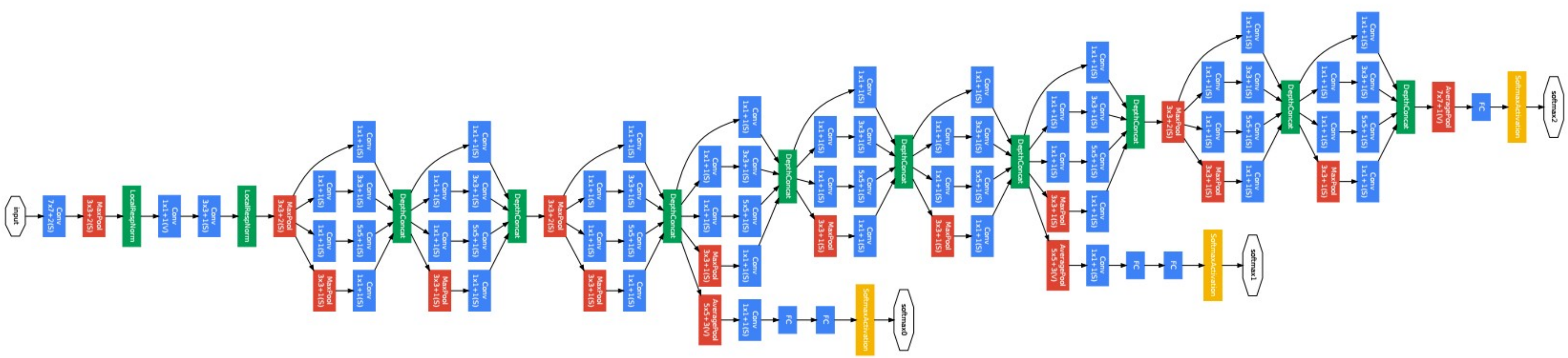
CSCI 497P/597P: Computer Vision

Scott Wehrwein

Convolutional Neural Networks and some of the practicalities that make them work



Reading

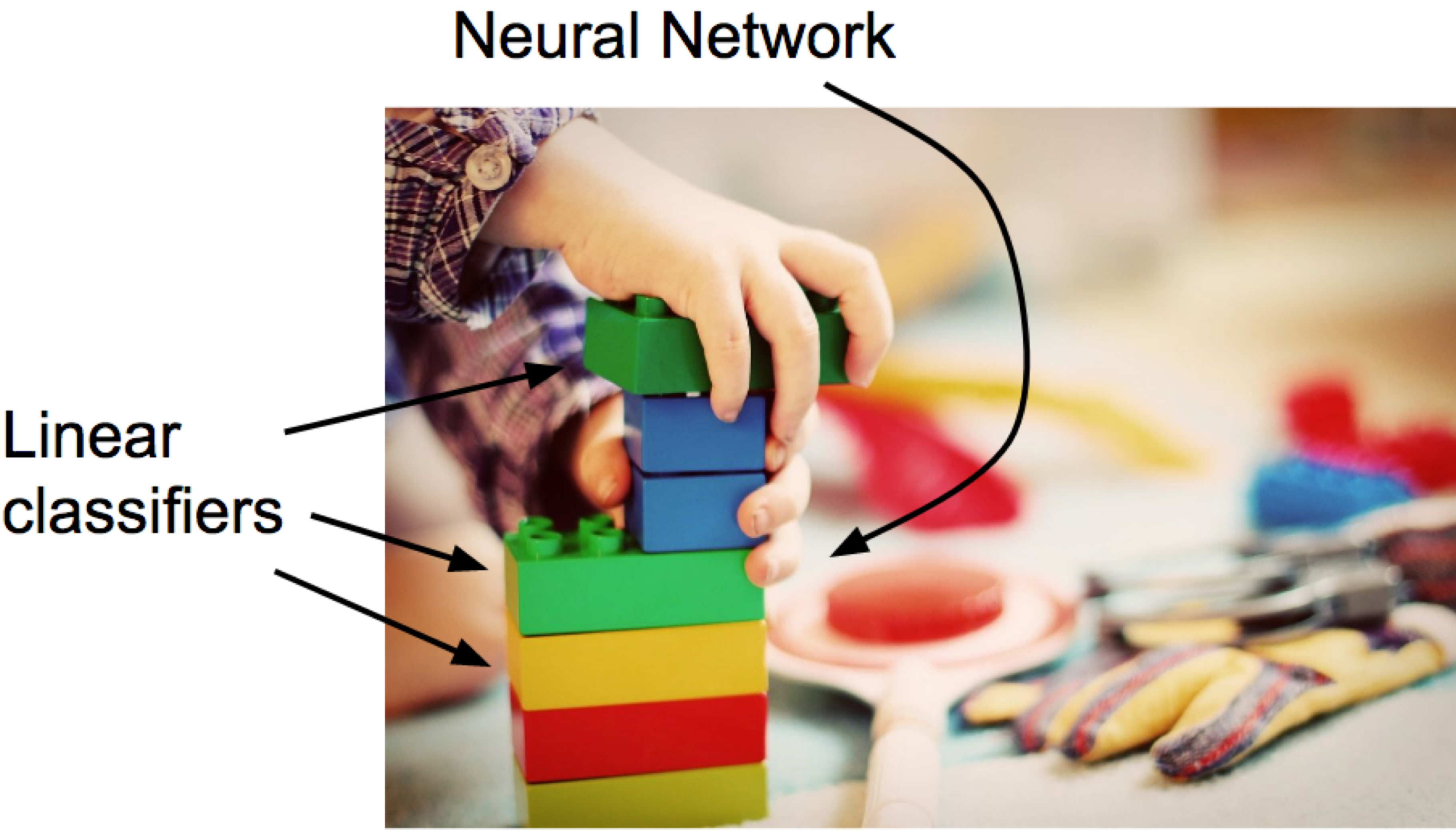
- http://cs231n.github.io/convolutional-networks/

Announcements

Goals

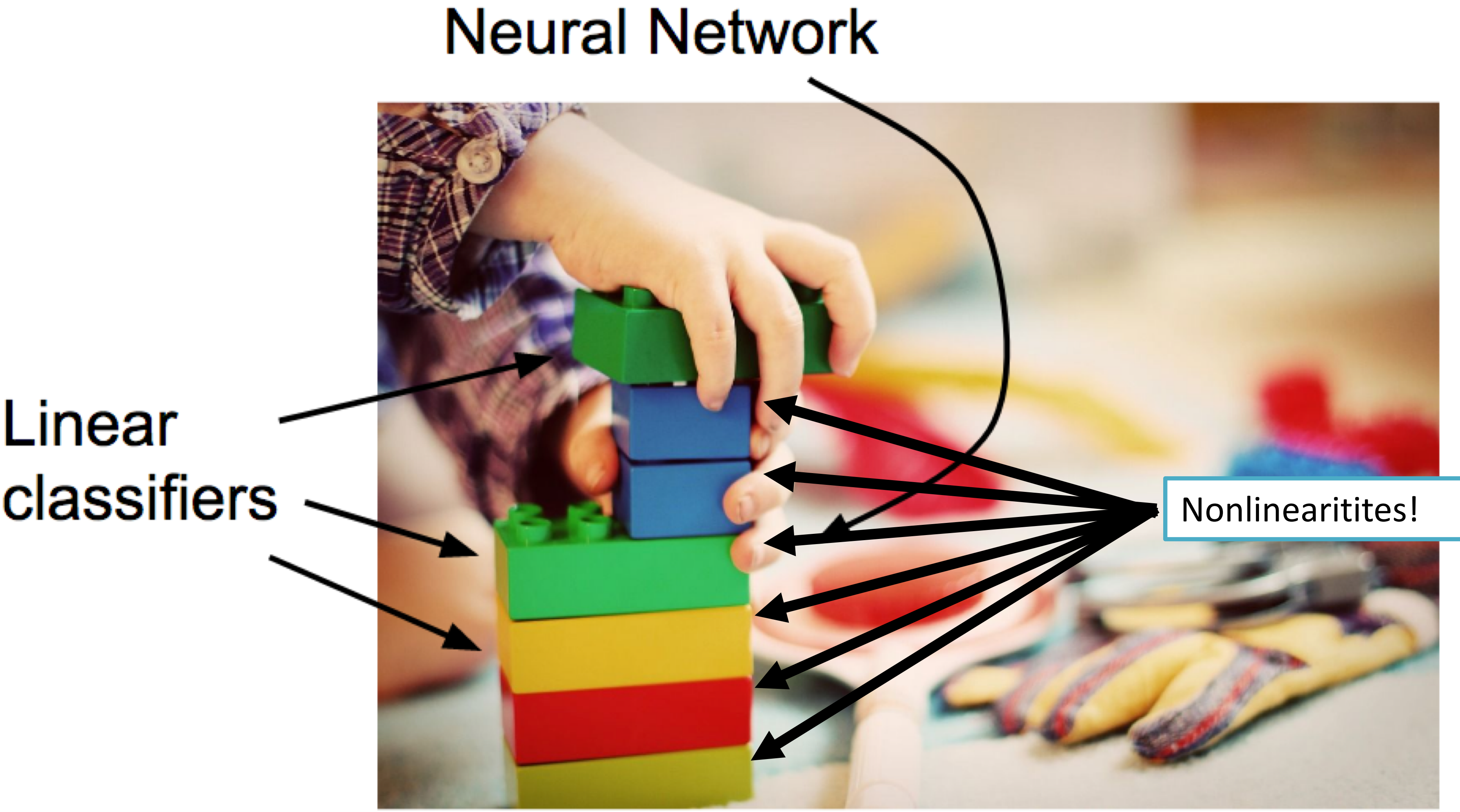
- Understand the motivation and behavior of convolutional layers in neural networks.
- Understand the degrees of freedom available in setting up a convolution layer:
  - Output channels, kernel size, padding, stride
- Know the meaning of the various basic layers involved in standard CNN architectures
  - Conv, ReLU, Pool, Fully Connected

Last time: Neural Networks



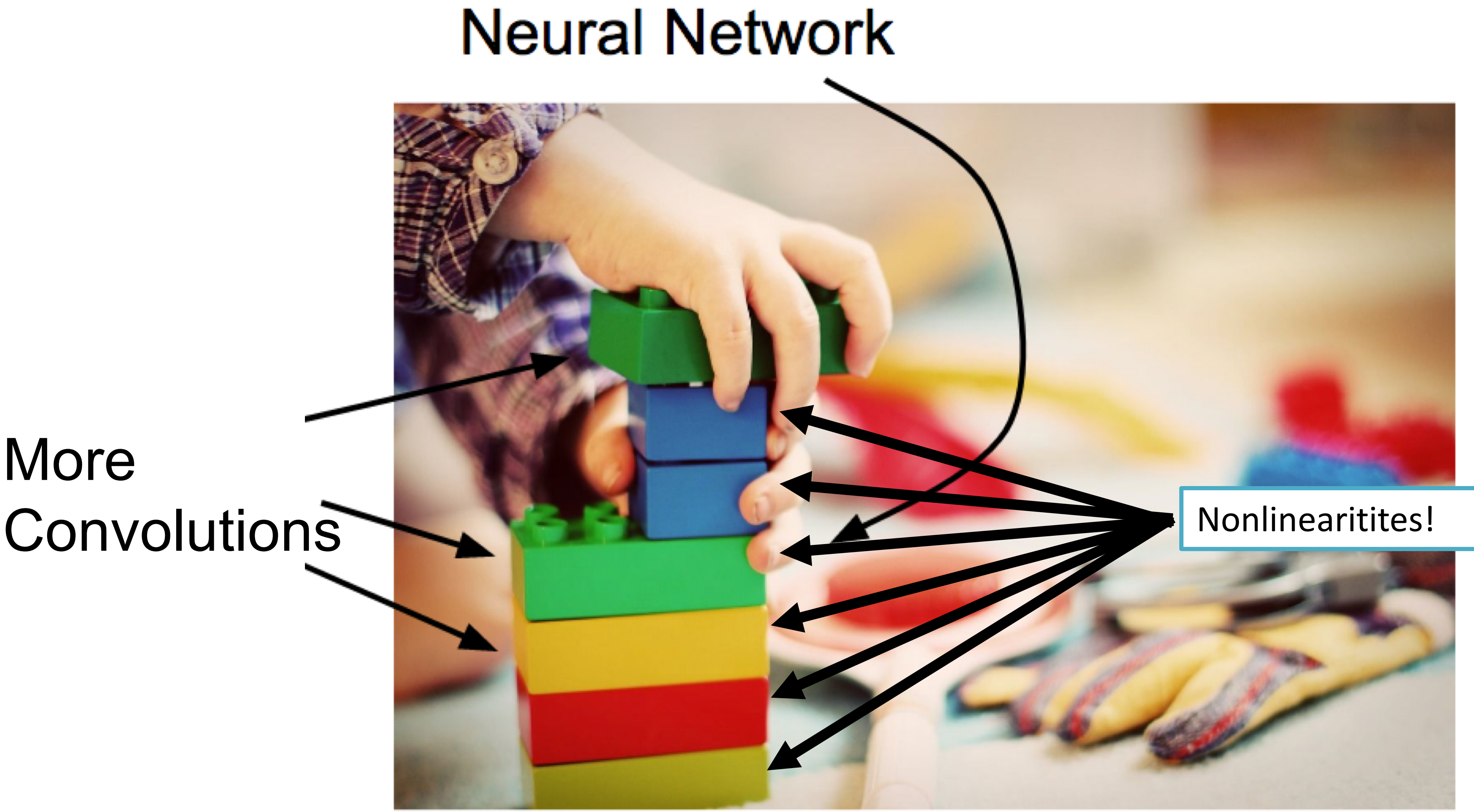
Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Last time: Neural Networks



Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

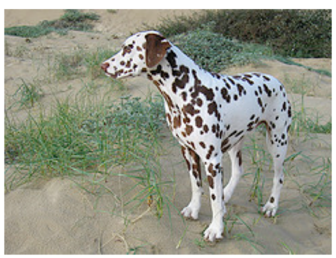
Today: Convolutional Neural Networks



Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Taking a step back: Image Recognition

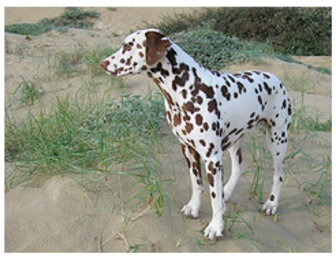
- We have images; ML works on vectors.
- To do machine learning, we need a function that takes an image and converts it into a vector.

