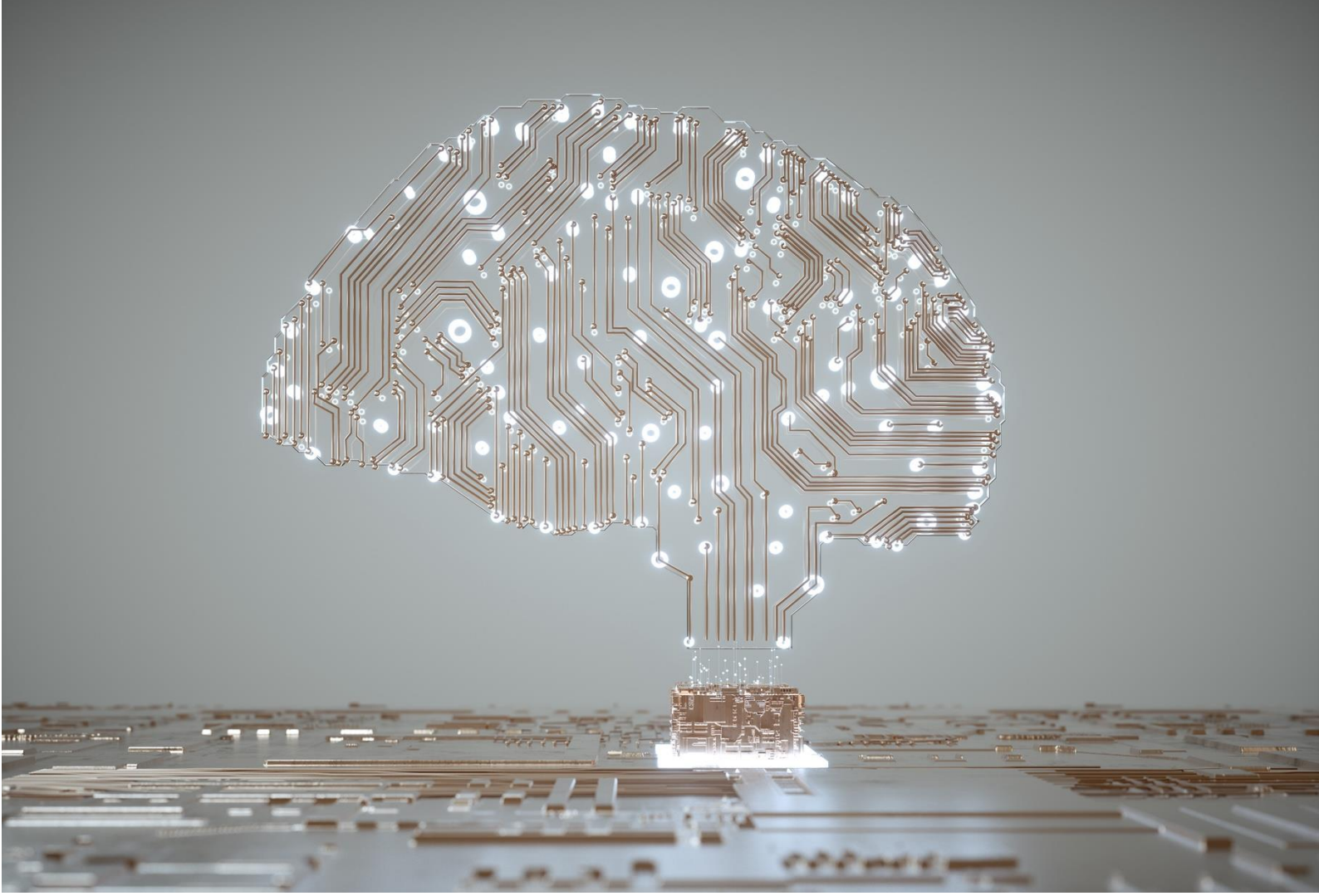


Bachelor's Thesis
Faculty of Technology, Art & Design
Oslo Metropolitan University

Spring 2022

Alexander Soudaei Mohamed Elmi Mikael Tsechoev
Zishan Khan Ahmed Osman

25-May-2022

OsloMet – Oslo Metropolitan University
Faculty of Technology, Art & Design
Department of Mechanical, Electronic & Chemical Engineering
Mail Address: Postboks 4 St. Olavs plass, 0130 Oslo
Phone Number: 67 23 50 00

Project number
2022-14

Availability
Public

Bachelor's Project

Title	Date
Neuroevolution Approach with Application to Energy Consumption Forecasting	**25.05.2022**
	Number of pages **118**
Participants **Alexander Soudaei** **Ahmed Osman** **Mikael Tsechoev** **Mohamed Elmi** **Zishan Khan**	Internal Supervisor **Jianhua Zhang**
	Client **OsloMet**

Abstract

The structure ANNs (Artificial neural networks) are usually set manually using a trial-and-error method. We have optimized the structure of deep neural networks using algorithms from two subsets of machine learning called evolutionary algorithms and swarm intelligence. To test our method, we have applied it to an energy consumption problem.

Energy consumption is something that concerns us all. The reason why we have chosen to solve a problem related to energy consumption is because an accurate energy prediction could give insight to make better decisions on energy purchase and generation. It will also have an impact on preventing overloading and make it possible to store energy in a more efficient way.

Our experiments gave better results than the other stand-alone algorithms that we've compared with, but the results were not easy to achieve.

Keywords
Energy Prediction
Deep Learning
Evolutionary Algorithm

Preface

In this report, we present our research project that was given by the Oslo Metropolitan University. As a group, we've been fascinated by the world of Artificial Intelligence and want to explore the exciting and rapidly evolving field further at Master-level in the future.

In addition to our fascination with this field, we also wanted to conduct research work by working with a like-minded group of people.
For this project, we aim to construct a solution that gives better prediction results than most – if not all – of the existing models.

All the chapters in this report are presented chronologically to give the readers a sense of our process of model building and validation in this research.

Lastly, we would like to thank our university - OsloMet and its many professors who have helped us throughout our study and especially Prof. Jianhua Zhang, who guided us on a fruitful and instructive path for our research project.

Contents

Summary 12
1. Introduction 13
1.1. Project group 13
1.2. Project Description 13
1.3. Libraries used for programming 14
1.4. Project Log 14
1.5. Tools used in our project 15
2. Development documentation 16
2.1. Evolutionary algorithms and Neuroevolution approach 16
2.1.1. Initializers 17
2.1.2. PSO – Particle Swarm Optimization 18
2.1.3. GA & MA – Genetic Algorithm & Memetic Algorithm 23
2.1.4. DE – Differential Evolution 27
2.1.5. NEAT – Neuroevolution of Augmenting Topologies 31
2.2. Dataset description 31
2.2.1. Data source 32
2.2.2. Attributes 32
2.2.3. Statistical analysis of the dataset 35
2.2.4. Feature importance results 41
2.3. Survey of existing work 43
2.4. Our idea of combining EA/SIA with DNN 47
2.5. Determining the details of our implementation 48
2.5.1. Batch normalization 48
2.5.2. Activation functions 48
2.5.2.1. Sigmoid function 49
2.5.2.2. ReLU function 49
2.5.2.3. TanH function 50
2.5.2.4. Softplus function 51
2.5.2.5. Swish function 52
2.5.2.6. SELU function 53
2.5.3. Testing activation functions 53
2.5.4. Adadelta optimizer 54

2.5.5. CPU & GPU 55
2.5.6. Multithreading 56
2.6. Running the algorithms on a server 57
3. Implementing Deep Neuroevolutionary algorithms and Energy prediction results 58
3.1. DNN combined with PSO 58
3.1.1. Encoding 58
3.1.2. PSO search 58
3.1.3. Two-phased training process 64
3.1.4. PSO search results 66
3.1.5. Summary of PSO-DNN experiment 70
3.2. DNN combined with DE 70
3.2.1. Encoding 70
3.2.2. DE search 71
3.2.3. Two-phased training process 76
3.2.4. DE search results 78
3.2.5. Summary of DE-DNN experiment 83
3.3. DNN combined with MA 84
3.3.1. Encoding 84
3.3.2. MA search 84
3.3.3. Two-phased training process 89
3.3.4. MA search results 91
3.3.5. Summary of MA-DNN experiment 95
4. Discussion 96
4.1. Comparison with existing work – Accuracy 96
4.2. Comparison with existing work – Computational Complexity 97
4.3. Feature importance 97
5. Conclusions and Future work 98
6. References 99
Appendices 102
A.1 Glossary 102
A.2 Acronyms 105
A.3 List of symbols 106
A.4 Taxonomy 107

A.5 WSN (Wireless Sensor Networks) 108
A.6 Information of the metrics used in the results 109
A.7 Gantt-Chart 111
A.8 WBS – Work Breakdown Structure 112
A.9 Most important parts of our code 113
A.9.1 Step function from PSO, summarizing what happens in a generation/iteration in PSO 113
A.9.2 Step function from DE, summarizing what happens in a generation/iteration in DE 114
A.9.3 Step function from MA, summarizing what happens in a generation/iteration in MA 114
A.9.4 Objective class, used for converting position vector into an DNN for training and evaluation 115
A.9.5 Using the gpustat command to ensure that all three GPUs are being utilized. The green numbers are the utilization in percentage of each GPU. 115
A.9.6 Assigning a specific GPU to be used by tensorflow, using os library in python .. 116
A.9.7 Assigning a specific set of CPU cores to a task, using the taskset command 116
A.9.8 Assuring that the assigned CPU cores (0-20) are being utilized, using the top command. 117

List of Tables

Table 1: An overview of Python libraries used in the project 14
Table 2: A log of important dates during the project and their description 14
Table 3: Names and descriptions of the tools used in the project 15
Table 4: explanations of different parameters used in both GA and MA 24
Table 5: Observations, results from different tests done with GA and MA 25
Table 6: statistics of how both MA and GA did in our tests 26
Table 7: Parameters of the DE algorithm 27
Table 8: Dataset description before feature engineering 31
Table 9: Dataset description after feature engineering 32
Table 10: attribute information from the dataset 32
Table 11: Names of ML models with their descriptions. 44
Table 12: Results of all tested activation functions, based on validation MSE 54
Table 13: Time consumption when using different numbers of threads at once to complete a total of 60 threads 56
Table 14: Overview of the parameter settings used in the PSO search 58
Table 15: Summary of results from the PSO search 66
Table 16: Table showing the metrics of the best architecture in train, validation, and test sets 67
Table 17: The statistics of the testing errors defined by the actual value minus the predicted value on every instance in the test set. 67
Table 18: The statistics of the absolute values of the errors due to too low and too high predictions. 69
Table 19: Overview of the parameter settings used in the DE search 71
Table 20: Summary of results from the DE search 78
Table 21: Table showing the metrics of the best architecture in train, validation, and test sets 79
Table 22: The statistics of the testing errors defined by the actual value minus the predicted value on every instance in the test set. 80
Table 23: The statistics of the absolute values of the errors due to too low and too high predictions. 82
Table 24: Overview of the parameter settings used in the MA search 84
Table 25: Summary of results from the MA search 91
Table 26: Table showing the metrics of the best architecture in train, validation, and test sets 92
Table 27: The statistics of the testing errors defined by the actual value minus the predicted value on every instance in the test set. 92
Table 28: The statistics of the absolute values of the errors due to too low and too high predictions. 95
Table 29: The results of the algorithms measured by different metrics 96
Table 30: Glossary definitions 102

Table 31: Table of acronyms 105
Table 32: Table of symbols 106
Table 33: Algorithms split into categories 107

List of Figures

Figure 1: This figure shows Random-, Quasirandom and – Sphere Initializer from left to right. Each of them is shown in a 2-dimensnional search space with 100 particles each. (Kneusel, 2021, p.22). Copyright by author. 17
Figure 2: Shows that particles are moving towards the “Leader” which possesses the best position in the search space. The rest of the particles are also trying to find the same position since no information is being shared (Fouad, M.M et.al, 2020). CC by 4.0. 19
Figure 3: Shows that particles are moving around (Fouad, M.M et.al, 2020). CC by 4.0. 19
Figure 4: Shows the decaying of linear inertia with respect to the number of iterations. It's also compared to a Sigmoid which is an S-shaped activation function (Dhirf, H., et. al., 2019, p. 4). Copyright by ResearchGate. 21
Figure 5: PSO swarm position updates clockwise from upper left: initial, iteration 12, iteration 24, iteration 30..... 22
Figure 6: An overview of what GA population consist of (Mallawaarachchi, V. 2017). Copyright by author 23
Figure 7: An example of crossover (Kneusel, 2021, p.79). Copyright by author. 23
Figure 8: MA swarm position updates clockwise from upper left: initial, iteration 12, iteration 24, iteration 30..... 26
Figure 9: Structure of DE 29
Figure 10: DE swarm position updates clockwise from upper left: initial, iteration 12, iteration 24, iteration 30..... 30
Figure 11: Overview of the data in terms of data types and missing values 34
Figure 12: Statistics of the “appliances” column in the dataset 35
Figure 13: Bar plots showing the distribution of all the variables in the dataset 35
Figure 14: Histogram showing the distribution of appliance values in the dataset..... 36
Figure 15: Boxplot showing the distribution of values in the dataset 36
Figure 16: Bar plot showing the appliance values in the dataset 37
Figure 17: Correlation matrix showing the correlation between the variables in the dataset .. 38
Figure 18: Scatterplots of the five most correlated features to appliances energy consumption: hour, lights, T2, T6, RH_out 38
Figure 19: Energy consumption of appliances with different dependent variables. From left to right: lights & hour, T2 & hour, T6 & hour. 39
Figure 20: Energy consumption of appliances with different dependent variables. From left to right: RH_out & hour, lights & T2, lights & T6. 39
Figure 21: Energy consumption of appliances with different dependent variables. From left to right: RH_out & lights, T2 & T6, RH_out & T2. 39
Figure 22: Appliances energy consumption as a dependent variable of RH_out and T6 40
Figure 23: Leftmost scatterplot in Figure 20 shown from a bird’s eye view 40
Figure 24: RF - Random Forest feature importance results and Lasso feature importance results respectively. Where the X-axis is relative importance, and the Y-axis contains the different features 41

Figure 25: DTR - decision tree regressor feature importance and GBM- gradient boosting machine feature importance respectively. where the X-axis is relative importance, and the Y-axis contains the different features 42
Figure 26: combined average feature importance. a collection of the different feature importance results 42
Figure 27: Comparison of RMSE and R2 – values of trained models (Candanedo et al., 2017, p. 90). Copyright 2017 by Elsevier B.V. 45
Figure 28: Results of each algorithm in both training and testing sets. (Candanedo et al., 2017, p. 91). Copyright 2017 by Elsevier B.V. 46
Figure 29: GBM performance with different parameters. (Candanedo et al., 2017, p. 91). Copyright 2017 by Elsevier B.V. 46
Figure 30: Flowchart showing the steps of combining the algorithms with DNN 47
Figure 31: Sigmoid function plot 49
Figure 32: ReLU function plot 50
Figure 33: Plot of TanH function. 51
Figure 34: Softplus function plot 52
Figure 35: Swish function plot 52
Figure 36: SELU function plot 53
Figure 37: A picture of a CPU without the metal plate cover (Leather, A., 2021). Copyright 2022 by Forbes Media LLC. 55
Figure 38: A picture of a GPU without a heatsink and fans, which are generally included (Shilov, A., 2019). Copyright 2022 by Anandtech. 56
Figure 39: Illustrates the change of the worst swarm MSE over 30 generations 61
Figure 40: Illustrates the change of the mean swarm MSE over 30 generations 62
Figure 41: Illustrates the change of the best swarm MSE over 30 generations 63
Figure 42: Illustrates the change of the best architecture MSE over epochs in the first training phase using SGD optimizer 64
Figure 43: Illustrates the change of the best architecture MSE over epochs in the second training phase using Adadelta optimizer 65
Figure 44: Visualization of the best architecture found by PSO (30-47-31-30-43-35-44-1) ... 66
Figure 45: The predicted vs. actual energy use. 68
Figure 46: The predicted vs. actual energy use (only the first 1000 instances in test dataset shown here as a sample). 68
Figure 47: The distribution of the testing errors. 69
Figure 48: Illustrates the change of the worst swarm MSE over 30 generations 73
Figure 49: Illustrates the change of the mean swarm MSE over 30 generations 74
Figure 50: Illustrates the change of the best swarm MSE over 30 generations 75
Figure 51: Illustrates the change of the best architecture MSE over epochs in the first training phase using SGD optimizer 76
Figure 52: Illustrates the change of the best architecture MSE over epochs in the second training phase using Adadelta optimizer 77
Figure 53: Visualization of the best architecture found by DE (30-36-33-53-46-1) 78
Figure 54: The predicted vs. actual energy use. 81

Figure 55: The predicted vs. actual energy use (only the first 1000 instances in test dataset shown here as a sample). 81
Figure 56: The distribution of the testing errors. 82
Figure 57: Illustrates the change of the worst swarm MSE over 30 generations 86
Figure 58: Illustrates the change of the mean swarm MSE over 30 generations 87
Figure 59: Illustrates the change of the best swarm MSE over 30 generations 88
Figure 60: Illustrates the change of the best architecture MSE over epochs in the first training phase using SGD optimizer 89
Figure 61: Illustrates the change of the best architecture MSE over epochs in the second training phase using Adadelta optimizer 90
Figure 62: Visualization of the best architecture found by MA (30-33-36-50-1) 91
Figure 63: The predicted vs. actual energy use. 93
Figure 64: The predicted vs. actual energy use (only the first 1000 instances in test dataset shown here as a sample). 94
Figure 65: The distribution of the testing errors. 94
Figure 66: Shows the whole connection of a typical WSN (Kumar, S., 2021). Copyright by GeeksForGeeks. 108
Figure 67: Shows MAE between a datapoint and the regression line (Pascual, C., 2018). Copyright 2022 by Dataquest. 110

List of Equations

Equation 1: Relationship between inertia, social component & cognitive component of the algorithm. 20
Equation 2: Relationship between xij, σ, and x. 21
Equation 3: Formula for position updates in Bare Bones PSO 21
Equation 4: Sample objective function 25
Equation 5: Formula for mutation vector v 29
Equation 6: Formula for candidate vector u 29
Equation 7: Sigmoid function 49
Equation 8: ReLU function 49
Equation 9: TanH function 50
Equation 10: Softplus function 51
Equation 11: Swish function 52
Equation 12: SELU function 53
Equation 13: Formula for RMSE. 109
Equation 14: Formula for R^2 109
Equation 15: Formula for MAE 110
Equation 16: Formula for MAPE 110

Summary

Energy consumption is something that concerns us all. We have based our work on trying to predict energy in a new way. The reason why we have chosen to solve a problem related to energy consumption is because an accurate energy prediction could give insight to make better decisions on energy purchase and generation. It will also have a remarkable impact on preventing overloading and make it possible to store energy in a more efficient way.

In this thesis, we have used three combinations of algorithms. The algorithms we have used consists of two evolutionary algorithms called Memetic algorithm and Differential evolution. The third one is an optimization algorithm called Particle Swarm Optimization, which belongs to another subset of machine learning called swarm intelligence. Each one of these algorithms were combined with a deep neural network to find its best architecture.

To see if our combinations of algorithms could provide better results than what is found in current research, we used different metrics for result evaluation. The metrics which were R^2, RMSE, MAE and MAPE have also been used by researchers with related work.

Our algorithms provided better results in three of those metrics. This is further described in the result section of this thesis. We have also applied 4 different machine learning algorithms to our dataset with the purpose of figuring out which features had the most impact on our results. Each of these algorithms have their own metrics for determining feature importance and therefore we decided to utilize multiple algorithms to get a bigger insight.

1. Introduction

1.1. Project group

Students/Researchers

Alexander Soudaei
Engineering - Electronics & IT

Ahmed Osman
Engineering - Electronics & IT

Mohamed Elmi
Engineering - Electronics & IT

Mikael Tsechoev
Engineering - Electronics & IT

Zishan Khan
Engineering - Electronics & IT

Internal supervisor

Prof. Jianhua Zhang
OsloMet Artificial Intelligence Lab, Department of Computer Science, Oslo Metropolitan University.

1.2. Project Description

Most of the current approaches to Deep Neural Networks (DNNs) require that the developer manually adjusts the number of hidden layers and the number of nodes in each hidden layer to make the network perform better. Different types of activation functions like ReLU, Sigmoid, SoftMax, and Tanh are also used for the output, and hidden layers to give the ideal output. This can be time consuming, nor is it possible to get a concrete answer for getting the best structure of the DNN. Our goal is to try to use SI algorithm such as PSO, and EAs such as MA, and DE to make the best structure of an DNN.
We use Python for coding in our project. Python is a multifunctional programming language which makes it well suited for AI modeling tasks due to its rich and powerful libraries.

1.3. Libraries used for programming

Table 1: An overview of Python libraries used in the project

Name of library	**Usage**
NumPy (Numerical Python)	Working with arrays, matrices mathematical functions
Pandas	Working with datasets. It includes functions for analyzing, cleansing, exploring, and manipulating data. Used for data analysis which is practical for handling data.
TensorFlow	Primarily used for deep learning but also for machine learning. Used to design DNN.
Matplotlib	Used for visualizing our modelling results
Seaborn	Plotting of data-analytics
Sklearn	Importing of train-test-split and data-scaling
Keras	TensorFlow
Threading	To do multithreading
Queue	Used to fetch results from the multithreading process.
os	Specifying which GPU to use

1.4. Project Log

Table 2: A log of important dates during the project and their description

Date	**Description**
15.10.2021	Formation of development team
15.11.2021	Project sketch/outline
23.11.2021	First meeting with the supervisor from OsloMet
14.01.2022	Start preliminary work on project
15.01.2022	Sign contract with OsloMet
31.01.2022 -	Bi-weekly meetings with the supervisor from OsloMet
01.02.2022 - 21.02.2022	Develop an appropriate neuroevolutionary algorithm
20.04.2022	Deliver first draft of the project report
25.05.2022	Deliver project report and project poster

1.5. Tools used in our project

Table 3: Names and descriptions of the tools used in the project

Tool	**Usage & description**
Zoom and Teams	Video meetings with supervisor and group members.
Facebook Messenger	Communication between group members
Discord	Project work, communication between group members. Sharing code, delegating tasks. Sharing research papers and useful videos with group members.
YouTube	Used for watching videos related to programming and supplementary explanation for algorithms.
Email	Formal communication with supervisor. Used to send e-books between group members.
Spyder, PyCharm, Jupyter Notebook	Used to write and run Python code
External device (PC)	Used to work on the project
Stack Exchange	Forum for sharing troubleshooting for code. Used when we got errors on our own code to effectively solve it.
Microsoft Word on Google Docs	Writing project thesis and taking notes from meetings with supervisor and saving it all on the cloud for easy access, protection, and effective file sharing.
Kaggle	Used to download the dataset.
IEEE Xplore, Science Direct, arxiv.org	Websites to find research papers for inspiration for our own work.
UCI Machine Learning Repository	To search for and download datasets we can use to apply our own algorithms.

2. Development documentation

In this chapter we will discuss the research work that has been done prior to the solution, and the dataset used in our solution - its source, attributes, feature importance and statistical analysis. In addition to this we will discuss related work, the idea for our solution and lastly determining the details of our style of implementation.

