Lehrstuhl für Data Science

Effects of CPU and GPU architectures and driver versions on the accuracy of Neural Networks

Masterarbeit von

Shashank Pandey

1. Prüfer
Prof. Dr. Michael Granitzer

2. Prüfer
Prof. Dr. Harald Kosch

Betreuer
Christofer Fellicious

September 6, 2021

Contents

Abstract

In recent times, more and more methods are claiming a position in leader board score with tiny improvement over the previous method. We wanted to investigate if the improvement in method claims by the authors is based on the algorithm changes or because of the serendipitous results achieved by the hardware and software.

In this thesis, we examine the effect of different CPUs, GPUs and different driver versions on the accuracy of the neural network. Neural networks are very complex, and therefore we performed the experiment in two parts, firstly, logistic map followed by image classification model based on VGG-11. The logistic map is a relatively simple mathematical equation that is deterministic and can induce randomness over multiple steps. Precisely the property of the logistic map equation to introduce randomness into the system is makes it an ideal baseline for our experiment. The image classification model is a relatively more complex problem than a logistic map with millions of connections, and over 100s of epoch give more opportunity for error to accumulate and showcase in the accuracy matrix of the model. We performed the experiment with both *float* and *double* image classification model in GPUs to comapre the results.

Acknowledgments

This thesis has been made possible because of many people. First of all, I want to thank my thesis advisor Christofer Felicious for guiding me and helping me in the research area. His help enabled me to better shape my research and develop innovative solutions for the challenges faced during the thesis work.

Next, I would further like to express my gratitude to Prof. Dr. Michael Granitzer and Prof. Dr. Harald Kosch for allowing me to work on this exciting research area. Prof. Dr. Michael Granitzer's inputs on my initial proposal and during my presentation were precious. They helped me form a valuable baseline for the research experiment and gave a solid direction to my thesis.

I would also like to thank the whole Data Science Chair team at the University of Passau, especially the infrastructure team, for providing the support and service necessary to conduct this experiment.

Finally, I would like to thank my parents, who trusted in me and invested their time, effort and resources in helping me achieve my dream. The emotional support of family and friends, especially during the isolation due to the Covid-19 pandemic, was incredible.

List of Figures

List of Tables

1 Introduction

In recent times, the progress in the machine learning domain has increased exponentially. One of the reasons behind this is its capability to solve complex problems, such as Autonomous Driving [20d] and Sign-to-speech translation[Zho+20]. Machine learning algorithms are ubiquitous in the modern world; the use of machine learning algorithms can be found from capturing a photograph in smartphones to cashier-less checkout systems like Amazon-Go where machine learning is used to generate the payment receipt automatically [Gro]. As the popularity of machine learning increases, this has lead to an increase in competition. For example, in the case of CIFAR-10[1] dataset Benchmark[20c], there are eight different methods between the accuracy range of 93%-94% Table 3.3 shows the methods in detail. The different methods compete fiercely for a position on the leaderboard, but the margins between the two consecutive methods are tiny. This lead to a worrying question of whether these methods are really better or the boost in performance is because of the hyperparameter optimisation and just plain luck when running the experiment.

Another worrying issue is that more than half of the methods in the machine learning domain are also non-reproducible. According to Consensus study[19] non-reproducibility occur mainly due to insufficient details such as data, code, models and the computation analysis needed. In the Consensus study[19], reproducibility is defined as "Obtaining consistent computational results using the same input data, computational steps, methods, code, and conditions of analysis." Since science progresses through peer review and experimental reproducibility, it is vital to make models reproducible. Reproducibility also helps to validate the experimental results and claims. Adapting an algorithm that is non-reproducible is risky because the output of the algorithm cannot be predicted. As a result, the model output cannot be trusted or verified. Non-reproducibility also hinders further research on the subject. For example, recently, Google announced a new AI

[1] `https://www.cs.toronto.edu/~kriz/cifar.html`

model[Wig20] to detect breast cancer in patients based on mammogram X-rays. Google claims that their model is better than a human radiologist. However, the unavailability of source code, methods used, and data have raised questions. Scientists are not ready to accept the model for real-world application despite the promising results because they cannot reproduce the model, and hence model validity is questionable[Wig20].

There are many reasons why a specific experiment might not be reproducible. Even when the code is available, the performance might vary based on the hardware and software used.

As the model runs on hardware, the variance could come from the hardware and the drivers used to run that hardware. Different hardware has different heuristics to perform the mathematical operations and thus could bring the variance in the model's performance. The change in the driver version of the GPUs (Graphics Processing Units) could have an impact on the performance of the model [16b]. The driver versions are updated frequently, tweaking the heuristic and therefore could impact on the performance of the model[16b]. As an example, the drivers of Nvidia Titan Series for Windows 10 64-bit are updated at least once a month and in some case even three times in a month [NVIb]. This source should also be examined for possible cause of variation in the accuracy of the model.

This calls for a question of whether the new methods are better than the previous methods or if the improvement is by virtue of the difference in hardware or software used to execute the method. Like Musgrave et al.[MBL20] pointed out in the recent report that even the state-of-the-art methods are not significantly improved, than traditional metric learning methods, without making much headway.

A hindrance to reproducibility is the unavailability of code. As mentioned by Huston[**b9**] only 6% of the researchers submit their code, as observed by the survey of 400 algorithms. He also suggested that the reason for the reluctance to submit the code is that the code is a work in progress or the code is owned by the company, amongst many other reasons.

In 2016, *Nature*[Bak16] reported that more than 70% of researchers have tried and failed to reproduce the result of other scientists. And more than 50% of researchers have failed to reproduce their own experiments. Machine Learning is struggling with the reproducibility crisis for over a decade now, like psychology, medicine and other scientific fields[**b9**]. More than 50% of researchers agree there is a significant reproducibility crisis in research, as shown in Figure 1.1.

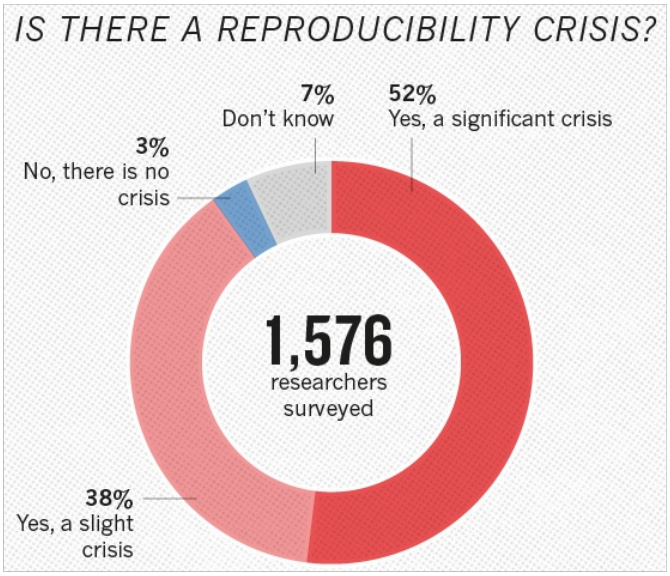


Figure 1.1: Survey results of reproducibility crisis in research[Bak16]

Non-reproducibility not only makes the experiment unavailable for scientific verification [Wig20] but also has monetary loss associated with it.

The cost associated with the non-reproducible research is high, as the labs have to make a significant investment of time and effort to test the algorithm in the environmental conditions of the lab. Currently, there is a push for larger and more complex models. For example, the training cost of GPT-3 could go as high as $4.3 million[20g], and this is unaffordable for many research centres and universities.

It could be even worse if a claim made by non-reproducible research turns out to be false, thereby risking the validation of the subsequent research too.

Therefore, now it is important more than ever to address this issue of non-reproducibility.

This particular topic motivates me because of its unique approach. The topic is not taking the state-of-the-art for granted, but moving in a different direction to see if the hype around them is actual or is it because of the serendipitous results achieved by the hardware and software.

1.1 Motivation

The topic for the master thesis is "The Effects of CPU and GPU architectures and driver versions on the accuracy of the Neural Networks". The topic is wide and can be extended from hardware architecture to different software versions of GPU. The scope of this master thesis is to study and report the effect of hardware and software changes on the performance of Convolutional Neural Network (CNN). The thesis goal is divided into three sections initially; we examine, *How much variation in the accuracy does the change in hardware cause*. Followed by, *How much variation does change in driver version cause* and finally, *Methods to achieve reproducibility*.

1.1.1 How much variation in the accuracy does the change in hardware cause?

The model performance varies based on the hardware used. The performance of the model is recorded when executed on different CPUs (Central Processing Units) and on the different GPUs.

In the case of CPU although the operations are performed sequentially, the results of the experiment will showcase how much variation in the accuracy is introduced by the change in CPU. In GPU the operations are performed in parallel because of the availability of numerous cores as shown in Figure 1.2 and as a result, the non-deterministic sequence of operation is observed[21c]. The reason for non-deterministic behaviour is the sequence of operation cannot always be determined and when it comes to GPU, the order of operation matters as discussed in detail later in section 2.3.

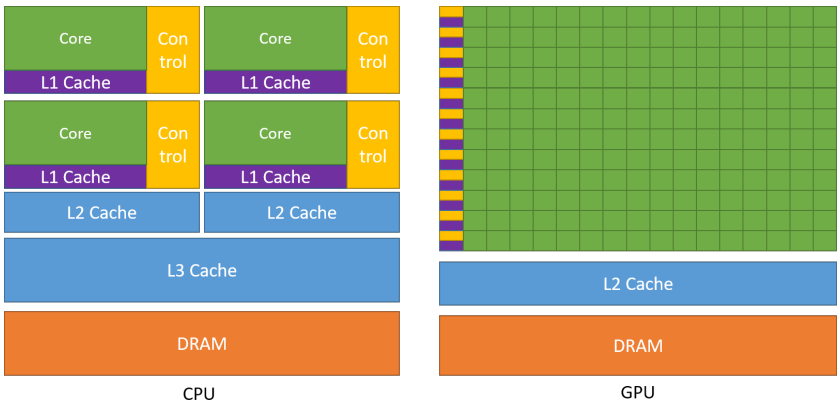


Figure 1.2: Multiple cores of CPU vs hundreds of cores of GPU [21c]

It would be interesting to know how much variation in the accuracy is caused by different CPUs and GPUs because based on the experiment, we could scientifically prove the effect of different hardware on the performance of the model.

1.1.2 How much variation does the change in software driver versions cause in the accuracy of the mode?

It is reported[16b] that the changes in driver versions of GPUs affect the performance of the model. This is because different driver versions use different heuristics to merge FMUL + FADD to FMA and hence inducing non-determinism into the model. FMUL stands for Floating-point multiplication and FADD stands for Floating-point addition. FMA stands for Fused multiply-add. FMA is performed by Nvidia GPUs to increase the accuracy of the operation by fusing the two operations (multiply and addition) into one operation. FMA calculates the expression using one rounding off step as opposed to two rounding off steps performed by FMUL and FADD individually; further details on FMA and FADD are discussed in the upcoming section 2.3.

Reduction of one rounding-off step makes a difference because even the small approximation error could quickly snowball into large approximation errors when the operations are performed thousands of times and across multiple epochs. For example, in the case of the U.S. Patriot missile, the small error of 0.000000095 was introduced while saving the *time since boot* of the system.

This small error accumulated to 0.3433 seconds over the period of 100 hours and lead the missile to miss the target by 687 meters, thereby falling to intercept the incoming missile. This 0.3433 seconds error resulted in the shift of the Range Gate Area by 687 meters, enough to miss the incoming missile or Scud[20a]. *Range Gate* is an electronic device in the radar system which set an area in the airspace where it should next look for the Scud. Range Gate calculates this area once the Patriot's radar has detected and classified an incoming airborne object as a Scud[20a]. Therefore, we need to consider such rounding-off error during our model training because training lasts 100's of epochs.

1.1.3 Methods to achieve reproducibility

Reproducibility ensures that the results achieved are not by serendipitous chance but with the help of a pre-determined heuristic. The idea of reproducibility is to ensure transparency and gives confidence in understanding exactly how the accuracy is achieved[02]. Reproducibility also increases openness and makes further study much more feasible.

The motivation behind the topic of the thesis is three-fold:-

1. Study the effects of different CPU & GPU on the performance of the model.
2. Study the effects of different GPU driver versions on the performance of the model.
3. Try to make the model reproducible by including the often neglected details of the experiment, such as the driver version of the GPUs.

In this report, following possible sources of variance in accuracy are considered for examination:-

- Different CPUs
- Different GPUs
- Different GPU driver versions

Different CPUs will be used to execute the same model, and performance of the model will be recorded. Similarly, different GPUs with different driver versions will be used to run the experiment and the performance of the model will be recorded to calculate any variance observed while testing. The model executed on all the different hardware will be the same, with fixed weight initialisation using a fixed seed. The random weight

initialisation seed will be used because the weights should always be initialised with the same random values throughout the experiment. This is to ensure the model itself does not contribute to variance. If there is a significant variance in results, it proves the hypothesis that the different CPU & GPU and different driver version affects the results of the experiment.

Performing this experimental study will not only help in determining the variance caused by different hardware and software, but will also demonstrate that often neglected details about the model like CPU and GPU details are important for model reproducibility. As these details make a significant impact on the performance of the model and should be part of the formal report as suggested by Prof. Joelle Pineau in her reproducibility Checklist[20b] discussed more in detail in chapter 2.

1.2 Research Questions

The primary goal of this thesis is to investigate the effect of different CPUs, GPUs and different GPU driver versions on the image classification model. In order to check the variance across the hardware and software, Accuracy is used as an evaluation matrix. Accuracy of a model is a standard matrix on leaderboards like Benchmark.io[20c]. Hence, for this experiment, an accuracy matrix is proposed for evaluation.

The following research questions will be answered in this report:-

1. How much does the accuracy vary with the difference in hardware (CPU & GPU)?

2. How much does the accuracy vary with the difference in GPU driver version?

3. How extending experiment tracking software like PyPads to log additional details like GPU driver versions could be useful in reproducibility of the experiment.

We will answer these questions with a model having three different optimisers, namely AdaDelta, SGD and NAG. For reproducibility of the experiment, constant seed values are used so that the experiment starts with the same random values. A single run is recorded with five different seed values to eliminate the advantage by any particular seed value.

1.3 Proposed approach

In order to investigate the effect of CPU and GPU architectures and different driver versions on the accuracy of the neural network, we propose the following three experiments to be conducted:

1. Logistic Map
2. Image classification model
3. Extend Pypads functionality to log GPU details

1.3.1 Logistic Map

In this experiment, we run the logistic map Equation 3.1 on different CPUs, GPUs and on different GPU driver versions and check for variance in the results over 100s of epochs. The logistic map by the Equation 3.1 [16a]

$$X_{n+1} = rX_n(1 - X_n) \tag{1.1}$$

This experiment will serve as a baseline for the item 1st research question when executed on different CPUs and GPUs.

This experiment will also serve as a baseline for the item 2nd research question when executed on GPUs with different driver versions.

Therefore, this experiments will help to answer item 1st and item 2nd research question.

1.3.2 Image classification

In this experiment, we will run the image classification model developed using Pytorch. Code flow from Pytorch to Hardware is shown in the Figure 1.3.

At first, changes in the bottom layers will be made, i.e., different CPUs and different GPUs will be used. In the second iteration, the hardware layer will remain the same, and only the layer above it, i.e., the driver layer, will change. The individual experiment

will help us to isolate the variance caused by the different layers in the experiment and will therefore help us to answer item 1st and item 2nd research question.

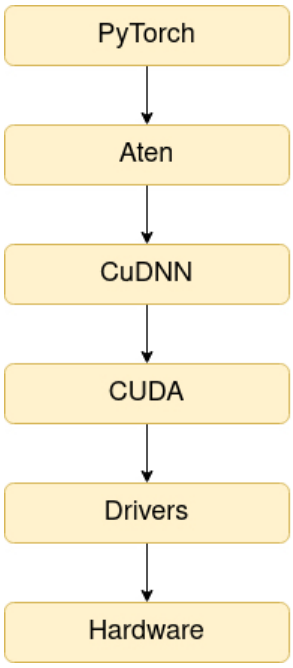


Figure 1.3: Overall architecture diagram

1.3.3 Modify Pypads to facilitate reproducibility of experiment

Pypads is an advanced experiment tracking software use to track the experiment details along with the hardware details of the machine. Pypads functionality included logging CPU and OS details, among other details. As part of this thesis, Pypads functionality will be extended to include crucial GPU details like driver version, name, total memory, UUID, serial number and CUDA version.

The pull request will be generated to contribute the feature in the Pypads GitHub repository[2]. Having these low level, crucial details logged will help to reproduce the experiment. This part will help to answer the item 3rd research question.

Pypads is discussed in more detail in the section 2.5.

[2] `https://github.com/padre-lab-eu/pypads`

2 Related work

There are several different initiatives for experiment reproducibility that can be divided into software such as Data Version Control(DVC) [1], MLflow [2], PyPaDS [3]or documentation methods such as Reproducibility Checklist[20b], Model Cards[Mit+19] and even for academic publishing such as Paper with Code[4].

2.1 Reproducibility Checklist

Prof. Joelle Pineau's student team at McGill University[Bar19] failed to rebuild two virtual characters, namely *half cheetah* and *ant* to its promised performance level. The reason Prof. Joelle Pineau suggest is that the networks are becoming complex with large data sets and colossal computation array that make the model hard to reproduce and expensive to study. The second reason she suggests is that crucial information for reproducibility is missing from the paper, like the number of models trained before the best model was selected, computing power used, and the source code link [Bar19]. Eventually, the model performed up to the promised performance level after the team made changes in the model, which were not mentioned in the lab's paper. Furthermore, the solution she proposed to avoid non-reproducibility of models is a *Reproducibility Checklist*[20b] which ask for such missing information. NeurIPS conference asks the participants to submit the reproducibility checklist along with a paper to encourage the trend of reproducibility in the Machine Learning domain [Bar19].

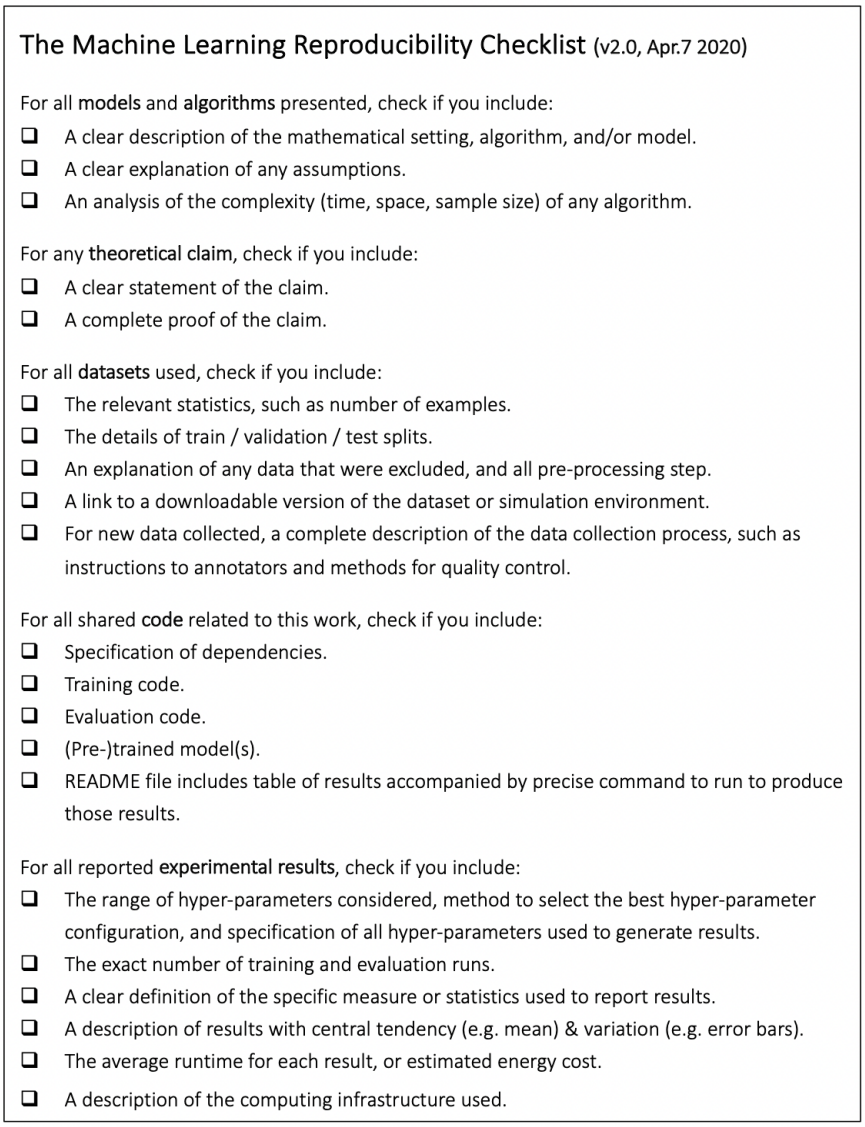
This incidence of non-reproducibility indicates the following aspects of ML:-

[1] `https://dvc.org/`
[2] `https://mlflow.org/`
[3] `https://pypi.org/project/pypads/`
[4] `https://paperswithcode.com/`

1. This showcase the bigger underlying problem in Machine Learning that the codes are not reproducible by other researchers. As a result, it is hard to start from where the previous researcher has left, negatively affecting the community.

2. Reproducibility of results is also essential to validate and understand the State-of-the-art algorithms that claim to have improved results over other methods. In the fragile space of State-of-the-art, the results could not be entirely reliable if they are non-reproducible.

3. Cost is also a problem when additional details related to reproducibility of the experiment are not mentioned like hardware required, computational power, after how many iterations the best result was achieved and hyperparameters. For example, the cost of training XLNet is $245,000, and it takes two and a half days to train on a 512 TPU v3 [Pen]. TPU stands for Tensor Processing Unit, and XLNet is a language model jointly developed by CMU (Carnegie Mellon University) and Google.



The Machine Learning Reproducibility Checklist (v2.0, Apr.7 2020)

For all **models** and **algorithms** presented, check if you include:

- ❑ A clear description of the mathematical setting, algorithm, and/or model.
- ❑ A clear explanation of any assumptions.
- ❑ An analysis of the complexity (time, space, sample size) of any algorithm.

