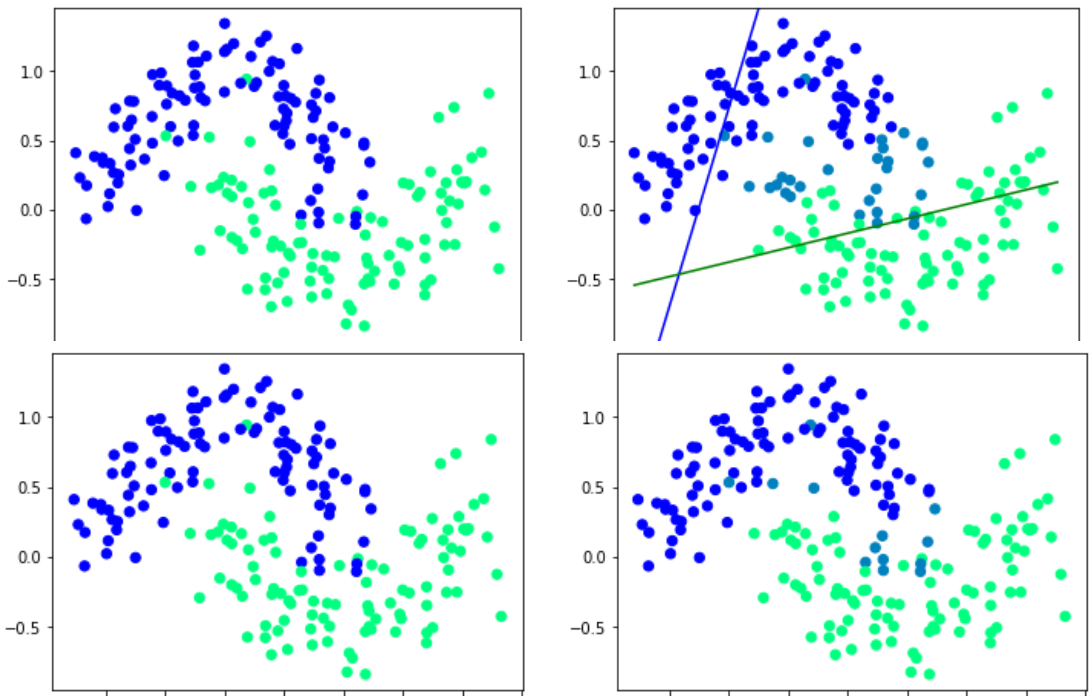
CSCI 497P/597P: Computer Vision

Neural Networks: Activation Functions
Gradient Descent in Neural Networks (Backpropagation)

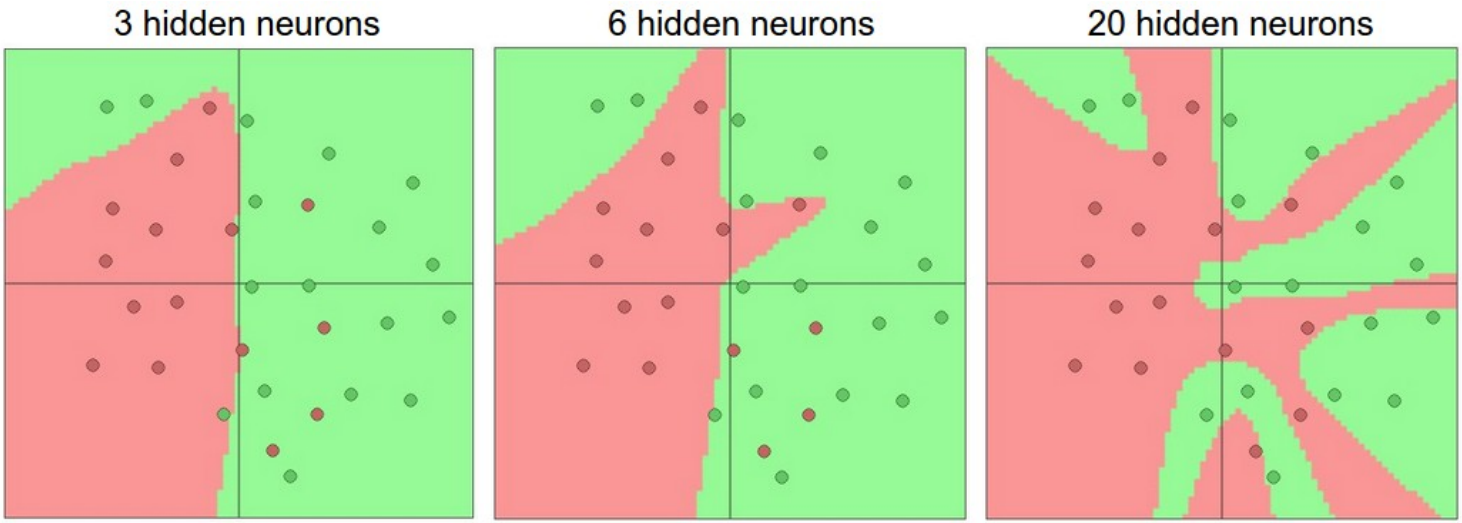


Readings

with a great deal more detail…

- http://cs231n.github.io/optimization-2/
- http://cs231n.github.io/neural-networks-1/
- http://cs231n.github.io/neural-networks-2/
- http://cs231n.github.io/neural-networks-3/

Neural Networks: Nonlinear Classifiers built from Linear Classifiers



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

Neural networks: without the brain stuff

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$
or 3-layer Neural Network

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

???

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

Neural Networks

Neural Network

Linear classifiers

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

Neural networks: without the brain stuff

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$
or 3-layer Neural Network

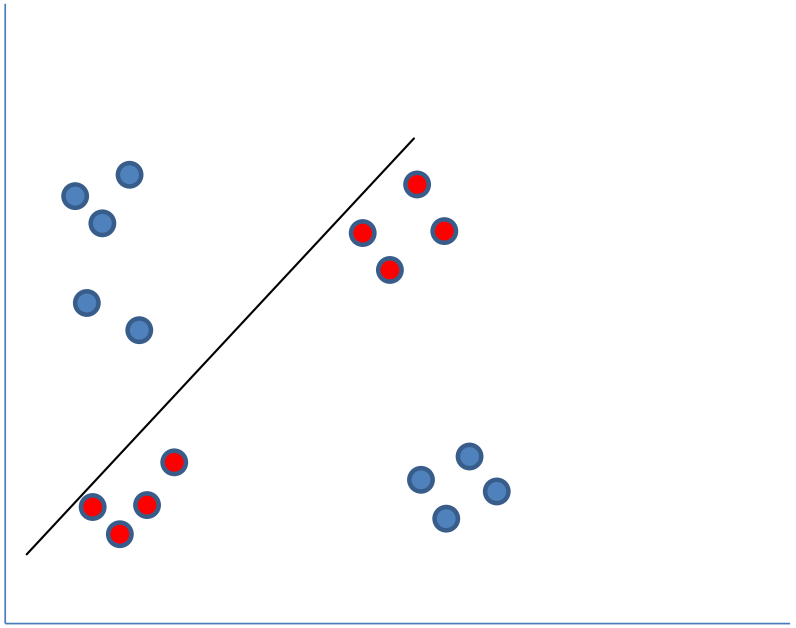
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

???

