Lecture 6 (Part 1): CNN Architectures
Recap: Feature Extractors

Feature Extraction

10 numbers giving scores for classes

training

Recap: Convolution

32x32x3 image

3x3x3 filter $w$

Padding:
Preserve input spatial dimensions in output activations

Stride:
Downsample output activations
3x32x32 image

Consider 6 filters, each 3x5x5

Convolution Layer

6 activation maps, each 1x28x28

Stack activations to get a 6x28x28 output image!
Recap: Convolution Layer

3x32x32 image

Don’t forget bias terms!

Convolution Layer

6x3x5x5 filters

Stack activations to get a 6x28x28 output image!

6 activation maps, each 1x28x28

Slide inspiration: Justin Johnson
Recap: Convolution Layer

3x32x32 image

Don’t forget bias terms!

Convolution Layer

6x3x5x5 filters

Stack activations to get a 6x28x28 output image!

6 activation maps, each 1x28x28

Activation Function!

(ReLU)
Recap: Pooling Layer

Single depth slice

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Pool with 2x2 filters and stride 2

Max Pooling

6 8
3 4

Average Pooling

3.25 5.25
2 2
Recap: Putting it All Together
Components of CNNs

Convolution Layers

Pooling Layers

Fully-Connected Layers

Activation Function
Components of CNNs

- **Convolution Layers**
- **Pooling Layers**
- **Fully-Connected Layers**

**Components of CNNs**

- Activation Function
- Normalization

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

Consider a single layer $y = WX$

The following could lead to tough optimization:
- Inputs $x$ are not centered around zero (need large bias)
- Inputs $x$ have different scaling per-element (entries in $W$ will need to vary a lot)

Idea: force inputs to be “nicely scaled” at each layer!
Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

\[
\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}
\]

this is a vanilla differentiable function...
Batch Normalization

**Input:** $x: N \times D$

- Per-channel mean, shape is $D$
  \[
  \mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}
  \]

- Per-channel var, shape is $D$
  \[
  \sigma^2_j = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
  \]

- Normalized $x$, Shape is $N \times D$
  \[
  \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma^2_j + \varepsilon}}
  \]

[ioffe and Szegedy, 2015]
Batch Normalization

**Input:** $x: N \times D$

- **Per-channel mean,** shape is D
  \[
  \mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}
  \]

- **Per-channel var,** shape is D
  \[
  \sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
  \]

- **Normalized x,** Shape is N x D
  \[
  \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}
  \]

**Problem:** What if zero-mean, unit variance is too hard of a constraint?

[ioffe and Szegedy, 2015]
Batch Normalization

**Input:** \( x : N \times D \)

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j} \quad \text{Per-channel mean, shape is D}
\]

**Learnable scale and shift parameters:**

\( \gamma, \beta : D \)

\[
\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is D}
\]

\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \quad \text{Normalized x, Shape is N x D}
\]

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

\[
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is N x D}
\]
Batch Normalization: Test-Time

Input: \( x : N \times D \)

Learnable scale and shift parameters:
\( \gamma, \beta : D \)

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

Estimates depend on minibatch; can’t do this at test-time!

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j} \quad \text{Per-channel mean, shape is D}
\]

\[
\sigma^2_j = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is D}
\]

\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma^2_j + \varepsilon}} \quad \text{Normalized x, Shape is N x D}
\]

\[
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is N x D}
\]
Batch Normalization: Test-Time

Input: $x : N \times D$

$\mu_j = \text{(Running) average of values seen during training}$

Per-channel mean, shape is D

$\sigma^2_j = \text{(Running) average of values seen during training}$

Per-channel var, shape is D

Learnable scale and shift parameters: $\gamma, \beta : D$

$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma^2_j + \varepsilon}}$

Normalized x, Shape is $N \times D$

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

Output, Shape is $N \times D$

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

\[
\hat{x}(k) = \frac{x(k) - E[x(k)]}{\sqrt{\text{Var}[x(k)]}}
\]

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

- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as a kind of regularization during training
- Behaves differently during training and testing: this is a very common source of bugs!
Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

\[ x: N \times D \]

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

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

\[ y = \gamma(x - \mu) / \sigma + \beta \]

Batch Normalization for **convolutional** networks

(Spatial Batchnorm, BatchNorm2D)

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

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

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

\[ y = \gamma(x - \mu) / \sigma + \beta \]
Layer Normalization

**Batch Normalization** for fully-connected networks

\[
\begin{align*}
x &: \mathbb{R}^N \times D \\
\mu, \sigma &: \mathbb{R}^1 \times D \\
\gamma, \beta &: \mathbb{R}^1 \times D \\
y &= \gamma (x - \mu) / \sigma + \beta
\end{align*}
\]

Layer Normalization for fully-connected networks

Same behavior at train and test!
Can be used in recurrent networks

\[
\begin{align*}
x &: \mathbb{R}^N \times D \\
\mu, \sigma &: \mathbb{R}^N \times 1 \\
\gamma, \beta &: \mathbb{R}^1 \times D \\
y &= \gamma (x - \mu) / \sigma + \beta
\end{align*}
\]

Instance Normalization

Batch Normalization for convolutional networks

\[ \mathbf{x} : N \times C \times H \times W \]
Normalize
\[ \mathbf{\mu}, \mathbf{\sigma} : 1 \times C \times 1 \times 1 \]
\[ \gamma, \beta : 1 \times C \times 1 \times 1 \]
\[ y = \gamma (\mathbf{x} - \mathbf{\mu}) / \mathbf{\sigma} + \beta \]

Instance Normalization for convolutional networks
Same behavior at train / test!

\[ \mathbf{x} : N \times C \times H \times W \]
Normalize
\[ \mathbf{\mu}, \mathbf{\sigma} : N \times C \times 1 \times 1 \]
\[ \gamma, \beta : 1 \times C \times 1 \times 1 \]
\[ y = \gamma (\mathbf{x} - \mathbf{\mu}) / \mathbf{\sigma} + \beta \]

Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017
Comparison of Normalization Layers

Wu and He, “Group Normalization”, ECCV 2018
Group Normalization

Batch Norm

Layer Norm

Instance Norm

Group Norm

Wu and He, “Group Normalization”, ECCV 2018
Components of CNNs

Convolution Layers
Pooling Layers
Fully-Connected Layers

Activation Function

\[ x_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \]

Question: How should we put them together?

