Administrative

- In-class midterm this Wednesday! (More on this in a bit)
- Assignment #3: out Wed
- Sample Midterm will be up in few hours

Lecture 10:

Squeezing out the last few percent

&

Training ConvNets in practice

Midterm during next class!

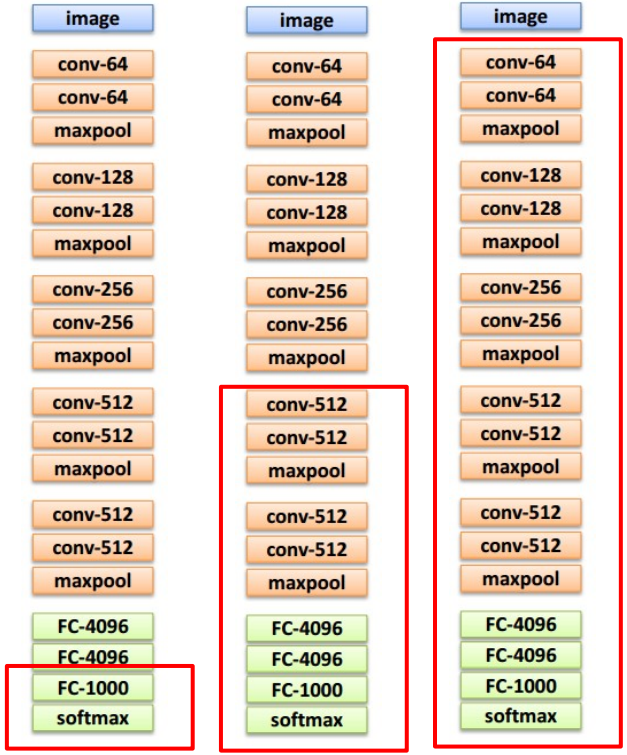
- Everything in the **notes** (unless labeled as **aside**) is fair game.
- Everything in the **slides** (until and including last lecture) is fair game.
- Everything in the **assignments** is fair game.
- There will be **no Python/numpy/vectorization** questions.
- There will be no questions that require you to know specific details of covered papers, but takeaways presented in class are fair game.

What it does include:

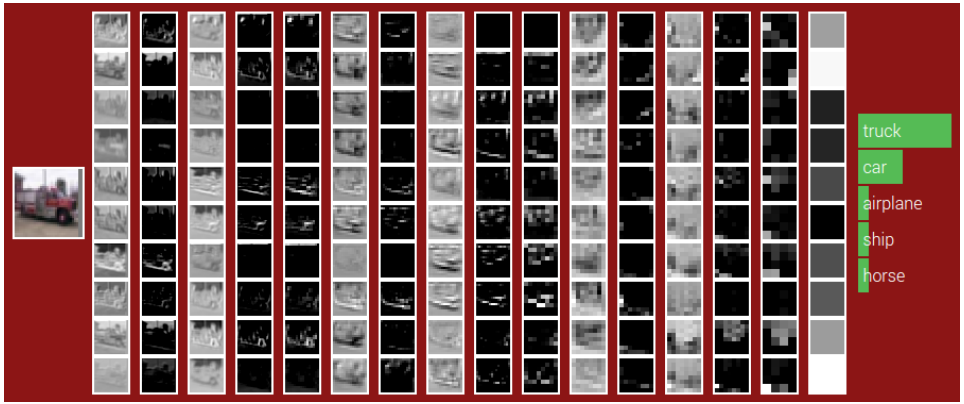
- Conceptual/Understanding questions (e.g. likes ones I like to ask during lectures)
- Design/Tips&Tricks/Debugging questions and intuitions
- Know your Calculus

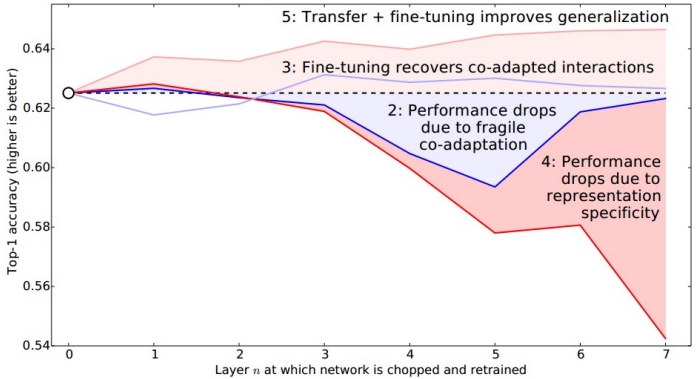
Where we are...

Transfer Learning



ConvNets





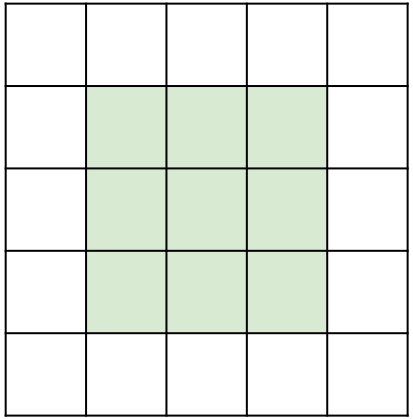
Bit more about small filters

The power of small filters

(and stride 1)

Suppose we stack two CONV layers with receptive field size 3x3
=> Each neuron in 1st CONV sees a 3x3 region of input.

1st CONV neuron view of the input:

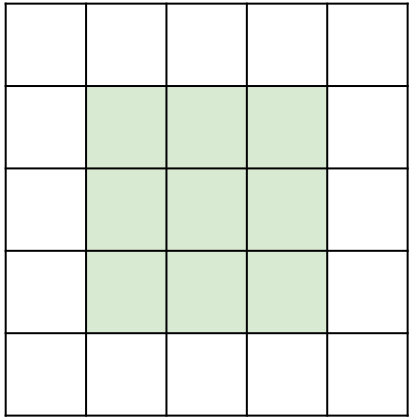


The power of small filters

Suppose we stack two CONV layers with receptive field size 3x3
=> Each neuron in 1st CONV sees a 3x3 region of input.

Q: What region of input does each neuron in 2nd CONV see?

2nd CONV neuron view of 1st conv:

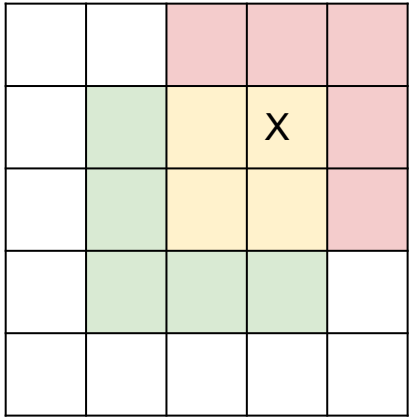


The power of small filters

Suppose we stack two CONV layers with receptive field size 3x3
=> Each neuron in 1st CONV sees a 3x3 region of input.

Q: What region of input does each neuron in 2nd CONV see?

2nd CONV neuron view of input:

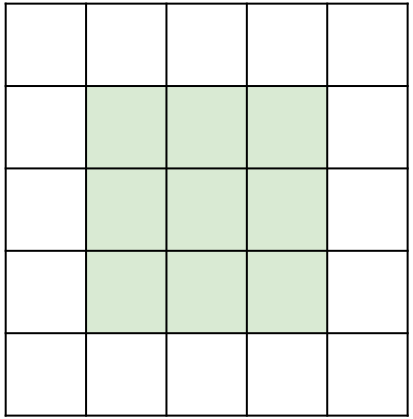


Answer: [5x5]

The power of small filters

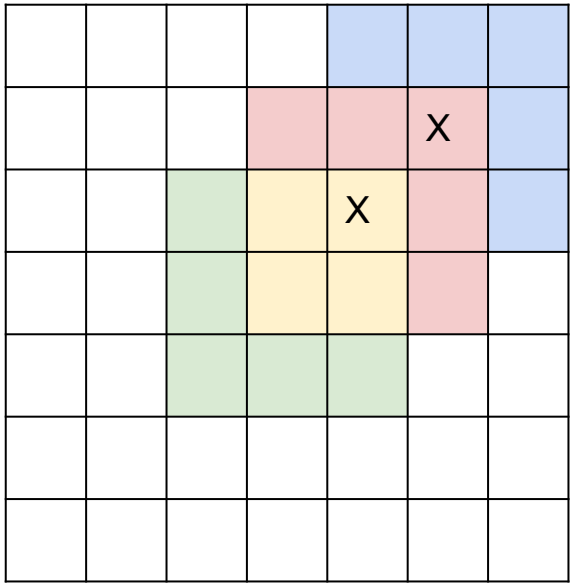
Suppose we stack **three** CONV layers with receptive field size 3x3
Q: What region of input does each neuron in 3rd CONV see?

