Lecture 21
Deep Learning: Initialization, Architectures, Object Detection and Other Applications

COS 429: Computer Vision

Thanks: most of these slides shamelessly adapted from Stanford CS231n: Convolutional Neural Networks for Visual Recognition Fei-Fei Li, Andrej Karpathy, Justin Johnson http://cs231n.stanford.edu/
\[ f(x, y, z) = (x + y)z \]

Example: \( x = -2, y = 5, z = -4 \)

- \( q = x + y \) \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \)

- \( f = qz \) \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \)

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:
\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x}
\]
Review: Backpropagation

\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}
\]

\[
\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}
\]

 activates

“local gradient”

gradients
Recall: Max Pooling

Question: what are the partial derivatives of a max pool layer?

Single depth slice

max pool with 2x2 filters and stride 2
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 Norm 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%
Case Study: 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: 15.4% -> 14.8%
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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

best model

11.2% top 5 error in ILSVRC 2013
->
7.3% top 5 error
TOTAL memory: 24M * 4 bytes ~ = 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
### INPUT: \([224 \times 224 \times 3]\)
- **memory:** \(224 \times 224 \times 3 = 150 \text{K}\)
- **params:** 0

### CONV3-64: \([224 \times 224 \times 64]\)
- **memory:** \(\textcolor{red}{224 \times 224 \times 64 = 3.2 \text{M}}\)
- **params:** \((3 \times 3 \times 3) \times 64 = 1,728\)

### CONV3-64: \([224 \times 224 \times 64]\)
- **memory:** \(\textcolor{red}{224 \times 224 \times 64 = 3.2 \text{M}}\)
- **params:** \((3 \times 3 \times 64) \times 64 = 36,864\)

### POOL2: \([112 \times 112 \times 64]\)
- **memory:** \(112 \times 112 \times 64 = 800 \text{K}\)
- **params:** 0

### CONV3-128: \([112 \times 112 \times 128]\)
- **memory:** \(\textcolor{red}{112 \times 112 \times 128 = 1.6 \text{M}}\)
- **params:** \((3 \times 3 \times 64) \times 128 = 73,728\)

### CONV3-128: \([112 \times 112 \times 128]\)
- **memory:** \(\textcolor{red}{112 \times 112 \times 128 = 1.6 \text{M}}\)
- **params:** \((3 \times 3 \times 128) \times 128 = 147,456\)

### POOL2: \([56 \times 56 \times 128]\)
- **memory:** \(56 \times 56 \times 128 = 400 \text{K}\)
- **params:** 0

### CONV3-256: \([56 \times 56 \times 256]\)
- **memory:** \(\textcolor{red}{56 \times 56 \times 256 = 800 \text{K}}\)
- **params:** \((3 \times 3 \times 128) \times 256 = 294,912\)

### CONV3-256: \([56 \times 56 \times 256]\)
- **memory:** \(\textcolor{red}{56 \times 56 \times 256 = 800 \text{K}}\)
- **params:** \((3 \times 3 \times 256) \times 256 = 589,824\)

### POOL2: \([28 \times 28 \times 256]\)
- **memory:** \(28 \times 28 \times 256 = 200 \text{K}\)
- **params:** 0

### CONV3-512: \([28 \times 28 \times 512]\)
- **memory:** \(\textcolor{red}{28 \times 28 \times 512 = 400 \text{K}}\)
- **params:** \((3 \times 3 \times 256) \times 512 = 1,179,648\)

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

### POOL2: \([14 \times 14 \times 512]\)
- **memory:** \(\textcolor{red}{14 \times 14 \times 512 = 100 \text{K}}\)
- **params:** 0

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

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

### POOL2: \([7 \times 7 \times 512]\)
- **memory:** \(\textcolor{red}{7 \times 7 \times 512 = 25 \text{K}}\)
- **params:** 0

### FC: \([1 \times 1 \times 4096]\)
- **memory:** 4096
- **params:** \(7 \times 7 \times 512 \times 4096 = 102,760,448\)

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

### FC: \([1 \times 1 \times 1000]\)
- **memory:** 1000
- **params:** \(4096 \times 1000 = 4,096,000\)

---

**TOTAL memory:** \(24\text{M} \times 4\text{ bytes} \approx 93\text{MB} / \text{image}\) (only forward! \(\sim 2\) for bwd)

**TOTAL params:** 138M parameters

---

**Note:**
- Most memory is in early CONV
- Most params are in late FC
Step 1: Preprocess the data

Assume $X \ [N \times D]$ is data matrix, each example in a row.

- Original data
- Zero-centered data
- Normalized data

$X = \text{np.mean}(X, \text{axis} = 0)$

$X = \text{np.std}(X, \text{axis} = 0)$
TLDR: In practice for Images: center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening
Q: what happens when W=0 init is used?
- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[ W = 0.01 \times \text{np.random.randn}(D,H) \]
- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[
W = 0.01 \times \text{np.random.randn}(D,H)
\]

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.
Let's look at some activation statistics.

