Lecture 11:

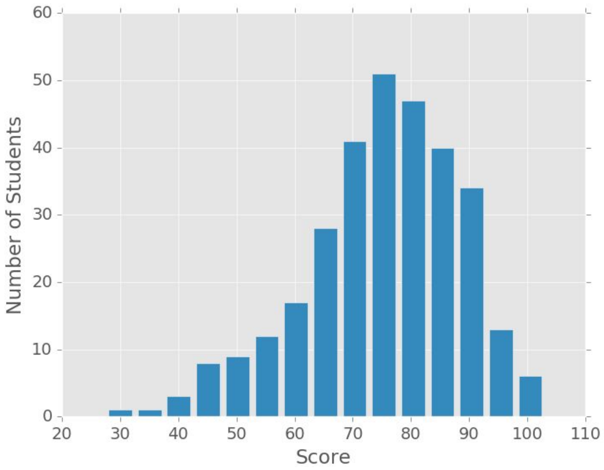
CNNs in Practice

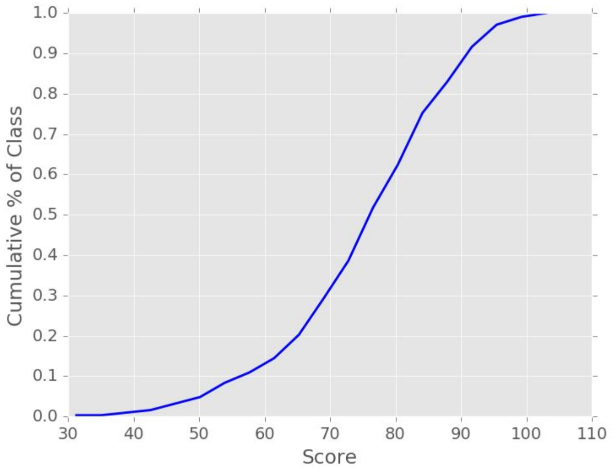
Administrative

- Midterms are graded!
    - Pick up now
    - Or in Andrej, Justin, Albert, or Serena’s OH
- Project milestone due today, 2/17 by midnight
    - Turn in to Assignments tab on Coursework!
- Assignment 2 grades soon
- Assignment 3 released, due 2/24

Midterm stats

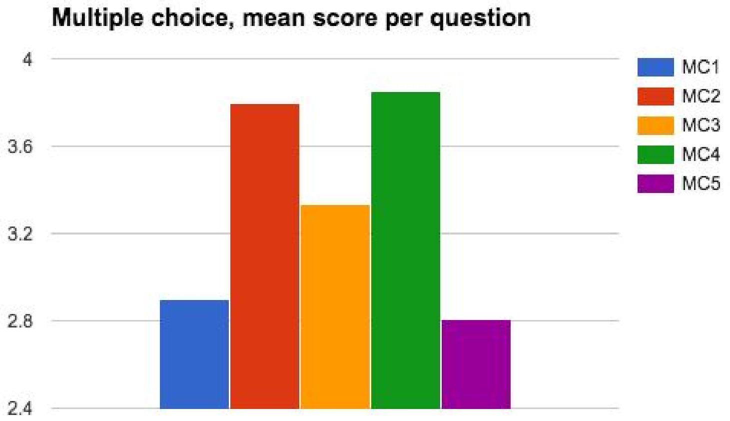
Mean: 75.0 **Median:** 76.3 **Standard Deviation:** 13.2
N: 311 **Max:** 103.0

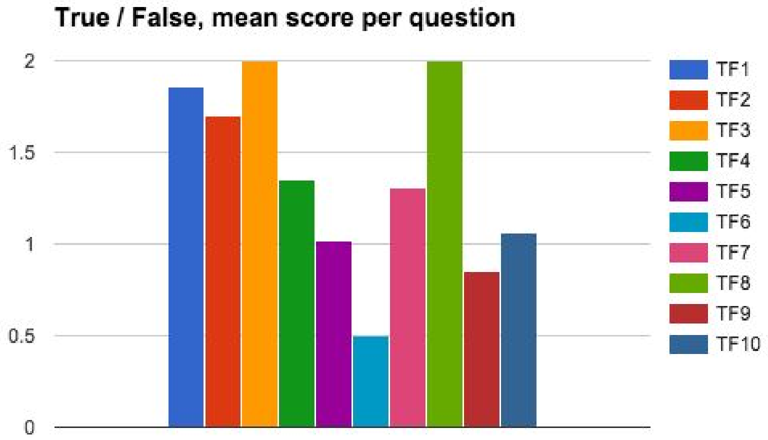




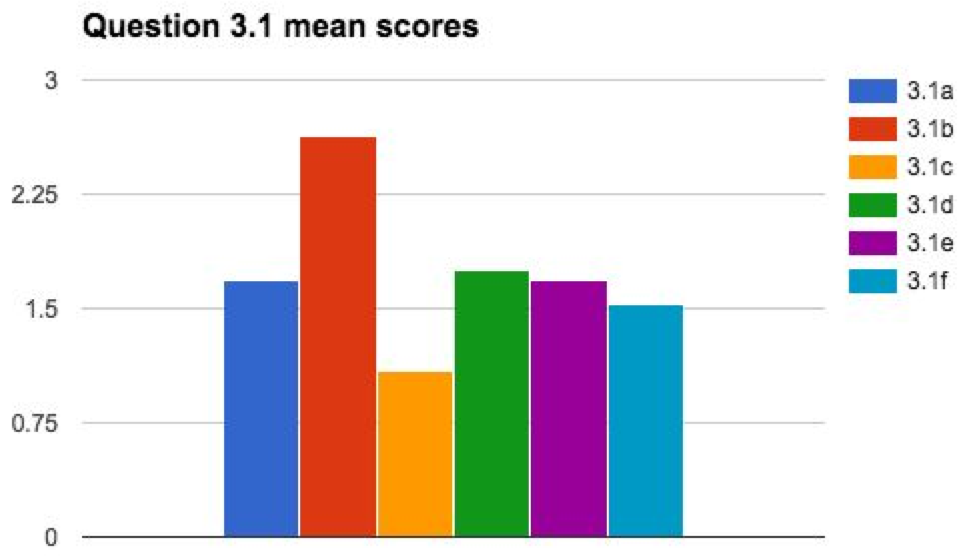
Midterm stats

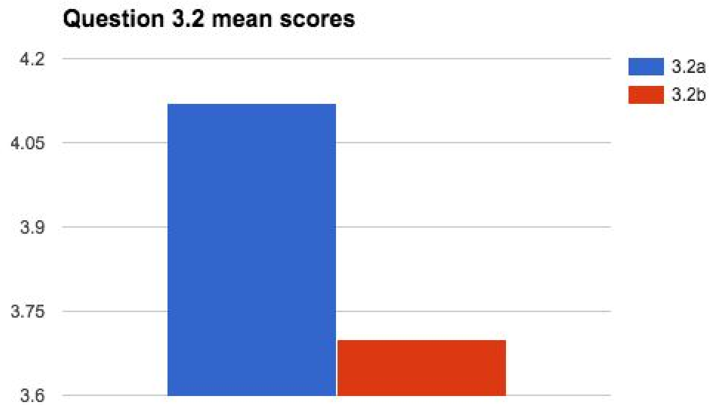
[We threw out TF3 and TF8]



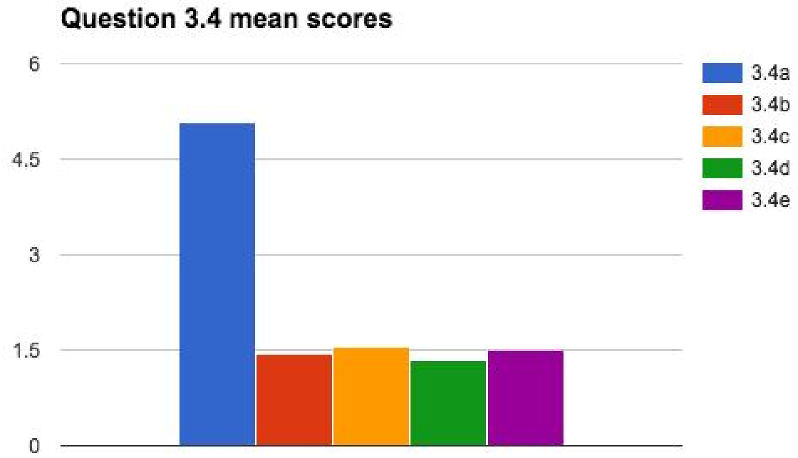


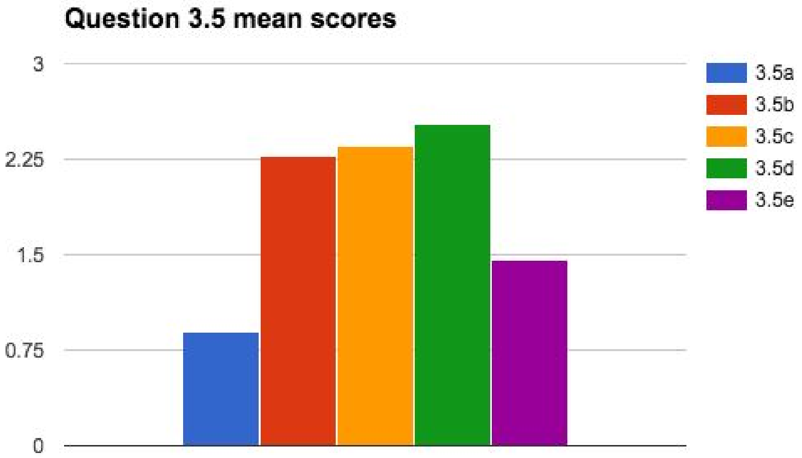
Midterm stats





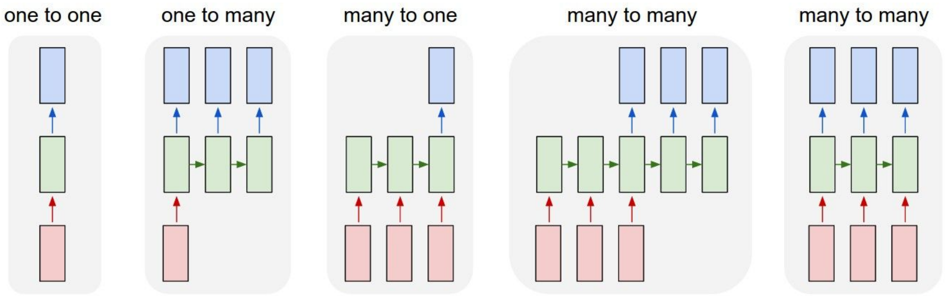
Midterm Stats





Bonus mean: 0.8

Last Time



Recurrent neural networks for modeling sequences

Vanilla RNNs

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

LSTMs

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

Last Time

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

Lemma 0.1. *Assume (3) and (3) by the construction in the description.*

Suppose $X = \lim |X|$ (by the formal open covering X and a single map $\underline{Proj}_X(\mathcal{A}) =$ Spec(B) *over U compatible with the complex*

$$Set(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \to \mathcal{C}_{Z/X}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If T is surjective we may assume that T is connected with residue fields of S. Moreover there exists a closed subspace $Z \subset X$ of X where U in X' is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1) *f is locally of finite type. Since $S =$ Spec(R) and $Y =$ Spec(R).*

Proof. This is form all sheaves of sheaves on X. But given a scheme U and a surjective étale morphism $U \to X$. Let $U \cap U = \coprod_{i=1,\ldots,n} U_i$ be the scheme X over S at the schemes $X_i \to X$ and $U = \lim_i X_i$. □

The following lemma surjective restrocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{X,\ldots,0}$.

Lemma 0.2. *Let X be a locally Noetherian scheme over S, $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq \mathfrak{p}$ is a subset of $\mathcal{J}_{n,0} \circ \overline{A}_2$ works.*

Lemma 0.3. *In Situation ??. Hence we may assume $\mathfrak{q}' = 0$.*

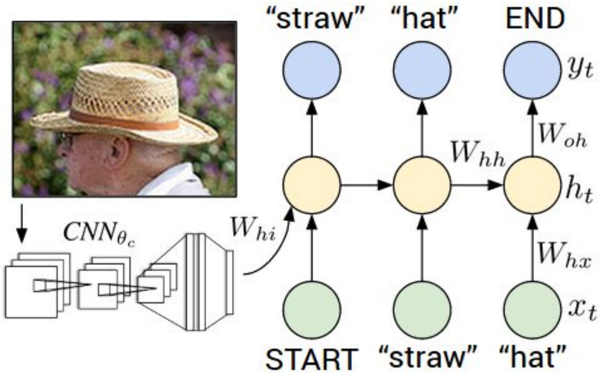
Proof. We will use the property we see that $\mathfrak{p}$ is the mext functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

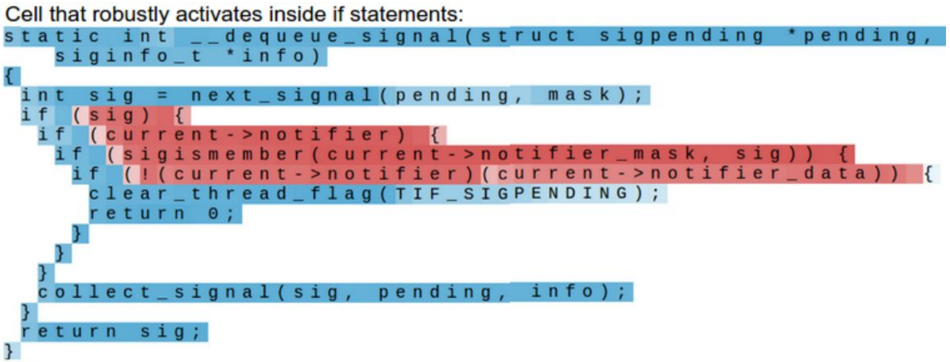
where K is an F-algebra where δ_{n+1} is a scheme over S. □

