Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische Friedrich-Wilhelms-Universität Bonn

Master thesis

Discriminative Convolutional Sum-Product Networks on GPU

Author:
Tobias Hartmann

First Examiner:
Prof. Dr. Sven Behnke

Second Examiner:
Prof. Dr. Joachim K. Anlauf

Advisor:
Hannes Schulz

Submitted: May 12, 2014

Declaration of Authorship

I declare that the work presented here is original and the result of my own investigations. Formulations and ideas taken from other sources are cited as such. It has not been submitted, either in part or whole, for a degree at this or any other university.

Location, Date

Signature

Abstract

Sum-Product Networks (SPNs) are a deep architecture recently proposed for image classification and modeling. In contrast to loopy graphical models commonly used in computer vision, exact inference and learning in SPNs is tractable. As long as consistency and completeness are ensured, an SPN allows to efficiently calculate the partition function and all marginals of graphical models. The proposed algorithms for generative and discriminative learning show good results on image classification benchmarks such as CIFAR-10. However, previous work did not learn image features from scratch, instead it builds on dictionary learning, leading to less comparable results. In this thesis we combined the two deep learning methods Convolutional Neural Networks and SPNs for image classification. To this end, we proposed a SPN implementation, operating in logspace for efficient computation on CPU and GPU, on top of convolutions. We found that some valid architectures for SPNs, lose information about locality of features, which significantly reduces learning capabilities. Due to time constraints, we were not able to wind up architectures, preserving this information. However, we were able to show that convolutions within the network learn reasonable structures, which show the functionality of this approach. The implementation was evaluated using the image classification benchmarks MNIST and CIFAR-10, achieving classification errors on the test datasets of 1.66% and 46.71% respectively.

Contents

List of Figures

List of Algorithms

1 Introduction

Graphical models are used in many applications, but inference for them in general is intractable (Roth, 1996). Poon and Domingos (2011) recently changed this by introducing Sum-Product Networks (SPNs), a new deep architecture, where inference is always tractable, given certain conditions. They showed that SPNs contain other models as special cases, such as hierarchical Mixture Models, and thin junction Trees. Additionally, they presented an algorithm for generative training of SPNs, which builds on gradient descent and Expectation Maximization (EM). Gradient diffusion is a well known issue for deep architectures. When the gradient is propagated to lower levels of the network, it becomes less informative since it depends on a lot of variables (Bengio, 2009). Poon and Domingos (2011) successfully applied Expectation Maximization to SPNs, which provides a strong but sparse learning signal. Using this, SPNs were successfully trained with very deep architecture and showed state-of-the-art results on several image completion tasks. Gens and Domingos (2012) enhanced the possibilities of training an SPN, by introducing back-propagation and an algorithm for discriminative training of SPNs. Another field which gained a lot of interest recently, is deep learning (Hadsell et al., 2008; Hamel and Eck, 2010; Huang and LeCun, 2006; Krizhevsky, Sutskever, and G. Hinton, 2012; LeCun and Bengio, 1995; Lee, Grosse, et al., 2011). Deep learning uses very deep architectures of Neural Networks, which are able to represent many problems in a significantly more compact way, than shallow networks. Modern Graphics Processing Units (GPUs) provide the computing power, necessary for those large neural networks.

Convolutional Neural Nets (CNNs) are a deep learning approach which is frequent topic of current research and achieved several state-of-the-art results in computer vision tasks like image classification and segmentation (Ciresan et al., 2012a; Farabet et al., 2013; Krizhevsky, Sutskever, and G. E. Hinton, 2012; Wan et al., 2013). Inspired by the mammal visual system (Hubel and Wiesel, 1968), CNNs use simple filters, which are applied to multiple sub-regions of an image. Therefore, filters within a CNN learn structures, contained at different positions within the input image. The result of different simple filters is combined by the network in order to learn filter combinations, representing more complex features.

Within this thesis we combined the two approaches of Sum-Product Networks

and Convolutional Neural Networks. The architecture examined uses convolutions to provide learned features, which are combined by a discriminative SPN, trained on top.

Modern GPUs provide thousands of computing cores in one device, which allow massive parallel computation. While Poon and Domingos (2011) relied on large clusters of Central Processing Units (CPUs), to provide the computing power necessary for SPNs, we implemented it efficiently on CPU and GPU. Computation in logspace allows efficient parallel computation of gradients from product node.

This master thesis is organized as follows: We introduce SPNs, concepts and techniques which are important for this thesis. Afterwards, we discuss related work in Chapter 3. Chapter 2 introduce SPNs and concepts and methods, which are the foundation of this work. In Chapter 4 we explain the convolutional interpretation of SPNs and the implementation. Within Chapter 5 we discuss the convolutional SPNs in detail for an exemplary dataset and present our experimental results. Finally, Chapter 6 concludes this thesis and outlines possible future work.

2 Basics

2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) (LeCun and Bengio, 1995) are one of the first successful deep learning methods. They can be viewed as a special case of Multi Layer Perceptrons (MLP). Motivated by the Neocognitron (Fukushima, 1980), they consist basically of several convolution layers, and a fully connected layer of neurons as output. Each convolutional layer generates several feature maps. A feature map is computed by convolving the input with linear filters. In order to combine the different results of the linear filter, hidden layers are composed of multiple feature maps. Since the linear filters can be represented by weights, we can think of a CNN as a Multi Layer Percepton (MLP) with shared weights. Often convolutional layers are followed by max-pooling layers. They down-sample regions by maximizing the activation, over non-overlapping rectangular regions.

For odd filter size f, input maps A, output maps B, weights $w \in f$ and input $I \in X \times Y$ the result of one convolutional filter with position $x \in X$ and $y \in Y$ is given by:

$$r(x,y) = \sum_{i=-f/2}^{f/2} \sum_{j=-f/2}^{f/2} \sum_{a \in A} w_{a,i,j} \cdot I_{a,x+i,y+j} \tag{2.1}$$

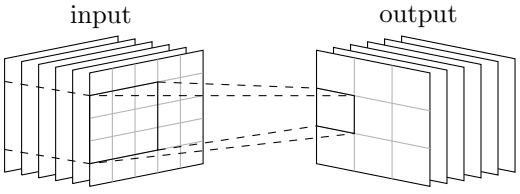

Figure 2.1: Illustration of a convolution with filters of size 3×3. The input is of size 5×5 and has six feature maps. Therefore, the output has size 3×3 and provides six feature maps too (Höft, 2014).

2.2 Maximum A Posteriori (MAP) Estimation

If we have prior information I about the investigated process, we want the most probable hypothesis h given I $P(h|I)$. Thus, using the Bayes theorem, we get

$$h_{MAP} = \arg\max_{h \in H} P(I|h)P(h)$$

where H is the set of all hypotheses.

There are several ways to estimate the MAP, one iterative method is a modified EM algorithm (Dempster et al., 1977).

2.3 Expectation-Maximization (EM)

The EM algorithm iteratively estimates the Maximum Likelihood Estimate (MLE), by two alternating steps. To this end, it introduces hidden random variables z, which represent unobserved data, starting with an initial set of hidden variables. As a result, the complete data x consists of the observed data y and z. Afterwards we consider the expectation of the log likelihood, which is a hidden variable again since it depends on random variables only. The EM algorithm maximizes this expectation, and therefore estimates the log likelihood for parameters Θ:

$$\Theta^{t+1} = \arg\max_{\Theta} E\left[\log L_C(\Theta)|y, \Theta^t\right] \tag{2.2}$$

EM is working by iteratively applying the following two steps:
Expectation Step:
Within the expectation step the value of the log likelihood is calculated, given the current set of estimated parameters Θ^t.

$$Q(\Theta|\Theta^t) = E_{x,\Theta^t}\left[\log L_C(\Theta)|y, \Theta^t\right] \tag{2.3}$$

Maximization Step:
The expectation step maximizes Q with respect to Θ:

$$\Theta^{t+1} = \arg\max_{\Theta} Q(\Theta|\Theta^t) \tag{2.4}$$

A variation of the EM algorithm to compute the MAP is given in Dempster et al. (1977). Bilmes (1997) is a good introduction for the EM algorithm.

2.4 Gibbs Sampling

Sometimes it is very expensive to evaluate a density function $f(x)$, which contains difficult integrals. Instead of calculating the joint distribution f directly, a Gibbs sampler (Casella and George, 1992) simulates the multivariate distribution in turn, and thus obtains a sample. By increasing the size of the sample, a Gibbs sampler can approximate f with arbitrary precision.

2.5 Sum-Product Networks

Sum-Product Networks are a new deep architecture, which are mainly build on the idea of a network polynomial, proposed by Darwiche (2003). They are able to compute exact inference efficiently, and contain other models like Hierarchical Mixture Models, and Thin Junction Trees (Poon and Domingos, 2011).

Definition 1 (Poon and Domingos, 2011) *A sum-product network (SPN) over variables $X_1, \ldots, X_D$ is a rooted directed graph whose leaves are the indicators $x_1, \ldots, x_d$ and $\overline{x}_1, \ldots, \overline{x}_d$ and whose internal nodes are sums and products. Each edge (i, j) emanating from a sum node i has a non-negative weight $w_{i,j}$. The value of a product node is the product of the values of its children. The value of a sum node is $\sum_{j \in Ch(i)} w_{ij} v_j$, where $Ch(i)$ are the children of i and v_j is the value of node j. The value of an SPN $S[x_1, \overline{x}_1, \ldots, x_d, \overline{x}_d]$ is the value of its root.*

In Figure 2.2 a small example of an SPN, over boolean variables, is given. There are two reasons to change the indicator variables, evidence, and marginalization. When there is evidence for a variable, the variable X_i is set to one, and $\overline{X}_i$ to zero. For Marginalization of variable X_i, we set $X_i = \overline{X}_i = 1$. Thus, we can compute the partition function Z, by setting all indicator variables to one. In case of multi-valued discrete variables, the Boolean indicators are replaced. The replacement is done by binary indicator nodes, if the set of random variables is finite. Otherwise by distribution nodes (e.g. Gaussian), when they are infinite.

Darwiche (2003) showed that all marginal probabilities can be computed for an SPN. Thus, a SPN performs exact inference. In order to do that efficiently, it has to be polynomial in the number of variables.

An SPN can be constructed in such a way, that it represents a probability distribution. In that case it is called *valid*.

Poon and Domingos (2011) have showed that two properties are sufficient for a valid SPN, namely completeness and consistency.

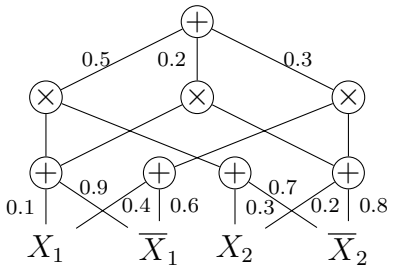

Figure 2.2: SPN over boolean variables X_1, X_2, implementing a naive Bayes Mixture Model, with three components and two variables (Gens and Domingos, 2012)

Definition 3. *A sum-product network is complete, iff all children of the same sum node have the same scope (Poon and Domingos, 2011).* Where the scope of an SPN is defined as the set off all variables that appear in it.

Definition 4. *A sum-product network is consistent, iff no variable appears negated in one child of a product node, and non-negated in another (Poon and Domingos, 2011).*

2.5.1 Training of Sum-Product Networks

Poon and Domingos (2011) showed how to train SPNs in a generative manner, by learning structure and parameters. Initially an SPN is initialized with a generic valid architecture, or learned from data (Dennis and Ventura, 2012; Gens and Domingos, 2013; Peharz et al., 2013). The weights are initialized with small positive values. Finally the learning procedure proceeds in a bottom up, top down manner. At first the SPN is evaluated bottom-up given the input, then inference is calculated in a top-down manner. Finally the weights are updated. The process is repeated until convergence, as described in Algorithm 1. Weights can be updated either by gradient descent, or EM. The final SPN then is obtained by pruning weights.

Algorithm 1: Learn algorithm for SPNs Gens and Domingos, 2012

Input: Set D of instances over variables $\mathbf{X}$, and label variables $\mathbf{Y}$, a valid SPN S with initialized parameters
Output: A SPN with learned weights
repeat
  forall the $d \in D$ **do**
    UpdateWeights(S, inference (S, x_d, y_d))
until *convergence, or early stopping condition*;

Gens and Domingos (2012) introduced a method for discriminative learning of an SPN, and back-propagation for SPNs. Using the back-propagation algorithm, it is useful to weight the weights w with $\exp(w)$. This is useful because the weights could attain negative values, due to weight updates with negative gradients, which would contradict the definition of an SPN otherwise. The exponential weighting adds an additional multiplication with $\exp(w_{ki})$ to the derivative of the weights, which is already included in Table 2.1.

In this thesis I am following the notation of Gens and Domingos (2012). Thus, an SPN $S[y, h|x]$ takes three disjoint sets of variables: hidden H, query Y and given X. And the value of the SPN is the root, denoted by S. Setting all indicator functions to one, is denoted by $S[y, 1|x]$, where the 1 is a unit vector.

To overcome the gradient diffusion problem Poon and Domingos (2011) found hard EM useful. In order to compute the Maximum a posteriori probability, which is necessary for hard EM, we just have to replace all sum nodes by max nodes. Gens and Domingos (2012) proposed an algorithm for hard gradient descent (see Algorithm 2), such that both methods are applicable for hard and soft inference. EM is typically not used for discriminative training, since it requires modification to lower bound conditional likelihood, and it is not yet known if there is a closed form for this problem (Gens and Domingos, 2012).

Since the thesis focuses on discriminative learning, only the back-propagation method is used.

In Tables 2.1 and 2.2 the different formulas for inference procedures and weight updates are given.

Algorithm 2: Backpropagation algorithm for SPNs (Gens and Domingos, 2012)

Input: A valid SPN S, where S_n denotes the value of node n after bottom-up evaluation.

