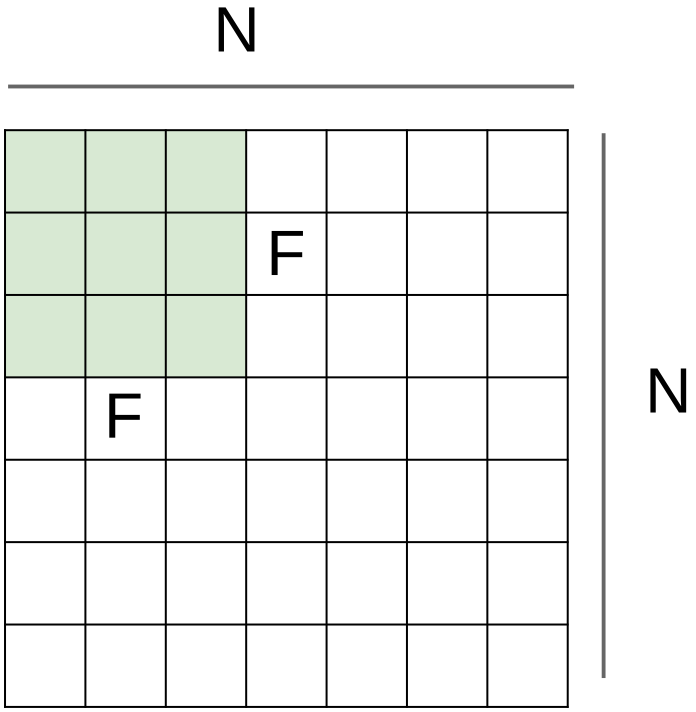
A closer look at spatial dimensions:



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

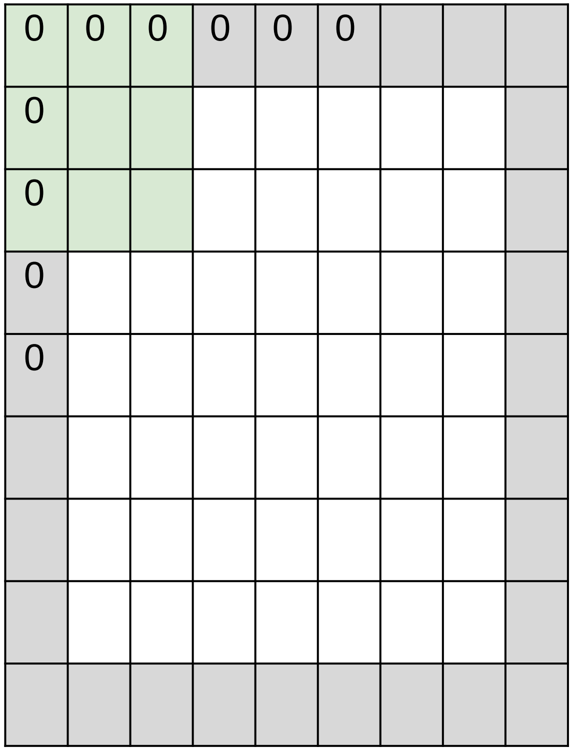


Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

Convolutions: More detail

In practice: Common to zero pad the border



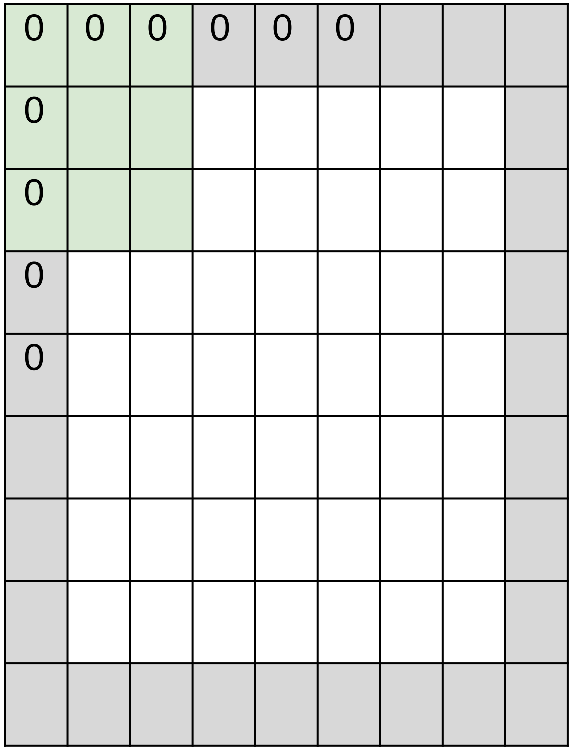
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

Andrej Karpathy

Convolutions: More detail

In practice: Common to zero pad the border



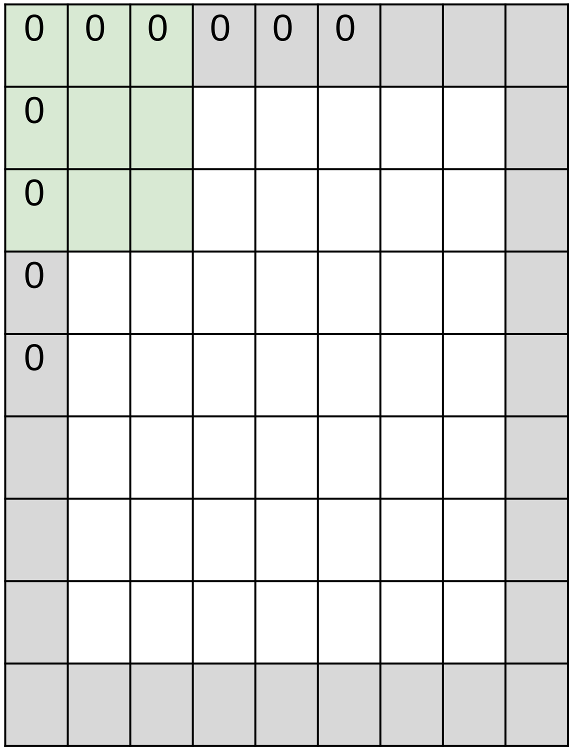
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

Andrej Karpathy

Convolutions: More detail

In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

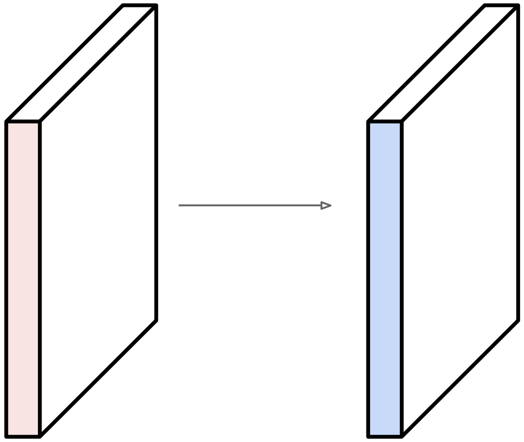
7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

(N + 2*padding - F) / stride + 1

Andrej Karpathy

Convolutions: More detail

Examples time:



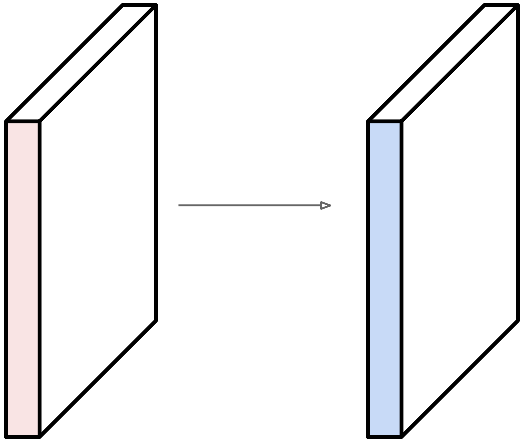
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?

Andrej Karpathy

Convolutions: More detail

Examples time:



Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Output volume size:

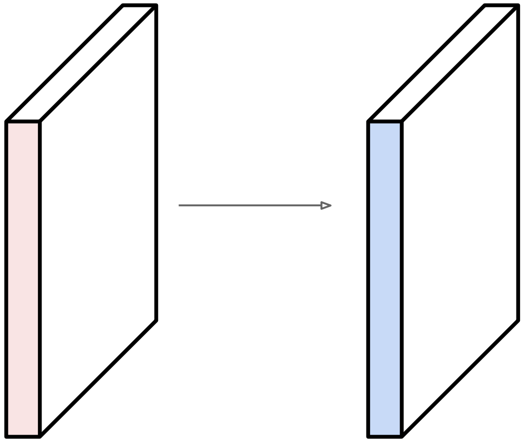
(32+2*2-5)/1+1 = 32 spatially, so

32x32x10

Andrej Karpathy

Convolutions: More detail

Examples time:

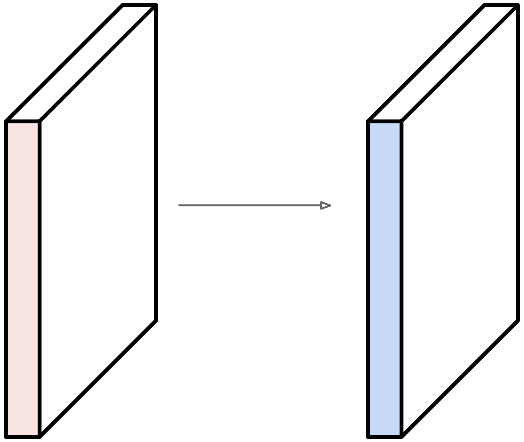


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

Convolutions: More detail

Examples time:



Input volume: **32x32x3**

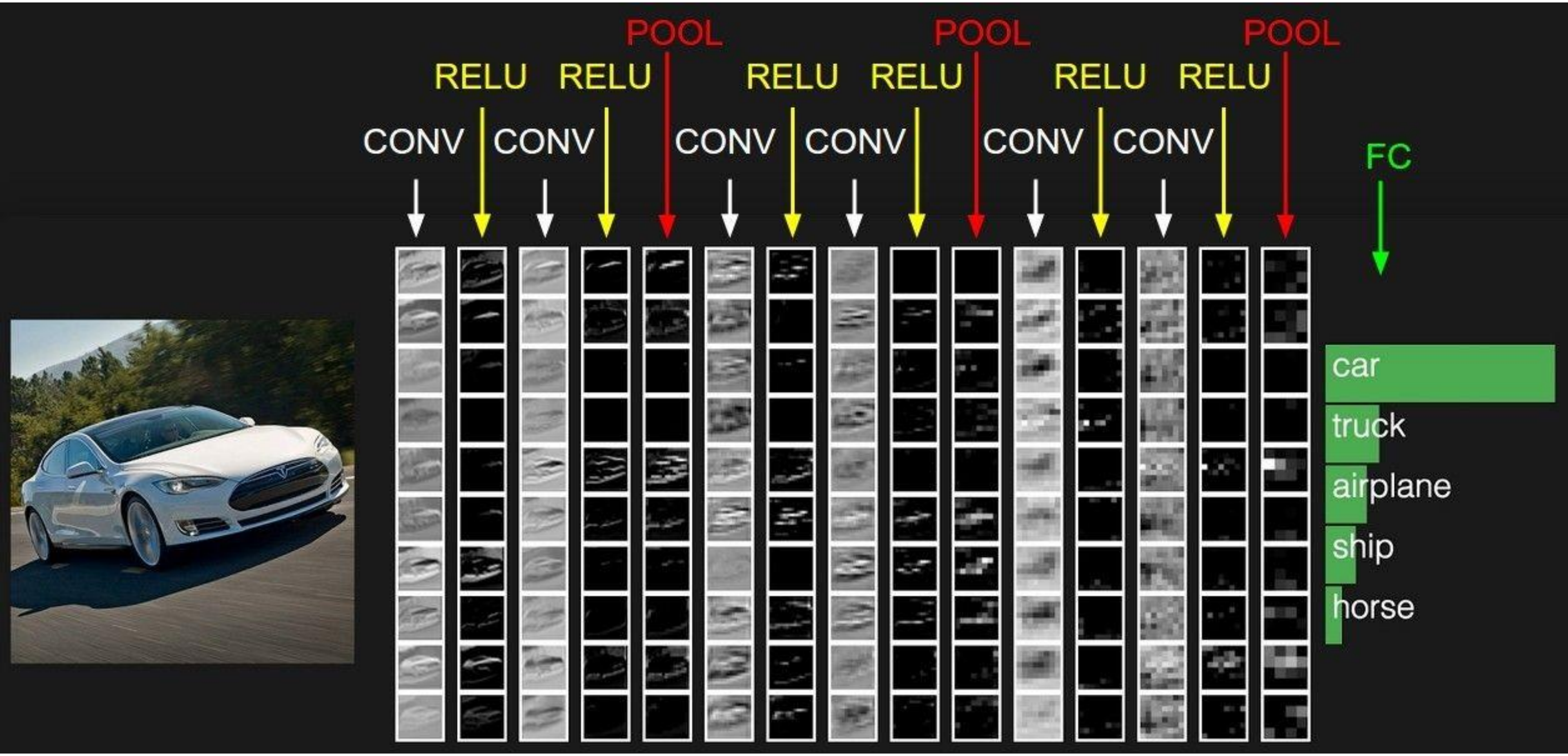
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

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

=> 76*10 = **760**

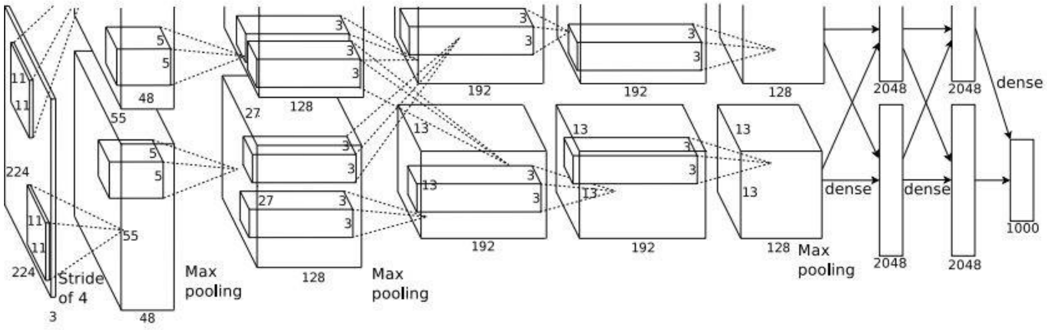
Putting it all together



Andrej Karpathy

Case Study: AlexNet

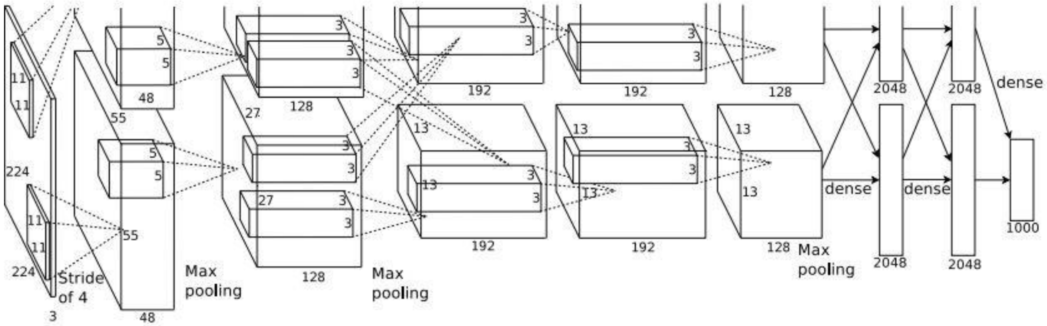
[Krizhevsky et al. 2012]



Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8

Case Study: AlexNet

[Krizhevsky et al. 2012]

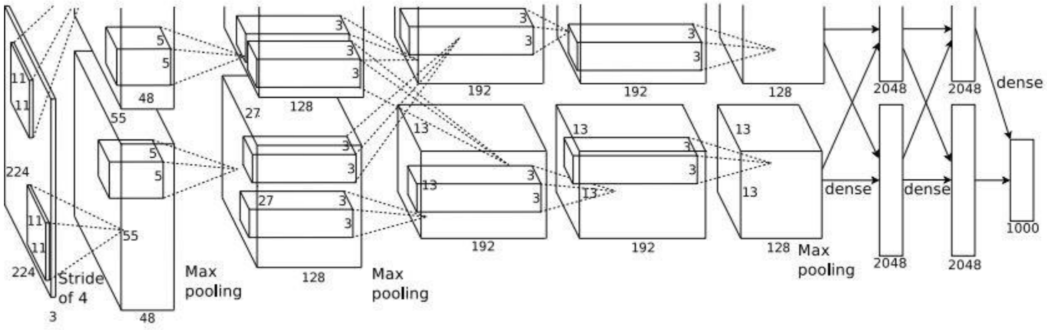


Input: 227x227x3 images

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]



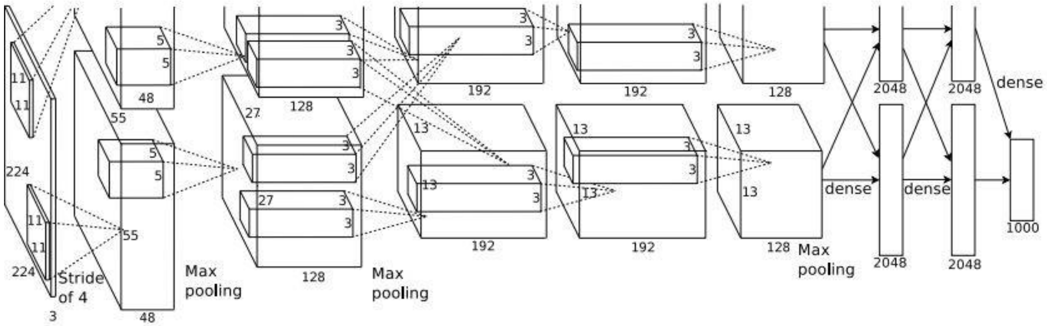
Input: 227x227x3 images
After CONV1: 55x55x96

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

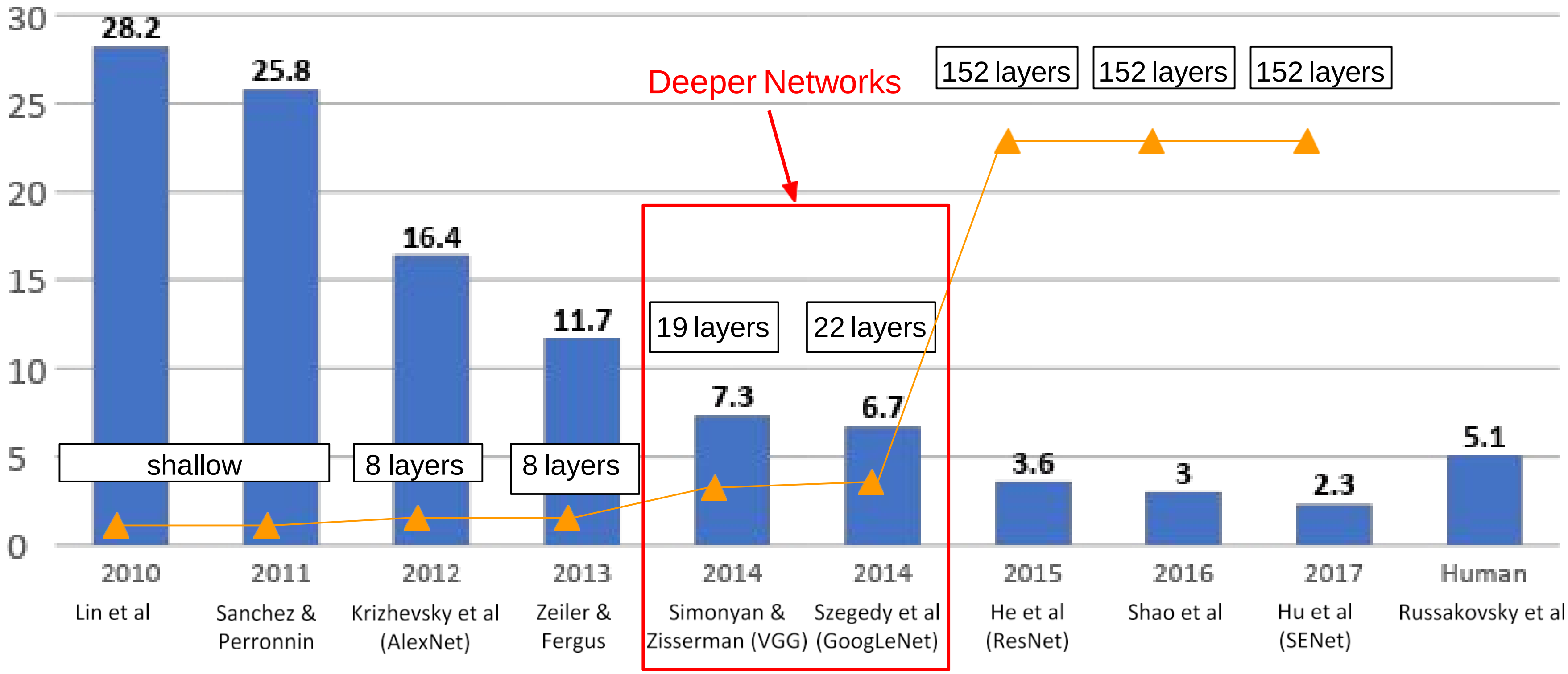
[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] 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

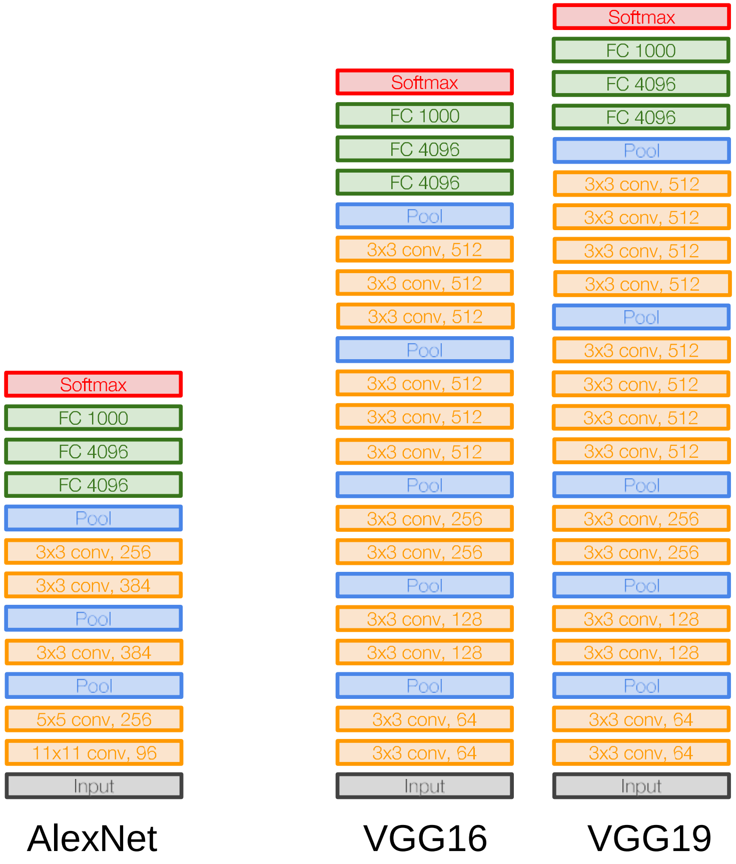
[Simonyan and Zisserman, 2014]

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

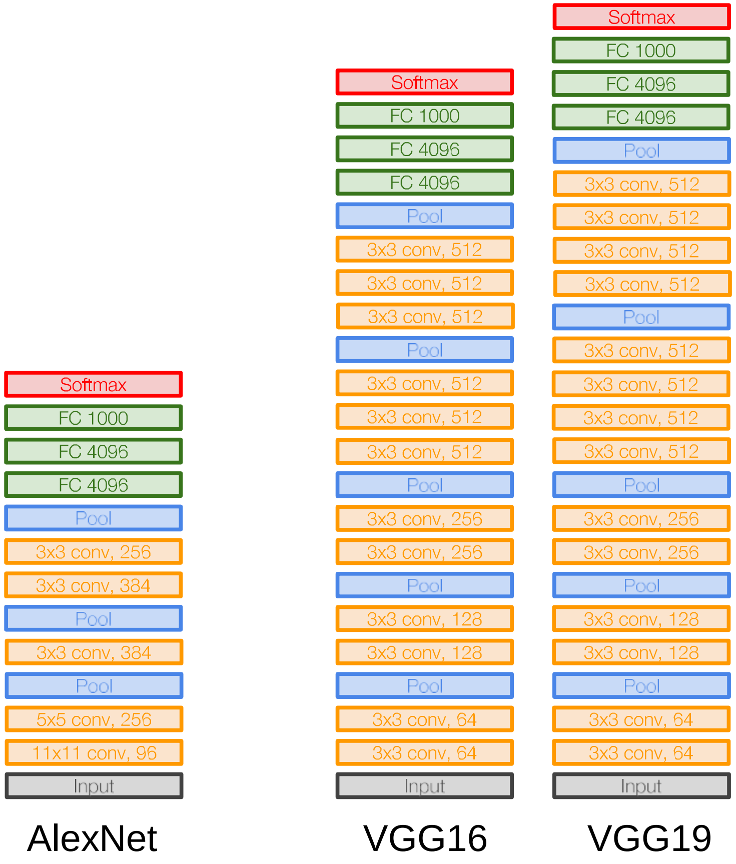
[Simonyan and Zisserman, 2014]

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



Softmax
FC 1000
FC 4096
FC 4096
Pool
3x3 conv, 256
3x3 conv, 384
Pool
3x3 conv, 384
Pool
5x5 conv, 256
11x11 conv, 96
Input

AlexNet

Softmax
FC 1000
FC 4096
FC 4096
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 256
3x3 conv, 256
Pool
3x3 conv, 128
3x3 conv, 128
Pool
3x3 conv, 64
3x3 conv, 64
Input

VGG16

Softmax
FC 1000
FC 4096
FC 4096
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 256
3x3 conv, 256
Pool
3x3 conv, 128
3x3 conv, 128
Pool
3x3 conv, 64
3x3 conv, 64
Input

VGG19

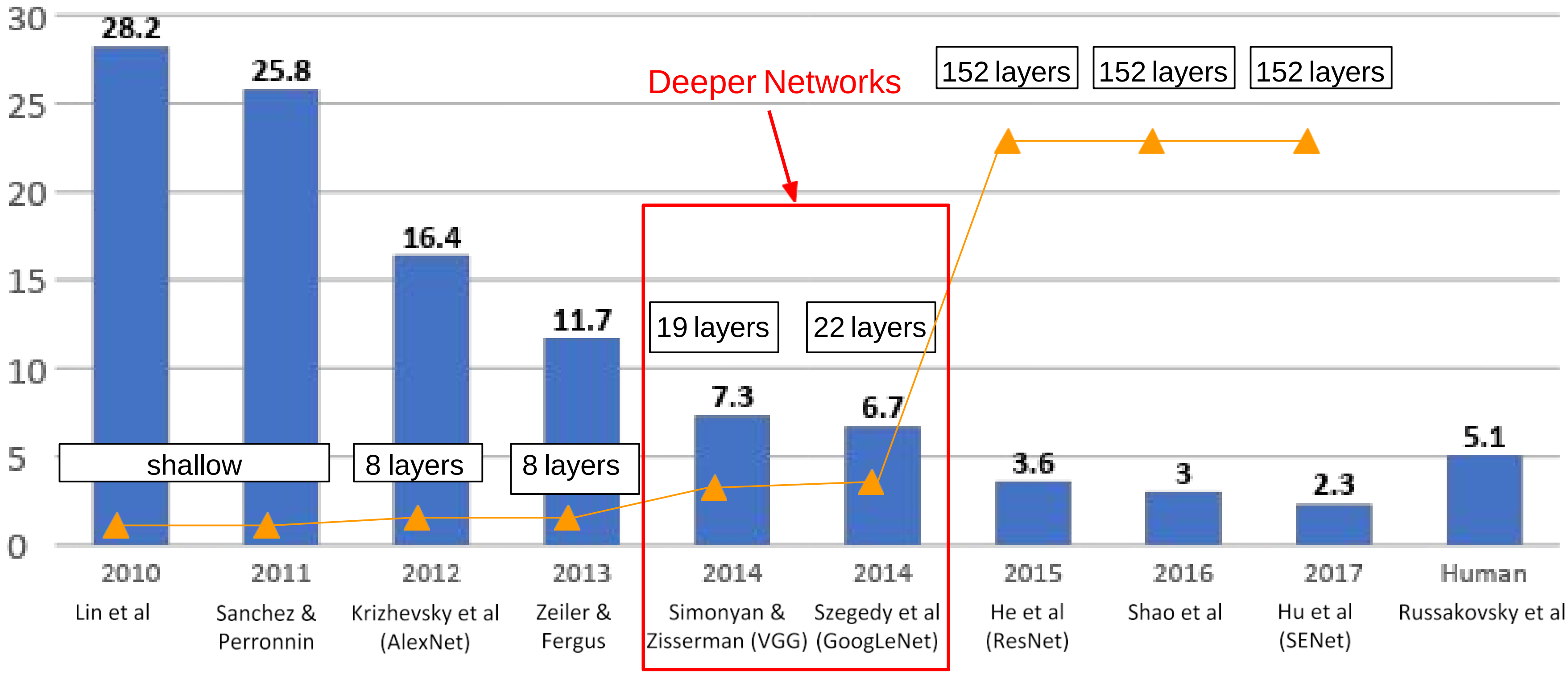
Case Study: VGGNet

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 bytes ~= 96MB / image (for a forward pass)
TOTAL params: 138M parameters

VGG16

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

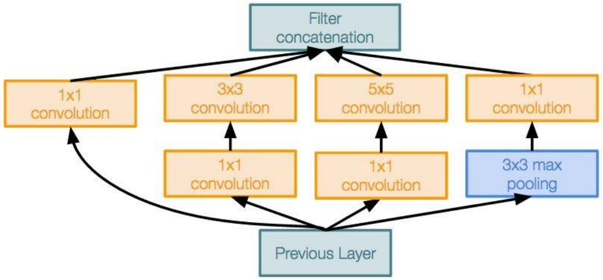


Case Study: GoogLeNet

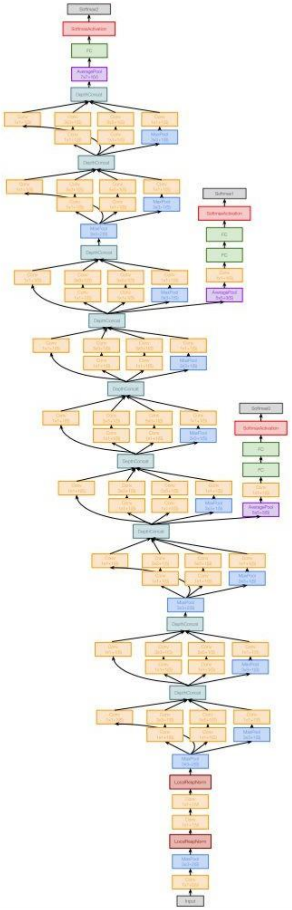
[Szegedy et al., 2014]

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)



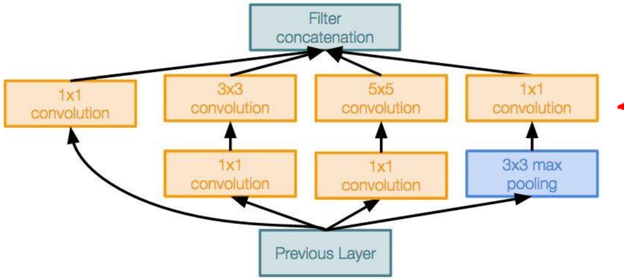
Inception module



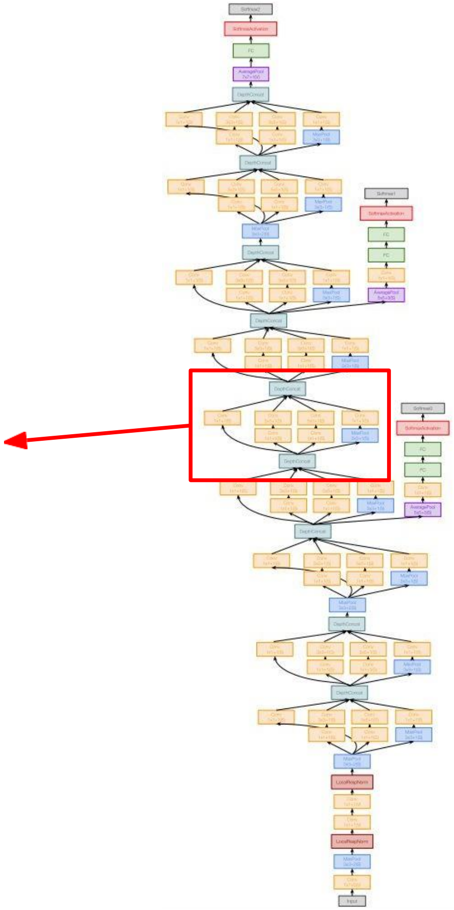
Case Study: GoogLeNet

[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

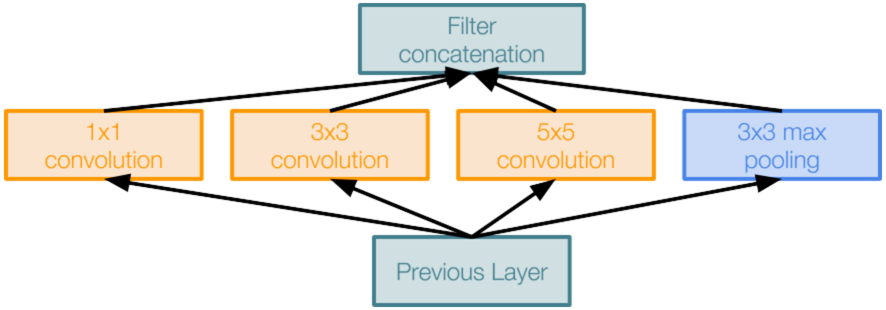


Inception module



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)

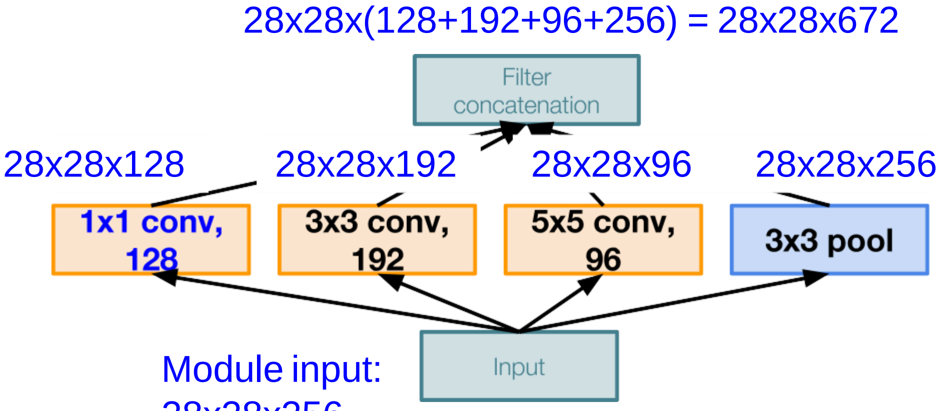
Concatenate all filter outputs together depth-wise

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



28x28x(128+192+96+256) = 28x28x672

28x28x128 28x28x192 28x28x96 28x28x256

Module input: 28x28x256

Naive Inception module

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

Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
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

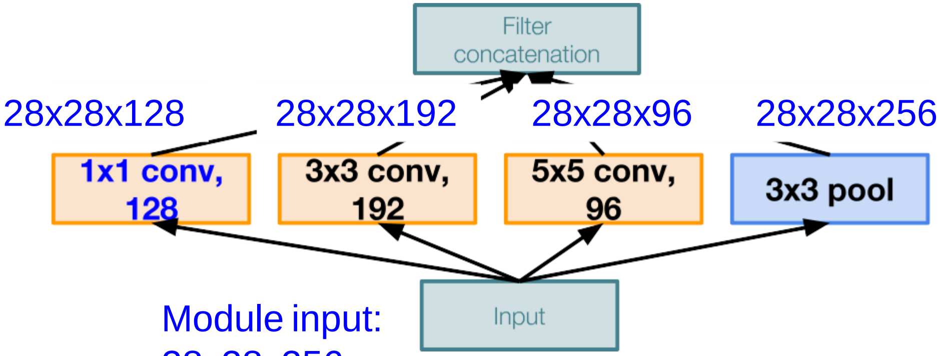
[Szegedy et al., 2014]

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

Example:

Q3:What is output size after filter concatenation?

