UNIVERSITY OF KENT

MASTER THESIS

Implementation and comparison of neural networks on CPU and GPU

Author:
Tom HERMANN

Supervisor:
Dominique CHU

A thesis submitted in fulfillment of the requirements for the degree of Artificial Intelligence - MSc

in the

School of Computing

November 3, 2023

width=!,height=!,pages=-

UNIVERSITY OF KENT

Abstract

School of Computing

Artificial Intelligence - MSc

Implementation and comparison of neural networks on CPU and GPU

by Tom HERMANN

Abstract The problems that artificial intelligence has to solve are evolving day by day, and this evolution is reflected in a considerable increase in the difficulty of the tasks to be accomplished, which has led to the use of ever larger models. But the more complex the model, the longer it takes to learn. This is where this study comes into its own, as even the slightest optimization can have a significant impact. We therefore chose to compare two libraries on two different languages, Julia Flux and Tensorflow Python. The comparison focused on the use of Central Processing Units (CPU), although a more minor comparison was made on Graphics Processing Units (GPU). It became clear that for the use of large neural networks, with several million parameters, Tensorflow Python was more suitable, with better overall performance, shorter process times, lower resource requirements and better scalability. However, Julia Flux performs better on smaller models or on less powerful hardware, whether GPUs or weak CPUs.

Contents

List of Figures

List of Tables

List of Abbreviations

GPU	**G**raphic **P**rocessing **U**nit
CPU	**C**entral **P**rocessing **U**nit
TPU	**T**ensor **P**rocessing **U**nit
AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etworks
NN	**N**eural **N**etwork

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

Chapter 1

Introduction

1.1 Backgound

The rapid evolution of technology has ushered in an era where computational power is paramount. As data continues to grow both in volume and complexity, the need for efficient computational methods and hardware has become increasingly evident. This is especially true in the realm of machine learning and, more specifically, neural networks. Neural networks, inspired by the human brain's architecture, have become the cornerstone of many modern artificial intelligence (AI) applications, from image recognition to natural language processing.

Historically, Central Processing Units (CPUs) have been the primary hardware for executing most computer programs, including those related to AI. CPUs are designed for general-purpose tasks and can handle a wide variety of operations. They excel in tasks that require sequential processing and have a broad set of instructions to manage multiple tasks simultaneously.

However, with the rise of graphics-intensive applications, especially video games, Graphics Processing Units (GPUs) were developed. Initially designed to accelerate image rendering, GPUs consist of thousands of small cores that can process tasks in parallel. This architecture, as it turns out, is particularly well-suited for the matrix and vector operations that are fundamental to neural network computations.

Recognizing the potential of GPUs in accelerating neural network training, researchers and practitioners began to harness their power for this purpose. Frameworks like TensorFlow in Python and Flux in Julia emerged, offering tools to develop neural networks and leverage both CPUs and GPUs. The choice between CPU and GPU, and between different frameworks, however, is not always straightforward. Factors such as the size of the dataset, the complexity of the neural network architecture, and the specific operations involved can influence which hardware and framework combination is most efficient.

This thesis delves into the comparative performance of CPUs and GPUs in the context of neural network training, focusing on the capabilities of Python's TensorFlow and Julia's Flux. Through this exploration, we aim to provide insights that can guide practitioners in selecting the optimal setup for their specific neural network applications.

1.2 Importance

The realm of machine learning and artificial intelligence (AI) has witnessed an exponential growth in the past decade. This growth is not just in terms of applications but also in the sheer volume of data and the complexity of models being used. The increasing demand for computational power in these domains is undeniable. As

models grow in depth and breadth, the computational requirements for training and inference also surge [9]

.

GPUs, originally designed for graphics rendering, have emerged as a game-changer in this scenario. Their architecture, which allows for parallel processing of thousands of threads, is particularly well-suited for the matrix operations that dominate neural network computations. The benefits of using GPUs for neural network training are manifold. For instance, training times can be significantly reduced, allowing for faster iterations and model development. Additionally, some studies suggest that training deep learning models on GPUs can lead to better accuracy, possibly due to the ability to process larger batches of data simultaneously [9].

However, the transition to GPU-based training is not without challenges. The cost associated with high-performance GPUs can be prohibitive for many researchers and institutions. Moreover, leveraging the full potential of GPUs often requires specialized software and libraries, adding to the complexity of the setup. There's also the challenge of memory constraints, as GPUs, despite their computational prowess, often have limited memory compared to traditional CPUs.

Looking ahead, the future of neural network training is poised for even more innovations. As the demand for computational power continues to grow, there's potential for the development of even more powerful hardware tailored specifically for AI workloads. This could include specialized AI chips or further advancements in GPU architectures.

1.3 Objective

The ever-evolving landscape of computational hardware and software frameworks presents a myriad of choices for researchers and practitioners in the field of machine learning. Among these choices, the decision to utilize CPUs or GPUs for neural network training remains pivotal. While both hardware options have their merits, understanding their comparative performance is crucial for efficient and effective model training.

The primary objective of this thesis is to conduct a succinct yet comprehensive comparison between CPUs and GPUs in the context of neural network training. This comparison aims to shed light on the advantages and limitations of each hardware type, providing insights that can guide the selection process for various machine learning tasks.

Furthermore, a significant portion of this investigation will focus on the performance of two prominent deep learning frameworks: TensorFlow and Flux. Both libraries have garnered attention in the machine learning community, but their performance, especially on GPUs, has not been extensively compared. By benchmarking these frameworks on GPU hardware, this thesis aims to observe and highlight the notable differences between TensorFlow and Flux. Such a comparison will not only provide a clearer understanding of each library's strengths and weaknesses but also offer valuable insights for those looking to optimize their neural network training pipelines.

1.4 Scope

This research endeavors to provide a comprehensive comparison between two distinct deep learning frameworks, specifically focusing on their efficacy in classification methods. Central to our investigation is the utilization of GPUs to train deep learning models, with the MNIST database serving as our primary dataset. A pivotal aspect of our study will be to scrutinize the correlation between the size of the neural network and the consequent training time, aiming to elucidate how increasing complexity impacts computational demands.

Chapter 2

Literature Review

2.1 Historical Perspective

The development of CPUs and GPUs has been a cornerstone in the evolution of computational capabilities. The origins of these technologies can be traced back to the late 20th century.

In the early days, CPUs were primarily designed for sequential processing tasks, while GPUs, originally intended for rendering graphics, have evolved into highly parallel architectures suitable for a wide range of applications [1]. By the turn of the millennium, the potential of GPUs for non-graphical tasks began to be recognized. For instance, the work by Miller et al. in 2010 highlighted the increasing need for powerful computational resources, especially in the realm of biological data analysis [12]. Similarly, Bonny et al. showcased the potential of hybrid multiprocessor techniques, leveraging both CPU and GPU, for bioinformatics sequence alignment [4].

While CPUs have traditionally been the workhorse of general-purpose computing, the rise of GPUs has expanded the computational landscape. Originally designed for rendering graphics, GPUs have evolved to handle a wide array of non-graphical tasks, particularly those that benefit from parallel processing. For instance, the development of the gem5-gpu simulator showcased the potential of modeling tightly integrated CPU-GPU systems, emphasizing the versatility of GPUs beyond graphics [13].

As the 2010s progressed, the application of GPUs expanded beyond traditional domains. Christley et al. demonstrated the use of GPUs in simulating 3D epidermal development, emphasizing the potential of GPUs in biological modeling [5]. Another study from the same period highlighted the efficiency of GPUs in performing small 2D convolutions, a foundational operation in many scientific computations [2].