Fei-Fei Li, Ehsan Adeli, Zane Durante
Lecture 6 - 25
April 17, 2024
Today: CNN Architectures

Case Studies
- AlexNet
- VGG
- ResNet

Also....
- GoogLeNet
- ZFNet
- SENet
- Wide ResNet
- ResNeXT
- DenseNet
- MobileNets
- NASNet
- EfficientNet
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- 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

- 2010: 28.2
- 2011: 25.8
- 2012: 16.4
- 2013: 11.7
- 2014: 7.3
- 2014: 6.7
- 2015: 3.6
- 2016: 3
- 2017: 2.3
- 2018: 5.1

- Shallow: 8 layers
- 19 layers
- 22 layers
- 152 layers
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- **Lin et al** (28.2) - 2010
- **Sanchez & Perronnin** (25.8) - 2011
- **Krizhevsky et al (AlexNet)** (16.4) - 2012
- **Zeiler & Fergus** (11.7) - 2013
- **Simonyan & Zisserman (VGG)** (7.3) - 2014
- **Szegedy et al (GoogLeNet)** (6.7) - 2014
- **He et al (ResNet)** (3.6) - 2015
- **Shao et al** (3.0) - 2016
- **Hu et al (SENet)** (2.3) - 2017
- **Russakovsky et al** (5.1) - Human

The **first CNN-based winner** was **Krizhevsky et al (AlexNet)** with 8 layers in 2012. The subsequent winners include **Zeiler & Fergus** (19 layers) in 2013, **Simonyan & Zisserman (VGG)** (22 layers) in 2014, **He et al (ResNet)** (152 layers) in 2015, and **Hu et al (SENet)** (152 layers) in 2017.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

**First layer** (CONV1): 96 11x11 filters applied at stride 4

=>

Q: what is the output volume size? Hint: \((227-11)/4+1 = 55\)

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

=>

Output volume [55x55x96]
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

**First layer** (CONV1): 96 11x11 filters applied at stride 4

=>

Output volume [*55x55x96*]

Q: What is the total number of parameters in this layer?
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

**First layer** (CONV1): 96 11x11 filters applied at stride 4

=>

Output volume [55x55x96]

Parameters: \((11 \times 11 \times 3 + 1) \times 96 = 35K\)

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2

Q: what is the output volume size? Hint: (55-3)/2+1 = 27

W' = (W - F + 2P) / S + 1

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96

Q: what is the number of parameters in this layer?

\[ W' = \frac{(W - F + 2P)}{S} + 1 \]
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images
After CONV1: 55x55x96

**Second layer** (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96
Parameters: 0!

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images
After CONV1: 55x55x96
After POOL1: 27x27x96
...

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0

[27x27x96] MAX POOL1: 3x3 filters at stride 2

[27x27x96] NORM1: Normalization layer

[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2

[13x13x256] MAX POOL2: 3x3 filters at stride 2

[13x13x256] NORM2: Normalization layer

[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1

[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1

[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used LRN layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- **Lin et al.** (2010) - 28.2%
- **Sanchez & Perronnin** (2011) - 25.8%
- **Krizhevsky et al. (AlexNet)** (2012) - 16.4%
  - **First CNN-based winner**
  - 8 layers
- **Zeiler & Fergus** (2013) - 11.7%
- **Simonyan & Zisserman (VGG)** (2014) - 7.3%
- **Szegedy et al. (GoogLeNet)** (2014) - 6.7%
- **He et al. (ResNet)** (2015) - 3.6%
- **Shao et al.** (2016) - 3%
- **Hu et al. (SENet)** (2017) - 2.3%
- **Russakovsky et al.** (Human) - 5.1%

**Layer Counts:**
- **AlexNet** (2012): 8 layers
- **SENet** (2017): 152 layers

**Yearly Improvement:**
- 2010-2017: Linear increase from 8 layers to 152 layers.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- Sanchez & Perronnin (2011)
- Zeiler & Fergus (2013)
- Simonyan & Zisserman (VGG) (2014)
- Szegedy et al (GoogLeNet) (2014)
- Shao et al (2016)

Deeper Networks:
- 152 layers
- 152 layers
- 152 layers

Shallow Networks:
- 8 layers
- 8 layers

Architecture Depth:
- 2010: Lin et al
- 2011: Sanchez & Perronnin
- 2012: Krizhevsky et al
- 2013: Zeiler & Fergus
- 2014: Simonyan & Zisserman
- 2014: Szegedy et al
- 2015: He et al
- 2016: Shao et al
- 2017: Hu et al
- Human
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13
(ZFNet)
-> 7.3% top 5 error in ILSVRC'14

[Simonyan and Zisserman, 2014]
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

AlexNet

VGG16

VGG19
**Case Study: VGGNet**  
*Simonyan and Zisserman, 2014*

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Filter Size</th>
<th>Receptive Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>7x7 conv</td>
<td></td>
<td>Effective Receptive Field</td>
</tr>
<tr>
<td>3x3 conv</td>
<td></td>
<td>3x3 conv (stride 1)</td>
</tr>
<tr>
<td>3x3 conv</td>
<td></td>
<td>3x3 conv (stride 1)</td>
</tr>
<tr>
<td>3x3 conv</td>
<td></td>
<td>3x3 conv (stride 1)</td>
</tr>
</tbody>
</table>

**AlexNet**

**VGG16**

**VGG19**
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?

VGG16

VGG19

Conv1 (3x3)  Conv2 (3x3)  Conv3 (3x3)
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?
Case Study: VGGNet
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

[7x7]
Case Study: VGGNet
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters: $3 \times (3^2C^2)$ vs. $7^2C^2$ for C channels per layer
INPUT: [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000
INPUT: [224x224x3] memory: 224*224*3 = 150K params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: 224*224*64 = 3.2M params: (3*3*3)*64 = 1,728

CONV3-64: [224x224x64] memory: 224*224*64 = 3.2M params: (3*3*64)*64 = 36,864

POOL2: [112x112x64] memory: 112*112*64 = 800K params: 0

CONV3-128: [112x112x128] memory: 112*112*128 = 1.6M params: (3*3*64)*128 = 73,728

CONV3-128: [112x112x128] memory: 112*112*128 = 1.6M params: (3*3*128)*128 = 147,456

