Convolutional Neural Networks

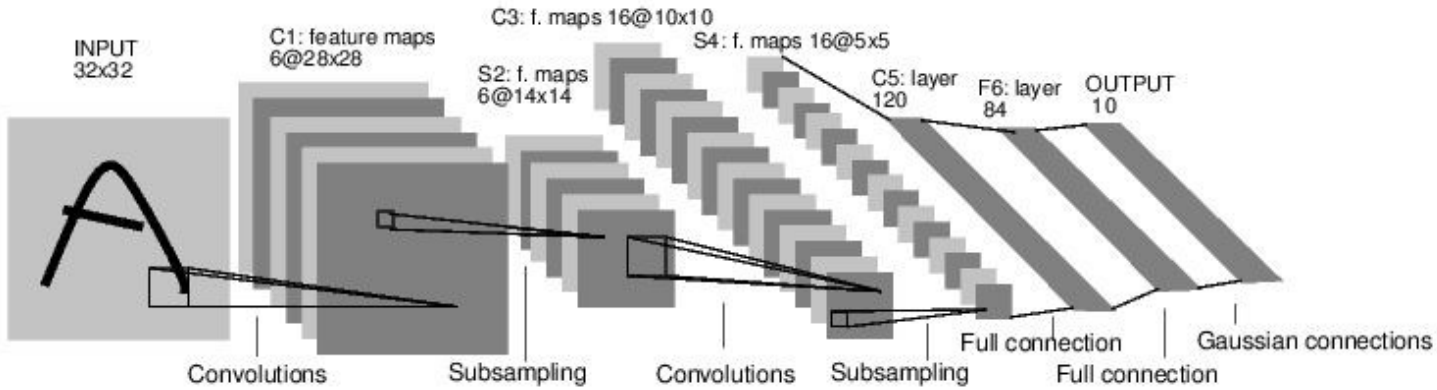
Samuel Cheng

School of ECE
University of Oklahoma

Spring, 2018

Table of Contents

Convolutional Neural Networks

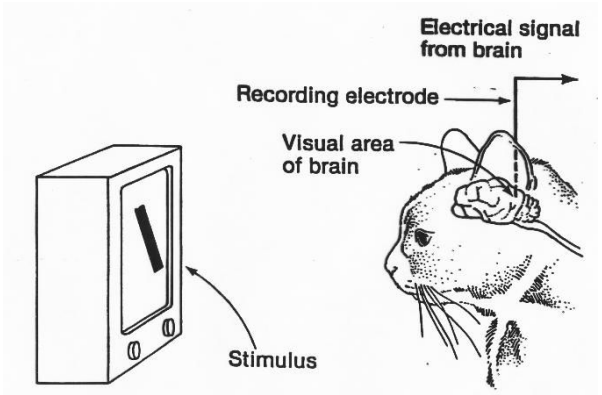


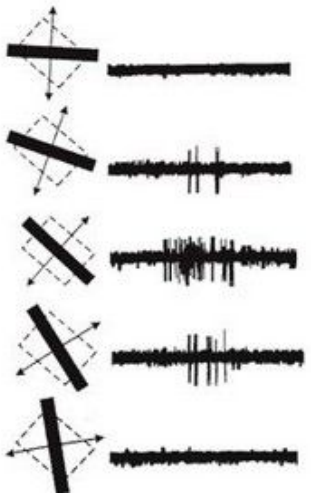
[LeNet-5, LeCun 1998]

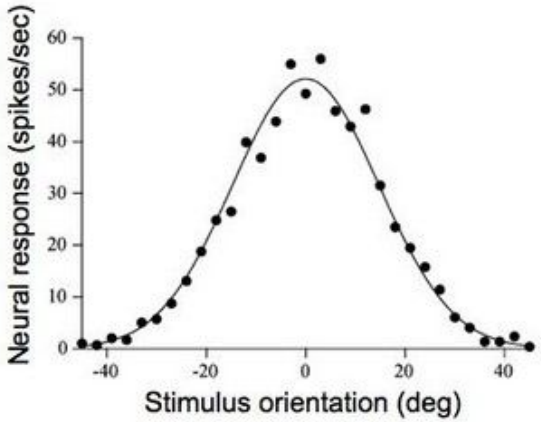
CNN history

A bit of history:

Hubel & Wiesel,
1959
RECEPTIVE FIELDS OF SINGLE NEURONES IN
THE CAT'S STRIATE CORTEX
1962
RECEPTIVE FIELDS, BINOCULAR INTERACTION
AND FUNCTIONAL ARCHITECTURE IN
THE CAT'S VISUAL CORTEX
1968...



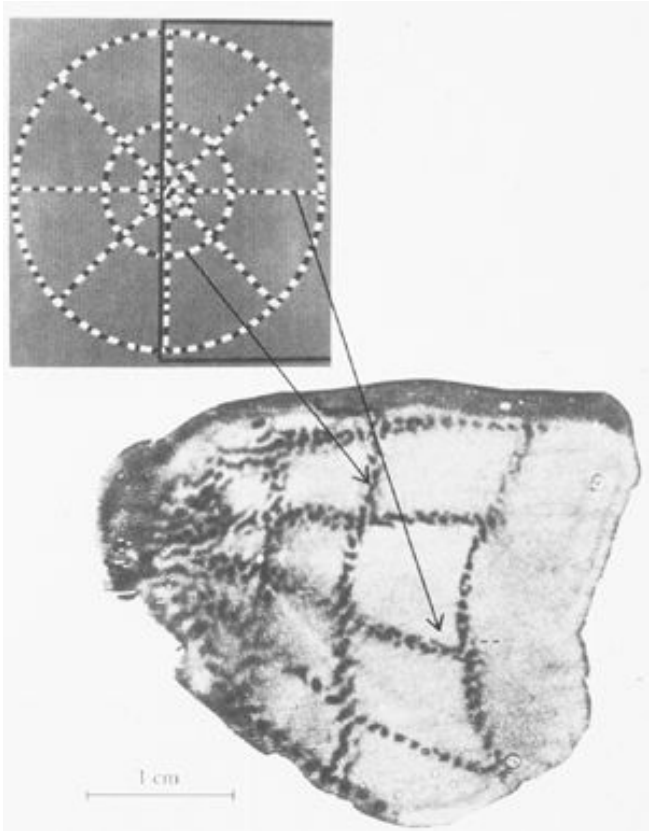




CNN history

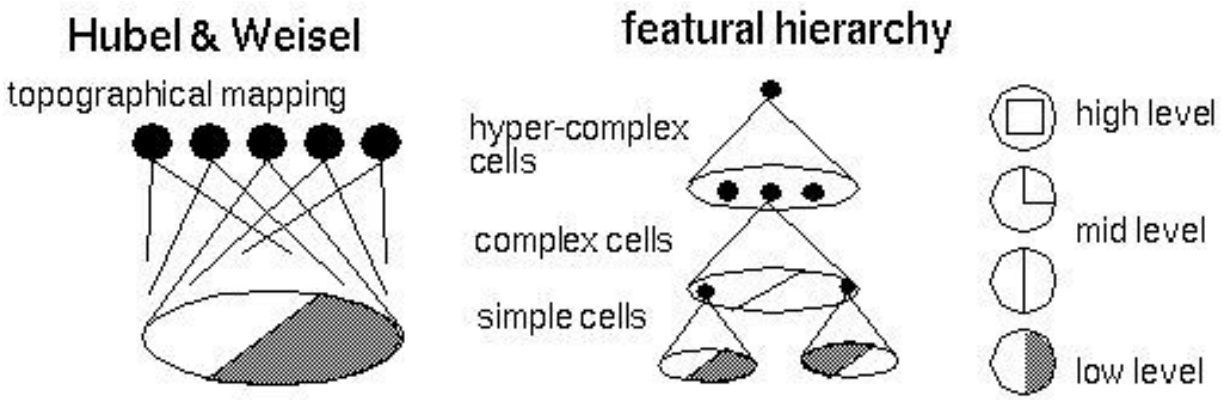
A bit of history

Topographical mapping in the cortex:
nearby cells in cortex represented
nearby regions in the visual field



CNN history

Hierarchical organization

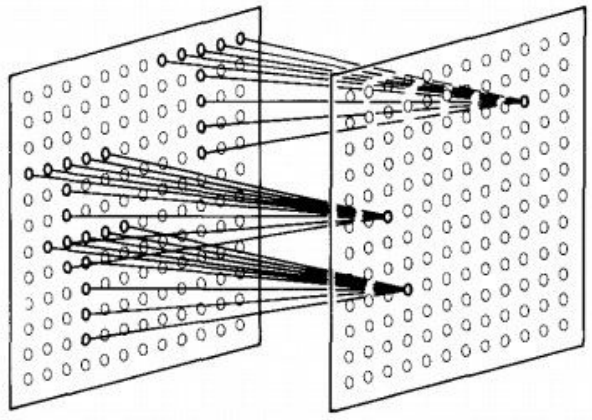


LGB (lateral geniculate body) → simple cells → complex cells → lower order hypercomplex cells → higher order hypercomplex cells
Experiment video, explanation

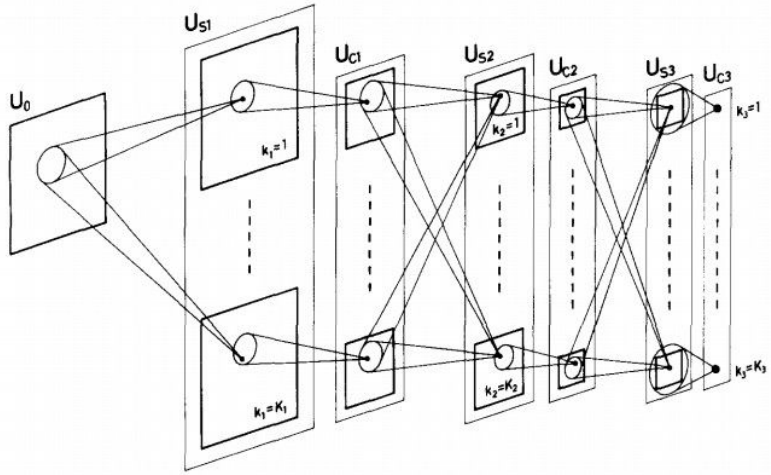
CNN history

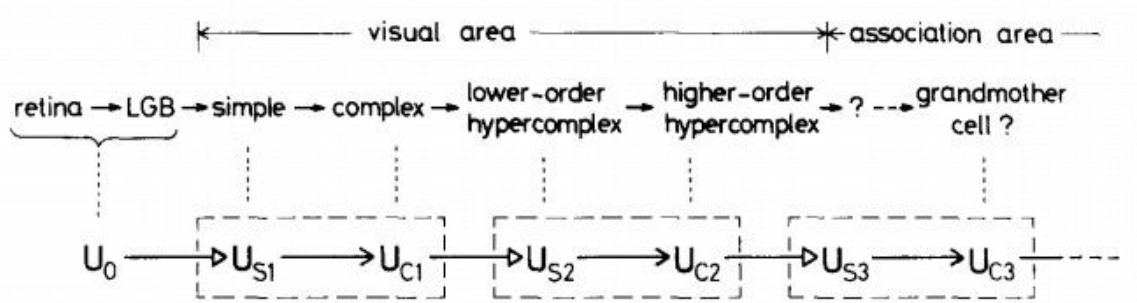
A bit of history:

Neurocognitron
[Fukushima 1980]



"sandwich" architecture (SCSCSC…)
simple cells: modifiable parameters
complex cells: perform pooling



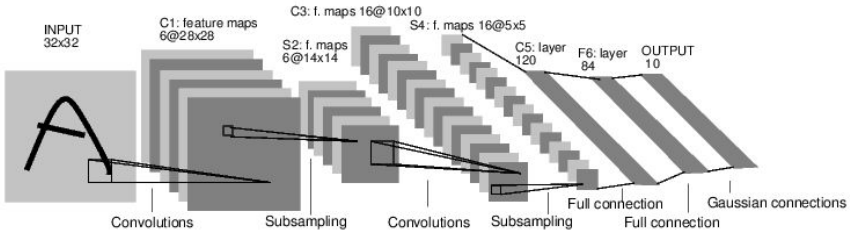


CNN history

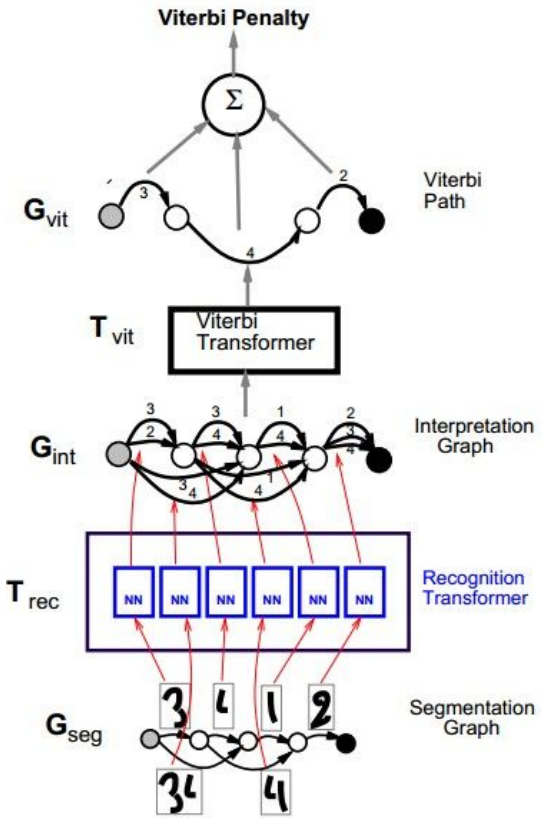
A bit of history:

Gradient-based learning applied to document recognition

[LeCun, Bottou, Bengio, Haffner 1998]



LeNet-5

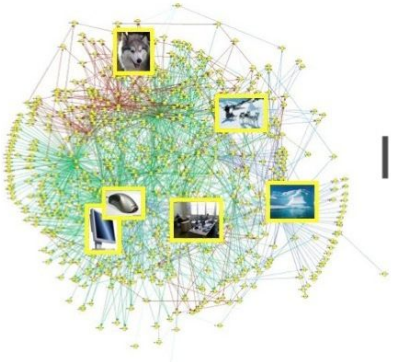


CNN today

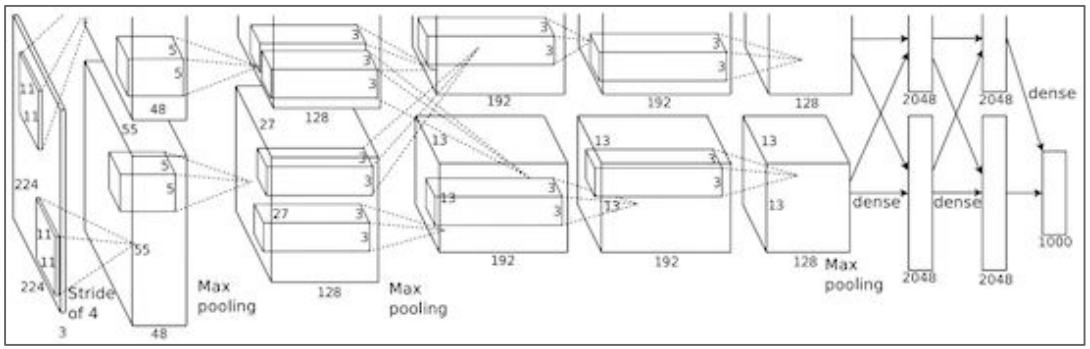
A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]



IMAGENET



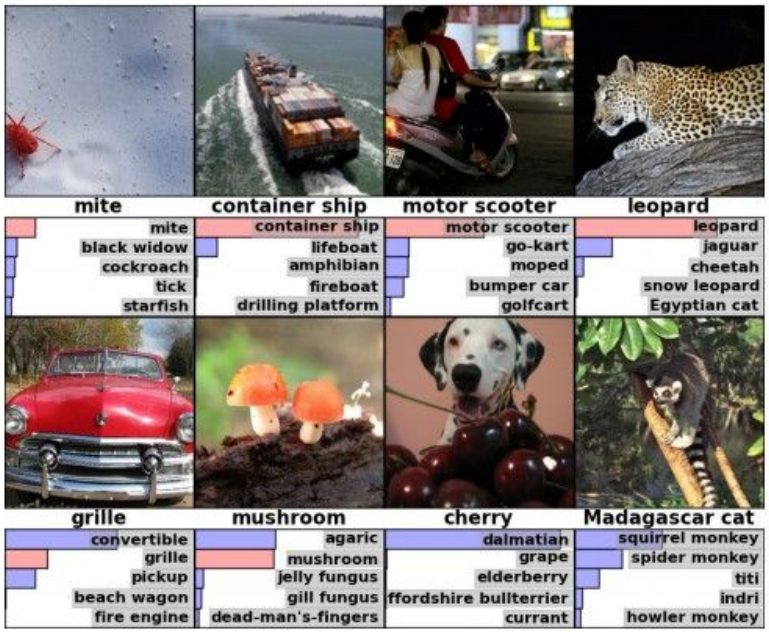
"AlexNet"

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 6 - 72 25 Jan 2016

CNN today

Fast-forward to today: ConvNets are everywhere

Classification



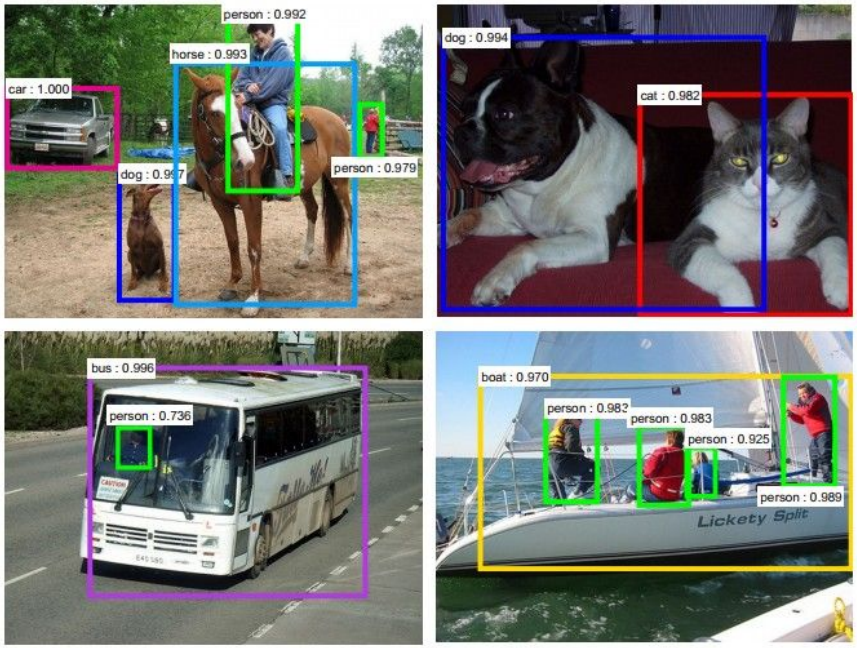
Retrieval

[Krizhevsky 2012]

CNN today

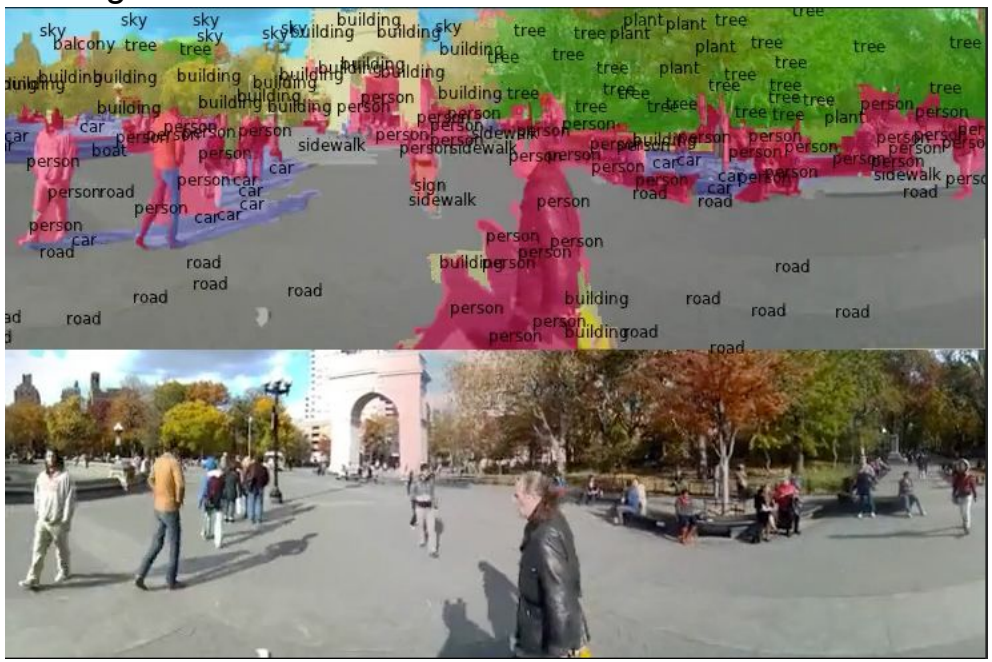
Fast-forward to today: ConvNets are everywhere

Detection



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

Segmentation



[Farabet et al., 2012]

CNN today

Fast-forward to today: ConvNets are everywhere

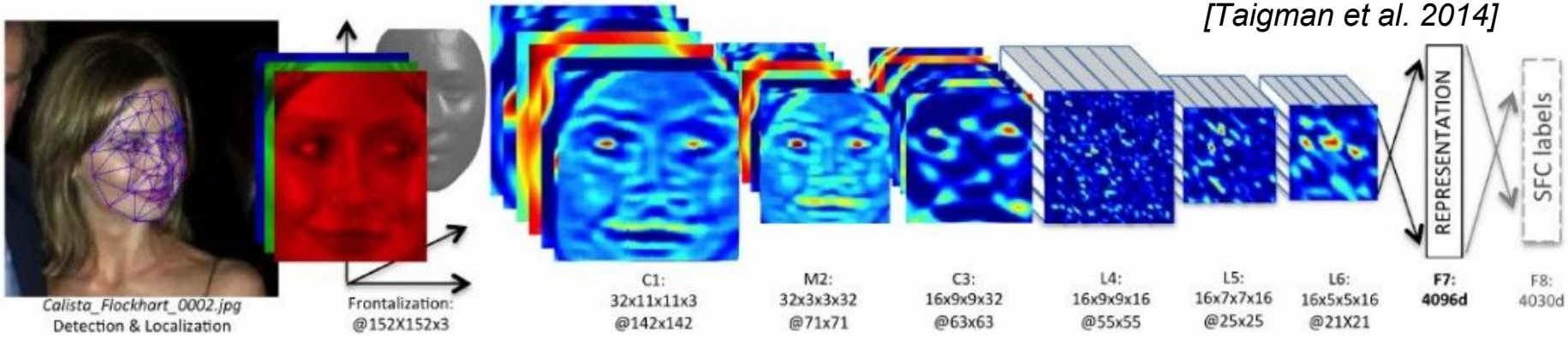
self-driving cars

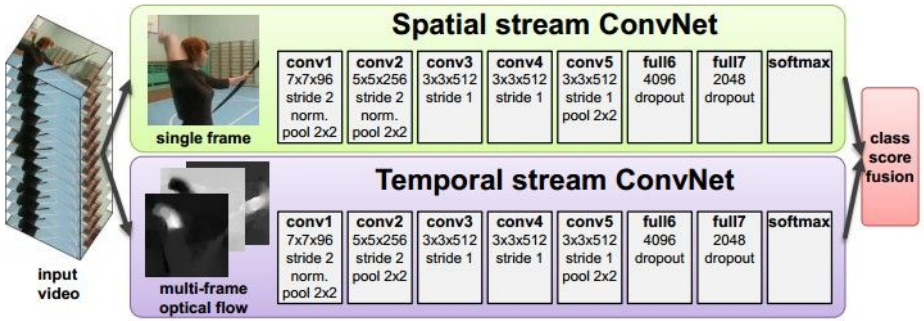
NVIDIA Tegra X1

CNN today

Fast-forward to today: ConvNets are everywhere

[Taigman et al. 2014]





[Simonyan et al. 2014]

[Goodfellow 2014]

CNN today

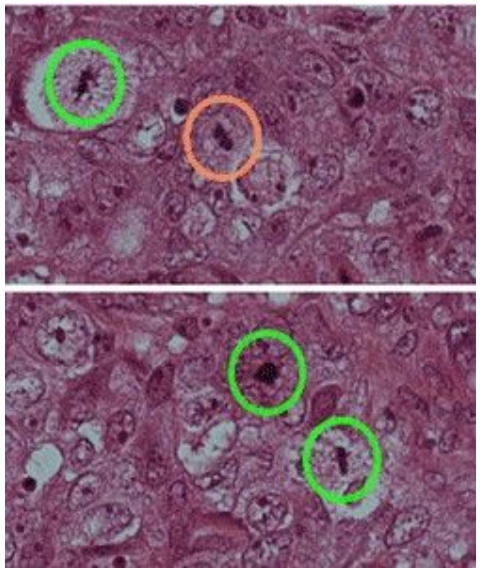
Fast-forward to today: ConvNets are everywhere

[Toshev, Szegedy 2014]

[Mnih 2013]

CNN today

Fast-forward to today: ConvNets are everywhere

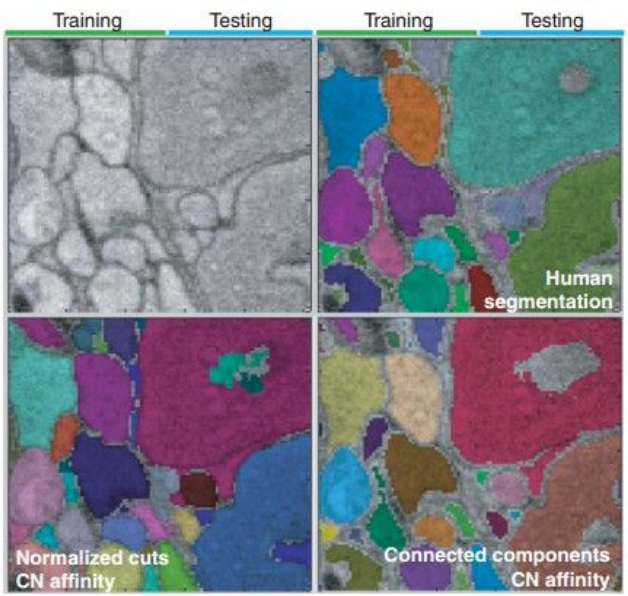


[Ciresan et al. 2013]

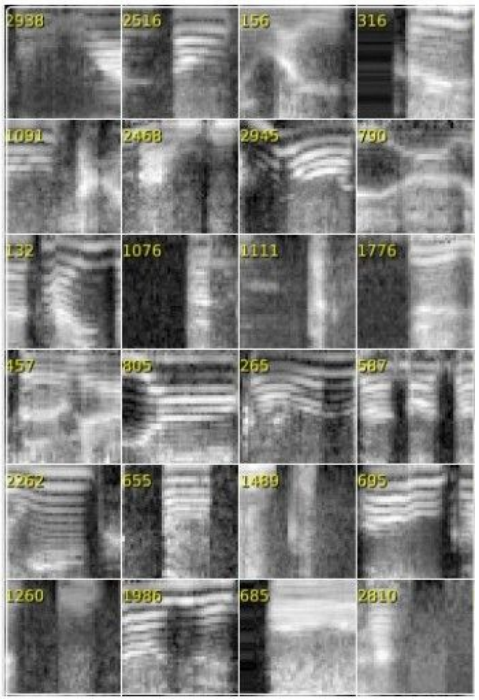
[Sermanet et al. 2011]
[Ciresan et al.]

CNN today

Fast-forward to today: ConvNets are everywhere



[Turaga et al., 2010]



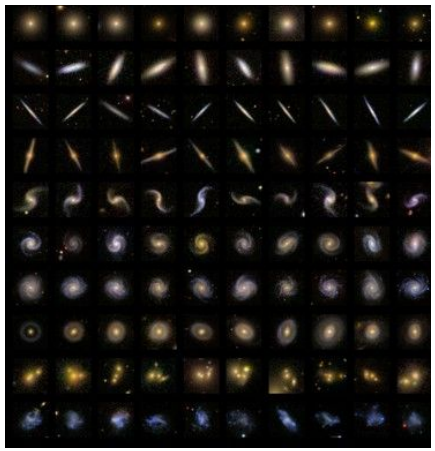
I caught this movie on the Sci-Fi channel recently. It actually turned out to be pretty decent as far as B-list horror/suspense films go. **Two guys (one naive and one loud mouthed a **) take a road trip to stop a wedding but have the worst possible luck when a maniac in a freaky, make-shift tank/truck hybrid decides to play cat-and-mouse with them.** Things are further complicated when they pick up a ridiculously whorish hitchhiker. What makes this film unique is that the combination of comedy and terror actually work in this movie, unlike so many others. The two guys are likable enough and there are some good chase/suspense scenes. Nice pacing and comic timing make this movie more than passable for the horror/slasher buff. **Definitely worth checking out.**

I just saw this on a local independent station in the New York City area. **The cast showed promise but when I saw the director, George Cosmotos, I became suspicious. And sure enough, it was every bit as bad, every bit as pointless and stupid as every George Cosmotos movie I ever saw.** He's like a stupid man's Michael Bey – with all the awfulness that accolade promises. There's no point to the conspiracy, no burning issues that urge the conspirators on. We are left to ourselves to connect the dots from one bit of graffiti on various walls in the film to the next. Thus, the current budget crisis, the war in Iraq, Islamic extremism, the fate of social security, 47 million Americans without health care, stagnating wages, and the death of the middle class are all subsumed by the sheer terror of graffiti. A truly, stunningly idiotic film.

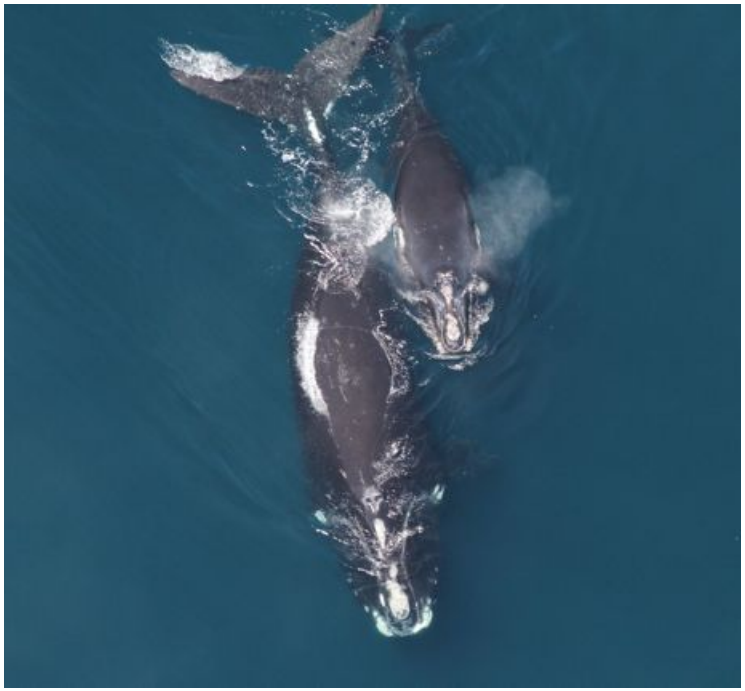
Graphics is far from the best part of the game. **This is the number one best TH game in the series.** Next to Underground. **It deserves strong love. It is an insane game.** There are massive levels, massive unlockable characters... it's just a massive game. **Waste your money on this game. This is the kind of money that is wasted properly.** And even though graphics suck, thats doesn't make a game good. Actually, the graphics were good at the time. Today the graphics are crap. WHO CARES? As they say in Canada, This is the fun game, aye. (You get to go to Canada in THPS3) Well, I don't know if they say that, but they might. who knows. Well, Canadian people do. Wait a minute, I'm getting off topic. This game rocks. Buy it, play it, enjoy it, love it. It's PURE BRILLIANCE.

The first was good and original. I was a not bad horror/comedy movie. So I heard a second one was made and I had to watch it . What really makes this movie work is Judd Nelson's character and the sometimes clever script. **A pretty good script for a person who wrote the Final Destination films and the direction was okay.** Sometimes there's scenes where it looks like it was filmed using a home video camera with a grainy - look. Great made - for - TV movie. **It was worth the rental and probably worth buying just to get that nice eerie feeling and watch Judd Nelson's Stanley doing what he does best.** I suggest newcomers to watch the first one before watching the sequel, just so you'll have an idea what Stanley is like and get a little history background.

[Denil et al. 2014]



CNN today



Whale recognition, Kaggle Challenge

Mnih and Hinton, 2010

CNN today

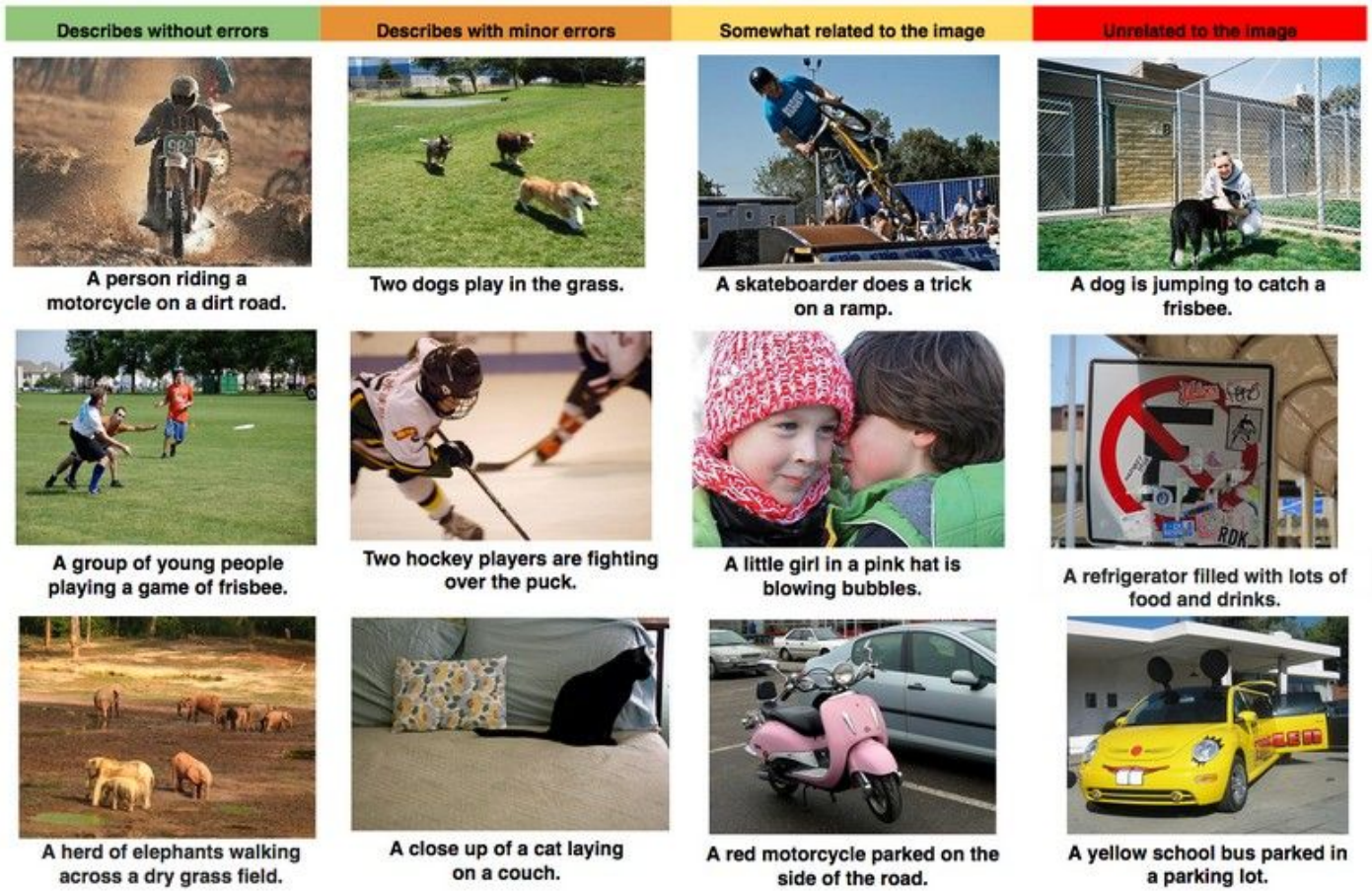
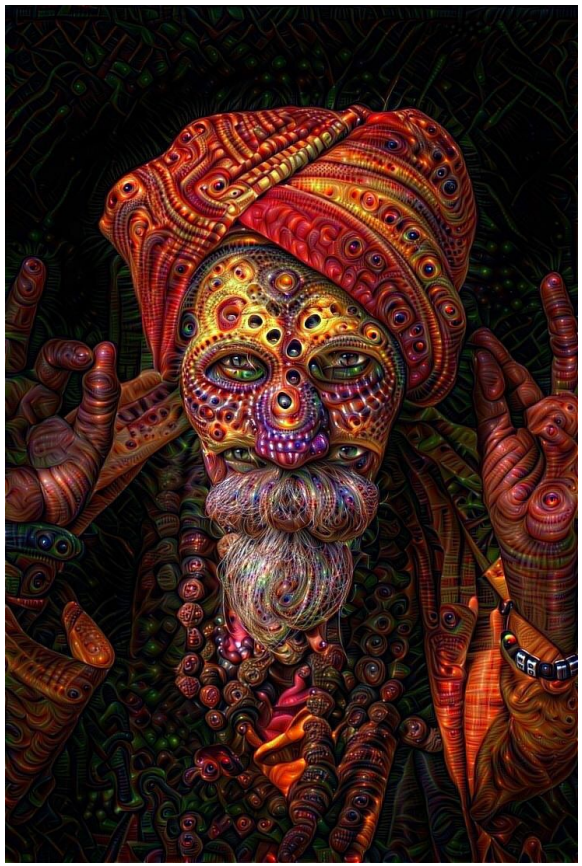
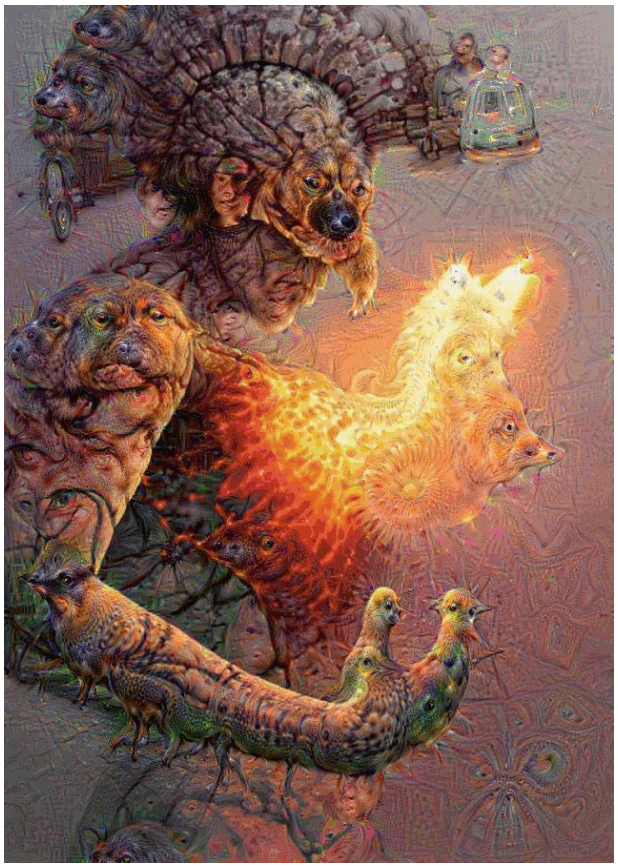


Image Captioning

[Vinyals et al., 2015]

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 6 - 81 25 Jan 2016

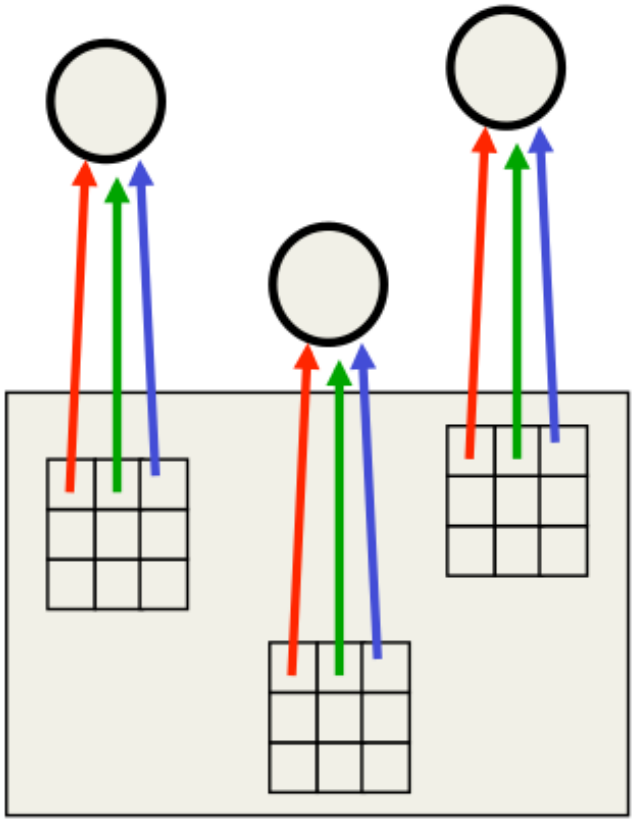
CNN today

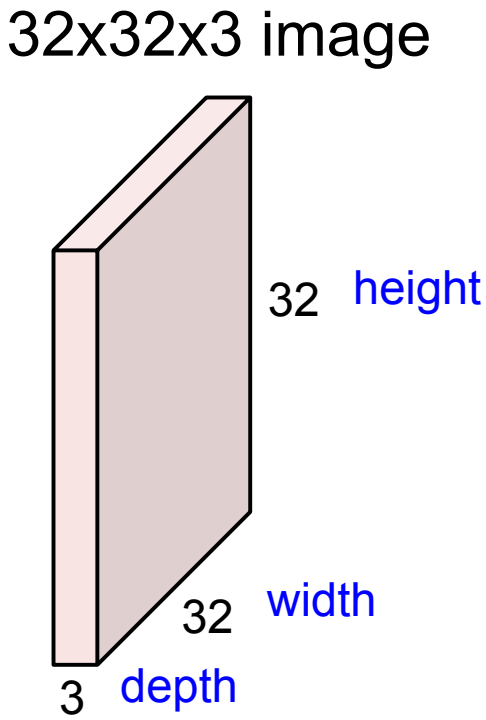
reddit.com/r/deepdream

Motivation of CNN

- A same object under different viewpoints is very different in pixel domain
  - A slightly horizontally shifted image has change imperceivable to us but can confuse naive recognition system
- Ideally, we may want to have shift-invariant features
- In practice, if we have local feature suitable for a particular region, the same feature should work well with other region
  - Weight sharing across space $\rightarrow$ CNN



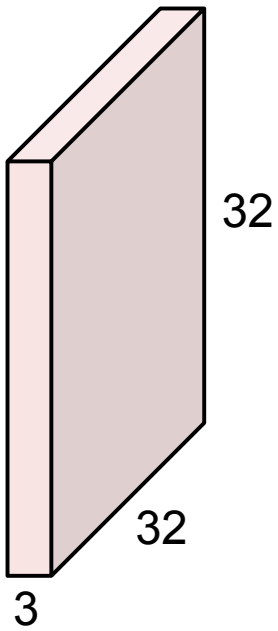
Convolution Layer



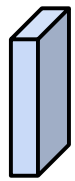
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 7 - 10 27 Jan 2016

Convolution Layer

32x32x3 image



5x5x3 filter

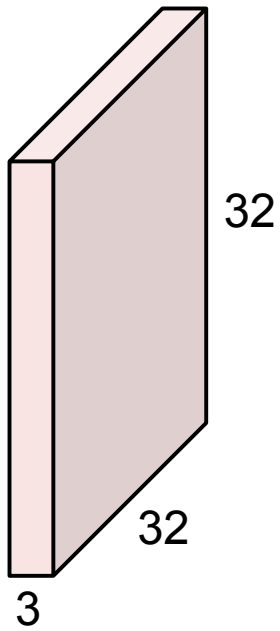


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

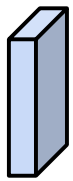
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