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

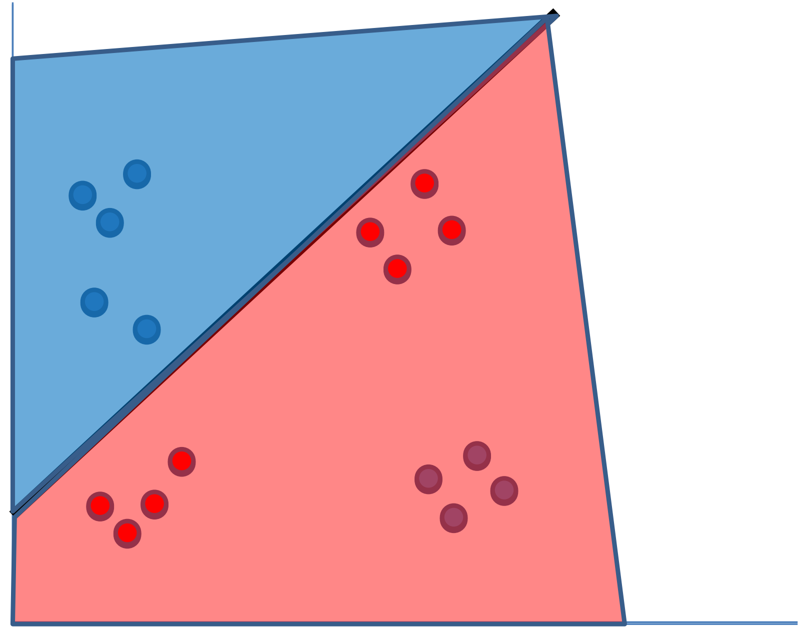
Activation Functions

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



Activation Functions

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

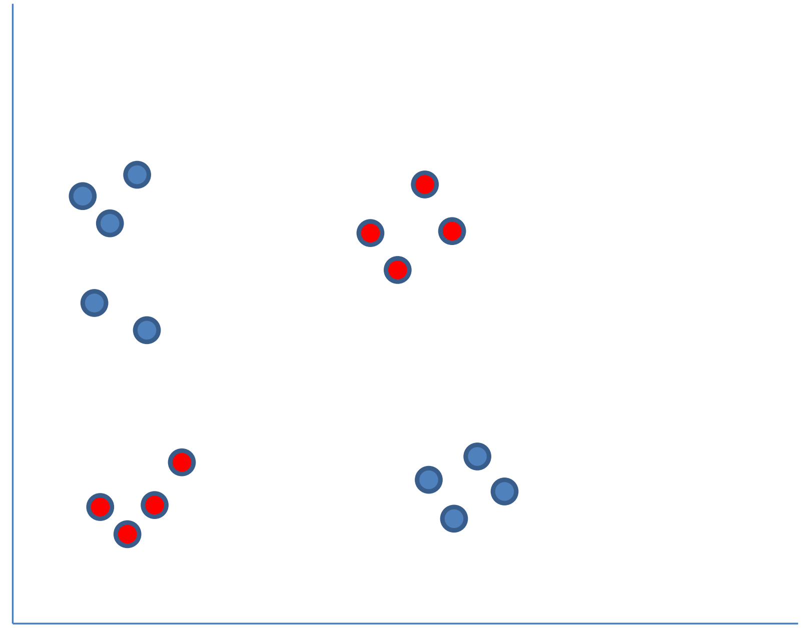


A linear classifier can only do so well…

Activation Functions

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

$$f(x, W_1, W_2) = W_1(W_2x)$$



Let's try stacking two linear classifiers together

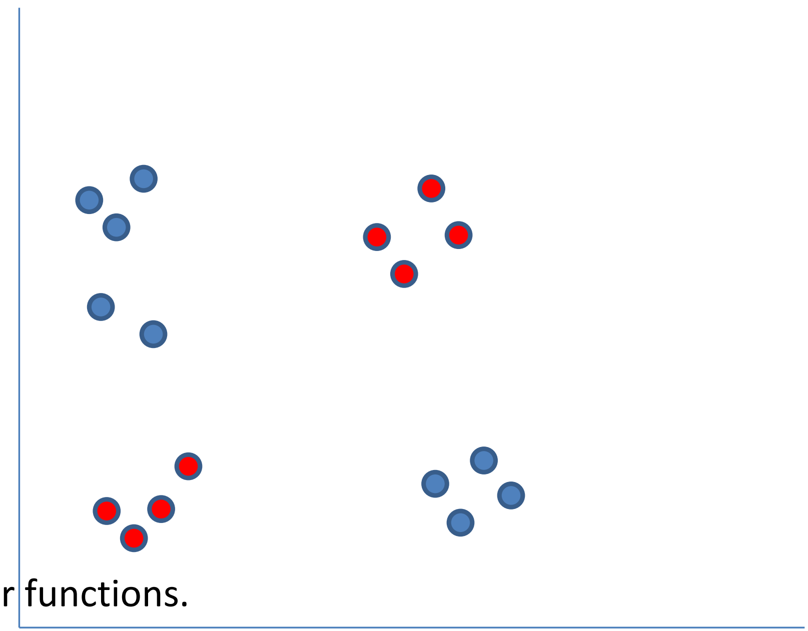
Activation Functions

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

$$f(x, W_1, W_2) = W_1(W_2 x)$$

$$W \leftarrow W_1 W_2$$

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

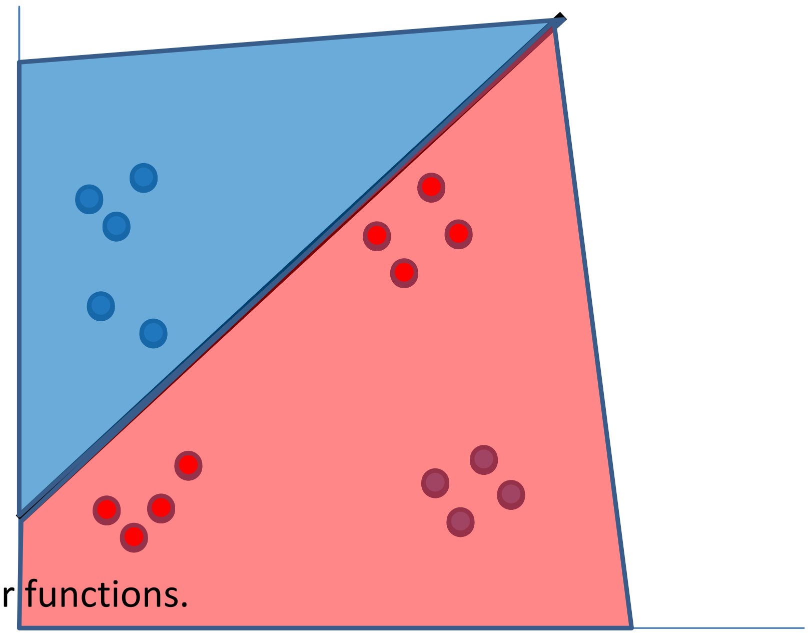


Uh oh – linear functions compose to linear functions.

Activation Functions

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

$$f(x, W_1, W_2) = W_1(W_2 x)$$



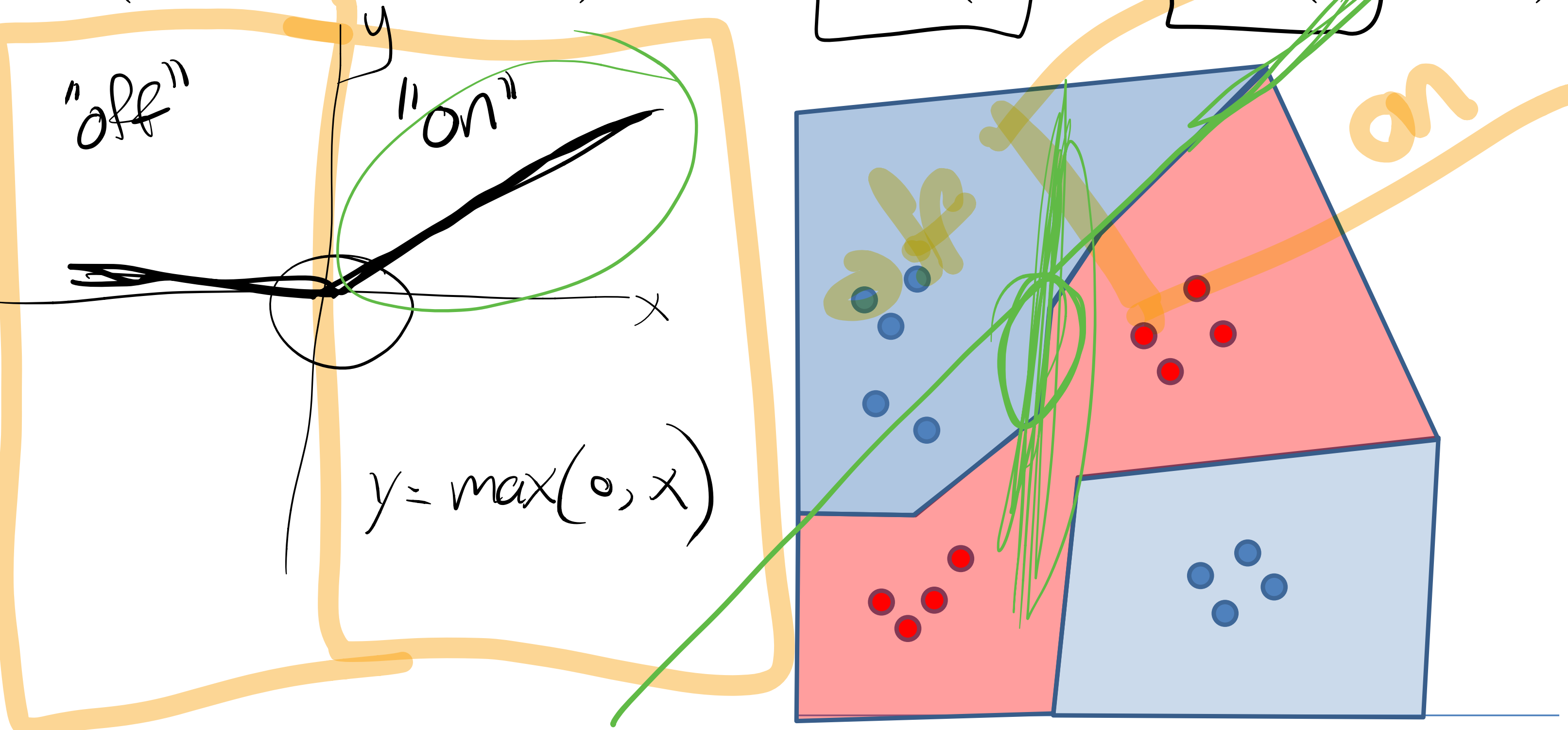
$$W \leftarrow W_1 W_2$$

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

Uh oh – linear functions compose to linear functions.

