GPU Implementation of Restricted Boltzmann Machines with Application to View Classification in Sports Videos

by

Andreas Fred Bernitzke

Dipl. Ing. (FH), Hochschule Karlsruhe, 2009

A project

presented to Ryerson University

and Hochschule Karlsruhe

in partial fulfillment of the

requirements for the degree of

Master of Engineering

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2011

I hereby declare that I am the sole author of this project.

This project is subject to the regulations of the double degree program contract between Ryerson University and University of Applied Science Karlsruhe. Therefore, any restrictions and conditions implied by the contract apply to the report. In particular, this work may be published under different terms in accordance to the regulations of University of Applied Science Karlsruhe.

I authorize Ryerson University to lend this project to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this project by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Masterthesis ohne unzulässige fremde Hilfe selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Toronto, den 23.11.2010

GPU Implementation of Restricted Boltzmann Machines with Application to View Classification in Sports Videos

Master of Engineering 2011

Andreas Fred Bernitzke

Electrical and Computer Engineering

Ryerson University and Hochschule Karlsruhe

A framework for Restricted Boltzmann Machines (RBM) accelerated by the power of graphic card processors (GPU) is presented in this project report. The framework and GPU speedup is analyzed on the one hand side and applications to Network Intrusion Detection and View Classification on the other hand. The framework is developed using C++ and CUDA on a Windows platform, and a 40x speedup is achieved. The structure is versatile and application-domain independent, so the framework can be applied to other research areas than the presented as well. The KDD 1999 Classifier Cup Data is employed to assess the performance on network intrusion detection, where an Artificial Immune System (AIS) is outperformed by our RBM. Novel approaches to facilitate RBMs for bag of word techniques are presented in the context of view classification. Bag of visual words have demonstrated outstanding capabilities, so a fusion of the techniques is a reasonable next step.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Neural Networks have caught the interest of researchers for decades, especially for function approximation and classification without the need of an explicit model derivation. Many different structures have been developed, usually inspired by similarities in nature or biology. Although results are encouraging, the current state of research is far away from artificial intelligence. The cause is partly due to the limited computational power and the complexity of the proposed techniques. Based on constraints, two main strains of research can be distinguished. In the first category, the underlying system is modeled as exactly as possible at the expense of limited size and performance. Indeed, derivations of mathematical approximations of the real world process are in the focus of this research to understand and explain the underlying coherences. By contrast, other research tries to simplify the models to yield efficient algorithms with adequate similarity. Those models are trained to find solutions to challenging tasks, and thus prove their potential despite the simplification.

One of the most promising descendants of the latter approach, called *Restricted Boltzmann Machine* is considered in this report[1]. The Restricted Boltzmann Machine raised a great interest in the scientific computing community due to unique key attributes, most notable the unsupervised training technique and generative property, accompanied by very promising results in many different domains of research. A framework to employ this technique in research is developed in the scope of the work at hand. In the end, applications of the Restricted Boltzmann Machine to Network Intrusion Detection and View Classification are presented.

In general, a neural network will not lead automatically to superior results. In fact, several iterations are required to find the best parameter values during research, so training speed gains more attention due to the size of the training data and the number of repetitions. At the same time, newer CPU architectures no longer yield higher execution speed, but try to offer more computational performance

[1]This work is presented to Ryerson University and HS Karlsruhe as part of a double degree program. This work is called thesis in the nomenclature of HS Karlsruhe, while this is considered as Master's project in the curriculum of Ryerson University. Hence, the terms thesis and project may be used equivalent throughout the document.

by providing more cores to cope with multiple tasks. In fact, current CPU architectures devote more than 80 % of silicon to general speed improvement techniques like cache and out-of-order execution, but the limit of reasonable return of further investment in this direction has been reached, so adding more cores has become the favoured path. The problem however, is that software is not naturally optimized to use these cores. Thus, software will not run faster on newer hardware if it does not exploit the new dimension of parallelism, so a paradigm change into parallel computing is needed.

With this trend in mind, another recent technique is employed in this thesis. In recent PCs, the GPU has superseded the computational power of a CPU for certain tasks. The design of a GPU processor is very closely connected to the usual tasks in image processing to provide the realistic graphics in modern computer games in real time for instance. Even general purpose software can exploit this performance if the computation maps properly onto the new architecture. Nevertheless, a GPU consists of hundreds of cores, so the paradigm change is more incisive than one may expect with a CPU in mind, although the ideas are identical. Recently, GPU development has gathered momentum with new tools, reports of significant speedups and a growing community.

In general, this thesis examines two major aspects. First, the suitability of the graphic processor to a Restricted Boltzmann Machine is analysed. This aspect focuses on implementations of different parts of the training and the evaluation of the obtained speedups and limitations. On the other hand, the suitability of the Restricted Boltzmann Machine for different real world problems is investigated. Two databases from completely different domains are used within the report. The first use case is the KDD Cup classification challenge, so different attack types in network traffic must be identified. Thus, the Restricted Boltzmann Machine is used as a network intrusion detection tool. By contrast, the second application domain originates in video processing. Different sports and view types must be distinguished here. Several utilization techniques of a Restricted Boltzmann Machine are examined to solve the shortcomings of the k-nearest neighbour clustering technique used in previous research. Similar for both applications, the unsupervised discovery of features based on examples during learning plays a key role. Thus, an implicit model is generated with the training data. Unlike common neural network techniques, the training algorithm of a Restricted Boltzmann Machine does not focus on telling the classes apart, so more effort needs to spent to achieve decisions. Popular techniques to attain classification are highlighted in the introducing chapters and used for the different tasks in later sections.

Three different databases are used in the scope of this thesis. The first one is the KDD Cup 1999 classifier challenge database [3]. The suitability for network intrusion detection has to be proved by detecting different attack patterns in features extracted from real network traffic. A database for video/genre detection has been chosen as second use case. The data was compiled for previous work at Ryerson University and is not publically available. Records of sport videos aired on television have been collected and key frames have been extracted and manually labelled. This database constitutes the second application domain presented in this project. Finally, the MNIST character database is used for several examples [2]. In fact, the first major success of the Restricted Boltzmann Machine was reported on this database and is used by many researches to demonstrate their extensions. Likewise, this dataset is used to evaluate and illustrate the impact of enhancements due to the characteristics of the database, most notably binary input values and short training times combined with many published results.

Contributions of this thesis

As noted before, the emphasis of this thesis is on the GPU based acceleration of Restricted Boltzmann Machines and their application to large-scale tasks. Two examples for both directions are presented, so the major contributions of this thesis are the following aspects:

- Implementation of back-propagation style fine-tuning on a GPU
- Application of the Restricted Boltzmann Machine to network intrusion detection
- Implementation of Contrastive Divergence learning on a GPU in the scope of a versatile and comprehensive framework
- Application of the Restricted Boltzmann Machine to View Classification

Herein, the application of a Restricted Boltzmann Machine to network intrusion detection is a novel contribution. Furthermore, some novel approaches to employ the Restricted Boltzmann Machine to view classification are developed, too.

Structure of this thesis

The structure of this thesis is as follows: The Restricted Boltzmann Machine is introduced in the **second chapter**. The original proposal and extensions to different input data spaces are presented. Another key issue is the construction of Deep Belief Networks based on Restricted Boltzmann Machines and the derivation of class decisions.

The current state of GPU computing is presented in the **third chapter**. The emphasis is on clarifying the functionality of the underlying hardware and the support tools. In fact, a GPU based implementation is very sensitive to both, software and hardware design. Thus, an introduction into both aspects is provided to reason the design presented later on.

The first application of Deep Belief Networks is presented in the **forth chapter**, namely the intrusion detection in networks like the internet. Two major aspects are tackled after an introduction to the database. First, the acceleration with GPU interaction is described. The performance on the classification task and a comprehensive comparison to other techniques constitutes the second part of the chapter.

Inspired by the results of the previous chapter, a native framework for Deep Belief Networks is presented in the **fifth chapter**. The framework is intended as general framework to solve any appropriate task. Hence, the focus resides on the framework itself and a comparison to other GPU based implementations.

The application of the framework to video view classification is presented in the **sixth chapter**. Previous work on this matter is outlined first, followed by the results obtained by using Deep Belief Networks. Thereby, some novel methods to combine Deep Belief Networks with Bag-of-visual-Word-techniques are introduced.

In the end, a summary of the topics addressed in the master thesis is given. Moreover, a conclusion is drawn and possible directions for further research are pointed out.

This page is blank intentionally.

Chapter 2

Deep Belief Networks

In machine learning, *Deep Belief Networks* (DBN) represent a class of stochastic, generative, multi–layer neural networks. In each layer, they capture correlations from the underlying layers and use a non-linear mapping to account for those correlations. Therefore, they can be distinguished from deterministic techniques like Support Vector Machines in terms of depth, as those are usually limited to a shallow structure with one input and one output layer.

By employing several layers, DBNs are able to detect higher order correlation. However, the learning of such networks is tricky due to the dependency across the layers. Common learning strategies like back propagation forego the power of capturing correlations in favour of a more direct path to the learning goal. Hereby, they are prone to several pitfalls like local minima.

Hinton et al. proposed a new architecture called *Restricted Boltzmann Machine* (RBM) in conjunction with a novel greedy layer–wise learning algorithm [22, 23]. While an RBM itself is a shallow structure, it can be easily extended into a multi–layer DBN. Indeed, the RBM facilitates an efficient learning algorithm, which can exploit the architectural benefits of parallel hardware. Furthermore, it has been shown to be an efficient way of extracting key features out of input data in an unsupervised manner. As such, it has been proven useful in many different domains of research, most notably character recognition [23, 24], texture synthesis [57], semantic hashing [53, 55] and object tracking [42].

The remainder of this chapter is organized as follows. First, we outline the binary RBM in detail because it has been proposed first and therefore, it is the basis for many derivatives explained later on. Next, we describe how to employ RBMs as building blocks for DBNs. We continue with some extensions into other input spaces, especially real-valued data. This extension is important because most databases employ continuous valued elements. The chapter is concluded with a short summary.

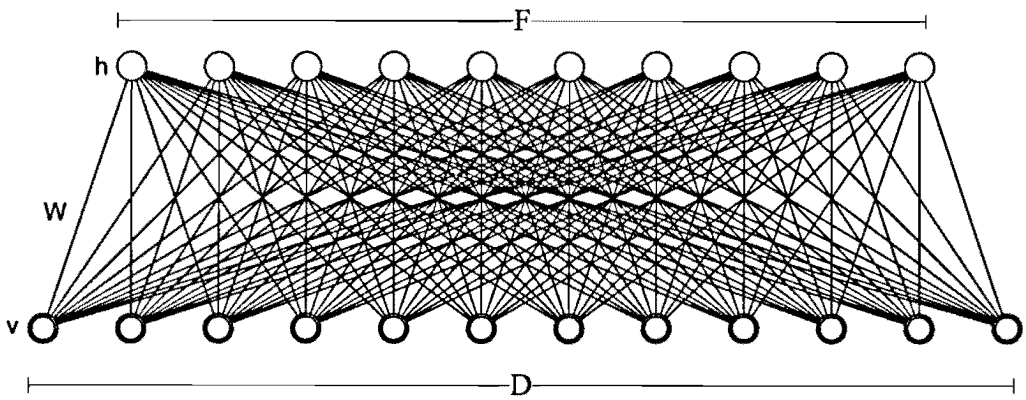


Figure 2.1: Structure of an RBM

2.1 Restricted Boltzmann Machine

The RBM is a particular type of such a probabilistic neural network. The structure can be visualized as an undirected bipartite graph with one type of nodes forming the visible layer v_i and the other kind the hidden layer h_j as depicted in Figure 2.1. The hidden and visible layer nodes are in binary states $v \in \{1,0\}^D$ and $h \in \{1,0\}^F$, where D and F denote the sizes of the visible layer and the hidden layer respectively. An important aspect resides in the symmetric weights between the layers, allowing the network to be used for discrimination and reconstruction of data. In fact, the reconstruction property leads to a new training algorithm called *Contrastive Divergence* (CD).

The underlying idea roots in an analogy with statistical mechanics, describing the energy of a particle distribution on the atomic scale. The composition on a large scale is determined by simple rules in the atomic domain yielding a minimization of the configuration energy. Due to this relationship with the physicist, the network architecture is called Boltzmann machine, although several other names have been used as well. Moreover, recent research also suggests similarities with human perception [30].

The essential definition resides in the formulation of the energy function. For a binary RBM as outlined above, the energy is defined to be

$$E(v,h,\theta) = -\sum_{i=1}^{D}(a_i v_i) - \sum_{j=1}^{F}(b_j h_j) - \sum_{i=1}^{D}\sum_{j=1}^{F}(v_i h_j w_{ij})\,. \tag{2.1}$$

In this definition, $\theta = \{W, b, a\}$ denotes the model parameters of the RBM, namely the connection weights between hidden and visible layer W, the biases of the visible units a and the biases of the hidden units b.

The joint probability for a certain state is derived from the energy function (2.1) to be the following:

$$P(v,h;\theta) = \frac{1}{Z(\theta)} \exp\left(-E(v,h,\theta)\right), \tag{2.2}$$
$$Z(\theta) = \sum_v \sum_h \exp\left(-E(v,h,\theta)\right).$$

The function $Z(\theta)$ is called partition function and reflects the total energy of all possible states in the current model. However, it is hardly feasible to compute the partition function exactly except for examples in the toy domain due to the immense number of states in the hidden and visible units. In fact, the computational demands grow exponentially with the size of the layers. Even approximations to the real value emerge as hardly computable [53].

The conditional probabilities given the opposite layer are needed for the learning algorithm, so we derive the formulas here for the sake of a self-contained presentation. In order to derive the conditional distribution, we need to outline the probability for a layer configuration first. Here, we exemplify the visible layer. The probability for a visible layer configuration is the energy ratio between all configurations containing the visible vector and all possible configurations. Hence, we need to consider the sum of energy over all possible configurations in the hidden layer

$$\begin{aligned} P(v;\theta) &= \frac{1}{Z(\theta)} \sum_h \exp\left(-E(v,h;\theta)\right) \\ &= \frac{1}{Z(\theta)} \sum_h \exp\left(\sum_{i=1}^{D}(a_i v_i) + \sum_{j=1}^{F}(b_j h_j) + \sum_{i=1}^{D}\sum_{j=1}^{F}(v_i h_j w_{ij})\right) \\ &= \frac{1}{Z(\theta)} \exp\left(\sum_{i=1}^{D}(a_i v_i)\right) \prod_{j=1}^{F} \sum_{h_j \in \{0,1\}} \exp\left(b_j h_j + \sum_{i=1}^{D}(v_i h_j w_{ij})\right) \\ &= \frac{1}{Z(\theta)} \exp\left(\sum_{i=1}^{D}(a_i v_i)\right) \prod_{j=1}^{F}\left(1 + \exp\left(b_j + \sum_{i=1}^{D}(v_i w_{ij})\right)\right). \end{aligned} \tag{2.3}$$

With those intermediate results, we can derive the conditional distributions

$$\begin{aligned} P(h|v;\theta) &= \frac{P(v,h;\theta)}{P(v;\theta)} \\ &= \frac{\frac{1}{Z(\theta)} \exp\left(-E(v,h;\theta)\right)}{\frac{1}{Z(\theta)} \exp\left(\sum_{i=1}^{D}(a_i v_i)\right) \prod_{j=1}^{F}\left(1 + \exp\left(b_j + \sum_{i=1}^{D}(v_i w_{ij})\right)\right)} \\ &= \frac{\exp\left(\sum_{i=1}^{D}(a_i v_i)\right) \exp\left(\sum_{j=1}^{F}(b_j h_j) + \sum_{j=1}^{F} h_j \sum_{i=1}^{D}(v_i w_{ij})\right)}{\exp\left(\sum_{i=1}^{D}(a_i v_i)\right) \prod_{j=1}^{F}\left(1 + \exp\left(b_j + \sum_{i=1}^{D}(v_i w_{ij})\right)\right)} \\ &= \prod_{j=1}^{F} \frac{\exp\left(h_j\left(b_j + \sum_{i=1}^{D}(v_i w_{ij})\right)\right)}{1 + \exp\left(b_j + \sum_{i=1}^{D}(v_i w_{ij})\right)}. \end{aligned}$$

The conditional probability $P(v|h)$ can be deduced likewise. Therefore, the probability of a certain layer configuration is simply the product of the probabilities of all neurons within that layer:

$$P(h|v) = \prod_j p(h_j|v),$$
$$P(v|h) = \prod_i p(v_i|h).$$

This leads to the frequently quoted formulas for the conditional probability of a single neuron

$$p(h_j = 1|v) = \varphi\left(b_j + \sum_i v_i w_{ij}\right),$$
$$p(v_i = 1|h) = \varphi\left(a_i + \sum_j h_j w_{ij}\right),$$
$$\varphi(x) = \frac{1}{1 + \exp(-x)}.$$

Logistic, sigmoid or activation function are common names for $\varphi(x)$. It is notable that the intractable partition function is not needed to evaluate the conditional probabilities. Hence, the conditional probability of a neuron is computable, although the joint probability is intractable.

Equipped with the probabilities with certain states, the way is paved for considering the learning of an RBM. It is reasonable to use stochastic optimization goals due the stochastic nature of an RBM, so maximum-likelihood training is employed. However, it is beneficial to simplify using the log-likelihood instead with regard to the structure of an RBM. Therefore, the probability for a visible state is the following:

$$\log(P(v;\theta)) = \log\left(\sum_h \exp(-E(v,h;\theta))\right) - \log\left(\sum_v \sum_h \exp(-E(v,h,\theta))\right), \tag{2.4}$$
$$\begin{aligned}\frac{\partial \log(P(v;\theta))}{\partial w_{ij}} &= \frac{\partial \log\left(\sum_h \exp(-E(v,h;\theta))\right)}{\partial w_{ij}} - \frac{\partial \log\left(\sum_v \sum_h \exp(-E(v,h,\theta))\right)}{\partial w_{ij}} \\ &= v_i P(h_j = 1|v) - P(v_i = 1|h_j = 1) \\ &= \mathrm{E}\langle vh\rangle_{\text{data}} - \mathrm{E}\langle vh\rangle_{\text{model}}.\end{aligned}$$

In analogy to Equation (2.2), the first term reflects the energy of all states with the given visible vector. The second term describes the total energy, so in a loose sense, it accounts for the predictions made by the given model. For a parameter update, we need to consider the derivative of Equation (2.4) with respect to the model parameters. The first term is easy to compute, because in the log-likelihood domain, it boils down to a sum over all hidden layer probabilities as shown in Equation (2.3). The second term involves the intractable partition function $\mathcal{Z}(\theta)$ and the derivate is practically infeasible to evaluate as well. Therefore, a Gibbs sampler is used to estimate the distribution.

Gibbs sampling is an iterative Markov Chain Monte Carlo based algorithm to approximate the joint distribution based on a conditional distribution. It has been introduced for stochastic simulation of atoms in equilibrium at the beginning of the computer era. The state of each unit is updated using the conditional probability over all the other units in each iteration. As outlined above, the conditional probability is easy to compute because the partition function is cancelled out. Furthermore, each iteration is highly parallel in the light of an RBM, because all units in one layer depend on the other layer only due to the special bipartite structure. Hence, the states of all units in a layer can be evaluated in parallel.

In fact, there are two major techniques to deploy Gibbs sampling in the RBM. In the first approach, the Gibbs sampler is initialized with random values. By running the Gibbs chain for several iterations, the distribution should converge towards the real distribution by the given model. This is commonly known as *Persistent Contrastive Divergence* (PCD) [59]. However, it may need a long time until the Gibbs sampler approaches the final distribution. Although some tweaks have been proposed to alleviate the need for a long-running sampler, this technique is used rather seldom in previous work.

The second approach applies another assumption to overcome the need for a long-running approximation. For each parameter update, the Gibbs sampler is initialized with a realization drawn from hidden layer probabilities given the current visible vector. Then, n iterations of Gibbs sampling are used to estimate the model distribution. This approach is called *Contrastive Divergence* (*CDn*) in the literature. Due to the fact that the initial values are assumed to be close to the real distribution of the model, only a few iterations are needed for a satisfactory estimation. However, those estimations may be correlated, especially for short iterations and low mixing rates of the Gibbs Sampler. Nevertheless, many research publications successfully employ *CD1* in very different domains. Hence, we used *CDn* for our project as well.

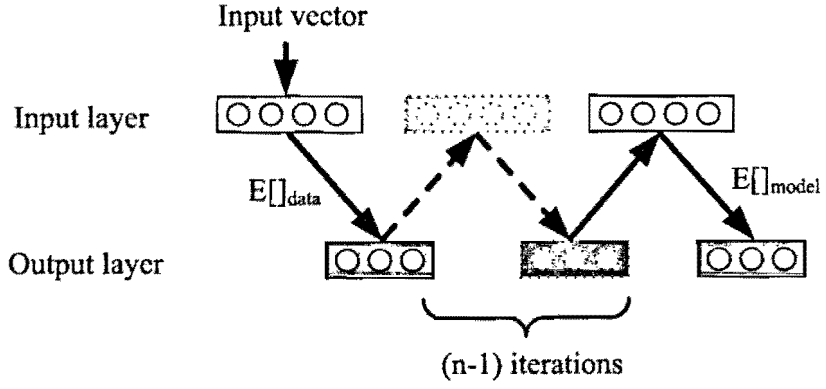


Figure 2.2: Training procedure for *Contrastive Divergence* with n iterations

The training procedure for CDn is illustrated in Figure 2.2. First, the visible units are initialized with the provided input data and the most likely state of the hidden units is calculated using the conditional probability. The correlation between the visible and hidden units $E\left\langle vh\right\rangle_{data}$ is computed based on the current state. Second, n iterations of Gibbs sampling are used to estimate the model probabilities. Each

Gibbs sampling is an iterative Markov Chain Monte Carlo based algorithm to approximate the joint distribution based on a conditional distribution. It has been introduced for stochastic simulation of atoms in equilibrium at the beginning of the computer era. The state of each unit is updated using the conditional probability over all the other units in each iteration. As outlined above, the conditional probability is easy to compute because the partition function is cancelled out. Furthermore, each iteration is highly parallel in the light of an RBM, because all units in one layer depend on the other layer only due to the special bipartite structure. Hence, the states of all units in a layer can be evaluated in parallel.

In fact, there are two major techniques to deploy Gibbs sampling in the RBM. In the first approach, the Gibbs sampler is initialized with random values. By running the Gibbs chain for several iterations, the distribution should converge towards the real distribution by the given model. This is commonly known as *Persistent Contrastive Divergence* (PCD) [59]. However, it may need a long time until the Gibbs sampler approaches the final distribution. Although some tweaks have been proposed to alleviate the need for a long-running sampler, this technique is used rather seldom in previous work.

The second approach applies another assumption to overcome the need for a long-running approximation. For each parameter update, the Gibbs sampler is initialized with a realization drawn from hidden layer probabilities given the current visible vector. Then, n iterations of Gibbs sampling are used to estimate the model distribution. This approach is called *Contrastive Divergence* (*CDn*) in the literature. Due to the fact that the initial values are assumed to be close to the real distribution of the model, only a few iterations are needed for a satisfactory estimation. However, those estimations may be correlated, especially for short iterations and low mixing rates of the Gibbs Sampler. Nevertheless, many research publications successfully employ *CD1* in very different domains. Hence, we used *CDn* for our project as well.

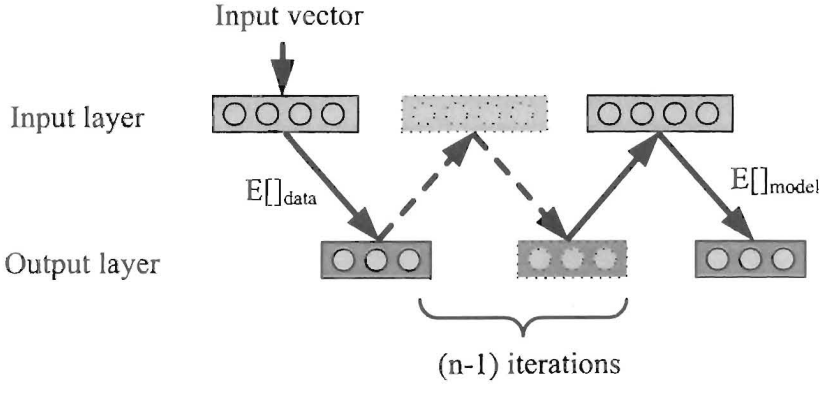


Figure 2.2: Training procedure for *Contrastive Divergence* with n iterations

The training procedure for CDn is illustrated in Figure 2.2. First, the visible units are initialized with the provided input data and the most likely state of the hidden units is calculated using the conditional probability. The correlation between the visible and hidden units $E\langle vh\rangle_{data}$ is computed based on the current state. Second, n iterations of Gibbs sampling are used to estimate the model probabilities. Each

step updates the visible units based on the hidden units, and then the hidden units based on the visible ones. Finally, the correlation between the updated visible and hidden units is considered as estimate for $E\langle vh\rangle_{model}$.

The main objective of the training procedure is to approximate log-maximum-likelihood learning. Indeed, it approximates the gradient of ML in favour of the absolute value. Most of the research has been focused on empirical observations due to the partition function. For example, CDn is considered as biased ML estimator in [16], but the bias turns out to be neglectable and a good approximation of ML learning is demonstrated using few iterations of Gibbs Sampling. A more theoretical elaboration with similar results can be found in [12]. Recently, it has been shown that Contrastive Divergence does not follow the gradient of any specific function and may not converge under some circumstances [58]. The objective of learning is even less definable taking extensions of the original RBM learning rule into account.

Despite the lack of an explicit learning goal, RBMs have successfully been applied to many different domains as outlined in the introduction. Furthermore, recent research indicates similarities with human perception in the visual cortex [30]. Therefore, the interest and research is increasing.

2.2 Forming DBNs based on RBMs

The training of a single two-layer network has been outlined so far, but deep structures are well-known to yield better results. For RBMs in particular, the higher layers capture higher order correlations in the input data. To illustrate the situation, consider an example of character recognition. If only a single layer is available, typical characters instances must be identified directly. By contrast, a DBN engages a more sophisticated processing. The characters may be decomposed into typical strokes in lower layers and recombined to characters in upper layers. Hence, each layer adds another level of abstraction and generalization to derive more sophisticated models in the top layer. This detour leads to a more robust

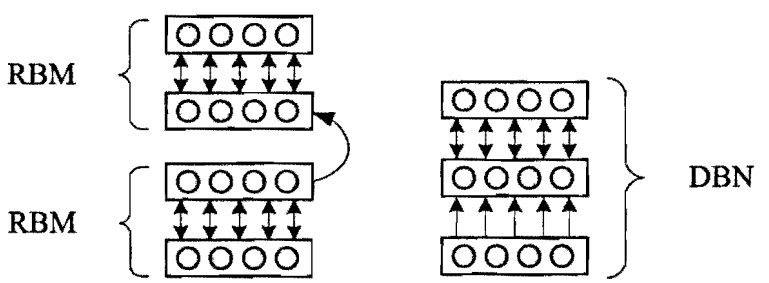


Figure 2.3: Extension of RBMs to DBNs. Most notably, the resulting DBN is directed, although the building blocks are undirected.

and flexible way to form a decision. Thus, a straight-forward way to build a DBN based on RBMs is demonstrated here.