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

```python
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)

act = {'relu': lambda x: np.maximum(0, x), 'tanh': lambda x: np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1]  # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01  # layer initialization
    H = np.dot(X, W)  # matrix multiply
    H = act[nonlinearities[i]](H)  # nonlinearity
    Hs[i] = H  # cache result on this layer

# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i, H in Hs.iteritems()]
layer_stds = [np.std(H) for i, H in Hs.iteritems()]
for i, H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i , H in Hs.iteritems():
    plt.subplot(1, len(Hs), i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```
Input layer had mean 0.000927 and std 0.990380
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000061 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
All activations become zero!

Q: think about the backward pass. What do the gradients look like?

Hint: think about backward pass for a W*X gate.
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

*1.0 instead of *0.01
"Xavier initialization"
[Glorot et al., 2010]

Reasonable initialization.
(Mathematical derivation assumes linear activations)
but when using the ReLU nonlinearity it breaks.
input layer had mean 0.00001 and std 0.999444
hidden layer 1 had mean 0.502488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.013655
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.060035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523

He et al., 2015
(note additional /2)
He et al., 2015
(note additional /2)
Proper initialization is (was?) an active area of research…

*Understanding the difficulty of training deep feedforward neural networks* by Glorot and Bengio, 2010

*Exact solutions to the nonlinear dynamics of learning in deep linear neural networks* by Saxe et al, 2013

*Random walk initialization for training very deep feedforward networks* by Sussillo and Abbott, 2014

*Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* by He et al., 2015

*Data-dependent Initializations of Convolutional Neural Networks* by Krähenbühl et al., 2015

*All you need is a good init*, Mishkin and Matas, 2015

On weight initialization in deep neural networks, Siddharth Krishna Kumar, 2017

…
Batch Normalization

“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

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

this is a vanilla differentiable function...
Batch Normalization

“I want unit gaussian activations? just make them so.”

1. compute the empirical mean and variance independently for each dimension.

2. Normalize

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

Problem: do we necessarily want a unit gaussian input to the nonlinear layer?

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

Normalize:

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

And then allow the network to squash the range if it wants to:

\[ y(k) = \gamma(k) \hat{x}(k) + \beta(k) \]

Note, the network can learn:

\[ \gamma(k) = \sqrt{\text{Var}[x(k)]} \]
\[ \beta(k) = E[x(k)] \]

to recover the identity mapping.
Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1...m\}$; Parameters to be learned: $\gamma, \beta$

**Output:** $\{y_i = BN_{\gamma,\beta}(x_i)\}$

\[
\begin{align*}
\mu_{\mathcal{B}} & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad // \text{mini-batch mean} \\
\sigma_{\mathcal{B}}^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize} \\
y_i & \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \quad // \text{scale and shift}
\end{align*}
\]

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe
**Batch Normalization**

[loffe and Szegedy, 2015]

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1,...,x_m\}$;  
Parameters to be learned: $\gamma, \beta$

**Output:** $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

\[
\begin{align*}
\mu_{\mathcal{B}} & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean} \\
\sigma_{\mathcal{B}}^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \quad \text{// mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad \text{// normalize} \\
y_i & \leftarrow \gamma\hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\end{align*}
\]

**Note:** at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.  
(e.g. can be estimated during training with running averages)
Fun Tips/Tricks:

1. Train multiple independent models
2. At test time average their results
=> Enjoy 2% extra performance

- can also get a small boost from averaging multiple model checkpoints of a single model.
- keep track of (and use at test time) a running average parameter vector:

