

# Deep Learning Lesson 3

Lecturer: Paolo Russo

Dip. Ingegneria Informatica, Automatica e Gestionale, Roma

Credits: slides extracted from Stanford Fei Fei Li and Andrej Karpathy CS231n deep learning course



SAPIENZA  
UNIVERSITÀ DI ROMA

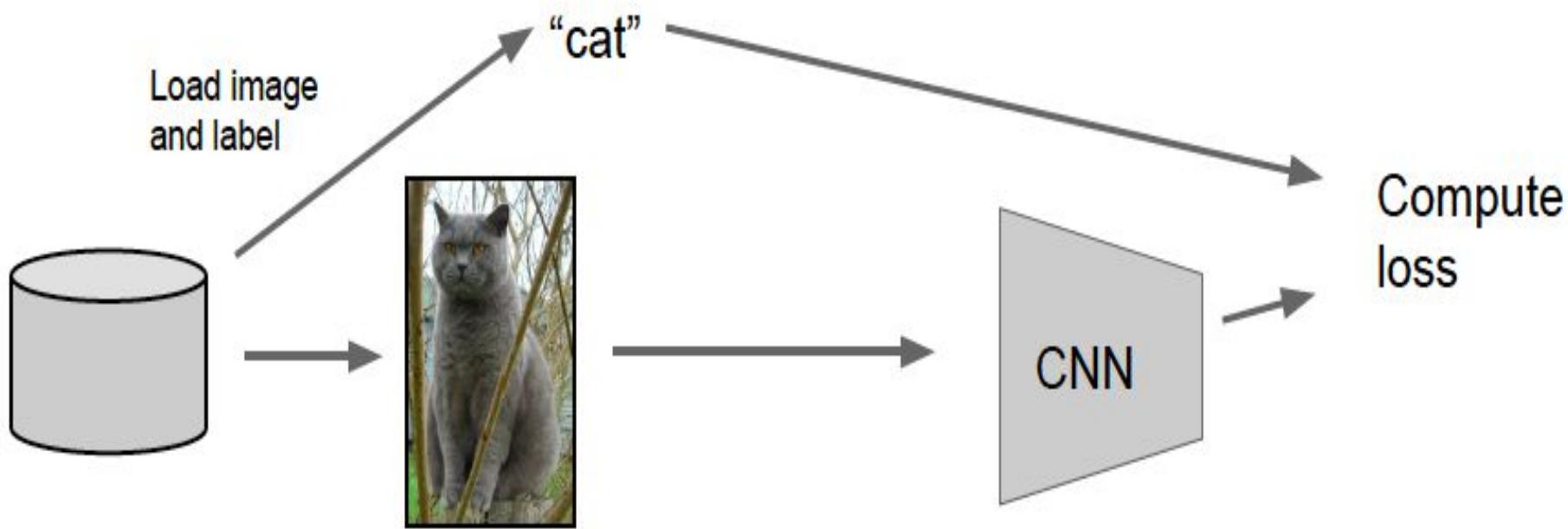
# Today

Working with CNNs in practice:

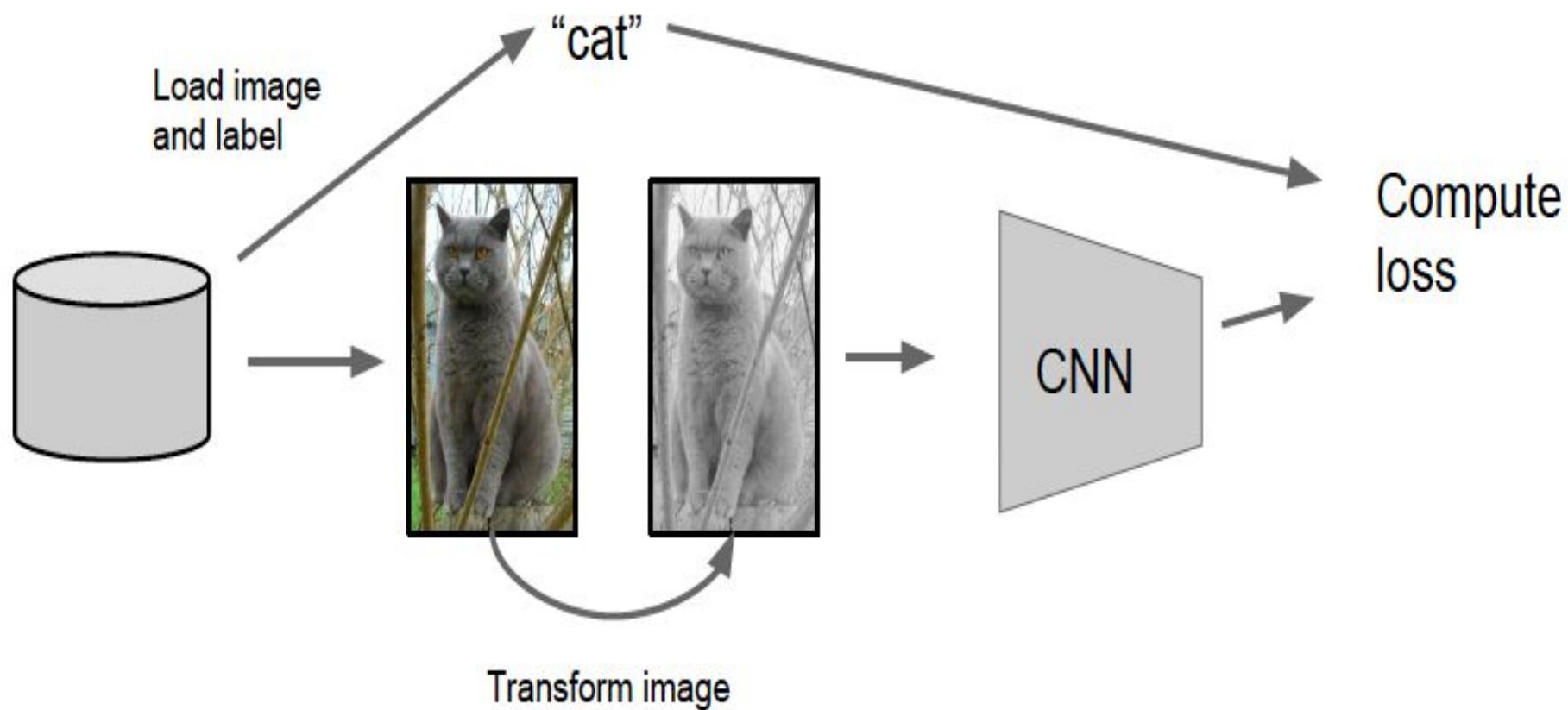
- Making the most of your data
  - Data augmentation
  - Transfer learning
- All about convolutions:
  - How to arrange them
  - How to compute them fast
- “Implementation details”
  - GPU / CPU, bottlenecks, distributed training

# Data Augmentation

# Data Augmentation

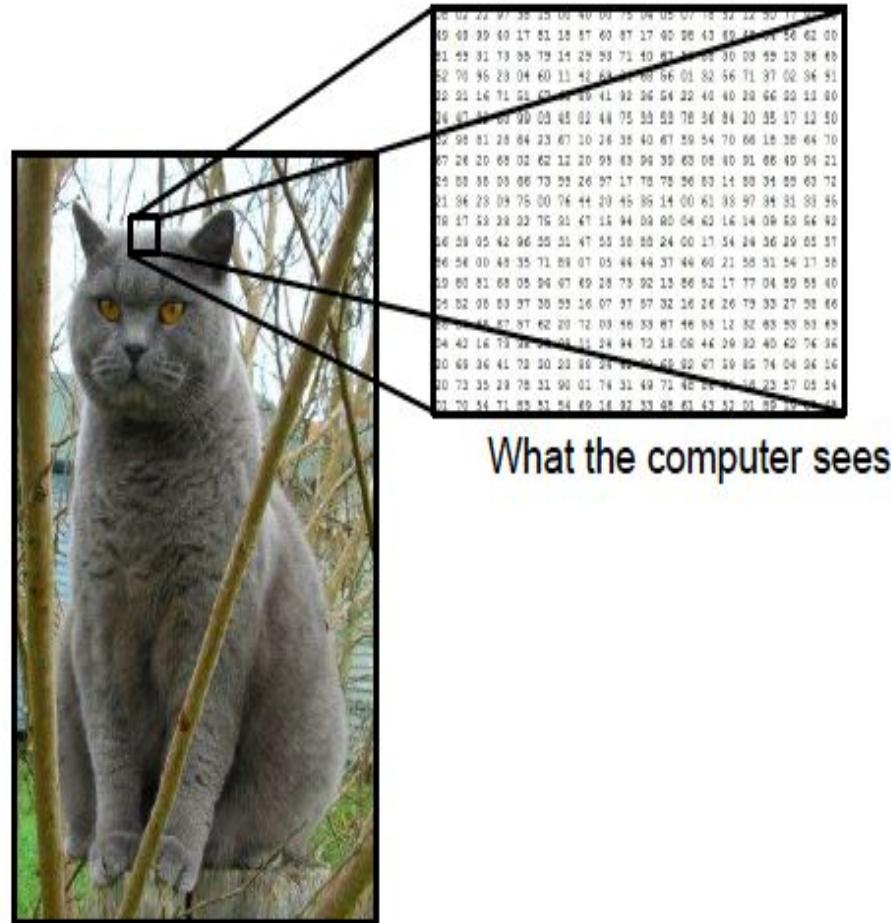


# Data Augmentation



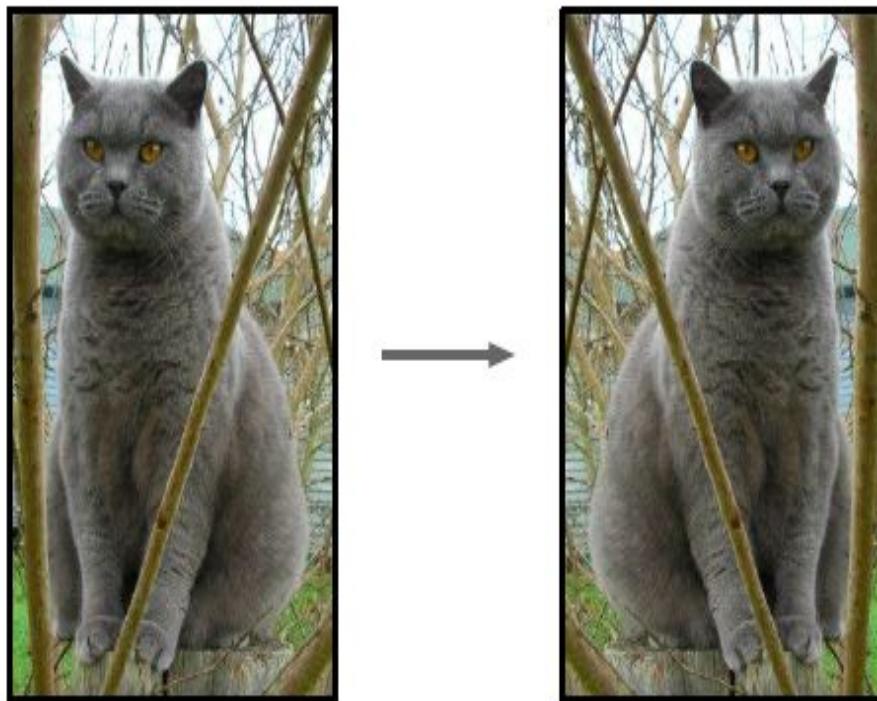
# Data Augmentation

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



# Data Augmentation

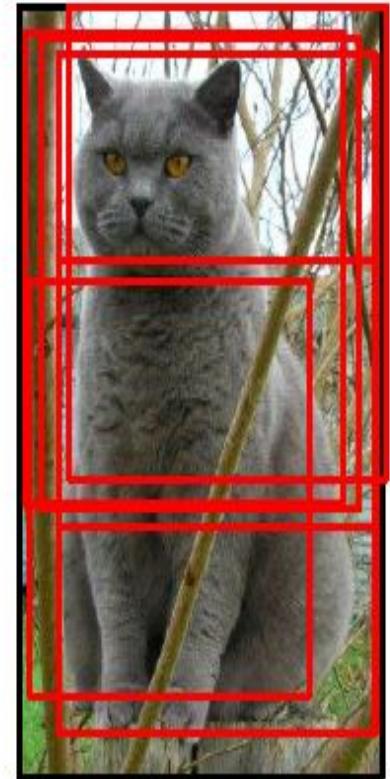
## 1. Horizontal flips



# Data Augmentation

## 2. Random crops/scales

Training: sample random crops / scales



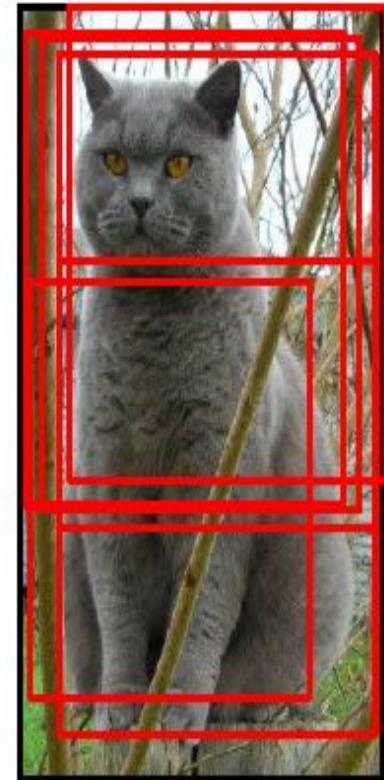
# Data Augmentation

## 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  $224 \times 224$  patch



# Data Augmentation

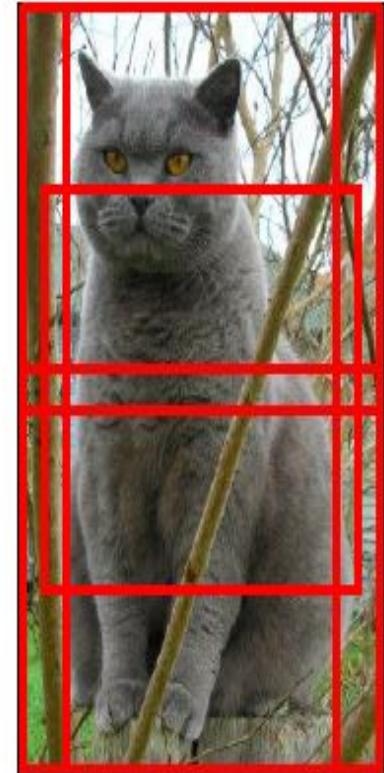
## 2. Random crops/scales

**Training:** sample random crops / scales

ResNet:

