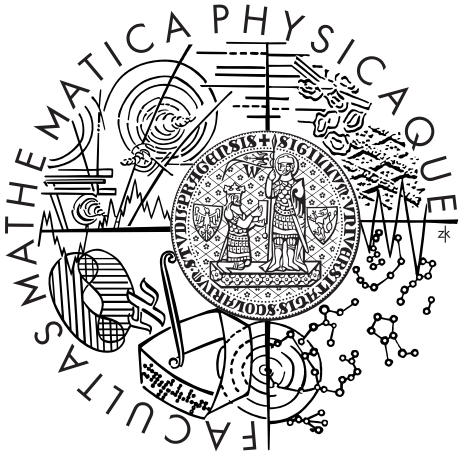
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Ján Vojt

Deep neural networks and their implementation

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: doc. RNDr. Iveta Mrázová CSc.

Study programme: Informatics

Specialization: Software Systems

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, dated ..... Author's signature

Title: Deep neural networks and their implementation

Author: Bc. Ján Vojt

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: doc. RNDr. Iveta Mrázová CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Deep neural networks represent an effective and universal model capable of solving a wide variety of tasks. This thesis is focused on three different types of deep neural networks – the multilayer perceptron, the convolutional neural network, and the deep belief network. All of the discussed network models are implemented on parallel hardware, and thoroughly tested for various choices of the network architecture and its parameters. The implemented system is accompanied by a detailed documentation of the architectural decisions and proposed optimizations. The efficiency of the implemented framework is confirmed by the results of the performed tests. A significant part of this thesis represents also additional testing of other existing frameworks which support deep neural networks. This comparison indicates superior performance to the tested rival frameworks of multilayer perceptrons and convolutional neural networks. The deep belief network implementation performs slightly better for RBM layers with up to 1000 hidden neurons, but has a noticeably inferior performance for more robust RBM layers when compared to the tested rival framework.

Keywords: multilayer neural networks, convolutional neural networks, deep belief networks

Název práce: Hluboké neuronové sítě a jejich implementace

Autor: Bc. Ján Vojt

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: doc. RNDr. Iveta Mrázová CSc., Katedra teoretické informatiky a matematické logiky

Abstrakt: Hluboké neuronové sítě jsou efektivní a univerzální model schopný řešit širokou škálu úloh. Tato práce je zaměřena na studium tří různých typů hlubokých neuronových sítí – vícevrstvý perceptron, konvoluční neuronové sítě, a sítě typu DBN (deep belief). Všechny popisované modely hlubokých neurónových sítí jsou naimplementovány na paralelní hardvérové architektuře, a otestovány pro různa nastavení architektury sítě i jejích parametrů. Implementovaný systém je doplněn detailní dokumentací softvérového návrhu a popisem použitých optimalizací. Efektivitu implemenetovaného frameworku dokládají i výsledky provedených výkonnostních testů. Významnou součást práce představuje i testování dalších existujících frameworků s podporou hlubokých neuronových sítí. Porovnání ukazuje, že framework vytvořený v rámci této práce dosáhl lepších výkonnostních výsledků než testované konkurenční implementace vícevrstvých perceptronů a konvolučních neuronových sítí. Implementace sítí typu DBN dosahuje v porovnání s konkurenční implementací mírně lepších výkonnostních výsledků pro RBM vrstvy o velikosti do 1000 neuronů, ale znatelně slabších výkonnostních výsledků pro robustnější RBM vrstvy.

Klíčová slova: vrstevnaté neuronové sítě, konvoluční neuronové sítě, hluboké sítě

ACKNOWLEDGEMENTS

I would like to thank my supervisor doc. RNDr. Iveta Mrázová, CSc for her guidance, the considerable time spent on consultations, proof reading, and for inspiring me to start the work on this thesis in the first place.

I would like to acknowledge the academic and technical support of the Charles University in Prague, namely for the provided hardware used for evaluating various deep learning frameworks. Additionally, I would like to thank Mgr. Pavel Semerád for maintaining the provided machines, and for promptly resolving the arose issues.

Special thanks to Mgr. Jana Hajduová, whose patience and support gave me the strength to work on this thesis.

Table of Contents

1. Introduction

The first known computational model of neural networks was formalized in 1943 by neurophysiologist Warren McCulloch and mathematician Walter Pitts [25]. Their work discusses how neurons in the brain might work. Their mathematical model marks the creation of a new field of study - Artificial neural networks (ANNs). The motivation for the research at this point in time was to provide an insight into the functionality of the brain.

A decade later, computers became powerful enough to simulate a hypothetical neural network. Eventually, researchers realized the pattern-matching and learning capabilities of neural networks could allow them to address a variety of problems which were difficult to be solved effectively using conventional methods and algorithms.

The first neural network model solving a real-world problem was developed by Bernard Widrow and Marcian Hoff of Stanford in 1959. Its name was ADALINE, and could recognize binary patterns. While reading a stream of bits from a phone line, it could predict the next bit, which could eliminate echoes from the stream. The research slowed down in the 1960's, because of the book Minsky and Papert published. They demonstrated that the simple two-layer perceptron is incapable of usefully representing or approximating functions outside a very narrow class [26]. Although they left the possibility of better performance open, their claims halted research of multilayer feed-forward networks for several years.

The 1980's brought a boom in the field, and showed that some of the perceptron limitations may be overcome [18]. This is mainly attributed to the back-propagation algorithm. It was invented a decade sooner, but the first computer experiments demonstrating it can generate useful representations were published in 1986 [29]. As a consequence, artificial neural networks proved to be a powerful tool able to solve complex problems such as perception, concept learning, the development of motor skills, voice recognition, etc.

The boom in the field continued throughout 1990's, however there was still a problem with the performance. Building more robust networks took weeks, even months to train on contemporary hardware. The situation changed after leveraging new learning algorithms and parallel computing to speed up the learning process. It allowed to construct deeper ANN models capable of recognizing very complex and abstract patterns.

The parallel architecture found its use in the gaming industry. Gamers represented a big market, which provided generous funding for developing high-performance chips with parallel architecture. In the past decade, these high performance chips became relatively cheap, and hence available for masses. This greatly fueled the research in the artificial intelligence field.

The achievements are astonishing. Artificial neural networks found applications in a wide range of fields. To name a few examples, they are used in medicine to identify brain structures in magnetic resonance images [24], in geology to analyze 3D seizmic data volumes [33], in energetics for forecasting regional electricity loads [19], or in logistics for driving trucks [2]. Autonomous navigation cars are currently being researched [21] and tested heavily, but the most apparent obstacles today are legal barriers. Although in some states, autonomous cars are

already legal today (Nevada, Assembly Bill 511, 2011). Their use is about to revolutionize the transportation of people and goods.

1.1 Motivation

Artificial neural networks are proving themselves to be a very effective and extremely universal tool. All the above examples of usefulness and effectiveness of artificial neural networks are showing a very promising future for their use. In the past decade they were almost exclusively a domain of revolutionary companies with virtually unlimited resources funding their research, like Google, Facebook, Netflix, etc. However today, the methods of artificial intelligence seem to shape as an indivisible part of any successful venture in the industry. Some experts even predict that artificial intelligence will lead to the next industrial revolution.

The main objectives of this thesis are to discuss the theory behind deep artificial neural networks, design and build a high-performance implementation using parallel architecture, and evaluate the obtained performance results. The network architectures studied and implemented will comprise the multilayer perceptron network, the convolutional neural network, and the deep belief network. This work is not focused on solving any specific problem, but is rather universal. The resulting implementation, however, should be very flexible in terms of the ability to create any desired network architecture, and in terms of configuring the network parameters. This will allow the users to model neural networks capable of solving any specific task at hand.

1.2 Methodology

The outcome of this work will be an implemented framework for modeling artificial neural networks. The supported network types will be the previously mentioned multilayer perceptron network, the convolutional network, and the deep belief network. The framework will provide a simple but user friendly approach to configuring the specific network architecture, and the network and training parameters. All the configuration options will be accessible without the need of recompiling the framework itself. This is an important property, as it will not require the users to understand and edit the code.

The evaluation of the implemented framework will also form an important part of this work. It will be done via automated test cases with different network architectures and parameters. These test cases will be defined as code, so that anyone can rerun the tests in the future and verify the achieved results. An important part of the evaluation will also be a comparison with other available frameworks capable of working with the above mentioned neural network types. These test cases will also be automated and defined as code for all of the frameworks. This will again guarantee repeatable and verifiable results.

All the test cases along with the results will be included in the optical disk attached to the physical copies of the thesis.

1.3 Thesis structure

This thesis consists of three chapters. The first chapter provides the theoretical background for the studied artificial neural networks. The second chapter documents the implemented framework for working with the supported neural network types. The last chapter presents the evaluation of the implemented framework.

The first chapter with the theory behind artificial neural networks explains the structure of a neuron as the fundamental building block of the network. It also discusses the universal concepts reused in all the implemented network types. In the following sections, this chapter defines each of the studied network types. It also describes the training process for each network type separately.

The second chapter compares the available parallel hardware architectures and discusses whether they are suitable for implementing artificial neural networks. The decision regarding the chosen hardware platform is made, along with explaining the supporting arguments. The chapter continues with a detailed description of the software architecture and the decisions made during framework implementation. The installation procedure and the configuration options can be also found here. The chapter concludes with the specifics in the implementation of each network type.

The last chapter begins with testing requirements before explaining the testing methodology. It continues with the presentation of the results obtained from testing. The last two sections in this chapter compares the implemented framework to other existing frameworks for working with deep neural networks. The results are presented in graphs comparing the time performance of each tested framework with the same network configuration and on the same hardware.

2. Artificial neural networks

Inspired by biological nervous systems, artificial neural networks (ANNs) aim at reaching their versatility through learning. ANNs are commonly employed in artificial intelligence, machine learning and pattern recognition. There has been substantial research into how the human brain's structure achieves such a high level of versatility. This research has provided some important insights, however the conclusions are far from completely explaining the complex functioning of the brain. Even though we have not been able to replicate the brain so far, the field of artificial intelligence offers very effective solutions to many problems by simulating the observations of biological research of various nervous systems.

It is estimated, that the average human brain contains 86 billion neurons [13]. Together they form a huge network. Even if we knew the detailed inner structure of the human brain, we would still not be able to simulate it with current technology because of its robustness. Our efforts are therefore rather different. We want to build a neural network with a good ratio between its size and its effectiveness.

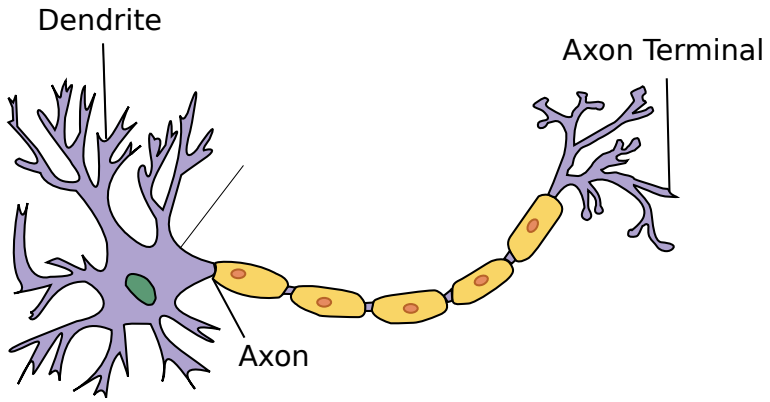


Figure 2.1: Model of a biological neuron. Image was taken from Wikipedia, where it is published under Creative Commons BY-SA 3.0 license. Image was adapted.

Generally, ANNs consist of a set of artificial neurons. Formally, an artificial neuron has n inputs represented as a vector $\vec{x} \in \mathbb{R}^n$. Inputs in an artificial neuron correspond to the dendrites in a biological neuron, while a single output of an artificial neuron corresponds to the axon in a biological neuron, which is depicted in Figure 2.1. Each input i, $1 \leq i \leq n$, has an assigned weight $w_1, \ldots, w_n$. Weighted input values are combined and run through an activation function producing some output y, as shown in Figure 2.2. The network is formed by connecting the neuron output with the input of a different neuron. ANN is therefore effectively described as an oriented graph as shown in Figure 2.4, where vertices represent the neurons, and oriented edges represent the output-input connections between them.

A set of input neurons consists of the neurons which are the first ones in any complete path in the graph. All input neurons have exactly one input, and all inputs together represent an instance of the problem to be solved by the ANN. A set of output neurons consists of the neurons which are the last ones in any complete path in the graph. All output neurons have exactly one output, and

all outputs together represent a possible solution to the problem to be solved by the ANN. A set of hidden neurons consists of the neurons which are not input, nor output neurons. Their number and organization into layers may vary even for the same problem, but is a key feature of the network vastly influencing its performance.

An ANN works by feeding the data into the input neurons. The data flows in the direction of oriented edges and ends when the output neurons are hit. The result is interpreted from the values obtained in the output neurons.

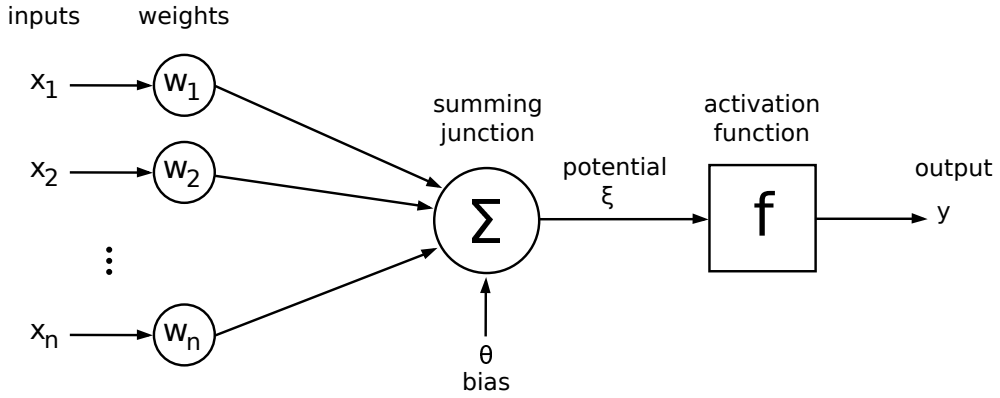


Figure 2.2: Model of an artificial neuron.

Formally, an ANN is a 6-tuple $M = (N, C, I, Y, w, t)$, where

- N is a finite non-empty set of neurons,
- $C \subseteq N \times N$ is a non-empty set of oriented edges between the neurons,
- $X \subset N$ is a non-empty set of neurons in the input layer,
- $Y \subset N$ is a non-empty set of neurons in the output layer,
- $w : C \mapsto \mathbb{R}$ is a weighting function,
- $t : N \mapsto \mathbb{R}$ is a function for network bias.

Let us consider neuron j with its input $\vec{x_j} = (x_{1j}, \ldots, x_{nj})$, weights $w_{1j}, \ldots, w_{nj}$ and bias θ_j. Then the potential of the neuron is computed:

$$\xi_j = \sum_{i=1}^{n} w_{ij} x_{ij} + \theta_j. \tag{2.1}$$

Consider the following activation function:

$$f(\xi_j) = \frac{1}{1 + e^{-\xi_j}}. \tag{2.2}$$

Then the output y_i of the neuron j is computed:

$$y_j = f(\xi_j) = f\left(\sum_{i=1}^{n} w_{ij} x_{ij} + \theta_j\right). \tag{2.3}$$

Considering ANN containing m such neurons in the output layer, we obtain the output of the network as $\vec{y} = (y_1, \ldots, y_m)$.

A more general definition is given by Atencia, Joya and Sandoval [1], where ANN is defined as a dynamic system whose state, at a given instant, is characterized by M. Network states are referred to as configurations. During the learning process, ANNs may change their weights, bias, or in some networks even the number of neurons and their setup. In contrast, our definition does not allow such modifications and M does not change in the process of learning.

2.1 Perceptron

The most trivial neural network is the perceptron that consists of one fully functional neuron only. Its output is calculated directly by Equation 2.3. In Figure 2.3 we can see the most commonly used activation functions: sigmoid (Eq. 2.2) and unit step function. The output of the sigmoid function produces continuous output, while the unit step function produces strictly binary outputs. By the standard definition, the single perceptron forming a trivial network uses step function exclusively.

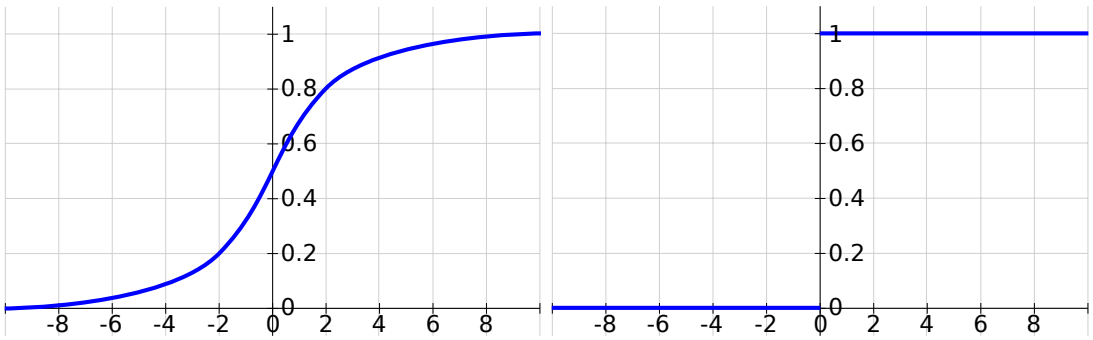


Figure 2.3: Graph of the sigmoid and the unit step function.

ANNs consisting of a single perceptron neuron are too trivial for complex tasks. However, they can be combined into a multilayer perceptron network (MLP) to address a wider set of problems. As we will see in Section 2.2.1 discussing training of an MLP, continuous functions are more suitable for gradient-based learning, because we can calculate their derivative. This is important to easily determine the weight adjustment rules.

2.2 Multilayer perceptron

The Multilayer perceptron (MLP) is a feed-forward neural network consisting of multiple mutually interconnected layers of neurons. The layers are stacked one onto each other. Every neuron in one layer is connected to every neuron in the following layer. The motivation behind designing multilayer networks is to be able to solve more complex tasks. The MLP network is built of perceptrons defined in Section 2.1. The unit step function is usually not a suitable activation function to be used with the perceptrons. Because of continuity and greater flexibility, the sigmoidal function is most commonly used instead. When choosing the most suitable activation function, we want it to be differentiable at every point of its

domain, and to be non-linear. Non-linearity is important, because in general we want the output to be non-linearly dependent on the given input.

Perceptrons are arranged into $\kappa \geq 2$ layers. Let us consider a network M with κ layers. The set of neurons C is split into mutually disjunct subsets called layers $L_1, \ldots, L_\kappa$. More formally it holds $\forall\ i, j : 1 \leq i, j \leq \kappa\ (L_i \neq \emptyset \wedge L_i \cap L_j \neq \emptyset) \Rightarrow i = j$. The network layers are stacked one onto each other, L_1 being the input layer, $L_2, \ldots L_{\kappa-1}$ being the hidden layers and L_κ being the output layer. As shown in Figure 2.4, the edges are all oriented in the direction from the input layer L_1 towards the output layer L_κ. Each neuron in layer L_i is connected to every neuron in layer L_{i+1}. In other words all neighboring layers form complete bipartite graphs.

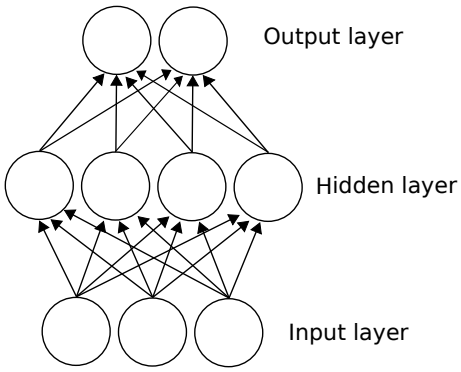


Figure 2.4: A Multilayer perceptron network consisting of 3 layers.

The output of the network is computed sequentially, layer by layer. We start with the input layer by directly assigning $\vec{y}^0 = \vec{x}$. Then the computation proceeds by assigning input $\vec{x}^i = \vec{y}^{i-1}$ for layer L_i. The weights and the activation function are given by the network, thus the output of each layer depends only on the output of the previous layer. The final output of the network is then produced as $\vec{y}^k$ in the output layer L_κ.

2.2.1 The training process

The ability to learn is the key concept of neural networks. The aim of the process is to find the optimal parameters (and structure) of the network for solving the given task. Before the training process starts, network parameters need to be initialized. Initial values are often chosen randomly, however using some heuristics may lead to a faster parameter adjustment towards the optimal values. Learning is then carried out on the training set by feeding the training data through the network. It is an iterative process, where the outputs produced on each input from the training set are analyzed and the network is repeatedly being adjusted to produce better results. The network is considered to be trained after reaching the target performance on the training data. There exist different metrics for assessing the network performance. I will use the mean squared error, which I will define formally in the following text (Eq. 2.7).

In this thesis I will focus on learning from a labelled dataset. Such dataset consists of input patterns for the network with their corresponding labels - expected network outputs. The process of learning from the labelled dataset is

reffered to as supervised learning. If the data labels are not available, it is possible to employ unsupervised learning using specialized algorithms, which will not be discussed here.

A common problem encountered in the process of learning is overfitting. It generally occurs when the learning is performed for too long, and especially when the training set is too small to evenly represent all types of patterns from the domain of possible network inputs. In such a case the learning may adjust the network to random features present in the training data. Overfitting is observed during the learning process, when network's predictive performance is improving on the training set, however worsening on previously unseen test data.

To combat this issue the labeled data is split into a training and a validation set. The main reason why to use the validation set is that it shows the error rates on the data independent from the data we are training on. A study by Guyon suggests that the optimal ratio between the size of training vs. validation data set depends on the number of recognized classes and the complexity of class features [11]. An estimation of feature complexity is however quite cumbersome. A good starting point for determining this ratio is to put 80% of the available data into the training set and 20% into the validation set. Further experimentation may help to move closer to the optimal ratio. While learning, the performance of the ANN is regularly examined on validation data set. When errors retrieved on validation data reach a stopping point, learning process is stopped (see Figure 2.5) and the network is considered trained.

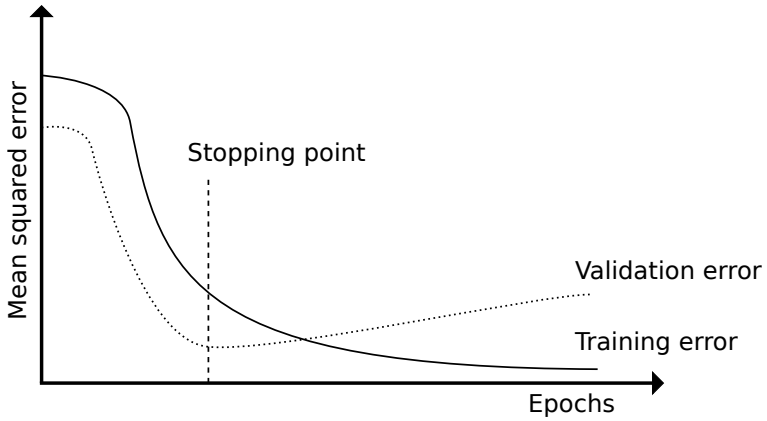


Figure 2.5: Graph comparing the evolution of the training error vs. the validation error.

To define the learning process formally and in more detail, let us consider P training patterns labeled $(\vec{x}^p, \vec{d}^p)$, where $\vec{x}^p$ is the input vector, $\vec{d}^p$ is the desired output vector, and $1 \leq p \leq P$. Given the current configuration of the network, the input $\vec{x}^p$ yields the output $\vec{y}^p$. Then for every pattern p we want $\vec{y}^p$ to be as close to the desired output $\vec{d}^p$ as possible. We can define the error of each neuron j in the output layer as:

$$e_j^p = y_j^p - d_j^p. \tag{2.4}$$

Now we can define the squared error for pattern p as:

$$E_p = \frac{1}{m_\kappa} \sum_{j=1}^{m_\kappa} \left(e_j^p\right)^2 = \frac{1}{m_\kappa} \sum_{j=1}^{m_\kappa} \left(y_j^p - d_j^p\right)^2, \tag{2.5}$$

where m_κ is the number of neurons in the output layer. Note that if the actual output is exactly the same as the desired output, we get zero for the squared error. In other words the following holds true:

$$\forall j : E_p = 0 \Leftrightarrow e_j^p = 0 \Leftrightarrow y_j^p = d_j^p. \tag{2.6}$$

It may be useful to sum up the average error for all input patterns to assess the network performance on the whole dataset, which can be achieved simply by computing the mean squared error [12, pp. 183]:

$$E_{avg} = \frac{1}{P} \sum_{p=1}^{P} E_p. \tag{2.7}$$

When learning, for each interconnected pair of neurons (i, j), where i is a neuron in layer l, j is a neuron in layer $l+1$ and $w_{i,j}$ weights their connection, we want to adjust $w_{i,j}$ to minimize the mean squared error E_{avg}. Provided the activation function is differentiable everywhere on its domain, E_{avg} is also differentiable. When adjusting the weight $w_{i,j}$ of the neuron j located in the output layer κ, we are therefore interested in the partial derivative:

$$\frac{\partial E_{avg}}{\partial w_{i,j}} = \frac{1}{P} \frac{\partial}{\partial w_{i,j}} \sum_{p=1}^{P} E_p = \frac{1}{P} \sum_{p=1}^{P} \frac{\partial E_p}{\partial w_{i,j}}. \tag{2.8}$$

To be able to adjust the network after each presented input pattern, we are actually interested in the derivative for each given pattern p and its corresponding E_p. In the following equations we will therefore omit the pattern index p. Applying the chain rule to Equation 2.8 we get:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w_{i,j}}. \tag{2.9}$$

Using Equation 2.5 we can evaluate the first factor as:

$$\frac{\partial E}{\partial y_j} = \left(y_j - d_j\right). \tag{2.10}$$

Then we evaluate the second factor:

$$\frac{\partial y_j}{\partial w_{i,j}} = \frac{\partial y_j}{\partial \xi_j} \frac{\partial \xi_j}{\partial w_{i,j}} = f'\left(\xi_j\right) \frac{\partial}{\partial w_{i,j}} \sum_k w_{k,j} y_k = f'\left(\xi_j\right) y_i. \tag{2.11}$$

By combining both evaluated factors we get:

$$\frac{\partial E}{\partial w_{i,j}} = \left(y_j - d_j\right) f'\left(\xi_j\right) y_i. \tag{2.12}$$

For convenience Haykin defines the so-called local gradient δ_j for neuron j in the output layer [12, pp. 185] as the following relation:

$$\delta_j = \frac{\partial E}{\partial \xi_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial \xi_j} = \left(y_j - d_j\right) f'\left(\xi_j\right). \tag{2.13}$$

Then we can reformulate Equation 2.12 into:

$$\frac{\partial E}{\partial w_{i,j}} = \delta_j y_i. \tag{2.14}$$

Using Equation 2.14 we can compute the gradient of the error function for each of the given patterns p. We need to adjust the weight $w_{i,j}$ proportionally to the gradient but in the opposite direction. However doing so for every input pattern would produce a very unstable system. To combat this problem we can use a learning parameter $0 < \eta < 1$. The weight adjustment is then computed by:

$$\Delta w_{i,j} = -\eta \frac{\partial E}{\partial w_{i,j}} = -\eta \delta_j y_i. \tag{2.15}$$

The weight adjustment $\Delta w_{i,j}$ in 2.15 is only applicable to the neurons in the output layer. Computation of the adjustment for neurons in the hidden layers is more complicated. For instance, consider 3 neurons $i, j,$ and k, all following each other on the same path along the layers $l-1, l$ and $l+1$, respectively, as illustrated in Figure 2.6. Then the adjustment of $w_{i,j}$ needs to be done carefully, because besides influencing the output of neuron i itself, it also impacts all the outputs (and thus errors) in all layers following l. Minding this, let us bring our attention to the the layers $l < L$ in the following text.