3rd CONV neuron view of 2nd CONV:



The power of small filters

Suppose we stack **three** CONV layers with receptive field size 3x3
Q: What region of input does each neuron in 3rd CONV see?



Answer: [7x7]

The power of small filters

Suppose input has depth C & we want output depth C as well

1x CONV with 7x7 filters

Number of weights:

3x CONV with 3x3 filters

Number of weights:

The power of small filters

Suppose input has depth C & we want output depth C as well

1x CONV with 7x7 filters

Number of weights:

C*(7*7*C)
= 49 C^2

3x CONV with 3x3 filters

Number of weights:

The power of small filters

Suppose input has depth C & we want output depth C as well

1x CONV with 7x7 filters

Number of weights:

C*(7*7*C)
= **49 C^2**

3x CONV with 3x3 filters

Number of weights:

C*(3*3*C) + C*(3*3*C) + C*(3*3*C)
= 3 * 9 * C^2
= **27 C^2**

The power of small filters

Suppose input has depth C & we want output depth C as well

1x CONV with 7x7 filters

Number of weights:

C*(7*7*C)
= **49 C^2**

3x CONV with 3x3 filters

Number of weights:

C*(3*3*C) + C*(3*3*C) + C*(3*3*C)
= 3 * 9 * C^2
= **27 C^2**

Fewer parameters and more nonlinearities = GOOD.

The power of small filters

“More non-linearities” and “deeper” usually gives better performance.

[Network in Network, Lin et al. 2013]

The power of small filters

“More non-linearities” and “deeper” usually gives better performance.

=> 1x1 CONV!

(Usually follows a normal CONV, e.g.
[3x3 CONV - 1x1 CONV]

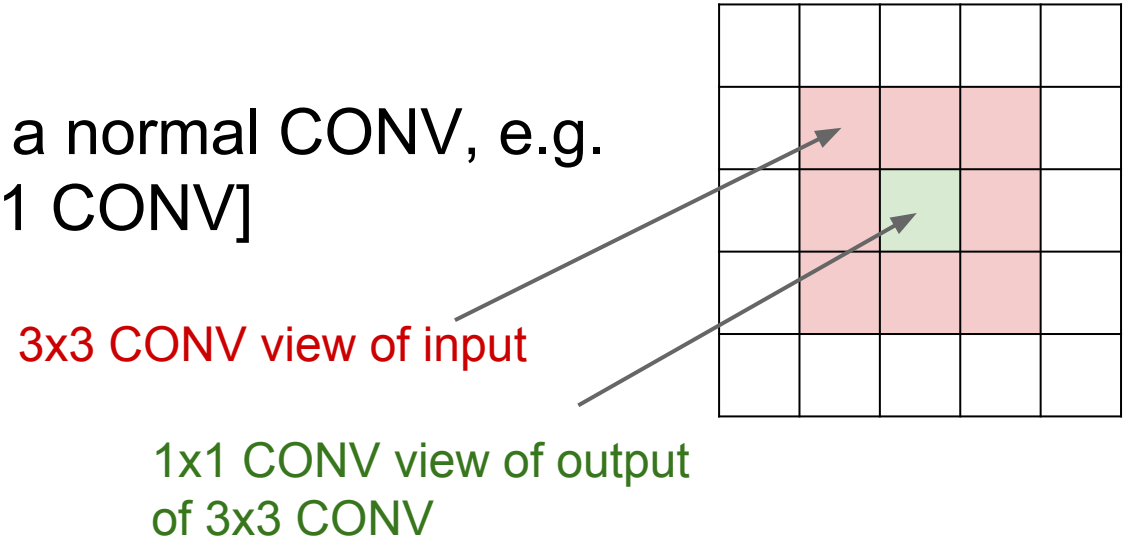
[Network in Network, Lin et al. 2013]

The power of small filters

"More non-linearities" and "deeper" usually gives better performance.

=> 1x1 CONV!

(Usually follows a normal CONV, e.g. [3x3 CONV - 1x1 CONV]



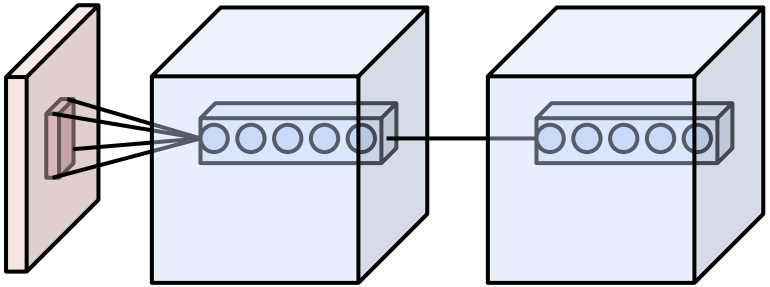
[Network in Network, Lin et al. 2013]

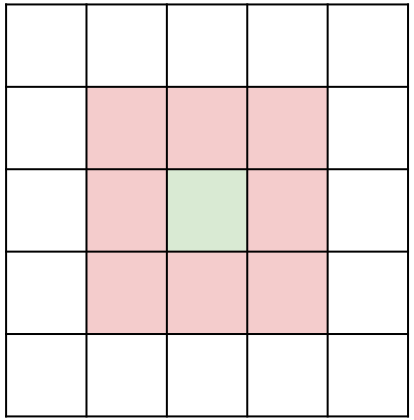
The power of small filters

"More non-linearities" and "deeper" usually gives better performance.

=> 1x1 CONV!

(Usually follows a normal CONV, e.g. [3x3 CONV - 1x1 CONV]





[Network in Network, Lin et al. 2013]

[Very Deep Convolutional Networks for Large-Scale Image Recognition, Simonyan et al., 2014]

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

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

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

ConvNet config. (Table 1)	smallest image side		top-1 val. error (%)	top-5 val. error (%)
	train (S)	test (Q)		
A	256	256	29.6	10.4
A-LRN	256	256	29.7	10.5
B	256	256	28.7	9.9
C	256	256	28.1	9.4
	384	384	28.1	9.3
	[256;512]	384	27.3	8.8
D	256	256	27.0	8.8
	384	384	26.8	8.7
	[256;512]	384	25.6	8.1
E	256	256	27.3	9.0
	384	384	26.9	8.7
	[256;512]	384	**25.5**	**8.0**

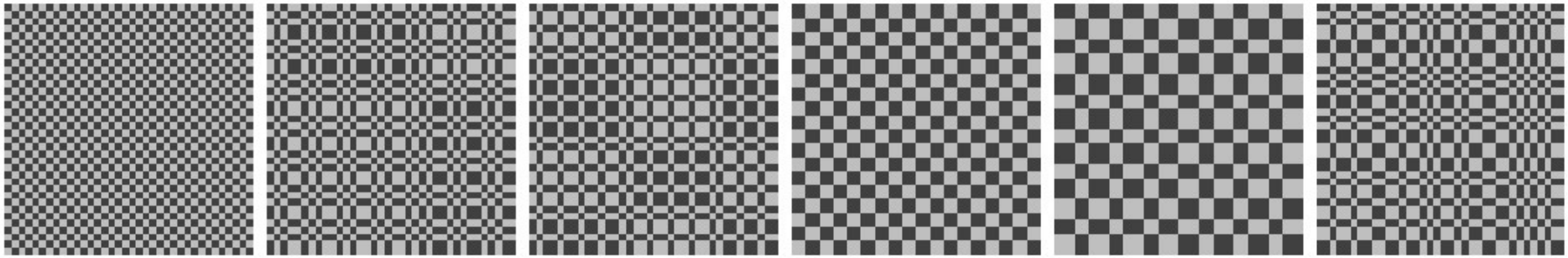
=> Evidence that using 3x3 instead of 1x1 works better