Output: Partial derivatives of the SPN with respect to every node $\frac{\partial S}{\partial S_n}$ and weight $\frac{\partial S}{\partial w_{i,j}}$

Initialize all $\frac{\partial S}{\partial S_n} = 0$ except for root $\frac{\partial S}{\partial S} = 1$

forall the $n \in S$ *in top-down order* **do**

  if n *is a sum node* **then**

    forall the $j \in Ch(n)$ **do**

      $\frac{\partial S}{\partial S_j} \leftarrow \frac{\partial S}{\partial S_j} + w_{n,j}\frac{\partial S}{\partial S_n}$

      $\frac{\partial S}{\partial w_{n,j}} \leftarrow S_j \frac{\partial S}{\partial S_n}$

  else

    forall the $j \in Ch(n)$ **do**

      $\frac{\partial S}{\partial S_j} \leftarrow \frac{\partial S}{\partial S_j} + \frac{\partial S}{\partial S_n} \prod_{k \in Ch(n)\setminus\{j\}} S_k$

Node	Soft inference	Hard inference
Sum	$\frac{\partial S}{\partial S_n} = \sum_{k \in Pa(n)} \frac{\partial S}{\partial S_k} \prod_{l \in Ch(k)\setminus\{n\}} S_l$	$\frac{\partial M}{\partial M_n} = \sum_{k \in Pa(n)} \frac{\partial M}{\partial M_k} \prod_{l \in Ch(k)\setminus\{n\}} M_l$
Product	$\frac{\partial S}{\partial S_n} = \sum_{k \in Pa(n)} w_{kn} \frac{\partial S}{\partial S_k}$	$\frac{\partial M}{\partial M_n} = \sum_{k \in Pa(n)} \begin{cases} w_{k_n} \frac{\partial M}{\partial M_k} & : w_{kn} \in W \\ 0 & : else \end{cases}$
Weight	$\frac{\partial S}{\partial w_{ki}} = \frac{\partial S}{\partial S_k} S_i)$	$\frac{\partial M}{\partial w_{ki}} = \frac{\partial M}{\partial M_k} M_i)$

Table 2.1: Inference procedures for hard and soft inference(Gens and Domingos, 2012)

Update	Soft inference	Hard inference
Gen. Gd	$\Delta w = \eta \frac{\partial S[x,y]}{\partial w}$	$\Delta w_i = \eta \frac{c_i}{w_i}$
Gen. EM	$P(H_k = i\|x,y) \propto w_{ki} \frac{\partial S[x,y]}{\partial S_k}$	$P(H_k = i\|x,y) = \begin{cases} 1 & : w_{ki} \in W \\ 0 & : else \end{cases}$
Disc. GD	$\Delta w = \eta \Big(\frac{1}{S[y,1\|x]+k} \frac{\partial S[y,1\|x]}{\partial w} - \frac{1}{S[1,1\|x]+k} \frac{\partial S[1,1\|x]}{\partial w} \Big)$	$\Delta w_i = \eta \frac{\Delta c_i}{w_i}$

Table 2.2: Weight updates with different inference procedures. c_i is the number of times w_i appears in W. $\Delta c_i = c_i' - c_i''$ is defined as the difference between the number of times w_i is traversed by two MPE inference paths in $M[y, 1|x]$ and $M[1, 1|x]$. k is a small constant for numerical stability. (Gens and Domingos, 2012)

2.6 Gabor Filter

Gabor filters are bandpass filters, which are generated by multiplying a Gaussian function with a complex oscillation. 2-Dimensional Gabor filters are often used as edge detector for classification or segmentation tasks. An input image $I(x, y)$ is convolved with a two dimensional Gabor function:

$$g(x,y) = e^{-\frac{x'^2+\gamma^2 \cdot y'^2}{2\sigma^2}} \cos\Big(2\pi \frac{x'}{\lambda} + \phi\Big)$$

within

$$x' := (x \cos\Theta + y \sin\Theta)$$

$$y' := (-y \sin\Theta + y \cos\Theta)$$

λ is the wavelength, σ the standard deviation of the Gaussian, the orientation $\Theta \in [0, \pi)$ and phase offset ϕ. Usually a bank of Gabor filters with different orientation is used to obtain edges with different orientation (Kruizinga et al., 2002). An illustration is given in Figure 2.3.

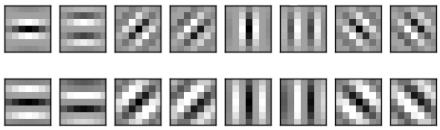

Figure 2.3: Illustration of a Gabor filter bank, containing filter for four different orientations.

2.7 The CUDA GPU Programming Framework

CPU and GPU have different strengths and weaknesses. CPUs tend to have only a few, fast computation units and therefore aim for sequential or slightly parallel computing. Modern graphics cards are specialized in highly parallel, compute-intensive computations. Therefore, they provide a lot more floating point operations and memory bandwidth per second than CPUs, as illustrated in Figure 2.4. As a result, GPUs are devoting more transistors to ALUs instead of flow control or caching, see Figure 2.5.

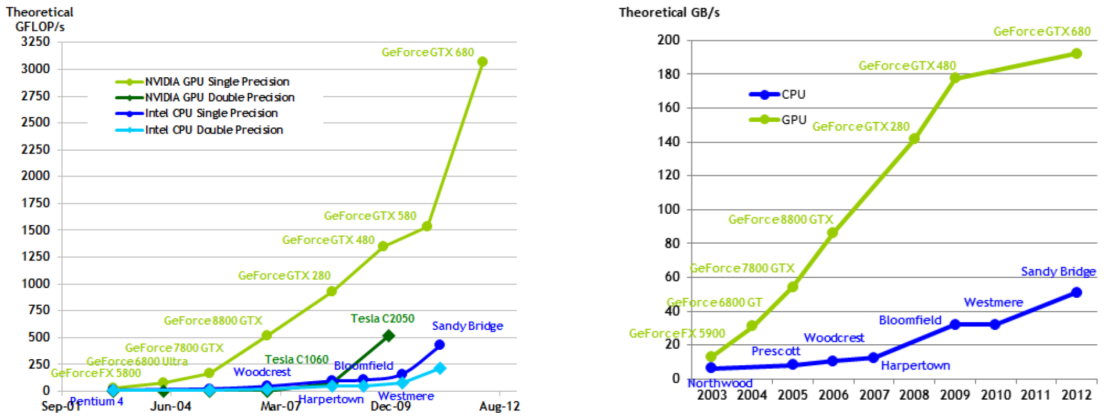

Figure 2.4: Left: Theoretical number of floating point operations per second on GPU and CPU. **Right:** Theoretical Memory bandwidth of CPU and GPU (NVIDIA, 2014a)
. The gap between performance of GPU and CPU is constantly increasing.

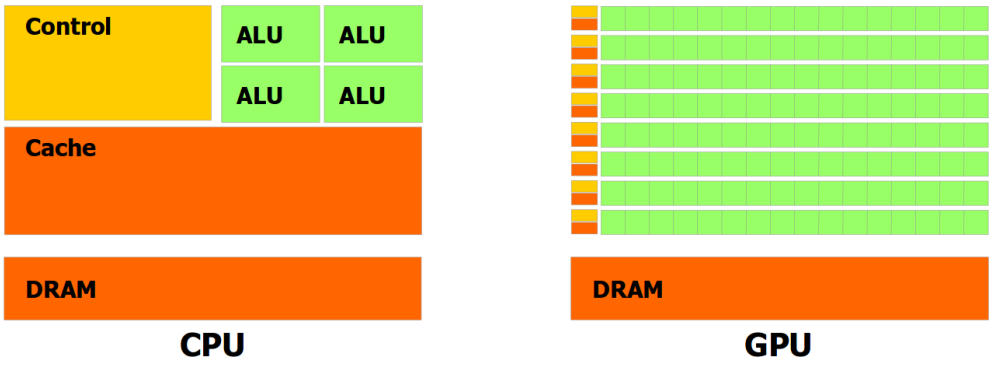

Figure 2.5: Schematic diagram of CPU (Left) and GPU (right). GPUs use larger areas for ALUs instead of caches or flow control, which is beneficial for highly parallel computations (NVIDIA, 2014a).

The CUDA Programming Model

The CUDA programming model allows the programmer to use high-level programming languages, such as C/C++ or Fortran to write GPU code. The notion of device and host memory space is used to distinguish between different memory spaces. Copying data from one to another memory space must be handled manually. In order to minimize the complexity for the programmer, the model provides three key abstractions: A hierarchy of thread groups, a hierarchy of shared memories and barrier synchronization.

The GPU is build around an array of streaming processors (SM), each executing blocks of threads independently. Each block consists of an array of threads and is located within an array of blocks, which is called "grid". All blocks can access the Global Memory space. For convenience, both, grid and block, can be one-, two-, or three-dimensional, as depicted in Section 2.7.1. The maximum number of threads is limited due to the capabilities of the SMs. The compute capability defines the features supported by NVIDIA graphics cards, which also define the maximum number of threads in one block. For devices with compute capability of $1.x$ the limit is 512, for compute capability $2.x$ (fermi) and $3.x$ (kepler) it is 1024.

Within each SM, a block is divided into units, called warps. All threads in a warp must execute the same instruction at any given time. Thus, diverging branches will be handled in several consecutive execution steps, causing a severe performance penalty. The warp size is 32 for both, fermi and kepler GPUs (NVIDIA, 2014c).

Due to this architecture exchanging intermediate results in between blocks is expensive and must be minimalized. The programmer thus has to decompose the problem into several sub problems, which can be solved by different blocks in parallel. He has to write so called "kernel methods", which are executed by every thread within a grid. The dimensionality of the grid must be specified when the kernel is called (NVIDIA, 2014a).

The programmer has not to care for the actual number of physical processors. Instead, he can use as many blocks as suited for the problem.

2.7.1 GPU Memory Hierarchy

On GPUs there are several memory types with access times in different magnitudes. Global, Constant and Texture memory are persistent during the whole kernel execution. In Figure 2.6 the memory hierarchy of GPUs is illustrated.

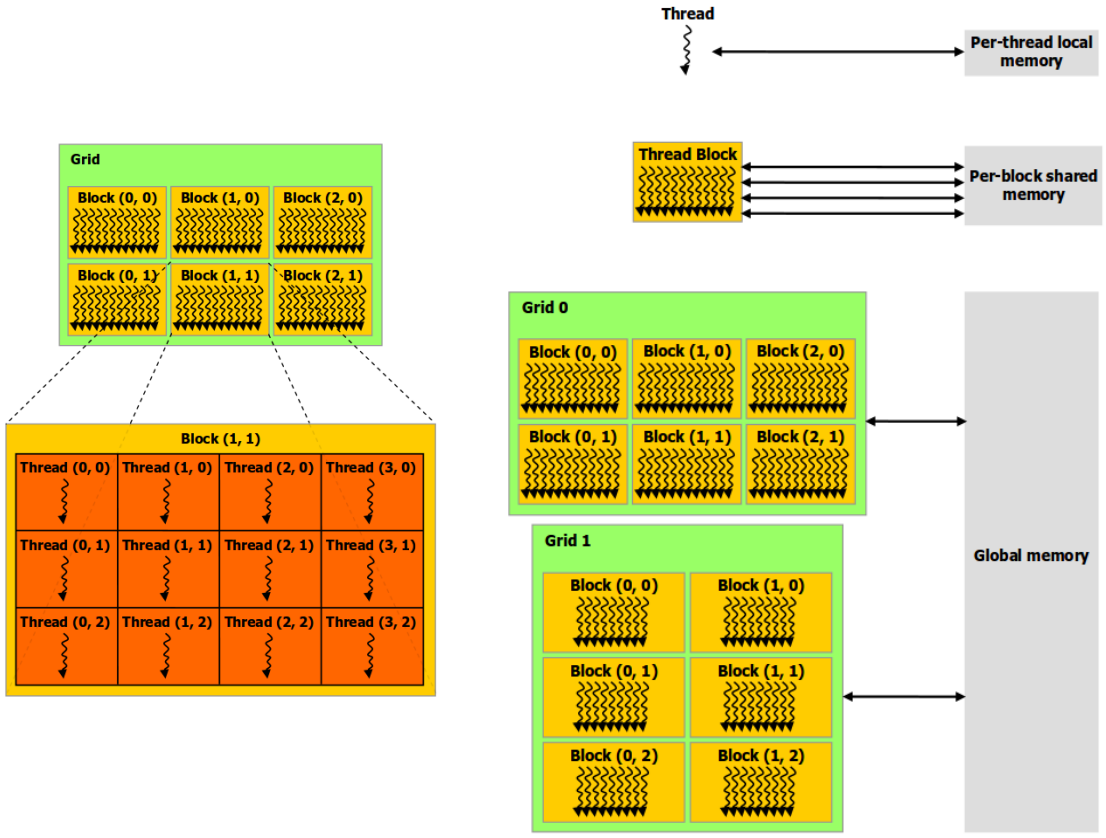

Figure 2.6: Left: Illustration of a grid with 2×3 blocks, consisting of 4×3 threads each. **Right:** Schematic diagram of the CUDA memory hierarchy. Shared memory enables efficient cooperation of threads within a block. (NVIDIA, 2014a).

Register & Local Memory Space

Every thread has an own limited amount of registers, which are the fastest memory type, but are non-persistent. On the fermi architecture a maximum of 63 registers per thread can be used, while on Kepler the maximum number of registers per thread has been increased to 255. Additionally, each thread can access Local Memory. Local Memory is in fact stored in Global Memory Space and managed by the compiler. However, since read and write operations within Local Memory are cached, Local Memory access does not always cause performance penalties.

Shared Memory Space

Each block has a small amount of Shared Memory, which can be accessed by threads within the same block. Shared Memory is slower than register, but still

about one hundred times faster than Global Memory. It is divided into several banks, which can be accessed simultaneously. However, multiple accesses to the same bank are serialized, so it is important how to distribute access to Shared Memory within banks. The number of banks is 16 for devices with compute capability 1.x and 32 for compute capability 2 or 3. In total there are up to 32kb for compute capability 1.x and up to 48kb Shared Memory for compute capability 2 or 3, located within the L1 cache for each block. Shared Memory can be used to share results between different threads. It is persistent during execution of a block.

Global Memory Space

Global Memory is the slowest type of GPU memory. Contrary to the previously mentioned memory types, Global Memory is persistent and has to be managed manually.

Constant Memory Space

There are 64kb of Constant Memory on compute capability 1.x - 3.x. Constant Memory is of the same type of memory as Global Memory and may not change during one kernel call. In contrast to Global Memory, Constant Memory is cached. Thus, access to Constant Memory only costs one read from Global Memory on a miss.

Texture Memory Space

Texture Memory is a read only Memory, which is cached similarly to the Constant Memory. Further there are some dedicated operations to optimize operations on textures, which can be used for optimization of general computing on GPU too.

2.7.2 Atomic operations

When several threads access the same element in Global Memory simultaneously and at least one access is a write operation, the outcome is not defined. In order to avoid these so called “race conditions” only one thread may write to one position in Global Memory at any given time. Thus, in general one should avoid these situations. However, sometimes this would be very restrictive with respect to parallelization.

Atomic operations are provided for Global Memory. They basically lock access to a position in Global Memory until the atomic operation is done. Other threads

that access the same position in Global Memory are delayed until the atomic operation has finished. However, sometimes they are necessary to allow parallelization through computation of partial results. Because of this atomic operations have been improved in the Fermi and Kepler architecture. Kepler GPUs are able to perform atomic operations 180 times faster than pre-fermi architectures. As a result Kepler GPUs can perform one atomic operation per clock cycle (NVIDIA, 2014b,c). This enables the use of atomic operations without any significant performance penalty in many situations.

Please note that some atomic operations are not available for compute capability $1.x$ and $2.x$.

2.8 Efficient Access to Global Memory

Global Memory is only accessed via words of size 32, 64 or 128 which are aligned. Thus, whenever a smaller area in Global Memory is accessed, memory bandwidth is wasted.

In order to optimize the access of Global Memory, a SM has dedicated hardware to minimize the memory calls of a warp. Therefore, it coalesces the memory access of all threads within a warp to the minimal number of words. Thus, the programmer has to write kernel methods in such a way, that threads within a warp access a consecutive area within Global Memory.

3 Related work

Classical learning algorithms, such as Multi Layer Perceptrons (MLP), Support Vector Machines (SVM), Radial Basis Functions (RBF), and much more, usually have a flat architecture with only few layers. While it has been proven that all of them are able to approximate universal functions, often using more layers leads to a more compact representation. Nonetheless deep networks often achieve better generalization capabilities and more compact representation. An overview over neural networks for image processing is given by Egmont-Petersen et al. (2002).

3.1 Deep learning methods

Convolutional neural networks (page 3) are a prominent deep learning architecture (Huang and LeCun, 2006; Krizhevsky, 2010a; Krizhevsky, Sutskever, and G. Hinton, 2012). One recent publication on CNNs in the field of image classification is given by Krizhevsky, Sutskever, and G. Hinton (2012). They used a deep CNN to classify the pictures of the ImageNet LSVRC-2010 contest. Usage of GPUs made the training significantly faster. Further, they used a new regularization method, called "Dropout", to reduce overfitting. "Dropout" inhibits the output of each hidden neuron, with a probability of 0.5, and thus samples different architectures. They achieved stare-of-the-art results.