28x28x(128+192+96+256) = **529k**

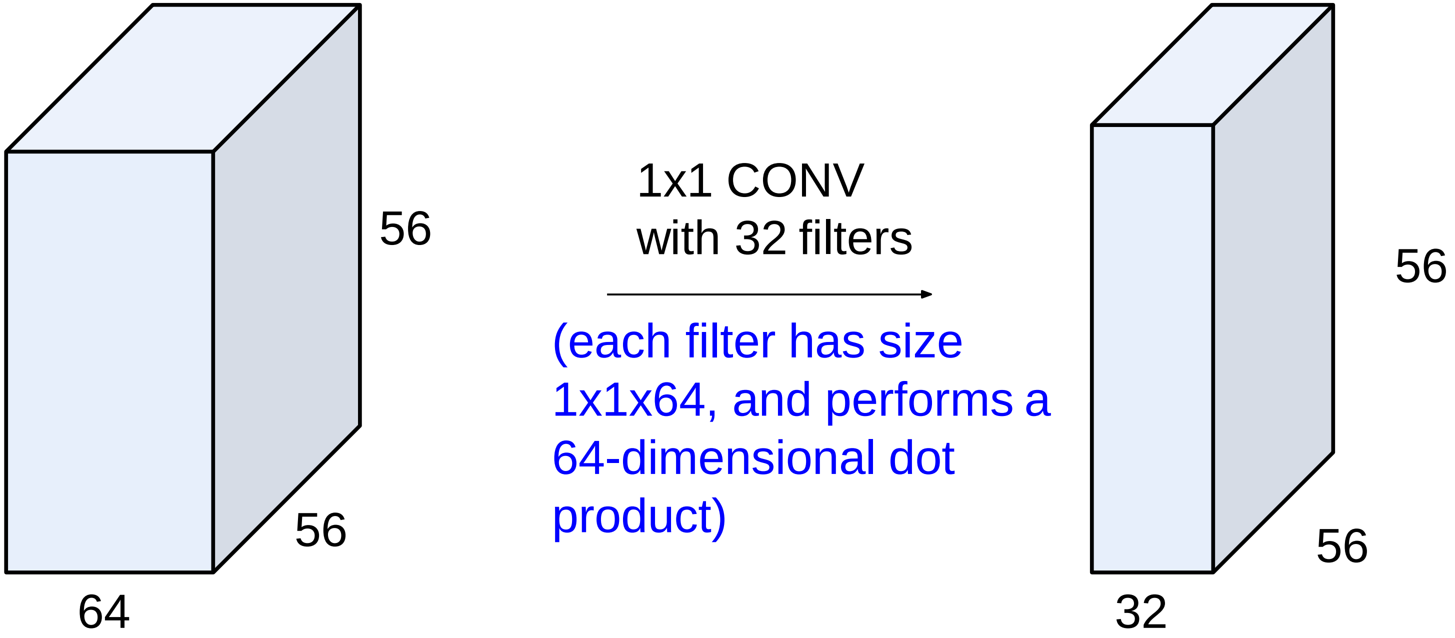


Module input: 28x28x256

Naive Inception module

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

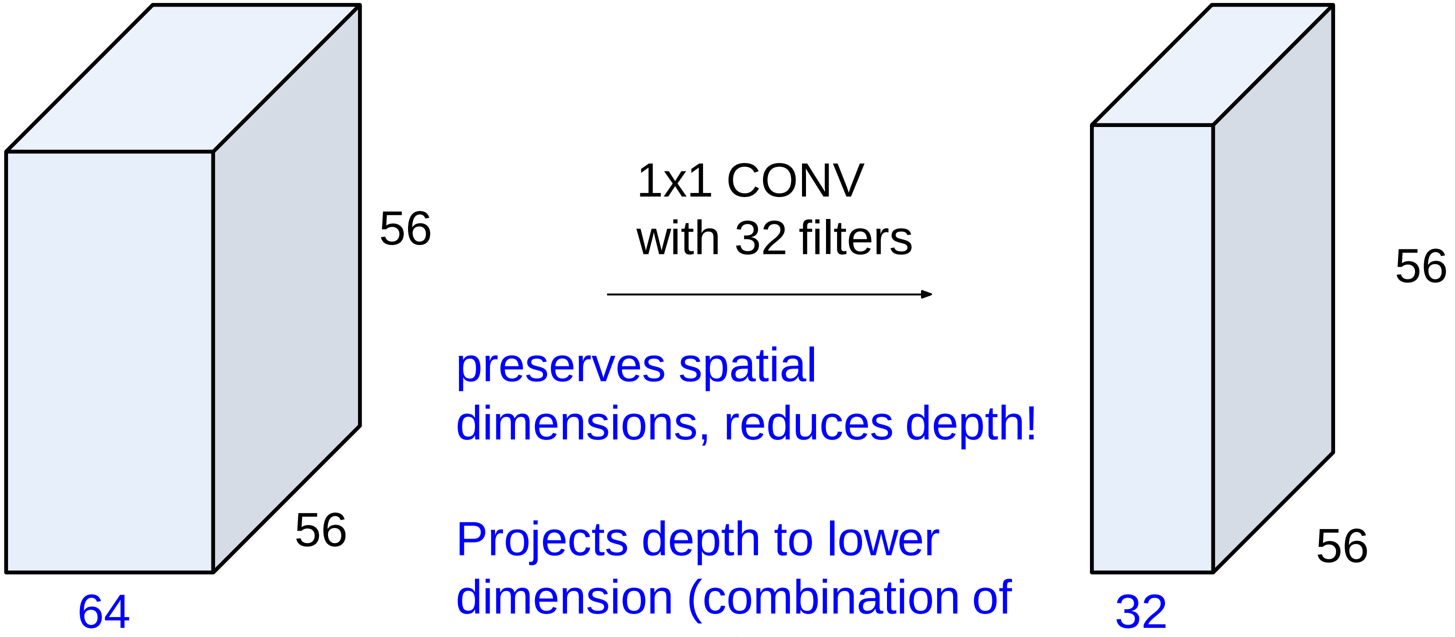
Reminder: 1x1 convolutions



1x1 CONV
with 32 filters

(each filter has size 1x1x64, and performs a 64-dimensional dot product)

Reminder: 1x1 convolutions



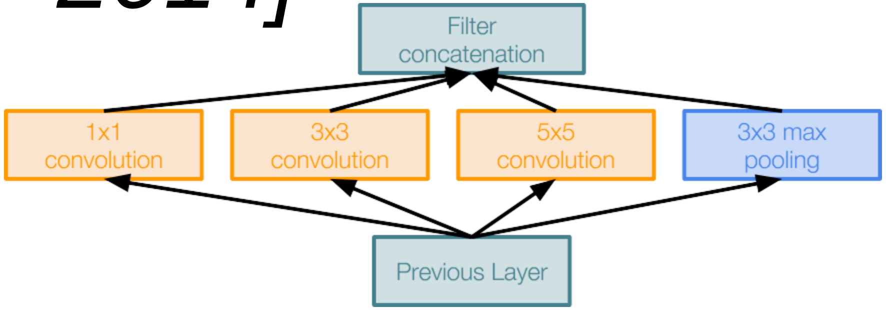
1x1 CONV
with 32 filters

preserves spatial dimensions, reduces depth!

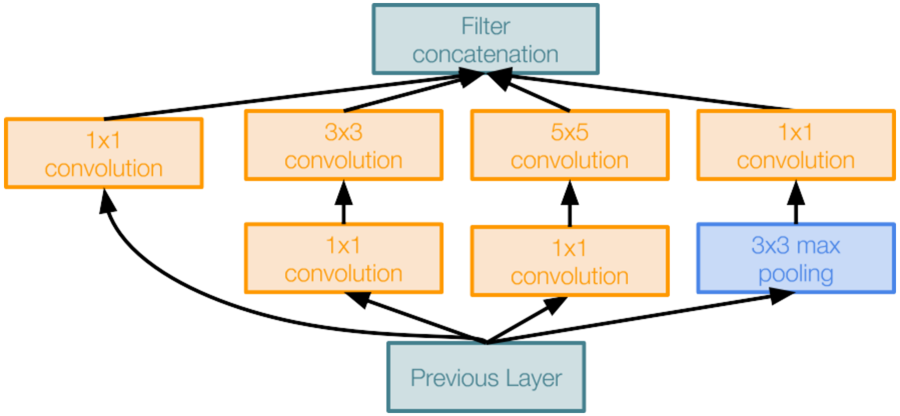
Projects depth to lower dimension (combination of feature maps)

Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

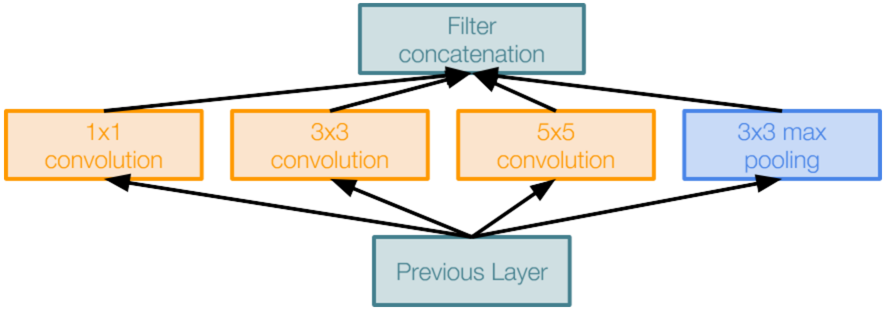


Inception module with dimension reduction

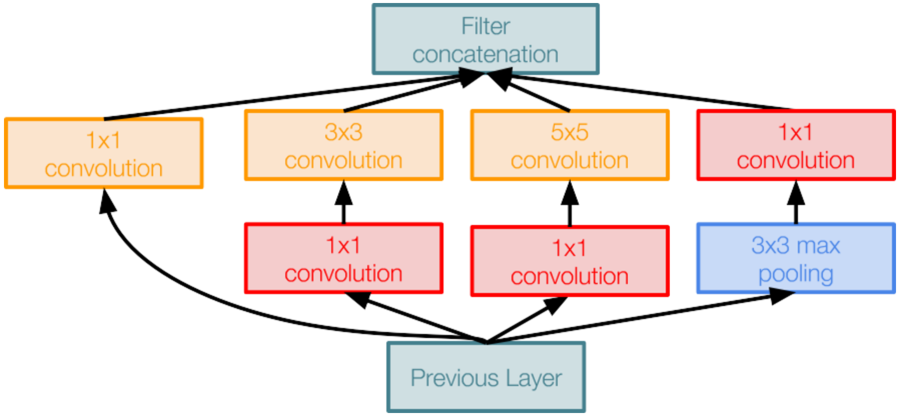
Case Study: GoogLeNet

[Szegedy et al., 2014]

1x1 conv “bottleneck” layers



Naive Inception module



Inception module with dimension reduction

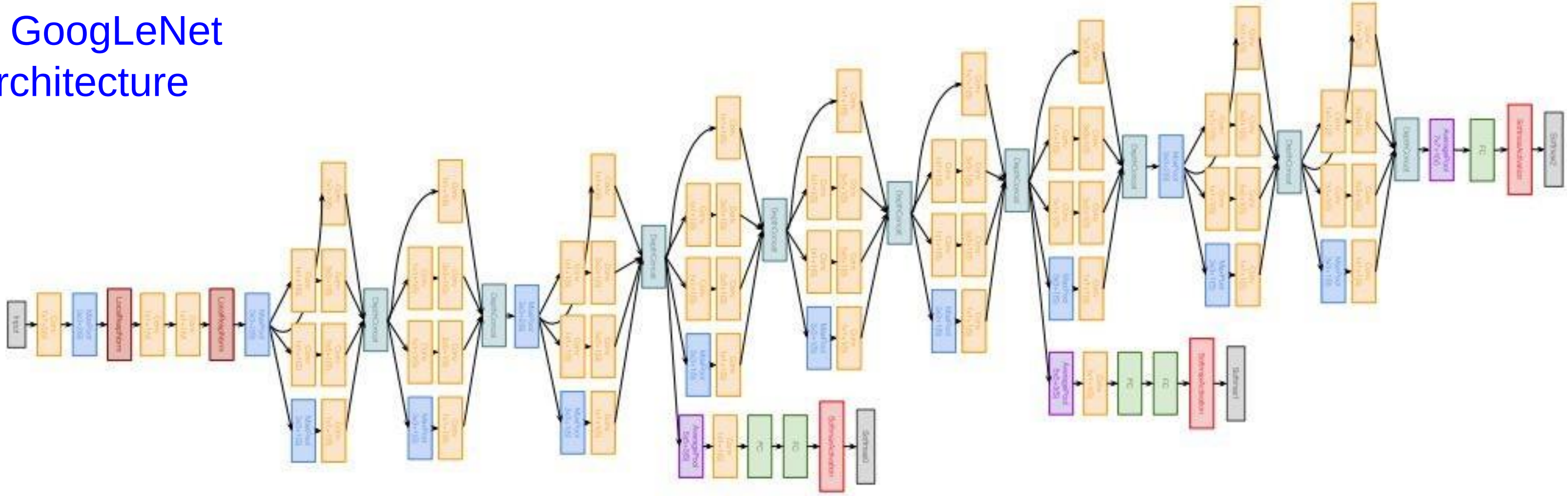
Total: 854M ops

Total: 358M ops

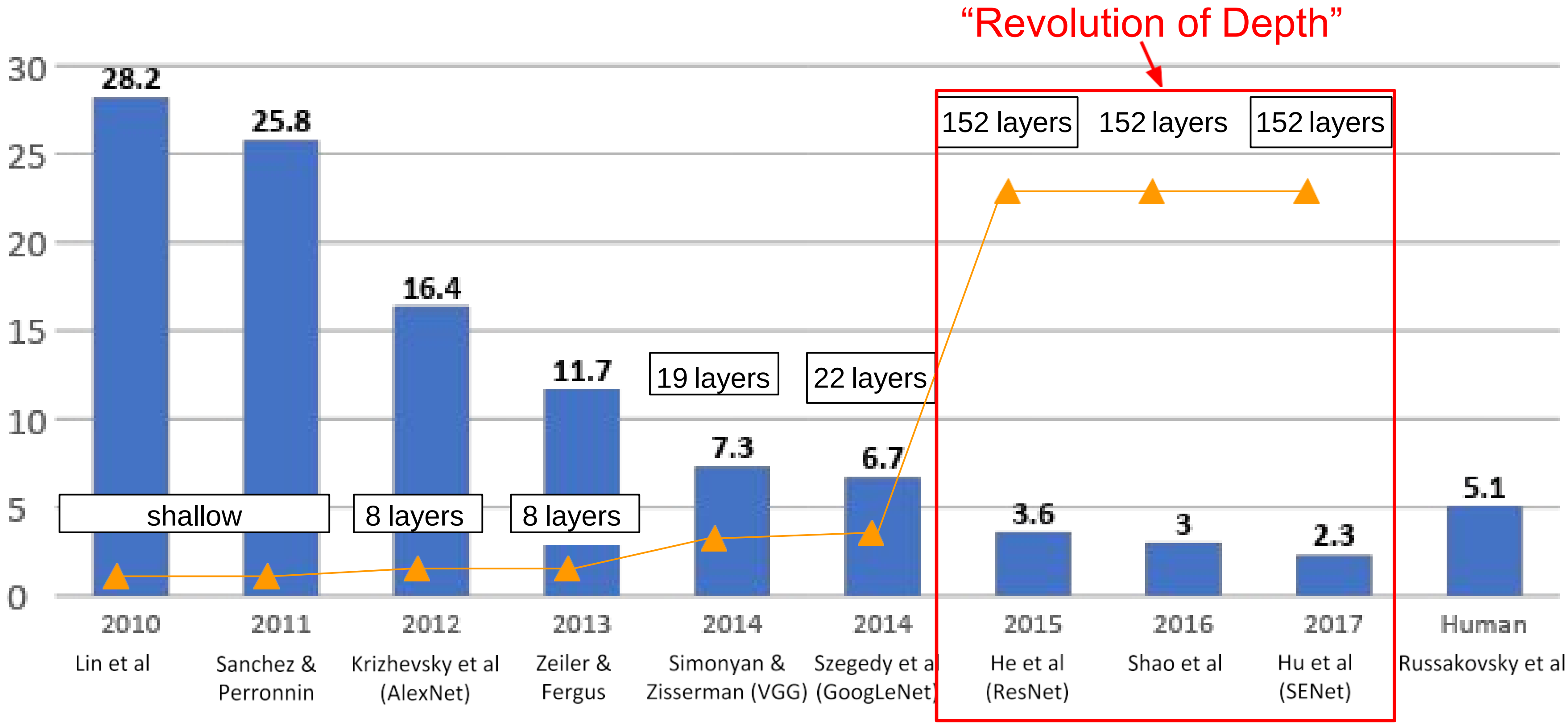
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet
architecture



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

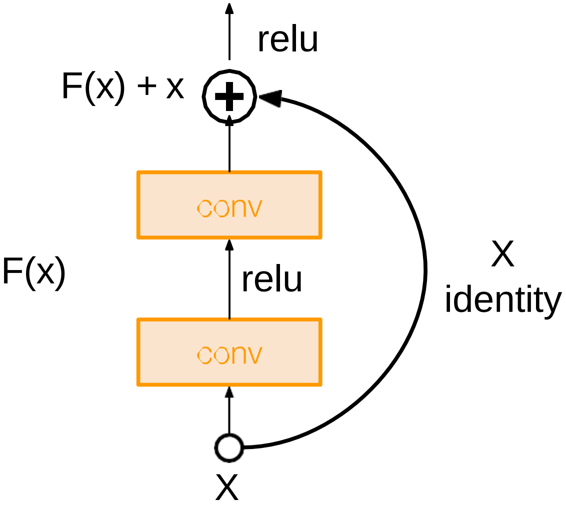


Case Study: ResNet

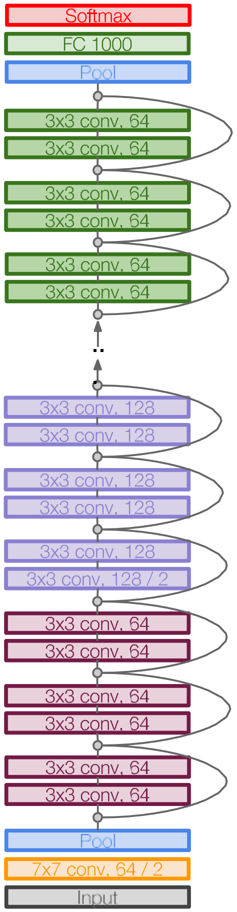
[He et al., 2016]

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!



Residual block



Case Study: ResNet

[He et al., 2016]

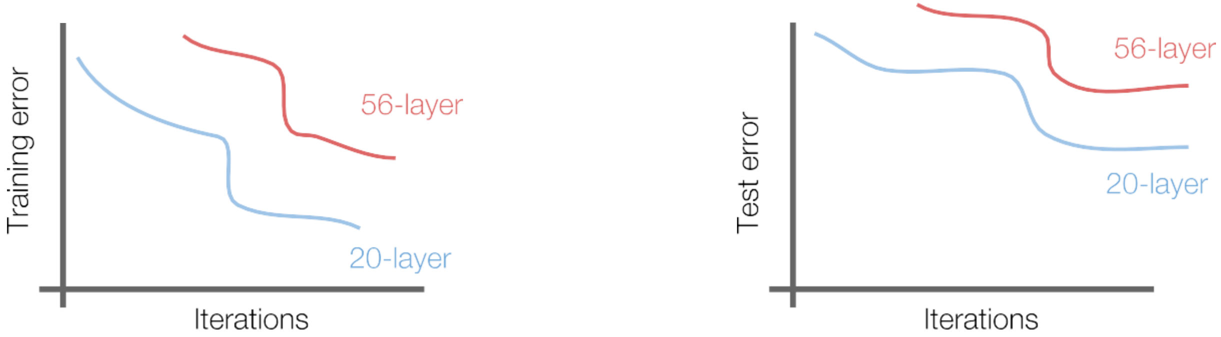
What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

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

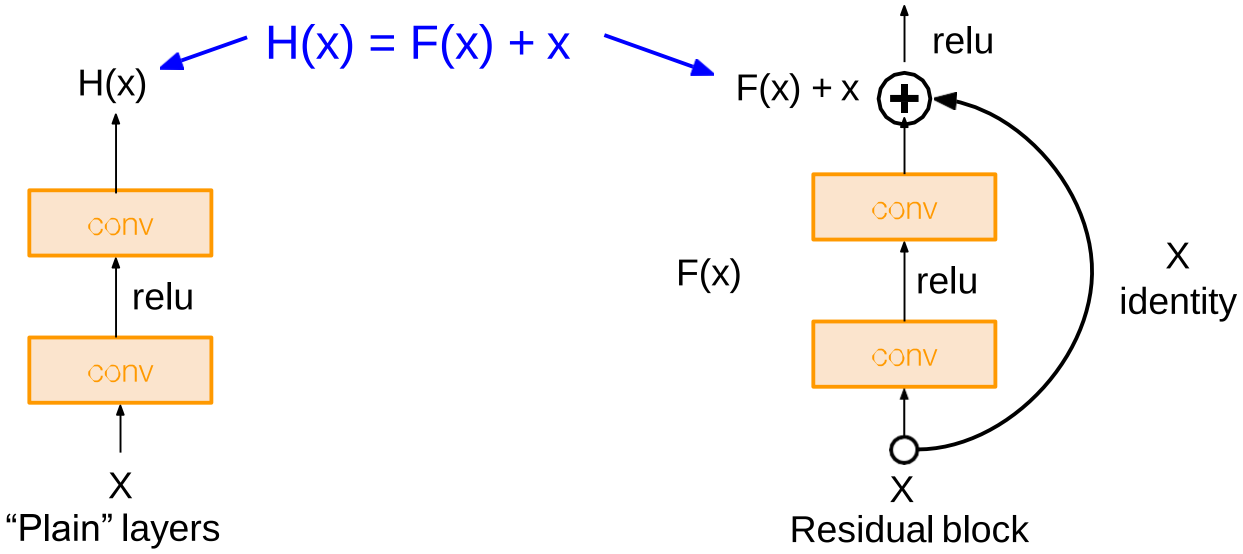
The deeper model should be able to perform at least as well as the shallower model.

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


H(x) = F(x) + x

“Plain” layers

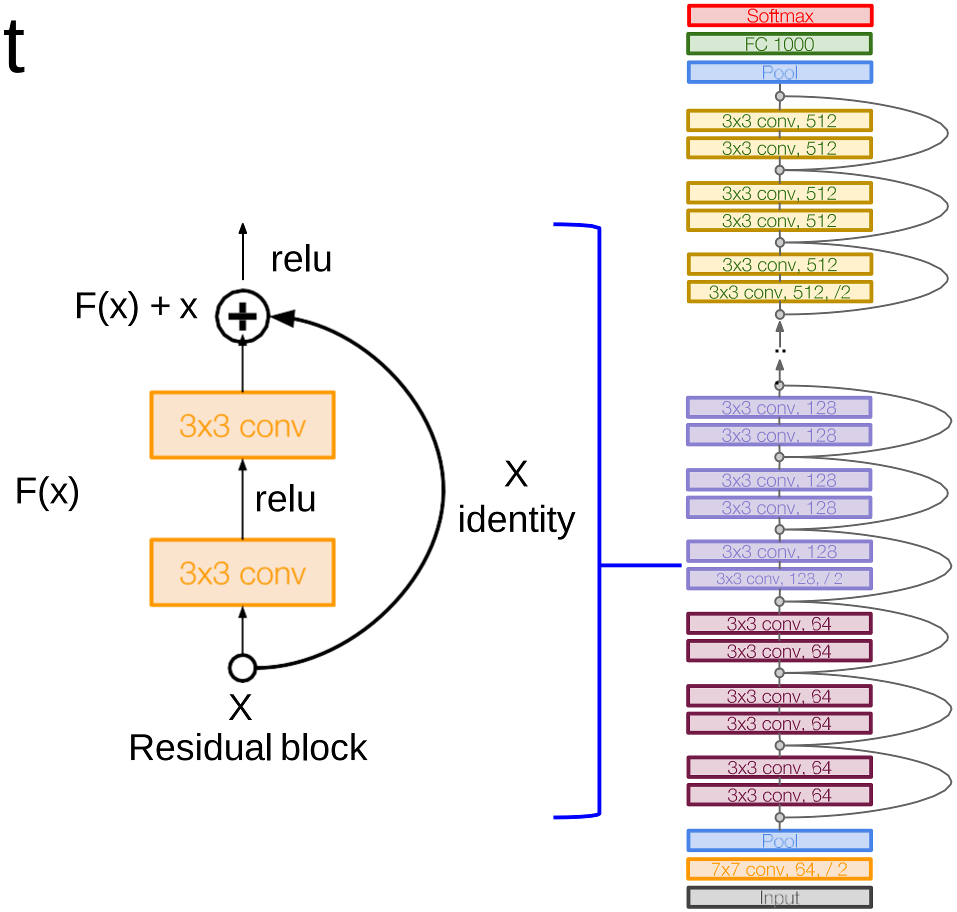
Residual block

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

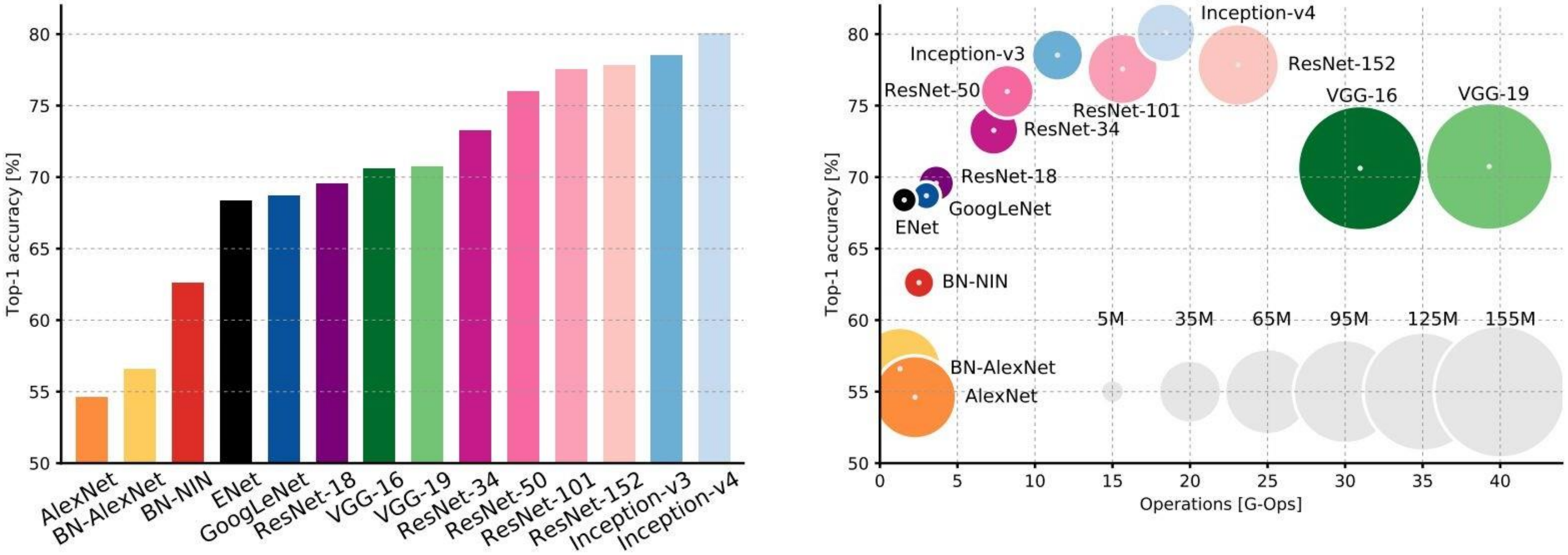
Case Study: ResNet

[He et al., 2016]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers



Comparing complexity...



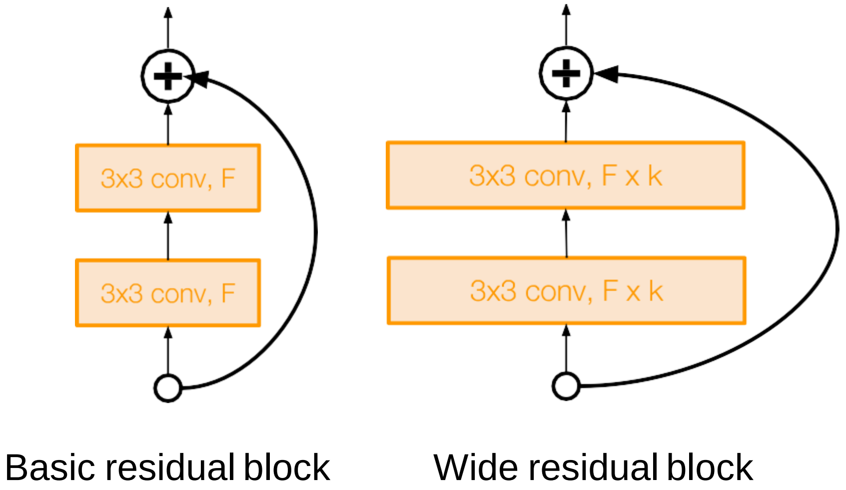
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Improving ResNets...

Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- User wider residual blocks (F x 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)

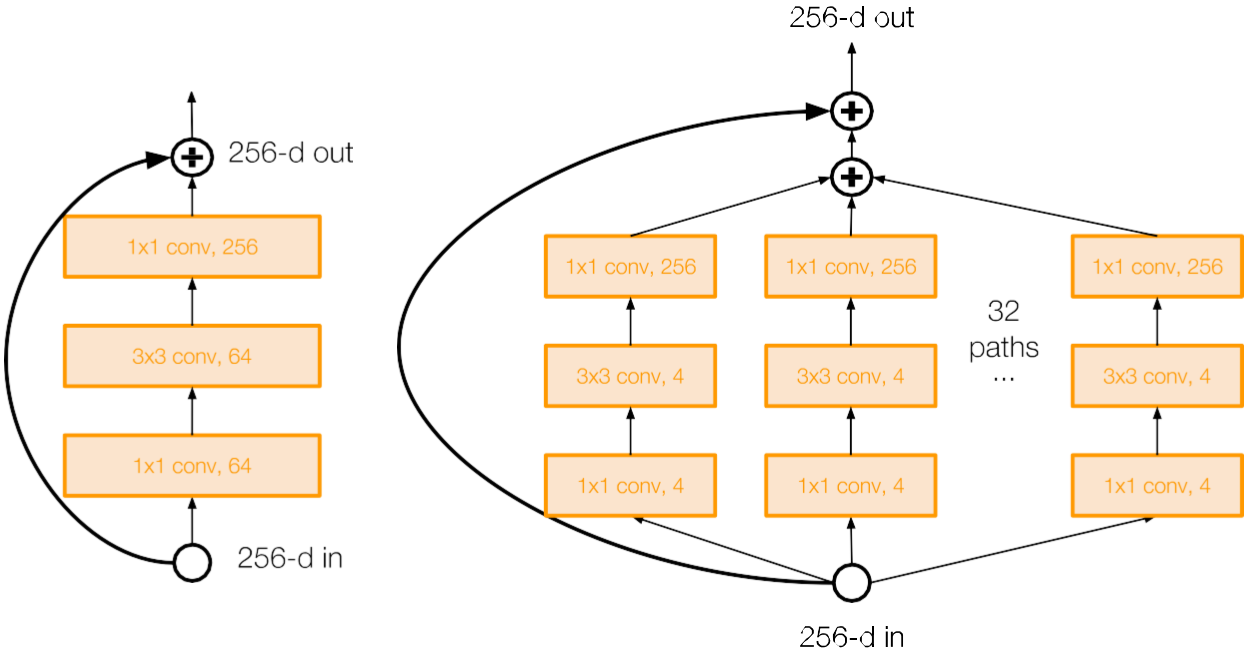


Improving ResNets...