2.1. Evolutionary algorithms and Neuroevolution approach

Before we took on this research-project we had, as a group, some experience from two different AI-related classes that is offered by the university in addition to our Python-course. One of the classes was theory-based introduction to AI, where we discussed the core principles of data such as data- wrangling and feature engineering among others, the basic idea behind the classifying algorithms into supervised and unsupervised learning, with some labs that were based on regression-model.

The other subject was about completing an AI project that would be submitted to a supervisor in the end. In this subject we gained some experience into how to introduce ourselves into a new field, new tools, and understand the inner workings of building an AI-algorithm and what goes in it.

This was a very integral part of our work and gave an intuition for what we can eventually explore, learn, and implement in a relatively short period amount of time.
When we started our research in January 2022, we had a meeting with our supervisor to get recommendations on how to start with the research. First, we had to build a knowledgebase from two fronts.

We started reading some research papers to get an idea of how we can come up with a better solution to a specific problem. Using different algorithms and combining different algorithms are two methods that can be used to find a potentially better solution.
While in some cases the problem already had a solution with a high degree of accuracy – in which case we would not bother to find any alternative solutions.

The second front we had to gain more knowledge on was evolutionary algorithms and swarm intelligence algorithms. In the field of AI, there are plenty of algorithms that can be categorized in different ways. Our goal for this part of the research was to get an overview and insight into the characteristics of different algorithms, and thus we've built a so-called Algorithm-portfolio. In this algorithm-portfolio, we have picked several algorithms that met certain requirements for our project. These requirements are:

- The algorithm must be able to work on a problem on its own and at the same time be combined with other types of algorithms.
- The algorithm must be able to provide a specific solution to the application problem considered.
- The algorithm should have different use-cases so we can understand, from the research-papers, the possibilities with this algorithm.

2.1.1. Initializers

Before the particles start wandering through the search space (see Appendix A.1 Glossary) to search for the optimal solutions, they need to have start positions. If the initialization is not proper, then the particles might explore unnecessary areas and fail to find the optimum solution. It is essential to initialize the particles in any metaheuristic (see Appendix A.1 Glossary) algorithm. (Bangyal et al., 2021, p.8)

We use initializers that operate differently, to give start positions to the particles. The different types of initializers we have used are Random Initializer, Quasirandom Initializer and Sphere Initializer.

Random Initializer, as indicated by the name, gives the particles random positions throughout the search space within the bounds (see Appendix A.1 Glossary) we define. This means that some areas in the search space might be more concentrated than others, while other areas might not be covered thoroughly.

Quasirandom Initializer uniformly scatters the particles, so the distance between each particle is approximately the same or at least more even, unlike Random Initializer. A Halton process is used to generate the distribution of particles within this initializer. “The Halton sequence is generated by selecting a base, b, usually a small prime, and recursively dividing the unit interval into b sub-intervals. The endpoints of these sub-intervals are then output from smallest to largest to create the space filling sequence”. The process works best when the bases are primes, as we have used in our programming. (Kneusel, 2018, p.76)

“The Sphere Initializer selects random points on a ndim-dimensional hypersphere” (Kneusel, 2021, p.21). The particles are on the edge of this hypersphere, which is bounded by the search space.

Figure 1 shows a plot of each one of the initializers, with bounds [0,1], as shown in the plot below.

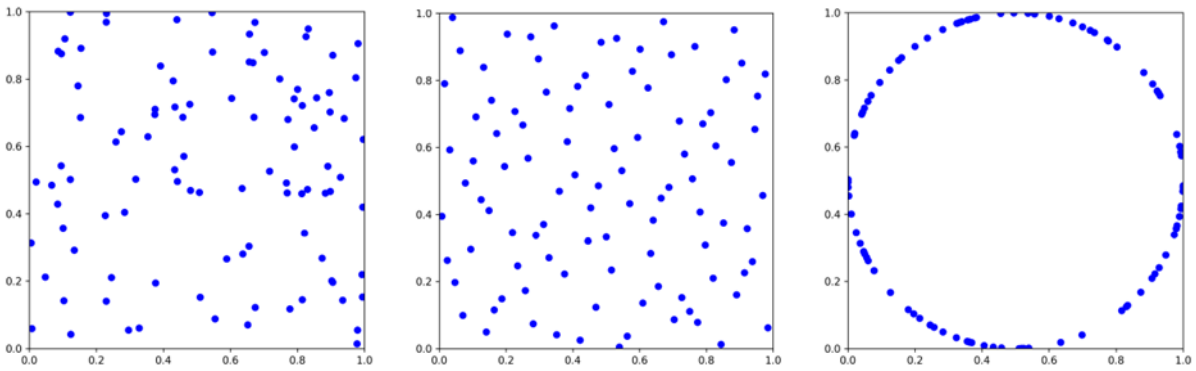


Figure 1: This figure shows Random-, Quasirandom and – Sphere Initializer from left to right. Each of them is shown in a 2-dimensnional search space with 100 particles each. (Kneusel, 2021, p.22). Copyright by author.

2.1.2. PSO – Particle Swarm Optimization

We started to build our algorithm-portfolio with the Particle Swarm Optimization algorithm (PSO) which falls under the umbrella of optimization algorithms. PSO is a population-based, stochastic gradient descent-based algorithm which means it is an iterative optimization algorithm with the main goal being that the population finds the position corresponding to the minimum cost function, also known as error function.

An example of how gradient descent works could be to imagine a blindfolded person climbing down a mountain. The first step would be to get a feel of the ground and start taking careful steps as they descend. In this process, the blindfolded person must make sure to neither take too long nor too short steps, as the former can lead to imprecise navigation, and the latter may take a vast amount of time before the descension is over and the person is on the ground.

How far apart the steps taken by the person to descend are the equivalent of what is called *Learning Rate (α)* also called *Alpha*, and the lowest point of descension is called *minimum cost (error) function*.
Reaching the optimal minimum cost function is our goal when PSO-algorithm is used. The Cost function is better known as *Objective function* in PSO.

The PSO algorithm has several particles that are confined within a defined search space. These particles communicate with each other to learn about their surroundings (search space) to find the best position. Each particle has two types of information:

1. The best position the particle itself found in the search space.
2. The best position the swarm has found.

Here we can clearly understand that the former is about the data that a specific particle has gathered to judge whether this is the best position or not. This is the cognitive part of the particle swarm intelligence since the particle is using and relying on its own findings.

The second is about the sharing of information between the particles. This is the social part, where the particle is communicating with the other particles to check if those other particles have better or worse positions. The end goal is that by thinking independently and at the same time communicating with the neighboring particles, the particles will converge towards the global minimum (see Appendix A.1 Glossary).

There are a lot of different variants of the PSO-algorithm, but for our project we focused on Canonical- and Barebones PSO as they seem to be targeting any problem that can be represented as a search space, and solutions that are represented as particles. Canonical PSO has a set of parameters that need to be defined and adjusted to make the algorithm functional.

To start, there is the position of a particle *i* referred to as x_i. This position is a ndim-dimensional vector, where ndim represents the number of dimensionalities of the search space. Each particle has two vectors. The first is v_i which represents particle's velocity. It's used to update the particle's position for the next iteration. $\hat{x}_i$ represents the best position the particle has found on its own.
Like many swarm algorithms, there is a problem that needs to be addressed when it comes to how a swarm goes about its work. This problem arises from the need to balance exploration

and exploitation. Exploitation means that the particles in the swarm search locally. A particle would solely think on its own and examine all the next steps from its current point. After that it will take the step that is closest to the minimum objective function value. Figure 2 shows Exploitation.

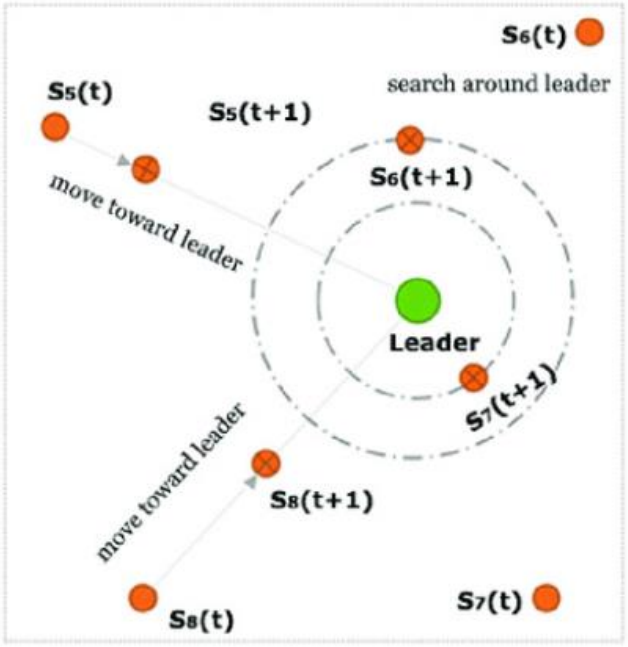


Figure 2: Shows that particles are moving towards the "Leader" which possesses the best position in the search space. The rest of the particles are also trying to find the same position since no information is being shared (Fouad, M.M et.al, 2020). CC by 4.0.

Exploration means that the swarm searches globally throughout the whole search space. The members of the swarm (the particles) will move around the search space to find the best objective function value. If the particle moves to a position that has a less optimal objective function value compared to the previous one, the particle will move back to that previous position and repeat this process. Figure 3 illustrates exploration in its purest form.

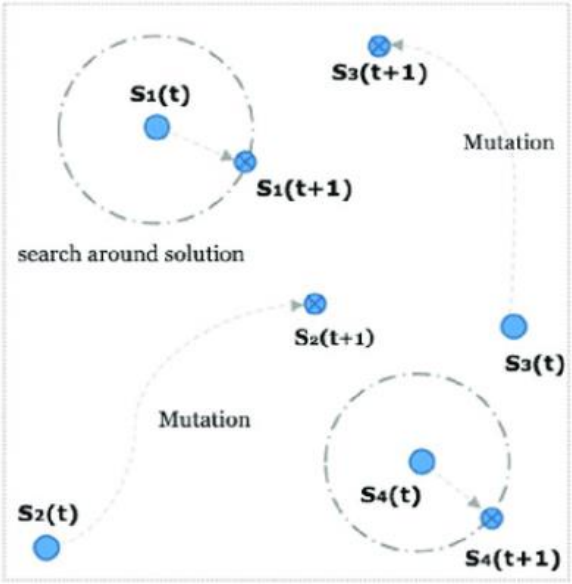


Figure 3: Shows that particles are moving around (Fouad, M.M et.al, 2020). CC by 4.0.

To strike an adequate balance between exploration and exploitation, c_1, c_2, and w are the parameters to adjust. c_1 and c_2 are defined as $c_1 = c_1 * U_1$ and $c_2 = c_2 * U_2$. They are

randomly generated for each iteration/generation of the swarm, where the U-components are a random number between the intervals of [0,1). c_1 and c_2 are also referred to as *Acceleration coefficients* as they manage the stochastic effects of the cognitive- and social components on the overall velocity of a particular particle (Engelbrecht, 2007, p.313)
Both the cognitive- and social-part are represented by the equations $c_1(\hat{x}_i - x_i)$ and c_2 $(g^i - xi)$ respectively. The former of the two equations shows that c_1 is producing a result with the difference of the best position the particle has found on its own and the current position.
The latter shows that c_2 is producing a result with the global best position (hence the social component) and the current best position.

There is a relationship between the Acceleration coefficients and their effect on the particles of the swarm. If c_1 = c_2= 0, particles keep running at their current speed until they hit the boundary of the search space given that the inertia (w) is not in use. If c_1 > 0 and c_2 = 0, all the particles are independent, which would mean that a local search is being performed. In the taxonomy of global optimization algorithms, this cognitive-focused approach makes the algorithm fit in the mountaineer category (see Appendix A.4 Taxonomy).

If c_1 = 0 and c_2> 0, all the particles would converge to a single point turning it into a single stochastic hill-climber. This is due to the extreme socializing & communication that happens between particles compared to the cognitive and independent searching which is practically non-existent.
For unimodal problems with a plane search space, a larger social component will be efficient, while rough, multi-model search spaces may find a larger cognitive component more beneficial (Engelbrecht, 2007, p.313).

To control the movement of the swarm we use Inertia (w). Inertia is normally a number between [0,1] and is typically never below 0.5. The purpose of the inertia is that it reduces the velocity of a particle for the next iteration and since we are under the impression that as the search continues, it is beneficial that w decreases as we get closer to the best position. For the PSO we defined two types of inertia, namely Linear inertia, and Random inertia.

Linear inertia reduces linearly w linearly as the number of swarm iterations increase (picture of linear inertia) Random inertia differs from linear inertia in that instead for linear decrease as iterations go by, it decreases with a random value. For our PSO we've used linear inertia as it's the most popular one and can function based on actual mathematical concepts instead of randomness.

Parameters c_1, c_2, and w have a relationship and therefore affect each other. This can be presented as a mathematical formula:

$$w > \frac{1}{2}(c_1 + c_2) - 1.$$

Equation 1: relationship between inertia, the social component, and the cognitive component of the algorithm.

Equation 1 shows that w must be bigger than $\frac{c_1+c_2}{2} - 1$. To make this a possibility, $c_1$and $c2$ must have the maximum values of 1.49 or else the w would not be applicable. Assuming both c_1 and c_2 are at their maximum value of 1.49, the result shows that $w > 0.49$. Figure 4 illustrates how a linear inertia works with the number of iterations.

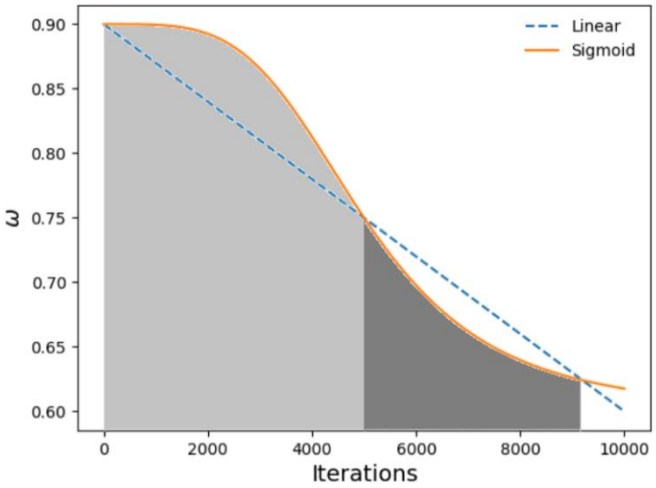


Figure 4: Shows the decaying of linear inertia with respect to the number of iterations. It's also compared to a Sigmoid which is an S-shaped activation function (Dhirf, H., et. al., 2019, p. 4). Copyright by ResearchGate.

There is also another PSO variant we explored called Bare Bones PSO which builds on the previously discussed algorithm, Canonical PSO. The difference between the two is the update component and velocity. Bare Bones has also several other variants like Gaussian Bare Bones PSO, which is based on the Gaussian distribution, and Levy Bare Bones PSO, which is based on Levy distribution (Wang & Qiu, 2013).
In Bare Bones PSO the velocity of a particle is not used and does nothing. The update component is changed "to a stochastic one that randomly updates particle position components with either the corresponding component of the particle best position or a new component value selected form a normal distribution with a mean between the particle best component and the neighborhood best component" (Kneusel, 2021, p.48). Equations 2 and 3 shows how positions are updated using Bare Bones PSO.

Given that $p_b > p - U[0,1)$

$$\bar{x} = \frac{1}{2}\left(\hat{g}_{ij} + \hat{x}_{ij}\right)$$
$$\sigma = \left|\hat{g}_{ij} - \hat{x}_{ij}\right|$$
$$x_{ij} <- \mathrm{N}(\bar{x}, \sigma)$$

Equation 2: shows the relationship between x_{ij}, σ, and $\bar{x}$.

Or else,

$$x_{ij} <- \hat{x}_{ij}$$

Equation 3: Formula for position updates in Bare Bones PSO

In Figure 5 we can see how the particle positions were updated using PSO, from iteration 0, 12, 24, and 30. The objective function we used for testing PSO is shown in Equation 4

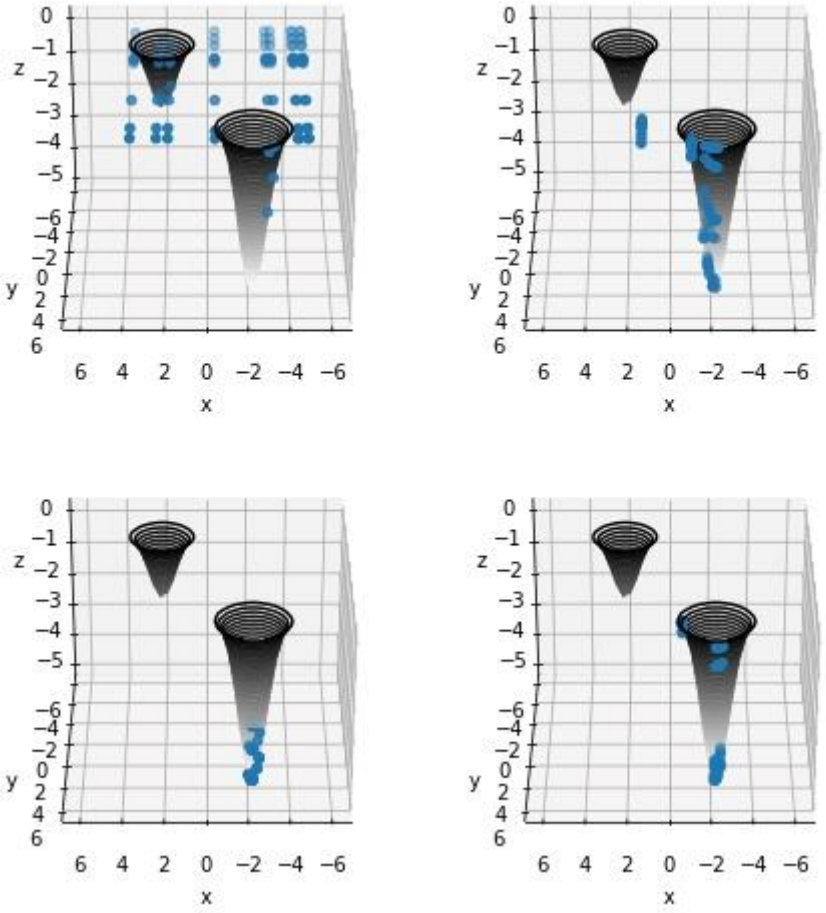


Figure 5: PSO swarm position updates clockwise from upper left: initial, iteration 12, iteration 24, iteration 30

2.1.3. GA & MA – Genetic Algorithm & Memetic Algorithm

Genetic algorithm (GA) uses concepts of evolution in biology, in other words, it mimics some of the aspects of evolution in our world. An important thing to note is that it only employs "some" aspects as biological evolution is far more random, seeing as it revolves around mutation and natural selection. This makes sense as nature consists of substantially more variables, therefore making it necessary to have random mutations. Whereas GA has a clear goal that we as "overlords" are trying to reach and thus can direct it towards that goal. We select what mutations we want, who can breed, when and how often they can breed and so on. There are numerous applications for GA/MA. They have been used to solve both real and theoretical problems.

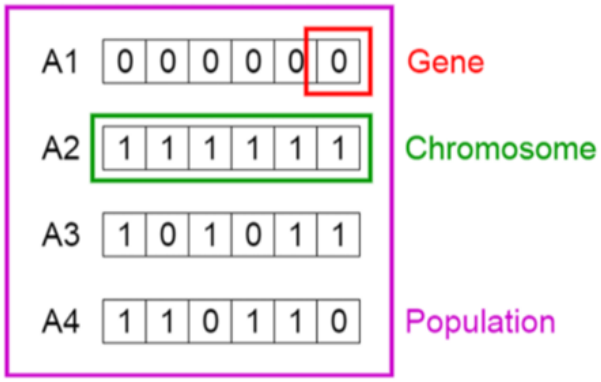


Figure 6: An overview of what GA population consist of (Mallawaarachchi, V. 2017). Copyright by author

GA uses a population of possible solutions (highlighted in purple, Figure 6). The collection of all the possible solutions combined at a given time during the process of running the algorithm is called a generation. There are two steps that must be completed for a new generation to be created.

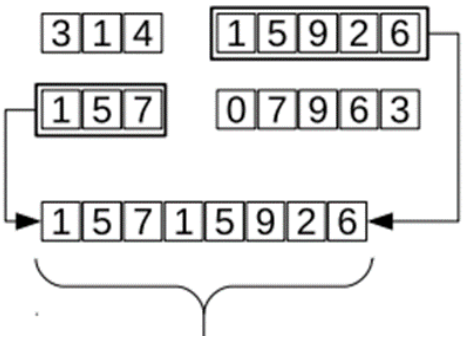


Figure 7: An example of crossover (Kneusel, 2021, p.79). Copyright by author.

The first step is called crossover. This is the process used to create offspring, which are a mix of two individuals (chromosomes). Where we draw a line to split the genes (dimensions) in each individual and take the genes from the left side of this line from one individual and the genes from the other side of the other individual and put them together to create an offspring, as depicted in figure 7. When choosing which individuals, we are going to breed, we first choose one at random and the second is chosen among the best performing individuals, depending on their fitness. This is done by setting a value for "top". If we set top = 0.5 then the top 50% individuals will be put in the best performing category and if we set top = 0.25 then the top 25% and so on. This lets individuals who have larger error function values improve by integrating genes from a more fit individual.

The second step is called mutation. The default mutation rate for GA is 5%, meaning every individual has a 5% chance to be mutated. If an individual is chosen to be mutated only one of its genes is going to get mutated. Let's say that every gene in our population has a value from 1 to 10, if a gene is selected to be mutated then its value is going to be a random value from 1 to 10. This incorporates some randomness to our algorithm which allows it to explore more solutions.

When talking about GA it's important to note that there are several algorithms that build on it, one of them is Memetic Algorithm (MA). MA uses all the concepts from GA but adds local search on top of it. This is done by moving every individual one position in each dimension, first in the positive direction and if this doesn't improve the result, we move it in the opposite direction. If neither direction improves the result, we return it to its original position. Table 4 explains the parameters used in GA and MA.

Table 4: explanations of different parameters used in both GA and MA

Parameter	**Explanation**
npart	Number of individuals in the swarm. A larger npart will result in a bigger variation in genes and in most cases a better end result, but the drawback is that this will require more computational power and more time used to calculate each generation.
ndim	Number of dimensions (see appendix A.4) in the search space of the swarm. The size of ndim depends entirely on what problem you are trying to solve. ndim is also the number of genes each individual possesses.
max_iter	Maximum number of iterations/generations our algorithm is going to run. This will decide when our algorithm is going to stop running, in addition to this you can also have a stopping point for when an individual gets an objective function value lower than a predetermined limit.
$\boldsymbol{CR}$	Crossover probability. This probability determines how likely an individual is to be switched out with a mixture of itself and one of the best performing individuals. In another words, whether an individual is going to go through the crossover process or not.
$\boldsymbol{F}$	Mutation probability. This determines how likely an individual is to be mutated.

top	Top is what determines how many (in decimal, which is going to be used as percentage) we are going to have in the best performing "category". If this parameter is set to 0.25, 25% of the best performing individuals are going to be put into the best performing category.

