Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of theoretical computer science

Master's thesis

Machine Learning in Astroinformatics Using Massively Parallel Data Processing

Bc. Tomáš Peterka

Supervisor: RNDr. Petr Škoda, CSc.

27th May 2015

Acknowledgements

I would like to thank my supervisor Dr. Petr Škoda for his endless patience and support. I greatly appreciate the help of Dr. Massimo Brescia from Astronomical Observatory of Capodimonte, National Institute of Astrophysics, who patiently explained certain concepts and was promptly modifying the DAME application. Finally I thank my friend Audrey Bonneau for asking the right questions and for her help in editing this thesis.

This research uses Caffe framework as an underlying implementation of deep networks. Caffe is maintained and developed by the Berkeley Vision and Learning Center (BVLC) with the help of an active community of contributors on GitHub.

This work uses SDSS-III datasets. Funding for SDSS-III has been provided by the Alfred P. Sloan Foundation, the Participating Institutions, the National Science Foundation, and the U.S. Department of Energy Office of Science. The SDSS-III web site is http://www.sdss3.org/.
SDSS-III is managed by the Astrophysical Research Consortium for the Participating Institutions of the SDSS-III Collaboration including the University of Arizona, the Brazilian Participation Group, Brookhaven National Laboratory, Carnegie Mellon University, University of Florida, the French Participation Group, the German Participation Group, Harvard University, the Instituto de Astrofisica de Canarias, the Michigan State/Notre Dame/JINA Participation Group, Johns Hopkins University, Lawrence Berkeley National Laboratory, Max Planck Institute for Astrophysics, Max Planck Institute for Extraterrestrial Physics, New Mexico State University, New York University, Ohio State University, Pennsylvania State University, University of Portsmouth, Princeton University, the Spanish Participation Group, University of Tokyo, University of Utah, Vanderbilt University, University of Virginia, University of Washington, and Yale University.

We also acknowledge the support of grant GAČR 13-08195S.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 27th May 2015 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Peterka, Tomáš. *Machine Learning in Astroinformatics Using Massively Parallel Data Processing.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Moderní astronomie a ostatní oblasti přírodních věd se potýkají s exponenciálně rostoucím objemem dat. Tento fenomén vedl k průniku počítačových věd do oblasti těchto čistě přírodních věd. V případě současné astronomie hovoříme o takzvané astroinformatice.

Tato práce je přehlídkou nejmodernějších paralelních algoritmů strojového učení a jejich použití na astronomických velkých datech. Sestrojili jsme klasifikátory založené na hlubokých neuronových sítích schopných běžet na grafických procesorech za použití Caffe frameworku. Vyvinuli jsme efektivní vstupní vrstvu pro Caffe modely, která je schopná pracovat s obecnými textovými soubory a má velmi intuitivní konfiguraci. Za použití klasifikátoru jsme natré-novali dva modely podle vzorových dat. Model pro spektra je dvouvrstvá konvoluční síť s přesností klasifikace přes 99% a průměrnou propustností 1440 MB/s.

Zdrojové kódy klasifikátoru jsou dostupné na github `https://github.com/vodev/vocloud-deeplearning` a big data layer pro framework Caffe na `https://github.com/atheiste/caffe/tree/big_data_layer`

Klíčová slova strojové učení, klasifikace, hluboké neuronové sítě, GPU, astroinformatika, caffe

Abstract

Modern astronomy and all natural sciences are dealing with exponentially increasing amounts of data. This phenomena resulted in the penetration of computer science into pure natural sciences. In the case of contemporary astronomy, this led to the creation of a new field called astroinformatics.

This thesis is a case study of modern parallel machine learning algorithms and their usage on astronomical big data. It resulted into deep neural network classifiers running on graphical processors and built on top of Caffe framework. An efficient input layer was added into Caffe so it is possible to use standard flat files for big data. The classifier comes with two pre-trained models to fit tabular and raw spectral data. The model for raw spectra is a two layered convolutional network whose accuracy is over 99% and average dataflow 1440 MB/s.

The source code of our classifier is available on github `https://github.com/vodev/vocloud-deeplearning` and the big data layer for Caffe framework at `https://github.com/atheiste/caffe/tree/big_data_layer`

Keywords machine learning, classification, deep neural networks, GPU, astroinformatics, caffe

Contents

List of Figures

Introduction

Motivation and objectives

Contemporary astronomy is facing a phenomena called "Data Avalanche". New astronomical devices measure in more complex manners than the previous ones and thus creating more data than current techniques can cope with. This presents a new problem when the knowledge hidden in data is sparse and can be obtained only by sophisticated techniques. Therefore a new kind of research methodology of contemporary astronomy was founded – astroinformatics. It is based on systematic application of modern informatics and advanced statistics on huge astronomical data sets. The modern approach of knowledge extraction and data understanding, which astroinformatics tackles, is sometimes being presented as the Fourth Paradigm[1].

Many Sky Surveys release their data to public such as Sloan Digital Sky Survey (SDSS). Its latest dataset DR12[1] contains 4.3 million spectra out of which 2.4 million are galaxies, 0.5 million quasars and 0.8 million are stars. Every release of a new dataset is cumulative and the latest contains observations until July 2014. Another sky survey is provided by Large Sky Area Multi-Object Fiber Spectroscopic Telescope (LAMOST), which is able to measure 4000 spectra in a single exposure because of its multi-fibre spectrograph. LAMOST has released 4.1 million spectra in its latest data release DR2[2].

Of course there are new, more powerful and bigger telescopes being built. One of them is the Large Synoptic Survey Telescope (LSST)[3], set to become operational in 2019. This telescope is expected to produce 800 panoramic images per night, the equivalent of 30 TB of data per night. Its catalogue should contain 50 trillions of records for a total size of 50 PB.

[1] www.sdss.org/dr12/scope/

[2] http://www.lamost.org/public/dr2?locale=en

[3] http://www.lsst.org/lsst/about

Problem statements

The recent breakthroughs in deep machine learning opened up to new possibilities in classification of very complex data. Those learning methods were improved to such a state that they are able to take advantage of modern GPGPUs. This became possible because the (GP)GPUs are now capable of computing with sufficient floating point precision and even contain synchronization routines. Since astronomical data are very complex in their nature, a new-type classifier could deal with them gracefully and provide both high performance and throughput of modern GPUs.

Two common astronomical classification tasks were chosen on which we will try new machine learning algorithms. It is necessary to select the appropriate machine learning algorithms which will fit the data. The first task is celestial object categorization into galaxies, quasars and stars based on their differences of magnitudes in several spectral filters. It is a classical machine learning problem dealing with a small dimensional space which has been solved by many methods so we will have comparison for accuracy. The second task is classification of Be stars based on their spectrum. The second problem is less explored area – high-dimensional space classification which resembles image classification. Since the data are very specific, the amount of models which are able to handle those data is limited.

Our aim is also to provide an efficient and versatile classifier with respect to complexity and big amounts of data in astronomy. The resulting application should be deployable in cloud and highly configurable so it is usable by any observatory willing to classify their own big amounts of data.

Chapter 1

Survey of massive parallel machine learning algorithms

In this chapter we provide a state-of-the-art overview based on the most recent conference proceedings and science publications. Many current applications are using specialised FPGA chips but that is not the aim of this thesis. Our goal is to use general purpose graphical processing unit (GPGPU) for classification so that our methods can be reused. We will omit using massive parallel technologies like MPI because machine learning tasks usually requires huge data flows which is a weak spot of distributed computations.

Our survey of literature is summarised in figure 1.1 and shows dates of publications of GPU implementations of ML algorithms. There was a tremendous increase of interest in this area starting in 2005. It might be linked to the unveiling of CUDA technology (officially) in 2006.

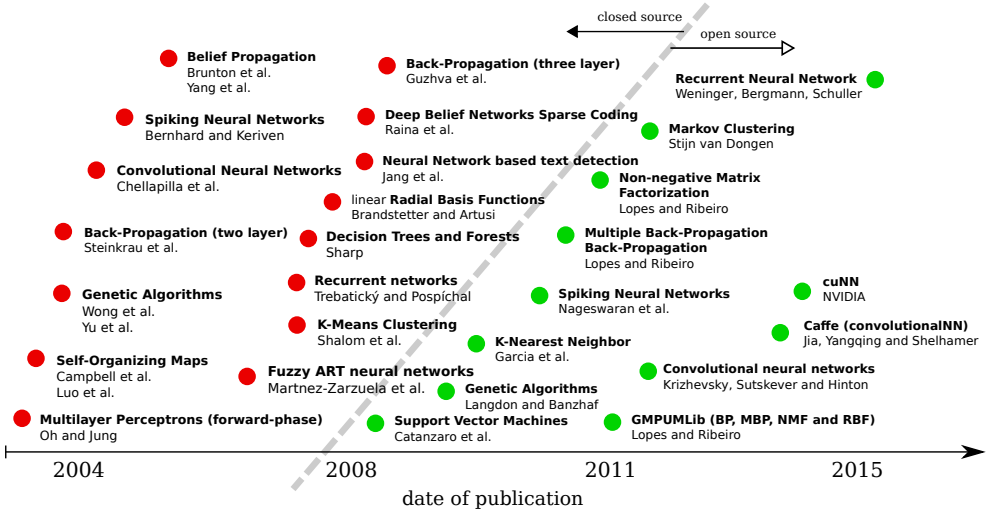


Figure 1.1: Dates of GPU implementation of ML algorithms

Following list is a summary of feasibility of ML algorithms for parallelization. The most suitable algorithms appear first in the list.

1. *Particle swarm optimization* – Parallelism is achieved through propagating global best solution slowly through neighboring particles. Even then there is approx. 300 times speedup [2].

2. *Genetic algorithms* – Island model has the highest speedup factor (cca 55) [3] over CPU and scales linearly with number of GPUs.

3. *Neural networks* – Ease of parallelism depends on interconnection between layers, the bottleneck is the learning of fully-connected layers [4].

4. *Self organizing maps* – They suffer from all-to-all communication in learning phase which can be partially suppressed by using batch learning. It will move the communication at the end of every batch resulting in 44 times speedup [5].

5. *SVM* – It is possible to learn SVM in smaller batches with an iterative version of Newton SVM algorithm. The speed-up achieved with this technique was about 45 times [6].

6. *Random forest* – It is hard to efficiently implement parallel learning for general purpose trees [7].

7. *Clustering* – Markovian chains and graph transformations are leading approaches in clustering but both are hardly decomposable.

1.1 Genetic algorithms

GAs are algorithms for local search in a huge state space. The principle is that we encode the solution of a problem into a "genome" of a gen and then mutate gens between each other and keep only the best solutions found so far. The algorithm consist of three phases:

- *crossover* – select two random genes and perform a genetic crossover
- *mutation* – select a random gene and mutate random parts of its genome
- *evaluation* – compute fitness for each gen (distance from the good solution)

As one can see GA has great potential for massive parallelization. There are three basic types of parallel genetic algorithms [8], [9]:

- *Master-slave* – One single processor performs the genetic operations and uses other processors for evaluation of individuals only. This model is useful when dealing with a small number of processors or with computationally intensive evaluations.

- *Island model* – In this model, every processor runs an independent evolutionary algorithm (EA) using a separate sub-population. The processors cooperate by regularly exchanging migrants (good individuals). The island model is particularly suitable for computer clusters, as communication is limited.
- *Diffusion model* – Here, the individuals are spatially arranged, and mate with other individuals from the local neighborhood. When parallelized, there is a lot of inter-processor communication (as every individual has to communicate with its neighbors in every iteration), but the communication is only local. Thus this paradigm is particularly suitable for massively parallel computers with a fast local intercommunication network.

One of the functional and general purpose implementation is done in project GAME[10]. The genetic algorithm in GAME can be used as a standalone model or can be incorporated into MLP as a support function for finding the best coefficients for links between neurons.

1.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is naturally decomposable local search algorithm. The main idea is that we introduce multiple agents (particles) and put them randomly in our state space. Those particles are evaluating the state they are at and they keep moving with some velocity in the state space. The direction is randomly set at the beginning but it is constantly forced towards the best solution found so far. One can see that this forces synchronization into the algorithm. Fortunately the synchronization can be weak here without any harm to the speed of convergence to the best solution.

All locals searchers can be unsynchronized and still follow the latest maximum as it is shown in work [2] which compares CPU and GPU implementations. There are too many limitations of the traditional von Neumann architecture which prohibit effective parallel implementation. Fortunately the GPU architecture suits this algorithm well.

New improvements to the original PSO algorithm [11] include the notion of *unhealthiness* to describe swarms or sub-swarms stuck at local optima, then applying random mutations to the unhealthy particles' positions.

Almost all recent GPU implementations assign one thread to each particle which, in turn, means that fitness evaluation has to be synchronized after every iteration. The latest version of GPU PSO algorithm brings two improvements on how to speedup the computation. First is allocation of a thread block per particle, each of which executes a thread per problem dimension. This way every particle evaluates its fitness function and updates position, velocity, and personal best for each dimension in parallel. Second is removal of the need to store and maintain the global best in global memory. Every particle checks

its K neighbours' personal best fitnesses, then updates its own personal best in global memory only if it is better than the previously found personal best fitness. This can speed up execution time dramatically [2].

1.3 Neural networks

Neural network can be viewed in two ways. The first one is as many nested (non)linear functions. The second one is that neural network is a sequence of matrices multiplications. Both views are valid and both suggest different ways of parallelization of the problem. There is one common obstacle for both views and that is the necessity to synchronize the results of previous operation. This requirement is indeed weakened by "interconnection" of the levels of computation but in the most common case there is full connection which means that the following phase has to wait for all results from the previous phase.

Today, most of neural networks are at least multi layer neural networks. Only this type of network is able to solve the XOR and harder problems. Neural networks are good candidates for massive parallelization since there has to be as many neurons as the inputs which are usually (nearly) fully connected to neurons in subsequent layer. The complexity of the whole structure grows exponentially with number of hidden layers that we usually suppose to be up to three fully-connected hidden layers.

There are many possibilities of implementing the parallel training phase which is the most computationally demanding one. We will describe basic ideas of parallelization mostly proposed by Krizhevsky and Yann Le Cun.