For any **theoretical claim**, check if you include:

- ❑ A clear statement of the claim.
- ❑ A complete proof of the claim.

For all **datasets** used, check if you include:

- ❑ The relevant statistics, such as number of examples.
- ❑ The details of train / validation / test splits.
- ❑ An explanation of any data that were excluded, and all pre-processing step.
- ❑ A link to a downloadable version of the dataset or simulation environment.
- ❑ For new data collected, a complete description of the data collection process, such as instructions to annotators and methods for quality control.

For all shared **code** related to this work, check if you include:

- ❑ Specification of dependencies.
- ❑ Training code.
- ❑ Evaluation code.
- ❑ (Pre-)trained model(s).
- ❑ README file includes table of results accompanied by precise command to run to produce those results.

For all reported **experimental results**, check if you include:

- ❑ The range of hyper-parameters considered, method to select the best hyper-parameter configuration, and specification of all hyper-parameters used to generate results.
- ❑ The exact number of training and evaluation runs.
- ❑ A clear definition of the specific measure or statistics used to report results.
- ❑ A description of results with central tendency (e.g. mean) & variation (e.g. error bars).
- ❑ The average runtime for each result, or estimated energy cost.
- ❑ A description of the computing infrastructure used.

Figure 2.1: Reproducibility checklist[20b]

2.2 Model Cards

It is a fact now that machine learning is used in many decision-making processes and directly impacting life. For example, in law enforcement, machine learning is being used to identify criminals. In the recent case of Robert William[**b42**], a machine learning algorithm wrongfully accused Robert William of stealing five watches from the store. The bias in judgement is that the dataset fed into the system is not sufficient to make the model perform equally robust for white and black faces.

Although machine learning algorithms are inherently unbiased, they seek the easiest way, and thus any biases in the dataset are amplified. If documentation about the input data is non-existent, such biases are difficult to identify. Model cards[Mit+19] is a documentation method addressing such problems. The purpose of this document is to include benchmarks evaluated in different conditions such as different demographics, race, location, skin type that are relevant for the intended use of the model. Model cards will help communicate the intended use of the machine learning model and minimize the use of the model in the context not suitable for its use[Mit+19]. The template suggested for Model Card by the Margaret et al.[Mit+19] is shown in Figure 2.2 and brief details about each of the sections are as follows:-

1. Model Details — Basic information about the model like person or organization developing the model
2. Intended Use — A use case that was kept in mind while developing the model
3. Factors — Include demographic, environment conditions, technical attributes, among others
4. Metrics — Chosen matrix to show the real-world impact of the model.
5. Evaluation Data — Dataset used for quantitative analysis of the model.
6. Training Data — When possible, provide the dataset and when it is not feasible to provide minimum allowable information about the training dataset.
7. Quantitative analysis — Quantitative analysis demonstrates the model's performance on each of the chosen matrix for evaluation, providing confidence interval value whenever possible.

Model Card

- **Model Details**. Basic information about the model.
  - Person or organization developing model
  - Model date
  - Model version
  - Model type
  - Information about training algorithms, parameters, fairness constraints or other applied approaches, and features
  - Paper or other resource for more information
  - Citation details
  - License
  - Where to send questions or comments about the model
- **Intended Use**. Use cases that were envisioned during development.
  - Primary intended uses
  - Primary intended users
  - Out-of-scope use cases
- **Factors**. Factors could include demographic or phenotypic groups, environmental conditions, technical attributes, or others listed in Section 4.3.
  - Relevant factors
  - Evaluation factors
- **Metrics**. Metrics should be chosen to reflect potential real-world impacts of the model.
  - Model performance measures
  - Decision thresholds
  - Variation approaches
- **Evaluation Data**. Details on the dataset(s) used for the quantitative analyses in the card.
  - Datasets
  - Motivation
  - Preprocessing
- **Training Data**. May not be possible to provide in practice. When possible, this section should mirror Evaluation Data. If such detail is not possible, minimal allowable information should be provided here, such as details of the distribution over various factors in the training datasets.
- **Quantitative Analyses**
  - Unitary results
  - Intersectional results
- **Ethical Considerations**
- **Caveats and Recommendations**

Figure 2.2: Model card [Mit+19]

8. Ethical consideration — Contains details about the topics like handling of sensitive information, is the model impacting human life in any way like making decisions in Healthcare.

9. Caveats and Recommendations — This section contains additional concerns not included in the above sections.

The above scaffold suggested is neither complete nor exhaustive and may be adjusted according to model, context and shareholder [Mit+19]. Since the model card aims to provide more details about the model to the user, it helps humans understand the system's limitations and access the helper system's decision accordingly. In the case of Robert William, if the limitation of the model were known to the police officers, the process of catching the right suspect would have been more efficient.

2.3 Execution behaviour of CPU and GPU

Randomness in model accuracy depends on different aspects of model definition, such as weight initialization, optimizer used, or how the dataset is parsed. A model may also produce different results when executed on CPU or GPU due to floating-point arithmetic or the execution sequence, and the model's accuracy might vary. There could be two reasons for this, first the sequence of operations and second, the approximation of floating-point numbers. In order to understand the effect of the sequence of operation (associative) on the result, the first floating-point number needs to be understood. Two types of representation are 32 bit and 64 bit, corresponding to the *float* and *double* data type in C as shown in Fig. 2.3. Since more bits are available to store the number, therefore, *Double* gives better precision than the *Float* datatype. Due to the nature of the floating-point number, addition, subtraction, multiplication, and division does not always give the same result when calculated using GPU. For example, in symbolic mathematics

$$(A + B) + C = A + (B + C) \tag{2.1}$$

however, with GPU, this is not the case even if the operation complies with IEEE 754[20f]. IEEE 754 is the standard for Floating-Point Arithmetic in computer programming. It provides arithmetic format and methods to interchange binary, and decimal floating-point numbers in computer programming [20f].

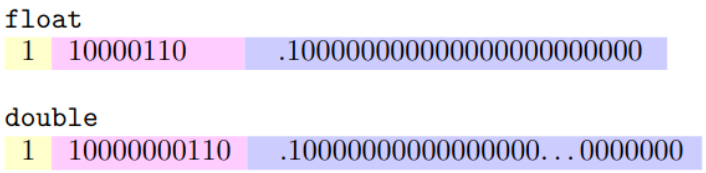


Figure 2.3: Float and Double representation of number -192 [11]

The computation equivalent of L.H.S and R.H.S of *Equation* 2.1 when handled by a GPU are as follows:-

$$LHS = rn(rn(A + B) + C) \tag{2.2}$$

$$RHS = rn(A + rn(B + C)) \tag{2.3}$$

where rn refers to rounding-off

As evident from the *Equations* 2.2 & 2.3 that they are not equal. Therefore, the answers vary based on the sequence of the operation in a GPU. This limitation is not specific to CUDA, but an inherent problem of using parallel computing on floating-point values [NVIa].

The problem of approximation as shown in *equation* 2.4 & 2.5 can be solved by using Fused Multiply-Add (FMA) which is included in IEEE 754 [20f] in 2008. FMA performs the operation of multiply and addition with only one rounding-off step, as a result, it is faster and more accurate as compared to the previous example where two rounding-off steps were executed [11].

Without FMA operation, result will be computed in two rounding steps as shown in equation 2.4

$$rn(rn(X * Y) + Z) \tag{2.4}$$

With FMA operation, result will be computed in one rounding step as shown in equation 2.5

$$rn(X * Y + Z) \tag{2.5}$$

where rn refers to rounding-off. The reason for the better result of FMA is that it converts the operation to a 1-instruction sequence instead of a 5-instruction sequence of traditional methods. FMA is by default enabled in NVIDIA CUDA Compiler (NVCC)[21b]. It has also been reported[20e] that the FMA flag, when turned off, has given a boost in performance by 5%. This boost is observed on a kernel dominated by only multiply operations because there are not many addition and multiplication fusion operations needed, hence the FMA flag=false is giving better results. In such a case, turning the FMA flag to false is more useful.

2.4 Experiment tracking software — MLflow

The machine learning models lack a standard project structure, making managing and tracking the projects hard. Although the version control systems are helpful to manage the state of the project, tracking the parameters responsible for a particular result is not possible with such a mechanism. Therefore, to solve this problem an open-source platform called MLflow[5] offers following tools for machine learning lifecycle [20d]

1. MLflow Tracking — Track which results and the hyperparameters responsible for those results.
2. MLflow Project — Provides a standard format for packaging data science projects.
3. MLflow Model — Provides a mechanism to package and deploy machine learning models.
4. MLflow Registry — Stores model centrally to enable collaborations among teams.

2.4.1 MLflow Tracking

According to the MLflow documentation[20d]"MLflow Tracking is an API and UI for logging parameters, code versions, metrics, and artifacts when running the machine learning code and for later visualising the results". It can be used in a script as well as in a notebook to log results locally. It also allows logging the results to a server for later retrieval and comparison amongst the different runs.

The details are saved for each run of the data science code. According to the document[20d] following information about the experiment are saved:-

1. Code version — Git commit hash that was used to run the code
2. Start and End time — start and end time of the run
3. Source — Filename which launched the run, or the MLflow Projects run file name, whichever is applicable.
4. Parameters — Takes user-defined key-value pairs in string format

[5] `https://mlflow.org/`

5. Metrics — Similar to *Parameters*, Metrics take key-value pair in numeric format. Additionally, it allows changing the key-value pair throughout the run and later visualises the change in parameter values using the history.

6. Artifacts — Save the file in any format, for example, images, data model or Apache Parquet files[21a].

MLflow documentation also reports that runs can be recorded to a local file, using a database compatible with SQLAlchemy, or a tracking server.

2.4.2 MLflow Projects

According to MLflow Projects documentation[21j] "MLflow Projects are a standard format for packaging reusable data science code. Each project is simply a directory with code or a Git repository, and uses a descriptor file or simply a convention to specify its dependencies and how to run the code".

According to the documentation[21j], MLflow projects are a method for organising and managing the code so that it is easier for other researchers or automation tools to execute the code. The MLflow projects could be a directory containing the code or a git repository containing the code. Placing files in the designated folders helps MLflow to access them and perform automated tasks, for example having a *conda.yaml* will be treated as a Conda[6] environment. Other details about the experiment could be specified in the *MLproject.yaml*. According to the documentation[21j] following three properties could be specified:-

1. Name — specifies the name of the project

2. Entry Points — Contains the information about the entry point to a project. For example, a .py file or a .sh file could be used to run the whole experiment. It is also possible to specify the parameters for these entry points, including data types and default values.

3. Environment — The software environment that should be used to run the project. For example, a Conda[6] environment to manage python-based dependencies and Docker[7] to manage non-python dependencies like Java.

[6] `https://docs.conda.io/en/latest/`
[7] `https://www.docker.com`

2.4.3 MLflow Models

According to the MLflow documentation[21i] "An MLflow Model is a standard format for packaging machine learning models that can be used in a variety of downstream tools—for example, real-time serving through a REST API or batch inference on Apache Spark. The format defines a convention that let us save a model in different "flavours" such that different downstream tools can understand."

Flavours are a key feature of MLflow models. The documentations says " they are a convention that deployment tools can use to understand the model, which makes it possible to write tools that work with models from any ML library without having to integrate each tool with each library." There are few standard flavours like Python_function, which tells the MLflow that the model should be executed as a regular python function. There could be another case where the model should be executed as a sklearn model, and therefore the model should be aware of it. In order to enable Mlflow model to run sklearn models, mlflow.sklearn[8] is defined.

It helps to load a model as a sklearn pipeline.

2.4.4 MLflow Registry

According to the MLflow Registry documentation[21g] "MLflow Model Registry component is a centralised model store, set of APIs, and UI, to collaboratively manage the entire lifecycle of an MLflow Model. It provides model lineage (which MLflow experiment and run produced the model), model versioning, stage transitions (for example from staging to production), and annotations."

The entire lifecycle of an MLflow registry according to documentation [21g] could be summed up in five concepts as follows:-

1. Model — MLflow model, once created from an experiment, could be used to register the model in Model Registry.

[8] `https://www.mlflow.org/docs/latest/python_api/mlflow.sklearn.html#module-mlflow.sklearn`

2. Registered Model — It is used to register an MLflow model in the model registry. The registered model has a unique name and stores other metadata like version and model lineage.

3. Model version — The registered model is initialised with version number one. The version is incremented when the model is registered with the same unique name again.

4. Model stage — It reflects the software stage in which the model is right now. Predefined *Model stages* are *Staging, Production and Archived*. The model could be easily marked from *Staging* to *Production* or vice versa.

5. Annotations and Descriptions — Models could be annotated using markdown files to include experiment details. The details could be version-specific dependencies or methodology in general.

2.5 Experiment tracking software — PyPads

PyPads is an advanced machine learning experiment tracking software build on top of MLflow to automatically log details like CPU cores and clock speed, among other parameters, to the source of choice. MLflow is discussed in details in the previous section 2.4 *Experiment tracking software — MLflow*. The source could be the local directory, local tracking server or even a remote-tracking server. The process to install and use PyPads is clean and simple. Make use of pip to install the PyPads and include just a few lines of code to start logging the details on a local machine in .pypads folder.

Following concepts forms the key idea of PyPads[21h]:-

1. Actuators — are used to manipulate the experiments. Therefore it may affect the result of the experiment. The changes could be to the machine learning code, setup amongst others. An example could be to set the value of random using the PyPads actuators.

2. API — Enable users to access PyPads functionalities. Functionalities like start and stop a run; log artifact, matrices or parameters; set tags with metadata. Additionally, some of the MLflow features can also be accessed via PyPads API.

3. Validators — Check for the validity of the experiment status or code against given properties. The validator task is to let the user know about the invalidated parameters at the runtime. The ideal scenario is that the validity report is logged without any error. A future feature would be to interrupt the execution of the programme if the validator fails.

4. Setup / Teardown functions — Also called pre- and post-run functions. They are used to logging in experimental details like git, hardware and environment details. Hardware details include a number of CPU physical cores, CPU total cores, maximum frequency and minimum frequency.

5. MappingFiles — Mapping files are YAML [9] files which are used to deliver hooks to the libraries. The function of these hooks is to enable the tracking functionality inside the libraries. This is achieved by marking up functions, classes and modules in the mapping files.

[9] `https://yaml.org`

6. Decorators — Like mapping files, decorators are also used to define the hooks. The difference is that decorators change the libraries, which is not a best practice. Therefore, decorators are not used often.

7. Logging functions — are the generic functions connected with hooks of libraries to track the parameters in the experiment. Configuring the logging function is easy as they require to be passed at the initialisation of the PyPads app via the constructor. A logging function is mapped to a hook. A hook can trigger single or multiple events, which results in the execution of the logging function. PyPads is very customizable and provides a possibility to define hooks and their events.

8. Checkpoints — are a work in progress. When fully developed, they will be able to provide the ability to rerun the experiment in the future from the defined checkpoint.

Motivation to choose PyPads over other available reproducibility software was three folds:-

1. Logs automatically
2. Easy to extend login functionality
3. MLflow visualisation
4. Log to different backend possible
5. Backend configured and provided by University of Passau
6. Open source
7. Based on MLflow — gives CPU details
8. Funded by Bavarian Govt and developed and maintained by Uni Passau, therefore wanted to contribute to the homegrown software.

Therefore, PyPads could help make the model reproducible by providing underlying details and hence is proposed to be used in this experiment.

2.6 Experiment tracking software — Data Version Control (DVC)

DVC is a data and machine learning tool that is making use of the existing tools like CI/CD and GIT [21f]. Initialising the DVC is as easy as initialising a git repository in the local machine. DVC gives the option to store and reference the data, features and model in a git commit format. Figure 2.4 demonstrate the timeline of commits. The leftmost commit 7fe5fc5 is the earliest commit and 020c55f is the latest commit. Every commit consists of a respective data, feature, and model reference, which could be used to pull and run them easily. Like git commits [10] the DVC also provides the feature to name each state for easy retrieval.

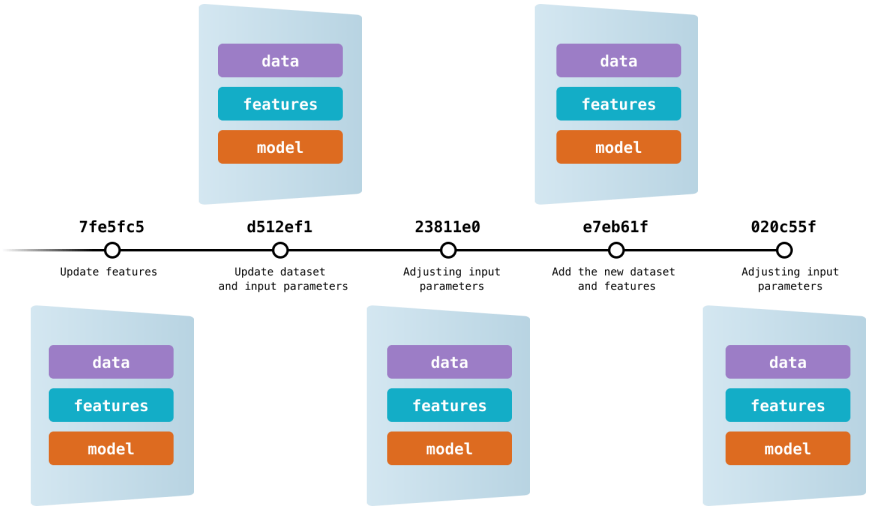


Figure 2.4: DVC saving right version of data, feature and model [21l]

DVC achieved this using a metadata file which is provided by the user once and committed to the git. The metadata files include the description of the dataset, model, or other machine learning artifacts are tacked. DVC tracks those changes and maintains them in a separate workspace, but it is still connected to the original Git repository to track changes. The process of checking out the data, features and model from the DVC

[10] https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

is as simple as check out a commit from the git repository. The following commands are used to check out from git and DVC[21f]c:- $gitcheckoutv1.0$ dvc checkout

2.6.1 Sharing data and model values

DVC allows collaboration just like git. With the help of remote storage, DVC enables data and model to be pulled together in one machine by the users. The supported remote storages are Amazon S3[11], Microsoft Azure Blob Storage[12], Google Drive [13], Google Cloud Storage[14], HDFS[15].

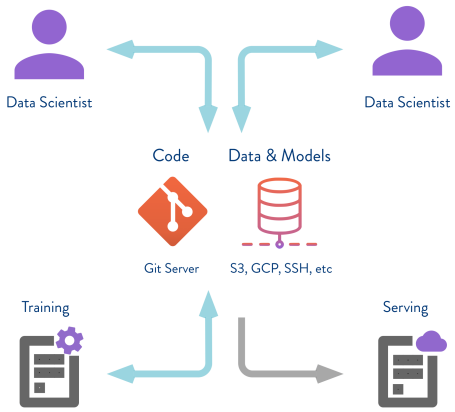


Figure 2.5: DVC sharing data and model values [21k]

As shown in figure 2.5 the user could save the code in the git and data & model in the DVC, which the other users could easily access. The way this setup is done is as follows:-

1. Create an S3 container.
2. Setup DVC remote by passing the S3 container URL setup in the previous step.
3. Commit code at git, respective data and feature changes will be committed automatically to DVC.
4. At the receiver side, clone the repository

[11] https://aws.amazon.com/s3/
[12] https://azure.microsoft.com/
[13] https://drive.google.com/
[14] https://cloud.google.com/storage/
[15] https://hadoop.apache.org

5. Pull the data from the DVC using $ dvc pull command
6. Whenever a new commit is available, it will be available for a pull and associated data, feature and models too, as shown in figure 2.4.

2.7 Experiment tracking software — Sacred

The Sacred documentation[21m] shares a short verse about the experiment, and every researcher could relate to it, and the verse goes:-

Every experiment is sacred
Every experiment is great
If an experiment is wasted
God gets quite irate

Sacred is a tool to help reproduce the experiments by configuring, organising and logging the experiment details[21m]. The responsibilities include[21m]:-

1. tracking all the parameters of the experiment
2. run experiments on different settings
3. configurations can be saved for individual runs
4. reproduce results

Following mechanism enable Sacred to achieve above stated functionalities[21m; 21k]:-

1. ConfigScopes — are function with a decorator that tell Sacred to turn local variables into the configuration entries.
2. Config Injection — All the functions have access to the configuration parameters by name. It is done by automatically injecting the configuration parameters.
3. Command-line interface — Could be used to change the parameters and run the experiment, making it a powerful CLI.
4. Observers — provides the functionality to log all the details about the experiment like the configuration used, the dependencies, the machine on which experiment is being executed, the results and the other details. These details could be saved in a MongoDB[16] for permanent and easy access in future.
5. Automatic seeding — enables reproducibility of the experiment by controlling the randomness in the experiment.

[16] `https://www.mongodb.com`

2.8 Virtual machine (VM)

Virtual machines similar to any other computing machine in modern-day work like servers, computers, smartphones have a CPU, memory, disks and the ability to connect to the internet when required[21o].

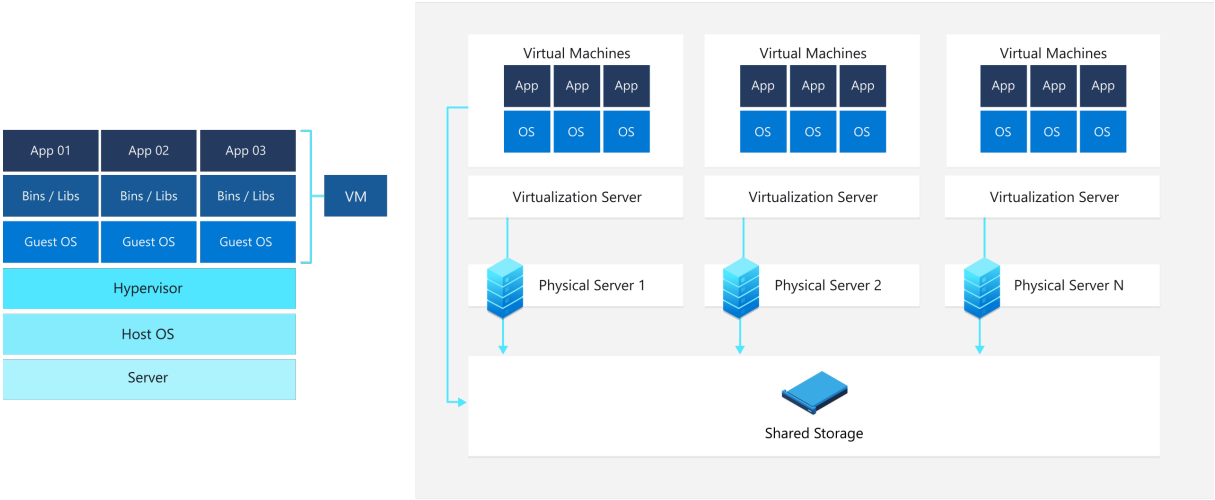


Figure 2.6: Virtual machine [21o]

The main difference between a computer and VM is that the components like CPU, memory, and disks are physical and tangible [21o], whereas, in the case of VMs, those components are not tangible but rather software-defined resources which, of course, run on top of the physical hardware. Figure 2.6 represents the high-level architecture diagram of VMs[21o].

