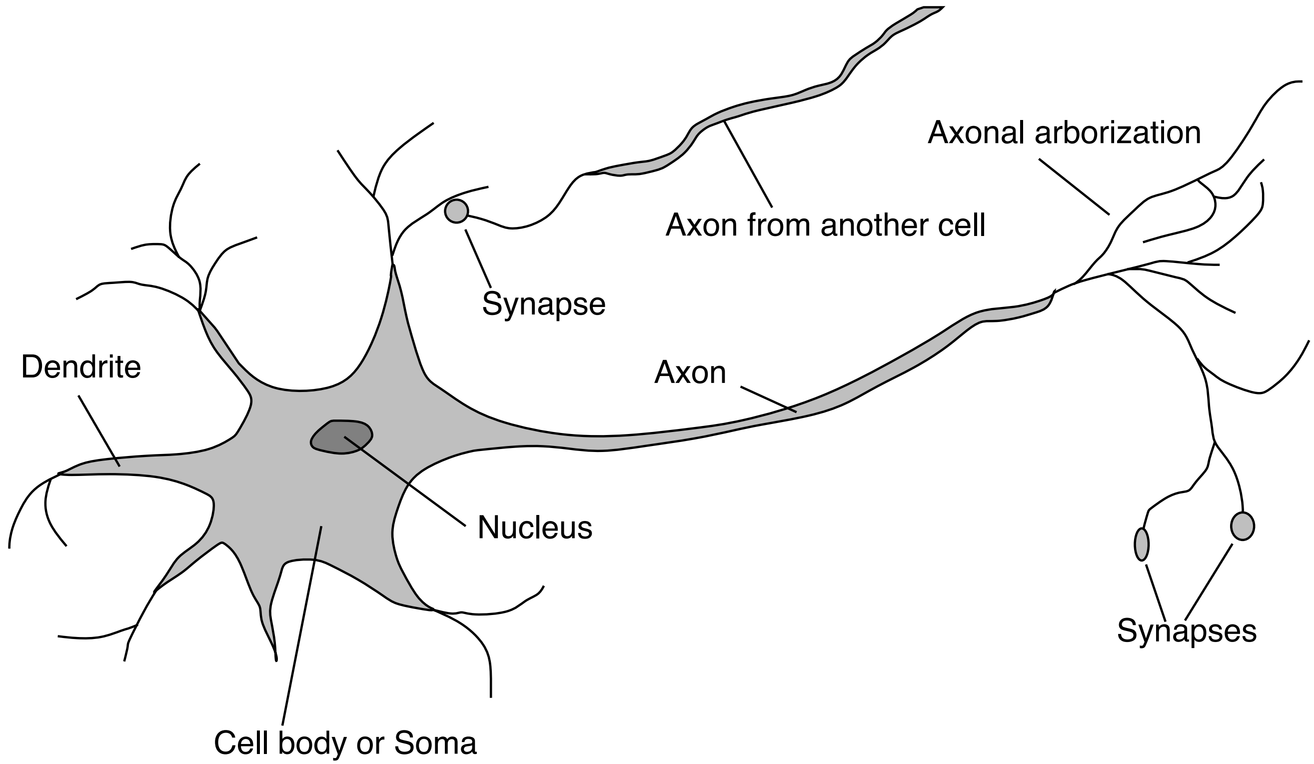
Neural networks

Slides adapted from Stuart Russell

Brains

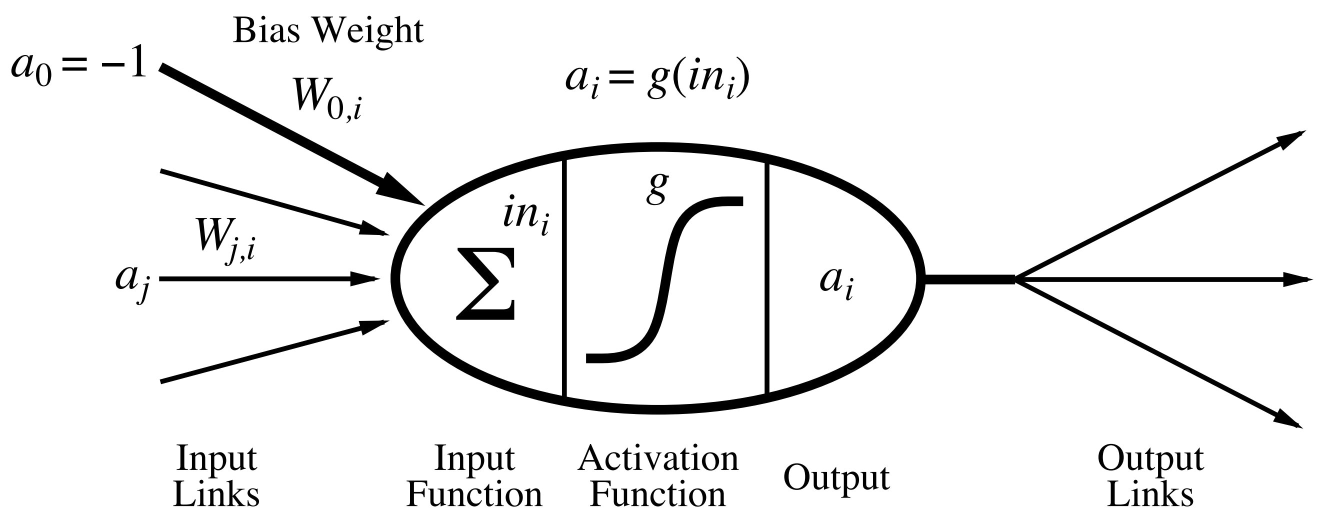
10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



McCulloch–Pitts "unit"

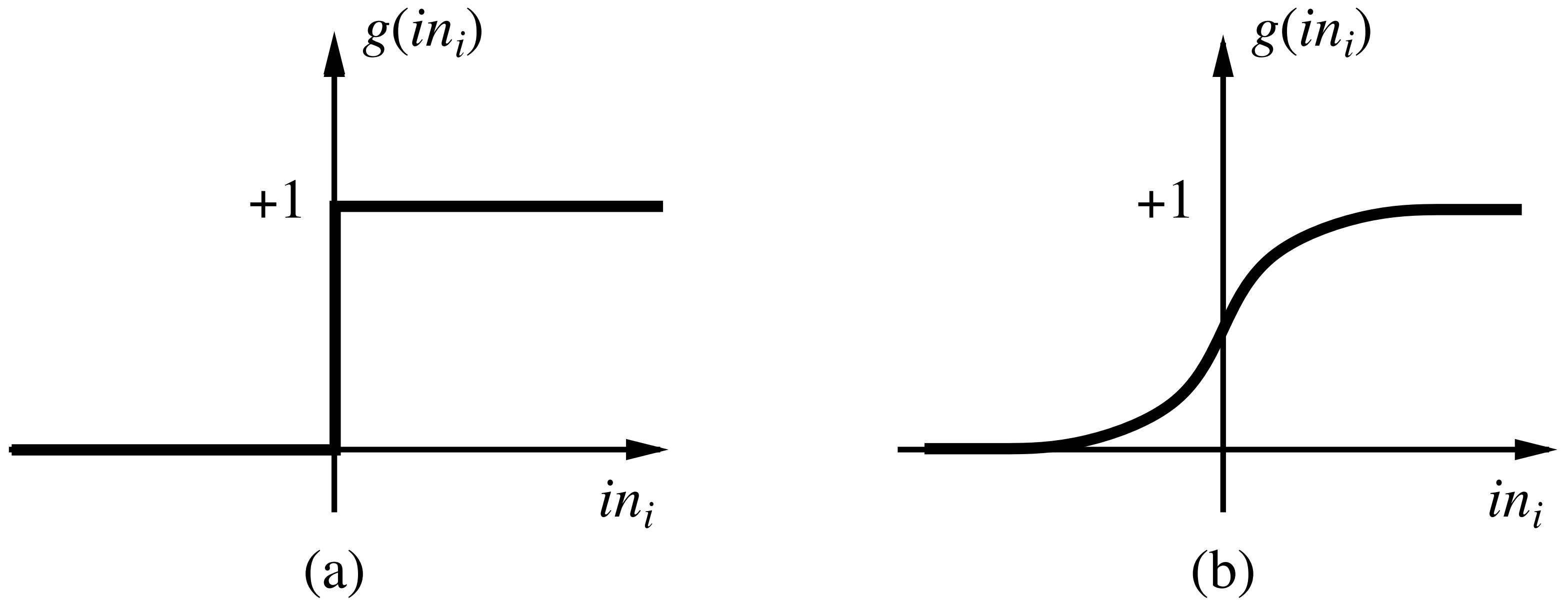
Output is a "squashed" linear function of the inputs:

$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$



A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

Activation functions



(a) is a step function or threshold function

(b) is a sigmoid function $1/(1+e^{-x})$

Changing the bias weight $W_{0,i}$ moves the threshold location

Network structures

Feed-forward networks:

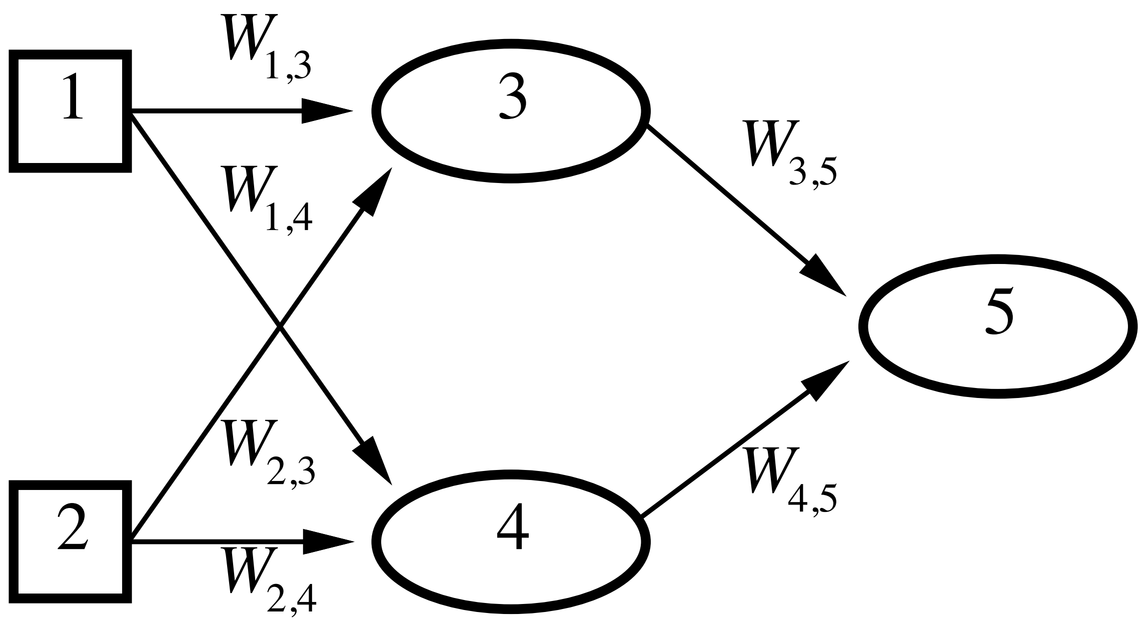
– single-layer perceptrons
– multi-layer perceptrons

Feed-forward networks implement functions, have no internal state

Recurrent networks:

– recurrent neural nets have directed cycles with delays
    $\Rightarrow$ have internal state (like flip-flops), can oscillate etc.

Feed-forward example

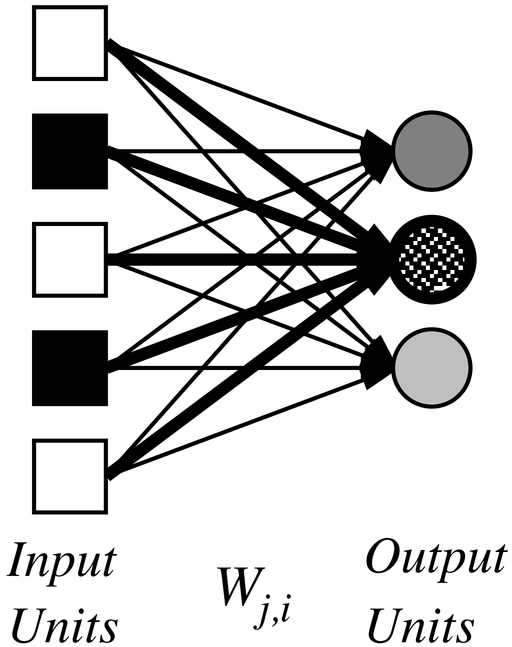


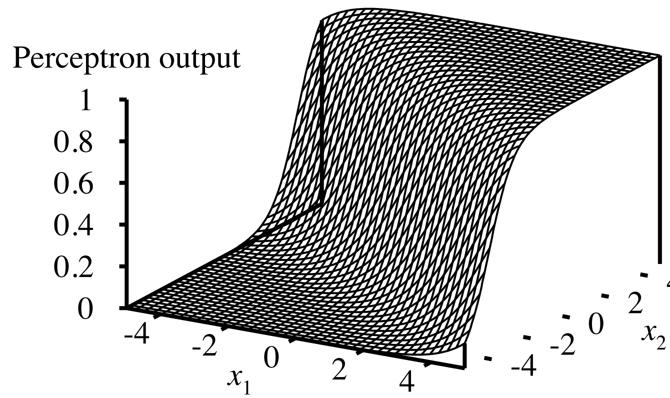
Feed-forward network = a parameterized family of nonlinear functions:

$$
\begin{aligned}
a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\
&= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))
\end{aligned}
$$

Adjusting weights changes the function: do learning this way!