ϕ () = 

- Given an image, use ϕ to get a vector representing a point in high dimensional space

Classifying Images: Pipeline

1. Represent the image in some *feature space*

ϕ () = 

2. Classify the image based on its feature representation.

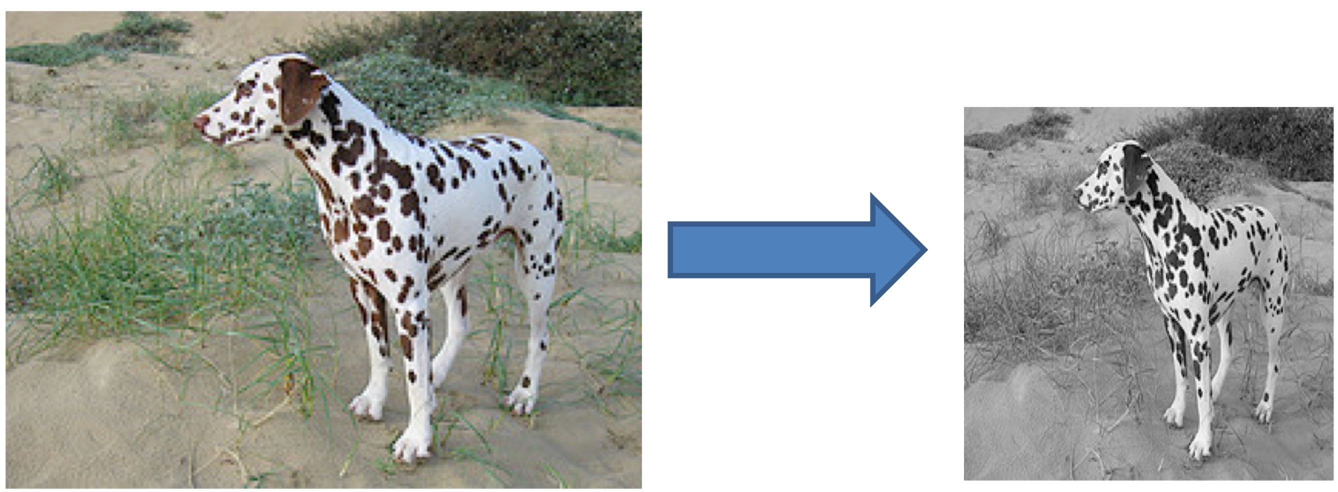
- h() = ”dog”

Two important pieces

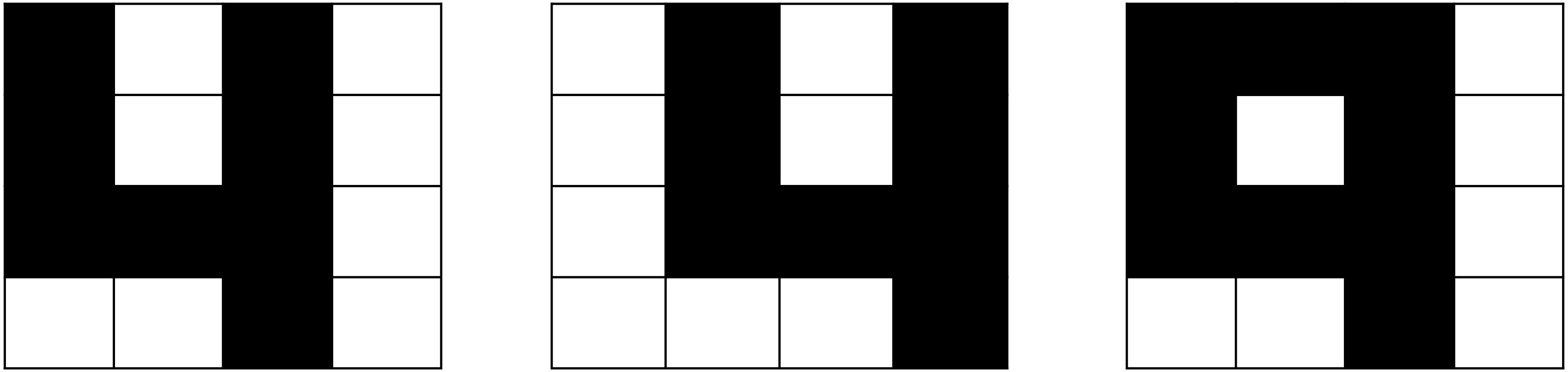
- The feature extractor (ϕ)
- The classifier (*h*)
  - (this is what we've been talking about this whole time: linear classifiers, now neural networks)

Let's make the simplest possible ϕ

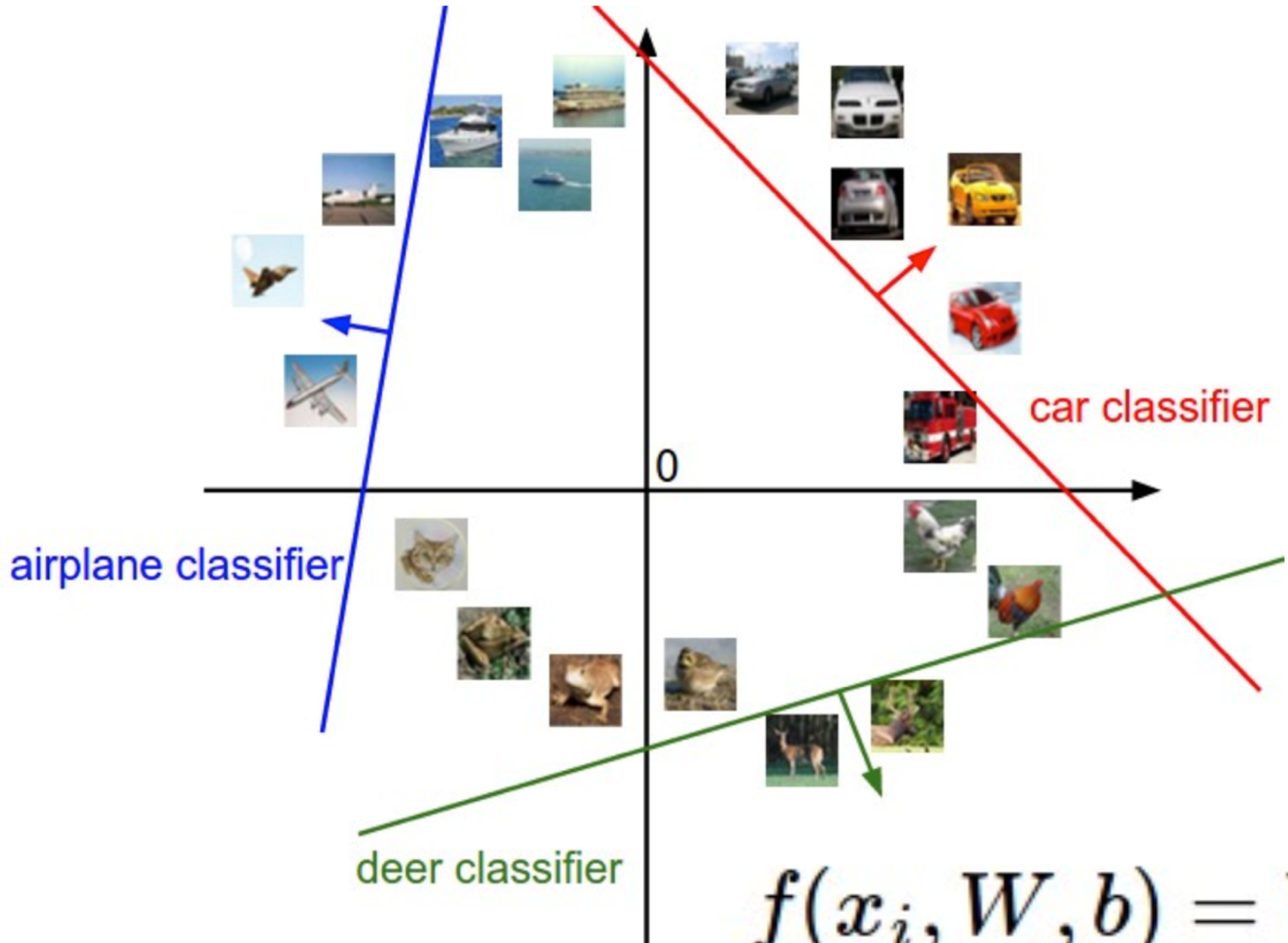
- Represent an image as a vector in $\mathbb{R}^d$
- Step 1: convert image to gray-scale and resize to fixed size



Linear classifiers on pixels are bad

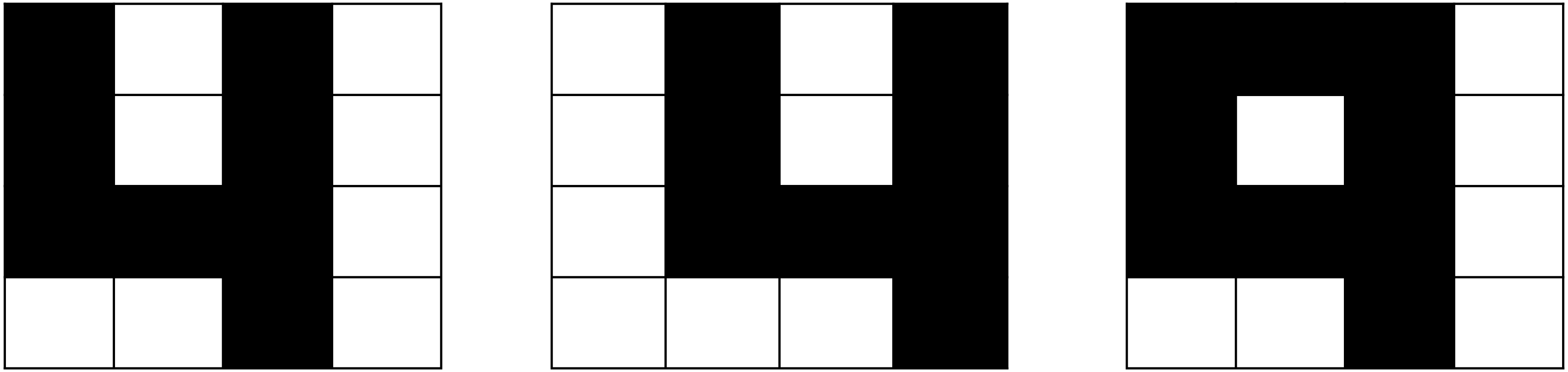


Linearly separable classes



$$f(x_i, W, b) = Wx_i + b$$

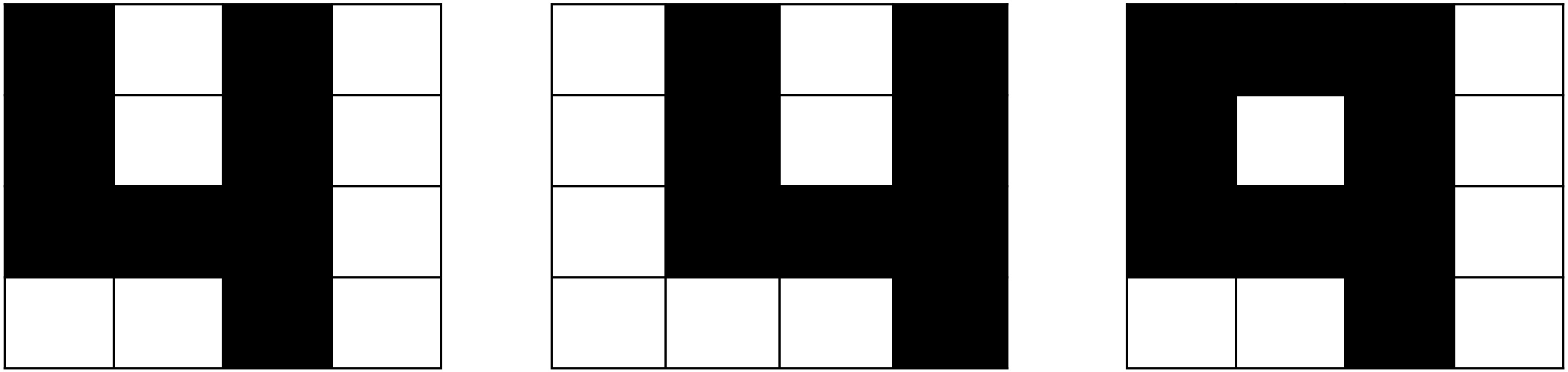
Linear classifiers on pixels are bad



How do we fix it?