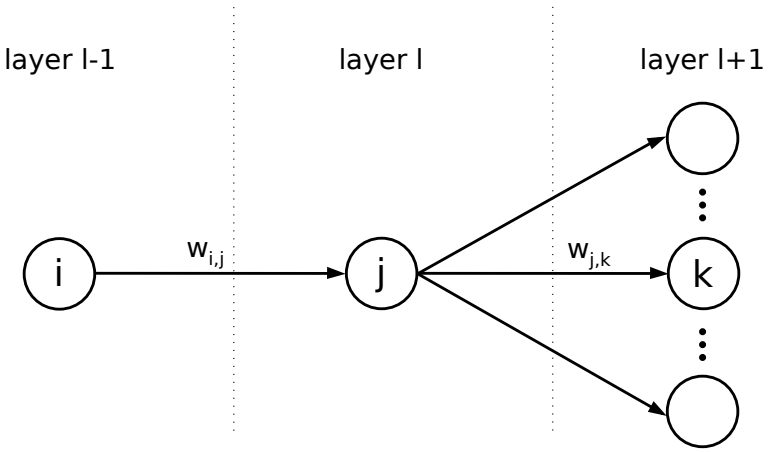


Figure 2.6: Illustration showing how a change in the weight $w_{i,j}$ of the neuron in the hidden layer $l-1$ influences the weight $w_{j,k}$ of the neuron in the following layer l.

Note that Equation 2.12 still applies to hidden layers. However, we need to look at the definition of the local gradient again. In the previous Equation 2.13 we are using the desired output d_j to calculate $\partial E / \partial y_j$. Of course, there is no desired output known in hidden layers. It is actually dependent on the network design. Because of this, we need to step back and use the following definition for δ_j:

$$\delta_j = \frac{\partial E}{\partial \xi_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial \xi_j} = \frac{\partial E}{\partial y_j} f'(\xi_j). \tag{2.16}$$

Now we need to redefine $\partial E/\partial y_j$ for hidden layers. For any hidden layer l, the following layer $l+1$ must exist (otherwise l would be the output layer). Given the neurons i, j and k each in a different layer as illustrated in Figure 2.6, we can use the potential ξ_k.

$$\frac{\partial E}{\partial y_j} = \sum_{k=1}^{m_{l+1}} \frac{\partial E}{\partial \xi_k} \frac{\partial \xi_k}{\partial y_j} = \sum_{k=1}^{m_{l+1}} \frac{\partial E}{\partial \xi_k} w_{j,k} = \sum_{k=1}^{m_{l+1}} \delta_k w_{j,k} \tag{2.17}$$

Combining equations 2.16 and 2.17 we get:

$$\delta_j = \left(\sum_{k=1}^{m_{l+1}} \delta_k w_{j,k} \right) f'(\xi_j). \tag{2.18}$$

Equation 2.18 tells us, that knowing the local gradient of neuron k in layer $l+1$, we can calculate the local gradient for neuron j in layer l. This fact will allow us to recursively adjust the network weights going layer by layer in the direction from the output to the input (backwards). It is used in the most common learning algorithm discussed in Section 2.2.2.

Finally, we can summarize the weight adjustment applicable to all the layers in a given network:

$$\Delta w_{i,j} = -\eta \frac{\partial E}{\partial w_{i,j}} = -\eta \delta_j y_i, \tag{2.19}$$

where

$$\begin{aligned} &\forall j \text{ in layer } l < L \qquad && \delta_j = \left(\sum_{k=1}^{m_{l+1}} \delta_k w_{j,k} \right) f'(\xi_j) \\ &\forall j \text{ in layer } L \qquad && \delta_j = (y_j - d_j) f'(\xi_j) \end{aligned} \tag{2.20}$$

2.2.2 Backpropagation algorithm

The purpose of the backpropagation algorithm is to adjust the synaptic weights of neurons, so that the network produces the desired output. The algorithm describes the process of training (also called learning). The result of this algorithm is a neural network configured to minimize the error when solving given problem.

Training must be performed on labeled data and therefore is supervised. Before the algorithm starts, the weights need to be initialized to some values. The initialization is not a part of the algorithm specification, as there may be different approaches, the most common being the most trivial - random initialization. Then the training algorithm starts.

Each input pattern with its class label $(\vec{x}_p, \vec{d}_p)$ is sequentially processed in two phases. The first phase called the forward phase puts the input pattern as the input of the network, setting $\vec{y}^0 = \vec{x}_p$. The network then computes its output $\vec{y}^L$. The sole purpose of the forward phase is to calculate the output for the presented pattern, and the network is not adjusted at all. At this point the backward phase starts. The purpose of this phase is to adjust the network weights to achieve a better assessment of the input data.

Learning is performed in multiple epochs. In each epoch, all the data from the training set is processed. The duration of an epoch directly depends on the size of the network and the size of the training set, as each input pattern

from the training set is processed exactly once. However the number of epochs is not limited. An important decision of the learning process is therefore the determination of the stopping criterion. After each epoch we validate the error on the validation data set. Once this error starts to increase, we achieved some (local or perhaps global) minimum, and it is usually wise to stop the learning process at this point (see Figure 2.5).

Formally, the backpropagation algorithm will be described below:

Input:

- training patterns $(\vec{x}^p, \vec{d}^p), 1 \leq p \leq P$
- validation patterns $(\vec{v}^q, \vec{d}^q), 1 \leq q \leq Q$
- activation function f and its derivation $f'(\xi)$
  - e.g. the sigmoidal function: $f(\xi) = 1/(1+e^{-\xi})$; $f'(\xi) = f(\xi)(1-f(\xi))$
- learning parameter $\eta \in (0, 1)$
- feed-forward neural network M with randomly initialized weights

Output:

- a trained feed-forward neural network M'

Algorithm:

1. Set $E_{avg} = \infty$.
2. Start a new epoch.
3. For each $p \in \{1, \ldots, P\}$ present the pattern $(\vec{x}^p, \vec{d}^p)$.
   - Set $\vec{y}^0 = \vec{x}^p$.
   - forward phase
     - For $l = 1, 2, \ldots, \kappa$ compute the output of M for $\vec{x}^p$ in the following way:
       $\forall j \in L_l$ compute $\xi_j = \sum_{i=1}^{n} w_i x_i + \theta$ and $y_j = f(\xi)$.
   - backward phase
     - $\forall i \in L_{\kappa-1}, j \in L_\kappa$ compute $\delta_j = (y_j - d_j^p) f'(\xi_j)$ and $\Delta w_{i,j} = -\eta \delta_j y_i$.
     - For $l = \kappa - 1, \ldots, 1$ do
       $\forall i \in L_{l-1}, j \in L_l$ compute $\delta_j = \sum_{k=1}^{m_{l+1}} \delta_k w_{j,k} f'(\xi_j)$, $\Delta w_{i,j} = -\eta \delta_j y_i$.
     - For $l = 1, 2, \ldots, \kappa$ $\forall (i, j) \in L_{l-1} \times L_l$ adjust $w_{i,j}$ by $\Delta w_{i,j}$.
     - Compute the error for the pattern p: $E_p = 1/m_\kappa \sum_{j=1}^{m_\kappa} (y_j^p - d_j^p)^2$.
4. Set $E_{prev} = E_{avg}$ and compute new $E_{avg} = 1/P \sum_{p=1}^{P} E_p$ for the validation data set.
5. If $E_{avg} < E_{prev}$ go to step 2.
6. Finish.

2.3 Convolutional neural networks

Convolutional neural networks (CNNs) are multi-layer feed-forward networks specifically designed to recognize features in 2-dimensional image data. The architecture of CNNs is inspired by Hubel and Wiesel's study of neurobiological signal processing in cat's visual cortex [20]. A typical application of CNNs consists of recognition of various objects in images. However convolutional networks have been successfully used for various different tasks [7] [6], too.

CNNs are primarly used for 2D image recognition, so we will illustrate their architecture on a 2D rectangular image consisting of pixels. Each pixel generally carries colour information. Colour can be represented by multiple channels (e.g. 3 RGB channels). For the sake of simplicity, we will consider only one single channel (shades of gray) while explaining the model.

The neurons in CNNs work by considering a small portion of the image, let us call it subimage. The subimages are then inspected for features that can be recognized by the network. As a simple example, a feature may be a vertical line, an arch, or a circle. These features are then captured by the respective feature maps of the network. A combination of features is then used to classify the image. Furthermore, multiple different feature maps are used to make the network robust to varying levels of contrast, brightness, colour saturation levels, noise, etc.

There are two types of layers, both consisting of feature maps. The purpose of the so-called convolutional layer is to recognize the features in the input image. It usually consists of several feature maps, each map recognizing certain feature. The subimages intentionally cover overlapping regions of the original image. Such a design is important in order to tolerate image distortions, like translation, rotation, skewing, etc.

The other layer type, the so-called subsampling layer, always follows after a convolutional layer. It consists of the same number of feature maps, where each map from the convolutional layer is used as an input for the corresponding feature map in the following subsampling layer. Subimages covered by the neurons from the subsampling layer usually do not overlap.

Depending on the network's depth, the convolutional and subsampling layers alternate until the last subsampling layer is reached. After the last subsampling layer, there may be any number of fully connected layers as described in Section 2.2, the last one of them is the output layer [22].

2.3.1 Architecture of CNNs

To explain the architecture of the convolutional network, let us consider the image data in the input layer. From a high-level point of view, a convolutional network is architecturally split into two important parts. Each part is designed to fulfill a different purpose. The first part of the network performs feature extraction, and is built of convolutional and subsampling layers. The second part performs classification based on the extracted features. We will be using fully-connected MLP for this part.

Let us describe the feature extraction process in more detail. Starting with the presented input image, each pixel represents the input for the neurons grouped in

feature maps of the first convolutional layer. The neurons in the feature map are organized in a 2-dimensional grid. All the neurons within the same feature map share their weights. This allows to optimize the implementation by requiring less memory and yielding a better performance.

More importantly, it is an architectural feature. Each neuron in a given feature map is expected be recognize the same feature. The feature is recognized by a combination of weights, which are essentially filtering neural inputs. Sharing the same weights for all the neurons within the given feature map ensures the same filter is used for each pixel.

Let F_l be the set of feature maps in the layer l. All feature maps in F_l have the same size, let us denote it $m_l \times n_l$. Further, $\forall\, 0 \leq i < m_l, 0 \leq j < n_l$ let $y_{i,j}^{\phi,l}$ be the output of the neuron (i, j, ϕ, l), at the position (i, j) in the feature map ϕ of the layer l.

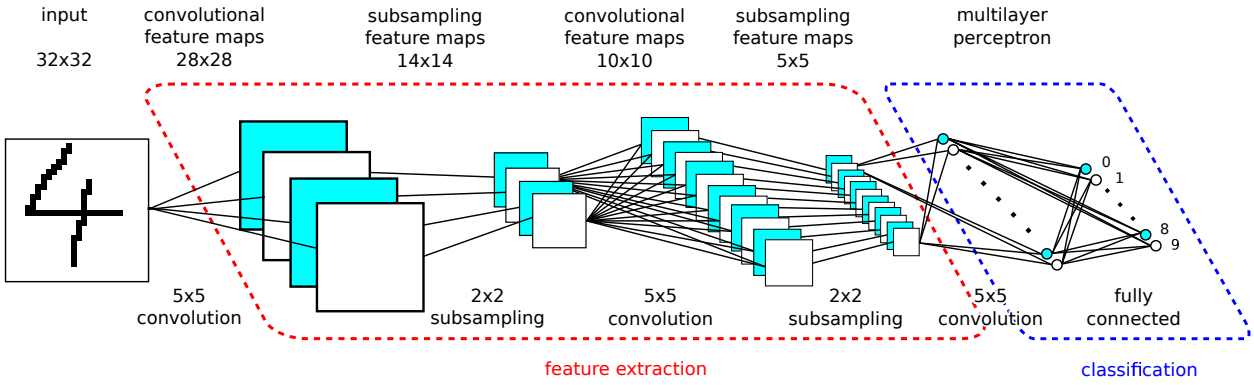


Figure 2.7: Example architecture of a convolutional neural network designed for the classification of handwritten digits from the MNIST dataset.

We will consider the input image as the output of the layer with zeroth index, having the size of $m_0 \times n_0$ pixels. Note that the input layer can be also considered as a subsampling layer with one single feature map. The input layer is then followed by alternating convolutional and subsampling layers. The neurons of the feature map ϕ in the convolutional layer l take the input from the set of feature maps $F'_{\phi,l}$, located in the previous subsampling layer ($\emptyset \neq F'_{\phi,l} \subset F_{l-1}$). The convolutional layer is then followed by a subsampling layer, which reduces the size of feature maps. The last subsampling layer ends the feature extraction process.

At this point the second part of the network begins the classification process. The classification is performed using a fully-connected MLP network, which classifies the inputs according to the extracted features. For an example of a CNN see Figure 2.7. In the following sections we will describe the convolutional and subsampling layers in more detail.

2.3.2 Convolutional layer

The purpose of convolutional layers is to detect the features in the presented images. It consists of multiple feature maps, each recognizing certain specific feature. The feature recognition can be thought of as running the subimage through a filter. The filtering is essentially done through weight adjustments.

A filter may highlight or suppress a pixel's potential based on the surrounding pixels. The filtering to use is determined automatically by the learning process via weight adjustments.

The number of feature maps in the convolutional layer is an important architectural decision to make when designing the convolutional network. The optimal value depends on the nature of the problem being solved. The best way to determine the number of feature maps is to experiment. As a rule of thumb, more complex images are recognized better using more features, while simple tasks produce better results using just a few features. In fact, if the chosen number of features is too big, the features learned will often be duplicated.

Let us consider a convolutional layer l with a set of feature maps F_l. In general, this layer is always preceded by a subsampling layer with a set of feature maps F_{l-1}. For the sake of simplicity, we will refer to a feature map in the convolutional layer as a convolutional map. A feature map in the subsampling layer will be referred to as a subsampling map.

All feature maps in the same layer have the same size. The size of convolutional maps is predetermined by the size of subsampling maps in the preceding layer, by the layer parameter r_c^l, and by the overlapping parameter s_c^l. Each neuron in a convolutional map takes its input from $(r_c^l)^2$ neurons in the corresponding subsampling map from the preceding layer. These input neurons form a square of dimension $r_c^l \times r_c^l$. The overlapping parameter $s_c^l \geq 1$ determines how many neurons apart are these squares from each other, as can be seen in Figure 2.8. In the following text we will refer to this square of inputs as the receptive field of the neuron.

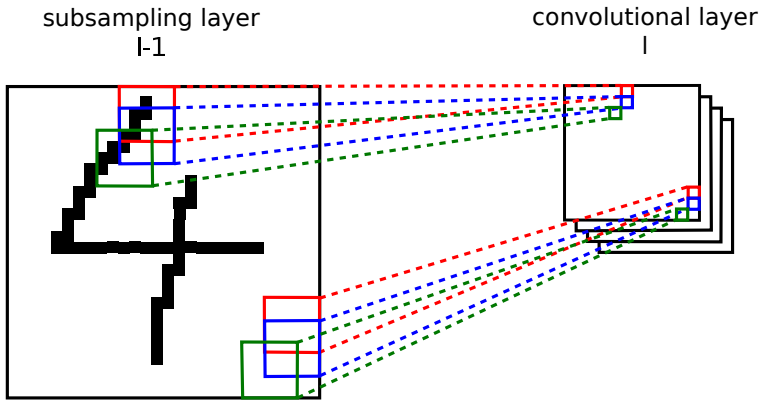


Figure 2.8: Illustration of the convolution operation on the input feature map of size 28×28 neurons in layer $l-1$. The convolution uses a receptive field of size 5×5 and overlapping parameter of 2 neurons, thus producing feature maps of size 12×12 neurons in layer l. Note that neighboring neurons in convolutional layers have overlapping receptive fields.

As illustrated in Figure 2.8, neighboring neurons in a convolutional map have neighboring receptive fields that overlap in exactly $(r_c^l - s_c^l)^2$ neurons. If the dimensions of the subsampling map are (m_{l-1}, n_{l-1}), the following convolutional maps must have the dimensions of:

$$(m_l, n_l) = \left(\left\lceil \frac{m_{l-1} - r_c^l + 1}{s_c^l} \right\rceil, \left\lceil \frac{n_{l-1} - r_c^l + 1}{s_c^l} \right\rceil \right). \tag{2.21}$$

Each neuron (i, j, ϕ, l) is thus connected to all the neurons $(is_c^l + \Delta i, js_c^l + \Delta j, \phi', l-1)$ from the preceding layer, where $\phi' \in F_{\phi,l}$ and $0 \leq \Delta i, \Delta j < r_c^l$. Each connection within the receptive field of neuron (i, j, ϕ, l) has the weight $w_{\Delta i,\Delta j}^{\phi,\phi',l}$. This weight is shared for all the neurons within the same feature map, i.e., for all i, j. At this point, we can compute the number of weights required by convolutional layer l. No matter what the dimensions of convolutional maps are, the total number of weights is:

$$|W_l| = |F_l| \cdot (r_c^l)^2. \tag{2.22}$$

The output $y_{i,j}^{\phi,l}$ and potential $\xi_{i,j}^{\phi,l}$ of the neuron (i, j, ϕ, l) are then computed as:

$$y_{i,j}^{\phi,l} = \xi_{i,j}^{\phi,l} = \sum_{\phi' \in F_{\phi,l-1}} \sum_{\Delta i=0}^{r_c^l-1} \sum_{\Delta j=0}^{r_c^l-1} y_{is_c^l+\Delta i, js_c^l+\Delta j}^{\phi',l-1} w_{\Delta i,\Delta j}^{\phi,\phi',l}. \tag{2.23}$$

The algorithm processing the convolutional layer l must sum the weights for each neuron. There are $|F_l| \cdot m_l \cdot n_l$ neurons in layer l. Each of them must sum up $|F_{l-1}| \cdot (r_c^l)^2$ weights (Eq. 2.22). This yields the following time complexity for processing the convolution in layer l.

$$O(|F_{l-1}| \cdot |F_l| \cdot (r_c^l)^2 \cdot m_l \cdot n_l) \tag{2.24}$$

Given the above definitions, there are 3 important parameters that need to be chosen to configure the convolutional layer.

- $|F_l|$: the number of convolutional maps that will be corresponding to a single preceding subsampling map,
- r_c^l : the receptive field size parameter,
- s_c^l : the overlapping parameter.

As we already discussed, an adequate number of convolutional maps depends on the complexity of features in the classified input. We will test different values for a specific problem in Chapter 4.

The optimal value of the receptive field parameter also depends on the given problem. Its values are often small, so that the network can detect simple and small local features. The correct combination of the local features can then be trained. For instance, LeNet-5 network [22] uses $r_c^l = 5$ for an input with $m_0 = n_0 = 32$.

The value of overlapping parameter is usually equal to 1. Increasing the parameter rarely makes sense, because if the receptive fields are too far apart, the network can miss important features. It might make sense to use higher overlapping parameter as an optimization technique for big images and big receptive fields. It is however disputable whether resampling the input images to a lower resolution would be a better approach. In our experiments in Chapter 4 we will be using relatively small input images, and therefore set this parameter to 1.

2.3.3 Subsampling layer

The subsampling layer is either the first layer in the network (the input layer), or it follows after the convolutional layer. Let us consider a subsampling layer l consisting of a set of subsampling maps F_l. The purpose of the subsampling layer is to reduce the sizes of feature maps to simplify and generalize the recognized features.

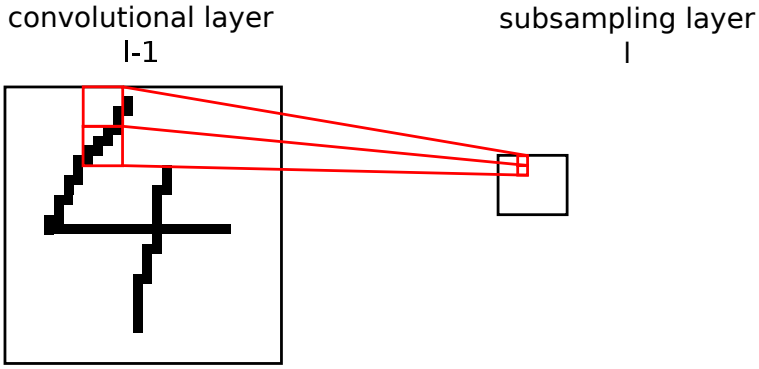


Figure 2.9: Illustration of the subsampling operation on the input feature map of the size 28×28 neurons from the layer $l-1$. In this case, the subsampling uses receptive fields of size 4×4 neurons, thus producing feature maps of the size 7×7 neurons in the layer l. Note, that in this illustrated case, the receptive fields of the subsampling neurons do not overlap.

Each convolutional map in layer $l-1$ is connected only to the corresponding subsampling map in layer l. The receptive fields of the subsampling neurons are specified by the parameters r_x^l and r_y^l, and usually do not overlap. Such a setting reduces the size of the feature maps by the factors r_x^l and r_y^l in each dimension, as illustrated in Figure 2.9. Formally, the new feature map sizes will be:

$$m_l = \left\lceil \frac{m_{l-1}}{r_x^l} \right\rceil \text{ and } n_l = \left\lceil \frac{n_{l-1}}{r_y^l} \right\rceil. \tag{2.25}$$

By this definition, each neuron from the convolutional layer is connected to exactly one neuron from the subsampling layer (and each convolutional map is connected to exactly one subsampling map). This results into a simple network design with the following time complexity when computing the output of the subsampling layer in the forward pass.

$$O(|F_{l-1}| \cdot m_{l-1} \cdot n_{l-1}) \tag{2.26}$$

Generally, the subsampling parameters can be chosen from the range $1 \ldots m_{l-1}$ for r_x^l and $1 \ldots n_{l-1}$ for r_y^l. The usual values are $r_x^l = r_y^l = 2$, which is the lowest value that makes sense. If the receptive fields should be out of the limits of the feature map bounds, their inputs will be simply considered to have the value of zero. It is effectively the same as if we padded the feature map with neurons with zero potential.

The neurons from the subsampling maps take their input from their receptive fields. Multiple inputs ($r_x^l \times r_y^l$) are then combined into a single value denoted as the neuron potential. The most common approaches to combine these inputs is

through averaging (Eq. 2.27), or finding the maximum value (Eq. 2.28). Having the feature map ϕ in layer l, its output $y_{i,j}^{\phi,l}$ is then multiplied by a trainable coefficient $a^{\phi,l}$, added to trainable bias $b^{\phi,l}$, and passed to an activation function f (Eq. 2.29).

$$\xi_{i,j}^{\phi,l} = \frac{1}{r_x^l r_y^l} \sum_{\Delta i=0}^{r_x^l-1} \sum_{\Delta j=0}^{r_y^l-1} y_{ir_x^l+\Delta i, jr_y^l+\Delta j}^{\phi,l-1} \tag{2.27}$$

$$\xi_{i,j}^{\phi,l} = \max_{\substack{\Delta i \in (0, r_x^l-1) \\ \Delta j \in (0, r_y^l-1)}} (y_{ir_x^l+dx, jr_y^l+dy}^{\phi,l-1}) \tag{2.28}$$

$$y_{i,j}^{\phi,l} = f(a^{\phi,l}\xi_{i,j}^{\phi,l} + b^{\phi,l}) \tag{2.29}$$

2.3.4 Backpropagation in Convolutional Networks

Convolutional networks are a kind of feed-forward networks. Hence it is possible to train them using the backpropagation algorithm. We will use the ideas and some definitions introduced in Section 2.2.1. The general objective will be the same - minimize the network error rate E through gradient descend technique. The weight adjustment will again start from the last layer. Using supervised learning, the expected neuron output is known here. We can use Equation 2.5 to compute the error E_p for the pattern p. We will be considering convolutional networks that have a conventional Multilayer Perceptron (MLP) as their last layer. The backpropagation process of MLP is defined in Section 2.2.2, and will be the same for this part of the convolutional network.

The details how to cope with different types of the network's lower layers will be discussed in the following subsections.

2.3.4.1 Backpropagation in subsampling layers

The subsampling layer does not have any trainable weight parameters. In fact, its only trainable parameters are the coefficient $a^{\phi,l}$ and bias $b^{\phi,l}$. As we are interested in the influence of these parameters on the errors, we need to consider $\partial E/\partial a^{\phi,l}$ and $\partial E/\partial b^{\phi,l}$ in the subsampling layer l. Using Equation 2.29 and applying the chain rule yields the following:

$$\frac{\partial E}{\partial a^{\phi,l}} = \sum_{i,j} \frac{\partial E}{\partial y_{i,j}^{\phi,l}} \cdot \frac{\partial y_{i,j}^{\phi,l}}{\partial a^{\phi,l}} = \sum_{i,j} \frac{\partial E}{\partial y_{i,j}^{\phi,l}} \cdot f'(a^{\phi,l} \cdot \xi_{i,j}^{\phi,l} + b^{\phi,l}) \cdot \xi_{i,j}^{\phi,l} \tag{2.30}$$

$$\frac{\partial E}{\partial b^{\phi,l}} = \sum_{i,j} \frac{\partial E}{\partial y_{i,j}^{\phi,l}} \cdot \frac{\partial y_{i,j}^{\phi,l}}{\partial b^{\phi,l}} = \sum_{i,j} \frac{\partial E}{\partial y_{i,j}^{\phi,l}} \cdot f'(a^{\phi,l} \cdot \xi_{i,j}^{\phi,l} + b^{\phi,l}) \tag{2.31}$$

The partial derivative $\partial E/\partial y_{i',j'}^{\phi',l-1}$ in the above equations is still unknown. It expresses how the output of the layer $l-1$ influences the error. Let ϕ denote the respective feature map in the layer l, and ϕ' the corresponding feature map producing the input for ϕ. Note that ϕ' can only represent one single feature map, because ϕ is a subsampling map. The chain rule can be leveraged here again:

$$\frac{\partial E}{\partial y_{i',j'}^{\phi',l-1}} = \sum_{i,j} \frac{\partial E}{\partial y_{i,j}^{\phi,l}} \cdot \frac{\partial y_{i,j}^{\phi,l}}{\partial \xi_{i,j}^{\phi,l}} \cdot \frac{\partial \xi_{i,j}^{\phi,l}}{\partial y_{i',j'}^{\phi',l-1}} \tag{2.32}$$

The derivative $\partial E / \partial y_{i,j}^{\phi,l}$ is known by assumption. The expression $\partial y_{i,j}^{\phi,l} / \partial \xi_{i,j}^{\phi,l}$ can be derived from Equation 2.29:

$$\frac{\partial y_{i,j}^{\phi,l}}{\partial \xi_{i,j}^{\phi,l}} = f'(a^{\phi,l} \cdot \xi_{i,j}^{\phi,l} + b^{\phi,l}) \cdot a^{\phi,l} \tag{2.33}$$

The evaluation of the last term from Equation 2.32 depends on the subsampling type. As mentioned before, we will be considering the average or maximum value.

Averaging yields:

$$\frac{\partial \xi_{i,j}^{\phi,l}}{\partial y_{i',j'}^{\phi',l-1}} = \frac{1}{r_x^l \cdot r_y^l} \tag{2.34}$$

Maximization yields:

$$\frac{\partial \xi_{i,j}^{\phi,l}}{\partial y_{i',j'}^{\phi',l-1}} = \begin{cases} 1 & \text{if } y_{i',j'}^{\phi',l-1} \text{ is the maximum value} \\ 0 & \text{otherwise} \end{cases} \tag{2.35}$$

We are now able to compute the error gradients for both trainable parameters $a^{\phi,l}$ and $b^{\phi,l}$. However, the formal description of the Backpropagation algorithm from Section 2.2.2 needs to be adjusted, as it computes adjustment for weights $\Delta w_{i,j}$ in fully-connected layer. In case of a subsampling layer, it needs to compute the adjustment of the trainable parameters $\Delta a^{\phi,l}$ and $\Delta b^{\phi,l}$.

$$\Delta a^{\phi,l} = -\eta \frac{\partial E}{\partial a^{\phi,l}} \tag{2.36}$$

$$\Delta b^{\phi,l} = -\eta \frac{\partial E}{\partial b^{\phi,l}} \tag{2.37}$$

At this point we are able to propagate the error down towards the input layer using Equation 2.32, and adjust the trainable parameters using Equation 2.36 and Equation 2.37.

2.3.4.2 Backpropagation in convolutional layer

In the case of convolutional layer, the only trainable parameters are the weights shared for neurons in the same feature map. Note that there is no activation function used in convolutional layers.