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

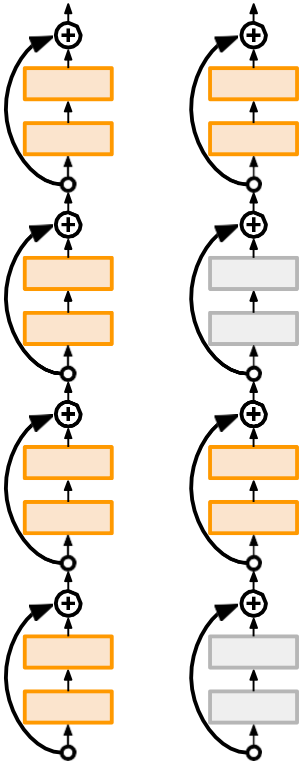


Improving ResNets...

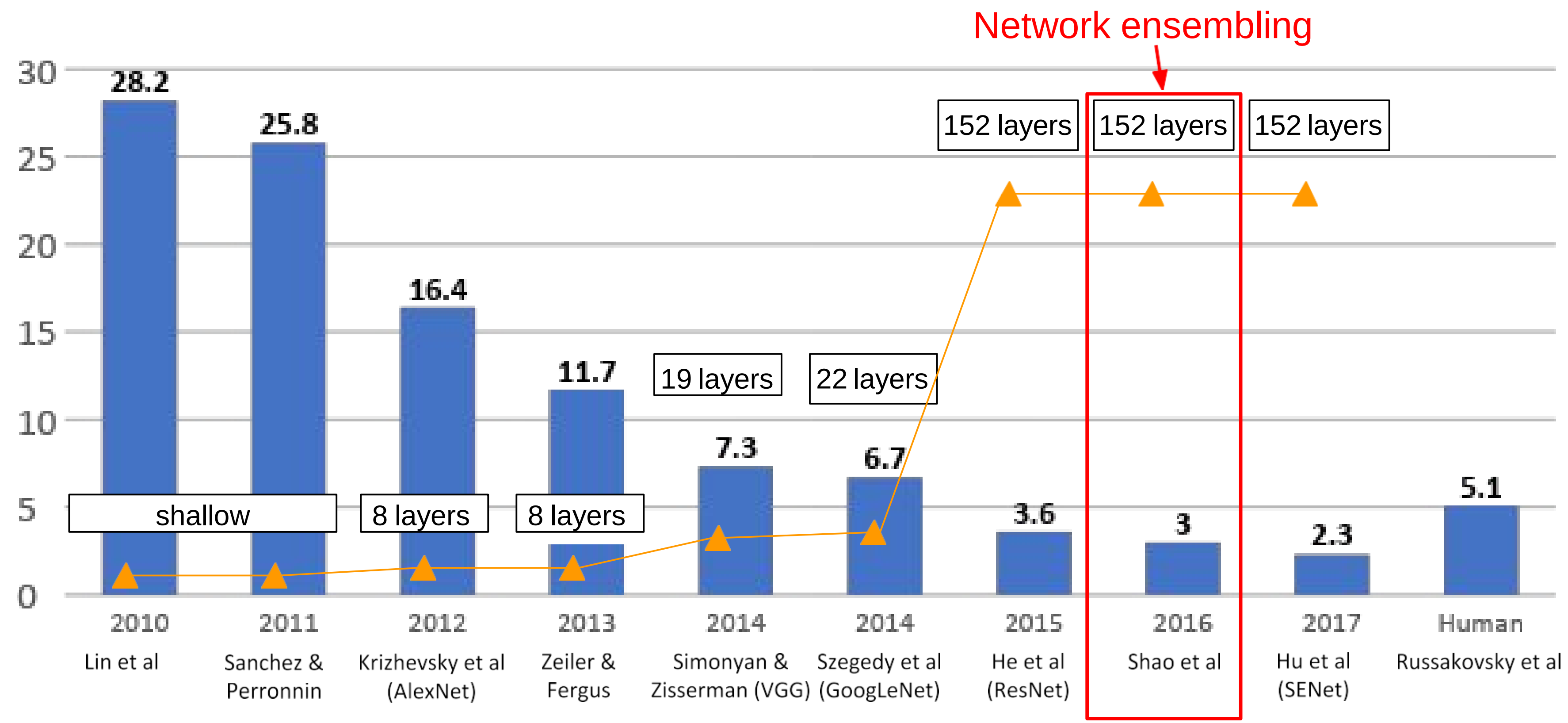
Deep Networks with Stochastic Depth

[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

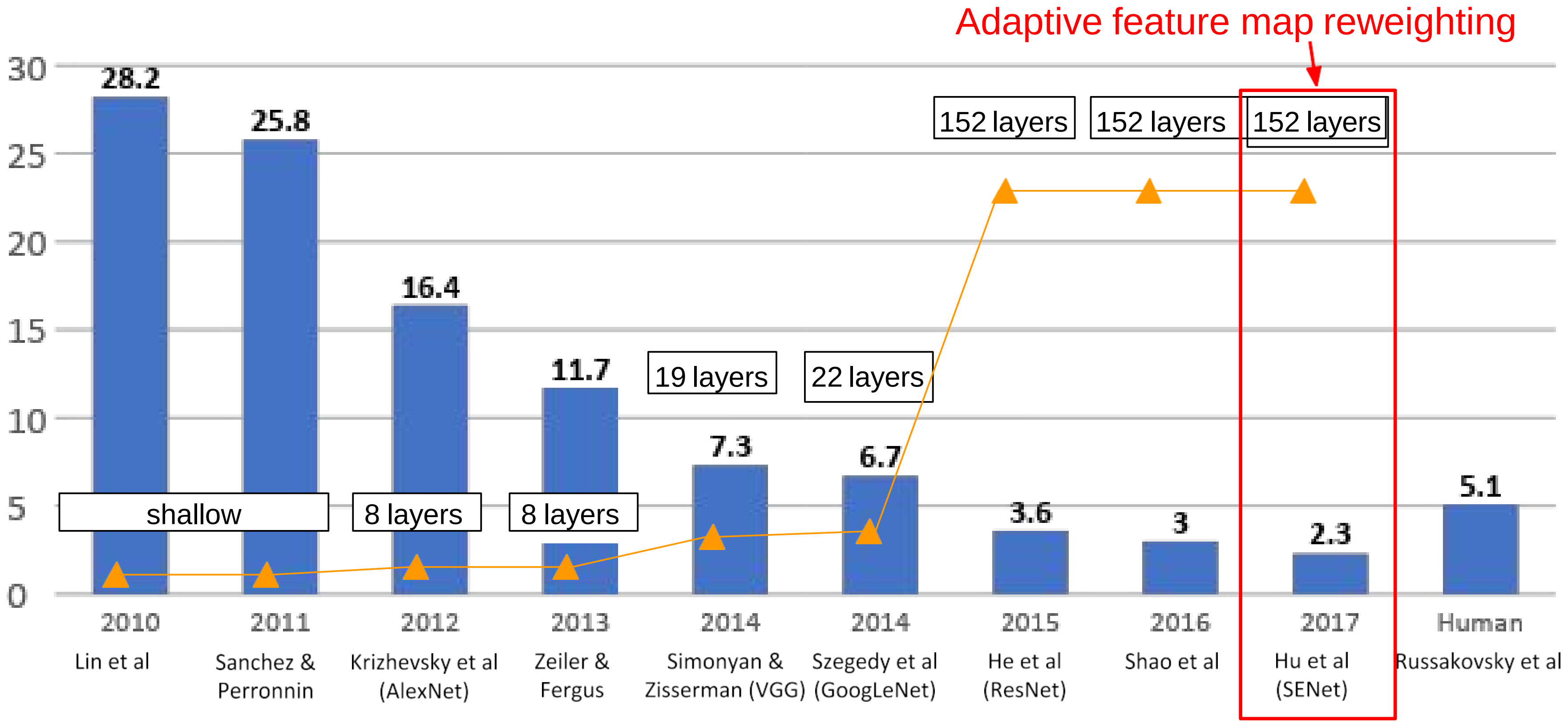


ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Network ensembling

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

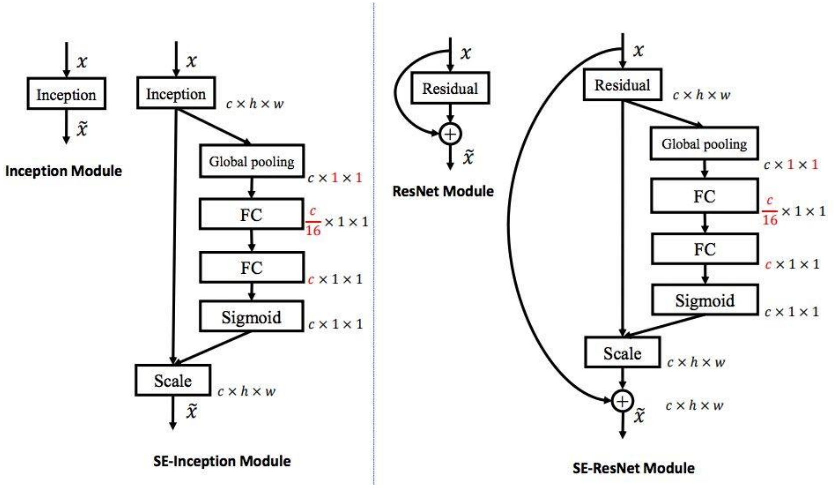


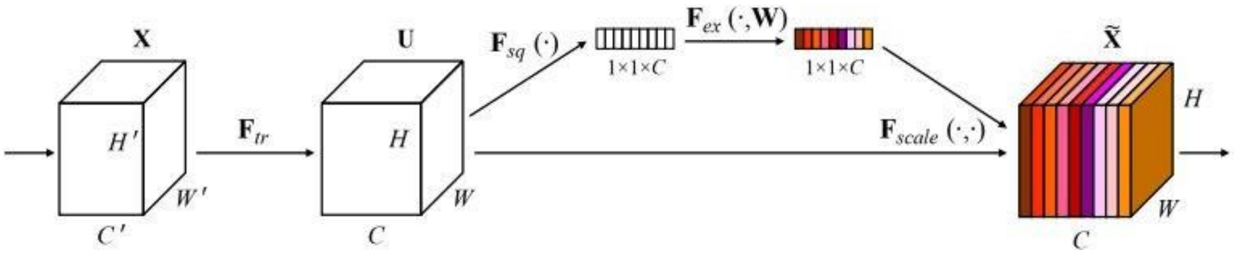
Improving ResNets...

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)



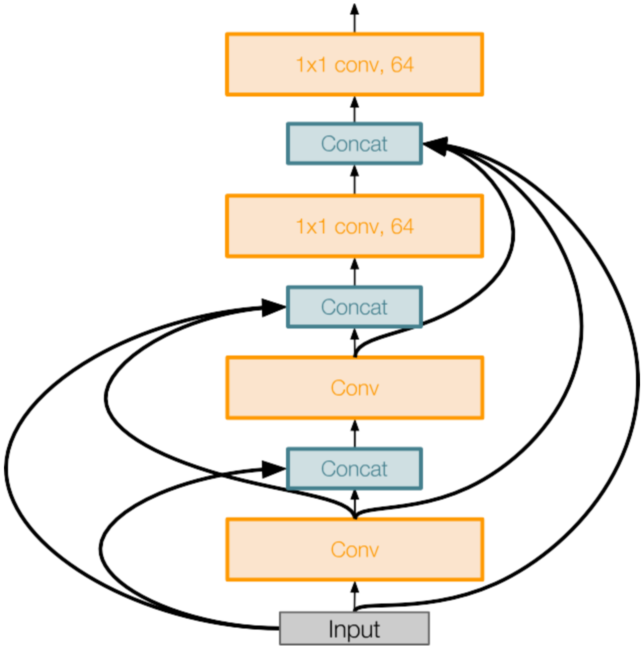


Beyond ResNets...

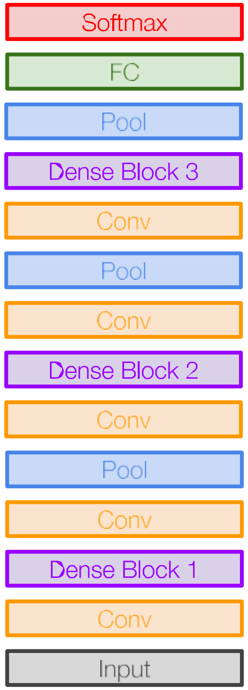
Densely Connected Convolutional Networks

[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



Dense Block



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)

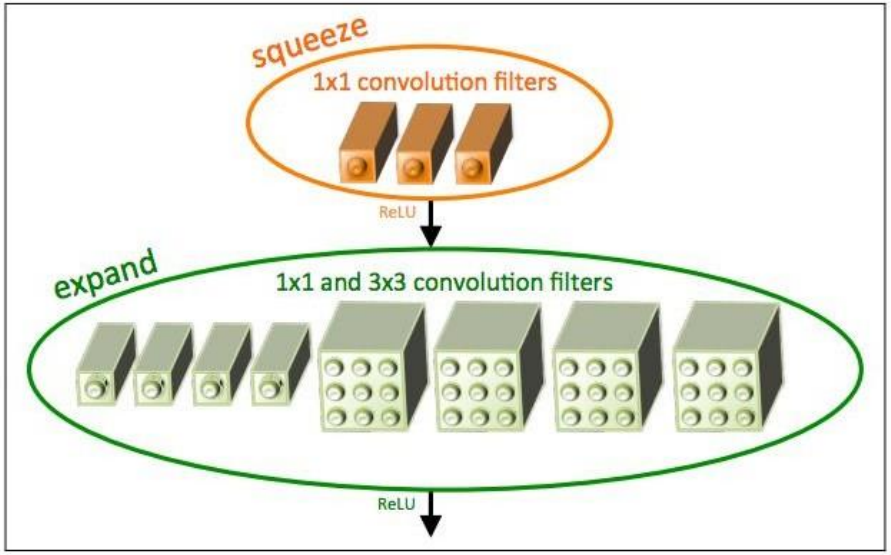


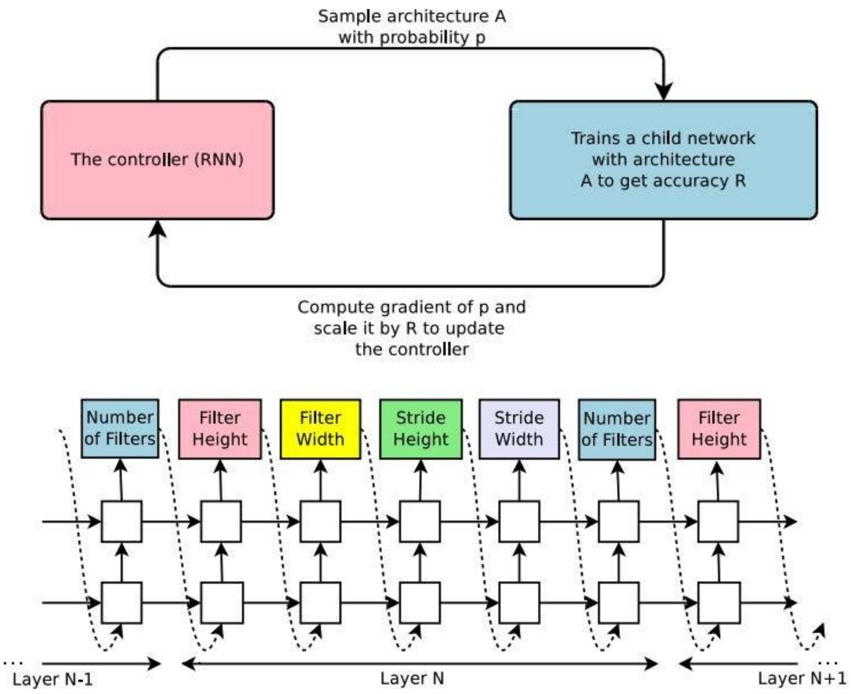
Figure copyright Iandola, Han, Moskewicz, Ashraf, Dally, Keutzer, 2017. Reproduced with permission.

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

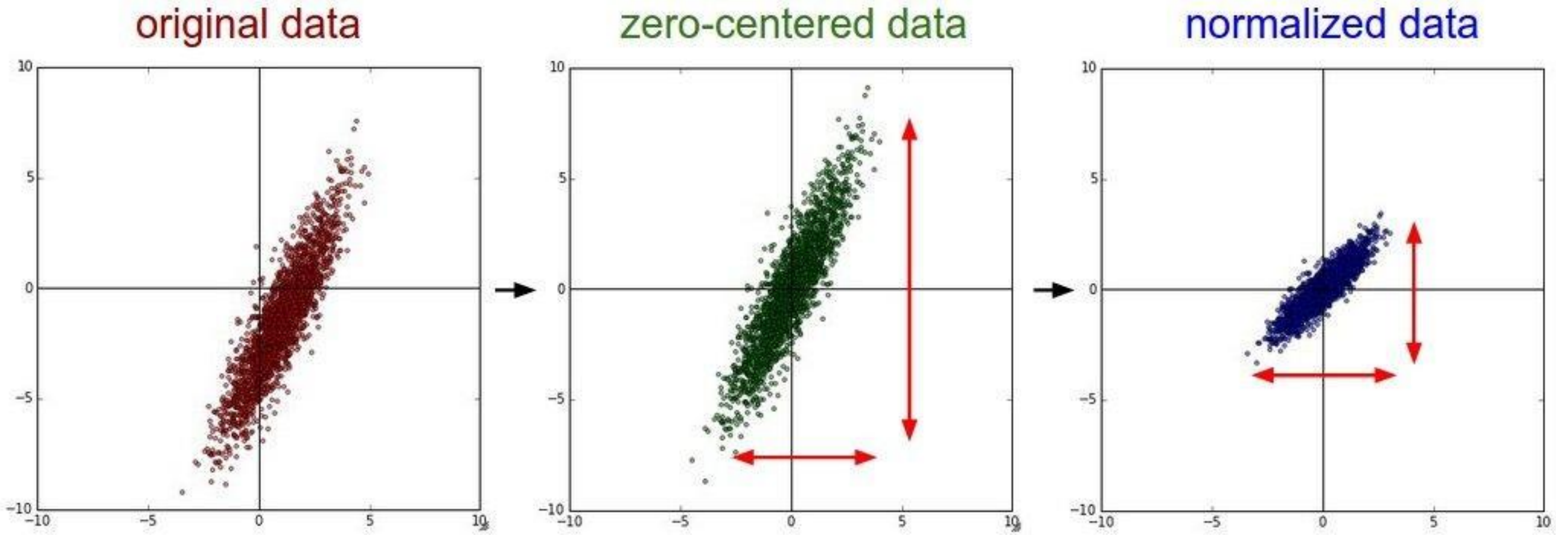
Practical matters

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

Understanding CNNs

- Visualization
- Breaking CNNs

Preprocessing the Data



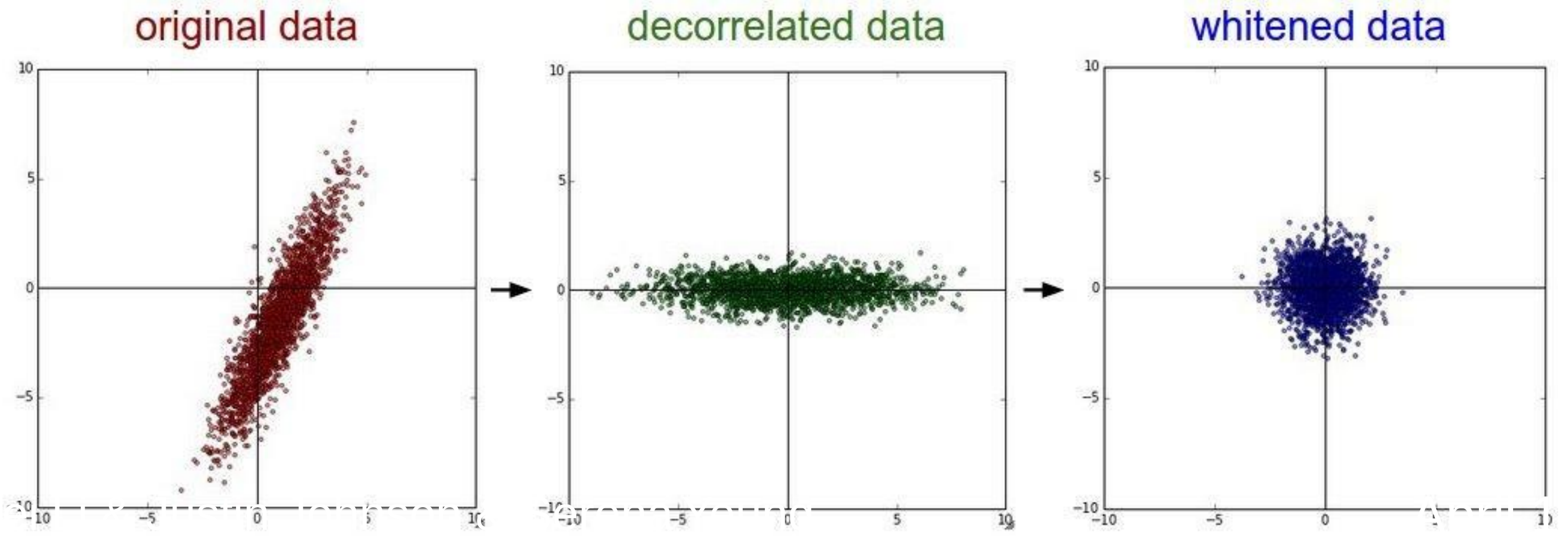
```
X -= np.mean(X, axis = 0)
```

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

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

Preprocessing the Data

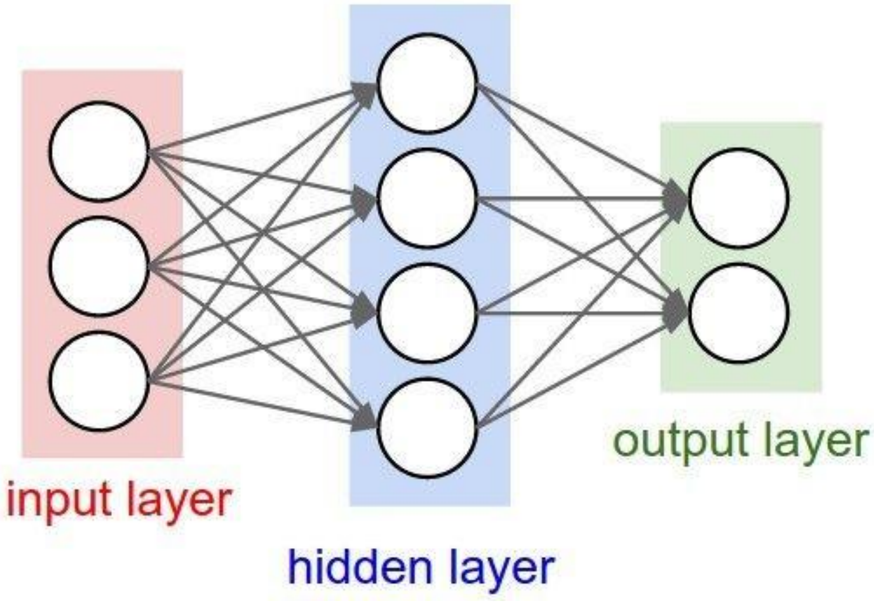
In practice, you may also see **PCA** and **Whitening** of the data



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

Weight Initialization



- Q: what happens when W=constant init is used?

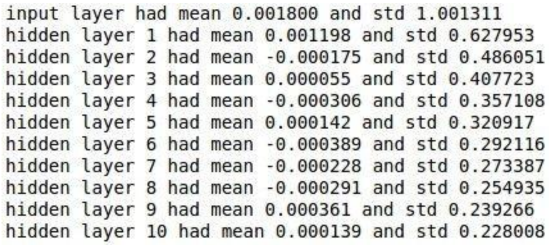
Weight Initialization

- Another idea: **Small random numbers**

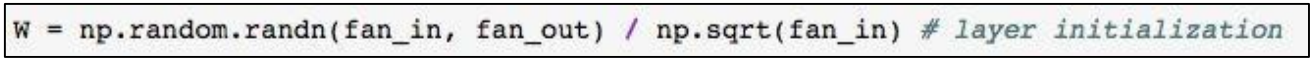
(gaussian with zero mean and 1e-2 standard deviation)

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

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

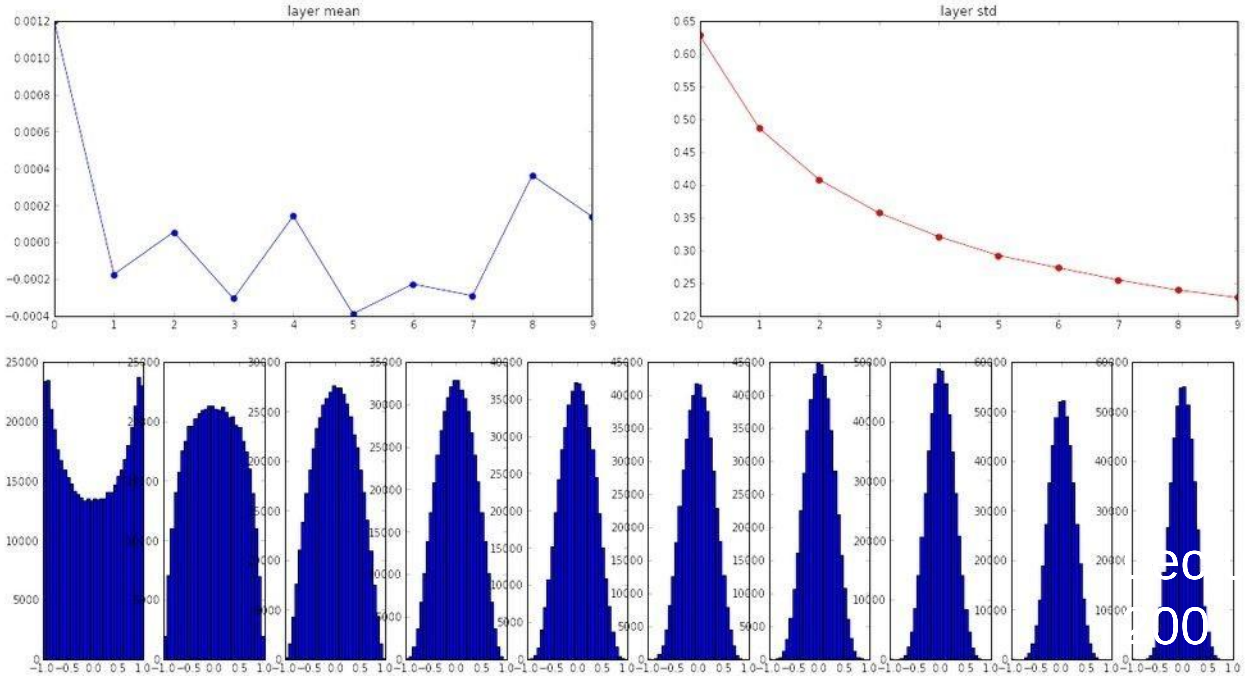


```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917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

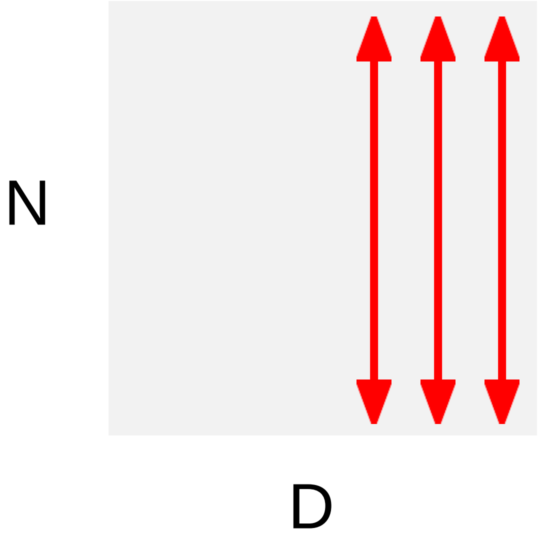
consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

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

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

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

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

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

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

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

Note, the network can learn:

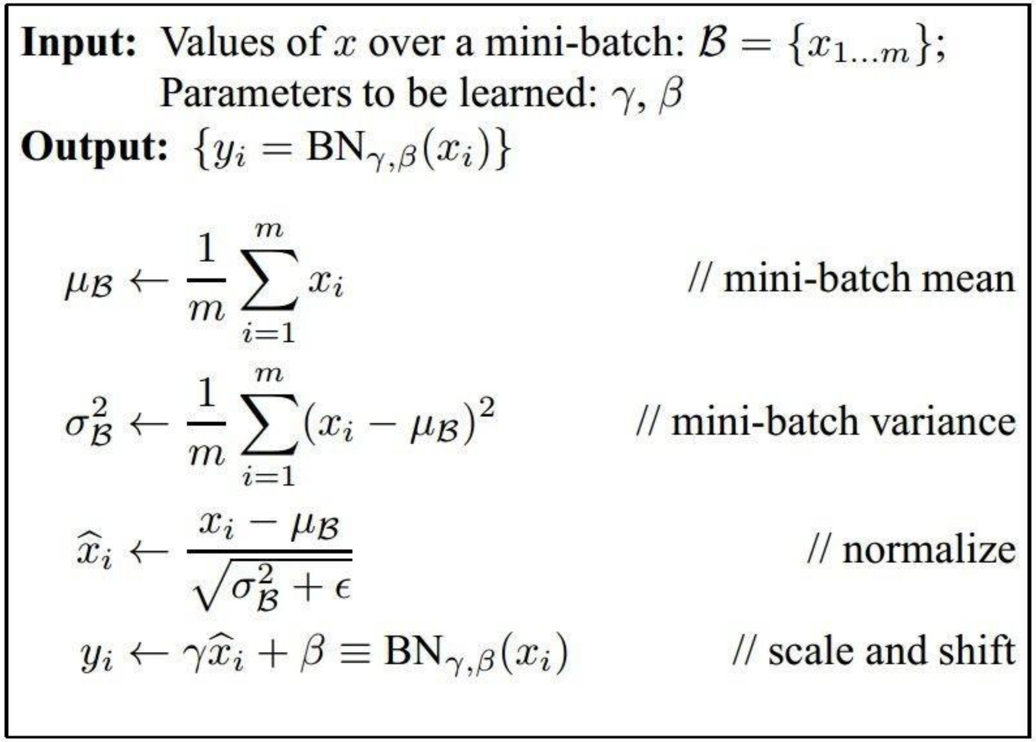
$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

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

to recover the identity mapping.

Batch Normalization

[Ioffe and Szegedy, 2015]

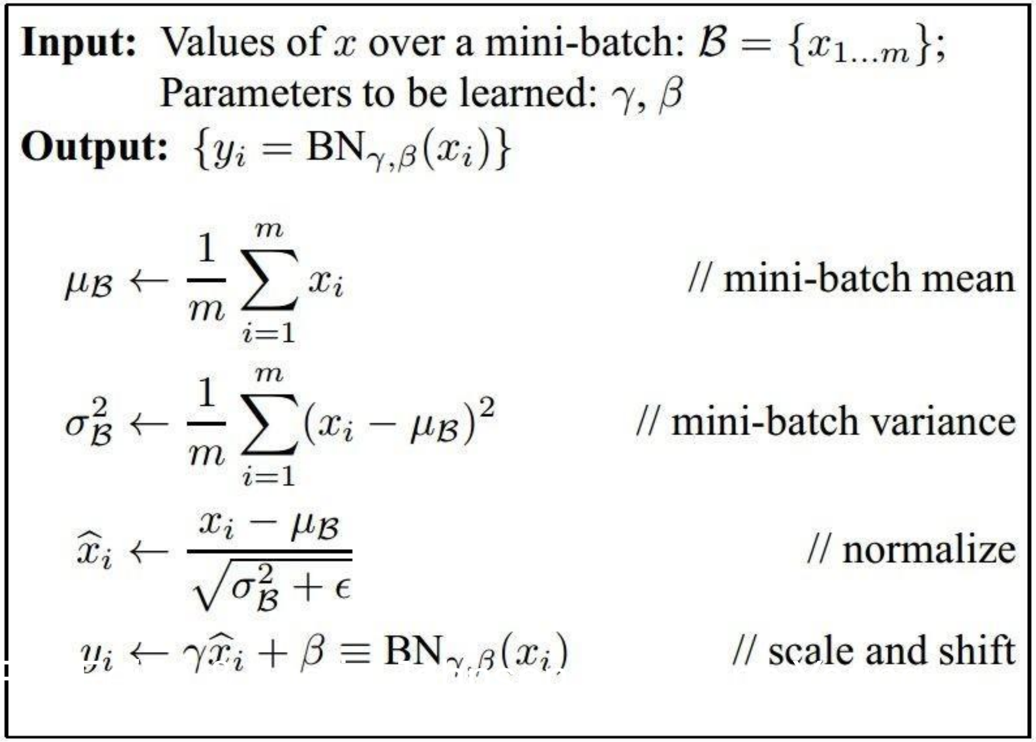


```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                              // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$     // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$   // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$       // 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: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

Note: at test time BatchNorm layer functions differently:

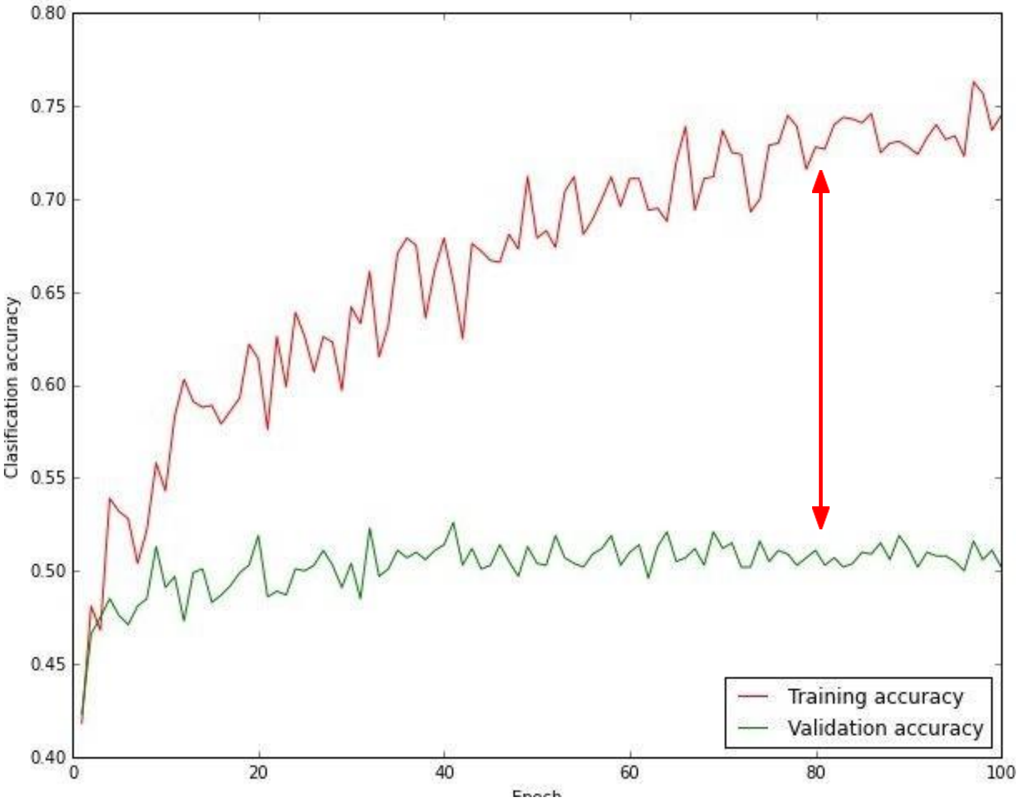
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

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

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 … 1e-5]
- Coarse-to-fine search for hyperparameters (e.g. learning rate, regularization)

Monitor and visualize accuracy



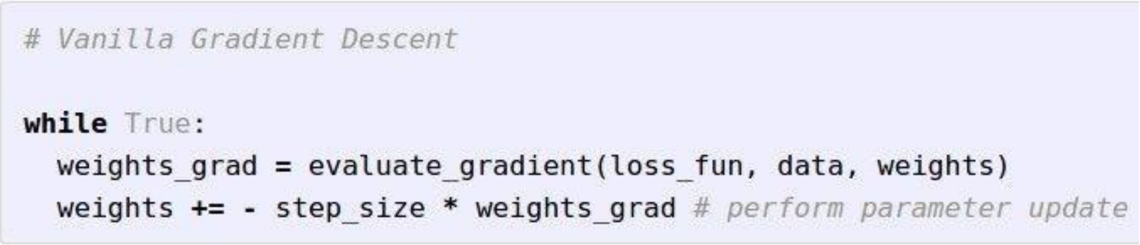
big gap = overfitting

=> increase regularization strength?

