Predicting Energy Consumption of Matrix Operations.

Optimisation for Computer Science Course *

Tobias M. FISCHBACH†
University of Luxembourg
2 Av. de l'Université
4365, Esch-Belval,
Esch-sur-Alzette
tobias.fischbach@uni.lu

Xavier-R. FRANÇOIS
University of Luxembourg
2 Av. de l'Université
4365, Esch-Belval,
Esch-sur-Alzette
xavier.francois.001@student.uni.lu

Elliot KOCH
University of Luxembourg
2 Av. de l'Université
4365, Esch-Belval,
Esch-sur-Alzette
0190432505@uni.lu

Gabriel S.J. SPADONI
University of Luxembourg
2 Av. de l'Université
4365, Esch-Belval,
Esch-sur-Alzette
gabriel.spadoni.001@student.uni.lu

ABSTRACT

This paper covers the work done by the authors under the mentoring of their supervisors for the course Optimisation for Computer Science during their second semester of the Master in Computer Science at the University of Luxembourg. The project focused on measuring and optimising the energy consumption of computer programs. This document provides a report including the research, experiments and their results.

1. INTRODUCTION

Nowadays, energy is a sensitive topic that attracts a lot of attention. Indeed, the ever-increasing global energy needs that comes with our scaling population is becoming more and more of a concern as we consume energy at a higher rate then we generate it, depleting consumable energy sources that cannot be regenerated. Because of that, there are conscious efforts of finding new alternatives to generate energy, but also to save energy and invest into developing sustainable energy[1].

In our case, we are interested in the latest, we were offered to work on this project with researchers from the University of Luxembourg to delve into computer program energy consumption and how to optimize them to lower their consumption and hopefully lead to energy savings.

There are a lot of computer programs, written in various programming languages, that are used in many fields and for many different purposes. We want to explore and generate numerical data to allow comparison of a selection of those programs and to have an idea of the energy consumption of specific programs. With this information, we will then attempt to build a neural network to predict energy usage and optimise the energy consumption for a specific program and present our findings.

2. STATE OF THE ART

We searched for different means to compute the energy cost of a program's execution, as well as what other researchers already did about this issue. Several articles redirected us towards the same source **SLE17**.

In this paper, they used **Intel's RAPL** (Running Average Power Limit) and **PERF** on a Linux machine to compute the energy cost of a selection of 10 different programs each written in 28 different programming languages, then they reported the results. But, this document dates from 2017 and we have not found other significant attempts published since then.

We also found **The Benchmark Game**, but this one focus on program execution speed and not their energy consumption. Yet, it was an interesting take that we considered when designing our own approach, and it is also mentioned and referenced in the SLE17 document.

There exist several other power measurement libraries and tools available that we could use instead of RAPL. Here are a few:

PowerAPI: An API that provides access to power and energy consumption information of applications running on modern processors.

LIKWID: A set of command-line tools and an API for monitoring and analyzing the performance and energy consump-

*Group project by semester 2 Master students from the University of Luxembourg.

†Project Supervisor

tion of CPUs and GPUs.

Intel Power Gadget: A software-based power measurement tool that provides real-time power and energy consumption data for Intel processors.

RAPL-TOOLS: A set of command-line tools that provide access to RAPL interface, allowing users to monitor and control power consumption on Intel processors.

These tools and libraries provide a range of features and functionality for measuring power consumption during program execution. However, it is important to note that it remains difficult to distinguish exactly which part of the energy cost corresponds to the program itself because there are often many threads running in the background and they may affect the overall CPU power usage.

3. PROBLEM PRESENTATION

The project aims to provide an understanding of the energy consumed by computer programs. We chose to target matrix multiplications operation to begin with. We want to analyze what characteristics will affect the energy consumption of a matrix operation and how it will impact it.

More precisely, we want to optimize the energy consumption of a matrix operation with regards to specific parameters such as the size, the value type and the programming language used to implement the program.

The idea behind it is that the matrix multiplications are units performed many times in neural networks training. So, by reducing the energy cost on a small unit, it could scale to significant energy savings.

3.1 Matrix Operation

In this project, we consider square matrix multiplication. Due to time constraints, we might not be able to evaluate non-square matrices. A matrix multiplication is defined as follow from the mathematical definition:

For two matrices A and B, the result is $C_{i,j} = A_i \cdot B_j$ for each entry i and j. Here A_i is the line i of matrix A and B_j is the j^{th} column of matrix B.

