UNIVERSITÀ DEGLI STUDI DI CATANIA
Dipartimento di Matematica e Informatica
Corso di Laurea Triennale in Informatica

Antonio Finocchiaro

An Outlook on Machine Learning Resource Requirements: Literature and Load Analysis Across Various Configurations

Report on Central Systems work

Professor:
Fabrizio Messina

Academic Year 2023 - 2024

Contents

Abstract

Nowadays Machine Learning and Deep Learning architectures have demonstrated how far the horizon of human capabilities has expanded in recent years. Despite their overlooked mechanisms and output results, the impact they have in the underlying running machines is still underestimated.

In a fine-grained view, what is the influence of the principal hyperparameters of a neural network on the computational and memory load? In this work, we will provide an overview trying to interweave different aspects of the topic.

We will discuss how these challenges can be addressed at various levels of hardware design, ranging from the dataflow strategies of deep neural networks to the availability of hardware components and related areas explored by NVIDIA. Subsequently, we will then create and fine-tune a multilayer perceptron to perform various experiments on a defined dataset, discussing about the related outcomes.

Chapter 1

Introduction

In recent years, the rapid advancement of Machine Learning and Deep Learning algorithms has revolutionized numerous fields, including Computer Vision and Natural Language Processing.

These algorithms and the architectures they rely on, are computationally intensive, given the complexity of the underlying mathematical formulas and the increasing number of parameters involved.

However, the efficiency and performance of these algorithms depend significantly on the underlying hardware infrastructure. Furthermore, as datasets continue to expand, this translates into an increased demand for models capable of automatically analyzing large, complex data and delivering precise results.

This work is divided in two main parts, a theoretically and a practical one. Hence, presented. In the first one, contained in Chapter 2, we will make a brief introduction about the Machine Learning and Deep Learning basics, discussing about how these challenges can be addressed at various levels of hardware design. In Chapter 2.3 we will introduce some computation architectures discussed in [14]; subsequently, in Chapter 2.5, we will explore the Multilayer Perceptron architecture, delving in some hyperparameters it depends on. Lastly, in Chapter 2.4 we will focus on the hardware required, talking about the actual possibilities.

Regarding the second part, in Chapter 3 we will introduce the libraries, dataset model used. Following that, in Chapter 4, we will perform some analyses on the computational and memory requirements for training a neural network, discussing the parameters that influence complexity and the resources needed.

Chapter 2

Literature

Machine Learning (ML) and Deep Learning (DL) are two rapidly expanding fields that continue to attract increasing interest and attention. As data availability increases and computing power advances, the range of the potential applications of these technologies is constantly evolving. With the involving of ML, we manage to extract meaningful information about the raw data that we sampled. There are many tasks to manage. At the same time, due to the significant volume and speed of data generation, a growing amount of computing is moving to the cloud. On other hand, from an individual perspective, the high costs of the actual solutions provided by cloud provider, reinforce the need to perform analyses locally. In Chapter 2.4 we will explore some of the NVIDIA solutions and different local alternatives.

2.1 Machine Learning Applications

As discussed in [14], the wide range of ML applications, ranging from text related to medical imaging tasks, demonstrates its potential to revolutionize other different domains.

For instance, computer vision constitutes the backbone for several applications. A lot of challenging tasks from video material such as object detection, visual tracking or image-related such as deblurring, denoising or impainting are leveraged by modern architectures that improve on the previous ones. Some possible applications of these models regard the first person vision with the development of various glasses able to perform augmented or mixed reality.

Regarding textual data, huge large language models such as GPTs, Claude or LLama have been developed and are revolutionising the daily life of everybody, helping users writing articles or provide an answer to (almost) any

question.

2.2 Machine Learning and Deep Learning basics

These application rely on ML discipline. It constitutes a subset of AI, allowing us to solve specific tasks without explicit programming, as cited by Tom Mitchell in 1997.

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

This is made possible through the development of an architecture that takes inputs and returns one or multiple predictions as outputs.

The learning process is managed by the training procedure which employs various techniques (not explored in this work) to enable the model to approximate a certain function that describes the input data through weights and bias adjusting. From the training procedure, we obtain the weights of the architecture (in a defined format e.g. pth).

By feeding them into the model, we can perform the evaluation phase, where the model is being tested on never-before-seen data in order to determine its ability to predict.

Once trained and tested, the model enters the inference phase, where it applies its learned knowledge to make predictions on new, unseen data in real time or batch processing scenarios.

A key point of the training phase is the feature extraction. Feature extraction plays a crucial role in preparing raw data for ML applications. Traditionally, experts have manually created features designed for specific tasks, such as identifying edges in images for object recognition. These hand-crafted features such as Histogram of Oriented Gradients and Scale Invariant Feature Transform, were designed to manage variations in illumination, 2D matrix transformations such as rotation or scaling.

However, modern approaches based on DL exploit the network itself to automatically learn relevant features directly from the data, significantly improving evaluation metrics across various tasks. This is the main difference between ML and DL.

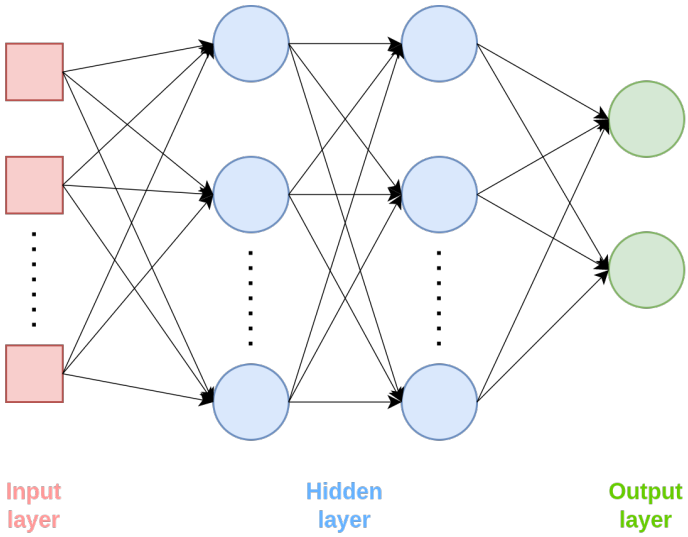

Figure 2.1: Architecture of a Deep Neural Network.

Without losing generalization, we make a brief description about the DNN, introducing some strategies to increase the efficiency of the network during training. A DNN is a neural network composed of several layers of neurons interconnected in a sparse or dense manner. These layers are divided in input, hidden and output, as shown in Fig. 2.1. In a DNN, the number of hidden layers is typically higher than 1.

Each layer consists of neurons that perform mathematical operations on input data, transforming it into a form that is progressively more abstract and meaningful.

The most common DNNs are Multilayer Perceptrons (MLPs, shown in Fig. 2.1, further explained in 3.3.1). Other common types of DNN include convolutional neural networks (CNNs) and recurrent neural networks (RNNs), which are referenced in the citations [9].

Meanwhile, MLPs mainly deal with generic numerical and text data, CNNs are specialised in handling images as input. It is composed of different layers, among which convolutional layers.

They are based on the convolution operation, which consists of applying a kernel with learnable weights to the input image iteratively across all the image, generating output feature maps with enhanced representations. During training, the CNN learns different granular features depending on the depth of the network. Shallow layers learn structural information, while deeper layers usually learn more abstract and detailed features.

RNNs, on the other hand, are designed to handle sequential data such as time series or text. This network relies on feedback, which means it is able

to memorise parts of the input and use them to make predictions.

The main drawback of these architectures is related to their always-growing computational complexity, driven for instance by convolutions, which constitute a significant portion of runtime and energy consumption during training a CNN. To mitigate this, we proceed discussing about some possible implementations, presented in [14].

2.3 Computation Architectures

There are some operations in a multi-stage pipeline that can be parallized with multiple ALU, such as the Multiply Accumulate Operations (MACs). To handle MACs operations in parallel, both GPU and CPU platforms use some temporal architectures such as *SIMD* or *SIMT*, where all the ALUs share the same control and memory. There are several libraries designed for CPUs such as *OpenBLAS* or *Intel MKL* and for GPUs such as *cuBLAS* or *cuDNN* that manage the algebraic operations such as matrix multiplication. From the idea that memory access represents a form of bottleneck and the fact that there are some types of data that can be reused, is possible to optimize ML model training by implementing efficient dataflow strategies.

2.3.1 DNN Dataflow Strategies

A dataflow is a template for understanding and designing a ML sequence of data movement. In a DNN, it can be achieved by strategically managing the flow of data between various components of the training process and exploiting the capabilities of hardware accelerators.

Dataflow strategies encompass a range of techniques aimed at minimising memory access latency, maximizing parallelism and optimizing computational efficiency during model training. The study of dataflows, particularly those adapted for DNNs, reveals strategies that leverage three forms of data reuse: convolutions, filters and images.

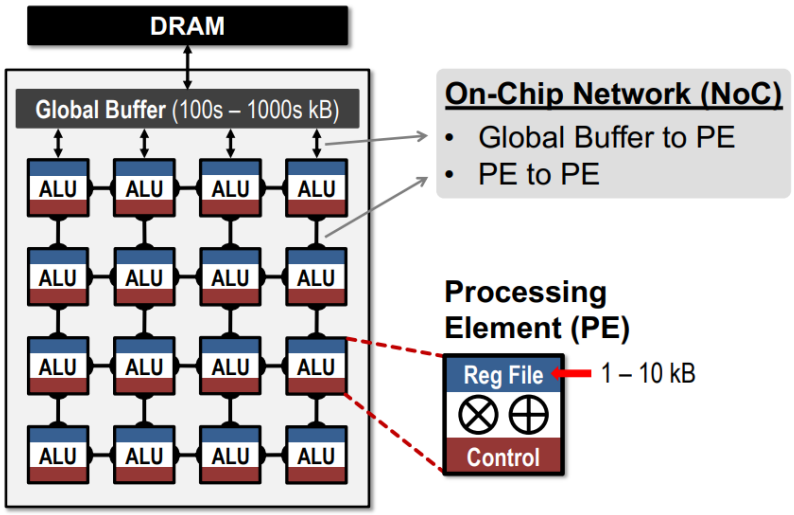

Figure 2.2: Diagram of spatial architecture for DNN [14].

The use of a spatial architecture (see Fig. 2.2) with local memory (register file) at each ALU processing element (PE), typically ranging from 0.5 - 1.0 kB, next to a shared memory (global buffer) ranging from 100 to 500 kB, facilitates this optimization. The motion of data within the PEs is made easier by an on-chip network that reduces access to the global buffer and off-chip memory.

This movement includes input pixels, filter weights and partial sums that are critical to output generation.

There are four main DNN dataflows, based on the mechanism of data handling:

2.3.2 Weight Stationary

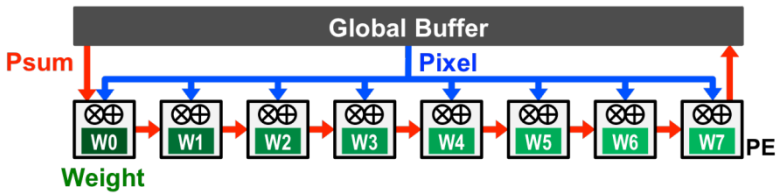

Figure 2.3: Weight stationary diagram [14].