```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += -learning_rate * dx
    x_test = 0.995*x_test + 0.005*x  # use for test set
```
Regularization: **Dropout**

“randomly set some neurons to zero in the forward pass”

(a) Standard Neural Net  
(b) After applying dropout.  

[Srivastava et al., 2014]
Regularization: **DisturbLabel**

“randomly change ground truth label of small % of examples”

=> Improves generalization, reduces need for dropout
Case Study: GoogLeNet  

[Case Study: GoogLeNet](Szegedy et al., 2014)

Inception module

ILSVRC 2014 winner (6.7% top 5 error)
### Case Study: GoogLeNet

#### Fun features:
- Only 5 million params! (Removes FC layers completely)

#### Compared to AlexNet:
- 12X less params
- 2x more compute
- 6.67% (vs. 16.4%)
Case Study: ResNet \cite{He2015}

ILSVRC 2015 winner (3.6% top 5 error)

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

*improvements are relative numbers

Slide from Kaiming He's recent presentation [https://www.youtube.com/watch?v=1PGLj-uKT1w](https://www.youtube.com/watch?v=1PGLj-uKT1w)
Revolution of Depth

152 layers

ImageNet Classification top-5 error (%)

ILSVRC'15 ResNet
ILSVRC'14 GoogleNet
ILSVRC'14 VGG
ILSVRC'13
ILSVRC'12 AlexNet
ILSVRC'11 shallow
ILSVRC'10

3.57
6.7
7.3
11.7
16.4
25.8
28.2

(slide from Kaiming He’s recent presentation)
CIFAR-10 experiments

CIFAR-10 plain nets

- 56-layer
- 44-layer
- 32-layer
- 20-layer

solid: test
dashed: train

CIFAR-10 ResNets

- 20-layer
- 32-layer
- 44-layer
- 56-layer
- 110-layer
Case Study: ResNet \textsuperscript{[He et al., 2015]}

ILSVRC 2015 winner (3.6\% top 5 error)

2-3 weeks of training on 8 GPU machine

at runtime: faster than a VGGNet! (even though it has 8x more layers)

(slide from Kaiming He’s recent presentation)
Case Study: ResNet

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

- Batch Normalization after every CONV layer
- Xavier/2 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
Case Study: ResNet [He et al., 2015]

- **all-3x3**: Simplified structure with all 3x3 convolutions.
- **bottleneck**: More complex with 1x1 convolutions, used for ResNet-50/101/152.

Diagram shows the flow of data through layers, with 64-d and 256-d representations, followed by relu activations.

Flowchart indicates the transformation and addition of layers, highlighting the contrast between all-3x3 and bottleneck architectures.

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

(this trick is also used in GoogLeNet)
Case Study: ResNet \[\text{[He et al., 2015]}\]

<table>
<thead>
<tr>
<th>layer name</th>
<th>output size</th>
<th>18-layer</th>
<th>34-layer</th>
<th>50-layer</th>
<th>101-layer</th>
<th>152-layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>112x112</td>
<td></td>
<td></td>
<td>7x7, 64, stride 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3x3 max pool, stride 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conv2_x</td>
<td>56x56</td>
<td>[3x3, 64] x2</td>
<td>[3x3, 64] x3</td>
<td>[1x1, 64]</td>
<td>[1x1, 64]</td>
<td>[1x1, 64]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3x3, 64]</td>
<td>[3x3, 64]</td>
<td>3x3, 64</td>
<td>3x3, 64</td>
<td>3x3, 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1x1, 128</td>
<td>1x1, 128</td>
<td>1x1, 128</td>
</tr>
<tr>
<td>conv3_x</td>
<td>28x28</td>
<td>[3x3, 128] x2</td>
<td>[3x3, 128] x4</td>
<td>[1x1, 512]</td>
<td>[1x1, 512]</td>
<td>[1x1, 512]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3x3, 128]</td>
<td>[3x3, 128]</td>
<td>3x3, 128</td>
<td>3x3, 128</td>
<td>3x3, 128</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1x1, 256</td>
<td>1x1, 256</td>
<td>1x1, 256</td>
</tr>
<tr>
<td>conv4_x</td>
<td>14x14</td>
<td>[3x3, 256] x2</td>
<td>[3x3, 256] x6</td>
<td>[1x1, 256]</td>
<td>[1x1, 256]</td>
<td>[1x1, 256]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3x3, 256]</td>
<td>[3x3, 256]</td>
<td>3x3, 256</td>
<td>3x3, 256</td>
<td>3x3, 256</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1x1, 1024</td>
<td>1x1, 1024</td>
<td>1x1, 1024</td>
</tr>
<tr>
<td>conv5_x</td>
<td>7x7</td>
<td>[3x3, 512] x2</td>
<td>[3x3, 512] x3</td>
<td>[1x1, 512]</td>
<td>[1x1, 512]</td>
<td>[1x1, 512]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3x3, 512]</td>
<td>[3x3, 512]</td>
<td>3x3, 512</td>
<td>3x3, 512</td>
<td>3x3, 512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1x1, 2048</td>
<td>1x1, 2048</td>
<td>1x1, 2048</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>average pool, 1000-d fc, softmax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOPs</td>
<td>1.8x10^9</td>
<td>3.6x10^9</td>
<td>3.8x10^9</td>
<td>7.6x10^9</td>
<td>11.3x10^9</td>
<td></td>
</tr>
</tbody>
</table>
Case Study Bonus: DeepMind’s AlphaGo
