DeepLearning on FPGAs
Artifical Neuronal Networks: Image classification

Sebastian Buschjäger

Technische Universität Dortmund - Fakultät Informatik - Lehrstuhl 8

November 9, 2016
Recap: Multilayer-Perceptrons

\[ w_{ij}^{(l+1)} \equiv \text{Weight from neuron } i \text{ in layer } l \text{ to neuron } j \text{ in layer } l + 1 \]
Backpropagation for sigmoid activation / RMSE loss

Gradient step:

\[
\begin{align*}
    w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta_j^{(l)} f_i^{(l-1)} \\
    b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)}
\end{align*}
\]

Recursion:

\[
\begin{align*}
    \delta_j^{(l-1)} &= f_j^{(l-1)} \left(1 - f_j^{(l-1)} \right) \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)} \\
    \delta_j^{(L)} &= - \left(y_i - f_j^{(L)} \right) f_j^{(L)} \left(1 - f_j^{(L)} \right)
\end{align*}
\]
Backpropagation for sigmoid activation / RMSE loss

Gradient step:

\[ w^{(l)}_{i,j} = w^{(l)}_{i,j} - \alpha \cdot \delta^{(l)}_j f_i \]

\[ b^{(l)}_j = b^{(l)}_j - \alpha \cdot \delta^{(l)}_j \]

Recursion:

\[ \delta^{(l-1)}_j = f_j^{(l-1)} \left( 1 - f_j^{(l-1)} \right) \sum_{k=1}^{M^{(l)}} \delta^{(l)}_k w^{(l)}_{j,k} \]

\[ \delta^{(L)}_j = \left( y_i - f_j^{(L)} \right) f_j^{(L)} \left( 1 - f_j^{(L)} \right) \]
Image classification

Our goal: Classify images with Deep learning

Recap: Neuronal Networks need vector input $\vec{x}$

Question: How are images represented?

Most simple representation: Bitmap of pixels

- Image has fixed number of pixels (height $\times$ width)
- Image has fixed number of color channels (e.g. RGB)
- Every pixel saves the color values of all color channels

Thus: An image is a matrix of pixels with multiple values (=vector) per entry

Sidenote: Mathematically this is called a tensor

Idea: Map every entry in the pixel matrix to exactly 1 input neuron
Image Representation: Example

Image: Matrix $M = [\vec{p}_{ij}]_{ij}$
Entry: $\vec{p}_{ij} = (r_{ij}, g_{ij}, b_{ij})^T$

Input neurons:
$\vec{x} = (r_{11}, g_{11}, b_{11}, r_{12}, g_{12}, \ldots)^T$

Example: 256 $\times$ 256 RGB image
$\Rightarrow 3 \cdot 256 \cdot 256 = 196,608$ input neurons
Image Representation

**Observation 1:** Even smaller images need a lot of neurons

- \( width \approx 256 - 1920 \)
- \( height \approx 256 - 1080 \)
- \( r_{ij}, g_{ij}, b_{ij} \in \{0, 1, \ldots, 255\} \)

**Observation 2:** This gets worse, if the neural network is “deep”

- Input-Layer: 196.608 neurons
- First hidden-layer: 1000 neurons
- Second hidden-layer: 100 neurons
- Output layer: 1 neuron

\[ 196.608 \cdot 1000 + 1000 \cdot 100 + 100 \cdot 1 = 196.708.100 \text{ weights} \]

**Thus:** Even for small images we need to learn a lot of weights
Image Representation: Making images smaller

Obviously: Images need to be smaller!

- Merge a $r \times r$ grid of pixels into a single pixel by applying reduction kernel channel-wise $k_c : \mathbb{N}^r \rightarrow \mathbb{N}$ over all pixels.
- By defining appropriate kernels, we can achieve smoothing, anti-aliasing etc.

