Lecture 18-2

Neural networks and CNN

Administrative

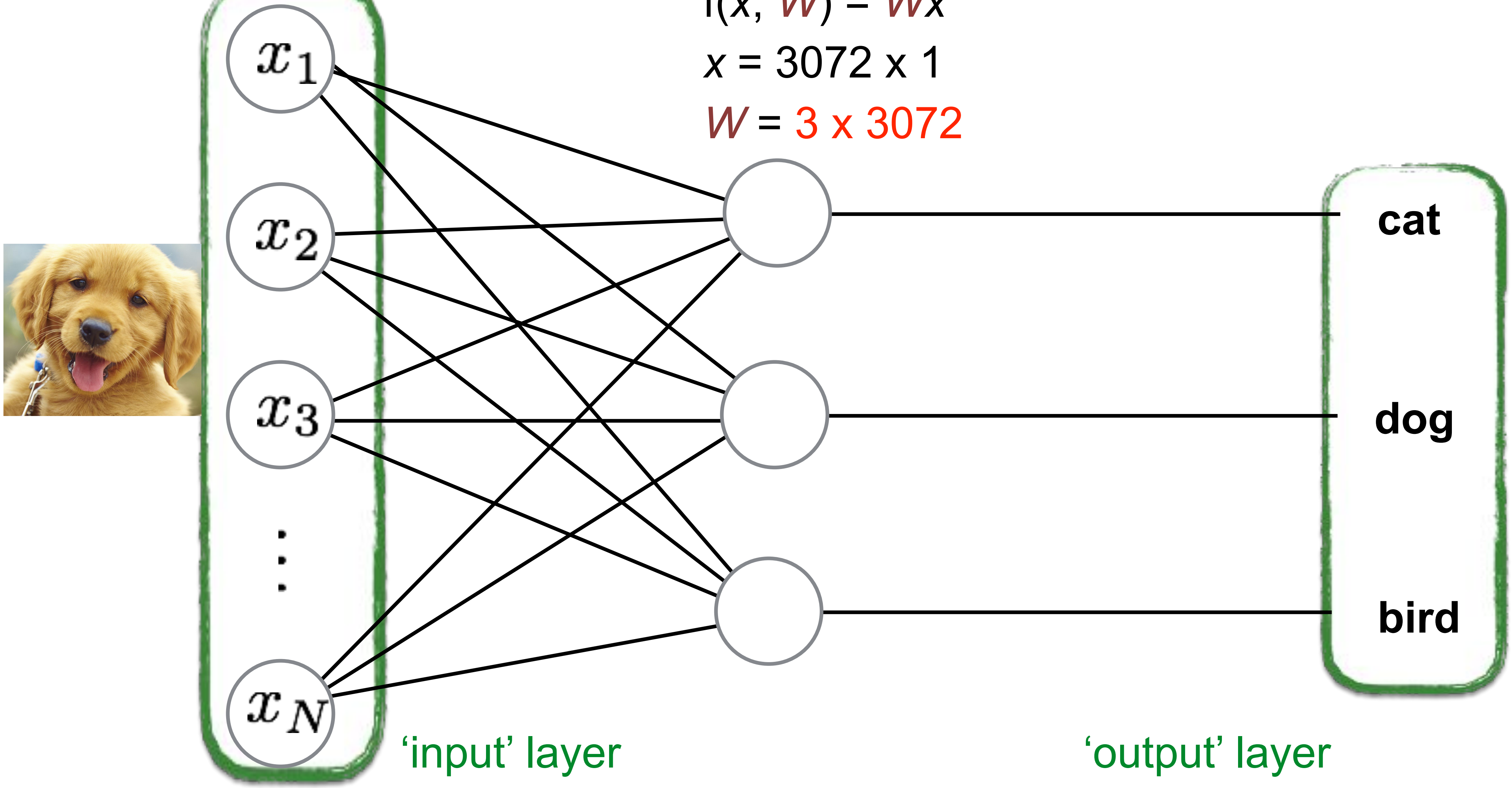
- A5 & A6 (bonus) are out
- Final Exam on 6/9 at 2:30 pm
- Makeup exam on 6/6
- Exam practice is out

Recall: Linear Model

$f(x, W) = Wx$

$x = 3072 \times 1$

$W = 3 \times 3072$

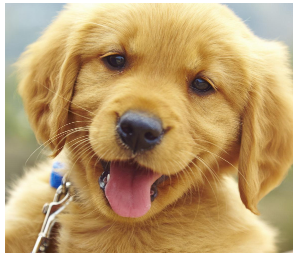


Recall: Softmax Classifier

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

We need a mechanism to convert or normalize the output into probability range [0, 1]

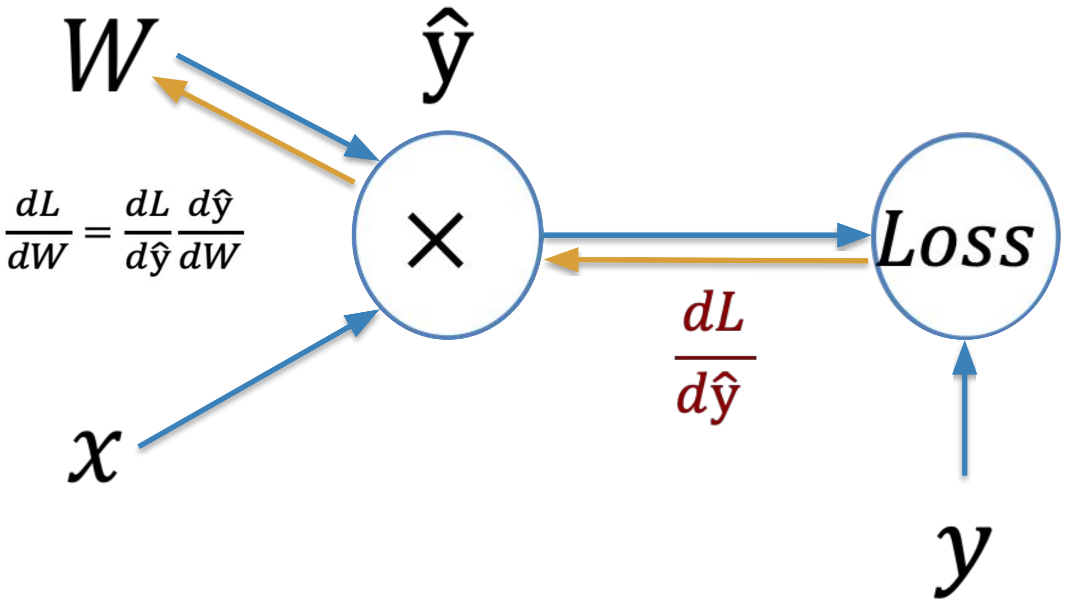
Recall: SOFTMAX: $Prob[f(x_i, W) == k] = \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}}$



$\begin{bmatrix} 3.2 \\ 5.1 \\ -1.7 \end{bmatrix}$ → exp → $\begin{bmatrix} 24.5 \\ 164 \\ 0.18 \end{bmatrix}$ → norm → $\begin{bmatrix} 0.13 \\ 0.87 \\ 0.00 \end{bmatrix}$ ↔ loss ↔ $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ dog / cat / bird

model outputs — probabilities — correct outputs

Recall: Gradient Descent through Backprop

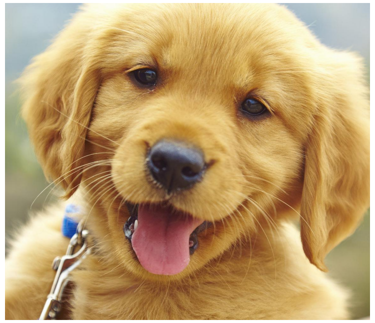


$$\hat{y} = Wx$$
$$L = Loss(\hat{y}, y)$$

$$\frac{dL}{dW} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dW}$$

Key Insight:
- visualize the computation as a graph flow
- Compute the forward pass to calculate the loss.
- Compute all gradients for each pair of nodes backwards

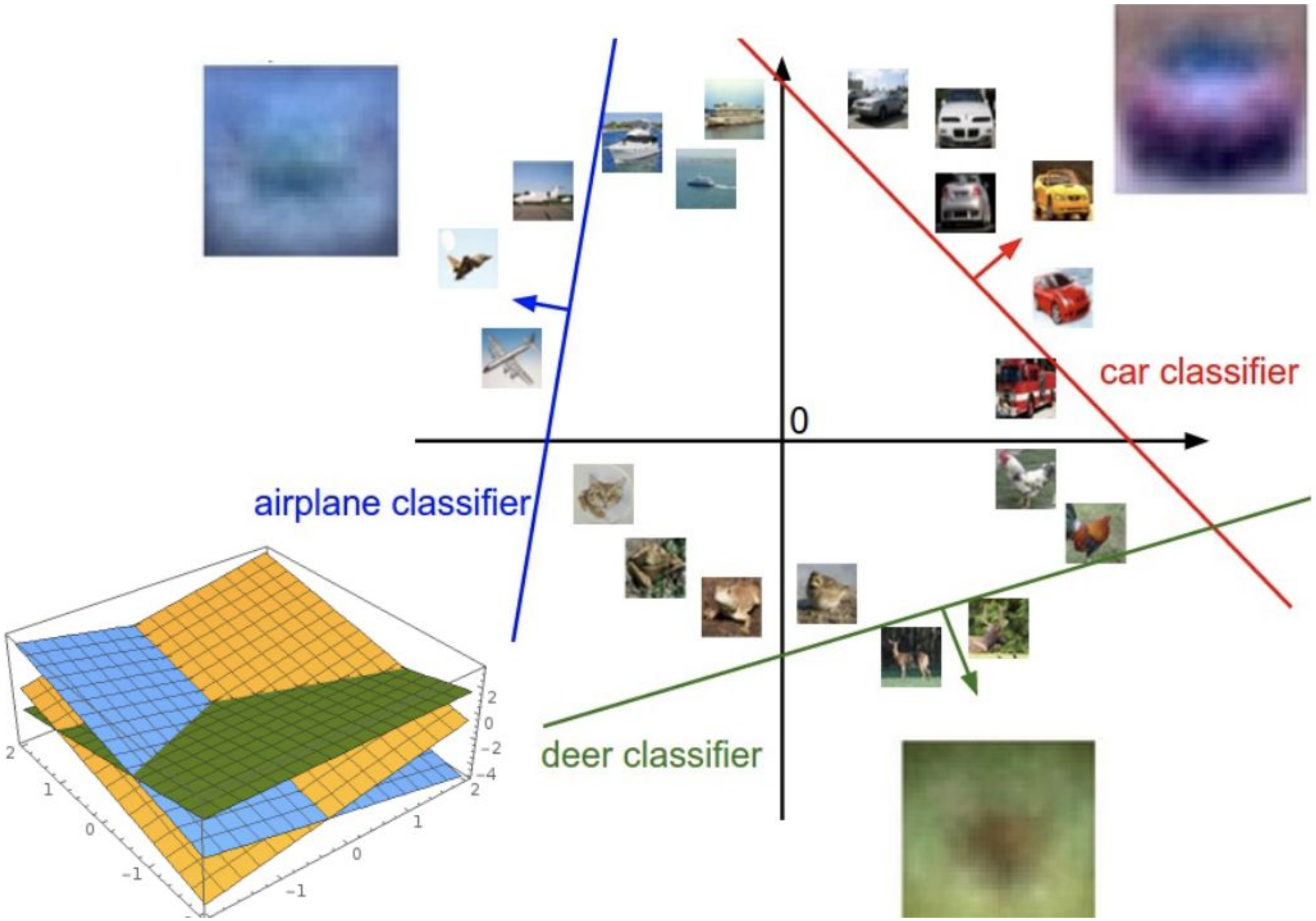
Recall: we can featurize images into a vector



Raw pixels
Raw pixels + (x,y)
PCA
LDA
BoW
BoW + spatial pyramids

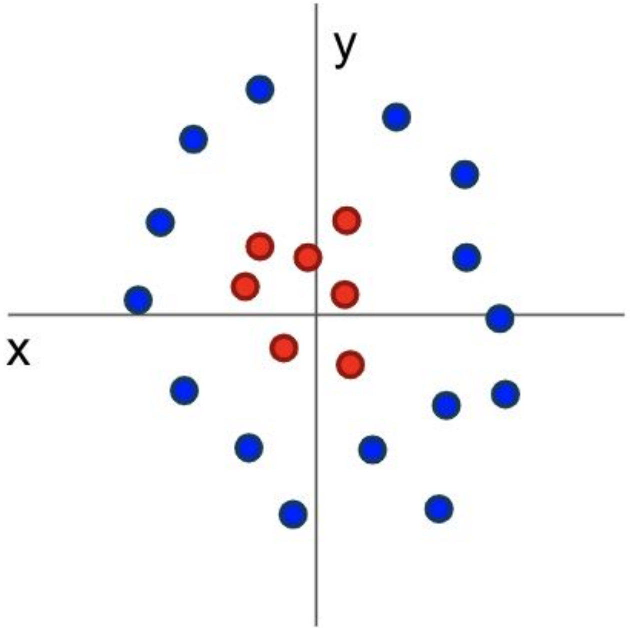
Image
Vector

Interpreting the linear weights geometrically



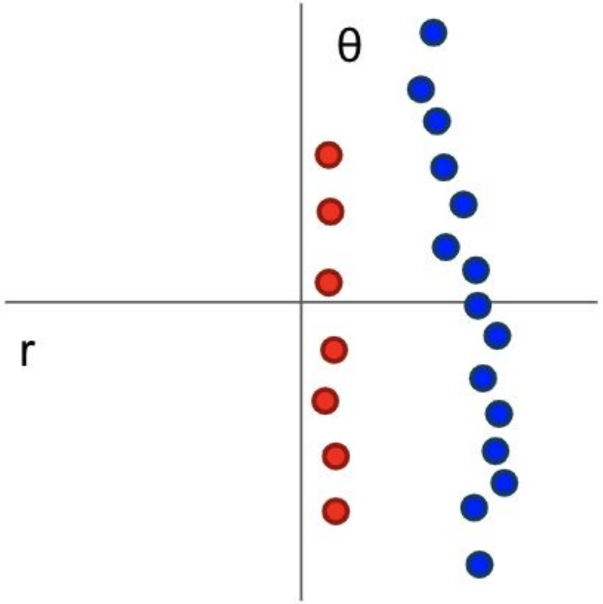
Plot Created using Wolfram Cloud

Features sometimes might not be linearly separable

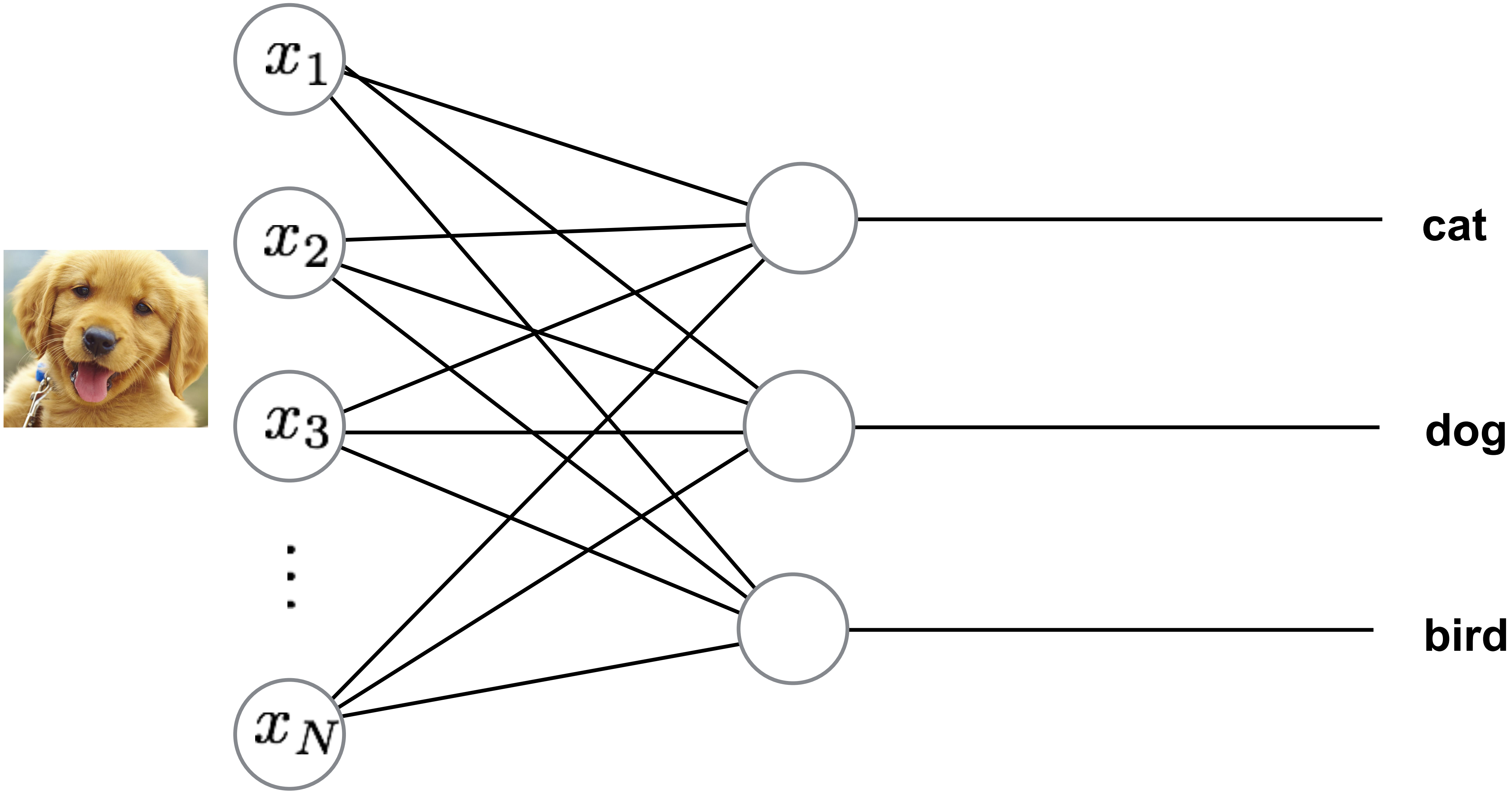


$$f(x,y) = (r(x,y), \theta(x,y))$$

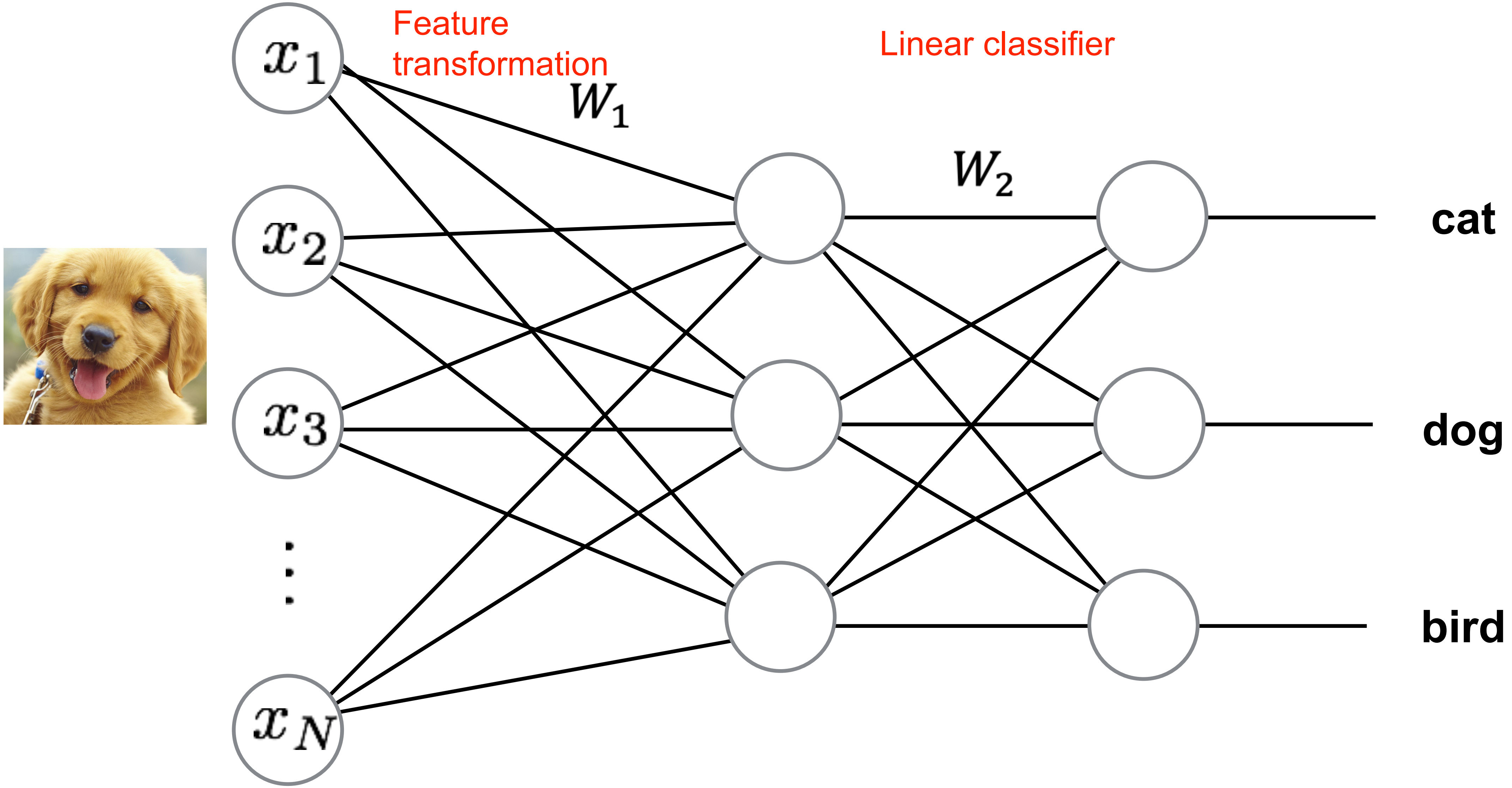
but some mappings are!



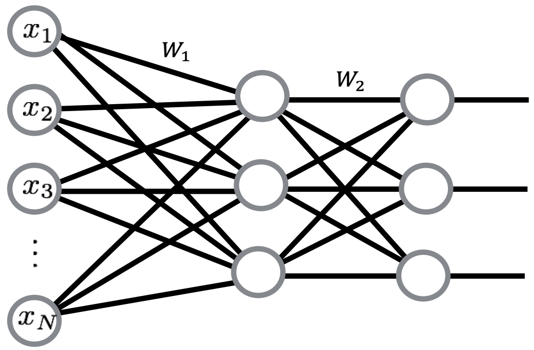
Remember our linear classifier



Let's change the features by adding another layer



2-layer network: mathematical formula



- Linear classifier: $y = Wx$
- 2-layer network: $y = W_2 \cdot \text{binarize}(W_1 x), \quad \text{where} \quad \text{binarize}(W_1 x) = \begin{cases} 1 & \text{if } W_1 x > 0 \\ 0 & \text{otherwise} \end{cases}$
- 3-layer network: $y = W_3 \cdot \text{binarize}(W_2 \cdot \text{binarize}(W_1 x))$

The number of layers is a new hyperparameter!

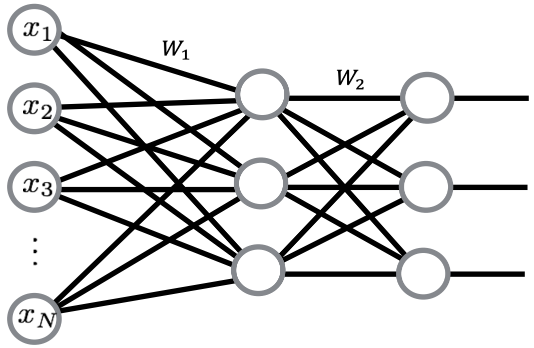
2-layer network: mathematical formula

- Linear classifier: $y = Wx$
- 2-layer network: $y = W_2 \cdot \text{binarize}(W_1 x), \quad \text{where} \quad \text{binarize}(W_1 x) = \begin{cases} 1 & \text{if } W_1 x > 0 \\ 0 & \text{otherwise} \end{cases}$

We know the size of *x* = 1 x 3072 and y = 10 x 1, so what are $\boldsymbol{W_1}$ and $\boldsymbol{W_2}$

$$W_1 = h \times 3072 \qquad W_2 = 10 \times h$$

h is a new hyperparameter!