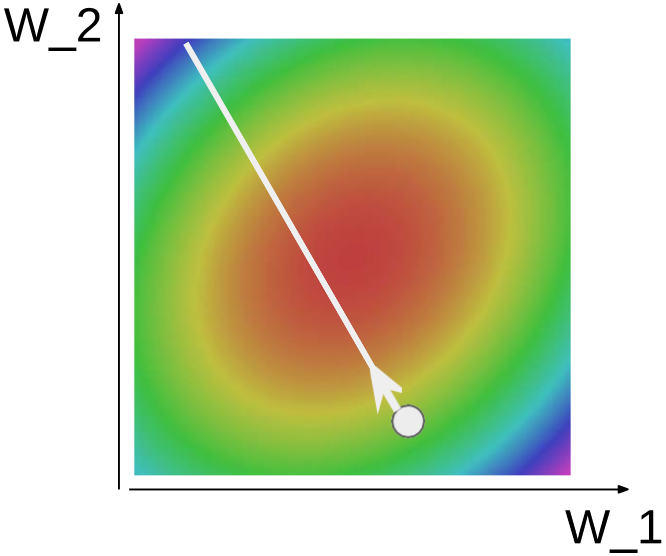
no gap

=> increase model capacity?

Optimization

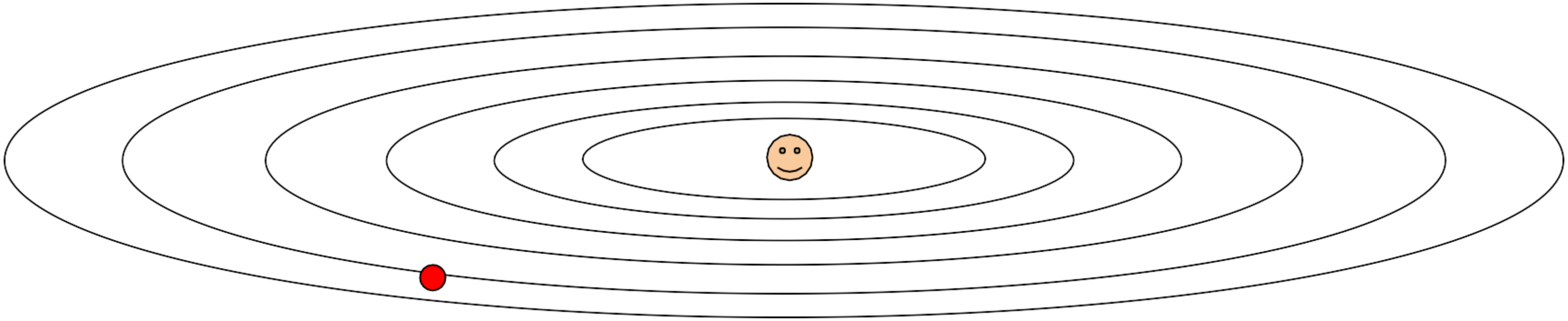


```
# Vanilla Gradient Descent

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



Optimization:
Problems with SGD

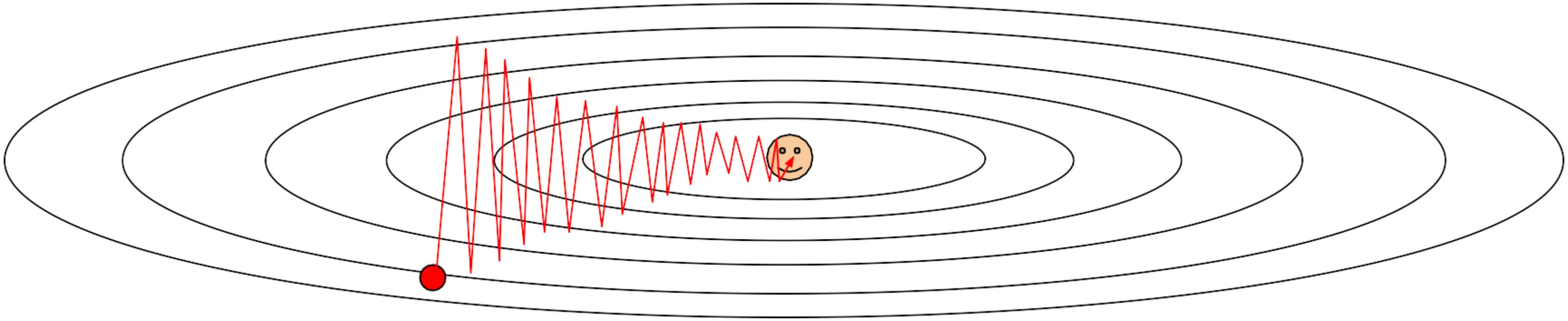
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?



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

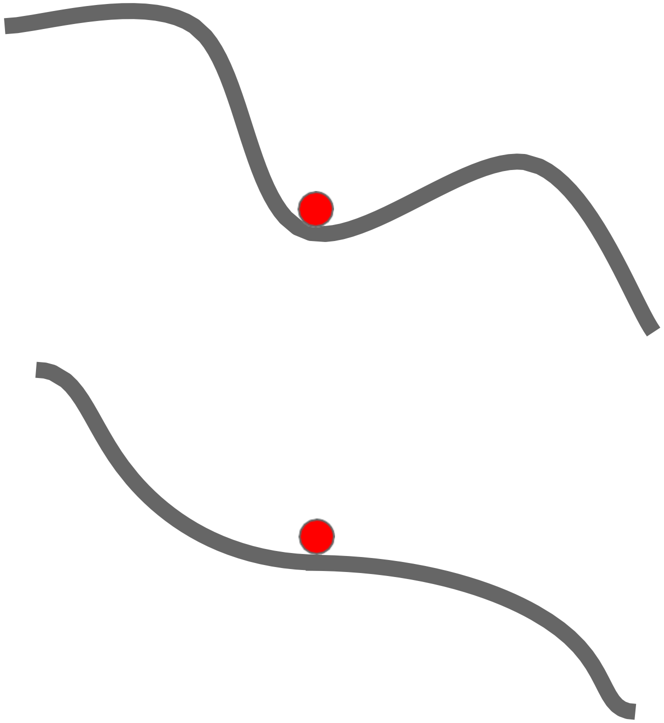
Optimization:
Problems with SGD

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**?

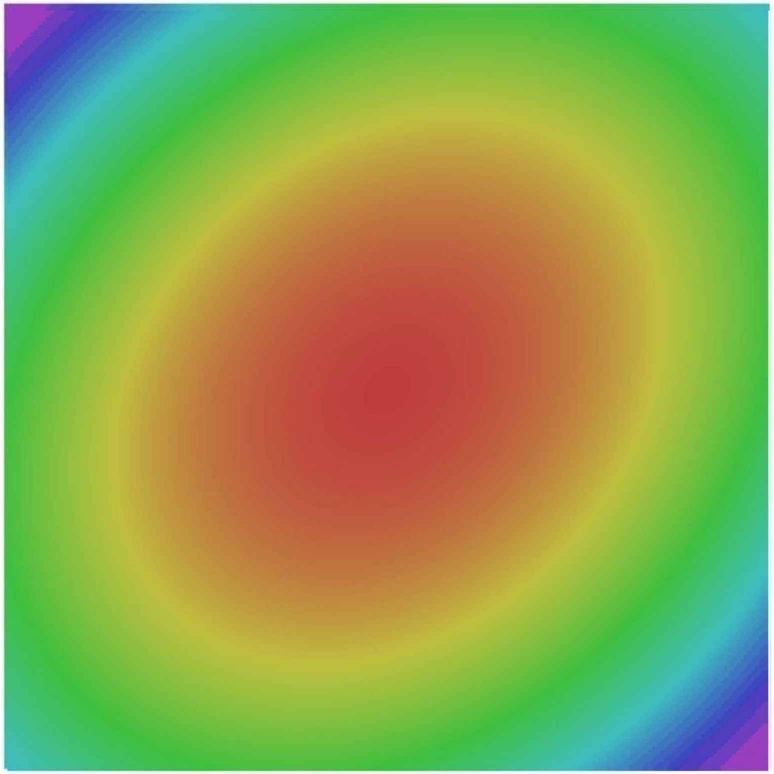
Zero gradient, gradient descent gets stuck

Optimization:
Problems with SGD

Our gradients come from minibatches so they can be noisy!

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

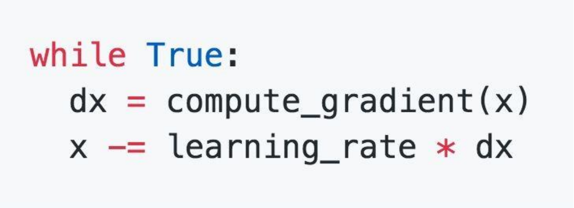
$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



SGD + Momentum

SGD

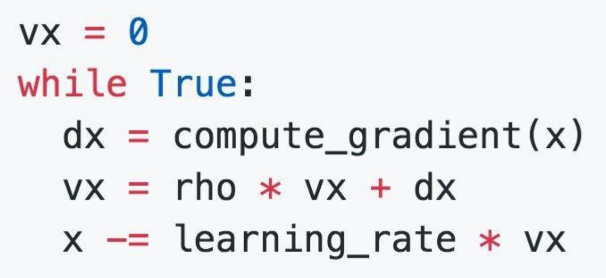
$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$



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

SGD+Momentum

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

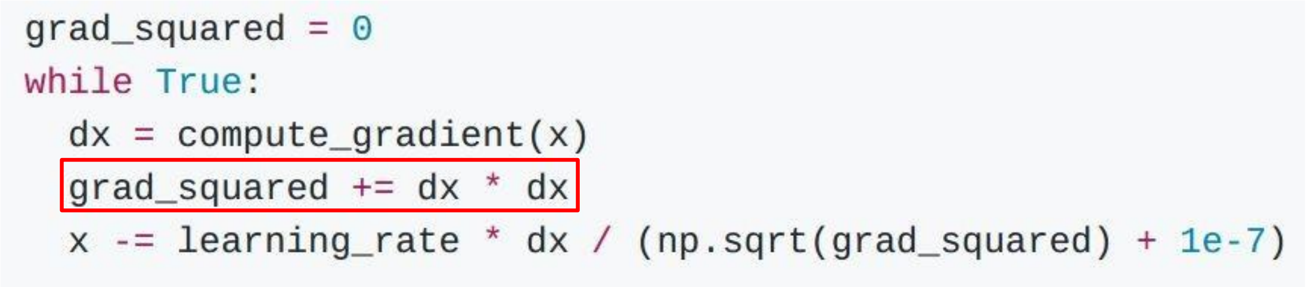


```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- 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)
```

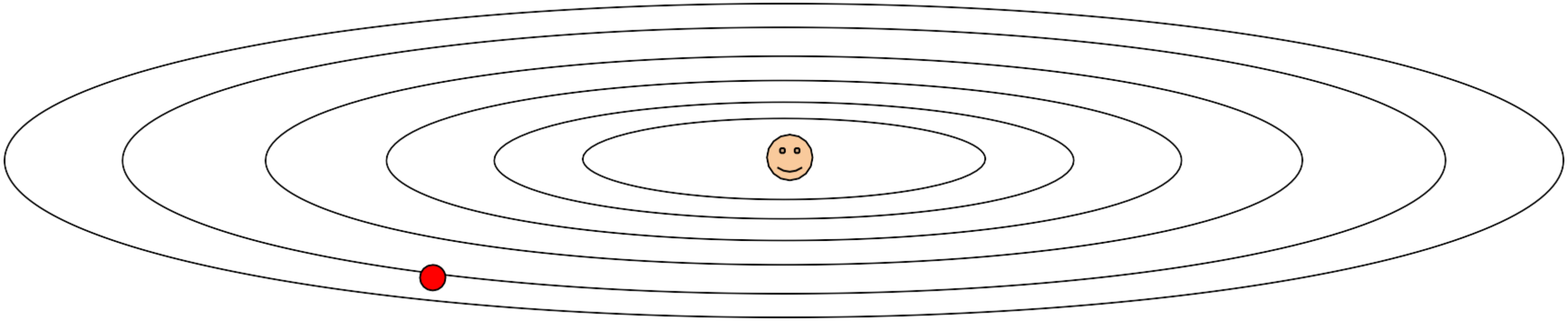
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

“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)
```

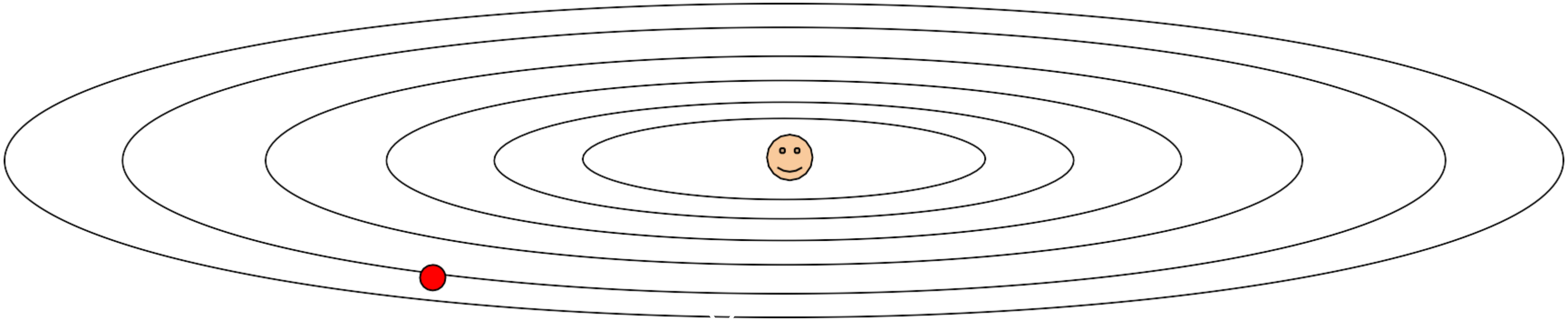


Q: What happens with AdaGrad?

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

AdaGrad

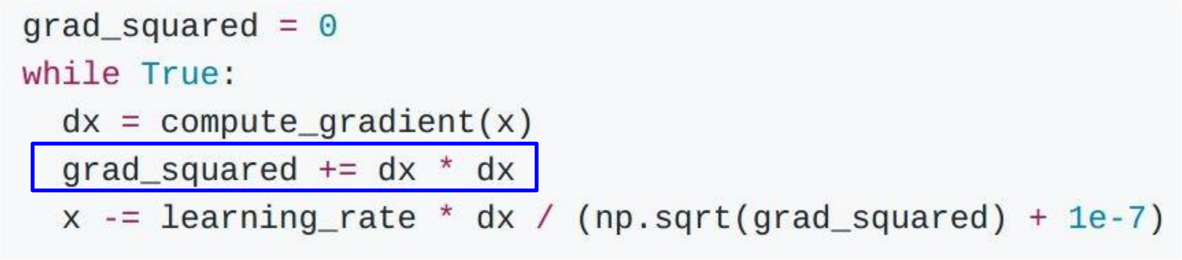
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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