Deep Neural Networks (DNNs) are another successful branch of CNNs (Ciresan, Giusti, et al., 2012; Cireşan et al., 2012; Ciresan et al., 2012b). Basically they are CNNs, with alternating convolutional, and max-pooling layers. Recently they were used by Ciresan et al. (2012b), who proposed a Multi-Column DNN (MCDNN). It averages the output of multiple single DNNs, to achieve higher tolerance for variations in contrast, and illumination of images. They basically trained multiple DNNs on differently preprocessed data. To meet the high computational effort, they used an efficient GPU implementation of DNNs; Image features where learned in a supervised way. Cireşan et al. (2012) achieved outstanding results in several image classification benchmarks. Further, they achieved a near-human classification on a traffic sign recognition benchmark (Stallkamp et al., 2011), using a MCDNN.

Another prominent deep learning technique are Deep Belief Networks (DBNs) (Hamel and Eck, 2010; G. E. Hinton, Osindero, et al., 2006; Larochelle et al., 2009; Lee, Ekanadham, et al., 2007; Lee, Grosse, et al., 2011; Nair and G. E. Hinton, 2009; Raina et al., 2009), which are probabilistic generative models. They consist of several stacked layers of Restricted Boltzmann Machines (RBMs). DBNs are trained layer wise in a greedy bottom-up manner, where layer $i+1$ is always trained using the output of already trained layer i. They are closely related to SPNs since they are another deep learning technique for probabilistic models.

An interesting approach which combines CNNs and DBNs was given by Lee, Grosse, et al. (2011). they presented a convolutional deep belief network (CDBN), which is basically a deep belief network, which uses weight sharing similar to CNNs. It can still perform bottom-up, and top-down inference. Thus, their approach allows DBNs to be scaled to high-dimensional, large images. They achieved good results in several visual recognition tasks.

3.1.1 Feature Extraction

One of the most critical steps for all learning algorithms in the image domain is feature extraction. A common way is to learn features in an unsupervised way from the data (Le et al., 2011; Ramirez et al., 2010; Vollmer et al., 2013; Yang et al., 2007). Coates, A. Y. Ng, et al. (2011) focused on this step and used only a simple MLP with one hidden layer as classifier. They were able to achieve state-of-the-art results, which shows how crucial this step is. An important insight is that different classifiers generally can only be compared in a fair way, if they use exactly the same features.

3.1.2 Pre-Training

A common problem, deep networks have to deal with, are vanishing gradients. Gradients become less informative, with increasing depth. One common way to deal with this are unsupervised pre-training methods, which aim for a good initial position on the error surface. Erhan et al. (2010) investigated how the two pre-training methods of DBNs and stacked denoising auto-encoders, effect the result in detail. The pre-training of a DBN is done by iteratively training the RBMs, in a greedy manner. Denoising auto-encoders work by iterative pre-training of the network. Each layer encodes the input x in the hidden layer $h(x)$ and has to learn the decoding function $d(x)$ in the output layer. Thus, each layer initially learns the function $d(h(x)) \approx x$. They showed that pre-training leads to more robustness, better generalization performance and avoidance of poor local minima.

3.2 Development of Sum Product Networks

Sum-Product Networks (page 5) recently gained a lot of interest.

Poon and Domingos (2011) proposed the first generative learning algorithm for SPNs. They demonstrated the algorithm on several image completion tasks, where it outperformed several other learning methods.

Gens and Domingos (2012) proposed an algorithm for discriminative learning of SPNs. Additionally, they enhanced the training possibilities by proposal of hard gradient descent, and delivered an algorithm for back-propagation for this. They achieved state-of-the-art results in two classification benchmarks for images.

So far SPNs had to be initialized with a generic structure. This forces a trade-off between precision and computational effort. In order to change this, different algorithms to learn the structure of SPNs from data were proposed.

Dennis and Ventura (2012) proposed an algorithm to learn the structure of SPNs using clustering on variables. The approach is quite limited, since it is prone to splitting highly dependent variables, and the worst-case cost of learning and size of the SPN is exponential (Gens and Domingos, 2013).

Gens and Domingos (2013) proposed a scheme for learning the structure of SPNs. The algorithm tries to divide the current variables, into approximately independent subsets, at every step. It returns the product of all recursive calls to the dataset, if successful and their sum otherwise. They evaluated the algorithm on several real-world datasets, and compared it to popular graphical model structure learning algorithms. The results were comparable in likelihood to graphical models, but inference in them was faster, and more accurate.

Peharz et al. (2013) proposed an algorithm that is learning the structure of SPNs, in a greedy bottom-up manner, and is independent of the image domain. It works by merging probabilistic models with small scope to larger, more complex, models. They achieved comparable results on an image completion tasks to (Dennis and Ventura, 2012) and (Poon and Domingos, 2011).

3.3 Inference Methods

Finding efficient representations, models and methods for fast, exact inference in Probabilistic Graphical Models (PGMs), was a topic of interest in the past decades. The development in this field is represented in several reviews (Frey and Jojic, 2005; Guo and Hsu, 2002; Jordan, 2004). Frey and Jojic (2005) compared several methods for inference, and learning in PGMs, using a vision model of multiple occluding objects and contrasts. They compared Bayesian Networks (Pearl, 2011), Directed Acyclic Graphs (DAGs) which represent a conditional probabil-

ity function for each variable in it, Markov Random Fields (MRFs) undirected graphical models where each node represents a set of random variables and a potential function for each maximal clique. Lastly Factor Graphs (FGs), which subsume both MRFs and BNs, were considered. They compared several methods for learning and inference, such as Maximum A Posteriori (MAP) estimation (page 4), Iterated Conditional Modes (ICM), the Expectation-Maximization Algorithm (EM) (page 4), Gibbs sampling (page 5), and the Sum-product algorithm.

4 Convolutional Formulation of SPNs

4.1 Motivation

Recently a lot of research focused on Deep Convolutional Neural Networks. They achieved good results in many image understanding task, such as image classification and segmentation (Ciresan et al., 2012a; Farabet et al., 2013; Krizhevsky, Sutskever, and G. E. Hinton, 2012; Wan et al., 2013).

Sum-Product Networks are a new architecture, based on exact inference, which shows good results in image classification tasks too. Gens and Domingos (2012) achieved state-of-the-art results for the CIFAR-10 image classification benchmark by discriminative training of a SPN. To accomplish this, they trained a SPN on top of learned features. Instead of learning features from images in a separate learning step, it would be desirable to learn them during training of the SPN.

By combination of the two approaches of Convolutional Neural Networks and SPNs, this could be achieved. Therefore, we investigated a SPN on top of a convolutional layer, which provides online learning of filter.

4.2 Implementation

In order to have a SPN architecture, which allows different image sizes and channels, we used the architecture for discriminative SPNs from Gens and Domingos (2012) and combined it with the scheme from Poon and Domingos (2011).

Given a classification task with C classes, the root node weights the result of C sum product networks, each representing one class. Every single SPN C_i is build in the following way:
At the top, there is a multiplication of P parts and the label y_i of class C_i. Each part is one complete decomposition of the image, but optionally two different parts may share some nodes. It consists of alternating sum/max and product layers, and a convolutional layer at the bottom.

At the bottom the convolutional layer learns a set of different distributions of the

image features. Probabilistically we consider the convolutional layer as a special type of sum layer. Therefore, it has to fulfill the property of completeness for valid SPNs. The number of filters F within this layer is defined by the number of different representations the network expects. F depends on P and the amount of different inputs for each sum node. There are two parameters for the convolutional layer: Stride S_c and filter size fx. The convolution is performed in an rectangular region of size $x \times x$. The gradient for weight w_i within the convolutional layer is given by $\frac{\partial S[y,1|x]}{\partial w_i} - \frac{\partial S[1,1|x]}{\partial w_i}$, similar to the gradient for weights in the sum/max layer.

While proceeding from top to bottom within the network, the different features of an input image are weighted and decomposed in every layer. Each sum/max layer is computing the weighted sum or weighted maximum, of an input area and reduces the number of disjoint representations of the image. This layer has two parameters: Stride S_l, which determines the distance of two consecutive nodes within the input of the layer and N, the number of input elements for each node.

Each Product layer reduces width and height of an input image, by multiplication within a $m \times m$ region. Additionally, the product layer has stride S_p, which determines the distance of the center of two consecutive filters within the input of this layer. The multiplication is implemented efficiently by sum pooling in log space, with possibly overlapping regions. It can be implemented on GPUs efficiently, since the gradient of each child can be computed completely in parallel. A detailed sketch of the sum/max and product layers is given in Figure 4.1.

Starting with a sum/max layer, the image is aggregated in this way until it is represented by one value for every part. Thus, the size of the network is determined by the strides S_p, S_l and S_c within the layers.

Moving from top to bottom, each layer has exactly S_l times the number of nodes of its predecessor. The number of layers L depends upon the amount an image is decreased within each pooling layer S_p. In order to have P independent parts, we set the size of each bottom layer to $P \cdot S_l^L$.

Finally the classification for a given image is determined by $\max_{i \in C} (w_i \cdot c_i)$. Where c_i is the result of class i and w_i refers to the weight from the root node to c_i. A detailed sketch is given in Figure 4.2.

Please note that most of the forward pass must be calculated only once, since only the two layers on top depend on the label y_i.

Finally it is easy to see that all sum nodes of the SPN are complete, since each sum/max layer represents a distribution over the same set of indicator variables. Consistency of the SPN is violated, as it is similar to the proposed architecture of Gens and Domingos (2012). Therefore, it can not be trained in a generative manner. Because consistency may be violated for evidence variables, it still can

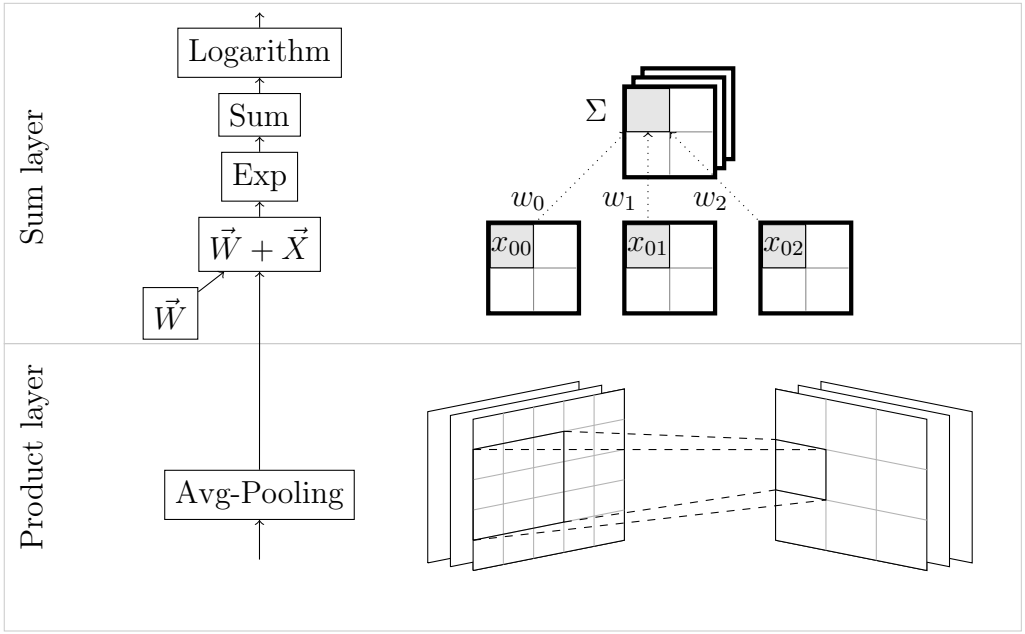

Figure 4.1: Detailed structure of the sum and product layer implementation. **Top:** The sum layer computes the weighted sum of several input maps and therefore reduces the number of different maps. The result of every position within the output map is interpreted as one sum node. Each weight is shared for the complete map and therefore for all sum nodes connected to it. In case of a Max layer, only the weighted max has to be calculated. Avoiding both exponential function and logarithm increases numerical stability. **Bottom:** The product layer is illustrated. It is implemented by average pooling in log space. Average-pooling normalizes the output of each product node.

be trained discriminatively (Gens and Domingos, 2012).

The library cuvnet (Schultz, 2013), a framework for gradient descent based algorithms on GPU was used for implementation. It basically works by exploitation of the chain-rule, which allows to calculate and derive each operation within a network independently. The cuvnet library builds upon the cuv library (Hannes Schulz, 2013), which provides matrix based operations on GPU. We enhanced both libraries for the SPN implementation.

4.3 Sum/Max layer

The sum layer performs most of the operations in an SPN, thus its performance is crucial. Let $e(l)$ be the number of sum nodes in layer l, further let i be the size of the input image in pixels and b be the batch size. A sum layer l then gets

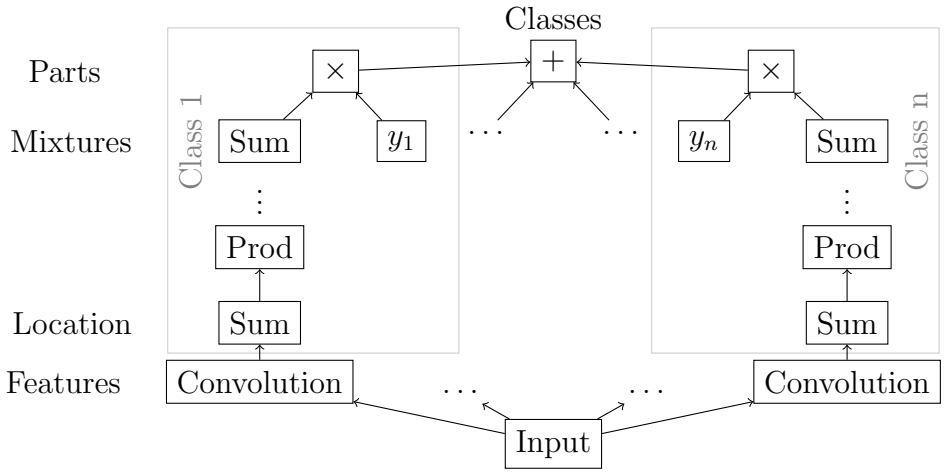

Figure 4.2: Detailed sketch from the structure of the SPN implementation. At the bottom a convolutional layer learns a feature distribution for each class. Afterwards the outcome of the convolutions is weighted and combined until each part within a class is represented by only one value. Finally the result for each class is given by the product over all parts and hte label y_i. The label is given by the maximum weighted result of all classes.

as input the three dimensional matrix, which is of shape: $e(l-1)\times i\times b$. Each sum node weights N child nodes and each weight is shared for a set of nodes, weighting an entire input image. The inference type defines the operation, which is performed for each node. In case of soft inference, the sum layer has to calculate a weighted sum, otherwise it calculates a weighted maximum. The sum/max layer is implemented using two different kernel implementations. One for the calculation of the function itself (fprop), and one for calculation of the gradient (bprop).