2-layer network: mathematical formula

- Linear classifier: $y = Wx$

- 2-layer network: $y = W_2 \cdot \text{binarize}(W_1 x), \quad \text{where} \quad \text{binarize}(W_1 x) = \begin{cases} 1 & \text{if } W_1 x > 0 \\ 0 & \text{otherwise} \end{cases}$

Why is the binarize necessary? Let's see what happen when we remove it:

$$y = W_2 W_1 x = Wx$$

Where: $W = W_2 W_1$

Activation is necessary to go from linear to non-linear models

2-layer network: mathematical formula

- Linear classifier: $y = Wx$

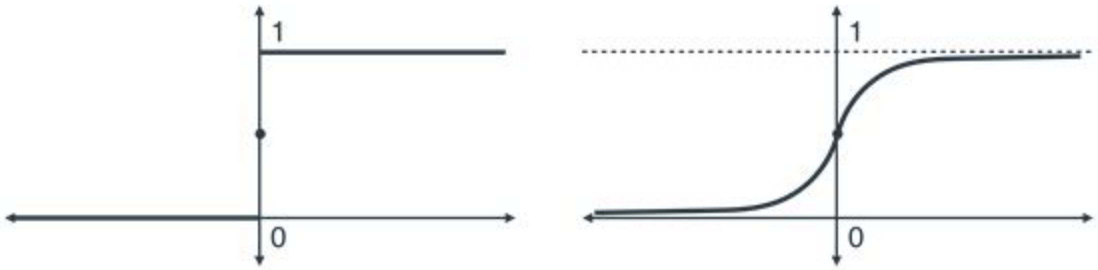
- 2-layer network: $y = W_2 \, \text{sigmoid}(W_1 x)$

Why is the binarize necessary?

- Neural science inspiration
- Non-differentiable

Let's approximate it with sigmoid

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



2-layer network: mathematical formula

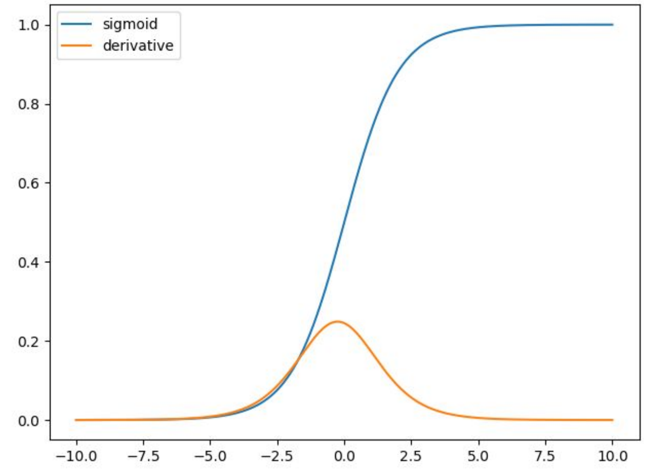
- Linear classifier: $y = Wx$

- 2-layer network: $y = W_2 \, \mathrm{ReLU}(W_1 x)$

Why is the sigmoid necessary?

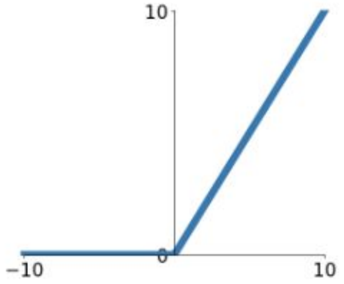
- Vanishing gradient

Let's replace it with ReLU



ReLU

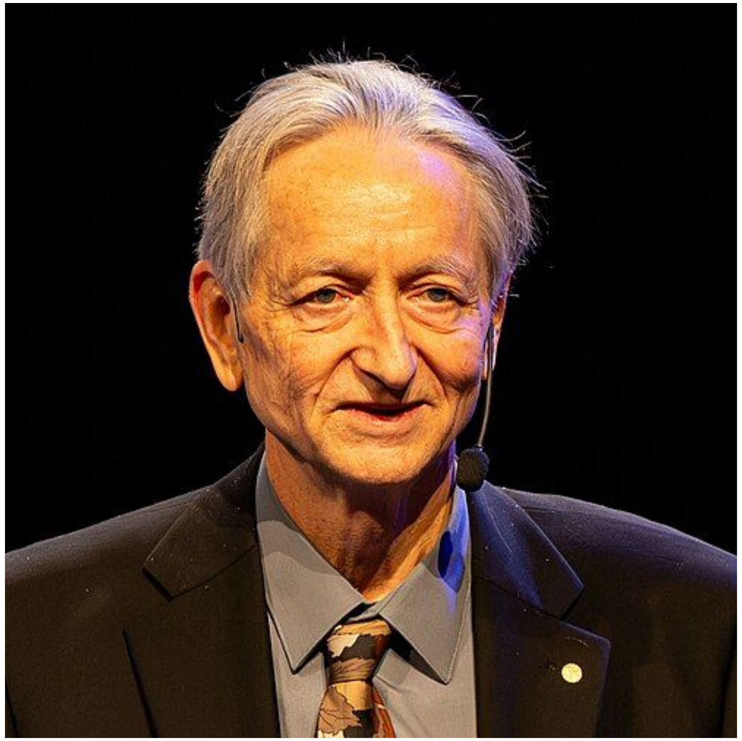
$\max(0, x)$



ReLU v.s. Sigmoid

ReLU is not from nowhere

- Connection between ReLU & Sigmoid



Rectified Linear Units Improve Restricted Boltzmann Machines

Vinod Nair VNAIR@CS.TORONTO.EDU
Geoffrey E. Hinton HINTON@CS.TORONTO.EDU
Department of Computer Science, University of Toronto, Toronto, ON M5S 2G4, Canada

Abstract

Restricted Boltzmann machines were developed using binary stochastic hidden units. These can be generalized by replacing each binary unit by an infinite number of copies that all have the same weights but have progressively more negative biases. The learning and inference rules for these "Stepped Sigmoid Units" are unchanged. They can be approximated efficiently by noisy, rectified linear units. Compared with binary units, these units learn features that are better for object recognition on the NORB dataset and face verification on the Labeled Faces in the Wild dataset. Unlike binary units, rectified linear units preserve information about relative intensities as information travels through multiple layers of feature detectors.

1. Introduction

Restricted Boltzmann machines (RBMs) have been used as generative models of many different types of data including labeled or unlabeled images (Hinton et al., 2006), sequences of mel-cepstral coefficients that represent speech (Mohamed & Hinton, 2010), bags of words that represent documents (Salakhutdinov & Hinton, 2009), and user ratings of movies (Salakhutdinov et al., 2007). In their conditional form they can be used to model high-dimensional temporal sequences such as video or motion capture data (Taylor et al., 2006). Their most important use is as learning modules that are composed to form deep belief nets (Hinton et al., 2006).

Appearing in *Proceedings of the 27^th International Conference on Machine Learning*, Haifa, Israel, 2010. Copyright 2010 by the author(s)/owner(s).

1.1. Learning a Restricted Boltzmann Machine

Images composed of binary pixels can be modeled by an RBM that uses a layer of binary hidden units (feature detectors) to model the higher-order correlations between pixels. If there are no direct interactions between the hidden units and no direct interactions between the visible units that represent the pixels, there is a simple and efficient method called "Contrastive Divergence" to learn a good set of feature detectors from a set of training images (Hinton, 2002). We start with small, random weights on the symmetric connections between each pixel i and each feature detector j. Then we repeatedly update each weight, w_{ij}, using the difference between two measured, pairwise correlations

$$\Delta w_{ij} = \epsilon(<v_i h_j>_{data} - <v_i h_j>_{recon}) \quad (1)$$

where ϵ is a learning rate, $<v_i h_j>_{data}$ is the frequency with which visible unit i and hidden unit j are on together when the feature detectors are being driven by images from the training set and $<v_i h_j>_{recon}$ is the corresponding frequency when the hidden units are being driven by reconstructed images. A similar learning rule can be used for the biases.

Given a training image, we set the binary state, h_j, of each feature detector to be 1 with probability

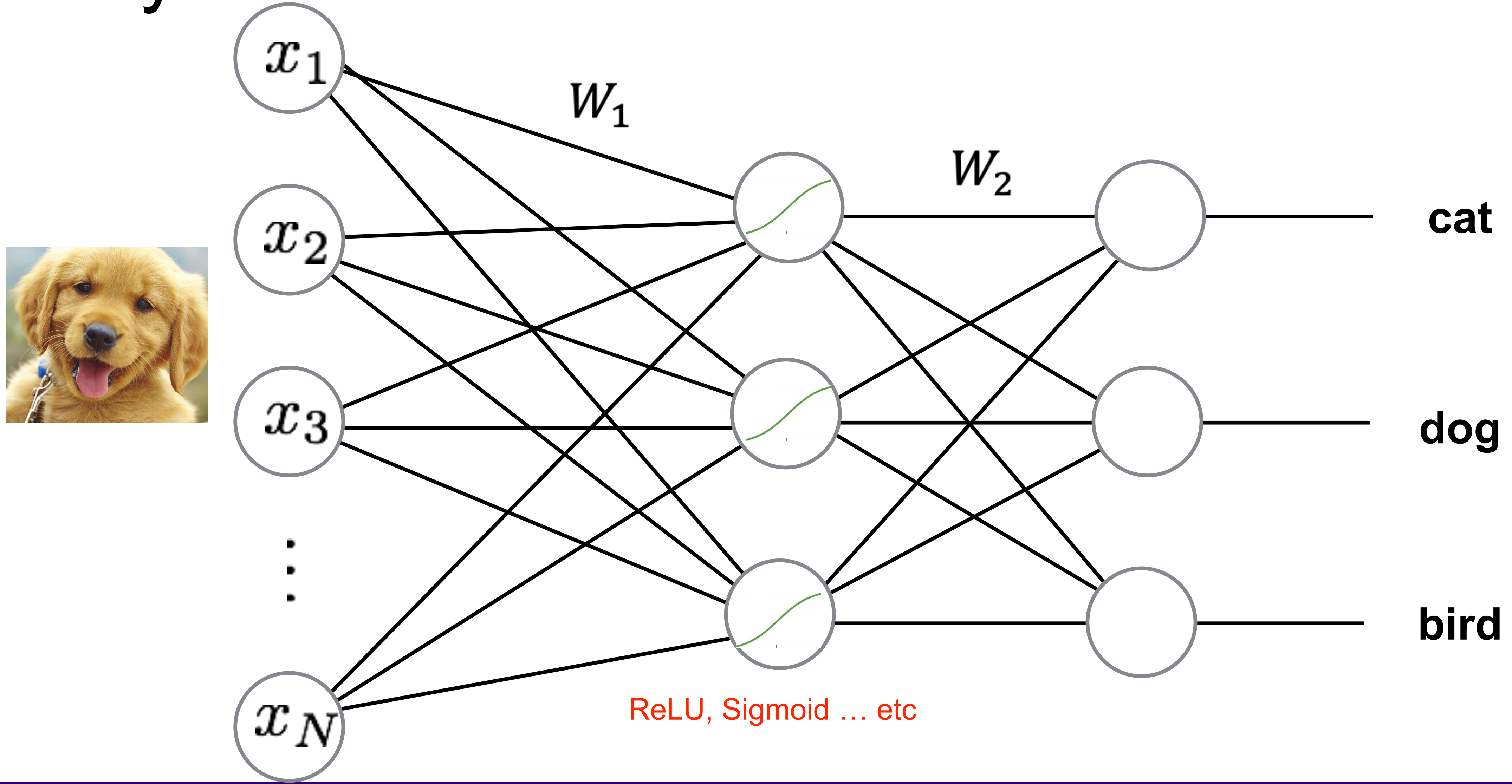
$$p(h_j = 1) = \frac{1}{1 + \exp(-b_j - \sum_{i \in \text{vis}} v_i w_{ij})} \quad (2)$$

where b_j is the bias of j and v_i is the binary state of pixel i. Once binary states have been chosen for the hidden units we produce a "reconstruction" of the training image by setting the state of each pixel to be 1 with probability

$$p(v_i = 1) = \frac{1}{1 + \exp(-b_i - \sum_{j \in \text{hid}} h_j w_{ij})} \quad (3)$$

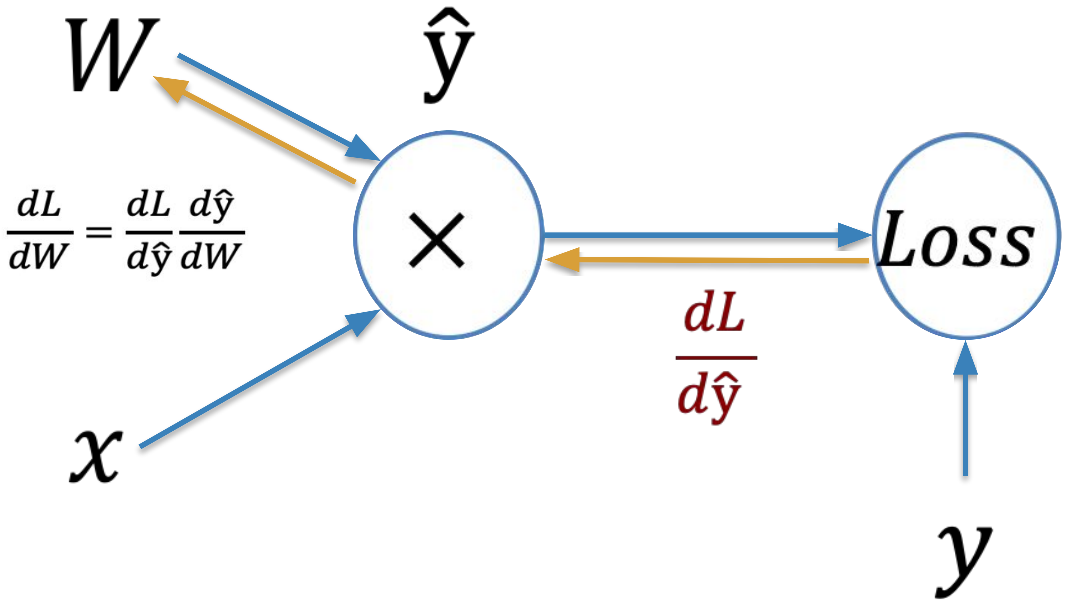
The learned weights and biases implicitly define a probability distribution over all possible binary images via the energy, $E(\mathbf{v}, \mathbf{h})$, of a joint configuration of the

2-layer Neural Network



Take-Home Exercise

Backprop on 2-layer neural network

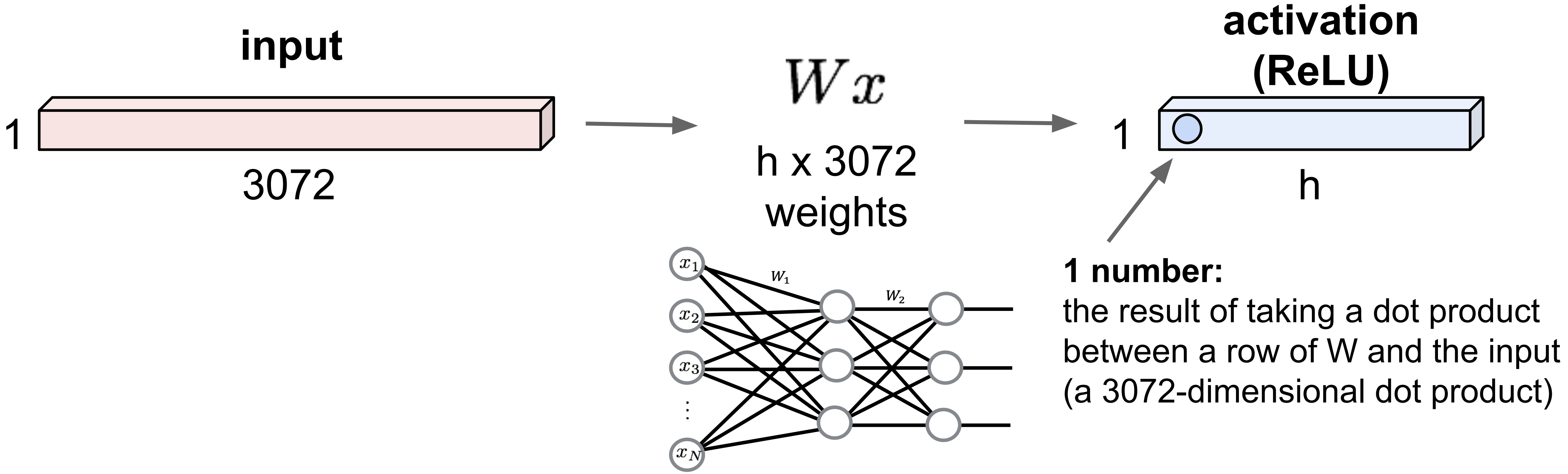


1-layer case

2-layer case ???

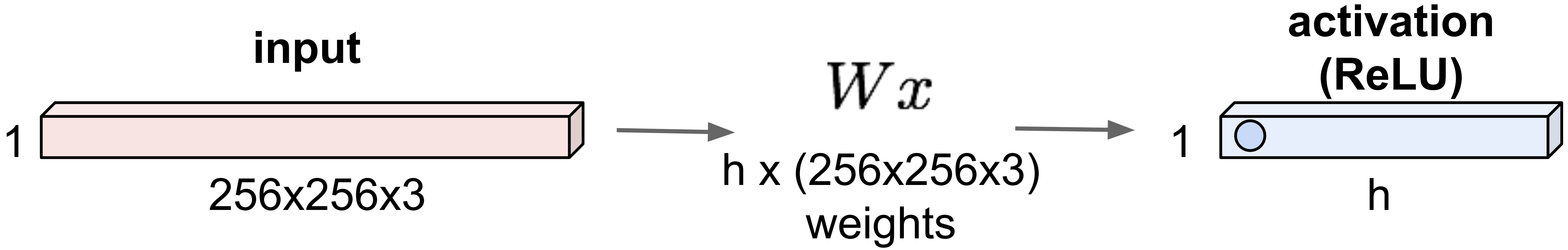
Fully Connected Layer

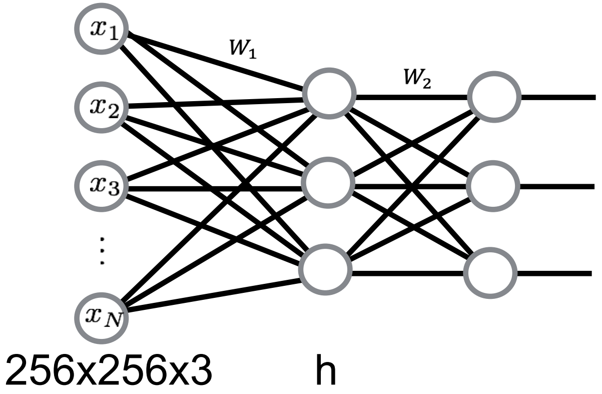
32x32x3 image -> stretch to 3072 x 1



FC Layer Issues

What if we are processing higher resolution image?

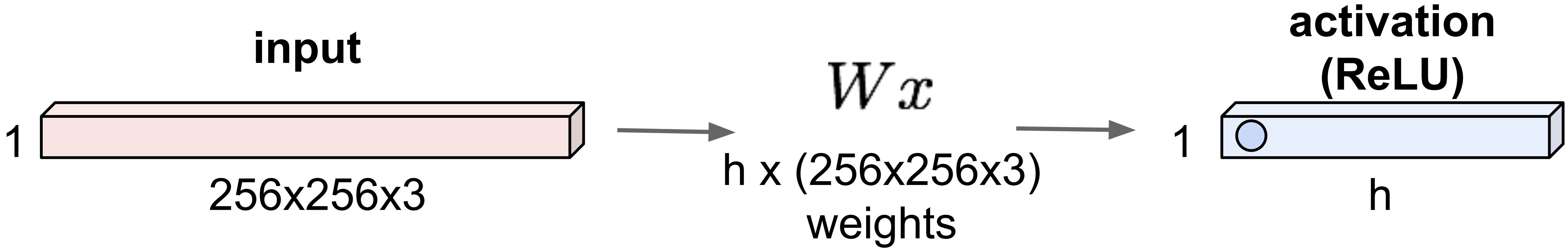




Q: If h=128x128x3, how many parameters of a single layer?

FC Layer Issues

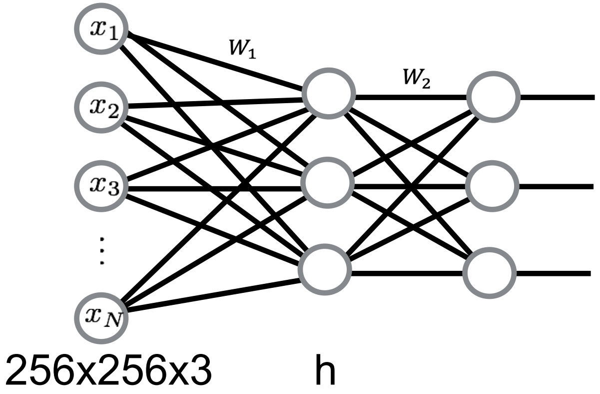
What if we are processing higher resolution image?



Q: If h=128x128x3, how many parameters of a single layer?

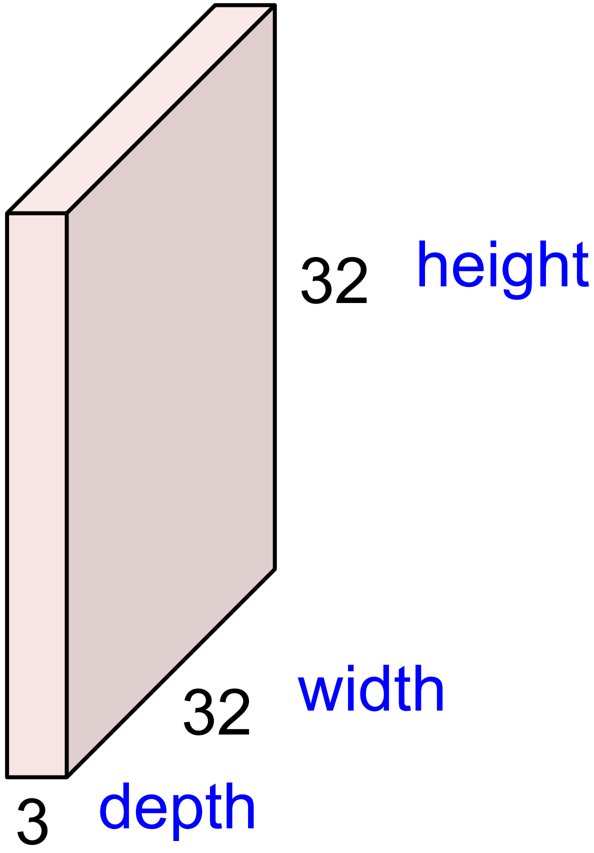
A: (256x256x3)x(128x128x3) ≈ 9.6 B

Too large to handle



Convolution Layer – A Special FC Layer

32x32x3 image -> preserve spatial structure

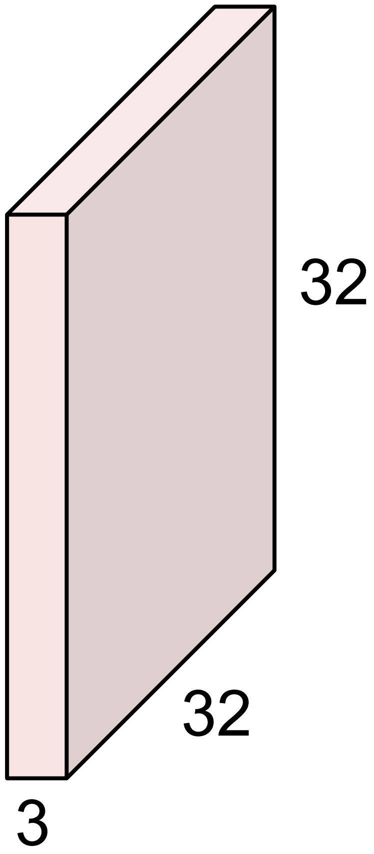


~~FC Layer: every output looks at the whole image~~

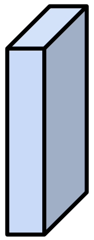
Main idea: every output only looks at small patches with small & shared number of parameters

Convolution Layer

32x32x3 image



5x5x3 filter

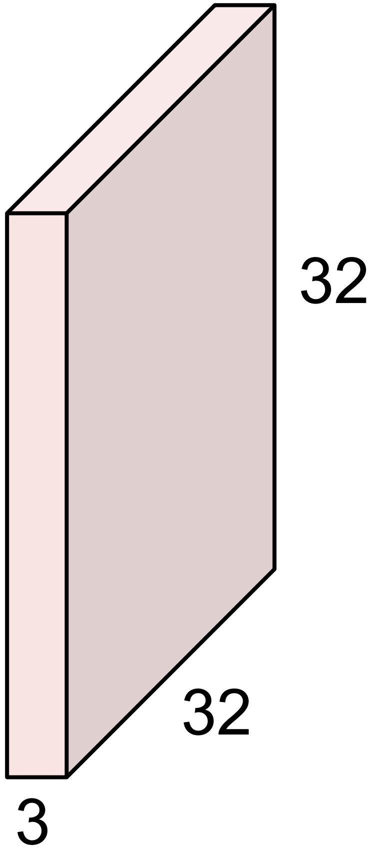


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

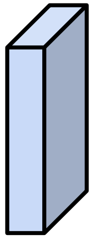
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