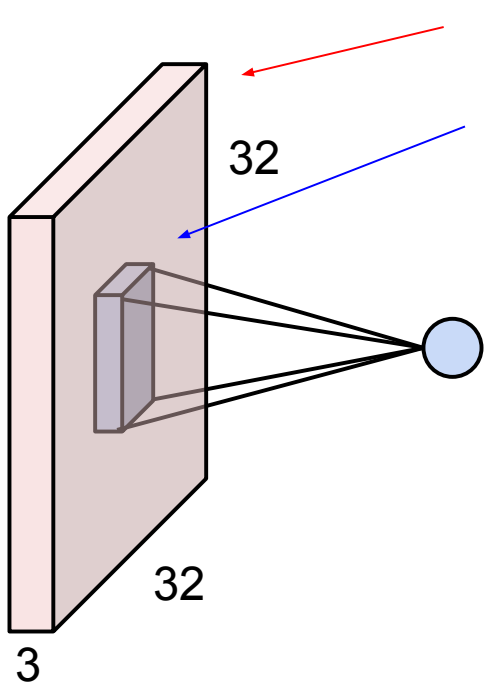


5x5x3 filter



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

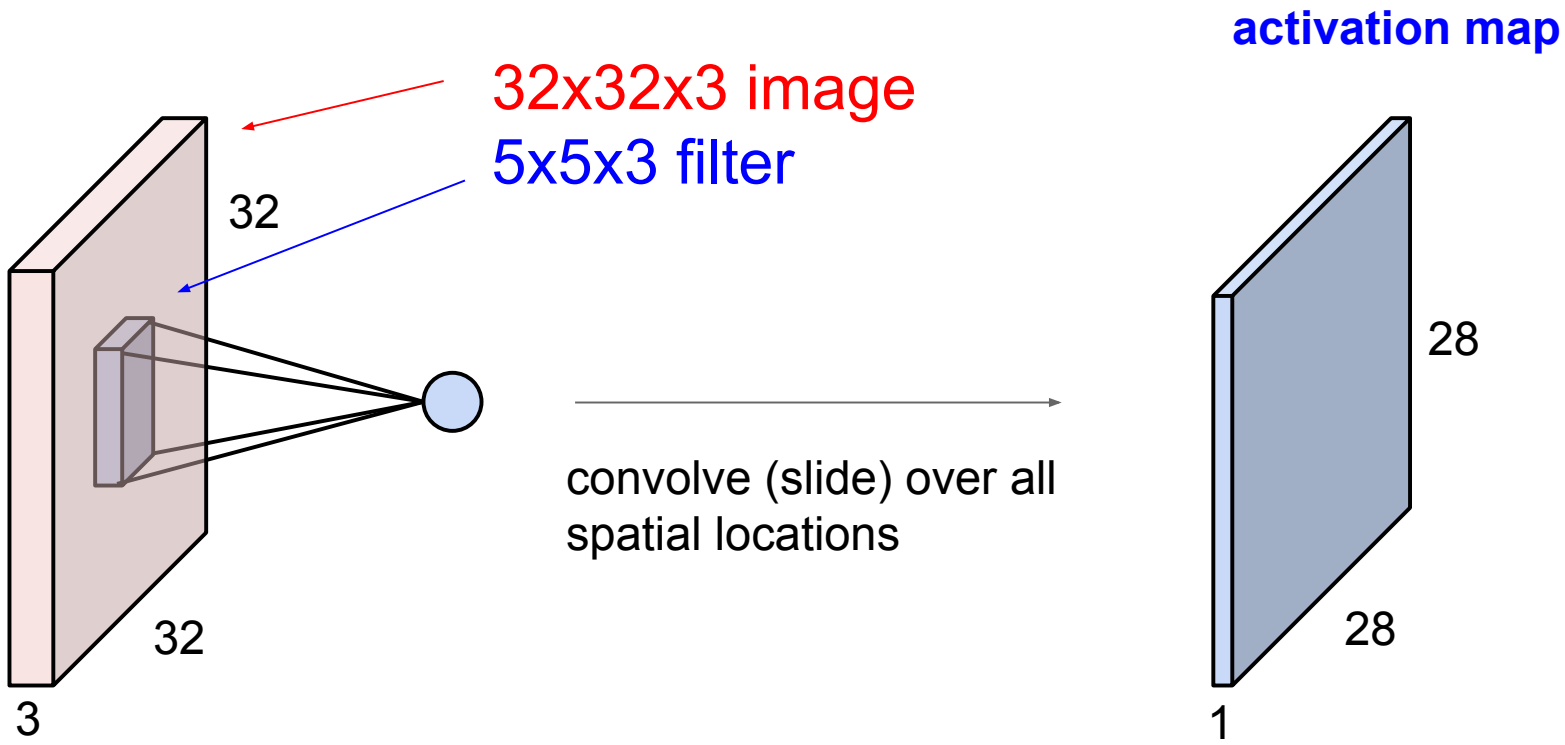
Convolution Layer



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

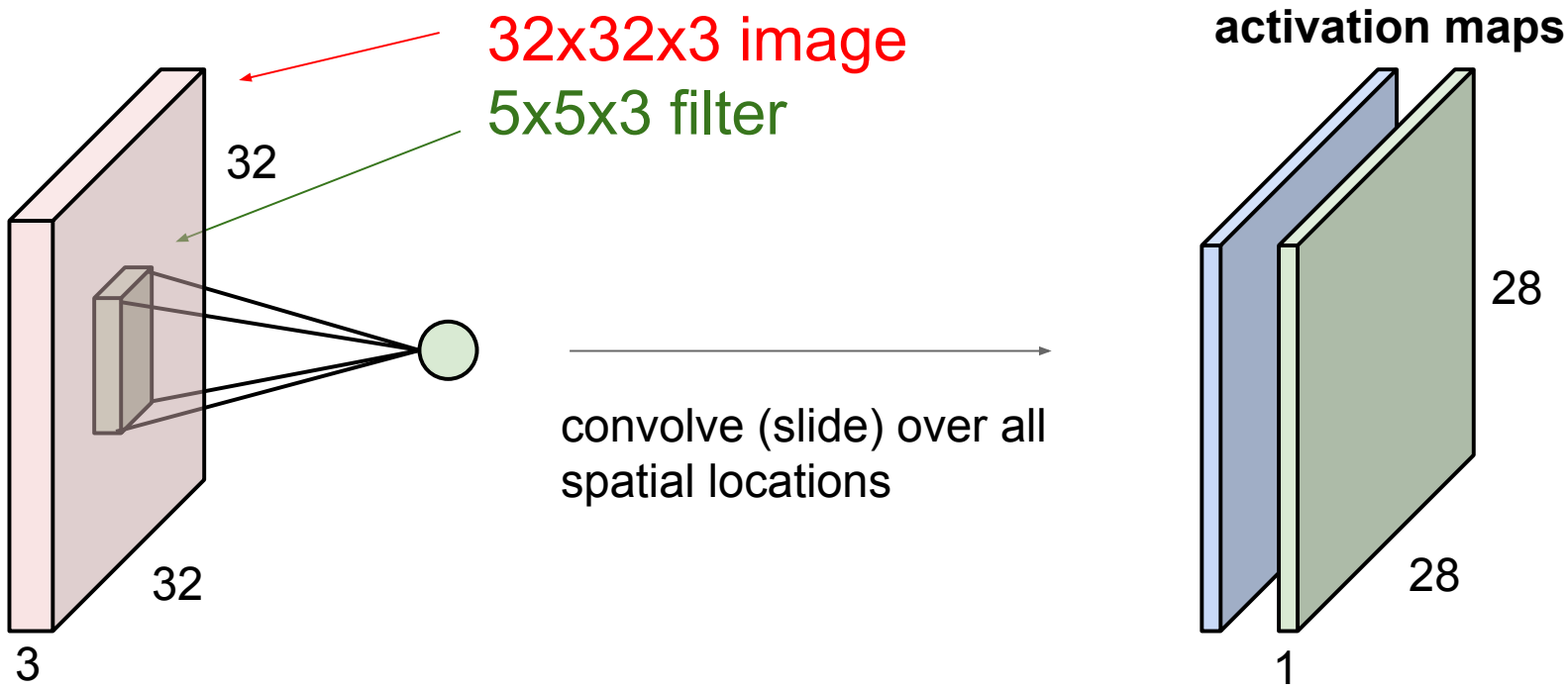
$$w^T x + b$$

Convolution Layer

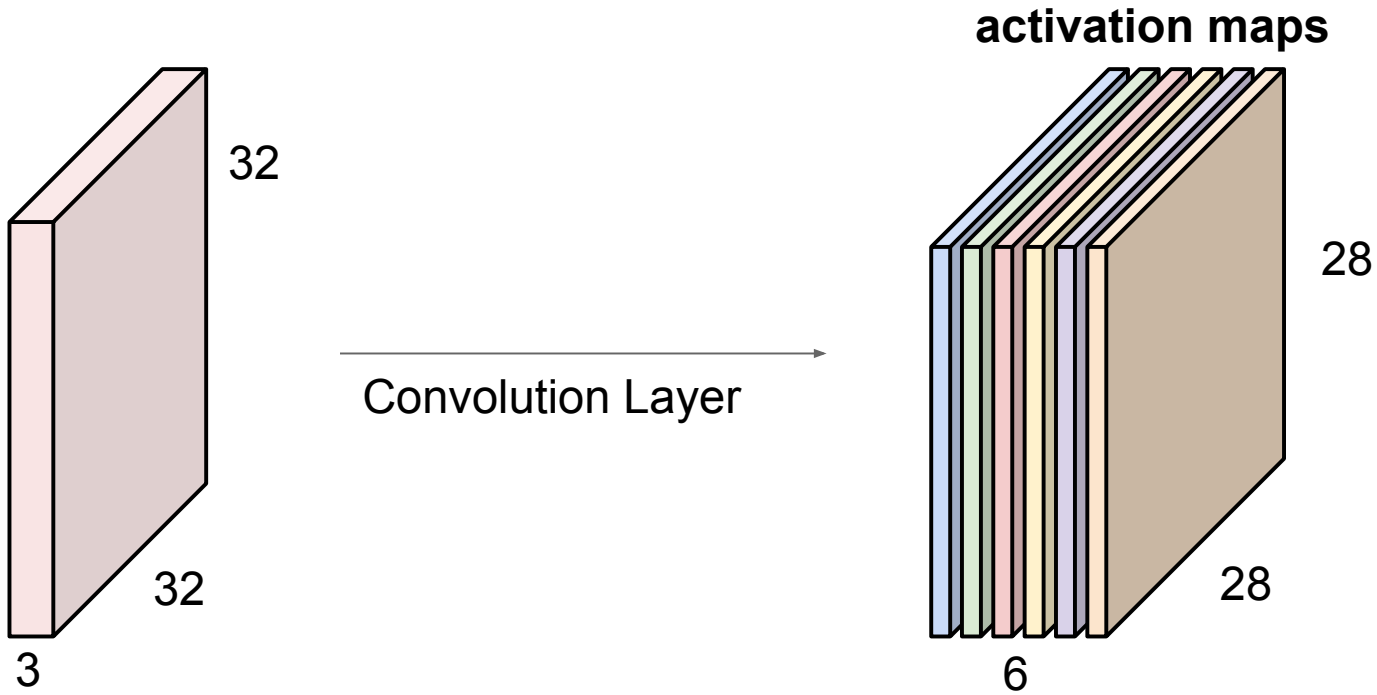


Convolution Layer

consider a second, green filter



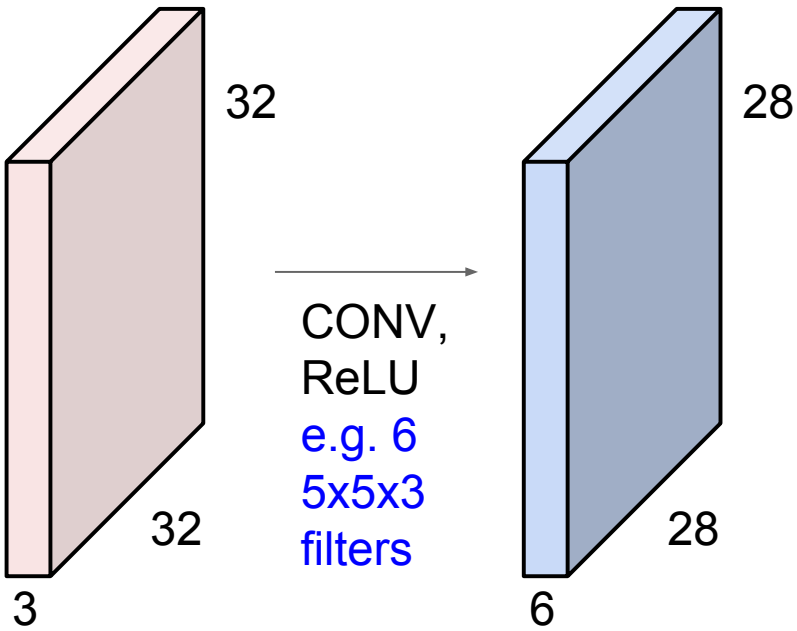
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



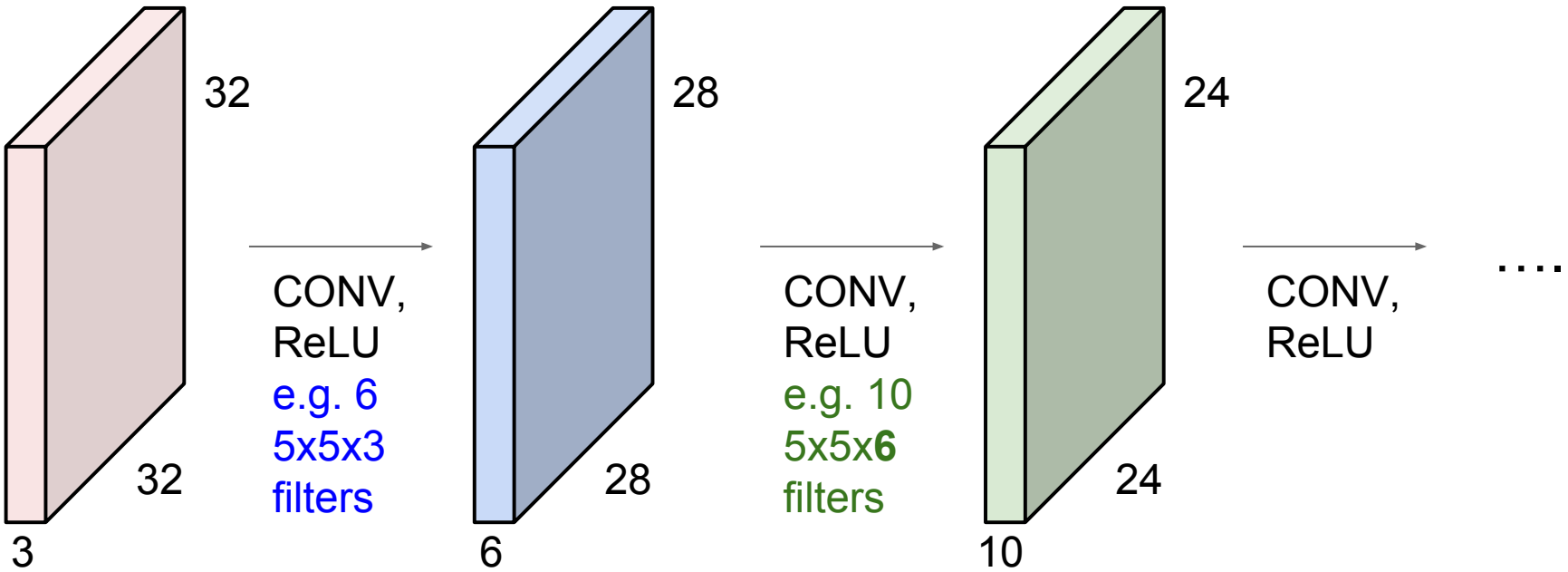
We stack these up to get a "new image" of size 28x28x6!

Fei-Fei Li & Andrej Karpathy & Justin Johnson | Lecture 7 - 16 | 27 Jan 2016

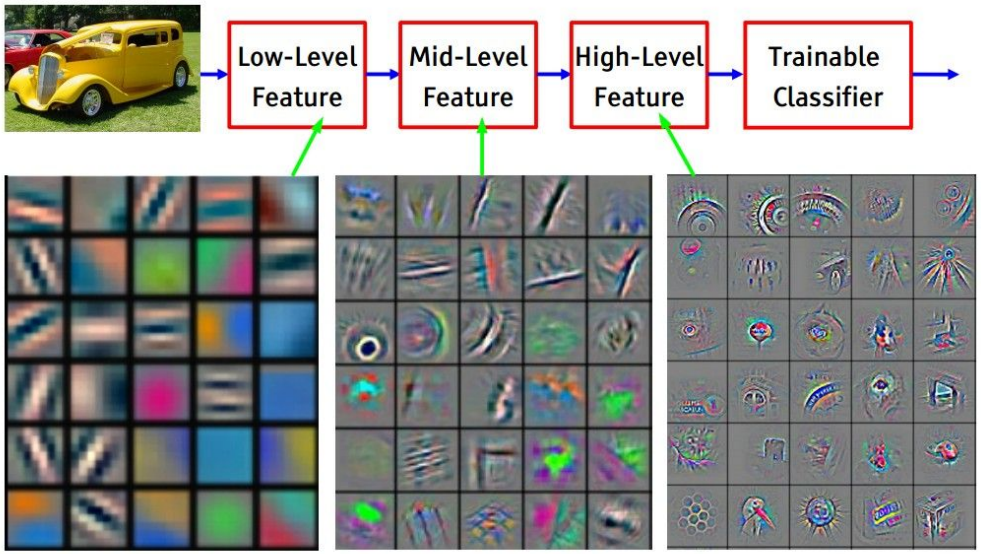
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



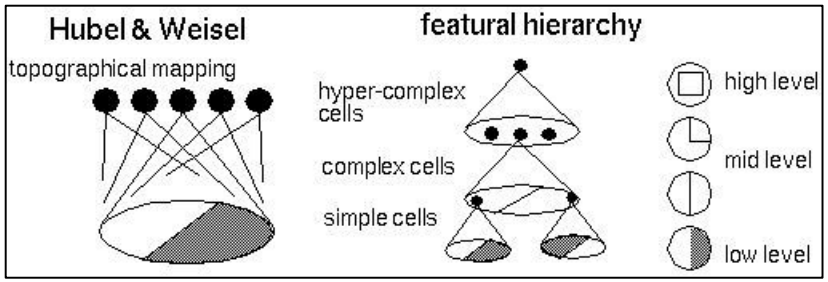
Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



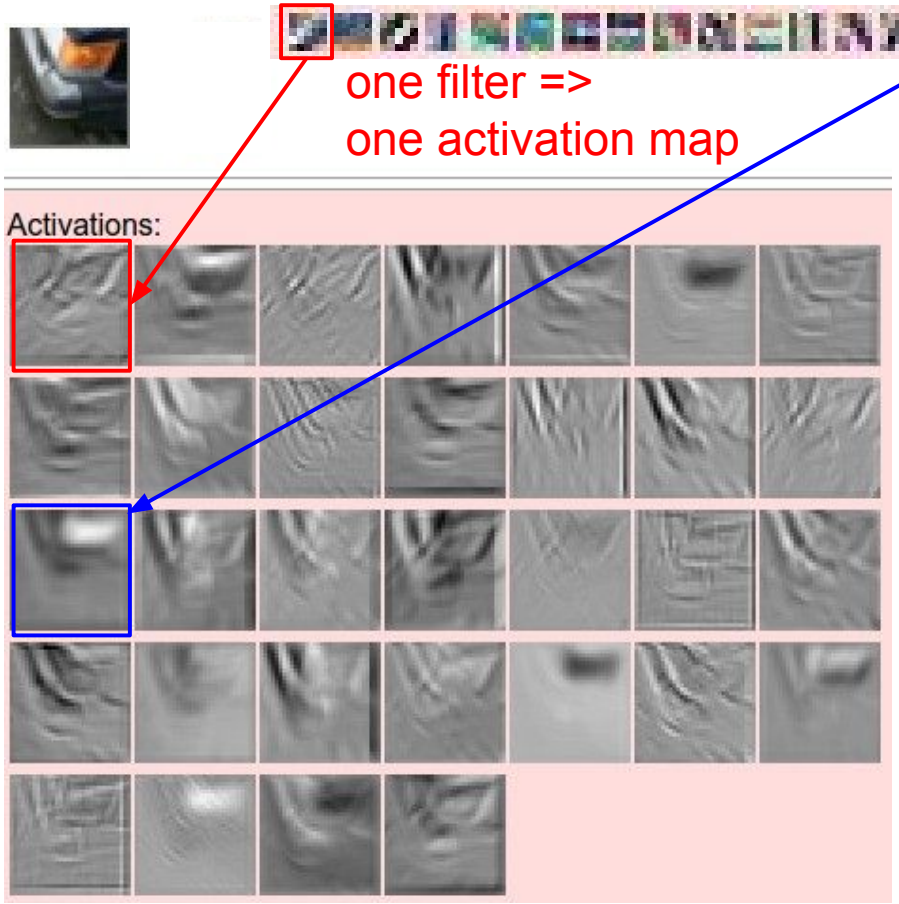
Preview



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



[From recent Yann LeCun slides]



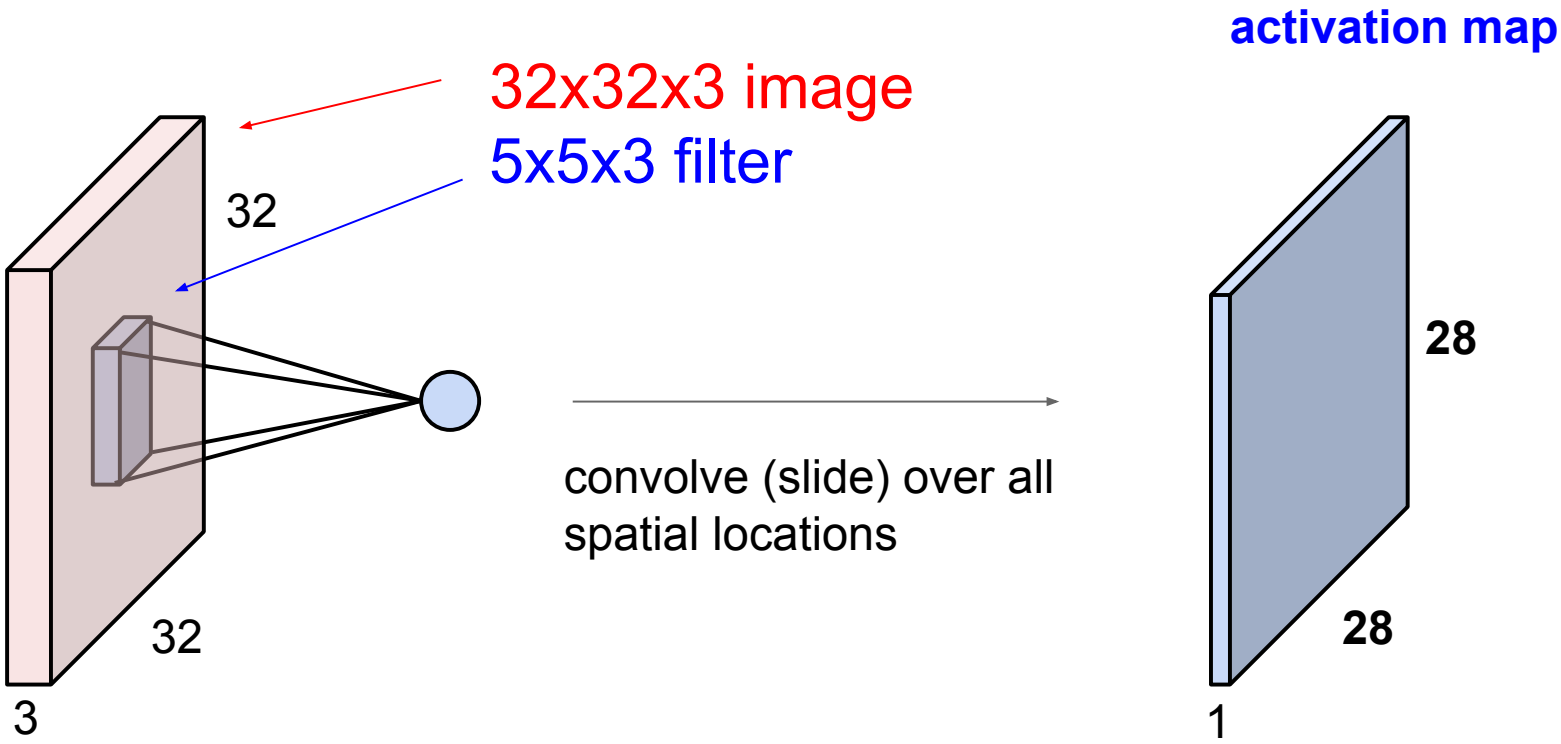
example 5x5 filters
(32 total)

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

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

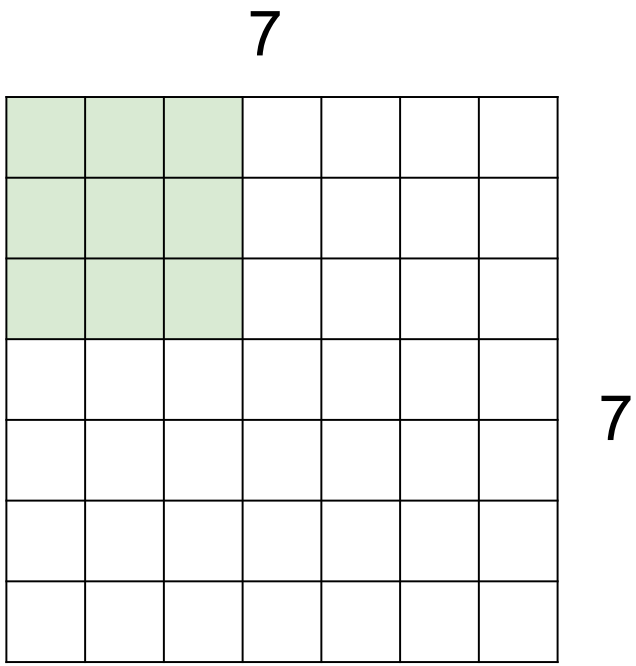
elementwise multiplication and sum of a filter and the signal (image)

A closer look at spatial dimensions:



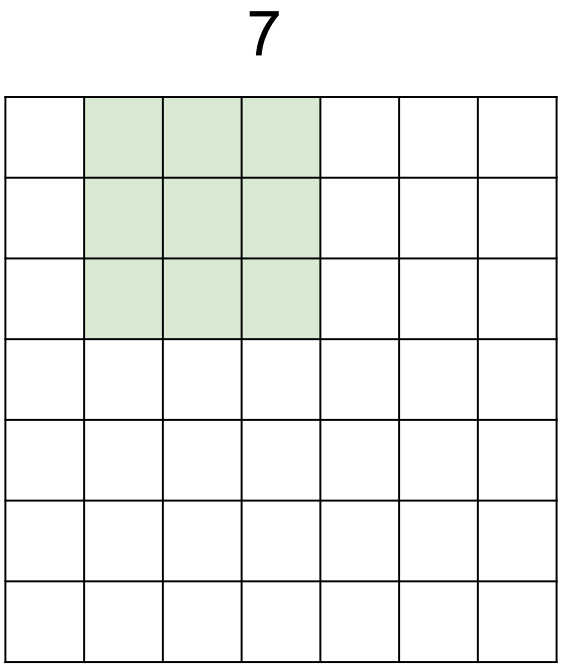
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 7 - 23 27 Jan 2016

A closer look at spatial dimensions:



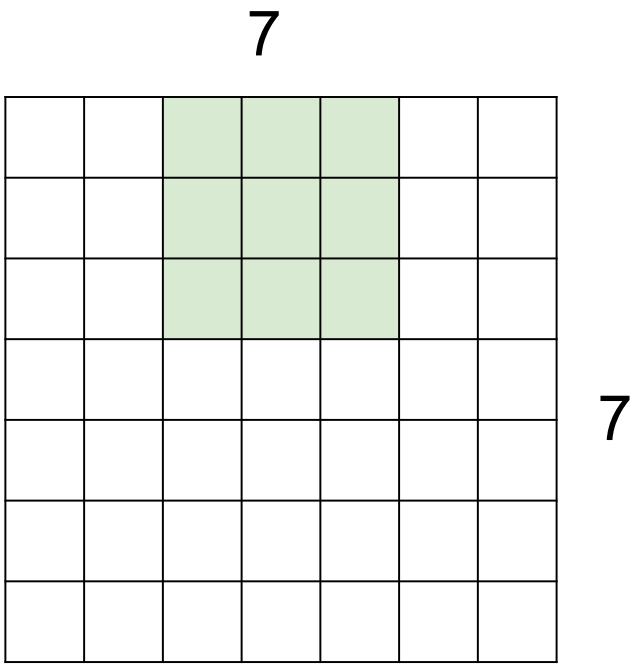
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



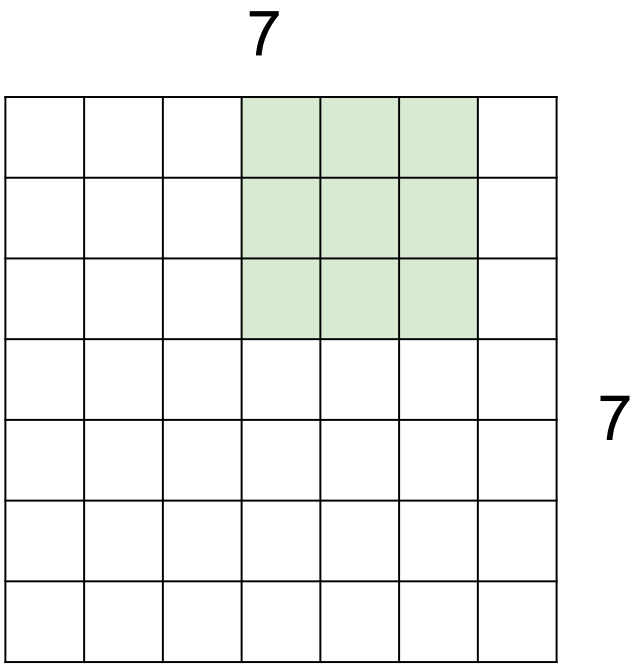
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



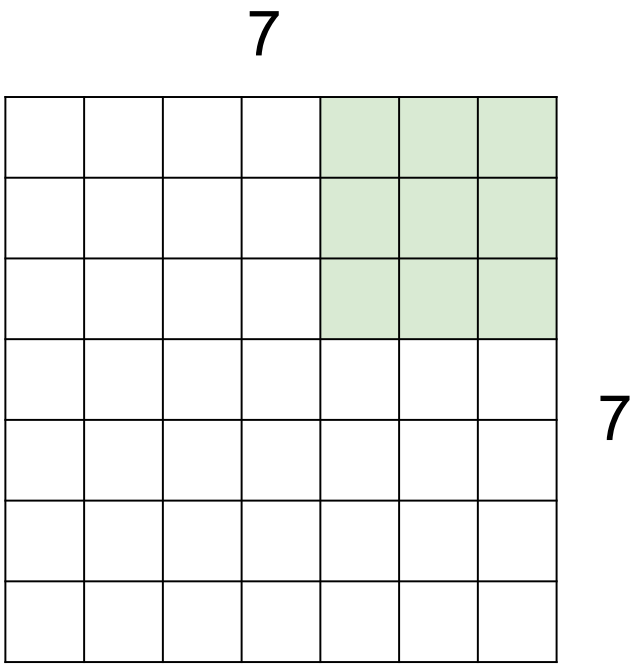
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

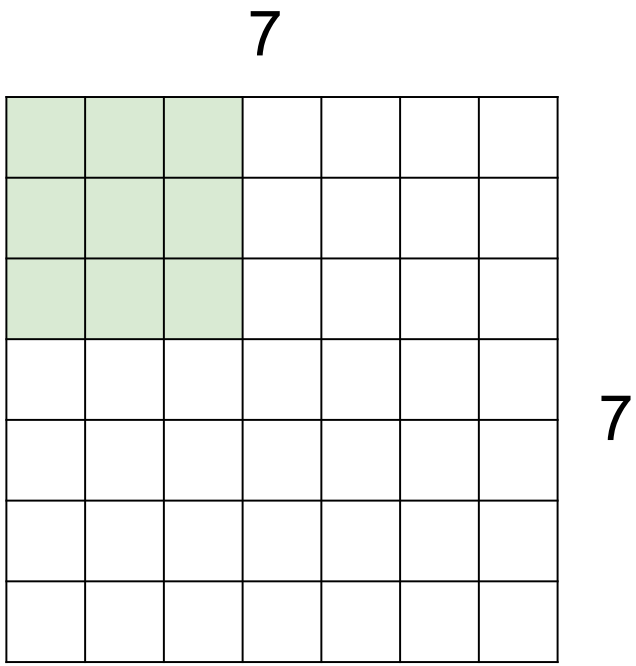
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

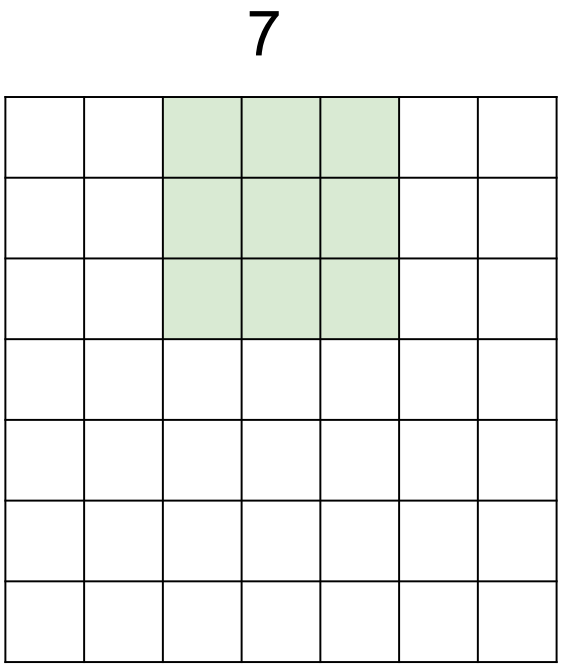
=> 5x5 output

A closer look at spatial dimensions:



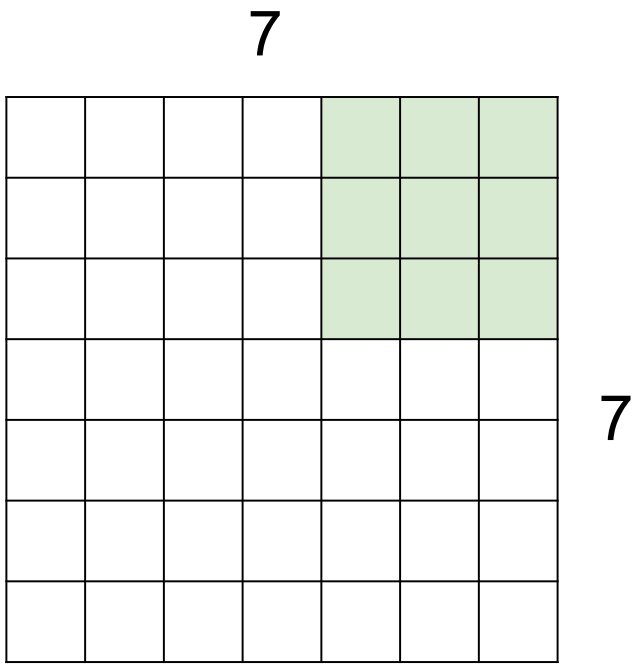
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



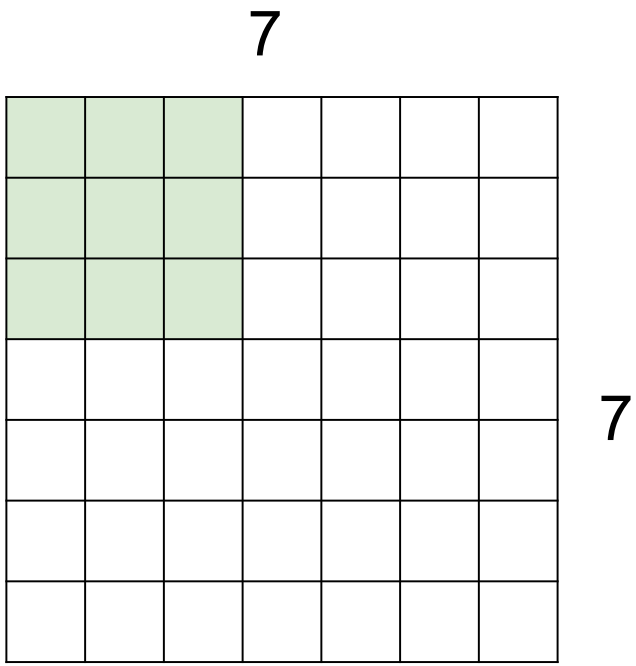
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



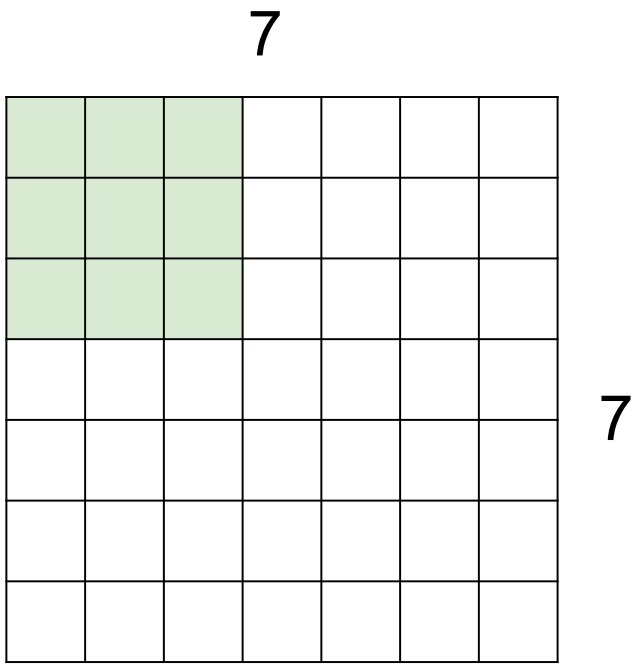
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



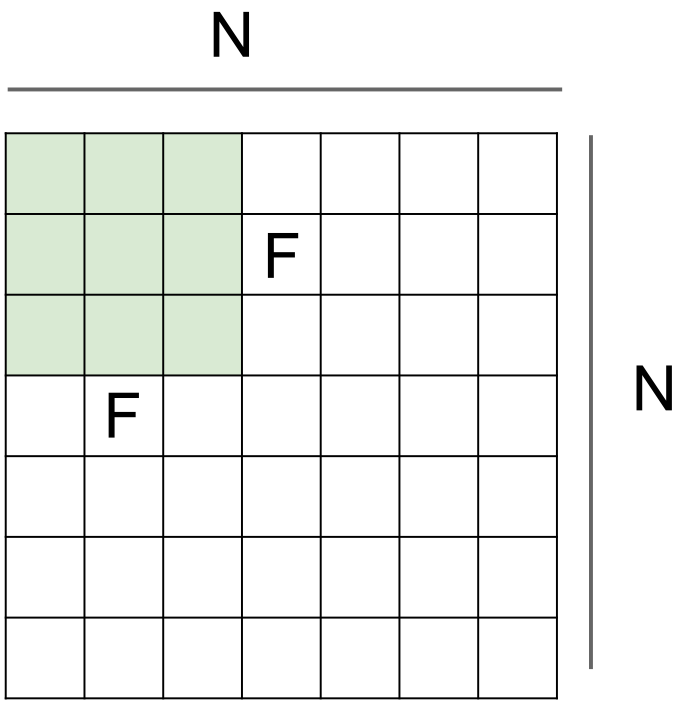
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



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

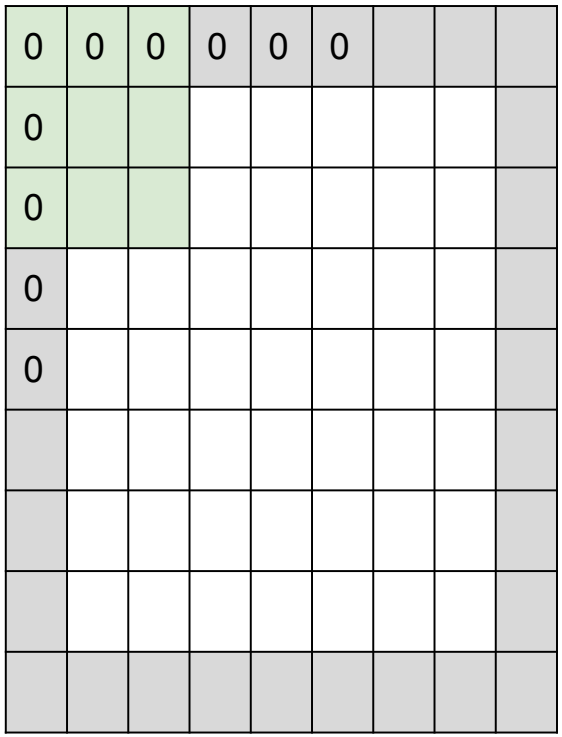
doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

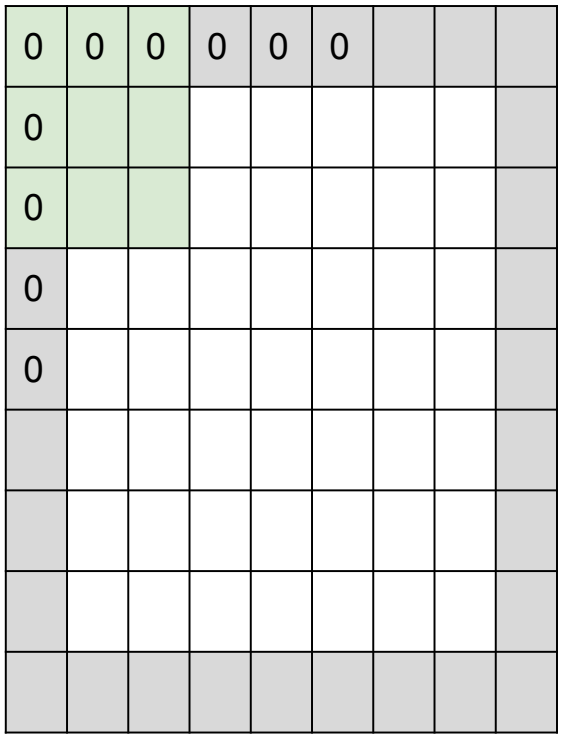
In practice: Common to zero pad the border



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

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

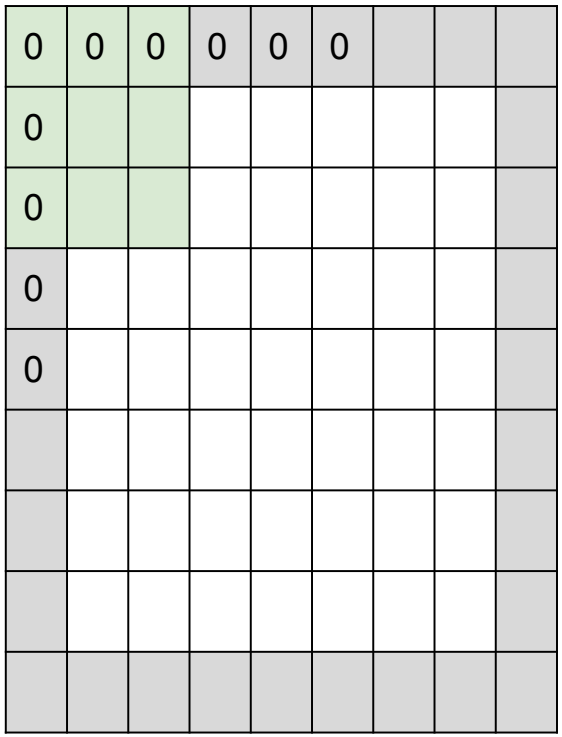
In practice: Common to zero pad the border



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

7x7 output!

In practice: Common to zero pad the border

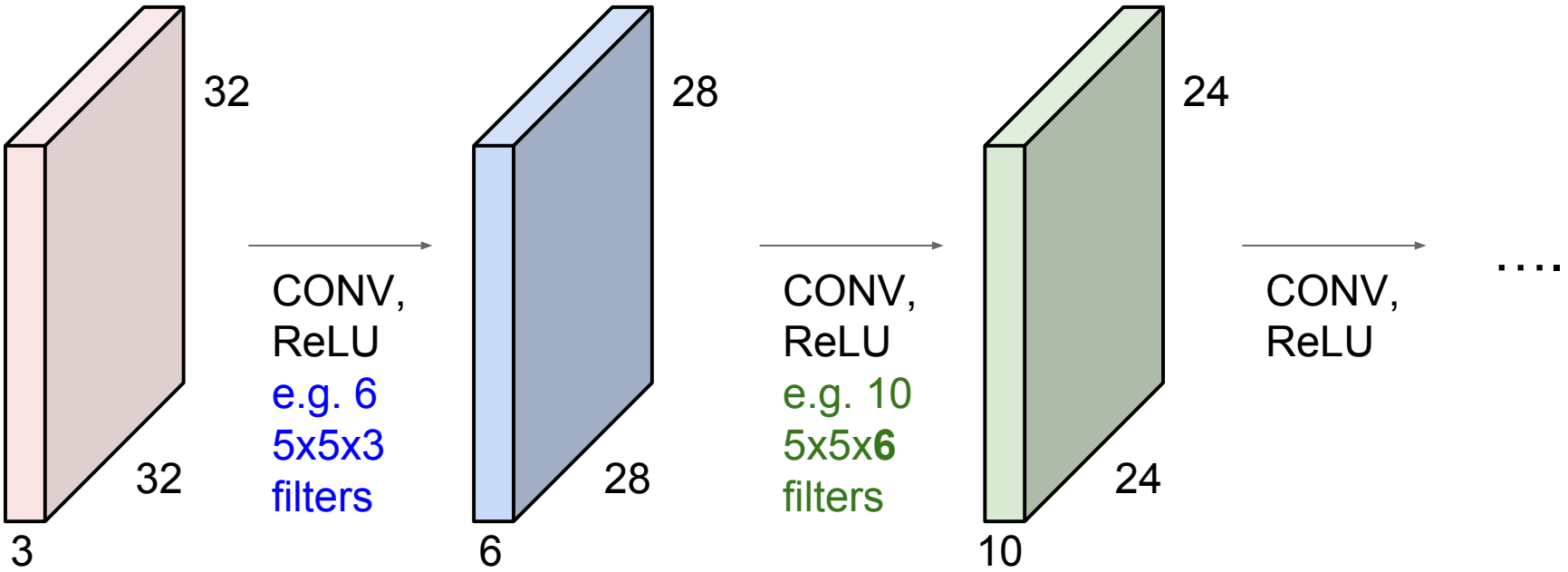


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

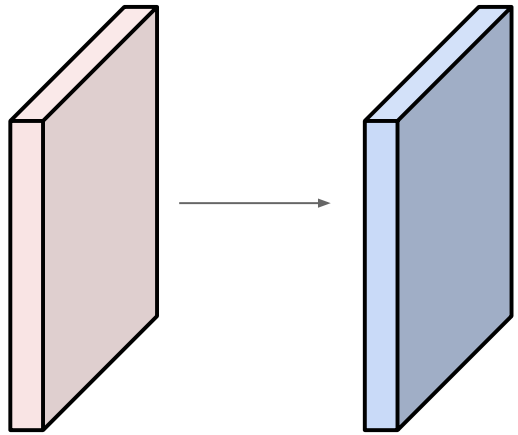
7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Remember back to…

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.



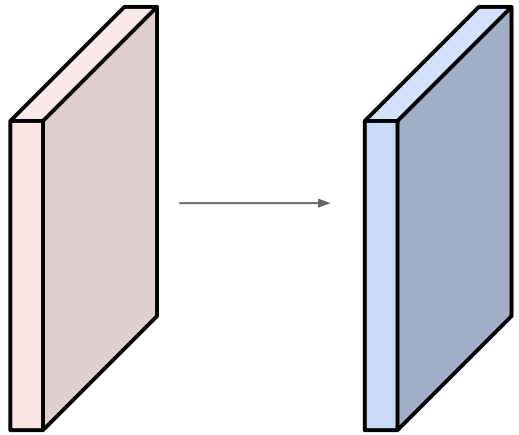
Examples time:



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

Output volume size: ?

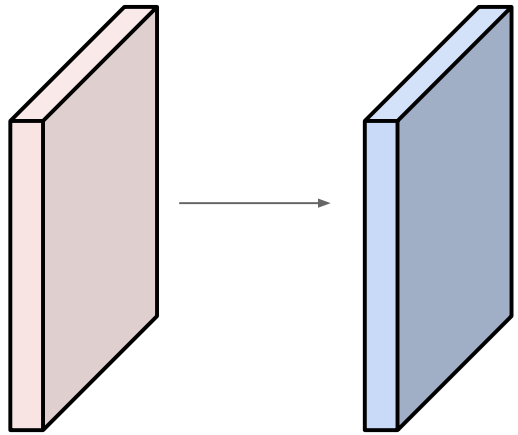
Examples time:



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

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

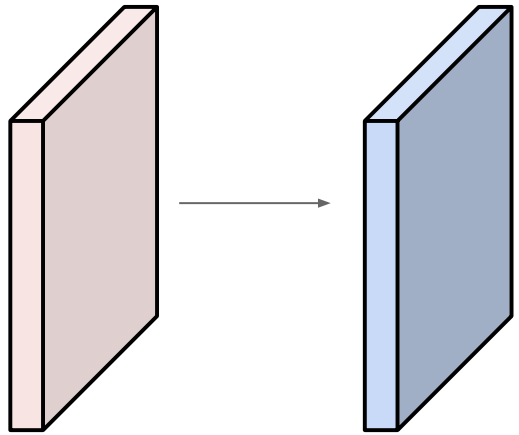
Examples time:



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

Number of parameters in this layer?