According to documentation[21o], virtualization is the process of creating a software-based version of the dedicated amount of computer resources like CPU, RAM and storage space. These resources could be part of the local laptop or the remote cloud cluster. A virtual machine is a computer file, commonly known as an image, which behaves like an actual machine[21o]. It can run a different operating system independently of the host operating system, acting as a workspace to a user. VM runs independently of the other host machine, which means it does not interfere in the execution of other operating systems[21o].

The independent nature of VMs is advantageous in many use cases, and some of them are as follows[21o]:-

1. Deploying and building cloud application.
2. Installing new operating system independently inside the existing operating system.
3. Creating new development or test environments for developers to code features and test them before deploying them in the production system.
4. Creating a backup of the existing operating system.
5. Accessing an old application by installing the older operating system instance.
6. Testing and running software or applications on a new operating system that might not be supported.

2.9 Docker

Docker is an open platform written in Go programming language [17] for developing, shipping, and running applications [21d]. Docker enables us to separate the application from the underlying hardware. As a result, we can migrate the code to new hardware and start running the code with relative ease[21d]. Docker achieves this using the container as shown in Figure 2.7.

2.9.1 Docker container

Docker makes use of many Linux features to deliver its functionality[21d]. It uses a technology called *namespace* to provide the isolated workspace, which is known as Containers [21d]. A container is defined as "a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another"[21n].

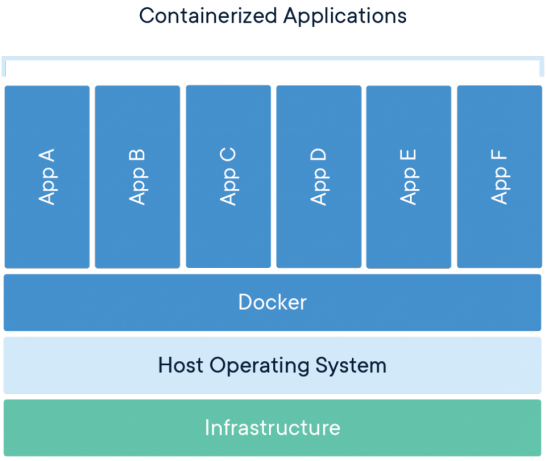


Figure 2.7: Docker container [21n]

When we execute a container, Docker creates a set of the namespace for that container. These namespaces are responsible to enables the container to be isolated. Each aspect

[17]https://golang.org

of the docker container runs in a separate namespace, and its access is linked to that namespace[21d]. In high level understating, the container encapsulates the applications and works on top of the host operating system. The application does not interact with the host operating system or the infrastructure directly but via Docker container. This helps to make the applications, operating system and infrastructure-independent. This isolation and security allows us to run containers simultaneously and on many systems[21d].

Docker streamlines the development process by providing an environment similar to the staging or production environment but scaled down to a developer local machine or development cloud instance. Containers are also useful for the Continuous integration and continuous delivery (CI/ CD)[21d]. The usefulness of containers could be explained using the following workflow[21d]:-

1. The developer is coding the features and sharing them with other developers using the docker container.
2. Developers are pushing the docker container into the test environment for further testing.
3. During testing, if the developer or other team member finds a bug, the developer fixes the bug in the development environment and pushes the updated docker image in the test environment.
4. Finally, when bugs are fixed and tested, the feature can be shared with the stakeholders by deploying this updated docker image on the production environment.

2.9.2 Docker architecture

Docker uses a client-server architecture[21d]. Figure 2.8 shows the three major parts of docker architecture, and we will discuss them in this section. The *Docker client* communicates with the *Docker daemon*. Daemon and client could be present on the same machine, or daemon could be present on the remote server[21d]. The daemon and client communicate using REST API over the UNIX socket or a network interface[21d].

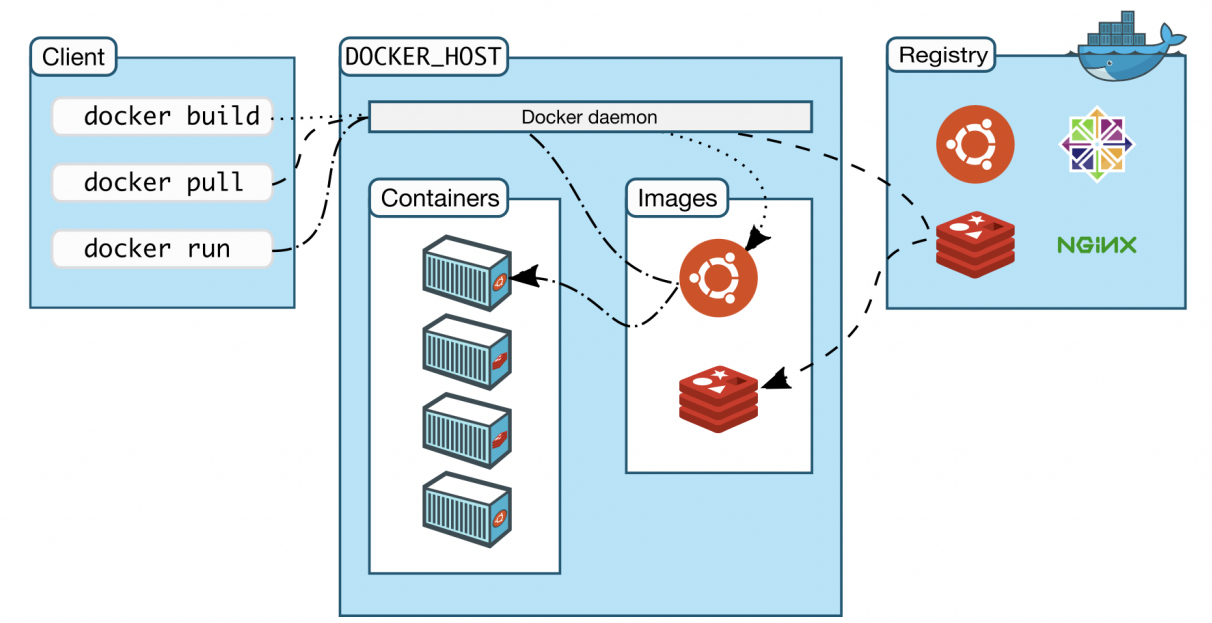


Figure 2.8: Docker architecture[21d]

Another docker client is *docker compose* that allows us to work with multiple containers at once.

Docker Daemon

The docker daemon, as shown in Figure 2.8 communicate with docker API and manages docker containers, volumes, images and network. The docker daemon can also communicate with other docker daemons[21d].

Docker client

Docker client, as the name suggests, is the primary way the user interacts with the docker. When the user runs the command, taking example from the Figure 2.8 *dockerbuild*, this command is sent to docker daemon, which then executes it[21d]. A docker command can communicate with multiple docker daemons[21d].

Docker registries

A docker registry stores the docker images. Docker hub is a public repository that is accessible to all and is, by default docker is configured to look for images in docker hub. Docker can be configured to point to its private repository instead of the docker hub.

When commands like *dockerpull* or *dockerrun* are executed, images are pulled from the configured registry. When a command like *dockerpush* is used, the image is pushed to the configured registry.

2.9.3 Docker vs Virtual machine(VM)

Docker uses containers, and containers are an abstraction of application and its dependencies together[21n]. Multiple containers can run on a machine sharing the same OS kernel but running totally in isolation[21n]. Containers are generally very small in size, for example, in few MB, and require fewer VMs and operating system[21n].

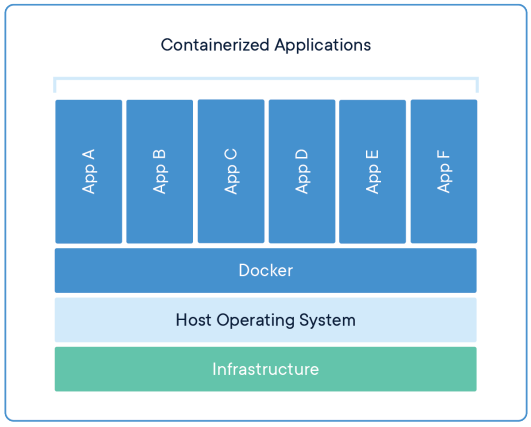


Figure 2.9: Docker[21n]

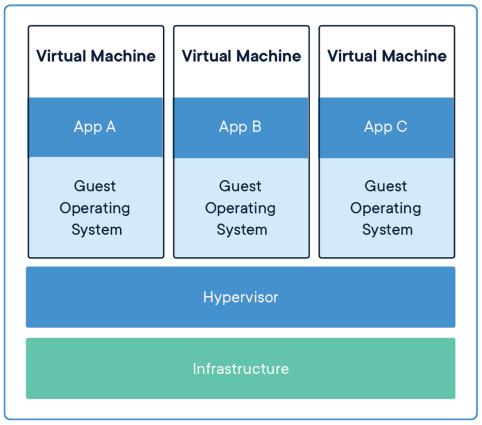


Figure 2.10: Virtual machine[21n]

Virtual machines, as shown in Figure 2.10 are an abstraction of physical hardware. This helps to divide one server into multiple different servers virtually, and each of them acting independently[21n]. The hypervisor allows multiple virtual machines to run single hardware. Each virtual machine contains its operating system, the application, required dependencies, thus taking up space in GBs[21n]. Our model needs to be deterministic, and we want the environment to be easily transferable to other servers. Therefore, docker is more suitable for us than a virtual machine.

3 Background

In this chapter, we present a brief introduction to the experiment related concepts and dataset before presenting our experimental setup in chapter 4. We explain briefly about the Logistic map, neural networks: Artificial neural networks, CNN, VGG, optimisers, and finally about the dataset.

3.1 Logistic Map

First-order difference equation finds their application in many contexts from biology and economics to social science. These nonlinear difference equations can possess an extraordinarily rich spectrum of dynamical behaviour, from stable points, through cascades of stable cycles, to a regime in which the behaviour (although fully deterministic) is in many respects "chaotic", or indistinguishable from the sample function of a random process [May76].

Owing to this simple nature and dynamic results, logistic map is chosen as the benchmark test for the experiment. Logistic map could showcase how much variation could be induced by hardware and driver versions over 100s of epochs when executing a relatively simple equation. The logistic map is defined by the equation[16a] 3.1

$$X_{n+1} = rX_n(1 - X_n) \tag{3.1}$$

where r being the growth parameter and remains constant for the whole experiment; X_n is the previous value and X_{n+1} is the current value [16a]. For our experiment, we have run the Logistic Map for 2000 epochs. At $n = 1$ equation 3.1 will be as follows:-

$$X_1 = rX_0(1 - X_0) \tag{3.2}$$

where X_0 is the initial value set at the starting of the experiment. For this experiment $n = 2000$ and $r = 3.7$. According to author Robert May [May76] the equation results to $-\infty$ when $1 < r < 4$ is not followed.

3.2 Neural Network

Image classification is an important branch of computer vision[Guo+17]. It is widely used nowadays because modern image classification techniques are faster and could find use in various domains like a scientific experiment, traffic analysis, medical science, among others[Guo+17]. Image classification techniques have evolved a lot from a simpler model with a single hidden layer as shown in figure 3.1 to a deep neural network with multiple hidden layers as shown in 3.2 [XD18].

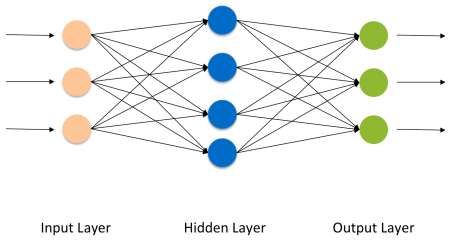


Figure 3.1: Simple neural network

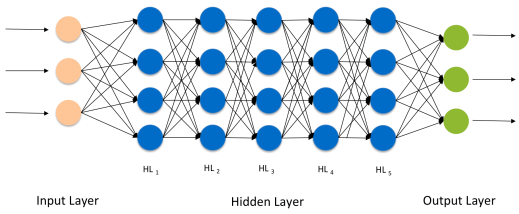


Figure 3.2: Deep neural network

3.2.1 Artificial neuron

In both shallow and deep neural networks, the key building block is an artificial neuron. An artificial neuron is an oversimplified approximation of a biological neuron that is useful for computer science applications. The *Perceptron* — the simplest artificial neural network, was invented by Frank Rosenblatt[Ros58] Properties of biological neurons like time delay are not taken simulated in the artificial neurons. A neuron is a mathematical function as shown in figure 3.3 which takes input $x_1 - x_m$ either from other neuron or from external inputs, multiply it with the corresponding weights $w_{i1} - w_{im}$ and add bias b_i to it. After calculating the weighted sum, the value is passed to the activation function, which decided if the neuron will fire provided the output of the activation function is higher than the threshold [Suz11]. By virtue of neural network architecture,

a deep neural network has more number of layers and subsequently more number of neurons. The availability of more neurons helps the deep neural network to extract more abstract details from the input.

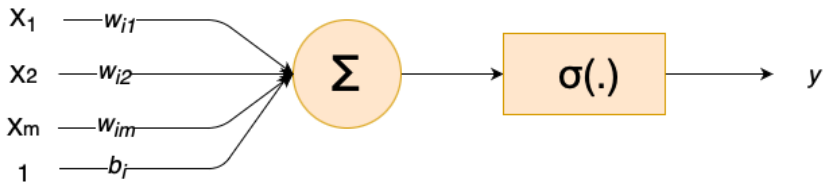


Figure 3.3: Single Neuron/ Perceptron

A typical image classification model includes Image preprocessing, Image segmentation, key feature extraction and matching identity. With the rise of deep learning, feature extraction and classifier has been integrated into the learning framework to eliminate the traditional method of feature selection difficulty. The idea behind deep learning is to find multiple levels of representation with the help of high-level features representing abstract semantics of data. [Guo+17]

3.2.2 Convolutional Neural Network (CNN)

In recent years for image classification, the use of one particular class of deep learning, i.e., Convolutional Neural Network, has risen mainly because of the success of deep learning models like AlexNet[KSH12] in improving the image classification capabilities of a computer. CNN was first designed in 1998 by LeCun[Lec+98] and consists of three main layers:-

1. Convolutional Layer
2. Pooling layer
3. Fully-connected layer

Convolutional Layer is the core layer of the CNN, consisting of local connection and weight of the shared characteristics[Guo+17]. The objective of this layer is to learn the feature representation of the input data, and it is achieved by feature maps[Guo+17]. The convolutional layer consists of many feature maps[Guo+17]. Each neuron of a feature map is used to extract local characteristics from the different positions of the input

image[Guo+17]. Different features could be obtained if the input layer is convolved with a different feature map[Guo+17]. Finally, results are passed to a nonlinear activation function like sigmoid, tanh and ReLu[Guo+17].

Pooling Layer is usually placed between two convolutional layers[Guo+17]. The pooling layer could reduce the size of the input image and increase the robustness of the feature extraction[Guo+17]. It acts as a secondary feature extraction layer[Guo+17]. Examples of pooling layers are average pooling layer[Wan+12] and max pooling layer[BPL10]. High-level characteristics could be extracted by arranging several convolutional layers and pooling layers next to each other[Guo+17].

Fully-connected Layer as the name suggests, takes neurons from the previous layer and connect them together in the current layer. This layer does not have spatial information, and the last fully-connected layer is followed by the output layer. The output layer is responsible for the classification task, and it is achieved by adding a softmax regression layer which gives the probability distribution of the output.[Guo+17]

As mentioned by author Tianmei et al.[Guo+17] many more models have been proposed to improve the performance of AlexNet[KSH12] and some of them are ZFNet[ZF13], GoogleNet[Sze+14] and VGGNet[SZ15]. We will discuss VGGNet[SZ15] in more detail in the upcoming section.

3.2.3 Image classification Model — VGG

The image classification model used for this experiment is based on the VGG[SZ15] architecture. VGG[SZ15] is chosen for this experiment because is the winner of ImageNet Challenge 2014, provide models publicly and their representation generalise for other datasets where they achieve state-of-the-art results [SZ15]. The experiment model is based on the VGG-11 configuration, where 11 stands for the number of weighted layers. The only change being the experiment uses only one Fully connected layer as opposed to the suggested three fully connected layers. The reason is that the dataset used is fairly small, and the training time of the model has also decreased. The experiment architecture and the VGG architecture is shown in table 3.1.

VGG stands for Visual Geometric Group, which is named after the group of Oxford where the researcher were working. VGG is

3.2.4 Gradient descent

There are three gradient descent variants[Rud17]:-

1. Batch gradient descent
2. Stochastic gradient descent
3. Mini-Batch gradient descent

3.2.5 Batch gradient descent

Batch gradient descent calculated the gradient of cost function w.r.t the θ for the whole dataset[Rud17]. Mathematical notation of the batch gradient descent is represented by the equation 3.3 [Rud17; 21e]

$$\theta = \theta - \eta.\nabla_\theta J(\theta) \tag{3.3}$$

where θ represents whole training dataset η is the learning rate or step and ∇ is the gradient function of θ.

As the batch gradient descent require to calculate the gradient of whole dataset therefore, it can be slow, and it might be not feasible to perform batch gradient descent on datasets which are large in size as they will not fit in memory.

It also does not work with real time data because it expects the whole dataset at the start of the calculation.

3.2.6 Stochastic gradient descent (SGD)

Stochastic gradient descent takes a different approach from the Batch gradient descent. SGD updates the parameters for each training. The mathematical notation for SGD is represented by the equation 3.4 [Rud17; 21e]

$$\theta = \theta - \eta.\nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{3.4}$$

where θ represents whole training dataset η is the learning rate or step and ∇ is the gradient function of θ, $x^{(i)}$ and $y^{(i)}$. $x^{(i)}$ being the training sample and $y^{(i)}$ being the label.

Batch gradient descent calculate gradient for whole dataset every time it needs to update a parameter. This leads to redundant updates. SGD saved this redundant calculation by updating the parameters one update at a time, as a result it is much faster than batch gradient descent. SGD update frequently and with high variance, resulting in the objective function to fluctuate heavily as shown in Figure 3.4 [Rud17]

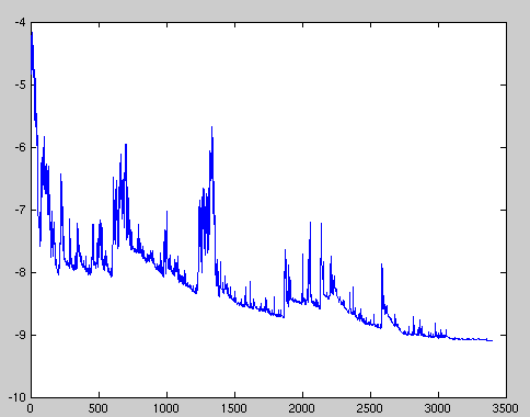


Figure 3.4: SGD fluctuation[Rud17]

Batch gradient descent converges to the minima of the basin in which parameters are placed[Rud17]. In case of SGD fluctuation enables it to jump to new minima which could be better, but it could also be possible that SGD keeps on shooting and therefore, making it hard to converge to the exact minimum[Rud17]. It is claimed by author[Rud17] that with the decrease in the learning rate of SGD it performs equivalent to batch gradient descent, i.e., converging to the local minima in case of non-convex optimisation and converging to global minimum in case of convex optimisation.

3.2.7 Momentum

Ravines as defined by author[Rud17] is "areas where the surface curves much more steeply in one dimension than in another, which are common around local optima." SGD could not effectively navigate through ravines, i.e., SGD keeps oscillating across the slope of the ravines and do not make much progress about the local optimum[Rud17].

Figure 3.5 and 3.6 showcase the behaviour of SGD with and without the momentum, respectively.

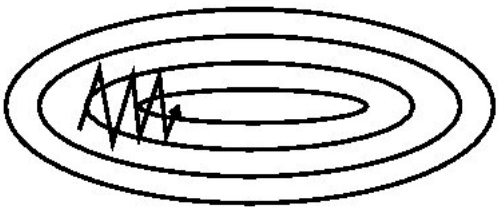


Figure 3.5: SGD without momentum[Rud17]

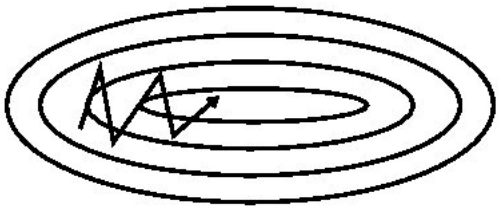


Figure 3.6: SGD with momentum[Rud17]

Momentum[Qia99] helps to accelerate the progress of SGD in the desired direction and dampens the oscillation[Rud17] as seen in the figure 3.6. This is achieved by adding a fraction γ of the past update vector to the current update vector, as shown in Equation 3.5 and Equation 3.6.

$$\nu_t = \gamma\nu_{t-1} + \eta\nabla_\theta J(\theta) \tag{3.5}$$

$$\theta = \theta - \nu_t \tag{3.6}$$

where θ represents the whole training dataset, γ represents the momentum term, ν represents update vector, η is the learning rate or step and ∇ is the gradient function of θ.

In order to understand by taking an example of physics, we could say that when we are using momentum, we are rolling the ball down the hill [Ben21]. The ball accumulates momentum as it rolls down the slope and increases its velocity until it reaches its terminal velocity [Ben21]. Similarly, with SGD, the momentum term increases for dimensions whose gradient points in the same direction. Conversely, it reduces the update for the dimensions whose gradients change direction. This behaviour of momentum leads to faster convergence and lower oscillation[Rud17].

