Lecture 16: Convolutional Networks II
Administrative

HW4 Released, due Monday March 29, 11:59pm ET

Course Project:
- We will give ~6 suggested project descriptions
- Choose one, or propose your own
- We expect ~1 HW of work per person for project
Last Time: Convolutional Networks

\[ y = Wx + b \]

Fully-Connected Layers

Activation Function

\[ y = \max(0, x) \]

Convolution Layers

Pooling Layers

Normalization

\[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \]
Components of a Convolutional Network

**Fully-Connected Layers**

\[ y = Wx + b \]

**Activation Function**

\[ y = \max(0, x) \]

**Convolution Layers**

**Pooling Layers**

**Normalization**

\[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \]
Batch Normalization

Idea: “Normalize” the outputs of each layer so they have zero mean and unit variance

Why? Helps reduce “internal covariate shift”, improves optimization

Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015
Batch Normalization

**Idea:** “Normalize” the outputs of each layer so they have zero mean and unit variance

Why? Helps reduce “internal covariate shift”, improves optimization

We can normalize a batch of activations like this:

\[
\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}
\]

This is a **differentiable function**, so we can use it as an operator in our networks and backprop through it!

Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015
Batch Normalization

Input: \( x \in \mathbb{R}^{N \times D} \)

\[
\begin{align*}
\mu_j &= \frac{1}{N} \sum_{i=1}^{N} x_{i,j} \\
\sigma_j^2 &= \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}
\end{align*}
\]

Per-channel mean, shape is D

Per-channel std, shape is D

Normalized x, Shape is N x D

Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015
Batch Normalization

Input: \( x \in \mathbb{R}^{N \times D} \)

\[
\begin{align*}
\mu_j &= \frac{1}{N} \sum_{i=1}^{N} x_{i,j} & \text{Per-channel mean, shape is D} \\
\sigma_j^2 &= \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 & \text{Per-channel std, shape is D} \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} & \text{Normalized x, Shape is N x D}
\end{align*}
\]

Problem: What if zero-mean, unit variance is too restrictive?

Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015
Batch Normalization

Input: \( x \in \mathbb{R}^{N \times D} \)

Learnable scale and shift parameters:
\( \gamma, \beta \in \mathbb{R}^D \)

Learning \( \gamma = \sigma, \beta = \mu \) will recover the identity function (in expectation)

\[
\begin{align*}
\mu_j &= \frac{1}{N} \sum_{i=1}^{N} x_{i,j} & \text{Per-channel mean, shape is D} \\
\sigma_j^2 &= \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 & \text{Per-channel std, shape is D} \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} & \text{Normalized x, Shape is N x D} \\
\gamma_{i,j} &= \gamma_j \hat{x}_{i,j} + \beta_j & \text{Output, Shape is N x D}
\end{align*}
\]
Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function (in expectation)

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is $D$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-channel std, shape is $D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized $x$, Shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is $N \times D$

Problem: Estimates depend on minibatch; can’t do this at test-time!
**Batch Normalization: Test-Time**

**Input:** \( x \in \mathbb{R}^{N \times D} \)

**Learnable scale and shift parameters:**

\[ \gamma, \beta \in \mathbb{R}^D \]

Learning \( \gamma = \sigma, \beta = \mu \) will recover the identity function (in expectation)

\[
\begin{align*}
\mu_j &= \text{(Running) average of values seen during training} \\
\sigma_j^2 &= \text{(Running) average of values seen during training} \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \\
y_{i,j} &= \gamma_j \hat{x}_{i,j} + \beta_j
\end{align*}
\]

- \( \mu_j \) \text{ Per-channel mean, shape is } D
- \( \sigma_j^2 \) \text{ Per-channel std, shape is } D
- Normalized \( x \), \text{ Shape is } N \times D
- Output, \text{ Shape is } N \times D
Batch Normalization: Test-Time

**Input:** \( x \in \mathbb{R}^{N \times D} \)

**Learnable scale and shift parameters:**
\[ \gamma, \beta \in \mathbb{R}^{D} \]

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

\[
\begin{align*}
\mu_j &= \text{(Running) average of values seen during training} \\
\sigma_j^2 &= \text{(Running) average of values seen during training} \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \\
y_{i,j} &= \gamma_j \hat{x}_{i,j} + \beta_j \\
\end{align*}
\]

Per-channel mean, shape is D
Per-channel std, shape is D
Normalized x, Shape is N x D
Output, Shape is N x D
Batch Normalization for ConvNets

**Batch Normalization for fully-connected networks**

\[ x : N \times D \]

Normalize

\[ \mu, \sigma : 1 \times D \]

\[ \gamma, \beta : 1 \times D \]

\[ y = \frac{(x - \mu)}{\sigma} \gamma + \beta \]

**Batch Normalization for convolutional networks**

(Spatial Batchnorm, BatchNorm2D)

\[ x : N \times C \times H \times W \]

Normalize

\[ \mu, \sigma : 1 \times C \times 1 \times 1 \]

\[ \gamma, \beta : 1 \times C \times 1 \times 1 \]

\[ y = \frac{(x - \mu)}{\sigma} \gamma + \beta \]
Batch Normalization

Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

\[ \hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}, \]
Batch Normalization

- Makes deep networks much easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Free at test-time: can be fused with conv!