Examples time:



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

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

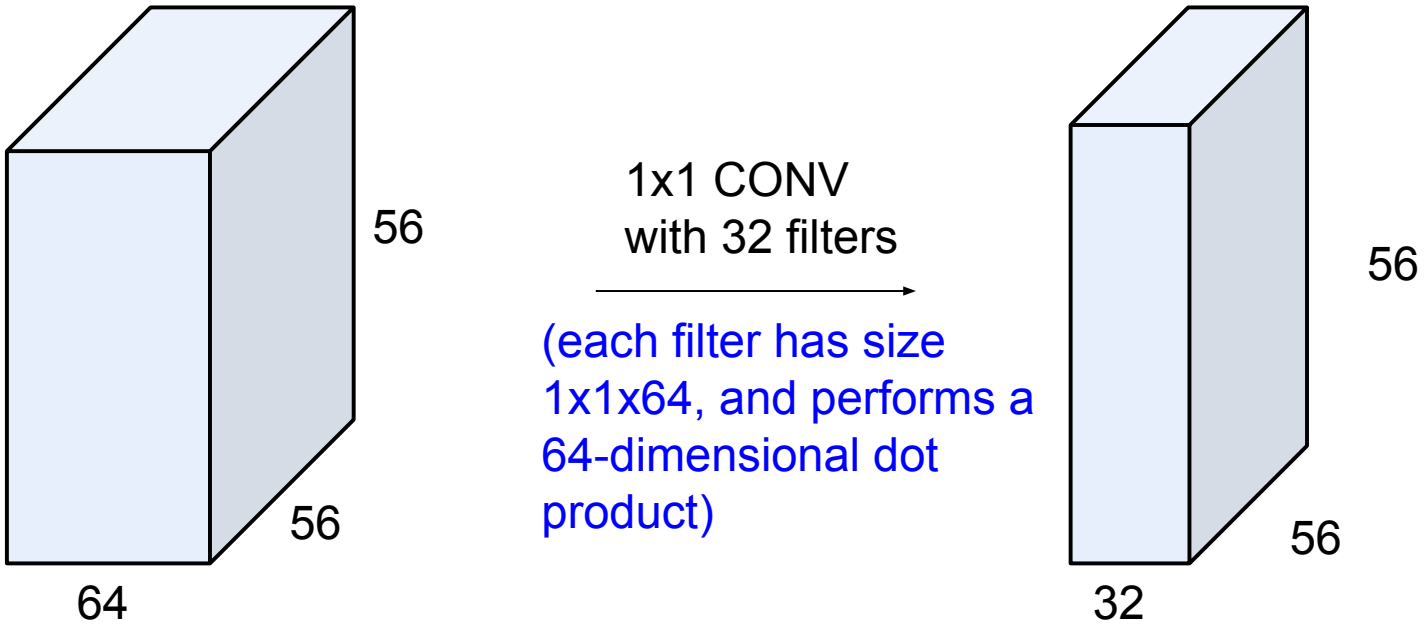
Common settings:

K = (powers of 2, e.g. 32, 64, 128, 512)
- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

(btw, 1x1 convolution layers make perfect sense)



Example: CONV layer in Torch

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.

SpatialConvolution

```
module = nn.SpatialConvolution(nInputPlane, nOutputPlane, kW, kH, [dW], [dH], [padW], [padH])
```

Applies a 2D convolution over an input image composed of several input planes. The `input` tensor in `forward(input)` is expected to be a 3D tensor (`nInputPlane x height x width`).

The parameters are the following:

- `nInputPlane` : The number of expected input planes in the image given into `forward()` .
- `nOutputPlane` : The number of output planes the convolution layer will produce.
- `kW` : The kernel width of the convolution
- `kH` : The kernel height of the convolution
- `dW` : The step of the convolution in the width dimension. Default is `1` .
- `dH` : The step of the convolution in the height dimension. Default is `1` .
- `padW` : The additional zeros added per width to the input planes. Default is `0` , a good number is `(kW-1)/2` .
- `padH` : The additional zeros added per height to the input planes. Default is `padW` , a good number is `(kH-1)/2` .

Note that depending of the size of your kernel, several (of the last) columns or rows of the input image might be lost. It is up to the user to add proper padding in images.

If the input image is a 3D tensor `nInputPlane x height x width` , the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth  = floor((width  + 2*padW - kW) / dW + 1)
oheight = floor((height + 2*padH - kH) / dH + 1)
```

Example: CONV layer in Caffe

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  # learning rate and decay multipliers for the filters
  param { lr_mult: 1 decay_mult: 1 }
  # learning rate and decay multipliers for the biases
  param { lr_mult: 2 decay_mult: 0 }
  convolution_param {
    num_output: 96     # learn 96 filters
    kernel_size: 11    # each filter is 11x11
    stride: 4          # step 4 pixels between each filter application
    weight_filler {
      type: "gaussian" # initialize the filters from a Gaussian
      std: 0.01        # distribution with stdev 0.01 (default mean: 0)
    }
    bias_filler {
      type: "constant" # initialize the biases to zero (0)
      value: 0
    }
  }
}
```

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 7 - 47 27 Jan 2016

Example: CONV layer in Lasagne

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.

class **lasagne.layers.Conv2DLayer**(*incoming, num_filters, filter_size, stride=(1, 1), pad=0, untie_biases=False, W=lasagne.init.GlorotUniform(), b=lasagne.init.Constant(0.), nonlinearity=lasagne.nonlinearities.rectify, flip_filters=True, convolution=theano.tensor.nnet.conv2d, **kwargs*) [source]

2D convolutional layer

Performs a 2D convolution on its input and optionally adds a bias and applies an elementwise nonlinearity.

Parameters:

incoming : a `Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape. The output of this layer should be a 4D tensor, with shape `(batch_size, num_input_channels, input_rows, input_columns)`.

num_filters : int

The number of learnable convolutional filters this layer has.

filter_size : int or iterable of int

An integer or a 2-element tuple specifying the size of the filters.

stride : int or iterable of int

An integer or a 2-element tuple specifying the stride of the convolution operation.

pad : int, iterable of int, 'full', 'same' or 'valid' (default: 0)

By default, the convolution is only computed where the input and the filter fully overlap (a valid convolution). When `stride=1`, this yields an output that is smaller than the input by `filter_size - 1`. The *pad* argument allows you to implicitly pad the input with zeros, extending the output size.

A single integer results in symmetric zero-padding of the given size on all borders, a tuple of two integers allows different symmetric padding per dimension.

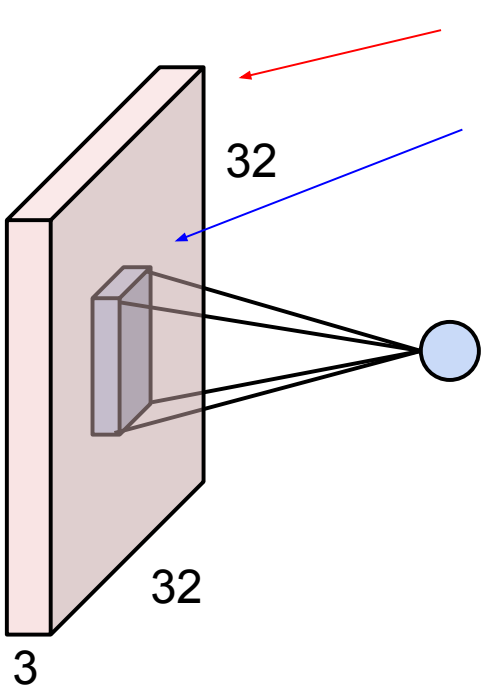
`'full'` pads with one less than the filter size on both sides. This is equivalent to computing the convolution wherever the input and the filter overlap by at least one position.

`'same'` pads with half the filter size (rounded down) on both sides. When `stride=1` this results in an output size equal to the input size. Even filter size is not supported.

`'valid'` is an alias for `0` (no padding / a valid convolution).

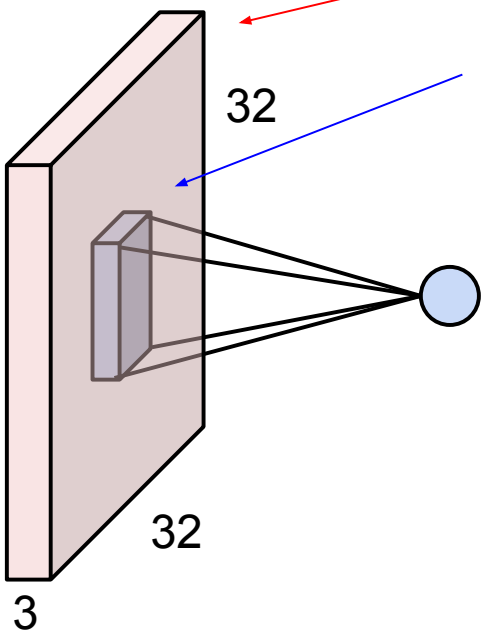
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 7 - 48 27 Jan 2016

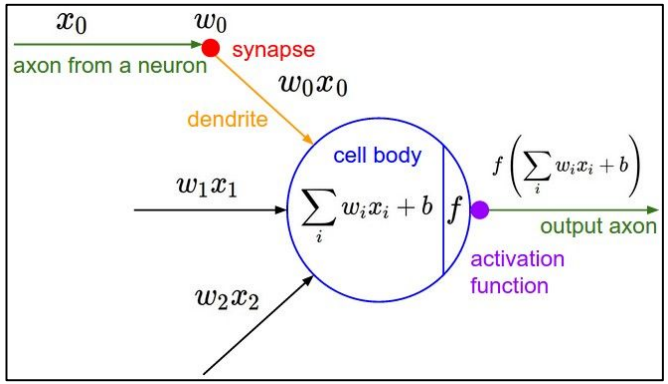
The brain/neuron view of CONV Layer



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)

The brain/neuron view of CONV Layer

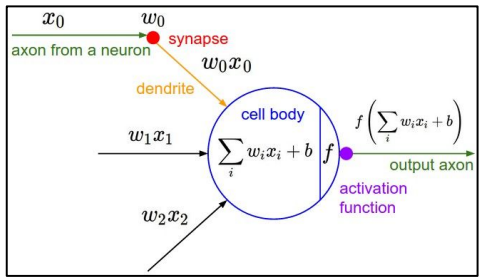


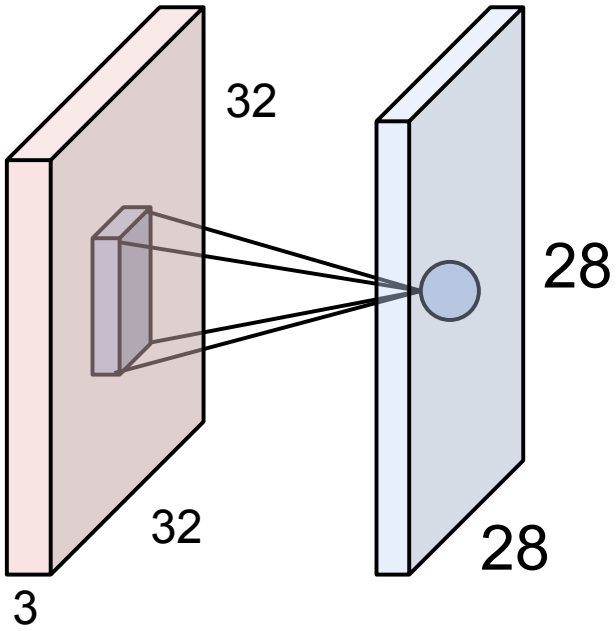


It's just a neuron with local connectivity...

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)

The brain/neuron view of CONV Layer



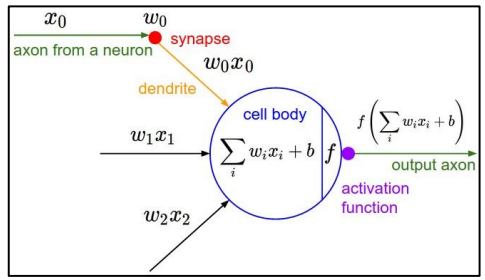


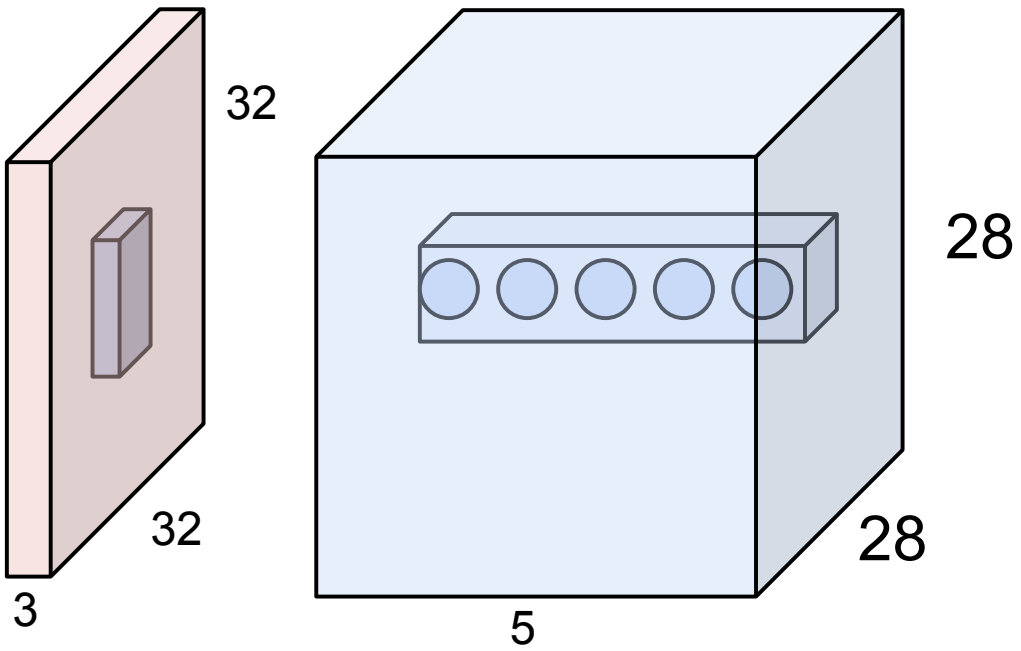
An activation map is a 28x28 sheet of neuron outputs:

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

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

The brain/neuron view of CONV Layer

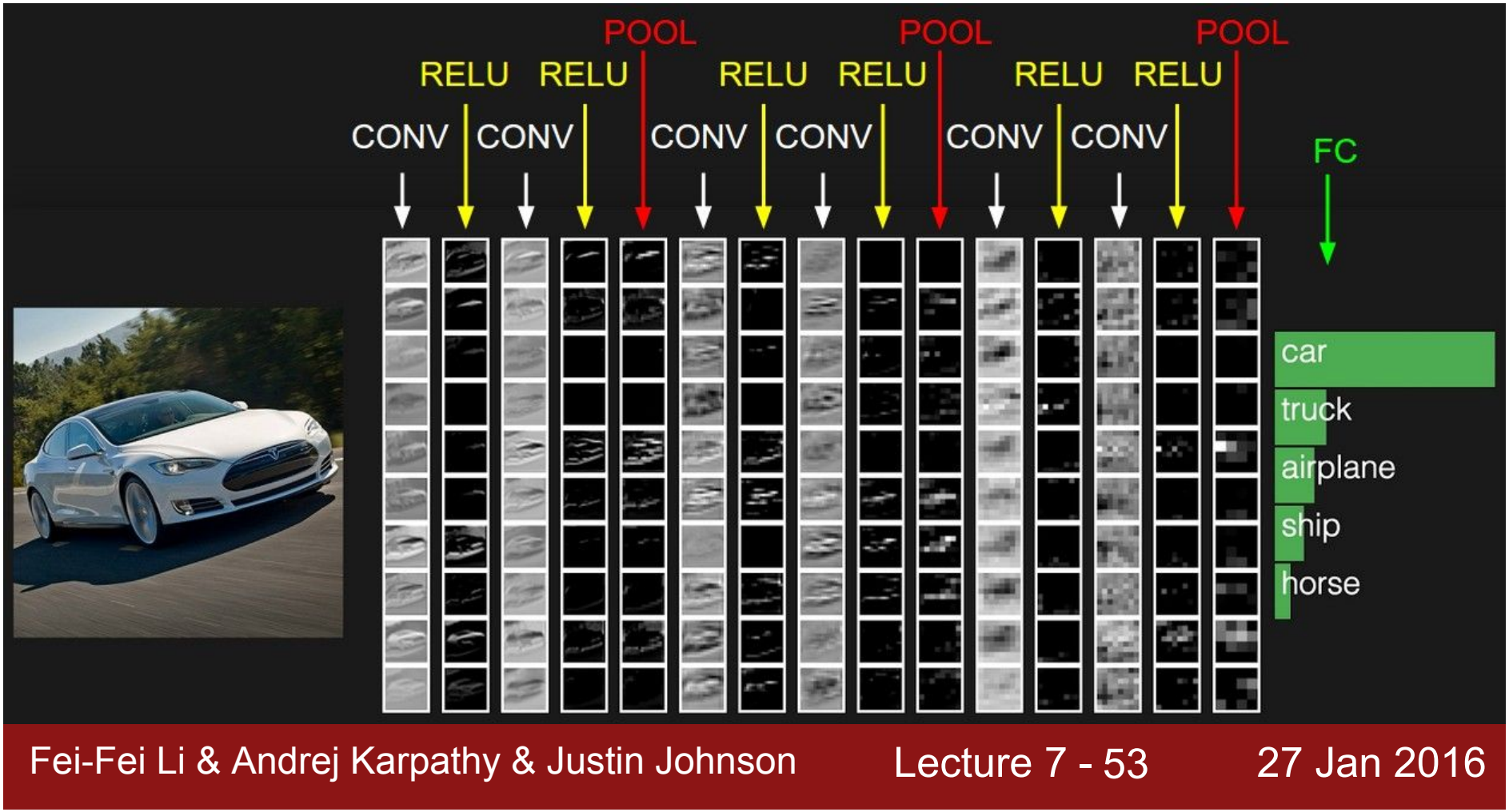




E.g. with 5 filters,
CONV layer consists of neurons arranged in a 3D grid (28x28x5)

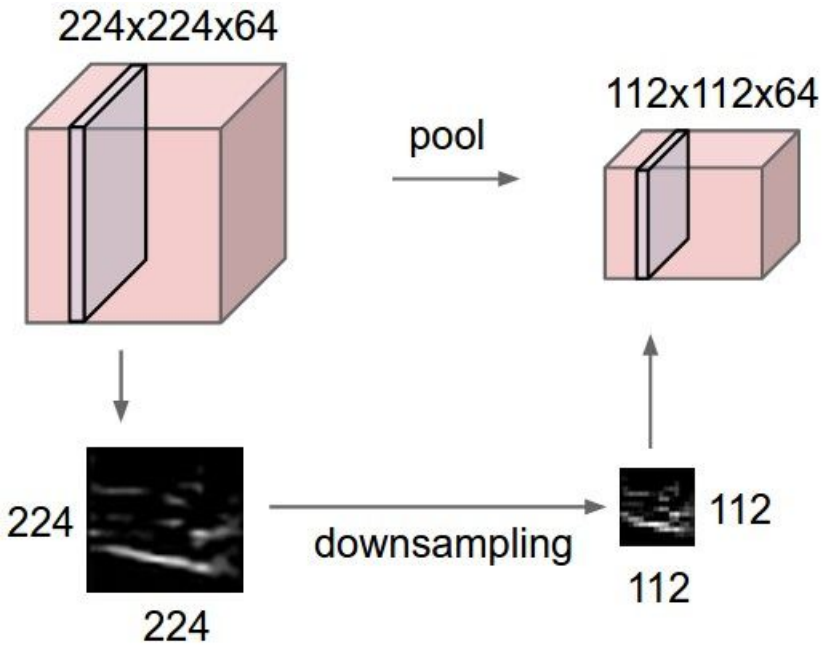
There will be 5 different neurons all looking at the same region in the input volume

two more layers to go: POOL/FC



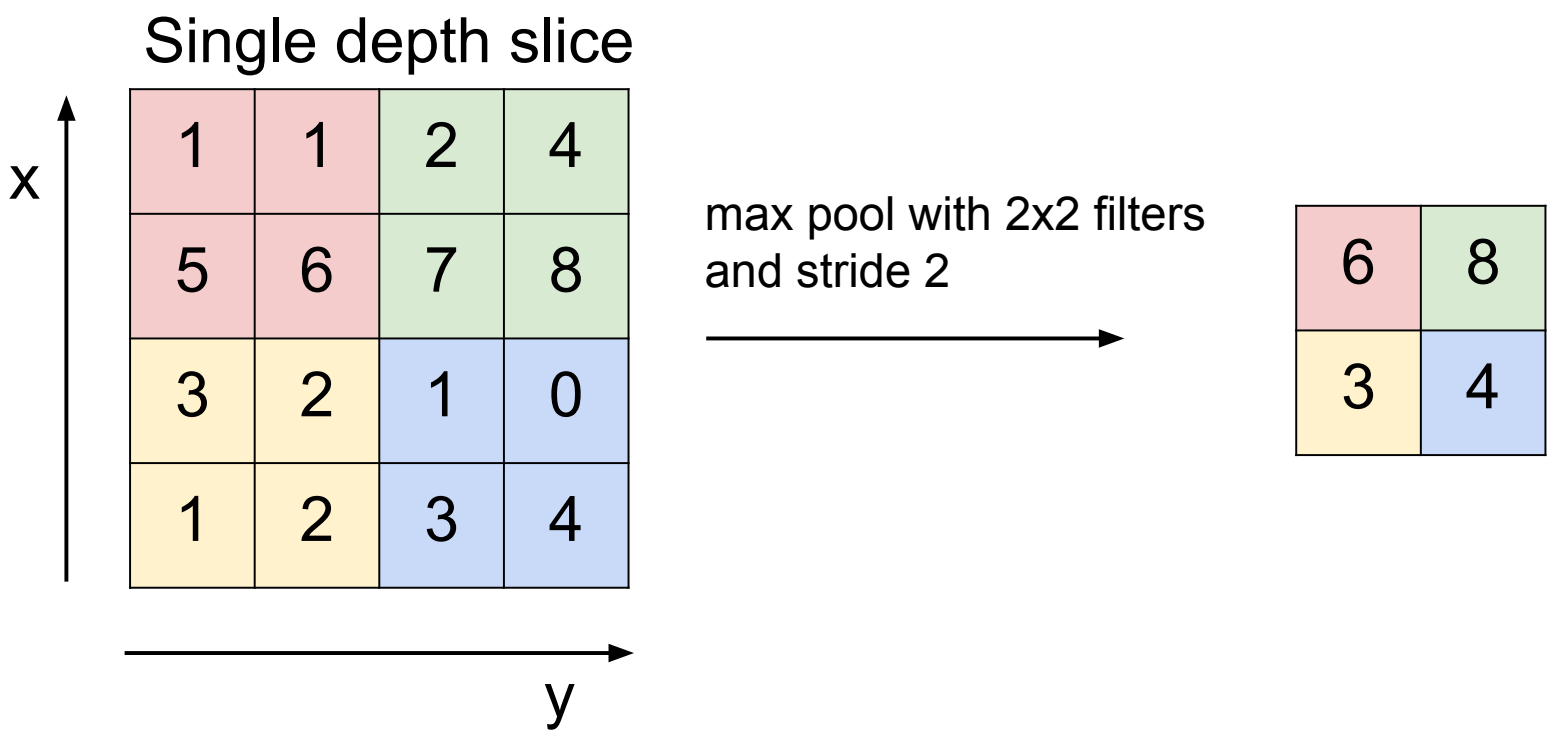
Pooling layer

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



Fei-Fei Li & Andrej Karpathy & Justin Johnson | Lecture 7 - 54 | 27 Jan 2016

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

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

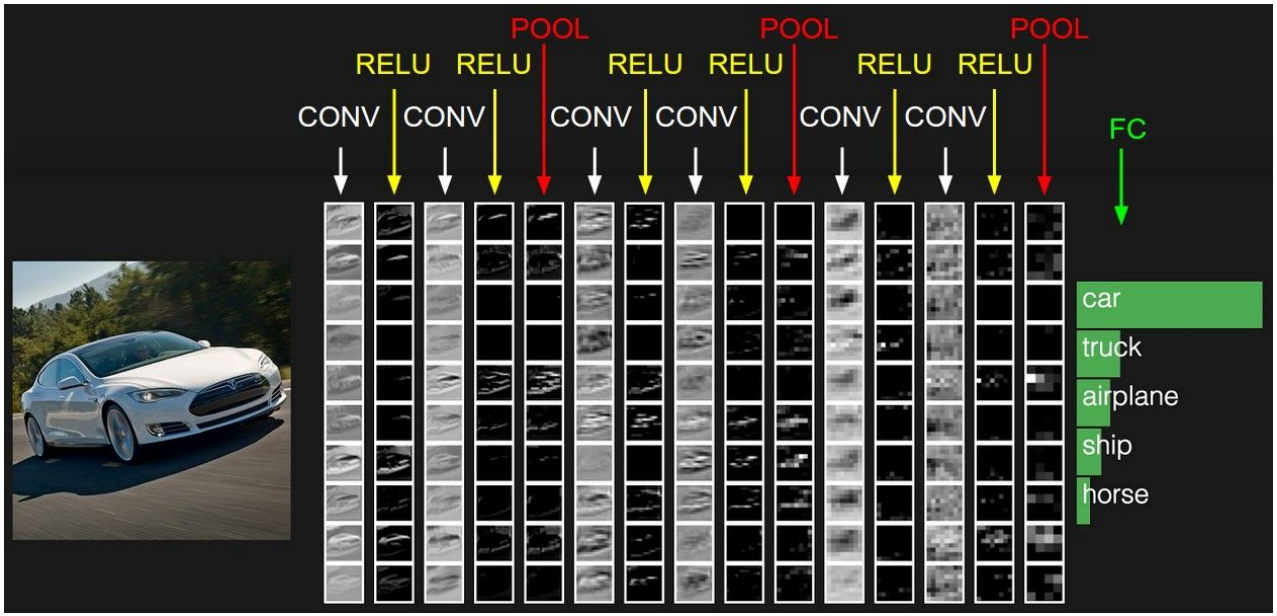
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

Common settings:

F = 2, S = 2
F = 3, S = 2

Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



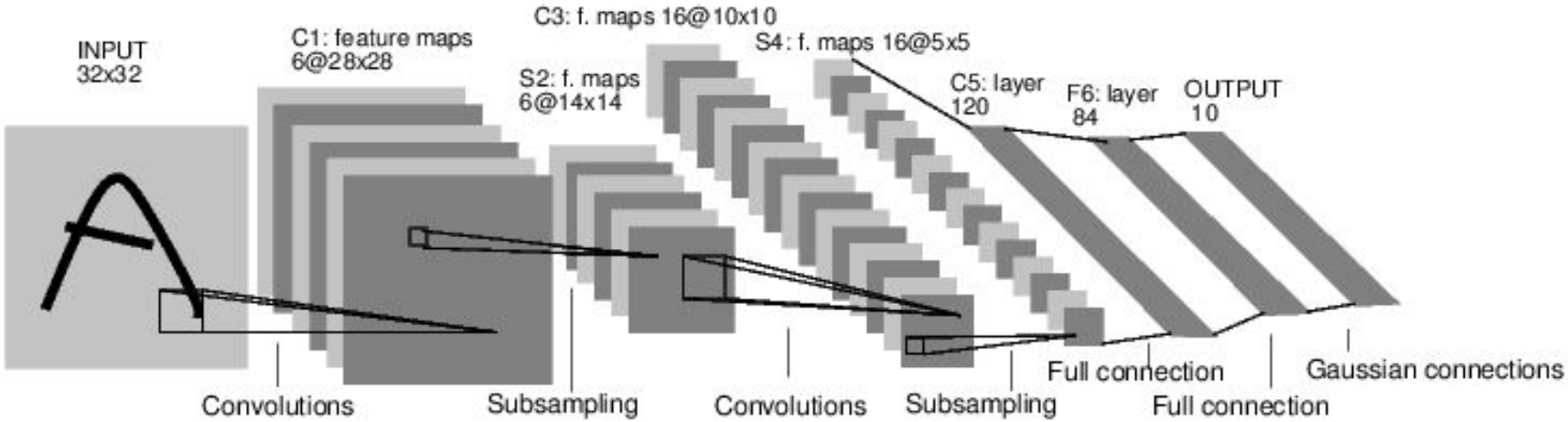
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 7 - 58 27 Jan 2016

Demo

ConvNetJS cifar10 demo

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]

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]

Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8

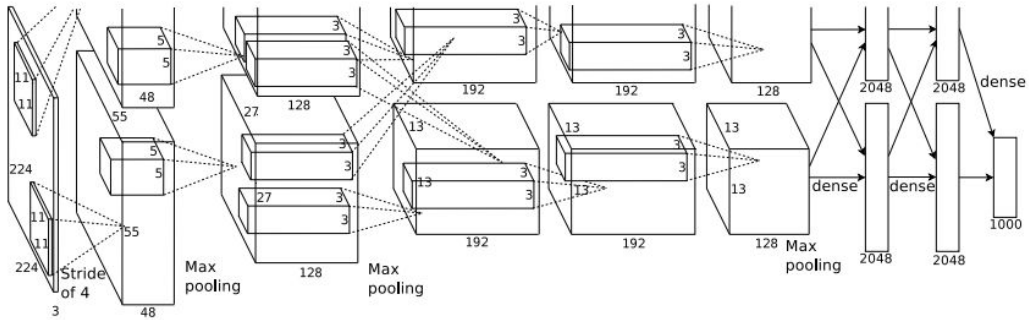
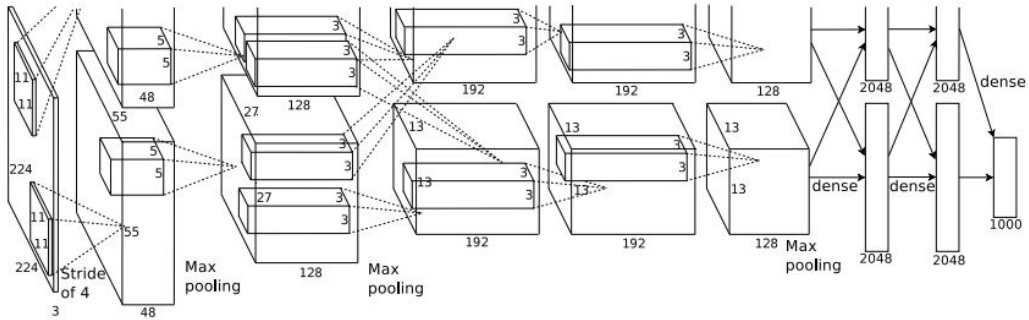


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



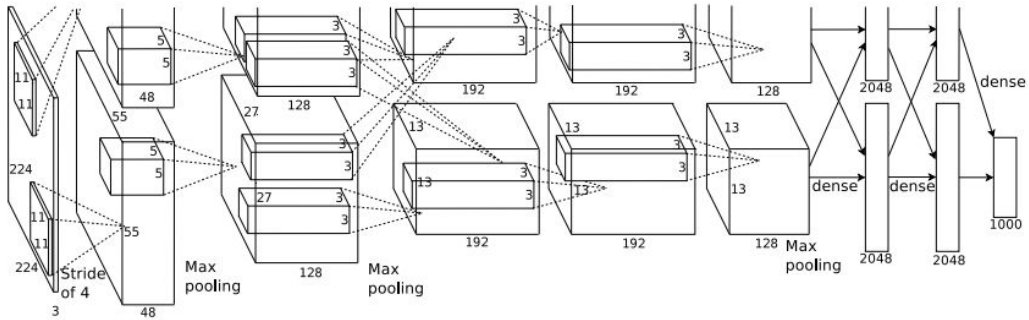
Input: 227x227x3 images

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

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

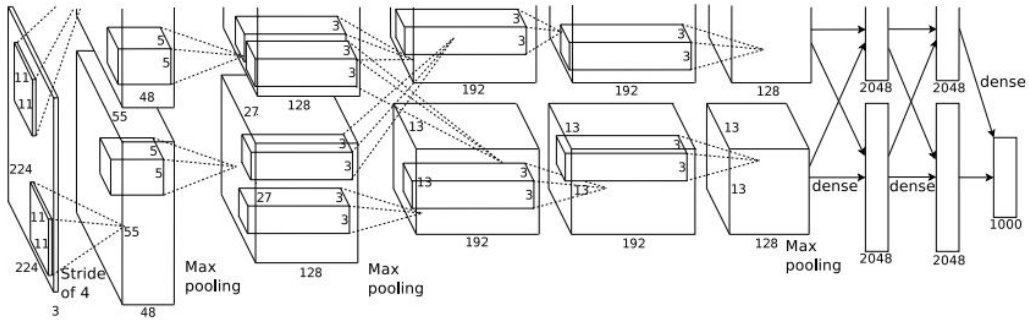
First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**

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

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



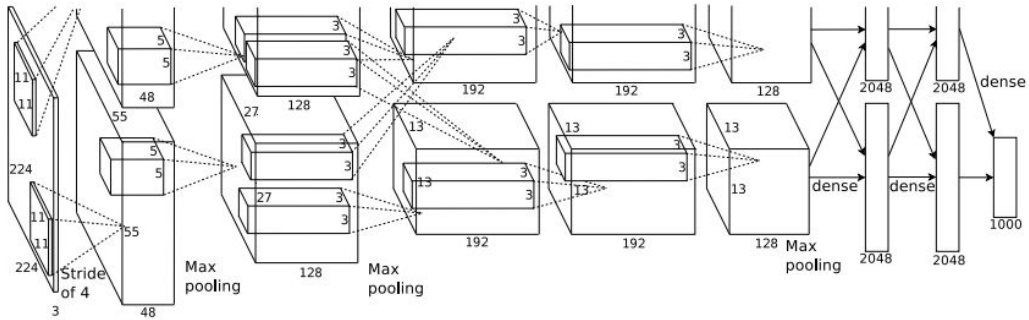
Input: 227x227x3 images

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

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images
After CONV1: 55x55x96

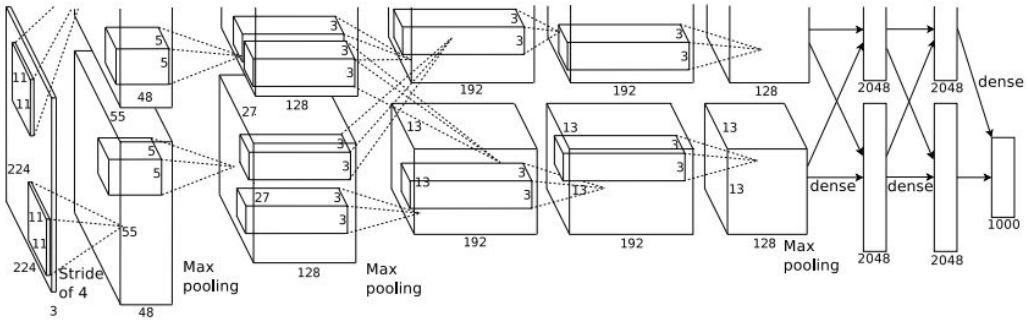
Second layer (POOL1): 3x3 filters applied at stride 2

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

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images
After CONV1: 55x55x96

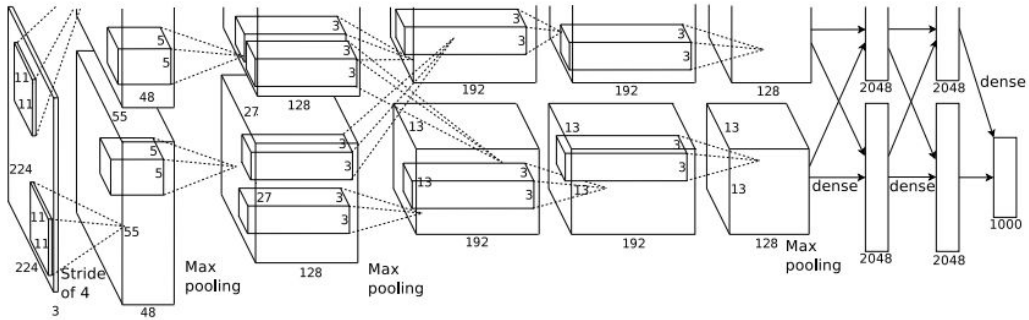
Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96

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

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



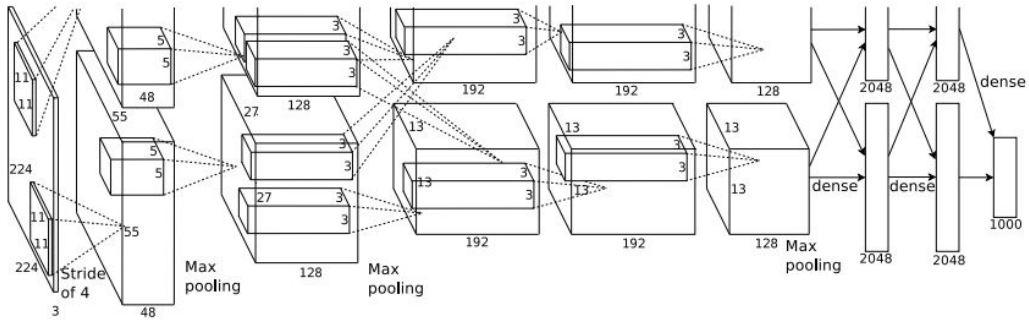
Input: 227x227x3 images
After CONV1: 55x55x96

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

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



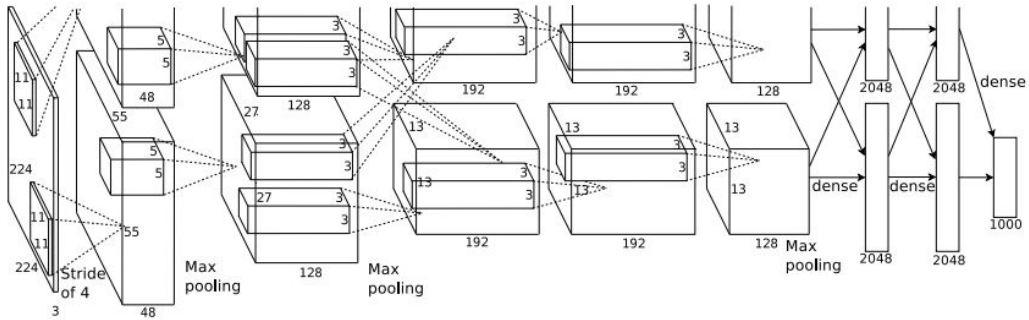
Input: 227x227x3 images
After CONV1: 55x55x96
After POOL1: 27x27x96
...

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]

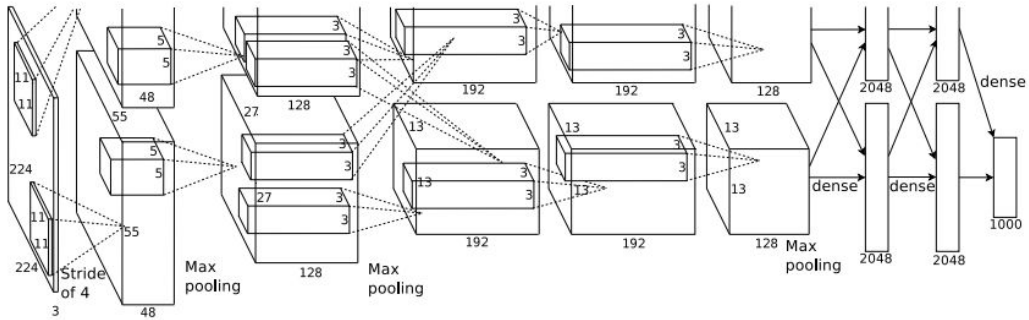


Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] 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)

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



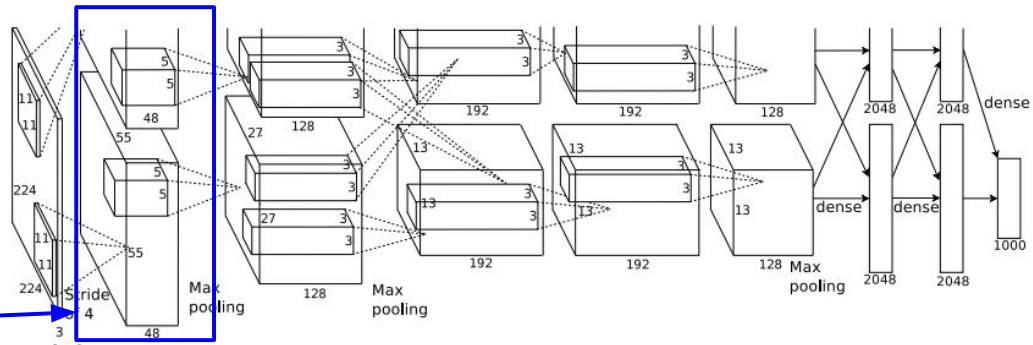
Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] 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%

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



[55x55x48] x 2

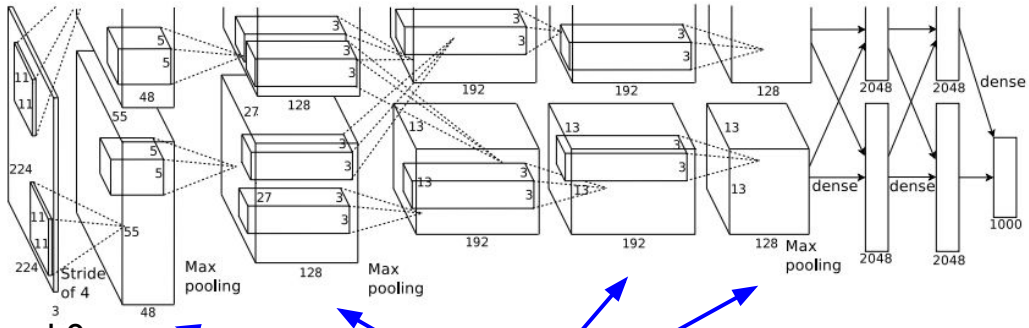
Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] 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)

Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

AlexNet

Case Study: AlexNet

[Krizhevsky et al. 2012]



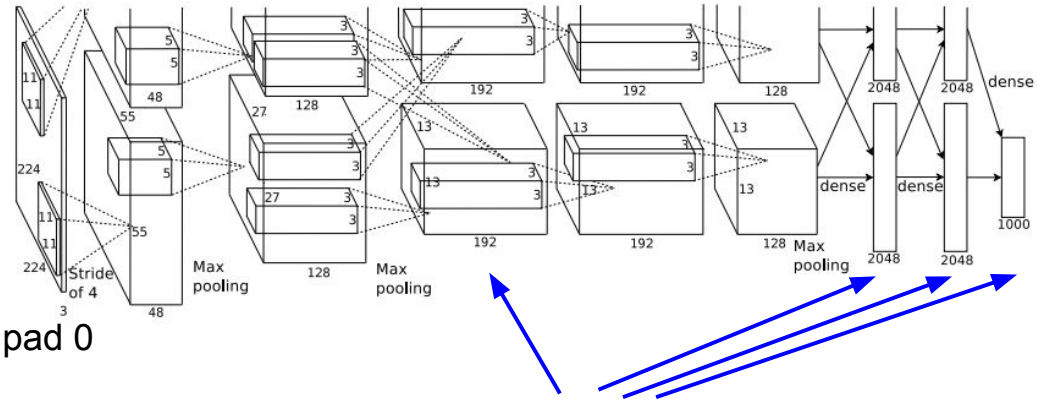
Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] 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)

CONV1, CONV2, CONV4, CONV5:
Connections only with feature maps on same GPU

AlexNet

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)

CONV3, FC6, FC7, FC8:
Connections with all feature maps in preceding layer, communication across GPUs

AlexNet

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

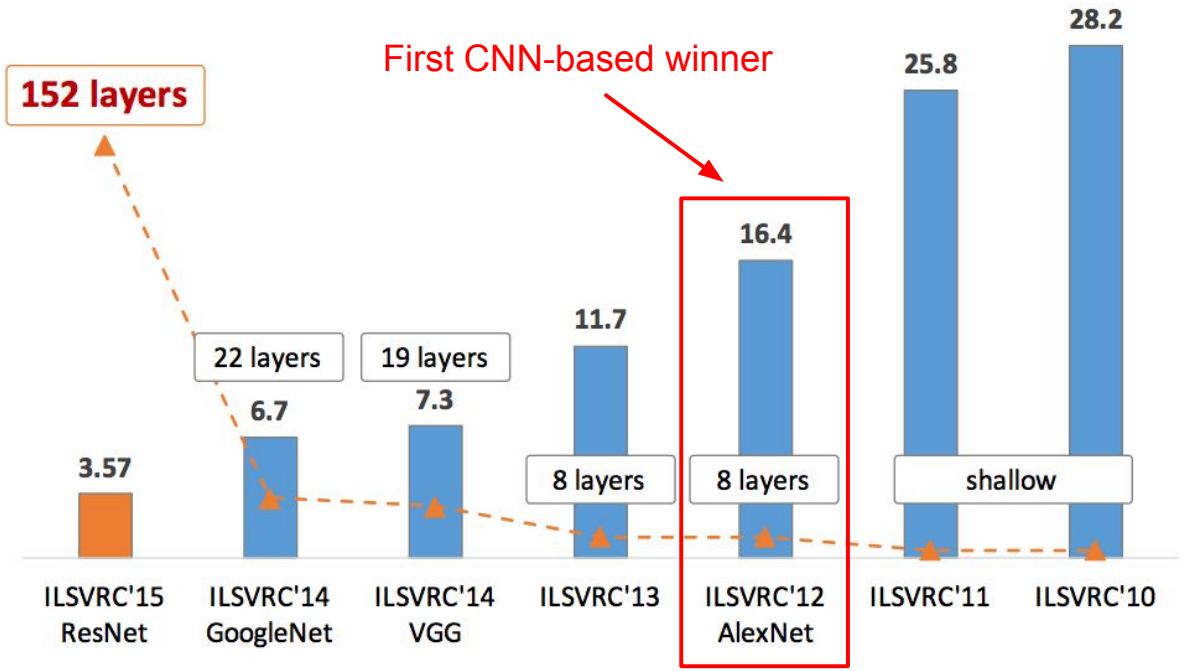


Figure copyright Kaiming He, 2016. Reproduced with permission.