The most common technique is to train a single layer and use the output probabilities of that RBM as input for the next one. The idea is shown in Figure 2.3. In this way, each additional RBM tries to extract higher order correlations of the underlying RBM, so the complexity of representable correlations ascends with each additional RBM. This leads to an iterative, layer–by–layer wise training of a DBN.

Quite surprisingly, the resulting DBN is no longer an undirected model, because the upper layers are trained after freezing the lower layer weights. This way, each RBM depends on the training results of the predecessors and cannot influence what to extract based on the current level of abstraction. However, the model is treated like an undirected model for illustrative purposes in several papers [23, 42]. This leads to conceivable demonstrations of learned features if the DBN is trained on image data.

2.3 DBNs as Classifiers

One of the major applications of neural networks is the classification of data. This may be influenced by back propagation learning, which needs an error signal to tune the weights. Such a signal is easy to obtain for a deterministic optimization goal like classification. In fact, we need a classifier to derive a certain view type or attack decision later on, so we outline the most common techniques here.

Most obviously, well-known classifier techniques like back propagation neural networks, support vector machines or logistic regression may be used to form a decision based on the output layer of the DBN. Thus, the DBN is used as a non-linear mapping to transform the input data into another presentation that is more useful for the classification task. Several successful applications with such a structure are reported in the literature [8, 23, 28]. However, the success is not guaranteed as the features learned by the DBN are trained in an unsupervised manner.

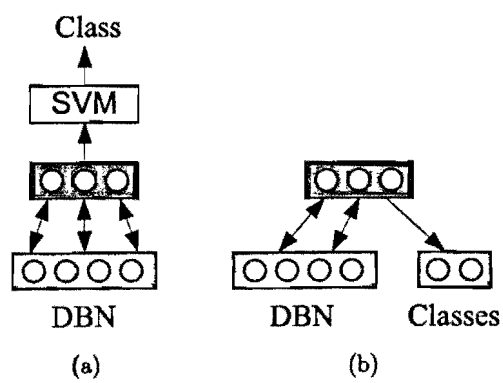


Figure 2.4: Different approaches to use DBNs as classifiers. Depicted is the top RBM of the DBN stack

A second approach uses the regenerative property of an RBM to assign a class to a test vector. This is depicted in Figure 2.4b. The labels are added to the input vector as additional input neurons, so the network is trained to regenerate the labels based correlations with on the captured features. For each presented test vector without labels, the network will provide probabilities for the labels as well. In our own research, we improved the performance of a back propagation based classifier if pre-trained in that manner.

Another possibility is to train an own DBN on each class and combine the results of the DBNs, i.e. by picking the most-likely class. Therefore, we want to evaluate $\arg\max_{\text{class}}(P_{\text{class}}(v)) = \arg\max_{\text{class}}\left(\frac{E_{\text{class}}(v)}{\mathcal{Z}_{\text{class}}}\right)$. The important point resides in the partition function $\mathcal{Z}_{\text{class}}$, which is different for each DBN and infeasible to compute. Hence, the probability cannot be derived easily. A possible solution is to train another RBM on the energies of the underlying class specific DBNs [41]. In practice, this approach is suitable for a very limited number for classes only, because the output of each class-dependent RBM must be fed into the classification RBM. Thus, the input layer size of the final RBM grows to infeasible values rather soon.

2.4 Extending to other input spaces

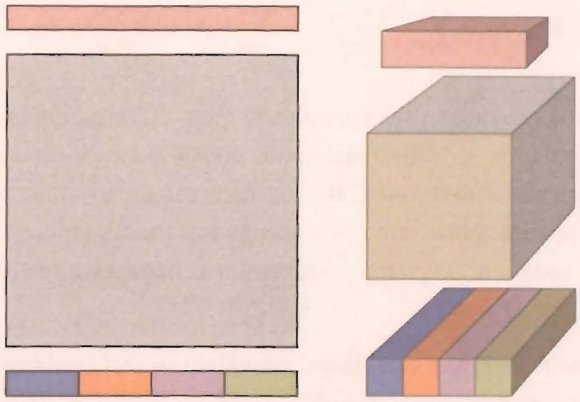
So far, we have considered the Bernoulli case with binary inputs and outputs only. In the light of image processing, this limits the application areas to preprocessed data like edge images. However, there are several options to extend the RBM to other input spaces and thereby generalize the eligibility of RBMs.

At first glance, it may seem straight forward to treat the continuous input data as probabilities, so only a rescaling step is necessary. Experiments unveil that this approach will not yield usable results. We tackle this experience in more detail in a later section, but continue with other approaches to overcome this shortage in the current section.

Obviously, quantized values are easy to feed into a binary RBM. Each input feature is encoded using n adjacent binary features. The learning and implementation are same as before, but the number of parameters is increased by a factor of $F(n-1)$ with F the size of the hidden layer. This straight way is illustrated in Figure 2.5a. It should be noted that this seemingly inefficient representation is needed in order to preserve the correlation structure. In our experience, a vector with first $\lfloor n * \overline{\text{value}} \rfloor$ units turned on yield best results, as illustrated in Table 2.1. Hereby, $\overline{\text{value}}$ denotes the input values normalized to fit into the range $[0, 1]$. In an illustrative sense, this can be regarded as binary histogram with n bins, with all the bins up to the current bin are activated. In terms of the idea, it may seem similar to the replicated softmax-model used for tabbed data [52]. However, only a single bin is activated due to the nature of the tabbed data. Our coding preserves the true distance in more detail. While quick to implement, the quantization may lead to infeasible training time in large scale problems. Additionally, the large number of additional weights can foster overfitting and stability problems.

Value	Representation		
[0 , 0.25)	○	○	○
[0.25 , 0.5)	●	○	○
[0.5 , 0.75)	●	●	○
[0.75 , 1)	●	●	●

Table 2.1: Input feature quantization for a binary RBM



(a) (b)

Figure 2.5: Different methods to feed RBMs with quantized values. A straight approach is depicted in (a), compared to a more sophisticated way in (b). Each coloured block represents one real–valued input feature.

In order to mitigate those drawbacks, the approach outlined in [52] can be used. The input data can be aligned in a two dimensional layout as depicted in Figure 2.5b, so the x dimension represents the features and the y dimension their quantized values. The weight matrix becomes three-dimensional, but the top layer is still one-dimensional as the sum of a column is used as activation probability for a neuron.

So far, the number of weights is still the same as before. But the three-dimensional weight matrix can be factorized into two lower-rank matrices:

$$W_{ij}^{k} = \sum_{c=1}^{C} A_{ic}^{k} B_{cj}.$$

The learning of the matrix W can be decomposed into two independent steps of learning A and B. Because both are lower-ranking, the number of parameters is significantly reduced. Hence, learning is faster than in the straight case. At the same time, there are enough parameters to tune and enough hidden variables to catch the interesting patterns. However, this approach is still constrained by the coarse binary quantization used for the input.

The most frequent approach to handle real-valued data with an RBM employs so called Gaussian units. They may be used either for the visible or hidden layer only or even for both layers. The

presentation outlines the usual type, namely Gaussian units for the visible units only. Thus, a mapping from a continuous input space $v \in \mathbb{R}^D$ and into a binary output space $h \in \{1,0\}^F$ is learned. Such a structure is needed as first layer of a DBN only, because the output is binary and well-suited for a binary RBM in the next layer as described in the beginning.

In fact, the difference emanates from a change in the energy function:

$$E(v,h) = \sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2} - \sum_j b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij}.$$

Thus, the energy function is enhanced with a quadratic term. This parabola has a width controlled by the parameter σ_i. A parabola has a definite maximum, so the network can express an exact preference value and a confidence with the σ parameter. By contrast, the binary units are constrained to a linear contribution, so they can present a bias only. A tendency is a meaningful endeavour in the binary case, but ambiguous for real-valued data. Therefore, feeding linear units with real-valued input yields inferior results in most cases.

In analogy to the previous case, the conditional probabilities can be derived for the network.

$$p(h_j = 1|v) = \varphi\left(b_j + \sum_i \frac{v_i}{\sigma_i} w_{ij}\right)$$

$$p(v_i = x|h) = \mathcal{N}\left(a_i + \sigma_i \sum_j h_j w_{ij}, \sigma_i^2\right)$$

$$\varphi(x) = \frac{1}{1 + \exp(-x)}$$

The formula for the visible units turns out to be a normal distribution with the mean value $a_i + \sigma_i \sum_j h_j w_{ij}$ and the variance σ_i^2. Thus, the visible units are linear with regard to the hidden layer, but the offset a_i conveys the preferred value. The parameter update rule changes to reflect the new parameter.

$$\frac{\partial \log(P(v;\theta))}{\partial w_{ij}} = \mathrm{E}\left\langle \frac{v_i}{\sigma_i} h_j \right\rangle_{\text{data}} - \mathrm{E}\left\langle \frac{v_i}{\sigma_i} h_j \right\rangle_{\text{model}}$$

$$\frac{\partial \log(P(v;\theta))}{\partial \sigma_i} = \mathrm{E}\left\langle \frac{(v_i - a_i)^2}{\sigma_i^3} - \sum_j \frac{v_i}{\sigma_i^2} h_j w_{ij} \right\rangle_{\text{data}} - \mathrm{E}\left\langle \frac{(v_i - a_i)^2}{\sigma_i^3} - \sum_j \frac{v_i}{\sigma_i^2} h_j w_{ij} \right\rangle_{\text{model}}$$

As shown above, the variance σ_i^2 can be learnt in a manner analogous to the other parameters of the neural network, or it can be fixed to a certain value [24, 41]. Quite frequently, this is accomplished by preprocessing the data in such a way that each component has zero mean and unit variance and let σ_i^2 be equal to one during training [24, 42].

The learning meta-parameters must be tuned a bit to allow successful training of a Gaussian unit based RBM. First, the learning rate needs to be one or two orders of magnitude lower than in the pure

binary case. Otherwise, the Gaussian units tend to overshoot and get stuck in wrong areas because the units are unconstrained and much more sensitive to the absolute value of the descent. However, the absolute value is not approximated accurately by Contrastive Divergence. The lower learning rate goes hand in hand with a longer training time until the weights converge to their final values. Moreover, learning a precise bias is a much more challenging task, so the increased number of epochs can be justified from this point of view, too.

2.5 Summary

In this chapter, the RBM has been introduced. This neural network architecture will be used throughout the whole thesis, so the basic idea in terms of a binary RBM has been highlighted in much detail. In addition to that, the binary RBM provides the theoretical basis for many enhancements, so the mathematical derivation has been provided as example.

Next, the most common technique to form DBNs with RBMs as building blocks has been depicted. There are a few other techniques with more theoretical background, but they are not used in practice due to unsatisfiable constrains.

Finally, extensions into other input spaces have been discussed. However, the output layer of those enhancements are binary still, so only the first layer of a DBN architecture needs to employ a different technique. The restriction to binary input features limits the applicableness to very simple examples, so this section provides the foundation for all applications presented in this work.

Several extensions to the theory outlined in this section are used in practice. Indeed, the introduction of those techniques has been postponed to a later chapter to enlarge the presentation with practical experienced and obtained examples.

This page is blank intentionally.

Chapter 3

GPU Computing

Techniques to exploit the computational power of graphic card in modern PCs to accelerate the performance of software are presented in this chapter. In fact, the quantity of transistors in recent graphic processing units (GPU) has superseded the amount in their central processing unit (CPU) counterparts. Due to the programming flexibility introduced in the last generations, the applicability of GPUs could be extended to many domains besides image rendering. Indeed, the thirst for realistic scenes computed in real-time has formed an architecture with a straightforward bias to the requirements of typical patterns. A modern GPU can clearly outperform a CPU for "suitable" tasks. The definition of "suitable" inspired two kinds of research: The elaboration of efficient algorithms for typical programming patterns on one hand side and demonstration of accelerated applications on the other hand. In this project, we focus on the latter as the training of a Restricted Boltzmann Machine as described in the previous chapter is accelerated by a graphic card.

In fact, the achievable speedup depends remarkably on the mapping of the algorithms onto the hardware architecture. Thus, the necessary theory is introduced in this chapter. After a short historic outline, the structure of a graphic card is illustrated. Next, some development tools to integrate the GPU into software are introduced. The chapter closes with a short summary of the important aspects.

The details presented in the chapter reflect the characteristics of the used hardware. However, new versions of graphic cards with enhanced features have since been released. Therefore, changes to the architecture are mentioned if it may contribute to the framework presented later on.

3.1 Historic outline

Moore's Law and the involved performance improvement have shown to hold much truth for many decades. Historically it has been envisaged in terms of the *central processing unit* (CPU) for which each

new generation could yield a speedup for almost all programs. The benefit was attained by a higher clock rate and additional functional units and cores. Indeed, most software uses a sequential program flow scheme. Although a higher clock rate affects such kind of software, the additional cores need more efforts. In fact, many techniques like dynamic scheduling and fast caches permitted most software to benefit at the cost of additional hardware. Up to 80 % of chip area is spent on the latter kind of enhancement in modern CPUs. Nevertheless, the tricks have practical limits and diminishing returns are notable for most of the techniques. For example, heat dissipation and signal propagation delays constrain the clock rate and the ability to add further units. A new approach has been employed as recent CPUs ship with several cores. Sequential programs cannot exploit the additional performance because no hardware support is provided to distribute the computation automatically. More sophisticated techniques like threads must be engaged by the developers to utilize the additional cores. However, most algorithms and programming patterns are oriented to serial code, so an extensive reorganization is necessary.

Meanwhile, Moore's Law had an influence on graphic cards as well. First, new hardware–based features have been included in each new generation. The demand for more realistic graphic effects however has entailed a major step from fixed hardware-operations to more flexible, software controlled shaders. All graphic devices with DirectX 10 support offer this new kind of adaptability. Thus, a GPU has turned into a conglomeration of processors with high flexibility. Hardware design must be heavily optimized to the typical characteristics of video processing in order that complex jobs may be handled by the GPU in real time.

The new flexibility of GPUs also meant that applications in fields other than image processing became possible. At first, the technical obstacles were very high, because no dedicated interface was available. Hence, the data had to be handled using the image rendering oriented API, limiting the scope and advantages of *general purpose GPU computing* (GPGPU) at the same time. Several GPU development languages emerged, starting with CUDA in 2007. Those technologies allowed a tight integration between CPU and GPU, an important aspect because the GPU is beneficial to a small subset of tasks only.

With those new technologies, outstanding improvements in many computational intensive domains of application have been reported in the literature. Examples range from databases [10], military and medical signal processing [34, 48] to molecular simulations [61]. In addition to that, the GPU has found its way into supercomputing, too, as some of the fastest number crunchers available exploit the power of GPU clusters in favour of CPU clusters.

3.2 API support

As noted above, a tight integration between CPU and GPU is essential in attaining a remarkable speedup. A prominent role played by straight software support archiving the computations, especially with regard to synchronization and parallelism. Nowadays, three APIs for GPGPU computing have been established:

CUDA This is the oldest and therefore most mature API proposed by NVidia. The name is an acronym for *Compute Unified Device Architecture*. Several support libraries, especially for FFT and BLAS[1], are included and relieve developers from the need to write their own code for very common operations. Those operations are heavily optimized by NVidia, so most own implementations should yield worse results. Due to library support and the leading role in early GPGPU computing, CUDA has been employed by many researchers and companies. The most notable drawback is the requirement of an NVidia graphic card, because no other GPU manufacturers or processor targets are supported. However, the close dependency opens out into the best match between hardware architecture and software features. In addition to that, this is the only API with mature double precision floating point support at the time of writing this thesis.

DirectCompute Starting with DirectX 11 in October 2009, Microsoft included a GPGPU API called DirectCompute. Although supported by different graphic cards, the Windows operation system Vista or 7 is necessary. Thus, the API has not reached the same level of popularity as CUDA yet.

OpenCL Initiated by Apple, another API has emerged recently called *Open Compute Language* or OpenCL. Unlike the previous APIs, it is based on an open royalty-free standard by the Khronos group, the committee behind OpenGL as well. All major operation systems like Windows, Linux and Mac OS X are supported, as well as heterogeneous platforms, consisting of CPUs, GPUs and other processors. A remarkable feature is the flexibility to adapt to different processors at run-time, so OpenCL code may be executed on the CPU if no GPU is available. By contrast, the code generation may be less optimized due to the broad spectrum of platforms. Nevertheless, support and popularity of OpenCL is increasing fast, especially due to the flexibility. Although the basic interface is standardized, different implementations may provide additional features to exploit benefits of particular hardware or additional libraries (e.g. libraries for common matrix operations). Obviously, the usage of those enhancements restricts portability to other hardware or platforms.

CUDA has been employed as GPU API in the scope of this thesis. The decision was influenced by the lack of mature alternatives and double precision support at the beginning of the thesis. In fact, OpenCL has caught up with CUDA meanwhile. As OpenCL matures, it may become the platform of choice for developers, as it will enable comparisons between architectures, e.g. CPU and GPU, at minimal additional programming efforts.

In fact, both are C/C++-style dialects, allowing a subset of the original standard on the one hand side, but including some extensions for specific features on the other hand. Examples for the latter case are the intrinsics for synchronization or detailed memory assignment. The focus in this thesis will be on CUDA and the NVidia processor architecture, wherein the general conditions of a balanced design will be presented. Due to the historic development from a graphic card to a general purpose processor, we first deal with the hardware architecture and then outline its realization in software.

[1]Basic Linear Algebra Subprograms (BLAS) is a set of basic linear algebra functions. It was proposed 1979 in the Fortran programming language and has developed to a de facto API for matrix and vector operations.

3.3 Hardware

An NVidia Quadro FX 5800 with 4 GB of RAM has been used as graphic card. Hence, the emphasis during the hardware description will be on this generation of GPUs. A more recent revision, called Femi, overcomes some of the restrictions outlined in this section. Notable changes relevant to the presented work are mentioned, but could not be employed as no appropriate hardware was available. Beforehand, it should be noted that the detailed architecture of graphic card processors is intellectual key knowledge of the manufactures and therefore subject to a strict non–disclosure policy. More details about the structure have been unveiled recently due to the close dependency between performance and hardware in GPGPU, but several aspects remain subject to speculation.

The GPU is designed as a throughput–oriented SIMD[2] processing unit. Thus, the architecture roots on two main components, namely the processors and the memory to deliver the data. As for most high–performance architectures, either instruction execution or data delivery constrain the maximum throughput. Therefore, kernels are usually classified to be either instruction or memory limited, with each one requiring different optimization strategies. Indeed, the division is not that clear, as the limiting factor of a kernel may change frequently during optimization.

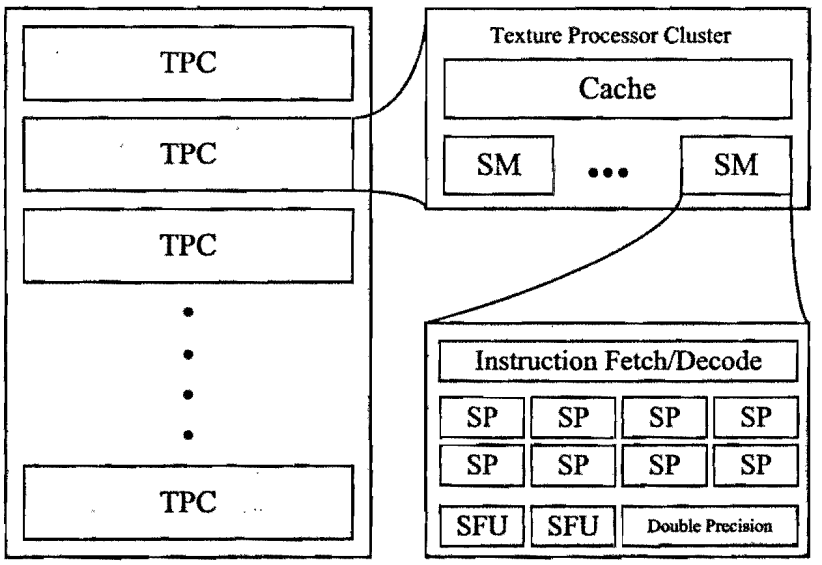


Figure 3.1: Hardware architecture of a NVidia GPU

With regard to the structure, the processor design is different from typical CPUs. Several performance improving facilities like branch prediction and out-of-order execution have been omitted in favour of higher clock speeds and additional cores. Thus, the performance of code segments is much closer dependent on the architecture than for the CPU counterparts. Furthermore, the hierarchy of processors is

[2]single instruction, multiple data

different from the usual SIMD configuration. The smallest building block, called *Stream Multiprocessor* (SM), is outlined in Figure 3.1. Each SM in turn is based on different functional units, namely:

Stream Processors (SP) [x8] are the flexible programming units to perform most of the computational work like integer arithmetic, branching and load/store operations. Floating point arithmetic is limited to single precision only.

Special Function Units (SFU) [x2] provide hardware accelerated special functions, mainly transcendental and trigonometric functions common in image processing. However, those implementations favour speed over accuracy, as the human eye is more sensitive to frame rates than small colour errors. A per-case examination is necessary with regard to GPGPU to balance required precision with maximum speed.

double precision unit [x1] offers double precision floating point operations. Especially older hardware revision ship without such a hardware support and need slow software emulation. Indeed, the double precision speed is about eight times lower than for single precision, due to the ratio of the units. In practice, the penalty is slightly lower as it depends on the ratio of double precision computations and SP/SFU instructions.

All those units share one instruction decode unit, so they cannot be addressed with different tasks in one instruction cycle, but form a fixed entity. Thus, a Stream Multiprocessor is the basic parallel building block, being replicated several times to constitute the processing power of modern graphic cards. Each SM is less powerful than a CPU core, but the sheer majority induces the performance benefit. Having said that, the most outstanding demand for a GPU-suitable algorithm is obvious: the algorithm must be able to be distributed among all processors equally to exploit the advantage. For example, there are 30 SMs and therefore 240 SPs in the graphic card used for this thesis.

Another key property arises due to the shared instruction decode unit per SM in the light of branches. Each SP can opt out from any command independently, so the flexibility is enhanced compared to usual SIMD implementations. In this case, the computational performance degenerates with the ratio of working processors. In fact, the situation is worse for diverging code paths, e.g. *if* {} ... *else* {} in C/C++. If a single SP needs to go along the other path, both code segments need to be delivered to all processors. This may cause a significant performance penalty for instruction limited kernels.

The second focus is on the memory architecture to supply the processors with data. The available types of memory are listed in Table 3.1. Indeed, constant and texture cache are not independent memories, but used to hide latencies of constant and global memory respectively. Nevertheless, they are listed twice to outline the difference in access cycles and limited size. Except for those two buffers, the whole architecture does not use any caches[3], so the memory access pattern may easily influence the runtime of a code segment by one or two orders of magnitude. A good understanding of the benefits and drawbacks of the different memories is essential to optimize an application. Therefore, the differences need to be outlined in more detail:

[3]Global memory access is partly cached in the Femi architecture, so some aspects of coalesced memory access patterns are alleviated

Memory	Location	Size	Latency	Notes
Register	on-chip	64 KB / SM	instant	Independent for each SP
Shared	on-chip	16 KB / SM	instant	Shared among all threads of a SM
Global	off-chip	up to 4 GB	400-600 cycles	
Constant Memory	off-chip	64 KB / device	400-600 cycles	
Constant Cache	on-chip	8 KB / SM	instant	only if cached
Texture Cache	on-chip	8 KB / SM	$\geq$ 100 cycles	

Table 3.1: Available memory types on a GPU with typical characteristics

Global memory With up to 4 GB in size[4], the global memory is the largest of all available memory locations. It is read/write-accessible by all kernels and the CPU, so global memory serves as memory for computation inputs and results. The latency for access is about 400 to 600 cycles, so it is the slowest of all memories, too. The connection between global memory and processors is a wide bus, 512 bit in the case of our card, so each request needs to transfer all bits. Unused data within a request are discarded due to the lack of cache, but the whole block needs to be transferred. Thus, a major concern for memory throughput–limited kernels is how to optimize the utilization ratio. Thereby, a full utilization of a data transfer is called *coalesced* memory access. This matter is revisited in section 3.4 with regard to the programming model.

Shared memory Shared memory is accessible by all threads running on particular SM, but it is not coherent between different SMs. Usual use-cases are software–managed global memory caches and data exchange between SPs of a SM. The memory is clustered into 16 banks. There are two favoured access modes: Either each SP accesses memory locations in different banks or all SPs read the same memory location (broadcast). The data will be available instantaneously in both cases. On the other hand, the access will be serialized by hardware if none of the patterns are fulfilled.

Constant memory Constant memory is a small region in global memory, but access is cached for each SM independently. The data is available at register speed in case of a cache hit and incurs global memory latency otherwise. The access is read–only to avoid any cache–coherency problems. Typically, data commonly used by different SM and/or frequent kernel invocations is stored here. Contrary to shared memory, access will be serialized if SPs request different memory locations at the same time.

Registers Registers have the lowest latency, but are very limited. Currently, 32 registers per thread are available only. The compiler will spill data to global memory if more registers are needed for computations, so the kernel will be able to execute at the expense of high latency global memory access. An optimized kernel should avoid this kind of issue by all means.

[4]The Femi architecture is not constrained to 32 bit pointers, so more memory is supported.

Texture cache The texture cache is a small cache per SM to alleviate the dependency on access patterns for global memory. Interpolation and normalization of data is available additionally. No cache–coherency is guaranteed, so the global memory is read–only using the texture reference. A predictive cache algorithm determines the data to store, so the performance may improve if a proper access pattern cannot be assured otherwise.

Thus, global memory is frequently used to transfer data between CPU and GPU. The transfer speed is about 8 GB/s for PCI-Express in our case, compared to a maximum of 74 GB/s within the GPU, e.g. coalesced memory reads in kernels. Hence, the transfer between CPU and GPU may consume any computational benefit. It will frequently turn out to be of advantage to implement some functions on a less suited architecture in favour of an additional data transfer.