![Diagram of Batch Normalization]

ImageNet accuracy vs. Training iterations

Conv → BN → ReLU → Conv → BN → ReLU

- Inception
- BN-Baseline
- BN-x5
- BN-x30
- BN-x5-Sigmoid

Steps to match Inception
Batch Normalization

- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Free at test-time: can be fused with conv!
- Not well-understood theoretically
- Behaves differently during training and testing: this is a very common source of bugs!
Convolutional Networks

Fully-Connected Layers

\[ y = Wx + b \]

Activation Function

\[ y = \max(0, x) \]

Convolution Layers

Pooling Layers

Normalization

\[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \]
Convolutional Networks

How can we combine these components into full architectures?

\[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \]
ImageNet Classification Challenge

<table>
<thead>
<tr>
<th>Year</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>28.2</td>
</tr>
<tr>
<td>2011</td>
<td>25.8</td>
</tr>
<tr>
<td>2012</td>
<td>16.4</td>
</tr>
</tbody>
</table>

- **Lin et al.**
- **Sanchez & Perronnin**
- **Krizhevsky et al.** (AlexNet)
- **Simonyan & Zisserman** (VGG)
- **Szegedy et al.** (GoogLeNet)
- **He et al.** (ResNet)

Shallow Layer: 8 Layers
227 x 227 inputs
5 Convolutional layers
Max pooling
3 fully-connected layers
ReLU nonlinearities
AlexNet

227 x 227 inputs
5 Convolutional layers
Max pooling
3 fully-connected layers
ReLU nonlinearities

Used “Local response normalization”;
Not used anymore

Trained on two GTX 580 GPUs – only 3GB of memory each! Model split over two GPUs

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.
## AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>C</th>
<th>H / W</th>
<th>Filters</th>
<th>Kernel</th>
<th>Stride</th>
<th>Pad</th>
<th>C</th>
<th>H / W</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>conv1</strong></td>
<td>3</td>
<td>227</td>
<td>64</td>
<td>11</td>
<td>4</td>
<td>2</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>Layer</td>
<td>C</td>
<td>H / W</td>
<td>filters</td>
<td>kernel</td>
<td>stride</td>
<td>pad</td>
<td>C</td>
<td>H / W</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-------</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
<td>11</td>
<td>4</td>
<td>2</td>
<td>64</td>
<td>?</td>
</tr>
</tbody>
</table>

Recall: Output channels = number of filters
### AlexNet

<table>
<thead>
<tr>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Layer</strong></td>
<td><strong>C</strong></td>
<td><strong>H / W</strong></td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
</tr>
</tbody>
</table>

Recall: \( W' = \frac{(W - K + 2P)}{S} + 1 \)

\[
= \frac{227 - 11 + 2*2}{4} + 1 \\
= \frac{220}{4} + 1 = 56
\]
## AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>H / W</td>
<td>C</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
</tr>
</tbody>
</table>

- **Input size**: Number of channels (C), height (H) and width (W) of the input image.
- **Layer**: The name of the layer.
- **Output size**: Number of channels (C), height (H) and width (W) of the output image.
- **Memory (KB)**: Memory required for the layer.
## AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>H / W</td>
<td>filters</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
</tr>
</tbody>
</table>

Number of output elements = $C \times H' \times W'$  
= $64 \times 56 \times 56 = 200,704$

Bytes per element = 4 (for 32-bit floating point)

KB = (number of elements) * (bytes per elem) / 1024  
= $200704 \times 4 / 1024$  
= $784$
### AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>C</th>
<th>H / W</th>
<th>filters</th>
<th>kernel</th>
<th>stride</th>
<th>pad</th>
<th>C</th>
<th>H / W</th>
<th>memory (KB)</th>
<th>params (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
<td>11</td>
<td>4</td>
<td>2</td>
<td>64</td>
<td>56</td>
<td>784</td>
<td>?</td>
</tr>
</tbody>
</table>
### AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>H / W</td>
<td>filters</td>
<td>kernel</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
</tr>
</tbody>
</table>

Weight shape = \( C_{\text{out}} \times C_{\text{in}} \times K \times K \)

