Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Master in Innovation and Research in Informatics (MIRI)

High Performance Computing (HPC)

2016-2017 |Spring Semester

Application Performance Evaluation using Deep Learning

Author:
Simon Söderlind

Advisor:
Daniel Jimenez
Computer Architecture Department (DAC)

Supervisor:
Judit Gimenez
Barcelona Supercomputing Center (BSC)

Acknowledgements

First of all, I would like to thank Jesus Labarta and Judit Gimenez for proposing and giving me the opportunity to work on this exciting project.

I would like to give another thank you to Judit Gimenez for her supervising role for this thesis.

Thank you Daniel Jimenez for agreeing and taking on a tutoring role.

Finally, I would also like to thank all my family for their constant support.

Abstract

Developing software for exascale systems will become even more challenging than for today's systems. Methods for evaluating the performance of applications and identifying potential weaknesses are essential for reaching optimal performance. Though the tools available today are not widely used, and generally require some expert knowledge.

In recent years different deep learning techniques have enjoyed great success in various fields, and especially in image recognition. Though it is still to find its way in to the area of application performance evaluation.

This work will take the first step towards introducing deep learning to the area of HPC performance evaluation, opening the door for others. Convolutional neural networks will be fed images of timeline views of HPC applications and will identify the intrinsic behavior of the application and return some principal performance metrics.

The results show that deep learning techniques indeed can be utilized for evaluating the performance of parallel applications, with the main limitation for its success being the sizes of the data sets available. Furthermore a number of exciting directions for taking the next step utilizing deep learning techniques with performance evaluation are suggested.

Contents

1 Introduction, Motivation and Goals **1**
1.1 High Performance Computing 2
1.2 Performance Evaluation 3
1.3 Deep Learning 5
1.4 Related Work 5
1.5 State of the Art CNN Architectures and Applications 6
1.6 Paper Outline 7

2 Performance Evaluation **9**
2.1 POP Performance Metrics 9
2.2 BSC Tools Parallel Performance Analysis Suite 11
2.2.1 Extrae 11
2.2.2 Paraver 12
2.3 Analysis Process 14

3 Deep Learning **15**
3.1 Neural Networks 15
3.1.1 Artificial Neural Networks 16
3.2 Training 21
3.2.1 Error Quantification 21
3.2.2 Backpropagation 22
3.2.3 Weight Initialization 25
3.3 Convolutional Neural Networks 26
3.3.1 The Convolution Layer 27
3.3.2 Pooling Layer 28
3.3.3 Example Architecture 29
3.3.4 Transfer Learning 30

4 Methods **31**
4.1 Software 31
4.2 Hardware 32
4.3 Neural Network Models 32
4.3.1 VGG-19 33
4.3.2 Custom Models 34
4.4 Data Generation 35
4.4.1 Data Pre-Processing 36
4.5 Training Process 37

5 Experiments **39**
5.1 Unprocessed Data 39
5.2 VGG-19 Models 40
5.2.1 Classification 40
5.2.2 Regression 41
5.3 Custom Models 42
5.4 Importance of Large Data Sets 43

6 Discussion, Conclusions and Future Work **45**
6.1 Discussion 45
6.2 Conclusions 46
6.3 Future Work 46

References **47**

Appendices **56**

A Detailed Results **57**
A.1 Unprocessed Data 57
A.2 VGG-19 Classification Models 58
A.3 VGG-19 Regression Models 59
A.4 Custom 60
A.5 Importance of Data 61

1 Introduction, Motivation and Goals

With the evolution of high performance computing (HPC) moving towards the exascale era, something which entails more complex hardware and often utilizing heterogeneous clusters, the development of suitable high performance applications will become more difficult as well. To optimize these applications, profiling and tracing tools will become even more important. Though the utilization of such tools and understanding of any anomalies typically require some work and expert knowledge.

In recent years different deep learning techniques have enjoyed great success in various fields, and are good at identifying structures from high dimensional data. Though they have still not entered the area of application performance evaluation. It would be interesting to see how they can be applied to this area.

This thesis aims to act as a proof-of-concept with the objective of introducing deep learning techniques to the area of performance evaluation of HPC applications. More specifically, this thesis will combine the tools developed at BSC with state-of-the-art techniques within the image recognition field, which will facilitate the initial analysis process, removing the need of an expert.

With one of the pillars on which Paraver is built upon being the ideology of visual inspection being the first step in an analysis, this angle will be approached from a deep learning standpoint. The recently very successful deep learning - computer vision - technique of convolutional neural networks will be utilized on timeline images generated with Paraver, to predict a number of performance metrics for the application in question.

This thesis investigates whether deep learning techniques could be successfully used for performance analysis by experimenting with different deep learning models and addresses the following questions:

- Will models primarily designed for other tasks be a good starting point or will a custom model for this task be more suitable?
- Will better results be achieved by pre-processing the input images?
- Is a lot of data essential for the models to perform well?

1.1 High Performance Computing

The values brought by supercomputers have become indispensable for governments, scientific researches and enterprises when it comes to generating new discoveries and innovative breakthroughs for products and services. Not only are supercomputers contributing to the scientific progress, but they are also aiding in things like national security, industrial competitiveness, and overall quality of life[34].

The broad spectrum of industries utilizing supercomputers include, amongst many others:

Aerospace

- Aircraft manufacturers such as Airbus and Boeing are able to significantly reduce the design-to-production timeline when launching new aircrafts by utilizing supercomputers, something which have translated into savings of tens of billions of dollars for the aerospace industry[34]. To put the need for for HPC in the production of aircrafts into context, a large passenger jet has got well over two millions individual parts, that need to be simulated individually and as part of a larger system. For this reason Airbus has got no less then three own supercomputers, all listed on the Top 500 list[25].

Automotive

- In the same way as the aerospace sector has benefited greatly from HPC, so have the automotive sector. General Motors is one of the companies which relies in HPC in their development process. The tools used by GM allow their engineers to make crash tests by performing simulations from every angle with tests ranging from airbag performance to pedestrian safety[9]. The IDC[1] made a study estimating that HPC has enabled the European automakers to reduce development time while still greatly improving the crashworthiness, environmental friendliness and passenger comfort of vehicles from 60 months, down to 24 months[44].

Energy

- Supercomputers are transforming both how energy is produced and how it is consumed[34]. With HPC being used to make building more energy efficient, as well as being fundamental in the research for more effective energy production there are now projects like HPC4E[2] dedicated for these questions[12].

[1] International Data Corporation
[2] HPC for Energy

Health Care

– Another field which has made revolutionary progress thanks to HPC is health care. For example, HPC-powered next-generation sequencing techniques have reduced both the time and cost of a complete human genome sequencing down to just a few days, with a cost as low as $\$1,000$. Something which, just 15 years ago, would have cost over $1 billion, and many person-years of work[23]. Similarly, researchers have now been able to model the human heart down to the cellular level, using one of the world's most powerful supercomputer[47].

While it is clear that scientific researchers benefit extremely from HPC, the use of supercomputers reach further than that. With at least one MLB[3] team utilizing supercomputers to evaluate how the different batters hold up against different types of pitchers[21]. And with the movie industry highly rely on supercomputer for animation of movies. In fact, media and entertainment make up around 10% of the overall HPC server business[8]. The fact is that results of HPC is all around us, even the creation of Pringles potato chips utilizes supercomputers[11]. With the general awareness of the importance of HPC being quite low, even though HPC plays an important role to the quality of our lives[14], the organizers of the yearly Supercomputing Conference[4] have launched an awareness campaign under the hashtag #`HPCMatters` where the industry share their stories of how HPC has transformed how they work[13].

The benefits of HPC and parallel programs are evident, but the creation of such programs is much harder than the creation of sequential programs[58]. Things developers for parallel programs need to consider include among others; how to divide the calculations, how to partition the data and where to place it, how the different processes communicate with each other.

1.2 Performance Evaluation

With exascale computers around the corner, the ASCAC[5] subcommittee on Exascale Computing has identified some challenges for reaching such extreme-scale computing[29]. One of the challenges was to have exascale ready applications. And for application to perform efficiently at such scales, tools for evaluating their performance are of great importance.

The European Commission has funded POP CoE[6] which provides services for

[3]Major League Baseball

[4]A yearly supercomputing conference for High Performance Computing, Networking, Storage and Analysis sponsored by ACM and IEEE Computer Society `http://supercomputing.org/index.php`.

[5]Advanced Scientific Computing Advisory Committee

[6]POP: The Performance Optimisation and Productivity Centre of Excellence in Computing Applications

academia as well as industry in the EU to help improve the performance of their HPC applications[16]. The POP project consists of different partners across Europe[7] providing teams with, among other areas, excellence in performance tools and tuning. There are a number of commercial performance tools available such as the Intel tool-set[8], CrayPat[9], and Allinea tools[10]. But there are also a number of open-source tool-sets developed by POP partners and collaborators, such as:

Paraver

Paraver is a tool utilized together with *Extrae*, both developed at BSC, which is based on the philosophy that visual inspection should be the first step when capturing a program's characteristics and behavior. Atop of a good visual view, Paraver provides a complementary statistical view which offers a different viewpoint of the application under analysis (Section 2.2 covers a more in-depth introduction to Paraver, and the BSC tools parallel performance analysis suite).

Scalasca

The *SCalable performance Analysis of LArge Scale Applications* (Scalasca) toolset, is a joint[11] German project[20]. Scalasca is an analysis tool for a post-mortem analysis of trace files created with the independent Score-P measurement system[19]. The analysis with Scalasca identifies potential performance bottlenecks, in particular those concerning communication and synchronization, and offers guidance in exploring their causes.

Even though there is no lack of available tools, the use of such tools are not very widely spread, with manual code instrumentation (i.e. inserting code for time measurements) still being the most common way of performance analysis. With one main reason for this being the inhibitive complexity of getting started with these tools[45].

[7] BSC (Spain), USTUTT (Germany), Jülich (Germany), NAG (UK), RWTH Aachen (Germany), TERATEC (France)

[8] Trace Analyzer and Collector[26], VTune Amplifier[27]

[9] Cray Performance Measurement ans Analysis Tools [6]

[10] MAP[1], Performance Reports[2]

[11] Forschungszentrum Jülich - Jülich Supercomputing Centre, Technische Universität Darmstadt - Laboratory for Parallel Programming, and German Research School for Simulation Sciences - Laboratory for Parallel Programming

1.3 Deep Learning

Deep Learning is a very hot area which has managed to drastically improve the state-of-the-art in multiple different AI tasks. It turns out that deep learning techniques are very good at identifying intricate structures in high dimensional data, something which has made it applicable in many domains of science[51]. It has not only beaten records in typical AI problems such as image recognition[49, 72], and speech recognition[66]. But it has also shown to be successful at predicting the activity of potential drug molecules[55], reconstructing brain circuits, language translation[71], analyzing particle accelerator data[10], and much more.

In short, deep learning is a machine learning genre inspired by the neurons in the visual cortex of the brain work, where certain neurons react to different scenarios, often through the weighted connection to previously activated neurons. The combinations of these neurons can be used to represent very complex scenarios. The weighted connections are not hand-designed, but rather learned through a learning process. A technique which allows for the models to improve with experience and data[40].

The origins of deep learning could arguably be traced back to 1873 when Alexander Bain introduced the neural groupings as the first models on neural networks (or even back to 300 BC when Aristotle began the attempts to understand the human brain)[74].

Deep learning techniques have since been limited by the computational resources and data sets available for training[62]. Though with GPUs[12] constantly improving and now being used to perform work beyond just graphics, the training of deep learning models can now be done in a more manageable time. And has recently become very popular again, arguably with AlexNet[49] winning the ILSVRC (ImageNet Large-Scale Visual Recognition Challenge)[13] back in 2012 (with groundbreaking results), as starting point.

1.4 Related Work

While, to the best of my knowledge, no work utilizing deep learning models for analyzing the performance of HPC applications have been done before, there exist works with flavors related to the work performed in this thesis.

Clustering, an unsupervised machine learning technique was utilized in [39] to group the computational regions with similar behavior into cluster for further grouped analysis. They utilize the DBSCAN[14] algorithm which is a density

[12] Graphical Processing Units

[13] The ImageNet challenge is evaluating algorithm for object detection and image classification at large scale. The data set includes over 1.3M images in 1000 categories. [65]

[14] Density-Based Spatial Clustering of Applications with Noise[33]

based algorithm which, from a 2D performance space (such as, instructions completed and IPC), based on the proximity of the behavior of computations, group them into clusters. Some practical uses of cluster analysis include extrapolation of performance data and reduction of input data for multilevel simulations.

In [54], Llort et al. turned to the computer vision field in their pursuit of insight on how the behavior of parallel applications evolve for different scenarios where the conditions for the execution change. They extended the work done in [39] and utilized a tracking algorithm to track and analyze how the behavior of computational clusters change for different scenarios. With this tracking technique it is possible to see how different factors (such as hardware, software versions, scales and program configurations) affect the performance of the application.

Efforts have also been made towards automating the analysis process, similar to this thesis, though using a numerical approach for obtaining the performance metrics. In [63] the authors extract performance metrics, utilizing Paraver, for multiple core counts and further provides a model for extrapolating the metrics and to predict their evolution with scale.

1.5 State of the Art CNN Architectures and Applications

The development of convolutional neural networks (see Section 3.3) is rapidly improving every year, a good example of this can be seen by looking at the winners for the ImageNet challenge for the last years. Figure 1.5.1 shows how the winners in the classification category performed over the last 5 years, with last years winners *Trimps-Soushen*'s model achieving a 2.99% top-5 error. A trend which many of the newer models follow is that the models are becoming deeper. AlexNet (winner of ILSVRC2012) is built up by five convolutional layers and three fully connected layers, with the winners of 2015 presenting one ensemble of 152 layers[42].

The convolutional neural networks are also used to achieve state of the art and groundbreaking results in different applications. In [56], the authors utilized CNNs for the image question answering task, and are able to answer questions such as: *What is the largest blue object in this picture?* and *How many pieces does the curtain have?*

The authors of [53] are utilizing convolutional networks for detecting extreme weather in climate data sets. A task which normally requires human expertise in defining events based on physical variables, but is now achieved with convolutional neural networks with an accuracy between $89 - 99\%$.

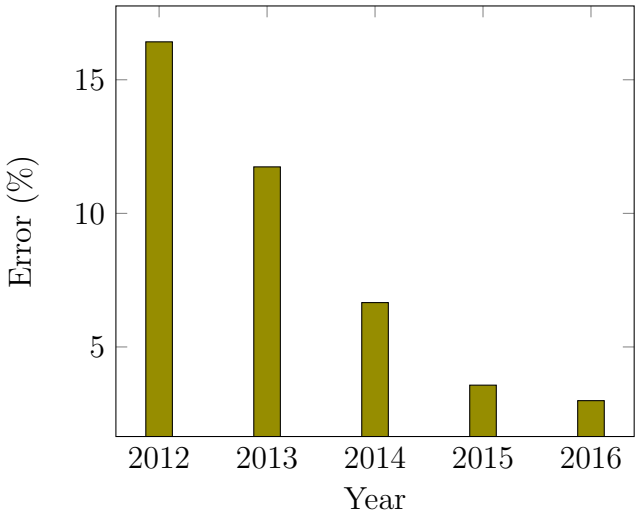


Figure 1.5.1: Improvement of the top 5 error for the classification task of the ImageNet challenge over the years.