Recently there was a big boom of publicly available GPU implementations of convolutional networks which are practically a synonym of deep neural networks. This can be stated because the convolutional network is required to have more types of layers than just the convolutional one. Usually there is a max-pooling layer which de-noise feature maps and then there is always one or more non-linear transformations in order to scale the data properly followed by fully-connected layer to produce the right amount of outputs. Therefore improving convolutional networks mean improving all types of neural networks too.

The number of open source projects has increased together with popularity of the deep neural networks around 2011. The most advanced technologies now are NVIDIA `cuDNN`, `Caffe`, `cuda-convnet2`, `Torch`, `NervanaSys` and Facebook's `fbfft`. We describe most of them in greater detail in Section 1.11. The main leaders of research in this area are Yann Le Cun, Yoshua Bengio, Ilya Sutskever, Alex Krizhevsky and Geoffrey Hinton.

In the scientific area there are few GPU enabled solutions. One of them is DAME environment developed by Università di Napoli Federico II. They have implemented an MLP using GPU with speedup value around 8 over a

sequential CPU version of similar computational category [12]. The resulting application is called `DameWare` which is a whole platform for data discovery with web UI interface, scheduler and many implemented machine learning algorithms including the previously mentioned one.

Another interesting and brand new area (1996) of neural networks is *spiking neural networks* [13]. They are called the 3^{rd} generation of neurons. Those networks do not have per-layer synchronization as classical artificial neural networks but their neurons "fire" whenever they have enough input. Obviously this approach resembles more the real brain and therefore it is claimed to have better accuracy in classification and decision making. Moreover spiking networks take time into account and therefore it can decide on more kinds of problems than artificial neural network. Since there is no synchronization involved then modelling of such networks is quite straightforward either using simple electrical circuits or GPUs.

The recent parallelization approaches are distinct for every type of layers. The layers are viewed as separate models. The parallelization techniques are discussed more in detail in chapter 2.

1.4 Association analysis

The association rule learning was the first data mining area that has been implemented on special- purpose hardware (probably because it is very useful in client classification in banking and marketing area). Unfortunately this algorithm is not useful in our case.

1.5 Clustering - unsupervised classification

The basics of clustering algorithm is to search for similarities between the data. That implies a lot of communication since every new unit has to be compared with all the other pieces of data we already have.

There are many theses and articles claiming parallelization of basic SCAN algorithm [14]. The SCAN algorithm is a clustering algorithm using core-nodes which are well defined by two parameters ϵ and μ where the first one stands for a maximal radius to search for nodes and the later one expresses how many nodes has to be in the neighbourhood for a node to be considered a core- node. The way to scale this algorithm is to sort edges definitions for core-nodes labelling and then use sub-trees in the graph for actual clustering called "linking" in case of SCAN algorithm. This algorithm supposes a graph structure as its input. There are indeed new methods of transforming any input data into a graph structure. We describe those in details in the sections bellow.

Markov Clustering is a decomposable clustering algorithm mainly used in bioinformatics. MCL uses two simple algebraic operations, expansion and inflation, on the stochastic (Markov) matrix associated with a graph. The Markov matrix M associated with a graph G is defined by normalizing all columns of the adjacency matrix of G. The clustering process simulates random walks (or flow) within the graph using expansion operations, and then strengthens the flow where it is already strong, or weakens it where it is already weak using inflation operations. The application of expansions and inflations creates regions with strong internal flow (clusters) separated by boundaries within which flow is absent [15].

K-Means as the original algorithm was recently extended in `k-means++` by Arthur and Vassilvitskii [16]. Since their work enable parallelization and streaming processing, many practical implementations follow. There were also custom hardware implementations which were, for example, clustering colours in images in realtime by generating the `kd-trees` dynamically on the FPGA. Finally, Ma et al. [17] proposed a processing structure especially for GPUs that can be efficiently utilized in a wide range of data mining algorithms, like `k-means` clustering or EM clustering [18].

1.6 Classification and Regression Trees

This approach seems to be very popular in past few years. The first attempts towards parallelization of the decision tree induction led to the proposal of two efficient software-based solutions[18]:

1. SPRINT which handle massive datasets by changing the CART [19] algorithm's nature
2. SLIQ which is trying to change the way the data are stored in the memory

The SPRINT algorithm implemented the same split selection method as the one utilized in the CART algorithm, and it is considered to be the successor of the SLIQ algorithm. The SPRINT and the SLIQ algorithms achieved an almost linear speedup with respect to the number of CPUs and the sample size.

1.7 Random forest

Random forest is a classifier consisting of a collection of tree-structured classifiers $\{h(x, \Omega_k), k = 1, ...\}$ where the $\{\Omega_k\}$ are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input x [20].

Random forests can be parallelized as shown in [21] even though the author does not precisely follow the "random approach". He uses dynamic sub-tree partitioning for higher throughput. It is usable for a random forest or a tree with high degree. Random forest is a model of many trees trained using sub-samples of the training data such that each sub-sample contains subset of input attributes. Thanks to this approach we obtain mostly uncorrelated trees which we put together using ensemble method bagging. There is a lot of active research around this method, mostly at Microsoft and Bell labs.

Parallel version of random forests has been already tried with success on the same data as we have [22].

1.8 Self organizing maps

SOM allow sampling of a n dimensional set in a topological map of lower dimensionality which keeps in its topology similarities between data.

The map is constituted of a set of n-dimensional structures that are called neurons. A set of data with same dimensions (weights) as the neurons is presented to the map one by one intending to be grouped. For each showed input, two distinct steps are made, the first is where we compute a function around the map to discover the most similar neuron with the input according to this function. We call this most similar neuron the Best Matching Unit (BMU). A common function to be applied is the Euclidean distance. The second step is where we propagate the input characteristics on the neighbourhood of the BMU. On every neuron of the map equation 1.1 is applied

$$w_i(t+1) = w_i(t) + h_{ci}[x(t) - w_i(t)] \tag{1.1}$$

where w_i is the weights vector of the neuron, $x(t)$ is the presented input at time t and h_{ci} is a function that decays exponentially in function of the distance between winner neuron and the updating one. This function also depends on a learning rate which decays at each iteration. After a group of inputs is presented, the map should converge to the existing clusters on the data set. In addition, the more similar clusters should also stay closer [5].

Parallel version of SOM was implemented on the same data by a bachelor's thesis [Lukáš Lopatovský]

1.9 Support Vector Machines

Very popular method for classification of single class (either accept or reject the class). SVM can use linear separator of the state space or use kernel trick to use more complex separation of the space (e.g. spherical separation). Since the learning of SVM model requires computation of distance to each training sample it is easy to parallelize. The input can be split into smaller chunks which are evaluated independently and after that a reduction function is used.

There are interesting parallel approaches such as using iterative learning in order to reduce memory consumption or Newton SVM which optimize different than QP function as shown in work [6]. This implementation showed performance as much as 100 times faster than sequential `LibSVM`.

1.10 MapReduce Framework

Finally, there are certain papers that combine software frameworks, such as `MapReduce`, and hardware platforms, such as FPGA and graphical processing unit (GPU), for the acceleration of data mining algorithms. It provides programming abstraction, hardware architecture, and basic building blocks to developers. A `Rankboost` algorithm was implemented in this framework, which is an effective ranking algorithm and it is widely used in applications which involve data mining tasks. The implementation achieved approximately 30 times better performance when compared to the performance of a CPU-based implementation [18].

Unfortunately this technology is limited in complexity of executed code so it would not make sense to use it on more complicated models. However the underlying technology of distributed computing can be used to further expand data throughput but not data parallelism.

1.11 Available technologies

We are considering only open source implementations of ML algorithms ideally backed up by some academic publication.

1.11.1 MLlib

Increasingly popular library developed by UC Berkeley for Distributed Machine learning running on `Apache Hadoop` and `Spark`. It contains most of the machine learning algorithms. It uses `Fortran` binaries to increase its performance and so far it is tightly bounded to CPU architecture. This library is in its early stage of development. It is getting popular mostly because of industry shift to concurrent data-crunching platforms such as `Spark`.

1.11.2 GPUMlib

A new library still under active development. The inception of the library was at academical grounds of Portugal in 2011 by the paper [23]. As they claim in the paper there is no standard and easy-to-use library providing GPU implementation of the most known machine learning algorithms. The aim of this project is to become such a standard library. The library is written using newest standard `C++11` in combination with the latest `CUDA toolkit`. The library already contains many popular models like *Radial basis function*,

Restricted Bolzmann machine, *SVM* and many more. Nevertheless it lacks more complicated models so there is still a lot of space for improvement.

On that note we can justify our decision not to use this library as one of our building blocks since in our opinion the project of `GPUMlib` will never go into areas of deep learning since there are many specialized frameworks for that such as `Torch`, `Caffe`, `cuDNN`, `NervanaSys` etc.

1.11.3 cuda-convnet(2)

It's a single purpose library implementing convolutional networks in NVIDIA CUDA framework. It uses pure C for the model but the data feeder is written in Python. In my opinion the documentation of data loading is insufficient so it is hard to start with the library. After initial struggles one can find out that there are three evolutions of the implementation. Every evolution contain many important optimizations by Alex Krizhevsky who won many times an international competition with this implementation.

This library is used by important CUDA machine learning frameworks such as `Caffe` and Torch. There are two important sources of code. First, the original by Alex Krizhevsky[4] and its improved version[5]. There is a fork[6] promoted by NVIDIA. It enhances the original with `dropout` layer and many standard CUDA libraries.

1.11.4 NVIDIA cuDNN

The NVIDIA CUDA Deep Neural Network library (`cuDNN`) is a GPU accelerated library of primitives for deep neural networks. It emphasizes performance, ease-of-use, and low memory overhead. `cuDNN` is designed to be integrated into higher-level machine learning frameworks, such as the popular `Caffe`, `Theano`, or `Torch` software frameworks [24].

NVIDIA claims that `cuDNN` accelerate `Caffe` convolutional layer by factor of 1.2 – 3. They give example of `AlexNet` Layer 2 where the forward phase had: 1.9x faster convolution, 2.7x faster pooling using `cuDNN` over their native implementation [25].

1.11.5 Torch

An academic framework built in Switzerland. The underlying concept is LuaJIT which is compiled into C/CUDA code in the end. This framework contains many machine learning functions and models and therefore it is widely used by professionals such as Yann LeCun. The framework implements models such as neural networks, deep convolution networks, energy-based models and

[4] `https://code.google.com/p/cuda-convnet/`
[5] `https://code.google.com/p/cuda-convnet2/`
[6] `https://github.com/dnouri/cuda-convnet`

numerical optimization routines just to name a few. However, a substantial amount of models are still the original implementations in C/C++ with light wrapper around them. Torch is mainly used by Facebook for their own AI research with NN implementations such as `fbnn`, `fbcunn` and `fbfft`.

1.11.6 Theano

It is a Python framework built on top of `NumPy` and `cuDNN`. Theano is an open source project primarily developed by a machine learning group at the Université de Montréal. Its primary aim is to compile mathematical expressions in Python into machine code. This combines the convenience of NumPy's syntax with the speed of optimized native machine language [26].

The most famous Theano user is the team behind site `deeplearning.org`[7] which promotes deep learning since 2006. We decided not to use it for its python bindings. If we are going to implement a parallel model we need a fine control over memory and how the parallelism is done. We would need to change the core of Theano rather than just tweaking bits of code.

1.11.7 Caffe

Caffe provides multimedia scientists and practitioners with a clean and modifiable framework for state-of-the-art deep learning algorithms and a collection of reference models. The framework is a BSD-licensed C++ library with Python and MATLAB bindings for training and deploying general-purpose convolutional neural networks and other deep models efficiently on commodity architectures. Caffe is maintained and developed by the Berkeley Vision and Learning Center (BVLC) with the help of an active community of contributors on GitHub [27].

Many machine learning applications are based on this framework. It is not because of its speed, being overall average. It is because of its nice design and ease of incorporation into existing applications. We can choose few examples as NVIDIA DIGITS or "Brain simulator" by Keen Software House. NVIDIA is surprisingly active in development of this framework.

Caffe started as a PhD thesis and it was released to public in September 2014. It claims to have high performance convolution and that was its main focus – convolutional neural networks. The latest update (release 2) brought fully-connected and recurrent layers, contrast normalization and many improvements to convolutional networks. The following update has promised multi-GPU support (backed by NVIDIA itself) which makes Caffe a safe bet for the future.

Even though it might seem that current lack of support for multi GPU is a deal breaker it is not so. The only disadvantage is that if we want to use many GPUs we need to write the application in distributed manner. Or even

[7] `http://deeplearning.net`

parallel implementation is sufficient. The code on CPU side will get more complicated but the result will be very similar as if it ran on many GPUs natively.

1.11.8 NVIDIA DIGITS

The NVIDIA Deep Learning GPU Training System (DIGITS) is a software built on top of `cuDNN` and `Caffe` and released to public on 14th of March 2015. It provides user interface for controlling, monitoring and analysing model's runtime and structure. The biggest advantage is realtime monitoring of loss function and accuracy, which makes it perfect for testing out new network architectures. The other useful feature is visualisation of intermediate layers. This feature is mostly intended for convolutional layers but can be useful in any type of layer.

The key features are

- Visualize DNN topology and how training data activates your network
- Manage training of many DNNs in parallel on multi-GPU systems
- Simple setup and launch
- Import a wide variety of image formats and sources
- Monitor network training in real-time
- Open source, so DIGITS can be customized and extended as needed

On the other hand DIGITS is strongly bound to image data and therefore it is of no use for us. NVIDIA claims to support generic type of data in the future but is has not been released during writing of this thesis.

1.11.8.1 DAMEWARE Application

DAME is a data mining infrastructure specialized for mining massive data sets. It offers complete toolset of machine learning algorithms. The main entry point to the application is an web application called Web Application REsource of DAME (DAMEWARE[8]). It can handle common astronomical data formats such as FITS and VOTable files. In release 1.0, the first parallel implementation of a machine learning model, fast multi-layer perceptron (FMLPGA), was added. The implementation is based on the GPGPU+CUDA environment, enabling a speedup of about 8 times. This algorithm extended already broad collection of machine algorithms which contains

- MLP + Back Propagation
- MLP + Quasi Newton
- MLP-LEMON (Levenberg-Marquardt Optimization Network) for classification/regression
- Random Forest – for multivariate classification and regression
- Support Vector Machines (SVM)
- K-means model (through KNIME engine)

[8] `http://dame.dsf.unina.it/dameware.html`

- CSOM (Clustering Self Organizing Feature Map) – Unsupervised Image segmentation
- GSOM (Generic Self Organizing Feature Map) – (Clustering) based on customized SOFM model
- SOM (feature selection) + autoPostProcessing, K-means, Two Winners Linkage (TWL) or U-matrix with Connected Components (UmatCC)
- ESOM Evolving SOM for clustering
- Probabilistic Principal Surfaces (PPS) – feature selection

The application itself is not freely installable because it is bound to cluster architecture on which it runs. There are, however, possibilities of own extensions via pluggable user data mining models known as `dmplugin` client application. We aim to create such a plug-in from our own implementation of one or two models. This would add the second massively parallel algorithm.