The power of small filters

[Fractional max-pooling, Ben Graham, 2014]

The power of small filters

[Fractional max-pooling, Ben Graham, 2014]



In ordinary 2x2 maxpool, the pooling regions are non-overlapping 2x2 squares

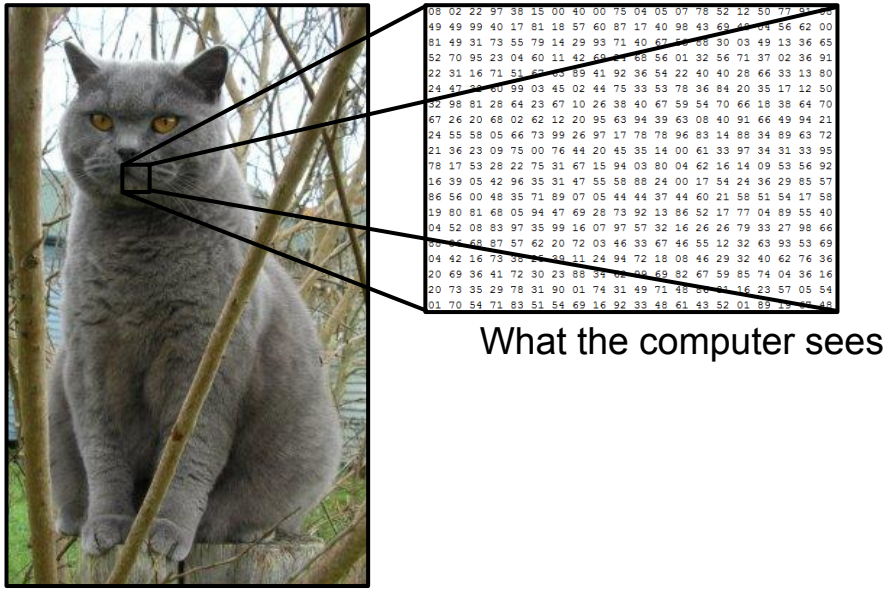
Fractional pooling samples pooling region during forward pass: A mix of 1x1, 2x1, 1x2, 2x2.

Data Augmentation

Data Augmentation

- i.e. simulating “fake” data

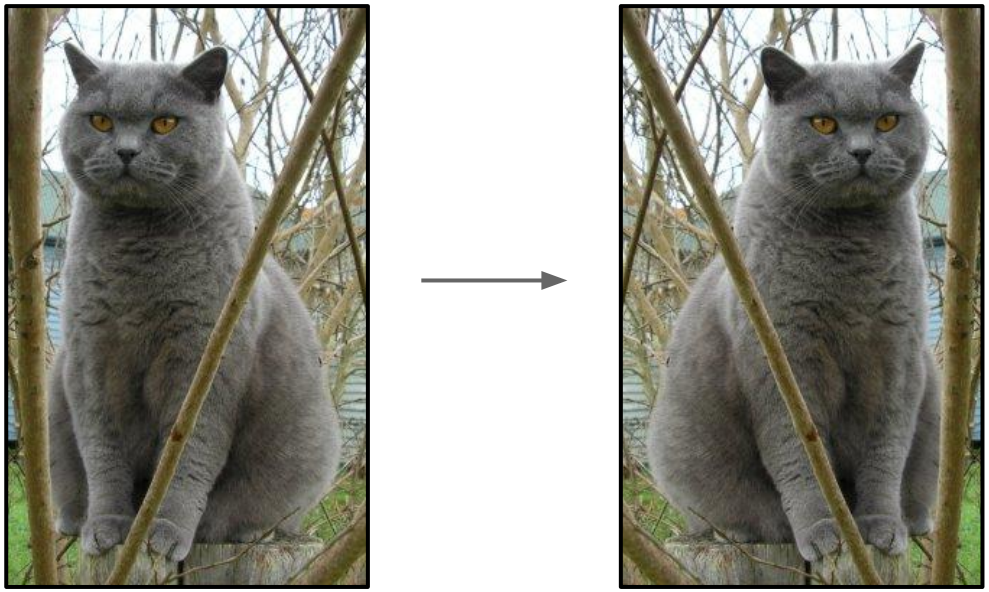
- explicitly encoding image transformations that shouldn’t change object identity.



What the computer sees

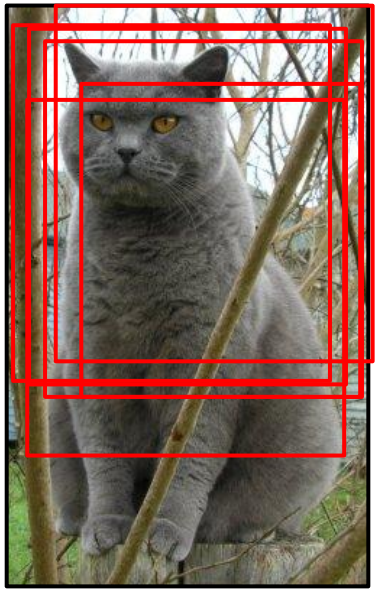
Data Augmentation

1. Flip horizontally



Data Augmentation

2. Random crops/scales



Sample these during training
(also helps a lot during test time)

e.g. common to see even up to 150 crops used

Data Augmentation

3.

Random mix/combinations of :

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

Data Augmentation

4. Color jittering

(maybe even contrast jittering, etc.)

- Simple: Change contrast small amounts, jitter the color distributions, etc.

- Vignette,... (go crazy)

Data Augmentation

4. Color jittering

(maybe even contrast jittering, etc.)

- Simple: Change contrast small amounts, jitter the color distributions, etc.

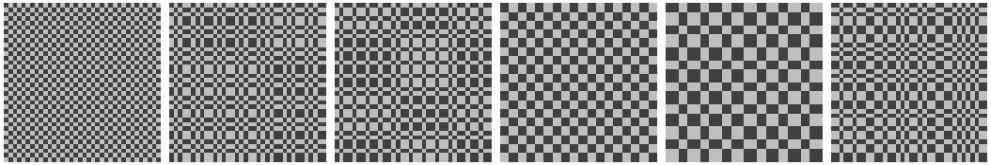
Fancy PCA way:

1. Compute PCA on all [R,G, B] points values in the training data
2. sample some color offset along the principal components at each forward pass
3. add the offset to all pixels in a training image

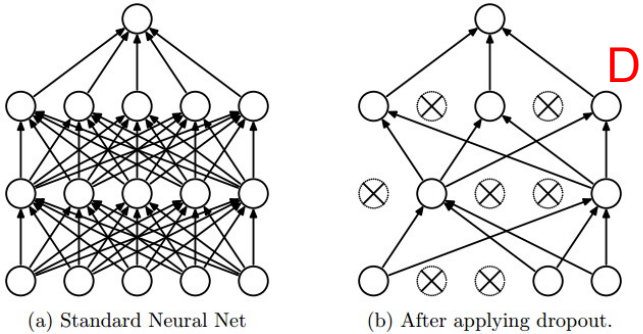
(As seen in *[Krizhevsky et al. 2012]*)

Notice the more general theme:

1. Introduce a form of randomness in forward pass
2. Marginalize over the noise distribution during prediction

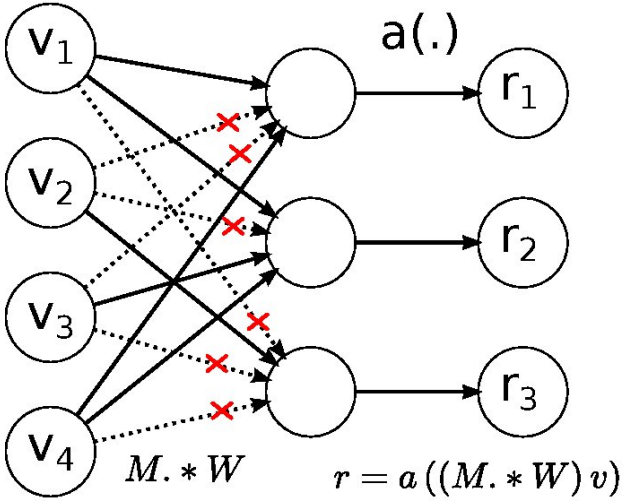


Fractional Pooling



Dropout

Data Augmentation,
Model Ensembles

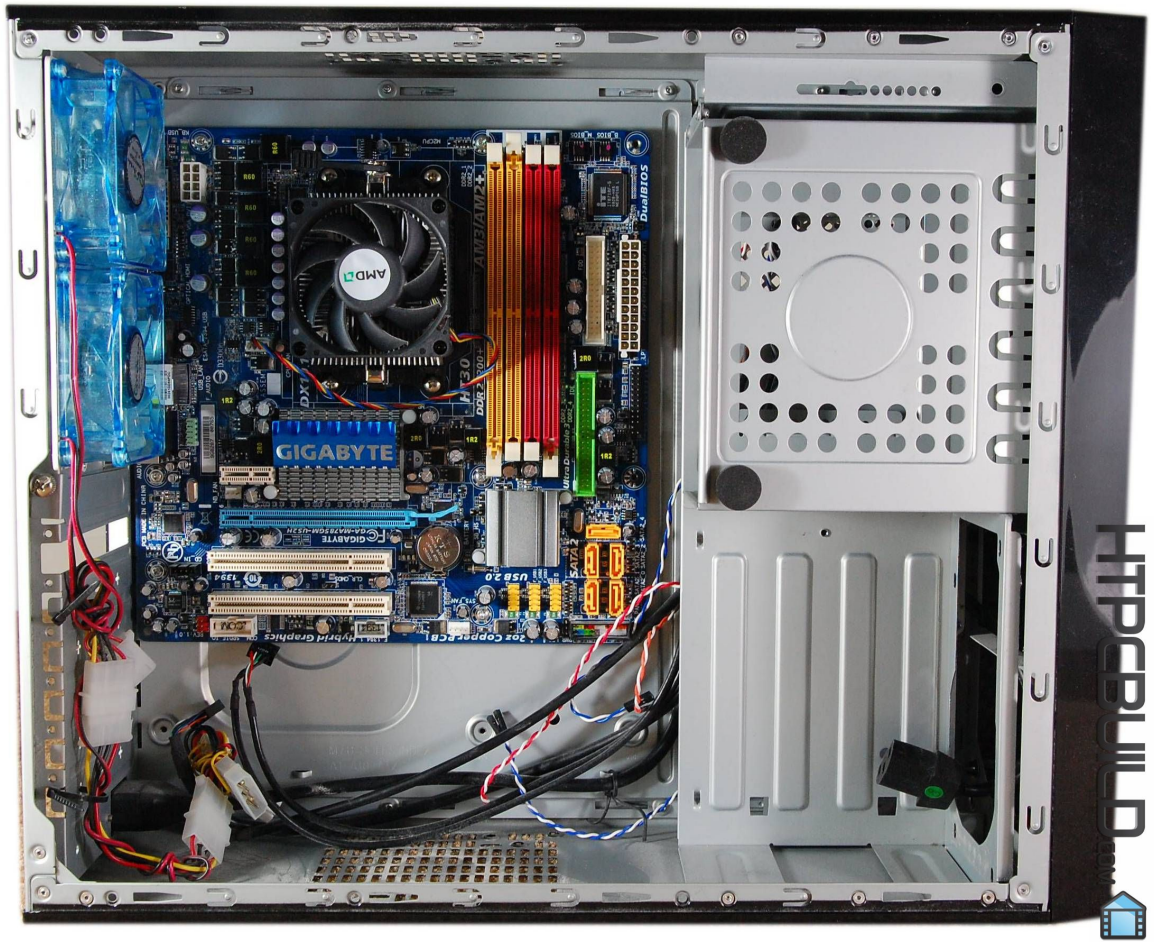


DropConnect

Training ConvNets in Practice

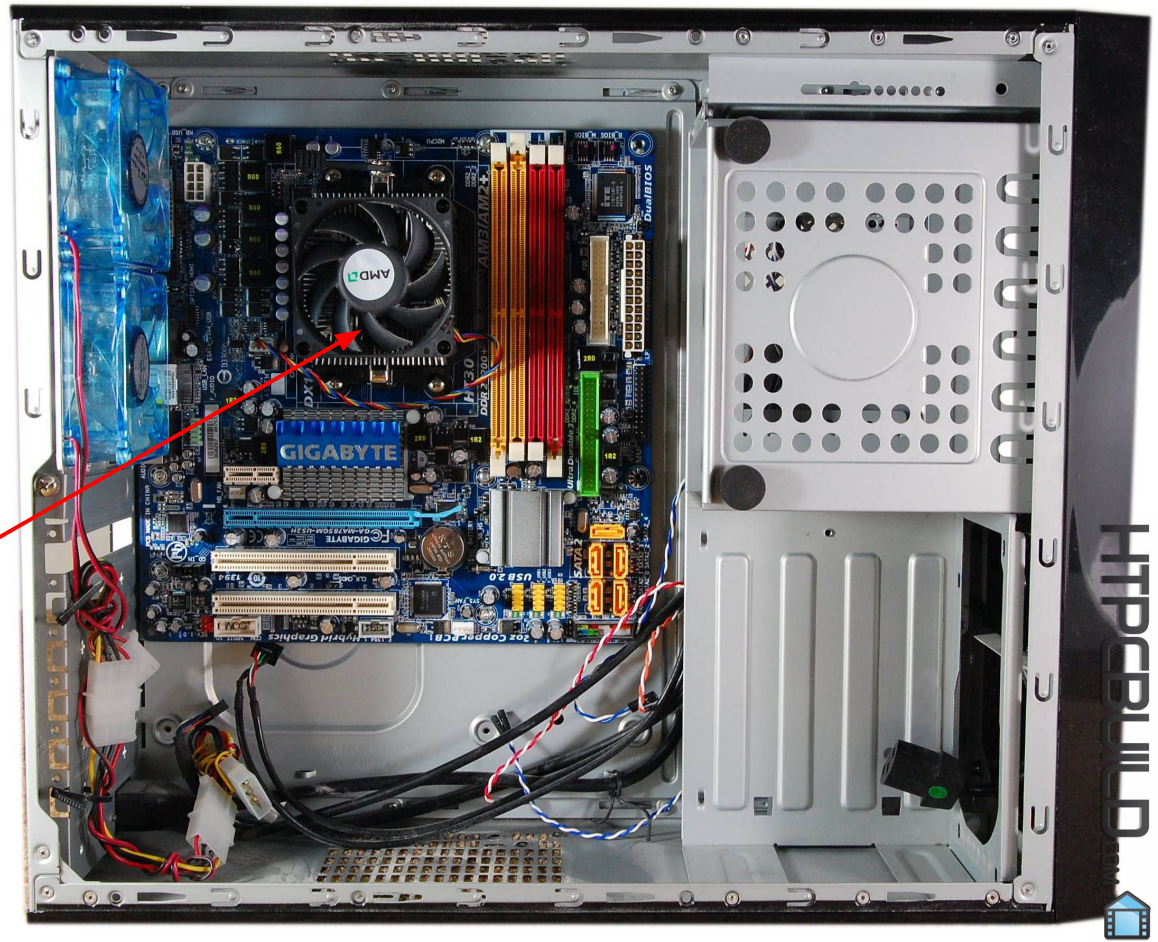

GIGABYTE
AMD
HTPCBUILD.com

Spot the CPU!