- Solution 1: Better feature vectors
- Solution 2: Non-linear classifiers

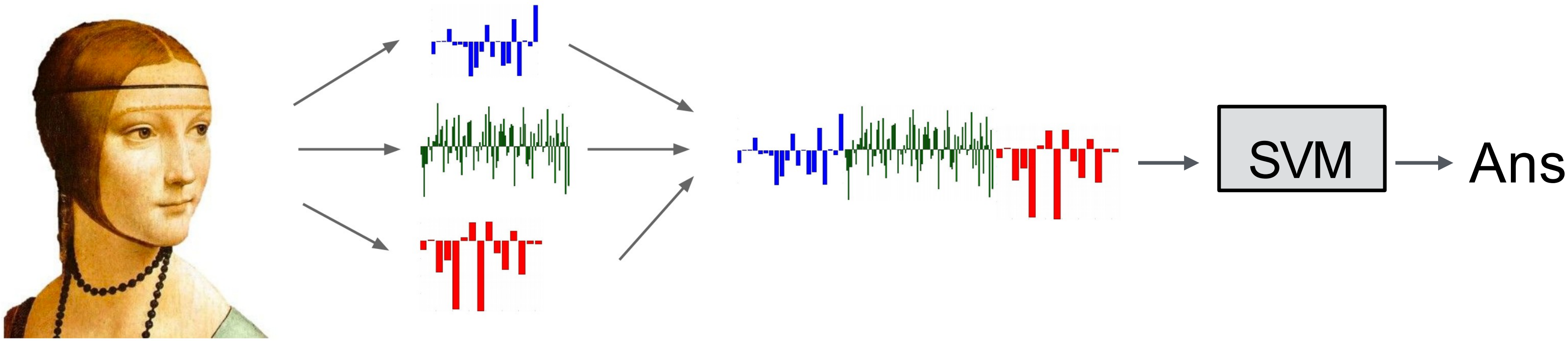
Linear classifiers on pixels are bad



How do we fix it?

- Solution 1: Better feature vectors
- Solution 2: Non-linear classifiers

Life Before Deep Learning



Input Pixels

Extract Hand-Crafted Features

Concatenate into a vector ***x***

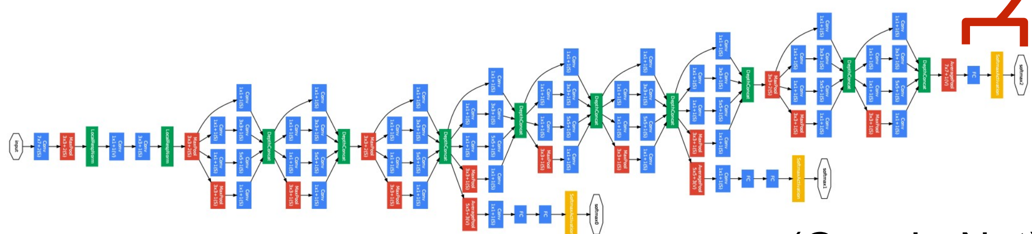
Linear Classifier

Key: cleverly design features so that by the time you get to the classifier, the classes are linearly separable

Figure: Karpathy 2016

The last layer of (most) CNNs are linear classifiers

This piece is just a linear classifier



(GoogLeNet)

→ Ans

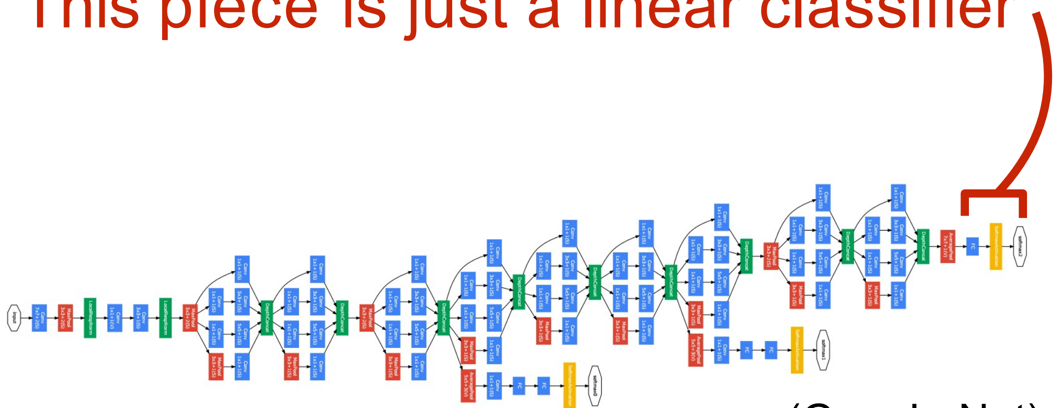
Input Pixels

Perform everything with a big neural network, trained end-to-end

Key: perform enough processing so that by the time you get to the end of the network, the classes are linearly separable

The last layer of (most) CNNs are linear classifiers

This piece is just a linear classifier



→ Ans

(GoogLeNet)

Input Pixels

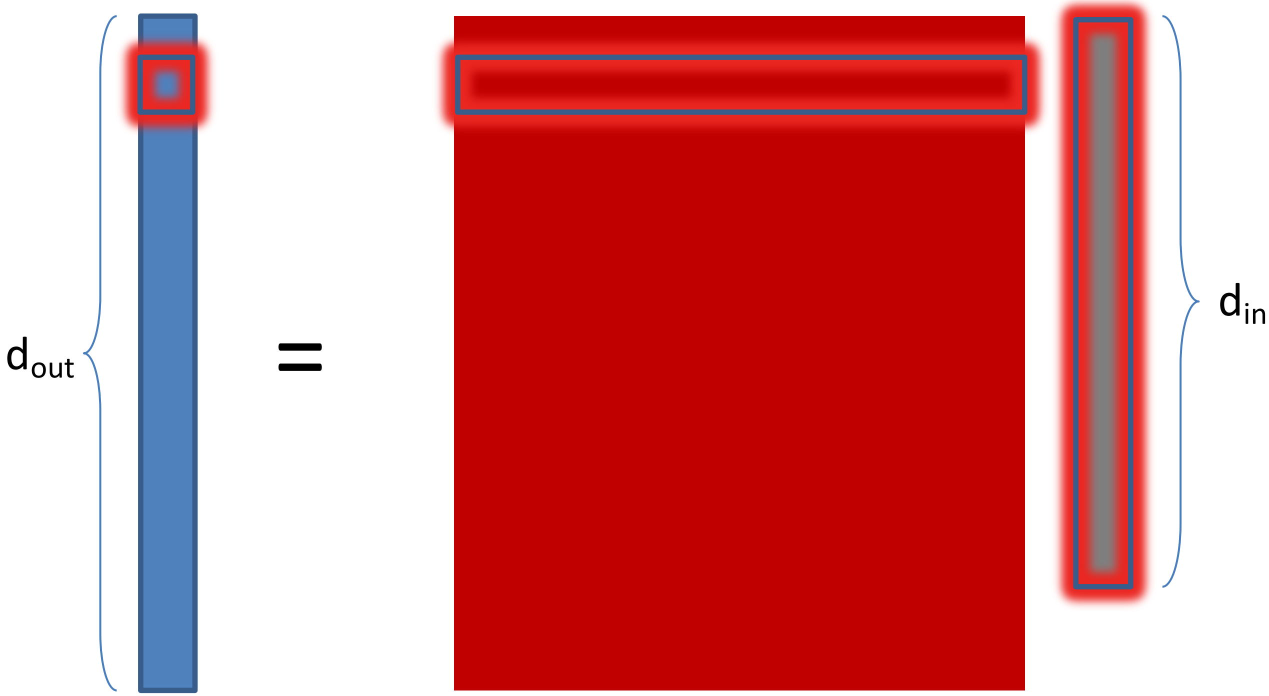
Perform everything with a big neural network, trained end-to-end

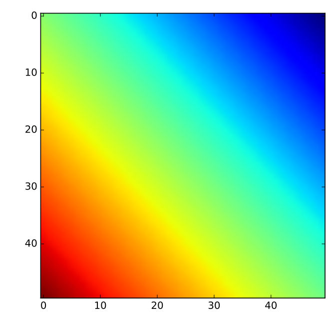
The network is the feature extractor *and* the classifier.

h **swallowed ϕ!**

A Linear Classifier

- $y = Wx + b$
- Every row of y corresponds to a hyperplane in x space

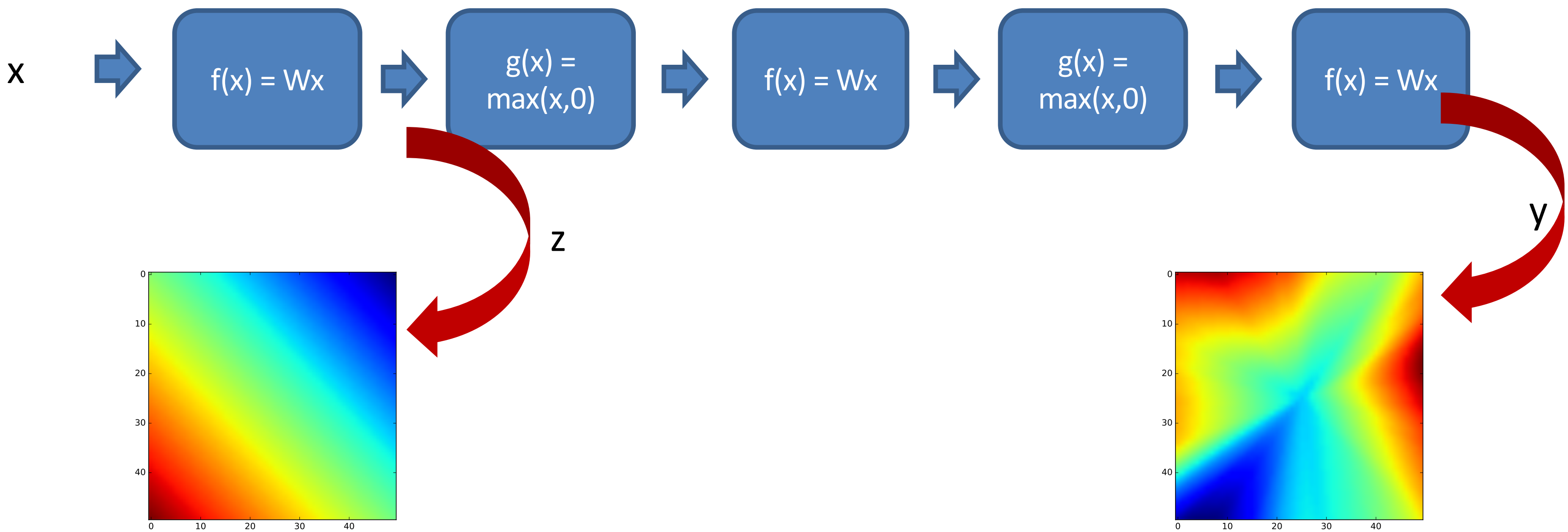




The case when $d_{in} = 2$. A single row in y plotted for every possible value of x