Note: Pixel values are integers (e.g. $0 - 255$). Reduction kernels can be defined over $\mathbb{R}$, meaning $k_c : \mathbb{R}^r \rightarrow \mathbb{R}$. Then values need to be mapped to integers again:

$$\tilde{k}_c = \max(0, \min(255, \lfloor k_c \rfloor))$$

Thus: Assume appropriate mapping and use $k_c : \mathbb{R}^r \rightarrow \mathbb{R}$
Reduction kernel: Example

Simple and fast: Averaging \( k_c = \frac{1}{r} \sum_{i=1}^{r} c_i \)

<table>
<thead>
<tr>
<th>160</th>
<th>210</th>
<th>133</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>39</td>
<td>70</td>
<td>130</td>
</tr>
<tr>
<td>110</td>
<td>240</td>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>100</td>
<td>66</td>
<td>88</td>
<td>93</td>
</tr>
</tbody>
</table>

Padding: The way you handle unknown inputs (e.g. image-border)
Overlapping: The way you move the grid over the image
Here: Kernel is applied non-overlapping with no padding
Reduction kernel: Example

**Simple and fast:** Averaging \( k_c = \frac{1}{r} \sum_{i=1}^{r} c_i \)

\[
\begin{array}{cccc}
160 & 210 & 133 & 111 \\
88 & 39 & 70 & 130 \\
110 & 240 & 10 & 120 \\
100 & 66 & 88 & 93 \\
\end{array}
\]

\[
\left\lfloor (110 + 240 + 100 + 66) \cdot 0.25 \right\rfloor = 86
\]

**Padding:** The way you handle unknown inputs (e.g. image-border)

**Overlapping:** The way you move the grid over the image

**Here:** Kernel is applied non-overlapping with no padding
Reduction kernel: Example

**Simple and fast:** Averaging \( k_c = \frac{1}{r} \sum_{i=1}^{r} c_i \)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>210</td>
<td>133</td>
<td>111</td>
</tr>
<tr>
<td>88</td>
<td>39</td>
<td>70</td>
<td>130</td>
</tr>
<tr>
<td>110</td>
<td>240</td>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>100</td>
<td>66</td>
<td>88</td>
<td>93</td>
</tr>
</tbody>
</table>

\[ (10 + 120 + 88 + 93) \cdot 0.25 \] = 120

**Padding:** The way you handle unknown inputs (e.g. image-border)

**Overlapping:** The way you move the grid over the image

**Here:** Kernel is applied non-overlapping with no padding
Reduction kernel: Example

**Simple and fast:** Averaging \( k_c = \frac{1}{r} \sum_{i=1}^{r} c_i \)

\[
\begin{bmatrix}
160 & 210 & 133 & 111 \\
88 & 39 & 70 & 130 \\
110 & 240 & 10 & 120 \\
100 & 66 & 88 & 93
\end{bmatrix}
\]

\[
\left\lfloor (160 + 210 + 88 + 39) \cdot 0.25 \right\rfloor = 81
\]

**Padding:** The way you handle unknown inputs (e.g. image-border)

**Overlapping:** The way you move the grid over the image

**Here:** Kernel is applied non-overlapping with no padding
Reduction kernel: Example

**Simple and fast:** Averaging $k_c = \frac{1}{r} \sum_{i=1}^{r} c_i$

\[
\begin{array}{cccc}
160 & 210 & 133 & 111 \\
88 & 39 & 70 & 130 \\
110 & 240 & 10 & 120 \\
100 & 66 & 88 & 93 \\
\end{array}
\]

\[
\left\lfloor \frac{(133 + 111 + 70 + 130) \cdot 0.25}{} \right\rfloor = 153
\]

**Padding:** The way you handle unknown inputs (e.g. image-border)

**Overlapping:** The way you move the grid over the image

**Here:** Kernel is applied non-overlapping with no padding

DeepLearning on FPGAs
Image Representation: Making images smaller (2)

**Observation 1:** We can apply the same kernel in many different ways $\rightarrow$ Pixel-padding and/or overlapping might occur$^1$

---

**For now:** We assume non-overlapping application with no padding

**But:** Other application schemes can obviously be implemented

---

$^1$Animations see: [https://github.com/vdumoulin/conv_arithmetic](https://github.com/vdumoulin/conv_arithmetic)
Image Representation: Making images smaller (3)

Observation 2: The average kernel uses the same coefficient \( \frac{1}{r} \)

\[
k_c = \frac{1}{r} \sum_{i=1}^{r} c_i = \sum_{i=1}^{r} \frac{1}{r} \cdot c_i
\]

More general: Convolution using arbitrary weights \( w_i \)

\[
k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} * \vec{c}
\]

Note: This is basically a weighted sum!