We were in contact with its key developer Dr. Massimo Brescia who was improving the application so it fit our needs. During the period of writing this thesis DAME's import procedure was enhanced so it accepts more than 500 columns. Furthermore the manual for MLP model has been extended with point 18 which is necessary for successful usage of the model[9]. There are ongoing improvements such as realtime error output and few more.

1.12 Performance measurements

First we would like to point out some performance metrics measured by NVIDIA itself. We expect those metrics to be slightly optimistic but still providing a good overview. The tested hardware was K40c which has following configuration.

GPU	2880 cores, clocks from 745 MHz up to 875 MHz (boost)
Memory	12 GB of GDDR5 3.0 GHz, 288 GB/s, 384-bit interface
Socket	PCI-E Gen3., 8 GT/s, 985 MB/s per line (total 16 lines)

CUDA Toolkit comes together with optimized libraries for mathematical operations. Table 1.2 shows computational power and compare performance of the libraries. The tests of CUDA libraries in version 6.5 were performed on graphic card NVIDIA Tesla K40c, ECC ON and input data were divided into 28M– 33MB elements, input and output data were already on the device and we exclude time necessary to create "plans". The CPU used for the test was Intel IvyBridge single socket 12-core E5-2697 v2 @ 2.70GHz and MKL library in version 11.0.4.

Facebook did performance analysis of publicly available implementations of convolutional networks. The performance report is available online[28] and we enclose their results here for relative comparison of frameworks.

[9] `http://dame.dsf.unina.it/dameware.html#mlpqnaman`

library	problem	single precision	double precision
cuFFT	1D,complex	700 GFLOPS	300 GFLOPS
cuBLAS	gemm	3000 GFLOPS	1200 GFLOPS
MKL	gemm	-	200 GFLOPS

Figure 1.2: Performance metrics by NVIDIA

implementation	**time**	**forward**	**backward**
NervanaSys-16	97	30	67
NervanaSys-32	109	31	78
fbfft	136	45	91
cudaconvnet2	177	42	135
CuDNN (R2)	231	70	161
Caffe (native)	324	121	203
Torch-7 (native)	342	132	210

Figure 1.3: Showcase of relative performance of NN implementations

Our testing device, provided by Astronomical Institute of the Academy of Sciences of the Czech republic, is an one-blade server with GeForce GTX 980 installed.

GPU	2048 cores, clocks from 1126 MHz up to 1216 MHz (boost)
Memory	4 GB of GDDR5 3.0 GHz, 224 GB/s, 256-bit interface
Socket	PCI-E Gen3., 8 GT/s, 985 MB/s per line (16 in lines in total)

Chapter 2

Scalability analysis

The only two possibilities in scalability are model parallelism and data parallelism. All other options result in higher throughput but they won't scale the computation.

- *model parallelism* – different workers train different parts of the model
- *data parallelism* – different workers train on different data examples

In model parallelism, whenever the model part (subset of neuron activities) trained by one worker requires output from a model part trained by another worker, the two workers must synchronize. In contrast, in data parallelism the workers must synchronize model parameters (or parameter gradients) to ensure that they are training a consistent model [4].

2.1 Terms and facts

Parallelism is when the same part of a code is executed by more processing units on one logical piece of data at the same time with implicit synchronization of the data. Parallel processing can be achieved using threads (MPI), OpenMP or natively on GPU. Shared memory is the most desirable state but in case of OpenMP we can work on distributed memory as well. We can say that parallelism is a special case of concurrency.

Concurrency (distributed computing) is the case of one or more programs processing one logical piece of data but with explicit synchronization by messages passing. There is no implicitly shared memory, everything has to be explicit. Indeed, concurrency can be running the same code for logically separated data or running two different codes because it is still executing two tasks at once but without any synchronization, which is not a requirement for concurrency. The most common architecture is multiple services running in distributed manner and exchanging information over (UNIX) sockets. The

approach that is most used is MapReduce framework and its implementation by `Apache Hadoop`.

Neural Networks is a sequence of (non)linear layers connected to their successor. The connection has to be both ways because we use forward connection in classification phase and backward connection in the learning phase. The connection transmits value and multiplies it with *a weight* associated to the connection. There are many types of neurons, layers and weights-update strategies which we are going to be described briefly in the following paragraphs.

Deep Neural Network is the most recent form of neural networks. The term deep neural network is vaguely defined but for our purposes we will use the definition: "Deep neural network is a sequence of layers where there are at least two hidden layers of a different type". Deep networks introduce specialized layers such as convolutional layer, dropout layer and pooling layers. Since the best known new-type layer is the convolutional layer, then deep networks are quite often called convolutional networks. Those two terms have vaguely the same meaning. The old-type multilayer neural network consisted of neurons with non-linear activation functions connected by weighted connections to other neurons. Deep networks consider a layer as the smallest computational unit and therefore connections between neurons in the old sense do not exist. Any computation has to be done inside a layer (even weighting of inputs) and therefore connections between layers are direct and without weights.

Fully connected layer is a layer implementing weighted connections in deep neural networks. It creates a fully connected bipartite graph between its input and output layers. Since every layer has to wait for the previous

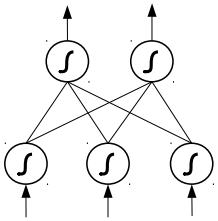


Figure 2.1: Fully connected bipartite graph

one in both passes (forward or backward) the parallelization is hard. There were many experiments to overcome the synchronization problem but non of them was successful. The only working solution is to feed the data in batches. Parallelization of the model is impossible because every neuron has to communicate with all neurons from the previous layer. We have to play

with implementation to make the communication as less resource-consuming as possible.

Convolutional layer resembles retina in human's eye and therefore is mainly used in image recognition. It can deal with rotation and translation of features in images because they are searched for separately. The main idea is to create small classifiers for every feature and then tile this classifier all over the image in a way that it doesn't matter where a learned feature appears in the image. We can even rotate the small classifiers in order to recognize not just translation but even rotation. That would indeed lead to greater complexity of the output. Therefore *pooling layers* were introduced. We describe them in the following section 2.1. In order to teach the small classifiers, a technique called *weight-sharing* is used.

The convolution itself is defined in one-dimensional space by Formula 2.1. The formula defines output of a convolution $o[n]$ as a multiplication of a vector $f[n]$ with a convolutional vector g of size M where n is a spatial index and $f[n]$ a slice of data from vector f centred around the index n and having the same size as g.

$$o[n] = f[n] * g = \sum_{u=0}^{M} f[n + (u - M/2)]g[u] \tag{2.1}$$

For example, when a convolutional network is trying to recognize faces, it looks for eyes, nose, mouth and other facial features independently and then combine findings of those separate elements. The combination procedure is usually implemented as max-pooling layer. In our example in figure 2.2 the network should obtain neural triggers for two eyes, nose and mouth which are positioned as they form a face. If it finds such an alignment it triggers a neuron for face at the position, which is usually implemented by fully-connected layer. Convolutional layer's features can be used in many ways. It does not need to be separate elements but for example different lighting or angles of an object.

To give a better image of how the convolutional layer works we have to consider that one feature is looked up in the whole input image. For example, if we have an input image of size $N \times N$ and a feature of size $k \times k$, the resulting feature map will be of size $(N - k + 1) \times (N - k + 1)$ in case of stride $= 1$. Every feature produces one *feature map* which has to be connected further so it creates many parallel networks.

Therefore there are many parallel implementations from which the best know is `cuda-convnet2` by Alex Krizhevsky. Parallelization of convolutional layer is easier than fully- connected layer because we train the same layer with different data. There is necessary communication when the layer synchronizes weights which is negligible compared to the computation needed for obtaining the weights.

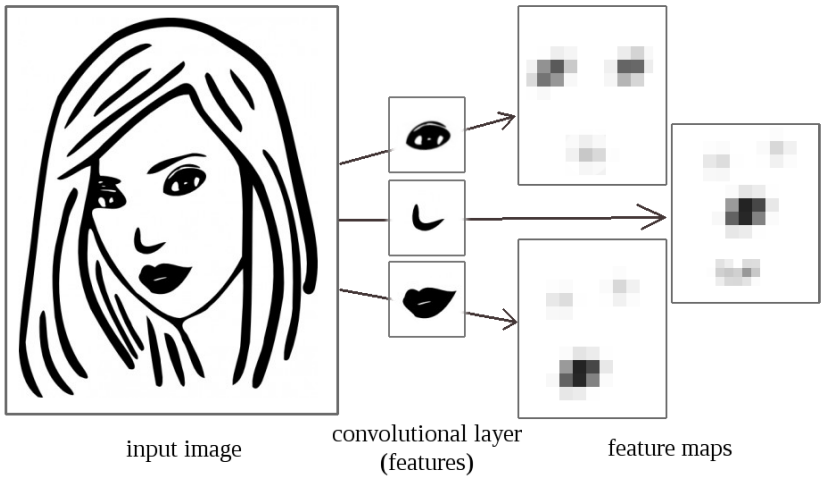


Figure 2.2: Example of possible features and their maps

The convolution produces massive amount of new data. Therefore *stride* was introduced to tackle this problem. Stride controls in what distant steps is the convolution applied. For example if the stride is equal to the feature size then two following convolutions will not overlap. If the stride is equal to 1 (which by default it is) then the convolution is moved by one pixel each time thus overlapping as many times as the feature size.

Pooling layer is a simple layer which takes maximal/average/minimal value from all neurons in a block. The main purpose of this layer is to denoise the data and scale them down. Pooling is mostly used after convolution because convolution multiply data by the factor of number of features. The variables in settings of this layer are kernel p and stride s. Kernel defines the pooled block size and stride the shift of kernel per step. The inputting data are scaled down from size (X, Y) to $((X - p)/s, (Y - p)/s)$.

Dropout layer randomly prevents some values from further propagation. It is mainly an enhancement for learning and should not be used in the recognition phase. The purpose is to avoid overfitting of the trained model on trained data.

2.2 Parallelism by Alex Krizhevsky

A good example of optimizing throughput is usage of batch processing. This approach together with very clever model parallelization is implemented in `cuda-convnet2` by Alex Krizhevsky who was leading benchmarks of convolutional networks for a long time. The optimizations, he is using, are described

in following sections. Those optimizations are not included directly in any other implementation of neural networks (such as NVIDIA `cuDNN` or `Caffe`) but the `cuda-convnet2` itself is a part of `Theano` and `Caffe` frameworks so one can benefit of that indirectly.

2.2.1 Fully connected layer

This layer is indeed the most used and the most demanding for communication. During learning and recall phase it was observed that only $5 - 10\%$ of the time is actually spent on computing, while the rest of the time is used on communicating the results. In order to scale the model we will count with K workers (let's say K=32) and batch processing of data in batches of size N (e.g. N=128). Therefore we rely on model parallelism here – all workers process the same batch of the data on different parts of the network. When the outcome (either activation value or gradient) is computed, there are two ways of communicating it.

- One of the workers sends its last-stage (computed on a batch of 128 examples) to all other workers. All workers then compute the outcomes from the broadcasted last-stage and begin to backpropagate their gradients for these 128 examples (in case of training) back to the sender. *In parallel with this computation*, the next worker sends its last-stage. The main consequence of this is that much (i.e. $(K-1)/K$) of the communication can be hidden – it can be done in parallel with the computation of the fully-connected layers. This seems fantastic, because this is by far the most significant communication in the network.
- This solution is very similar to the previous one. Each worker has computed the last-stage for 128 examples. This 128-example batch was assembled from 128/K examples contributed by each worker, so to distribute the gradients correctly we must reverse this operation. The rest proceeds as in the previous solution. This one's advantage is that the communication- to-computation ratio is constant in K. In the previous solution it was proportional to K therefore it was always bottlenecked by the outbound bandwidth of *the one* worker that had to send data at a given "step". This solution enables the use of many workers for this task. This is a major advantage for large K.

We are free either to update the fully-connected weights during each of the backward passes, or to accumulate a gradient and then update the entire net's weights after the final backward pass.

2.2.2 Convolutional layer

This optimization is very similar to the one right above but with modifications to fit the more independent nature of convolution.

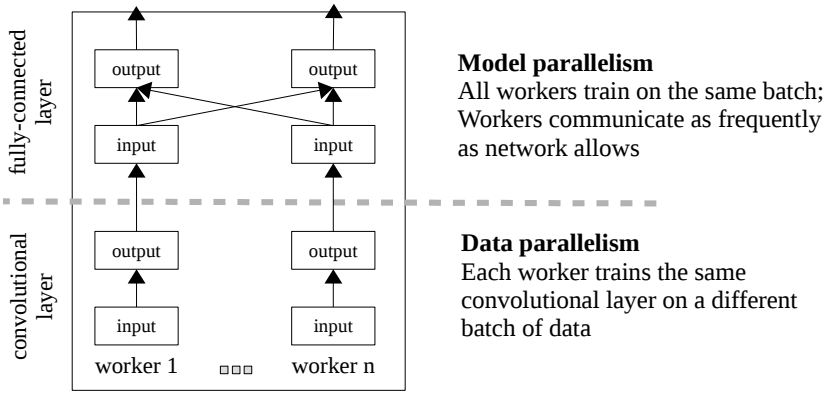


Figure 2.3: Illustration of a convolutional layer meeting fully-connected layers

When training convolutional nets in parallel, we rely heavily on data parallelism because convolutional layers communicate their intermediate outcomes straight to one area instead of to all neurons in the following layer as the fully-connected ones. Data parallelism means that we split the input data between workers so that every worker has different data but trains the same layer. Parallelization was done mainly by Krizhevsky in his papers [29] and [4].