A Neural Network

- Key idea: build complex functions by composing simple functions

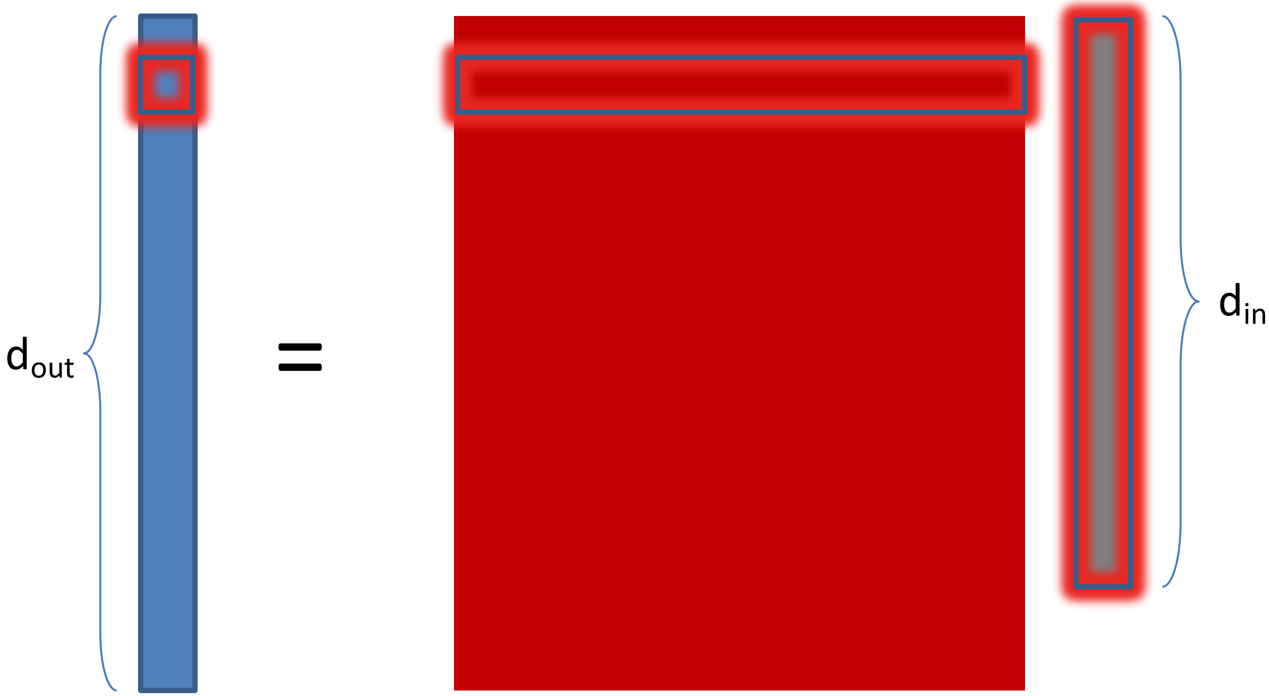


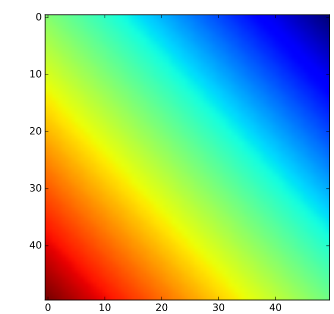
1 row of z plotted for every value of x

1 row of y plotted for every value of x

Linear Classifier: Parameter Count

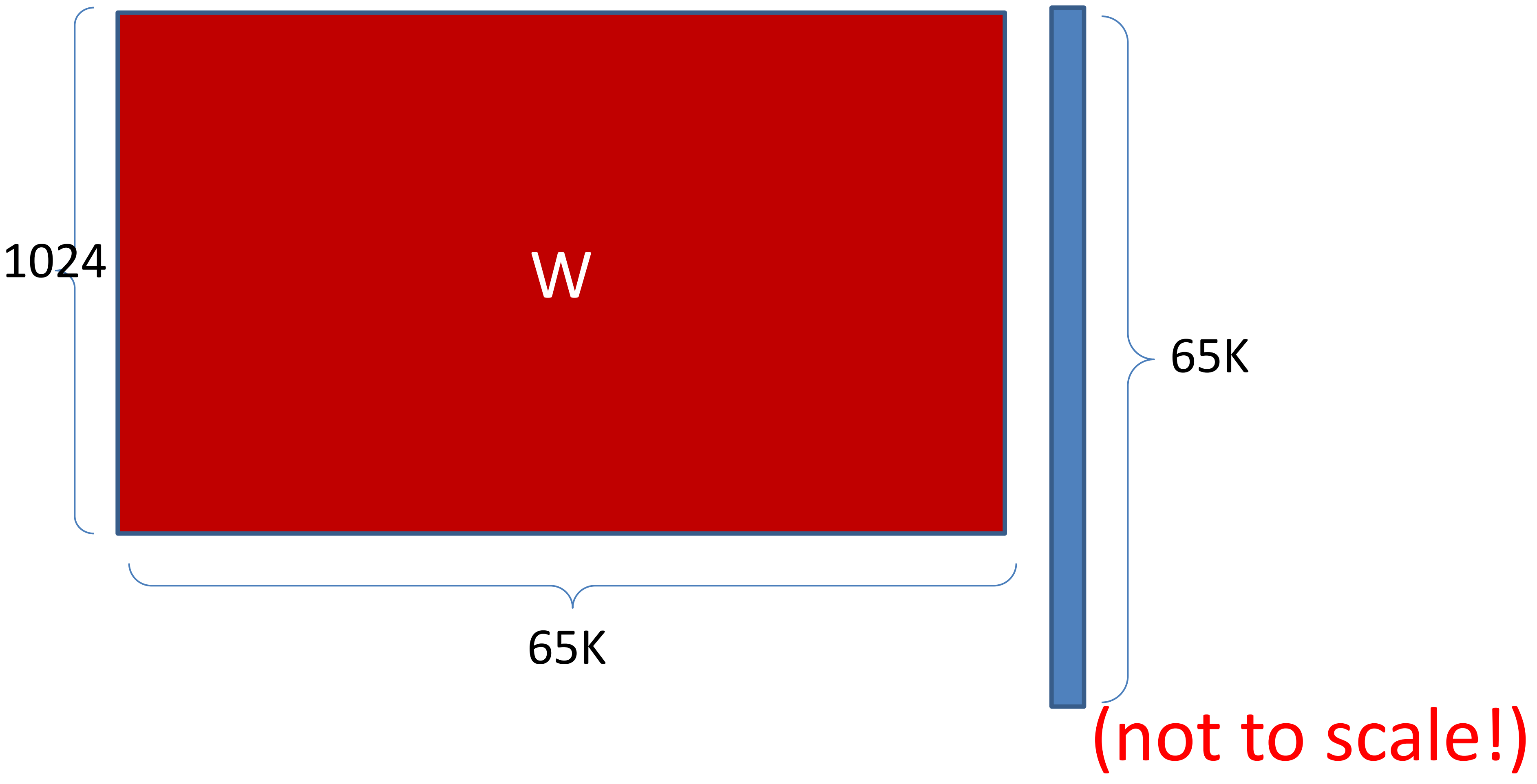
- How many parameters does a linear function have? Suppose:
  - # pixels = 256*256 = 65536
  - # classes = 1024





The case when d_{in} = 2. A single row in y plotted for every possible value of x

The linear function for images



Linear Classifier: Parameter Count

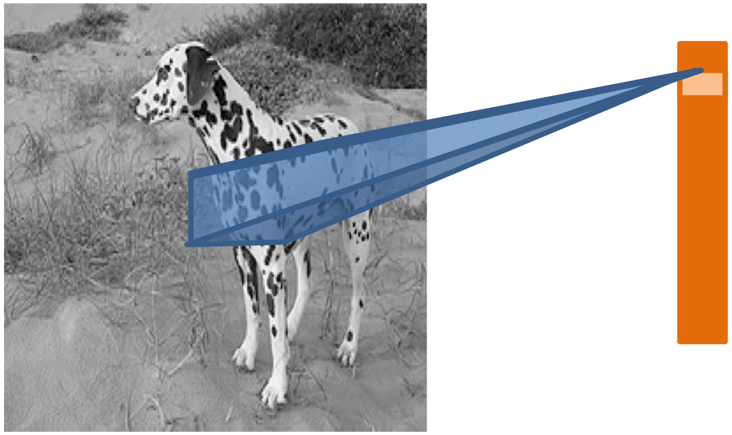
- How many parameters does a linear function have? Suppose:
  - # pixels = 256*256 = 65536 = 2^{16}
  - # classes = 1024 = 2^{10}

Linear Classifier: Parameter Count

- How many parameters does a linear function have? Suppose:
  - # pixels = 256*256 = 65536 = 2^{16}
  - # classes = 1024 = 2^{10}
- 2^{26} parameters for a one-layer network on a tiny image.
- More layers means more parameters:
  - more computation
  - difficult to train
- Can we make better use of parameters?

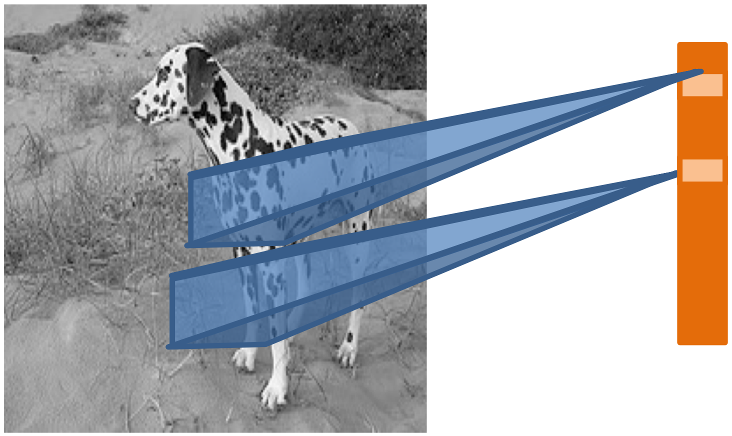
Idea 1: local connectivity

- Pixels only connected to *nearby* pixels in the prior layer



Idea 2: Translation invariance

- Pixels only connected to *nearby* pixels
- Weights should not depend on the location of the neighborhood



Linear function + translation invariance = *convolution*

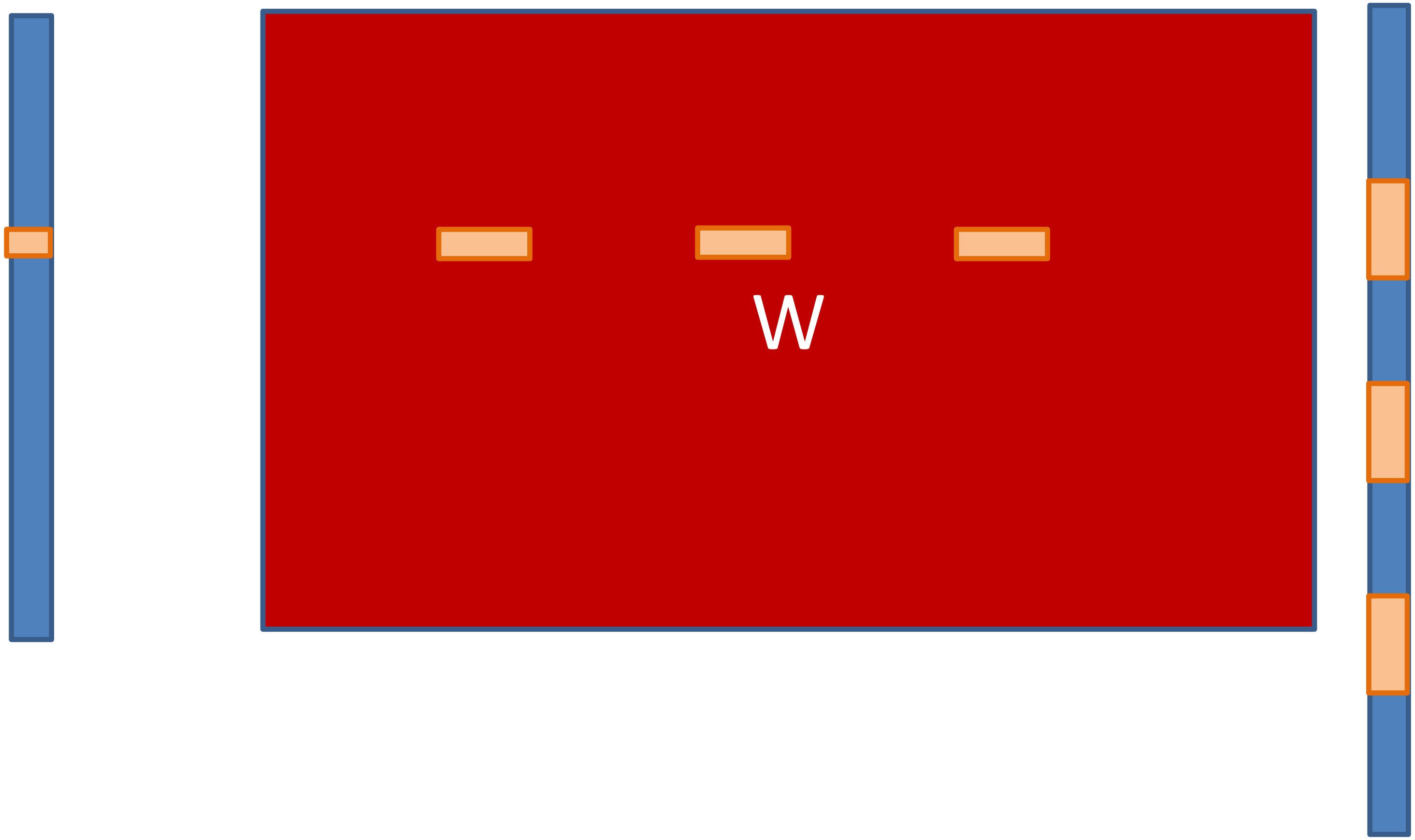
- Local connectivity determines kernel size