POOL2: [56x56x128] memory: 56*56*128 = 400K params: 0

CONV3-256: [56x56x256] memory: 56*56*256 = 800K params: (3*3*128)*256 = 294,912

CONV3-256: [56x56x256] memory: 56*56*256 = 800K params: (3*3*256)*256 = 589,824

CONV3-256: [56x56x256] memory: 56*56*256 = 800K params: (3*3*256)*256 = 589,824

POOL2: [28x28x256] memory: 28*28*256 = 200K params: 0

CONV3-512: [28x28x512] memory: 28*28*512 = 400K params: (3*3*256)*512 = 1,179,648

CONV3-512: [28x28x512] memory: 28*28*512 = 400K params: (3*3*512)*512 = 2,359,296

CONV3-512: [28x28x512] memory: 28*28*512 = 400K params: (3*3*512)*512 = 2,359,296

POOL2: [14x14x512] memory: 14*14*512 = 100K params: 0

CONV3-512: [14x14x512] memory: 14*14*512 = 100K params: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14*14*512 = 100K params: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14*14*512 = 100K params: (3*3*512)*512 = 2,359,296

POOL2: [7x7x512] memory: 7*7*512 = 25K params: 0

FC: [1x14096] memory: 4096 params: 7*7*512*4096 = 102,760,448

FC: [1x14096] memory: 4096 params: 4096*4096 = 16,777,216

FC: [1x1100] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~ 96MB / image (for a forward pass)

TOTAL params: 138M parameters
INPUT: [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: \(224^2*224^2*64=3.2\text{M}\) params: \((3*3^3)*64 = 1,728\)

CONV3-64: [224x224x64] memory: \(224^2*224^2*64=3.2\text{M}\) params: \((3*3^64)*64 = 36,864\)

POOL2: [112x112x64] memory: 112*112*64=800K params: 0

CONV3-128: [112x112x128] memory: \(112^2*112^2*128=1.6\text{M}\) params: \((3*3^64)^2*128 = 73,728\)

CONV3-128: [112x112x128] memory: \(112^2*112^2*128=1.6\text{M}\) params: \((3*3^128)^2*128 = 147,456\)

POOL2: [56x56x128] memory: 56*56*128=400K params: 0

CONV3-256: [56x56x256] memory: \(56^2*56^2*256=800\text{K}\) params: \((3*3^128)*256 = 294,912\)

CONV3-256: [56x56x256] memory: \(56^2*56^2*256=800\text{K}\) params: \((3*3^256)*256 = 589,824\)

CONV3-256: [56x56x256] memory: \(56^2*56^2*256=800\text{K}\) params: \((3*3^256)*256 = 589,824\)

POOL2: [28x28x256] memory: 28*28*256=200K params: 0

CONV3-512: [28x28x512] memory: \(28^2*28^2*512=400\text{K}\) params: \((3*3^256)^2*512 = 1,179,648\)

CONV3-512: [28x28x512] memory: \(28^2*28^2*512=400\text{K}\) params: \((3*3^512)^2*512 = 2,359,296\)

CONV3-512: [28x28x512] memory: \(28^2*28^2*512=400\text{K}\) params: \((3*3^512)^2*512 = 2,359,296\)

POOL2: [14x14x512] memory: 14*14*512=100K params: 0

CONV3-512: [14x14x512] memory: \(14^2*14^2*512=100\text{K}\) params: \((3*3^512)^2*512 = 2,359,296\)

CONV3-512: [14x14x512] memory: \(14^2*14^2*512=100\text{K}\) params: \((3*3^512)^2*512 = 2,359,296\)

CONV3-512: [14x14x512] memory: \(14^2*14^2*512=100\text{K}\) params: \((3*3^512)^2*512 = 2,359,296\)

POOL2: [7x7x512] memory: 7*7*512=25K params: 0

FC: [1x1x4096] memory: 4096 params: \(7*7^2*512*4096 = 102,760,448\)

FC: [1x1x4096] memory: 4096 params: \(4096*4096 = 16,777,216\)

FC: [1x1x1000] memory: 1000 params: \(4096^2*1000 = 4,096,000\)

**TOTAL memory: 24M * 4 bytes \~= 96MB / image (only forward! \~*2 for bwd)**

**TOTAL params: 138M parameters**
**INPUT**: $[224 \times 224 \times 3]$  
\[\text{memory: } 224 \times 224 \times 3 = 150k \quad \text{params: } 0\]  

<table>
<thead>
<tr>
<th>Layer</th>
<th>Output</th>
<th>Memory</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV3-64: $[224 \times 224 \times 64]$</td>
<td>$224 \times 224 \times 64 = 3.2M$</td>
<td>$(3 \times 3 \times 3) \times 64 = 1,728$</td>
<td></td>
</tr>
<tr>
<td>CONV3-64: $[224 \times 224 \times 64]$</td>
<td>$224 \times 224 \times 64 = 3.2M$</td>
<td>$(3 \times 3 \times 64) \times 64 = 36,864$</td>
<td></td>
</tr>
<tr>
<td>POOL2: $[112 \times 112 \times 64]$</td>
<td>$112 \times 112 \times 64 = 800k$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>CONV3-128: $[112 \times 112 \times 128]$</td>
<td>$112 \times 112 \times 128 = 1.6M$</td>
<td>$(3 \times 3 \times 64) \times 128 = 73,728$</td>
<td></td>
</tr>
<tr>
<td>CONV3-128: $[112 \times 112 \times 128]$</td>
<td>$112 \times 112 \times 128 = 1.6M$</td>
<td>$(3 \times 3 \times 128) \times 128 = 147,456$</td>
<td></td>
</tr>
<tr>
<td>POOL2: $[56 \times 56 \times 128]$</td>
<td>$56 \times 56 \times 128 = 800k$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>CONV3-256: $[56 \times 56 \times 256]$</td>
<td>$56 \times 56 \times 256 = 800k$</td>
<td>$(3 \times 3 \times 128) \times 256 = 294,912$</td>
<td></td>
</tr>
<tr>
<td>CONV3-256: $[56 \times 56 \times 256]$</td>
<td>$56 \times 56 \times 256 = 800k$</td>
<td>$(3 \times 3 \times 256) \times 256 = 589,824$</td>
<td></td>
</tr>
<tr>
<td>CONV3-256: $[56 \times 56 \times 256]$</td>
<td>$56 \times 56 \times 256 = 800k$</td>
<td>$(3 \times 3 \times 256) \times 256 = 589,824$</td>
<td></td>
</tr>
<tr>
<td>POOL2: $[28 \times 28 \times 256]$</td>
<td>$28 \times 28 \times 256 = 200k$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: $[28 \times 28 \times 512]$</td>
<td>$28 \times 28 \times 512 = 400k$</td>
<td>$(3 \times 3 \times 256) \times 512 = 1,179,648$</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: $[28 \times 28 \times 512]$</td>
<td>$28 \times 28 \times 512 = 400k$</td>
<td>$(3 \times 3 \times 512) \times 512 = 2,359,296$</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: $[28 \times 28 \times 512]$</td>
<td>$28 \times 28 \times 512 = 400k$</td>
<td>$(3 \times 3 \times 512) \times 512 = 2,359,296$</td>
<td></td>
</tr>
<tr>
<td>POOL2: $[14 \times 14 \times 512]$</td>
<td>$14 \times 14 \times 512 = 100k$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: $[14 \times 14 \times 512]$</td>
<td>$14 \times 14 \times 512 = 100k$</td>
<td>$(3 \times 3 \times 512) \times 512 = 2,359,296$</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: $[14 \times 14 \times 512]$</td>
<td>$14 \times 14 \times 512 = 100k$</td>
<td>$(3 \times 3 \times 512) \times 512 = 2,359,296$</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: $[14 \times 14 \times 512]$</td>
<td>$14 \times 14 \times 512 = 100k$</td>
<td>$(3 \times 3 \times 512) \times 512 = 2,359,296$</td>
<td></td>
</tr>
<tr>
<td>POOL2: $[7 \times 7 \times 512]$</td>
<td>$7 \times 7 \times 512 = 25k$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>FC: $[1 \times 1 \times 4096]$</td>
<td>$4096$</td>
<td>$7 \times 7 \times 512 \times 4096 = 102,760,448$</td>
<td></td>
</tr>
<tr>
<td>FC: $[1 \times 1 \times 4096]$</td>
<td>$4096$</td>
<td>$4096 \times 4096 = 16,777,216$</td>
<td></td>
</tr>
<tr>
<td>FC: $[1 \times 1 \times 1000]$</td>
<td>$1000$</td>
<td>$4096 \times 1000 = 4,096,000$</td>
<td></td>
</tr>
</tbody>
</table>