Again we will count with parallelization using K workers and bath processing of size N data per batch. The ideas come from the same source as in the fully-connected layer optimization. The convolution layer is easier to parallelize because there is no inter-communication during computation, only during weight update phase. The workers must also synchronize the weights (or weight gradients) with one another so that:

1. Each worker is designated 1/Kth of the gradient matrix to synchronize.

2. Each worker accumulates the corresponding $1/K$th of the gradient from every other worker.

3. Each worker broadcasts this accumulated $1/K$th of the gradient to every other worker.

2.2.3 Softmax layer

A softmax layer normalizes its output values accoding to the maximal value. The definition of softmax function is given in equation 2.2 and shows that in order to compute the output value o at index j, the softmax function accesses

all its input data x thus creating a synchronization point.

$$o_j(x_j) = e^{x_j} / \sum_{i}^{N} e^{x_i} \tag{2.2}$$

To overcome the synchronization penalty, N independent logistic units, specially trained to minimize cross-entropy, can be used to replace this layer. This cost function performs equivalently to multinomial logistic regression but it is easier to parallelize, because it does not require a normalization across classes. The synchronization penalty is not an important bottleneck with only 1000 classes, but with tens of thousands of classes, the cost of normalization becomes noticeable.

2.2.4 Asynchronous Stochastic Gradient Descent

Asynchronous SGD is an example of data parallelism. The original (non-stochastic) gradient descent is a way to update parameters of a classifier in such a way that it minimizes the error produced by a network. The error function is also called loss (or energy) function $\bigtriangledown E(W_t)$ where W_t are network parameters at time t. The loss function is usually computed by a specialized layer such as `EuclideanLoss` layer. Equation 2.3 describes the update of the parameters with respect to the values of the parameters from the previous iteration and to the learning rate α.

$$W_{(t+1)} = W_t - \alpha \bigtriangledown E(W_t) \tag{2.3}$$

The loss function is computationally demanding and therefore we rather approximate it from a random sample of training data. Suppose we select N random samples of n elements each. The stochastic gradient descent is computed as

$$W_{(t+1)} = W_t - \alpha \bigtriangledown E_n(W_t) \tag{2.4}$$

$$\text{where} E(W_t) = \sum_{n=1}^{N} E_n(W_t) \tag{2.5}$$

The asynchronism can be introduced by split those samples into mini-batches and run those in parallel.

Chapter 3

Design

We have designed a general purpose classifier using CUDA technology. First, we planned to implement the classifier from scratch because there were no publicly available implementations of parallel deep neural networks. Therefore sections 3.6 and 3.5 contain brief description of CUDA and thorough description of many types of neurons and layers. Due to current fast progress in the field of deep learning a new thesis emerged and brought an efficient implementation of cNN on GPUs called Caffe library, developed by Computer Vision Group, Berkeley University, California[27]. Their design is very elegant and we will describe it the following sections. We decided to build our classifier on top of Caffe as a mature technology, recently adapted by NVIDIA, with bright future.

Our classifier is required to read various file formats and support many types of input data. It ought to be easily deployable in cloud and scheduled with `VO-Cloud` (section 3.3) job scheduler. Another important feature is a flexible definition of neural networks. Every model has to be adjusted to fit its input data so the classifier is also required to have either fully automatic adoption to inputting data or really simple manual configuration.

3.1 Used technologies

One requirement is implementation using NVIDIA CUDA toolkit which follows standard C99. `C++11` was chosen as the application language. It offers modern features, speed and direct communication with CUDA routines so it is possible to fine-tune the resulting classifier.

The solution should be platform independent. Since our technological demands are very humble, we can get by with boost and standard libraries, which will be merged together in `C++14` specification.

The build system should be platform agnostic too. We chosen CMake partially because it is used by Caffe itself. We even introduced the same build flags as Caffe has, so the build configuration of both products is the same.

3.2 Application architecture

We need to be able to seamlessly access many input formats such as CSV, VO-tables and FITs files. Since we are using parts of Caffe library we are designing a thin wrapper which will make deployment in cloud possible. In order to follow "loose coupling and high cohesion" we split application into three modules where every module exports a factory function.

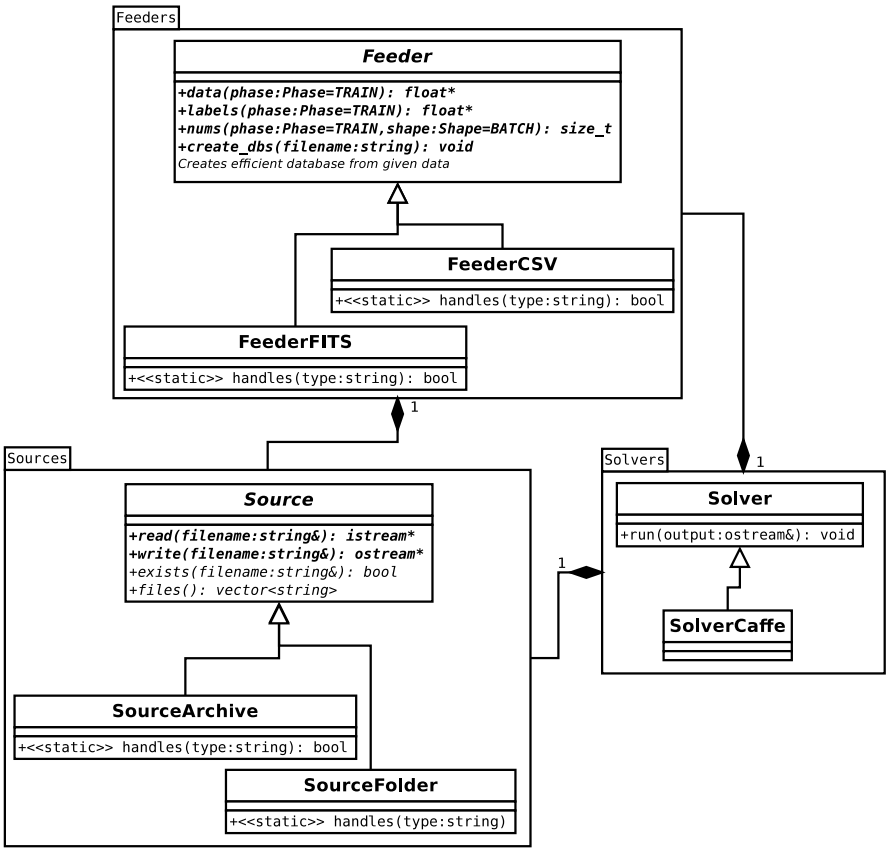


Figure 3.1: Class diagram of future application

This architecture gives us the opportunity to easily extend input routines by adding for example new source for HDFS if the application will be successful.

The class `Source` does not need to be optimised because it is purely an utility class to unify all possible sources. The high demands will be put on Feeder and Solver. Feeder has to be able to handle huge input files (possibly terabytes). Solver has to be able to efficiently update model's parameters and therefore implemented at least partially in CUDA.

3.3 VO-Cloud

The final application is expected to work inside VO-Cloud. That dictates usage of JSON configuration files and eventually implementing a deployment script which will register our service into a master server.

VO-Cloud is a scheduling server with web interface currently being developed by a small team of students of informatics lead by Dr. Petr Škoda at Ondřejov observatory[30]. It aims to become a whole processing pipeline for astronomical spectra obtained from external archives using protocols specified by Virtual Observatory standards. At the end of the transformation pipeline, there will be a choice to select a machine learning algorithm for building a prediction model.

3.4 Neural Network Architecture

Learning rate decay is a useful feature against overfitting of a network. It lowers the learning rate with time. There are many ways how to implement it but the simplest, which we are using, is called `step LR decay` meaning that in every N steps the learning rate is multiplied by a factor $\gamma < 1$.

Regularisation (a.k.a. weight decay ξ) is another technique which prevents model from overfitting by lowering complexity of layer's weights. The main idea is similar to Occam's razor in adding a constraint in order to cut out unwanted complexity. The regularisation is performed by adding transformed weights to the error function as shown in equation 3.1. Thereafter the model tends to keep its weights low because they are increasing the error function.

$$\vec{E_i} = \vec{E_i} + \xi * ||\vec{w}|| \tag{3.1}$$

$$\text{where} ||\vec{w}|| = \begin{cases} L1 & ||\vec{w}|| = |w_i| \\ L2 & ||\vec{w}|| = w_i^2 \end{cases}$$

where i denotes i-th layer, $\vec{E_i}$ vector of partial errors.

Learning momentum helps to overcome local minima in loss function. The network needs to keep track of historical changes to weights and take those into account with strength $s \in (0, 1)$.

Batch processing is a technique for speeding up and possibly parallelizing the whole network. When a batch of data is sent to the network, the parameters update happens only when the batch is finished propagating. The gradient is computed for every datum from the batch but the application of

the gradient happens only once. That allows the gradients to be computed asynchronously. More importantly, some layers can operate on one datum identically as on a batch of data because there are no dependencies between a value and the neighbouring values. For example sigmoid, hyperbolic or ReLU transformation are simply applying their own function on a value no matter what the value means. It can even be implemented as a in-place transformation.

Weights initialisation is commonly known to not be very important. The default form of weights initialisation is by using a random numbers generator. If we imagine weights as a matrix then its eigenvalues has to be close to 1. If the eigenvalues differs greatly from 1 then the information from input will either disappear or exponentially increase leaving nothing but noise [31].

Error contribution is a solution to the biggest problem of deep neural networks – gradient decay. Gradient decay describes the fact that error, which is back-propagated through layers, slowly disappears. It is caused by dividing the error between many neurons in every layer. Caffe indeed has implemented a solution to this problem that any layer can act as a *loss contributor* with a certain ratio. Hence it is possible to train even very deep networks and ensure that the gradient will not decay in the first few bottom layers. In the configuration file, it is controlled by the parameter *loss_weight*$\in [0, 1]$.

3.5 Neurons

Neuron is an object holding *activation function* and input and output value. The types of neurons do not distinguish just by their activation function but, as a consequence, by the method of learning and therefore capability of classifying.

3.5.0.1 Terms

Indices i, j refers to two subsequent neurons where j is the later one. Index j can be viewed as an output neuron and i as a neuron in the hidden layer.
t is the expected output of a neuron.
y is an output of a neuron (result of it's activation function).
x is an input of a neuron from *one* other neuron therefore x_j is equal to y_i.
$w_{ik,jl}$ is a weight of a connection which transports value y_i from the k-th neuron to the l-th neuron in the higher level

z_{jl} is a total input to the l-th neuron given by

$$\begin{aligned} z_{jl} &= \sum_{n \in N} w_{in,jl} y_{in} \\ &= \sum_{n \in N} w_{in,jl} x_{jl} \end{aligned}$$

The first kind of artificial neuron is known as McCulloch-Pitts neuron developed around 1943. It takes into account bias b when producing its output z as

$$z = w_{0n} b + \sum_{n \in N} w_{in,jl} x_{jn}$$

and has a non-differentiable binary activation function

$$y = \begin{cases} 0 & if z < 0 \\ 1 & \text{otherwise} \end{cases}$$

Learning method for this type of neuron is updating weights by input values.

$$\Delta \vec{w} = \text{sgn}(t - y) * \vec{x}$$

The learning method guarantees that the weights are getting always closer to the desired weights. The McCulloch-Pitts neurons are still in use because of how fast they can learn even with huge amount of inputs. Therefore companies like Google are still using them.

Perceptron It is a neuron with linear, differentiable activation function which is simple weighted sum of it's inputs

$$y = \sum_{n \in N} \vec{w}^T \vec{x} \tag{3.2}$$

with lower bound .

The learning method is based on minimising *residual error* $E = t - y$ which uses input value to update weights

$$\Delta w_i = \epsilon \, x_i (t - y) \tag{3.3}$$

where $\epsilon << 1$ is a learning rate.

The formula 3.3 is a specific case of the generic backpropagation formula. The learning method guarantees that the output is getting always closer to the desired output. The expressibility of a network compounded by linear neurons is unfortunately very limited. It can't do more than a linear regression model since the whole network can be described as a linear combination of inputs.

Non-linear neurons They use weighted sum as input to their activation function but the result is everything but linear. The most common functions are

- **Logistic (a.k.a. Sigmoid)** $o(x) = (1 - e^{-x})^{-1}$
  is the most common neuron in all networks. There are two disadvantages compared to the others. First is that it is computationally demanding (learning and evaluating) and the second is that it might become easily saturated.

- **Hyperbolic** $o(x) = \tanh(x)$
  was introduced because it learns faster in certain cases when the data are really distinct (for example contrasting images).

- **Rectified Linear Unit (ReLU)** $o(x) = max(0, x)$
  were introduced because of their performance. They are several times faster than ordinary logistics neuron and yet bring similar nonlinearity into the system. Those units were introduced by Nair and Hinton [32]

- **Softmax** $o(x)_j = e^{x_j} * (\sum_i e^{x_i})^{-1}$
  is mostly used as the last layer for the final output. The main reason for that is that since it is defined as a sum the derivative of this expression is complicated

- **Dropout** blocks some values from random neurons so they won't contribute to the final result. It is used to mitigate over-fitting and it is getting on popularity.

For better reasoning about the nonlinear units we add a plot of two the most common ones.

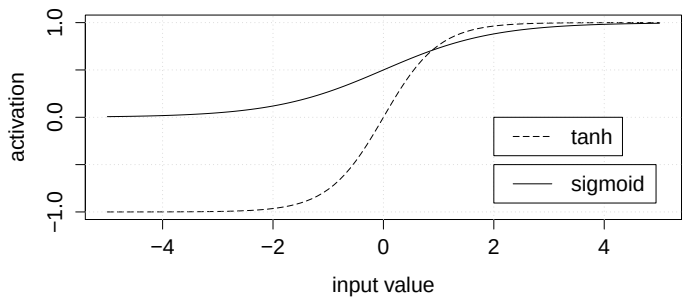


Figure 3.2: Plot of sigmoid and tanh activations

3.6 CUDA

Architectures of CUDA enabled cards changes with time. NVIDIA introduced 4 architectures so far: Tesla (2008), Fermi (2010), Kepler (2012), Maxwell (2014). Every architecture brought a new technology; Tesla – atomic operations, Fermi – synchronisation routines and 3D grid of thread blocks, Kepler – unified memory programming (pinned memory with automatic management from the side of GPU which, if necessary, transfers the memory back and forth), Maxwell – Dynamic Parallelism (nested threads, when a device thread can launch new threads so the thread grid can become heterogeneous).