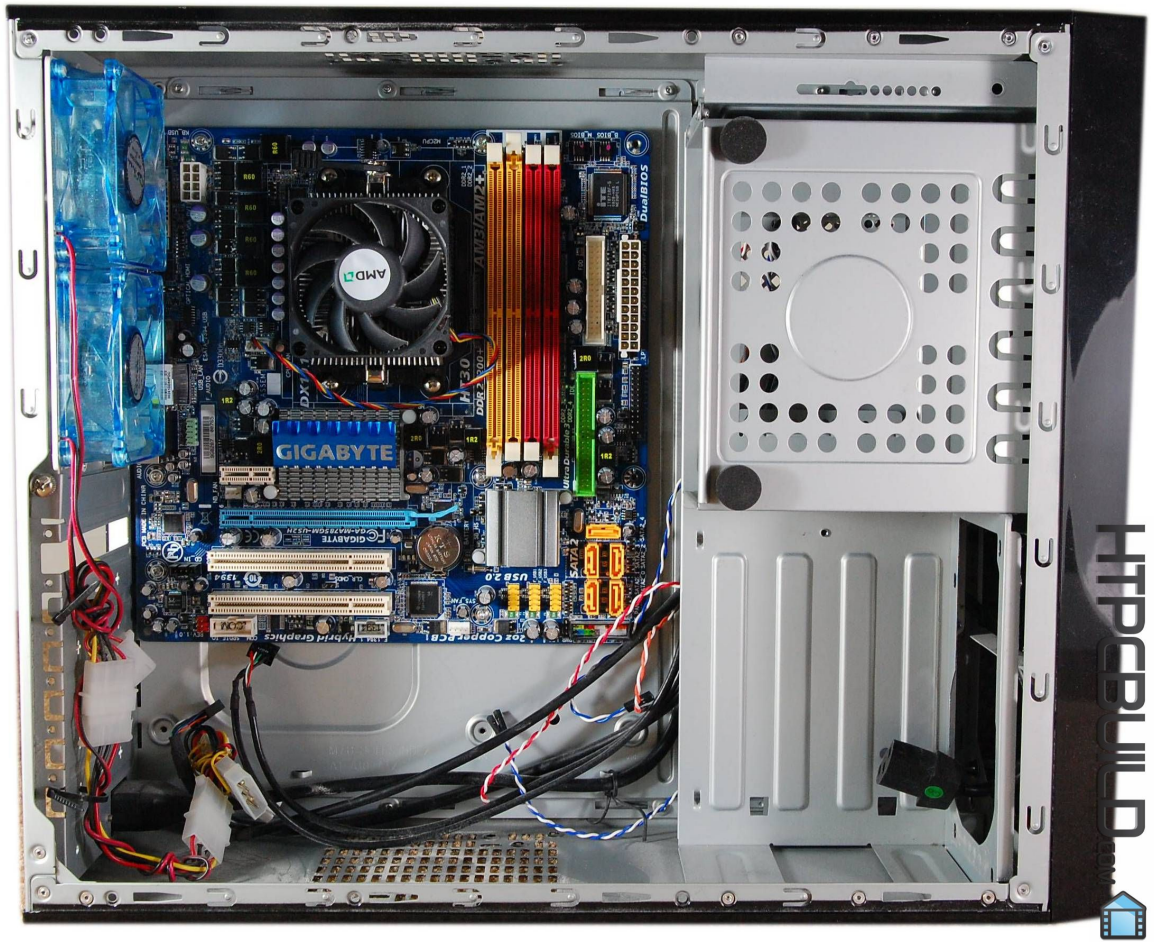
Spot the CPU!

"central processing unit"



Spot the GPU!

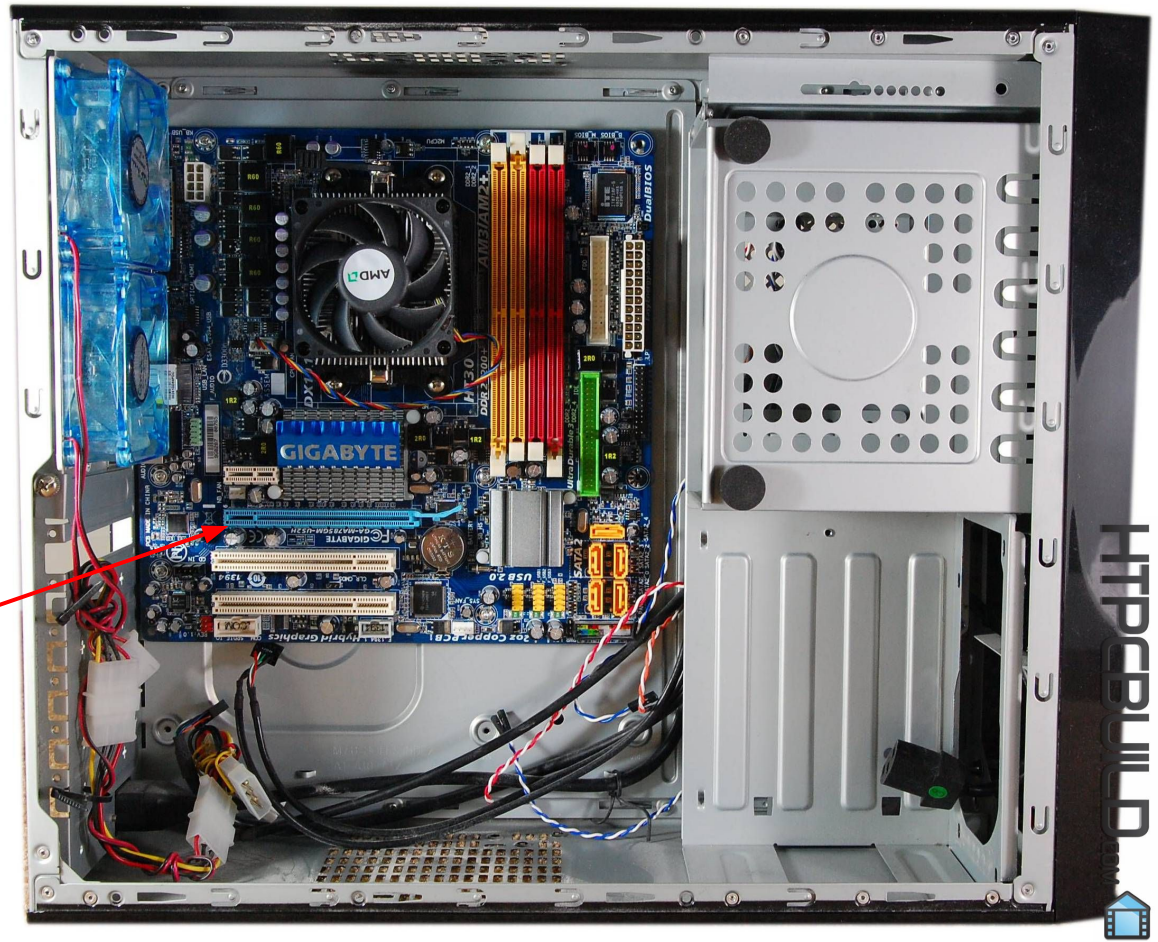
"graphics processing unit"



Spot the GPU!