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

**Testing:** average a fixed set of crops



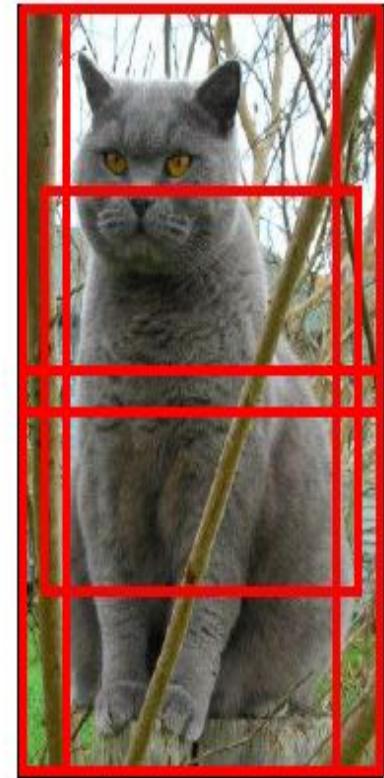
# Data Augmentation

## 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  $224 \times 224$  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  $224 \times 224$  crops: 4 corners + center, + flips

# Data Augmentation

## 3. Color jitter

**Simple:**

Randomly jitter contrast

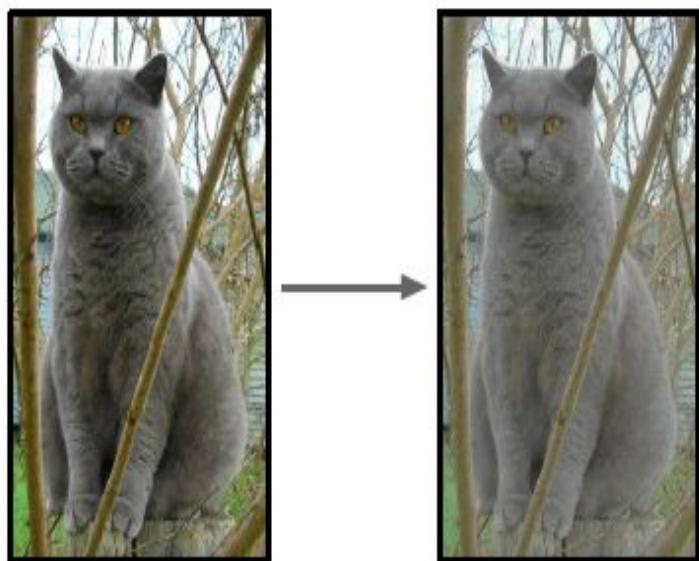


# Data Augmentation

## 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

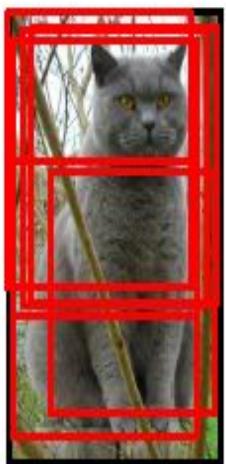
## 4. Get creative!

Random mix/combinations of :

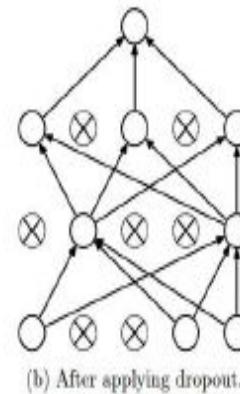
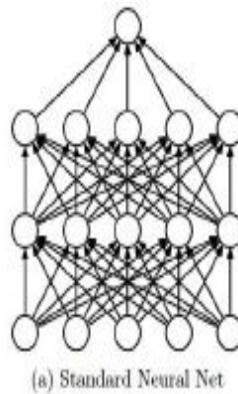
- translation
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)

# A general theme:

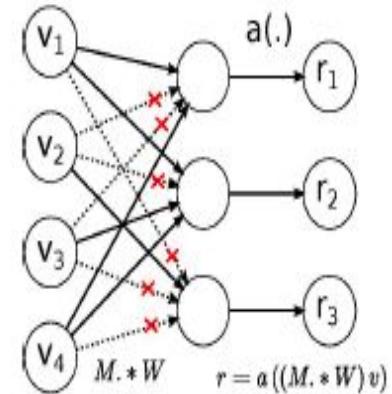
1. **Training:** Add random noise
2. **Testing:** Marginalize over the noise



Data Augmentation



Dropout



Batch normalization, Model ensembles

# Data Augmentation: Takeaway

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

# Transfer Learning

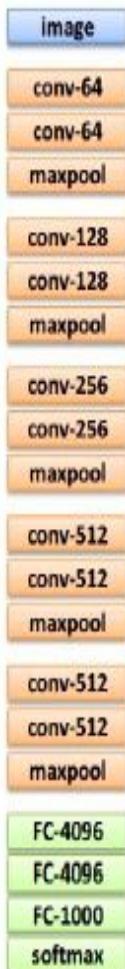
“You need a lot of data if you want to  
train/use CNNs”

# Transfer Learning

“You need a lot of data if you want to train/bo CNNs”

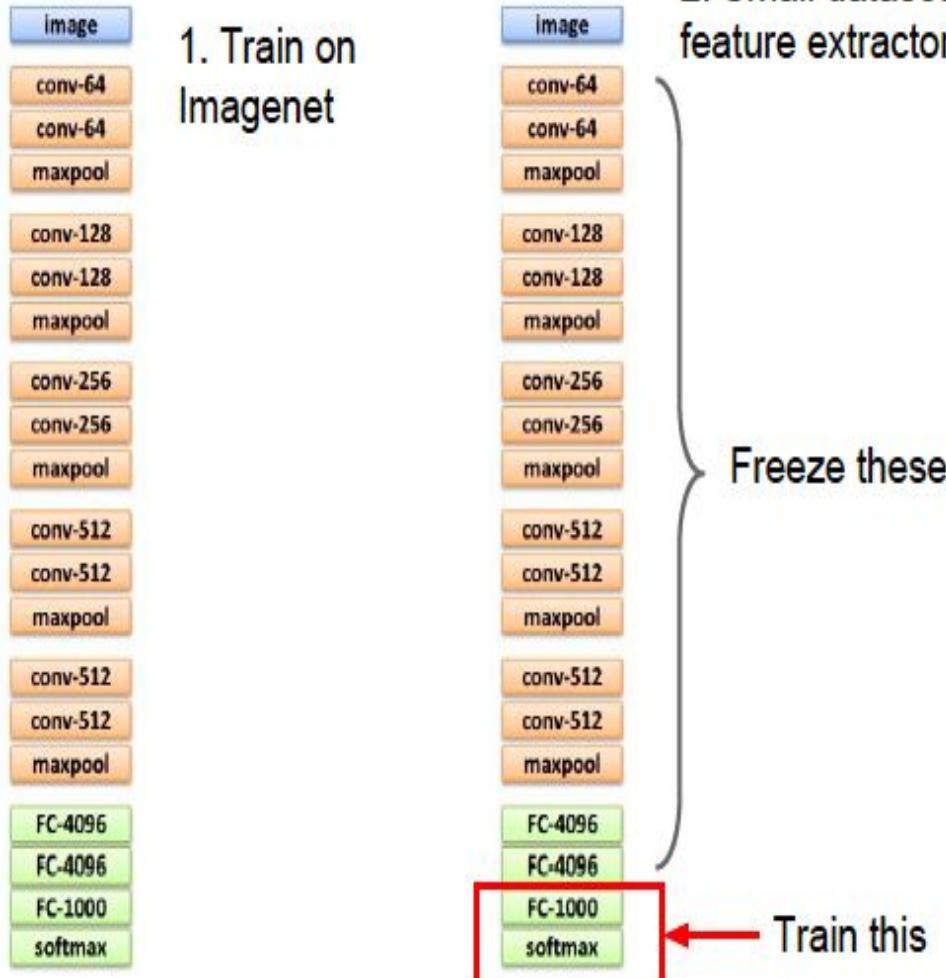
**BUSTED**

# Transfer Learning with CNNs

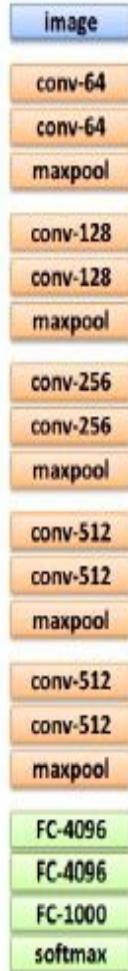


1. Train on  
Imagenet

# Transfer Learning with CNNs



# Transfer Learning with CNNs

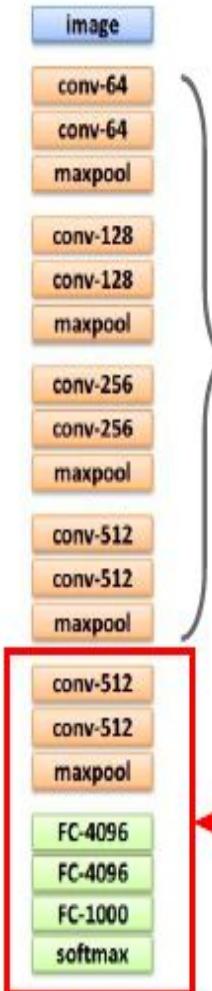


2. Small dataset:  
feature extractor



Freeze these

Train this



3. Medium dataset:  
finetuning

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

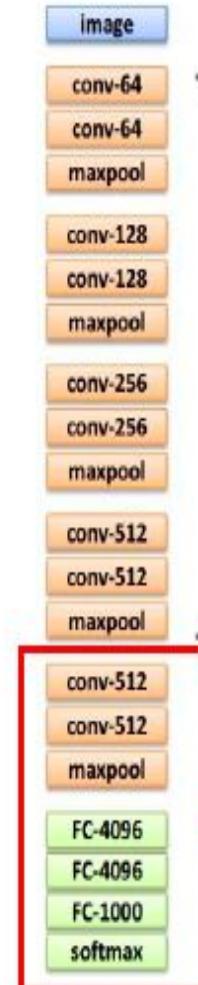
Freeze these

Train this

# Transfer Learning with CNNs



Freeze these



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

Freeze these

tip: use only ~1/10th of the original learning rate in finetuning top layer, and ~1/100th on intermediate layers

Train this

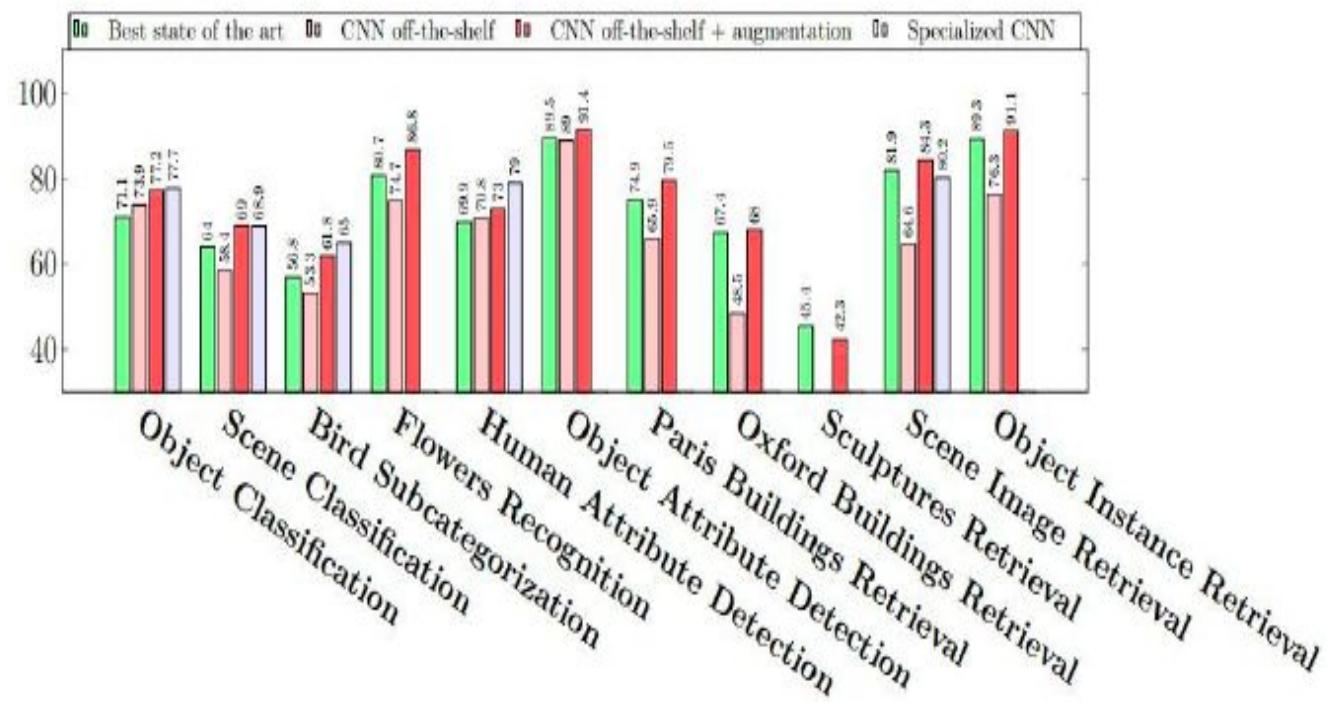
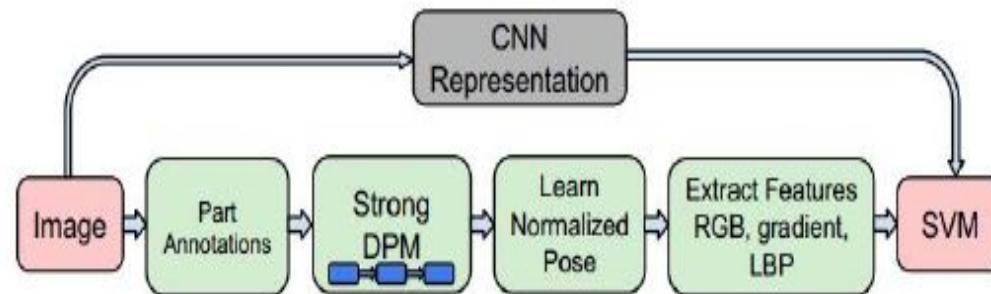
## CNN Features off-the-shelf: an Astounding Baseline for Recognition