A way to ease the impact is offered by the CUDA API with *pinned memory*. This special kind of memory is non–pageable memory (therefore pinned), so transfers can start immediately and can employ asynchronous DMA transfers. In fact, the transfer is asynchronous regarding CPU and GPU on recent hardware architectures, so functions may be executed while data is transferred. However, the greedy allocation of pinned memory may decrease the performance of the CPU part, as frequent paging for other data is enforced due to the reduced amount of movable memory. Thus, pinned memory should only be used wisely and sparsely for those buffers that will boost the application.

Different hardware revisions with enhanced features are available. A number scheme called compute capability is employed to distinguish the supported abilities. All compute capabilities are backward–compatible, so a device with compute capability 1.3 can execute code compiled for any previous revision as well. This is not the case the other way round, so the hardware revision needs to be provided at compile time. In practice, up to three different revisions may be set and most appropriate one is selected at runtime. With regard to our project, compute capability 1.3 is the most important as double precision support has been added.

3.4 CUDA Programming model

As outlined before, the GPU is essentially a SIMD processor with enhanced flexibility. The model is called *single instruction multiple threads* (SIMT) by NVidia to reflect the additional adaptability and describe the underlying philosophy. In fact, the hardware design is abstracted by means of thousands of independent lightweight threads, scheduled by hardware to execute on the SMs. The program flow is controlled by a CPU thread, issuing asynchronous GPU functions, memory transfers and other CPU code. Nevertheless, the scheduling properties of the GPU are very important to tune the design to the hardware.

In the context of GPU scheduling, the most important buzzword is the *wrap*, the smallest number of threads being executed concurrently on a SM. Against all odds, the wrap size is 32, but subject to

changes in future hardware revisions according to the documentation. As a consequence, a SM needs at least 4 cycles to complete a single instruction for all wraps. It is further separated into *half-wraps*, namely the upper and lower 16 consecutive threads of a wrap.

A wrap plays an important role regarding most of the structural implications mentioned before. First, all threads of a wrap must access contiguous memory locations for coalesced global memory transfers[5]. Next, all threads of a half-wrap must access different shared memory banks to avoid serialization. Moreover, divergent code paths are bound to wraps, too. Therefore, if one thread in a wrap needs to follow another code path, both code paths will be fed to all wraps as outlined before. By contrast, no penalty occurs if threads in distinct wraps pursue divergent paths.

The wrap leads to the matter of issuing threads. Two parameters provided for each kernel launch to control the number of threads, called *block* and *grid*. All kernels of a block are executed on a single SM and can synchronize and share data using shared memory. Although threads of different grid blocks may run on the same SM, they are treated as independent and cannot interact. In general, there is no synchronization mechanism in between SMs, so the tasks need to be completely independent[6]. The synchronization available in CUDA is limited to threads of a block and kernel launches, avoiding any expensive large-scale synchronization hardware.

The definition of a CUDA thread is significantly different from a CPU thread. There is hardly any generation or destruction overhead and switching between threads works instantaneously, because no stack is allocated. In fact, each SM can handle up to 512 threads at the same time. Beside hardware, the maximum number of threads per SM is constrained by shared memory and registers. Reducing the limiting factor will cause more threads to be scheduled for each SM.

The threads are scheduled to hide latencies, especially for global memory access, so one wrap may compute while another one is waiting for data. This leads to *occupancy*, the ratio of used thread slots of the SMs, as next key factor for memory access limited kernels. However, a higher occupancy does not imply a higher performance. A lower number may be sufficient for compute bound kernels to hide latencies completely.

The different philosophy of threads is clearly evident in vector addition for example. A typical GPU implementation uses an own thread for the addition of each vector component, while a CPU implementation favours a loop. The special variables *blockIdx* and *gridIdx* are available in kernels to split the workload to threads.

The trade-off between more kernels and additional workload per kernel must be based on experience and experiments with different setups. A fine-grained kernel approach may easily be limited by the memory transfer rate. By contrast, a heavy workload partitioning tends to less occupancy and therefore speed limitations due to memory latencies and register spilling.

[5]Further restrictions apply for the access pattern with respect to the compute capability

[6]Recent revisions support atomic operations on global memory, so synchronization may be implemented, although very inefficient due to the high latency of global memory.

We want to limit the scope of this presentation to a fundamental basis to understand the differences between CPU and GPU software development. In fact, there are much more aspects to consider, partly due to the graphic processing background. For example, the native integer size is 24 bit instead of 32 bit, as colour graphics tend to use 8 bits per colour only. Therefore, a 32 bit multiplication needs multiple cycles, whereas a 24 bit one can be accomplished in a single cycle[7].

Even more influence has the chosen algorithm. Several high–performance CPU algorithms do not map well onto the new architecture and lead to less impressive improvements. The research for more effective alternatives is the most time–demanding and unpredictable part, but often necessary to yield the superior speedups reported in literature.

3.5 Development tools

As outlined before, the CUDA API has been used for GPGPU in this work. In fact, two similar ways are offered for programming the GPU, namely through the runtime or driver API. While the latter allows direct access to functions, the former uses a more abstracted interface and generates implicit stubs at compile time, so the code is more concise. An example to illustrate the difference is given in Listings 3.1 and 3.2. Error handling has been stripped in both cases for the sake of clarity. Both APIs are equal in terms of features and may be blended as well. However, any file using runtime API extensions must be processed by the NVidia nvcc compiler.

In general, the NVidia compiler is a pre-processor to separate host and device code, feed both to different compilers and merge the results into one single executable. The compile flow is outlined in Figure 3.2. The source code is scanned by the nvcc compiler and GPU relevant code is forwarded to the PTX compiler. The compiler generates GPU suitable code in a binary intermediate format and embeds the code as character resources in a format called *fat binary*. In practice, this code will be processed by the graphic card driver to run on the GPU, especially due to backward–compatibility support. The enriched code is forwarded to the host compiler to generate a platform specific executable. GCC is supported on Linux and Mac Os X and Microsoft Visual Studio 2008 on Windows platforms. Although the native Mircosoft compiler is used, some features are not available due to the preprocessing, most notable the *edit and resume*-option[8]. The lack of this feature is unfortunate due to the limited monitoring possibilities within the GPU and long simulation times.

As the development was conducted on the Windows platform, the emphasis here is on the development support on this platform. As mentioned before, the Microsoft C compiler is the only supported compiler for the Windows platform. The compiler in the free edition of Microsoft Visual Studio Express 2008 is fine for this purpose.

[7] Indeed, NVidia notes this may change in future hardware revisions.

[8] Edit and resume allows to modify the code during the runtime of the program without restarting the program. Thus, it is a convenient method to enhance source code with monitoring commands or fix errors during debugging.

```
1 //Load PTX source
2 cuModuleLoadDataEx(&cuModule, ptx_source.c_str(), jitNumOptions,
3                 jitOptions, (void **)jitOptVals);
4 cuModuleLoad(&cuModule, module_path.c_str());
5 cuModuleGetFunction( &cuFunction, cuModule, "matrixMul" );
6
7 //setup parameters
8 cuParamSetv( matrixMul, offset, &ptr, sizeof(ptr));
9 cuParamSetv( matrixMul, offset, &ptr, sizeof(ptr));
10 cuParamSetv( matrixMul, offset, &ptr, sizeof(ptr));
11 cuParamSeti( matrixMul, offset, Matrix_Width_A );
12 cuParamSeti( matrixMul, offset, Matrix_Width_B );
13
14 //configure launch options
15 cuParamSetSize( matrixMul, offset );
16 cuFuncSetBlockShape( matrixMul, BLOCK_SIZE, BLOCK_SIZE, 1 );
17 cuFuncSetSharedSize( matrixMul, 2*BLOCK_SIZE*BLOCK_SIZE*sizeof(float) );
18
19 //launch kernel
20 cuLaunchGrid( matrixMul, WC / BLOCK_SIZE, HC / BLOCK_SIZE );
```

Listing 3.1: Example of kernel invocation using CUDA driver API. Some code has been stripped for the clarity of presentation, most notably error handling.

```
1 matrixMul<<< grid, threads >>>(d_C, d_A, d_B, uiWA, uiWB);
```

Listing 3.2: Example of same kernel invocation as above using CUDA runtime API.

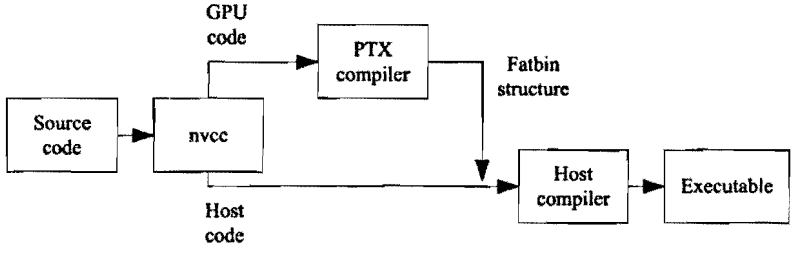


Figure 3.2: Dataflow in the CUDA compile process

By default, the compilation is a command–line driven process. A Visual Studio integration, called *Parallel Nsight*, has been released recently [7]. Debugging and monitoring are major improvements beside the integrated project maintenance. The free edition covers debugging only, but the professional edition offers monitoring and contains a profiling and comprehensive timeline presentation including CPU and GPU execution.

Nevertheless, the debugging gives rise to several issues and restrictions outlined next. First, break

points are accomplished in a straight way during debugging, so the whole graphic card will be stalled including all video operations as well. Therefore, a single graphic card cannot be used for development and debugging at the same time, as the display would freeze. NVidia suggests dedicated development and debugging machines connected with TCP/IP. The code and results are transferred transparently by an additional debugging client on the debugging machine.

GPU debugging is available this way, but CPU debugging is not possible meanwhile. In fact, it is impossible to debug both, CPU and GPU code, simultaneously with only a single instance of Visual Studio[9]. This situation frequently limits the debugging insight, because intermediate results cannot be judged with respect to the whole program state. Furthermore, the benefit of GPU debugging covers error correction mainly, because hundreds of threads cannot be examined at the same time. Hence, a *test driven development* is suggested by the limitations and the fact, that several optimizations may be necessary to yield good performance.

Tests to ensure the proper operation of the code are written before the actual code in this development style. In the light of GPU development, a working CPU implementation serves as a reference implementation, usually called *gold code*. In a loose sense, this kind of development can be regarded as black box development, as different implementations are compared by their results only. Automatic tests on different scales of granularity, ranging from functions to whole segments, are defined. Most development systems ship with built-in support for such tests using the buzzword *unit testing*. In fact, the development is slowed down first due to the additional work for the CPU implementation and the tests, but will pay off in the long term view as regressions and flaws can be traced down to certain functions automatically.

The actual GPU code can be assessed using different methods. A free standalone profiler is provided beside the comprehensive and expensive Parallel Nsight monitoring tools. Moreover, the CUDA runtime supports events to capture the timestamp at specific points in the asynchronous GPU processing queue. The timestamp is independent from any CPU based timer, so no correlation of CPU and GPU execution can be deduced. CPU profiling techniques on the other hand may yield less meaningful figures due to the asynchronous nature of the CUDA API. In fact, deep understanding is needed to put the different perspectives together into an informative overall picture.

The estimation of speedups due to parallelism of code has been in the focus of interest as soon as multi-core CPUs became popular. *Amdal's Law*, shown in Equation 3.1, offers a rough idea about the maximum speedup possible by separating the runtime of a program into serial and parallel sections. In this sense, parallel sections can take advantage of multiple processors, whereas serial sections cannot benefit and the runtime remains constant. It is assumed that the task can be distributed among all processors equally and neglects any synchronization overhead for the sake of simplicity.

$$\text{Speedup}_{\text{max}} = \frac{1}{\text{Serial Code}_{\%} + \frac{\text{Parallel Code}_{\%}}{\text{Number of processors}}} \tag{3.1}$$

[9]It is possible to debug both using different instances of Visual Studio on the development and debugging PC at the same time, but this configuration is prone to freezes, most likely due to assumptions in the debug client.

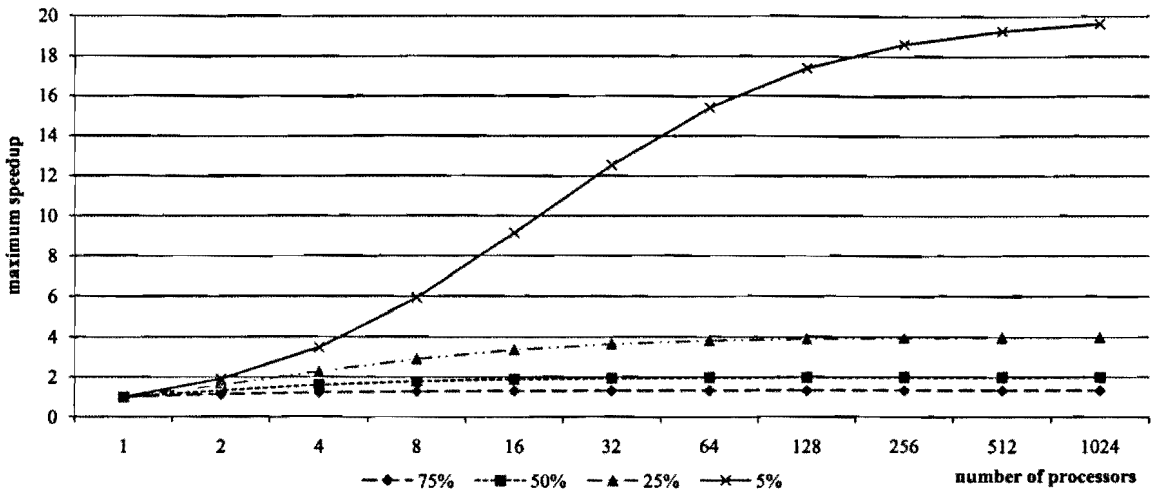


Figure 3.3: Estimation of speedup with Amdal's Law based on percentage of serial code

The trend for various percentages of serial code is depicted in Figure 3.3. The key observation resides in the very limited benefit of multi-processor architectures for algorithms with serial code segments. For example, if only 5 % of time are spent on serial code segments, the speedup is limited to a maximum of 20 times. Typical serial code portions are allocation, initialization and evaluation, so 5 % may be achieved easily.

By contrast, there is no way to predict the actual speedup a GPU implementation will provide over a CPU counterpart due to the dependency on multiple factors. Amdal's Law cannot be applied literally as the GPU cores are different from CPU cores and therefore violate a key assumption. Indeed, it can be used to estimate the upper limit of benefit due to changing minor aspects, e.g. functions. Therefore, it is assumed that the function may be distributed perfectly among an infinite set of processors, so the time spent in the function is avoided. The estimation is reasonable for small portions only, as the assumption to neglect a major part yields pointless conclusions. Furthermore, major modifications like different algorithms cannot be assessed.

3.6 Summary

In this chapter, the hardware and API to facilitate GPGPU computing has been introduced. After a short historic outline, common APIs have been presented. In so, it becomes obvious that GPU computing is a very recent matter with a fast-moving evolution. At the same time, information is spread by means of new technologies like web tutorials and conference presentations to allay the thirst of additional knowledge of the research community. Hence, this chapter provides a comprehensive summary of information gathered during the thesis.

The processor and memory design of the used graphic card have been outlined in detail first, followed by the realization in software. The instruction should provide a comprehensive understanding of limitations and prospects, although the aspects are covered at a high level only.

Finally, the current development support provided by NVidia has been presented. However, the situation is subject to rapid changes because GPU computing is a rather new topic with a fast growing community. Amdal's Law has been presented to outline the importance of a suitable algorithm and assess the impact of code sections.

This page is blank intentionally.

Chapter 4

KDD 99 Cup Evaluation

In this chapter, the performance of a Restricted Boltzmann Machine on the KDD 99 Cup data is examined. In a chronological context, this work has been done first and the promising results and limitations of the presented approach encouraged the development of the framework introduced later on. The organization of the chapter is as follows: First, the dataset and the task is introduced. Second, we present our approach to employ an RBM on the problem and outline the possible speedup through a GPU implementation. The section is concluded with an evaluation of classification results achieved with the suggested technique.

4.1 KDD 99 Cup Dataset

The KDD Cup dataset was compiled using *tcpdump* data of the 1998 Defense Advanced Research Projects Agency (DARPA) Intrusion Detection System (IDS) Evaluation dataset. The original data was gathered by simulating a network using traffic generators and hand-injected attacks, so the type of each package, either benign or malicious, can be assured [36]. The network was simulated for seven weeks, producing four gigabytes of compressed binary TCP dump data. An example of the dump data is shown in Figure 4.1.

This data was processed by KDD and around 5 million connection records were extracted. A connection is a sequence of packets starting and ending at some well defined times, between which data flows to and from a source IP address to a target IP address under some well defined protocol. Each connection is labeled as either normal, or as an attack, with exactly one specific attack type. Each connection record consists of about 100 bytes. An example of such an connection record is shown in Figure 4.2. As we can see from Figure 4.2, each connection is described by 41 features. A complete list about all features can be found in [3]. In general, the features can be distinguished into three types:

Attack type	Training		Testing	
	[#]	[%]	[#]	[%]
normal	972781	19,86	60591	19,48
buffer overflow	30	0,00	22	0,01
loadmodule	9	0,00	2	0,00
perl	3	0,00	2	0,00
neptune	1072017	21,88	58001	18,65
smurf	2807886	57,32	164091	52,76
guess passwd	53	0,00	4367	1,40
pod	264	0,01	87	0,03
teardrop	979	0,02	12	0,00
portsweep	10413	0,21	354	0,11
ipsweep	12481	0,25	306	0,10
land	21	0,00	9	0,00
ftp write	8	0,00	3	0,00
back	2203	0,04	1098	0,35
imap	12	0,00	1	0,00
satan	15892	0,32	1633	0,53
phf	4	0,00	2	0,00
nmap	2316	0,05	84	0,03
multihop	7	0,00	18	0,01
snmpguess	0	0,00	2406	0,77

Attack type	Training		Testing	
	[#]	[%]	[#]	[%]
warezmaster	20	0,00	1602	0,52
warezclient	1020	0,02	0	0,00
spy	2	0,00	0	0,00
rootkit	10	0,00	13	0,00
snmpgetattack	0	0,00	7741	2,49
named	0	0,00	17	0,01
xlock	0	0,00	9	0,00
xsnoop	0	0,00	4	0,00
sendmail	0	0,00	17	0,01
saint	0	0,00	736	0,24
apache2	0	0,00	794	0,26
udpstorm	0	0,00	2	0,00
xterm	0	0,00	13	0,00
mscan	0	0,00	1053	0,34
processtable	0	0,00	759	0,24
ps	0	0,00	16	0,01
httptunnel	0	0,00	158	0,05
worm	0	0,00	2	0,00
mailbomb	0	0,00	5000	1,61
sqlattack	0	0,00	2	0,00

Table 4.1: Distribution of attack types in training and testing database

1. Basic features of individual TCP connections:
   - **duration** - length (number of seconds) of the connection
   - **protocol_type** - type of the protocol, e.g. tcp, udp, etc.
   - **service** - network service on the destination, e.g., http, telnet, etc.
   - **src_bytes** - number of data bytes from source to destination
   - **dst_bytes** - number of data bytes from destination to source
   - **flag** - normal or error status of the connection
2. Content features within a connection suggested by domain knowledge:
   - **logged_in** - 1 if successfully logged in; 0 otherwise
   - **num_failed_logins** - number of failed login attempts
   - **num_shells** - number of shell prompts

- **num_root** - number of "root" accesses

3. Traffic features computed using a two-second time window:
   - **count** - number of connections to the same host as the current connection in the past two seconds
   - **srv_count** - number of connections to the same service as the current connection in the past two seconds
   - **serror_rate** - % of connections that have "SYN" errors

As stated in [3], the whole dataset is heavily peppered with attacks. In total there are 38 different types of attacks that can be grouped into four categories:

1. **DoS (Denial of Service):** an attacker tries to prevent legitimate users from using a service (e.g. syn flood)
2. **Probe:** an attacker tries to gather information about the target host (e.g. port scanning)
3. **U2R (User to Root):** an attacker has local access to a host and tries to get root privileges
4. **R2L (Remote to local):** an attacker tries to get local access on a host

A training set as well as a test set are provided by KDD. Because of the huge size of each dataset, 10% subsets of both have been provided. Regarding both sets it has to be noted that:

- the test data is not from the same probability distribution as the training set.
- the training dataset includes only 24 different types of attacks
- the test dataset includes all 38 attack forms

The distribution of attacks in both datasets is shown in Table 4.1.

The KDD Cup 99 dataset has become widely discredited [40] and it is recommended by the publishers to stop using this dataset to evaluate real world IDSs [15]. The main criticisms are, that the underlying DARPA dataset does not represent real world traffic and that some irregularities within this dataset have been discovered. Thus, the feature extraction done by KDD might have failed in a few cases. However, we do not intend to develop a real IDS system, but want to compare the performance of Deep Belief Networks with results by other researches. In addition to that, the unseen attacks are very interesting because we can analyze the generalization capability of a DBN. As noted before, the decomposition into general features should to more robust definitions of attack patterns, so one of the major advantages of the novel training algorithm can be assessed.

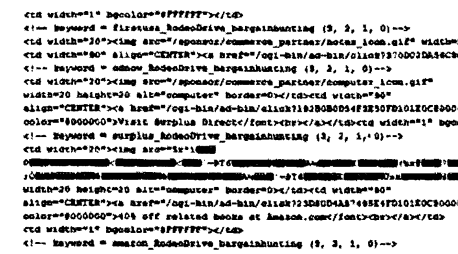


Figure 4.1: TCP Dump data as provided in the 98 DARPA IDS Evaluation dataset

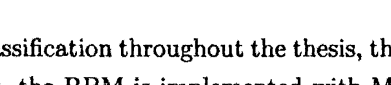


Figure 4.2: Data provided in the KDD Cup 99 dataset

4.2 RBM Classification

Although we use the RBM as for classification throughout the thesis, the focus in this section and the rest of the work is quite different. First, the RBM is implemented with MATLAB as high-level framework, and some parts are accelerated by the graphic card using MEX-files. Second, Contrastive Divergence is used to pre–learn a network and back propagation learning to trim the weights more precisely to the demanded classification task. In the other sections, back propagation is abandoned in favour of a better generalization performance. A flow chart of the training is depicted in Figure 4.3. The part accelerated by CUDA is highlighted in blue. The proposed network structure is depicted in Figure 4.4. The implementation is based on a publication of G. Hinton [24].

The test setup has been enhanced using a tight integration of different software products. A MySQL database delivers the test datasets, controlled through the MATLAB Database Toolbox to compile the training data adaptable to specific needs. An additional benefit of the database solution is the normalization of the benchmark data. A textual description is used for some categorical features like attack or connection type. Neither MATLAB, nor a Boltzmann network can handle character data efficiently, so each feature must be replaced with a unique integer representation. This task is called database normalization in computer science and can be achieved efficiently using a SQL database, even for huge data pools like the whole KDD Cup data.

An additional quantization is needed as integer values and real-valued ratios are not suitable for the binary RBM used in this experiment. This conversion has been implemented using *user defined functions* (UDF) in MySQL. UDFs are native machine code functions, loaded from shared libraries at runtime. Thus, UDFs are comparable to MEX-files for MATLAB. They are an intermediate step between fast native functions, which are time consuming to implement, and complex SQL statements, which are slow regarding execution time. The two different kinds of types of continuous data, namely classes and ratios, need to be represented slightly differently. Therefore, two user defined functions have been implemented.

- Binarize: Based on the value m of the feature, m bits are one.

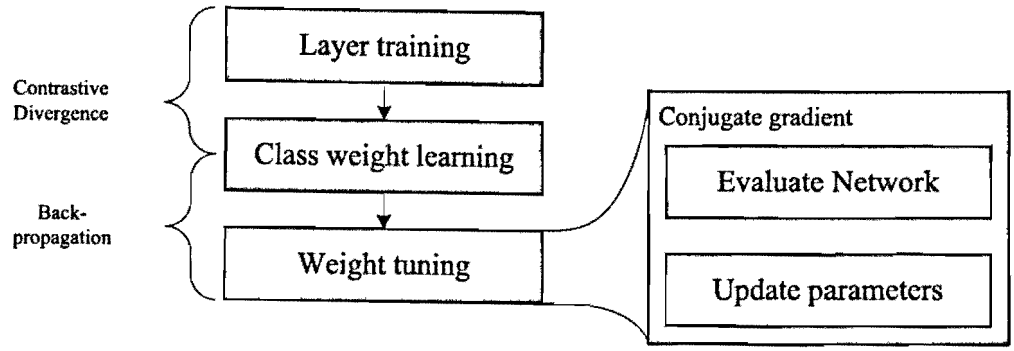


Figure 4.3: Learning stages of the DBN architecture to solve the KDD Cup 99 task. The part accelerated with CUDA is highlighted in blue

- Classifier: Based on the class m of the feature, the m^{th} bit is one

In fact, *Binarize* implements the idea depicted in Table 2.1, but classes are treated differently because there is no reasonable distance metric to mimic. The binary data is returned as character array of ASCII zeros, ones and spaces because MySQL allows each UDF call to return a single value only. A practical disadvantage is the slow string handling in MATLAB, so the data was written into an intermediate file to moderate the drawback.

A notable issue is the memory consumption. The network grows quite large and requires big matrices to hold the weights. Thereby, only 64 bit machines are capable of running the simulation as there is not enough contiguous memory available otherwise. MATLAB's demand for memory cannot be satisfied with the *3G* parameter either. The memory demands are partly due to representation of the input features, but restriction to small scale problems is evident.

The implementation itself has been enhanced in several aspects, too. First, the classification weights have been pre-learned with Contrastive Divergence using the classification idea in Figure 2.4b. Those weights were trained using back propagation only in the original proposal. Furthermore, the most time-consuming part of the back propagation learning has been implemented in CUDA. There are several reasons to pick this small portion in favour of other parts:

- The section is used up to 20 times for each gradient search, so an efficient implementation pays of well.
- It is called with the same batch data repeatedly, so data structures can be transferred once and used several times. This is a major advantage as the data transfer is a well-known limiting factor with regard to the maximum speed improvement.