**TOTAL memory**: $24M \times 4 \text{ bytes} \approx 96MB / \text{image}$  
**TOTAL params**: $138M \text{ parameters}$
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Details:
- ILSVRC’14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks

AlexNet
VGG16
VGG19
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>Lin et al</td>
<td>shallow</td>
</tr>
<tr>
<td>2011</td>
<td>Sanchez &amp; Perronnin</td>
<td>8 layers</td>
</tr>
<tr>
<td>2012</td>
<td>Krizhevsky et al (AlexNet)</td>
<td>8 layers</td>
</tr>
<tr>
<td>2013</td>
<td>Zeiler &amp; Fergus</td>
<td>8 layers</td>
</tr>
<tr>
<td>2014</td>
<td>Simonyan &amp; Zisserman (VGG)</td>
<td>19 layers</td>
</tr>
<tr>
<td>2014</td>
<td>Szegedy et al (GoogLeNet)</td>
<td>22 layers</td>
</tr>
<tr>
<td>2015</td>
<td>He et al (ResNet)</td>
<td>152 layers</td>
</tr>
<tr>
<td>2016</td>
<td>Shao et al</td>
<td>152 layers</td>
</tr>
<tr>
<td>2017</td>
<td>Hu et al (SENet)</td>
<td>152 layers</td>
</tr>
<tr>
<td>2018</td>
<td>Russakovsky et al</td>
<td>Human</td>
</tr>
</tbody>
</table>

Human performance is the lowest, indicating that deep learning models outperform humans in image recognition tasks.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

“Revolution of Depth”

- Lin et al (2010, 28.2)
- Sanchez & Perronnin (2011, 25.8)
- Krizhevsky et al (AlexNet) (2012, 16.4)
- Zeiler & Fergus (2013, 11.7)
- Simonyan & Zisserman (VGG) (2014, 7.3)
- Szegedy et al (GoogLeNet) (2014, 6.7)
- He et al (ResNet) (2015, 3.6)
- Shao et al (2016, 3)
- Hu et al (SENet) (2017, 2.3)
- Russakovsky et al (2018, 5.1)

Shallow: 8 layers
8 layers
19 layers
22 layers
152 layers
152 layers
152 layers
Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC’15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC’15 and COCO’15!
Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?
Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

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

![Graph showing test and training error over iterations for 20-layer and 56-layer ResNet models.](image)
Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

56-layer model performs worse on both test and training error

-> The deeper model performs worse, but it’s not caused by overfitting!
Case Study: ResNet

[He et al., 2015]

Fact: Deep models have more representation power (more parameters) than shallower models.

Hypothesis: the problem is an *optimization* problem, **deeper models are harder to optimize**
Case Study: ResNet

[He et al., 2015]

Fact: Deep models have more representation power (more parameters) than shallower models.

Hypothesis: the problem is an optimization problem, deeper models are harder to optimize

What should the deeper model learn to be at least as good as the shallower model?

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.
Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

![Diagram of ResNet layers: X, conv, relu, conv, H(x)]
Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

```
H(x) = F(x) + x

Identity mapping: H(x) = x if F(x) = 0
```

"Plain" layers

Residual block
Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping.

Identity mapping: \( H(x) = x \) if \( F(x) = 0 \)

Use layers to fit residual \( F(x) = H(x) - x \) instead of \( H(x) \) directly.
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers

\[
F(x) + x
\]

Residual block

\[
F(x)
\]

relu

3x3 conv

relu

3x3 conv

identity
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
Reduce the activation volume by half.
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning (stem)

![ResNet Architecture Diagram](image)
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning (stem)
- No FC layers at the end (only FC 1000 to output classes)
- (In theory, you can train a ResNet with input image of variable sizes)
Case Study: ResNet

[He et al., 2015]

Total depths of 18, 34, 50, 101, or 152 layers for ImageNet
Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)
Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)

For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)

1x1 conv, 256 filters projects back to 256 feature maps (28x28x256)

3x3 conv operates over only 64 feature maps

1x1 conv, 64 filters to project to 28x28x64
Case Study: ResNet
[He et al., 2015]

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

[He et al., 2015]
Case Study: ResNet

[He et al., 2015]

Experimental Results
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lower training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: “Ultra-deep” (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd
Case Study: ResNet

[He et al., 2015]

Experimental Results
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lower training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

ILSVRC 2015 classification winner (3.6% top 5 error) -- better than “human performance”! (Russakovsky 2014)
Summary: CNN Architectures