Single-layer perceptrons





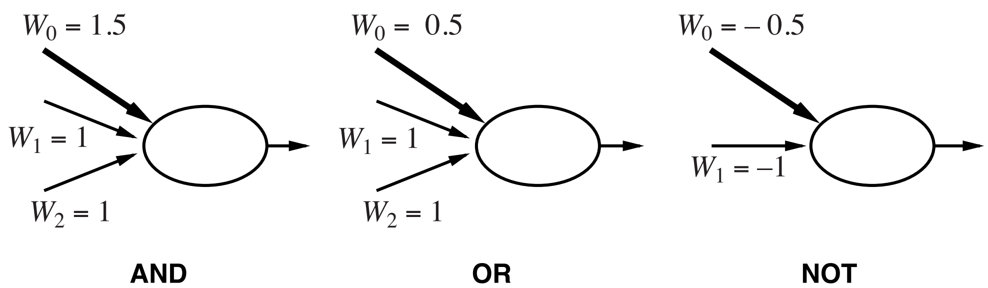
Adjusting weights moves the location, orientation, and steepness of cliff

Expressiveness of perceptrons

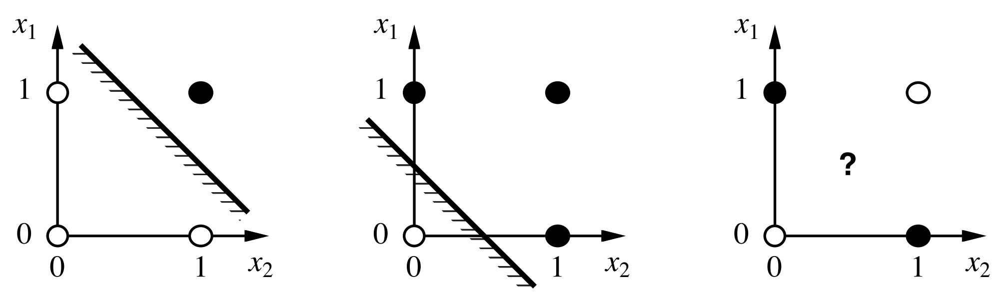
Consider a perceptron with g = step function (Rosenblatt, 1957, 1960). Represents a linear separator in input space:

$$\Sigma_j W_j x_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{x} > 0$$

Can represent AND, OR, NOT, majority, etc.:



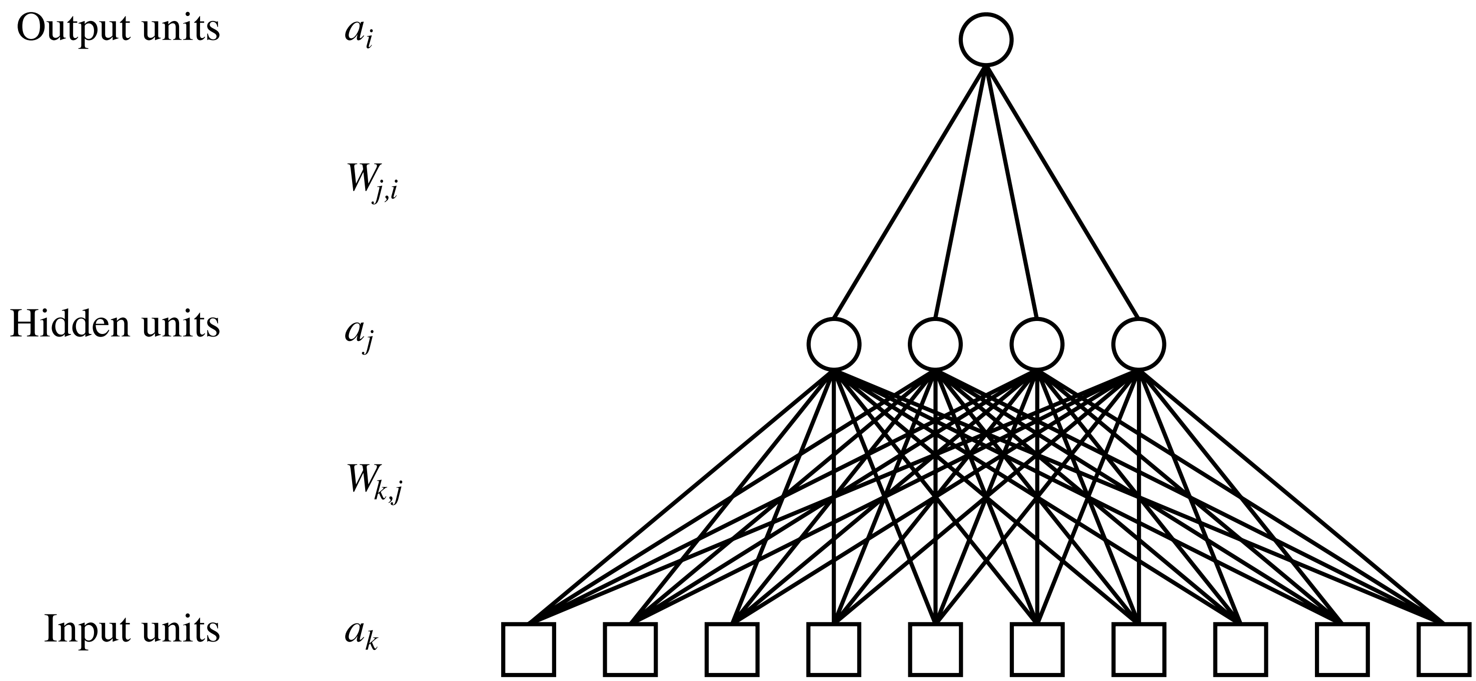
But not XOR:



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

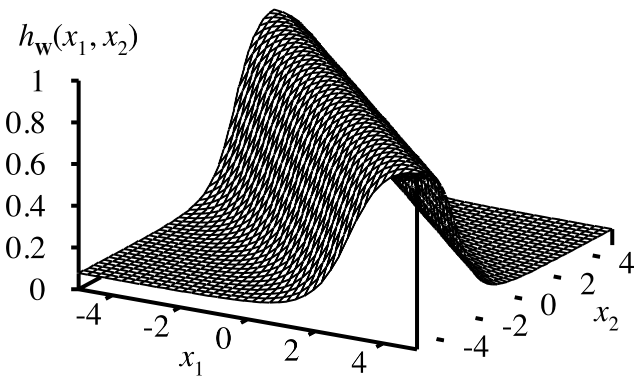
Multilayer perceptrons

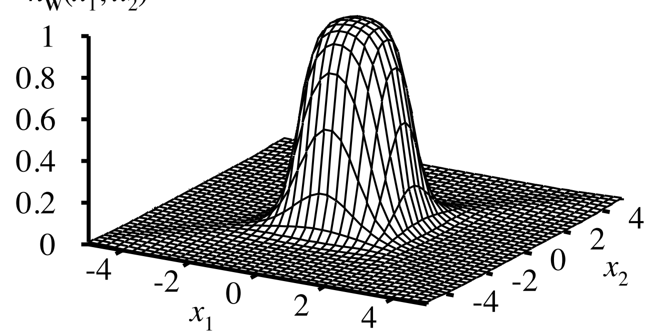
Layers are usually fully connected;
numbers of hidden units typically chosen by hand



Expressiveness of MLPs

All continuous functions w/ 2 layers, all functions w/ 3 layers





Combine two opposite-facing threshold functions to make a ridge

Combine two perpendicular ridges to make a bump

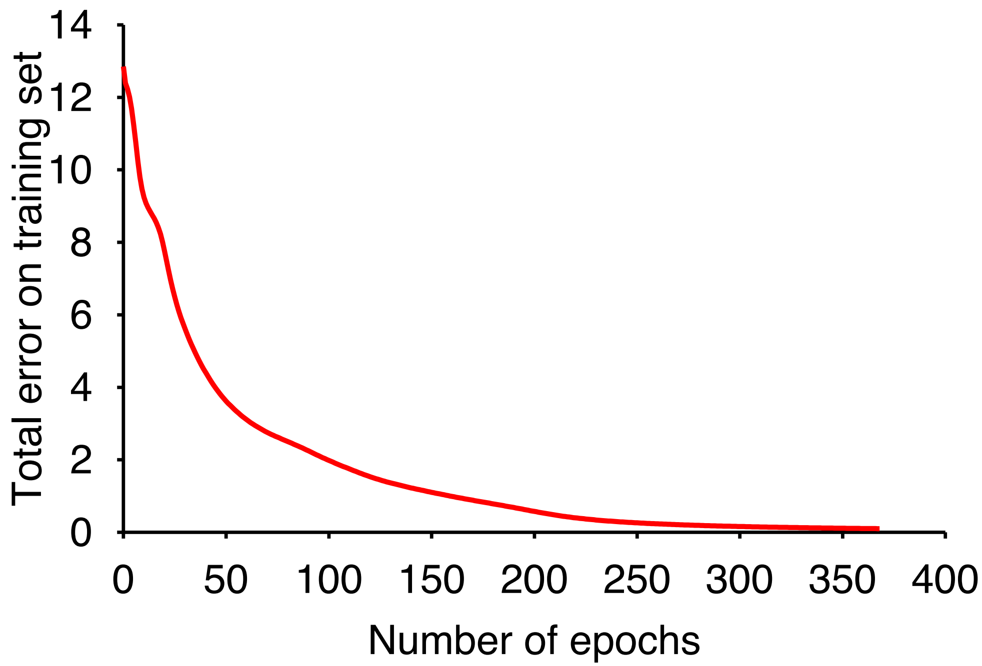
Add bumps of various sizes and locations to fit any surface

Proof requires exponentially many hidden units

Back-propagation learning

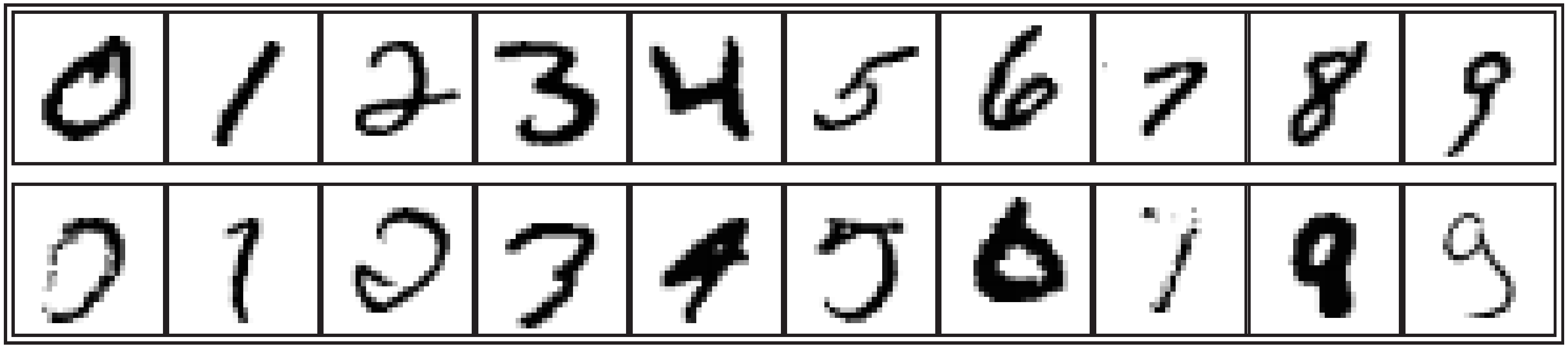
At each epoch, sum gradient updates for all examples and apply

Training curve for 100 restaurant examples: finds exact fit



Typical problems: slow convergence, local minima

Handwritten digit recognition



3-nearest-neighbor = 2.4% error
400–300–10 unit MLP = 1.6% error
LeNet (1998): 768–192–30–10 unit MLP = 0.9% error
SVMs: ≈ 0.6% error

Current best: 0.24% error (committee of convolutional nets)

Example: ALVINN

steering direction

[Pomerleau, 1995]

Backpropagation

Slides adapted from Kyunghyun Cho

Learning as an Optimization

Ultimately, learning is (*mostly*)

$$\boldsymbol{\theta} = \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} c\left((x_n, y_n) \mid \boldsymbol{\theta}\right) + \lambda \Omega\left(\boldsymbol{\theta}, D\right),$$

where $c\left((x, y) \mid \boldsymbol{\theta}\right)$ is a per-sample cost function.