3.2.8 Nesterov accelerated gradient (NAG)

In continuation with the explanation of the momentum as a rolling ball from the previous section, we could observe the scenario where the ball is rolling without the notion of direction is unsatisfactory[Rud17]. The Nesterov accelerated gradient[NES83] is based on the idea to make the ball small, aware of it's surrounding and stop the ball rolling when the slope is changing direction, i.e., when the hill slopes up[Rud17]. The following equations Equation 3.7 and Equation 3.8 can be used to represent the NAG [Rud17; 21e]:-

$$\nu_t = \gamma\nu_{t-1} + \eta\nabla_\theta J(\theta - \gamma\nu_{t-1}) \tag{3.7}$$

$$\theta = \theta - \nu_t \tag{3.8}$$

where θ represents the whole training dataset, γ represents the momentum term, ν represents update vector, η is the learning rate or step and ∇ is the gradient function of θ.

3.2.9 ADADELTA

Adadelta [Zei12] is based on ADAGRAD[DHS11] and improves the two main drawbacks of the latter, namely:-

1. Learning rates decay throughout the training,
2. Requirement to provide a global learning rate

In the ADAGRAD[DHS11] method, the squared gradient is accumulated from the beginning of training. As each term is positive, the cumulative sum increases throughout the training keep on increasing. As a result, the learning rate is decreased, and eventually, it becomes infinitesimally small[Zei12].

To solve this problem in ADAGRAD the sum of squared gradients is not accumulated from the start of the training, but this is restricted to some fixed size W. The benefit of this window is that the denominator of ADAGRAD cannot tend to infinity, but to

a finite value based on the recent gradients. This will make sure the learning rate will not tend to be infinitesimally small, and the model will continue to run iteration over iteration[Zei12].

The author suggests even a better way to store the previous gradients, and that is to store them exponentially decaying average of the squared gradient. This can be explained by the following equation[Zei12]:-

Let at time t the running average is, $E[g^2]_t$ then we could say:-

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2 \tag{3.9}$$

where ρ is decay constant like momentum method. As we require the square root of this Equation 3.9 for parameter update, therefore it becomes RMS and can be rewritten as

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon} \tag{3.10}$$

where ϵ is added to better condition the denominator as in [BL89]. The resulting equation is

$$\Delta x_t = \frac{\nu}{RMS[g]_t} g_t \tag{3.11}$$

3.2.10 Datasets

In order to perform image classification, a dataset is required. One could provide all the images by themselves by mining or clicking photographs on their own. There is another way to use the standardised dataset used in research work. There are many such image datasets available for the experiment. From huge and complex datasets like ImageNet[1] which have over 14 million instances; 20,000 unique classes, and size of over 150 GB. To German Traffic Sign Detection Benchmark dataset [2] consisting of over 900 instances. The dataset chosen for this experiment is CIFAR-10[17] which has about 60,000 instances, ten classes and a size over 175 MB. Details about the CIFAR-10 dataset is as follows[17]:-

[1]https://image-net.org
[2]https://benchmark.ini.rub.de

CIFAR-10	
Attribute	**Value**
Total Classes	10
Total Images	60000
Resolution	32 x 32
Images in training set	50000
Images in testing set	10000
Name of all classes	airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck

Table 3.2: CIFAR-10 dataset statistics[17]

CIFAR-10 is chosen because it is a standard dataset for image classification, and there are eight different methods between the accuracy range of 93%-94% on the benchmark[3] and the difference between the two consecutive methods is also very small. This gives a unique opportunity to test our hypothesis that is the difference in the methods is due to the change in model or is it a serendipitous result achieved by the hardware and software.

Figure 3.7: CIFAR-10 dataset example images

The figure 3.8 shows that the dataset shows equal distribution of classes in Training, testing and the whole dataset. The training set consists of randomly selected 50,000 images per class. The testing set consists of 10,000 images randomly selected per class; similarly the whole dataset consists of 60,000 images per class.

[3]https://benchmarks.ai/cifar- 10

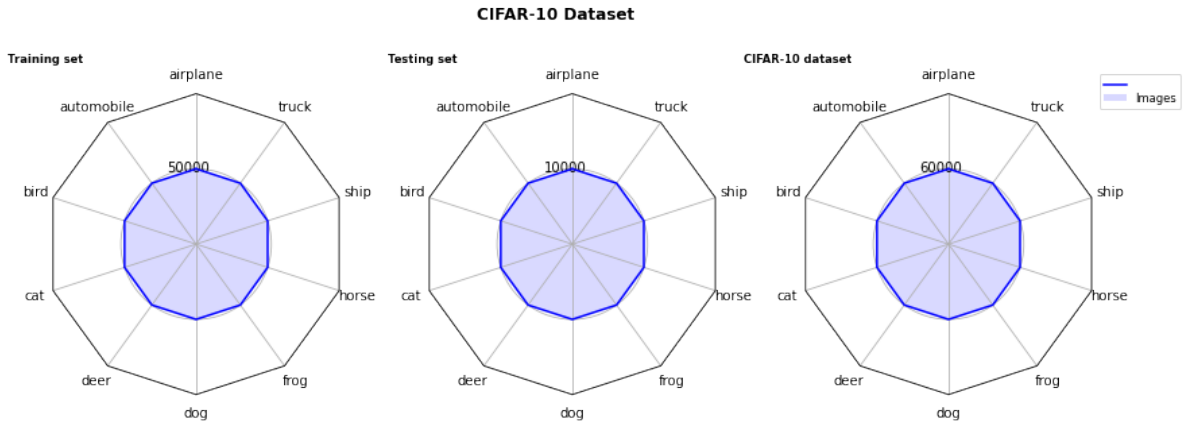


Figure 3.8: Image showing distribution of classes in Training, Testing and whole CIFAR-10 dataset

The table 3.3 shows the eight methods on benchmark.ai leaderboard[3]. We can see the difference between the two consecutive methods is sometimes even as low as 0.06%, as in the case of methods 3 and 4. The fact that there are so many methods already available for the CIFAR-10 dataset, and they are so close to each other in terms of accuracy, makes CIFAR-10 dataset ideal for this experiment. A large number of methods shows that this is a standard dataset that has been worked upon quite a lot. The presence of accuracy so close to each other gives us a unique opportunity to test our hypothesis that variance is induced with change in CPUs, GPUs and/ or driver versions of the GPUs.

CIFAR-10		
#	**Method**	**Accuracy**
1	Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree (Sep 2015, AISTATS 2016) [LGT15]	93.95%
2	Spatially-sparse convolutional neural networks (Sept 2014) [Gra14]	93.72%
3	Scalable Bayesian Optimization Using Deep Neural Networks (Feb 2015, ICML 2015) [Sno+15]	93.63%
4	Deep Residual Learning for Image Recognition (Dec 2015) [He+16]	93.57%
5	Fast and Accurate Deep Network Learning by Exponential Linear Units (Nov 2015) [CUH15]	93.45%
6	Universum Prescription: Regularization using Unlabeled Data (Nov 2015) [ZL15]	93.34%
7	Batch-normalized Maxout Network in Network (Nov 2015) [CC15]	93.25%
8	Competitive Multi-scale Convolution (Nov 2015) [LC15]	93.13%

Table 3.3: CIFAR-10 leaderboard methods between 93% and 94% [3]

3.2.11 Learning Rate Scheduler

Determining an optimal learning rate for the experiment is a challenging task. An ideal learning rate will quickly learn and will converge to a good solution. If the learning rate is too high, it might result in jumping to the other side of the valley on the gradient descent curve[Gro17], therefore defeating the purpose of finding the minima of the valley as shown in Figure 3.9. On the other hand, if the learning rate is too low, then the steps taken will be small and the time to converge will be high for the algorithm Figure 3.10 [Gro17].

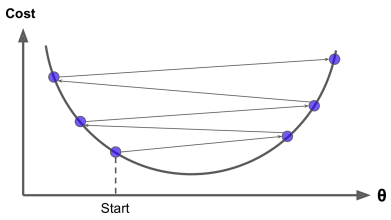


Figure 3.9: SGD without momentum[Gro17]

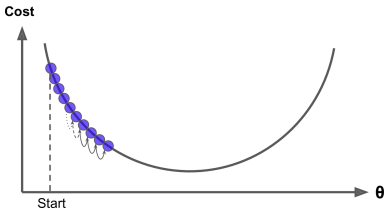


Figure 3.10: SGD with momentum[Gro17]

The author Aurélien Géron[Gro17] claims that instead of the constant learning rate if the experiment starts with a high learning rate, and then we reduce it once it starts to stagnate, then a good solution could be reached fast. There are several strategies to achieve these phenomena of reducing the learning rate during the model training, and they are called learning rate (LR) schedulers.

There are four most common types of LR according to the author[Gro17]:-

1. Predetermined piecewise constant learning rate — This type of scheduler set the learning rate, for example, $\eta_0 = 0.1$ and after fixed number of epochs e.g., 75, the new learning rate will be $\eta_1 = 0.001$. This is easy to understand, but require experimentation to determine at which epoch the learning rate should be changed and by how much.

2. Performance scheduling — It measures the validation error every after every N step, and when the error is stagnated, then it decreases the learning rate by the pre factor λ.

3. Exponential scheduling — The learning rate is calculated by the function $\eta(t) = \eta_0 10^{-t/r}$. This scheduler requires providing $\eta andr$. At every r step, the learning rate will drop by the factor of 10.

4. Power scheduling — The learning rate is the function of $\eta(t) = \eta_0(1 + t/r)^{-c}$ where c is the hyperparameter and usually set to 1. As compared to Exponential scheduling, the learning rate drops very quickly.

ConvNet Configuration						
Experiment	A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input(224 x 224 RGB image						
conv3-64	conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool						
conv3-128	conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool						
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool						
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 **conv3-512** **conv3-512**
maxpool	maxpool					
FC-4096	FC-4096					
	FC-4096					
	FC-4096					
	soft-max					

Table 3.1: **ConvNet** configurations from Column A to E[SZ15]. Experiment column shows the configuration of this experiment. The depth of the configuration increases from left to right as more layers are added. The added layers are shown in bold

4 Experimental Setup

This section will describe the experimental setup for this thesis and will discuss in detail the design decisions made during the designing of the experiment. This section will also include the motivation behind the current setup and how it has enabled us to record the variance in the accuracy of the neural network across different CPUs, GPUs and driver versions.

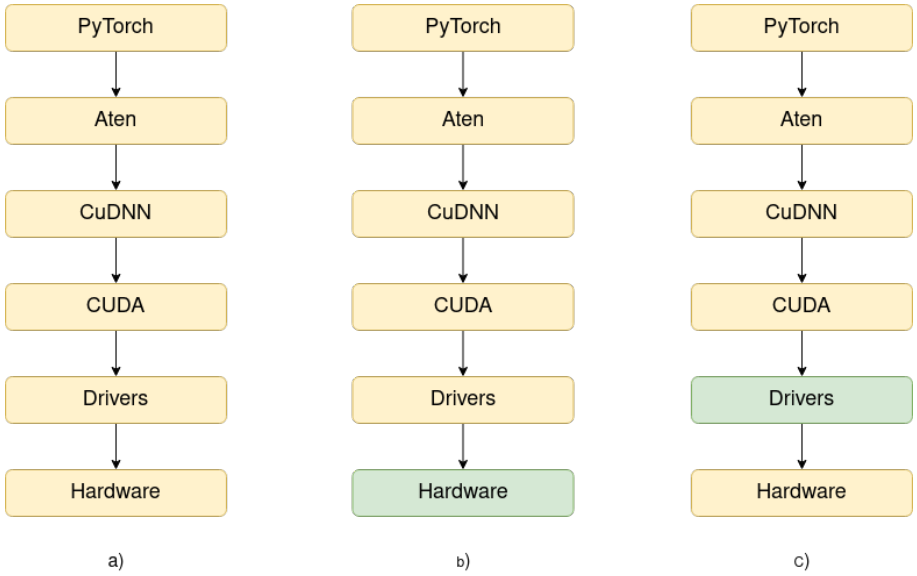


Figure 4.1: Experimental Setup.

a) Software packages used on the hardware in the experiment.

b) Only hardware changes (CPU/GPU) to record the effect of change in hardware on accuracy of Neural Network.

c) Only GPU Driver version changes, all other blocks remains the same to record the effect of GPU driver versions on accuracy of Neural Network.

Figure 4.1 showcase the code flow of the experiment. The yellow blocks showcase the static block, and the green coloured block to showcase the block that is changed for that

run.

The flow of the code as shown in Figure 4.1 a) will be to create CNN in PyTorch[1] using CuDNN operations to leverage GPU performance. The CuDNN library communicates with the CUDA library to pass operations to GPU driver APIs to perform operations on the hardware. This is the basic flow of the code, and ideally, it should give the same accuracy score when executed on all the machines. However, we have discussed this in the subsection 1.1.1 that the CPUs and GPUs have different nature of code execution; CPU being sequential in nature, therefore, more deterministic and GPU being parallel in the execution of threads; therefore, the exact order of thread execution is not deterministic. This motivated me to test the effect of different CPUs have on the accuracy of our network. Figure 4.1 b) shows the hardware as the green block representing the model is executed on different CPUs and different GPUs.

The list of CPUs used in this experiment are as follows:-

1. AMD Ryzen 5 1600
2. i5 7^{th} generation 7200U
3. ebro - AMD Opteron(tm) Processor 6344
4. zaire - AMD Opteron(tm) Processor 6344
5. oberyn - Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz

This list is comprised of CPU from the personal computers and for the CPU instances provided by the University of Passau.

The list of the GPUs used in this experiment are as follows:-

1. NVIDIA GeForce GTX 1060 – 6 GB
2. NVIDIA GeForce 940MX — 2 GB

This list consists of the GPU available from personal computers.

Figure 4.1 b) shows that the *Drivers* are changed in the code flow as we have seen in the subsection 1.1.2 change in the drivers caused a significant change in the results of the experiment. This provided the motivation to look into the variance caused in

[1] `https://pytorch.org`

the accuracy of the model by different driver versions. Selection of driver version was difficult because of numerous intermediate and beta versions available, we have chosen three stable versions from this year to test our hypothesis, and those versions are as follows:-

1. Drivers 460.91.03 - The latest driver version
2. 450.102.04 - The earliest stable driver version from this year i.e., 2021
3. 460.67 - Intermediate driver version

4.1 Image classification model

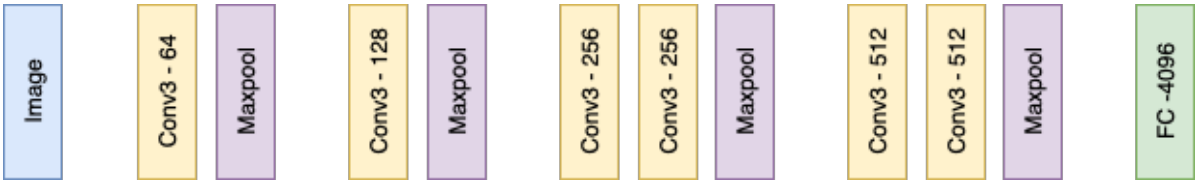


Figure 4.2: VGG 11 based image classification model

This is the actual network that is being used and trained in this experiment. This network is based on the VGG-11 architecture and is being adapted to train on a small and less complex dataset for the experiment, i.e., the CIFAR-10. The main difference between our network and the VGG-11 is in the number of Fully connected layers, from 3 layers in the original paper to one in our experiment. This downsizing is done because the dataset is not complex. It has only ten classes, and therefore lesser fully connected layers will give us an advantage on better runtime while had little impact on the accuracy of the model. As the focus of the thesis is to check the variance of one more and not develop a better algorithm with higher accuracy, saving execution times was more important for us. A single run of the experiment on a machine consists of training on five different seed values and three different optimizers. Training on a CPU will take approximately 9.5 days if the CPU is available 100%, which is often not possible at the university cluster. The time is calculated by running the experiment on the AMD Ryzen 5 1600. This is just for indication; actual time may differ.

And training on a GPU takes approximately 15 hours, and this is also when the CPU is available 100%. The time is calculated by running the experiment on the Nvidia GeForce 1060 6 GB. This is just for indication; actual time may differ.

For the experiment, we set the seed to be constant, made sure the dataset is the same and not shuffled, and the environment has all the same packages with identical versions.

There is also a possibility to use the deterministic flag in PyTorch. Still, we did not use that because, in real life, researchers do not use it. We wanted to run our experiment as close to real-life experiments as possible, of course, putting required constraints like constant seed value, dataset in NumPy arrays, same python packages across systems to enable scientific study.

4.1.1 Determining gamma values

In order to determine the optimal value of various LR scheduler parameters, different side experiments were conducted. Table 4.1 and Table 4.2 show the average accuracy for different gamma values at *milestones* values [50,100] and [50,75,100,125] respectively. As observed from the above-mentioned tables, *gamma*=0.5 and *milestone*=[50,100] gives better result for learning rate 0.01.

Number of Trail run	Before Training	After Training	Gamma	Average Accuracy
1	11.77%	82.85%		
2	11.77%	82.37%	0.4	82.63%
3	11.77%	82.69%		
1	11.77%	82.69%		
2	11.77%	82.88%	**0.5**	**82.69%**
3	11.77%	82.51%		
1	11.77%	82.32%		
2	11.77%	82.88%	0.6	82.41%
3	11.77%	82.51%		

Table 4.1: Accuracy table for different values of Gamma values. Milestone values = [50, 100]

Number of Trail run	Before Training	After Training	Gamma	Average Accuracy
1	11.77%	82.85%		
2	11.77%	82.35%	**0.4**	**82.47%**
3	11.77%	82.48%		
1	11.77%	82.67%		
2	11.77%	82.16%	0.5	82.45%
3	11.77%	82.54%		
1	11.77%	82.91%		
2	11.77%	82.61%	0.6	82.07%
3	11.77%	82.71%		

Table 4.2: Accuracy table for different values of Gamma values. Milestone values = [50,75,100,125]

4.1.2 Determining learning rate

The optimal learning rate should be determined for quick convergence of the network, therefore after performing the series of tests on different learning rates as shown in

Figure 4.7.

Figure 4.3: Learning rate 0.4

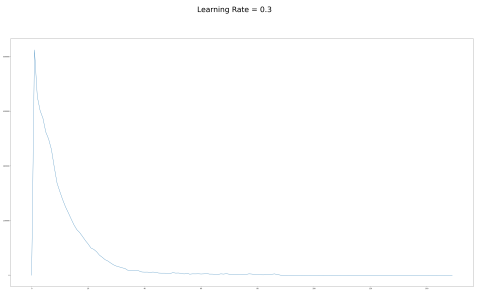


Figure 4.4: Learning rate 0.3

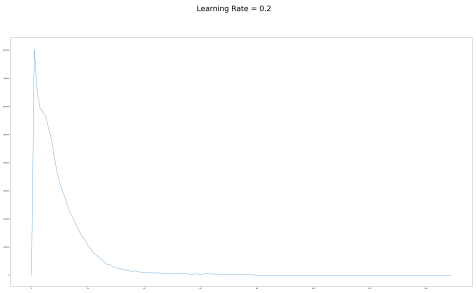


Figure 4.5: Learning rate 0.2

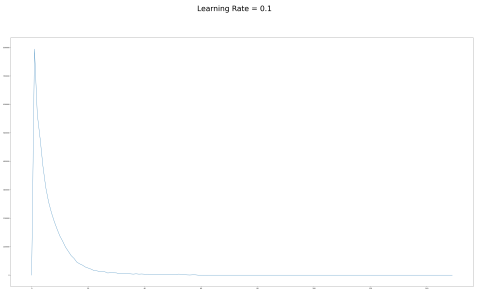


Figure 4.6: Learning rate 0.1

Figure 4.7: Different LR schedulers results; X-axis epochs; Y-axis error value

For the learning rate value of 0.1 the graphs converge smoothly as represented by Figure 4.6 and on decreasing the value of the learning rate, no considerable difference in error value was observed therefore, LR=0.1 was chosen for the experiment. For a complete list of LR values and figures, please refer to appendix section C.1.

4.1.3 Determining milestone values

From the previous experiments, we have concluded that *Learning Rate*=0.1 is optimal for our experiment. The next step is to determine the optimal *milestone* value for the experiment by experiment.

Upon closer inspection of the Figure 4.8 eliminate the first 30 epochs of turbulence, and after 30 epochs in the graph and 50 epochs in the actual count, we observe that the

learning rate is becoming stagnate; therefore, the milestone [50,75,100,125] is selected for the experiment.

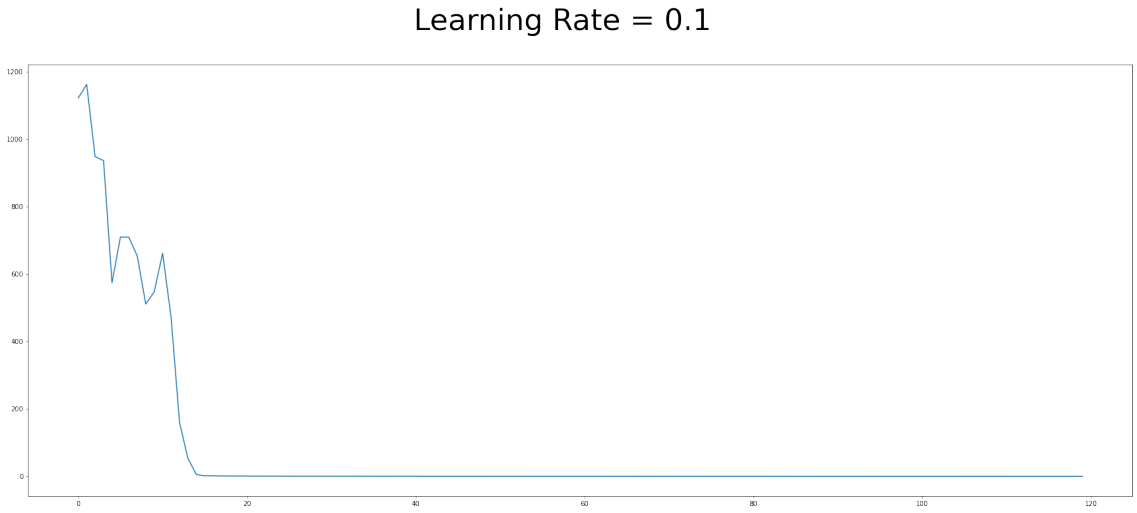


Figure 4.8: LR=0.1 from 30-150 epochs

Learning Rate	Milestones	Gamma	Accuracy
0.1	50,100	0.5	83.73%
0.1	50,75,100,125	0.5	84.40%
0.1	50,75,100,125	0.4	83.73%
0.1	50,75,100,125	0.8	84.13%

Table 4.3: Accuracy of network with Adadelta optimizer

4.1.4 Determining Momentum value

In the previous experiments, we determined the optimal learning rate 0.1 and optimal milestone values to be [50,75,100,125]. In this experiment, the idea is to determine the momentum value for the SGD and NAG optimizer. The gamma value is taken to be 0.8 for the experiment based on these value experiment was run, and Table 4.4 compiles the results. As observed from the table momentum value of 0.5 gives the highest accuracy; therefore momentum value of 0.5 is adopted for the experiment.