1.6 Paper Outline

The outline of the rest of the thesis is the following: Chapter 2 covers the performance metrics sought in the the experiments as well as a more in depth explanation of the performance tools utilized.

Chapter 3 describes the deep learning techniques utilized in obtaining the aforementioned performance metrics.

Next, Chapter 4 explains the techniques, together with the software and hardware, used for the experiments.

Chapter 5 presents the experiments performed and their results in a flowing fashion.

Finally, conclusions and a discussion together with recommendations for future work are given in Chapter 6.

2 Performance Evaluation

An introduction to performance evaluation for HPC systems was given in Section 1.2. This chapter will give a more thorough explanation of the concepts used in this thesis. First the performance metrics used in this work are introduced, followed by a more detailed introduction to the performance tools this work are utilizing.

2.1 POP Performance Metrics

The pursuit of optimizing performance of a parallel code can be an intimidating task, and it is often hard to know where to start. The communication pattern for the application might be inefficient? Or perhaps the computational distribution is not very well divided? To provide guidance for questions like these POP has defined a methodology for analysis of parallel codes, which provides a quantitative measurement of the relative impact of the factors inherent in parallelisation[17].

The POP methodology is using a hierarchy of metrics where each metric represent a common cause of inefficiency for parallel programs. The metrics make it possible to compare the parallel performance across different scenarios (e.g. different core counts, different input, etc.) to identify potential weaknesses of the code. The metrics are calculated as efficiencies in the range between 0 and 1, where a higher number indicates a better efficiency (which also can be presented as a percentage $0 - 100\%$). As a general guideline, an efficiency above 0.8 can be regarded as acceptable, whereas values lower than that indicate performance issues where a deeper look is needed.

The hierarchical structure of the metrics can be seen in Figure 2.1.1. At the very top is the *Global Efficiency* which can be used to assess the overall quality of the parallelisation of the application. There are typically two main causes of inefficiencies in parallel applications, which lead to the next level in the hierarchy.

The Global Efficiency is calculated by taking the product of the *Computation Efficiency* and *Parallel Efficiency*. The Computation efficiency metric represents how well the application is scaling and is calculated by computing the total time spent performing useful computations[1] across all processes. In a

[1]Useful computation refers to the time the application is performing actual computations

strong scaling scenario the sum of the useful computation time should be the same at all scales for a perfect efficiency. Two possible causes for poor Computation Efficiency are that dividing the work increases the total amount of computations required, and that the use of additional processes lead to contention on shared resources. These can be examined through the *Instruction Efficiency* and *IPC Efficiency*, respectively. Whereas they do not directly make up the Computational Efficiency, they can provide good insight.

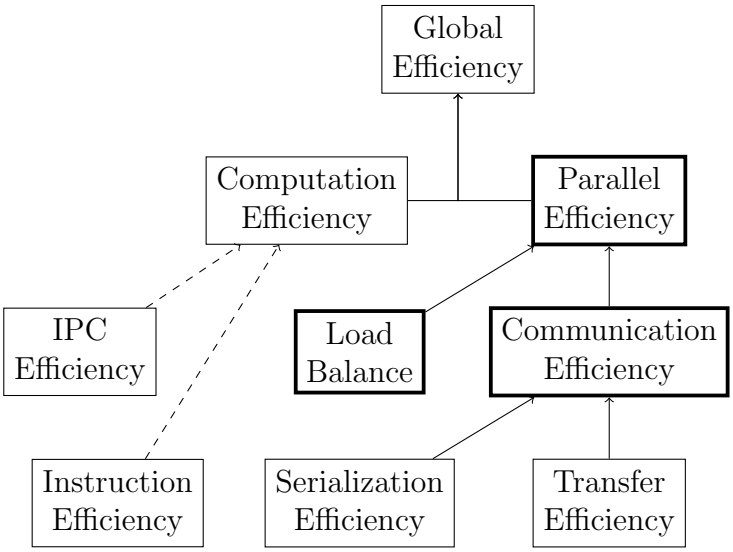


Figure 2.1.1: Hierarchical structure of the POP metrics, with directly affecting dependencies shown with whole arrows, and indirect dependencies with dashed arrows. The three metrics important for this thesis are shown in bold.

The Parallel Efficiency exposes the inefficiencies caused by dividing the workload over processes and then communicating in between them. Similarly to the Global Efficiency, the Parallel Efficiency is defined as the product of two metrics lower in the hierarchy. In this case those metrics are *Load Balance* and *Communication Efficiency*. The Load Balance tells how good the distribution of the workload is, and is calculated as the ratio between the average time each processor is performing useful work and the maximum time.

$$LoadBalance = \frac{\sum_{p=1}^{n}(UsefulComputationTime_p)/n}{MAX(UsefulComputationTime_n)} \tag{2.1}$$

The Communication Efficiency identifies applications which suffer because they spend too much time communicating, instead of performing useful work. The communication efficiency is defined as the ratio between the processor spending the most time performing useful work and the total runtime.

which progresses the program, and not, for example, time spent waiting in a barrier.

$$CommunicationEfficiency = \frac{MAX(UsefulComputationTime_n)}{TotalRuntime} \quad (2.2)$$

From the Communication Efficiency metric it is possible to compute two sub-metrics which breaks it further down. The metrics are *Serialization Efficiency* and *Transfer Efficiency*, but to derive these metrics a simulation of an ideal network[2] is required. By comparing the total runtime on a simulated ideal network and the real network we can measure the inefficiencies due to data transfer, which is quantified with the Transfer Efficiency metric. Once the Transfer Efficiency is calculated, it is straight-forward to compute how much of the communication inefficiency comes from processors waiting for other processors to be ready to communicate. This is computed by dividing the Communication Efficiency by the Transfer Efficiency, and is finally quantified in the Serialization Efficiency metric.

2.2 BSC Tools Parallel Performance Analysis Suite

While there exist different tools for performance evaluation, in this thesis the tools from BSC tools[4] are used. The main tools used during this work include: *Extrae* - for creating trace files; and *Paraver* - for visualizing and obtaining metrics form the traces.

2.2.1 Extrae

The Extrae package is devoted to generating the trace files of the applications which is later used by Paraver and Dimemas[3]. Extrae is able to intercept calls to parallel runtimes, such as MPI, and creates a sequence of time stamped records, which are collected in a trace file for post mortem analysis. Extrae can also capture when and where entries and exits to parallel runtimes occur, at nanosecond precision and with minimum overhead. Extrae can capture much more information than that, such as IPC and cache misses (by utilizing PAPI counters[5]), who communicates with who, and from where in the code a certain call was made. For this thesis the above mentioned features will suffice.

The trace files generated are made up of three types of records:

State: A record which associates a state value for a thread during a certain time interval. Paraver is not associating any semantics to the encoding of the state field, allowing for great flexibility to take on any value. Examples of state values are; *Running* - the thread is performing normal computations; *Waiting*

[2]In an ideal network the communication occurs instantly.

[3]Dimemas - a simulation tool to answer "what if" questions. Will not be used in this work.

- the thread is idle, waiting for some criterion (e.g. waiting for other threads to reach a barrier); *Not yet created* - the thread is not included in the set of working threads (e.g. in the beginning of an application before the master thread has created its slaves).

Event: A record representing a punctual event occurring at a certain time for a specific object. Just as for the *state* record, Paraver is not associating any semantics to this record, and they are simply encoded as two integers, *type* and *value*. Examples of events recorded by Extrae include, *cache misses* for different levels, *instructions completed* and *entry/exit points* of MPI routines, user functions, OpenMP parallel regions and CUDA kernels.

Relation (Communication): A record establishing a relationship between two object in two points in time. Examples for relation records include communications between processes in MPI applications, movement of tasks between threads in OmpSs applications, and memory transfers in CUDA or OpenCL applications.

2.2.2 Paraver

Paraver is a visual API, a browser, for traces generated with Extrae. Key features in the design of Paraver were: expressive power, flexibility and capability to efficiently handle large traces. The modular structure of the tools helps achieving this goal.

Two of the main design choices on which the analytic power of the tool stand on are its lack of semantics and hardwired metrics. Not having fixed semantics allows for extending the tools to support new performance data or programming models without any requirements of changing the visualizer, capturing it in a Paraver trace is enough. In a similar fashion, with the metrics not being hardwired, but instead programmed, a huge number of metrics may be displayed. To compute the metrics the tool offers a large set of time functions, a filter module as well as a mechanism which allows for combining two *timelines*. Any metrics once computed can easily be saved in configuration files which makes it as simple as loading the saved file when the metric is to be used again.

Paraver offers a minimalistic set of views, with the philosophy that, different views should provide qualitatively different types of information. The two views available are a *timeline* view and a *statistical* view. The timeline view represents the behavior along time and processes, with each window displaying a single view; a piecewise constant function of time. The types of functions displayed fall into three groups; Categorical, Logical, and Numerical. The *categorical* functions show things such as, in which state the thread is in. The categorical functions take on an integer value $[0, n]$ (where n is a small number). The *logical* functions display information such as, whether the thread is in a specific user function or MPI call. The logical functions are mapped to the

set $\{0, 1\}$. Finally, the *numerical* functions display things like the duration of an MPI call, or the number of cache misses. The numerical functions can take on any real value. An example of a timeline view can be seen in Figure 2.2.1. The functions can be represented in different ways in the views, where two of the most common ways are *color encoding* (where each functional values is represented with an individual color) and *Not null gradient* (where the functional values are represented according to a gradient scheme with low values taking a light green color and high values a dark blue color. Zero values are represented with a black color).

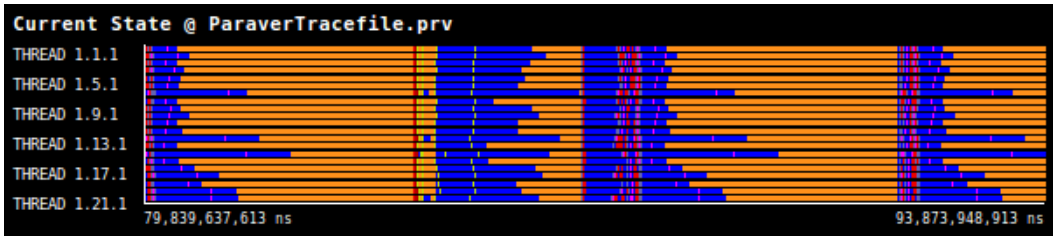
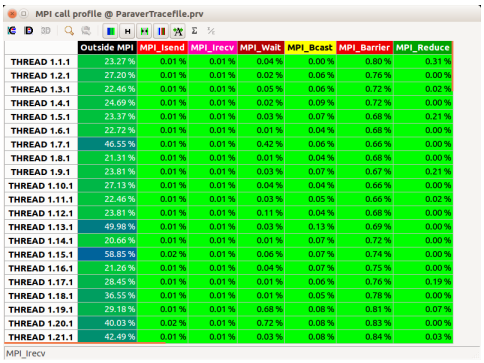
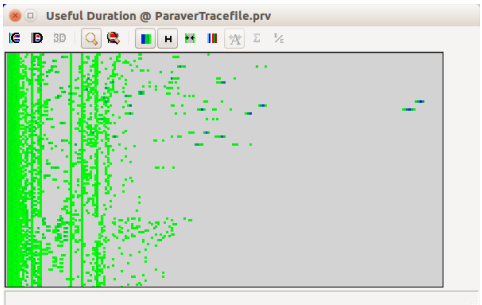


Figure 2.2.1: Paraver timeline view, with the different threads showing on the y-axis and time on the x-axis. The different colors represent different states of the application.

The statistical view provides numerical analysis of the traces. The statistical view can be illustrated as both tables and histograms (both in 2D and 3D), Figure 2.2.2 shows an example of both a table view as well as a histogram view. The table view allows for analyzing statistics for specific categorical control values (such as different MPI calls, or different user functions). Relevant statistics to display include time (in seconds or percentage) spent in each call, number of calls, average duration of calls, etc. The histogram view displays bins of numeric control values. Each bin represents a range of values, which allows for an analysis providing insight on things such as how many instructions the different computation bursts perform for different threads, which IPC value is most common, etc.



(a) Table view



(b) Histogram view

Figure 2.2.2: Statistical views for Paraver traces.

With trace files potentially being fairly large, Paraver offers different manipulation methods for reducing the data down to a more manageable size. The traces can be *filtered*, only keeping the relevant data. Such filtered traces could, for example, be generated such that it only shows computations larger than a certain threshold, or only showing runtime related calls (removing information about hardware counter, etc.). A feature for *cutting* the traces exists as well, keeping all information but only for a certain time interval, or for certain processes. Typical reasons for cutting the traces are: removing initialization and finalization periods (typically not of interest) and for analysis of local behaviors not globally representative.

The Paraver kernel also provides a non-graphic interface, *Paramedir*. Paramedir can read traces, apply standard configurations and write a human readable ASCII output. The non graphical version is very useful in scripts and batch processing.

2.3 Analysis Process

Many applications tend to have a repetitive behavior, and often a few iterations are doing a good job at representing the global behavior of an application (though within each iteration there might be many regions with separate behaviors). Reducing the traces to fewer iterations could then significantly increase the manageability of the traces while still providing a representative analysis. Something which is exploited in this work, by fitting one iteration into a single timeline view.

The initial search space for the analysis would then be the metrics defined in Section 2.1. Once the fundamental factors for the performance have been identified, a deeper analysis can be made towards finding causes for any inefficiencies.

3 Deep Learning

This chapter introduces some basic background knowledge needed, for the unfamiliar reader, to understand the concepts used in this thesis. This chapter begins by introducing some basics about neural networks, followed by explanations on the training process, and ends with covering convolutional neural networks.

3.1 Neural Networks

Neural networks, under which deep learning falls, have taken its inspiration from the human brain, which can be described as a biological neural network. The principles embraced are how the brain is able to process information and train itself. Much about the human brain remain a mystery, but the computer scientists work on the understanding that the brain works by having neurons collecting signals from other neurons through a number of fine structures called *dendrites*. The neurons are sending out electrical activity of different strengths along an *axon*, a long thin stand. The axons then splits to thousands of small branches, at the ends of which there is a *synapse*. The synapse takes the activity coming from the axon and converts it into an electrical effect, which can then either excite, or inhibit the connection to the receiving neurons. Would the inputs to a neuron be excitatory large enough compared to its inhibitory input, the neuron would "fire" and electrical activity down its axon. Figure 3.1.1 illustrates this concept. The learning process then consists of altering the effectiveness of the synapses, in such that the influence of a certain neuron on another changes.

Neural networks have many noteworthy characteristics, which make them an attractive solution for machine learning. Their curious ability to be able to take complicated or imprecise data and obtain meaning from it, is very useful when trying to identify patterns and trends which would otherwise be too complex for a human or more conventional computer techniques to see. Furthermore, its ability to learn how to perform tasks based on the data given and to create its own representation and organize what it has learnt are all advantages which are appealing for machine learning.

