Convolutional Neural Networks
Lecture in Neural Networks course, LTAT.02.001

Ardi Tampuu

Tartu, October 2017
Disclaimer

These slides are adapted from Stanford University course CS231 on "Convolutional Neural Networks for Visual Recognition"
So far you have seen that there are shallow and deep networks.

You have seen that deep networks can be trained using backpropagation.

You have heard that deep networks can learn more complex things than shallow ones.
Seen so far

Shallow network weights (Practice 2: Softmax):

Product between input and weights gives higher value if the image is similar to weights — weights look like average of each class.
Deep network:

- can capture more complex patterns (not just pixelwise similarity)
- can hopefully recognize any color of car.
Convolutional network:
  - can hopefully recognize a car even if it is not at the center of the image
A bit of history:

Hubel & Wiesel,
1959
RECEPTIVE FIELDS OF SINGLE NEURONES IN THE CAT'S STRIATE CORTEX
1962
RECEPTIVE FIELDS, BINOCULAR INTERACTION AND FUNCTIONAL ARCHITECTURE IN THE CAT'S VISUAL CORTEX
1968...
Hierarchical organization

Illustration of hierarchical organization in early visual pathways by Lane McIntosh, copyright CS231n 2017

Simple cells: Response to light orientation

Complex cells: Response to light orientation and movement

Hypercomplex cells: response to movement with an end point

No response
Response (end point)
A bit of history:

Neocognitron

[Fukushima 1980]

“sandwich” architecture (SCSCSC…)
simple cells: modifiable parameters
complex cells: perform pooling
A bit of history:
Gradient-based learning applied to document recognition
[LeCun, Bottou, Bengio, Haffner 1998]
A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]
Fast-forward to today: ConvNets are everywhere

Classification

Retrieval

Fast-forward to today: ConvNets are everywhere

Detection

Segmentation

Figures copyright Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, 2015. Reproduced with permission.

[Faster R-CNN: Ren, He, Girshick, Sun 2015]


[Farabet et al., 2012]
Fast-forward to today: ConvNets are everywhere

[Levy et al. 2016]

[Dieleman et al. 2014]

[Sermanet et al. 2011]

[Ciresan et al.]
Convolutional Neural Networks
(First without the brain stuff)
Reminder: Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1

Each neuron looks at the full input volume

W has 3072x10 weights

The result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)
Convolution Layer

32x32x3 image -> preserve spatial structure
Convolution Layer

32x32x3 image

5x5x3 filter

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”
Convolution Layer

Filters always extend the full depth of the input volume

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”
Convolution Layer

32x32x3 image
5x5x3 filter \( w \)

1 number: the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

\[ w^T x + b \]
Convolution Layer

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map
Convolution Layer

- 32x32x3 image
- 5x5x3 filter
- Convolve (slide) over all spatial locations
- Activation maps

Consider a second, green filter
For example, if we had 6 5x5 filters, we’ll get 6 separate activation maps:

We stack these up to get a “new image” of size 28x28x6!
**Preview:** ConvNet is a sequence of Convolution Layers, interspersed with activation functions.

![Diagram](image)

- CONV, ReLU
- e.g. 6 5x5x3 filters
Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions.
[Zeiler and Fergus 2013]

Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].
preview:
A closer look at spatial dimensions:

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially) 
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter

=> 5x5 output
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
applied with stride 2
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2 => 3x3 output!
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 3?
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 3?

doesn’t fit! cannot apply 3x3 filter on 7x7 input with stride 3.
Output size:

\[(N - F) / \text{stride} + 1\]

e.g. \(N = 7, F = 3:\)

- \(\text{stride 1} \Rightarrow (7 - 3)/1 + 1 = 5\)
- \(\text{stride 2} \Rightarrow (7 - 3)/2 + 1 = 3\)
- \(\text{stride 3} \Rightarrow (7 - 3)/3 + 1 = 2.33\)
In practice: Common to zero pad the border

e.g. input 7x7
3x3 filter, applied with stride 1
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1
In practice: Common to zero pad the border

e.g. input 7x7
3x3 filter, applied with **stride 1**
**pad with 1 pixel** border => what is the output?

