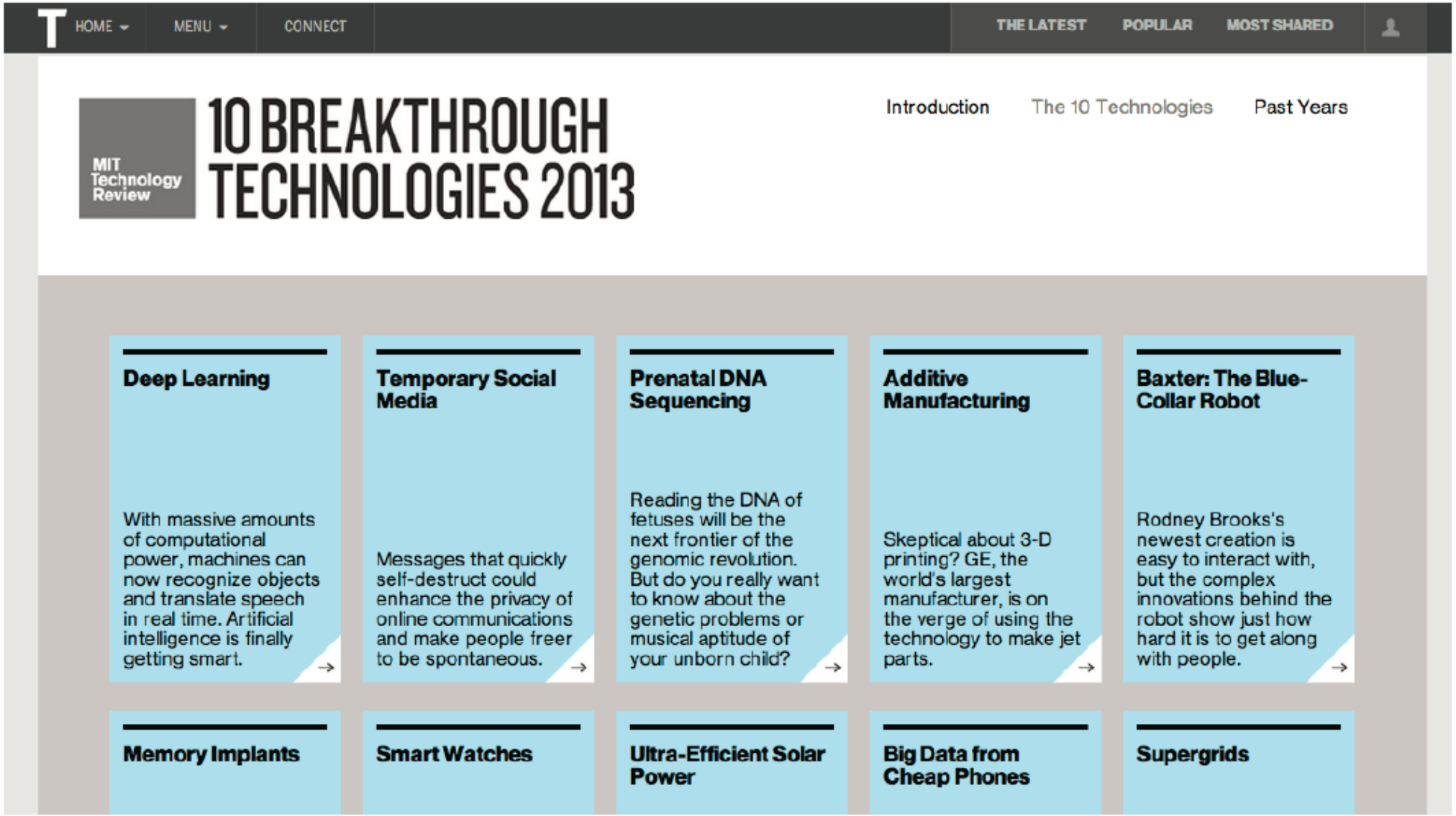
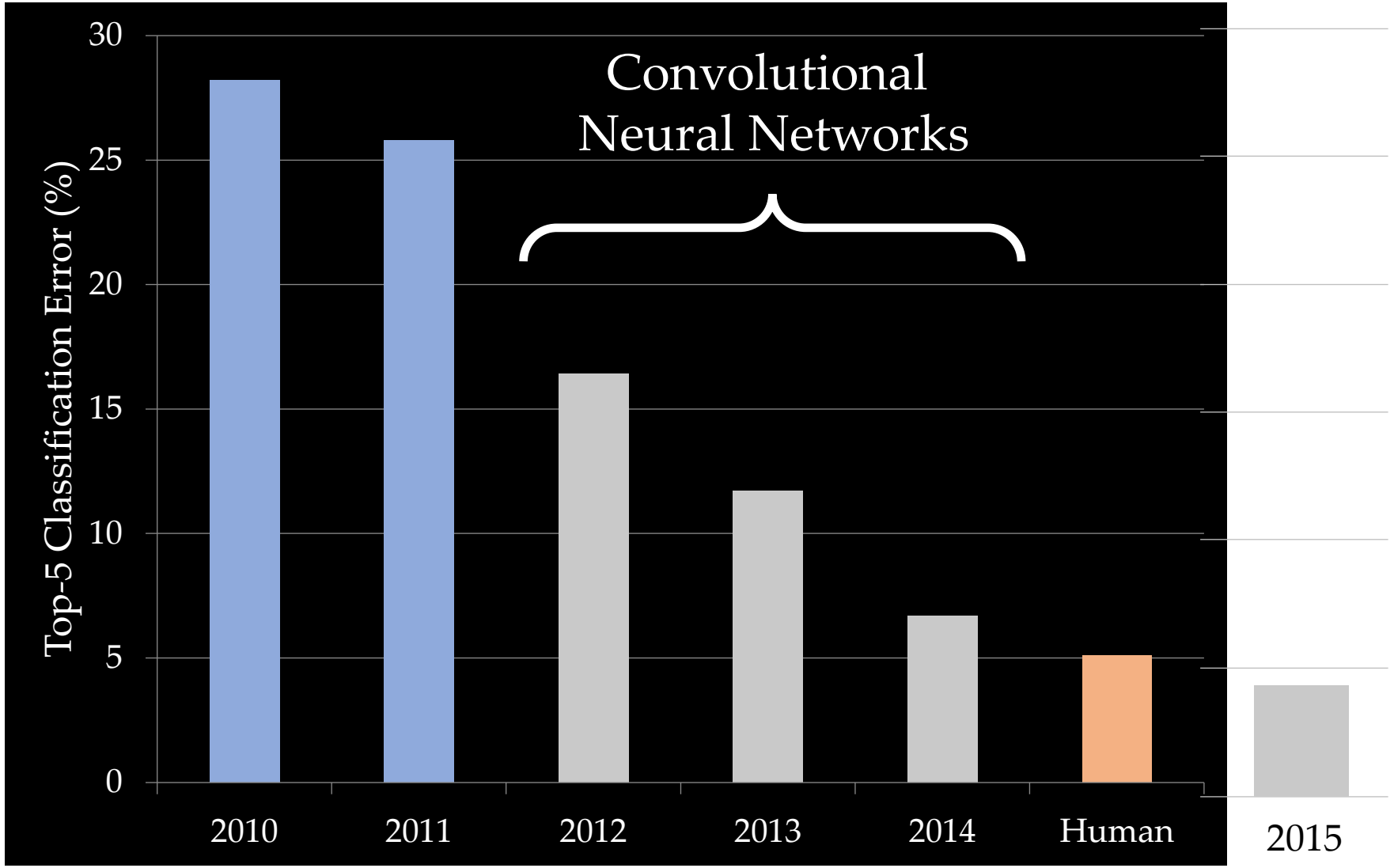
Convolutional Neural Networks



MIT Technology Review, April 23rd, 2013

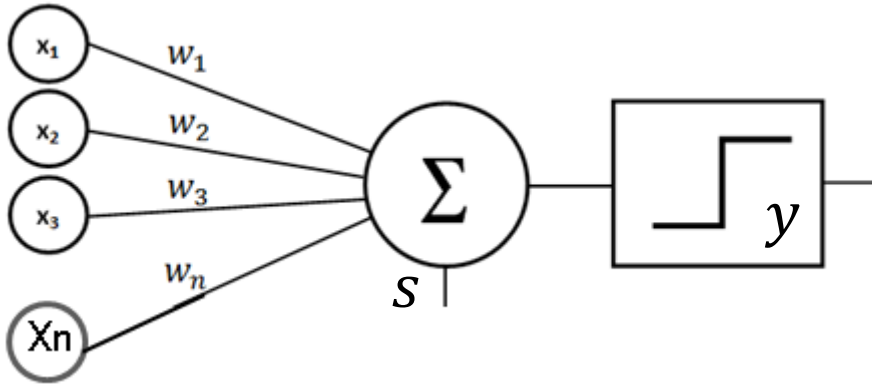
ImageNet Classification (2010 – 2015)



The Perceptron Model

Frank Rosenblatt

- Psychologist, Logician
- Inventor of the solution to everything, aka the Perceptron (1958)



$$s = \sum_{1}^{n} w_i x_i$$

$$s = \sum_{1}^{n} w_i x_i$$

- A number of inputs combine linearly
- Threshold logic: fire if combined input exceeds a threshold

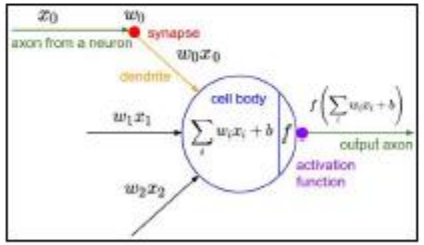
Perceptron

The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.

The machine was connected to a camera that used 20×20 cadmium sulfide photocells to produce a 400-pixel image.

recognized
letters of the alphabet

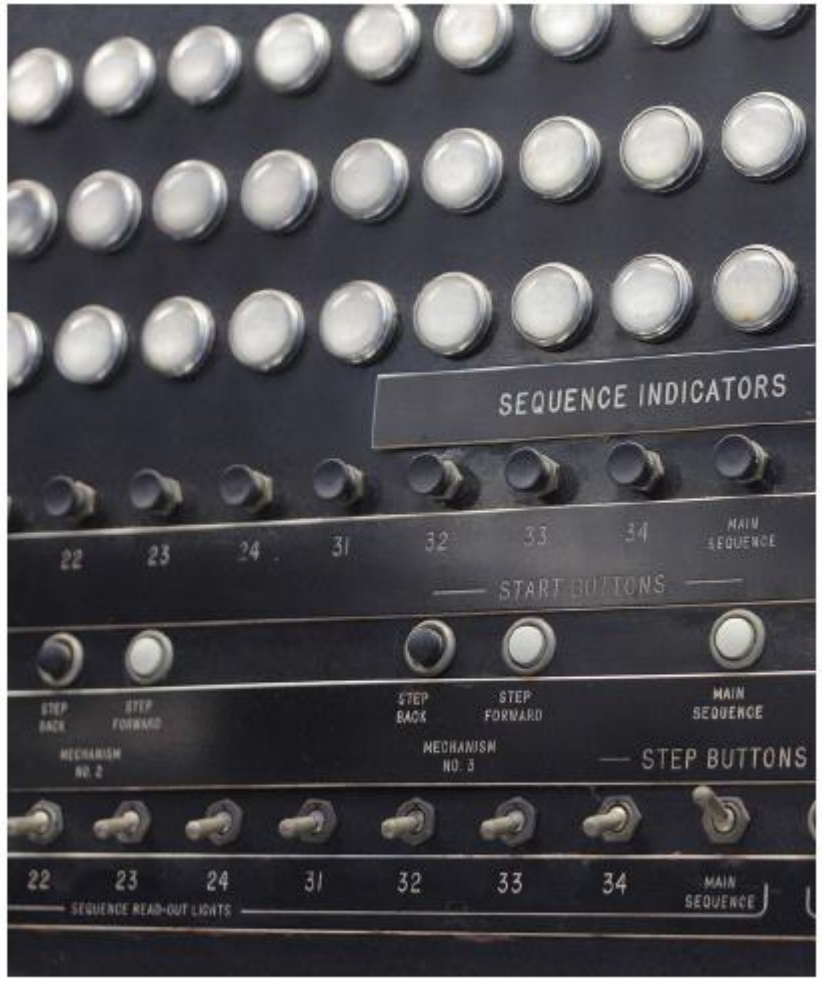
$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$



update rule:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i},$$

Frank Rosenblatt, ~1957: Perceptron



1958 New York Times...

NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

In today's demonstration, the "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.

original of modern networks

- Yann Lecun, LeNet, 1998
  - Recognizing hand written digits

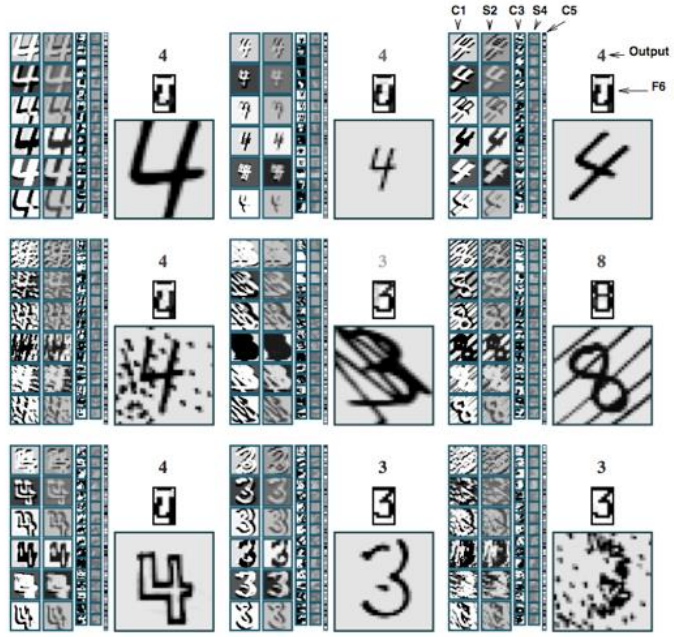


Fig. 13. Examples of unusual, distorted, and noisy characters correctly recognized by LeNet-5. The grey-level of the output label represents the penalty (lighter for higher penalties).

PROC. OF THE IEEE, NOVEMBER 1998

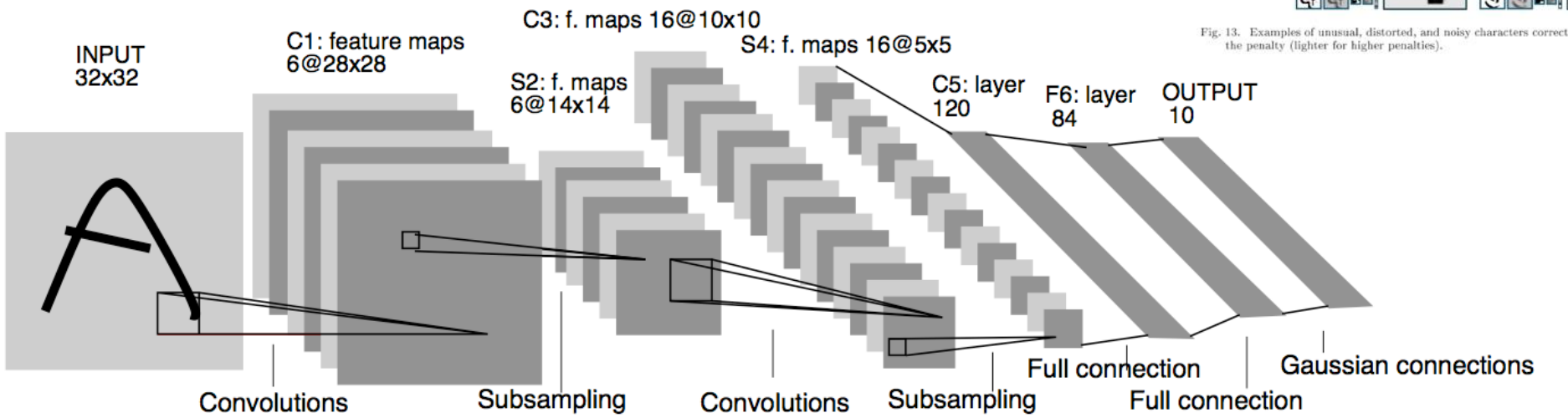
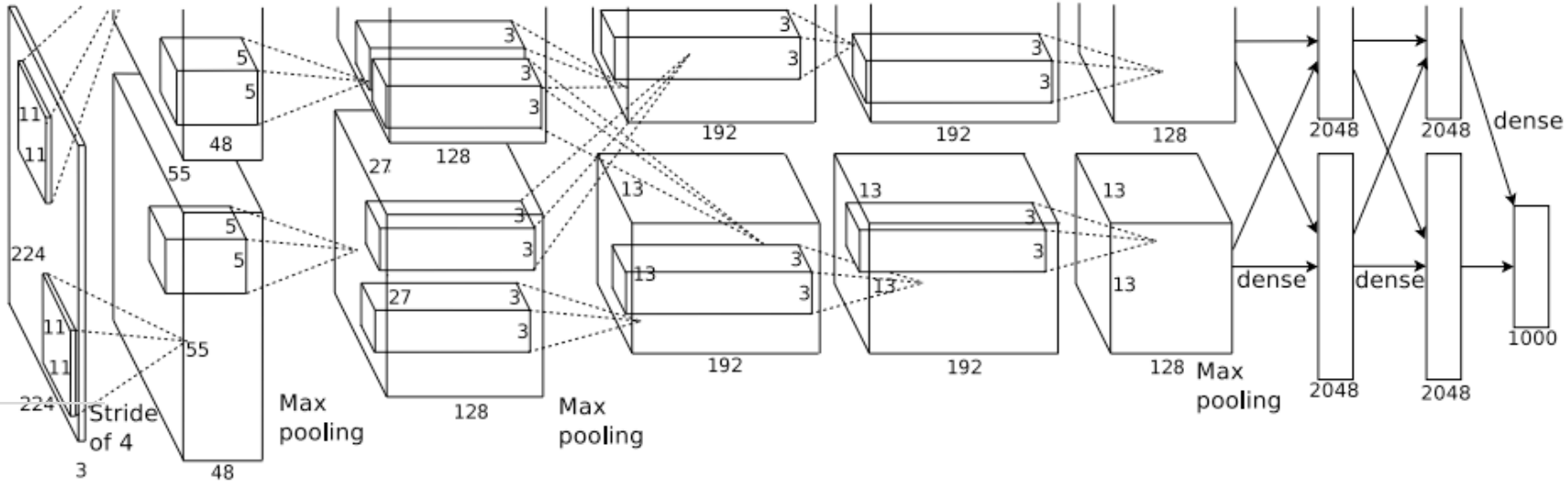
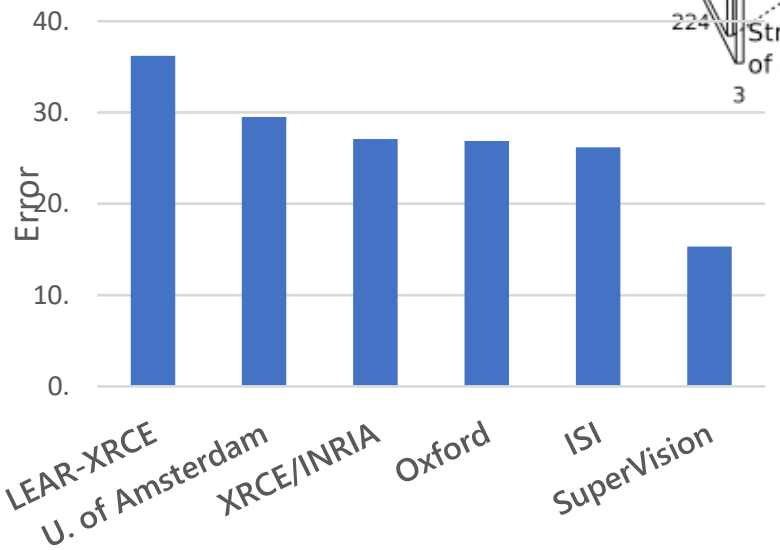


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

The First Strong Results in Vision

- Krizhevsky, Sutskever, and Hinton, AlexNet, 2012 NIPS

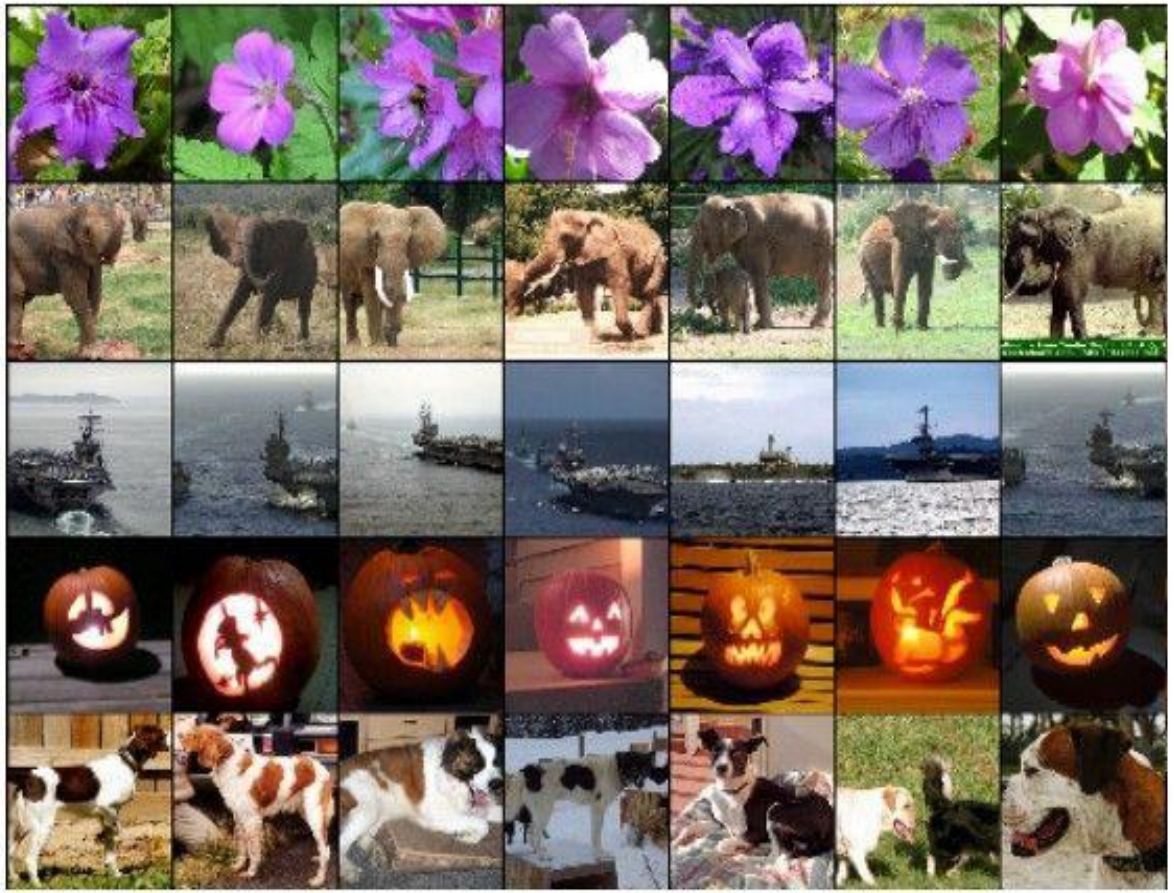




Today: ConvNets are everywhere

Classification

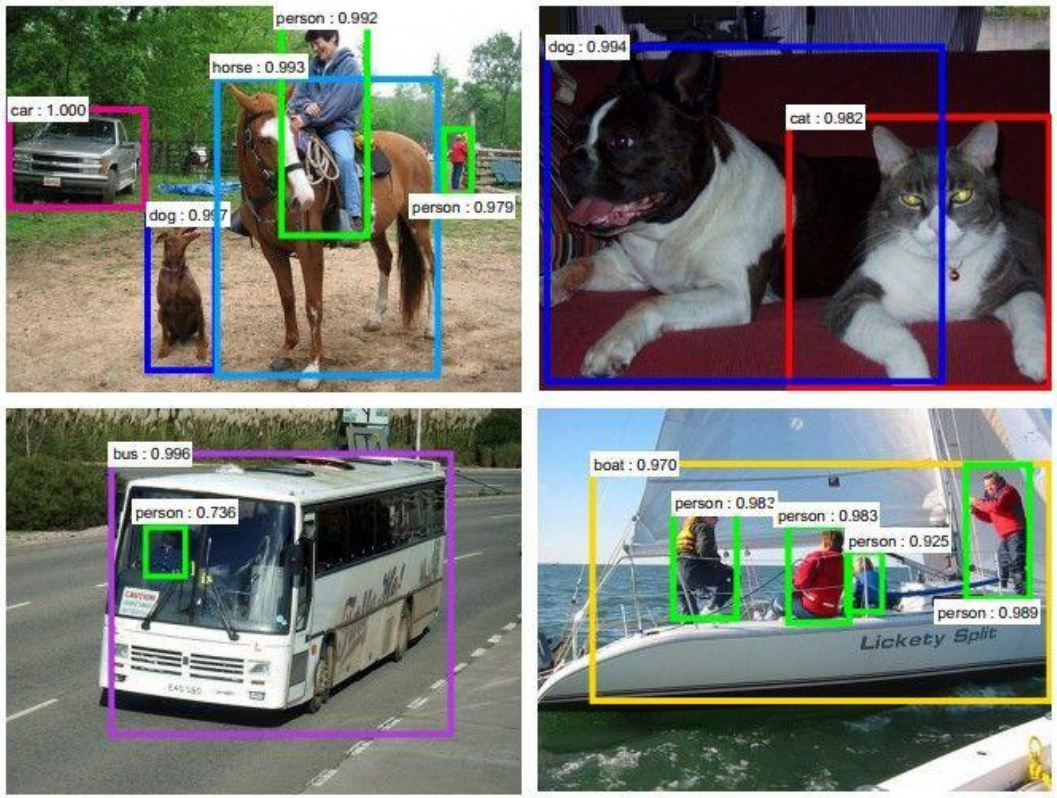
Retrieval



[Krizhevsky 2012]