Before we decided to use either MA or GA, we ran multiple tests to determine whether using MA would be beneficial. MA in theory will result in better results, but the issue with using MA instead of GA is the time consumption, as the algorithm must check in each generation if moving particles in the positive and then negative direction will achieve a lower objective function value. This may increase the run time of the algorithm threefold. To combat this, we can change the code so it will only run local search every n-th generation.

When we ran these tests, we decided to use the default values for each parameter, except npart, ndim and max_iter. The default values for these being 10, 3 and 200 respectively, while the values we used are listed in the rightmost column of Table 5

The objective function we used to compare the performance of GA and MA is shown in Equation 4.

$$y = -5e^{\left(-0.5*\left(\frac{(x+2.2)2}{0.4}+\frac{(y-4.3)2}{0.4}\right)\right)} - 2e^{\left(-0.5*\left(\frac{(x-2.2)2}{0.4}+\frac{(y+4.3)2}{0.4}\right)\right)}$$

Equation 4: Sample objective function

Table 5: Observations, results from different tests done with GA and MA

Observations				**Acronyms and parameter values**
GA		**MA**		**SBU – number of swarm best updates**
SBU	**GB**	**SBU**	**GB**	**GB – Global best**
3	-1.99	8	-4.98	
9	-4.65	8	-4.99	**npart = 30**
4	-4.92	4	-4.99	**ndim = 2**
3	-4.93	11	-4.97	**max_iter = 30**
5	-1.99	10	-4.9	**CR = 0.5**
7	-1.94	6	-4.99	**F = 0.05**
4	-4.99	4	-4.99	**top = 0.5**
2	-4.3	6	-4.99	
7	-4.97	6	-4.99	

5	- 4.95	11	-4.97	

Table 6: statistics of how both MA and GA did in our tests

Statistics				**Acronym**
GA		**MA**		**MAPE – Mean absolute percentage error** **MAE – Mean absolute error** **MGB – Mean global best** **MSBU – Mean number of swarm best updates**
MAPE	20.58116%	**MAPE**	0.28%	
MAE	1.02	**MAE**	0.014	
MGB	-3.96	**MGB**	-4.976	
MSBU	4.9	**MSBU**	7.4	

From both the observations (table 5) and the statistics (table 6) we can see that MA did better in every category, except for where GA had a larger number of swarm best updates in 3 out of the 10 tests we ran. This is negligible when we compare the MSBU where MA was vastly superior. From these results we saw that it would be more appropriate to opt for MA, but as previously mentioned, it would also be more time consuming.

In Figure 8 we can see how the particle positions were updated using MA, in iteration 0, 12, 24, and 30. The objective function we used for testing MA is shown Equation 4.

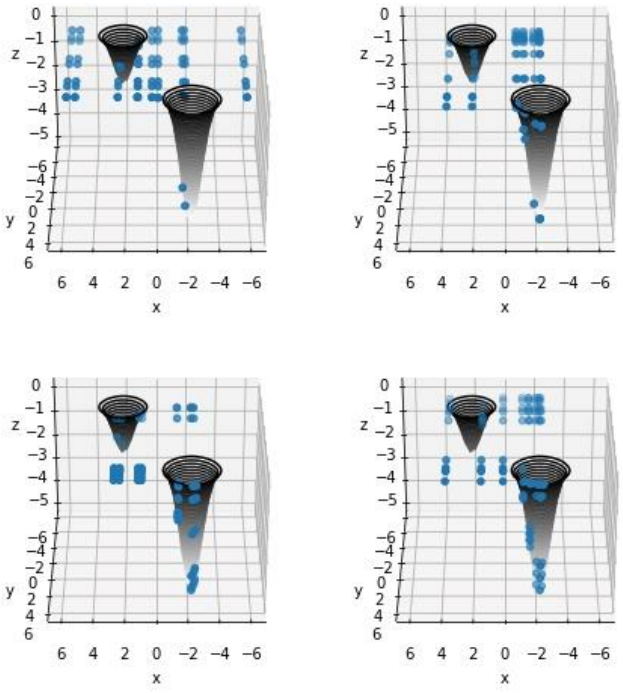


Figure 8: MA swarm position updates clockwise from upper left: initial, iteration 12, iteration 24, iteration 30

2.1.4. DE – Differential Evolution

The differential evolution algorithm (often called DE) is a metaheuristic stochastic population-based direct search method (Storn & Price, 1997). Metaheuristic search methods are problem-independent techniques that can be applied to a wide range of optimization problems. When performing a search, DE utilizes a combination of randomness and evolution to search a broad search space. DE is part of a broader family of algorithms that are inspired by nature. These evolutionary algorithms, as they are called, imitate aspects of the biological evolution and nature to solve a specific problem. In contrast to biological evolution which evolves randomly due to various factors that affect it such as mutation or natural selection, DE has a specific direction and a goal, which is to find the position with the minimal objective function value.

DE is popularly applied alongside other algorithms. It performs well with combinations of various swarm sizes and iterations and finds the global minimum in most cases. This versatility makes it a popular swarm optimization algorithm that is widely utilized. Some application areas that utilize DE can range from trying to solve a simple mathematical problem to solving a nuclear engineering optimization problem (Sacco et al., 2009, p.1093). Table 7 shows the parameters used by the DE algorithm.

Table 7: Parameters of the DE algorithm

Parameter	Description
npart	The number of particles in the swarm. This affects the amount of time and computational resources that will be used to find the solution.
ndim	Number of dimensions in the search space of the swarm. This parameter is bounded by the problem it is being used in.
max_iter	Maximum number of iterations. Due to some characteristics of DE, such as the ability to give up easily on exploration if the objective function lacks local minima or has a strong minimum that can be found easily, it needs fewer iterations to find the solution.
top	This parameter determines the percentage of best performing particles in a swarm. Adjusting this parameter can cause reduced time to find a solution, and bloat inside the program.

$\boldsymbol{v}$	Mutation vector. This is a position in the search space that is caused by mutating three donor vectors.
$\boldsymbol{v_1, v_2\ and\ v_3}$	Donor swarm vectors used to update positions.
$\boldsymbol{CR}$	Crossover probability. Probability of a particle breeding with one of the top performing particles during a crossover. Adjusting this parameter will lead to an increase or decrease in breeding of a particle during crossover.
$\boldsymbol{F}$	The Amplification factor. This value affects the balance of exploration and exploitation in the algorithm which then affects particle convergence and the speed of convergence.
$\boldsymbol{u}$	Candidate vector. In a crossover, this value inherits some elements from v.
mode	Determines the variant of DE that is used to select the donor vector v_1
cmode	Determines the version of crossover that is used. It can be GA style crossover or Bin (Bernoulli experiments)
bin	Crossover is implemented using individual Bernoulli experiments
rand	Sets v_1 to a randomly selected swarm position
best	Use current swarm best position as v_1
toggle	Each swarm update toggles between rand and best

The structure of DE is different from other algorithms in a crucial way. In contrast to other algorithms in which the process of crossover occurs before mutation, DE performs mutation and uses the mutation vector inside the crossover step. After going through both steps, DE performs selection. The three last steps are then repeated until the problem is solved and the goal is reached as shown in Figure 9 (Ahmad et al., 2022, p.3836).

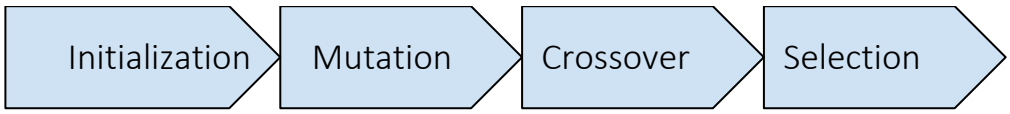


Figure 9: Structure of DE

DE, like most algorithms, is structured in a known and uniform way. As shown in figure 9, initialization is the first step in any swarm algorithm (Kneusel, 2021, p.18). Our choice was Quasirandom Initializer which scatters the particles around the search space but in a less random way, there is some space between each particle.

After the search has been initialized, mutation occurs. DE performs mutation in a unique way, it finds a mutation vector by using three different donor vectors. The difference between two vectors v_2 and $v_3$is amplified by the amplification factor F, which is a value between [0, 2]. This is a crucial parameter responsible for the balancing of exploration and exploitation. Increasing F subsequently increases the exploration in the swarm causing the particles to use more time to find the solution, meanwhile decreasing this value causes the opposite. This limits the premature convergence of the particles, which is a drawback in the DE algorithm (Ahmad et al., 2022, p.3837). This difference is offset to a third donor vector v_1.The formula for the mutation vector is shown in equation 5.

$$v = v_1 + F\,(v_2 - v_3)$$

Equation 5: Formula for mutation vector v

Like many metaheuristic algorithms, DE is a highly customizable algorithm which leads to the convergence of many variants (Kneusel, 2021, p.97).

After the process of mutation is done and we find our mutation vector v, we then apply crossover. Crossover in DE is unique to itself and is one of many quirks that puts it at an advantage from the competition. This is because in DE, the goal of the crossover is to create a candidate vector u, by using an element from the mutation vector and selecting each element of this candidate one at a time. This makes it highly diverse and improves its ability to find the solution faster. To calculate the candidate vector, we use equation 6.

$$u_i = \begin{cases} v_i, if\ r < CR\ or\ i = I \\ x_i \end{cases}$$

Equation 6: Formula for candidate vector u

This equation uses different parameters and the mutation vector v from previous step. x_i is the current particle position. CR is the crossover probability that must be larger than the random number r [0,1]. If CR is set to a low percentage, it causes a low amount of breeding between

particles, consequently if set to high it causes high amount of breeding between particles. the variable I makes sure that at least one element from candidate vector u comes from the mutation vector v. Using both mutation and crossover methods we reach our goal of finding candidate vector u. The last step is selection where the candidate vector is decided if it will be included in the upcoming generations, this is done by comparing to the current particle position x_i using the greedy criterion (Huang W. et al., 2019, p.3). This criterion makes sure that a new parameter vector is selected and accepted into the new generation only if it reduces the objective function value.

In figure 10 we can see how the particle positions were updated using DE, from iteration 0, 12, 24, and 30. The objective function we used for testing DE is shown in Equation 4.

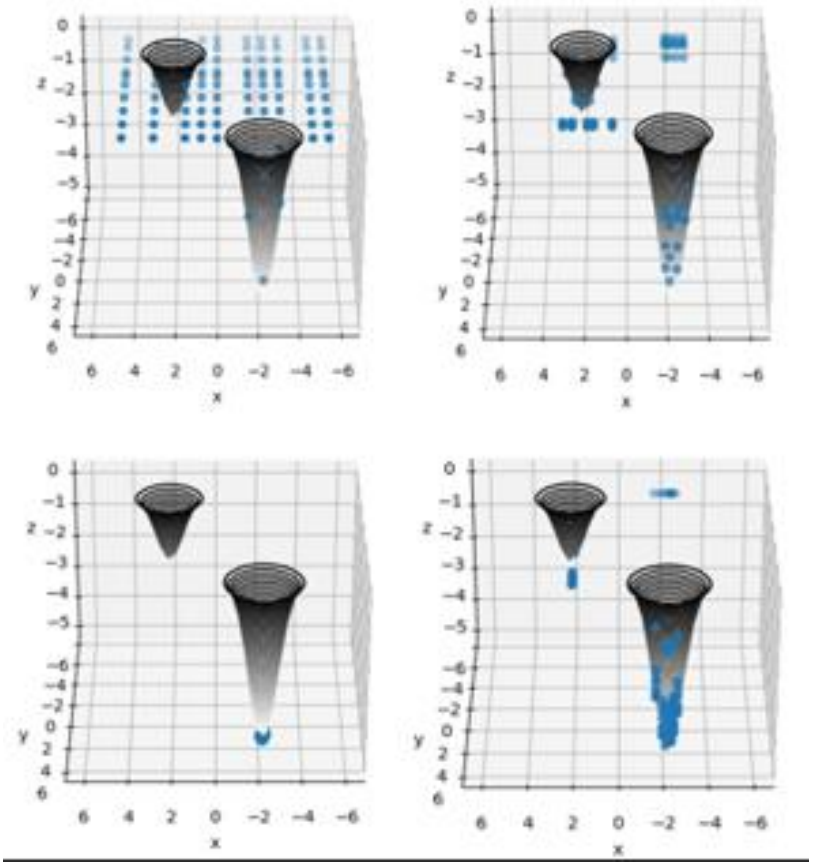


Figure 10: DE swarm position updates clockwise from upper left: initial, iteration 12, iteration 24, iteration 30

2.1.5. NEAT – Neuroevolution of Augmenting Topologies

While we were in the research phase, we were looking for different algorithms that could be used to reach our goal. While surveying the internet and loads of published papers we stumbled upon an algorithm called NEAT, which looked promising at first glance. We delved into it and studied how it works, what it does, use cases and so on. What we failed to realize at first was its shortcomings. This algorithm works wonders with small, or easy problems, meaning problems with few variables. Which was not the case with our problem. Not realizing this, we decided to code NEAT from scratch to both learn how it works and be able to make small changes along the way if needed, in addition to this we thought we could maybe take some aspects from NEAT and incorporate it into other algorithms, such as the concept of speciation.

To sum up how NEAT works, the algorithm starts off with just the input and output nodes of a neural network. From there, it adds/removes nodes and connections while changing the weights and biases, all to try and reach an optimal neural network architecture for a specific problem. Unfortunately, because of the methods it employs it struggles to reach a favorable architecture for more demanding problems. Regrettably we did not realize this before it was too late, we had already spent almost 3 weeks developing this algorithm and this issue did not become clear before we started testing it on different problems. After its shortcomings became clear, we decided to cross it off as a potential candidate.

2.2. Dataset description

The dataset shown in Table 8 has 19736 instances, 28 columns which represent the features and 1 column that represents the output (Appliances-column).

Table 8: Dataset description before feature engineering

Dataset characteristics	Time-series, Multivariate	Number of instances	19736
Attribute characteristics	Real	Number of attributes	28
Associated tasks	Regression	Missing values	N/A
Area	Energy		

After feature engineering, the table transforms into Table 9, where the only difference is the number of attributes after feature engineering.

Table 9: Dataset description after feature engineering

Dataset characteristics	Time-series, Multivariate	Number of instances	19736
Attribute characteristics	Real	Number of attributes	30
Associated tasks	Regression	Missing values	N/A
Area	Energy		

2.2.1. Data source

The data is sourced from Chievres airport station using a ZigBee Wireless Sensor Network (WSN) to make data more reliable. You can find more information about WSN in Appendix A.5 WSN (Wireless Sensor Networks)**.**

2.2.2. Attributes

This dataset contains measurements of temperature and humidity. It also utilizes important features such as pressure, and wind speed. The dataset has 19736 instances and 29 columns which represent the features and the output. Given the nature of this dataset, namely that it is in the time-series format, we had to find a way to incorporate this dataset into our algorithm and we came up with a solution that utilized our data effectively.

This dataset is also suitable for regression algorithms since we intended to predict the energy for the appliances after a given time-interval as defined by each instance. The intervals are 10 minutes apart and measured from 11-01-2016 to 27-05-2016.

Table 10 gives detailed information about all features, also called attributes, that are involved in the dataset.

Table 10: attribute information from the dataset

Date	The Date-feature is represented with time in a format of *dd-mm-yyyy hh:min:ss*
Appliances	Energy use for appliances measured in *Wh*
Lights	Energy use of light fixtures measured in *Wh*
T1	Temperature in the kitchen area in *C°*
RH_1	Humidity in kitchen area in *%*
T2	Temperature in the living room in *C°*
RH_2	Humidity in the living room in *%*

T3	Temperature in the laundry room area in *C*°
RH_3	Humidity in the laundry room in *%*
T4	Temperature in the office room in *C*°
RH_4	Humidity in office area in *%*
T5	Temperature in the bathroom in *C*°
RH_5	Humidity in the bathroom in *%*
T6	Temperature outside the building in *C*°
RH_6	Humidity outside the building in *%*
T7	Temperature in the ironing room in *C*°
RH_7	Humidity in the ironing room in *%*
T8	Temperature in teenage room 2 in *C*°
RH_8	Humidity in teenager room 2 in *%*
T9	Temperature in parent's room in *C*°
RH_9	Humidity in parent's room in *%*
T_out	Temperature outside in *C*°
Press_mm_hg	Celsius pressure in mm Hg
RH_out	Humidity outside in *%*
Windspeed	Windspeed form Chievres Weather station in *m/s*
Visibility	From Chievres Weather station in *km*
Tdewpoint	Temperature the air needs to be cooled to achieve a relative humidity of 100%
rv1	Nondimensional random variable 1
rv2	Nondimensional random variable 2

```
Data columns (total 29 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   date         19735 non-null  object
 1   Appliances   19735 non-null  int64
 2   lights       19735 non-null  int64
 3   T1           19735 non-null  float64
 4   RH_1         19735 non-null  float64
 5   T2           19735 non-null  float64
 6   RH_2         19735 non-null  float64
 7   T3           19735 non-null  float64
 8   RH_3         19735 non-null  float64
 9   T4           19735 non-null  float64
 10  RH_4         19735 non-null  float64
 11  T5           19735 non-null  float64
 12  RH_5         19735 non-null  float64
 13  T6           19735 non-null  float64
 14  RH_6         19735 non-null  float64
 15  T7           19735 non-null  float64
 16  RH_7         19735 non-null  float64
 17  T8           19735 non-null  float64
 18  RH_8         19735 non-null  float64
 19  T9           19735 non-null  float64
 20  RH_9         19735 non-null  float64
 21  T_out        19735 non-null  float64
 22  Press_mm_hg  19735 non-null  float64
 23  RH_out       19735 non-null  float64
 24  Windspeed    19735 non-null  float64
 25  Visibility   19735 non-null  float64
 26  Tdewpoint    19735 non-null  float64
 27  rv1          19735 non-null  float64
 28  rv2          19735 non-null  float64
dtypes: float64(26), int64(2), object(1)
```

Figure 11: Overview of the data in terms of data types and missing values

The output of the data.info() function is shown in Figure 11, which tells us about the datatypes of the columns in the dataset and which columns have missing values.

As we can see here, all the columns are numerical except for the date column, which is a string object. This means that we must do some feature engineering on the date column, as we cannot use strings directly as inputs to our DNN-based algorithm.

We can also see that none of the columns have missing values, as they all have 19735 defined non-null values. Null values are undefined values, also referred to as NaN.

2.2.3. Statistical analysis of the dataset

Using the data[«Appliances»].describe() function, we get statistical insight into the "appliances" column. As we can see in Figure 12, the max value here is far higher than the 75th percentile, which tells us that we have at least one extremely high value.

count	19735.000000
mean	97.694958
std	102.524891
min	10.000000
25%	50.000000
50%	60.000000
75%	100.000000
max	1080.000000

Figure 12: Statistics of the "appliances" column in the dataset

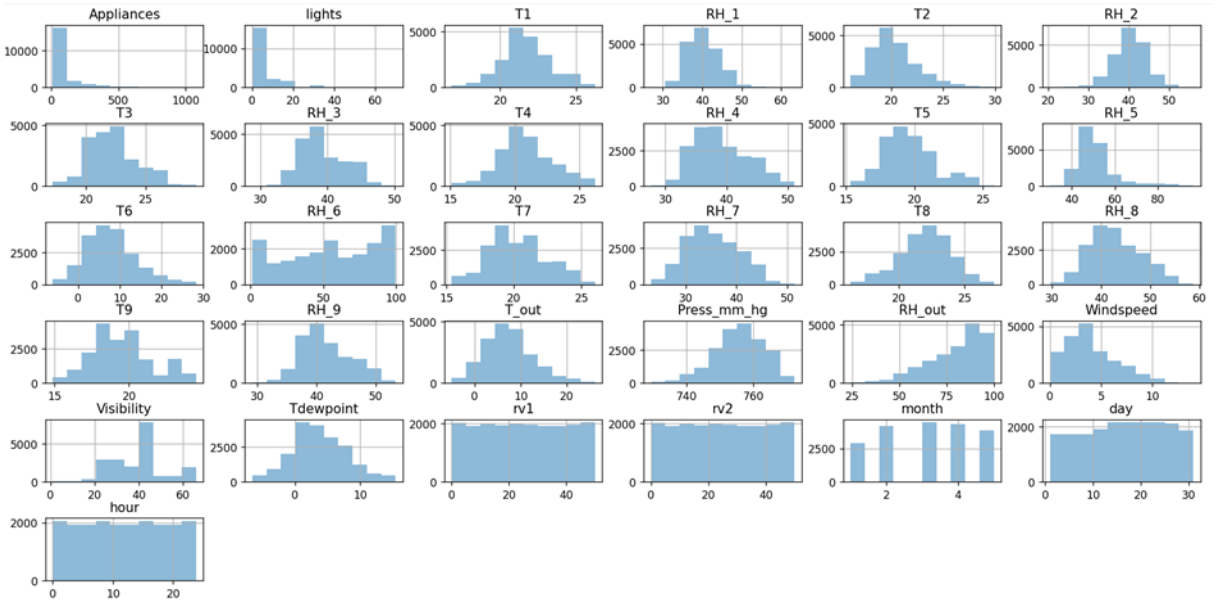


Figure 13: Bar plots showing the distribution of all the variables in the dataset

Looking at the distribution of the different values in Figure 13, we can see that most of the variables are evenly distributed, while a few of them stand out.

The Appliances-column is slightly right-skewed, which is something important to consider as this is the column we are trying to predict.

Lights also stands out here, as it appears to be a discrete variable which only takes on values in certain intervals.

In figure 14 we can clearly see that we have a right-skewed distribution with some extremely high values. This is common for real-world datasets

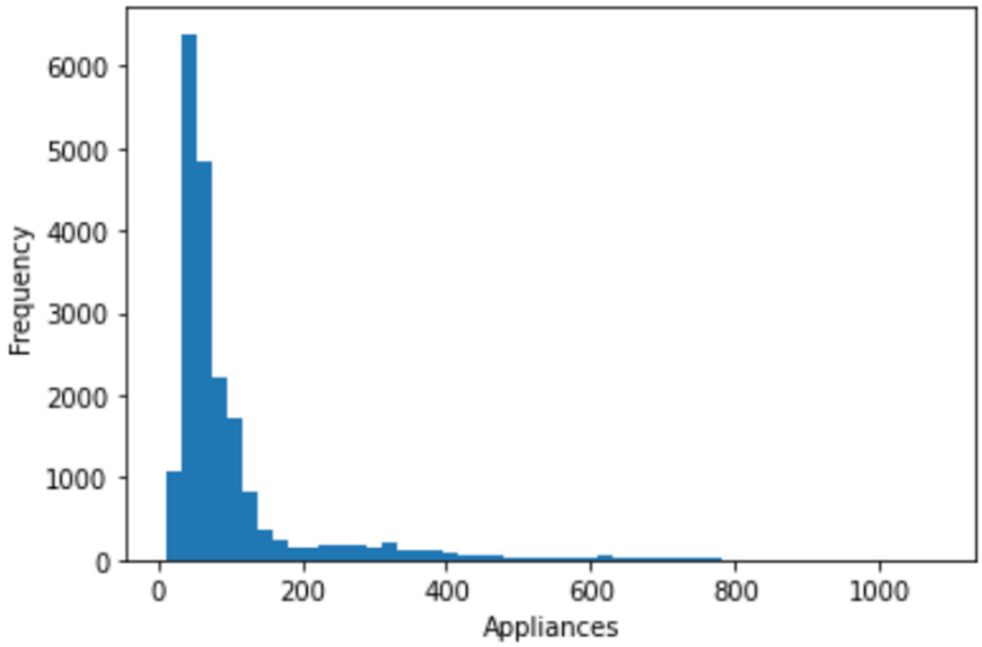


Figure 14: Histogram showing the distribution of appliance values in the dataset

In Figure 15 the boxplot confirms that the appliances column has many outliers with extremely high values. Based on this, we decided to remove the outliers as this often gives better results.