But name-overloading here: Convolution is a well-known operation in signal processing and statistics
Convolution: Some intuitions

**In system theory:** Given a system with a transfer-function $f$ we can compute its reaction to an input signal $g$ by computing the convolution $f \ast g = \int f(\tau)g(t - \tau)d\tau$

**In statistics:** Given two time series as continuous functions $f$ and $g$, we can measure the similarity of these two functions by computing the cross-correlation $f \ast g = \int f(\tau)g(t + \tau)d\tau$

**Note:** Both are basically the same with different perspective and a slightly different index-shift

**Bottom-Line:** A kernel reacts to specific parts of a function / signal / image, thus **filtering** out important features

⇒ This is some kind of feature extraction
Convolution: Example

**Note:** In discrete convolution integrals become summation:

\[
k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} \ast \vec{c}
\]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>20</td>
<td>153</td>
<td>11</td>
</tr>
<tr>
<td>122</td>
<td>39</td>
<td>70</td>
<td>200</td>
</tr>
<tr>
<td>180</td>
<td>80</td>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>45</td>
<td>140</td>
</tr>
</tbody>
</table>

image

* 

kernel / weights / filter

result
Convolution: Example

**Note:** In discrete convolution integrals become summation:

\[ k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} \ast \vec{c} \]

<table>
<thead>
<tr>
<th>170</th>
<th>20</th>
<th>153</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>122</td>
<td>39</td>
<td>70</td>
<td>200</td>
</tr>
<tr>
<td>180</td>
<td>80</td>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>45</td>
<td>140</td>
</tr>
</tbody>
</table>

image

<table>
<thead>
<tr>
<th>1</th>
<th>-0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.5</td>
<td>1</td>
</tr>
</tbody>
</table>

kernel / weights / filter

result

\[ 180 \cdot 1 - 80 \cdot 0.5 - 20 \cdot 0.5 + 120 \cdot 1 = 250 \]
Convolution: Example

**Note:** In discrete convolution integrals become summation:

\[
k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} \ast \vec{c}
\]

<table>
<thead>
<tr>
<th>170</th>
<th>20</th>
<th>153</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>122</td>
<td>39</td>
<td>70</td>
<td>200</td>
</tr>
<tr>
<td>180</td>
<td>80</td>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>45</td>
<td>140</td>
</tr>
</tbody>
</table>

*  \[
\begin{bmatrix}
1 & -0.5 \\
-0.5 & 1
\end{bmatrix}
\]

=  \[
\begin{bmatrix}
250 & 67
\end{bmatrix}
\]

\[
10 \cdot 1 - 120 \cdot 0.5 - 45 \cdot 0.5 + 140 \cdot 1 = 67
\]
Convolution: Example

**Note:** In discrete convolution integrals become summation:

\[ k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} \ast \vec{c} \]

\[
\begin{array}{cccc}
170 & 20 & 153 & 11 \\
122 & 39 & 70 & 200 \\
180 & 80 & 10 & 120 \\
20 & 120 & 45 & 140 \\
\end{array}
\]

\[
\begin{array}{cc}
1 & -0.5 \\
-0.5 & 1 \\
\end{array}
\]

\[
\begin{array}{cc}
138 \\
250 & 67 \\
\end{array}
\]

\[
170 \cdot 1 - 20 \cdot 0.5 - 122 \cdot 0.5 + 39 \cdot 1 = 138
\]

image \hspace{2cm} kernel / weights / filter \hspace{2cm} result
Convolution: Example

**Note:** In discrete convolution integrals become summation:

\[ k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} \ast \vec{c} \]

\[
\begin{array}{cccc}
170 & 20 & 153 & 11 \\
122 & 39 & 70 & 200 \\
180 & 80 & 10 & 120 \\
20 & 120 & 45 & 140 \\
\end{array}
\]

\[
\begin{array}{cc}
1 & -0.5 \\
-0.5 & 1 \\
\end{array}
\]

\[
\begin{array}{cc}
138 & 255 \\
250 & 67 \\
\end{array}
\]

\[
153 \cdot 1 - 11 \cdot 0.5 - 70 \cdot 0.5 + 200 \cdot 1 = 255
\]

image

kernel / weights / filter

result
Convolutional neural networks (CNN)

Observation 1: Convolution can reduce the size of images
Observation 2: Convolution can perform feature extraction
Observation 3: Neural networks can learn weights $\vec{w}$
$\Rightarrow$ Convolutional neural networks (CNN) (~ LeCun, 1989)