= \( 64 \times 3 \times 11 \times 11 \)

Bias shape = \( C_{\text{out}} = 64 \)

Number of weights = \( 64 \times 3 \times 11 \times 11 + 64 \)

= \( 23,296 \)
## AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>3 227 64 11 4 2</td>
<td>64 56</td>
<td>784 23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layer</th>
<th>C H / W filters kernel stride pad</th>
<th>C H / W memory (KB) params (k) flop (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>3 227 64 11 4 2</td>
<td>64 56 784 23</td>
</tr>
</tbody>
</table>
### AlexNet

<table>
<thead>
<tr>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer</td>
<td>C</td>
<td>H / W</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
</tr>
</tbody>
</table>

Number of floating point operations (multiply+add)  
= (number of output elements) * (ops per output elem)  
= (C_{out} x H' x W') * (C_{in} x K x K)  
= (64 * 56 * 56) * (3 * 11 * 11)  
= 200,704 * 363  
= **72,855,552**
## AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>C</th>
<th>H / W</th>
<th>filters</th>
<th>kernel</th>
<th>stride</th>
<th>pad</th>
<th>C</th>
<th>H / W</th>
<th>memory (KB)</th>
<th>params (k)</th>
<th>flop (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
<td>11</td>
<td>4</td>
<td>2</td>
<td>64</td>
<td>56</td>
<td>784</td>
<td>23</td>
<td>73</td>
</tr>
<tr>
<td>pool1</td>
<td>64</td>
<td>56</td>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C H / W</td>
<td>filters kernel stride pad</td>
<td>C H / W memory (KB) params (k) flop (M)</td>
</tr>
<tr>
<td>conv1</td>
<td>3 227</td>
<td>64 11 4 2</td>
<td>64 56 784 23 73</td>
</tr>
<tr>
<td>pool1</td>
<td>64 56</td>
<td>3 2 0</td>
<td>64 27</td>
</tr>
</tbody>
</table>

For pooling layer:

#output channels = #input channels = 64

\[ W' = \text{floor}((W - K) / S + 1) \]

\[ = \text{floor}(53 / 2 + 1) = \text{floor}(27.5) = 27 \]
### AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Output size</th>
<th>memory (KB)</th>
<th>params (k)</th>
<th>flop (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>3 x 227 x 227 64 x 11 x 4 x 2</td>
<td>64 x 56</td>
<td>784</td>
<td>23</td>
<td>73</td>
</tr>
<tr>
<td>pool1</td>
<td>64 x 56</td>
<td></td>
<td>182</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#output elems = \( C_{out} \times H' \times W' \)

Bytes per elem = 4

KB = \( C_{out} \times H' \times W' \times 4 / 1024 \)

\[ = 64 \times 27 \times 27 \times 4 / 1024 \]

\[ = 182.25 \]
### AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
<th>Memory (KB)</th>
<th>Params (k)</th>
<th>Flop (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>H / W</td>
<td>filters</td>
<td>kernel</td>
<td>stride</td>
<td>pad</td>
<td>C</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
<td>11</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>pool1</td>
<td>64</td>
<td>56</td>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Pooling layers have no learnable parameters!
## AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>3 227</td>
<td>filters 64 kernel 11 stride 4 pad 2</td>
<td>conv1 64 56</td>
<td>pool1</td>
<td>64 56</td>
<td>pool1 64 27</td>
</tr>
<tr>
<td>pool1</td>
<td>64 56</td>
<td>3 2 0</td>
<td>pool1 64 27</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Floating-point ops for pooling layer
= (number of output positions) * (flops per output position)
= \((C_{out} \times H' \times W') \times (K \times K)\)
= \((64 \times 27 \times 27) \times (3 \times 3)\)
= 419,904
= **0.4 MFLOP**
### AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>H / W</td>
<td>filters</td>
<td>kernel</td>
<td>stride</td>
<td>pad</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
<td>11</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>pool1</td>
<td>64</td>
<td>56</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>conv2</td>
<td>64</td>
<td>27</td>
<td>192</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>pool2</td>
<td>192</td>
<td>27</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>192</td>
</tr>
<tr>
<td>conv3</td>
<td>192</td>
<td>13</td>
<td>384</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>conv4</td>
<td>384</td>
<td>13</td>
<td>256</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>conv5</td>
<td>256</td>
<td>13</td>
<td>256</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pool5</td>
<td>256</td>
<td>13</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>flatten</td>
<td>256</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flatten output size = $C_{in} \times H \times W$