Learning Rate	Milestones	Gamma	Momentum Value	Accuracy
0.1	50,75,100,125	0.8	0.9	81.82%
0.1	50,75,100,125	0.8	0.3	83.57%
0.1	50,75,100,125	0.8	0.4	83.78%
0.1	50,75,100,125	0.8	0.5	**84.40%**

Table 4.4: Accuracy of network with SGD optimizer

4.1.5 Dataset

The batch size of 64 gave an accuracy of 84.13%, whereas the batch size of 32 provided an accuracy of 83.13%. Therefore, for the experiment 64 batch size is taken. The number of the image in the training set is 50,000 and diving them into batched means 781 complete batches or 49984 images. Similarly, for the test set, there are 10,000 images, and for the batch size of 64, there are in total 156 complete batches or 9984 images.

4.1.6 Ensuring all the environments are same

In order to make the environment the same on all the machines, a *compare.py* script is developed as part of the thesis. This script checks all the packages required for the experiment. If packages are unavailable, then they are installed; if packages are installed and have a different version, then the correct version is installed after informing the user. Finally, if there are extra packages installed, they are removed after taking consent for an individual from the user. A bash script *start.sh* is also developed as part of the thesis to facilitate the execution of the experiment on different machines.

5 Results

In this chapter, we present the results achieved after executing the experiments discussed in chapter 4. First, we will present the results from the baseline experiment logistic map, and later we present the results from the image classification experiment. Finally, we will show how we made the model reproducible and also presents its advantages and disadvantages.

5.1 Logistic Map

In this section, we will discuss the results of logistic map experiment when executed on *Different CPUs*; *Different GPUs*; and *Different graphic driver versions*. For simplicity and readability, tables with identical results are attached in the appendix for further reference.

5.1.1 Different CPUs

Logistic map was executed for 2000 steps on four different CPUs and five different systems and as shown in Table 5.1, Table B.1, Table 5.3, Table B.2 and Table B.3. The four different CPUs are as follows:-

1. AMD Opteron(tm) Processor 6344 @ 2.6GHz[1]
2. Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz[2]

[1] https://www.amd.com/en/products/cpu/6344
[2] https://ark.intel.com/content/www/us/en/ark/products/186604/intel-core-i7-9700k-processor-12m-cache-up-to-4-90-ghz.html

3. Intel(R) Core(TM) i5-7200U CPU @ 2.50 GHz[3]

4. AMD Ryzen(TM) 5 1600 Processor@3.2GHz[4]

and the results all the four different CPUs came out to be identical down to the 16^{th} decimal place. The table Table 5.1 showcase the results received after 2000 steps on AMD Opteron Processor 6344, which were identical across all other systems. The AMD Opteron Processor 6344, consisting of 12 cores and 12 threads, launched in 2012.

In the table, column *CPU* defines the name of the CPU on which the logistic map was executed. As the title suggests, *Seed value* defines the seed value used to execute the experiment. For our investigation, we have five different seed values, randomly selected between 0 and 10000.

X_0 defines the initial value of the X. r is growth parameter. X_{2000} displays the value of X at 2000^{th} iteration for the given seed value. As mentioned earlier, the value of X_{2000} is identical across all seeds on the CPU. This observation will be useful to compare the results from other CPU results in the upcoming paragraphs.

CPU	Seed value	X_0	r	Steps	X_{2000}
	7184				0.2772458722770831
AMD	13474				0.2772458722770831
Opteron	32889	0.4	3.7	2000	0.2772458722770831
6344	56427				0.2772458722770831
	59667				0.2772458722770831

Table 5.1: AMD Opteron 6344 — Logistic map

Table 5.2 combines the results from all the four different processors. The results were identical; therefore, we have combined them together in one table. The values are rounded up to 6 decimal places due to space constraints. As we can observe from the table, that logistic map resulted in the same value of X_{2000} after 2000 epochs on four different CPUs; therefore, we can infer from our experiment that there is no effect of CPU on the results of the logistic map.

[3] https://ark.intel.com/content/www/us/en/ark/products/95443/intel-core-i5-7200u-processor-3m-cache-up-to-3-10-ghz.html

[4] https://www.amd.com/en/products/cpu/amd-ryzen-5-1600

Seed value	Steps	AMD Opteron 6344	Intel Core i7 7200U	Intel Core i5 7200U	AMD Ryzen 5 1600
7184		0.277246	0.277246	0.277246	0.277246
13474		0.277246	0.277246	0.277246	0.277246
32889	2000	0.277246	0.277246	0.277246	0.277246
56427		0.277246	0.277246	0.277246	0.277246
59667		0.277246	0.277246	0.277246	0.277246

The results are rounded-off to six decimal places due to space constraint. Full tables are available in appendix

Table 5.2: Logistic map - Results from all the CPUs

Individual tables used to create Table 5.2 are Table 5.1, Table B.1, Table B.2, and, Table B.3.

From the previous table, we have established that there is no effect of different CPUs on the logistic map. In the next paragraph, we tested if there is any variance when we run the experiment on the same machine multiple times.

CPU	Run	Seed value	X_0	r	Steps	Run 1 X_{2000}	Run 2 X_{2000}
Intel i5	1	7184	0.4	3.7	2000	0.2772458722770831	0.2772458722770831
		13474				0.2772458722770831	0.2772458722770831
		32889				0.2772458722770831	0.2772458722770831
		56427				0.2772458722770831	0.2772458722770831
		59667				0.2772458722770831	0.2772458722770831

Table 5.3: Intel i5-7200U — Logistic map

The Table 5.3 showcase the logistic map results observed after 2000 epochs on a mobile processor from the Intel. The Intel i5-7200U is a two core and four-thread processor usually used in mobile computing devices like a laptop. It showcases that when the logistic map experiment is executed on the same CPU twice, it gives identical results. From this behaviour, we infer that the results will be identical for all the subsequent runs on the CPU. The Table 5.3 results are also identical to the results achieved on other CPUs as shown in Table 5.2.

After performing the experiment on the four different types of CPUs, there was an instance available at the University of Passau Data Science chair, which use AMD Opteron Processor 6344. Since we have already executed the code on this processor earlier in Table 5.1 it was not required to run it again. As the experiment takes considerable less time to execute and the CPU instance was available, we were curious to check further how the logistic map perform when executed on the same type of processor but a different machine. As expected, the results were identical to the other CPU results, as shown in Table 5.2. The results from the AMD Opteron Processor 6344 is available in Table B.3.

Table 5.4 demonstrate the key takeaway from this logistic map experiment, running on CPU. The average accuracy is identical across all the CPUs and therefore have a variance of 0%.

Seed value	$\mu(X_{2000})$	σ
7184	0.2772458722770831	0.0
13474	0.2772458722770831	0.0
32889	0.2772458722770831	0.0
56427	0.2772458722770831	0.0
59667	0.2772458722770831	0.0

Table 5.4: Logistic Map average value across runs and their variance

5.1.2 Different GPUs

In this section, we present the results of the logistic map from different GPUs Table 5.5 compiles these results. The logistic map was first executed on Nvidia GeForce 1060 graphic card, results are under column name $X_{2000}^{GPU^1}$. Then the experiment was executed on Nvidia GeForce 940 MX, and the results are under column name $X_{2000}^{GPU^2}$. Both the columns have identical values; therefore we can infer from our experiment that there is no effect of different GPUs on the logistic map.

Seed value	X_0	r	Steps	$X_{2000}^{GPU^1}$	$X_{2000}^{GPU^2}$	X_{2000}^{CPU}
7184				0.29076236	0.29076236	0.27724587
13474				0.29076236	0.29076236	0.27724587
32889	0.4	3.7	2000	0.29076236	0.29076236	0.27724587
56427				0.29076236	0.29076236	0.27724587
59667				0.29076236	0.29076236	0.27724587

1: Nvidia GeForce 1060 2: Nvidia GeForce GeForce 940MX

Table 5.5: Nvidia GeForce 1060 vs Nvidia GeForce 940 MX vs CPU results — Logistic map

The value of the X_{2000} of GPU or X_{2000}^{GPU} is higher than the X_{2000} of CPU or X_{2000}^{CPU}. As discussed earlier in the chapter 1 about the approximation error in floating-point numbers, we can confirm this phenomenon from our results in Table 5.5. Figure 5.1 demonstrates the divergence in value of X_{n+1}^{CPU} and, X_{n+1}^{GPU} where red dotted line denotes CPU, and solid blue line denoted GPU. It is to be noted that in the figure, only first 100 steps are shows due to the space constraint showing 2000 epochs was not possible. Despite the limitation of space, the graphs shows the divergence between CPU and GPU value start at epoch 38. The major deviations can be seen between 58 – 63 epochs and between 88 – 98 epochs.

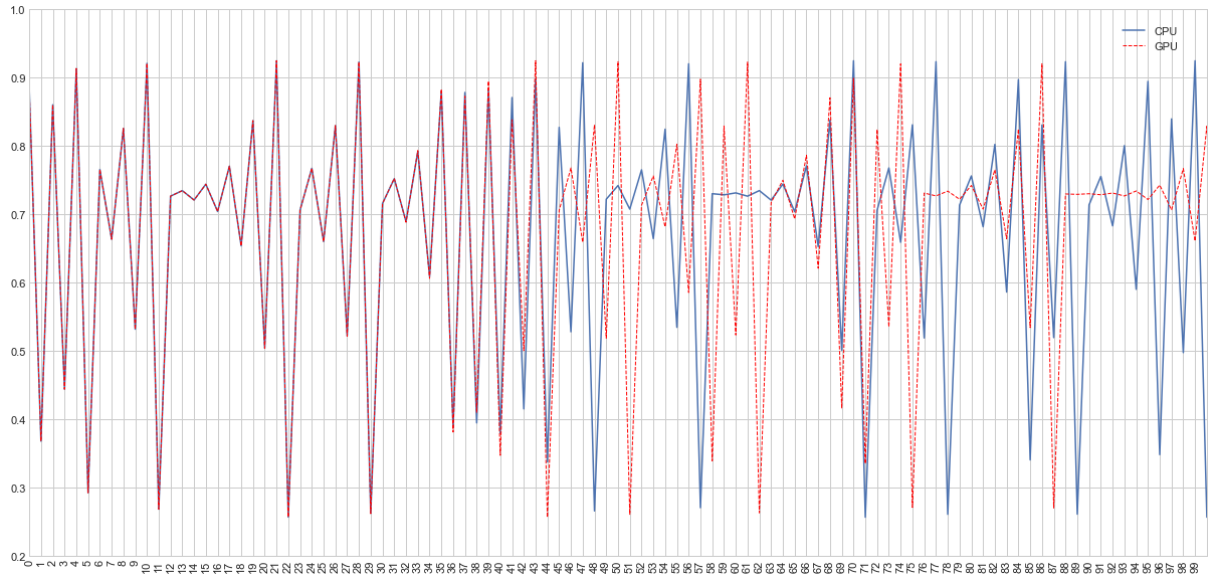


Figure 5.1: Comparison of first 100 epochs of CPU and GPU during logistic map

5.1.3 Different GPU driver versions

In this section, we present the results of the logistic map obtained after executing the experiment on different graphic driver versions. The three driver version chosen for the experiment are:-

1. 460.91.03 — The latest stable driver version at the time of the experiment.
2. 460.67 — Previous stable driver version.
3. 450.142.00 — The earlier stable driver version from the start of the current year, i.e., 2021.

System	Driver version	Seed value	X_0	r	Epochs	X_{2000}^{GPU}
Nvidia GeForce 1060	460.91.03	7184	0.4	3.7	2000	0.29076236
		13474				0.29076236
		32889				0.29076236
		56427				0.29076236
		59667				0.29076236

Table 5.6: Nvidia GeForce 1060 — Logistic map

First, we executed the experiment with driver version *460.91.03* and the results are tabulated in Table 5.6. Then we executed the experiment on the driver version *460.67* and *450.142.00* results tabulated in Table B.5 and Table B.6 respectively.

All the three driver version shows the exact same value of, X_{2000}^{GPU}, therefore, we could say that the change in driver version has no effect on the logistic map. This showcase that even though there is an innate approximation error in the GPUs, this approximation error remains constant throughout the driver version we tested.

Seed value	$\mu(X_{2000}^{GPU})$	σ
7184	0.29076236	0.0
13474	0.29076236	0.0
32889	0.29076236	0.0
56427	0.29076236	0.0
59667	0.29076236	0.0

Table 5.7: Logistic map average value and variance across different driver GPUs and driver versions

The logistic map tensor used until now is of *float* type or half-precision-floating-point number. We tested the output of the experiment with *double* type tensor or in double-precision-floating-point number. The results are compiled in the next section.

5.1.4 GPU runs with double precision model

In the subsection 5.1.3 and subsection 5.1.2, the tensor to store the values of X_{2000} was of type $dtype = torch.float32$ i.e., single-precision-floating-point number and for this section the data type is $dtype = torch.double$ i.e., double-precision-floating-point number. The advantage of using the double-precision-floating-point format is the X_{2000}^{CPU} and X_{2000}^{GPU} value after 2000 steps came out to be the same, i.e., 0.2772458722770831. The Table 5.8 compare the *float* and *double* values of GPU with the CPU results.

GPU	Seed value	Float:X_{2000}^{GPU}	Double: X_{2000}^{GPU}	X_{2000}^{CPU}
Nvidia GeForce 1060 6GB	7184	0.29076236	0.2772458722770831	0.2772458722770831
	13474	0.29076236	0.2772458722770831	0.2772458722770831
	32889	0.29076236	0.2772458722770831	0.2772458722770831
	56427	0.29076236	0.2772458722770831	0.2772458722770831
	59667	0.29076236	0.2772458722770831	0.2772458722770831

Table 5.8: Comparison of float and double tensor results of GPU with the CPU.

From Table 5.8 we observe that using *float* tensor in our experiment, we get the approximation error of about 4.86% on GPU. This approximation error could be removed if we use tensor of type *double*, but the time taken to compute the result increases significantly.

In the next section, we discuss the image classification results when executed on CPUs, GPUs and different driver versions. We also discuss the float and double image classification model, as we did for the logistic map in this section.

5.2 Image classification

In this section, we will present the results of the image classification experiment and test them against the pre-established baseline from the logistic map experiment. Baselines from the Logistic map experiment are:-

1. Change in CPU does not affect the results.
2. Change in GPU does not affect the results.
3. There will be a change between the results of CPU and GPU.
4. Change in driver versions does not affect the results.
5. If the GPU is utilizing double-precision-floating-point format, then there is no difference in the value calculated by CPU and GPU.

In the upcoming subsections, we will share details of the experiment results and corresponding findings.

5.2.1 Different CPUs

First, we executed the experiment on Intel core i7-9700K CPU available at the University of Passau, and results are tabulated in Table 5.9. The model trained in just over 23 days. Due to resource sharing with other users, it took longer than the estimated nine days to complete training.

In the upcoming sections, we will compare this table with other tables to find out if there is any variance in the accuracy of the model when executed on different CPUs.

CPU	CPU type	Seed value	Optimizer	Initial	Final
Intel Core i7-9700K	x86_64	7184	Adadelta	10.15%	83.65%
			SGD	10.15%	82.91%
			NAG	10.15%	83.37%
		13474	Adadelta	9.60%	83.88%
			SGD	9.60%	84.44%
			NAG	9.60%	83.96%
		32889	Adadelta	9.21%	83.14%
			SGD	9.21%	84.01%
			NAG	9.21%	83.85%
		56427	Adadelta	9.93%	83.56%
			SGD	9.93%	83.84%
			NAG	9.93%	83.80%
		59667	Adadelta	10.44%	84.11%
			SGD	10.44%	83.88%
			NAG	10.44%	84.05%

Table updated on 13 August 2021, 2147 hours CEST
Experiment started on 23 July 2021 1600 hours CEST

Table 5.9: Intel Core i7-9700K — Image classification

Table 5.10 compiles the results from the AMD Opteron 6344 CPU available at the University of Passau. The CPU is from a different vendor, i.e., AMD, and the CPU is an Opteron 6344. On comparing this table with the previous Table 5.9 we can already see the change in the accuracy of the model when compared against each seed and optimizer.

This behaviour already shows deviation from our first baseline that CPU does not affect the result. We will see how the results hold up when we test this on other CPUs.

Note:- As of writing this report, the experiment is still running on two CPU instances since 23 July 2021 and, therefore, will keep the Table 5.10 and Table 5.11 updated as the results will be coming in.

CPU	CPU type	Seed value	Optimizer	Initial	Final
AMD Opteron 6344	x86_64	7184	Adadelta	10.15%	83.94%
			SGD	10.15%	83.01%
			NAG	10.15%	83.87%
		13474	Adadelta	9.60%	83.92%
			SGD	9.60%	83.88%
			NAG	9.60%	-
		32889	Adadelta	9.21%	-
			SGD	-	-
			NAG	-	-
		56427	Adadelta	9.93%	-
			SGD	-	-
			NAG	-	-
		59667	Adadelta	-	-
			SGD	-	-
			NAG	-	-

Table updated on 10 August 2021, 2147 hours CEST
Experiment started on 23 July 2021 1630 hours CEST

Table 5.10: AMD Opteron 6344 — Image classification on CPU

System	CPU type	Seed value	Optimizer	Initial	Final
AMD Opteron 6344	x86_64	7184	Adadelta	10.15%	83.63%
			SGD	10.15%	83.06%
			NAG	10.15%	83.68%
		13474	Adadelta	9.60%	83.86%
			SGD	9.60%	84.32%
			NAG	9.60%	83.97%
		32889	Adadelta	9.21%	84.07%
			SGD	9.21%	84.27%
			NAG	9.21%	84.05%
		56427	Adadelta	9.93%	83.80%
			SGD	9.93%	84.10%
			NAG	9.93%	84.07%
		59667	Adadelta	10.44%	83.97%
			SGD	10.44%	-
			NAG	-	-

Last updated: 04 September 2021, 1750 hours CEST; Experiment started on 23 July 2021 1700 hours CEST

Table 5.11: AMD Opteron 6344 — Image classification on CPU

The Table 5.2.1 compiles all the results and showcase the variance in the accuracy of the image classification model when compared across the different CPUs. As we can see, the variance is non-zero from the table. Therefore, we can conclude that there is an effect of CPUs on the accuracy of the image classification model, unlike our baseline example logistic map.

Seed value	Optimizer	$\mu(accuracy)$	σ
7184	Adadelta	83.74%	0.02007
	SGD	82.99%	0.00250
	NAG	83.64%	0.04247
13474	Adadelta	83.89%	0.00062
	SGD	84.21%	0.05796
	NAG	83.97%	0.00003
32889	Adadelta	83.61%	0.21623
	SGD	84.14%	0.016899
	NAG	83.95%	0.010000
56427	Adadelta	83.68%	0.01439
	SGD	83.97%	0.01689
	NAG	83.94%	0.01822
59667	Adadelta	84.04%	0.00490
	SGD	-	-
	NAG	-	-

Table 5.12: Average accuracy and variance on CPU for image classification.
Table updated on 05 September 2021, experiment is still executing on CPU instances

5.2.2 Different GPUs

In this section, we present the results obtained after executing the experiment on different GPUs. The two available Nvidia GPUs are GeForce 1060 6 GB and Nvidia GeForce 940 MX 2 GB.

Logistic map behaviour on the different GPUs forms the baseline for this experiment. The logistic map did not show variation in results when executed on different GPUs. In the upcoming tables, we will investigate and see if the baseline established by the logistic map will hold true for our image classification model. In the Table 5.13 and Table 5.14 represents the results of the experiment when executed on different GPUs.

GPU name	D. version	Total memory	Seed value	Optimizer	Initial	Final
GeForce GTX 1060	460.91.03	6 GB	7184	Adadelta	10.15%	83.94%
				SGD	10.15%	82.67%
				NAG	10.15%	83.46%
			13474	Adadelta	9.60%	83.93%
				SGD	9.60%	84.06%
				NAG	9.60%	84.06%
			32889	Adadelta	9.21%	84.06%
				SGD	9.21%	83.54%
				NAG	9.21%	84.07%
			56427	Adadelta	9.93%	84.07%
				SGD	9.93%	83.68%
				NAG	9.93%	84.10%
			59667	Adadelta	10.44%	84.01%
				SGD	10.44%	84.28%
				NAG	10.44%	83.80%

Table 5.13: 1060 - Accuracy table for GPU instances

GPU name	Driver version	Total memory	Seed value	Optimizer	Initial	Final
Nvidia GeForce 940 MX	460.91.03	2 GB	7184	Adadelta	10.15%	84.11%
			13474	Adadelta	9.60%	84.35%
			32889	Adadelta	9.21%	83.62%
			56427	Adadelta	9.93%	84.05%
			59667	Adadelta	10.44%	84.14%

Table 5.14: Nvidia GeForce 940 MX: Image classification on GPU

5.2.3 Different GPU driver versions

The logistic map showed no sign of variance when executed on different driver versions. In this section, we will present the image classification model results from different driver versions and see how it compares to the logistic map behaviour.

Drivers chosen to execute this experiment are as follows:-

1. 460.91.03 — The latest stable driver version at the time of the experiment.
2. 460.67 — Previous stable driver version.
3. 450.142.00 — The stable driver version at the start of the current year, i.e., 2021.