4.3.1 Forward Propagation Kernel (fprop)

In case of soft inference, the sum layer is calculating a weighted sum over the input nodes. Since the network is implemented in logspace, the weighted sum becomes a logaddexp function, which is calculated pairwise for all children of a node. For sum node s with two child nodes i and j it is defined by:

$$s = \log \sum (\exp(i) + \exp(j))$$

In case of hard inference, it hast to calculate a weighted maximum. Since the logarithm is a continuous and monotonous function, we can perform this operation in logspace. The operations for the second and third dimension are the same. Therefore, we can denote the input matrix X as two dimensional $X_{i,k}$, where K

refers to all elements in $i \times b$. Finally let $w_{i,N}$ be the set of weights of the i-th map of layer l. The borders are treated as input with a value of zero, which causes that the last few nodes weight less elements. However, the network can easily learn a different magnitude for these weights. Thus, the output $A_{i,k}$ is given by:

$$A_{i,k} = \begin{cases} \log(\sum_{n=0}^{N} \exp(w_{i,n} + X_{i+n,k})) & \text{if soft inference} \\ \max_{n=0}^{N} w_{i,n} + X_{i+n,k} & \text{if hard inference} \end{cases} \tag{4.1}$$

Algorithm 3: Pseudocode: Forward propagation of a sum / max layer on GPU

Input: Input $X \in I \times K$, weights $W \in I \times N$, inference type T, Stride S_l
Output: Result matrix $O \in (I/S_l) \times K$
for $(i = 0,\ \ i \in I,\ \ i \mathrel{+}= S_l)$ **do**
  load $W_{i,N}$ into shared memory array w_N
  for $(k \in K)$ **do**
    $r = w[0] + k[0]$
    for $(n = 1,\ \ n \in N,\ \ n \mathrel{+}= 1)$ **do**
      switch T **do**
        case *soft inference*
          $r = logAddExp\,(r, w[n] + X[i+n][k])$
        case *hard inference*
          $r = \max\,(r, w[n] + X[i+n][k])$
    $O[i/S_l][k] = r$

The fprop kernel starts a grid with $e(l)$ blocks and $\min\left(512, \left\lceil \frac{j}{32} \right\rceil\right)$ threads. Thus, the kernel is using I/S_l blocks. The K elements are handled by separate threads, see Algorithm 3. The maximum number of threads is given by the device capability and due to the hardware architecture it is known that GPUs work good with multiples of 32. Due to the arrangement of blocks, each block now can load the weights once into its shared memory, thus the weights must be read from global memory exactly once for each block. When restricting batch sizes to multiple of 32, a warp always operates within the same batch. Therefor it although operates within the same matrix line and access to Global Memory is performed efficiently.

4.3.2 Back Propagation Kernel (bprop)

In the bprop kernel, we have to calculate the two gradients $\frac{\partial A_{i,k}}{\partial w_{i,n}}$ and $\frac{\partial A_{i,k}}{\partial X_{i,k}}$. Let $\delta_{i,k}$ be the derivative of the parent layer p. Using the chain rule, we get the following

formula in case of soft inference:

$$\begin{aligned}\frac{\partial p}{\partial S} \cdot \frac{\partial A_{i,k}}{\partial X_{i,k}} &= \delta_{i,k} \cdot \frac{\exp(w_{i,n} + X_{i+n,k})}{\sum_{n=0}^{N} \exp(w_{i,n} + X_{i+n,k})} \\ &= \delta_{i,k} \cdot \frac{\exp(w_{i,n} + X_{i+n,k})}{\exp(A_{i,k})} \end{aligned} \tag{4.2}$$

$$\frac{\partial p}{\partial S} \cdot \frac{\partial A_{i,k}}{\partial w_{i,n}} = \delta_{i,k} \cdot \sum_{k} \frac{\exp(w_{i,n} + X_{i+n,k})}{\exp(A_{i,k})} \tag{4.3}$$

Hard inference:

$$\frac{\partial p}{\partial M} \cdot \frac{\partial A_{i,k}}{\partial X_{i,k}} = \begin{cases} \delta_{i,k} \cdot A_{i,k} & \text{if } A_{i,k} = \max_{n=0}^{N} w_{i,n} + X_{i+n,k} \\ 0 & \text{else} \end{cases} \tag{4.4}$$

Let

$$c_i = \begin{cases} 1 \text{ if } \max_{n=0}^{N} w_{i,n} + X_{i+n,k} \\ 0 \text{ else} \end{cases} \tag{4.5}$$

then

$$\frac{\partial A_{i,k}}{\partial w_{i,n}} = \sum_{k} c_i \tag{4.6}$$

The bprop kernel is configured similar to the fprop kernel. Again it has $e(l)$ blocks, but the threads are organized as 2-dimensional matrix of fixed size 16×16. We chose this architecture since experiments showed that the resulting 2-dimensional memory access results in a slightly better performance. In order to reduce the number of calls to global memory, again each block loads the corresponding weights into shared memory once. The 2-dimensional thread matrix calculates 16 items of 16 batches at once and is moved over the output matrix, defined by $e(l)$, in an iterative manner. Thus, each thread calculates the derivative of one node at a time. In order to reduce the computational effort, we use the stored result of either the fprop step, or the argmax, depending on the inference type.

Since the weights of a node are shared, we would have to store the weight derivatives for each thread in shared memory, causing additional $256 \cdot N \cdot size\_of(float)$ shared memory usage. Because of the limited amount of shared memory, we sum the result after calculation of each partial weight derivative. Thus, we just have to store one temporary value per thread and additional one temporary sum for each weight resulting in $(256 + n) \cdot size\_of(float)$ shared memory usage. The summation after each step is done by parallel reduction.

Another possible conflict arises out of the possibly overlapping input areas of

Algorithm 4: Pseudocode: Back propagation of a sum / max layer on GPU

Input: Input $X \in I \times K$, weights $W \in I \times N$, inference type T, Stride S_l, Argmax $A \in (I/S_l) \times K$, result matrix $O \in (I/S_l) \times K$, derivative of parent layer $\delta_{I/S_l,\,K}$, small numeric constant ϵ

Output: $\Delta X \in I \times k$ and $\Delta W \in (I/S_l) \times N$, , (Derivatives $\frac{\partial A_{i,k}}{\partial X_{i,k}}$, $\frac{\partial A_{i,k}}{\partial w_{i,n}}$ for all nodes within the layer)

for $(i = 0,\ \ i \in I,\ \ i \mathrel{+}= S_l)$ **do**
- load $W_{i,N}$ into shared memory array w_N
- shared memory array $d[N]$
- **for** $(k \in K)$ **do**
  - **switch** T **do**
    - **case** *soft inference*
      - $d = \frac{\delta[i][k]}{\exp(O[i][k]) + \epsilon}$
    - **case** *hard inference*
      - $d = \delta[i][k]$
  - **for** $(n = 0,\ \ n \in N,\ \ n \mathrel{+}= 1)$ **do**
    - **switch** T **do**
      - **case** *soft inference*
        - $tmp = \exp(w[n] + X[i+n][k])$
        - $\Delta X[i+n][k] \mathrel{+}= tmp$ `// atomic add`
        - $d[n] \mathrel{+}= tmp$
      - **case** *hard inference*
        - **if** $n = A[i][k]$ **then**
          - $\Delta X[i+n][k] \mathrel{+}= p$ `// atomic add`
          - $d[n] \mathrel{+}= 1$
- **for** $(n \in N)$ **do**
  - $\Delta W[i][n] = d[n]$

the nodes. A large overlap between nodes would significantly reduce the learning capabilities of the SPN. Therefore, we can safely assume that two blocks operate mostly with disjoint variables. As a result we can use atomic addition to store the derivatives of $\frac{\partial A_{i,k}}{\partial X_{i,k}}$. Only in very few cases two threads will try to access the same position in Global Memory at the same time and thus have to wait. Therefore, using atomic operations to store the derivatives practically does not slow down the computation significantly on modern GPUs (Section 2.7).

4.3.3 Performance Evaluation

Finally we compared the time consumption of GPU and CPU implementation, using a single CPU core. The experiments were performed on a System with an Intel Core i7 X980 CPU and a Tesla K20c GPU (Kepler). In Figure 4.3 we can see that the fprop kernel achieved a large speedup, even for a small number of nodes. Since the result of the forward propagation is used in the back propagation step, less computation is necessary. Additionally, more write operations to global memory must be performed, since several derivatives are calculated. As a result, the back propagation kernel is significantly slower than the forward propagation. Anyway, it achieves a large speedup when compared to CPU performance.

4.3.4 Weight Updates

Since our SPN implementation calculates the result in logspace, we have to recalculate the gradient of the conditional log likelihood. It takes the form:

$$\begin{aligned}\frac{\partial}{\partial w}\log P(y|x) &= \frac{\partial}{\partial w}\log\sum_h \Phi(Y=y,H=h|x) - \frac{\partial}{\partial w}\log\sum_{y',h}\Phi(Y=y',H=h|x)\\ &= \frac{\partial \log S[y,1|x]}{\partial w} - \frac{\partial \log S[1,1|x]}{\partial w}\end{aligned} \tag{4.7}$$

The two summations are separate bottom-up evaluations of the SPN with indicators set as $S[y,1|x]$ and $S[1,1|x]$.

In case of soft inference weight updates in logspace are given by:

$$\begin{aligned}\Delta w &= \eta\cdot\left(\frac{1}{S[y,1|x]}\cdot\frac{\partial S[y,1|x]}{\partial w}\cdot\frac{1}{S[1,1|x]}\frac{\partial S[1,1|x]}{\partial w}\right)\\ &= \frac{\partial \log S[y,1|x]}{\partial w} - \frac{\partial \log S[1,1|x]}{\partial w}\end{aligned} \tag{4.8}$$

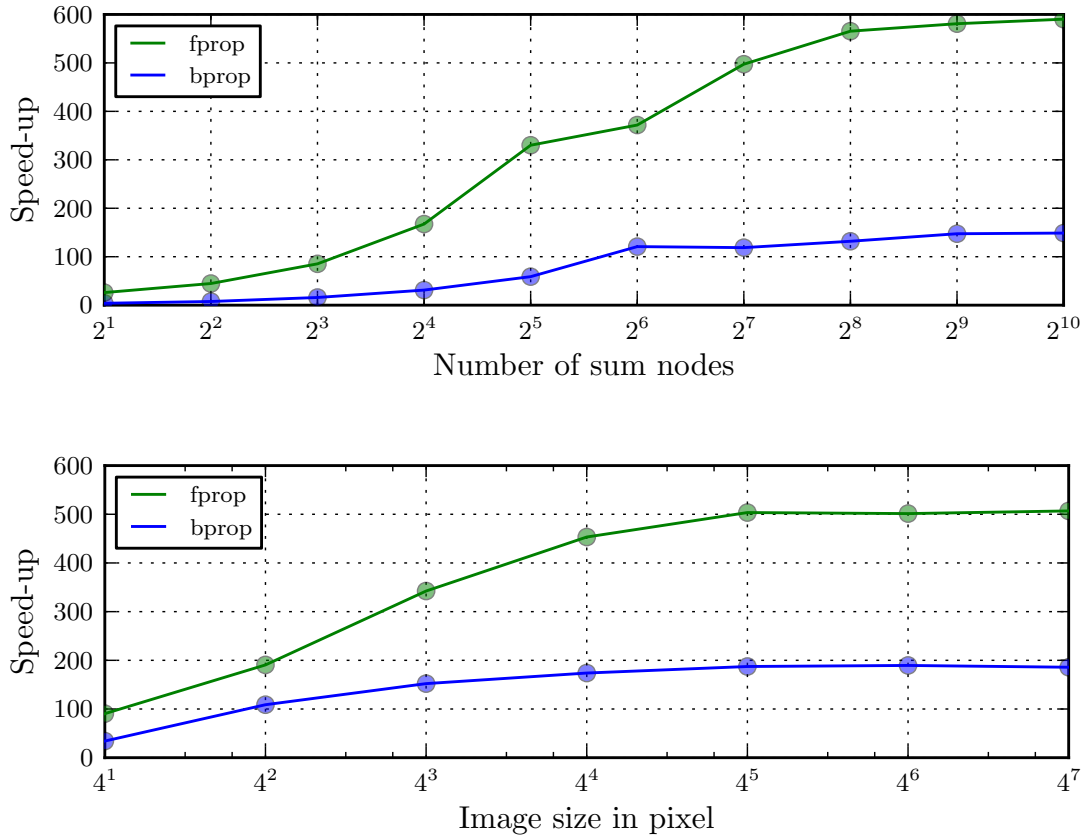

Figure 4.3: Comparison: Performance of CPU vs GPU implementation. The implementation scales well for increasing image size and number of sum nodes. Both kernel are significantly faster than the corresponding CPU implementation. **Top:** Performance comparison for increasing number of sum nodes (input maps). Within the measurement, we used a batch and image size of 32, yielding a matrix of size $N \times 1024$. **Bottom:** Performance comparison for increasing size of the input image. Within the measurement we used a batch size of 32 and 128 sum nodes.

Since the weights within the SPN are in logspace, we get the weight update rule for weight w and soft inference:

$$\log w_{t+1} = \log \left(\exp(\log w_t) + \eta \cdot \left(\frac{\partial \log S[y, 1|x]}{\partial w_t} - \frac{\partial \log S[1, 1|x]}{\partial w_t} \right) \right) \tag{4.9}$$

In case of hard inference, weight updates in logspace become $\Delta w_i = \eta \cdot \Delta c_i$, where c_i is the difference between the number of times w_i is traversed by two inference paths in $M[y, 1|x]$ and $M[1, 1|x]$ (Gens and Domingos, 2012).

The weight sharing in our implementation led to significant differences in the magnitudes of gradients between layers in case of hard inference. Since each weight is shared for the whole input image, the magnitude of Δc_i depends upon the number of nodes w_i is used in, instead of the actual magnitude of the gradient. Because the size of the image is strictly decreasing from bottom to top of the network, the magnitude is increasing with depth of the network. In order to cope with this property, we introduced a factor $\eta_l = \frac{1}{imgSize}$, in order to normalize the magnitude of gradients for weights.

However, this property does not hold for the convolutional layer, which still depends on the magnitude of the gradient. Because of this, we used a separate factor η_c, which allows the usage of a different learn rate for the convolutional layer.

In case of soft inference this effect did not occur, since the magnitude of the gradient decreases significantly with increasing depth.

Additionally, we rescaled the weights after every weight update, in order to prevent large weights, which might cause diverging values within the network. Gens and Domingos (2012) projected the weights on the surface of the unit sphere for this purpose. We found empirically, that projecting the weights into the unit sphere, after each weight update, led to better results. Thus, all weights are projected into the unit sphere after every weight update.

4.4 Discussion

We were able to implement efficient kernels for computation of the sum/max layers on GPU. When considering the amount of physical cores of the CPU, the fprop implementation on GPU was approximately 12.5 up to 75 times faster than on CPU. The back prop implementation on GPU was approximately 2 up to 25 times faster than on CPU. Since the second dimension of the input matrix is defined by $i \times b$ we can safely assume that very small matrices won't usually be used.

In our first approach, we implemented a version of the network where all classes

shared one convolutional layer. This would have the advantage that features, which occur in multiple classes must only be learned once. Additionally, more complex features could use more filters than simple ones. However, we found that the magnitude of gradients differs by more than 10^{-5}. Because of this property, some classes dominate the filter, which harms the learning capability of the network significantly.

The proposed architecture can easily be adapted for computation on multiple GPUs, since synchronization between multiple GPUs would have to take place at the root node only. This would yield even better performance than a single GPU. Additionally, larger networks could be realized, since more GPU memory is available.