"graphics processing unit"

plugs in to PCI express slot



CEO of NVIDIA:
Jen-Hsun Huang

(**Stanford** Master's degree in EE from 1992 by the way)

GPUs

are very good at local, parallel operations

e.g. in rendering

GPUs can be programmed:

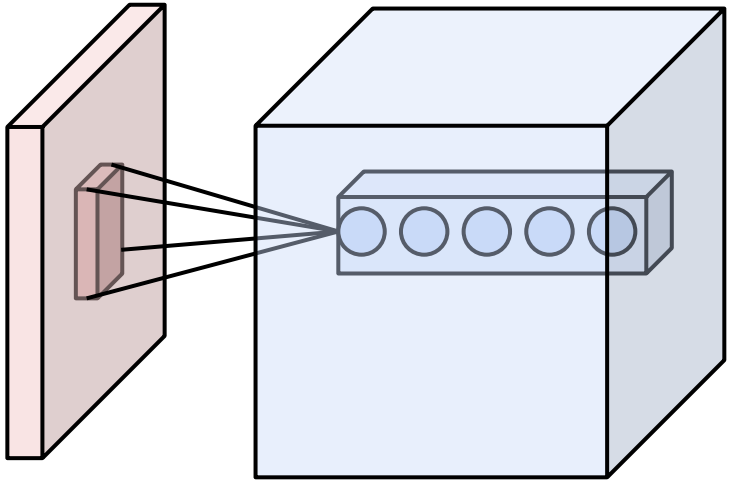
- CUDA
- + higher-level API (e.g. cuBLAS, cuDNN)

Resources:

- Interview with Dan Ciresan
http://www.nvidia.com/content/cuda/spotlights/dan-ciresan-idsia.html
- CUDA@MIT https://sites.google.com/site/cudaiap2009/
- Intro to Parallel Programming on Udacity https://www.udacity.com/course/cs344

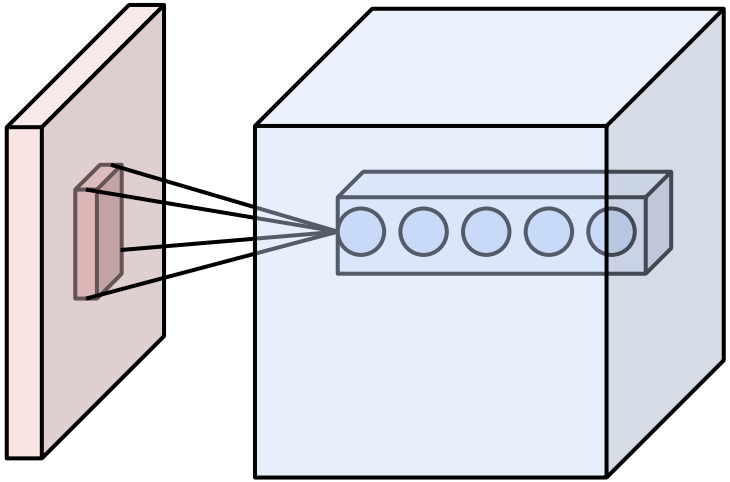
Convolutional Neural Networks

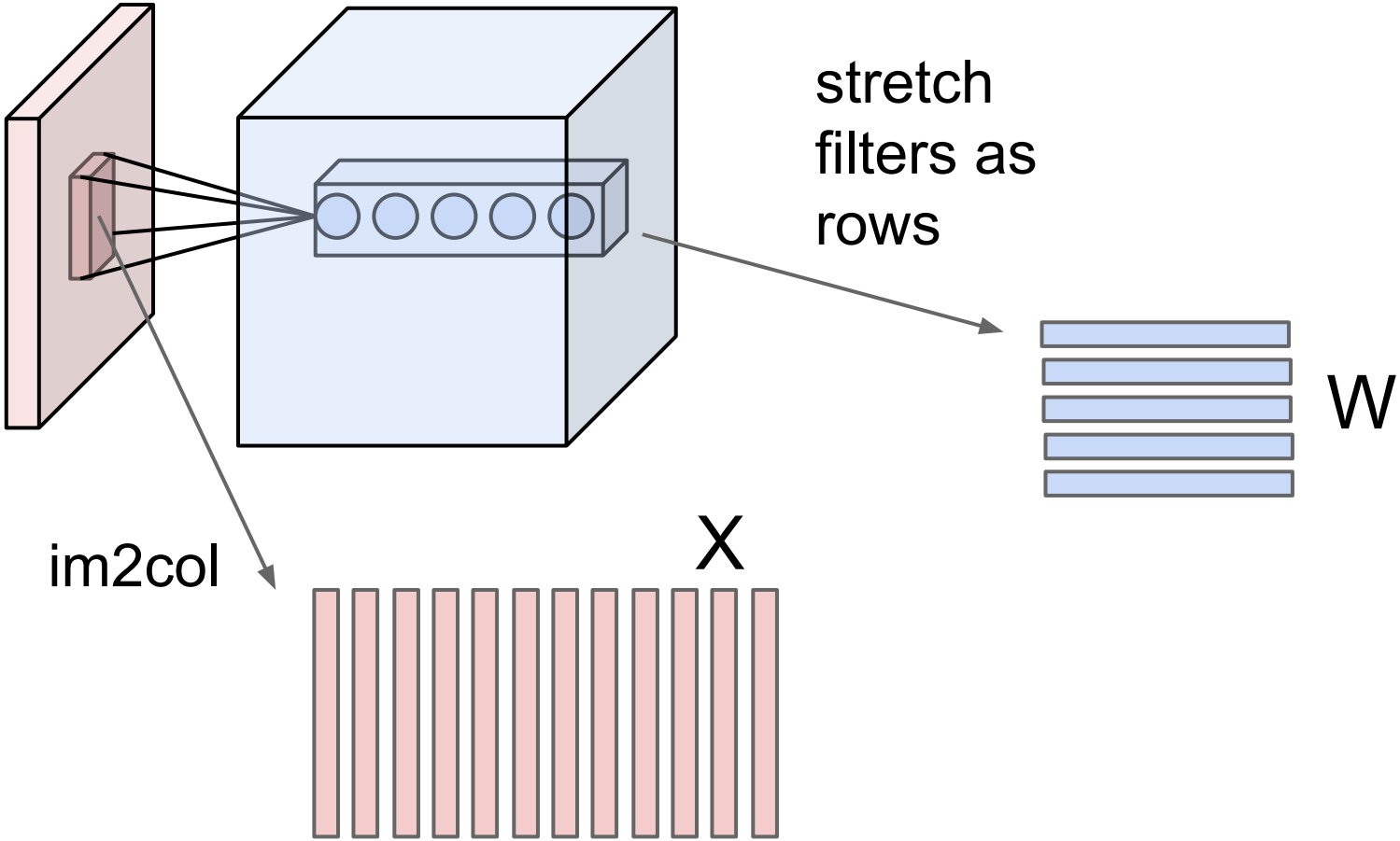
- Basically perfect for GPUs

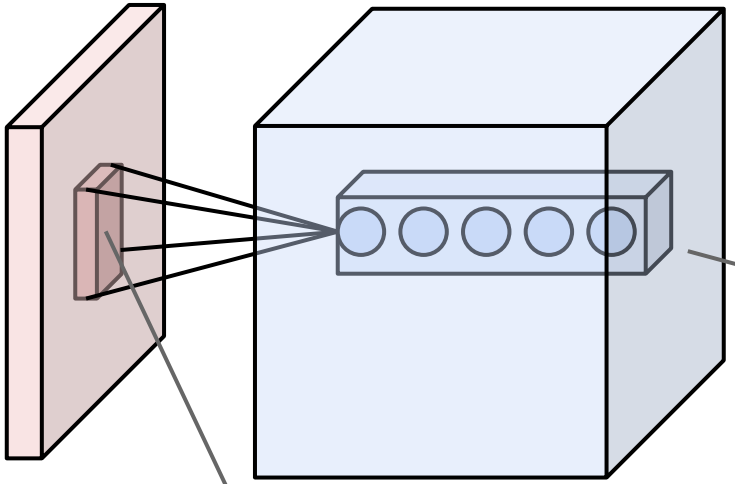


```
template <typename Dtype>
void ConvolutionLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      vector<Blob<Dtype>*>* top) {
  for (int i = 0; i < bottom.size(); ++i) {
    const Dtype* bottom_data = bottom[i]->gpu_data();
    Dtype* top_data = (*top)[i]->mutable_gpu_data();
    Dtype* 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




stretch
filters as
rows
W
im2col
X

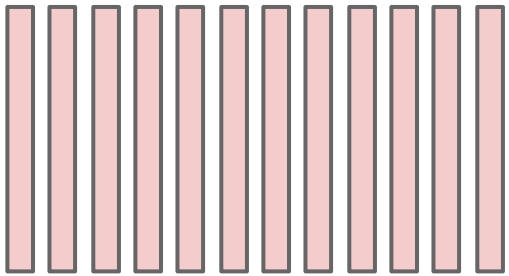


stretch
filters as
rows

W

im2col

X



matrix multiply W*X,
then reshape back into
a volume