Gradient Descent

Gradient-descent Algorithm:

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta \nabla L(\boldsymbol{\theta}^{t-1})$$

where, in our case,

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} l\left((x_n, y_n) \mid \boldsymbol{\theta}\right).$$

Let us assume that $\Omega(\boldsymbol{\theta}, D) = 0$.

Stochastic Gradient Descent

Often, it is too costly to compute $C(\boldsymbol{\theta})$ due to a large training set.

Stochastic gradient descent algorithm:

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta^t \nabla l\left((x', y') \mid \boldsymbol{\theta}^{t-1}\right),$$

where (x', y') is a randomly chosen sample from D, and

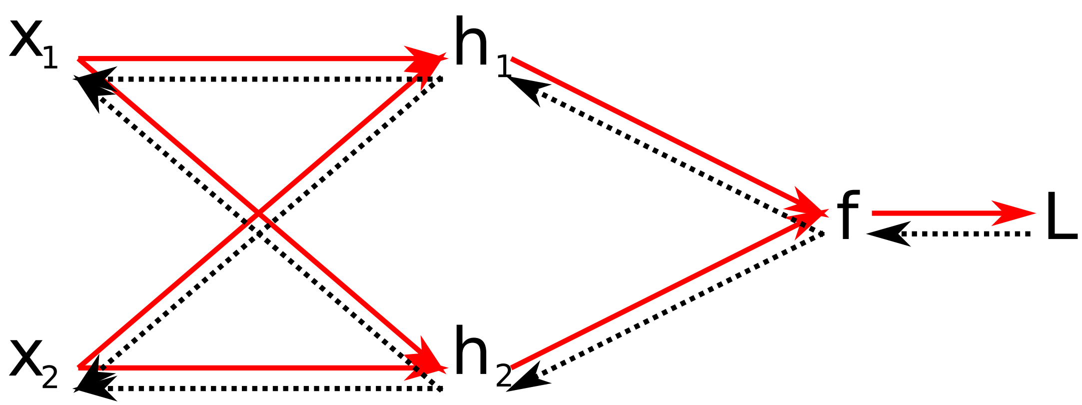
$$\sum_{t=1}^{\infty} \eta^t \to \infty \text{ and } \sum_{t=1}^{\infty} \left(\eta^t\right)^2 < \infty.$$

Let us assume that $\Omega\left(\boldsymbol{\theta}, D\right) = 0$.

Almost there. . .

How do we compute the gradient efficiently for neural networks?

Backpropagation Algorithm – (1) Forward Pass



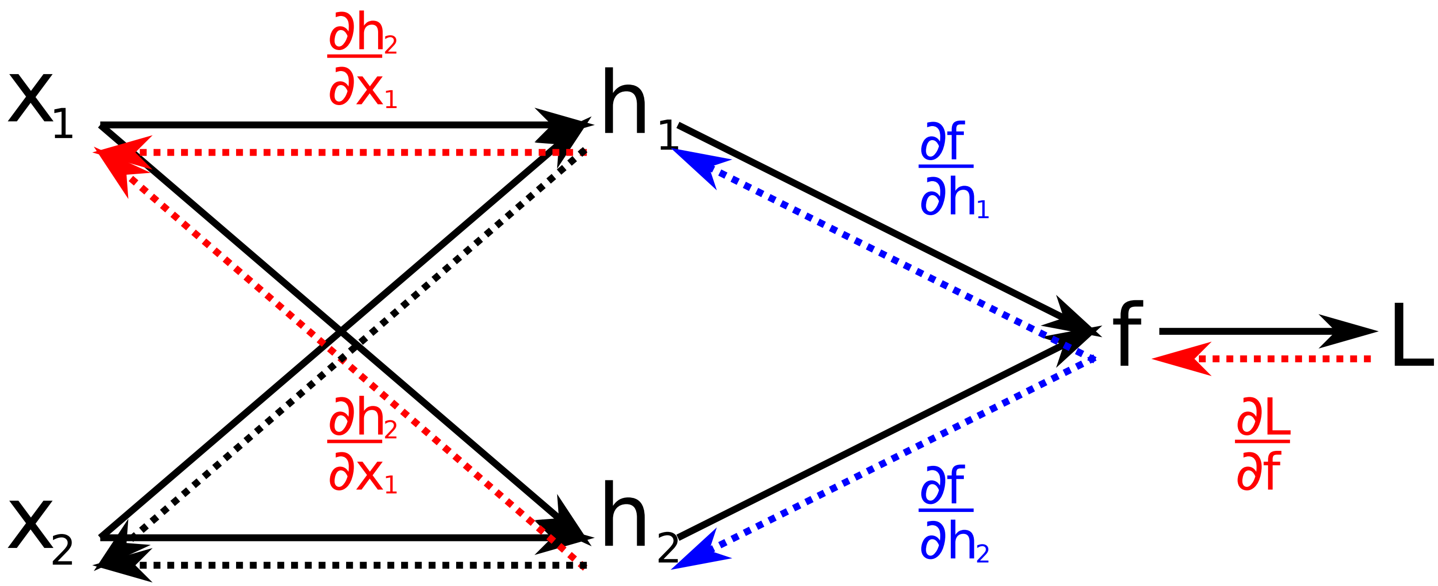
Forward Computation:

$$L(f(h_1(x_1, x_2, \boldsymbol{\theta}_{h_1}), h_2(x_1, x_2, \boldsymbol{\theta}_{h_2}), \boldsymbol{\theta}_f), y)$$

Multilayer Perceptron with a single hidden layer:

$$L(\mathbf{x}, y, \boldsymbol{\theta}) = \frac{1}{2}\left(y - \mathbf{U}^\top \phi\left(\mathbf{W}^\top \mathbf{x}\right)\right)^2$$

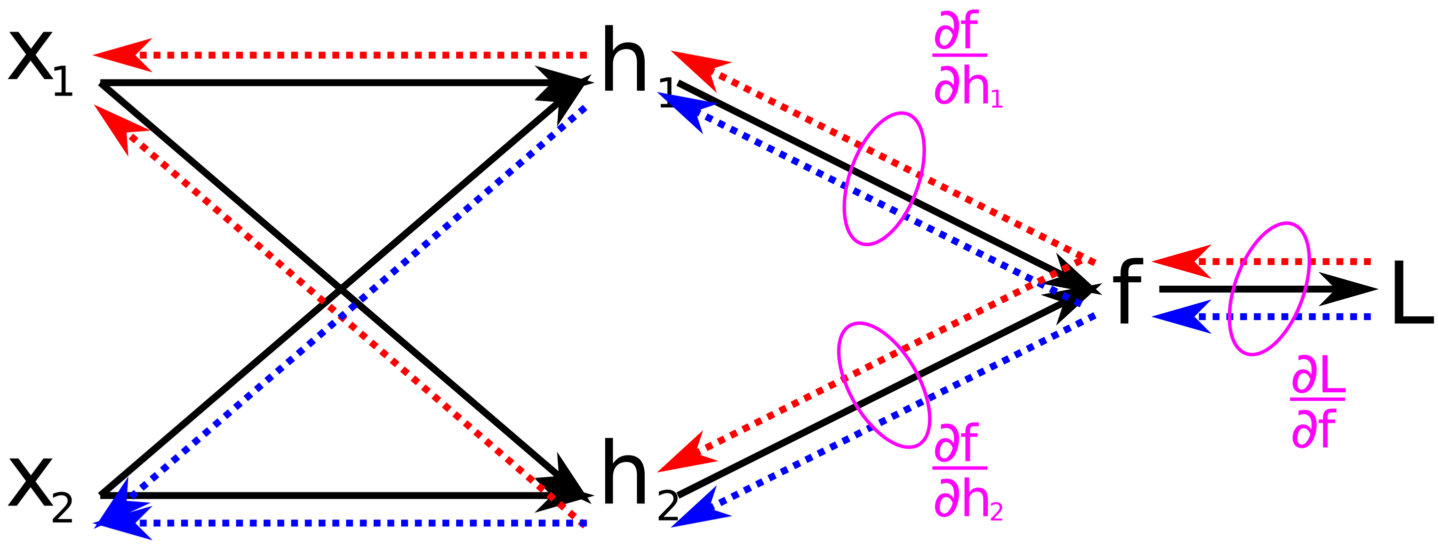
Backpropagation Algorithm – (2) Chain Rule



Chain rule of derivatives:

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial x_1} = \frac{\partial L}{\partial f}\left(\frac{\partial f}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial x_1}\right)$$

Backpropagation Algorithm – (3) Shared Derivatives

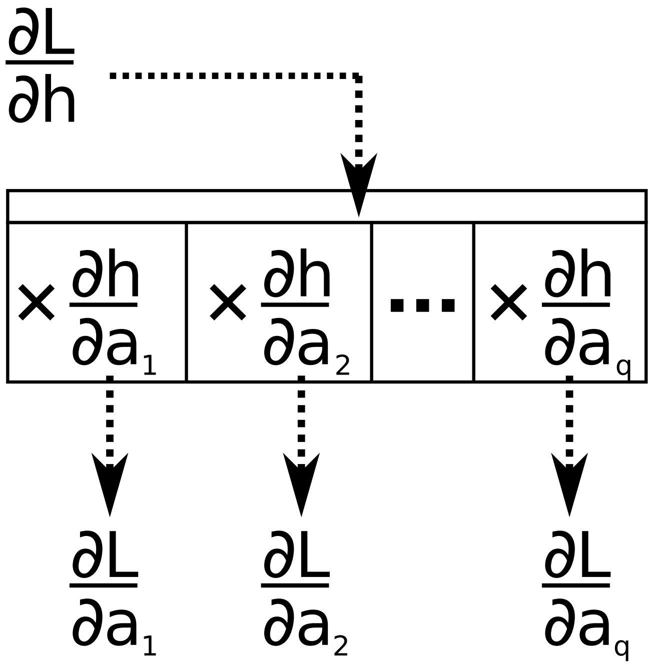


Local derivatives are *shared*:

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial f}\left(\frac{\partial f}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial x_1}\right)$$

$$\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial f}\left(\frac{\partial f}{\partial h_1}\frac{\partial h_1}{\partial x_2} + \frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial x_2}\right)$$

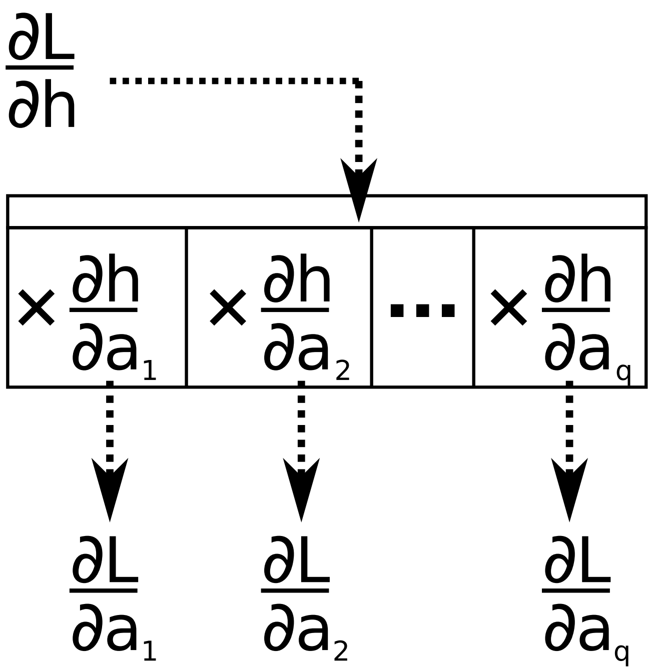
Backpropagation Algorithm – (4) Local Computation



Each node computes

- Forward: $h(a_1, a_2, \dots, a_q)$
- Backward: $\frac{\partial h}{\partial a_1}, \frac{\partial h}{\partial a_2}, \dots, \frac{\partial h}{\partial a_q}$

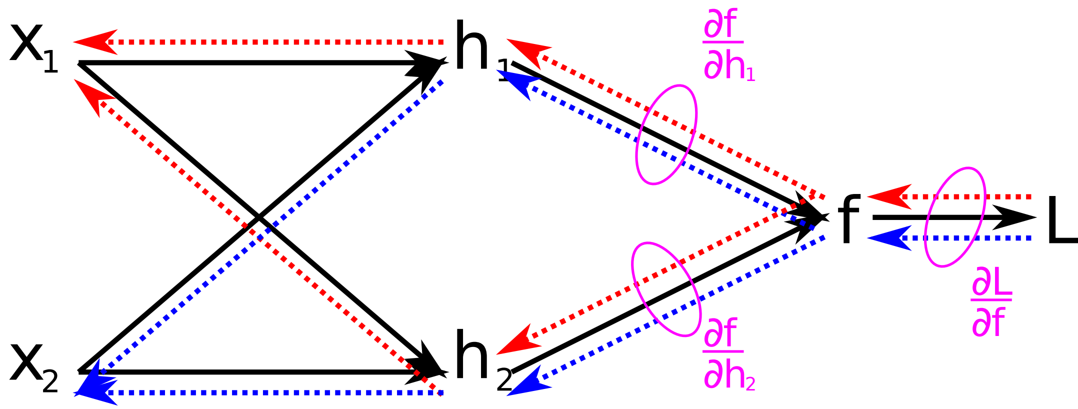
Backpropagation Algorithm – Requirements



- Each node computes a *differentiable* function[1]
- Directed Acyclic Graph[2]

[1]Well. . . ?
[2]Well. . . ?