Sampling from RNN language models to generate text

Last Time



CNN + RNN for image captioning



Interpretable RNN cells

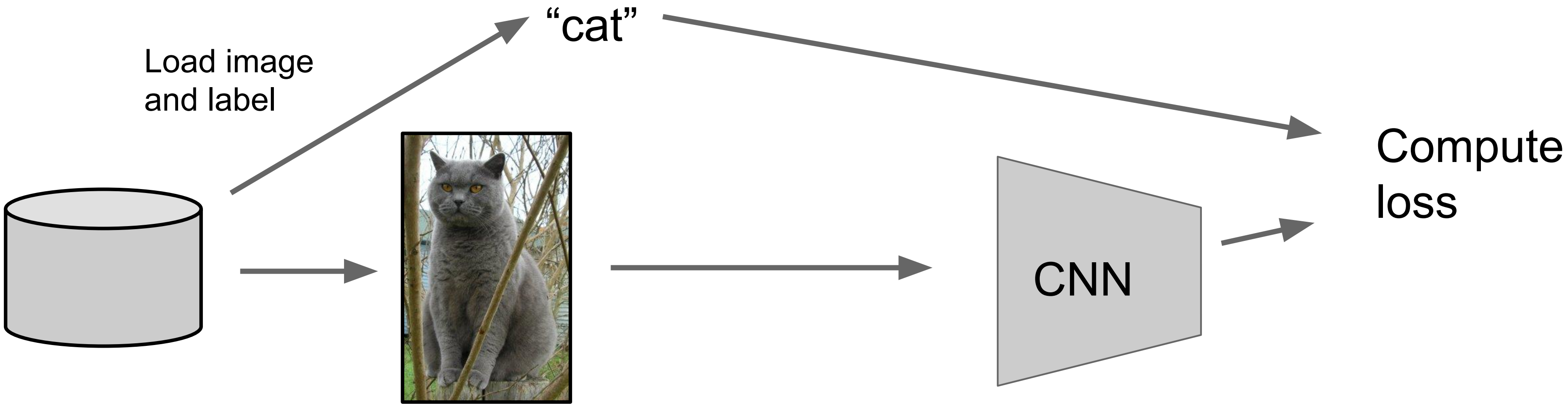
Today

Working with CNNs in practice:

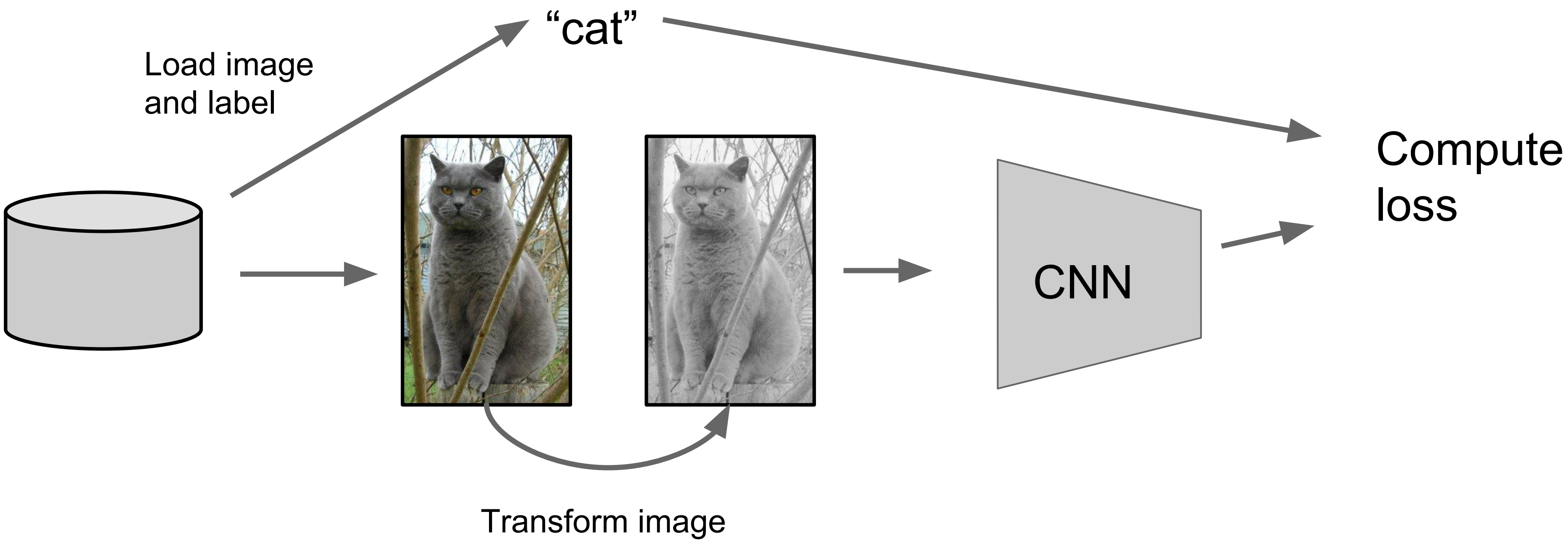
- Making the most of your data
  - Data augmentation
  - Transfer learning
- All about convolutions:
  - How to arrange them
  - How to compute them fast
- "Implementation details"
  - GPU / CPU, bottlenecks, distributed training

Data Augmentation

Data Augmentation



Data Augmentation

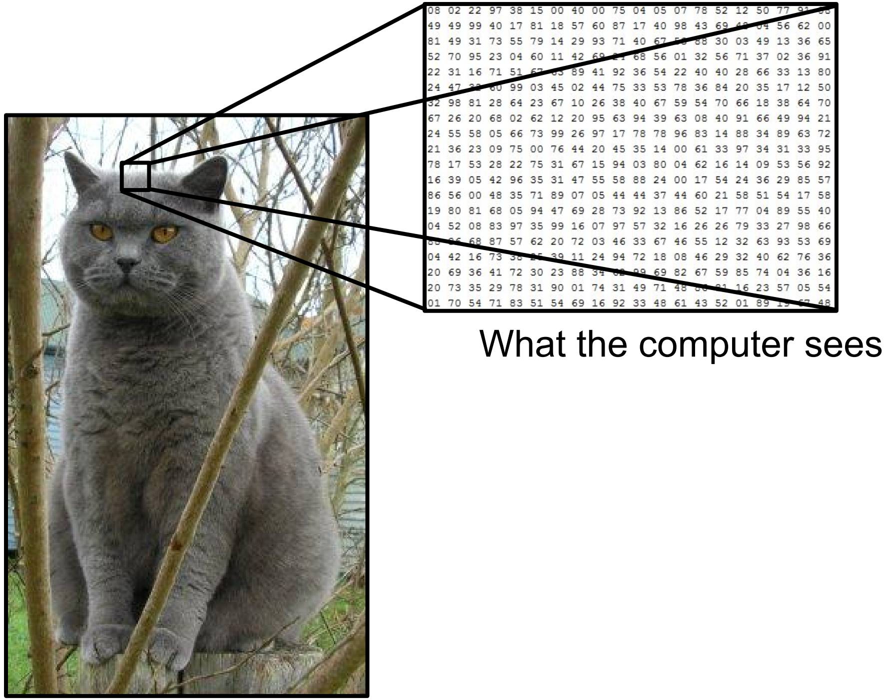


Data Augmentation

- Change the pixels without changing the label

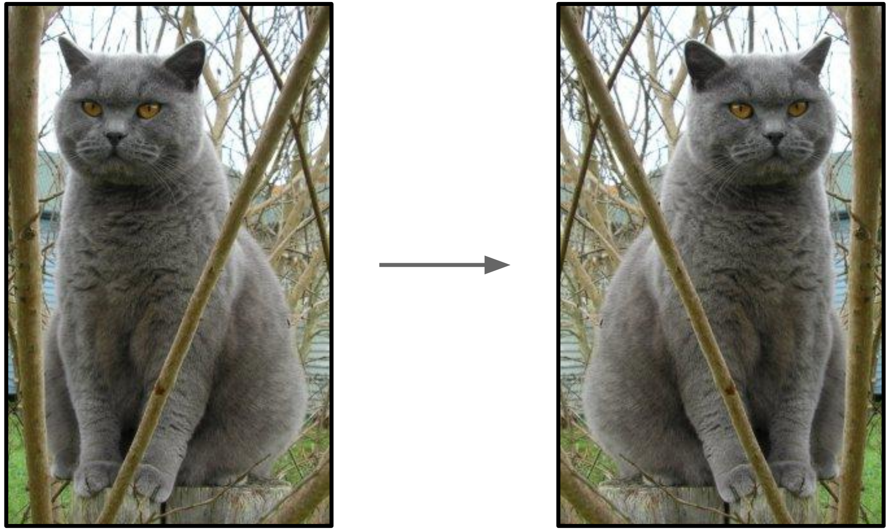
- Train on transformed data

- VERY widely used



Data Augmentation

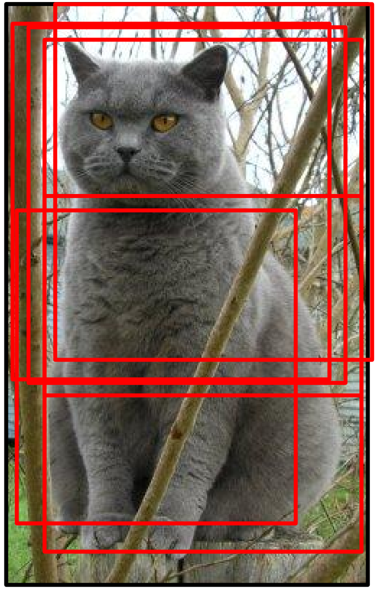
1. Horizontal flips



Data Augmentation

2. Random crops/scales

Training: sample random crops / scales

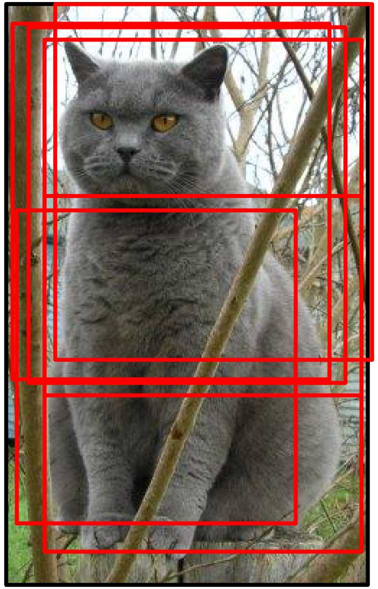


Data Augmentation
2. Random crops/scales

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 x 224 patch



Data Augmentation

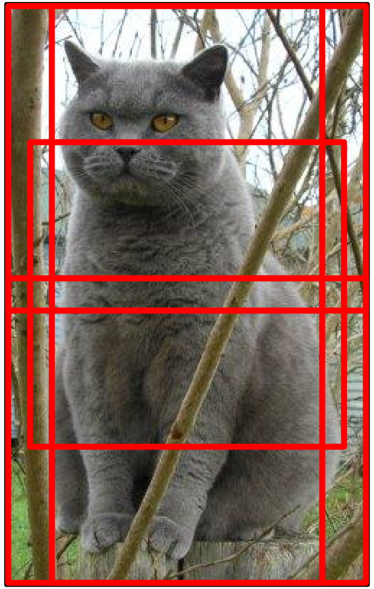
2. Random crops/scales

Training: sample random crops / scales

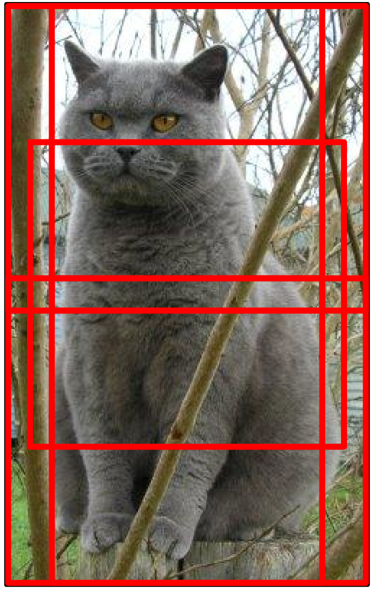
ResNet:

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

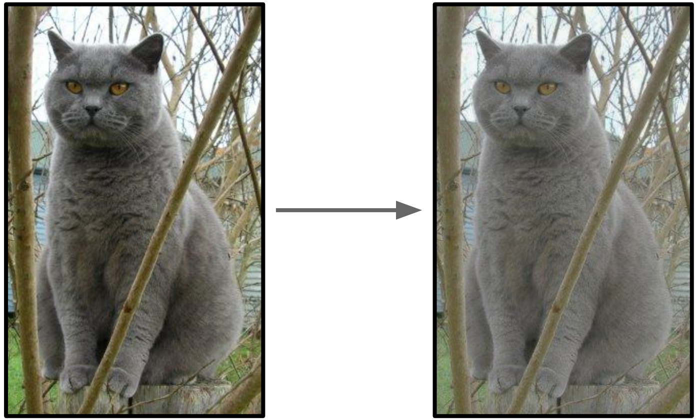
ResNet:

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

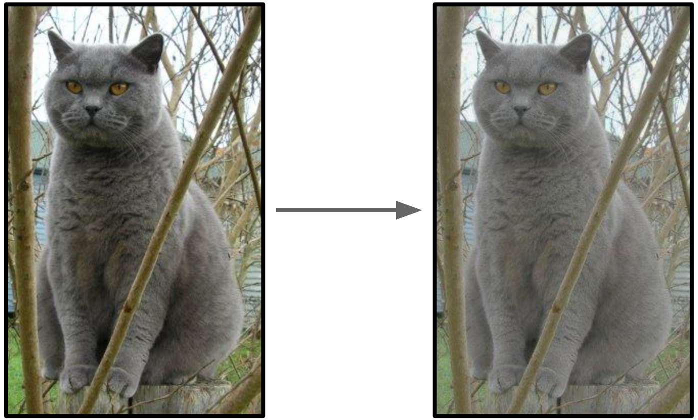
3. Color jitter

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

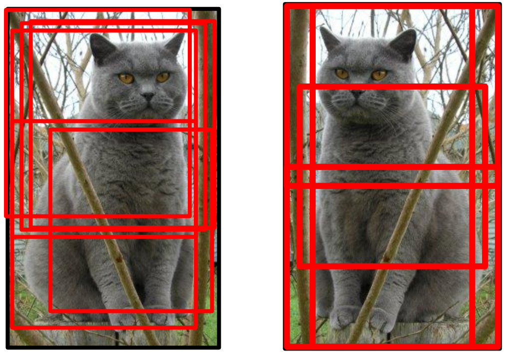
4. Get creative!