Threads, Warps and Blocks are computational units which are controlled by the programmer. The biggest and physical computational unit of device is the *streaming multiprocessor* (SM) which accommodates usually few hundreds of cores and possesses small memory called *shared memory*. The next computational unit in size is a block, which is a virtual unit. There are up to 16 blocks per SM. A programmer can demand his code to run on up to $2^{16} - 1$ blocks. The smallest units are threads. There is maximum of 1024 threads per block. Threads are scheduled in warps – 32 threads together by a warp-scheduler. Warp share instruction counter so if one thread takes different execution path then it has to NOP through instructions when it does not do anything. Until all 32 threads has the same execution path, the performance is optimal. Since Fermi architecture, there are 2 (and further architectures have more) warp schedulers without any dependency checking between two running warps.

Global memory is a built-in RAM in the graphic card. Fermi architecture offers up to 6 GB of RAM with 6 lines (64b each) for reading. It is the main and largest memory on the device. The trade-off is the speed of the memory because it is the slowest memory on the device.

Shared memory is the own memory of each core which is used to make intermediate load/store operations faster. Since the third generation of NVIDIA devices (Fermi architecture), the shared memory size is 64KB per SM. This amount of memory is divided between running blocks so the actual available size can be 4 – 32 times smaller. The 64KB can be split in ration 48-16 between actual shared memory and L1 cache.

Constant memory read only memory of size 64KB. Read operation from this memory can be broadcast to group of 16 threads. We should try to serialise any classification model into this memory. Also since it is constant memory it is heavily cached.

Local memory is on per-thread level. The size ranges up to 512kB.

Streaming Two independent queues that stack memory and computational operations. We will heavily use this technology because the copy and compute operations might take about the same time. In order to be able to use this feature of graphical cards we need to have access to pinned memory so we can use direct memory access.

3.7 Google Protocol Buffer

Google's protocol buffers are serialisation format with bindings to many programming languages. The serialised data can be stored as textual or binary files known as `protobuf` files. Textual representation of the data is very similar to JSON except it is statically typed and all relations, attributes and compound types have to be declared in a definition file. The main advantage is that parsing files according to a precompiled definition is simple thus fast. The precompiled definition is what made efficient binary format possible. As the name of the library suggests, it was primarily designed for protocol definitions and therefore the basic class is called `Message`. The whole documentation is available online[10]. Currently supported languages are C++, Go and Python.

Caffe takes advantage of protobuf's textual and binary formats. The textual files are used as configuration files. We follow this concept and therefore our classifier requires every model and solver to be defined in this standard format. We keep those configurations separate from our application's configuration so it stays framework agnostic. In order to be fully framework agnostic we have moved some options (such as running mode and numbers of iterations) from Caffe solver definition to out application configuration. The binary files are used for serialization of models and solvers from which we use only the model serialization.

3.8 Caffe

Caffe is primarily a library which provides implementation of many different layers of neural network in two codes – CPU and GPU. The biggest advantage is that switching between those two implementations is done in runtime. No recompilation is needed. Considering that this is such a huge advantage, this framework was hence selected for further use. Caffe indeed comes with a few types of solvers which can be used for training or testing but they lack the ability to classify unknown data. A third state would be needed in addition to TRAIN and TEST. We call the missing phase GUESS internally within our application.

[10] `https://developers.google.com/protocol-buffers/`

The following section describes the implementation details of the Caffe framework. It is essential that those details are understood in order to use and continue in developing of our classifier.

3.8.1 Caffe Blob and Synced Memory

Synced Memory is the bedrock of Caffe's design. It provides seamless manipulation of GPU and CPU memory by implementing lazy synchronization so it mitigates any unnecessary memory movements until the memory is not actually needed.

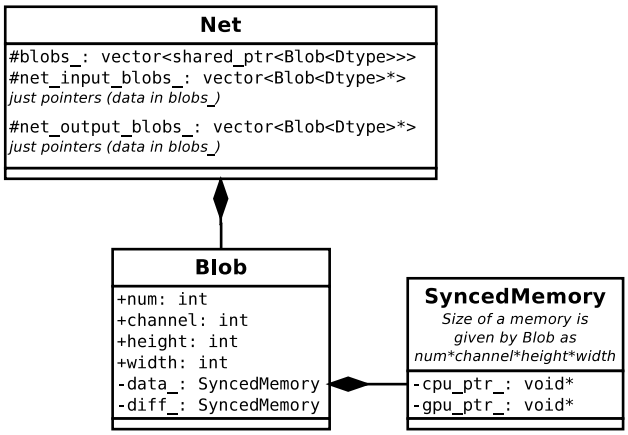


Figure 3.3: UML class diagram of Net - Memory relation

Figure 3.3 shows how `Blob` and `SyncedMemory` are positioned in Caffe's design. As we mentioned earlier, every blob is defined by a quadruple *(num, channel, height, width)* where every element is called an *axis* so we can think about it as a 4 dimensional object. The underlying implementation is a simple one dimensional array where data are continuous in width and therefore a position of an element is computed as

$$((n * \text{channels} + c) * \text{height} + h) * \text{width} + w \tag{3.4}$$

where the one-letter variables denote the position of the element and the words are constants for a given Blob.

The primary aim of blobs is to hold image data. For our data we are using blobs of shape $(N, 1, 1, W)$ where W is maximal width of spectra and N is number of spectra per training batch. N should be optimized such as it fits into memory of all GPUs. The maximal memory consumption on GPUs is defined by a constant in the compilation phase of Caffe.

3.8.2 Caffe Layers

We will start from memory handling. Every layer keeps its *parameters* (as the Caffe developers call it) in caffe::Blobs within the layer itself. Those parameters are stored in a vector called `blobs_` but a developer rarely touches them. This is the only thing which gets serialized and restored from a layer so it has appropriate definition in a `proto` file.

Even though there are "bottom" and "top" blobs mentioned in the layer's definition, the layer does not own them. Those blobs have to have unique names and that is set within layer definition in config file by attributes *bottom* and *top*. The connection between layers is then deduced from the blob names. If a layer B defines its bottom blob with the same name as some other layer A defined its top blob, then a connection A-¿B is created. Moreover, the result of a layer can be broadcast to many layers. In the implementation, the layers do not posses their top and bottom blobs. Those blobs are owned by the wrapping `caffe::Net` object and passed to layers when doing forward or backward pass.

There are no weighted connections between layers as one would expect. Caffe's solution is that the full connection is implemented as a special layer `caffe::InnerProductLayer`. This solution greatly simplifies the implementation of generic layers and wrapping `caffe::Net` class as well.

When a layer is instantiated by a `caffe::Net` object it reports its top and bottom blob names. Those blobs are then allocated in the Net and referenced as possible top and bottom blobs. If there is no bottom blob that would make use of a top blob then the top blob is marked as *output_blobs* and can be accessed via function `caffe::Net::output_blobs`.

The network structure is usually described by a graph of layers and data. Here we show the simplest neural network defined using Caffe.

The label object which goes directly from input layer to the output layers is a `blob` of size `(batch_size, 1, 1, 1)` therefore the label is put into one neuron as it is and all the output layers will encode it to `one-hot` encoding. On the contrary, which does make sense, the output of a network is expected to be a blob of size $(N, C, 1, 1)$ where N is the *batch_size* and C is a number of distinct classes.

The labels and data are produced by `<Something>Data` layers and are expected to be in separate Blobs. By implementation the ordering matters here – data first, labels last. Having more blobs in forward and forward functions is made possible by the parameters to those functions which are vector of blobs.

MemoryData layer is one of the possible input layer of our application. It is expected to have 2 *top blobs* as any other *¡Something¿DataLayer*. The setup phase expect to have `MemoryDataParameter` protocol buffer message as an input. This message doesn't do anything but specify the shape `(batch_size,`

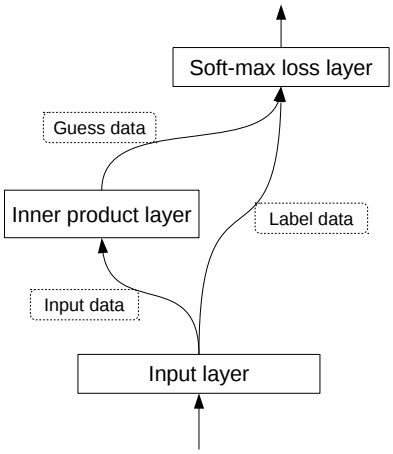


Figure 3.4: Graph of very simple neural network defined using Caffe

`channels, height, width)`. We spare the user of specifying those and we compute them automatically.

Internally the label Blob has shape $(N, 1, 1, 1)$ where N is number of samples for the neural network. The important note is that *batch_size* which we know from the protobuf configuration file for neural network (not solver) finetunes how much data will get transferred to GPU memory. Therefore, there is a crucial condition $\mod(N, \text{batch}_\text{s}\text{ize}) == 0$ in the official version which can not be omitted. We have implemented shrinkage of the source data in the last batch but it turned out that convolutional layers require the batch size to be constant.

InnerProduct Layer is the name for what widely known as fully connected layer. In this and convolutional layer we can define a weight initialization algorithm which has been proven as significant part in constructing neural networks. The reason is that we have to initialize weights in such manner that no information (input) gets amplified or muted. Therefore if we take the weights as matrix, then its eigenvalues should be close to 1. This layer is often the last one in the prediction part because it transforms any shape into `(batch_size, num_outputs, 1, 1)` where `num_outputs` is a parameter of the network. If the layers is the one providing final prediction then `num_outputs` has to be equal to the number of classes in the prediction.

Accuracy Layer is intended as output layer. It expects two bottom blobs – label and 1hot encoded prediction. Suppose we have C distinct classes therefore the prediction has to be in shape of `(batch_size, C, 1, 1)`. The

layer takes optional parameter *top-k* so it returns a success if the correct label appears in the *top-k* results.

3.8.3 Caffe Net

The network object is a container for layers. It possesses the data flowing through the network and moves them from one layer to another. Moreover, it offers interface to access layers and blobs by their name and function (for example output blobs).

3.8.3.1 Caffe Solver

Solver is the component updating weight thus using the gradients in the network. Not all layers have parameters to update but they still provide gradient values. The update rule differs between solvers. The basic solvers are `SDG` (stochastic gradient descent), Nesterov and AdaGrad. All of the solvers are using common basic parameters such as momentum μ and learning coefficient α. The basic weight update is

$$W_{t+1} = W_t + (\mu \Delta W_t + \alpha \Delta W_{t+1})$$

where ΔW_t is weight update at time t.

A good strategy for deep learning with SGD is to initialize the learning rate α to a value around $\alpha \simeq 0.01 = 10^{-2}$, and dropping it by a constant factor (e.g., 10) throughout training when the loss begins to reach an apparent "plateau", repeating this several times. Generally, the momentum should be close to 1 in order to decay the learning rate slowly. Usually a good value is $\mu = 0.9$. By smoothing the weight updates across iterations, momentum tends to make deep learning with SGD both stabler and faster. This was the strategy used by Krizhevsky et al. in their famously winning CNN entry to the ILSVRC-2012 competition. Note that the momentum setting μ effectively multiplies the weights updates by a factor of $\frac{1}{1-\mu}$. It is a recommended technique to decrease the learning rate α meanwhile increasing μ [33]

Our case of classifying spectral shapes is a problem of searching for exactly one out of K exclusive classes. Therefore the end of our network has exactly K neurons followed by a *Softmax loss layer* which is a multinomial logistic regression and transforms variadic values into probabilities.There are more possible loss functions such as

- Sigmoid Cross-Entropy Loss – predicts K independent values in range of $(0, 1)$
- Euclidean Loss – regressing real-valued labels (possibly infinite classes distributed in continuous space) $(-\infty, +\infty)$

3.9 Application usage

The application is configurable via JSON files with sections designated to surrounding server. We provide a commandline interface with few arguments.

```
./vodigger --train <repository>
./vodigger --test <model-snapshot> <repository>
./vodigger --time <repository>
./vodigger --dump <model-snapshot> <repository>
```

The mandatory argument for all parameters is the repository in which the classifier operates. It has to be a folder with a configuration file named `config.json` by default.

```
{
    "name": "classifier_name",
    "parameters":
    {
        "mode": "GPU",
        "solver": "solver.prototxt",
        "model": "model.prototxt",
        "test_iters": 1,
        "bench_iters": 20
    }
}
```

Figure 3.5: The minimal configuration file of our classifier

The parameter `model` and `solver` are very similar. They contain names of a model and solver definition files, respectively. Both files has to be proto files following the requirements for Caffe model and solver definition, respectively. There are two differences. First is, that we set up the running mode (either CPU or GPU) directly in this config file (by `parameters.mode`) and not in a solver. The reason is simply that the solver is not always needed and therefore it should not be responsibility of a solver to set up the mode. Second difference is that we have two test_iteration values – one in solver and second in config file. The one in solver says number of test iterations while training a network (for validation). The `test_iter` parameter in the config file determines a number of iterations when the classifier is running in `--test` mode. If the networks outputs ArgMax layer in testing phase the classifier creates a confusion matrix out of it. Last parameter `bench_size` denotes how many iterations of forward-backward pass (in training mode) should be done in order to benchmark given model.

Chapter 4

Implementation

In this chapter we describe our contributions to Caffe framework by implementing a generic input layer. The layer was not merged into the official version of Caffe[11], but we plan to send a pull request when the testing is done so we have a better chance for accepting our pull request.

4.1 Caffe BigData Layer

We have developed a general purpose input layer for Caffe framework to allow bigger data to be easily classified using deep learning. There are two problems with contemporary input layers from the general-purpose point of view. The first one is that they require "efficient" storage formats such as `leveldb`, `lmdb`, `HDF5` or other uncommon data formats. The second is that the input layers takes `batch_size` as input parameter which is not intuitive. Solution to the second problem is straightforward – we introduced input parameter `batch_size` which represents number of megabytes sent to a model every iteration. This parameter practically controls how much data is send to GPU when the model runs in GPU mode. Solution to the second problem is either to create utilities to convert any data into one of the efficient storages or to develop an efficient layer for handling big text files from scratch. We decided that utilities would add only more complexity and therefore we built a new input layer.

