DeepLearning on FPGAs

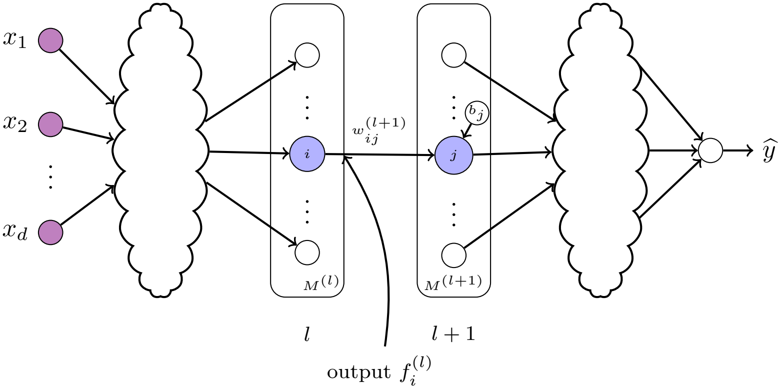
Artifical Neuronal Networks: Image classification

Sebastian Buschjäger

Technische Universität Dortmund - Fakultät Informatik - Lehrstuhl 8

November 9, 2016

Recap: Multilayer-Perceptrons



$\mathbf{w_{i,j}^{(l+1)}} \widehat{=}$ Weight from neuron i in layer l to neuron j in layer $l+1$

Backpropagation for sigmoid activation / RMSE loss

Gradient step:

$$\begin{aligned} w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta_j^{(l)} f_i^{(l-1)} \\ b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)} \end{aligned}$$

Recursion:

$$\begin{aligned} \delta_j^{(l-1)} &= f_j^{(l-1)} \left(1 - f_j^{(l-1)}\right) \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)} \\ \delta_j^{(L)} &= -\left(y_i - f_j^{(L)}\right) f_j^{(L)} \left(1 - f_j^{(L)}\right) \end{aligned}$$

Backpropagation for sigmoid activation / RMSE loss

Gradient step:

$$\begin{aligned} w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta_j^{(l)} f_i^{(l-1)} \\ b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)} \end{aligned}$$

Recursion:

$$\begin{aligned} \delta_j^{(l-1)} &= \underbrace{\boxed{f_j^{(l-1)} \left(1 - f_j^{(l-1)}\right)}}_{\text{derivative of activation function}} \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)} \\ \delta_j^{(L)} &= \underbrace{\boxed{-\left(y_i - f_j^{(L)}\right)}}_{\text{derivative of loss function}} \underbrace{\boxed{f_j^{(L)} \left(1 - f_j^{(L)}\right)}}_{\text{derivative of activation function}} \end{aligned}$$

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 if pixels (height × 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

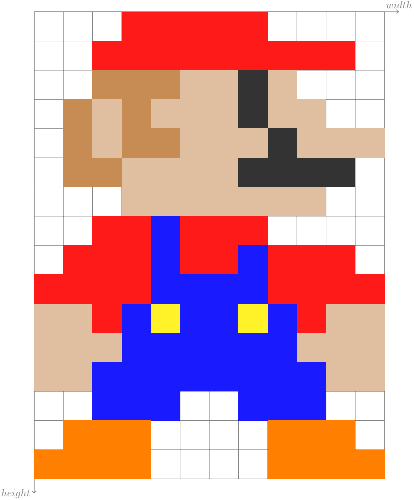
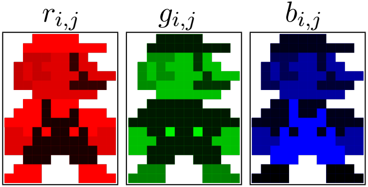


Image: Matrix $M = [\vec{p}_{ij}]_{ij}$
Entry: $\vec{p}_{ij} = (r_{ij}, g_{ij}, b_{ij})^T$



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

Example: 256×256 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

$\Rightarrow 196.608 \cdot 1000 + 1000 \cdot 100 + 100 \cdot 1 = 196.708.100$ 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-alising 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