Idea: Every convolutional layer has its own weight matrix
- Move convolution kernel over input data (with padding etc.)
- Apply activation function to create another (smaller) image
- Once the images are small enough, use fully connected layers
- During backpropagation, compute errors for the kernel weights

Question: How do we compute the kernel weights?
Short: Use backpropagation - Long: We need some more notation
CNNs: Some remarks

**Note 1:** Since convolution is used internally, there is no need for mapping values *inside* the net $\rightarrow$ use computed values directly

**Note 2:** The size of the resulting image depends on the size of your convolution kernel and your padding / overlapping approach

**Note 3:** The kernel matrix is *shared* between multiple input neurons $\rightarrow$ A $5 \times 5$ convolutional layer only has 25 parameters!

**Note 4:** Since the kernel is moved over the whole input image, we can extract features in every location

**Note 5:** CNNs somewhat model receptive fields in biology
CNN: Notation and weight sharing

<table>
<thead>
<tr>
<th></th>
<th>$f_{00}$</th>
<th>$f_{01}$</th>
<th>$f_{02}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{10}$</td>
<td>$f_{11}$</td>
<td>$f_{12}$</td>
<td></td>
</tr>
<tr>
<td>$f_{20}$</td>
<td>$f_{21}$</td>
<td>$f_{22}$</td>
<td></td>
</tr>
</tbody>
</table>

Input $\vec{f}$  

Weights $\vec{w}$

Output $\vec{y}$

\[
\begin{align*}
\vec{y} &= w_{00} f_{00} + w_{01} f_{01} + w_{10} f_{10} + w_{11} f_{11} + w_{00} f_{10} + w_{01} f_{11} + w_{00} f_{20} + w_{01} f_{21} \\
&+ w_{10} f_{10} + w_{11} f_{11} + w_{10} f_{20} + w_{11} f_{21}
\end{align*}
\]
CNN: Notation and weight sharing

\[
\begin{array}{ccc}
f_{00} & f_{01} & f_{02} \\
f_{10} & f_{11} & f_{12} \\
f_{20} & f_{21} & f_{22} \\
\end{array}
\]

\[
\begin{array}{cc}
w_{00} & w_{01} \\
w_{10} & w_{11} \\
\end{array}
\]

\[
\begin{array}{cc}
w_{00}f_{00} + w_{01}f_{01} + w_{10}f_{10} + w_{11}f_{11} + b_{00} + b_{01} \\
w_{00}f_{10} + w_{01}f_{11} + w_{10}f_{20} + w_{11}f_{21} + b_{10} + b_{11} \\
\end{array}
\]

\[
\begin{array}{cc}
w_{00}f_{01} + w_{01}f_{02} + w_{10}f_{11} + w_{11}f_{12} + b_{00} + b_{01} \\
w_{00}f_{11} + w_{01}f_{12} + w_{10}f_{21} + w_{11}f_{22} + b_{10} + b_{11} \\
\end{array}
\]

\[
\begin{aligned}
\mathbf{y}^{(l)}_{i,j} &= \sum_{i'=0}^{M^{(l)}} \sum_{j'=0}^{M^{(l)}} w^{(l)}_{i,j} \cdot f^{(l-1)}_{i+i',j+j'} + b^{(l)}_{i,j} = w^{(l)} \ast f^{(l-1)} + b^{(l)} \\
f^{(l)}_{i,j} &= \sigma(y^{(l)}_{i,j})
\end{aligned}
\]

Mathematically (here with cross-correlation):
CNN: How to compute $\frac{\partial E}{\partial w_{i,j}^{(l)}}$ and $\frac{\partial E}{\partial b_{i,j}^{(l)}}$?

Mathematically (here with cross-correlation):

$$y_{i,j}^{(l)} = \sum_{i'=0}^{M^{(l)}} \sum_{j'=0}^{M^{(l)}} w_{i,j}^{(l)} \cdot f_{i+i',j+j'}^{(l-1)} + b_{i,j}^{(l)} = w^{(l)} \ast f^{(l-1)} + b^{(l)}$$

$$f_{i,j}^{(l)} = \sigma(y_{i,j}^{(l)})$$

$M^{(l)} \times M^{(l)}$ bias matrix!
Backpropagation for sigmoid activation

Gradient step:

\[ w^{(l)}_{i,j} = w^{(l)}_{i,j} - \alpha \cdot \delta^{(l)} \cdot \text{rot180}(f)^{(l-1)} f^{(l-1)}_{i,j} \]

\[ b^{(l)}_j = b^{(l)}_j - \alpha \cdot \delta^{(l)}_j \]