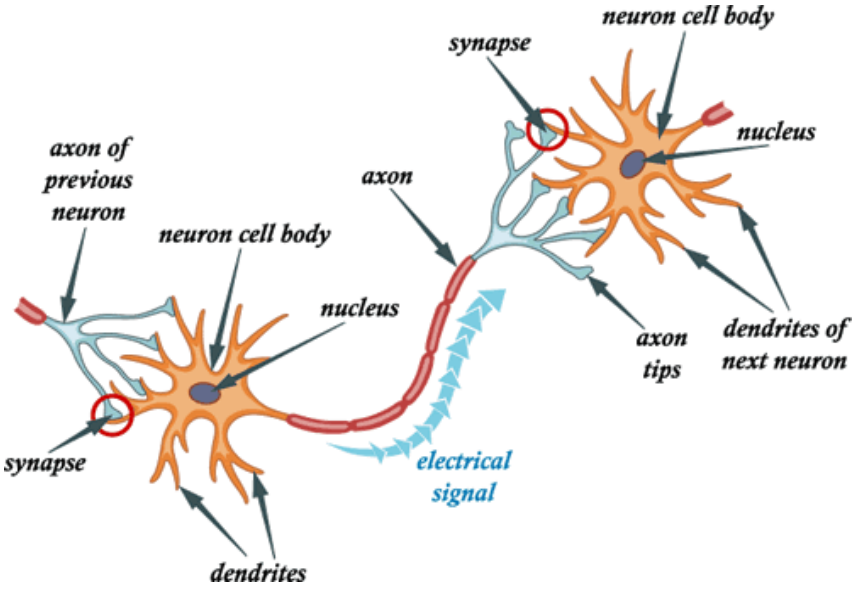


Figure 3.1.1: Illustration of how the communication between neurons occur.[1]

3.1.1 Artificial Neural Networks

The artificial equivalent of the biological neuron was first mathematically defined in 1943 by Warren McCulloch and Walter Pitts, and is known as the McCulloch-Pitts-Neuron, or MCP neuron[57]. The MCP neurons work under certain assumptions which can be summarized with the McCulloch-Pitts output rule:

$$N_i = \begin{cases} 1 & \sum_j W N_j \geq \theta \textbf{ AND } \text{ no inhibition} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

and visually in Figure 3.1.2. The rules say that each neuron is a binary device with a fixed threshold, theta. That the weights (W) of the excitatory inputs are all of identical weight and how the inhibitory input possesses the absolute veto over all excitatory inputs.

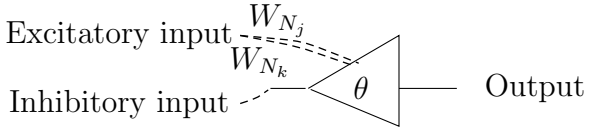


Figure 3.1.2: Illustration of the McCulloch-Pitts output rule.

While the MCP neuron was an important definition as it introduced how neuron-like elements could execute, it had no learning capabilities, so it was

[1]Image from `http://www.mysearch.org.uk/website1/html/106.Connectionist.html`.

basically a hard coded device. The next major stepping stone came in 1958, with the introduction of the *Perceptron*, which is similar to what we are using today[64]. There are some important differences between the perceptron and the MCP neuron. The weights and thresholds for perceptrons are not all identical, and the weights can take-on both positive and negative values. The inhibitory synapse is removed, but most important of all, the perceptron has got a learning rule. The perceptron can be defined as

$$N_i = \begin{cases} 1 & u_i > 0 \\ 0 & u_i \leq 0 \end{cases} \tag{3.2}$$

where

$$u_i = \sum_j W_j N_j + \theta_j \tag{3.3}$$

as well graphically in Figure 3.1.3, with W_j, N_j and θ_j being the weight, input and bias of input neuron j, respectively.

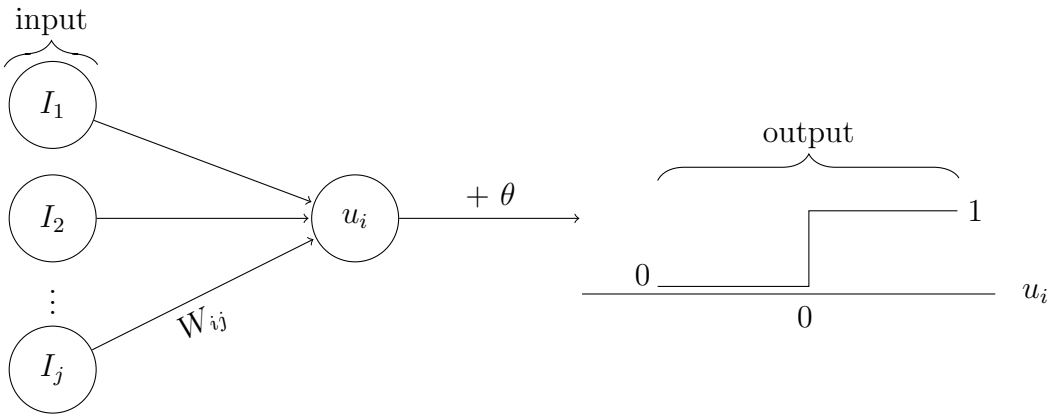


Figure 3.1.3: Illustration of the behavior of a perceptron. A neuron u_i takes j inputs, sums them together and adds a bias θ. If the result is higher than a threshold, 1 is returned, otherwise 0.

The learning schema for the perceptron fairly is straightforward. Let δ be the difference between the desired target and the output (N) of the perceptron for a given input. Then the change in the weight is chosen such that it remains proportional to delta. For a certain learning rate $0 < \epsilon < 1$, we then have:

$$\Delta W = \epsilon \delta N. \tag{3.4}$$

3.1.1.1 Feed-Forward Neural Networks

The most basic use of a perceptron would be to have two inputs and a single output, which could, for example, classify which of the inputs is larger. This is an example of a *single-layer feed-forward* neural network (like Figure 3.1.3 illustrates), where the information is only flowing in one direction, different from *recurrent* neural networks[2] where the information can flow backwards too, recurrent neural networks will not be further mentioned in this thesis.

Single-layer perceptrons can have more than just two input and a single output. They can have an arbitrary number of both and can be used for linear regression. Though the single-layer perceptron is only able to distinguish linearly separable pattern, illustrated in Figure 3.1.4, and in 1969 it was shown that it was incapable of learning an *XOR*-function[3][59].

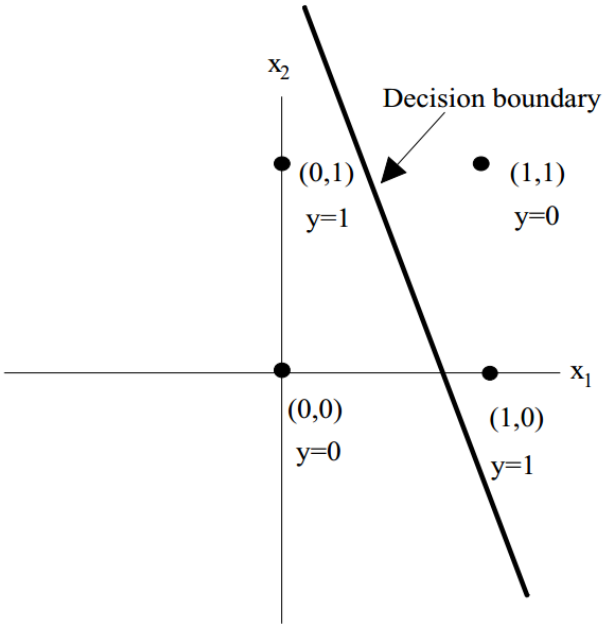


Figure 3.1.4: Limitations of the single-layer perceptron.[4]

The shortcomings of single-layer neural networks can be addressed with *multi-layer* neural networks which adds one, or multiple, extra layers between the input and the output layers. These extra layers are referred to as *hidden layers*. Figure 3.1.5 illustrates a multi-layer feed-forward network with one hidden layer. In fact, in 1989 it was showed in the first version of the *universal approximation theorem* that a feed-forward network with only a single hidden

[2]Recurrent neural networks create an internal memory as connections creates cycles, something which makes them very useful for processing sequences of inputs for tasks such as, speech recognition, translation and self-driving cars.

[3]Exclusive Or ($\oplus$): $a \oplus b = (a \vee b) \wedge \neg(a \wedge b)$

[4]Image from `http://www.cs.nott.ac.uk/~pszqiu/Teaching/G53MLE/ffnets-note.pdf`.

layer could approximate any continuous function $f \in \mathbb{R}$, although it mentions nothing about its learnability[43].

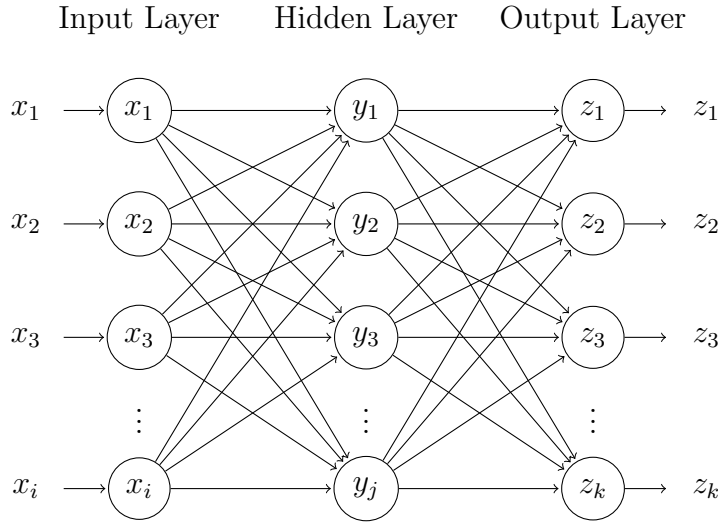


Figure 3.1.5: Illustration of a multi-layer feed-forward neural network with i neurons in the input layer, j neurons in the single hidden layer, and k neurons in the output layer.

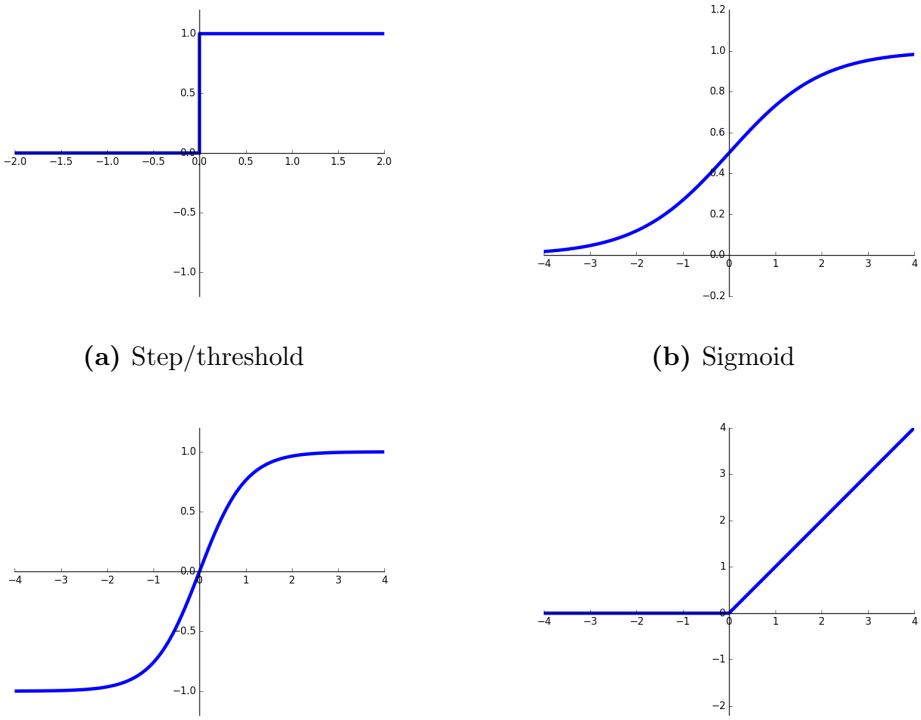
3.1.1.2 Activation Functions

The original MCP neuron defined the output as one if a certain condition was met and zero otherwise. That is, the MCP neuron *activates* if a certain condition is met, according to its *activation function*. The MCP neuron had only a single activation function defined, namely the *step/threshold* function, see Figure 3.1.6(a).

But since the MCP neuron a number of activation functions have become popular. The activation functions provide a crucial role in the neural networks as they add *non-linearity* to the networks, without which none of the complex task could be solved. Different activation function posses different properties making some more suitable than others. The historically most popular one is the *sigmoid* non-linearity, which can be expressed as:

$$\sigma(x) = 1/(1 + e^{-x}) \tag{3.5}$$

and illustrated in Figure 3.1.6(b). The sigmoid function takes on values "squeezed" into the range $[0, 1]$, where very large negative numbers takes on zero and large number takes on one. This was a nice interpretation of how the firing rate of a biological neuron was perceived to behave. Though in practice it has some major drawbacks which make them impractical during training (training explained in Section 3.2), something which has lead to them rarely ever being used any longer.



(a) Step/threshold

(b) Sigmoid

(c) Tanh

(d) ReLU

Figure 3.1.6: Illustrations of different activation functions.

A non-linearity which deals with some of the drawbacks of the sigmoid is the *tanh* non-linearity. A tanh neuron is actually just a scaled sigmoid neuron which maps the input to a number in the range $[-1, 1]$, which actually solves one of the big issues of the sigmoid, and for that reason it is in practice always preferred over the sigmoid. Tanh can be seen in Figure 3.1.6(c) and can be expressed as

$$tanh(x) = 2\sigma(2x) - 1. \tag{3.6}$$

Though the most popular activation function the last few years has been the $ReLU$ (Rectified Linear Unit) which simply thresholds the input at zero

$$f(x) = max(0, x), \tag{3.7}$$

as illustrated in Figure 3.1.6(d). The ReLU non-linearity solves the big common issue with the tanh and sigmoid, namely the problem with *vanishing gradients*[5], and is also significantly faster, as no expensive computations are needed (exponentials, etc.). Though the ReLU neuron is not without its issues,

[5] A well known problem, where the gradient for the earlier layers become so small, where even a large change in the input would cause a small change in the output.

it turns out that it is fairly brittle during training. A large enough gradient flowing through a neuron can cause updates in such that they never will change again, and in effect "dying". One may find that as much as 40% of the neurons are "dead"[7].

There are multiple attempts to deal with the dying ReLU problem, with varying success, such as, setting a roof, Γ, on the output:

$$f(x) = max(0, min(x, \Gamma)), \tag{3.8}$$

or like the $CReLU$ which concatenates a positive and a negative ReLU:

$$\rho_c \triangleq (max(0, x), max(0, -x)), \tag{3.9}$$

or the *leaky ReLU*, which allows negative values too, though only at a fraction (α) of the magnitude of the positive ones:

$$f(x) = \begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases} \tag{3.10}$$

3.2 Training

The feed-forward networks described have shown good results on many tasks, but for them to perform adequately their weights need to be properly set. The original MCP neuron required manual definition of the weights. Though this can now be done through a training step, where the model *learns* appropriate values for the weights.

3.2.1 Error Quantification

For the network to be able to learn, a way to quantify how good/bad a network performs is needed. This is done by defining a *loss function* suitable for the task in question. There are a number of different loss functions, both for *classification* problems and for *regression* problems.