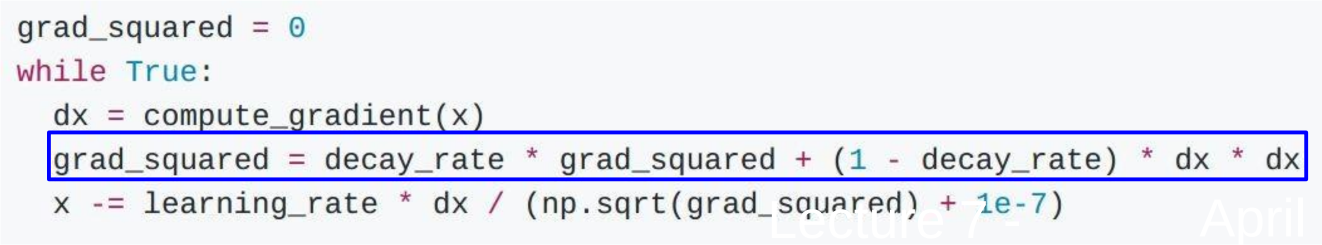
RMSProp

AdaGrad



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

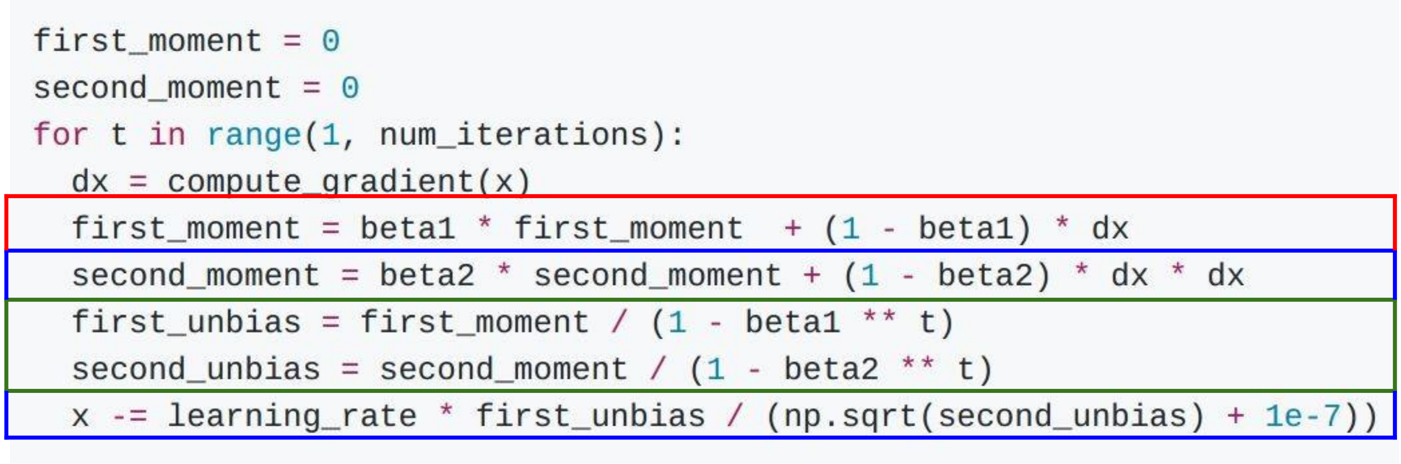
RMSProp



```
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

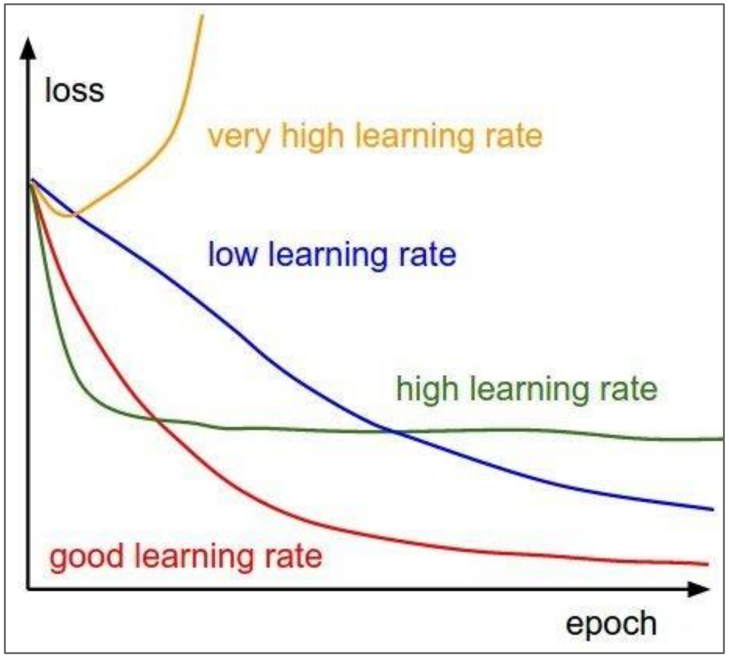
AdaGrad / RMSProp

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

Adam with beta1 = 0.9,
beta2 = 0.999, and learning_rate = 1e-3 or 5e-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.

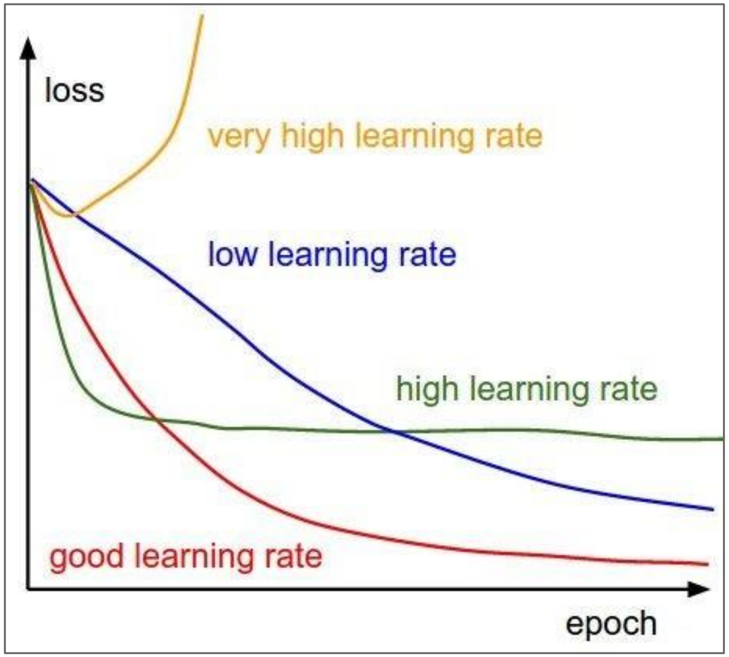
exponential decay:

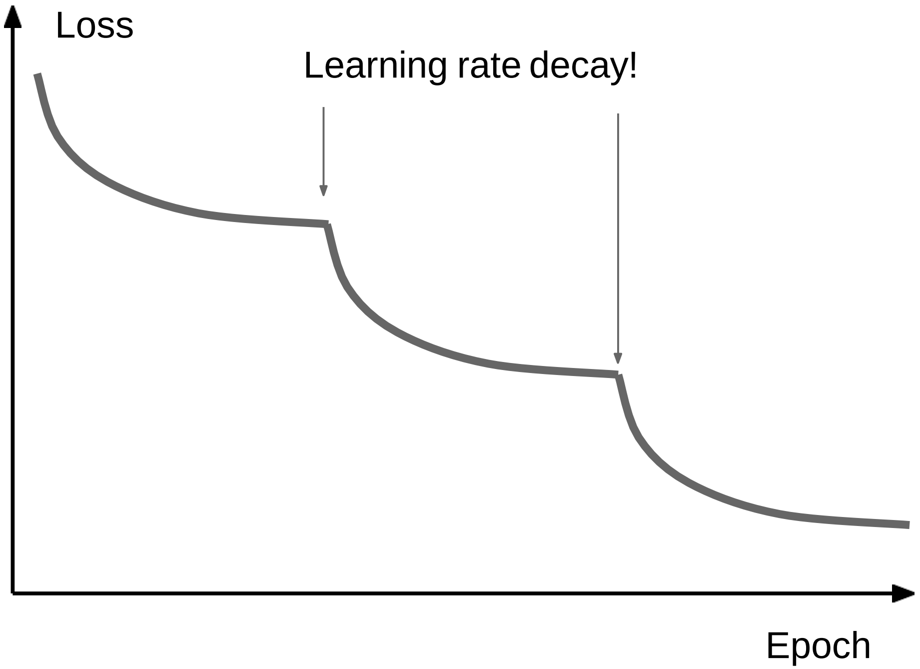
$$\alpha = \alpha_0 e^{-kt}$$

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

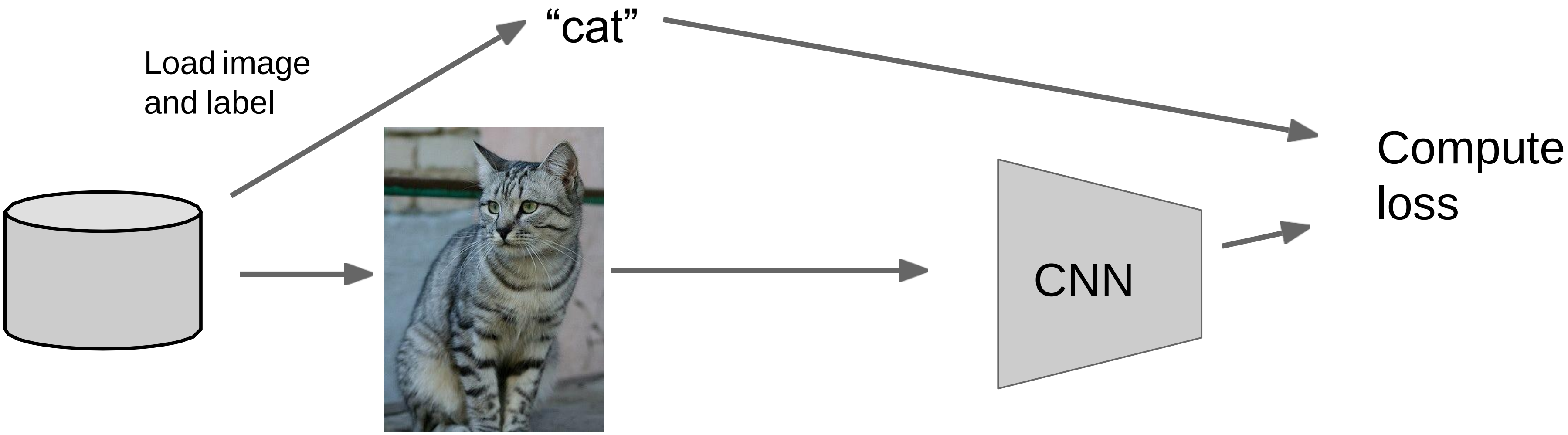
Practical matters

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

Understanding CNNs

- Visualization
- Breaking CNNs

Data Augmentation



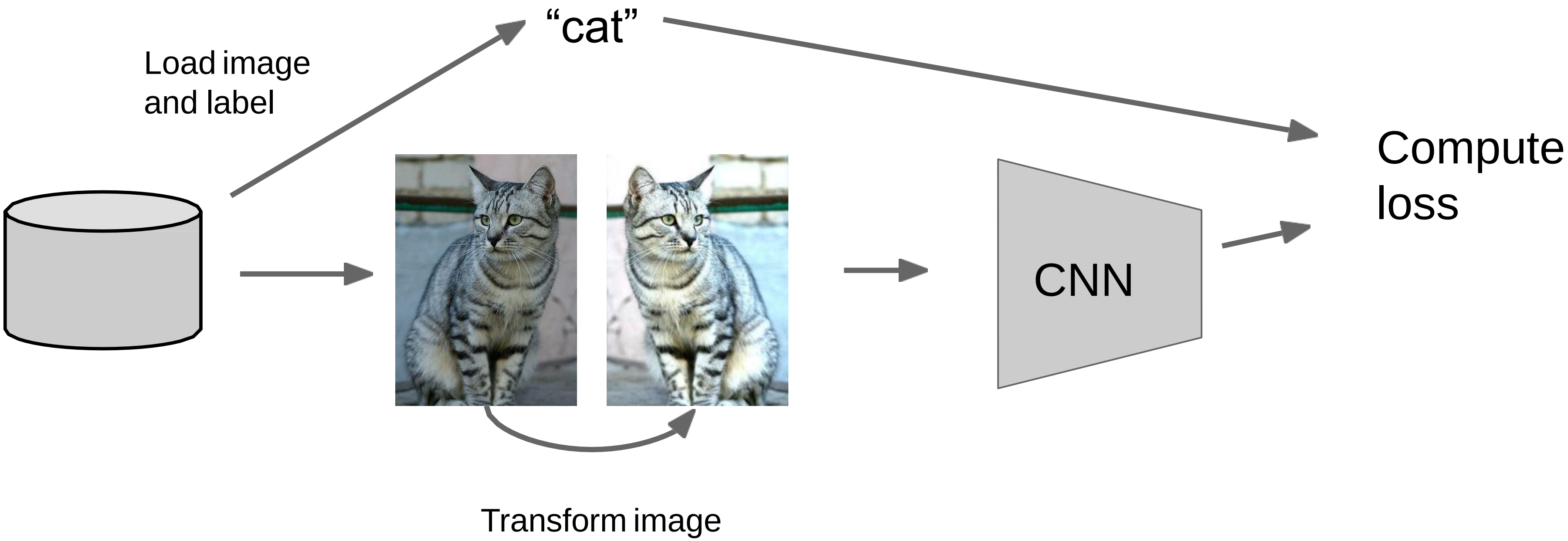
Load image and label

“cat”

CNN

Compute loss

Data Augmentation



Load image and label

“cat”

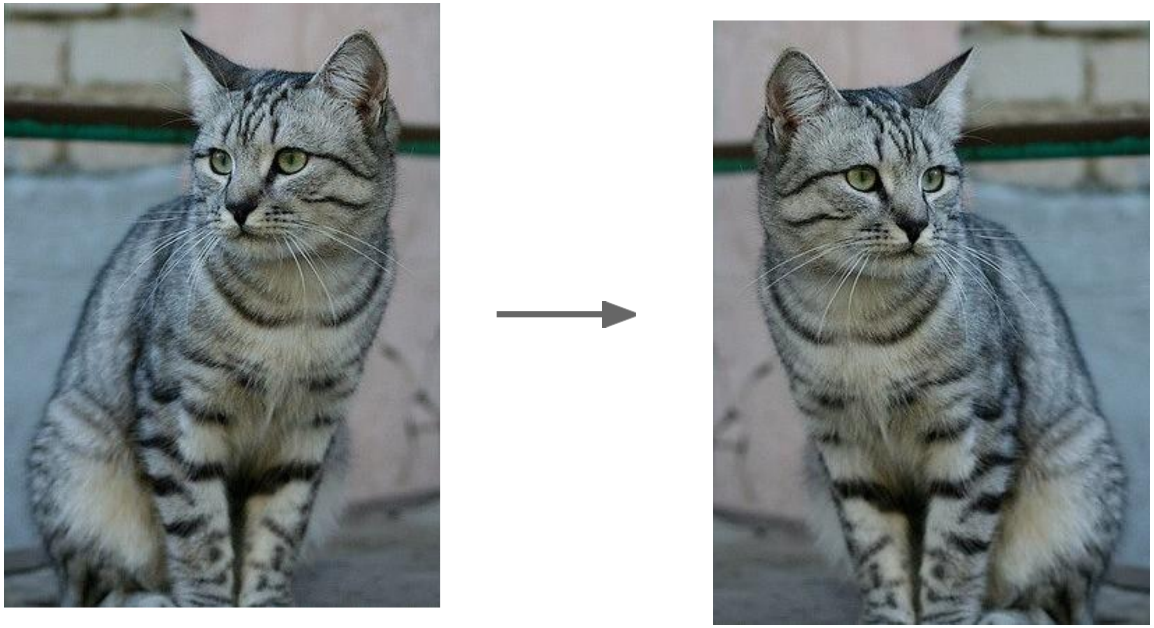
Compute loss

CNN

Transform image

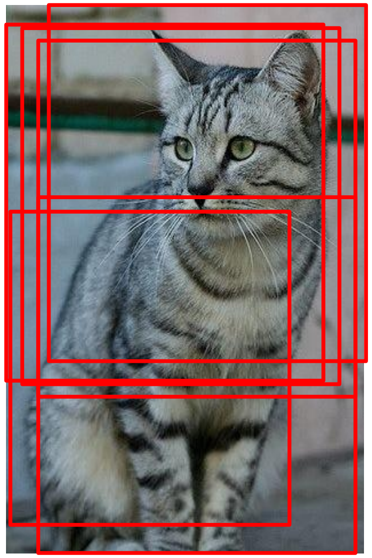
Data Augmentation

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 x 224 patch

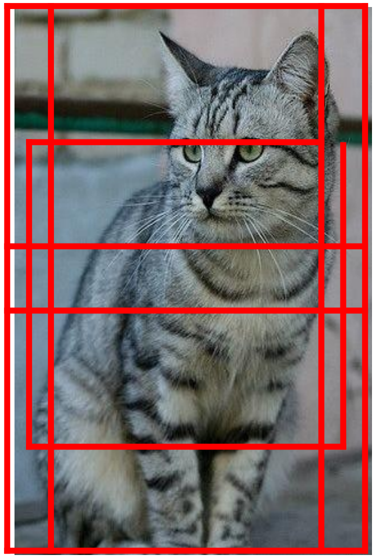
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

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



Testing: average a fixed set of crops

ResNet:

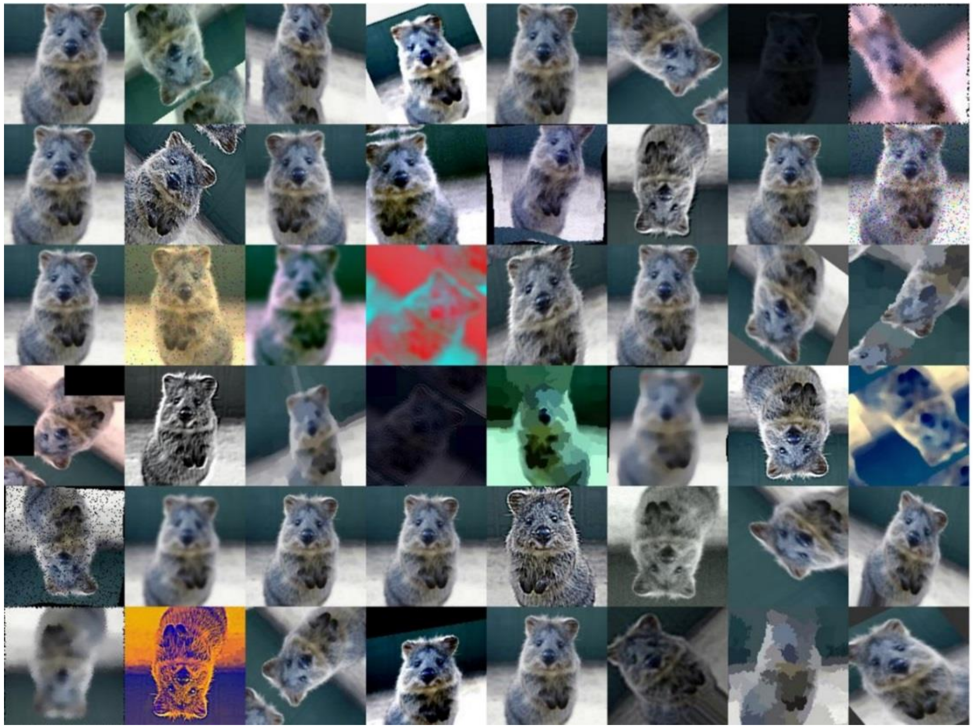
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data Augmentation

Get creative for your problem!

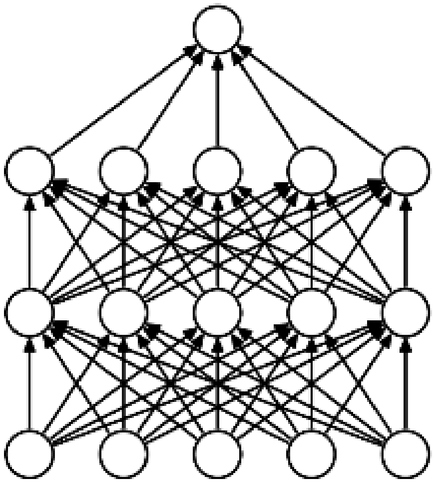
Random mix/combinations of :

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

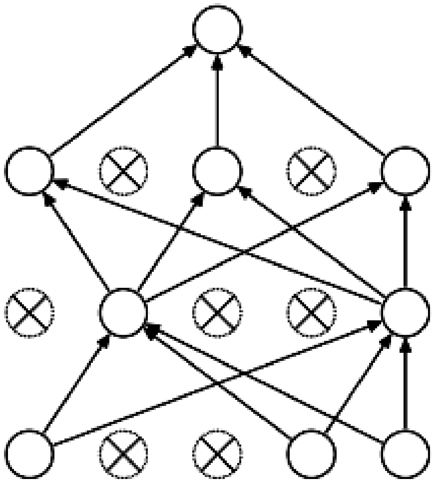


Fei-Fei Li, Andrej Karpathy, Justin Johnson, Serena Yeung; Image: https://github.com/aleju/imgaug

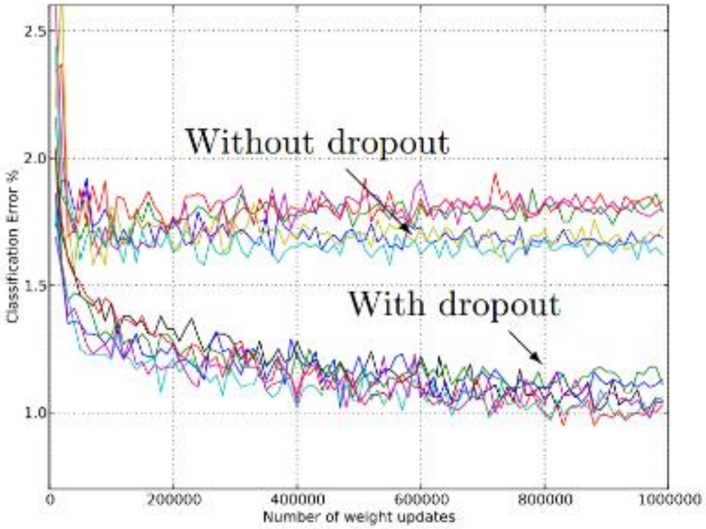
Regularization: Dropout



(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]

Adapted from Jia-bin Huang

Transfer Learning

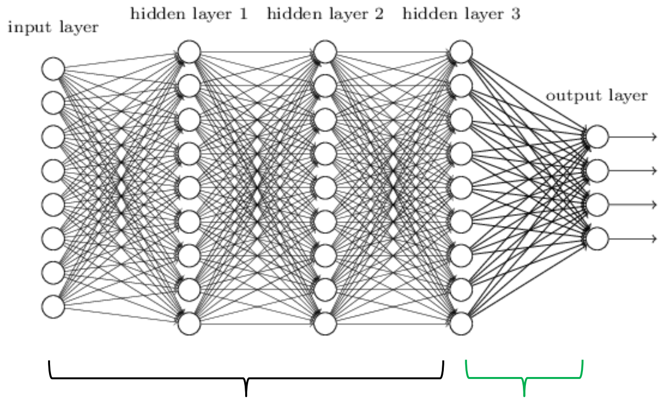
"You need a lot of a data if you want to train/use CNNs"

BUSTED

Andrej Karpathy

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:



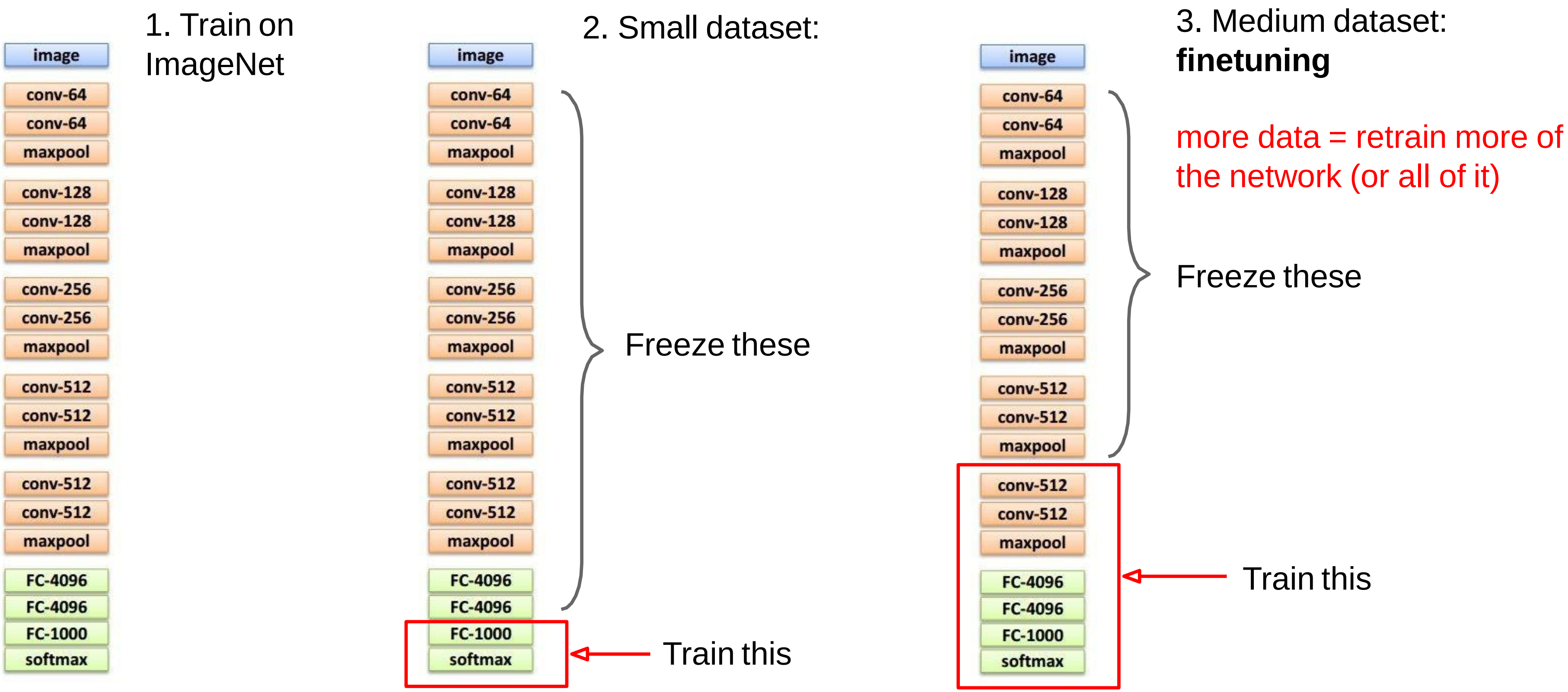
Set these to the already learned weights from another network

Learn these on your own task

Transfer Learning with CNNs

Source: classification on ImageNet

Target: classification on Places



1. Train on ImageNet

2. Small dataset:

Freeze these

Train this

3. Medium dataset: **finetuning**

more data = retrain more of the network (or all of it)

Freeze these

Train this

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

Adapted from Andrej Karpathy

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

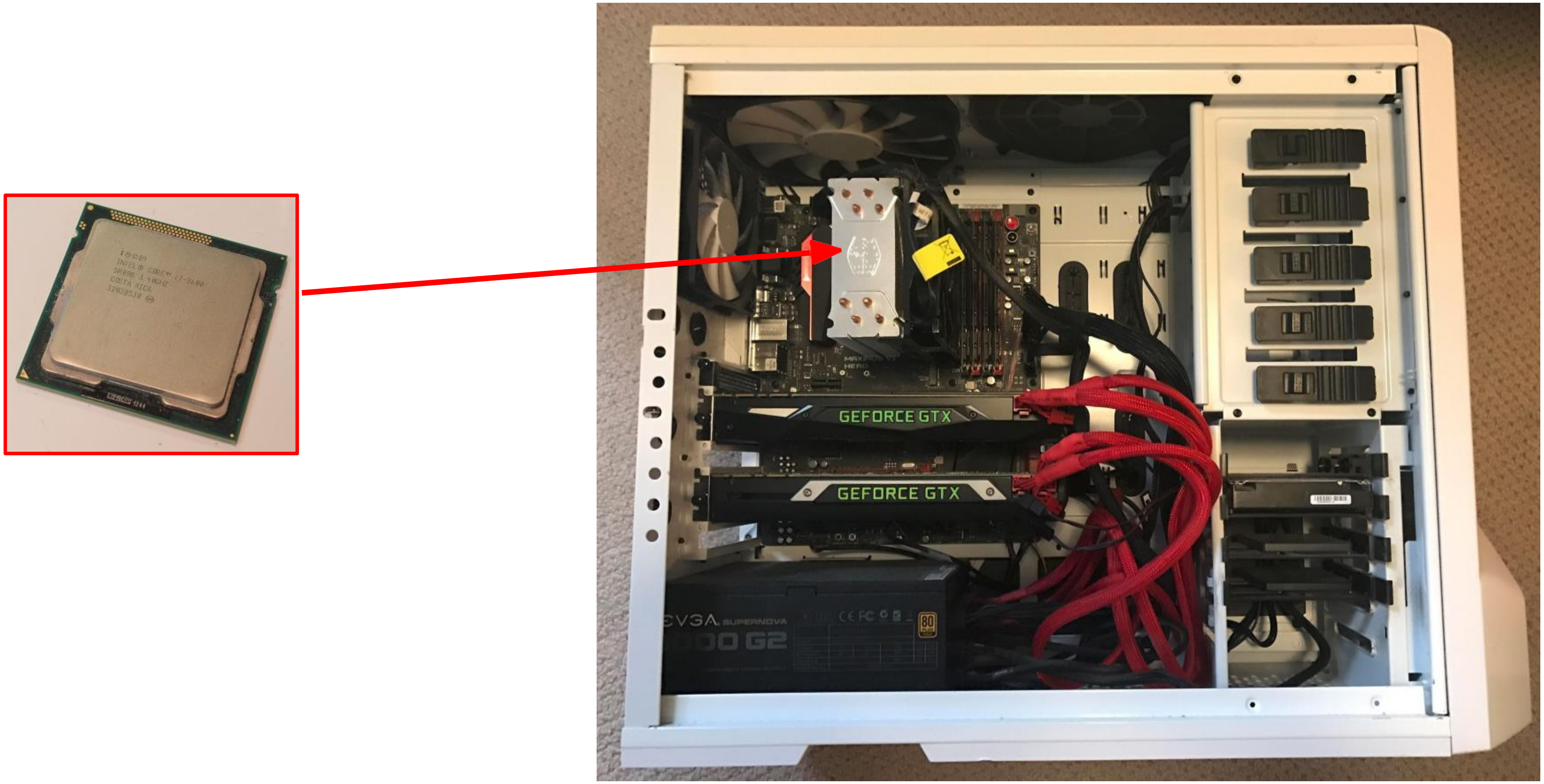
- Getting started: Preprocessing, initialization, optimization, normalization
- Improving performance: regularization, augmentation, transfer
- Hardware and software

Understanding CNNs

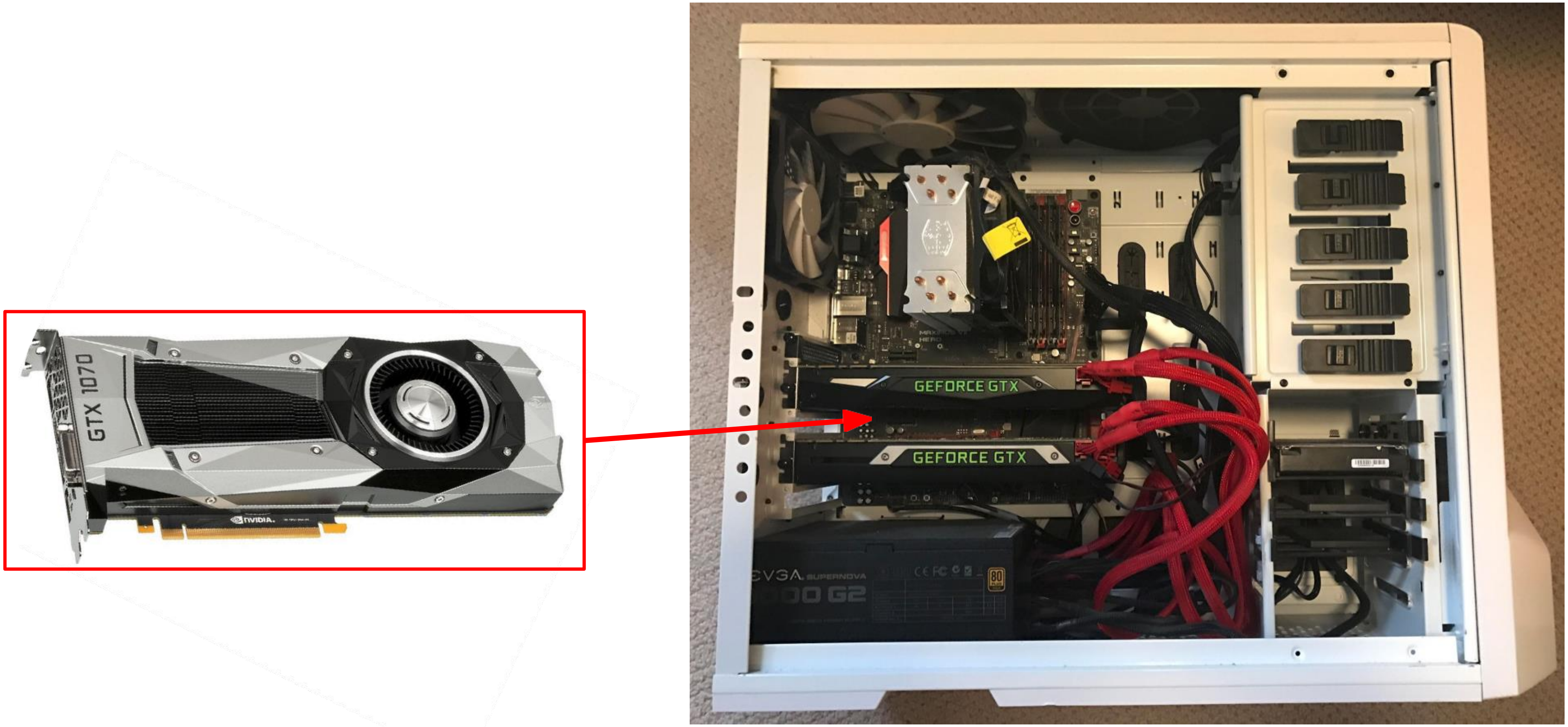
- Visualization
- Breaking CNNs

Hardware and software

Spot the CPU! (central processing unit)



Spot the GPUs! (graphics processing unit)



CPU vs GPU

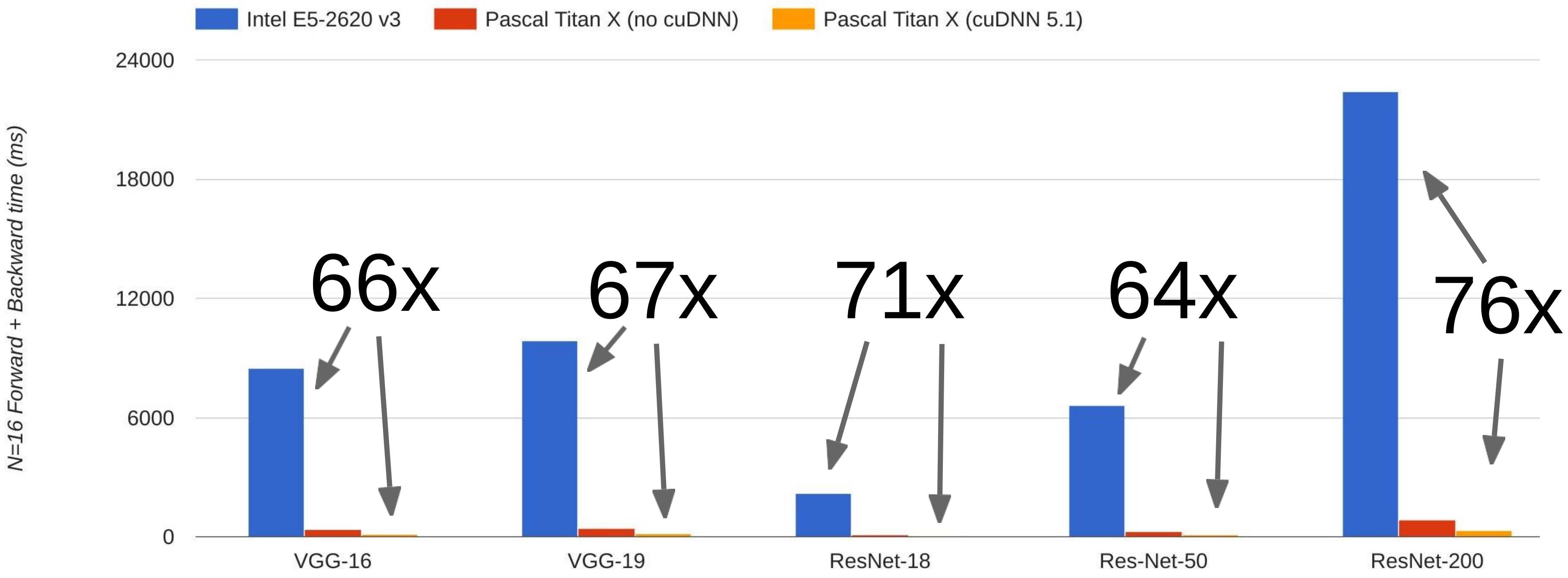
	Cores	**Clock Speed**	**Memory**	**Price**	**Speed**
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	$699	~11.4 TFLOPs FP32

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

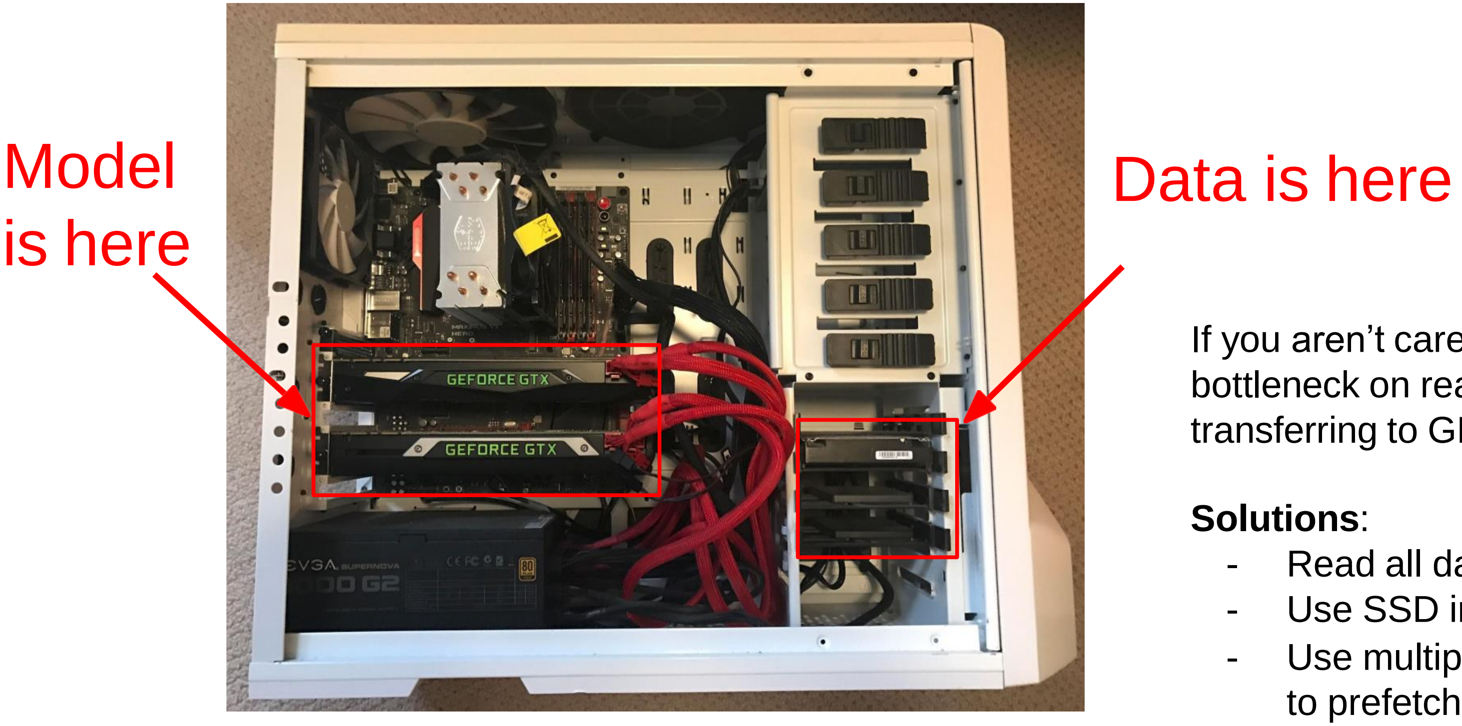
CPU vs GPU in practice

(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!

Caffe (UC Berkeley) → Caffe2 (Facebook)

Torch (NYU / Facebook) → PyTorch (Facebook)

Theano (U Montreal) → TensorFlow (Google)

PaddlePaddle
(Baidu)

Chainer

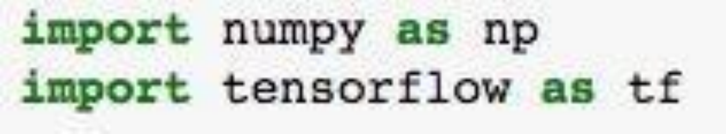
MXNet
(Amazon)

CNTK
(Microsoft)

Deeplearning4j

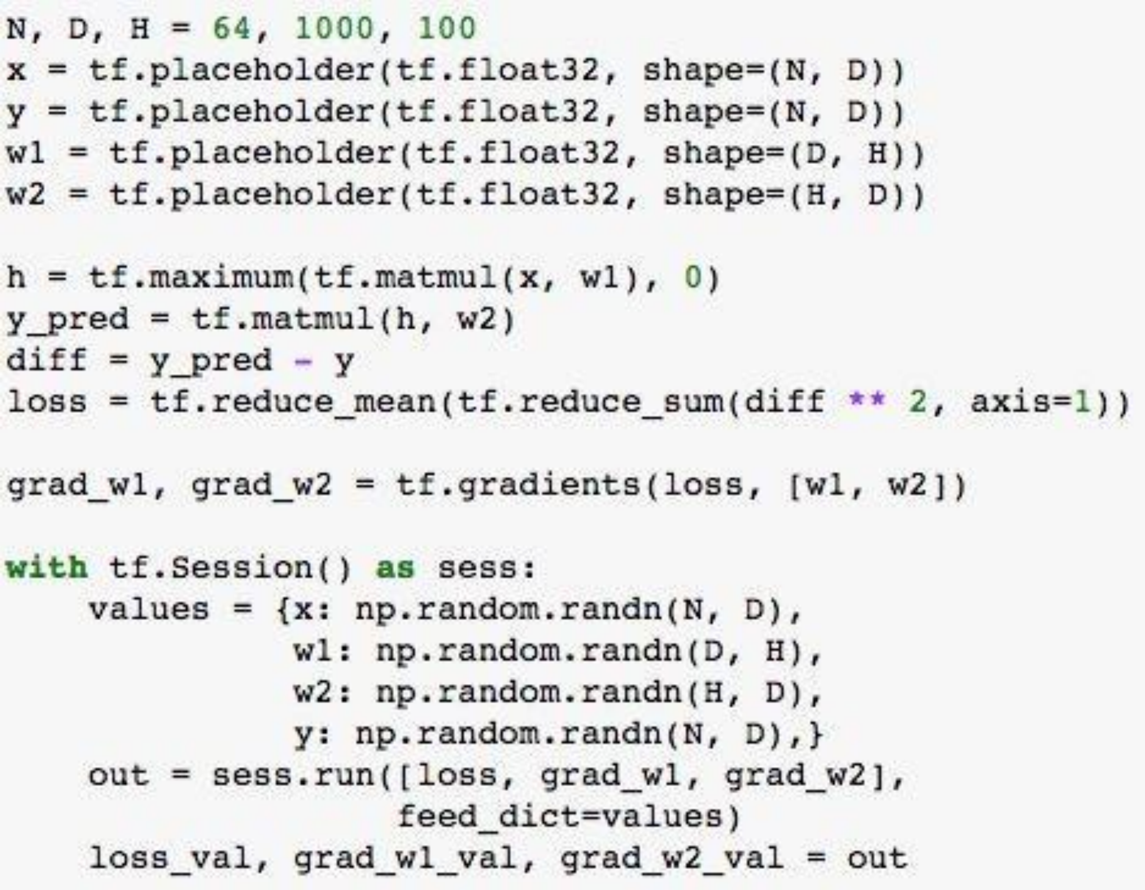
And others...

TensorFlow: Neural Net