The input to the policy network is a 19 × 19 × 48 image stack consisting of 48 feature planes. The first hidden layer zero pads the input into a 23 × 23 image, then convolves $k$ filters of kernel size 5 × 5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21 × 21 image, then convolves $k$ filters of kernel size 3 × 3 with stride 1, again followed by a rectifier nonlinearity. The final layer convolves 1 filter of kernel size 1 × 1 with stride 1, with a different bias for each position, and applies a softmax function. The match version of AlphaGo used $k = 192$ filters; Fig. 2b and Extended Data Table 3 additionally show the results of training with $k = 128, 256$ and 384 filters.

**policy network:**

[19x19x48] Input
CONV1: 192 5x5 filters, stride 1, pad 2 => [19x19x192]
CONV2..12: 192 3x3 filters, stride 1, pad 1 => [19x19x192]
CONV: 1 1x1 filter, stride 1, pad 0 => [19x19] (*probability map of promising moves*)
Computer Vision Tasks

Classification

Classification + Localization

Object Detection

Instance Segmentation

Single object

Multiple objects

CAT

CAT

CAT, DOG, DUCK

CAT, DOG, DUCK
Computer Vision Tasks

- **Classification**
- **Classification + Localization**
- **Object Detection**
- **Instance Segmentation**

Images illustrating each task:
- Left: A cat, highlighting its classification.
- Middle: A cat with a bounding box, highlighting localization.
- Right: A dog with multiple bounding boxes, illustrating object detection.
- Far right: A bucket with two cats and a dog, demonstrating instance segmentation.
**Classification + Localization: Task**

**Classification**: C classes  
- **Input**: Image  
- **Output**: Class label  
- **Evaluation metric**: Accuracy

**Localization**:  
- **Input**: Image  
- **Output**: Box in the image (x, y, w, h)  
- **Evaluation metric**: Intersection over Union

**Classification + Localization**: Do both
Classification + Localization: ImageNet

1000 classes (same as classification)

Each image has 1 class, at least one bounding box

~800 training images per class

Algorithm produces 5 (class, box) guesses

Example is correct if at least one guess has correct class AND bounding box at least 0.5 intersection over union (IoU)

Krizhevsky et. al. 2012
Idea #1: Localization as Regression

**Input:** image

**Output:**
- Box coordinates (4 numbers)

**Correct output:**
- box coordinates (4 numbers)

**Loss:**
- L2 distance

Only one object, simpler than detection
Simple Recipe for Classification + Localization

**Step 1:** Train (or download) a classification model (AlexNet, VGG, GoogLeNet)

- Convolution and Pooling
- Fully-connected layers
- Final conv feature map
- Class scores
- Softmax loss
Simple Recipe for Classification + Localization

**Step 2:** Attach new fully-connected “regression head” to the network
Simple Recipe for Classification + Localization

**Step 3:** Train the regression head only with SGD and L2 loss
Step 4: At test time use both heads

Simple Recipe for Classification + Localization
Per-class vs class agnostic regression

Assume classification over C classes:

**Classification head:**
- C numbers (one per class)

**Class agnostic:**
- 4 numbers (one box)

**Class specific:**
- \( C \times 4 \) numbers (one box per class)
Where to attach the regression head?

- After conv layers: Overfeat, VGG
- After last FC layer: DeepPose, R-CNN
Aside: Localizing multiple objects

Want to localize **exactly** $K$ objects in each image

(e.g. whole cat, cat head, cat left ear, cat right ear for $K=4$)
Aside: Human Pose Estimation

Represent a person by K joints

Regress (x, y) for each joint from last fully-connected layer of AlexNet

(Details: Normalized coordinates, iterative refinement)

Localization as Regression

Very simple

Think if you can use this for projects
Idea #2: Sliding Window

- Run classification + regression network at multiple locations on a high-resolution image

- Convert fully-connected layers into convolutional layers for efficient computation

- Combine classifier and regressor predictions across all scales for final prediction
Sliding Window: Overfeat

Image: 3 x 221 x 221

Convolution + pooling

Feature map: 1024 x 5 x 5

Winner of ILSVRC 2013 localization challenge

4096 - FC - 4096 - FC - Class scores: 1000

Softmax loss

4096 - FC - 1024 - FC - Boxes: 1000 x 4

Euclidean loss

Sliding Window: Overfeat

Network input:
3 x 221 x 221

Larger image:
3 x 257 x 257
Sliding Window: Overfeat

Network input: 3 x 221 x 221

Larger image: 3 x 257 x 257

Classification scores: $P(\text{cat}) = 0.5$
Sliding Window: Overfeat

Network input: 3 x 221 x 221

Larger image: 3 x 257 x 257

Classification scores: P(cat)
Sliding Window: Overfeat

Network input: 3 x 221 x 221

Larger image: 3 x 257 x 257

Classification scores: P(cat)

<table>
<thead>
<tr>
<th>0.5</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td></td>
</tr>
</tbody>
</table>
Sliding Window: Overfeat

Network input: 3 x 221 x 221

Larger image: 3 x 257 x 257

Classification scores: P(cat)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.75</td>
</tr>
<tr>
<td>0.6</td>
<td>0.8</td>
</tr>
</tbody>
</table>
Sliding Window: Overfeat

Network input:
3 x 221 x 221

Larger image:
3 x 257 x 257