Parallel to the evolution of CPUs and GPUs, the field of Artificial Neural Networks (ANNs) has seen significant advancements. The late 20th century witnessed foundational work in ANNs, with applications for learning control of robotic manipulators based on self-tuning of computed torque gains [17] to the development of neural network modules for rapid application in accounting and finance [6]. By the early 2000s, the potential of ANNs in simulating biological processes was being explored, as evidenced by the work of Astor and Adami [3].

The recent decade has seen a resurgence in the interest and application of ANNs, especially with the advent of deep learning. Walter's 2023 paper provides a comprehensive overview of the rapid advancements in machine learning, emphasizing the social risks and benefits associated with these technologies [18]. Another recent study by Shao and Shen delves into the comparison between ANNs and brain networks, highlighting the potential future directions for ANN development [14].

In summary, the journey of CPUs, GPUs, and ANNs has been marked by continuous innovation and expansion of applications. From their foundational roles in computing and graphics to their current applications in diverse fields like biology, finance, and artificial intelligence, these technologies have profoundly influenced the trajectory of scientific and technological advancements.

2.2 Previous Work

The comparison between CPUs and GPUs for neural network training has been a topic of interest for researchers, given the rapid advancements in both hardware technologies and the increasing complexity of neural network architectures. Several studies have delved into this comparison, each considering specific tasks, methodologies, findings, and inherent limitations.

In a comprehensive study by Elshawi et al., six popular deep learning frameworks, including TensorFlow, were evaluated using various deep learning architectures such as Convolutional Neural Networks (CNN), Faster Region-based Convolutional Neural Networks (Faster R-CNN), and Long Short Term Memory (LSTM) networks [7]. Their evaluation spanned different metrics, including accuracy, training time, convergence, and resource consumption patterns. The experiments were conducted on both CPU and GPU environments using diverse datasets. Their findings highlighted the distinct performance characteristics of each framework, emphasizing the advantages of GPU acceleration in training deep learning models.

Another study by McNally et al. aimed to predict the price of Bitcoin using machine learning [11]. While the primary focus was on the prediction accuracy, the study also benchmarked deep learning models on both CPU and GPU. The results showcased that training on GPU outperformed the CPU implementation by a significant margin, reinforcing the computational advantages of GPUs for specific neural network tasks.

Zeng et al. explored the energy-efficient implementation of federated edge learning in wireless networks, emphasizing the benefits of CPU-GPU heterogeneous computing [20]. Their study introduced a novel computation-and-communication resource management framework, which included bandwidth allocation, CPU-GPU workload partitioning, and speed scaling. The results highlighted the energy efficiency gains achieved through optimal resource management in CPU-GPU integrated systems.

While these studies provide valuable insights into the comparative advantages of CPUs and GPUs for neural network training, they also come with limitations. For instance, the choice of datasets, the specific neural network architectures considered, and the hardware configurations can influence the outcomes. Moreover, the rapid evolution of hardware technologies means that findings from older studies might not hold true for the latest generation of CPUs and GPUs.

In conclusion, while GPUs generally offer superior performance for neural network training due to their parallel processing capabilities, the choice between CPU and GPU should be based on the specific requirements of the task, the available hardware, and the associated costs.

2.3 Frameworks

Neural network training has witnessed a surge in interest, leading to the development of numerous software frameworks to facilitate this process. Among the myriad

of available frameworks, TensorFlow and Flux stand out, and they have been chosen for comparison in this study for several compelling reasons.

TensorFlow, developed by Google Brain, is written in Python and has quickly become one of the most popular deep learning frameworks. Its widespread adoption can be attributed to its comprehensive libraries, scalability, and support for various hardware platforms, including CPUs, GPUs, and TPUs. TensorFlow's flexibility allows it to cater to a broad spectrum of tasks, from image classification to natural language processing. However, like any framework, TensorFlow has its limitations. Some users find its static computation graph approach less intuitive, and its verbosity can sometimes pose challenges for beginners.

On the other hand, Flux is a machine learning library for Julia, a high-level, high-performance programming language. Flux's design philosophy revolves around simplicity and extensibility. It offers a more dynamic approach to neural network training, making it easier for users to experiment with custom models. While Julia and Flux might not be as widely recognized as Python and TensorFlow, they offer competitive performance and a refreshing take on neural network design and training.

Comparing TensorFlow and Flux is not just a matter of pitting two frameworks against each other. It's about understanding the nuances of two different programming languages and their respective ecosystems. Such a comparison is crucial because it sheds light on how different languages and their libraries can influence the efficiency, ease of use, and flexibility of neural network training. This understanding can guide researchers and practitioners in choosing the right tools for their specific needs and in recognizing the strengths and weaknesses inherent in each approach.

Chapter 3

Methodology

3.1 Hardware Specifications

In the realm of neural network training, the hardware's capabilities play a pivotal role in determining the efficiency and speed of the training process. The system used for this thesis boasts a robust set of specifications tailored for high-performance computing tasks.

The CPU of the system is an Intel Xeon, equipped with 2 virtual CPUs. The number of cores, in this case, two, is indicative of the CPU's ability to handle multiple tasks simultaneously. This parallel processing capability is crucial for tasks that can be distributed across cores, enhancing the overall computational throughput. The CPU operates at a clock speed of 2.30GHz, determining the rate at which it can execute instructions. A higher clock speed, like the one in this setup, ensures that tasks are processed more swiftly, which is especially beneficial for iterative processes inherent in neural network training.

Complementing the CPU is the Nvidia T4 GPU, a powerhouse in graphical processing. The type of graphics card, particularly one as advanced as the Nvidia T4, is essential for tasks that demand high graphical processing power, such as image-based neural network training. With a performance capability of 8.1 TFLOPS and support for mixed precision, it is well-suited for intricate computations. The GPU comes with a memory of 6GB, operating at a memory clock of 1.59GHz. This substantial memory capacity ensures that the GPU can handle large datasets efficiently, a critical aspect for training comprehensive neural networks.

Furthermore, the system is equipped with 12GB of RAM, providing ample space for data storage and ensuring smooth operations, especially when dealing with extensive datasets. The available disk space stands at 358GB, offering sufficient room for data storage and software installations.

In summary, the hardware specifications of the system provide a balanced combination of processing power and storage capacity, making it well-equipped for the demands of neural network training.

3.2 Software Specifications

In this study, the primary focus is on evaluating the performance of neural networks using two prominent software frameworks: TensorFlow with Keras in Python and Flux in Julia. Both frameworks have been chosen due to their widespread recognition and distinct capabilities in the realm of deep learning.

TensorFlow, an open-source library developed by Google Brain, has established itself as a leading tool for machine learning and deep learning tasks. Paired with Keras, a high-level neural networks API, TensorFlow offers an intuitive interface for

building and training complex models. For this research, we utilized TensorFlow in conjunction with Python 3.10, capitalizing on Python's extensive ecosystem and TensorFlow's robust functionalities.

On the other hand, Flux, a machine learning library designed for Julia, offers a fresh perspective on neural network training. With Julia's version 1.8.3, Flux provides a seamless experience, combining the high-level expressiveness of Julia with the power and flexibility required for advanced neural network architectures. Julia's performance-centric design ensures that the neural network training is both efficient and effective.

