# Fast Convolutional Neural Network Training and Classification on CUDA GPUs

Master Thesis in Computer Science

Daniel Strigl
Klaus Kofler

Distributed and Parallel Systems Group
Institute of Computer Science
University of Innsbruck

Supervisor: Dr. Stefan Podlipnig

# Overview

- The Goal
- A Review of Neural Networks
- Convolutional Neural Networks
- Implementation
- Benchmarks and Results
- Applications of CNNs
  - Convolutional Face Detector
  - Handwritten Digit Recognition
- Conclusion and Future Work

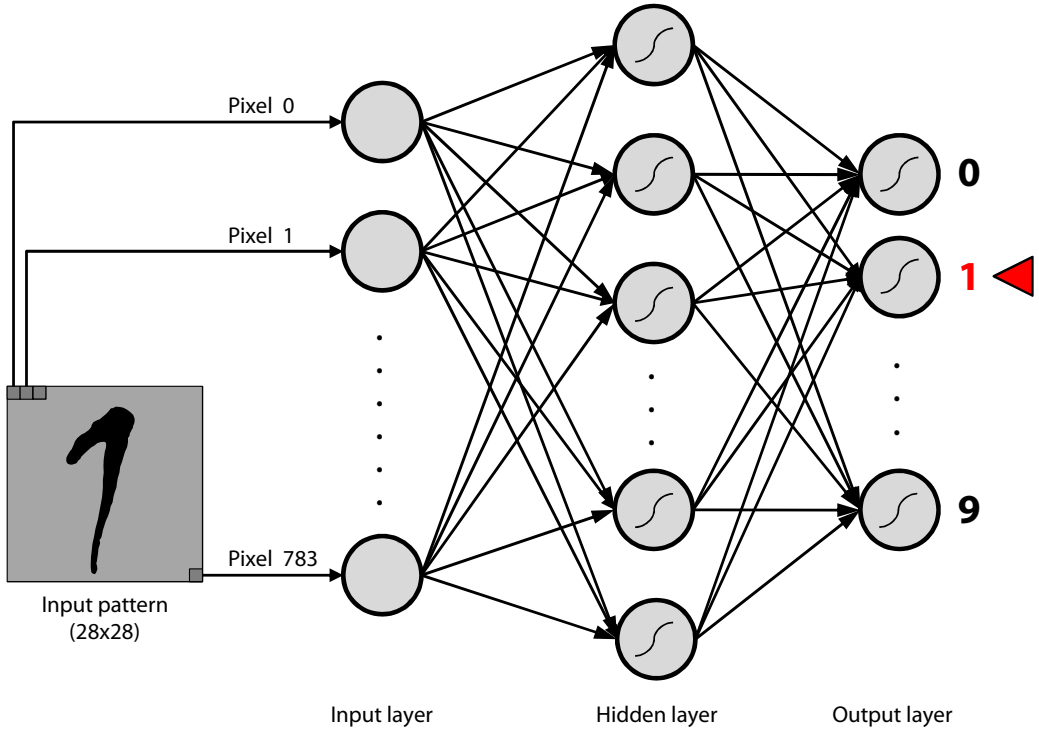
# The Goal

Our goal was to demonstrate the performance and scalability improvement that can be achieved by shifting the computation-intensive tasks of a convolutional neural network to the GPU using NVIDIAs CUDA architecture.

# A Review of Neural Networks

# Neural Networks (NNs)

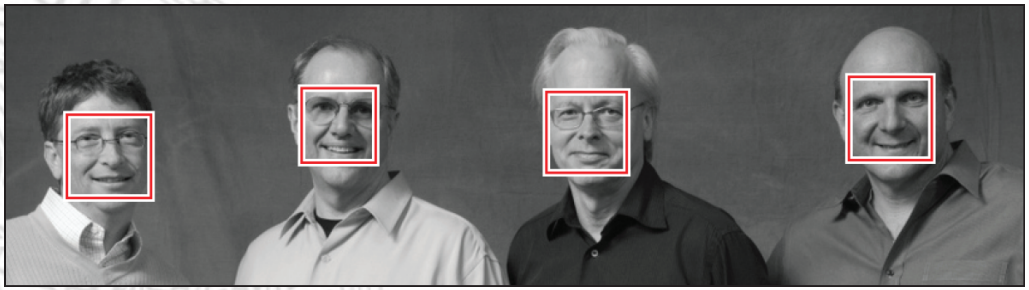
- Kind of machine learning algorithm which has to be trained.
- Are trained using a preclassified dataset.
- Often used for pattern recognition/classification tasks.
- Consist of many neurons which are organized in several layers.
- Have weighted connections between the single neurons.
- Training consists of forward- and backpropagation.

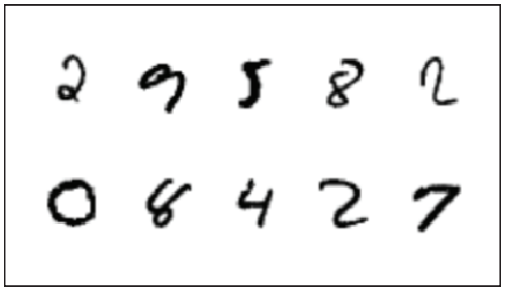


# Convolutional Neural Networks

## Convolutional Neural Networks (CNNs)

- Invented by Yann LeCun in the early 1990s [1].
- Special kind of neural networks, optimized for 2D problems.
- Can operate on raw images without separate feature extractor.
- Offer a certain invariance to shifting, scaling, and other forms of distortion.
- Are trained via the gradient decent method using error backpropagation.
- Successfully applied to handwriting recognition [2, 3], face detection [4], etc.

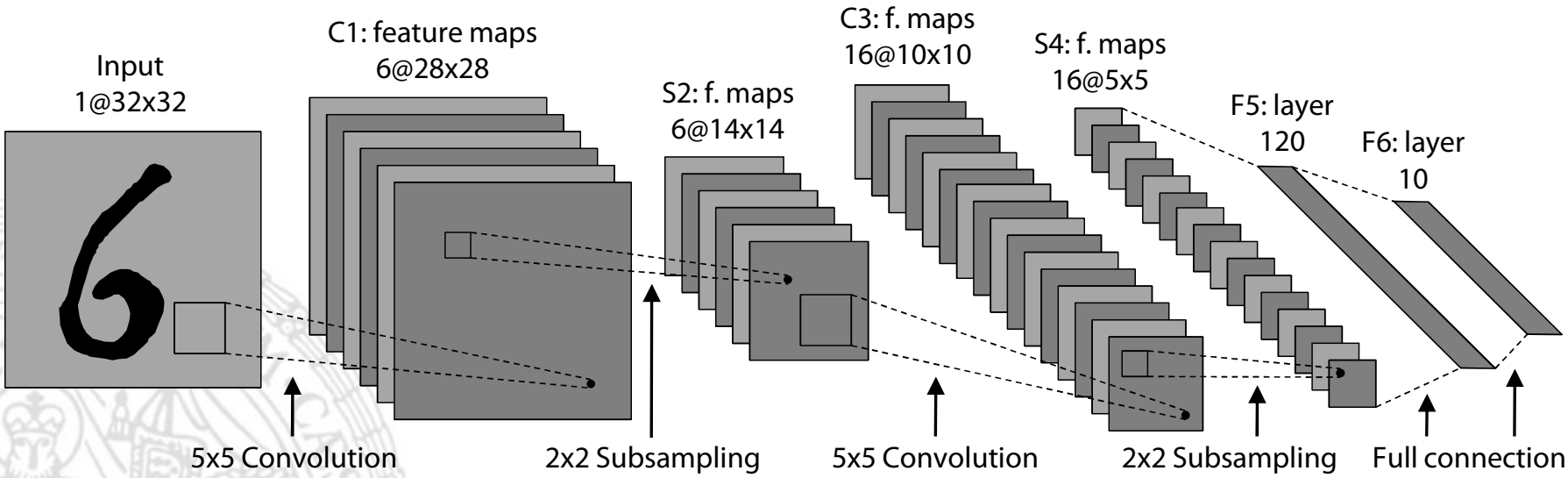




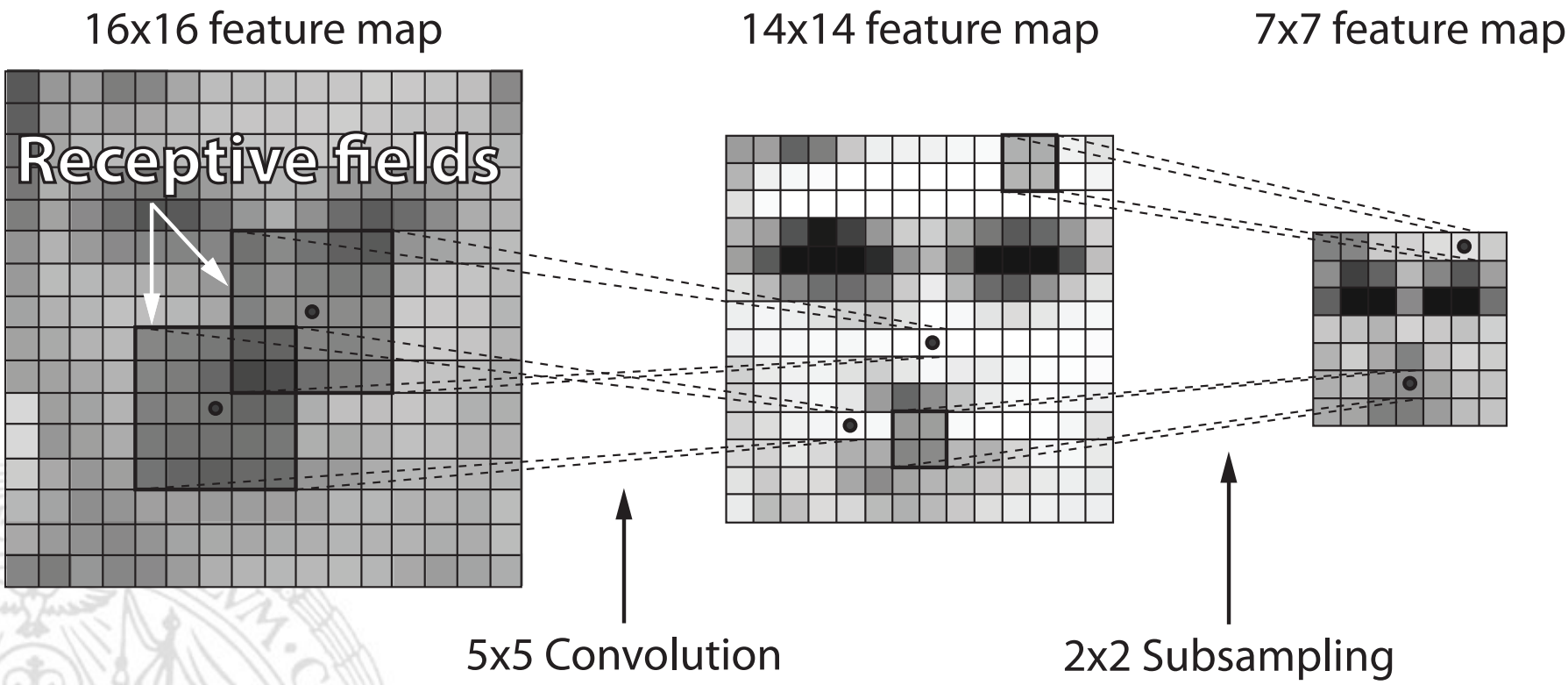
# CNN Topology

Feed-forward network, composed of three different types of layers:

- Convolution Layer (2D): feature extraction
- Subsampling Layer (2D): shift and distortion insensitivity
- Fully Connected Layer (1D): classification



# Convolution and Subsampling in Detail



## Drawbacks of CNNs

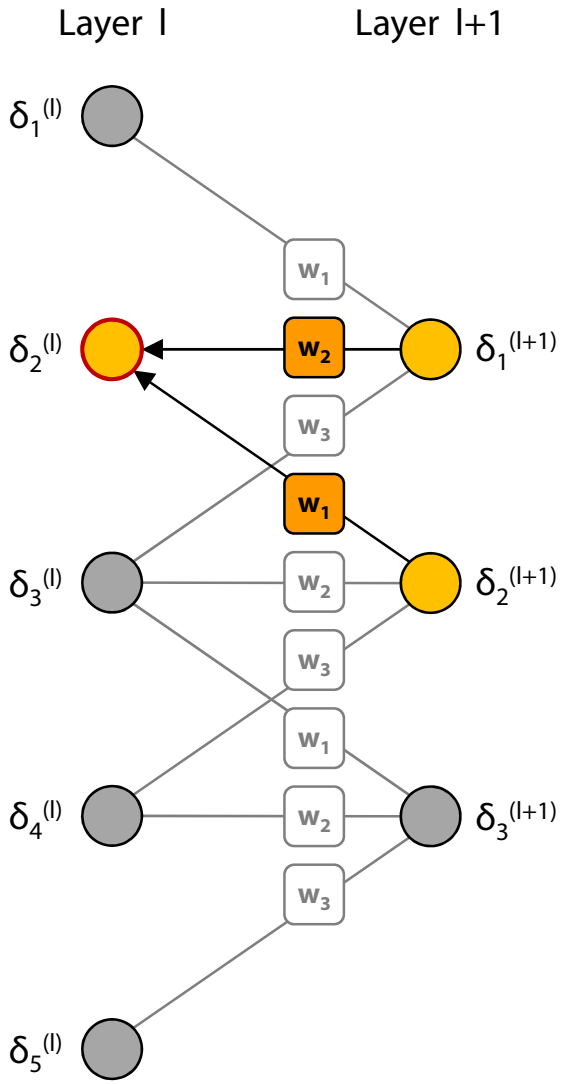
- Complex implementation
- Computational expensive
- Long training time
- Large number of configurable parameters which can only be determined by evaluating and testing:
  - Network topology
  - Number of feature maps/neurons inside the layers
  - Training parameters
  - ...