Fig. 2.3 illustrates the diagram of this configuration. The weights are stored into the register file at each PE and remain fixed throughout the computation.

By keeping weights stationary, there is no need to repeatedly fetch them. The inputs and partial sums must move through the spatial array and global buffer. In those models where the number of weights is not too high, this method works very well.

2.3.3 Output Stationary

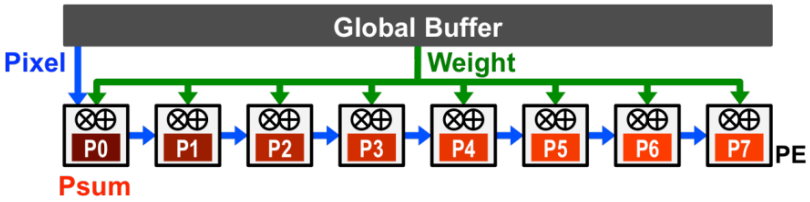

Figure 2.4: Output stationary diagram [14].

In this schema, illustrated in Fig. 2.4, the outputs stored in the register file do not change. The inputs and weights move through the spatial array and global buffer.

2.3.4 No Local Reuse

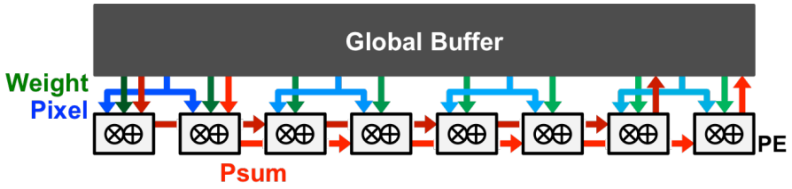

Figure 2.5: No Local Reuse diagram [14].

This strategy, illustrated in Fig. 2.5 focuses on maximizing storage capacity and minimizing off-chip memory bandwidth by sacrificing local storage at the processing elements. Instead of allocating storage within the PE, all available area is allocated to the global buffer, enhancing its capacity. However, this approach leads to increased traffic on the spatial array and global buffer for all data types due to the absence of local storage. While energy efficiency may be improved, area efficiency is compromised.

2.3.5 Row Stationary

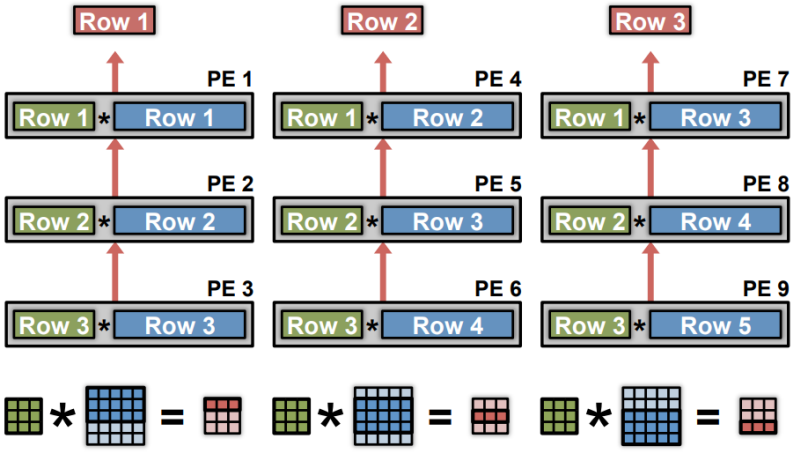

Figure 2.6: Row Stationary diagram [14].

The row stationary approach, represented in Fig. 2.6, aims to maximize reuse of all types of data, including weights, pixels and partial sums. Within this method, a row of the filter convolution remains stationary within each PE, exploiting 1D convolutional reuse. Multiple 1D rows are combined in the spatial array to exhaustively utilize all convolutional reuse, which reduces accesses to the global buffer. Multiple passes through the spatial array ease additional image and filter reuse using the global buffer.

2.3.6 Observations

Comparative analyses, conducted on a spatial array with identical PE count equal to 256, area cost and the CNN network AlexNet, illustrates the energy efficiency of different dataflows. The row stationary approach emerges as 1.4x to 2.5x more energy-efficient compared to other methods for convolutional layers.

These outcomes are attributed to the coherent reduction in energy consumption across all data types, encompassing both on-chip and off-chip energy considerations.

2.4 Hardware Discussion

Once we have introduced the main concepts of machine learning and discussed the dataflow strategies to enhance the efficiency of a DNN, in this

section, we explore the current state of devices that can be utilized for training and testing ML and DL models.

Various processors have been developed, each offering different levels of performance.

For instance, a Graphic Processing Unit (GPU) is the key component when referring to ML and DL fields and related. It allows to compute operations quite faster than CPUs, considering their unmatched parallel processing capabilities and ability to handle massive computational workloads.

They contain thousands of ALU in a single processor. Optimized for parallel computation. This is crucial to execute multiple operations on large datasets. Furthermore, they are well-suited to perform mathematical operations between data structures such as array or matrices. Considering that lots of ML algorithms include operations with matrices such as matrix multiplication or convolutions, they are the first choice in the field.

2.4.1 NVIDIA Solutions

From the advent of ML and DL, NVIDIA always served as the pioneer in the field. It makes available different solutions to work on data science workflows such as GPUs, workstations, data centers or cloud services.

2.4.1.1 NVIDIA GDX

One of the main contribution to the AI field is **NVIDIA GDX** [10].

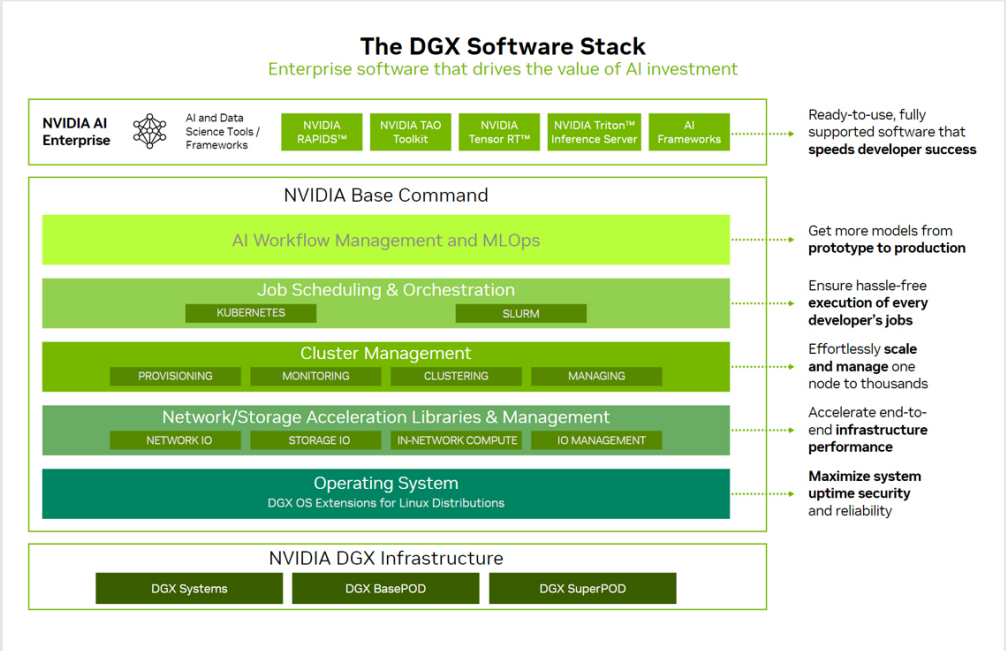

Figure 2.7: NVIDIA DGX software stack, figure taken from [10].

As shown in Fig. 2.7, it is a full-stack technology solution that allows organizations to handle LLM and generative AI models. It serves as a platform that encompasses NVIDIA software, infrastructure and expertise in a unified solution.

The main infrastructures considered encompass:

- **NVIDIA DGX SuperPOD** is an AI data center that guarantees performance for every user's workload.
- **NVIDIA DGX BasePOD** provides the reference architecture for the industries who want to create and scale their AI infrastructure.
- **NVIDIA DGX B200** is a unified platform for training, fine-tuning and inference based on the Blackwell architecture. Its usage is referred to business who is looking for a single platform for their develop-to-deploy pipelines.
- **NVIDIA DGX H200** is a powerhouse leveraged by the NVIDIA H200 Tensor Core GPU, aimed for enterprises.
- **NVIDIA Base Command** this infrastructure includes enterprise-grade orchestration and cluster management, libraries, that accelerate

compute storage and network infrastructure and an operating system optimized for AI workloads.

NVIDIA GDX is available in cloud too, leveraged by the most popular service provider such as Amazon AWS, Google Cloud, Microsoft Azure or Oracle Cloud.

Some years ago, NVIDIA hand-delivered the first GDX system to OpenAI in order to train their models. Considering that ChatGPT 3.0 has been trained with approximatively 10,000 NVIDIA GPUs and the architecture has about 175 billion parameters, the the utility of the DGX system became clear. The DGX system, designed for high-performance computing, enables the efficient training of extensive models by providing the necessary computational power and related scalability.

Regarding the diffusion of GPU cards for individuals and not only for businesses, NVIDIA is the current leader. This is due to the development of several key components such as CUDA [4] and cuDNN [1].

2.4.1.2 CUDA

CUDA is both a parallel computing platform and a programming model, launched in 2006. It actually counts more than 150 libraries based on it. It is considered a landmark by developers and researchers, due to its flexibility and programmability. It further exploits the latest GPU architecture to achieve significant speedups in computational tasks by leveraging thousands of cores to perform parallel processing efficiently.

There are several components related to it, such as:

CUDA Toolkit [11], an environment for creating high-performance GPU accelerated applications. It helps users create, optimize and deploy applications on various platforms such as desktop computers, data centers and cloud services.

Furthermore, it includes libraries, debugging tools, a C/C++ compiler and a runtime library. It also supports the distribution of tasks across multiple NVIDIA GPUs, allowing applications to scale from single GPU setups to large cloud systems with thousands of GPUs.

CUDA-X AI is a suite composed of various tools and libraries that accelerate and optimizes data analytics, HPC workloads, ML and DL tasks. As shown in Fig. 2.8. It provides the underlying acceleration for different frameworks such as Chainer, MXNet, PyTorch and TensorFlow, as well as for

Cloud ML Services such as Microsoft Azure, Databricks and Google Cloud ML.