3.2 Value types

We will consider different types of values inside the matrices. We will focus a mix of positive, negative integers and floating point numbers.

Integers are positive natural numbers. They are represented with 32-bit or 64 bit. A multiplication of integers is simply the multiplication of both numbers.

Floating Point numbers are used to represent real numbers. They are structured with 32-bit or 64-bit, where 1 bit is the sign, for 32 bit length 8 bit for the exponent and 23 for the significant. So we have: $n = sign \times significant \times 2^{exponent}$

From this, we see that a floating point multiplication consists of the multiplication of signs and significants and of the addition of exponents.

4. APPROACH

For our measurements, we built a pipeline that executes a bash script on the **HPC**(High Performance Computer).

It loops through different values for a selection of variables of interest and tracks them. For each iteration, it computes the energy consumption when executing the matrix multiplication operation program written in Python or Java according to the variables' values for that iteration. The energy results are obtained with the help of RAPL and PERF and called by the bash script. But first we needed to generate said matrix with respect to the parameters. To do so, we first generated the matrix inside the program and accounted for this extra code execution in the results.

Yet, we also worked on a solution where the matrix would be generated before hand so that it would not affect the energy cost of the matrix multiplication itself. We successfully achieved that goal towards the end of the project.

For practical reasons, The program multiplies the generated square matrix by itself. We paid attention to limit other factors that could influence the energy results, such as importing and using external packages or presence of irrelevant code. Also, we tried isolating the program's execution and minimizing the number of other tasks happening on the CPU when evaluating the energy costs. We also measured a Python "sleep" program to get a baseline of the CPU energy consumption while close to inactive.

After the bash script's loop is done, it writes the parameters together with their related results in a .CSV file. We created one pipeline for Python and one for Java. We used the data generated to build datasets. With them, we trained a neural network to predict the energy cost based on the chosen features. Lastly, we computed the energy cost of the neural network's training process itself.

5. EXECUTION ENVIRONMENT

We worked with the HPC of the University of Luxembourg on which Linux is installed. We used the IRIS-cluster, on a dedicated node for our project which had 28 cores. We used 1 node and 1 core when doing tests on the interactive session. We used a reserved node specifically iris-168 which has 128 GB of RAM and "2 Xeon Gold 6132 @ 2.6GHz [14c/140W]" 2. Then for the run outside of testing we reserved every core of the node in order to make sure no else is using the node in order to have accurate results. We used the default python installation currently present on the HPC which is 3.9.7. We set up a virtual environment that contains the specific packages that can be found in the 'requirements.txt' file present in the **Github** repository of the project.

For the java part, we used the following version openjdk version "1.8.0_362".

We setup a Github repository containing a copy of all the code we used on the HPC and the folder structure. This helped to keep track on the different modifications we have done and to make sure everyone is using the same versions of scripts.

6. DATA GENERATION AND COLLECTION

In order to measure energy consumption of a matrix operation, we use the `perf` command as said earlier. This command with the options we used, outputs a text file containing the following information: How long the process took to execute, the number of cache misses that happened while executing, the energy consumed by the RAM in Joules and the energy consumed by the PKG in Joules.

We use **-a** which corresponds to all CPUs of the system and **-e** to select the particular events we want. This is why we are selecting the cache misses, the energy consumed by the RAM and the energy consumed by the PKG.

A cache miss corresponds to the moment when the process tries to access data present in the cache memory and fails to find it.

In our bash script, we implemented a system so that the parameters for that given execution would be written together in that output text file.

6.1 Matrix Operation Energy Consumption Dataset

We created a dataset where the important features regarding energy consumption are present. This is why we use a **.csv** format with each column representing one feature. The different features are:

- The matrix size: 10, 100 and 1000 (square matrices)
- The value type: integer, floating point or negative
- The duration of the process in seconds: it corresponds to the time given by the `perf` command
- The number of cache misses
- The range between the extreme possible values
- The programming language: Python or Java
- The PKG energy in Joules
- The RAM energy in Joules

The other concern is that we wanted to measure the energy of one single multiplication at a time. This is why we created a bash script that iterates and runs the PERF command on the matrix operation program with the parameters we want passed into the program as argument.