7x7 output!
In practice: Common to zero pad the border

e.g. input 7x7
3x3 filter, applied with **stride 1**
**pad with 1 pixel** border => what is the output?

7x7 output!
in general, common to see CONV layers with
stride 1, filters of size FxF, and zero-padding with
(F-1)/2. (will preserve size spatially)
e.g. $F = 3$ => zero pad with 1
    $F = 5$ => zero pad with 2
    $F = 7$ => zero pad with 3
Remember back to…
E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn’t work well.
Examples time:

Input volume: $32 \times 32 \times 3$
10 $5 \times 5$ filters with stride 1, pad 2

Output volume size: ?
Examples time:

Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
\[(32+2\times2-5)/1+1 = 32\] spatially, so **32x32x10**
Examples time:

Input volume: 32x32x3
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
Examples time:

Input volume: $32 \times 32 \times 3$
10 $5 \times 5$ filters with stride 1, pad 2

Number of parameters in this layer?
each filter has $5 \times 5 \times 3 + 1 = 76$ params (+1 for bias)

$=> 76 \times 10 = 760$
Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters $K$,
  - their spatial extent $F$,
  - the stride $S$,
  - the amount of zero padding $P$.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and $K$ biases.
- In the output volume, the $d$-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the $d$-th filter over the input volume with a stride of $S$, and then offset by $d$-th bias.
Common settings:

- \( K = \text{(powers of 2, e.g. 32, 64, 128, 512)} \)
- \( F = 3, S = 1, P = 1 \)
- \( F = 5, S = 1, P = 2 \)
- \( F = 5, S = 2, P = ? \) (whatever fits)
- \( F = 1, S = 1, P = 0 \)
The brain/neuron view of CONV Layer

32x32x3 image
5x5x3 filter

1 number: the result of taking a dot product between the filter and this part of the image (i.e. 5*5*3 = 75-dimensional dot product)
Reminder: Fully Connected Layer

32x32x3 image -> stretch to $3072 \times 1$

Each neuron looks at the full input volume

$W \cdot x$

$W$ has $3072 \times 10$ weights

10 x $3072$

weights

1 number: the result of taking a dot product between a row of $W$ and the input (a 3072-dimensional dot product)
The brain/neuron view of CONV Layer

32x32x3 image
5x5x3 filter

1 number: the result of taking a dot product between the filter and this part of the image (i.e. 5*5*3 = 75-dimensional dot product)

It’s just a neuron with local connectivity...
The brain/neuron view of CONV Layer

An activation map is a 28x28 sheet of neuron outputs:
1. Each is connected to a small region in the input
2. All of them share parameters

“5x5 filter” -> “5x5 receptive field for each neuron”
The brain/neuron view of CONV Layer

E.g. with 5 filters, CONV layer consists of neurons arranged in a 3D grid (28x28x5)

There will be 5 different neurons all looking at the same region in the input volume
two more layers to go: POOL/FC
Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:

![Diagram of pooling layer](image)
Max Pooling

Single depth slice

```
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
```

Max pool with 2x2 filters and stride 2

```
6 8
3 4
```
• Accepts a volume of size $W_1 \times H_1 \times D_1$
• Requires three hyperparameters:
  ◦ their spatial extent $F$,
  ◦ the stride $S$,
• Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  ◦ $W_2 = (W_1 - F)/S + 1$
  ◦ $H_2 = (H_1 - F)/S + 1$
  ◦ $D_2 = D_1$
• Introduces zero parameters since it computes a fixed function of the input
• Note that it is not common to use zero-padding for Pooling layers
Fully Connected Layer (FC layer)
- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks
Last time: Deep learning frameworks

```python
class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)

    def backward(self, grad_y):
        x, = self.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```
Define model architecture as a sequence of layers

```python
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```
Today: CNN Architectures

Case Studies
- AlexNet
- VGG
- GoogLeNet
- ResNet

Also....
- NiN (Network in Network)  
- Wide ResNet  
- ResNeXT  
- Stochastic Depth
- DenseNet
- FractalNet
- SqueezeNet
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]
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: 

\[
(227-11)/4+1 = 55
\]
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*11*3)*96 = **35K**
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

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?