= $256 \times 6 \times 6$

= 9216
### AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
<th>Memory (KB)</th>
<th>Params (k)</th>
<th>Flop (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>H / W</td>
<td>filters</td>
<td>kernel</td>
<td>stride</td>
<td>pad</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
<td>11</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>pool1</td>
<td>64</td>
<td>56</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>conv2</td>
<td>64</td>
<td>27</td>
<td>192</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>pool2</td>
<td>192</td>
<td>27</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>conv3</td>
<td>192</td>
<td>13</td>
<td>384</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>conv4</td>
<td>384</td>
<td>13</td>
<td>256</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>conv5</td>
<td>256</td>
<td>13</td>
<td>256</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pool5</td>
<td>256</td>
<td>13</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>flatten</td>
<td>256</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fc6</td>
<td>9216</td>
<td>4096</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FC params** = \( C_{in} \times C_{out} + C_{out} \)

\[
= 9216 \times 4096 + 4096 \\
= 37,725,832
\]

**FC flops** = \( C_{in} \times C_{out} \)

\[
= 9216 \times 4096 \\
= 37,748,736
\]
## AlexNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
<th>Memory (KB)</th>
<th>Params (k)</th>
<th>Flop (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>H / W</td>
<td>filters</td>
<td>kernel</td>
<td>stride</td>
<td>pad</td>
<td>C</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
<td>11</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>pool1</td>
<td>64</td>
<td>56</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>conv2</td>
<td>64</td>
<td>27</td>
<td>192</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>pool2</td>
<td>192</td>
<td>27</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>192</td>
</tr>
<tr>
<td>conv3</td>
<td>192</td>
<td>13</td>
<td>384</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>conv4</td>
<td>384</td>
<td>13</td>
<td>256</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>conv5</td>
<td>256</td>
<td>13</td>
<td>256</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pool5</td>
<td>256</td>
<td>13</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>flatten</td>
<td>256</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>9216</td>
</tr>
<tr>
<td>fc6</td>
<td>9216</td>
<td>4096</td>
<td></td>
<td></td>
<td></td>
<td>4096</td>
</tr>
<tr>
<td>fc7</td>
<td>4096</td>
<td>4096</td>
<td></td>
<td></td>
<td></td>
<td>4096</td>
</tr>
<tr>
<td>fc8</td>
<td>4096</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
</tbody>
</table>
### AlexNet

How to choose this?

Trial and error =(  

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C  H/W filters kernel stride pad</td>
<td>C  H/W memory (KB) params (k) flop (M)</td>
</tr>
<tr>
<td>conv1</td>
<td>3 227</td>
<td>64 56</td>
</tr>
<tr>
<td>pool1</td>
<td>64 56</td>
<td>3 2</td>
</tr>
<tr>
<td>conv2</td>
<td>64 27</td>
<td>192 5</td>
</tr>
<tr>
<td>pool2</td>
<td>192 27</td>
<td>3 2</td>
</tr>
<tr>
<td>conv3</td>
<td>192 13</td>
<td>384 3</td>
</tr>
<tr>
<td>conv4</td>
<td>384 13</td>
<td>256 3</td>
</tr>
<tr>
<td>conv5</td>
<td>256 13</td>
<td>256 3</td>
</tr>
<tr>
<td>pool5</td>
<td>256 13</td>
<td>3 2</td>
</tr>
<tr>
<td>flatten</td>
<td>256 6</td>
<td>9216 36</td>
</tr>
<tr>
<td>fc6</td>
<td>9216 4096</td>
<td>4096 16</td>
</tr>
<tr>
<td>fc7</td>
<td>4096 4096</td>
<td>4096 16</td>
</tr>
<tr>
<td>fc8</td>
<td>4096 1000</td>
<td>1000 4</td>
</tr>
</tbody>
</table>
### AlexNet

**Interesting trends here!**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input size</th>
<th>Layer</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>H / W</td>
<td>filters</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>227</td>
<td>64</td>
</tr>
<tr>
<td>pool1</td>
<td>64</td>
<td>56</td>
<td>3</td>
</tr>
<tr>
<td>conv2</td>
<td>64</td>
<td>27</td>
<td>192</td>
</tr>
<tr>
<td>pool2</td>
<td>192</td>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>conv3</td>
<td>192</td>
<td>13</td>
<td>384</td>
</tr>
<tr>
<td>conv4</td>
<td>384</td>
<td>13</td>
<td>256</td>
</tr>
<tr>
<td>conv5</td>
<td>256</td>
<td>13</td>
<td>256</td>
</tr>
<tr>
<td>pool5</td>
<td>256</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>flatten</td>
<td>256</td>
<td>6</td>
<td>9216</td>
</tr>
<tr>
<td>fc6</td>
<td>9216</td>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>fc7</td>
<td>4096</td>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>fc8</td>
<td>4096</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>
AlexNet