We first created a Python program that takes as argument input the size, the type and the range of the matrix desired and then generates it within the program before applying the multiplication. For simplicity sake, we focused on square matrices and we multiply the matrix by itself.

After this the bash file takes the output of the `perf` command and writes it into a text file containing the inputs used for the matrix operation. When all operations have been done, we regroup them in a unique csv file containing the information for each single matrix operation and delete all temporary files.

This method has the problem of generating the matrix inside the perf command. This is why, in the next step, we improved the program so that it separates matrix generation and multiplication operation in two different scripts and only the matrix multiplication phase is evaluated for energy consumption. However, removing the generation inside the program means we have to account for the energy cost of retrieving the matrix from the argument.

The approach of this method is to generate the matrix using a Python program by taking the same argument inputs as previously and then to write it in the terminal in order to cast it inside a variable which will be used as input for the matrix operation program. We then proceed as before. Here, one of the problem was the size of the matrix in the terminal was randomly generating floats of large length, this is why we use the `round` method in Python to make them smaller.

6.2 Neural Network Energy Consumption Dataset

We created our second dataset with a different program. It is about the energy consumed by a standard neural network created using TensorFlow in Python. For this dataset we measure the energy consumed by the training of the neural network and how it changes according to modification made to the set of hyper parameters.

It was generated with the following features:

- the batch size: 32, 64, 128 and 256
- the epochs: 5, 10, 15, 20, 25, 30, 35, 40, 45 and 50
- the learning rate: 0.1, 0.001 and 0.0001
- The duration of the process in seconds: it corresponds to the time given by the `perf` command
- Total Error Margin
- The cache misses
- The PKG energy in Joules
- The RAM energy in Joules

The approach used to generate this dataset is similar to the other one as we iterate through the different values possible for each hyper parameters and we run the `perf` command on the training of the neural network. After each `perf`, we write the outputs into a text file. At the end, every text file is used to create the csv file and then the temporary text files are all deleted.

7. NEURAL NETWORK

This section will be about the Neural network we built, what it does, and the parameters we chose.

7.1 Dataset Pre-Processing

Before delving into the Neural Network, we conducted essential pre-processing steps on the datasets to improve the accuracy of our predictions.

7.1.1 *Energy*

The first step is to add an 'Energy' column to the datasets. This column will be the one the Neural Networks would have to predict.
The datasets created are composed of 9 columns: 'File', 'Language', 'Size', 'Range', 'Type', 'PKG_Energy', 'RAM_Energy', 'Cache Misses', and 'Time'. Based on that, the new 'Energy' column was created by summing the values of the 'PKG_Energy' and 'RAM_Energy' columns for each row. After calculating the sum of the 'PKG_Energy' and 'RAM_Energy' columns to create the 'Energy' column, both 'PKG_Energy' and 'RAM_Energy' were subsequently removed. This was done to prevent the Neural Network from learning that the sum of the two columns is the correct energy prediction.
In addition to that, the 'File' column was removed and the 'Language' one was also removed except for the combined dataset of Python and Java.

7.1.2 *Normalisation*

Now that we have created the 'Energy' column, we still have 8 columns in the datasets: 'Size', 'Range', 'Type', 'Cache Misses', 'Time', and 'Energy'.

The next step is to normalise the dataset to predict more accurately. But here, it was tricky because we could not normalise all the columns.
For example, if we normalise the 'Energy' column, we would have to take the predicted energy for each prediction and perform an inverse normalisation to retrieve the energy consumption in joules. Also, the 'Size' and 'Type' columns are composed of fixed labels: [10, 100, 1000] and [1, 2, 3], respectively. Normalising these values would not change the prediction for the better.
To conclude the pre-processing phase, the 'Range', 'Cache Misses', and 'Time' columns were normalised for all datasets.

7.2 Model

We locally implemented a neural network using Keras. The network consists of four layers: an input layer with 64 nodes, two hidden layers with 32 nodes each, and an output layer with 1 node. All layers are dense, with ReLU activation used in the first three layers and linear activation in the output layer. The network was compiled with the Huber loss function and Adam optimizer.