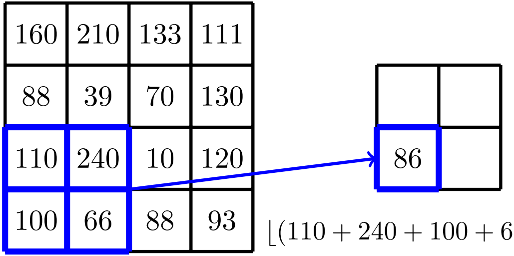
Simple and fast: Averaging $k_c = \frac{1}{r}\sum_{i=1}^{r} c_i$

160	210	133	111
88	39	70	130
110	240	10	120
100	66	88	93

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 $k_c = \frac{1}{r}\sum_{i=1}^{r} c_i$



160	210	133	111
88	39	70	130
110	240	10	120
100	66	88	93

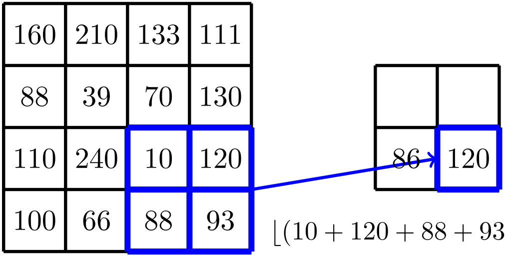
86	

$\lfloor(110 + 240 + 100 + 66) \cdot 0.25\rfloor = 86$

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 $k_c = \frac{1}{r}\sum_{i=1}^{r} c_i$



160	210	133	111
88	39	70	130
110	240	10	120
100	66	88	93

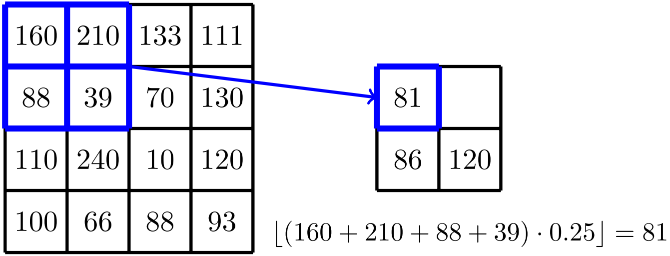
86	120

$\lfloor (10 + 120 + 88 + 93) \cdot 0.25 \rfloor = 120$

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 $k_c = \frac{1}{r}\sum_{i=1}^{r} c_i$



160	210	133	111
88	39	70	130
110	240	10	120
100	66	88	93

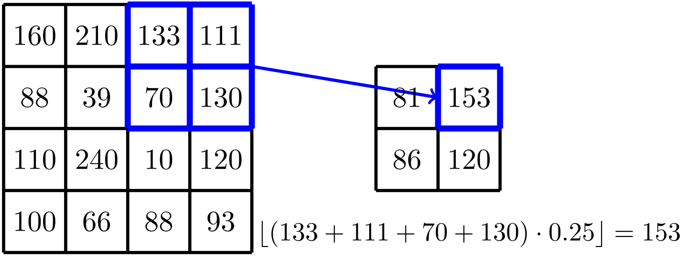
81	
86	120

$$\lfloor (160 + 210 + 88 + 39) \cdot 0.25 \rfloor = 81$$

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 $k_c = \frac{1}{r}\sum_{i=1}^{r} c_i$



160	210	133	111
88	39	70	130
110	240	10	120
100	66	88	93

81	153
86	120

$\lfloor (133 + 111 + 70 + 130) \cdot 0.25 \rfloor = 153$

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

Image Representation: Making images smaller (2)

Observation 1: We can apply the same kernel in many different ways → 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

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 $f * g = \int f(\tau)g(t-\tau)d\tau$

