Convolutional Networks (part 2)

Steve Renals

Machine Learning Practical — MLP Lecture 8
9 November 2016
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 – Pooling Layer

- Input 28x28
- 3x24x24 Feature Maps
- 3x12x12 Pooling Layers
- 6x8x8 Feature Maps
- 6x4x4 Pooling Layers
- fully connected
- sigmoid hidden layer
- fully connected softmax output layer

DELTA'S
\[ \delta^{l-3} = \delta^{l-2} G^{l-2\top} \circ f'(a^{l-3}) \]
[133x3]
\[ \delta^{l-2} = \delta^{l-1} W^{l-2\top} \]
\[ \delta^{l-1} = \delta^{l-1} W^{l-1\top} \circ f'(a^{l-1}) \]
\[ \delta^l = h^l - t^l \]

GRADIENTS
\[ \frac{\partial E}{\partial W^{l-1}} = h^{l-2\top} \delta^{l-1} \]
\[ \frac{\partial E}{\partial W^{l}} = h^{l-1\top} \delta^l \]
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 minibatch.
Training the convolutional layer is more complicated.
Training Convolutional Networks – Convolutional Layer

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

Simplify by only considering one feature map.

\[ \delta^{L-3} = \delta^{L-2} G^{L-2\top} \circ f'(a^{L-3}) \]
In the forward propagation, each hidden unit is connected to a region of input units (the receptive field)
In the forward propagation, each hidden unit is connected to a region of input units (the receptive field)
In the forward propagation, each hidden unit is connected to a region of input units (the receptive field).
In the forward propagation, each hidden unit is connected to a region of input units (the receptive field).
For backprop we need to consider the region of hidden units connected to each input unit.
For backprop we need to consider the region of hidden units connected to each input unit.

The top-left input unit (1,1) is connected to just one hidden unit.
For backprop we need to consider the region of hidden units connected to each input unit.

Input unit (2,2) is in the receptive fields of $2 \times 2 = 4$ hidden units.
For backprop we need to consider the region of hidden units connected to each input unit.

\[(3,3) \text{ is in the receptive fields of } 3 \times 3 = 9 \text{ hidden units}\]
For backprop we need to consider the region of hidden units connected to each input unit.

(4,4) is in the receptive fields of $4 \times 4 = 16$ hidden units
For backprop we need to consider the region of hidden units connected to each input unit.

(5,5) and all units away from the edge are in the receptive fields of $5 \times 5 = 25$ hidden units.
Convolutional Layer – Back Prop

For backprop we need to consider the region of hidden units connected to each input unit.

(5,5) and all units away from the edge are in the receptive fields of \(5 \times 5 = 25\) hidden units
As usual we want to back-propagate the $\delta$ values:

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

Look at the shared weights used for back prop:
As usual we want to back-propagate the $\delta$ values:

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

Look at the shared weights used for back prop:
As usual we want to back-propagate the $\delta$ values:

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

Look at the shared weights used for back prop:
As usual we want to back-propagate the $\delta$ values:

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

Look at the shared weights used for back prop:
As usual we want to back-propagate the $\delta$ values:

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

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

As usual we want to back-propagate the $\delta$ values:

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

Look at the shared weights used for back prop:

![Diagram showing convolutional layer connections and weights](image)
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 $180^\circ$ as shown before.
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 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 $180^\circ$ as shown before.
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° as shown before.
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 $180^\circ$ as shown before.
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 \textit{weight matrix is rotated by 180°} as shown before.
Back-propagation in the convolution layer, is also a convolution! But we have to rotate the weight matrix $W$ by $180^\circ$, $W^R$. Using the convolution operator we saw we can write the forward prop as:

$$h^{L-3} = \text{sigmoid}(W^{L-3} \ast h^{L-4} + b^{L-3})$$

And we can write the back-prop as:

$$\delta^{L-4} = W^{L-3}^R \ast \delta^{L-3} \circ f'(a^{L-4})$$
Implementing multilayer networks

Example at a time:

- Input vector
- Weight matrix
- Output vector

MLP Lecture 8  Convolutional Networks (part 2)
Implementing multilayer networks

Minibatch:

\[
\begin{array}{c}
\text{input vector} \\
\text{(minibatch)}
\end{array}
\begin{array}{c}
\text{weight matrix}
\end{array}
\begin{array}{c}
\text{output vector} \\
\text{(minibatch)}
\end{array}
\]
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

\[ \begin{array}{c}
\text{input image} \\
\text{weight matrix (kernel)} \\
\text{feature map}
\end{array} \]
Implementing Convolutional Networks

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

- Input image
- Weight matrices (kernels)
- Feature maps
Implementing Convolutional Networks

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

- Multiple input images
- Weight matrices (kernels)
- Feature maps
Implementing Convolutional Networks

Minibatch, multiple input images, multiple feature map:

- Minibatch of multiple input images
- Weight matrices (kernels)
- Minibatch of 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\(^1\) (array): \((\text{minibatch-size}, \text{num-fmaps}, x, y)\)

- **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: \((\text{num-fmaps-in}, \text{num-fmaps-out}, x, y)\)
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.
Recent advances using 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>Top-1</th>
<th>Top-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse coding [2]</td>
<td>47.1%</td>
<td>28.2%</td>
</tr>
<tr>
<td>SIFT + FVs [24]</td>
<td>45.7%</td>
<td>25.7%</td>
</tr>
<tr>
<td>CNN</td>
<td>37.5%</td>
<td>17.0%</td>
</tr>
</tbody>
</table>
ImageNet Classification ("VGGBNet")


http://www.robots.ox.ac.uk/~vgg/research/very_deep/

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

Total 134M wts
Deep Residual Learning ("ResNets")


http://arxiv.org/abs/1512.03385
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:
  Yoshua Bengio et al, *Deep Learning* (ch 9)
  http://goodfeli.github.io/dlbook/contents/convnets.html