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

![Histogram of scores](image1.png)

![Cumulative distribution](image2.png)
Midterm stats

We threw out TF3 and TF8
Midterm stats

Question 3.1 mean scores

Question 3.2 mean scores
Midterm Stats

Bonus mean: 0.8
Last Time

Recurrent neural networks for modeling sequences

Vanilla RNNs

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

LSTMs

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigmoid} \\ \text{sigmoid} \\ \text{sigmoid} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_{t-1}^l \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \text{tanh}(c_t^l)$$
Last Time

- Sampling from RNN language models to generate text

Lemma 0.1. Assume (3) and (3) by the construction in the description. Suppose $X = \lim_{\rightarrow} X$ (by the formal open covering $X$ and a single map $\text{Proj}(A) = \text{Spec}(B)$ over $U$ compatible with the complex $\text{Set}(A) = \Gamma(X, \mathcal{O}_X)$).

When in this case of to show that $Q \rightarrow C_{2,X}$ 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 cutaway). If $T$ is surjective we may assume $T$ is connected with residue fields of $S$. Moreover there exists a closed subspace $Z \subset X$ of $X$ where $U \cap Z$ 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 $S = \text{Spec}(R)$ and $Y = \text{Spec}(R)$.

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

The following lemma surjective restecomposes of this implies that $\mathcal{F}_{U_0} = \mathcal{F}_{U_0} = \mathcal{F}_{X,...,0}$.

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

Lemma 0.3. In Situation ??, hence we may assume $q' = 0$.

Proof. We will use the property we see that $p$ is the next 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 $\delta_{n+1}$ is a scheme over $S$. □
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

- Load image and label
- "cat"
- CNN
- Compute loss
Data Augmentation

Load image and label

“cat”

Transform image

CNN

Compute loss
Data Augmentation

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

What the computer sees
Data Augmentation

1. Horizontal flips
Data Augmentation

2. Random crops/scales

Training: sample random crops / scales
Data Augmentation

2. Random crops/scales

**Training**: sample random crops / scales

ResNet:

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

2. Random crops/scales

**Training:** sample random crops / scales

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

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

2. Random crops/scales

**Training**: sample random crops / scales

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

![Example of color jitter](image_url)
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/combos 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

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 data if you want to train/use CNNs”
Transfer Learning

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

1. Train on Imagenet
Transfer Learning with CNNs

1. Train on Imagenet
2. Small dataset: feature extractor

Freeze these

Train this
Transfer Learning with CNNs

1. Train on Imagenet

2. Small dataset: feature extractor
   - Freeze these
   - Train this

3. Medium dataset: finetuning
   - more data = retrain more of the network (or all of it)
   - Freeze these
   - Train this
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]

<table>
<thead>
<tr>
<th></th>
<th>DeCAF_0</th>
<th>DeCAF_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogReg</td>
<td>40.94 ± 0.3</td>
<td>40.84 ± 0.3</td>
</tr>
<tr>
<td>SVM</td>
<td>39.36 ± 0.3</td>
<td>40.66 ± 0.3</td>
</tr>
<tr>
<td>Xiao et al. (2010)</td>
<td>38.0</td>
<td></td>
</tr>
<tr>
<td>more generic</td>
<td>very similar dataset</td>
<td>very different dataset</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>more specific</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **very little data**
  - quite a lot of data

<table>
<thead>
<tr>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>more generic</td>
<td>very similar dataset</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>more specific</td>
<td></td>
</tr>
<tr>
<td>very little data</td>
<td>Use Linear Classifier on top layer</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>Finetune a few layers</td>
</tr>
<tr>
<td>Dataset Type</td>
<td>Action 1</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>very little data</td>
<td>Use Linear Classifier on top layer</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>Finetune a few layers</td>
</tr>
</tbody>
</table>

**Diagram:**
- More generic
- More specific

**Notes:**
- Use Linear Classifier on top layer for very little data.
- Finetune a few layers for quite a lot of data.
- Finetune a larger number of layers for very different data.
- You’re in trouble… Try linear classifier from different stages for very little data with very little data.
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)

Object Detection
(Faster R-CNN)

Image Captioning: CNN + RNN

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

Object Detection
(Faster R-CNN)

Image Captioning: CNN + RNN

CNN pretrained on ImageNet

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 $H \times W \times C$ 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 x 7 filters
- Number of weights:
- three CONV with 3 x 3 filters
- Number of weights:
The power of small filters

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

- **one CONV with 7 x 7 filters**
  
  Number of weights: 
  \[ = C \times (7 \times 7 \times C) = 49 \ C^2 \]

- **three CONV with 3 x 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 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 $H \times W \times C$ 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:
$= (H \times W \times C) \times (7 \times 7 \times C)$
$= 49 \ HWC^2$
	hree 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:
$= 3 \times (H \times W \times C) \times (3 \times 3 \times C)$
$= 27 \ HWC^2$
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:
  \[ = C \times (7 \times 7 \times C) = 49 \, C^2 \]

- 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
The power of small filters

Why stop at 3 x 3 filters? Why not try 1 x 1?
The power of small filters

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

1. “bottleneck” 1 x 1 conv to reduce dimension
The power of small filters

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

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

H x W x C

\[ \text{Conv 1x1, C/2 filters} \]

\[ H \times W \times \left( \frac{C}{2} \right) \]

\[ \text{Conv 3x3, C/2 filters} \]

\[ H \times W \times \left( \frac{C}{2} \right) \]
The power of small filters

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

H x W x C

1. “bottleneck” 1 x 1 conv to reduce dimension

H x W x (C / 2)

2. 3 x 3 conv at reduced dimension

H x W x (C / 2)

3. Restore dimension with another 1 x 1 conv

H x W x C

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)
Conv 1x1, C filters
H x W x C

Bottleneck sandwich

H x W x C
Conv 3x3, C filters
H x W x C

Single 3 x 3 conv
The power of small filters

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

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

More nonlinearity, fewer params, less compute!

