Recap: Convolutional Network

Simple ConvNet:
- One convolutional layer with max-pooling
- Final fully connected hidden layer (no sharing weight)
- Softmax output layer
Recap: Stacking convolutional layers

- Local receptive fields
- Weight sharing
- Pooling/subsampling
Training Convolutional Networks

Remember: $g^\ell_j = \partial E / \partial a^\ell_j$, is the error signal for unit $j$ in layer $\ell$
Training Convolutional Networks – Pooling Layer

$G$ is a “pseudo-weight matrix” for max-pooling which is set during the forward propagation:

$G_{ba} = 1$ if feature map unit $b$ is contained in max-pool $a$ and is the maximum value for the current input. Note that $G$ is different for each item in the mini-batch.
To update the shared weights of the convolutional layer, we take account of all units to which a shared weight is connected, by summing over the convolutional units:

\[
\frac{\partial E}{\partial w_{k,\ell}^{L-3}} = \sum_{i=0}^{D-1} \sum_{j=0}^{D-1} \frac{\partial E}{\partial a_{i,j}^{L-3}} \frac{\partial a_{i,j}^{L-3}}{\partial w_{k,\ell}^{L-3}}
\]

where the kernel has dimension \(m \times m\), the feature map has dimension \(D \times D\) and \(a_{i,j}\) is the activation of the \((i,j)\)th unit in the feature map (see slide 16 in lecture 7):

\[
a_{i,j}^{L-3} = \sum_{r=0}^{m-1} \sum_{s=0}^{m-1} w_{r,s}^{L-3} h_{i+r,j+s}^{L-4} + b^{L-3}
\]

Recalling that \(g_{i,j}^{L-3} = \frac{\partial E}{\partial a_{i,j}^{L-3}}\), then we have:

\[
\frac{\partial E}{\partial w_{k,\ell}^{L-3}} = \sum_{i=0}^{D-1} \sum_{j=0}^{D-1} g_{i,j}^{L-3} h_{i+k,j+\ell}^{L-4} = g_{i,j}^{L-3} \otimes h_{i+k,j+\ell}^{L-4}(k, \ell)
\]
Training the convolutional layer is more complicated

\[
g^{L-3} = g^{L-2} G^{L-2T} \circ f'(a^{L-3})
\]

\[
\frac{\partial E}{\partial W^{L-3}} = g^{L-3} \otimes h^{L-4}
\]
Training Convolutional Networks – Convolutional Layer

**3x12x12 Pooling Layers**

**1x8x8 Feature Maps**

<table>
<thead>
<tr>
<th>Backprop Gradients</th>
<th>?</th>
<th>$g^{L-3} = g^{L-2} G^{L-2} \circ f'(a^{L-3})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight Gradients</td>
<td></td>
<td>$\frac{\partial E}{\partial W_{L-3}} = g^{L-3} \otimes h^{L-4}$</td>
</tr>
</tbody>
</table>

*Only need to consider one pooling layer*
Training Convolutional Networks – Convolutional Layer

3x12x12 Pooling Layers

1x8x8 Feature Maps

Weight

Gradients

Backprop Gradients

\[ \frac{\partial E}{\partial W_{L-3}} = g_{L-3} \odot h_{L-4} \]

\[ g_{L-3} = g_{L-2} G_{L-2}^T \circ f'(a_{L-3}) \]

Simplify by only considering one feature map
Convolutional Layer – Forward Pass

\[ h_{i,j} = \text{sigmoid} \left( \sum_{k=0}^{m-1} \sum_{\ell=0}^{m-1} w_{k,\ell} x_{i+k,j+\ell} + b \right) \]

Forward pass: each hidden unit connected to a region of input units (receptive field)
Convolutional Layer – Forward Pass

Forward pass: each hidden unit connected to a region of input units (receptive field)
Convolutional Layer – Forward Pass

Forward pass: each hidden unit connected to a region of input units (receptive field)

\[ h_{i,j} = \text{sigmoid} \left( \sum_{k=0}^{m-1} \sum_{\ell=0}^{m-1} w_{k,\ell} x_{i+k,j+\ell} + b \right) \]
Convolutional Layer – Forward Pass

Forward pass: each hidden unit connected to a region of input units (receptive field)
Backward pass: consider the region of hidden units connected to each input unit.
Backward pass: consider the region of hidden units connected to each input unit.
Convolutional Layer – Backward Pass