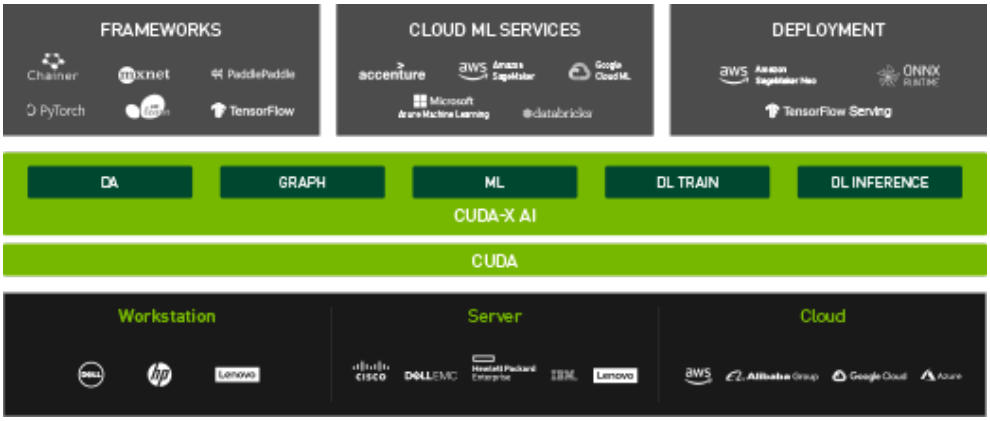

Figure 2.8: Overview of NVIDIA CUDA-X ecosystem, taken from [12].

2.4.1.3 CuDNN

cuDNN stands for CUDA Deep Neural Network library, it is a GPU-accelerated library used to speed up the computationally intensive tasks involved in training and deploying deep learning models.

By exploiting the computational power of NVIDIA GPUs, it provides significant performance gains, efficiency and optimization for deep learning tasks.

2.4.2 Specialized Hardware Components

In the field of ML, hardware sizing methodology is crucial. It strongly depends on the architectures that are planned to build and their underlying algorithms [13].

Generally, both in the ML and DL domains, the CPUs are not the right choice to train and test a model. However, it can be used to perform some pre and post processing operations such as data cleaning or data normalization. The CPU may be the secondary compute engine when the GPU requirements (such as the VRAM) are saturated.

CPUs on the other side, are really flexible. Approximatively any kind of software can be executed, it basically loads values from memory, perform calculations on them and send them back to the memory.

However, togheter with GPUs, in the pipeline aforementioned, the memory access to read and store operands represents a bottleneck.

This led Google to design a Tensor Processing Unit (TPU) [6], a matrix processor that can compute operations at fast speeds. A TPU contains thousands of multiply-accumulators, directly connected to each other in a way to form a large physical matrix. The third version of the Cloud TPU contains two matrices of 128 × 128 ALUs in a single processor.

The TPU host streams data into an infeed queue, that consequently the TPU loads into HBM memory.

After computation, results are moved to the outfeed queue and then read by the host. For performing matrix operations, the TPU loads parameters from the HBM memory into the matrix multiplication unit, processes the data and passes the results through multiply-accumulators without the need for further memory access during the matrix multiplication process. This mechanism guarantees a high-computational throughput for the algorithms involved in a neural network.

2.5 Neural Network Hyperparameters and their Impact on Hardware

The efficient utilization of GPU, CPU and RAM resources during neural network training is crucial for optimizing performance and reducing training time. Several hyperparameters of a neural network and the related training process significantly influence the load on the overall system. Understanding these hyperparameters and setting them accordingly can help in fine-tuning the model for finding an optimal trade-off between performance and system efficiency. Below are some key parameters and their impact on the computational load of the system.

2.5.1 Number of Hidden Blocks

Considering a single hidden block as a linear and an activation layer, the number of hidden blocks determine the capacity of the model.

It is defined as the range of functions that it can approximate. A little capacity can lead to poor performance, defined as underfitting; however, a model with a high capacity can lead to overfitting, a common problem in the ML literature. Underfitting occurs when the model is not able to minimize the error value on the training set. Overfitting occurs when the error in training is much lower than in testing set. The concepts of capacity, underfitting and overfitting are explained in detail in Chapter 5.2 “Capacity, Overfitting and Underfitting” of [5].

The number of hidden blocks has a meaningful impact on the computational load required by the model. Increasing the number of layers results in more matrix multiplications and other operations needed to perform both forward and backpropagation. This lead to the need for more memory and time required to train the network.

2.5.2 Number of Hidden Units

The number of hidden units per block in the network is another hyperparameter that has very similar consequences to the number of hidden blocks. More neurons, means more memory required to store additional parameters. If the number of hidden blocks represents the depth of the network, the number of neurons contained in each layer represents the width of it.

2.5.3 Batch Size

During training phase, many machine learning algorithm decomposes the objective function in a sum over the training examples. In statistical terms, minimizing the loss function means computing an expectation over the entire dataset. This is unrealistic, due to the high quantity of observations usually contained in a dataset. The solution to this problem, is to choose a sample of observations, contained in a subset of it. This subset is commonly named batch size and represents the number of samples involved in the computation of the loss function in each iteration. The algorithms that follow this criterion are named *minibatch* or *minibatch stochastic methods*.

Larger batches provide a more precise estimation of the gradient and exploit the multicore architectures. As batches are processed in parallel, the batch size directly impacts memory requirements, often becoming the limiting factor for many hardware setups. Certain hardware configurations, particularly GPUs, optimize runtime with power of 2 batch sizes ranging typically from 32 to 2048, depending on the architecture and dataset.

Conversely, small batches can induce a regularization effect in training due to the noise they introduce to the learning process, potentially improving generalization. However, training with such small batch sizes can significantly increase the total runtime, requiring more steps to effectively cover the entire training set. Figure 2.9 illustrates a graphical representation of the correlation between mini-batch size and various aspects of model behavior.

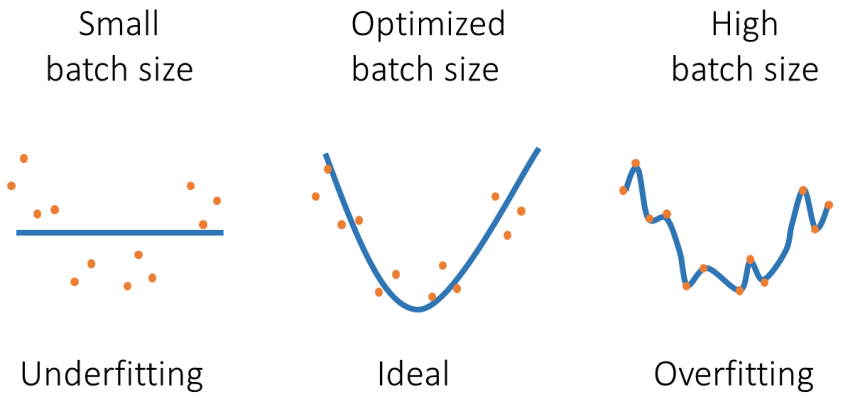

Figure 2.9: Correlation between batch size and the model behavior.

A more detailed description of the influence of batch size on model training and performance related to GPU load is discussed in [2].

2.5.4 Regularization Techniques

The only method of modifying a learning algorithm is to vary the model capacity. For instance, in a regression model this can be done choosing the degree of the polynomial function. This can be very challenging because in this way we are enlarging or lowering the set of possible functions that the model can approximate. A solution to this is the Weight Decay regularization, where we add a term to the loss function in order to penalize weights using a parameter called learning rate. The value of this parameter can strongly affect the model, leading it to underfitting or overfitting as shown in Fig. 2.10.

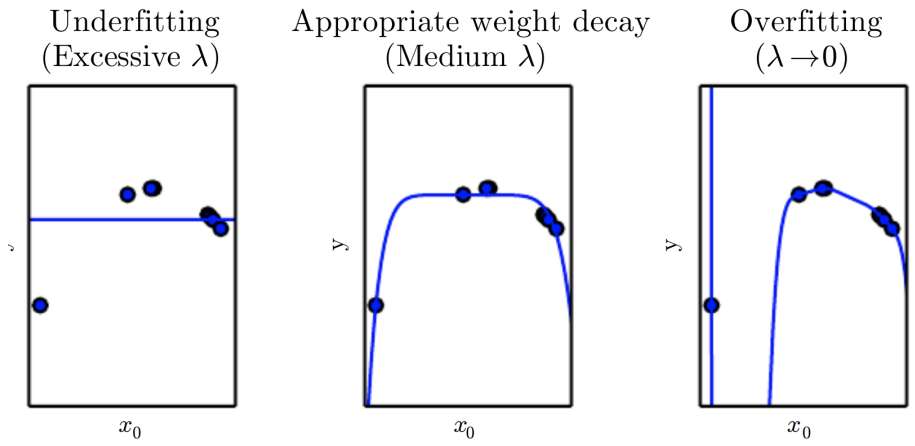

Figure 2.10: Effects of learning rate on the model behavior, taken from subsection 5.2.2 "Regularization" of [5].

Another type of regularization often used in neural networks, is the addition of dropout layers. This method introduces stochasticity by randomly setting to zero neurons with a predefined probability threshold.

2.5.5 Activation Functions

The activation function is a crucial element in every ML model. It defines the shape of the decision boundary that transitions from one prediction value to another. It constitutes a way to determine the output.

In a neural network, an activation layer is a part of the model where each neuron's output is passed through an activation function. They are used to learn non-linear pattern. Its choice affects the performance of a neural network.

We will dive deeper into the effects of various activation functions on the performance of our model in Chapter 4.2. We will compare how different activation functions influence the accuracy of the architecture. We will do it while monitoring the consequences of changing activation functions on the system load to understand their impact on computational resources.

Some instances of different activation functions are then shown in Fig. 2.11.

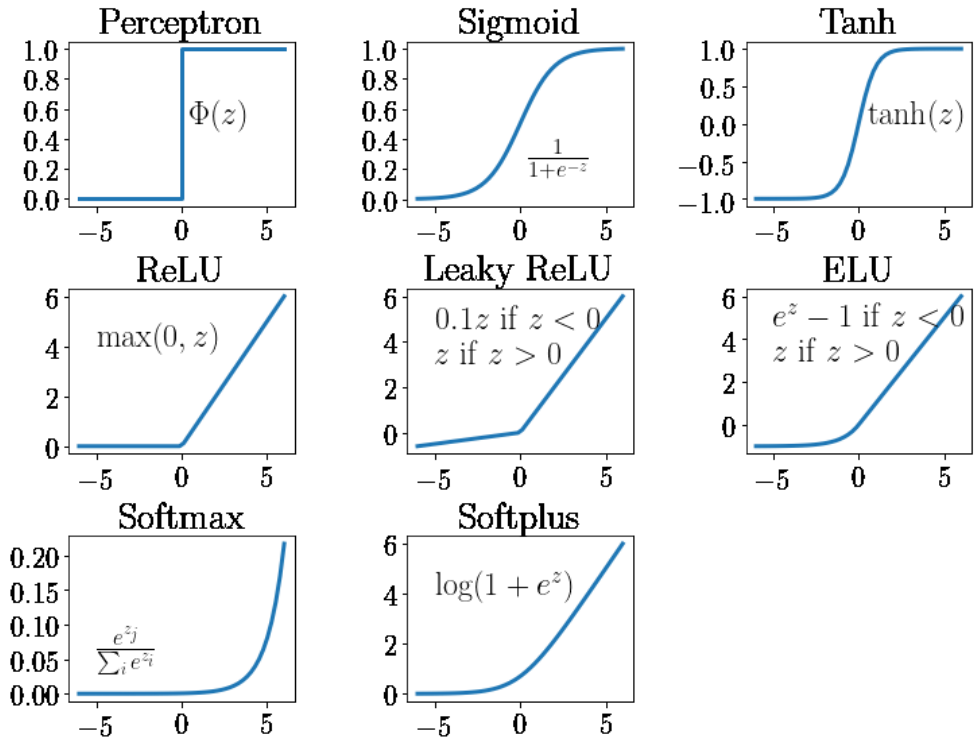

Figure 2.11: Different activation function instances, figure taken from [7].