7.3 Optimisation Process

In our optimization process, we utilized nested for-loops to systematically evaluate the performance of different hyperparameters. We started by iterating over different epoch values (5, 10, 25, and 50) to identify the optimal number of training iterations that yielded the best results.
Next, we repeated the process by iterating over various batch sizes (32, 64, 128, and 256) to determine the batch size that produced the most favourable outcomes for model performance.
Finally, we iterated over different learning rates (0.0001, 0.001, 0.01, and 0.1) to pinpoint the optimal learning rate that led to the most accurate predictions.

We trained the model and evaluated its performance by analyzing the loss and margin error between the predicted energy values and the corresponding energy costs for a sample set of 1000 data points at each iteration. By systematically varying the hyper-parameters and observing the resulting performance metrics, we were able to select the hyperparameter values that produced the best overall results for our energy consumption prediction model.

7.4 Energy / Loss Results

Below is the overall optimization process diagram for the entire datasets. To have a better comparison of the varying energy / loss results based on different parameters, it has been decided to group them together.

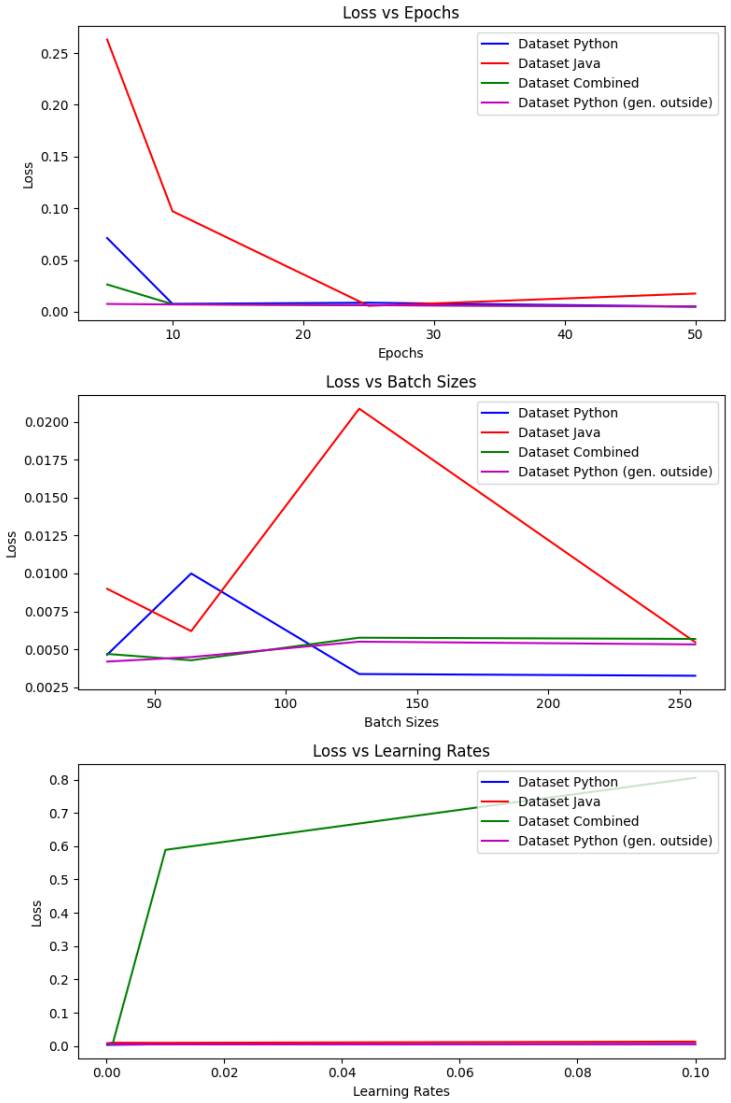

Figure 1: Final loss of the training depending of the hyperparameters. (The plot style was adapted to color blind persons)