Most of the **memory usage** is in the early convolution layers

Nearly all **parameters** are in the fully-connected layers

Most **floating-point ops** occur in the convolution layers

**Memory (KB)**

**Params (K)**

**MFLOP**
ImageNet Classification Challenge

Error Rate

- 2010: Shallow (Lin et al.)
- 2011: Shallow (Sanchez & Perronnin)
- 2012: Deep (Krizhevsky et al. (AlexNet); 8 Layer)

28.2
25.8
16.4
ImageNet Classification Challenge

- **2010**: Lin et al.
- **2011**: Sanchez & Perronnin
- **2012**: Krizhevsky et al. (AlexNet)
- **2013**: Zeiler & Fergus
- **2014**: Simonyan & Zisserman (VGG)
- **2014**: Szegedy et al. (GoogLeNet)

Error Rate:
- **2010**: 28.2%
- **2011**: 25.8%
- **2012**: 16.4%
- **2013**: 11.7%
- **2014**: 7.3%

Shallow:
- **8 Layer**
- **22 Layer**

Note: AlexNet refers to Alex Krizhevsky's deep convolutional neural network (CNN) architecture, which was named after its creators, Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. AlexNet won the ImageNet Large Scale Visual Recognition Challenge in 2012.
VGG: Deeper Networks, Regular Design

VGG Design rules:
All conv are 3x3 stride 1 pad 1
All max pool are 2x2 stride 2
After pool, double #channels

VGG: Deeper Networks, Regular Design

**VGG Design rules:**
All conv are 3x3 stride 1 pad 1
All max pool are 2x2 stride 2
After pool, double #channels

Network has 5 convolutional stages:
Stage 1: conv-conv-pool
Stage 2: conv-conv-pool
Stage 3: conv-conv-pool
Stage 4: conv-conv-conv-[conv]-pool
Stage 5: conv-conv-conv-[conv]-pool

(VGG-19 has 4 conv in stages 4 and 5)

VGG: Deeper Networks, Regular Design

VGG Design rules:
All conv are 3x3 stride 1 pad 1
All max pool are 2x2 stride 2
After pool, double #channels

Option 1:
Conv(5x5, C -> C)

Params: 25C²
FLOPs: 25C²HW

VGG: Deeper Networks, Regular Design

VGG Design rules:
All conv are 3x3 stride 1 pad 1
All max pool are 2x2 stride 2
After pool, double #channels

Option 1:
Conv(5x5, C -> C)
Conv(3x3, C -> C)
Params: 25C^2
FLOPs: 25C^2HW

Option 2:
Conv(3x3, C -> C)
Conv(3x3, C -> C)
Params: 18C^2
FLOPs: 18C^2HW

## VGG: Deeper Networks, Regular Design

**VGG Design rules:**

- **All conv are 3x3 stride 1 pad 1**
- **All max pool are 2x2 stride 2**
- **After pool, double #channels**

Two 3x3 conv has same receptive field as a single 5x5 conv, but has fewer parameters and takes less computation!

<table>
<thead>
<tr>
<th>Option 1:</th>
<th>Option 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv(5x5, C -&gt; C)</td>
<td>Conv(3x3, C -&gt; C)</td>
</tr>
<tr>
<td>Conv(3x3, C -&gt; C)</td>
<td>Conv(3x3, C -&gt; C)</td>
</tr>
</tbody>
</table>

**Params:** 25C^2  
**FLOPs:** 25C^2HW

<table>
<thead>
<tr>
<th>Option 1:</th>
<th>Option 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv(3x3, C -&gt; C)</td>
<td>Conv(3x3, C -&gt; C)</td>
</tr>
<tr>
<td>Conv(3x3, C -&gt; C)</td>
<td>Conv(3x3, C -&gt; C)</td>
</tr>
</tbody>
</table>

**Params:** 18C^2  
**FLOPs:** 18C^2HW

**AlexNet**


---

Justin Johnson & David Fouhey  
EECS 442 WI 2021: Lecture 16 - 49  
March 16, 2021
## VGG: Deeper Networks, Regular Design

### VGG Design rules:

- All conv are 3x3 stride 1 pad 1
- All max pool are 2x2 stride 2
- After pool, double #channels

### Input:

- C x 2H x 2W

### Layer:

- Conv(3x3, C->C)

### Memory:

- 4HWC

### Params:

- 9C^2

### FLOPs:

- 36HWC^2

---

VGG: Deeper Networks, Regular Design

VGG Design rules:
All conv are 3x3 stride 1 pad 1
All max pool are 2x2 stride 2
After pool, double #channels

Input: C x 2H x 2W
Layer: Conv(3x3, C->C)
Memory: 4HWC
Params: 9C^2
FLOPs: 36HWC^2

Input: 2C x H x W
Layer: Conv(3x3, 2C -> 2C)
Memory: 2HWC
Params: 36C^2
FLOPs: 36HWC^2

VGG: Deeper Networks, Regular Design

VGG Design rules:
- All conv are 3x3 stride 1 pad 1
- All max pool are 2x2 stride 2
- After pool, double #channels

Conv layers at each spatial resolution take the same amount of computation!