```
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));
      }
    }
  }
}
```

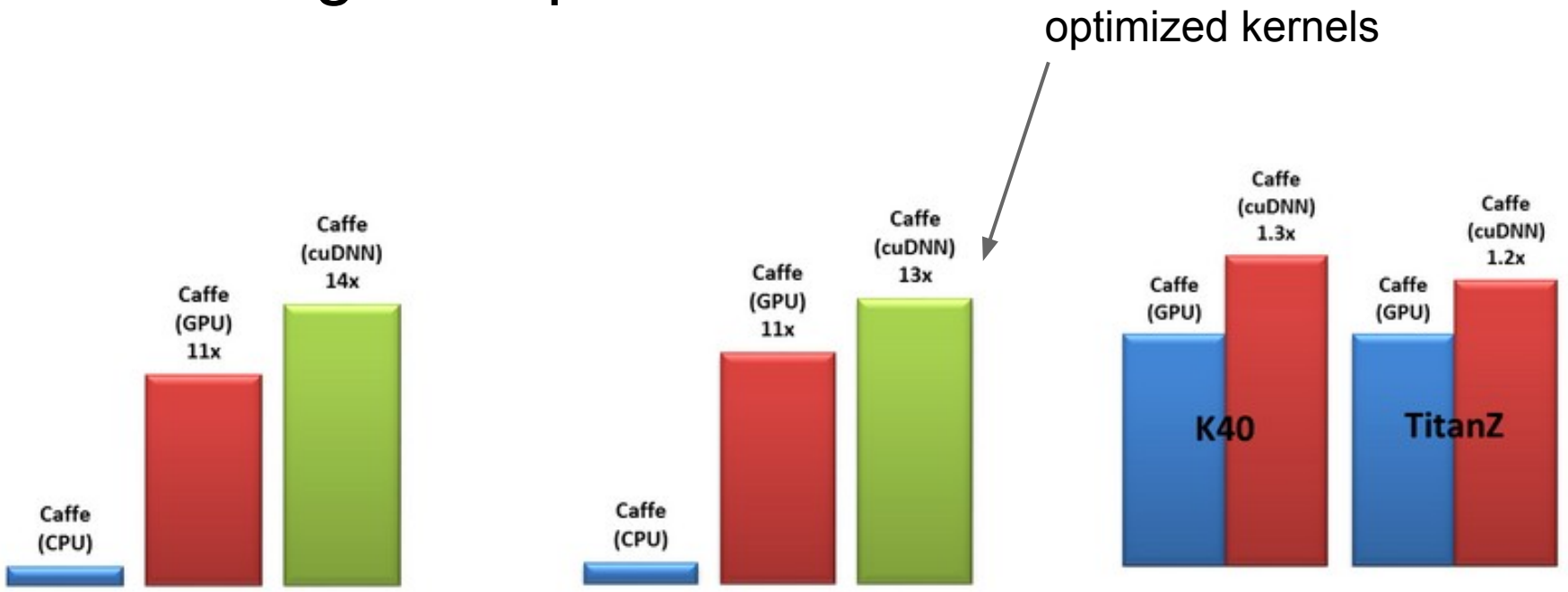
Case study: **CONV forward in Caffe library**

im2col

matrix multiply: call to cuBLAS

bias offset

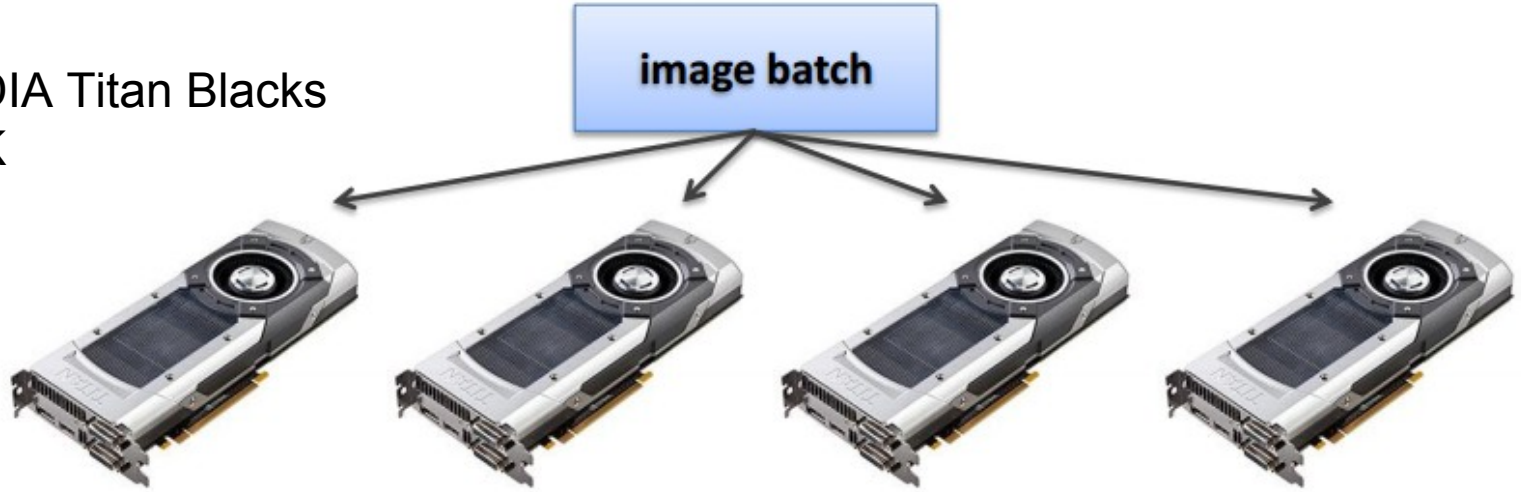
GPU timings comparison:



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

E.g. **VGG net:**
~2-3 weeks training with 4 GPUs

NVIDIA Titan Blacks
~$1K



Speeding up Convolutions with FFT

"The Fourier transform of a convolution of two functions is the product of the Fourier transforms of those functions" - **convolution theorem**

1. Transform input, filters with FFT
2. Perform elementwise product
3. Inverse FFT the result back to original domain

See e.g. *[Fast Convolutional Nets With fbfft: A GPU Performance Evaluation]*

Speeding up Convolutions with FFT

"The Fourier transform of a convolution of two functions is the product of the Fourier transforms of those functions" - **convolution theorem**

Unfortunately, FFT Conv is slower with smaller filter sizes :((backwards!)

1. Transform input, filters with FFT
2. Perform elementwise product
3. Inverse FFT the result back to original domain

See e.g. *[Fast Convolutional Nets With fbfft: A GPU Performance Evaluation]*

Bottlenecks

to be aware of

GPU - CPU communication is a bottleneck.

=>

CPU data prefetch thread running

while

GPU performs forward/backward pass

CPU - disk bottleneck

Harddisk is slow to read from

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

GPU memory bottleneck

Tesla K40: 12GB <- currently the max
Titan Black: 6GB

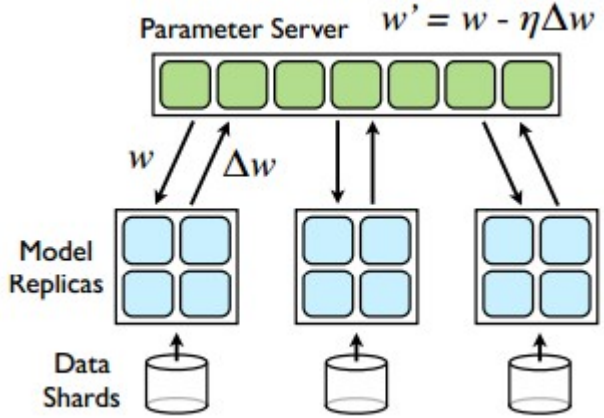
e.g.
AlexNet: ~3GB needed with batch size 256

Caffe typical numbers:

- Can train AlexNet on about **40M images / day** with an NVIDIA K40 or Titan GPU

~ 5ms/image for forward/backward/update
(or **~2ms/image for forward**)

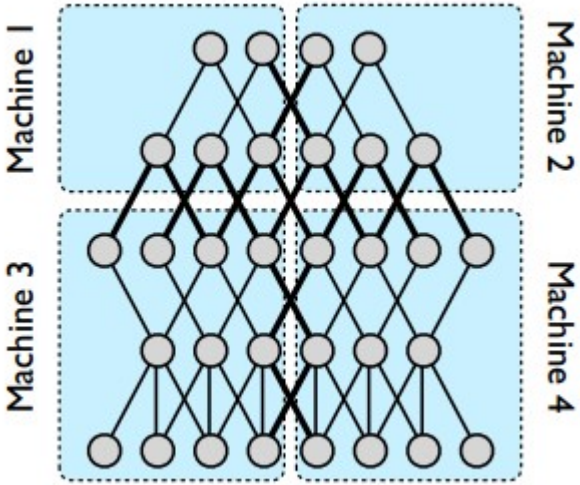
Google: Pushing CPU to the limit



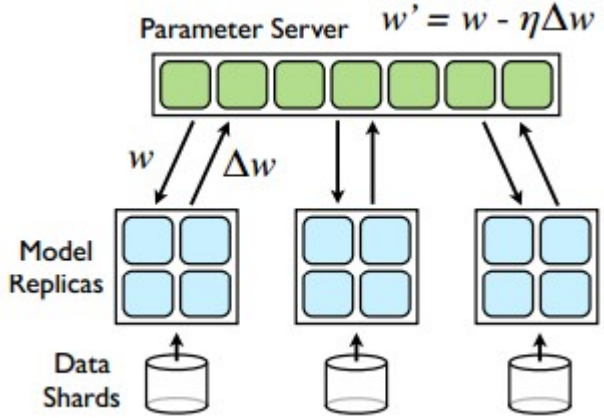
Data parallelism

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

Google: Pushing CPU to the limit



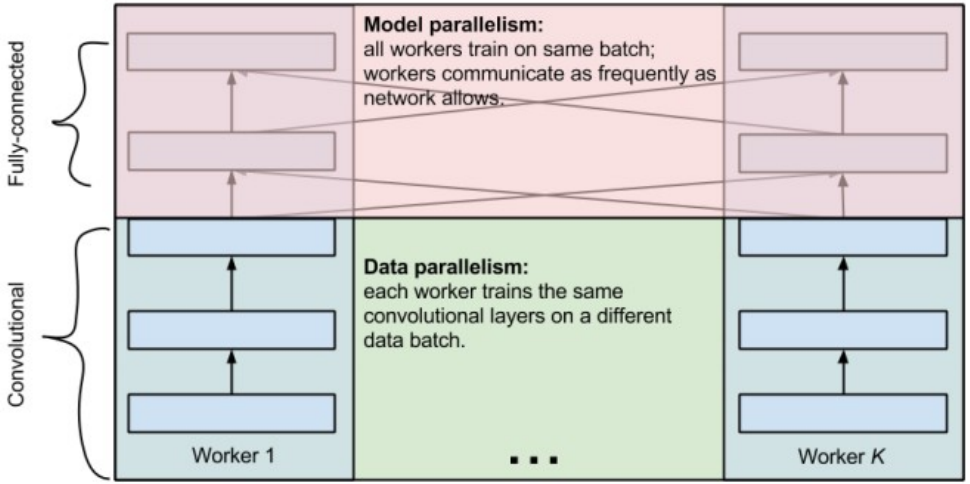
Model parallelism



Data parallelism

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

Multi-GPU training
E.g. **cuda-convnet2**



Observation:

- Conv Layers contain 90-95% compute but only about 5% parameters
- FC Layers contain 5-10% compute but 95% parameters

[One weird trick for parallelizing convolutional neural networks, Krizhevsky 2014]
also see: *[Deep learning with COTS HPC systems, Coates et al. 2013]*

When Computer Vision papers start to look like Systems papers...