Classification problems try to determine to which class a certain data belong to. This could be, for example, to determine whether an animal is a dog or a cat. Some popular loss functions for classification problems include the *Softmax* classifier and the *Sigmoid* classifier. The Softmax classifier calculates a probability distribution for the classes as the following equation shows.

$$S_i = \frac{e^{f_{yi}}}{\sum_j e^{f_j}} \tag{3.11}$$

Equation (3.11) normalizes the class scores f to fit in the range 0-1. Softmax is typically used together with *cross-entropy* to then get a sense of how "off" this prediction is from the ground truth. With the probability distribution vector S and the ground truth one-hot encoded[6] label vector L we then define the cross-entropy loss as:

$$C(S, L) = -\sum_i L_i log(S_i). \tag{3.12}$$

Whereas the classification problems try to put the input into a certain category, the regression problems are trying to predict a real-valued quantity. It could be, for example, the price of a house. For regression problems, it is common to compute the difference between the predicted values f and the ground truth y. The L1 norm or the L2 squared norm would then measure on differences. As will be explained in subsection 3.2.2 it is common to compute multiple examples in one "batch". One would than take the mean error over all examples, so for the mean L2 squared norm that would then be:

$$\bar{L_2} = \frac{\sum_{i=1}^{n} (f_i - y_i)^2}{n} \tag{3.13}$$

which also goes under the name, mean squared error (MSE).

3.2.2 Backpropagation

The backpropagation phase of neural network is where the actual learning occurs. In backpropagation, the error of the predicted output is calculated, and then propagated backwards through the network, hence the name *backpropagation*.

The goal of the backpropagation is to tune the weights in such a way that the *loss function* is minimized. This is achieved by taking the derivative of the equations in the model and following the gradient in the direction which minimizes the error. This method of following the gradient to minimize a problem is called *gradient descent*; a simple example of its usage can be seen in Figure 3.2.1. This work will not go into details on how gradient descent works, for more detailed explanation the reader is directed to [40].

Something which separates machine learning algorithms from the general optimization algorithm is that the objective function is typically made up of a sum over the training data. There are different approaches to manage this, where the straightforward thing would be to compute the gradients for all training data in each training step, an approach called *Batch Gradient Descent*. Whereas computing the gradient for the entire training set would certainly

[6] In a one-hot encoding, a vector with as many elements as there are classes have one zero values, except for the correct category which has value one.

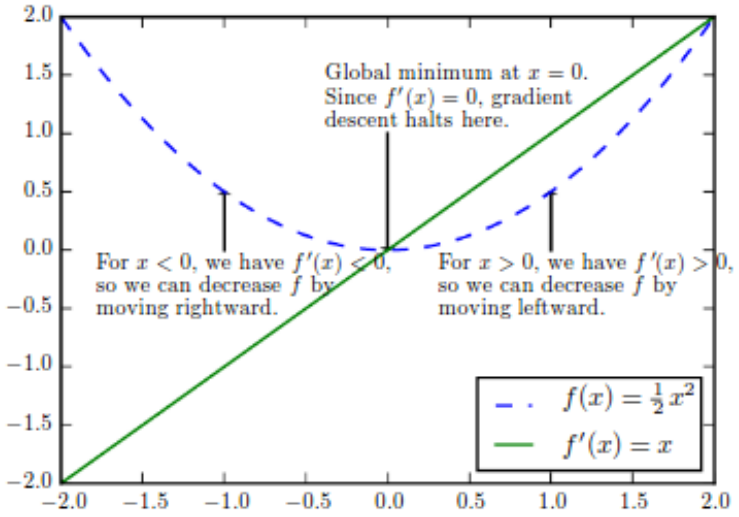


Figure 3.2.1: Example behavior of gradient descent.[7]

lead to the most correct update, these computations are very costly and it has been shown that the improvements in the calculated gradients are far from linear with the increase of data used during the calculations.

Another approach would then be to only utilize a single training example per step, a method called *Stochastic Gradient Descent* (SGD). SGD might be the method which gives most value for each computation, though it could lead to very abrupt changes when encountering outliers. It is also unable exploit the parallel capabilities provided by GPUs, a parallelity which has been vital in the recent popularity of neural networks.

Instead the most common approach is to use a small, randomly sampled, subset of the training data for each training step, a method known as *Mini-batch Gradient Descent*. The number of training images chosen for each training step, the *batch size*, will vary widely, but are driven by the following considerations[40]:

- Larger batch sizes will result in a more accurate gradient estimate, though with less than linear returns.
- GPUs tend to be underutilized by very small batches, leading to a minimum batch size below which no reduction in time will be witnessed by reducing the batch size.
- The memory requirements tend to scale with the batch size, and is often the limiting factor for GPUs when increasing the batch-sizes.
- Some architectures show improved performance when choosing batch-sizes as powers of two.

[7] Image from [40].

While the gradient descent strategy is a popular one for optimization, it can be slow. To speed up the learning, a technique called *momentum*[61] could be added. The momentum algorithm uses past gradients to calculate an exponentially decaying moving average and continues to move in its direction. The concept of momentum is illustrated in Figure 3.2.2.

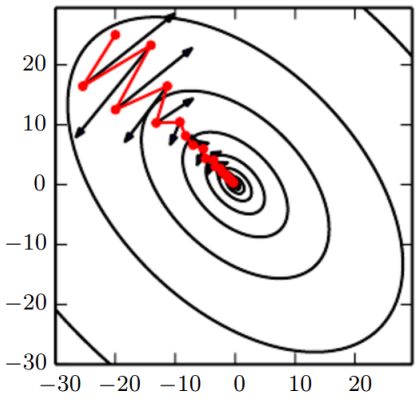
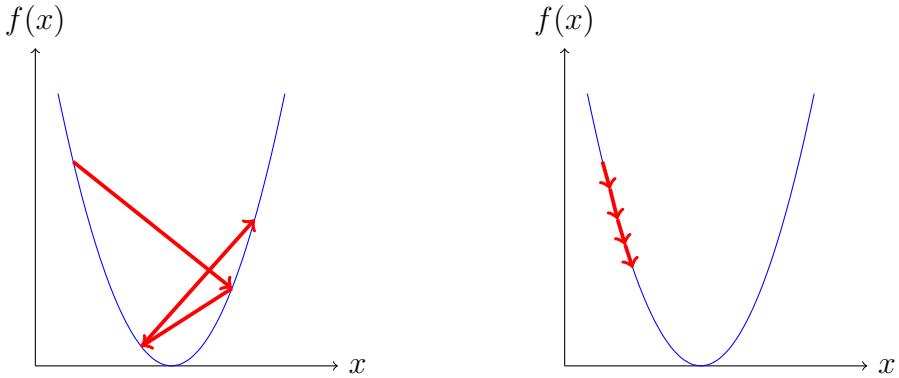


Figure 3.2.2: An example of the behavior of the momentum technique, with the black arrows showing the gradient, and the red lines the path taken when momentum is added.[8]

A final concern when backpropagating through the network is how big of a change to make to the weight, that is, what should the *learning rate* be? Too large learning rate and we might never converge, too small learning rate and it might take very long time converge (see Figure 3.2.3).



(a) Large learning rate **(b)** Small learning rate

Figure 3.2.3: Illustration of the problem of choosing the correct learning rate. Too high and it will never converge, too small and it will take too long time.

An approach to manage this issue is by having an optimizing algorithm with *adaptive learning rate*. One such algorithm is the *Adam Optimizer*[48]. Adam

[8]Image from [40].

(formally described in Algorithm 1) calculates adaptive learning rates for each parameter and updates them individually. Similar to the *momentum* technique it keeps an exponentially decaying average of the past gradients m, and in addition, it stores the exponentially decaying average of past squared gradients v. m and v are estimates of the first and second moments (the mean and uncentered variance) of the gradients and are used the update the weights θ, based on the initial learning rate ϵ using the following (simplified) update rule:

$$\theta_{t+1} = \theta_t - \epsilon \frac{m}{\sqrt{v}}. \tag{3.14}$$

There are other optimization algorithms with adaptive learning rates, such as *Adadelta*[76], *RMSprop*[18] and *Adagrad*[32]. Exactly which learning algorithm to use is still unclear, but it has been shown that algorithms with an adaptive learning rate perform fairly robust over a wide range of tasks[67].

Algorithm 1 Adam Optimizer Algorithm

Input: Step size ϵ
Input: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in [0,1).
Input: Small constant for δ for numerical stabilization
Input: Initial parameters θ
procedure TRAIN(ϵ, ρ, δ, θ)
    Initialize 1st and 2nd moment variables $s = 0$,$r = 0$
    Initialize time step $t = 0$
    while stopping criterion not met **do**
        Sample a minibatch of m examples from the training set $\{x^{(1)}, ..., x^{(m)}\}$ with corresponding targets $y^{(i)}$.
        Compute gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
        $t \leftarrow t + 1$
        Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1) g$
        Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$
        Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$
        Correct bias in second moment: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
        Compute update: $\nabla\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$
        Apply update: $\theta \leftarrow \theta + \nabla\theta$

3.2.3 Weight Initialization

Before the training can begin, the weights in the network need to be initialized to some value. Without any knowledge on what the final values should be, it would be reasonable to assume that half of the values would be positive, and half negative. Something sounding reasonable would then be to initialize all weights to zero, which could be expected to be "the best guess". Though this would be an error, since this would lead to all neuron to compute the same

output, something which would also lead to them computing the same gradient during the backpropagation, and thus undergoing the same changes. To solve this, some type of symmetry breaking will be needed. Something which can be achieved by choosing small random numbered, centered around zero and with unit standard deviation (e.g. from a Gaussian) as initial values for the weights.

A problem emerging with this approach is that the variance grows with the number of inputs, and ideally the variance should remain the same for each layer. This can be achieved by utilizing the Xavier initializer[37], where the variance is set to $1/N$, with N being the number of input neurons.

3.3 Convolutional Neural Networks

While there are a number of different classes of artificial neural networks, when it comes to image processing, convolutional neural networks (CNNs) are the most popular ones. CNNs are very similar to ordinary Neural Networks, they have weights and biases which are learnable. They have some non-linearity introducing element and in the end will predict a class score or some real value. What makes CNNs different is their assumption that the input data has a grid-like topology, such as with images.

As described earlier, and seen in Figure 3.1.5, in regular NNs all neurons in one hidden layer is connected to all neurons in the next hidden layer. It is clear to see that, with a normal sized image of $200x200$ pixels and three color channels, resulting in $120'000$ weights, the amount of parameters as more neurons are added will become overwhelming. Instead, what the CNNs do is taking advantage of correlations in the neighborhoods of features, which allows for a significant reduction in the number of parameters, without reducing the performance. Two main ideas leveraged by CNNs are *sparse interactions* and *parameter sharing*.

As mentioned earlier, having fully connected neurons when images become larger (thousands or even millions of pixels) is not desirable. CNNs utilize *sparse interactions* by making the kernel smaller than the input image. This can be done as, even though the entire image may contain millions of pixels, meaningful feature such as edges, can be found with kernels of only tens or hundreds of pixels. Figure 3.3.1 illustrates the difference between fully connected and sparse interactions. This approach allows for a reductions of parameter of many orders of magnitude. Looking at Figure 3.3.2 it can be seen how, in deep networks, neurons in deeper layers can be connected, indirectly, with a larger part of the input.

The *parameter sharing* idea refers to idea of reusing the parameters in more than just a single place in a model, which is the case for the normal neural networks, where each element is used in a single multiplication, and then never

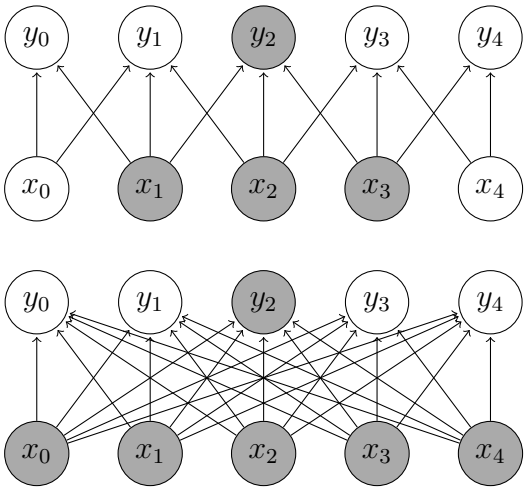


Figure 3.3.1: Difference between sparse connections and fully connected. As seen, a lot less connection are formed for the sparse connected one.

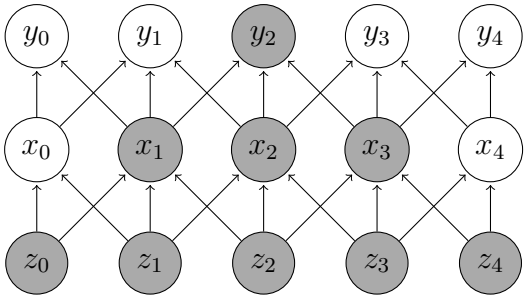
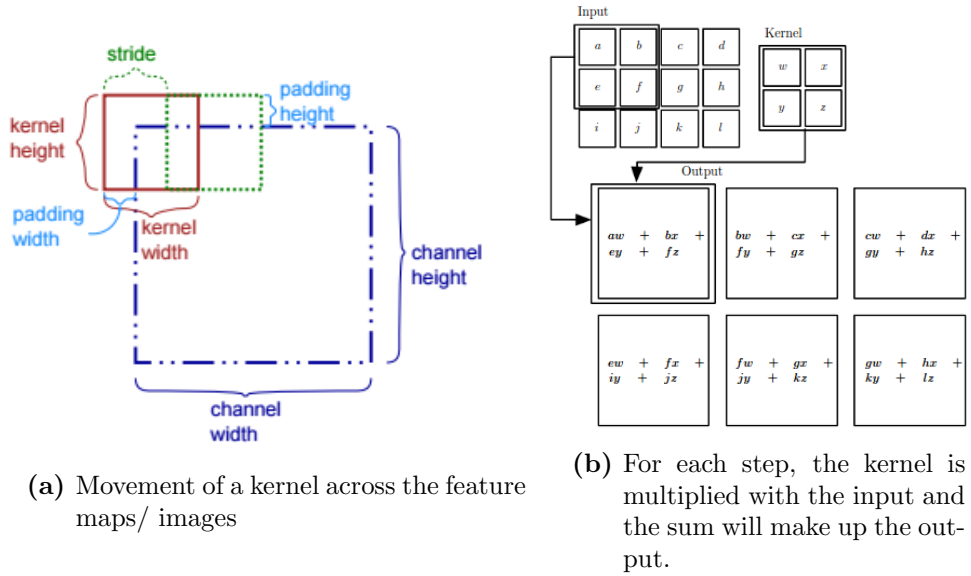


Figure 3.3.2: In deep architectures, even with sparse connections, a large part of the input can interact indirectly with deeper neurons.

revisited again. This is a quite logical idea, as if a kernel was used to identify a vertical edge at a certain location, that same kernel would be able to identify edges at other locations as well.

3.3.1 The Convolution Layer

In the convolution layer, a certain numbers of kernels of varying size (both dependent on the architecture used) move over the image performing a dot-product and reducing the sum which produce a single output per kernel (known as *feature maps*). The movement of the kernels and its operations are nicely illustrated in Figure 3.3.3. Each kernel creates a feature map which could be, depending on the the usage of *padding* or not, and the *stride* used, either the same size as the input or, more commonly, smaller. The stride determines how many steps the kernels should move between the calculations, a stride of two would roughly half the size of the image. Adding padding to an image entails adding a frame of zeros to the image. This is done in some architecture to make sure that each element of the kernel reaches all parts of the image.



(a) Movement of a kernel across the feature maps/ images

(b) For each step, the kernel is multiplied with the input and the sum will make up the output.