Random mix/combinations of :

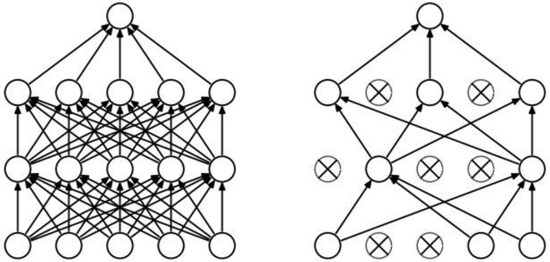
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

A general theme:

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



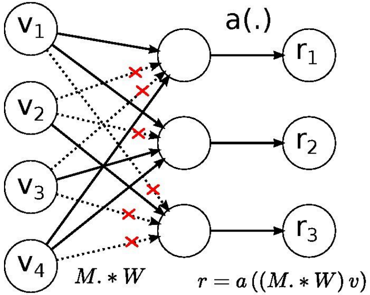
Data Augmentation



(a) Standard Neural Net

(b) After applying dropout.

Dropout



DropConnect

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 a data if you want to train/use CNNs"

Transfer Learning

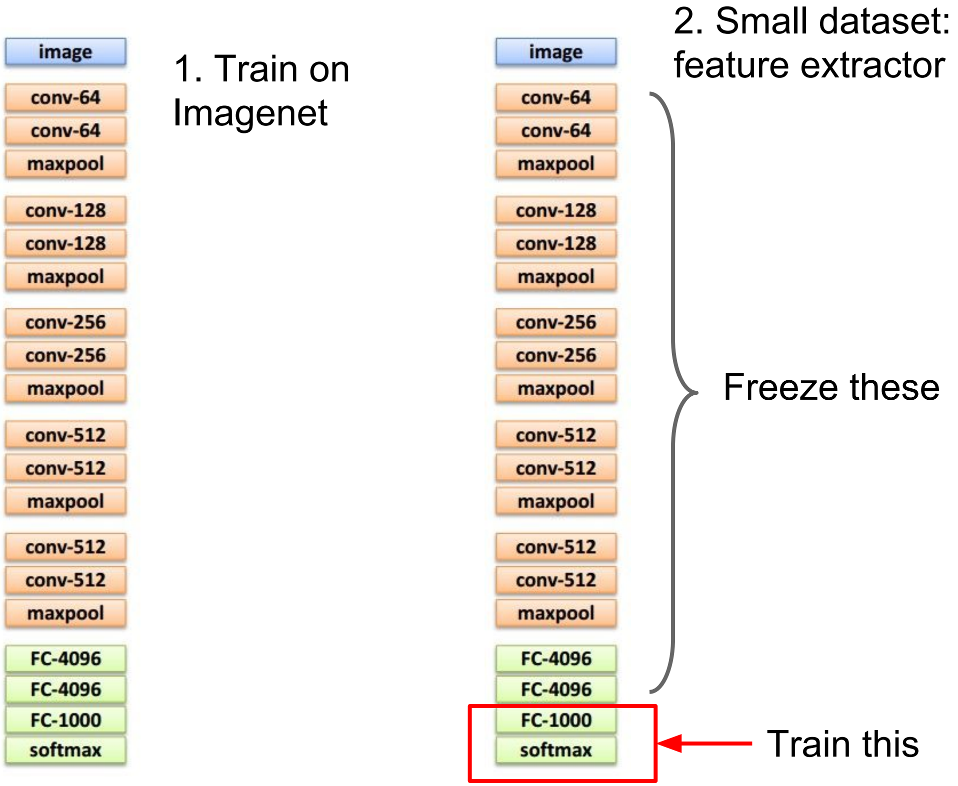
"You need a lot of a data if you want to train/use CNNs"

BUSTED

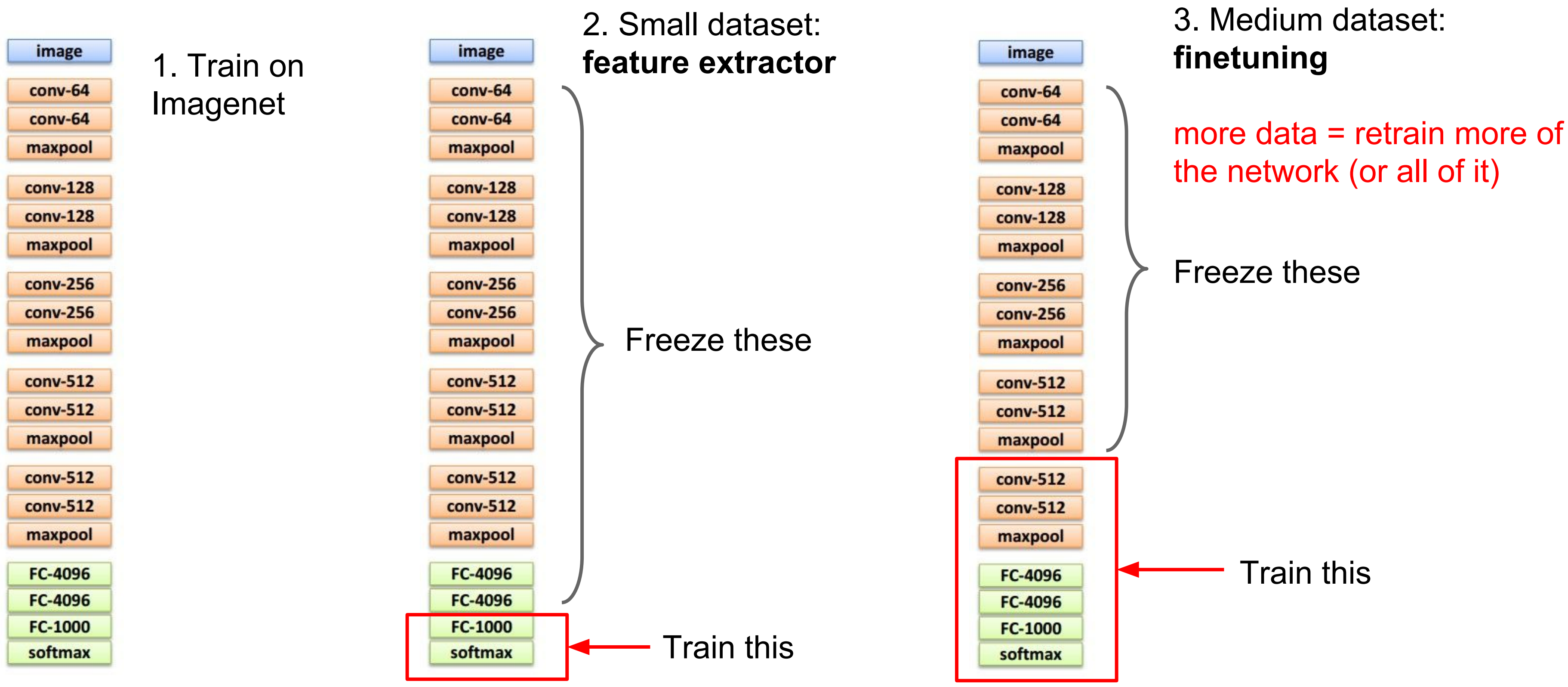
Transfer Learning with CNNs

1. Train on Imagenet

Transfer Learning with CNNs

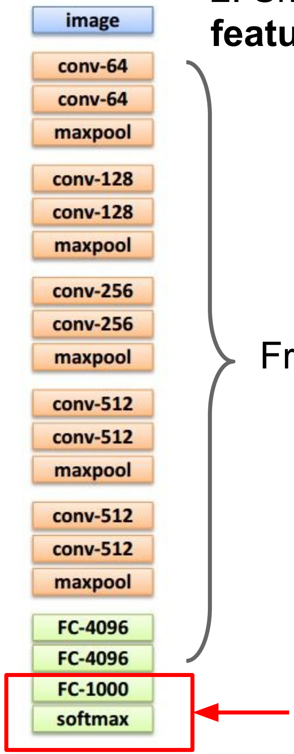


Transfer Learning with CNNs



Transfer Learning with CNNs

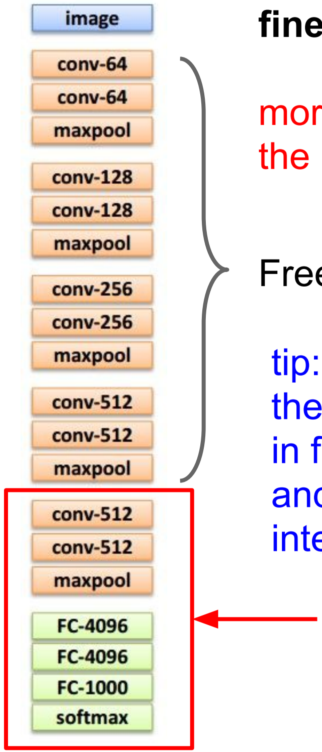
1. Train on Imagenet



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

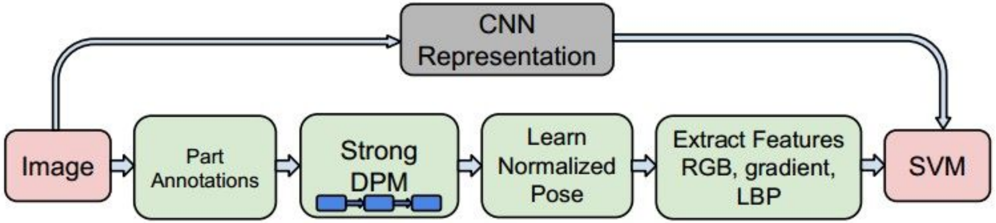
tip: use only ~1/10th of the original learning rate in finetuning top layer, and ~1/100th on intermediate layers

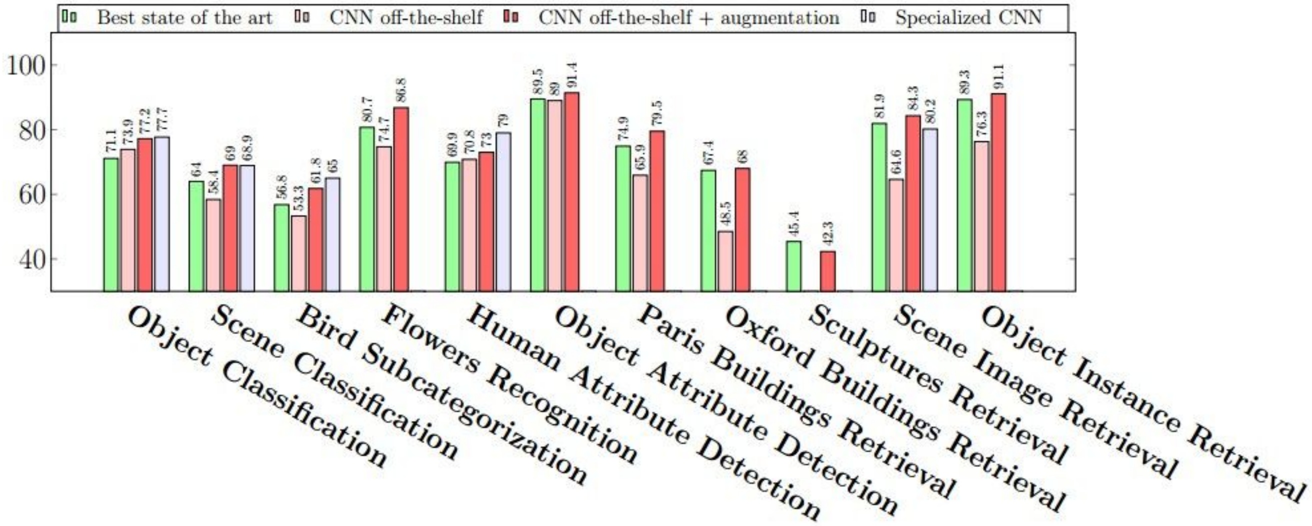
Train this

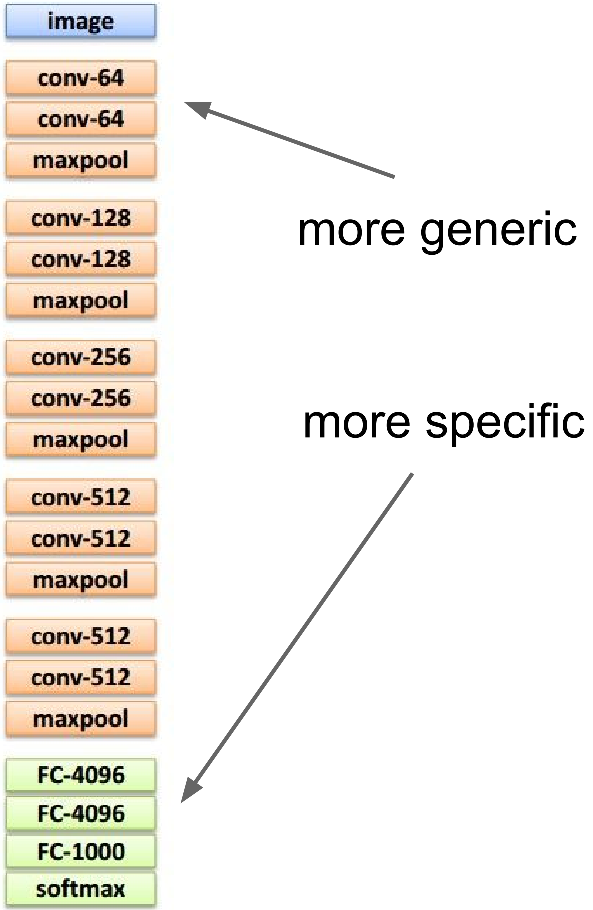
CNN Features off-the-shelf: an Astounding Baseline for Recognition [Razavian et al, 2014]