The layer provides cyclical read from its source file. Therefore during testing we can't go through a file once. Instead we need to set up the number of iterations so it matches the number of data. To our defence we can say that "going through data once" is not possible with any input data layer in Caffe. It is given by Caffe's design decisions.

In order to speedup reading of inefficient CSV format the layer is using more threads and furthermore it transform data into binary files and store

[11] `https://github.com/BVLC/caffe`

them along with the original text files. This transformation is performed meanwhile reading the textual format so no preprocessing is needed. When we reach end of CSV file we automatically switch to reading the binary file. The binary file is not deleted at the end so it can be reused next time. Changes in model (concerning batch size or other than BigData layer) won't invalidate the binary file.We wanted to cache the data in memory but since we are supposed to handle huge data we can not fit them all in RAM. However, if the file is smaller than the `batch_size` then we do cache the file.

4.1.0.2 Performance

Figure 4.1 shows relative performance of BigData Layer compared to `HDF5` input layer which is bundled within standard distribution of Caffe. The test was performed using CPU and 0.5MB chunks of data. Data, solver and model were identical. The measurements are average values from 4 repetition of the same experiment. The performance is linear with respect to chunk sizes. The `BigData no cache` reads plain text CSV and transforms it into binary file called cache meanwhile. The noticeable increase of performance happens when the layer finishes reading from a flatfile and in the next iteration it starts reading from the transformed binary file. The *round trip time* is the time needed for the forward and backward pass of a whole network (in this case it is the simple MLP network).

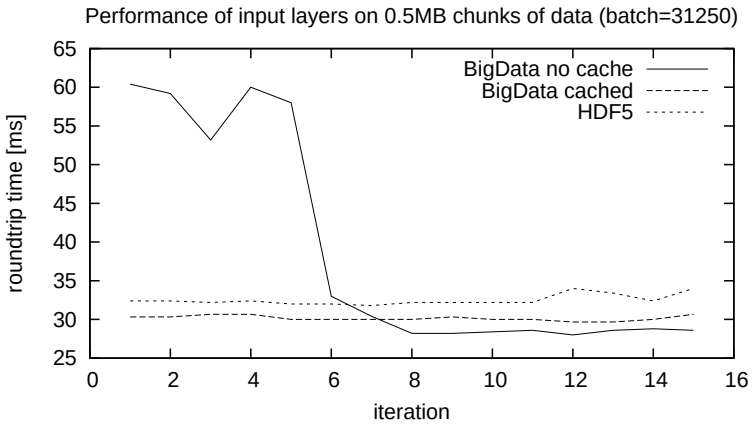


Figure 4.1: Performance of BigData Layer compared to HDF5

Another optimization was added in order to lower disk usage. If the chunk size is greater than the size of a source file then the data stay in memory and are not read from a disc over and over again.

4.1.0.3 Configuration

Our BigData layer can be added to standard model definition as shown in Figure 4.2. It has many default values and therefore the only required are `source`, `chunk_size` and all data indices. The source can omit `label` index which tells the network that data are to be classified.

```
layer {
  name: "traindata"    # name can be arbitrary
  top: "data"          # first layer will be used for data
  top: "label"         # second layer for labels
  top: "ids"           # third layer for IDs (row numbers)
  type: "BigData"
  big_data_param {
    source: "spectra.train.csv"
    chunk_size: 1.5    # value in MB
    separator: " "     # separator is "," by default
    newline: "\n"      # newline is also "\n" by default
    header: 0          # skips # lines, by default 1
    # following indices are always inclusive and 0-based
    label: 0           # column index of label (optional)
    data_start: 1      # column index of data start
    data_end: 1863     # column index of data end
  }
}
```

Figure 4.2: Example configuration of BigData layer

There are few rules to the data format. Labels has to be integers from interval $< 0, N >$ where N is number of classes. The layer suppose 1D input data and therefore constructs blobs where all data are put into `width` part of Blobs.

4.1.0.4 Memory consumption

Unfortunately implementation of Caffe's GPU parallelism prevents us from putting really big data on the graphic card. Caffe is using, in some layers, parallelism of such a degree that every column/pixel is assigned one thread. If we take into account constraints imposed by CUDA 3.x then we can have only 2^{16} blocks and 2^{10} threads per block. That means we can send a maximum of 64 MB in one batch. Newer graphic cards offer higher number of blocks so any model will work even on bigger chunk of data than 64MB. Nevertheless we should count with the worst case.

64MB does not seem much but we have to take into account expansion of data. Every layer allocates storage for its results so basically we have to

multiply input data by number of layers. We elaborate more precise memory consumption in equation 4.1. The equation shows overall memory consumption M based on size of inputting data. In the following computations, we suppose that average depth of deep network is between 5 and 10 layers.

$$\begin{aligned} M &= S_{input} * C_e * D + S_{weights} \\ M &= 64 * 2.5 * (5, 10) + S_{weights} \\ M &= (800, 1600) + S_{weights} \end{aligned} \tag{4.1}$$

where S_{input} is the input data size in megabytes (MB), C_e is a *network expansion coefficient* which we discuss later, D is depth of network for which we used a vector of two values for usual min and max and $S_{weights}$ is negligible memory needed for storing weights of connections. Note that average desktop graphics card has 3GB of global memory. So if we instantiate S_{input} with our limiting 64MB of data we will end up with memory consumption between 1GB and 1.5GB based on the depth of our network.

All the participants in equation 4.1 are obvious except the *network expansion coefficient* C_e. We estimated that number of neurons per layer is in average 2.5 times more than neurons in the input layer. The estimated value 2.5 is based on LeNet and few other standard networks. For example LeNet expansion coefficient is ≈ 3.2 based on layer sizes $[1.0, 14.7, 3.7, 4.1, 1.0, 0.6, 0.6, 0.01]$.

Chapter 5

Performance and Precision

In this chapter we describe development of two models based on deep neural networks. The models were build to match the given datasets. The first model is a proof of concept to create a simple functional neural network and to measure its performance on CPU and GPU. The other model is a convolutional network which unleash the possibilities of deep networks.

5.1 GPU Multi-Layer Perceptron

We tested a simple multi-layer perceptron model on standard astronomical dataset SDSS DR12 and compared our results with classification performed by the authors of scikit-learn[34]. We used the same download method as described at astroML[12]. The dataset is designed for classification of stars and quasars based on multiple features from which were selected magnitudes in standard SDSS filters. The dataset is kindly offered by Sloan Digital Sky Survey (SDSS)[13].

5.1.1 SDSS Quasar-Star dataset

We followed data preparation described by scikit-learn[34]. The dataset comes in two separate files – for every category one file. Table 5.1 shows brief statistics of the dataset and another one, which was used for the naive bayes classifier.

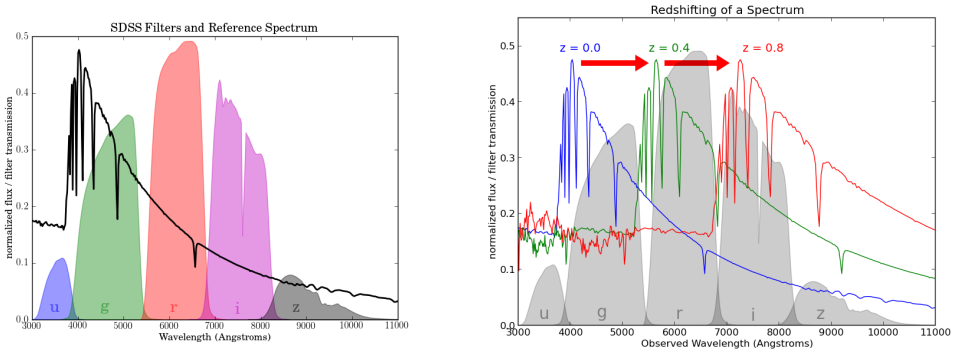
All classifiers except the Naive Bayes were tested on the same dataset as our model. Naive Bayes, however, uses data from the same source (SDSS) but of a different size. Its dataset is a mixture of 700,000 objects out of which 500,000 is used for training and the ratio of stars and quasars approximately 14:1.

[12] `http://www.astroml.org/user_guide/datasets.html`
[13] `http://www.sdss.org/`

	Total	Stars (S)	Quasars (Q)	S/Q ratio
Training	390,040	294,814	95,226	3:1
Testing	43,003	32,446	10,557	3:1
Naive Bayes (train)	500,000	-	-	14:1

Figure 5.1: Brief statistics of the quasar-star datasets.

Both datasets are tabular values of magnitudes in a few different filters altogether with labels. Magnitudes are scalar values which denote radiation flux relative to a reference object. The Figure 5.2a shows a spectrum of an object (in our case Vega) and as a background there are filters in different wavelengths.



(a) Filters

(b) Redshift of spectra

Figure 5.2: Filters and redshift explanation

The filters, used in SDSS, are ultraviolet(u), green(g), red(r) and infra red(i, z). In order to get magnitude we need to compute photon flux I first. It is an integral of object spectrum multiplied by filter's throughput as shown in equation 5.1. Since we compute photon flux for every filter, we denote it by its filter sign (eg. I_u).

$$I_x = \int_0^\infty f_x(\lambda) S_\nu(\lambda) \frac{d\lambda}{\lambda} \tag{5.1}$$

where $x \in u, g, r, i, z$ or other set naming filters used for observation, $f_x(\lambda)$ is the filter value at wavelength λ and $S_\nu(\lambda)$ is the photon flux at the wavelength λ. Equation 5.2 shows the final transformation of a flux value into *magnitude* m_1

$$m_1 - m_{ref} = -2.5 \log_{10}(\frac{I_x}{I_{ref}}) \tag{5.2}$$

Magnitude is a historical relict which is still used. It is defined as negative logarithm so it has actually lower value for brighter object than a reference

object. It is sometimes called an apparent brightness. The astronomers use star Vega as the reference object. Following this point of reference, the Moon is given a magnitude of -13, while Venus -5 and Sirius -1.5.

Different groups of objects usually have a different redshift due to different cosmological distances. For example galaxies has lower redshift than quasars. A redshift moves the spectral line on the x-axis. It doesn't introduce any changes in the shape of spectra as shown in Figure 5.2b.

We constructed two different feature sets. First dataset consisted from unprocessed magnitudes - therefore having 5 columns of features. The second dataset was constructed based on domain knowledge of differences between quasars and stars in their spectra. The features were subtractions of magnitudes from the subsequent filters $(u-g, g-r, r-i, i-z)$. The difference between following features identifies the positioning of a spectra according to the filters thus it can guess the overall redshift. This is the best known feature selection with this dataset. We are using the second dataset most of the time so we can compare with astroML results.

5.1.2 Model

We constructed a simple multi-layer perceptron network for comparison with other classifiers which were used on the same dataset. This implementation also provides a baseline for performance and accuracy testing for our other models.

The first version has only one layer. We didn't introduce any scaling nonlinearities because the classes are distinct based on difference between intensities in subsequent filters so any scaling down would damage the accuracy of the classification.

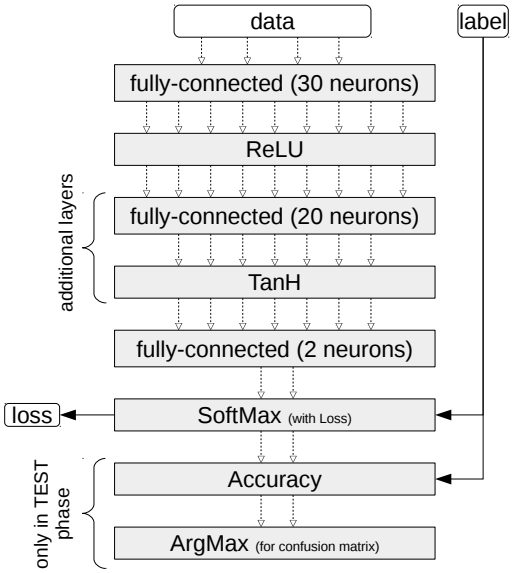
The first fully-connected layer with 10 dimensional output showed as sufficient. Best practices published by the DAME team [12] say that number of neurons should descend with layers and the starting value, which is equal to the number of neurons in the second layer, should be $2N-1$ where N is number of input neurons so we are very close to the recommended value which would be in our case 7. Our data are separable by a polynomial of first degree in 3D space.

5.1.2.1 Accuracy

We were building the model incrementally from the input layer. The accuracy was measured after every step and we ended up with architecture shown in figure 5.3b. With only one hidden layer and no non-linearities in the network we were able to achieve 96.3% accuracy with loss=0.3. Additional linear layers did not improve average result. However additional non-linearity was able to improve the loss down to 0.1 and accuracy up to 98.59%. More surprisingly a ReLU unit improved average result more than a hyperbolic-tangent unit.

Solver	
iterations	10,000
learning rate	0.3
step interval	300
gamma	0.9
batch size	5,000

(a) Simple MLP solver for QS problem



(b) Simple MLP model for QS problem

Figure 5.3: Simple MLP network configuration

	Star	Quasar	Accu.
Star	33,766	180	99.5%
Quasar	456	10,598	95.9%

(a) 2-layer network with total accuracy 98.59%.

	Star	Quasar	Accu.
Star	33,751	195	99.4%
Quasar	215	10,839	98.1%

(b) 3-layer network with total accuracy 99.1%.

Figure 5.4: Confusion matrix of simple QS model. Rows denote true class, columns output of the network.

With the addition of another layer we noticed an increase in accuracy up to 99%. The accuracy was always around this level and with fine-tuning of the weights and number of outputs we were able to increase up to 99.1%. The network with more layers is able to distinguish between more classes while the simple 2-layer network was more precise with one class. Any other additional layer worsened the result. There are two possible explanations. The first is that the gradient dissolves in too many layers. The second is that more layers bring higher dimensionality in which the data might not be that easily separable.

With the introduction of another layer we switched to the first dataset because the second dataset is produced as a linear combination of the first one. But we never reached the precision of the second dataset. Even with a `dropout` layer and uniform weights initialization in a wide interval such as

$(-3, +3)$ the net wasn't able to produce the correct linear combination.