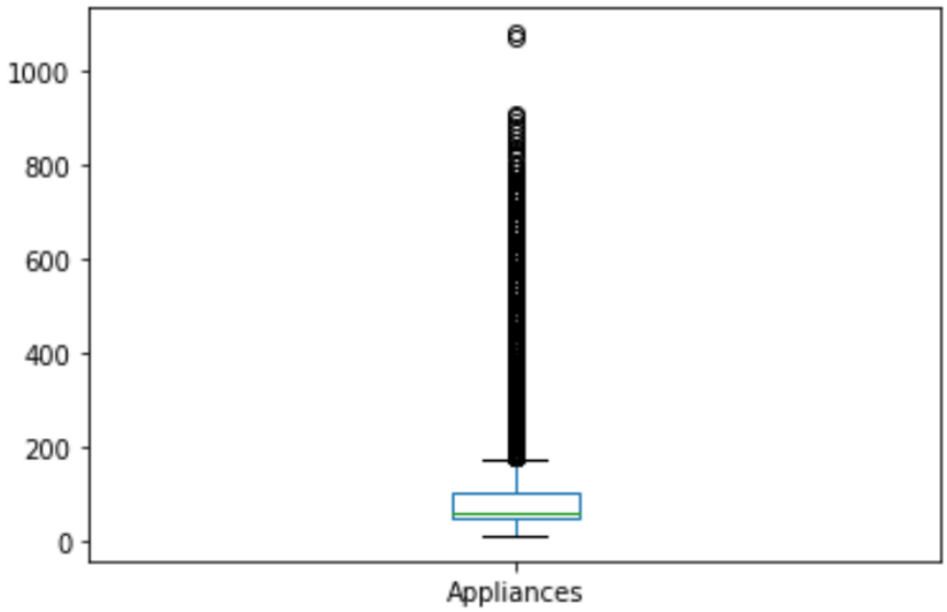


Figure 15: Boxplot showing the distribution of values in the dataset

Detecting and removing outliers is a common practice in machine learning, as this allows us to focus on and get better results on the “regular” day-to-day values. If we were to include the outliers, it would take some of that focus away from the day-to day values and have a negative impact on the results.

This makes sense as we would rather want higher accuracy on values that we commonly see, than to get a decent accuracy on both the common and extreme values.

To achieve this, we will use the 3-sigma method. In other words, instances where the appliance value is less than mean-3*std or higher than mean+3*std, will be removed from the dataset prior to machine learning.

After this process, 540 instances have been removed from the dataset and 19195 instances remain.

In Figure 16 we can see the appliance values throughout the months. The values are evenly distributed, except for January, where we see some extremely high values.

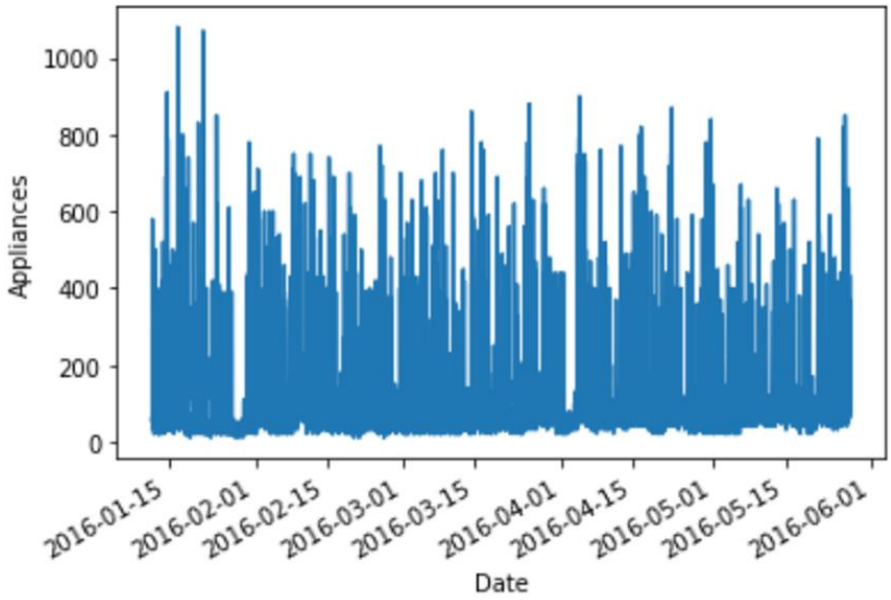


Figure 16: Bar plot showing the appliance values in the dataset

This makes sense as energy consumption generally peaks during wintertime.

Figure 17 shows the correlation between the different variables in the dataset, from which we can see that the five most correlated features to appliances are hour, lights, T2, and T6 with a positive correlation and RH_out with a negative correlation.

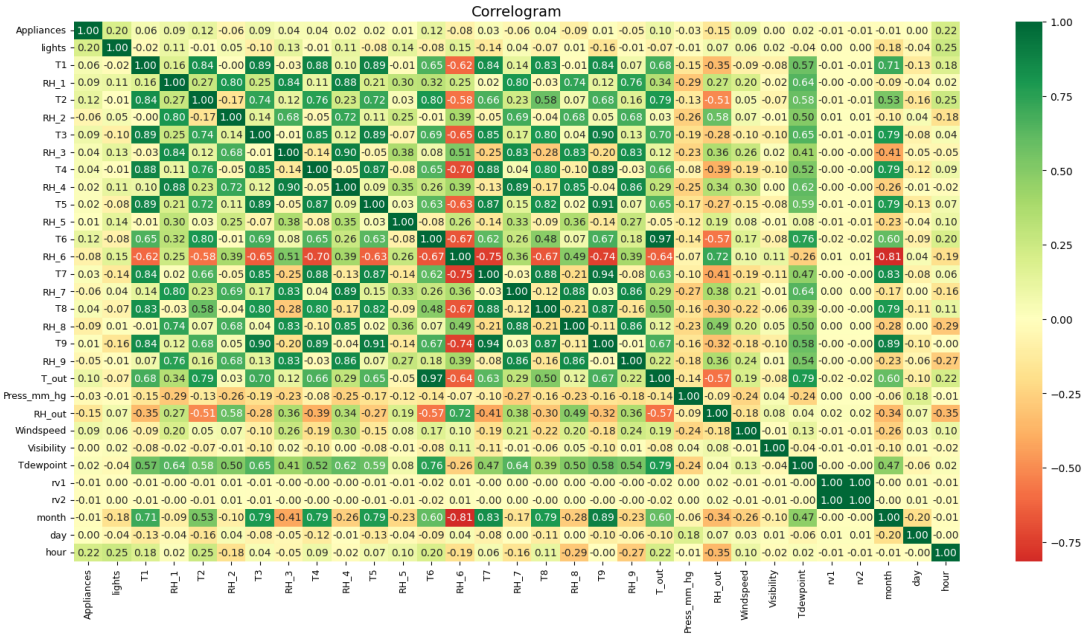


Figure 17: Correlation matrix showing the correlation between the variables in the dataset

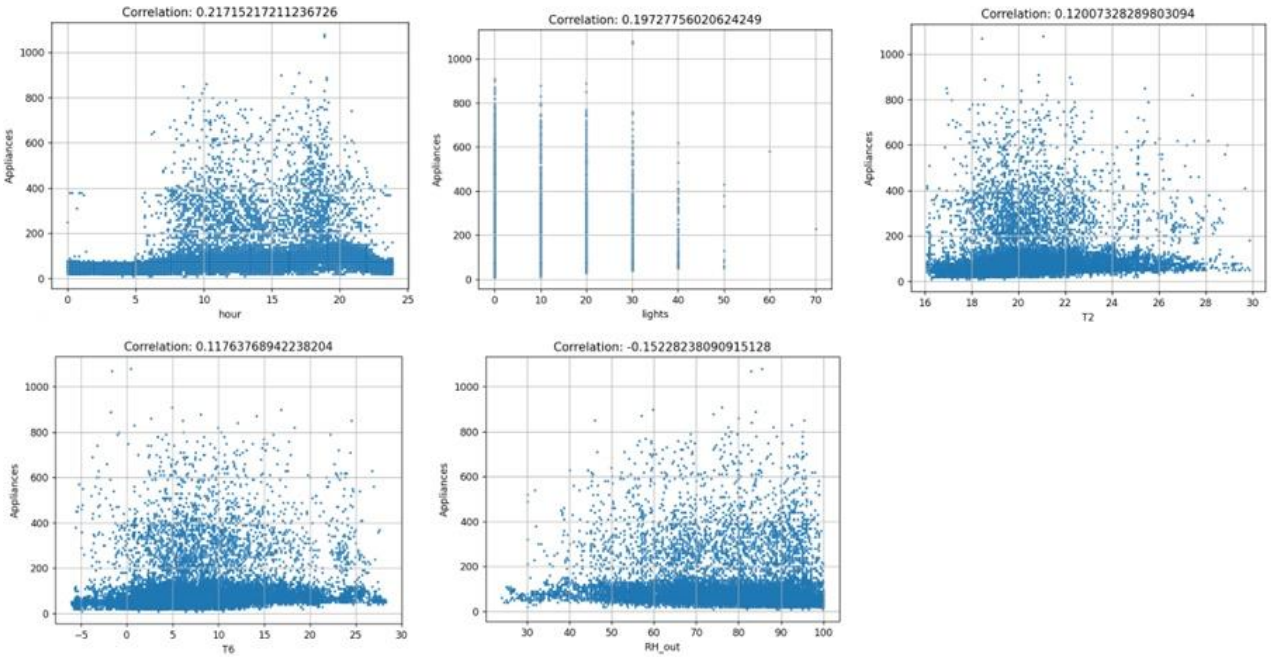


Figure 18: Scatterplots of the five most correlated features to appliances energy consumption: hour, lights, T2, T6, RH_out

Figure 18 shows scatterplots of the five most correlated features to appliances energy consumption. Using the five most correlated features, we can make 10 different 3D scatterplots with two features as input variables and appliances energy consumption as output variable. To make them easier to interpret, datapoints with lower energy consumption are shown as lighter colors and datapoints with higher energy consumption are shown as darker colors. The 3D scatterplots are shown from the angles we consider to be the most informative. Figures 19-22 below show the 3D scatterplots.

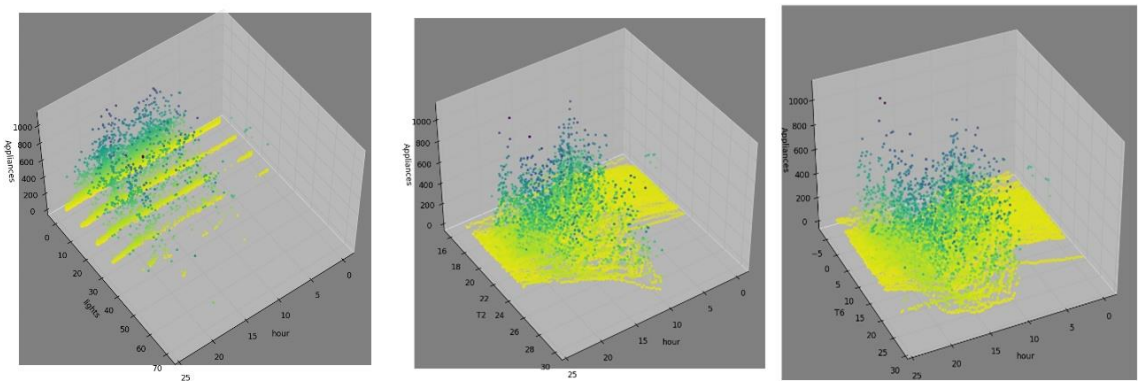


Figure 19: Energy consumption of appliances with different dependent variables. From left to right: lights & hour, T2 & hour, T6 & hour.

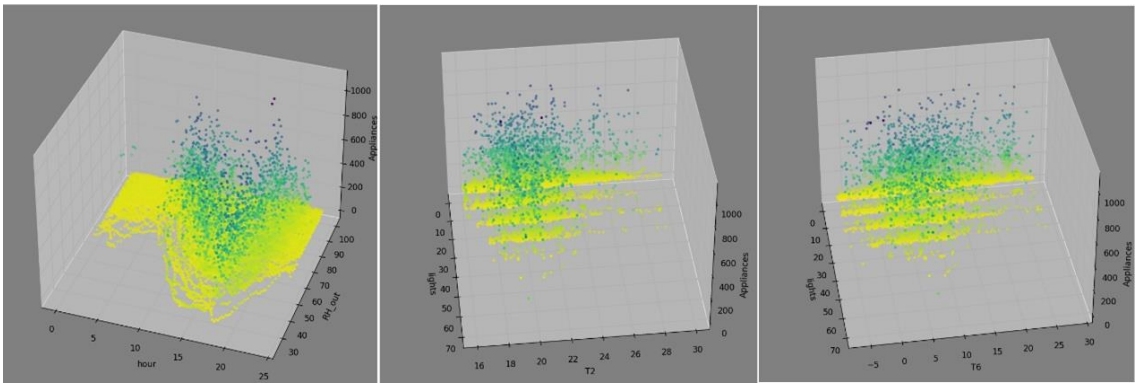


Figure 20: Energy consumption of appliances with different dependent variables. From left to right: RH_out & hour, lights & T2, lights & T6.

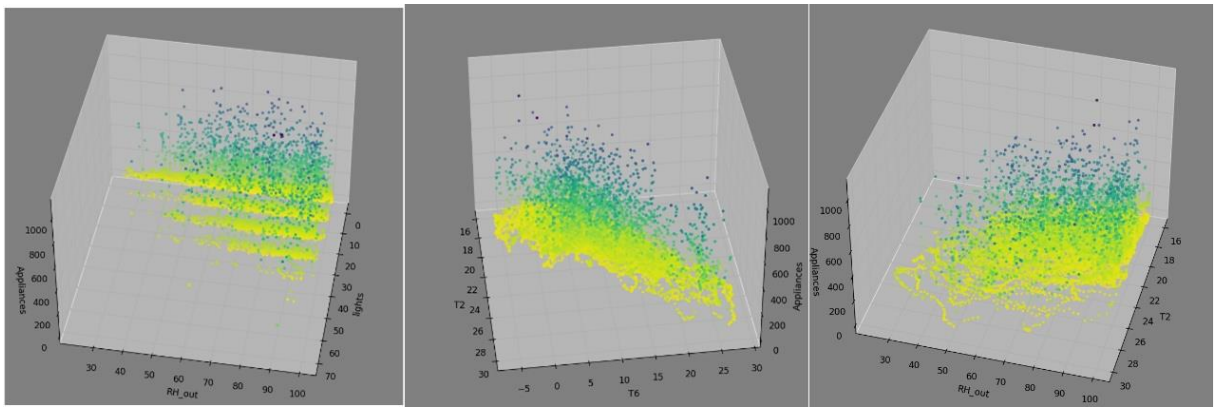


Figure 21: Energy consumption of appliances with different dependent variables. From left to right: RH_out & lights, T2 & T6, RH_out & T2.

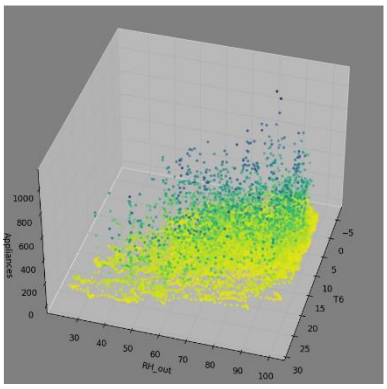


Figure 22: Appliances energy consumption as a dependent variable of RH_out and T6

Looking at the 3D scatterplots, we can clearly see that they are more informative than the 2D ones. Within the group we concluded that the leftmost scatterplot in Figure 20 is the most informative one, because we can see a clear V-shape of higher values. Below we can see it from a bird's eye view.

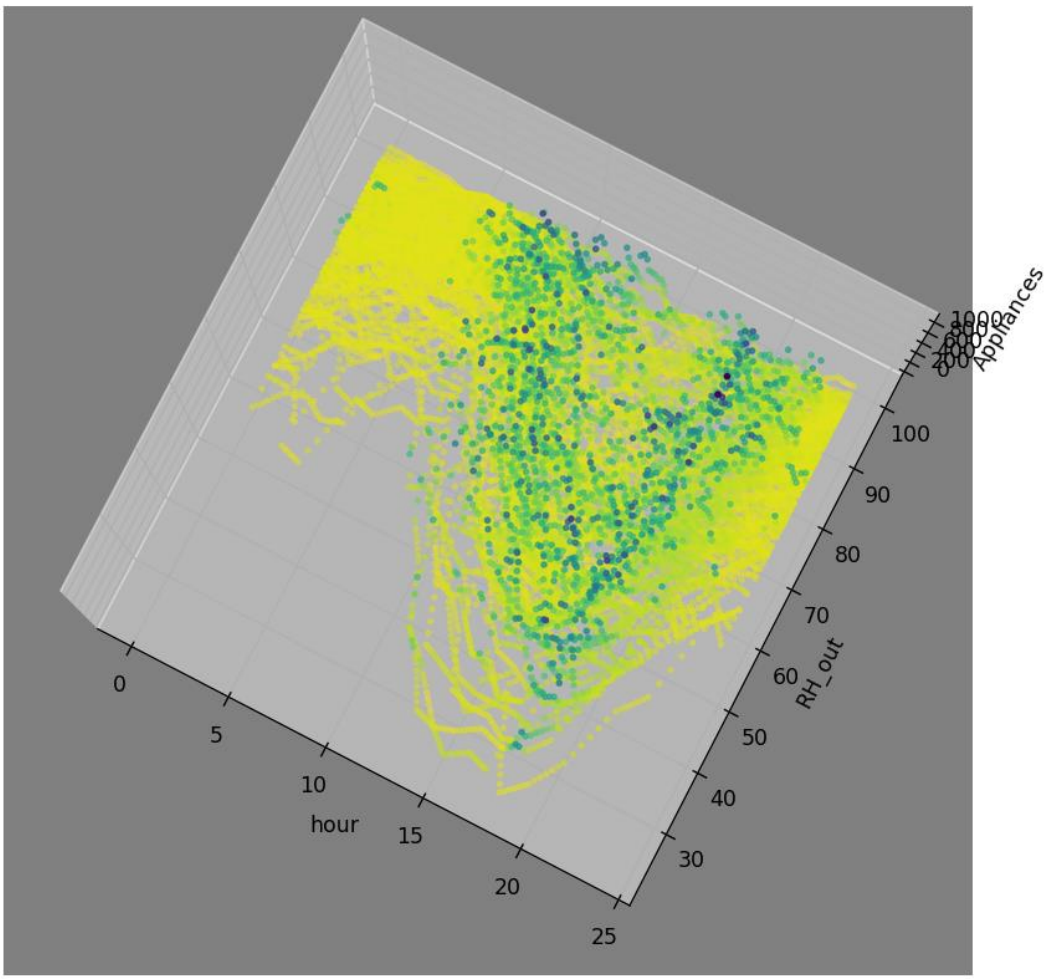


Figure 23: Leftmost scatterplot in Figure 20 shown from a bird's eye view

2.2.4. Feature importance results

When analyzing large-scale datasets using ML models, it can be hard to determine which features have the highest impact on model performance. Nevertheless, knowing which features are more important and which less important plays a significant role in understanding what the data represents and what insights we can extract by analyzing the data. We consider the problem of predicting the energy use in a low-energy house using several state-of-the-art ML models. In this application problem under study, we have 30 input features to the model. The information about the degree of importance for each feature in this case can tell us what rooms or input variables (i.e., the features) cause the most energy use in the house.

Some models and algorithms have been proposed in literature to determine the degree of importance of features and as a result the ranking of all features, but every one of them employed their own metrics for quantifying the degree of importance and thus we may see different rankings of features by using different models even over the same dataset. We compare several state-of-the-art ML models, trying to get a more unifying, consistent, and conclusive results on the ranking of features in terms of their relative degree of importance for the model prediction accuracy. By doing so we could single out some features that have the most impact upon the ML model prediction capacity. The comparative results are presented and analyzed to show how well we address, to some extent, the well-known interpretability issue for ML when applied to energy forecast problem.