Classification scores:
P(cat)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.75</td>
</tr>
<tr>
<td>0.6</td>
<td>0.8</td>
</tr>
</tbody>
</table>
Sliding Window: Overfeat

Greedily merge boxes and scores (details in paper)

Network input:
3 x 221 x 221

Larger image:
3 x 257 x 257

Classification score:
P(cat) = 0.8
Sliding Window: Overfeat

In practice use many sliding window locations and multiple scales

Window positions + score maps
Box regression outputs
Final Predictions

Efficient Sliding Window: Overfeat

Convolution + pooling

Image: 3 x 221 x 221

Feature map: 1024 x 5 x 5

4096

Class scores: 1000

1024

Boxes: 1000 x 4

4096

4096

FC

1024

FC

FC
Efficient Sliding Window: Overfeat

Efficient sliding window by converting fully-connected layers into convolutions

Image: 3 x 221 x 221

Convolution + pooling

Feature map: 1024 x 5 x 5

4096 x 1 x 1

5 x 5 conv

1024 x 1 x 1

1 x 1 conv

5 x 5 conv

1024 x 1 x 1

1 x 1 conv

4096 x 1 x 1

1 x 1 conv

1024 x 1 x 1

Box coordinates: (4 x 1000) x 1 x 1

Class scores: 1000 x 1 x 1

1 x 1 conv
Efficient Sliding Window: Overfeat

**Training time:** Small image, 1 x 1 classifier output

**Test time:** Larger image, 2 x 2 classifier output, only extra compute at yellow regions

ImageNet Classification + Localization

Localization Error (Top 5)

- **AlexNet**: Localization method not published
- **Overfeat**: Multiscale convolutional regression with box merging
- **VGG**: Same as Overfeat, but fewer scales and locations; simpler method, gains all due to deeper features
- **ResNet**: Different localization method (RPN) and much deeper features
Computer Vision Tasks

Classification

Classification + Localization

Object Detection

Instance Segmentation
Computer Vision Tasks

- Classification
- Classification + Localization
- Object Detection
- Instance Segmentation

Images showing examples of each task:
Detection as Regression?

DOG, (x, y, w, h)
CAT, (x, y, w, h)
CAT, (x, y, w, h)
DUCK (x, y, w, h)

= 16 numbers
Detection as Regression?

DOG, (x, y, w, h)
CAT, (x, y, w, h)

= 8 numbers
Detection as Regression?

CAT, (x, y, w, h)
CAT, (x, y, w, h)
....
CAT (x, y, w, h)
= many numbers

Need variable sized outputs
Detection as Classification

CAT? NO

DOG? NO
Detection as Classification

CAT? YES!
DOG? NO
Detection as Classification

CAT? NO

DOG? NO
Detection as Classification

**Problem:** Need to test many positions and scales

**Solution:** If your classifier is fast enough, just do it
Histogram of Oriented Gradients

- Compute HOG of the whole image at multiple resolutions
- Score every subwindow of the feature pyramid
- Apply non-maxima suppression

Dalal and Triggs, “Histograms of Oriented Gradients for Human Detection”, CVPR 2005
Slide credit: Ross Girshick
Deformable Parts Model (DPM)

Aside: Deformable Parts Models are CNNs?

Girschick et al, “Deformable Part Models are Convolutional Neural Networks”, CVPR 2015
Detection as Classification

**Problem:** Need to test many positions and scales, and use a computationally demanding classifier (CNN)

**Solution:** Only look at a tiny subset of possible positions
Region Proposals

- Find “blobby” image regions that are likely to contain objects
- “Class-agnostic” object detector
- Look for “blob-like” regions
Region Proposals: Selective Search

Bottom-up segmentation, merging regions at multiple scales

### Region Proposals: Many other choices

<table>
<thead>
<tr>
<th>Method</th>
<th>Approach</th>
<th>Outputs</th>
<th>Outputs</th>
<th>Control</th>
<th>Time (sec.)</th>
<th>Repeatability</th>
<th>Recall Results</th>
<th>Detection Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bing [18]</td>
<td>Window scoring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0.2</td>
<td>***</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>CPMC [19]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>250</td>
<td>-</td>
<td>**</td>
<td>*</td>
</tr>
<tr>
<td>EdgeBoxes [20]</td>
<td>Window scoring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0.3</td>
<td>**</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>Endres [21]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>100</td>
<td>-</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td>Geodesic [22]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>*</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td>MCG [23]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>30</td>
<td>*</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td>Objectness [24]</td>
<td>Window scoring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>3</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Rahtu [25]</td>
<td>Window scoring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>3</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>RandomizedPrim’s [26]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>*</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td>Rantalankila [27]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10</td>