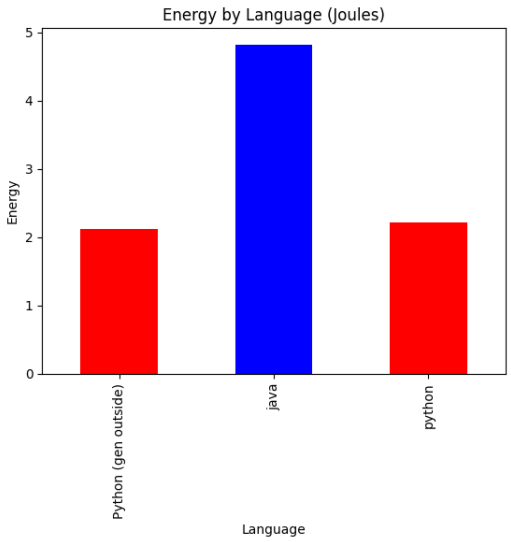

Figure 2:
Python = 2.20 (Joules)
Java = 4.82 (Joules)
Python (gen. outside) = 2.11 (Joules)

7.5 Python Dataset

The result for the first Python dataset is presented in this section, starting with the epochs number.

Margin Error for the epochs optimization process:

- 5 epochs: 0.065
- 10 epochs: 0.078
- 25 epochs: 0.050
- 50 epochs: 0.042

Margin Error for the batch-size optimization process:

- 32 as batch-size: 0.041
- 64 as batch-size: 0.044
- 128 as batch-size: 0.037
- 256 as batch-size: 0.760

Margin Error for the learning-rate optimization process:

- 0.0001 as learning rate: 0.059
- 0.001 as learning rate: 0.041
- 0.01 as learning rate: 0.048
- 0.1 as learning rate: 0.055

Final results on the optimization process for the first python dataset:

- Epochs: 50
- Batch-size: 128
- Learning Rate: 0.001
- Margin Error: 0.04021
- Loss: 0.0032

7.6 Java Dataset

The result for the Java dataset is presented in this section, starting with the epochs number.

Margin Error for the epochs optimization process:

- 5 epochs: 0.578
- 10 epochs: 0.126
- 25 epochs: 0.055
- 50 epochs: 0.063

Margin Error for the batch-size optimization process:

- 32 as batch-size: 0.045
- 64 as batch-size: 0.066
- 128 as batch-size: 0.041
- 256 as batch-size: 0.496

Margin Error for the learning-rate optimization process:

- 0.0001 as learning rate: 0.132
- 0.001 as learning rate: 0.059
- 0.01 as learning rate: 0.0737
- 0.1 as learning rate: 0.0732

Final results on the optimization process for the Java dataset:

- Epochs: 25
- Batch-size: 128
- Learning Rate: 0.001
- Margin Error: 0.055
- Loss: 0.0053

7.7 Combined Dataset (Python / Java)

The result for the combined dataset is presented in this section, starting with the epochs number.

Margin Error for the epochs optimization process:

- 5 epochs: 0.732
- 10 epochs: 0.064
- 25 epochs: 0.058
- 50 epochs: 0.044

Margin Error for the batch-size optimization process:

- 32 as batch-size: 0.044
- 64 as batch-size: 0.047
- 128 as batch-size: 0.041
- 256 as batch-size: 0.046

Margin Error for the learning-rate optimization process:

- 0.0001 as learning rate: 0.045
- 0.001 as learning rate: 0.101
- 0.01 as learning rate: 0.894
- 0.1 as learning rate: 1.310

Final results on the optimization process for the combined dataset:

- Epochs: 50
- Batch-size: 128
- Learning Rate: 0.001
- Margin Error: 0.048
- Loss: 0.0045

7.8 Python Dataset (Matrix Gen. Outside)

The result for the second Python dataset is presented in this section, starting with the epochs number.

Margin Error for the epochs optimization process:

- 5 epochs: 0.087
- 10 epochs: 0.089
- 25 epochs: 0.084
- 50 epochs: 0.086

Margin Error for the batch-size optimization process:

- 32 as batch-size: 0.093
- 64 as batch-size: 0.084
- 128 as batch-size: 0.088
- 256 as batch-size: 0.090

Margin Error for the learning-rate optimization process:

- 0.0001 as learning rate: 0.091
- 0.001 as learning rate: 0.087
- 0.01 as learning rate: 0.085
- 0.1 as learning rate: 0.098

Final results on the optimization process for the second Python dataset:

- Epochs: 25
- Batch-size: 128
- Learning Rate: 0.001
- Margin Error: 0.080
- Loss: 0.0066

7.9 PKG and RAM Energy

In the energy consumption analysis, the sum of PKG_energy and RAM_energy is traditionally used to calculate the overall energy usage. In this new section, we will construct separate models for each dataset. These models will focus on predicting the energy consumption of PKG and RAM components.