Let us discuss how the weights affect the error by computing the derivative $\partial E / \partial w_{i,j}^{\phi,\phi',l}$ for the preceding convolutional layer l, feature map ϕ, and the respective feature map ϕ' in the layer $l-1$. This derivative can be computed using the chain rule and Equation 2.23. The expression $\partial E / \partial y_{p,q}^{\phi,l}$ is assumed to be known.

$$\frac{\partial E}{\partial w_{i,j}^{\phi,\phi',l}} = \sum_{p=0}^{m_l-1} \sum_{q=0}^{n_l-1} \frac{\partial E}{\partial y_{p,q}^{\phi,l}} \cdot \frac{\partial y_{p,q}^{\phi,l}}{\partial w_{i,j}^{\phi,\phi',l}} = \sum_{p=0}^{m_l-1} \sum_{q=0}^{n_l-1} \frac{\partial E}{\partial y_{p,q}^{\phi,l}} \cdot y_{p+i,q+j}^{\phi',l-1} \tag{2.38}$$

The last derivative we need to know to be able to backpropagate the errors is $\partial E/\partial y_{p,q}^{\phi',l-1}$. To compute it, we need to consider the set of neurons in layer l connected to the output $y_{p,q}^{\phi',l-1}$. Let us denote this set $\Theta_{p,q}^{\phi',l-1}$. Let us further denote the output of neuron $n \in \Theta_{p,q}^{\phi',l-1}$ with y_n, and denote the weight which connects neuron $(p, q, \phi', l-1)$ to neuron n with w_n. Then we can apply the chain rule to obtain:

$$\frac{\partial E}{\partial y_{p,q}^{\phi',l-1}} = \sum_{n \in \Theta_{p,q}^{\phi',l-1}} \frac{\partial E}{\partial y_n} \cdot \frac{\partial y_n}{\partial y_{p,q}^{\phi',l-1}} = \sum_{n \in \Theta_{p,q}^{\phi',l-1}} \frac{\partial E}{\partial y_n} \cdot w_n \tag{2.39}$$

Since we know all the outputs $y_{i,j}^{\phi,l}$ in layer l, and $y_n \in \cup_{\phi,i,j} y_{i,j}^{\phi,l}$, the derivative $\partial E/\partial y_n$ is also assumed to be known.

2.4 Deep belief networks

The first description modeling a Deep Belief Network (DBN) was published in 1986 by Smolensky [32]. At the time, the model was referred to as "harmonium". It is a type of deep neural network composed of multiple layers, each layer consisting of visible neurons representing the layer input, and hidden neurons representing the layer output. In this text we will consider a layer to own the hidden neurons. The visible neurons will be owned by the preceding layer, for which these neurons are hidden. The visible neurons are fully interconnected with the hidden ones. The distinctive feature of a DBN is that there are no connections between the visible neurons, and no connections between the hidden neurons. The connections are symmetric, and are exclusively between the visible neurons and the hidden ones.

In the following text we will start with a definition of a stochastic neuron, which will be used in the DBNs. Then we will describe the architecture of Restricted Boltzmann Machines, which represent the main building block of DBNs. The following sections will explain the network's architecture and the training process.

2.4.1 Stochastic model of a neuron

The model of a standard neuron depicted in Figure 2.2 is deterministic in a sense that its output is exactly defined for a given input. On the other hand the output of a stochastic neuron used in Boltzmann networks is probabilistic. The output y is binary, given by the probability $P(\xi)$.

$$y = \begin{cases} 1 & \text{with probability of } P(\xi) \\ 0 & \text{with probability of } 1 - P(\xi) \end{cases} \tag{2.40}$$

The probability function used here is again the sigmoid-shaped function:

$$P(\xi) = \frac{1}{1 + e^{-\frac{\xi}{T}}}, \tag{2.41}$$

where $T > 0$ is the so-called *pseudotemperature* parameter used to control the level of noise in the probability. The *pseudotemperature* T may be thought of as a parameter representing the effect of thermal fluctuations in the neural synapses

causing noise in the signal being transferred. Note that as T approaches 0, this stochastic model becomes deterministic:

$$\lim_{T\to 0^+} P(\xi) = \lim_{T\to 0^+} \frac{1}{1+e^{-\frac{\xi}{T}}} = \begin{cases} 1 & \text{for } \xi > 0 \\ \frac{1}{2} & \text{for } \xi = 0 \\ 0 & \text{for } \xi < 0 \end{cases} \tag{2.42}$$

2.4.2 Boltzmann machine

The Boltzmann machine is a stochastic neural network consisting of stochastic neurons first introduced by Hinton et al. [9]. It is referred to as a recurrent network, because all its connections are bidirectional, and there is a connection between every pair of neurons, even within the same layer. Therefore the graph of a Boltzmann network is always fully connected, as illustrated in Figure 2.10.

Let $w_{i,j}$ be the weight of the connection between neuron i and j, then the following invariants hold true:

$$\begin{aligned} \forall\, i \neq j : w_{i,i} &= 0 \\ w_{i,j} &= w_{j,i} \\ w_{i,j} &\geq 0 \end{aligned} \tag{2.43}$$

The weights are often represented as a matrix W with zeros on the diagonal. Each neuron is binary and therefore can be either in 'on' or 'off' state, which is represented by boolean truth values $s_i \in \{0, 1\}$. We will again denote the bias of the neuron i as θ_i.

The neurons are partitioned into two functional groups: the so-called visible and hidden neurons. The visible neurons provide an interface between the network and its environment. The inputs are given as binary vectors. The hidden neurons on the other hand are trained to represent underlying constraints contained in the input vectors. In the following text, we will assume a sequential numbering of neurons in the network. For a network with n neurons in total and m visible neurons, the visible neurons will be numbered $(1, \ldots, m)$, and the hidden neurons $(m+1, \ldots, n)$. Having m visible neurons and an m-dimensional input vector $\vec{v}$, the visible neurons can be initialized to the corresponding elements of $\vec{v}$, where $\forall i \in (1, \ldots, m) : s_i = v_i$.

The primary goal of a Boltzmann network is to correctly model input patterns according to Boltzmann distribution. The network can thus perform pattern reconstruction. Specifically, provided the network has learned the underlying model correctly, when presented only partial input pattern, it can complete the missing values by computing the missing states of the visible neurons.

A good approach to assessing the performance of a network is to define a measure for evaluating how its internal parameters and structure represent the internal constraints in the input data. Such a measure can then be addressed as a typical best-fit problem, solved by iterative attempts to decrease the value of this measure, until reaching (perhaps the global) minimum. As Hopfield showed in his research [17], energy can be well used as such measure, and is analogously defined in the context of Boltzmann machines. The energy of Boltzmann machine for a configuration of neuron states s can be defined as:

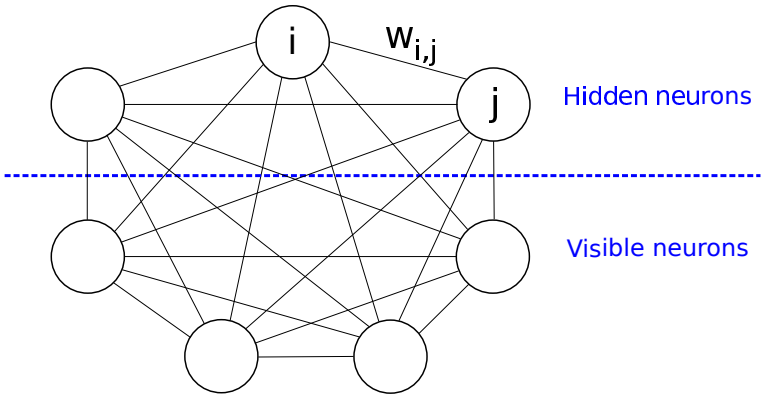


Figure 2.10: An illustration of a Boltzmann Machine with 3 hidden neurons and 4 visible neurons.

$$E(s) = -\sum_{i<j} w_{i,j} s_i s_j - \sum_i \theta_i s_i. \tag{2.44}$$

where s_i are the binary states of neurons, $w_{i,j}$ are the weights between them, and θ_i are their biases. The impact of a single unit's state s_i on global energy can then be computed simply by

$$\Delta E(s_i) = \sum_j w_{ij} s_j + \theta_i \tag{2.45}$$

The energy can be used in the training process using gradient descend method to find the lowest possible energy of the system for the given input.

2.4.3 Restricted Boltzmann Machine

Equation 2.45 suggests an iterative training, where the energy differential ΔE for each state s_i must be computed sequentially. The root cause are the interconnections between the visible and hidden neurons, which make the neuron states dependent on each other. The Restricted Boltzmann Machine (RBM) parallelizes this process by removing these connections, creating a bipartite graph between visible and hidden neurons (see Figure 2.11).

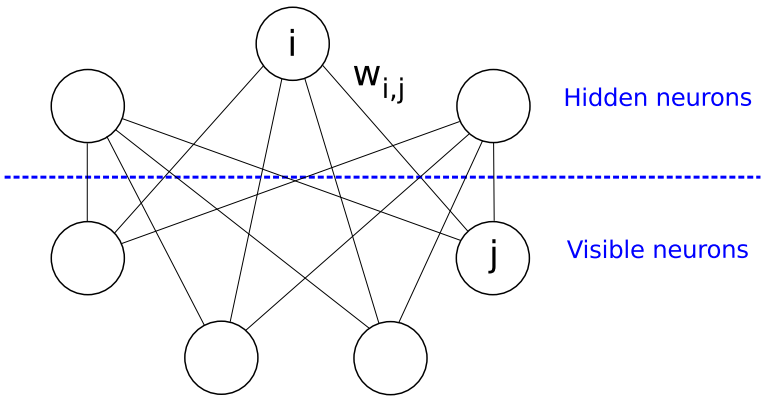


Figure 2.11: An illustration of a Restricted Boltzmann Machine with 3 hidden neurons and 4 visible neurons.

Eliminating these connections results into new energy definitions for a joint configuration of visible and hidden neurons (v, h):

$$E(\vec{v}, \vec{h}) = -\sum_{(i,j)} w_{i,j} v_i h_j - \sum_i a_i v_i - \sum_j b_j h_j. \tag{2.46}$$

$$\Delta E(v_i, \vec{h}) = \sum_j w_{ij} h_j + a_i \tag{2.47}$$

$$\Delta E(\vec{v}, h_j) = \sum_i w_{ij} v_i + b_j \tag{2.48}$$

where v_i, h_j are the binary states of the visible unit i and the hidden unit j, a_i, b_j are their biases, and $w_{i,j}$ is the weight between them. Unlike in the case of the standard Boltzmann Machine (Eq. 2.45), the Restricted Boltzmann Machine does not depend on visible or hidden neurons when computing the energy differential for the visible or hidden neuron, respectively (Eq. 2.47 and 2.48). For this reason we will be using this model and take advantage of its parallelization potential in our implementation.

2.4.4 Training Restricted Boltzmann Machines

RBMs are trained using unsupervised learning. They are not performing classification themselves, but instead they are able to learn to reconstruct data in an unsupervised fashion. Let us assume we have some training set V, a matrix where each row represents the input visible vector $\vec{v}$. The further text in this section we will be inspired by Hinton's Practical guide to training RBMs [14].

RBMs learn by encoding the probability distribution of the input data into the weight parameters. The purpose of training is to maximize the product of probabilities assigned to the training patterns from V, thus we are looking for weight assignment W_m producing the maximal probability:

$$W_m = \max_W \prod_{\vec{v} \in V} p(\vec{v}) \tag{2.49}$$

The RBM assigns a probability to every possible pair of a visible and hidden vector via the energy function defined in Equation 2.46:

$$p(\vec{v}, \vec{h}) = \frac{1}{Z} e^{-E(\vec{v}, \vec{h})} \tag{2.50}$$

where Z is the so-called partition function defined by summing over the energy of all possible neuron states:

$$Z = \sum_{\vec{v}, \vec{h}} e^{-E(\vec{v}, \vec{h})} \tag{2.51}$$

The probability RBM assigns to a visible vector $\vec{v}$ is then defined by summing over all possible hidden states:

$$p(\vec{v}) = \frac{1}{Z} \sum_{\vec{h}} e^{-E(\vec{v}, \vec{h})} \tag{2.52}$$

The probability assigned to a visible vector (i.e. input pattern) can be effectively raised by adjusting the weights and bias to lower the energy of that vector, and to raise the energy of other visible vectors. To determine the desirable weight adjustment we need the derivative of the probability with respect to the adjusted weight, which becomes surprisingly simple for the log probability:

$$\frac{\partial \log p(\vec{v})}{\partial w_{i,j}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \tag{2.53}$$

where the notation $\langle\rangle$ denotes the expectations under the distribution specified by the training data and the model respectively. From the above equation we can directly deduce the rule for performing stochastic steepest ascent in the log probability of the training data:

$$\Delta w_{i,j} = \eta \left(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \right) \tag{2.54}$$

where $\eta > 0$ is the learning rate.

The expectations under the distribution specified by the training data can be obtained easily by sampling the hidden neuron states from the visible ones. Given a training input pattern $\vec{v}$, the probability of the binary state h_i being 1 (the "on" state) is:

$$p(h_j = 1|\vec{v}) = \sigma(b_j + \sum_i v_i w_{i,j}) \tag{2.55}$$

where σ is the activation function. The most commonly used activation function is sigmoid (Eq. 2.2). After sampling the hidden state, $v_i h_j$ represents the expectation under the data distribution in equation 2.54.

$$p(v_i = 1|\vec{h}) = \sigma(a_i + \sum_j h_j w_{i,j}) \tag{2.56}$$

The sampling of the expectations under the distribution specified by the model is much more difficult to obtain. It can be done by randomly initializing the visible states and performing sampling back and forth using Eq. 2.55 and Eq. 2.56. Since the initial visible states are random, it takes a very long time until the RBM converges to the model distribution.

2.4.4.1 Contrastive Divergence

Instead of randomly initializing the visible states, a much more effective approach is used in the constrastive divergence (CD) learning procedure first introduced by Hinton [15]. Note that for readability reasons, we will be labeling the visible vector with v and hidden vector with h, instead of $\vec{v}$ and $\vec{h}$. Since we eventually expect $p_{data}(v) \approx p_{model}(v)$, we can initialize the visible states with input data, and reconstruct the model expectations from them. This results in a much faster convergence to the model distribution. The CD algorithm can be summarized by the below steps, which are iterated over each sample in the training dataset:

1. Pick training sample v and clamp it onto the visible neurons (binary input is assumed).

2. Compute the probabilities of hidden neurons p_h by multiplying the visible vector v with the weights matrix W as $p_h = \sigma(v \cdot W)$ (see Eq. 2.55).

3. Sample the hidden states h from the probabilities p_h.

4. Compute the outer product of vectors v and p_h, let us call it positive gradient $\phi^+ = v \cdot p_h^T$.

5. Sample a reconstruction of the visible states v' from the hidden states h (see Eq. 2.56). Then resample the hidden states h' from the reconstruction of the visible states v'. (Gibbs sampling step)

6. Compute the outer product of v' and h', let us call it negative gradient $\phi^- = v' \cdot h'^T$.

7. Compute the weight updates as the positive gradient minus the negative gradient: $\Delta W = \eta(\phi^+ - \phi^-)$.

8. Update the weights with new values: $w'_{i,j} = w_{i,j} + \Delta w_{i,j}$.

The Gibbs sampling step can be repeated multiple times to converge closer to the model distribution. The number of repetitions is often included in the abbreviation of the algorithm, where for k repetitions the algorithm is referred to as CD-k [15, 14]. Experiments reported so far show, that CD-1 is already producing very decent results [5].

2.4.4.2 Persistent Contrastive Divergence

At present, constrastive divergence is one of the most popular gradient approximations for RBMs. However, there are multiple alterations to the standard CD algorithm, and it is not obvious which is the best one. A very common alternation is Persistent CD, abbreviated as PCD. Some research claims that PCD produces more meaningful feature detectors, and outperforms the other variants of CD algorithms [35].

PCD algorithm uses a different approximation for sampling states than CD. It eliminates Step 3 when compared to standard CD described in the above steps. This step initializes the hidden states based on the input pattern. As a result, the sampling chain is being restarted for every observed pattern. Instead, PCD initializes the hidden states only once at the beginning of the training. This causes the single chain of sampling throughout the whole training, which helps move faster towards the model distribution p_{model} rather then p_{data}. The smaller the learning rate, the better PCD works [35]. It is because the smaller parameter updates are then small enough compared to changes in the sampling chain (mixing rate of Markov chain), and the chain can easier catch up to the changes in the model.

2.4.5 Architecture of a Deep Belief Network

The Restricted Boltzmann Machines themselves are capable of detecting and extracting features from input data. Several layers of RBMs can be stacked one onto each other to form a multilayer network [16]. Each RBM layer uses

the hidden neurons from preceding RBM layer as its input (see Figure 2.12). The deep architecture with multiple layers can then extract deep hierarchical representation of the training data.

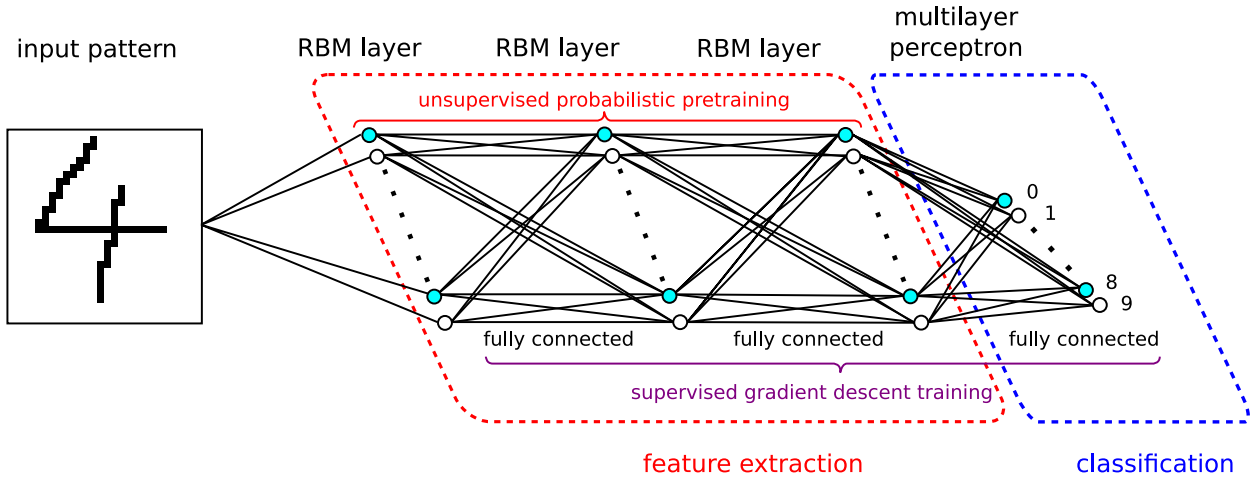


Figure 2.12: Architecture of a Deep Belief Network.

A set of RBM layers performs the feature detection task, while the classification task is performed by using a multilayer perceptron as the last layer. The last MLP layer is again using the hidden neurons of the preceding RBM layer as its input. The resulting architecture is a mixture of probabilistic neurons in the feature extraction phase, and deterministic neurons in the classification phase.

2.4.6 Training a Deep Belief Network

The DBN architecture consists of RBM and MLP layers. However, RBMs employ unsupervised learning, while MLPs employ supervised learning. This results into a two-phase training process.

The first phase, called pretraining, performs unsupervised training of each RBM layer separately [16, 4], as described in Section 2.4.4. The second phase uses the gradient descend method for supervised training of the MLP as well as RBM layers. It can be implemented using, for example, the standard backpropagation algorithm described in Section 2.2.2.

The pretraining and training process can be summarized as follows:

1. The network is initialized with small random weights, biases and other parameters.

2. The first RBM layer is initialized with input data representing potentials in its visible neurons. Then the unsupervised training is performed on this layer iterating over the training dataset for predefined number of epochs.

3. The next layer obtains its input by sampling the potentials generated in the hidden neurons of the previous layer. Then the unsupervised training is performed on this layer iterating over the training dataset.

4. Iterate the previous step for the desired number of layers. In each iteration the samples are propagated upwards deeper into the network. The pretraining phase is finished when the first MLP layer is reached.

5. Fine-tuning via supervised gradient descend starts (see algorithm in Section 2.2.2). The training is stopped after reaching a predefined number of epochs, or is finished successfully after reaching the target error rate.

Note that structurally, the neurons of the DBN are interconnected in the same manner as the MLP network (see Figure 2.12). This allows the second phase of training to be performed exactly as if training MLP network. Therefore the whole training procedure is equivalent to initializing the weights and biases of a deep MLP network with the values obtained in the unsupervised probabilistic pretrainig.

After the network is trained, the classification of the presented input data is performed exactly like in the case of MLP network.

3. Implementation

The main goal of this thesis consists of an efficient implementation of 3 different models of deep neural networks. Being the simplest one among the implemented models, the architecture of the Multilayer Perceptron (MLP) was described in Section 2.2. Convolutional Neural Networks (CNNs) were discussed in Section 2.3. The last implemented model represents the Deep Belief Networks (DBNs) described in Section 2.4. Each model has slightly different applications depending on the required speed, size, and error rates.

While MLPs are relatively simple to implement, the number of their neural inter-connections grows exponentially with the number of their neurons. For this reason they can quickly become robust. CNNs on the other hand are not fully inter-connected, and share some connection weights. This facts allows to significantly optimize network performance when processing high resolution images. DBNs are able to also recognize abstract features. These different properties will be discussed in more detail at the end of this chapter. The testing of the model implementations should provide some interesting comparisons of the different network architectures.

In Section 3.1, I will first discuss the available hardware platforms, which can be used for parallel processing and are suitable for solving the problems of artificial neural networks. From the presented options, I will choose one hardware platform, which I will use to implement all 3 neural network models. The implementation should be flexible enough to enable processing of user-defined input data of variable size and structure. Section 3.2 will state the requirements imposed on the implementation and explain them in more detailed. The following Section 3.3 will be focused on the high-level architectural overview of the implemented application, describing its modules and how they complement each other. Section 3.4 will walk guide the reader through the process of installation, verification, and launch of the implemented software. The next Section 3.5 will describe the available configuration options to customize network design, training, validation and testing process. The following Section 3.6 will define the supported input data formats. Section 3.7 will be split into subsections, each one of them dedicated to a specific application module describing some key concepts of its implementation.

In the text of this chapter, I will often refer to a graphic card with CUDA support as a *device*, and the CPU with standard RAM memory as a *host*. This jargon is present in the CUDA documentation, as it is prevalent in CUDA communities.

3.1 Parallel computing platforms

The network models will be implemented on a single-threaded CPU and on a parallel computing platform. There are a few candidates to choose the parallel platform from: the so-called Field-Programmable Gate Arrays FPGA, NVDIA CUDA, and AMD Stream. In the below section we will briefly describe each of them and choose the most suitable platform for the network models implemented in this thesis.

3.1.1 FPGA

The so-called Field-Programmable Gate Arrays (FPGAs) form an integrated circuit that can be programmed to rewire itself to be hardware-optimized for a given task. The research and development of FPGAs started in the late 1980s. It was funded as an experiment by the Naval Surface Warfare Department, and first patented in 1992. The circuitry in FPGAs is programmed by means of the Hardware Description Language (HDL) specifically developed for the task. The work with HDL is, however, tedious, because the resulting design is quite difficult to visualize.

The architecture of Graphics Processing Units (GPUs) conceptually differs from FPGAs most importantly because GPUs run software. The instructions need to be fetched and stacked, perform computational operations and send the results to the operating memory. On the other hand FPGAs are designed to provide a higher level of parallelism, which can be taken advantage of and lead to a better performance.

The key criteria for the choice of the most suitable platform for our task are cost and performance – cost being the most important one. Each architecture might yield different performance results for different problems. A universal computing performance is measured in instructions per seconds. Computing the throughput in neural networks requires mostly floating point operations. Therefore, a more accurate approach is to compare the performance in floating point operations per second (FLOPs). For this reason, I will choose the best performing model based on FLOPs and compare their peak performance and price.

model / property	**peak 32bit performance**	**peak 64bit performance**	**price**
Virtex-7 FPGA	1700 GFLOPs	671 GFLOPs	$ 11 995
NVIDIA GTX Titan Z	8122 GFLOPs	2707 GFLOPs	$ 1 500
AMD Radeon HD 7990	8200 GFLOPs	1894 GFLOPs	$ 695

Table 3.1: Comparison of performance and cost of GPU and FPGA parallel architectures.

As you can see in Table 3.1, the FPGA architecture is significantly more expensive than GPU architecture from both its vendors. While FPGA might perform better for some problems, the FLOPs that I am interested in are in favour of the GPU architecture. The comparison yields a clear conclusion about the performance and price ratio. There are, however, more aspects, which I will summarize in Table 3.2 without going into too much detail. As this thesis is about the implementation, an important argument for the choice of the hardware platform is also the quality of the development toolkit. The community grouped around the platform is thus also important, because it predicts the future efforts devoted to the platform. The implementation in this thesis will benefit from these improvements, as we expect to use the latest APIs so compatibility with the upcoming releases is highly probable.

When compared to FPGAs, GPUs are superior except for power consump-

tion. For instance, GTX Titan Z from NVIDIA has its peak consumption at 375 Watts. At current electricity prices ($0.3 per 1 kWh) operating this device costs $0.11 per hour at full utilization. Compared to other expenditures on staff and hardware supporting the device, the operation costs are negligible. The conclusion is therefore clear, GPUs are more suitable for our task to implement artificial networks.

criterion / architecture	**FPGA**	**GPU**
cost	-	+
performance (FLOPs)	-	+
development toolkit	-	+
community size	-	+
power consumption	+	-
availability	-	+

Table 3.2: A brief comparison of GPU and FPGA parallel architectures. Superiority of the platform with regard to the considered criterion is marked with a plus sign, inferiority with a minus sign.

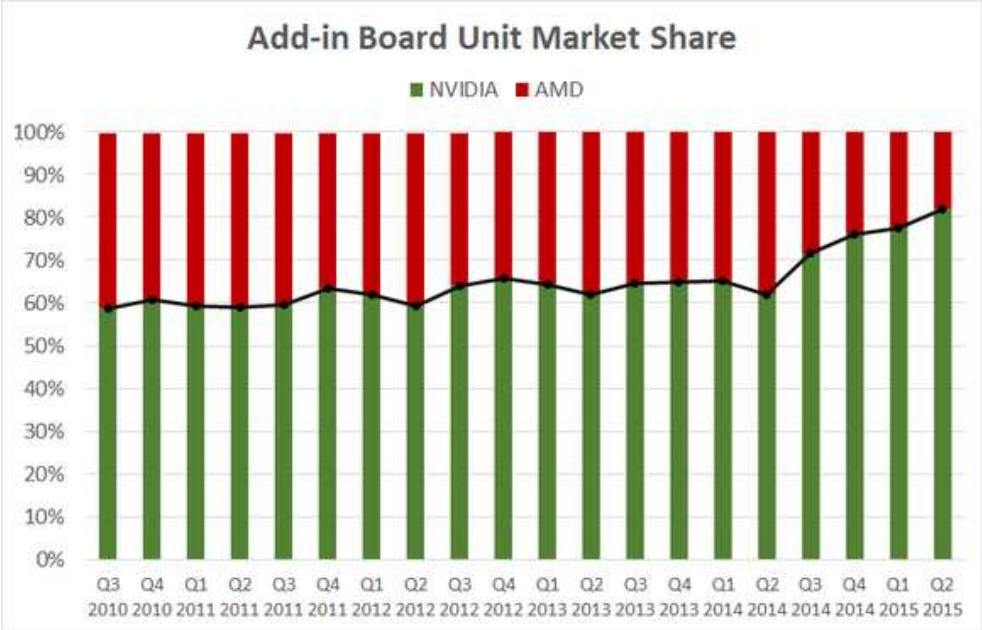


Figure 3.1: Shares of NVIDIA and AMD on GPU market. Research was carried out by Jon Peddie Research and Mercury Research, graph provided by The Montley Fool [10].

The presented comparisons justify the choice of GPU architecture for the implementations in this thesis. GPUs are primarly used for gaming, where parallel operations are suitable for computing multi-dimensional graphic objects. Gaming represents a big market, which allows for an affordable price. There are two

mainstream vendors of GPUs - AMD and NVIDIA. The parallel framework for working with the GPU cards from AMD is Stream, and the framework provided by NVIDIA is named CUDA. Both vendors are strong competitors with a big market share. NVIDIA started to increase their share recently [10] as can be seen in Figure 3.1.