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

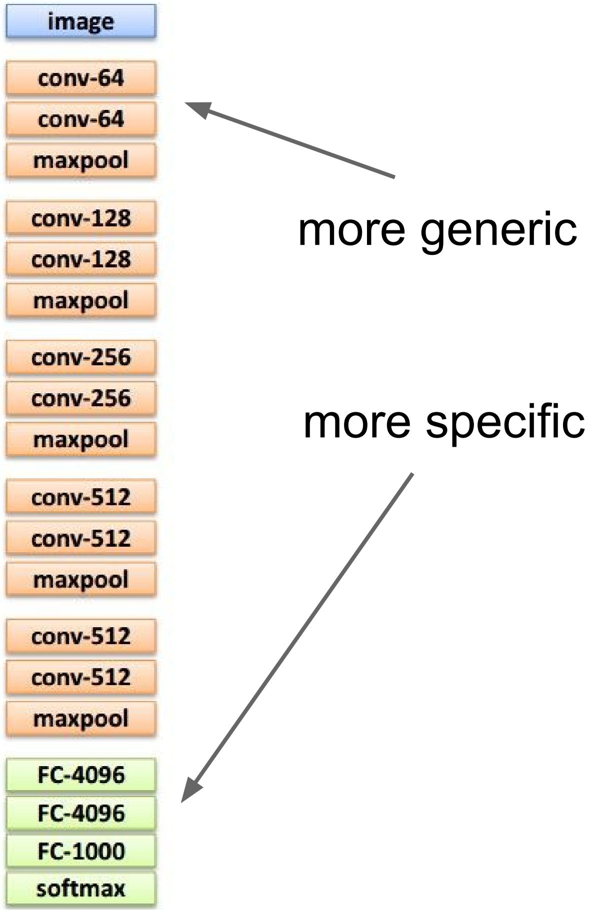
	$DeCAF_6$	$DeCAF_7$
LogReg	**40.94 ± 0.3**	40.84 ± 0.3
SVM	39.36 ± 0.3	40.66 ± 0.3
Xiao et al. (2010)	38.0	



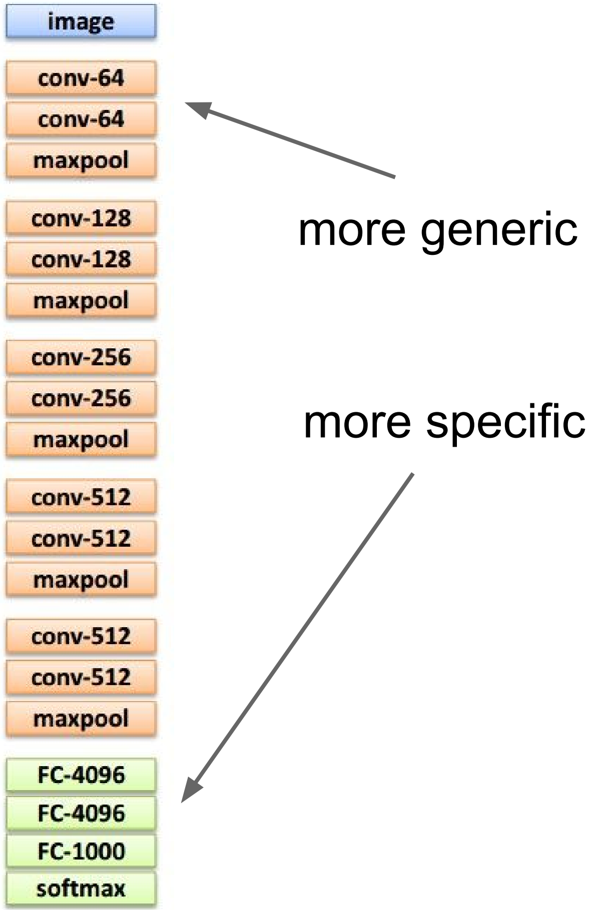




	very similar dataset	**very different dataset**
very little data	?	?
quite a lot of data	?	?

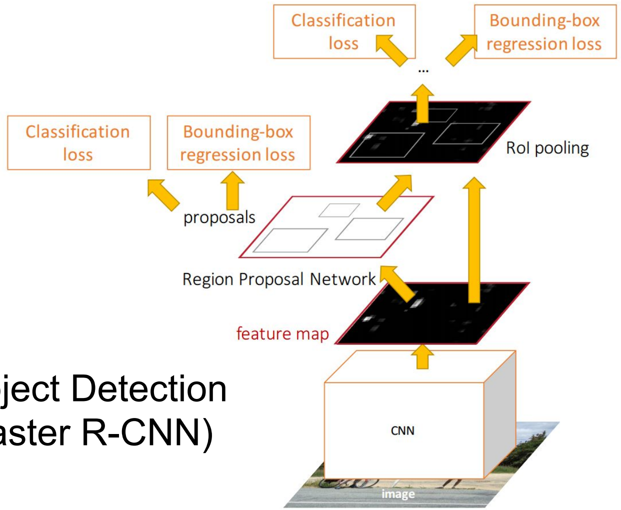


	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



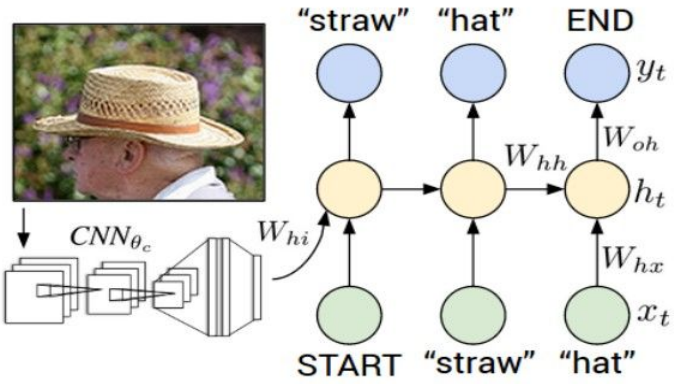
	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

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

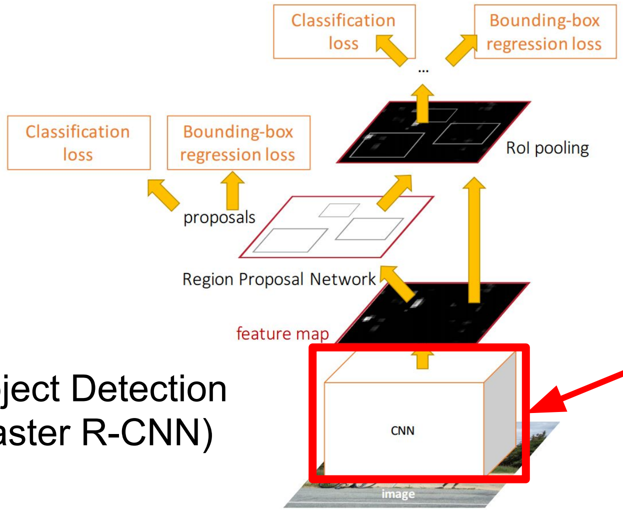


Object Detection
(Faster R-CNN)

Image Captioning: CNN + RNN



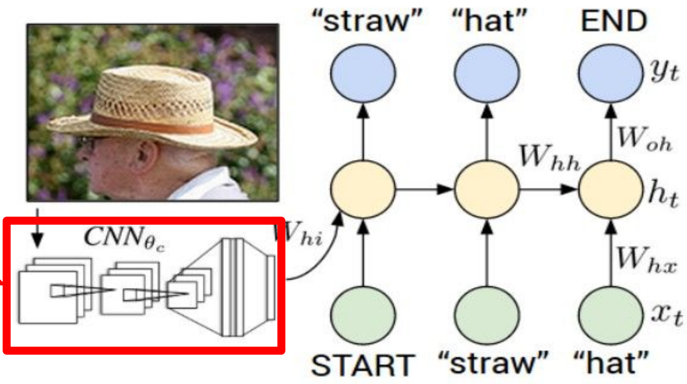
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)



Object Detection
(Faster R-CNN)

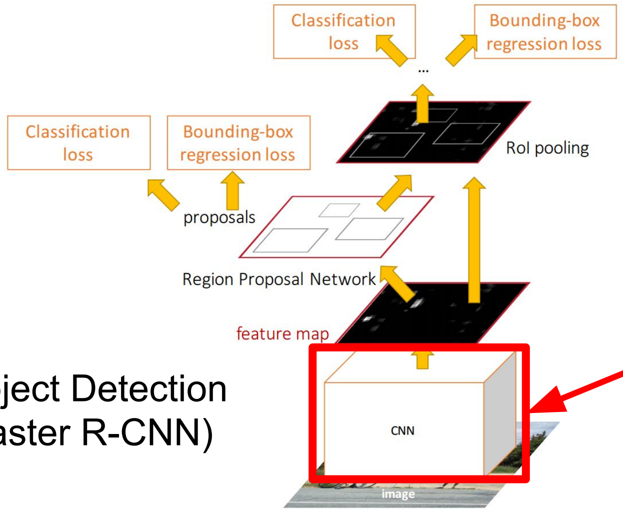
CNN pretrained on ImageNet

Image Captioning: CNN + RNN



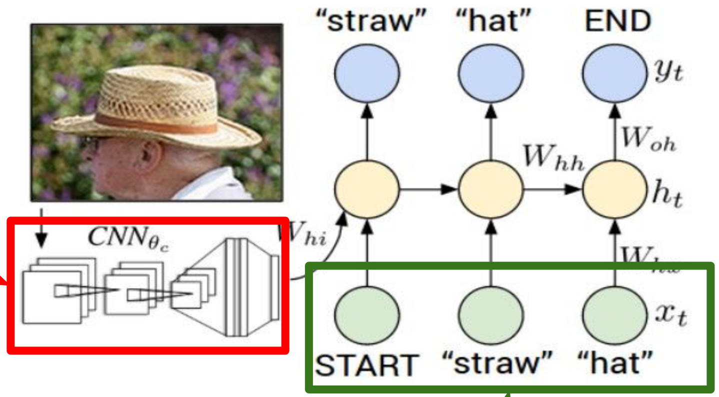
Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)

Object Detection
(Faster R-CNN)