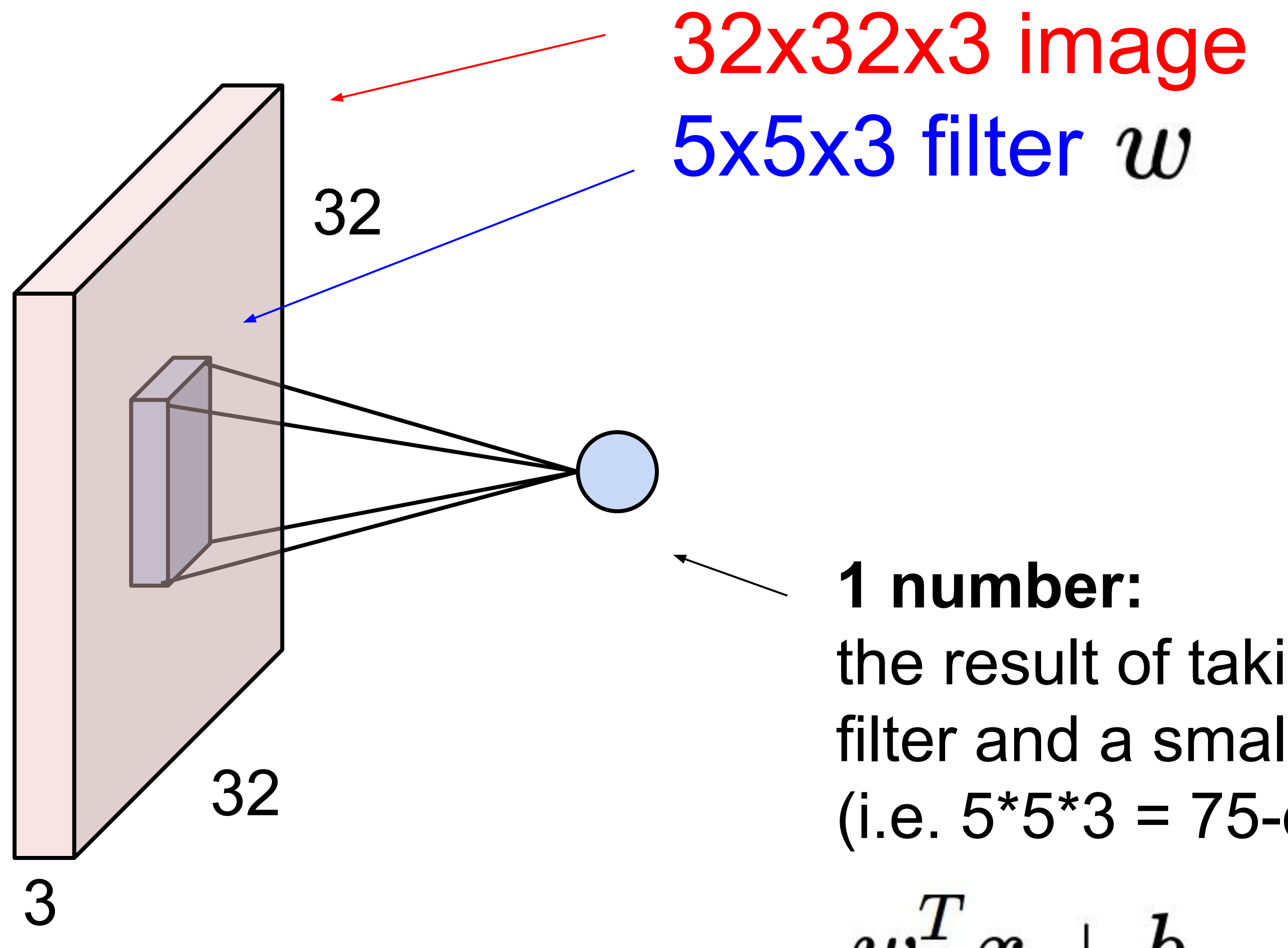


5x5x3 filter



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

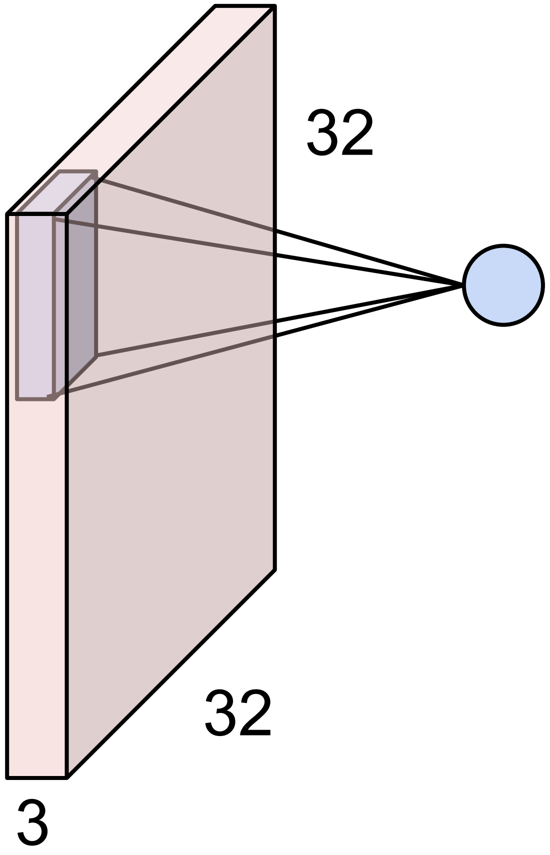
Convolution Layer



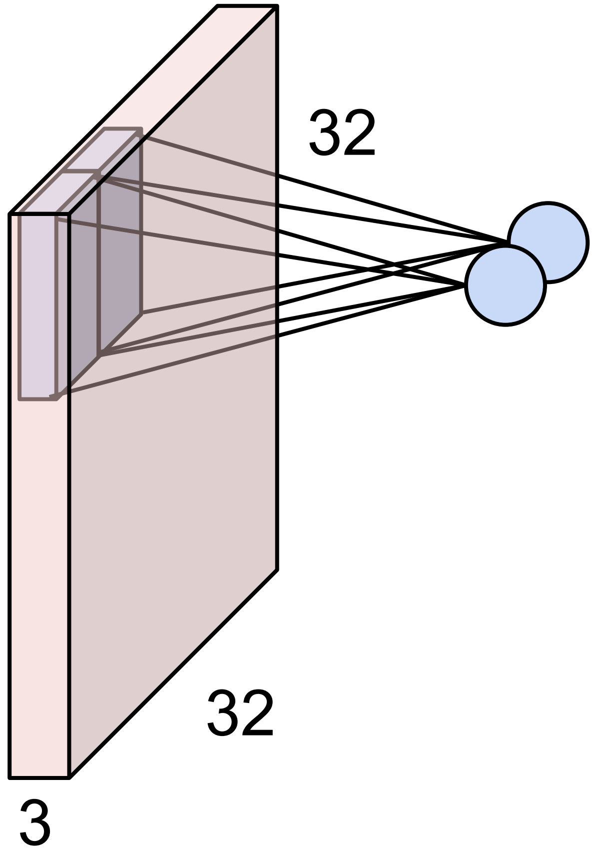
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

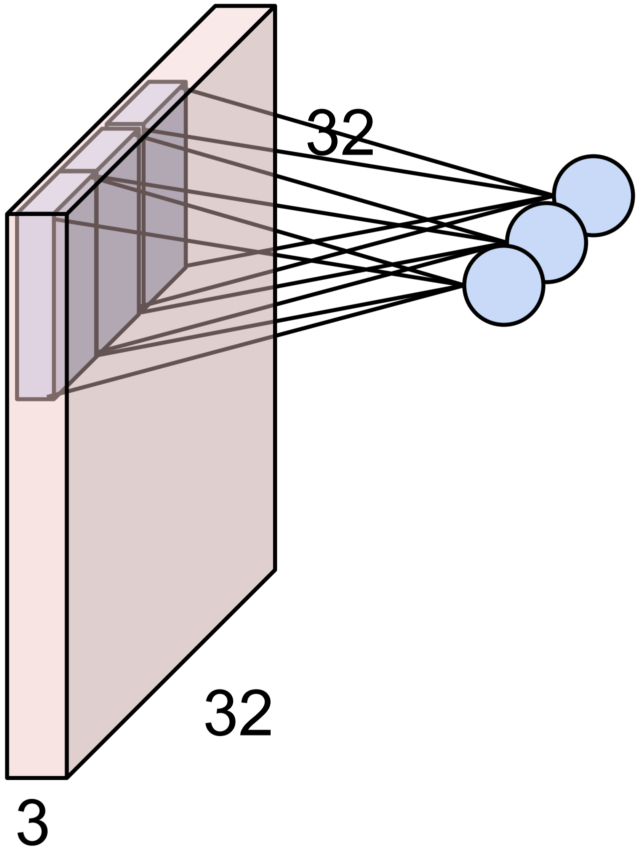
Convolution Layer



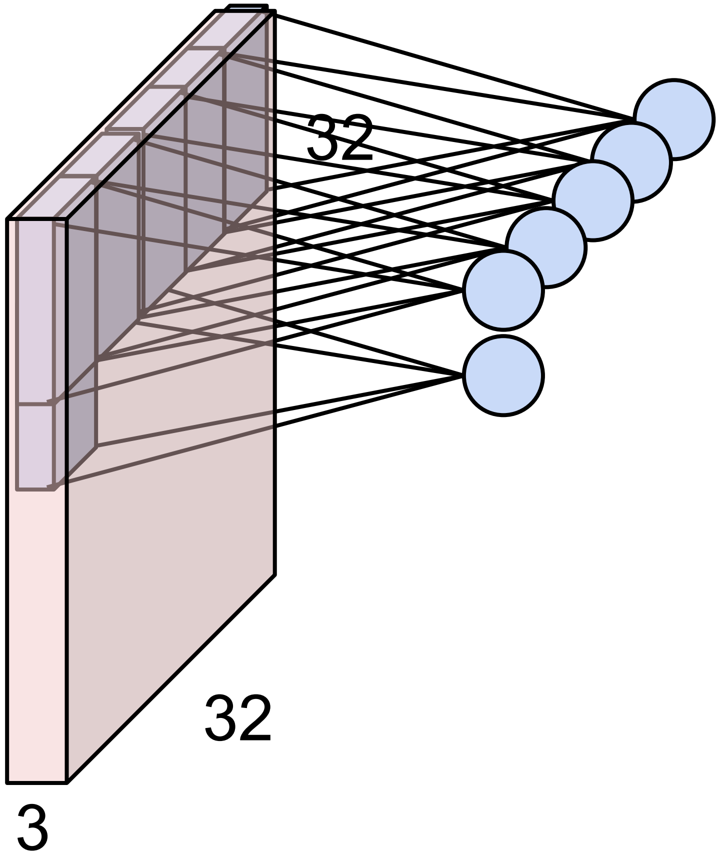
Convolution Layer



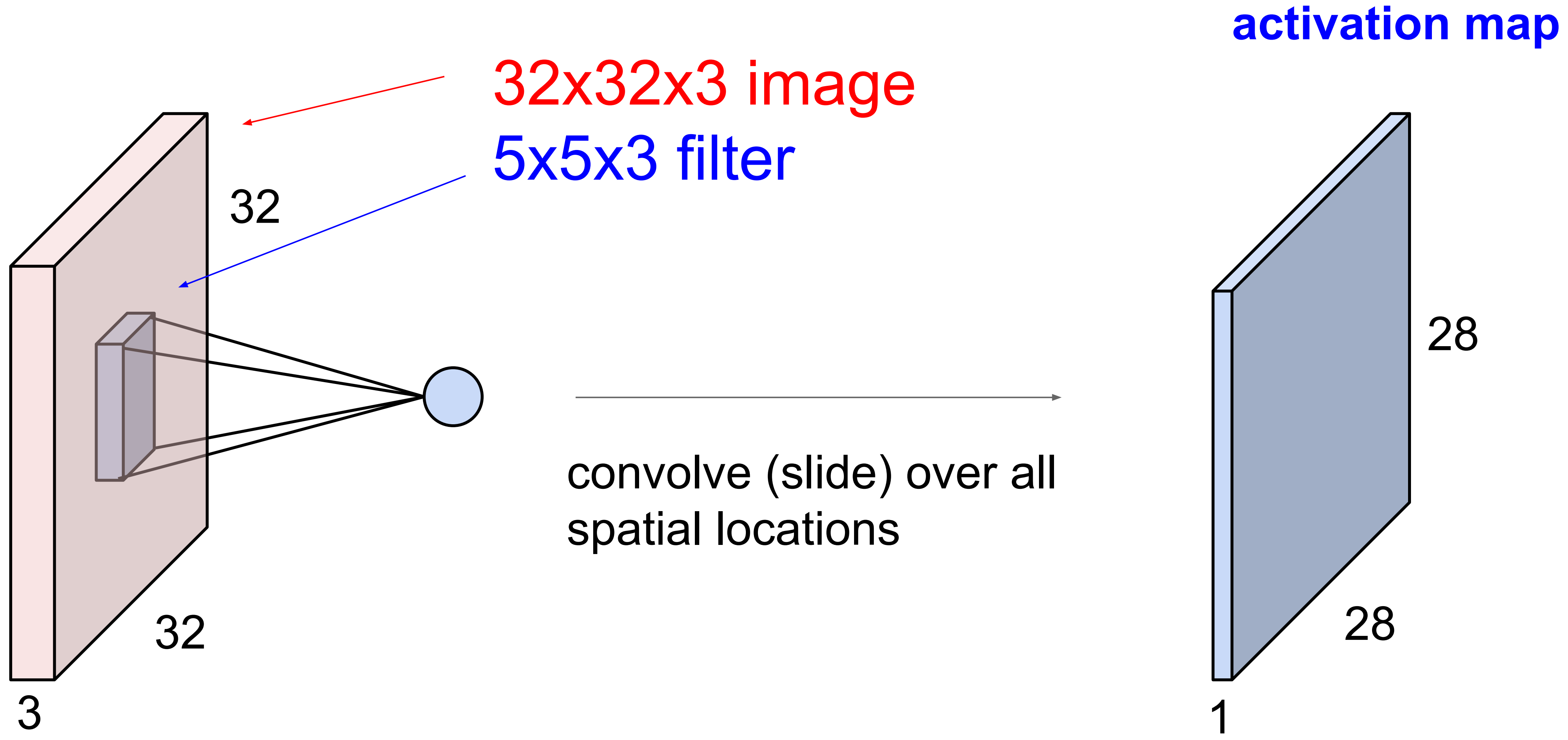
Convolution Layer



Convolution Layer



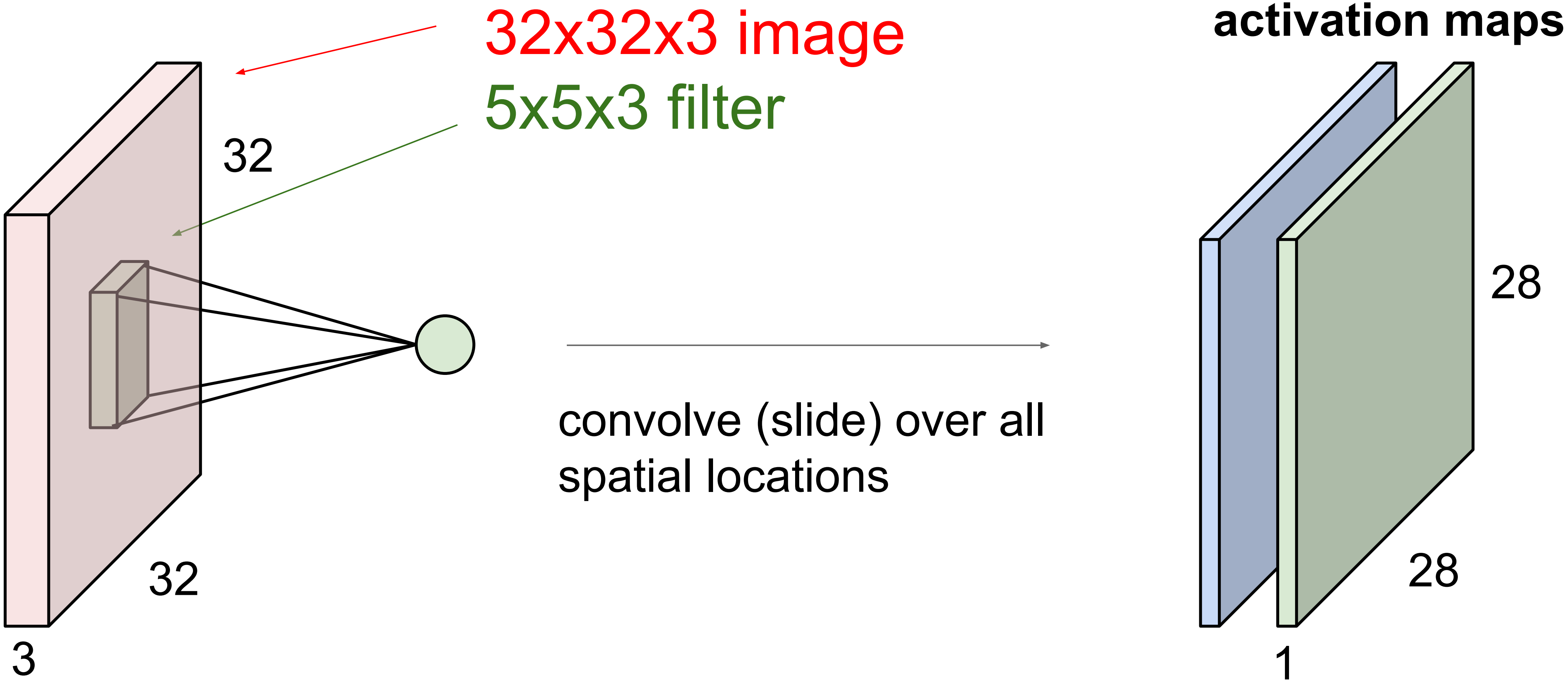
Convolution Layer



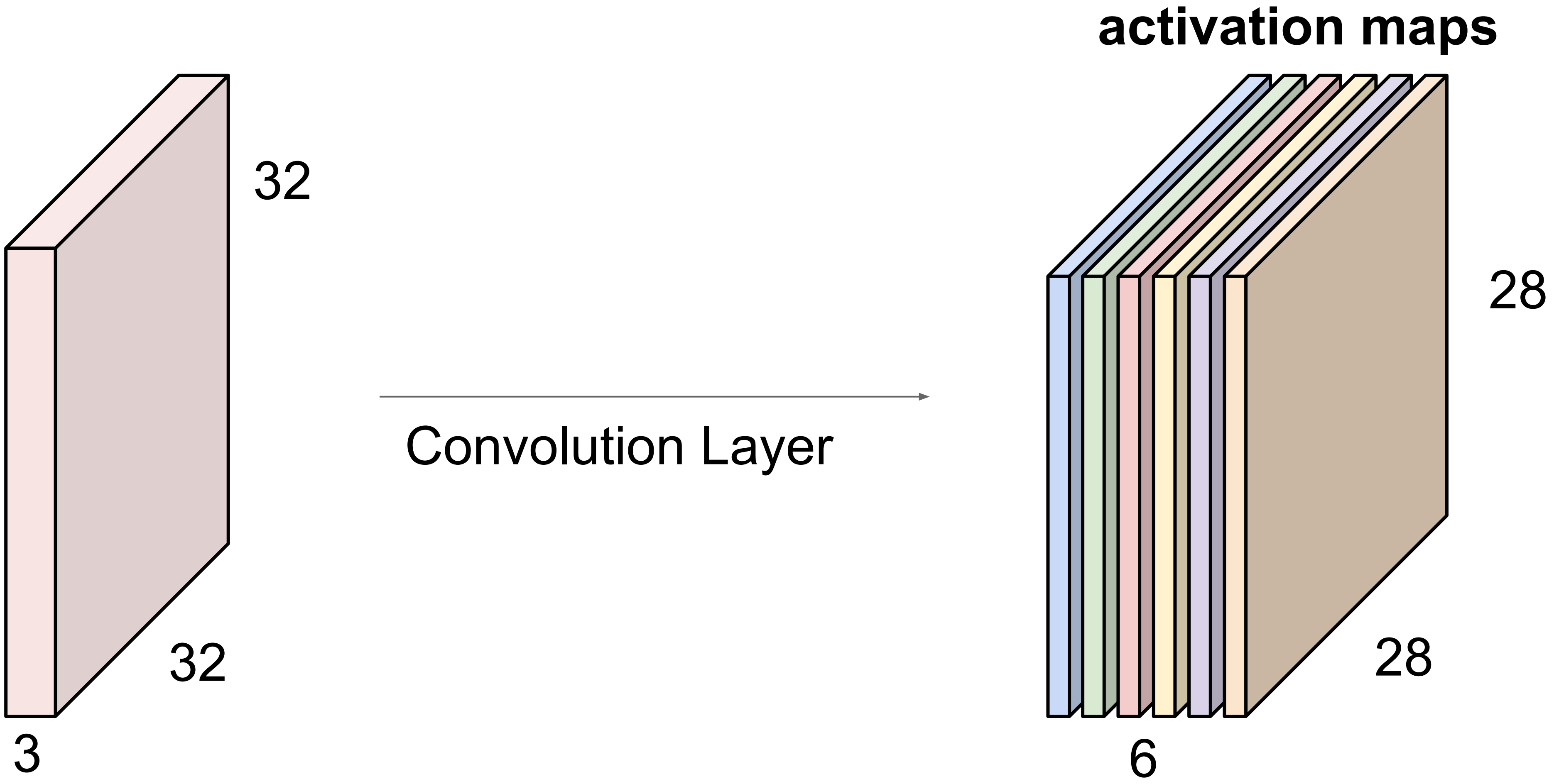
Remark: A special case of a (32x32x3) x (28x28x1) linear layer!!

Convolution Layer

consider a second, green filter

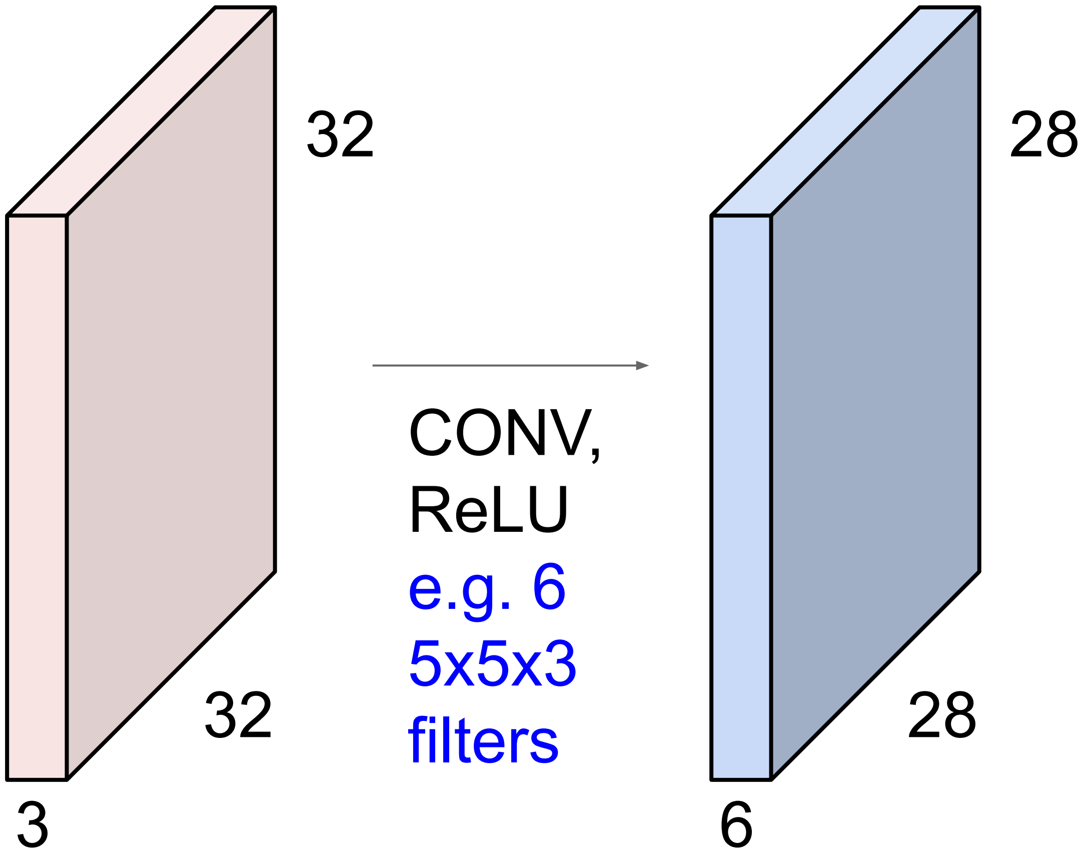


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

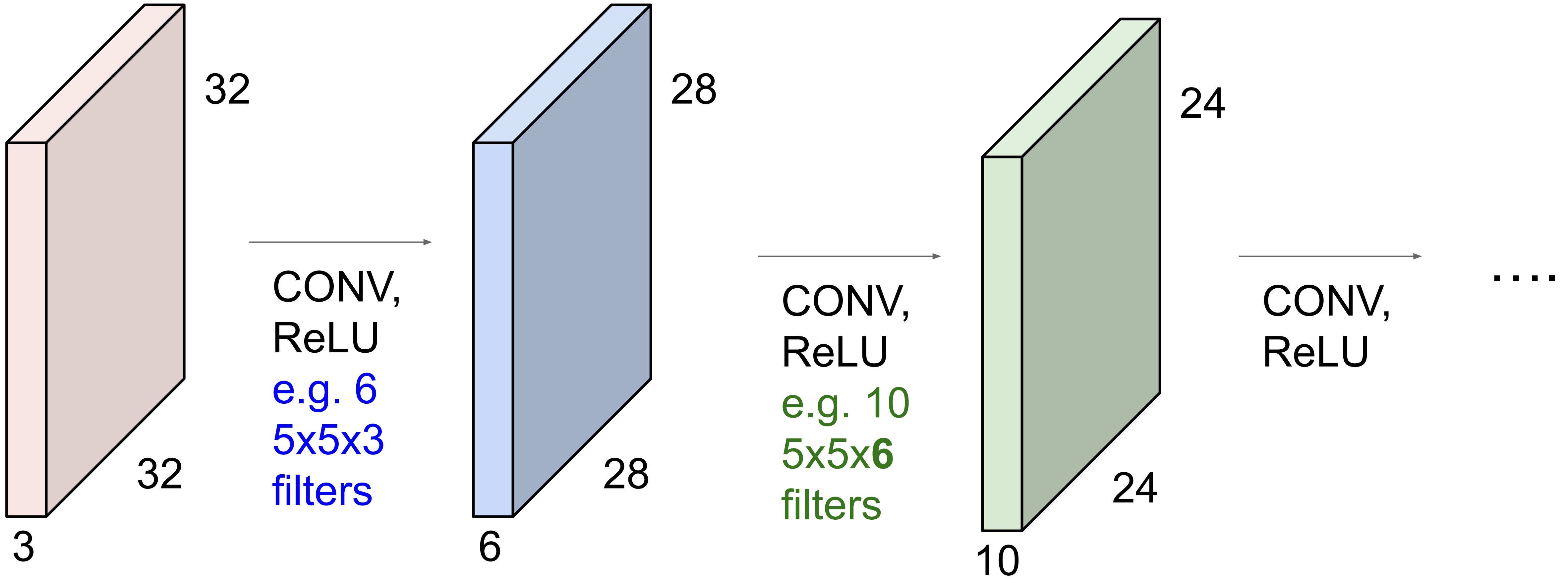


We stack these up to get a "new image" of size 28x28x6!