<td>**</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td>Rigor [28]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10</td>
<td>*</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>SelectiveSearch [29]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10</td>
<td>**</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>Gaussian</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>SlidingWindow</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>0</td>
<td>***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superpixels</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniform</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hosang et al, “What makes for effective detection proposals?”, PAMI 2015
Region Proposals: Many other choices

<table>
<thead>
<tr>
<th>Method</th>
<th>Approach</th>
<th>Outputs</th>
<th>Outputs</th>
<th>Control #proposals</th>
<th>Time (sec.)</th>
<th>Repeatability</th>
<th>Recall Results</th>
<th>Detection Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bing [18]</td>
<td>Window scoring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0.2</td>
<td>***</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>CPMC [19]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>250</td>
<td>-</td>
<td>**</td>
<td>*</td>
</tr>
<tr>
<td>EdgeBoxes [20]</td>
<td>Window scoring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0.3</td>
<td>**</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>Endres [21]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>100</td>
<td>-</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td>Geodesic [22]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>*</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td>MCG [23]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>30</td>
<td>*</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td>Objectness [24]</td>
<td>Window scoring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>3</td>
<td>-</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Rahtu [25]</td>
<td>Window scoring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>3</td>
<td>-</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>RandomizedPrim’s [26]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>*</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td>Rantalanikila [27]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10</td>
<td>**</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td>Rigor [28]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10</td>
<td>*</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>SelectiveSearch [29]</td>
<td>Grouping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10</td>
<td>**</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>Gaussian</td>
<td></td>
<td>✓</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SlidingWindow</td>
<td></td>
<td>✓</td>
<td>0</td>
<td></td>
<td>0</td>
<td>***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superpixels</td>
<td></td>
<td>✓</td>
<td>1</td>
<td></td>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniform</td>
<td></td>
<td>✓</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hosang et al, “What makes for effective detection proposals?”, PAMI 2015
Putting it together: R-CNN


Slide credit: Ross Girshick
R-CNN Training

**Step 1:** Train (or download) a classification model for ImageNet (AlexNet)

1. **Image**
2. **Convolution and Pooling**
3. **Final conv feature map**
4. **Fully-connected layers**
5. **Class scores 1000 classes**
6. **Softmax loss**
R-CNN Training

**Step 2:** Fine-tune model for detection
- Instead of 1000 ImageNet classes, want 20 object classes + background
- Throw away final fully-connected layer, reinitialize from scratch
- Keep training model using positive / negative regions from detection images

![Diagram](image)

- Convolution and Pooling
- Final convolution feature map
- Fully-connected layers
- Class scores: 21 classes
- Softmax loss
- Re-initialize this layer: was 4096 x 1000, now will be 4096 x 21
**R-CNN Training**

**Step 3: Extract features**
- Extract region proposals for all images
- For each region: warp to CNN input size, run forward through CNN, save pool5 features to disk
- Have a big hard drive: features are ~200GB for PASCAL dataset!
R-CNN Training

**Step 4**: Train one binary SVM per class to classify region features

- **Training image regions**
  - Positive samples for cat SVM
  - Negative samples for cat SVM

- **Cached region features**
R-CNN Training

Step 4: Train one binary SVM per class to classify region features

Training image regions

Cached region features

Negative samples for dog SVM

Positive samples for dog SVM
R-CNN Training

**Step 5** (bbox regression): For each class, train a linear regression model to map from cached features to offsets to GT boxes to make up for “slightly wrong” proposals.

- **Training image regions**
- **Cached region features**
- **Regression targets**