[Razavian et al, 2014]

## DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition

[Donahue\*, Jia\*, et al., 2013]

	DeCAF <sub>6</sub>	DeCAF <sub>7</sub>
LogReg	40.94 ± 0.3	40.84 ± 0.3
SVM	39.36 ± 0.3	40.66 ± 0.3
Xiao et al. (2010)	38.0	



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

more generic

more specific

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

image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

more generic

more specific

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

image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

more generic

more specific

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

# Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

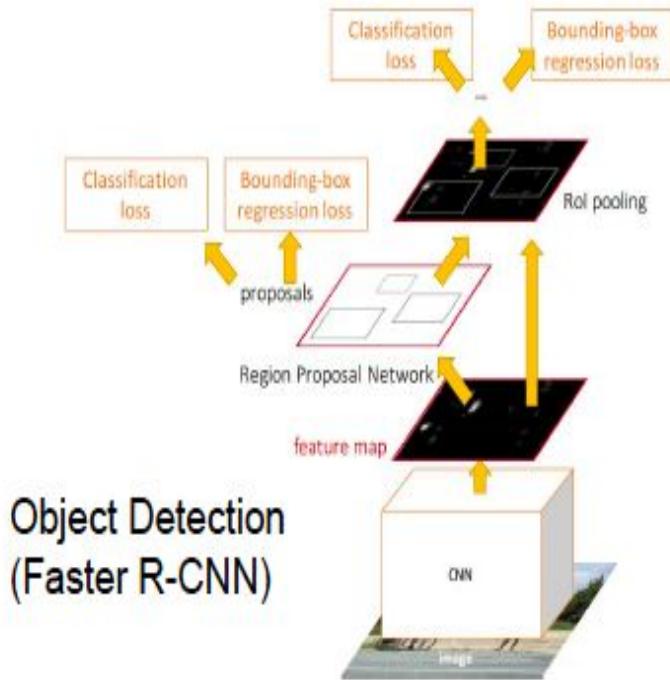
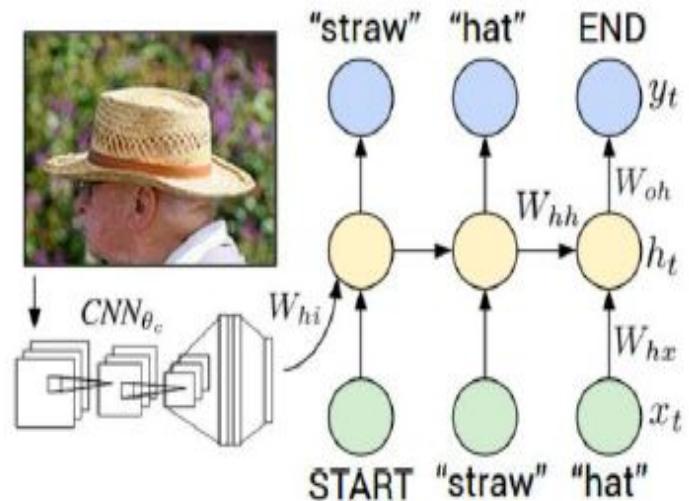
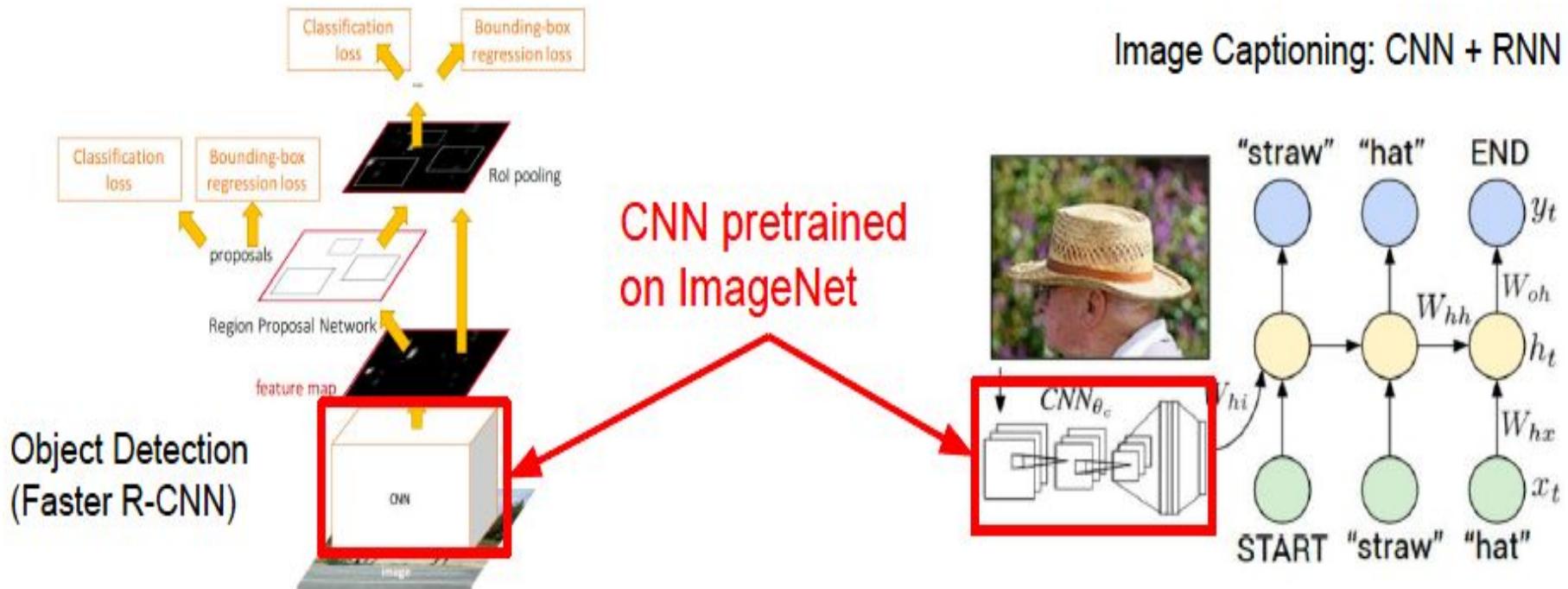