5.4	0.1	3.6
1.8	2.3	4.5
1.1	3.4	7.2

Convolution is still linear

Convolution layers can be written as matrix multiplications

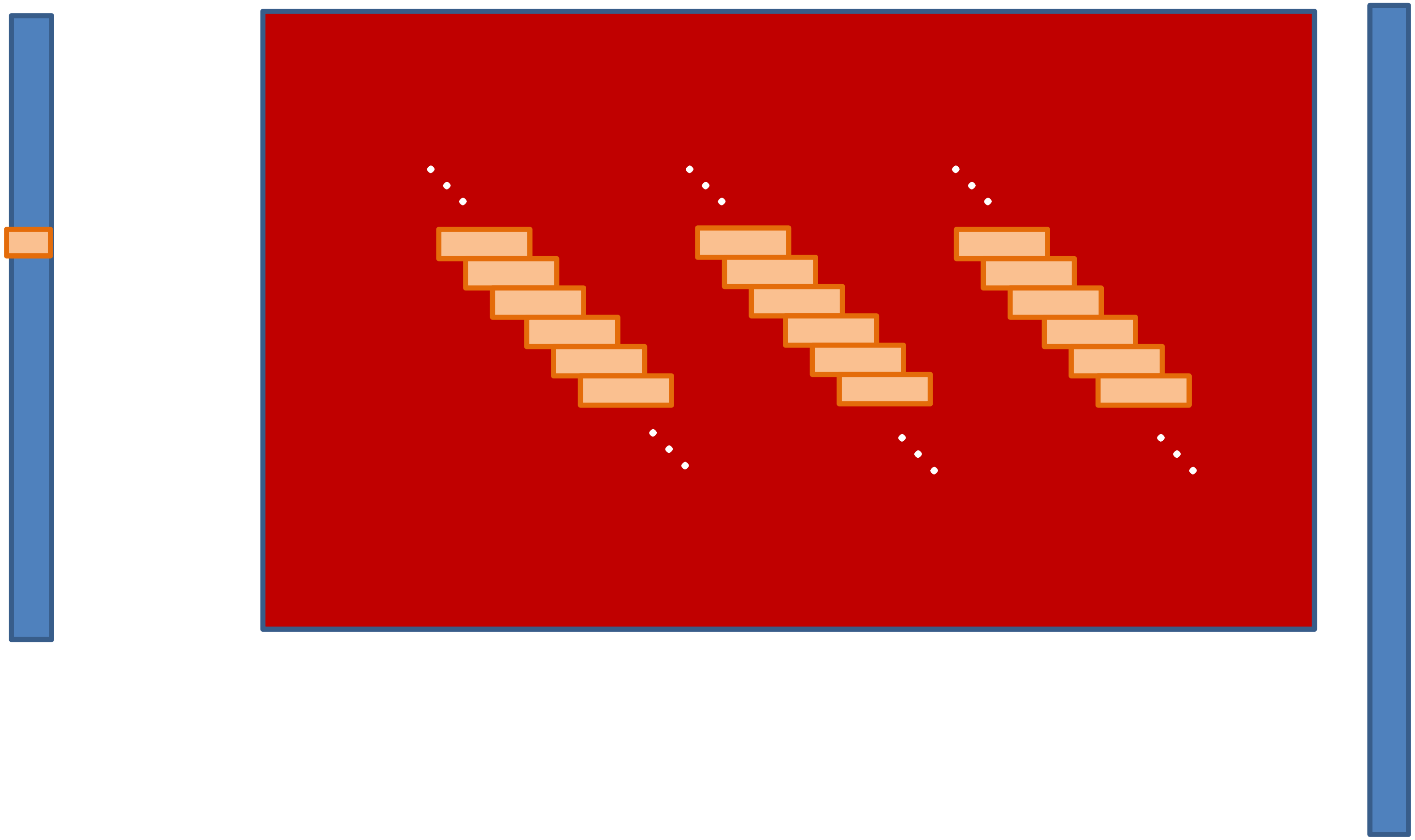
- The matrix is sparse: an output pixel only depends on neighboring inputs.



Convolution is still linear

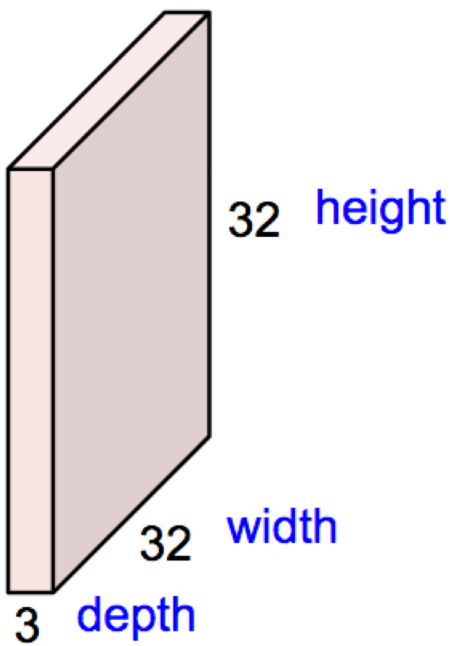
Convolution layers can be written as matrix multiplications

- The matrix is sparse: an output pixel only depends on neighboring inputs.
- The weights are shared across rows of W!

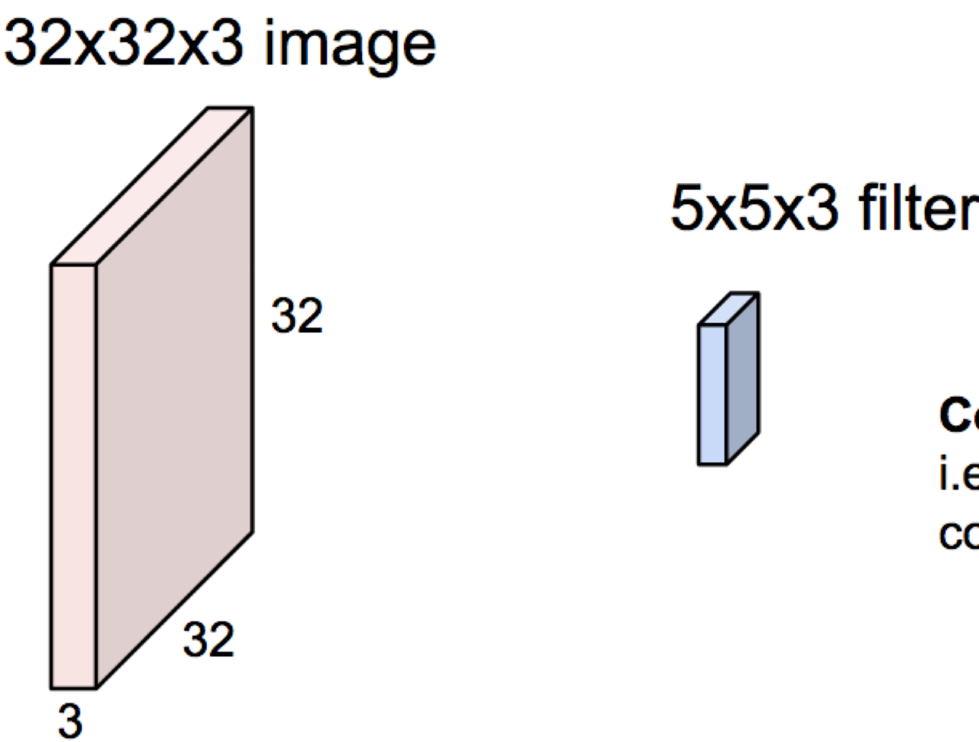


Convolution Layer

32x32x3 image -> preserve spatial structure



Convolution Layer

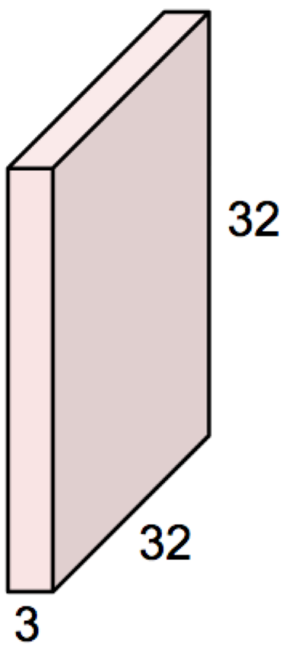


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

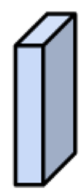
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

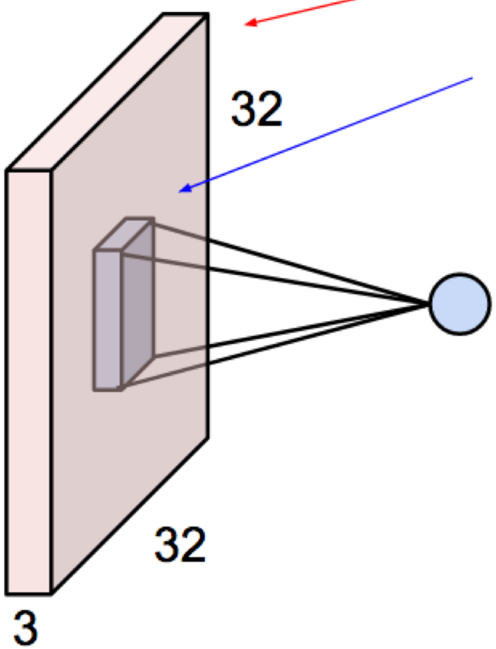


5x5x3 filter



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

Convolution Layer



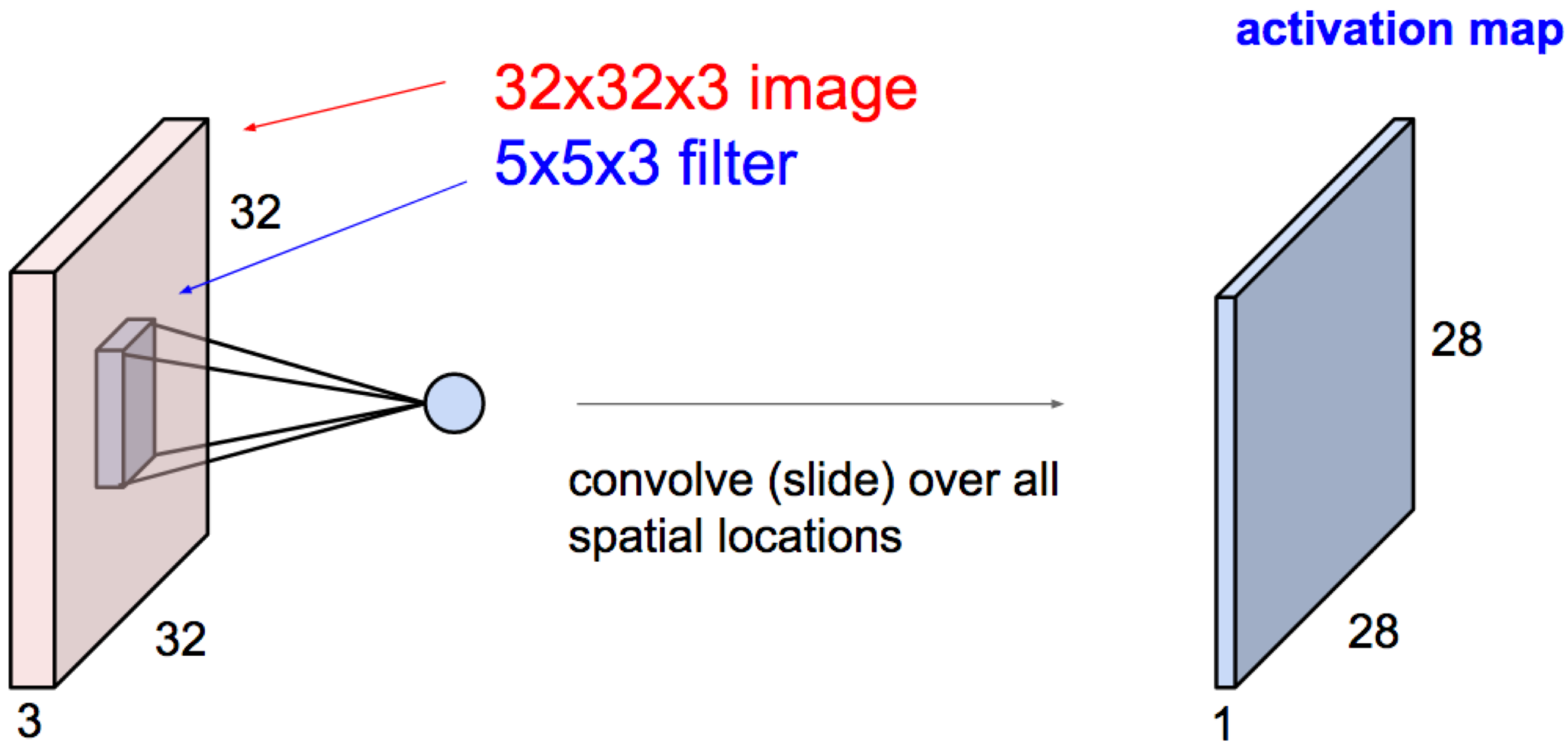
32x32x3 image

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)