2.5.6 Optimization Algorithm

In machine learning, optimizers and loss functions are essential components that enhance model performance. A loss function evaluates a model by calculating the difference between expected and actual outputs. Common loss functions include Cross-entropy loss and mean squared error. Optimizers improve the model by adjusting its parameters to minimize the loss function value, thus enhancing prediction accuracy. The role of the optimizer is to determine the best combination of the neural network's weights and biases to achieve accurate predictions. Examples of optimizers are RMSProp, ADAM, SGD, Adagrad and so on..

Further details regarding the mathematical formulation behind each optimizer can be found in [8].

Chapter 3

Prerequisities

In this Chapter we present the programming language and its related libraries adopted in Chapter 4. Then we discuss about the dataset used for the model's training and the optimal configuration of the architecture designed.

3.1 Environment and Python Libraries

Anaconda is a popular open source distribution of Python and R programming languages used for data science. It is composed by a huge collection of pre-built packages and tools, making it suitable for users to set up their development environments in an easily way. One of its key features is the ability to create isolated virtual environments.

Anaconda virtual environments are independent workspaces that allow users to manage different sets of packages and dependencies for their projects. This kind of isolation allows projects to have their specific configurations without affecting the Python installation at the system level.

All the experiments performed have been conducted within a Conda environment.

Python is a high level, object oriented and interpreted programming language. It provides to developers many resources to create complex and high-performance applications. A lot of its libraries have been used in this project, including Scipy, PyTorch, Pandas in particular.

PyTorch is an open source library developed by Facebook AI Research. It offers a wide range of features for building ML models, including DNN, optimization algorithms, loss functions, data visualization tools and much more. PyTorch provides functionality for processing multidimensional arrays, similar to what is provided by Numpy, but with a more comprehensive and powerful library.

Pandas is a data manipulation library that offers a variety of powerful tools to quickly and efficiently analyze, clean and transform data. It provides functionalities such as merging, reshaping, indexing, slicing and grouping, which are essential for data manipulation. Its primary data structure, DataFrame, is a flexible and robust data structure that allows users to work with data in a tabular form, similar to a spreadsheet. In our project, Pandas has been extensively used to work with CSV files and to manipulate data in some algorithms.

Numpy is a numerical computing library for Python. It provides a multi-dimensional array object, togheter with a set of mathematical functions to operate on these arrays. It is widely used in scientific field due to its ability to perform complex mathematical operations efficiently and effectively.

It provides a high-performance array computing functionality and tools for working with them, such as linear algebra or random number generation.

Matplotlib is a Python library used for creating static, animated and interactive visualizations. It offers a range of customization options to make graphs, histograms, scatterplots, piecharts and other visualizations.

GPUtil is a Python library developed to handle GPU management and monitoring within ML and DL environments. With GPUtil, users can easily access and query GPU information, including metrics like GPU utilization, memory usage and temperature. This library enables real time monitoring of GPU components such as its load or the VRAM usage.

psutil is a Python library designed for providing system monitoring and management functionalities, including CPU, memory, disk and network utilization. In this project, psutil has been utilized specifically for monitoring CPU and RAM usage. Throughout psutil, users can access real-time metrics

regarding CPU utilization and memory consumption, allowing for efficient resource management and optimization.

3.2 Dataset

The dataset in which the MLP is being trained is taken from [3].

It is composed by 25 features representing all the joints detectable by MODEL_25B.

1. **Nose**
2. **Left Eye**
3. **Right Eye**
4. **Left Ear**
5. **Right Ear**
6. **Left Shoulder**
7. **Right Shoulder**
8. **Left Elbow**
9. **Right Elbow**
10. **Left Wrist**
11. **Right Wrist**
12. **Left Hip**
13. **Right Hip**
14. **Left Knee**
15. **Right Knee**
16. **Left Ankle**
17. **Right Ankle**
18. **Upper Neck**
19. **Head top**
20. **Left Big Toe**
21. **Left Small Toe**
22. **Left Heel**
23. **Right Big Toe**
24. **Right Small Toe**
25. **Right Heel**

Where for each joint we obtain:

- X coordinate
- Y coordinate
- Confidence Score

Table 3.1: Frame classes occurrences in the keypoints dataset.

Skills	BL	FL	FLAG	IC	MAL	NONE	OAFL	OAHS	PL	VSIT	TOTAL
Frame Occurrences	10236	7561	6964	9791	7309	23150	6389	7950	8694	13092	101136

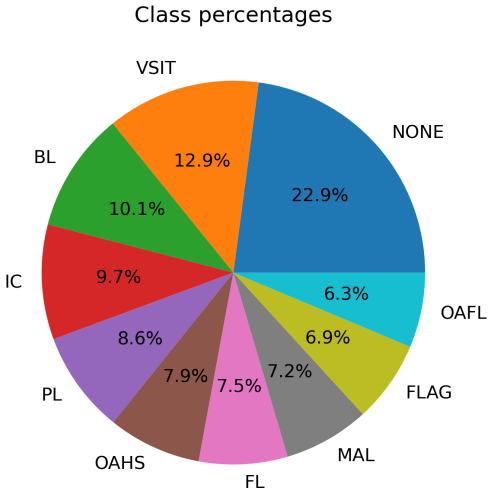

Figure 3.1: Frame classes percentage in the keypoints dataset.

The table in Table 3.1 illustrates the occurrences of classes within the dataset on a per-frame basis. Additionally, Fig. 3.1 presents a pie chart showing the percentage distribution of these occurrences. Notably, the 'NONE' class frames cover nearly a quarter of the entire dataset. This predominance is attributed to the significantly higher number of poses to discriminate as absence of movement compared to those showcasing specific skills.

The dataset was randomly splitted, with 80% of the videos allocated to the training split and the remaining 20% to the test split.

3.3 Model

3.3.1 Multilayer Perceptron

The Multilayer Perceptron (MLP) is an advanced type of artificial neural network that enhances the simple perceptron model by incorporating multiple layers of interconnected neurons in a feedforward network structure. MLPs can solve various problems such as classification, regression and prediction, thanks to their multiple layers of neurons and non-linear activation functions. Unlike simple perceptrons, MLPs can handle complex and non-linear problems.

An MLP typically consists of an input layer, one or more hidden layers and an output layer. The input layer receives data, with each neuron

representing a single input feature. Hidden layers perform most of the computation, with neurons in each layer receiving input from the previous layer and producing an output based on weighted sums of inputs. During training, weights between neurons are adjusted to improve network performance.

Activation functions like Tanh, ReLU and Sigmoid make the model learn complex patterns and relationships in the data. The output layer produces the final network output, such as class predictions in a classification task. Training an MLP involves adjusting weights based on the error between predicted and true outputs. This process continues until the error is minimized.

Optimizers are important in this process. They adjust network parameters during training to reduce errors. They determine how weights are updated based on the calculated errors, improving network performance.

The MLP that has been designed and utilized for this work is hence described:

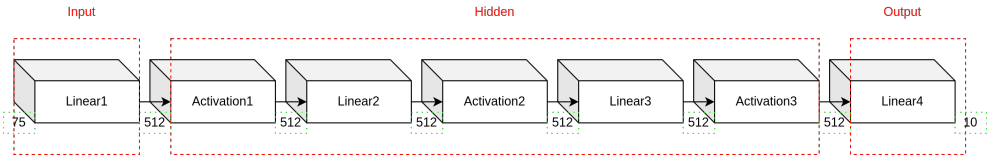

Figure 3.2: Multilayer Perceptron architecture.

As illustrated in Fig. 3.2, the input size is 75, representing the number of values extracted by OpenPose. The hidden layers are structured into three blocks, each comprising an activation layer followed by a linear layer, with 512 neurons in each layer. The network's output size is 10, encompassing the distinct skills along with the 'NONE' frames, allowing for comprehensive prediction capabilities.

For the activation function, the LeakyReLU was specifically chosen due to its ability to alleviate the vanishing gradient problem, facilitating more effective learning. The optimizer selected for training is Adam; it is well-known for its faster convergence related to the SGD algorithm. A learning rate of 0.0001 was set to ensure stable convergence during training. The Cross-entropy loss function was employed to measure the disparity between predicted and actual distributions, optimizing the network's performance in classification tasks.

Throughout the training phase, the model was subjected to intensive optimization across 400 epochs, with each epoch iterating through the entire dataset in batches of size 512. This approach not only ensures comprehensive learning but also mitigates issues like overfitting and poor generalization. All hyperparameters, including learning rate, batch size and some other regard-

ing the network structure, were carefully fine-tuned using cross-validation techniques with 5 folds on the original dataset.

Chapter 4

Experiments and Results

In this chapter, we will perform some analyses on the resources needed to train a Multilayer Perceptron with different configurations, highlighting the influence of each parameter in the various components.

4.1 Analyses Specifications

The experiments were conducted using a desktop PC equipped with the following specifications:

- **CPU**: AMD Ryzen 7 2700 8C/16T 3.2GHz
- **GPU**: ASUS TUF Nvidia GeForce RTX 3060 12GB
- **RAM**: CORSAIR Vengeance RGB Pro Series 32GB (2X 16GB) DDR4 3200MHz CL16

The dataset used and the optimal configuration has been introduced in 3.

The resources to be monitored include:

- **GPU Load**: This measures the percentage of time the GPU is actively processing tasks. It indicates how much of the GPU's capacity is being used.
- **VRAM Usage**: This measures the percentage of the GPU's VRAM that is currently in use. It indicates how much memory is being consumed by the active processes.
- **CPU Load**: This measures the percentage of time the CPU is actively processing tasks over a given period. It indicates how much of the CPU's capacity is being used.

- **RAM Usage**: This measures the percentage of the system's RAM that is currently in use. It indicates how much memory is being consumed by running processes and the operating system.

The values of the resources, retrieved by the libraries presented in 3.1 have been normalized in a range $[0, 1]$, in order to allow the comparison between different experiments.

For instance, if the VRAM usage is 1.00, it refers to 12.288Mib.

The experiments will investigate the resource requirements when changing the following parameters, discussed in 2.4.2.

- Number of Hidden Blocks
- Number of Hidden Units per Layer
- Batch Size
- Activation Function
- Optimizer

All the experiments have been performed in the same system conditions. No other software was opened during the tests and the system was in idle condition.

In addition, in order to define the best configuration, for each hyperparameter, we will keep track of the following parameters:

- Training time
- Accuracy
- F1 score

In each experiment, one parameter will be modified while the others remain constant as the optimal configuration described in 3.3.1 obtained using Cross-validation with 5 folds. The model will be trained for 400 epochs. We will focus on the impact that different architectures have on the overall system; despite this, we will consider the performance of the model too, by making some observations on its performance through accuracy and F1 score. This is needed to discuss about the best trade-off accross all the configurations involved.