Table 5.15 shows the image classification model result for the three driver versions discussed earlier.

Seed value	Optimizer	Initial	Final[1]	Final[2]	Final[3]
7184	Adadelta	10.15%	83.94%	83.88%	84.26%
	SGD	10.15%	82.67%	82.16%	82.94%
	NAG	10.15%	83.46%	83.62%	83.50%
13474	Adadelta	9.60%	83.93%	84.00%	84.06%
	SGD	9.60%	84.06%	84.16%	84.16%
	NAG	9.60%	84.06%	83.71%	83.12%
32889	Adadelta	9.21%	84.06%	83.59%	84.06%
	SGD	9.21%	83.54%	84.21%	84.91%
	NAG	9.21%	84.07%	84.33%	83.38%
56427	Adadelta	9.93%	84.07%	83.86%	84.04%
	SGD	9.93%	83.68%	83.97%	84.11%
	NAG	9.93%	84.10%	84.00%	84.13%
59667	Adadelta	10.44%	84.01%	84.14%	84.33%
	SGD	10.44%	84.28%	84.56%	83.92%
	NAG	10.44%	83.80%	83.83%	83.34%

1:- Driver version 460.91.03; 2: Driver version 460.67; 3: Driver version 450.142.00

Table 5.15: GeForce GTX 1060 – 6 GB — Accuracy table for different driver versions

From the table, it is observed that the variance is observed when executing the experiment on different driver versions. This behaviour is different from the logistic map, where there was no effect of driver version on the experiment result.

In Table 5.16 we compile the average accuracy of optimizer per seed and also calculate the variance. This will help us to quantify the variance we observed when running the image classification experiment on different driver versions.

Seed value	Optimizer	μ (accuracy)	σ
7184	Adadelta	84.03%	0.02782
	SGD	82.59%	0.06503
	NAG	83.53%	0.00462
13474	Adadelta	84.00%	0.00282
	SGD	84.13%	0.00222
	NAG	83.63%	0.15047
32889	Adadelta	83.90%	0.04901
	SGD	84.22%	0.31287
	NAG	83.93%	0.16069
56427	Adadelta	83.99%	0.00860
	SGD	83.92%	0.32067
	NAG	84.08%	0.00309
59667	Adadelta	84.16%	0.01727
	SGD	84.25%	0.06862
	NAG	83.66	0.05029

Table 5.16: Average accuracy and variance on GPU for Image classification

To test the variance between the two runs on the same system, we executed the experiment again on GeForce GTX 1060 to find out that there is variance even between the runs on the same machine. The Table 5.15 and Table B.7 shows the difference in accuracy.

5.2.4 GPU runs with double precision model

The non-deterministic runs on the same machine lead us to investigate the cause of the variance because even after setting the seed, creating NumPy arrays of the dataset and recreating the same environment on a different machine, we did not expect this non-deterministic. We recreated the model on the system, this time using double data type. The model became deterministic on the same system, but it has a major downside that the execution time increased by a factor of 22.

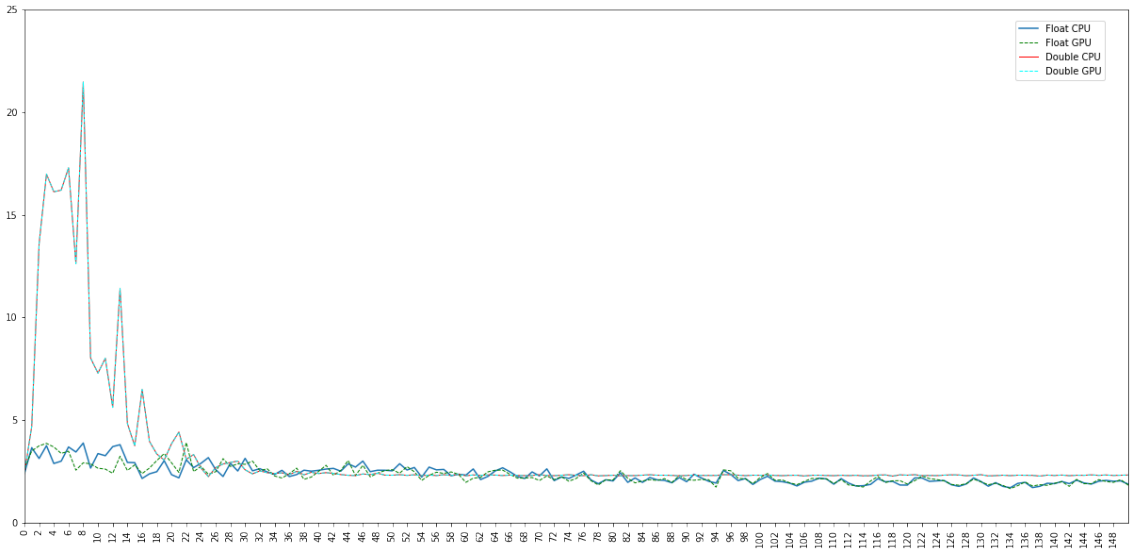


Figure 5.2: Comparison of loss at each step of 1st 150 steps between CPU float, GPU float and GPU double model

Figure 5.2 shows the comparison of the *float* and *double* model on CPU and GPU. The solid line represents the CPU and the dotted line represents the GPU. The figure shows the first 150 steps for both CPU and GPU. From the figure, we can observe that the *double* CPU and GPU lines overlap each other perfectly. Additionally, loss for double model is very high for the first ten steps, and we do not know the exact reason for this high loss.

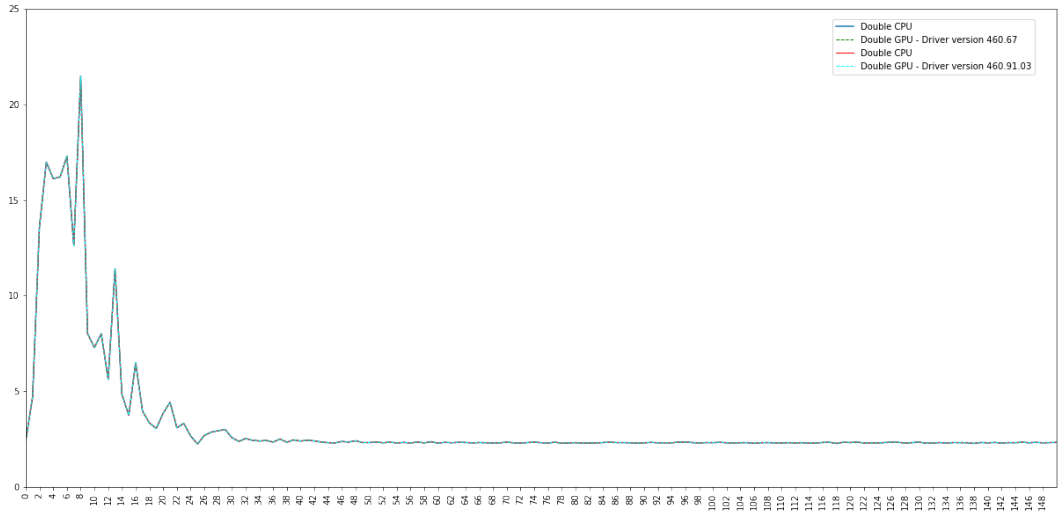


Figure 5.3: Comparison of loss at each step of 1st 150 steps between different driver versions for double precision model

Figure 5.3 shows the comparison of the loss values for the first 150 steps of an image classification model. From the figure, it seems there is no effect of driver versions on the output of the image classification model with a double-precision-floating-point system. When we look at the individual loss at the final step of every epoch, 1.9654094929675220 9818 and 1.9654094929731358 2989 for driver version 460.91.03 and 460.67, respectively. There is a slight change from 11^{th} decimal place in the loss value, which might snowball into a large number over $100s$ of epochs. Therefore, we would like to test the experiment for 150 epochs and see if there is any change in the final accuracy of the model. $460.91 - 1.9654094929675220 9818460.67 - 1.9654094929731358 2989$ We tested the image classification model with a double-precision-floating-point format for 150 epochs on two different GPUs, namely:-

1. Nvidia GeForce GTX 2080 Ti
2. Nvidia Tesla K80

GPU	Seed value	Optimizer	Accuracy
GTX 2080 Ti	7184	SGD	80.89%
Tesla K80	7184	SGD	81.09%

Table 5.17: Accuracy of image classification model on different GPU

From Table 5.17 we can infer that the accuracy of the image classification model with double-precision-floating-point number varies with change in GPUs. Further testing is required with different optimizers and seed values. We could not do that due to time constraints. The time constraint was GPU instances were available to run the experiment on 03 September 2021 at 1213 CEST, and the submission deadline was on 06 September 2021.

6 Discussion

In this chapter, we discuss our findings from each experiment. We will also answer each research question in this section based on the results achieved. We set out to answer the following three fundamental questions in this thesis work, and the three research questions are as follows:-

6.1 How much does the accuracy vary with the difference in hardware (CPU & GPU)?

We will present our findings from the logistic map and image classification model separately for CPU and GPU. We will also present the result of the float and double model in the GPU section.

6.1.1 CPU

For the Logistic map experiment for 2000 steps, the results on all the four different types of CPUs came out to be identical to the 16^{th} decimal place. From our experiment, we can infer that there is no effect of CPU on the performance of the logistic map.

For the Image classification experiment with a half-precision model, the results were different for all the four CPUs. A variance of at least 0.00250 is observed. This proves there is some form of approximation happening in the Image Classification experiment, which is not accounted for. The source of the approximation is not investigated, as it was out of the scope of this thesis work. Nevertheless, this could be a potential extension to the current thesis and could make for a promising future work.

For the Image classification experiment with the double-precision model, the results were deterministic i.e. identical for the same CPU runs, unlike the half-precision model. The experiment on other CPUs is executing, and the results from them will be presented shortly.

6.1.2 GPU

When executed Logistic map experiment on GPU, the results were identical even between the runs, which tell us that even though there is approximation error as we discussed in chapter 1 this approximation is constant between the runs.

However, when it comes to the image classification half-precision-format model, the results were non-identical between GPUs. The model could be deterministic with the use of the double-precision-format model. Using the double-precision-format model, the results of the GPU came out to be consistent between different runs on same GPU. When executed the double-precision-format model on different GPUs variance is observed. The disadvantage of the double-precision model is the huge increase in the execution time. In our experiment, the time to train the image classification model increased by a factor of 22. As an example, if it takes one day to train a model with half-precision, it will take 22 days with double precision.

6.2 How much does the accuracy vary with the difference in GPU driver version?

The change in the driver version had no impact on the logistic map experiment.

All the results from the logistic map half-precision after 2000 epochs came out to be identical to the last decimal place. As mentioned in the previous section, the result of half-precision was different from the result of half-precision on CPU, but among the half-precision GPU results, there was no variance. Therefore, from our experiment, we can infer that the logistic map had no impact of change in the driver version.

When executing the logistic map with double-precision, the result came out to be identical to the results of the CPU.

For the image classification experiment with half-precision, there is a variance observed between the different driver versions.

6.3 How extending experiment tracking software like PyPads to log additional details like GPU driver versions could be useful in the reproducibility of the experiment.

As we have observed, there is a variance when it comes to different CPUs and GPUs, therefore extending the functionality of PyPads to include GPU details has proved to be of significant importance in the reproducibility of experiments.

Although the variance in results is not observed with double-precision, in real life, double-precision is not used often because it increases the cost of training the model. For a small percentage point increase in the accuracy, the cost of training is too high from our experiment; it is 22 times.

Therefore, having a reproducibility experiment will help log necessary details about the CPU, GPU, and other driver versions to set up the environment as close to the author had in his/her research.

6.4 Challenges faced

1. Running time and availability of the resources — It should have taken approximately nine days to run the experiment on the CPUs, however, due to resource sharing among different users, the fastest instance took 23 days, and other instances are still executing the experiment at the time of writing this report.

2. The availability of GPU instances — The availability of the GPU instances was the biggest challenge. I am running the experiment on my local machine and partially on the advisor's machine. The GPU instances from the university were available only three days before final submission of the report, i.e. on 03 September 2021 and final submission is on 06 September 2021.

3. The image classification experiment could not successfully execute on my Nvidia GeForce 940 MX 2GB because graphic card memory was only 2 GB, and the code was crashing after training the model with Adadelta optimizer. Image classification experiments could not be trained for all three optimizers because of memory constraints. The PyTorch takes 800 MB, the model takes 583 MB, and there is only 400 MB memory left; therefore, the system crashes, aborting the experiment.

4. The possibility to change drivers on GPU instances — The second part of the GPU experiment is to change the driver versions. This was a challenge for the infrastructure team, as the docker container does not allow the pass-through of driver versions. The possibility to use Nvidia-docker container[1] was ruled out because it requires a server restart.

5. Resource allocation for different seeds — Executing experiment for five different seeds increased the execution time by five-folds and put additional burden on the resources available, but this was important for the empirical study of the results.

6. Creating the identical environment — It was difficult to create a similar environment on a different machine; with the help of a compare script, this was made possible.

7. Choosing the drivers — The latest driver was chosen, then the earliest stable driver version from the year 2021 was chosen and finally, an intermediate stable driver version was chosen between Jan 2021 and August 2021.

[1] `https://github.com/NVIDIA/nvidia-docker`

7 Conclusion

We developed a new feature within Pypads for logging the GPU details like GPU name, driver version, total memory, UUID, serial number and CUDA version. This helped in the experiment's reproducibility by providing more details of the system environment on which the experiment was executed.

The logistic map had no effect of change in CPU or GPU, or driver versions. The result came out to be identical even to the 16^{th} decimal place for the CPU and to the $8th$ decimal place for the GPU. When *float* tensor was used, the result of GPU was 4.8% higher than that of the CPU. When *double* tensor was used, the result of the GPU was precisely equal to the result of the CPU down to the 16^{th} decimal place.

The image classification model in half-precision-floating-point format observed variance with the change in CPU, GPU, and driver version due to approximation of the half-precision. The variance was observed when the experiment was executed on an identical CPU. Additionally, the variance was also observed between the run on the same machine; therefore, we cannot make the deterministic model with a half-precision-floating-point format.

We also explored the behaviour of the image classification model in a double-precision-floating-point system. There was no variance between the runs on the same machine; therefore, we can infer from our experiment that the model is reproducible on the same machine with the double-precision-floating-point system. We tested the same model on different GPUs and found out there is variance in the accuracy of the model. This means there is the effect of different GPUs on the neural network.

Therefore, from our experiment, we conclude that we cannot make a deterministic image classification model with half-precision-floating-point numbers. Using double-precision-floating-point numbers, we could make the mode reproducible on the same machine, but

this will increase the training time by a factor of 22. When it comes to different GPUs the variance is still there in the accuracy of the image classification model.

A Code

Code is available at zenodo under the link `https://doi.org/10.5281/zenodo.5460861`

B Tables

CPU	CPU type	Seed value	X_0	r	Epochs	X_{n+1}
Intel i7-9700K	x86_64	7184	0.4	3.7	2000	0.2772458722770831
		13474				0.2772458722770831
		32889				0.2772458722770831
		56427				0.2772458722770831
		59667				0.2772458722770831

Table B.1: Intel i7-9700K — table for logistic map

CPU	CPU Architecture	Seed value	X_0	r	Epochs	X_{n+1}
AMD Ryzen 5 - 1600	x86_64	7184	0.4	3.7	2000	0.2772458722770831
		13474				0.2772458722770831
		32889				0.2772458722770831
		56427				0.2772458722770831
		59667				0.2772458722770831

Table B.2: AMD Ryzen 5 1600 — table for logistic map

CPU	CPU type	Seed value	X_0	r	Epochs	X_{n+1}
AMD Opteron 6344	x86_64	7184	0.4	3.7	2000	0.2772458722770831
		13474				0.2772458722770831
		32889				0.2772458722770831
		56427				0.2772458722770831
		59667				0.2772458722770831

Table B.3: AMD Opteron 6344 — table for logistic map

GPU	Run	Driver version	Seed value	X_0	r	Epochs	X_{n+1}
940MX	1	460.91.03	7184	0.4	3.7	2000	0.29076236
			13474				0.29076236
			32889				0.29076236
			56427				0.29076236
			59667				0.29076236
940MX	2	460.91.03	7184	0.4	3.7	2000	0.29076236
			13474				0.29076236
			32889				0.29076236
			56427				0.29076236
			59667				0.29076236

Table B.4: Nvidia GeForce 940MX — Logistic map

GPU	Driver version	Seed value	X_0	r	Epochs	X_{n+1}
Nvidia GeForce 1060 6GB	460.67	7184	0.4	3.7	2000	0.29076236
		13474				0.29076236
		32889				0.29076236
		56427				0.29076236
		59667				0.29076236

Table B.5: Nvidia GeForce 1060 — Accuracy table for Logistic Map

GPU	Driver version	Seed value	X_0	r	Epochs	X_{n+1}
Nvidia GeForce 1060 6GB	450.142.00	7184	0.4	3.7	2000	0.29076236
		13474				0.29076236
		32889				0.29076236
		56427				0.29076236
		59667				0.29076236

Table B.6: Nvidia GeForce 1060 — Accuracy table for Logistic Map

GPU name	Driver version	Total memory	Seed value	Optimizer	Initial	Final
GeForce GTX 1060	450.142.00	6 GB	7184	Adadelta	10.15%	84.26%
				SGD	10.15%	82.94%
				NAG	10.15%	83.58%
			13474	Adadelta	9.60%	84.08%
				SGD	9.60%	83.98%
				NAG	9.60%	83.75%
			32889	Adadelta	9.21%	83.93%
				SGD	9.21%	84.01%
				NAG	9.21%	84.04%
			56427	Adadelta	9.93%	84.21%
				SGD	9.93%	84.12%
				NAG	9.93%	83.90%
			59667	Adadelta	10.44%	84.55%
				SGD	10.44%	84.20%
				NAG	10.44%	83.57%