  - (dx, dy, dw, dh)
  - Normalized coordinates
  - (0, 0, 0, 0) Proposal is good
  - (.25, 0, 0, 0) Proposal too far to left
  - (0, 0, -0.125, 0) Proposal too wide
## Object Detection: Datasets

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of classes</td>
<td>20</td>
<td>200</td>
<td>80</td>
</tr>
<tr>
<td>Number of images (train + val)</td>
<td>~20k</td>
<td>~470k</td>
<td>~120k</td>
</tr>
<tr>
<td>Mean objects per image</td>
<td>2.4</td>
<td>1.1</td>
<td>7.2</td>
</tr>
</tbody>
</table>
Object Detection: Evaluation

We use a metric called “mean average precision” (mAP)

Compute average precision (AP) separately for each class, then average over classes

A detection is a true positive if it has IoU with a ground-truth box greater than some threshold (usually 0.5) (mAP@0.5)

Combine all detections from all test images to draw a precision / recall curve for each class; AP is area under the curve

TL;DR mAP is a number from 0 to 100; high is good
R-CNN Results

R-CNN Results

Big improvement compared to pre-CNN methods

![Bar Chart]

Mean Average Precision (mAP)

- DPM (2011): 33.7 / 29.6
- Regionlets (2013): 41.7 / 39.7
- R-CNN (2014, AlexNet): 54.2 / 50.2
- R-CNN + bbox reg (AlexNet): 58.5 / 53.7
- R-CNN (VGG-16): 66 / 62.9

VOC 2007
VOC 2010
R-CNN Results

Bounding box regression helps a bit

Mean Average Precision (mAP)

- DPM (2011): 33.7, 29.6
- Regionlets (2013): 41.7, 39.7
- R-CNN (2014, AlexNet): 54.2, 50.2
- R-CNN + bbox reg (AlexNet): 58.5, 53.7
- R-CNN (VGG-16): 66, 62.9

VOC 2007
VOC 2010
R-CNN Results

Features from a deeper network help a lot

Mean Average Precision (mAP)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VOC 2007</td>
<td>33.7</td>
<td>41.7</td>
<td>54.2</td>
<td>58.5</td>
<td>66</td>
</tr>
<tr>
<td>VOC 2010</td>
<td>29.6</td>
<td>39.7</td>
<td>50.2</td>
<td>53.7</td>
<td>62.9</td>
</tr>
</tbody>
</table>
R-CNN Problems

1. Slow at test-time: need to run full forward pass of CNN for each region proposal

2. SVMs and regressors are post-hoc: CNN features not updated in response to SVMs and regressors

3. Complex multistage training pipeline
Fast R-CNN (test time)

Regions of Interest (Rois) from a proposal method

ConvNet

Forward whole image through ConvNet

"Conv5" feature map of image

"RoI Pooling" (single-level SPP) layer

Fully-connected layers

Linear

Bounding-box regressors

Linear + softmax

Softmax classifier


Slide credit: Ross Girshick
Fast R-CNN (test time)

R-CNN Problem #1:
Slow at test-time due to independent forward passes of the CNN

Solution:
Share computation of convolutional layers between proposals for an image
R-CNN Problem #2: Post-hoc training: CNN not updated in response to final classifiers and regressors

R-CNN Problem #3: Complex training pipeline

Solution: Just train the whole system end-to-end all at once!
Fast R-CNN: Region of Interest Pooling

Hi-res input image: 3 x 800 x 600 with region proposal

Hi-res conv features: C x H x W with region proposal

Problem: Fully-connected layers expect low-res conv features: C x h x w
Fast R-CNN: Region of Interest Pooling

**Problem:** Fully-connected layers expect low-res conv features: $C \times h \times w$

**Hi-res input image:** $3 \times 800 \times 600$

**Hi-res conv features:** $C \times H \times W$

**Fully-connected layers**

**Project region proposal onto conv feature map**

**Convolution and Pooling**

**Project region proposal with region proposal**
Fast R-CNN: Region of Interest Pooling

Hi-res input image: 3 x 800 x 600 with region proposal

Hi-res conv features: C x H x W with region proposal

Divide projected region into h x w grid

Problem: Fully-connected layers expect low-res conv features: C x h x w

Convolution and Pooling

Fully-connected layers
Fast R-CNN: Region of Interest Pooling

Hi-res input image: 3 x 800 x 600 with region proposal

Hi-res conv features: C x H x W with region proposal

Max-pool within each grid cell

RoI conv features: C x h x w for region proposal

Fully-connected layers expect low-res conv features: C x h x w
Fast R-CNN: Region of Interest Pooling

Hi-res input image:
3 x 800 x 600
with region proposal

Hi-res conv features:
C x H x W
with region proposal

Convolution and Pooling

Can back propagate similar to max pooling

Rol conv features:
C x h x w
for region proposal

Fully-connected layers expect
low-res conv features:
C x h x w

Fully-connected layers
Fast R-CNN Results

<table>
<thead>
<tr>
<th></th>
<th>R-CNN</th>
<th>Fast R-CNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Time:</td>
<td>84 hours</td>
<td>9.5 hours</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>8.8x</td>
</tr>
</tbody>
</table>

Faster!

Using VGG-16 CNN on Pascal VOC 2007 dataset
<table>
<thead>
<tr>
<th></th>
<th>R-CNN</th>
<th>Fast R-CNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Time:</td>
<td>84 hours</td>
<td>9.5 hours</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>8.8x</td>
</tr>
<tr>