CNN pretrained
on ImageNet

Image Captioning: CNN + RNN



Word vectors pretrained
from word2vec

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

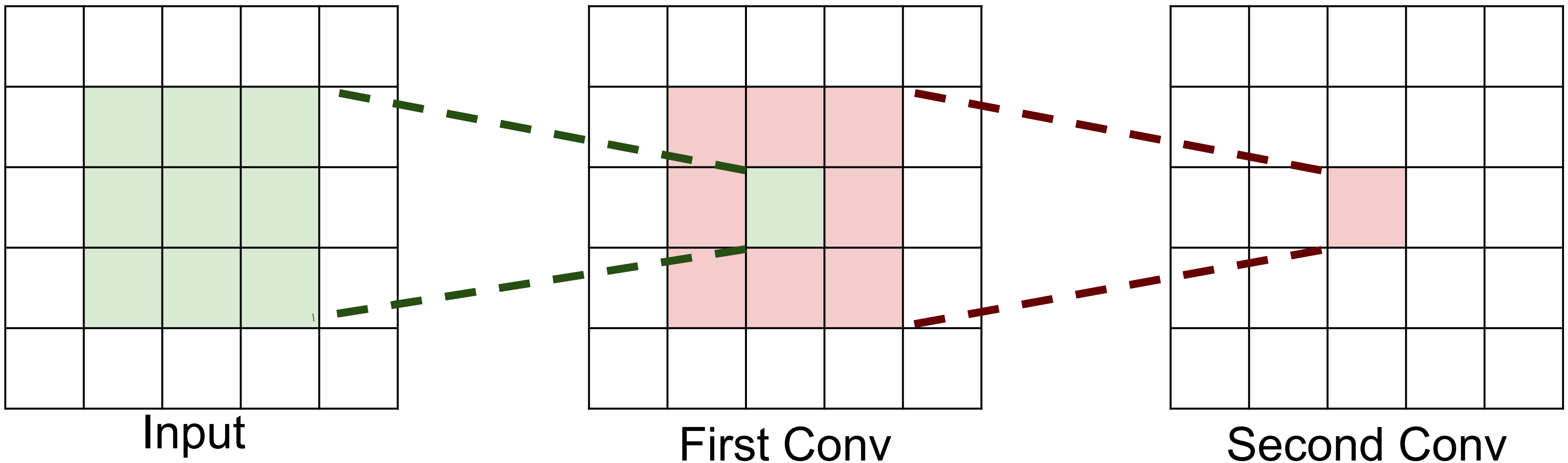
All About Convolutions

All About Convolutions

Part I: How to stack them

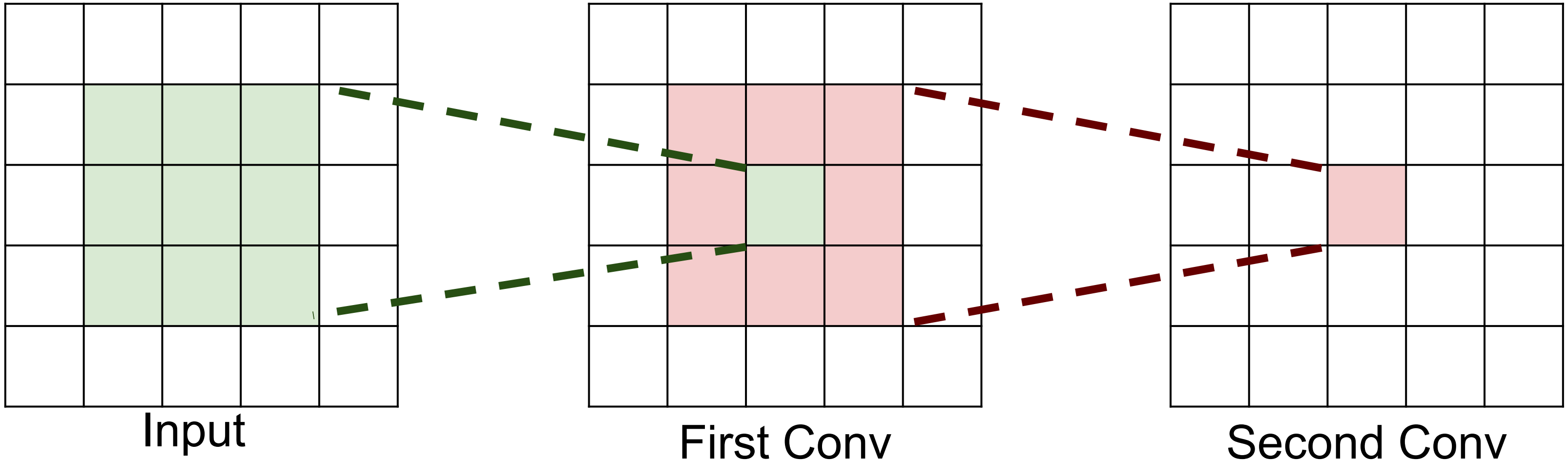
The power of small filters

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



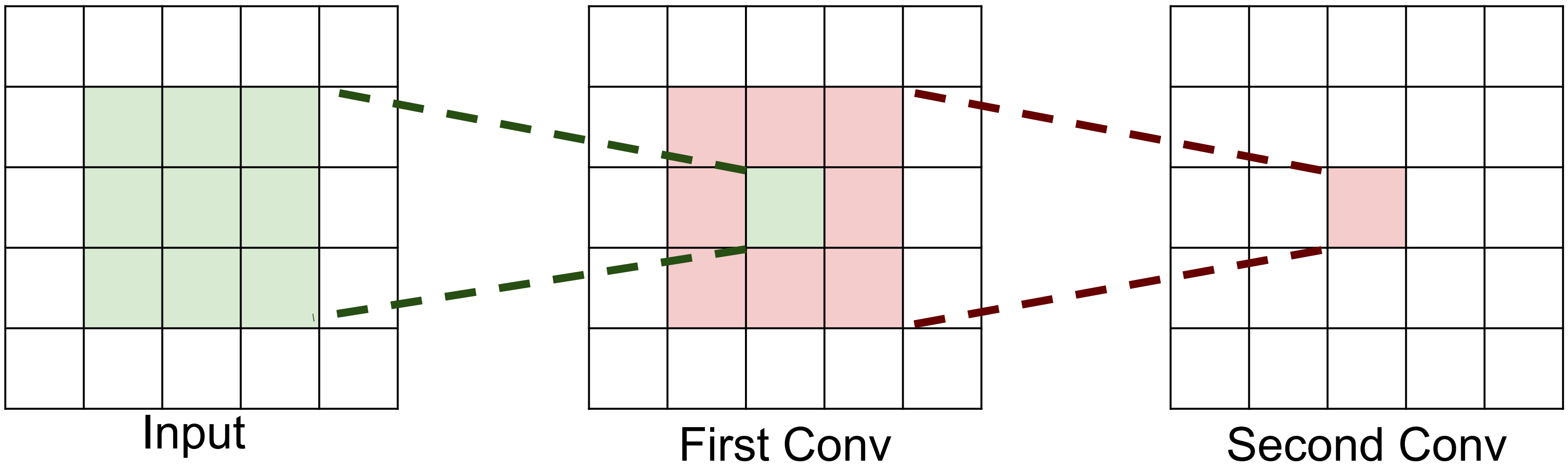
The power of small filters

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



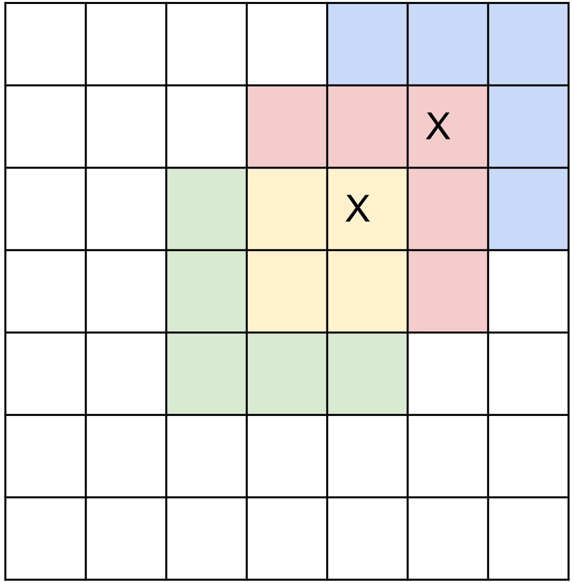
The power of small filters

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?

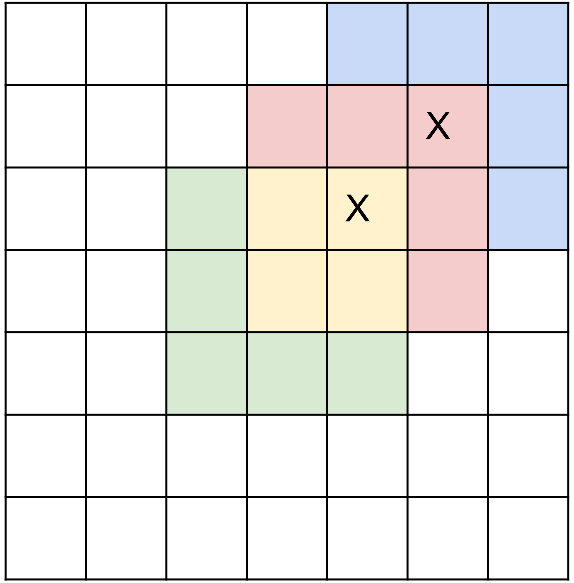
Answer: 7 x 7



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

Number of multiply-adds:

three CONV with 3 x 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 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:
$= C \times (7 \times 7 \times C) = 49\ C^2$

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

three CONV with 3 x 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

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?

H x W x C

Conv 1x1, C/2 filters ↓

H x W x (C / 2)

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?

H x W x C

Conv 1x1, C/2 filters ↓

H x W x (C / 2)

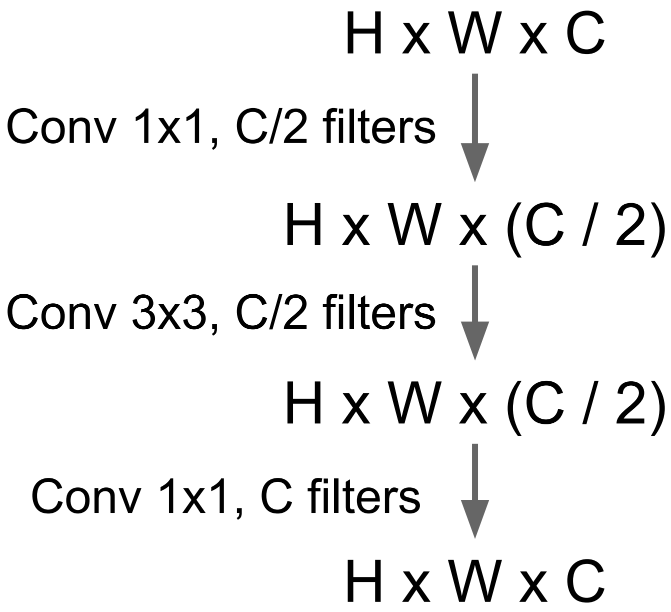
Conv 3x3, C/2 filters ↓

H x W x (C / 2)

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?

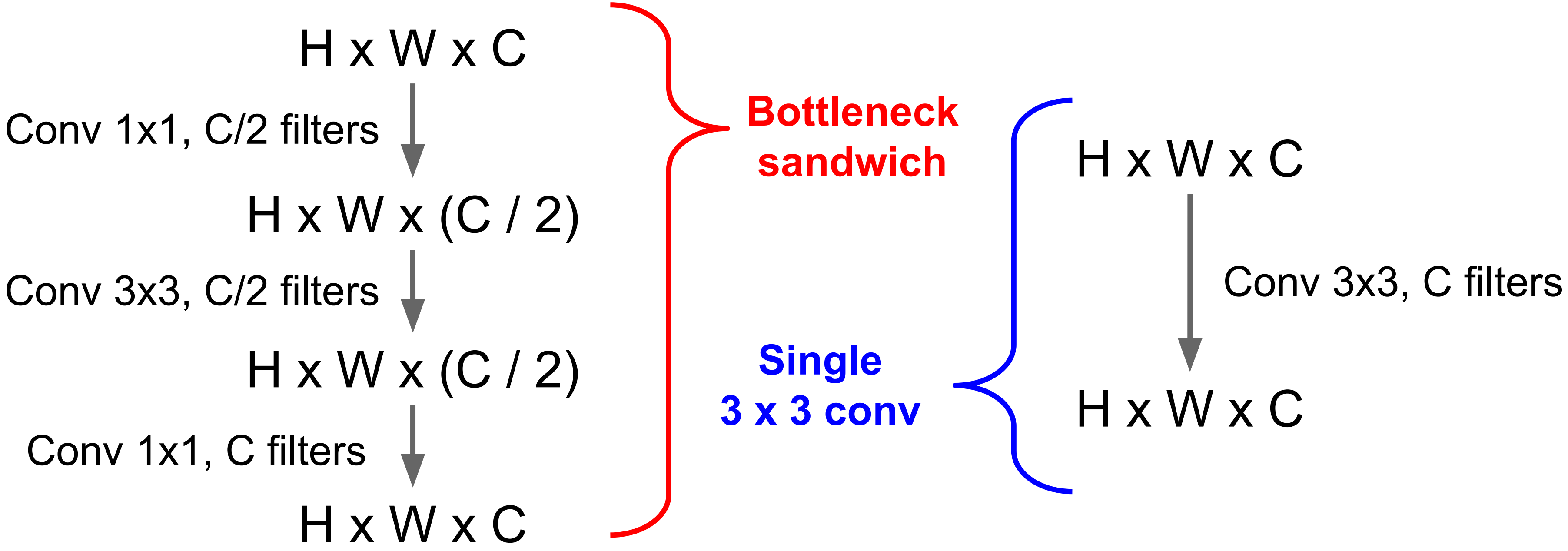


1. "bottleneck" 1 x 1 conv to reduce dimension
2. 3 x 3 conv at reduced dimension
3. Restore dimension with another 1 x 1 conv

[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!

H x W x C

Conv 1x1, C/2 filters

H x W x (C / 2)

Conv 3x3, C/2 filters

H x W x (C / 2)

Conv 1x1, C filters

H x W x C

3.25 C^2 parameters

9 C^2 parameters

H x W x C

Conv 3x3, C filters

H x W x C

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?

H x W x C

Conv 1x3, C filters

H x W x C

Conv 3x1, C filters

H x W x C

The power of small filters

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

More nonlinearity,
fewer params,
less compute!

H x W x C

Conv 1x3, C filters

H x W x C

Conv 3x1, C filters

H x W x C

$6\ C^2$ parameters

$9\ C^2$ parameters

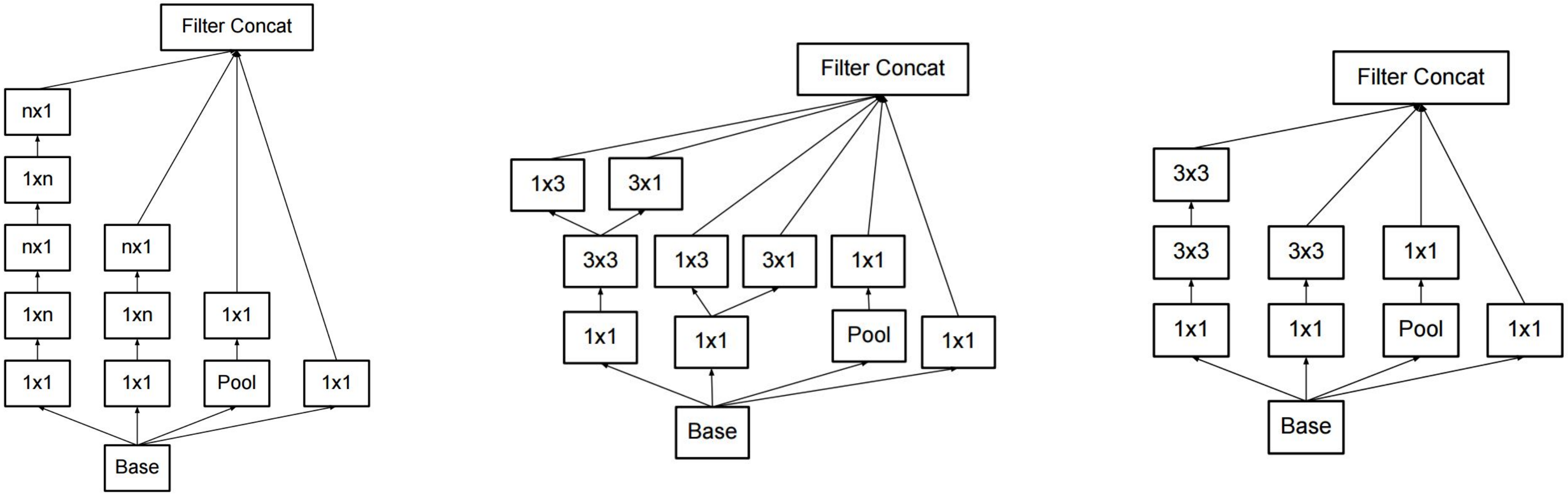
H x W x C

Conv 3x3, C filters

H x W x C

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

All About Convolutions
Part II: How to compute them

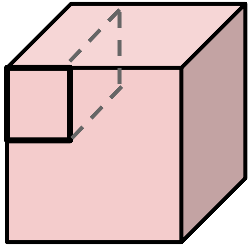
Implementing Convolutions: im2col

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

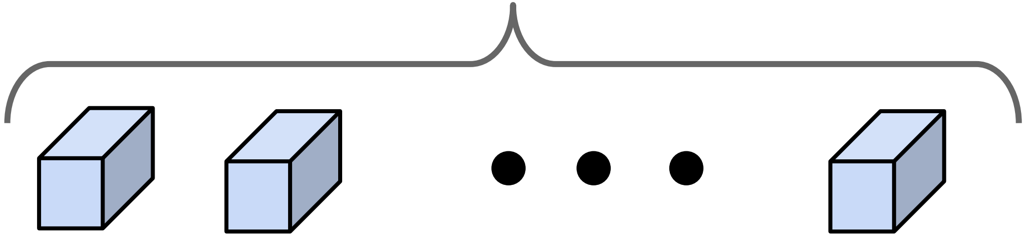
Can we turn convolution into matrix multiplication?