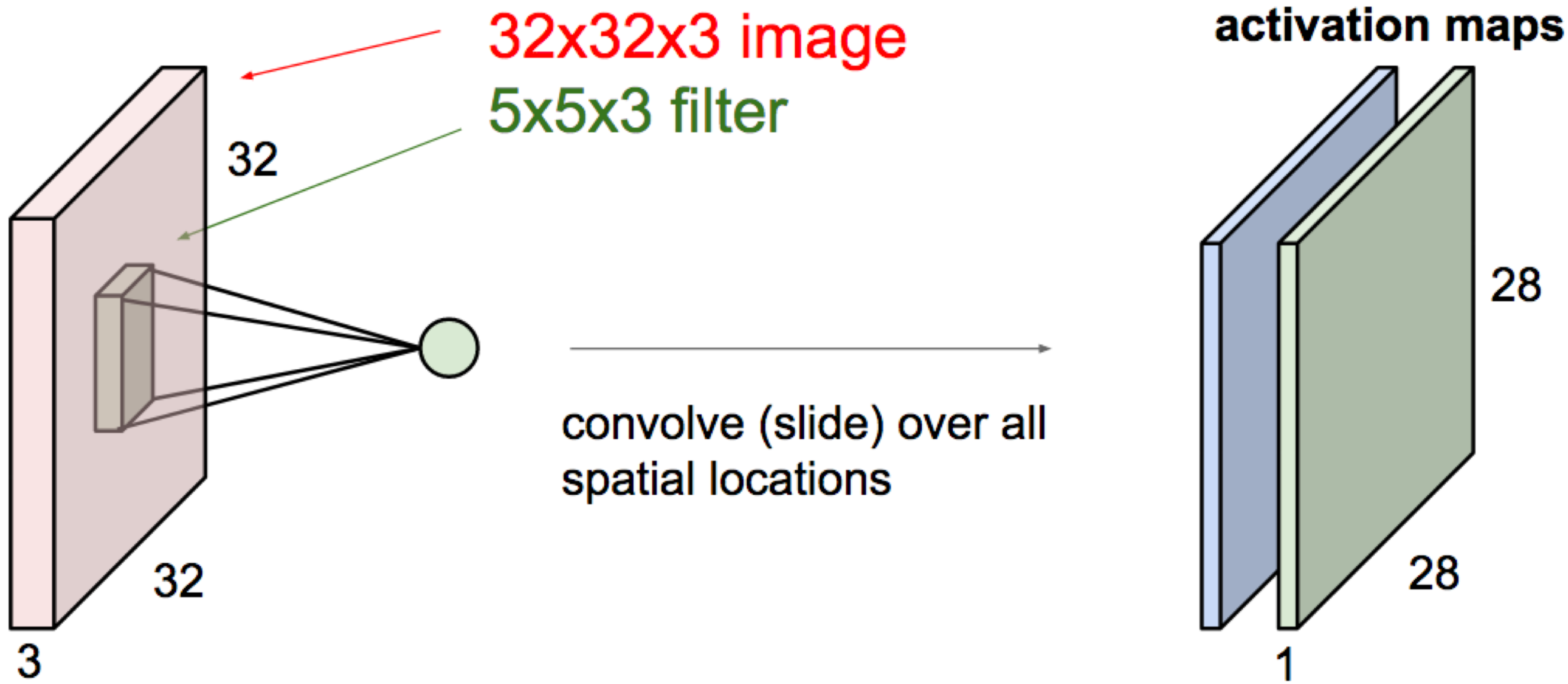
$$w^T x + b$$

Convolution Layer



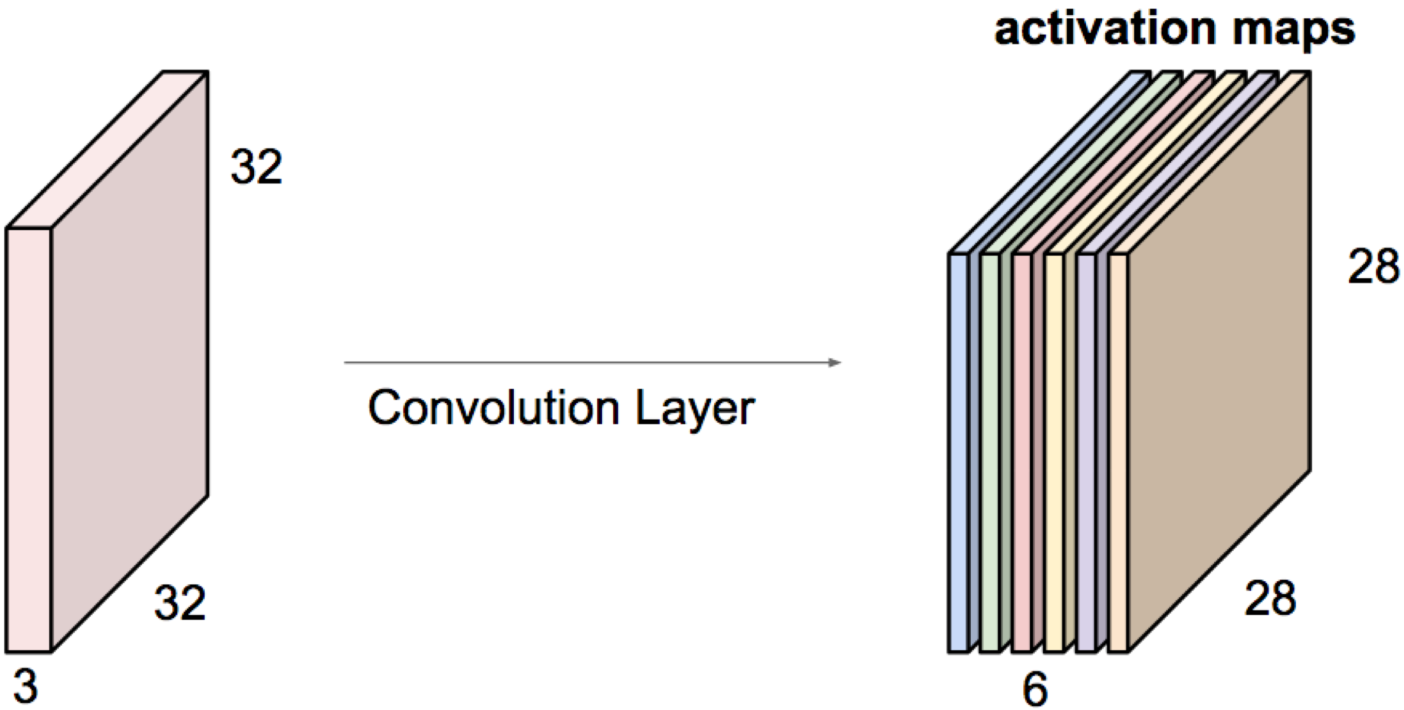
Convolution Layer

consider a second, green filter



Convolution as a general layer

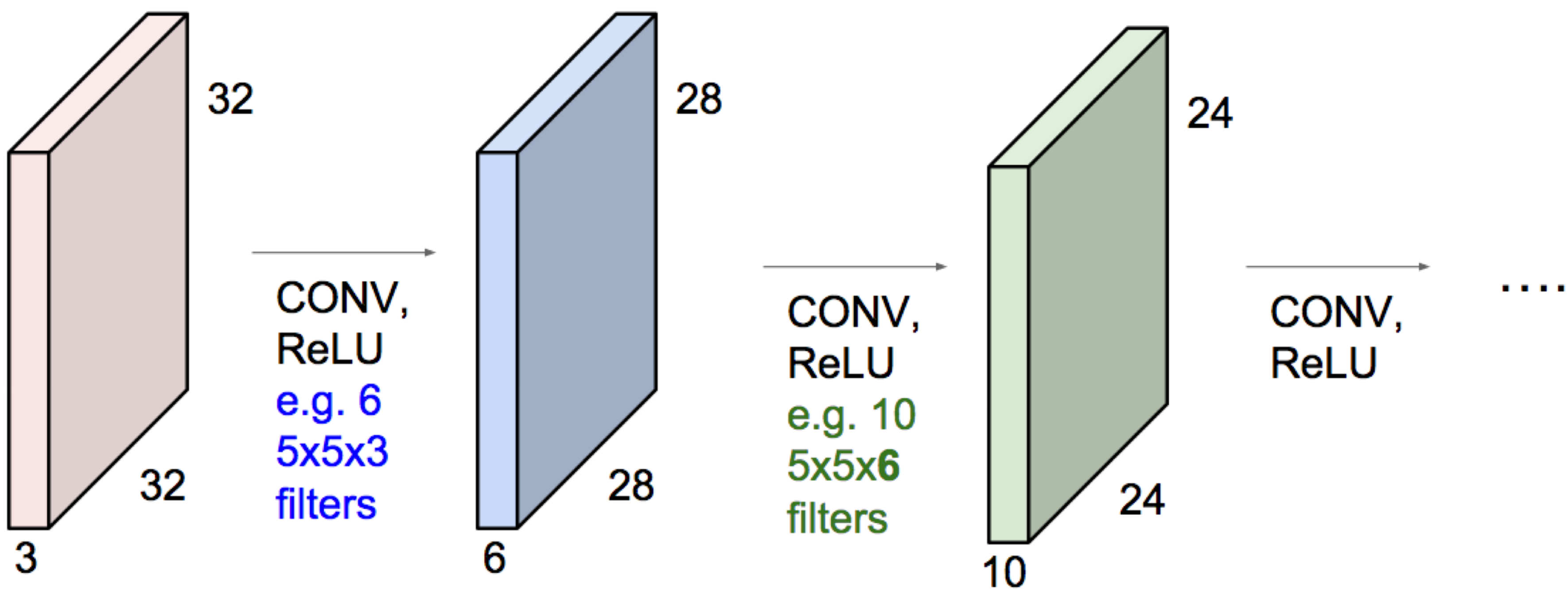
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



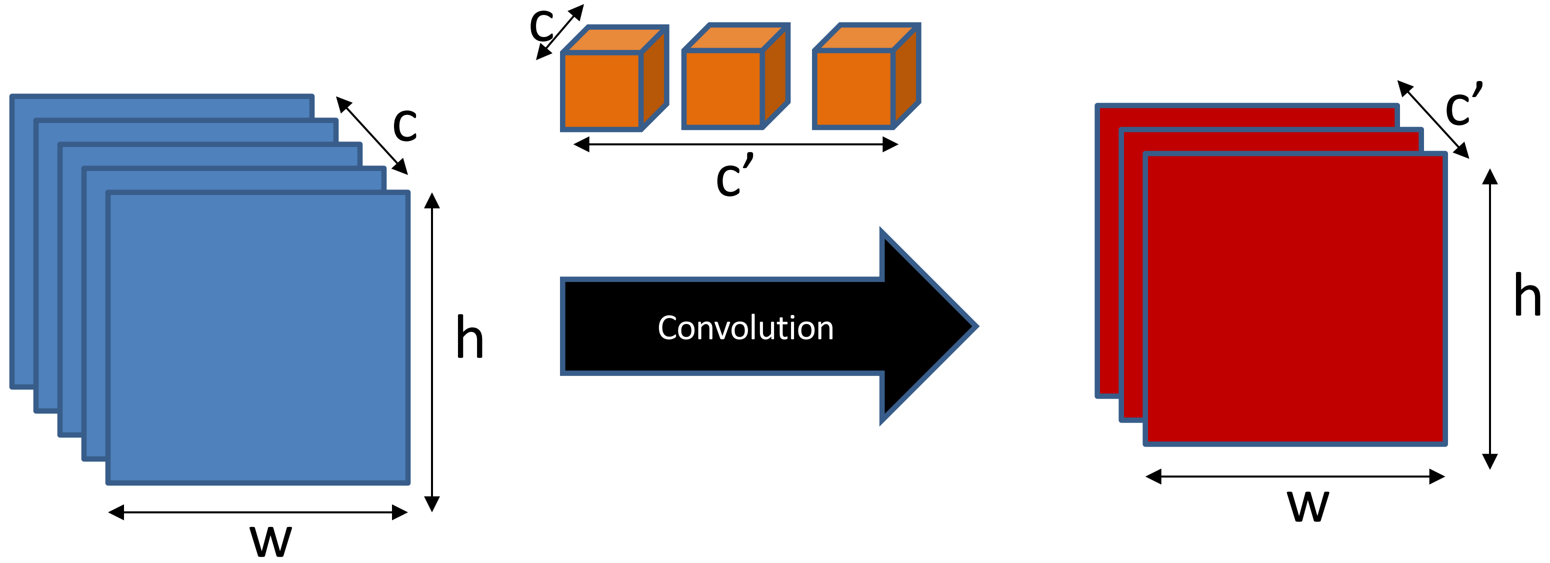
We stack these up to get a "new image" of size 28x28x6!