3.1.2 NVIDIA CUDA

CUDA is a complete parallel computing platform consisting of multi-purpose graphic cards and a programming model invented by NVIDIA. CUDA is released as a toolkit supporting all major operating systems (Linux, OS X, Windows). The core of the tookit is CUDA Runtime API and CUDA Driver API used for low-level programming and access to the device. Optimized mathematical functions provided by NVIDIA are included in CUDA Math API. NVIDIA invested heavily into supporting researchers, providing additional libraries:

- **cuBLAS** basic linear algebra subprograms,
- **cuFFT** Fast Fourier Transform,
- **cuRAND** efficient generators of pseudorandom and quasirandom numbers,
- **cuSPARSE** computing with sparse matrices,
- **cuDNN** primitives for deep neural networks.

CUDA offers three programming languages to choose from for development, namely CUDA C, CUDA Fortran, and Python. The syntax of the first two languages is a superset of native language constructs of C/C++ and Fortran respectively, additionally providing extra libraries and extra hints for the CUDA compiler.

3.1.3 AMD FireStream

AMD FireStream is a brand name for the Radeon-based product line targeting general purpose computing leveraging the parallel processing power of AMD GPUs. AMD is a member of the Khronos group, a not-for-profit member-funded industry consortium standardizing cross-platform parallel programming (OpenCL). OpenCL is an attractive framework that is supported by many different vendors and architectures (including FPGA), thus prevents vendor lock-in. AMD releases the Accelerated Parallel Processing SDK (APP SDK) to provide the developers with documentation, samples, libraries and other materials to get started with OpenCL.

3.1.4 Comparison of AMD and NVIDIA

In Table 3.1 we can see that the performance difference measured in FLOPs for single precision is negligible. The cost is slightly more favourable for AMD products. For the double precision case, NVIDIA seems to perform better. This

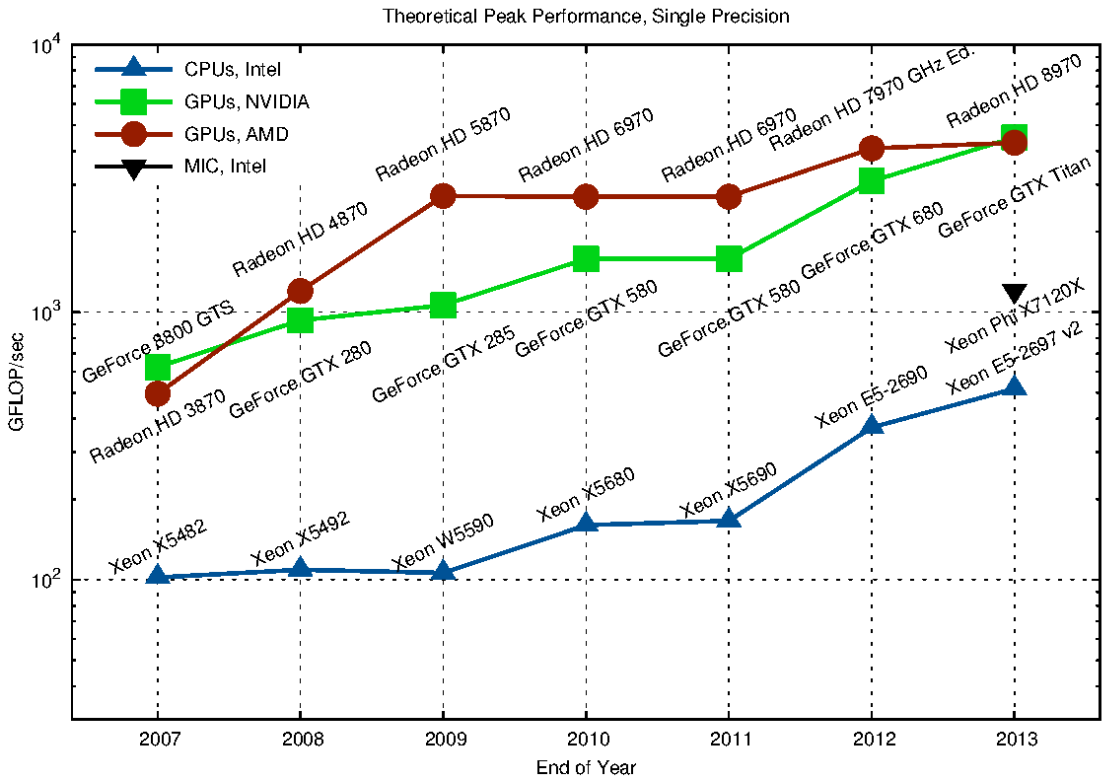


Figure 3.2: Historical evolution of 32bit FLOPs in GPU products of NVIDIA, AMD, Intel and CPU products of Intel [30].

comparison is, however, based only on the currently best performing models of the respective vendors.

A more objective overview [30] over their performance is displayed as a historical evolution in Figure 3.2. Here we can see that AMD had a significant lead in 2009, but NVIDIA quickly caught up. Intel joined the market in 2013 and started putting embedded GPUs into its line of CPUs, which makes them a cheap bundle. While being enough for rendering graphics, the performance still lacks behind separate dedicated devices.

While AMD and NVIDIA have comparable performance for single precision, in the case of double precision Figure 3.3 shows that NVIDIA is performing somewhat better [30]. In 2009 NVIDIA did not consider the market for double precision to be big enough. However, a year later they decided to put double precision into consumer GPUs and made a huge leap towards improved performance, which defeated AMD in 2012.

To summarize our considerations, NVIDIA products have a slightly higher price, and similar performance with a small lead of NVIDIA in the field of double precision computing. Although AMD uses an open vendor-independent standard OpenCL supporting several architectures, NVIDIA provides a much better development toolkit. For this reason, I chose the NVIDIA CUDA SDK for the implementational part of this thesis.

CUDA SDK supports several programming languages to communicate with the involved devices. Available options are C/C++, Fortran, and Python. Python

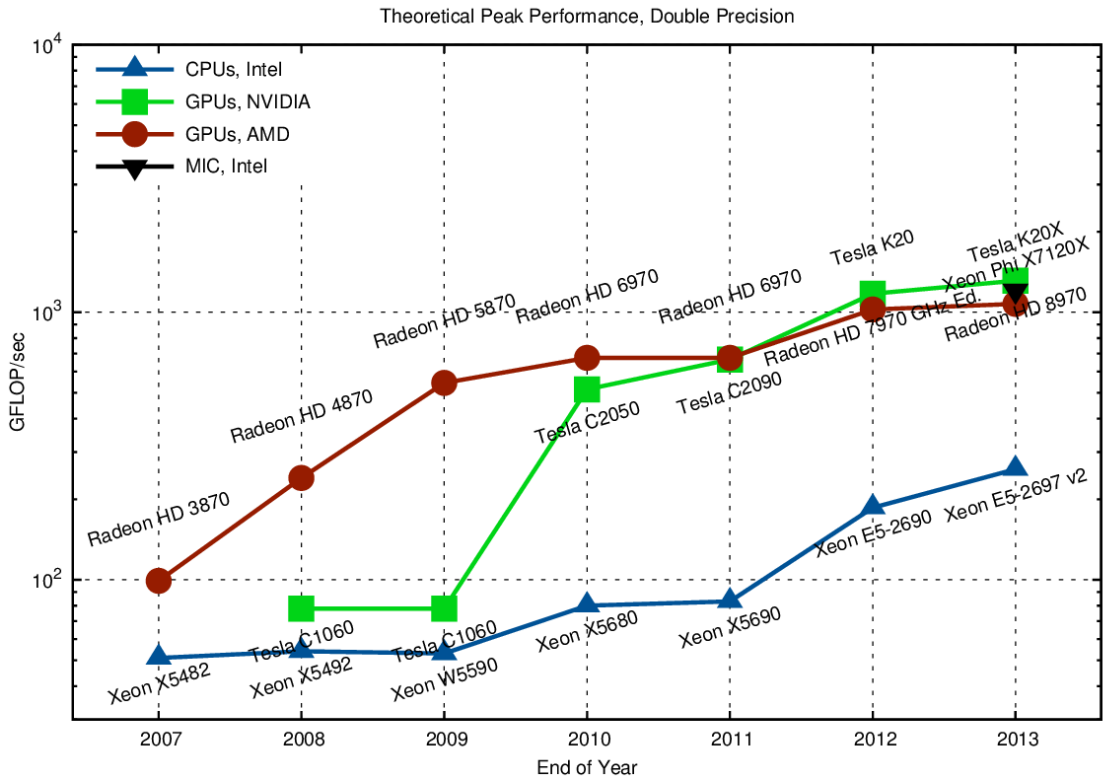


Figure 3.3: Historical evolution of 64bit FLOPs in GPU products of NVIDIA, AMD, Intel and CPU products of Intel [30].

is a high-level programming language. With the aim to be able to optimize the usage of parallel hardware, I will rather prefer a low-level programming language instead. Although Fortran is used mainly for scientific computing, unfortunately, it does not support object-oriented programming (OOP). OOP is on the other hand important for an easily readable code supporting concepts like encapsulation, polymorphism, abstraction and decoupling. Moreover, Fortran is not popular outside of the scientific community, leaving C++ with a better community support. For these reasons I decided to prefer C++ and use it for the implementation of artificial networks.

3.2 Requirements

Before diving into the implementational details, let us impose some requirements that will serve as a guidepost when facing major architectural decisions.

- efficiency
  - deep neural networks generally require a long time to be trained
  - many computationally intensive operations are repeated several thousand/million times
  - even a minor optimization can make a difference of days of runtime

- flexibility
  - support for multiple input data formats
  - support human-readable as well as optimized machine-only-readable binary data format
- extensibility
  - use the Object Oriented Programming approach to support abstraction resulting in an easily readable code
  - use a loosely coupled architecture with well separated concerns
  - use comments documenting code
  - provide high-level documentation
- customization
  - provide rich configuration options
- convention over configuration
  - the user should not be overwhelmed by a complex configuration options
  - use defaults for missing configuration parameters
  - derive smart defaults based on other configuration properties

3.3 Architecture

All the models of neural networks to be implemented within the framework of this thesis share several concepts common to feed-forward artificial neural networks. All of them consist of neurons, weighted connections between them, and layers of neurons organized into the network. The respective learning algorithms also work in a very similar manner, which will be discussed in more detail in Section 3.7.5.

For consistency reasons, the data format serving as the input for the network should be as universal as possible. For these reasons I decided to create one single application implementing all three models. It will allow me to reuse the code and save some significant implementation efforts. For instance, all networks will use the same datasets, the same output logging, and the same error computations. The differences in specific implementations of the models will be dealt with using an object-oriented approach, in particular polymorphism, loose coupling, and good separation of concern. More details on this topic can be found in Section 3.7 describing the different application modules specifically.

I named the application `deepframe` and will refer to it by this name in the following text. The name is derived from the fact that it is essentially a framework for building and training deep neural networks. It is released under the GNU Affero GPL v3.0 license and is publicly available on GitHub at `https://github.com/janvojt/deepframe`. The `deepframe` application is architecturally divided into several modules depicted in Figure 3.4. Each of the modules consists of objects modeling the behaviour specific for the given business logic.

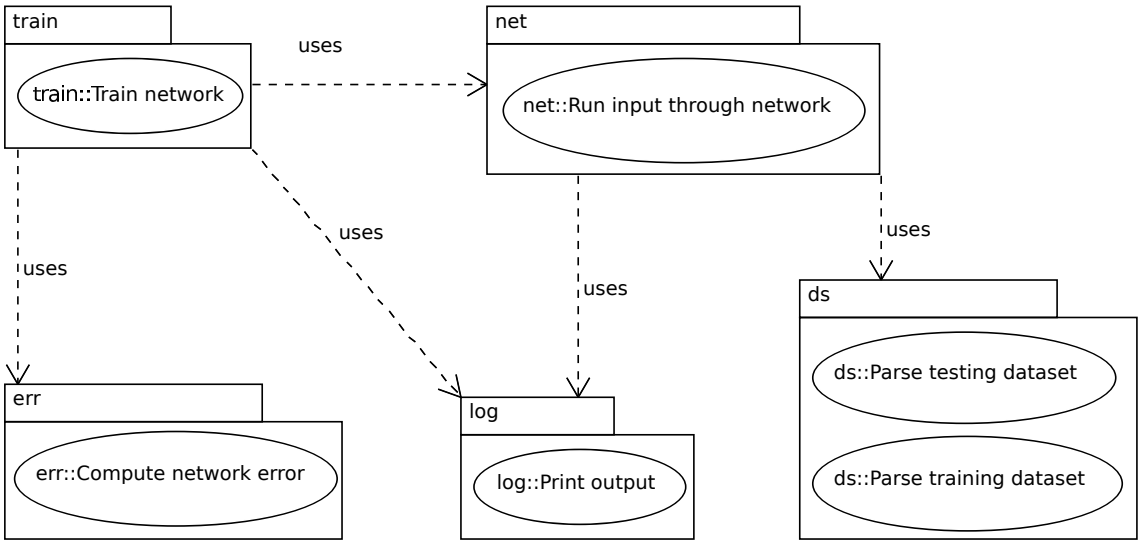


Figure 3.4: A simplified high-level package diagram showing the decomposition of application logic into separate packages and their respective use cases.

3.4 Installation procedure and launch

The `deepframe` application is written in C++ and does not use any platform-specific operations. Therefore it is possible to build and run it on any Operating System (OS). In this thesis, Linux OS was used both for development and testing. For this reason the build script bundled with the application works only in Linux. I decided to use this OS, because it is open, stable, secure, and free. Furthermore, unlike Windows/OSX for example, it can be setup and run without a desktop manager. It saves resources and allows for independent testing with as little irrelevant processes as possible. It is also relatively popular among the scientific community. Furthermore, some of NVIDIA hardware setups targeted for developers run only Linux, as is the case for currently the most powerful desktop deep learning system [27].

The purpose of this thesis is not to create a cross-platform compiler. However, if it is necessary to run the application on a different platform, a platform specific build script can be created. To find out details about how the application is built, please consult the `Makefile` located in the application root folder.

Also, several bash scripts are provided that can be used to run the tests conducted in Chapter 4. These prepared scripts are, however, not guaranteed to run on a different OS. They are not necessary to run the application, but I still recommend to compile and run on Linux.

To build and run the application, the following libraries must be installed in the system:

- **log4cpp** logging framework for C++,
  - library for flexible logging to files, syslog, IDSA and other destinations
  - based on the successful Log4j Java library, staying as close to their API as is reasonable

  - chosen mainly because of having the functionality of deffered string building
  - used for generating the output of the application (includes log messages about network configuration, learning process, validation, and testing)

- **gtest** testing framework from Google Inc.,
  - based on the xUnit architecture
  - supports automatic test discovery, a rich set of assertions, user-defined assertions, death tests, fatal and non-fatal failures, value- and type-parameterized tests, and XML test report generation
  - used for unit testing to aid development and early discovery of regressions bugs

- **cuRAND** set of random number generators working on GPU,
  - included in CUDA SDK
  - provides facilities that focus on a simple and efficient generation of high-quality pseudorandom and quasirandom numbers
  - random numbers can be generated on the GPU device or on the host CPU
  - used for efficient random initialization of connection weights and biases

- **cuBLAS** Basic Linear Algebra Subprograms implementation for GPU.
  - included in CUDA SDK
  - GPU-accelerated version of the complete standard BLAS library
  - (currently) delivers 6x to 17x faster performance than the latest Math Kernel Library implementing BLAS on CPU
  - leveraged for computing linear algebra operations (e.g. matrix multiplication in forward network run)

The application build is defined in a standard `Makefile`. The build itself is triggered using the make utility. The resulting executable files are generated in the `bin` folder. Application is then launched by executing the file `bin/deepframe`. Another generated executable `bin/deepframe-test` runs unit tests. I recommend to launch the unit tests prior to launching the application for the first time after fresh installation. These tests will perform a basic functionality verification printing the test results to standard output (`stdout`).

The CUDA C/C++ framework is used for leveraging the GPU for parallel computations. Parallelized parts of the application are written in CUDA C++, and because of this the build process must use NVIDIA CUDA compiler `nvcc`. If CUDA SDK is not installed in standard path (`/usr/local/cuda`), the correct path must be set in the environmental variable `CUDA_HOME`.

The application source code is versioned in Git - a distributed revision control. Once all the dependencies listed above are satisfied, to setup a local copy including sources, and build the application, the following commands can be issued using a Git client in console:

```
$ git clone git@github.com:janvojt/deepframe.git
$ cd deepframe
$ make install
```

Snippet 3.1: Shell commands used to download and compile the deepframe application.

If the build proceeded without errors, first run the tests by issuing the following:

```
$ ./bin/deepframe-test
Running main() from gtest_main.cc
[==========] Running 7 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 6 tests from Network
[ RUN      ] Network.NeuronCounters
[       OK ] Network.NeuronCounters (0 ms)
[ RUN      ] Network.InputSetTest
[       OK ] Network.InputSetTest (0 ms)
[ RUN      ] Network.SimpleRun
[       OK ] Network.SimpleRun (0 ms)
[ RUN      ] Network.SimpleWeightTest
[       OK ] Network.SimpleWeightTest (0 ms)
[ RUN      ] Network.WeightsOffsetTest
[       OK ] Network.WeightsOffsetTest (0 ms)
[ RUN      ] Network.NeuronInputOffsetTest
[       OK ] Network.NeuronInputOffsetTest (0 ms)
[----------] 6 tests from Network (0 ms total)

[----------] 1 test from SimpleInputDataset
[ RUN      ] SimpleInputDataset.BinaryDatasetCreation
[       OK ] SimpleInputDataset.BinaryDatasetCreation (0 ms)
[----------] 1 test from SimpleInputDataset (0 ms total)

[----------] Global test environment tear-down
[==========] 7 tests from 2 test cases ran. (0 ms total)
[  PASSED  ] 7 tests.
```

Snippet 3.2: An example output of running unit test verifying correct behaviour of the deepframe application.

The above snippet contains an example output, where all the tests passed successfully. In case of an error tests fail, information about the problem is printed, and you need to investigate the cause of the problem. If using a stable release the usual cause are unsatisfied dependencies specified at the beginning of this Section. Once the tests are passing successfully, proceed with running the actual application:

```
$ ./bin/deepframe
2015-02-16 23:32:50 [INFO] : Seeding random generator with 85108201.
2015-02-16 23:32:50 [INFO] : Using CPU for computing the network runs.
2015-02-16 23:32:50 [INFO] : Started training with limits of 100000 epochs
    and target MSE of 0.000100.
2015-02-16 23:33:14 [INFO] : Training successful after 42856 epochs with MSE
    of 0.000100.
[ 0, 0 ] -> [ 0.0100136 ]
[ 0, 1 ] -> [ 0.988549 ]
```

```
[ 1, 0 ] -> [ 0.990581 ]
[ 1, 1 ] -> [ 0.00893504 ]
```

Snippet 3.3: An example output of teaching a multilayer perceptron network compute XOR operator on CPU.

Running the application with no arguments will set default values for all the options. This means running a simple multilayer perceptron network with one hidden layer consisting of two neurons and learning the XOR dataset. The learnt results are printed on standard output. As you can see in the snippet, the network successfully learnt the XOR bit operator. As the second line in the output says, all the computations were run on CPU. To compute on GPU, we can run the application with the option `-p`:

```
$ ./bin/deepframe -p
2015-02-16 23:42:54 [INFO] : Seeding random generator with 507738670.
2015-02-16 23:42:54 [INFO] : GPU Device 0: "GeForce GTX 560" with compute
    capability 2.1.
2015-02-16 23:42:54 [INFO] : Using GPU for computing the network runs.
2015-02-16 23:42:54 [INFO] : Started training with limits of 100000 epochs
    and target MSE of 0.000100.
2015-02-16 23:43:43 [INFO] : Training successful after 43165 epochs with MSE
    of 0.000100.
[ 0, 0 ] -> [ 0.0110211 ]
[ 0, 1 ] -> [ 0.990521 ]
[ 1, 0 ] -> [ 0.990493 ]
[ 1, 1 ] -> [ 0.00991258 ]
```

Snippet 3.4: An example output of teaching a multilayer perceptron network compute XOR operator on GPU.

This time, we can see the GPU hardware, as well as its CUDA compute capability.

3.5 Network configuration

The application can be configured by providing command line options. The options are parsed by a standard shell utility getopts. Each option has a single-letter variant, as well as a more verbose long alternative. To list all the available options, launch the application with the `--help` option.

```
$ ./bin/deepframe -h
Usage: deepframe [OPTIONS]

Option     GNU long option     Meaning
--------------------------------------------------------------------------------
-h         --help              This help.

-b         --no-bias           Disables bias in neural network. Bias is
    enabled by default.

-l <value> --rate <value>      Learning rate influencing the speed and
    quality of learning. This is a global setting used in case of MLP
    configured via options. The external layer configuration file overrides
    this setting and allows to assign a different learning rate for each
    layer. Default value is 0.3.
```

```
-a <value> --init <value>      In case of uniform distribution, minimum and
    maximum value network weights are initialized to. In case of Gaussian
    distribution, the standard deviation. Default is uniform distribution
    with interval (-1,1).

-e <value> --mse <value>       Target Mean Square Error to determine when to
    finish the learning. Default is 0.01.

-k <value> --improve-err <value> Number of epochs during which improvement of
    error is required to keep learning. Default is zero (=disabled).

-m <value> --max-epochs <value> Sets a maximum limit for number of epochs.
    Learning is stopped even if MSE has not been met. Default is 100,000

-n <value> --pretrain <value>  Configures the number of pretraining epochs
    for Deep Belief Network. Default is zero (no pretraining).

-f <value> --func <value>      Specifies the activation function to be used.
    Use 's' for sigmoid, 'h' for hyperbolic tangent. Sigmoid is the default.

-c <value> --lconf <value>     Specifies layer configuration for the MLP
    network as a comma separated list of integers. Alternatively, it can
    contain a path to configuration in an external file. Default value is
    "2,2,1".

-s <value> --labels <value>    File path with labeled data to be used for
    learning. For IDX format separate the data and labels filepath with a
    colon (":").

-t <value> --test <value>      File path with test data to be used for
    evaluating networks performance. For IDX data with labels for testing
    dataset separate the data and labels filepath with a colon (":").

-v <value> --validation <value> Size of the validation set. Patterns are
    taken from the training set. Default is zero.

-q <value> --k-fold <value>    Number of folds to use in k-fold cross
    validation. Default is one (=disabled).

-o         --best-fold         Uses the best network trained with k-fold
    validation. By default epoch limit is averaged and network is trained on
    all data.

-i         --idx               Use IDX data format when parsing files with
    datasets. Human readable CSV-like format is the default.

-r <value> --random-seed <value> Specifies value to be used for seeding
    random generator.

-j         --shuffle           Shuffles training and validation dataset do
    the patterns are in random order.

-u <value> --use-cache <value> Enables use of precomputed lookup table for
    activation function. Value specifies the size of the table.

-p         --use-gpu           Enables parallel implementation of the network
    using CUDA GPU API.
```

```
-d        --debug            Enable debugging messages.
```

Snippet 3.5: The list of available options for configuring deepframe application with the explanation of their respective purposes.

The options that are found are processed sequentially from the first to the last one. In case of conflicts or duplications, the later option always overrides the former one. The network configuration might be quite complex in some cases. For this reason, I recommend to use shell script templates dedicated for certain datasets or tasks with default configuration. Thanks to the possibility of overriding former options, these default settings can be easily overridden when calling the script templates. Some useful configuration templates may be found in the `examples` folder.

The application business objects are built and configured based on the above options in the `main.cpp` file, which represents the application entry-point. After the options are parsed from command line, network configuration is created. The configuration relevant to network is represented by the `NetworkConfiguration` class. If GPU is selected for the computation, it also has to be configured specifically for the given task and for the given hardware. These configuration properties are represented by the `GpuConfiguration` class. Both configuration representations are created in their dedicated factory methods.

Another responsibility of the configuration factory is also to seed the pseudo-random generators. CPU generator uses the `srand` function from the standard C library. In case of generation on GPU, the cuRAND library is leveraged and the pseudo-random generator is built using the following code:

```
1  GpuConfiguration *gpuConf = GpuConfiguration::create();
2  curandGenerator_t *gen = new curandGenerator_t;
3  curandCreateGenerator(gen, CURAND_RNG_PSEUDO_DEFAULT);
4  curandSetPseudoRandomGeneratorSeed(*gen, conf->seed);
5  gpuConf->setRandGen(gen);
```

Snippet 3.6: The IDX data format definition.

After the configurations are built, the factory method creates the neural network and initializes its parameters (weights and bias). The next step involves parsing of the the datasets. Afterwards, the factory for the learner creates the backpropagation learner. The actual training process is then triggered by calling the `train` method of the learner object, while passing the training and validation datasets as arguments.

After the network is learnt (meets the target mean square error) or the maximum number of epochs is reached, training is stopped and the testing phase starts. The network is presented with the testing datasets and the output is printed to the standard output. Finally, the application releases all acquired memory and terminates.

3.5.1 Layer configuration

An important part of the network design is the configuration of the layer setup. Configuring MLP can be done easily via the command-line option `--lconf` or its

shorthand alternative `-c`. The parameter takes a comma-separated list of integer values, where each integer corresponds to the number of neurons in the layer given by the position in the list. For example, `-l 2,5,1` configures input layer with 2 neurons, one hidden layer with 5 neurons, and an output layer with 1 neuron.

The configuration of an MLP is quite simple, but the configuration of other more complex layers is done via external configuration files. The path to the configuration file is again specified via the option `--lconf`. The configuration syntax obeys the following rules:

- a line starting with a hash sign ("#") is a comment,
- if a line is not a comment, it represents a layer configuration,
- layer configuration starts with a layer type followed with a ":",
- the rest of the layer configuration is layer-specific.

The recognized layer types and their specific configurations are listed below. All configurable parameters are separated by a colon (":").

- FullyConnected
  1. number of output neurons (integer value)
  2. learning rate (float value)
  3. bias (boolean value)
- Convolutional
  1. width of the convolution window (integer value)
  2. height of the convolution window (integer value)
  3. number of feature maps (integer value)
  4. learning rate (float value)
  5. bias (boolean value)
- Subsampling
  1. width of the subsampling window (integer value)
  2. height of the subsampling window (integer value)
  3. learning rate (float value)
- Rbm
  1. number of hidden neurons (integer value)
  2. use of persistent contrastive divergence (boolean value)
  3. number of steps in Gibbs sampling (integer value)
  4. learning rate (float value)
  5. bias (boolean value)

For examples with comments explaining different configuration setups you can consult directory `examples` in the root of the project. Out of the box included configurations use `cfg` file extension (not mandatory).

3.6 Data format

The implemented feed-forward network uses supervised learning. Therefore in the training phase, the data with input patterns with their respective labels are needed. These are read from a file on the filesystem. Path to the training data is specified in argument to the option `--labels`. Both relative and absolute paths are allowed.

The default data format is a customized CSV format. Data is stored in human-readable plain text. Each line consists of a single pattern with its respective label. The pattern is stored as a comma separated list of values. These values will be set as a potential of input neurons. The label is then separated by an arrow `->`, which designates that the expected output follows. The label itself is again a list of comma separated values. These represent the ideal values of output neurons. The testing phase then does not need labels for the data. The data format used is a simple CSV format with a comma separated list of values representing input pattern.

The benefit of the plain text format is that it is human-readable. However the disadvantage is bad storage efficiency. When working with big datasets a lot of storage can be saved by using binary data files. Also, the data parsing process is more efficient, therefore the application run time can be improved. The `deepframe` application can parse IDX binary data format, which is a simple format for vectors and multidimensional matrices of various numerical types. IDX format is also used by the MNIST dataset - the most popular dataset with handwritten digits. The basic IDX format for a dataset with `n` dimensions is [23]:

```
magic number
size in dimension 0
size in dimension 1
...
size in dimension n-1
data
```

Snippet 3.7: The IDX data format definition.

The magic number is an integer written in the most significant bit first format. The first two bytes are always zero. The third byte codes the data type in the following manner:

```
0x08: unsigned byte
0x09: signed byte
0x0B: short (2 bytes)
0x0C: int (4 bytes)
0x0D: float (4 bytes)
0x0E: double (8 bytes)
```

Snippet 3.8: Meaning of the magic number in IDX data format.

The fourth byte codes the number of dimensions (`n`) of the matrix. The sizes in each dimension are always four-byte integers, written in the most significant bit first format and in high endian. The magic number and the dimension sizes are then followed by the data itself. No data separators are needed, because the dimensions are defined at the beginning of the file. This approach provides universal format with very effective storage capabilities. To parse IDX data files

use the `--idx` option.