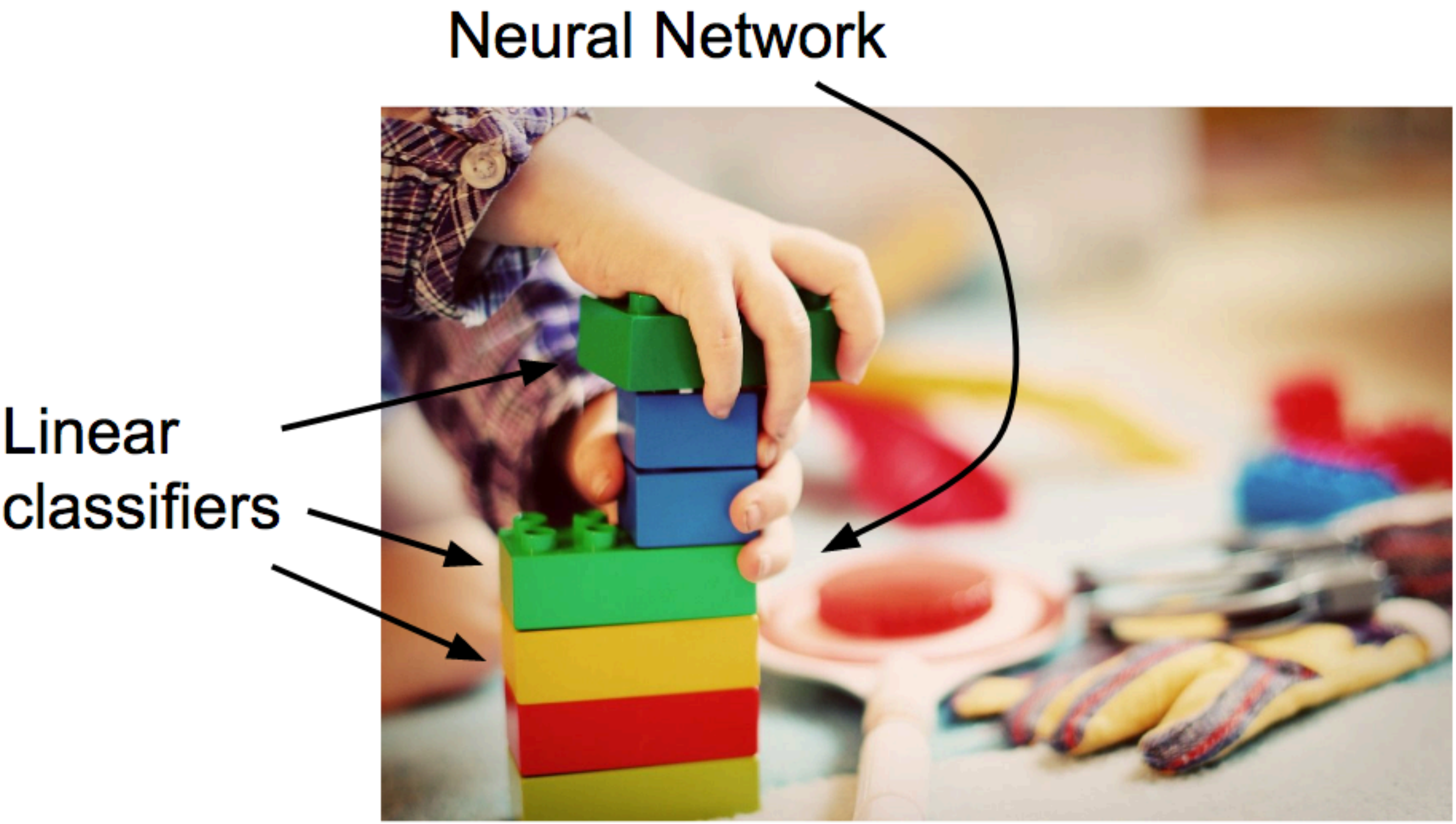
Activation Functions

$$f(x, W_1, W_2, W_3) = W_3 \max(0, W_2 \max(0, W_1 x))$$



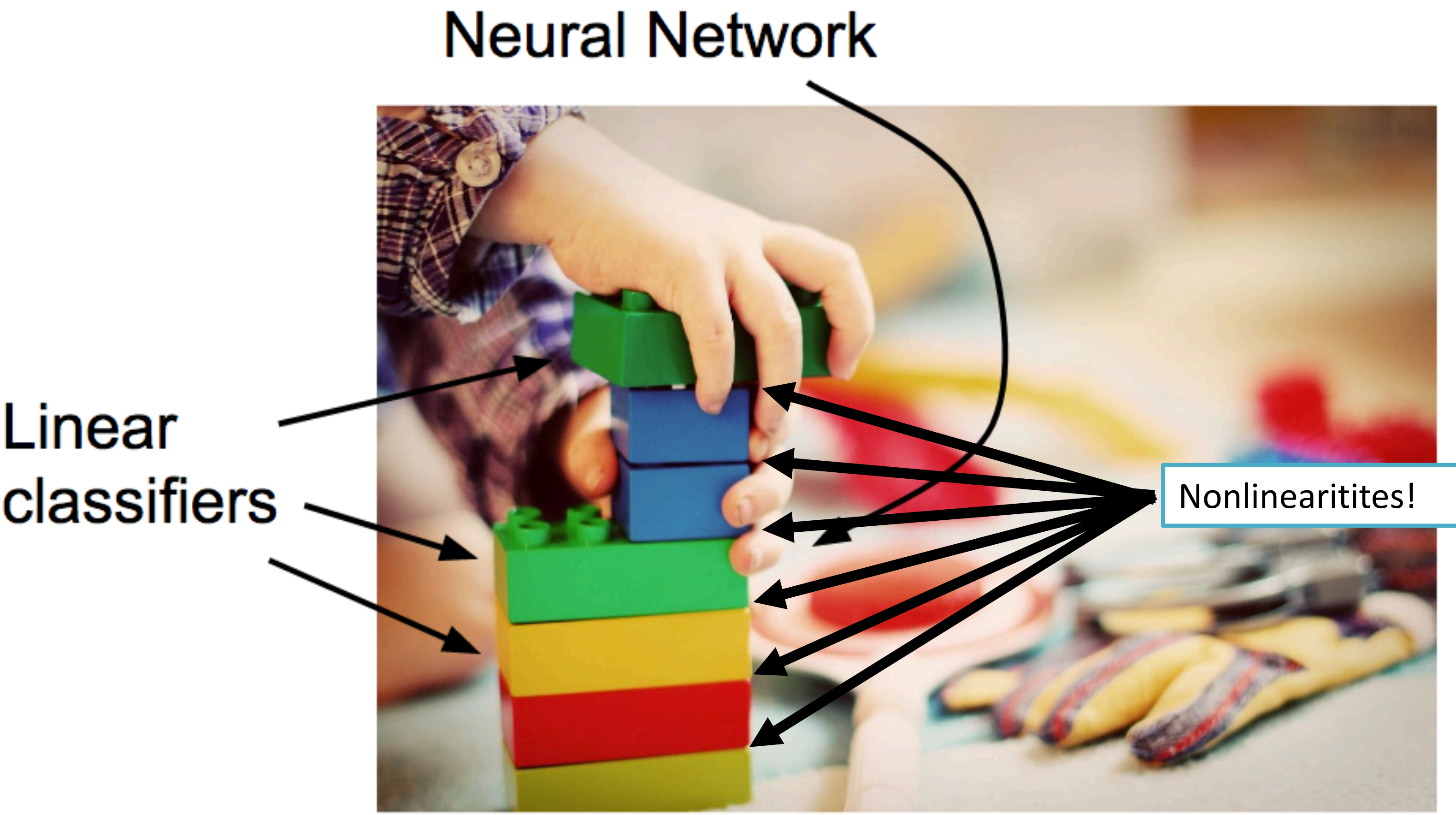
Nonlinearities prevent the composed linear functions from collapsing into a single one.

Neural Networks



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

Neural Networks



Neural Networks: Nonlinear Classifiers built from Linear Classifiers

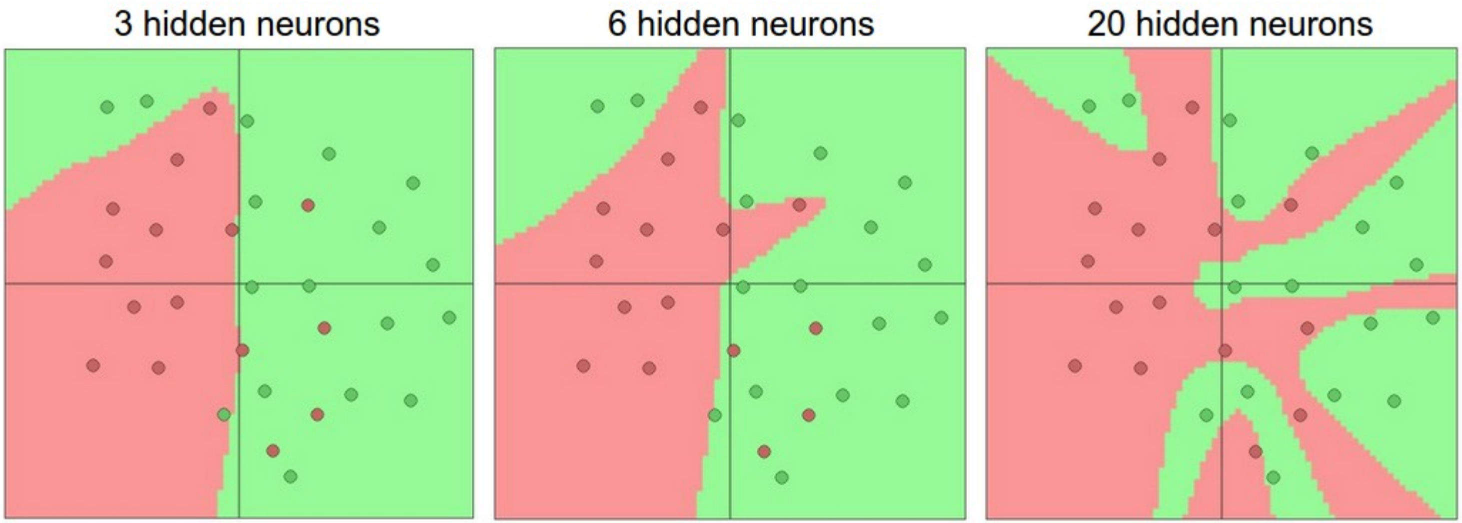
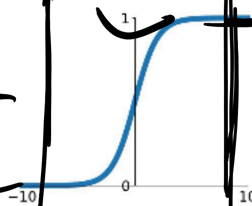


Figure: Fei-Fei Li, Justin Johnson, & Serena Yeung

Activation Functions

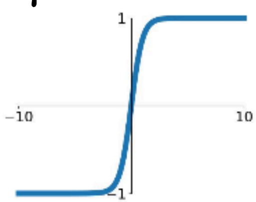
Sigmoid

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



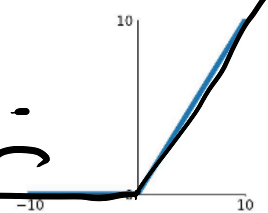
tanh

$\tanh(x)$



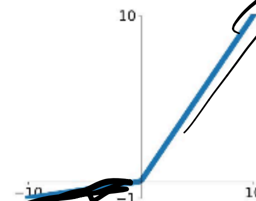
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

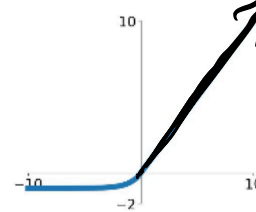


Maxout

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

ELU

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



$\sigma(w^T x + b)$

0 grad

vanishing gradients

rectified linear unit = max(0, x)