```
import numpy as np
import tensorflow as tf
```

(Assume imports at the top of each snipppet)

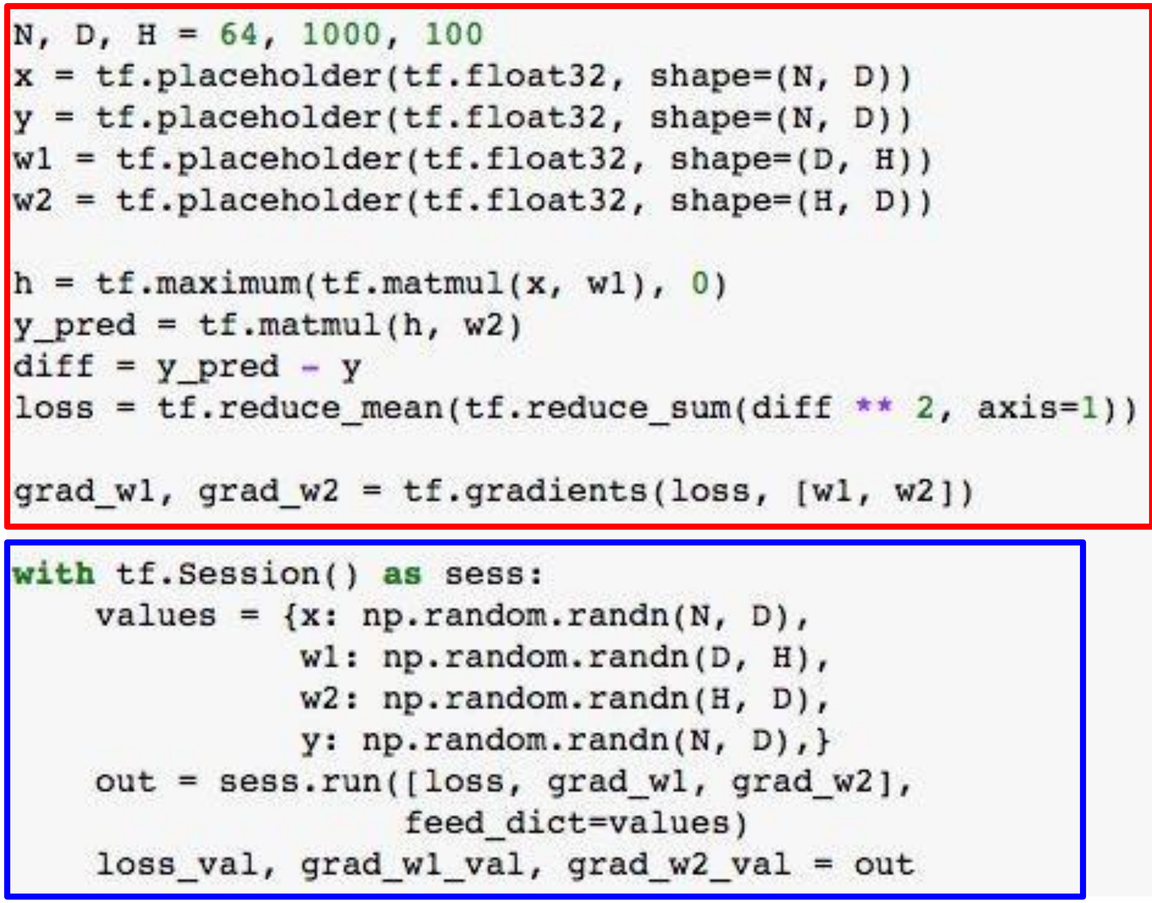


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

First **define** computational graph

Then **run** the graph many times

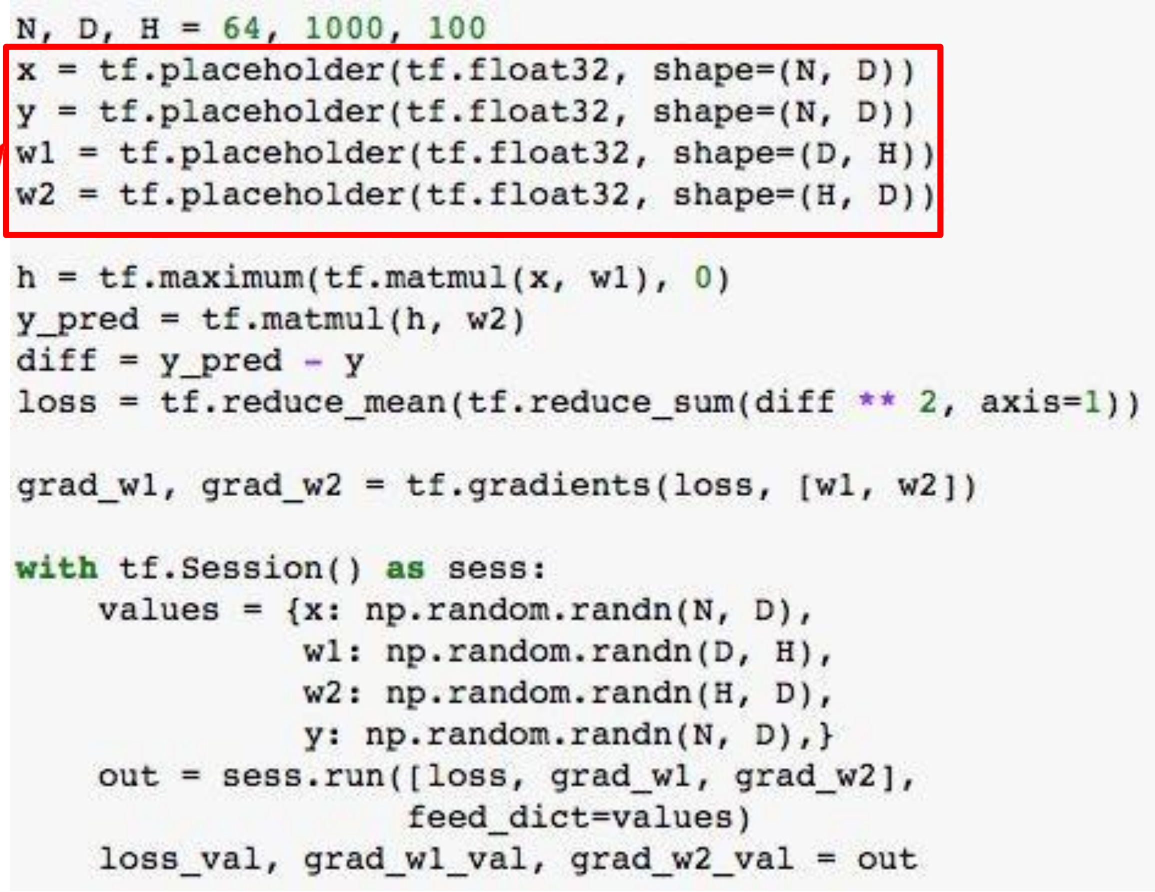


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

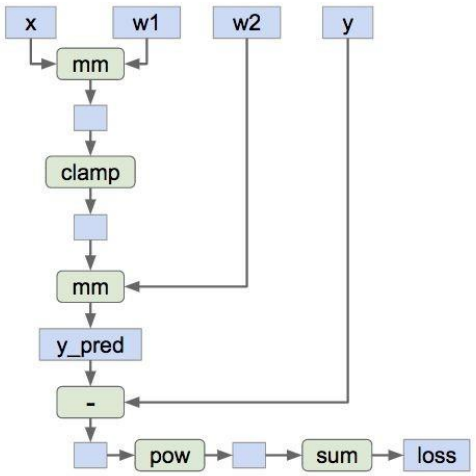
TensorFlow: Neural Net

Create **placeholders** for input x, weights w1 and w2, and targets y

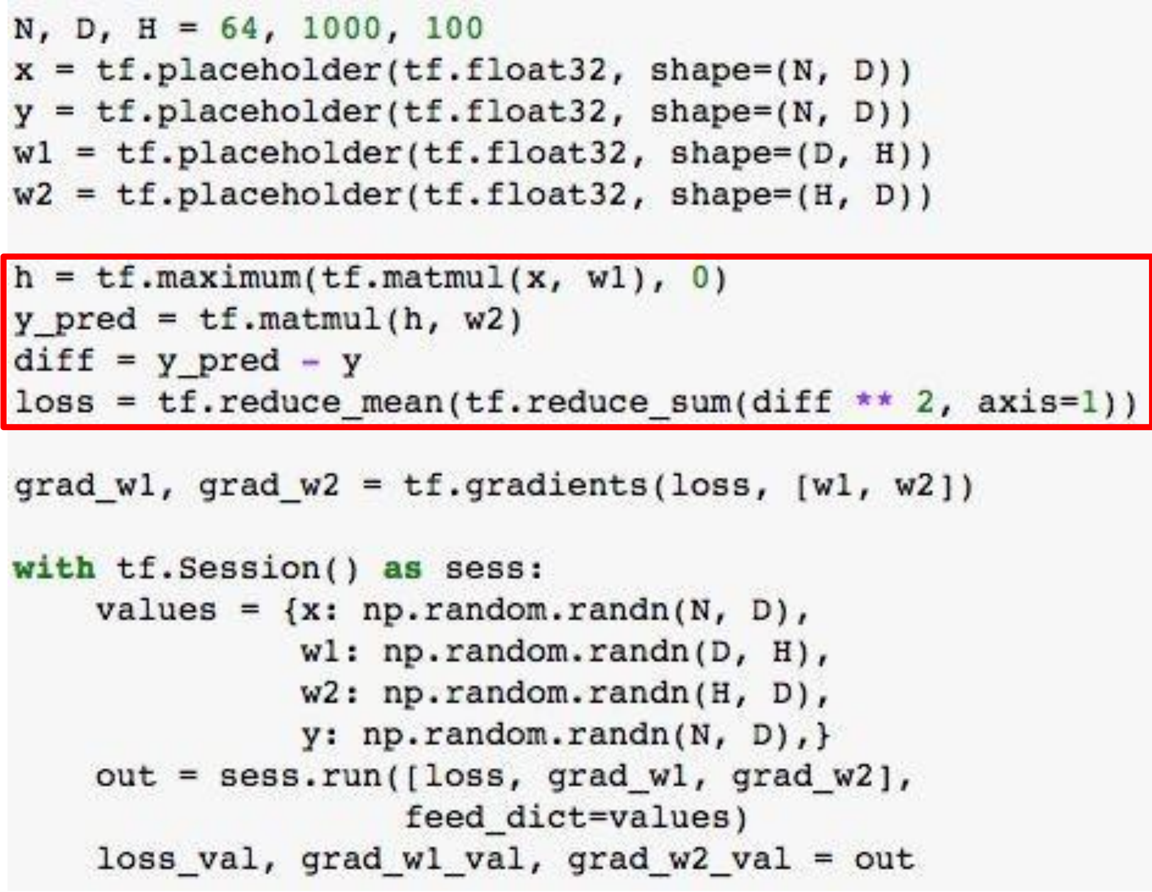


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

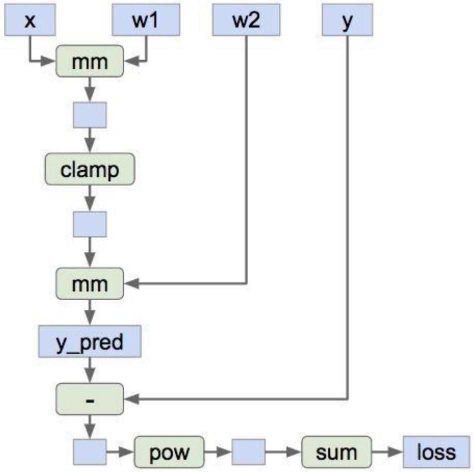


Forward pass: compute prediction for y and loss. No computation - just building graph

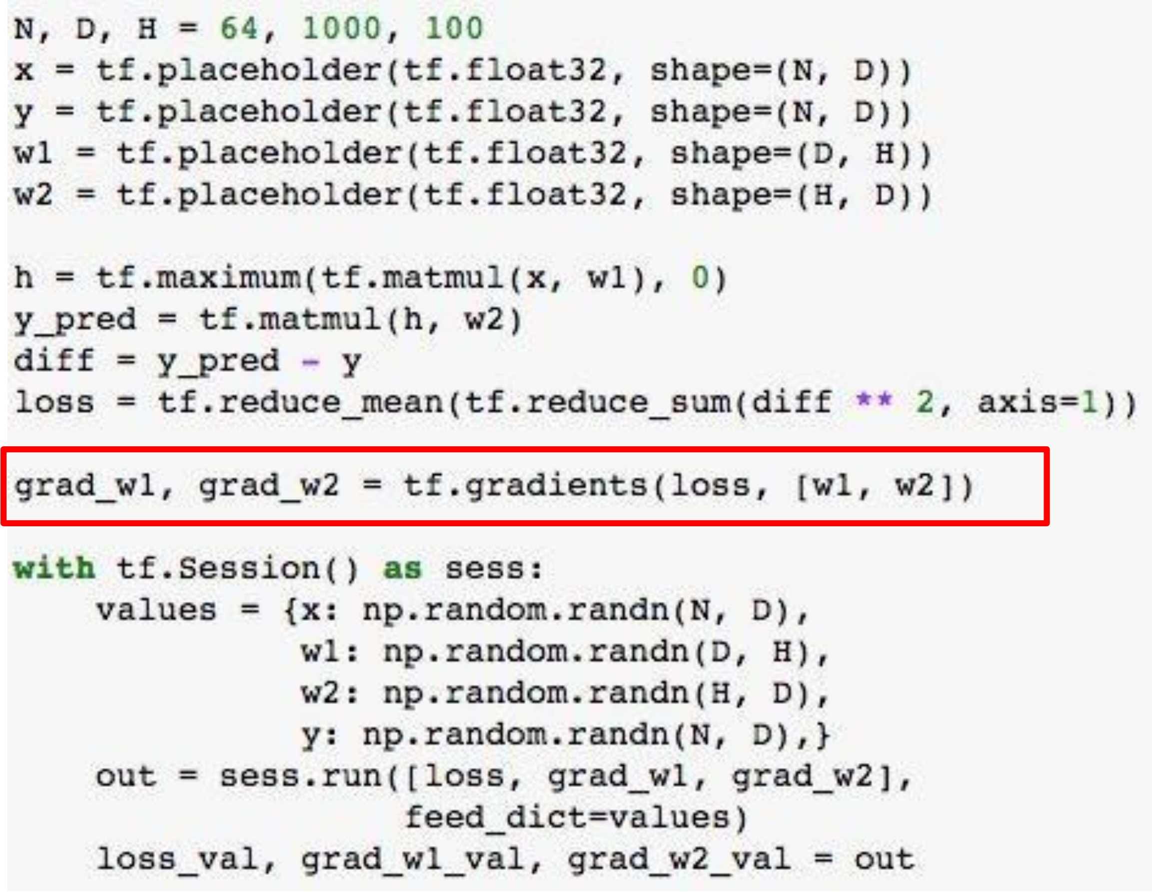


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

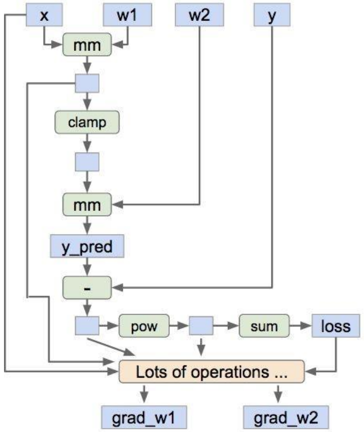


Tell TensorFlow to compute loss of gradient with respect to w1 and w2. No compute - just building the graph

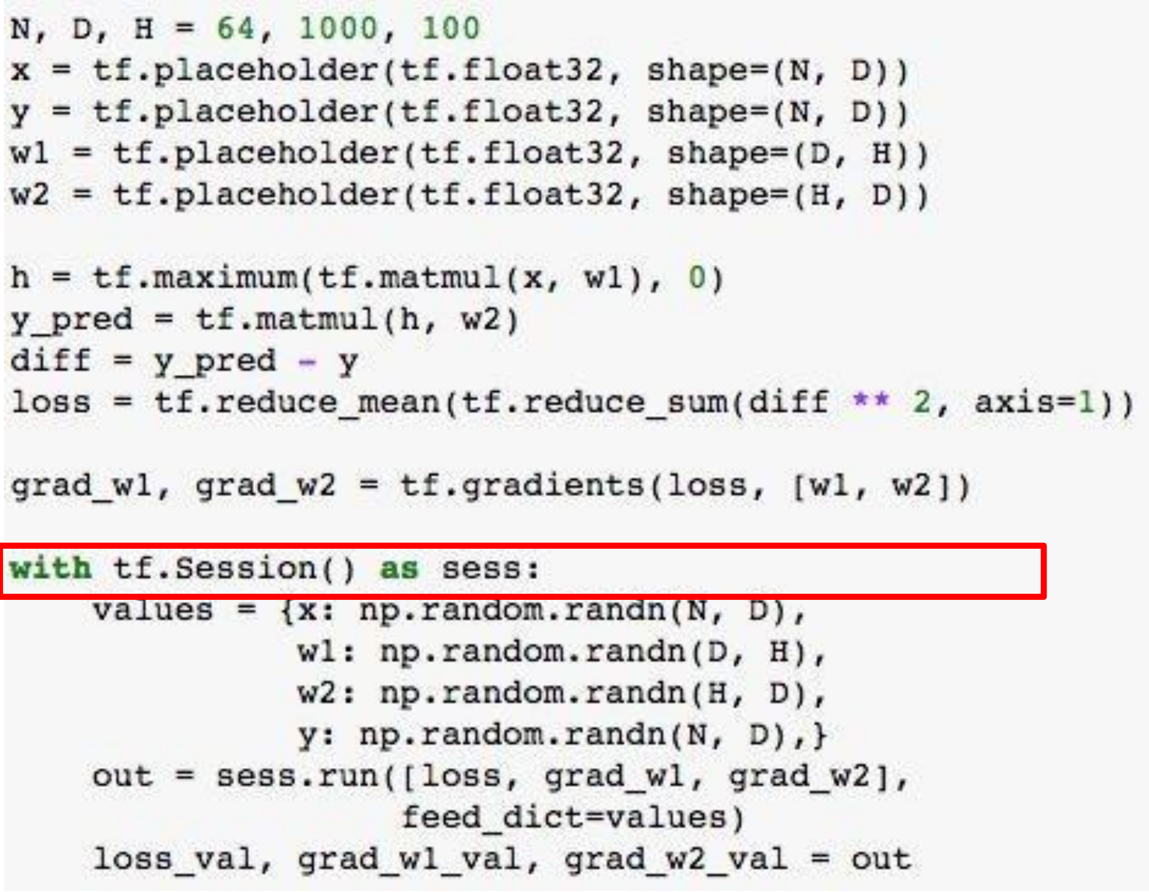


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

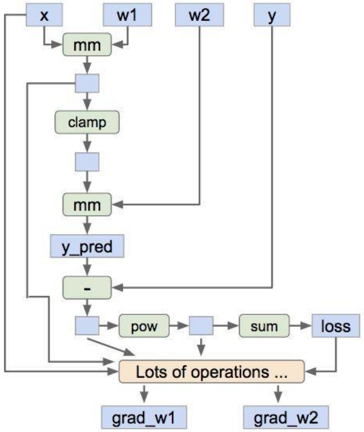


Now done building our graph, so we enter a **session** so we can actually run the graph

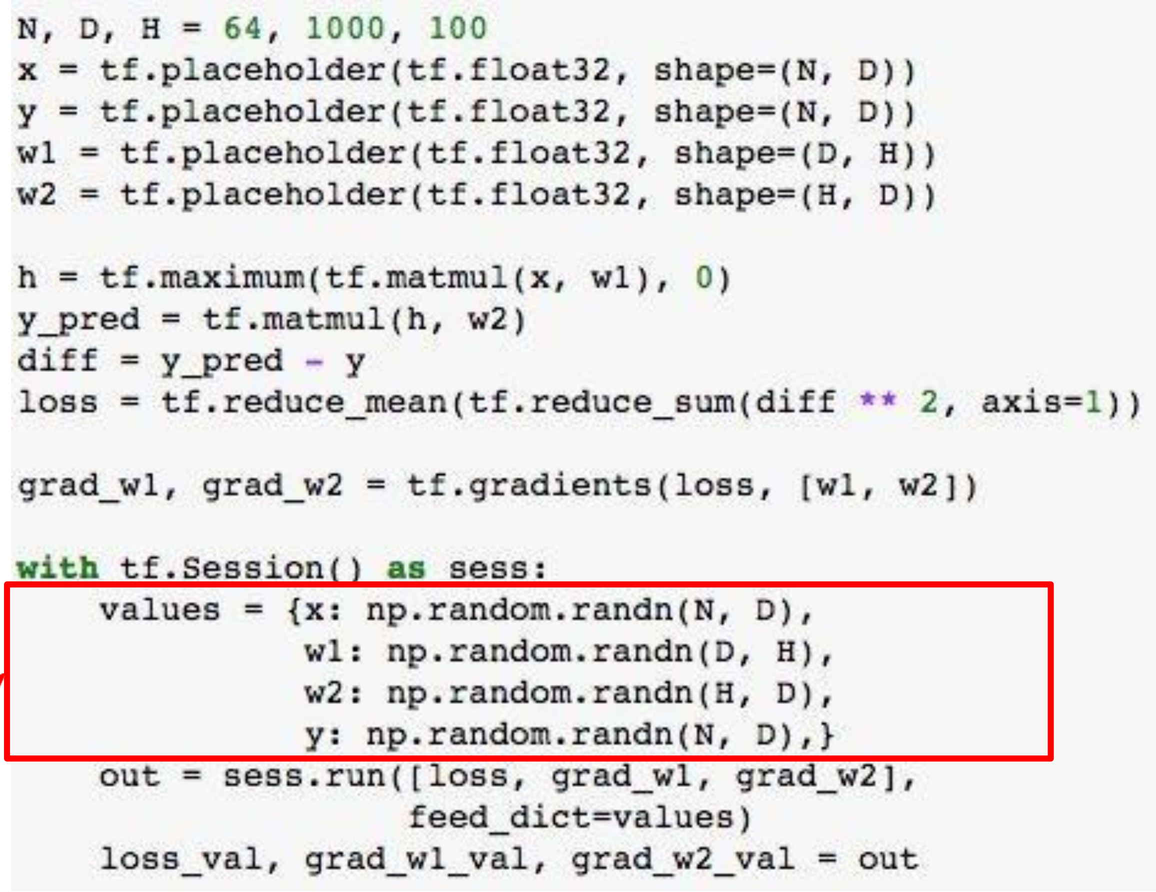


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

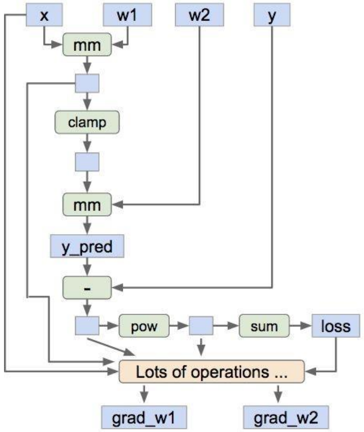


Create numpy arrays that will fill in the placeholders above

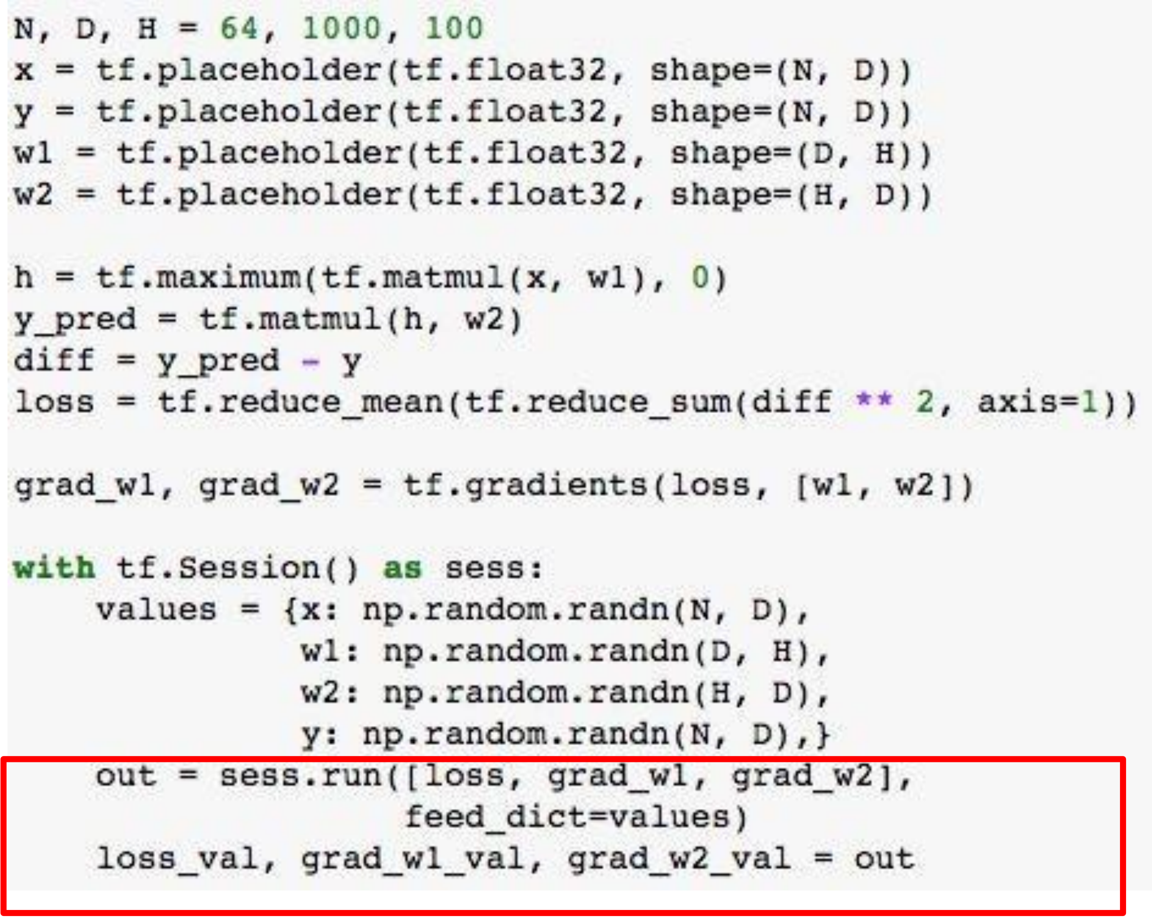


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

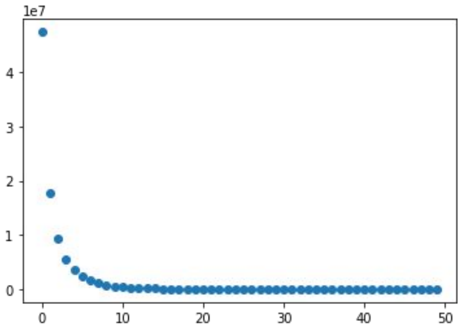


Run the graph: feed in the numpy arrays for x, y, w1, and w2; get numpy arrays for loss, grad_w1, and grad_w2

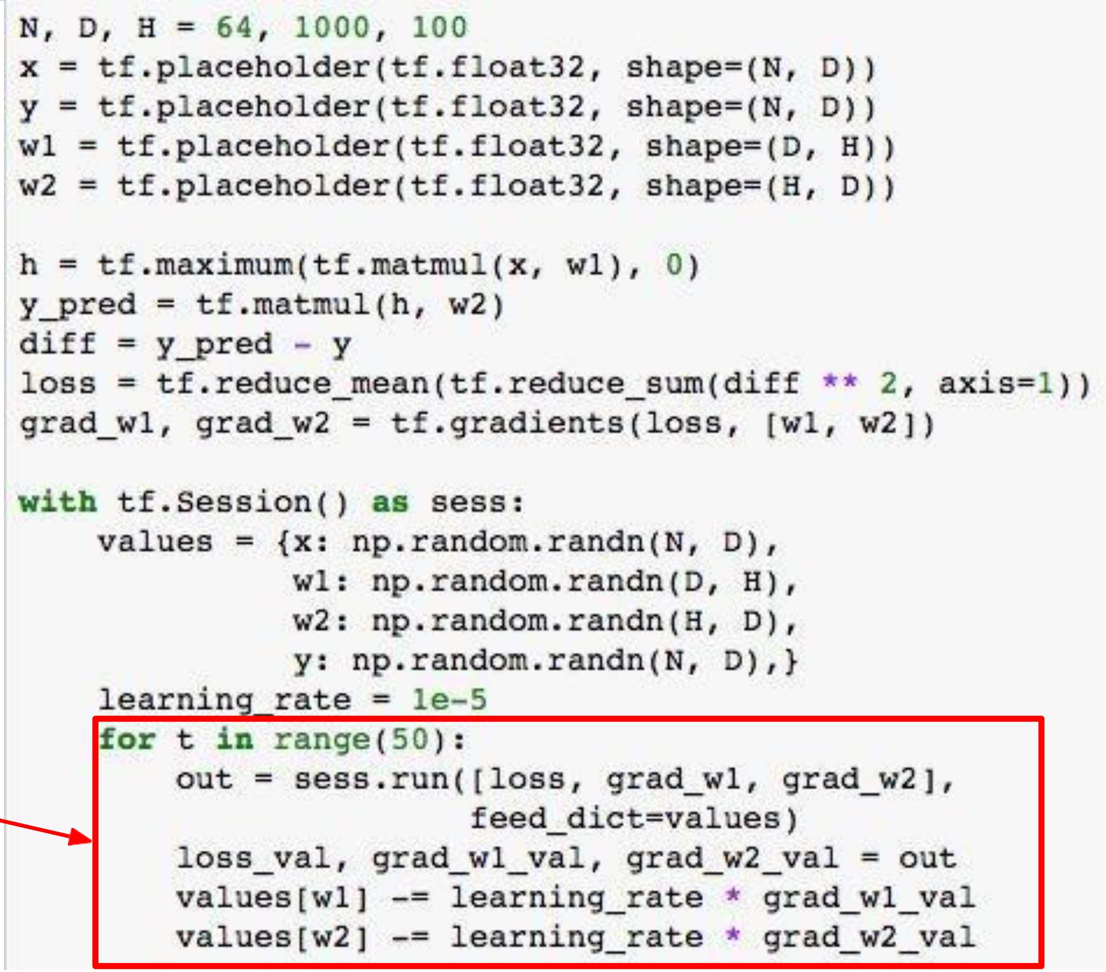


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Train the network: Run the graph over and over, use gradient to update weights

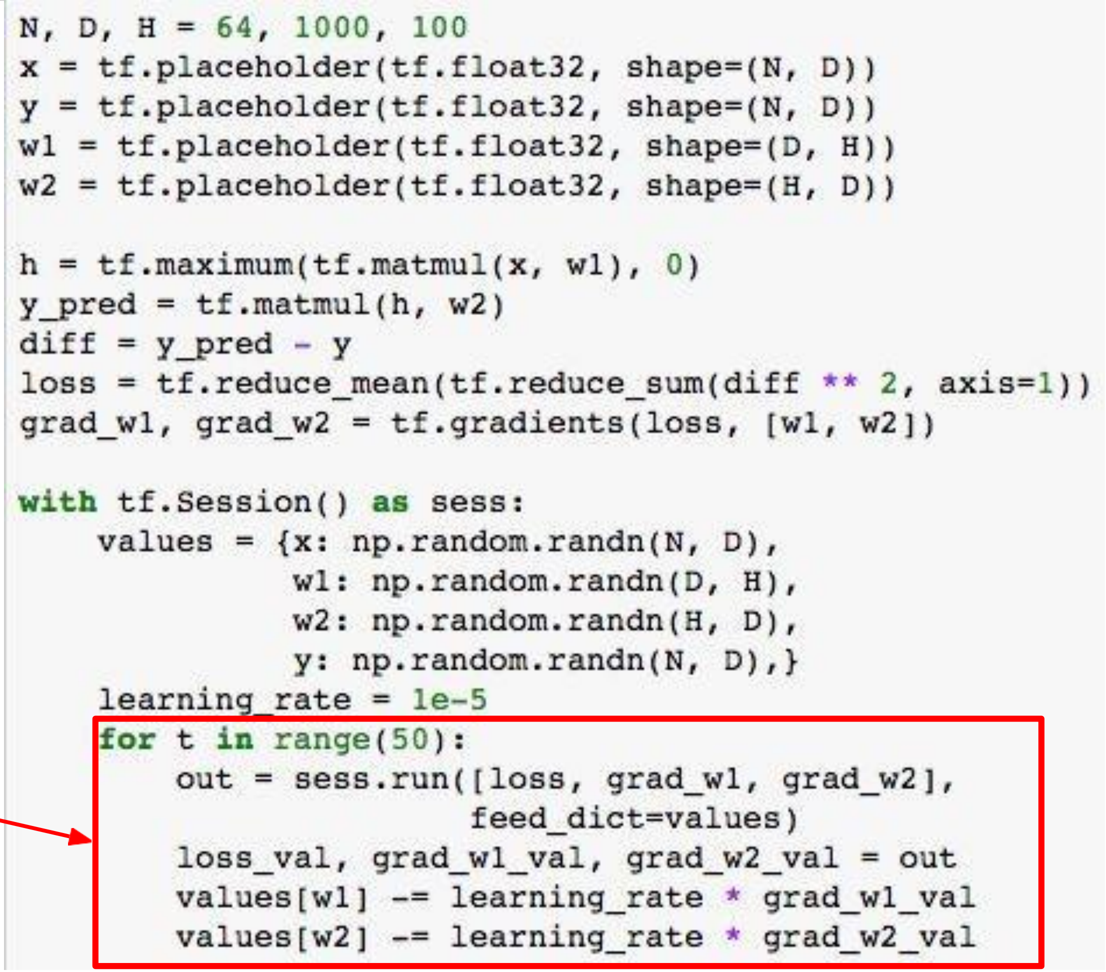


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    learning_rate = 1e-5
    for t in range(50):
        out = sess.run([loss, grad_w1, grad_w2],
                       feed_dict=values)
        loss_val, grad_w1_val, grad_w2_val = out
        values[w1] -= learning_rate * grad_w1_val
        values[w2] -= learning_rate * grad_w2_val
```

TensorFlow: Neural Net

Problem: copying weights between CPU / GPU each step

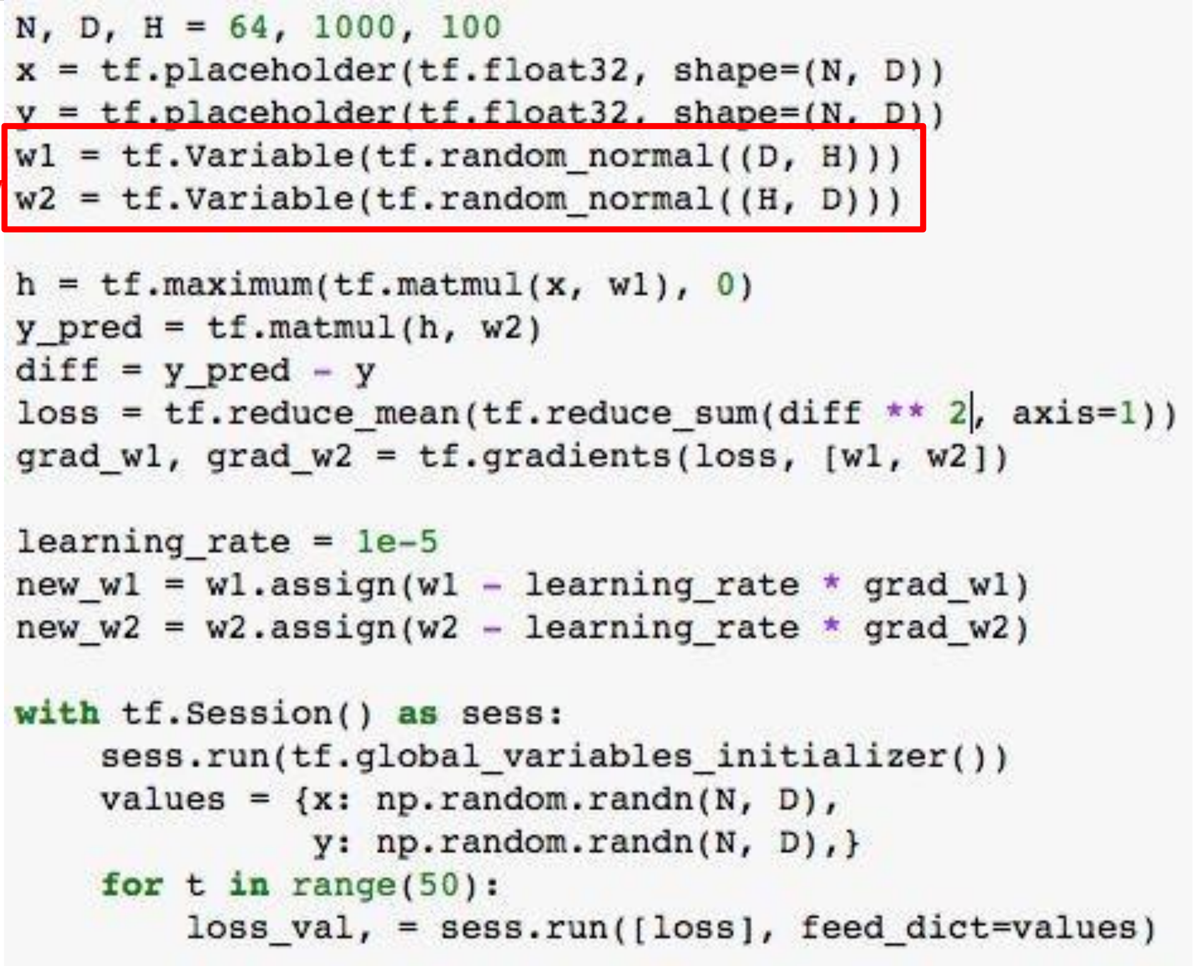
Train the network: Run the graph over and over, use gradient to update weights



```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
             y: np.random.randn(N, D),}
    learning_rate = 1e-5
    for t in range(50):
        out = sess.run([loss, grad_w1, grad_w2],
                       feed_dict=values)
        loss_val, grad_w1_val, grad_w2_val = out
        values[w1] -= learning_rate * grad_w1_val
        values[w2] -= learning_rate * grad_w2_val
```