3.7 Module description

The `deepframe` application is built of the modules depicted in Figure 3.4. In this section, I will describe the role of each module in more detail. For simplicity I will be using the term "interface" when describing the set of public methods the objects or components use to mutually cooperate. The language syntax of C++ does not include the term, but is in fact represented by a header file.

3.7.1 Dataset module

The dataset module is responsible for reading, parsing and representing the processed datasets. The central part of the module is the model of an input dataset represented by interface `InputDataset`. It contains only the methods necessary for fetching the input patterns:

```
1  // Returns a pointer to an array of next input values in the dataset.
2  virtual double *next() = 0;
3  // Tells whether the dataset contains more input patterns to process.
4  virtual bool hasNext() = 0;
5  // Resets the cursor to the beginning of the dataset.
6  virtual void reset() = 0;
7  // Returns the dimension of input patterns.
8  virtual int getInputDimension() = 0;
```

Snippet 3.9: Methods provided by InputDataset interface.

All other classes modeling the datasets inherit from this interface and provide custom implementation of the methods. Probably the most trivial implementation of the dataset is storing the data in an array and reading from it. This is the case of `SimpleInputDataset`. It is, however, possible to provide any other suitable implementation, for example reading from a live stream of an unknown length. Notice that this is possible because the `InputDataset` interface does not have any method returning its size.

The representation of training datasets, however, needs labels corresponding to desired output values for input patterns to make supervised training possible. This is addressed by the `LabeledDataset` interface. It inherits the methods from `InputDataset` and defines the following extra method for getting the dimensions of the label:

```
1  // Returns the dimension of labels for input data.
2  virtual int getOutputDimension() = 0;
```

Snippet 3.10: Methods provided by LabeledInputDataset interface.

The label itself is then read from the memory pointed to by the return of `next()` method. The datasets are created by parsers. Their responsibility is to read the data from external locations, parse it, and build the datasets. Each data

format has thus its own parser. It is also possible to implement the parsing logic directly in the dataset, however this approach would be against the principle of separation of concern.

The dataset module further contains `idx` package for parsing IDX files and representing them in the form of the appropriate models. The IDX data format was described in more detail in Section 3.6. IDX is the format used by the authors of the MNIST dataset [23]. The `LabeledMnistParser` is specialized in parsing the MNIST IDX data and contains validations as well as correct configuration for the IDX files.

The `fold` package further specializes in providing support for k-fold cross validation. It contains `FoldDatasetFactory`, which produces the respective training and validation datasets. These are built from any standard datasets implementing `LabeledDataset` interface. They consist of a collection of k embedded standard datasets, representing the k validation folds. Internally, both training and validation datasets use the same physical data used by the original dataset they were created from. This approach saves memory by avoiding duplication of data and also improves efficiency, because there is no need to copy the underlying data. The validation and training datasets, however, have customized iterators pointing to the correct patterns in the data. The iterator in the training dataset iterates over the $k - 1$ datasets meant for the training. At the same time the validation dataset iterates over the one embedded dataset meant for validation.

3.7.2 Network module

The network module encapsulates application logic for the network as a group of layers. The actions this module is responsible for are the configuration of the network parameters, setup of the network layers, passing the configuration to layers and triggering forward and backward network runs. We will discuss these roles in more detail in the paragraphs below. We will start with the general description of the interface and follow with the descriptions of specialized implementations.

The `Network` constructor takes a single parameter - `NetworkConfiguration`. It represents the user input for the configuration as an object, so it is easy to work with. After the `Network` object is constructed, we need to call the `setup()` method, which does the heavy initialization. I decided to incorporate quite a lot of responsibilities into the `Network` objects. This allows for performance optimizations and reduces boiler plate code.

The network setup starts with the configuration and setup of layers. The layer configuration logic is actually delegated to the layers themselves, which allows for flexibility and layer polymorphism. The `Network` itself does not need to know what kind of layer it is dealing with. It just passes the configuration data, and the layer configures itself.

The next step is the allocation of memory. This is a specialty of `deepframe`, as for the contrary some learning frameworks actually delegate this to the layers (see Section 4.7). Instead, I am asking the layers how many weights they need, how many neurons they need, and then manage this global chunk of memory for them. This results in a better collocation of the memory, and allows to parallelize some tasks on a global network level. It allows me, for example, to apply the weight adjustments Δw in one single CUDA call for the whole network. The memory

allocation is also faster, as multiple malloc calls can be relatively expensive.

The next step is initialization of random weights and bias. This step also benefits from memory management at the network level, because when using GPU I can generate random numbers in one CUDA call for the whole network. The generator itself can take a seed from the user input, so that the computation can be repeated. This is very useful for debugging purposes. For the production runs, the generator by default generates the seed based on the output of the `/dev/urandom` device. It would be possible to use current timestamp, however this approach can cause problems on clusters with parallel runs - producing the same random sequences.

The last step in the network setup is to assign correct pointers to the layers. They need to point to the right place in the big chunk of globally managed network memory. This helps to keep the layers working on their dedicated memory partitions.

The network interface has two implementations out of the box. The first one is computing on the CPU, the second one on the GPU. They each call the relevant methods on the layers, so the computation takes place consistently on the same device. Most of the logic is already implemented in the `Network` class. A special behavior is required for `GpuNetwork` when allocating memory on GPU device, when generating random weights and bias, and when copying memory from device to host and back.

The copying of the memory content between the host and the device is an expensive operation. I implemented lazy copying to only perform the operation when it is really needed. This is implemented by keeping the state variables on `GpuNetwork` tracking whether the given device memory is in sync with the host memory (and vice versa). Once a network method requiring this memory is called, the state variable is consulted first. The copying takes place only in the case the device/host memory has changed since the last copy.

3.7.3 Layer module

The layer module is a submodule of the Network module. All layers must inherit from the `Layer` class. It defines generic layer operations, which allows for polymorphism. Layer logic is highly dependent on the layer type. For this reason, the `Layer` class does not contain much business logic itself. It mainly consists of property getters and setters.

Several layer implementations come out of the box with `deepframe` application. These usually implement:

- parsing of the custom configuration,
- deriving and computing their parameters from configuration,
- allocating specialized buffers if they need them,
- forward run on the CPU,
- forward run on the GPU,
- backward run on the CPU,

- backward run on the GPU.

A notable feature of the layers is a specialized method for backpropagating the last layer. This method is called instead of the standard backpropagation in the case we are dealing with the output layer. The motivation behind this is the fact that the last layer computes its differentials Δw from the expected output (for better understanding, see Section 2.3.4). Therefore it saves us the trouble of checking whether we are in the last layer in the backpropagation step for every hidden layer. This results into a slightly improved performance, and better readability and method scope.

Additional layer types can be added by putting relevant classes into the package folder for the layers (`src/main/net/layers`). These classes need to inherit from `Layer` and implement its virtual methods.

```
1 virtual void forwardCpu() = 0;
2 virtual void forwardGpu() = 0;
3
4 virtual void backwardCpu() = 0;
5 virtual void backwardGpu() = 0;
6
7 virtual void backwardLastCpu(data_t *expectedOutput) = 0;
8 virtual void backwardLastGpu(data_t *expectedOutput) = 0;
```

Snippet 3.11: The virtual methods of Layer interface.

If the newly implemented layers contain some specialized initialization logic, this goes into `setup()` method, which can be overridden for such purposes. For examples on how to implement new layers consult the source code for existing layers. Once the layer implementation is ready, it needs to be registered in the application. For example, if the class name of the new layer is `MyLayer`, and we want to choose this layer type by typing `My` into the configuration, we can register it with the following statement at the end of `MyLayer.cpp` file.

```
1 static LayerRegister<MyLayer> reg("My");
```

Snippet 3.12: The layer registration code.

Note that placing the registration code is an important design feature, as it allows the layer to be self-contained. This way the layer is not polluting the application code outside its own header and `cpp` file. To remove the layer, we can simply delete these two files.

3.7.4 Error computing module

The error computing module contains different implementations for computing network error. The base interface declares a single method with the signature below.

```
1 virtual data_t compute(Network *net, data_t *expectedOutput) = 0;
```

Snippet 3.13: The virtual method of ErrorComputer interface.

It takes the network itself as the first parameter, and the expected output as the second. It defines a contract to compute the network error from these inputs. In fact, the network output would be sufficient parameter instead of the whole network object. However, in that case we would also need to pass the number of outputs. Having the whole network object provides more flexibility for possible future enhancements, and allows for fewer parameters in the method signature.

In the testing part of this work, I relied mainly on the Mean Square Error to assess the accuracy of tested network models (see Chapter 4). It is implemented in the `MseErrorComputer` class.

Additional implementations for computing error can be added by inheriting from the `ErrorComputer` base class and implementing its virtual method (see snippet 3.13). To start using it in the learning process, one needs to configure the learner to use it via the provided setter `setErrorComputer()`. For example consult the `main.cpp` file, where it is setup for `BackpropagationLearner`.

3.7.5 Training module

The training module has the responsibility of training the network by updating their learning parameters towards the state in which they can perform the desired task. The training process is dependent on the layer type used, so the parameter updates are actually taking place in the layer implementation, outside of the training module. However, the training module manages this process.

The training algorithm implemented and used for supervised learning in all layer types supported out of the box is the backpropagation algorithm, and is implemented in class `BackpropagationLearner`. It implements the gradient descend method described in more detail in Section 2.2.2. It handles initializing the network with input data, performing the forward and backward runs, which within the specific layer implementations update the learning parameters. Backpropagation also checks the training and validation error, and terminates the training when target error rates or epoch limits are achieved.

The DBN network is composed of RBM layers, which require a special pretraining phase. This is also implemented in the training module, more specifically in class `NetworkPretrainer`. It manages the whole pretraining process, including initializing the network input and pretraining layers sequentially. More details on the implementation of pretraining can be found in Section 3.10.

The `TrainingResult` class, also located in the training module, is a data transfer object. Its role is to carry information about the results of the training process. Its attributes include the number of epochs run during training, and achieved training and validation errors.

3.7.6 Logging module

The purpose of the logging module is to provide a flexible API to easily print messages informing about the application state. Each logged message contains a timestamp, which is produced by the logging framework `gtest`. The framework also provides flexibility in configuring output at one place. By default, the application is logging to standard output. This can be changed in the logger factory, and it is possible to easily redirect all the logged messages into a specific file. An

important functionality of the logger is the ability to choose a logging level. This can be used to easily suppress logging of the debugging messages for the user, while allowing the developer to enable the debugging messages by a command-line option. Another advantage is to decouple the logic of concatenating strings, and to delegate the building of strings to the logger. The logger can then optimize and decide to truncate the message if it is configured not to log it (for example because of low severity level).

The `log` module contains a factory for the logger. This is the place where the logger is configured and built. To change the log destination (both files and consoles may be used), just edit the appropriate setter in the factory. The following snippet demonstrates how to build a `log4cpp` logger with `INFO` priority level and logging to console:

```
1  log4cpp::Category* LoggerFactory::create() {
2
3      log4cpp::Category *logger = &log4cpp::Category::getRoot();
4      log4cpp::Appender *p_appender = new
           log4cpp::OstreamAppender("console", &std::cout);
5      log4cpp::PatternLayout *layout = new log4cpp::PatternLayout();
6      layout->setConversionPattern("%d{%Y-%m-%d %H:%M:%S} [%p] %c: %m%n");
7      p_appender->setLayout(layout);
8
9      logger->setPriority(log4cpp::Priority::INFO);
10     logger->addAppender(p_appender);
11     return logger;
12 }
```

Snippet 3.14: Factory building a logger instance of INFO priority logging to console (stdout).

The produced logger is meant to be a singleton across the whole application. The only proper way to obtain it is through calling the static `getLogger()` method on the factory.This method always returns the same instance:

```
1  log4cpp::Category *LoggerFactory::getLogger() {
2      if (!isCreated) {
3          isCreated = true;
4          instance = create();
5      }
6      return instance;
7  }
```

Snippet 3.15: Definition of method for obtaining logger.

To obtain the logger in a less verbose manner, the following helper macro is defined in `LoggerFactory.h`:

```
1  #define LOG() Log::LoggerFactory::getLogger()
```

Snippet 3.16: Static getter for obtaining logger singleton instance.

To log a message all that is needed is to include the `LoggerFactory.h`, `Category.hh` and invoke the macro. The `log4cpp` library handles the string building using given parameters according to the specified format.

```
1 #include "../log/LoggerFactory.h"
2 #include "log4cpp/Category.hh"
3
4 LOG()->debug("Message logging string parameter '%s' and decimal
    parameter '%d'.", stringParam, floatParam);
```

Snippet 3.17: Example code logging a sample message.

Note that this approach builds the string only if it is actually appended to the log. Thus if the logging level is above the level specified for the message, there is little overhead in invoking the logger.

3.8 Multilayer perceptron network

The first implemented model is Multilayer perceptron. It consists of a variable number of mutually fully inter-connected layers of variable size. The design characteristics of this network are very simple. The architecture can vary based on the number of layers and the number of neurons in each layer. Because of such a simple design, I decided to configure the MLP architecture via a command-line option `--lconf`. This allows for a very flexible reconfiguration of the network's architecture. For a complete list of configuration options, please read Section 3.5.

All the layers of an MLP have the same functionality, inter-connections, and differ only in their size. For this reason, I decided to implement their logic in the `Network` object. The forward run of the network simply performed a standard matrix multiplication. The backward run in version `v0.1` was handled in the `BackpropagationLearner`. This would have allowed for using a different learner implementation, for example genetic algorithms, or exhaustive search, simply by swapping the learner implementation. Because of the architectural simplicity, I also implemented custom CUDA kernels for testing and training the network. The argument in my mind was to optimize the kernels for the task. You can see the performance achieved in version `v0.1` in Figure 3.5.

After I compared my performance results with existing frameworks (see Section 4.7), I was not satisfied with the results. Further investigation and testing revealed, that my custom CUDA kernels actually performed worse than standard linear algebra libraries. Those are apparently optimized at the assembly code level, and heavily tested by their maintainers. I researched and analyzed the algorithms searching for places where standard linear algebra operations could be leveraged, and implemented these using the `cuBLAS` library provided by `NVIDIA`. This approach led to a significant improvement in the performance, compare Figures 3.5 and 3.6.

A careful reader may notice a slight jump in the performance around 1500 hidden neurons. This is caused by the way `CUDA` operates. The number of parallel threads running a kernel is usually a multiple of 32 (depends on GPU model). If the dimension of data is a multiple of 32, it can be efficiently aligned in the

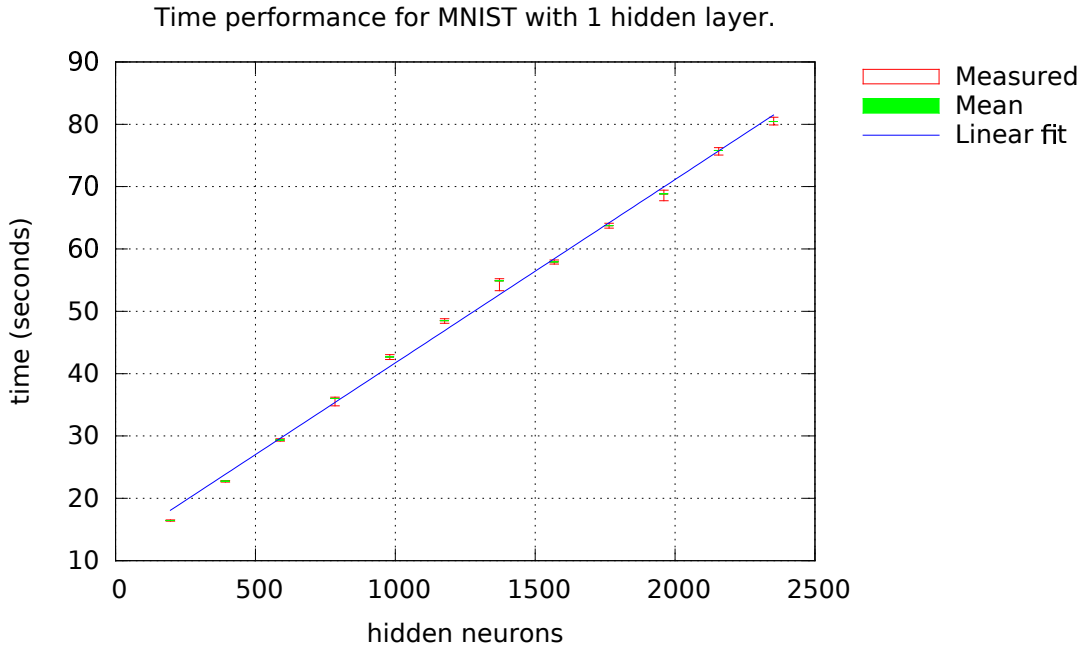


Figure 3.5: Time results of the MNIST test run with different number of neurons in one hidden layer. Values are averaged out of 20 runs. Test was run with version v0.1 of deepframe on the test machine using GPU for parallel computations.

memory so that all threads have data to work with. In such case there is no need to check the data boundaries, and performance is improved noticeably. This phenomenon is replicated and explained in the research of Barrachina [3].

Before I started implementing the CNN model, I realized that the layers cannot be generally considered as having the same architecture as in simple MLP case. Therefore I abstracted out the logic of the forward run into a new `Layer` object. The backpropagation also differs by layer type, so I moved that logic into the `Layer`'s responsibilities. This took away the possibilities to switch learner implementations, however needed to be done as a preparation for implementing CNN and DBN models.

Then I implemented the only layer used in MLP - the `FullyConnectedLayer`. It contains a single vector of neurons. The neurons in the neighboring layers are fully interconnected with each other. Let us consider a layer with a set of m input neurons I, we will denote the set of n output neurons O. The weights between I and O form a matrix W of dimensions $m \times n$. The forward propagation in this model can be implemented using the matrix multiplication operation:

$$I_m \cdot W_{m \times n} = O_n \tag{3.1}$$

The CPU implementation of `FullyConnectedLayer` uses simple `for` loops I implemented to perform the matrix multiplication. The parallel implementation uses the `gemm` call from cuBLAS library, which is highly optimized. The CPU performance could be very likely improved by using a third-party library for linear algebra operations. The most popular are MKL, and ATLAS. MKL is from Intel and is optimized for Intel hardware. ATLAS is opensourced under

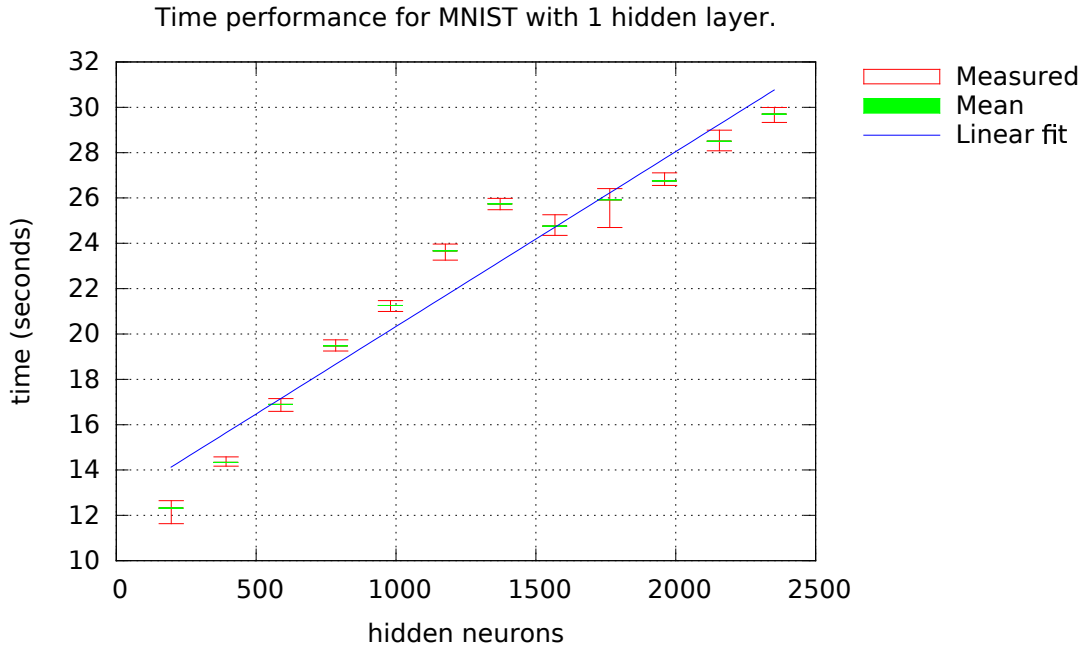


Figure 3.6: Time results of the MNIST test run with different number of neurons in one hidden layer. Values are averaged out of 20 runs. Test was run with version v0.2 of deepframe on the test machine using GPU for parallel computations.

the BSD license, and is dynamically optimized for the given hardware. I did not experiment with these libraries because of time constraints and the focus of this work on GPU implementation.

The backward run in the fully connected layer is again implemented using a custom code on CPU. For the parallel implementation I used custom CUDA kernels. I have been getting good results with these (see Section 4.7), so I never tried to replace them with cuBLAS alternatives.

3.9 Convolutional neural network

The model of convolutional neural network introduces two new layer types on top of fully connected perceptron layer: Convolutional and Subsampling layer. Their architectural design is described in Section 2.3.1. In the text below I will focus on the implementational details.

The GPU implementation of the forward run in the Convolutional layer uses a matrix multiplication call provided by cuBLAS (the `gemm` call). However, to convert the forward run into a matrix multiplication problem, it is required to reorganize the data in the memory. This is done using cuBLAS call `im2col`, which rearranges the rectangular convolution kernels into matrix columns (see Figure 3.7). Let us call such a matrix to be a column buffer. Afterwards we can use a standard `gemm` call to multiply the column buffer with the weights vector to produce layer outputs. For optimization reasons, both `im2col` and `gemm` calls can be run in parallel for all feature maps in the given layer.

The CPU implementation of the forward run in the Convolutional layer uses

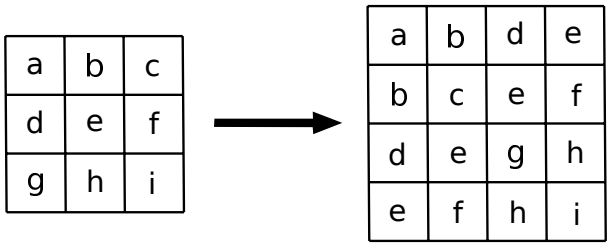


Figure 3.7: Illustration of converting a 3×3 feature map into a 4×4 matrix with kernel images rearranged into columns using 2×2 convolution.

a set of nested for loops. Again, using specialized libraries might prove to provide better performance. However, I did not test this, and instead focused on the parallel implementation on GPU. On the other hand, having custom loops saves us the trouble of rearranging the memory (as in Figure 3.7), so the performance penalty for not using optimized BLAS calls is likely to be negligible.

The backpropagation of the Convolutional layer on GPU starts with converting the convolution kernels for input features into the column buffer (see Figure 3.7). The produced column buffer matrix is then multiplied by local gradients for the layer outputs ($\partial E/\partial y_n$). The resulting matrix then consists of the weight differentials Δw. To backpropagate the differentials down to the input we can multiply the local gradients for output with the weights, and put the result into the column buffer. The data then needs to be rearranged back to image data by calling the `col2im` operation (reverse process of the one illustrated in Figure 3.7). The produced image data represent the total differentials for the layer input. The backpropagation process can then continue in the layer below. For explanation of the theory behind these computations please consult Section 2.3.1.

A notable optimization I implemented was leveraging the coefficients for applying learning rate directly in the matrix multiplication calls. This saves one vector multiplication for every layer at each backpropagation step. This optimization is used in the Convolutional layer as well as the Subsampling layer. The backpropagation of the Convolutional layer on the CPU again uses custom loops for computing the differentials.

The forward run in the Subsampling layer on the GPU is implemented via a custom-coded CUDA kernel. Subsampling operates by picking the neuron with maximum potential and passing that value forward. In this process the index of the neuron is stored in a cache, so it can be used in the backpropagation step. The forward run on CPU obeys the same logic, however in serial nested for loops.

Backpropagation in the Subsampling layer on GPU is also implemented by a custom CUDA kernel. The cached index of the activated neuron is used, and the total differential is passed down from the output neuron, only to this one input neuron. The CPU implementation performs the same logic again in serial nested for loops.

Please note, that the implementation of the Subsampling layer is different from the one explained in the theoretical part in Section 2.3.3. The implementation is not using any learning parameters. The subsampling simply chooses the maximum potential and passed that to the upper layer. In the backpropagation step, the total differential is simply passed from the output to the activated input neuron. Also, the activation function is not used. By not implementing these

extra features I was able to cut down on performance cost. The reason why I did not include these features is that it led to a better classification accuracy without slowing down the learning.

3.10 Deep belief network

The Deep Belief Network introduces a new layer type, the RBM layer. It is formally described in Section 2.4.3. Its implementation in `deepframe` application is located in the `net/layers` package, in the `RbmLayer` class. Its configuration options are described in Section 3.5.1.

The RBM layer is the only layer in `deepframe` application, which is provided out of the box and does not support CPU implementation. The reason for this is the very poor performace of CPU, caused by performance intensive sampling in contrastive divergence algorithm in the pretraining phase. This implies extremely long running time for single-threaded implementation. Instead, the GPU implementation is highly optimized taking advantage of parallel computing.

The forward run in RBM layer computes the inner product multiplying the visible neurons with their respective weights. This produces the potentials for hidden neurons, and is implemented using the `gemm` call from cuBLAS library. The potentials are then normalized by the sigmoid function. Note that the implementation of the forward run is in fact the same as in the MLP network. The backward run in RBM layer does not do anything, because all the learning takes place prior to supervised learning phase – during pretraining.

The RBM layers in the Deep Belief Network introduce the pretraining process, which is not used in any of the previously described networks. The pretraining is launched just before the supervised training, and is run for a predefined number of epochs. As the help explains, the number of pretraining epochs is configured by the option `-n` or its longer alternative `--pretrain`. The default is zero epochs, which effectively disables pretraining.

The class responsible for the pretraining is named `NetworkPretrainer` and is located in the `train` package. Its main logic is encapsulated in the `pretrain()` method.

```
1  void NetworkPretrainer::pretrain(LabeledDataset *trainingSet) {
2
3      LOG()->info("Started pretraining in %d epochs...", epochs);
4
5      // input is never trainable -> start from 1
6      int noLayers = netConf->getLayers();
7      for (int i = 1; i<noLayers; i++) {
8
9          Layer* layer = net->getLayer(i);
10         if (layer->isPretrainable()) {
11
12             for (int e = 0; e<epochs; e++) {
13
14                 LOG()->info("Pretraining layer %d in epoch %d.", i, e);
15
16                 // iterate over all training patterns and pretrain them
```

```
17                 trainingSet->reset();
18                 while (trainingSet->hasNext()) {
19                     data_t* input = trainingSet->next();
20 
21                     // do a forward run till the layer we are pretraining
22                     net->setInput(input);
23                     for (int j = 1; j<i; j++) {
24                         Layer* l = net->getLayer(j);
25                         if (useGpu) {
26                             l->forwardGpu();
27                         } else {
28                             l->forwardCpu();
29                         }
30                     }
31 
32                     // pretrain the layer
33                     if (useGpu) {
34                         layer->pretrainGpu();
35                     } else {
36                         layer->pretrainCpu();
37                     }
38                 }
39             }
40         }
41     }
42 
43     LOG()->info("Finished pretraining.");
44 }
```

Snippet 3.18: Implementetition of pretraining process for the Deep Belief Network.

As we can see in Code Snippet 3.18, the pretrainer runs the `pretrain()` method on the layer. The pretraining capability of the layer is determined by calling `layer->isPretrainable()`, which returns `false` by default. To implement pretraining, this method can be overridden in the layer implementation to return `true`. Also, the `pretrain()` method obviously needs to be implemented in such case, otherwise no learning would take place in the pretraining phase.

The pretrainer also handles assigning input from the training dataset into the visible neurons of the first layer. Then it makes sure all the layers performed the forward pass propagating the signal potentials upward reaching the visible neurons of the layer being pretrained. This ensures pretraining is performed with the correct input signals.

The RBM layer then implements the pretraining by performing the Gibbs sampling. This is done by a predefined number of Gibbs sampling steps. The implementation of sampling activations in the hidden neurons from the activations in the visible neurons can be seen in Code Snippet 3.19.

```
1 void RbmLayer::sample_vh_gpu() {
2 
```

```
3     propagateForwardGpu(sInputs, shPotentials, sOutputs);
4
5     k_generateUniform(*curandGen, randomData, outputsCount);
6     k_uniformToCoinFlip(sOutputs, randomData, outputsCount);
7 }
```

Snippet 3.19: Implementetition of pretraining process for the RBM layer.

