Lecture 10: Training CNNs

Thursday March 2, 2017
Announcements!

- Final Project proposals due this **Today**

- I will be out of town next week. Rishit will lead class discussions.

- Next paper: **March 7** *You Only Look Once: Unified, Real-Time Object Detection*. If this paper seems too deep or confusing, look at *Fast R-CNN, Faster R-CNN*
Opportunity: Google Brain Residency

What Is The Brain Residency Program?
The Google Brain Residency Program is a one-year intensive residency program focused on Deep Learning. Residents will have the opportunity to conduct cutting-edge research and work alongside some of the most distinguished deep learning scientists within the Google Brain team. To learn more about the team and what we do, visit g.co/brain

- Email contact for questions: brain-residency@google.com
- For more information on the Residency Program, check out our website at g.co/brainresidency
- More recently, we published a blog post on the Google Research Blog where we discuss updates on current Residents’ progress and our program focus for 2017.
Data Augmentation
Data Augmentation

Load image and label

“cat”

CNN

Compute loss
Data Augmentation

Load image and label

“cat”

Transform image

Compute loss

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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 \( 224 \times 224 \) patch

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

2. Random crops/scales

**Training:** sample random crops / scales

ResNet:
1. Pick random L in range \([256, 480]\]
2. Resize training image, short side = L
3. Sample random 224 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
1. Add offset to all pixels of a training image
   (As seen in [Krizhevsky et al. 2012], ResNet, etc)
Data Augmentation
3. Color jitter

Random mix/combinations of:
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)
A general theme:

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

- Data Augmentation
- Dropout
- Batch normalization, Model ensembles
Data Augmentation: Takeaway

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

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

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

Not True
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
   - Freeze these
   - more data = retrain
   - more of the network (or all of it)
   - Train this

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

comp150dl
Transfer Learning with CNNs

1. Train on Imagenet

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

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

Tip: use only ~1/10th of the original learning rate in finetuning top layer, and ~1/100th on intermediate layers
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>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>very little data</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
<table>
<thead>
<tr>
<th></th>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>very little data</strong></td>
<td>Use Linear Classifier on top layer</td>
<td>?</td>
</tr>
<tr>
<td><strong>quite a lot of data</strong></td>
<td>Finetune a few layers</td>
<td>?</td>
</tr>
</tbody>
</table>

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
<table>
<thead>
<tr>
<th>more generic</th>
<th>more specific</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>very little data</strong></td>
<td><strong>Use Linear Classifier on top layer</strong></td>
</tr>
<tr>
<td><strong>quite a lot of data</strong></td>
<td><strong>Finetune a few layers</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Finetune a larger number of layers</strong></td>
</tr>
</tbody>
</table>

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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)

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)

CNN pretrained on ImageNet

Word vectors pretrained from word2vec

Image Captioning: CNN + RNN

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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
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 $7 \times 7$ filters

Number of weights:

three CONV with $3 \times 3$ filters

Number of weights:
The power of small filters

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

one CONV with $7 \times 7$ filters

Number of weights:

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

three CONV with $3 \times 3$ filters

Number of weights:

$$= 3 \times C \times (3 \times 3 \times C) = 27 \ C^2$$
The power of small filters

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

- **One CONV with 7 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:
    \[ = 3 \times C \times (3 \times 3 \times C) = 27 C^2 \]

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

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:
$= 3 \times (H \times W \times C) \times (3 \times 3 \times C)
= 27 \ HWC^2$
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?
(note: 1x1 filters sum across all channels of the input)

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

\[ H \times W \times C \]

Conv 1x1, C/2 filters

\[ H \times W \times (C / 2) \]
The power of small filters

Why stop at 3 x 3 filters? Why not try 1 x 1?
(note: 1x1 filters sum across all channels of the input)

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?
(note: 1x1 filters sum across all channels of the input)

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

Single 3 x 3 conv

H x W x C

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
The power of small filters

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

- Conv 1x1, C/2 filters
  \[ H \times W \times (C / 2) \]
- Conv 3x3, C/2 filters
  \[ H \times W \times (C / 2) \]
- Conv 1x1, C filters
  \[ H \times W \times C \]

- Conv 1x1, C/2 filters
  \[ 3.25 \, C^2 \text{ parameters} \]

- Conv 3x3, C filters
  \[ 9 \, C^2 \text{ parameters} \]

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

Still using 3 x 3 filters … can we break it up?
The power of small filters

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

\[ H \times W \times C \]

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 x W x C

H x W x C

Conv 3x1, C filters

H x W x C

More nonlinearity, fewer params, less compute!

Conv 3x3, C filters

H x W x C

9 C^2 parameters

6 C^2 parameters

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
INCEPTION MODULES

aka GoogLeNet

* figure courtesy Aaditya Parkash
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: $H \times W \times C$

Conv weights: $D$ filters, each $K \times K \times C$
Implementing Convolutions: im2col

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

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

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

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Case study: CONV forward in Caffe library

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

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Implementing convolutions: FFT

1. Compute FFT of weights: $F(W)$

2. Compute FFT of image: $F(X)$

3. Compute elementwise product: $F(W) \circ F(X)$

4. Compute inverse FFT: $Y = F^{-1}(F(W) \circ F(X))$
Implementing convolutions: FFT

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

Vasilache et al, Fast Convolutional Nets With fbfft: A GPU Performance Evaluation
Implementing convolution: “Fast Algorithms”

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

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

\[
\begin{align*}
A &= \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \\
B &= \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} \\
C &= \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{2,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \\
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6
\end{align*}
\]

From Wikipedia

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Implementing convolution: “Fast Algorithms”

Similar cleverness can be applied to convolutions

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

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

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

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.