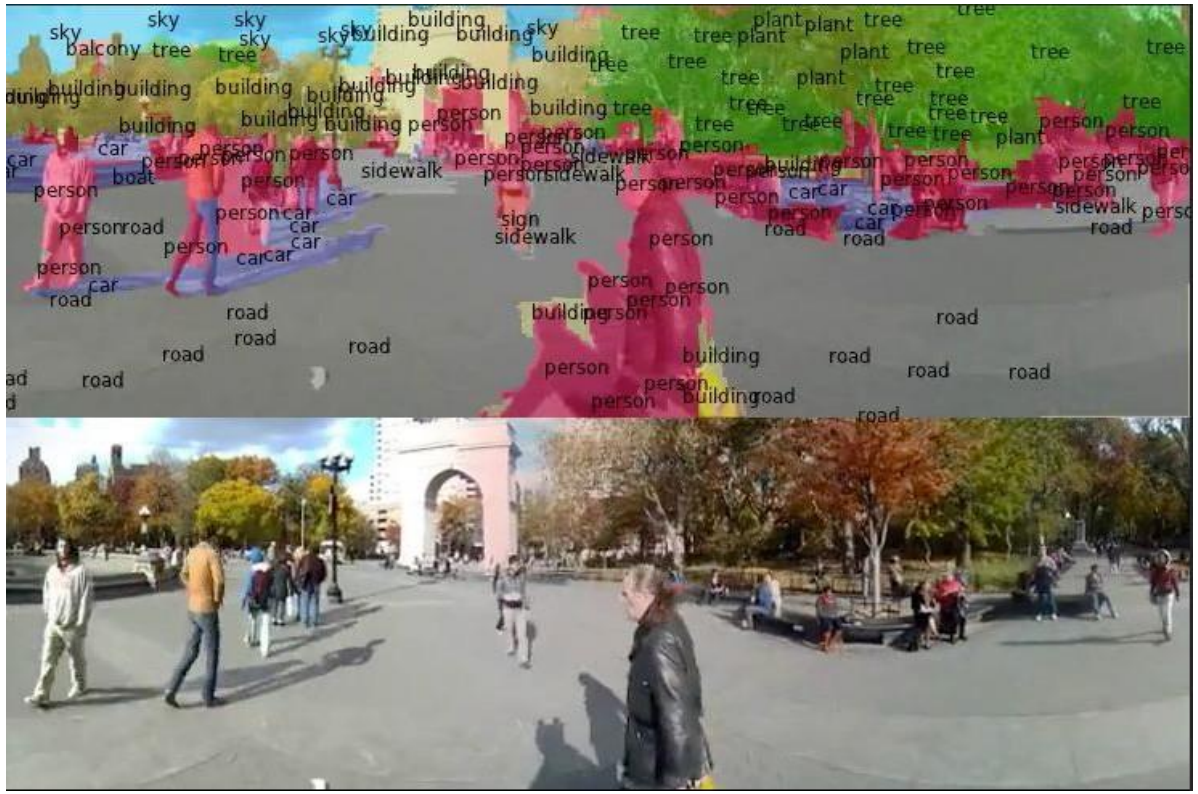
Today: ConvNets are everywhere

Detection



[Faster R-CNN: Ren, He, Girshick, Sun 2015]

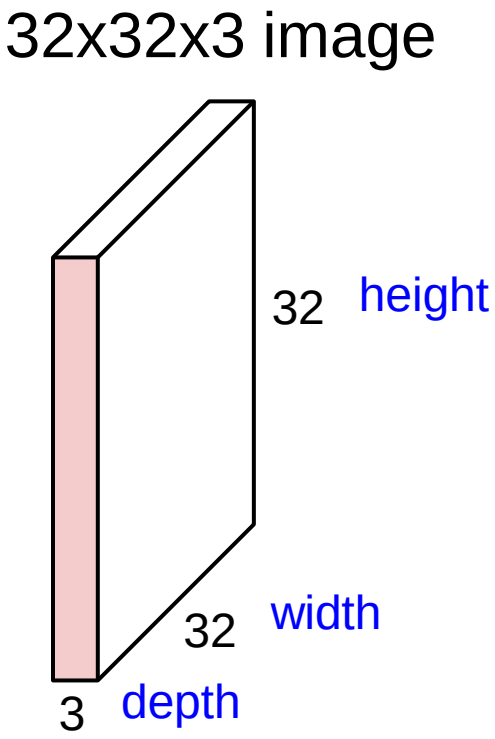
Segmentation



[Farabet et al., 2012]

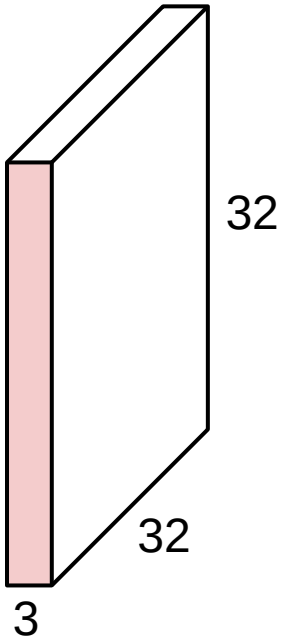
Questions?

Convolution Layer

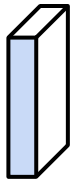


Convolution Layer

32x32x3 image



5x5x3 filter

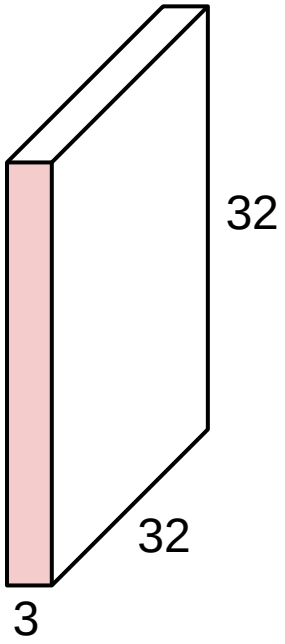


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

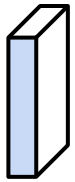
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

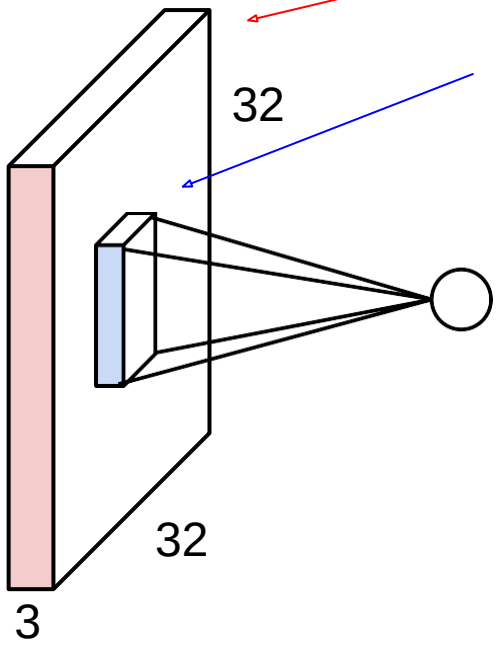


5x5x3 filter



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

Convolution Layer



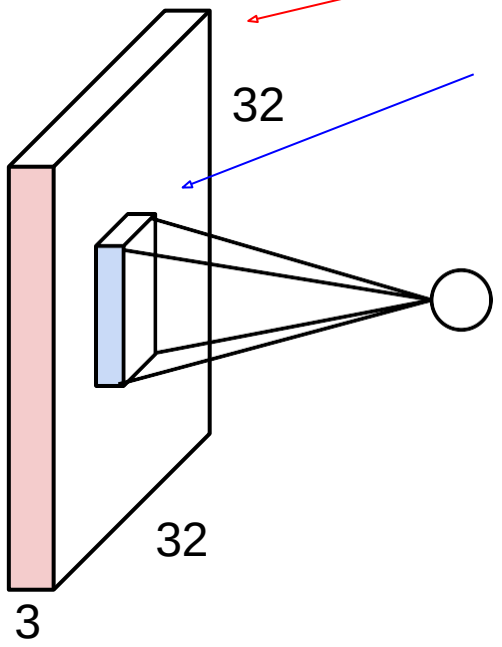
32x32x3 image

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)

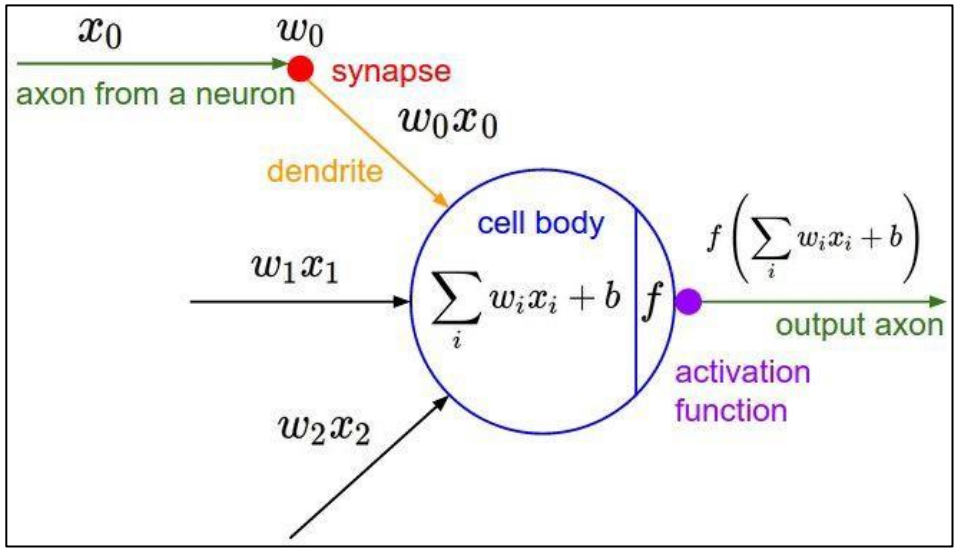
$$w^T x + b$$

The Neuron View of CONV Layer



32x32x3 image

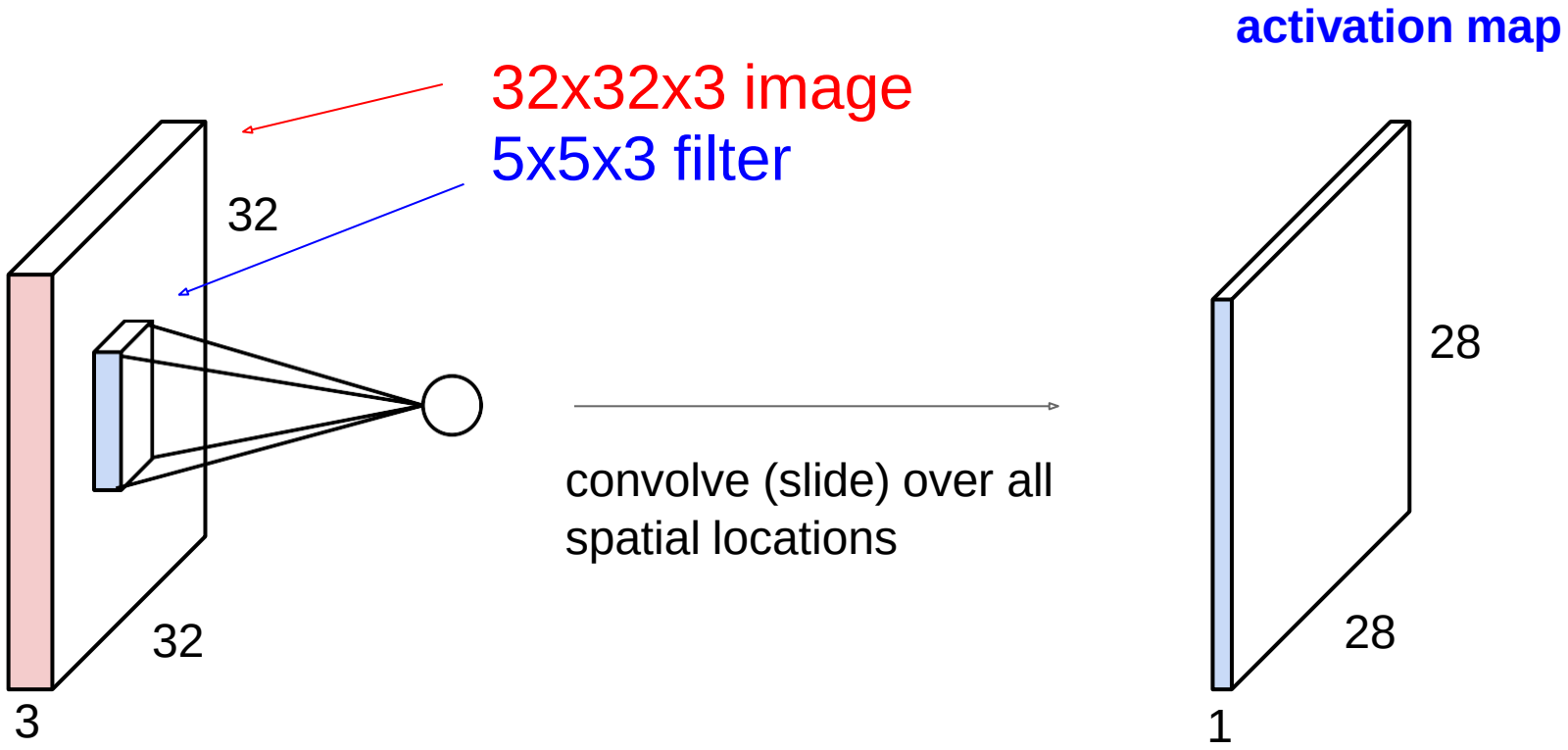
5x5x3 filter



It's just a neuron with local connectivity...

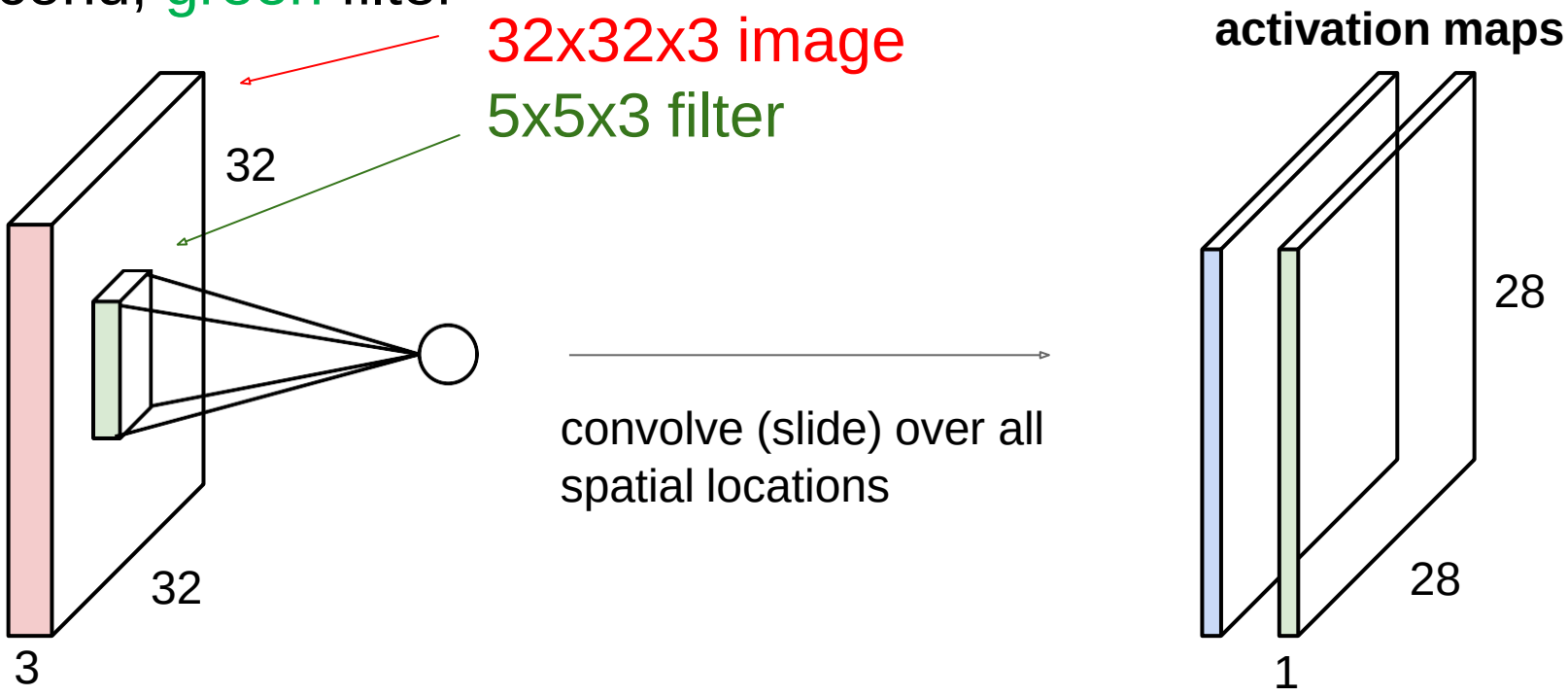
1 number:
the result of taking a dot product between the filter and this part of the image
(i.e. 5*5*3 = 75-dimensional dot product)

Convolution Layer

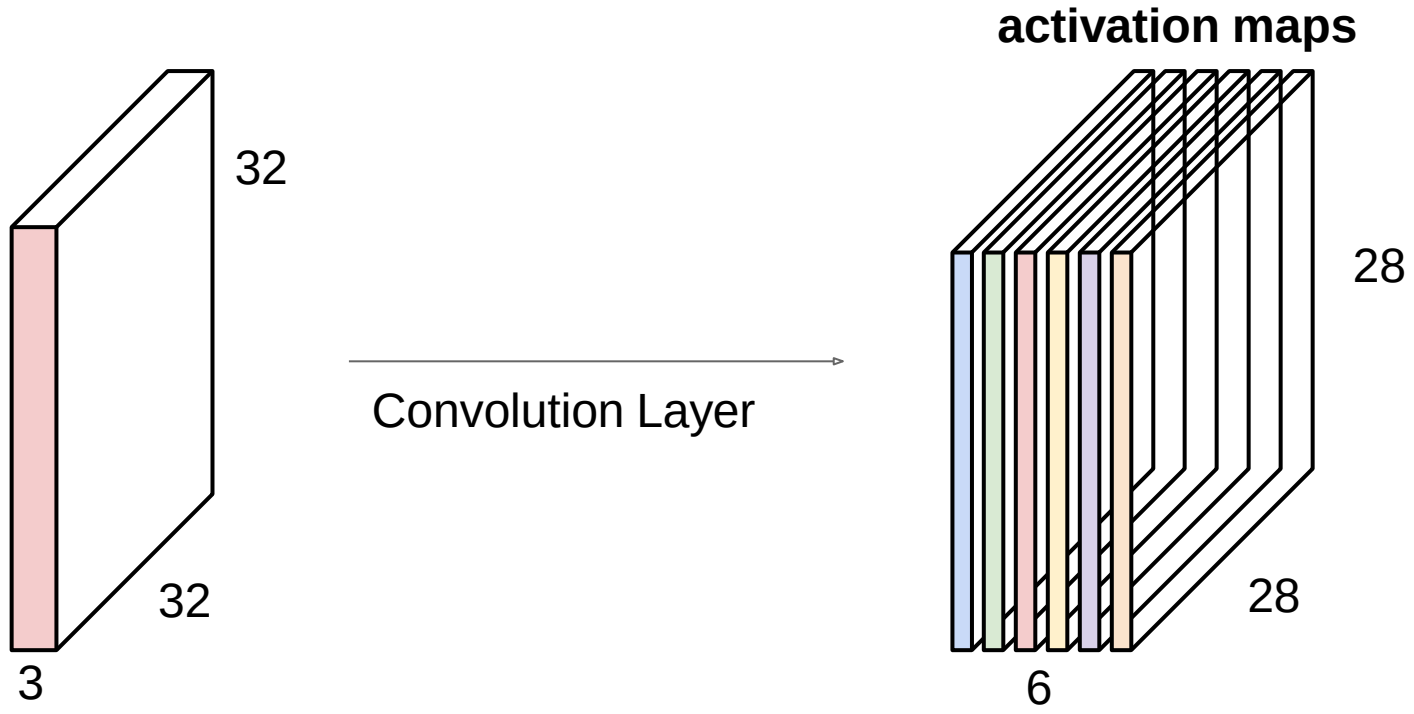


Convolution Layer

consider a second, green filter

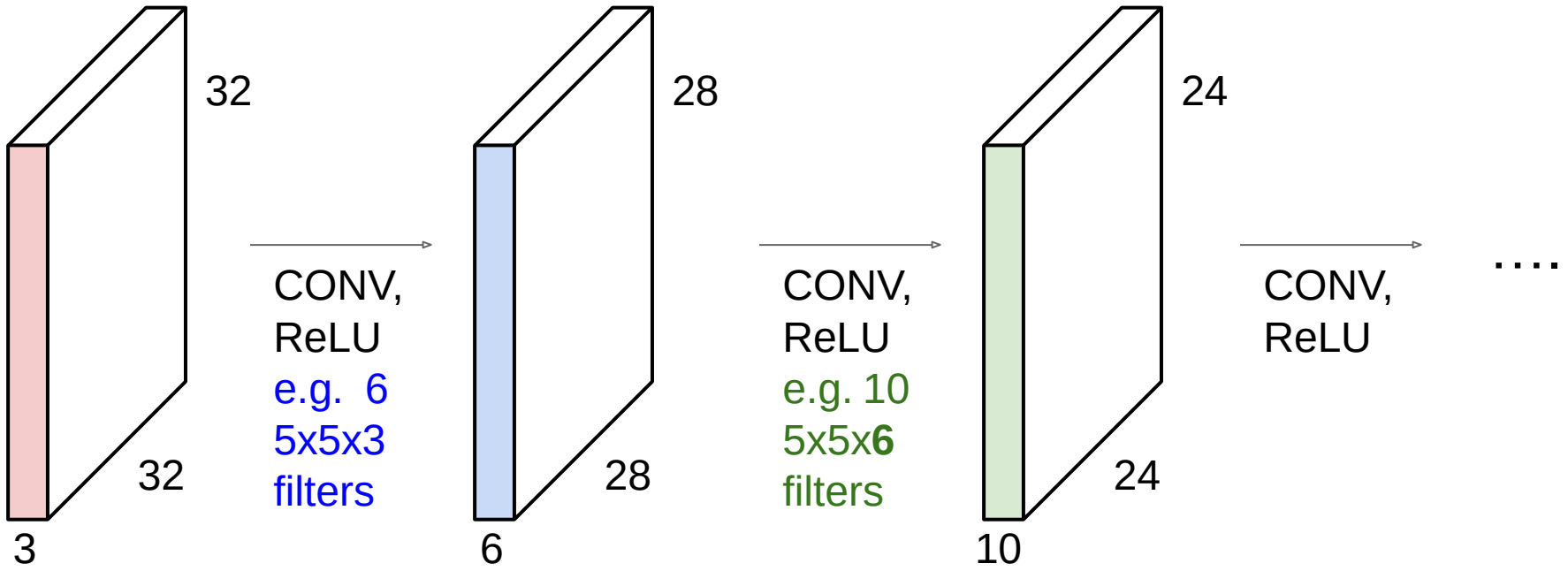


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

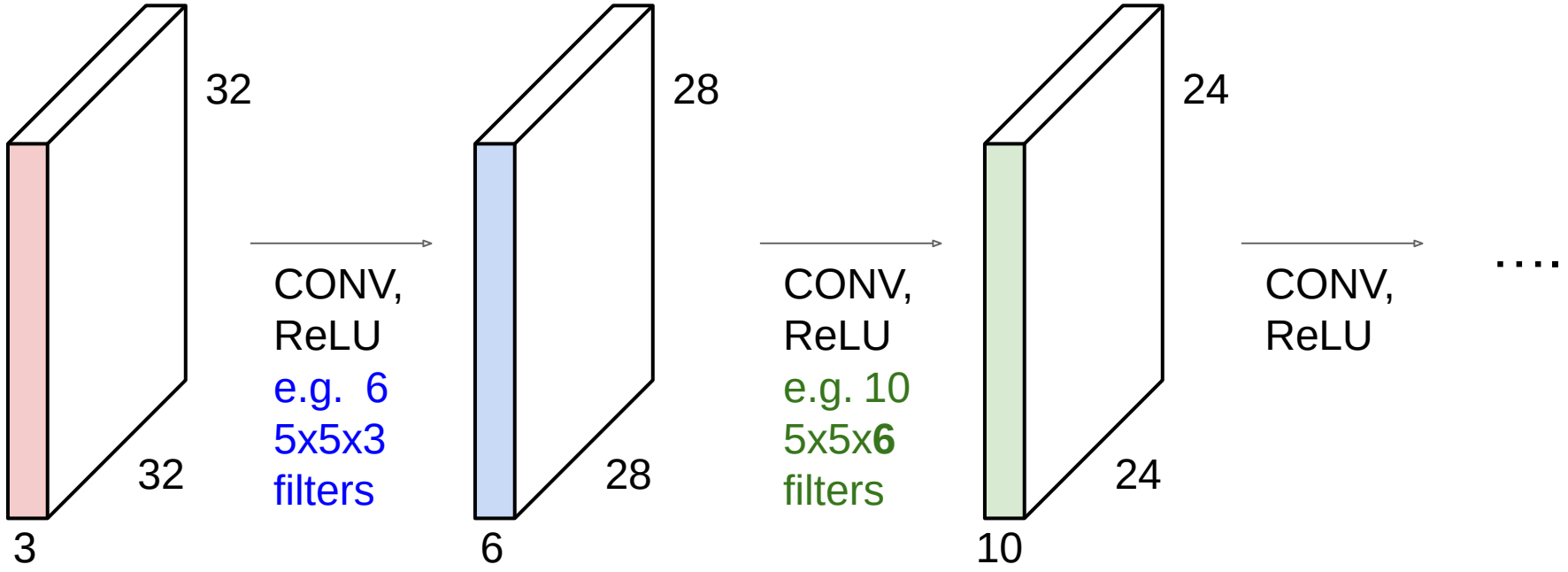


We stack these up to get a "new image" of size 28x28x6!