$\Longrightarrow$ High-performance library which shortens the training time using GPGPU computing and hides complexity from the developer.

# Implementation

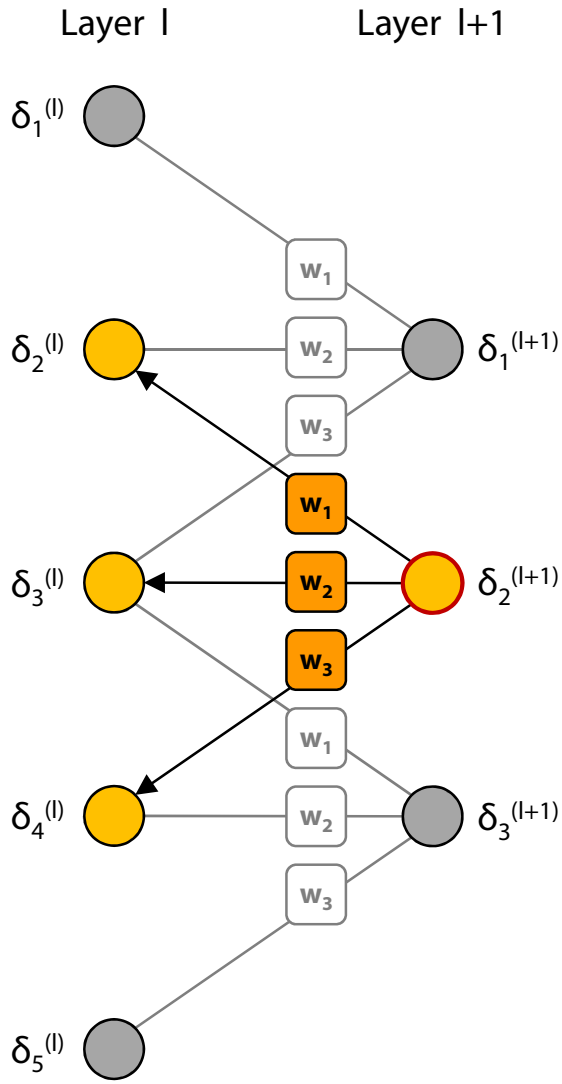
## Making CNNs simpler

- Traditional approach "pulls" the error back from the following layer during training phase.
- To calculate the error gradient each neuron needs the error of all neurons in the following layer $l + 1$ which are connected to it.
- This requires knowledge about the following layer $l + 1$.



## Making CNNs simpler – cont.

- Instead of pulling the error from the neurons in layer $l+1$ this layer could "push" the needed information back to layer $l$, providing one value for each neuron in it.
- Pushing the error back does not require any additional information about neighboring layers and results in a more regular structure.



## Making CNNs faster

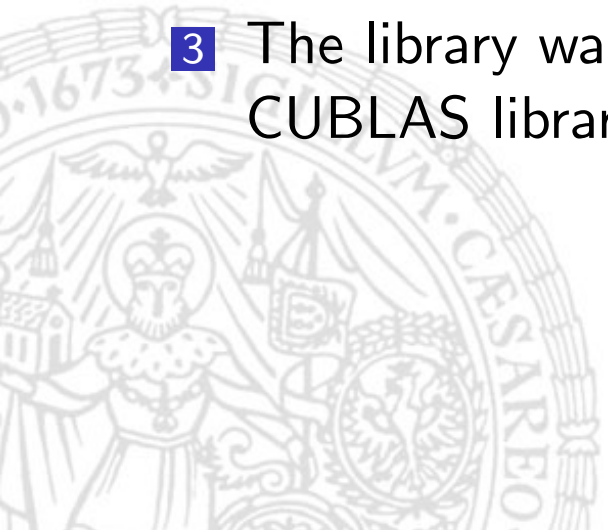
- Convolutions are not easy to optimize because of their irregular memory access pattern.
- To make the convolution more regular and easier to optimize one can "unfold" the convolution [5].

$\Longrightarrow$ Convolution can be implemented as a simple matrix-matrix product (see Appendix).

## Implementation Procedure

We implemented a high performance but still flexible library in C++ and CUDA to accelerate the training and classification process of arbitrary CNNs:

1. We started with a straight forward implementation without any manual parallelization or vectorization ($\mathrm{CPU}_{\mathrm{triv.}}$).
2. This implementation was optimized using functions from Intel's Performance Libraries IPP and MKL ($\mathrm{CPU}_{\mathrm{opt.}}$).
3. The library was ported to CUDA-enabled GPUs using the CUBLAS library and manually implemented functions (GPU).

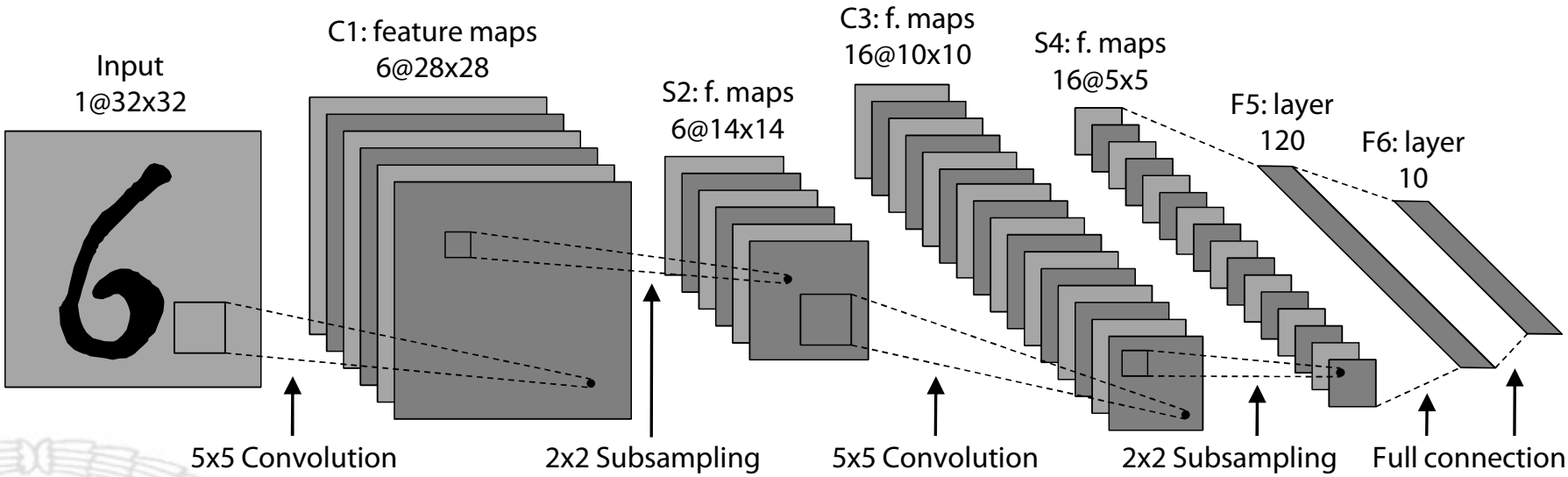


# Benchmarks and Results

**All benchmarks in our work were performed**

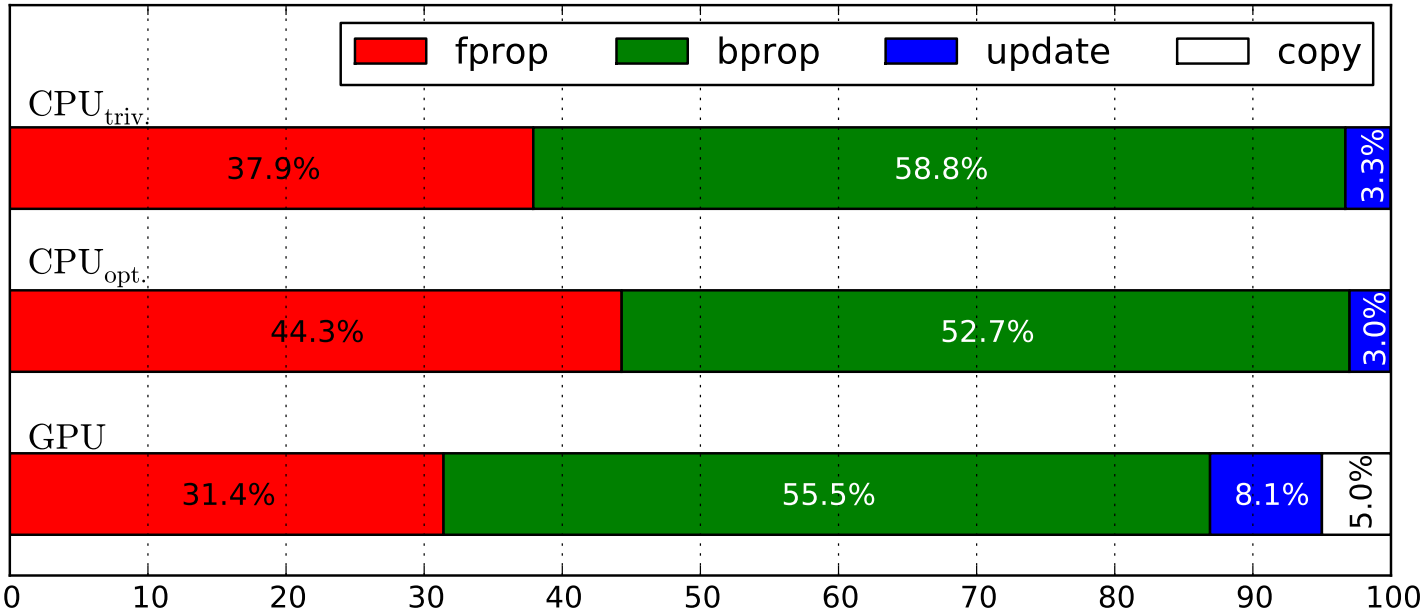
- on a PC equipped with an Intel Core i7 860 and a NVIDIA GeForce GTX 275,
- using the GNU C++ compiler for the CPU code and the NVIDIA NVCC compiler to generate the GPU code,
- in floating-point single precision (32-bit),
- measuring the execution time of 1,000 (online) training iterations using patterns form the MNIST database [6].

# Benchmarks on the LeNet5*




* Proposed by Yann LeCun in [3].

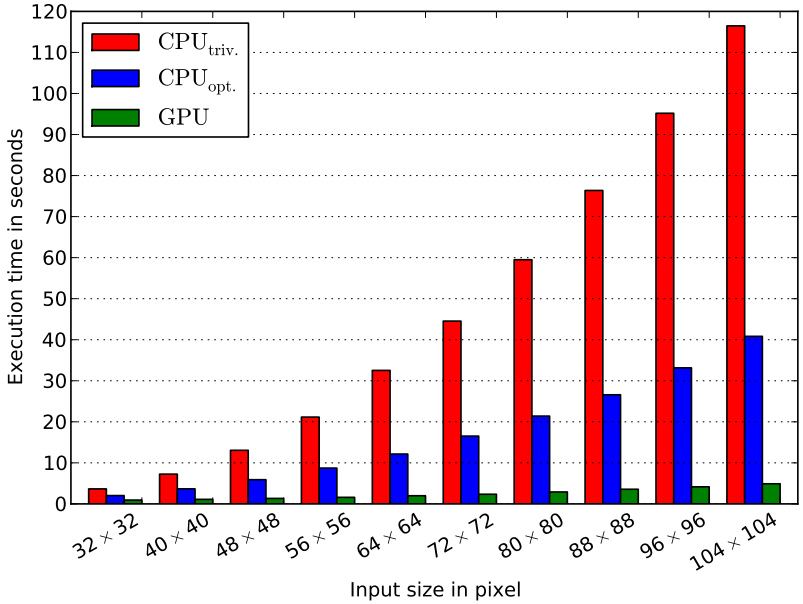
# Composition of Execution Time



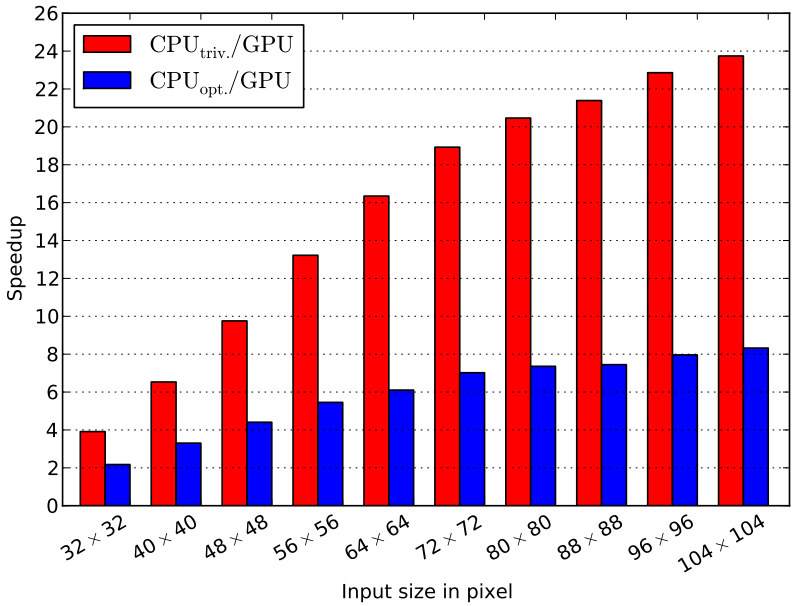

| | |
|---|---|
| **fprop** | Forward propagation of a pattern through the network. |
| **bprop** | Backward propagation of the error to obtain the gradient. |
| **update** | Adaption of the weights according to the gradient descent method. |
| **copy** | Transferring data from the CPU to the GPU and vice versa. |