We compared our multi-layer perceptron classifier with *Gaussian Naive Bayes* whose results are available online[14] and with classifiers originally used by `scikit-learn` on this dataset. The values in the table 5.5 are the best obtained from a few iterations.

Classifier	Stars	QSOs	Total
Decision tree	99%	100%	99%
kNN	98%	100%	99%
our GPU MLP	**99%**	**98%**	**99%**
GMM Bayes	75%	100%	75%
Naive Bayes	99%	14%	94%
Logistic regression	75%	0%	75%

Figure 5.5: Comparison of accuracy of different classifiers

The skew in Naive Bayes' dataset explains relatively bad metrics of the classifier. The skew in training data is forcing the model to put more stress on one class. There are ongoing debates if training samples should have homogeneous distribution of classes with sacrifice of training data. The results show that even bigger amount of data did not save the model from biasing. Our and some other astromML's models handle smaller skew in the data gracefully. The results show that two bottom-line models could not handle the skew well.

5.1.2.2 Performance

First we investigated how the performance changes with respect to amount of data in one batch. The results show that both CPU and GPU implementations scale linearly with different coefficients. We computed those coefficients and put them into an equation which shows time demand based on batch size S_B

$$T_{CPU} = 55 * S_B$$
$$T_{GPU} = 6.2 * S_B$$

Since our network accommodates most of the common layers (fully-connected, ReLU, tanh) we can say that those coefficients approximate GPU speed-up over CPU. Thus we can state that GPU implementation of a simple MLP network is about 9 times more efficient than CPU implementation.

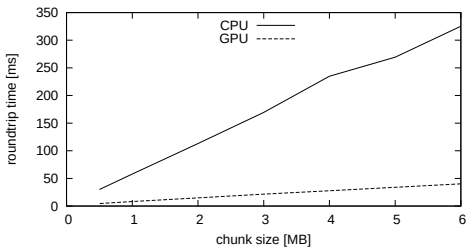


Figure 5.6: Performance based on chunk size.

[14] `http://www.astroml.org/sklearn_tutorial/classification.html`

Linear scale with respect to block
size shows that Caffe does not use `CUDA streams`. CUDA streams are way of asynchronous data transfer and computations. If CUDA streams were involved then we would get sub-linear time with special case where the time of data transfer is equals to computations and in this case the time should stay constant for small range of chunk size.

In order to further investigate performance differences between CPU and GPU implementations we present a breakdown by layers, implementations and phases in table 5.7.

Layer	phase	CPU[ms]	GPU[ms]
train	F	3.10	2.65
	B	0.00	0.00
ip1	F	13.70	1.37
	B	20.88	3.03
relu1	F	5.02	1.28
	B	10.26	1.12
ip2	F	10.70	1.32
	B	19.26	3.22
tanh1	F	17.40	1.36
	B	1.51	1.25
ip4	F	7.43	1.27
	B	14.27	3.13
loss	F	197.73	12.30
	B	3.81	0.17

Figure 5.7: Time breakdown by layers and implementation. Phase specifies further the run, if it was F resp. B forward resp. backward run.

In terms of classification of unknown object we have measured average throughput 11,400 objects per millisecond which is equal to dataflow of 760 MB/s. This value is for forward pass exclusively and therefore it states only the classification throughput, not the performance while training.

We were about to compare performance of our MLP model with DAME. Despite immense help by Dr. Massimo Brescia we were unable to load our data into DAMEWARE and perform any benchmarks. DAME offers massive parallel version of MLP model with optional quasi newton algorithm (QNA) which would be great for comparison. We did try again for our next convolutional model with slightly better result.

5.2 GPU Convolutional Network

We introduce a new technique for classifying astronomical spectra. The idea comes from image classification where the convolutional networks are heavily

used. Convolutional networks were already used for classification of spectra [35] with accuracy over 90% but with a very expensive preprocessing phase. The preprocessing they used was to convert spectra into 60x60 pixel images. Those images were fed into standard `LeNet-5` and the network achieved accuracy 96.5%. However, we cannot compare our accuracies because they used a different dataset. In order to put this classification method to a practical use we have built a model which can deal with spectra in the form as they are available through Virtual Observatory tools. We imagine the spectra as 1D images where we already have brightness of pixels. The brightness, which is represented by flux, needs to be normalized to continuum but that is usually done in spectra processing pipeline. Therefore our model is able to classify massive amounts of continuum normalized spectra obtained from standard sources without any further preprocessing.

5.2.1 Ondřejov's Be-stars dataset

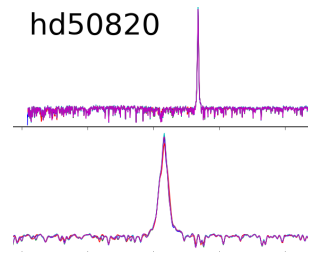
The classical Be stars are non-supergiant B type stars whose spectra have or have had at some time, one or more emission lines in the Balmer series [36]. In particular the H_α emission is the dominant feature in spectra of these objects. Characteristic for Be stars are the single or double-peak profiles and sometimes so called shell lines deep absorptions in centre of the emission. Figure 5.8 show spectral shapes of two example objects for each category. The upper half of every image is a complete spectra ranging from 6200 to 6800 Ångström. The bottom half shows the interesting part in greater detail. The y-axis is normalized flux.

Our dataset contains 1696 spectra samples of stars divided into 5 classes. The distribution of classes is shown in table 5.9 in row "Total". All classes except #2 are Be stars. Each spectrum composes from approximately 2000 flux values around H_α line (656.28 nm = 6562.8 Å). The spectra in our dataset were binned into 0.15 Å wide bins which unifies all spectra with respect to x-axis. There is no need for wavelengths to be stored in the data so every record is only 1 dimensional array of flux values.

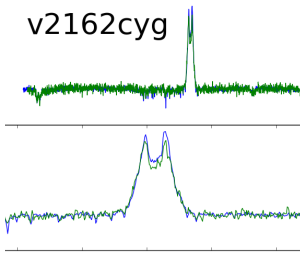
There is clear dominance of class #2 in the raw dataset because this class is assigned to generic absorption stars which are not Be stars. The model tends to put more weight to the class thus skew the results of classification. It is always better to sacrifice some data in favour of homogeneous distribution of training samples. We redistributed the classes for training and testing samples as shown in table 5.9.

5.2.2 Model

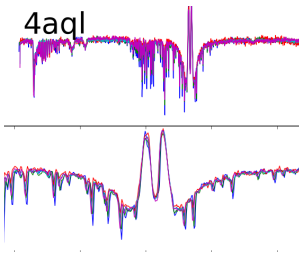
The base for this model was taken from the previous multi-layer perceptron. We have constructed two cNN models. First model has one wide convolutional layer and the second has two narrow convolutional layers. In both cases it



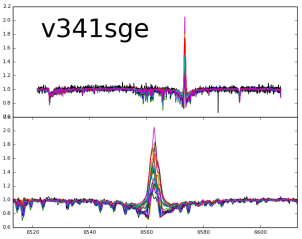
(a) Be star with emission, class #0



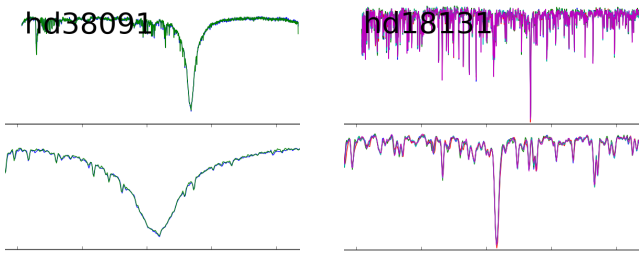
(b) Be star, double peak emission, class #1



(c) Be star, absorption inside emission, class #3



(d) Be star, chaotic emission, class #4



(e) Ordinary absorbing stars, class #2

Figure 5.8: Example spectra lines

	records	#0	#1	#2	#3	#4
Total	1,696	178	172	1,159	56	131
Training	663	163	152	180	52	116
Testing	94	15	20	40	4	15

Figure 5.9: Ondřejov's datasets class distributions

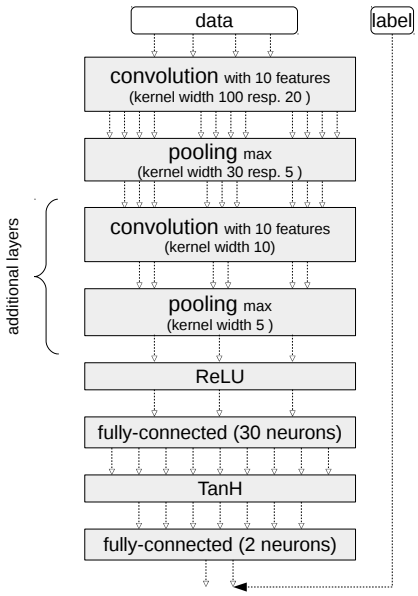
was necessary to put a hyperbolic tangent layer right in front of the output layer otherwise the output grew too high and made its loss function to diverge. Hyperbolic tangent scales any number between $(-1, 1)$ so it is ideal for scaling data before computing a loss.

Convolutional layer is usually followed by a pooling layer to sharpen feature map by removing partial fit of features. We have chosen max-pooling layer which works similarly to convolution but instead of a convolution matrix it uses `max` function over a region of neurons.

As in the previous MLP model, it was necessary to introduce some non-linearity. We have used ReLU unit which is preferred over other non-linear units for its speed. A non-linear unit is usually placed between last pooling layer and before first fully-connected layer. We followed this practice. The resulting network is depicted in figure 5.10b.

Solver	
iterations	10,000
learning rate	0.3
step interval	300
gamma	0.9
batch size	5,000

(a) Solver used for convolutional network



(b) Architecture of convolutional network for classifying spectra.

Figure 5.10: Convolutional network configuration

5.2.2.1 Accuracy

During training, we experienced that convolutional networks are very sensitive to initial learning rate. Convolutional networks require smaller initial learning rate and slower decay. The experiments show that 0.1 is optimal initial value

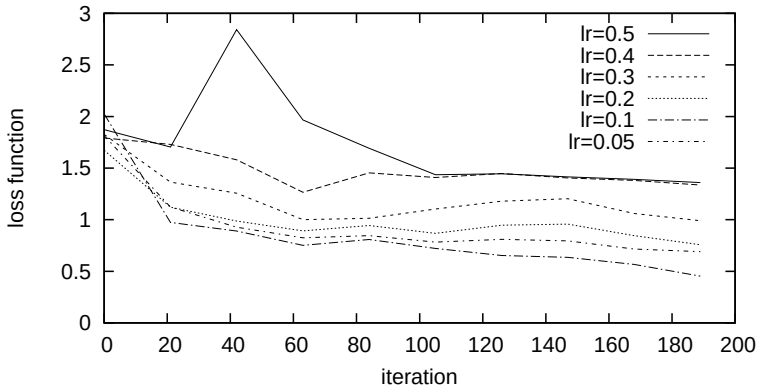


Figure 5.11: Progress of loss function based on learning rate.

The kernel size is very important factor in optimization of a convolutional layer. The most complicated spectral profile of any Be-star is double peak. If a convolutional network should match the important features it needs to

overcome the small disturbing waves which are presented, for example, in spectra of class #2. The ideal width of kernel is wide enough to smooth turbulent waves but narrow enough to capture the top "saddle" point between two peaks. Convolutional layer should match any possible shape so one of those shape can be the saddle. The figure 5.12 shows the dependency between kernel width and a loss function. It confirms for one layered convolutional networks that kernel has to be rather bigger than smaller in order to overcome jitter.

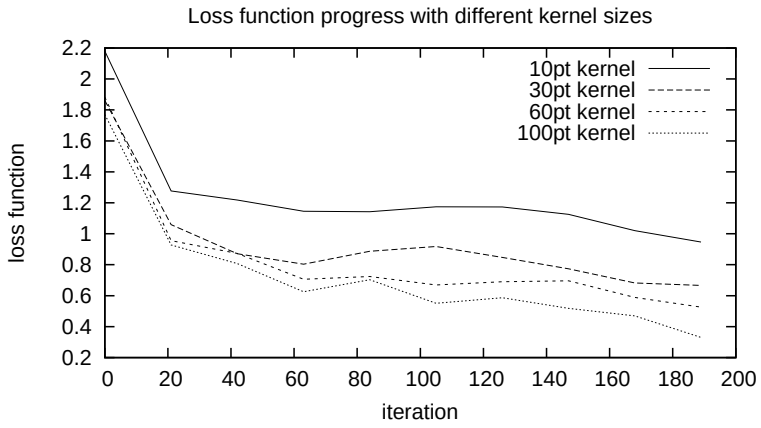


Figure 5.12: Progress of loss function based on kernel size

The reason for having wider kernels in one layered cNN is that we can not say in advance how wide the features will be. It is better to let the convolutional layer to extract the important knowledge by itself. Figure 5.13 shows one of the feature maps which was yielded by our convolutional network. This feature matches negative slope in spectral shape.

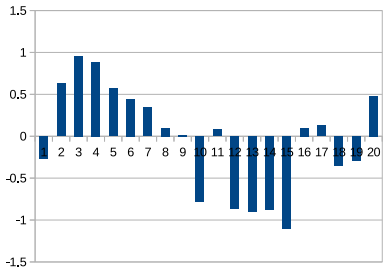


Figure 5.13: A smaller feature from two-layer convolutional network

The most successful kernel of width 100 had accuracy 94.7% with confusion matrix presented in table 5.14. This value seems to be the limit for one layer convolutional network on our data. If we train the network further it starts to overfit.

guess:	#0	#1	#2	#3	#4
true #0	95	0	0	0	13
true #1	0	128	0	0	7
true #2	0	0	272	0	0
true #3	0	0	7	21	0
true #4	0	7	0	0	94

Figure 5.14: Confusion matrix for one-layered cNN after 1500 iterations. Accuracy 91.32%

The patterns in spectra are simple – a classifier should learn few shapes like slops up and down, flats and saddle points. Therefore it is crucial not to make the search space too complex. During training, we observed that ideal number of convolutional features is between 10 and 20.

As we said earlier, it is vital to let the convolutional network converge slowly. During our experiments it took more than 1500 iterations for two-layers cNN to find the right features. The simpler one-layered cNN converged before 1500 iterations where it achieved its maximal accuracy and since then it was slowly overfitting.