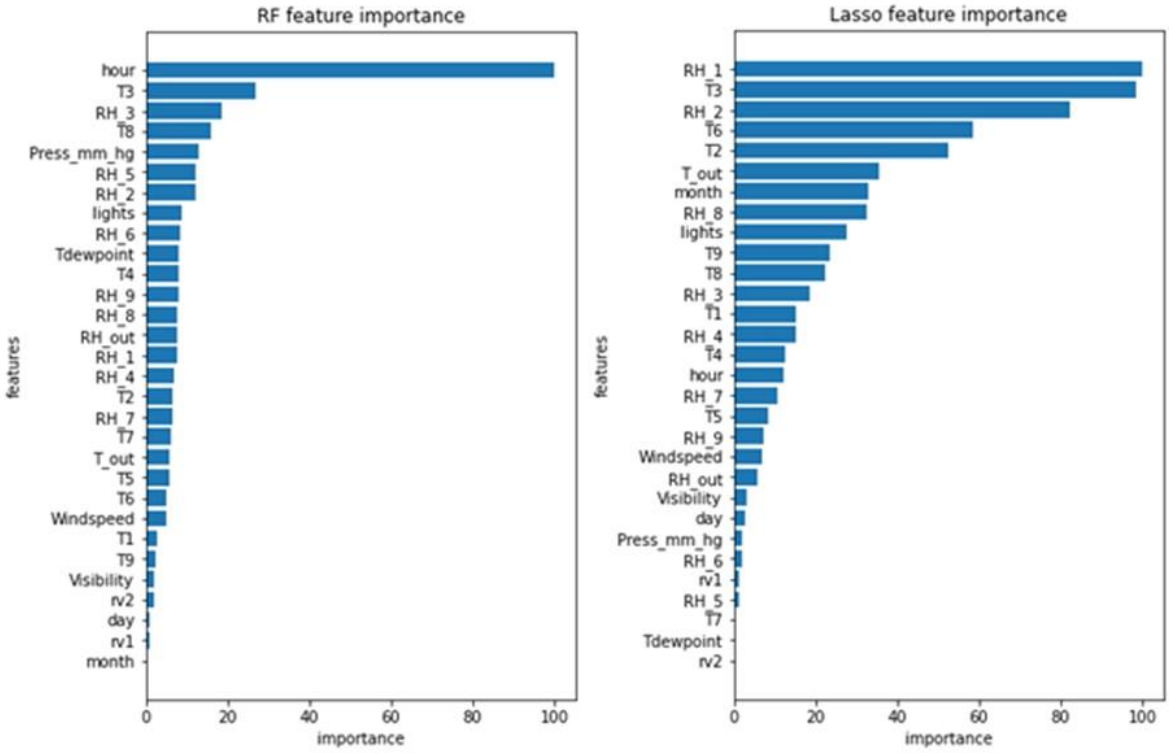


Figure 24: RF - Random Forest feature importance results and Lasso feature importance results respectively. Where the X-axis is relative importance, and the Y-axis contains the different features

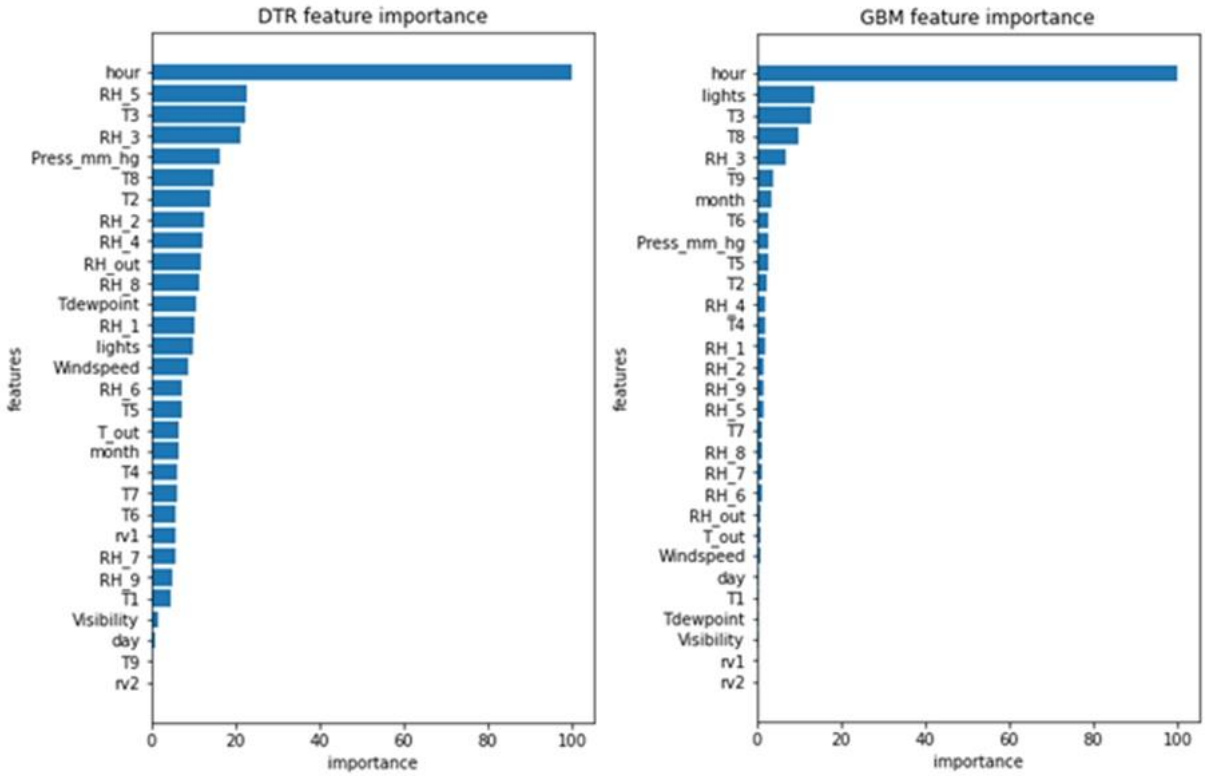


Figure 25: DTR - decision tree regressor feature importance and GBM- gradient boosting machine feature importance respectively. where the X-axis is relative importance, and the Y-axis contains the different features

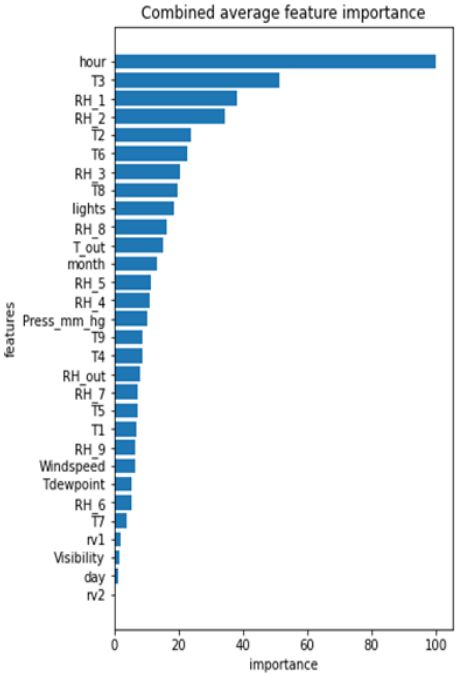


Figure 26: combined average feature importance. a collection of the different feature importance results

To determine which features, have the most impact we decided to go with 4 state of the art machine learning models. Lasso, RF - Random Forest, DTR - Decision tree regressor and GBM - Gradient boosting machine. Using these models, we can get a good estimate on the impact of each feature. By looking at the results we can single out some features that are clearly in the lead. “hour” is the number 1 most impactful feature from these results, T3, lights, RH_1 and RH_2 are also all placed very high. These placements are further discussed in 4.3 - Feature importance. Looking at the results from Figure 24 and Figure 25 it is easy to miss some important features, therefore we added all the results from these models, scaled the result from 0-100 and got a combined average feature importance

This is maybe not an accurate representation of feature importance as it will combine results from different models with different metrics. However, it does serve to visualize the results from the models used and it will be more accurate than examining the results independently. If we look at Figure 26 we can see that “lights” might not have been as important as we first thought, and “hour” is clearly still superior.

Before the split, we must shuffle our dataset. This means that we change the order of data instances to make sure the model gets and be created with reliable training-set and test-set.

Imagine a time-series dataset like the one we are using, without shuffling the dataset, the top 50% may go directly to the training-set, while maybe the bottom 50% goes to the test-set. This is a problem since the model is then trained on only the top half of the whole dataset, and thus will fail to do reliable predictions because of the lack of training with any of the bottom 50% of the dataset.

Shuffling makes the training of our model resilient to memorizing the patterns of the dataset, which is not something we want, since the neural network must learn rather than memorize. It also helps with faster convergence which is convenient when working with a lot of data instances. The shuffling can easily be done with the *sklearn*-library (see Table 1).

When working with any dataset and implementing it to a certain algorithm, we must split the dataset into training set and a testing set to evaluate its accuracy on unseen data. This is done by using the train/test split. The training set is used for training the model, and the test set is used for testing the model's accuracy. In recent years the validation set has been introduced. The objective of the validation set is to make sure that our model doesn't overfit by making sure that we stop the training process if the training loss keeps decreasing while the validation error keeps increasing. The validation-test is specially used for training neural networks, although it can be used for other algorithms also.

To train our model, we have allocated 60% of the original dataset as a training-set, 15% as validation-set, and 25% as a test-set.

2.3. Survey of existing work

Candanedo et al. based their research on solving the problem related to the high energy demand from appliances in buildings. According to them, two factors explain the electricity consumption in buildings, "the type and number of electrical appliances and the use of the appliances by the occupants" (Candanedo et al., 2017, p. 81). The purpose of their work was to look at the relationship between the energy consumption of appliances and different parameters, using the dataset that we are using now (see 2.2 Dataset description).

Several machine learning algorithms were implemented in their research to look at the relationship between different variables, and to measure their impact. Table 11 gives an overview of the algorithms used by the researchers with a short description of each of them.

Table 11: Names of ML models with their descriptions.

Model	**Description**
Multiple linear regression (lm)	Uses multiple inputs for the output in contrast to simple linear regression. (Aggarwal, 2020)
Support vector machines with radial basis function kernel (SVM-radial)	A supervised machine learning algorithm, supervised means that the input is labeled. The idea behind this algorithm is to separate the dataset into classes by a hyperplane. This is only possible if the classes are linearly separable. If we have a non-linear problem, we must use a kernel function to convert it to linear by increasing the dimensions. (Dobilas, 2021)
Random Forest (RF)	A supervised algorithm that can be used for both classification and regression. It is built up of several decision trees. In a decision tree, we start our process from the bottom of the tree, which is called root node. The objective is to split up the data based on features until the data is clearly separated. The structure of a decision tree is similar to a flowchart. When used in classification, the output will be the answer which most of the trees made. In regression, the value returned will be the average prediction of the trees. Random forest consists of several decision trees that make prediction where the class with most predictions becomes the model's prediction. It is like a voting system. (Yiu, 2019)

Gradient boosting machines (GBM)	This algorithm is based on optimization of a loss function, predictions of a weak learner and minimizing loss function by adding weak learners. Logarithmic loss and squared error are examples of loss functions that can be used. Regression trees are used as weak learners. Their output values are added together for minimizing the loss. (Brownlee, 2016)

All the algorithms were trained with 10-fold cross validation to choose the best one. After training the algorithms, they ended up with 30 results for each model. Neither DNNs nor any EA or SIA (see were used in their research. To compare how the models performed, they used different metrics like RMSE, R^2, MAE and MAPE.

Figure 27 shows the RMSE and R^2 values together with their confidence intervals. The best models are those with the lower RMSE and higher R^2 values.

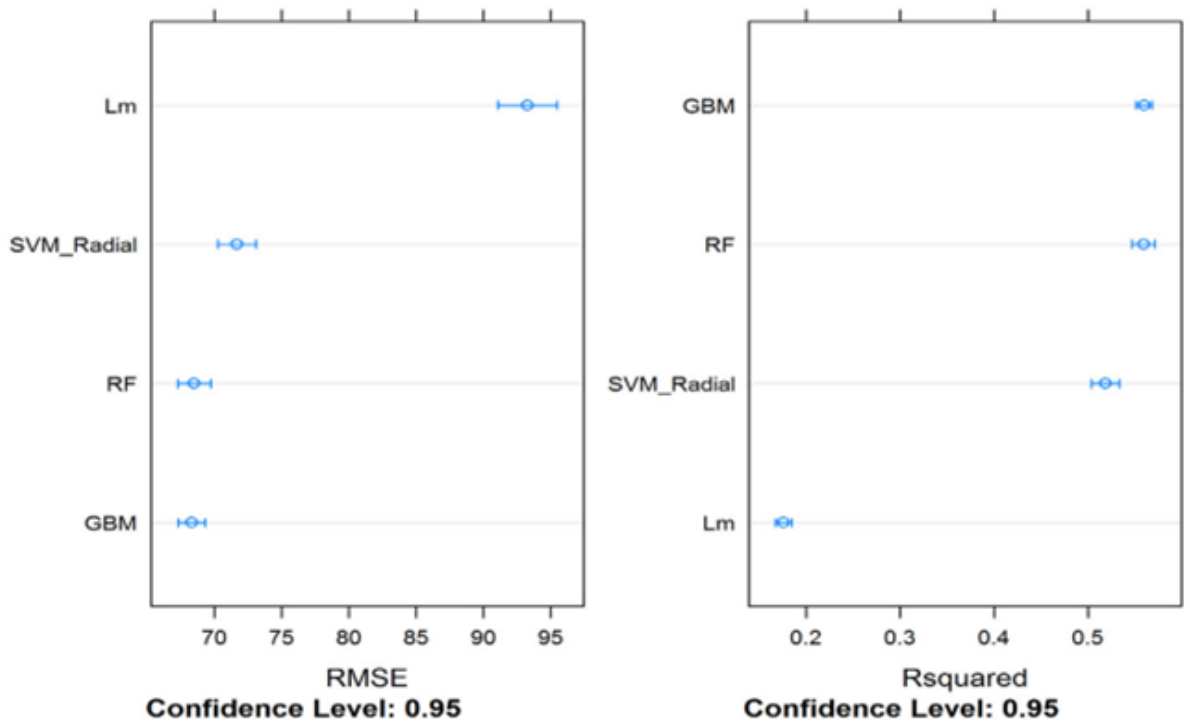


Figure 27: Comparison of RMSE and R2 – values of trained models (Candanedo et al., 2017, p. 90). Copyright 2017 by Elsevier B.V.

As shown in Figure 27, RF and GBM provides the best results with very similar performances. We can also see that SVM performs a lot better than the lm algorithm. Figure 28 shows the performance of the trained algorithms in both the training and testing sets. The RMSE values are very similar to those shown in Figure 27.

Models performance.

Model	Parameters/features	Training				Testing			
		RMSE	R^2	MAE	MAPE %	RMSE	R^2	MAE	MAPE %
LM	Light, T1,RH1,T2,RH2,T3, RH3,T4, RH4,T5,RH5,T6, RH6, T7,RH7,T8,TH8,T9,RH9, To,Pressure,Rho,WindSpd, Tdewpoint, NSM, WeekStatus, Day of Week	93.21	0.18	53.13	61.32	93.18	0.16	51.97	59.93
SVM Radial	Light,T1,RH1,T2,RH2,T3,RH3, T4,RH4,T5,RH5,T6,RH6,T7,RH7,T8,TH8,T9,RH9,To, Pressure,Rho,WindSpeed, Tdewpoint,NSM, WeekStatus, Day of Week	39.35	0.85	15.08	15.60	70.74	0.52	31.36	29.76
GBM	Light,T1,RH1,T2,RH2,T3,RH3, T4,RH4,T5,RH5,T6,RH6, T7,RH7,T8,TH8,T9,RH9,To, Pressure,Rho,WindSpeed, Tdewpoint,NSM, WeekStatus, Day of Week	17.56	0.97	11.97	16.27	66.65	0.57	35.22	38.29
RF	Light,T1,RH1,T2,RH2,T3,RH3, T4,RH4,T5,RH5,T6,RH6,T7, RH7,T8,TH8,T9,RH9,To, Pressure,Rho,WindSpeed, Tdewpoint,NSM, WeekStatus, Day of Week	29.61	0.92	13.75	13.43	68.48	0.54	31.85	31.39

Figure 28: Results of each algorithm in both training and testing sets. (Candanedo et al., 2017, p. 91). Copyright 2017 by Elsevier B.V.

After the researchers found out that GBM gave the best RMSE and R^2 values, they experimented further by removing some of the parameters. We can see the results in Figure 29.

Model	Parameters/features	RMSE training				RMSE testing			
		RMSE	R^2	MAE	MAPE %	RMSE	R^2	MAE	MAPE %
GBM – no lights	T1,RH1,T2,RH2,T3,RH3, T4,RH4,T5,RH5,T6,RH6, T7,RH7,T8,TH8,T9,RH9,To, Pressure,Rho,WindSpeed, Tdewpoint,NSM, WeekStatus, Day of Week	17.90	0.97	12.24	16.66	66.21	0.58	35.24	38.65
GBM – no lights and no weather data	T1,RH1,T2,RH2,T3,RH3, T4,RH4,T5,RH5,T6,RH6, T7,RH7,T8,TH8,T9,RH9, NSM, WeekStatus, Day of Week	18.83	0.97	12.85	17.44	68.59	0.54	36.21	39.23
GBM – no Temp and humidity inside house	Light, To,Pressure,Rho, WindSpeed,Tdewpoint,NSM, WeekStatus, Day of Week	27.47	0.93	18.29	23.71	72.64	0.49	40.32	45.33
GBM – only weather and time information	To,Pressure,Rho,WindSpeed, Tdewpoint,NSM, WeekStatus, Day of Week	28.29	0.92	18.85	24.41	72.45	0.49	40.73	46.53

Figure 29: GBM performance with different parameters. (Candanedo et al., 2017, p. 91). Copyright 2017 by Elsevier B.V.

GBM – no lights (GBM model trained on all features except lights) in Figure 29 is very similar to the GBM model in Figure 28. For *GBM-no lights*, the R^2 value is 0.58 in the testing set. In the second model, the R^2 value has decreased to 0.54. The third and the fourth model share the same R^2 values at 0.49 in the testing set, and 0.93 and 0.92 in the training set.

Candanedo et. al. pointed to weaknesses in their study by the fact that they only analyzed one house, and that they could have found valuable information by analyzing more houses. They argued that they could have found relationships between energy consumption of appliances and other factors like the number of people living in the houses, their age and if they owned pets or not. They mentioned that different patterns for the change in energy use throughout the year could also have been discovered. The location of the weather station could also have had a bearing for better energy prediction results.

2.4. Our idea of combining EA/SIA with DNN

Our proposed method of finding an optimal DNN architecture is to start off by initializing random architectures in a predetermined search space, represented as position vectors. We determine the search space by setting the minimum and maximum number of nodes per hidden layer, and the number of hidden layers.

For example, an architecture that has two hidden layers with 50 nodes each, will be represented as [50 50]. Then we train and evaluate these architectures. The number of hidden layers is represented by the number of elements in the vector, and the elements in the vector represent the number of nodes in the respective layers. Input and output nodes are constant and therefore not represented in the position vector.

Training and evaluating is done twice per architecture to get a more accurate evaluation of each architecture, as random weight and bias initialization leads to different results in different trials for the same architecture. The best result from the two trials is used as a final evaluation of the architecture. Using the evaluations of every position/architecture, we update these positions using an evolutionary algorithm or swarm intelligence algorithm.

This is repeated until we reach the preset number of maximum generations, before returning the best architecture with trained weights and biases for further use. Figure 30 gives an overview of the different steps.

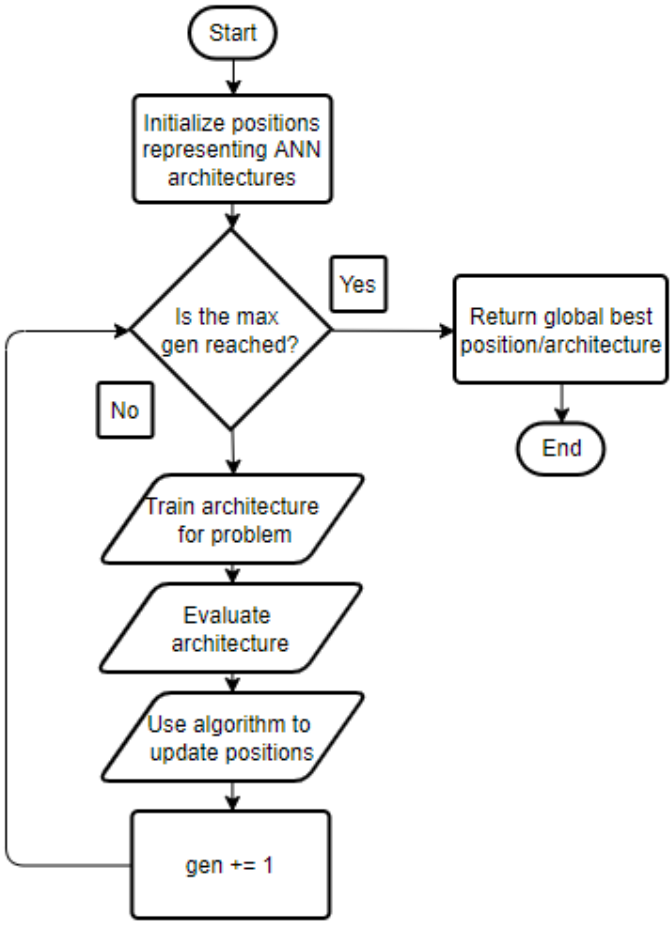


Figure 30: Flowchart showing the steps of combining the algorithms with DNN

The implementation of our idea achieved good results in comparison to existing methods. To present those results, we wrote a research paper and delivered them for peer review. This research paper was about the combination of DE with DNN. The results are delivered to the International Federation of Automatic Control, also known as IFAC[1]. This is multinational organization representing different fields in engineering, sciences, and control and communications. It was founded in September 1957 and has its headquarters in Laxenburg, Austria. As for the two other combinations, we have plans of sending them to a conference in Hong-Kong.

2.5. Determining the details of our implementation

In this section we will be explaining the intricacies of some concepts used in our implementation of the solution. We will be explaining batch normalization, activation functions, Adadelta optimizer, processors, and the concept of multithreading. In addition to this, we will be using tests to determine the optimal activation function and the optimal way of multithreading.

2.5.1. Batch normalization

Batch normalization (batch norm) is a method used in assisting neural network training. It works by adding layers to the neural network that perform normalization on the input of a layer coming from a previous layer. This positively effects the speed of the training process and the stability of the neural network. We use batch normalization in our implementation.

2.5.2. Activation functions

Activation functions are used in Deep Neural Networks after the weighted sum of inputs and biases has been computed, to decide whether a node should be activated or not, and to what degree. Without activation functions, neural networks would just be a fancy linear combination. They allow us to train neural networks to find non-linear relationships between inputs and outputs. One can never know which activation function is the best for a specific problem. We have many different activation functions, and we considered 6 of them for our implementation.

[1] For more details about the research paper sent to IFAC click the link below.
https://github.com/mikeair8/Reasearch-paper/blob/main/IFAC%20HMS2022_DE_final.pdf

2.5.2.1. Sigmoid function

The activation function Sigmoid outputs a value in the range [0, 1], regardless of what the input value is. Equation 7 shows the formula for sigmoid(x) and Figure 31 shows the plot for the sigmoid function.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Equation 7: Sigmoid function

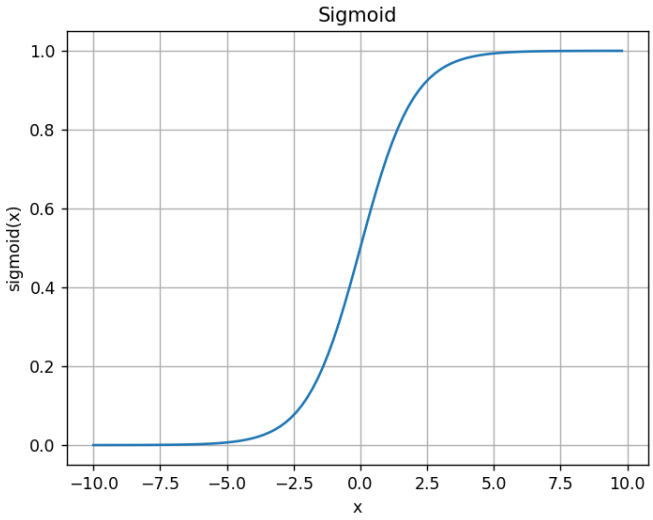


Figure 31: Sigmoid function plot

2.5.2.2. ReLU function

Rectified Linear Unit (ReLU) is,

$$relu(x) = \begin{cases} x, & if\ x > 0 \\ 0, & otherwise \end{cases}$$

Equation 8: ReLU function

Equation 8 shows that if x is less than 0 then the input is set to 0. If on the other hand x is greater than 0, then the input is set to the input value. Figure 32 shows the ReLU function plotted.