ZFNet

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

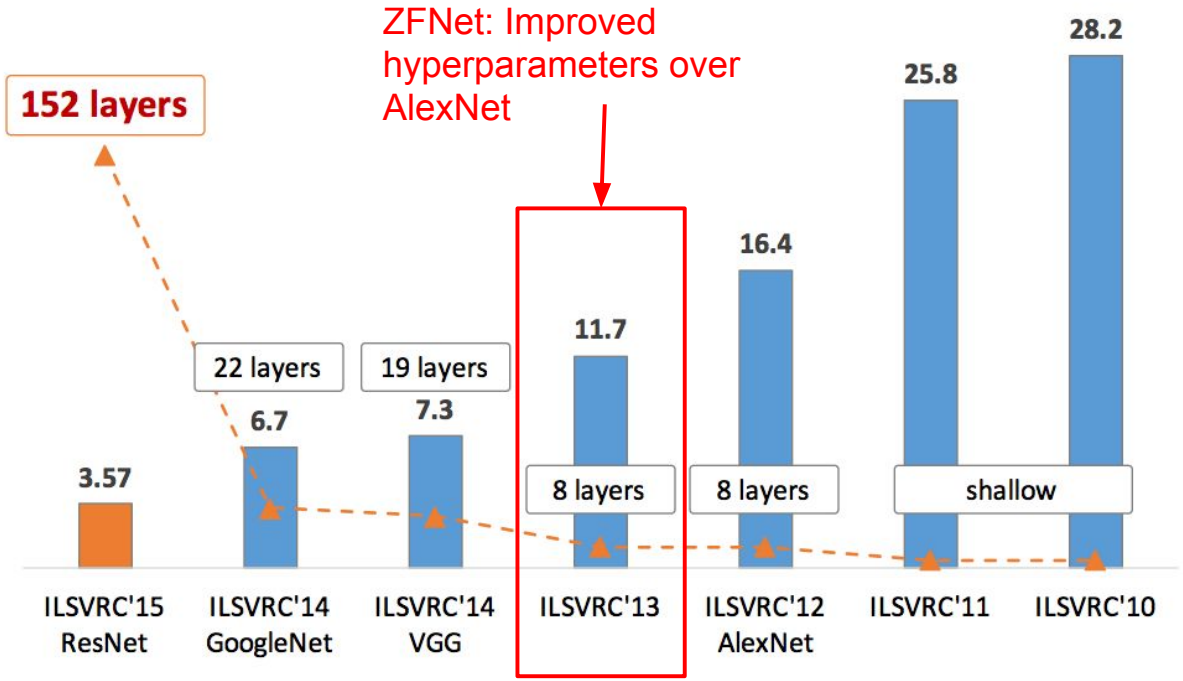
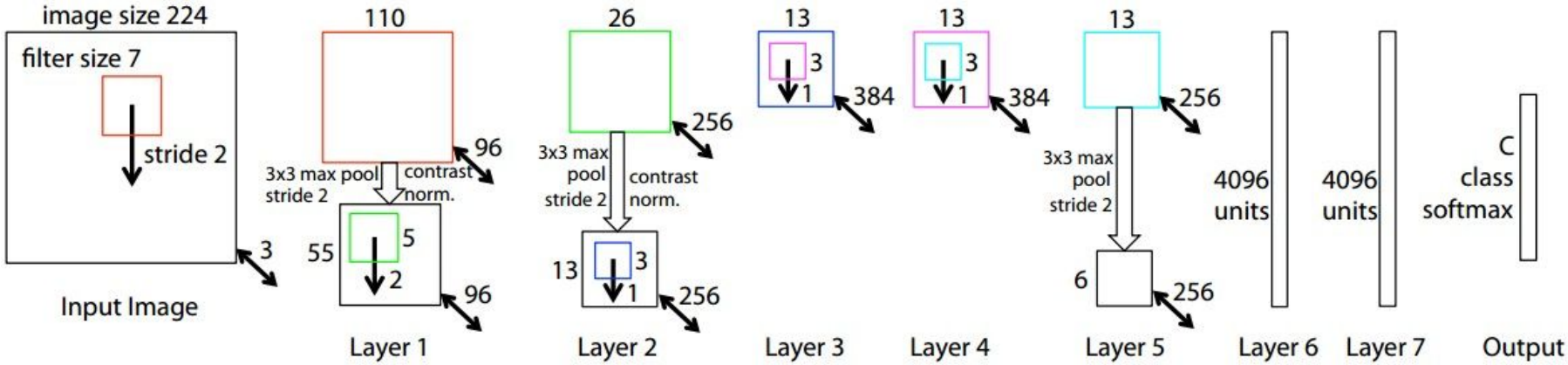


Figure copyright Kaiming He, 2016. Reproduced with permission.

ZFNet

ZFNet

[Zeiler and Fergus, 2013]



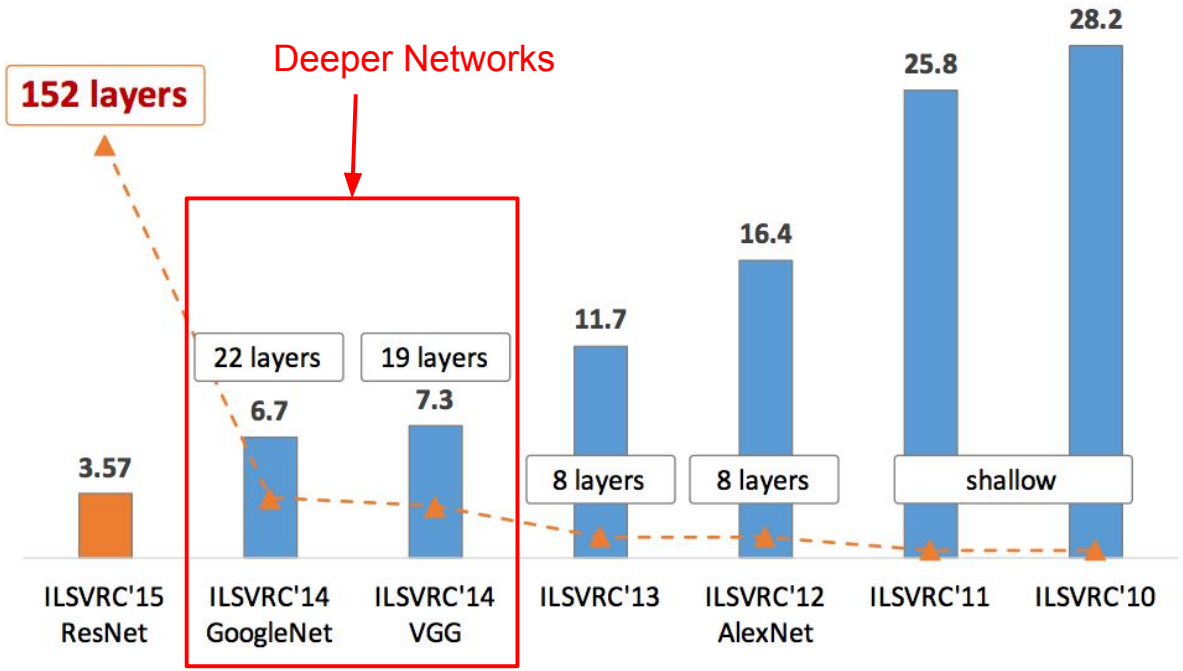
TODO: remake figure

AlexNet but:
CONV1: change from (11x11 stride 4) to (7x7 stride 2)
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 16.4% -> 11.7%

VGGNet

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



VGGNet

Case Study: VGGNet

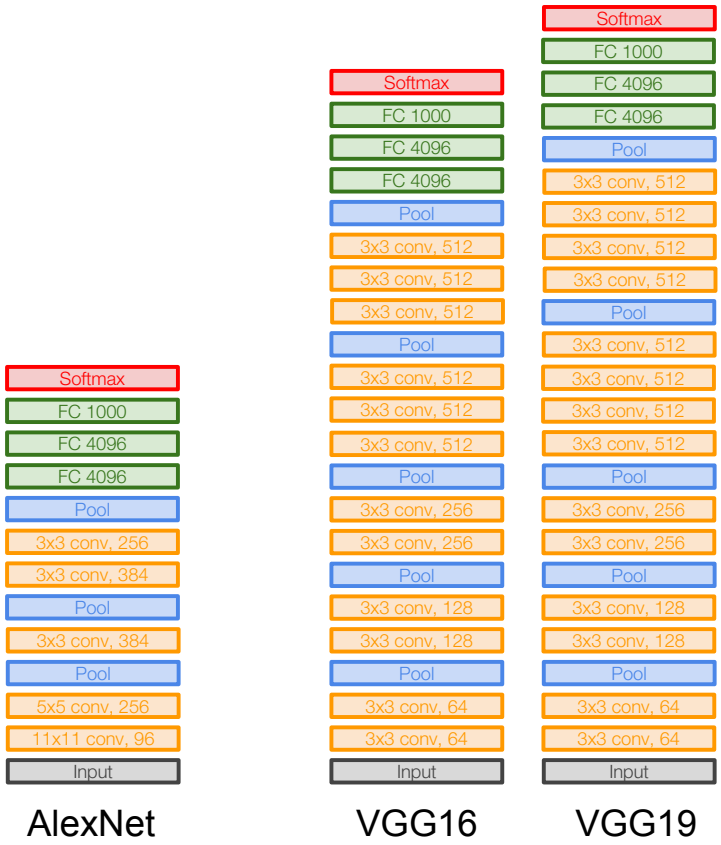
[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

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

11.7% top 5 error in ILSVRC'13
(ZFNet)
-> 7.3% top 5 error in ILSVRC'14

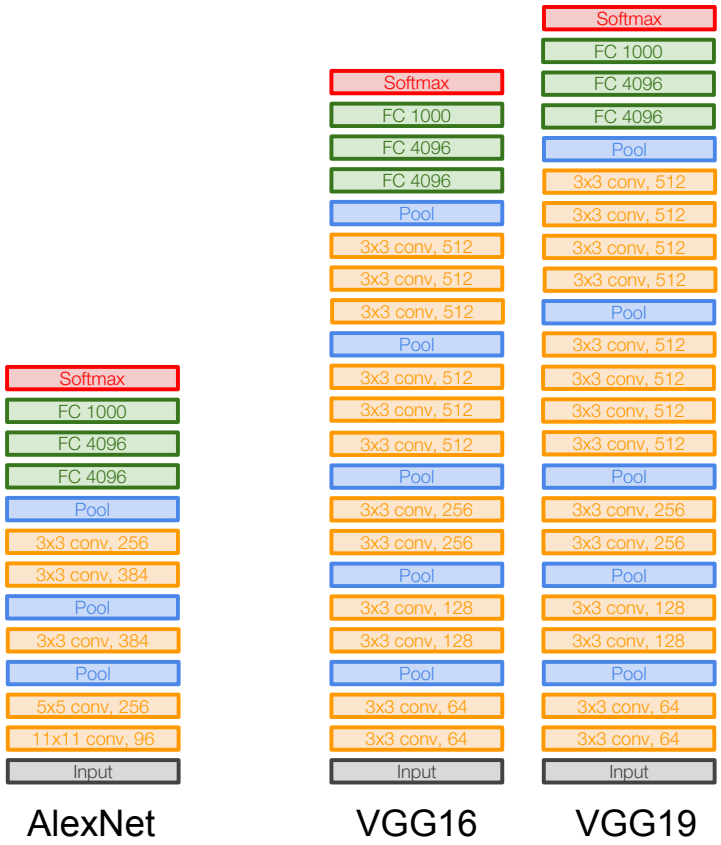


VGGNet

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)



VGGNet

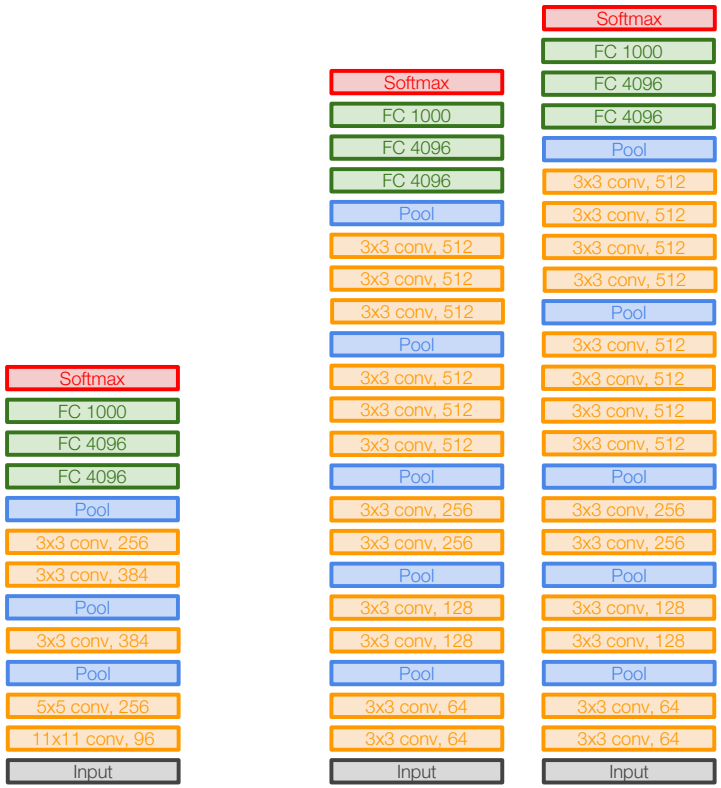
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?



VGGNet

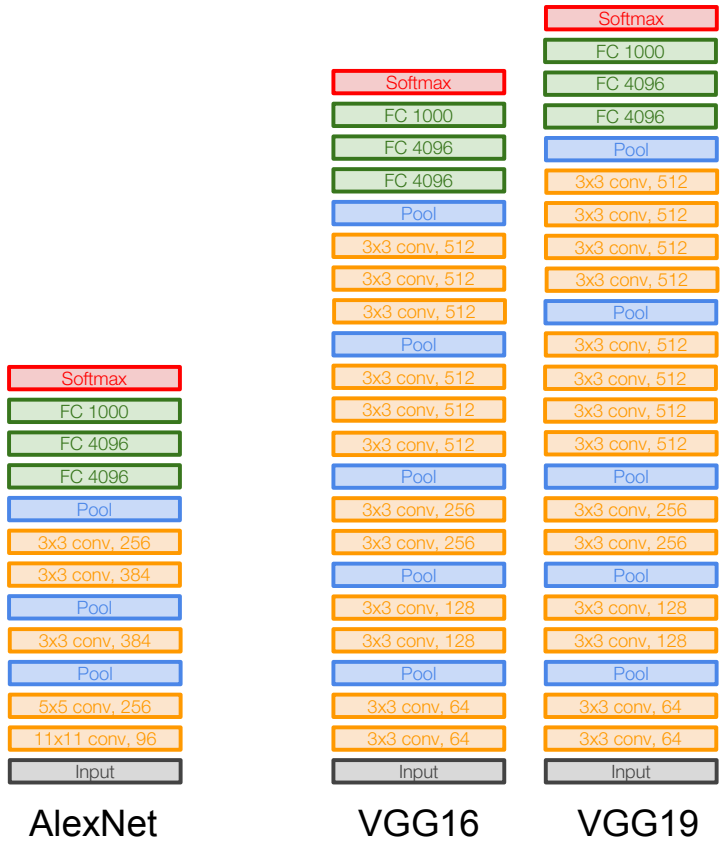
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

[7x7]



VGGNet

Case Study: VGGNet

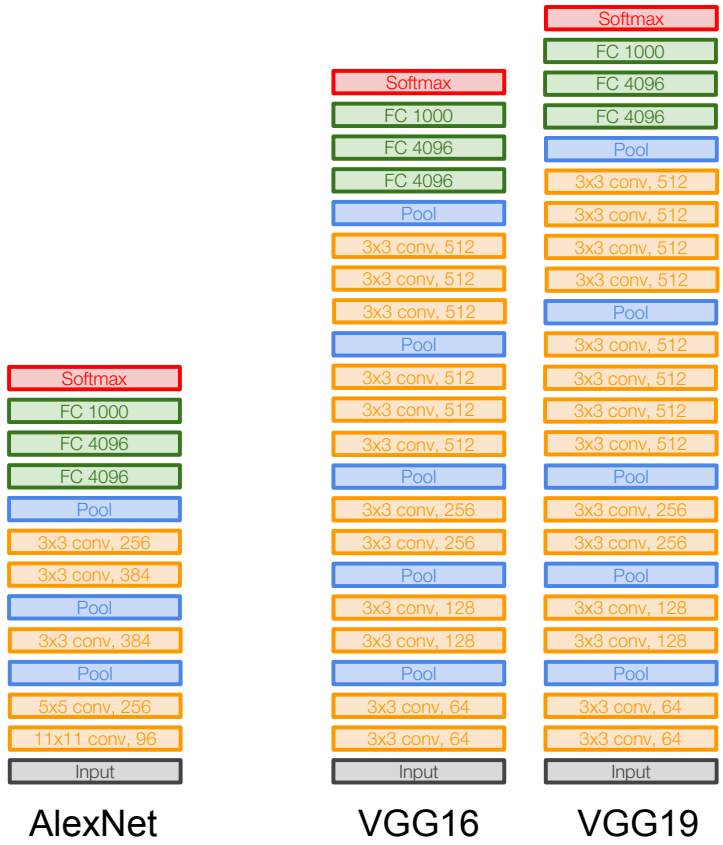
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters: 3 * (3^2C^2) vs. 7^2C^2 for C channels per layer



VGGNet

(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

VGG16

VGGNet

(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

VGG16

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

VGGNet

(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

Note:

Most memory is in early CONV

Most params are in late FC

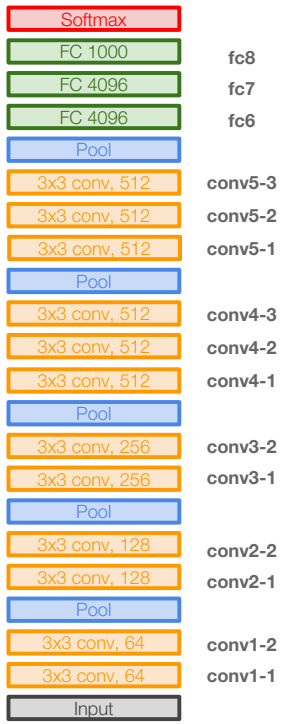
TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

VGGNet

(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 ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters



VGG16

Common names

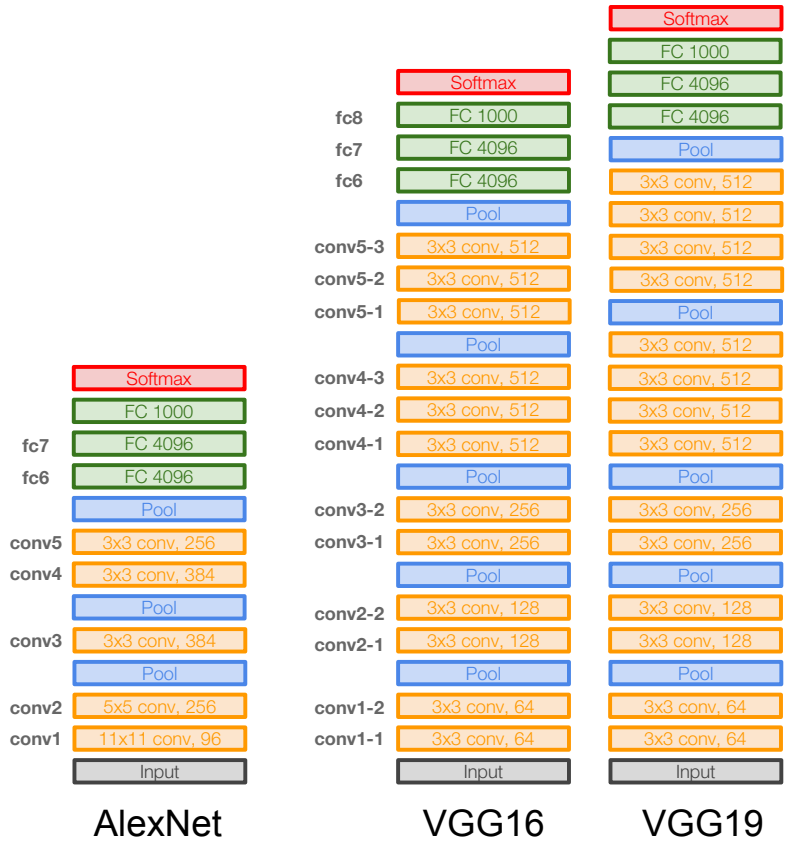
VGGNet

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

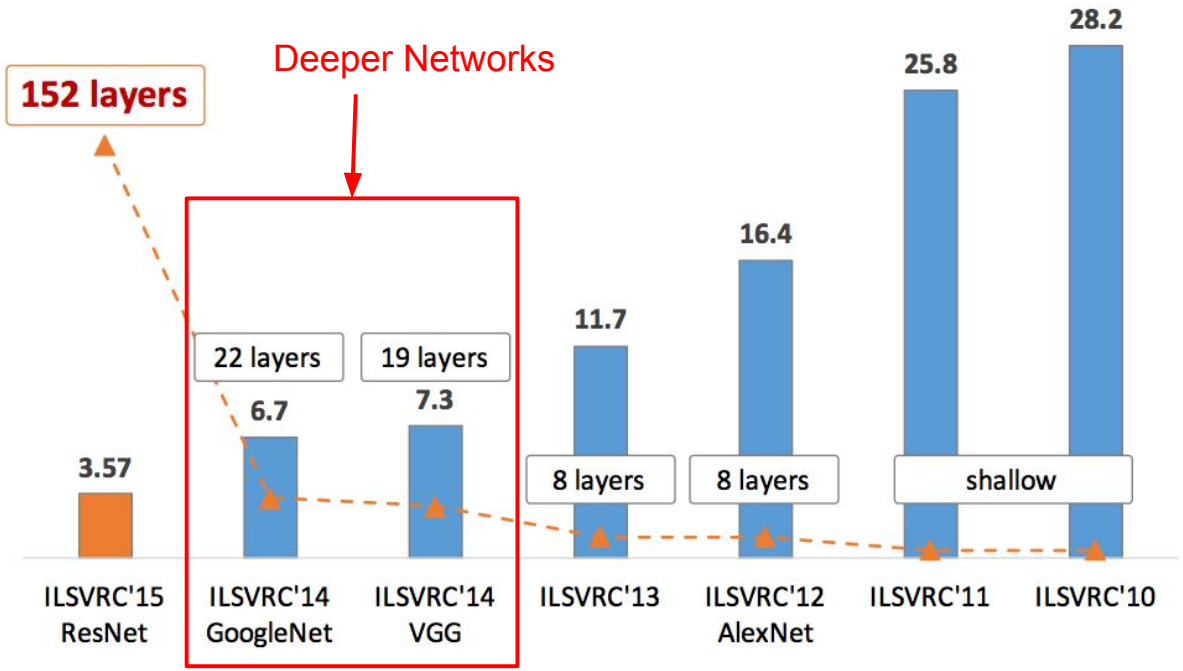
Details:

- ILSVRC'14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks



VGGNet

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



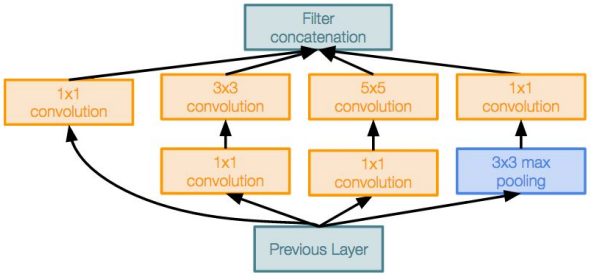
GoogleNet

Case Study: GoogLeNet

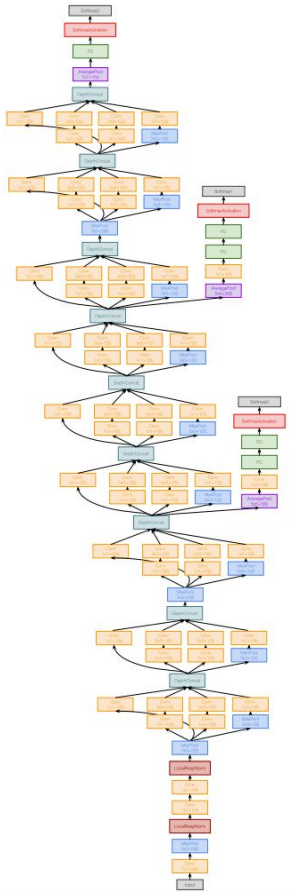
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient "Inception" module
- No FC layers
- Only 5 million parameters!
  12x less than AlexNet
- ILSVRC'14 classification winner
  (6.7% top 5 error)



Inception module

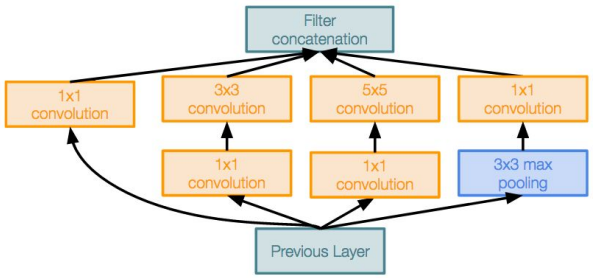


GoogleNet

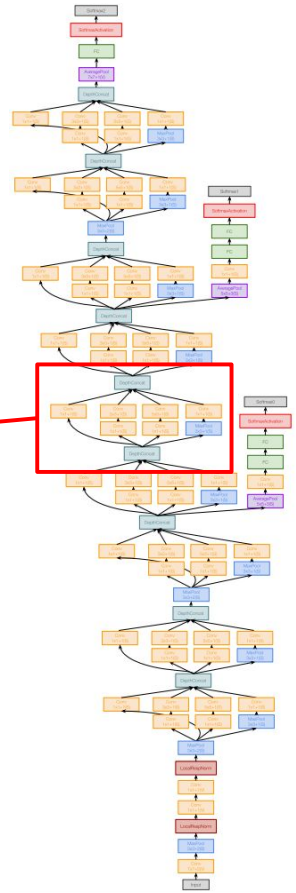
Case Study: GoogLeNet

[Szegedy et al., 2014]

"Inception module": design a good local network topology (network within a network) and then stack these modules on top of each other



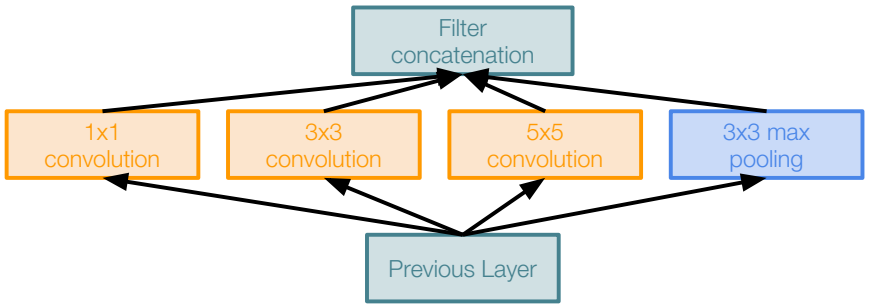
Inception module



GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

Apply parallel filter operations on the input from previous layer:

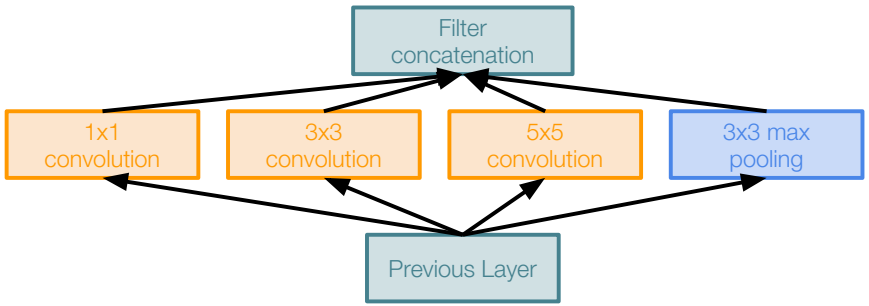
- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together depth-wise

GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together depth-wise

Q: What is the problem with this? [Hint: Computational complexity]

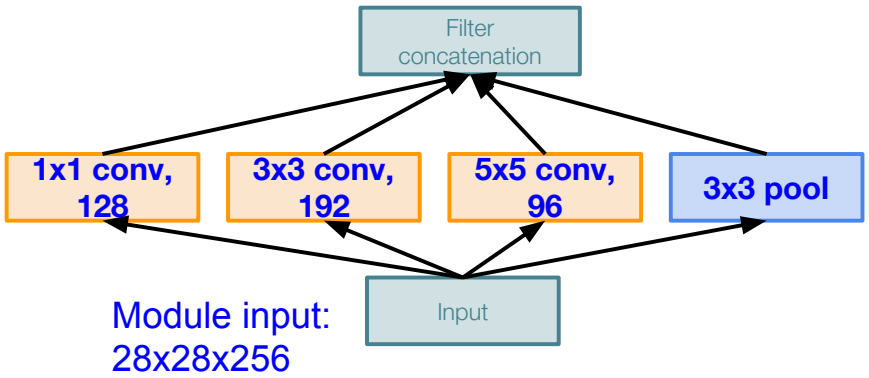
GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:



Naive Inception module

GoogleNet

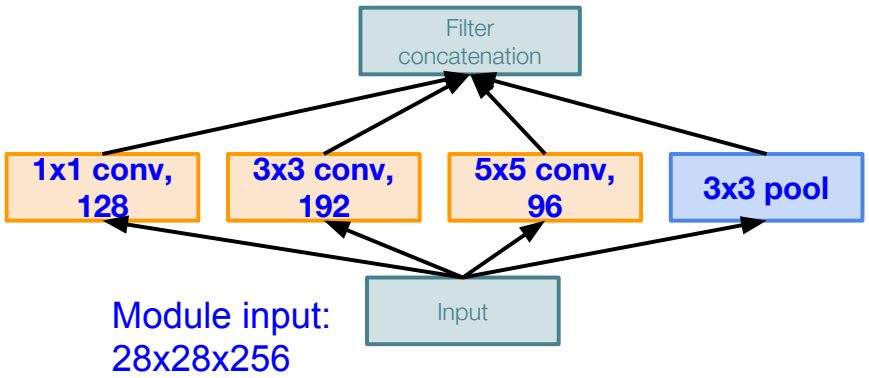
Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q1: What is the output size of the 1x1 conv, with 128 filters?

Q: What is the problem with this?
[Hint: Computational complexity]



Naive Inception module

GoogleNet

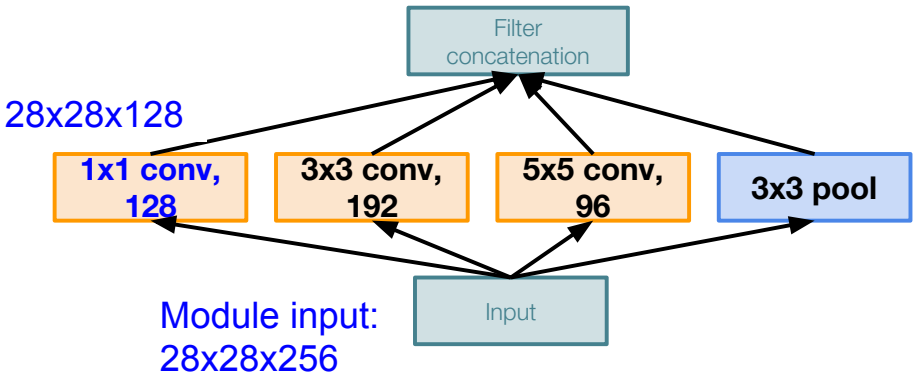
Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q1: What is the output size of the 1x1 conv, with 128 filters?

Q: What is the problem with this?
[Hint: Computational complexity]



Naive Inception module

GoogleNet

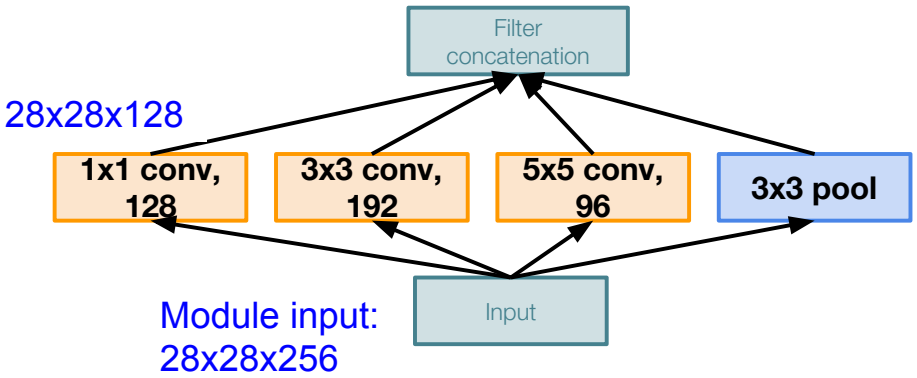
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q2: What are the output sizes of all different filter operations?



Naive Inception module

GoogleNet

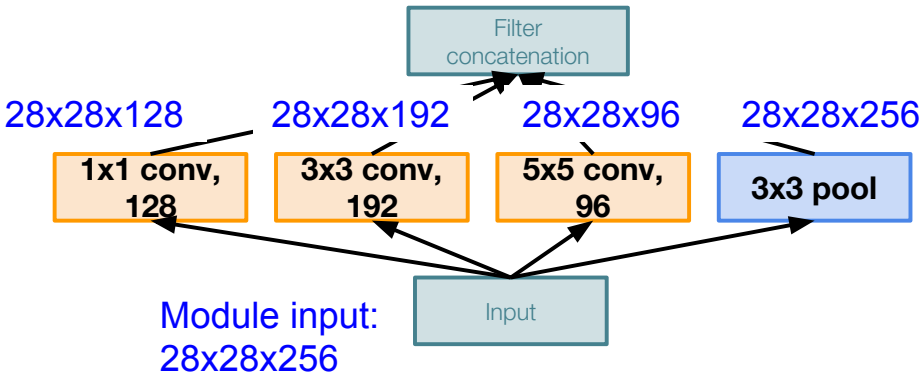
Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q2: What are the output sizes of all different filter operations?

Q: What is the problem with this? [Hint: Computational complexity]



Naive Inception module

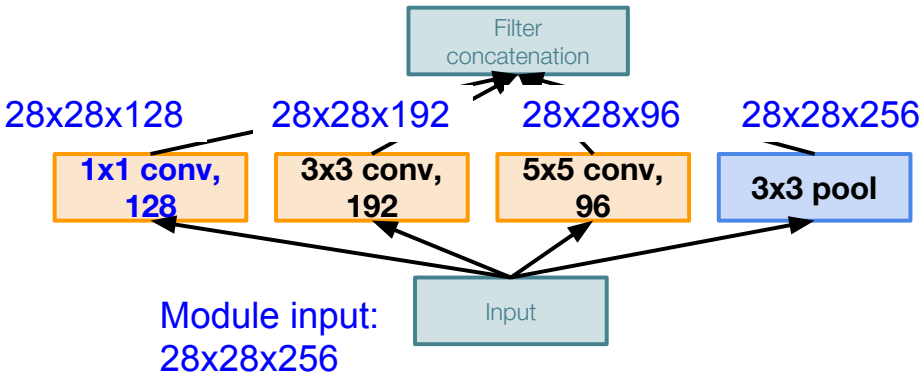
GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

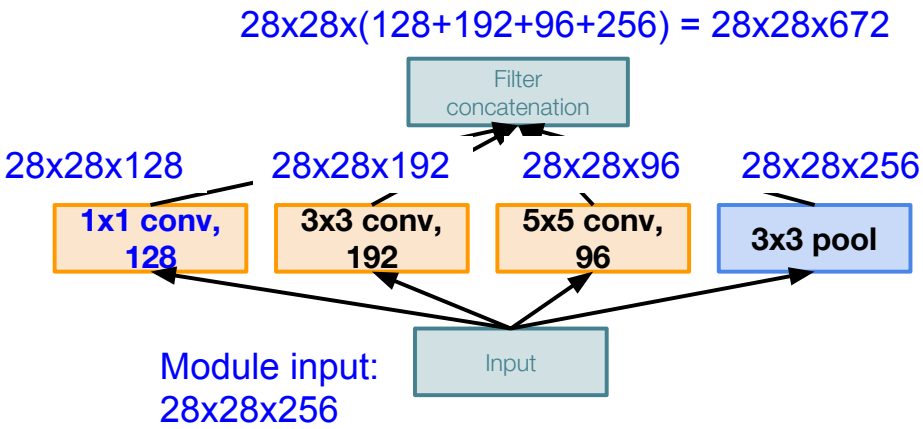
GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

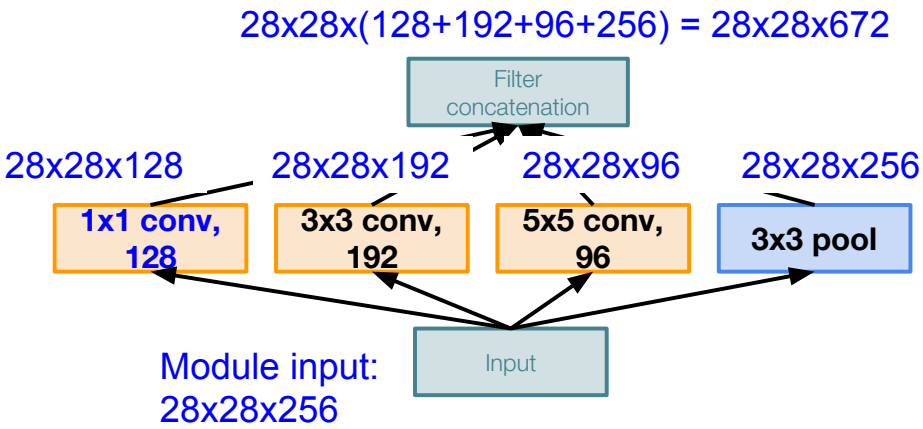
GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops

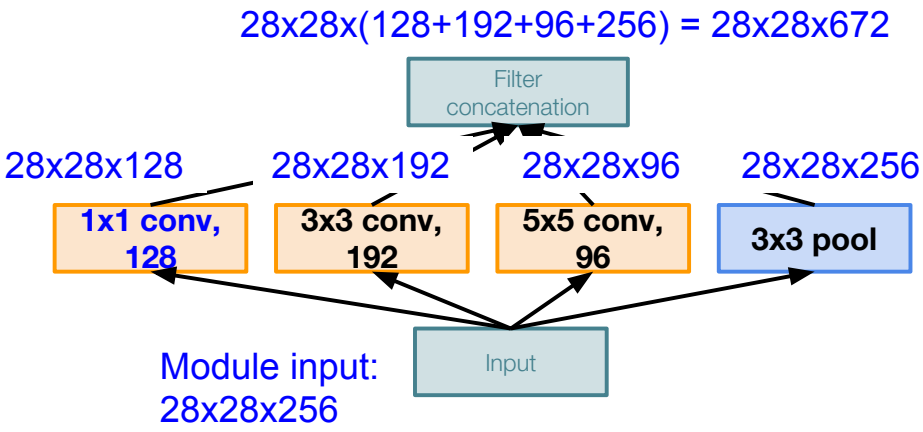
GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops

Very expensive compute

Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

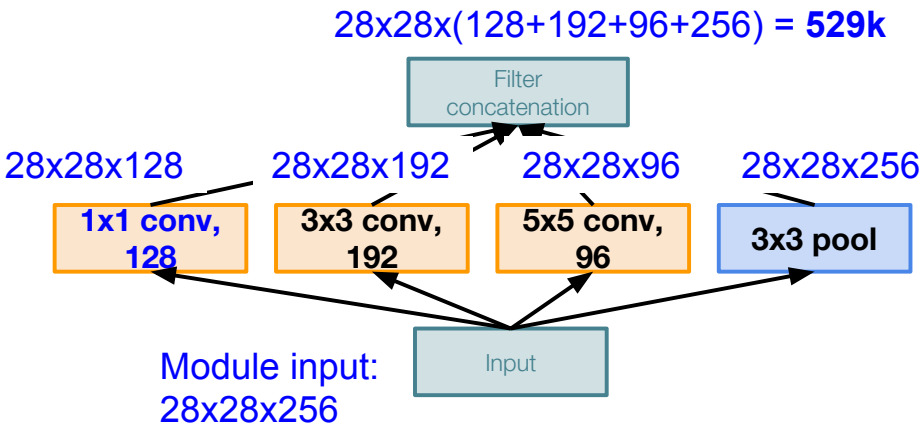
GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



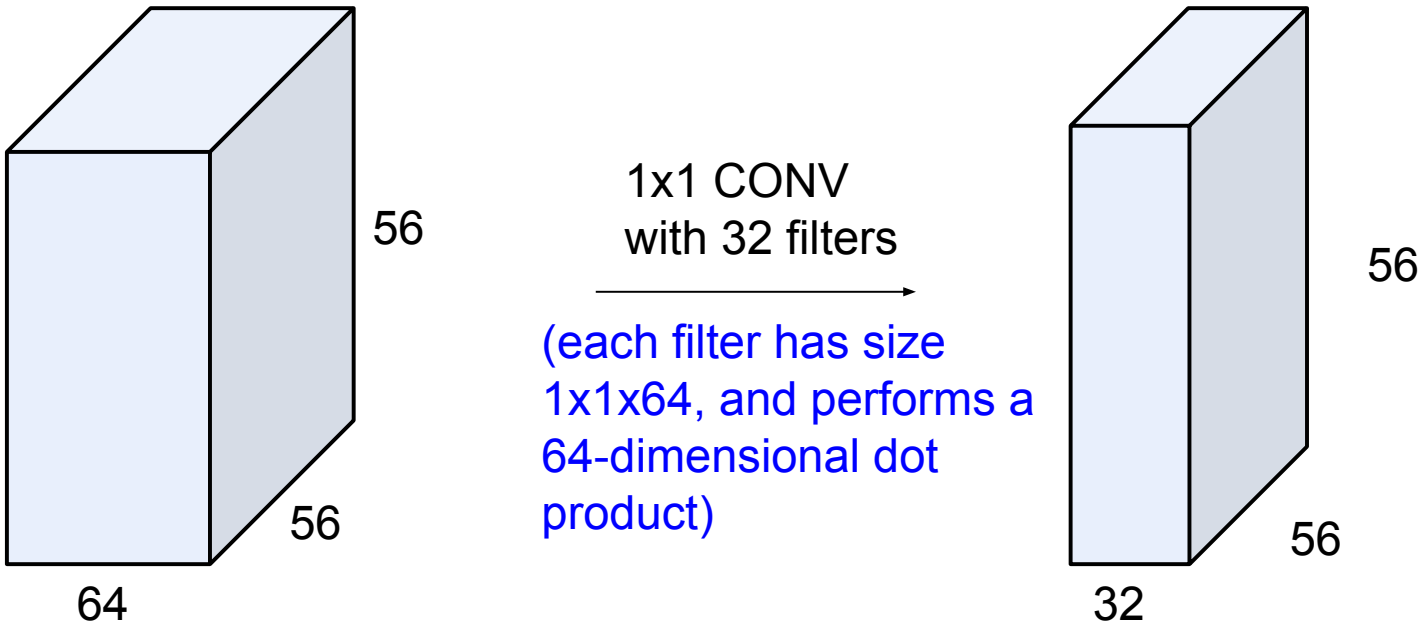
Naive Inception module

Q: What is the problem with this? [Hint: Computational complexity]

Solution: "bottleneck" layers that use 1x1 convolutions to reduce feature depth

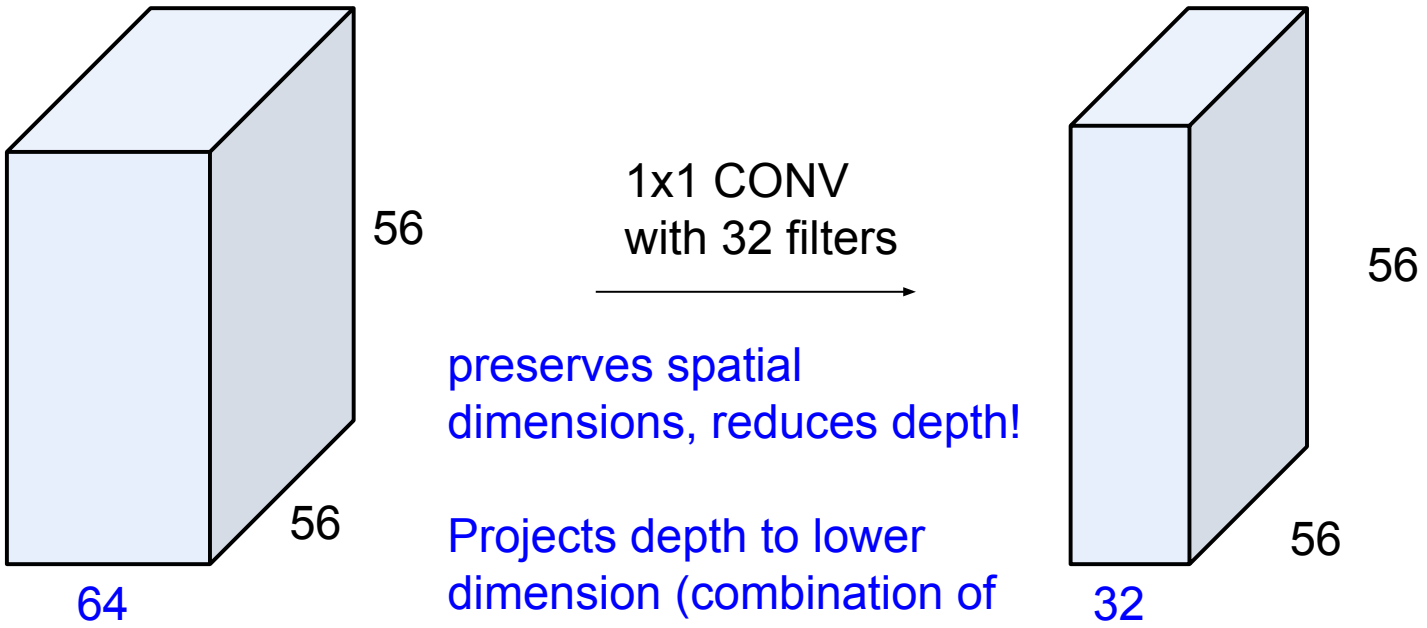
GoogleNet

Reminder: 1x1 convolutions



GoogleNet

Reminder: 1x1 convolutions



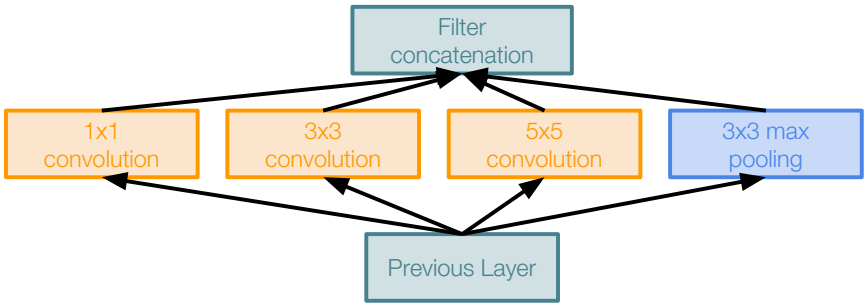
preserves spatial dimensions, reduces depth!