Backpropagation Algorithm – Automatic Differentiation



theano

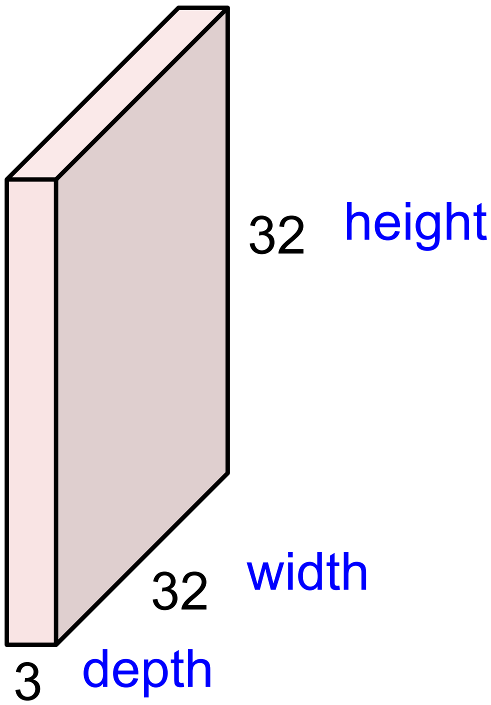
- Generalized approach to computing partial derivatives
- As long as your neural network fits the requirements, you do *not* need to derive the derivatives yourself!
  - Theano, Torch, . . .

Convolutional Neural Networks

(First without the brain stuff)

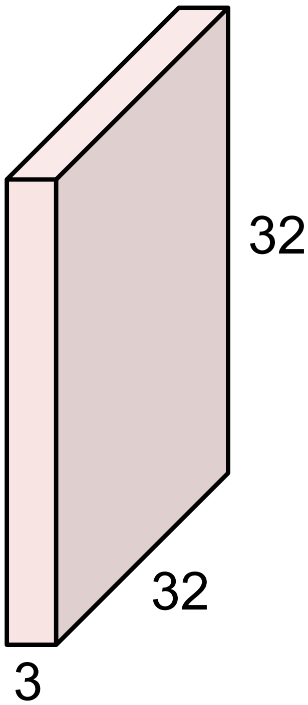
Convolution Layer

32x32x3 image

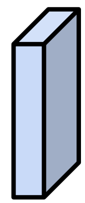


Convolution Layer

32x32x3 image



5x5x3 filter

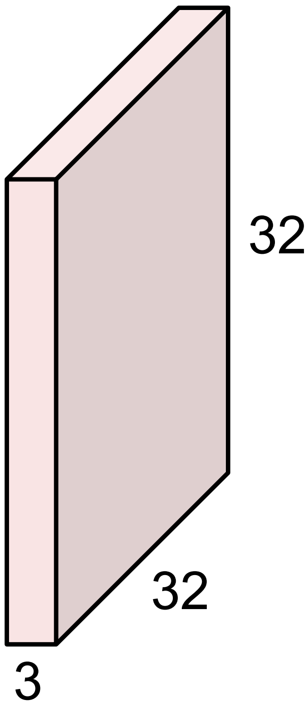


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

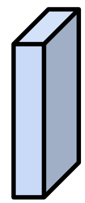
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image



5x5x3 filter

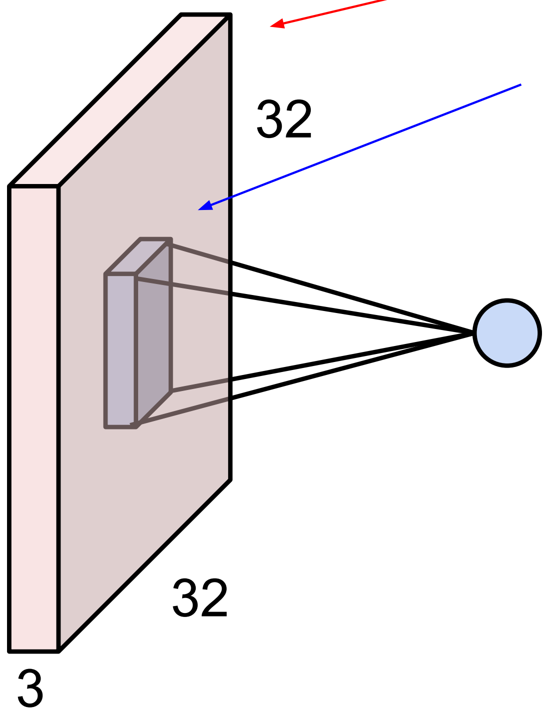


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Convolution Layer

32x32x3 image

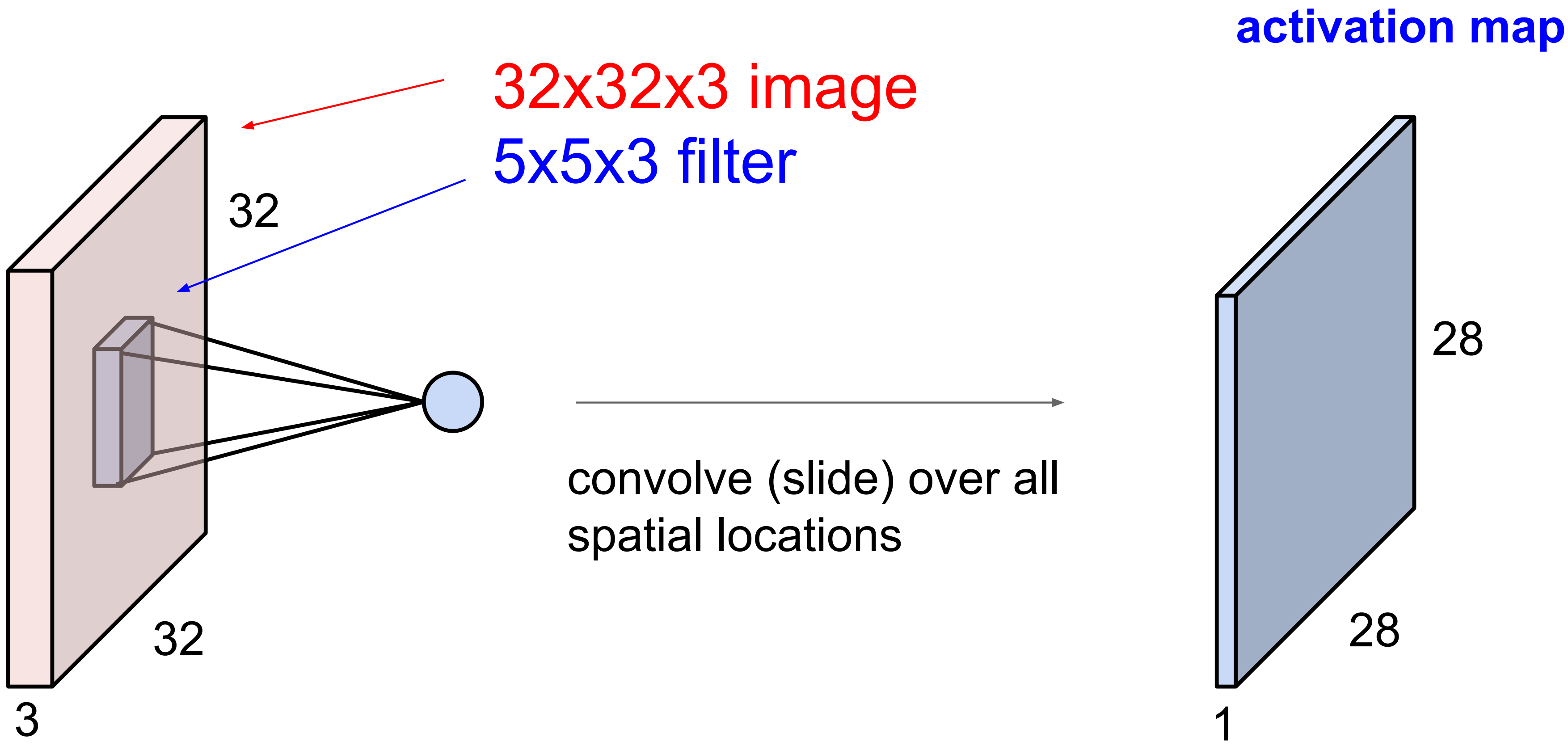
5x5x3 filter w



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)