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

(Stochastic) Gradient Descent: In Practice

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

→ W_1, W_2

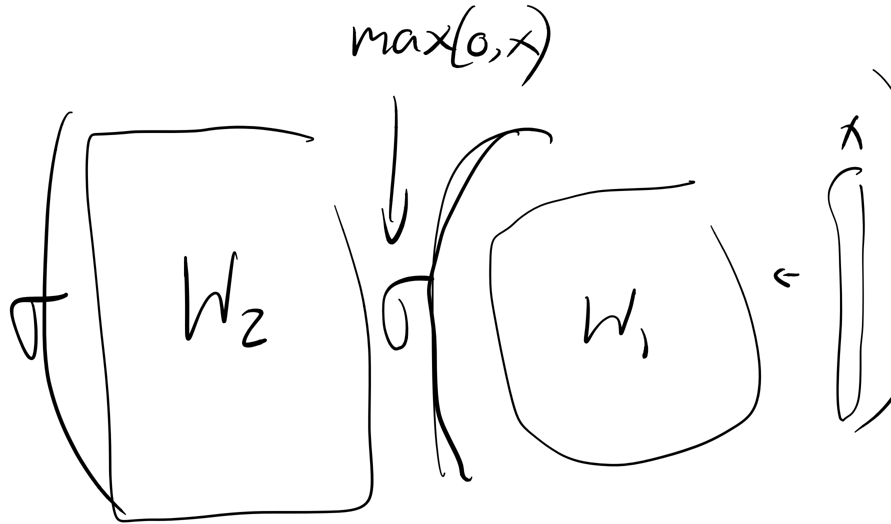
$W_1 \leftarrow -ss \cdot W_1.grad.$

W_2 - - - -

$$l(x) = f(W_2, g(W_1, x))$$

$$\frac{\partial l}{W_1} = \frac{\partial l}{\partial f} \cdot \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial W_1}$$

↑ $\frac{\partial f}{\partial W_2}$



Backpropagation in Pytorch

- Your deep learning framework knows how to differentiate anything you might want to do.
- Exmple, in pytorch:
  - Your classifier inherits from torch.nn.Module
  - You implement its forward() method
  - Torch generates a backward() method for you!
  - Training looks like this (pseudocode)

```
output = classifier(data) # uses W, b
loss = loss_function(output, true_labels)
loss.backward() # (backprop magic here!)
dW = w.grad
db = b.grad
W -= step_size * dW
b -= step_size * db
```

Backpropagation in Pytorch

- Example, in pytorch (pseudocode):

```
output = classifier(data, W, b) # uses W, b
loss = loss_function(output, true_labels)
loss.backward() # (backprop magic here!)
dW = w.grad
db = b.grad
W -= step_size * dW
b -= step_size * db
```

- In practice, an Optimizer performs the updates instead:

```
optimizer = torch.optim.SGD(net.parameters(),
                lr=0.0001)
output = classifier(data)
loss = loss_function(output, true_labels)
loss.backward()
optimizer.step()
```

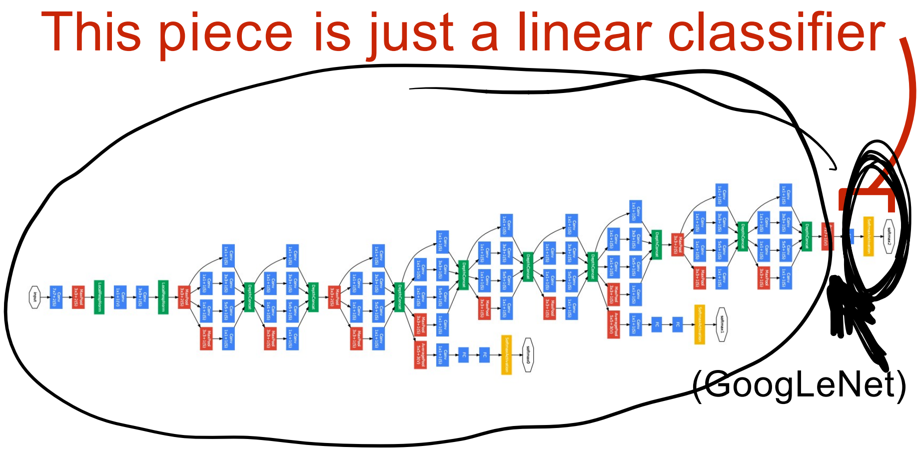
Demo

- A hand-rolled linear classifier in pytorch.
- Takeaways:
  - compute loss = my_loss_fn(X, y, W, …)
  - call loss.backward()
  - W.grad now contains the gradient of the loss with respect to W!

Two important pieces

- The feature extractor (ϕ)
- The classifier (h)

The last layer of (most) neural networks are linear classifiers



→ Ans

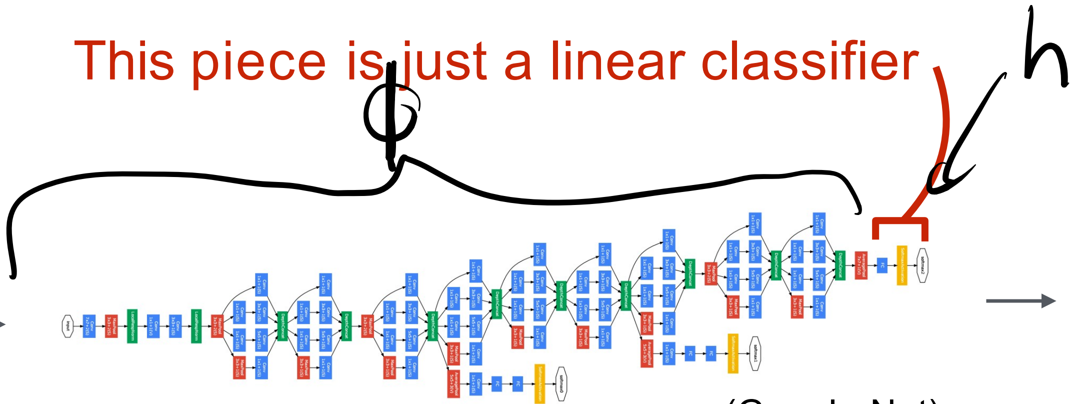
Input Pixels

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) neural networks are linear classifiers

This piece is just a linear classifier



→ Ans

(GoogLeNet)

Input Pixels

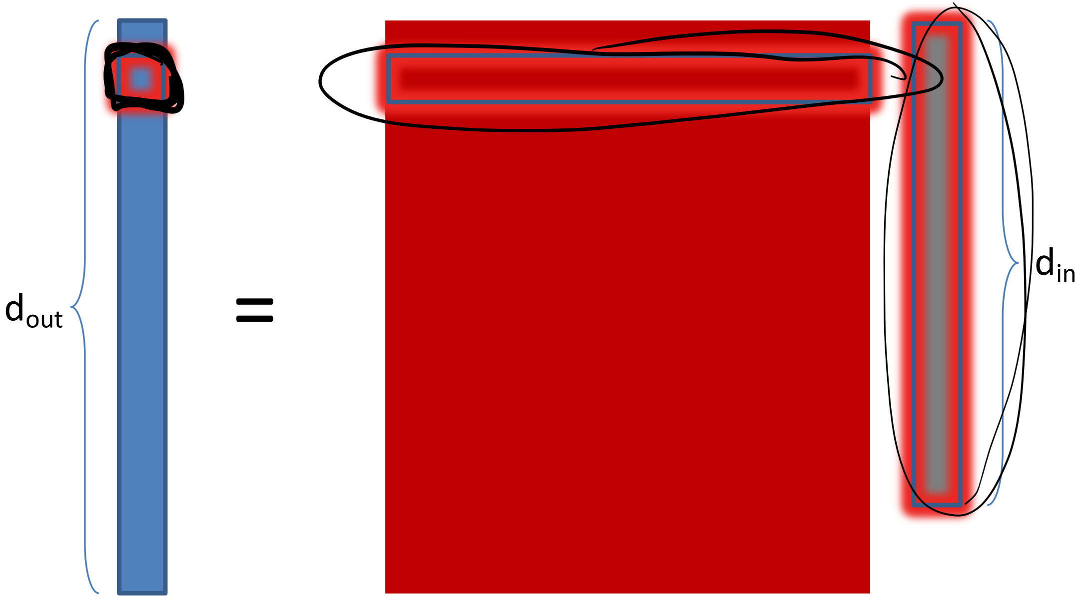
Perform everything with a big neural network, trained end-to-end

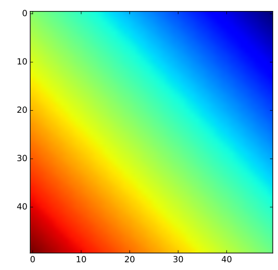
The network is the feature extractor *and* the classifier.

h swallowed ϕ!

A Linear Classifier

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

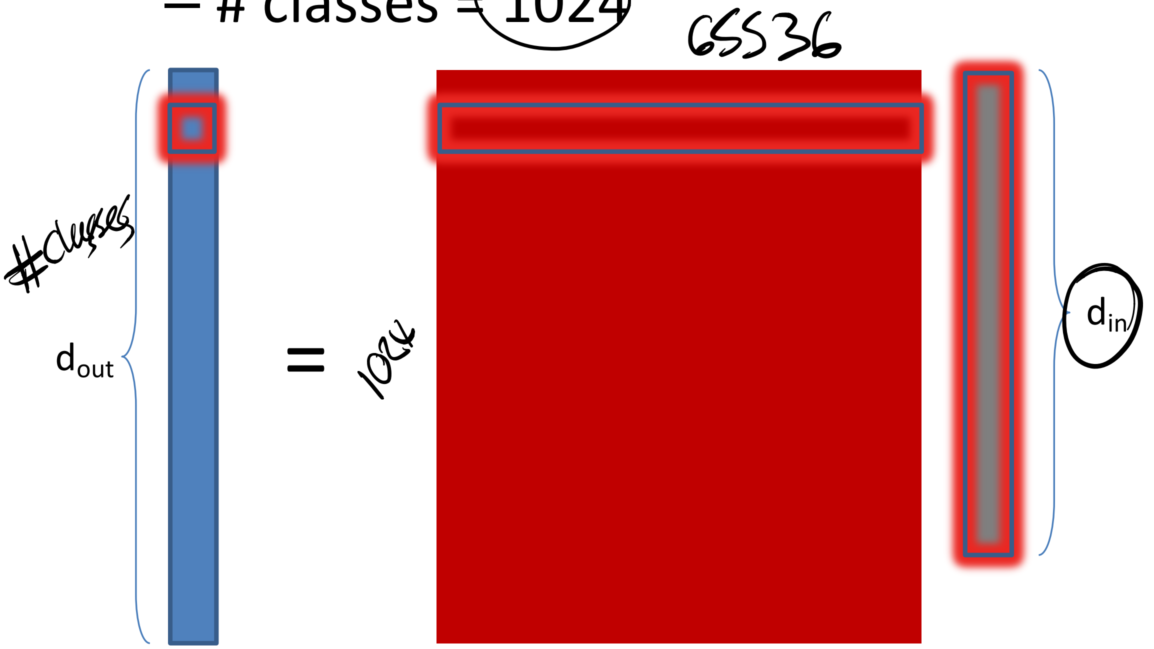


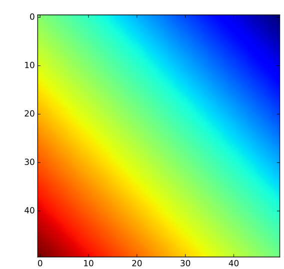


The case when $d_{in} = 2$. A single row in y plotted for every possible value of x