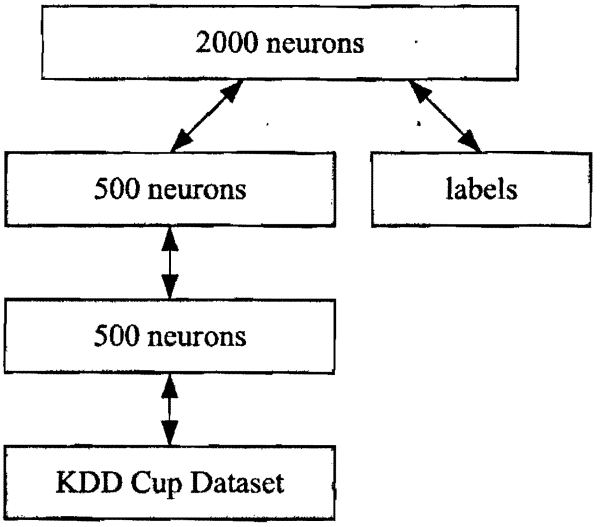


Figure 4.4: Deep Belief Network setup to solve the KDD Cup 99 task.

- It uses mainly forward operations like matrix multiplication and hardly any control code. Hence, the code section can easily be mapped into the parallel paradigm of a graphic card.
- Unlike most parts of the implementation, it is already separated as independent function.

The CUDA implementation uses own kernels for several per element tasks and the CUBLAS library for matrix operation. Profile results of the CUDA part are shown in Table 4.2. The own kernels could

Method	CPU time [usec]	[%]
dgemm_main_hw_na_nb	64494.90	25.74
dgemm_main_hw_na_tb	58549.00	23.37
dgemm_main_hw_ta_nb	64964.40	25.93
kernel1	19195.30	7.66
kernel_expsum	28.35	0.01
kernel_ix	28744.59	11.47
kernel_norm_log	4218.82	1.68
kernel_setValueMatrix	41.50	0.01
memcpyDtoH	4501.57	1.79
memcpyHtoD	5763.74	2.30

Table 4.2: Summary table of CUDA profiling results

be further optimized, but this work has been postponed in favour of network simulations first. Because more than 80 % of the time are spent on CUBLAS or memory transfer functions, the possible effect of further optimization, namely a speedup of $\frac{1}{0.8} = 1.25$, is minor compared to the required development time demands for optimization. The kernels kernel1, kernel_expsum, kernel_ix, kernel_norm_log and kernel_setValueMatrix are own kernels. The other functions belong to CUBLAS.

In its current state, the CUDA implementation is about two times faster than the MATLAB implementation on an eight core 2.66 GHz PC. The effect is more obvious on less powerful host CPUs, but even on this PC, the back propagation time for 100 epochs decreases form approximately 6 hours to less than 3 hours. Additionally, the CUDA time is hardly dependent on the data size. The parallel power of the GPU is only partially used by the current matrix sizes though.

In a later state, the CUDA functions have been revised based on new experiences with the best practices gained in the development of the framework presented in later sections. The time spent in self-developed kernels could be reduced to 4.5 % by updating the access pattern. The results are shown in Table 4.3. Additionally, the speed-up could be enhanced by using a bigger batch size, even so the batch size was constrained by MATLAB's memory demands. Once more, the benefits of CUDA have been restricted by MATLAB. Nevertheless, a three times speed-up for double precision and a roughly nine times speed-up for single precision could be observed with those adoptions. Indeed, the single precision implementation yields infinite values for the weights, so the back propagation training fails. A single infinite value will spread through the whole weight matrix very soon due to the matrix multiplication, so they should be avoided by any means. Most of the speedup may be preserved by using a mixed precision implementation, e.g. limit the use of double precision to the critical sections only.

Beside the computation, most time is spent on memory transfer. In fact, the situation is worse than suggested by the profiling results because the data transfers between CPU and GPU are not considered

Method	CPU time	
	[usec]	[%]
dgemm_main_hw_na_nb	64489.80	31.09
dgemm_main_hw_na_tb	58460.30	28.18
dgemm_main_hw_ta_nb	65121.80	31.39
kernel1	3371.33	1.62
kernel_expsum	28.54	0.01
kernel_ix	1692.83	0.81
kernel_norm_log	4223.23	2.03
kernel_setValueMatrix	44.10	0.02
memcpyDtoH	5417.76	2.61
memcpyHtoD	4549.89	2.19

Table 4.3: Summary table of CUDA profiling results with revised kernels

completely. The numbers include the time to setup the transfer only, instead of the complete transfer time.

A straight way to reduce the transfer time by a factor of five is pinned memory. By contrast, this cannot be used in conjunction with MEX-files because the MATLAB memory manager is incompatible with the other allocation and will force MATLAB to crash. Moreover, the allocated memory is not released properly and restrains the overall system performance.

Furthermore, the actual speedup due to the CUDA implementation is higher, but the gradient descent algorithm uses more iterations compared to the pure MATLAB implementation. The reason is the high dynamic range due to the exponential and logarithm functions used. The current hardware does not support fully IEEE-compatible floating points natively, but maps to functions with up to 2 ulp[1] difference. This causes an additional noise component to interfere with the gradient descent optimization. No degeneration of the final result can be observed, so the descent is still stable. This effect may be magnified by mixed precision code, so this is another issue to consider.

4.3 Simulation results

The first difficulty in practise was the distribution of samples within the different classes as shown in Table 4.1. A challenge of the KDD Cup was that the distribution of training and testing attacks are different. Another difficulty is the unequal distribution of attack types within the training set as there were less than ten samples of an attack for several classes only. Because the training goal is to capture common patterns in the data, those attack patterns will get lost in favour of more frequent ones. The amount of samples from each class has been limited to a maximum number of 1500 instances to ease the impact during training. However, a more sophisticated technique may improve the classification performance.

As a first attempt, the binary RBM has been fed with the data using the ratios as probabilities of an underlying layer. All data was either classified as attack or normal, depending on the result of the pretraining. Therefore, we could yield a 50 % correct classification ratio regarding the selected training and validation datasets. This result did not depend on parameters like training epochs or network size. Hence, this approach did not lead to a usable classifier and has been abandoned.

Next, the quantized input representation has been analyzed. Based on a small subset, the best parameter settings haven been explored. In our experience, 25 neurons per ratio feature yield a good trade-off between feasibility and classification performance. Thereby, the 41 features of the KDD Cup dataset are converted into a 772 bit input vector. Using classes one to five only, it was possible to observe

[1] Unit in the last place (ulp) is a measure of floating point precision in computer science. A maximum of 0.5 ulp is necessary to exploit the full representational power of floating point numbers. Indeed, the most recent GPU architecture by NVidia called Femi supports floating points at full precision.

Class	Non Attack [%]	Attack [%]	# of datasets used for training	# of datasets in final classification	Misclassified test dataset
normal	0.95	0.05	2000	60591	3030
buffer overflow	0.05	0.95	30	22	1
loadmodule	0.00	1.00	9	2	0
perl	0.50	0.50	3	2	1
neptune	0.00	1.00	2000	58001	0
guess passwd	0.58	0.42	53	4367	2533
portsweep	0.00	1.00	2000	354	0
land	0.11	0.89	21	9	1
ftp write	0.33	0.67	8	3	1
back	0.00	1.00	2000	1098	0
imap	0.00	1.00	12	1	0
satan	0.00	1.00	2000	1633	0
phf	0.00	1.00	4	2	0
nmap	0.00	1.00	2000	84	0
multihop	0.22	0.78	7	18	4
warezmaster	0.11	0.89	20	1602	176
rootkit	0.15	0.85	10	13	2
named	0.29	0.71	0	17	5
xlock	0.00	1.00	0	9	0
xsnoop	0.00	1.00	0	4	0
sendmail	0.41	0.59	0	17	7
saint	0.01	0.99	0	736	7
apache2	0.00	1.00	0	794	0
xterm	0.08	0.92	0	13	1
mscan	0.00	1.00	0	1053	0
processtable	0.00	1.00	0	759	0
ps	0.19	0.81	0	16	3
httptunnel	0.02	0.98	0	158	3
worm	1.00	0.00	0	2	2
mailbomb	0.24	0.76	0	5000	1200
sqlattack	0.00	1.00	0	2	0

Table 4.4: Identification mistakes for all classes

	Class		
	Normal	Pretrained attacks	Unseen attacks
Non Attack	95%	13%	16%
Attack	5%	87%	84%
# of datasets in final training	2000	10177	0
# of datasets test classification	60591	67211	8580
Misclassified datasets	3030	8611	1373

Table 4.5: Confusion table trained vs. untrained classes

a misclassification of 2 out of 3000 (0.67 %) in the training dataset and 94 out of 3000 (3.13 %) in the validation dataset.

The research regarding the optimal layer sizes has been limited due to the required training time of the MATLAB implementation. Tests suggested that a 500-500-2000 neurons configuration yields good results. This is the same configuration used for the MNIST character classification in [24].

The confusion diagram of the final run against the whole test set is provided in Table 4.4. The training goal was set to classify attack or no attack only to handle the unseen attacks properly. A comprehensive comparison with other techniques is outlined in the next section, but we continue with a short discussion of further improvements based on those results.

Obviously, all previously truncated classes were classified 100 % correctly, so the results may be improved by using a more sophisticated equalization technique like repetition. The performance is impressive, even for many attack types not present in the training at all. Within the trained classes, most errors occur in classes with a significant higher incidence in the test data than in the training data, for example class 1, 7 and 20. Overall, a 5.1 % misclassification rate is achieved.

We summarized the recognition rate for trained and untrained connection classes in Table 4.5. The classification rate is similar for both kinds of attack, so the Boltzmann network seems to provide a good generalization of the underling attack schemes.

4.4 Comparison to other approaches

First, the detection results are compared with an *Artificial Immune System* (AIS) approach. This technique tries to mimic the characteristics of the human immune systems by generating detectors to distinguish between own and foreign cells. Key characteristics are the robustness and flexibility in terms of classification and parallelism in terms of operation. Especially the first aspect and the obvious similarity between the domains of origin and application motivated the choice. The AIS techniques try to find instances of examples, which are sensitive to malicious pattern. Those instances are called

	correct classified [%]		
	AIS 1	AIS 2	RBM
Non-Attack	50.6	70.7	95.0
Attack	99.9	90.8	86.8

Table 4.6: Correct classification for different AIS approaches compared to RBM technique

antibodies in analogy with the medical jargon. Sensitivity is defined by some similarity measure like Hamming Distance or r-Continuous Bits. Two attributes of the immune systems are used to generate antibodies in general, namely Negative and Clonal Selection. The former technique starts with a random population and retains only antibodies with a low sensitivity to the benign samples. Thus, any unknown pattern, which is most likely malicious, is identified. The idea is adequate for tasks where the target can be expressed as complement of a know subset only. The other technique, Clonal Selection, proliferates antibodies with a high sensitivity to attacks during training. Thereby, each cloned antibody is mutated to extend the scope to similar, but unseen patters so far.

Different variations have been analyzed and the final results are outlined in Table 4.6. For the first kind, labelled as *AIS 1*, an initial set of antibodies was generated using several iterations of Negative Selection and matured by Clonal Selection afterwards. Floating thresholds have been employed to allow widespread features in the beginning, but accentuate the more mature ones during training. The approach yields a good detection of malicious connections, but classifies almost every second normal connection as attack. In a detailed analysis, it was revealed that many antibodies classified everything as attack. Those antibodies could proliferate due to the over-proportional amount of malicious connections in the training dataset.

To overcome the limitations, the training procedure has been altered for *AIS 2*. First, antibodies for normal and malicious connections are trained. Moreover, the cloning and mutation rate in the Clonal Selection stage is proportional to the misclassification rate of each antibody, so bad antibodies will either be rejected or heavily altered. Thereby, ideas from function optimization are imitated. Negative Selection is discarded due to the fact that it did not improve the classification rate. Hence, the attacks cannot be considered as complementary to normal connections. This may be due to the feature extracted presentation, because results on the raw datasets suggest an improvement [20]. As a result, the classification proficiency for normal connections has been improved at the expense of a slightly worse performance for malicious samples.

The AIS performs inferior compared with the Deep Belief Network in the last column. Although the detection of malicious samples is significantly higher in the *AIS 1*-case, the normal classification ratio disqualifies the attempt. The enhancements in *AIS 2* alleviate the drawback, but the performance of the RBM for normal connections is still significantly better. At the same time, the classification ratio for attacks is almost identical. Therefore, research on immune system inspired algorithms has been limited in favour of the Restricted Boltzmann Machine.

		correct classified [%]		
		DBN	Bagged Boosting	Kernel Miner
labeled	normal	95.00	99.45	99.42
	probe	99.75	87.73	89.01
	DOS	97.95	97.72	97.57
	U2R	93.90	26.32	22.37
	R2L	42.38	10.27	7.38

Table 4.7: Correct classification on simplified KDD Cup 99 task

		predicted					predicted
		normal	probe	DOS	U2R	R2L	correlty [%]
labeled	normal	52855	5012	775	1135	816	87.23
	probe	480	3402	214	24	46	81.66
	DOS	2673	2724	165494	55494	3468	72.00
	U2R	41	65	27	58	37	25.44
	R2L	13347	578	181	194	1889	11.67
recongnized correlty [%]		76.16	28.88	99.28	0.10	30.20	

Table 4.8: Confusion matrix of DBN on KDD Cup 99 dataset

		predicted					predicted
		normal	probe	DOS	U2R	R2L	correlty [%]
labeled	normal	60262	243	78	4	6	99.50
	probe	511	3471	184	0	0	83.30
	DOS	5299	1328	223226	0	0	97.10
	U2R	168	20	0	30	10	13.20
	R2L	14527	294	0	8	1360	8.40
recongnized correlty [%]		74.60	64.80	99.90	71.40	98.80	

Table 4.9: Confusion matrix of classification by the winning entry [49]

		predicted					predicted
		normal	probe	DOS	U2R	R2L	correlty [%]
labeled	normal	60244	239	85	9	16	99.42
	probe	458	3521	187	0	0	84.52
	DOS	5595	227	224029	2	0	97.47
	U2R	177	18	4	27	2	11.84
	R2L	14994	4	0	6	1185	7.32
recongnized correlty [%]		73.95	87.83	99.88	61.36	98.50	

Table 4.10: Confusion matrix of the commercial tool Kernel Miner, scoring the 2^{nd} place [32]

The classification task has been eased for the comparison with artificial immune system. Indeed, only a subtask of the original challenge has been tackled so far by distinguishing between benign or malicious connections. Four different attack types as explained in the introduction must be identified to solve the competition. In addition, the misclassification between classes is not accounted for equally, but penalizes some confusions harder than others. Due to that, the winning entry in the original contest was the best in terms of the scoring, but not the best classifier. The second one was a bit more accurate, except for the most important class. However, the confusion is considered in favour of the score in this section, because the aim of this work is to judge the classification performance of a Deep Belief Network instead of its suitability for real-world intrusion detection systems.

The top two results of the original KDD Cup are considered as reference. Both entries use automatic feature extraction techniques to form their decisions. Other teams used expert knowledge, but yield inferior results. Those approaches haven't been taken into account as no result tables are provided.

The winning entry used C5 decision trees for classification [49]. C5 is a tree construction technique successfully used in many different areas. A commercial implementation is available with See5. Although available for such a long time, it is still used today. The algorithm is tuned for fast learning with low memory consumption, but retains a good classification performance. Those aspects are important for processing such a huge data basis, especially with the hardware available in 1999. Boosting has been used to improve the performance of the trees. In addition to that, several trees have been trained on different subsets of the database. Finally, the decisions have been combined by minimizing the conditional risk. The database for each tree has been pre-processed to moderate the dominance of attack types and reduce the training data to a feasible amount. The way used is identical to our selection scheme, although more examples per class have been picked, leading to a more unequal distribution. Moreover, duplicate datasets have been removed before. The classification results are shown in Table 4.9.

The other submission employed the self-developed commercial tool *Kernel Miner*. Again, decision trees are constructed as in the previous approach. By contrast, the data has not been cut down in a special way, but the provided 10 % database has been used for training. Deepening on the class, the data is divided into many partitions and every partition leads to a decision tree. The best trees are selected

in a final stage. The great amount of partitions is necessary because there is no obvious relationship between a partitioning and the prediction quality of the derived tree. Thus, the selection is postponed to a phase when the performance can be taken into account. Reliability and stability of trees are involved in the selection to compensate the unequal distributions and avoid overfitting. Unlike all other attempts, classifiers for each specific attack have been trained. Furthermore, the final cost of each type is embedded into the training goal. The results obtained by Kernel Miner are provided in Table 4.10.

The results by using a Deep Belief Network are shown in Table 4.8 and differ significant from the decision tree based results. Furthermore, the results for identifying attacks only are provided in Table 4.7. The data is aggregated into the same attack schemes as for the other tables.

First, consider the correctly predicted labels in the full task as depicted in Table 4.8 . Obviously, the performance for the classes *normal*, *probe* and *DOS* is worse, but the remaining classes are classified much more accurately. The distribution of the classes needs to be taken into account to explain those results, because the first classes are over-represented during training. Thus, the number of instances is truncated to a small fraction of the available data. Thereby, not all the typical patterns of those classes are present, so the DBN cannot extract the adequate characteristics.

By contrast, if the classification task is eased to tell benign and malicious connections apart only, the DBN network outperforms the tree-based approaches in all classes except for the normal connections. Especially this exception undermines the truncation based explanation. For sure, the patterns of normal traffic vary a lot, but this is not sufficiently described by such a small subset. Nevertheless, increasing the number of *normal* connections has been analyzed, but did not lead to significant improved results for the first class, but slightly impaired the performance on the other classes.

Another observation is the distribution of predictions within an attack class. The tree based techniques can eliminate some predictions completely, but the DBN assigns at least a few samples to every class. Therefore, the percentage of correctly recognized samples drops to very low values for the seldom classes. The most impressive example is the *U2R* attack type with only 58 instances in the testing set, leading to a correct recognition ratio of 0.1 %.

Finally, all classifiers fail to identify the categories *U2R* and *R2L* with an acceptable level. This issue has been further analyzed in [51]. They detect that 80 % of *U2R* and 60 % of *R2L* connections in the testing set belong to new attack types. They demonstrate significant differences in the connections by examining rules generated by C4.5 decision trees trained on the training and testing data. Indeed, both classifiers share only 6 out of 24 selected features. In the light of a Deep Belief Network, it seems to be possible to distinguish the *U2R* attacks from normal connections with a good ratio, but the labels are equally distributed in the more difficult task. The allotment may be fostered by the lack of training samples, but the DBN outperforms decision trees notably. A remarkable preference for the correct prediction is obvious especially in the *R2L* section.

Moreover, there is another remarkable issue to note for comparing the results. Because the data is extracted from an isolated network with artificial traffic generators, both, normal traffic and the attacks,

are uniform in their characteristics [9,35]. Thereby, specific values for features are very strong indicators of certain attacks. Those indicators can easily be exploited by decision trees, but lead to overfitting as the characteristics of the artificial traffic generators are extracted. Hence, the adaptation may reason the weak performance on unseen attacks.

On the other hand, the Deep Belief Network can hardly benefit from such structure due to the coarse-grained data quantization. This may justify the fact that our approach performs better on the new attacks, but worse on the known types. However, the database needs to be examined in more detail to prove this theory.

4.5 Summary

Two major aspects have been presented in this chapter. First, the application of a Deep Belief Network as described in the previous chapters has been presented. Moreover, this is the first time that a DBN has been used the domain of network anomaly detection.

The results have been contrasted with winning entries of the competition regarding benign/malicious -detection and attack type classification. The DBN outperformed the other techniques in the first task, but stood behind in the latter one. Based on the results, the DBN is sensitive to the unequal distribution of the samples for different attack types. Additionally, the input feature quantization may be a practical disadvantage. Nevertheless, the DBN technique yields encouraging results and outperformed other techniques like artificial immune systems.

The other major aspect has been the implementation. Several aspects like the data handling, memory consumption and execution speed have demonstrated the practical limitations of MATLAB in the domain of larger scale neural networks. Especially the runtime has been tackled with a GPU based acceleration component, because many simulation settings need to be analyzed to tune the setup.

It has been demonstrated that the training of a Restricted Boltzmann Machine is well-suited for GPU architectures. More than 90 % of time has been spent on matrix multiplication, one of the key applications for GPU computing. Unfortunately, the GPU component was hindered by constraints due to MATLAB again, e.g. pinned memory and data layout. Those facts suggest a native implementation to render a more detailed research and evaluation of real-world datasets possible.

This page is blank intentionally.

Chapter 5

Native GPU RBM Implementation

The developed framework for training RBMs with GPU support is presented in this chapter. The framework is written in C++ using the CUDA runtime API for GPU computing. In general, the development flow can be divided into several phases. First, interesting ideas were identified by literature reviews throughout the work. The building blocks of those ideas have been implemented in MATLAB as gold reference, so the following GPU implementation could be tested against a working example. Using MATLAB as prototype development tool for the gold implantation leads to a faster development time, because many components for the implementation and tools to examine the results are provided. Finally, the GPU code is embedded in the framework.

In terms of this chapter, the structure of and observations based on the final implementation are outlined. This is a framework developed in Visual Studio 2008 using C++ and CUDA. The most recent CUDA revisions, namely 2.3, 3.0 and 3.1, have been used during development. An NVidia Quadro FX 5800 is used as graphic card, featuring 240 stream processors, 4 GB GDDR3 Ram at a 800 MHz clock rate and 512 bit memory interface. A PCI Express x16 Gen2 port connects the graphic card to the rest of the system. Windows 7 x64 is used as operating system, so the framework is compiled for a x64 target.

The chapter is organized as follows. First, a review of related work is presented. Then, the general structure of our framework is presented, starting with the coarse grained partitioning between CPU and GPU. Next, the building blocks are visited in more details. The performance of the framework is assessed and compared in the following section. Finally, implemented extensions of the objective function for training and their practical implications are illustrated. The chapter is concluded by a short conclusion.

5.1 Related work

G. Hinton proposed Contrastive Divergence as new learning technique for the Restricted Boltzmann Machine in 2002. With this technique, the learning became feasible even for todays computers. He

published the source code to reproduce his results on the MNIST digits and Olivetti face data [23]. For a long time, this was the only public available RBM implementation.

In 2007, the machine learning group of the University of Toronto joined the NetFlix competition [52]. This was a competition based on a large-scale real-world database of movie ratings by users of an online-rental service [4]. The task was to recommend other movies to users based on their previous ratings. Although the machine learning group participated in the first year of the competition only, they could yield the best results up to that time. It is noteworthy that most groups which finally exceeded the RBM results used a blend of multiple classifiers. Thus, the performance of the RBM is still outstanding. Another interesting point is the technique used to feed data into the RBM, because the input rating is given as integer value between one and five. However, no Gaussian style RBM was used in favour of a binary one with a five neurons per rating. The technique called *Replicated Softmax* bears major similarity with the quantization used in many sections of this work.

The results encouraged many other teams to implement their own RBMs, too. In fact, the MATLAB implementation previously published leads to an infeasible execution time, so each team had to develop its own native code. The struggles during this task are illustrated by the endless number of questions on implementation and training parameters of an RBM in the NetFlix forums. Hence, some teams disclosed parts of their working RBM implementations after the competition ended in 2008.

Many researchers found interest in RBM techniques, but had to face long running simulations. Hence, several ways to speed up the training time have been examined. One approach uses reconfigurable hardware like FPGAs to yield an up to 140 times improvement [27,29]. FPGA can exploit the inherent parallelism of the training algorithm due to its great connection flexibility. On the other hand, the fine-grained adaptability may hinder the speedup due to floating point operations. Although several platforms have dedicated hardware resources to implement the core functionality, a major amount of wiring is used to implement the task.

Another novel way of accelerating the learning is GPU computing. As the GPU hardware is not as flexible as a FPGA, the algorithm must map properly onto the architecture. Indeed, this is the case as shown before. To highlight the differences in the platforms, recap that a CPU looses 80% of area on general purpose improvement techniques like dynamic scheduling and cache. On the other hand, a FPGA devotes 80% of area on wiring to provide such fine granularity. By contrast, a GPU spends most of the silicon on computational resources. Furthermore, a GPU is available in every PC, so it seems natural to analyze the speedup of a DBN on such a platform.

First results have been announced recently. In 2009, the source code of a CUDA implementation of a binary RBM called Leonard has been published [5]. There is hardly any information available, neither documentation nor comments in the sources. Regarding information given on the webpage, it originated in a research project to evaluate the performance advantage of the RBM on a GPU. It is stated, that a huge performance advantage has been observed, but no real comparison is provided. The author mentions that he yields a training rate of more than 1000 images per second on the MNIST

task with the same network sizes as Hinton used for his MATLAB implementation. Hence, he needs about 1 minute training time per epoch including all transfer times. Based on this spare information, it is conjectured that he compared his implementation to the MATLAB implementation only. This is reinforced by the fact that only features used in the MNIST example are implemented. Moreover, the development seems to have stopped at the end of 2009. Although the framework presented here is developed independently from this work, the results seem encouraging to yield good results with an optimized implementation.

A more scientific research has been done in [50]. They evaluate the prospects of evolving neural networks from the toy size domain to real-world applications using the power of GPU computing. As reference, they consider an RBM neural network and sparse coding. They claim a 13 and 72 times speed up over their fastest 3.16 GHz dual–core CPU implementation for an RBM. The performance advantage increases with the layer size and observe memory transfer time as a primary limitation. Five major factors justify those outcomes:

- Transferring greater amounts of data is more efficient as a constant setup-time is needed.
- The main operations of an RBM are more efficient on a GPU than a CPU. The amount of computations per transfer grows as the problem size increases.
- The full power of the GPU is exploited as all stream processors are used.
- Every stream processor has enough threads to execute, which is necessary for global memory latency hiding.

A convolution style RBM has been used, so the impact is moderated due to the increased ratio of computation to transfers. Indeed, the sources are not available unlike announced in the report.

During the time of writing, another GPU based convolution implementation has been published by [8, 28]. It is a Python based framework with the training as runtime-shared library with the CUDA extension. The speedup is mentioned to be close to 140 times, but is not discussed in further detail. Indeed, the training time for the most complex networks is stated to be 45 hours, so for sure, the GPU renders this research possible. The training would require approximately 262 days otherwise.

To summarize the literature review, several implementations have been proposed. Some more recent ones employ the GPU and demonstrate a major impact to render the RBM–based research feasible. Indeed, a GPU implementation seems reasonable to work with this new technique, because many parameters need to be tuned for good performance. In the next sections, the main contribution of this work is presented, namely the development of an own framework in C++.