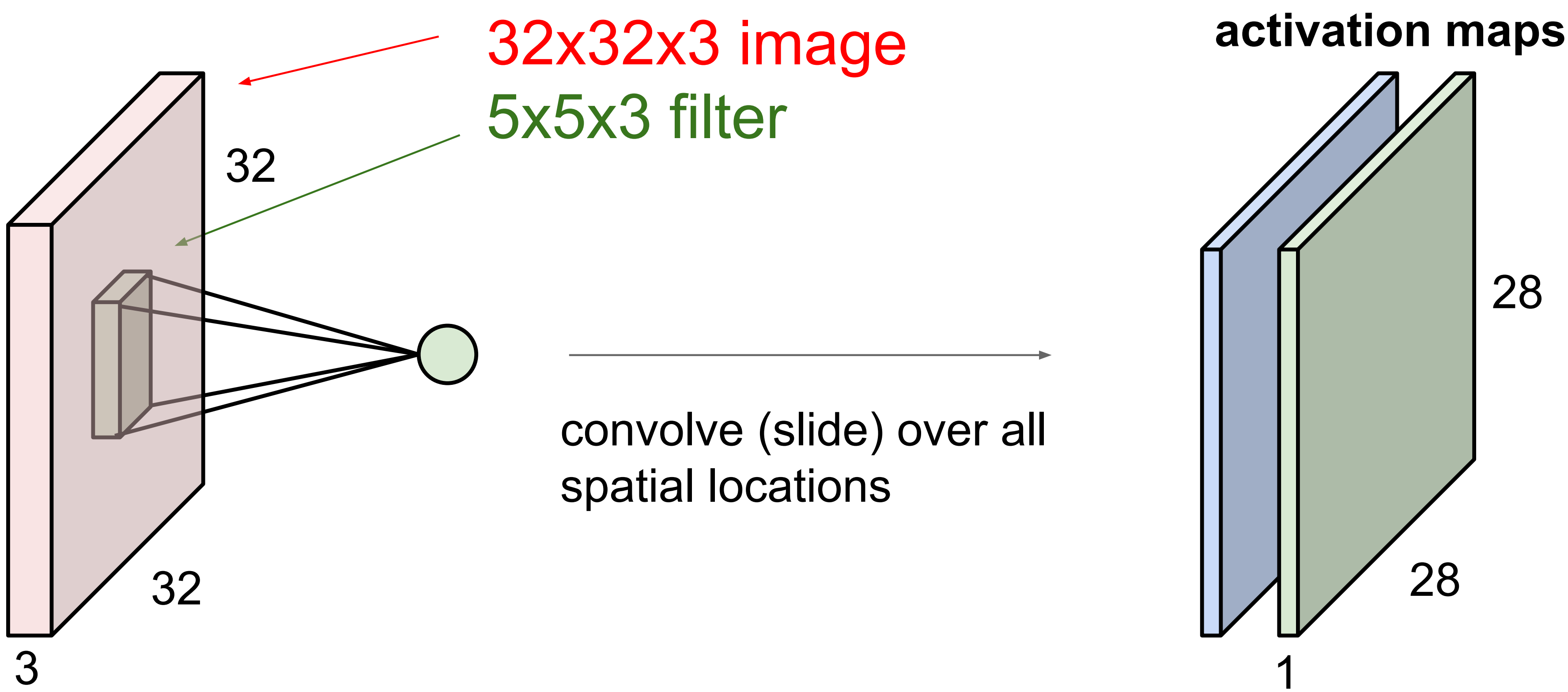
$$w^T x + b$$

Convolution Layer

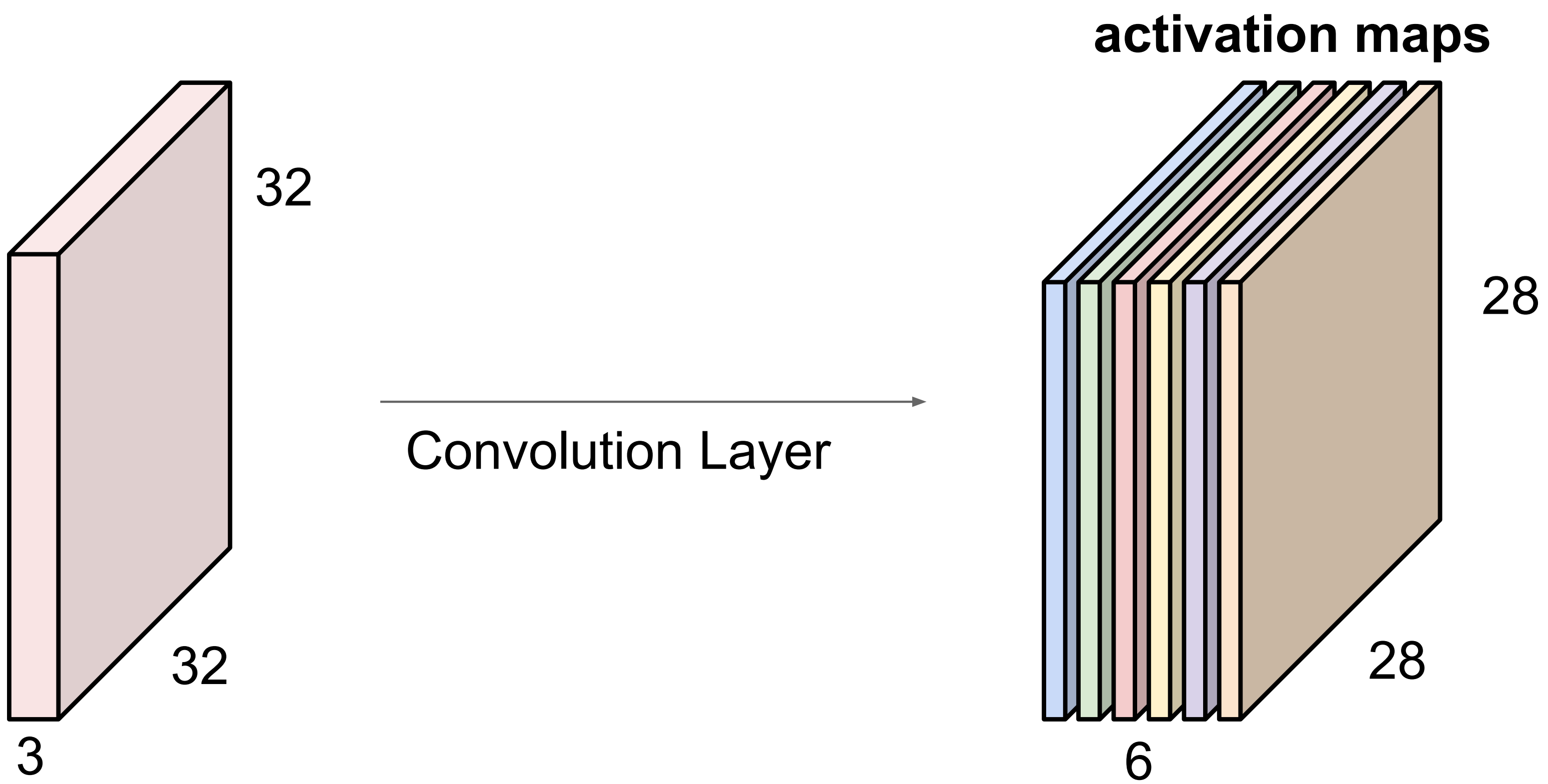


Convolution Layer

consider a second, green filter

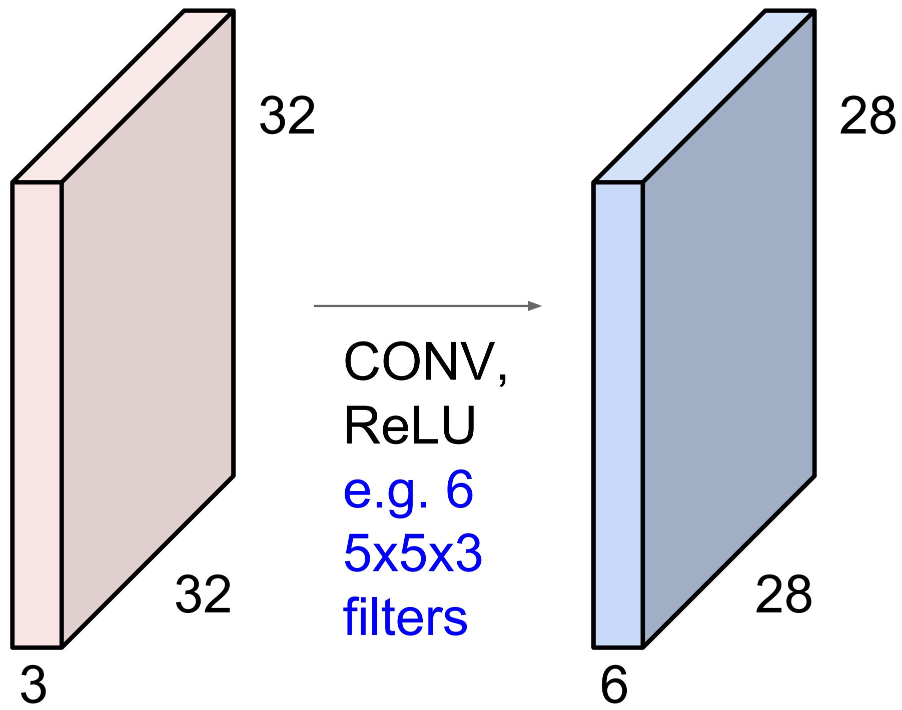


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

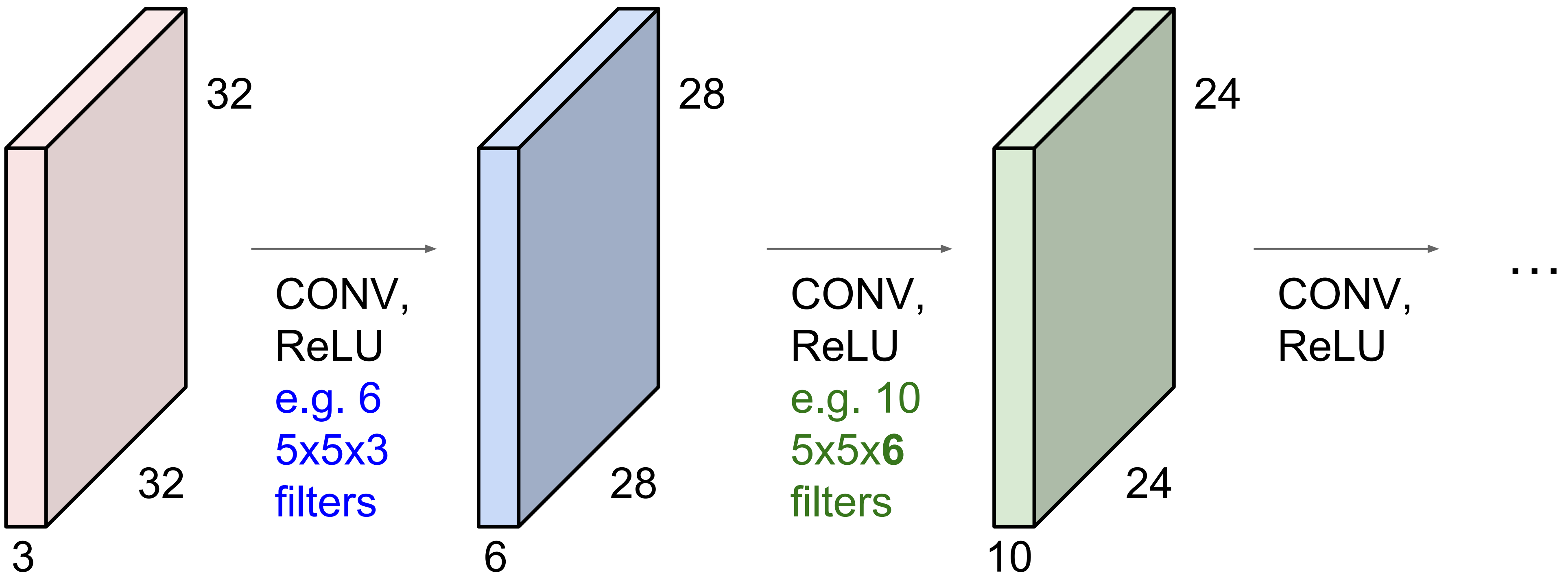


We stack these up to get a "new image" of size 28x28x6!

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

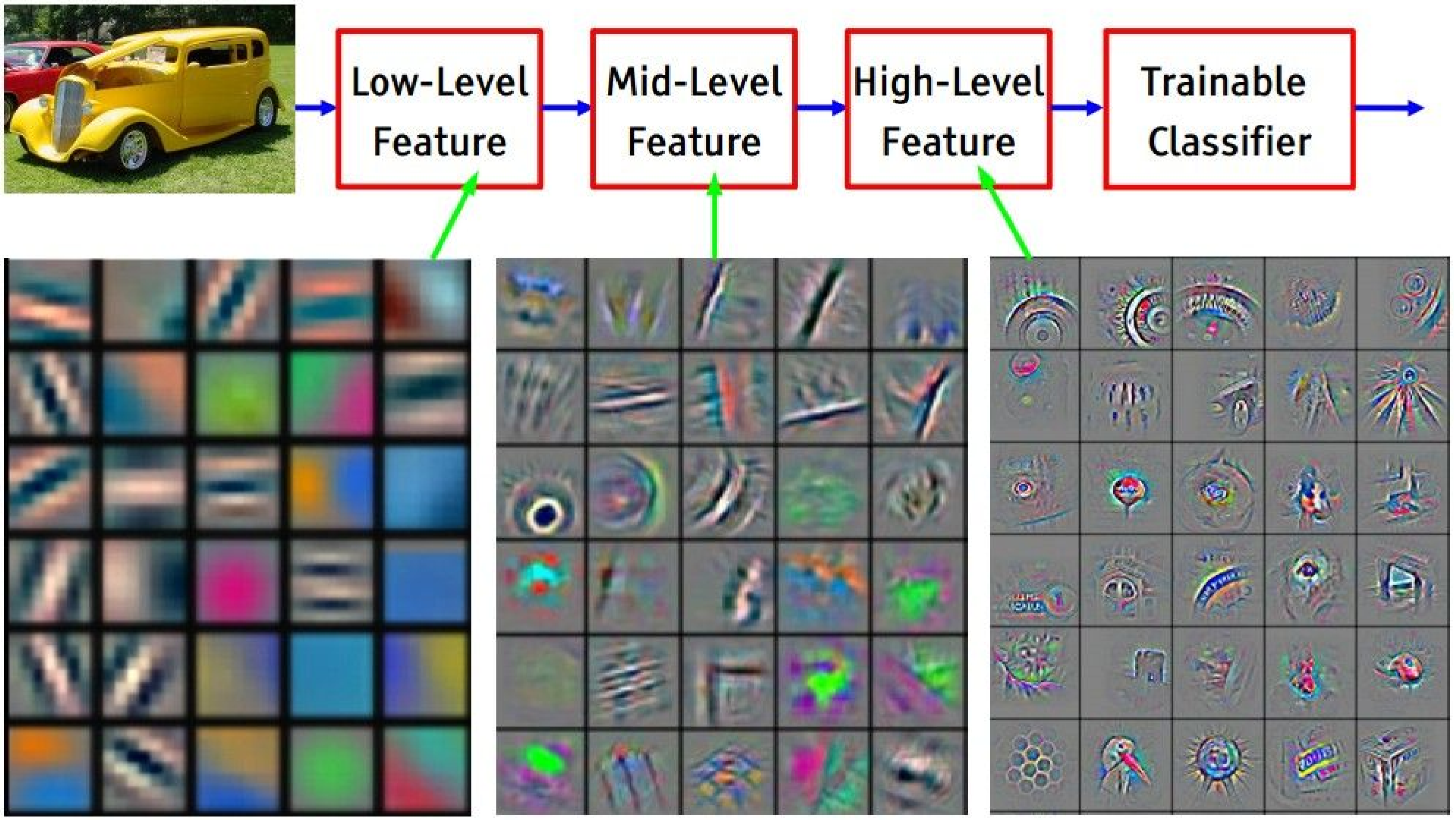


Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



Preview

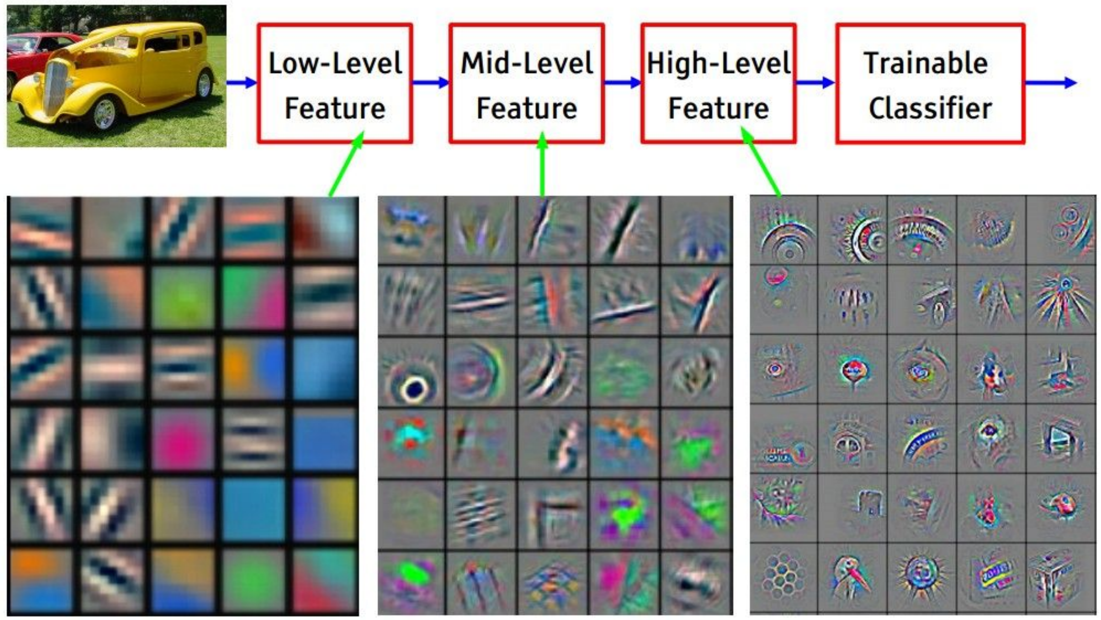
[From recent Yann LeCun slides]