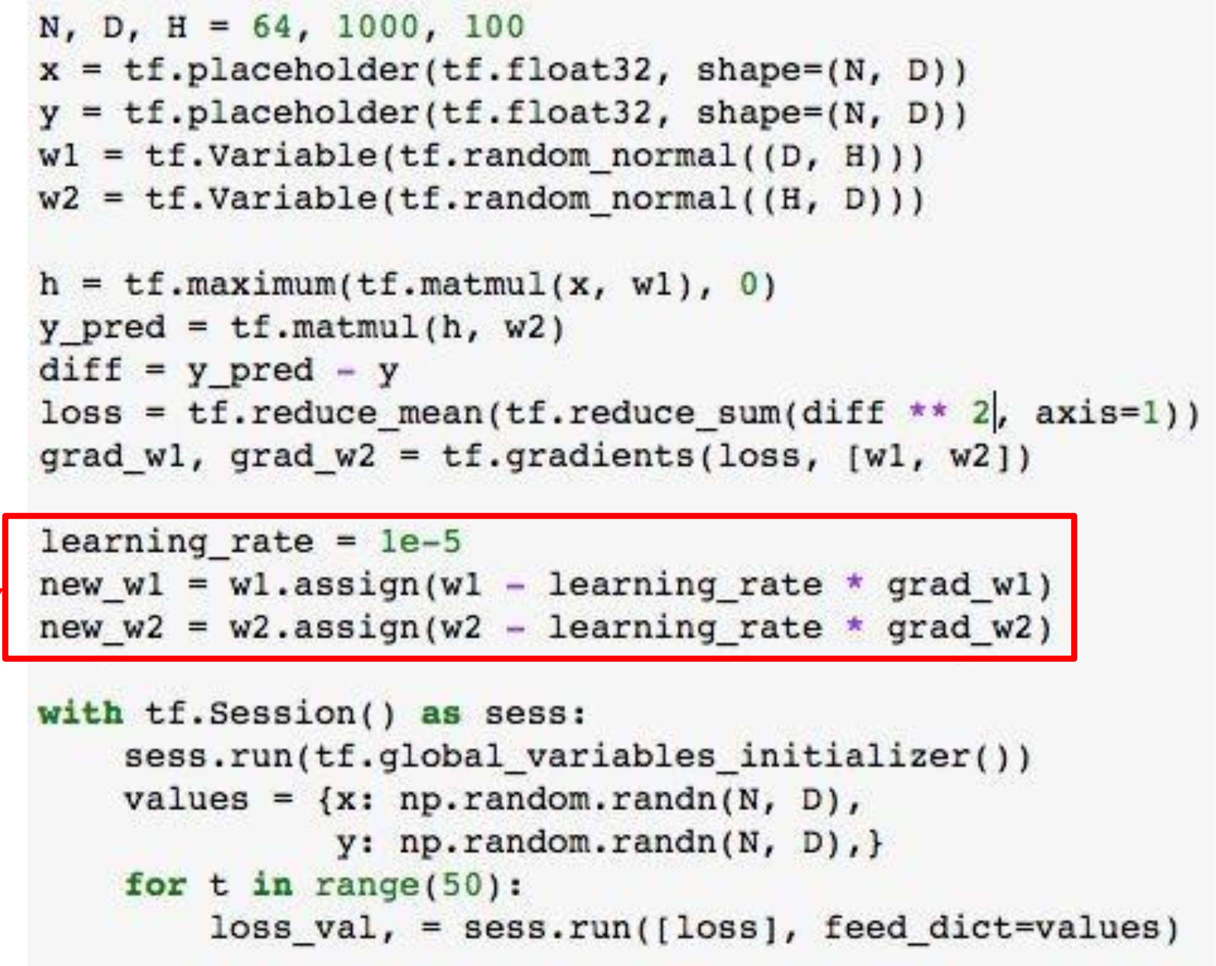
TensorFlow: Neural Net

Change w1 and w2 from **placeholder** (fed on each call) to **Variable** (persists in the graph between calls)



```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

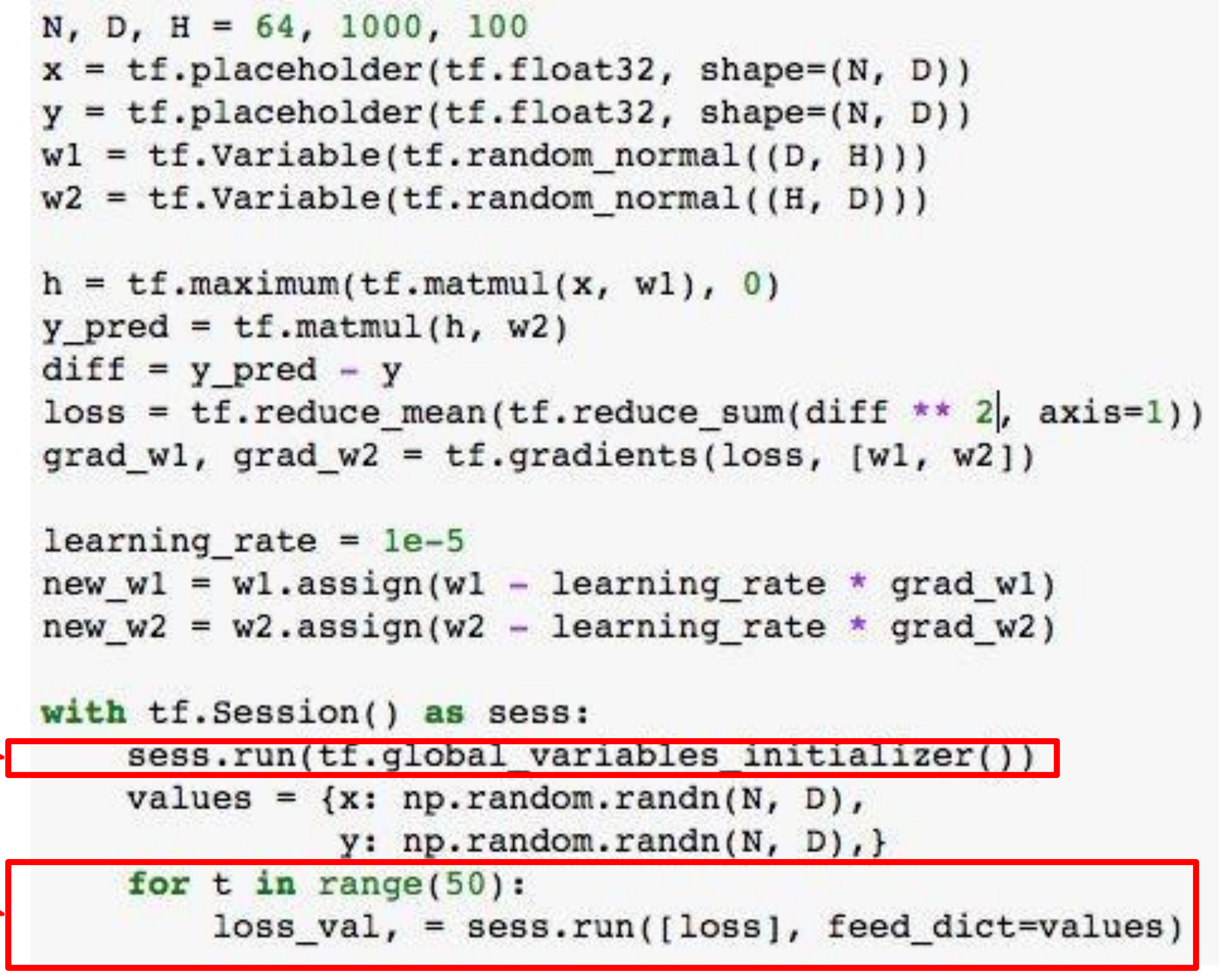
TensorFlow: Neural Net



```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

Add **assign** operations to update w1 and w2 as part of the graph!

TensorFlow: Neural Net

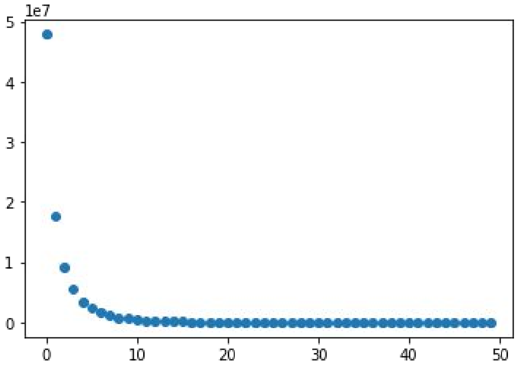


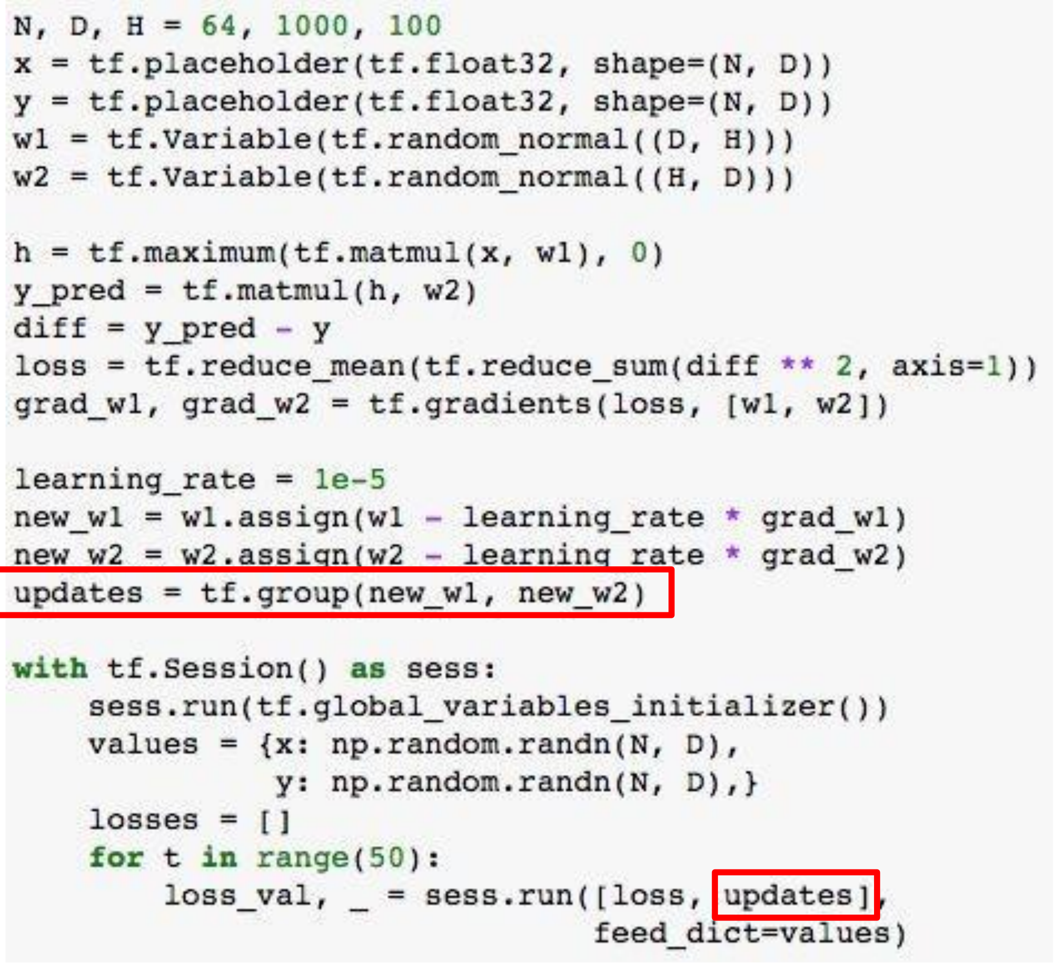
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

Run graph once to initialize w1 and w2

Run many times to train

TensorFlow: Neural Net



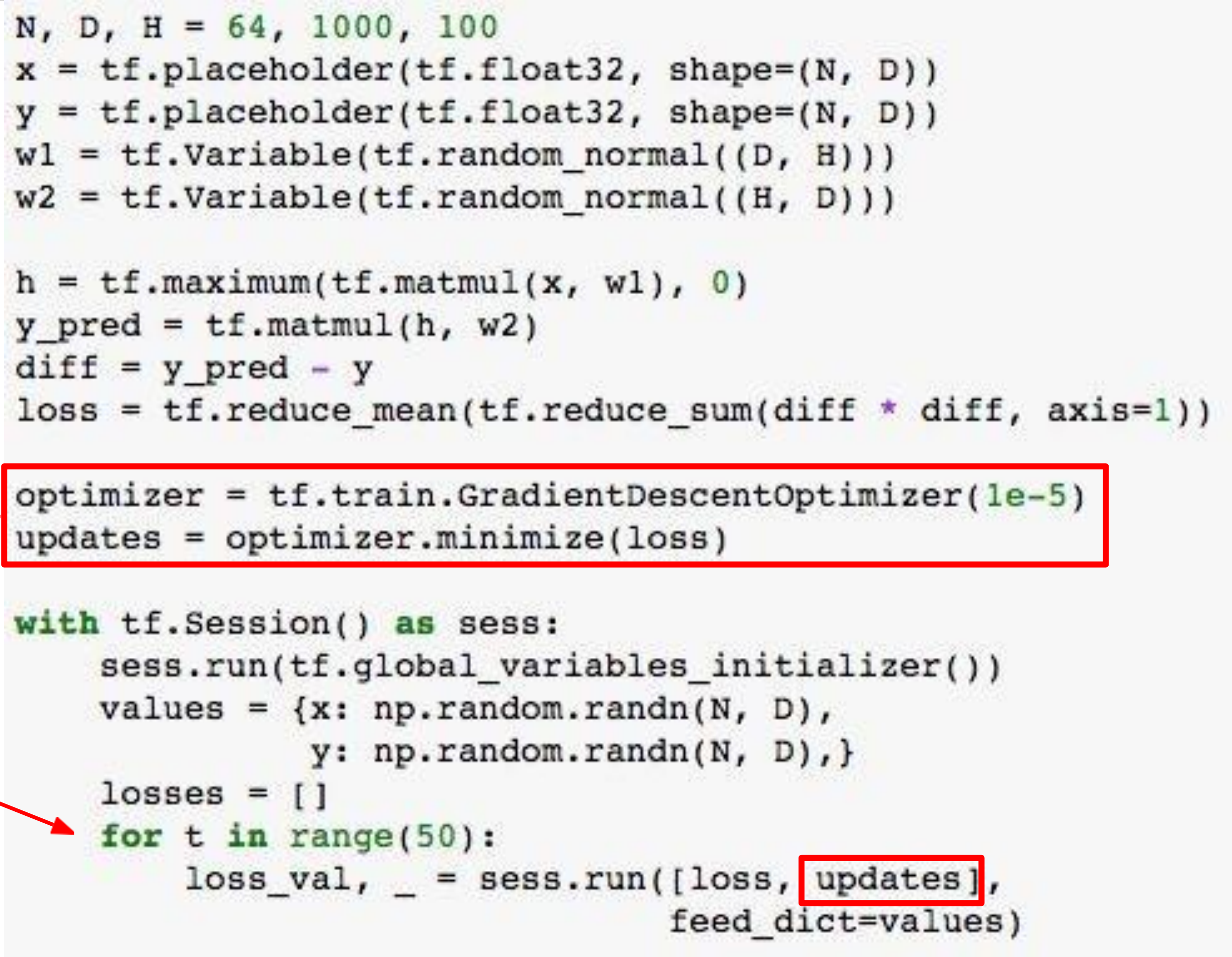


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Add dummy graph node that depends on updates

Tell TensorFlow to compute dummy node

TensorFlow: Optimizer



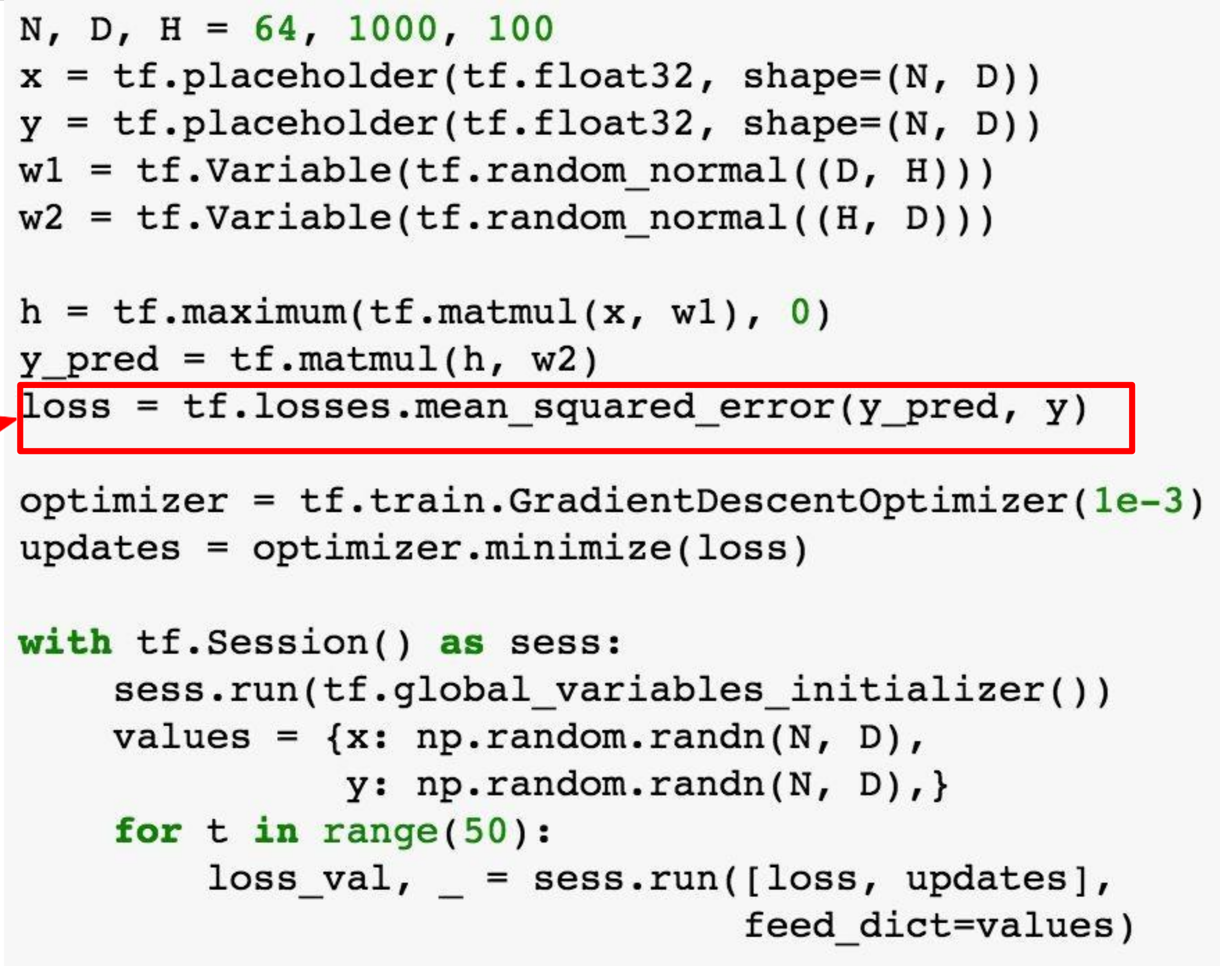
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff * diff, axis=1))

optimizer = tf.train.GradientDescentOptimizer(1e-5)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Can use an **optimizer** to compute gradients and update weights

Remember to execute the output of the optimizer!

TensorFlow: Loss

Use predefined
common lossees



```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-3)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

TensorFlow: Layers

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))

init = tf.variance_scaling_initializer(2.0)
h = tf.layers.dense(inputs=x, units=H,
        activation=tf.nn.relu, kernel_initializer=init)
y_pred = tf.layers.dense(inputs=h, units=D,
        kernel_initializer=init)

loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e0)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

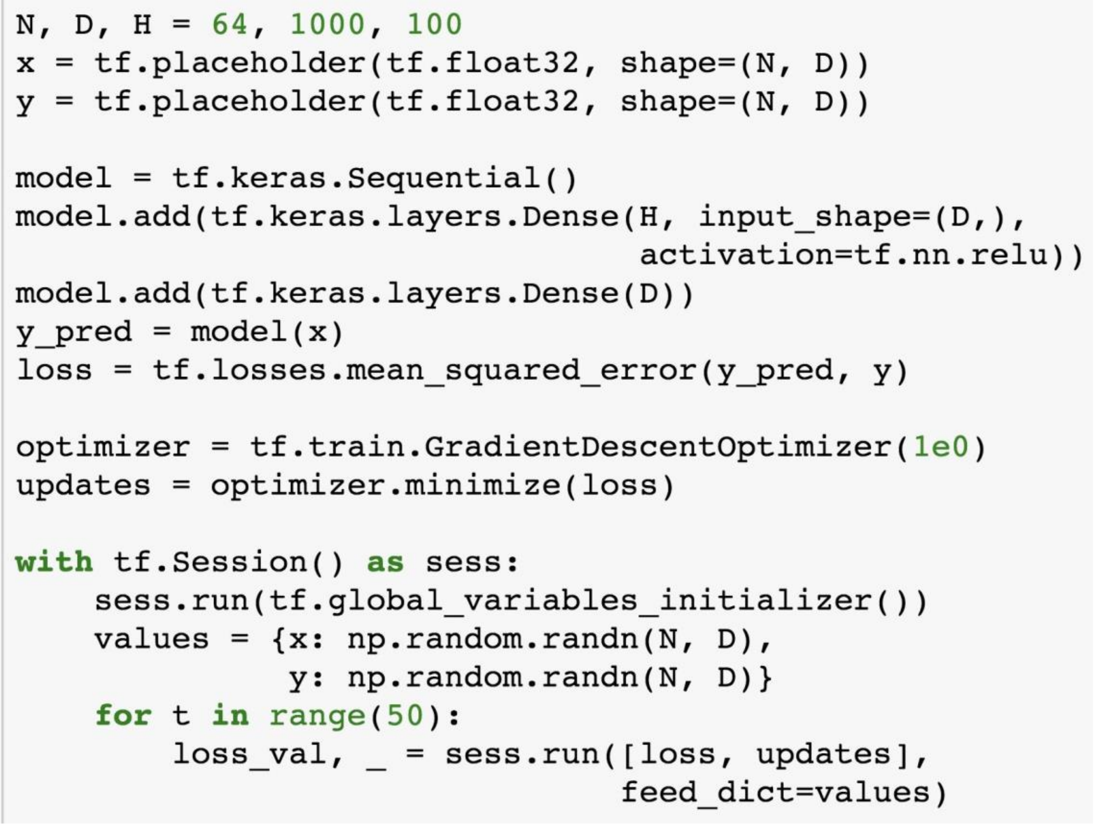
Use He
initializer

tf.layers automatically sets up weight and (and bias) for us!

Keras: High-Level Wrapper

Keras is a layer on top of TensorFlow, makes common things easy to do

(Used to be third-party, now merged into TensorFlow)

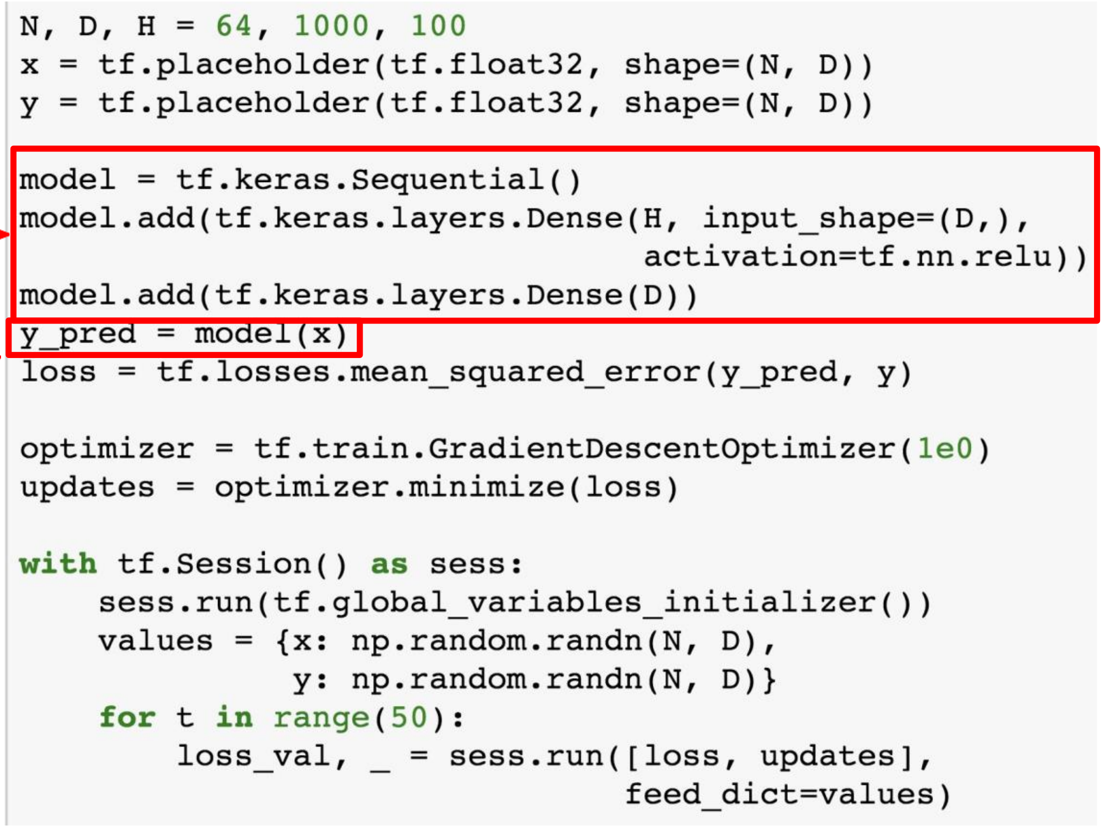


```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
y_pred = model(x)
loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e0)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Keras: High-Level Wrapper

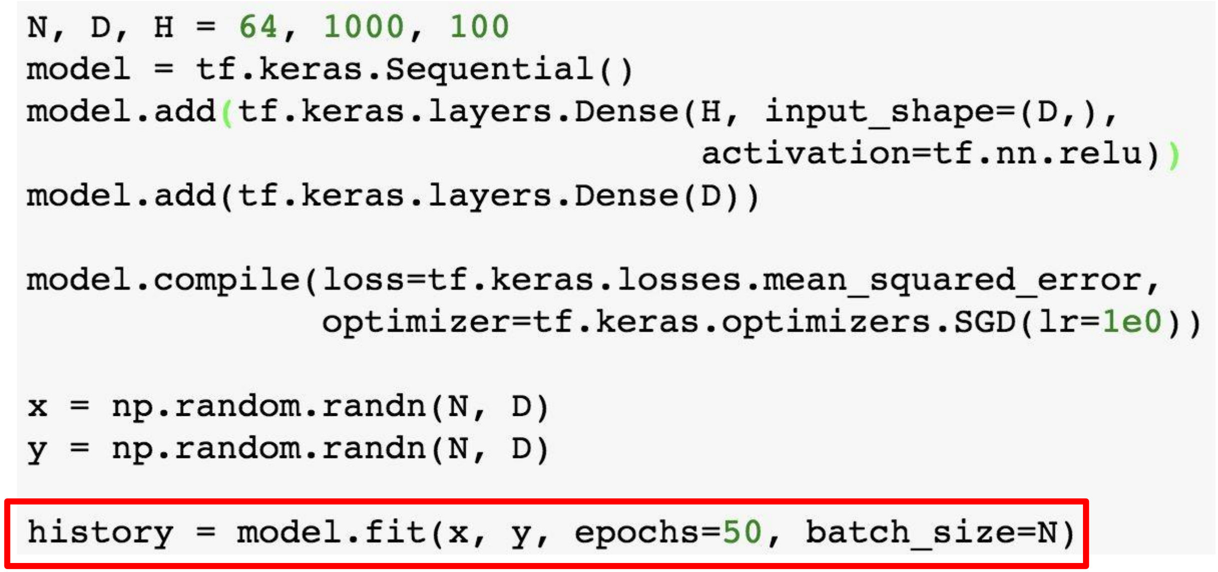
Define model as a sequence of layers

Get output by calling the model



```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
y_pred = model(x)
loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e0)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Keras: High-Level Wrapper

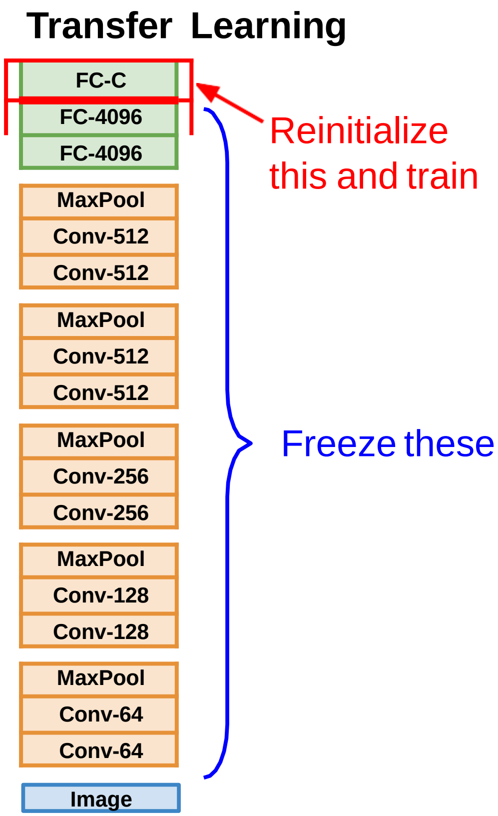