To facilitate GPU-accelerated computations, the NVIDIA CUDA Toolkit version 12.0 was employed. CUDA's parallel computing platform and application programming interface model leverages the power of the NVIDIA T4 GPU, ensuring that the neural network training is optimized for GPU computations.

Lastly, to manage and allocate resources for the GPU tasks, SLURM, a highly scalable cluster management and job scheduling system, was utilized. Specifically, the GPU partitioner in SLURM ensured that the computational tasks were appropriately distributed and executed on the GPU, maximizing efficiency and performance.

In essence, the chosen software stack, encompassing TensorFlow, Flux, CUDA, and SLURM, provides a comprehensive environment for rigorous neural network training and evaluation, ensuring both accuracy and computational efficiency.

3.3 Dataset

For the purpose of this study, the MNIST dataset was chosen as the primary source of data to train and evaluate the artificial neural networks. The MNIST dataset, widely recognized in the machine learning community, comprises handwritten digits and serves as a benchmark for various classification algorithms.

The dataset contains a total of 70,000 grayscale images, each of size 28x28 pixels. These images are split into a training set of 60,000 images and a test set of 10,000 images. Each image is labeled with a digit from 0 to 9, representing the handwritten digit in the image. The dataset is stored in a matrix format, where each row corresponds to an image and each column corresponds to a pixel value.

Before feeding the data into the neural network models, several preprocessing steps were undertaken using Julia. Firstly, the dataset was reshaped to fit the input requirements of the neural network. This involved reshaping each image to dimensions of 28x28x1. Subsequently, the target values were one-hot encoded, transforming them into binary matrices for better compatibility with classification tasks. Lastly, to ensure the data values fall within a similar scale, the pixel values of the images, originally ranging from 0 to 255, were normalized by dividing them by 255. This normalization step aids in improving the convergence speed and overall performance of the neural network during training.

3.4 Evaluation Metrics

Evaluating the performance of artificial neural networks requires a set of well-defined metrics that can provide insights into various aspects of the model's behavior. In this study, we have chosen a combination of metrics that not only assess the model's accuracy but also its efficiency and resource utilization.

- **Epoch Memory Usage**: This metric measures the amount of memory consumed during each training epoch. It provides insights into the memory efficiency of the model and helps in identifying potential bottlenecks or memory leaks. Monitoring memory usage is crucial, especially when working with limited hardware resources, to ensure smooth training sessions.
- **Epoch Time**: Representing the duration taken for each epoch to complete, this metric is indicative of the model's computational efficiency. By tracking the epoch time, we can gauge the speed of the training process and make necessary adjustments to optimize training times.
- **Throughput**: Throughput measures the number of samples processed per unit of time, typically per second. A higher throughput indicates that the model can handle more data in less time, making it a valuable metric for assessing the model's scalability and efficiency.
- **Accuracy**: One of the most straightforward and commonly used metrics, accuracy calculates the proportion of correctly predicted labels to the total predictions made. It provides a clear picture of the model's performance in terms of its prediction capabilities.
- **Loss**: The loss metric, often referred to as the cost or error, quantifies how well the model's predictions align with the actual labels. A lower loss value indicates better model performance. Monitoring the loss during training helps in understanding the model's convergence behavior and can be used to adjust hyperparameters or detect issues like overfitting.

In essence, these metrics collectively offer a comprehensive view of the model's performance. While accuracy and loss provide insights into the model's predictive capabilities, epoch memory usage, epoch time, and throughput shed light on its efficiency and resource utilization. Utilizing these metrics ensures a holistic evaluation, enabling informed decisions and refinements throughout the model's development and deployment phases.

Chapter 4

Results

4.1 Models and details

For this comparison, we used 3 networks of different sizes. A small network, with 101,770 parameters, is a simple feedforward architecture. A medium-sized network, a convolutional neural network (CNN) with 220,234 parameters, introduces convolutional layers and pooling, which adds to its complexity. Finally, a large network, the most complex of the trio, has 1,683,722 parameters and incorporates deeper convolutional layers. All these networks use the same hyperparameters. All the layers and hyperparameters of neural networks are described in Appendix A.

4.2 GPU Performances

To compare the performance of the two libraries, we used the 3 different networks to observe the evolution of performance as the number of parameters to be controlled increased.

4.2.1 Time

Across these architectures, we observed distinct performance patterns.

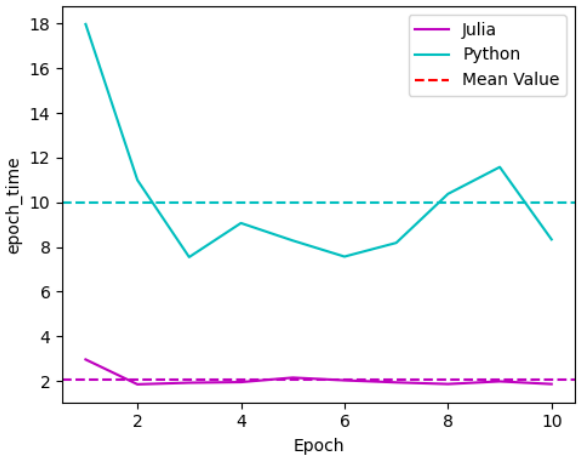

FIGURE 4.1: Epoch times (s) for 10 epochs, small NN

Figure 4.1 show that for the Small network on GPU, TensorFlow exhibited an epoch time of 10.00 second while Flux had an epoch time of 2.05 second on average.

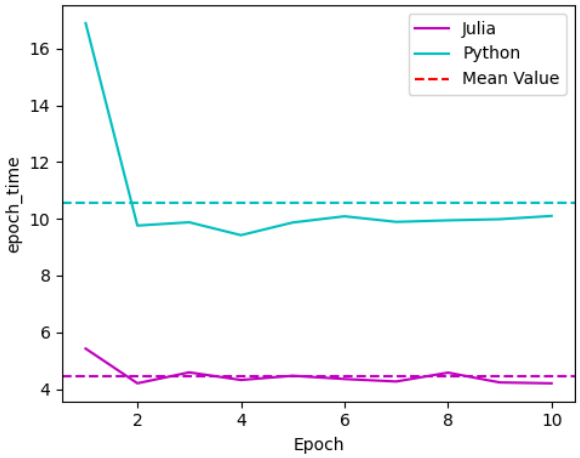

FIGURE 4.2: Epoch times (s) for 10 epochs, medium NN

When network complexity increased to medium, TensorFlow's splicing time was 10.58 seconds, while Flux took an average of 4.46 seconds, as shown in Figure 4.2.

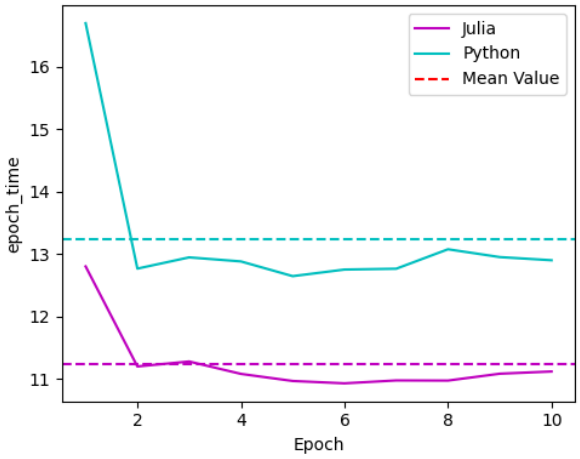

FIGURE 4.3: Epoch times (s) for 10 epochs, big NN

Finally, Figure 4.3 shows that for the Big network, TensorFlow's splicing time increased to 13.24 seconds and Flux's to 11.24 seconds.