# Scaling the Input Size of a LeNet5

## Execution Time




## Speedup

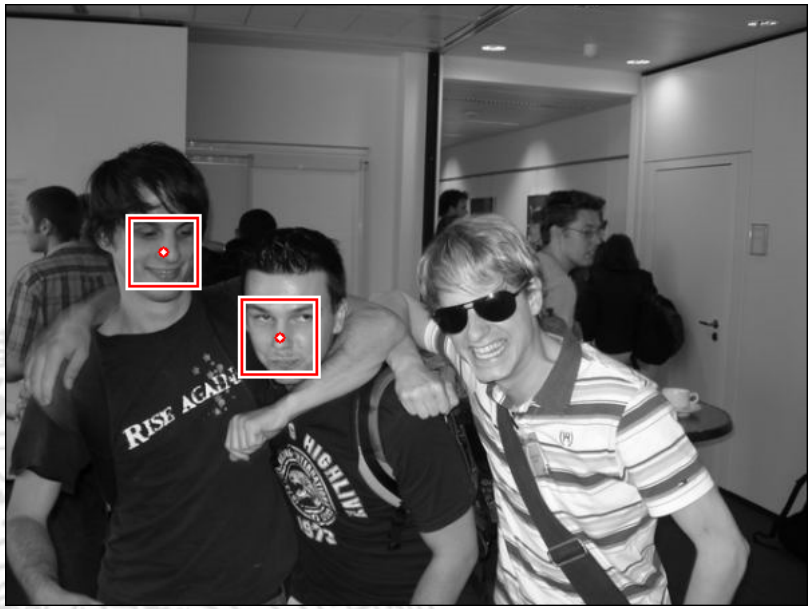


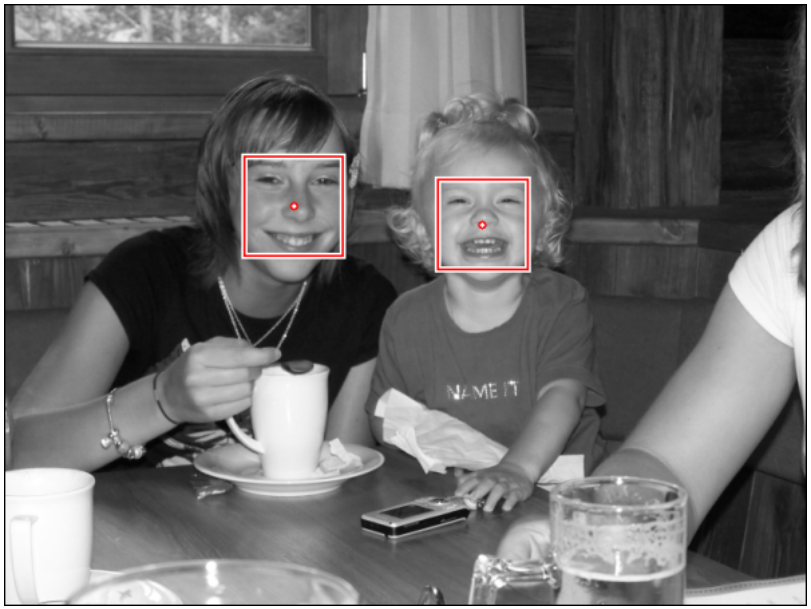

Execution time and corresponding speedup of the three different implementations performing 1,000 learning iterations of a LeNet5.

# Applications of CNNs

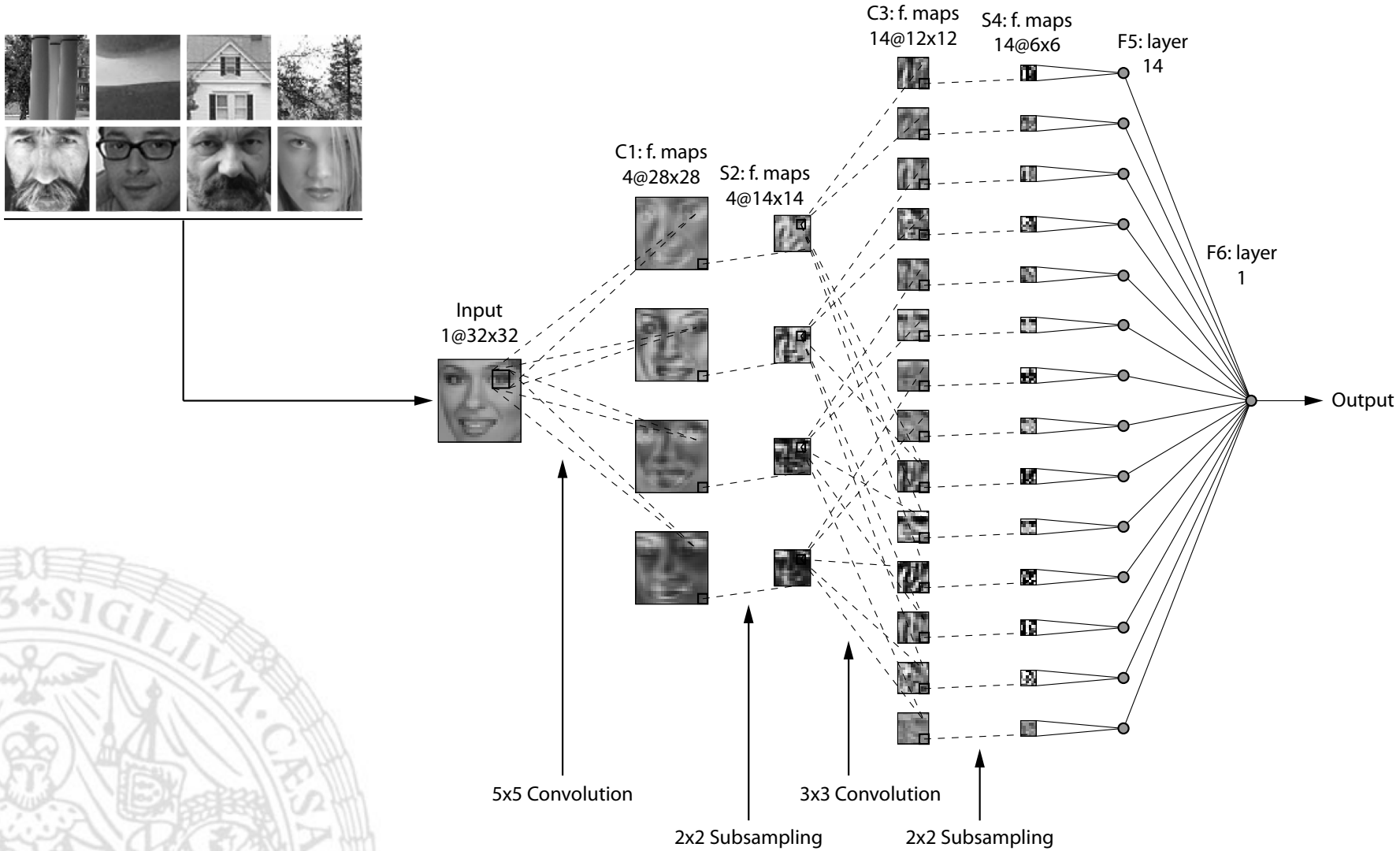
## Convolutional Face Detector

*Determine whether or not there are any faces in an image and, if present, return the image location and region of each face.*

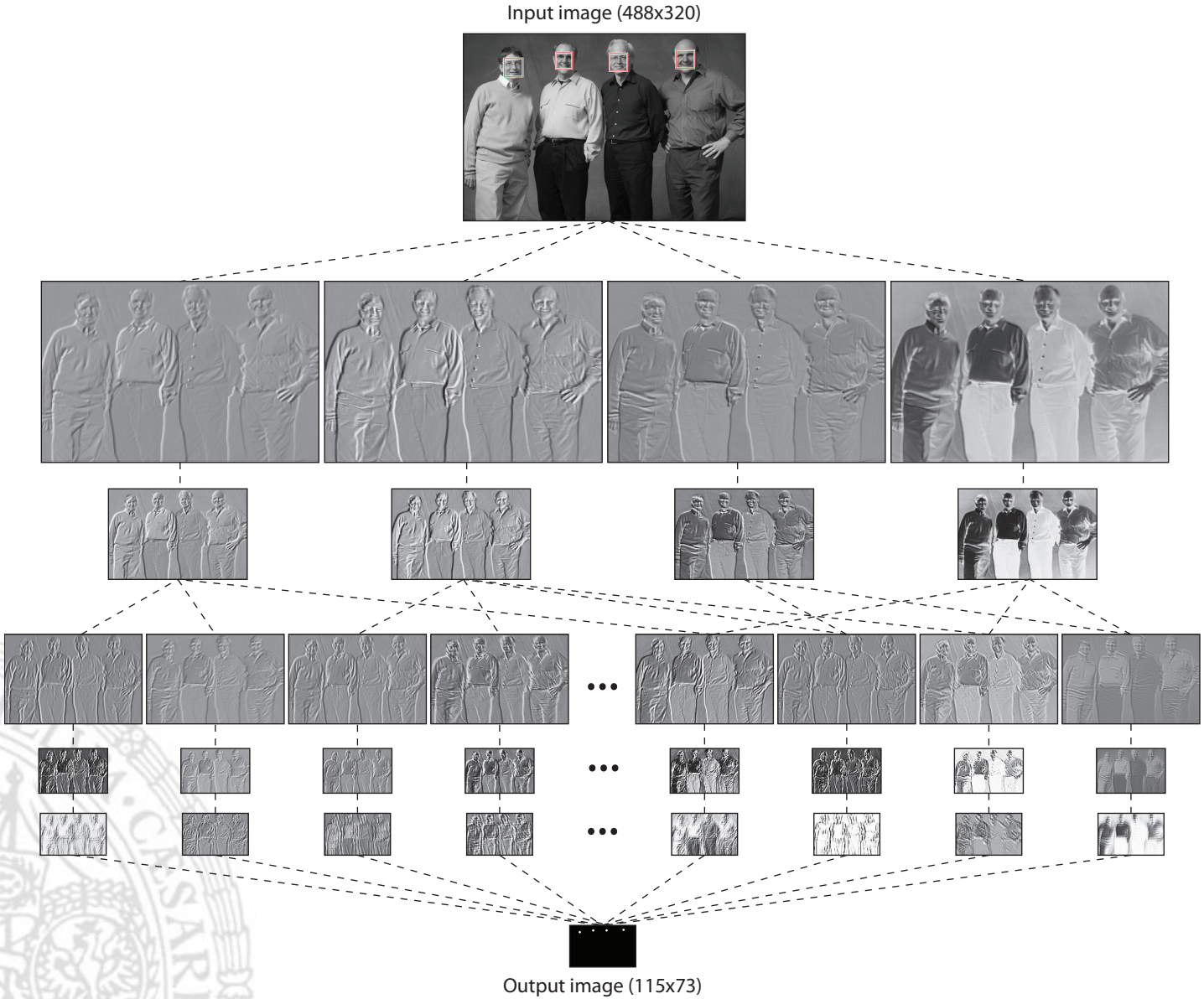


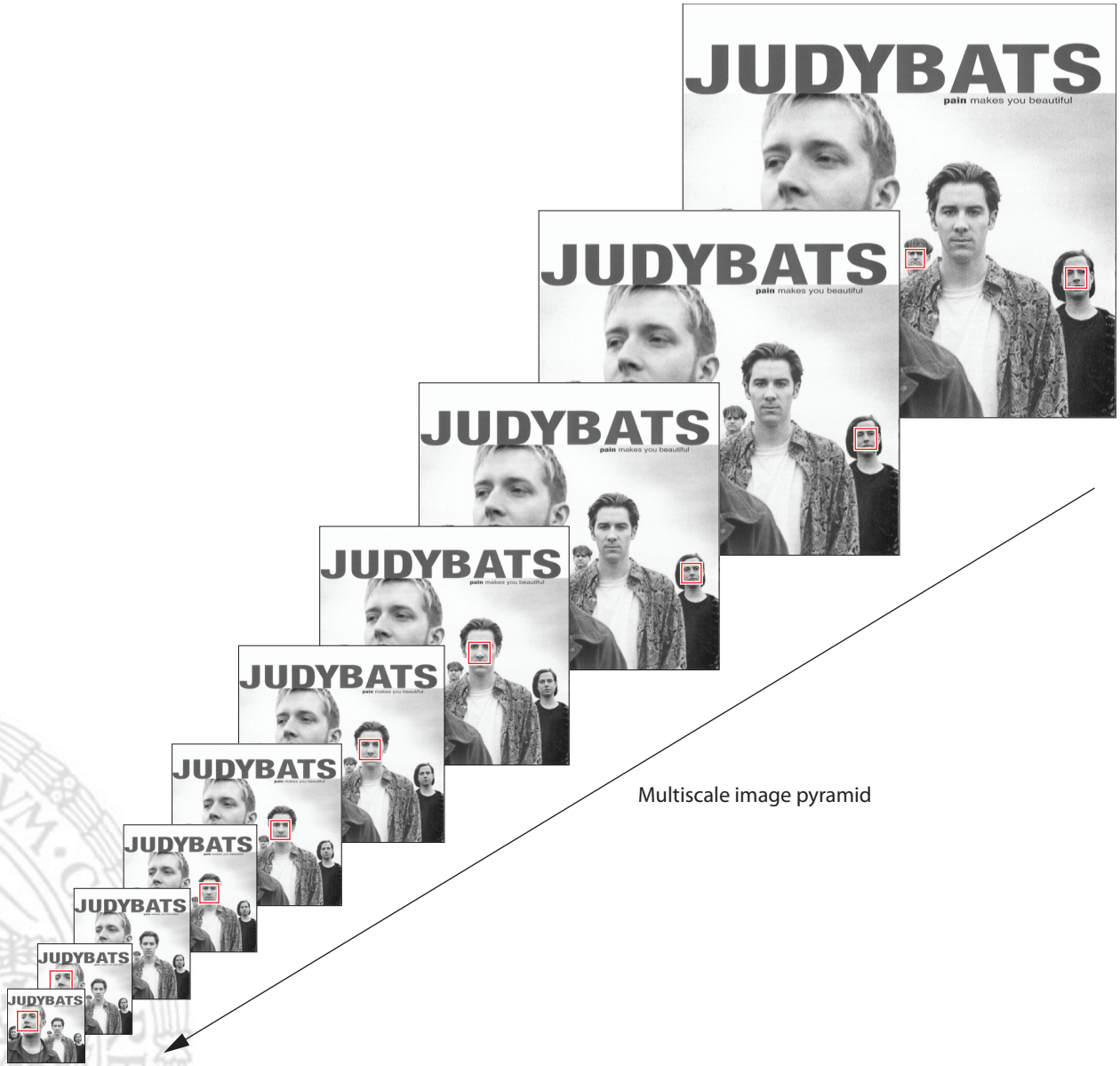
# Network Architecture [4]