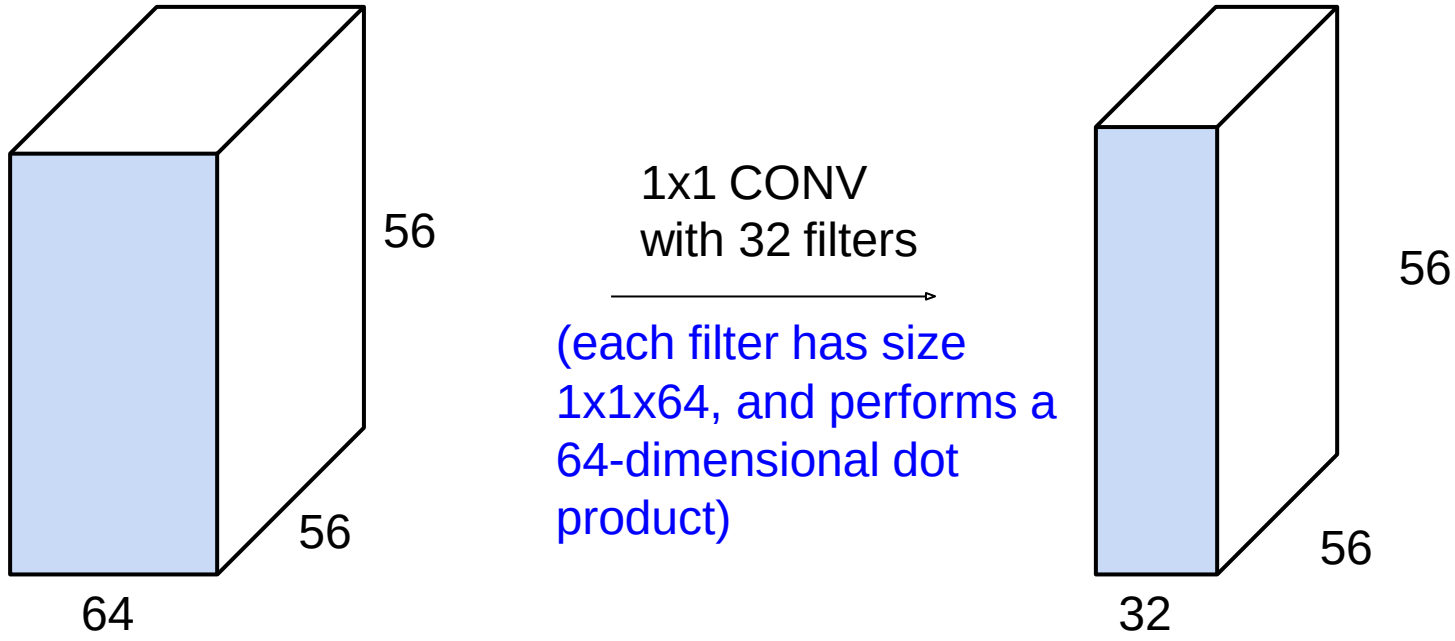
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



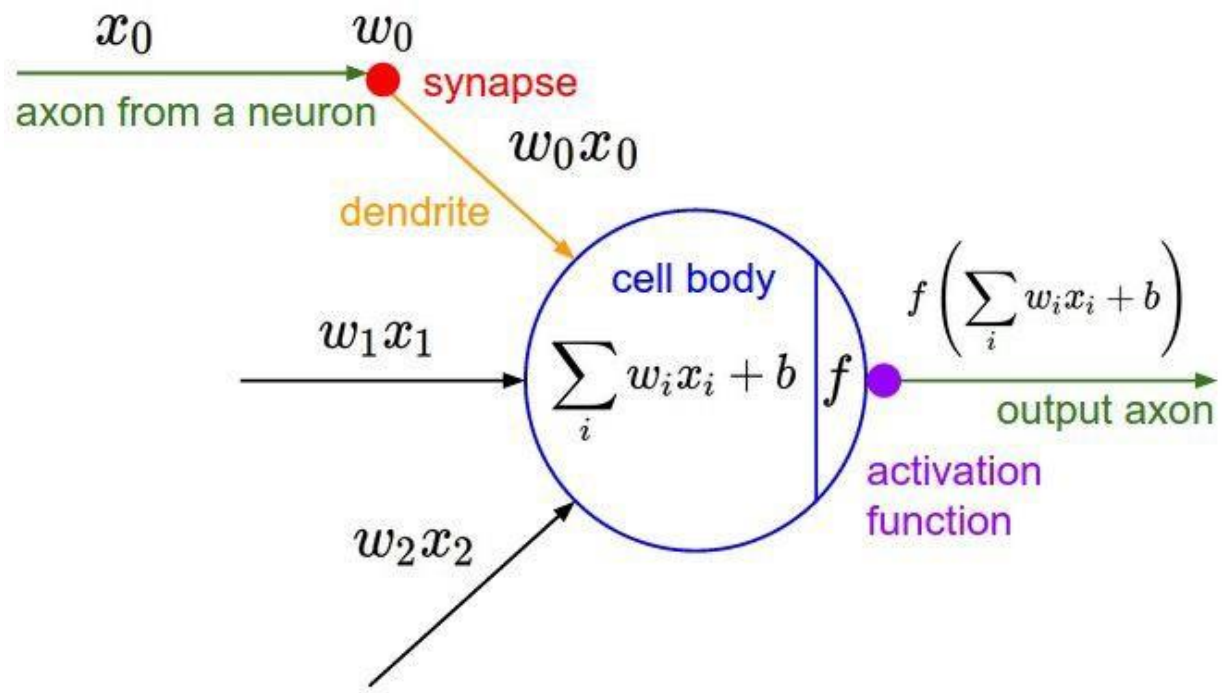
E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.



(btw, 1x1 convolution layers make perfect sense)



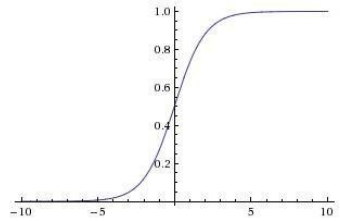
Activation Functions



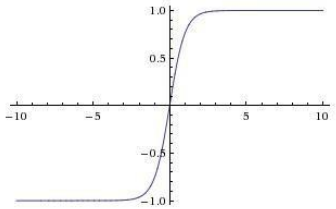
Activation Functions

Sigmoid

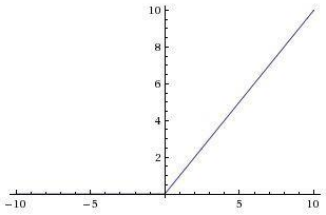
$$\sigma(x) = 1/(1+e^{-x})$$

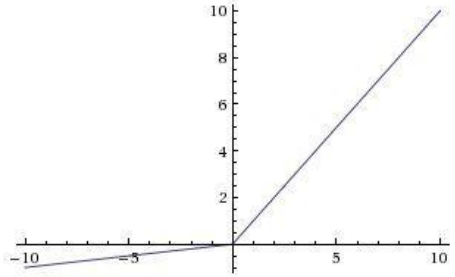


tanh $\tanh(x)$



ReLU $\max(0, x)$

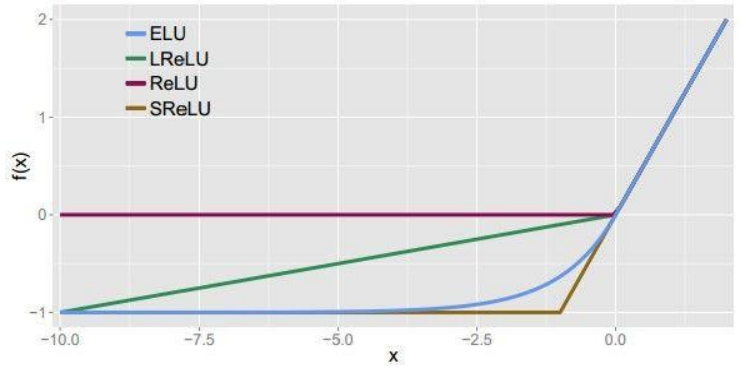




Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

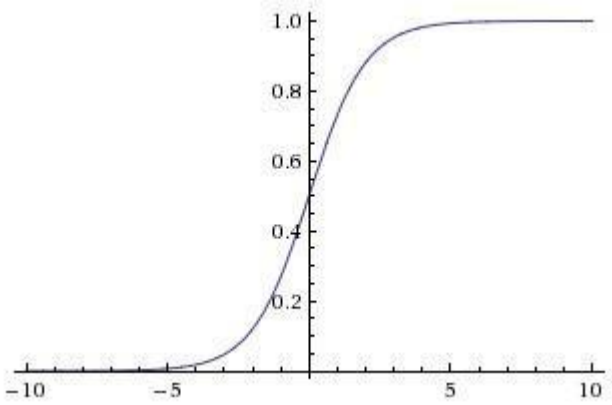
ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \leq 0 \end{cases}$$



Activation Functions

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

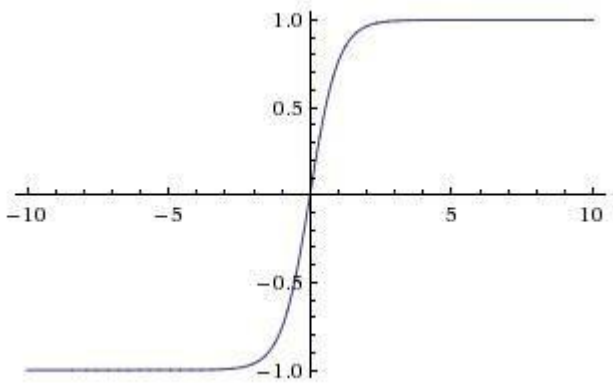


Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

- 3 problems:
  1. Saturated neurons “kill” the gradients
  2. Sigmoid outputs are not zero-centered
  3. exp() is a bit computational expensive

Activation Functions



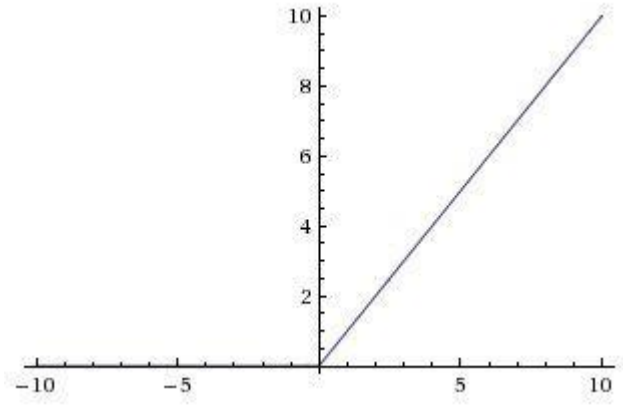
$\tanh(x)$

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

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

Activation Functions



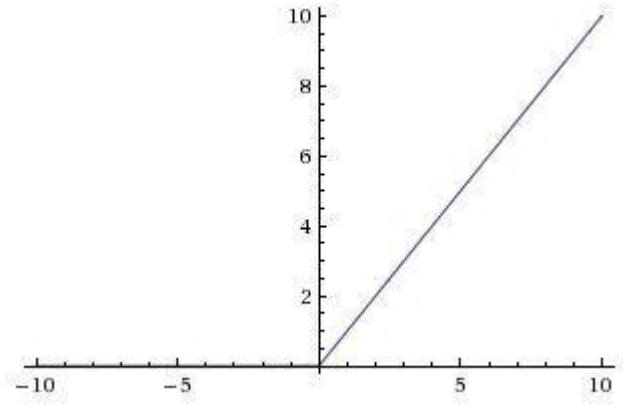
ReLU
(Rectified Linear Unit)

- Computes $\mathbf{f}(\boldsymbol{x}) = \mathbf{max}(\mathbf{0}, \boldsymbol{x})$

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

[Krizhevsky et al., 2012]

Activation Functions



ReLU
(Rectified Linear Unit)

- Computes $\mathbf{f}(\boldsymbol{x}) = \mathbf{max}(\mathbf{0}, \boldsymbol{x})$

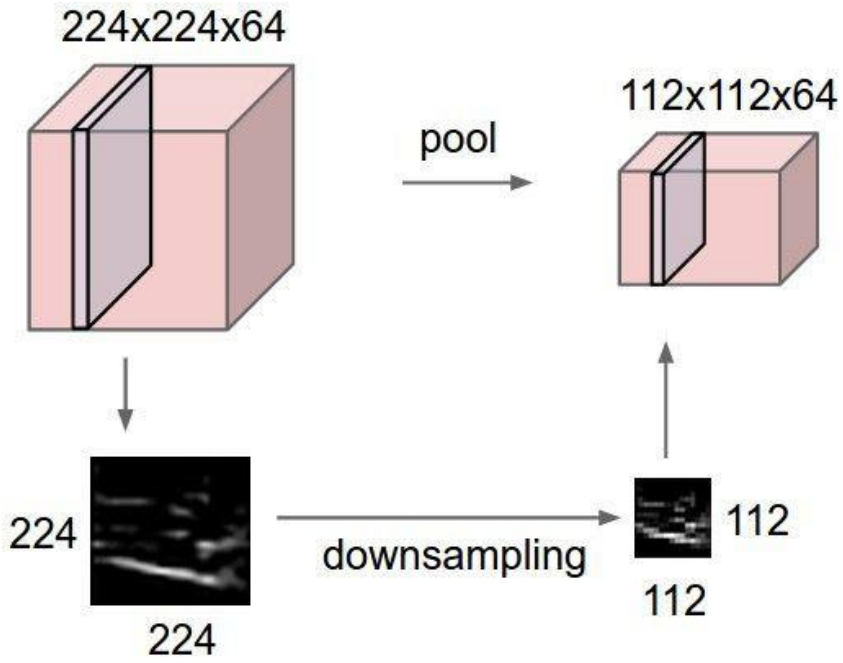
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

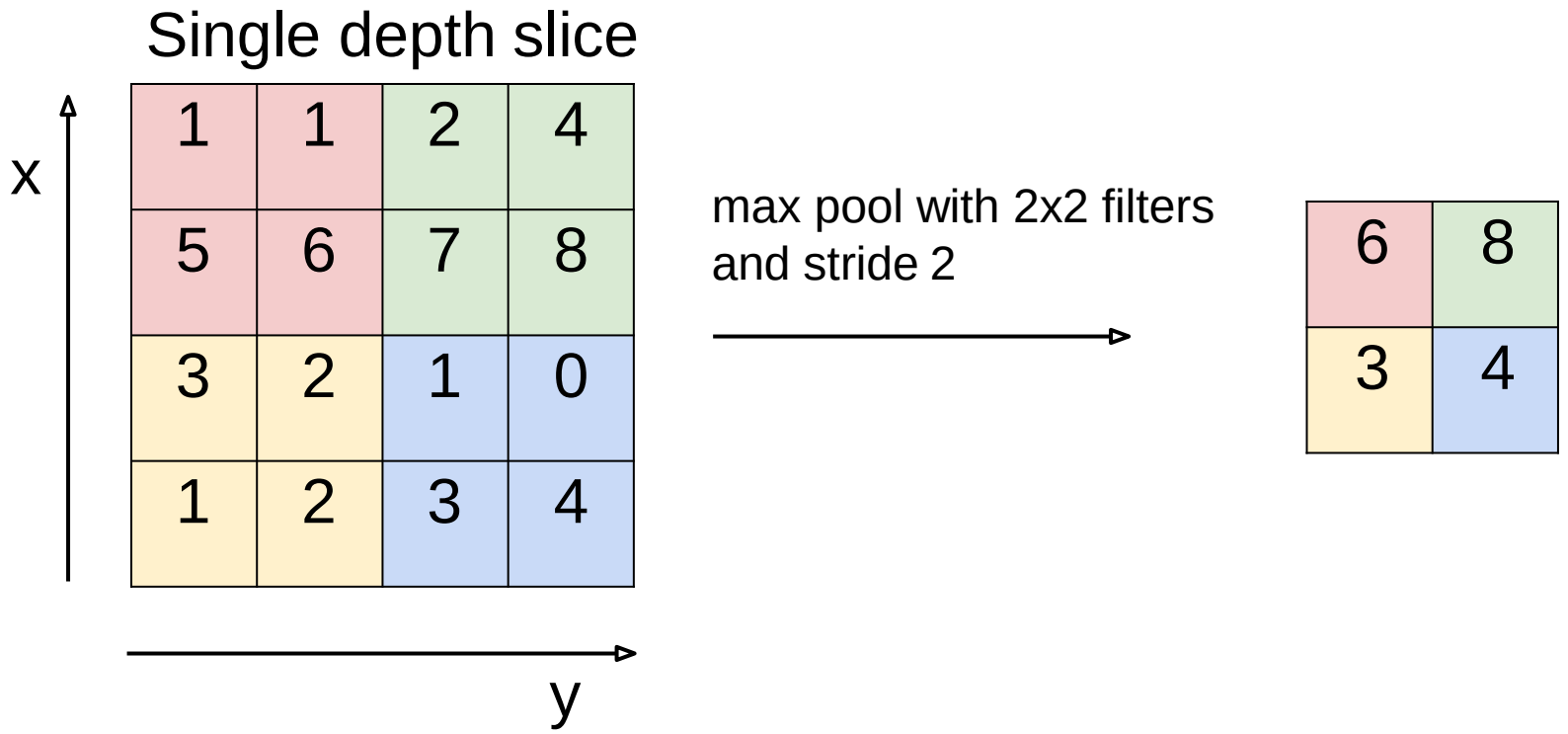
hint: what is the gradient when x < 0?

Pooling layer

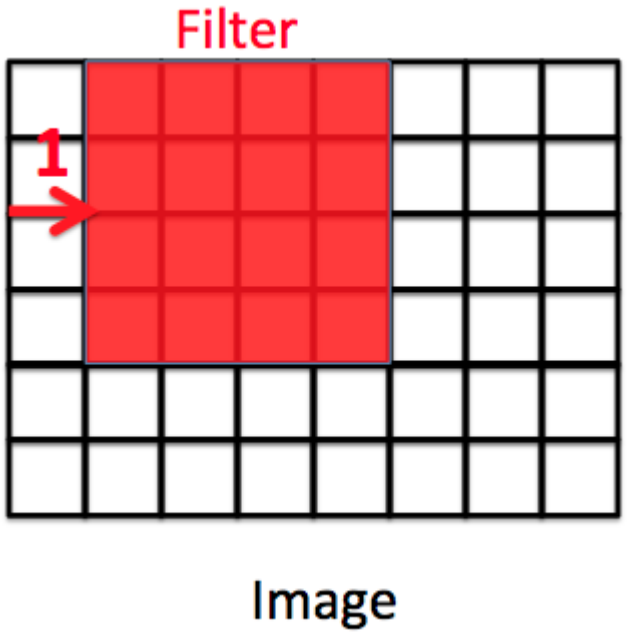
- makes the representations smaller and more manageable
- operates over each activation map independently:



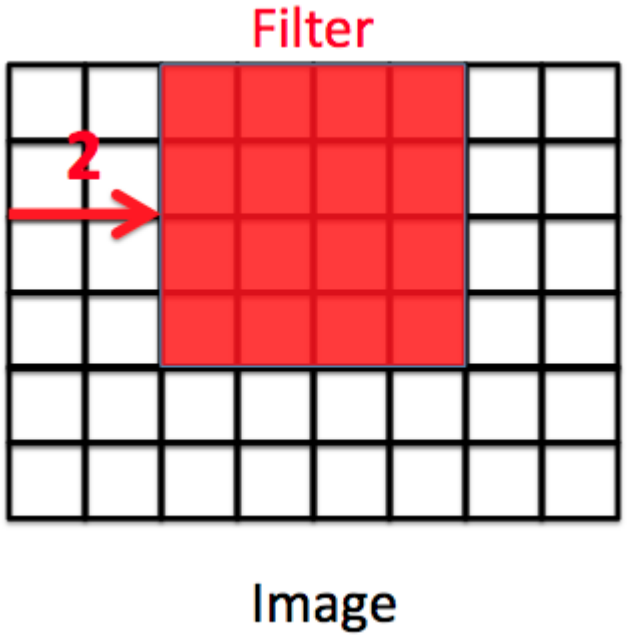
Max Pooling


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

Stride



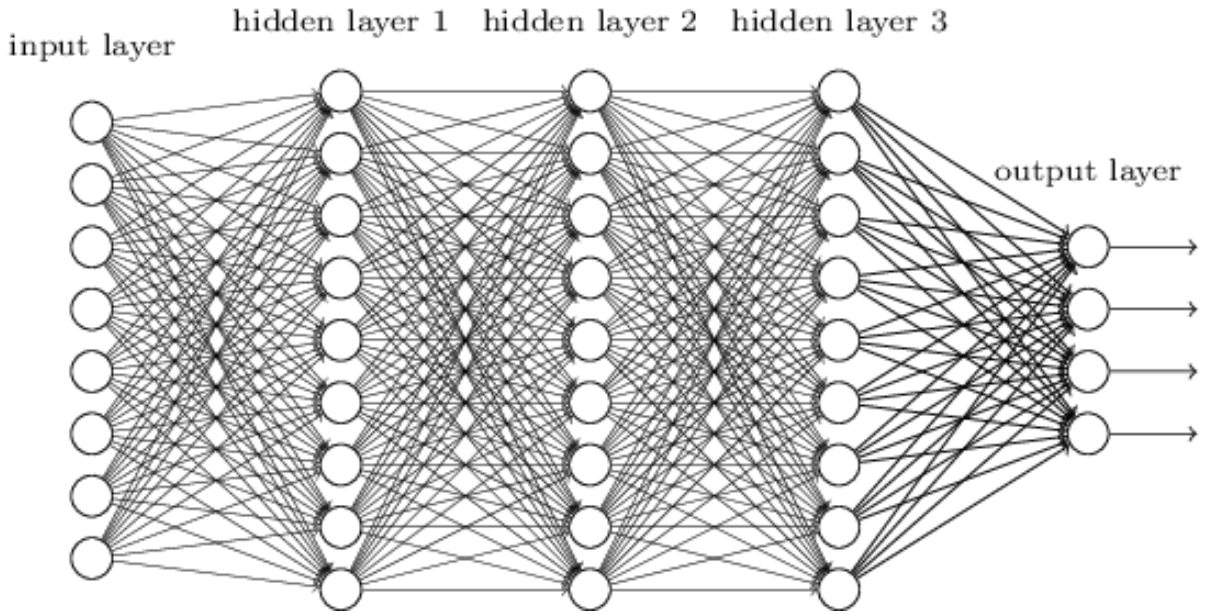
Stride = 1



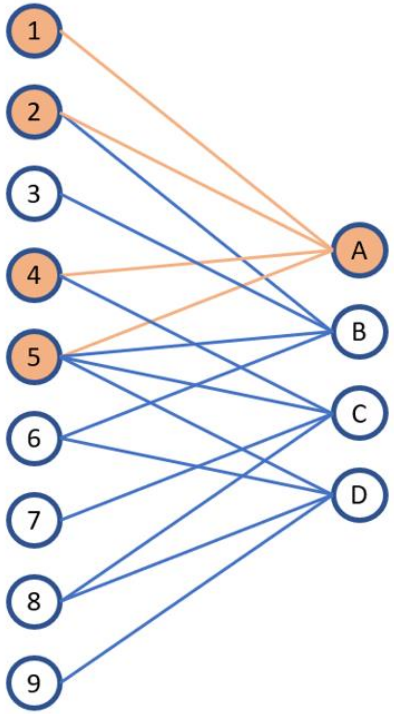
Stride = 2

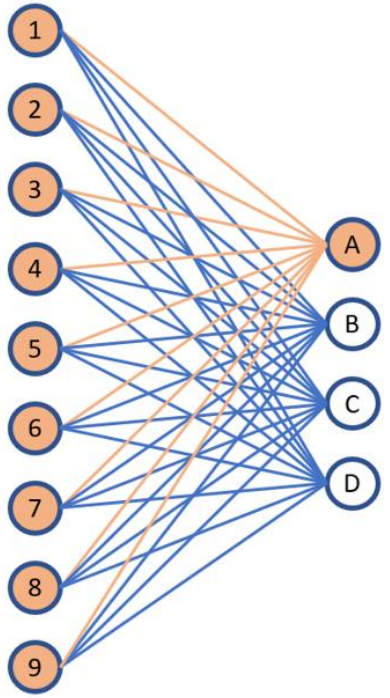
Fully Connected Layer (FC layer)

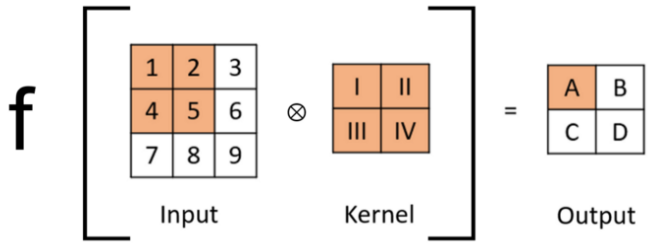
- Each neuron connects all neurons in the input volume
  - This is the conventional design of neural networks

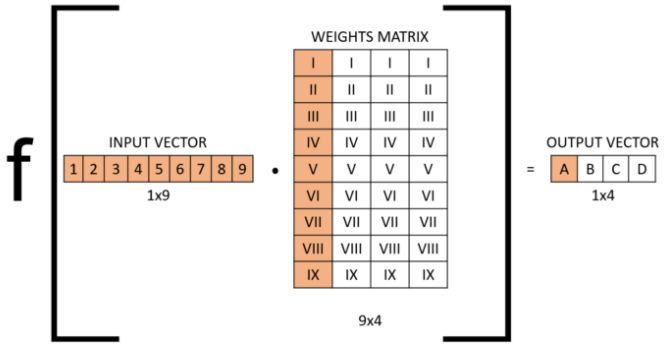


Convolution Layer vs FC layer

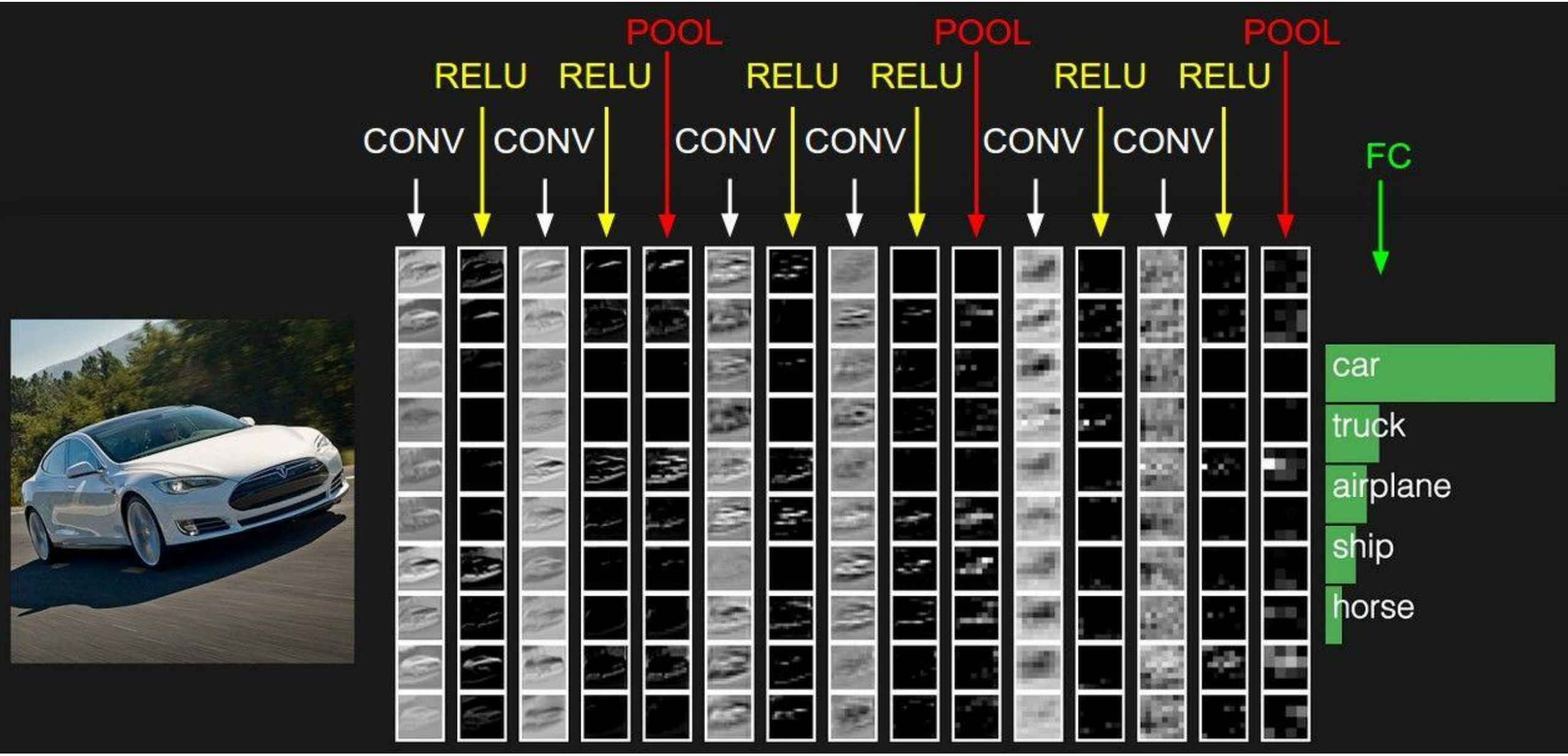








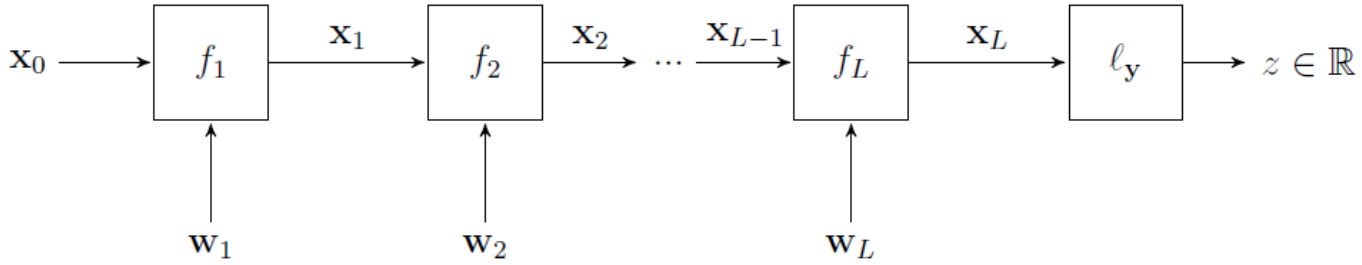
Overview


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

Questions?