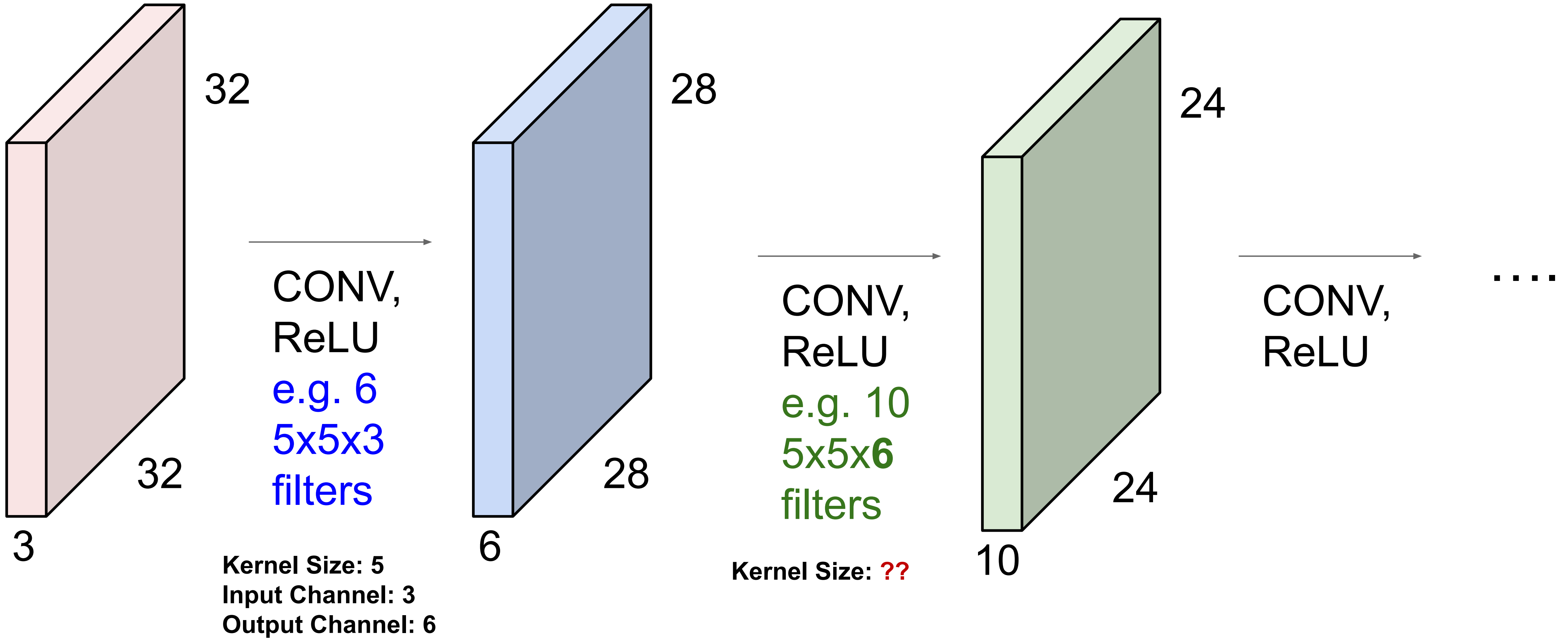
ConvNet: Sequence of Convolution Layers, interspersed with activation functions



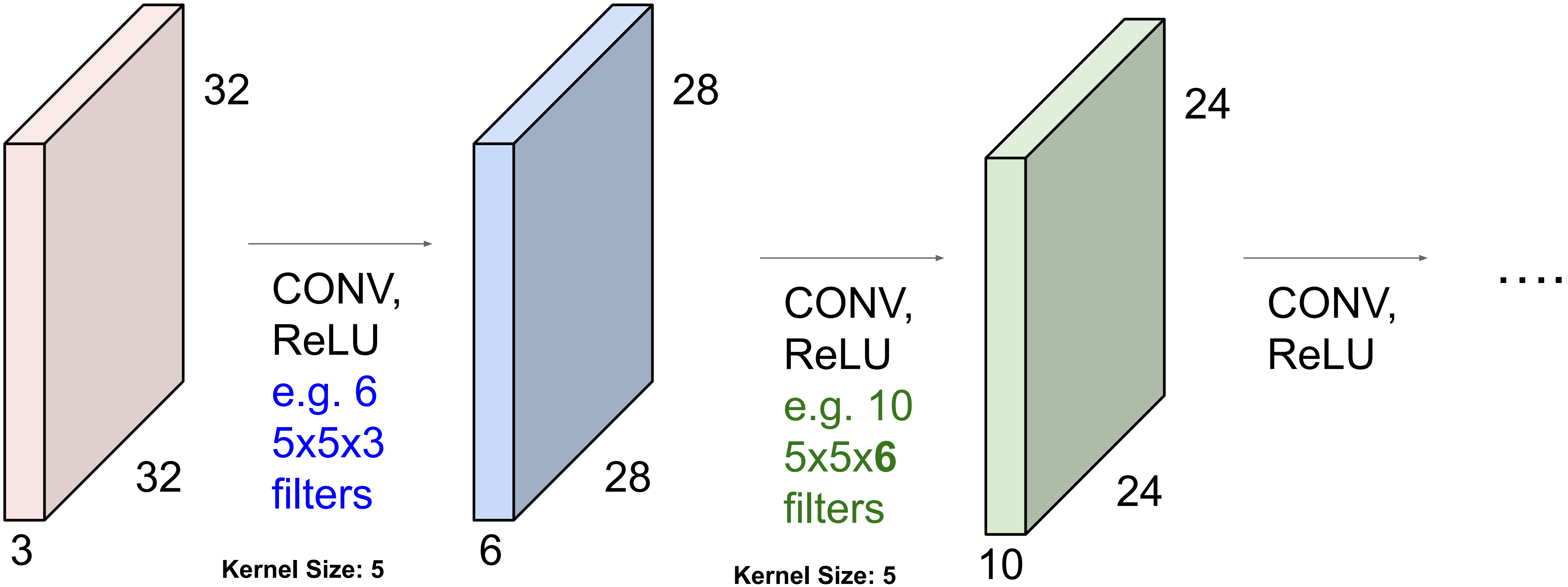
ConvNet: Sequence of Convolution Layers, interspersed with activation functions



ConvNet: Sequence of Convolution Layers, interspersed with activation functions



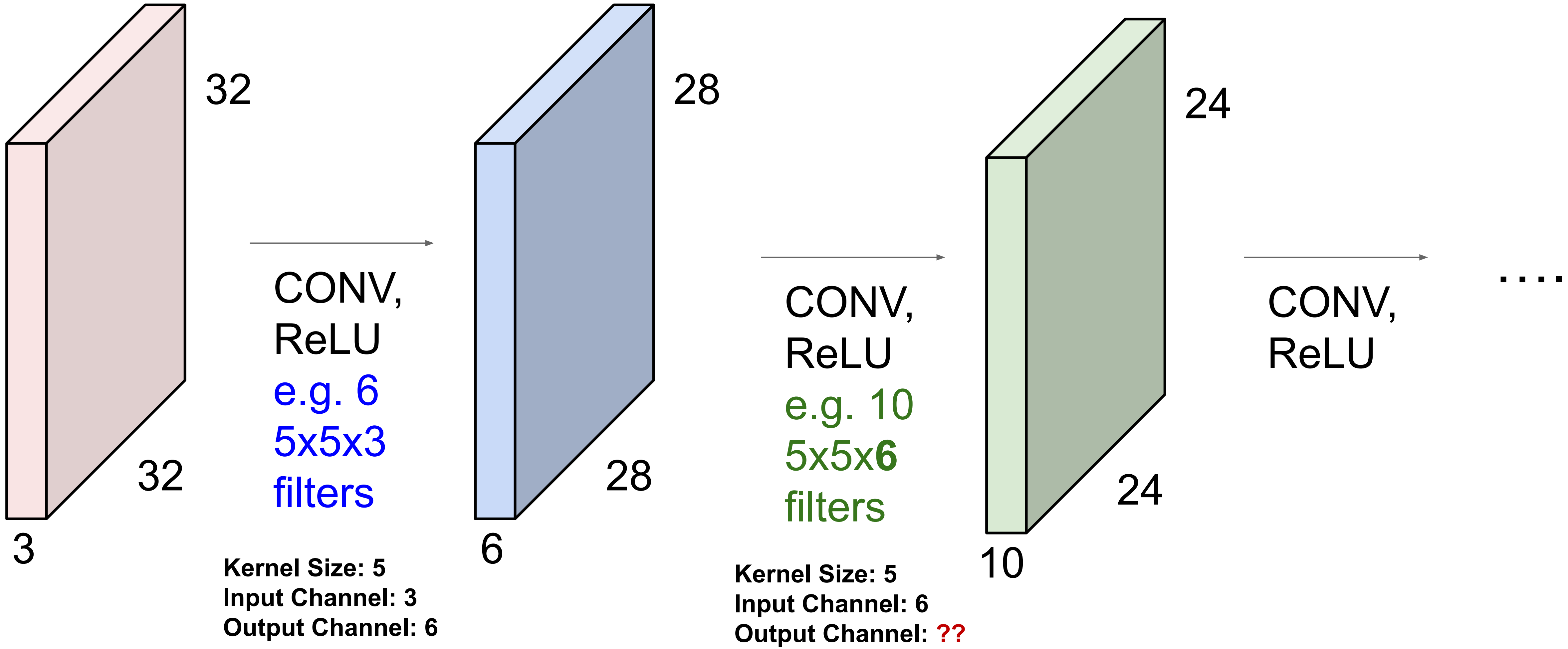
ConvNet: Sequence of Convolution Layers, interspersed with activation functions



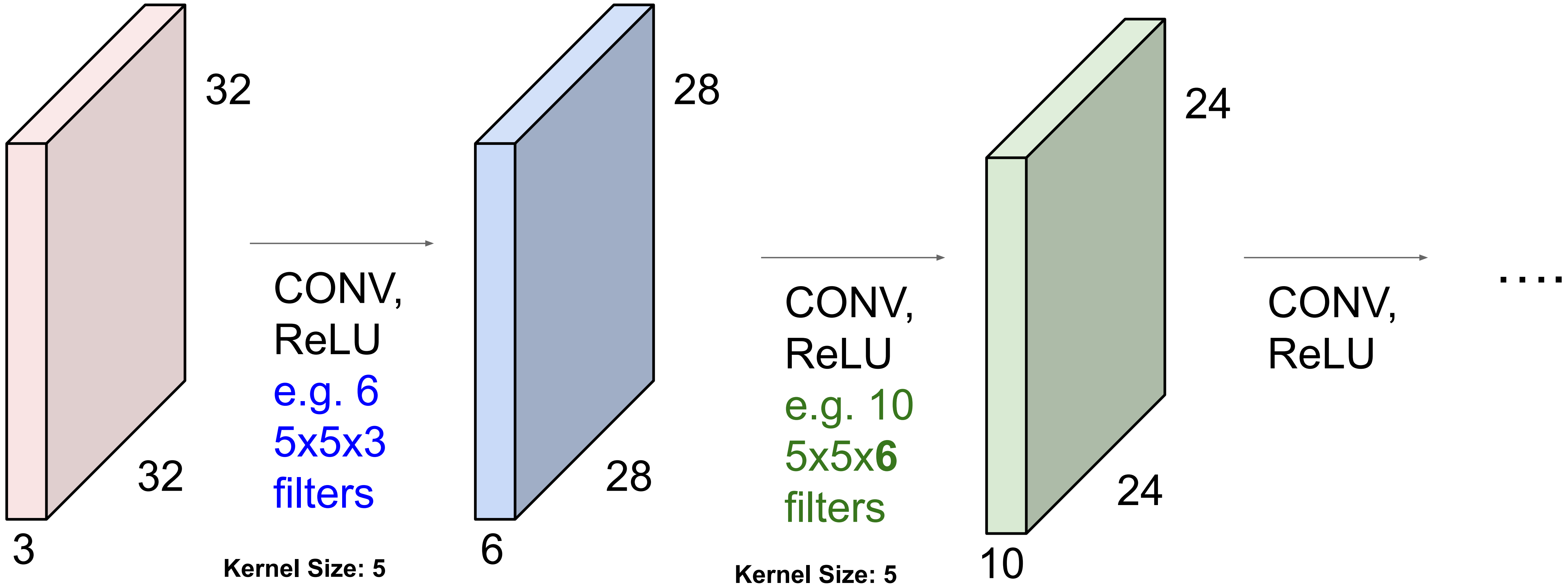
Kernel Size: 5
Input Channel: 3
Output Channel: 6

Kernel Size: 5
Input Channel: ??

ConvNet: Sequence of Convolution Layers, interspersed with activation functions



ConvNet: Sequence of Convolution Layers, interspersed with activation functions



Kernel Size: 5
Input Channel: 3
Output Channel: 6

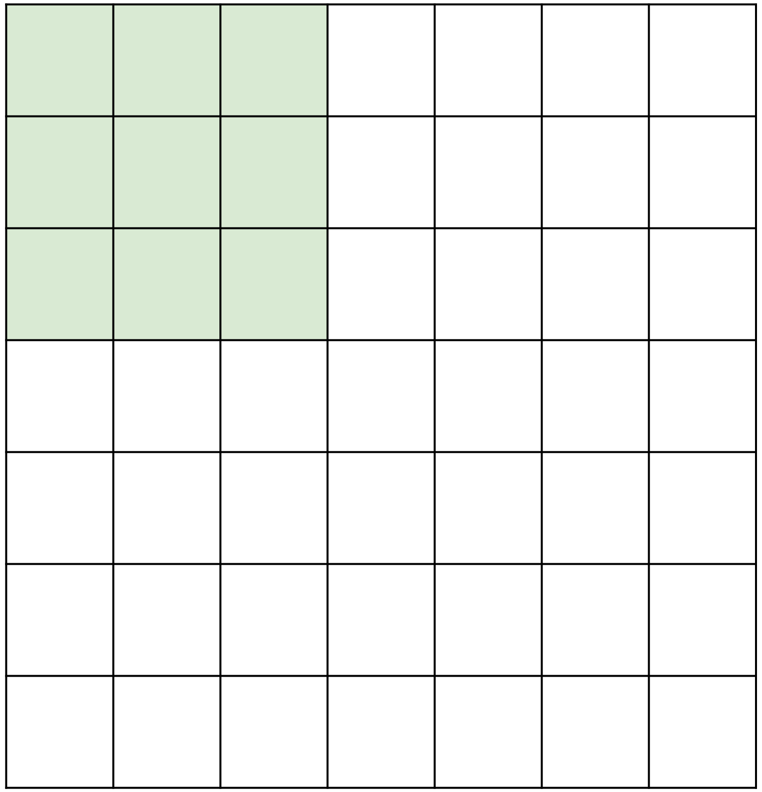
Kernel Size: 5
Input Channel: 6
Output Channel: 10

A closer look at spatial dimensions:

A closer look at spatial dimensions:

(Step 1)

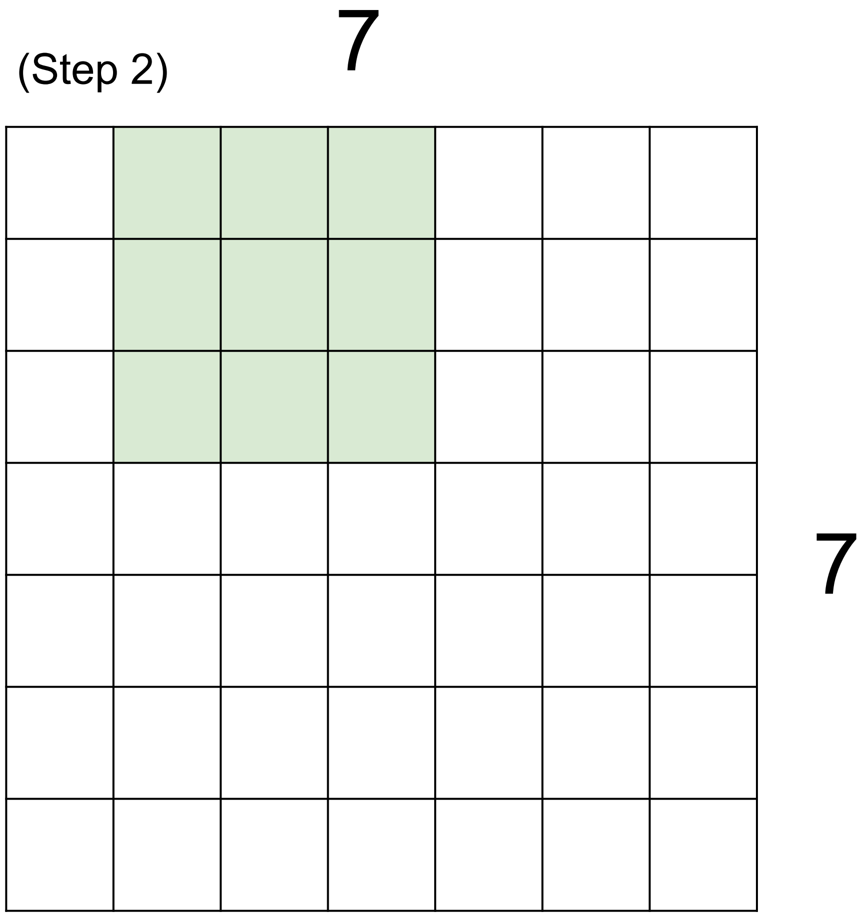
7



7

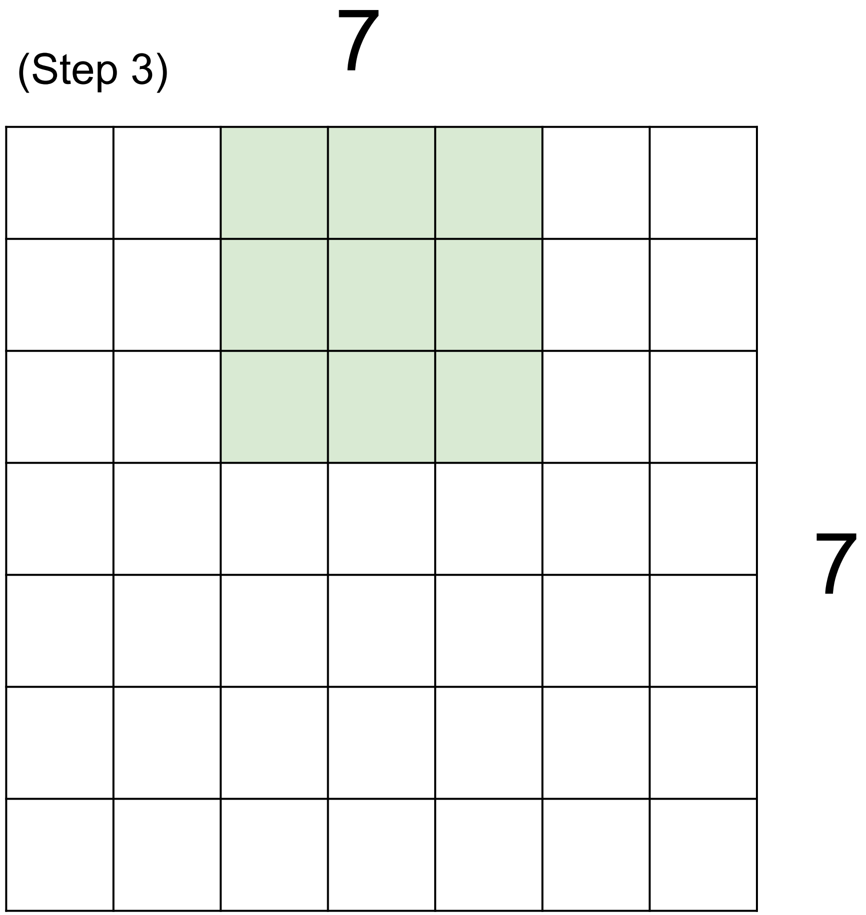
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



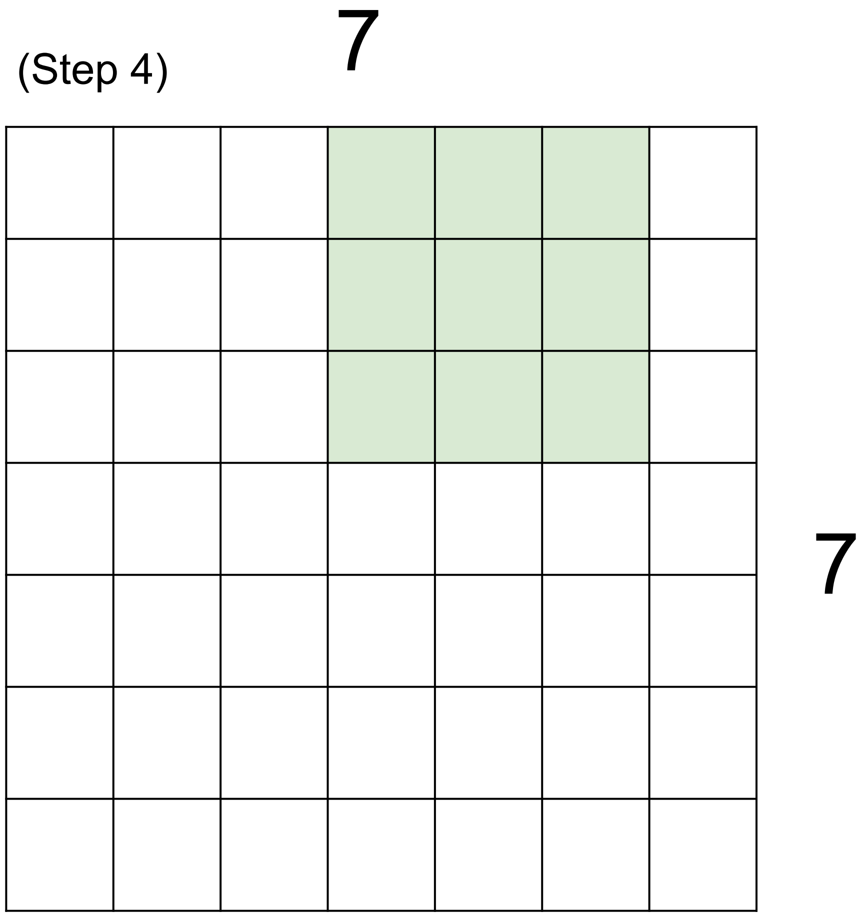
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