Input: C x 2H x 2W
Layer: Conv(3x3, C->C)
Memory: 4HWC
Params: 9C^2
FLOPs: 36HWC^2

Input: 2C x H x W
Layer: Conv(3x3, 2C -> 2C)
Memory: 2HWC
Params: 36C^2
FLOPs: 36HWC^2

AlexNet vs VGG-16: Much Bigger!

AlexNet vs VGG-16 (Memory, KB)

AlexNet total: 1.9 MB
VGG-16 total: 48.6 MB (25x)

AlexNet vs VGG-16 (Params, M)

AlexNet total: 61M
VGG-16 total: 138M (2.3x)

AlexNet vs VGG-16 (MFLOPs)

AlexNet total: 0.7 GFLOP
VGG-16 total: 13.6 GFLOP (19.4x)
ImageNet Classification Challenge

- 2010: Lin et al
- 2011: Sanchez & Perronnin
- 2012: Krizhevsky et al (AlexNet)
- 2013: Zeiler & Fergus
- 2014: Simonyan & Zisserman (VGG)
- 2014: Szegedy et al (GoogLeNet)

Error Rate:
- Shallow: 28.2
- 8 Layer: 25.8
- 8 Layer: 16.4
- 19 Layer: 11.7
- 19 Layer: 7.3
- 22 Layer: 6.7

ImageNet Classification Challenge

<table>
<thead>
<tr>
<th>Year</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>28.2</td>
</tr>
<tr>
<td>2011</td>
<td>25.8</td>
</tr>
<tr>
<td>2012</td>
<td>16.4</td>
</tr>
<tr>
<td>2013</td>
<td>11.7</td>
</tr>
<tr>
<td>2014</td>
<td>7.3</td>
</tr>
<tr>
<td>2015</td>
<td>6.7</td>
</tr>
<tr>
<td>2016</td>
<td>3.6</td>
</tr>
</tbody>
</table>

**Layers:**
- Shallow
- 8 Layer
- 19 Layer
- 22 Layer
- 152 Layer

**Authors:**
- Lin et al
- Sanchez & Perronnin
- Krizhevsky et al (AlexNet)
- Zeiler & Fergus
- Simonyan & Zisserman (VGG)
- Szegedy et al (GoogLeNet)
- He et al (ResNet)
Residual Networks

Once we have Batch Normalization, we can train networks with 10+ layers. What happens as we go deeper?
Residual Networks

Once we have Batch Normalization, we can train networks with 10+ layers. What happens as we go deeper?

Deeper model does worse than shallow model!

Initial guess: Deep model is overfitting since it is much bigger than the other model

Residual Networks

Once we have Batch Normalization, we can train networks with 10+ layers. What happens as we go deeper?

![Graph showing training and test error over iterations for 20-layer and 56-layer networks.](image)

Training error

Test error

In fact the deep model seems to be **underfitting** since it also performs worse than the shallow model on the training set! It is actually **underfitting**

Residual Networks

A deeper model can emulate a shallower model: copy layers from shallower model, set extra layers to identity

Thus deeper models should do at least as good as shallow models

**Hypothesis:** This is an optimization problem. Deeper models are harder to optimize, and in particular don’t learn identity functions to emulate shallow models

Residual Networks

A deeper model can emulate a shallower model: copy layers from shallower model, set extra layers to identity.

Thus deeper models should do at least as good as shallow models.

**Hypothesis:** This is an optimization problem. Deeper models are harder to optimize, and in particular don’t learn identity functions to emulate shallow models.

**Solution:** Change the network so learning identity functions with extra layers is easy!

Residual Networks

**Solution:** Change the network so learning identity functions with extra layers is easy!

Residual Networks

**Solution**: Change the network so learning identity functions with extra layers is easy!

If you set these to 0, the whole block will compute the identity function!

Residual Networks

A residual network is a stack of many residual blocks.

Regular design, like VGG: each residual block has two 3x3 conv.

Network is divided into stages: the first block of each stage halves the resolution (with stride-2 conv) and doubles the number of channels.