5 Experiments

For the experiments we initialized the weights w of all sum/max nodes with $log(5/N) + r$, where r denotes a random value in range $-0.05 \leq w \leq 0.05$. Weights of the convolutional layer were initialized with values in range of $-0.5 \leq w \leq 0.5$.

In order to cope with boundaries of the image, the pooling layer must have overlapping pooling areas for two adjacent nodes. For stride and filter size within pooling, we found that a filter size of 2×2 and stride $S_p = 2$ usually achieves best results, since less information concerning the position of a feature is lost per step.

During training we had to choose multiple parameters. Unfortunately, training the network on a small dataset like MNIST was already time consuming. Therefore, we were not able to optimize all parameters automatically using grid search, or more sophisticated methods. As a result, we optimized the sizes of the filter masks, strides S and the number of parts of the network by hand. After this, we optimized the learn rate η and the learn rate factor η_c by grid-search on a small subset of 256 training patterns.

Given the initial learn rate, the network was trained for i epochs, until the SPN error, given by $|S[y, 1|x] - S[1, 1|x]|$, did not improve by at most ϵ percent. Afterwards the learn rate was decreased by a factor f. This procedure was repeated t times. The detailed algorithm is described in Algorithm 5.

Algorithm 5: The training algorithm for the SPN

Input: Learn rate η, number of repetitions t, factor f, number of iterations it

for $(i = 0;\quad i < t;\quad i \mathrel{+}= 1)$ **do**
  repeat
    Train network for it epochs
    Obtain SPN error e
  until $(e \cdot f) < e$;
  $\eta = \eta \cdot f$

5.1 Small Toy Dataset

In order to check whether the convolutional layer is working as expected, we generated a small dataset. It consists of 28×28 gray-scale images, each representing one of the four classes: Circle, rectangle, horizontal line and vertical line. Structures within discriminatively trained filters may be hard to recognize. The simplicity of this dataset helps to get a deeper understanding of the network and discriminatively trained filters should be recognizable.

Normally distributed noise with mean zero and a standard deviation of $\nu = 0.5$ was added to each image before an epoch, such that the network can not learn the images by rote memorization, as it did not get exactly the same images twice.

Before training, the images were normalized to range $[0 : 1]$.

5.1.1 Experiment 1: Small Filter

As first experiment we trained a SPN with only sum layer on the images. As parameters we chose the number of parts $P = 1$, a batch size of 32, a pool filter of 2×2, a stride within pooling of $S_p = 2$ and constant for numerical stability $\epsilon = 10^{-6}$. Within the sum layer, we did set $S_l = 2$ and the number of children for each sum node to $N = 4$. Further, we used filters of size 7×7 and a stride of $S_c = 3$ within the convolutional layer. Thus, the resulting SPN had 6 layers, including the convolutional layer and root, which provided 16 filters for each class. It was trained for 200 epochs with learn rate $\eta = 0.002$ within the SPN and a learn rate factor of $\eta_c = 50$, resulting in an initial learn rate of 0.1 for the convolutional layer.

Weight decay forces the network to use less of the filter and therefore helps to learn clearer structures, which are easier to understand for humans. Therefore, we introduced a weight decay factor of $wd = 0.1$ for experiments with this dataset only.

Results

In Figure 5.1 we see that the SPN was able to correctly classify the four classes after approximately 40 epochs. Afterwards the SPN error is still decreasing, and converges slowly to a value of approximately 0.2.

Discussion

In Figure 5.4 we illustrated some of the 64 learned filters for each class. The structure from the learned filters of the classes “vertical” and “horizontal line”

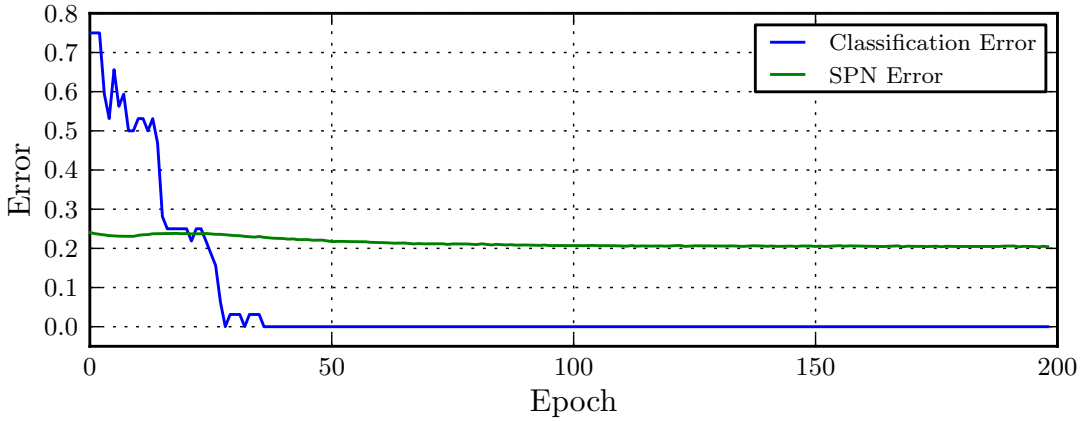

Figure 5.1: Development of classification error and SPN error during training with 7×7 filter in Experiment 1.

seem reasonable. We can easily recognize an learned edge filter, and its inverted version, which will get a maximal response above and below the vertical line. The maximal number of filters, with recognizable structure, in one class was three. This shows that the network has a lot of capacity left.

While the filter from the class "cross" might still show some reasonable structure, the filter from the circle class seemed odd. Within the learned filter of this class we could not find any recognizable structure. Additionally, we noticed that the learned filter within this class are very similar, while we can find significant difference within the learned filter of the other classes.

5.1.2 Experiment 2: Large Filter

In order to get a better understanding of the network, we did a second experiment with the same dataset. We used the same parameters as in experiment one, but used large filters of size 21×21 within the convolutional layer. For this experiment we padded the images with one row and column of zeros, in order to get the same size of the network. Afterwards we trained the network for 200 epochs with learn rate $\eta = 0.004$ and $\eta_c = 10$, which was found empirically.

Results

In Figure 5.2 we can observe that the SPN error is increasing slightly in the beginning, before the error starts to decrease and slowly converges to approximately 0.2. After two hundred epochs it is not yet converged completely. However, the classification is learned much faster than in experiment one and after only ten epochs

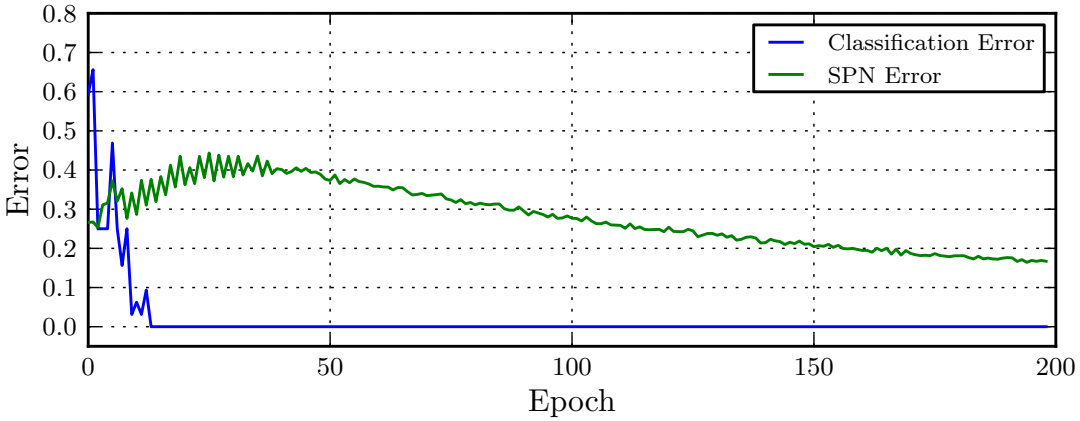

Figure 5.2: Development of classification error and SPN error during training with 21×21 filter in Experiment 2.

all images are classified correctly.

Discussion

We can explain the slower convergence of this network by the significantly larger amount of weights in the convolutional layer, which simply need more epochs to be learned correctly. When we look at the learned filters in Figure 5.4, we can recognize the structure from each class clearly. Additionally we can find structures with negative weights within the filters, representing structure from a different class. For example in the class "cross" the lines of the cross are followed by a negative line, which helps do differentiate between cross and horizontal or vertical line.

Additionally, we noticed that within both of the simple classes of horizontal and vertical line only five filters learned weights of significant magnitude. For the class "cross" there are six and finally nine significant filters for the class "circle". Thus, the network automatically uses more filters for more complex structures.

5.1.3 Experiment 3: Max Pooling

Because of the results of the first two experiments on the toy dataset, we did one last experiment with it. In order to avoid excessive weight sharing, we replaced the average pooling with a max pooling layer. The result of this alteration is that filters are no longer combined within the pooling layer. Instead, the largest results are weighted within the next sum layer. In the end a part may consist of features with completely different positions within the image.

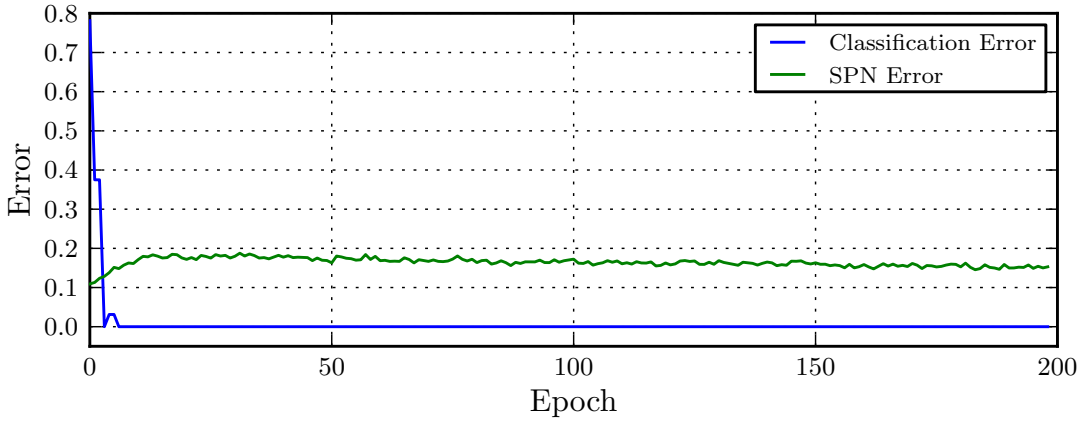

Figure 5.3: Development of classification error and SPN error during training with 7×7 filter and max-pooling in of Experiment 3

The only product node remaining in the network, is the one combining the parts within each class. This alteration of the network significantly decreases the amount of weight sharing and therefore small filters should be able to learn structures within an input image.

We repeated experiment one with the mentioned alteration to the network. Thus, all parameters were the same, except for the number of parts. We used $P = 4$ parts within this experiment since the expressiveness of one part was significantly reduced by the alteration.

Results

In Figure 5.3 we can observe that the dataset is classified correctly within the first ten epochs. The SPN error is increasing in the beginning and afterwards it starts converging slowly.

The learned features clearly represent the structures of the different classes. In their illustration in Figure 5.4, we can find partial arcs and parts of the cross. The features learned for the classes of horizontal and vertical lines are similar to the learned filters from experiment one.

5.1.4 Discussion

The three experiments with the small dataset delivered interesting results. In all experiments the SPN learned to distinguish the four classes easily. When using small filters sizes in the convolutional layer, we could not find the expected structures of the corresponding images within the filters of all classes. Increasing

the filter size did yield better results with respect to the learned filters. The number of filter which showed a recognizable structure were different for different classes. More complex structures within the images of a class led to more filters with recognizable structures. This can be explained by the learning algorithm of the SPN. As soon as a class within the SPN yields the same result in the marginalized evaluation and the evaluation with labels, it does not need to learn more filters. Additional filters are not used in this case and simply get a small weight.

Note, that the SPN error does not reach zero in any of the experiments. This is caused by the root sum node of the network. The SPN error will only reach zero, if the only class which generates results larger than zero is the correct one. Since some features may overlap, the two terms of the gradient compete against each other. This allows further learning after all images are classified correctly.

When choosing a max node as root, the behavior is clearly different. Since only the class with maximum value is considered by the SPN, the SPN error directly reaches zero value when all examples are classified correctly. However, the SPN has finished learning as soon as all training images are classified correctly.

Within experiment three, we found that very large filters are necessary in order to learn the structures within the images. One would expect much smaller filter sizes, similar to the size of filters in experiment one.

Convolutional Neural Networks use alternating layers of convolutions and pooling operations too. When comparing CNNs to the convolutional architecture for SPNs in this thesis, we find two major differences:

The first difference is that a convolutional network still has at least a small set of weights within each pooling layer. The convolutional use of this weight mask allows learning of patterns, which represent higher order features. This is desirable since we assume some repetitive structure within the images of one class. Additionally, it provides the CNN with some information about the positions of features within the image.

The second difference is that a CNN usually reduces the input image to a $n \times n$ output by succeeding layers of convolution and pooling. It is followed by a fully connected neural layer, which again preserves some information about the position of features.

In contrast to this, our SPN applies alternating layers of sum or max nodes and pooling until the whole image is represented by only one value for every part. The sum layer within our SPN implementation can be interpreted as 1×1 convolution on every input map. By alternating application of these degenerated convolutions and average-pooling, the network therefore combines the results of all filters before. All information about the position of features is lost. Therefore, it is no longer able to detect a pattern of higher order features with respect to different

positions within the image. Rather, it learns a set of filters which generate a large response at most positions, a convolutional filter from the bottom layer is applied to. Therefore, patterns which occur frequently can still be learned, while less frequent patterns, such as the arc of a circle, can hardly be learned anymore. As a result the SPN can not weight a learned filter depending on its position within the image at all.

The learned filters of both experiments can be explained by this insight. In case of the experiment with filter size 7×7 we can recognize the structure of the lines within the learned filters. They can be learned since the filter has the same response at several positions. Within the class "circle" we would expect the filters to represent a curve. Since every part of the filters would fit exactly one position, only the intersect of all circle parts is learned. Within learned filters of experiment two, we can find that the same feature is learned at several positions within one filter, see Figure 5.4. This allows the network to recognize the same feature at different positions within the image. However, it is clear that this will usually result in bad generalization capabilities of the network.

By replacing average-pooling with max-pooling in experiment three, the network preserves information about the position of features. One part within the SPN now represents a set of learned filters from the convolutional layer. This set may consist of different filters which are located at arbitrary position within the image. A part however no longer represents one decomposition of the whole image. This is not harmful, since we still use multiple parts and a filter of one class can learn the representation within the whole image.

The Probabilistically max-pooling is interpreted as max node with fixed weights of one. Therefore, the SPN does not fulfill the property of completeness anymore. Incomplete sum nodes undercount the true marginals (Gens and Domingos, 2012), which will lead to worse results. However, results of experiments presented in the following chapter suggest that he location of features is even more important than the violation of completeness.

The network is now able to learn less frequent patterns with max-pooling, as demonstrated in experiment three. Unfortunately the changes prevent the network from representing arbitrary SPNs, since just one product node is left within the network.