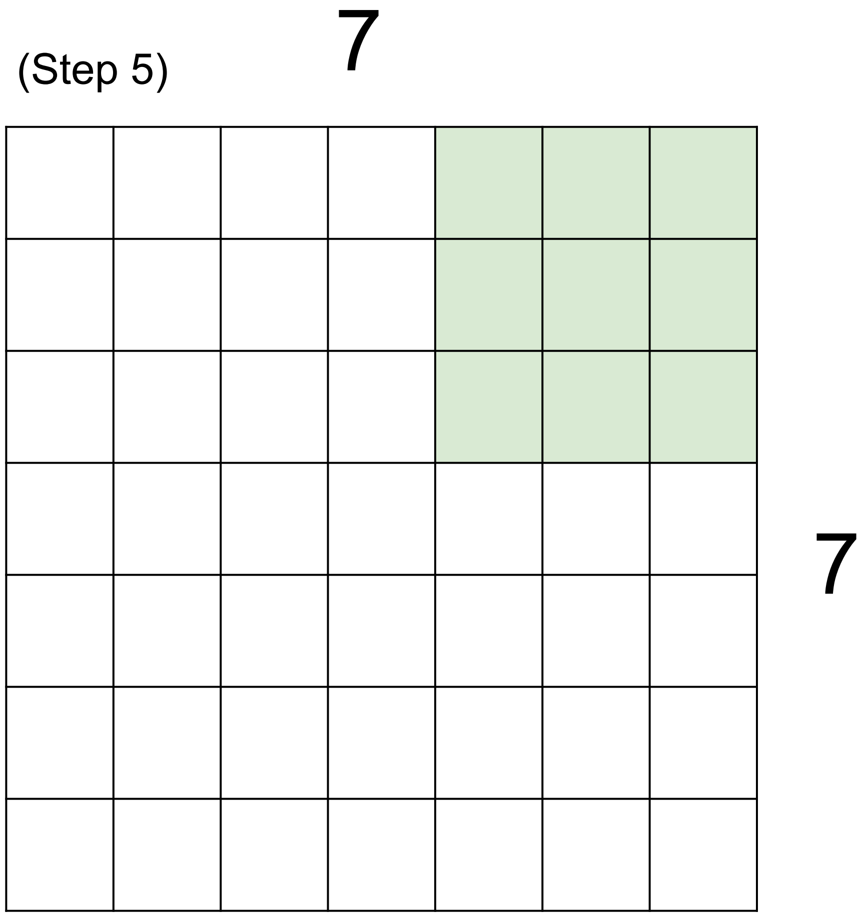
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:

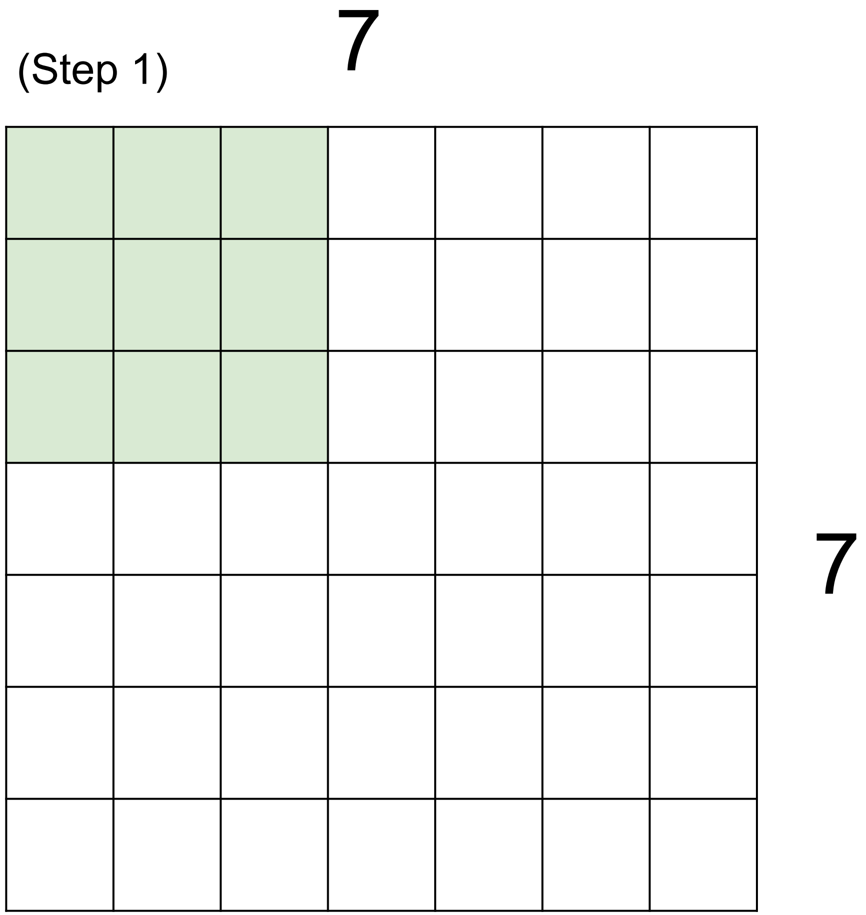


7x7 input (spatially)
assume 3x3 filter

=> 5x5 output

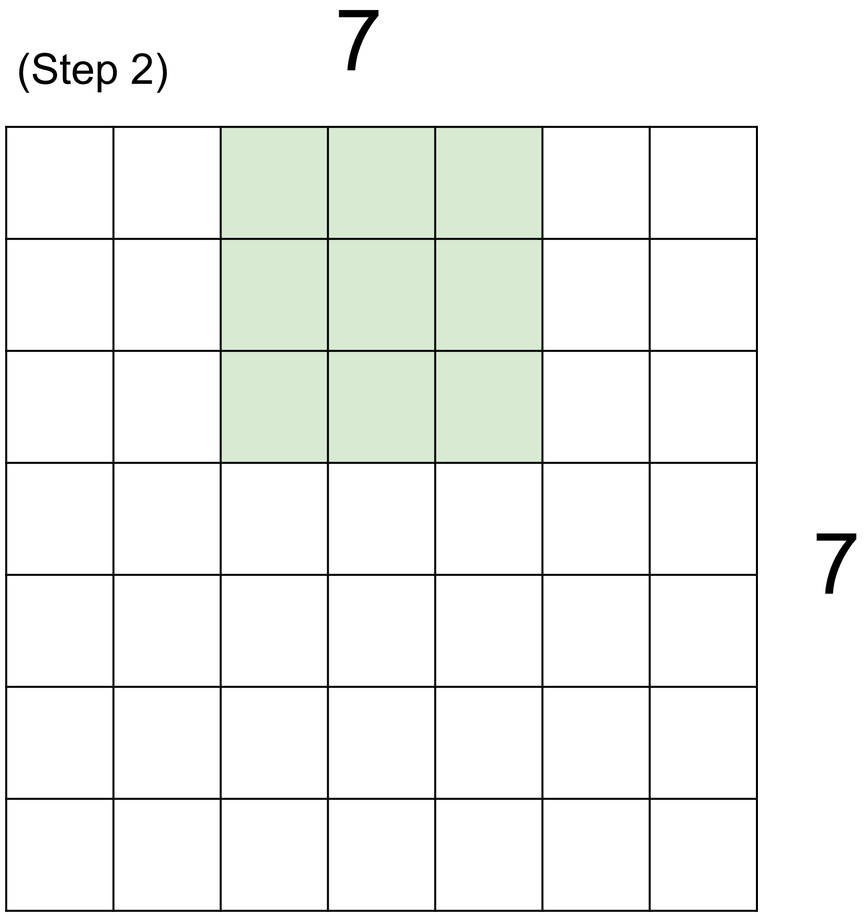
(7-3+1) x (7-3+1) = 5 x 5

A closer look at spatial dimensions:



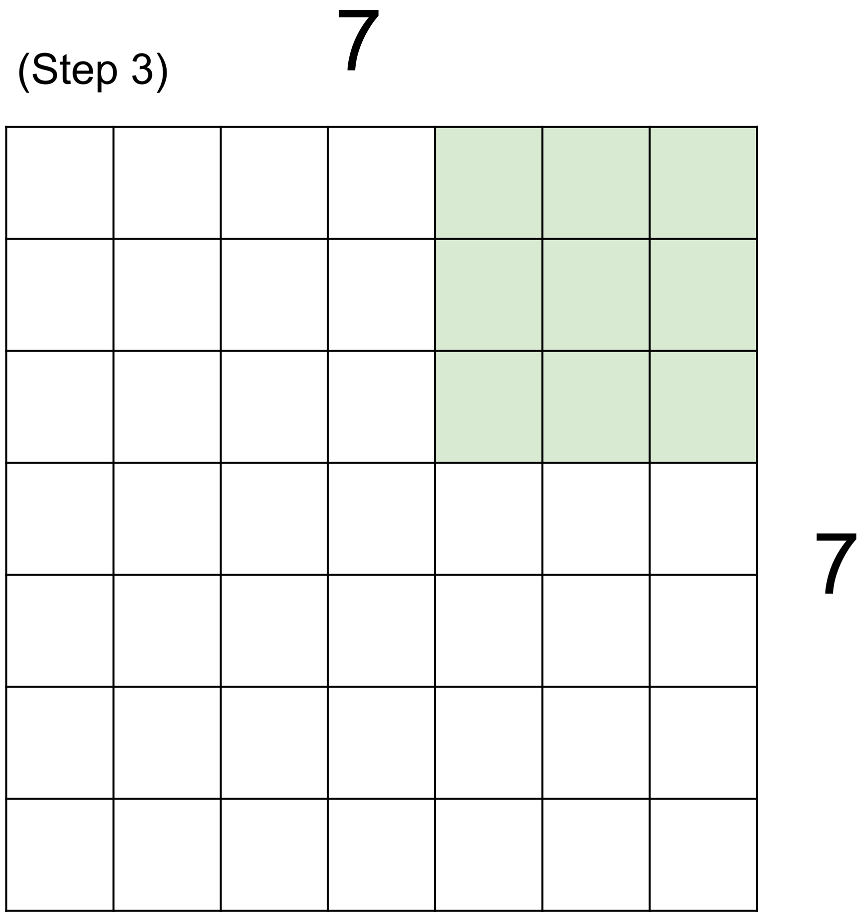
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



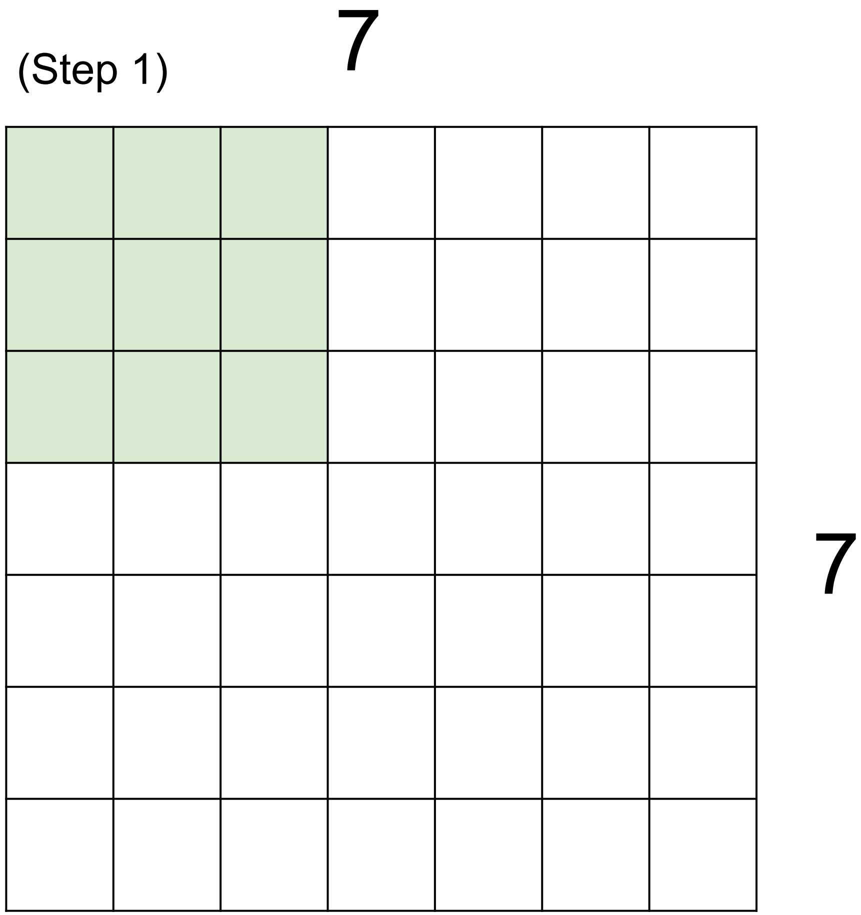
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



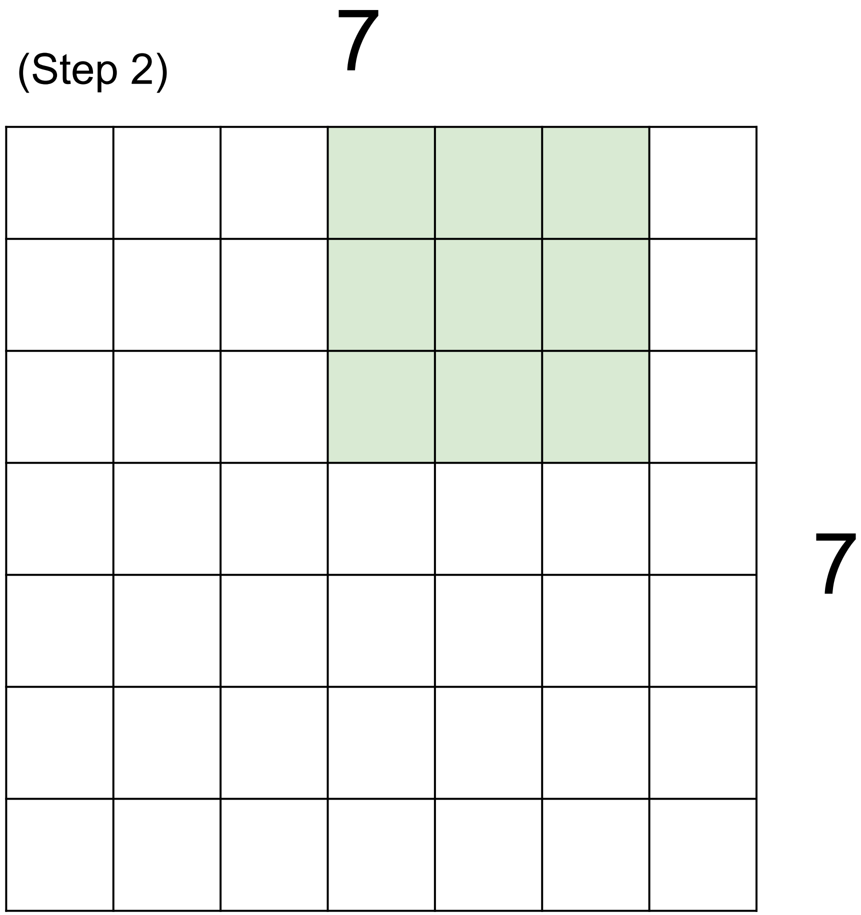
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



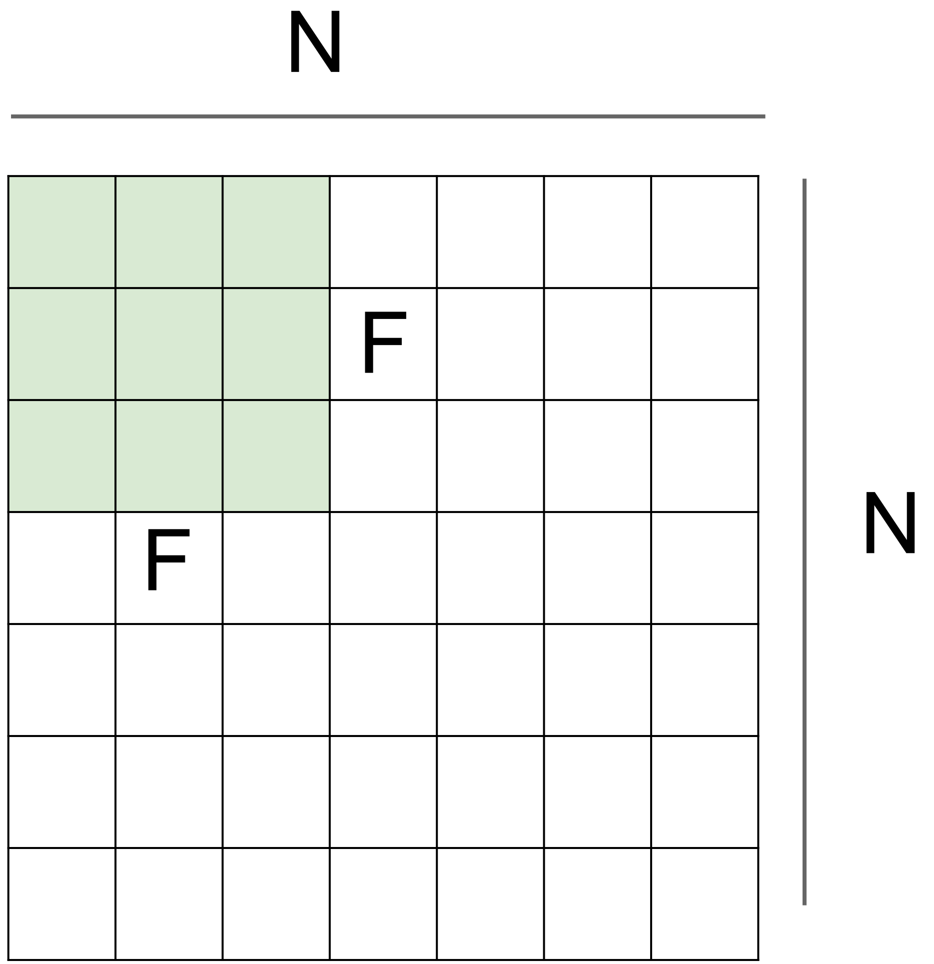
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



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

Next?



Output size:

(N - K) / stride + 1

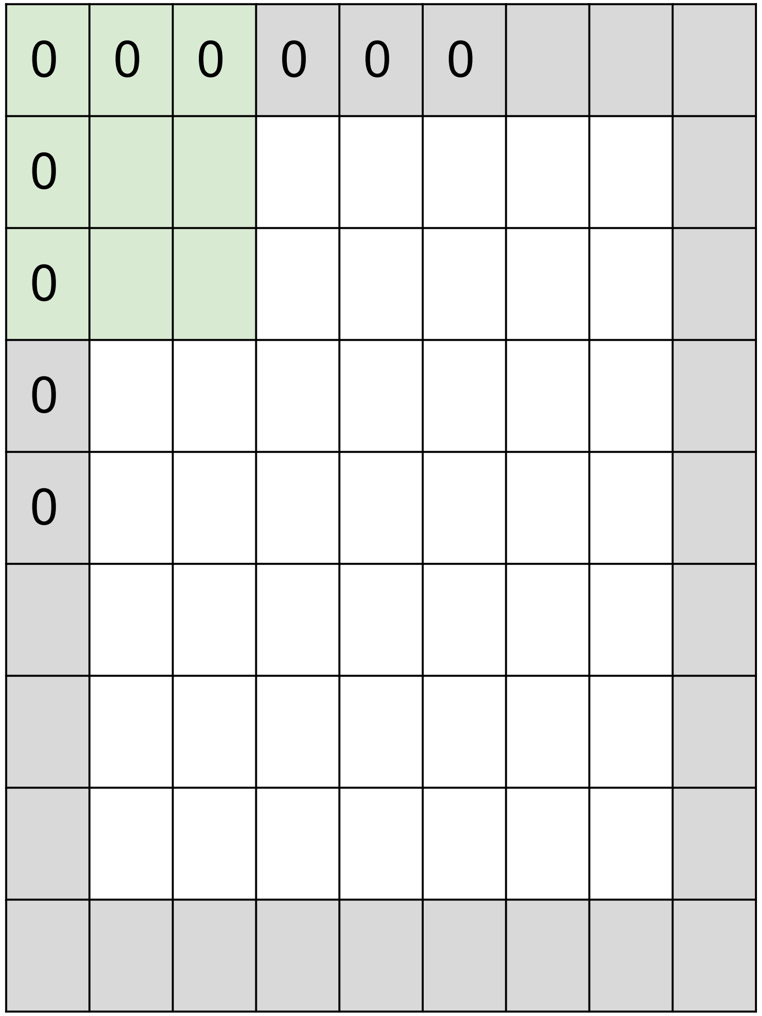
e.g. N = 7, K = 3:

stride 1 => (7 - 3)/1 + 1 = 5

stride 2 => (7 - 3)/2 + 1 = 3

stride 3 => (7 - 3)/3 + 1 = 2.33 :\

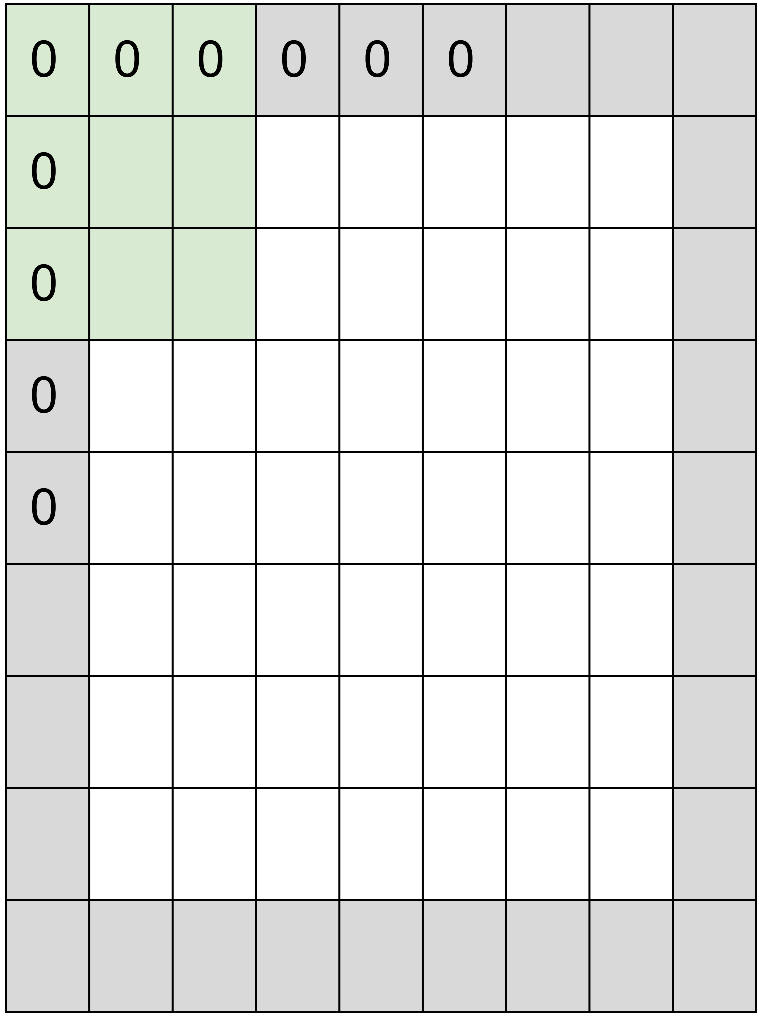
In practice: Common to zero pad the border