Table B.7: GeForce GTX 1060 — Accuracy table for GPU instances — Run 2

C Plots

C.1 Learning rate

X — axis represents number of epochs, Y — axis represents error value per epoch

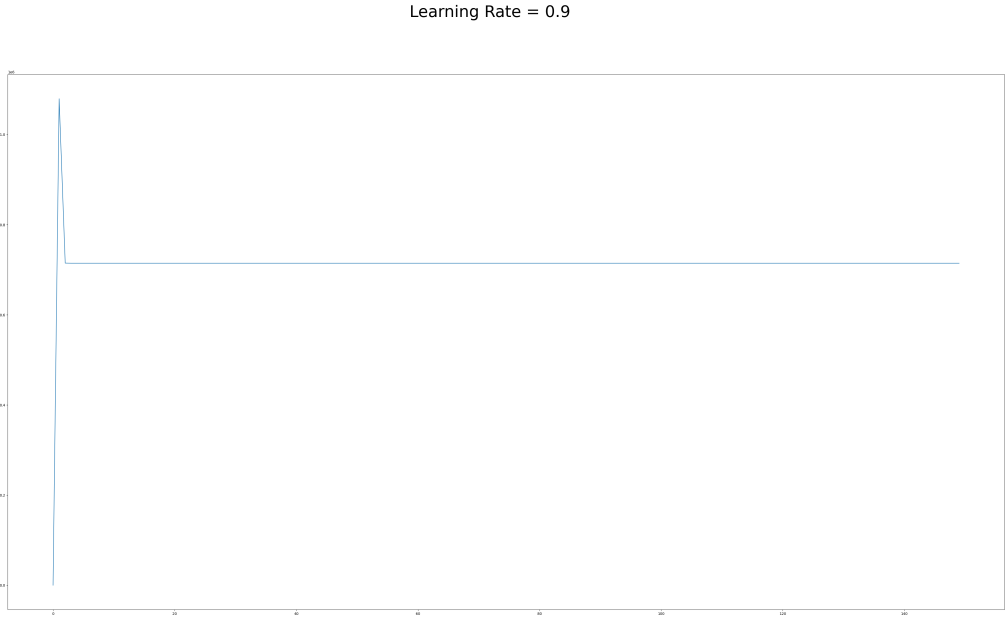


Figure C.1: Learning Rate 0.9

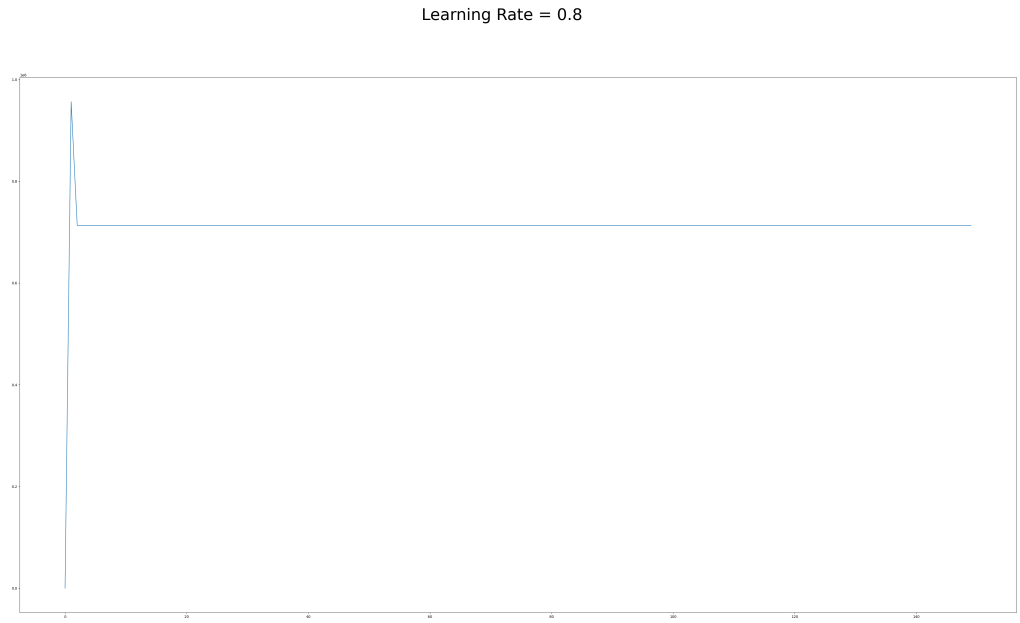


Figure C.2: Learning Rate 0.8

Figure C.3: Learning Rate 0.7

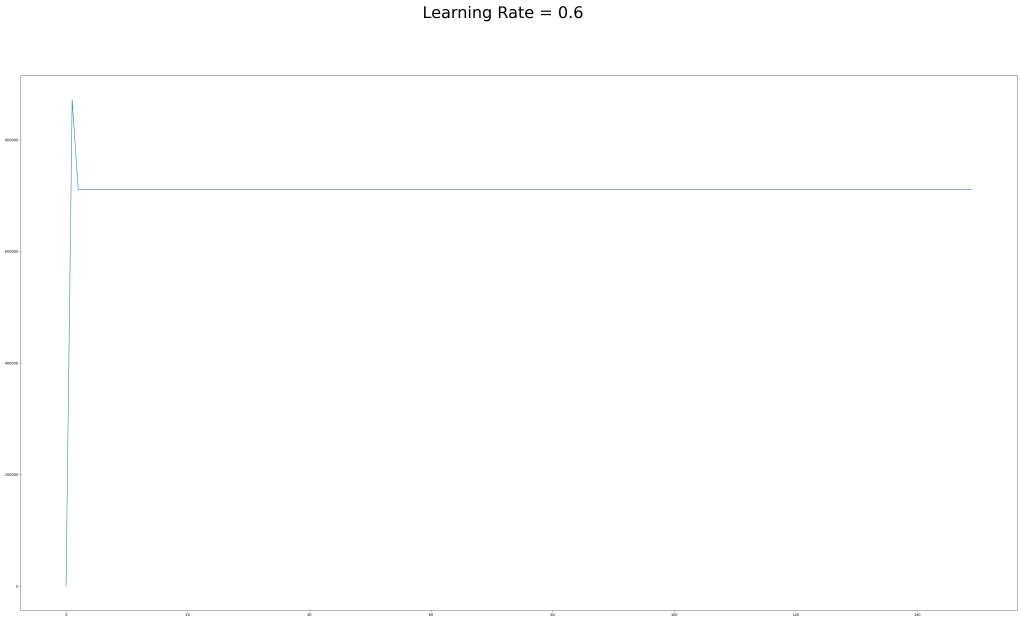


Figure C.4: Learning Rate 0.6

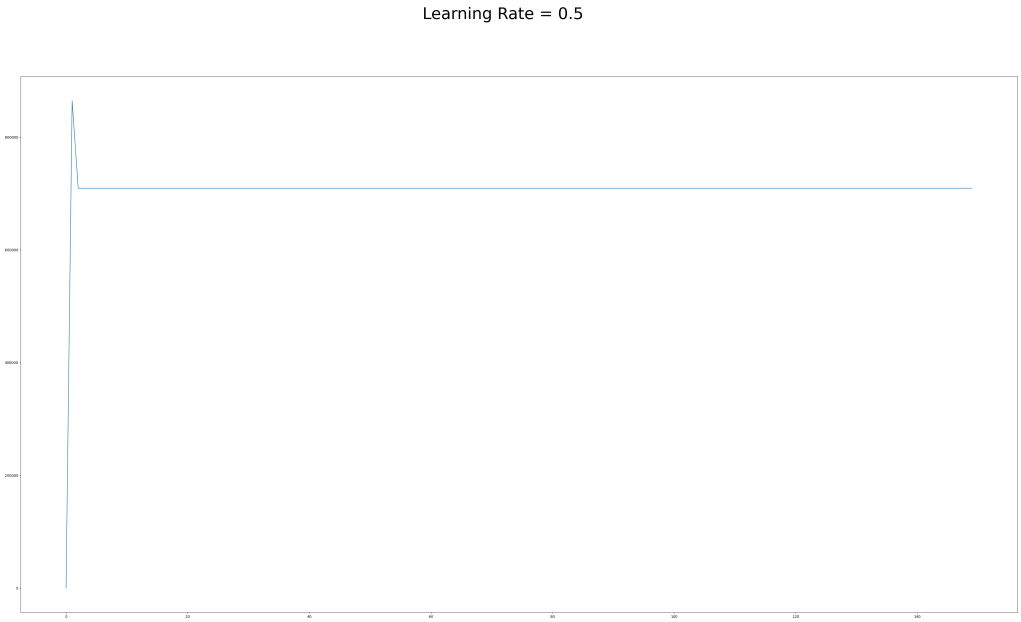


Figure C.5: Learning Rate 0.5

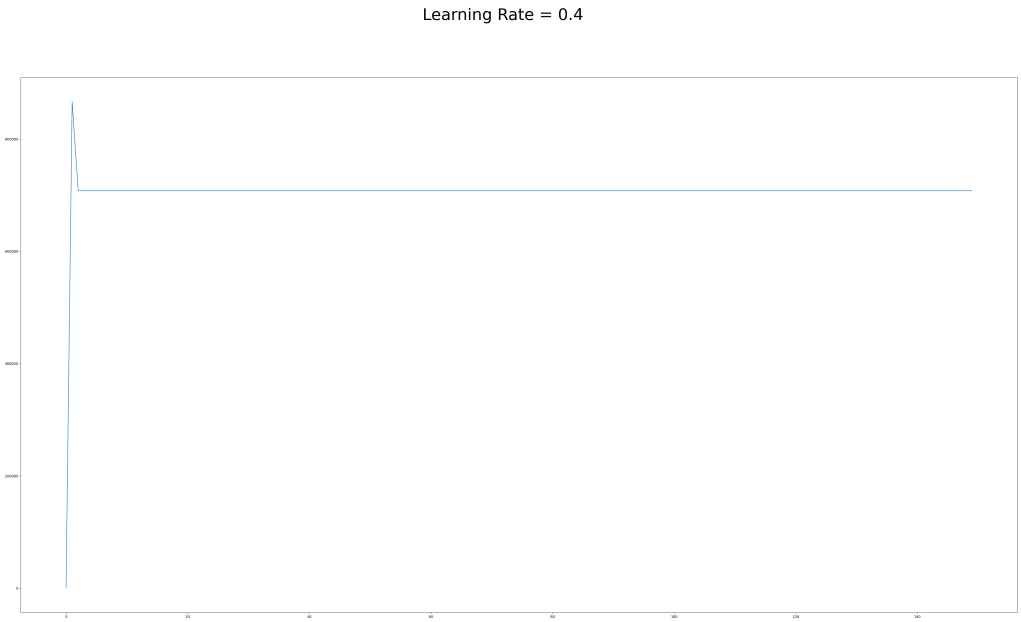


Figure C.6: Learning Rate 0.4

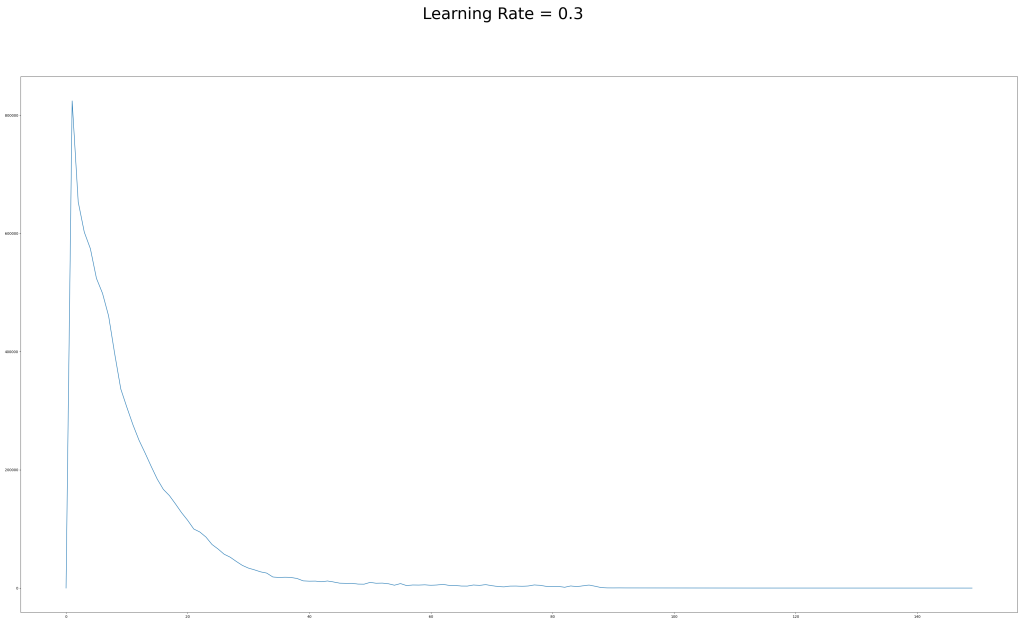


Figure C.7: Learning Rate 0.3

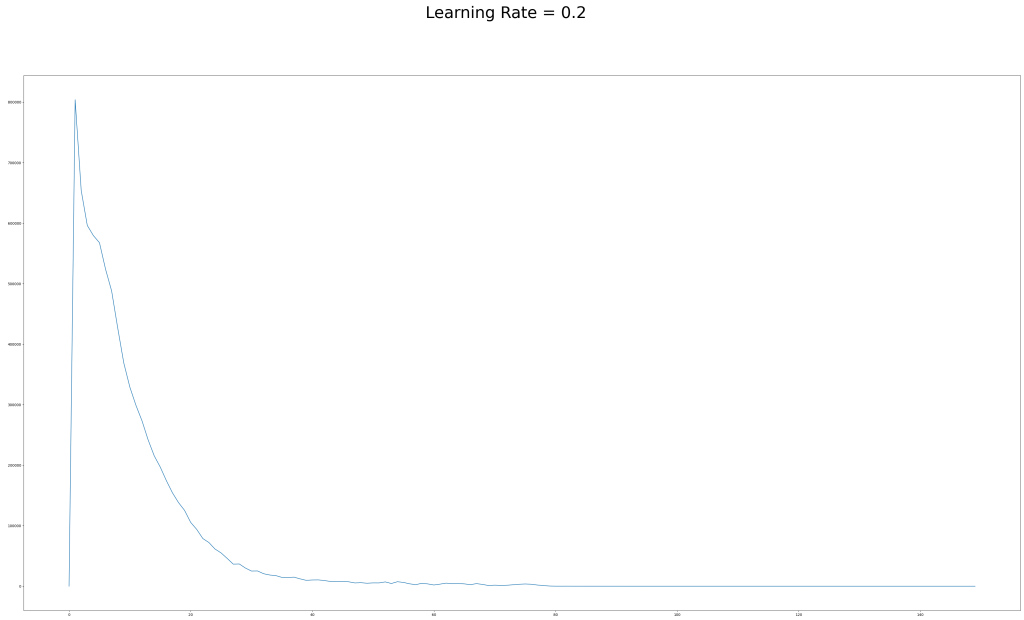


Figure C.8: Learning Rate 0.2

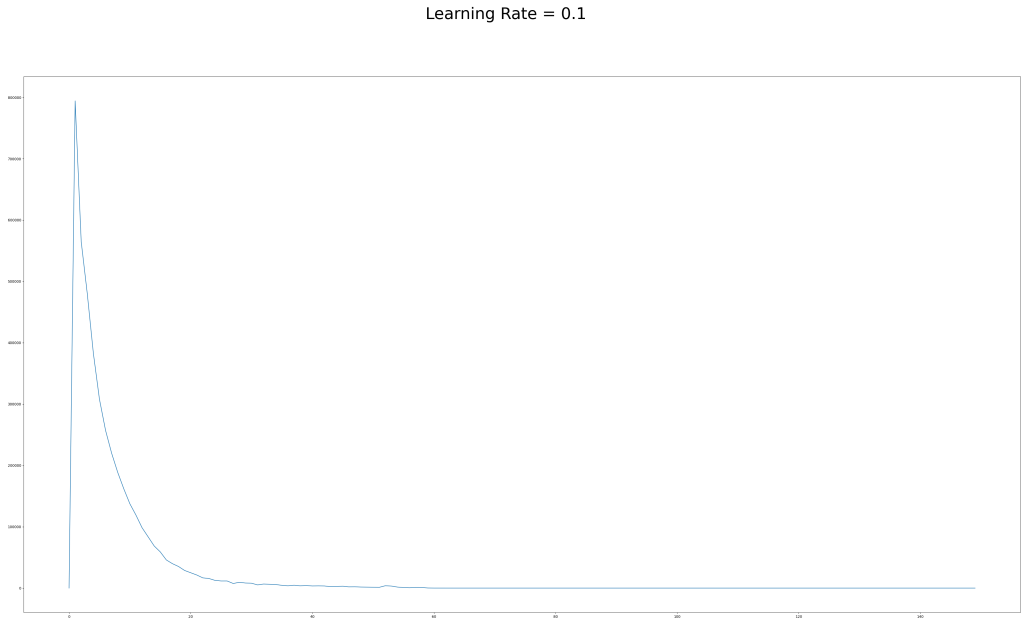


Figure C.9: Learning Rate 0.1

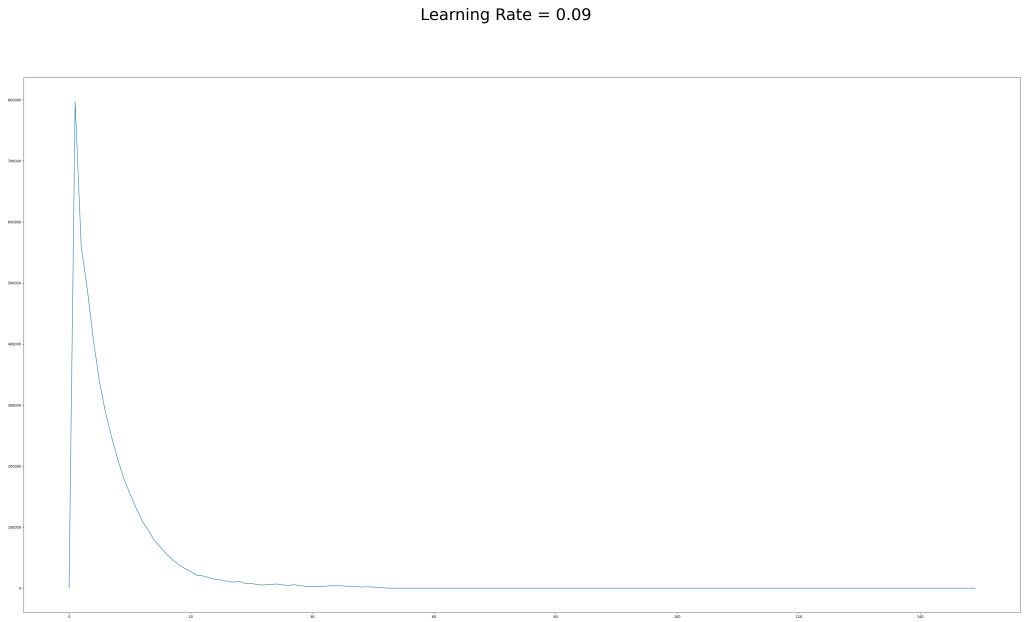


Figure C.10: Learning Rate 0.09

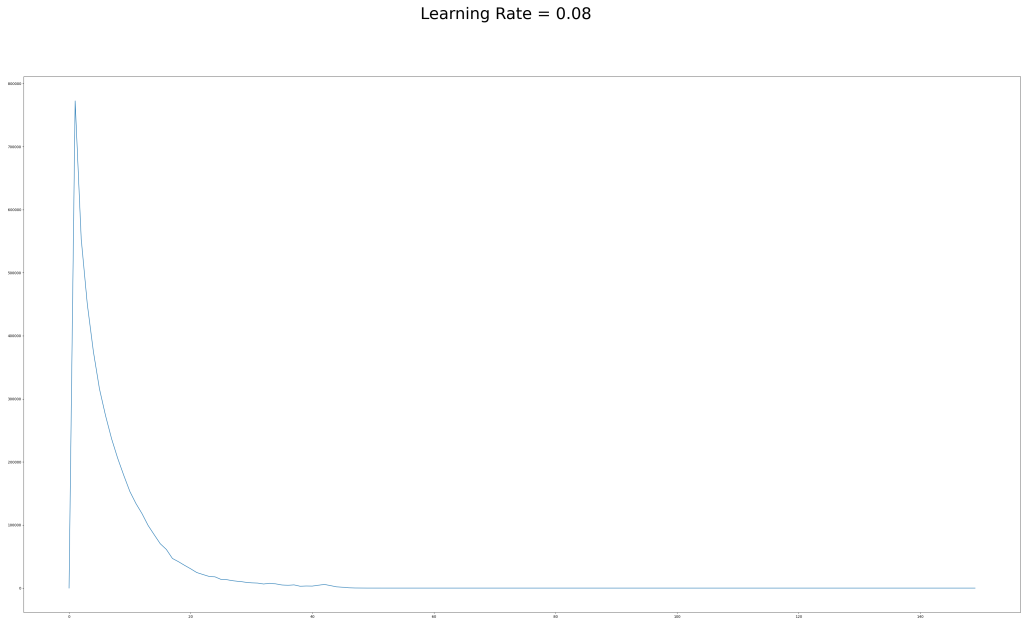


Figure C.11: Learning Rate 0.08

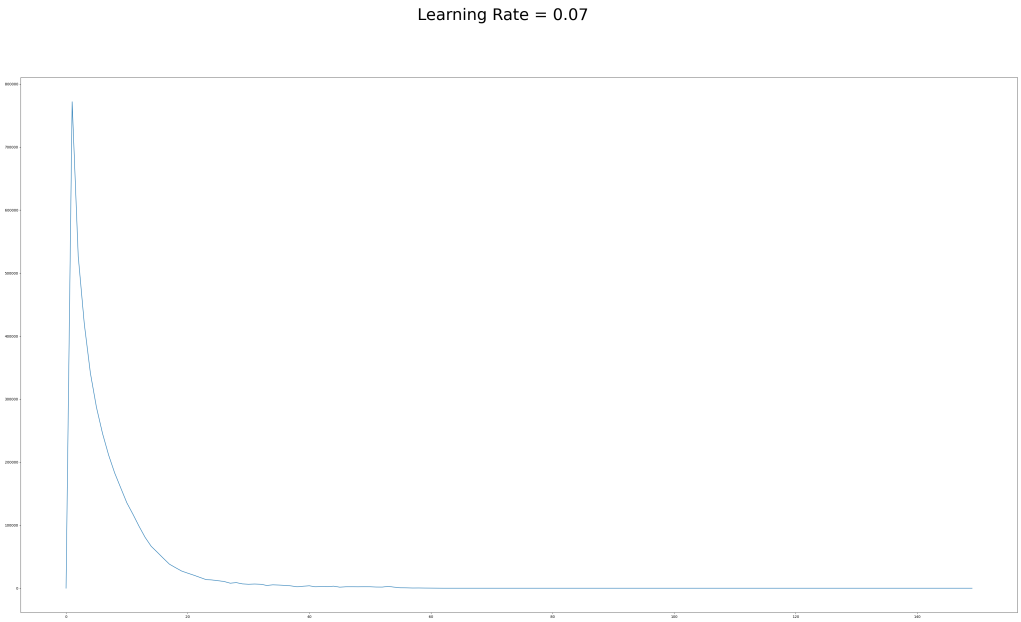


Figure C.12: Learning Rate 0.07

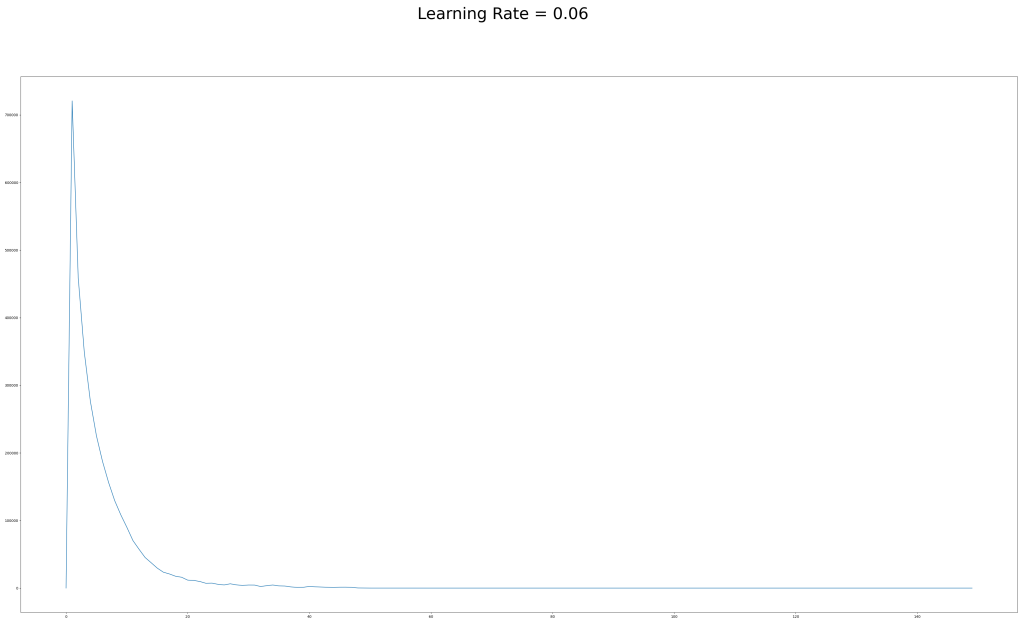


Figure C.13: Learning Rate 0.06

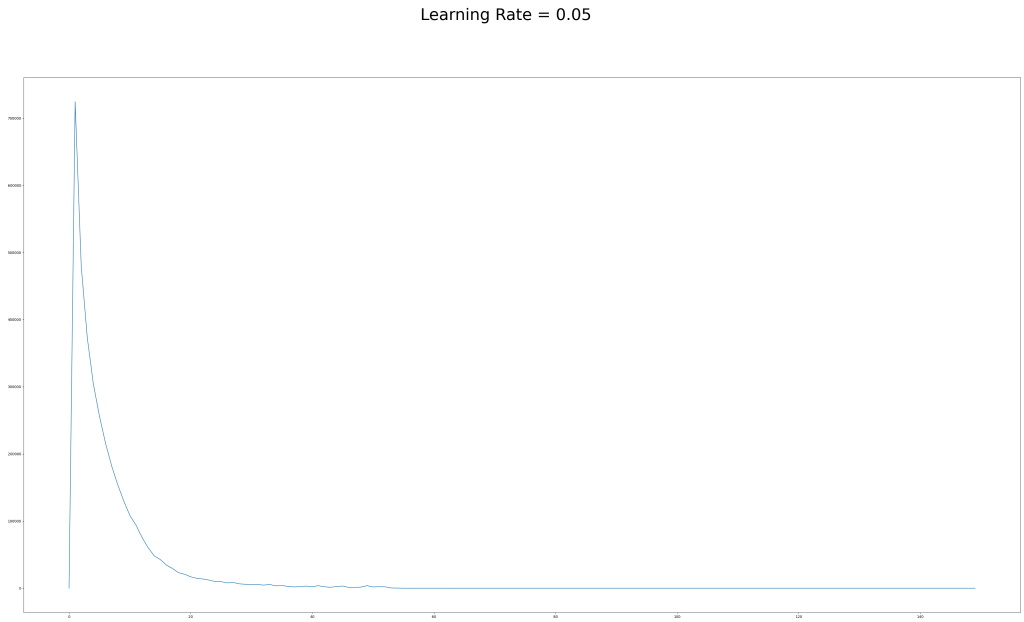


Figure C.14: Learning Rate 0.05

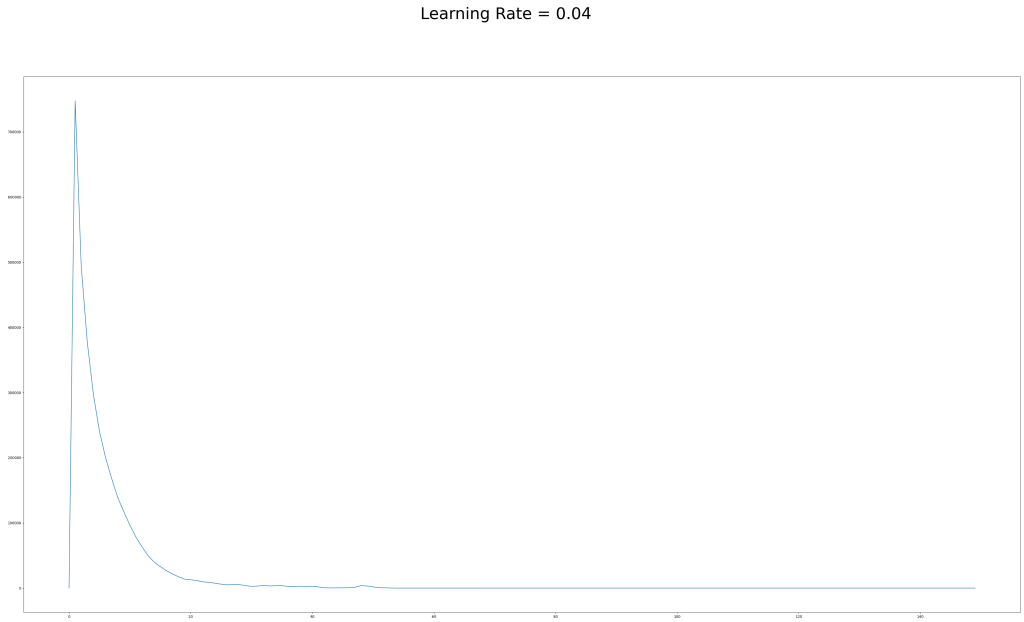


Figure C.15: Learning Rate 0.04

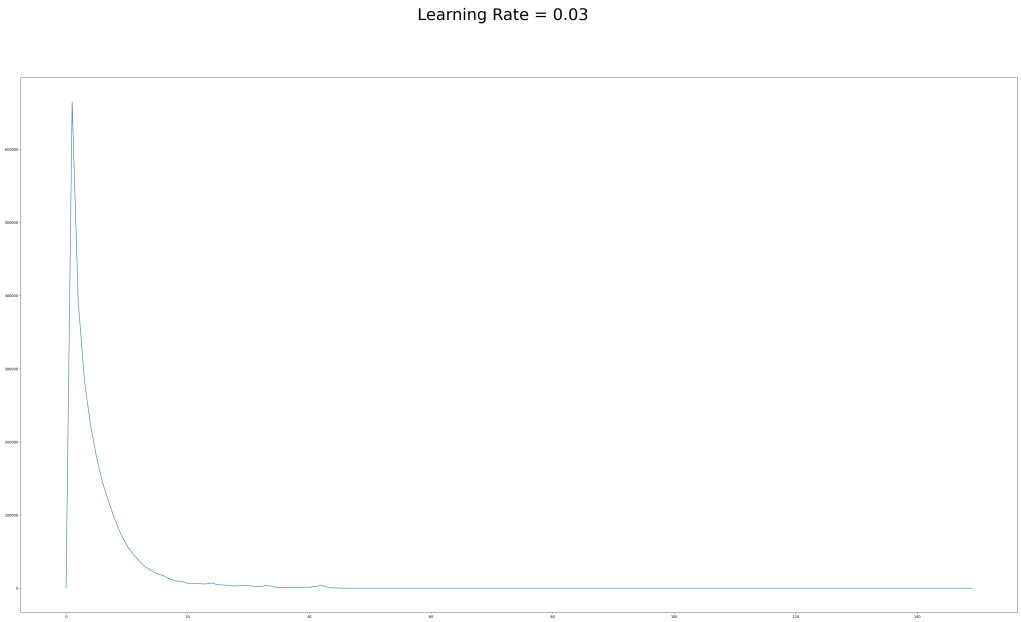


Figure C.16: Learning Rate 0.03

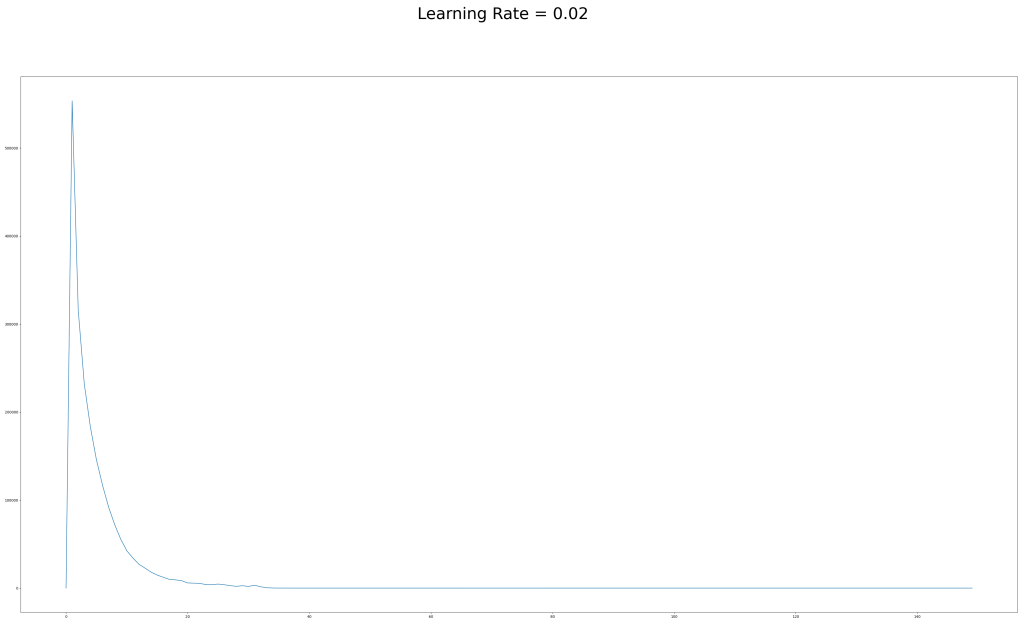


Figure C.17: Learning Rate 0.02

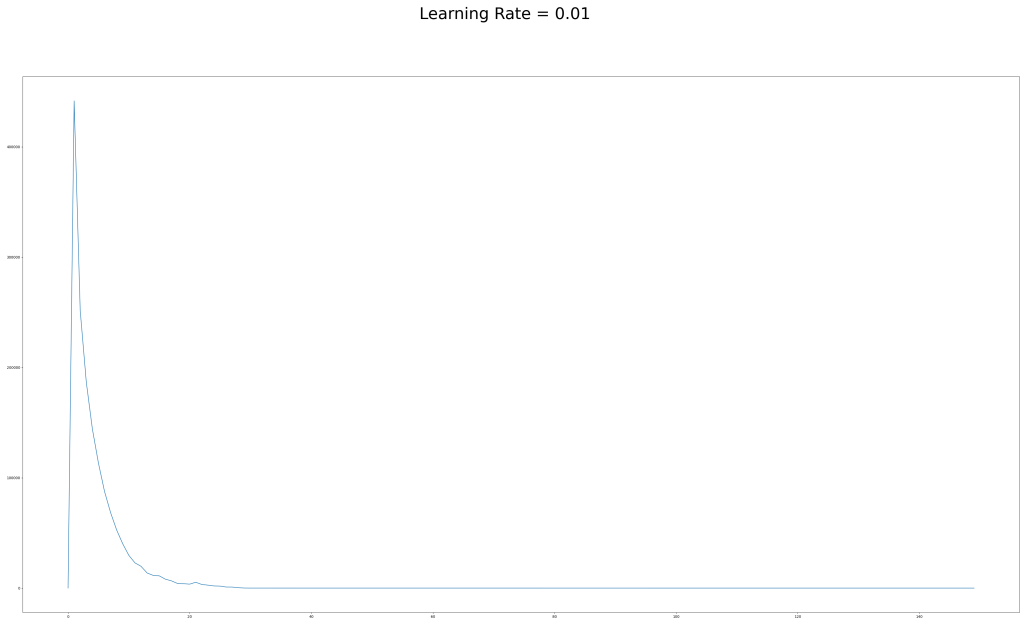


Figure C.18: Learning Rate 0.01

Bibliography

[11] [Online; accessed 30. Aug. 2020]. June 2011. URL: https://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf (cit. on pp. 16, 17).

[16a] [Online; accessed 25. Jun. 2021]. May 2016. URL: http://rocs.hu-berlin.de/D3/logistic (cit. on pp. 8, 34).

[19] [Online; accessed 20. Sep. 2020]. May 2019. URL: https://www.nap.edu/resource/25303/R&R.pdf (cit. on p. 1).

[20a] [Online; accessed 13. Oct. 2020]. Sept. 2020. URL: http://www-users.math.umn.edu/~arnold//disasters/GAO-IMTEC-92-96.pdf (cit. on p. 6).

[20b] [Online; accessed 20. Sep. 2020]. Oct. 2020. URL: https://www.cs.mcgill.ca/~jpineau/ReproducibilityChecklist.pdf (cit. on pp. 7, 10, 12).

[21a] *Apache Parquet*. [Online; accessed 4. Aug. 2021]. Mar. 2021. URL: https://parquet.apache.org (cit. on p. 19).

[Bak16] Monya Baker. "1,500 scientists lift the lid on reproducibility". In: *Nature* 533 (May 2016), pp. 452–454. ISSN: 1476-4687. DOI: 10.1038/533452a (cit. on pp. 2, 3).

[Bar19] Gregory Barber. "Artificial Intelligence Confronts a 'Reproducibility' Crisis". In: *Wired* (Sept. 2019). URL: https://www.wired.com/story/artificial-intelligence-confronts-reproducibility-crisis (cit. on p. 10).

[BL89] Suzanna Becker and Yann Lecun. "Improving the Convergence of Back-Propagation Learning with Second-Order Methods". In: Jan. 1989 (cit. on p. 42).

[Ben21] Sarat Moka Benoit Liquet. *3 Optimization Algorithms | The Mathematical Engineering of Deep Learning*. Mar. 2021. URL: https://deeplearningmath.org/optimization-algorithms.html (cit. on p. 40).

[BPL10] Y-Lan Boureau, Jean Ponce, and Yann LeCun. "A theoretical analysis of feature pooling in visual recognition". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 111–118 (cit. on p. 37).

[CC15] Jia-Ren Chang and Yong-Sheng Chen. *Batch-normalized Maxout Network in Network*. 2015. arXiv: 1511.02583 [cs.CV] (cit. on p. 45).

[21b] *Changes from Previous Version*. [Online; accessed 20. Sep. 2020]. May 2021. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html/#options-for-steering-gpu-code-generation-fmad (cit. on p. 17).

[17] *CIFAR-10 and CIFAR-100 datasets*. [Online; accessed 9. Aug. 2021]. Apr. 2017. URL: https://www.cs.toronto.edu/~kriz/cifar.html (cit. on pp. 42, 43).

[20c] *CIFAR-10 on Benchmarks.AI*. [Online; accessed 04. Oct. 2020]. Oct. 2020. URL: https://benchmarks.ai/cifar-10 (cit. on pp. 1, 7).

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. 2015. arXiv: 1511.07289 [cs.LG] (cit. on p. 45).

[20d] *Concepts — MLflow 1.18.0 documentation*. [Online; accessed 23. Oct. 2020]. Oct. 2020. URL: https://www.mlflow.org/docs/latest/concepts.html (cit. on pp. 1, 18).

[21c] *CUDA C++ Programming Guide*. [Online; accessed 30. Aug. 2020]. May 2021. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#mathematical-functions-appendix (cit. on pp. 4, 5).

[16b] *Do results vary depending on GPU or driver versions? - CUDA / CUDA Programming and Performance - NVIDIA Developer Forums*. [Online; accessed 04. Oct. 2020]. Oct. 2016. URL: https://forums.developer.nvidia.com/t/do-results-vary-depending-on-gpu-or-driver-versions/45539/6 (cit. on pp. 2, 5).

[21d] *Docker overview*. [Online; accessed 29. Aug. 2021]. Aug. 2021. URL: https://docs.docker.com/get-started/overview (cit. on pp. 30–32).

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of machine learning research* 12.7 (2011) (cit. on p. 41).

[20e] *fmad=false gives good performance*. [Online; accessed 19. Oct. 2020]. Oct. 2020. URL: https://stackoverflow.com/questions/12011708/fmad-false-gives-good-performance (cit. on p. 17).

[21e] *Gradient descent (article) | Khan Academy*. [Online; accessed 12. Aug. 2021]. Aug. 2021. URL: https://www.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/optimizing-multivariable-functions/a/what-is-gradient-descent (cit. on pp. 38, 41).

[Gra14] Benjamin Graham. *Spatially-sparse convolutional neural networks*. 2014. arXiv: 1409.6070 [cs.CV] (cit. on p. 45).

[Gro17] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491962291 (cit. on pp. 45, 46).

[Gro] Ryan Gross. *How the Amazon Go Store's AI Works*. URL: https://towardsdatascience.com/how-the-amazon-go-store-works-a-deep-dive-3fde9d9939e9 (visited on 08/30/2020) (cit. on p. 1).

[Guo+17] Tianmei Guo et al. "Simple convolutional neural network on image classification". In: *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*. 2017, pp. 721–724. DOI: 10.1109/ICBDA.2017.8078730 (cit. on pp. 35–37).

[He+16] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2016). DOI: 10.1109/cvpr.2016.90. URL: http://dx.doi.org/10.1109/cvpr.2016.90 (cit. on p. 45).

[21f] *Home*. [Online; accessed 5. Aug. 2021]. Aug. 2021. URL: https://dvc.org/doc (cit. on pp. 24, 25).