Language	NN	Standard Deviation
Python Tensorflow	Small	2.98
	Medium	2.11
	Big	1.15
Julia Flux	Small	0.31
	Medium	0.34
	Big	0.53

TABLE 4.1: Standard deviation of each NN type

Table 4.1 shows that, as well as having an average epochtime up to 5 times greater, the Tensorflow epochtimes are much more disparate than those of Julia.

4.2.2 Throughput

Throughput, which measures the number of operations a system can perform in a given period of time, is another essential measure. In this study, the delay is set at 1 second. The unit of throughput is the number of operations per second.

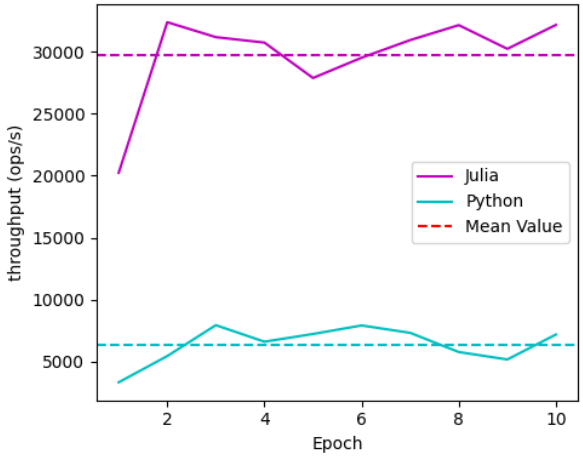

FIGURE 4.4: Throughput (ops/s) for 10 epoch, small NN

For the Small network, TensorFlow achieved an average throughput of 6399 operations per second, well below Flux's average value of 29737 (ops/s).

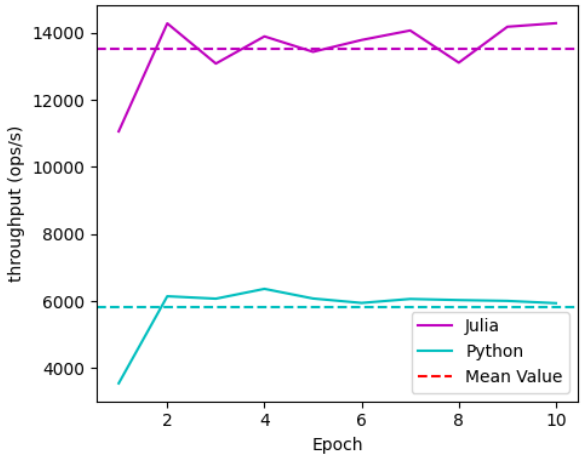

FIGURE 4.5: Throughput (ops/s) for 10 epoch, medium NN

The Medium network saw TensorFlow's throughput at 5821 (ops/s), while Flux managed 13517 operations per second.

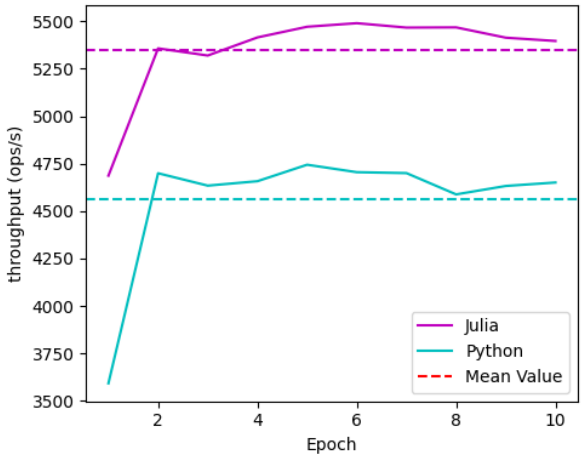

FIGURE 4.6: Throughput (ops/s) for 10 epoch, big NN

The Big network had TensorFlow's throughput at 4560 ops/s, and Flux's at 5348 ops/s.

Given that for each epoch, the number of patches was equal whatever the language, the epochtime and throughput values are directly linked, so the standar deviation values will be the same. We can therefore make the same observation as for execution time: python has a much lower throughput than julia, regardless of the size of the network, and has much more disparate values.

4.2.3 Accuracy and Loss

While performance is crucial, the accuracy and loss of the models cannot be overlooked. Across all architectures, both TensorFlow and Flux demonstrated comparable accuracies, with minor variations.

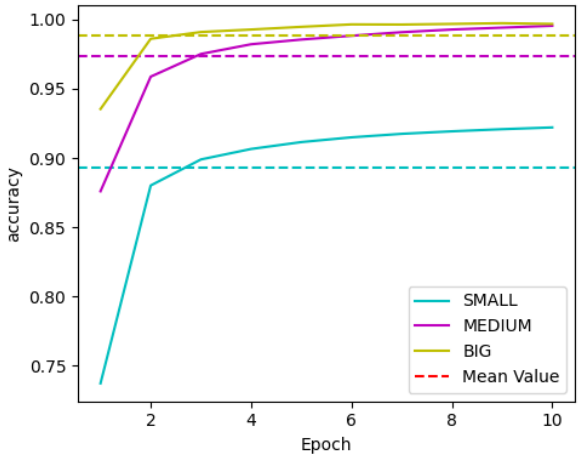

FIGURE 4.7: Julia accuracy for all NN types

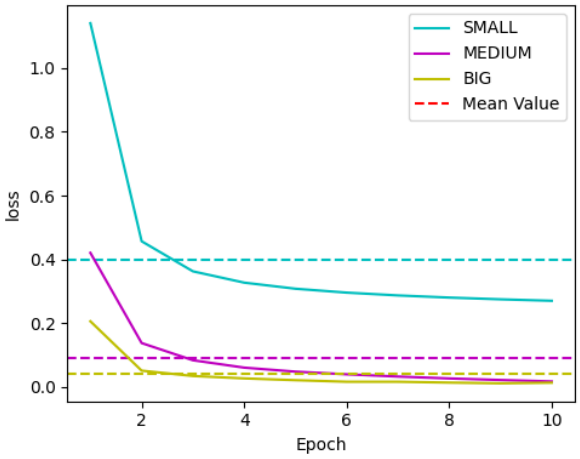

FIGURE 4.8: Julia loss for all NN types

The first and most obvious observation in Figure 4.7 is the constant increase in values, with each epoch having an accuracy greater than or equal to the previous epoch. We can also see that increasing the size of the network gives a higher accuracy each time.