Linear Classifier: Parameter Count

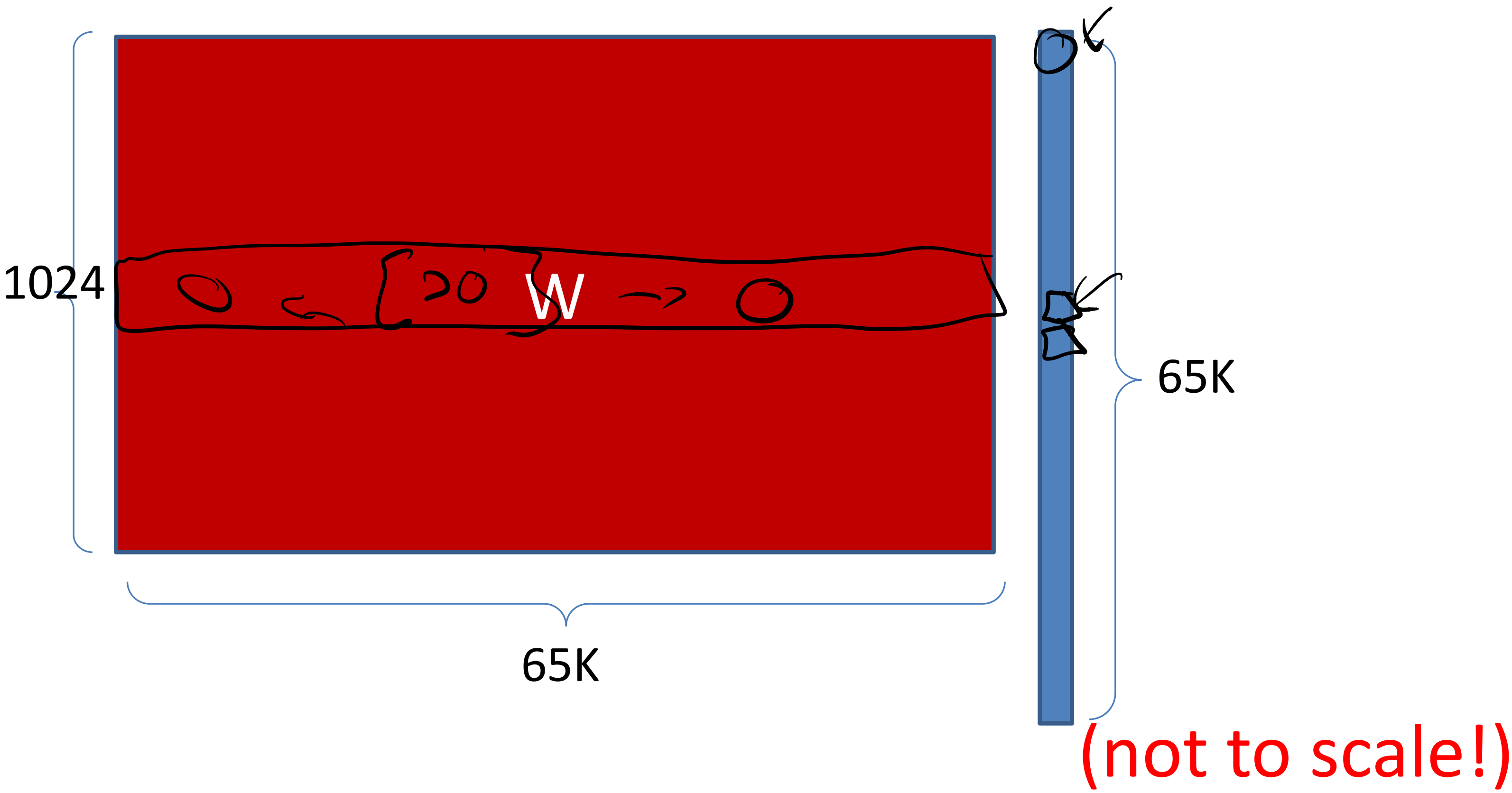
- How many parameters does a linear function have? Suppose:
  - # pixels = 256*256 = 65536
  - # classes = 1024





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

The linear function for images

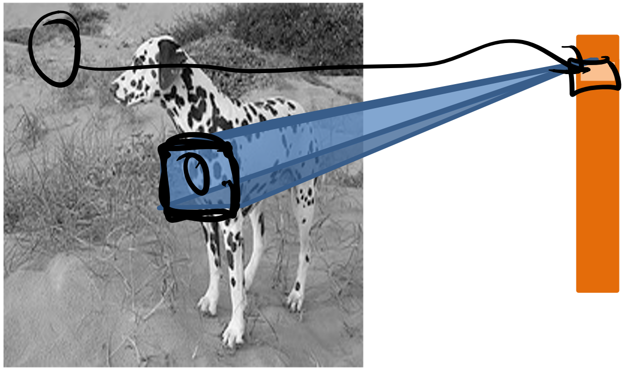


Linear Classifier: Parameter Count

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

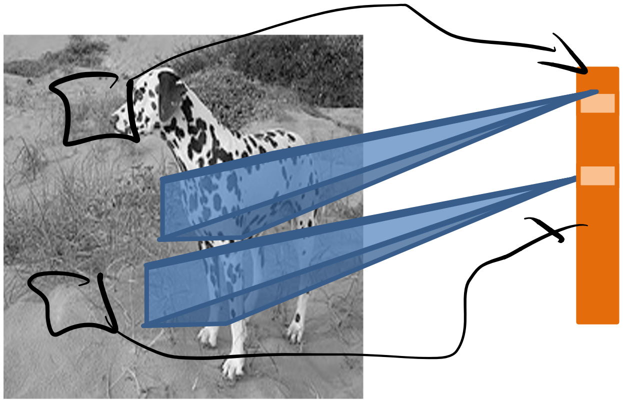
Idea 1: local connectivity

- Pixels only related to nearby pixels



Idea 2: Translation invariance

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



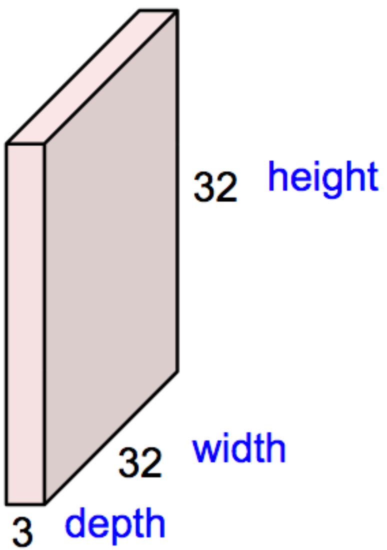
Linear function + translation invariance = *convolution*

- Local connectivity determines kernel size

5.4	0.1	3.6
1.8	2.3	4.5
1.1	3.4	7.2

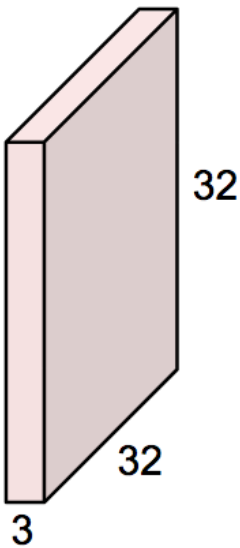
Convolution Layer

32x32x3 image -> preserve spatial structure

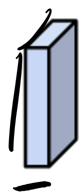


Convolution Layer

32x32x3 image



5x5x3 filter

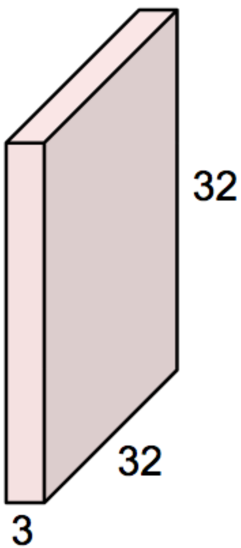


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

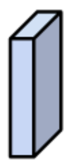
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image



5x5x3 filter

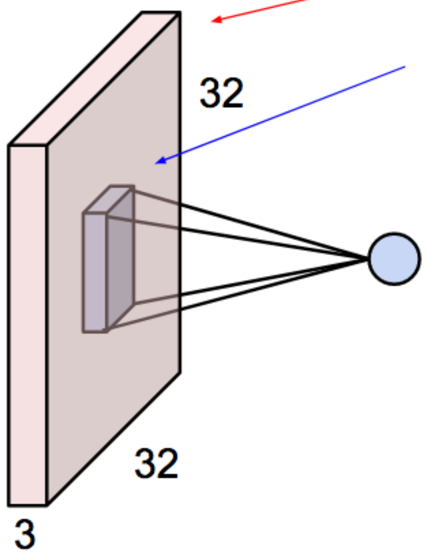


Convolve the filter with the image i.e. "slide over the image spatially, computing dot products"