The trend of the functions of the values of the resources considered in the following plots has been post-processed with a smoothing function. The smoothing method used is a kernel moving average, which calculates the

mean of the w neighboring values, where $w = \{50, 100\}$, based on the noise level.

Note: the maximum and minimum values discussed may not be present in the plots. This is due to the mean local kernel applied to the functions to reduce noise.

4.2 Technical Analyses

In this section, we will conduct the technical analyses, plotting all the values of each experiment and discussing about the results obtained.

In each plot, the x-axis represents the iterations, where a single iteration is equal to

$$iteration = batchsize/4$$

All the analyses are performed setting seeds that allow their repeatability. All the seeds considered are those responsible for the dataset split, the torch manual seed and the random seed.

4.2.1 Number of Hidden Blocks

The first analysis regards the depth of the network. In the optimal configuration, the number of hidden blocks has been set to 3, so in the following analysis we try to lower and increase it in the range $[2, 5]$.

Fig. 4.1 shows the trend of the GPU Load over iterations.

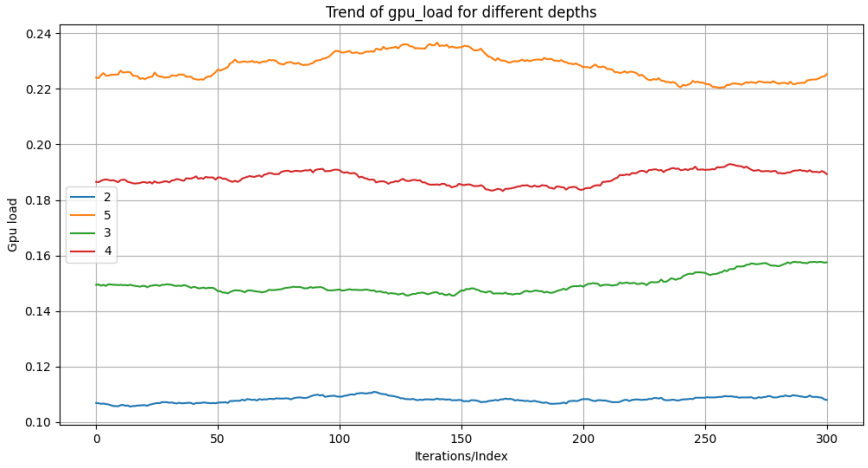

Figure 4.1: Trend of GPU load for different depths.

As can be noted, the load on the GPU is proportional to the number of hidden blocks of the network. This behavior is coherent as we expected. The minimum value is equal to 0.04, reached by the network with only 2 hidden blocks. The maximum value is obtained with 5 hidden blocks and is equal to 0.27. It is worth noting that less than a third of the GPU load is used on each experiment.

The next analysis, represented in Fig. 4.2 represents the normalized value of the VRAM used. For depths 2, 4 and 5, the VRAM usage remains relatively constant throughout the iterations, around 0.045.

In contrast, the VRAM usage for depth 3 starts at approximately 0.0425 and shows a slight increase and some fluctuations after around 150 iterations. Overall, the VRAM usage across different depths shows stability, considering that the range is comprised between 0.0423 and 0.0454. Hence, we can infer that the different number of hidden blocks does not affect the VRAM usage.

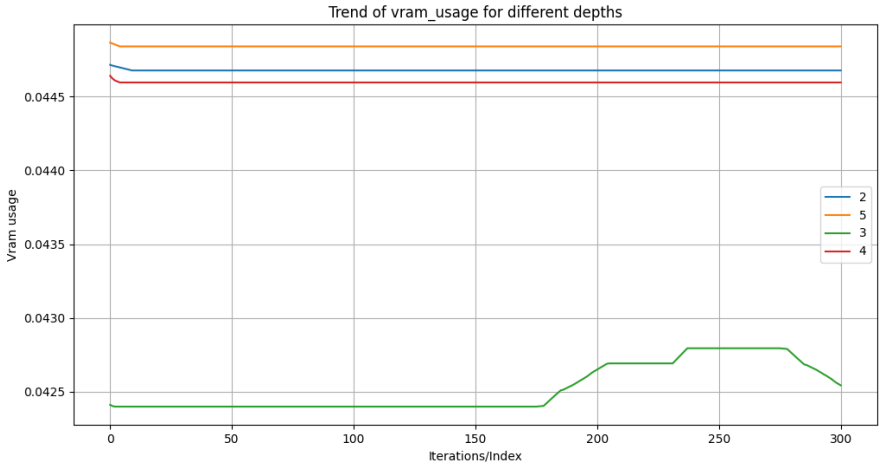

Figure 4.2: Trend of VRAM usage for different depths.

The plot represented in Fig. 4.3 shows the trend of CPU usage over iterations. Surprisingly, the deepest network obtains lower values than the others, reaching a minimum of 0.103. From the plot considered, the CPU usage seems to be inversely proportional to the number of hidden blocks. This may be due to the increased number of layers that shifts the computational load to the GPU, resulting in lower CPU usage.

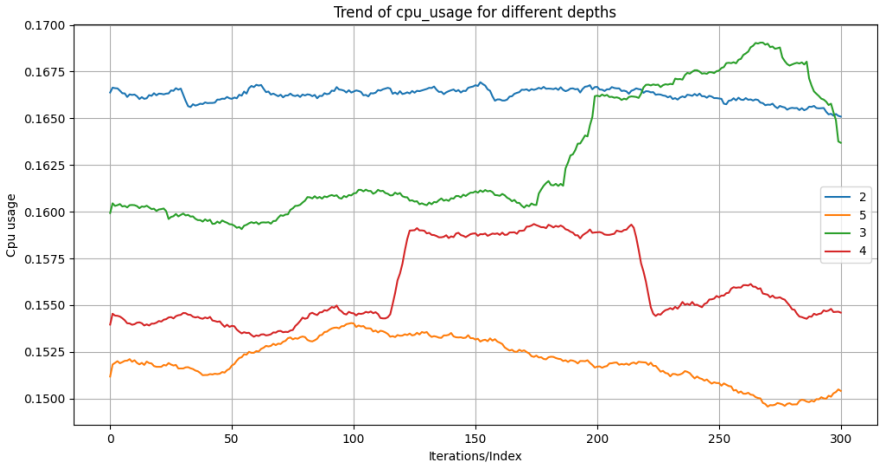

Figure 4.3: Trend of CPU usage for different depths.

The plot illustrated in Fig. 4.4 shows the trend of RAM usage. As can be noted, the values range within a very small interval, suggesting that there is no significant relationship between the number of layers and the RAM usage. Despite this, we can observe that, on average, the network with 3 hidden blocks differs by approximately 0.01 from the others.

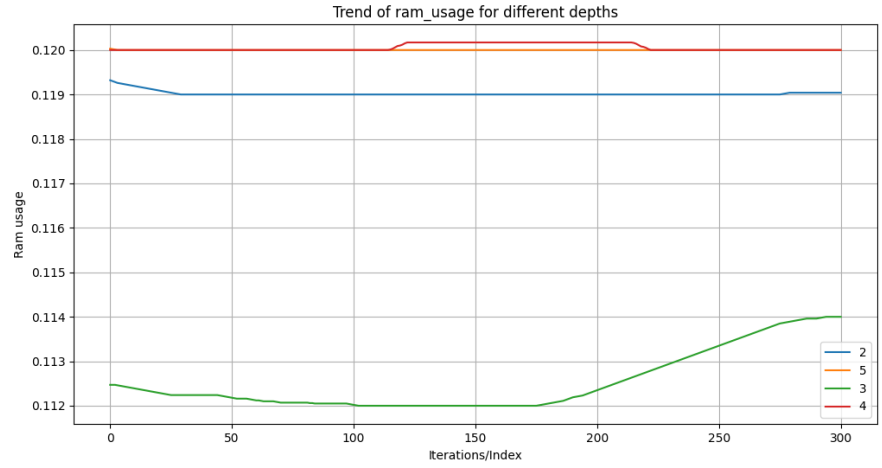

Figure 4.4: Trend of RAM usage for different depths.

The Table 4.1 represents the training time, accuracy and F1 score at various depths. The network with 2 hidden blocks has the lowest training time and the highest accuracy. Increasing the number of hidden blocks there

is an accuracy drop to 0.73 with the 5 hidden blocks one, that achieve the highest training time and the highest F1 score. The network with 3 hidden blocks by the way, seems the most balanced, being the chosen one for the optimal configuration.

	Time(s)	Accuracy	F1 Score
2	**325.95**	**0.807**	0.857
3	346.66	0.791	0.882
4	367.34	0.77	0.839
5	379.31	0.738	**0.887**

Table 4.1: Training time, accuracy and F1 score at different depths.

4.2.2 Number of Hidden Units per Layer

In this subsection, we will analyze the influence of the number of hidden units per layer, starting with the GPU load experiment. The various trends represented in this experiment are illustrated in Fig. 4.5.

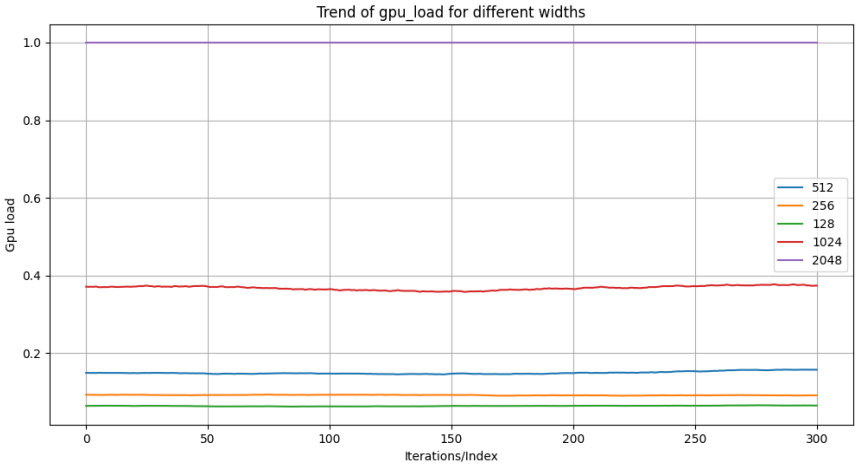

Figure 4.5: Trend of GPU load for different widths.

As can be noted from the plot, there is a clear correlation between the number of neurons and the GPU load. The GPU load for the values {128, 256, 512} ranges from 0.02 to 0.26. The load of the network with 1024 neurons goes around 0.37. Meanwhile, with the configuration of 2048 neurons

per layer, the GPU load reaches 100%, leading to GPU saturation across all iterations.

The impact of the various widths is noticeable from the VRAM usage too, as illustrated in Fig. 4.6.