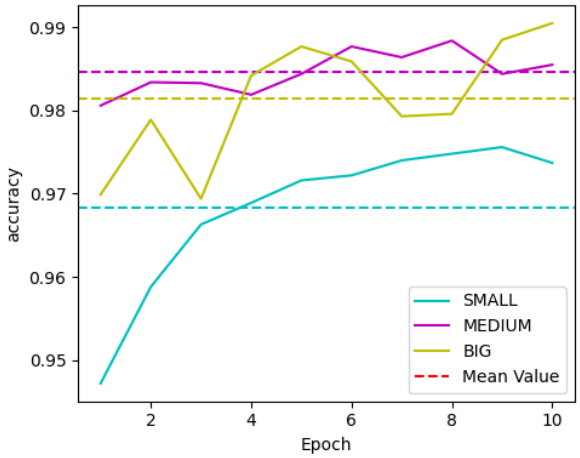

FIGURE 4.9: Python accuracy for all NN types

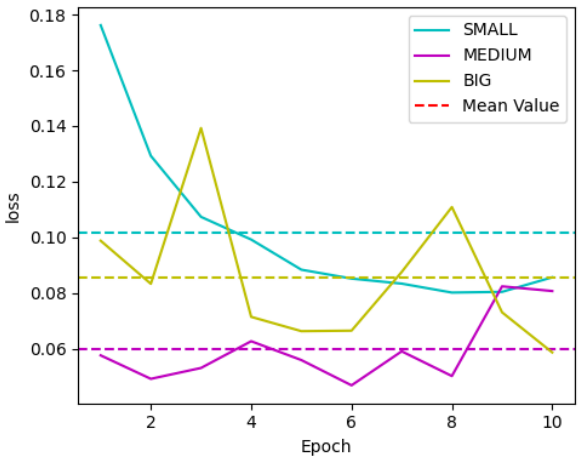

FIGURE 4.10: Python loss for all NN types

In contrast to execution speed, in terms of accuracy, Python has values that fluctuate much less and have higher limit values, which can be seen in Figure 4.9. However, unlike Flux, Tensorflow's accuracy does not follow a constant trend, with some periods having lower accuracy than the previous period, which is not observed in Flux's values. The final observation is that, unlike Flux, the best accuracy is not for the heaviest network but for the medium-sized network.

Figure 4.8 and Figure 4.10 shows that these observations are also valid for loss, where the values have a stable trend and where an increase in size gives better results, which, for loss, is reflected in lower values.

4.2.4 Memory usage

Memory usage only concerns the GPU part. The higher the demand, the less the model will perform on less advanced GPUs. Memory usage per epoch is given in bytes (b).

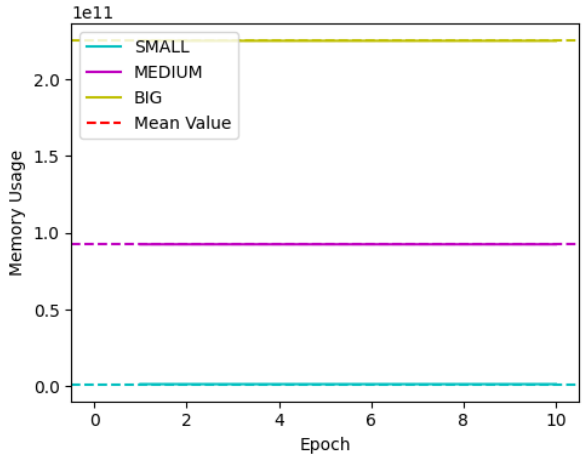

FIGURE 4.11: Memory Usage of Julia for each NN type

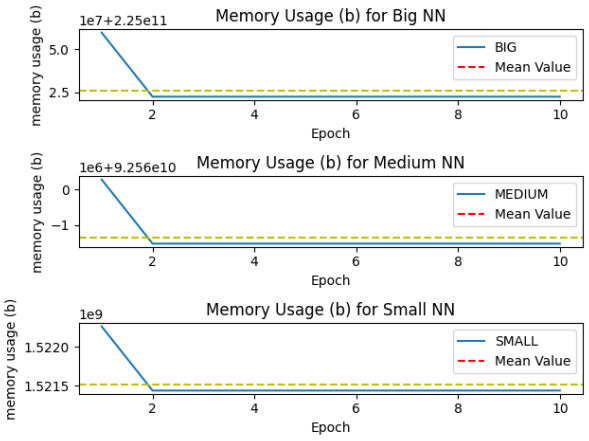

FIGURE 4.12: Memory Usage of Julia for each NN type separate

	First epoch memory usage (GByte)	Second epoch memory usage (GByte)	Coeff
Small	1.41	1.41	1.000165
Medium	86.20	86.20	1.000019
Big	209.60	209.56	1.000539

TABLE 4.2: Coefficient of increase between epoch 2 and 1

Figure 4.11 shows several things. Firstly, memory usage increases as the number of parameters increases. Memory usage is very stable, with little or no variation between each epoch. However, Figure 4.12 shows that the first epoch requires much more memory than the others, however, this increase does not seem to be directly linked to the number of network arguments, as shown in Table 4.2.

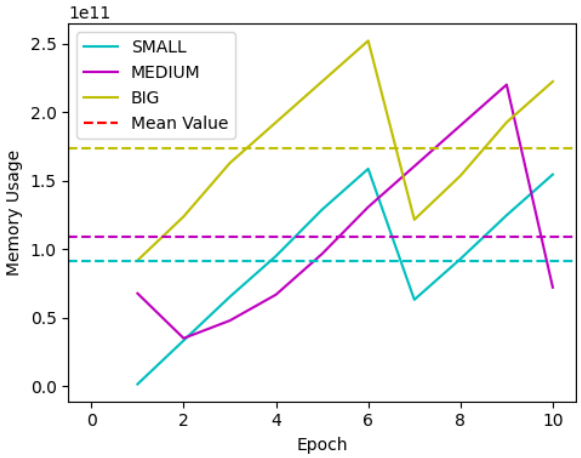

FIGURE 4.13: Memory Usage of Python for each NN type separate

	Mean Julia (GByte)	**Mean Python (GByte)**	**Coeff**
Small	1.417	85.402	60.268
Medium	86.201	101.227	1.174
Big	209.571	161.613	0.771

TABLE 4.3: Python's memory usage coefficient compared with Julia's

Figure 4.13 shows that Python's memory usage is much more chaotic than Julia's and changes enormously from one epoch to the next, although there are similar patterns (particularly between small and large networks). Although Python uses more (up to 80 times more) memory on average on small networks (in this case, small and medium), it uses less than Julia for big networks. This can be seen from Table 4.3.

4.3 CPU Performances

Unlike performance on the GPU, we're only going to look at performance on medium-sized networks. However, the hyper-parameters remain unchanged

4.3.1 Time

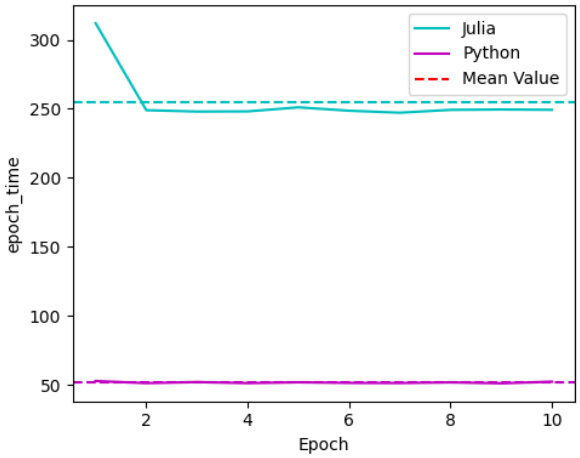

FIGURE 4.14: Epoch times (s) for 10 epochs on CPU

Figure 4.14 tells us several things. Firstly, there is a huge difference between the execution times of Python Tensorflow and Julia Flux. We can see that Python is about 50 times faster than Julia. We then see that although both our curves seem fairly constant, Julia's has a huge difference between the first and second epochs of 50 seconds (i.e. 0.2 times more time). Although apart from this first epoch, the others seem constant in their execution time.

4.3.2 Accuracy and Loss

Once again, although excusion time is important, it is not the only factor that determines a good NN. A fast NN with poor accuracy will be considered useless.