By adopting this approach, we aim to gain more insights into the specific energy usage patterns and enhance the accuracy of our predictions. For each dataset, we will utilize the optimized model discovered through the hyperparameter optimization process outlined in the previous section.

7.9.1 PKG Energy

First Python dataset:

- Margin Error: 0.040
- Loss: 0.0034

Java dataset:

- Margin Error: 0.080
- Loss: 0.0075

Combined dataset:

- Margin Error: 0.038
- Loss: 0.0036

Second Python dataset:

- Margin Error: 0.0747
- Loss: 0.0046

Overall, these findings suggest that the PKG Energy predictions are generally accurate, with some variations depending on the specific dataset.

7.9.2 RAM Energy

First Python dataset:

- Margin Error: 0.009
- Loss: 0.0000256

Java dataset:

- Margin Error: 0.022
- Loss: 0.0000709

Combined dataset:

- Margin Error: 0.011
- Loss: 0.0000908

Second Python dataset:

- Margin Error: 0.004
- Loss: 0.0000154

Overall, these findings indicate that the RAM Energy predictions are consistently accurate across different datasets, demonstrating the model's effectiveness in capturing the energy consumption patterns of RAM in various programming languages.

7.9.3 Interpretation

The smaller margin errors and losses observed for RAM Energy predictions across the datasets suggest that the energy variation by the RAM for different matrix multiplications is relatively smaller compared to the PKG component. This implies that the RAM's energy consumption may be more consistent and less dependent on specific matrix multiplication tasks.

On the other hand, the PKG component's energy consumption may be influenced by factors beyond just the matrix multiplication task, leading to more significant variations in energy usage. This could contribute to the higher margin errors and losses observed for PKG Energy predictions.

In summary, the error rates for RAM Energy predictions compared to PKG Energy predictions suggest that the RAM component's energy consumption exhibits less variation and is relatively more consistent across different matrix multiplication scenarios.

8. NEURAL NETWORK BASED ON THE ENERGY CONSUMPTION

In the previous section, we employed a method of finding the best solution by testing various hyper-parameters and selecting the combination that resulted in the lowest loss and margin error.

In this section, we will focus specifically on the Combined dataset to determine the optimal solution. We will evaluate the margin error and energy consumption during the neural network's training.

By analyzing these metrics during the training process, we aim to identify the combination of hyper-parameters that minimizes the margin error but ensures efficient energy utilization.

This analysis enables us to find the most suitable combination of hyper-parameters that maximizes the accuracy of predictions while minimizing energy consumption during the training process.

8.0.1 *Constructing the dataset*

As described in section 6, we used the perf command to measure the energy consumption that takes the training of the neural network in the HPC. Moreover, we measure the energy consumption for each hyper-parameters combination between the epochs number (5, 10, 15, 20, 25, 30, 35, 40, 45, 50), the batch size (32, 64, 128, 256) and the learning rate (0.001, 0.001, 0.01, 0.1).

We ended up with a dataset of 10*4*4 = 160 rows, where each row is a unique combination of hyper-parameters. Here are the columns of this dataset: 'File', 'Epochs', 'Batch Size', 'Learning Rate', 'Total Error Margin', 'PKG_Energy', 'RAM_Energy', 'Cache Misses', and 'Time'. Similarly, an 'Energy' column was created by summing the ones for the PKG and RAM.

8.0.2 *Focus on the data*

The first step performed on the dataset was to identify the element with the minimum and maximum energy consumption. Additionally, we calculated the average energy usage across the dataset and selected the element with energy closest to this average. This allowed us to make meaningful comparisons and analyze how individual elements' energy consumption deviated from the average.

Minimal

- Energy: 451.65
- Epoch: 5
- Batch: 256
- Learning Rate: 0.0001
- Margin Error: 0.852
- Loss: 0.5505

Average

- Energy: 2450.08
- Epoch: 45
- Batch: 128
- Learning Rate: 0.001
- Margin Error: 0.073
- Loss: 0.0066

Maximum

- Energy: 8775.39
- Epoch: 50
- Batch: 32
- Learning Rate: 0.0001
- Margin Error: 0.0508
- Loss: 0.5889

We gained insights into the range and distribution of energy consumption by examining the minimum, maximum, and average energy values. This initial analysis provided a foundation for further investigations and comparisons related to energy consumption patterns in the dataset.