Figure 3.3.3: Behavior of a kernel in a convolutional operation.[11]

Each of the kernels reacts to different features, at lower levels these could be things like edges or colors. Whereas the later layers react on combinations of previous layer which could be things like eyes, feet, etc. Figure 3.3.4 shows the filters learned in the early layers by Krizhevsky et. al. in the famous AlexNet[9].

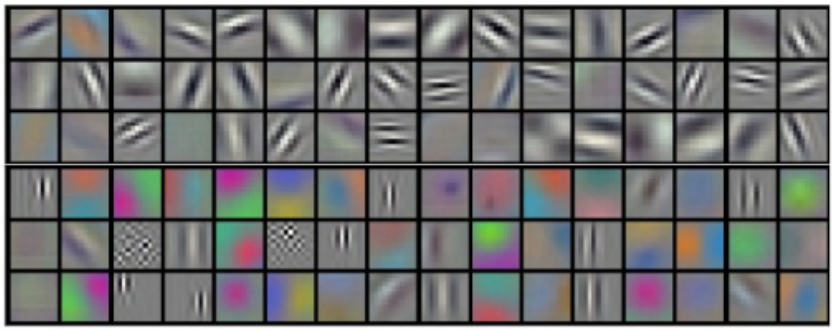


Figure 3.3.4: Illustration of the filters learned in AlexNet. Some are responsive of certain colors, others by certain structures.[12]

3.3.2 Pooling Layer

It is common to introduce a *pooling* layer occasionally between the convolutional layers. The pooling layer's main task is to reduce the dimensionality of the data, and is also useful for reducing noise and overfitting. One can describe the idea with the pooling layer by clarifying how, if one wants to identify a

[9]Winning architect of the 2012 ImageNet challenge.
[11]Image from [7].
[12]Image from [7].

face in an image, the pixel-wise precision on where the eye is located is not needed, merely if it is on the left or right side of the face would suffice.

The pooling layer works by a kernel of a specific size which moves over the image (similar to the convolutional layer) and performs some type of pooling operations; the most popular ones are *average pooling* and *max pooling*. A very nice visual example of the downsampling of a $224x224$ image using max pooling can be seen in Figure 3.3.5.

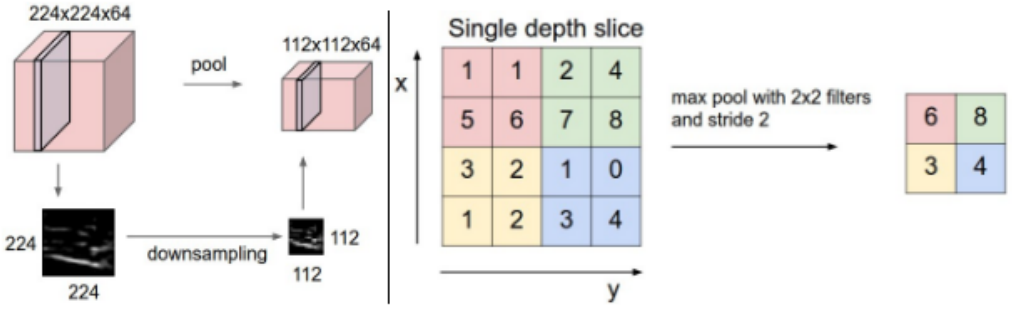


Figure 3.3.5: Example of a max pooling operation with a $2x2$ kernel with i stride of two, which downsamples the image from $224x224$ to $112x112$.[10]

Whereas having some pooling layers in the architecture is still the most popular approach, there are arguments that the pooling layers should be removed, in favor of extra convolutional layers[70]. As seen earlier, this can be achieved by increasing the stride in the convolutional layers, which in effect reduces the dimensionality.

3.3.3 Example Architecture

The main building blocks of convolutional neural networks have now been introduced; an example architecture using these building blocks can be seen in Figure 3.3.6 which depicts the architecture of *LeNet-5* [52].

LeNet-5 is a CNN architecture designed for digit recognition. The model takes a $32x32$ grey-scale image as input and passes it through a series of seven layers, to finally receive an prediction [0-9]. It consists of two convolutional layers; the first generating six feature maps of $28x28$ pixels, by using a $5x5$ kernel. The second convolutional layer also uses $5x5$ kernels, but are producing 16 feature maps of $10x10$ pixels. After each of the convolutional layer there is a sub-sampling (pooling) layer, which reduces the dimensionality to one fourth. After the second sub-sampling, three fully connected layer are added, which perform the classification based on the features extracted by the previous layers. The fully connected layers behave like the original neural networks, where all neurons are connected to all others.

[10]Image from [7].

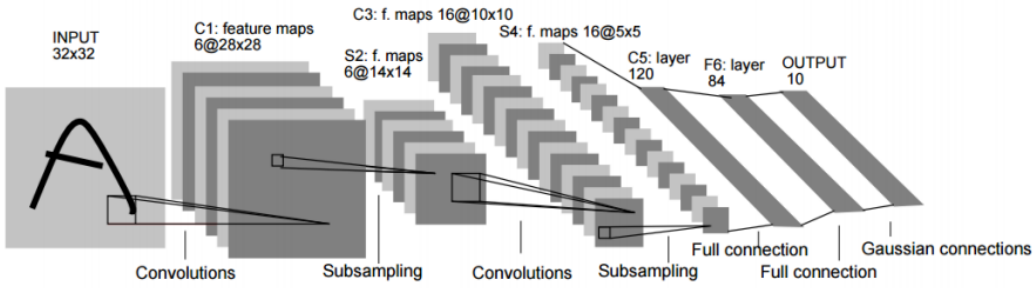


Figure 3.3.6: Illustration of the LeNet-5 architecture.[11]

3.3.4 Transfer Learning

Deep neural networks have one big flaw; they are fairly hard to train. Typically huge amount of data and a decent amount of compute power are needed. In fact, it has become less and less common for people to train their own model from scratch, i.e. with randomly initialized weights. Instead it is more common to *pre-train* the model on a very large data set (such as ImageNet with 1.3 million images) and then utilize the trained model, either as a fixed feature extractor or as initial weights for later fine-tuning. Andrew Ng[12] claims that transfer learning will be the next driver of machine learning's commercial success[24]. this is possible due to the interesting phenomenon that the features extracted in the earlier layers tend to be useful for other applications as well. A feature map trained to identify a straight line trained on cat recognition will still be useful when trying to identify birds[75].

Whereas utilizing a large data-set to pre-train the network is helpful when only a small data set for the task at hand is available, general difficulties of training deep neural networks remain. A much quicker way is to use an already pre-trained model made publicly available[13] and adapt the final classification or regression to fit the problem in hand, and to keep the earlier stages of the model fixed.

[11] Image from [52].

[12] Chief scientist at Baidu and professor at Stanford.

[13] For example, Model Zoo[3] has multiple pre-trained models available for other people to adapt to their task at hand.

4 Methods

In this chapter the methodologies and techniques, as well as software and hardware used in this thesis are described.

4.1 Software

For obtaining the training data the BSC performance tools with Paraver and Extrae (as introduced in Section 2.2) will be used.

For generating, and preprocessing (Section 4.4.1) the training data from the Paraver trace files, no special software was used. A simple combination of bash scripts and python code with common libraries (e.g. NumPy[1] and Matplotlib[2]) were used.

When it comes to software for the deep learning model there are quite a few to choose from, such as: *Caffe*[46], *Theano*[73], *Torch*[30] and *scikit-learn*[60]. But for this thesis *Tensorflow*[28], the open source Google developed scalable machine learning library is used.

Tensorflow was released in November 2015 and has quickly gained popularity. It is now one of the most popular deep learning libraries and has over 18′000 commits on GitHub[3]. In Tensorflow, machine learning algorithms are represented as computational graphs. Computational graphs (also known as, dataflow graphs) is a form of directed graphs where the nodes describe some type of computations, and the edges represent the data flowing between these operations[38]. The usage of data flow graphs allows for leveraging the parallel computational capabilities of multi-core CPU, GPU and even multiple GPUs[36]. The data in Tensorflow is represented by *tensors* (by which the name came from) which are typed n-dimensional arrays, and on which the computations are performed. Once a computational graph has been defined, it must be launched in a *Session*, which places the graph onto a CPU or GPU and provides the methods to run the computations.

[1] NumPy: Fundamental package for scientific computing in Python, http://www.numpy.org/

[2] Matplotlib: Python plotting library, producing publication quality figures, https://matplotlib.org/

[3] https://github.com/

4.2 Hardware

The computations performed in this thesis were executed on two different machines, one for training the models, and another for generating data and testing the models.

The training was done on a *bullx R421-E4* server which consists of[15]:

- 2 Intel Xeon E5-2630 v3 (Haswell) 8-core processors (each core at 2.4GHz with 20MB L3 cache)
- 2 K80 NVIDIA GPU Cards
- 128 GB Main memory, 8 DIMMs of 16 GB -DDR4 @ 2113 MHz - ECC - SDRAM
- Peak Performance: 250.94 TFlops
- 120 GB SSD local storage
- 1 PCIe 3.0 x8 8 GT/s, Mellanox ConnectX-3FDR 56 Gbit
- 4 Gigabit Ethernet ports

The usage of a GPU allowed for training more models and with larger images within a time frame otherwise not feasible with just a CPU.

The image generation and testing of the model were performed on a laptop with a Intel Core i7-4600U CPU, running at 2.1GHz, and having 8GiB DDR3 1600MHz memory.

Whereas training the model on a laptop would not have been feasible, no GPU is needed for evaluating or utilizing the trained models.

4.3 Neural Network Models

In the endeavor of achieving a model which was able to predict the metrics sought after, a number of different approaches were taken, all of which were utilizing convolutional neural networks. The two main approaches for training the network were:

1. Designing own models.
2. Using existing popular models and re-training them.

While the task at hand naturally would be considered a *regression* problem, typically *classification* problems are somewhat easier to train[7]. In this thesis both approaches will be explored. To achieve this, the final fully connected layer together with the *loss function* will be adapted for each of the two approaches.

4.3.1 VGG-19

As pre-existing model, the *VGG-19*[68] architecture was used, although with some alterations. VGG-19 was introduced for the 2014 ILSVRC, where it won first and second place in localization and classification tasks respectively. The authors have openly released the weight used in the Imagenet challenge, which were trained on the Imagenet data set[31], containing 1.3 million images of 1000 different categories. The weights were released to the Caffe model zoo repository[3] and have later been converted to be compatible with Tensorflow models[22].

The original VGG-19 model is built up by 16 convolutional layers, as seen in Table 1, and three fully connected layers. The convolutional layers are utilizing fairly small kernels sizes of $3x3$, and have set the *stride* and *padding* set to 1, which leads to the output feature maps have the same spatial dimensions as the input feature maps. For each of the convolutional- and fully connected layers ReLUs (explained in Section 3.1.1.2) are utilized to achieve non-linearity. For down-sampling of the feature maps, the convolutional layers were intertwined with *max pooling* layers (introduced in Section 3.3.2). The max pooling layers utilized non-overlapping $2x2$ kernels (i.e. stride two), which reduced the feature vector sizes by a factor of two. The full spatial configuration of the convolutional and pooling layers of VGG-19 can be seen in Table 1 and more thoroughly explained in the paper [68].

Layer	**Weight dimension**	**Kernel Size**	**Stride**	**Size reduction factor**	**Pooling/ Activation**
Data				(1,1)	
Conv1_1	(64,3,3,3)	3	1		ReLU
Conv1_2	(64,64,3,3)	3	1		ReLU
Pool1		2	2	(2,2)	Max Pooling
Conv2_1	(128,64,3,3)	3	1		ReLU
Conv2_2	(128,128,3,3)	3	1		ReLU
Pool2		2	2	(4,4)	Max Pooling
Conv3_1	(256,128,3,3)	3	1		ReLU
Conv3_2	(256,256,3,3)	3	1		ReLU
Conv3_3	(256,256,3,3)	3	1		ReLU
Conv3_4	(256,256,3,3)	3	1		ReLU
Pool3		2	2	(8,8)	Max Pooling

Conv4_1	(512,256,3,3)	3	1		ReLU
Conv4_2	(512,512,3,3)	3	1		ReLU
Conv4_3	(512,512,3,3)	3	1		ReLU
Conv4_4	(512,512,3,3)	3	1		ReLU
Pool4		2	2	(16,16)	Max Pooling
Conv5_1	(512,512,3,3)	3	1		ReLU
Conv5_2	(512,512,3,3)	3	1		ReLU
Conv5_3	(512,512,3,3)	3	1		ReLU
Conv5_4	(512,512,3,3)	3	1		ReLU
Pool5		2	2	(32,32)	Max Pooling

Table 1: Structure of the original VGG-19's convolutional and pooling layers.

4.3.1.1 Architecture modifications

With the original VGG-19 model as a baseline, a number of alternative models were produced. For the models created for this thesis, only the convolutional and pooling layers (i.e. the feature extraction) from VGG-19 were used. Three fully connected layers were then used to produce the desired output (regression/classification).

As utilizing *average* pooling instead of *max* pooling has been seen to show some improvements[35], one alteration tested was to replace the original max pooling layers with average pooling layers throughout the network. For similar reasons, tests were performed where the activation functions were changed from ReLUs to Leaky ReLUs.

The models based on the VGG-19 architecture will be referred to using the following naming convention:

VGG-19_Pooling_Activation_Output

Each of the parameters will take on a single letter with the following mapping:

Pooling		Activation		Output	
Max pool	Average pool	ReLU	Leaky ReLU	Classification	Regression
M	A	R	LR	C	R

4.3.2 Custom Models

Apart from the VGG-19 based models, attempts at creating and training new models from scratch were also made. The *custom* models created were designed following the design principles presented in [69]. More specifically, the

design patterns which the custom models will aim at fulfilling are: to *strive for simplicity*, to *increasing symmetry*, and following a *pyramid shape*.

It has been shown that simple architectures with very few different types of units can perform very well[70]. Based on these findings the custom models will only consist of four types of computations; *convolutions*, *ReLUs*, *average pooling*, and fully connected *multiply and adds*.

There is also evidence of having symmetry (i.e. repeating structures) in an architecture to facilitate training of deeper networks, such as in [50]. With this in mind, the custom models will be constructed of blocks of two convolutional operations and two ReLUs, followed by an average pooling.

$$Block = \{CONV - ReLU - CONV - ReLU - AVG.POOL\}$$

The final design principle on which the models were based on was for the architecture to have a pyramid shape. One of the fundamental design patterns for convolutional networks is the trade-off between maximum representational power and elimination of redundant information. With the architecture having a pyramid shape refers to that there should be an overall smooth downsampling, combined with an increasing number of channels in throughout the architecture, which is exemplified in [41]. Taking this in consideration, the number of channels added from one layer to another is kept small, and the pooling performed is only done with a stride of two, not higher.

The custom models created consist of a number n of the previously explained blocks, followed by three fully connected layers.

$$Architecture = Block \ \times \ n - FC6 - FC7 - FC8$$

Architectures up to four blocks ($n = 4$) were used for the custom models. The naming convention for the custom models is the following:

$$Custom\_n, \quad n \in [1, 2, 3, 4].$$

4.4 Data Generation

The models were trained against Paraver MPI timeline images of different traces available within the POP project at BSC. A total of $20'418$ images were generated, for different core counts, ranging from 6 cores up to 48 cores, with efficiency metrics evenly divided between $0 - 100\%$.