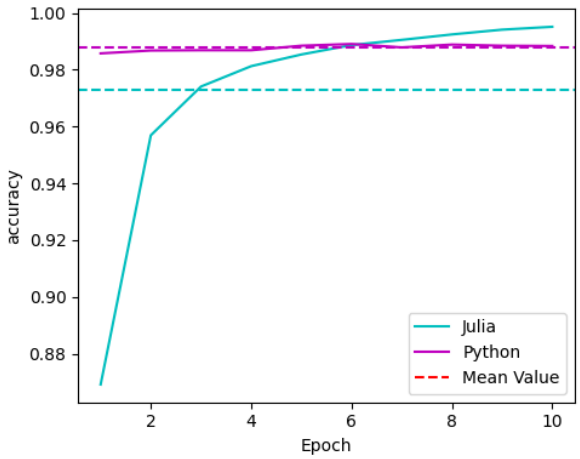

FIGURE 4.15: CPU accuracy

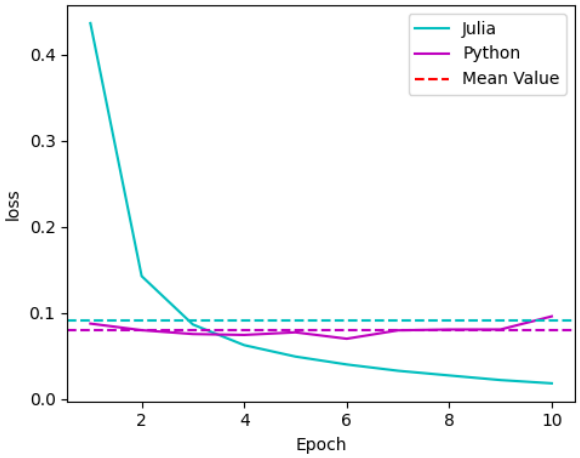

FIGURE 4.16: CPU Loss

Once again, the precision and loss results of the two libraries are similar. Python, in both Figure 4.15 and Figure 4.16, performs slightly better on average, but if we look at the best results from the two networks, we see that Julia performs better, both in terms of accuracy (0.995 for Julia against 0.989 for Python) and loss (0.018 for Julia against 0.069 for Python). What's more, where we see an improvement between each epoch for Julia, Python is sometimes less accurate and has more loss.

Chapter 5

Discussion

5.1 Strengths and Weaknesses

5.1.1 Variation and Exploration

As can be seen from 4.9, 4.10, 4.13 and 4.1 that Python is much more variable than Flux, in terms of accuracy, loss, calculation time and memory management. Several factors may explain these variations. The use of Adam's optimizar has the effect of adapting the learning rate [21], which can lead to variations. Neural network learning involves non-convex optimisation. This means that there are multiple local minima and that the optimisation process can stall or bounce around these minima, leading to variations in loss and accuracy [10]. Sometimes the underlying hardware or software (such as TensorFlow) can introduce non-determinism into the learning process. This is particularly true when using GPUs [8].

All these variations are not necessarily bad, as large variations can lead to better exploration. In the field of optimisation and machine learning, there is a well-known trade-off between exploration (trying out new solutions) and exploitation (refining known solutions). When significant variations between epochs are observed, the model can explore different regions of the solution space rather than exploiting a single region. This can help avoid local minima and find a more global optimum [15] [16].

Similarly, if the model is stuck in a local minimum, significant variations in updates (due to factors such as learning rate, regularisation, etc.) can help the model escape and explore other regions of the solution space [19].

We can therefore say that Python Tensorflow may have better exploration than Julia, but this exploration, in the case of the NMIST dataset, does not always lead to better precision 4.7 4.9. However, the differences in performance between the two aren't enough to say that these variations are fundamentally bad for certain problems. Python Tensorflow a is therefore better than Julia Flux in terms of exploration and equal in precision on the GPU.

5.1.2 Scalability

To determine which of the two libraries we will compare the increase in the number of parameters and their influence on the calculation time and memory used.

Table 5.1 highlights the lack of scalability of Julia compared to Python. When multiplying the number of parameters by 16.54, the execution time of Python was only 1.32 times higher, which was quite similar, whereas Julia saw its execution time multiplied by 5.47. This observation is exacerbated for memory usage where Julia sees its memory consumption increase 147.89 times compared to 1.89 times for Python.

	Small to Medium	**Medium to Big**	**Small to Big**
Param	2.16	7.64	16.54
Python Time	1.05	1.25	1.32
Julia Time	2.17	2.51	5.47
Python Memory usage	1.18	1.59	1.89
Julia Memory usage	60.83	2.43	147.89

TABLE 5.1: Multiplier coefficient for data (time and memory usage) in relation to the increase in the number of parameters

Although, as Table 4.3 shows, Julia generally uses much less memory than Python, this trend is reversed as we use more and more models. For a model with 1 million parameters, Julia uses more memory than Python. Remember that today's models use billions of parameters (175 billion parameters for chat gpt4, for example). The excesive increase in resources required by Julia shows that for large models, Python has much better scalability.

5.2 Difference between CPU and GPU

5.2.1 Code compatibility

To go from a CPU to a GPU version, some changes are necessary, but where Tensorflow has a GPU version available and requires no code modification, Flux will require a code modification.

Indeed, when defining the model, a redirection to GPU will have to be made:

LISTING 5.1: Flux CPU code

```
using Flux

...

model = Chain(
    # model
)
...
```

LISTING 5.2: Flux GPU code

```
using Flux, CUDA

...

model = Chain(
    # model
) |> gpu
...
```

Although minimal, a code change will still be necessary, unlike Tensorflow.

5.2.2 Performances

Whether on CPU or GPU, we can see that for an average model, Julia outperforms Python 4.14 4.2. However, this is not true when we look at accuracy and loss, where, contrary to GPU results 4.7 4.8 4.9 4.10, Julia's CPU results are better than those of Python 4.15 4.16. Flux CPU is therefore more optimized than Tensorflow CPU, with a faster execution time and better accuracy and loss for a model with around 200,000 parameters A.

5.3 Limitation and Future work

5.3.1 Neural networks

In this study, we measured the performance of just three networks with similar structures and relatively small sizes.

Size

As mentioned above, the neural networks we use are much larger. Using neural networks with sizes closer to what is actually used would be very interesting, not least to study the scalability of libraries in an appronfodial way.

Layers

In our three networks, we have three types of layer, Conv2D and Dense A. However, there are many different layers (Dropout, RNN, LSTM, ...), all with different numbers of parameters, different resource costs ... In the current state of the study, we can't say whether, for example, Flux is particularly suited to Natural Language Processing, a dicipline that uses a lot of RNN, because we don't use this type of layer. This is undoubtedly the most limiting aspect of this research, as it covers only two types of layer.

5.3.2 Dataset

During this study we used the well-known MNIST dataset, but many other datasets are available and have their own specificities, and although the type of hardware used (GPU or CPU) has no direct link to the dataset, it would be interesting to see if we can observe major changes depending on the dataset used.

5.3.3 Energy consumption

Although, as we've seen, GPUs are much faster, they also consume much more electricity than CPUs, so it's fair to ask whether using a GPU is energy-efficient.

Chapter 6

Conclusion

The aim of this thesis, which was to compare Flux and Tensorflow in the GPU computing environment, has been achieved. Although not complete, this study was able to determine the best use cases for each library.