Back-Propagation

- Given a neural network as the following:

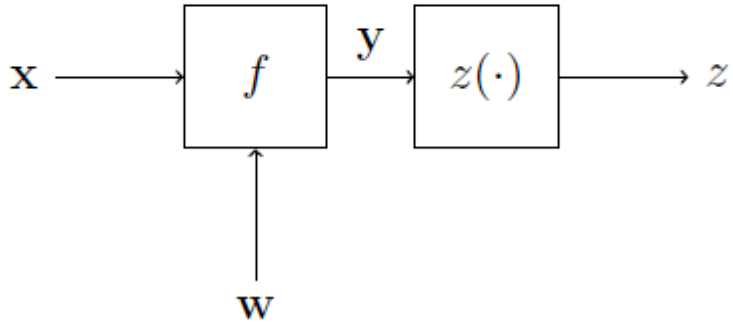


- We determine its parameters by gradient descent (to minimize some loss function)
- The gradient we need to compute is: dL/dw
  - w includes the parameters $w_1, w_2, \ldots, w_L$ at all layers
  - Typically, there are many parameters. We need fast computation.
- Back Propagation: compute gradient from backwards
  - The result of dL/dw_{l+1} can be re-used to compute dL/dw_l by chain rule

Back-Propagation

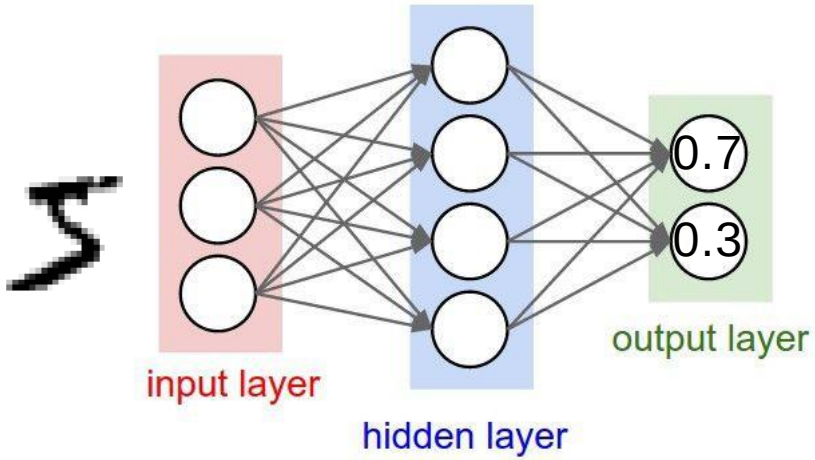
- In this way, gradients are ‘propagated’ from back layers

$$\underbrace{\ell \circ f_L(\cdot, \mathbf{w}_L) \circ f_{L-1}(\cdot, \mathbf{w}_{L-1}) \cdots \circ f_{l+1}(\cdot, \mathbf{w}_{l+1})}_{z(\cdot)} \circ f_l(\mathbf{x}_l, \mathbf{w}_l) \circ \dots$$



$$\frac{dL}{dx} = \frac{dL}{dy}\frac{dy}{dx}$$

Example

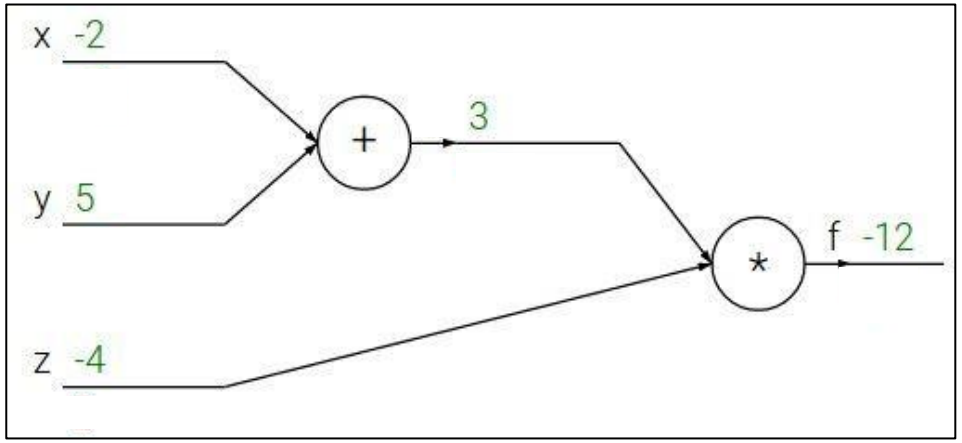


Probabilities being 5 and 7

1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4



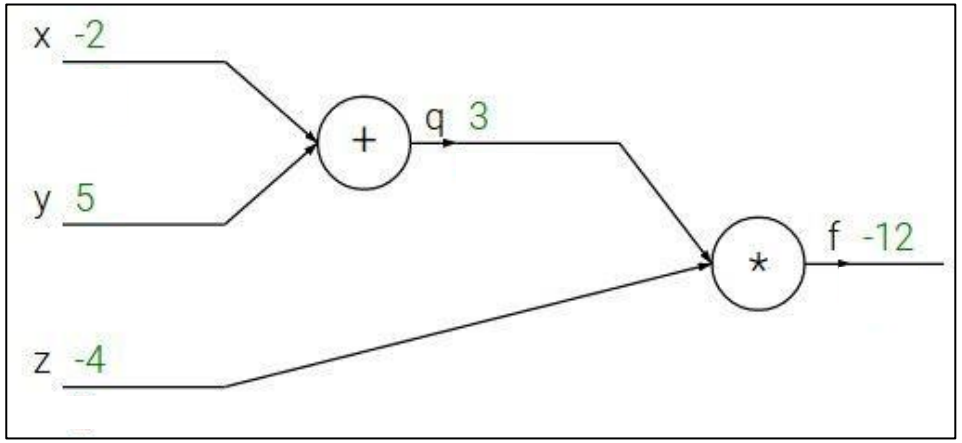
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



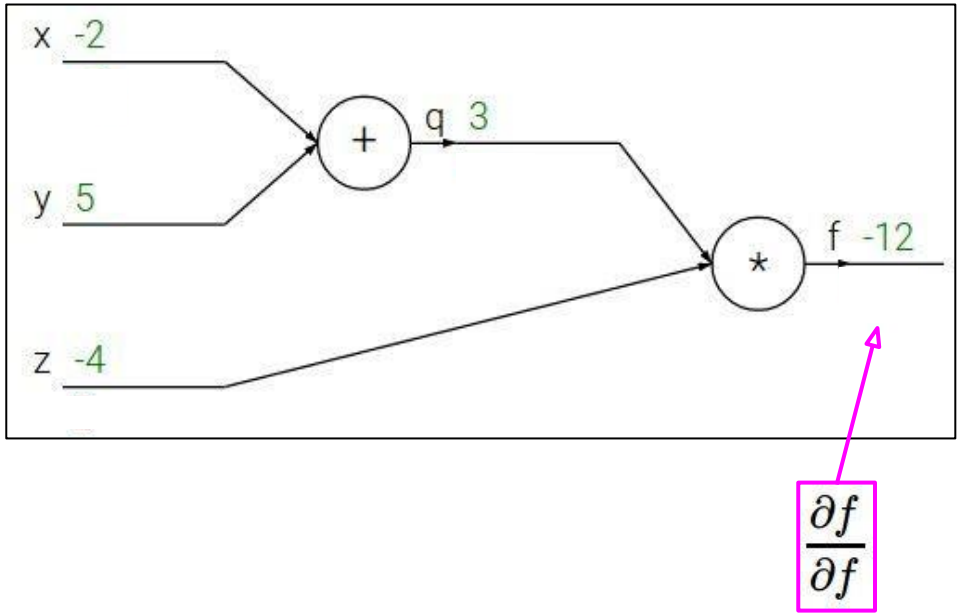
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



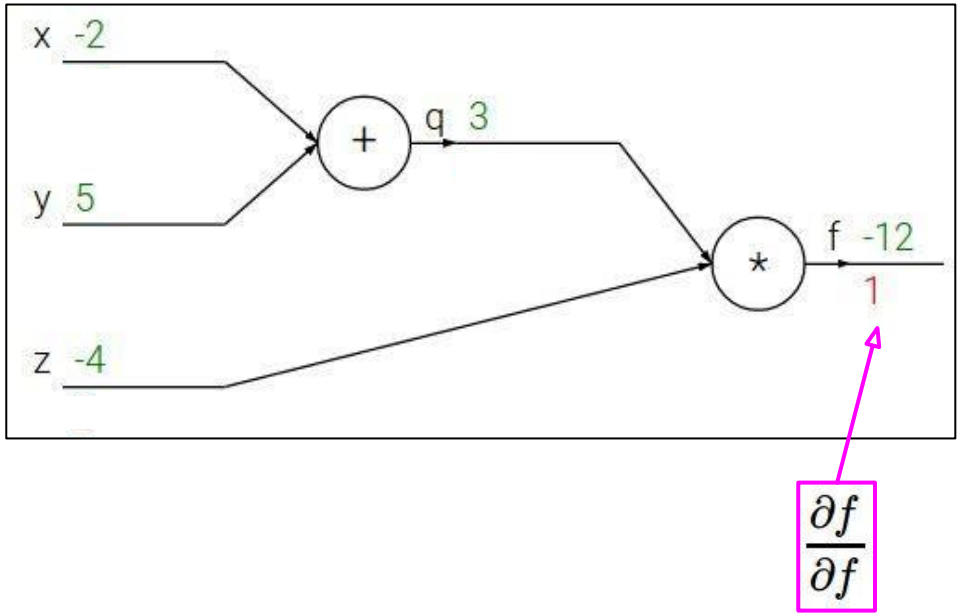
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



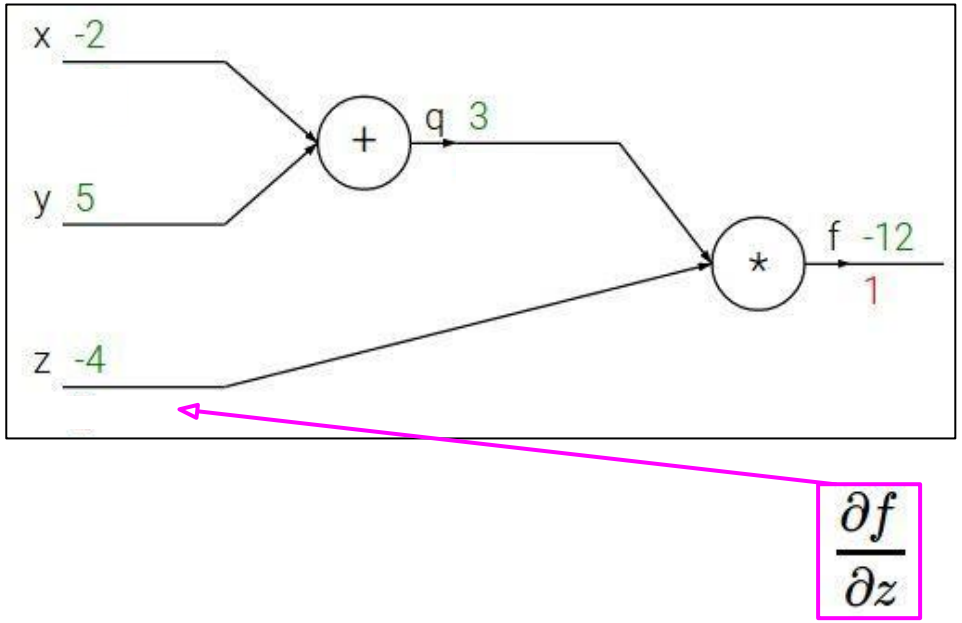
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

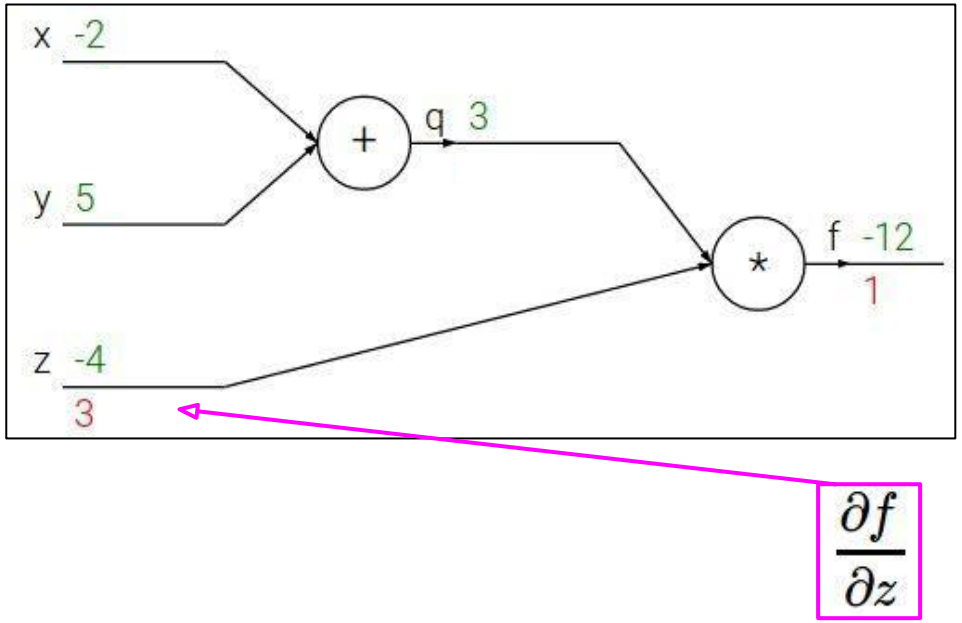


$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$



Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

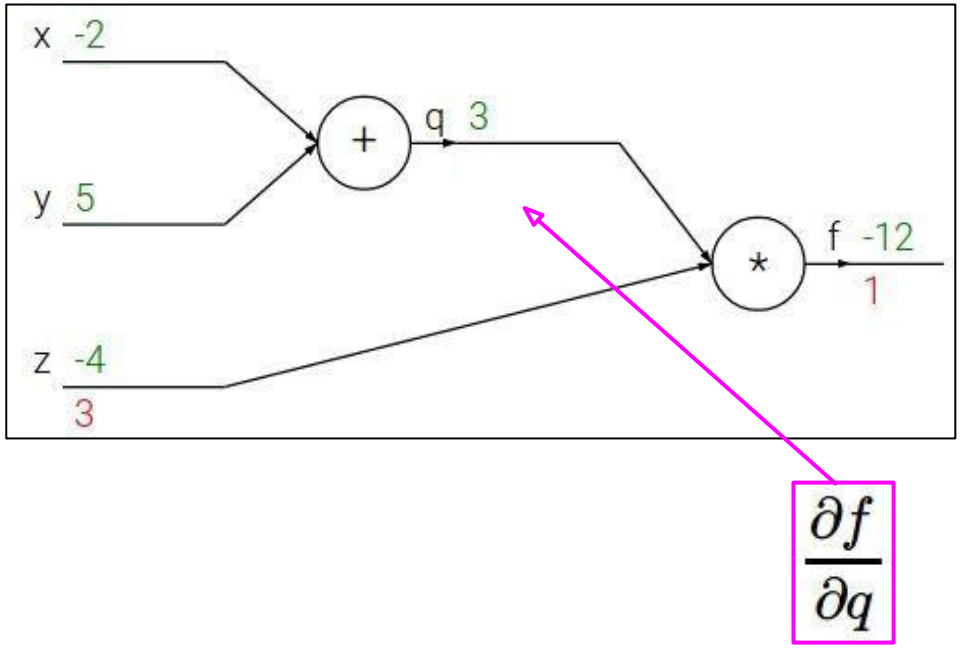
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

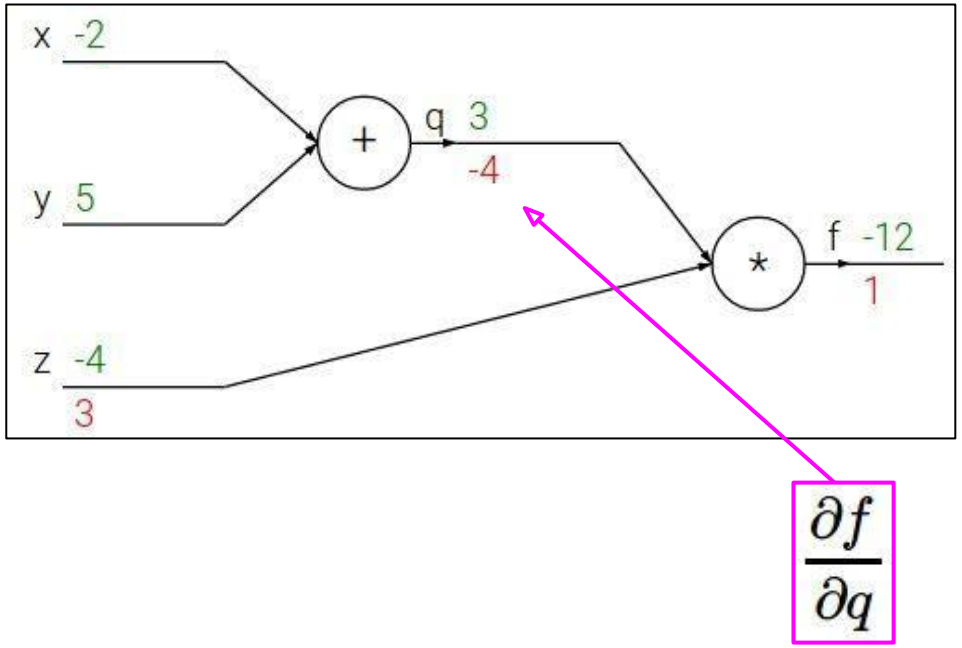


$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$



Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

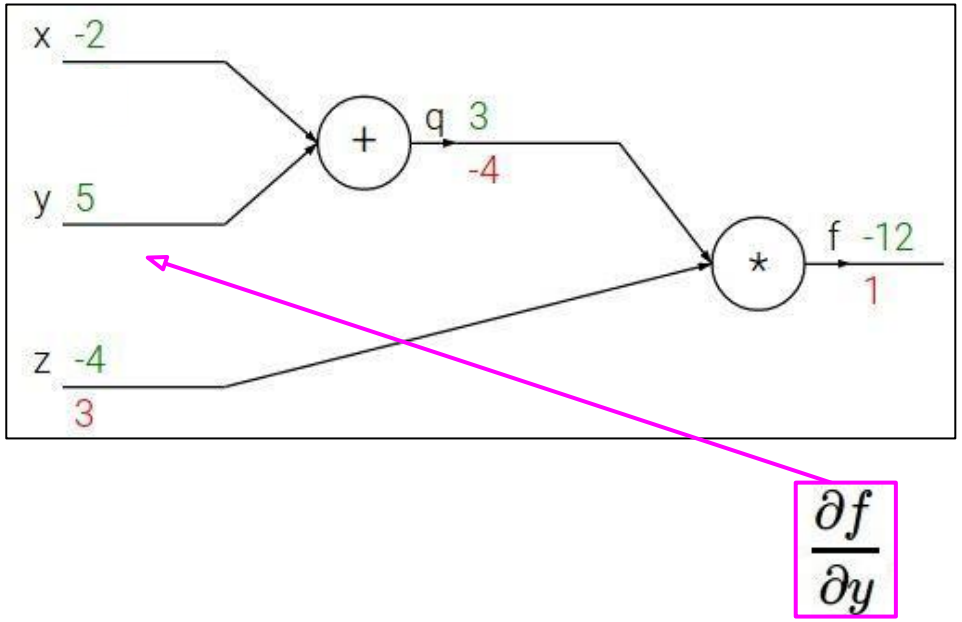
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



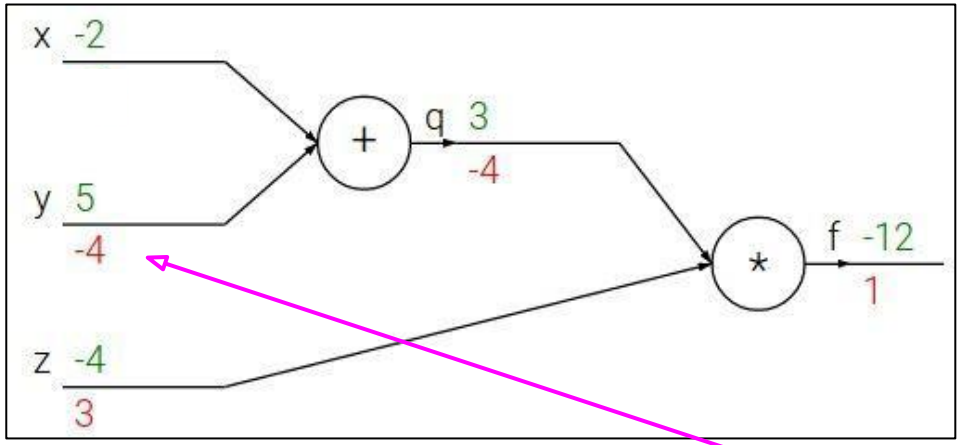
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial y}$

Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

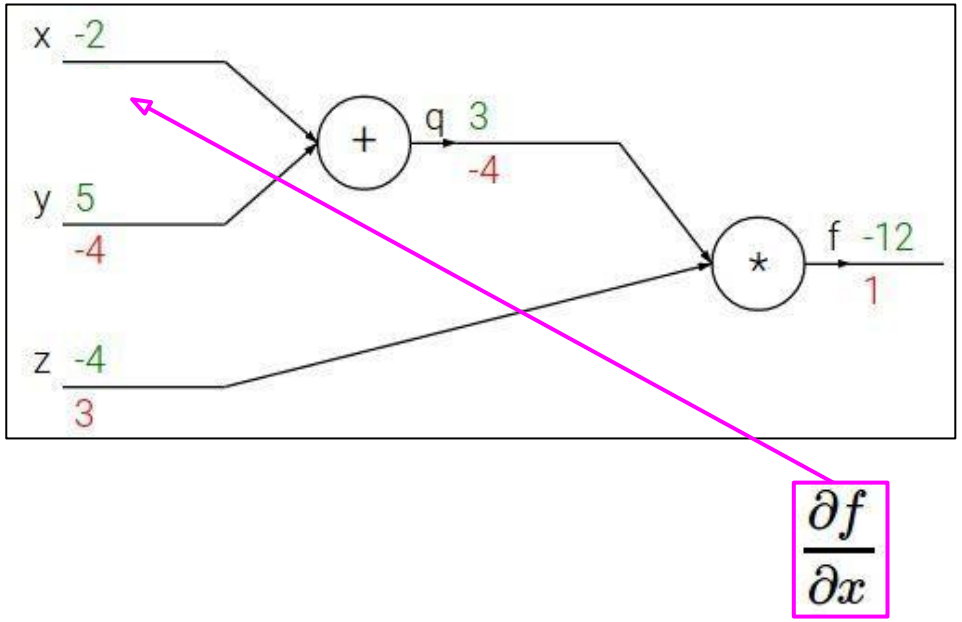
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