After 2000 iterations convergence was almost final and we obtained 99.07% accuracy with only one misclassified spectrum as shown in confusion table 5.15.

guess:	#0	#1	#2	#3	#4
true #0	16	0	0	0	0
true #1	0	25	0	0	0
true #2	0	0	42	0	1
true #3	0	0	0	6	0
true #4	0	0	0	0	18

Figure 5.15: Confusion matrix for two-layered cNN after 2000 iterations. Accuracy 99.07%

5.2.2.2 Performance

We are interested in how convolutional layer performs under bigger data sizes and with different parameters. As we did in the previous section, we measured performance based on chunk size. It turned out to be linear again but with different coefficients.

$$T_{CPU} = 61.0 * B_S$$
$$T_{GPU} = 0.95 * B_S$$

In order to see why the coefficients changed we measured performance of every layer separately. The results are shown in table 5.16. The table con-

firms that convolutional layer is the heaviest for computation. The computed coefficients are valid only for this network architecture with one convolutional layer. Having more convolutional layers than the fully-connected ones will bring less synchronization points thus the theoretical speedup can reach up to 110 times. The overall speed-up by using GPU for the particular network shown in table 5.16 is around 55 times.

Layer	phase	CPU[ms]	GPU[ms]
train	F	2.39	2.29
	B	0.00	0.00
conv1	F	186.47	0.61
	B	138.71	1.24
pool1	F	32.45	0.83
	B	24.97	0.76
relu1	F	0.02	0.02
	B	0.05	0.02
ip1	F	0.45	0.04
	B	0.84	0.07
hyp1	F	1.02	0.02
	B	0.05	0.02
ip2	F	0.40	0.04
	B	0.59	0.08
loss	F	0.79	0.20
	B	0.01	0.06

Figure 5.16: Time breakdown of convolutional network.

The average throughput of the two-layered cNN is 192 spectra per millisecond which is equal to 1440 MB/s.

We tried to benchmark performance with DAME again, using MLP model with QNA optimization. After few modifications to DAME, performed by Dr. Massimo Brescia, we were able to load our data to the application. Unfortunately any MLP used model gave us estimate of 3 months to finish. DAME implementation is not suitable for such a multicolumn data.

Conclusion

While this thesis was being written a PhD thesis [27] was published, and contained objectives that came out to be very close to this thesis' aims: developing a massive parallel classifier which was following the most active field of classification – deep neural networks. As a consequence, it was decided that duplicating the results was not a viable option. The PhD thesis outcome was a framework called `Caffe` and it met exactly our requirements. The reason underlying the use of Caffe, and going against the initial plan, resides in its efficient design of layers. Caffe conceptualizes layers as matrices, operations as matrix multiplications and connections between layers as just another layer. One of the outcomes of our pre-Caffe phase is a thorough summary of the latest achievements and techniques in neural networks in general.

However, Caffe's input layers were not as generic as required for this thesis and therefore we have developed `BigData` layer which allows the classifier to load generic and big files in CSV format. Since most of big data applications either directly work with CSV or are able to export into them, we decided to use this format as the common ground. Until now, Caffe was more of an academical framework with a very narrow focus on images. Further developments of `BigData` layer could bring multi-file support or even the ability to communicate with distributed file system to fully unleash the possibility of working with unlimited big data.

Two deep networks were designed for benchmarking. The first one was a multi-layer perceptron network designed for standard tabular astronomical dataset. Its accuracy was slightly above average in comparison with many other classifiers used for the same dataset. Our main objective was to develop performing classifier for bigger data suitable for practical use. No comparable works on performance were found in the literature therefore the outcomes of our current experiments remain the only data available. Our throughput was 11,400 objects per millisecond which is equal to 760 MB/s on GeForce GTX 980, using SSD discs. The size of the input file is theoretically unlimited.
The second model was aiming to prove applicability of convolutional networks

for raw astronomical spectra classification. The model was successful, having a higher accuracy than a similar master's thesis model [35] which focused solely on classification of stellar spectra using convolutional networks. We planned to compare the performance of the model with DAME models but we were unable to even load our data into the application. With the help of Dr. Massimo Brescia, the most active DAME developer, who proceeded into making changes to the application for the sole purpose of this thesis, we were able to push further our experimentation. In the end, we succeeded in using DAME for our purpose. Unfortunately the DAME models do not allow classification of raw spectra and therefore we do not have any performance comparison. Our performance was measured as 192 spectra per millisecond which is equal to 1440 MB/s. The higher throughput of the second model is caused by smaller fully- connected layers and bigger data per record resulting in less synchronization points.

The first dataset we were given is not well suited for deep neural networks. Deep neural network is a unique model which can find patterns in very complex data. The more complex the data are, the more layers has to be added. Classifying such a tabular data, which were in the first dataset, with neural networks was expected to yield lower accuracy than decision trees, which was confirmed. The reason for lower accuracy is that deep networks has potential to solve complex problem and cannot match witch simpler models who are designed to decide from simple tabular data. Since the astronomers know the approximate operations to obtain the labels from the data, it would be more efficient to build a specialized classifier with this domain knowledge.
The spectra shapes, however, offer more complexity than tabular values. This is the type of data which is tailored to neural networks. Data where the features leading to the labels are unknown or hardly describable. In order to find the crucial patterns for classification, it is necessary to use a convolutional network. To make the problem even more challenging, we could add another level of complexity by linking the spectra of the same object together and provide time information to them. Such data would require very deep networks or even a new type of networks called spiking networks. Another very interesting direction in research would be to analyse the feature maps yielded by convolutional networks in order to find a new knowledge hidden in the data.

The parallelism in Caffe is rather limited at the time of writing this thesis. According to our measurements the time demand grows linearly with data size sent to a GPU. Ideally, the time should grow sub-linearly or be constant. In order to improve the performance it is necessary to start using the latest CUDA features such as `streaming` along with a multi-device execution. Since we were active in the development of Caffe, we noticed that parallelization in the way we present it is the main development effort of the core developers of Caffe. The implementation is almost finished and is expected to be released in a few months.

Bibliography

[1] Hey T., T. K., Tansley S. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, 2010.

[2] Mussi, L.; Nashed, Y. S.; Cagnoni, S. GPU-based Asynchronous Particle Swarm Optimization. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, New York, NY, USA: ACM, 2011, ISBN 978-1-4503-0557-0, pp. 1555–1562, doi:10.1145/2001576.2001786. Available from: `http://doi.acm.org/10.1145/2001576.2001786`

[3] Jaros, J. Multi-GPU Island-Based Genetic Algorithm for Solving the Knapsack Problem. In *WCCI 2012 IEEE World Congress on Computational Intelligence*, June 2012, pp. 10–15.

[4] Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *CoRR*, volume abs/1404.5997, 2014. Available from: `http://arxiv.org/abs/1404.5997`

[5] Felipe C. Moraes, N. D. F., Silvia C. Botelho; Gaya, J. F. Parallel High Dimensional Self Organizing Maps Using CUDA. In *Robotics Symposium and Latin American Robotics Symposium (SBR-LARS)*, October 2012, pp. 302–306.

[6] Do, T.-N.; Nguyen, V.-H.; Poulet, F. A Fast Parallel SVM Algorithm for Massive Classification Tasks. In *Modelling, Computation and Optimization in Information Systems and Management Sciences*, *Communications in Computer and Information Science*, volume 14, edited by H. Le Thi; P. Bouvry; T. Pham Dinh, Springer Berlin Heidelberg, 2008, ISBN 978-3-540-87476-8, pp. 419–428, doi:10.1007/978-3-540-87477-5_45. Available from: `http://dx.doi.org/10.1007/978-3-540-87477-5_45`

[7] Yisheng Liao, A. R. e. a. Learning Random Forests on the GPU. Technical report, Department of Computer Science, New York University, 2013. Available from: `https://github.com/EasonLiao/CudaTree/`

[8] Xu, K.; Louis, S. J.; Mancini, R. C. A Scalable Parallel Genetic Algorithm for X-ray Spectroscopic Analysis. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, New York, NY, USA: ACM, 2005, ISBN 1-59593-010-8, pp. 811–816, doi:10.1145/1068009.1068145. Available from: `http://doi.acm.org/10.1145/1068009.1068145`

[9] Alba, E.; Tomassini, M. Parallelism and evolutionary algorithms. In *IEEE Transactions on Evolutionary Computation*, 6, 2002, pp. 443–462.

[10] Garofalo, M. *GPU Computing for Machine Learning Algorithms*. Master's thesis, Faculty of Engineering, Università degli Studi di Napoli Federico II, 2011.

[11] Kennedy, J.; Eberhart, R. Particle swarm optimization. In *IEEE Int. conf. on NeuralNetworks*, volume IV, IEEE CS Press, 1995, pp. 1942–1948.

[12] M. Brescia, A. S. Fast Multi Layer Perceptron with Genetic Algorithm. 05 2014. Available from: `http://dame.dsf.unina.it/documents/MLPQNA_UserManual_DAME-MAN-NA-0015-Rel1.3.pdf`

[13] Maass, W. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 1996.

[14] T. R. Stovall, S. K. GPUSCAN: GPU-based Parallel Structural Clustering Algorithm for Networks. In *IEEE Transactions on Parallel and Distributed Systems*, 2014, p. 1.

[15] Dongen, S. Graph Clustering via a Discrete Uncoupling Process. In *SIAM J. Matrix Analysis and Applications*, volume 30, 2008, pp. 121–141.

[16] Arthur, D.; Vassilvitskii, S. K-means++: The Advantages of Careful Seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, ISBN 978-0-898716-24-5, pp. 1027–1035. Available from: `http://dl.acm.org/citation.cfm?id=1283383.1283494`

[17] Ma, W.; Agrawal, G. A translation system for enabling data mining applications on gpus. In *23rd International Conference on Supercomputing (ICS'09)*, ACM Press, 2009, pp. 400 – 409.

[18] Chrysos, G.; Dagritzikos, P.; Papaefstathiou, I.; et al. HC-CART: A Parallel System Implementation of Data Mining Classification and Regression Tree (CART) Algorithm on a multi-FPGA System. *ACM Trans. Archit. Code Optim.*, volume 9, no. 4, jan 2013: pp. 47:1–47:25, ISSN 1544-3566, doi:10.1145/2400682.2400706. Available from: `http://doi.acm.org/10.114z5/2400682.2400706`

[19] Leo Breiman, e. a., Jerome H. Friedman. *Classification and regression trees*. Brooks/Cole Publishing, Monterey, 1984.

[20] BREIMAN, L. Random Forests. In *Machine Learning*, October 2001, pp. 5–32.

[21] Zaki, M. J.; Ho, C.-T.; Agrawal, R. Parallel Classification for Data Mining on Shared-Memory Multiprocessors. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, volume 0, 1999: p. 198, ISSN 1063-6382, doi:http://doi.ieeecomputersociety.org/10.1109/ICDE.1999.754925.

[22] Palička, A. *Application of Random Decision Forests in Astroinformatics*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2014.

[23] Lopes, N.; Ribeiro, B. GPUMLib: An Efficient Open-Source GPU Machine Learning Library. In *International Journal of Computer Information Systems and Industrial Management Applications*, volume 3, 2011, ISSN 2150-7988, pp. 355–362.

[24] NVIDIA. *GPU Accelerated Deep Learning*. 2015, [Online; accessed 19-March-2015].

[25] NVIDIA. CUDA 6.5 Performance Report. Technical report, NVIDIA, 2014. Available from: `http://developer.download.nvidia.com/compute/cuda/6_5/rel/docs/CUDA_6.5_Performance_Report.pdf`

[26] J. Bergstra, O. B. e. a. Theano: A CPU and GPU Math Compiler in Python. In *Proceeding of the 9th Python in Science Conference (SCIPY 2010)*, 2010, pp. 1–7.

[27] Jia, Y.; Shelhamer, E.; Donahue, J.; et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[28] Chintala, S. Convolutional Networks benchmarking. Technical report, Facebook, 2015, [Online; accessed 19-April-2015]. Available from: `https://github.com/soumith/convnet-benchmarks`

[29] Krizhevsky, H., Sutskever. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, 2012, pp. 1106–1114.

[30] Koza, J. *Design and implementation of a distributed platform for data mining of big astronomical spectra archives*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2014.

[31] Sutskever, I. Machine Learning and Magical Thinking. Said in a podcast, 1 2015, accidentally slipped out of his lips while doing interview for www.thetalkingmachines.com. Available from: `http://www.thetalkingmachines.com/blog/2015/1/15/machine-learning-and-magical-thinking`

[32] V. Nair, G. E. H. Rectified linear units improve restricted boltzmann machines. In *27th International Conference on Machine Learning*, 2010, pp. 807–814.

[33] Yangqing Jia, E. S. e. a. *Caffe*. 2015, [Online; accessed 25-March-2015].

[34] Pedregosa, F.; Varoquaux, G.; Gramfort, A.; et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, volume 12, 2011: pp. 2825–2830.

[35] Hála, P. *Klasifikace spekter pomocí konvolučních neuronových sítí*. Master's thesis, Masaryk University, Faculty of Science, 2014.

[36] Porter, . R. T., J. M. Classical Be Stars. In *The Publications of the Astronomical Society of the Pacific*, volume 115, 2003, pp. 1153–1170.

Appendix A

Acronyms

C++11 a new standard of C++ language approved in 2011.

cNN convolutional neural network.

direct memory access a device has access to RAM bypassing CPU.

FPGA Field-programmable gate array is a chip configurable after manufacturing using hardware description language (HDL).

GPGPU general purpose graphical processing unit.

GPU graphical processing unit.

GT/s giga transfers per second.

ML machine learning.

MLP multi-layer perceptron.

redshift Redhift happens when a light increases in wavelength of shifts toward the red end. Its value is usually represented by letter z.

Virtual Observatory Worldwide scientific organisation enabling integrated global access to the data gathered by astronomical observatories.

Appendix B

Contents of CD

- `vodigger` ..... the classifier directory
  - `readme.md` ..... the installation and usage instructions
- `caffe` ..... the fork of Caffe with BigDataLayer included
- `doc` ..... the thesis source directory
  - `DP_Peterka_Tomas_2015.tex` ..... the LaTeX source of the thesis
- `DP_Peterka_Tomas_2015.pdf` ..... the Diploma thesis in PDF format