Convolution Layer

32x32x3 image

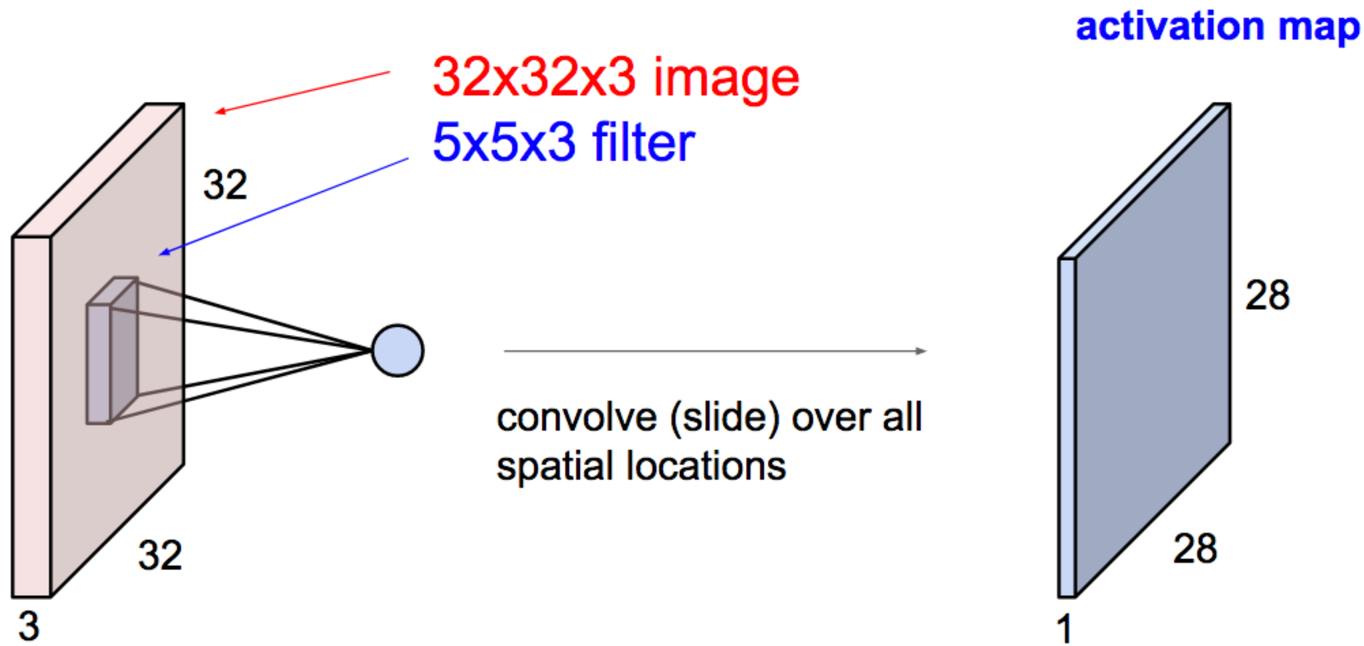
5x5x3 filter w



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)

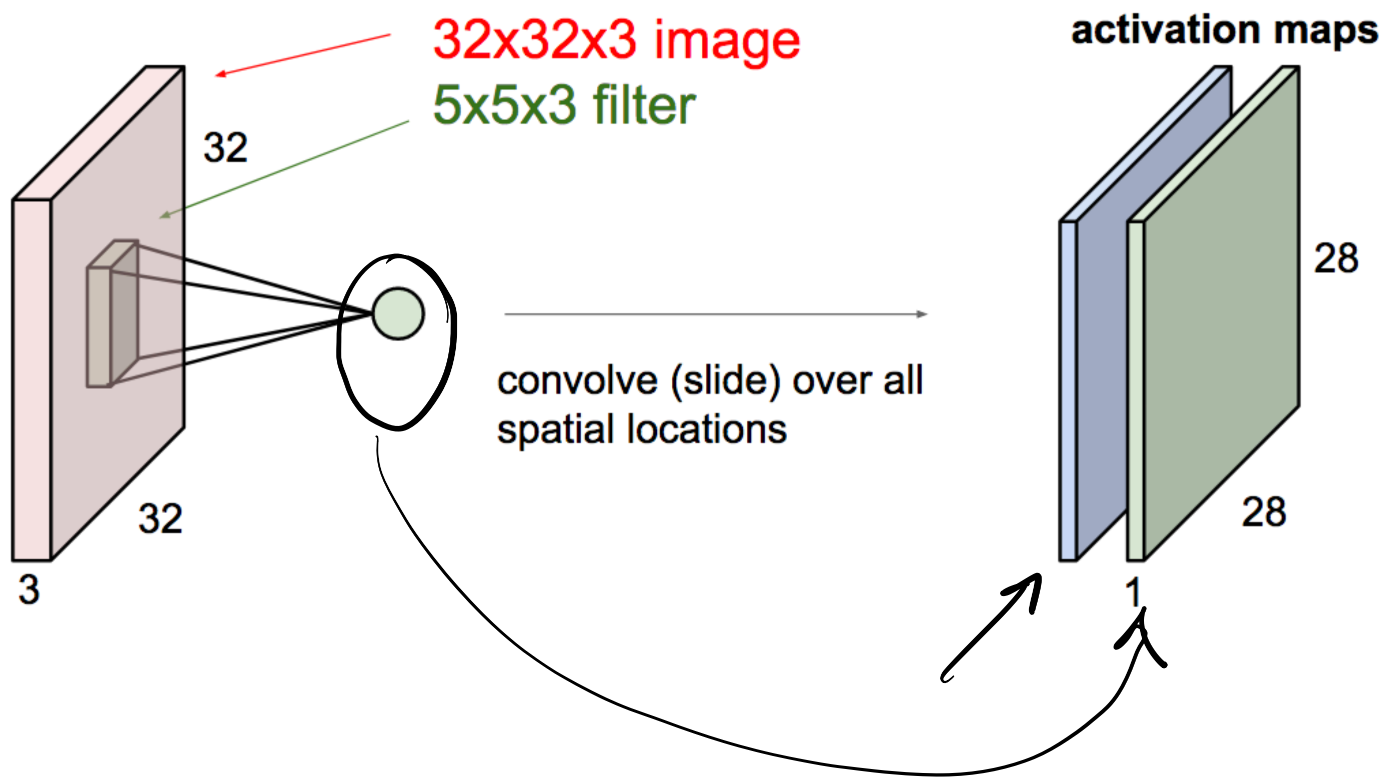
$$w^T x + b$$

Convolution Layer



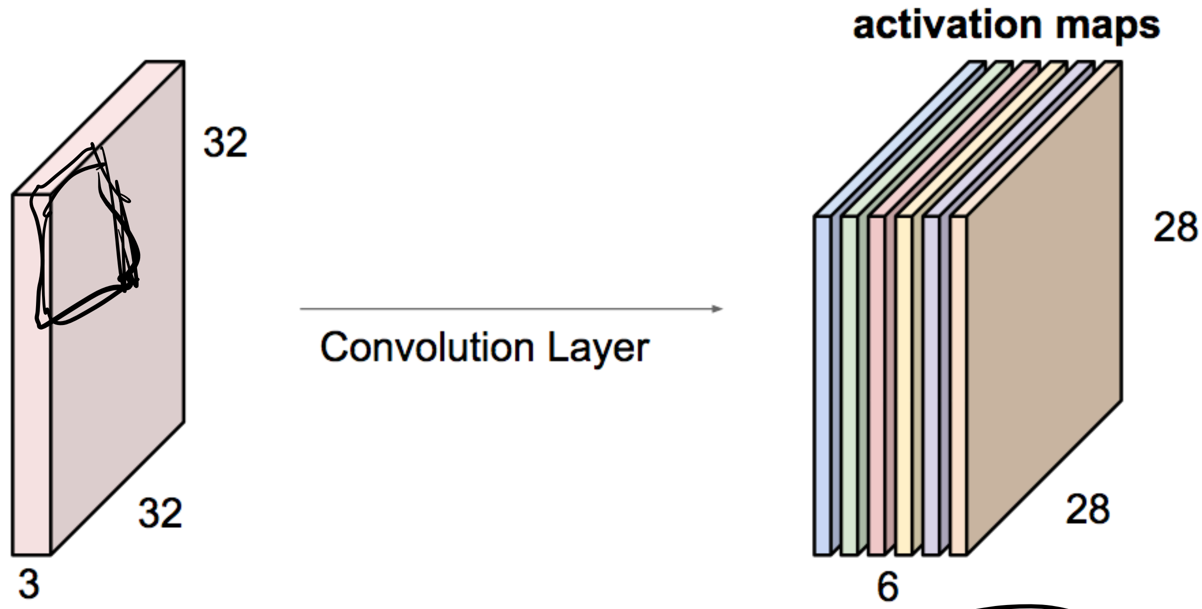
Convolution Layer

consider a second, green filter



Convolution as a general layer

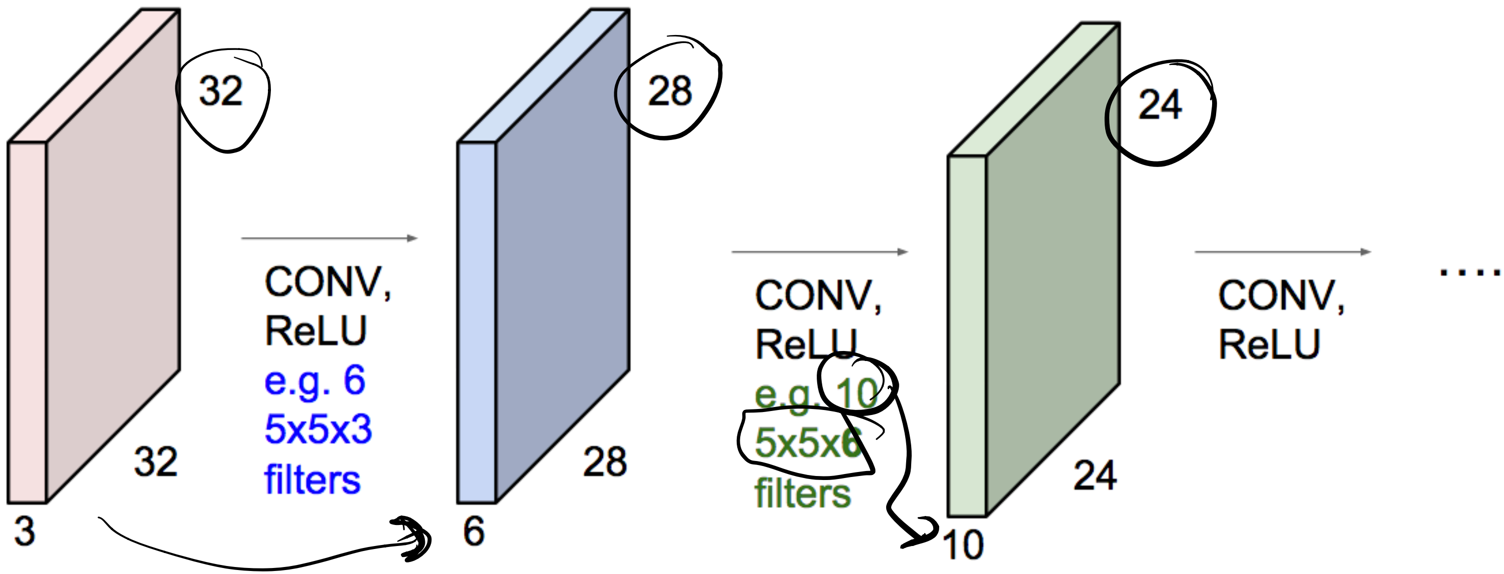
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