3.25 C^2 parameters

9 C^2 parameters
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?

\[ H \times W \times C \]

Conv 1x3, C filters

\[ H \times W \times C \]

Conv 3x1, C filters

\[ H \times W \times C \]
The power of small filters

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

Conv 1x3, C filters

\[ H \times W \times C \]

Conv 3x1, C filters

\[ H \times W \times C \]

\[ H \times W \times C \]

\[ 6 C^2 \text{ parameters} \]

Conv 3x3, C filters

\[ H \times W \times C \]

\[ 9 C^2 \text{ parameters} \]

More nonlinearity, fewer params, less compute!
The power of small filters

Latest version of GoogLeNet incorporates all these ideas

Szegedy et al, “Rethinking the Inception Architecture for Computer Vision”
How to stack convolutions: Recap

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

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

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

Reshape $K \times K \times C$
receptive field to column
with $K^2C$ elements
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 $K \times K \times C$

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

Elements appearing in multiple receptive fields are duplicated; this uses a lot of memory
Implementing Convolutions: im2col

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

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

$(K^2C) \times N$ matrix

Reshape each filter to $K^2C$ row, making $D \times (K^2C)$ matrix
Implementing Convolutions: im2col

Feature map: \( H \times W \times C \)

Conv weights: \( D \) filters, each \( K \times K \times C \)

\( (K^2C) \times N \) matrix

\( D \times (K^2C) \) matrix

Matrix multiply

\( D \times N \) result; reshape to output tensor
Case study: CONV forward in Caffe library

```cpp
void ConvolutionLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*>& bottom,
                                          vector<Blob<Dtype>*>& top) {
  for (int i = 0; i < bottom.size(); ++i) {
    Dtype* bottom_data = bottom[i]->gpu_data();
    Dtype* top_data = (*top)[i]->mutable_gpu_data();
    Dtype* col_data = col_buffer_.mutable_gpu_data();
    const Dtype* weight = this->blobs_[0]->gpu_data();
    int weight_offset = M_ * K_;
    int col_offset = K_ * N_;
    int top_offset = M_ * N_;
    for (int n = 0; n < num_; ++n) {
      // im2col transformation: unroll input regions for filtering
      // into column matrix for multiplication
      im2col_gpu(bottom_data + bottom[i]->offset(n), channels_, height_,
                  width_, kernel_h_, kernel_w_, pad_h_, pad_w_, stride_h_, stride_w_,
                  col_data);
      // Take inner products for groups.
      for (int g = 0; g < group_; ++g) {
        caffe_gpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, M_, N_, K_,
                               (Dtype)1., weight + weight_offset * g, col_data + col_offset * g,
                               (Dtype)8., 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));
      }
    }
  }
}
```

- **im2col**
- **matrix multiply: call to cuBLAS**
- **bias offset**
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 ffbft: A GPU Performance Evaluation

Fei-Fei Li & Andrej Karpathy & Justin Johnson

Lecture 11 - 77
17 Feb 2016
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 $O(N^\log_2(7)) \sim O(N^{2.81})$

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

$$M_1 := (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$
$$M_2 := (A_{2,1} + A_{2,2})B_{1,1}$$
$$M_3 := A_{1,1}(B_{1,2} - B_{2,2})$$
$$M_4 := A_{2,2}(B_{2,1} - B_{1,1})$$
$$M_5 := (A_{1,1} + A_{1,2})B_{2,2}$$
$$M_6 := (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$
$$M_7 := (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$
$$C_{1,2} = M_3 + M_5$$
$$C_{2,1} = M_2 + M_4$$
$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

From Wikipedia
Implementing convolution: “Fast Algorithms”

Similar cleverness can be applied to convolutions

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

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

\[
m_1 = (d_0 - d_2)g_0 \\
m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2} \\
m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}
\]

Lavin and Gray, “Fast Algorithms for Convolutional Neural Networks”, 2015
Implementing convolution: “Fast Algorithms”

Huge speedups on VGG for small batches:

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.

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”
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](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
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](http://torch.ch/blog/2016/02/04/resnets.html)
Multi-GPU training: More complex

Alex Krizhevsky, “One weird trick for parallelizing convolutional neural networks”
Google: Distributed CPU training

Data parallelism

[Large Scale Distributed Deep Networks, Jeff Dean et al., 2013]
Google: Distributed CPU training

Data parallelism

Model parallelism

[Large Scale Distributed Deep Networks, Jeff Dean et al., 2013]
Google: Synchronous vs Async

Abadi et al, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”
Bottlenecks
to be aware of
GPU - CPU communication is a bottleneck. =>

CPU data prefetch+augment thread running while

GPU performs forward/backward pass
CPU - disk bottleneck

Hard disk is slow to read from

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

Moving parts lol
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!
Floating point precision

**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 =(  

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

How low can we go?

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

CNNs on MNIST

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 precision 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