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Computing Convolutions: Recap

- \text{im2col}: Easy to implement, but big memory overhead

- \text{FFT}: Big speedups for small kernels

- “Fast Algorithms” seem promising, not widely used yet
Implementation Details
* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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
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)

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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 in your 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*
Floating point precision

- How low can we go?

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

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
Recap

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

https://www.tensorflow.org
TensorFlow

- From Google
- Very similar to Theano - all about computation graphs
- Easy visualizations (TensorBoard)
- Multi-GPU and multi-node training
TensorFlow: Two-Layer Net

```python
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20
21 xx = np.random.randn(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[np.arange(N), np.random.randint(C, size=N)] = 1
24
25 with tf.Session() as sess:
26   sess.run(tf.initialize_all_variables())
27   for t in range(100):
28     _, loss_value = sess.run([train_step, loss],
29                               feed_dict={x: xx, y: yy})
30     print 'loss_value'
```

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Create placeholders for data and labels: These will be fed to the graph

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10
x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = tf.reduce_sum(y * tf.log(probs))

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    for t in range(100):
        loss_value = sess.run([train_step, loss],
                              feed_dict={x: xx, y: yy})
        print loss_value
```
Create Variables to hold weights; similar to Theano shared variables

Initialize variables with numpy arrays
TensorFlow: Two-Layer Net

**Forward**: Compute scores, probs, loss (symbolically)
TensorFlow: Two-Layer Net

Running train_step will use SGD to minimize loss
TensorFlow: Two-Layer Net

Create an artificial dataset; y is one-hot like Keras

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
a = tf.matmul(x, w1)
relu = tf.nn.relu(a)
 scores = tf.matmul(relu, w2)
probs = tf.nn.softmax(scores)
loss = tf.reduce_sum(y * tf.log(probs))

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())

for t in xrange(100):
    _, loss_value = sess.run([train_step, loss],
                                feed_dict={x: xx, y: yy})
    print loss_value
```
TensorFlow: Two-Layer Net

Actually train the model
TensorFlow: Tensorboard

Tensorboard makes it easy to visualize what’s happening inside your models
TensorFlow: Tensorboard

Tensorboard makes it easy to visualize what's happening inside your models

Same as before, but now we create summaries for loss and weights

```python
import tensorflow as tf
import numpy as np

x = tf.placeholder(tf.float32, shape=[None, 1])
y = tf.placeholder(tf.float32, shape=[None, 1])
w1 = tf.Variable(np.random.randn(1, 100).astype(np.float32))
w2 = tf.Variable(np.random.randn(100, 1).astype(np.float32))
a = tf.matmul(x, w1)
a1 = tf.nn.relu(a)
scores = tf.matmul(a1, w2)
prob = tf.nn.softmax(scores)
loss = -tf.reduce_sum(tf.log(prob))
loss_summary = tf.scalar_summary('loss', loss)
w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)
```

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

comp150dl
TensorFlow: Tensorboard

Tensorboard makes it easy to visualize what’s happening inside your models

Create a special “merged” variable and a SummaryWriter object

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10
x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, H])
w1 = tf.Variable(tf.random_normal([N, W]), dtype=tf.float32)
w2 = tf.Variable(tf.random_normal([H, C]), dtype=tf.float32)
a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = tf.reduce_sum(-tf.log(probs))

loss_summary = tf.scalar_summary('loss', loss)

w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)

with tf.Session() as sess:
    merged = tf.merge_all_summaries()
    writer = tf.train.SummaryWriter('/tmp/tf_logs', sess.graph_def)
    sess.run(tf.initialize_all_variables())
    for t in range(100):
        summary_str, loss_value = sess.run([merged, train_step, loss],
                                           {feed_dict:x: x, y: y})
        writer.add_summary(summary_str, t)
        print loss_value
```
TensorFlow: Tensorboard

Tensorboard makes it easy to visualize what’s happening inside your models

In the training loop, also run merged and pass its value to the writer

```python
for t in range(100):
    summary_str, loss_value = sess.run([merged, train_step, loss],
        feed_dict={x: xx, y: yy})
    writer.add_summary(summary_str, t)
```

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
TensorFlow: Tensorboard

Start Tensorboard server, and we get graphs!
TensorFlow: TensorBoard