The sampling starts by forward propagation of the signal potentials, which essentially represents the probabilities of neuron activations. The activations are then sampled from these probabilities. It is implemented by generating an array of random numbers from uniform probability distribution. Then a custom CUDA kernel is used to determine the activations by comparing the random data with the probabilities of activations (see Code Snippet 3.20). The sampling of activations in the visible neurons from the activations in the hidden neurons is implemented analogously.

```
1 __global__
2 void uniformToCoinFlip(data_t *p, data_t *dArray, int elements) {
3     int i = blockDim.x * blockIdx.x + threadIdx.x;
4     if (i < elements) {
5         p[i] = (dArray[i] < p[i]) ? 1 : 0;
6     }
7 }
```

Snippet 3.20: Implementetition of a custom CUDA kernel sampling the activations from their probabilitues and random data generated under uniform probability distribution.

The implementation of the pretraining for RBM layer can be seen in Code Snippet 3.21. It starts with doing a forward run on the layer, which computes the probabilities of activations p_h.

In the next step the sampling chain is restarted by sampling the hidden activation states in case PCD is enabled. Then the visible activation states are sampled from hidden states, and these are resampled back to hidden states. This is the Gibbs sampling step, and is performed for a preconfigured number of times.

Pretraining continues by computing the positive gradient matrix ϕ^+ and subtracting the negative gradient matrix ϕ^-, which produces the weight differentials. Differentials for biases are computed by `axpy` calls from cuBLAS library, for both the visible and hidden neurons separately. Pretraining is finished by applying all the computed parameter updates (weights and biases).

```
1 void RbmLayer::pretrainGpu() {
2
3     data_t *inputs = previousLayer->getOutputs();
4
5     // single forward run will compute original results
6     // needed to compute the differentials
7     forwardGpu();
8
```

```
 9      // Reset the parameters sampled from previous training
10      if (!conf.isPersistent || !samplesInitialized) {
11          samplesInitialized = true;
12          int memSize = inputSize * sizeof(data_t);
13          checkCudaErrors(cudaMemcpy(sInputs, inputs, memSize,
                cudaMemcpyDeviceToDevice));
14          sample_vh_gpu();
15      }
16
17      // perform CD-k
18      gibbs_hvh(conf.gibbsSteps);
19
20      // COMPUTE THE DIFFERENTIALS
21
22      // First we will compute the matrix for sampled data,
23      // then for real data and subtract the sampled matrix.
24      k_gemm(cublasHandle, CblasNoTrans, CblasNoTrans,
25             outputsCount, inputSize, 1,
26             lr, sOutputs, sInputs, (data_t) 0., weightDiffs);
27
28      k_gemm(cublasHandle, CblasNoTrans, CblasNoTrans,
29             outputsCount, inputSize, 1,
30             lr, outputs, inputs, (data_t) -1., weightDiffs);
31
32      if (conf.useBias) {
33
34          data_t *vdiffs = weightDiffs + genuineWeightsCount;
35          data_t *hdiffs = vdiffs + inputSize;
36
37          // clear the bias diffs just before working with them
38          checkCudaErrors(cudaMemset(vdiffs, 0, (inputSize +
                outputsCount) * sizeof(data_t)));
39
40          // compute bias for visible neurons
41          k_axpy(cublasHandle, inputSize, (data_t) lr, inputs, 1, vdiffs,
                1);
42          k_axpy(cublasHandle, inputSize, (data_t) -lr, sInputs, 1,
                vdiffs, 1);
43
44          // compute bias for hidden neurons
45          k_axpy(cublasHandle, outputsCount, (data_t) lr, outputs, 1,
                hdiffs, 1);
46          k_axpy(cublasHandle, outputsCount, (data_t) -lr, sOutputs, 1,
                hdiffs, 1);
47      }
48
49      // adjust RBM parameters according to computed diffs
50      k_sumVectors(weights, weightDiffs, weightsCount);
51  }
```

Snippet 3.21: Implementetition of pretraining process for the RBM layer.

It is important to note, that all RBM layers need to be pretrained in a sequential order from the first hidden layer to the last one. This process cannot be parallelized well, because the input signals coming from the preceding layer must already be coming from the modeled probability distribution, and hence must be coming from an already well-pretrained layer.

4. Testing

In this chapter I will test all the implemented models of neural networks. In Section 4.1, I will specify the requirements on the test cases, test scenarios and testing workflows. I will describe the hardware and software I used for testing in Section 4.2. In Section 4.3, I will describe the methodology and specify the properties of the implementation I want to investigate and assess. In the following 3 sections, I will summarize the results of each network implementation for different sets of problems. In Section 4.7, I will compare my implementation with an existing alternative solution tackling the same problems.

4.1 Testing requirements

First, let me specify important requirements that I want to follow when designing performance tests for the implemented system. Each item in the following list of requirements is accompanied with an explanation of the motivation behind the requirement.

- easily repeatable tests

  Tests will be repeated multiple times with different input pseudorandom sequences to ensure representative results. It would be time-demanding to launch each test case manually. To prevent this, it should be possible to launch a batch of tests via a single command.

- automated process from testing to reporting

  The whole process from launching the tests, through processing the results, to presenting the results should be automated. This will make it possible to easily repeat the test scenarios, which helps to identify regression bugs. It will also eliminate human error and inconsistencies when dealing with many test cases.

- testing scripts must be tied to a specific application version

  If the application configuration or API changes, the testing scripts must be adjusted so they are compatible. To ensure this, the version of the code running the tests must be tied to a specific version of the application. It will also allow to repeat the tests in future, even though application API changed. Note that there is no need to version the test results, only the launch scripts. Test results can be generated from these launch scripts at any point in time.

- future reuse of results

  The test results should contain all potentially useful data, even if they are not used in the presentation of results at the moment. It will allow to compare different tested properties and additionally include them in the presentation of the results without the need to rerun the tests.

By obeying these requirements I will achieve stable test cases, that can be repeatable and verifiable in future. It will also allow me to easily launch long-running batch of tests, and work on the implementation in the meantime.

4.2 Testing environment

While working on the implementation, I was using two machines. Both were standard desktop machines. The first one was used primarily for the development, let us call it the development machine. It had desktop manager along with other active services installed that were used for the development. The testing performed on this machine occasionally showed spikes and anomalies in the test results caused by other services running simultaneously.

The second machine was on the other hand used primarily for testing purposes, let us call it the testing machine. This machine had no desktop manager and no services other than the required dependencies for the implemented `deepframe` application. This allowed to test as independently of other running services as possible and also allowed to perform long-running tests, while still having an extra machine for development purposes.

4.2.1 Hardware specification

The development machine hardware specification follows below. A detailed specification of the GPU installed in the development machine can be found in Attachment 3.

- CPU
  - Intel Core i7-950 3.06GHz
  - 4 cores, 8 threads
  - 64bit instruction set
  - 8MB cache
- RAM
  - PATRIOT 12GB KIT DDR3 2000MHz Viper Xtreme Series
  - triple channel (3 x 4GB)
  - CL9-11-9-27
  - actual frequency 1066Mhz
- HDD
  - Data disk: WESTERN DIGITAL Caviar Green 2000GB 64MB
  - System disk: SSD OCZ Vertex 3 Series 60GB
- GPU
  - GIGABYTE N56GOC-1GI
  - NVIDIA GeForce GTX 560 GPU

  - 1GB GDDR5, 256-bit memory interface
  - 336 CUDA cores

The testing machine has a faster memory with higher capacity. The processor has the same number of cores, but being i5, it does not support hyper-threading. This should give the development machine an advantage when computing on the CPU only. However the testing machine has a GPU with a significantly higher memory capacity and a higher number of CUDA cores. This should support faster computations on GPU. A detailed specification of the GPU can be found in Attachment 3. A brief hardware specification follows:

- CPU
  - Intel Core i5-4570S 2.90 GHz
  - 4 cores, 4 threads
  - 64bit instruction set
  - 6MB cache
- RAM
  - 16GB DDR3
- HDD
  - SSD ADATA SX910 XPG 256GB
- GPU
  - NVIDIA GeForce GTX 980 GPU
  - 4GB GDDR5, 256-bit memory interface
  - 2048 CUDA cores

4.2.2 Software specification

The development machine is running under the Operating System (OS) Linux Debian Jessie. The compiler used to compile `deepframe` was `gcc 4.8.3`. Further this machine was running the Gnome Desktop Manager, Apache server, and many other services at all times, even during testing. Therefore the test results obtained on this machine may suffer from occasional inconsistencies.

The testing machine is running under the OS Linux Gentoo 2.2. This machine was used primarly for testing, and was therefore not running any extra services, in order to provide as independent tests as possible. The machine does not have any graphical interface installed, and all the tests were run from the console.

4.3 Testing methodology

I will test the implementation of each network model on an appropriate problem. The aim of testing is not to get the best classification results, but to assess the performance of my implementation and compare it to the performance of third party implementations. The testing conforms to all the requirements stated in Section 4.1.

The requirement of an easy launch of the test scenarios will be tackled by means of bash scripts. They will allow running a batch of tests by issuing a single command.

The second requirement of an automated process will be also solved by bash scripts. The presentation of the results will be done using graphs. These will also be generated by scripts using the gnuplot. Gnuplot is a portable command-line driven graphing utility for Linux, OS/2, MS Windows, OSX, VMS, and many other platforms. Gnuplot is able to generate graphs in vector graphic, which I will leverage in this thesis. The generated graphs are saved in encapsulated postscript files (eps).

The third requirement of having tests and application source tied together in the version control will be satisfied by using git for versioning the test scripts along with the application. You can find the scripts in the `tools` folder in the root of the application that has been included in the attached disk 1. The `tools` folder contains the scripts for launching a batch of tests and generating the graphs with the results. Additional script templates for running a single test of learning and testing of one single network on one dataset, are included in the `examples` folder. These scripts can be used as examples of how to configure the application to solve the provided sample problems.

The last requirement of possible future test reuse will be satisfied by keeping the output produced by the application. As discussed in Section 3.7.6, all the log messages are printed on the standard output by default. These logs should contain all the important data about the application run. The specific data we will be interested in to present the testing results will be parsed out from these logs. As we have the logs, we can later decide again to parse different values, if we need them. This satisfies the requirement.

Our motivation behind the testing is to asses the performance of my implementation. It is quite problematic to do an unbiased assessment, given the variety of available hardware architectures. To get around this, I will be comparing all the implementations on the same hardware, which will be the testing machine. Obviously, different implementations might be taking advantage of certain architectural features of a specific hardware. Therefore, the fact that an implementation A is performing better than implementation B on hardware X, does not imply that it will perform better on hardware Y as well. However, since I have access to a high-end GPU GTX 980, and an older less-performing GPU GTX 560, testing on both should yield a good comparison across hardware with different performance.

Before we start the testing, it is important to specify the key performance indicators for each implemented model. The most important indicator is the time it took for the application to run, which I will measure in seconds. Besides time constraints, it is also important to look at the RAM memory consumed by the

process. When choosing hardware for the testing, we cannot let the application request to allocate more memory than the capacity we have available. Once we are allocating more, the memory will start to swap into a much slower persistent storage (HDD), which will heavily impact the performance. I will be measuring the memory consumption in kilobytes.

Note, that the GPU has its own memory for parallel operations that is not included in the RAM memory metrics. This is intentional, because parallel calculations proceed in small units called warps, and each warp has a memory limit which depends on hardware specification. In case we were using too much memory in a warp and exceed the limit, the process would just crash. Furthermore, there is a limit for the number of warps that can be run in parallel. If we were using too many warps, then only a limited portion of them would run in parallel, and the remaining ones would have to wait for the next cycle.

Therefore the usage of memory within a warp is restricted by the design. Using more warps to redistribute memory will impact the time performance, which will be visible in the time reports. For these reasons it is sufficient to track time, and for our purposes not necessary to track GPU memory usage.

The last interesting performance indicator of a model is the Mean Square Error that was obtained for the solution of a given problem. This will be used to assess the properties of the implemented model rather than the achieved hardware performance.

We explained the metrics we want to be looking at to understand the performance. To measure them, I chose to use the standard Linux utility `time`. It supports tracking of all the indicators we need. To summarize, I decided to use the following as performance indicators:

- run time
  - the total number of CPU-seconds used directly by the process (in user mode).
  - this also includes the time when CPU was waiting on GPU to finish its parallel computations
  - will be measured using the `U` specifier of `time`
- RAM memory
  - the maximum resident set size of the process during its lifetime
  - note that we are only interested in the maximum, since we want to prevent any swapping when choosing sufficient hardware to solve certain problem
  - will be measured using the `M` specifier of `time`
- Mean Square Error
  - a metric used to asses the variations of actual results learnt from the provided dataset
  - defined in Equation 2.7
  - will be measured by a direct computation using `deepframe`

In the default setting, `deepframe` has a certain error threshold. Once it is achieved, the learning process is stopped, and the network model is considered learnt at this point. For our testing purposes this scenario is not desirable. Instead we want each test run to perform exactly the same number of training epochs. This will keep the performance indicators independent of the random initialization of network parameters. Note that this independence is made on the assumption, that identical operations performed with different numbers always result in only negligible performance differences. To account for the performance differences caused by the random initialization of variables (along with similar other aspects), I will run each test multiple times and average out the results.

4.4 Multilayer Perceptron

The first tested model is the Multilayer Perceptron. As already explained in previous chapters, it is a fully-connected feed-forward network build solely from perceptrons - neurons formally described in Section 2.1. The fully-connectiveness implies a big number of network weights. For this reason the model is quite robust for big numbers of hidden neurons. The first two problems to be tested are quite small in terms of network size. This is intentional. It allows us to run a lot of tests quickly, while still being able to show the linear trends of increased computational requirements with increased network size. We will start with the XOR operator. It will be followed by the problem of summing two 4-bit numbers. For both of these models quite small networks are satisfactory, as will be seen from the test results. We will focus on showing the complexity of the software implementation and its scalability on these 2 problems. The last tested problem of recognizing handwritten digits will require a relatively robust network. This model should advertise the advantage of using parallel processing on GPU for the computation.

4.4.1 Exclusive OR operator

The exclusive OR operator (XOR) is considered as a "Hello World" application in the field of artificial intelligence. It is a trivial problem, however more complicated than the other binary logical operators, because of its linear inseparability.

input x_1	**input** x_2	**output** y
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.1: A table with all possible inputs and outputs for XOR operator.

The XOR problem has 2 binary inputs, and a single binary output. The binary values represent *true* and *false* boolean values. All the possible inputs with their respective outcomes are represented in Table 4.1. Since there are only 4 possible cases, all of them will be used for training in the experiment. Otherwise we would suffer from lack of information in the training dataset. The same dataset will be used for testing the network's performance, too.

To perform the test experiment, we need the dataset to learn from. Given the problem has only 4 possible inputs, the dataset was created manually and is located at `resources/xor-labels.dat`. The test cases can be run by executing the command `./tools/run-tests.sh tools/xor.conf.sh`. The tests are configured to run for 1000 epochs in each of 100 iterations. Each iteration is initialized with random network weights and bias, which result in independent test runs. The configuration repeats these iterations with various different network architectures. The first set of tests continually increases the number of neurons in hidden layers. These tests should verify the scalability of an increasing layer size. The second set of tests keeps the number of neurons in hidden layers constant, and increases the number of hidden layers. These tests should verify the scalability of incorporating more layers into the network and making it deeper.

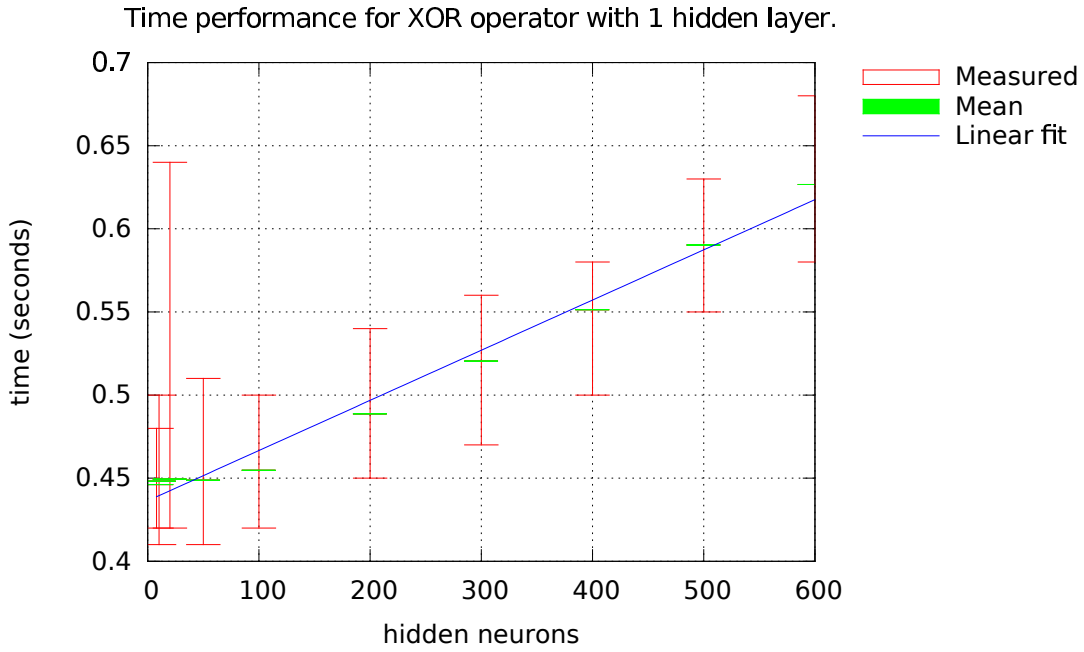


Figure 4.1: The time results of the MLP network solving XOR with a variable number of neurons in one hidden layer. The resulting values are averaged over 100 runs. The test was run on the development machine using GPU for parallel computations.

As depicted in Figure 4.1, the time it takes for the network to run increases with the number of hidden neurons. The important fact to notice is, that the time increases linearly. What happens if we are increasing the number of layers can be observed in Figure 4.2. Each hidden layer consists of 8 neurons. Again, in this case the time costs increase also linearly with the number of layers. This is an

Figure 4.2: The time results of the MLP network solving XOR with a constant number of neurons grouped into a variable number of hidden layers. The values are averaged over 100 runs. Test was run on the development machine using GPU for parallel computations.

important property of my implementation, as, e.g., an exponential growth would make this implementation inappropriate for bigger networks. The measured time intervals are depicted by red ranges, which also look stable.

The graph of mean square error in Figure 4.3 shows, that increasing the number of hidden neurons is not always producing better results. The trend is rather opposite. Having 50 hidden neurons causes big variations in the MSE values. After reaching 100 hidden neurons, the network's MSE stabilizes around 0.25. Increasing the number of hidden neurons produces MSE of 0.5, which is a terrible result. The reason behind this phenomenon is that having too many connections between neurons causes diminishing of gradient values. Therefore it takes much longer for the network to learn the patterns from the data.

The graphs with the results obtained on memory consumption can be found on the attached Disk 1. They are not providing any valuable information which would allow to make any conclusions about the memory requirements of the `deepframe` application. This is because the network modeling the XOR operator is too small to affect the memory significantly. To optimize storage requirements, each weight and potential of a neuron is represented by a single 4-byte number (or 8-byte number if compiled with double precision). An increase in memory consumption of several bytes caused by adding several neurons is thus negligible.

Additional graphs where the tests were run on the CPU can also be found on the attached Disk 1. Note, that the time performance is even better in this case. This implies that for small networks it is not beneficial to leverage GPU and its parallelization capabilities. The bottleneck here is caused by the copying of data

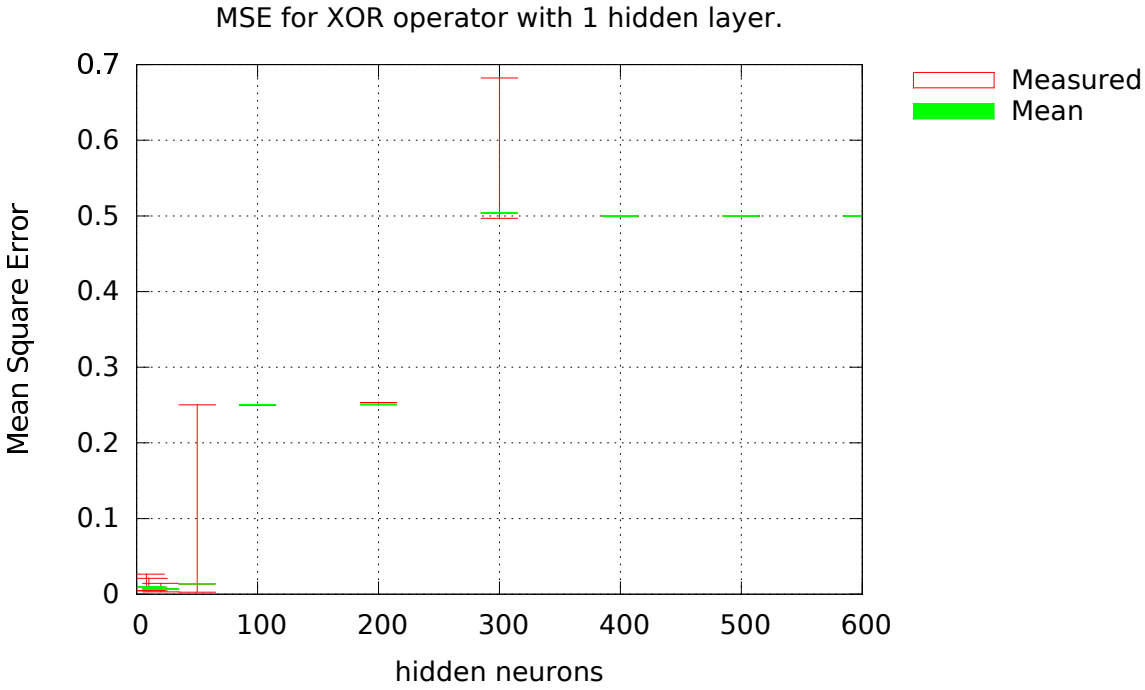


Figure 4.3: Mean Square Error results of the MLP network solving XOR with variable number of neurons in one hidden layer. Values are averaged over 100 runs. Test was run on the development machine using GPU for parallel computations.

from the RAM memory on the host, to the memory on the GPU device. The copying overhead is then not compensated enough by the parallel computations, since there are only few neurons in each layer.

Let us summarize the results of the experiment. In Figure 4.3 we can see low MSE values. This proves that the network is learning, which can be also verified directly by inspecting the test logs included on the attached disk. Furthermore, Figure 4.1 and Figure 4.2 show linear increase in performance costs with an increased network's size.

4.4.2 The sum of two four-bit numbers

The next problem I decided to use for the test of my implementation of MLP is the sum of two 4-bit numbers. Given two numbers, each represented by 4 bits, the task is to calculate their 5-bit sum. It is modeled by a network with 8 input neurons, and 5 output neurons representing the result. The testing was conducted using the same methodology as in the case of the XOR operator in Section 4.4.1.

The training dataset comprises all possible inputs, totaling to 256 (2^8) training patterns. The testing dataset contains again all the possible input patterns. The dataset was generated by a PHP script included on the attached disk. This script takes an argument that specifies the number of bits in each of the summed numbers. The generated dataset is printed out to the standard output. Code snippet **??** demonstrates how to generate such datasets and execute the tests.

```
$ php tools/create-bitsum-dataset.php 4 > resources/4bitsum-labels.dat
```

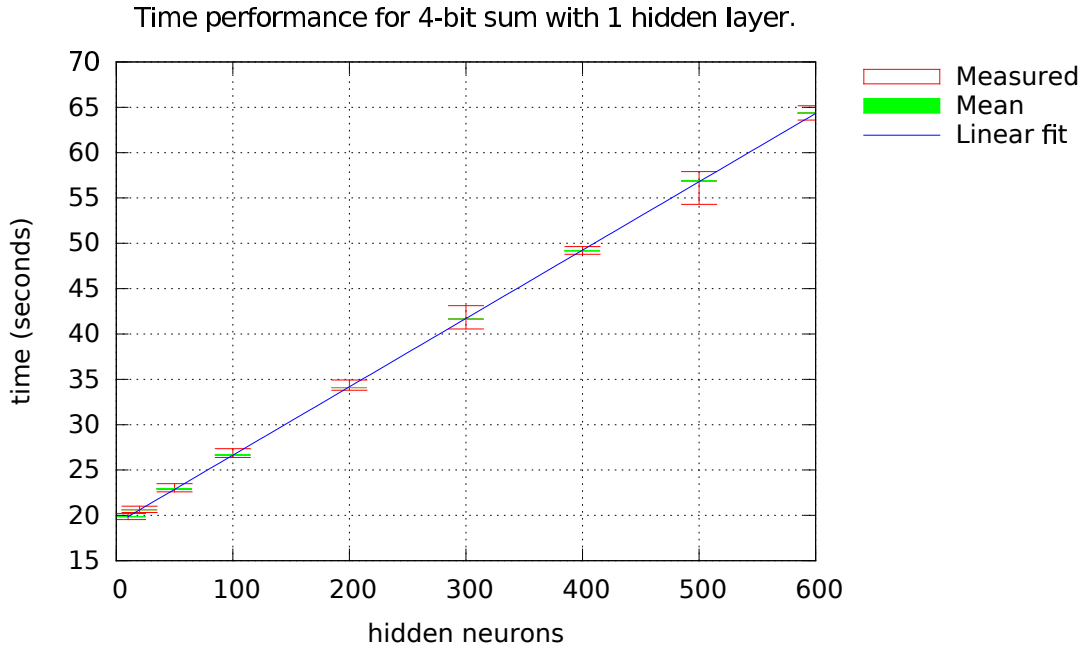


Figure 4.4: The time results of the MLP network solving the 4-bit sum problem with a variable number of neurons in one hidden layer. The values are averaged over 100 runs. The test was run on the testing machine using GPU for parallel computations with 64bit precision.

```
$ cat resources/4bitsum-test.dat | sed "s/\\s>.*$//" >
    resources/4bitsum-test.dat
$ ./tools/run-tests.sh tools/4bitsum.conf.sh
```

Snippet 4.1: Shell commands used to generate the training and testing datasets for 4-bit sum and to run the tests.

The obtained test results show the same patterns that were revealed in the case of the XOR operator. In Figure 4.4 we can see that the time increases linearly with the number of hidden neurons. The same applies also in the case of increasing the number of hidden layers when keeping the number of neurons in each layer constant.

Additional graphs where the tests were run on both the GPU and the CPU can be found on the attached Disk 1. Similarly as in the case of the XOR operator, the 4-bit sum problem requires a relatively small neural network. The test run for 600 hidden neurons takes for example around 28 seconds to complete on the CPU and 65 seconds on the GPU. Clarly, it is still not beneficial to leverage the GPU and its parallelization capabilities for this problem. The bottleneck is again the copying of the data from the RAM memory on the host, to the memory on the GPU device.

4.4.3 Recognition of handwritten digits

The last problem chosen to test the MLP implementation is aiming at the construction of a bigger neural network, which can take better advantage of parallel

computations on the GPU. A suitable problem is the recognition of handwritten digits. A well-known and well-tested MNIST dataset serving this very purpose is publicly available for research purposes [23]. It comprises of 60000 labeled patterns to be used for training, and 10000 labeled patterns for validation and testing. Each pattern represents an image of the size 28×28 pixels, each pixel colored with one of 256 shades of gray. The MNIST dataset is published in the IDX format. A detailed explanation describing the representation of the patterns is provided in Section 3.6.