Projects depth to lower dimension (combination of feature maps)

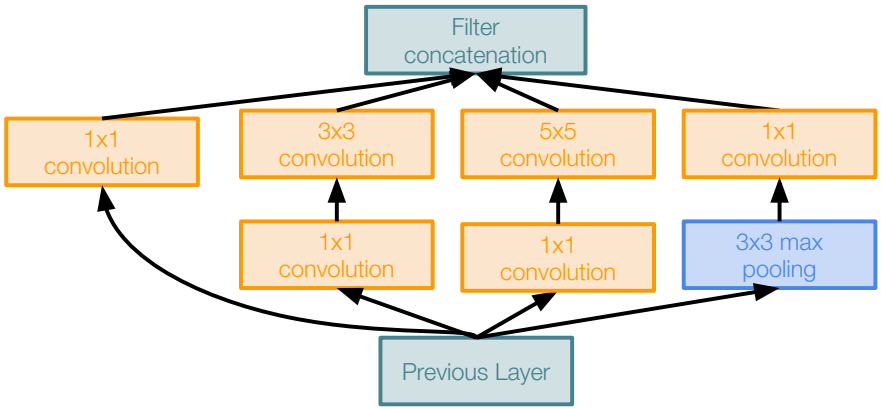
GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

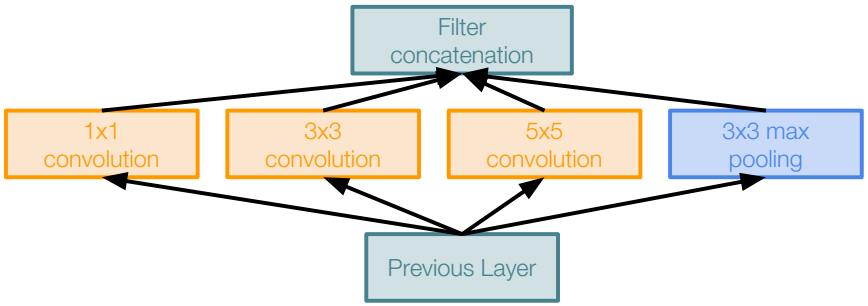


Inception module with dimension reduction

GoogleNet

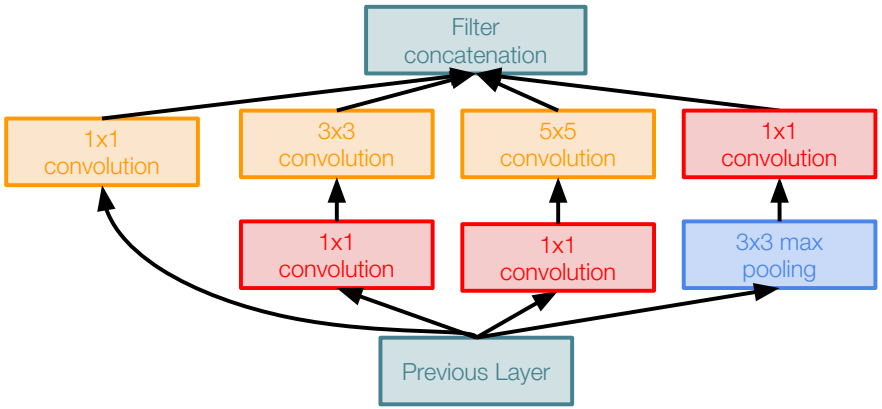
Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

1x1 conv "bottleneck" layers

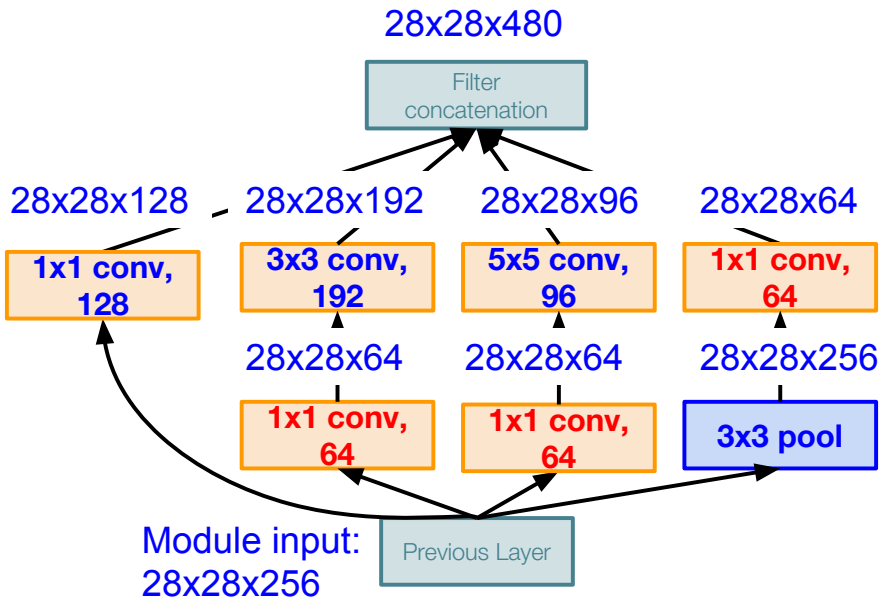


Inception module with dimension reduction

GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]



Inception module with dimension reduction

Using same parallel layers as naive example, and adding "1x1 conv, 64 filter" bottlenecks:

Conv Ops:
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256
Total: 358M ops

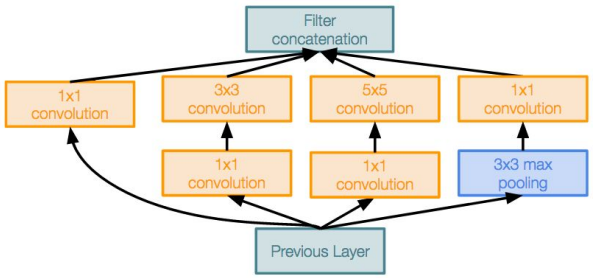
Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer

GoogleNet

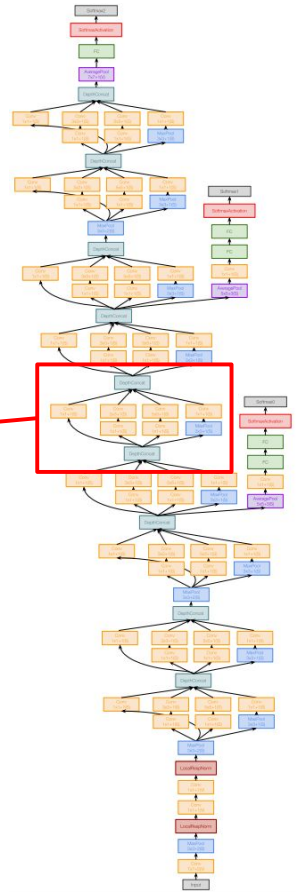
Case Study: GoogLeNet

[Szegedy et al., 2014]

Stack Inception modules with dimension reduction on top of each other



Inception module

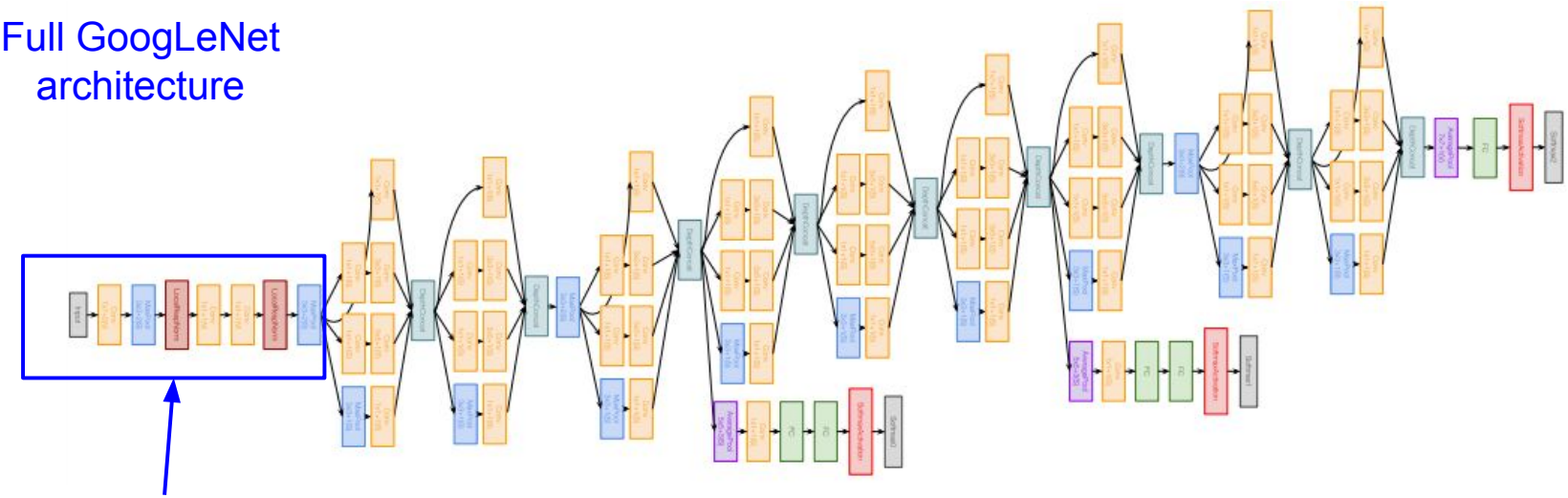


GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture



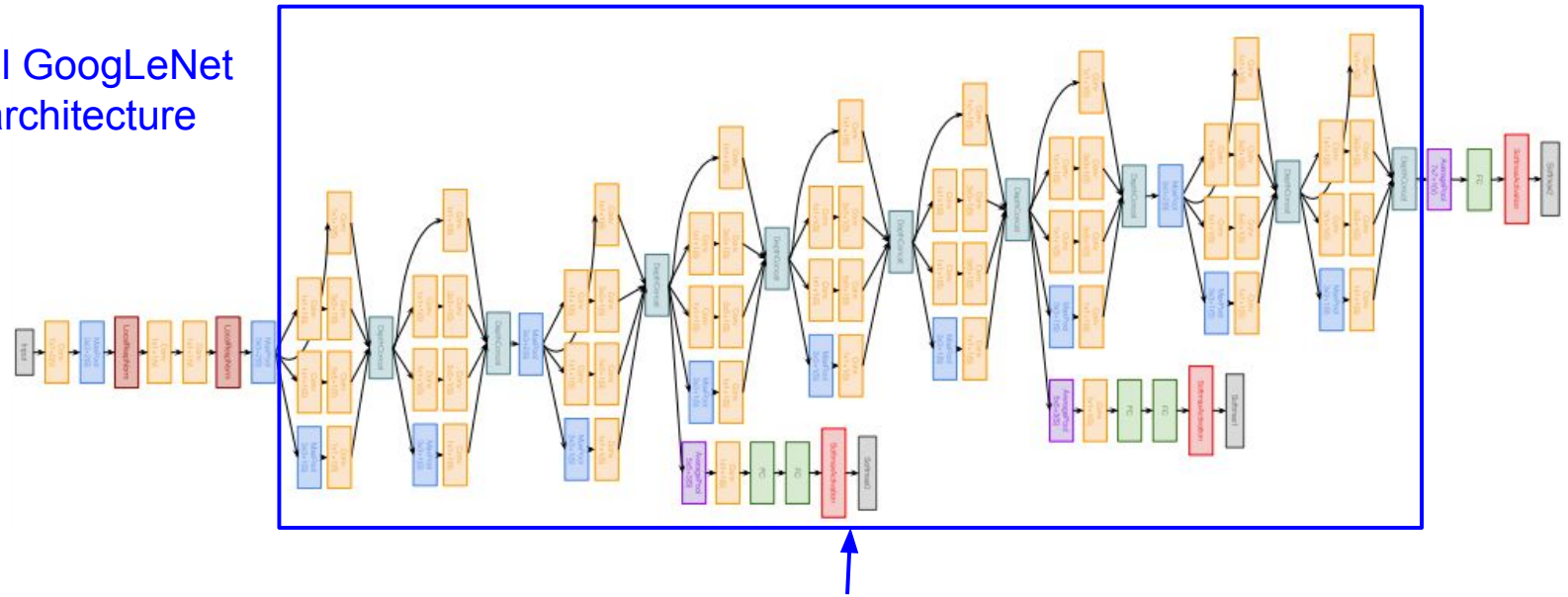
Stem Network: Conv-Pool- 2x Conv-Pool

GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture



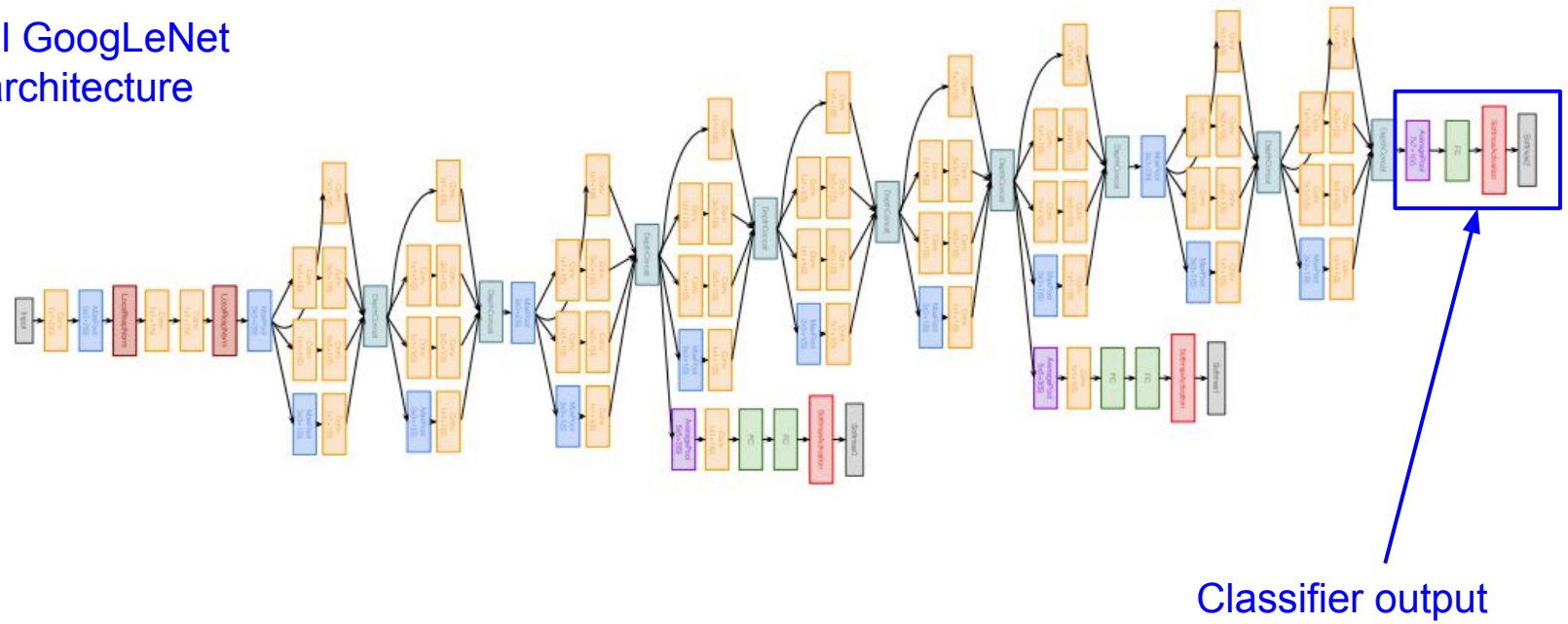
Stacked Inception Modules

GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

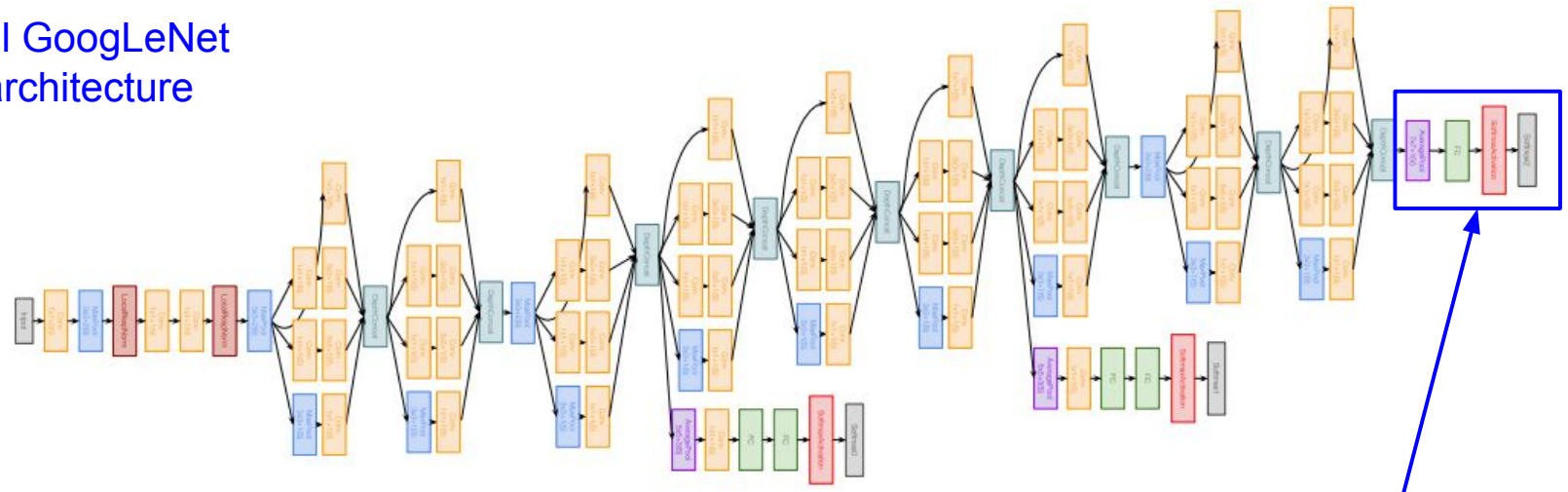


GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture



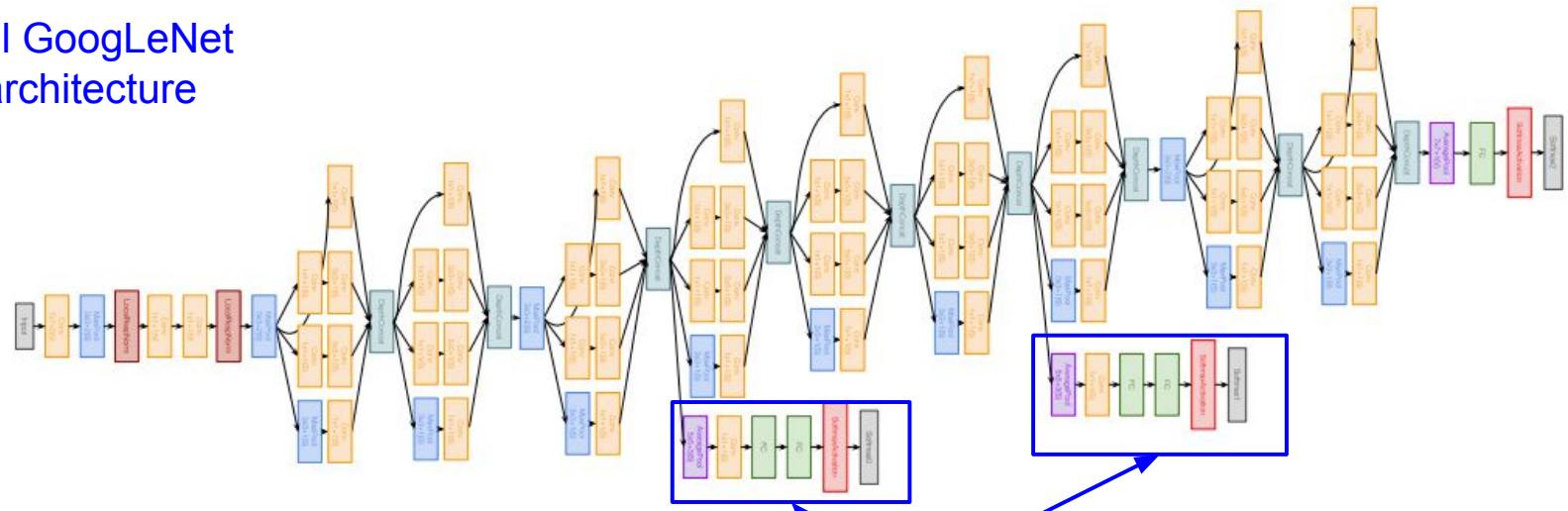
Classifier output
(removed expensive FC layers!)

GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture



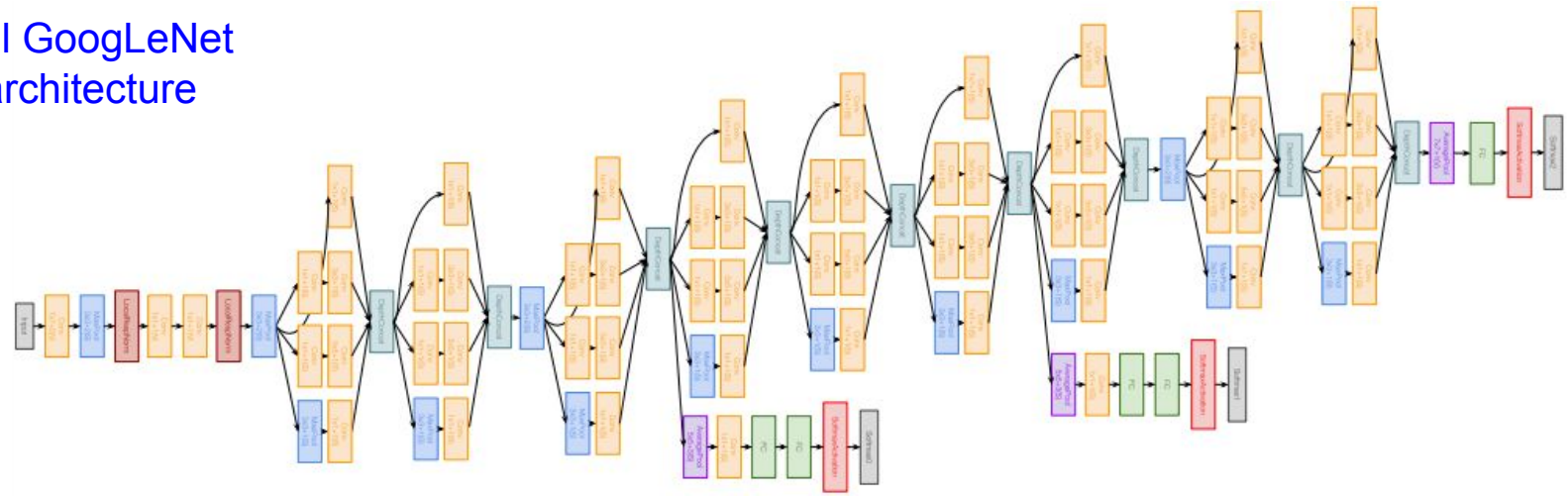
Auxiliary classification outputs to inject additional gradient at lower layers (AvgPool-1x1Conv-FC-FC-Softmax)

GoogleNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture



22 total layers with weights (including each parallel layer in an Inception module)

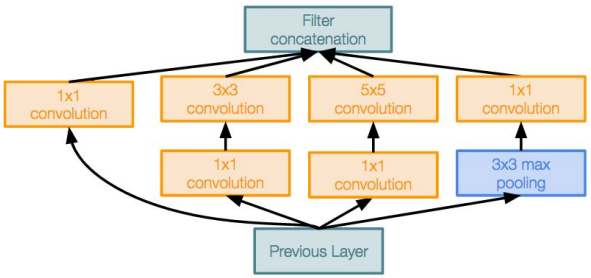
GoogleNet

Case Study: GoogLeNet

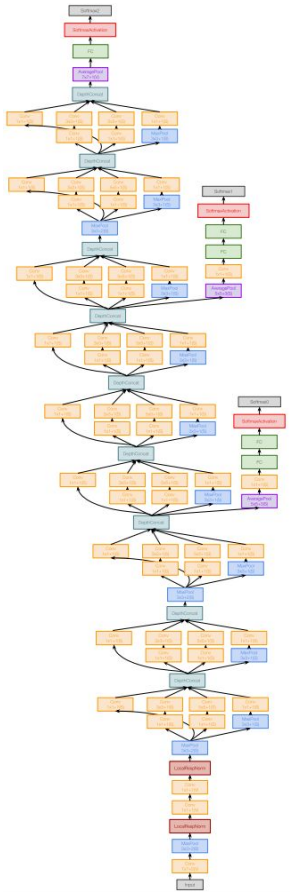
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient "Inception" module
- No FC layers
- 12x less params than AlexNet
- ILSVRC'14 classification winner (6.7% top 5 error)

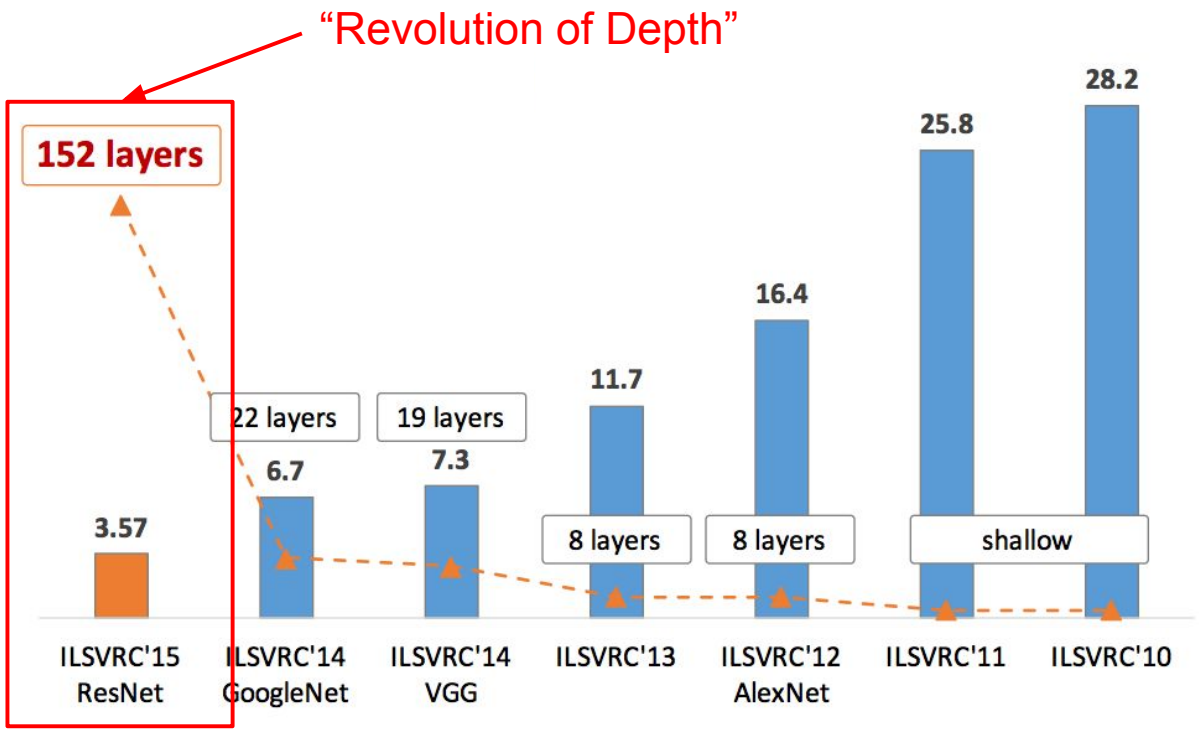


Inception module



ResNet

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



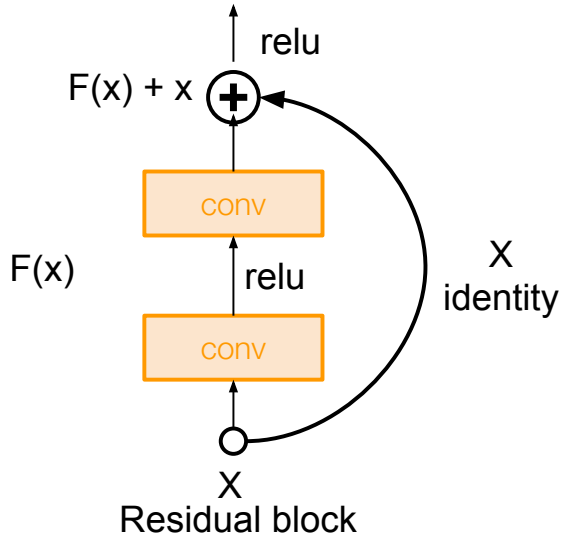
ResNet

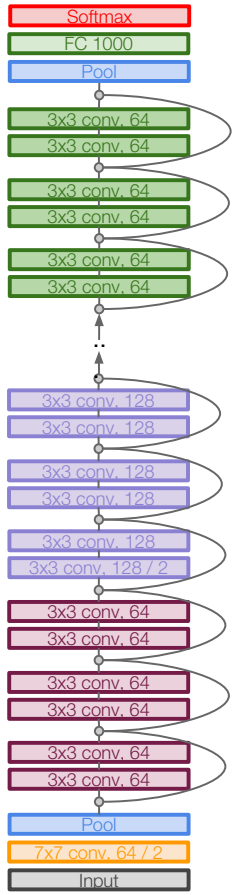
Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!





ResNet

Case Study: ResNet

[He et al., 2015]

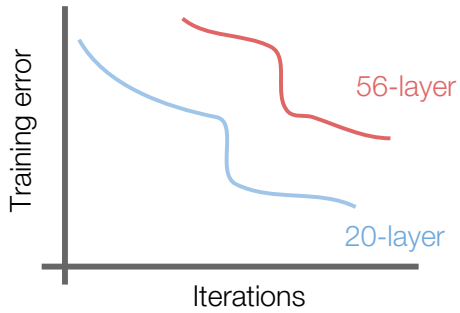
What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

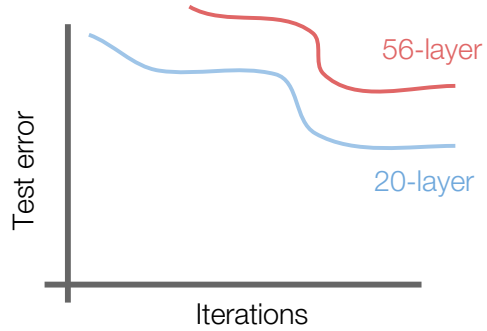
ResNet

Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?





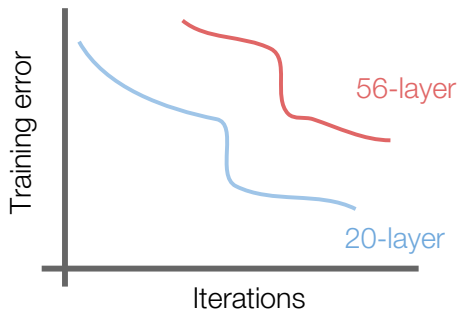
Q: What’s strange about these training and test curves?
[Hint: look at the order of the curves]

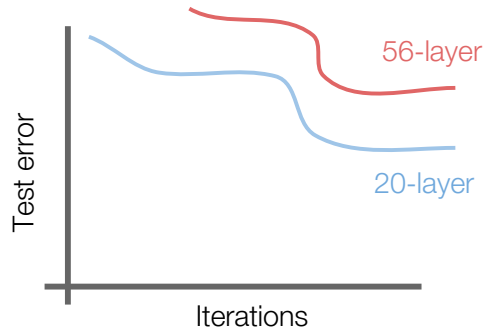
ResNet

Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?





56-layer model performs worse on both training and test error
-> The deeper model performs worse, but it’s not caused by overfitting!

ResNet

Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

ResNet

Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

The deeper model should be able to perform at least as well as the shallower model.

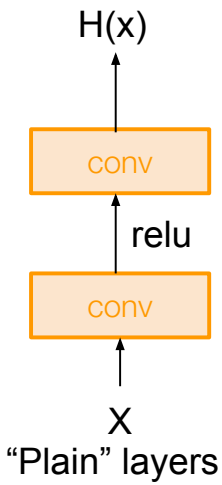
A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

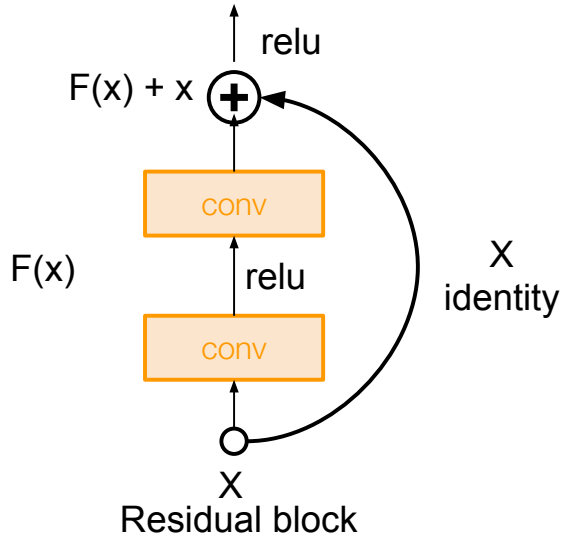
ResNet

Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



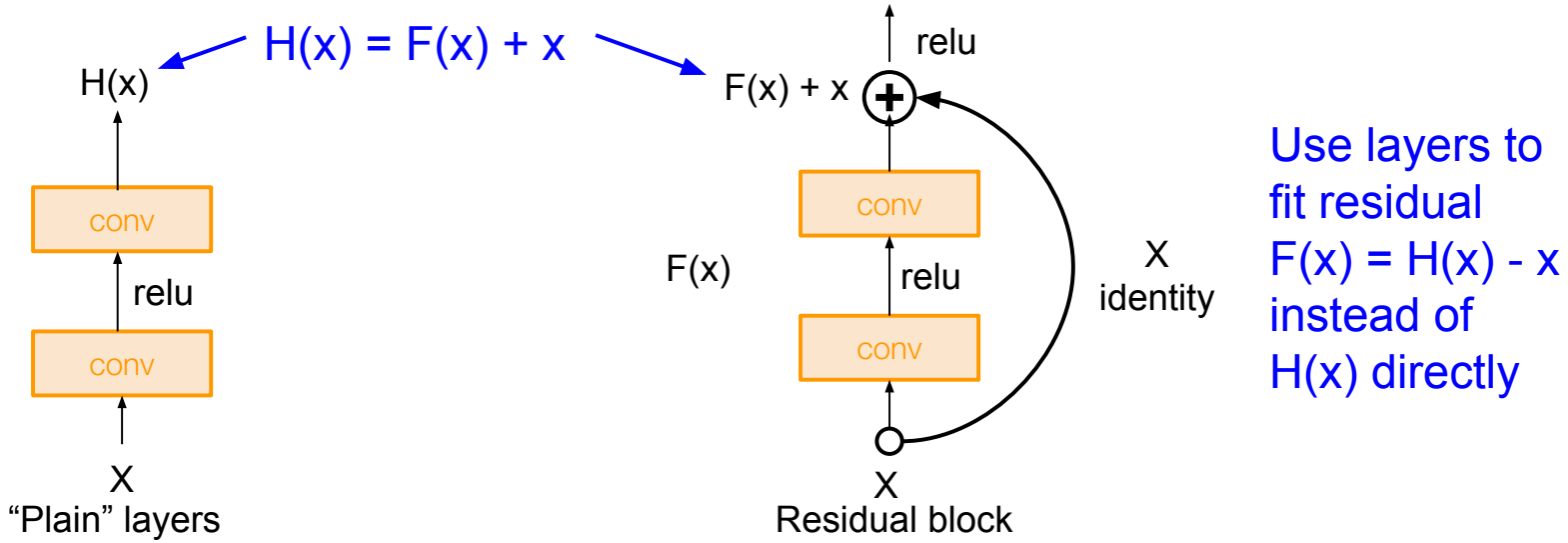


ResNet

Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



Use layers to fit residual F(x) = H(x) - x instead of H(x) directly

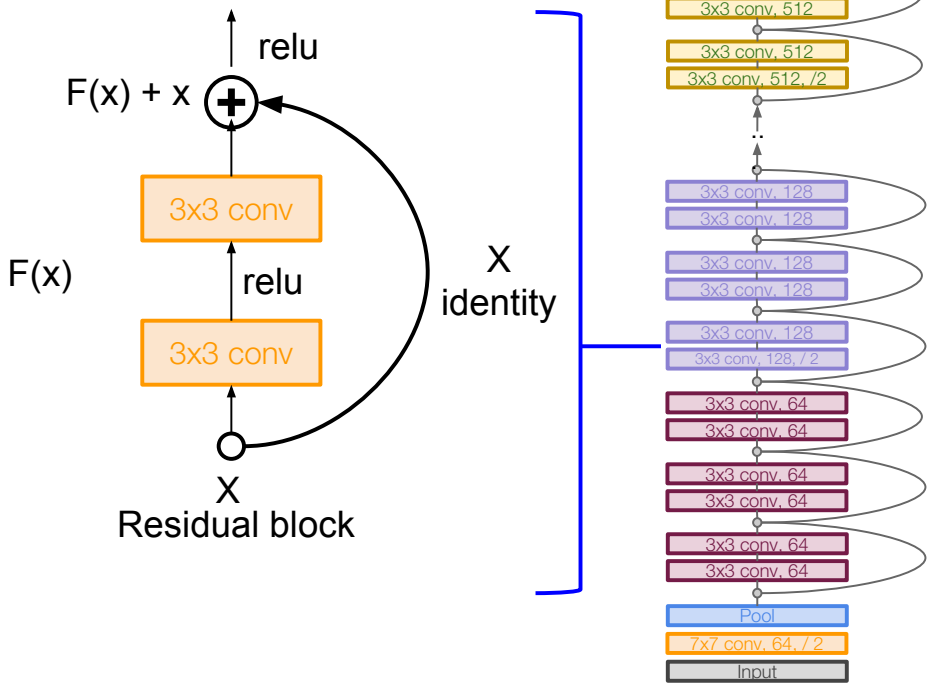
ResNet

Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers



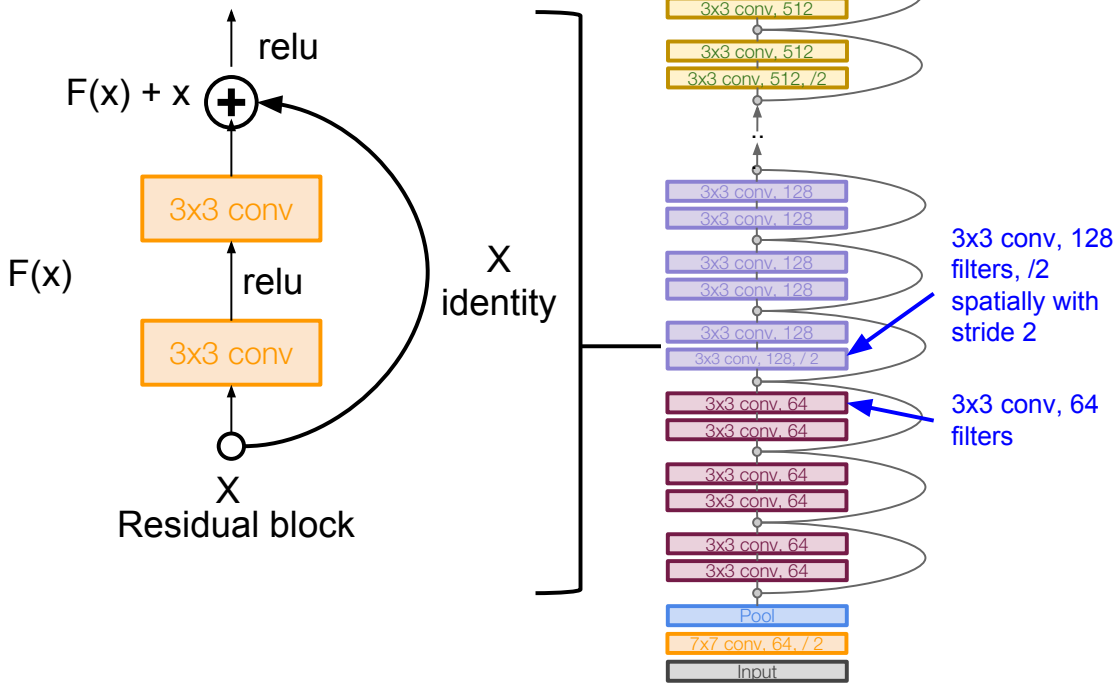
ResNet

Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)



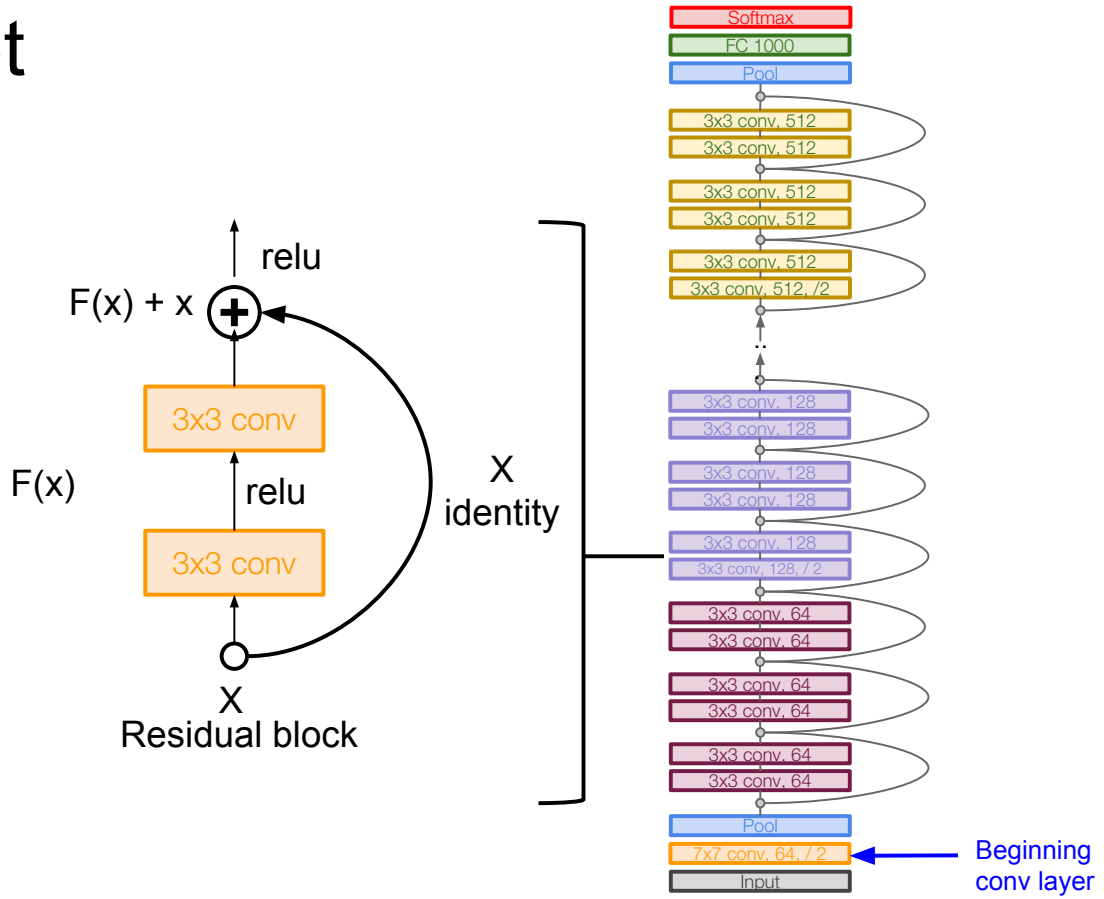
ResNet

Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning



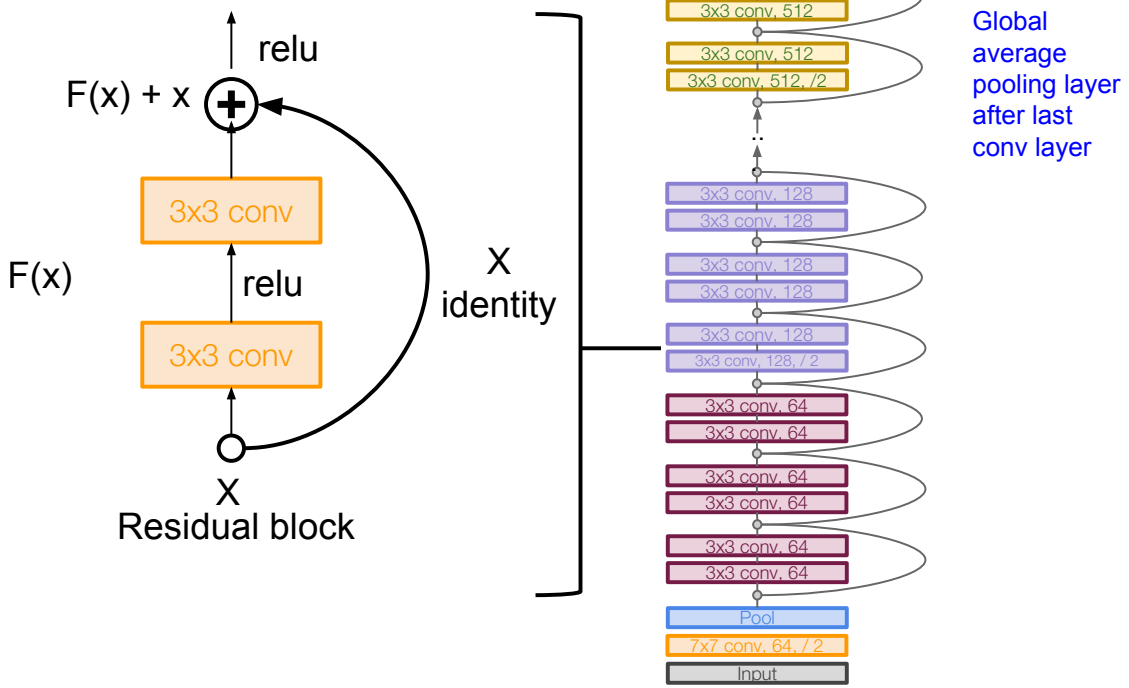
ResNet

Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)

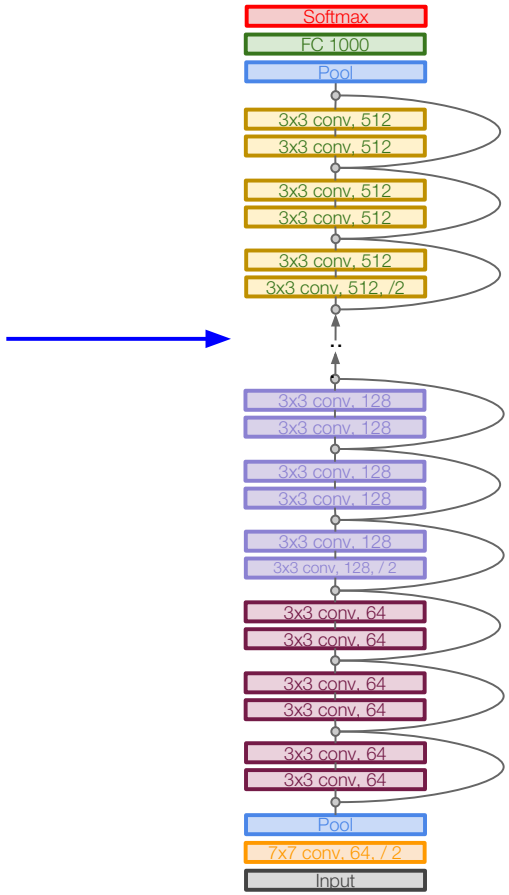


ResNet

Case Study: ResNet

[He et al., 2015]