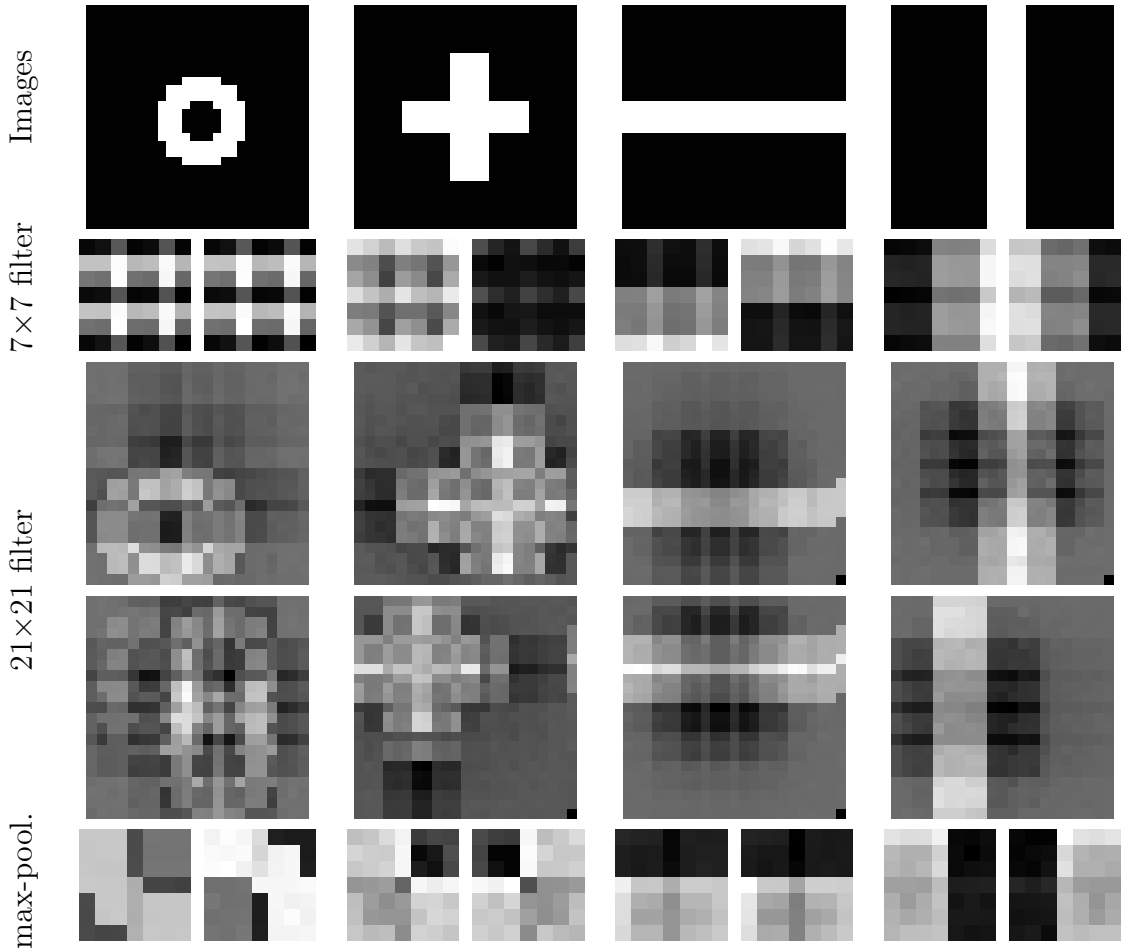

Figure 5.4: Toy dataset: Illustration of images and learned filters. **Top:** Sample images of the toy dataset (without noise). **Upper center:** Learned filters of size 7×7 from experiment one, structures within the "line" classes are observable. **Lower center:** Learned filters of size 21×21 from experiment two. Structures from the images are clearly recognizable, but often learned at multiple positions by the same filter. **Bottom:** Learned filters from experiment three with 7×7 filters and max-pooling. Parts of the structure, showed by the images, like the partial arc of a circle is recognizable. Filters within a column are from the filter bank, which represents the class, illustrated by the image on top. The filter banks were rescaled in range of $[0 : 255]$ for better visualization. Therefore, bright pixel represent large weights and dark pixel represent small weights.

5.2 MNIST

5.2.1 The Dataset

MNIST (LeCun and Cortes, 2010) is a handwritten digit recognition benchmark. It consists out of a total of 70000 images of handwritten digits in gray scale, 10000 are reserved as test set. The images are size-normalized, while preserving their aspect ratio and centered within a 28×28 image. Therefore, the images contain gray-level, while the original images from NIST are bipolar. Since there are a lot of results reported with different learning methods, MNIST is well suited for comparison of results.

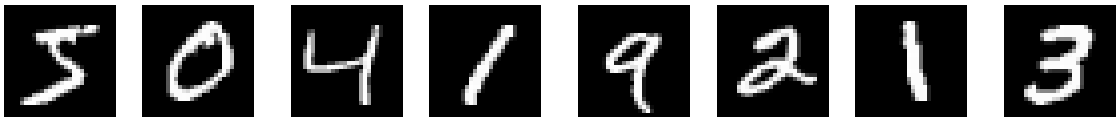

Figure 5.5: Example images: The first 8 images of the MNIST training set.

5.2.2 Experiments

While experimenting with different configuration of the network we examined different parameters, including different combinations of sum and max layers. The root node has the most influence on the result. When a sum node is chosen as root for the network, all classes get a gradient proportional to their result of the forward step within the partial gradient $\frac{\partial \log S[1,1|x]}{\partial w}$. Since just one class gets a gradient from the evaluation with labels $\frac{\partial \log S[y,1|x]}{\partial w}$, most of the classes get a negative gradient proportional to their result. Therefore, using a sum node as root enforces discriminative training in a broad manner.

When a max node is chosen as root, only classes which are responsible for a misclassification will get a negative gradient within the gradient of the marginalization step. Therefore, only classes, which are relatively similar to each other are forced to use different features.

We accomplished best results with a max node as root of the SPN, and several sum layer below. Parameters were optimized by hand to parts $P = 100$, a batch size of $b = 256$ and $\epsilon = 10^{-}8$. Further, we chose $N = 4$ children for every sum / max node, and a stride $S_l = 2$ for sum layer. Max-pooling was performed with a filter size of 2×2 and stride $S_p = 2$. Finally, within the convolutional layer we chose stride $S_c = 4$ and a filter size of 9×9. Each image was padded with three additional pixels of value zero in order to avoid effects at the border of the images within the pooling operation.

Given these parameters, we found the initial learn rates by gridsearch, resulting in initial learn rate $\eta = 0.00351$ and learn rate factor for the convolutional layer of $\eta_f = 51$.

The parameters above yield a SPN with five layers and eight hundred filters within the convolutional layer of each class.

The network was trained with these parameters using Algorithm 5. When the learn rate did not improve at least $f = 2\%$ within $it = 25$ epochs, the learn rate was multiplied by $f = 0.5$. This procedure was repeated six times.

Before training the images were rescaled to a range of $[0 : 1.0]$. In order to enforce more generalization within the filter of the convolutional layer we added normally distributed noise, with standard deviation of $\nu = 0.5$, to each image before every epoch.

Results

The SPN achieved a classification error 1.66% on the test set and 1.26% on the dataset for training. Learn rates were reduced before epoch $30, 180, 480$ and 600. We can observe the impact of the learn rate changes in Sections 5.2.2 and 5.2.2.

For each class, we illustrated some learned filters in Figure 5.7. We can clearly recognize parts of the digits, which were learned by filters of the corresponding class.

5.2.3 Discussion

When looking at the illustration of the classification error during training in Section 5.2.2, we observe that the SPN did not achieve a classification error of zero on the training set. This is an effect of the noisy images. Without noise, we found that the classification error on the training set reached zero, but we achieved significantly worse results on the test set in this case.

Additionally, one might notice the large amount of filters, that is learned within the network. Due to max-pooling within the network, we have to learn significantly more filters, than actually used within the network. When looking at the learned filters, we noticed that not all of them show structure. This can be explained with the networks properties induced by max-pooling. When a set of filters which is used by only one sum node is learned, the remaining filters of the corresponding part may never yield the maximal value and thus some filters, may not be learned at all.

Our best results on MNIST achieved a classification error of 1.66%. In Figure 5.7 we illustrated some hand picked filters the SPN learned on the MNIST dataset.

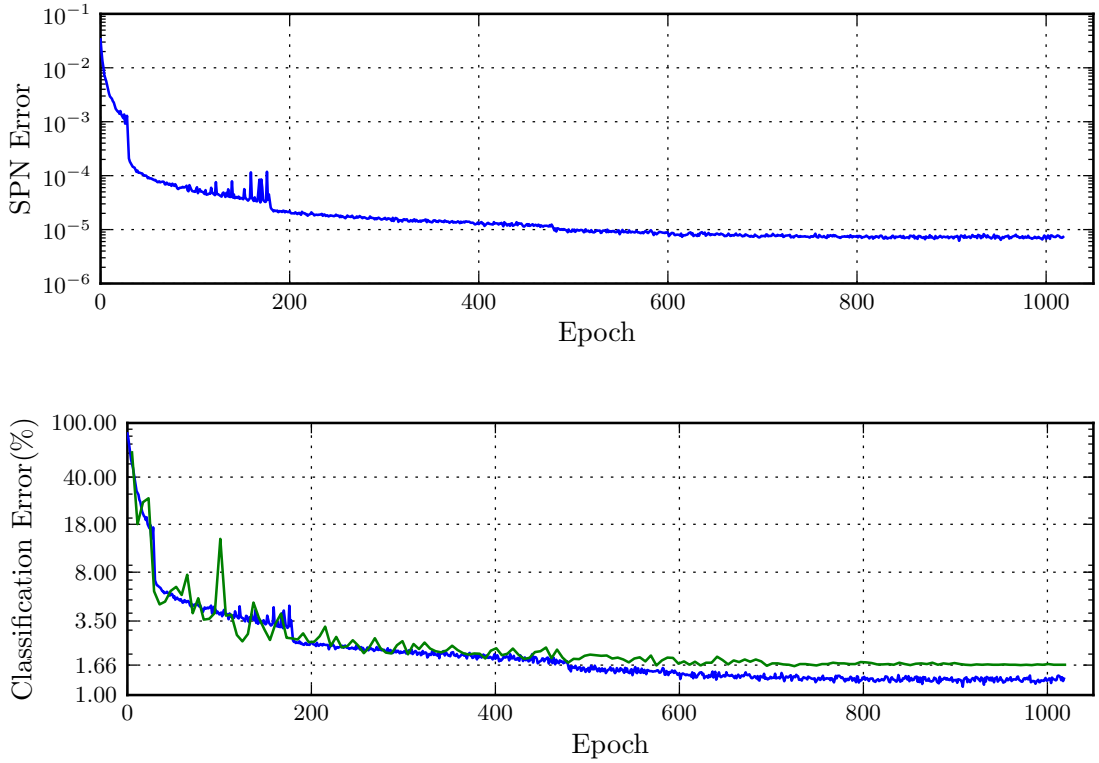

Figure 5.6: Development of SPN error (top) and classification error on train and test set (bottom) during training of MNIST. A step of SPN error and classification error on the train set can be observed after changes of the learn rate.

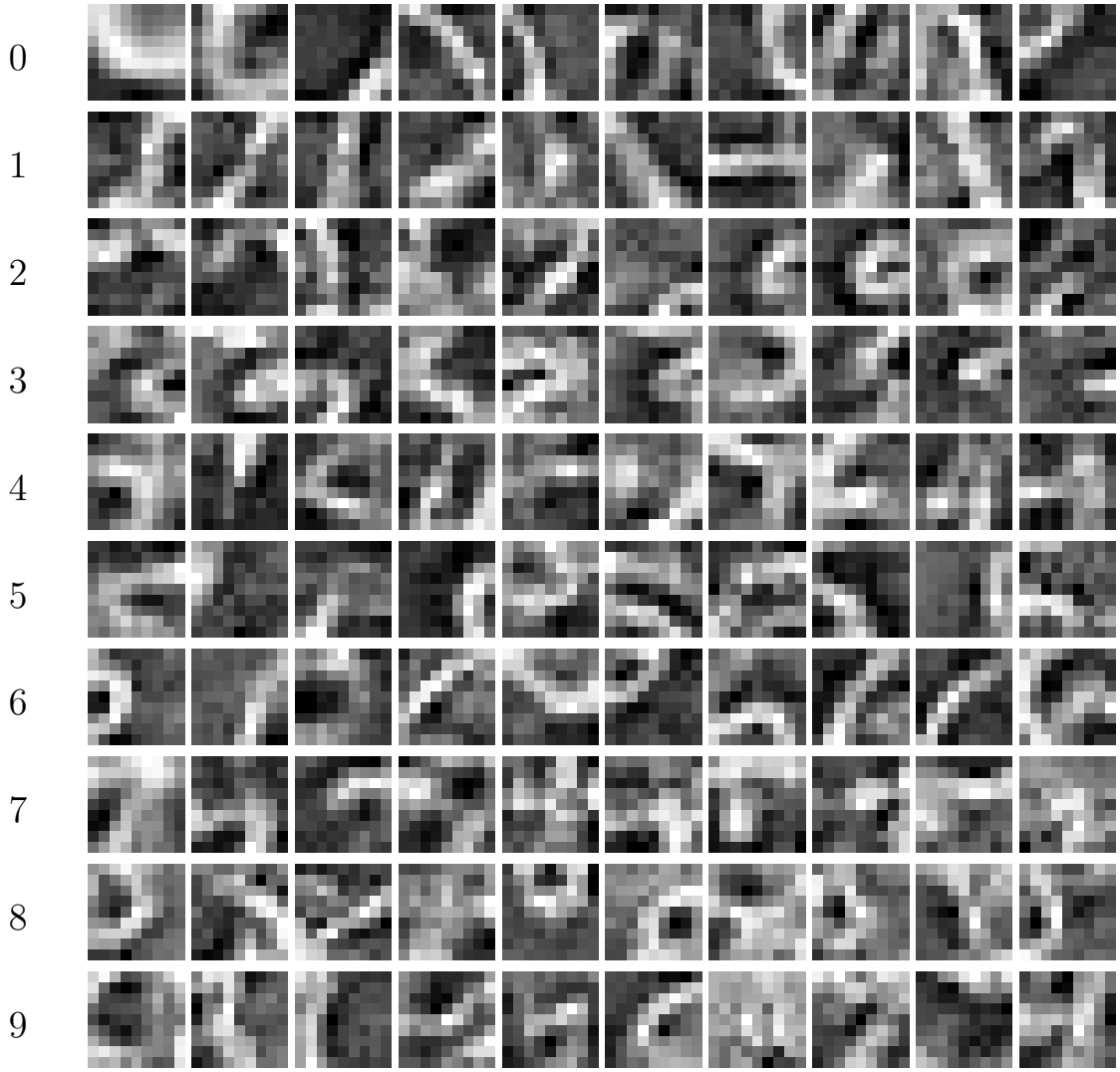

Figure 5.7: Illustration of filters, learned for the MNIST dataset. Filters within a row are from the same class, representing the digit of the respective class number. We can recognize different versions of parts from all digits in the corresponding class. Each filter was rescaled into the range of $[0 : 255]$ for better visualization. Therefore, bright pixel represent large weights and dark pixel represent small weights.