Convolutional Neural Networks

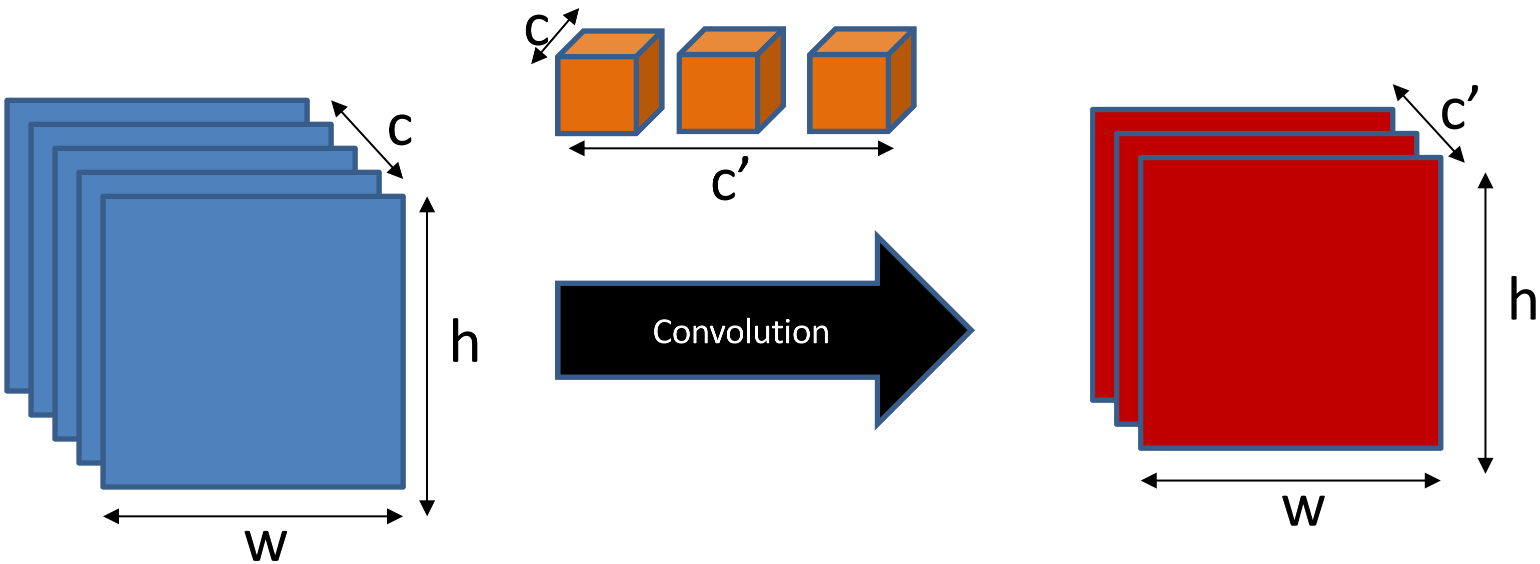
- Convolution layers interspersed with activation functions.



Convolution as a primitive



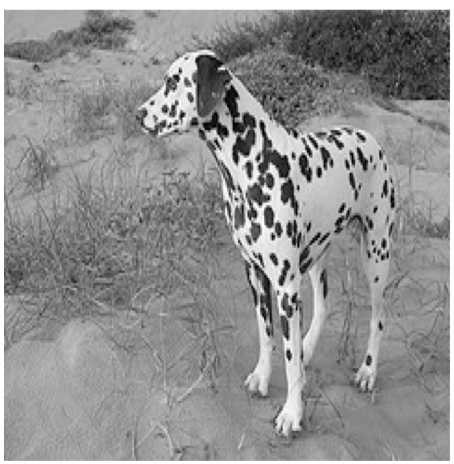
Convolution as a primitive

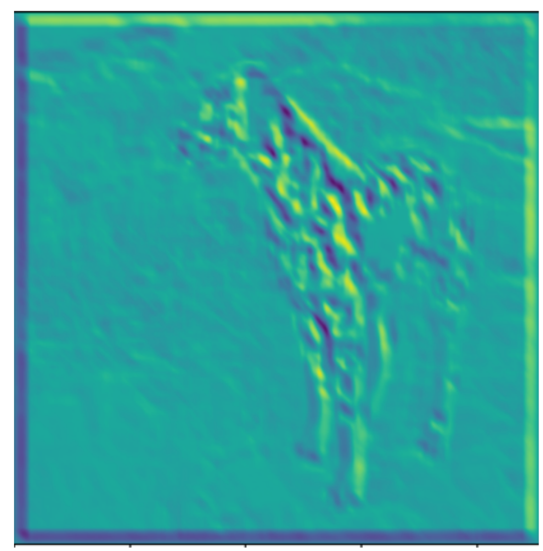


- How many parameters?
  - in_channels * Kw * Kh * out_channels
  - Example: 3x3x10 kernel, 10 output channels = 900 parameters!

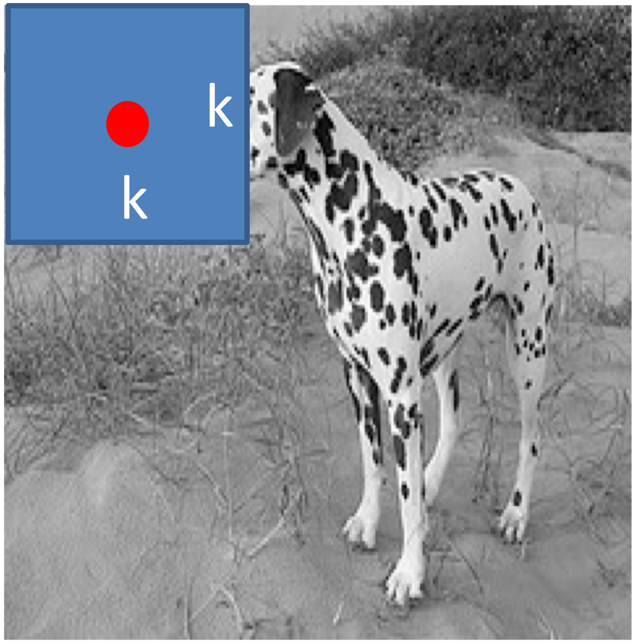
Convolution as a feature detector

- score at (x,y) = dot product (filter, image patch at (x,y))
- Response represents similarity between filter and image patch



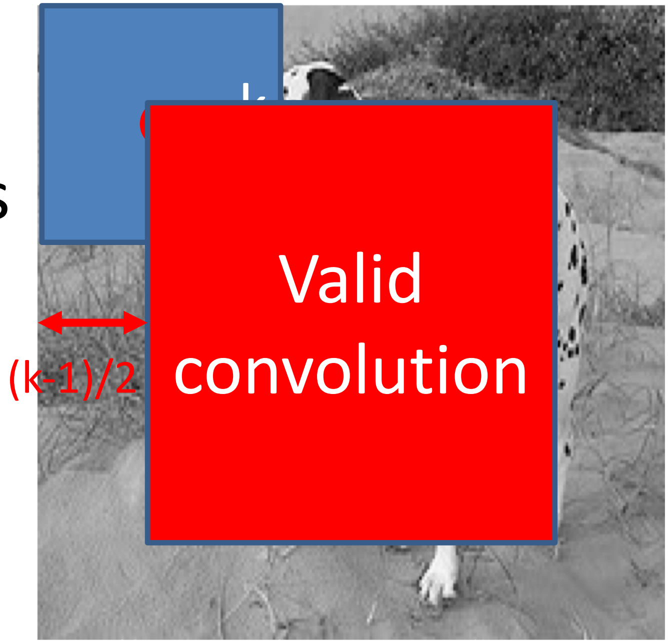


Kernel sizes and padding



Kernel sizes and padding

- Valid convolution decreases size by (k-1)/2 on each side
  - Pad by (k-1)/2, or
  - Allow spatial dimensions to shrink.



torch.nn.Conv2d

- torch.nn.Conv2d(

```
in_channels,   # channels in input feature map
out_channels, # filters to learn (== channels in the output)
kernel_size,     # size of each filter kernel
stride=1,          # move this many pixels when sliding filter
padding=0,       # pad the input by this much (can be tuple)
dilation=1,
groups=1,
bias=True         # add a bias after convolution?
)
```

Convolutional Layers

- Feature maps (“hidden layers”, “activations”, etc.) are no longer column vectors but 3D blobs:
  - Input # 256x256x3
  - Conv2d(in: 3, out:10) # Blob size: 255x255x10
  - Conv2d(in: 10, out:20) # Blob size: 255x255x20
  - …

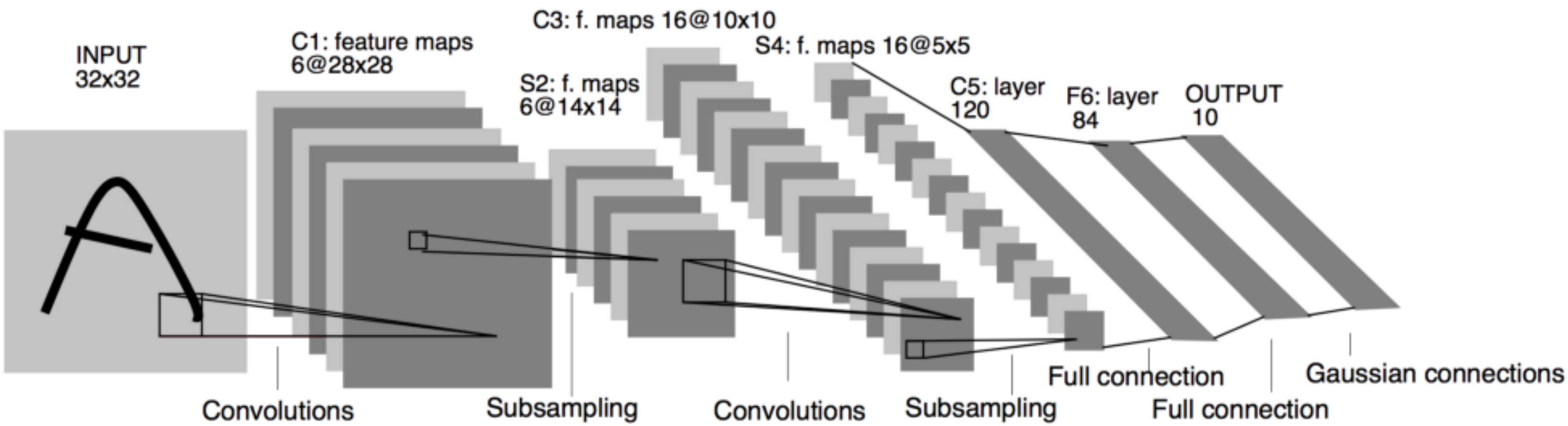
Convolutional Layers

- Feature maps ("hidden layers", "activations", etc.) are no longer column vectors but 3D blobs:
  - Input # 256x256x3
  - Conv2d(in: 3, out:10) # 255x255x10
  - Conv2d(in: 10, out:20) # 254x254x20
  - … this could get large quickly, and we ultimately need a vector that we can apply a linear classifier to.