Implementing Convolutions: im2col

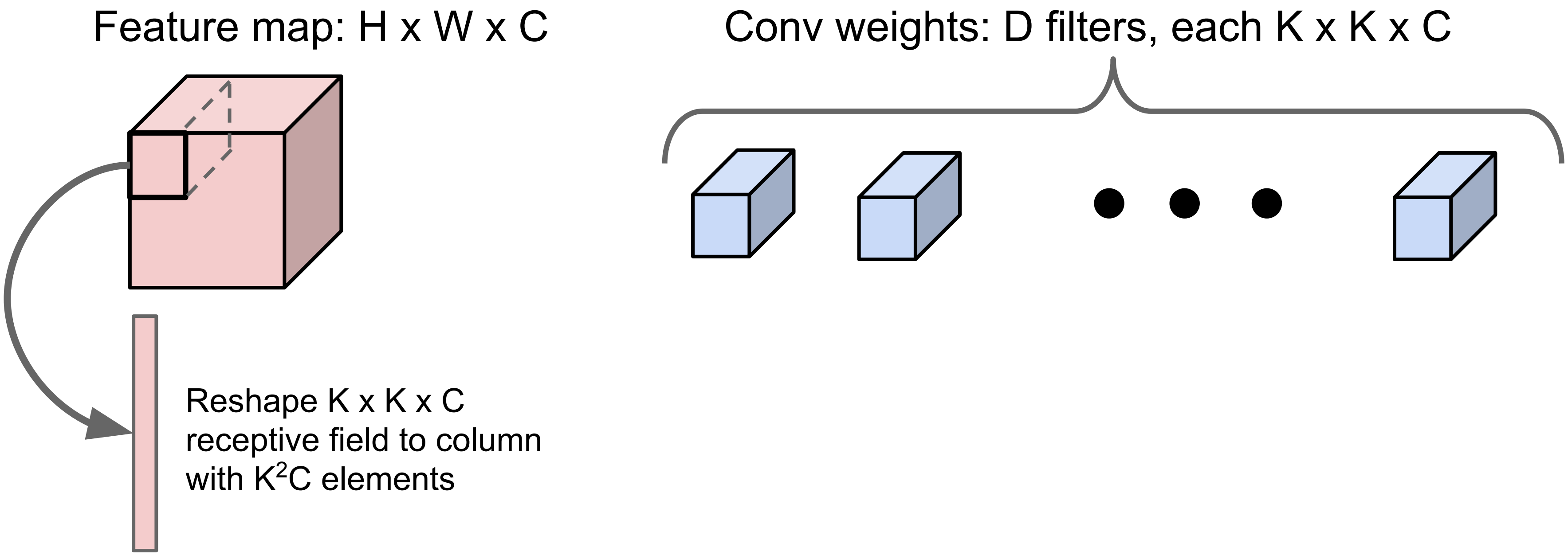
Feature map: H x W x C



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



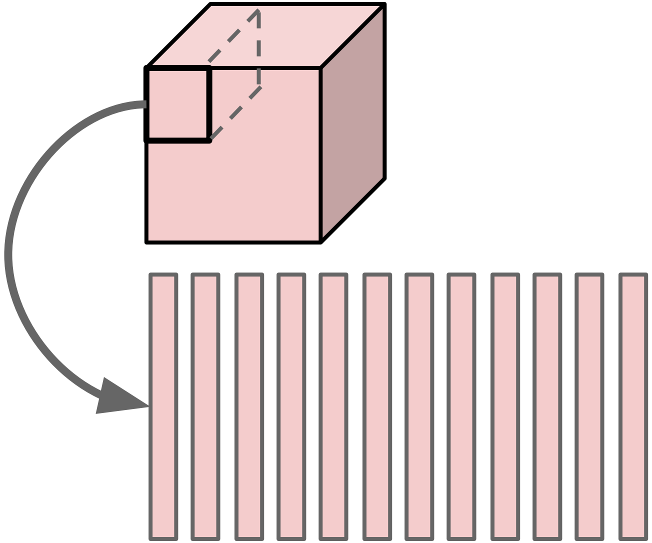
Implementing Convolutions: im2col

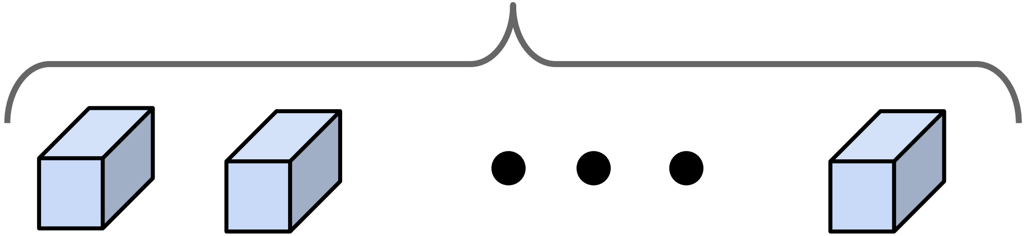


Implementing Convolutions: im2col

Feature map: H x W x C

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



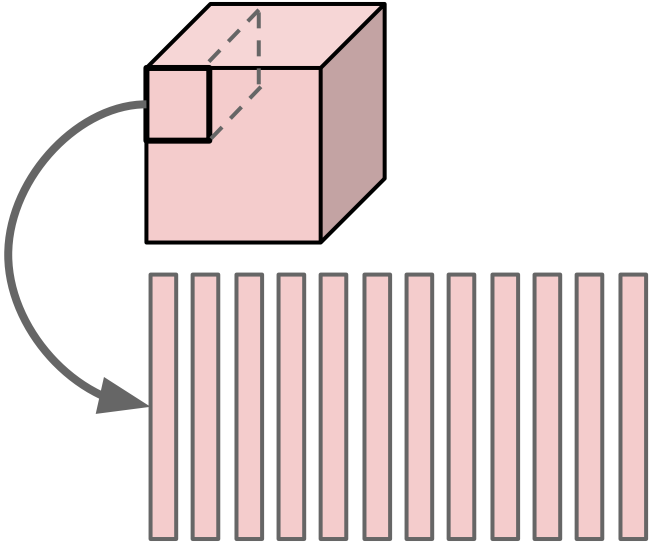


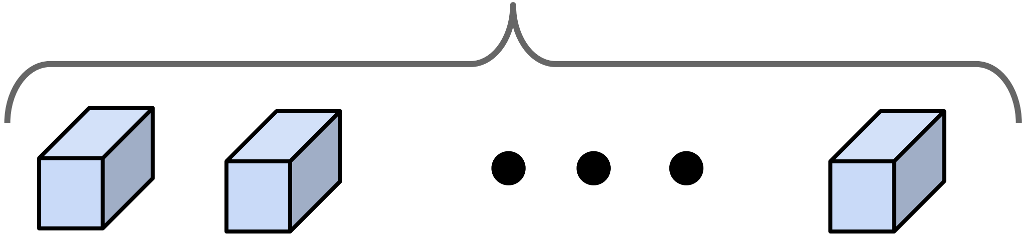
Repeat for all columns to get $(K^2C) \times N$ matrix
(N receptive field locations)

Implementing Convolutions: im2col

Feature map: H x W x C

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

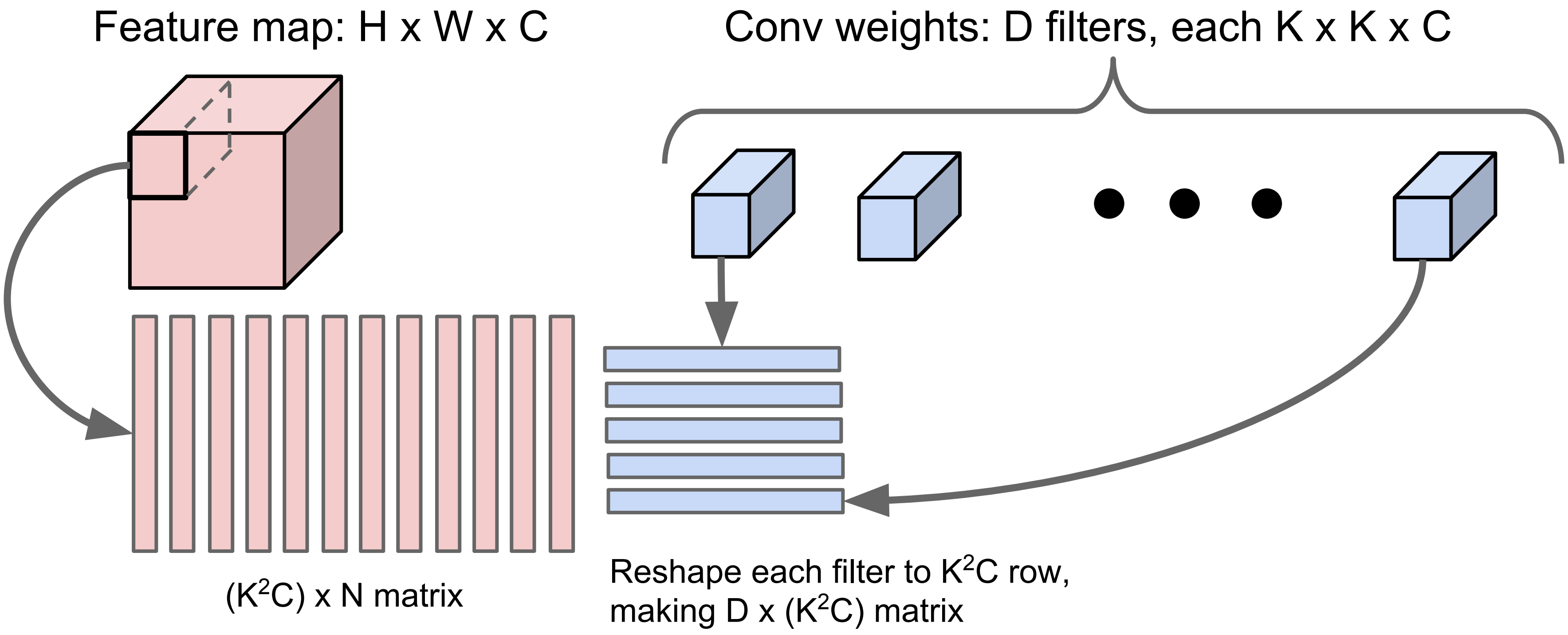




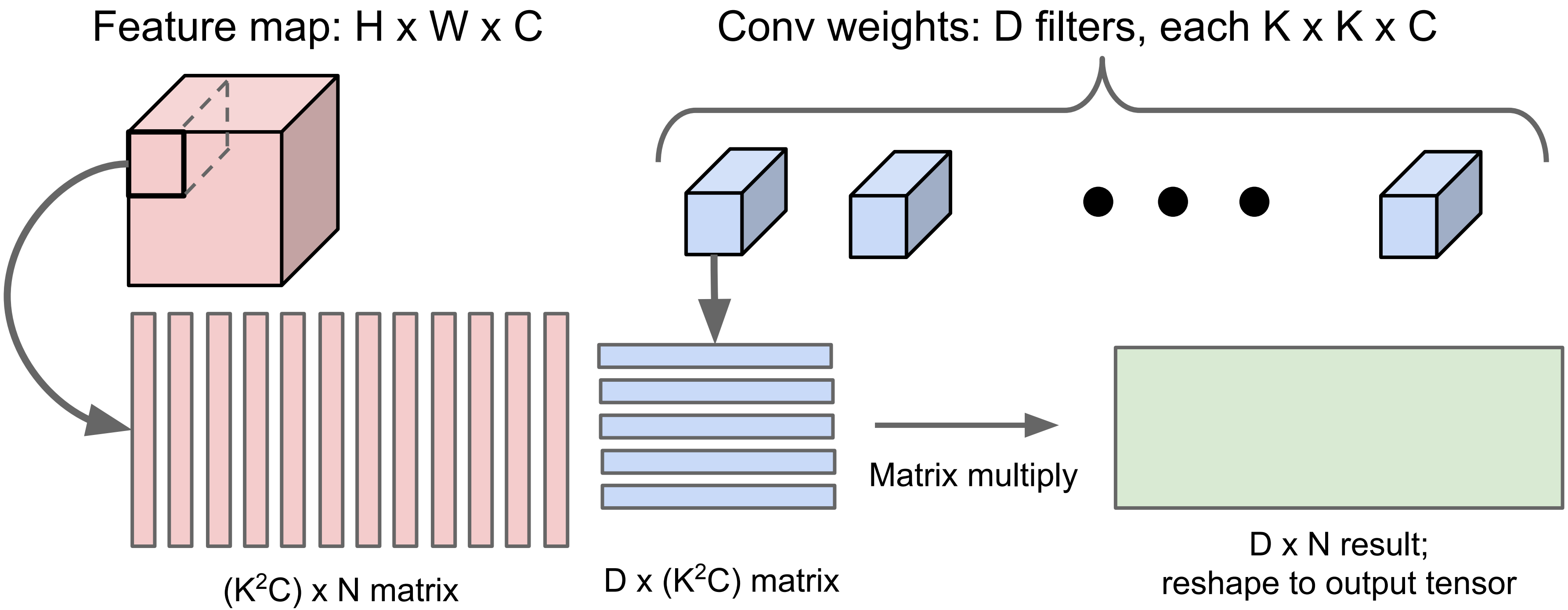
Elements appearing in multiple receptive fields are duplicated; this uses a lot of memory