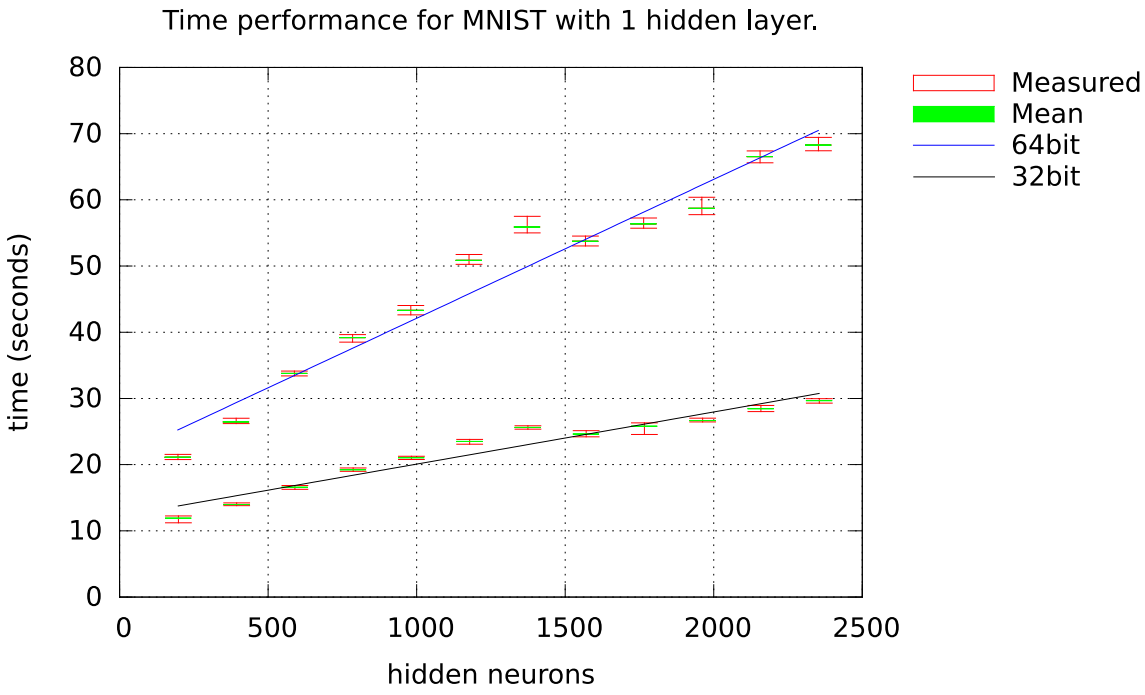


Figure 4.5: The time results obtained for the MLP network solving MNIST with a variable number of neurons in one hidden layer. The values are averaged over 20 runs. The test was run on the testing machine using GPU for parallel computations with 32bit and 64bit precision.

The MNIST datasets were downloaded from MNIST website and are included on the attached Disk 1, too. The testing was conducted with the same methodology as in the previous problems. To run the tests you can issue the shell command demonstrated in Code snippet 4.2.

```
$ ./tools/run-tests.sh tools/mnist-mlp.conf.sh
```

Snippet 4.2: Shell commands used to run the tests of MLP implementation on MNIST dataset.

The testing configuration can be customized by editing the `mnist.conf.sh` file. It is possible to adjust the architecture of tested networks, number of epochs, number of tests run for each network, run on GPU vs. CPU, learning rate, initialization intervals, and much more.

In Figure 4.5 you can see the time results for the test runs on MNIST dataset. The tested networks had one single hidden layer with a variable number of neurons (x-axis). The time performance for both single and double precision is displayed.

As expected, double precision is more performance intensive. Note that the increase in number of neurons causes only a linear increase in time performance. The network and learning parameters used were:

- 1 learning epoch,
- 20 repeated test runs for each network type,
- network bias enabled,
- weights and biased initialized randomly with normal distribution within interval (-0.3, 0.3),
- learning rate of 0.2.

In figure 4.6 you can see the time results for networks with multiple hidden layers (x-axis). Each hidden layer contained 784 neurons, the same number as the size of input. Otherwise the network and learning parameters were the same as in previous tests.

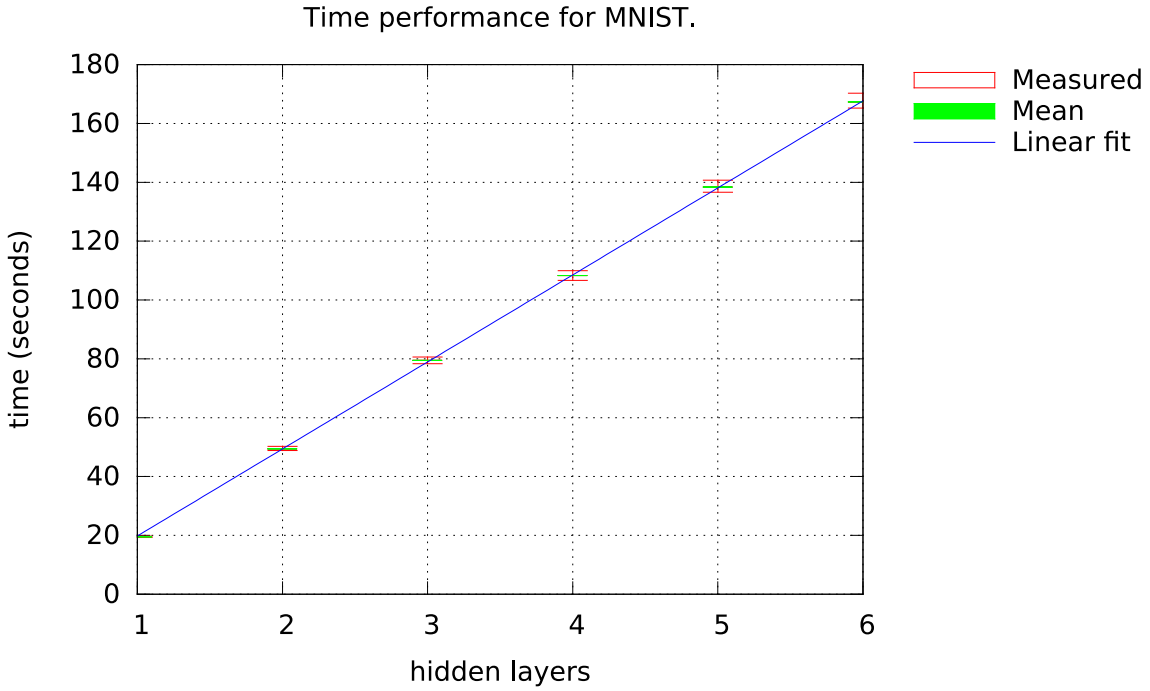


Figure 4.6: The time results of the MLP network solving MNIST with a constant number of neurons in variable number of hidden layers. The values are averaged over 20 runs. The test was run on the testing machine using GPU for parallel computations with 32bit precision.

Note that for 32 bit precision, having 2352 hidden neurons in one hidden layer takes around 28 seconds to compute. Having the same number of hidden neurons spread across 3 different layers, however, takes around 80 seconds. Such an increase occurs, because neurons in one layer can be easily parallelized. In case of multiple layers, before we can start the computation in any layer, we must wait

for the computation of previous layer to finish. Only then we can proceed with the next layer. As a consequence such serialized computation takes more time.

Figure 4.7 illustrates the accuracy of the model for the task of classifying the handwritten digits. The accuracy is measured in Mean Square error, for formal definition see Equation 2.7. The Mean Square Error (MSE) is depicted for various numbers of neurons in hidden layer. Otherwise the network parameters used were the same as in previous testing.

We can see that for more than 1000 hidden neurons the error is increasing, and the variability of error is increasing as well. This happens because there are too many weights, which slows down the backpropagation. The networks with less than 1000 hidden neurons on the other hand are producing stable results with low error rates.

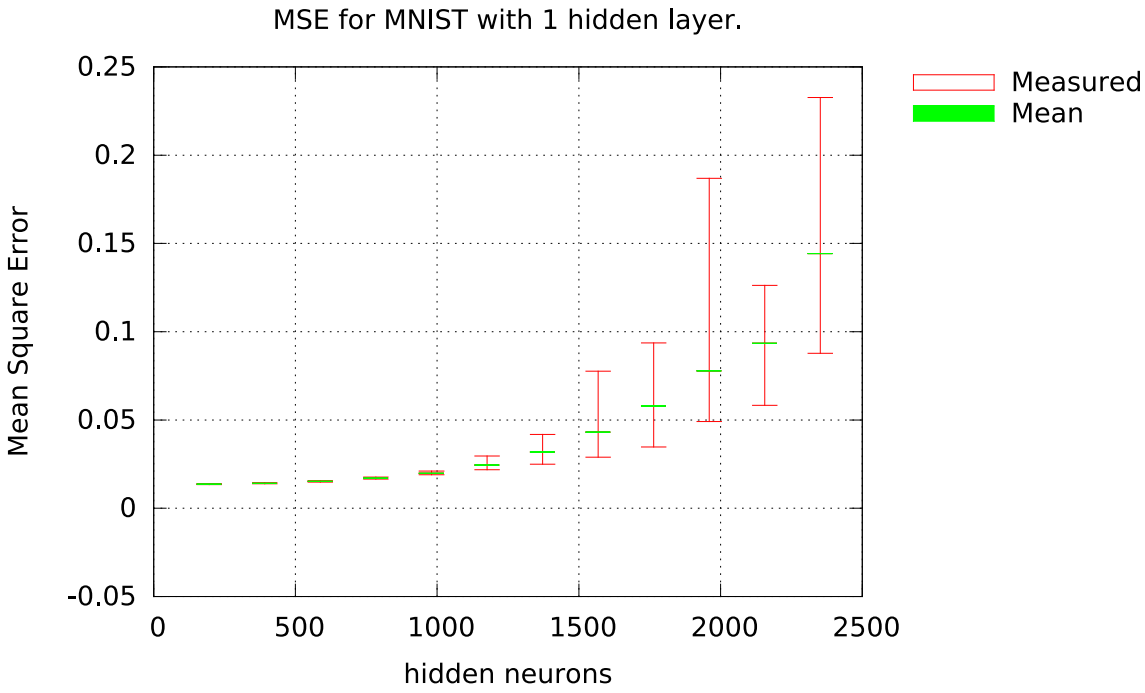


Figure 4.7: MSE results of the MLP network solving MNIST with a variable number of neurons in one hidden layer. The values are averaged over 20 runs. The test was run on the testing machine using GPU for parallel computations with 32bit precision.

4.5 Convolutional Neural Network

The second tested model is the Convolutional Neural Network. It is formally described in Section 2.3. The main architectural difference when compared to MLP is the reduction of neuron interconnections. Less connections quite naturally imply a faster computation. On the other hand, CNNs usually involve more layers when compared with traditional MLPs. Having a deeper network then supports improved recognition of more abstract patterns.

4.5.1 Recognition of handwritten digits

I tested the Convolutional network on the MNIST dataset, so we can easily compare the results achieved with MLP network in Section 4.4.3. The testing conducted followed exactly the same procedures, only with different network parameters to better suit the given model.

The purpose of the first set of tests I ran was to determine suitable learning rates for Convolutional network. The test can be run by launching a shell script in Code Snippet 4.3 below.

```
$ ./tools/run-tests.sh tools/mnist-cnn-lr.conf.sh
```

Snippet 4.3: Shell commands used to run the tests with varying learning rates for CNN implementation on the MNIST dataset.

I tested different learning rate values in the range between 0.0001 and 1. The results can be seen in Figure 4.8. The Mean Square Error keeps slightly improving up to the learning rate of 0.001. At a rate of 0.0001 the error significantly increases. For this reason I used the learning rate of 0.001 for the following tests.

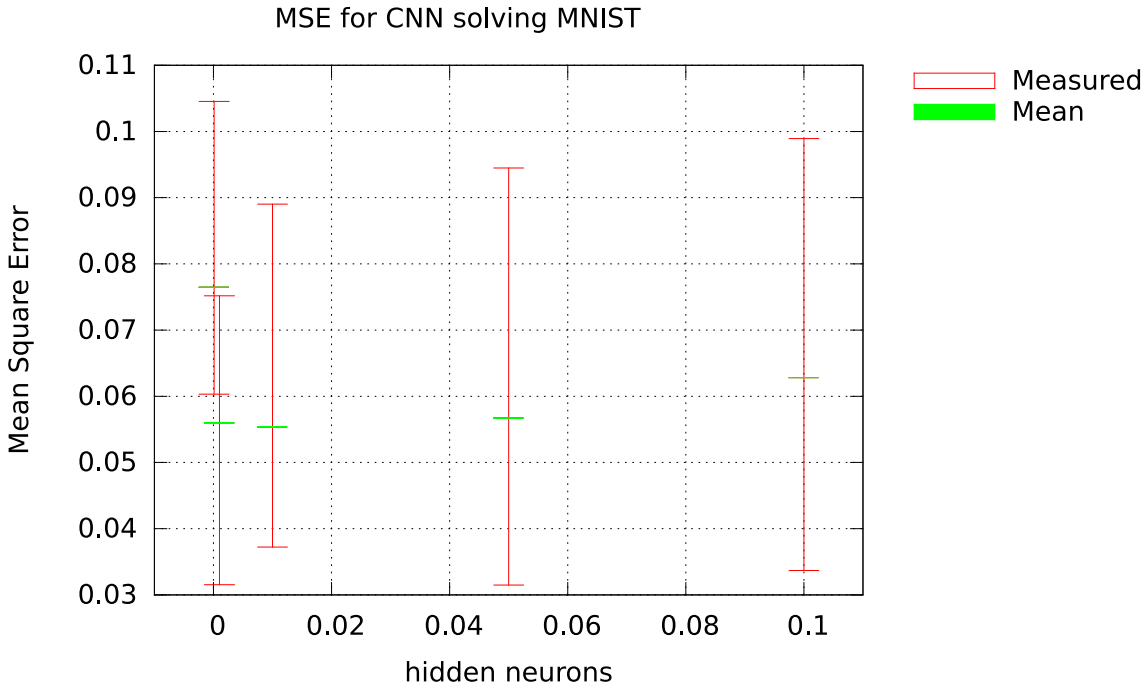


Figure 4.8: MSE results of the CNN solving MNIST with variable learning rates. The values are averaged over 20 runs. The test was run on the testing machine using GPU for parallel computations with 32bit precision.

The next set of tests I conducted was to determine the time performance of my implementation of CNNs in relation to the number of features in the first convolutional layer. This set of tests can be run by launching a shell script in Code Snippet 4.4 below.

```
$ ./tools/run-tests.sh tools/mnist-cnn-lr.conf.sh
```

Snippet 4.4: Shell commands used to run the tests of CNN implementation on the MNIST dataset.

I used the network configuration below:

- input layer
  - 28×28 neurons
- convolutional layer
  - 5×5 convolutional window
  - 20 - 100 feature maps
  - learning rate of 0.001
  - bias enabled
- subsampling layer
  - 2×2 subsampling window
  - learning rate of 1
- convolutional layer
  - 5×5 convolutional window
  - 40 feature maps
  - learning rate of 0.001
  - bias enabled
- subsampling layer
  - 2×2 subsampling window
  - learning rate of 1
- fully connected layer
  - 500 neurons
  - learning rate of 0.001
  - bias enabled
- output layer
  - 10 neurons
  - learning rate of 0.001
  - bias enabled

The results of the test runs can be seen in Figure 4.9. Both single and double precision was tested. Double precision again requires more time to compute.

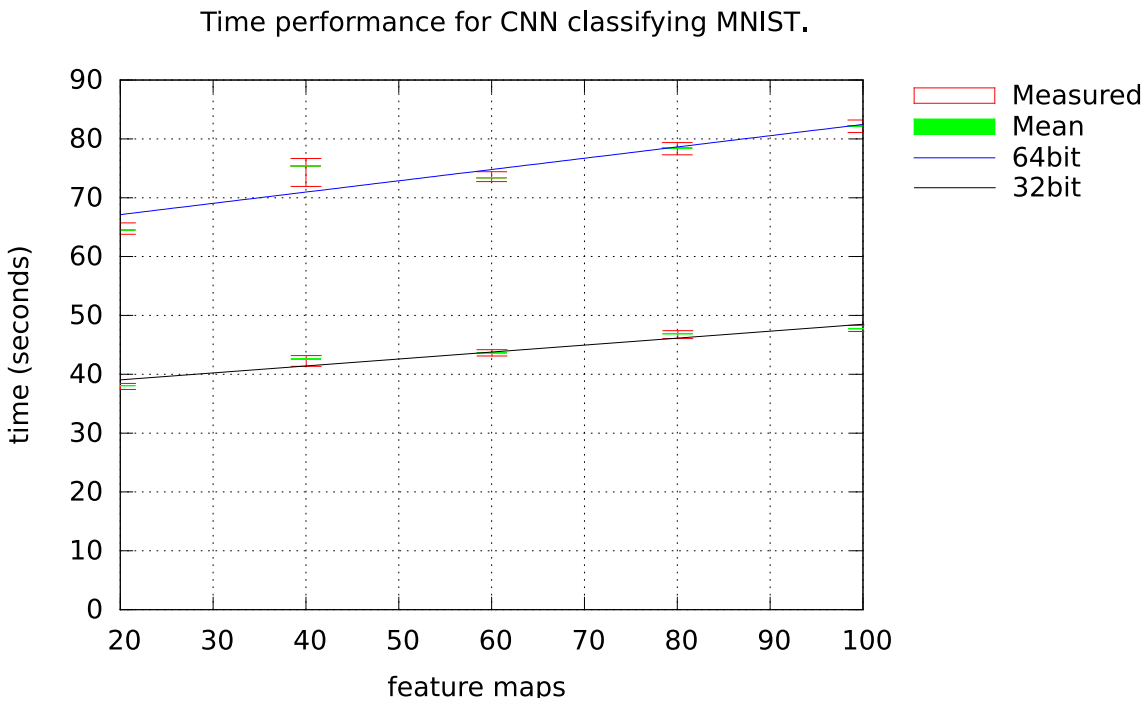


Figure 4.9: The time results of the Convolutional Neural Network solving the MNIST problem with a variable number of feature maps of size 5×5 pixels. The graph compares the results obtained for both 32bit and 64bit precisions. The values are averaged over 20 runs. The test was run on the testing machine using GPU for parallel computations.

4.6 Deep Belief Network

The last tested network type is the Deep Belief Network. It is the most performance demanding one of all the implemented models. The reasons are the extra pretraining phase, which needs to train sequentially layer by layer iterating the training dataset multiple times, the probabilistic nature of the sampling algorithms, and the deep architecture. However, once the network is trained, it behaves the same way as a basic MLP network. This makes it suitable for tasks where the performance of classification of unseen data is more important than the training time.

The depth of the DBN network is very flexible, which allows to construct a deeper network for recognizing very abstract features. In case of solving simpler tasks, a more shallow network will suffice and provide the benefit of a better performance. The number of hidden neurons is also very flexible, unlike in the case of CNN for example, which needs to adjust their number according to the size of the input images.

4.6.1 Recognition of handwritten digits

Again, I tested the DBN network on the MNIST dataset, so that the results are easily comparable between the different network types, as well as different implementations. The testing was performed with the same methodology, by

running the automated Code Snippet 4.5 below.

```
$ ./tools/run-tests.sh tools/mnist-dbn.conf.sh
```

Snippet 4.5: Shell commands used to run the tests of DBN implementation on the MNIST dataset.

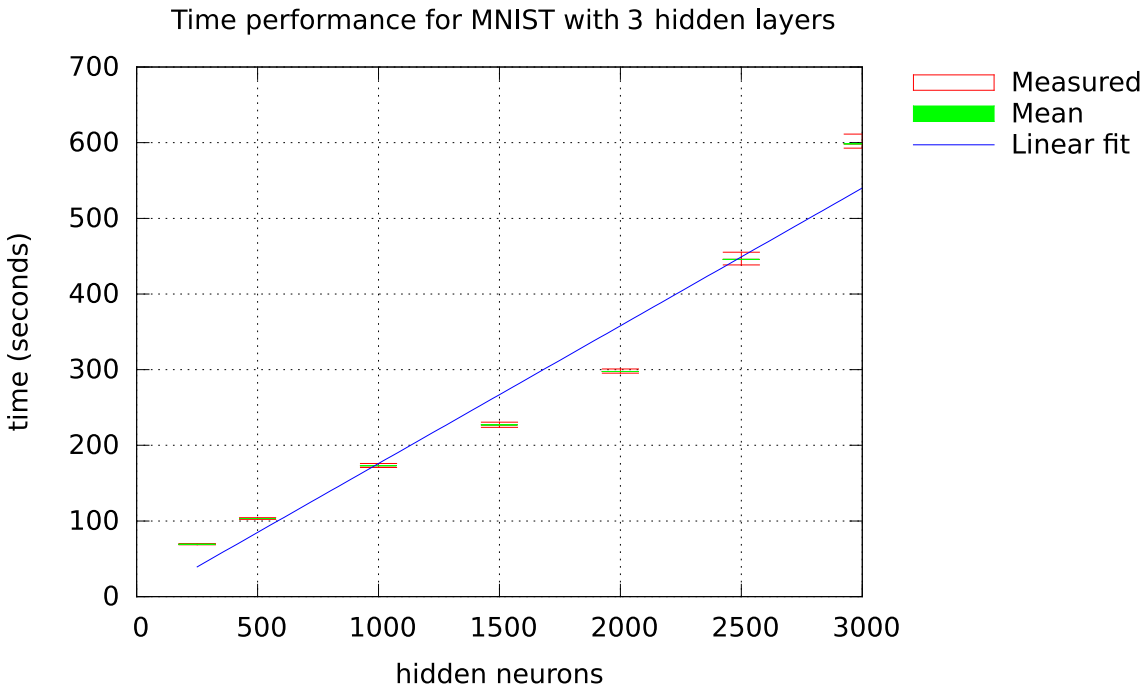


Figure 4.10: The time results of the DBN solving MNIST in 1 pretraining and 1 training epoch with a variable number of neurons in three hidden layers. The values are averaged over 10 runs. The test was run on the testing machine using GPU for parallel computations with 32bit precision.

The DBN network is initialized with random weights and biases polled from normal probability distribution with zero mean and standard deviation of 0.01. The network is further configured to be composed of the below layers:

- input layer
  - 28×28 neurons
- RBM layer (repeated 3 times)
  - 250 - 3000 hidden neurons
  - non-persistent contrastive divergence
  - 1 Gibbs sampling step (CD-1)
  - learning rate of 0.01
  - with bias
- MLP layer (output layer)

- 10 output neurons
- learning rate of 0.1
- with bias

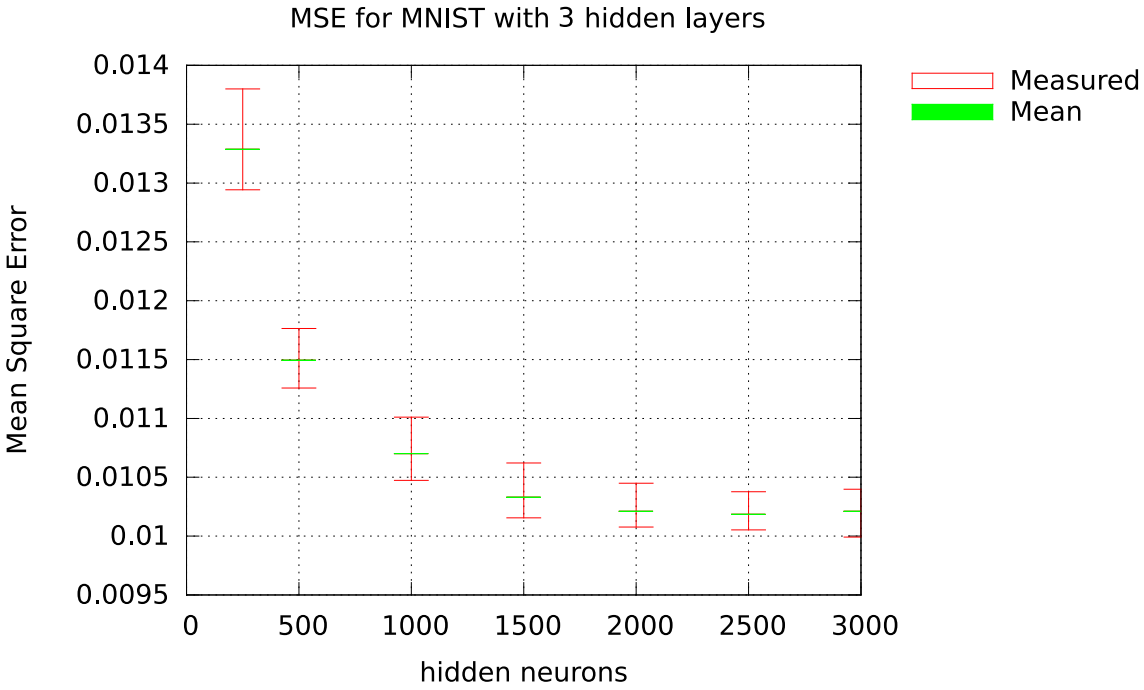


Figure 4.11: MSE results of the DBN solving MNIST in 1 pretraining and 1 training epoch with a variable number of neurons in three hidden layers. The values are averaged over 10 runs. The test was run on the testing machine using GPU for parallel computations with 32bit precision.

The time performance is the most demanding one of all implemented network types, which can be seen in Figure 4.10. The time required for the computation grows linearly with the number of neurons within a single hidden layer. This is also true when stacking multiple layers one onto each other. Additional tests and reports supporting this statement are available on the attached optical disk.

The Mean Square Error rates for solving MNIST can be seen in Figure 4.11. Here we can see that, in a single pretraining epoch, the bigger the network, the better results are achieved. Having less neurons makes it harder for the network to comprehend the features. Having more neurons quickly enables better recognition.

4.7 Comparison with other testing frameworks

A big part of this work is to learn how to parallelize and optimize artificial neural networks. The previous sections and the results attached on the optical disk sufficiently show that the parallel implementation using GPUs significantly improves the performance when compared with the serial CPU implementation.

In this section we will instead focus on how the `deepframe` implementation created within the framework of this work can be compared to other state of the art frameworks.

The frameworks chosen for comparison were picked based on their popularity, size of their user base, maturity, and development activity. These criteria yielded two winners - Caffe and Theano.

During testing, both frameworks were installed and run on the same hardware. They were configured with the same parameters for the considered network types, the same network architecture, learning rates, bias, and number of epochs. All comparison tests were performed on the same problem - recognition of handwritten digits from the MNIST dataset. All tests were carried out using automated scripts, which are repeatable, and automatically generate all the reports used in the comparisons below. The conclusions should therefore be as fair as possible.

4.7.1 Caffe framework

Caffe is a deep learning framework developed by the Berkeley Vision and Learning Center at University of Californuia, Berkeley, and by other community contributors. The Caffe community tightly cooperates with NVIDIA, which allows them to support the latest NVIDIA libraries, including cuDNN library for deep learning. This results into a very efficient implementation supporting the latest software and hardware optimizations.

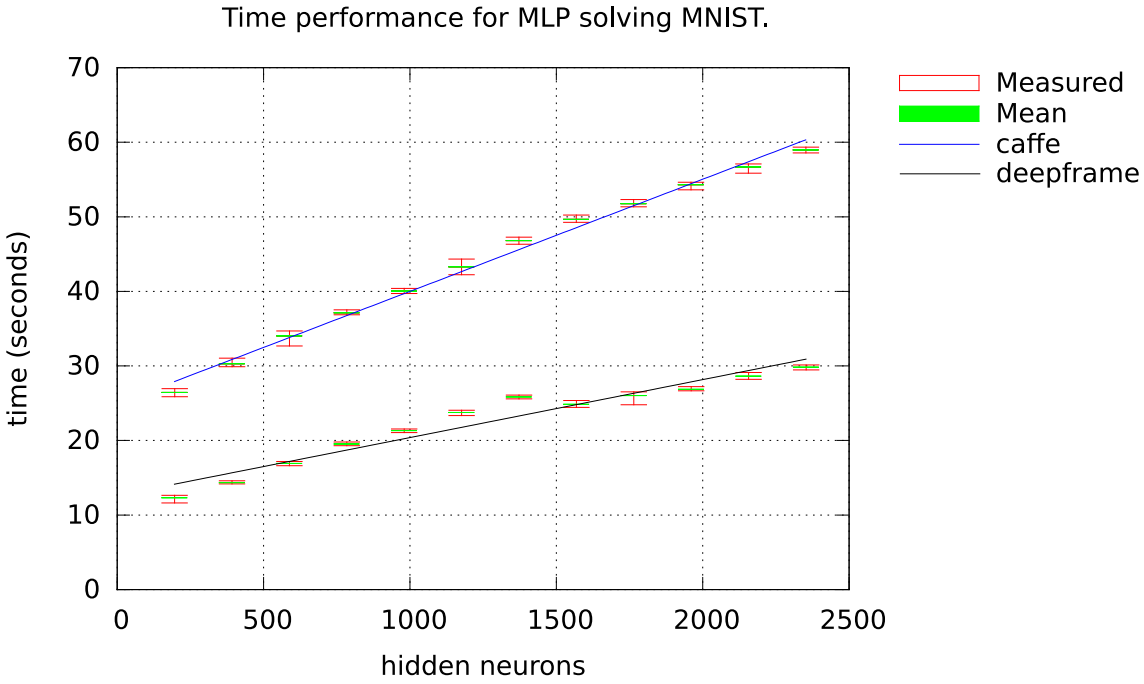


Figure 4.12: The time results of the MNIST test run on the MLP network with a variable number of neurons in one hidden layer using deepframe vs. caffe. The values are averaged over 20 runs. The test was run on the testing machine using GPU for parallel computations with 32bit precision.