# Face Localization



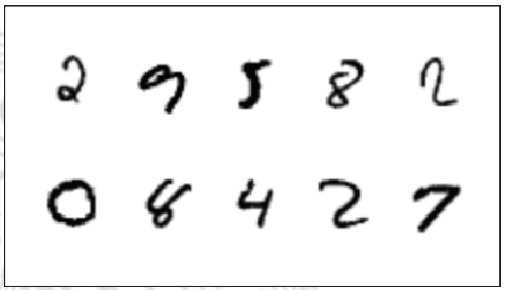
# Face Localization – cont.




2 face(s)
1 [1.00]
2 [1.0
LIVE ORF 1
Land der Hämmer, zukunftsreich

## Optical Character Recognition

- The MNIST database was used for evaluation.
- The first 54,000 patterns of the training set were used to train the network.
- The last 6,000 patterns of the training set were used as validation set for the early stopping algorithm.
- The test set, containing 10,000 patterns, was used to determine the network's classification performance.

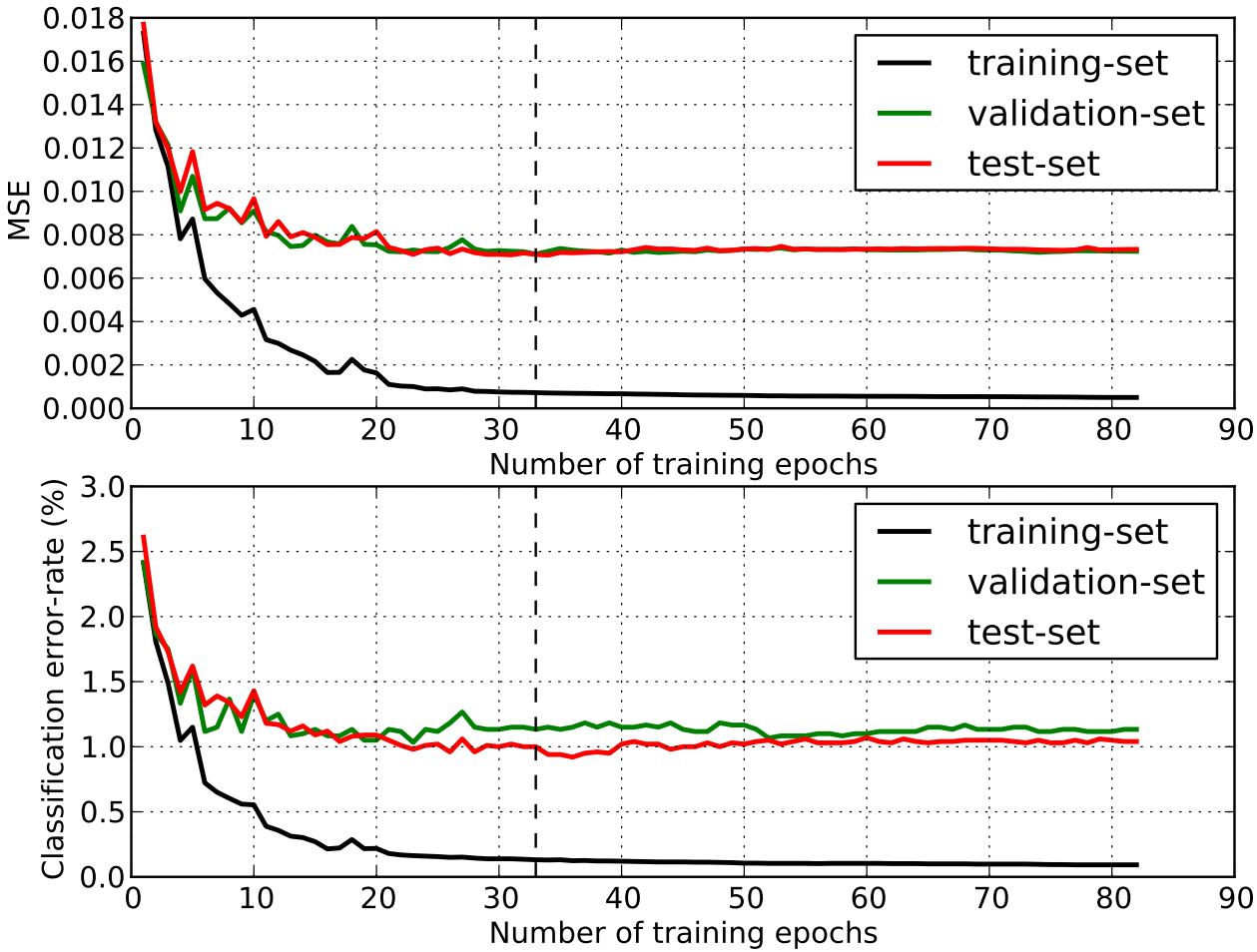


**MNIST Handwritten Digit Database**

- Provided by Yann LeCun and Corinna Cortes.
- Images of handwritten digits $(0, 1, \ldots, 9)$.
- Each of size $28 \times 28$ pixels with 256 gray levels.
- Contains 60,000 training and 10,000 test patterns.

# LeNet5 Classification Performance

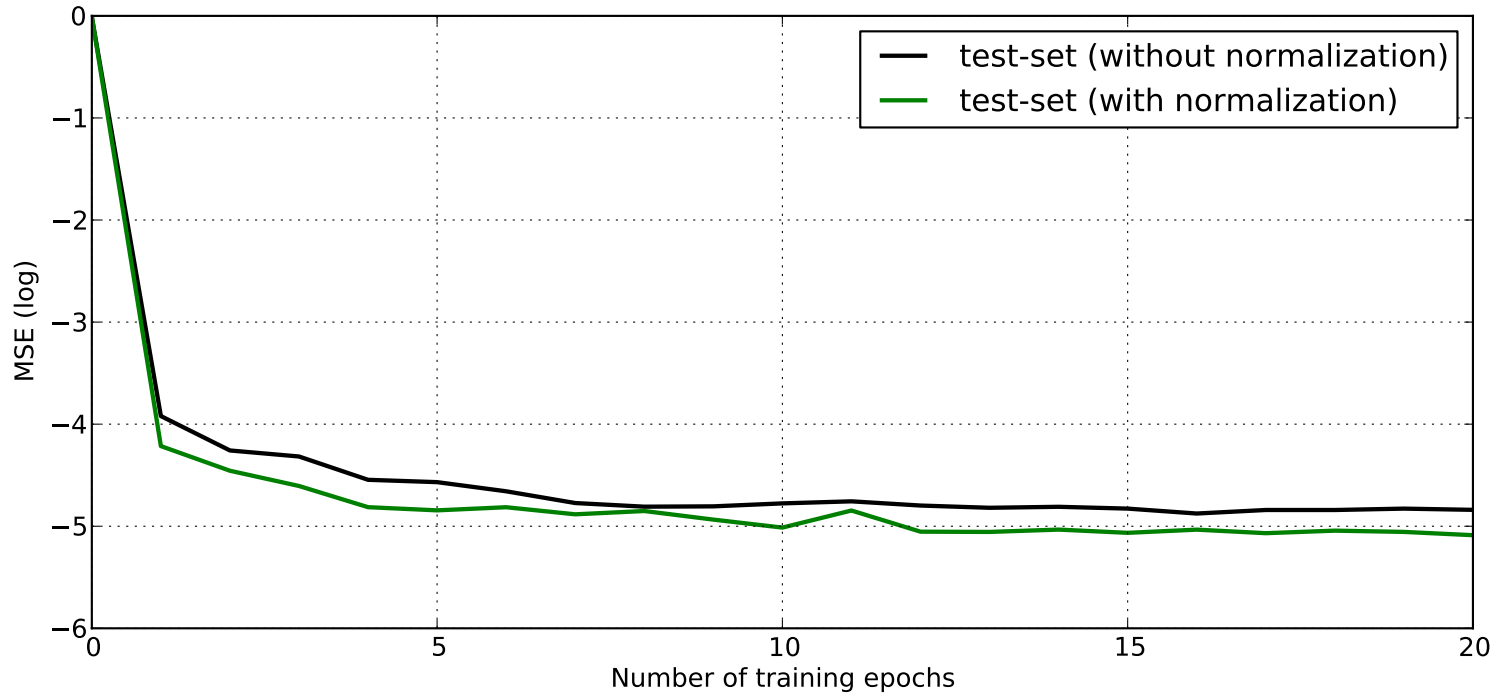
Avg. Classification Rate: 98,91%




Development of the error on all three datasets during the learning of the MNIST database by a LeNet5. The dashed line indicates the epoch with the best performance on the validation set.

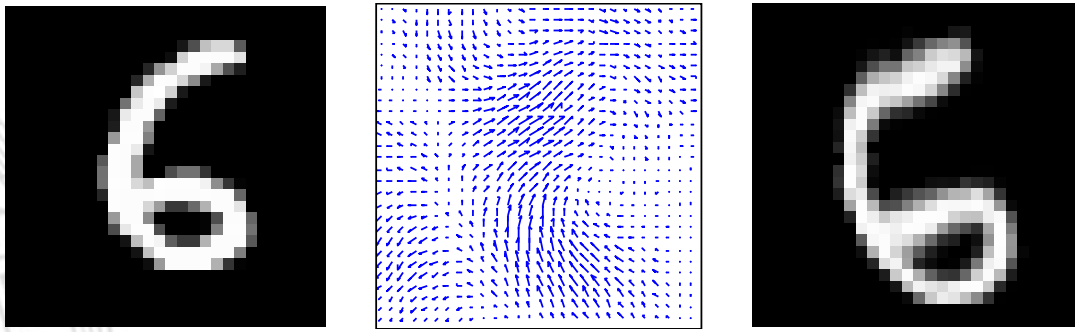
# Enhancing Classification Result by Normalization

- The input data is preprocessed so that the mean of the whole dataset is 0 and its standard deviation is 1.
- To achieve this we calculate the mean and the standard deviation of the whole training dataset.
- Before a pattern is fed into the network the mean is subtracted from it and the result is divided by the standard deviation.
- Classification rate improves by approximately 0.05 - 0.1%.



## Enlarge Training Dataset by Elastic Distortions

- More training data often improves the accuracy of a neural network.
- Therefore we enlarge the training dataset using elastic distortions:
  - For every existing training pattern we generate 9 virtual ones using 9 different displacement fields.
  - The newly generated patters are used for training along with the label of their original pattern.
  - Those displacement fields are regenerated in every training epoch.
  - Classification rate improves by approximately 0.3 - 0.6%.



# Conclusion and Future Work

## Conclusion

- GPUs work quite well for convolutional neural networks.
- The GPU implementation scales much better than the CPU implementations with increasing network size.
- The shorter training time allows more network evaluations which often lead to better results.
- The modifacitons applied to the training data and network improved the network's accuracy.

## Future Work

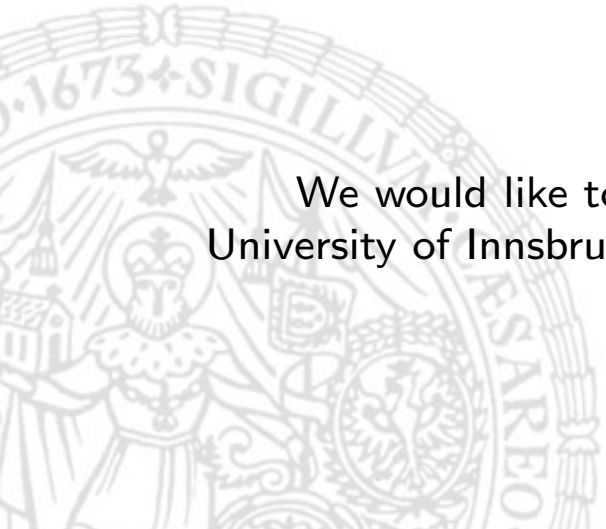
- Accomplish complex experimental tests with different network topologies and parameter settings using our fast GPU implementation.
- Analyze and compare the power consumption of our CPU implementation to our GPU versions.