```
N, D, H = 64, 1000, 100
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))

model.compile(loss=tf.keras.losses.mean_squared_error,
              optimizer=tf.keras.optimizers.SGD(lr=1e0))

x = np.random.randn(N, D)
y = np.random.randn(N, D)

history = model.fit(x, y, epochs=50, batch_size=N)
```

Keras can handle the training loop for you! No sessions or feed_dict

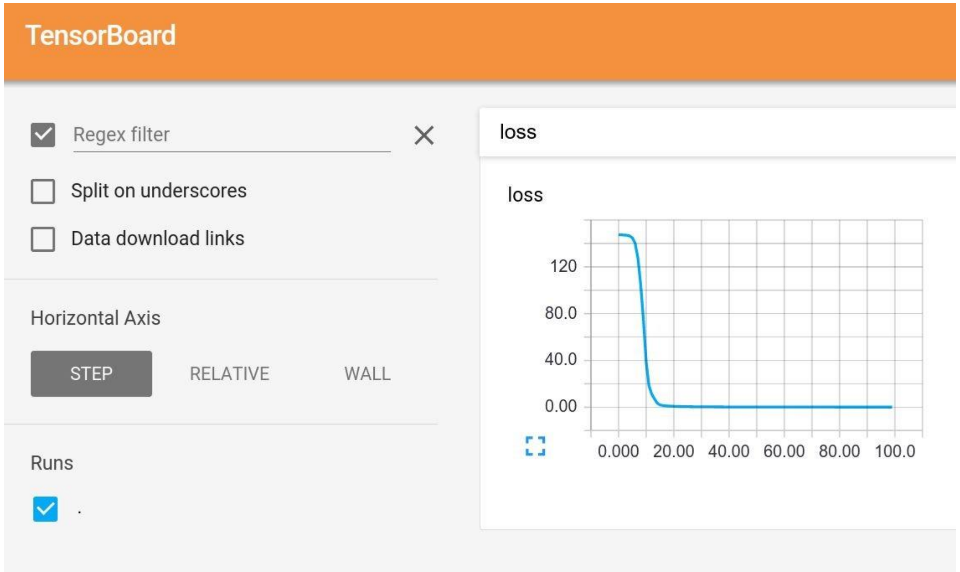
TensorFlow: Pretrained Models

tf.keras: (https://www.tensorflow.org/api_docs/python/tf/keras/applications)



TensorFlow: Tensorboard

Add logging to code to record loss, stats, etc
Run server and get pretty graphs!



☑ Regex filter

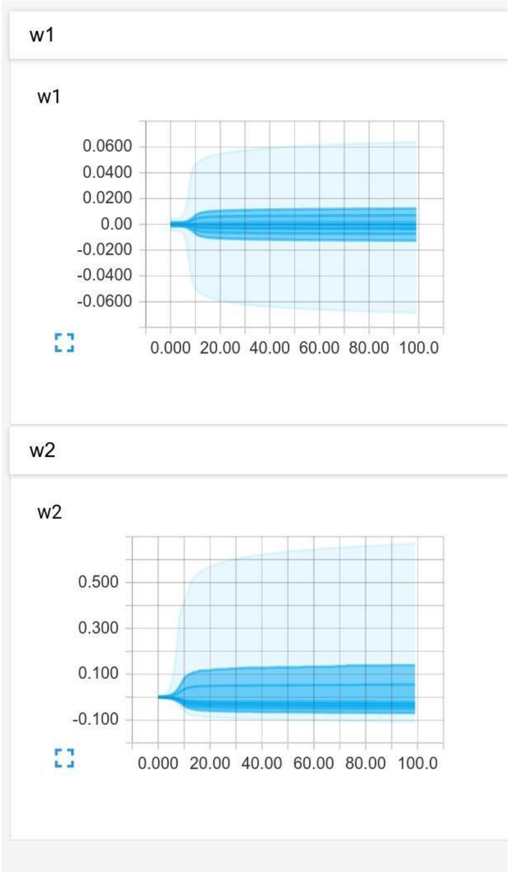
☐ Split on underscores

☐ Data download links

Horizontal Axis: STEP

Runs

☑ .



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

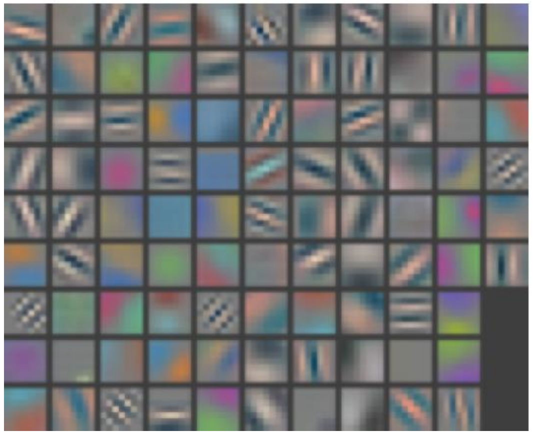
- Getting started: Preprocessing, initialization, optimization, normalization
- Improving performance: regularization, augmentation, transfer
- Hardware and software

Understanding CNNs

- Visualization
- Breaking CNNs

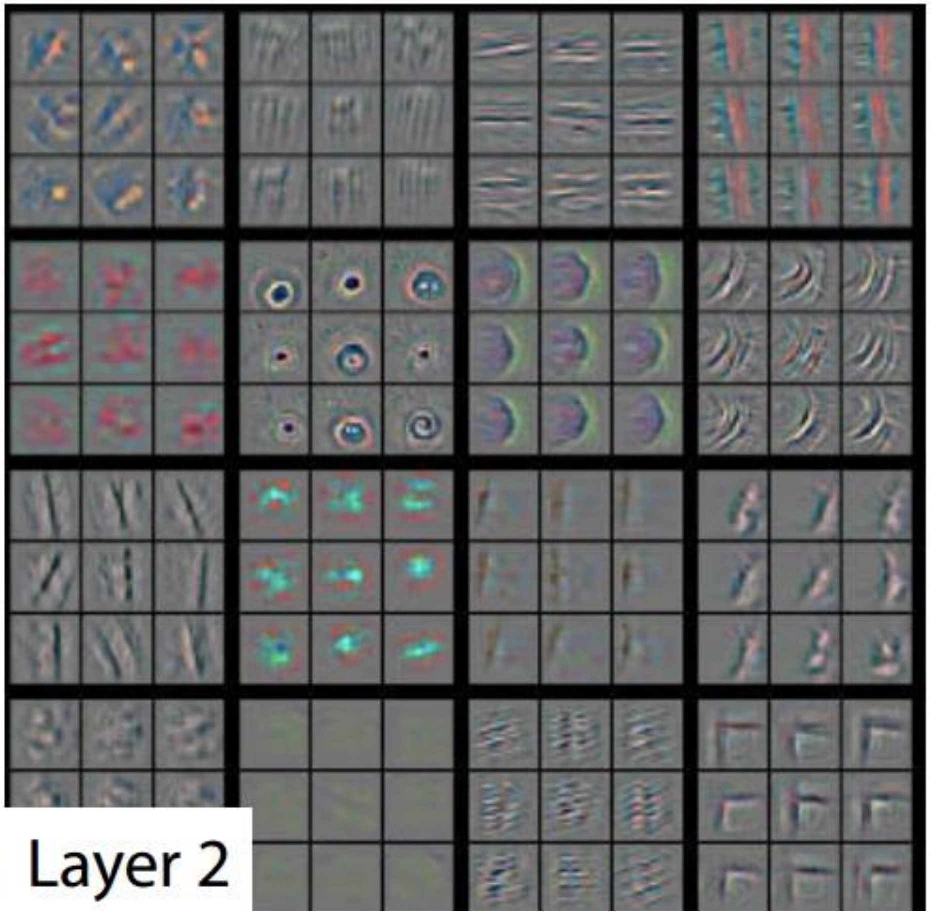
Understanding CNNs

Layer 1

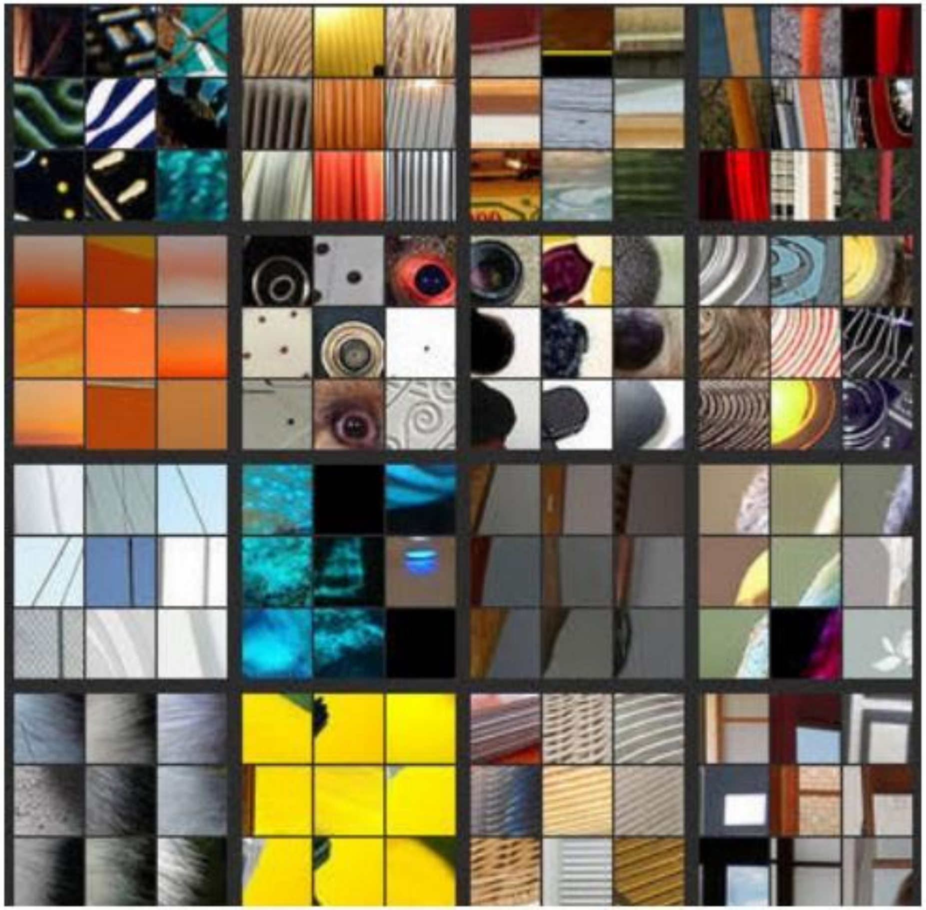


Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

Layer 2



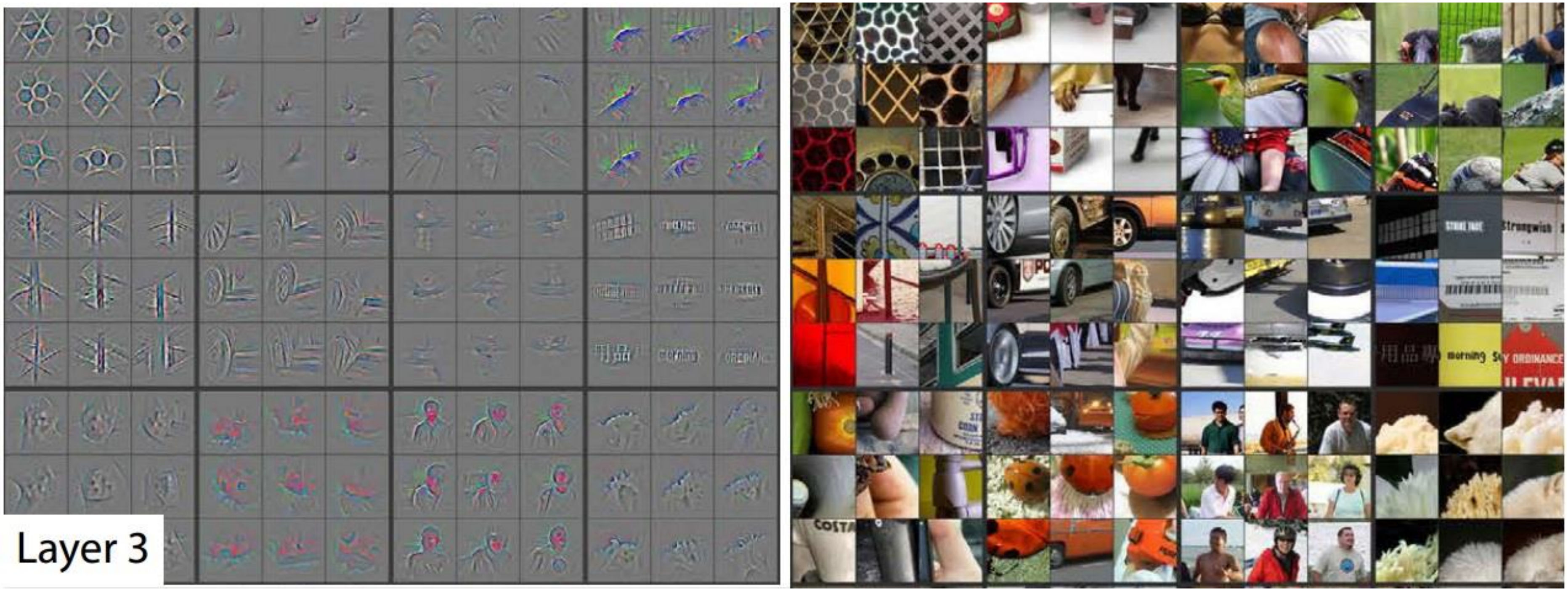
- Activations projected down to pixel level via decovolution



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

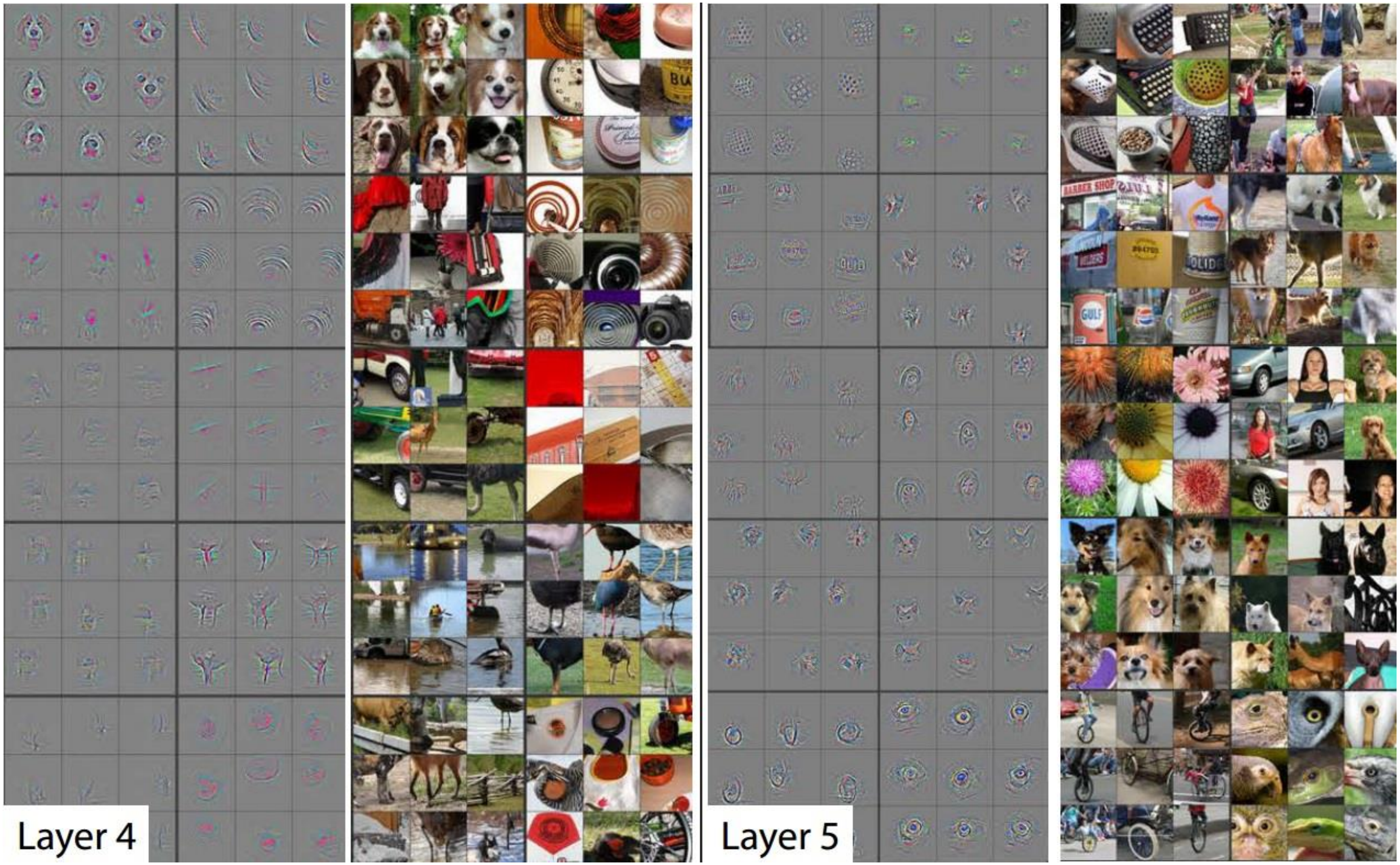
Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

Layer 3



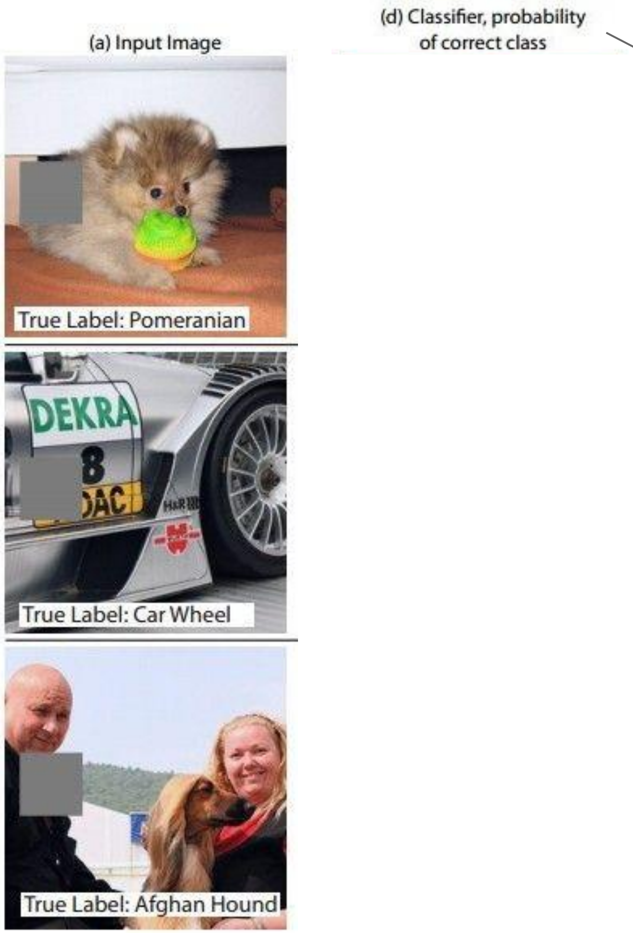
Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

Layer 4 and 5



Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

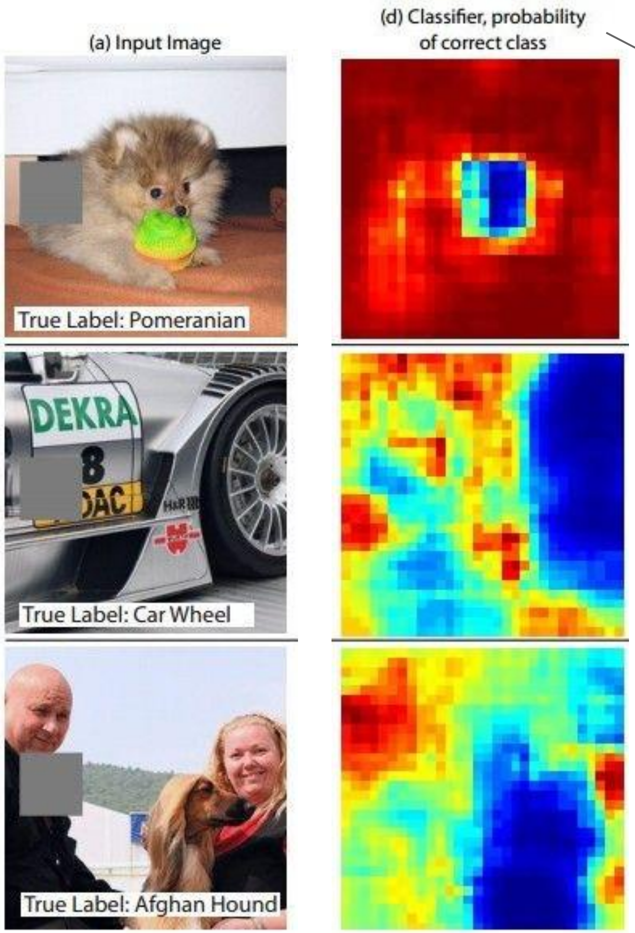
Occlusion experiments



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

[Zeiler & Fergus 2014]

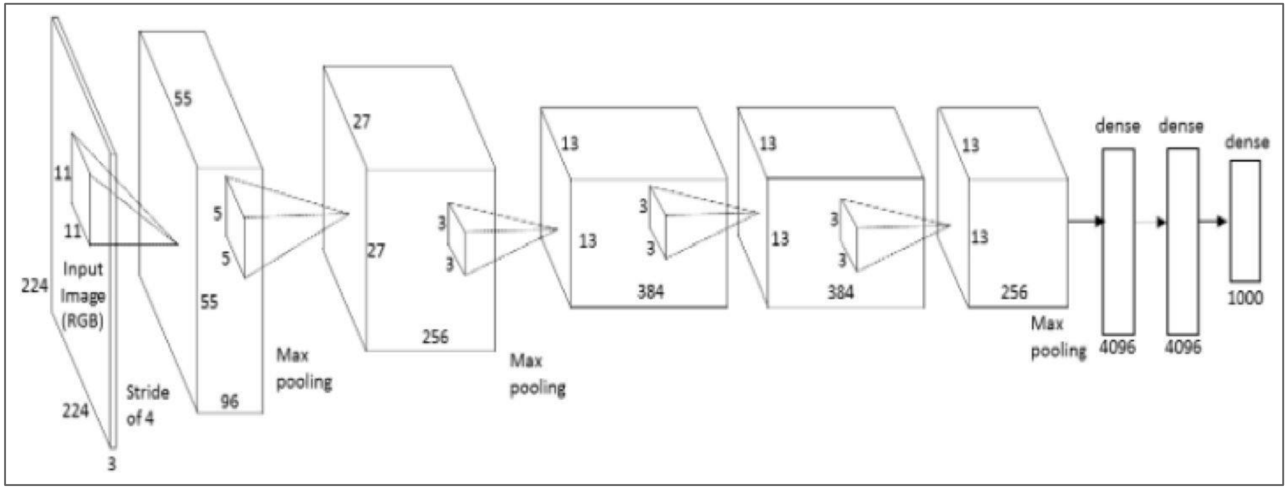
Occlusion experiments



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

[Zeiler & Fergus 2014]

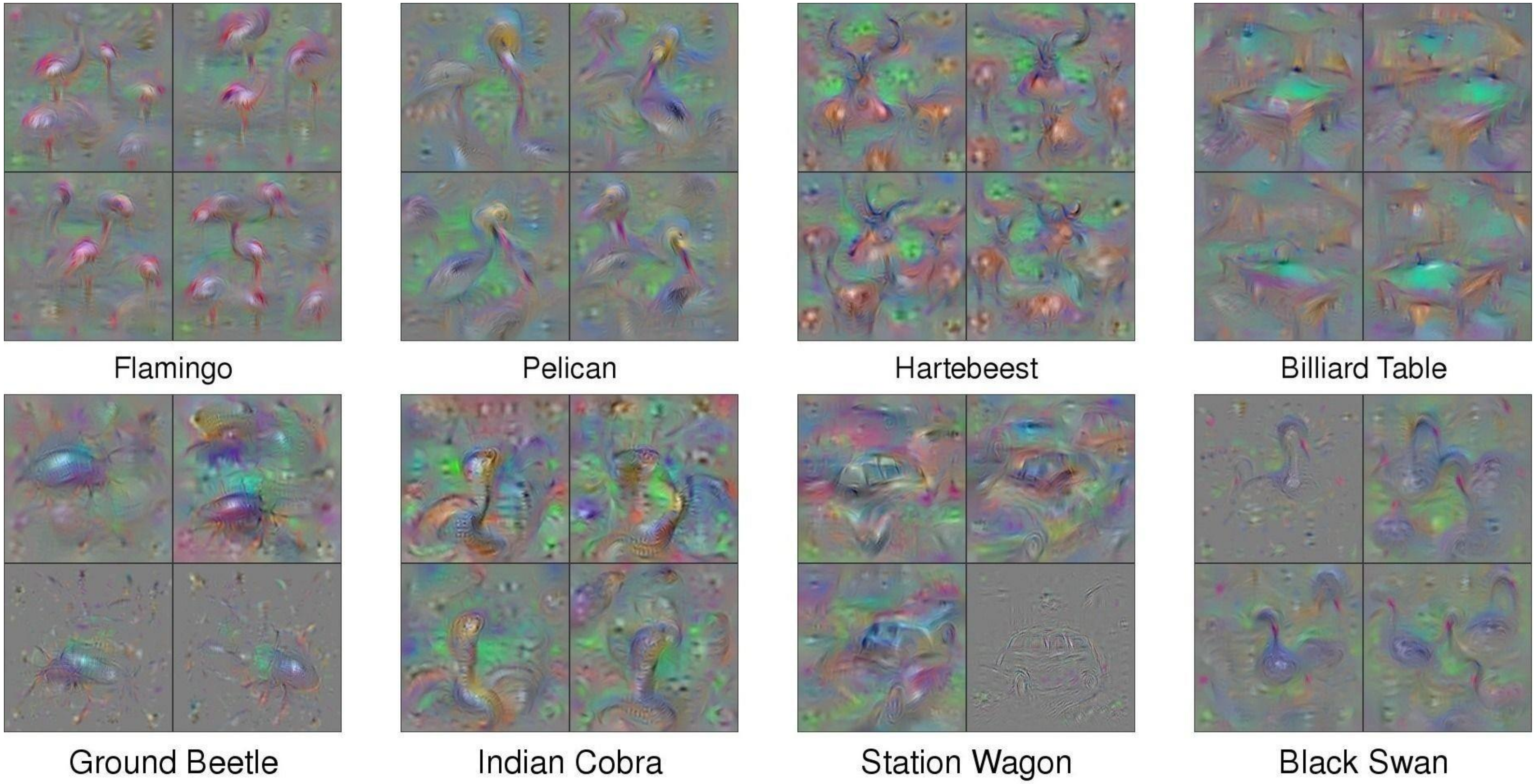
What image maximizes a class score?



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”

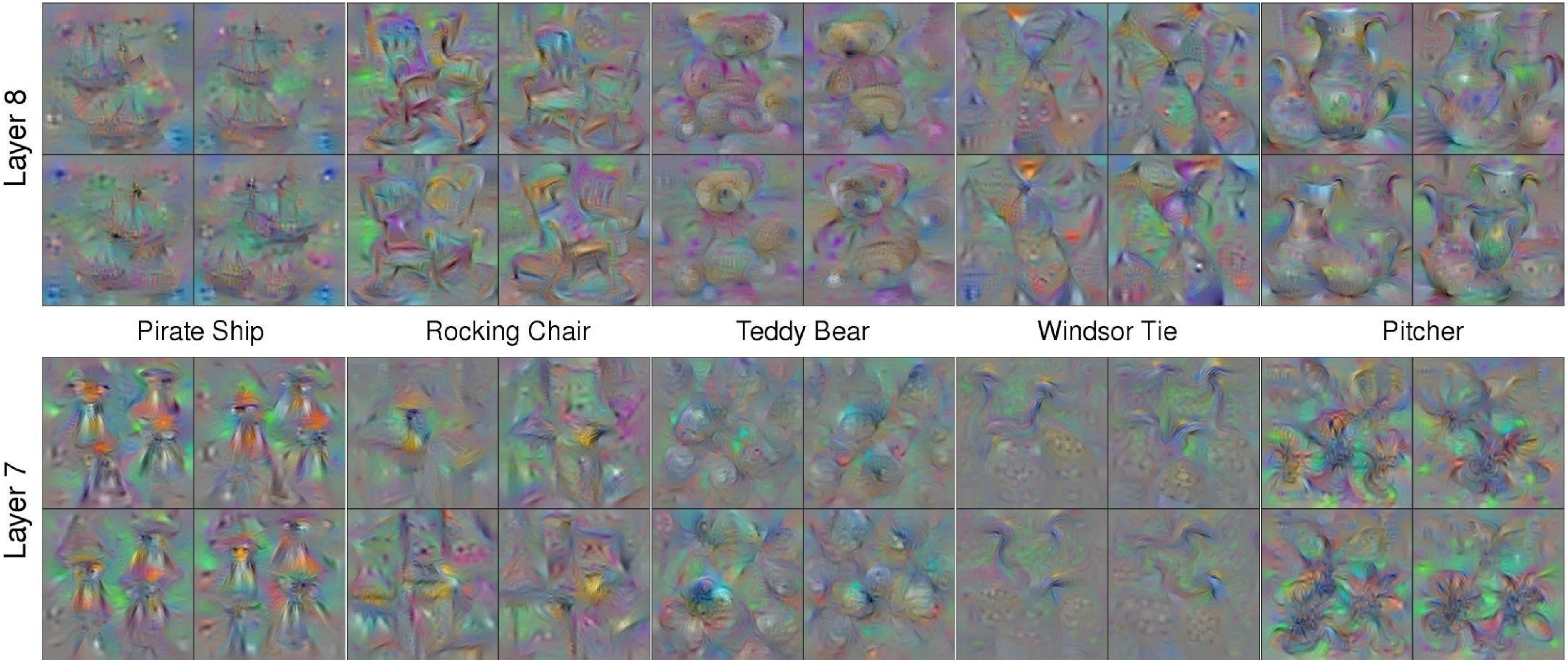
What image maximizes a class score?



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

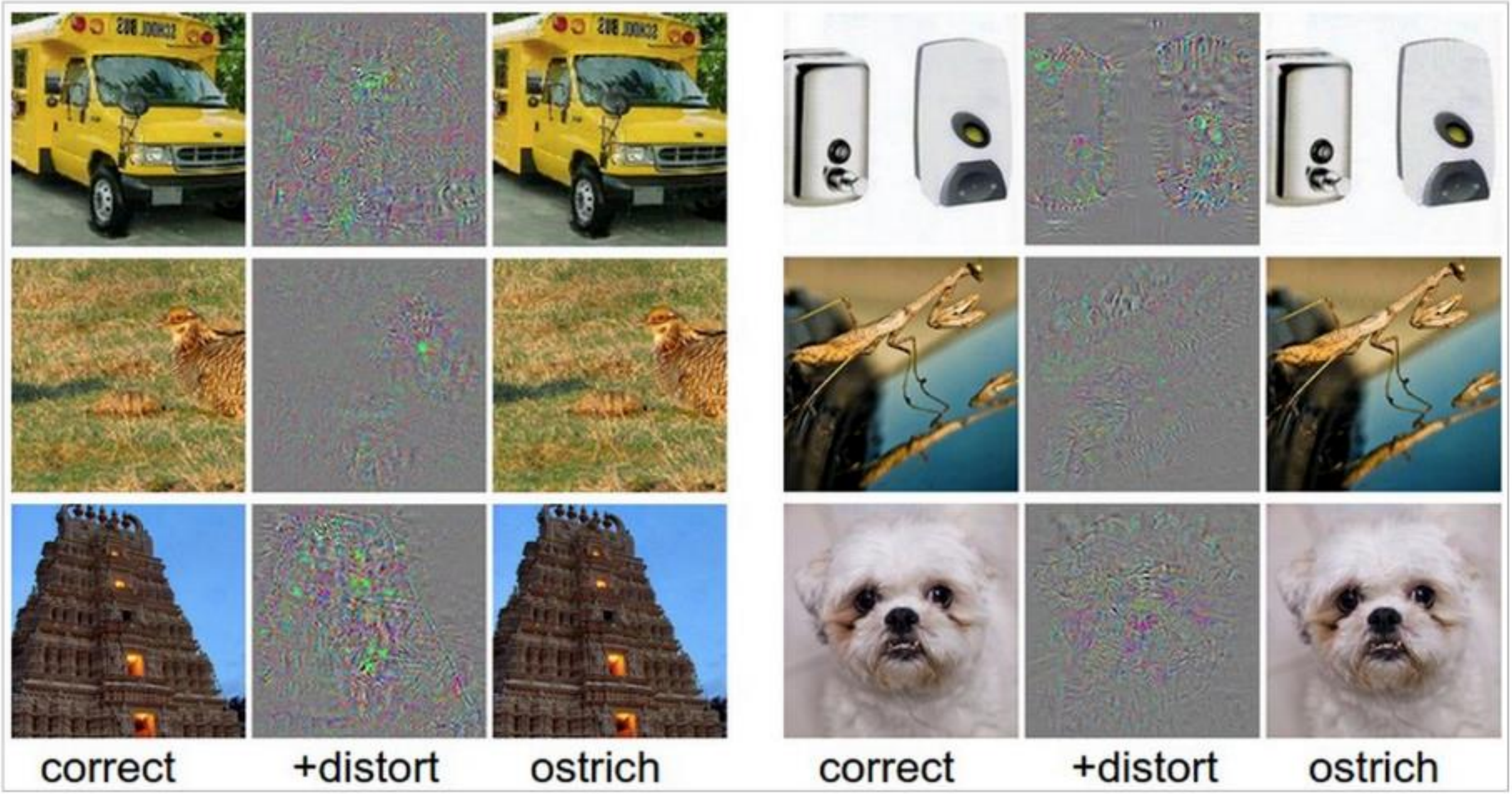
http://yosinski.com/deepvis

What image maximizes a class score?



Andrej Karpathy

Breaking CNNs

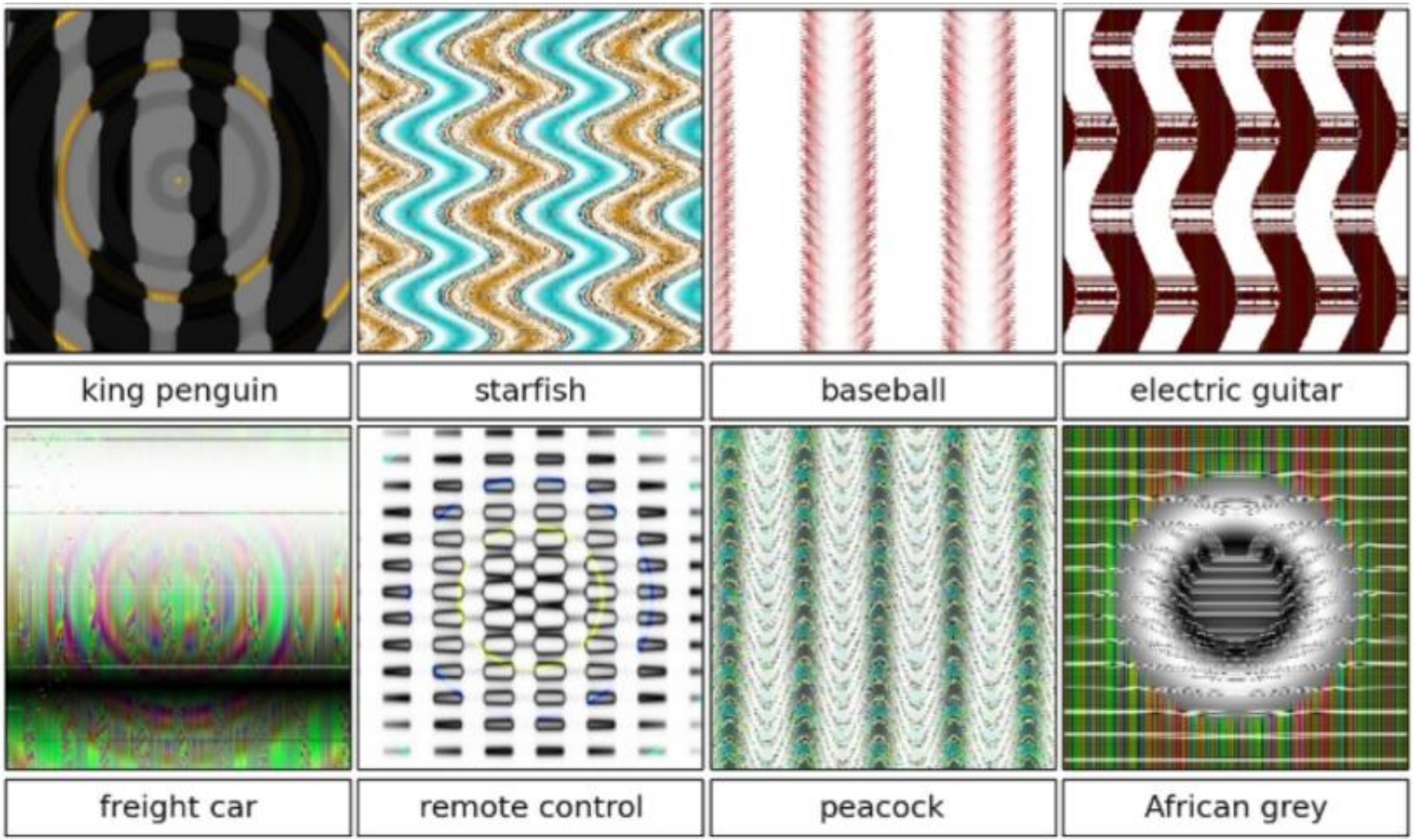


Take a correctly classified image (left image in both columns), and add a tiny distortion (middle) to fool the ConvNet with the resulting image (right).

Andrej Karpathy

Intriguing properties of neural networks [Szegedy ICLR 2014]

Breaking CNNs



king penguin

starfish

baseball

electric guitar

freight car

remote control

peacock

African grey

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