Total depths of 34, 50, 101, or 152 layers for ImageNet

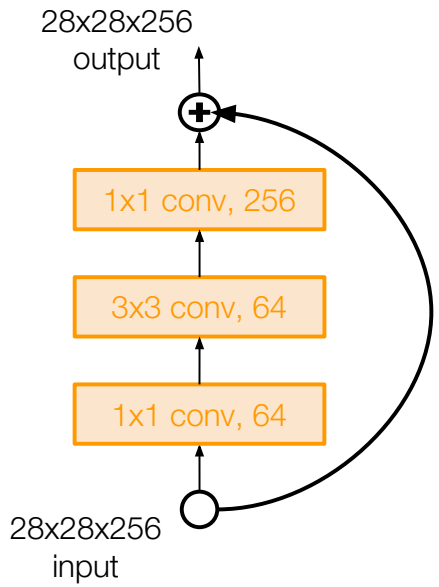


ResNet

Case Study: ResNet

[He et al., 2015]

For deeper networks
(ResNet-50+), use “bottleneck”
layer to improve efficiency
(similar to GoogLeNet)

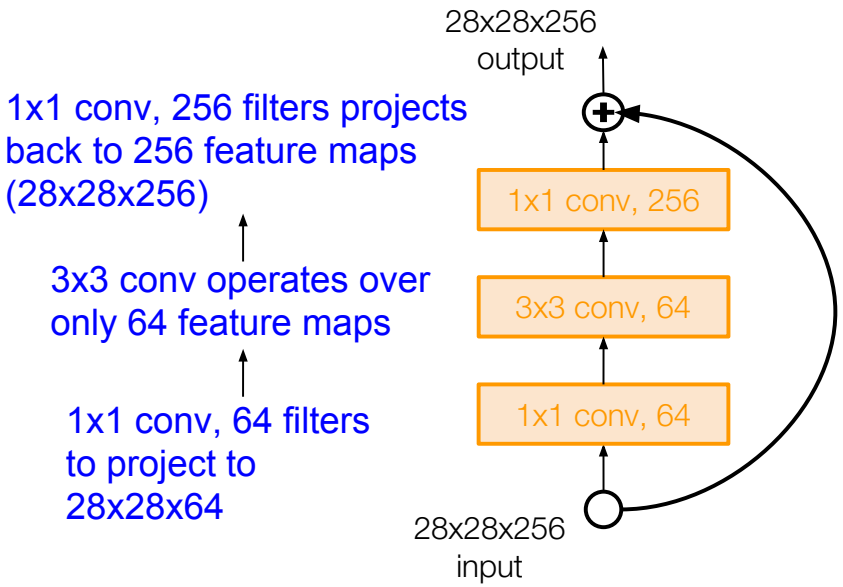


ResNet

Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+), use "bottleneck" layer to improve efficiency (similar to GoogLeNet)



ResNet

Case Study: ResNet

[He et al., 2015]

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier/2 initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

ResNet

Case Study: ResNet

[He et al., 2015]

Experimental Results

- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lowing training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: "*Ultra-deep*" (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

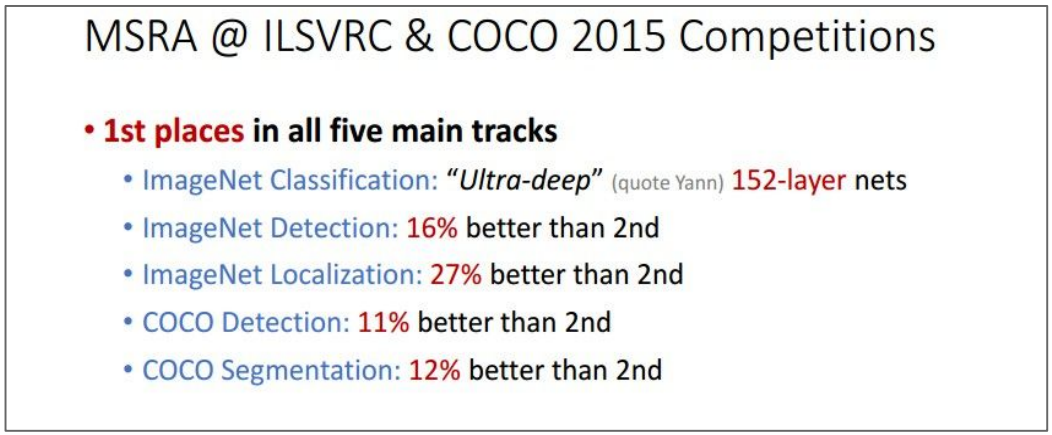
ResNet

Case Study: ResNet

[He et al., 2015]

Experimental Results

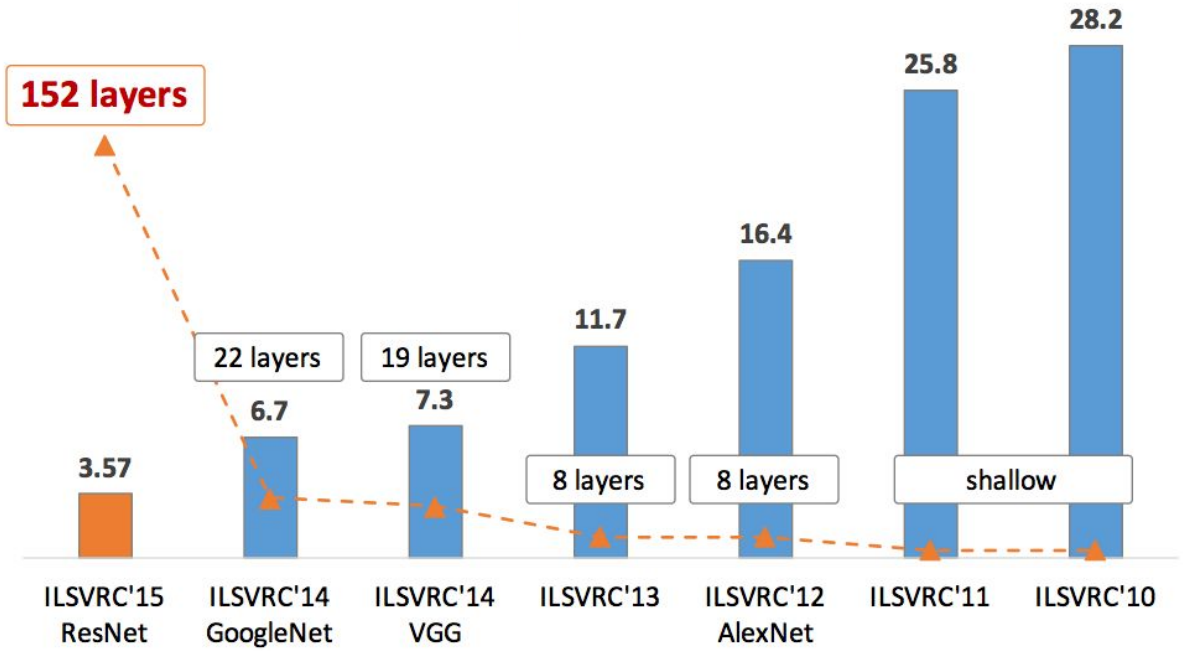
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lowing training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions



ILSVRC 2015 classification winner (3.6% top 5 error) -- better than "human performance"! (Russakovsky 2014)

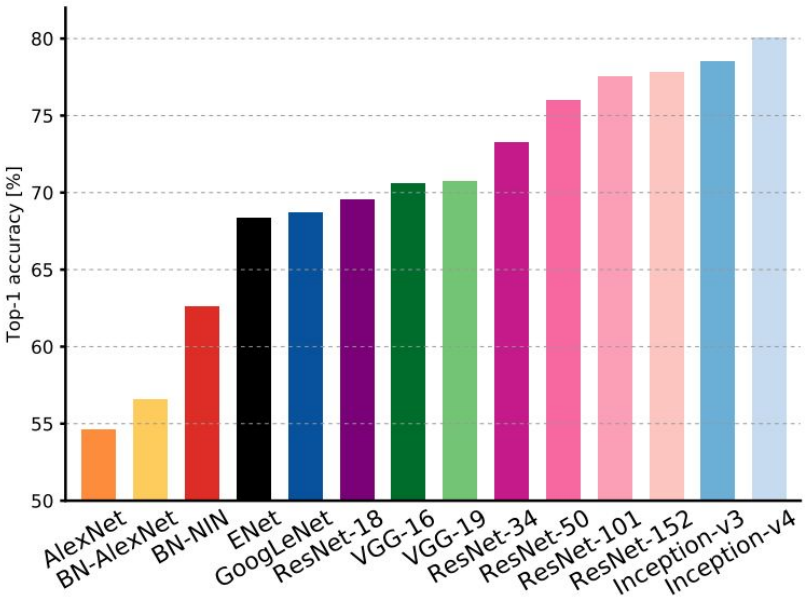
Architecture comparison

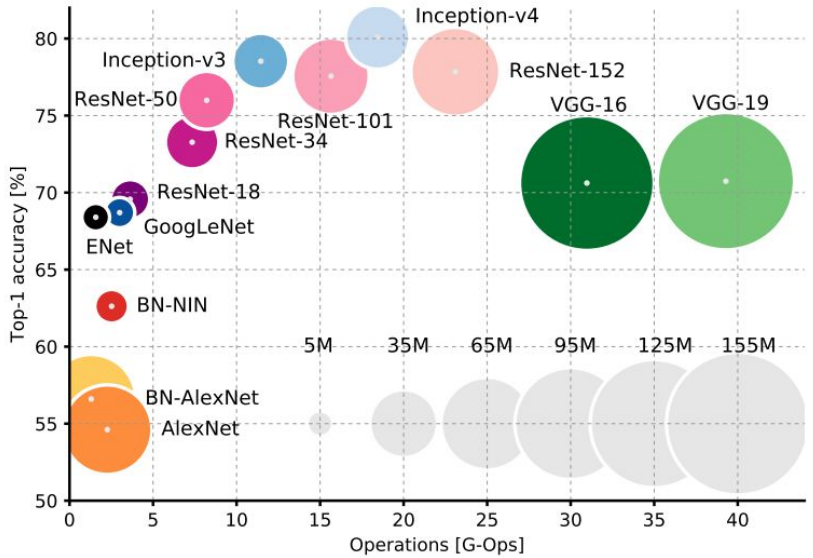
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Architecture comparison

Comparing complexity...



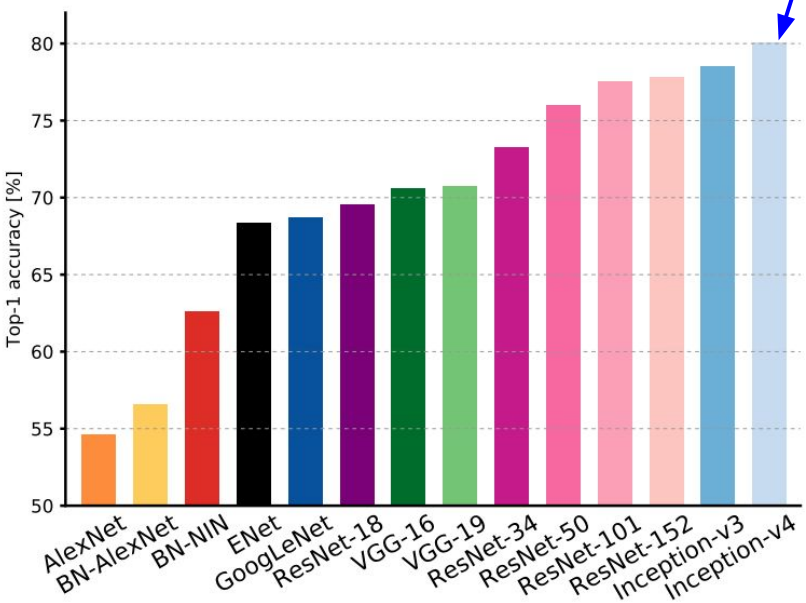


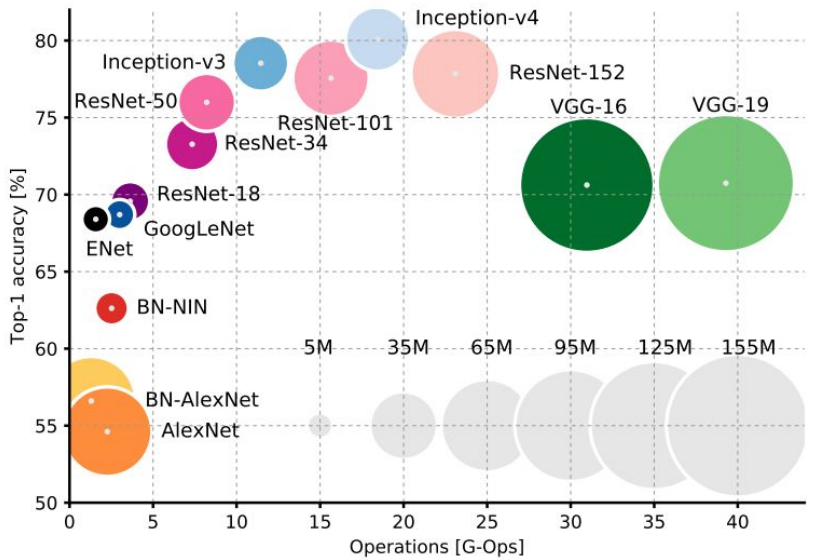
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Architecture comparison

Comparing complexity...

Inception-v4: Resnet + Inception!



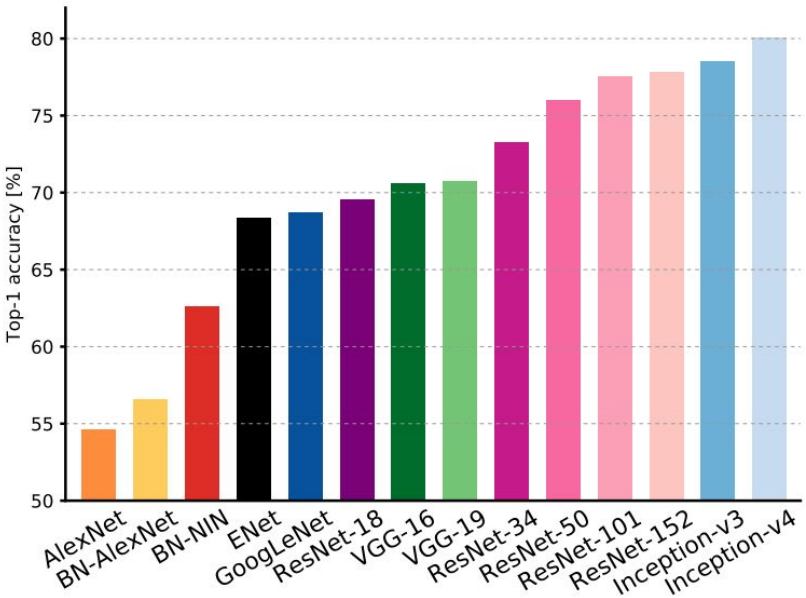


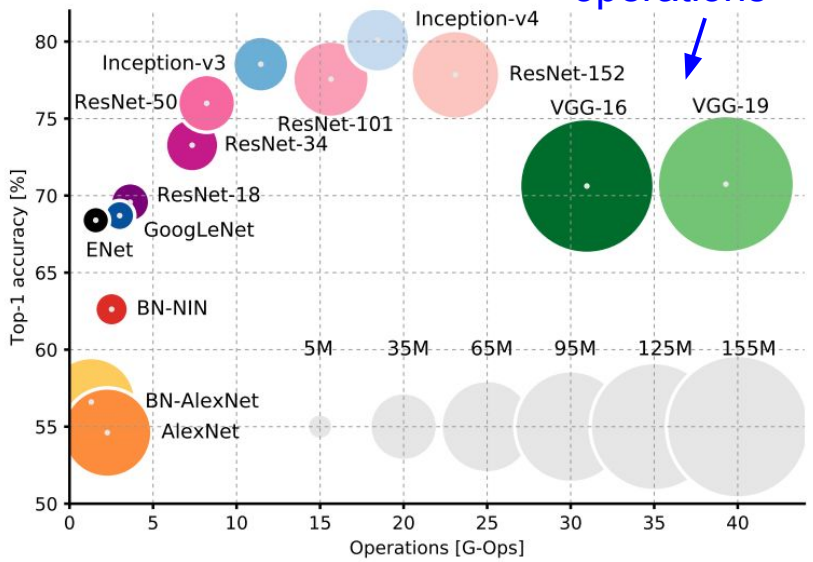
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Architecture comparison

Comparing complexity...

VGG: Highest memory, most operations

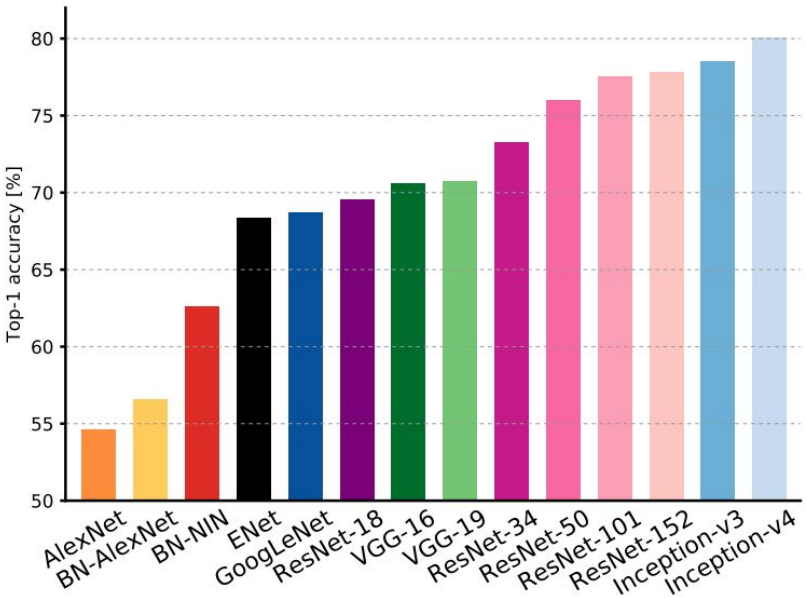


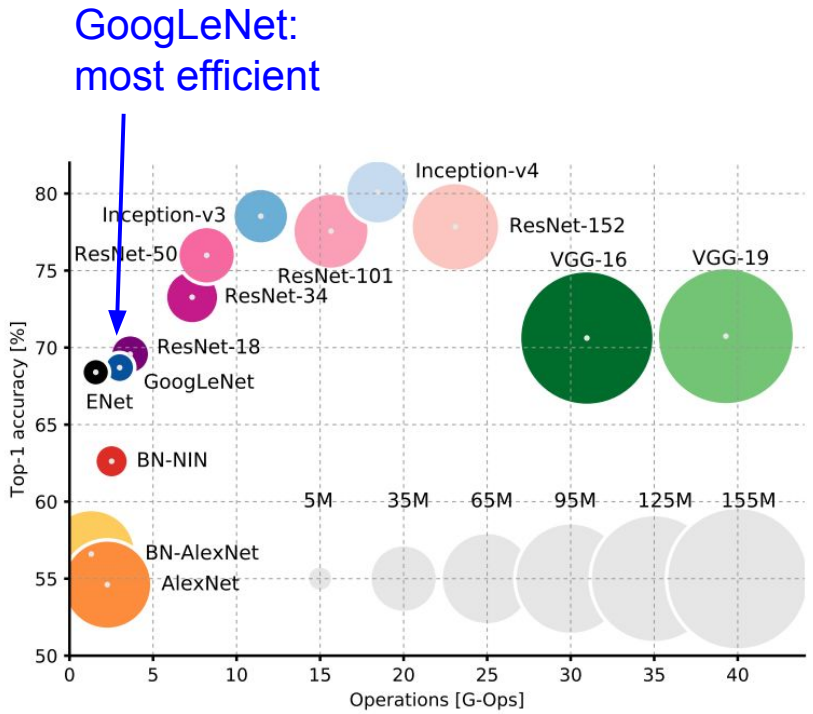


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Architecture comparison

Comparing complexity...

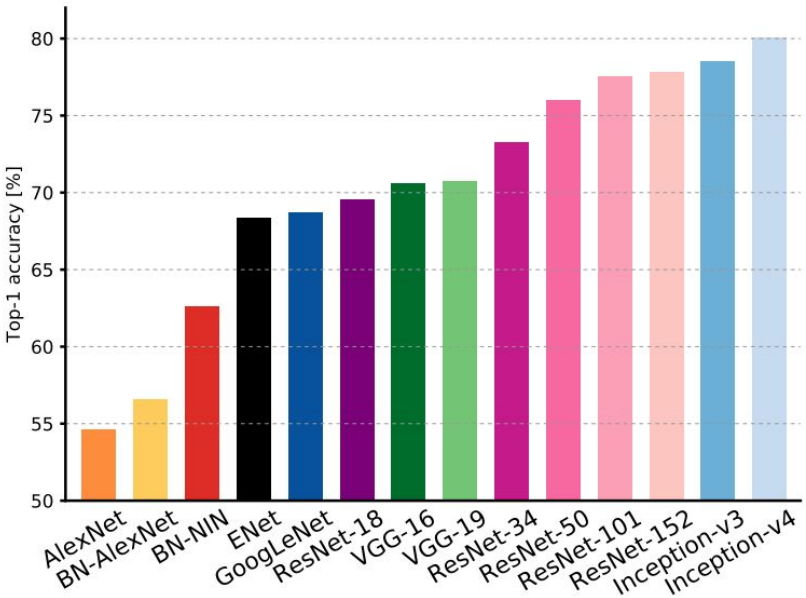




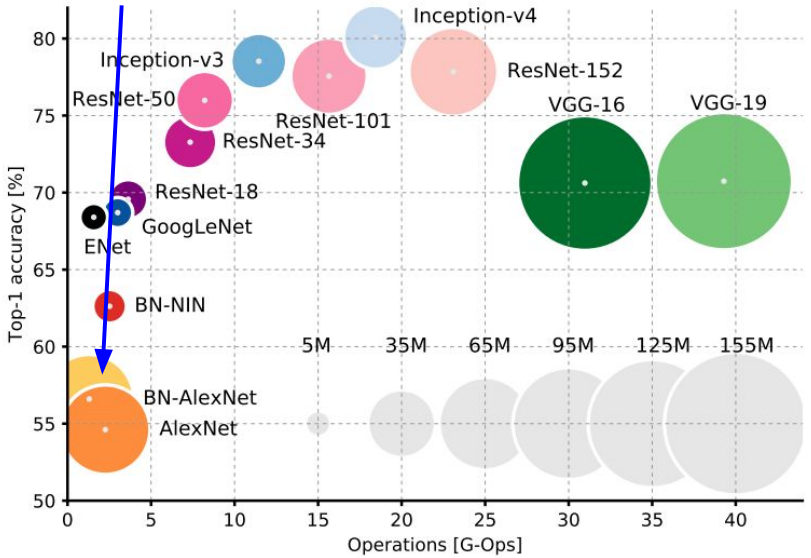
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Architecture comparison

Comparing complexity...



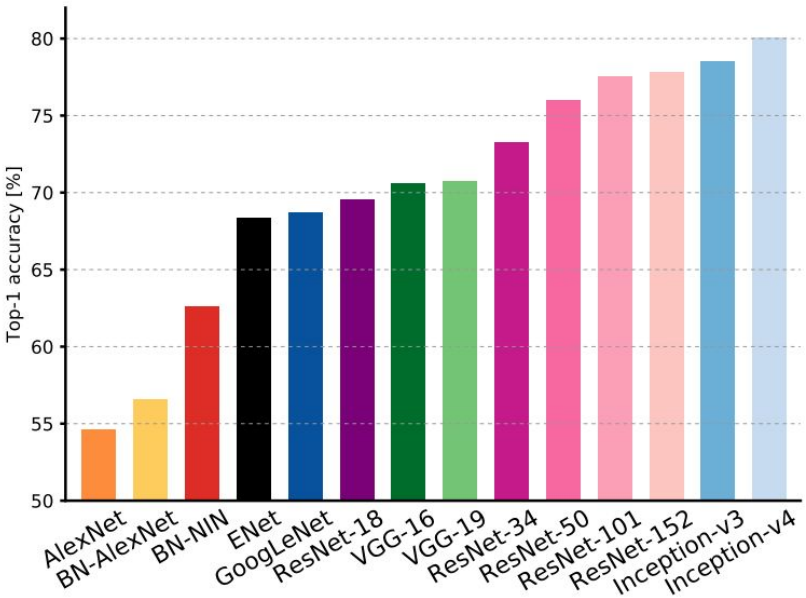
AlexNet:
Smaller compute, still memory heavy, lower accuracy



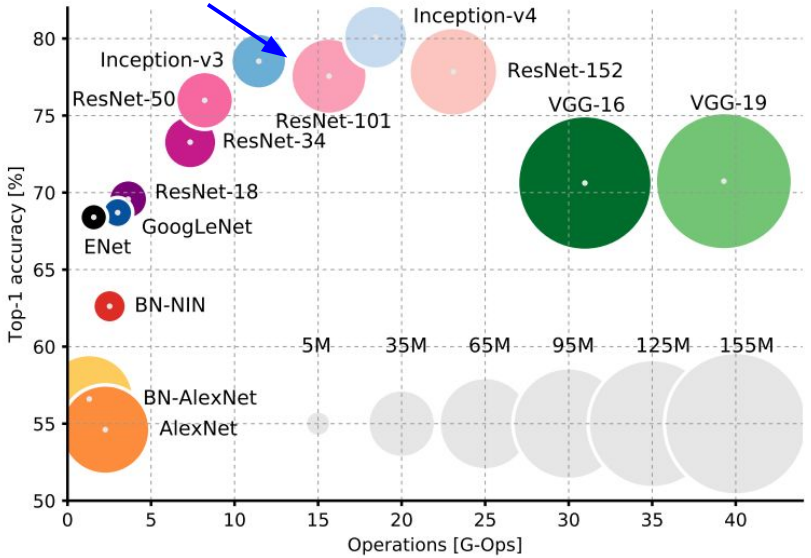
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Architecture comparison

Comparing complexity...



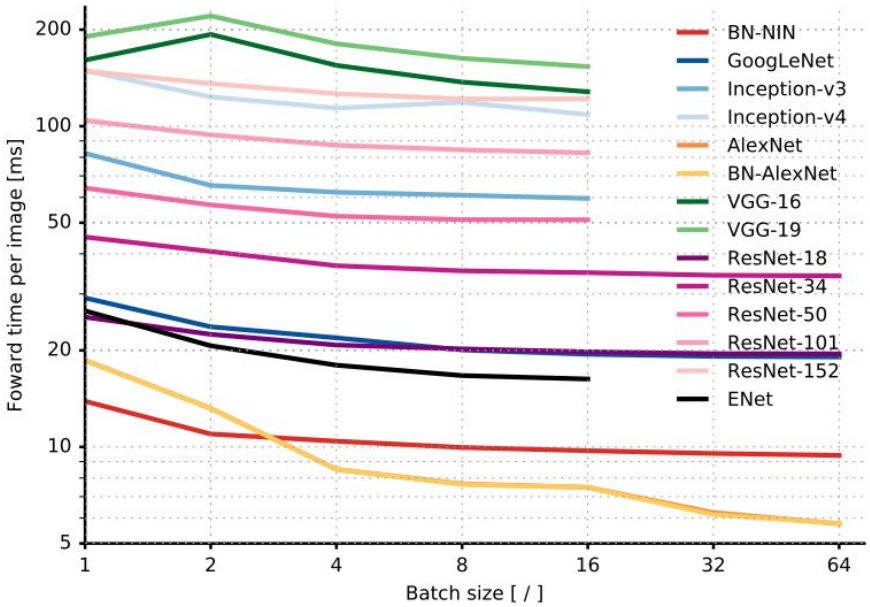
ResNet:
Moderate efficiency depending on model, highest accuracy

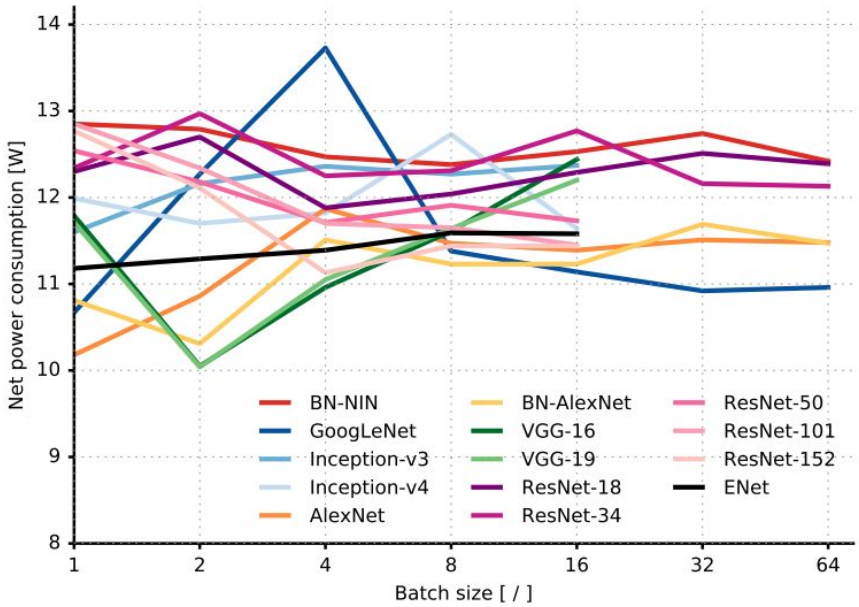


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Architecture comparison

Forward pass time and power consumption





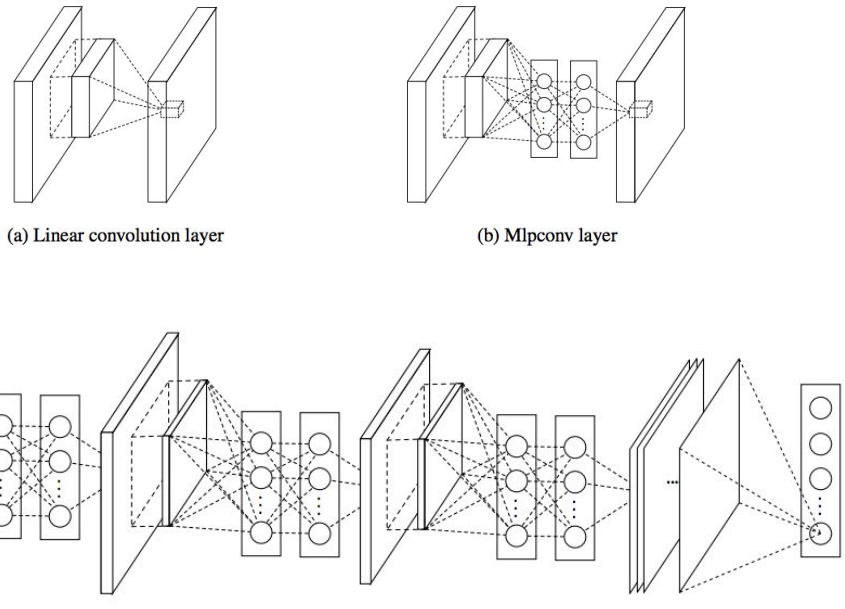
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Other Architecture

Network in Network (NiN)

[Lin et al. 2014]

- Mlpconv layer with "micronetwork" within each conv layer to compute more abstract features for local patches
- Micronetwork uses multilayer perceptron (FC, i.e. 1x1 conv layers)
- Precursor to GoogLeNet and ResNet "bottleneck" layers
- Philosophical inspiration for GoogLeNet



(a) Linear convolution layer (b) Mlpconv layer

Figures copyright Lin et al., 2014. Reproduced with permission.

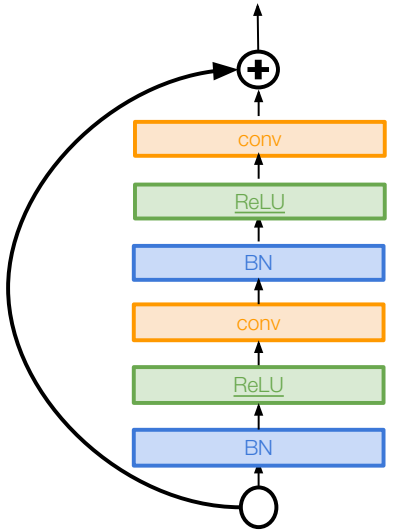
Other Architecture

Improving ResNets...

Identity Mappings in Deep Residual Networks

[He et al. 2016]

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network (moves activation to residual mapping pathway)
- Gives better performance



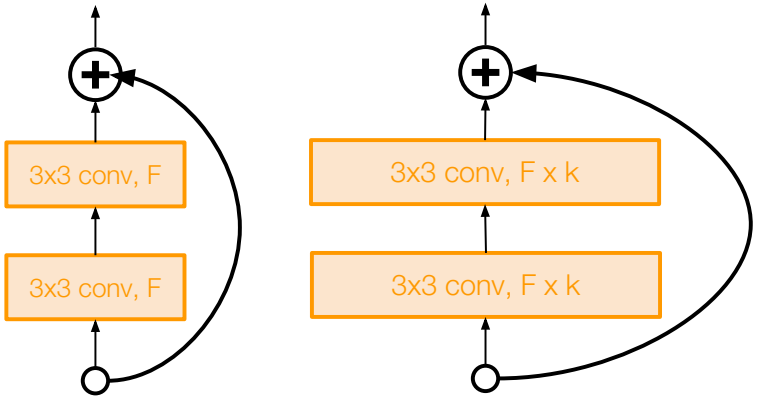
Other Architecture

Improving ResNets...

Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- User wider residual blocks (F x k filters instead of F filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)



Basic residual block | Wide residual block

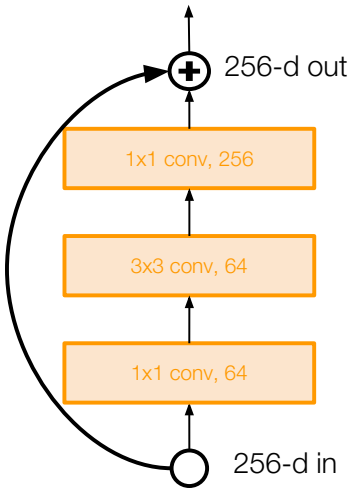
Other Architecture

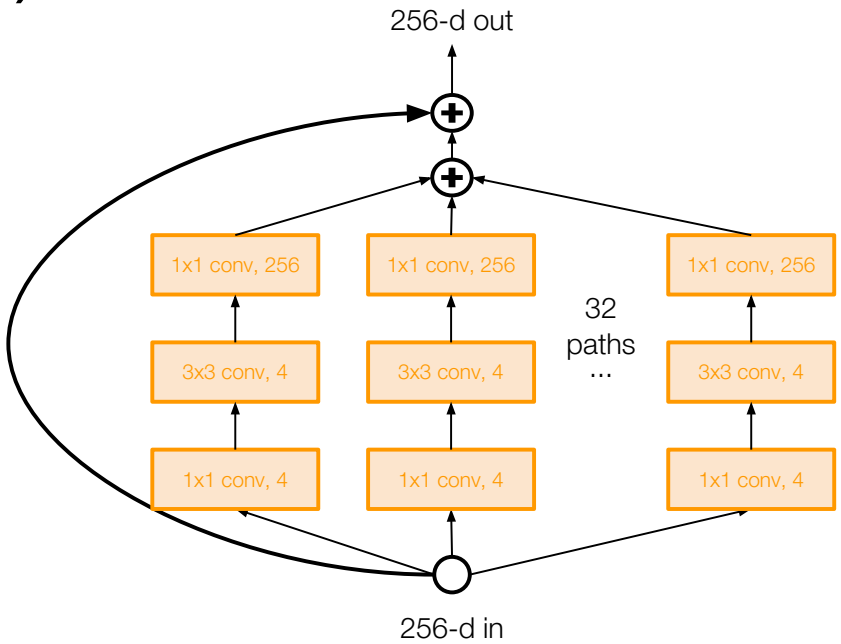
Improving ResNets...

Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways ("cardinality")
- Parallel pathways similar in spirit to Inception module





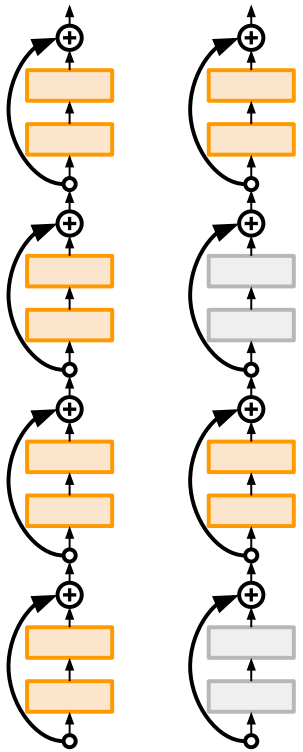
Other Architecture

Improving ResNets...

Deep Networks with Stochastic Depth

[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time



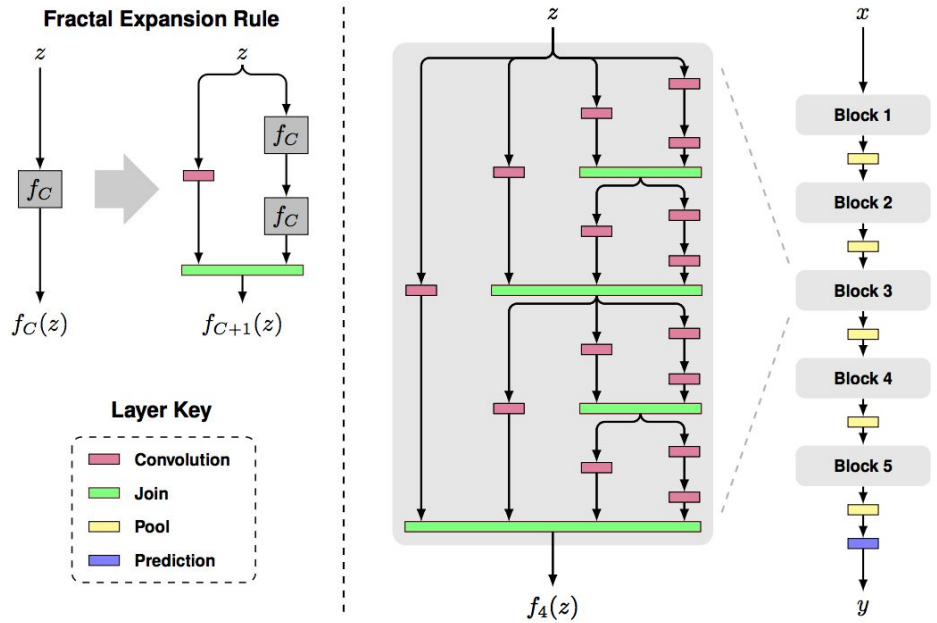
Other Architecture

Beyond ResNets...

FractalNet: Ultra-Deep Neural Networks without Residuals

[Larsson et al. 2017]

- Argues that key is transitioning effectively from shallow to deep and residual representations are not necessary
- Fractal architecture with both shallow and deep paths to output
- Trained with dropping out sub-paths
- Full network at test time



Figures copyright Larsson et al., 2017. Reproduced with permission.

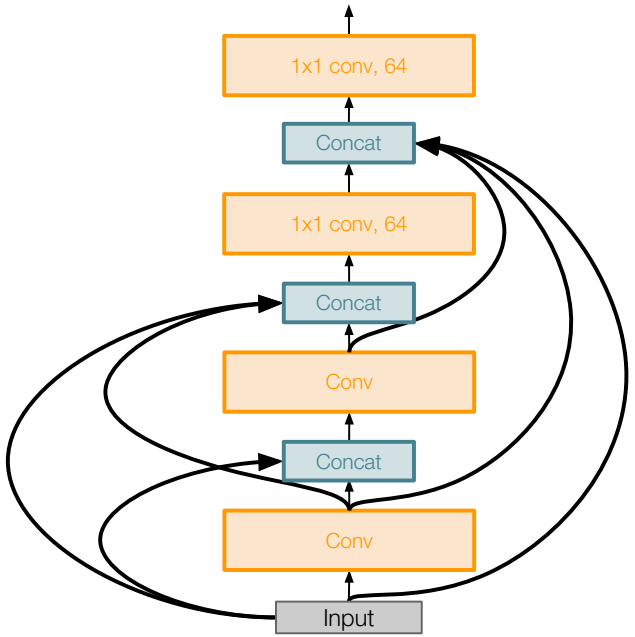
Other Architecture

Beyond ResNets...

Densely Connected Convolutional Networks

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse



Dense Block

SqueezeNet Strategies

Strategy 1: Replace 3×3 by 1×1 filters

Strategy 2: Decrease # input channels of 3×3 filters

Strategy 3: Delay downsampling of the networks

Other Architecture

Efficient networks...

SqueezeNet: AlexNet-level Accuracy With 50x Fewer Parameters and <0.5Mb Model Size

[Iandola et al. 2017]

- Fire modules consisting of a 'squeeze' layer with 1x1 filters feeding an 'expand' layer with 1x1 and 3x3 filters
- AlexNet level accuracy on ImageNet with 50x fewer parameters
- Can compress to 510x smaller than AlexNet (0.5Mb)

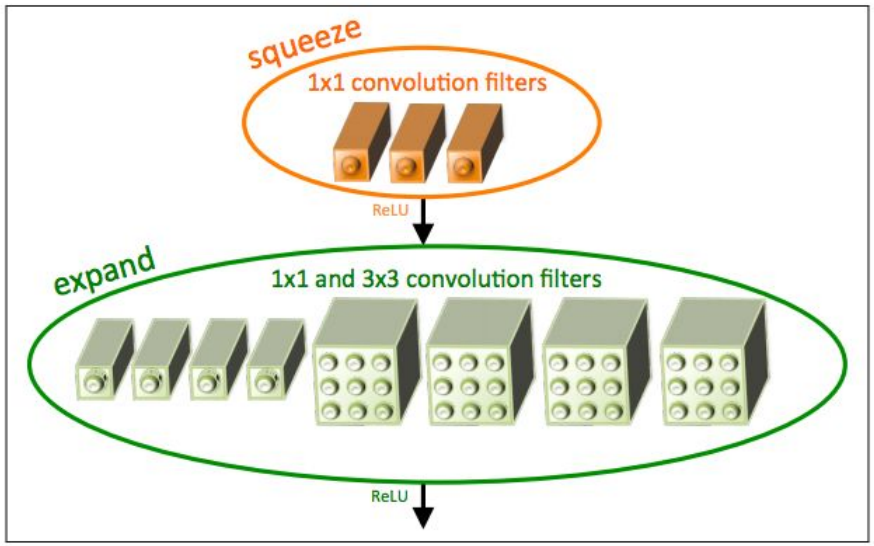


Figure copyright Iandola, Han, Moskewicz, Ashraf, Dally, Keutzer, 2017. Reproduced with permission.

Other Architecture

Summary: CNN Architectures

Case Studies

- AlexNet
- VGG
- GoogLeNet
- ResNet

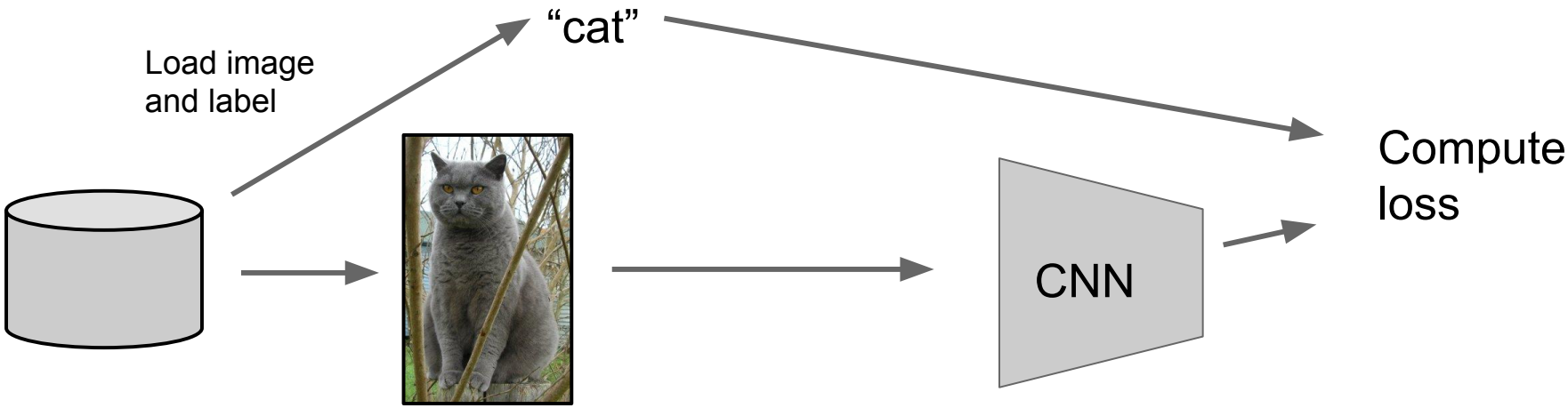
Also....