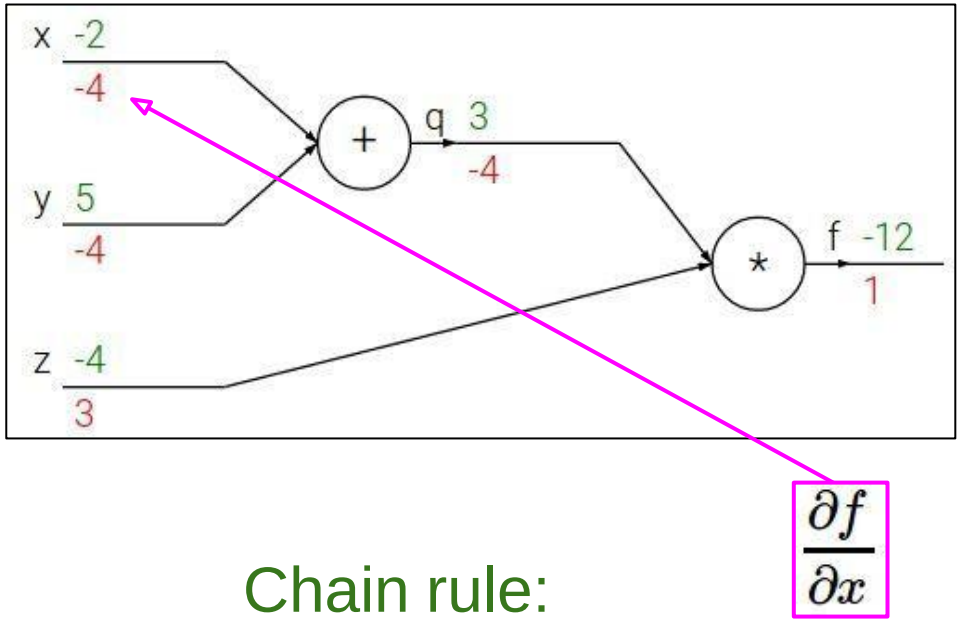
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

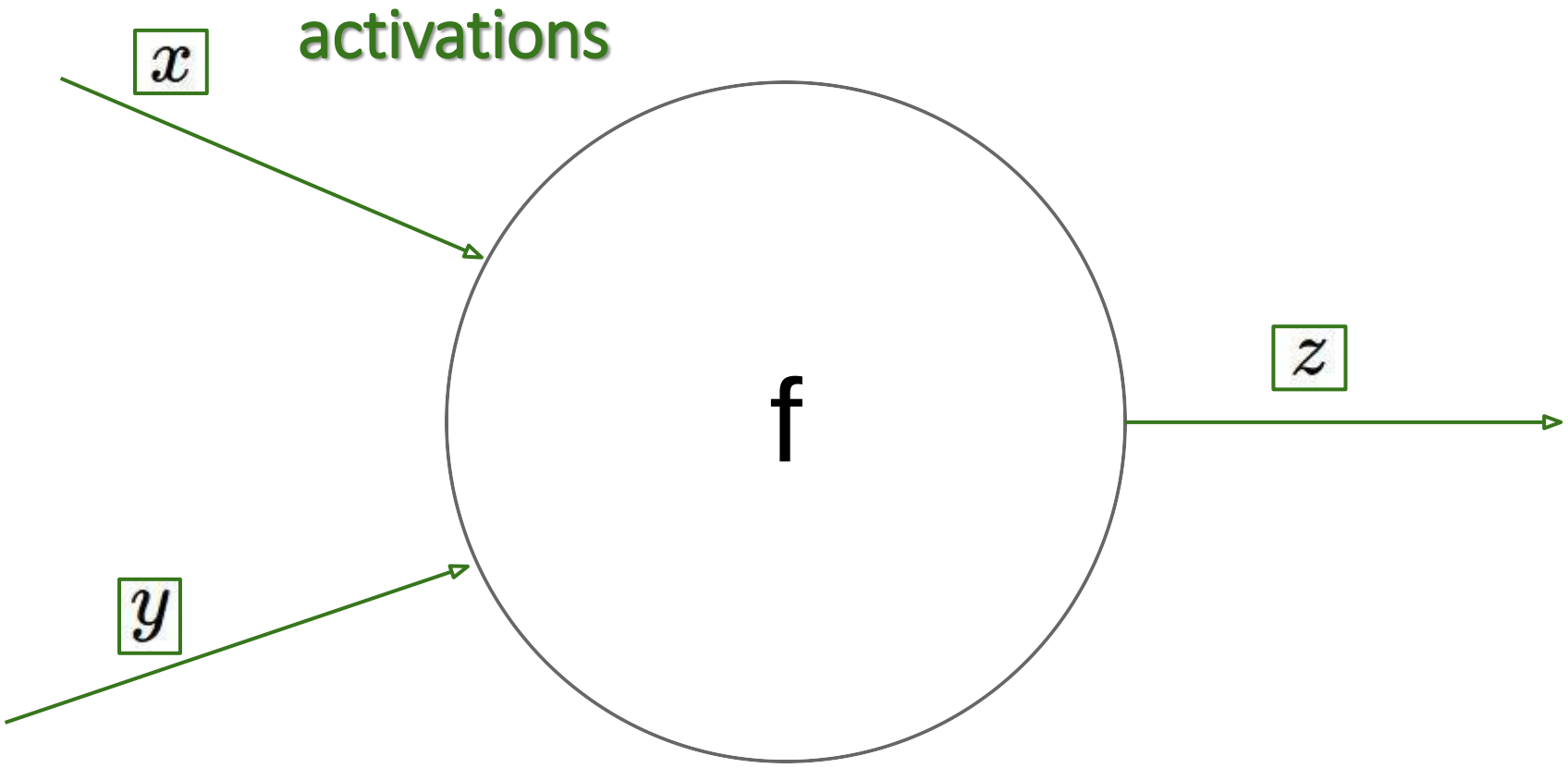
$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

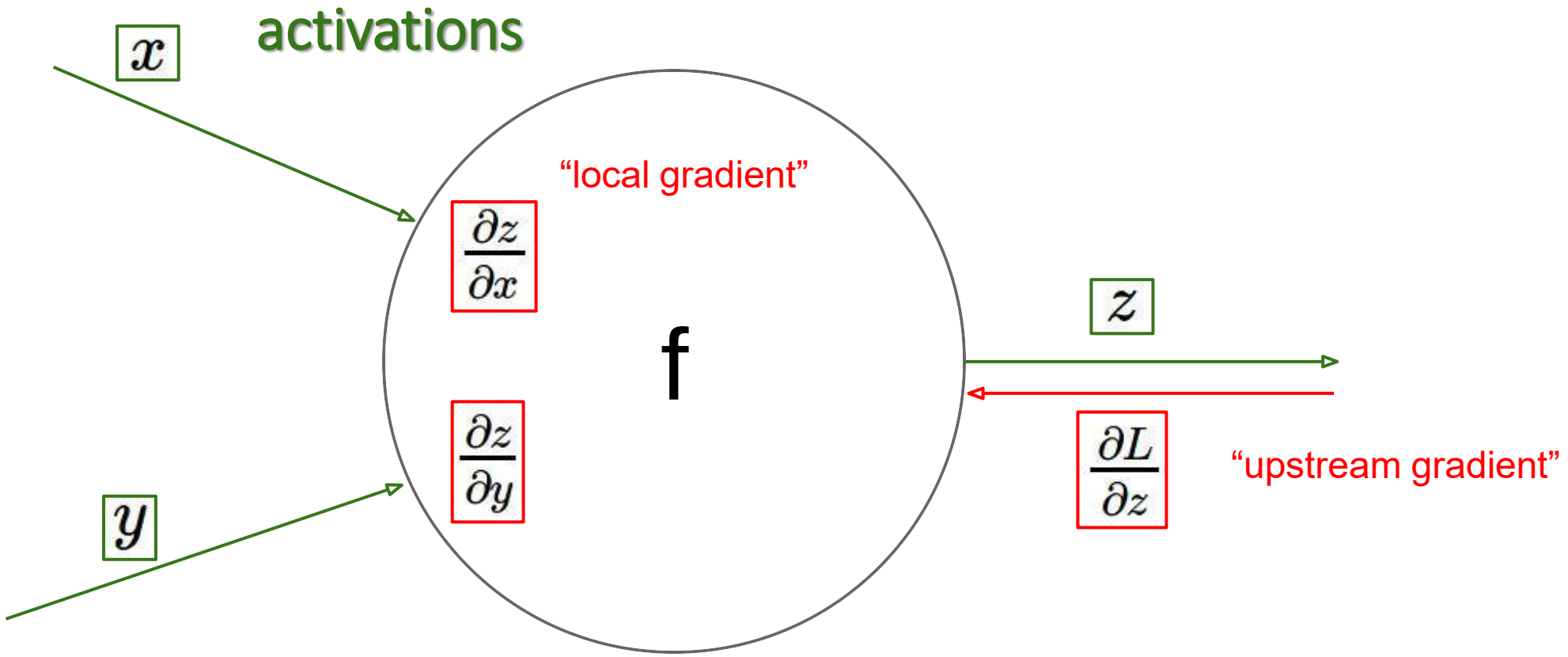
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

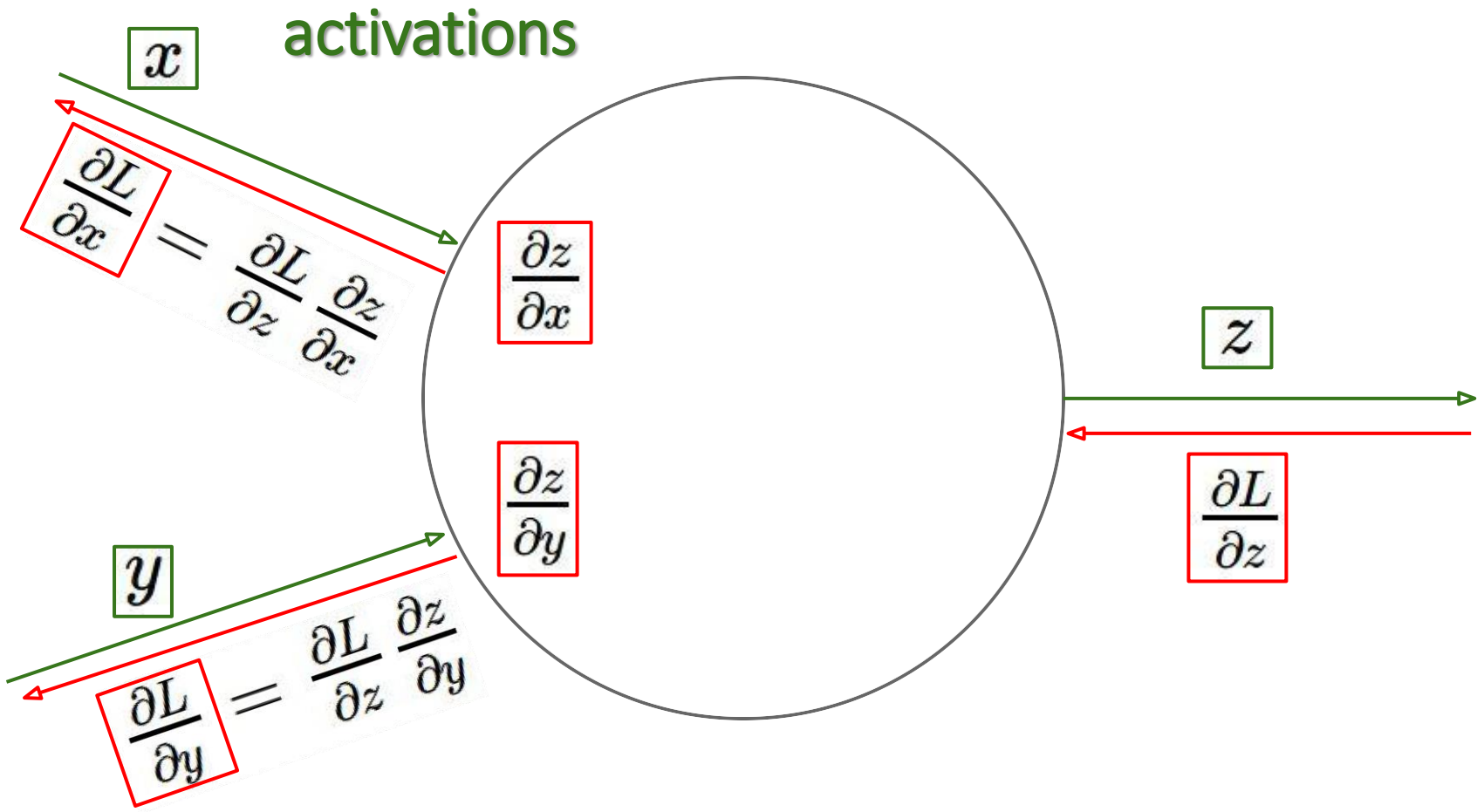


Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$


activations
x
y
f
z


activations
x
y
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
"local gradient"
f
z
$\frac{\partial L}{\partial z}$
"upstream gradient"


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

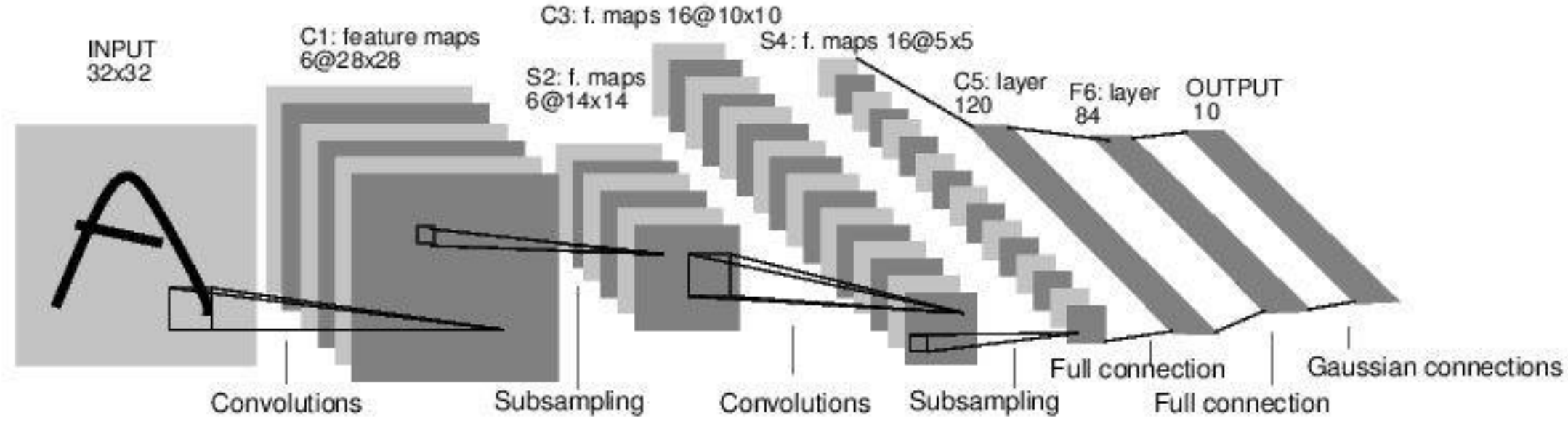
Recap

- Neural networks have many parameters
- Backpropagation = recursively apply the chain rule to compute gradients more efficiently
- Forward pass: compute the result at a neuron and save intermediate results
- Backward pass: apply the chain rule to compute the gradient of the loss wrt the input

Questions?

Case Study: LeNet-5

[LeCun et al., 1998]



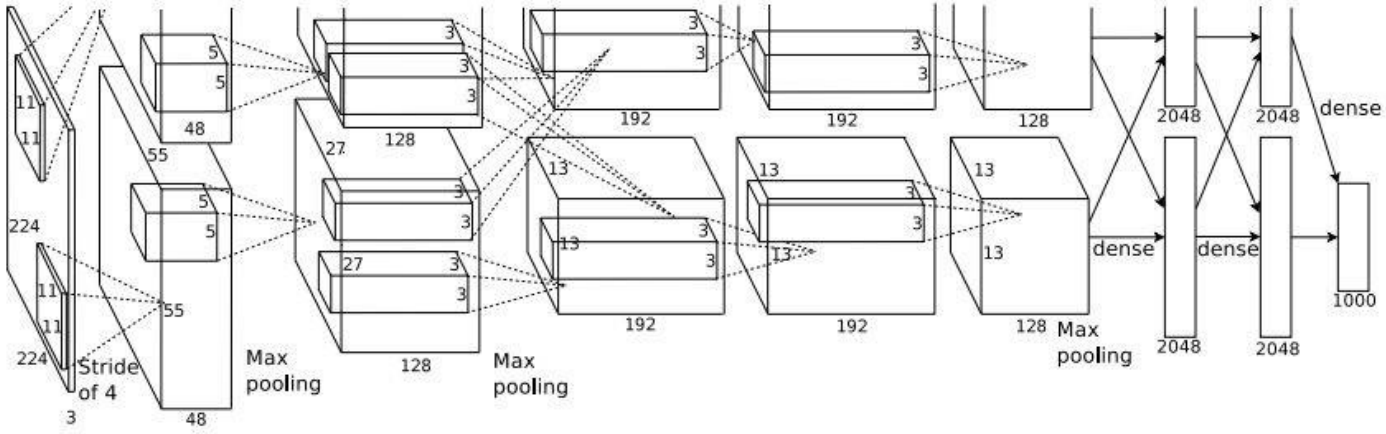
Conv filters were 5x5, applied at stride 1

Subsampling (Pooling) layers were 2x2 applied at stride 2

i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

Case Study: AlexNet

[Krizhevsky et al. 2012]

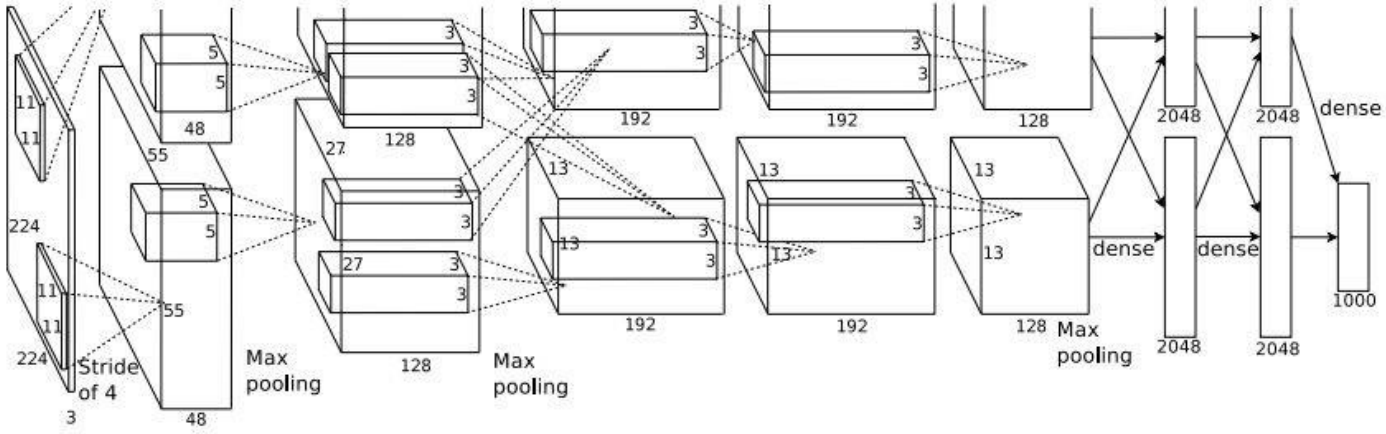


Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Q: what is the output volume size? Hint: (227-11)/4+1 = 55

Case Study: AlexNet

[Krizhevsky et al. 2012]



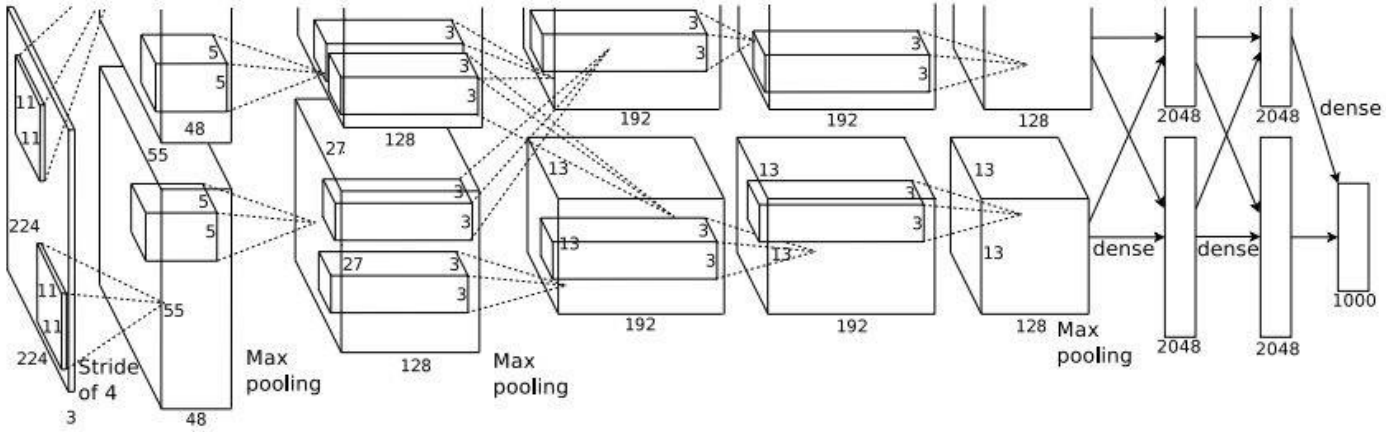
Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**

Q: What is the total number of parameters in this layer?

Case Study: AlexNet

[Krizhevsky et al. 2012]

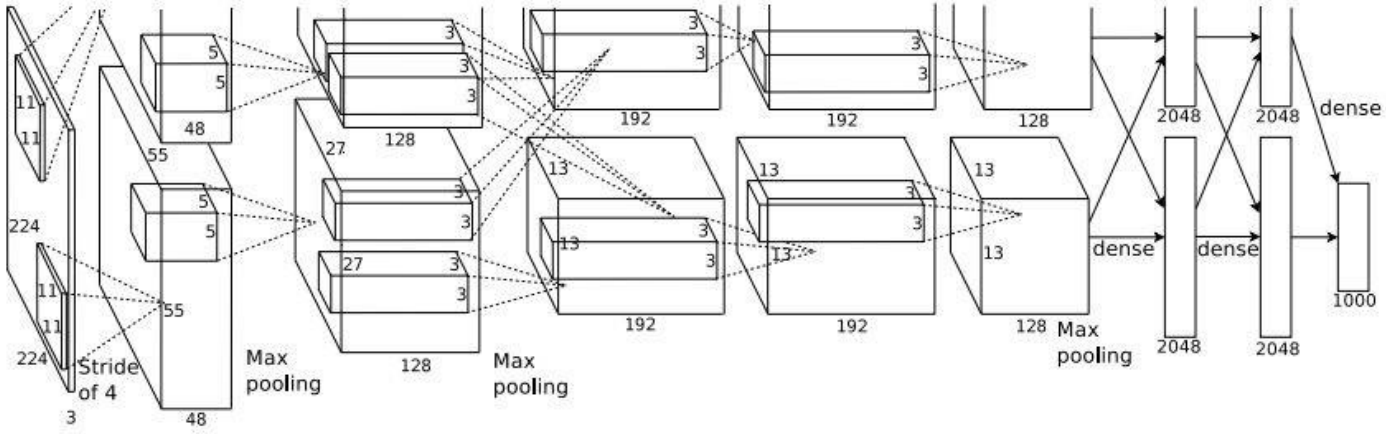


Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**
Parameters: (11*11*3)*96 = **35K**

Case Study: AlexNet

[Krizhevsky et al. 2012]



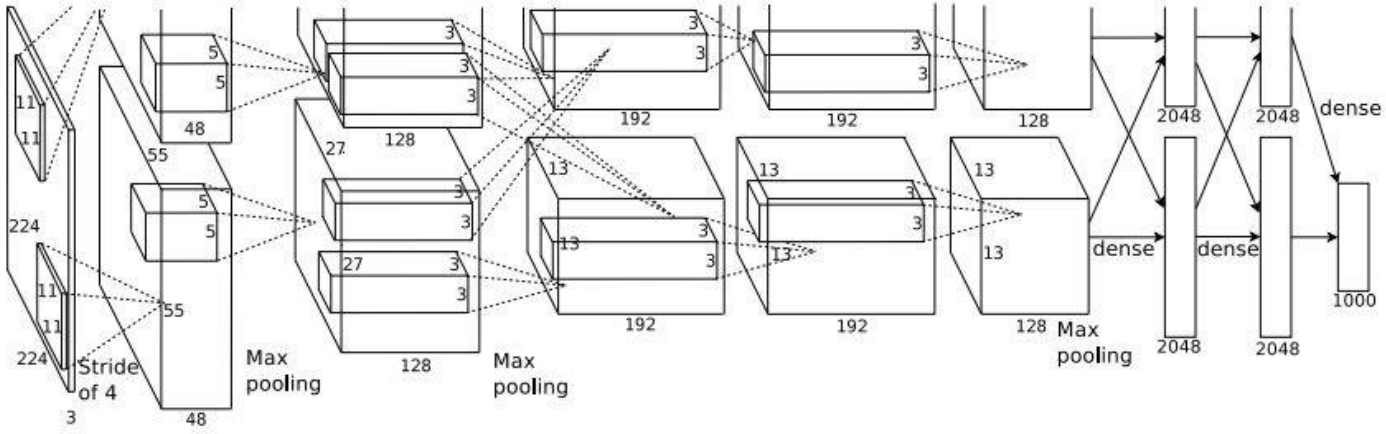
Input: 227x227x3 images After
CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2

Q: what is the output volume size? Hint: (55-3)/2+1 = 27

Case Study: AlexNet

[Krizhevsky et al. 2012]



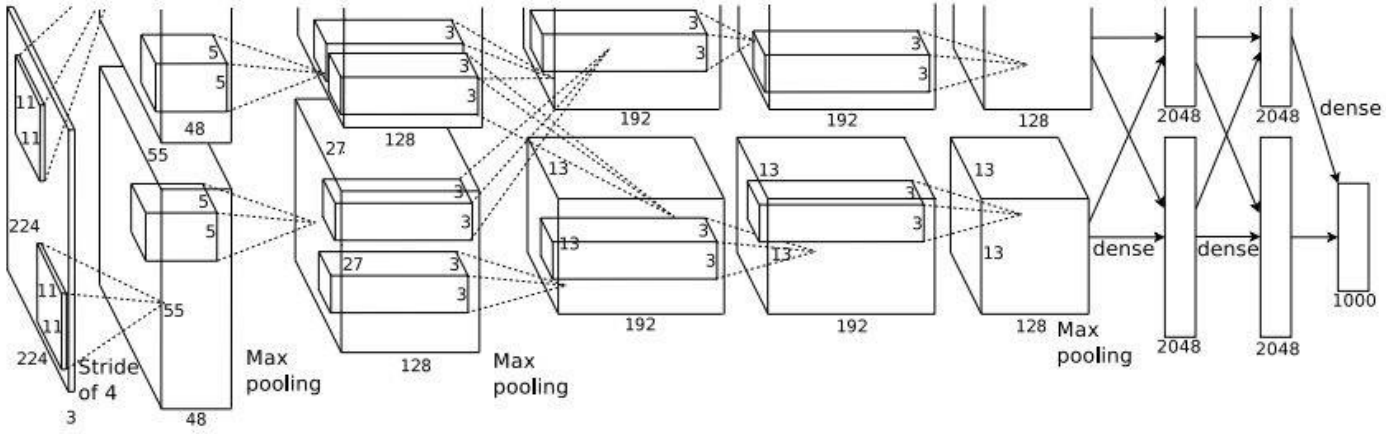
Input: 227x227x3 images After
CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96

Q: what is the number of parameters in this layer?