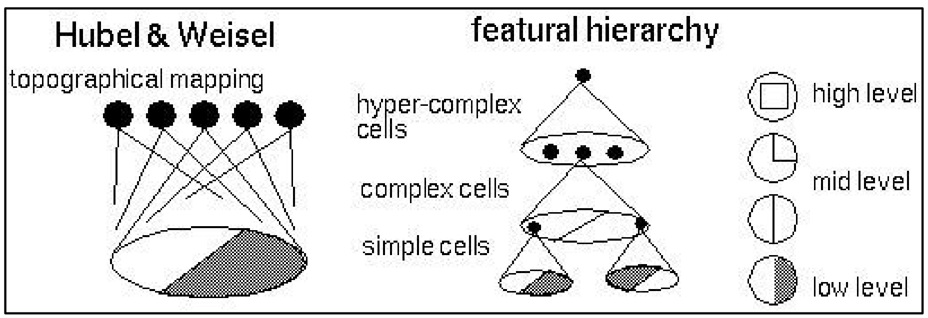
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

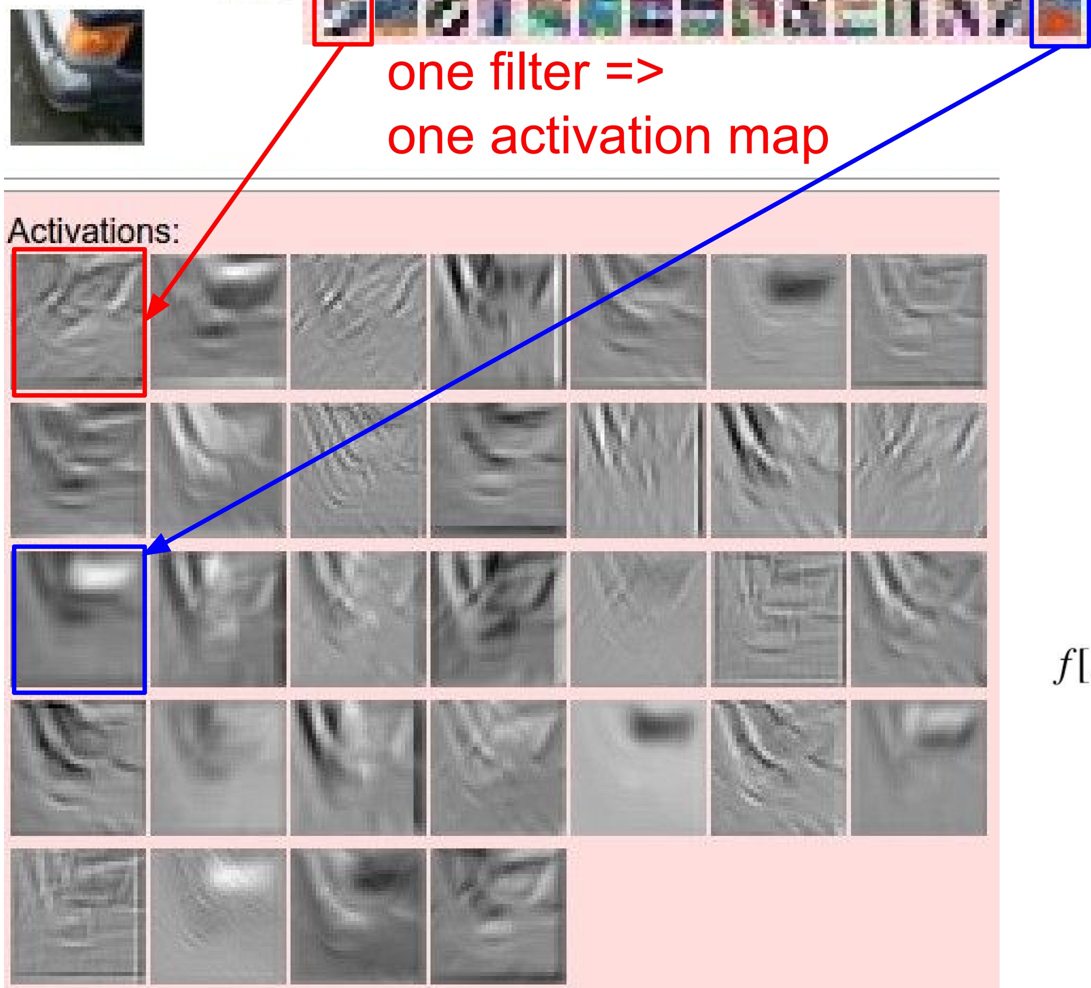
Preview

[From recent Yann LeCun slides]



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



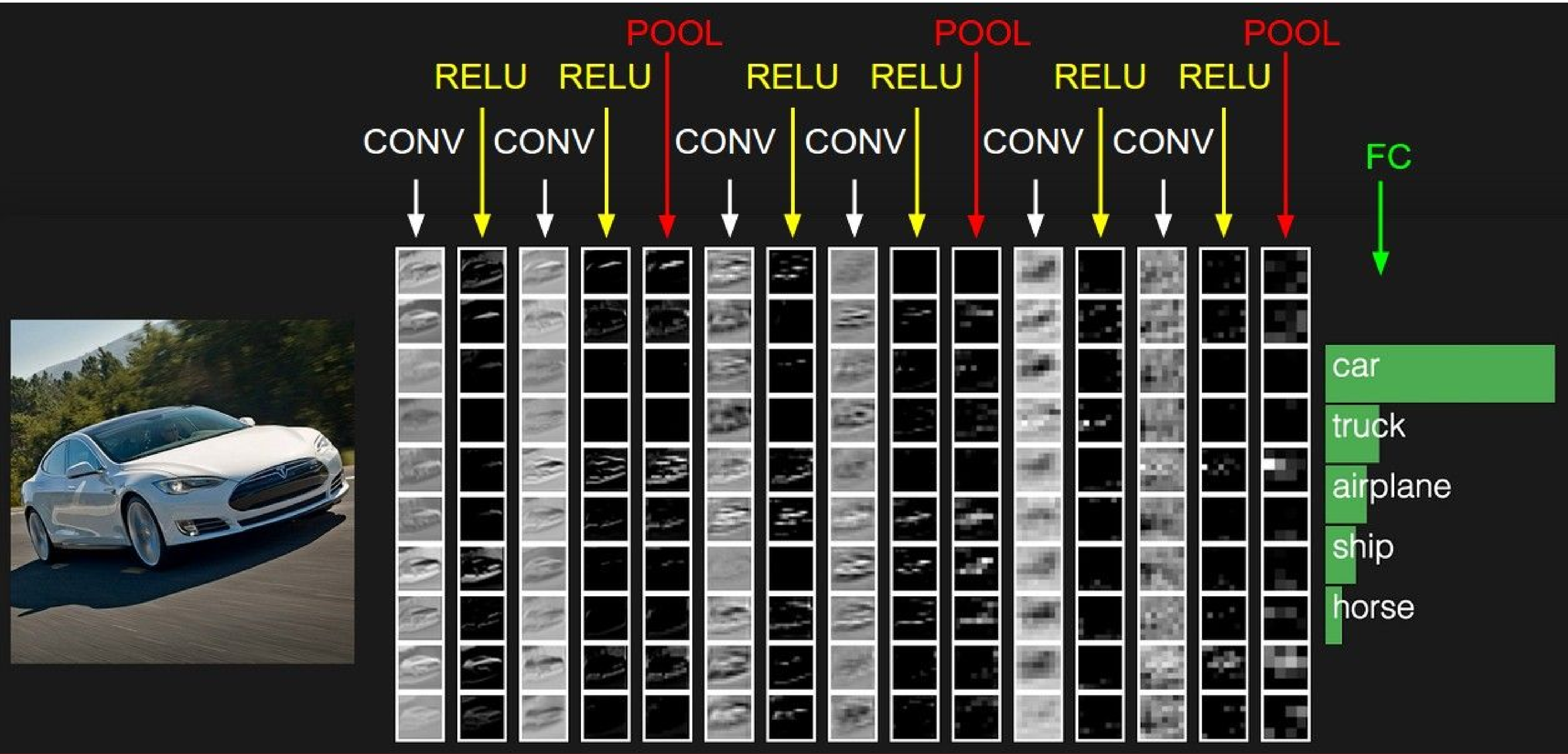


example 5x5 filters
(32 total)

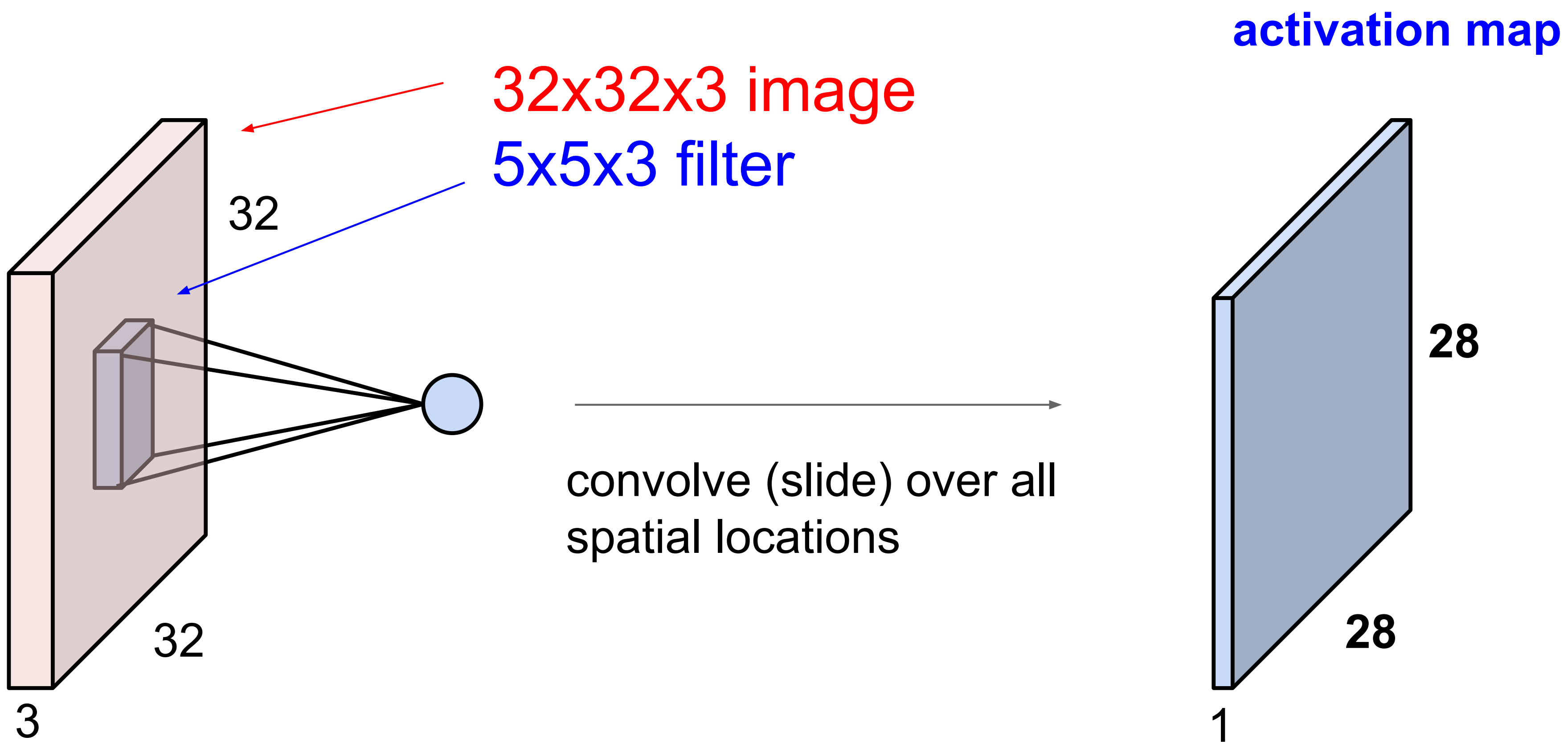
We call the layer convolutional because it is related to convolution of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1, n_2] \cdot g[x-n_1, y-n_2]$$