Caffe is implemented in C++, which is the same language I picked for `deepframe`, too. Caffe supports a wide variety of network layers, which can be combined into

a neural network of any desired architecture. It supports both MLP and CNN architectures, so I decided to test these and compare the results with my implementation.

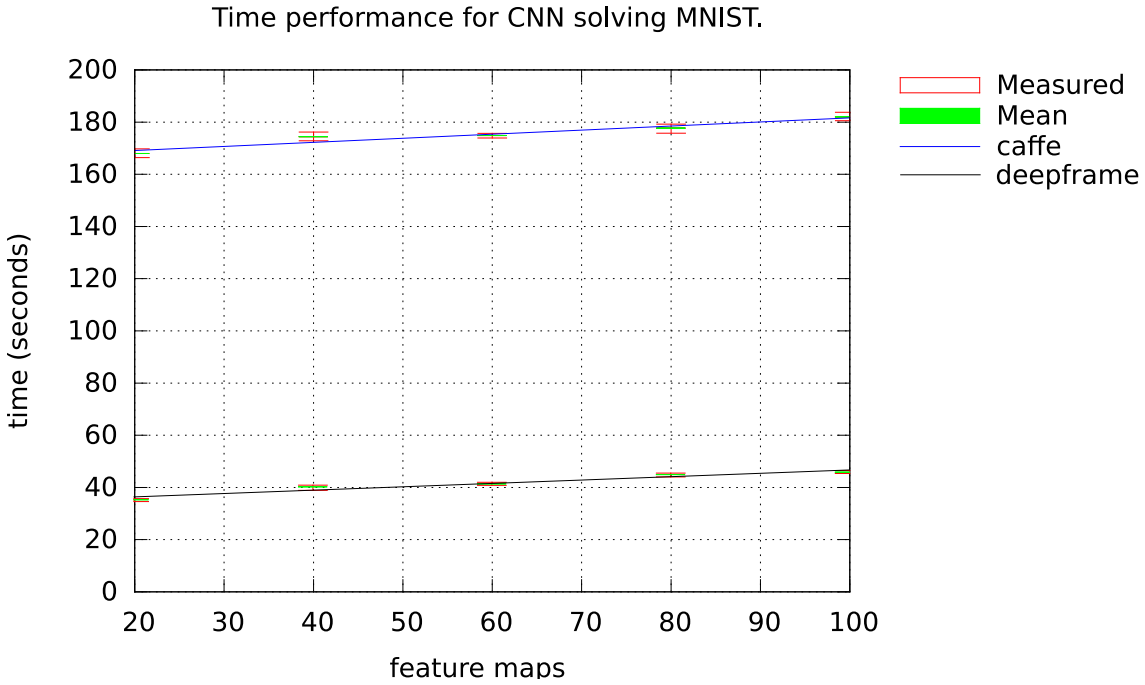


Figure 4.13: The time results of the MNIST test run on the CNN network with a variable number of feature maps using deepframe vs. caffe. The values are averaged over 20 runs. The test was run on the testing machine using GPU for parallel computations with 32bit precision.

The scripts used for testing can be found on the attached optical disk. They need to be copied into the `examples` directory inside the Caffe installation folder. The test set for MLP network can be run by entering this newly created directory and launching the commands in Code Snippet 4.6. The first command will run the tests, the second command will generate the test results. Once finished, the test results will be located in the `test-output` folder. The test results referenced in this work can be found in the same folder on the attached optical disk.

```
$ ./tools/run-tests.sh tools/mnist-mlp.conf.sh
$ ./tools/generate-reports.sh tools/mnist-mlp.conf.sh
```

Snippet 4.6: Shell commands used to run the tests of MLP implementation on MNIST dataset with Caffe framework.

The test set for MLP network consists of 20 repeated runs for each network configuration. The configurations differ by the number of neurons in one hidden layer. The results show, that the parallel implementation of `deepframe` requires roughly half of the computational time of parallel Caffe implementation (see Figure 4.12). Note that `deepframe` also scales better.

The test sets for CNN can be launched by the commands in Code Snippet 4.7. They consist of 20 repeated runs for each tested network configuration. The configurations differ by the number of features in the convolutional layer. The

results show, that the parallel implementation of `deepframe` requires roughly 1/4 of the computational time of parallel Caffe implementation (see Figure 4.13). Although both implementations scale with the same ratio, it is quite a significant improvement.

```
$ ./tools/run-tests.sh tools/mnist-cnn.conf.sh
$ ./tools/generate-reports.sh tools/mnist-cnn.conf.sh
```

Snippet 4.7: Shell commands used to run the tests of MLP implementation on MNIST dataset with Caffe framework.

The testing described above presents `deepframe` as a better performing framework for working with MLP and CNN networks on GPU. We already described the optimizations made to achieve these remarkable results in Chapter 3, but I will highlight the most important ones here. One of the key optimizations is the global memory allocation for the whole neural network. This improves the memory collocation and thus enables faster memory reads and updates. Another important optimization was to incorporate the learning rate coefficient directly into the matrix multiplication call. This saves an additional vector multiplication, which would be otherwise required in each backpropagation step. Yet another optimization was done in the backpropagation step of the fully-connected layer, where I implemented custom CUDA kernels for efficient computations of the gradients.

4.7.2 Theano

Theano is a Python library that allows to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. It is one of the most used CPU and GPU mathematical compilers – especially in the machine learning community [34]. It integrates with NumPy, the fundamental package for scientific computing with Python. Numpy contains, among other things, a powerful n-dimensional array object, and useful linear algebra and random number capabilities. On top of NumPy, Theano provides a transparent use of GPU to perform data-intensive calculations, C code generation to evaluate expressions faster, and other speed and stability optimizations.

An exceptionally useful feature of Theano is its efficient symbolic differentiation capability. This means it can evaluate derivatives for mathematical functions with one or many input variables. This is achieved by overloading the standard Python operators to construct a computational graph for any given function. Theano then optimizes this graph and generates sequences of operations which evaluate the required derivatives. The derivative function produced by this process has optimal complexity and does not suffer from round-off errors.

The above mentioned features of Theano make it extremely suitable for fast prototyping of new machine learning algorithms. However, Theano itself is in essence a framework for evaluating mathematical expressions. It does not contain any implementation of algorithms used in machine learning or artificial intelligence. To perform the testing on DBN network, I used the Theano-based implementation from Deep Learning Tutorials [8] published by the Motreal Institute for Learning Algorithms.

The scripts used for testing can be found on the attached optical disk accom-

panied with the code from Deep Learning Tutorials. Python interpreter must be installed on the system to be able to run the DBN code, and `bash` must be installed to run the testing scripts. Note that only computations with 32bit floating point precision can be accelerated on GPU, as Theano currently does not support 64bit precision. The test sets can be run by launching the commands in Code Snippet 4.8 in the root directory of Deep Learning Tutorials.

```
$ ./tools/run-tests.sh tools/mnist-dbn.conf.sh
$ ./tools/generate-reports.sh tools/mnist-dbn.conf.sh
```

Snippet 4.8: Shell commands used to run the tests of DBN implementation on MNIST dataset with Theano framework.

The test sets for DBN network consists of 10 repeated runs of each network configuration. All the configurations define a network with three hidden RBM layers followed by a single fully-connected perceptron layer with 10 output neurons. Each test is run for exactly one pretraining and one training epoch. For a more detailed specification of the network configuration see Section 4.6.1, as the same configuration was used for `deepframe` test cases.

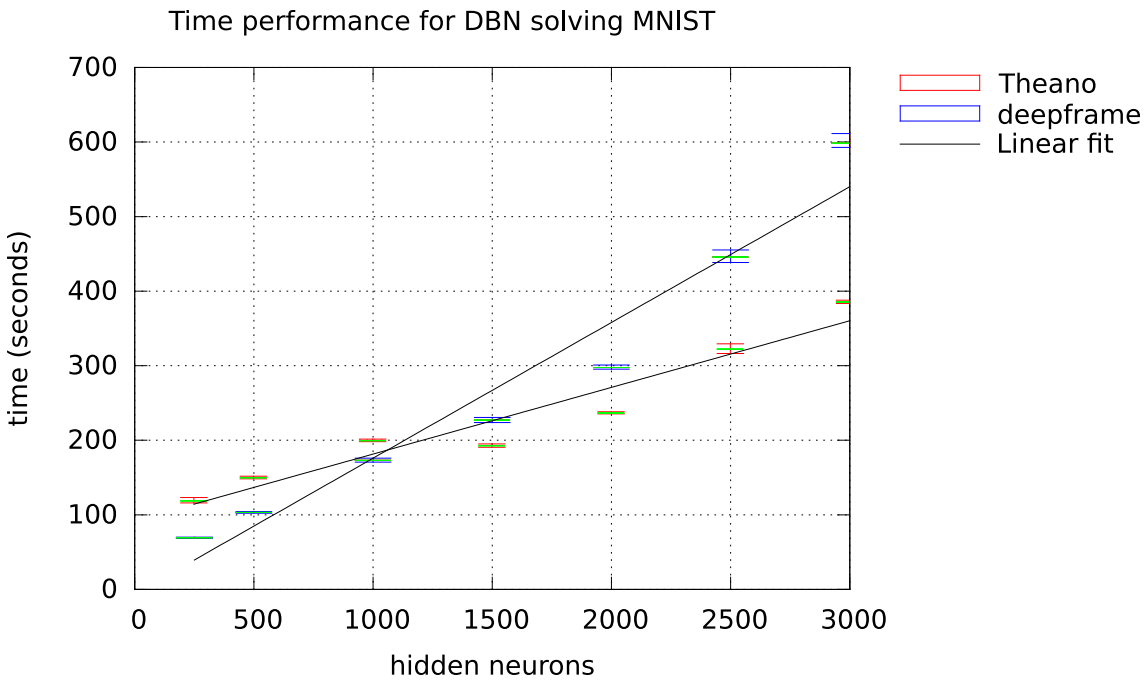


Figure 4.14: The time results of the MNIST test run on the DBN network with three hidden RBM layers after one pretraining and one training epoch using deepframe vs. Theano. The values are averaged over 10 runs. The test was run on the testing machine using GPU for parallel computations with 32bit precision.

The results show, that the implementation of `deepframe` performs slightly better for RBM layers with up to 1000 hidden neurons (see Figure 4.14). However, the Theano implementation starts to show better performance for the test cases with RBM layers containing 1500 and more hidden neurons. This is noticable especially with 2500 and 3000 hidden neurons, where the performance gap gets even bigger. The results suggest, that `deepframe` does not scale linearly.

Further investigation could be done to identify the bottlenecks in the parallel implementation and resolve this scaling problem.

5. Conclusion

Artificial neural networks represent a universal model, which can be leveraged to solve a great variety of tasks. The latest research showed a significant progress of this field in the past few decades. Deep architectures of neural networks started to gain attention after proving success in image and speech recognition in the years after 2000. After the parallel hardware became available for reasonable prices, it fueled the research of efficient optimization of deep neural networks by leveraging parallel architectures.

In this thesis I studied three different models of deep neural networks. I explained the theory behind each model and described how it is trained to recognize the features in input patterns. The models were also implemented on parallel hardware leveraging CUDA-capable GPUs, and subsequently tested using custom-crafted automation scripts. All these scripts and tested configurations for all problems are available on the attached optical disk and can be launched by issuing simple commands (see Chapter 4).

The first model of the fully-connected multilayer perceptron (MLP) was implemented with the option to perform extensive computations on both the CPU or the GPU. The capabilities of this model were tested on the trivial problems of evaluating XOR operator and 4-bit sum, as well as on a more complex handwritten digit recognition using the MNIST dataset.

The performance comparisons between CPU and GPU confirmed GPU with its parallelization capabilities as a clear winner, with the exception of extremely small network architectures (XOR problem, 4-bit sum problem). Further comparisons of the `deepframe` application implemented as a part of this thesis with the `caffe` framework showed, that the MLP model performed significantly better when run on `deepframe`.

The second model of the convolutional neural network (CNN) was also implemented with the possibility to perform extensive computations on both the CPU or the GPU. The CNN model is designed primarily for image recognition, and was therefore only tested on the problem of handwritten digit recognition again using the MNIST dataset.

The performance comparisons of CNN between CPU and GPU also presented GPU as a clear winner. The comparison with `caffe` framework again resulted in a significantly faster execution in favor of `deepframe`. The difference was even more noticeable than with MLP, however, both frameworks scaled evenly with an increasing number of feature maps.

The model of the deep belief network (DBN) was again tested on the problem of handwritten digit recognition using the MNIST dataset. This last model was implemented only on GPU. The recurrent networks are much more computationally intensive than feed-forward networks, which was presented in Section 4.6. Therefore running DBNs on CPU does not yield acceptable run times for non-trivial problems, and definitely does not surpass the performance benefits of parallel architecture. This is also supported by the experiments with the MLP and CNN models. These reasons led me into making the decision of not implementing the DBNs on CPU, and instead devote more effort into optimizing the parallel implementation.

The performance achievements of the DBN implementation were compared with a third-party framework `Theano`. The results showed, that for small hidden layers (up to 1000 neurons) `deepframe` has a better performance. With bigger layers `Theano` takes the lead and scales noticeably better.

5.1 Further work

The new tool for working with artificial neural networks achieved remarkable performance results for certain network types. However, the other contemporary frameworks support a much wider functionality regarding layer types, network or layer parametrization, learning algorithms, network architectures, error estimation, support for input formats, etc. This is achieved with at least tens or hundreds of active developers, years of research and big user communities. The scope of this work only included a minimum viable product, which achieves a subset of the above mentioned functionalities.

The most common and useful functionality missing in the `deepframe` framework is the support for parallel processing of mini-batches. This would enable faster training by postponing the weight updates till after the mini-batch is processed.

The comparison of deep belief network implementation with the `Theano` framework showed some potential for improvement in performance. An important improvement would be to rework this implementation to optimize some of the parallel code by implementing custom kernels. Further performance improvements could be achieved by implementing the support for the latest cuDNN library from NVIDIA.

In the current implementation, the weight initialization interval is configured on per-network basis. However, automatically adjusting both learning rates and weight initialization intervals based on the layer size should also lead into improved classification accuracy. This would be very beneficial for MLPs and CNNs composed of layers with considerably different sizes.

Since the deep neural networks are currently a popular research topic, the future enhancements should bear this in mind and watch for the latest advances in this field. One such example happened recently, when computers were able to beat the best human players in the game of Go using deep neural networks trained with a novel combination of supervised and reinforcement learning [31]. Any successful artificial intelligence framework must keep pace with the latest scientific breakthroughs, as they enable new possibilities to solve problems, which were never possible before.

Bibliography

[1] Atencia, M. A.; Joya, G. and Sandoval, F. *A Formal Model for Definition and Simulation of Generic Neural Networks*. In: Neural Processing Letters. 2000, Volume 11, Issue 2, pp 87-105. ISSN: 1370-4621, 2000.

[2] Aydin, M. M., Yildirim M. S., Karpuz O. and Ghasemlou, K. *Modeling of driver lane choice behavior with Artificial neural networks (ANN) and Linear regression (LR) analysis on deformed roads*. In: Computer Science & Engineering. 2014, Volume 4, Issue 1, p. 47.

[3] Barrachina, Sergio, Castillo, Maribel, Igual, Francisco D, Mayo, Rafael and Quintana-Orti, Enrique S. *Evaluation and tuning of the level 3 CUBLAS for graphics processors*. In: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on 2008, pp. 1–8.

[4] Bengio, Y., Lamblin, P., Popovici, D. and Larochelle, H. *Greedy Layer-Wise Training of Deep Networks*. In: Advances in Neural Information Processing Systems. 2007, 19(NIPS'06):153-160. MIT Press.

[5] Carreira-Perpinan, M. A., and Hinton, G. E. *On Contrastive Divergence Learning*. In: AISTATS. 2005, 10:33-40.

[6] Clark, Christopher and Storkey, Amos. *Teaching Deep Convolutional Neural Networks to Play Go*. arXiv preprint arXiv:1412.3409 2014.

[7] Collobert, Ronan and Weston, Jason. *A unified architecture for natural language processing: Deep neural networks with multitask learning*. In: Proceedings of the 25th international conference on Machine learning. 2008, ACM, pp. 160–167.

[8] Deep Learning Tutorials Development Team. *Deep Learning Tutorials*. Downloaded 5 November 2015. URL: https://github.com/lisa-lab/DeepLearningTutorials.git

[9] Fahlman, S. E., Hinton, G. E. and Sejnowski, T. J. *Massively parallel architectures for A.I.: Netl, Thistle, and Boltzmann machines*. In: Proceedings of the National Conference on Artificial Intelligence. 1983, Washington DC.

[10] Green, Timothy. *Advanced Micro Devices Inc. Loses More Market Share to NVIDIA*. In: The Motley Fool. 2015. Downloaded 10 April 2016. Data sourced from Jon Peddie Research and Mercury Research. URL: http://www.fool.com/investing/general/2015/08/23/advanced-micro-devices-inc-loses-more-market-share.aspx

[11] Guyon, Isabelle. *A Scaling Law for the Validation-Set Training-Set Size Ratio*. In: AT&T Bell Laboratories. 1997.

[12] Haykin, Simon. *Neural networks : a comprehensive foundation*. Second edition. Delhi: Pearson Education, 1999. ISBN 81-7808-300-0.

[13] Herculano-Houzel, Suzana and Lent, Roberto. *Isotropic Fractionator: A Simple, Rapid Method for the Quantification of Total Cell and Neuron Numbers in the Brain*. In: The Journal of Neuroscience. 9 March 2005, 25(10): 2518-2521. ISSN: 0270-6474, 2005.

[14] Hinton, G. E. *A practical guide to training restricted Boltzmann machines*. In: Momentum. 2010, 9(1):926.

[15] Hinton, G. E. *Training products of experts by minimizing contrastive divergence*. In: Neural Computation. 2002, 14(8):1711–1800.

[16] Hinton, G. E. and Salakhutdinov, R. R. *Reducing the Dimensionality of Data with Neural Networks*. In: Science. 2006, 313(5786):504-507.

[17] Hopfield, J. J. *Neural networks and physical systems with emergent collective computational abilities*. In: Proceedings of the National Academy of Sciences. 1982, Volume 79, Issue 8, pp 2554-2558.

[18] Hornik, Kurt, Stinchcombe, Maxwell and White, Halbert. *Multilayer feedforward networks are universal approximators*. In: Neural networks. 1989, Volume 2, Issue 5, pp 359–366.

[19] Hsu, Che-Chiang and Chia-Yon, Chen. *Regional load forecasting in Taiwan - applications of artificial neural networks*. In: Energy conversion and Management. 2003, Volume 44, Issue 12, pp 1941–1949.

[20] Hubel, D. and Wiesel, T. *Receptive fields and functional architecture of monkey striate cortex*. In: Journal of Physiology. 1968, Volume 195, Issue 1, pp 215-243. ISSN: 0022-3751, 1968.

[21] Jia, Baozhi, Weiguo, Feng, and Ming, Zhu. *Obstacle detection in single images with deep neural networks*. In: Signal, Image and Video Processing 2015, pp. 1–8.

[22] LeCun, Yann, Bottou, Léon, Bengio, Yoshua and Haffner, Patrick. *Gradient-based learning applied to document recognition*. In: Proceedings of the IEEE 86, no. 11 1998, no. 86, pp. 499–514. ISSN: 2278-2324, 1998.

[23] LeCun, Yann, Cortes, Corinna and Burges, Christopher J. C. *The MNIST database of handwritten digits*. 1998. Downloaded 13 February 2015. URL: http://yann.lecun.com/exdb/mnist/

[24] Magnotta, Vincent, Heckel, Dan, Andreasen, Nancy, Cizadlo, Ted, Corson, Patricia, Ehrhardt, James and Yuh, William. *Measurement of Brain Structures with Artificial Neural Networks: Two- and Three-dimensional Applications*. In: Radiology. 1999, Volume 211, Issue 3, pp. 781–790.

[25] McCulloh, Warren S. and Pitts, Walter. *A logical calculus of the ideas immanent in nervous activity*. In: The bulletin of mathematical biophysics. 1943, Volume 5, Issue 4, pp 115–133. ISSN: 0007-4985, 1943.

[26] Minsky, Marvin and Papert, Seymour. *Perceptrons*. 1969, MIT Press, Cambridge.

[27] NVIDIA. *NVIDIA DIGITS Devbox*. 2015. Downloaded 23 March 2015. URL: https://developer.nvidia.com/devbox

[28] Perry, O. Patrick. *Cross-Validation for Unsupervised Learning*. In: arXiv preprint arXiv:0909.3052. 2009.

[29] Rumelhart, David E., Hinton, Geoffrey E. and Williams, Ronald J. *Learning representations by back-propagating errors*. In: Nature. 1986, Volume 323, pp 533–536.

[30] Rupp, Karl. *CPU, GPU and MIC Hardware Characteristics over Time*. 2013. Downloaded 26 February 2015. URL: http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/. Published under Attribution 4.0 International license.

[31] Silver, David, Huang, Aja, Maddison, Chris J., Guez, Arthur, Sifre, Laurent, Driessche, George van den, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, Dieleman, Sander, Grewe, Dominik, Nham, John, Kalchbrenner, Nal, Sutskever, Ilya, Lillicrap, Timothy, Leach, Madeleine, Kavukcuoglu, Koray, Graepel, Thore and Hassabis, Demis. *Mastering the game of Go with deep neural networks and tree search*. In: Nature 2016, Volume 529, Issue 7587: 484–489.

[32] Smolensky, Paul, Rumelhart, David E. and McLelland, James L. *Chapter 6: Information Processing in Dynamical Systems: Foundations of Harmony Theory*. In: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations, 1986. MIT Press. pp. 194–281. ISBN 0-262-68053-X.

[33] Strecker, Uwe and Uden, Richard. *Data mining of 3D poststack seismic attribute volumes using Kohonen self-organizing maps*. In: The Leading Edge. 2002, Volume 21, Issue 10, pp. 1032.

[34] Theano Development Team. *Theano: A Python framework for fast computation of mathematical expressions*. In: arXiv e-prints. 2016, Volume: 1605.02688.

[35] Tieleman, T. *Training restricted Boltzmann machines using approximations to the likelihood gradient*. In: Proceedings of the 25th international conference on Machine learning. 2008, ACM'08:1064-1071.

Glossary

ANN Artificial Neural Network is a computational model based on the structure and functions of biological neural networks. Information that flows through the network affects the structure of the ANN, which enables learning capabilities of the model based on its input and output. 3, 7–9, 11

API Application Programming Interface is a set of functions and procedures that allow to access the features or data of a third party application, or other service. 32, 34, 38, 50, 61

BLAS The Basic Linear Algebra Subprograms are routines that provide standard building blocks for performing basic vector and matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software. 39, 55, 56, 58

CNN Convolutional neural network is a multi-layer network specifically designed to recognize features in 2-dimensional image data. 16, 17, 31, 53, 73, 74, 80, 81, 85, 86

complete path A path in a graph that cannot be prolonged. 7

CPU Central Processing Unit is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output operations specified by the instructions. Its instruction set and memory workflow is designed for universal computational tasks. 31, 35, 36, 39, 41, 43, 48, 53–55, 78, 81, 85

DBN Deep Belief Network is a type of Deep Neural Network composed of multiple layers of hidden units. The hidden units are inter-connected between the layers, however there are no connections between the units within each layer. The connections between the units are bidirectional. 23, 29–31, 50, 53, 76, 81, 82, 85, 86

FPGA Field-Programmable Gate Arrays is an integrated circuit that can be programmed with hardware description language to rewire itself to be optimized for a given task. 31–34

GPU Graphics Processing Unit is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. GPUs are however increasingly leveraged for high-throughput computations that exhibit data-parallelism, making them suitable for a wider set of problems. 32–36, 39, 41, 43, 48, 49, 52–55, 78, 81, 82, 85

kernel CUDA kernel is a portion of an application code which is executed on a GPU device. Only one kernel can be executed at a time. Each kernel can be executed by multiple arrays of parallel threads. 52, 54, 55

MLP Multilayer perceptron network is a feedforward artifical neural network consisting of at least two layers. Each layer is built solely of perceptrons interconnected with all perceptrons in the neighbouring layers. 9, 16, 17, 21, 29–31, 43, 44, 56, 69, 70, 73, 74, 80, 81, 85, 86

outer product The outer product is an algebraic operation, which is applied to a pair of vectors, treats them as matrices, and produces a matrix by performing matrix multiplication on them. 28

overfitting Occurs when a neural network captures noise as random error, or misinterprets the relationships contained in the training data. 11

pseudorandom A pseudorandom sequence of numbers satisfies most of the statistical properties of a truly random sequence but is generated by a deterministic algorithm. 34, 39, 61

quasirandom A quasirandom sequence of n-dimensional points is generated by a deterministic algorithm designed to fill an n-dimensional space evenly. 34, 39

supervised learning Learning from a dataset with known outputs. For each input pattern its expected output must be included in the dataset. 11, 21, 29

unsupervised learning Learning from a dataset, which does not contain any information about expected output. 11, 26, 29

List of Figures

Attachments

1. An optical disk is enclosed with the printed version of this thesis. The disk contains:
   (a) deepframe application
      i. application source code
      ii. bash scripts with the test cases used in this work
      iii. Git repository with the source revision history
   (b) Caffe framework
      i. application source code
      ii. Git repository with the source revision history
      iii. bash scripts with the test cases used in this work
      iv. Git repository with the revision history of the bash scripts used for testing
   (c) Deep Learning Tutorials
      i. source code of the tutorials
      ii. bash scripts with the test cases used in this work
      iii. Git repository with the source revision history
   (d) test logs and reports
      i. test results for deepframe
      ii. test results for Caffe
      iii. test results for Theano-based implementation
   (e) electronic version of this thesis
      i. the PDF file
      ii. Git repository with the revision history of LaTeX source files
2. A detailed specification of the GPU installed in the development machine.

```
Device 0: "GeForce GTX 560"
CUDA Driver Version / Runtime Version     7.0 / 6.5
CUDA Capability Major/Minor version number: 2.1
Total amount of global memory:            1023 MBytes (1072889856
    bytes)
( 7) Multiprocessors, ( 48) CUDA Cores/MP: 336 CUDA Cores
GPU Clock rate:                           1660 MHz (1.66 GHz)
Memory Clock rate:                        2004 Mhz
Memory Bus Width:                         256-bit
L2 Cache Size:                            524288 bytes
Maximum Texture Dimension Size (x,y,z)    1D=(65536), 2D=(65536,
    65535), 3D=(2048, 2048, 2048)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048
    layers
Total amount of constant memory:          65536 bytes
Total amount of shared memory per block:  49152 bytes
Total number of registers available per block: 32768
```

```
Warp size:                                     32
Maximum number of threads per multiprocessor: 1536
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (65535, 65535, 65535)
Maximum memory pitch:                          2147483647 bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
Run time limit on kernels:                     Yes
Integrated GPU sharing Host Memory:            No
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Disabled
Device supports Unified Addressing (UVA):      Yes
Device PCI Bus ID / PCI location ID:           3 / 0
```

3. A detailed specification of the GPU installed in the testing machine.

```
Device 0: "GeForce GTX 980"
CUDA Driver Version / Runtime Version          7.0 / 6.5
CUDA Capability Major/Minor version number: 5.2
Total amount of global memory:                 4096 MBytes (4294770688
    bytes)
(16) Multiprocessors, (128) CUDA Cores/MP: 2048 CUDA Cores
GPU Clock rate:                                1278 MHz (1.28 GHz)
Memory Clock rate:                             3505 Mhz
Memory Bus Width:                              256-bit
L2 Cache Size:                                 2097152 bytes
Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536,
    65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048
    layers
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch:                          2147483647 bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
Run time limit on kernels:                     No
Integrated GPU sharing Host Memory:            No
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Disabled
Device supports Unified Addressing (UVA):      Yes
Device PCI Bus ID / PCI location ID:           1 / 0
```