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

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

In practice: Common to zero pad the border

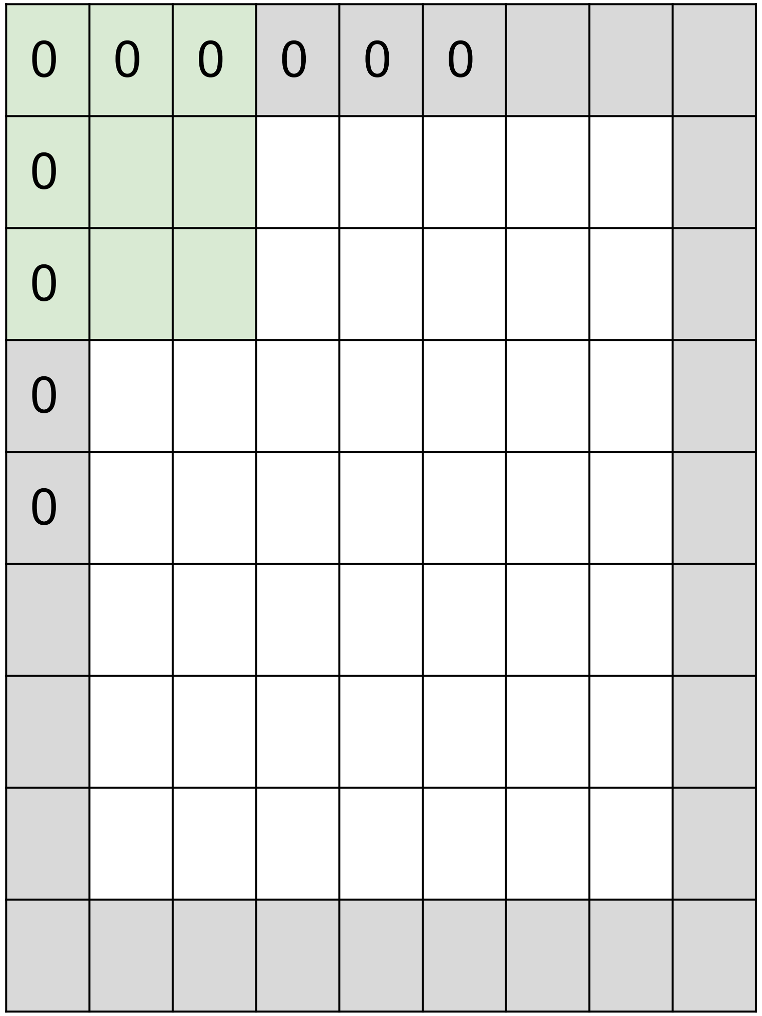


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

(recall:)
(N + 2P - K) / stride + 1

In practice: Common to zero pad the border



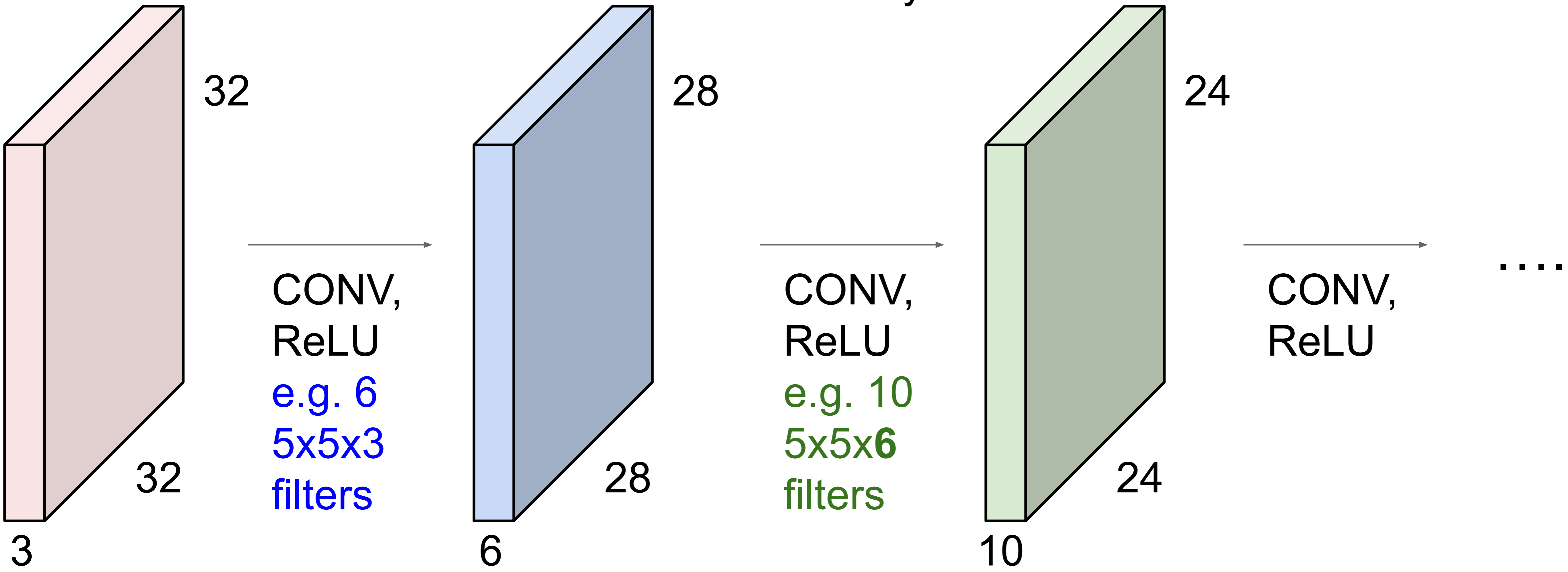
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

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

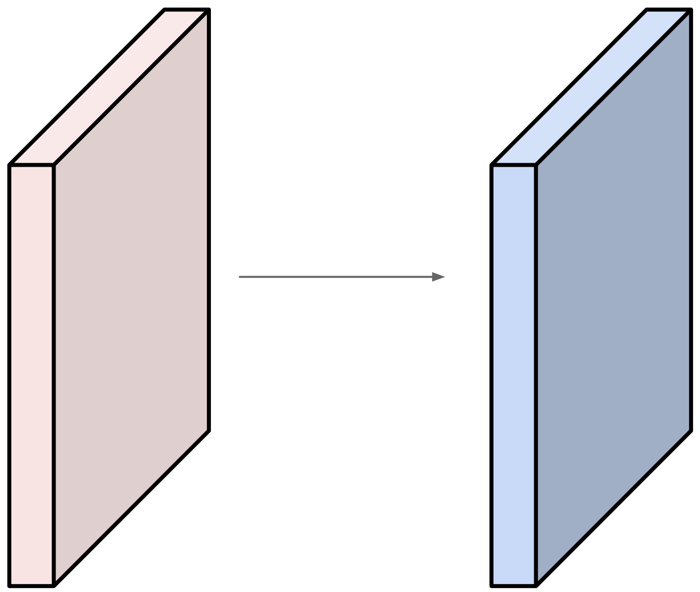
Why zero padding?

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially!
(32 -> 28 -> 24 ...)

- can't stack deeply
- use stride to reduce size whenever we really wants



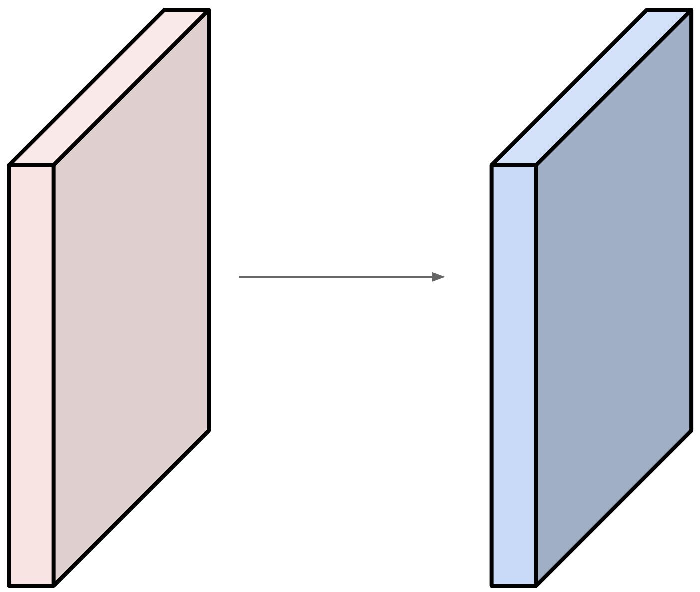
Examples time:



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

Let's assume output size is HxWxD.
What is D?

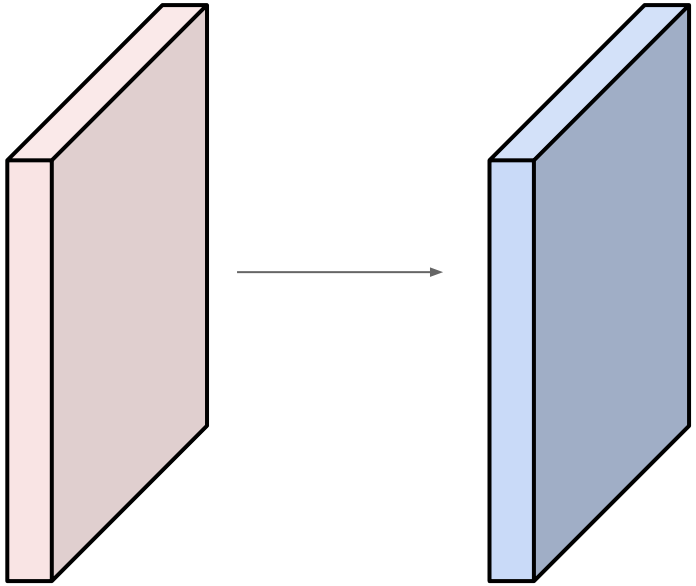
Examples time:



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

Let’s assume output size is HxWxD.
What is D? **10**

Examples time:



Input volume: **32x32x3**

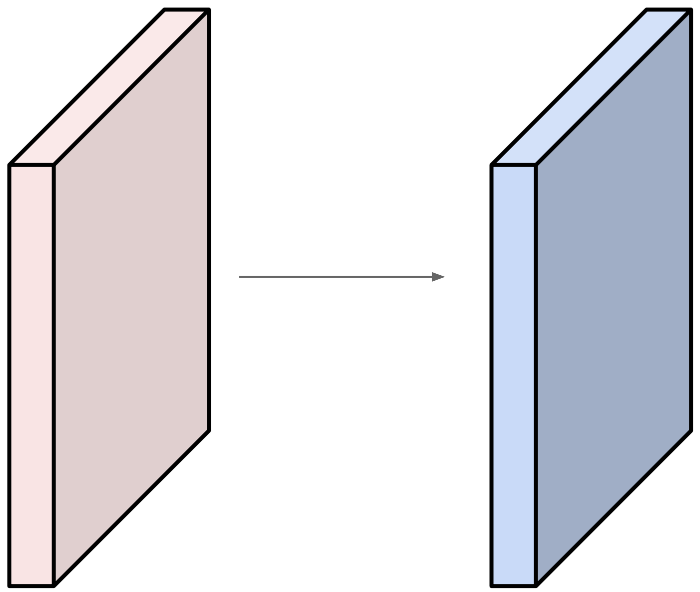
10 5x5 filters with stride 1, pad 2

Let's assume output size is HxWxC.

What is C? **10**

What is H or W?

Examples time:



Input volume: **32x32x3**

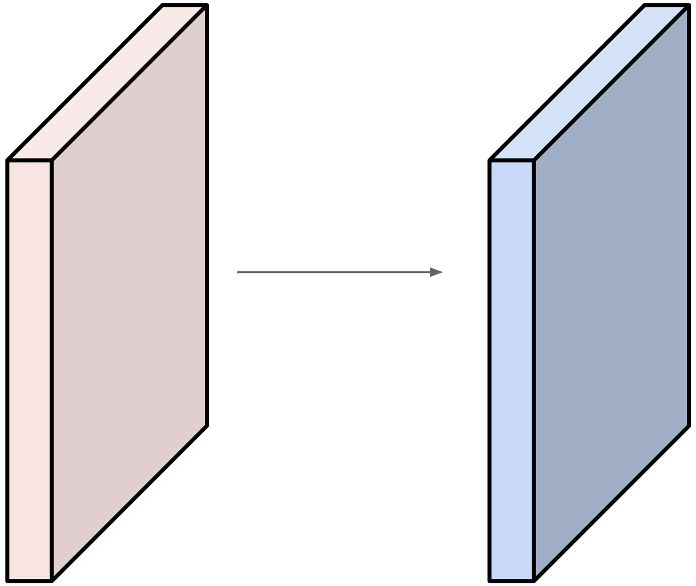
10 5x5 filters with stride 1, pad 2

Let's assume output size is HxWxC.

What is C? 10

What is H or W? (32+2*2-5)/1+1 = 32

Examples time:



Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

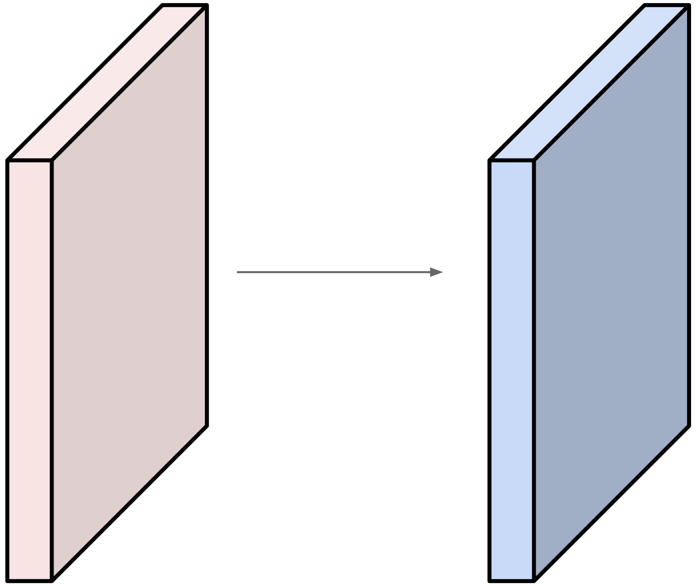
Let's assume output size is HxWxD.

What is D? 10

What is H or W? (32+2*2-5)/1+1 = 32

So the total output size is: **32x32x10**

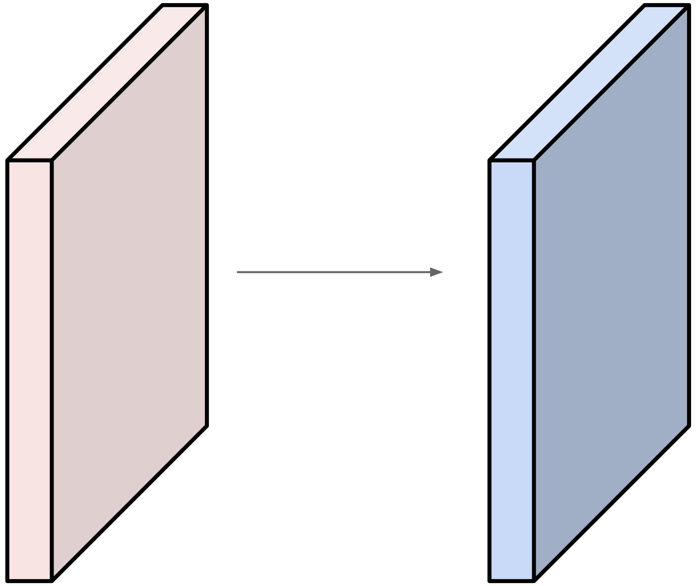
Examples time:



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

Number of parameters in this layer?

Exercise:



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**

Convolution layer: summary

Let's assume input is $W_1 \times H_1 \times C_1$

Conv layer needs 4 hyperparameters:

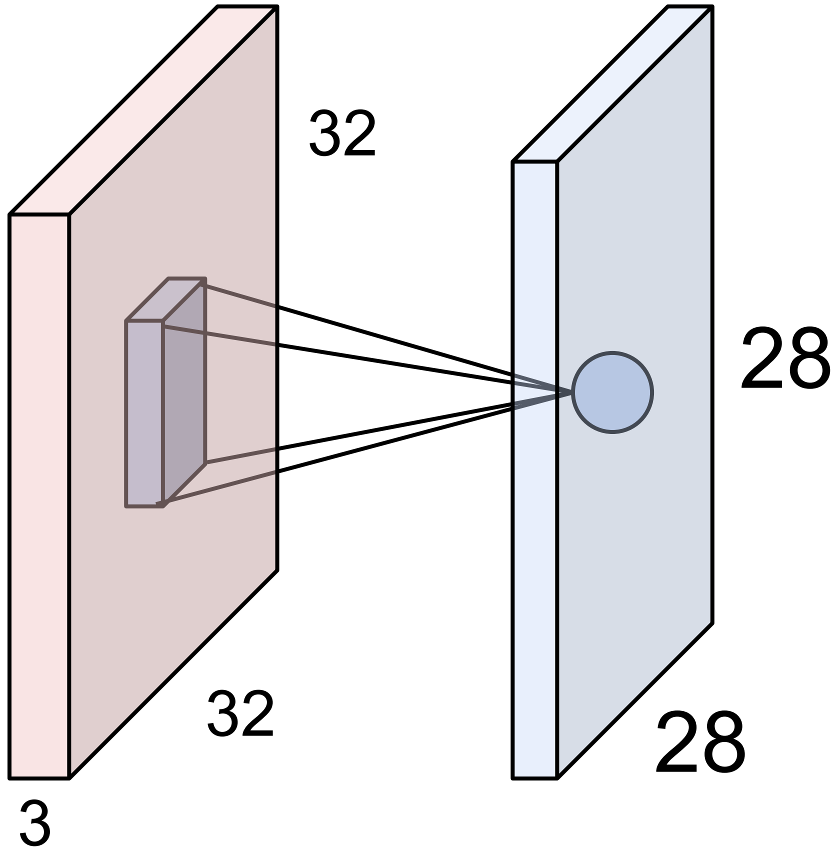
- Number of filters C_2 (output channels)
- The filter size **K**
- The stride **S**
- The zero padding **P**

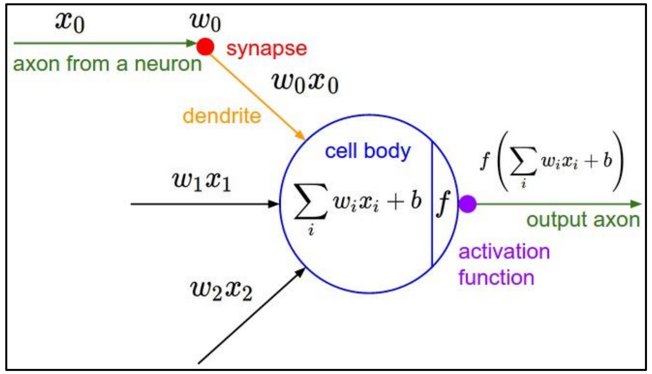
This will produce an output of $W_2 \times H_2 \times C_2$ where:

- $W_2 = (W_1 - K + 2P)/S + 1$
- $H_2 = (H_1 - K + 2P)/S + 1$

Number of parameters: $K^2C_1C_2$ and C_2 biases

Receptive field



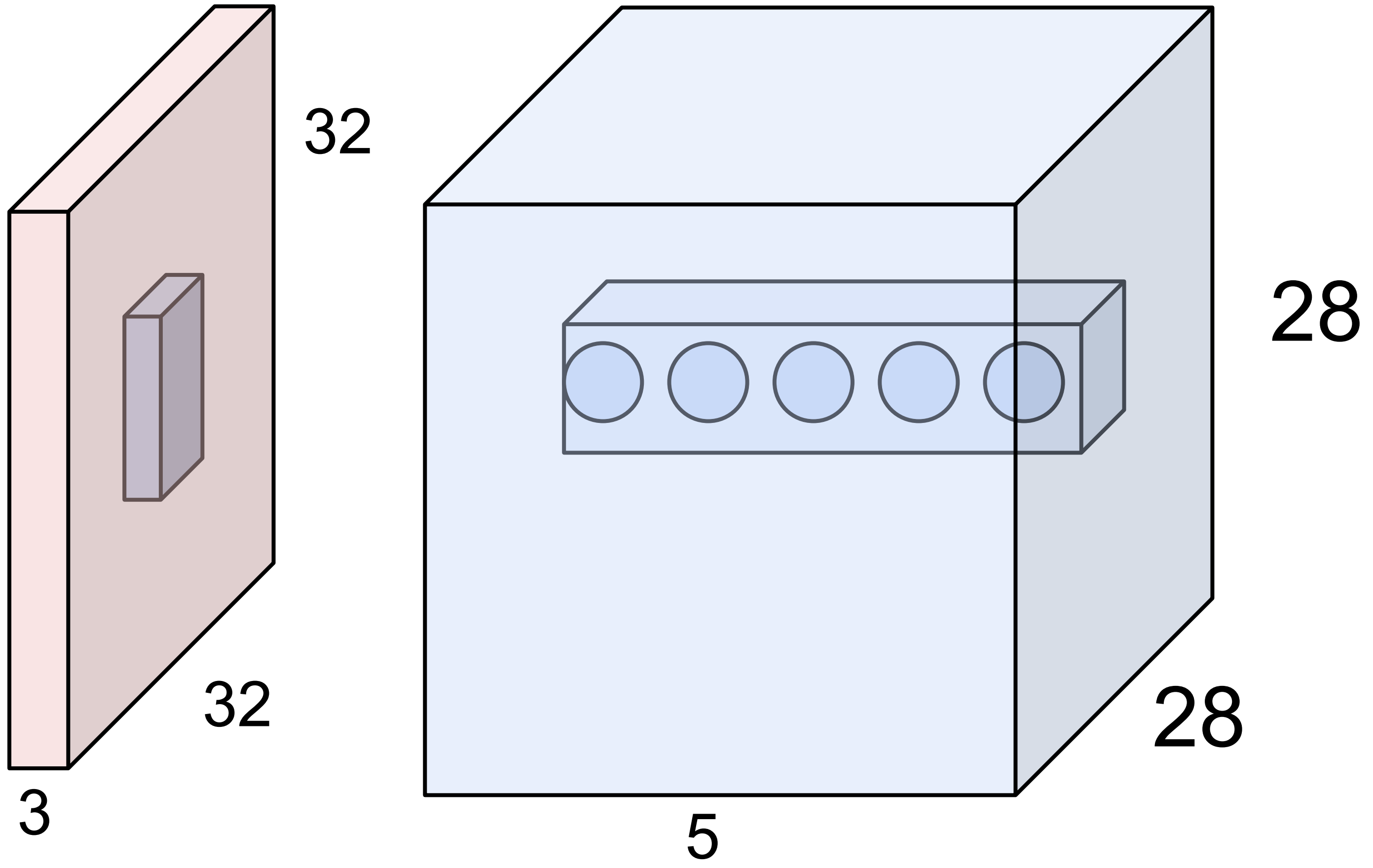


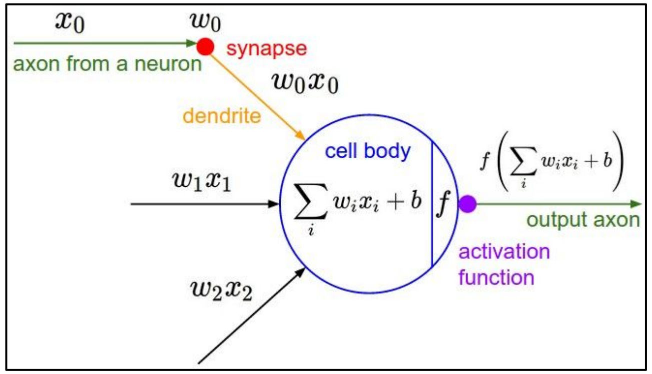
An activation map is a 28x28 sheet of neuron outputs:

1. Each is connected to a small region in the input
2. All of them share parameters

"5x5 filter" -> "5x5 receptive field for each neuron"

The brain/neuron view of CONV Layer



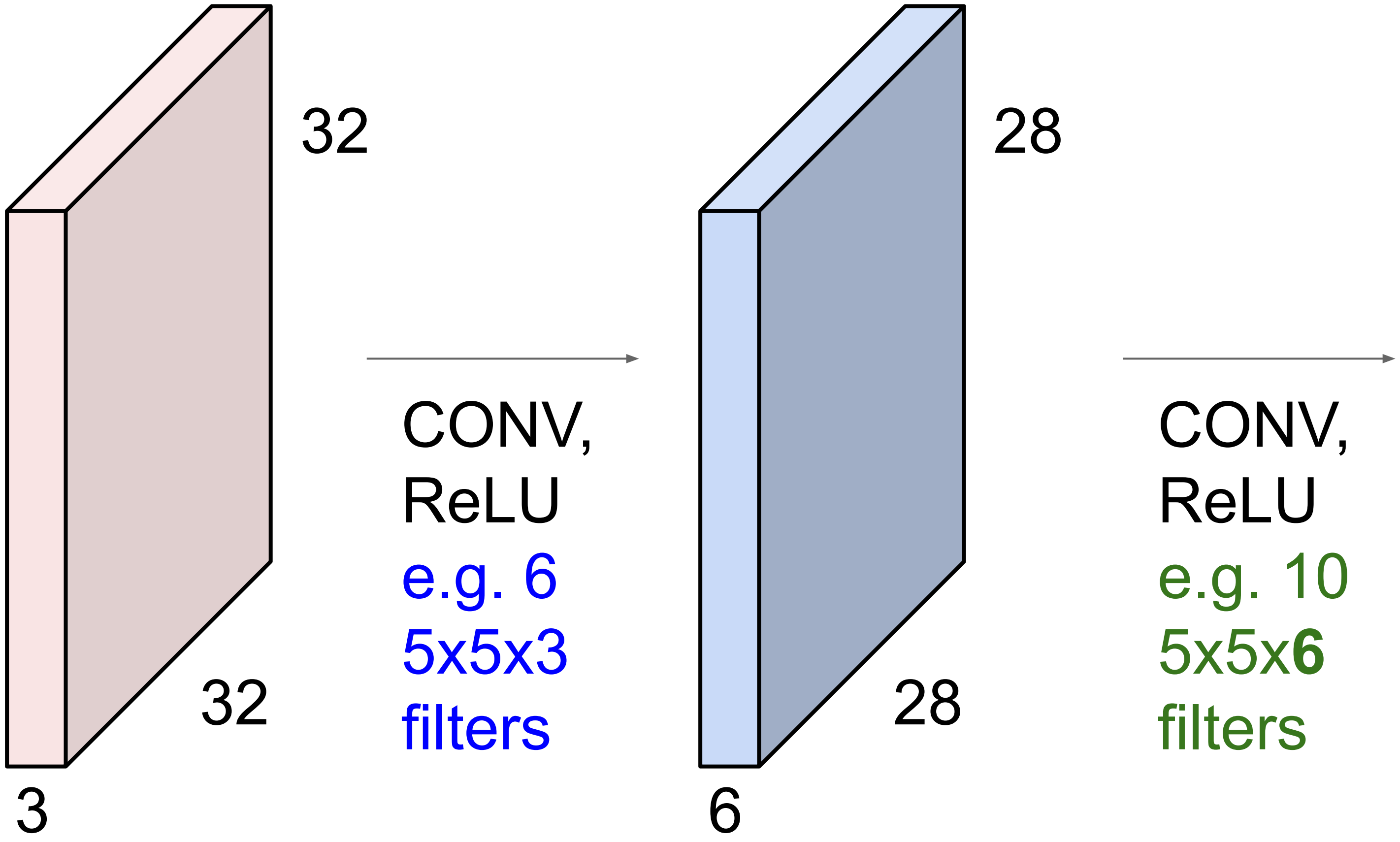


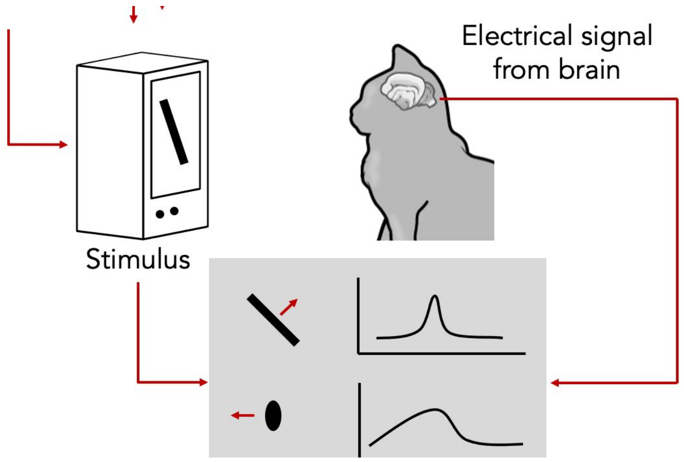
E.g. with 5 filters,
CONV layer consists of neurons arranged in a 3D grid (28x28x5)

There will be 5 different neurons all looking at the same region in the input volume

The brain/neuron view of CONV Layer

Stack of Conv layers v.s. hierarchical organization in the visual processing system.