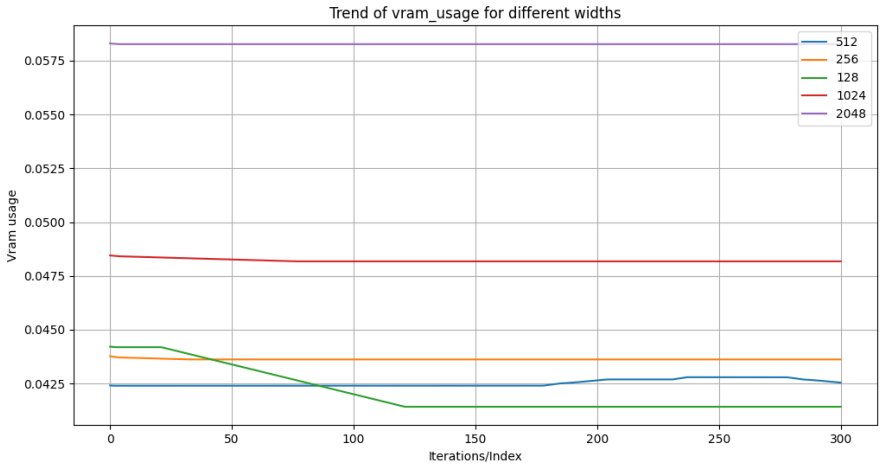

Figure 4.6: Trend of VRAM usage for different widths.

The VRAM usage ranges from 0.040 to 0.044 for the first 3 values, increases to 0.048 for 1024 neurons and further increases to 0.058 for 2048 neurons, with a maximum value of 0.059.

The plot represented in Fig. 4.7, shows the CPU usage varying the number of neurons. As can be noted, the behavior of the functions is similar to the previous analysis regarding the number of hidden blocks. Increasing the width of the network, the computational load of the CPU, is shifted to the GPU.

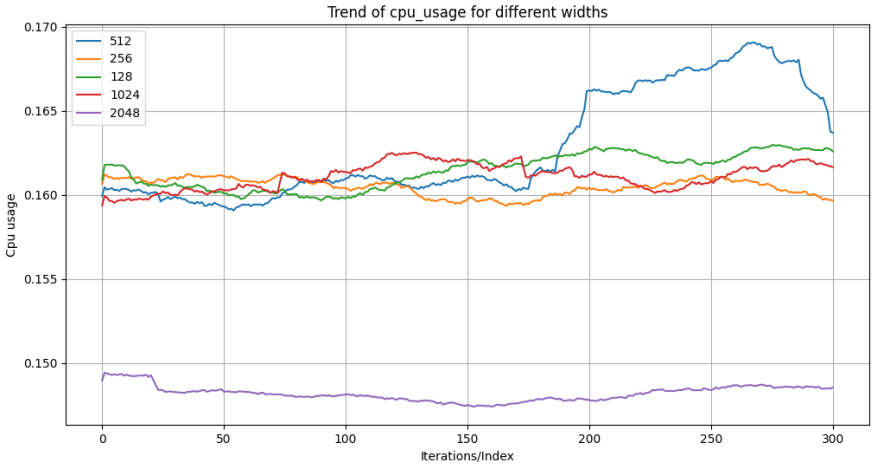

Figure 4.7: Trend of CPU usage for different widths.

The distribution of 2048 neurons is approximately 0.147. All other distribution values range from 0.160 to 0.170, with the highest peak observed in the 512 distribution at 0.268.

The next analysis regards the effect on the RAM usage of the various number of neurons.

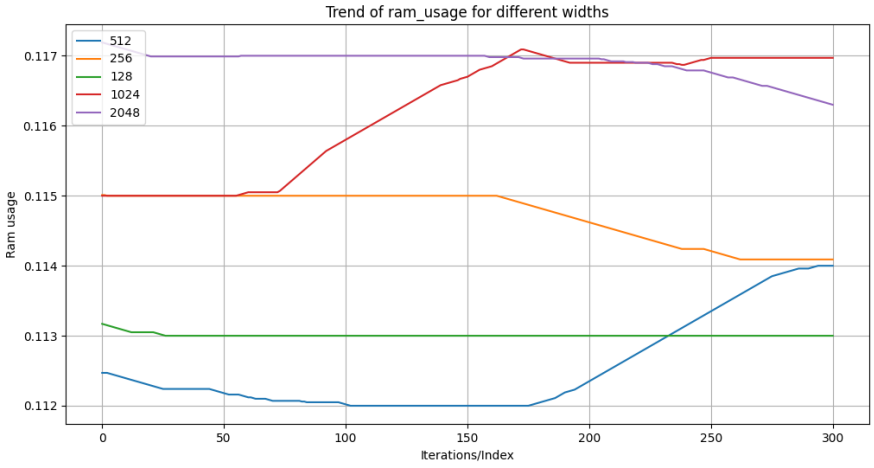

Figure 4.8: Trend of RAM usage for different widths.

As can be noted from Fig. 4.8, despite small oscillations in the configurations considered, the values range from 0.111 to 0.118. This suggests that the width of the network is independent of RAM usage.

	Time(s)	Accuracy	F1 Score
128	**340.54**	0.781	0.840
256	342.35	0.782	**0.904**
512	346.66	0.791	0.882
1024	343.27	0.785	0.881
2048	380.47	**0.80**	0.864

Table 4.2: Training time, accuracy and F1 score at different widths.

Table 4.2 shows the evaluation metrics at different values of neurons per layer. The biggest network width and the smallest one differ for 40 seconds in training time, the accuracy seems stable across all values, the F1 score reach a value equal to 0.904 with 256 neurons per layer. The optimal configuration contains 512 neurons per layer, considered the best trade-off together with the 256 one.

4.2.3 Batch Size

In this subsection we will analyse the influence of different batch sizes. It is worth noting that the iterations on the x-axis are strictly related to the batch sizes, as aforementioned. This leads to different distribution lengths.

We start considering the GPU load first.

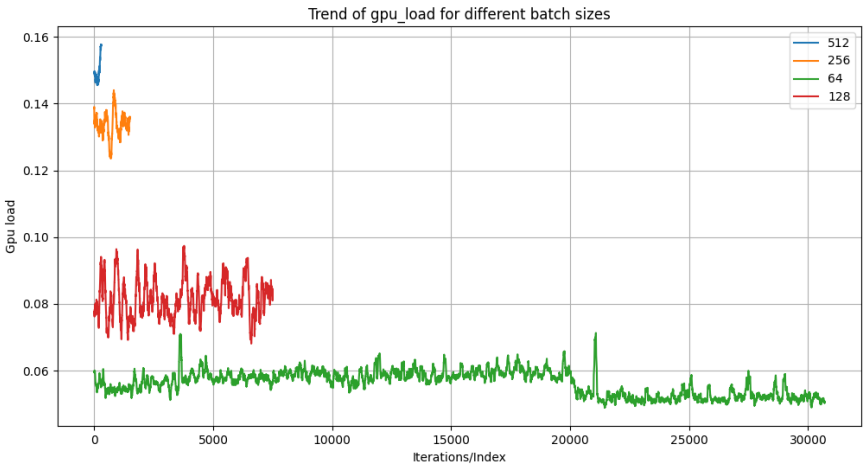

Figure 4.9: Trend of GPU load for different batch sizes.

As shown in Fig. 4.9, all distributions show significant noise (despite the

previous normalization). The GPU load is proportional to the batch size, while the number of iterations is inversely proportional to the batch size, as expected. The mean GPU load for the network with a batch size of 64 is 0.056 over 30,800 iterations, for a batch size of 128 it is 0.08 over 7,600 iterations, for a batch size of 256 it is 0.134 over 1,600 iterations and for a batch size of 512 it is 0.15 over only 400 iterations.

The VRAM usage is then illustrated in Fig. 4.10.

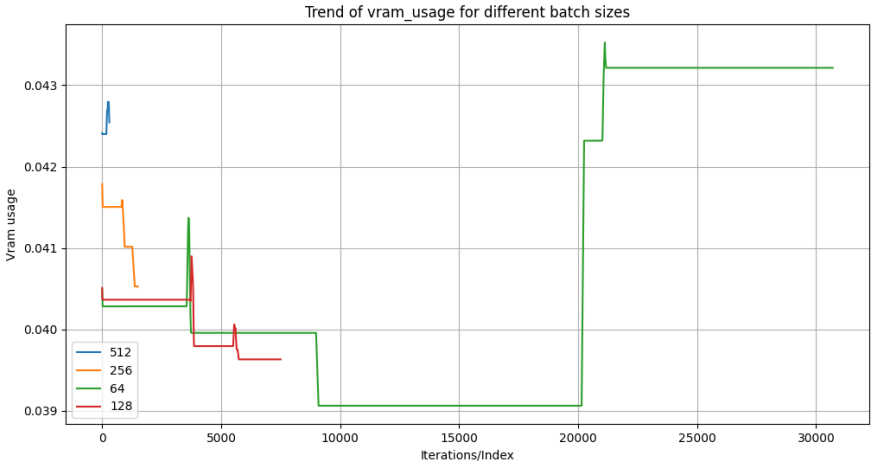

Figure 4.10: Trend of VRAM usage for different batch sizes.

The batch size of 64 has an unusual trend, decreasing to 0.039 before the 10,000th iteration, maintaining a constant trend until the 20,000th iteration and then increasing to a maximum value of 0.044. The batch size of 128 decreases from 0.043 to 0.039 and the batch size of 256 has a similar trend, ranging from 0.043 to 0.040. The batch size of 512 has a mean value of 0.0425. Given this small range, we can conclude that batch size does not have a significant impact on VRAM usage.

The next analysis regards the impact of CPU usage.

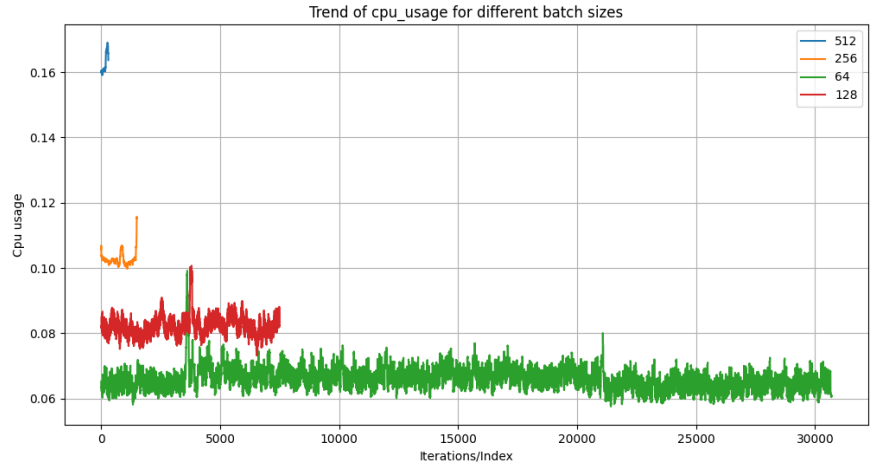

Figure 4.11: Trend of CPU usage for different batch sizes.

Conversely to the previous analysis, in this one, represented in 4.11, an increase in batch size results in higher CPU usage. All the distributions are very noisy, so we report the mean of each. For a batch size of 64, the mean value is 0.065; for 128, the value is 0.081; for 256, the value is 0.103; and for 512, the value is 0.162.

Lastly, Fig. 4.12 shows the impact of batch size on RAM usage. From a visual inspection of the plot, we can infer that, in this case too, RAM usage is not influenced by varying the number of samples processed iteratively.