Case Studies
- AlexNet
- VGG
- ResNet

Also....
- ZFNet
- GoogLeNet
- SENet
- Wide ResNet
- ResNeXt
- DenseNet
- MobileNets
- NASNet
Main takeaways

**AlexNet** showed that you can use CNNs to train Computer Vision models.
**VGG** shows that bigger networks work better.
**ResNet** showed us how to train extremely deep networks:
- Limited only by GPU & memory!
- Showed diminishing returns as networks got bigger

After ResNet: CNNs were better than the human metric and focus shifted to other topics:
- **Efficient Networks**: **MobileNet**, **ShuffleNet**
- **Neural Architecture Search** can now automate architecture design
Summary: CNN Architectures

- Many popular architectures are available in model zoos.
- ResNets are good defaults to use.
  True for > 8 years!
- Networks have gotten increasingly deep over time.
- Many other aspects of network architectures are also continuously being investigated and improved.
Transfer learning
You need a lot of data if you want to train/use CNNs?
Transfer Learning with CNNs
Transfer Learning with CNNs

AlexNet:
64 x 3 x 11 x 11

(More on this in Lecture 13)
Transfer Learning with CNNs

Test image  L2 Nearest neighbors in feature space

(More on this in Lecture 13)
Transfer Learning with CNNs

1. Train on Imagenet

Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014
Transfer Learning with CNNs

1. Train on Imagenet

2. Small Dataset (C classes)

- Freeze these
- Reinitialize this and train

Transfer Learning with CNNs

1. Train on Imagenet

2. Small Dataset (C classes)

Reinitialize this and train

 Freeze these

Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning with CNNs

1. Train on Imagenet
   - FC-1000
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-256
   - Conv-256
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-64
   - Conv-64
   - Image

2. Small Dataset (C classes)
   - FC-C
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-256
   - Conv-256
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-64
   - Conv-64
   - Image
   - Reinitialize this and train

3. Bigger dataset
   - FC-C
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-256
   - Conv-256
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-64
   - Conv-64
   - Image
   - Train these
   - Freeze these
   - With bigger dataset, train more layers
   - Freeze these
   - Lower learning rate when finetuning; 1/10 of original LR is good starting point
### More specific vs. More generic

<table>
<thead>
<tr>
<th>More specific</th>
<th>More generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv-512</td>
<td>Conv-128</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-128</td>
</tr>
<tr>
<td>Conv-512</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>MaxPool</td>
<td>MaxPool</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-64</td>
</tr>
</tbody>
</table>

### Table

<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>
<tr>
<td>More generic</td>
<td>More specific</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td>FC-1000</td>
<td>Conv-512</td>
<td></td>
</tr>
<tr>
<td>FC-4096</td>
<td>Conv-512</td>
<td></td>
</tr>
<tr>
<td>FC-4096</td>
<td>Conv-512</td>
<td></td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-512</td>
<td></td>
</tr>
<tr>
<td>Conv-256</td>
<td>Conv-512</td>
<td></td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-256</td>
<td></td>
</tr>
<tr>
<td>Conv-128</td>
<td>Conv-256</td>
<td></td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-128</td>
<td></td>
</tr>
<tr>
<td>Conv-64</td>
<td>Conv-128</td>
<td></td>
</tr>
<tr>
<td>MaxPool</td>
<td>Conv-64</td>
<td></td>
</tr>
<tr>
<td>Conv-64</td>
<td>Conv-64</td>
<td></td>
</tr>
<tr>
<td>Image</td>
<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>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><strong>More specific</strong></td>
<td><strong>More generic</strong></td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><strong>Image</strong></td>
<td><strong>FC-1000</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>Conv-64</strong></td>
<td><strong>FC-4096</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-128</strong></td>
</tr>
<tr>
<td><strong>Conv-128</strong></td>
<td><strong>Conv-512</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-256</strong></td>
</tr>
<tr>
<td><strong>Conv-256</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-512</strong></td>
<td><strong>Conv-512</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-256</strong></td>
</tr>
<tr>
<td><strong>Conv-256</strong></td>
<td><strong>MaxPool</strong></td>
</tr>
<tr>
<td><strong>Conv-128</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>Conv-64</strong></td>
<td><strong>FC-4096</strong></td>
</tr>
</tbody>
</table>

### Table

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

You’re in trouble… Try linear classifier from different stages.
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

Image Captioning: CNN + RNN

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN

Word vectors pretrained with word2vec

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Transfer learning with CNNs - Architecture matters

Object detection on MSCOCO

Transfer learning with CNNs is pervasive…
But it might not always be necessary!

Training from scratch can work just as well as training from a pretrained ImageNet model for object detection.

But it takes 2-3x as long to train.

They also find that collecting more data is better than finetuning on a related task.

He et al, "Rethinking ImageNet Pre-training", ICCV 2019
Figure copyright Kaiming He, 2019. Reproduced with permission.
Takeaway for your projects and beyond:
Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big model there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision
Next time: Training Neural Networks
Appendix – Slides from Previous Years of the Course
Review: LeNet-5

[LeCun et al., 1998]

- Conv filters were 5x5, applied at stride 1
- Subsampling (Pooling) layers were 2x2 applied at stride 2
- i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- Sanchez & Perronnin (2011) - 25.8%
- Krizhevsky et al (AlexNet) (2012) - 16.4%
- Zeiler & Fergus (2013) - 11.7%
- Simonyan & Zisserman (VGG) (2014) - 7.3%
- Szegedy et al (GoogLeNet) (2014) - 6.7%
- He et al (ResNet) (2015) - 3.6%
- Shao et al (2016) - 3%
- Hu et al (SENet) (2017) - 2.3%
- Russakovsky et al (Human) - 5.1%

ZFNet: Improved hyperparameters over AlexNet

- ZFNet: Improved hyperparameters over AlexNet (2013) - 8 layers
- Zeiler & Fergus (2013) - 8 layers
- Simonyan & Zisserman (VGG) (2014) - 19 layers
- Szegedy et al (GoogLeNet) (2014) - 22 layers
- He et al (ResNet) (2015) - 152 layers
- Shao et al (2016) - 152 layers
- Hu et al (SENet) (2017) - 152 layers
- Russakovsky et al (Human) - 152 layers
**Case Study: AlexNet**