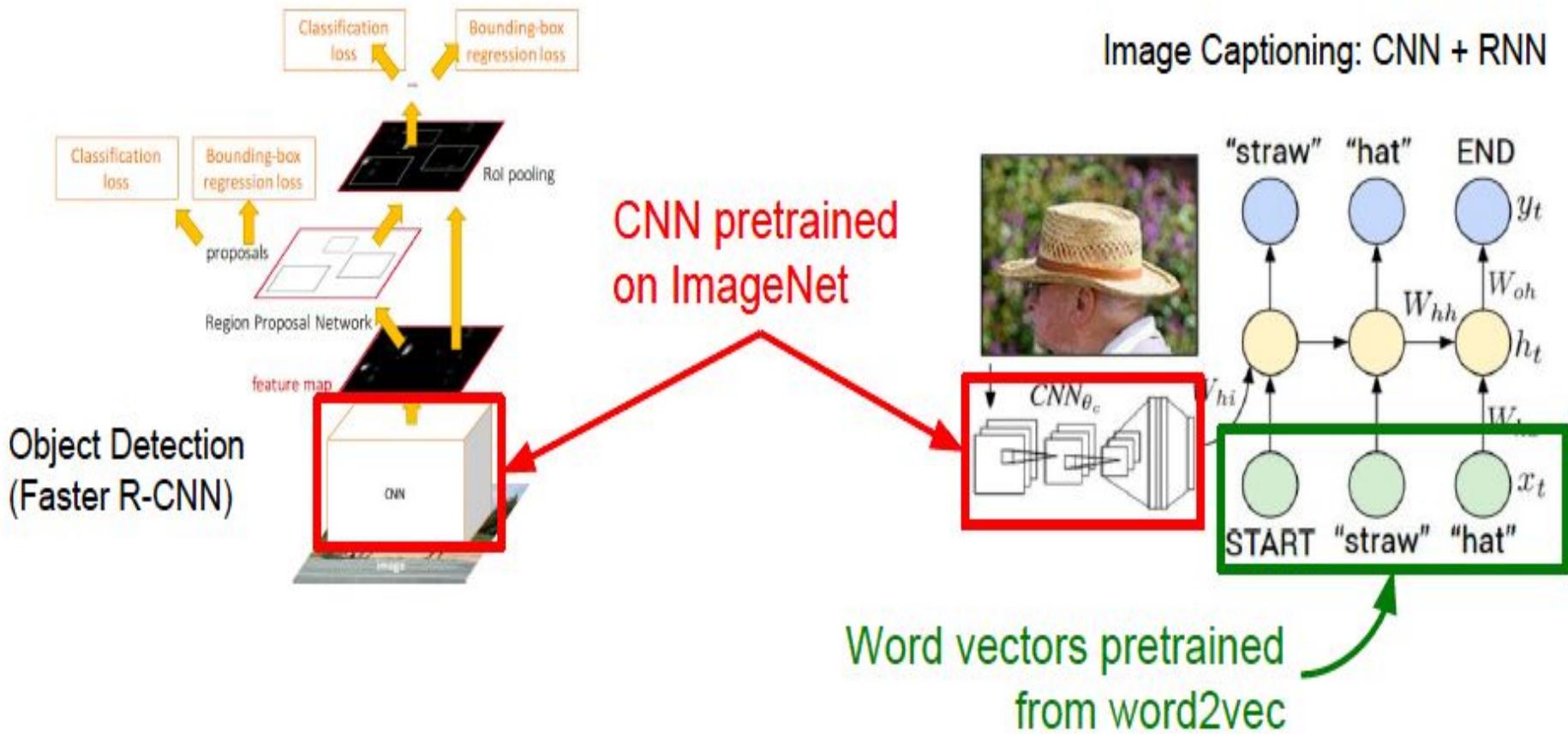
Image Captioning: CNN + RNN



# Transfer learning with CNNs is pervasive... (it's the norm, not an exception)



# Transfer learning with CNNs is pervasive... (it's the norm, not an exception)



## **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>

# All About Convolutions

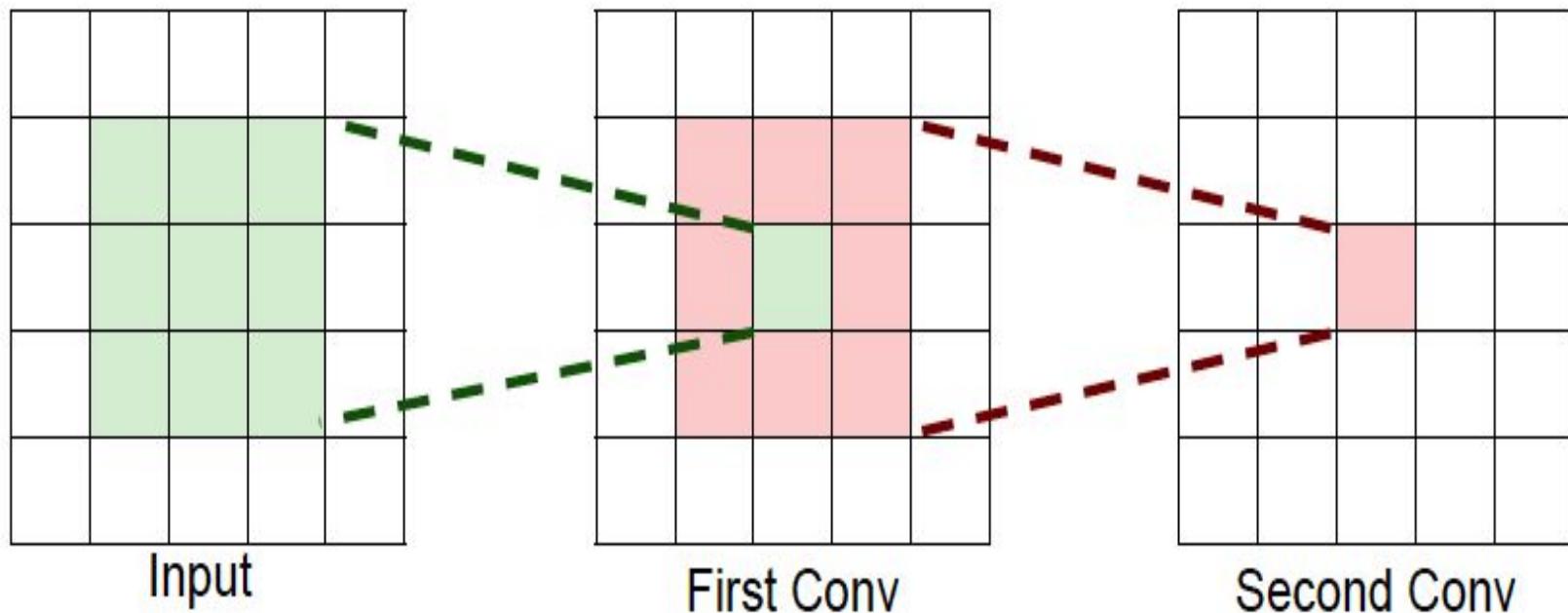
# All About Convolutions

## Part I: How to stack them

# The power of small filters

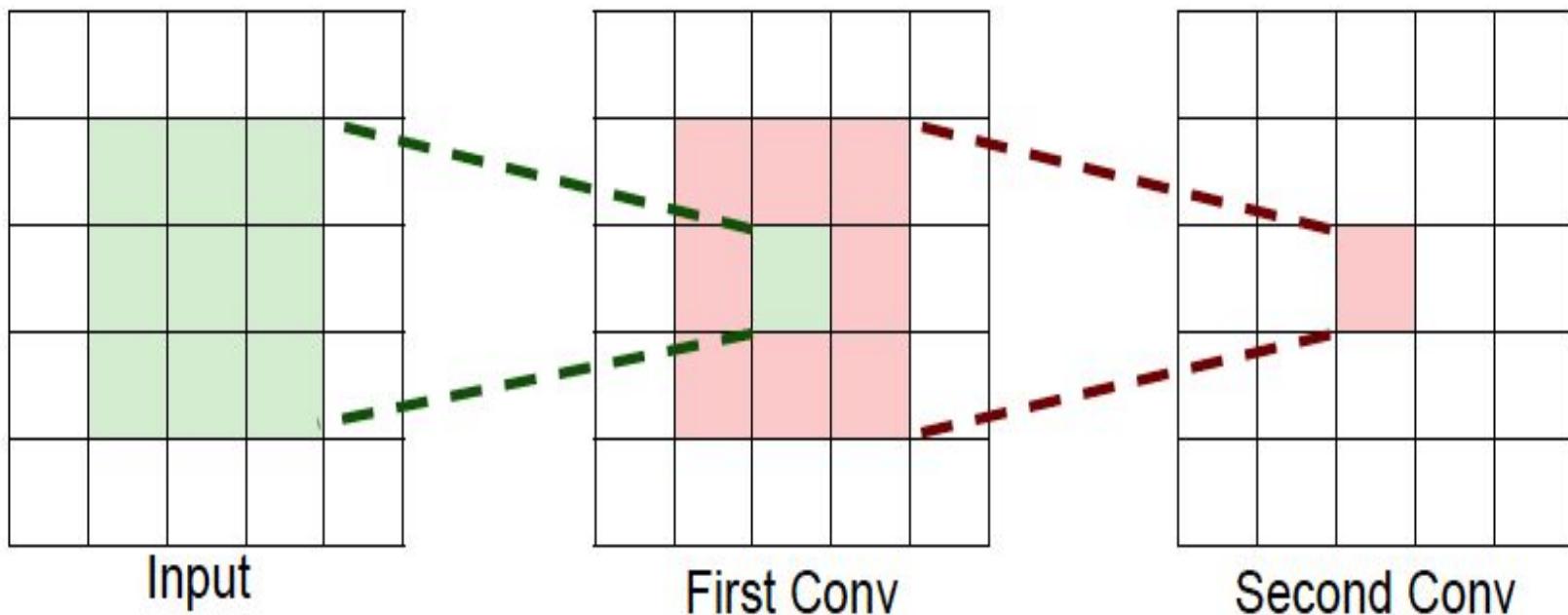
Suppose we stack two  $3 \times 3$  conv layers (stride 1)

Each neuron sees  $3 \times 3$  region of previous activation map



# The power of small filters

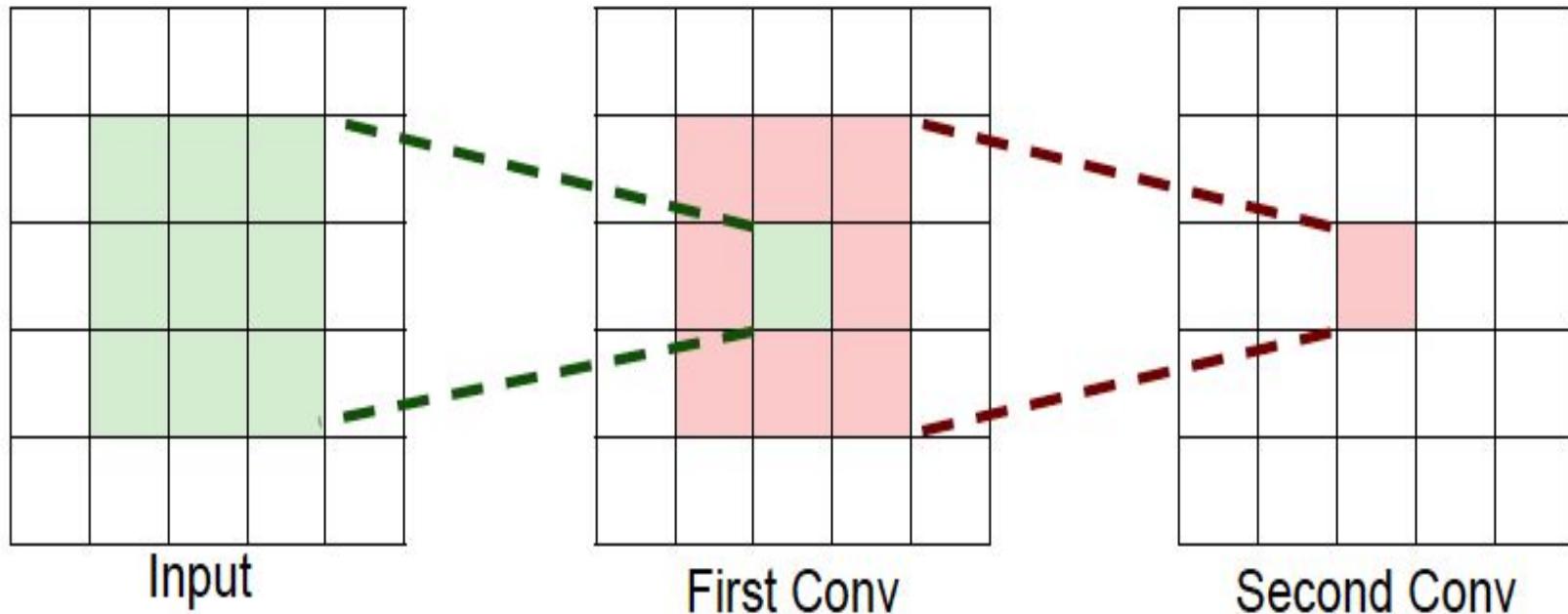
**Question:** How big of a region in the input does a neuron on the second conv layer see?



# The power of small filters

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

**Answer:**  $5 \times 5$

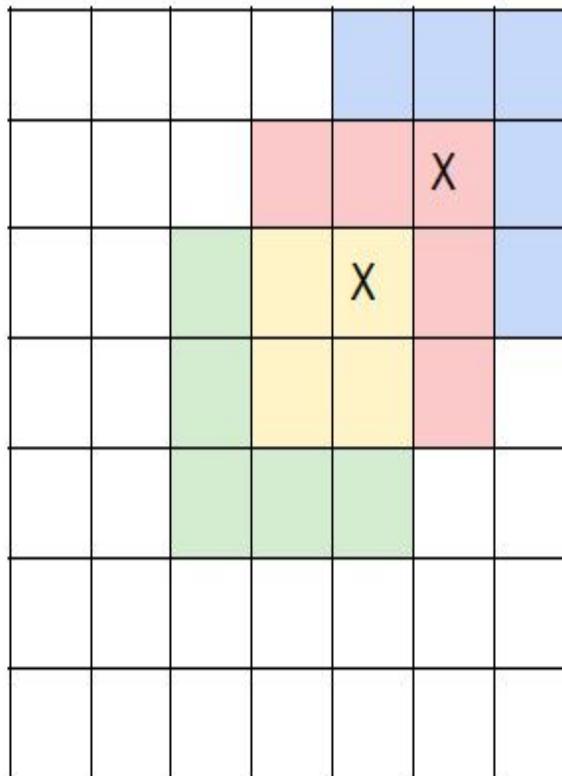


# The power of small filters

**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?

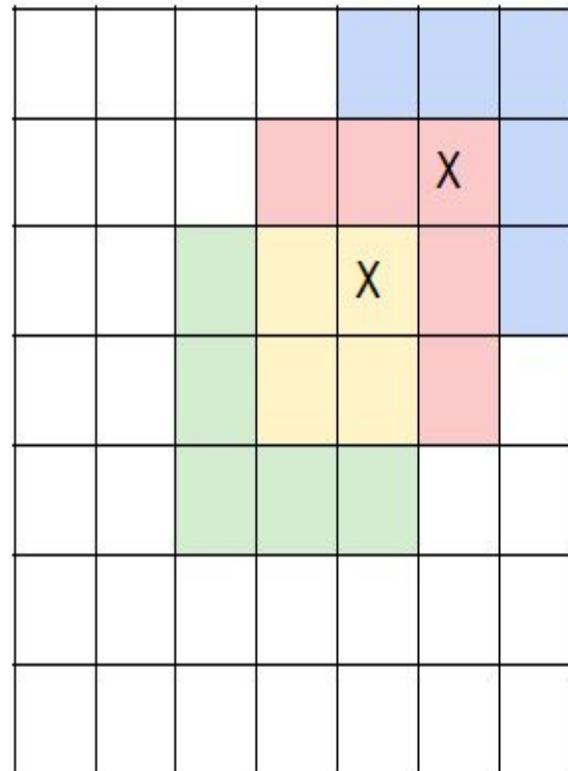


**Answer:** 7 x 7

# 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  $H \times W \times C$  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  $H \times W \times C$  and we use convolutions with  $C$  filters to preserve depth (stride 1, padding to preserve  $H, W$ )

one CONV with  $7 \times 7$  filters

Number of weights:

three CONV with  $3 \times 3$  filters

Number of weights:

# The power of small filters

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

one CONV with  $7 \times 7$  filters

Number of weights:

$$= C \times (7 \times 7 \times C) = 49 C^2$$

three CONV with  $3 \times 3$  filters

Number of weights:

$$= 3 \times C \times (3 \times 3 \times C) = 27 C^2$$

# The power of small filters

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

one CONV with  $7 \times 7$  filters

Number of weights:

$$= C \times (7 \times 7 \times C) = 49 C^2$$

three CONV with  $3 \times 3$  filters

Number of weights:

$$= 3 \times C \times (3 \times 3 \times C) = 27 C^2$$



Fewer parameters, more nonlinearity = GOOD

# The power of small filters

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

one CONV with  $7 \times 7$  filters

Number of weights:

$$= C \times (7 \times 7 \times C) = 49 C^2$$

Number of multiply-adds:

three CONV with  $3 \times 3$  filters

Number of weights:

$$= 3 \times C \times (3 \times 3 \times C) = 27 C^2$$

Number of multiply-adds:

# The power of small filters

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

one CONV with  $7 \times 7$  filters

Number of weights:

$$= C \times (7 \times 7 \times C) = 49 C^2$$

Number of multiply-adds:

$$\begin{aligned} &= (H \times W \times C) \times (7 \times 7 \times C) \\ &= \mathbf{49 HWC^2} \end{aligned}$$

three CONV with  $3 \times 3$  filters

Number of weights:

$$= 3 \times C \times (3 \times 3 \times C) = 27 C^2$$

Number of multiply-adds:

$$\begin{aligned} &= 3 \times (H \times W \times C) \times (3 \times 3 \times C) \\ &= \mathbf{27 HWC^2} \end{aligned}$$

# The power of small filters

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

one CONV with  $7 \times 7$  filters

Number of weights:

$$= C \times (7 \times 7 \times C) = 49 C^2$$

Number of multiply-adds:

$$= 49 HWC^2$$

three CONV with  $3 \times 3$  filters

Number of weights:

$$= 3 \times C \times (3 \times 3 \times C) = 27 C^2$$

Number of multiply-adds:

$$= 27 HWC^2$$

Less compute, more nonlinearity = GOOD

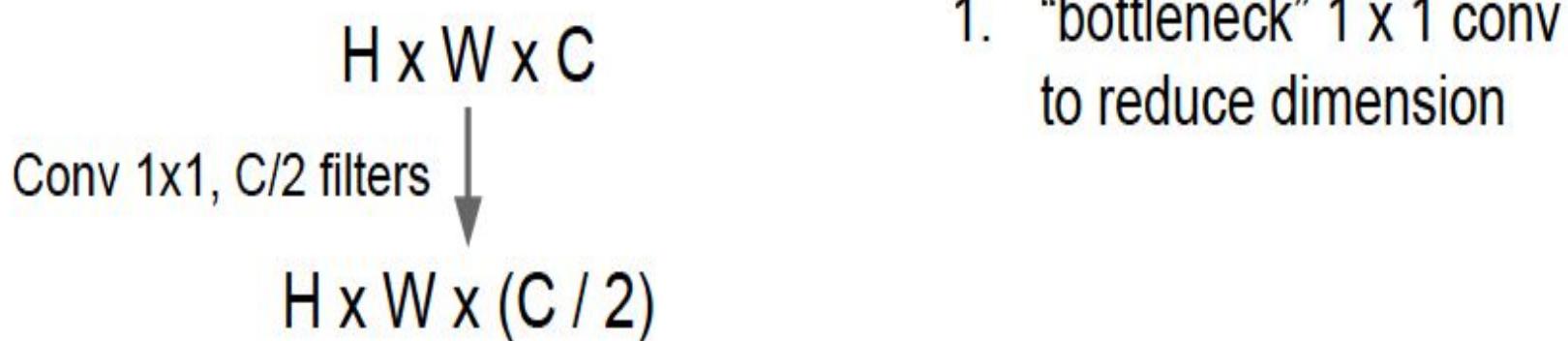
**5 minutes break**

# The power of small filters

Why stop at  $3 \times 3$  filters? Why not try  $1 \times 1$ ?

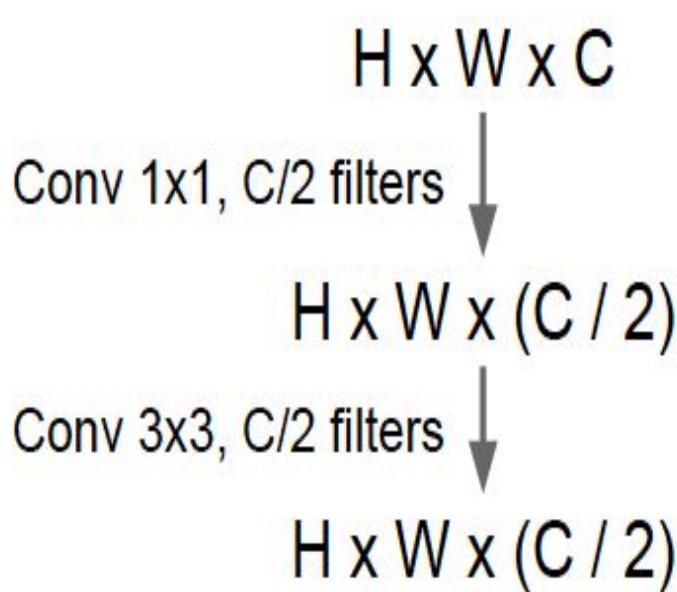
# The power of small filters

Why stop at  $3 \times 3$  filters? Why not try  $1 \times 1$ ?