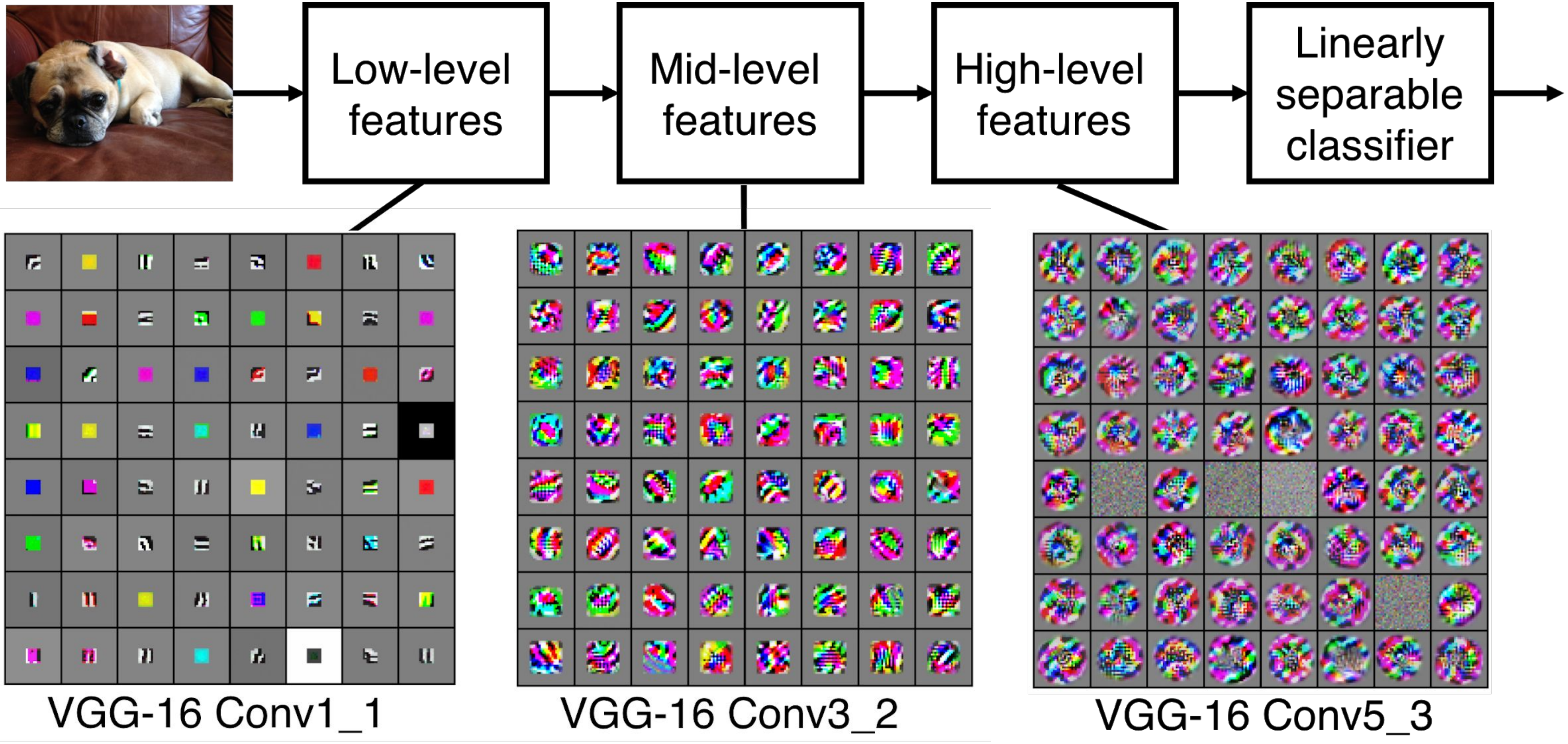


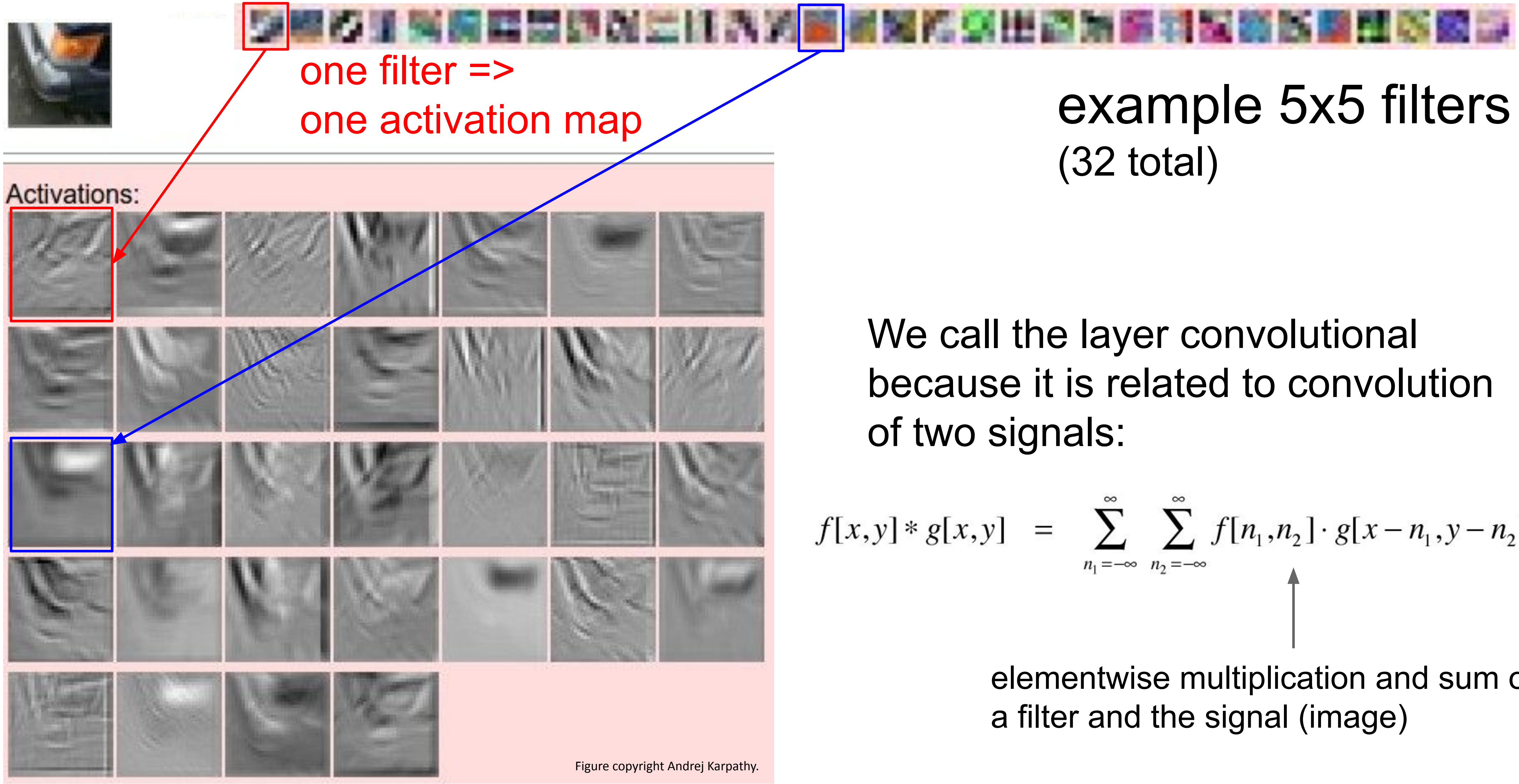
What CNN learns?

What CNN learns?

[Zeiler and Fergus 2013]

Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].





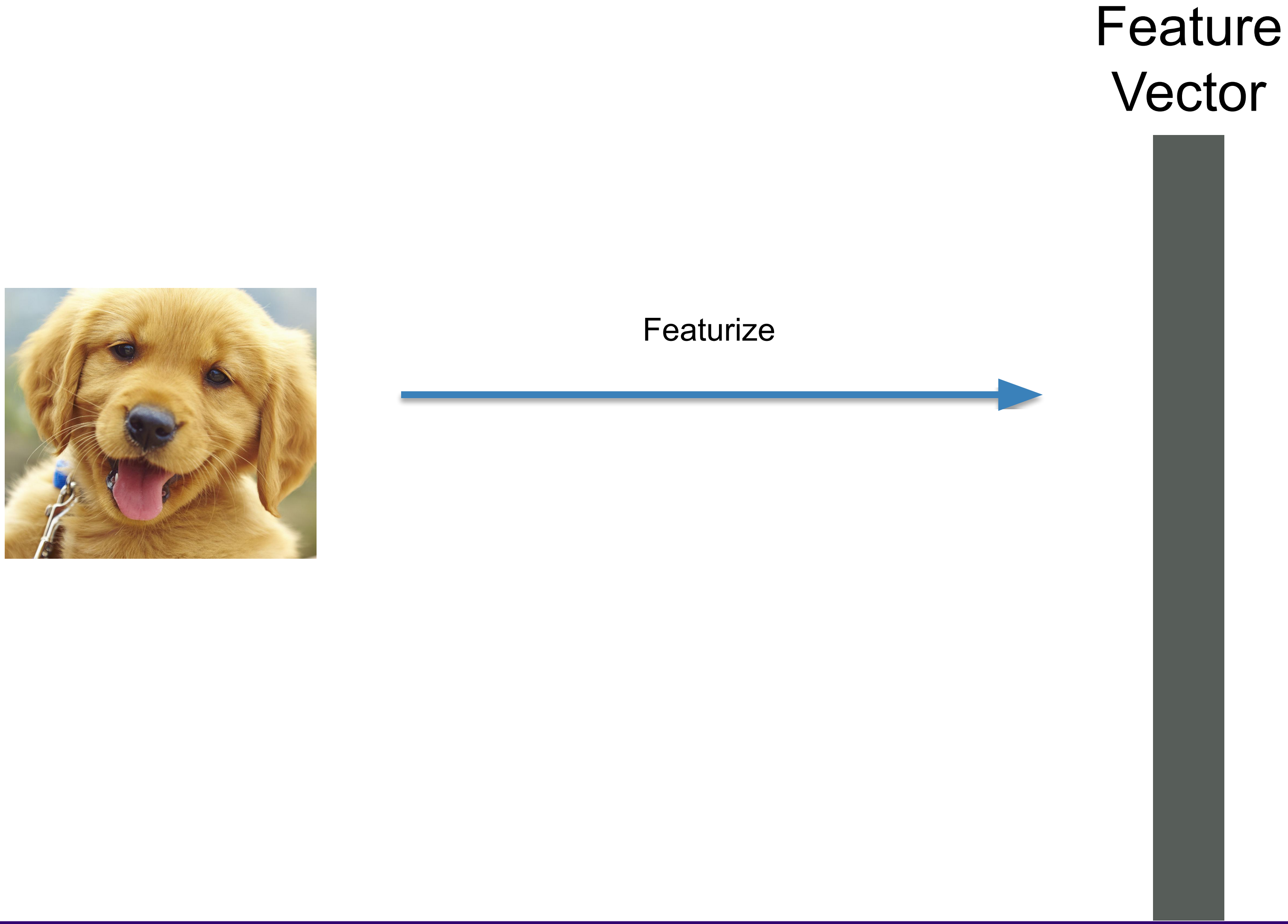
example 5x5 filters
(32 total)

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

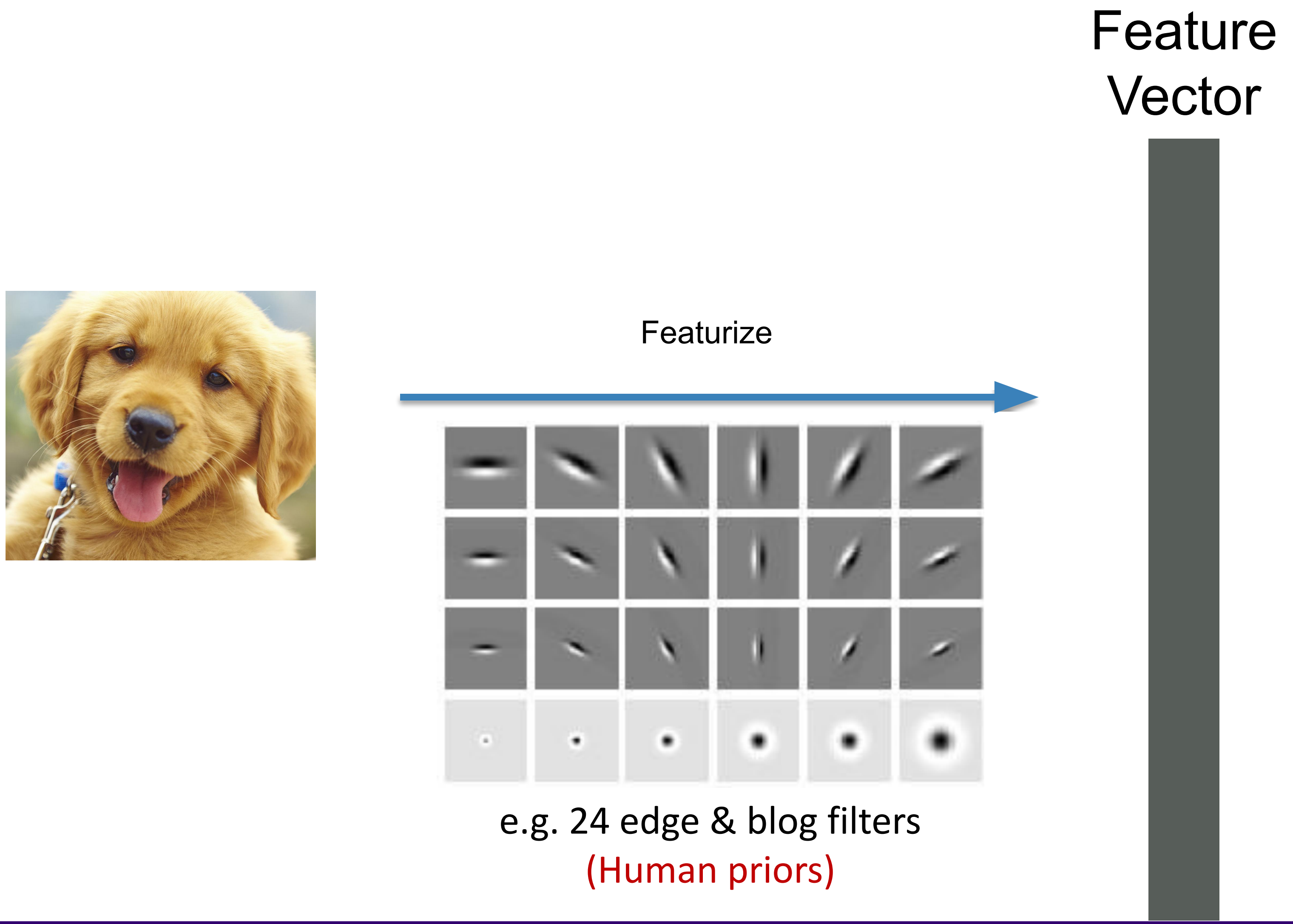
$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

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

Recall:



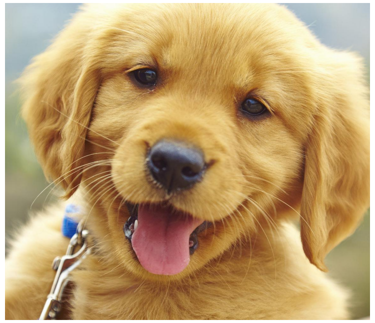
Recall:



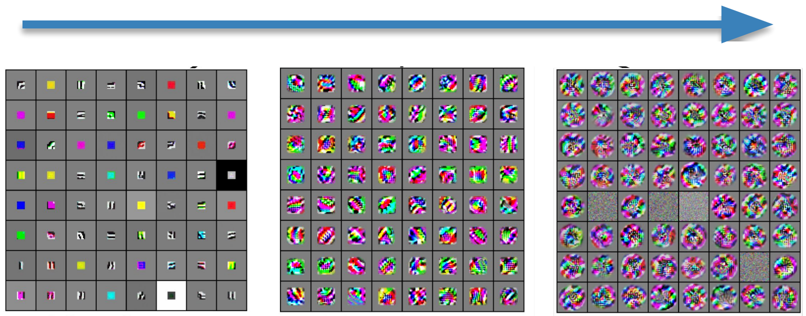
Feature Vector
Featurize
e.g. 24 edge & blog filters
(Human priors)

With CNN:

Feature Vector



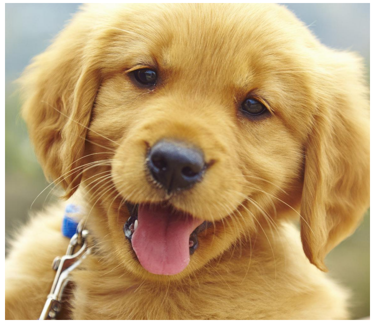
Featurize



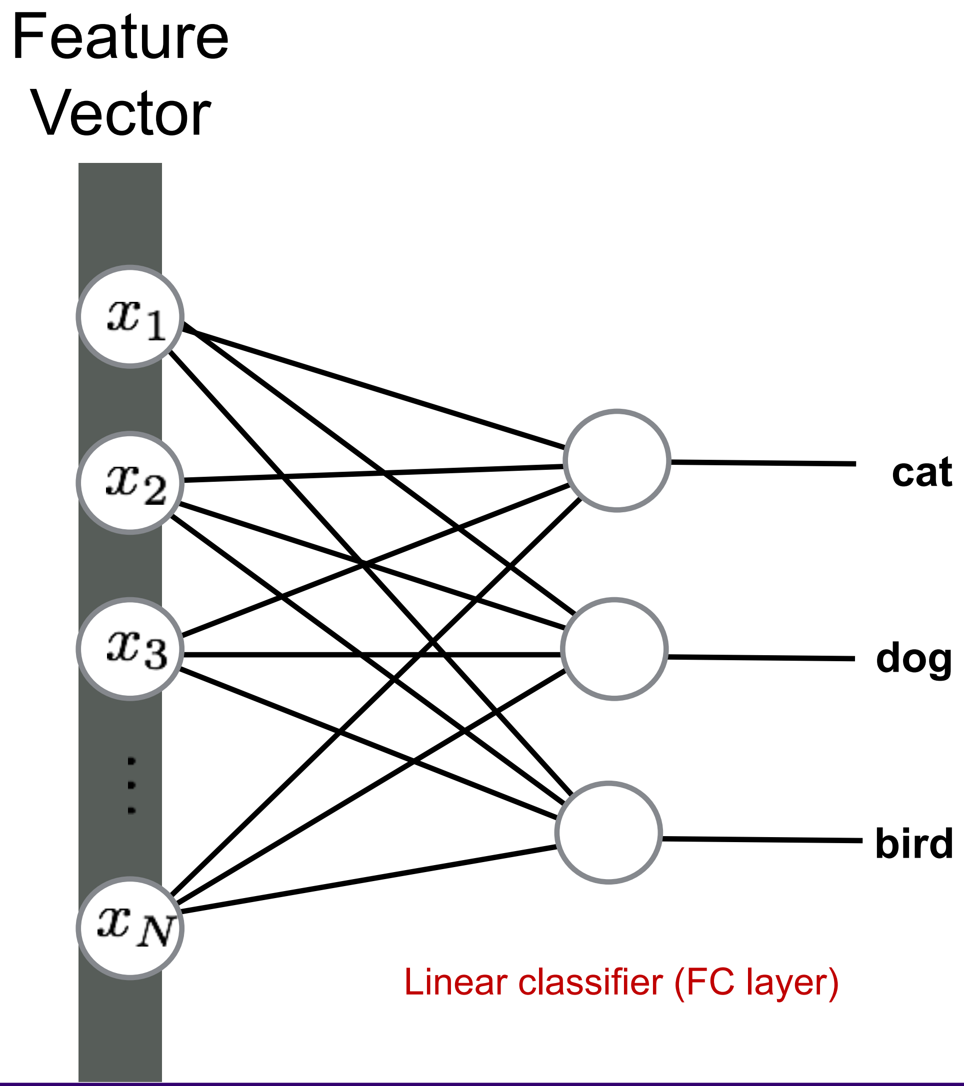
CNN: Learning (more diverse) filters for you!!

CNN (or Neural Network)

Representation (feature) learning + linear classifier

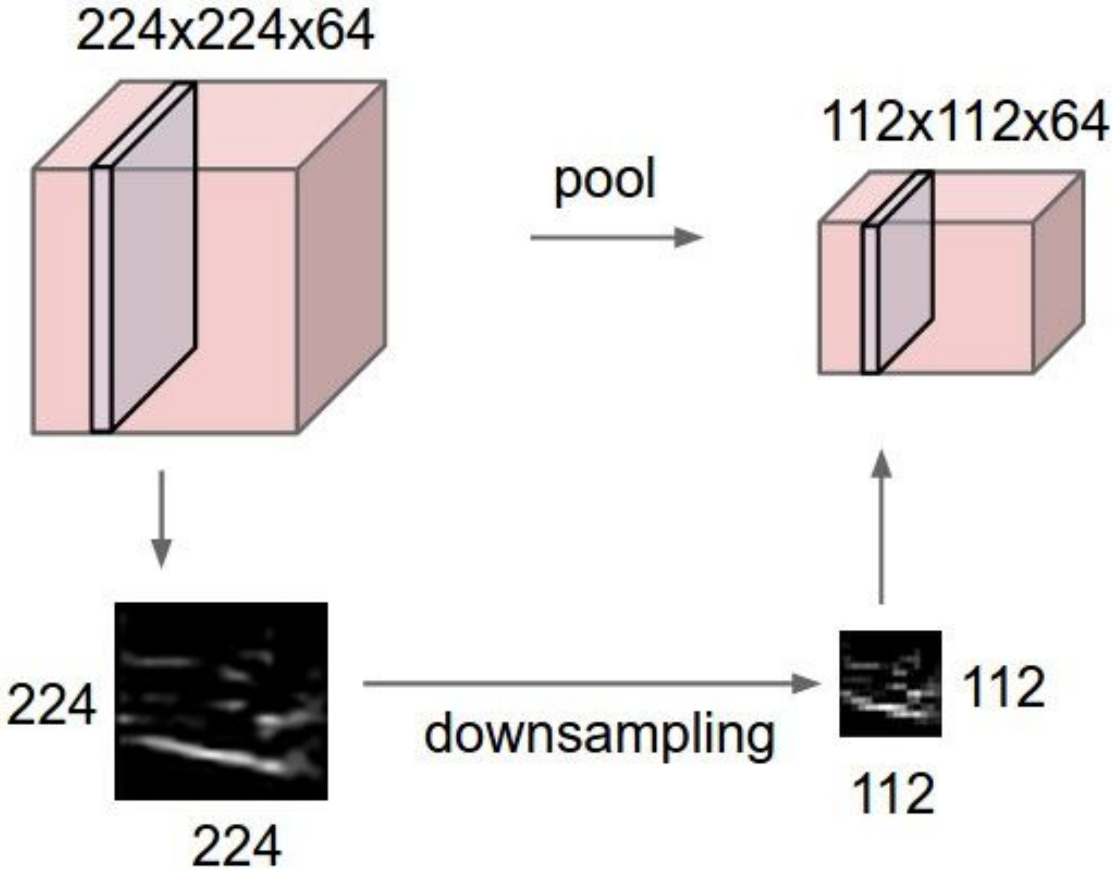


CNN: Learning (more diverse) filters for you!!