To generate the images, a small chop[4] of the trace file was made using Paramedir. The chops were then loaded with Paraver, which produced a timeline image,

[4]Chopping a trace file reduces the size of the trace. For this work, the chops were of a

similar to Figure 4.4.1, and calculated the performance metrics for that specific chop. The workflow of the data generation can be seen in Algorithm 2.

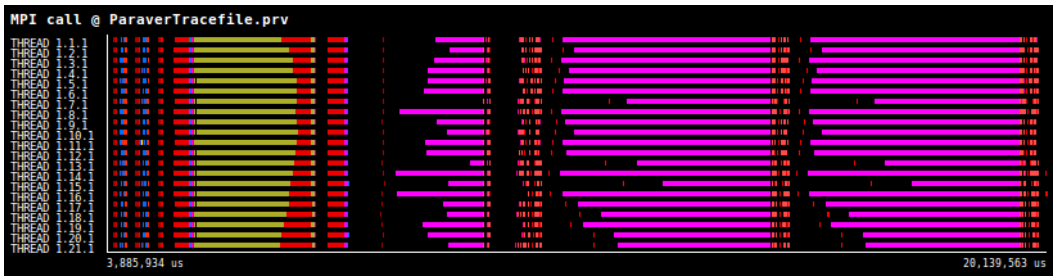


Figure 4.4.1: Example of an unprocessed training image. Different ranks are projected on the y-axis, and time along the x-axis. The different colors represent time spent in different MPI calls, with black being time outside of MPI.

Algorithm 2 Data Generation Workflow

1: **procedure** GENERATE DATA(Tracefile F, Time Interval $T_{i,j}$)
2: Chop ← Paramedir(F,$T_{i,j}$)
3: Image,Metrics ← Paraver(Chop)
4: **Return** Image, Metrics

The images generated in this process were of $200x525$ pixels. Having the images being larger in width than in height allows for more detailed information about the trace in the same image.

4.4.1 Data Pre-Processing

During the work, whether pre-processing the images would produce better results or not was investigated. Pre-processing the images produces two main benefits for the training. Firstly, non-relevant information in the image (such as the name of the trace) are filtered out, thus reducing the intellectual capabilities needed for the model. Secondly, the dimensionality of the images is reduced, which in turn means less computations and less memory needed for the model, leading to faster convergence.

The pre-processing consists of three phases. First the black borders were cropped out of the image (See Figure 4.4.1 for example of an image before pre-processing.). Secondly, the black padding between the ranks was removed, and the timeline was re-painted with time spent within the parallel runtime in blue, and time performing useful work in green. Finally, the image was re-scaled to fit the dimensions $144x490$ pixels (re-scaling the images allows for different core counts to be trained on the same model), which is about a third

(application dependent) fraction of the total execution time, resulting in a trace file with the same number of ranks, but just a small time frame.

of the pixels of the original image. The workflow of the pre-processing can be seen in Algorithm 3 and an example output image is depicted in Figure 4.4.2.

Algorithm 3 Image Pre-Processing

```
1: procedure PROCESS IMAGE(Image I)
2:     Cropped ← CropImage(I)
3:     ReColored ← ReColor(Cropped)
4:     ReScaled ← ReSacale(ReColored,144,490)
5:     Return ReScaled
```

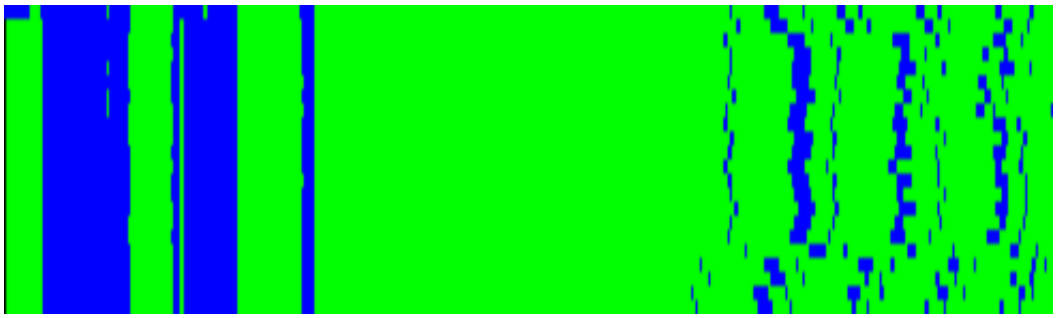


Figure 4.4.2: An example of a pre-processed image, where the blue color shows time spent in a parallel runtime and green color displays performing useful work.

4.5 Training Process

For the experiments, the total data set was divided into two sets; a training set and a test set. The data was divided in a 85/15 split, where 85% of the data formed the training set and 15% was used for the test set. The purpose of this division is to allow for the models to be evaluated on data never seen during the training process, thus being able to evaluate the model passed the point of overfitting. During the training phase, any alterations on the weights and biases will only be done based on the training set. The performance of the models will then be evaluated based on the test set.

During the experiments both the *loss*-function (Section 3.2.1) as well as the training, and testing accuracy will be monitored. Where the final conclusions will be drawn from the test accuracy. With the predictions being real valued number (although represented as a classification problem for some of the models) representing the percentage of the different metrics, the accuracy will be evaluated by how close the prediction is to the actual value. The accuracy is evaluated at seven thresholds, checking how much of the validation data are correctly predicted with an error less than that threshold. The thresholds are: $1\%, 2\%, 3\%, 4\%, 5\%, 10\%$ and 15%. The values the models are trained to

predict are the three performance metrics introduced in Section 2.1, which are *parallel efficiency*, *load balance*, and *communication efficiency*. The models are trained for each of the metrics separately.

5 Experiments

This chapter presents the experiments for the different models, together with their results in a running fashion.

As mentioned in Section 4.5, the experiments done are training the models introduced in Section 4.3 to, based on an input image, predict the three performance metrics shown in Section 2.1 (Parallel Efficiency, Load Balance, and Communication Efficiency).

The 85/15 split of the data of 20′417 images left a training set with 17′355 images and a test set with 3′062 images. Where the models will be evaluated based on how they perform against the images in the test set.

Common for all the models was how the weights were initialized, as well as the learning algorithm used. All weights were initialized using the Xavier initializer (see Section 3.2.3) and the biases are all set to zero. All the models will also utilize the Adam optimizer for their weight updating (see Section 3.2).

The classification models all used softmax and the cross-entropy loss for the training. And all the regression models used the MSE loss.

The first experiments considered data which had no pre-processing done to them and will be covered in Section 5.1. Then the experiments for the VGG-19 based models and the custom models are given in Section 5.2 and Section 5.3, respectively. The importance of large amounts of training data will be evaluated in Section 5.4.

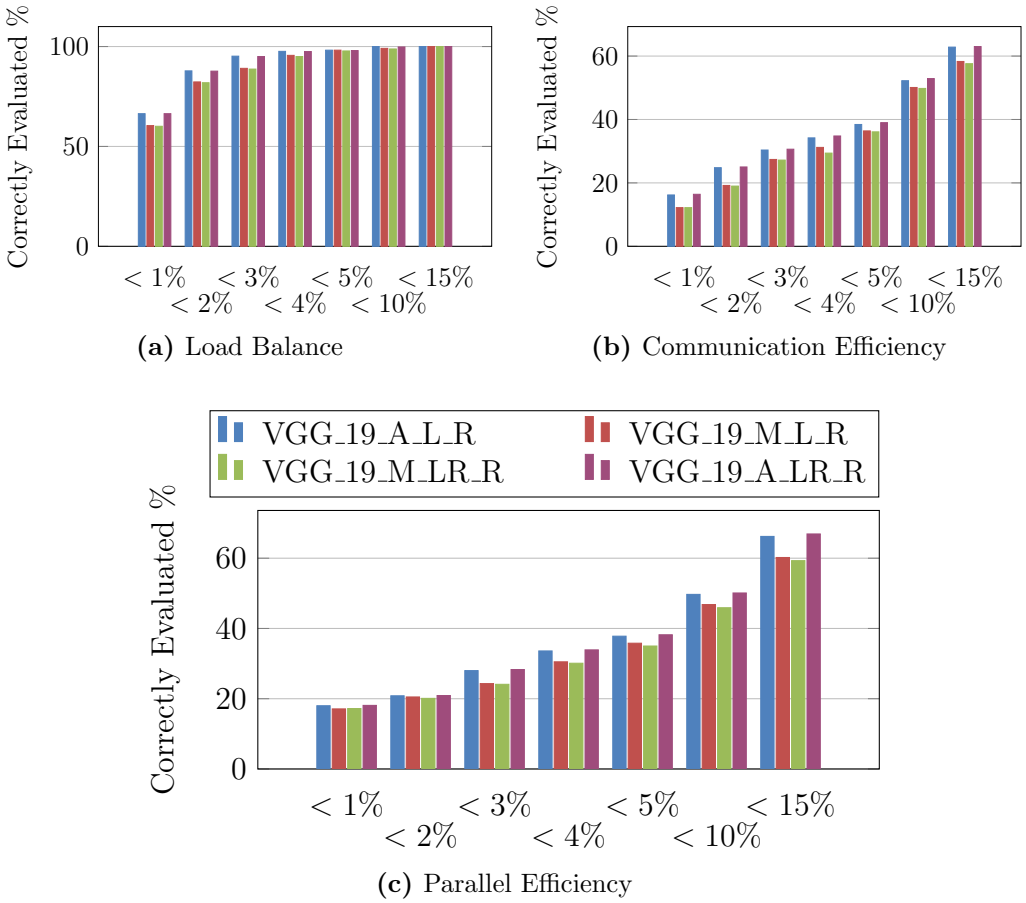
The results of the performance of the models are presented in bar plots, and a numerical representation of the results can be found in Appendix A. The bar plots will project the percentage of the test data which was correctly predicted within the specific threshold values.

5.1 Unprocessed Data

For the evaluation of the unprocessed data (images such as in Figure 4.4.1), the VGG-19 based regression models are used. The learning rate is set to 10^{-5} and the training was stopped after 10′000 epochs (each epoch consisting of a batch of 32 images), as no further improvements were seen. The training had then been going on for just over seven hours.

As seen in Figure 5.1.1 the models did not achieve a very high accuracy for the

three performance metrics, with only 18% of the data being predicted within 1% of the actual value for the parallel efficiency, and even worse (16.2%) for the communication efficiency. Even though somewhat acceptable values for the load balance metric were achieved (with 95.2% of the data being predicted within 3% of the actual value), the experimentations for the rest of the models will use pre-processed data.



(a) Load Balance

(b) Communication Efficiency

(c) Parallel Efficiency

Figure 5.1.1: Results for the models trained with unprocessed data.

5.2 VGG-19 Models

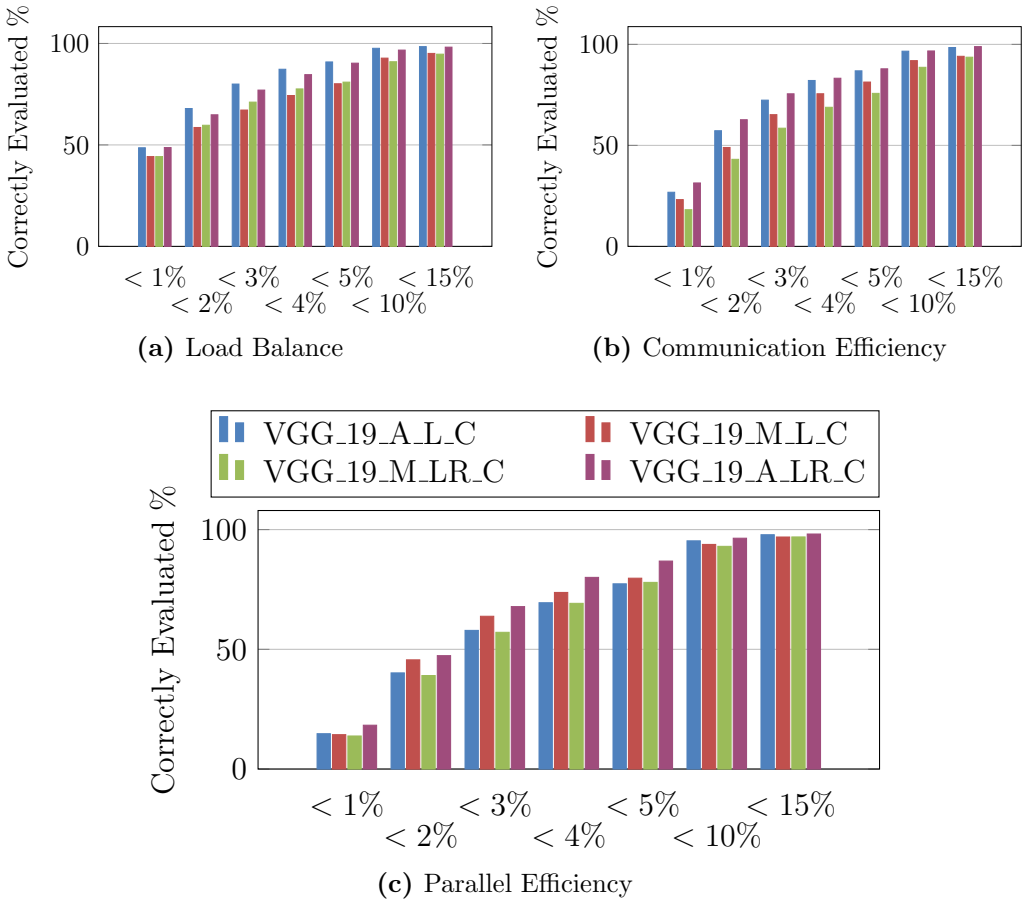
The experiments for the VGG-19 based models are presented in this section. All the training was performed on preprocessed data, as explained in Section 4.4.1.

5.2.1 Classification

The training for the classification models was performed in batches of 32 images at a time and was run during a total of 18′000 epoch, after which no further

improvements were seen (controlled by monitoring the loss function), which took around eleven hours. The learning rate for the training was set to 10^{-5}.

In Figure 5.2.1 the results for the classification models can be seen. For the best performing of these models to reach a half-decent accuracy of close to 90%, an error of 5% in the predictions will still be present. To get close to (still without reaching) a 100% accuracy, up to a 10% and 15% error will be present in the predictions, while only reaching a 97% and 98% accuracy, respectively. Such performance would be considered too imprecise to be useful.