- NiN (Network in Network)
- Wide ResNet
- ResNeXT
- Stochastic Depth
- DenseNet
- FractalNet
- SqueezeNet

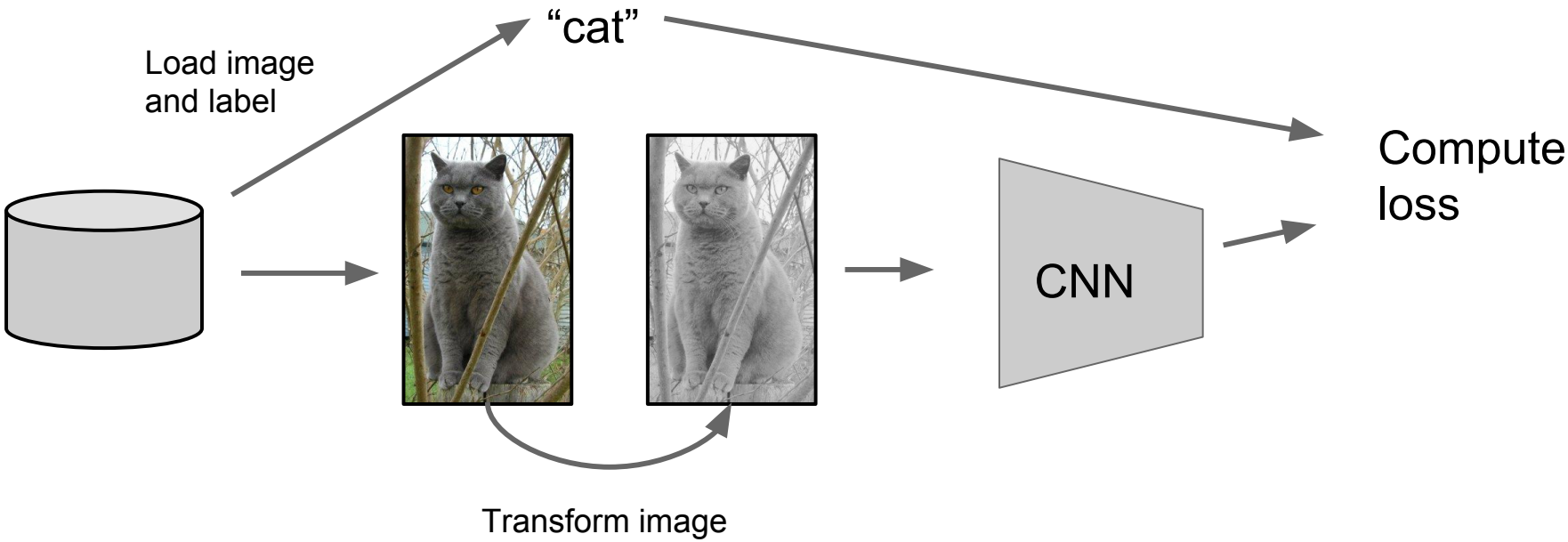
Some CNN tricks

- Data augmentation
- Transfer learning
- Use of small filters
- Implementing CNN efficiently
- Use of GPUs
- About floating point precision

Data Augmentation



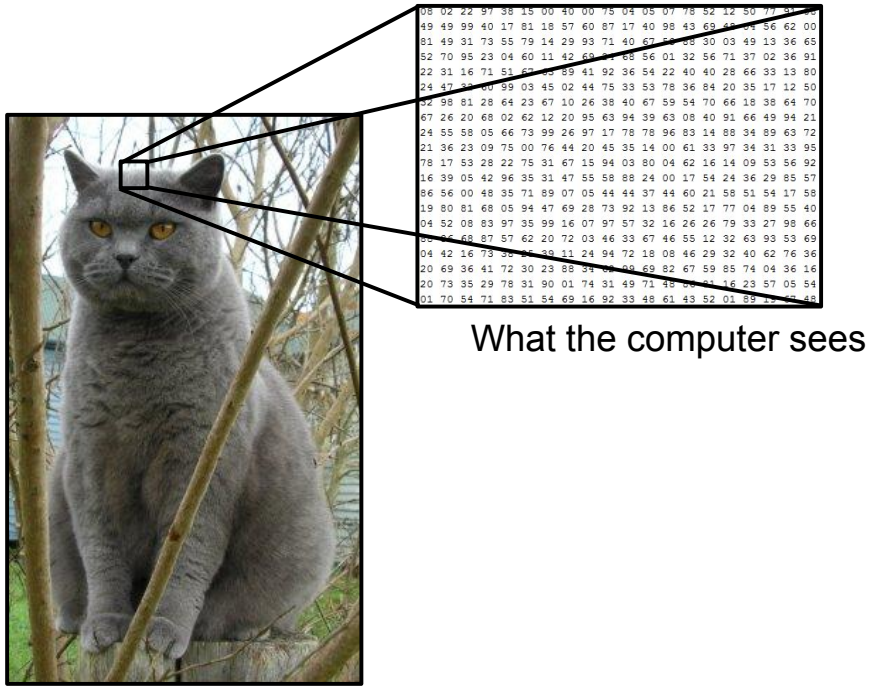
Data Augmentation



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 11 - 13 17 Feb 2016

Data Augmentation

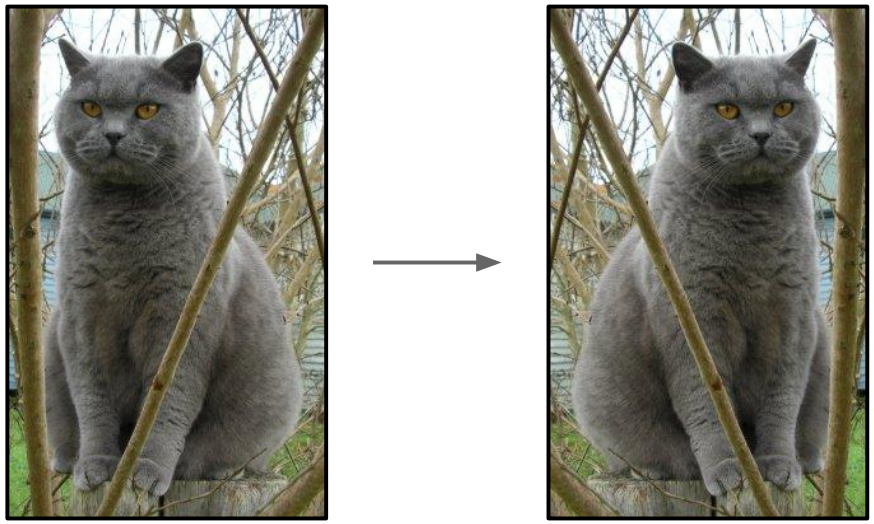
- Change the pixels without changing the label
- Train on transformed data
- VERY widely used



What the computer sees

Data Augmentation

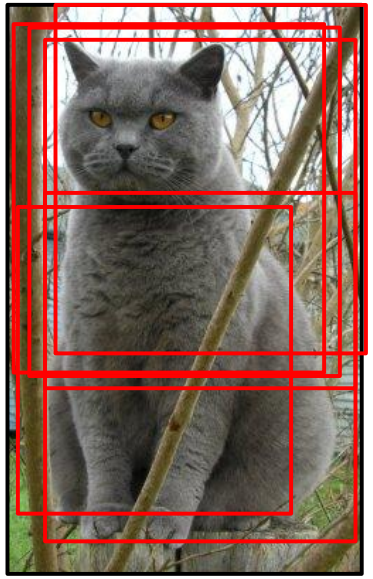
1. Horizontal flips



Data Augmentation

2. Random crops/scales

Training: sample random crops / scales



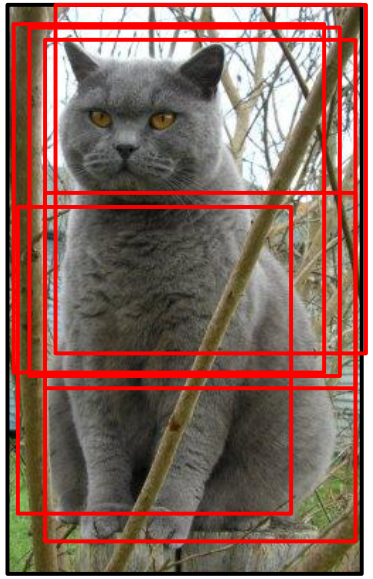
Data Augmentation

2. Random crops/scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



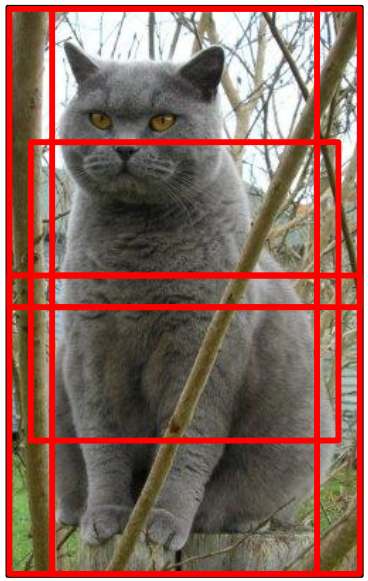
Data Augmentation

2. Random crops/scales

Training: sample random crops / scales

ResNet:

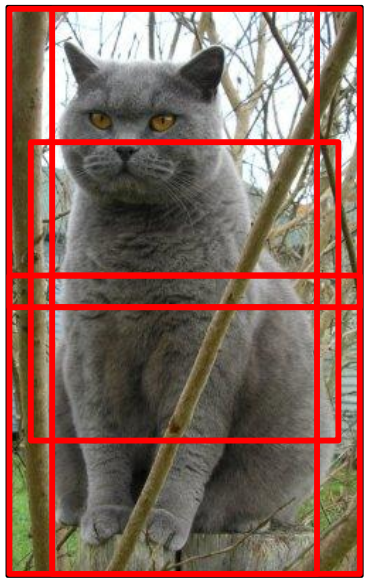
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Testing: average a fixed set of crops

Data Augmentation

2. Random crops/scales



Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

Testing: average a fixed set of crops

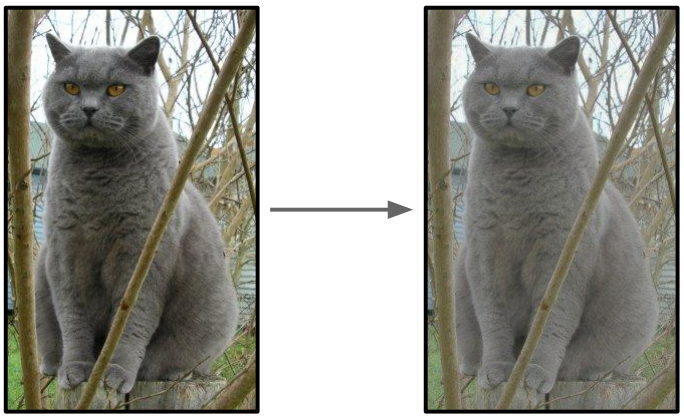
ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data Augmentation

3. Color jitter

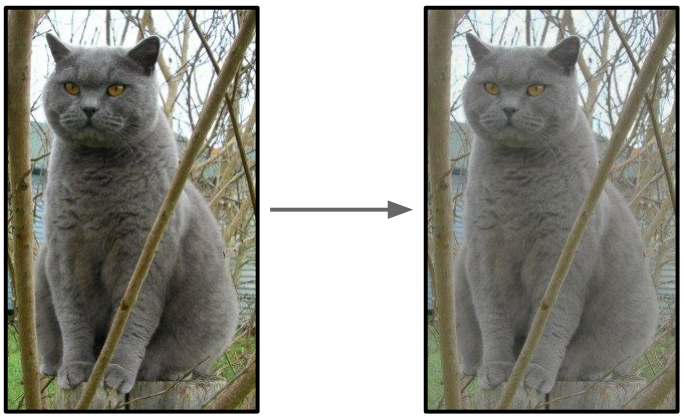
Simple:
Randomly jitter contrast



Data Augmentation

3. Color jitter

Simple:
Randomly jitter contrast



Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a "color offset" along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012],* ResNet, etc)

Data Augmentation

4. Get creative!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Data Augmentation: Takeaway

- Simple to implement, use it
- Especially useful for small datasets
- Fits into framework of noise / marginalization

Don't necesarily need lots of data for CNN

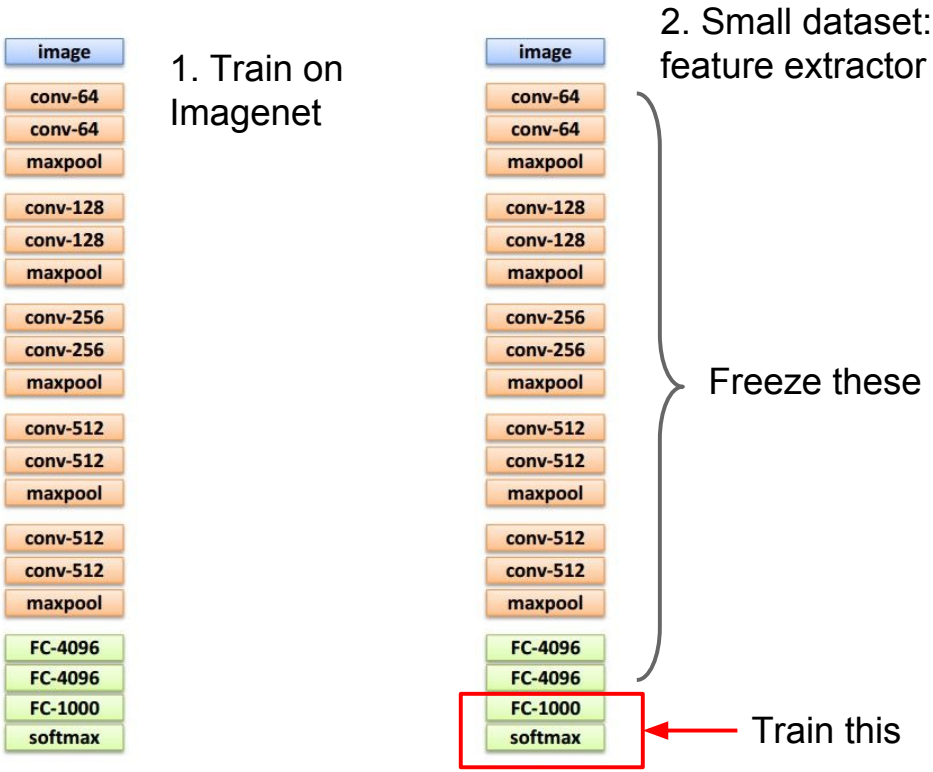
Transfer Learning with CNNs

1. Train on Imagenet

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 11 - 27 17 Feb 2016

Don't necesarily need lots of data for CNN

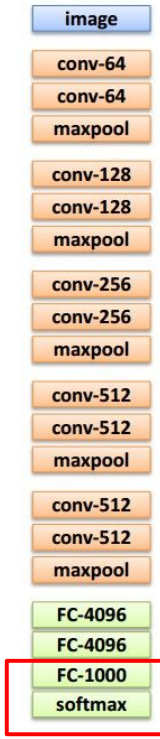
Transfer Learning with CNNs



Don't necesarily need lots of data for CNN

Transfer Learning with CNNs

1. Train on Imagenet



2. Small dataset: **feature extractor**

Freeze these

Train this

3. Medium dataset: **finetuning**

more data = retrain more of the network (or all of it)

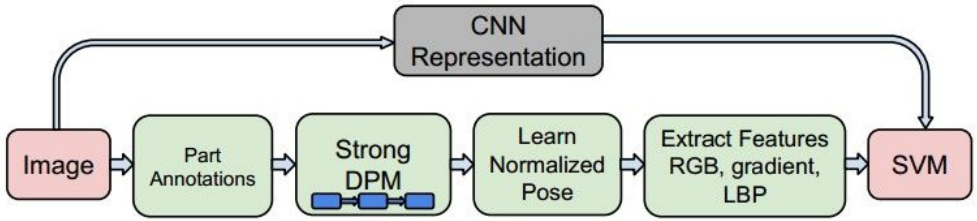
Freeze these

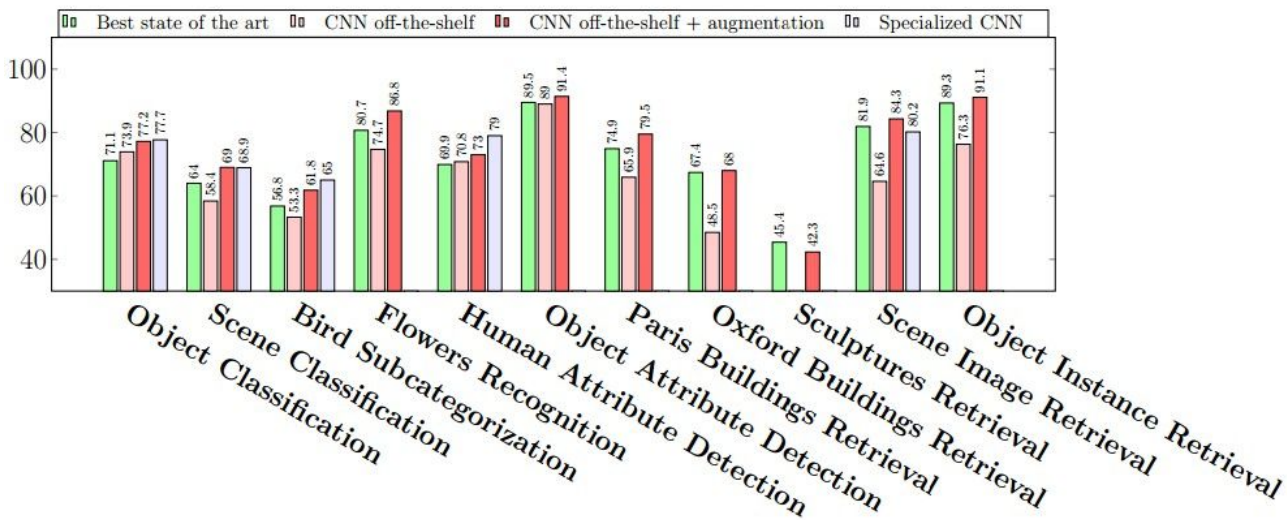
Train this

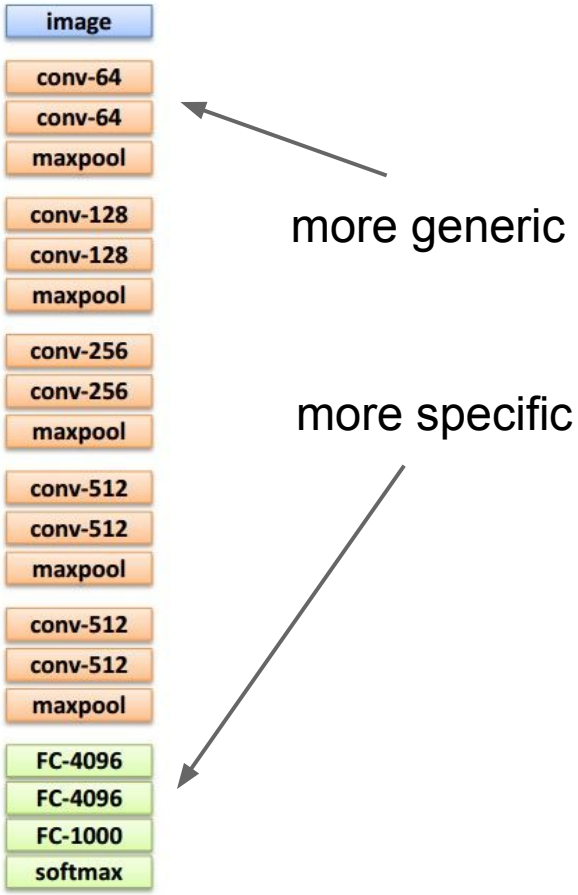
CNN Features off-the-shelf: an Astounding Baseline for Recognition [Razavian et al, 2014]

DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition [Donahue, Jia*, et al., 2013]*

	$DeCAF_6$	$DeCAF_7$
LogReg	**40.94 ± 0.3**	40.84 ± 0.3
SVM	39.36 ± 0.3	40.66 ± 0.3
Xiao et al. (2010)	38.0	

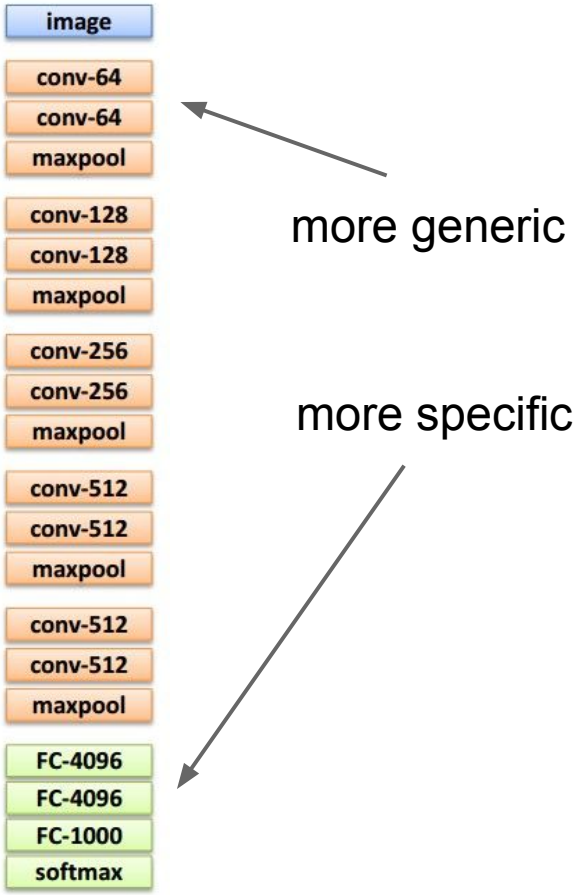




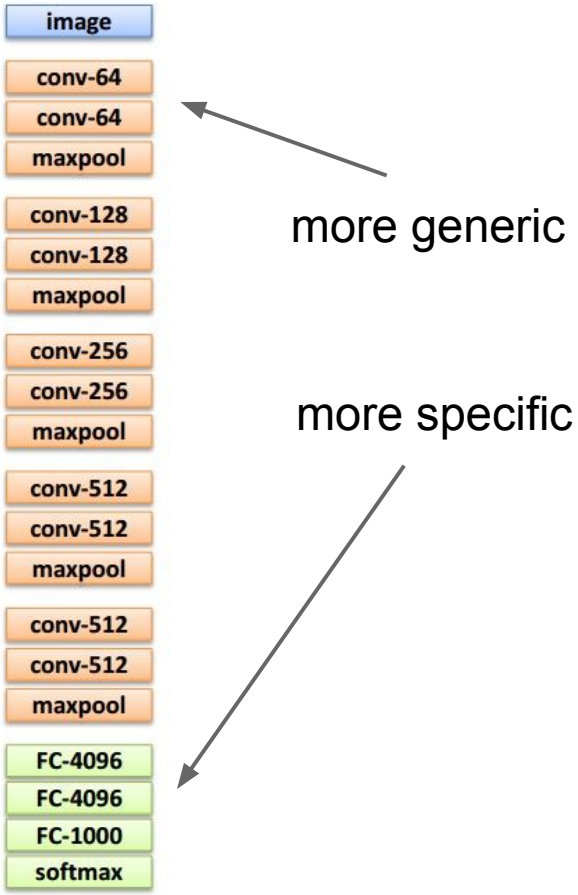


	very similar dataset	very different dataset
very little data	?	?
quite a lot of data	?	?

Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 11 - 32 17 Feb 2016



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Takeaway for your projects/beyond:

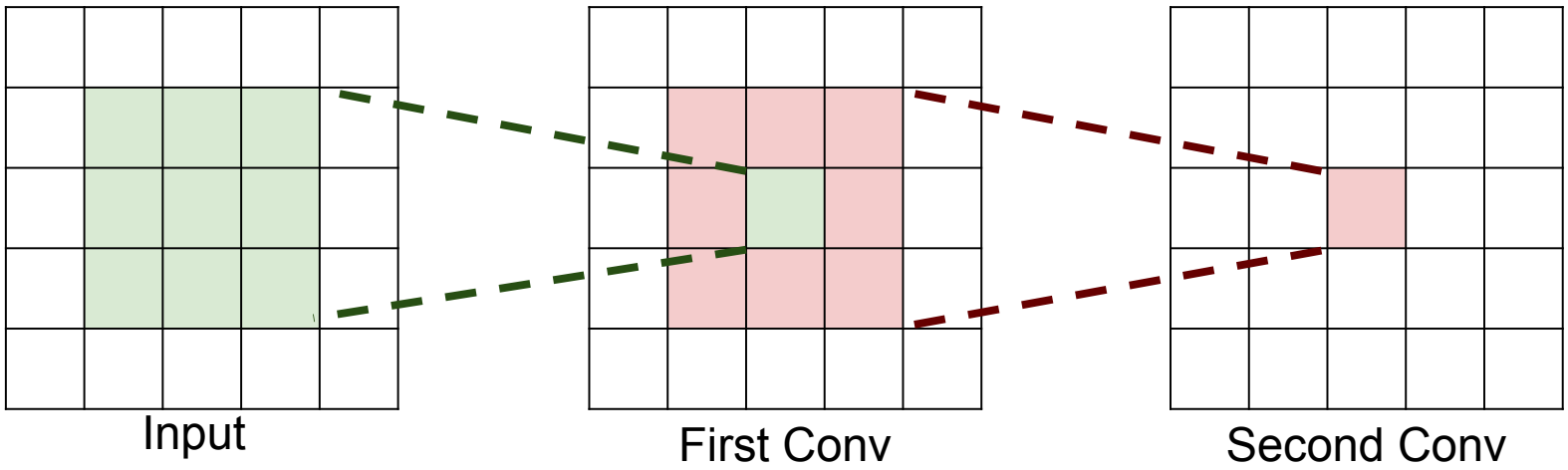
Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big ConvNet there.
2. Transfer learn to your dataset

Caffe ConvNet library has a "**Model Zoo**" of pretrained models:
https://github.com/BVLC/caffe/wiki/Model-Zoo

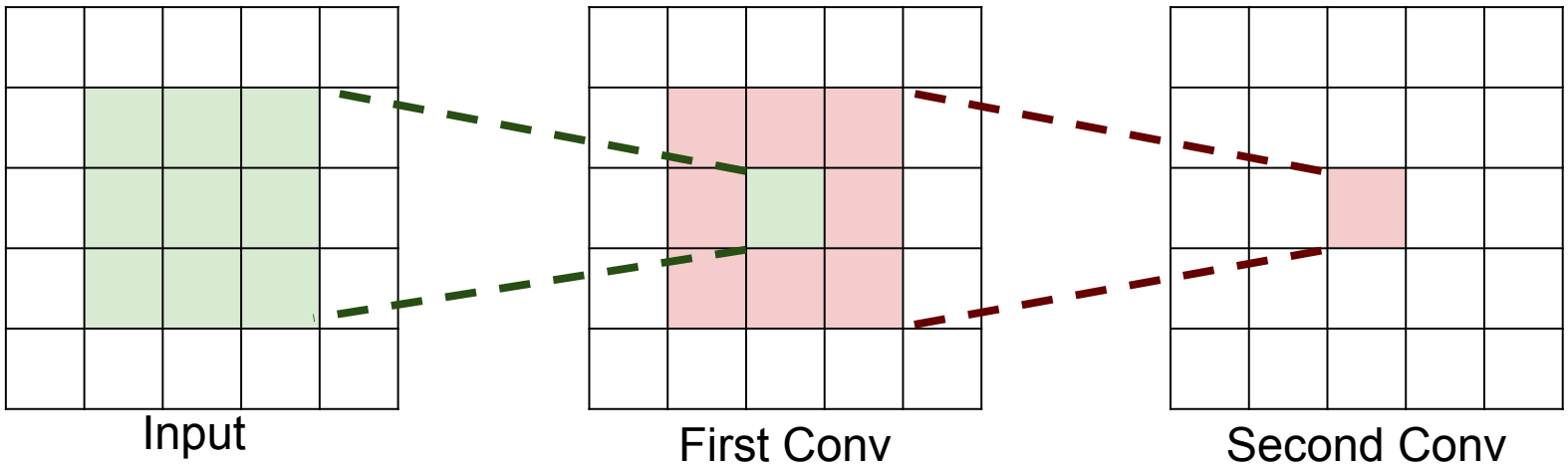
The power of small filters

Suppose we stack two 3x3 conv layers (stride 1)
Each neuron sees 3x3 region of previous activation map



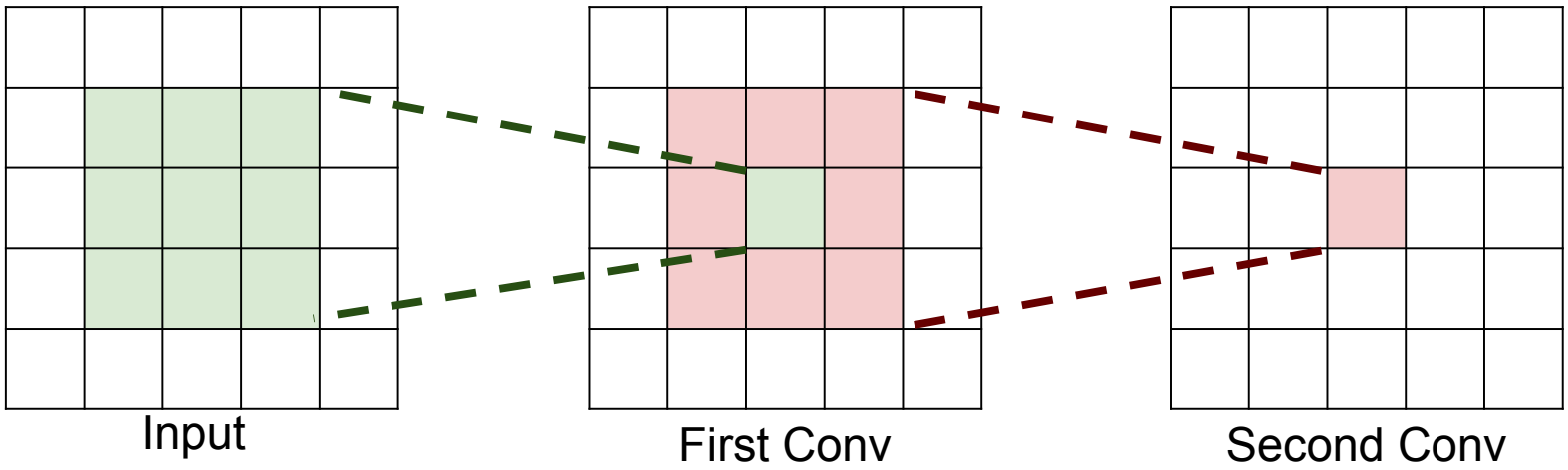
The power of small filters

Question: How big of a region in the input does a neuron on the second conv layer see?



The power of small filters

Question: How big of a region in the input does a neuron on the second conv layer see?
Answer: 5 x 5



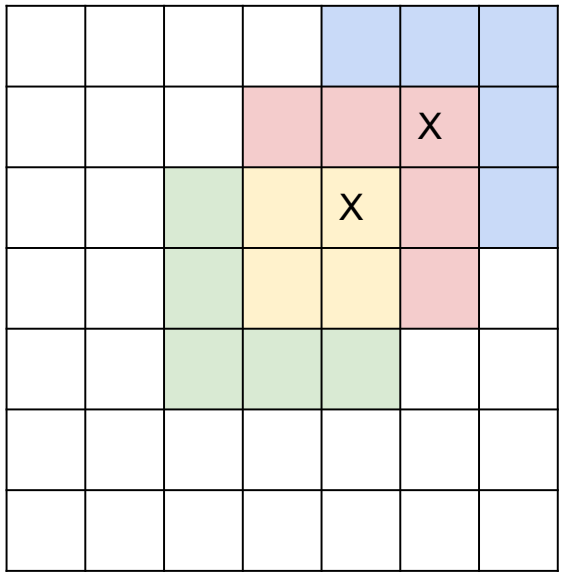
The power of small filters

Question: If we stack **three** 3x3 conv layers, how big of an input region does a neuron in the third layer see?

The power of small filters

Question: If we stack **three** 3x3 conv layers, how big of an input region does a neuron in the third layer see?

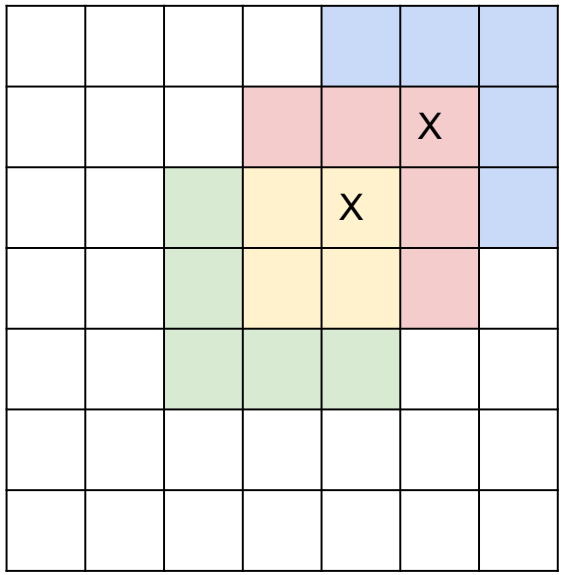
Answer: 7 x 7



The power of small filters

Question: If we stack **three** 3x3 conv layers, how big of an input region does a neuron in the third layer see?

Answer: 7 x 7



Three 3 x 3 conv gives similar representational power as a single 7 x 7 convolution

The power of small filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

The power of small filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7 x 7 filters

Number of weights:

three CONV with 3 x 3 filters

Number of weights:

The power of small filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7 x 7 filters

Number of weights:
= C x (7 x 7 x C) = **49 C^2**

three CONV with 3 x 3 filters

Number of weights:
= 3 x C x (3 x 3 x C) = **27 C^2**

The power of small filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7 x 7 filters

Number of weights:
= C x (7 x 7 x C) = **49 C^2**

three CONV with 3 x 3 filters

Number of weights:
= 3 x C x (3 x 3 x C) = **27 C^2**

Fewer parameters, more nonlinearity = GOOD

The power of small filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7 x 7 filters

Number of weights:
= C x (7 x 7 x C) = 49 C^2

Number of multiply-adds:

three CONV with 3 x 3 filters

Number of weights:
= 3 x C x (3 x 3 x C) = 27 C^2

Number of multiply-adds:

The power of small filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7 x 7 filters

Number of weights:
= C x (7 x 7 x C) = 49 C^2

Number of multiply-adds:
= (H x W x C) x (7 x 7 x C)
= **49 HWC^2**

three CONV with 3 x 3 filters

Number of weights:
= 3 x C x (3 x 3 x C) = 27 C^2

Number of multiply-adds:
= 3 x (H x W x C) x (3 x 3 x C)
= **27 HWC^2**

The power of small filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7 x 7 filters

Number of weights:
= C x (7 x 7 x C) = 49 C^2

Number of multiply-adds:
= **49 HWC^2**

three CONV with 3 x 3 filters

Number of weights:
= 3 x C x (3 x 3 x C) = 27 C^2

Number of multiply-adds:
= **27 HWC^2**

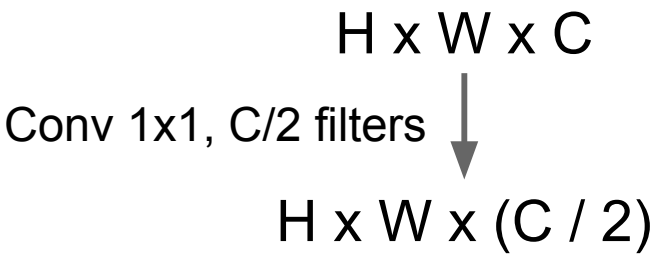
Less compute, more nonlinearity = GOOD

The power of small filters

Why stop at 3 x 3 filters? Why not try 1 x 1?

The power of small filters

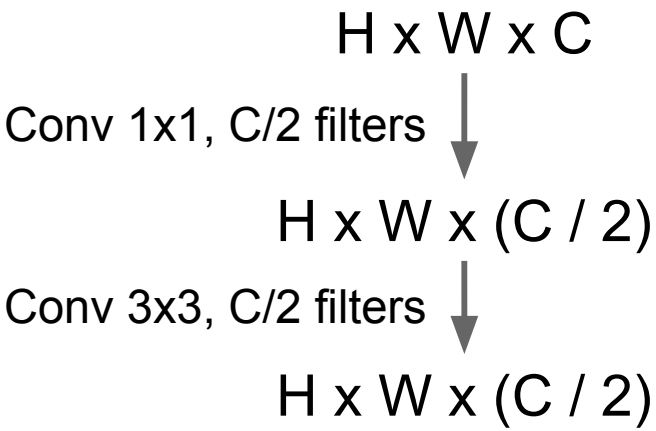
Why stop at 3 x 3 filters? Why not try 1 x 1?



1. "bottleneck" 1 x 1 conv to reduce dimension

The power of small filters

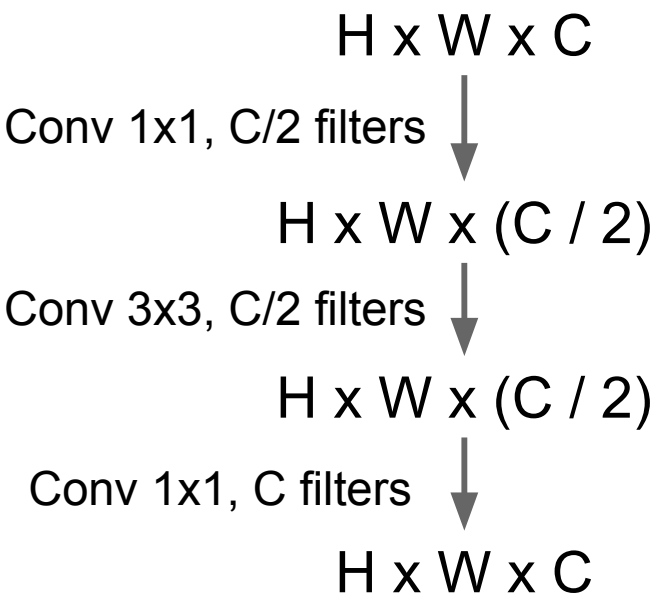
Why stop at 3 x 3 filters? Why not try 1 x 1?



1. "bottleneck" 1 x 1 conv to reduce dimension
2. 3 x 3 conv at reduced dimension

The power of small filters

Why stop at 3 x 3 filters? Why not try 1 x 1?

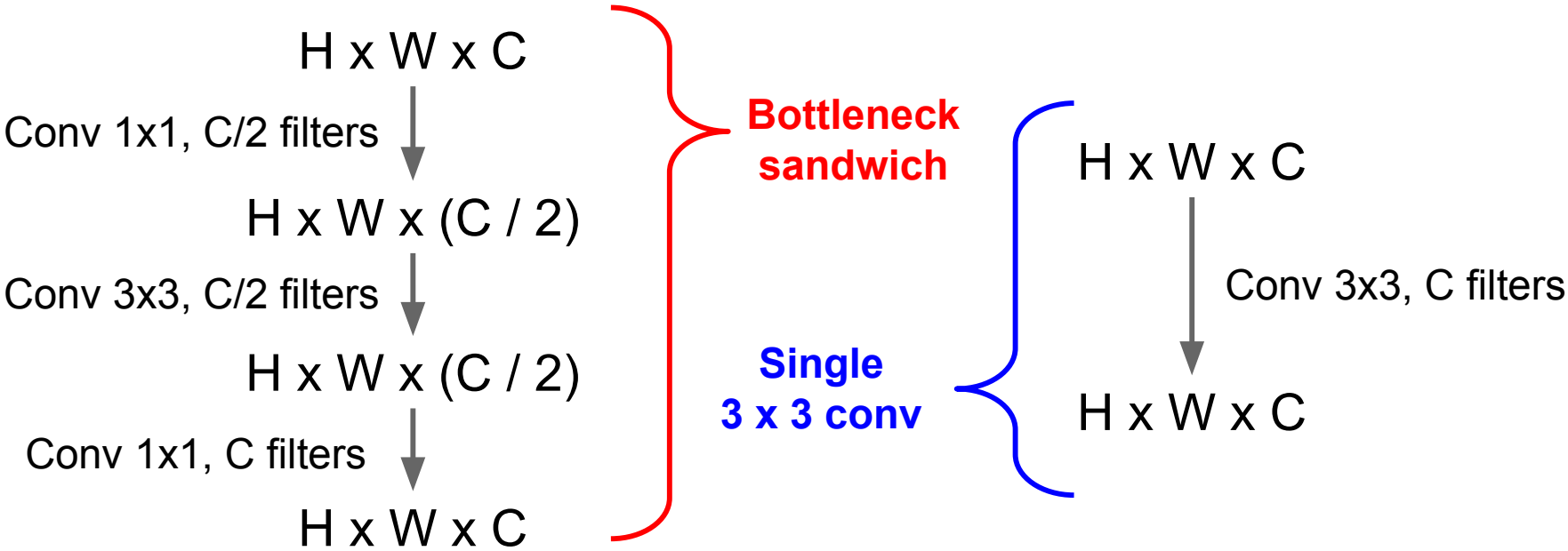


1. "bottleneck" 1 x 1 conv to reduce dimension
2. 3 x 3 conv at reduced dimension
3. Restore dimension with another 1 x 1 conv

[Seen in Lin et al, "Network in Network", GoogLeNet, ResNet]

The power of small filters

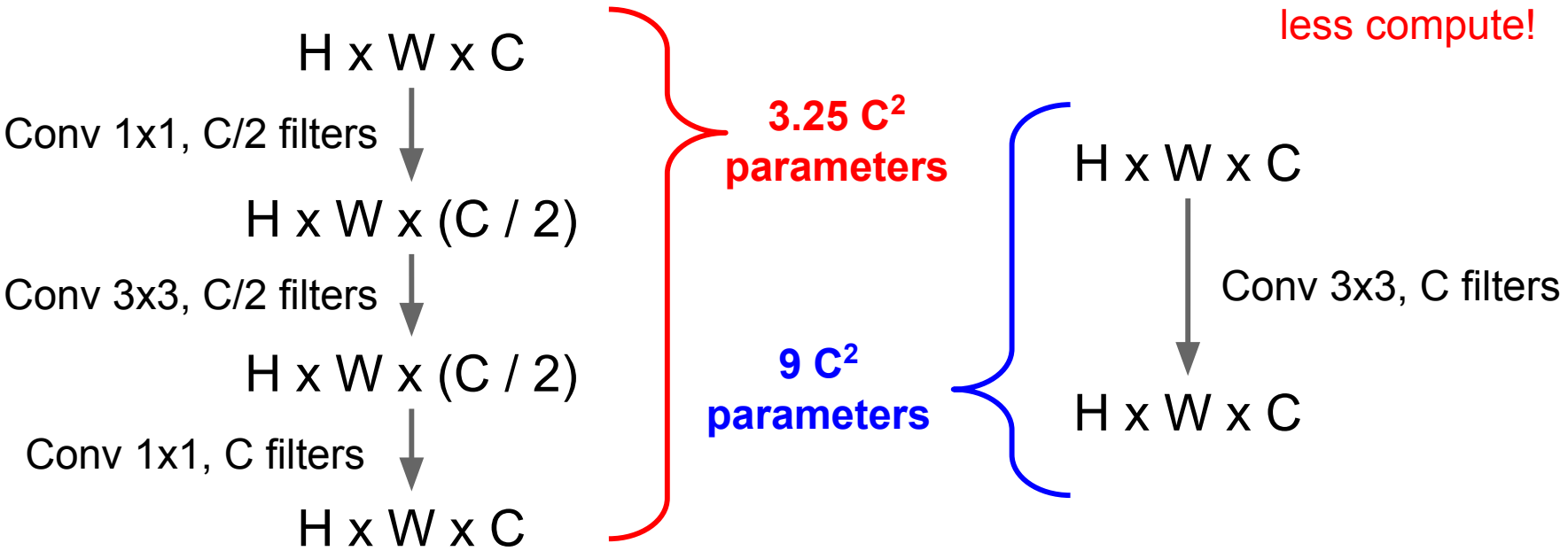
Why stop at 3 x 3 filters? Why not try 1 x 1?



The power of small filters

Why stop at 3 x 3 filters? Why not try 1 x 1?

More nonlinearity, fewer params, less compute!

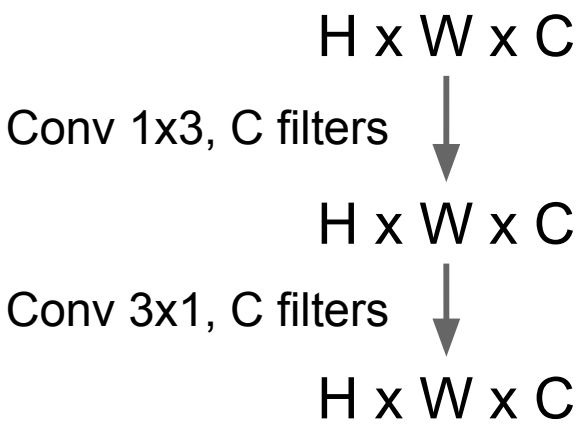


The power of small filters

Still using 3 x 3 filters … can we break it up?

The power of small filters

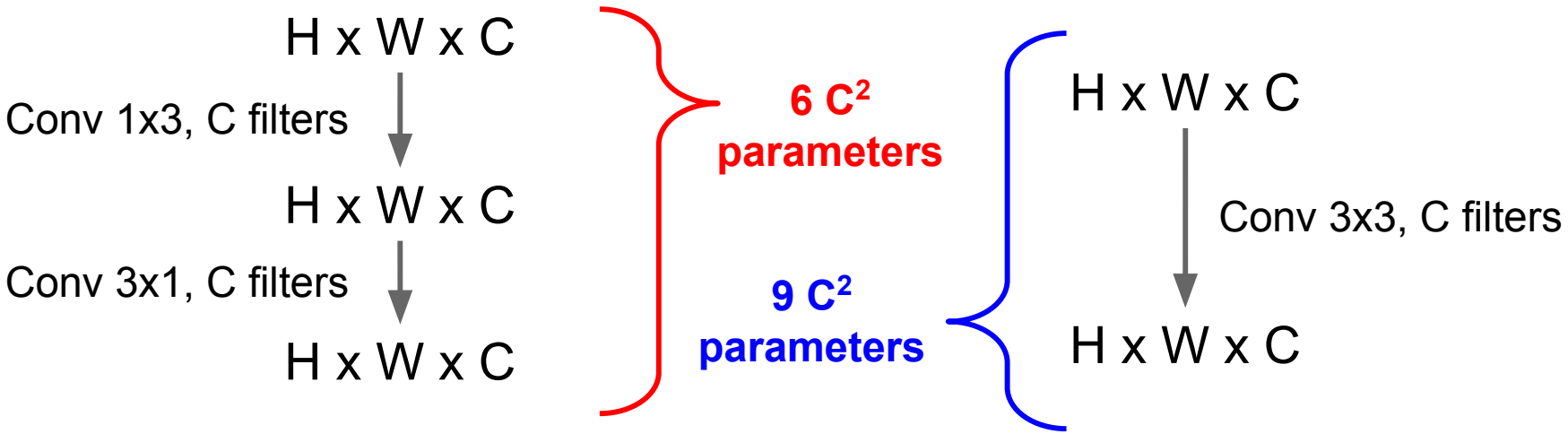
Still using 3 x 3 filters … can we break it up?



The power of small filters

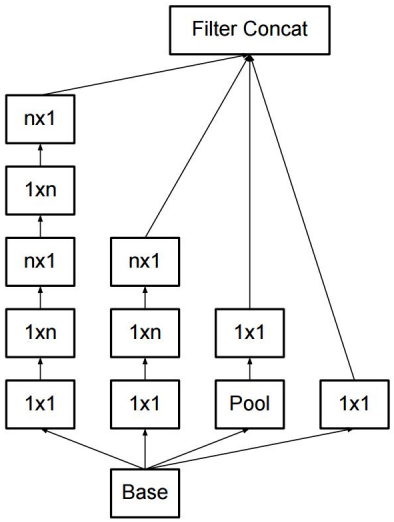
Still using 3 x 3 filters … can we break it up?