# Thanks for your attention!

Questions?

According to the statements from Sonnenburg et al. in [7] which emphasis
the need for open source software in machine learning
we make our library [8] public available:

http://cnnlib.sourceforge.net (coming soon)



**Acknowledgments**

We would like to thank Dr. Alfred Strey for his helpful comments and the University of Innsbruck who supported this work with the "Förderungsstipendium".

# Bibliography

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel.
Handwritten Digit Recognition with a Back-Propagation Network.
In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2 (NIPS 1989)*, pages 396–404, Denver, Colorado, USA, 1990. Morgan Kaufman.

P. Y. Simard, D. Steinkraus, and J. C. Platt.
Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis.
In *Proceedings of the 7th International Conference on Document Analysis and Recognition (ICDAR 2003)*, pages 958–962, Edinburgh, Scotland, August 2003.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner.
Gradient-Based Learning Applied to Document Recognition.
*Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

C. Garcia and M. Delakis.
Convolutional Face Finder: A Neural Architecture for Fast and Robust Face Detection.
*IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(11):1408–1423, November 2004.

K. Chellapilla, S. Puri, and P. Y. Simard.
High Performance Convolutional Neural Networks for Document Processing.
In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition (IWFHR 2006)*, La Baule, France, October 2006.

Y. LeCun and C. Cortes.
MNIST Handwritten Digit Database.
http://yann.lecun.com/exdb/mnist, August 2009.

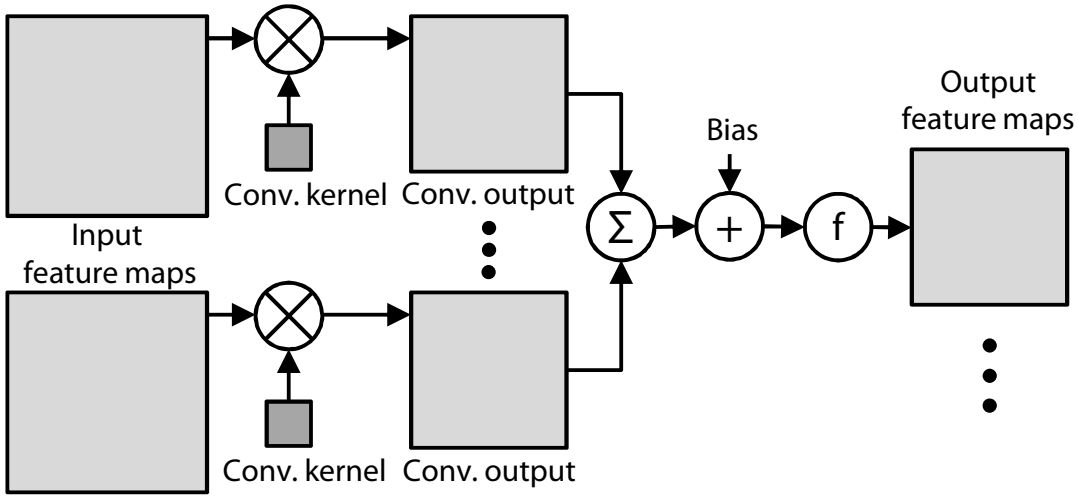
S. Sonnenburg, M. L. Braun, C. S. Ong, S. Bengio, L. Bottou, G. Holmes, Y. LeCun, K.-R. Müller, F. Pereira, C. E. Rasmussen, G. Rätsch, B. Schölkopf, A. Smola, P. Vincent, J. Weston, and R. Williamson.
The Need for Open Source Software in Machine Learning.
*The Journal of Machine Learning Research*, 8:1532–4435, December 2007.
Available at: http://jmlr.csail.mit.edu/papers/volume8/sonnenburg07a/sonnenburg07a.pdf.

D. Strigl and K. Kofler.
CNNLIB: A library for fast convolutional neural network training and classification.
http://cnnlib.sourceforge.net, October 2009.

# Appendix

# Illustration of the operating principle of a convolutional and subsampling layer inside a CNN:

## Convolutional Layer




## Subsampling Layer

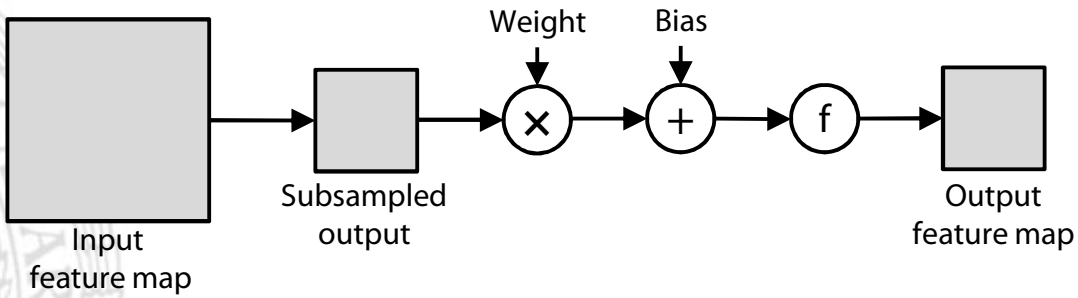
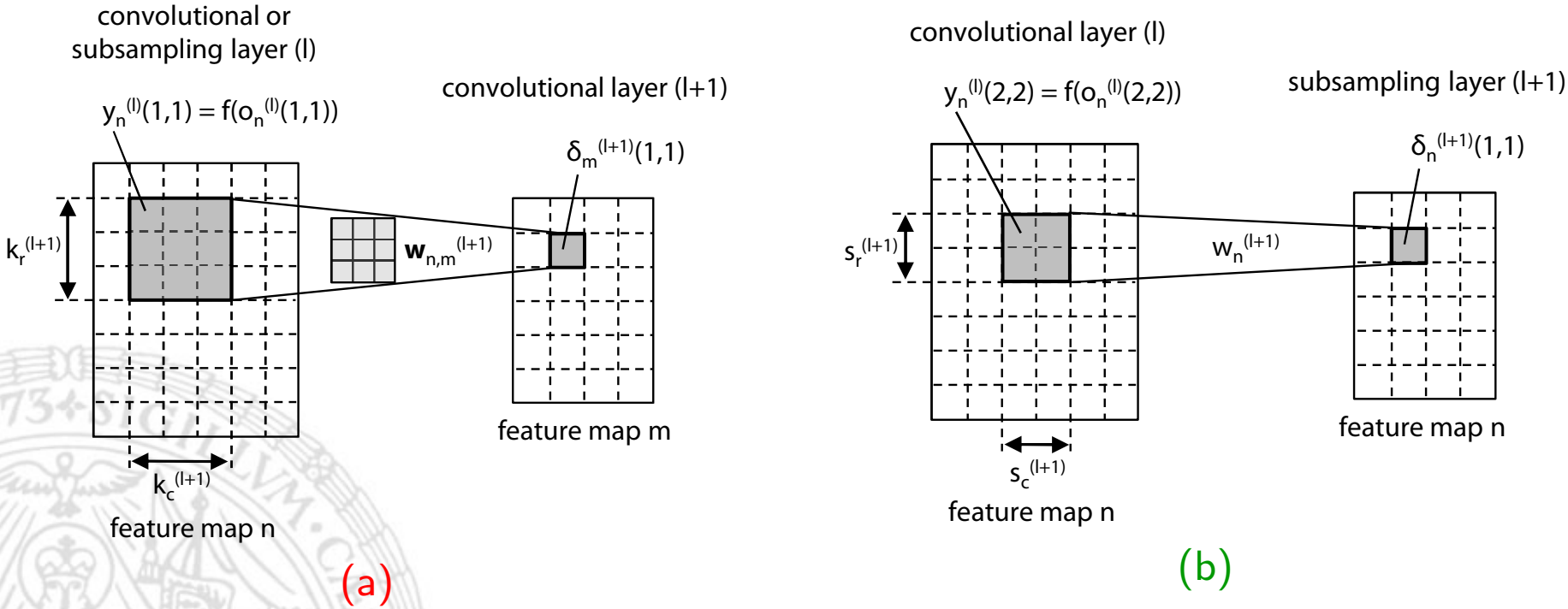
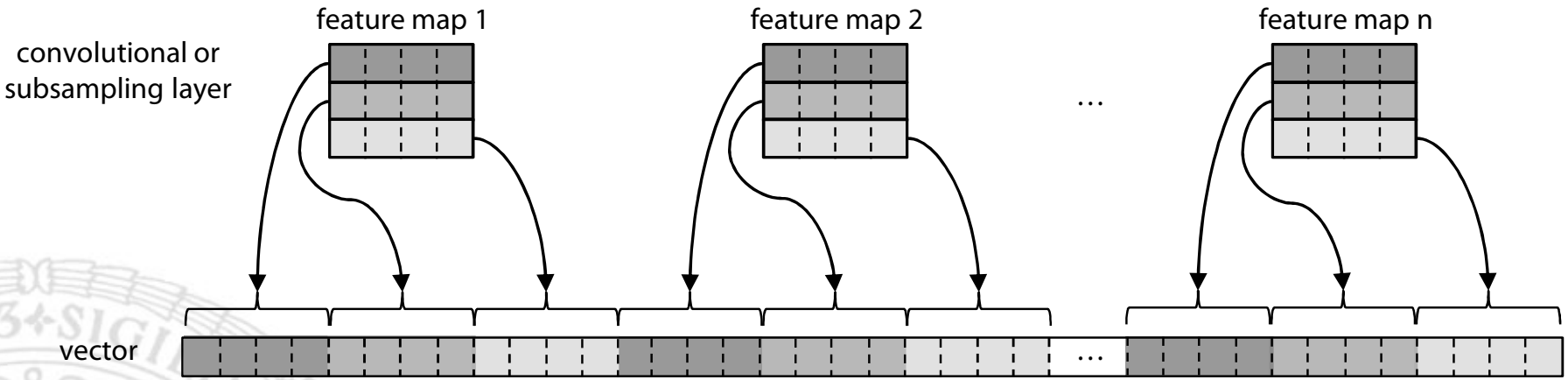


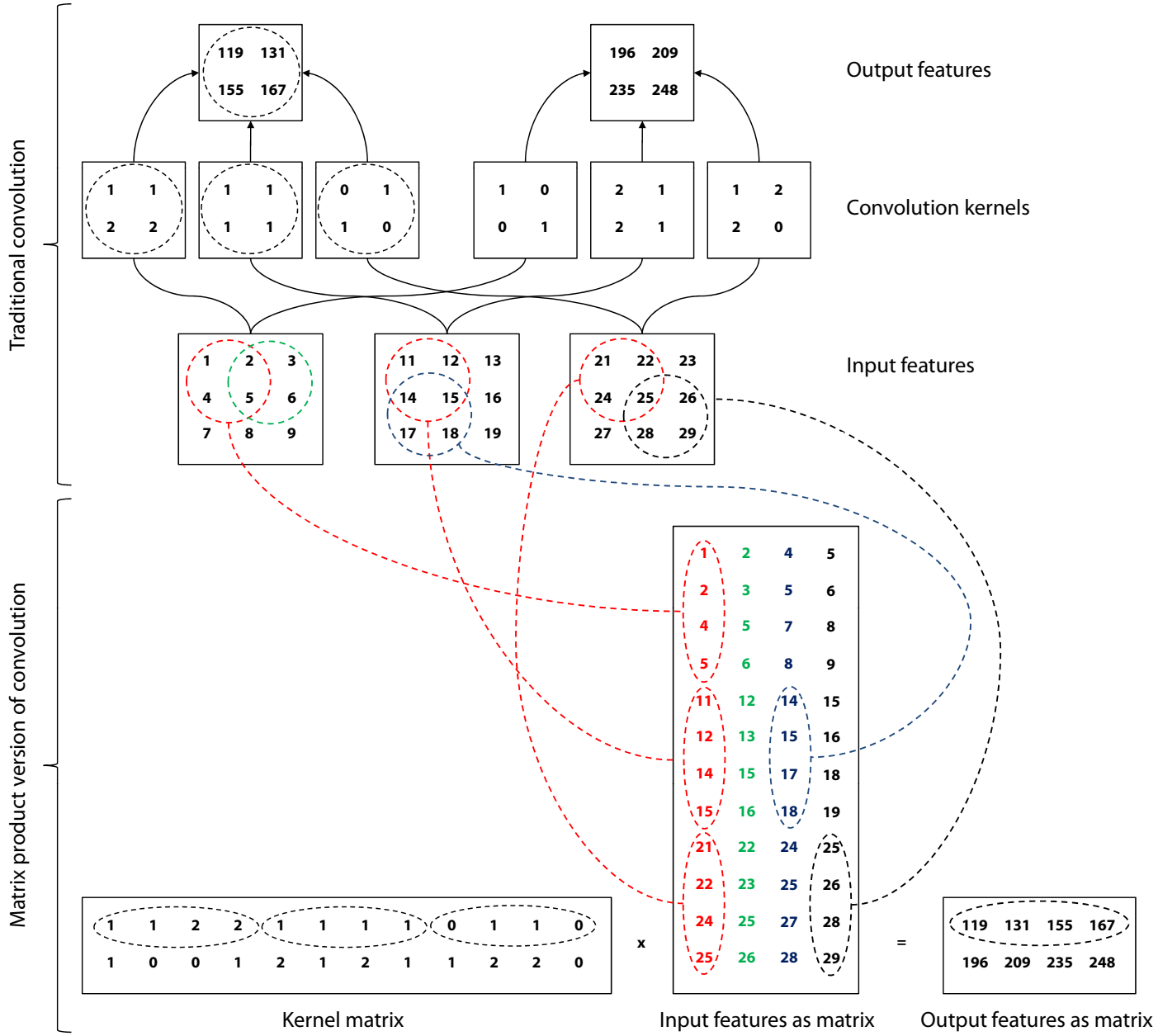
**Illustration of the error backpropagation for a convolutional layer followed by a subsampling layer (a) and a convolutional or subsampling layer followed by a convolutional layer (b):**