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:
- **[227x227x3] INPUT**
- **[55x55x96] CONV1**: 96 11x11 filters at stride 4, pad 0
- **[27x27x96] MAX POOL1**: 3x3 filters at stride 2
- **[27x27x96] NORM1**: Normalization layer
- **[27x27x256] CONV2**: 256 5x5 filters at stride 1, pad 2
- **[13x13x256] MAX POOL2**: 3x3 filters at stride 2
- **[13x13x256] NORM2**: Normalization layer
- **[13x13x384] CONV3**: 384 3x3 filters at stride 1, pad 1
- **[13x13x384] CONV4**: 384 3x3 filters at stride 1, pad 1
- **[13x13x256] CONV5**: 256 3x3 filters at stride 1, pad 1
- **[6x6x256] MAX POOL3**: 3x3 filters at stride 2
- **[4096] FC6**: 4096 neurons
- **[4096] FC7**: 4096 neurons
- **[1000] FC8**: 1000 neurons (class scores)

Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

CONV1, CONV2, CONV4, CONV5: Connections only with feature maps on same GPU

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

- [227x227x3] INPUT
- [55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
- [27x27x96] MAX POOL1: 3x3 filters at stride 2
- [27x27x96] NORM1: Normalization layer
- [27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
- [13x13x256] MAX POOL2: 3x3 filters at stride 2
- [13x13x256] NORM2: Normalization layer
- [13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
- [13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
- [13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
- [6x6x256] MAX POOL3: 3x3 filters at stride 2
- [4096] FC6: 4096 neurons
- [4096] FC7: 4096 neurons
- [1000] FC8: 1000 neurons (class scores)

CONV3, FC6, FC7, FC8: Connections with all feature maps in preceding layer, communication across GPUs

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
ZFNet

[Zeiler and Fergus, 2013]

AlexNet but:
CONV1: change from (11x11 stride 4) to (7x7 stride 2)
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 16.4% -> 11.7%
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

<table>
<thead>
<tr>
<th>Year</th>
<th>Winner</th>
<th>Hyperparameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>Lin et al</td>
<td>8 layers</td>
</tr>
<tr>
<td>2011</td>
<td>Sanchez &amp; Perronnin</td>
<td>16.4</td>
</tr>
<tr>
<td>2012</td>
<td>Krizhevsky et al (AlexNet)</td>
<td>16.4</td>
</tr>
<tr>
<td>2013</td>
<td>Zeiler &amp; Fergus</td>
<td>11.7</td>
</tr>
<tr>
<td>2014</td>
<td>Simonyan &amp; Zisserman (VGG)</td>
<td>7.3</td>
</tr>
<tr>
<td>2014</td>
<td>Szegedy et al (GoogLeNet)</td>
<td>6.7</td>
</tr>
<tr>
<td>2015</td>
<td>He et al (ResNet)</td>
<td>3.6</td>
</tr>
<tr>
<td>2016</td>
<td>Shao et al</td>
<td>3</td>
</tr>
<tr>
<td>2017</td>
<td>Hu et al (SENet)</td>
<td>2.3</td>
</tr>
<tr>
<td>Human</td>
<td>Russakovsky et al</td>
<td>5.1</td>
</tr>
</tbody>
</table>

ZFNet: Improved hyperparameters over AlexNet

152 layers 152 layers 152 layers
19 layers 22 layers
Case Study: GoogLeNet

[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- ILSVRC’14 classification winner (6.7% top 5 error)
- 22 layers
- Only 5 million parameters!
  12x less than AlexNet
  27x less than VGG-16
- Efficient “Inception” module
- No FC layers
Case Study: GoogLeNet

[Szegedy et al., 2014]

“Inception module”: design a good local network topology (network within a network) and then stack these modules on top of each other.
Case Study: GoogLeNet
[ Szegedy et al., 2014 ]

Naive Inception module

Apply parallel filter operations on the input from previous layer:
- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together channel-wise
Case Study: GoogLeNet

[Szegedy et al., 2014]

Naive Inception module

Filter concatenation

Previous Layer

1x1 convolution

3x3 convolution

5x5 convolution

3x3 max pooling

Apply parallel filter operations on the input from previous layer:
- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together channel-wise

Q: What is the problem with this? [Hint: Computational complexity]
Case Study: GoogLeNet
[Szegedy et al., 2014]

Example:

Q: What is the problem with this?
[Hint: Computational complexity]
Case Study: GoogLeNet

[Szegedy et al., 2014]

Input

3x3 pool

5x5 conv,
96

3x3 conv,
192

1x1 conv,
128

Filter concatenation

Q: What is the problem with this?
[Hint: Computational complexity]

Q1: What are the output sizes of all different filter operations?

Example:

Module input:
28x28x256

Naive Inception module
Case Study: GoogLeNet
[Szegedy et al., 2014]

Example:

Q1: What are the output sizes of all different filter operations?

Example:

- Module input: 28x28x256
- 1x1 conv, 128
- 3x3 conv, 192
- 5x5 conv, 96
- 3x3 pool
- 28x28x128
- 28x28x192
- 28x28x96
- 28x28x256

Q: What is the problem with this? [Hint: Computational complexity]
Case Study: GoogLeNet
[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q2: What is output size after filter concatenation?

Module input: 28x28x256

Naive Inception module

Filter concatenation

28x28x128
1x1 conv, 128

28x28x192
3x3 conv, 192

28x28x96
5x5 conv, 96

28x28x256
3x3 pool
Case Study: GoogLeNet
[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q2: What is output size after filter concatenation?

28x28x(128+192+96+256) = 28x28x672

Naive Inception module
Case Study: GoogLeNet
[Szegedy et al., 2014]

Example:

Q2: What is output size after filter concatenation?

Filter concatenation

28x28(128+192+96+256) = 28x28x672

Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops
Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Input

3x3 pool, 5x5 conv, 96

1x1 conv, 128

Filter concatenation

28x28x128

28x28x192

28x28x96

28x28x256

28x28x(128+192+96+256) = 28x28x672

Naive Inception module

Q: What is the problem with this? [Hint: Computational complexity]

Conv Ops:

[1x1 conv, 128] 28x28x128x1x1x256

[3x3 conv, 192]

28x28x192x3x3x256

[5x5 conv, 96] 28x28x96x5x5x256

Total: 854M ops

Very expensive compute

Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!
Case Study: GoogLeNet
[Szegedy et al., 2014]

Naive Inception module

Q: What is the problem with this? [Hint: Computational complexity]

Example:

Q2: What is output size after filter concatenation?