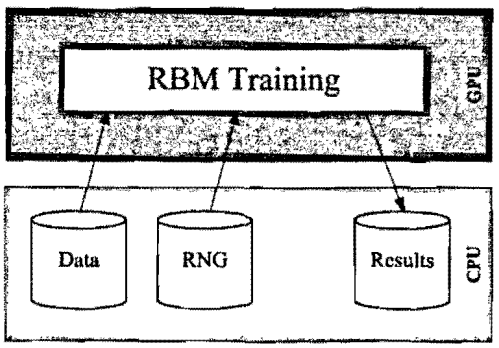


Figure 5.1: First layout of software partitioning between CPU and GPU

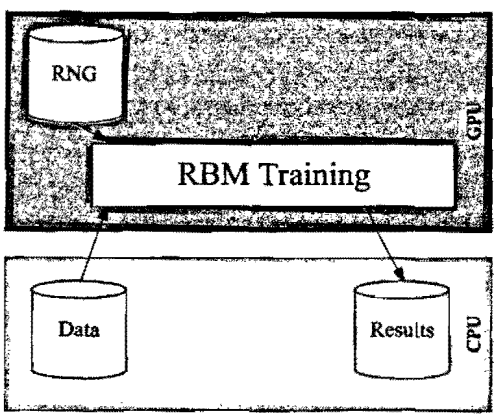


Figure 5.2: Revised layout of software partitioning between CPU and GPU

5.2 Overview

The first design decision is the partitioning of the computation between CPU and GPU. At the same time, this the most important with regard to the final performance. On a commercial scale, several months may be spent comparing the performance of several different partitioning and algorithms to solve the problem at hand. Moreover, the partitioning provides a coarse grained introduction into the structure of the framework, so the partitioning is presented first in the following sections.

There are three major independent components in the framework for learning: file reading, random number generation and RBM training. Those parts are encapsulated into own classes to employ derivatives like different readers or different RBM types easily. Most of the partitioning is predefined: The training should be on the GPU and depends on data from the other components. The data handling has to be on the CPU to read from the hard drive, so the random number generator is the only degree of freedom. The computation in the reader and random number generator class are implemented in own

threads to exploit the parallelism of the multi-core CPU. Synchronization is achieved with a multi-buffer strategy.

In the first approach depicted in Figure 5.1, both, file reading and random number generation (RNG) are handled by the CPU, while the RBM training is performed on the GPU. Although the training depends on those components, the transfers can be overlapped by GPU code execution, so another level of parallelism offered by the hardware is exploited. In addition to that, CUDA supports pinned memory, so transfers can be performed by the DMA controller without any CPU or GPU interaction. But there are other aspects as well. First, most random number generators are serial structures like shift-registers, which do not map properly on the GPU architecture. Second, several well-analyzed CPU algorithms for RNGs have been proposed and published. In the light of an RBM, this may be important to avoid any influence on the training due to false correlations. Finally, there is no use in shifting all the computation onto the GPU so the CPU cores idle meanwhile.

By examining the performance results of the first implementation, the random number generator turned out to cause a bottleneck. The GPU had to wait until new data has been delivered, so most of the GPU computing benefit was diminished. Several causes contributed to that situation: First, the computation itself on the GPU is much faster than expected based on previous experience. In anticipation of later results, an RBM with an input layer size of 12800 and an output layer size of 6144 neurons could be processed in approximately 7 ms. At the same time, at least $12800 * 6144 * \text{sizeof(float)} = 300\,\text{MB}$ of random numbers are needed for training. The transfer requires 37 ms assuming the theoretical maximum transfer speed of 8 GB/s. Second, the transfer rate was below our expectations. For the partitioning, we expected transfer rates as reported by the benchmark tool included in the SDK. The results are shown in Table 5.1. The transfer time increases to 52 ms with those rates. Transfer times of approximatly 130 ms could be observed in practice. Moreover, the transfer time for the input data should not be neglected as that data needs to transferred for each batch, too. The precise numbers are unknown due to the lack of a profiling tool at that time, but have been analyzed in more detail as it became possible. Finally, although most time is spent in BLAS functions, the provided library does not support parallel execution and transfers[1]. Those results lead obviously to a design change with the random number generator implemented on the GPU and thus the topic of the next section, the analysis of GPU based random number generators.

5.3 GPU random number generators

The majority of random number generators in use for the CPU are shift register based and don't map properly onto the architecture of the GPU. Some implementations have been proposed in the literature, but their suitability for Monte Carlo simulations is often obscure. Many papers focus on the possible speedup instead of the quality and suitability of their output. For comparison purposes, several GPU

[1]Starting with CUDA 3.1, CUBLAS supports streams for the BLAS operations. However, the partitioning design was changed by that time, so we didn't analyze the effects introduced by that release.

Memory type	Transfer direction	Transfer speed
	Device to Device	74 GB/s
paged	Host to Device	3.3 GB/s
	Device to Host	2.9 GB/s
pinned	Host to Device	5.6 GB/s
	Device to Host	4.9 GB/s
write-combined	Host to Device	5.7 GB/s
	Device to Host	4.9 GB/s

Table 5.1: Memory transfer speed for the used NVidia Quadro FX 5800. The values are obtained using the benchmark program included in the SDK.

based random number generators have been picked with regard to their applicability and performance. The three most appropriate ones have been implemented as derivations of a random number generator base class. Thus, all random number generators presented here can be used by creating instances of different classes. As all the code depends on the base class interface only, no other code changes are necessary.

CUDPP

CUDPP is a library with several data parallel primitives implemented [1]. It is not used in the rest of the work in favour of more specifically tailored functions to avoid additional actions like memory allocation or movement. For us, it is interesting in terms of the random number generator shipped with it. This is an implementation of the algorithm presented in [60], which is intended for Monte Carlo style simulations with a proven high quality of random numbers. The RNG reveals to be rather slow during profiling. This is partly due to the way some flexibility is accomplished. Each step of the underlying algorithms produces four random numbers at a time. To provide any arbitrary amount of random values, a temporary storage is allocated, filled and the demanded part copied to the destination buffer. Then, another transfer is initiated by the framework to ensure the proper padding of the data. This causes a performance penalty of $2 * \frac{37.5\,\mathrm{MB}}{77\,\mathrm{Gb/s}} = 4\,\mathrm{ms}$ in a best case estimation based on the internal transfer speed measured in Table 5.1.

rand48

The second random number generator is based on the publication by Meela [61]. The authors focus on different simulations of molecular dynamics on the GPU. A random number generator is presented as necessary part of their work. The source code is available and easy to adapt into the framework structure. Although the generator is intended and used for Monte Carlo simulations in the publication, the quality of the random numbers is not analyzed. Indeed, the molecular dynamics and the statistical mechanics of an RBM suggest similar requirements, so the generator has been taken into account.

Mersenne Twister

The Mersenne Twister random number generator is a rather new random number generator with excellent statistical properties proven with statistical tests [39]. Most notable are the very long period of $2^{19937}-1$ elements, the good distribution property and the fact that it has been designed with Monte Carlo simulations in mind[2]. Maybe one of the most sophisticated approaches to map random number generators onto the GPU is employed. A theory is derived to find initial conditions for uncorrelated generators [38]. Thus, each thread advances a random number generator independent from all the others, following the same operations on its current state. This procedure maps perfectly onto the structure of a GPU.

Indeed, this approach comes with some minor drawbacks. First, the algorithm builds on the assumption that random number sequences are independent if the generator polynomials are prime to each other. This is not proven in a strict sense, although there is strong evidence upon practical observations. Second, it takes a long time to find generator polynomials with such properties, and the time demand is increasingly non-linear with the required number of generators. The search process may take weeks, even on modern computers. Finally, the algorithm recovers only slowly from bad initialization, but this is no disadvantage in the light of this implementation, as the initial conditions are carefully picked. Thus, the initial conditions are usually found once offline and saved to a file. The random number generators need to be initialized with those conditions before first use.

For the framework, an implementation included in the NVidia GPU Computing SDK has been used. Admittedly, the code is rather proof-of-concept style, i.e. it is limited to a single call because the generator states are not saved at the end. Hence, the code had to be enhanced to fully implement the random number generator and comply with the padding scheme used throughout the application.

All the introduced random number generators seem to work without any problems in the framework, so the decision is mainly based on the speed of the algorithm. The benchmark results are shown in Table 5.3. As setup, all generators had to produce 100 times 150 MB of random numbers and the average time per call was used. Mersenne Twister turns out to be the fastest, so this one was used for all experiments demonstrated in the scope of this work.

The bad performance of CUDPP is surprising as it claims to be a high performance library, but could be confirmed in several tests. The additional transfers mentioned above are a partial explanation only. Another unexpected effect has been revealed by analysing the performance data. Memory allocations took much longer than expected. Accurate profiling results are shown in Table 5.2 and demonstrates up to 6 ms for allocation and 21 ms for deallocation. Further research suggested no direct correlation with the size of allocated data. However, this experience motivated the preallocation of memory for learning as the next design choice, because the additional time is significant compared to the learning time of several ms only. The memory requirements are discussed in more detail in a later section. At last, the

[2]It is the default random number generator used in many software products, including MATLAB.

Name	Time (μs)		
	Min	Avg	Max
cuMemAllocPitch	3.6	1419.8	6105.6
cuMemFree	40.8	1850.0	21831.9

Table 5.2: Profiling results for the CUDA runtime API functions *cuMemAllocPitch* and *cuMemFree* . Those functions are used to allocate and free memory.

exact factors for the unexpected performance of the CUDPP random number generator have not been further investigated due to the promising alternatives.

Finally, the upcoming CUDA SDK 3.2 release includes an additional random number generation library. Perhaps, those routines are more efficient. However, the library is not available at the time of writing yet, so no appraisal is possible. In anticipation of later results, any other generator won't yield significant performance improvements. The more examined suitability for Monte Carlo simulations is the major argument for using another random number library instead.

5.4 Data layout

Besides the partitioning of the components between CPU and GPU, the data layout has a major influence on the performance. All data structures have been transposed based on experience with the KDD Cup dataset. This leads to a significant performance improvement, because the lead dimension is the batch data. At the same time, training of an RBM is composed of three major actions, namely matrix multiplication, element-wise operations and batch-wise operations. Examples for the latter one are sums for each component over all samples in a batch to derive the learning signals. Those sums can be implemented very efficiently if the data is contiguous in memory using parallel reductions. The execution speed of the former operations hardly depends on the orientation of the data. Indeed, the CUBLAS library provided by NVidia follows the Fortran convention and expects the data in column major order[3]. Indeed, this is no reason, as the relationship $C = A * B \equiv C^T = B^T * A^T$ allows to use the same matrix operations. However, the code is more easily readable, because a possible source of confusion between mathematical formulation and implementation is cleared away.

At the same time, the CUBLAS library introduces another change to memory layout throughout the application. Due to the fact that the memory is partitioned across several memory banks, the performance of the matrix operations depends heavily on the way the data is divided into those banks. The library has been optimized for the case that each row of data begins at the same memory bank. Padding the data may be necessary to satisfy this property, so CUBLAS provides wrappers around the

[3]BLAS is a standardized set of functions for matrix and vector operations. It was developed in Fortran, which used column-major matrices. However, row-major matrices are common in C/C++, but most BLAS implementations follow the original data layout.

Random number generator	ø execution time
Mersenne Twister	0.52 ms
rand48	2.79 ms
CUDPP	29.11 ms

Table 5.3: Benchmark of different pseudorandom number generators on the GPU. Execution time is averaged over 100 iterations.

standard CUDA memory management functions to ensure a proper layout. There is a notable issue with the padding though. The padding data is included in the computation as well, so it must be initialized to be zero to yield correct results. Hence, there are two possibilities to guarantee the padding area to be zero. Either it is reset to zero before each CUBLAS call or it set once and never touched by any other kernel. The former approach includes another kernel call and degrades performance. In fact, the latter approach has been taken at the cost that each own written kernel has to obey the exact size of the matrix unlike the CUBLAS functions.

5.5 Kernel design

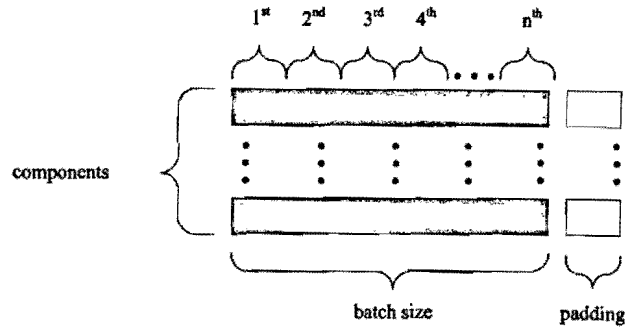


Figure 5.3: Employed data processing pattern within kernels. Each row corresponding to an input component is handled by an own thread block. Each row is processed by a loop fetching a n contiguous elements until all elements are handled.

There are several possibilities to obey the matrix size. Most examples provided by the SDK don't care about the memory layout and assign a certain contiguous pool of data to each thread block. Since such element-wise operations would affect the padding data, this pattern could not be used. Instead, a rather straight but powerful template has been used to ensure the correct data access throughout the framework: each thread block processes one row of the input data. The processing pattern is depicted in Figure 5.3. This design implies some assumptions for good performance as outlined in the next paragraphs.

First, each thread block should be busy, so the height of the matrix should be greater than the number of available thread blocks. The number of thread blocks is 30 blocks, but the actual number

varies for each graphic card model. Second, each wrap within a block should be busy, so the width of the matrix should be a multiple of 32 to fully utilize all wraps.

Typically, the width of a processed matrix is either the input layer, output layer or batch size and the height one of the remaining values. Usually, the smallest number is the batch size with 100-500 samples. At first glance, this has a major performance impact with regard to the number of threads used to process each row, because big static values will initialize lots of unused threads for small batch sizes and small values won't yield to effective global memory latency hiding. Beyond fixed thread sizes, the number of threads could be chosen dynamically for each processed matrix size. By contrast, the thread size turned out to have negligible effect in experiments. This is due to the fact that one block may process several rows at once if the core can hold further threads. Moreover, the assignment can be done better if the number of threads per block is rather small. However, this holds true if the height of the matrix is significantly larger than the number of available thread blocks only. This should be fulfilled for all realistic configurations of an RBM on the provided hardware.

The row-wise processing per blocks leads to three phases in each kernel. At first, the row-offset and eventual row-wise parameters are fetched from global RAM. In the second phase, the data is being processed, either element-wise or by summing up the values for each row. In the later case, a final step to save the sum of the current row to global RAM is needed. All the stages in this process have been examined for further speedup, because any structural benefit will pay off in all the kernels. It would have been interesting to examine the produced GPU assembly code to gain experience with the produced code, but such an option was not provided by the toolkit. Thus, profiling was the only way to measure the impact of changes.

In the first phase, row-offsets are calculated and parameters are fetched from global RAM. Those parameters can be shared among all threads within a block, so one alternative is to use shared memory to store the value. This way, one thread initializes the values and all threads will access this single value. A thread barrier must be included after the initialization to ensure a proper initialization before usage. This barrier, namely the function *__syncthreads()*, adds a penalty of about 8 computation cycles to all threads. By contrast, only a single register in shared memory is used to store the value. Another option is to define the values using the *CONST* keyword. The compiler seems to exhibit quite flexible behaviour when doing so. The value will be stored in a register for each thread if there is no register pressure and spill into shared memory otherwise. However, each thread will have its own instance of that value in the shared memory, so the shared memory usage is increased. This may limit the maximum number of threads per block if the amount of shared memory is critical for a kernel. In fact, most kernels didn't suffer from a lack of shared memory, so there is no penalty for using the *CONST* approach. Additionally, there is no synchronization penalty, so this approach has been used for most kernels. By contrast, older architectures have less shared memory per core, the best decision may be the other way round for those architectures.

In the second phase, either element-wise operations or sums are performed. For both operations, the memory access pattern has been designed to fulfill the demands of coalescing; e.g. each thread within

a wrap loads a contiguous block of memory. This block is shifted by the thread block size for the next loop iteration, until all threads in a wrap are beyond the width of the matrix. Indeed, there is another optimization for sums called parallel reduction. To put it into a nutshell, the summing operation is divided into a tree-like structure to distribute the task among the stream processors. Each stream processor computes one sum at each tree level and the size of the tree is halved. In the implementation, the threads use the scheme above to sum up the values in a shared memory buffer and this buffer is handled using the parallel reduction algorithm.

Obviously, a bigger number of threads is beneficial as there are more independent levels of the tree available. However, the impact of increased block sizes seems to be minor as row parallelism is exploited otherwise. Therefore, reductions are used with 128 or 256 threads only. By contrast, the reduction scheme improves the computation speed by one order of magnitude. The scheme has been implemented as a template function, taking the data type and the thread block size as parameters into account. In this way, an optimized version is available whenever needed. The synchronizations are included wherever needed only due to the fact that all operations within a wrap are synchronous by hardware design. So no synchronization is needed for tree sizes smaller than 32 on current GPUs. This is fulfilled most times because such a small thread size is chosen as initial size. Moreover, the employed GPU generation does not support calling functions from within kernels, so the reduction function is inlined by the compiler. Thus, any unnecessary synchronization command is removed by the optimizer.

The final step, writing the results back to the global memory, can hardly be optimized due to the fact that they are either identical to the loading scheme used in phase two for element-wise kernels, or there is no way to use coalescing, e.g. for sum operations. In fact, another template function is used to write results to global memory to support different store operations like decrement or increment throughout the code base. Resulting kernels are thus quite flexible at the expense of an additional parameter, but the kernel code remains compact. Moreover, some specialized kernels became obsolete by the increased flexibility, enhancing the readability and maintainability of the code. A minor speed penalty is introduced, but negligible due to the fact that all threads share the same code path. The penalty is offset by the fact that the return of investment in development time for optimizations is increased.

As mentioned before, the whole GPU code is realized as template functions to support single and double precision for all functions. In terms of GPU code, there is one notable point in the C++ standard which may hurt performance. Single precision constants need to end with *f*, e.g. *1.0f*. It will be treated as double precision value otherwise. In fact, the precision of computations is affected, because the computation is done in a higher precision in the later case. The CUDA compiler obeys the standard and generates proper code[4]. However, a significant amount of time will be spent on converting between the precision values if used within loops with levity. Due to the fact that the same code may be used for both, single and double precision, there is no straight way to add the proper suffix to constants. In

[4]All data is converted to single precision for hardware without double precision support, e.g. hardware revision 1.2 and below. The target hardware revision must be provided at compile time.

early stages, the precision had to be set using *define* at compile time, so the conversion was done using C++-preprocessor macros. But this is not possible for the template based implementation. Instead, values have been declared as *CONST* variables as they are converted only once, usually by the compiler at compile time[5].

Another optimization examined was the usage of constant memory to reduce the number of parameters passed to a function call. In fact, many functions use the same values which are set once for training a Restricted Boltzmann Machine. Indeed, those parameters are transferred every time a kernel needs that value. As transfer time turned out to be a major bottleneck, reducing the number of needed parameters seems quite reasonable. Constant memory is a promising alternative, because it is cached, so frequently used values can be fetched at a high speed. In fact, a cache hit is equivalent to a shared memory access. By contrast, there is no measureable effect on the performance of the program. A time demand of $0.7\,\mu s$ per parameter could be observed using the Parallel Nsight profiler at a later stage. Therefore, the impact is not measureable and the optimization has not been deployed to the kernels

Profiling results of the kernels are presented later in the scope of a complete examination of the final framework. Memory consumption as another design issue is highlighted before. As stated before, memory allocation and deallocation turned out to consume several milliseconds, independent from the size of the memory pool. Example figures obtained by a single allocation are depicted in Table 5.4. Thus, all memory arrays required during training are allocated at the beginning, to eliminate any additional dependency of the execution time on the simulation setting.

At the same time, this greedy allocation pattern may limit the suitability for large scale problems as a GPU does not support paging to extend the memory available. The issue is intensified by some simulation options like cross-validation, where several Restricted Boltzmann Machines are trained in parallel to observe effects like over-learning. The used graphic card was equipped with 4 GB of RAM, so memory was a minor concern except for very memory consuming settings. Some values are shown in Table 5.5 to provide an example. Those values represent a configuration where a whole image with 128 pixel × 100 pixel is fed into an RBM. Cross-validation is not enabled for the sake of simplicity, but a huge output layer size of 9600 neurons is used. The total amount of memory needed is 1875.52 MB for such a configuration. To be more precise, the table is split into three sections. The top section represents the connection weights and is essential for the training of an RBM. In addition to that, storage for the momentum parameters is added, as a momentum is necessary for effective learning. This fact is discussed in more detail in section 5.7, but twice the amount of memory is needed to store the previous gradients. The second section contains variables for monitoring purpose. Those are not necessary, but have been proven very useful. The last section contains buffers necessary for training again. Unlike the first section, those buffers may be shared by multiple RBMs during cross-validation.

A design decision to render the sharing of the last section of Table 5.5 possible is the division of the training database. For n-fold validation, the database is divided into n equally sized chunks, and each contiguous part belongs to one RBM. Thus, the RBMs are trained one after another for each epoch,

[5]The exact behaviour is uncertain because the GPU code cannot be examined

so the intermediate learning buffers can be reused. More than one third of memory is saved this way. On the other hand, an opportunity for an additional speedup is forgone. First, there a completely independent computations if several RBMs are trained simultaneously. Those computations may be run in different streams, so the next operation can be setup while the previous operation is still running. Thereby, initiation times are minimized. Moreover, bigger chunks of data can be transferred at a time, so transfers are minimized. Another aspect encouraged the design choice besides the ability to run large scale simulations. The CULBAS library version available at the time of development did not support streams, but executed all computations on stream zero. Due to the fact that most time is spent on BLAS operations, the benefit to spread the remaining own kernels onto different streams turned out to be negligible.

To conclude the memory design pattern, a greedy approach has been chosen to mitigate the influence of memory allocations and render large scale simulations possible. Currently, the GPU is hardly used by other applications, so almost all the GPU memory can be consumed by the RBM simulation. Other applications using GPU accelerated routines, e.g. Windows Aero, may interfere with the greedy allocation in future though. Indeed, such concerns mainly arise in the light of raw image processing without weight sharing. Closer inspection reveals that the main cause of memory usage in all three sections is the weights between visible and hidden layer. The size of those matrices may be reduced by using weight-sharing flavours like convolution style RBMs [31, 42, 50]. Especially in the domain of raw image processing, this leads to the advantageous feature of shift invariance. However, convolutional RBMs have not been implemented in the scope of this project, but excel as one option for future work.

5.6 Performance evaluation

In this section, the performance of the RBM GPU implementation is examined. As measurement, the runtime of kernels obtained by profiling and the runtime of the whole program is used and compared. Due to the design process of going from MATLAB components to a GPU implementation directly, there is no intermediate CPU implementation to compare the speedup with. Therefore, the performance is compared to the MATLAB implementation.

The MNIST digit recognition task has been used as reference in Table 5.7. This is a three-layer Deep Belief Network with a 784-500-500-2000 configuration. The MATLAB code has been executed on an Intel Core i7 926 CPU with four cores at 2.66 GHz and 4 GB RAM. The 64 bit versions of both, the operating system Windows 7 and MATLAB 2010a, have been used. The results are compared to the presented framework on an NVidia Quadro FX 5800. Overall, the GPU implementation is roughly 38 times faster than the MATLAB version. Indeed, it is obvious that the speedup is dependent on the size of the trained RBM. A closer analysis reveals the transfer time of the training data as major reason. By contrast, the computation time is proportional to the size of input and output layer. In fact, the speedup is inferior compared to gains reported in literature, so the components of the proposed framework are

Memory size	allocate	free
200 MB	5.353 ms	2.127 ms
150 MB	12.348 ms	12.348 ms
100 MB	3.057 ms	1.214 ms
50 MB	4.924 ms	33.027 ms

Table 5.4: Benchmark of memory allocation and deallocation on GPU

Variable	Width	Height	Size [MB]	Sum [MB]
VisHid	12800	9600	468.75	937.68
VisHidInc	12800	9600	468.75	
VisBias	12800	1	0.05	
VisBiasInc	12800	1	0.05	
HidBias	12800	1	0.05	
HidBiasInc	9600	1	0.04	
VisHidIncTrace	12800	9600	468.75	468.92
SumError	12800	1	0.05	
SumTrace	9600	1	0.04	
SumThreshold1	12800	1	0.05	
SumThreshold2	9600	1	0.04	
PosNegProbs	12800	9600	468.75	468.92
PosHidProbs	512	9600	18.75	
NegHidProbs	512	12800	25.00	
InputBatch	512	12800	25.00	
OutputDump	9600	512	18.75	
PosVisAct	12800	1	0.05	
NegVisAct	12800	1	0.05	
PosHidAct	9600	1	0.04	
NegHidAct	9600	1	0.04	
RandomData1	512	12800	25.00	
RandomData2	512	12800	25.00	

Table 5.5: Example of GPU memory allocation for a binary RBM. The input layer has 12800 neurons, the output layer 9600 neurons and the batch size is 512 samples at single precision.

	Batchsize		
	64	128	512
gen_sgemmNN	50.21	44.04	38.10
sgemmNT2	25.71	27.30	27.27
sgemm_main_gld_hw_ta_nb	13.62	18.87	25.68
kernel_UpdateWeight	2.62	1.94	0.57
kernel_Sum_X	2.54	1.97	0.80
kernel_Prob	2.51	2.44	3.31
RNG_rand48_get_int	1.07	1.30	1.08
memcpyHtoD	0.78	1.10	1.23
kernel_RandomFlip	0.77	0.90	1.87
kernel_UpdateWeight_Bias	0.10	0.07	0.02
BoxMullerGPURows	0.00	0.00	0.00
kernel_Shift_Rescale	0.00	0.00	0.00
memcpyDtoH	0.00	0.01	0.01

Table 5.6: Profiling results of GPU functions using a binary RBM without any monitoring. The size of the input and output layer was 728 and 500 respectively. 6000 instances were trained in total, split into batches of different sizes. All values are percentages of the total GPU runtime

		run time [s]		Speedup
		MATLAB	GPU	
Layer 1	784 x 500	1924.8	49.8	38.65
Layer 2	500 x 500	1282.4	43.2	29.68
Layer 3	500 x 2000	4947.0	125.4	39.44

Table 5.7: Comparison of execution time between MATLAB and the presented GPU framework. The time for the GPU framework includes all transfer and initialization times. It has been configured for single precision arithmetic.