[20f] *IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic*. [Online; accessed 30. Aug. 2020]. Aug. 2020. URL: https://standards.ieee.org/standard/754-2019.html (cit. on pp. 16, 17).

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105 (cit. on pp. 36, 37).

[Lec+98] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (cit. on p. 36).

[LGT15] Chen-Yu Lee, Patrick W. Gallagher, and Zhuowen Tu. *Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree*. 2015. arXiv: 1509.08985 [stat.ML] (cit. on p. 45).

[LC15] Zhibin Liao and Gustavo Carneiro. "Competitive Multi-scale Convolution". In: *arXiv* (Nov. 2015). eprint: 1511.05635. URL: https://arxiv.org/abs/1511.05635v1 (cit. on p. 45).

[May76] Robert M. May. "Simple mathematical models with very complicated dynamics". In: *Nature* 261 (June 1976), pp. 459–467. ISSN: 1476-4687. DOI: 10.1038/261459a0 (cit. on pp. 34, 35).

[Mit+19] Margaret Mitchell et al. "Model Cards for Model Reporting". In: *Proceedings of the Conference on Fairness, Accountability, and Transparency* (Jan. 2019). DOI: 10.1145/3287560.3287596. URL: http://dx.doi.org/10.1145/3287560.3287596 (cit. on pp. 10, 13–15).

[21g] *MLflow Model Registry — MLflow 1.19.0 documentation*. [Online; accessed 5. Aug. 2021]. July 2021. URL: https://www.mlflow.org/docs/latest/model-registry.html (cit. on p. 20).

[21h] *MLflow Model Registry — MLflow 1.19.0 documentation*. [Online; accessed 5. Aug. 2021]. July 2021. URL: https://www.mlflow.org/docs/latest/model-registry.html (cit. on p. 22).

[21i] *MLflow Models — MLflow 1.19.0 documentation*. [Online; accessed 4. Aug. 2021]. July 2021. URL: https://www.mlflow.org/docs/latest/models.html (cit. on p. 20).

[21j] *MLflow Projects — MLflow 1.19.0 documentation*. [Online; accessed 4. Aug. 2021]. July 2021. URL: https://www.mlflow.org/docs/latest/projects.html (cit. on p. 19).

[MBL20] Kevin Musgrave, Serge Belongie, and Ser-Nam Lim. *A Metric Learning Reality Check*. 2020. arXiv: 2003.08505 [cs.CV] (cit. on p. 2).

[NES83] Y. NESTEROV. "A method for unconstrained convex minimization problem with the rate of convergence o($1/k^2$)". In: *Doklady AN USSR* 269 (1983), pp. 543–547. URL: https://ci.nii.ac.jp/naid/20001173129/en/ (cit. on p. 41).

[NVIa] NVIDIA. *CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#numerical-accuracy-and-precision (visited on 08/30/2020) (cit. on p. 17).

[NVIb] NVIDIA. *NVIDIA Driver Downloads*. URL: https://www.nvidia.com/Download/Find.aspx (visited on 11/10/2020) (cit. on p. 2).

[Pen] Tony Peng. *The Staggering Cost of Training SOTA AI Models*. URL: https://syncedreview.com/2019/06/27/the-staggering-cost-of-training-sota-ai-models/ (cit. on p. 11).

[Qia99] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural Networks* 12.1 (Jan. 1999), pp. 145–151. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(98)00116-6 (cit. on p. 40).

[Ros58] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519 (cit. on p. 35).

[Rud17] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG] (cit. on pp. 38–41).

[21k] *Sharing Data and Model Files*. [Online; accessed 8. Aug. 2021]. Aug. 2021. URL: https://dvc.org/doc/use-cases/sharing-data-and-model-files (cit. on pp. 25, 27).

[SZ15] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV] (cit. on pp. 37, 47).

[Sno+15] Jasper Snoek et al. *Scalable Bayesian Optimization Using Deep Neural Networks*. 2015. arXiv: 1502.05700 [stat.ML] (cit. on p. 45).

[Suz11] Kenji Suzuki. *Artificial Neural Networks - Methodological Advances and Biomedical Applications*. Apr. 2011. ISBN: 978-953-307-243-2 (cit. on p. 35).

[Sze+14] Christian Szegedy et al. *Going Deeper with Convolutions*. 2014. arXiv: 1409.4842 [cs.CV] (cit. on p. 37).

[02] *The Real Reason Reproducible Research is Important · Simply Statistics*. [Online; accessed 04. Oct. 2020]. Oct. 202. URL: https://simplystatistics.org/2014/06/06/the-real-reason-reproducible-research-is-important (cit. on p. 6).

[20g] *The untold story of GPT-3 is the transformation of OpenAI*. [Online; accessed 20. Sep. 2020]. Sept. 2020. URL: https://bdtechtalks.com/2020/08/17/openai-gpt-3-commercial-ai (cit. on p. 3).

[21l] *Versioning Data and Models*. [Online; accessed 7. Aug. 2021]. Aug. 2021. URL: https://dvc.org/doc/use-cases/versioning-data-and-model-files (cit. on p. 24).

[Wan+12] Tao Wang et al. "End-to-end text recognition with convolutional neural networks". In: *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*. 2012, pp. 3304–3308 (cit. on p. 37).

[21m] *Welcome to Sacred's documentation! — Sacred 0.8.2 documentation*. [Online; accessed 5. Aug. 2021]. Jan. 2021. URL: https://sacred.readthedocs.io/en/stable (cit. on p. 27).

[21n] *What is a Container? | Docker*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: kl (cit. on pp. 30, 33).

[21o] *What Is a Virtual Machine and How Does It Work | Microsoft Azure*. [Online; accessed 2. Sep. 2021]. Sept. 2021. URL: https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/#overview (cit. on pp. 28, 29).

[Wig20] Kyle Wiggers. "Google's breast cancer-predicting AI research is useless without transparency, critics say". In: *VentureBeat* (Oct. 2020). URL: https://venturebeat.com/2020/10/14/googles-breast-cancer-predicting-ai-research-is-useless-without-transparency-critics-say (cit. on pp. 2, 3).

[XD18] Wanli Xing and Dongping Du. "Dropout Prediction in MOOCs: Using Deep Learning for Personalized Intervention". In: *Journal of Educational Computing Research* 57.3 (Mar. 2018), pp. 547–570. ISSN: 0735-6331. DOI: 10.1177/0735633118757015 (cit. on p. 35).

[ZF13] Matthew D Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*. 2013. arXiv: 1311.2901 [cs.CV] (cit. on p. 37).

[Zei12] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012. arXiv: 1212.5701 [cs.LG] (cit. on pp. 41, 42).

[ZL15] Xiang Zhang and Yann LeCun. *Universum Prescription: Regularization using Unlabeled Data*. 2015. arXiv: 1511.03719 [cs.LG] (cit. on p. 45).

[Zho+20] Zhihao Zhou et al. "Sign-to-speech translation using machine-learning-assisted stretchable sensor arrays". In: *Nat. Electron.* 3 (Sept. 2020), pp. 571–578. ISSN: 2520-1131. DOI: 10.1038/s41928-020-0428-6 (cit. on p. 1).

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, September 6, 2021

_____

Shashank Pandey