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

Implementing Convolutions: im2col



Implementing Convolutions: im2col



```
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 multplication.
      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

```python
def conv_forward_strides(x, w, b, conv_param):
  N, C, H, W = x.shape
  F, _, HH, WW = w.shape
  stride, pad = conv_param['stride'], conv_param['pad']

  # Check dimensions
  assert (W + 2 * pad - WW) % stride == 0, 'width does not work'
  assert (H + 2 * pad - HH) % stride == 0, 'height does not work'

  # Pad the input
  p = pad
  x_padded = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')

  # Figure out output dimensions
  H += 2 * pad
  W += 2 * pad
  out_h = (H - HH) / stride + 1
  out_w = (W - WW) / stride + 1

  # Perform an im2col operation by picking clever strides
  shape = (C, HH, WW, N, out_h, out_w)
  strides = (H * W, W, 1, C * H * W, stride * W, stride)
  strides = x.itemsize * np.array(strides)
  x_stride = np.lib.stride_tricks.as_strided(x_padded,
                shape=shape, strides=strides)
  x_cols = np.ascontiguousarray(x_stride)
  x_cols.shape = (C * HH * WW, N * out_h * out_w)

  # Now all our convolutions are a big matrix multiply
  res = w.reshape(F, -1).dot(x_cols) + b.reshape(-1, 1)

  # Reshape the output
  res.shape = (F, N, out_h, out_w)
  out = res.transpose(1, 0, 2, 3)

  # Be nice and return a contiguous array
  # The old version of conv_forward_fast doesn't do this, so for a fair
  # comparison we won't either
  out = np.ascontiguousarray(out)