Learning method	Test Error Rate (%)
Convolutional net LeNet-1 (LeCun, Jackel, et al., 1995)	1.7
Convolutional net LeNet-4 (LeCun, Jackel, et al., 1995)	1.1
Convolutional net LeNet-5 (LeCun, Jackel, et al., 1995)	0.95
Large conv. net, unsup pretraining (Jarrett et al., 2009)	0.53
Large conv. net, unsup features (Poultney et al., 2006)	0.60
Convolutional SPN	1.66
State-of-the-art:	
Wan et al. (2013)	0.21

Table 5.1: Reported error rates for convolutional neural networks (Top) and state-of-the-art error rate (Bottom) on the MNIST dataset

The learned filters show clear structures of the respective class and clearly represent actual parts of the corresponding digit. For example we can recognize different version of the arc, defining the upper half of a six in class six. Another nice example is shown by the filters of class three. There we can clearly recognize different versions of the two meeting arcs of a three.

State-of-the-art, reported for this dataset, is an error rate of 0.21%, achieved by Wan et al. (2013). Classification errors on the test set, published for convolutional neural networks directly on top of rgb data, are in range of 1.7% and 0.53% (LeCun and Cortes, 2010).

Therefore, when compared our results are more at the bottom line of results for convolutional nets. Probably this is due to incomplete nodes within the network, which causes nodes to undercount marginals. Additionally the degenerated 1×1 convolutions within the network lead to worse results since they are not capable to learn patterns of higher order.

However, the results clearly show that the convolutional Sum-Product Networks are working.

5.3 CIFAR-10

The CIFAR-10 dataset (Krizhevsky, 2009), is an established image classification benchmark, which consists of 60000 color images in 10 classes. All images are down-sampled to rgb images of size 32×32 and labeled by hand. Thy are divided in a training set of size 50000 and a test set of 10000 images. Each image contains one object of the 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship or truck.

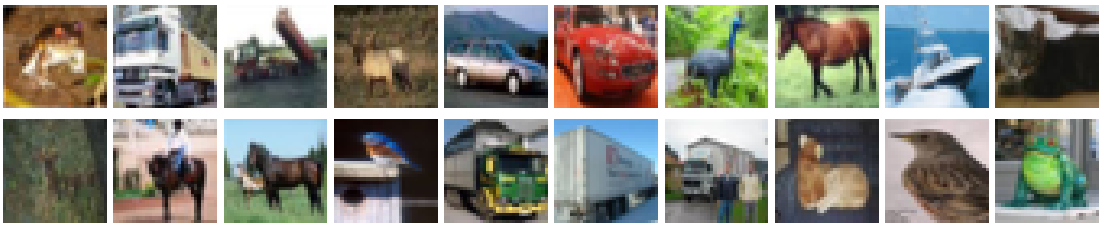

Figure 5.8: Example images from CIFAR-10

5.3.1 CIFAR-10 Experiment

In order to optimize the classification error for this dataset, we experimented with different parameters. However, we found that parameters similar to the parameter set from the experiment on MNIST achieved best results.

Therefore, the network has again one max node as root, which is followed by sum layer only. Parameter within the experiment were parts $P = 100$, a batch size of $b = 256$ and $\epsilon = 10^{-}8$. Further, we chose $N = 2$ children for every sum / max node, and a stride $S_l = 2$ for sum layer. Max-pooling was performed with a filter size of 2×2 and stride $S_p = 2$. Finally, within the convolutional layer we chose stride $S_c = 4$ and a filter size of 9×9. Each image was padded with one additional pixel of value zero in order to avoid effects at the border of the images within the pooling operation.

Given these parameters, we found the initial learn rates by gridsearch. Resulting in initial learn rate $\eta = 0.00001334$ and learn rate factor for the convolutional layer of $\eta_f = 101$.

The parameter above yield an SPN with five layer and eight hundred filters within the convolutional layer of each class.

The network was trained with these parameters using Algorithm 5. When the learn rate did not improve at least $f = 2\%$ within $it = 25$ epochs, the learn rate was multiplied by $f = 0.5$. This procedure was repeated nine times.

Before training the images were rescaled to range $[0 : 1.0]$.

Results

Within the experiments on CIFAR-10 the SPN achieved a classification error of approximately 7.08% on the training dataset and 46.71% on the test dataset. In Figure 5.9 we can observe the impact of learn rate changes on both, classification error and SPN error. The change of the learn rate results in a step on these learning curves. While the classification error on the train dataset is almost converged at approximately epoch 750, the error on the training dataset is still decreasing.

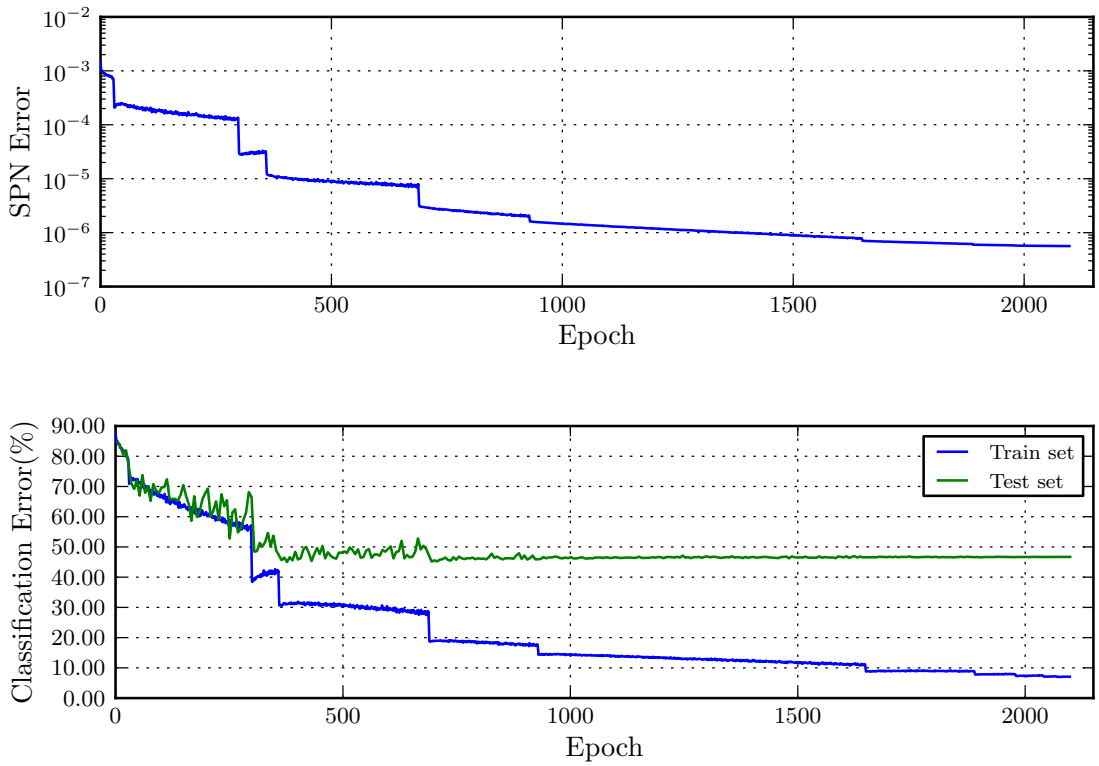

Figure 5.9: Development of SPN error (top) and classification error on train and test set (bottom) of CIFAR-10. The classification error on the test set did not improve anymore after approximately 750 epochs.

In Figure 5.10 we illustrated some learned filters. For the visualization, each filter was normalized separately to values in between 0 and 255. Structure is clearly observable in some of of the filters.. For example class six, representing the class “frog”, contains filters which look quite similar to Gabor filter.

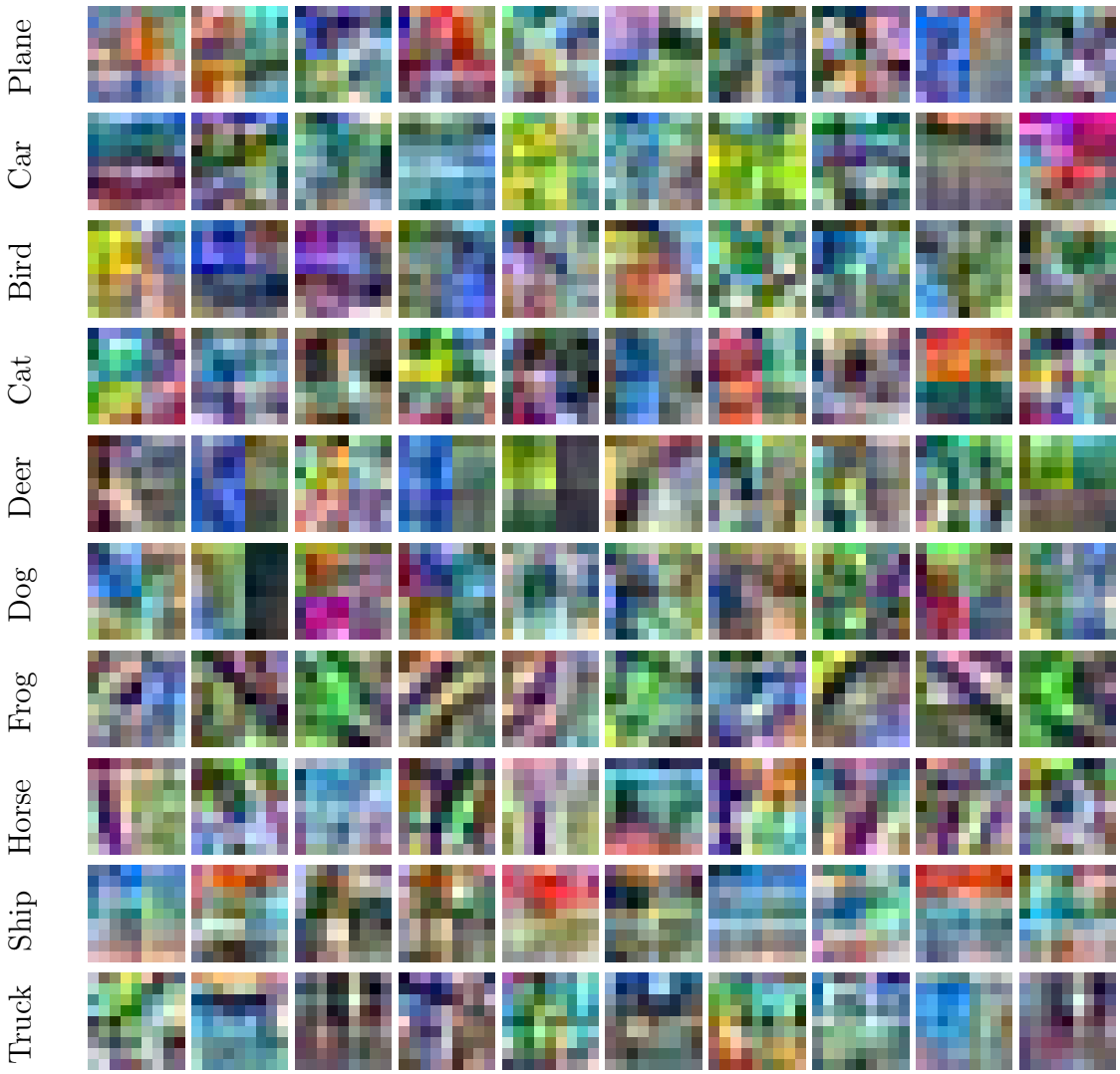

Figure 5.10: Illustration of filters, learned for the CIFAR dataset. The object contained within the class is given on the left of the corresponding row. Some structures like the edge filter from class six show recognizable structures.

5.3.2 Discussion

Within the experiments, a classification error rate of 46.71% was achieved on the test dataset. Gens and Domingos (2012) achieved a significant lower classification error of 16.04%. However, Coates and A. Ng (2011) showed, how crucial the choice of features is. They used only a simple MLP, with only one hidden layer and achieved state-of-the-art error rates of 18.5% on the test dataset of CIFAR-10, by careful choice of learned features. Gens and Domingos (2012) build their SPN on top of the same learned features and achieved a classification error, which is only 1.46% lower. Therefore, the question arises how other, more complex classifiers would perform on the same feature basis.

Section 5.3.2 provides an overview of current results on CIFAR-10. We found that the SPN implementation within this thesis clearly achieved worse results, even when compared to results, reported on rgb data only, as G. E. Hinton, Srivastava, et al. (2012). Therefore, the reasons for the results are probably the already mentioned flaws within the architecture, causing undercounting of marginals and prohibiting the network from learning features of higher order. When comparing the result to Gens and Domingos (2012), the choice of features certainly is another reason.

Krizhevsky and G. Hinton (2009) reported one of the first results with convolutional networks on the CIFAR-10 dataset. The result we achieved is slightly better than his reported error rate. Therefore, the result for this dataset again is in range of results, reported for CNNs.

We already pointed out the flaw within the architecture, used for the SPN. Additionally, the results presented within this thesis are within the range of results, reported for CNNS. Therefore, it is likely that convolutional SPNs can compete with Convolutional Neural Networks, when non-degenerated convolutions are provided within the SPN.

Learning method	Test Error Rate (%)
Krizhevsky and G. Hinton (2009)	58.87
Gens and Domingos (2012)	16.04
Convolutional SPN	46.71
Krizhevsky (2010b)	21.1
Coates and A. Ng (2011)	18.5
G. E. Hinton, Srivastava, et al. (2012)	16.6
State-of-the-art:	
Lin et al. (2013)	8.8

Table 5.2: CIFAR-10 error rates on raw data (top) and state-of-the-art error rates (bottom)

5.4 Approach: Fully Connected SPNs

Within the experiments we found that the architecture used has a capacity which is too small and therefore loses information about locations of features. This problem could be solved by using fully connected layers within the network, which would yield a valid SPN.

With a fully connected SPN on top, using just one layer of convolutions would cause a large number of weights for every sum node succeeding the convolutional layer. Therefore, using several layers of convolutions, before training the SPN on top, would allow to learn features of higher order. This would significantly reduce the number of weights for a sum node and therefore could provide better results. Additionally it is promising to experiment with non-linear units in between the convolutional layers.

In this case an efficient implementation of a fully connected sum layer would be possible through altered matrix multiplications.

The convolutional layer produces four dimensional outputs, which consists out of the number of maps N, horizontal and vertical dimensions of a map X and Y and finally the batch size B. By concatenation of the first three dimensions, we would get a two dimensional input to the SPN layer. Let $D \in (N \times X \times Y)$ be this concatenation. Then, we have input $I \in D \times B$.

The forward propagation for a layer with M fully connected sum nodes can simply be calculated through the matrix multiplication of weights $W \in M \times D$ and I, resulting in output $A \in M \times B$:

Let $a \in A$ be the result of the forward propagation, $s \in S$ the original input, weights $w \in W$, $\delta \in \Delta$ be the derivative for the network so far. Additionally, let $d \in D$ be an element of the first dimension of input I, and $b \in B$ be a batch within

input I. Then, derivatives for an element of the input of a fully connected layer is given by:

$$\frac{\partial A}{\partial i} = \sum_{m \in M} \delta_{m,b} \cdot \exp(s_{d,b} + w_{m,d} - a_{m,b}) \tag{5.1}$$

the derivative for a weight is described by

$$\frac{\partial A}{\partial w} = \sum_{b \in B} \delta_{m,b} \cdot \exp(s_{d,b} + w_{m,d} - a_{m,b}) \tag{5.2}$$

The sum layer can easily be calculated through exploitation of two matrix multiplications, which are already implemented efficiently on GPU. The derivative for input i can be calculated by exploitation of the matrix product from the transposed weight matrix W^T and Δ:

$$W^T \cdot \Delta = \sum_{m \in M} w_{i,m} \cdot \delta_{m,b} \tag{5.3}$$

by insertion of the formula for $\frac{\partial A}{\partial i}$, we get:

$$\frac{\partial A}{\partial i} = \sum_{m \in M} \delta_{m,b} \cdot \exp(w_{d,m} + s_{d,b} - a_{m,b}) \tag{5.4}$$

Further derivatives of weights can be calculated by exploitation of the matrix product from A and the transposed input matrix I:

$$A \cdot I^T = \sum_{b \in B} a_{m,b} \cdot s_{b,d} \tag{5.5}$$

by insertion of the formula for $\frac{\partial A}{\partial w}$, we get:

$$\frac{\partial A}{\partial w} = \sum_{b \in \in B} \delta_{m,b} \cdot \exp(w_{d,m} + s_{b,d} - a_{m,b}) \tag{5.6}$$

The implementation of max-layers can be achieved easily by exploitation of the same matrix multiplication and replacement of the function. We implemented the functions described above, but unfortunately were not able to experiment with this fully connected architecture due to time constraints.