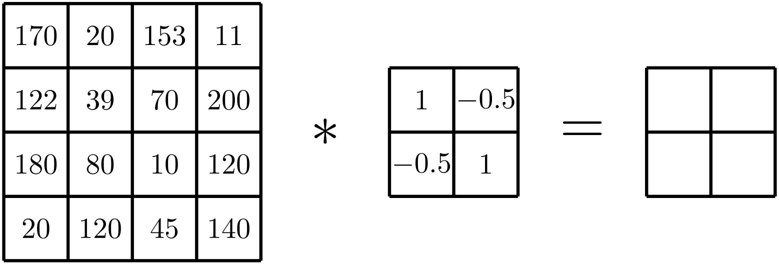
In statistics: Given two time series as continuous functions f and g, we can measure the similarity of these two functions by computing the cross-correlation $f \star g = \int f(\tau)g(t+\tau)d\tau$

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:

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



$$\begin{array}{|c|c|c|c|}\hline 170 & 20 & 153 & 11 \\ \hline 122 & 39 & 70 & 200 \\ \hline 180 & 80 & 10 & 120 \\ \hline 20 & 120 & 45 & 140 \\ \hline \end{array} * \begin{array}{|c|c|}\hline 1 & -0.5 \\ \hline -0.5 & 1 \\ \hline \end{array} = \begin{array}{|c|c|}\hline & \\ \hline & \\ \hline \end{array}$$

image

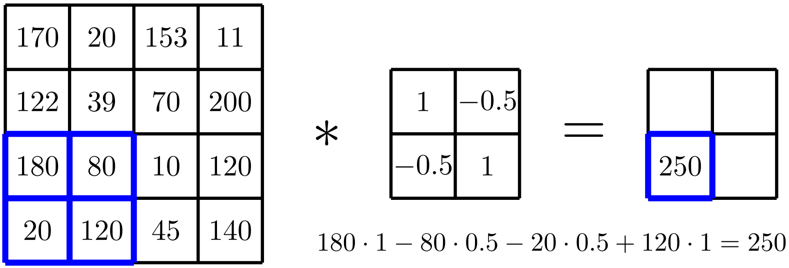
kernel / weights / filter

result

Convolution: Example

Note: In discrete convolution integrals become summation:

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



170	20	153	11
122	39	70	200
180	80	10	120
20	120	45	140

*

1	−0.5
−0.5	1

=

250	

$$180 \cdot 1 - 80 \cdot 0.5 - 20 \cdot 0.5 + 120 \cdot 1 = 250$$

image

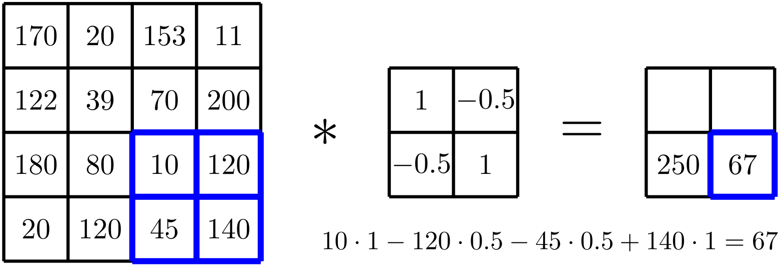
kernel / weights / filter

result

Convolution: Example

Note: In discrete convolution integrals become summation:

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



$$\begin{array}{|c|c|c|c|}\hline 170 & 20 & 153 & 11 \\ \hline 122 & 39 & 70 & 200 \\ \hline 180 & 80 & 10 & 120 \\ \hline 20 & 120 & 45 & 140 \\ \hline \end{array} * \begin{array}{|c|c|}\hline 1 & -0.5 \\ \hline -0.5 & 1 \\ \hline \end{array} = \begin{array}{|c|c|}\hline & \\ \hline 250 & 67 \\ \hline \end{array}$$

$$10 \cdot 1 - 120 \cdot 0.5 - 45 \cdot 0.5 + 140 \cdot 1 = 67$$

image

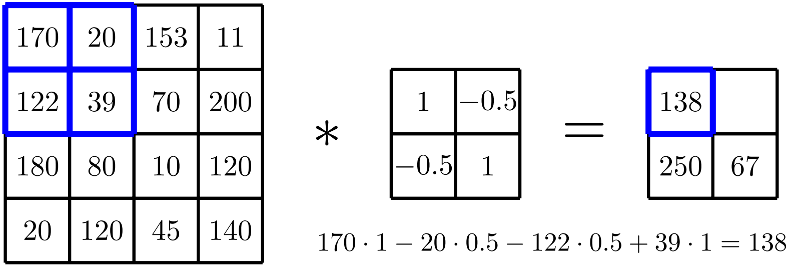
kernel / weights / filter

result

Convolution: Example

Note: In discrete convolution integrals become summation:

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



170	20	153	11
122	39	70	200
180	80	10	120
20	120	45	140

$*$

1	−0.5
−0.5	1

$=$

138	
250	67

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

image

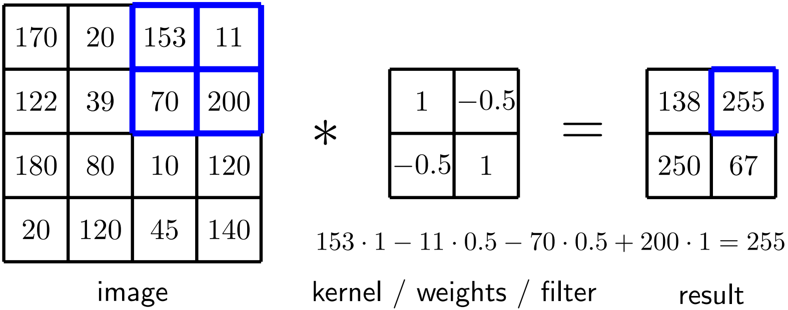
kernel / weights / filter

result

Convolution: Example

Note: In discrete convolution integrals become summation:

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



170	20	153	11
122	39	70	200
180	80	10	120
20	120	45	140

image

$*$

1	−0.5
−0.5	1

kernel / weights / filter

$=$

138	255
250	67

result

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

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) ($\sim$ **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 → 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 → A 5×5 convolutional layer only has 25 parameters!

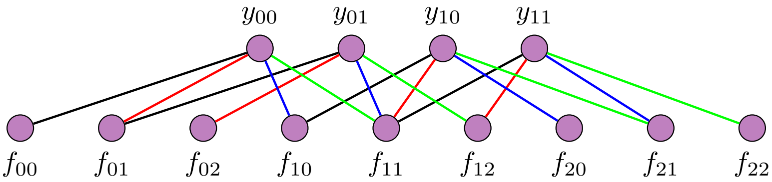
Note 4: Since the kernel is moved over the whole input image, we can extract features in every location

Note 5: CNNs somewhat model receptive fields in biology

CNN: Notation and weight sharing

$$\begin{array}{|c|c|c|}\hline f_{00} & f_{01} & f_{02} \\ \hline f_{10} & f_{11} & f_{12} \\ \hline f_{20} & f_{21} & f_{22} \\ \hline \end{array} * \begin{array}{|c|c|}\hline w_{00} & w_{01} \\ \hline w_{10} & w_{11} \\ \hline \end{array} = \begin{array}{|l|l|}\hline w_{00}f_{00} + w_{01}f_{01} & w_{00}f_{01} + w_{01}f_{02} \\ + w_{10}f_{10} + w_{11}f_{11} & + w_{10}f_{11} + w_{11}f_{12} \\ \hline w_{00}f_{10} + w_{01}f_{11} & w_{00}f_{11} + w_{01}f_{12} \\ + w_{10}f_{20} + w_{11}f_{21} & + w_{10}f_{21} + w_{11}f_{22} \\ \hline \end{array}$$