# The power of small filters

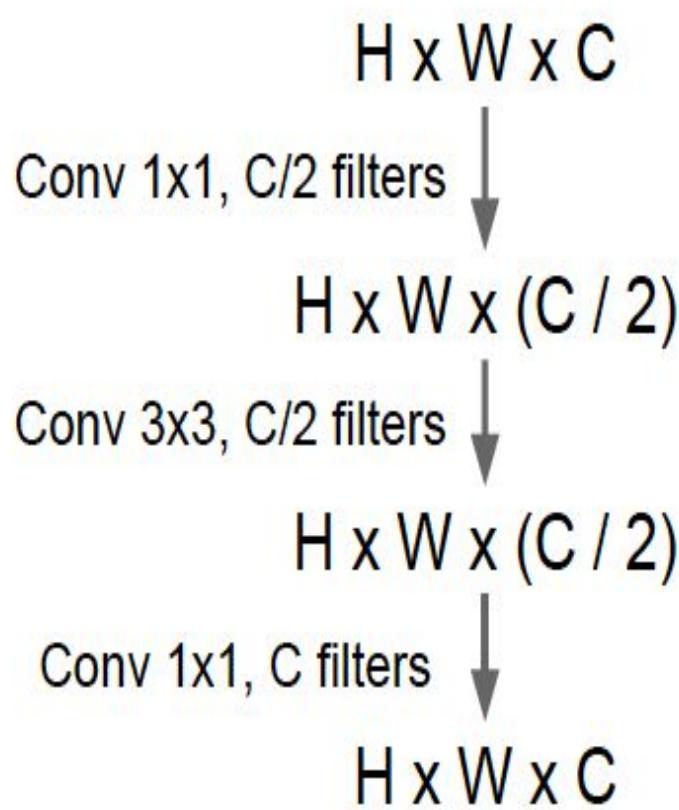
Why stop at  $3 \times 3$  filters? Why not try  $1 \times 1$ ?



1. “bottleneck”  $1 \times 1$  conv to reduce dimension
2.  $3 \times 3$  conv at reduced dimension

# The power of small filters

Why stop at  $3 \times 3$  filters? Why not try  $1 \times 1$ ?

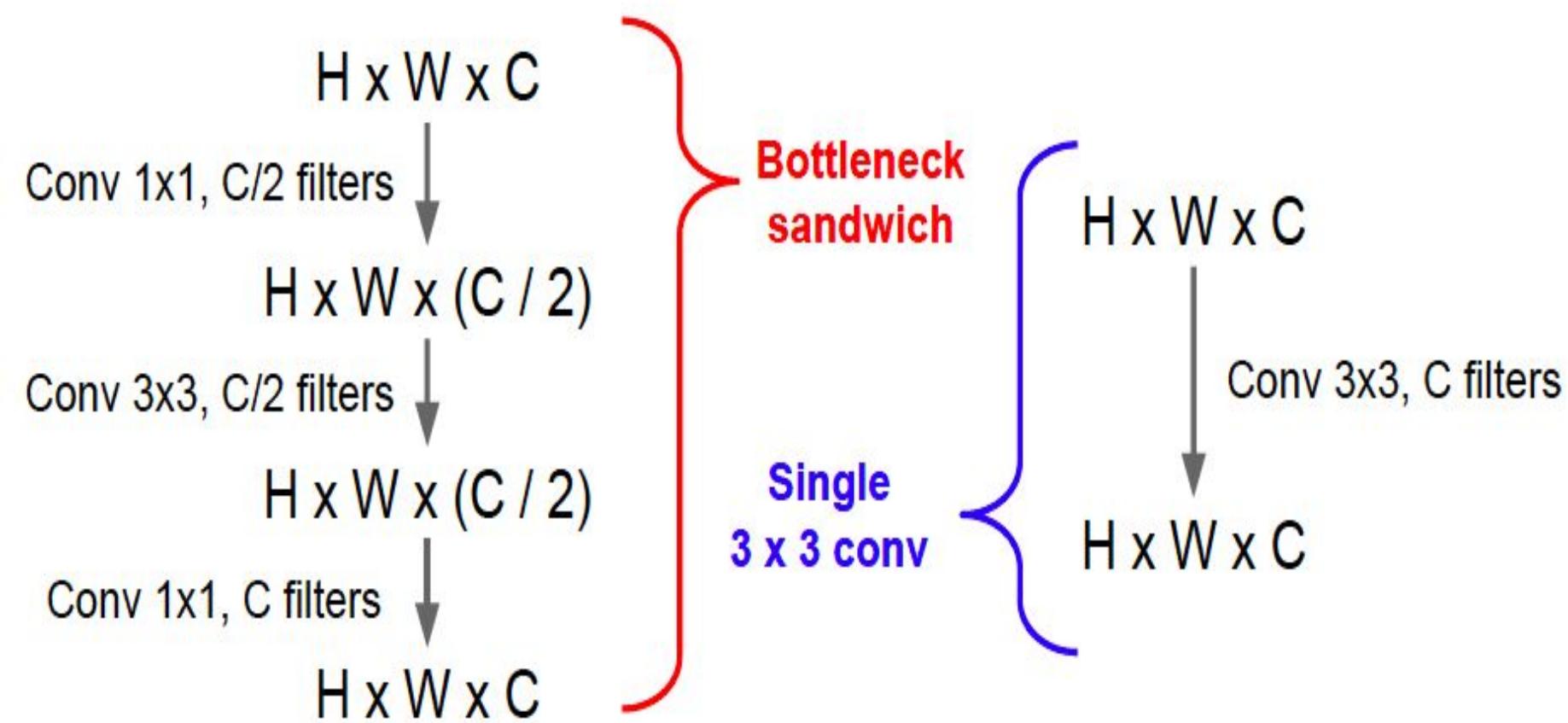


1. “bottleneck”  $1 \times 1$  conv to reduce dimension
2.  $3 \times 3$  conv at reduced dimension
3. Restore dimension with another  $1 \times 1$  conv

[Seen in Lin et al, “Network in Network”, GoogLeNet, ResNet]

# The power of small filters

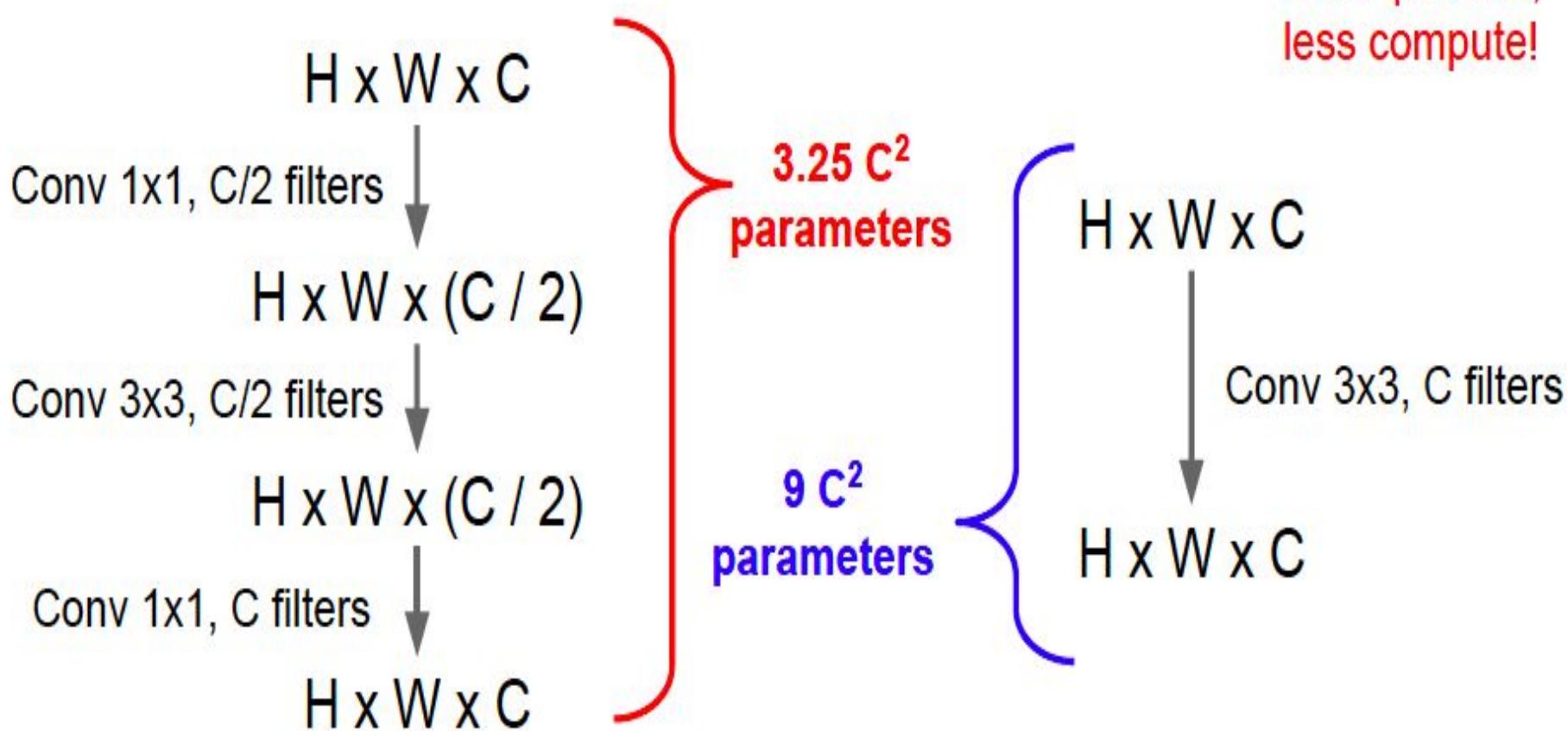
Why stop at  $3 \times 3$  filters? Why not try  $1 \times 1$ ?



# The power of small filters

Why stop at  $3 \times 3$  filters? Why not try  $1 \times 1$ ?

More nonlinearity,  
fewer params,  
less compute!

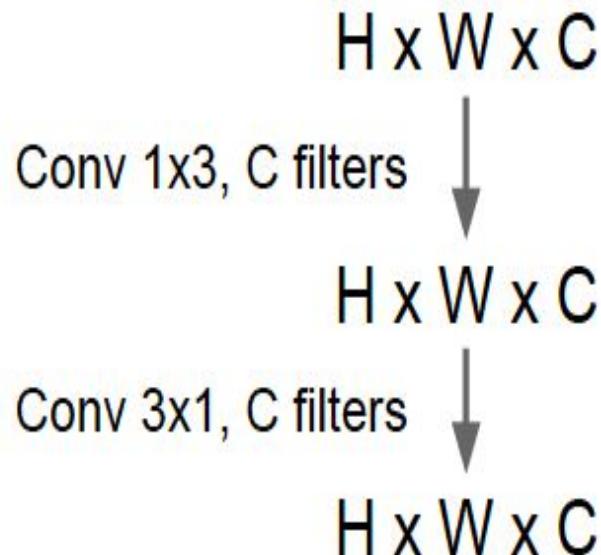


# The power of small filters

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

# The power of small filters

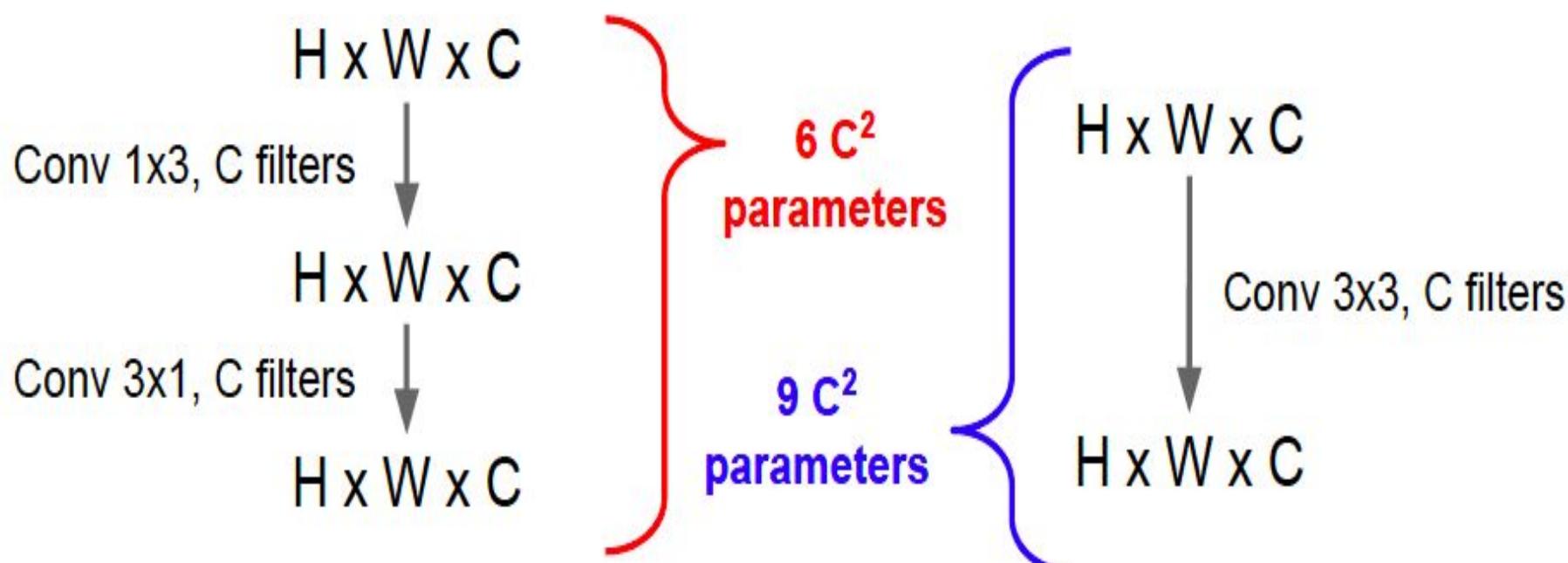
Still using 3 x 3 filters ... can we break it up?