Recursion:

\[ \delta^{(l+1)} = \delta^{(l)} \cdot \text{rot180}(w^{(l+1)}) \cdot f^{(l)}_{i,j} (1 - f_{i,j})^l \]
Backpropagation for sigmoid activation

**Gradient step:**

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \cdot \delta^{(l)} \cdot \text{rot180}(f)^{(l-1)} \cdot f_{i,j}^{(l-1)}$$

$$b_{j}^{(l)} = b_{j}^{(l)} - \alpha \cdot \delta_{j}^{(l)}$$

**Recursion:**

$$\delta^{(l+1)} = \delta^{(l)} \cdot \text{rot180}(w^{(l+1)}) \cdot f_{i,j}^{(l)}(1 - f_{i,j})^{l}$$
Backpropagation for activation $h$

**Gradient step:**

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \cdot \delta^{(l)} \ast \text{rot180}(f)^{(l-1)} f_{i,j}^{(l-1)}$$

$$b_j^{(l)} = b_j^{(l)} - \alpha \cdot \delta_j^{(l)}$$

**Recursion:**

$$\delta^{(l+1)} = \delta^{(l)} \ast \text{rot180}(w^{(l+1)}) \cdot \frac{\partial h(y_i^{(l)})}{\partial y_i^{(l)}}$$

**Observation:** A convolution during forward-step results in cross-correlation on the backward step and vice-versa

**Note:** The values (and thus positions) of the weights are learnt

**Thus:** Does not matter if we implement convolution or cross-correlation. Just need to “reverse” it during backprop.
CNN: Some architectural remarks

So far: We assumed 1 color channel - what about 3 channels?

Idea 1: Merge color channels into single value

- **Average:** \((r_{i,j} + g_{i,j} + b_{i,j}) / 3\)
- **Lightness:** \((\max(r_{i,j}, g_{i,j}, b_{i,j}) - \min(r_{i,j}, g_{i,j}, b_{i,j})) / 2\)
- **Luminosity:** \(0.21r_{i,j} + 0.72g_{i,j} + 0.07b_{i,j}\)

Observation: Average and Luminosity look like weighted sums...

→ Given \(k^{(l)}\) input channels in layer \(l\), for every pixel \(j\) do:

\[
\begin{align*}
  f^{(l)}_j &= h \left( \sum_{k=1}^{k^{(l)}} f^{(l-1)}_j \cdot w^{(l)}_{k,j} + b_j \right)
\end{align*}
\]

Thus: Use standard backprop. to learn weights
CNN: Some architectural remarks (2)

Idea 2: Use 1 weight matrix per channel and extract 1 feature

More general: Perform $k^{(l)}$ convolutions per layer

- Use and learn $k^{(l)}$ weight matrices per layer
- Generating $k^{(l)}$ smaller images per layer
- So that multiple features are extracted per layer

⇒ Build a tree-like convolution structure, where more sophisticated features are extracted based on already extracted features

Finally: Use fully connected layers to perform classification

Usually: A combination is used between feature extraction and channel reduction
CNN: Example

Source: [http://www.ais.uni-bonn.de/deep_learning/images/Convolutional_NN.jpg](http://www.ais.uni-bonn.de/deep_learning/images/Convolutional_NN.jpg)
CNN: Some architectural remarks (3)

**Sometimes:** We want to reduce the image size even further without too much computation

**Downsampling/Pooling:** Merge a $r \times r$ grid into a single pixel

- **Max:** $f_{i,j}^{(l)} = \max (p_{i,j}, p_{i,j+1}, \ldots p_{i+r,j+r})$
- **Avg:** $f_{i,j}^{(l)} = \frac{1}{r \cdot r} \sum_{i'=0}^{r} \sum_{j'=0}^{r} p_{i+i',j+j'}$
- **Sum:** $f_{i,j}^{(l)} = \sum_{i'=0}^{r} \sum_{j'=0}^{r} p_{i+i',j+j'}$

**Note:** This is the same as convolution, but without parameters

**Thus:** No backpropagation-step needed for this layer

$\Rightarrow$ Just “upsample” delta-values from next layer and backward upsampled values to the previous layer
Neural Networks and generalization