Convolutional Networks

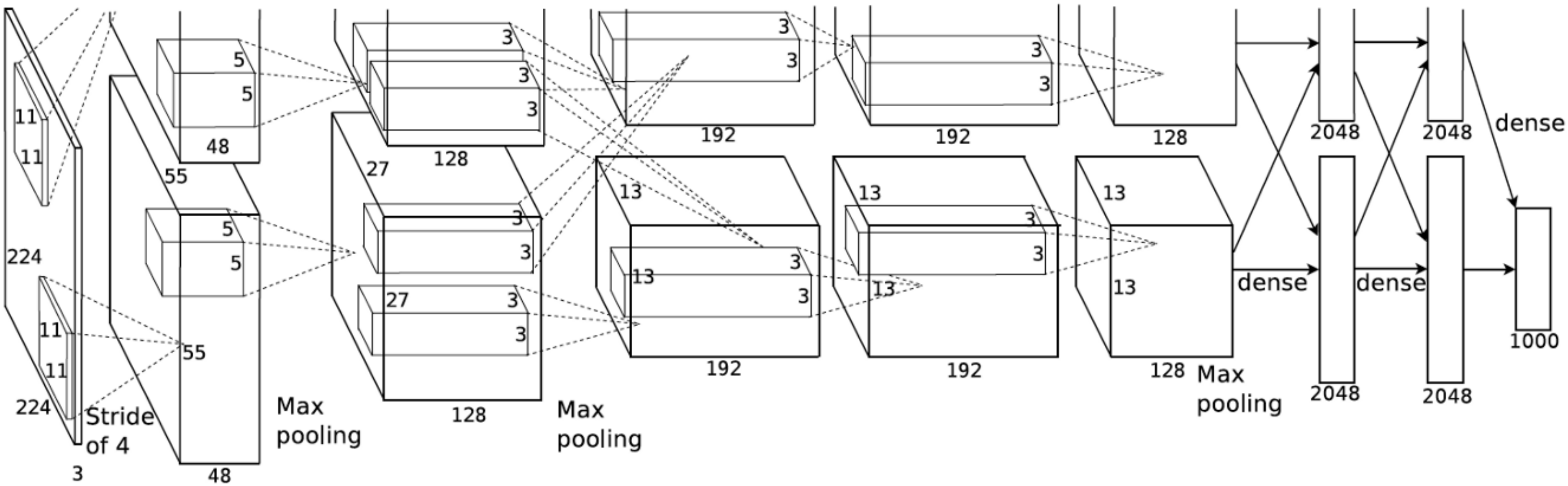
- Feature maps (“hidden layers”, “activations”, etc.) are no longer column vectors but 3D blobs:
  - Input # 256x256x3
  - Conv2d(in: 3, out:10) # 255x255x10
  - Subsample (2x2)
  - Conv2d(in: 10, out:20) # 127x127x20
  - …
  - Conv/subsample until 1x1xC
  - Or at some point, just unravel HxWxC into HWCx1 vector.
  - Then apply a linear classifier!

CNNs before they were cool: LeNet-5 [LeCun et al., 1998]



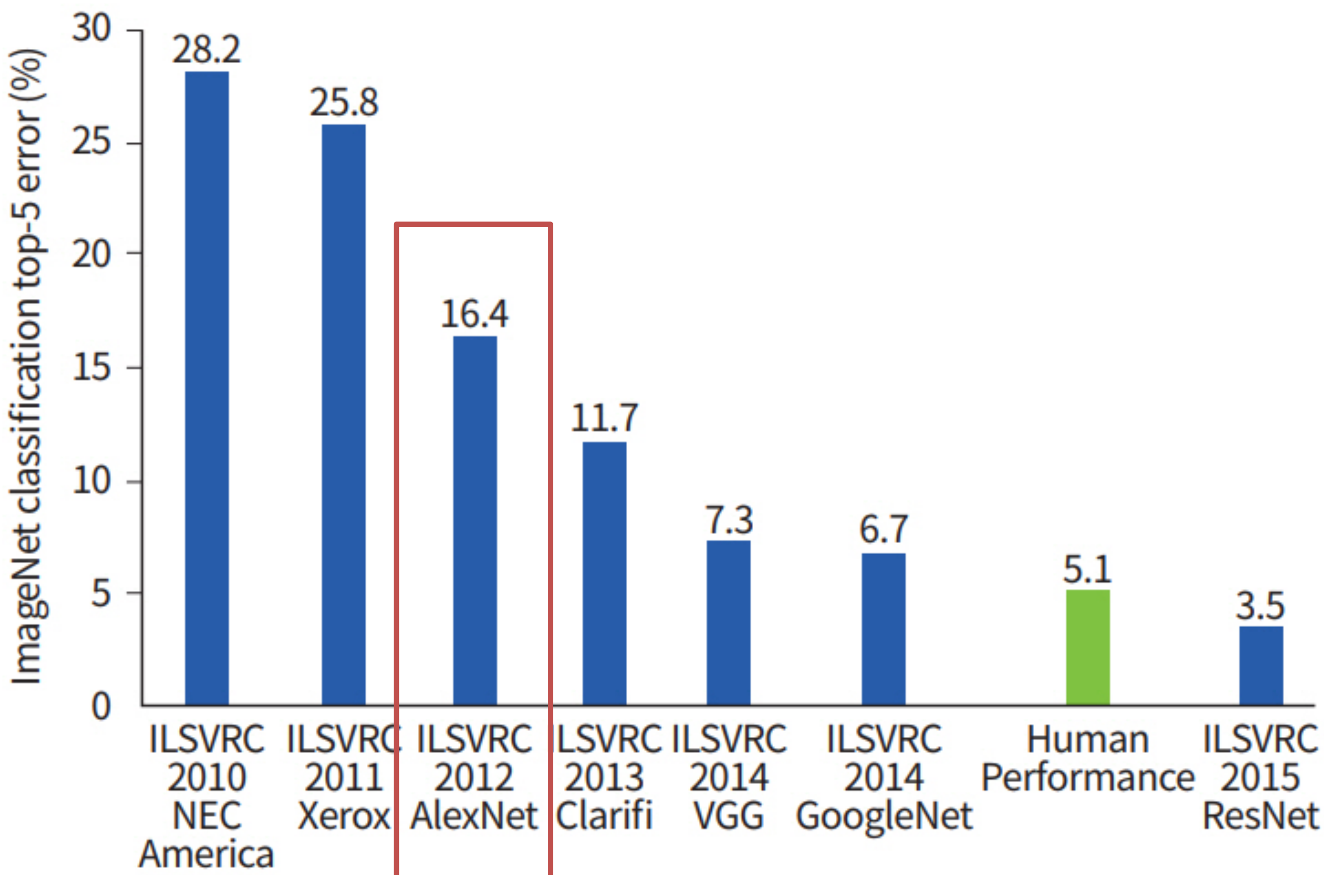
- Today's architectures still look a lot like this!

The CNN that made them cool: AlexNet [Krizhevsky et al. 2012]



The CNN that made them cool: AlexNet [Krizhevsky et al. 2012]

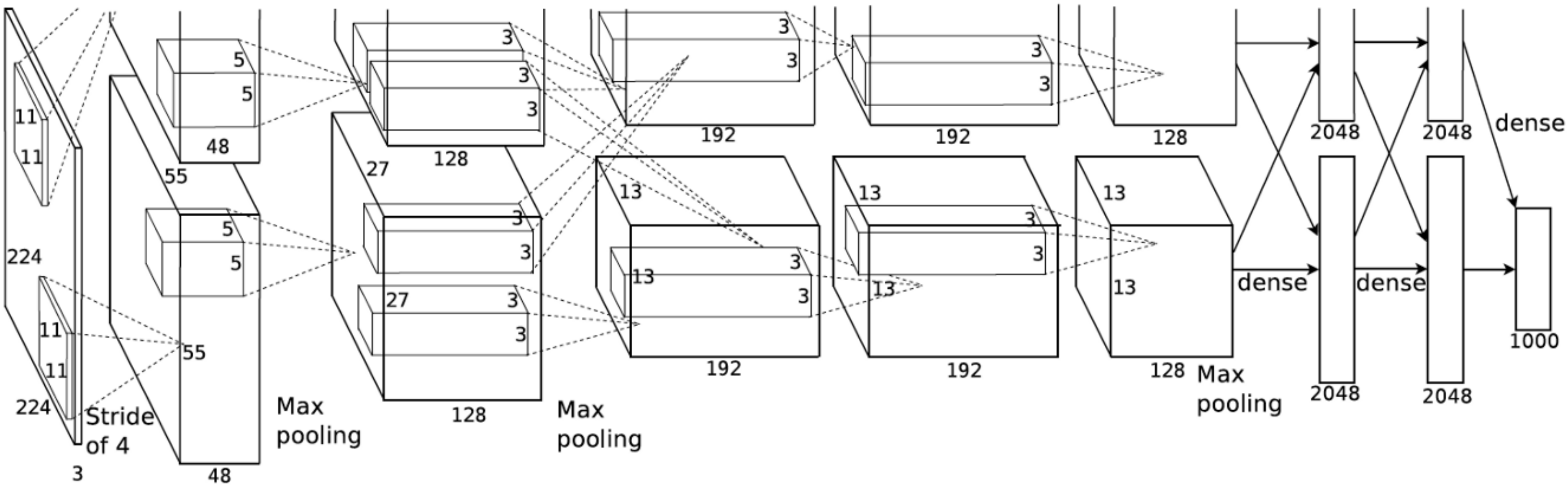
- What happened?



The CNN that made them cool: AlexNet [Krizhevsky et al. 2012]

- What changed?
  - Bigger training data: ImageNet has 14 million images and 20,000 categories.
    - (performance numbers are on a 1000-category subset)
  - GPU implementation of ConvNets
    - Train bigger, deeper networks for longer than before
  - ReLU
    - Not new in AlexNet, but a necessary design choice to avoid vanishing gradients in deep network
- Hence "deep learning":
  - a rebranding of formerly unfashionable neural networks

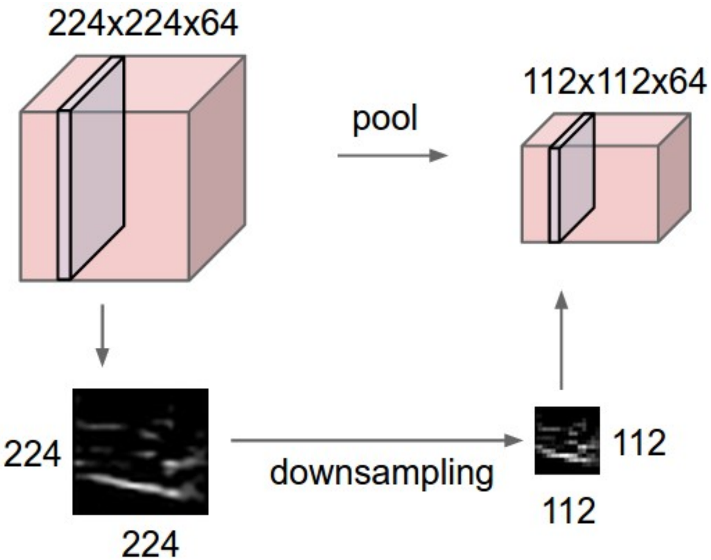
The CNN that made them cool: AlexNet [Krizhevsky et al. 2012]



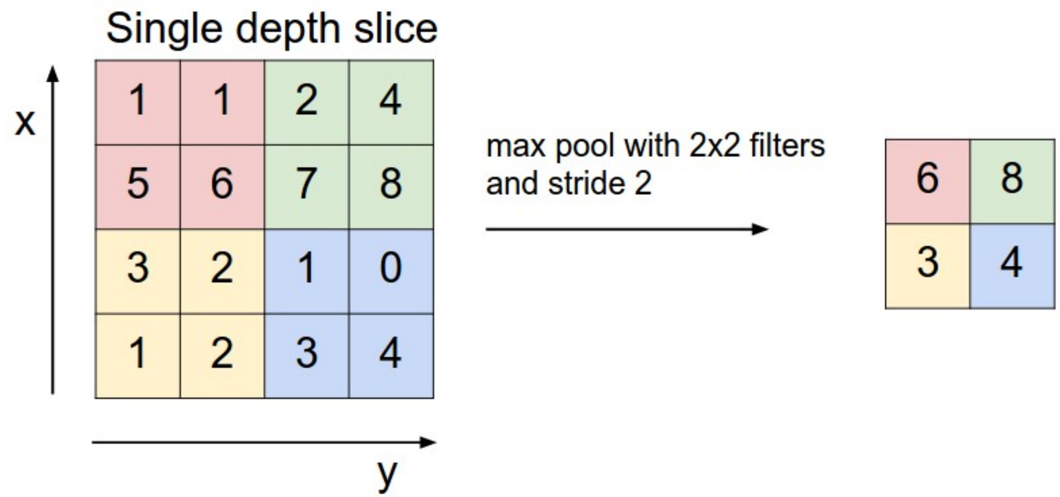
- What else is in this network?
  - ReLU after each layer (not pictured)
  - Dense = Fully connected = Linear layer = a matrix multiply
  - Max pooling

Downsampling, Subsampling, Pooling

Downsampling:



Max pooling:



- Reducing spatial dimensions:
  - Subsample (e.g. throw away every other pixel)
  - Average pooling
  - Max pooling (most commonly used)