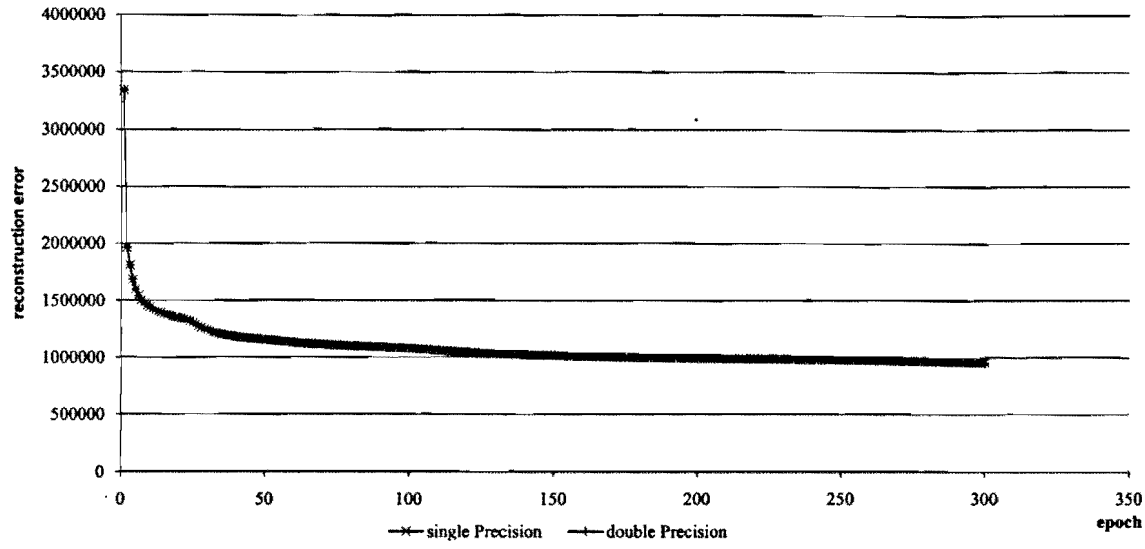


Figure 5.4: Reconstruction error obtained by single and double precision computations. Identical initial conditions have been chosen to account for the statistical nature of the network. Thus, the weights have been initialized with the same data and the random number generator with the same seed value. Indeed, only a single curve is visible in the plot because both overlay precisely.

examined in more detail in the following paragraphs. A special emphasis is on factors with influence on the execution time.

As noted before, the performance of a GPU implementation is very sensitive to a lot of parameters. The most important regarding the architecture of a GPU is the required floating point precision. To analyse the impact, the MNIST digit dataset has been processed for 300 epochs with both options. There is no handicap visible as the reconstruction error depicted in Figure 5.4 underlines. Both curves show exactly the same behaviour. This is a significant discrepancy to the observations for back propagation training. The difference can be explained by examining the training procedures in more detail. First, back propagation is much more sensitive due to the gradient computation. Moreover, quantization noise is amplified by the inverse exponential and logarithm functions employed during training. By contrast, the contrastive divergence learning remains in the log-likelihood domain. Additionally, the error does not accumulate because only a single layer is considered for an RBM. The error does not aggregate for multiple contrastive divergence iterations either, because deterministic states are picked based on the probabilities in each iteration. This feature prevails the quantization noise by orders of magnitude. Thus, a single precision arithmetic yields adequate accuracy for contrastive divergence training. The speedup for single over double precision is 4.41 times for the computation and 4.22 times including all memory transfer. Hence, all figures shown in this thesis are obtained with single precision arithmetic.

Besides the precision, the sizes of the input and output layer have a great impact on the execution time, but those are constrained by the problem at hand. A higher input or output dimension will pay

Dimension	quantity of parameters	training time [s]	parameters per second [s^{-1}]
782 x 2000	1564000	138.4	11300.57
2000 x 2500	5000000	428.6	11665.88
2500 x 3000	7500000	647.6	11581.22

Table 5.8: Comparison of execution time for different layer sizes. The considered quantity of parameters equals the number of elements in the weight matrix. Any transfer times have been neglected. It can be seen that the execution time is proportional with regard to the parameter count.

off compared to a CPU counterpart as a better utilization of the parallel GPU resources is achieved. However, the effect yields saturation rather soon as all processors on the GPU are fully occupied. From that point on, the time demand will grow with $O(D * F * B)$ with D the input layer size, F the output layer size and B the batch size for both platforms due to matrix multiplication as main complexity contribution. An example is shown in Table 5.8. Indeed, this reasonable relationship is clearly demonstrated by the results in [50], but an improvement with regard to the native implementation is claimed. Evaluating their results in the same way as in Table 5.8, we can see a major penalty for the CPU version starting at a 2304 x 16000 configuration. Because the authors don't annotate the effect, the reasons remain subject to speculations. Most likely, the CPU system is slowed down due to paging as not all network parameters can be stored in main memory.

By contrast, an important degree of freedom is the size of a batch processed at once, because fixed initiation times are alleviated with less kernel launches and memory transfers as most important factors. Therefore, the profiling results for a binary RBM implementation for different batch sizes are shown in Table 5.6. The most notable result resides in the ratio of time spent in own functions and CUBLAS routines for matrix multiplication. The top three functions are CUBLAS functions and are separated from own kernels with a line. In fact, more than 90% of time is used for the CUDA functions and can be decreased by switching to another BLAS implementation only. Using Amdahl's Law to approximate the maximum achievable speedup by tweaking the self-written functions, we receive $\text{Speedup}_{\text{max}} = \frac{1}{0.9} = 1.11$. Therefore, hardly any remarkable speedup can be expected by further optimizing the GPU code. Although research indicates that some of the CUBLAS functions may be further optimized [11], this may hardly change the ratio significantly[6].

Profiling results as shown in Table 5.6 are very common for CPU implementations, but they are not suitable to judge the quality of a GPU implementation due to the fact that several important components like transfer and initiation time are not taken into account properly. In general, CPU profiling results comparable to the ones shown in Table 5.6 are very useful to examine the bottlenecks in the CPU implementation part and judge the partitioning between CPU and GPU. Indeed, profiling is supported

[6]First, the comparison is based on an older CUBLAS library, so most likely, any significant performance improvements are included in newer releases. In addition to that, the upcoming versions are optimized for the most-recent Fermi-architecture, so the performance on our GPU model may decrease by updating CUBLAS releases.

in the used Visual Studio 2008 Professional Edition, but the evaluation of the results is reserved to the Team Suite Editions[7].

Another issue arises from the way CUDA is designed. The whole program is controlled by the host thread by initiating transfers and computations, but the execution of the GPU code is asynchronous with the CPU. Moreover, several aspects are not covered accurately, most notable the transfer times. The times shown in the profiling results convey the initiation time only, e.g. the time to set up the actual transfer. Due to the fact that pinned memory is used and the data is fixed in memory in this way, the transfers do not need much setup time. If pageable memory is used, the time may increase if the data has to be read from the swap file first. Nevertheless, the CPU program continues as soon as the transfer is initiated, so even CPU profiling may be limited to evaluate the expensive parts of the code.

In fact, there are two options to gather useful information. First, Parallel Nsight, a Visual Studio extension provided by NVidia, may be used to monitor CPU and GPU code. The results can be visualized in a timeline by the professional version of the tool. However, asynchronous transfers cannot be evaluated. The only way to yield accurate results is the usage of events provided by the CUDA toolkit. Those events are enqueued in the processing sequence and capture the time at their execution. Indeed, separated GPU timers are used, so time differences between events can be retrieved only. Hence, the cross-correlation between CPU and GPU processing cannot be examined.

In summary, the GPU introduces major degrees of parallelism to a program. The effects of this parallelism are hard to grasp with a single tool, thus require a combination of several different techniques. Moreover, this setup is very sensitive to Heisenberg effects, so any changes to examine performance may lead to completely different runtimes. For example, synchronization events may cause significant stalls of the host code and hinder latency hiding of following operations, since kernel parameters of the next calls cannot be copied to the GPU. Thus, expertise is needed to gather the information adequately and draw the right conclusion.

A first starting point is the total runtime of the program subject to different parameters. The most interesting results are depicted in Table 5.9. The total runtime of the program subject to the batch size and number of *Contrastive Divergence* iterations n is shown in Table 5.9a. As highlighted before, the runtime is anti-proportional to the size of a batch. The other dimension, the number of *CDn*-steps, rearranges the calling frequency of the functions and provides a rough idea about the percentage of time spent in different phases of the computation. Therefore, the total time must be divided into a constant and linear component as shown in Table 5.9b. For sure, the linear component represents the time necessary for a single contrastive divergence iteration. The constant time conveys the transfer time, computation of correlations and the final parameter update. This result is interesting for two reasons: First, it is based on examination of the total runtime, so any kind of Heisenberg effect can be excluded. Second, the constant time is about twice as long as a single iteration. Although it is hard to detail the profiler results into the sections precisely, this result contradicts the suggestion based on the previous tables.

[7]The Professional Editions can collect profile data, but the use is limited to automatic program optimizations.

Upon deeper analysis, the transfer time turns out to cause the major bottleneck. Roughly one third is spent on transfer, correlation estimation and contrastive divergence. As outlines above, the transfer time can only be captured using either CUDA events or synchronisation At the same time, the transfer time cannot be optimized any further as it is determined by the hardware. All possible steps to alleviate the bottleneck, namely the implementation of the RNG on the GPU and the usage of pinned memory, have been employed. Therefore, it is natural to analyse how other researchers tackled the issue. Indeed, most papers focus on the results and not on the benefits of their GPU implementation. Some performance are provided in [50] and a roughly 140 times speedup is claimed in [28]. However, both implementations use a convolutional RBM, so the ratio of computation to transfers is shifted to yield better speedups.

5.7 Extensions of the objective function

Depending on the problem at hand, the RBM training algorithm may be extended to yield a better performance. All the published results use at least a few tweaks. The most interesting enhancements with regard to classification and image processing have been integrated in the presented framework. In this section, those additions are described and the effects are outlined.

State quantisation

According to the theory of Gibbs Sampling, a binary realization must be drawn based on the probabilities of the units. However, using the probabilities for the visible units turns out to be beneficial due to less noise in the estimation. On the other hand, the hidden units should be flipped in one particular state during learning, because this causes an information bottleneck in the representation of the input. This bottleneck enforces the learning process to tune the weights to reproduce the input data distribution as closely as possible. For the expectation estimation, either binary states or probabilities may be used in both layers. Again, the latter introduces less noise, but there is no guarantee for a positive effect.

Momentum

A momentum is added to the learning in most RBMs as it leads to much superior results. The benefit roots in the sensitivity of the estimation, especially due to the approximation and the usually small batch sizes. Hence, the estimated gradient on each batch may vary a lot and result in oscillations in the updates. By contrast, a momentum distributes the change over several parameter estimations and stabilizes the direction of the descent. Any overshoot may be corrected by the next iterations before the arguable update is in full effect. Additionally, the effective weight update is much higher, so the training time is reduced, without the above-mentioned problems.

Typically, the momentum is chosen around 0.5 for the first integrations to compensate for the unstable updates. It is increased to 0.9 after a few iterations to increase the learning speed. In addition to that, the

Batchsize	CD Iterations 1	10	20
128	6.38 s	23.62 s	42.75 s
256	5.63 s	22.36 s	40.94 s
512	5.49 s	22.01 s	40.36 s
1204	5.36 s	21.78 s	40.03 s

(a) Total training time

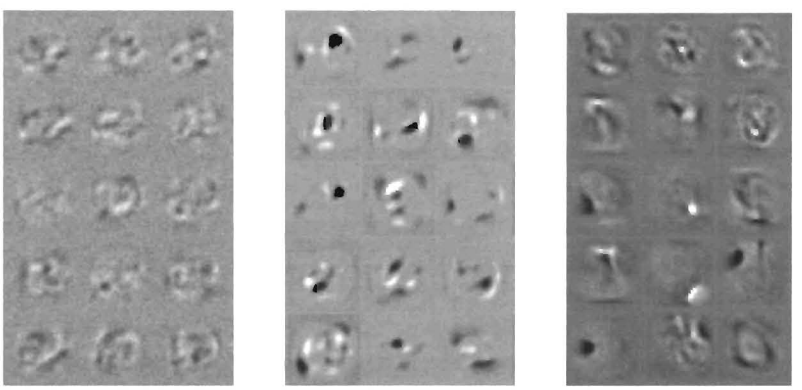
Batchsize	CD Iterations 10 offset	10 CD	20 offset	20 CD
128	4.47 s	1.91 s	4.47 s	1.92 s
256	3.77 s	1.86 s	3.77 s	1.86 s
512	3.65 s	1.84 s	3.65 s	1.84 s
1204	3.53 s	1.83 s	3.53 s	1.83 s

(b) Training time seperated

Table 5.9: Runtime requirements per epoch against batch size and iterations. The time spent for training is shown in Table a. The training time has been divided into a fixed offset time and a varying time for CD learning in Table b. Results are shown for a binary RBM without any monitoring with an input layer size of 12800 and an output layer size of 6144. The used database contained 2001 samples in total.

coarse direction of the weights coming out of complete randomness is determined in the first iterations. Once this direction is set, the high momentum encourages the weights to continue along that path. This may be a significant difference to other learning techniques with much lower momentum values.

Weight decay



(a) No norm (b) L1 norm (c) L2 norm

Figure 5.5: Examples of perceptive fields learnt on the MNIST dataset. The same settings have been used to produce the filters except for the weight decay.

Another extension with major impact is the weight decay. It penalizes large values in the weight matrix by adding an extra term to the update rule. Constraining large weights achieves several benefits.

- The input of hidden units is usually more localized by further shrinking small weights. Hence, in the case of raw image data as input, the characteristic feature of a hidden unit is much more

obvious. Additionally, the network is less prone to over fitting due to the fact that those small weights are usually caused by the selected training instances.

- It helps to recover weights being stuck in either the *on* or *off* state during the early stages of training. Usually, several epochs are needed to make those neurons useful again.
- Smaller weights lead to an increased mixing rate of the Gibbs Sampler. Because large weights cause strong correlations between single components in input and reconstruction, the approximation of the true model probability after a few Gibbs Sampling iterations is much worse. Thus, the learning signal to capture other correlations is weakened.

The update function is extended with another term, namely the derivate of a penalty function scaled with a factor. So far, L2 and L1 norm penalty functions have been implemented. The L2 norm is $\frac{1}{2}\sum_{i,j} w_{i,j}^2$ and leads to the easiest penalty term $w_{i,j}$. On the other hand, the L1 norm is defined as $\sum_{i,j} |w_{i,j}|$. In general, the L1 norm encourages even more localized features, so most of the weights are exactly zero while others grow rather large.

The influence of varying penalties is shown in Figure 5.5. The perceptive fields look very noise and unordered if no norm function is used. By contrast, the L1 norm causes very sharp and localized structures as sensitive areas. The L2 norm is in between both extremes and yields a mixture of sprawling and confined filters. However, there is more noise compared to the L1 filters.

Sparsity Target

So far, the activation probability of neurons is not controlled at all. The neurons capture frequently occurring common patterns if operated in this way. This can be justified by the learning goal of an RBM: to strengthen correlations between frequently seen patterns in the positive phase and reduce the weights for other samples in the negative phase. However, several advantages brought a great deal of attention to sparsity. First, sparsity tends to suppress over fitting if used properly. Second, rare activation probabilities are advantageous to classification, as each neuron conveys much more unique information. Indeed, the sparsity goal for the last layer should be relaxed. Additionally, RBMs tend to learn Gabor–like filters when trained with a sparsity goal. Especially multi–layer DBNs benefit from sparse activations and those Gabor-like filters in the lower layers. In addition, research indicates similarities with human perception for such networks [30].

Again, different researchers describe different techniques to achieve the sparsity goal. The easiest way is to decrease the bias of the hidden units by an offset δ, so they need a stronger input from the data in order to be triggered. In some implementations, a fixed value for δ was used [42], in others, the value was tuned according to a general sparsity goal [30]. Indeed, both ideas lead to large negative biases and large positive weights, because the training algorithm tries to compensate the negative offset by increasing the weights.

Instead, we implemented a sparsity goal penalizing biases and weights according to the difference between measured and desired activation probabilities as proposed by [25]. To achieve the goal, two steps are necessary. First, the activation probabilities need to be monitored. Similar to the weight update, a momentum is beneficial to make up for any dependency on the composition of the current training batch.

Two possibilities to tune the activation probability have been implemented. The first one is the following:

$$\text{Penalty} = \epsilon_{sp} \operatorname{sgn}(\Delta) \log(|\Delta|) \tag{5.1}$$

$$\Delta = (\text{sparsity goal} - \overline{P_A}) \tag{5.2}$$

$$\operatorname{sgn}(x) = \begin{cases} +1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases} \tag{5.3}$$

with $\overline{P_A}$ as averaged activation probability and ϵ_{sp} as sparsity cost parameter. This function forces almost all neurons to strictly obey the desired probability. By contrast, this strict goal may hamper the practical contribution of the neurons as some useful vectors may be more frequent as others. Thus, the neurons cannot employ their full power of representation.

The second penalty is inspired by [8, 28]. In fact, it takes both, the average $\overline{P_A}$ and the current activation probability P_A of each output neuron into account. It is defined to be the following:

$$\text{Penalty} = \epsilon_{sp} \left(\overline{P_A} - \text{sparsity goal}\right) P_A \left(1 - P_A\right) \tag{5.4}$$

This function relaxes the penalty for minor violations, but the desired activation goal was sometimes hard to achieve depending on the dataset.

For both penalty implementations, the same penalty value is applied to the hidden bias and weights during the update. However, a reduced cost value for the bias turned out to be beneficial. Apparently, the bias is less sensitive if the general weight matrix is penalized with the sparsity cost, because this observation contradicts the other methods focusing solely on the bias term.

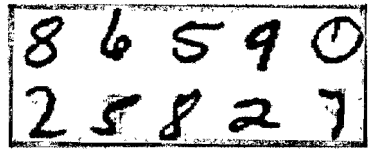


Figure 5.6: Examples of perceptive fields of an RBM trained with a very low sparsity target of 1 %. The RBM is no longer sensitive to common features, but rather typical instances of a whole digit within the training data.

Nevertheless, care should be taken for specifying the sparsity target. Obviously, if the number of hidden units times the activation target is smaller than the number of training instances, the goal becomes infeasible. However, the network will focus on instances of training data if the goal is close to this border, so the network is forced into over fitting. An example of such perceptive fields trained on the MNIST digit dataset is given in Figure 5.6. Thus, the number of output neurons should be significantly higher to render a representation in terms of small characteristic features possible. Indeed, it should mimic feature extraction techniques like SIFT if used in this way.

5.8 Summary

The proposed framework has been described from different perspectives in this chapter. First, related work has been outlined. Although some GPU implementations emerged recently, the power of a graphic processor is rather seldom used to enhance the performance of a Restricted Boltzmann Machine. The benefit of other implementations is not examined in much detail as other papers do not stress the implementation itself. Nevertheless, but the few numbers provided seem quite promising. Moreover, an efficient implementation is necessary to apply a Deep Belief Network in the domain of multimedia applications.

The presented framework has been examined from different perspectives. Starting with the coarse-gained distribution of components, the need for a GPU based random number generator has been highlighted. Several different generators have been examined and the best suited has been outlined. Based on the surprising performance of one of the generators, the memory allocation has been outlined as one of the factors which may hinder the speedup by the graphic card.

Next, the data layout has been described. Most notable, the transposed storage of the data allows an efficient implementation of many batch-wise functions. Furthermore, padding is necessary to achieve best performance with the provided BLAS implementation. The data layout leads quite naturally to the design of the kernels to process the data. A general pattern to yield good results has been highlighted and several attempts to optimize the layout are highlighted. As shown later in the scope of comprehensive performance evaluation, less than 10 % of GPU time are spent in own kernels, so the design efforts seem to pay off well. Moreover, the memory allocation has been tackled. A greedy allocation scheme has been used to overcome the time demands. The limitations and the design choice to moderate the memory consumption have been described.

Finally, the performance of the framework has been analyzed. A 40 times speedup compared with a CPU implementation has been demonstrated. This speedup is lower than the figures reported by other researchers, so the distribution of runtime has been examined in more detail. It has been revealed that transfer time consumes about one third of computation time, but all means to alleviate the impact have been employed. Thus, all GPU architectures presented in literature use convolutional RBM architectures.

Those structures naturally require more computation on given data and ease the influence of transfer times.

More than 90 % of time in the presented framework is spent on matrix operations provided by NVidia. Those functions are heavily optimized, so most of those remaining two thirds of computation time cannot be optimized any further. The remaining own kernels have been tuned for good performance given the available time. They may be further modified, but this work has been postponed due to the limited return of investment in terms of the overall performance. Instead, time has been spent on further options and flexibility to render the application in the next chapter possible.

Chapter 6

DBN for view classification

An application of Deep Belief Networks to view and genre classification is presented in this chapter. The thesis carries on the work presented in [33, 65]. View type and genre classification aims to fill the semantic gap between low level statistics and high level semantics in automatic video processing. The main contribution of the current work is to replace the *k-nearest-neighbour* algorithm used in previous research with an RBM, due to well-known limitations of the clustering technique, especially in terms of reliability and run-time.

There are several different ways to employ the RBM for this goal. First, approaches using raw image data are highlighted. Although rather expensive in terms of computation, Deep Belief Networks have demonstrated a great ability to extract meaningful features out of images. At the same time, inappropriate general features have unveiled limitations in the classification performance for similar sports. Thus, the Deep Belief Network should be able to fill the gap and lead to superior results for those sports.

Second, approaches using SIFT features are outlined, as the previous work did. That approach is tempting to limit the computation size and yield invariance to several properties, most notable shift, scale and illumination. However, the feature extraction process is time-consuming as well, and the suitability of SIFT features for a Deep Belief Network has to be analyzed.

The research outlined in this section reflects an early stage and several improvements as presented in the end may enhance the performance significantly. Therefore, the research has been limited to three different kinds of sports, namely speed-skating, snooker and swimming. Indeed, the benefits, drawbacks and future work directions can be directly derived from results with the modified database.

The organization of the chapter is as follows: First, a short summary of previous work is provided. Before presenting results, a novel method to generate codebooks using Deep Belief Networks is highlighted, as this technique is used for both, raw images and SIFT features. The results on both types of input features are shown in the next sections. Finally, a conclusion about the performance of the proposed techniques in the light of view classification is drawn.

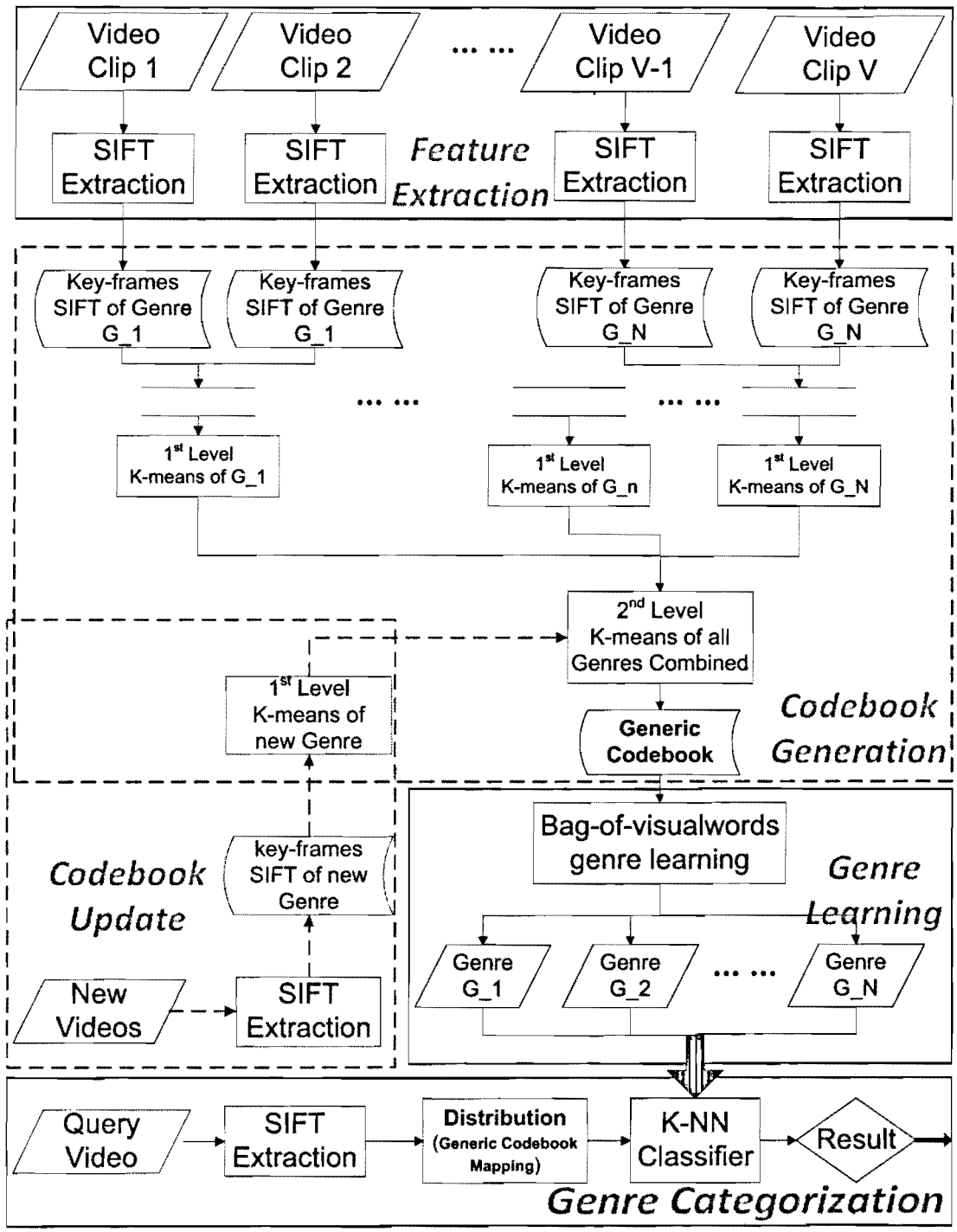


Figure 6.1: Outline of genre classification in previous work [33]. A multi-stage approach based on k-means clustering and SIFT features is employed.

6.1 Previous work

In the domain of video processing are mainly two types of features: Low-level descriptors like edges and colour and high level events like goals in soccer. Although a human can easily identify a genre or label events, this task has proven to be tough for automatic processing. At the same time, the amount of data available online grows each year at an increasing rate. This data is usually unlabeled or roughly annotated, e.g. on YouTube, indexing is necessary to keep track of the available information. Manual processing is infeasible due to the immense amount of data. View and genre classification are intended as intermediate features to fill the semantic gap and aid the higher level processing tasks. For example, a score event in different games like soccer and table tennis are completely different. Hence, they can be detected with more adapted techniques if the type of sport is known. Thereby, new application domains like the automatic audience-oriented compilation of information are rendered possible in the long term view.

Many approaches in literature employ domain-knowledge based feature detectors to distinguish between the different classes. For example, a green colour detector is used as indicator in soccer games. Obviously, the detector is sensitive to any kind of game played on a grass field, so the suitability roots in the limited selection of alternative sports to differentiate. Thereby, the application is limited to domains in the research environment with controlled conditions. A major contribution of our previous work on genre classification was the proposal of a completely domain-knowledge independent framework. Likewise, an RBM offers the promise of an efficient, domain-independent framework, and it is towards this good that we evaluate RBMs in this chapter.