Case Study: AlexNet

[Krizhevsky et al. 2012]

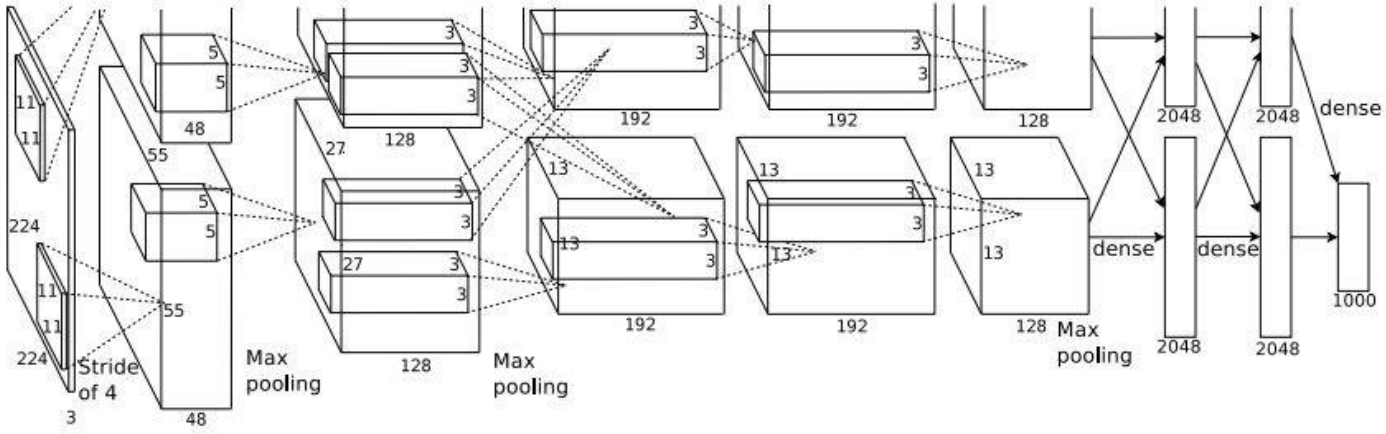


Input: 227x227x3 images After
CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96
Parameters: 0!

Case Study: AlexNet

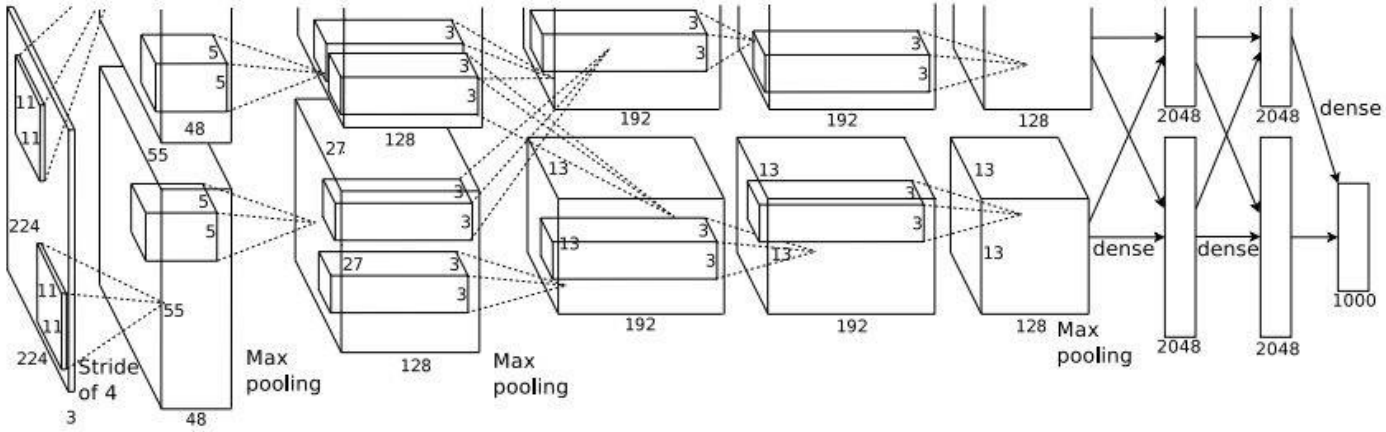
[Krizhevsky et al. 2012]



Input: 227x227x3 images After
CONV1: 55x55x96
After POOL1: 27x27x96
...

Case Study: AlexNet

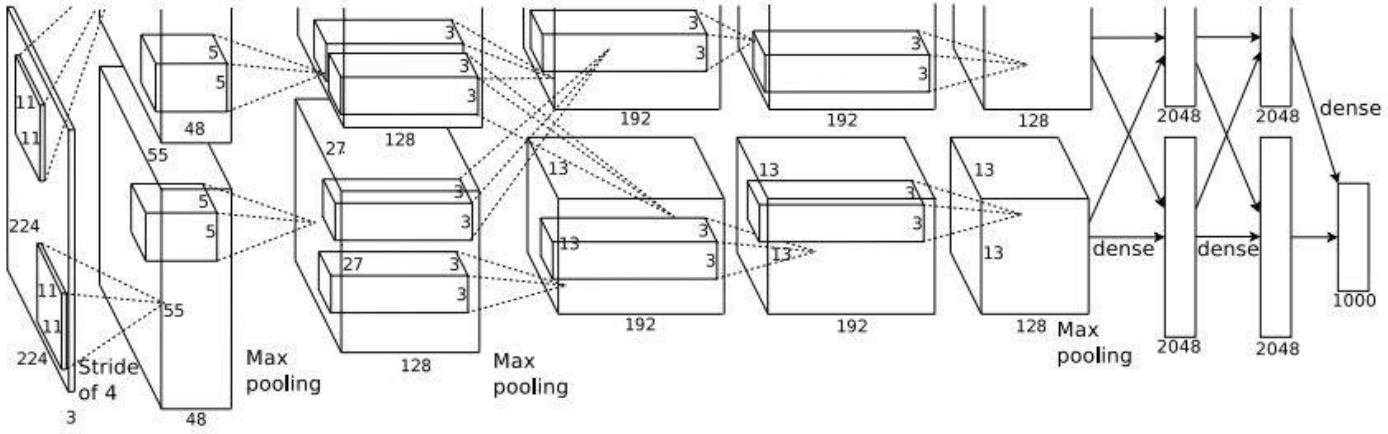
[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Case Study: AlexNet

[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
-first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
-Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

best model

11.2% top 5 error in ILSVRC 2013
->
7.3% top 5 error

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

(not counting biases)

INPUT: [224x224x3]	memory: 224*224*3=150K	params: 0
CONV3-64: [224x224x64]	memory: 224*224*64=3.2M	params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]	memory: 224*224*64=3.2M	params: (3*3*64)*64 = 36,864
POOL2: [112x112x64]	memory: 112*112*64=800K	params: 0
CONV3-128: [112x112x128]	memory: 112*112*128=1.6M	params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]	memory: 112*112*128=1.6M	params: (3*3*128)*128 = 147,456
POOL2: [56x56x128]	memory: 56*56*128=400K	params: 0
CONV3-256: [56x56x256]	memory: 56*56*256=800K	params: (3*3*128)*256 = 294,912
CONV3-256:[56x56x256]	memory:56*56*256=800K	params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]	memory: 56*56*256=800K	params: (3*3*256)*256 = 589,824
POOL2: [28x28x256]	memory: 28*28*256=200K	params: 0
CONV3-512: [28x28x512]	memory: 28*28*512=400K	params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]	memory: 28*28*512=400K	params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]	memory: 28*28*512=400K	params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]	memory: 14*14*512=100K	params: 0
CONV3-512: [14x14x512]	memory: 14*14*512=100K	params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]	memory: 14*14*512=100K	params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]	memory: 14*14*512=100K	params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]	memory: 7*7*512=25K	params: 0
FC: [1x1x4096]	memory: 4096	params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]	memory: 4096	params: 4096*4096 = 16,777,216
FC: [1x1x1000]	memory: 1000	params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

ConvNet Configuration			
B	C	D	
13 weight layers	16 weight layers	16 weight layers	19
put (224 × 224 RGB image)			
conv3-64	conv3-64	conv3-64	cc
conv3-64	conv3-64	conv3-64	cc
maxpool			
conv3-128	conv3-128	conv3-128	co
conv3-128	conv3-128	conv3-128	co
maxpool			
conv3-256	conv3-256	conv3-256	co
conv3-256	conv3-256	conv3-256	co
	conv1-256	**conv3-256**	co
			co
maxpool			
conv3-512	conv3-512	conv3-512	co
conv3-512	conv3-512	conv3-512	co
	conv1-512	**conv3-512**	co
			co
maxpool			
conv3-512	conv3-512	conv3-512	co
conv3-512	conv3-512	conv3-512	co
	conv1-512	**conv3-512**	co
			co
maxpool			
FC-4096			
FC-4096			
FC-1000			
soft-max			

(not counting biases)

INPUT: [224x224x3]	memory: 224*224*3=150K	params: 0
CONV3-64: [224x224x64]	**memory: 224*224*64=3.2M**	params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]	**memory: 224*224*64=3.2M**	params: (3*3*64)*64 = 36,864
POOL2: [112x112x64]	memory: 112*112*64=800K	params: 0
CONV3-128: [112x112x128]	memory: 112*112*128=1.6M	params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]	memory: 112*112*128=1.6M	params: (3*3*128)*128 = 147,456
POOL2: [56x56x128]	memory: 56*56*128=400K	params: 0
CONV3-256: [56x56x256]	memory: 56*56*256=800K	params: (3*3*128)*256 = 294,912
CONV3-256:[56x56x256]	memory:56*56*256=800K	params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]	memory: 56*56*256=800K	params: (3*3*256)*256 = 589,824
POOL2: [28x28x256]	memory: 28*28*256=200K	params: 0
CONV3-512: [28x28x512]	memory: 28*28*512=400K	params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]	memory: 28*28*512=400K	params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]	memory: 28*28*512=400K	params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]	memory: 14*14*512=100K	params: 0
CONV3-512: [14x14x512]	memory: 14*14*512=100K	params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]	memory: 14*14*512=100K	params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]	memory: 14*14*512=100K	params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]	memory: 7*7*512=25K	params: 0
FC: [1x1x4096]	memory: 4096	**params: 7*7*512*4096 = 102,760,448**
FC: [1x1x4096]	memory: 4096	params: 4096*4096 = 16,777,216
FC: [1x1x1000]	memory: 1000	params: 4096*1000 = 4,096,000

Most memory is in early CONV

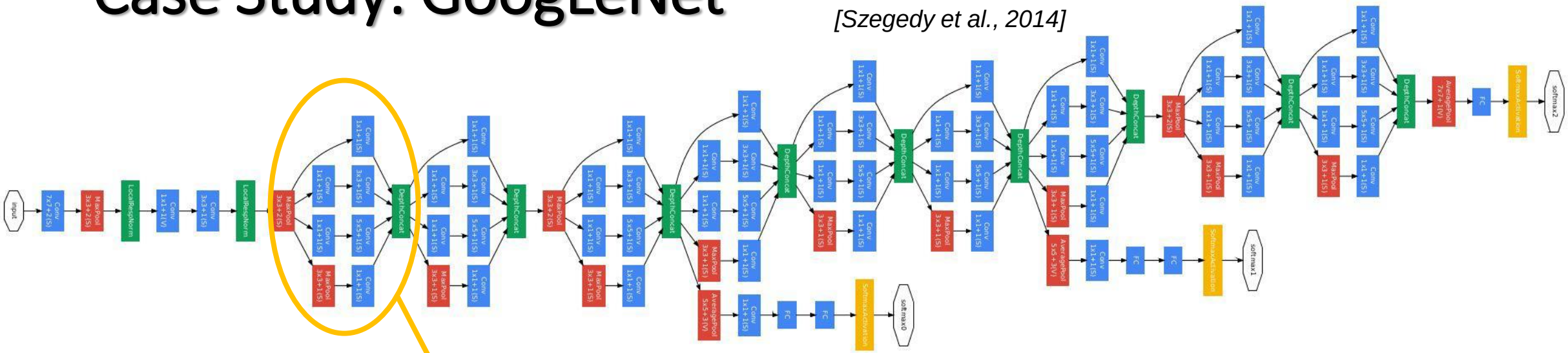
Most params are in late FC

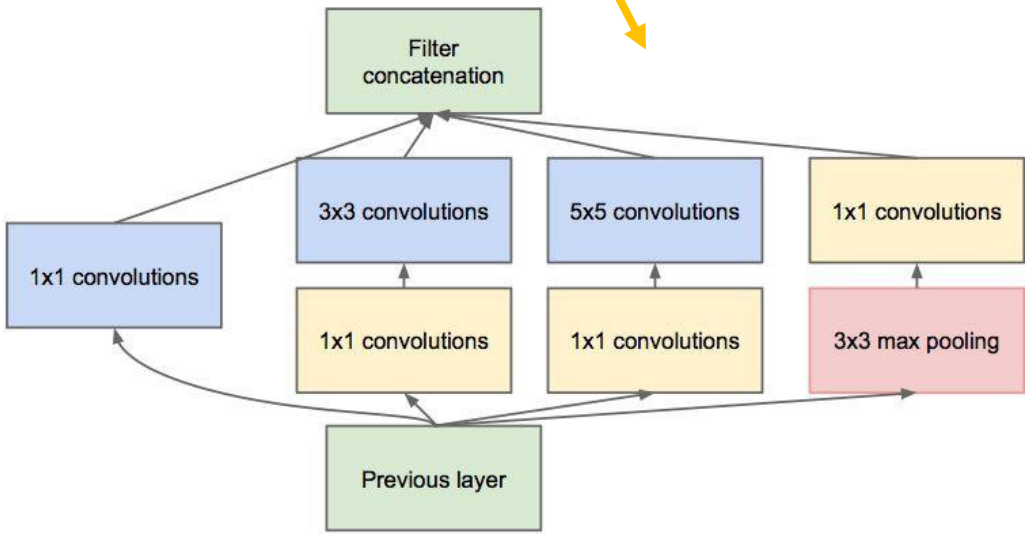
TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)

TOTAL params: 138M parameters

Case Study: GoogLeNet

[Szegedy et al., 2014]





Inception module

ILSVRC 2014 winner (6.7% top 5 error)

Case Study: GoogLeNet

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

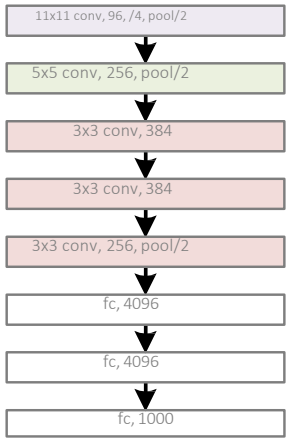
Fun features:

-Only 5 million params! (Removes FC layers completely)

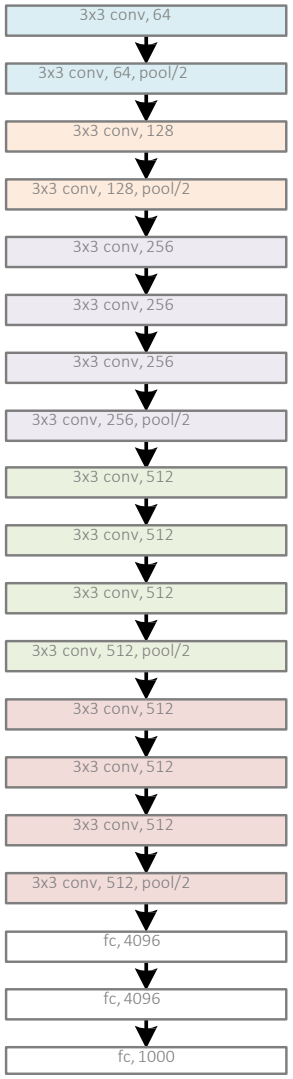
Compared to AlexNet:
- 12X less params
- 2x more compute
- 6.67% (vs. 16.4%)

Revolution of Depth

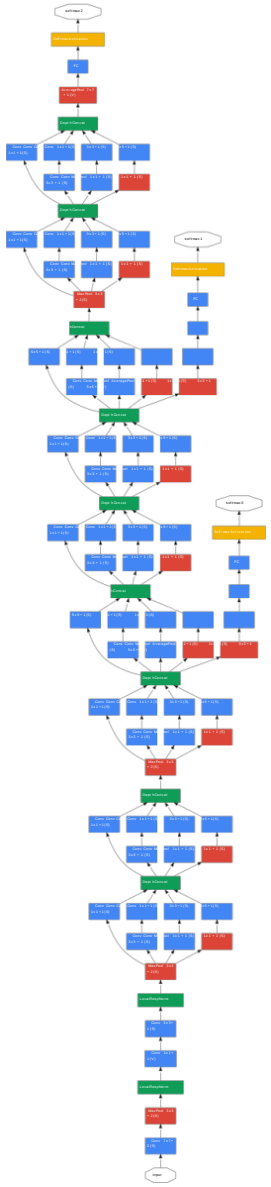
AlexNet, 8 layers
(ILSVRC 2012)



VGG, 19 layers
(ILSVRC 2014)



GoogleNet, 22 layers
(ILSVRC 2014)

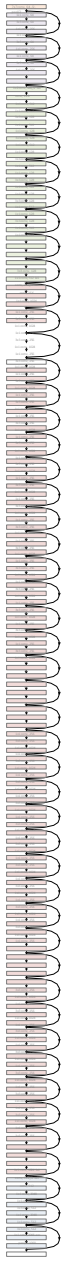


Revolution of Depth

AlexNet, 8 layers
(ILSVRC 2012)

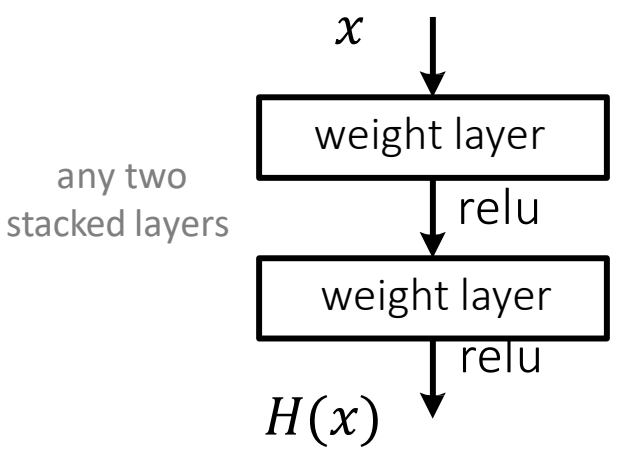
VGG, 19 layers
(ILSVRC 2014)

ResNet, 152 layers
(ILSVRC 2015)



Deep Residual Learning

- Plaint net

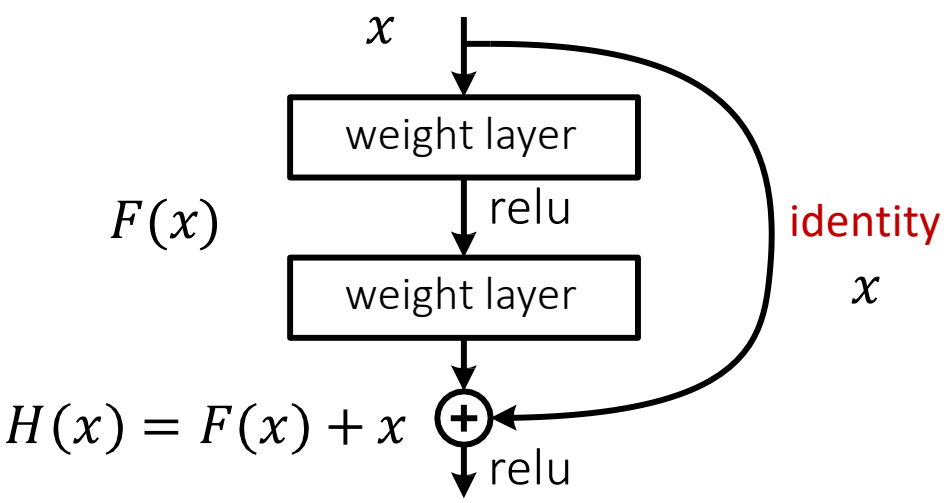


$H(x)$ is any desired mapping,

hope the 2 weight layers fit $H(x)$

Deep Residual Learning

- Residual net



$H(x)$ is any desired mapping,

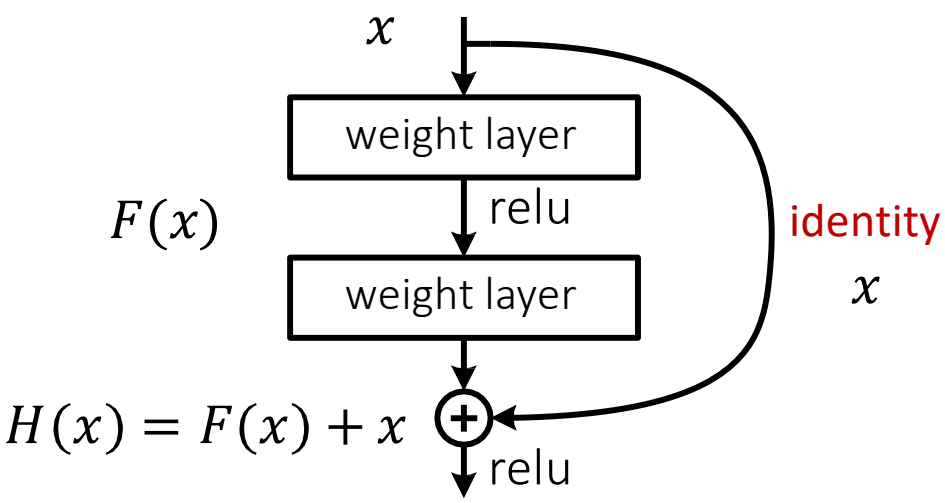
~~hope the 2 weight layers fit $H(x)$~~

hope the 2 weight layers fit $F(x)$

let $H(x) = F(x) + x$

Deep Residual Learning

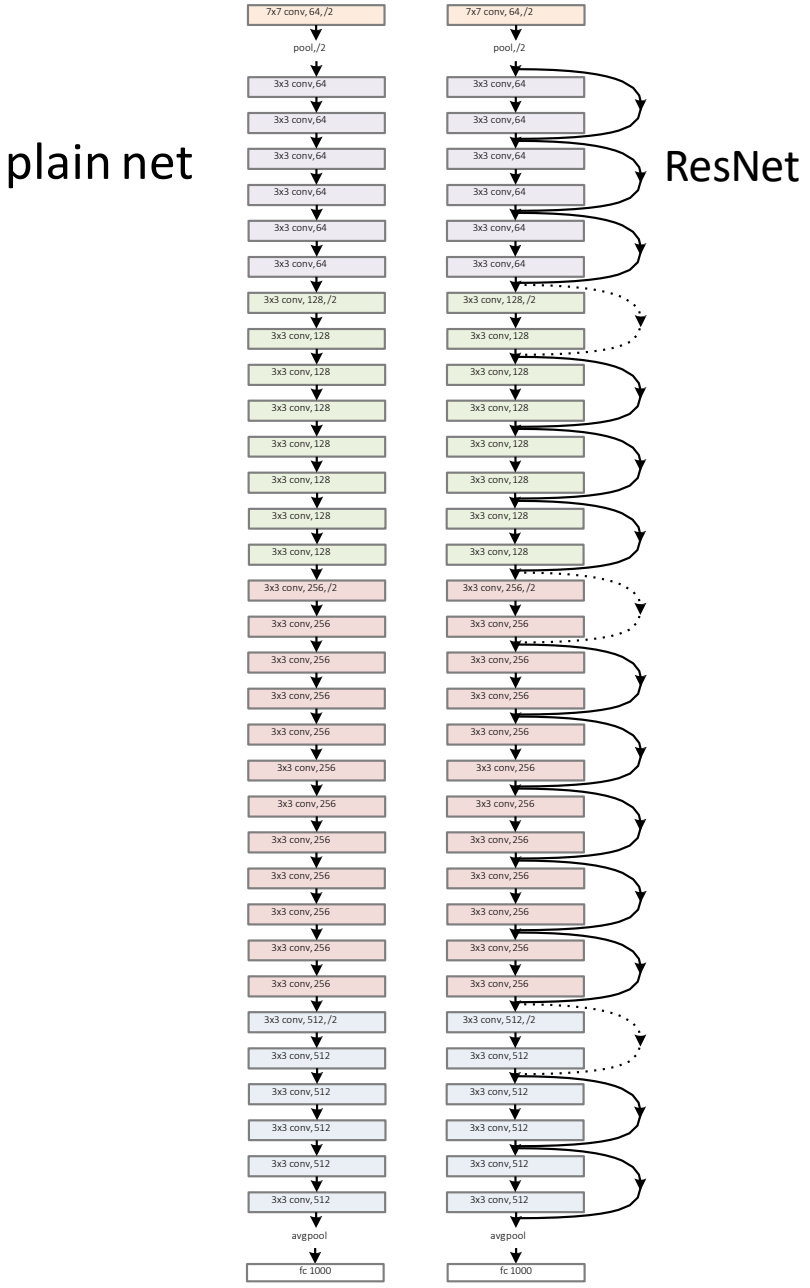
- $F(x)$ is a residual mapping w.r.t. identity

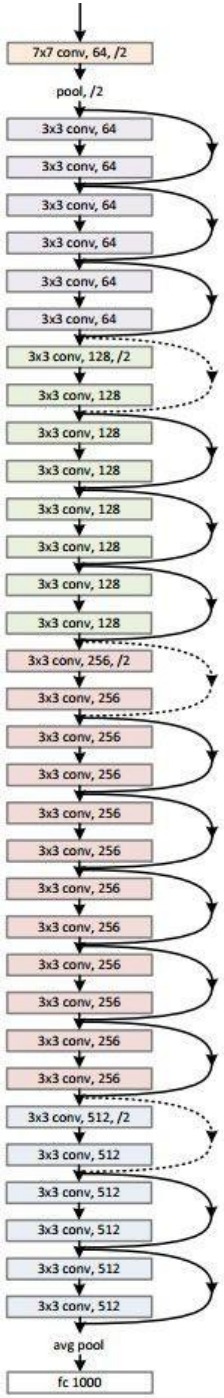


- If identity were optimal, easy to set weights as 0
- If optimal mapping is closer to identity, easier to find small fluctuations

Network “Design”

- Keep it simple
- Basic design (VGG-style)
  - all 3x3 conv (almost)
  - spatial size /2 => # filters x2
  - Simple design; just deep!