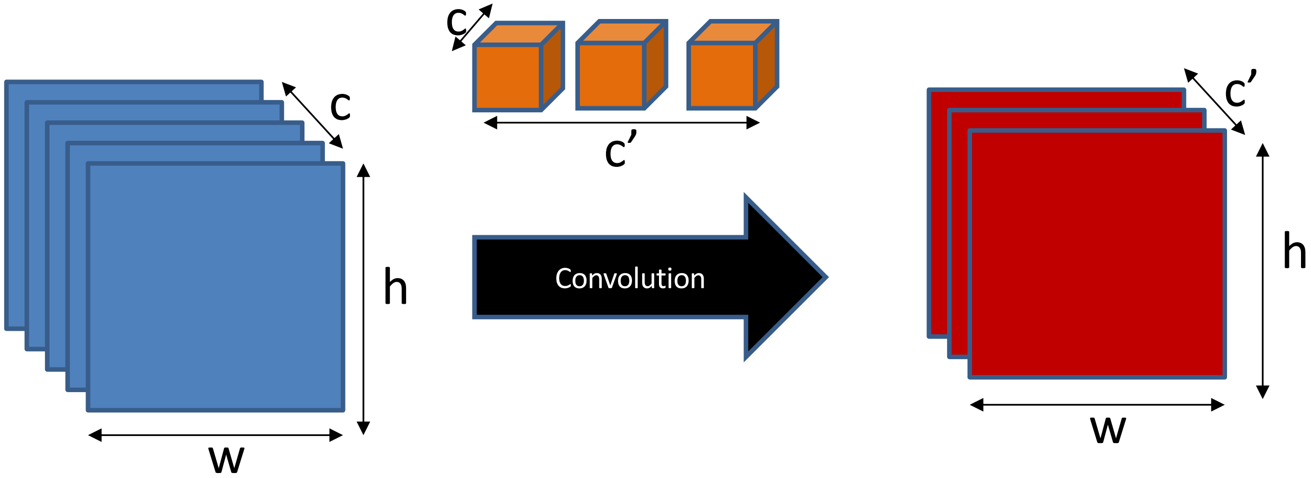
We stack these up to get a "new image" of size 28x28x6!

Convolutional Neural Networks

- Convolution layers interspersed with activation functions.

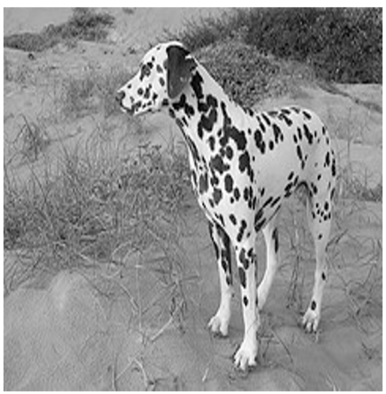


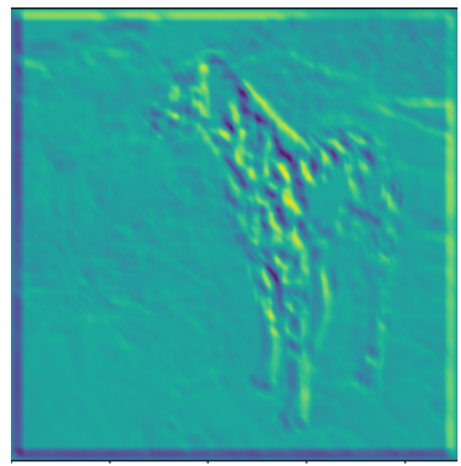
Convolution as a primitive



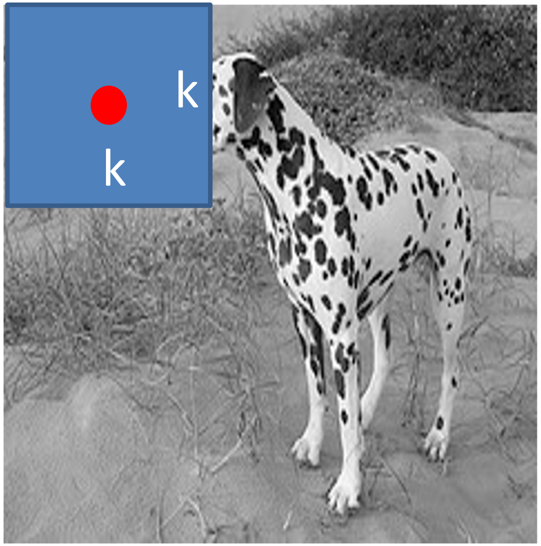
Convolution as a feature detector

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



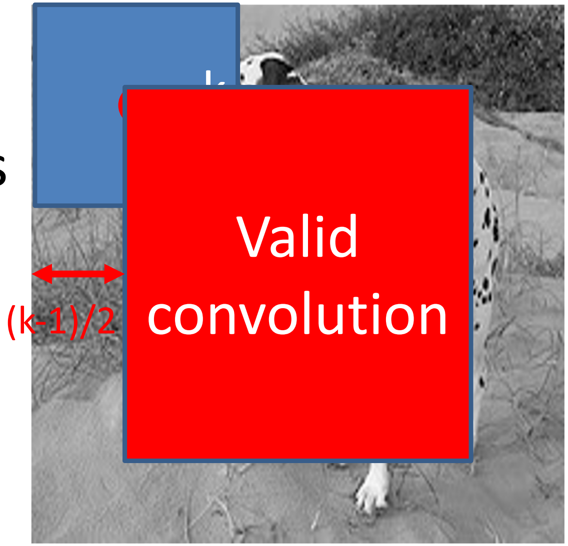


Kernel sizes and padding



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

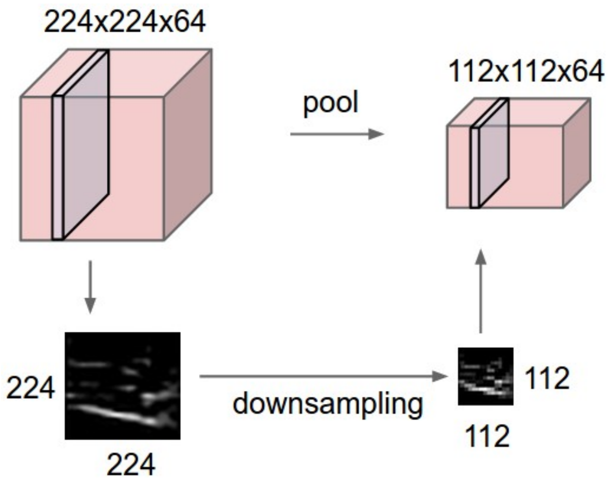
- Feature maps (“hidden layers”, “activations”, etc.) are no longer column vectors but 3D blobs:
  - Input # 256x256x3
  - Conv2d(in: 3, out:10) # 255x255x10
  - Conv2d(in: 10, out:20) # 255x255x20
  - …

Convolutional Layers

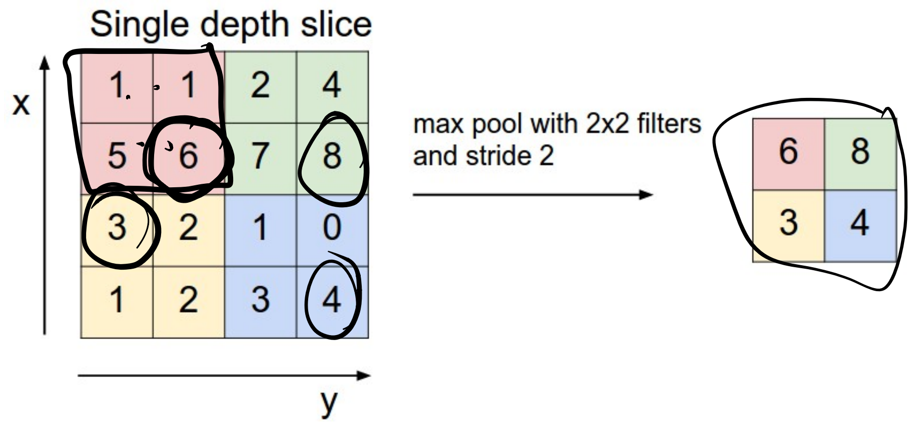
- Feature maps ("hidden layers", "activations", etc.) are no longer column vectors but 3D blobs:
  - Input # 256x256x3
  - Conv2d(in: 3, out:10) # 255x255x10
  - Conv2d(in: 10, out:20) # 254x254x20
  - … this could get large quickly, and we ultimately need a vector that we can apply a linear classifier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

Downsampling, Subsampling, Pooling

Downsampling:



Max pooling:

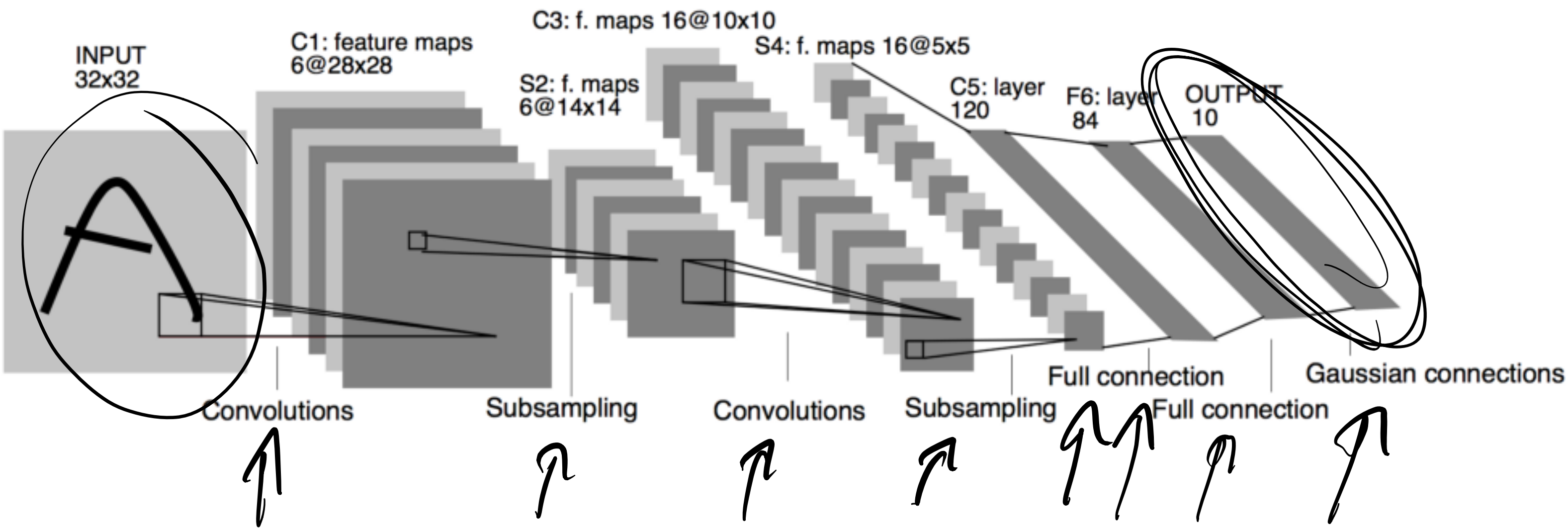


- Reducing spatial dimensions:
  - Subsample (e.g. throw away every other pixel)
  - Average pooling
  - Max pooling (most commonly used)

Convolutional Networks

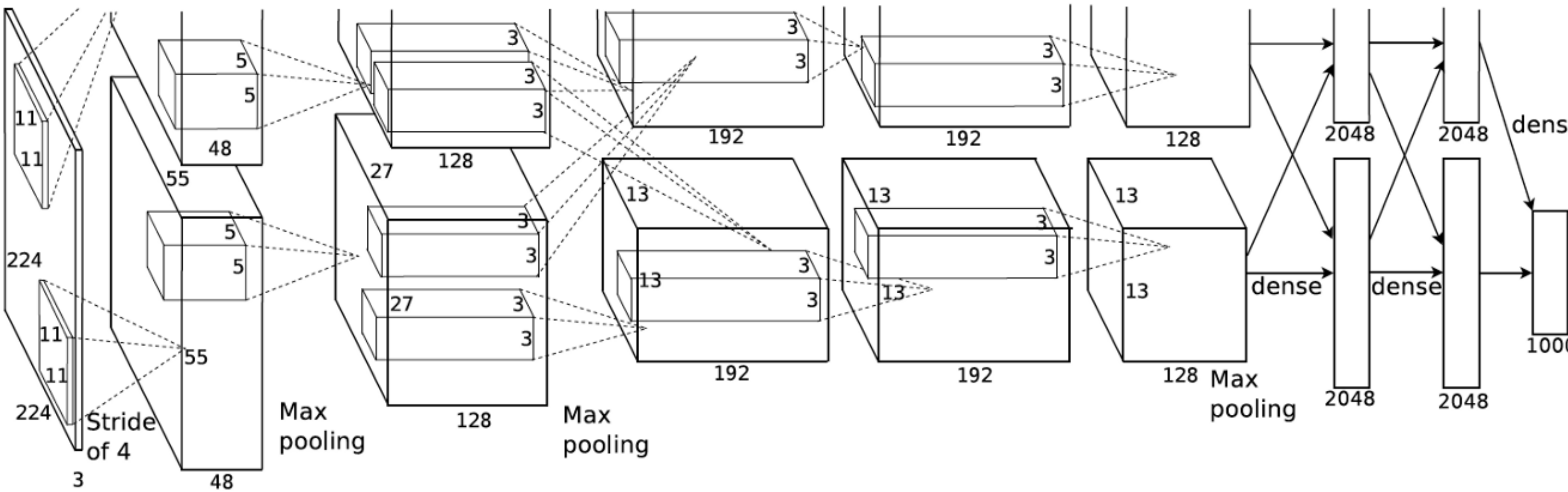
- Feature maps (“hidden layers”, “activations”, etc.) are no longer column vectors but 3D blobs:
  - Input # 256x256x3
  - Conv2d(in: 3, out:10) # 255x255x10
  - Subsample (2x2)
  - Conv2d(in: 10, out:20) # 127x127x20
  - …
  - Conv/subsample until 1x1xC
  - Or at some point, just unravel HxWxC into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

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

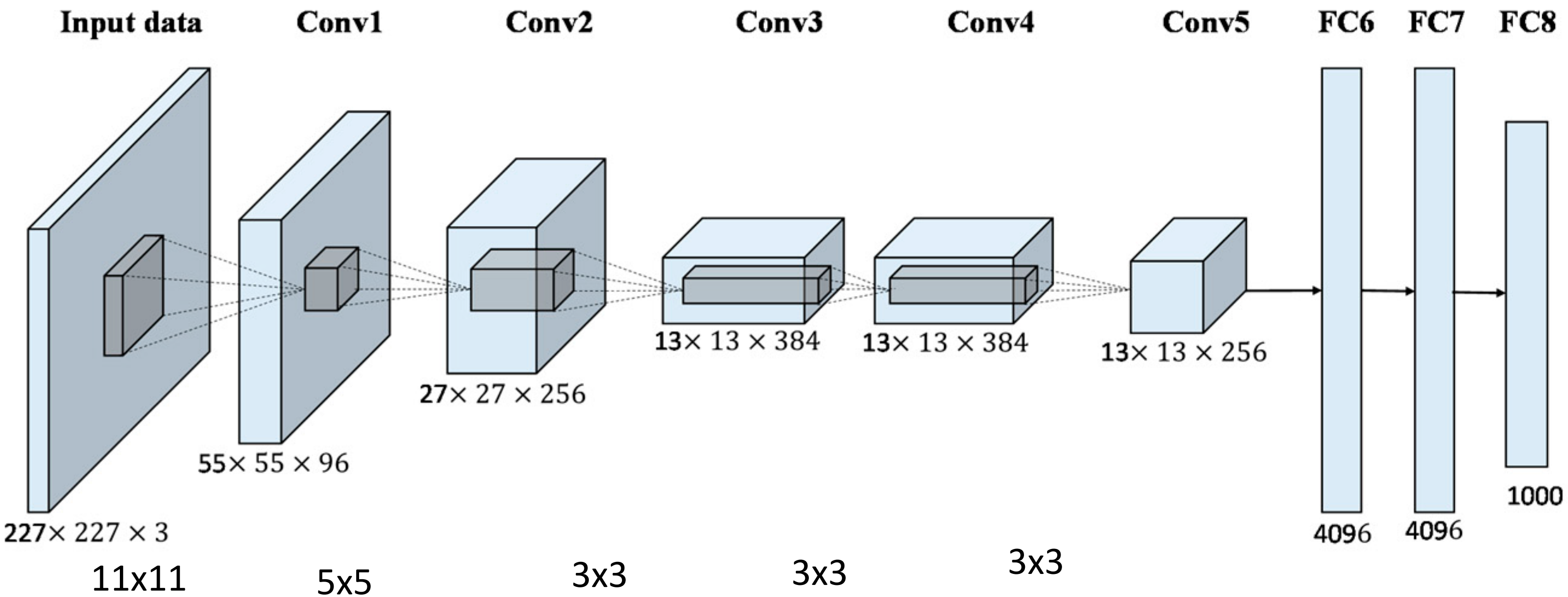


- Today's architectures still look a lot like this!

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

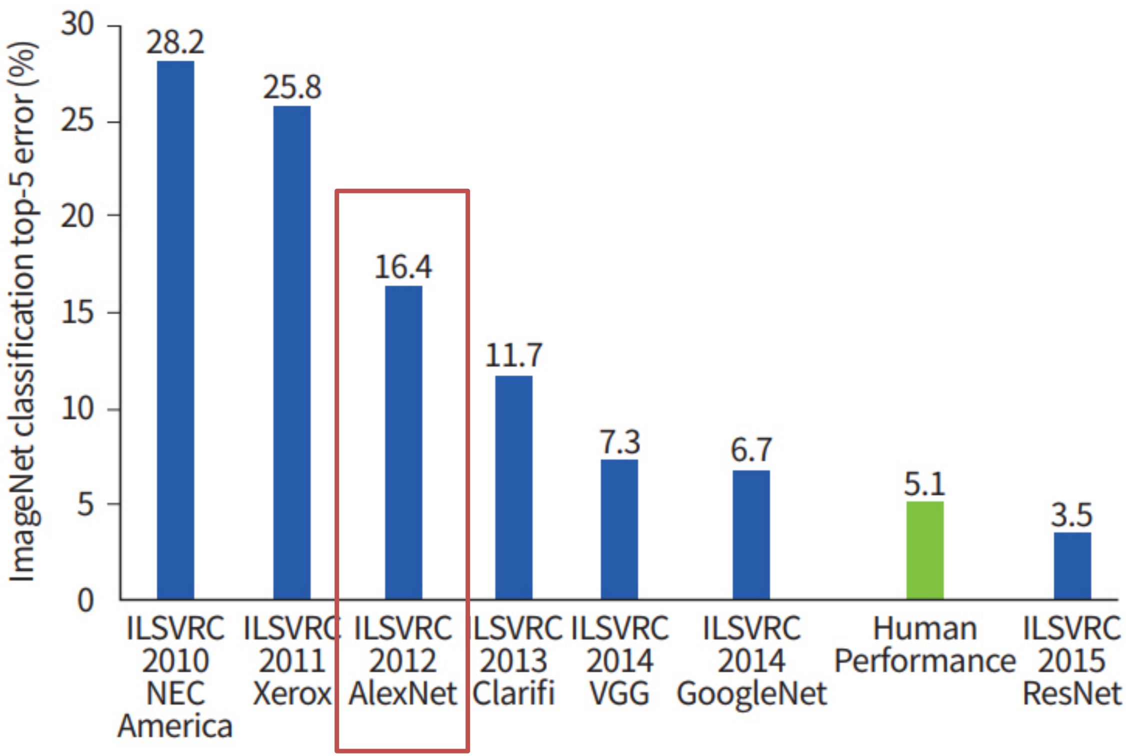


The CNN that made them cool: AlexNet [Krizhevsky et al. 2012]



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