When using lightweight neural networks, i.e. around 500.00 parameters, Flux tends to have better performance and computation times than Tensorflow, whether on GPU or CPU. What's more, its low GPU memory consumption gives it a further advantage over neural networks with few parameters. However, this also means that if GPUs are less advanced and have less available memory, even on heavier neural networks, Flux may need to transfer data between CPU and GPU, thereby boosting performance and avoiding potential errors.

However, the increase in the number of parameters makes Julia much more demanding in terms of memory and time, without having any repercussions on the performance of the network, unlike Python, which, when using heavier networks (1,000,000 parmeters or more) seems to be much less demanding in terms of memory than Julia, having a similar computation time and similar performance.

What's more, the evolution of the various metrics shows a better evolution than Flux, i.e. the larger the NN, the greater the difference in performance between Tensorflow and Flux. This fact is particularly visible in memory usage, where simply doubling the size of the network also doubles memory usage.

The larger the network, the better the performance of Tensorflow compared with Flux. The smaller the network and the lower the hardware performance, the better Flux's performance compared to Tensorflow.

Appendix A

Neural Networks Configuration

Small:

Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(None, 784)	0
dense_4 (Dense)	(None, 128)	100480
dense_5 (Dense)	(None, 10)	1290

Total params: 101,770
Trainable params: 101,770

Medium:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 64)	0
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 64)	200768
dense_1 (Dense)	(None, 10)	650

Total params: 220,234
Trainable params: 220,234

Big:

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)            (None, 28, 28, 64)        640

 max_pooling2d_4 (MaxPooling  (None, 14, 14, 64)        0
 2D)

 conv2d_5 (Conv2D)            (None, 14, 14, 128)       73856

 max_pooling2d_5 (MaxPooling  (None, 7, 7, 128)         0
 2D)

 conv2d_6 (Conv2D)            (None, 7, 7, 256)         295168

 max_pooling2d_6 (MaxPooling  (None, 3, 3, 256)         0
 2D)

 flatten_4 (Flatten)          (None, 2304)              0

 dense_8 (Dense)              (None, 512)               1180160

 dense_9 (Dense)              (None, 256)               131328

 dense_10 (Dense)             (None, 10)                2570

=================================================================
Total params: 1,683,722
Trainable params: 1,683,722
```

Compilation parameters:

```
Optimizer: Adam(learning_rate=0.001),
Loss: CategoricalCrossentropy()
Metrics: accuracy
```

Appendix B

Model results

TABLE B.1: CPU JULIA-FLUX

Epoch	Epoch Time (s)	Throughput (ops/s)	Accuracy	Loss
1	312.03	192.29	0.8692	0.4366
2	249.00	240.96	0.9569	0.1427
3	248.00	241.93	0.9741	0.0865
4	248.13	241.81	0.9813	0.0626
5	251.07	238.98	0.9854	0.0493
6	248.60	241.35	0.9887	0.0401
7	247.18	242.74	0.9906	0.0328
8	249.23	240.75	0.9925	0.0273
9	249.51	240.47	0.9942	0.0220
10	249.27	240.70	0.9952	0.0182
Mean value	255.20	236.20	0.9728	0.0918

TABLE B.2: SMALL GPU JULIA-FLUX

Epoch	Epoch Time (s)	Epoch Memory Usage (b)	Throughput (ops/s)	Accuracy	Loss
1	2.97	1522261557	20233.63	0.7374	1.1406
2	1.85	1521440840	32378.75	0.8801	0.4569
3	1.92	1521440840	31172.15	0.8989	0.3627
4	1.95	1521440840	30733.99	0.9065	0.3272
5	2.15	1521440840	27878.33	0.9115	0.3081
6	2.03	1521440840	29513.88	0.9149	0.2959
7	1.94	1521440840	30934.66	0.9174	0.2871
8	1.87	1521440840	32141.37	0.9192	0.2804
9	1.98	1521440840	30227.57	0.9208	0.2749
10	1.87	1521440840	32158.55	0.9220	0.2703
Mean value	2.05	1521522911.70	29737.29	0.8929	0.4004

TABLE B.3: MEDIUM GPU JULIA-FLUX

Epoch	Epoch Time (s)	Epoch Memory Usage (b)	Throughput (ops/s)	Accuracy	Loss
1	5.42	92560279460	11061.07	0.8760	0.4206
2	4.20	92558474824	14280.57	0.9587	0.1378
3	4.59	92558474824	13082.29	0.9751	0.0839
4	4.32	92558474824	13893.01	0.9821	0.0607
5	4.47	92558474824	13432.29	0.9855	0.0482
6	4.35	92558474824	13785.64	0.9882	0.0398
7	4.26	92558474824	14068.80	0.9908	0.0328
8	4.58	92558474824	13110.11	0.9927	0.0268
9	4.23	92558474824	14179.74	0.9941	0.0220
10	4.20	92558474824	14285.95	0.9954	0.0174
Mean value	4.46	92558655287.60	13517.95	0.9738	0.0890

TABLE B.4: BIG GPU JULIA-FLUX

Epoch	Epoch Time (s)	Epoch Memory Usage (b)	Throughput (ops/s)	Accuracy	Loss
1	12.80	225059716802	4686.53	0.9353	0.2060
2	11.20	225022488648	5357.10	0.9860	0.0513
3	11.28	225022488648	5319.39	0.9909	0.0345
4	11.08	225022488648	5414.74	0.9927	0.0269
5	10.97	225022488648	5470.96	0.9947	0.0213
6	10.93	225022488648	5489.73	0.9964	0.0164
7	10.98	225022488648	5466.54	0.9963	0.0164
8	10.97	225022488648	5467.64	0.9968	0.0138
9	11.08	225022488648	5413.38	0.9973	0.0115
10	11.12	225022488648	5396.08	0.9968	0.0135
Mean value	11.24	225026211463.40	5348.21	0.9883	0.0412

TABLE B.5: CPU PYTHON-TENSORFLOW

Epoch	Epoch Time (s)	Throughput (ops/s)	Accuracy	Loss
1	52.77	1136.96	0.9858	0.0876
2	51.25	1170.72	0.9868	0.0799
3	52.03	1153.21	0.9869	0.0755
4	51.24	1171.06	0.9869	0.0744
5	51.87	1156.65	0.9885	0.0775
6	51.42	1166.79	0.9891	0.0700
7	51.31	1169.28	0.9879	0.0797
8	51.80	1158.20	0.9889	0.0807
9	51.10	1174.13	0.9885	0.0808
10	52.31	1146.97	0.9884	0.0960
Mean value	51.71	1160.40	0.9878	0.0802

TABLE B.6: SMALL GPU PYTHON-TENSORFLOW

Epoch	Epoch Time (s)	Epoch Memory Usage (b)	Throughput (ops/s)	Accuracy	Loss
1	17.98	1325435136	3337.58	0.9472	0.1762
2	11.00	33251964928	5454.87	0.9588	0.1293
3	7.55	65101342208	7947.02	0.9663	0.1074
4	9.07	94892749824	6613.40	0.9689	0.0992
5	8.29	128800096768	7236.86	0.9716	0.0884
6	7.57	158591096832	7923.57	0.9722	0.0852
7	8.19	63042964992	7327.70	0.9740	0.0834
8	10.38	92833965056	5780.72	0.9748	0.0802
9	11.58	124683342336	5181.05	0.9756	0.0805
10	8.34	154474545664	7194.13	0.9737	0.0857
Mean value	10.00	91699750374.40	6399.69	0.9683	0.1015

TABLE B.7: MEDIUM GPU PYTHON-TENSORFLOW

Epoch	Epoch Time (s)	Epoch Memory Usage (b)	Throughput (ops/s)	Accuracy	Loss
1	16.90	67615483136	3551.05	0.9806	0.0577
2	9.76	34846414848	6146.72	0.9834	0.0492
3	9.88	47749147648	6074.19	0.9833	0.0531
4	9.42	66695792128	6367.11	0.9819	0.0627
5	9.87	96486792192	6079.24	0.9844	0.0560
6	10.09	130694466560	5947.19	0.9877	0.0469
7	9.89	160485670400	6065.92	0.9864	0.0591
8	9.95	190276670464	6032.58	0.9884	0.0502
9	9.99	220067466752	6008.17	0.9844	0.0824
10	10.10	72001900544	5939.92	0.9855	0.0807
Mean value	10.58	108691980467.20	5821.21	0.9846	0.0598

TABLE B.8: BIG GPU PYTHON-TENSORFLOW

Epoch	Epoch Time (s)	Epoch Memory Usage (b)	Throughput (ops/s)	Accuracy	Loss
1	16.70	91700360960	3592.85	0.9699	0.0988
2	12.77	123790825472	4699.40	0.9789	0.0833
3	12.95	162771043840	4634.10	0.9694	0.1393
4	12.88	192562043392	4657.43	0.9842	0.0715
5	12.65	222352764928	4744.28	0.9877	0.0663
6	12.75	252143764992	4704.98	0.9859	0.0665
7	12.77	121492761600	4700.11	0.9793	0.0876
8	13.08	153581546496	4588.00	0.9796	0.1109
9	12.95	192561764864	4632.38	0.9885	0.0731
10	12.90	222352764928	4650.61	0.9905	0.0587
Mean value	13.24	173530964147.20	4560.41	0.9814	0.0856

Bibliography

[1] "A History of the AMD Opteron Processor". In: *ResearchGate* (). URL: https://www.researchgate.net/publication/221517571_A_History_of_the_AMD_Opteron_Processor.

[2] R. Al Umairy and S. Vialle. "On the use of small 2d convolutions on GPUs". In: *Proceedings of the 2010 International Multiconference on Computer Science and Information Technology*. 2010. URL: https://hal.inria.fr/inria-00493873/file/A4MMC-al-umairy.pdf.

[3] J. C. Astor and C. Adami. "A Developmental Model for the Evolution of Artificial Neural Networks". In: *Artificial Life* 6.3 (2000), pp. 189–218. URL: https://doi.org/10.1162/106454600568834.

[4] Mohamed Bonny, Zaid Al-Ars, and Koen Bertels. "An Adaptive Hybrid Multiprocessor technique for bioinformatics sequence alignment". In: *Parallel Computing* (2010). URL: https://repository.kaust.edu.sa/bitstream/10754/236113/1/An_Adaptive_Hybrid_Multiprocessor_Technique_for_Bioinformatics_Sequence_Alignment.pdf.

[5] Scott Christley, Qing Nie, and Xiaohui Xie. "Integrative multicellular biological modeling: a case study of 3D epidermal development using GPU algorithms". In: *BMC Systems Biology* (2010). URL: https://bmcsystbiol.biomedcentral.com/track/pdf/10.1186/1752-0509-4-107.

[6] Jerry Coakley and David Brown. "Artificial neural networks in accounting and finance: modeling issues". In: *International Journal of Intelligent Systems in Accounting, Finance and Management* (2000). URL: https://onlinelibrary.wiley.com/doi/pdfdirect/10.1002/1099-1174%28200006%299%3A2%3C119%3A%3AAID-ISAF182%3E3.0.CO%3B2-Y.

[7] Radwa Elshawi et al. "DLBench: a comprehensive experimental evaluation of deep learning frameworks". In: *Cluster Computing* (2021). URL: https://link.springer.com/content/pdf/10.1007/s10586-021-03240-4.pdf.

[8] José M. Faria. "Non-determinism and Failure Modes in Machine Learning". In: (2017), pp. 310–316. DOI: 10.1109/ISSREW.2017.64.

[9] F. Iandola et al. "FireCaffe: Near-Linear Acceleration of Deep Neural Network Training on Compute Clusters". In: *CVPR* (2015). URL: http://arxiv.org/pdf/1511.00175.

[10] Prateek Jain and Purushottam Kar. "Non-convex Optimization for Machine Learning". In: *Foundations and Trends® in Machine Learning* 10.3-4 (2017), pp. 142–363. ISSN: 1935-8237. DOI: 10.1561/2200000058. URL: http://dx.doi.org/10.1561/2200000058.

[11] S. McNally, Jason Roche, and Simon Caton. "Predicting the Price of Bitcoin Using Machine Learning". In: (2018). URL: http://trap.ncirl.ie/2496/1/seanmcnally.pdf.

[12] Mark A. Miller, Wayne Pfeiffer, and Terri Schwartz. "Creating the CIPRES Science Gateway for inference of large phylogenetic trees". In: *Gateway Computing Environments Workshop (GCE)*. 2010. URL: http://www.phylo.org/sub_sections/portal/sc2010_paper.pdf.

[13] Jason Power et al. "gem5-gpu: A Heterogeneous CPU-GPU Simulator". In: *Computer Architecture Letters*. 2014. URL: http://research.cs.wisc.edu/multifacet/papers/cal14_gem5gpu.pdf.

[14] Feng Shao and Zheng Shen. "How can artificial neural networks approximate the brain?" In: *Frontiers in Psychology* (2023). URL: https://www.frontiersin.org/articles/10.3389/fpsyg.2022.970214/pdf.

[15] Fathma Siddique, S. Sakib, and M. Siddique. "Handwritten Digit Recognition using Convolutional Neural Network in Python with Tensorflow and Observe the Variation of Accuracies for Various Hidden Layers". In: *Preprints* 2019 (2019), p. 0039. DOI: 10.20944/PREPRINTS201903.0039.V1. URL: https://ieeexplore.ieee.org/document/8975496.

[16] Nidhi Sindhwani et al. "Performance Analysis of Deep Neural Networks Using Computer Vision". In: *EAI Endorsed Transactions on Computer Vision* (2021). DOI: 10.4108/eai.13-10-2021.171318. URL: https://eudl.eu/doi/10.4108/eai.13-10-2021.171318.

[17] M. Teshnehlab and Keigo Watanabe. "Flexible Structural Leaning Control of a Robotic Manipulator Using Artificial Neural Networks". In: *Journal of the Japan Society of Mechanical Engineers* 38.3 (1995), p. 510. DOI: 10.1299/JSMEC1993.38.510.

[18] Yoshija Walter. "The rapid competitive economy of machine learning development: a discussion on the social risks and benefits". In: *AI & SOCIETY* (2023). URL: https://link.springer.com/content/pdf/10.1007/s43681-023-00276-7.pdf.

[19] Kaichao You et al. *How Does Learning Rate Decay Help Modern Neural Networks?* 2019. arXiv: 1908.01878 [cs.LG].

[20] Qunsong Zeng et al. "Energy-Efficient Resource Management for Federated Edge Learning With CPU-GPU Heterogeneous Computing". In: *IEEE Transactions on Wireless Communications* (2020). URL: https://arxiv.org/pdf/2007.07122.

[21] Zijun Zhang. "Improved Adam Optimizer for Deep Neural Networks". In: *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. 2018, pp. 1–2. DOI: 10.1109/IWQoS.2018.8624183.