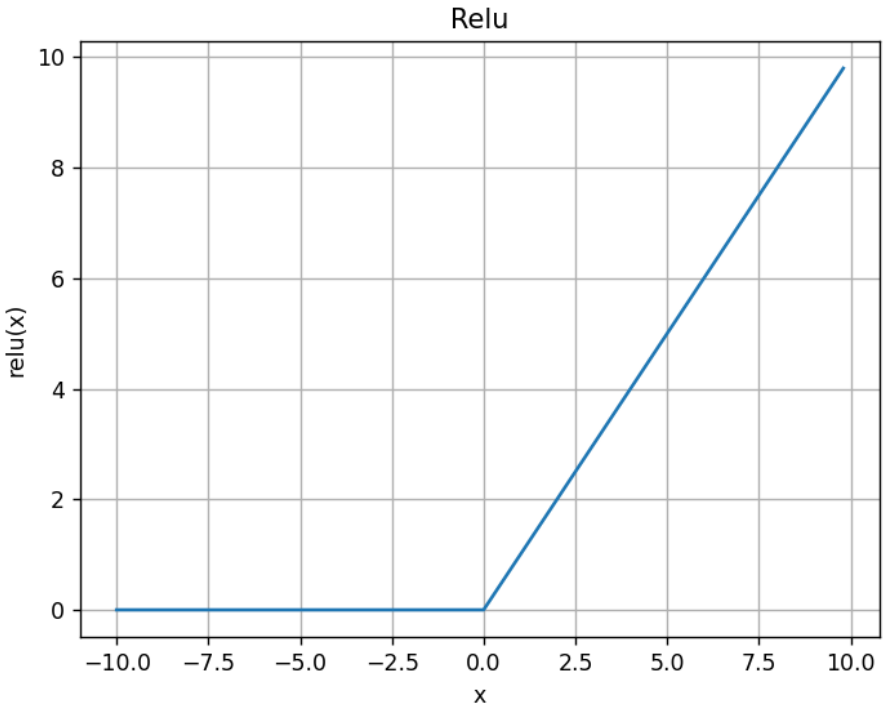


Figure 32: ReLU function plot

2.5.2.3. TanH function

The hyperbolic tangent activation function shown in figure 33 is commonly referred to as TanH function. It outputs a value in the range [-1, 1] when the input is any real value. The larger the input value gets, the closer the output value is to 1, likewise, the smaller the input value is, the closer the output value is to -1 proportionally. The formula for calculating TanH(x) is given in Equation 9 and Figure 33 shows the plot for the TanH function.

$$f(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

Equation 9: TanH function

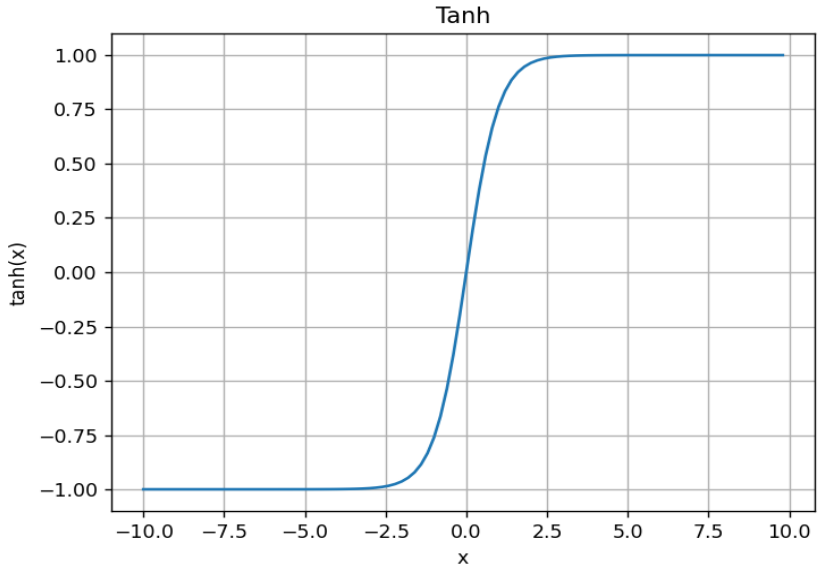


Figure 33: Plot of TanH function.

We can see from the graph that tanh function has a similar s-shape to it as the sigmoid function.

2.5.2.4. Softplus function

Softplus, also called smoothReLU function is a variant of the ReLU activation. The visual difference between softplus, shown in Figure 34 and regular ReLU function, shown in Figure 32 is the smoother shape of its graph compared to ReLU function when plotted. The formula for calculating softplus(x) is given in Equation 10 below. Softplus outputs a value in the range $[0, \infty)$.

$$softplus(x) = \log(1 + e^{x})$$

Equation 10: Softplus function

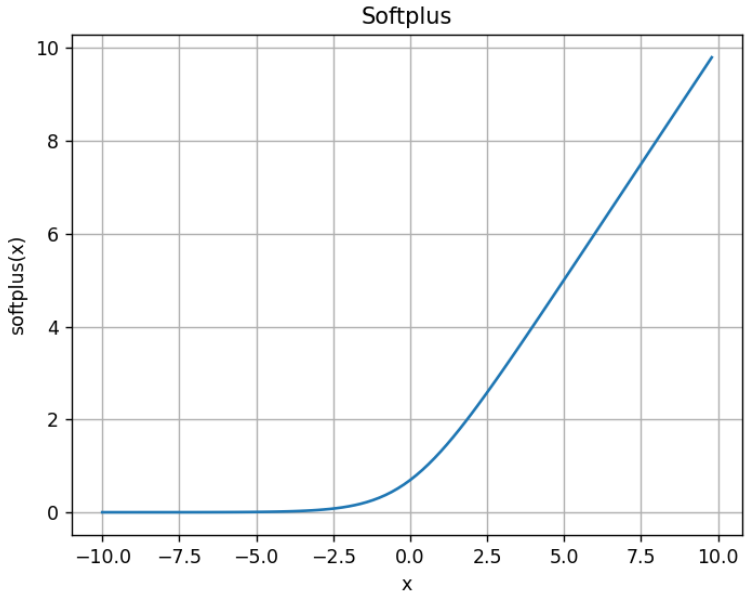


Figure 34: Softplus function plot

2.5.2.5. Swish function

Another activation function similar to ReLU is Swish. Developed in Cornell University, it shows promising results in working on deeper models across range of challenging datasets (Ramachandran et al., 2017). Its simplicity and ability to replace ReLU makes it a good choice. The formula for swish is shown in Equation 11, and Figure 35 shows how it looks when plotted.

$$f(x) = x \times sigmoid(\beta x)$$

Equation 11: Swish function

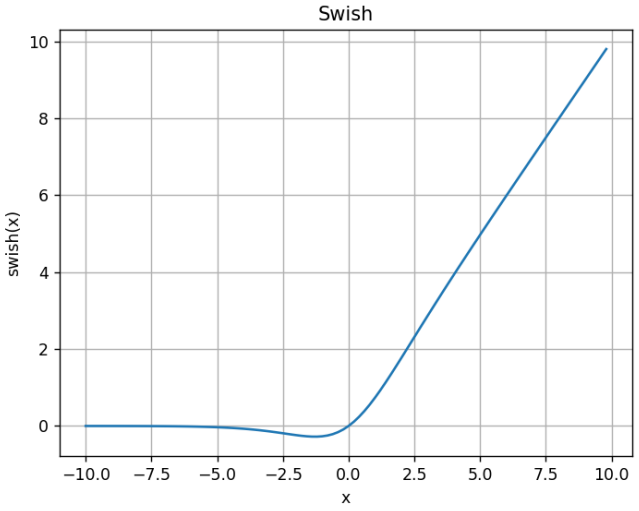


Figure 35: Swish function plot

2.5.2.6. SELU function

The last activation function we tested was Scaled Exponential Linear Unit (SELU). The formula for calculating SELU(x) is given by Equation 12 below.

$$SELU(x) = \lambda \begin{cases} x & if\ x > 0 \\ \alpha e^x - \alpha & if\ x < 0 \end{cases}$$

Equation 12: SELU function

The values for α and λ are both predetermined precise values. This equation shows that if x is greater than 0 then the output value becomes $x\lambda$. If on the other hand x is less than, then the output value is the alpha value multiplied with the exponential of the x-value and subtracted from the alpha value, multiplying it with the lambda value. Figure 36 shows the SELU function plotted.

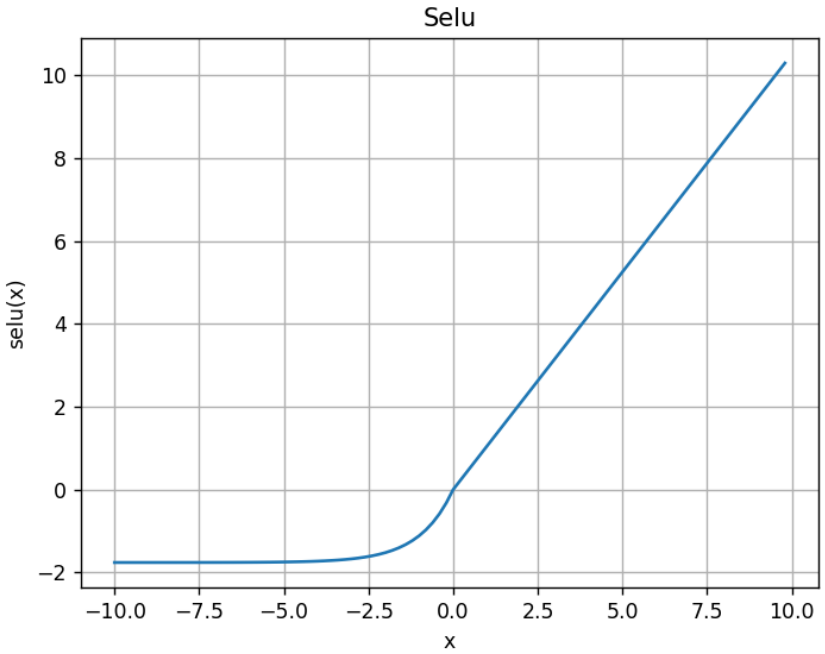


Figure 36: SELU function plot

2.5.3. Testing activation functions

To determine which activation function to use in the hidden layers, we chose an architecture with two hidden layers and 50 nodes each. The same architecture was tested with 6 different activation functions and trained 80 times per activation function. We documented the results to compare the activation functions with statistical analysis. Looking at softplus, we can see that is has the best minimal MSE, and the tightest gap between min and 75th percentile, and we concluded to use softplus. Table 12 shows the results from each activation function.

Table 12: Results of all tested activation functions, based on validation MSE

	Sigmoid	ReLU	TanH	**Soft-plus**	Swish	SELU
max	4406.78	2846.58	3463.97	**4406.78**	2959.05	2938.29
75th percentile	3328.06	2728.05	3198.45	**2688.47**	2792.00	2790.47
Mean	3187.23	2673.86	3092.96	**2806.57**	2708.78	2690.13
25th percentile	2745.68	2610.17	2968.54	**2587.97**	2623.95	2594.53
min	2536.95	2463.03	2634.70	**2452.91**	2470.59	2486.14
std	704.97	78.55	183.01	**539.83**	118.73	109.43

2.5.4. Adadelta optimizer

A recurring problem that optimizers face is the inevitable decay of learning rates while training a neural network. To solve this, adaptive learning methods are used. One such method is Adadelta, which is an improved version of Adaptive Gradient method (commonly referred to as Adagrad). It is a stochastic gradient descent method that addresses issues faced by neural networks by continuously adapting its learning rates to each gradient update instead of using all past gradients. We chose to have a two-phase training process for our DNNs where we use the common SGD optimizer in the first phase and Adadelta optimizer in the second phase for refinement.

2.5.5. CPU & GPU

When working with artificial intelligence, we as engineers must decide if we are going to run the algorithms on our GPU or CPU, depending on the problem. There is no clear guidance on how to use GPU and CPU for every application.

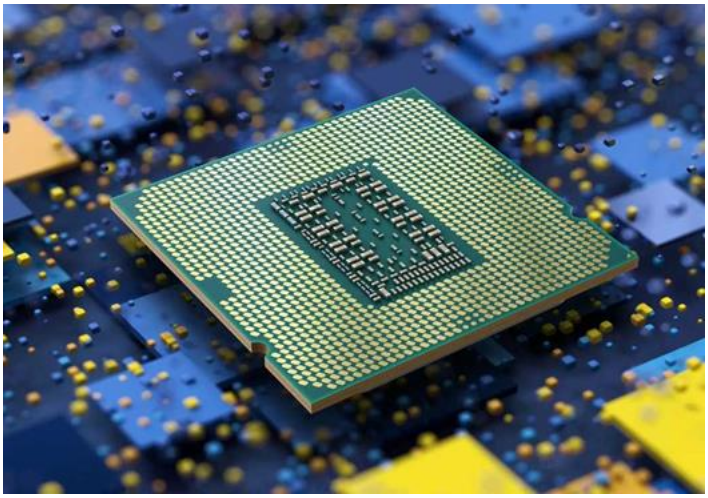


Figure 37: A picture of a CPU without the metal plate cover (Leather, A., 2021). Copyright 2022 by Forbes Media LLC.

Central Processing Unit or CPU, shown in Figure 37 is what handles most of the daily calculations on your computer. They are called general-purpose processors for this reason; they can compute almost any calculation. If you run any algorithm on your pc without specifying which component to run it on, it will run on the CPU, as that's the default. When talking about CPU's we also must mention some key aspects, Clock speed, cores, and temperature.

Clock speed is measured in Hz and tells us how many cycles the CPU will execute per second. most CPUs today will have a max clock speed of around 3GHz, which will result in 3 billion cycles per second. In the early days of CPUs, they only had one core that did all the work. These days the average CPU has 4 or 8 cores which divide the work between them, with a process called simultaneous multithreading. Each core is basically a CPU in and of itself. Having multiple cores in a single CPU allows us to divide the work required to optimize our neural networks between these cores to increase productivity.

The more load on a CPU, the more power it consumes, which in turn results in an increase in temperature. Power consumption and temperature have a positive linear correlation. Therefore, it's important to have suitable cooling, to prevent the CPU from overheating and be able to use most of its computing power.

Graphics Processing Unit or GPU, shown in Figure 38 is in most cases used to generate graphics in computers among other things, that's where its name comes from. While a CPU will handle most tasks it doesn't excel in repetitive simple tasks, and that's where the GPU comes in. As mentioned earlier a CPU has maybe around 4-8 cores, while a GPU will have thousands. Each core in a CPU is of course superior to GPU cores, but GPUs make up for what they lack in quality, with quantity. GPUs are for example much better suited for matrix multiplications in addition to some other tasks. TensorFlow has made their library compatible with GPU, therefore using GPU for Deep Learning isn't as difficult as it used to be.

Figure 38: A picture of a GPU without a heatsink and fans, which are generally included (Shilov, A., 2019). Copyright 2022 by Anandtech.

2.5.6. Multithreading

Multithreading is a Central Processing unit (CPU) feature that allows for threads to execute tasks in parallel or by switching between threads using very little time. The threads share resources from multiple cores or a single core. Multithreading improves the ability of an operative system (OS) to function with comparative speed and avoid overloading the system. In our test we experimented with different amounts of threads and noted the results, where each thread has the task of optimizing a neural network. In table 13 we can see that multithreading, using six threads at once used the least time (17.7 minutes) compared to the rest. This means that the fastest way to train a total of 60 different neural networks is to train 6 by 6.

Table 13: Time consumption when using different numbers of threads at once to complete a total of 60 threads

Threads	Time (sec, min)
4	1260 sec (21 min)
6	1062 sec (17.7 min)
10	1114 sec (18.56 min)

12	1577 sec (26.28 min)

2.6. Running the algorithms on a server

We tried running our algorithmic combinations on our own laptops to begin with. Note that we combine evolutionary algorithms with multiple neural networks trained at once, resulting in very computationally expensive combinations. We set the number of generations to 30, however the program simply crashed at around 14 to 16 generations after around 12 hours, every time we tried to run it, possibly because we only had 16GB RAM.

To solve this, we went to the website Vast.ai and rented a powerful server with 3x RTX 3090 GPU's and AMD Ryzen Threadripper CPU with 16 CPU cores. Once we were connected via SSH-protocol to the server, we transferred our files using WinSCP.

When everything was set up, we used the Linux command taskset to assign 8 CPU cores to one algorithmic combination, and 8 separate CPU cores to another algorithmic combination, (see Appendix A.9.6 p.116). They were also assigned one GPU each, using a couple lines of code in python (see Appendix A.9.7 p.116).

This resulted in a speed increase of around 50%, as well as stability since the programs didn't crash anymore. We were successfully able to complete all 30 generations.

3. Implementing Deep Neuroevolutionary algorithms and Energy prediction results

After successfully developing an algorithm portfolio and determining the details of our idea and how to implement it, we will run the experiments and document the results.
This chapter contains three main subsections, namely PSO, DE, MA combined with DNN. Under each subsection we will also discuss the encoding of DNNs, parameters used in the experiments, the implementation processes, and lastly results.

3.1. DNN combined with PSO

In this work we combine the PSO algorithm and backpropagation to optimize DNN's. The PSO will be used for optimizing the number of nodes in the hidden layers of an DNN with a predetermined number of hidden layers.

3.1.1. Encoding

We represent DNN architecture as a position vector where the first component represents the number of nodes in the first hidden layer, the second component represents the number of nodes in the second hidden layer and so on. The input and output layers are not included as they are predetermined and constant.

Here's an example of an DNN with 50 nodes in the first hidden layer and 50 nodes in the second hidden layer, represented as a position vector [50 50].

The positions are then normalized to numbers between 0 and 1, with the goal of a faster swarm convergence.

3.1.2. PSO search

Table 14 shows the parameters we used in our PSO experiment.

Table 14: Overview of the parameter settings used in the PSO search

Parameter	**Description**	**Value**
npart	Number of particles in the swarm	20
ndim	Number of dimensions in the search space of the swarm	6
m	Max number of swarm generations	30

c_1	Cognitive parameter, tells us how much the particles should value their own personal best positions when updating to new positions	1.49
c_2	Social parameter, tells us how much the particles should value the global best position when updating to new positions	1.49
w	How much of the previous generation velocity should be reused when updating particle positions	0.9, decreased linearly over the generations down to 0.6
minnodes	Minimum number of nodes in a hidden layer	30
maxnodes	Maximum number of nodes in a hidden layer	60
n_inputs	Number of input nodes	30
n_outputs	Number of output nodes	1
nhidden	Number of hidden layers	6
batch_size	Number of datapoints to use per parameter update	250
epochs	Number of times the whole dataset is revised by the neural network during the training process	1300
patience	Number of epochs without improvement before stopping the training process of a	50

optimizer	Optimization algorithm used for updating the parameters in the neural network	SGD and Adadelta
lr	Learning rate, how much we take a step in the proposed direction of the optimizer to update DNN parameters	0.01, decreased by 25% if there's no no improvement for the last 30 epochs
Hidden layer activation function	Activation function used in the hidden layer nodes	Softplus
Output layer activation function	Activation function used in the output layer node	ReLU
Loss function	The function we use to evaluate the fitness of a particle	MSE (Mean squared error)

Figure 39 shows the change of the worst training and validation MSE in the swarm across 30 generations. We can see that the worst validation MSE and the worst train MSE decreased. Each y value shows the worst MSE present in the swarm in generation x.

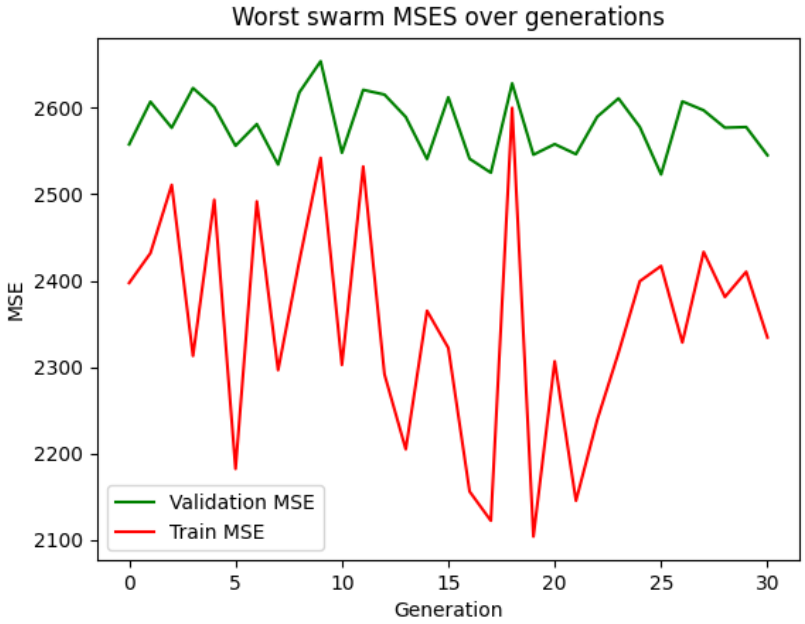


Figure 39: Illustrates the change of the worst swarm MSE over 30 generations

Figure 40 shows the change of the mean training and validation MSE in the swarm across 30 generations. We can see that the mean validation MSE and the mean train MSE decreased. Each y value shows the mean MSE present in the swarm in generation x.

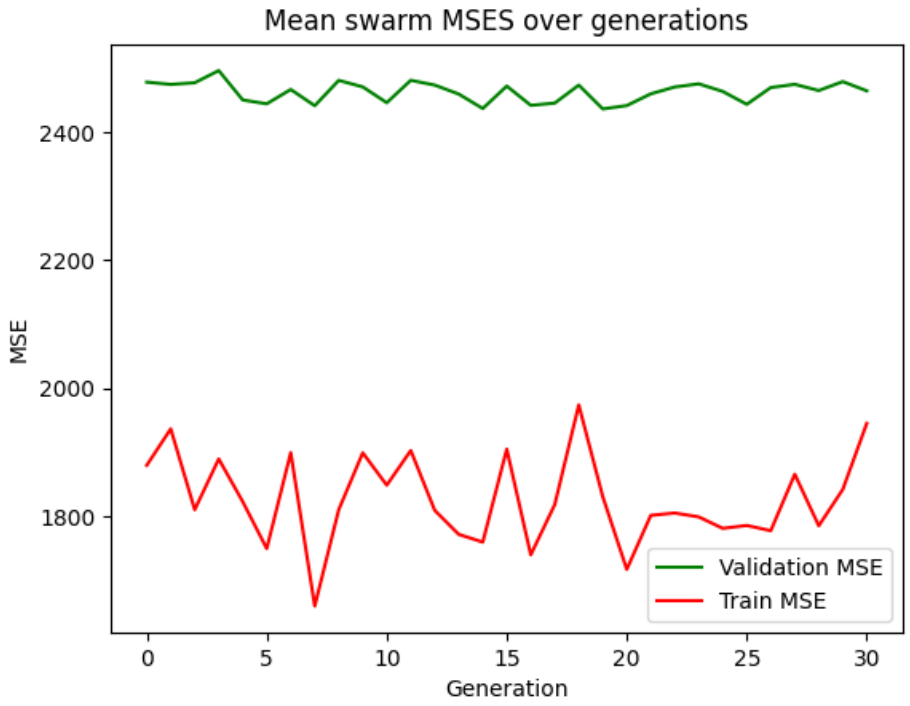


Figure 40: Illustrates the change of the mean swarm MSE over 30 generations

Figure 41 shows the change of the best training and validation MSE in the swarm across 30 generations. We can see that the best validation MSE and the best train MSE decreased. The architecture with the lowest validation MSE was found in the 10th generation. Each y value shows the best MSE present in the swarm in generation x.