<td>Test time per image</td>
<td>47 seconds</td>
<td>0.32 seconds</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>146x</td>
</tr>
</tbody>
</table>

Using VGG-16 CNN on Pascal VOC 2007 dataset
## Fast R-CNN Results

<table>
<thead>
<tr>
<th></th>
<th>R-CNN</th>
<th>Fast R-CNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Time:</td>
<td>84 hours</td>
<td>9.5 hours</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>8.8x</td>
</tr>
<tr>
<td>Test time per image</td>
<td>47 seconds</td>
<td>0.32 seconds</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>146x</td>
</tr>
<tr>
<td>mAP (VOC 2007)</td>
<td>66.0</td>
<td>66.9</td>
</tr>
</tbody>
</table>

Using VGG-16 CNN on Pascal VOC 2007 dataset
Fast R-CNN Problem:

Test-time speeds don’t include region proposals

<table>
<thead>
<tr>
<th></th>
<th>R-CNN</th>
<th>Fast R-CNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test time per image</td>
<td>47 seconds</td>
<td>0.32 seconds</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>146x</td>
</tr>
<tr>
<td>Test time per image with Selective Search</td>
<td>50 seconds</td>
<td>2 seconds</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>25x</td>
</tr>
</tbody>
</table>
Fast R-CNN Problem Solution:

Test-time speeds don’t include region proposals
Just make the CNN do region proposals too!

<table>
<thead>
<tr>
<th></th>
<th>R-CNN</th>
<th>Fast R-CNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test time per image</td>
<td>47 seconds</td>
<td>0.32 seconds</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>146x</td>
</tr>
<tr>
<td>Test time per image</td>
<td>50 seconds</td>
<td>2 seconds</td>
</tr>
<tr>
<td>with Selective Search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>25x</td>
</tr>
</tbody>
</table>
Faster R-CNN:

Insert a **Region Proposal Network (RPN)** after the last convolutional layer

RPN trained to produce region proposals directly; no need for external region proposals!

After RPN, use RoI Pooling and an upstream classifier and bbox regressor just like Fast R-CNN


Slide credit: Ross Girshick
Faster R-CNN: Region Proposal Network

Slide a small window on the feature map

Build a small network for:
- classifying object or not-object, and
- regressing bbox locations

Position of the sliding window provides localization information with reference to the image

Box regression provides finer localization information with reference to this sliding window

Slide credit: Kaiming He
Faster R-CNN: Region Proposal Network

Use **N anchor boxes** at each location

Anchors are **translation invariant**: use the same ones at every location

Regression gives offsets from anchor boxes

Classification gives the probability that each (regressed) anchor shows an object
Faster R-CNN: Training

In the paper: Ugly pipeline
- Use alternating optimization to train RPN, then Fast R-CNN with RPN proposals, etc.
- More complex than it has to be

Since publication: Joint training!
One network, four losses
- RPN classification (anchor good / bad)
- RPN regression (anchor -> proposal)
- Fast R-CNN classification (over classes)
- Fast R-CNN regression (proposal -> box)
## Faster R-CNN: Results

<table>
<thead>
<tr>
<th></th>
<th>R-CNN</th>
<th>Fast R-CNN</th>
<th>Faster R-CNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test time per image (with proposals)</td>
<td>50 seconds</td>
<td>2 seconds</td>
<td>0.2 seconds</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>25x</td>
<td>250x</td>
</tr>
<tr>
<td>mAP (VOC 2007)</td>
<td>66.0</td>
<td>66.9</td>
<td>66.9</td>
</tr>
</tbody>
</table>
Object Detection State-of-the-art: ResNet 101 + Faster R-CNN + some extras

<table>
<thead>
<tr>
<th>Training data</th>
<th>COCO train</th>
<th>COCO trainval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test data</td>
<td>COCO val</td>
<td>COCO test-dev</td>
</tr>
<tr>
<td>mAP baseline Faster R-CNN (VGG-16)</td>
<td>@.5 21.2</td>
<td>@.5 21.2</td>
</tr>
<tr>
<td>baseline Faster R-CNN (ResNet-101)</td>
<td>48.4 27.2</td>
<td>53.3 32.2</td>
</tr>
<tr>
<td>+box refinement</td>
<td>49.9 29.9</td>
<td>55.7 34.9</td>
</tr>
<tr>
<td>+context</td>
<td>51.1 30.0</td>
<td>53.3 32.2</td>
</tr>
<tr>
<td>+multi-scale testing</td>
<td>53.8 32.5</td>
<td>55.7 34.9</td>
</tr>
<tr>
<td>Ensemble</td>
<td>59.0 37.4</td>
<td></td>
</tr>
</tbody>
</table>

ImageNet Detection 2013 - 2015

ImageNet Detection (mAP)

- NeoNet ensemble (2015): 53.57
- Faster R-CNN single (2015): 42.94
- GoogleNet ensemble (2014): 43.93
- NUS ensemble (2014): 37.21
- SPP ensemble (2014): 35.11
- UvA-Eur vision (2013): 22.56
- Overfeat (2013): 19.4
YOLO: You Only Look Once
Detection as Regression

Divide image into $S \times S$ grid

Within each grid cell predict:
- B Boxes: 4 coordinates + confidence
- Class scores: C numbers

Regression from image to
$7 \times 7 \times (5 \times B + C)$ tensor

Direct prediction using a CNN

YOLO: You Only Look Once
Detection as Regression

Faster than Faster R-CNN, but not as good