# The power of small filters

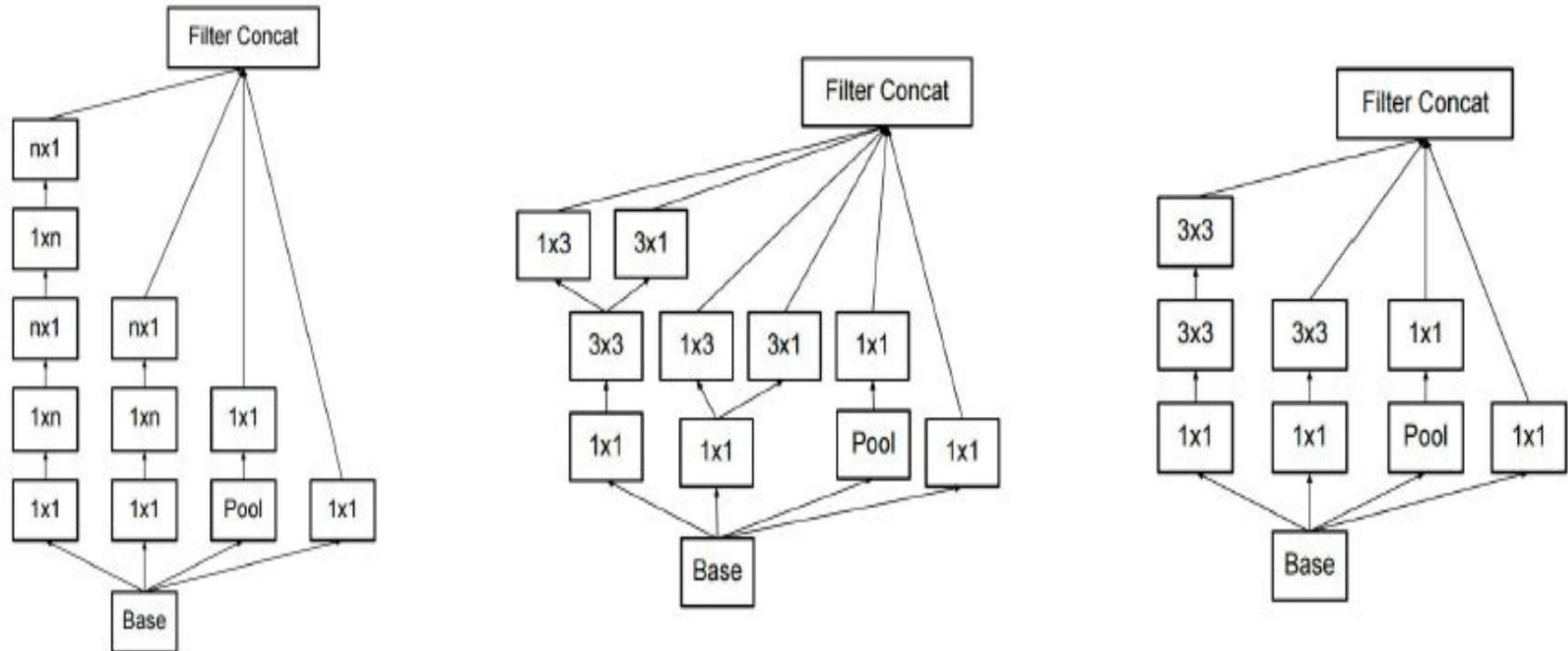
Still using 3 x 3 filters ... can we break it up?

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

# All About Convolutions

## Part II: How to compute them

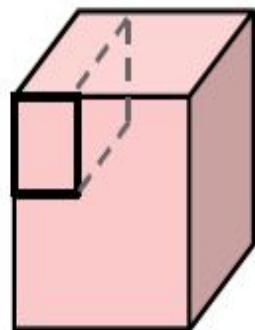
# Implementing Convolutions: im2col

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

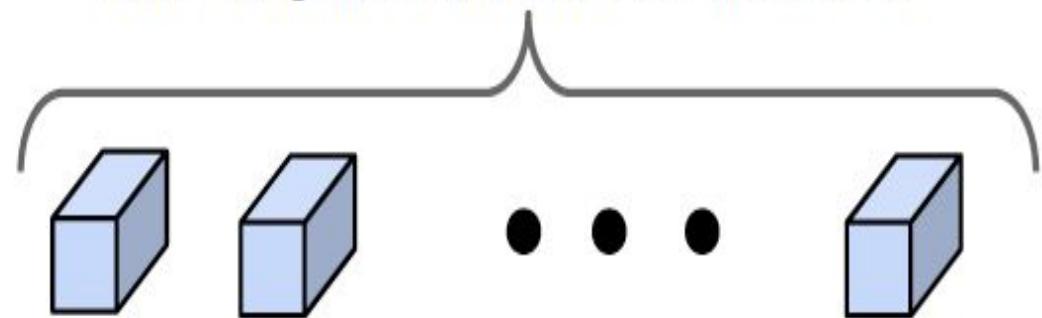
Can we turn convolution into matrix multiplication?

# Implementing Convolutions: im2col

Feature map:  $H \times W \times C$

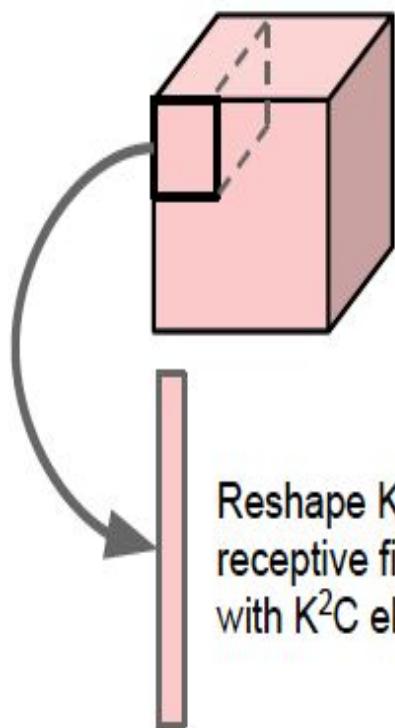


Conv weights: D filters, each  $K \times K \times C$

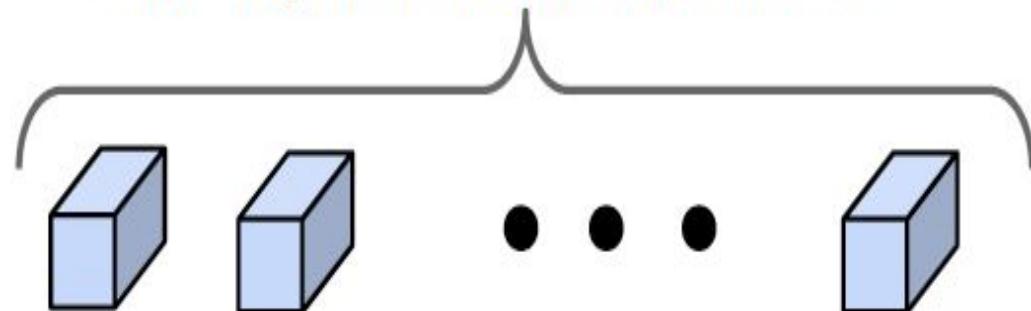


# Implementing Convolutions: im2col

Feature map:  $H \times W \times C$

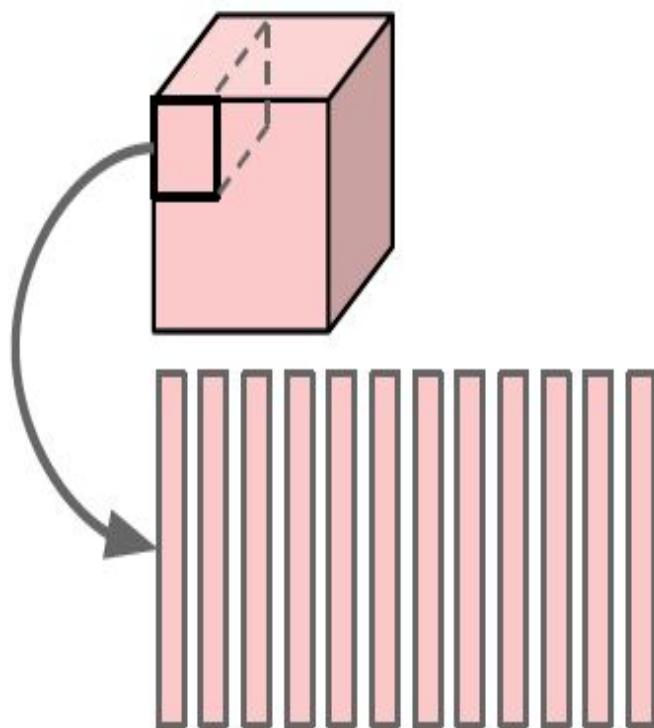


Conv weights: D filters, each  $K \times K \times C$

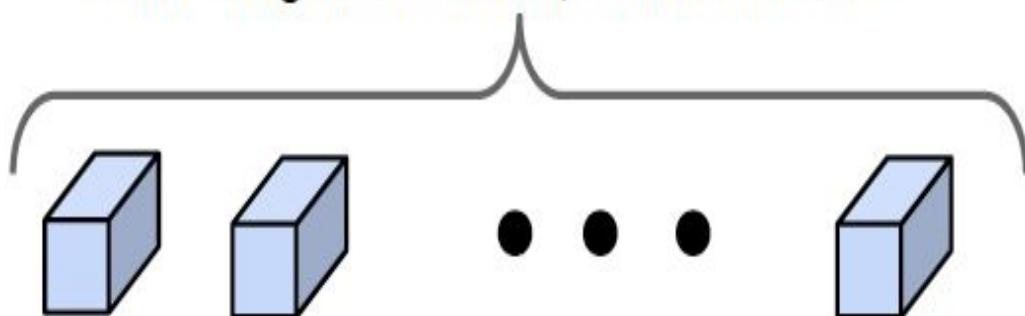


# Implementing Convolutions: im2col

Feature map:  $H \times W \times C$



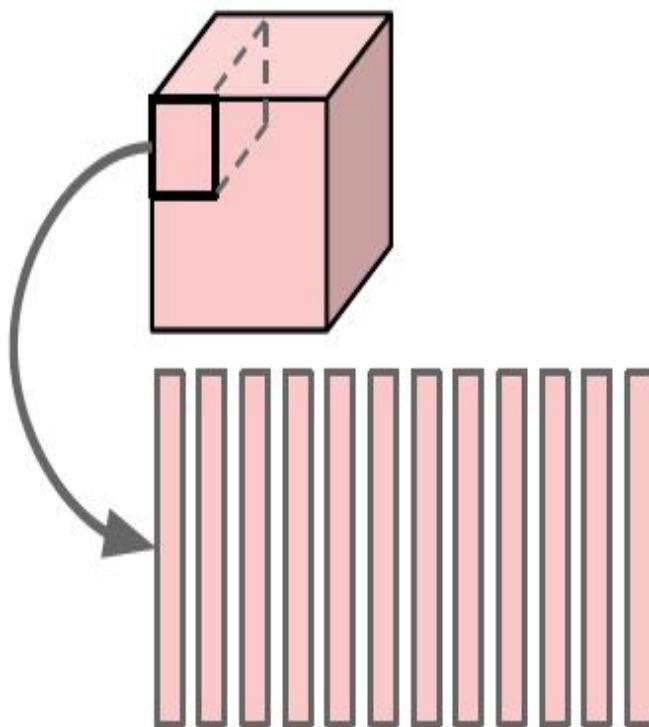
Conv weights: D filters, each  $K \times K \times C$



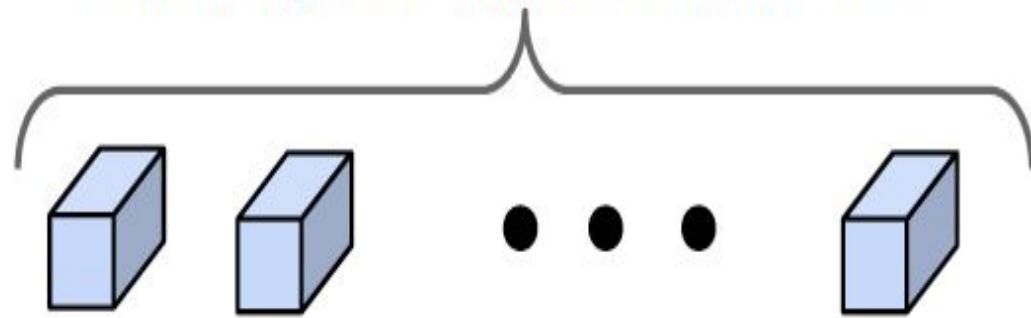
Repeat for all columns to get  $(K^2C) \times N$  matrix  
( $N$  receptive field locations)