28x28x(128+192+96+256) = 529k

Solution: “bottleneck” layers that use 1x1 convolutions to reduce feature channel size
Review: 1x1 convolutions

1x1 CONV with 32 filters
(each filter has size 1x1x64, and performs a 64-dimensional dot product)
Review: 1x1 convolutions

1x1 CONV with 32 filters
(each filter has size 1x1x64, and performs a 64-dimensional dot product)

Alternatively, interpret it as applying the same FC layer on each input pixel.
Review: 1x1 convolutions

Alternatively, interpret it as applying the same FC layer on each input pixel

1x1 CONV with 32 filters

preserves spatial dimensions, reduces depth!

Projects depth to lower dimension (combination of feature maps)
Case Study: GoogLeNet

[Szegedy et al., 2014]

Naive Inception module

Inception module with dimension reduction
Case Study: GoogLeNet
[Szegedy et al., 2014]

Naive Inception module

Inception module with dimension reduction

1x1 conv “bottleneck” layers

Fei-Fei Li, Ehsan Adeli, Zane Durante
Lecture 6 - 125
April 17, 2024
Case Study: GoogLeNet
[Szegedy et al., 2014]

Inception module with dimension reduction

Using same parallel layers as naive example, and adding “1x1 conv, 64 filter” bottlenecks:

**Conv Ops:**
- [1x1 conv, 64] 28x28x64x1x1x256
- [1x1 conv, 64] 28x28x64x1x1x256
- [1x1 conv, 128] 28x28x128x1x1x256
- [3x3 conv, 192] 28x28x192x3x3x64
- [5x5 conv, 96] 28x28x96x5x5x64
- [1x1 conv, 64] 28x28x64x1x1x256

**Total: 358M ops**

Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer
Case Study: GoogLeNet

[Szegedy et al., 2014]

Stack Inception modules with dimension reduction on top of each other
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Stem Network: Conv-Pool-2x Conv-Pool
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Stacked Inception Modules
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Classifier output
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Note: after the last convolutional layer, a global average pooling layer is used that spatially averages across each feature map, before final FC layer. No longer multiple expensive FC layers!

Classifier output
Case Study: GoogLeNet  
[Szegedy et al., 2014]

Full GoogLeNet architecture

Auxiliary classification outputs to inject additional gradient at lower layers (AvgPool-1x1Conv-FC-FC-Softmax)
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

22 total layers with weights
(parallel layers count as 1 layer => 2 layers per Inception module. Don’t count auxiliary output layers)
Case Study: GoogLeNet

[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- Avoids expensive FC layers
- 12x less params than AlexNet
- 27x less params than VGG-16
- ILSVRC’14 classification winner (6.7% top 5 error)
Comparing complexity...


Comparing complexity...

Inception-v4: Resnet + Inception!


Comparing complexity...


Comparing complexity...


Comparing complexity...


AlexNet:
Smaller compute, still memory heavy, lower accuracy

Comparing complexity...

ResNet:
Moderate efficiency depending on model, highest accuracy


ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- Sanchez & Perronnin (2011)
- Zeiler & Fergus (2013)
- Simonyan & Zisserman (VGG) (2014)
- Szegedy et al (GoogLeNet) (2014)
- Shao et al (2016)

Shallow: 8 layers
8 layers
19 layers
22 layers
152 layers
152 layers
152 layers

Adaptive feature map reweighting
Squeeze-and-Excitation Networks (SENet)

[Hu et al. 2017]

- Add a “feature recalibration” module that learns to adaptively reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC’17 classification winner (using ResNeXt-152 as a base architecture)
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- 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

- 2010: 28.2
- 2011: 25.8
- 2012: 16.4
- 2013: 11.7
- 2014: 7.3
- 2014: 6.7
- 2015: 3.6
- 2016: 3
- 2017: 2.3
- Human: 5.1

Shallow: 8 layers
19 layers
22 layers
152 layers
152 layers
152 layers
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

Completion of the challenge:
Annual ImageNet competition no longer held after 2017 -> now moved to Kaggle.
But research into CNN architectures is still flourishing
Improving ResNets...

Identity Mappings in Deep Residual Networks

[He et al. 2016]

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network
- Gives better performance
Improving ResNets...

Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- User wider residual blocks (F x k filters instead of F filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)
Improving ResNets...
Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways ("cardinality")
- Parallel pathways similar in spirit to Inception module
Other ideas...

Densely Connected Convolutional Networks (DenseNet)  
[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse
- Showed that shallow 50-layer network can outperform deeper 152 layer ResNet
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- Sanchez & Perronnin (2011)
- Zeiler & Fergus (2013)
- Simonyan & Zisserman (VGG) (2014)
- Szegedy et al (GoogLeNet) (2014)
- Shao et al (2016)
- Russakovsky et al (Human)

Network ensembling

- Lin et al: shallow, 8 layers
- Sanchez & Perronnin: 16.4, 8 layers
- Krizhevsky et al: 11.7, 19 layers
- Zeiler & Fergus: 7.3, 22 layers
- Simonyan & Zisserman: 6.7, 152 layers
- Szegedy et al: 3.6, 152 layers
- He et al: 3, 152 layers
- Shao et al: 2.3, 152 layers
- Hu et al: 5.1, 152 layers
- Russakovsky et al: 152 layers

Fei-Fei Li, Ehsan Adeli, Zane Durante

Lecture 6 - 150

April 17, 2024
Improving ResNets...

“Good Practices for Deep Feature Fusion”

[Shao et al. 2016]

- Multi-scale ensembling of Inception, Inception-Resnet, Resnet, Wide Resnet models
- ILSVRC’16 classification winner

<table>
<thead>
<tr>
<th>Model</th>
<th>Inception-v3</th>
<th>Inception-v4</th>
<th>Inception-Resnet-v2</th>
<th>Resnet-200</th>
<th>Wrn-68-3</th>
<th>Fusion (Val.)</th>
<th>Fusion (Test)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err. (%)</td>
<td>4.20</td>
<td>4.01</td>
<td>3.52</td>
<td>4.26</td>
<td>4.65</td>
<td>2.92 (-0.6)</td>
<td>2.99</td>
</tr>
</tbody>
</table>
Efficient networks...

MobileNets: Efficient Convolutional Neural Networks for Mobile Applications  
[Howard et al. 2017]

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018

<table>
<thead>
<tr>
<th>Standard network</th>
<th>MobileNets</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C^2HW</td>
<td>9C^2HW + C^2HW</td>
</tr>
<tr>
<td>Conv (3x3, C-&gt;C)</td>
<td>Conv (3x3, C-&gt;C, groups=C)</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>BatchNorm</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>Total compute:9C^2HW</td>
<td>Depthwise convolutions</td>
</tr>
</tbody>
</table>

Efficient networks...