Previously, to achieve the goal of a domain-knowledge independent classifier, a bag-of-visual-words approach has been examined as outlined in Figure 6.1. First, the video is decomposed into key frames and SIFT features are extracted for each frame. SIFT is a technique to detect key points in an image and summarize the characteristics of each point in a histogram based descriptor [37]. Those key points represent scale, shift, and illumination invariant features of the greyscale input image, most notable edges. The actual number of key points depends on the image. They have been selected for their invariance properties and domain-knowledge independence. In order to derive a meaningful classifier, a multi-stage approach outlined in Figure 6.1 has been employed on top of the SIFT features. Ultimately, the SIFT features are general descriptors and are not tailored to distinguish anything, so the discriminative points have to be discovered. Therefore, k-means clustering is used to reduce the 128 dimensional SIFT space to a limited set of representative patterns for each genre independently. The genre results are combined in a second round of k-means clustering to derive a final codebook. The features of each genre are projected onto that codebook, so a characteristic frequency distribution for each genre is generated. In order to classify an unknown video, the SIFT features of key frames are computed and the frequency distribution with regard to the codebook is evaluated. A k-nearest neighbour classifier based on the Kullback-Leibler divergence is employed to search for the most similar distribution among the trained genres. Roughly 80 % of testing data can be classified correctly by this framework, depending on the codebook sizes.

The bottom–up two–way k–means clustering may seem surprising at first glance, but renders this framework feasible in several aspects. First, the complexity of k–means clustering increases non–linearly with the problem size, especially taking convergence into account. Furthermore, the process is more robust and flexible to new genres because only a new class and final clustering needs to be recomputed. In general, the time demand decreases from almost a month to several weeks, in conjunction with a 10 % improvement in accuracy. Nevertheless, the computation time is rather high and the approach suffers from general drawbacks of the k–means algorithm.

The most notable disadvantage is the susceptibility to local minima and therefore the initial conditions. Several iterations with different starting points need to be cross-validated to guarantee global convergence. The practical limitation is obvious in the scope of week–lasting simulations. In addition to that, more iterations are needed for convergence if the amount of data is increased. This aspect partly reasons the performance benefit of a multi-stage approach.

Thus, the ultimate goal of this chapter is analyze ways to replace the k–means clustering with a more stable Deep Belief Network as core component. A key characteristic is shared by both, namely the application domain independence and unsupervised training. Those properties encouraged the k–means clustering approach in first place.

6.2 Code book generation

One approach to classify the view types sticks close to the outlined k–means approach, but replaces the error-prone clustering with the RBM training. The fundamental idea resides in the observation that the output of an RBM is often notably clustered with regard to final task, although the training has been completely unsupervised. An example of this observation is shown in Figure 6.2 by using the MNIST digit dataset with a completely unsupervised training. The output layer has been reduced to two dimensions using PCA to plot the data, but only 16 % of the data variation could be captured this way. All classes can be distinguished by taking further dimensions into account.

The idea can be illustrated as follows: The RBM is trained to retrieve the input data properly using a distribution of binary latent variables in the hidden layer. Thereby, those latent variables represent a basis to embody the significant characteristics of the trained data. The output layer forms a unique fingerprint to recreate the shown input, and samples of one class lead to similar fingerprints as demonstrated in the example above. The fingerprint of a new sample is compared to the average fingerprints for all available classes and the most similar one is picked. The Kullback–Leibler divergence is used as distance measurement between the probability distributions of the output layer.

Additionally, this renders a convenient way to feed SIFT features into an RBM. A key problem is the varying number of SIFT features derived from a single image, as a neural network needs a fixed number of input features. This gap can be closed by creating the fingerprint, because each SIFT feature

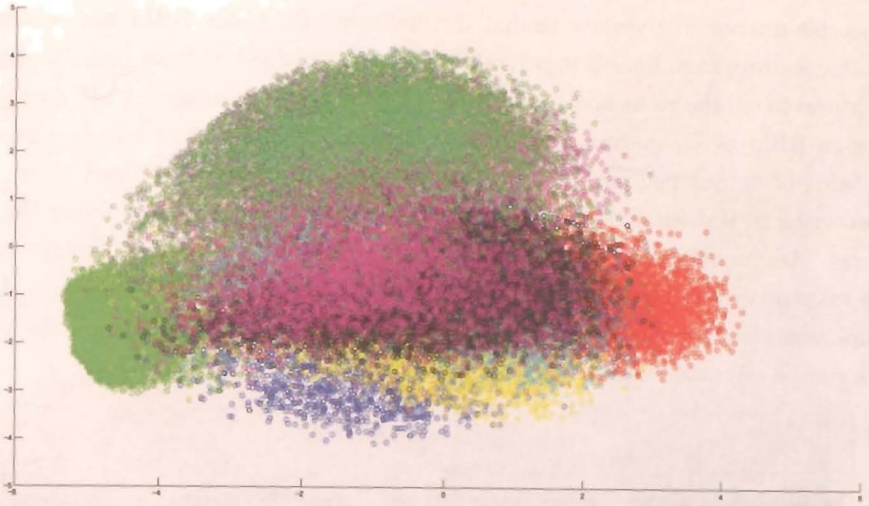


Figure 6.2: Example of data spreading achieved by an RBM. Depicted is the output layer of a 784-500-300 configuration on the MNIST character data set, trained completely unsupervised. The output is reduced to two dimensions using PCA. Each colour represents one class. Some classes overlap while others are clearly distinct. Indeed, only 16 % of the variance can be captured in the two dimensions. Most classes can be told apart taking other dimensions into account.

contributes to the final shape. Thus, the correlation of features is preserved. The previous approach achieved that by mapping each SIFT feature onto the closest codebook vector. Although very similar, it has a significant difference in the fact that only a single vector is considered in the previous case, whereas the contribution to each output is taken into account in the proposed architecture. This may lead to classification improvements comparable to hard and soft decision in coding theory.

The clustering property has been successfully used in other domains like semantic hashing [55] and as pre-stage to k–means clustering [54]. To our knowledge, Deep Belief Networks have proven to be useful for all the underlying tasks in the process, but have never been used in this way before. To summarize the idea, it is a bag of words approach, but the words to describe the data are found by the unsupervised training of an RBM. Thereby, each neuron of the output layer corresponds to a word. Finally, the similarity of the word count is evaluated like in other bag of word models, e.g. by using the Kullback–Leibler divergence of the neuron activation frequency in the output layer. Therefore, the unsupervised generation of code book vectors is a novel idea to view classification.

6.3 Raw image based approaches

First, we consider subsampled input images as features for the Restricted Boltzmann Machine. At first glance, it may seem unpractical to use raw image data for classification. This limits the application

domain to research, as training time is not constrained and expensive hardware is needed. Indeed, at least two notable ulterior motives are behind this decision. First, the RBM has proven its power to extract sensible features from images in several applications. Instead of using predefined SIFT features, adequate features to tell the views apart are learnt. Therefore, we can analyse if SIFT features are useful as input for an RBM by comparing those results. Second, training result can be visually inspected. Perceptive fields of output neuron have been shown previously for image based input. Thereby, all weights connecting to a single hidden neuron can be interpreted as a filter defining the sensitivity to input features. Additionally, the reconstruction of an image can be compared to the original image. This aspect takes on greater significance as no previous experience with RBMs are available at Ryerson and any implementation and knowledge needs to be build up from scratch. Hence, the performance of the network can be estimated in terms of figures of merit and meaningful images.

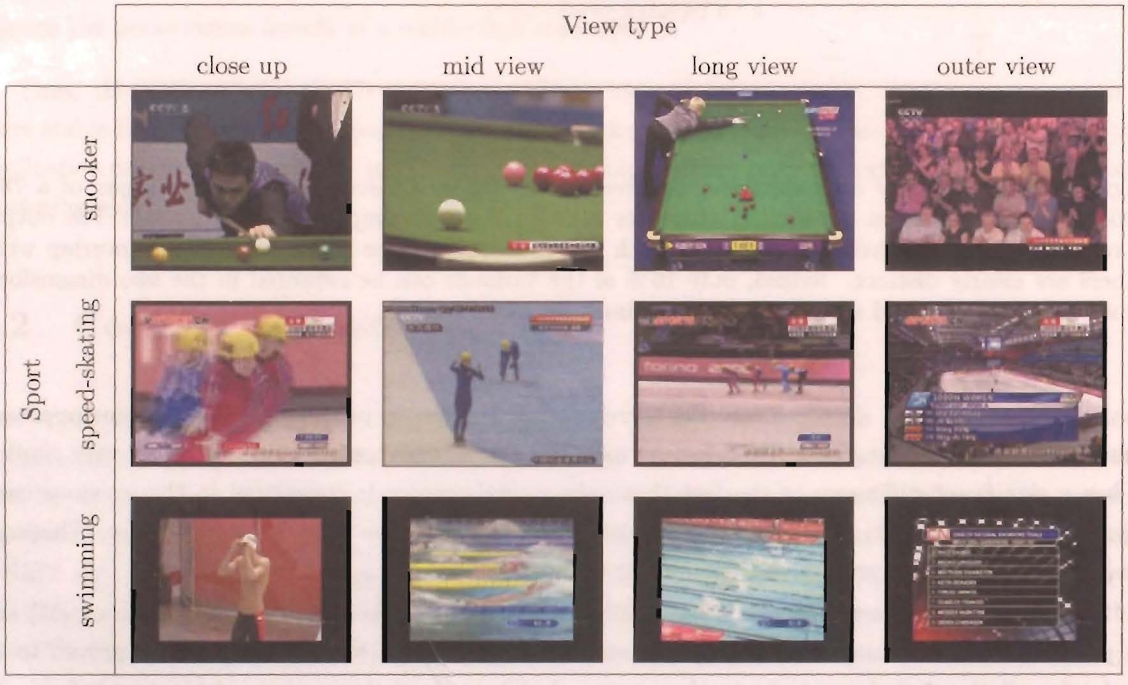


Figure 6.3: Examples of images for the view classification task.

Example images for different views in the sport genres are depicted in Figure 6.3. Generally speaking, the division into the view types is the following. A single person is shown in the close up views and the whole setting in the long view shots. The mid view focuses on parts of the setting only, but may be quite different according to the sport. The outer view is composed by result displays and audience images. The sport genre may be hard to identify for close ups and audience shots, even for humans. With regard to the view classification task, the definition of mid view is very dependent on the sport type, so a general model is hard to derive due to the manifold characteristics.

The videos are captured from a limited set of television stations. Therefore, many pictures exhibit

network logos and similar result tables and camera angles, especially for long view shots. Moreover, several stations have striking characteristics like black bars, e.g. in swimming. However, those features are identical for all examples of a particular station, so they do not contribute to the view type classification.

First, a classification as suggested in section 2.3 is analyzed, so the regenerative property of a Deep Belief Network is used to annotate each image with probabilities for each class. Logistic regression is used for a comparison as provided by LibLinear [19]. The non-linear mapping of a DBN and the large output layer size reasons the choice of a linear classifier over other well-established techniques, most notably support vector machines. In addition, the non-linear extraction of a DBN cannot be examined if a non-linear classifier is applied in the end. Moreover, logistic regression has been used successfully in similar work [28].

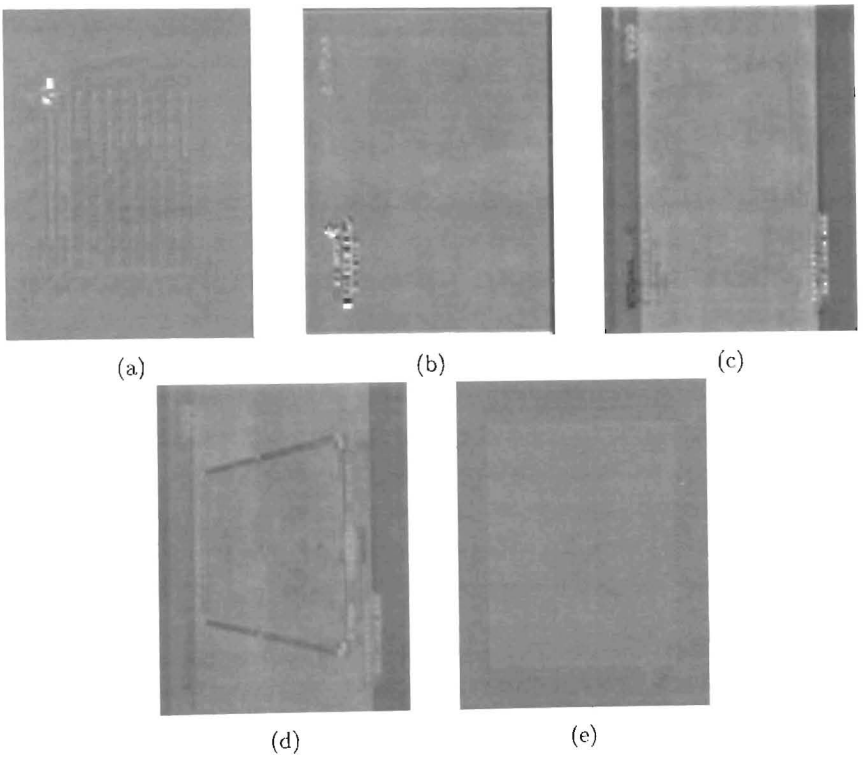


Figure 6.4: Some perceptive fields for a binary RBM trained on grayscale image data are shown as examples. The network focuses on static parts of the scenery, e.g. black borders, network logos and snooker tables.

To work on the raw image data, some suitable representation must be found. For all our work, subsampled grayscale images with a resolution of 128×100 pixel were used. Out of curiosity, the performance of a binary RBM on the greyscale image data has been evaluated. Thus, the greyscale values are treated like probabilities of an underlying layer as explained in section 2.4. Some of the perceptive fields of the hidden neurons are shown in Figure 6.4. The Deep Belief Networks performs well to extract

frequent static features like network logos and result displays. Even typical sceneries like snooker tables can be clearly recognized. At the same time, the construction of such features works quite well, too. On the other hand, dynamic content is not captured at all. Dynamic and local features are out ruled by global features. This is demonstrated by Figure 6.4e, where only the outer black borders are visible, but no features in between. The situation remained the same with regard to the size of the output layer, so issues due to overrepresentation can be foreclosed.

The same analysis has been done using a Gaussian flavour Restricted Boltzmann Machine as outlined as appropriate way to handle real-valued input data in section 2.4. Indeed, the situation remained exactly the same as described before in the binary case. Most neurons capture the static features in favour of any dynamic ones. The focus on this kind of features is demonstrated by the sheer amount of filters for a particular sport bar in Figure 6.7. This behaviour could not be altered by any means highlighted before, e.g. different weight norms or sparseness targets. Coherent results could be obtained by evaluating the reconstructions. Static features could be restored properly, but dynamic scenes are blurry at best. Thus, the classification performance is affected accordingly.

Results obtained by a Gaussian RBM and logistic regression are outlined in Table 6.1 and 6.2 respectively. The results in Table 6.1 are split into the different genres to outline the dependency of classification on the image features. The best performance with 94.5 % is achieved for far views in snooker. The whole snooker table is shown in those shots, as illustrated by Figure 6.3 or the perceptive field in Figure 6.4d. This kind of feature is reflected by many neurons, so the classification performance can be justified easily. Furthermore, comparatively good results are demonstrated for outer view scenes. Two major kind of shots constitute the outer shots view, namely the audience and result displays. The result displays for each sport are very characteristic for a certain sport due to the fact that examples for each genre have been selected from a limited number of channels. Thereby, speed-skating receives an inferior ratio among all genres due to the semi-transparent overlay of result tables. Hence, other filters specialized on the underlying long and mid views, are triggered as well and influence the decision.

The results yielded by logistic regression are shown in Table 6.2 as the baseline technique. Different parameter settings have been used to analyze the dependency on the output layer size and contrastive divergence. Two settings have been used for the later aspect, namely an approximation with one iteration of Gibbs Sampling and another one with an increasing number of iterations every 20^{th} epoch. The results are comparable with for the above technique in general, but are hardly dependent on any parameter setting. Apparently, the focus on global scale features limits the merit of logistic regression as well.

The previous section has highlighted the drawbacks due to a global scale feature extraction. Several papers solve this issue by using a convolutional RBM. The general structure of a convolutional network is depicted in Figure 6.6. The difference resides in the smaller weight matrix in conjunction with a convolutional filtering of the input image with the weights. Thereby, the filters are applied to all displacements to achieve shift invariance. An additional step is necessary to aggregate the responses of the filters in all locations. Max-pooling is the most common way to go for RBMs, so the highest

	View type			
	close up	mid view	far view	outer view
snooker	24.4 %	41.8 %	94.5 %	31.6 %
speed-skating	26.7 %	32.4 %	36.9 %	19.4 %
swimming	23.8 %	27.5 %	42.7 %	22.8 %

Table 6.1: Results of view classification using label regeneration on raw images. The results are separated into genre and view type, but only the correct classification percentage for the view classification task is shown. The results are obtained using a single layer DBN with an output layer size of 3072.

		without labels		with labels	
		CD1	CD1 + 1	CD1	CD1 + 1
Output size	2024	27.5 %	29.3 %	29.3 %	29.3 %
	3072	29.2 %	29.3 %	29.3 %	29.2 %

Table 6.2: Results of view classification using logistic regression on raw images. The results are shown for training with and without labels and different iteration counts of Contrastive Divergence. CD1+1 denotes, that the training has been started with a single iteration of Gibbs Sampling and extended with another iteration every 20^{th} epoch. Logistic regression has been used to form a decision based on the output layer of the RBM.

	View type			
	close up	mid view	far view	outer view
snooker	19.4 %	23.8 %	17.4 %	20.3 %
speed-skating	21.8 %	23.2 %	22.7 %	19.2 %
swimming	18.7 %	25.4 %	19.9 %	21.1 %

Table 6.3: Results of view classification using codebook generation on raw images. The results are separated into genre and view type, but only the correct classification percentage for the view classification task is shown as before. The results are obtained using a single layer DBN with an output layer size of 3072.

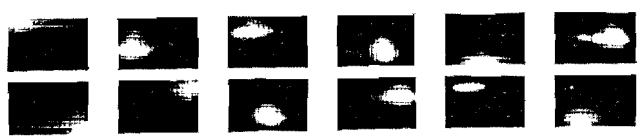


Figure 6.5: Features learned by a Gaussian RBM on sub-patches of the training database. Each patch had a size of 32 x 20 pixel and was randomly picked out of whole image. This way, the possible benefits of a weight-sharing approach can be swiftly assessed. The filters are not Gabor-like as noted by other publications for convolutional architectures, but demonstrate a preference for localized features.

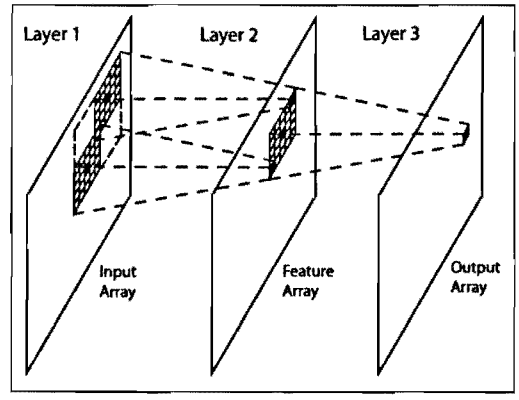


Figure 6.6: A two-layer convolutional neural network is depicted here. Different locations of origin in the lower layers to trigger one particular neuron in the output layer are exemplified. This is accomplished by a convolution between the layer and the weight matrix in mathematical terms. The required pooling step to aggregate the activity for each neuron is omitted for the clarity of presentation.

probability is picked for each neuron. Hence, the output layer size is the size of the reduced weight matrix.

To assess the suitability of a convolutional RBM, we considered an intermediate step by training on patches out of the whole image. Those patches were extracted from random locations all over the image, so the trained features should be shift invariant as well. Examples of such features are shown in Figure 6.5. Unlike other research, the features are not Gabor-like, but exhibit a strong localization like for the MNIST digit data. Therefore, we conclude that the type of extracted feature is related to the size of the perceptive window used for training. This raises the question about the optimal size of the filters, especially because Gabor-like filters are considered as natural building blocks and are highly appreciated in similar research. Nevertheless, those matters can be addressed with a proper convolutional RBM implementation only and have been postponed to future work therefore.

Nevertheless, indications for directed local features are detectable in the globally trained perceptive fields as well. Examples are shown in Figure 6.7 with result displays for swimming. The according input images look like the example provided in Figure 6.3 for the outer view in swimming. Besides the outstanding network logo, directed structures are visible. Those structures emerge especially in areas with small changes, e.g. text in the displays. Apparently, the RBM reflects those differences with directive features, although out ruled by the composition of features in the global scale so far. Hence, a convolutional feature should favour this kind of feature significantly.

As mentioned before, a code book approach has been examined, too. The results are depicted in Table 6.3. In fact, the results are a bit worse than the results before, but coincide with previous outcomes. However, the strong affinity to snooker tables does not pay off any more, but yields the worst results among all. The bag-of-words approach needs to be reconsidered for a potential explanation. The idea

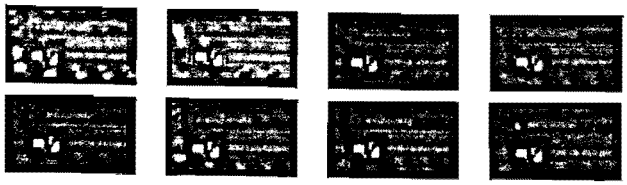


Figure 6.7: Features learned by a Gaussian RBM on edges of a swimming result display as shown before in Figure 6.3. The corner around the network logo is magnified. Structures steering in different directions are noticeable at the borders. Thus, an RBM seems to be able to extract local, directed features if trained properly.

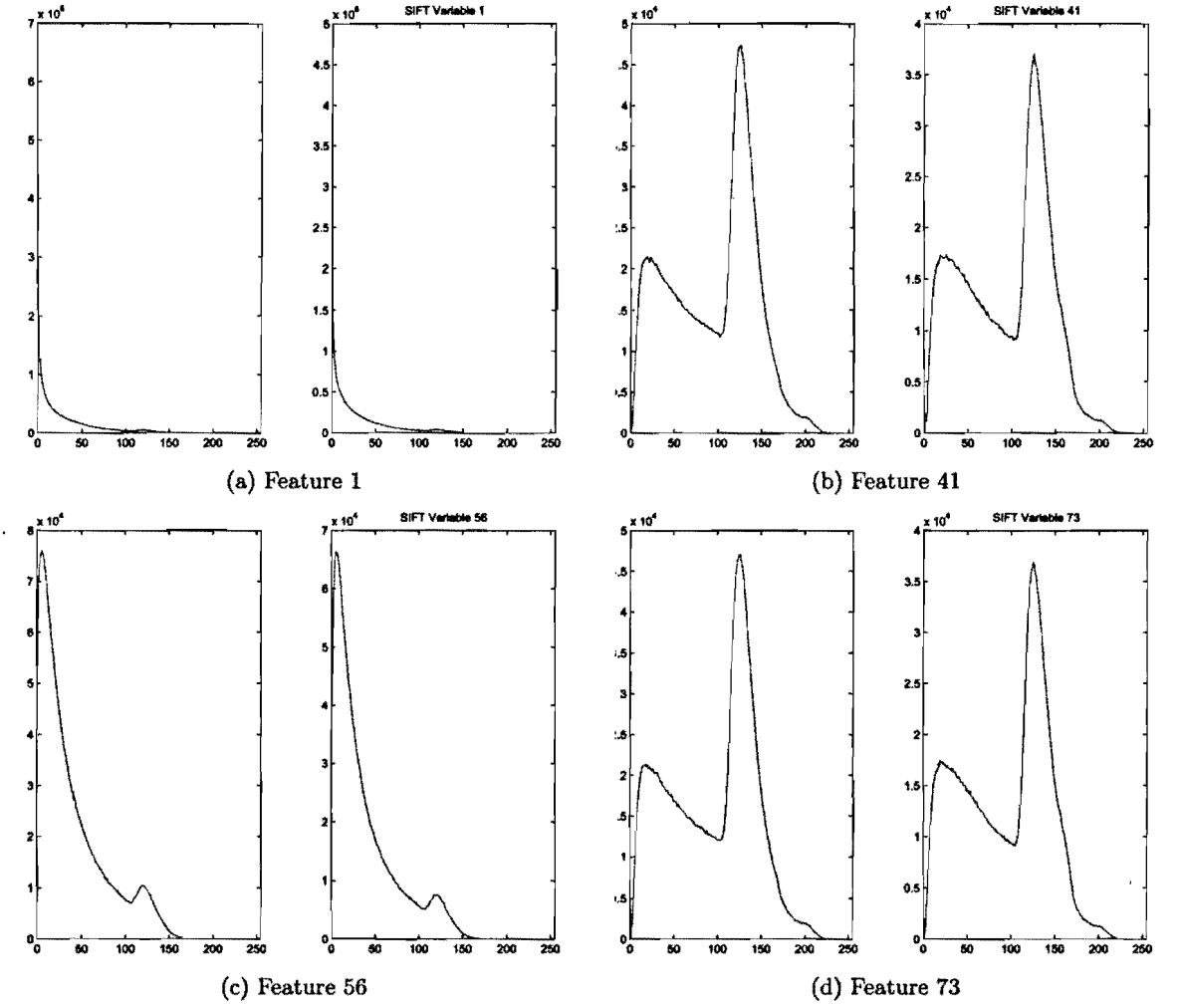
was to derive a characteristic fingerprint in the frequency of words domain. Indeed, there are lots of similar words to describe the same thing, namely a snooker table, because a lot of filters represent a typical snooker table in different channels and positions. In this context, it has to be noted that the current implementation is not invariant to shift or scale, so any variation will encourage another neuron to represent the new variation of a pool table. But because so many possibilities exist to describe the same view of a pool table, no average feature vector can be specified. Thus, the performance is deteriorated by the variety of representations for the same view.

The benefit of several layers has been highlighted several times, and also proven practical advantages in network intrusion detection. Hence, it seems reasonable to expect similar benefits for view classification as well. By contrast, the opposite effect could be observed in our experiments. Our analysis unveils the preference for static objects as reason once more. Several distinct neurons capture identical objects due to the shift variance, but they are actually uncorrelated in the sense of an RBM training as they are never triggered in common. Thereby, the discriminative features become increasingly indistinct with each additional layer as the few desired and conclusive neurons mingle with neurons of other classes instead.