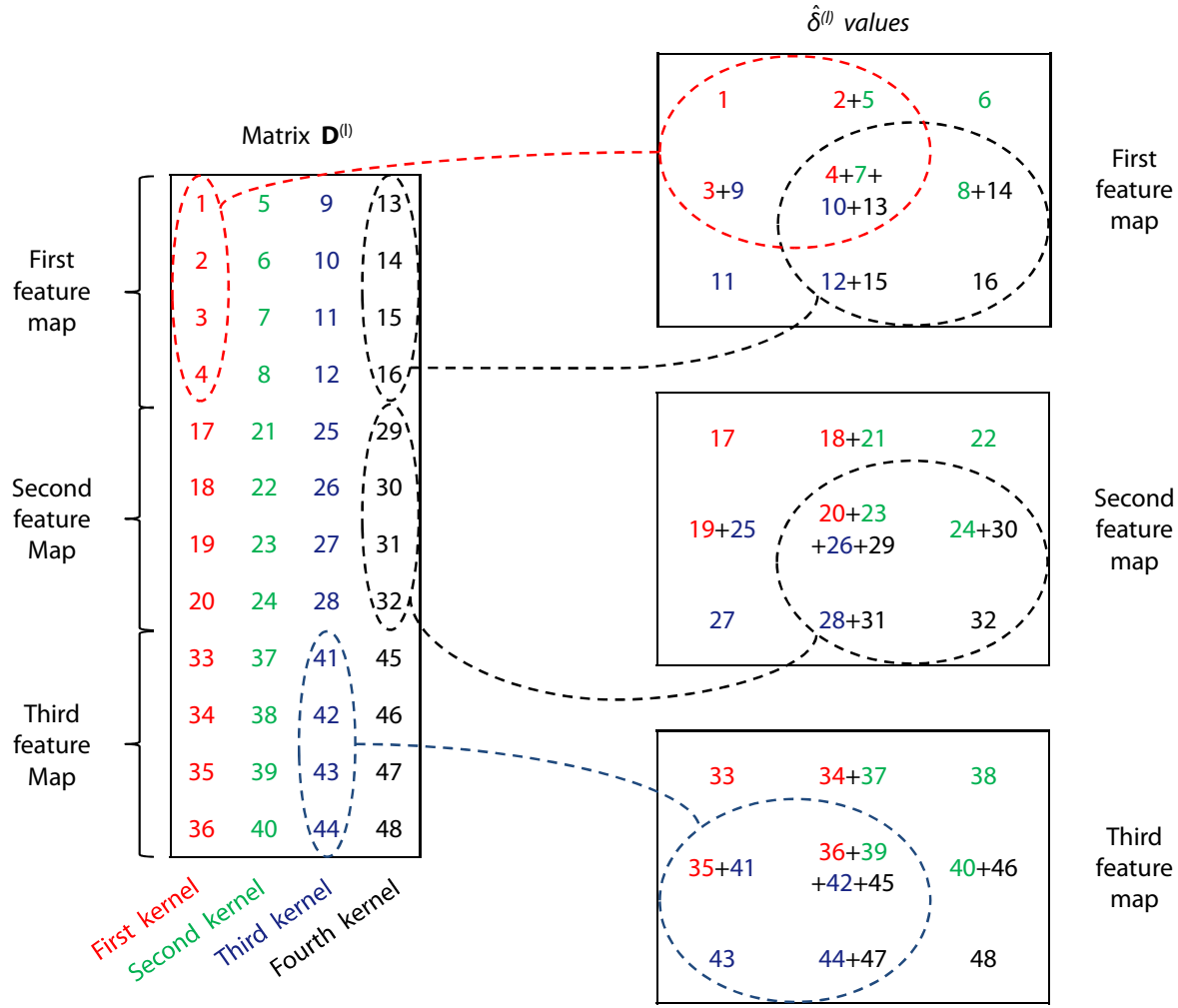
**Rearrangement of the feature maps of a convolutional or subsampling layer into a single vector, so that it can be used in the succeeding (1D) fully connected layer:**



# Unfolding the Convolution



# Backfolding



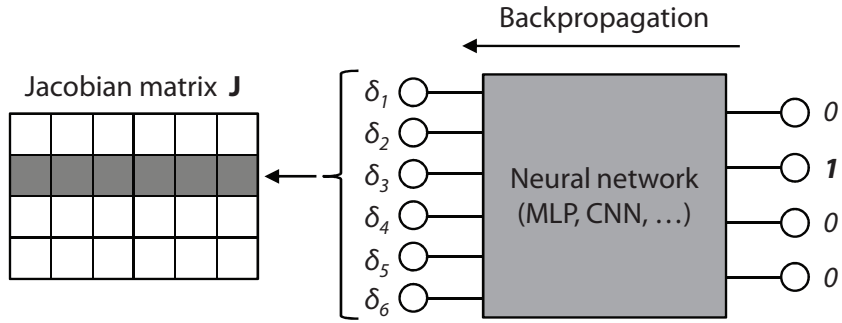
## Numerical Differentiation

$$\frac{\partial E}{\partial w^{(l)}(i,j)} = \frac{E_{\left[w^{(l)}(i,j)+\epsilon\right]} - E_{\left[w^{(l)}(i,j)-\epsilon\right]}}{2 \cdot \epsilon} \tag{1}$$

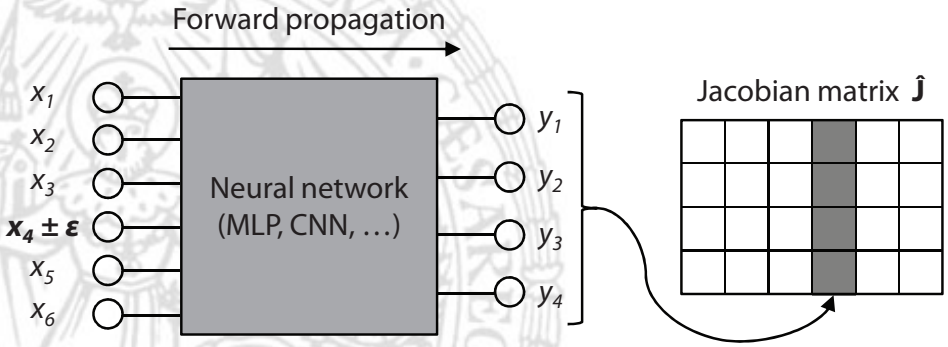
# Jacobian Matrix

$$J(i,j) = \frac{\partial y^{(L)}(j)}{\partial x(i)} \tag{2}$$

Constructing the Jacobian matrix **J** using the backpopagation function:



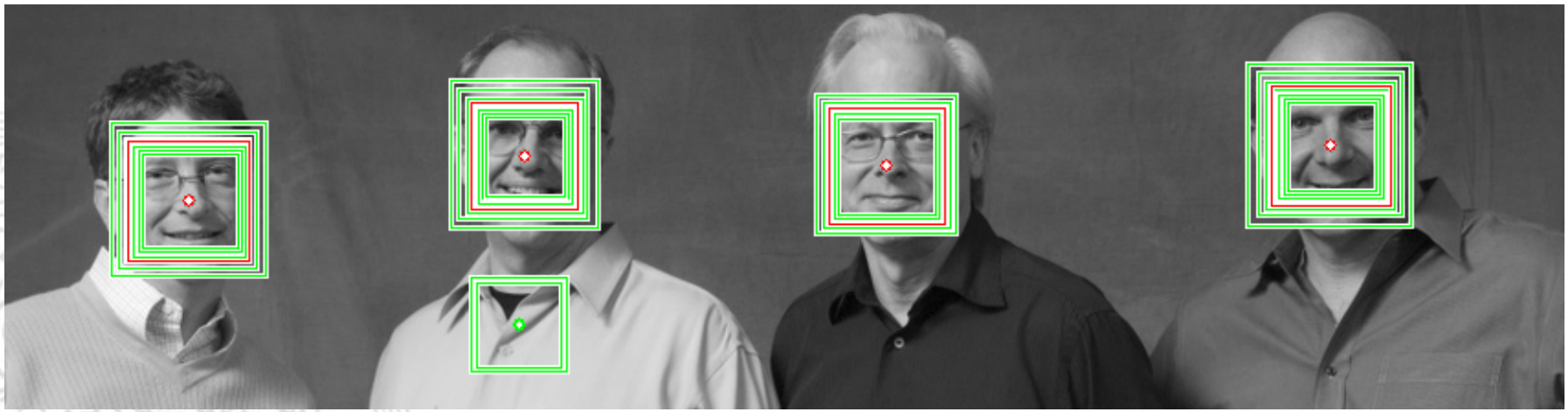

Constructing an approximation $\hat{\mathbf{J}}$ of the Jacobian matrix using the forward propagation function:



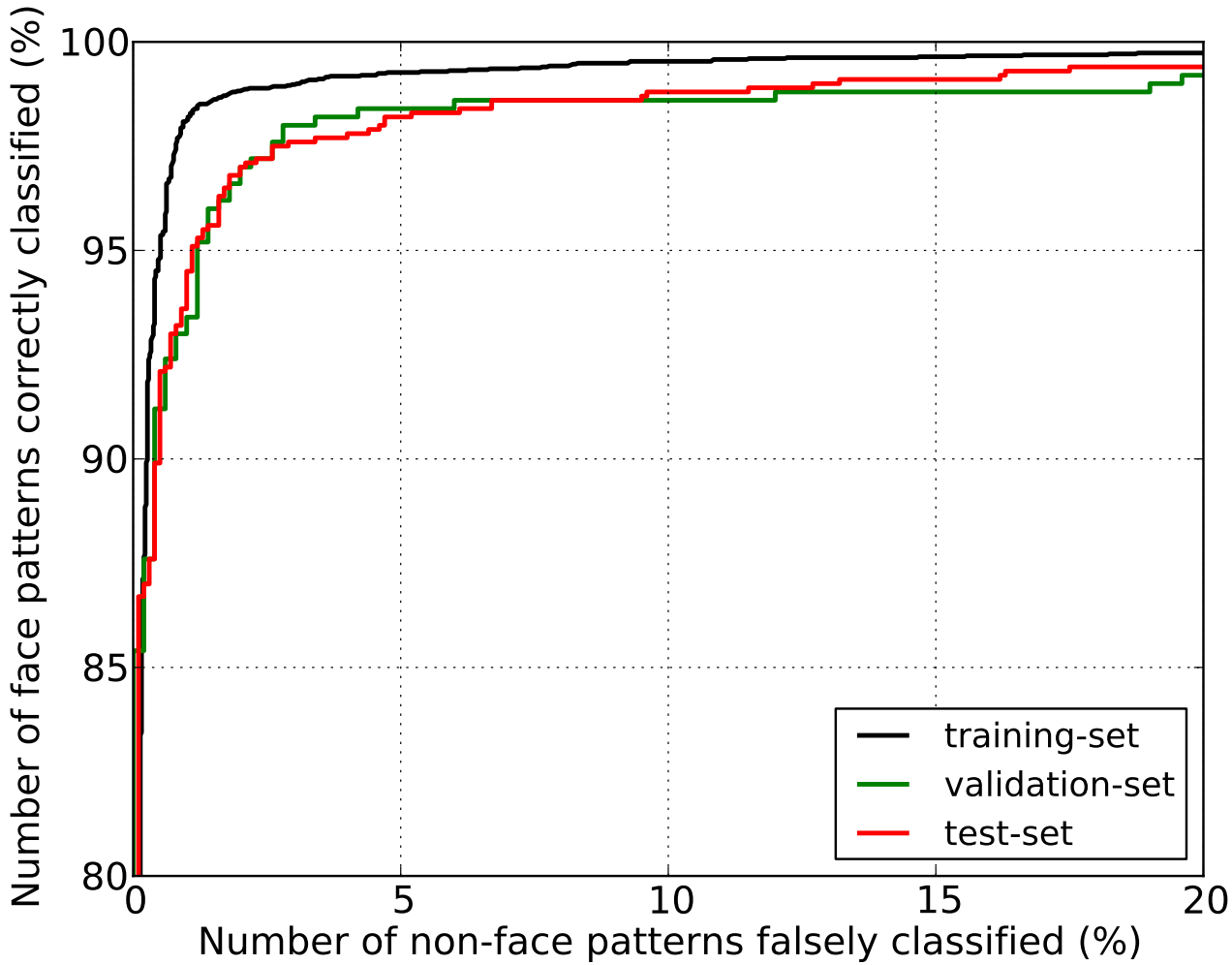

$$\hat{J}(i,j) = \frac{y^{(L)}_{[x(i)+\epsilon]}(j) - y^{(L)}_{[x(i)-\epsilon]}(j)}{2 \cdot \epsilon} \tag{3}$$

**Additional verification step to reduce the number of false detections:**

1. Each face candidate is tested at six scales in the original input image, ranging from 0.7 to 1.3 of the detected size.
2. At each scale the presence of a face is evaluated and compared to a threshold.
3. True faces will give high network responses in a certain number of consecutive scales, while non-faces don't.