6 Conclusion

In this master's thesis we combined the two successful deep learning approaches of discriminative Sum-Product Networks and Convolutional Neural Networks. The result is a SPN, which is trained on top of convolutional layers, providing learning of image features online. Large datasets or images of high resolution demand a lot of computing power. Therefore, we provided an efficient implementation on GPU and CPU, which is working through computations in logspace.

Within the thesis we showed experimentally that filters within the convolutions are learned correctly and provide reasonable structures. This implies that the combination of the two approaches is working properly.

However, we discovered that the chosen architecture within the SPN, contrary to CNNs, does not preserve any information about the positions of filters, learned by the convolutions. By replacing the products within the SPN, which are implemented by pooling in logspace, with max-pooling operations, we preserved the information of locality. Unfortunately, the altered SPN does no longer ensure completeness for all nodes, which causes undercounted marginals.

We evaluated the convolutional SPN on the two established image classification benchmarks MNIST and CIFAR-10. Experimental results showed that the preserved local information is even more important completeness of the SPN. We achieve a classification error on the test dataset, of 1.66% on MNIST and 46.71% on CIFAR-10. Results reported for Convolutional Neural Networks, are within range of these results.

The architecture of the SPN can be altered in order to preserve local information and ensure completeness at the same time. One approach achieving this is an architecture with fully connected layers. Details for an efficient implementation of this architecture on GPU by exploitation of matrix multiplications, was described in Section 5.4. Unfortunately we were not able to experiment with this architecture due to time constraints. Convolutional SPNs with this alteration most likely will obtain much better results.

6.1 Future Work

As Future Work it would be promising to investigate Convolutional SPNs with a different architecture, which ensures completeness for all nodes and preserves local information of features.

Further, usage of a single GPU prohibits training on large images, because of the limited amount of Global Memory. Using multiple GPUs instead would avoid this problem. Discriminative SPNs are especially suited for computation on multiple GPUs, since each class can be computed separately. Only the root node has to synchronize results of the network.

Finally, finding an appropriate learn rate is crucial within learning of SPNs. For Neural Networks there are several more advanced gradient descent methods like Adagrad (Duchi et al., 2011), which adjust the learn rate for each unit within a network automatically. Transferring these methods to Sum-Product Networks might improve their results.

Bibliography

Bengio, Y. (Jan. 2009). "Learning Deep Architectures for AI". In: *Found. Trends Mach. Learn.* 2.1, pp. 1–127. ISSN: 1935-8237. URL: http://dx.doi.org/10.1561/2200000006 (cit. on p. 1).

Bilmes, J. (1997). *A Gentle Tutorial on the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models.* URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.613 (cit. on p. 4).

Casella, G. and E. I. George (1992). "Explaining the Gibbs sampler". In: *The American Statistician* 46.3, pp. 167–174 (cit. on p. 5).

Ciresan, D., A. Giusti, J. Schmidhuber, et al. (2012). "Deep neural networks segment neuronal membranes in electron microscopy images". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 2852–2860 (cit. on p. 15).

Cireşan, D., U. Meier, J. Masci, and J. Schmidhuber (2012). "Multi-column deep neural network for traffic sign classification". In: *Neural Networks* 32, pp. 333–338 (cit. on p. 15).

Ciresan, D., U. Meier, and J. Schmidhuber (2012a). "Multi-column deep neural networks for image classification". In: *Computer Vision and Pattern Recognition (CVPR)*, pp. 3642–3649 (cit. on pp. 1, 19).

— (2012b). "Multi-column deep neural networks for image classification". In: *Computer Vision and Pattern Recognition (CVPR)*. IEEE, pp. 3642–3649 (cit. on p. 15).

Coates, A. and A. Ng (June 2011). "The Importance of Encoding Versus Training with Sparse Coding and Vector Quantization". In: *International Conference on Machine Learning (ICML)*. Ed. by L. Getoor and T. Scheffer. ICML '11. Bellevue, Washington, USA: ACM, pp. 921–928. ISBN: 978-1-4503-0619-5 (cit. on pp. 47, 48).

Coates, A., A. Y. Ng, and H. Lee (2011). "An analysis of single-layer networks in unsupervised feature learning". In: *International Conference on Artificial Intelligence and Statistics*, pp. 215–223 (cit. on p. 16).

Darwiche, A. (2003). "A differential approach to inference in Bayesian networks". In: *Journal of the ACM (JACM)* 50.3, pp. 280–305 (cit. on p. 5).

Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). "Maximum likelihood from incomplete data via the EM algorithm". In: *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 1–38 (cit. on p. 4).

Dennis, A. and D. Ventura (2012). "Learning the architecture of sum-product networks using clustering on variables". In: *Advances in Neural Information Processing Systems*, pp. 2042–2050 (cit. on pp. 6, 17).

Duchi, J., E. Hazan, and Y. Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization". In: *The Journal of Machine Learning Research* 12, pp. 2121–2159 (cit. on p. 52).

Egmont-Petersen, M., D. de Ridder, and H. Handels (2002). "Image processing with neural networksa review". In: *Pattern Recognition* 35.10, pp. 2279 –2301. ISSN: 0031-3203. URL: http://www.sciencedirect.com/science/article/pii/S0031320301001789 (cit. on p. 15).

Erhan, D., Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio (Mar. 2010). "Why Does Unsupervised Pre-training Help Deep Learning?" In: *J. Mach. Learn. Res.* 11, pp. 625–660. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1756006.1756025 (cit. on p. 16).

Farabet, C., C. Couprie, L. Najman, and Y. LeCun (2013). "Learning Hierarchical Features for Scene Labeling". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8, pp. 1915–1929. ISSN: 0162-8828 (cit. on pp. 1, 19).

Frey, B. J. and N. Jojic (2005). "A comparison of algorithms for inference and learning in probabilistic graphical models". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 27.9, pp. 1392–1416 (cit. on p. 17).

Fukushima, K. (1980). "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4, pp. 193–202 (cit. on p. 3).

Gens, R. and P. Domingos (2012). "Discriminative learning of sum-product networks". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 3248–3256 (cit. on pp. v, 1, 6–9, 17, 19–21, 28, 37, 47, 48).

— (2013). "Learning the Structure of Sum-Product Networks". In: *International Conference on Machine Learning (ICML)*. Vol. 28. W&CP (cit. on pp. 6, 17).

Guo, H. and W. Hsu (2002). "A survey of algorithms for real-time Bayesian network inference". In: *AAAI/KDD/UAI02 Joint Workshop on Real-Time Decision Support and Diagnosis Systems*. Edmonton, Canada (cit. on p. 17).

Hadsell, R., A. Erkan, P. Sermanet, M. Scoffier, U. Muller, and Y. LeCun (2008). "Deep belief net learning in a long-range vision system for autonomous off-road driving". In: *Intelligent Robots and Systems (IROS)*. IEEE, pp. 628–633 (cit. on p. 1).

Höft, N. (2014). *Bildsegmentation in Objekt-Klassen mit Konvolutionalen Neuronalen Netzen*. Undergraduate Bachelor Thesis (cit. on p. 3).

Hamel, P. and D. Eck (2010). "Learning Features from Music Audio with Deep Belief Networks." In: *ISMIR*. Utrecht, The Netherlands, pp. 339–344 (cit. on pp. 1, 16).

Hannes Schulz, A. M. (2013). *cuv library*. URL: http://www.ais.uni-bonn.de/deep_learning/doc/html/index.html (cit. on p. 21).

Hinton, G. E., S. Osindero, and Y.-W. Teh (2006). "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7, pp. 1527–1554 (cit. on p. 16).

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2012). "Improving neural networks by preventing co-adaptation of feature detectors". In: *CoRR* abs/1207.0580. URL: `http://dblp.uni-trier.de/db/journals/corr/corr1207.html#abs-1207-0580` (cit. on pp. 47, 48).

Huang, F. J. and Y. LeCun (2006). "Large-scale learning with svm and convolutional for generic object categorization". In: *Computer Vision and Pattern Recognition (CVPR)*. Vol. 1. IEEE, pp. 284–291 (cit. on pp. 1, 15).

Hubel, D. H. and T. N. Wiesel (1968). "Receptive Fields and Functional Architecture of Monkey Striate Cortex". In: *Journal of Physiology (London)* 195, pp. 215–243 (cit. on p. 1).

Jarrett, K., K. Kavukcuoglu, M. Ranzato, and Y. LeCun (Sept. 2009). "What is the best multi-stage architecture for object recognition?" In: *International Conference on Computer Vision (ICCV)*, pp. 2146–2153 (cit. on p. 43).

Jordan, M. I. (2004). "Graphical models". In: *Statistical Science*, pp. 140–155 (cit. on p. 17).

Krizhevsky, A. (2009). *Learning multiple layers of features from tiny images*. Tech. rep. (cit. on p. 43).

— (2010a). "Convolutional deep belief networks on cifar-10". In: *Unpublished manuscript* (cit. on p. 15).

— (2010b). *Convolutional deep belief networks on cifar-10* (cit. on p. 48).

Krizhevsky, A. and G. Hinton (2009). "Learning multiple layers of features from tiny images". In: *Computer Science Department, University of Toronto, Tech. Rep* (cit. on pp. 47, 48).

Krizhevsky, A., I. Sutskever, and G. Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 1106–1114 (cit. on pp. 1, 15).

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems (NIPS)*. Ed. by F. Pereira, C. Burges, L. Bottou, and K. Weinberger. Curran Associates, Inc., pp. 1097–1105. URL: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf` (cit. on pp. 1, 19).

Kruizinga, P., N. Petkov, and S. E. Grigorescu (2002). "Comparison of texture features based on gabor filters". In: *IEEE Transactions on Image Processing* 11, pp. 1160–1167 (cit. on p. 9).

Larochelle, H., Y. Bengio, J. Louradour, and P. Lamblin (2009). "Exploring strategies for training deep neural networks". In: *The Journal of Machine Learning Research* 10, pp. 1–40 (cit. on p. 16).

LeCun, Y. and Y. Bengio (1995). "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361 (cit. on pp. 1, 3).

LeCun, Y. and C. Cortes (2010). *MNIST handwritten digit database*. URL: `http://yann.lecun.com/exdb/mnist/` (cit. on pp. 39, 43).

LeCun, Y., L. Jackel, L. Bottou, A. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Müller, E. Säckinger, P. Simard, and V. Vapnik (1995). "Comparison of Learning Algorithms for Handwritten Digit Recognition". In: *Artificial Neural Networks and Machine Learning (ICANN)*, pp. 53–60 (cit. on p. 43).

Lee, H., C. Ekanadham, and A. Ng (2007). "Sparse deep belief net model for visual area V2". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 873–880 (cit. on p. 16).

Lee, H., R. Grosse, R. Ranganath, and A. Y. Ng (2011). "Unsupervised learning of hierarchical representations with convolutional deep belief networks". In: *Communications of the ACM* 54.10, pp. 95–103 (cit. on pp. 1, 16).

Le, Q. V., M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng (2011). "Building high-level features using large scale unsupervised learning". In: *arXiv preprint arXiv:1112.6209* (cit. on p. 16).

Lin, M., Q. Chen, and S. Yan (2013). "Network In Network". In: *CoRR* abs/1312.4400 (cit. on p. 48).

Nair, V. and G. E. Hinton (2009). "3D object recognition with deep belief nets". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 1339–1347 (cit. on p. 16).

NVIDIA (2014a). *Cuda Programming guide*. URL: `http://docs.nvidia.com/cuda/cuda-c-programming-guide` (cit. on pp. 10–12).

— (2014b). *Fermi Compute Architecture Whitepaper*. URL: `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf` (cit. on p. 14).

— (2014c). *Kepler Compute Architecture Whitepaper*. URL: `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf` (cit. on pp. 11, 14).

Pearl, J. (2011). "Bayesian networks". In: URL: `http://escholarship.org/uc/item/53n4f34m` (cit. on p. 17).

Peharz, R., B. C. Geiger, and F. Pernkopf (2013). "Greedy Part-Wise Learning of Sum-Product Networks". In: *Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 612–627 (cit. on pp. 6, 17).

Poon, H. and P. Domingos (2011). "Sum-product networks: A new deep architecture". In: *International Conference on Computer Vision (ICCV)*. IEEE, pp. 689–690 (cit. on pp. 1, 2, 5–7, 17, 19).

Poultney, C., S. Chopra, and Y. Lecun (2006). "Efficient learning of sparse representations with an energy-based model". In: *Advances in Neural Information Processing Systems (NIPS)*. MIT Press (cit. on p. 43).

Raina, R., A. Madhavan, and A. Y. Ng (2009). "Large-scale deep unsupervised learning using graphics processors." In: *International Conference on Machine Learning (ICML)*. Vol. 9, pp. 873–880 (cit. on p. 16).

Ramirez, I., P. Sprechmann, and G. Sapiro (2010). "Classification and clustering via dictionary learning with structured incoherence and shared features". In: *Computer Vision and Pattern Recognition (CVPR)*. IEEE, pp. 3501–3508 (cit. on p. 16).

Roth, D. (1996). "On the hardness of approximate reasoning". In: *Artificial Intelligence* 82.1, pp. 273–302 (cit. on p. 1).

Schultz, H. (2013). *cuvnet, a framework for neural nets on GPU* (cit. on p. 21).

Stallkamp, J., M. Schlipsing, J. Salmen, and C. Igel (2011). "The German traffic sign recognition benchmark: a multi-class classification competition". In: *International Joint Conference on Neural Networks (IJCNN)*. IEEE, pp. 1453–1460 (cit. on p. 15).

Vollmer, C., H.-M. Gross, and J. P. Eggert (2013). "Learning Features for Activity Recognition with Shift-Invariant Sparse Coding". In: *Artificial Neural Networks and Machine Learning (ICANN)*. Springer, pp. 367–374 (cit. on p. 16).

Wan, L., M. D. Zeiler, S. Zhang, Y. LeCun, and R. Fergus (2013). "Regularization of Neural Networks using DropConnect." In: *International Conference on Machine Learning (ICML)*. Vol. 28. JMLR Proceedings. JMLR.org, pp. 1058–1066. URL: `http://dblp.uni-trier.de/db/conf/icml/icml2013.html#WanZZLF13` (cit. on pp. 1, 19, 43).

Yang, J., G. Jiang, A. G. Hauptmann, and C.-W. Ngo (2007). "Evaluating bag-of-visual-words representations in scene classification". In: *Proceedings of the international workshop on Workshop on multimedia information retrieval*. ACM, pp. 197–206 (cit. on p. 16).