# Implementing Convolutions: im2col

Feature map:  $H \times W \times C$



Conv weights: D filters, each  $K \times K \times C$

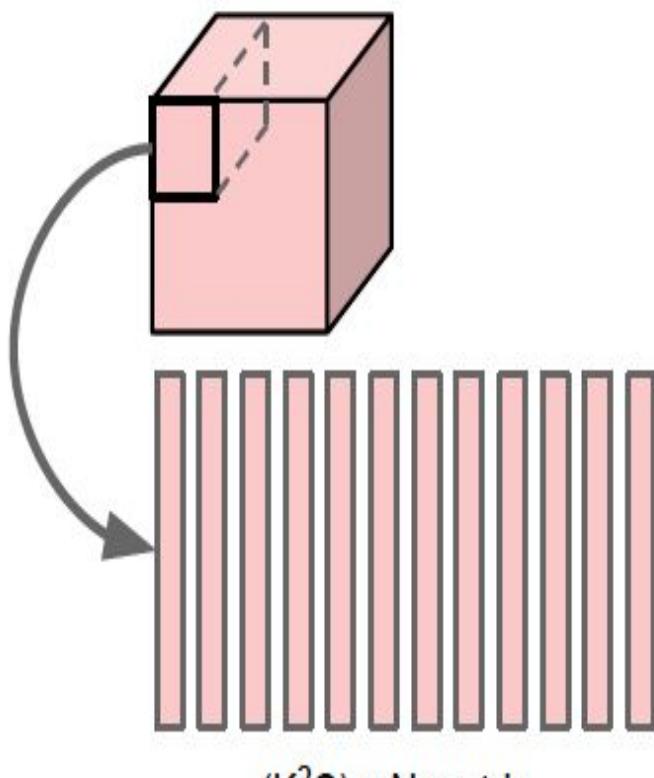


Elements appearing in multiple receptive fields are duplicated; this uses a lot of memory

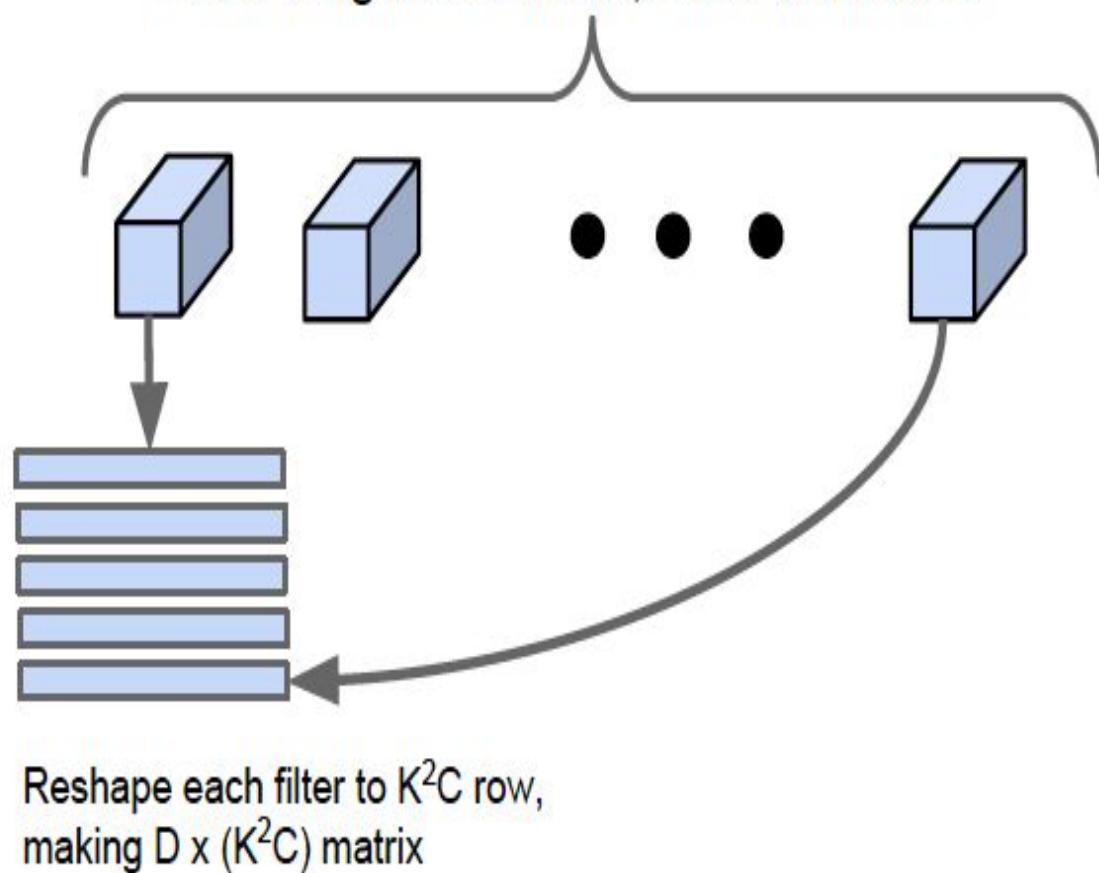
Repeat for all columns to get  $(K^2C) \times N$  matrix  
(N receptive field locations)

# Implementing Convolutions: im2col

Feature map:  $H \times W \times C$

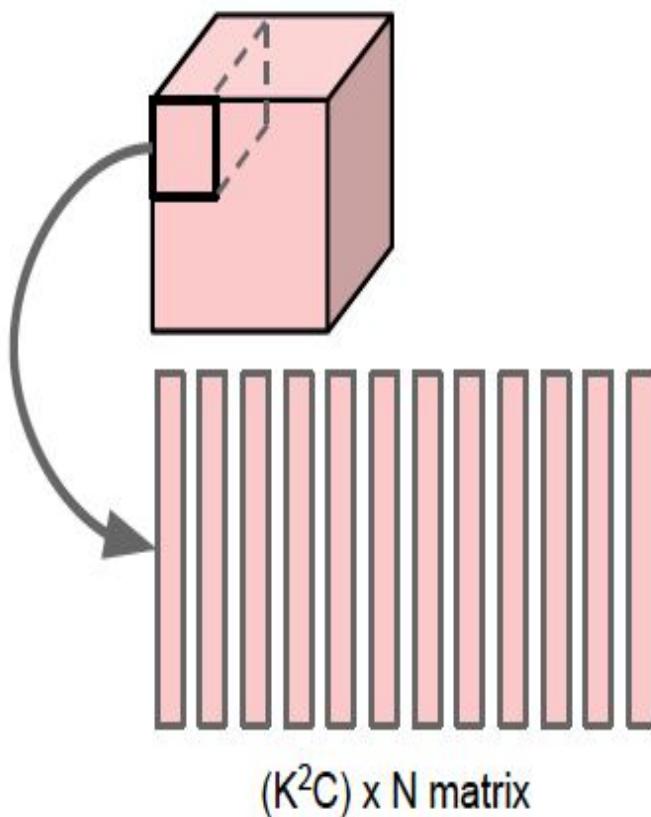


Conv weights: D filters, each  $K \times K \times C$

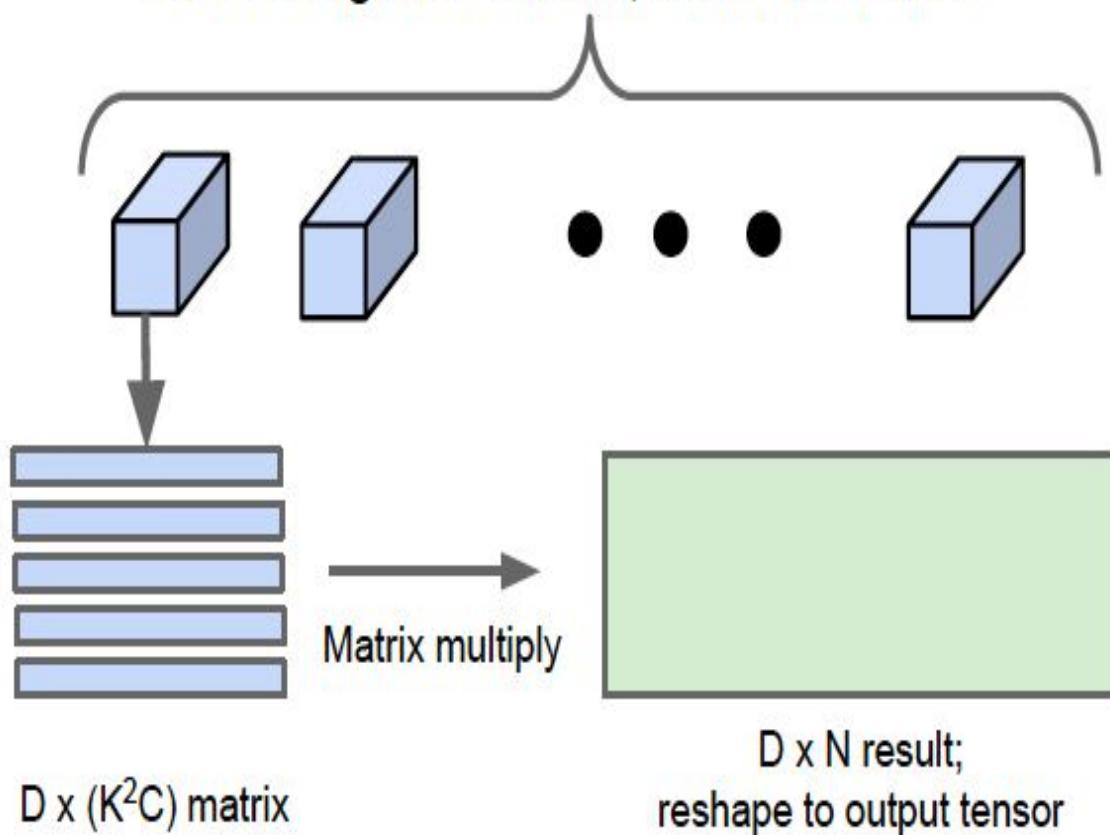


# Implementing Convolutions: im2col

Feature map:  $H \times W \times C$



Conv weights: D filters, each  $K \times K \times C$



# Implementing convolutions: FFT

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

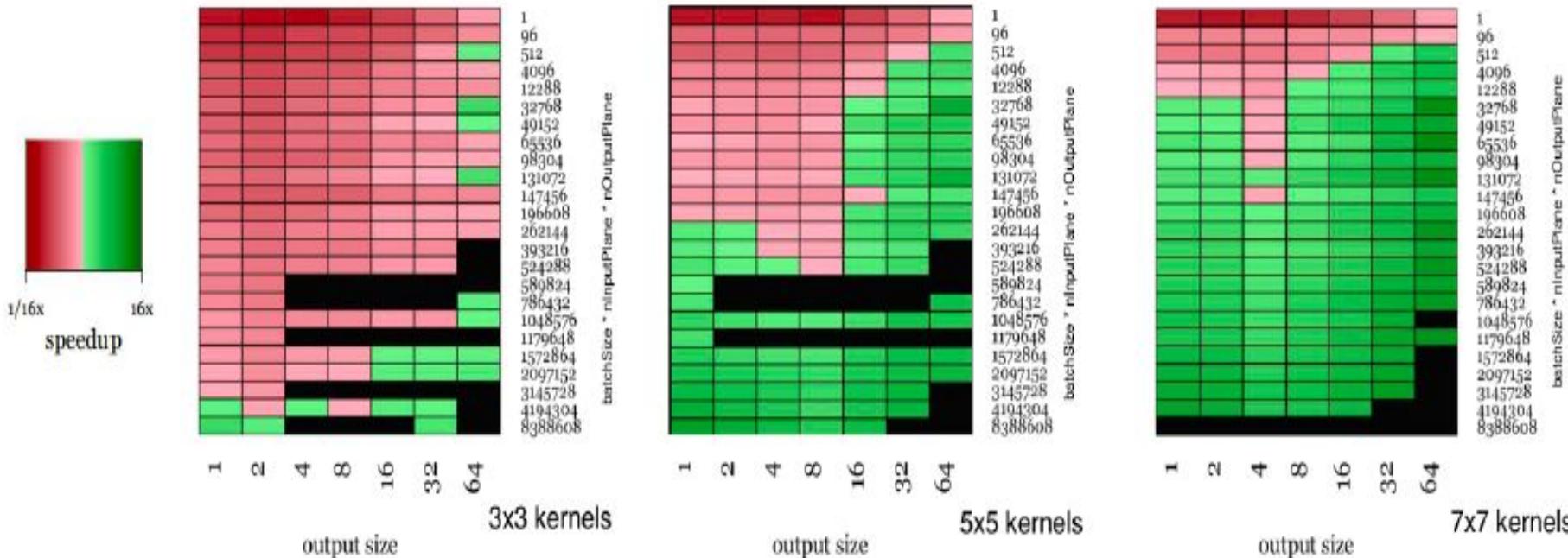
$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$$

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  $N \times N$  matrices takes  $O(N^3)$  operations

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

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

$$M_1 := (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$

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

$$M_2 := (A_{2,1} + A_{2,2})B_{1,1}$$

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

$$M_3 := A_{1,1}(B_{1,2} - B_{2,2})$$

$$M_4 := A_{2,2}(B_{2,1} - B_{1,1})$$

$$M_5 := (A_{1,1} + A_{1,2})B_{2,2}$$

$$M_6 := (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$

$$M_7 := (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$

$$C_{1,2} = M_3 + M_5$$

$$C_{2,1} = M_2 + M_4$$

$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

From Wikipedia

# Implementing convolution: “Fast Algorithms”

Similar cleverness can be applied to convolutions

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

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

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

$$\begin{aligned} B^T &= \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \\ G &= \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \\ A^T &= \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \\ g &= [g_0 \ g_1 \ g_2]^T \\ d &= [d_0 \ d_1 \ d_2 \ d_3]^T \end{aligned}$$

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

# Implementation Details



# Spot the CPU!



# Spot the CPU!

“central processing unit”



# Spot the GPU!

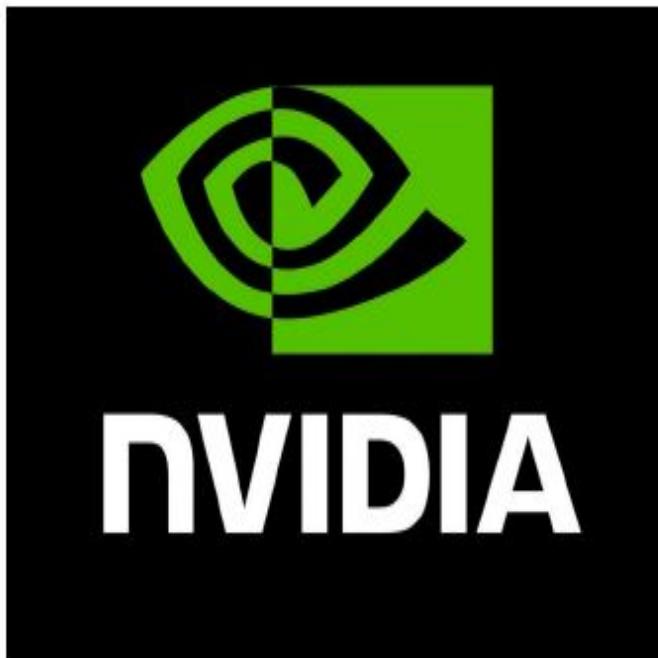
“graphics processing unit”