The result is the custom-built supercomputer, which we call Minwa . It is comprised of 36 server nodes, each with 2 six-core Intel Xeon E5-2620 processors. Each sever contains 4 Nvidia Tesla K40m GPUs and one FDR InfiniBand (56Gb/s) which is a high-performance low-latency interconnection and supports RDMA. The peak single precision floating point performance of each GPU is 4.29TFlops and each GPU has 12GB of memory. Thanks to the GPUDirect RDMA, the InfiniBand network interface can access the remote GPU memory without involvement from the CPU. All the server nodes are connected to the InfiniBand switch. Figure 1 shows the system architecture. The system runs Linux with CUDA 6.0 and MPI MVAPICH2, which also enables GPUDirect RDMA.

In total, Minwa has 6.9TB host memory, 1.7TB device memory, and about 0.6PFlops theoretical single precision peak performance.

[Deep Image: Scaling up Image Recognition, Wu Ren et al. 2015] (Baidu)

When Computer Vision papers start to look like Systems papers...

The result is the custom-built supercomputer, which we call Minwa . It is comprised of 36 server nodes, each with 2 six-core Intel Xeon E5-2620 processors. Each sever contains 4 Nvidia Tesla K40m GPUs and one FDR InfiniBand (56Gb/s) which is a high-performance low-latency interconnection and supports RDMA. The peak single precision floating point performance of each GPU is 4.29TFlops and each GPU has 12GB of memory. Thanks to the GPUDirect RDMA, the InfiniBand network interface can access the remote GPU memory without involvement from the CPU. All the server nodes are connected to the InfiniBand switch. Figure 1 shows the system architecture. The system runs Linux with CUDA 6.0 and MPI MVAPICH2, which also enables GPUDirect RDMA.

In total, Minwa has 6.9TB host memory, 1.7TB device memory, and about 0.6PFlops theoretical single precision peak performance.

Brute-force approach:

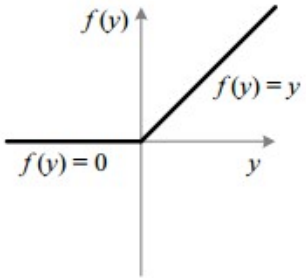
- Many AlexNet-like models at different resolutions
- Strong data augmentations

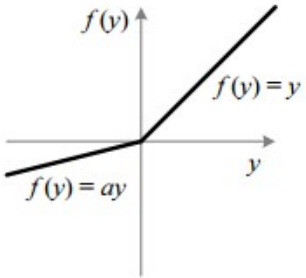
ImageNet classification Hit@5 error: **5.33%**
(Recall, human error is ~5.1%,
and optimistic human error is ~3%)

[Deep Image: Scaling up Image Recognition, Wu Ren et al. 2015] (Baidu)

Delving Deep into Rectifiers:
Surpassing Human-Level Performance on ImageNet Classification

[Kaiming He et al., 2015] (MSR)





4.94% error

\+ Careful initialization of the weights

Summary:

- We discussed why **small filters** are a good idea: They pack more non-linearities and decrease number of parameters
- We talked about **Data Augmentation**
- We noticed that many ConvNets take advantage of **noise in forward pass** + at test tune evaluating the expected output w.r.t. the noise distributions.
- We talked about **ConvNets in practice**, the CPU/GPU, CPU/disk bottlenecks, CUDA, etc.

Next Lecture:

(in-class)