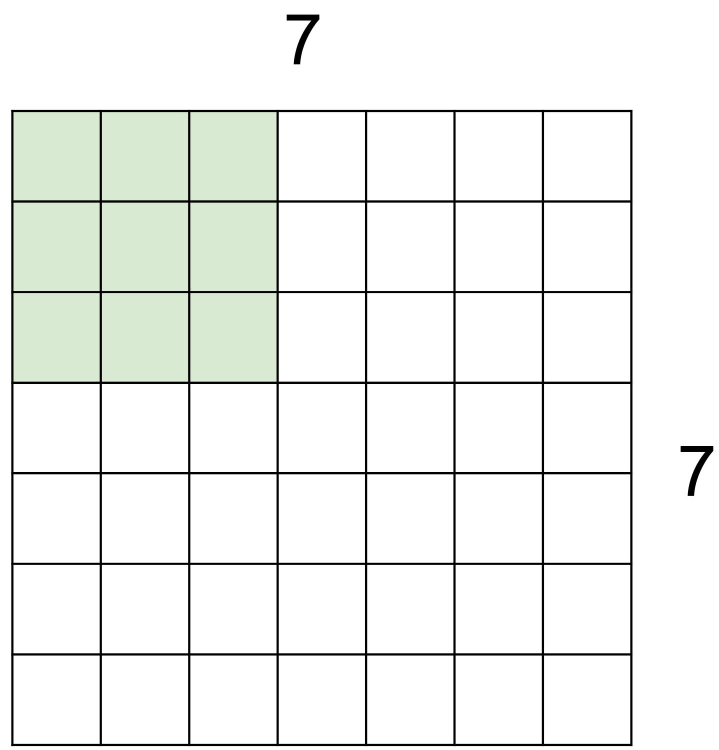
elementwise multiplication and sum of a filter and the signal (image) preview:


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

A closer look at spatial dimensions:

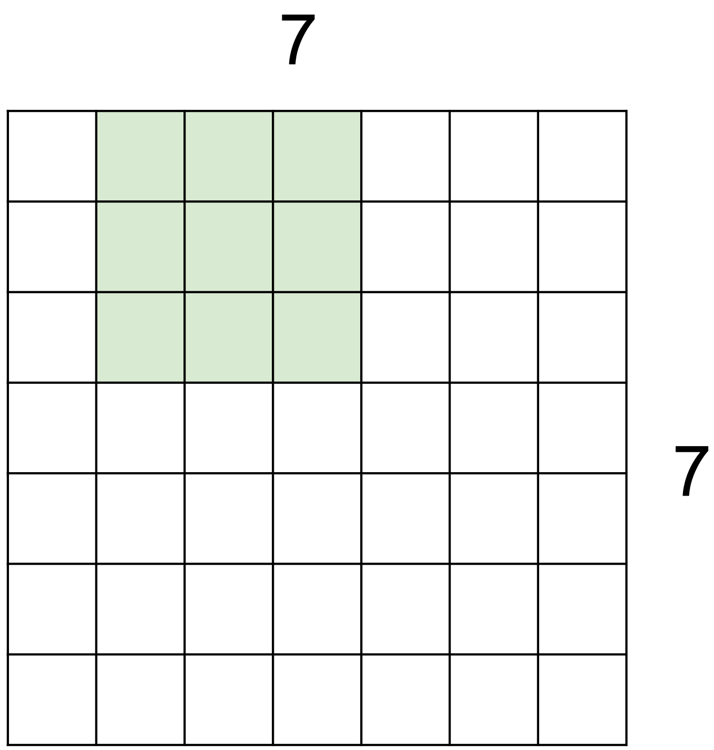


A closer look at spatial dimensions:



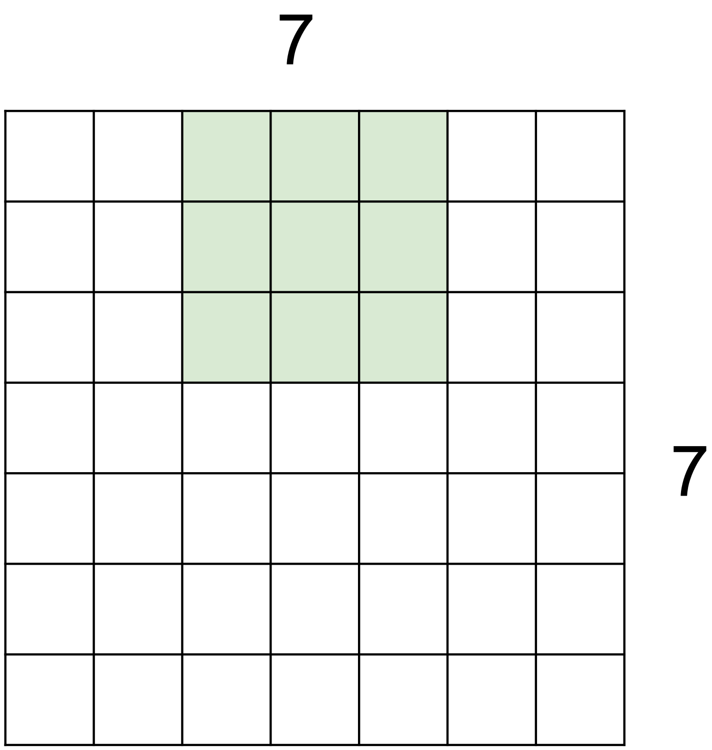
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



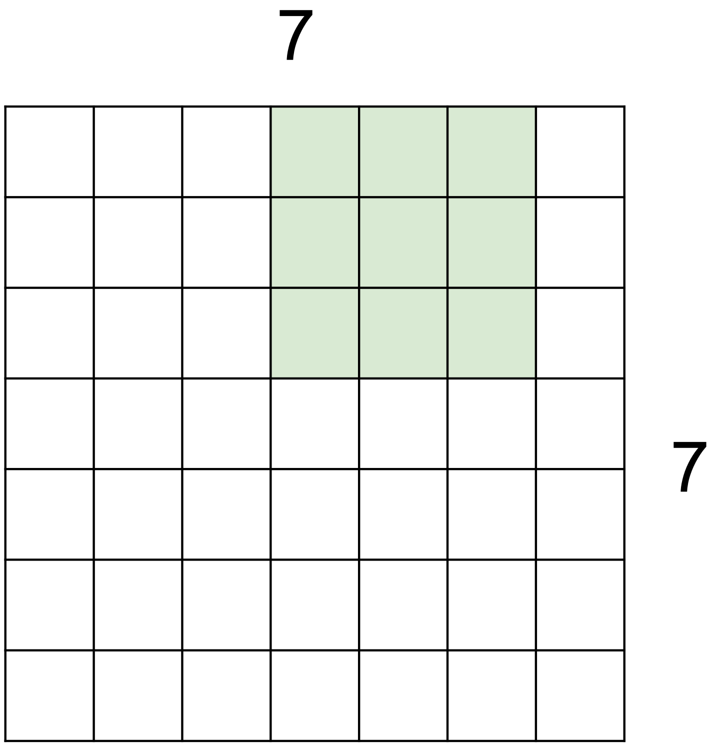
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



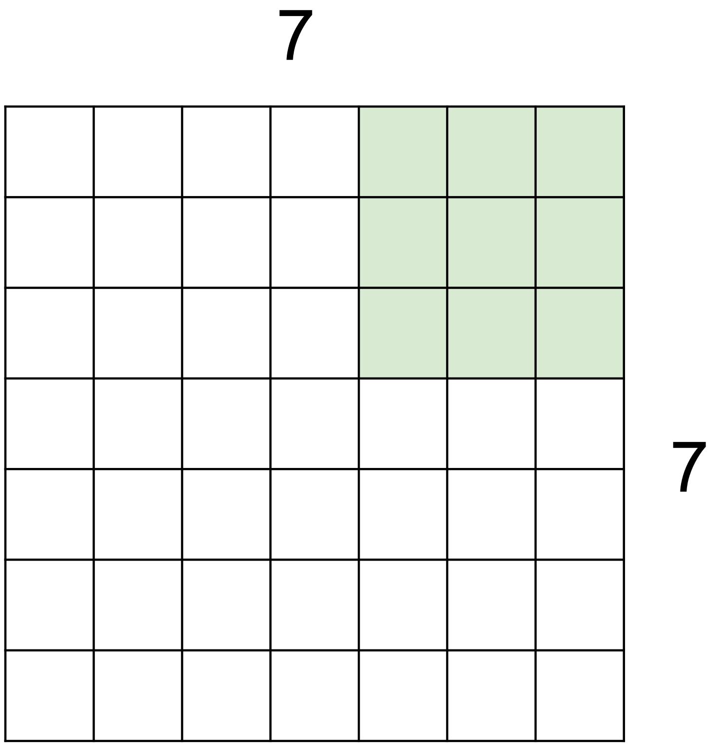
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

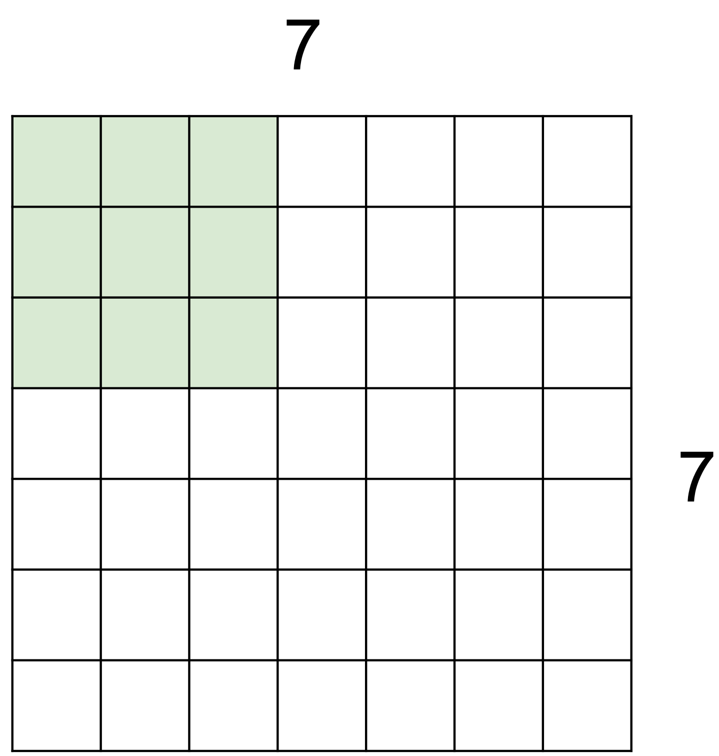
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

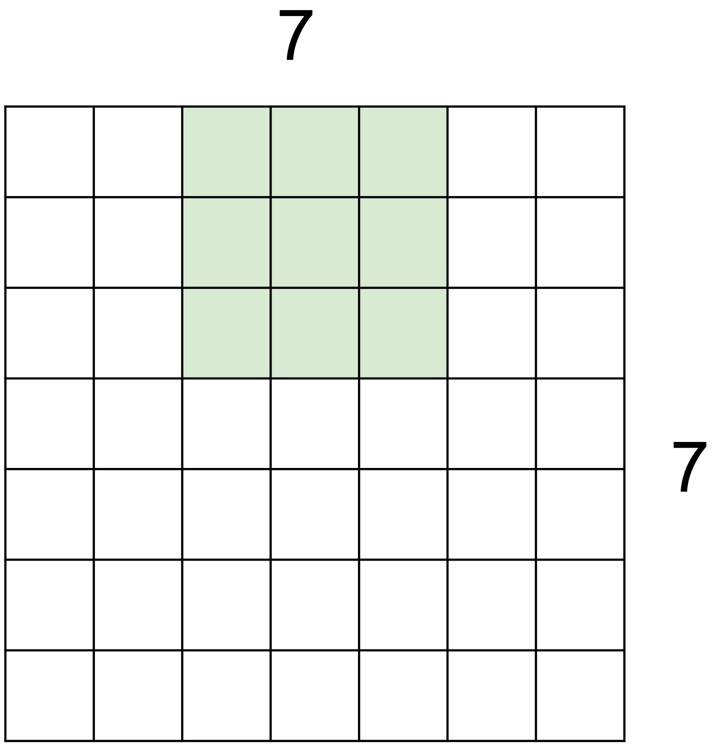
=> 5x5 output

A closer look at spatial dimensions:



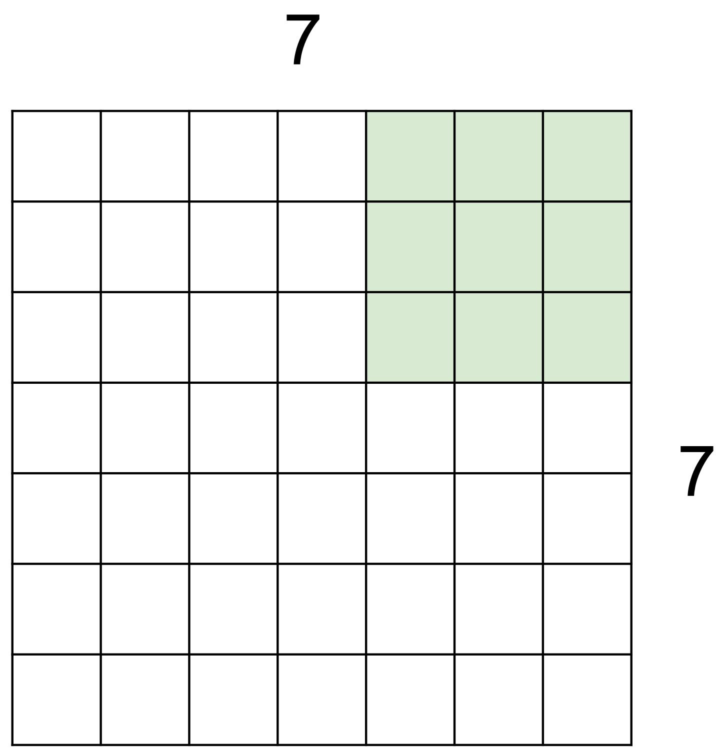
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



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

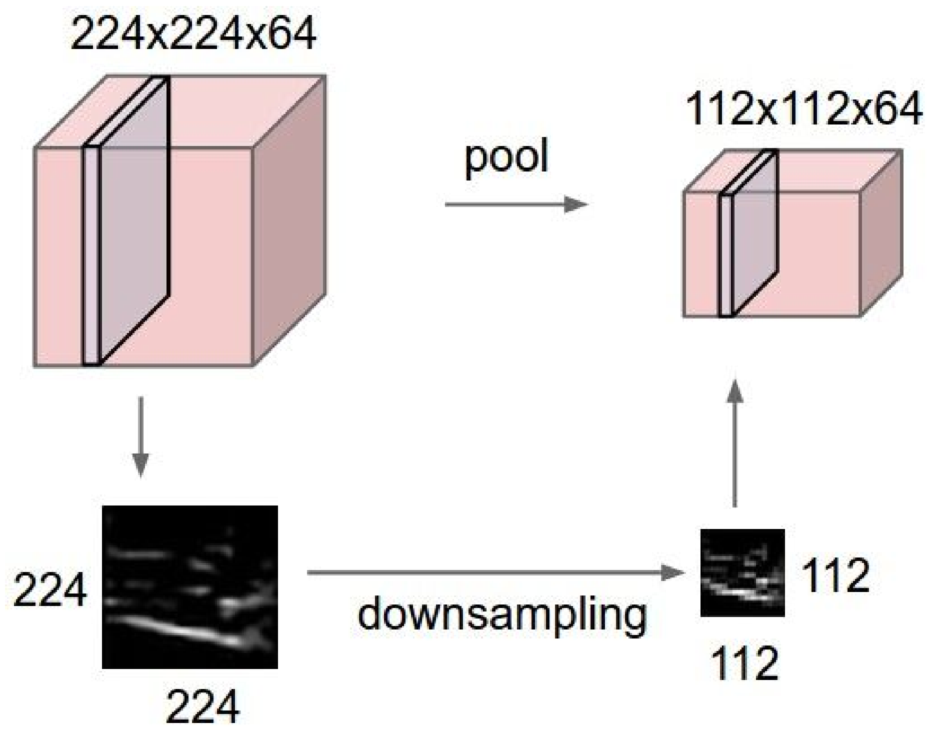
A closer look at spatial dimensions:



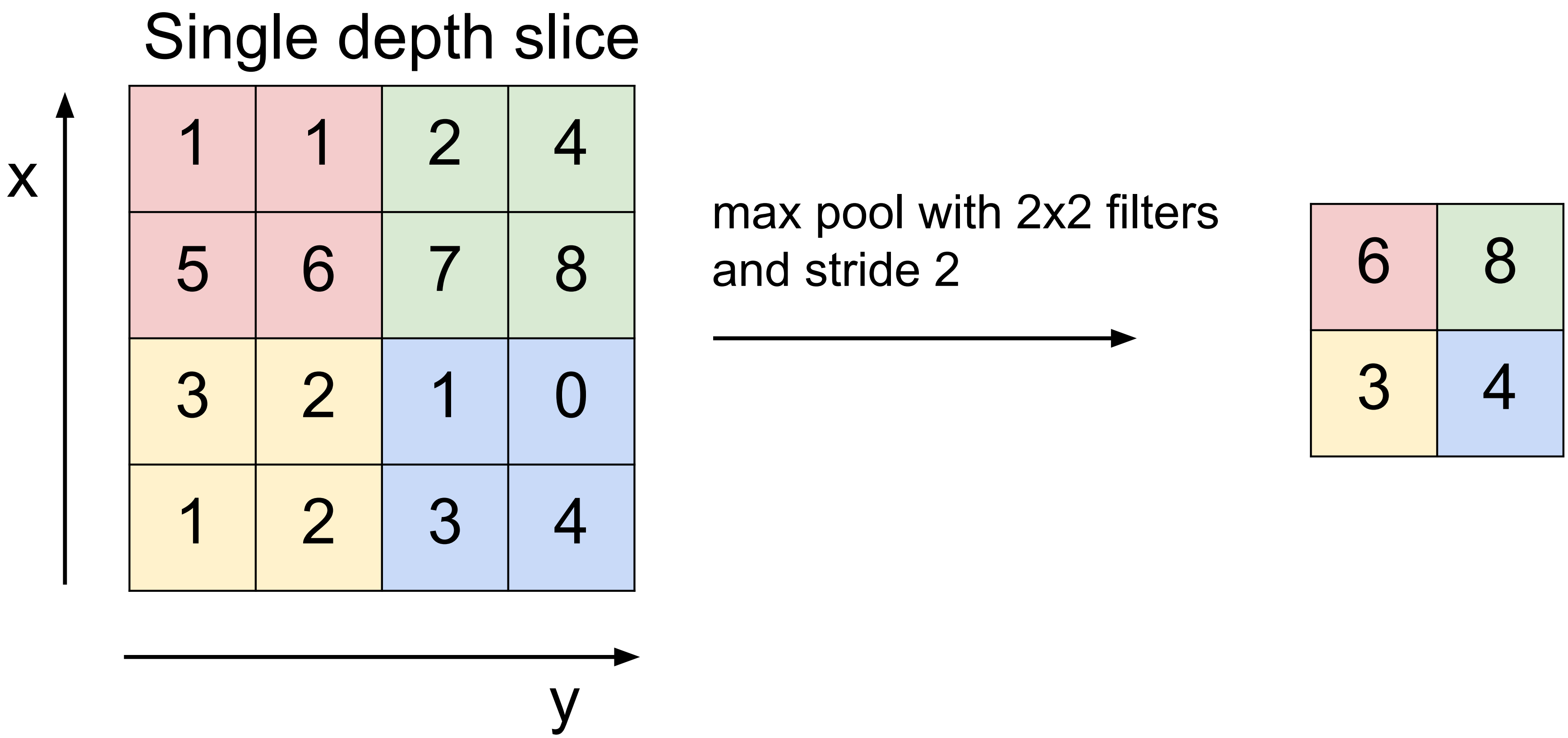
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



MAX POOLING


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

What is a word embedding?

Suppose you have a dictionary of words.

The i^{th} word in the dictionary is represented by an embedding:

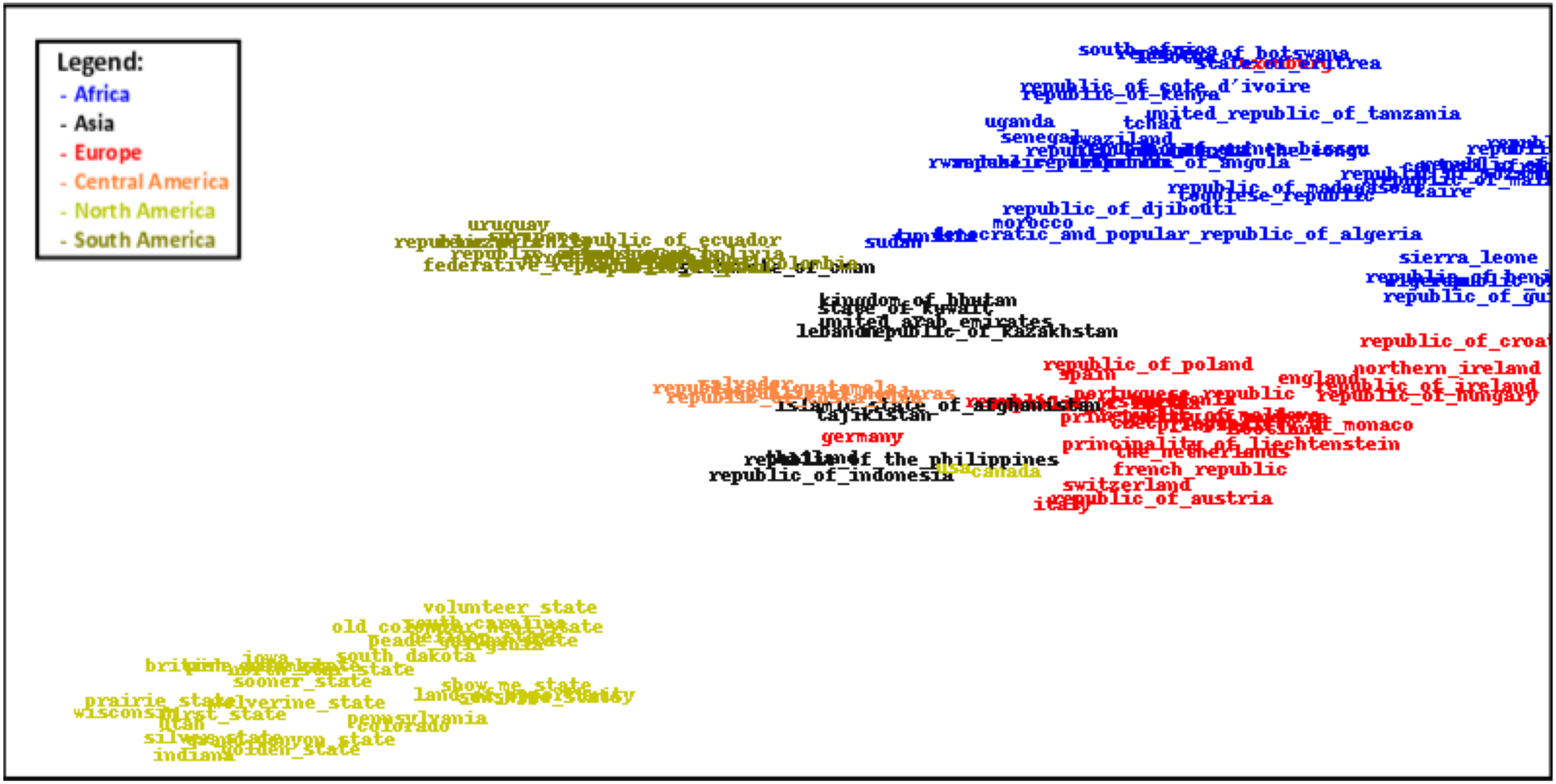
$$w_i \in \mathbb{R}^d$$

i.e. a d-dimensional vector, which is **learnt!**

- d typically in the range 50 to 1000.
- Similar words should have similar embeddings (share latent features).
- Embeddings can also be applied to *symbols* as well as words (e.g. Freebase nodes and edges).
- Discuss later: can also have embeddings of phrases, sentences, documents, or even other modalities such as images.

Learning an Embedding Space

Example of Embedding of 115 Countries (Bordes et al., '11)



How well can we do with a simple CNN?

Collobert-Weston style CNN with pre-trained embeddings from `word2vec`