**ROC curves of the trained GarciaNet applied to the training, validation, and test dataset:**



## Input Data Normalization

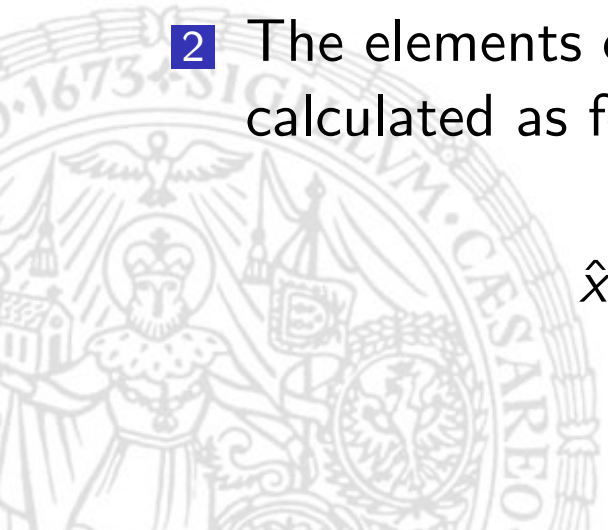
1. First of all the mean $m$ of the entire training set $S$ and its standard derivation $\sigma$ have to be calculated as follows:

$$m = \frac{1}{|S|} \cdot \sum_{x \in S} \frac{1}{|x|} \cdot \sum_{i=0}^{|x|-1} x(i) \tag{4}$$

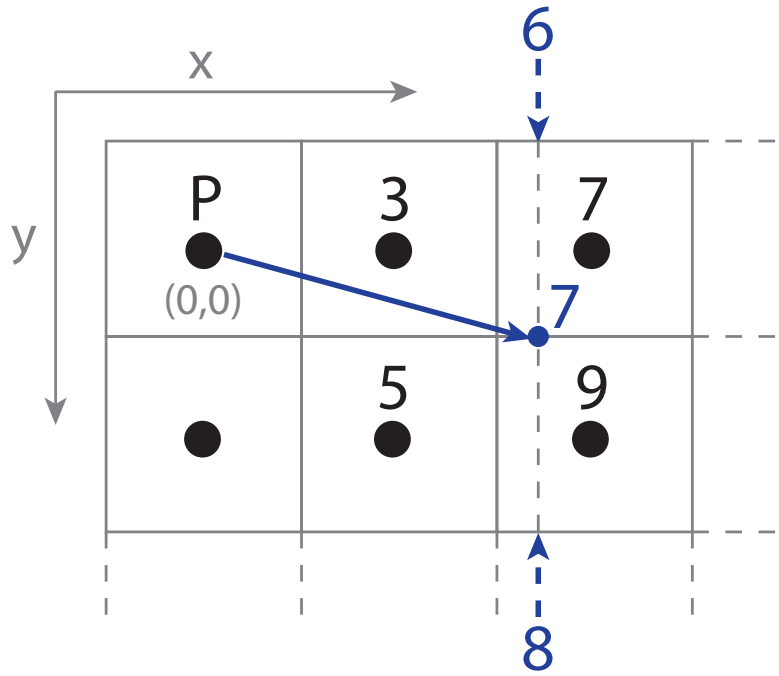
$$\sigma = \sqrt{\frac{1}{|S|} \cdot \sum_{x \in S} \frac{1}{|x|} \cdot \sum_{i=0}^{|x|-1} (x(i) - m)^2} \tag{5}$$

2. The elements of the normalized input vector $\hat{\mathbf{x}}$ can then be calculated as follows:

$$\hat{x}(i) = \frac{x(i) - m}{\sigma}, \quad i = 0, 1, \ldots, |x| - 1 \tag{6}$$

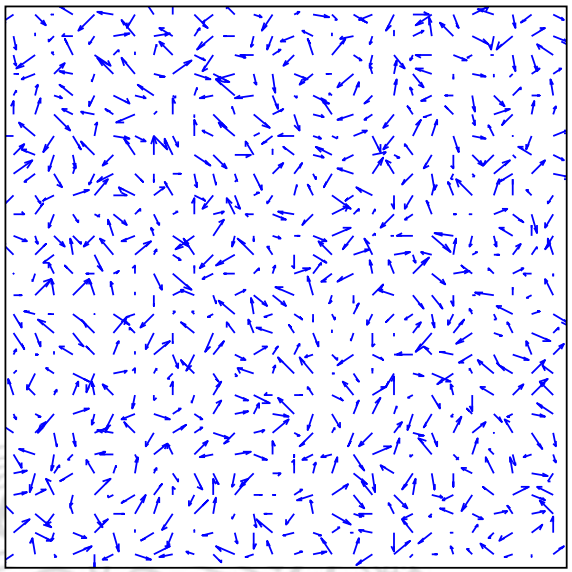


## Bilinear Interpolation for $\Delta x = 1.75$ and $\Delta y = 0.5$:

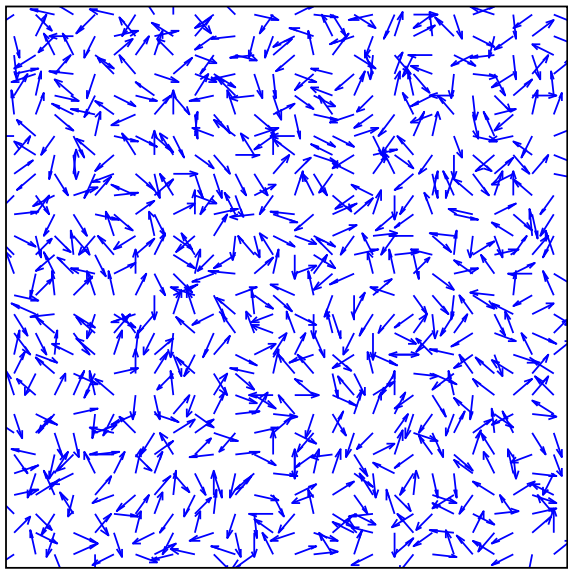



$$P_{x_1} = 3 + 0.75 \cdot (7 - 3) = 6$$
$$P_{x_2} = 5 + 0.75 \cdot (9 - 5) = 8$$
$$P = 6 + 0.5 \cdot (8 - 6) = 7 \tag{7}$$

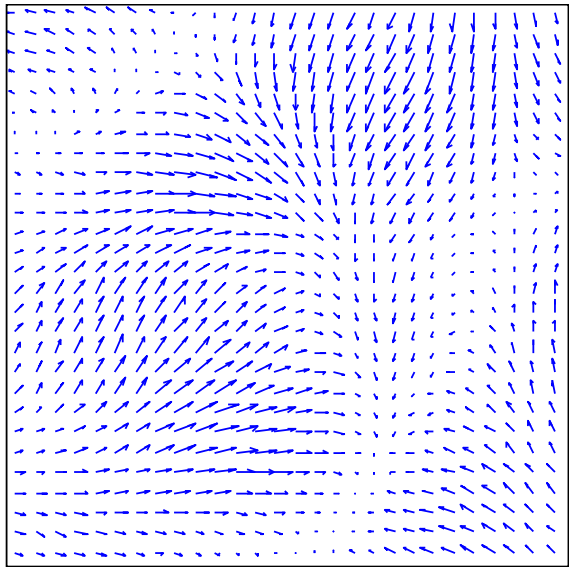
**Illustration of the single steps to generate a displacement field for elastic deformations:**



after initialization

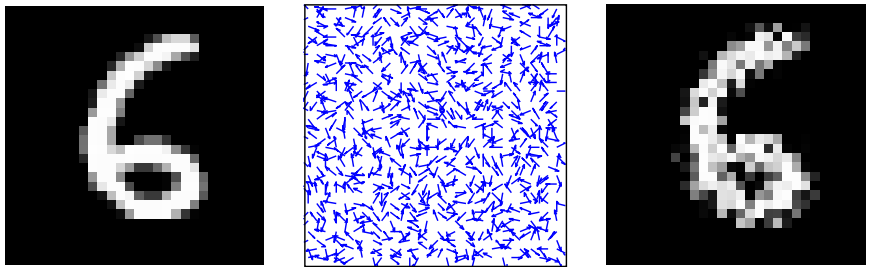


after normalization

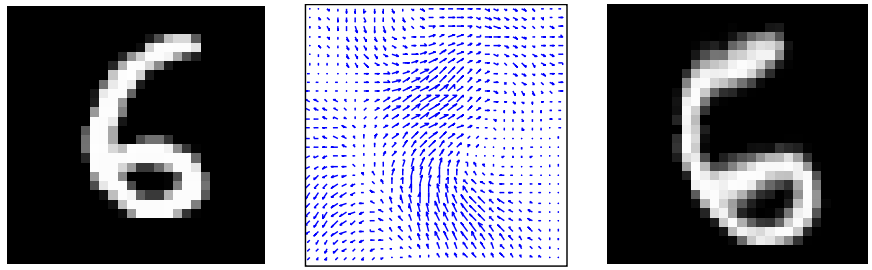


after Gaussian ($\sigma = 4$)

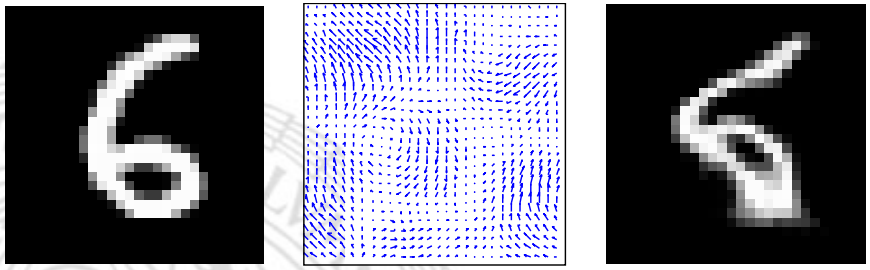
## Effects of $\sigma$ and $\alpha$:



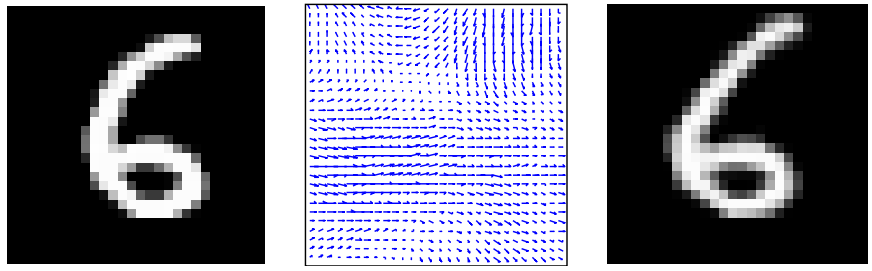
$\sigma = 0.1$ and $\alpha = 1$



$\sigma = 4$ and $\alpha = 34$



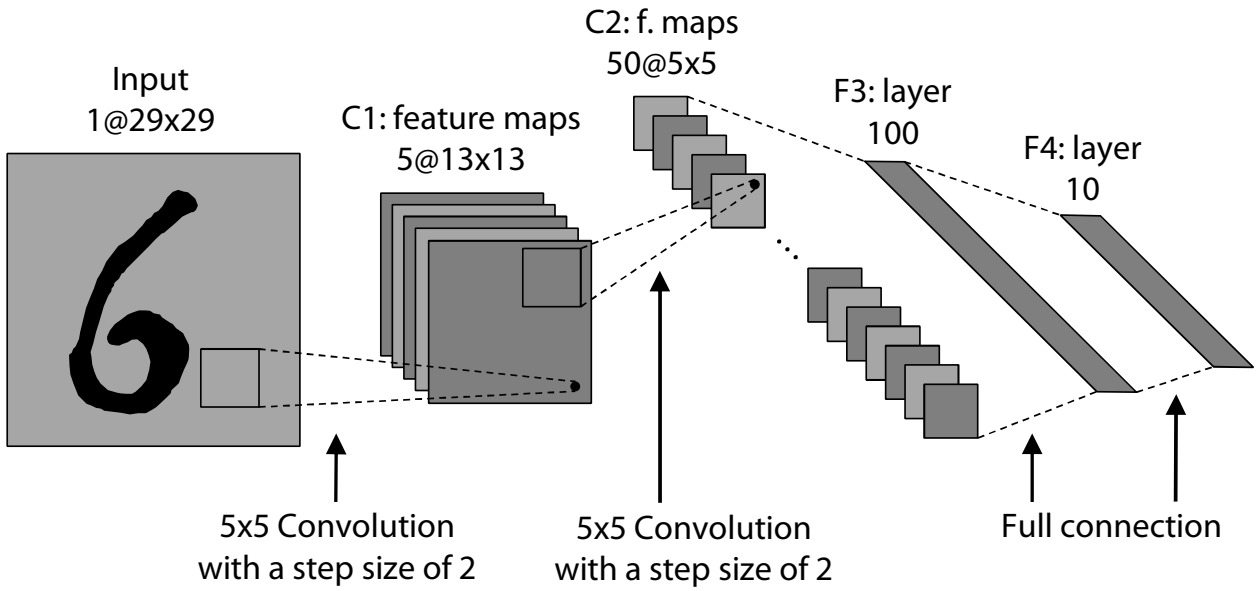
$\sigma = 3$ and $\alpha = 30$



$\sigma = 6$ and $\alpha = 22$

| **Properties** | **LeNet5** | **JahrerNet** |
|---|---|---|
| Structure | 5 Layer (C1 S2 C3 S4 F5 F6) | 5 Layer (C1 S2 C3 S4 C5 F6) |
| Connection S2 to C3 | Sparse, predefined | Random, probability 0.14 |
| Connection S4 to C5/F5 | Full | Random, probability 0.20 |
| # neurons | 8,010 | 21,160 |
| # connections | 51,046 | $\sim$ 100,000 |
| # trainable parameters | 331,114 | $\sim$ 770,000 |
| Rel. execution time (CPU) | 1 | $\sim$ 2.95 |
| Rel. execution time (GPU) | 1 | $\sim$ 1.06 |