  cache = (x, w, b, conv_param, x_cols)
  return out, cache
```

Case study:
fast_layers.py from HW

im2col

matrix multiply:
call np.dot
(which calls BLAS)

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

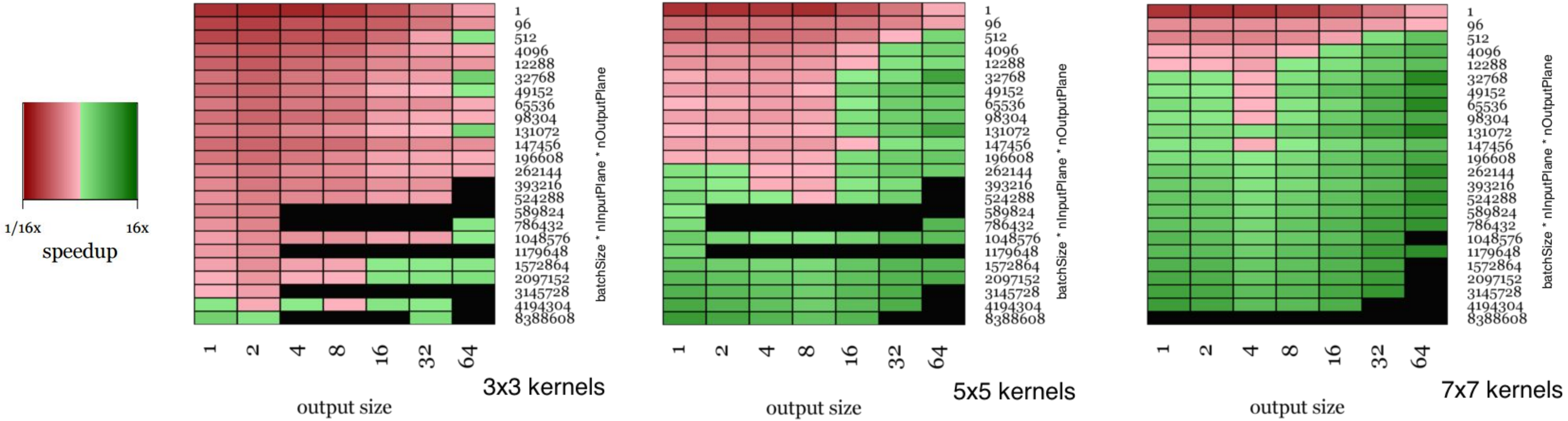
1. Compute FFT of weights: F(W)

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 $O(N^{\log2(7)}) \sim O(N^{2.81})$

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

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

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

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

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

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 msec	cuDNN TFLOPS	F(2x2,3x3) msec	F(2x2,3x3) TFLOPS	Speedup
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 msec	cuDNN TFLOPS	F(2x2,3x3) msec	F(2x2,3x3) TFLOPS	Speedup
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


ASUS X99-E WS
GEFORCE GTX
CORSAIR
AX1200i

Spot the CPU!

Spot the CPU!

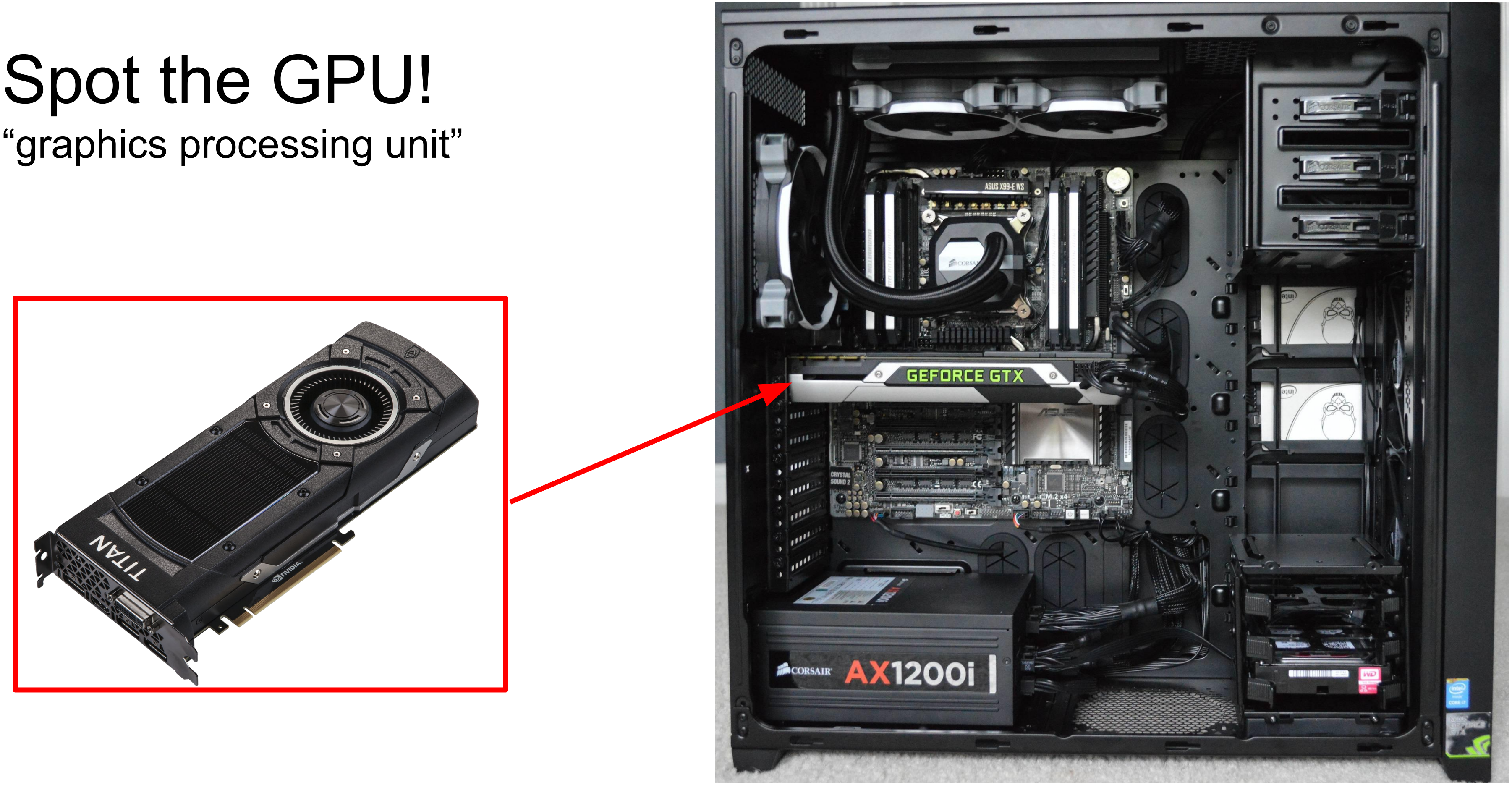
"central processing unit"

Spot the GPU!

"graphics processing unit"

Spot the GPU!

"graphics processing unit"



NVIDIA vs AMD

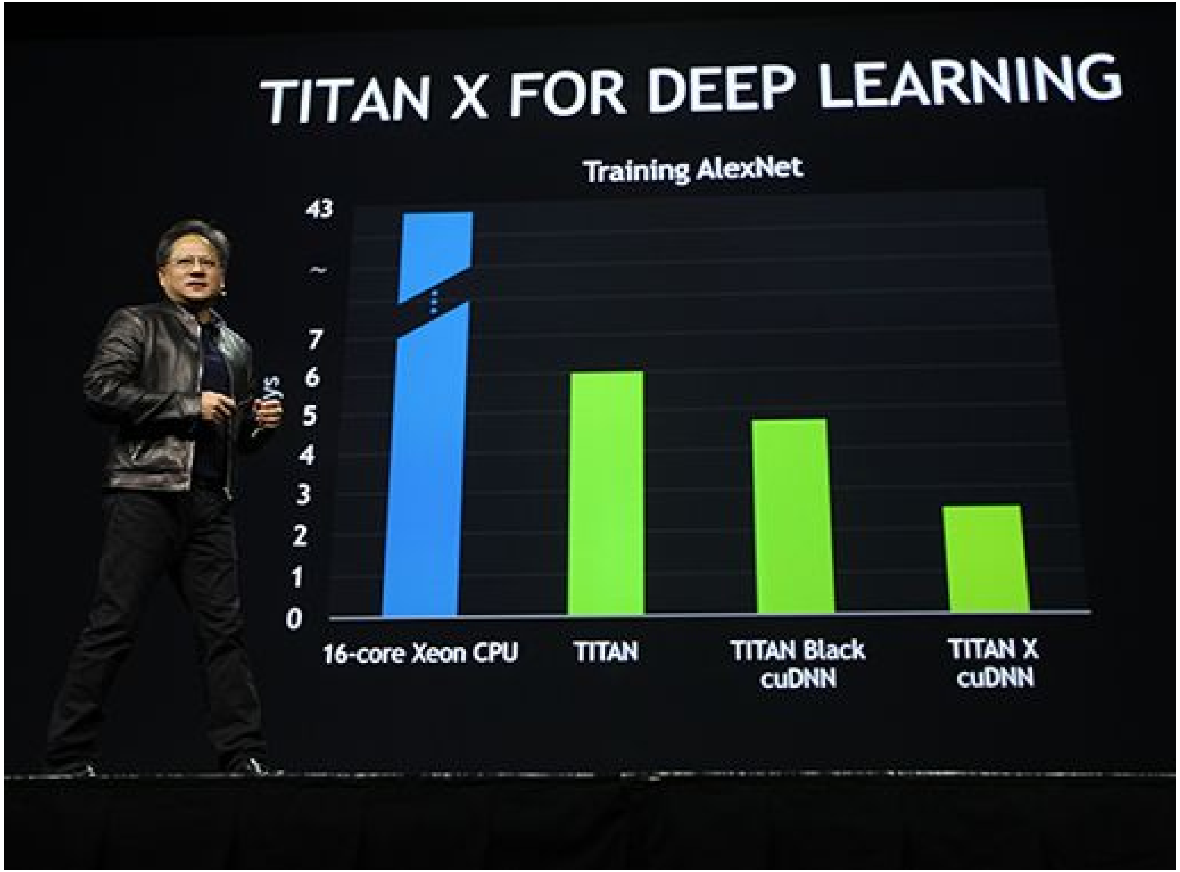
VS

NVIDIA is much more common for deep learning

CEO of NVIDIA:
Jen-Hsun Huang

(Stanford EE Masters 1992)

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



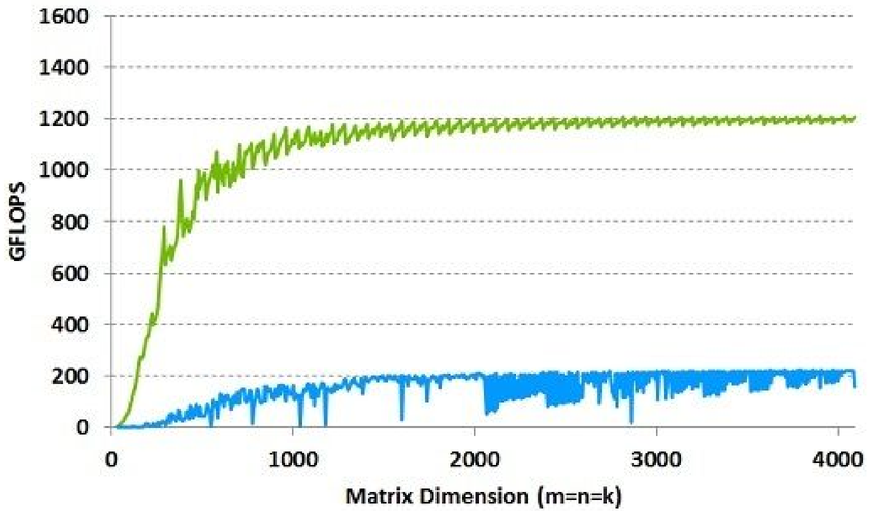
CPU
Few, fast cores (1 - 16)
Good at sequential processing

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

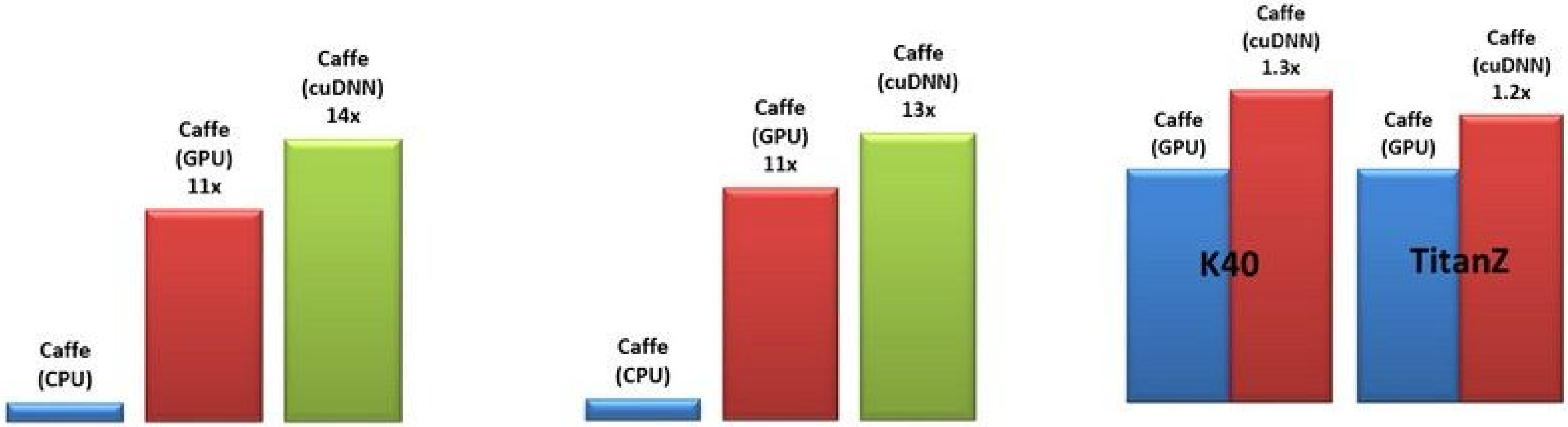
GPUs are really good at matrix multiplication:



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):

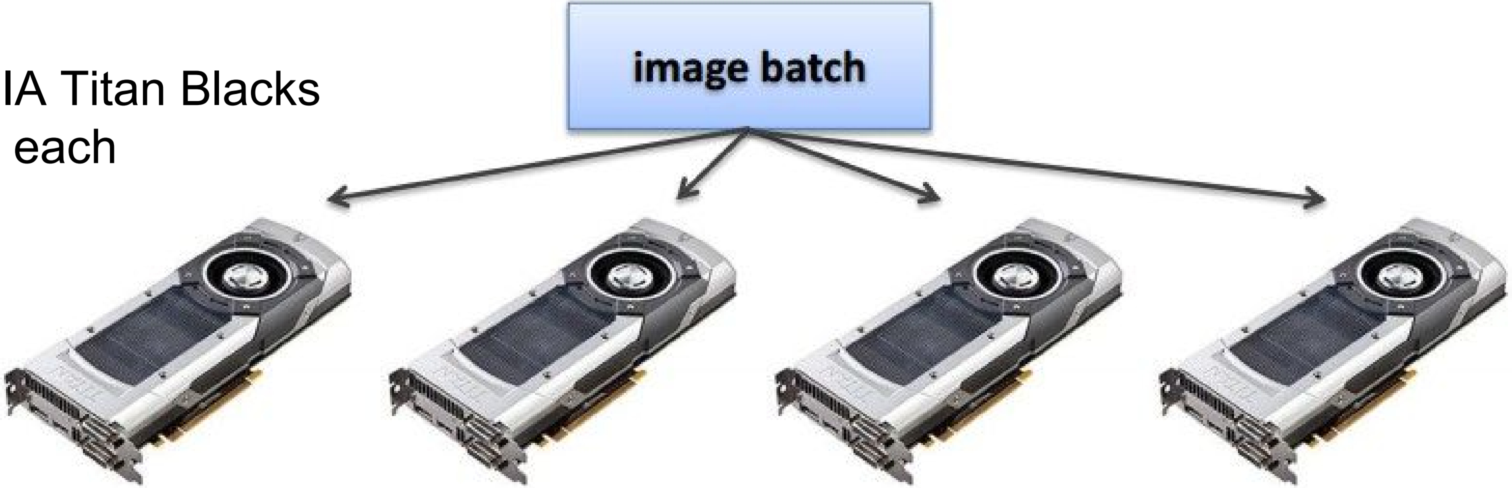


All comparisons are against a 12-core Intel E5-2679v2 CPU @ 2.4GHz running Caffe with Intel MKL 11.1.3.

Even with GPUs, training can be slow

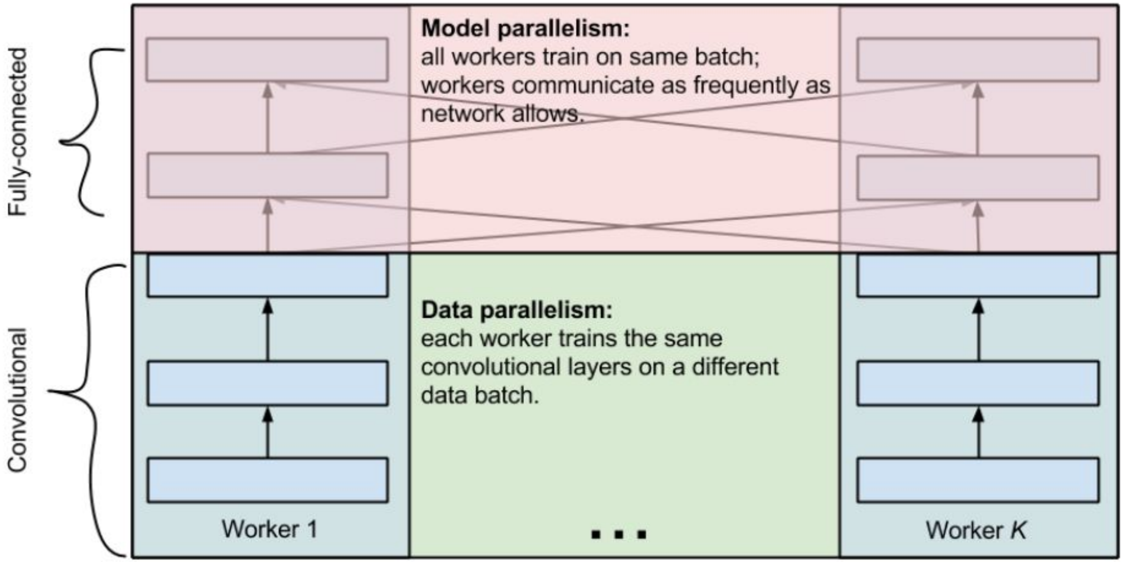
VGG: ~2-3 weeks training with 4 GPUs
ResNet 101: 2-3 weeks with 4 GPUs

NVIDIA Titan Blacks
~$1K each



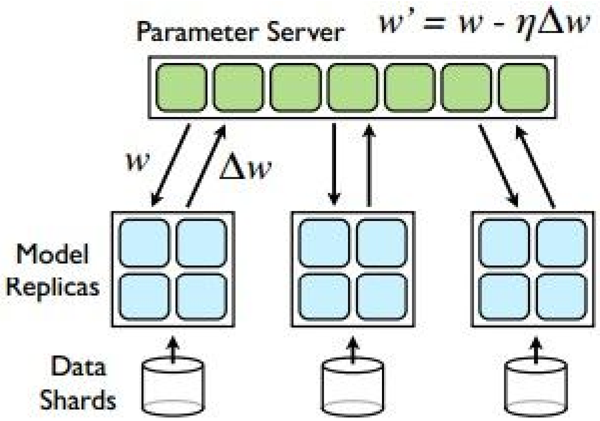
ResNet reimplemented in Torch: http://torch.ch/blog/2016/02/04/resnets.html

Multi-GPU training: More complex



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

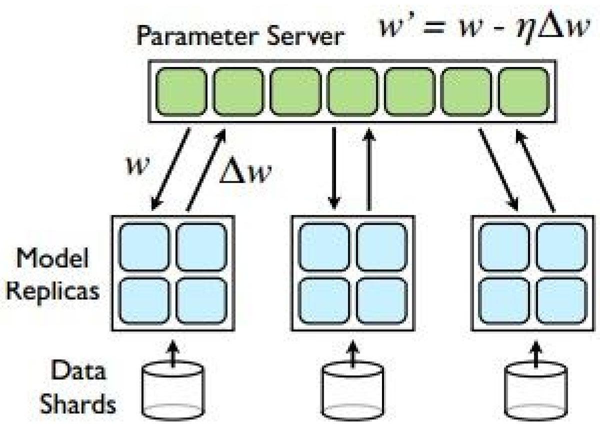
Google: Distributed CPU training



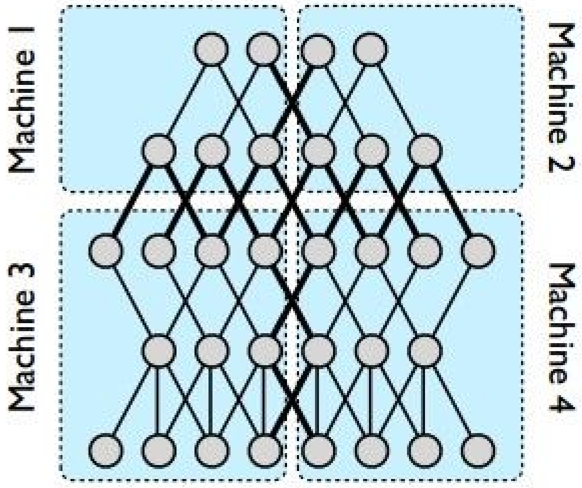
Data parallelism

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

Google: Distributed CPU training



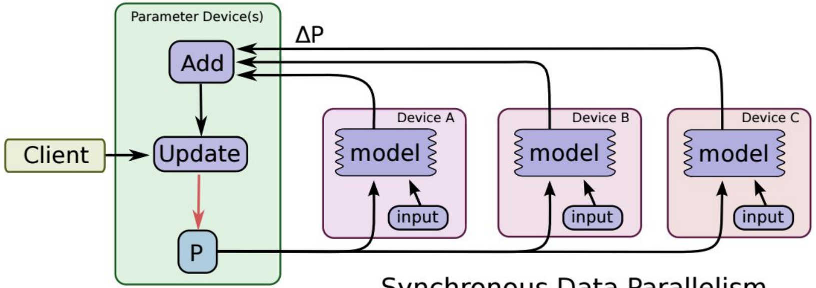
Data parallelism



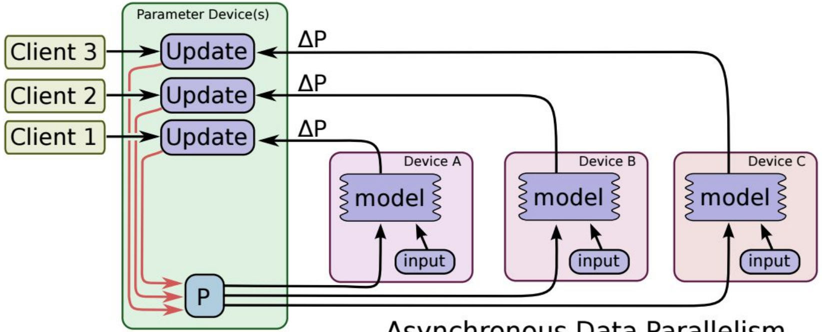
Model parallelism

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

Google: Synchronous vs Async



Synchronous Data Parallelism



Asynchronous Data Parallelism

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

Moving parts lol

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

Floating Point Precision

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

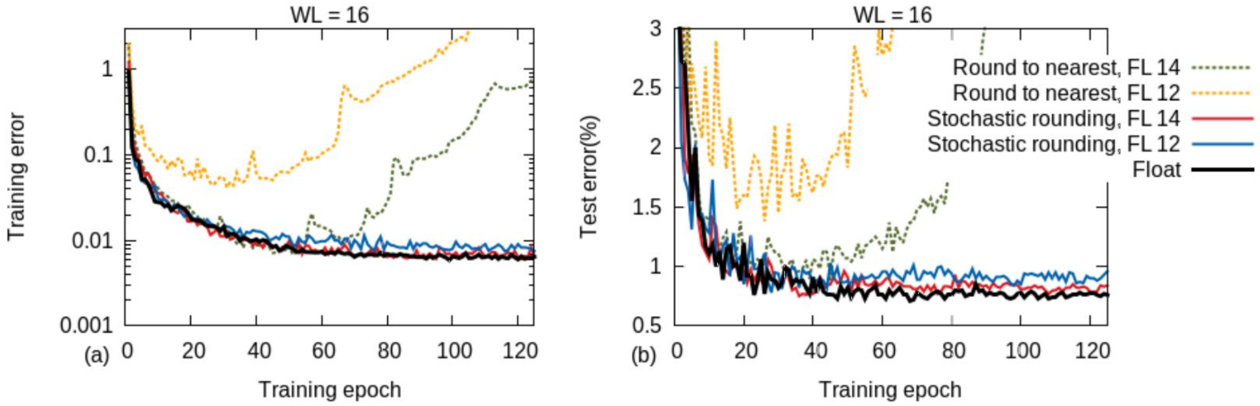
GoogleNet V1 - Input 128x3x224x224

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?

Recap

- Data augmentation: artificially expand your data
- Transfer learning: CNNs without huge data
- All about convolutions
- Implementation details