ImageNet Classification Challenge

<table>
<thead>
<tr>
<th>Year</th>
<th>Error Rate</th>
<th>Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>28.2</td>
<td>Shallow</td>
</tr>
<tr>
<td>2011</td>
<td>25.8</td>
<td>8 Layer</td>
</tr>
<tr>
<td>2012</td>
<td>16.4</td>
<td>8 Layer</td>
</tr>
<tr>
<td>2013</td>
<td>11.7</td>
<td>19 Layer</td>
</tr>
<tr>
<td>2014</td>
<td>7.3</td>
<td>22 Layer</td>
</tr>
<tr>
<td>2014</td>
<td>6.7</td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td>3.6</td>
<td>152 Layer</td>
</tr>
</tbody>
</table>

Contributors:
- Lin et al
- Sanchez & Perronnin
- Krizhevsky et al (AlexNet)
- Zeiler & Fergus
- Simonyan & Zisserman (VGG)
- Szegedy et al (GoogLeNet)
- He et al (ResNet)
ImageNet Classification Challenge

Error Rate

2010: Lin et al  
2011: Sanchez & Perronnin (AlexNet)  
2012: Krizhevsky et al (AlexNet)  
2013: Zeller & Fergus  
2014: Simonyan & Zisserman (VGG)  
2014: Szegedy et al (GoogLeNet)  
2015: He et al (ResNet)  
2016: Shao et al  
2017: Hu et al (SENet)

Shallow

8 Layer

152 Layer

152 Layer

152 Layer

19 Layer

22 Layer

28.2

25.8

16.4

11.7

7.3

6.7

3.6

3

2.3
ImageNet Classification Challenge

Error Rate


28.2 25.8 16.4 11.7 7.3 6.7 3.6 3 2.3 5.1

Shallow

Lin et al, Sanchez & Perronnin, Krizhevsky et al (AlexNet), Zeiler & Fergus, Simonyan & Zisserman (VGG), Szegedy et al (GoogLeNet), He et al (ResNet), Shao et al, Hu et al (SENet), Russakovsky et al

Justin Johnson & David Fouhey  EECS 442 WI 2021: Lecture 16 - 66  March 16, 2021
Training Convolutional Networks

1. Download big datasets
2. Design CNN architecture
3. Initialize Weights
4. For $t = 1$ to $T$:
   1. Form minibatch
   2. Compute loss + gradient
   3. Update Weights
5. Apply trained model to task
Training Convolutional Networks

1. Download big datasets
2. Design CNN architecture
3. Initialize Weights
4. For $t = 1$ to $T$:
   1. Form minibatch
   2. Compute loss + gradient
   3. Update Weights
5. Apply trained model to task
Forward pass for a 6-layer net with hidden size 4096

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```
Weight Initialization: Activation Statistics

Forward pass for a 6-layer net with hidden size 4096

```python
import numpy as np

# Layer dimensions
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
x = np.tanh(x.dot(W))
hs.append(x)
```

All activations tend to zero for deeper network layers

**Q:** What do the gradients \(\frac{dL}{dW}\) look like?
Weight Initialization: Activation Statistics

Forward pass for a 6-layer net with hidden size 4096

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

All activations tend to zero for deeper network layers

**Q:** What do the gradients \(dL/dW\) look like?

**A:** All zero, no learning =(

Weights are **too small** at initialization!
Weight Initialization: Activation Statistics

Increase scale of weights at initialization 0.01 -> 0.05

dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
Weight Initialization: Activation Statistics

Increase scale of weights at initialization $0.01 \rightarrow 0.05$

dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)

All activations saturate

Q: What do the gradients look like?

Weights are too big at initialization!
Weight Initialization: Activation Statistics

Increase scale of weights at initialization 0.01 -> 0.05

dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)

Q: What do the gradients look like?
A: Local gradients all zero, no learning =(  

Weights are too big at initialization!
Weight Initialization: Xavier

"Xavier" initialization: \( \text{std} = \frac{1}{\sqrt{\text{Din}}} \)

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])

for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010
Weight Initialization: Xavier

“Xavier” initialization: \( \text{std} = \frac{1}{\sqrt{D_{in}}} \)

definitions:
- \( \text{dims} = [4096] \times 7 \)
- \( \text{hs} = [\] \)
- \( \text{x} = \text{np.random.randn}(16, \text{dims}[0]) \)

for \( \text{Din}, \text{Dout} \) in zip(dims[:-1], dims[1:]):
    \( \text{W} = \text{np.random.randn}(	ext{Din}, \text{Dout}) / \text{np.sqrt}(\text{Din}) \)
    \( \text{x} = \text{np.tanh}(\text{x}.\text{dot}(\text{W})) \)
    \( \text{hs}.\text{append}(\text{x}) \)

Weights are just right at initialization!

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010
Weight Initialization: Xavier

“Xavier” initialization: \( \text{std} = 1 / \sqrt{\text{Din}} \)

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

For conv layers, DIN is \( \text{kernel_size}^2 \times \text{input_channels} \)

Weights are **just right** at initialization!