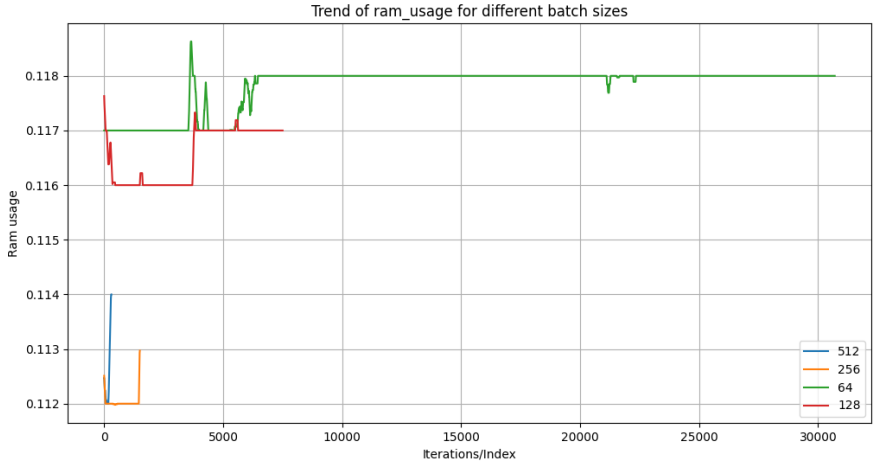

Figure 4.12: Trend of RAM usage for different batch sizes.

	Time(s)	Accuracy	F1 Score
64	3426.29	0.788	0.666
128	1406.50	0.774	**0.942**
256	558.14	0.786	0.889
512	**346.66**	**0.791**	0.882

Table 4.3: Training time, accuracy and F1 score at different batch sizes.

The Table 4.3 represents the different metric values obtained varying the batch size. As expected smaller batch sizes require a higher training time; the network with 512 size, is 9.90 times faster than the 64 one. It also achieves the highest accuracy and a great F1 score value, being the most suitable choice for our configuration.

4.2.4 Activation Function

In this subsection, we focus on the impact of different activation functions on each component.

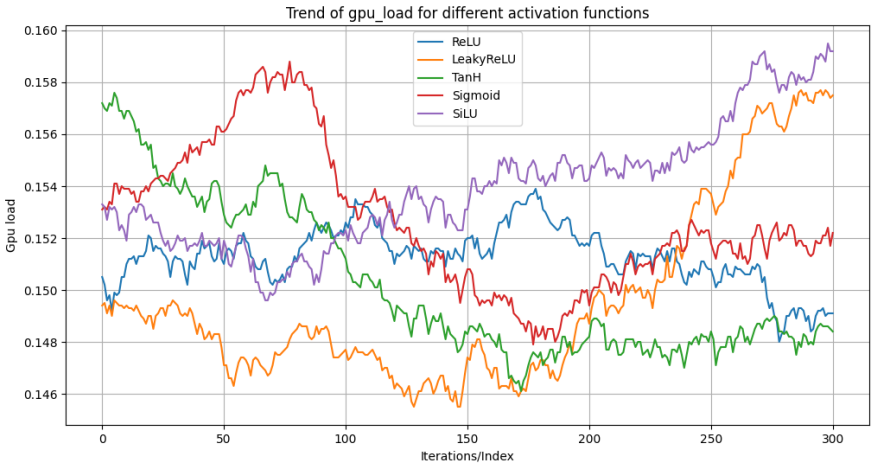

Figure 4.13: Trend of GPU load for different activation functions.

Starting from the GPU load, the plot illustrated in Fig. 4.13 shows various trends, ranging from 0.06 (TanH) to 0.26 (LeakyReLU). We report the mean values for simplicity: ReLU and LeakyReLU have a similar behavior having a mean both equal to 0.150. TanH has mean value of 0.151, Sigmoid has a peak of 0.25 and a mean of 0.152, lastly, SiLU has a mean of 0.154, resulting in the

highest one. This variability is due to the various computational complexities and activation patterns of each function.

The next plot, shown in Fig. 4.14 contains the various trends at varying the activation functions. Considering the small range of values that goes from 0.040 (Sigmoid) to 0.048 (SiLU) we can affirm that the variety of mathematical operations on each activation functions, does not particularly affect the VRAM usage.

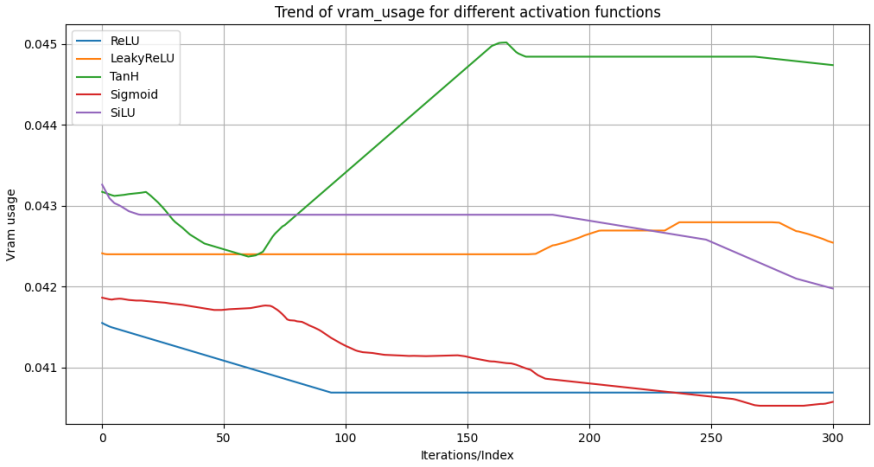

Figure 4.14: Trend of VRAM usage for different activation functions.

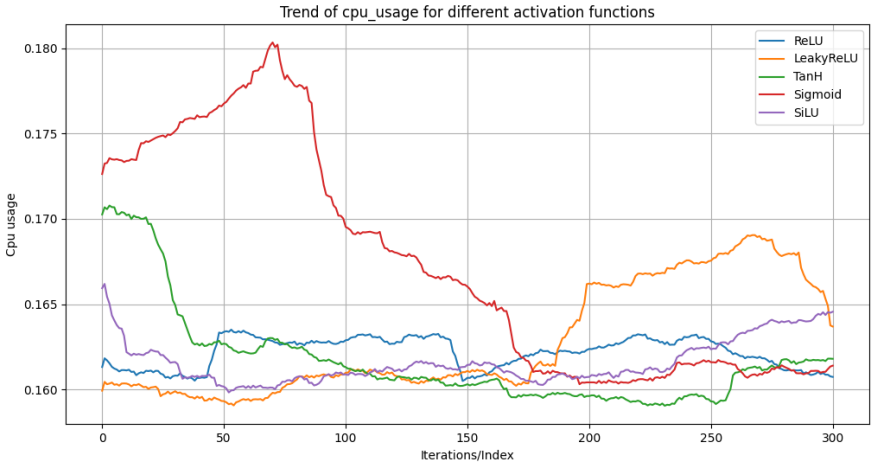

Figure 4.15: Trend of CPU usage for different activation functions.

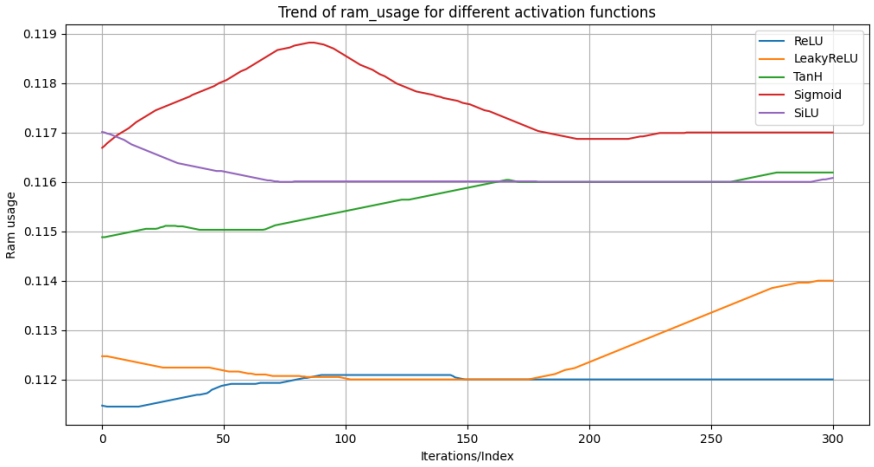

Figure 4.16: Trend of RAM usage for different activation functions.

The considerations made for VRAM usage can also be done to CPU usage, shown in Fig. 4.15 and RAM usage in Fig. 4.16. Regarding the CPU usage plot, all distributions have a mean value ranging around 0.16, with Sigmoid peaking at 0.33. In contrast, the RAM usage plot shows distributions ranging from 0.111 to 0.12.

RAM usage is not significantly influenced by these parameters.

The next analysis regards the time, accuracy and F1 score obtained with the presented activation functions.

	Time(s)	Accuracy	F1 Score
ReLU	346.66	**0.791**	0.882
LeakyReLU	**342.70**	0.782	**0.898**
TanH	347.84	0.781	0.869
Sigmoid	350.37	0.778	0.678
SiLU	346.30	0.780	0.848

Table 4.4: Training time, accuracy and F1 score at different activation functions.

As can be noted from Table 4.4 all training times range in $[342.70, 350.37]$ where the lowest one belongs to LeakyReLU, that obtain the second best accuracy value and the highest F1 score. The highest training time is obtained with Sigmoid that achieves a lower F1 score (0.678). The choice of the activation function does not particularly affect the training time. Furthermore, the behaviors of LeakyReLU, ReLU, TanH and SiLU are quite similar. For the optimal configuration, LeakyReLU is the chosen one.

4.2.5 Optimizer

In this subsection we study the behavior of the network with various optimizers. Starting from Fig. 4.17, it shows the distribution of the GPU loads. Trends seem to be very similar, Adagrad obtains the minimum value, 0.05 while Adam reaches the maximum one (0.26).

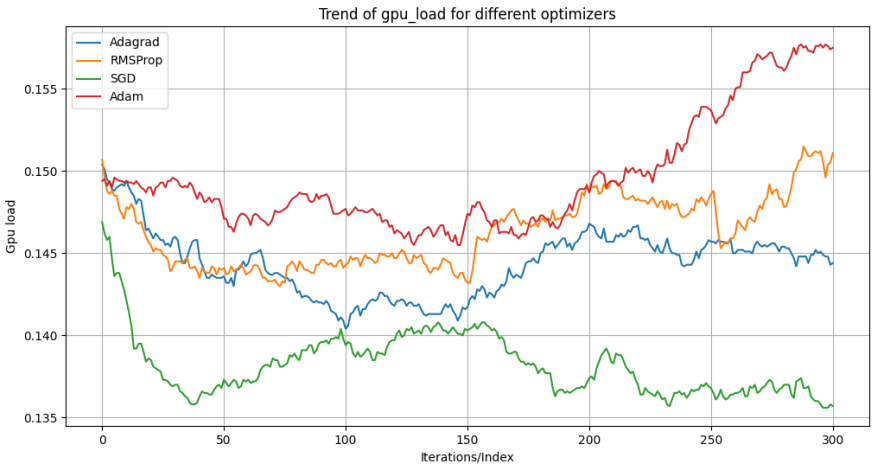

Figure 4.17: Trend of GPU load for different optimizers.