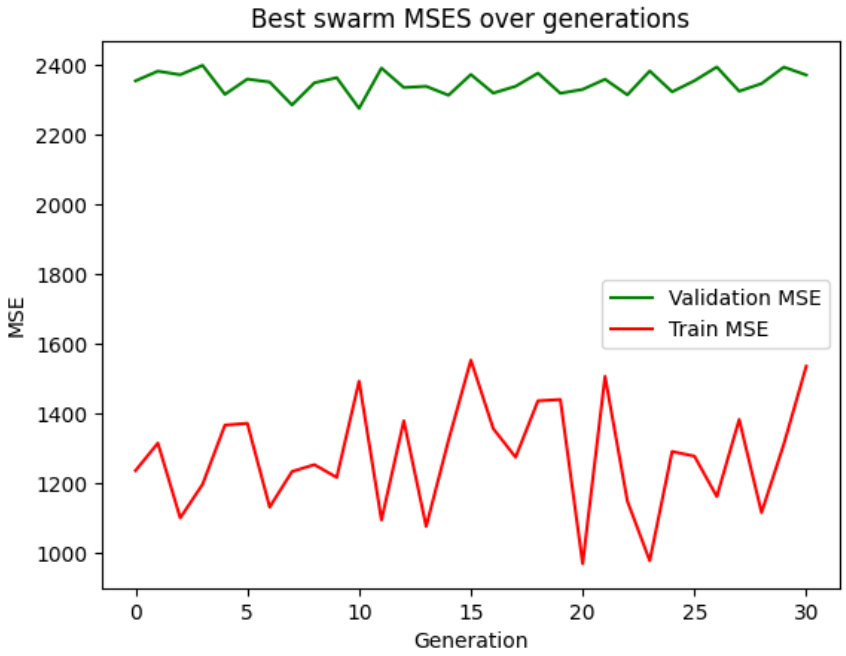


Figure 41: Illustrates the change of the best swarm MSE over 30 generations

3.1.3. Two-phased training process

Figure 42 shows the first training phase of the best architecture found, using SGD optimizer. We can see that both validation and training MSE decreased up until around 250 epochs.

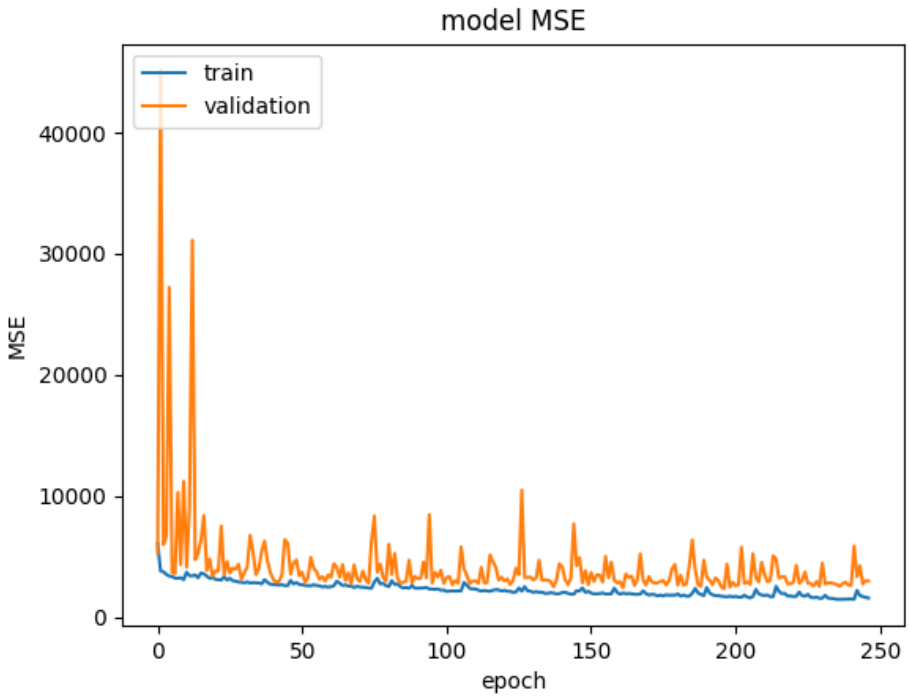


Figure 42: Illustrates the change of the best architecture MSE over epochs in the first training phase using SGD optimizer

Figure 43 shows the second training phase of the best architecture found, using Adadelta optimizer. We can see that we were able to further decrease the validation MSE in the second training phase.

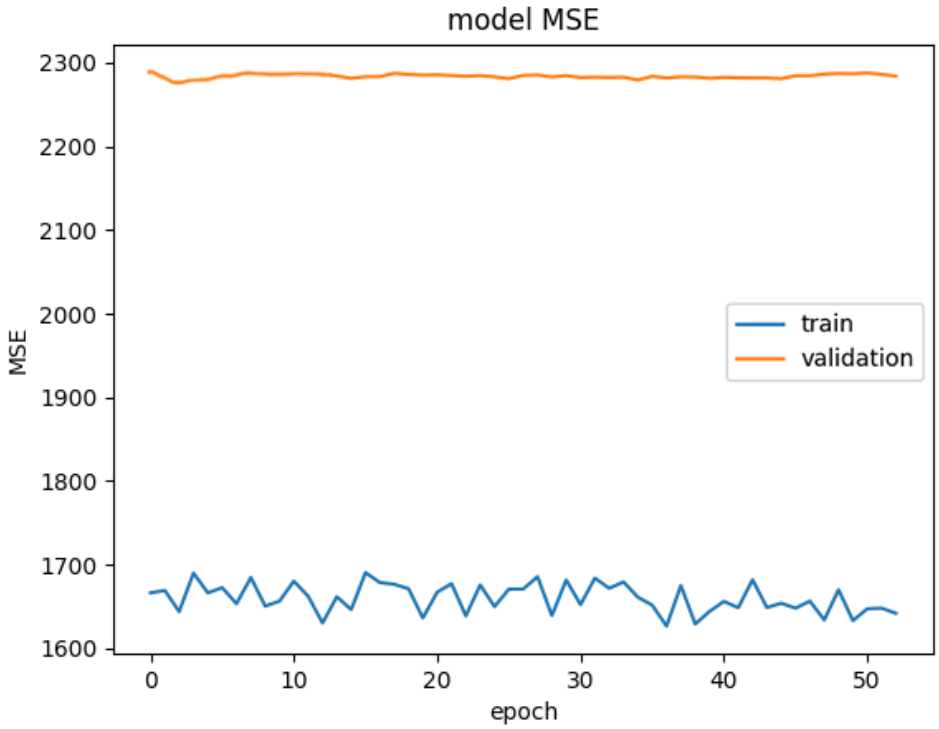


Figure 43: Illustrates the change of the best architecture MSE over epochs in the second training phase using Adadelta optimizer

3.1.4. PSO search results

The PSO search results are summarized in table 15.

Table 15: Summary of results from the PSO search

Best validation MSE in generation 0	2355.72
Best architecture found	30-47-31-30-43-35-44-1
Best architecture validation MSE	2276.71
Generation in which the best architecture was found	10
Swarm MSE reduction	79.01

Figure 44 represents the best architecture found by PSO. We can see an interesting architecture with varying layer sizes.

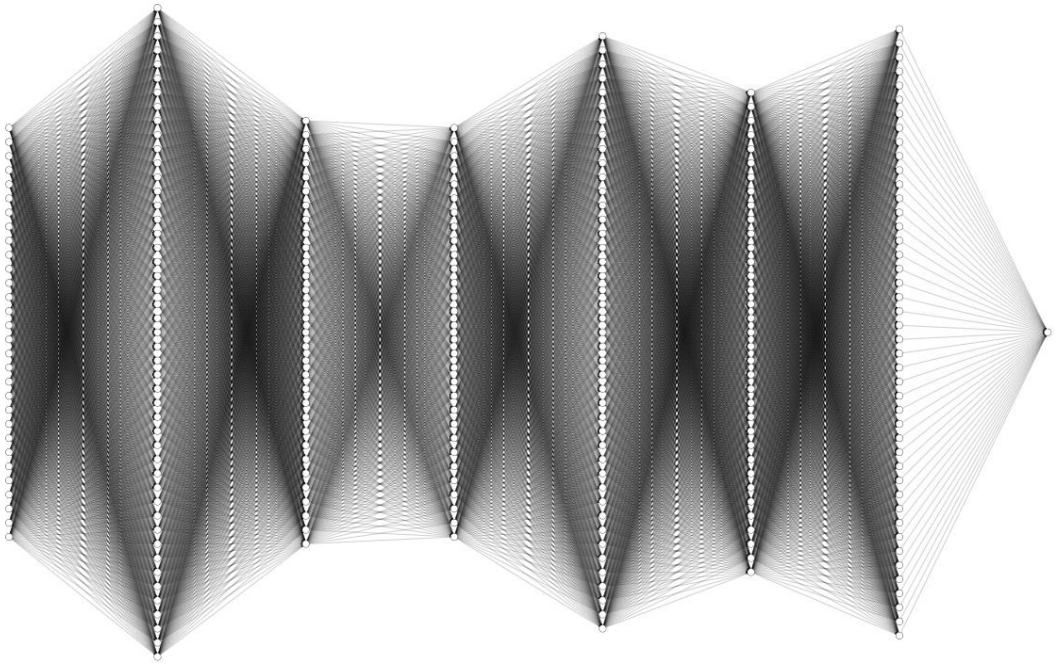


Figure 44: Visualization of the best architecture found by PSO (30-47-31-30-43-35-44-1)

Table 16 shows how the best architecture performs on the training, validation, and test set. We can see that the model performs better on the training- and validation sets than on the test set, which is expected.

Table 16: Table showing the metrics of the best architecture in train, validation, and test sets

Metric results using PSO-DNN combination											
Train				Validation				Test			
RMSE	R^2	MAE	MAPE	RMSE	R^2	MAE	MAPE	RMSE	R^2	MAE	MAPE
39.02	0.67	21.50	25.74	47.71	0.48	25.01	30.27	52.96	0.38	26.64	30.23

Statistical analysis of model testing errors

Getting a deeper understanding of the testing error is an important step to get practical knowledge about the model. Without digging into it, we only have the metrics to tell us how it is performing. The metrics do not give us a clear enough image of which instances we have good accuracy on and which ones we do not. Table 17 shows statistical error measurements to gain a more profound understanding of how the proposed model performs.

Table 17: The statistics of the testing errors defined by the actual value minus the predicted value on every instance in the test set.

Standard Deviation	52.79
Max	348.53
75^{th} percentile	9.48
Mean	4.23
25^{th} percentile	-14.29
Min	-226.05
Proportion of negative errors	0.55
Proportion of positive errors	0.45

Notice that the model usually predicts too high values as indicated by the proportion of negative errors being over 0.5, but the mean error being positive means that the magnitude of too low predictions outweighs the magnitude of too high predictions. The max error is 348 and the minimum error is -226. This means that the worst error was when a prediction was lower than the actual value.

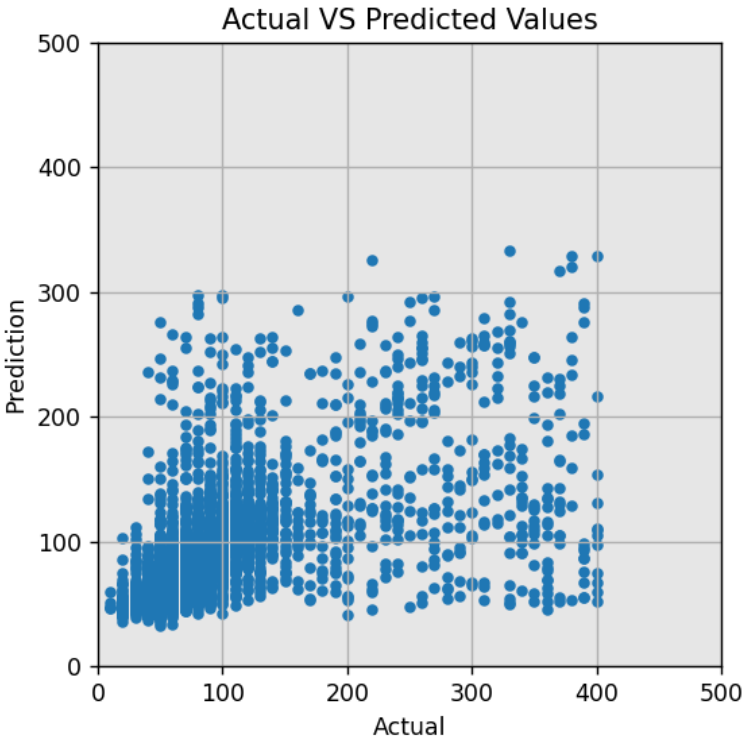


Figure 45: The predicted vs. actual energy use.

Figure 45 shows predictions as a dependent variable of actual values. A perfect model with zero error would show us points following a linear relationship with a slope of 1. When the actual values are in the lower range of approximately 10-150, the slope looks relatively good. However, values above 150 are clearly harder to predict for the model, where it frequently predicts too low values.

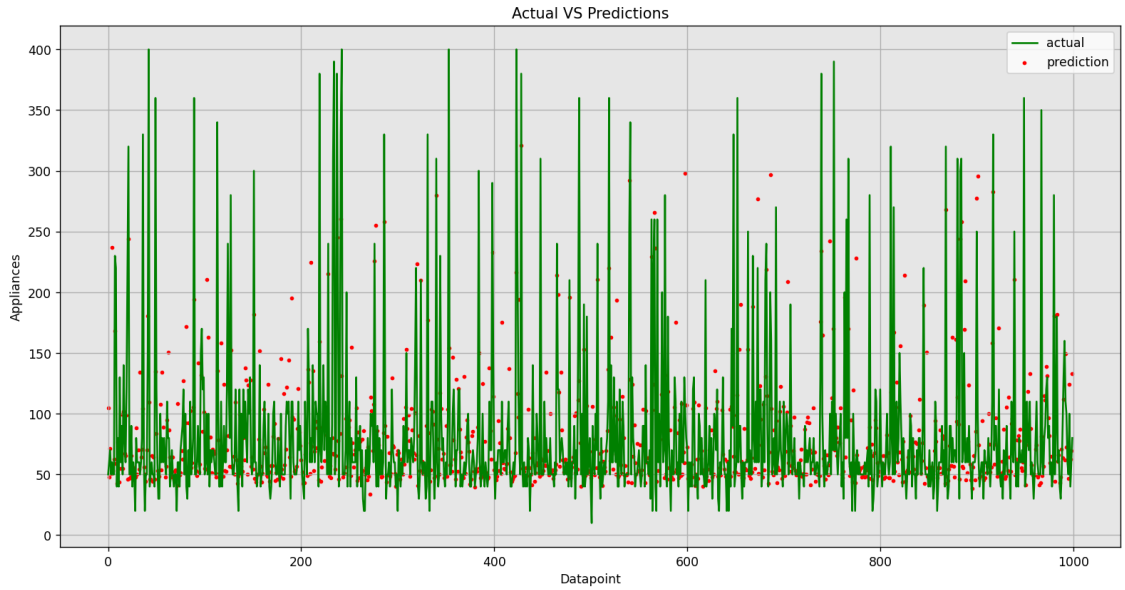


Figure 46: The predicted vs. actual energy use (only the first 1000 instances in test dataset shown here as a sample).

Figure 46 confirms that we have better accuracy when the actual values are in the lower range. When the actual values are in the higher range, the model usually predicts values much lower than the actual values. We can clearly see that when the actual values are higher than 150, the model has difficulty making accurate predictions.

Figure 47 shows that the majority of errors are negative (too high predictions), however we have many positive errors with greater magnitude.

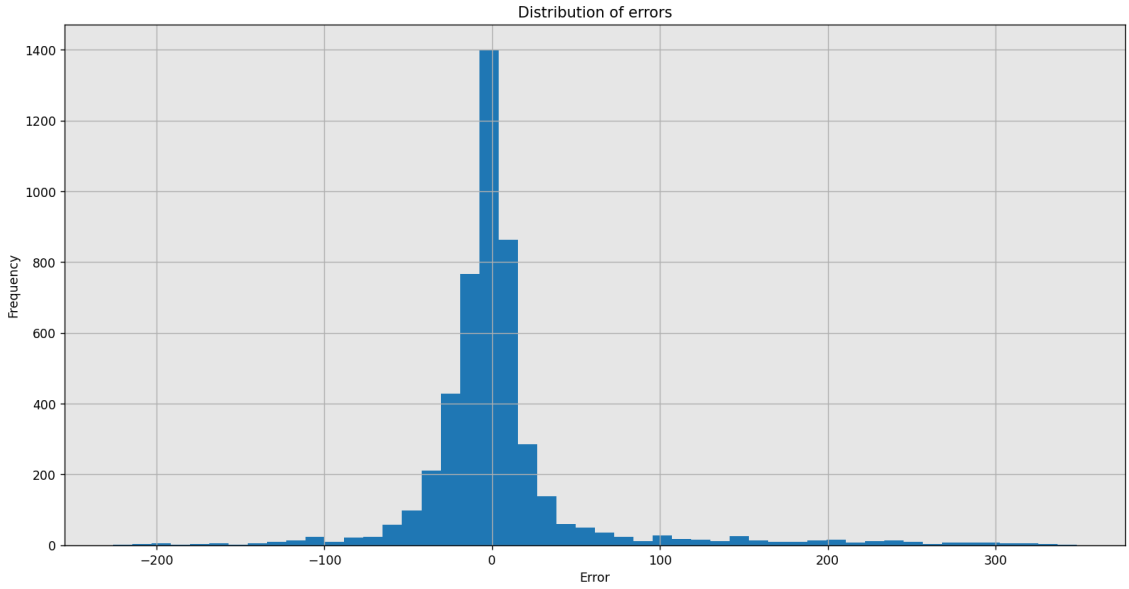


Figure 47: The distribution of the testing errors.

Table 18 shows comparison of errors from too high predictions vs errors from too low predictions (in comparison with the actual values). In this table we use the absolute value of the testing errors, even though too high predictions result in negative errors and too low predictions lead to positive errors.

Table 18: The statistics of the absolute values of the errors due to too low and too high predictions.

	Absolute errors due to too high predictions	Absolute errors due to too low predictions
Standard Deviation	26.59	60.48
Max	226.05	348.53
75^{th} percentile	25.79	27.49
Mean	20.58	33.90
25^{th} percentile	5.41	4.43
Min	0.0013	0.01

Notice that 75% of the too-high predictions are between 0.0013 to 25.79 different from the actual value, while 75% of too-low predictions are between 0.01 and 27.49 different from the actual value.

To sum it all up, the test error analysis shows that the model frequently predicts values higher than the actual value, however this is outweighed by the magnitude of the error made when predicting values lower than the actual value. When the actual value is in the higher range, the model has difficulty predicting it accurately.

3.1.5. Summary of PSO-DNN experiment

In this work, we present a new approach to finding the optimal structure of deep neural network for predicting the energy consumption of a low-energy house. The PSO algorithm is used to determine the optimal architecture of a neural network.

The results showed that our proposed method reduced MSE by 79.01 from the best of the initial architectures to the best architecture found by PSO algorithm (with a final MSE of 2276.71), after 10 generations.

Testing error analysis shows that 55% of predictions are higher than the actual value (negative errors), and 45% are lower than the actual value (positive errors). Though most errors are negative, the mean error is 4.23 because the magnitude of the positive errors outweigh the magnitude of the negative errors.

3.2. DNN combined with DE

In this work we combine the DE algorithm and backpropagation to optimize DNN's. The DE will be used for optimizing the number of nodes in the hidden layers of an DNN with a predetermined number of hidden layers.

3.2.1. Encoding

We represent DNN architecture as a position vector where the first component represents the number of nodes in the first hidden layer, the second component represents the number of nodes in the second hidden layer, and so on. The input- and output layers are not included as this is predetermined and constant.

Here's an example of an DNN with 50 nodes in the first hidden layer and 50 nodes in the second hidden layer, represented as a position vector [50 50].

The positions are then normalized to numbers between 0 and 1, with the goal of faster swarm convergence.

3.2.2. DE search

Table 19 shows the parameters we used in our DE experiment.

Table 19: Overview of the parameter settings used in the DE search

Parameter	Description	Value
npart	Number of particles in the swarm	20
ndim	Number of dimensions in the search space of the swarm	4
m	Max number of swarm generations	30
CR	Probability of a particle in the swarm being used in crossover	0.5
F	Mutation rate	0.8
mode	Determines the variant of DE that is used to select the donor vector v_1	toggle
cmode	Determines the version of crossover that is used. It can be GA style crossover or Bin (Bernoulli experiments)	bin
minnodes	Minimum number of nodes in a hidden layer	30
maxnodes	Maximum number of nodes in a hidden layer	80
n_inputs	Number of input nodes	30
n_outputs	Number of output nodes	1

n_hidden	Number of hidden layers	4
batch_size	Number of datapoints to use per parameter update	250
epochs	Number of times the whole dataset is revised by the neural network during the training process	1000
patience	Number of epochs without improvement before stopping the training process of a	50
optimizer	Optimization algorithm used for updating the parameters in the neural network	SGD and Adadelta
lr	Learning rate, how much we take a step in the proposed direction of the optimizer to update DNN parameters	0.001, decreased by 25% if there’s no improvement for the last 30 epochs
Hidden layer activation function	Activation function used in the hidden layer nodes	Softplus
Output layer activation function	Activation function used in the output layer node	Relu
Loss function	The function we use to evaluate the fitness of a particle	MSE(Mean squared error)

Figure 48 shows the change of the worst training and validation MSE in the swarm across 30 generations. We can see that the worst validation MSE and the worst train MSE decreased.