(a) Load Balance

(b) Communication Efficiency

(c) Parallel Efficiency

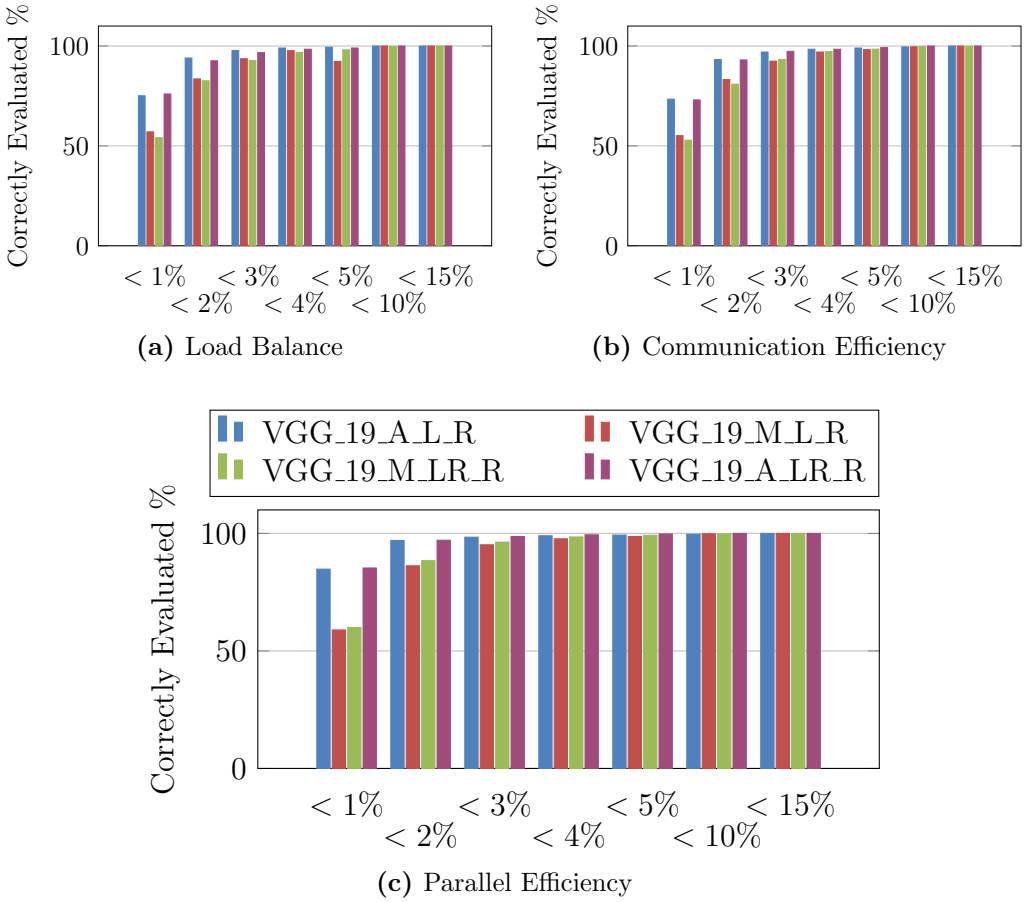
Figure 5.2.1: Results for the VGG-19 based classification models.

5.2.2 Regression

The training for the regression models was performed in batches of 32 images at a time, and was executed during a total of 16′000 epoch, after which no further improvements were seen (controlled by monitoring the loss function), which took around twelve hours. The learning rate for the training was set to 10^{-5}.

Figure 5.2.2 shows the accuracies achieved by the VGG-19 based models with regression heads for the three performance metrics. As seen in the bar plots,

the two models with *average pooling* are the ones with highest performance, with close to 100% accuracy with less than a 3% error, for all three metrics, which would be considered more than acceptable. For the highest performing model, not a single image is mispredicted with an error of 10%.



(a) Load Balance

(b) Communication Efficiency

(c) Parallel Efficiency

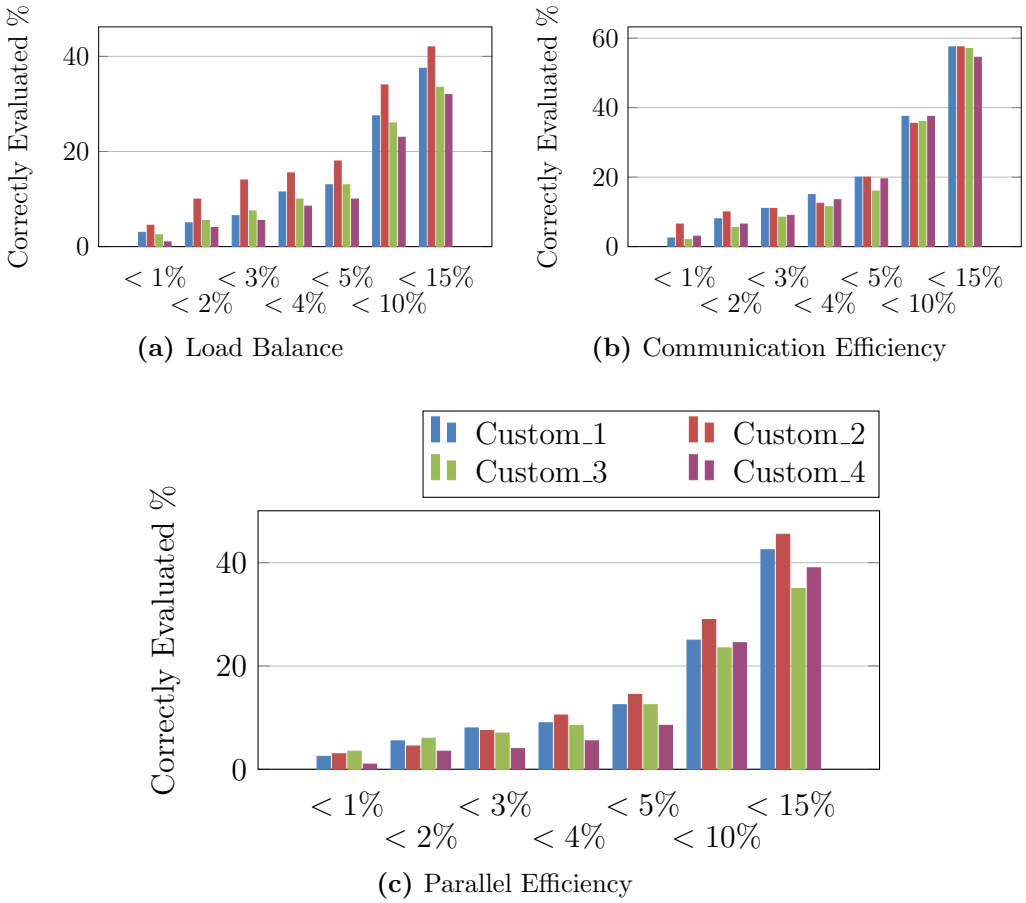
Figure 5.2.2: Results for the VGG-19 based regression models.

5.3 Custom Models

The custom models were all trained using batches of 32 preprocessed images, with a learning rate set to 10^{-4}. The training for the custom models was performed during 10′000 epochs, after which no further improvements were seen, something which took between two and six hours, depending on the number of blocks in the model.

In Figure 5.3.1, where the results for the custom models are presented, overall very low accuracies are seen. Even for the best performing model, *Custom_2*, less than 20% of the data is correctly predicted with less than a 5% error, for all metrics (except communication efficiency which just reaches 20%). Extending the error tolerance to 15% still only roughly half of the data is predicted

correctly, making the resulting trained models useless.



(a) Load Balance

(b) Communication Efficiency

(c) Parallel Efficiency

Figure 5.3.1: Results for the custom models.

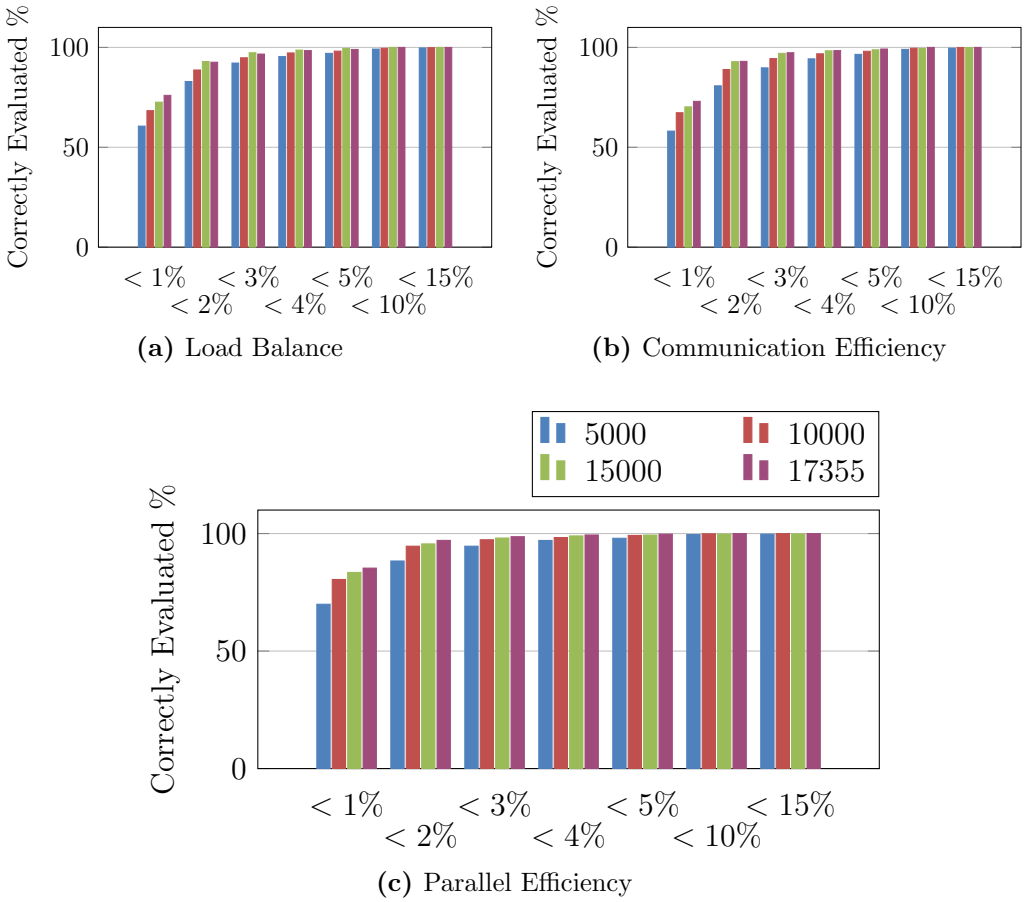
5.4 Importance of Large Data Sets

With a common requirement for many deep learning methods being a large data set. And seeing how the models pre-trained on a data set of $1.3M$ images clearly outperforms the models trained from scratch with the specific data set generated of $20K$ images, further experiments were performed to investigate how the sizes of the data sets are affecting the results.

The best performing model, *VGG_19_A_LR_R*, was trained again, this time while reducing the available images in the training set. The model was trained again three times, against training sets of $5'000, 10'000$, & $15'000$ images, and compared to the $17'355$ images in the full-sized training set.

The training was performed similarly to the full data set, with batches of 32 images, and a learning rate of 10^{-5}. The models ran for between $12'000$ and $16'000$ epoch after which no further improvements were seen.

The results of this experiments can be seen in Figure 5.4.1, which clearly show how the accuracy of the models improves as the training set is increased.



(a) Load Balance

(b) Communication Efficiency

(c) Parallel Efficiency

Figure 5.4.1: Results when the model was trained using limited data.

6 Discussion, Conclusions and Future Work

6.1 Discussion

The experiments identified two models (VGG_19_A_L_R, and VGG_19_A_LR_R) with the capability of predicting accurate performance metrics within a few percentages for close to 100% of the test data. Whereas the performance of many of the models did not reach satisfactory accuracies, the accuracies in the top two models can be considered useful in an actual performance evaluation.

The experiments clearly showed how assisting the models by reducing the complexity of the task at hand really improves their performance. In this work this was done by preprocessing the images, and removing any non-relevant information. Not only did this improve the performance of the models, it also reduced their memory requirements as well as the amount of calculations needed, which in turn reduces the total computation time.

With both pre-trained models as well as custom models being experimented with, it is clear that the pre-trained models outperform the custom ones. A plausible explanation to this would be the vast difference in training data available during the training. With batch sizes of 32 images used, and a training set of only 17′355 images, it would take the training less than 545 epochs to iterate through the entire data set. Meaning that the same image would be going through the training step on 18 occasions or more (depending on the number of epochs trained for), which is less than ideal for generalization purposes.

The last experiments with limited amount of data clearly showed how a single model's performance is significantly increased as more training data is provided. This result together with the pre-trained models outperforming the custom models are clear indicators that large amounts of data are paramount for achieving good results.

The duration of the training for the different models took between two and twelve hours on a GPU. With the amount of different models and experiments run, it is clear that this would not have been feasible on a CPU alone, within a reasonable time frame.

6.2 Conclusions

In this work, deep learning techniques have been introduced to the field of HPC performance evaluation. A number of different convolutional neural network models have been trained and tested on timeline views from Paraver, with the purpose of predicting different performance metrics for an application, which would allow performance evaluation without the need of an expert.

The models extract features from the timeline views and are, based on that, able to assess the performance of the application. Experiments showed how the performance of the models improved significantly when the input images are preprocessed, and in such reducing the complexity of the prediction task. It was also seen that the performance of the models increase with the size of the data set, and continuing the training of the best performing model, with an even larger data set would most likely improve its performance even further.

One of the fundamental limitations of the approach with utilizing timeline images from Paraver is the pixel limitation on the images. With the granularity of events in a timeline being limited to the size of a pixel, would an application have a granularity of its events finer than what can be expressed with a single pixel, this information will be lost in the images generated.

Based on the results in this work, it is clear to say that deep learning techniques indeed are potential instruments for the HPC performance evaluation field. Hopefully this work will inspire others to continue researching the possibilities utilizing deep learning in automating the performance analysis process. Which would allow for users without the expert knowledge to in a simpler manner identify issues and to improve the performance of their applications.

6.3 Future Work

Having introduced a working model for predicting the three top efficiency metrics (Parallel Efficiency, Load Balance, Communication Efficiency), a natural extension to this work would be to train the models for metrics lower in the hierarchy (Serialization Efficiency and Transfer Efficiency). The limiting factor for such models would be the generation of training data. As experiments for this work showed, for the models to generalize well, a large amount of data is needed. And for generation of data for the above mentioned metrics, a Dimemas simulation of the application on an ideal network is needed, an extra step which would increase the labor of generating large amounts of data. Increasing the current data set and continuing the training for the presented models would also be an interesting continuation, which would likely improve the accuracies achieved even further.

With Paraver offering views of different information for the obtained trace, and having been designed for visual analysis, many interesting opportunities

for alleviating the analysis process with deep learning models exist.

For example identifying the impact of MPI arrival times (entering a communication phase with some skew may lead to the communications taking longer), as shown in Figure 6.3.1. Or identifying when applications suffer from end-point contention, as in Figure 6.3.2. The common hurdle for all these continuations is the generation of a large enough data set.

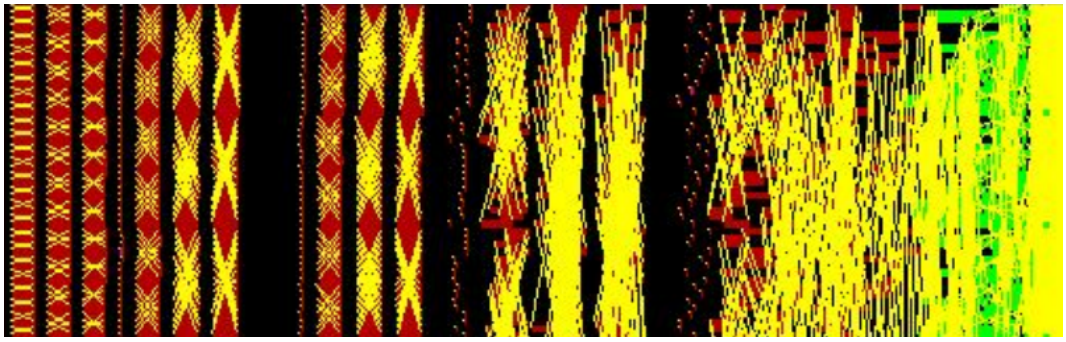


Figure 6.3.1: Application with some imbalances, causing ranks to reach communication phases skewed, which can increase the communication time.