Backward pass: consider the region of hidden units connected to each input unit.
Backward pass: consider the region of hidden units connected to each input unit.
Backward pass: consider the region of hidden units connected to each input unit.
Backward pass: consider the region of hidden units connected to each input unit.
Convolutional Layer – Backward Pass

Backward pass: consider the region of hidden units connected to each input unit.
As usual we want to back-propagate the gradients:

\[ g_{s}^{L-4} = \sum_{j \in \text{connected to } s} w_{js}g_{j}^{L-3}f'(a_{s}) \]

Look at the shared weights used for back prop:
As usual we want to back-propagate the gradients:

\[ g_s^{L-4} = \sum_{j \in \text{connected to } s} w_{js} g_j^{L-3} f'(a_s) \]

Look at the shared weights used for back prop:
As usual we want to back-propagate the gradients:

\[
g_{s}^{L-4} = \sum_{j \in \text{connected to } s} w_{js}g_{j}^{L-3}f'(a_{s})
\]

Look at the shared weights used for back prop:
Convolutional Layer – Back Propagation

As usual we want to back-propagate the gradients:

\[ g_s^{L-4} = \sum_{j \in \text{connected to } s} w_{js} g_j^{L-3} f'(a_s) \]

Look at the shared weights used for back prop:

12x12 Pooling Layer

8x8 Feature Map
As usual we want to back-propagate the gradients:

\[ g_s^{L-4} = \sum_{j \in \text{connected to } s} w_{js} g_j^{L-3} f'(a_s) \]

Look at the shared weights used for back prop:

MLP Lecture 8
Convolutional Networks 2: Training, deep convolutional networks
As usual we want to back-propagate the gradients:

$$g_{s}^{L-4} = \sum_{j \in \text{connected to } s} w_{js}g_{j}^{L-3}f'(a_s)$$

Look at the shared weights used for back prop:

12x12 Pooling Layer  
8x8 Feature Map
Backprop as convolution

If we have an $m \times m$ kernel size, we can pad the feature map with $(m - 1)$ rows and columns of 0s top and bottom, left and right.
Backprop as convolution

If we have an \( m \times m \) kernel size, we can pad the feature map with \((m - 1)\) rows and columns of 0s top and bottom, left and right.

Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the weight matrix is rotated by 180° (flipped)
If we have an \( m \times m \) kernel size, we can pad the feature map with \((m - 1)\) rows and columns of 0s top and bottom, left and right.

Backprop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the weight matrix is rotated by \(180^\circ\) (flipped).
Backprop as convolution

If we have an $m \times m$ kernel size, we can pad the feature map with $(m - 1)$ rows and columns of 0s top and bottom, left and right.

Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the weight matrix is rotated by $180^\circ$ (flipped).
If we have an $m \times m$ kernel size, we can pad the feature map with $(m - 1)$ rows and columns of 0s top and bottom, left and right.

Backprop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the weight matrix is rotated by $180^\circ$ (flipped).
If we have an $m \times m$ kernel size, we can pad the feature map with $(m - 1)$ rows and columns of 0s top and bottom, left and right.

Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the weight matrix is rotated by $180^\circ$ (flipped)
Back-propagation in a convolutional layer, is also a convolution. But we have to *rotate* the weight matrix $W$ by $180^\circ$ (flip the weight matrix), $W^R$. Using the convolution operator we saw we can write the forward pass as:

$$a^{L-3} = W^{L-3} \ast h^{L-4} + b^{L-3} \quad ; \quad h^{L-3} = f(a^{L-3})$$

And we can write the back-propagation as:

$$g^{L-4} = W^{L-3R} \ast g^{L-3} \circ f'(a^{L-4})$$

- The backward pass flips the weight matrix compared with the forward pass
- If the forward pass is a correlation, the backward pass is a convolution
- If the forward pass is a convolution, the backward pass is a correlation
- (Either is OK)
Implementing multilayer networks

Example at a time:

- **Input vector**: \( d \)
- **Weight matrix**: \( k \)
- **Output vector**: \( k \)

Diagram:

```
input vector
\[ d \]

weight matrix
\[ k \times k \]

output vector
\[ k \]
```
Implementing multilayer networks

Minibatch:

- Input vector (minibatch)
- Weight matrix
- Output vector (minibatch)

MLP Lecture 8
Convolutional Networks 2: Training, deep convolutional networks
Implementing multilayer networks

Minibatch:

input vector (minibatch)

weight matrix

output vector (minibatch)

input dimension x minibatch: Represent each layer as a 2-dimension matrix, where each row corresponds to a training example, and the number of minibatch examples is the number of rows.
Implementing Convolutional Networks

Example at a time, single input image, single feature map:

- Input image
- Weight matrix (kernel)
- Feature map

Diagram showing the process of convolution with labeled axes and shapes.
Implementing Convolutional Networks

Example at a time, single input image, multiple feature map:

input image \times \text{weight matrices (kernels)} \rightarrow \text{feature maps}
Implementing Convolutional Networks

Example at a time, multiple input images, multiple feature map:
Minibatch, multiple input images, multiple feature map:

- Minibatch of multiple input images
- Weight matrices (kernels)
- Minibatch of multiple feature maps
Implementing Convolutional Networks

- **Inputs / layer values:**
  - Each input image (and convolutional and pooling layer) is 2-dimensions (x,y)
  - If we have multiple feature maps, then that is a third dimension
  - And the minibatch adds a fourth dimension
  - Thus we represent each input (layer values) using a 4-dimension tensor (array):
    (minibatch-size, num-fmaps, x, y)

- **Weight matrices (kernels):**
  - Each weight matrix used to scan across an image has 2 spatial dimensions (x,y)
  - If there are multiple feature maps to be computed, then that is a third dimension
  - Multiple input feature maps adds a fourth dimension
  - Thus the weight matrices are also represented using a 4-dimension tensor:
    (num-fmaps-in, num-fmaps-out, x, y)
4D tensors in numpy

Both forward and back prop thus involves multiplying 4D tensors. There are various ways to do this:

- Explicitly loop over the dimensions: this results in simpler code, but can be inefficient. Although using cython to compile the loops as C can speed things up

- Serialisation: By replicating input patches and weight matrices, it is possible to convert the required 4D tensor multiplications into a large dot product. Requires careful manipulation of indices!

- Convolutions: use explicit convolution functions for forward and back prop, rotating for the backprop
Deep convolutional networks


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

Network Design

**Key design choices:**
- 3x3 conv. kernels – very small
- conv. stride 1 – no loss of information

**Other details:**
- Rectification (ReLU) non-linearity
- 5 max-pool layers (x2 reduction)
- no normalisation
- 3 fully-connected (FC) layers

![Diagram of the VGGNet architecture](image)
Deep Residual Learning ("ResNets")


http://arxiv.org/abs/1512.03385

Figure 2. Residual learning: a building block.

<table>
<thead>
<tr>
<th>method</th>
<th>top-1 err.</th>
<th>top-5 err.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG [41] (ILSVRC’14)</td>
<td>-</td>
<td>8.43(^{\dagger})</td>
</tr>
<tr>
<td>GoogLeNet [44] (ILSVRC’14)</td>
<td>-</td>
<td>7.89</td>
</tr>
<tr>
<td>VGG [41] (v5)</td>
<td>24.4</td>
<td>7.1</td>
</tr>
<tr>
<td>BN-inception [16]</td>
<td>21.99</td>
<td>5.81</td>
</tr>
<tr>
<td>ResNet-34 B</td>
<td>21.84</td>
<td>5.71</td>
</tr>
<tr>
<td>ResNet-34 C</td>
<td>21.53</td>
<td>5.60</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>20.74</td>
<td>5.25</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>19.87</td>
<td>4.60</td>
</tr>
<tr>
<td>ResNet-152</td>
<td><strong>19.38</strong></td>
<td><strong>4.49</strong></td>
</tr>
</tbody>
</table>
Summary

- Convolutional networks include local receptive fields, weight sharing, and pooling leading
- Backprop training can also be implemented as a “reverse” convolutional layer (with the weight matrix rotated)
- Implement using 4D tensors:
  - Inputs / Layer values: minibatch-size, number-fmaps, x, y
  - Weights: number-fmaps-in, number-fmaps-out, x, y
- Reading:
  Goodfellow et al, *Deep Learning* (ch 9)
  http://www.deeplearningbook.org/contents/convnets.html