input $\vec{f}$ weights $\vec{w}$ output $\vec{y}$



CNN: Notation and weight sharing

$$\begin{array}{|c|c|c|}\hline f_{00} & f_{01} & f_{02} \\ \hline f_{10} & f_{11} & f_{12} \\ \hline f_{20} & f_{21} & f_{22} \\ \hline \end{array} * \begin{array}{|c|c|}\hline w_{00} & w_{01} \\ \hline w_{10} & w_{11} \\ \hline \end{array} = \begin{array}{|l|l|}\hline \begin{array}{l} w_{00}f_{00}+w_{01}f_{01} \\ +w_{10}f_{10}+w_{11}f_{11} \end{array} & \begin{array}{l} w_{00}f_{01}+w_{01}f_{02} \\ +w_{10}f_{11}+w_{11}f_{12} \end{array} \\ \hline \begin{array}{l} w_{00}f_{10}+w_{01}f_{11} \\ +w_{10}f_{20}+w_{11}f_{21} \end{array} & \begin{array}{l} w_{00}f_{11}+w_{01}f_{12} \\ +w_{10}f_{21}+w_{11}f_{22} \end{array} \\ \hline \end{array}$$

input $\vec{f}$ weights $\vec{w}$ output $\vec{y}$

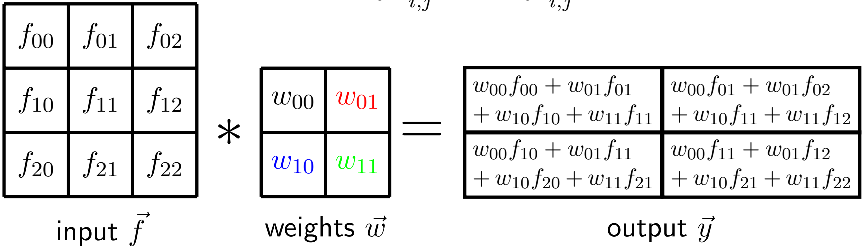
Mathematically (here with cross-correlation):

$$y^{(l)}_{i,j} = \sum_{i'=0}^{M^{(l)}} \sum_{j'=0}^{M^{(l)}} w^{(l)}_{i,j} \cdot f^{(l-1)}_{i+i',j+j'} + \boxed{b^{(l)}_{i,j}} = w^{(l)} * f^{(l-1)} + \boxed{b^{(l)}}$$

$$f^{(l)}_{i,j} = \sigma(y^{(l)}_{i,j})$$

$M^{(l)} \times M^{(l)}$ bias **matrix**!

CNN: How to compute $\frac{\partial E}{\partial w^{(l)}_{i,j}}$ and $\frac{\partial E}{\partial b^{(l)}_{i,j}}$?



$$\begin{array}{|c|c|c|}\hline f_{00} & f_{01} & f_{02} \\ \hline f_{10} & f_{11} & f_{12} \\ \hline f_{20} & f_{21} & f_{22} \\ \hline \end{array} * \begin{array}{|c|c|}\hline w_{00} & w_{01} \\ \hline w_{10} & w_{11} \\ \hline \end{array} = \begin{array}{|l|l|}\hline w_{00}f_{00}+w_{01}f_{01} & w_{00}f_{01}+w_{01}f_{02} \\ +w_{10}f_{10}+w_{11}f_{11} & +w_{10}f_{11}+w_{11}f_{12} \\ \hline w_{00}f_{10}+w_{01}f_{11} & w_{00}f_{11}+w_{01}f_{12} \\ +w_{10}f_{20}+w_{11}f_{21} & +w_{10}f_{21}+w_{11}f_{22} \\ \hline \end{array}$$

input $\vec{f}$ weights $\vec{w}$ output $\vec{y}$

Mathematically (here with cross-correlation):

$$y^{(l)}_{i,j} = \sum_{i'=0}^{M^{(l)}} \sum_{j'=0}^{M^{(l)}} w^{(l)}_{i,j} \cdot f^{(l-1)}_{i+i',j+j'} + \boxed{b^{(l)}_{i,j}} = w^{(l)} * f^{(l-1)} + \boxed{b^{(l)}}$$

$$f^{(l)}_{i,j} = \sigma(y^{(l)}_{i,j})$$

$M^{(l)} \times M^{(l)}$ bias **matrix**!