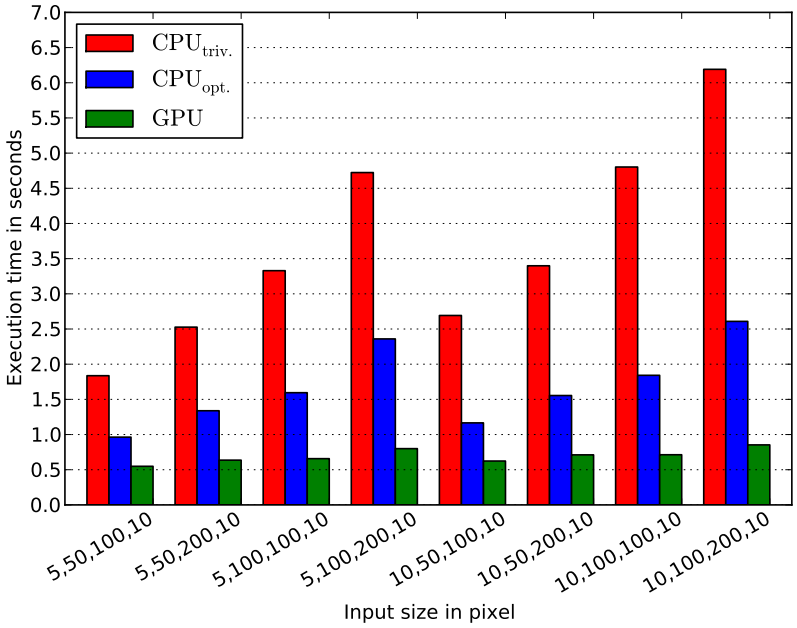
# Benchmarks on the SimardNet*



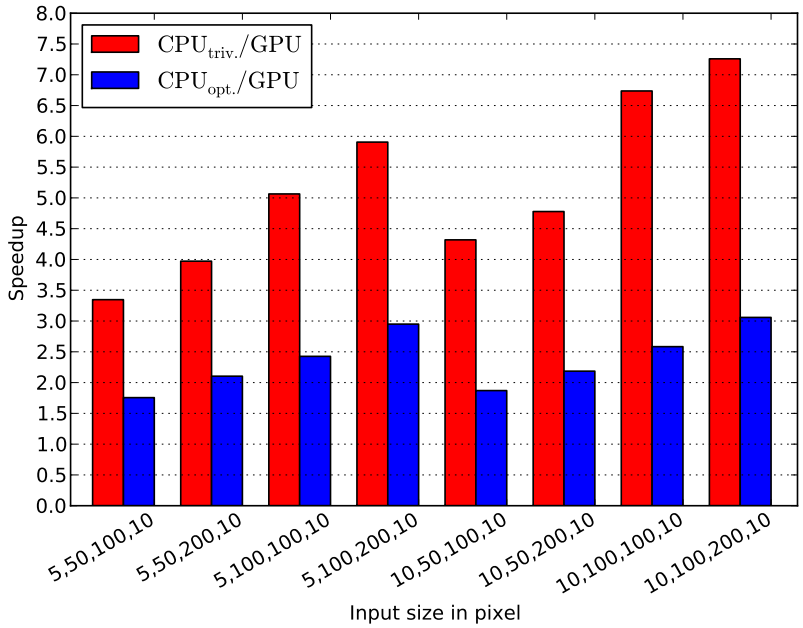

* Proposed by Simard et al. in [2] (one of the best performing networks on the MNIST database).

# Scaling Feature Maps and Inner Neurons of a SimardNet

## Execution Time

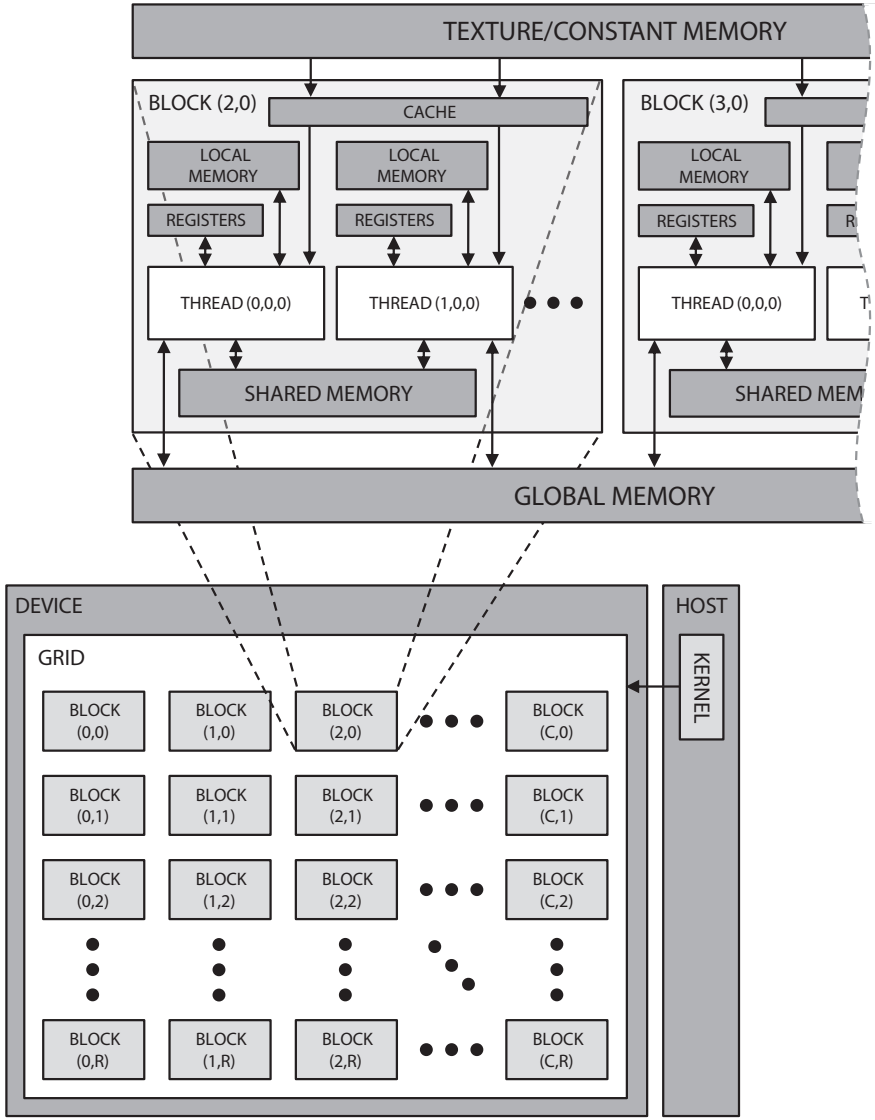



## Speedup

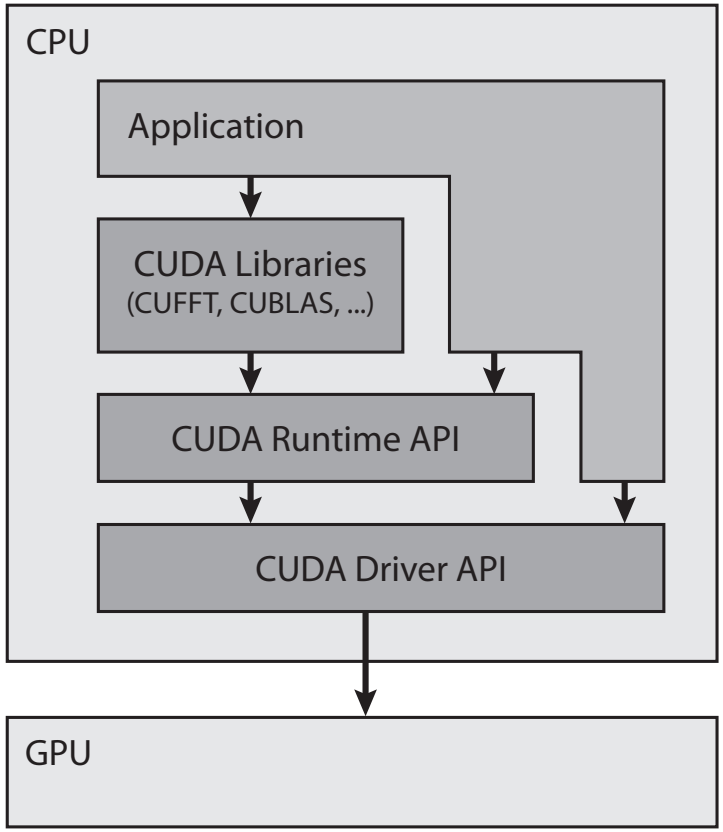



Execution time and corresponding speedup of the three different implementations performing 1,000 learning iterations of a SimardNet.

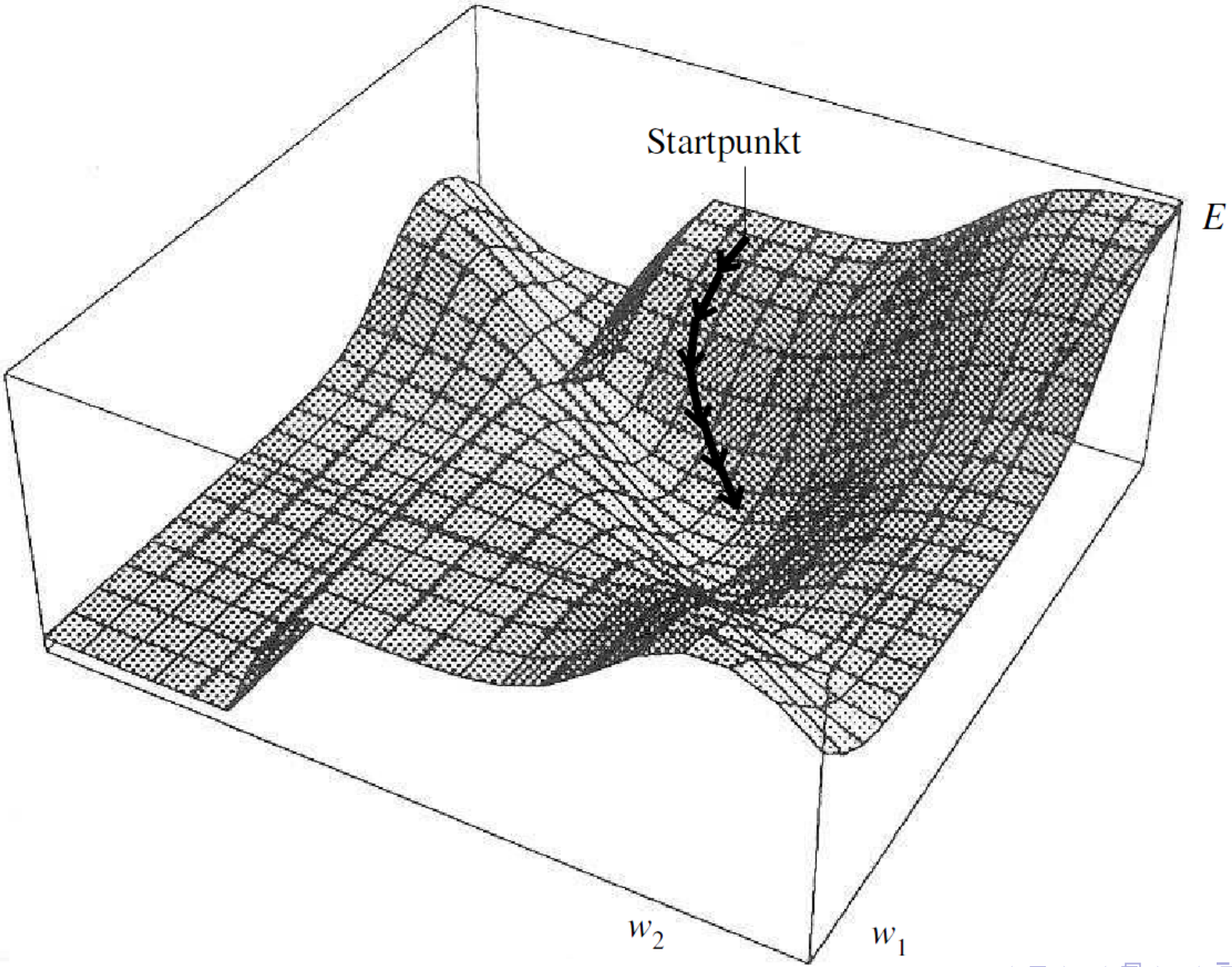
# CUDA Threading and Memory Topology



# CUDA Software Stack



# Gradient Descent Method



## CNNs using Random Connections

- Sparse connections break the symmetry of the network and can improve the network's performance.
- Michael Jahrer [9] advanced this approach by introducing random connections:
  - The connections between feature maps of consecutive layers are generated randomly at network initialization.
  - The used connection probability is around 0.1 - 0.2. Therefore a much larger network is needed to unveil its potential.
  - Using a large network with sparse random connections improves the network's accuracy but leads to a longer execution time.
  - Without modification of the dataset this network reaches a classification rate on the MNIST database of 99.15%.