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
Parameters: 0!
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.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

First CNN-based winner

Figure copyright Kaiming He, 2016. Reproduced with permission.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- **ZFNet**: Improved hyperparameters over AlexNet

The diagram shows the progression of models over the years, with ZFNet leading in 2013 with 8 layers. ZFNet is noted for having 152 layers.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners
## 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

---

### AlexNet
- 11x11 conv, 96
- 5x5 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
- 3x3 conv, 256
- Pool
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

<table>
<thead>
<tr>
<th>AlexNet</th>
<th>VGG16</th>
<th>VGG19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Input</td>
<td>Input</td>
</tr>
<tr>
<td>3x3 conv, 128</td>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>3x3 conv, 128</td>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>3x3 conv, 128</td>
<td>Pool</td>
<td>Pool</td>
</tr>
<tr>
<td>5x5 conv, 256</td>
<td>FC 4096</td>
<td>FC 4096</td>
</tr>
<tr>
<td>11x11 conv, 96</td>
<td>FC 4096</td>
<td>FC 4096</td>
</tr>
<tr>
<td>Softmax</td>
<td>FC 1000</td>
<td>FC 1000</td>
</tr>
<tr>
<td>FC 1000</td>
<td>FC 1000</td>
<td>FC 1000</td>
</tr>
<tr>
<td>FC 4096</td>
<td>FC 4096</td>
<td>FC 4096</td>
</tr>
<tr>
<td>Pool</td>
<td>Pool</td>
<td>Pool</td>
</tr>
</tbody>
</table>
INPUT: [224x224x3] memory:  224*224*3 = 150K  params: 0

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 = 800K  params: 0

CONV3-256: [56x56x256] memory:  56*56*256 = 1.6M  params: (3*3*128)*256 = 589,824

CONV3-256: [56x56x256] memory:  56*56*256 = 1.6M  params: (3*3*256)*256 = 1,179,648

CONV3-256: [56x56x256] memory:  56*56*256 = 1.6M  params: (3*3*256)*256 = 2,359,296

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

TOTAL memory: 24M * 4 bytes ≈ 96MB / image (not counting biases)
TOTAL params: 138M 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

Details:
- ILGVRC’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
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

Deeper Networks

ILSVRC'15 ResNet
ILSVRC'14 GoogleNet
ILSVRC'14 VGG

ILSVRC'13 11.7
ILSVRC'12 AlexNet 16.4
shallow

ILSVRC'11 25.8
ILSVRC'10 28.2

Figure copyright Kaiming He, 2016. Reproduced with permission.
Case Study: GoogLeNet

[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- No FC layers
- Only 5 million parameters!
  12x less than AlexNet
- ILSVRC’14 classification winner
  (6.7% top 5 error)
Case Study: GoogLeNet
[Szegedy et al., 2014]

Example:

Q3: What is output size after filter concatenation?

\[ 28 \times 28 \times (128 + 192 + 96 + 256) = 529k \]

Naive Inception module

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

Solution: “bottleneck” layers that use 1x1 convolutions to reduce feature depth
Case Study: GoogLeNet

[Szegedy et al., 2014]

Naive Inception module

Inception module with dimension reduction
Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

22 total layers with weights (including each parallel layer in an Inception module)
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

“Revolution of Depth”

Figure copyright Kaiming He, 2016. Reproduced with permission.
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!

ImageNet classification winner

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?

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

-> The deeper model performs worse, but it’s not caused by overfitting!
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.

\[ H(x) = F(x) + x \]

“Plain” layers

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
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)
Case Study: ResNet

[He et al., 2015]

Total depths of 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]

Training ResNet in practice:

- 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]

Experimental Results
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lowing 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)
Batch Normalization
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:

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

this is a vanilla differentiable function...
Batch Normalization

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

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

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

[ioffe and Szegedy, 2015]
Batch Normalization

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

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

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

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

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

Normalize:

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

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:

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

to recover the identity mapping.

[Ioffe and Szegedy, 2015]
Batch Normalization

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

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

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

- 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

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)

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

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

$$
\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad // \text{mini-batch mean}
$$

$$
\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}
$$

$$
\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \quad // \text{normalize}
$$

$$
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad // \text{scale and shift}
$$
Thanks!

You can now thank me for this awesome presentation!