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010
Weight Initialization: MSRA

For ReLU networks: std = 2 / sqrt(Din)

dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
hs.append(x)

"Just right" – activations nicely scaled for all layers

Weights are just right at initialization!

Training Convolutional Networks

1. Download big datasets
2. Design CNN architecture
3. Initialize Weights
4. For \( t = 1 \) to \( T \):
   1. Form minibatch
   2. Compute loss + gradient
   3. Update Weights
5. Apply trained model to task
Training Convolutional Networks

1. Download big datasets
2. Design CNN architecture
3. Initialize Weights
4. For $t = 1$ to $T$:
   1. Form minibatch
   2. Compute loss + gradient
   3. Update Weights
5. Apply trained model to task

If the model is big, won’t we overfit?
Regularizing CNNs: Weight Decay

\[ L_{reg} = \frac{1}{2} \sum_{\ell} \|W_{\ell}\|^2 \quad \frac{\partial L_{reg}}{\partial W_{\ell}} = W_{\ell} \]

Add L2 regularization term \( L_{reg} \) to the loss penalizing large weight matrices

Usually don’t regularize bias terms, or BatchNorm scale / shift params

*Technical note: Adding an explicit term to the loss is “L2 Regularization”; “Weight decay” adds a term to the gradient. They are equivalent for SGD, but not quite the same for other optimizers like Adam*
Regularizing CNNs: Data Augmentation

Hippo
Hippo?
Hippo?
Hippo?

Horizontal Flip
Color Jitter
Image Cropping
Regularizing CNNs: Data Augmentation

Apply random transformations to input images during training
Artificially “inflate” the size of your dataset

Training Image

Augmented images

Hippo

Hippo

EECS 442 WI 2021: Lecture 16 -  83
Training Convolutional Networks

1. Download big datasets
2. Design CNN architecture
3. Initialize Weights
4. For \( t = 1 \) to \( T \):
   1. Form minibatch
   2. Compute loss + gradient
   3. Update Weights
5. Apply trained model to task

If the model is big, won’t we overfit?
Training Convolutional Networks

1. Download big datasets
2. Design CNN architecture
3. Initialize Weights
4. For $t = 1$ to $T$:
   1. Form minibatch
   2. Compute loss + gradient
   3. Update Weights
5. Apply trained model to task

What if we can’t find one?
Transfer Learning: Feature Extraction

1. Train on ImageNet

Transfer Learning: Feature Extraction

1. Train on ImageNet

2. CNN as feature extractor

Use your small dataset to train a linear classifier on top of pretrained CNN features

Transfer Learning: Fine-Tuning

1. Train on ImageNet
   - FC-1000
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-64
   - Conv-64
   - Conv-64
   - Image

2. CNN as feature extractor
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-64
   - Conv-64
   - Conv-64
   - Image

3. Bigger dataset: Fine-Tuning
   - FC
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-64
   - Conv-64
   - Conv-64
   - Image

   - Freeze these
   - Remove last layer

Reinitialize last layer and continue training whole network on your dataset.
Transfer Learning: Fine-Tuning

1. Train on ImageNet

2. CNN as feature extractor

3. Bigger dataset: **Fine-Tuning**

   Reinitialize last layer and continue training whole network on your dataset

Some tricks:
- Train with feature extraction first before fine-tuning
- Lower the learning rate: use ~1/10 of LR used in original training
- Sometimes freeze lower layers to save computation
Recap: Convolutional Networks

Fully-Connected Layers

$$y = Wx + b$$

Activation Function

$$y = \max(0, x)$$

Convolution Layers

Pooling Layers

Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$
Recap: CNN Architectures

<table>
<thead>
<tr>
<th>Year</th>
<th>Error Rate</th>
<th>8 Layer</th>
<th>152 Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>28.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>25.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>16.4</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td></td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>7.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>6.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td>3.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2016</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>2.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>5.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- AlexNet
- VGG
- ResNet
- Human

Lin et al, Sanchez & Perronnin, Krizhevsky et al (AlexNet), Zeiler & Fergus, Simonyan & Zisserman (VGG), Szegedy et al (GoogLeNet), He et al (ResNet), Shao et al, Hu et al (SENet), Russakovsky et al
Recap: Training CNNs

1. Download big datasets  
2. Design CNN architecture  
3. Initialize Weights  
4. For $t = 1$ to $T$:  
   1. Form minibatch  
   2. Compute loss + gradient  
   3. Update Weights  
5. Apply trained model to task
So Far: Image Classification

Cat image is CC0 public domain
What about **Localizing** Objects?
Next time: Detection + Segmentation