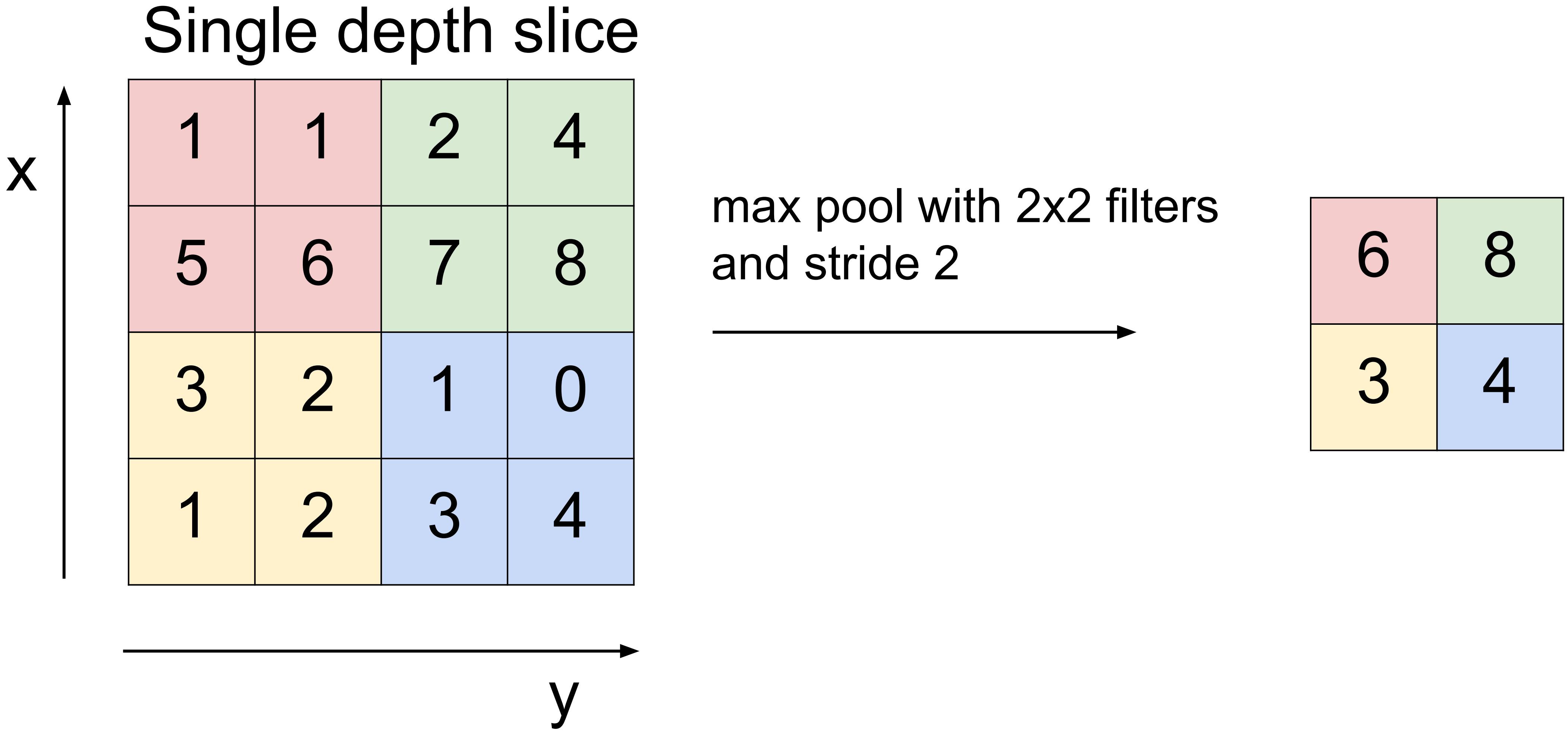


Pooling layer

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



MAX POOLING


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

Pooling layer: summary

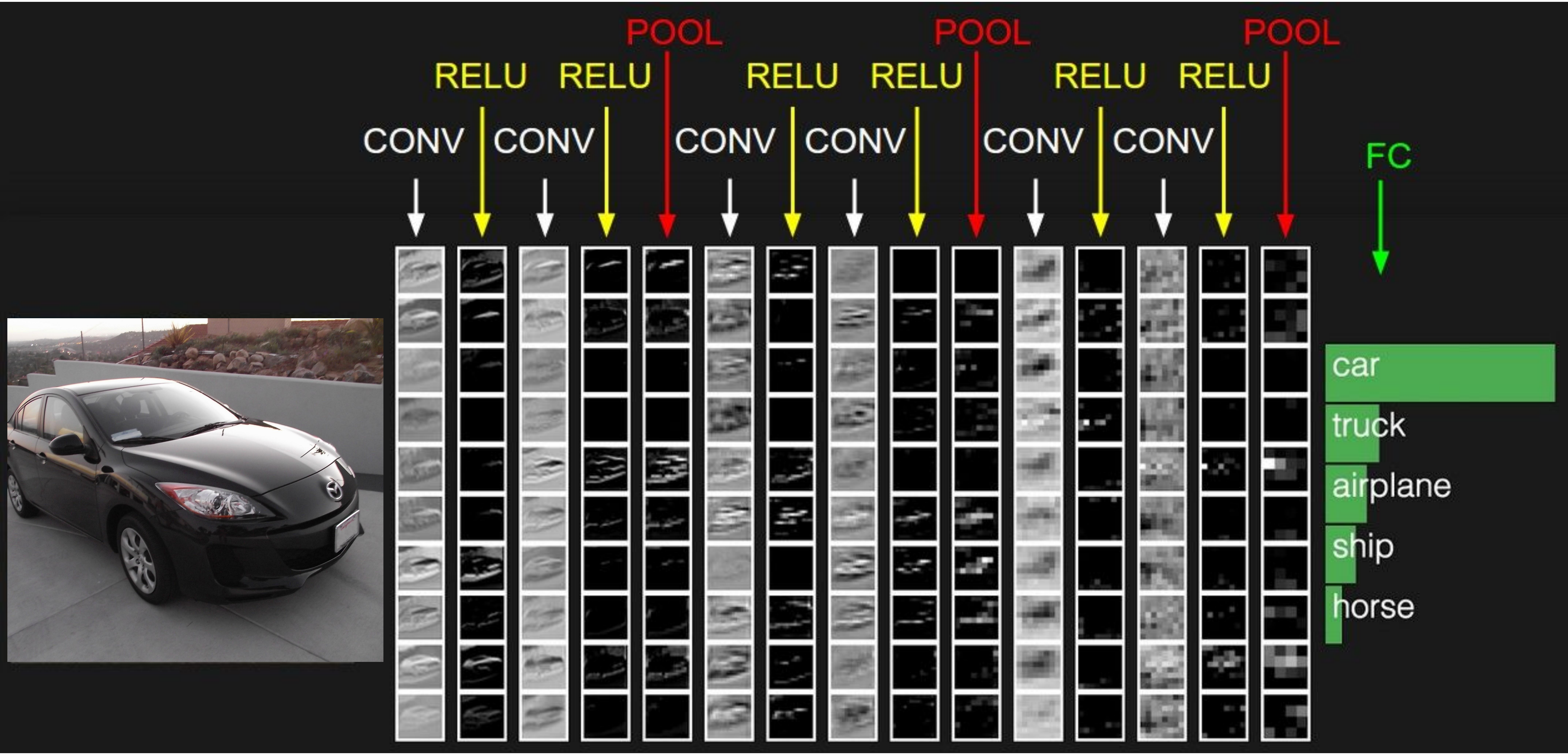
Let's assume input is W_1 x H_1 x C
Pooling layer needs 2 hyperparameters:

- The kernel size **K**
- The stride **S**

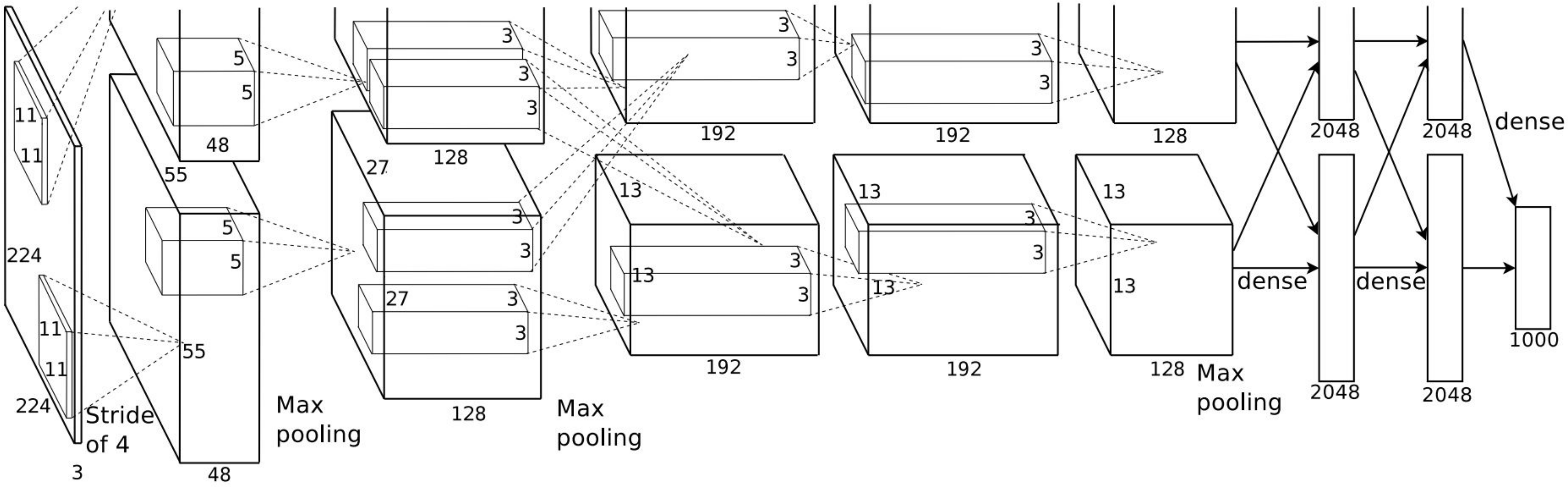
This will produce an output of W_2 x H_2 x C where:

- $W_2 = (W_1 - K)/S + 1$
- $H_2 = (H_1 - K)/S + 1$

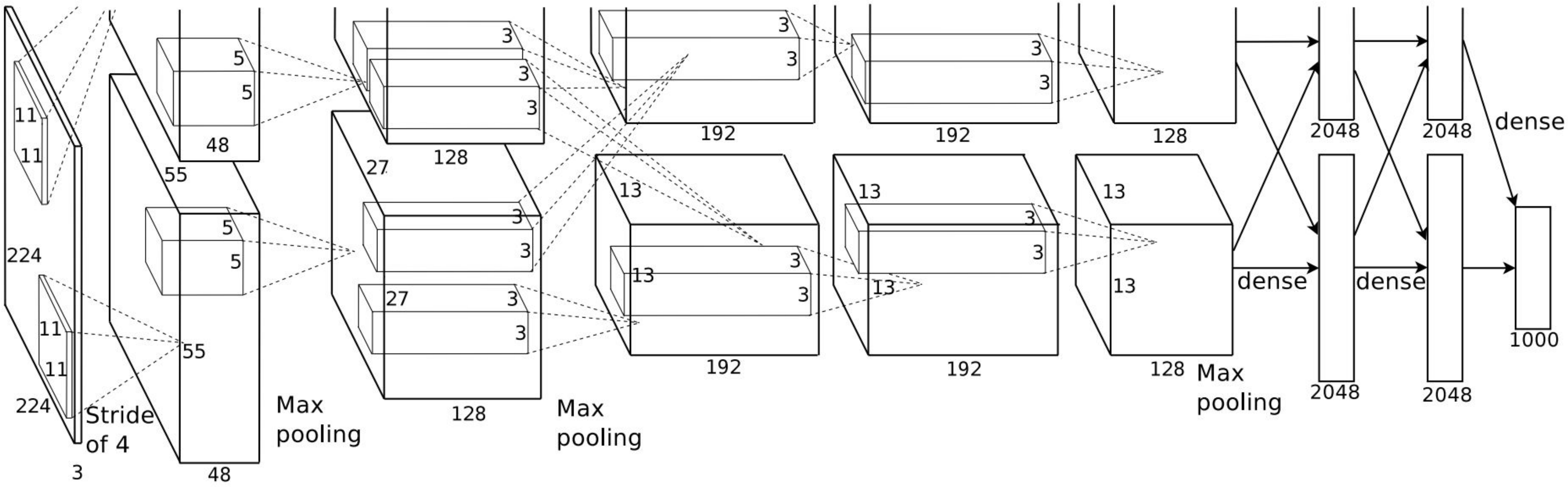
Number of parameters: 0


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

AlexNet (2012) – The start of modern deep learning



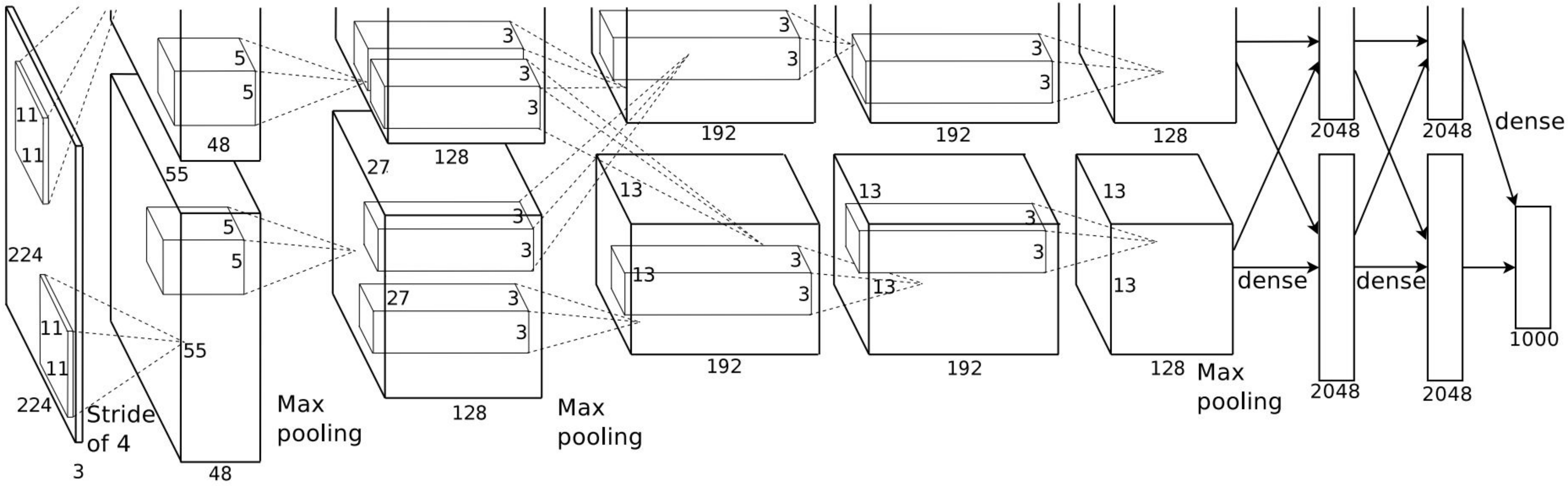
AlexNet (2012) – The start of modern deep learning



Common Practice:

- Reduce the spatial dimension while increasing channels (why?)
- Output size is no larger than the input size (why?)

AlexNet (2012) – The start of modern deep learning

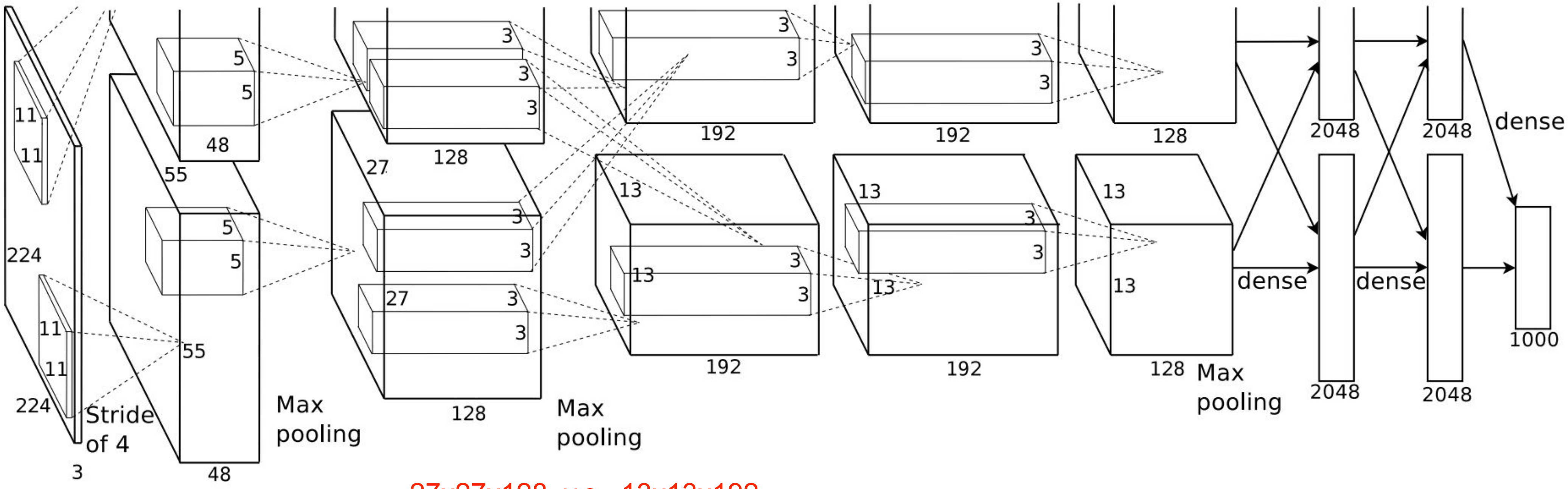


224x224x3 v.s. 55x55x48

Common Practice:

- Reduce the spatial dimension while increasing channels
- Output size is no larger than the input size

AlexNet (2012) – The start of modern deep learning



27x27x128 v.s. 13x13x192
Reduction ratio: (1/2)x(1/2)*1.5=0.375

Common Practice:
- Reduce the spatial dimension while increasing channels
- Output size is no larger than the input size

Example: DPM is CNN

Deformable Part Models are Convolutional Neural Networks

Tech report

Ross Girshick Forrest Iandola Trevor Darrell Jitendra Malik
UC Berkeley
{rbg,forresti,trevor,malik}@eecs.berkeley.edu

Abstract

Deformable part models (DPMs) and convolutional neural networks (CNNs) are two widely used tools for visual recognition. They are typically viewed as distinct approaches: DPMs are graphical models (Markov random fields), while CNNs are "black-box" non-linear classifiers. In this paper, we show that a DPM can be formulated as a CNN, thus providing a novel synthesis of the two ideas. Our construction involves unrolling the DPM inference algorithm and mapping each step to an equivalent (and at times novel) CNN layer. From this perspective, it becomes natural to replace the standard image features used in DPM with a learned feature extractor. We call the resulting model DeepPyramid DPM and experimentally validate it on PASCAL VOC. DeepPyramid DPM significantly outperforms DPMs based on histograms of oriented gradients features (HOG) and slightly outperforms a comparable version of the recently introduced R-CNN detection system, while running an order of magnitude faster.

1. Introduction

Part-based representations are widely used for visual recognition tasks. In particular, deformable part models (DPMs) [7] have been especially useful for generic object category detection. DPMs update pictorial structure models [8, 11] (which date back to the 1970s) with modern image features and machine learning algorithms. Convolutional neural networks (CNNs) [12, 23, 27] are another influential class of models for visual recognition. CNNs also have a long history, and have come back into popular use in the last two years due to good performance on image classification [5, 22] and object detection [14, 28] tasks.

These two models, DPMs and CNNs, are typically viewed as distinct approaches to visual recognition. DPMs are graphical models (Markov random fields), while CNNs are "black-box" non-linear classifiers. In this paper we describe how a DPM can be formulated as an equivalent CNN, providing a novel synthesis of these ideas. This formulation (DPM-CNN) relies on a new CNN layer, *distance transform pooling*, that generalizes max pooling. Another innovation of our approach is that rather than using histograms of oriented gradients (HOG) features [4], we apply DPM-CNN to a feature pyramid that is computed by another CNN. Since the end-to-end system is the function composition of two networks, it is equivalent to a single, unified CNN. We call this end-to-end model *DeepPyramid DPM*.

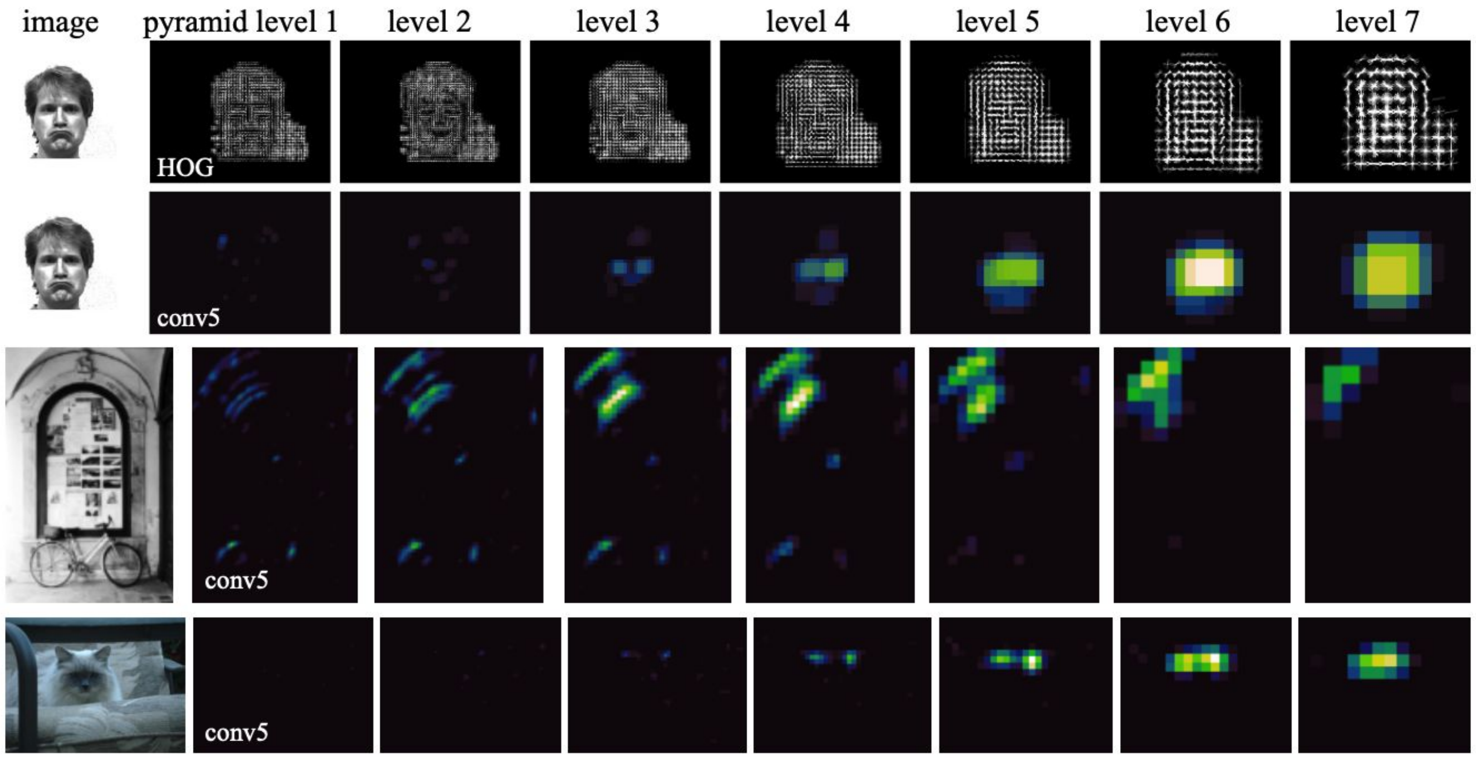
We also show that DeepPyramid DPM works well in practice. In terms of object detection mean average precision, DeepPyramid DPM slightly outperforms a comparable version of the recently proposed R-CNN [14] (specifically, R-CNN on the same $conv_5$ features, without fine-tuning), while running about 20x faster. This experimental investigation also provides a greater understanding of the relative merits of region-based detection methods, such as R-CNN, and sliding-window methods like DPM. We find that regions and sliding windows are complementary methods that will likely benefit each other if used in an ensemble.

HOG-based detectors are currently used in a wide range of models and applications, especially those where region-based methods are ill-suited (poselets [1] being a prime example). Our results show that sliding-window detectors on deep feature pyramids significantly outperform equivalent models on HOG. Therefore, we believe that the model presented in this paper will be of great practical interest to the visual recognition community. An open-source implementation will be made available, which will allow researchers to easily build on our work.

2. DeepPyramid DPM

In this section we describe the DeepPyramid DPM architecture. DeepPyramid DPM is a convolutional neural network that takes as input an image pyramid and produces as output a pyramid of object detection scores. Although the model is a single CNN, for pedagogical reasons we describe it in terms of two smaller networks whose function composition yields the full network. A schematic diagram of the

1



Next time

Last Lecture: CV Frontier