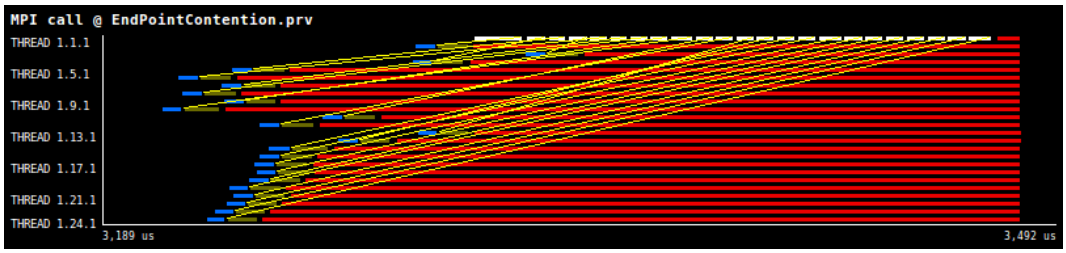


Figure 6.3.2: Application suffering from end-point contention, where all ranks are communicating with rank 0 at the same time.

Bibliography

[1] Allinea MAP. https://www.allinea.com/products/allinea-performance-reports. Accessed: 2017-04-24.

[2] Allinea Performance Reports. https://www.allinea.com/products/map. Accessed: 2017-04-24.

[3] Berkley Caffe Model Zoo, Jia Yangqing. http://caffe.berkeleyvision.org/model_zoo.html. Accessed: 2017-04-24.

[4] BSC Performance Tools. https://tools.bsc.es/. Accessed: 2017-04-24.

[5] BSC Performance Tools - paraver. https://tools.bsc.es/paraver. Accessed: 2017-04-24.

[6] Cray Performance Measurement and Analysis Tools. http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=S-2376-610. Accessed: 2017-04-24.

[7] cs231n - Convolutional Neural Networks for Visual Recognition andrej karpathy, stanford. http://cs231n.github.io/. Accessed: 2017-04-18.

[8] Film Studios Refocus HPC Management Lens nicole hemsoth. https://www.hpcwire.com/2014/08/25/film-studios-refocus-hpc-management-lens/. Accessed: 2017-04-27.

[9] General Motors is literally tearing its competition to bits sean gallagher. https://arstechnica.com/information-technology/2013/09/general-motors-is-literally-tearing-its-competition-to-bits/. Accessed: 2017-04-27.

[10] Higgs Boson Machine Learning Challange kaggle. https://www.kaggle.com/c/higgs-boson. Accessed: 2017-04-24.

[11] High Performance (Potato) Chips michael feldman. https://www.hpcwire.com/2006/05/05/high_performance_potato_chips/?utm_content=50621565utm_medium=socialutm_source=twitter. Accessed: 2017-04-27.

[12] HPC for Energy. https://hpc4e.eu/. Accessed: 2017-04-29.

[13] HPC Matters Plenary. http://sc14.supercomputing.org/program/hpc-matters-plenary.html. Accessed: 2017-04-27.

[14] HPC Matters to our Quality of Life and Prosperity don johnston. http://www.scientificcomputing.com/article/2014/11/hpc-matters-our-quality-life-and-prosperity. Accessed: 2017-05-03.

[15] Mino Tauro User's Guide. https://www.bsc.es/support/MinoTauro-ug.pdf. Accessed: 2017-04-24.

[16] Performance Optimisation and productivity, a centre of excellence in computing applications. https://pop-coe.eu/. Accessed: 2017-04-27.

[17] POP Standard Metrics for Parallel Performance Analysis. https://pop-coe.eu/node/69. Accessed: 2017-04-24.

[18] RMSprop Gradient Optimizer. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Accessed: 2017-05-14.

[19] Scalable Performance Measurement Infrastructure for Parallel Codes - SCORE-P. http://www.vi-hps.org/projects/score-p. Accessed: 2017-05-21.

[20] Scalasca. http://www.scalasca.org/. Accessed: 2017-05-21.

[21] Supercomputers a hidden power center of Silicon Valley pete carey. http://www.mercurynews.com/2015/05/07/supercomputers-a-hidden-power-center-of-silicon-valley/. Accessed: 2017-05-07.

[22] Tensorflow VGG. https://github.com/machrisaa/tensorflow-vgg. Accessed: 2017-03-10.

[23] The insideHPC Guide to HPC in Life Sciences (sgi and intel). http://insidehpc.com/2015/03/the-insidehpc-guide-to-hpc-in-life-sciences/. Accessed: 2017-04-24.

[24] The nuts and bolts of machine learning, Sebastian Ruder. http://sebastianruder.com/highlights-nips-2016/index.html#thenutsandboltsofmachinelearningl. Accessed: 2017-04-16.

[25] The Supercomputing Strategy That Makes Airbus Soar nicole hemsoth. https://www.nextplatform.com/2015/07/22/the-supercomputing-strategy-that-makes-airbus-soar/. Accessed: 2017-04-26.

[26] Trace Analyzer and Collector, Intel. https://software.intel.com/en-us/intel-trace-analyzer. Accessed: 2017-04-24.

[27] VTune Amplifier, Intel. https://software.intel.com/en-us/intel-vtune-amplifier-xe. Accessed: 2017-04-24.

[28] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[29] Steve Ashby, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, et al. The opportunities and challenges of exascale computing. *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*, pages 1–77, 2010.

[30] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[32] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[33] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[34] Stephen J Ezell and Robert D Atkinson. The vital importance of high-performance computing to us competitiveness. *Information Technology and Innovation Foundation, April*, 28, 2016.

[35] Leon Gatys, Alexander S Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 262–270, 2015.

[36] Francesco Giannini, Vincenzo Laveglia, Alessandro Rossi, Dario Zanca, and Andrea Zugarini. Neural networks for beginners. a fast implemention in matlab, torch, tensorflow. *arXiv preprint arXiv:1703.05298*, 2017.

[37] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[38] Peter Goldsborough. A tour of tensorflow. *arXiv preprint arXiv:1610.01178*, 2016.

[39] Juan González García. Application of clustering analysis and sequence analysis on the performance analysis of parallel applications. 2013.

[40] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[41] Dongyoon Han, Jiwhan Kim, and Junmo Kim. Deep pyramidal residual networks. *arXiv preprint arXiv:1610.02915*, 2016.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[43] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[44] IDC. High performance computing in the eu:progress on the implementation of the european hpc strategy. 2015.

[45] Thomas Ponweiser Thomas Steinreiter Jerry Eriksson, Pedro Ojeda-May. Profiling and tracing tools for performance analysis of large scale applications. *PRACE: Partnership for Advanced Computing in Europe*, 2016.

[46] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of*

the 22nd ACM international conference on Multimedia, pages 675–678. ACM, 2014.

[47] Earl C Joseph, Chirag Dekate, and Steve Conway. Real-world examples of supercomputers used for economic and societal benefits: A prelude to what the exascale era can provide. *International Data Corporation, May*, 20, 2014.

[48] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[49] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[50] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Fractalnet: Ultra-deep neural networks without residuals. *arXiv preprint arXiv:1605.07648*, 2016.

[51] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[52] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[53] Yunjie Liu, Evan Racah, Joaquin Correa, Amir Khosrowshahi, David Lavers, Kenneth Kunkel, Michael Wehner, William Collins, et al. Application of deep convolutional neural networks for detecting extreme weather in climate datasets. *arXiv preprint arXiv:1605.01156*, 2016.

[54] Germán Llort, Harald Servat, Juan González, Judit Giménez, and Jesús Labarta. On the usefulness of object tracking techniques in performance analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 29. ACM, 2013.

[55] Junshui Ma, Robert P Sheridan, Andy Liaw, George E Dahl, and Vladimir Svetnik. Deep neural nets as a method for quantitative structure–activity relationships. *Journal of chemical information and modeling*, 55(2):263–274, 2015.

[56] Lin Ma, Zhengdong Lu, and Hang Li. Learning to answer questions from image using convolutional neural network. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[57] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[58] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a). *CoRR*, abs/1701.00854, 2017.

[59] Marvin Minsky and Seymour Papert. Perceptrons. 1969.

[60] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[61] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[62] Thomas E Potok, Catherine D Schuman, Steven R Young, Robert M Patton, Federico Spedalieri, Jeremy Liu, Ke-Thia Yao, Garrett Rose, and Gangotree Chakma. A study of complex deep learning networks on high performance, neuromorphic, and quantum computers. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, pages 47–55. IEEE Press, 2016.

[63] Claudia Rosas, Judit Giménez, and Jesús Labarta. Scalability prediction for fundamental performance factors. *Supercomputing frontiers and innovations*, 1(2):4–19, 2014.

[64] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[65] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[66] Tara N Sainath, Abdel-rahman Mohamed, Brian Kingsbury, and Bhuvana Ramabhadran. Deep convolutional neural networks for lvcsr. In *Acoustics, speech and signal processing (ICASSP), 2013 IEEE international conference on*, pages 8614–8618. IEEE, 2013.

[67] Tom Schaul, Ioannis Antonoglou, and David Silver. Unit tests for stochastic optimization. *CoRR*, abs/1312.6055, 2013.

[68] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[69] Leslie N Smith and Nicholay Topin. Deep convolutional neural network design patterns. *arXiv preprint arXiv:1611.00847*, 2016.

[70] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[71] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[72] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[73] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[74] Haohan Wang, Bhiksha Raj, and Eric P Xing. On the origin of deep learning. *arXiv preprint arXiv:1702.07800*, 2017.

[75] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[76] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

Appendices

A Detailed Results

This appendix presents the results of the experiments numerically in a table format. Each column in the tables shows the percentage of test data which was correctly predicted with an error less than the column header.

A.1 Unprocessed Data

Table 2 Results for VGG_19_A_L_R trained on unprocessed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	18	20.8	28	33.6	37.8	49.8	66.2
Load Balance	66.4	87.8	95.2	97.6	98.2	100	100
Communication Efficiency	16.2	24.8	30.4	34.2	38.4	52.2	62.8

Table 3 Results for VGG_19_M_L_R trained on unprocessed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	17.1	20.5	24.3	30.5	35.8	46.8	60.2
Load Balance	60.4	82.3	89.1	95.6	98.2	99.1	100
Communication Efficiency	12.2	19.2	27.4	31.2	36.4	50.1	58.3

Table 4 Results for VGG_19_M_LR_R trained on unprocessed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	17.2	20.1	24	30.1	35	45.9	59.3
Load Balance	60.1	81.9	88.7	95	97.8	98.8	100
Communication Efficiency	12.2	19	27.2	29.4	36.1	49.8	57.6

Table 5 Results for VGG_19_A_LR_R trained on unprocessed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	18.1	20.9	28.3	33.9	38.2	50.1	66.9
Load Balance	66.4	87.7	95.0	97.5	98	99.8	100
Communication Efficiency	16.4	25	30.6	34.8	39	52.9	63

A.2 VGG-19 Classification Models

Table 6 Results for VGG_19_A_L_C trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	14.75	40.15	57.9	69.45	77.35	95.2	97.85
Load Balance	48.6	67.95	80	87.3	90.9	97.65	98.45
Communication Efficiency	26.8	57.25	72.35	82.15	86.95	96.6	98.45

Table 7 Results for VGG_19_M_L_C trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	14.35	45.65	63.8	73.75	79.7	93.8	96.9
Load Balance	44.2	58.55	67.15	74.35	80.15	92.75	95.1
Communication Efficiency	23.15	48.95	65.4	75.55	81.35	91.95	94.1

Table 8 Results for VGG_19_M_LR_C trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	13.8	39.05	57.1	69.2	77.95	93	96.95
Load Balance	44.25	59.7	71.05	77.6	81.35	91.8	94.75
Communication Efficiency	18.15	43.1	58.55	68.85	75.75	88.65	93.65

Table 9 Results for VGG_19_A_LR_C trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	18.3	47.35	67.8	80.05	86.85	96.4	98.15
Load Balance	48.7	64.9	77.05	84.65	90.3	96.75	98.2
Communication Efficiency	31.4	62.7	75.5	83.2	87.9	96.8	98.85

A.3 VGG-19 Regression Models

Table 10 Results for VGG_19_A_L_R trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	84.8	97	98.4	99	99.3	99.63	100
Load Balance	75.1	94	97.76	99	99.43	100	100
Communication Efficiency	73.4	93.26	97	98.4	99	99.53	100

Table 11 Results for VGG_19_M_L_R trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	58.93	86.23	95.16	97.7	98.67	99.83	100
Load Balance	57.03	83.53	93.63	97.7	99.27	100	100
Communication Efficiency	55.13	83.2	92.4	97	98.1	99.6	100

Table 12 Results for VGG_19_M_LR_R trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	60	88.4	96.3	98.5	99.2	99.8	100
Load Balance	54.1	82.6	92.7	96.8	98.1	99.76	100
Communication Efficiency	52.87	80.93	93.27	97.17	98.43	99.76	99.93

Table 13 Results for VGG_19_A_LR_R trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	85.3	97.1	98.67	99.4	99.77	100	100
Load Balance	76.01	92.6	96.67	98.43	99	100	100
Communication Efficiency	73.4	93.03	97.33	98.33	99.16	100	100

A.4 Custom

Table 14 Results for Custom_1 trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	2.5	5.5	8	9	12.5	25	42.5
Load Balance	3	5	6.5	11.5	13	27.5	37.5
Communication Efficiency	2.5	8	11	15	20	37	55.5

Table 15 Results for Custom_2 trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	3	4.5	7.5	10.5	14.5	29	45.5
Load Balance	4.5	10	14	15.5	18	34	42
Communication Efficiency	6.5	10	11	12.5	20.5	35.5	57.5

Table 16 Results for Custom_3 trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	3.5	6	7	8.5	12.5	23.5	35
Load Balance	2.5	5.5	7.5	10	13	26	33.5
Communication Efficiency	2	5.5	8.5	11.5	16	36	57

Table 17 Results for Custom_4 trained on pre-processed data

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	1	3.5	4	5.5	8.5	24.5	39
Load Balance	1	4	5.5	8.5	10	23	32
Communication Efficiency	3	6.5	9	13.5	19.5	37.5	54.5

A.5 Importance of Data

Table 18 Results for VGG_19_A_LR_R trained on 5′000 images

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	69.93	88.33	94.63	97.07	98.03	99.57	99.76
Load Balance	60.6	82.93	92.17	95.4	97.03	99.17	99.73
Communication Efficiency	58.1	80.77	89.97	94.27	96.53	99.03	99.63

Table 19 Results for VGG_19_A_LR_R trained on 10′000 images

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	80.47	94.63	97.37	98.33	99.2	99.9	100
Load Balance	68.33	88.7	94.8	97.23	98.13	99.57	99.87
Communication Efficiency	67.3	88.93	94.4	96.8	98.07	99.6	99.93

Table 20 Results for VGG_19_A_LR_R trained on 15′000 images

<	1%	2%	3%	4%	5%	10%	15%
Parallel Efficiency	83.23	95.6	98.1	99	99.37	99.8	99.87
Load Balance	72.6	92.93	97.33	98.67	99.47	100	100
Communication Efficiency	70.23	92.87	97.03	98.3	98.8	99.5	99.93