MobileNets: Efficient Convolutional Neural Networks for Mobile Applications  
[Howard et al. 2017]

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018

<table>
<thead>
<tr>
<th>Standard network</th>
<th>MobileNets</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C^2HW</td>
<td>9C^2HW + C^2HW</td>
</tr>
<tr>
<td>Conv (3x3, C-&gt;C)</td>
<td>Conv (3x3, C-&gt;C, groups=C)</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>BatchNorm</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>Total compute:9C^2HW</td>
<td>Depthwise convolutions</td>
</tr>
</tbody>
</table>

Efficient networks...

MobileNets: Efficient Convolutional Neural Networks for Mobile Applications  
[Howard et al. 2017]

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018

<table>
<thead>
<tr>
<th>Standard network</th>
<th>MobileNets</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C^2HW</td>
<td>9C^2HW + C^2HW</td>
</tr>
<tr>
<td>Conv (3x3, C-&gt;C)</td>
<td>Conv (3x3, C-&gt;C, groups=C)</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>BatchNorm</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>Total compute:9C^2HW</td>
<td>Depthwise convolutions</td>
</tr>
</tbody>
</table>

Efficient networks...

MobileNets: Efficient Convolutional Neural Networks for Mobile Applications  
[Howard et al. 2017]

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018

<table>
<thead>
<tr>
<th>Standard network</th>
<th>MobileNets</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C^2HW</td>
<td>9C^2HW + C^2HW</td>
</tr>
<tr>
<td>Conv (3x3, C-&gt;C)</td>
<td>Conv (3x3, C-&gt;C, groups=C)</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>BatchNorm</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>Total compute:9C^2HW</td>
<td>Depthwise convolutions</td>
</tr>
</tbody>
</table>

Efficient networks...

MobileNets: Efficient Convolutional Neural Networks for Mobile Applications  
[Howard et al. 2017]

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018

<table>
<thead>
<tr>
<th>Standard network</th>
<th>MobileNets</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C^2HW</td>
<td>9C^2HW + C^2HW</td>
</tr>
<tr>
<td>Conv (3x3, C-&gt;C)</td>
<td>Conv (3x3, C-&gt;C, groups=C)</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>BatchNorm</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>Total compute:9C^2HW</td>
<td>Depthwise convolutions</td>
</tr>
</tbody>
</table>

Efficient networks...

MobileNets: Efficient Convolutional Neural Networks for Mobile Applications  
[Howard et al. 2017]

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018

<table>
<thead>
<tr>
<th>Standard network</th>
<th>MobileNets</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C^2HW</td>
<td>9C^2HW + C^2HW</td>
</tr>
<tr>
<td>Conv (3x3, C-&gt;C)</td>
<td>Conv (3x3, C-&gt;C, groups=C)</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>BatchNorm</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>Total compute:9C^2HW</td>
<td>Depthwise convolutions</td>
</tr>
</tbody>
</table>

Efficient networks...

MobileNets: Efficient Convolutional Neural Networks for Mobile Applications  
[Howard et al. 2017]

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018

<table>
<thead>
<tr>
<th>Standard network</th>
<th>MobileNets</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C^2HW</td>
<td>9C^2HW + C^2HW</td>
</tr>
<tr>
<td>Conv (3x3, C-&gt;C)</td>
<td>Conv (3x3, C-&gt;C, groups=C)</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>BatchNorm</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>Total compute:9C^2HW</td>
<td>Depthwise convolutions</td>
</tr>
</tbody>
</table>

Efficient networks...

MobileNets: Efficient Convolutional Neural Networks for Mobile Applications  
[Howard et al. 2017]

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018

<table>
<thead>
<tr>
<th>Standard network</th>
<th>MobileNets</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C^2HW</td>
<td>9C^2HW + C^2HW</td>
</tr>
<tr>
<td>Conv (3x3, C-&gt;C)</td>
<td>Conv (3x3, C-&gt;C, groups=C)</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>BatchNorm</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>Total compute:9C^2HW</td>
<td>Depthwise convolutions</td>
</tr>
</tbody>
</table>

Efficient networks...

MobileNets: Efficient Convolutional Neural Networks for Mobile Applications  
[Howard et al. 2017]

- Depthwise separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018

<table>
<thead>
<tr>
<th>Standard network</th>
<th>MobileNets</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C^2HW</td>
<td>9C^2HW + C^2HW</td>
</tr>
<tr>
<td>Conv (3x3, C-&gt;C)</td>
<td>Conv (3x3, C-&gt;C, groups=C)</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>BatchNorm</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>Total compute:9C^2HW</td>
<td>Depthwise convolutions</td>
</tr>
</tbody>
</table>
Learning to search for network architectures...

Neural Architecture Search with Reinforcement Learning (NAS)

[Zoph et al. 2016]

- “Controller” network that learns to design a good network architecture (output a string corresponding to network design)
- Iterate:
  1) Sample an architecture from search space
  2) Train the architecture to get a “reward” R corresponding to accuracy
  3) Compute gradient of sample probability, and scale by R to perform controller parameter update (i.e. increase likelihood of good architecture being sampled, decrease likelihood of bad architecture)
Learning to search for network architectures...

Learning Transferable Architectures for Scalable Image Recognition

[Zoph et al. 2017]

- Applying neural architecture search (NAS) to a large dataset like ImageNet is expensive
- Design a search space of building blocks (“cells”) that can be flexibly stacked
- NASNet: Use NAS to find best cell structure on smaller CIFAR-10 dataset, then transfer architecture to ImageNet
- Many follow-up works in this space e.g. AmoebaNet (Real et al. 2019) and ENAS (Pham, Guan et al. 2018)
But sometimes smart heuristic is better than NAS ...

EfficientNet: Smart Compound Scaling

[Tan and Le. 2019]

- Increase network capacity by scaling width, depth, and resolution, while balancing accuracy and efficiency.
- Search for optimal set of compound scaling factors given a compute budget (target memory & flops).
- Scale up using smart heuristic rules

\[
d = \alpha^\phi \\
w = \beta^\phi \\
r = \gamma^\phi
\]

s.t. \( \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \)

\( \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \)
Efficient networks...

https://openai.com/blog/ai-and-efficiency/