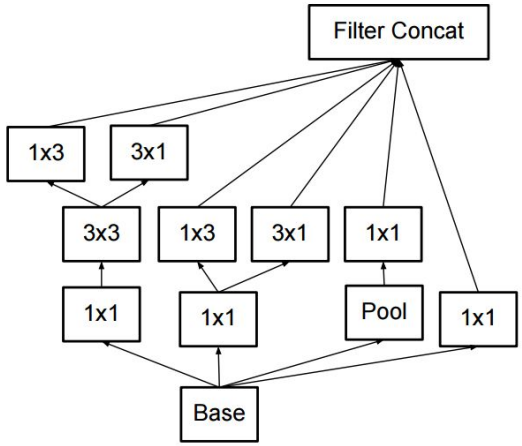
More nonlinearity, fewer params, less compute!

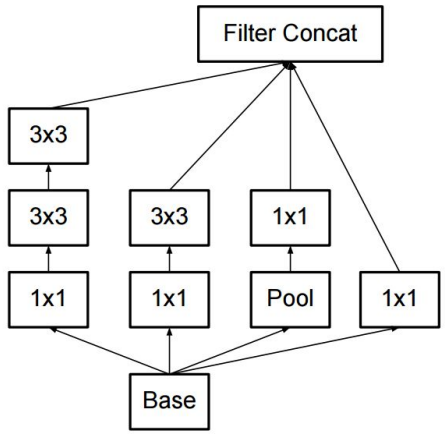


The power of small filters

Latest version of GoogLeNet incorporates all these ideas







Szegedy et al, “Rethinking the Inception Architecture for Computer Vision”

How to stack convolutions: Recap

- Replace large convolutions (5 x 5, 7 x 7) with stacks of 3 x 3 convolutions
- 1 x 1 “bottleneck” convolutions are very efficient
- Can factor N x N convolutions into 1 x N and N x 1
- All of the above give fewer parameters, less compute, more nonlinearity

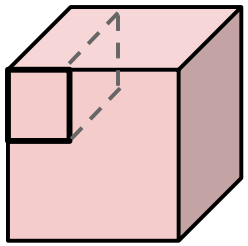
Implementing Convolutions: im2col

There are highly optimized matrix multiplication routines for just about every platform

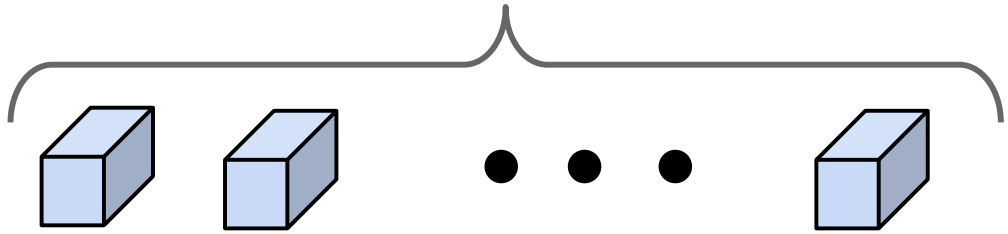
Can we turn convolution into matrix multiplication?

Implementing Convolutions: im2col

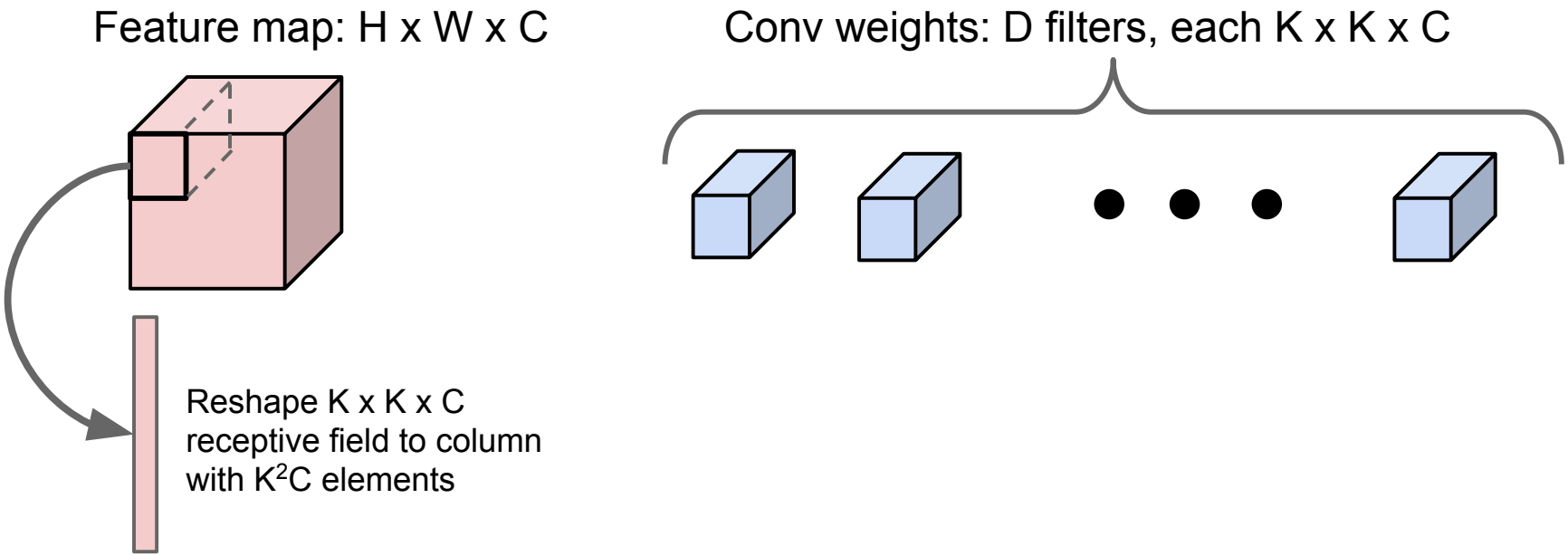
Feature map: H x W x C



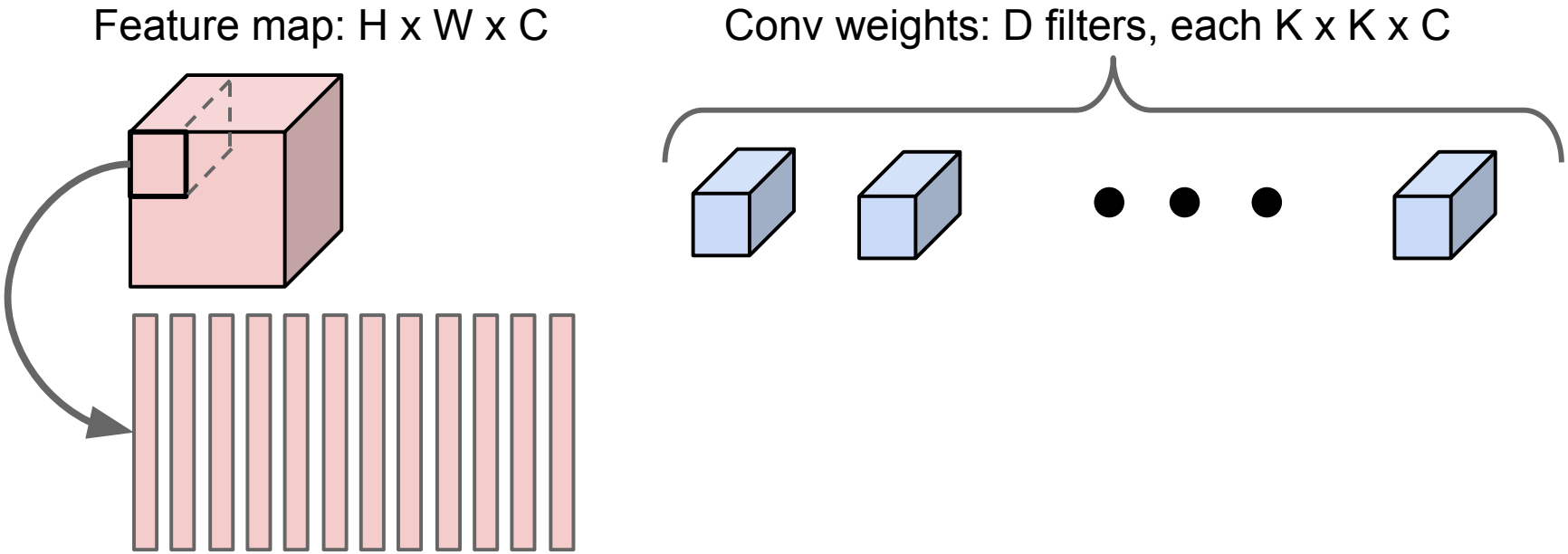
Conv weights: D filters, each K x K x C



Implementing Convolutions: im2col



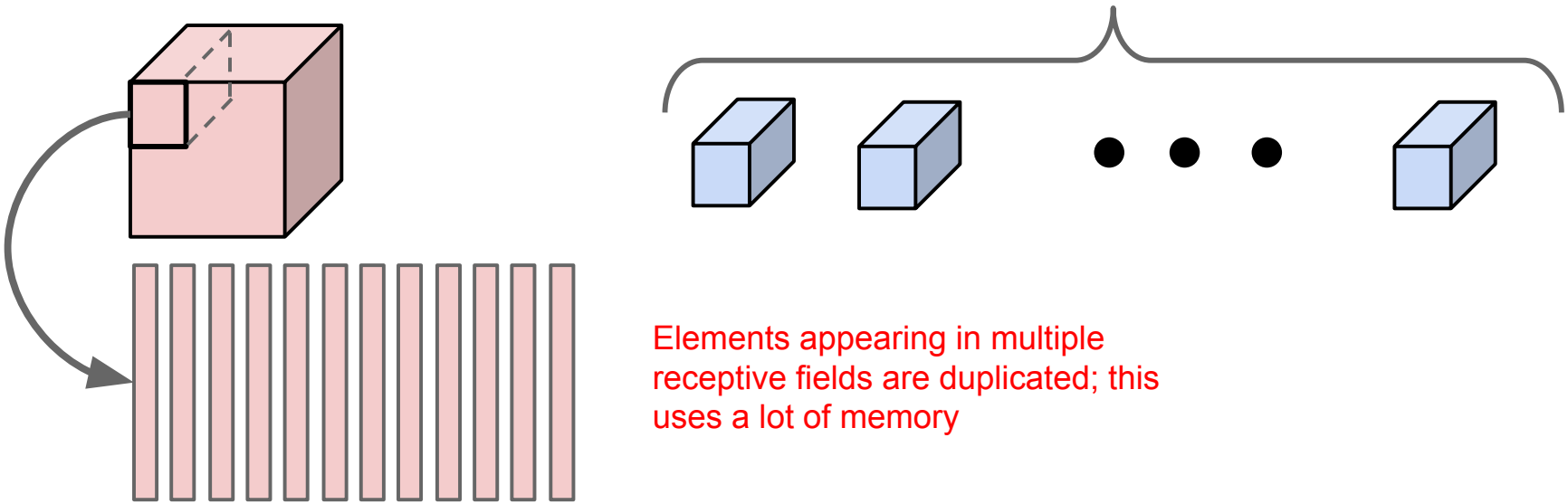
Implementing Convolutions: im2col



Implementing Convolutions: im2col

Feature map: H x W x C

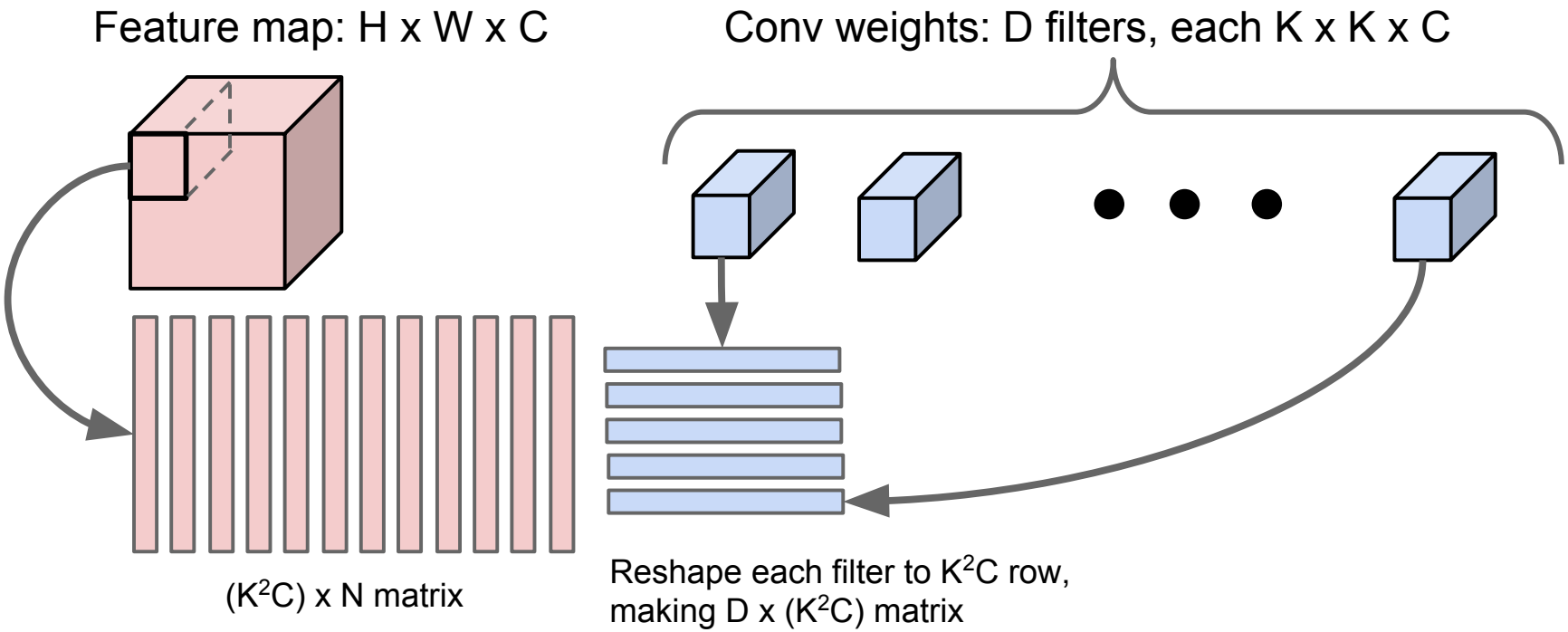
Conv weights: D filters, each K x K x C



Elements appearing in multiple receptive fields are duplicated; this uses a lot of memory

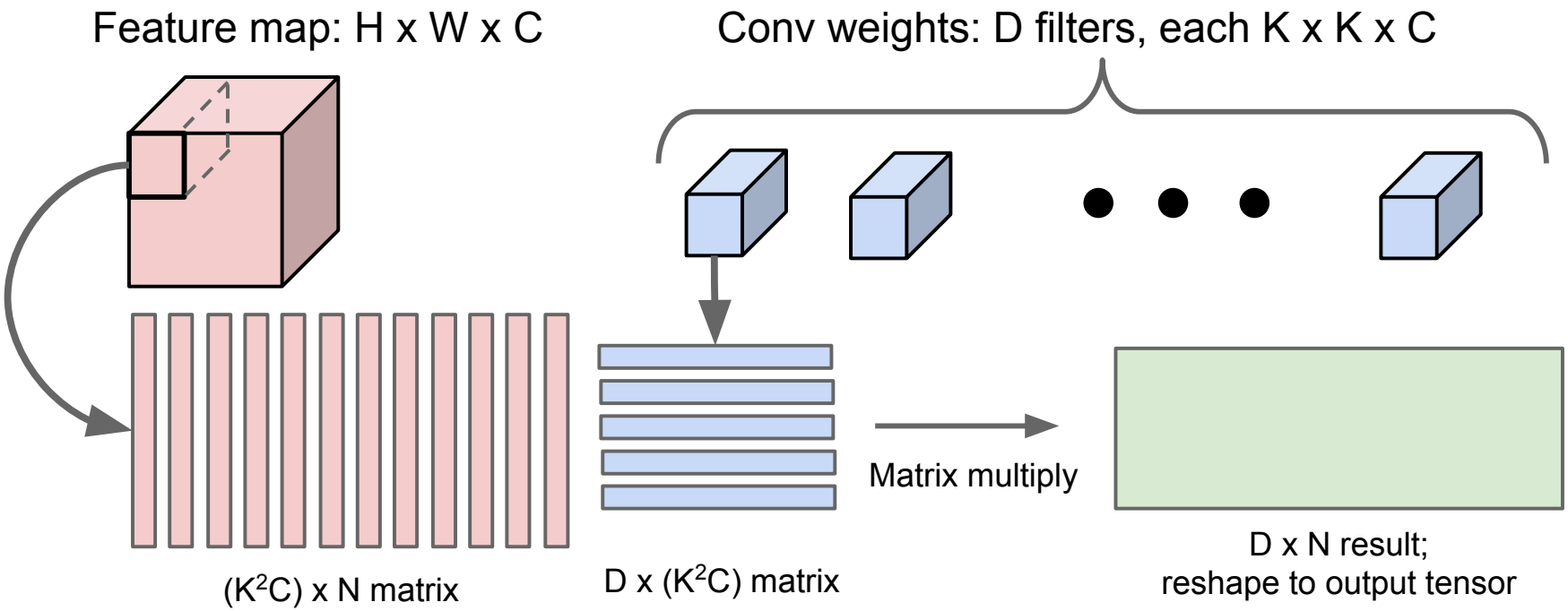
Repeat for all columns to get (K^2C) x N matrix
(N receptive field locations)

Implementing Convolutions: im2col



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 11 - 17 Feb 2016

Implementing Convolutions: im2col



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 11 - 17 Feb 2016

```
template <typename Dtype>
void ConvolutionLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      vector<Blob<Dtype>*>* top) {
  for (int i = 0; i < bottom.size(); ++i) {
    const Dtype* bottom_data = bottom[i]->gpu_data();
    Dtype* top_data = (*top)[i]->mutable_gpu_data();
    Dtype* col_data = col_buffer_.mutable_gpu_data();
    const Dtype* weight = this->blobs_[0]->gpu_data();
    int weight_offset = M_ * K_;
    int col_offset = K_ * N_;
    int top_offset = M_ * N_;
    for (int n = 0; n < num_; ++n) {
      // im2col transformation: unroll input regions for filtering
      // into column matrix for multiplication.
      im2col_gpu(bottom_data + bottom[i]->offset(n), channels_, height_,
          width_, kernel_h_, kernel_w_, pad_h_, pad_w_, stride_h_, stride_w_,
          col_data);
      // Take inner products for groups.
      for (int g = 0; g < group_; ++g) {
        caffe_gpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, M_, N_, K_,
          (Dtype)1., weight + weight_offset * g, col_data + col_offset * g,
          (Dtype)0., top_data + (*top)[i]->offset(n) + top_offset * g);
      }
      // Add bias.
      if (bias_term_) {
        caffe_gpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num_output_,
            N_, 1, (Dtype)1., this->blobs_[1]->gpu_data(),
            bias_multiplier_.gpu_data(),
            (Dtype)1., top_data + (*top)[i]->offset(n));
      }
    }
  }
}
```

Case study:
CONV forward in Caffe library

im2col

matrix multiply: call to cuBLAS

bias offset

Implementing convolutions: FFT

Convolution Theorem: The convolution of f and g is equal to the elementwise product of their Fourier Transforms:

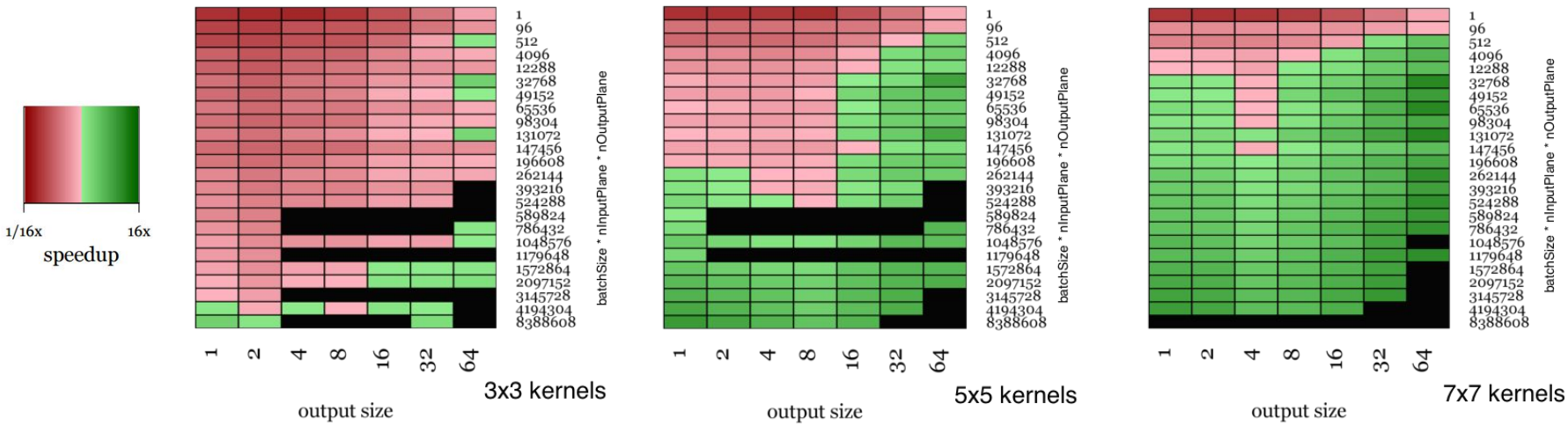
$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$$

Using the **Fast Fourier Transform**, we can compute the Discrete Fourier transform of an N-dimensional vector in O (N log N) time (also extends to 2D images)

Implementing convolutions: FFT

1. Compute FFT of weights: F(W)
2. Compute FFT of image: F(X)
3. Compute elementwise product: $F(W) \circ F(X)$
4. Compute inverse FFT: $Y = F^{-1}(F(W) \circ F(X))$

Implementing convolutions: FFT



FFT convolutions get a big speedup for larger filters
Not much speedup for 3x3 filters =(

Vasilache et al, Fast Convolutional Nets With fbfft: A GPU Performance Evaluation

Implementing convolution: "Fast Algorithms"

Naive matrix multiplication: Computing product of two N x N matrices takes $O(N^3)$ operations

Strassen's Algorithm: Use clever arithmetic to reduce complexity to $O(N^{\log_2(7)}) \sim O(N^{2.81})$

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

$$\begin{aligned}
\mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}$$

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

From Wikipedia

Implementing convolution: "Fast Algorithms"

Similar cleverness can be applied to convolutions

Lavin and Gray (2015) work out special cases for 3x3 convolutions:

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$$

$$d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

Lavin and Gray, "Fast Algorithms for Convolutional Neural Networks", 2015

Implementing convolution: "Fast Algorithms"

Huge speedups on VGG for small batches:

N	cuDNN		F(2x2,3x3)		Speedup
	msec	TFLOPS	msec	TFLOPS	
1	12.52	3.12	5.55	7.03	2.26X
2	20.36	3.83	9.89	7.89	2.06X
4	104.70	1.49	17.72	8.81	5.91X
8	241.21	1.29	33.11	9.43	7.28X
16	203.09	3.07	65.79	9.49	3.09X
32	237.05	5.27	132.36	9.43	1.79X
64	394.05	6.34	266.48	9.37	1.48X

Table 5. cuDNN versus $F(2 \times 2, 3 \times 3)$ performance on VGG Network E with fp32 data. Throughput is measured in Effective TFLOPS, the ratio of direct algorithm GFLOPs to run time.

N	cuDNN		F(2x2,3x3)		Speedup
	msec	TFLOPS	msec	TFLOPS	
1	14.58	2.68	5.53	7.06	2.64X
2	20.94	3.73	9.83	7.94	2.13X
4	104.19	1.50	17.50	8.92	5.95X
8	241.87	1.29	32.61	9.57	7.42X
16	204.01	3.06	62.93	9.92	3.24X
32	236.13	5.29	123.12	10.14	1.92X
64	395.93	6.31	242.98	10.28	1.63X

Table 6. cuDNN versus $F(2 \times 2, 3 \times 3)$ performance on VGG Network E with fp16 data.

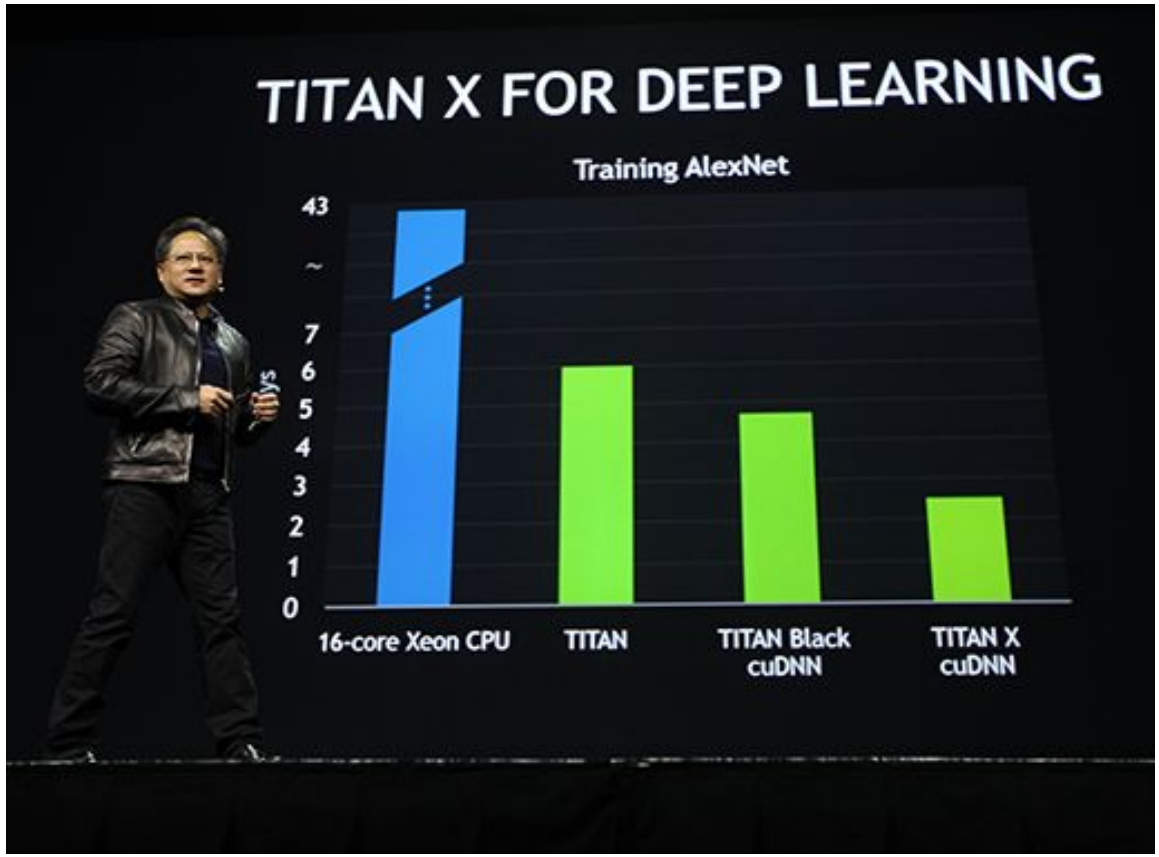
Computing Convolutions: Recap

- im2col: Easy to implement, but big memory overhead
- FFT: Big speedups for small kernels
- "Fast Algorithms" seem promising, not widely used yet

CEO of NVIDIA:
Jen-Hsun Huang

(Stanford EE Masters 1992)

GTC 2015:
Introduced new Titan X GPU by bragging about AlexNet benchmarks



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 11 - 90 17 Feb 2016

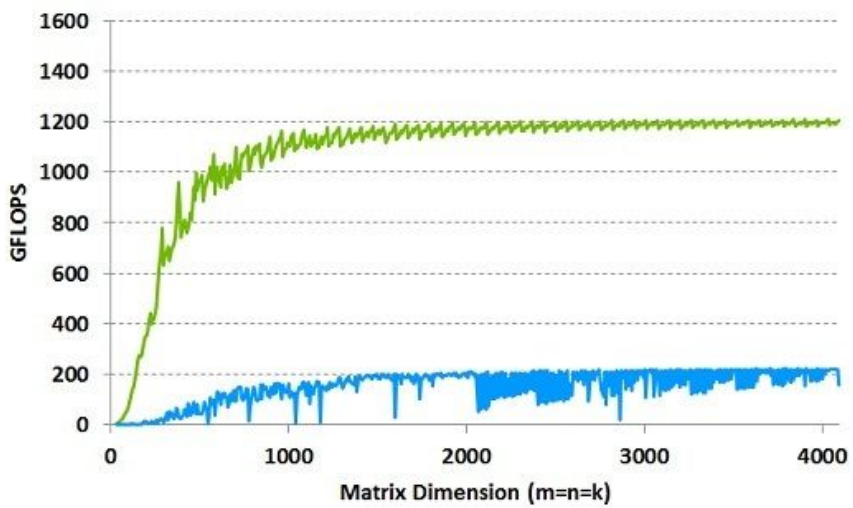
CPU
Few, fast cores (1 - 16)
Good at sequential processing

GPU
Many, slower cores (thousands)
Originally for graphics
Good at parallel computation

GPUs can be programmed

- CUDA (NVIDIA only)
  - Write C code that runs directly on the GPU
  - Higher-level APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
  - Similar to CUDA, but runs on anything
  - Usually slower :(
- Udacity: Intro to Parallel Programming https://www.udacity.com/course/cs344
  - For deep learning just use existing libraries

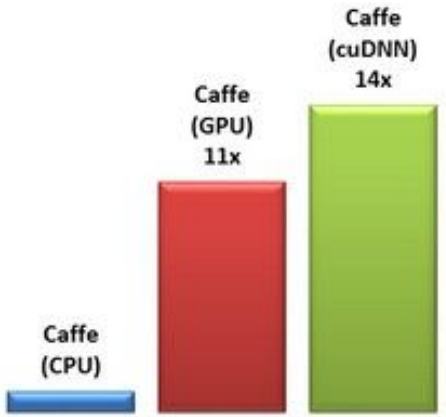
GPUs are really good
at matrix multiplication:

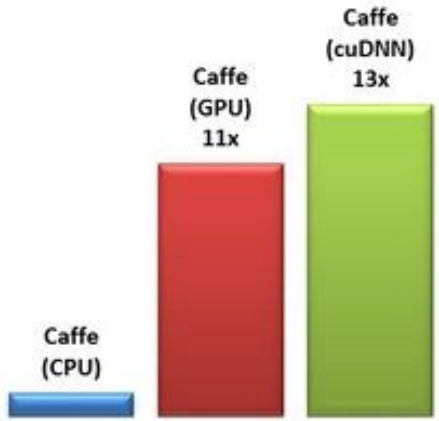


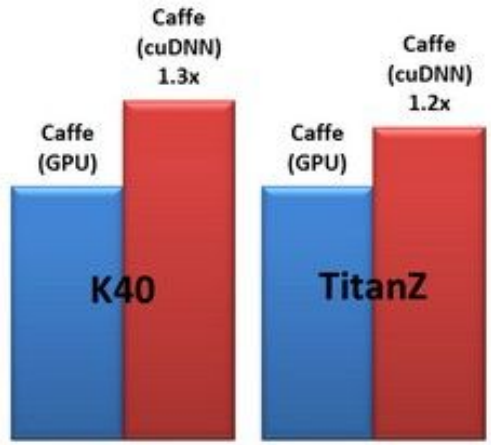
GPU: NVIDA Tesla K40
with cuBLAS

CPU: Intel E5-2697 v2
12 core @ 2.7 Ghz
with MKL

GPUs are really good at convolution (cuDNN):







All comparisons are against a 12-core Intel E5-2679v2 CPU @ 2.4GHz running Caffe with Intel MKL 11.1.3.

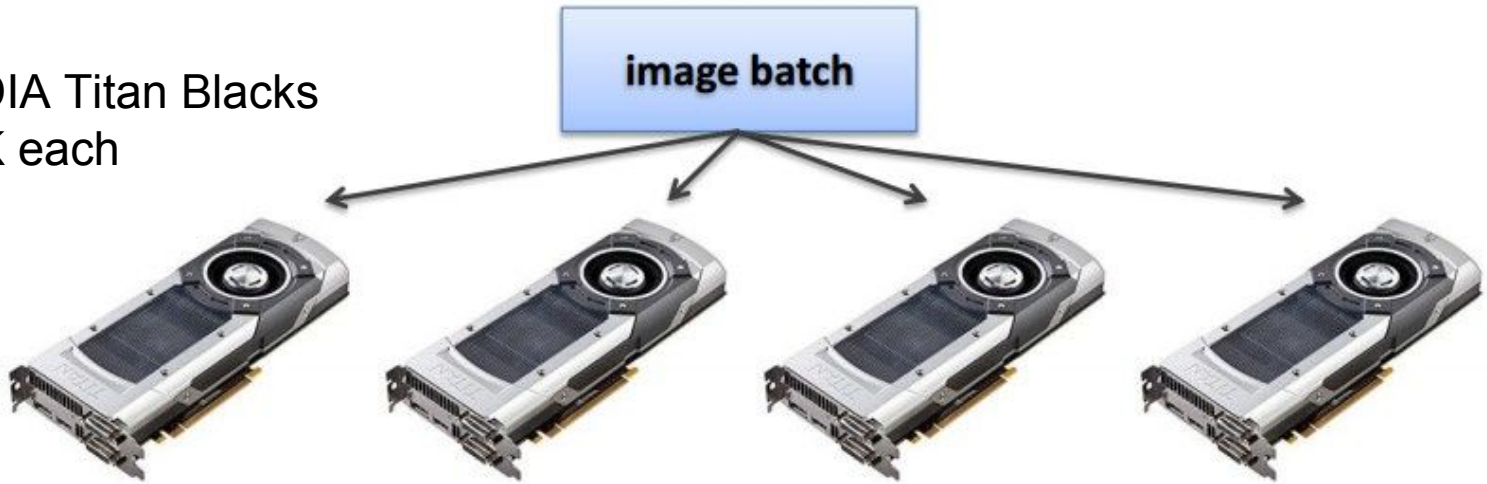
Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 11 - 94 17 Feb 2016

Even with GPUs, training can be slow

VGG: ~2-3 weeks training with 4 GPUs

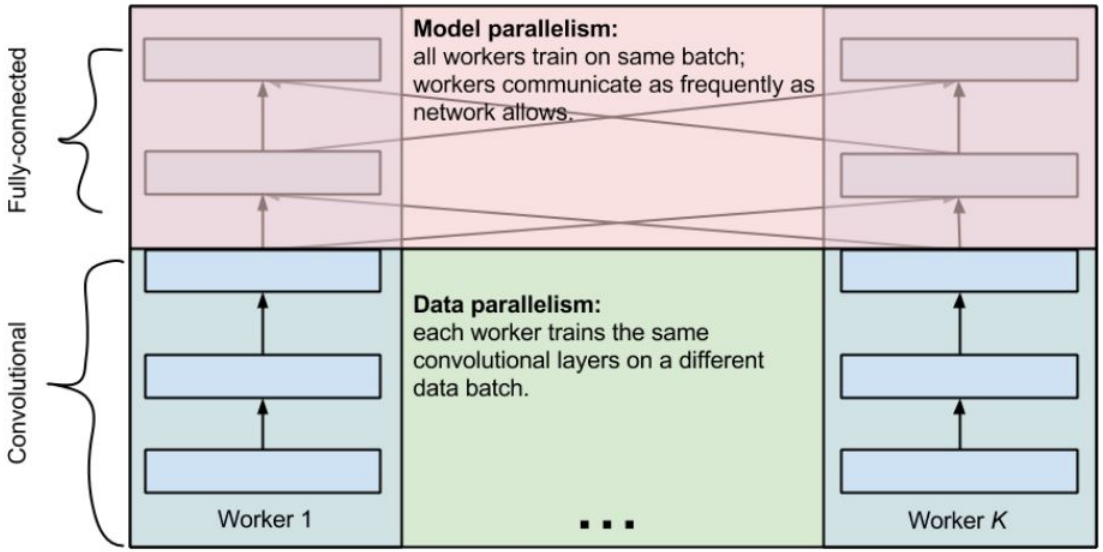
ResNet 101: 2-3 weeks with 4 GPUs

NVIDIA Titan Blacks
~$1K each



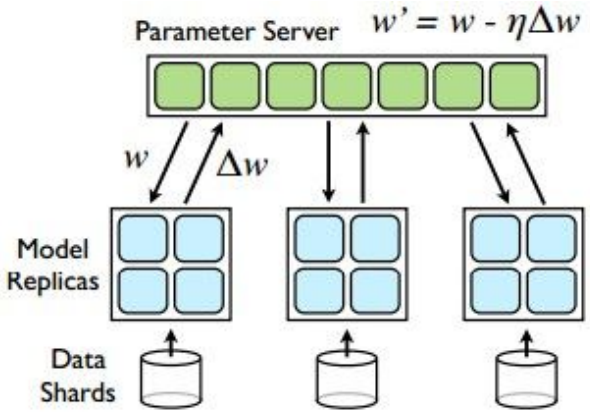
ResNet reimplemented in Torch: http://torch.ch/blog/2016/02/04/resnets.html

Multi-GPU training: More complex



Alex Krizhevsky, “One weird trick for parallelizing convolutional neural networks”

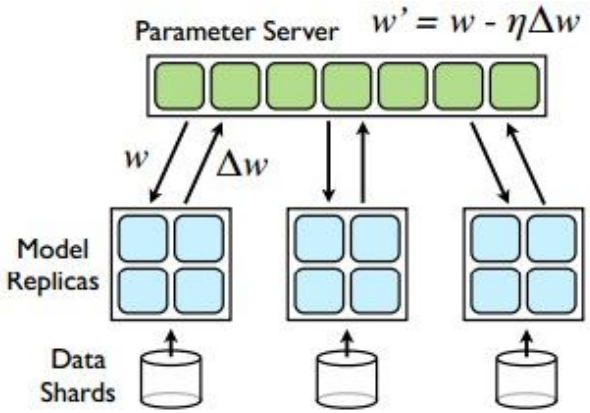
Google: Distributed CPU training



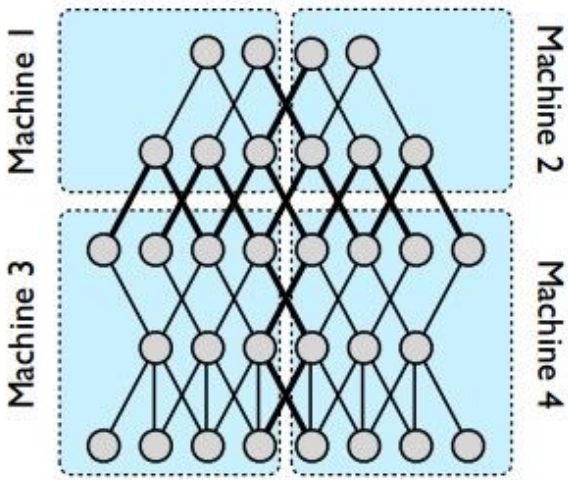
Data parallelism

[Large Scale Distributed Deep Networks, Jeff Dean et al., 2013]

Google: Distributed CPU training



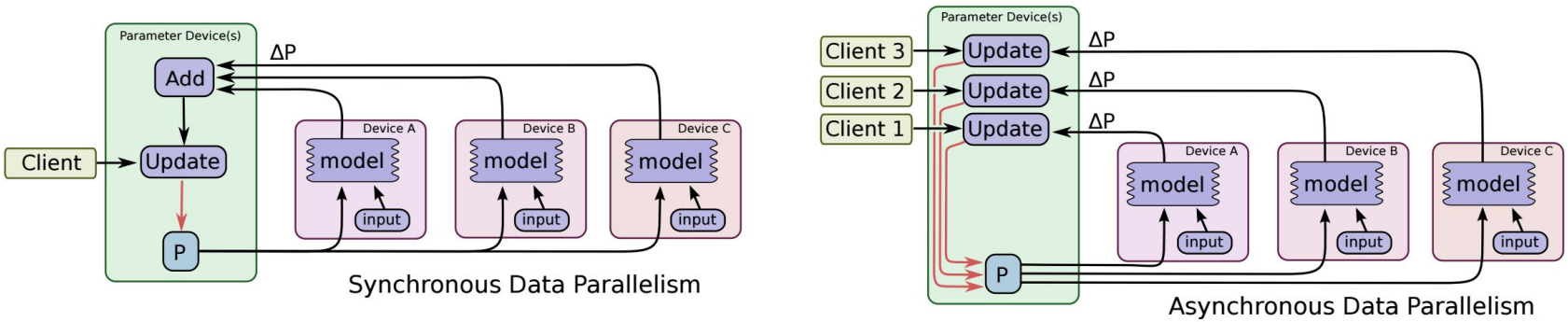
Data parallelism



Model parallelism

[Large Scale Distributed Deep Networks, Jeff Dean et al., 2013]

Google: Synchronous vs Async



Abadi et al, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems"

Bottlenecks

to be aware of

GPU - CPU communication is a bottleneck.
=>

CPU data prefetch+augment thread running

while

GPU performs forward/backward pass

CPU - disk bottleneck

Hard disk is slow to read from

Moving parts lol

=> Pre-processed images
stored contiguously in files, read as
raw byte stream from SSD disk

GPU memory bottleneck

Titan X: 12 GB <- currently the max
GTX 980 Ti: 6 GB

e.g.
AlexNet: ~3GB needed with batch size 256

Floating point precision

- 64 bit “double” precision is default in a lot of programming
- 32 bit “single” precision is typically used for CNNs for performance

Floating point precision

- 64 bit “double” precision is default in a lot of programming
- 32 bit “single” precision is typically used for CNNs for performance
  - Including cs231n homework!

```
class FullyConnectedNet(object):
  """
  A fully-connected neural network with an arbitrary number of hidden layers,
  ReLU nonlinearities, and a softmax loss function. This will also implement
  dropout and batch normalization as options. For a network with L layers,
  the architecture will be

  {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

  where batch normalization and dropout are optional, and the {...} block is
  repeated L - 1 times.

  Similar to the TwoLayerNet above, learnable parameters are stored in the
  self.params dictionary and will be learned using the Solver class.
  """

  def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
               dropout=0, use_batchnorm=False, reg=0.0,
               weight_scale=1e-2, dtype=np.float32, seed=None):
```

Floating point precision

Benchmarks on Titan X, from https://github.com/soumith/convnet-benchmarks

Prediction: 16 bit "half" precision will be the new standard

- Already supported in cuDNN
- Nervana fp16 kernels are the fastest right now
- Hardware support in next-gen NVIDIA cards (Pascal)
- Not yet supported in torch =(

AlexNet (One Weird Trick paper) - Input 128x3x224x224

Library	Class	Time (ms)	forward (ms)	backward (ms)
Nervana-fp16	ConvLayer	**92**	**29**	**62**
CuDNN[R3]-fp16 (Torch)	cudnn.SpatialConvolution	96	30	66
CuDNN[R3]-fp32 (Torch)	cudnn.SpatialConvolution	96	32	64

OxfordNet [Model-A] - Input 64x3x224x224

Library	Class	Time (ms)	forward (ms)	backward (ms)
Nervana-fp16	ConvLayer	**529**	**167**	**362**
Nervana-fp32	ConvLayer	590	180	410
CuDNN[R3]-fp16 (Torch)	cudnn.SpatialConvolution	615	179	436

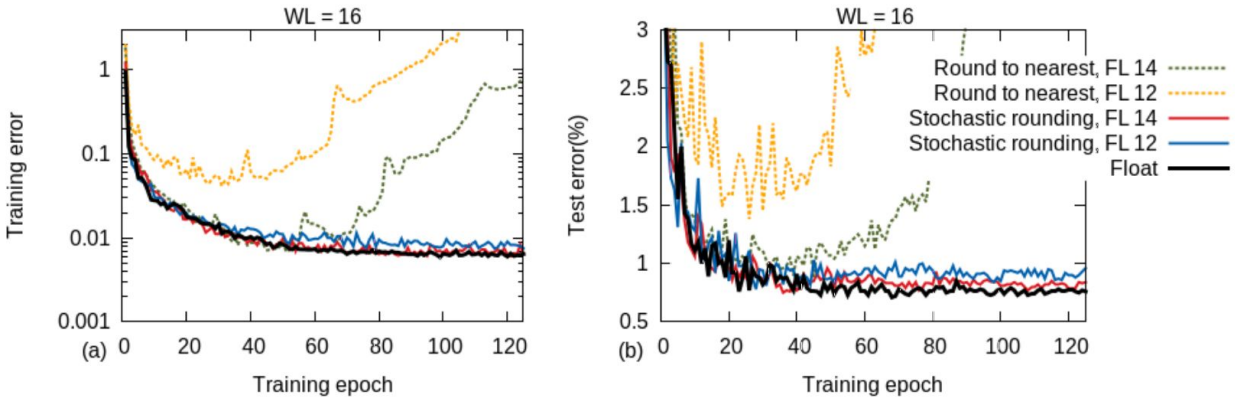
GoogleNet V1 - Input 128x3x224x224

Library	Class	Time (ms)	forward (ms)	backward (ms)
Nervana-fp16	ConvLayer	**283**	**85**	**197**
Nervana-fp32	ConvLayer	322	90	232
CuDNN[R3]-fp32 (Torch)	cudnn.SpatialConvolution	431	117	313

Floating point precision

How low can we go?

Gupta et al, 2015:
Train with **16-bit fixed point** with stochastic rounding



CNNs on MNIST

Gupta et al, "Deep Learning with Limited Numerical Precision", ICML 2015

Floating point precision

How low can we go?

Courbariaux et al, 2015:
Train with **10-bit activations**, **12-bit parameter updates**

Courbariaux et al, “Training Deep Neural Networks with Low Precision Multiplications”, ICLR 2015

Floating point precision

How low can we go?

Courbariaux and Bengio, February 9 2016:

- Train with **1-bit activations and weights**!
- All activations and weights are +1 or -1
- Fast multiplication with bitwise XNOR
- (Gradients use higher precision)

Courbariaux et al, "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1", arXiv 2016

Implementation details: Recap

- GPUs much faster than CPUs
- Distributed training is sometimes used
  - Not needed for small problems
- Be aware of bottlenecks: CPU / GPU, CPU / disk
- Low precison makes things faster and still works
  - 32 bit is standard now, 16 bit soon
  - In the future: binary nets?

Conclusions

- "Classic" CNN composed of conv layers, pooling layers, and fully connected layers
  - Date back to LeNet-5 by Yann Lecun in 90's
  - But gaining lots of attention since AlexNet 2012
- Some recent trends
  - ResNet is becoming default
  - Average pooling instead of fc layers
  - Small filter decomposition
  - Delay downsampling could help (see SqueezeNet)
  - Width vs depth tradeoff
  - Design to improve gradient flow (e.g., by skipping connection)

Conclusions

- "Classic" CNN composed of conv layers, pooling layers, and fully connected layers
  - Date back to LeNet-5 by Yann Lecun in 90's
  - But gaining lots of attention since AlexNet 2012
- Some recent trends
  - ResNet is becoming default
  - Average pooling instead of fc layers
  - Small filter decomposition
  - Delay downsampling could help (see SqueezeNet)
  - Width vs depth tradeoff
  - Design to improve gradient flow (e.g., by skipping connection)