# Spot the GPU!

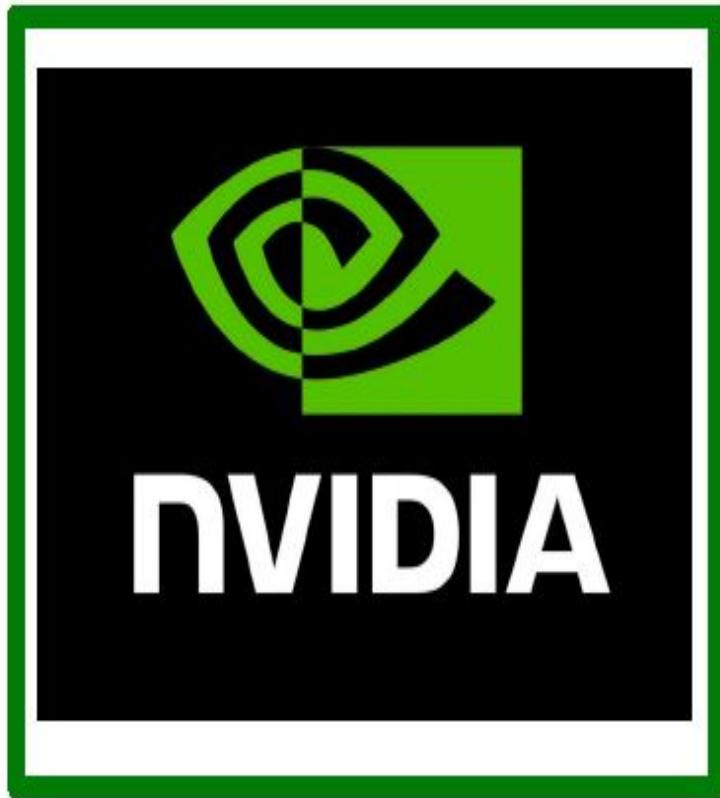
“graphics processing unit”





VS





VS



NVIDIA is much more  
common for deep learning

# CPU

Few, fast cores (1 - 16)

Good at sequential processing



# GPU

Many, slower cores (thousands)

Originally for graphics

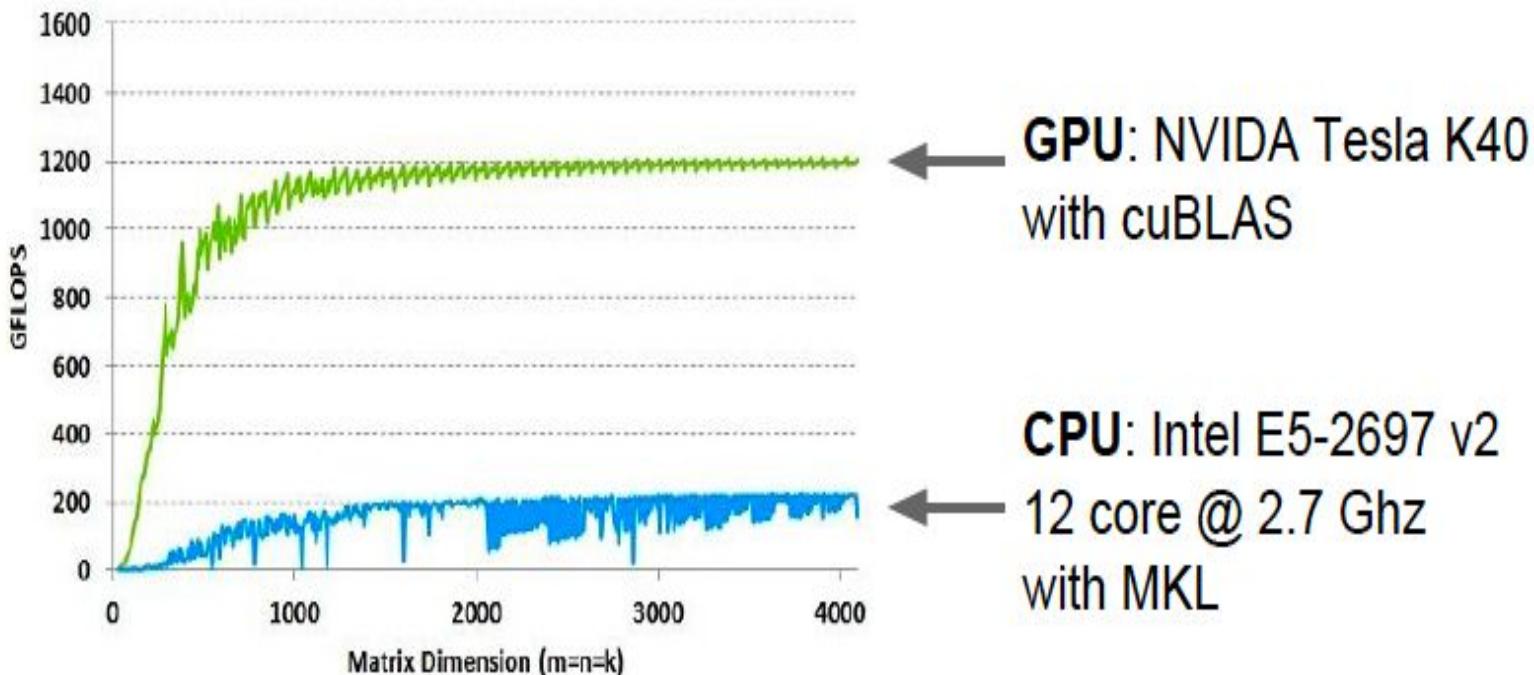
Good at parallel computation



# 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:



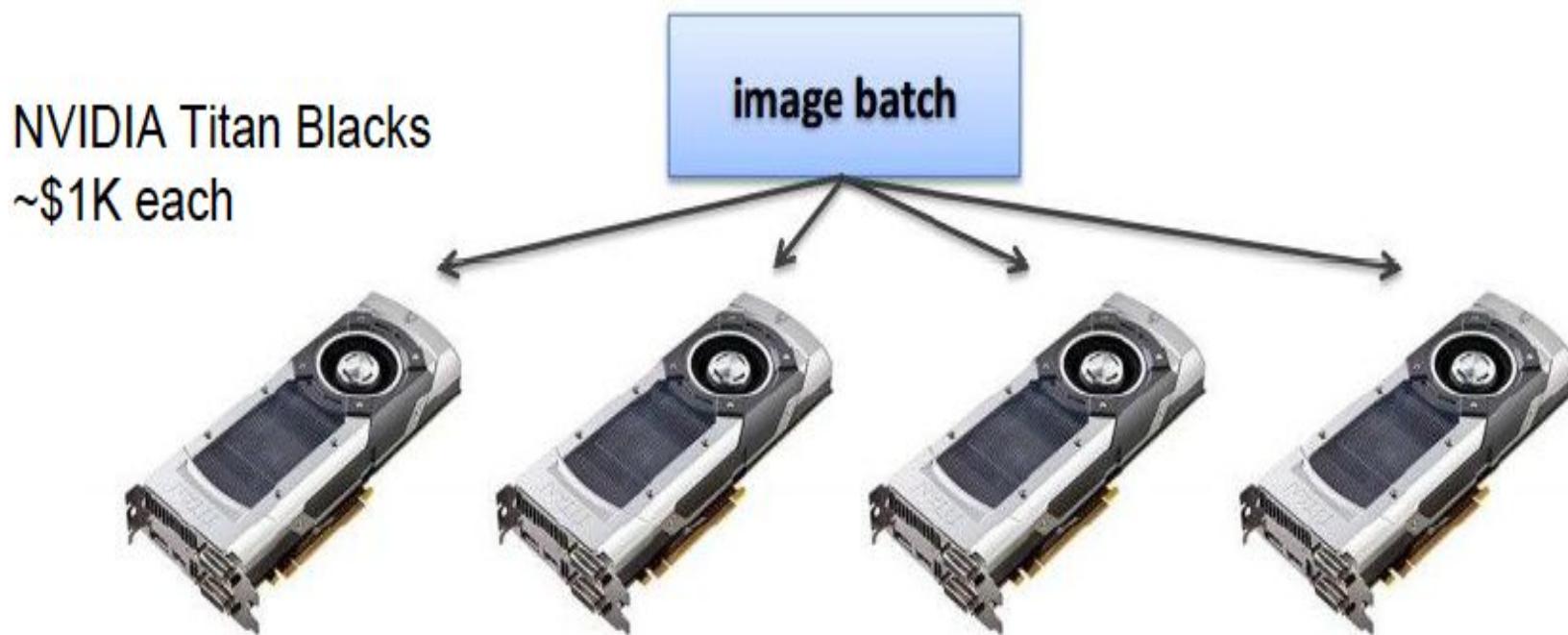
GPU: NVIDIA Tesla K40  
with cuBLAS

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

Even with GPUs, training can be slow

**VGG:** ~2-3 weeks training with 4 GPUs

**ResNet 101:** 2-3 weeks with 4 GPUs



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

# Bottlenecks

to be aware of



# **GPU - CPU communication is a bottleneck.**

=>

**CPU** data prefetch+augment thread running

while

**GPU** performs forward/backward pass

Moving parts lol

# CPU - disk bottleneck

Hard disk is slow to read from

=> 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

# Tips & Tricks

# Tips & Tricks

- Always shuffle. Never allow your network to go through exactly the same minibatch. If your framework allows it shuffle at every epoch.

# Tips & Tricks

- Always shuffle. Never allow your network to go through exactly the same minibatch. If your framework allows it shuffle at every epoch.
- Before training on the whole dataset try to overfit on a very small subset of it, that way you know your network can converge. Try a small, reliable network first.

# Tips & Tricks

- Always shuffle. Never allow your network to go through exactly the same minibatch. If your framework allows it shuffle at every epoch.
- Before training on the whole dataset try to overfit on a very small subset of it, that way you know your network can converge. Try a small, reliable network first.
- Always use Weight Decay & Dropout to minimize the chance of overfitting.

# Tips & Tricks

- Always shuffle. Never allow your network to go through exactly the same minibatch. If your framework allows it shuffle at every epoch.
- Before training on the whole dataset try to overfit on a very small subset of it, that way you know your network can converge. Try a small, reliable network first.
- Always use Weight Decay & Dropout to minimize the chance of overfitting.
- Avoid Sigmoid's , TanH's gates they are expensive and get saturated and may stop back propagation. Use the much cheaper and effective ReLU's and LeakyReLU's instead.

# Tips & Tricks

- Always shuffle. Never allow your network to go through exactly the same minibatch. If your framework allows it shuffle at every epoch.
- Before training on the whole dataset try to overfit on a very small subset of it, that way you know your network can converge. Try a small, reliable network first.
- Always use Weight Decay & Dropout to minimize the chance of overfitting.
- Avoid Sigmoid's , TanH's gates they are expensive and get saturated and may stop back propagation. Use the much cheaper and effective ReLU's and LeakyReLU's instead.
- Don't use ReLU or LeakyReLU's gates before max pooling, instead apply it after to save computation

# Tips & Tricks

- Always shuffle. Never allow your network to go through exactly the same minibatch. If your framework allows it shuffle at every epoch.
- Before training on the whole dataset try to overfit on a very small subset of it, that way you know your network can converge. Try a small, reliable network first.
- Always use Weight Decay & Dropout to minimize the chance of overfitting.
- Avoid Sigmoid's , TanH's gates they are expensive and get saturated and may stop back propagation. Use the much cheaper and effective ReLU's and LeakyReLU's instead.
- Don't use ReLU or LeakyReLU's gates before max pooling, instead apply it after to save computation
- Normalize the data!

# Tips & Tricks

- Always shuffle. Never allow your network to go through exactly the same minibatch. If your framework allows it shuffle at every epoch.
- Before training on the whole dataset try to overfit on a very small subset of it, that way you know your network can converge. Try a small, reliable network first.
- Always use Weight Decay & Dropout to minimize the chance of overfitting.
- Avoid Sigmoid's , TanH's gates they are expensive and get saturated and may stop back propagation. Use the much cheaper and effective ReLU's and LeakyReLU's instead.
- Don't use ReLU or LeakyReLU's gates before max pooling, instead apply it after to save computation
- Normalize the data! \*try\* to squeeze into [-1,1] (Pytorch approach)

# Tips & Tricks (2)

- Use **xavier** initialization as much as possible, especially on FC layers, instead of usual random gaussian init.

# Tips & Tricks (2)

- Use **xavier** initialization as much as possible, especially on FC layers, instead of usual random gaussian init.
- If your input data has a spatial parameter try to go for CNN's end to end (conv layer retain spatiality informations, remember!)

# Tips & Tricks (2)

- Use **xavier** initialization as much as possible, especially on FC layers, instead of usual random gaussian init.
- If your input data has a spatial parameter try to go for CNN's end to end (conv layer retain spatiality informations, remember!)
- Modify your models to use 1x1 CNN's layers where it is possible

# Tips & Tricks (2)

- Use **xavier** initialization as much as possible, especially on FC layers, instead of usual random gaussian init.
- If your input data has a spatial parameter try to go for CNN's end to end (conv layer retain spatiality informations, remember!)
- Modify your models to use 1x1 CNN's layers where it is possible
- Don't even try to train anything without a high end GPU. Well, you can try on some veeeeery small networks ;)

# Tips & Tricks (2)

And last but not least understand what you are doing!

# Tips & Tricks (2)

And last but not least understand what you are doing!

Deep Learning is the **Neutron Bomb** of Machine Learning, BUT...

# Tips & Tricks (2)

And last but not least understand what you are doing!

Deep Learning is the **Neutron Bomb** of Machine Learning, BUT...

Understand the architecture you are using and what you are trying to achieve.

Don't mindlessly copy models!

**That's all!**