Case Study: ResNet *[He et al., 2015]*

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

FLOP = floating-point operation

Recap

- Convolutional Networks stack CONV, POOL, FC layers
- Typical architectures (AlexNet) look like

  [(CONV – RELU)*N – POOL]*M – (FC – RELU)*K, SOFTMAX

  - N typically up to 5, M is large, K can be 0, 1, or 2
  - More recent networks (ResNet/DenseNet/GoogleNet) change this paradigm

- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)

Questions?

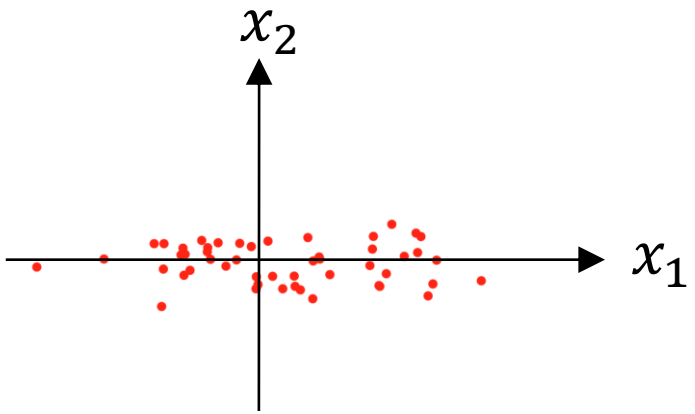
Training Neural Networks Involves Many Tweaking

- Before you start:
  - Data preprocessing
  - Parameter initialization
- Tricks in training
  - Regularization
  - Dropout
  - Batch Normalization
  - Momentum, RMSprop, Adam, Learning rate decay
- Tuning
  - Hyperparameter tuning
  - Data augmentation
  - Transfer learning

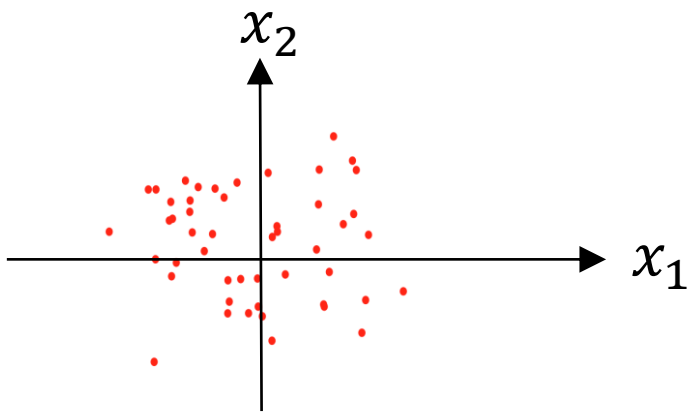
Data Preprocessing

original input data

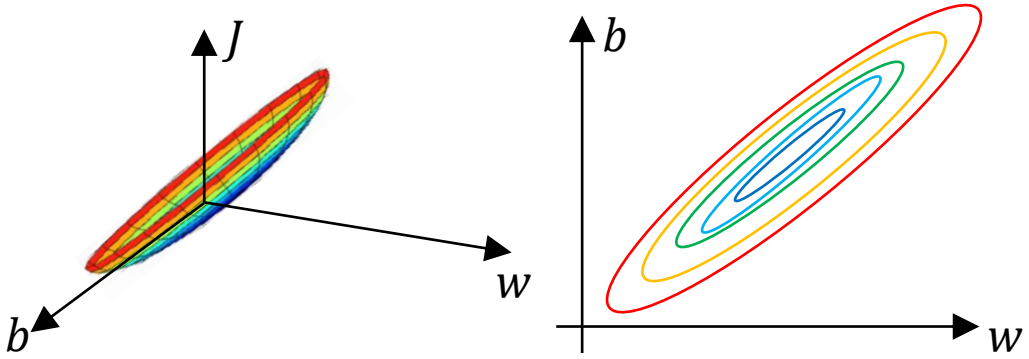
zero-centered data



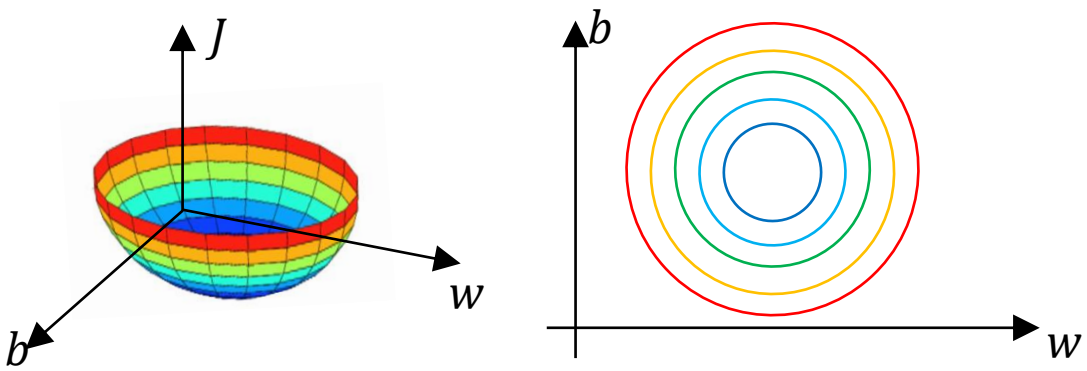
normalized data



Loss on Unnormalized data



Loss on Normalized data



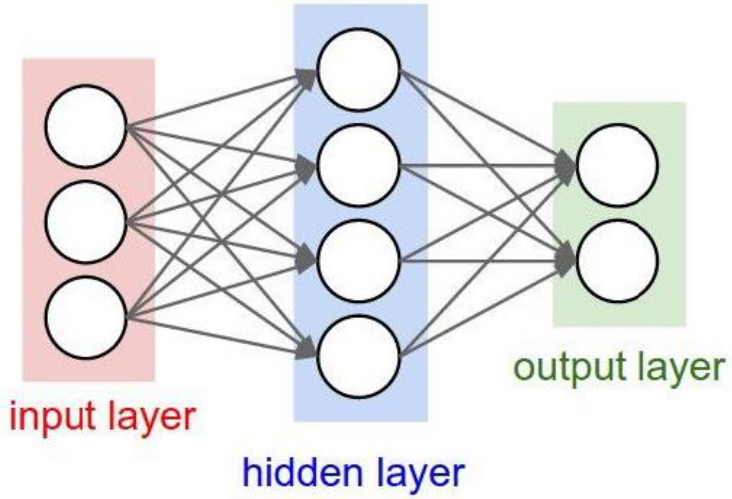
Data Preprocessing

- In practice for images: center only
  - Subtract the mean image (e.g. AlexNet)
    mean image = an [32, 32, 3] image
  - Subtract per-channel mean value (e.g. VGGNet)
    mean along each channel = 3 numbers

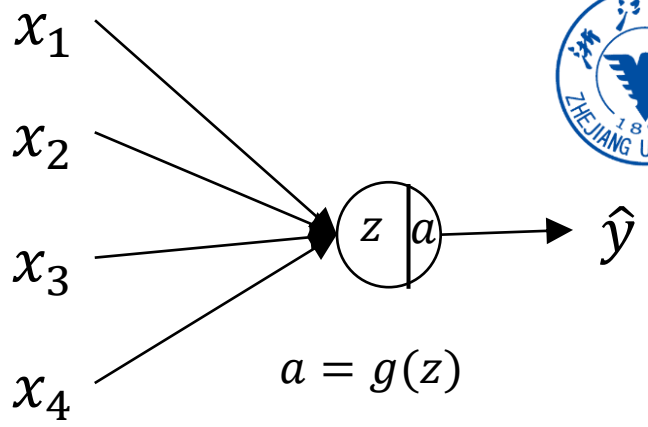
Not common to normalize variance, to do PCA or whitening, etc

Parameter Initialization

- Gradient descent requires an initial value W_0
- How do we initialize the weights in a neural network??



Parameter Initialization



- Examine a single neuron

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

  - When n is large, w_i should be smaller
  - So that z is kept at a stable numerical scale
- Suppose we want to keep $var(z) = 1$
  - If x_i has a variance of unit 1
  - $var(w_i^{[l]})$ should be $\sqrt{1/n^{[l-1]}}$ → Xavier initialization, used for the $tanh$ nonlinear response
  - Or $var(w_i)$ is set to $\sqrt{2/n^{[l-1]}}$ → Used for ReLu response, [He et al. 2015] (Delving Deep into Rectifiers)

Training Neural Networks Involves Many Tweaking

- Before you start:
  - Data preprocessing
  - Parameter initialization
- Tricks in training
  - Regularization
  - Dropout
  - Batch Normalization
  - Momentum, RMSprop, Adam, Learning rate decay
- Tuning
  - Hyperparameter tuning
  - Data augmentation
  - Transfer learning

Regularization

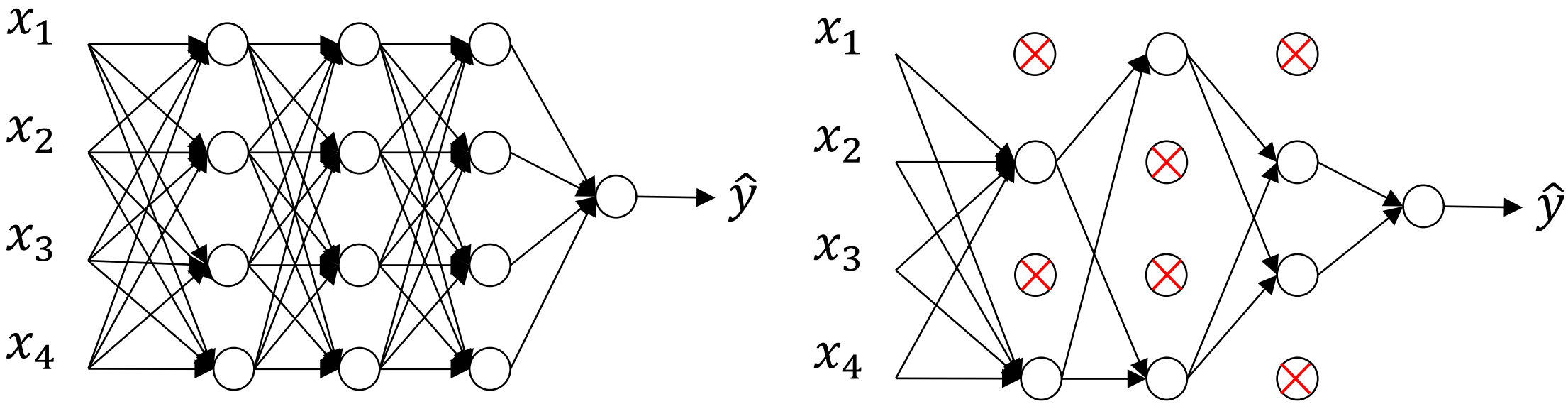
$$W^* = \arg\min_{\mathrm{W}} \sum_i L_i(y_i, \mathrm{s}_i) + \lambda R(W)$$

- L_2 regularization: $R(W) = \sum w_j^2 = W^T W$
- L_1 regularization: $R(W) = \sum |w_j|$
- In back propagation, the gradient is modified:

$$dw = (\text{gradient from back prop}) + 2\lambda W$$

Dropout

- In the forward pass, randomly set some neurons to zero
- The probability of dropping is a hyperparameter, $drop\_prob$, e.g. 0.5
  - Looks like crazy! But it helps!



Implementing Dropout (Inverted Dropout)

- At a layer l, compute a random number between [0,1] for each node

$$d^{[l]} = np.random(a^{[l]}.shape)$$

- If the random number < $keep\_prob$ (e.g. 0.8), do the convolution

$$d^{[l]} = (d^{[l]} < keep\_prob)$$
$$a^{[l]} = a^{[l]} * d^{[l]}$$

in both forward & backward passes

- Rescale the remaining notes by $keep\_prob$

$$a^{[l]} = a^{[l]} / keep\_prob$$

No change to the network in testing time.

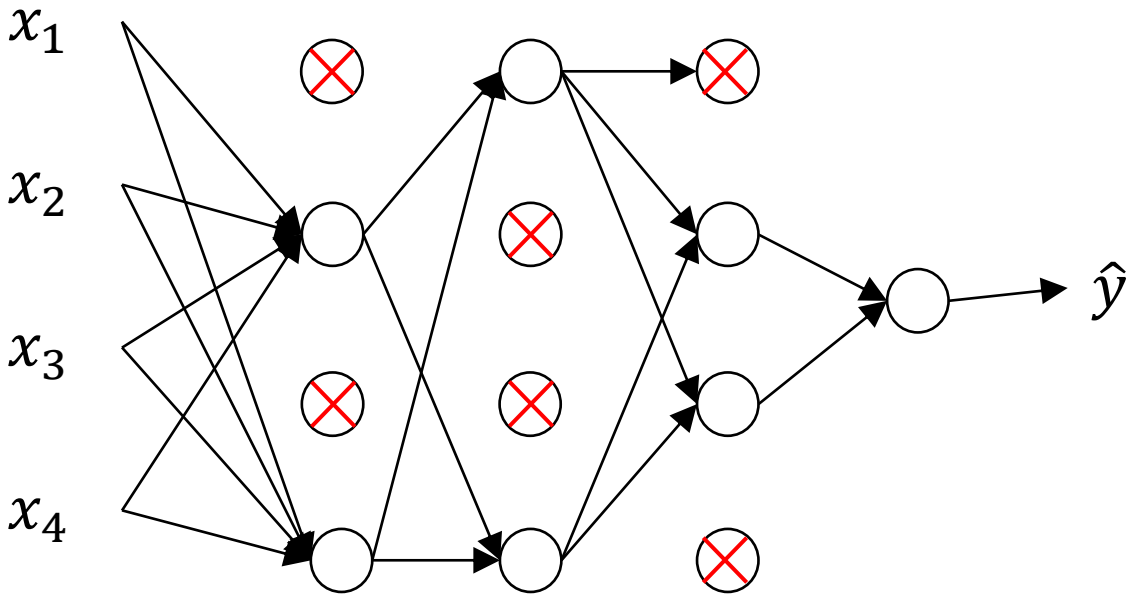
  - Why? $z^{[l+1]} = w^{[l]} * a^{[l]}$

20% of the a is set to zero, so $/0.8$ to get z at the right scale

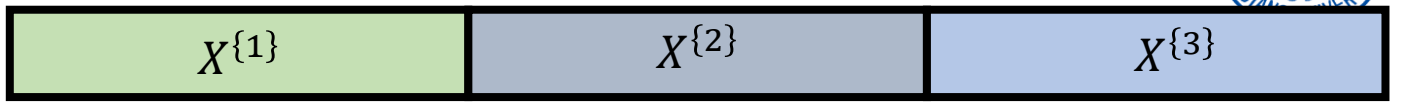
Why Dropout Helps

- Explanation 1: cannot rely on any particular feature, so have to spread out weights (making learning more stable)

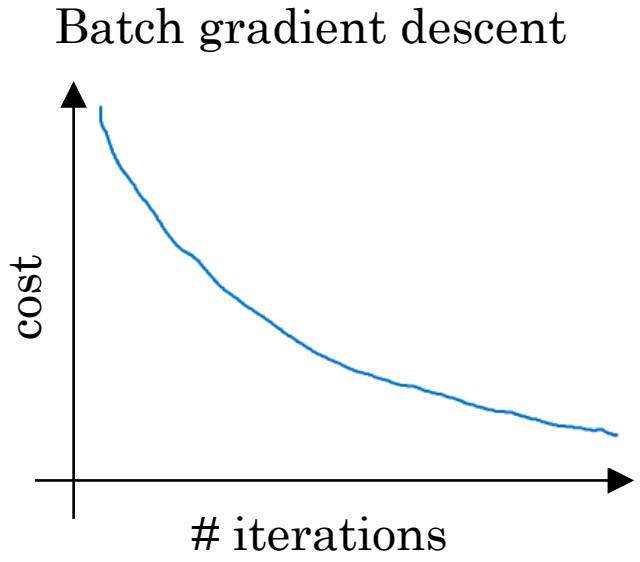
- Explanation 2: learning multiple smaller networks (regularization to avoid overfitting)

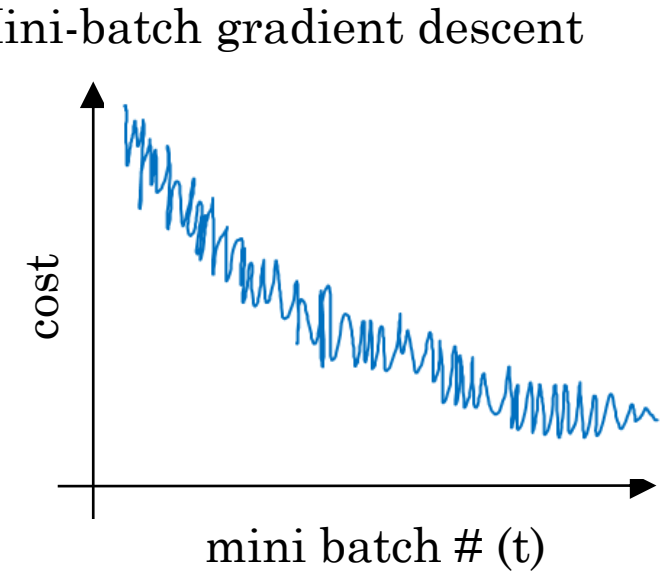


SGD with Mini Batch



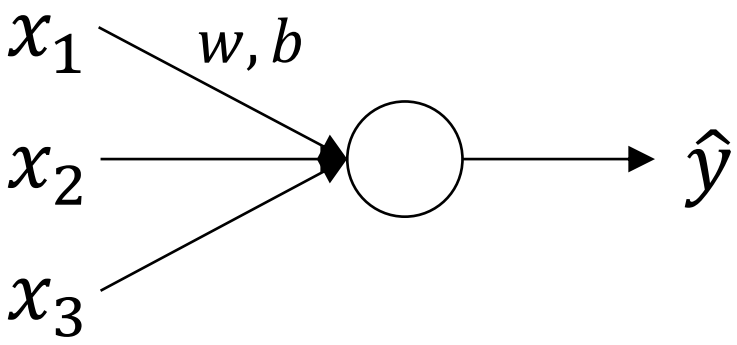
- Instead of using all training data, use a subset of samples at a time
- As a result, the loss function does not always go down at each step





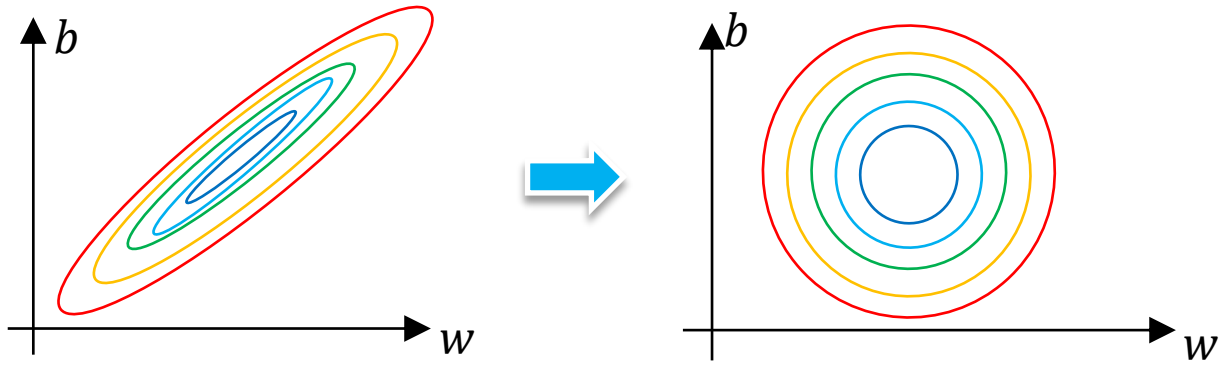
- Choose the number of samples in a mini batch
  - All data can fit in the memory of GPU/CPU
  - Try different sizes and choose the one makes training fast

Batch Normalization

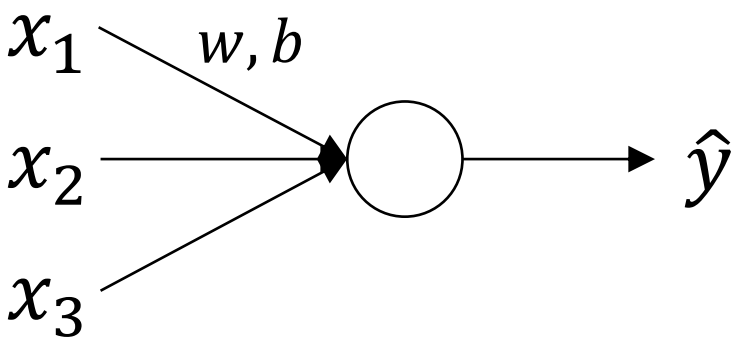


- Input date normalization:
  - Compute the mean μ and variance σ^2 of input data

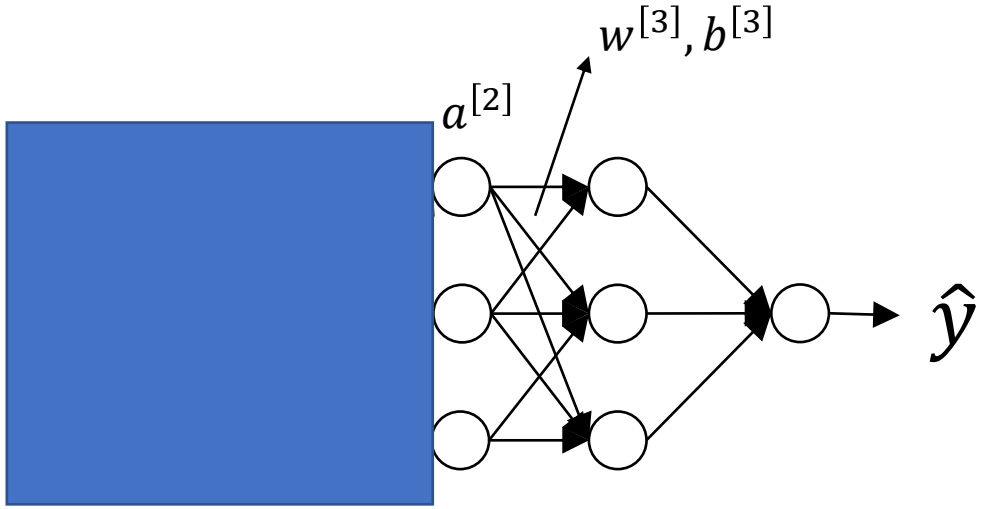
$$x^{(i)} = x^{(i)} - \mu$$
$$x^{(i)} = x^{(i)} / \sigma$$



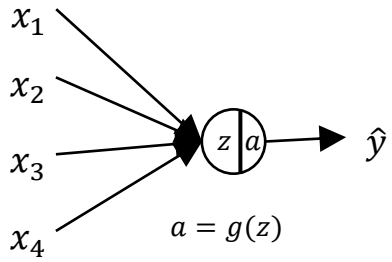
Batch Normalization



- In a deeper model, say we want to train the parameters in the 3rd layer, $w^{[3]}, b^{[3]}$
- We can apply a similar normalization to the input of the 3rd layer, i.e. $a^{[2]}$
  - So that the training of $w^{[3]}, b^{[3]}$ will be faster



There are debates about whether we should normalize $a^{[2]}$ or $z^{[2]}$. But most of people normalize $z^{[2]}$ in practice.