8.0.3 *Optimal Solution*

Our focus is now to select the combination that achieves the smallest margin error while minimizing energy consumption. For that, we selected among all the dataset the 10 smallest margin error, and we took the one that was using the least energy. Please find below the graph depicting the relationship between the 10 smallest margin errors and their corresponding energy values. The circled data point represents the optimal solution, which excels in terms of both energy efficiency and error performance.

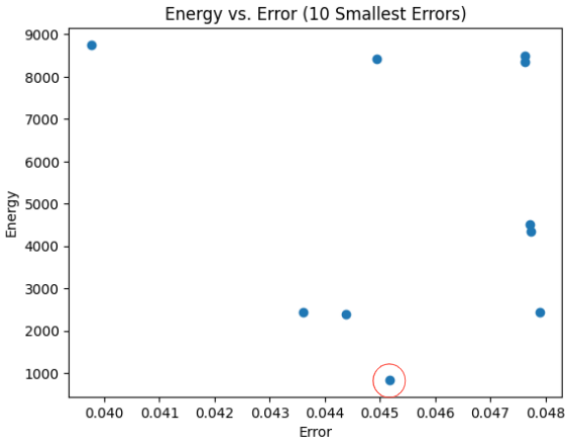

Figure 3:

This dual focus on prediction performance and energy efficiency allows us to identify an hyperparameter combination that balance accuracy and energy consumption.

This optimal solution demonstrates the effectiveness of our approach in finding an hyperparameter combination that strikes an optimal trade-off between energy efficiency and error performance.

Optimal Solution based on the Error/Energy

- Energy: 836.41
- Epoch: 15
- Batch: 256
- Learning Rate: 0.01
- Margin Error: 0.045
- Loss: 0.0043

The idea is to now compare this solution with the one we found by optimizing locally the hyper-parameters:

Optimal Solution based on the Error/Loss

- Energy: 4275.95
- Epoch: 50
- Batch: 128
- Learning Rate: 0.001
- Margin Error: 0.048351
- Loss: 0.0045

8.0.4 Interpretation

Comparing these two solutions, we can draw several conclusions. Firstly, the error/energy-based solution achieves a lower energy consumption of 836.41, significantly less than the energy consumption of 4275.95 in the error/loss-based solution. This indicates that the former configuration utilizes energy more efficiently.

Additionally, the error/energy-based solution exhibits a margin error of 0.045 and a loss of 0.0043, while the error/loss-based solution has a slightly higher margin error of 0.048351 and a comparable loss of 0.0045. Although the error/energy-based solution has a marginally lower margin error, the difference is relatively small.

Overall, we can conclude that the error/energy-based solution offers a more energy-efficient configuration while maintaining a reasonably low margin of error and loss. This highlights the importance of considering energy consumption alongside prediction performance when optimizing hyperparameters for neural networks.

9. CONCLUSIONS

To conclude, we reached our research goals and managed to find an optimal solution to the training with respect to energy consumption. Exploring the energy consumption aspect of programming was really interesting. Yet, much more can be done on this topic.

We recognise that the matrix range being random was a weird decision and it would have been better to pre-select different ranges like we did for the size like for example between 0 and 1 or -1000 to 1000 to see more clearly if the range of values affects the power usage. We also did not get to play around with non-square matrices for comparison due to lack of time.

What we can remember is that Java uses more energy than Python for a similar task and that the hyperparameter choice affect the total amount of energy consumed during the training.

10. ACKNOWLEDGMENTS

We would like to address a special thanks to our tutors T. Fischbach and Dr. E. Kieffer for their disponibility, guidance and support provided throughout the project.

11. REFERENCES

[1] WION Web Team, Last Updated Nov 15, 2021, *Pressure on global energy crisis deepens with population growth,* WION, <wionews.com>. <https://www.wionews.com/india-news/pressure-on-global-energy-crisis-deepens-with-population-growth-422913/>.

[2] ULHPC Technical Documentation, Last Updated May 22, 2023, *Iris Compute Nodes,* <hpc-docs.uni.lu>. <https://hpc-docs.uni.lu/systems/iris/compute/>.