6.4 SIFT based approaches

The presented raw image based approaches suffer from a focus on global scale scenery instead of meaningful local features. Hence, the decomposition is transferred to a pre-processing stage. SIFT descriptors have been used in the previous work to extract useful domain-knowledge free local features from the key frames. Thus, the qualification of SIFT descriptors for Deep Belief Networks is assessed in this section.

First, the characteristics of SIFT descriptors have been analyzed. Typical distributions of SIFT descriptor components are outlined in Figure 6.8. The distribution of the training database is shown left and the testing database right hand side of each feature display. Most notable, the distributions in the training and testing database are very similar, so any model derived from the training data should be acceptable for the testing data, too. Except for about six features, the pattern shown in Figure 6.8a



(a) Feature 1 (b) Feature 41

(c) Feature 56 (d) Feature 73

Figure 6.8: Distribution of some SIFT features in the database, on the left hand side the training set and on the right hand side the testing set. Although some exceptions are shown, most of the features are distributed like in (a). Furthermore, the distribution within each feature of the training and test set is almost identical as in the depicted examples.

is the most common. Thereby, a strong dominance for small values is obvious. However, some features have a different shape, but usually with a dominating peak as depicted in the remaining examples. The representation with a multi-bit approach as outlined before as intermediate step can hardly represent the unequal distribution detailed enough. Therefore, the slightly better results achieved by a Gaussian RBM are presented here.

However, SIFT descriptors cannot be used easily with regard to neural networks. Each key frame of the video is described by a composition of SIFT key points, with 128 elements per point. Indeed, a neural network expects a fixed size for the input features. If 128 is picked as input layer size, each SIFT

	View type			
	close up	mid view	far view	outer view
snooker	37.4	32.4	28.7	35.9
speed-skating	31.2	25.6	39.5	38.1
swimming	37.6	34.9	36.2	29.7

Table 6.4: Results of view classification using codebook generation on SIFT features of key frames. The results are separated into genre and view type, but only the correct classification percentage for the view classification task is shown as before. The results are obtained using a single layer DBN with an output layer size of 512.

descriptor is treated independent from all the others; But a single edge is hardly enough evidence for any decision, so a way to combine the descriptors needs to be found. Again, the code book approach outlined in the beginning suits the need perfectly. Thus, all SIFT features are fed as independent input vectors to achieve a separate output layer activation pattern for each. Finally, the activation frequency for the output layer considering all SIFT points of a key image is computed and used in same way as before.

Nevertheless, the results presented in Table 6.4 cannot catch up with the previous work. Although the best results obtained so far, the correct classification percentage is between 25 % and 40 % only. Indeed, a significant drift in performance for the snooker long view shots is no longer detectable, so the SIFT extraction seems to lead to a more homogenous feature extraction among all different genres.

A typical fingerprint for the different views is depicted in Figure 6.9. All plots seem very similar, so a test sample seems hard to classify to one particular view. The shape cannot attributed to a definite reason due to the abstract nature and sheer amount of SIFT descriptors. Analysis suggests the regenerative property to cause most neurons to behave alike. Because several SIFT descriptors of different views share common properties, those characteristics are captured by neurons. Hence, the generalization into common traits drives a high activation rate for most neurons independent of the view. On the other hand, characteristic descriptors get repressed by the focus on frequent patterns. As before, the results obtained with additional layers yield a slightly inferior classification performance.

In the light of local scale SIFT features, another enhancement has been considered as well. Training with a ridiculously low sparseness target yields extracted features to mimic typical instances instead of common patterns, as previously outlined in section 5.7. Thereby, the result should imitate a clustering, but hopefully without the typical instability drawbacks of the k–means technique. By contrast, the results of classification are not affected by the new settings. The effect may either root in the suitability of SIFT descriptors for Deep Belief Networks or the focus on frequent patterns, alike the previous approach. For sure, the extracted typical instances are most frequent SIFT descriptors, and therefore the least discriminative representatives.

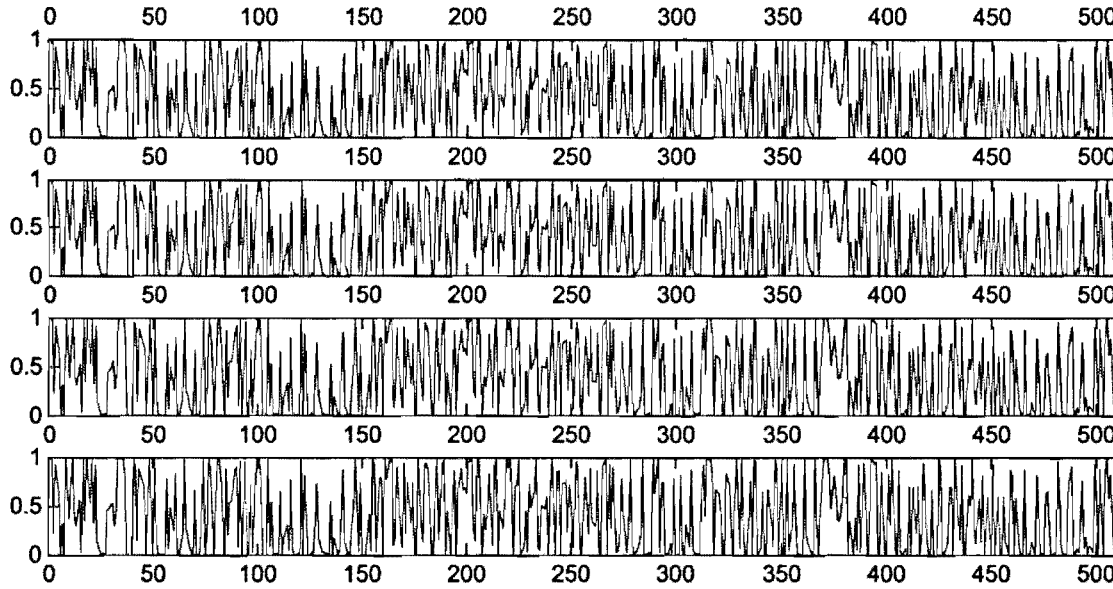


Figure 6.9: Examples of the mean distributions for class centers codebook approach. The typical fingerprints for close up, mid, long and outer view are shown from top to bottom. Apparently, the distributions are very similar.

6.5 Summary

The application of Deep Belief Networks to View Classification has been presented in this chapter. First, the previous approach has been presented as baseline technique. A practical limitation of this framework was the k–means clustering, which is prone to inferior results on the one hand side and required immense computation time on the other hand. The restrictions of the framework should be overcome by using a Deep Belief Network in favour of k-means clustering.

On a large scale, two use cases have been examined. On the one hand side, SIFT descriptors have been used as input, so the focus was on the clustering performance to derive an adopted codebook. By contrast, raw pictures have been used in the second case. Thereby, we tried to live up with previous success with regard to proper feature extraction from large-scale input [8,24,28,55].

Indeed, the results for raw images fall short of expectations due to the fact, that global scale features are extracted only and the sensitivity to variances, most notable shift, rotation and pan. Obviously, this was no issue in other research as variances have been excluded by the design of their database [24]. Nevertheless, convolutional style Deep Belief Networks have proven to tackle such variances successfully by focusing on more local features [8,28]. Precursors of convolutional networks validate the expectation as demonstrated. Thus, such techniques should be examined in more detail to proceed with the raw image based approaches. They should surpass the SIFT based frameworks due to the adapted selection

of appropriate features. The benefits of domain-knowledge independent learning are retained at the same time.

A more straight-forward kind of application has been demonstrated by using SIFT descriptors directly. The codebook approach has been used to fuse the independent descriptors in the end. To our best knowledge, this kind of technique to combine the advantages of Deep Belief Networks with the flexibility of a bag of words representation has not been analyzed before. The outcome supersedes the image based attempts, but stands behind the classification performance of the previous work. The exact reason is hard to analyze due to the abstract nature of the data, but the frequency distribution of different classes seemed to be similar. Thus, the KL-divergence measurement can hardly derive a distinguishable fingerprint. Maybe, enhancements similar to the ideas presented in [54] can mitigate the issue. Thereby, the difference between learning to reconstruct and classify data is highlighted. In the case of characters, the shape and width of the stroke needs to be captured to reconstruct the data properly. By contrast, the width of the stroke does not contribute to the meaning of the character, but distorts the decision. The bag of words technique seems to be very sensitive to this kind of inappropriate frequency components, because all basis vectors contribute equally to the KL-divergence. The analysis of ways to separate both kinds of features may be a direction for future work to improve the outcome of code books using both input features, raw images and SIFT features.

This page is blank intentionally.

Chapter 7

Conclusion

The thesis involved four major aspects, namely GPU Computing, Deep Belief Networks, the framework to combine both, and the application of the framework to view classification. The structure of the work has been chosen accordingly.

First, a comprehensive introduction into Deep Belief Networks based on Restricted Boltzmann Machines has been provided. Besides the original proposal, ways to employ those neural networks in practice have been outlined. Thereby, fundamental questions like the constitution of multi-layer networks based on the layer-wise learning algorithm and the application as classifier have been tackled. Finally, extensions to different input spaces to enhance the suitability to great variety of domains have been presented.

Next, GPU computing has been introduced, as all further chapters deal with GPU computing to a greater or lesser extent. Priority was given to the software tools, but also the underlying hardware architecture, because the software needs to be tuned to the hardware design choices to exploit any benefit. A cardinal understanding is required to reason many software design aspects presented later on.

Equipped with this foundation, Network Intrusion Detection has been presented as first use case for the Restricted Boltzmann Machine. Thereby, two aspects have been highlighted. On the one hand side, the classification results have been shown and compared to other techniques like decision trees and artificial immune systems. Our approach could outperform AIS clearly, but stood back compared to other participants in the contest. Indeed, our results exceed the other ones in a few cases, though. However, the results point out that the overall constraints of our machine learning technique resides in the unequal distribution of training examples for the different attack patterns.

The other aspect was the acceleration with a GPU implementation. A key part of the MATLAB program has been rewritten using CUDA, leading to a three times speedup. Drawbacks and limitations with regard to the embedding into MATLAB have been addressed as they lead over to the next chapter, the development of a standard framework to overcome the restrictions.

The structure of the framework has been explained, starting from the coarse-grained partitioning between GPU and CPU to the fine-grained tuning of the GPU kernels. The time requirements of the program components have been examined next by considering GPU profiling and runtime of the program. More than 90% of GPU time is spent in matrix operations provided by NVidia, so the employed optimization strategies pay off well. Indeed, the general runtime analysis reveals that about one third of total time is spent on memory transfer to the GPU. This cannot be detected by most tools because the asynchronous memory transfers are not handled properly by most profiling techniques. Nevertheless, the transfer time and matrix operations cannot be reduced any more in the scope of this thesis, so the best solution with regard to the given hardware has been presented. Finally, the speedup is compared to other implementations. A 40 times speedup can be measured compared to a MATLAB implementation, but the benefit is implicitly limited by the toy size of the problem enforced by the MATLAB memory management. By contrast, other GPU implementations report a speedup between 12 and 140 times, but all of them employ a convolutional style RBM with more computational demands per transfer. Thus, the impact of the inevitable bottleneck is less significant.

Finally, the presented framework is applied to another application domain, namely the classification of view types. The foundation was provided by the presentation of previous work. We tried to live up to that with a Deep Belief Network as core component to overcome limitations imposed by k-means clustering. Therefore, a novel technique to utilize Deep Belief Networks for code book generation has been presented. The outcome of this attempt has been compared to other Deep Belief Network classification techniques in the following sections of the chapter. The DBN could not match up with the previous results, but solutions to overcome some difficulties are outlined.

7.1 Contributions of the Thesis

Implementation of back-propagation style fine-tuning on a GPU

We accelerated the learning of a discriminative Deep Belief Network to render research on Network Intrusion Detection feasible. The emphasis was on the most time-consuming part of learning, namely the gradient computation during back propagation fine tuning. Although the task itself is well-suited to a GPU, the MATLAB framework imposed some restrictions. Nevertheless, a three–times speedup has been achieved. Thereby, the training time for a Network could be reduced from days to several hours.

Application of the Restricted Boltzmann Machine to network intrusion detection

Based on the extended MATLAB implementation outlined before, the task of Network Intrusion Detection has been tackled. To our best knowledge, this was the first time that Deep Belief Networks have been used in this application domain. We could outperform an artificial immune system on the training database significantly. By contrast, the original winning entries yield a less uniform comparison. The classification ratio of the Deep Belief Network is more equal than with the other techniques. Thus, the classification performance is significantly better at the weak

spots of the other approaches at the expense of other attack patterns. Overall, the outcome of the experiment demonstrates the superb generalization capabilities of the chosen technique.

Implementation of a framework for learning Restricted Boltzmann Machines on GPUs
The speedup achieved with CUDA above pointed out the potential and suitability of a GPU for the training of Deep Belief Networks. However, the acceleration was limited by MATLAB due to software restrictions in terms of supported features and network sizes. Thus, we implemented a native framework with flexibility and speed in mind. The maximum speedup is roughly about 40 times and renders research on raw images possible in this way. The most frequently used kinds of RBM layers, binary and Gaussian, have been implemented and extended with many extensions viable for multimedia processing as well. Thus, a comprehensive foundation for further simulations and research has been created.

Application of the Restricted Boltzmann Machine to view classification
Finally, the proposed framework has been applied to view classification. Again, this is a novel domain of application for Deep Belief Networks. Therefore, a novel technique to generate code books with Deep Belief Networks has been developed and compared to traditional classification setups. The novel technique performed best among the considered Deep Belief Network based approaches. So far, the k-means based technique performs better than Deep Belief Networks in this task. Nevertheless, we presented conclusive reasons and highlighted the way to overcome the restrictions. We undermined our assessment of future directions with results of precursors as well.

7.2 Future work

In the scope of this thesis, we caught up with the last ten years of research with Deep Belief Networks and extend research to new application domains by accelerating the training with a GPU. Nevertheless, the RBM is a versatile core component with lots of possible applications. The improved training time renders new research in other fields of multimedia-processing feasible.

In addition to that, there are some possible directions for future work with regard to the presented framework and its applications. Most outstanding is the convolutional Restricted Boltzmann Machine for several aspects. In terms of learning, it will lead to local features derived from raw images, as we demonstrated. Additionally, convolutional RBMs seem to be a promising enhancement to cope with shift variance, because most recent work on raw images employs this kind of architecture successfully. Furthermore, the maximum speedup is raised as the transfer time as major bottleneck is alleviated.

A more comprehensive analysis should be conducted for code book generation to overcome the current restrictions. The feature extraction focuses on regenerating the input by design. Based on current results, the extracted properties do not necessarily contribute to the demanded classification. At the same time, the KL-divergence seems to be quite sensitive to those components. Thus, one

possible direction for future work are different distance measurements, assessing each component with a confidence. By contrast, extending the training function of a RBM to separate between discriminative and regenerative kind of features can be examined, akin to ideas in [54]. Both approaches aim at the same goal, but modify different stages in the code book generation.

Moreover, the appropriate input data format should be examined in more detail. We started to contrast SIFT descriptors with raw images in the scope of this work, but the extension highlighted above may lead to completely different results. In general, we expect better results by exploiting the adopted features extracted by a RBM. At the same time, the RBM imposes a computational expensive way to generate features. Thus, a comparison between different feature extraction techniques with regard to performance and time demands is another direction for future work.

Furthermore, CUDA has been used to employ the GPU into the program. Although this decision was reasonable because it was the most mature way to integrate GPUs into CPU programs, OpenCL has been established as standard meanwhile. Promising benefits are the inherit flexibility to run on GPUs by different manufacturers, but also on CPUs. Thus, the comparison between CPU and GPU performance does not require a dedicated CPU development. Upcoming products by AMD are of particular interest, as they are based on a single-die combination of CPU and GPU, and may mitigate the transfer time penalty. In fact, CUDA is not able to run on those new architectures, so a refactoring of the code to OpenCL is necessary.

Finally, the Restricted Boltzmann Machine is a core component. With the power of GPU computing and the versatility of the presented framework, the foundation is laid to exploit the performance is new application domains. By design, the framework does not depend on any particular data presumptions, so it may be applied to a broad range of input features, from images to abstract feature descriptors. The investigation of advantages, but also limitations and solutions to facilitate Deep Belief Networks may be one of the most exciting and future–oriented issues to follow this fundamental work.

Bibliography

[1] CUDPP - CUDA Data Parallel Primitives Library. [Online]. Available: http://code.google.com/p/cudpp/

[2] The MNIST data set. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[3] (1999) KDD Cup 1999 - Task description and database. [Online]. Available: http://www.sigkdd.org/kddcup/index.php?section=1999

[4] (2006) Netflix Prize - Rules and information. [Online]. Available: http://www.netflixprize.com/

[5] (2009) Leonard - a CUDA RBM implementation. [Online]. Available: http://github.com/IanCal/leonard/wiki

[6] (2009) Supercomputing 2009 CUDA Tutorial. [Online]. Available: http://www.nvidia.com/object/SC09_Tutorial.html

[7] (2010) Parallel NSight - Visual Studio integration fur CUDA development. [Online]. Available: http://developer.nvidia.com/object/nsight.html

[8] Alex Krizhevsky, "Convolutional Deep Belief Networks on CIFAR-10," Department of Computer Science, University of Toronto, Tech. Rep., Aug 2010. [Online]. Available: http://www.cs.toronto.edu/ kriz/conv-cifar10-aug2010.pdf

[9] N. Athanasiades, R. Abler, J. Levine, H. Owen, and G. Riley, "Intrusion detection testing and benchmarking methodologies," in *Information Assurance, 2003. IWIAS 2003. Proceedings. First IEEE International Workshop on.* IEEE, 2003, pp. 63–72.

[10] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units.* ACM, 2010, pp. 94–103.

[11] S. Barrachina, M. Castillo, F. Igual, R. Mayo, and E. Quintana-Ortı, "Evaluation and tuning of the level 3 CUBLAS for graphics processors," in *9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing–PDSEC'08*, 2008.

[12] Y. Bengio and O. Delalleau, "Justifying and generalizing contrastive divergence," Département d'Informatique et Recherche Opérationnelle, Université de Montréal, Tech. Rep. 1311, 2007.

[13] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in Neural Information Processing Systems 19 (NIPS'06)*, 2007, pp. 153–160. [Online]. Available: http://www.iro.umontreal.ca/ lisa/pointeurs/BengioNips2006All.pdf

[14] M. Browne and S. Ghidary, "Convolutional neural networks for image processing: an application in robot vision," *AI 2003: Advances in Artificial Intelligence*, pp. 641–652, 2003.

[15] S. Brugger, "KDD Cup'99 dataset (Network Intrusion) considered harmful," *KDnuggets newsletter*, vol. 7, no. 18, p. 15, 2007.

[16] M. A. Carreira-Perpignan and G. E. Hinton, "On contrastive divergence learning," in *Artificial Intelligence and Statistics*, 2005.

[17] Z. Chen, S. Haykin, J. J. Eggermont, and S. Becker, *Correlative Learning: A Basis for Brain and Adaptive Systems (Adaptive and Learning Systems for Signal Processing, Communications and Control Series)*. Wiley-Interscience, 2007.

[18] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *J. Mach. Learn. Res.*, vol. 11, pp. 625–660, 2010.

[19] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

[20] C. Haag, G. Lamont, P. Williams, and G. Peterson, "An artificial immune system-inspired multiobjective evolutionary algorithm with application to the detection of distributed computer network intrusions," in *Proceedings of the 6th international conference on Artificial immune systems*. Springer-Verlag, 2007, pp. 420–435.

[21] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Upper Saddle River, NJ: Prentice Hall, 1999, 2nd edition.

[22] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural Computation*, vol. 14, p. 1711–1800, 2002.

[23] G. E. Hinton, S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, pp. 1527–1554, 2006.

[24] G. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, p. 504, 2006.

[25] G. Hinton, "A practical guide to training restricted boltzmann machines," Department of Computer Science, King's College Rd, Toronto, University of Toronto M5S 3G4, Canada, Tech. Rep. UTML TR 2010-003, Aug 2010.

[26] M. Houston and N. Govindaraju. (2007) SIGGRAPH 2007 GPGPU Course. [Online]. Available: http://gpgpu.org/s2007

[27] S. Kim, L. McAfee, P. McMahon, and K. Olukotun, "A highly scalable Restricted Boltzmann Machine FPGA implementation," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on.* IEEE, 2009, pp. 367–372.

[28] A. Krizhevsky, "Learning multiple layers of features from tiny images," MSc. thesis, Dept. of Computer Science, University of Toronto, April 2009. [Online]. Available: http://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf

[29] D. Le Ly, "A High-Performance, Reconfigurable Architecture for Restricted Boltzmann Machines," Ph.D. dissertation, University of Toronto, 2009.

[30] H. Lee, C. Ekanadham, and A. Ng, "Sparse deep belief net model for visual area V2," *Advances in neural information processing systems*, vol. 20, 2007.

[31] H. Lee, R. Grosse, R. Ranganath, and A. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in *Proceedings of the 26th Annual International Conference on Machine Learning.* ACM, 2009, pp. 609–616.

[32] I. Levin, "KDD-99 classifier learning contest LLSoft's results overview," *ACM SIGKDD Explorations Newsletter*, vol. 1, no. 2, pp. 67–75, 2000.

[33] L. Li, N. Zhang, L. Duan, Q. Huang, J. Du, and L. Guan, "Automatic sports genre categorization and view-type classification over large-scale dataset," in *Proceedings of the seventeen ACM international conference on Multimedia.* ACM, 2009, pp. 653–656.

[34] F. Lionetti, "GPU accelerated cardiac electrophysiology," MSc. thesis, Dept. Computer Science, University of California, San Diego, 2010.

[35] R. Lippmann, J. Haines, D. Fried, J. Korba, and K. Das, "The 1999 DARPA off-line intrusion detection evaluation," *Computer Networks*, vol. 34, no. 4, pp. 579–595, 2000.

[36] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, *et al.*, "Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation," in *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition*, vol. 2. Citeseer, 2000, pp. 12–26.

[37] D. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[38] M. Matsumoto and T. Nishimura, "Dynamic creation of pseudorandom number generators," *Monte Carlo and Quasi-Monte Carlo Methods*, pp. 56–69, 1998.

[39] Matsumoto, M. and Nishimura, T., "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[40] J. McHugh, "Testing intrusion detection systems: A critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, pp. 262–294, 2000.

[41] V. Nair and G. Hinton, "Implicit mixtures of restricted Boltzmann machines," *Advances in neural information processing systems*, vol. 21, 2008.

[42] M. Norouzi, M. Ranjbar, and G. Mori, "Stacks of convolutional restricted boltzmann machines for shift-invariant feature learning," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, jun. 2009, pp. 2735 –2742.

[43] *CUDA CUBLAS Library 3.1*, NVidia, May 2010.

[44] *NVIDIA CUDA 3.1*, NVidia, June 2010.

[45] *NVIDIA CUDA C - Best Practices Guide 3.1*, NVidia, May 2010.

[46] C. Oei, G. Friedland, and A. Janin, "Parallel training of a multi-layer perceptron on a gpu," Department of Computer Science, International Computer Science Institute, Tech. Rep. TR-09-008, Oct 2009.

[47] S. J. Patel. (2010) ECE 498 AL : Applied Parallel Programming. [Online]. Available: http://courses.engr.illinois.edu/ece498/al/

[48] J. Pettersson and I. Wainwright, "Radar Signal Processing with Graphics Processors (GPUs)," MSc. thesis, Dept. of Computational Science, Uppsala University, January 2010. [Online]. Available: http://www.hpcsweden.se/files/RadarSignalProcessingwithGraphicsProcessors.pdf

[49] B. Pfahringer, "Winning the KDD99 classification cup: bagged boosting," *ACM SIGKDD Explorations Newsletter*, vol. 1, no. 2, p. 66, 2000.

[50] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*. New York, NY, USA: ACM, 2009, pp. 873–880.

[51] M. Sabhnani and G. Serpen, "Why machine learning algorithms fail in misuse detection on KDD intrusion detection data set," *Intelligent Data Analysis*, vol. 8, no. 4, pp. 403–415, 2004.

[52] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted Boltzmann machines for collaborative filtering," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, p. 798.

[53] R. Salakhutdinov, "Learning deep generative models," Ph.D. dissertation, Dept. of Computer Science, University of Toronto, Sep 2009.

[54] Salakhutdinov, R. and Hinton, G., "Learning a nonlinear embedding by preserving class neighbourhood structure," in *AI and Statistics*, vol. 3, no. 5. Citeseer, 2007.

[55] Salakhutdinov, R. and Hinton, G., "Semantic hashing," *International Journal of Approximate Reasoning*, vol. 50, no. 7, pp. 969–978, 2009.

[56] M. Shyu, S. Chen, K. Sarinnapakorn, and L. Chang, "A novel anomaly detection scheme based on principal component classifier," in *Proceedings of the IEEE foundations and new directions of data mining workshop*. Citeseer, 2003.

[57] J. Susskind, G. Hinton, J. Movellan, and A. Anderson, "Generating facial expressions with deep belief nets," *Affective Computing, Emotion Modelling, Synthesis and Recognition*, pp. 421–440, 2008.

[58] Sutskever, I. and Tieleman, T., "On the convergence properties of contrastive divergence," in *Proc. Conference on AI and Statistics (AI-Stats)*, 2010.

[59] T. Tieleman, "Training restricted Boltzmann machines using approximations to the likelihood gradient," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 1064–1071.

[60] S. Tzeng and L.-Y. Wei, "Parallel white noise generation on a gpu via cryptographic hash," in *Symposium on Interactive 3D Graphics and Games*, 2008, p. 79–87. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=70502

[61] J. van Meela, A. Arnolda, D. Frenkelb, S. Zwartc, and R. Bellemand, "Harvesting graphics power for MD simulations," *Molecular Simulation*, vol. 34, no. 3, pp. 259–266, 2008.

[62] M. Vladimir, V. Alexei, and S. Ivan, "The MP13 approach to the KDD'99 classifier learning contest," *ACM SIGKDD Explorations Newsletter*, vol. 1, no. 2, pp. 76–77, 2000.

[63] M. Welling, M. Rosen-Zvi, and G. E. Hinton, "Exponential family harmoniums with an application to information retrieval," *Advances in Neural Information Processing Systems*, vol. 17, 2005.

[64] M. Zechner and M. Granitzer, "Accelerating K-Means on the Graphics Processor via CUDA," in *Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on*. IEEE, 2009, pp. 7–15.

[65] N. Zhang and L. Guan, "An efficient framework on large-scale video genre classification," in *Proc. IEEE Workshop on Multimedia Signal Processing, Saint Malo, France*, 2010, pp. 505–510.

This page is blank intentionally.