Implementing Batch Normalization

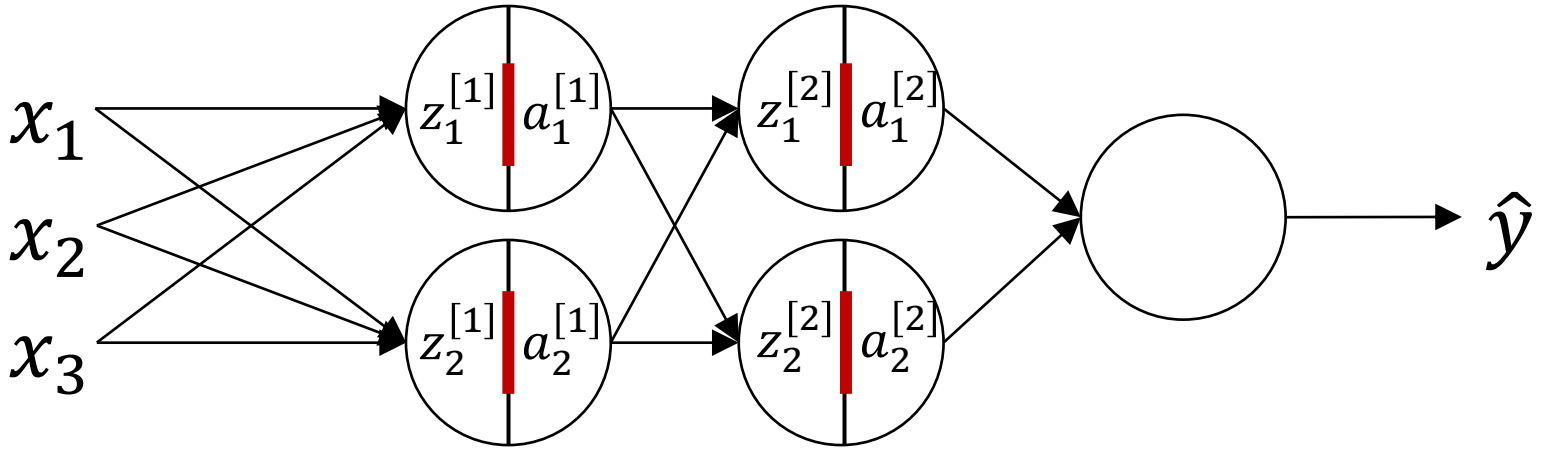
- Given some intermediate values in NN, $z^{[l](1)}, z^{[l](2)}, \ldots, z^{[l](M)}$
- Compute the mean and variance, $\mu^{[l]}, {\sigma^{[l]}}^2$
- Normalize them as: $z^{[l](i)} = (z^{[l](i)} - \mu)/(\sigma + \epsilon)$
- Two additional parameters to control the new mean and variance:

$$\tilde{z}^{[l](i)} = \gamma^{[l]} z^{[l](i)} + \beta^{[l]}$$

parameters to be learned

- Use $\tilde{z}^{[l]}$ instead of $z^{[l]}$

Implementing Batch Normalization



$$x \xrightarrow{w^{[1]}, b^{[1]}} z[1] \xrightarrow[\text{Batch Normalization}]{\gamma^{[1]}, \beta^{[1]}} \tilde{z}[1] \longrightarrow a[1] \xrightarrow{w^{[2]}, b^{[2]}} z[2] \xrightarrow[\text{Batch Normalization}]{\gamma^{[2]}, \beta^{[2]}} \tilde{z}[2] \longrightarrow a[2]$$

- Parameters: $w^{[1]}, \cancel{b^{[1]}}, w^{[2]}, \cancel{b^{[2]}}, \dots, w^{[L]}, \cancel{b^{[L]}}$

  $\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]}$

$$z^{[l]} = w^{[l]} * a^{[l-1]} + \cancel{b^{[l]}}$$

$$\tilde{z}^{[l]} = \gamma^{[l]} * z^{[l]} + \beta^{[l]}$$

Implementing Batch Normalization

- for t = 1, … M

  compute forward prop

    in each hidden layer, use BN to replace $z^{[l]}$ by $\tilde{z}^{[l]}$

  use back prop to compute gradients, $dw^{[l]}, \cancel{db^{[l]}}, d\beta^{[l]}, d\gamma^{[l]}$

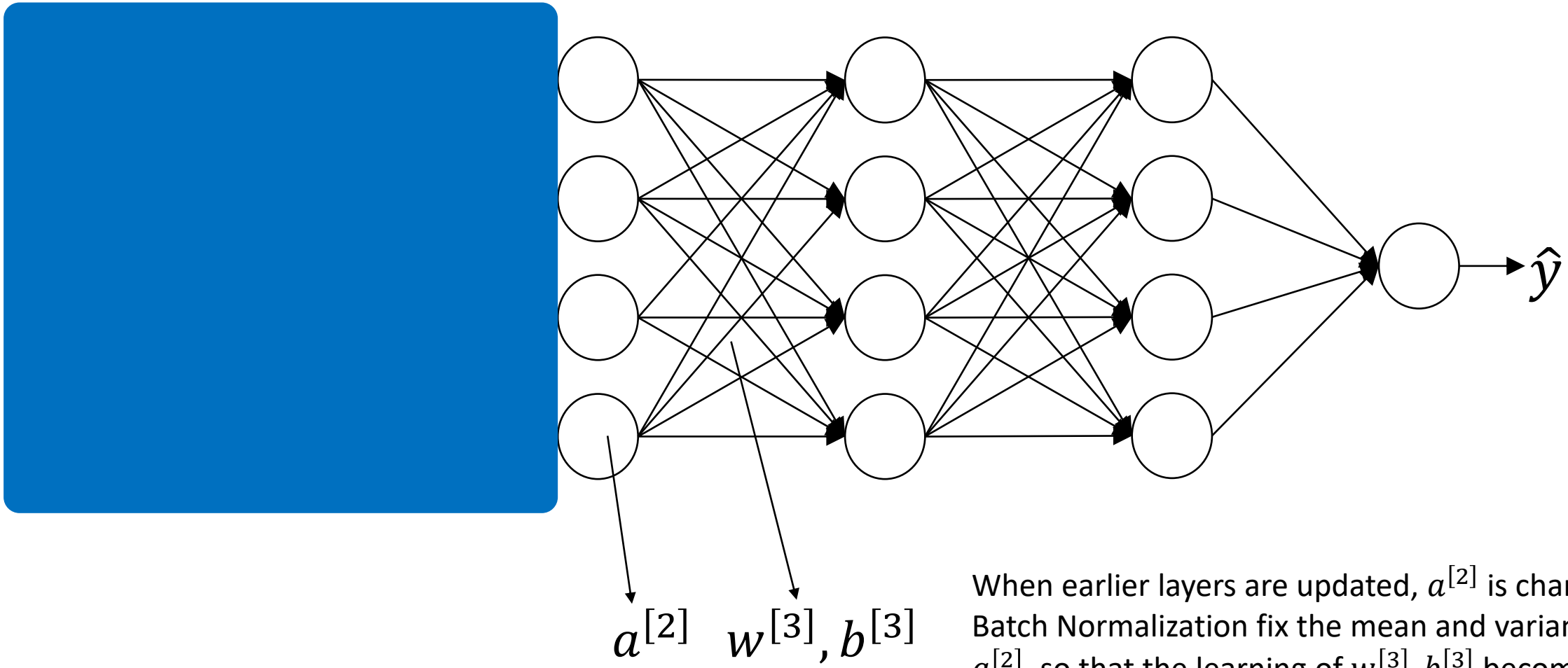
  update parameters:

$$w^{[l]} = w^{[l]} - \alpha * dw^{[l]}$$

$$\beta^{[l]} = \beta^{[l]} - \alpha * d\beta^{[l]}$$

$$\gamma^{[l]} = \dots$$

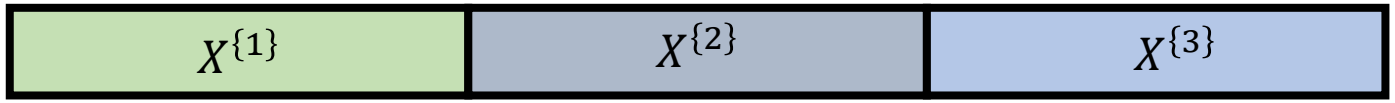
Why Batch Normalization Works?



When earlier layers are updated, $a^{[2]}$ is changing. Batch Normalization fix the mean and variance of $a^{[2]}$, so that the learning of $w^{[3]}, b^{[3]}$ becomes easier.

Batch Normalization in Testing Time

- In training, working on a mini-batch at a time
  - So that mean and variance can be computed
- At testing time, you have only a single input …
  - So you need an estimation of the mean and variance during training
  - Taking a running average over different mini batches



- Each mini batch has an estimated mean and variance

$$\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]}$$

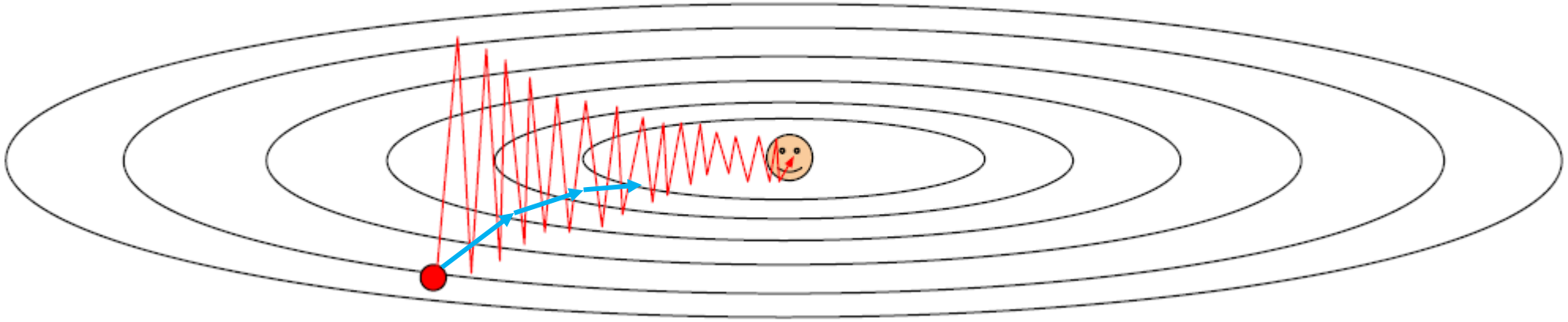
- Compute the mean $\mu^{[l]}$, which is updated at the i-th mini batch as:

$$\mu^{[l]} = \beta\mu^{[l]} + (1-\beta)\mu^{\{i\}[l]}$$

$$\beta = 0.9$$

Momentum

The learning rate must be very slow if standard gradient descent is used.



Initial $v = 0$

On iteration t:

compute dW on current mini-batch

$$v = \beta\, v + (1 - \beta) dW$$

$$w = w - \alpha * v$$

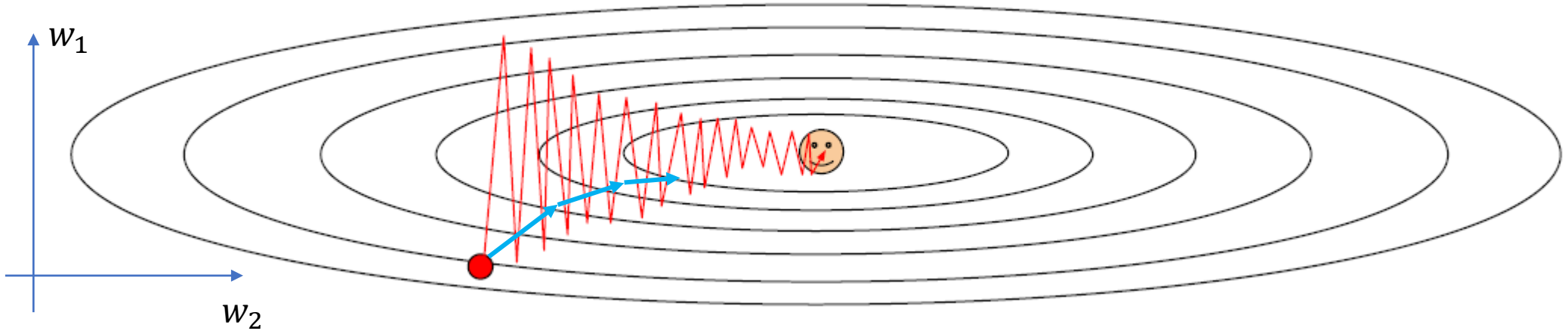
v is an exponentially weighted average of multiple gradients in the past.

Roughly, v will compute the average of the most recent $1/(1-\beta)$ gradient directions.

Typically, $\beta = 0.9$ and $\frac{1}{1-\beta} = 10$

In the above example, it leads to slower learning in vertical direction and faster learning in horizontal direction.

RMSprop



Initial S=0

On iteration t:

compute dW on current mini-batch

$$S = \beta\, S + (1-\beta)\, dW^2$$

$$w = w - \alpha * dw/\sqrt{S + \epsilon}$$

this is an elementwise square

S is an exponentially weighted average of gradient magnitude (squared).

The original gradient direction (red arrows) has larger value in the direction of w_1 (vertical) than w_2 (horizontal)

So, dividing by S will reduce the learning speed in vertical direction (i.e. divided by larger number)

Adam (adaptive momentum estimation)

Initial $v = 0, S = 0$

On iteration t:

compute dW on current mini-batch

$v = \beta_1 v + (1 - \beta_1) dW$ ⟶ Momentum

$v^{correct} = v/(1 - \beta_1^t)$ ⟶ A correction to avoid the bias by the initial 0 value

$S = \beta_2 S + (1 - \beta_2)\, dW^2$ ⟶ RMSprop

$S^{correct} = S/(1 - \beta_2^t)$ ⟶ A correction to avoid the bias by the initial 0 value

$w = w - \alpha * v^{correct} / \sqrt{S^{correct} + \epsilon}$

Common choices of the hyperparameters:

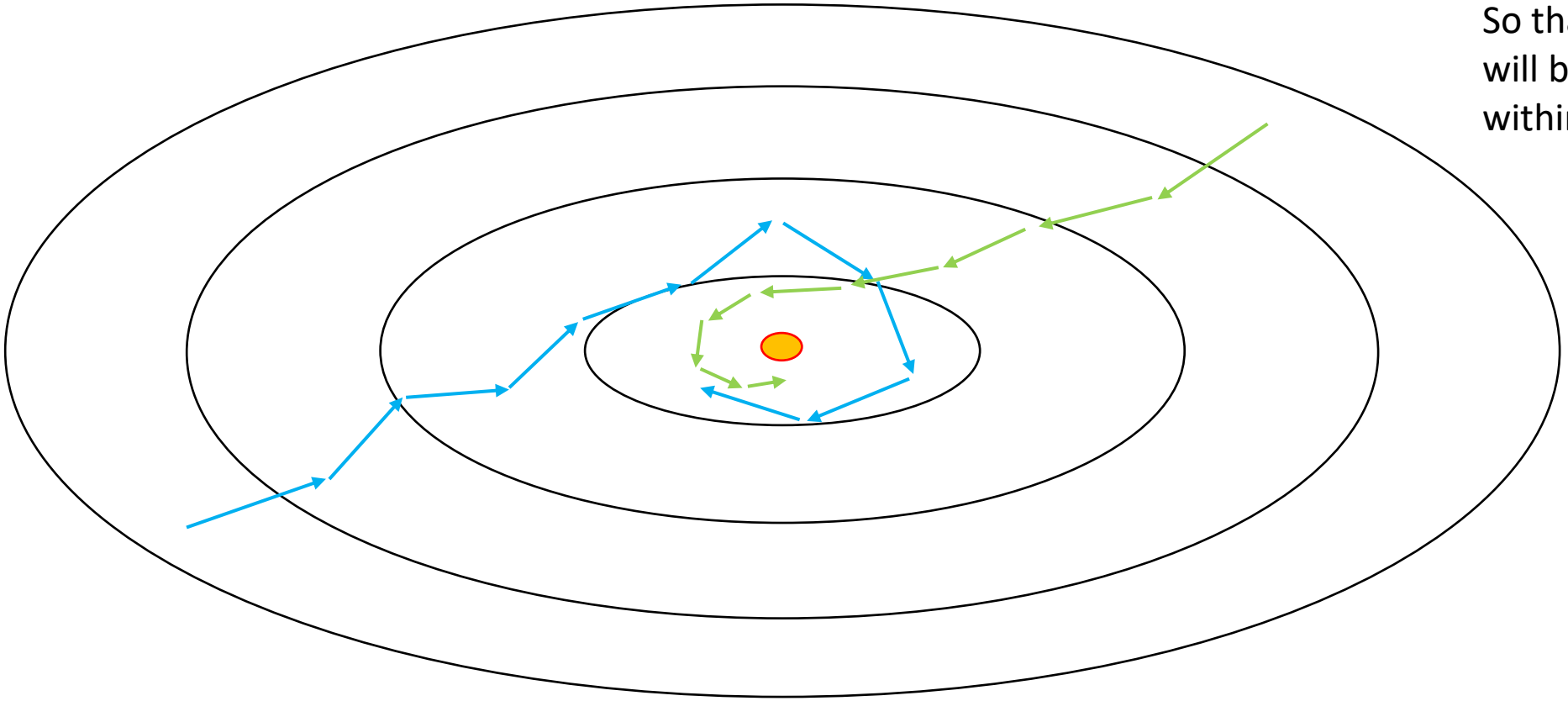
α: needs to be tuned

β_1: 0.9

β_2: 0.999

ϵ: 10^{-8}

Learning Rate Decay

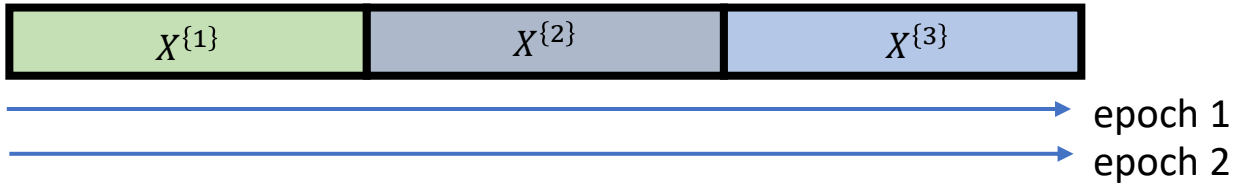


Reduce the learning rate (the step size of gradient descent)

So that the searching algorithm will be more stable (wandering within a tighter neighborhood)

Learning Rate Decay

- Some typical choices of decay:
  - $\alpha = \frac{1}{1+decay\_rate \ *epoch\_num}$
  - $\alpha = 0.95^{epoch\_num}$
  - $\alpha = k/\sqrt{epoch\_num}$
  - …



Training Neural Networks Involves Many Tweaking

- Before you start:
  - Data preprocessing
  - Parameter initialization
- Tricks in training
  - Regularization
  - Dropout
  - Batch Normalization
  - Momentum, RMSprop, Adam, Learning rate decay
- Tuning
  - Hyperparameter tuning
  - Data augmentation
  - Transfer learning

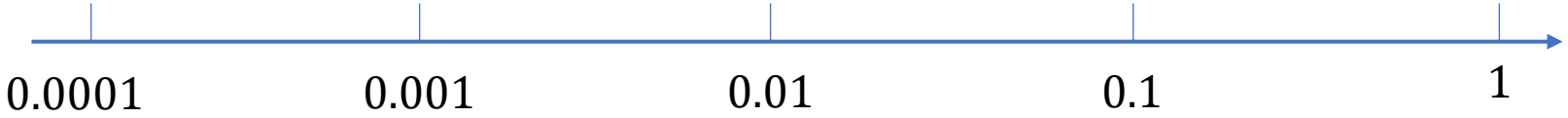
Hyperparameter Tuning

- Many hyperparameters, some are more important
  - Network structure
  - Learning rate: α (often the most important one)
  - Momentum term: β (when using Adam, $\beta_1, \beta_2, \epsilon$ are often not tuned)
  - Mini-batch size
  - # layers
  - Learning rate decay

Hyperparameter Tuning

- Sampling in the right scale is important
  - Consider the learning rate α, maybe you want to search it between $[0.0001, 1]$

Uniform sampling is poor, spending 90% of the time searching $[0.1, 1]$



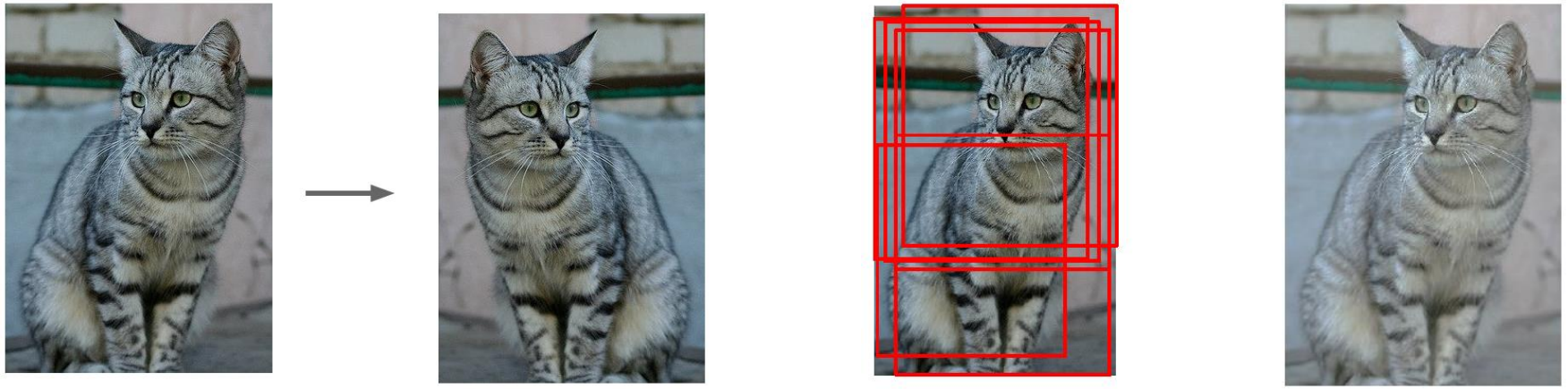
Better to uniform sampling in the log space

  - Think about momentum term β, maybe you want to search between $[0.9, 0.999]$

Search in the log space, and search for $1 - \beta$

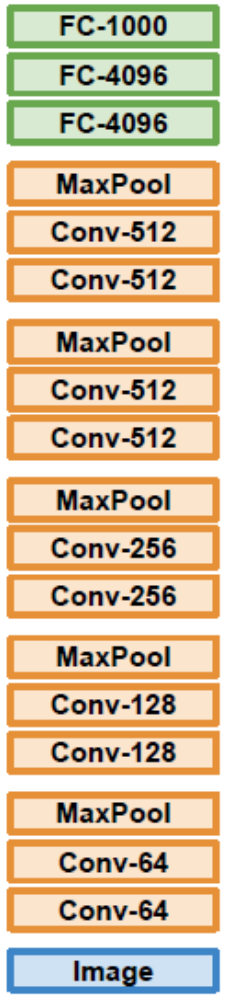
Data Augmentation

- CNNs are always hunger for data
- Generating more data by applying transformations
  - Horizontal flipping
  - Random cropping
  - Adjust contrast & brightness
  - Others: rotation, shearing, stretching, lens distortion (go crazy), etc…

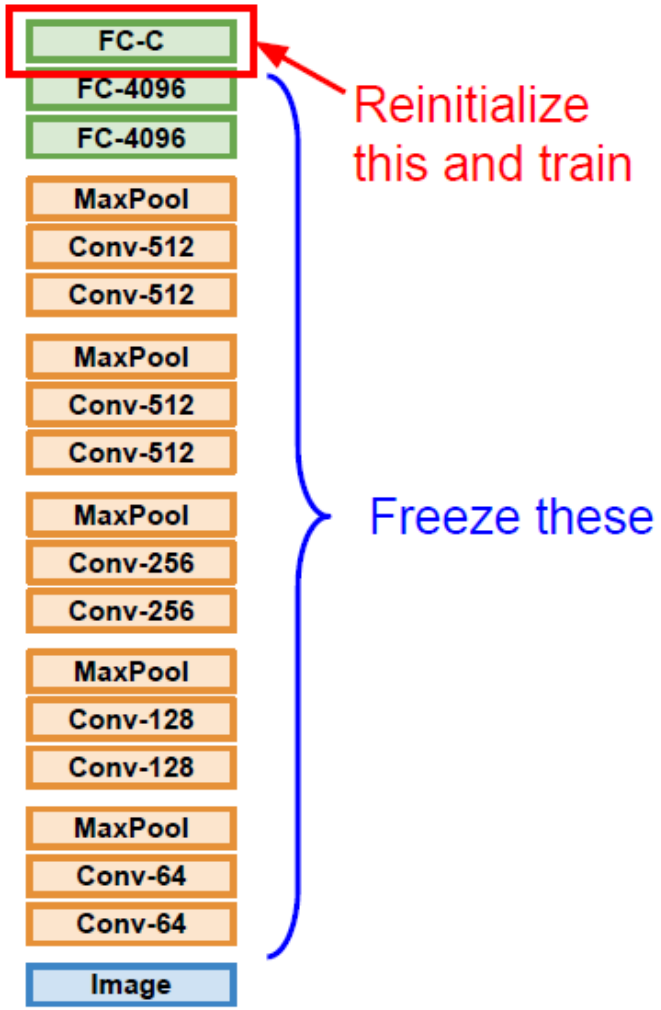


Transfer Learning

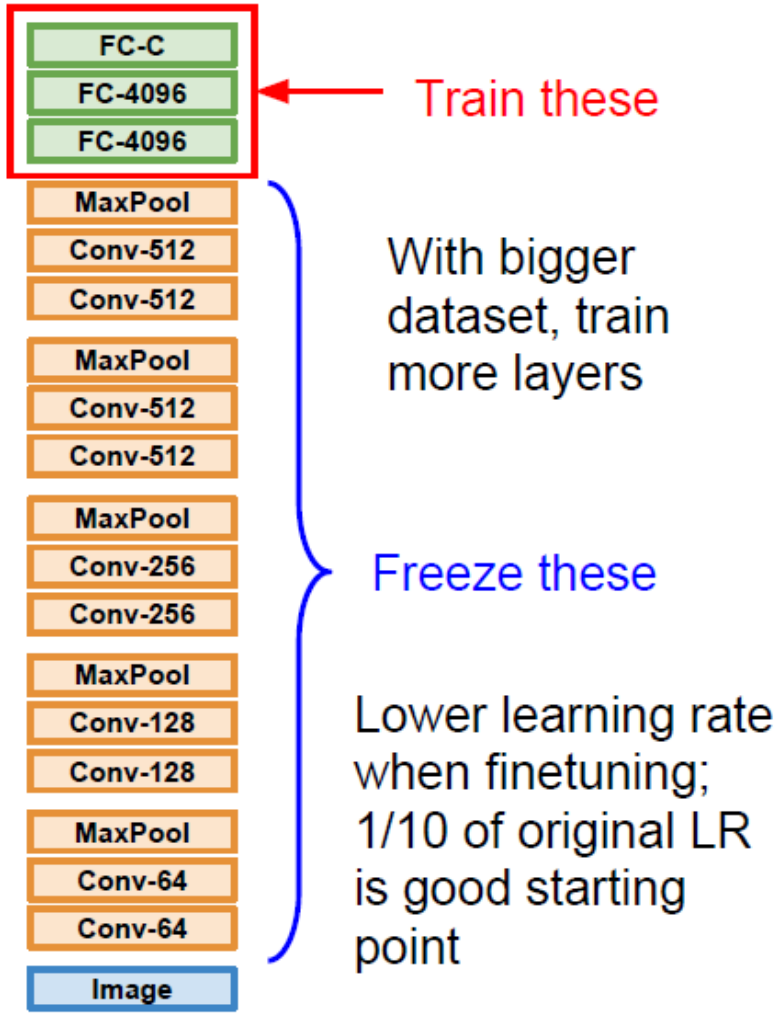
1. Train on Imagenet



2. Small Dataset (C classes)



3. Bigger dataset



Questions?