• • •

• • •

• • •

Backpropagation for sigmoid activation

Gradient step:

$$\begin{aligned} w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta^{(l)} * rot180(f)^{(l-1)} f_{i,j}^{(l-1)} \\ b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)} \end{aligned}$$

Recursion:

$$\delta^{(l+1)} = \delta^{(l)} * rot180(w^{(l+1)}) \cdot f_{i,j}^{(l)} (1 - f_{i,j})^{l}$$

Backpropagation for sigmoid activation

Gradient step:

$$\begin{aligned} w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta^{(l)} * \boxed{rot180(f)^{(l-1)}}\, f_{i,j}^{(l-1)} \\ b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)} \end{aligned}$$

Recursion:

$$\delta^{(l+1)} = \delta^{(l)} * \boxed{rot180(w^{(l+1)})} \cdot f_{i,j}^{(l)}(1 - f_{i,j})^{l}$$

$$\boxed{rot180 \begin{array}{|c|c|} \hline {\color{blue}w_{10}} & {\color{green}w_{11}} \\ \hline w_{00} & {\color{red}w_{01}} \\ \hline \end{array} = \begin{array}{|c|c|} \hline {\color{red}w_{01}} & w_{00} \\ \hline {\color{green}w_{11}} & {\color{blue}w_{10}} \\ \hline \end{array}}$$

Backpropagation for activation h

Gradient step:

$$\begin{aligned} w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta^{(l)} * rot180(f)^{(l-1)} f_{i,j}^{(l-1)} \\ b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)} \end{aligned}$$

Recursion:

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

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

Observation: Average and Luminosity look like weighted sums...
→ Given $k^{(l)}$ input channels in layer l, for every pixel j do:

$$f_j^{(l)} = h\left(\sum_{k=1}^{k^{(l)}} f_j^{(l-1)} \cdot w_{k,j}^{(l)} + b_j\right)$$

Thus: Use standard backprop. to learn weights

CNN: Some architectural remarks (2)

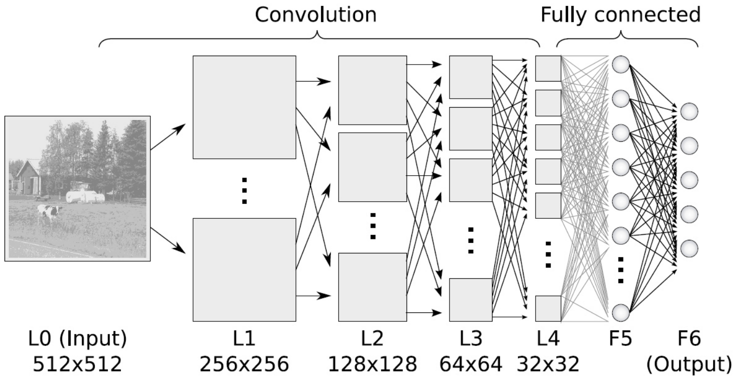
Idea 2: Use 1 weight matrix per channel and extract 1 feature
More general: Perform $k^{(l)}$ convolutions per layer

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

$\Rightarrow$ 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[2]



---

[2] Source: 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:** $f_{i,j}^{(l)} = max\left(p_{i,j}, p_{i,j+1}, \ldots p_{i+r,j+r}\right)$
- **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

Typicall 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)

$$\ell(\mathcal{D}, \widehat{\theta}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(y_i - f_{\widehat{\theta}}(\vec{x}_i)\right)^2} + \lambda \sum_{l} ||\vec{w}^{(l)}||$$

$$\ell(\mathcal{D}, \widehat{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \ln\left(f_{\widehat{\theta}}(\vec{x}_i)\right) + (1 - y_i)\ln\left(1 - f_{\widehat{\theta}}(\vec{x}_i)\right)\right) + \lambda \sum_{l} ||\vec{w}^{(l)}||$$

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 ℓ_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 convienience 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 parallism

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

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

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-batchs in parallel
Note:

1) is always possible for convolutional networks
2) is usally 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 dont have a theoretical guarantee for convergence anymore, but it works well in practice.

CNN: Network architecture

Question: So whats 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:

- Grayscaled 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 neurnal network for the MNIST data-set (Note: convolution is not required yet)

Whats you accuracy?