```python
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D], name='x')
7 y = tf.placeholder(tf.float32, shape=[None, C], name='y')
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32), name='w1')
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32), name='w2')
11
12 with tf.name_scope('scores') as scope:
13     a = tf.matmul(x, w1)
14     a_relu = tf.nn.relu(a)
15     scores = tf.matmul(a_relu, w2)
16
17 with tf.name_scope('loss') as scope:
18     probs = tf.nn.softmax(scores)
19     loss = -tf.reduce_sum(y * tf.log(probs))
20
21     loss_summary = tf.scalar_summary('loss', loss)
22     w1_hist = tf.histogram_summary('w1', w1)
23     w2_hist = tf.histogram_summary('w2', w2)
24
25     learning_rate = 1e-2
26
27     train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
28
29     xx = np.random.randn(N, D).astype(np.float32)
30     yy = np.zeros((N, C)).astype(np.float32)
31     yy[np.arange(N), np.random.randint(C, size=N)] = 1
```
TensorFlow: TensorBoard

Add names to placeholders and variables
TensorFlow: TensorBoard

Add names to placeholders and variables

Break up the forward pass with name scoping
TensorFlow: TensorBoard

Tensorboard shows the graph!
TensorFlow: TensorBoard

Tensorboard shows the graph!

Name scopes expand to show individual operations

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
TensorFlow: Multi-GPU

Data parallelism: synchronous or asynchronous
TensorFlow: Multi-GPU

Data parallelism:
synchronous or asynchronous

Model parallelism:
Split model across GPUs
TensorFlow: Distributed

Single machine:
Like other frameworks

Many machines:
Not open source (yet) =(
TensorFlow: Pretrained Models

- You can get a pretrained models here:
  - https://github.com/tensorflow/models

- Has inception, resnet, some different autoencoders
TensorFlow: Pros / Cons

(+): Python + numpy
(+): Computational graph abstraction, like Theano; great for RNNs
(+): Much faster compile times than Theano
(+): Slightly more convenient than raw Theano?
(+): TensorBoard for visualization
(+): Data AND model parallelism; best of all frameworks
(+/-): Distributed models, but not open-source yet
(-): Slower than other frameworks right now
(-): Much “fatter” than Torch; more magic
(-): Not many pretrained models

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Torch

http://torch.ch/docs/getting-started.html
Torch Overview

- From NYU + IDIAP
- Written in C and Lua
- Used a lot a Facebook, DeepMind
Torch: Pretrained Models

- **loadcaffe**: Load pretrained Caffe models: AlexNet, VGG, some others
  https://github.com/szagoruyko/loadcaffe

- **GoogLeNet v1**: https://github.com/soumith/inception.torch

- **GoogLeNet v3**: https://github.com/Moodstocks/inception-v3.torch

Torch: Pros / Cons

(-) Lua
(-) Less plug-and-play than Caffe
    You usually write your own training code
(+ ) Lots of modular pieces that are easy to combine
(+ ) Easy to write your own layer types and run on GPU
(+ ) Most of the library code is in Lua, easy to read
(+ ) Lots of pretrained models!
(-) Not great for RNNs
Theano

http://deeplearning.net/software/theano/
Theano Overview

- From Yoshua Bengio’s group at University of Montreal
- Embracing computation graphs, symbolic computation
- High-level wrappers: Keras, Lasagne
- Has Conditional flow (if else, switch)
Theano: Pretrained Models

- **Lasagne Model Zoo** has pretrained common architectures:
  - https://github.com/Lasagne/Recipes/tree/master/modelzoo

- **AlexNet with weights**: https://github.com/uoguelph-mlrg/theano_alexnet

- **sklearn-theano**: Run OverFeat and GoogLeNet forward, but no fine-tuning? http://sklearn-theano.github.io

- **caffe-theano-conversion**: CS 231n project from last year: load models and weights from caffe! Not sure if full-featured https://github.com/kitofans/caffe-theano-conversion

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Theano: Pros / Cons

(+) Python + numpy
(+) Computational graph is nice abstraction
(+) RNNs fit nicely in computational graph
(-) Raw Theano is somewhat low-level
(+ ) High level wrappers (Keras, Lasagne) ease the pain
(-) Error messages can be unhelpful
(-) Large models can have long compile times
(-) Much “fatter” than Torch; more magic
(-) Patchy support for pretrained models

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Caffe</th>
<th>Torch</th>
<th>Theano</th>
<th>TensorFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td>C++, Python</td>
<td>Lua</td>
<td>Python</td>
<td>Python</td>
</tr>
<tr>
<td><strong>Pretrained</strong></td>
<td>Yes ++</td>
<td>Yes ++</td>
<td>Yes (Lasagne)</td>
<td>Inception</td>
</tr>
<tr>
<td><strong>Multi-GPU:</strong></td>
<td><strong>Data parallel</strong></td>
<td>Yes</td>
<td>Yes platoon</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td>Experimental</td>
<td>Yes (best)</td>
</tr>
<tr>
<td><strong>Multi-GPU:</strong></td>
<td><strong>Model parallel</strong></td>
<td>No</td>
<td>Yes (best)</td>
<td></td>
</tr>
<tr>
<td><strong>Readable source code</strong></td>
<td>Yes (C++)</td>
<td>Yes (Lua)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Good at RNN</strong></td>
<td>No</td>
<td>Mediocre</td>
<td>Yes</td>
<td>Yes (best)</td>
</tr>
</tbody>
</table>