Consequently, in Fig. 4.18, the effect of various optimizers on VRAM usage can be observed. The memory consumption is almost identical across each configuration, with the minimum value achieved by SGD at 0.410 and the maximum reached by RMSProp at 0.451.

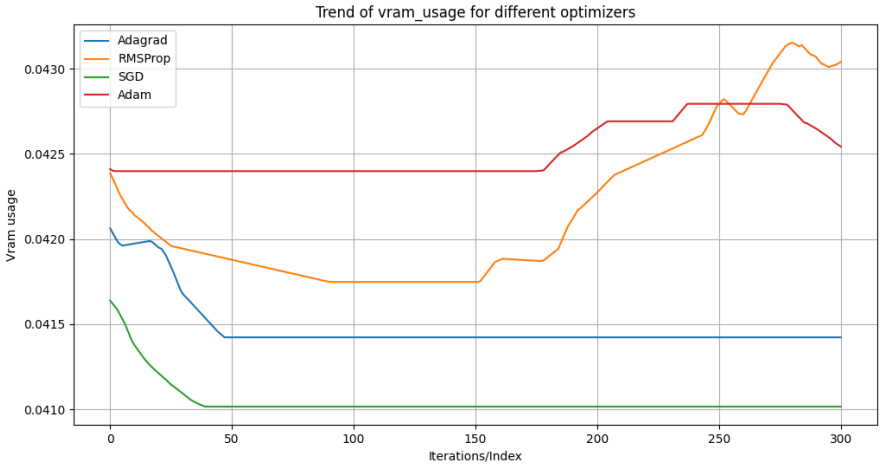

Figure 4.18: Trend of VRAM usage for different optimizers.

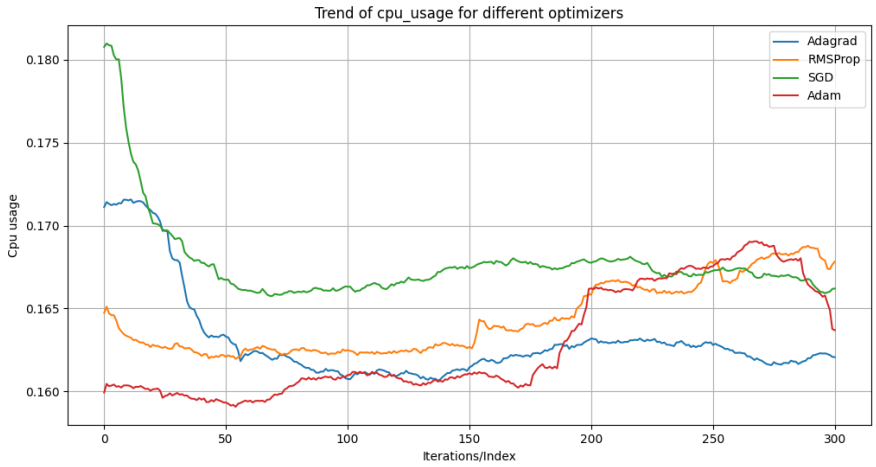

Figure 4.19: Trend of CPU usage for different optimizers.

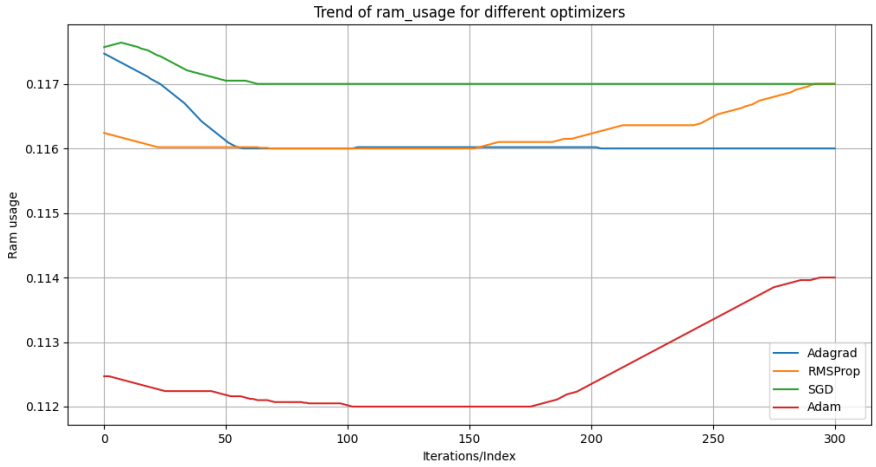

Figure 4.20: Trend of RAM usage for different optimizers.

Lastly, in this analysis too, the CPU usage (see Fig. 4.19) and the RAM usage (see Fig. 4.20) are not affected by the current hyperparameter. The maximum value in CPU usage is reached by SGD with 0.336, it has the highest mean value equal to 0.17. All other distributions have a mean value around 0.16. The RAM usage ranges from 0.104 with Adam to 0.286 with Adagrad.

The last analysis regards the optimizer-related metrics.

	Time(s)	Accuracy	F1 Score
Adagrad	342.40	0.780	**0.928**
RMSProp	340.90	0.473	0.651
SGD	**328.81**	0.743	0.913
Adam	346.66	**0.791**	0.882

Table 4.5: Training time, accuracy and F1 score at different optimizers.

As shown in Fig. 4.5, Adagrad achieves the highest F1 score equal to 0.928, a great level of accuracy and a training time similar to RMSProp and Adam. SGD has the lowest training time equal to 328.82, 5.14% faster than Adam. However, it obtains a lower accuracy. The worst results are obtained by RMSProp that has an accuracy equal to 0.473.

Adam is chosen for its superior accuracy and competitive F1 score, this choice was supported by some visual inspections on the output images too.

4.3 Insights from Experiments

All the plot and the table represented, have been adequately discussed. From the experiments made, there are different considerations that can be done.

As seen, the GPU load is strongly impacted from the number of layers of the network and the number of neurons for each layer. This is motivated by the increase of the mathematical operations during the feed-forward and back propagation phases. The size of the batch size represents another component that is proportional to the GPU load. Increasing the number of samples processed in each iteration, it translates in a higher load required. However, the activation functions and the optimizers do not significantly affect the GPU load.

The VRAM usage is not particularly influenced by the various depths conversely from the increase of neurons per layer. This is due to the exponentially increases of the number of weights and the size of activation matrices. Meanwhile, adding more layers increases VRAM usage gradually as it only adds a fixed number of weights per layer. The VRAM usage does not depend from the size of the batches, the activation functions and the optimizers considered.

From some analyses, the CPU usage seems to decrease with the increasing computation required. Shifting the load to the GPU load instead. Despite this observation, the different batch sizes proved the opposite. The highest batch size considered, has the most meaningful impact on the CPU usage. For both activation function and optimizer analyses, we cannot infer anything regarding the CPU usage impact, considering they have similar behaviors.

From all the experiments, we can affirm that the RAM usage is independent of the hyperparameters considered.

Regarding the training time, as expected the shallowest network obtains lower training time, as the one with less neurons. This is due to the fewer number of operations performed. The batch size has a strong impact on the training time, we range from 400 to more than 30,000 iterations due to the higher number of batches considered. Both activation functions and optimizers do not significantly affect the training time.

Lastly, the accuracy and F1 score provided two important performance metrics to evaluate the goodness of the various configurations. The network with 3 hidden blocks is the most balanced one, 512 neurons have been chosen for the good trade-off between performances and network size. The optimal size of the batches is set to 512, having the highest accuracy and a great F1 score value. The best activation functions tested are ReLU and LeakyReLU; we chose the last one considering its slightly slope over negative x-axis values to avoid the dead neurons problem. Lastly, Adam is the optimizer that

achieves the highest accuracy and a great value of F1 score, hence the adopted one.

Chapter 5

Conclusions

In this work we presented an overview regarding the relation between ML/DL and the underlying hardware.

We began talking about the applications of AI technologies in various fields, proceeding about providing some description of the backbone of a neural network and the main derivative architectures.

Then, we talked about some strategies adoptable in the flow of data within a neural network. For each proposal, some observations have been made about their pros and cons.

Subsequently, a hardware discussion has been made, talking about the most suitable components to perform the training of a model. We compared GPUs with CPUs justifying their utilization. Furthermore, we introduced some of the NVIDIA libraries and both local and cloud infrastructures, describing them accordingly.

Following this, we examinated the main hyperparameters of a neural network and conducted some experiments to understand their influence on a multilayer perceptron specifically designed to perform multiclass classification on a Calisthenics skills video dataset, presented and briefly analyzed.

All the experiments have been extensively discussed, motivating the choice of each hyperparameter in the optimal configuration. From the outcomes obtained, GPU load, VRAM usage and CPU usage are the main influenced components, exhibiting various trends at different values. The RAM usage does not dependent from the network configuration changes.

The dataset and the various codes written for both model and analyses can be found in: `https://github.com/antof27/An-Outlook-on-Machine-Learning-Resource-Requirements`.

Through this work, we emphasize the need for ML and DL architectures to decrease in size and number of parameters, considering their impact on current hardware. Recent studies are exploring the concept of Liquid Networks, which aim to enhance the computational capacity of each neurons, therefore reducing the overall network size.

In conclusion, nowadays, the advancement of the AI in terms of computational power and libraries is assigned to NVIDIA which cover the main actor, providing several solutions for both individuals and companies.

The current developments in these fields promise to make AI technologies more efficient and accessible.

Bibliography

[1] 1kg. https://medium.com/@1kg/what-is-cudnn-3f67d535a580.

[2] Adil Lheureux. https://blog.paperspace.com/how-to-maximize-gpu-utilization-by-finding-the-right-batch-size/.

[3] Antonio Finocchiaro., Giovanni Farinella., and Antonino Furnari. "Calisthenics Skills Temporal Video Segmentation". In: *Proceedings of the 19th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 2: VISAPP*. INSTICC. SciTePress, 2024, pp. 182–190. ISBN: 978-989-758-679-8. DOI: 10.5220/0012400600003660.

[4] Fred Oh. https://blogs.nvidia.com/blog/what-is-cuda-2/.

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

[6] Google. https://cloud.google.com/tpu/docs/intro-to-tpu.

[7] N. Johnson et al. *Machine Learning for Materials Developments in Metals Additive Manufacturing*. May 2020.

[8] Maciej Balawejder. https://medium.com/nerd-for-tech/optimizers-in-machine-learning-f1a9c549f8b4.

[9] Mayank Mishra. https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939.

[10] NVIDIA. https://resources.nvidia.com/en-us-dgx-platform/nvidia-dgx-platform-solution-overview-web-us?ncid=no-ncid.

[11] NVIDIA. https://docs.nvidia.com/cuda/.

[12] NVIDIA. https://www.nvidia.com/it-it/technologies/cuda-x/.

[13] Puget Systems. https://www.pugetsystems.com/solutions/ai-and-hpc-workstations/machine-learning-ai/hardware-recommendations/.

[14] Vivienne Sze et al. "Hardware for machine learning: Challenges and opportunities". In: Apr. 2018, pp. 1–8. DOI: 10.1109/CICC.2018.8357072.