Recap: Overfitting can happen if we learn the training data without any generalization

Typical approach: Force the model to generalise from the data by limiting the number of parameters to be used

Formal: This is called regularization

- **Per construction**: Define network with less parameters
- **Per dropout**: Randomly ignore values of certain neurons
  - During forward computation, set output of random neuron to 0
  - Network has now to deal with missing neurons and thus will include some redundancy
- **Per loss function**: Use loss function that punishes overfitting
  - **Obviously 1**: If a parameter is near 0, it is not used
  - **Obviously 2**: Fewer parameters means less overfitting
  - **Thus**: Punish large absolute parameter values $||w^{(l)}||$
Neural Networks and generalization (2)

\[ \ell(D, \hat{\theta}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - f_\hat{\theta}(\vec{x}_i))^2 + \lambda \sum_{l} ||\vec{w}(l)||} \]

\[ \ell(D, \hat{\theta}) = -\frac{1}{N} \sum_{i=1}^{N} (y_i \ln (f_\hat{\theta}(\vec{x}_i)) + (1 - y_i) \ln (1 - f_\hat{\theta}(\vec{x}_i))) + \lambda \sum_{l} ||\vec{w}(l)|| \]

**Note 1:** You’ll need to re-compute the derivative for backprop.

**Note 2:** This form of regularization is mathematically sound, but computationally intensive → we have to go over all matrices

**Note 3:** Here we used \( \ell_2 \) norm - more general \( p \)-Norm

\[ ||x||_p = \left( \sum_{i=0}^{n} |x_i| \right)^{\frac{1}{p}} \]
CNN: Some implementation remarks

**Obviously 1:** Convolution is a special kind of layer
→ implementation should be freely combinable with activation function and other layers

**Note:** Size of input is problem specific, size of kernel is a user parameter, number of kernels is also a user parameter

**But:** Size of output also depends on padding / striding approach
→ For convenience layer-sizes should be automatically computed
→ For compilers layer-sizes should be known at compile time
⇒ Define a compile-time macro / template for easier programming, but high speed implementation

**Obviously 2:** Pooling is a special kind of layer

**Note:** Backprop. is not required here, but just correct sampling
CNN: Some implementation remarks (2)

**Parallelism:** Neural network offer three kind of parallelism

**First:** On feature-extraction level

→ We can perform every convolution per layer in full parallel

**Note:** This requires some form of synchronization once we reach the fully-connected layer

**Second:** On computational level

→ A convolution requires $r \times r$ independent multiplications

$$
\sum_{i' = 0}^{M^{(l)}} \sum_{j' = 0}^{M^{(l)}} w_{i,j}^{(l)} \cdot f_{i+i',j+j'}^{(l-1)} + b_{i,j}^{(l)} = w^{(l)} \ast f^{(l-1)} + b^{(l)}
$$

**Additionally:** Activation function needs to be evaluated independently for every pixel
CNN: Some implementation remarks (3)

**Question:** On gradient level
- Perform gradient computations in parallel on parts of the data
- Compute mini-batches in parallel

**Note:**
1) is always possible for convolutional networks
2) is usually done by the compiler, if the system supports vectorization instructions (More later)
3) is always possible, but will result in stochastic gradient descent. Thus we don’t have a theoretical guarantee for convergence anymore, but it works well in practice.
CNN: Network architecture

**Question:** So what's a good network architecture?

**Answer:** As always, depends on the problem. But the same general ideas as with MLPs still hold.

**Additionally for image classification:**

- Grayscale images usually give already a fair performance
- Input images should have the same dimension
- Convolution kernels should be large enough to capture features, but small enough to be fast to compute. Usually we use \(3 \times 3 - 7 \times 7\)
- Convolution tends to overfit, so regularization should be used
- Deeper architectures usually perform well with pooling
Summary

Important concepts:
- **Convolution** is an important concept in image classification
  - We can extract image features on every part of the image
  - We share parameters in small kernel matrices
- For **image classification** we combine convolution layers and fully-connected layers with backpropagation
- Sometimes pooling is necessary
- Sometimes regularization is necessary

**Homework** until next meeting
- Extend your backpropagation implementation to a more general approach → variable number of neurons etc.
- Design a neuronal network for the MNIST data-set
  (Note: convolution is not required yet)

What's your accuracy?