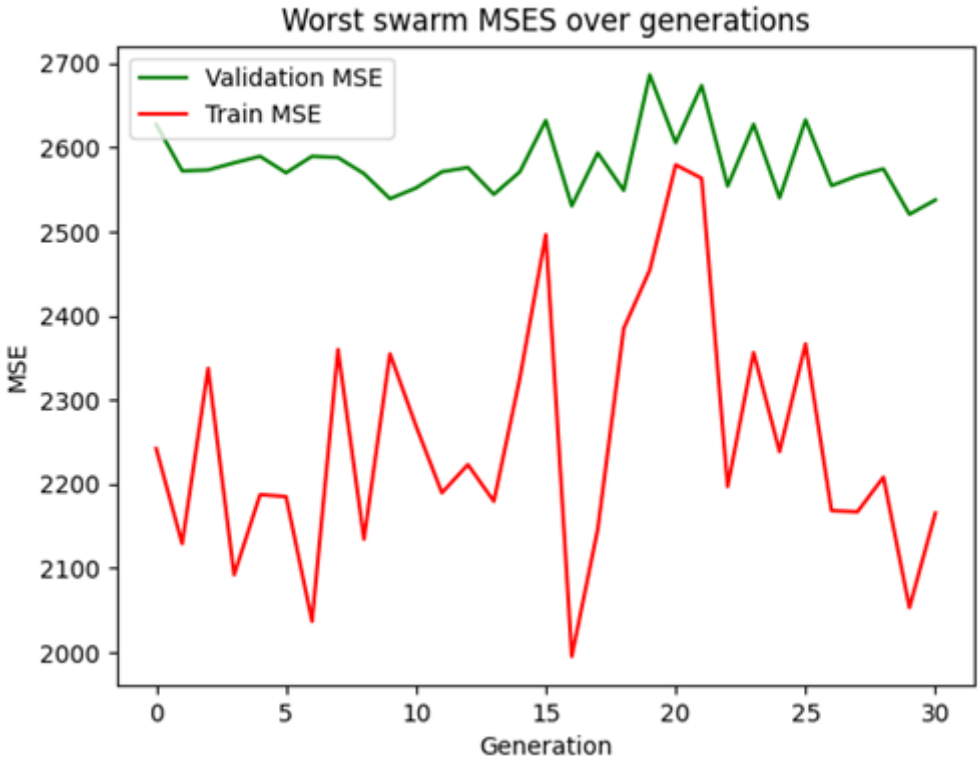


Figure 48: Illustrates the change of the worst swarm MSE over 30 generations

Figure 49 shows the change of the mean training and validation MSE in the swarm across 30 generations. We can see that the mean validation MSE and the mean train MSE decreased.

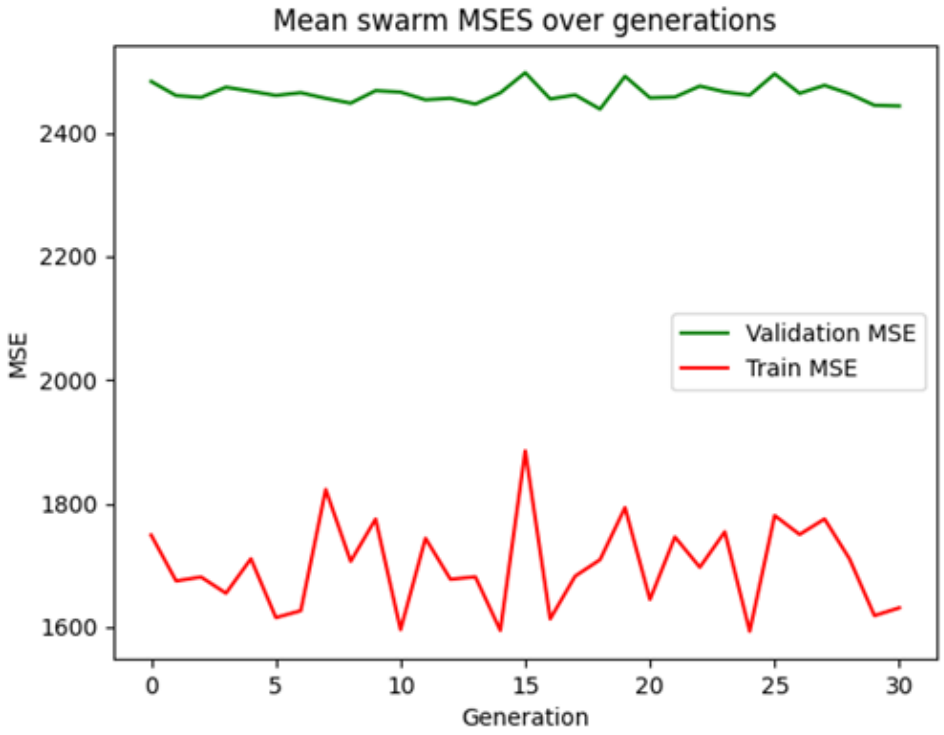


Figure 49: Illustrates the change of the mean swarm MSE over 30 generations

Figure 50 shows the change of the best training and validation MSE in the swarm across 30 generations. We can see that the best validation MSE and the best train MSE decreased. The architecture with the lowest validation MSE was found in the 17th generation.

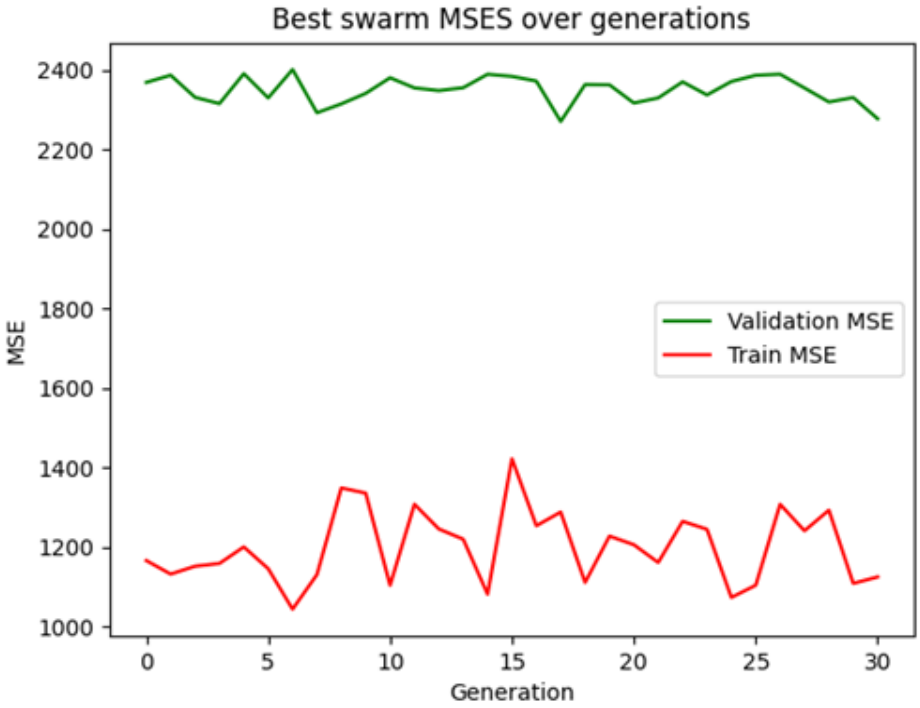


Figure 50: Illustrates the change of the best swarm MSE over 30 generations

3.2.3. Two-phased training process

Figure 51 shows the first training phase of the best architecture found, using SGD optimizer. We can see that both validation and training MSE decreased up until around 200 epochs.

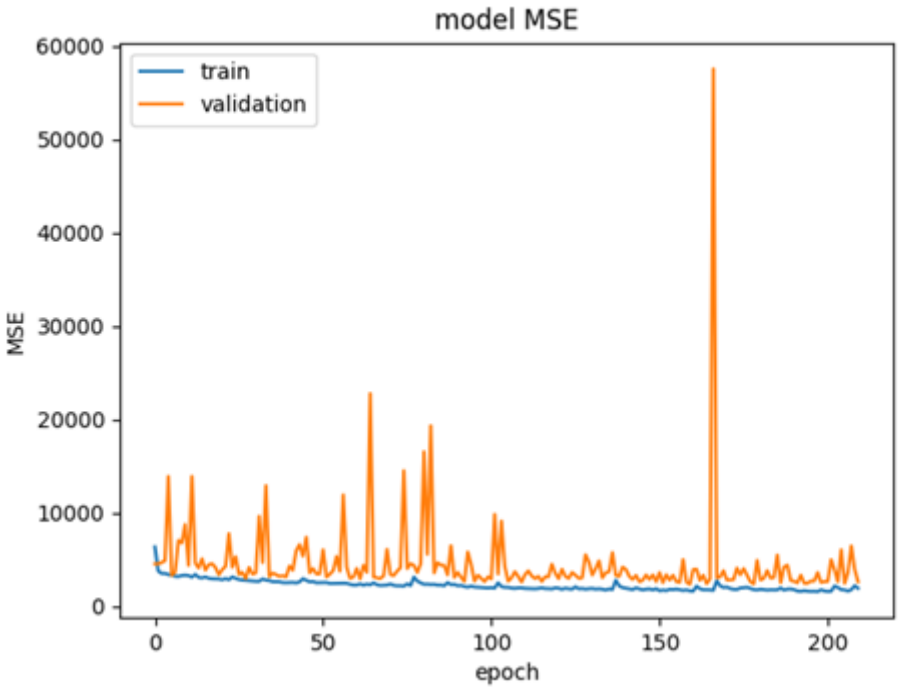


Figure 51: Illustrates the change of the best architecture MSE over epochs in the first training phase using SGD optimizer

Figure 52 shows the second training process of the best architecture found, using Adadelta optimizer. We can see that we were able to further decrease the validation MSE in the second training phase.

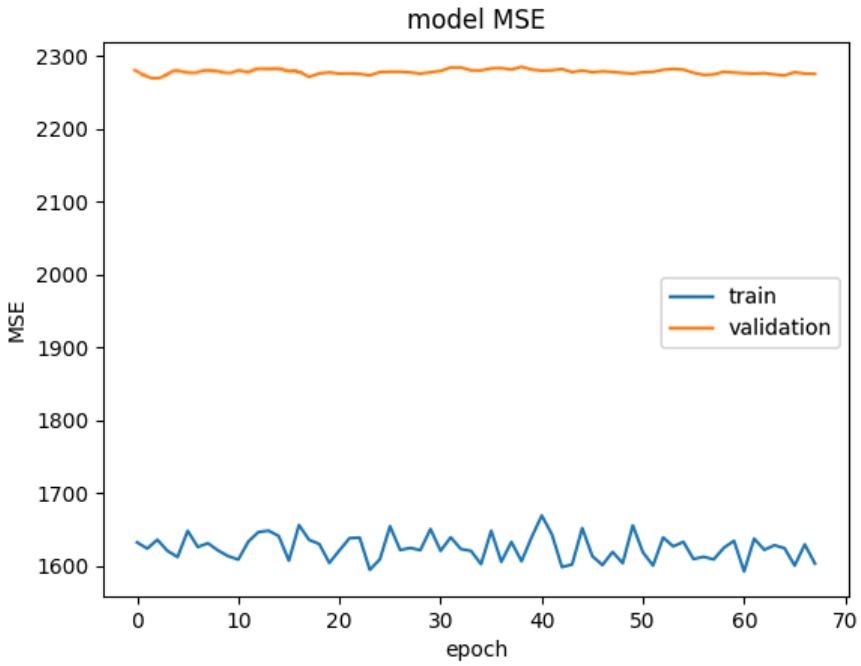


Figure 52: Illustrates the change of the best architecture MSE over epochs in the second training phase using Adadelta optimizer

3.2.4. DE search results

The DE search results are summarized in table 20.

Table 20: Summary of results from the DE search

Best validation MSE in generation 0	2369.34
Best architecture found	30-36-33-53-46-1
Best architecture validation MSE	2271.02
Generation in which the best architecture was found	17
Swarm MSE reduction	98.32

Figure 53 visualizing the best architecture found by DE. We can see that the information is being condensed in the first two hidden layers, before spreading out.

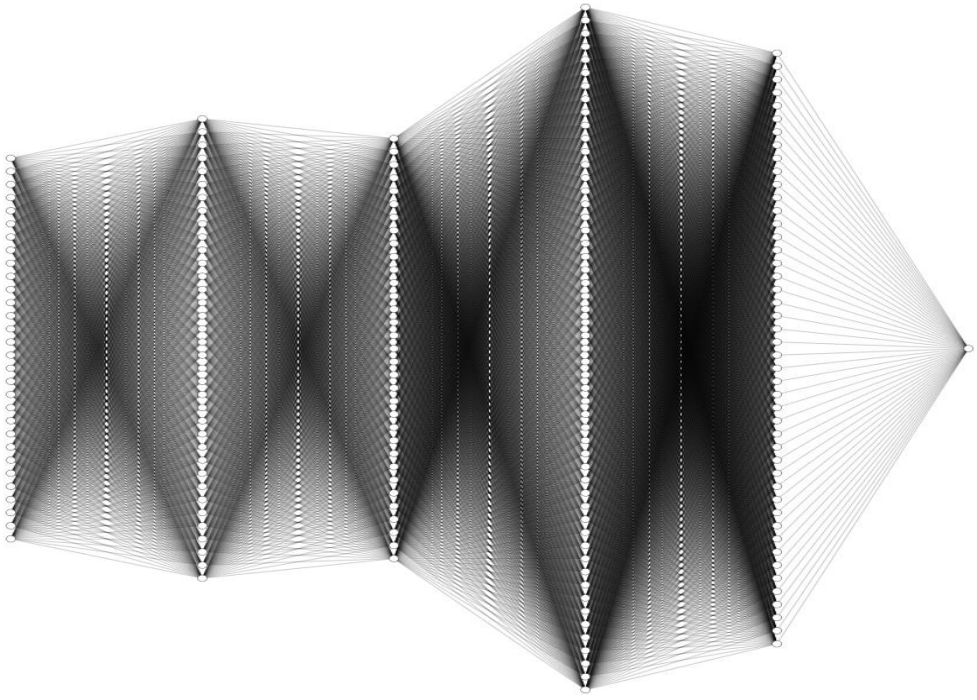


Figure 53: Visualization of the best architecture found by DE (30-36-33-53-46-1)

Table 21 shows how the best architecture performs on the training, validation and test set in terms of metrics. We can see that the model performs better on the training and validation sets than the test set, which is expected.

Table 21: Table showing the metrics of the best architecture in train, validation, and test sets

Metric results using DE-DNN combination											
Train				Validation				Test			
RMSE	R^2	MAE	MAPE	RMSE	R^2	MAE	MAPE	RMSE	R^2	MAE	MAPE
38.49	0.68	21.61	26.80	47.65	0.48	25.88	32.34	52.79	0.38	27.63	32.66

Statistical analysis of model testing errors

Getting a deeper understanding of the testing error is an important step to get practical knowledge about the model. Without digging into it, we only have the metrics to tell us how it really is performing. The metrics do not give us a clear enough image of which instances we have good accuracy on and which ones we do not. Table 22 shows statistical error measurements to gain a more profound understanding of the performance of the proposed model.

Table 22: The statistics of the testing errors defined by the actual value minus the predicted value on every instance in the test set.

Standard Deviation	52.69
Max	333.22
75^{th} percentile	10.09
Mean	3.23
25^{th} percentile	-14.87
Min	-278.82
Proportion of negative errors	0.55
Proportion of positive errors	0.45

Notice that the model usually predicts too high values as indicated by the proportion of negative errors being over 0.5, but the mean error being positive means that the magnitude of too low predictions outweighs the magnitude of too high predictions. The max error is 333 and the minimum error is -278. This means that the worst error was when a prediction was lower than the actual value.

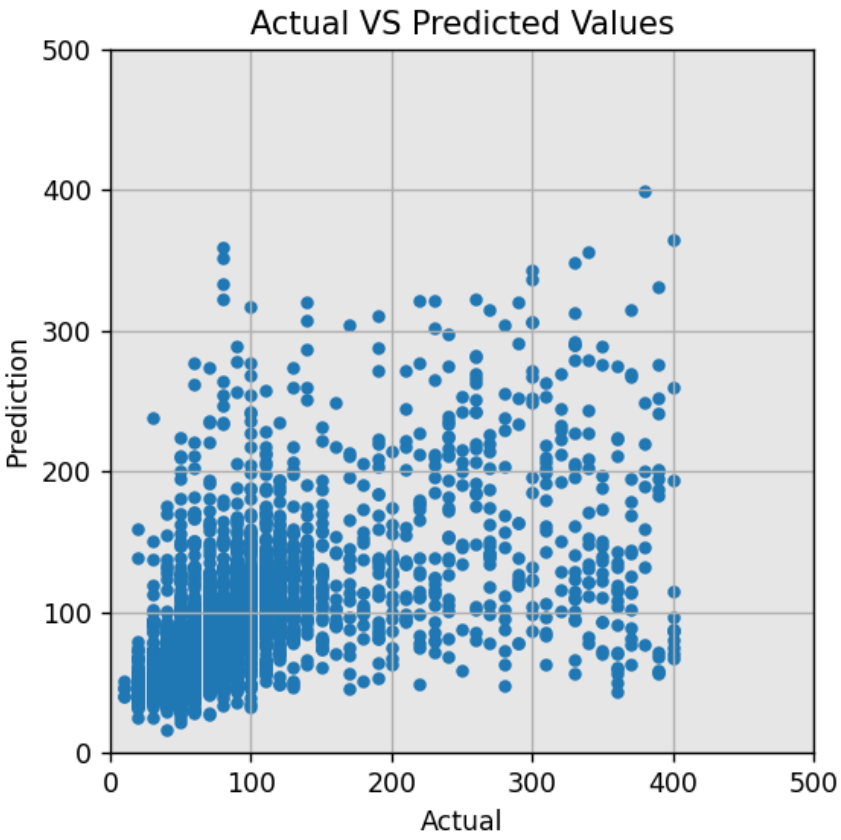


Figure 54: The predicted vs. actual energy use.

Figure 54 shows predictions as a dependent variable of actual values. A perfect model with zero error would show us points following a linear relationship with a slope of 1. When the actual values are in the lower range of approximately 10-150, the slope looks relatively good. However, values above 150 are clearly harder to predict for the model, where it frequently predicts too low values.

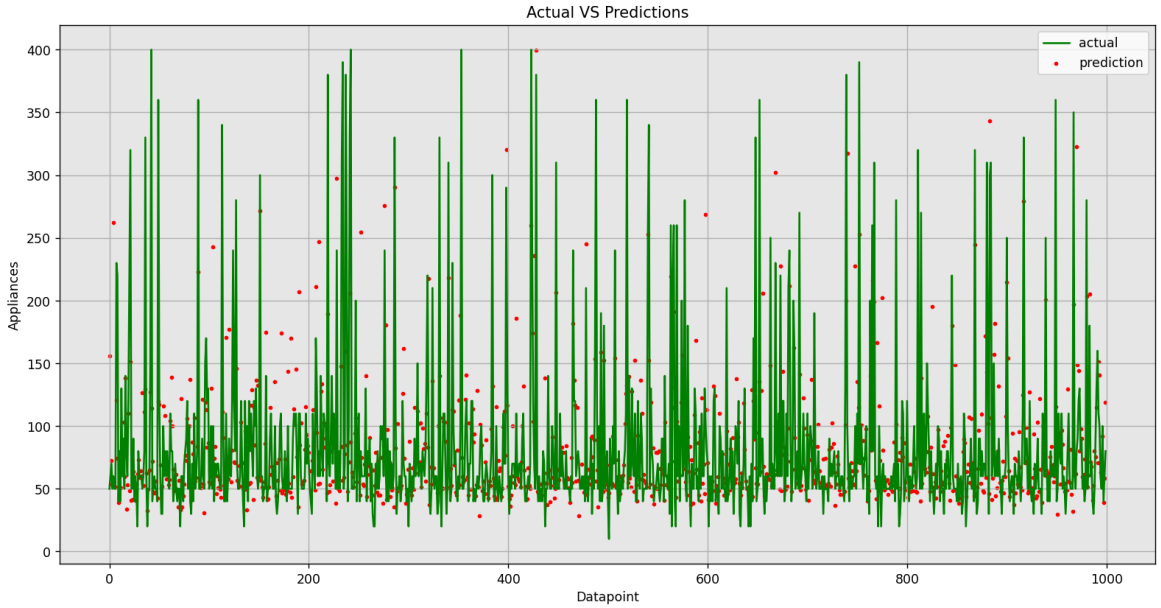


Figure 55: The predicted vs. actual energy use (only the first 1000 instances in test dataset shown here as a sample).

Figure 55 confirms that we have better accuracy when the actual values are in the lower range. When the actual values are in the higher range, the model usually predicts values much lower than the actual values. We can clearly see that when the actual values are higher than 150, the model has difficulty making accurate predictions.

Figure 56 shows that the majority of errors are negative (too high predictions), however we have many positive errors with greater magnitude.

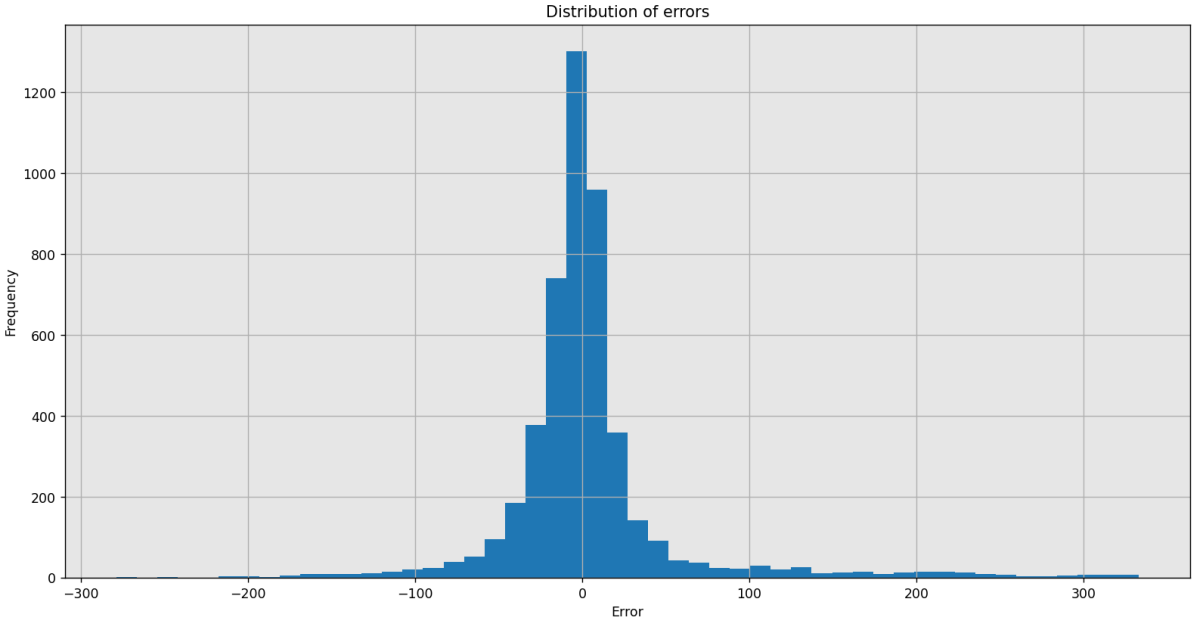


Figure 56: The distribution of the testing errors.

Table 23 shows comparison of errors from too high predictions vs errors from too low predictions (in comparison with the actual values). In this table we use the absolute value of the testing errors, even though too high predictions result in negative errors and too low predictions lead to positive errors.

Table 23: The statistics of the absolute values of the errors due to too low and too high predictions.

	Absolute errors due to too high predictions	Absolute errors due to too low predictions
Standard Deviation	29.00	58.13
Max	278.02	333.22
75^{th} percentile	26.72	29.00
Mean	22.23	34.20
25^{th} percentile	5.56	5.38
Min	0.006	0.01

Notice that 75% of the too high predictions are between 0.006 to 26.72 different from the actual value, while 75% of too low predictions are between 0.01 and 29.00 different from the actual value.
To sum it all up, the test error analysis shows that the model more frequently predicts values higher than the actual value, however this is outweighed by the magnitude of the error made when predicting values lower than the actual value. When the actual value is in the higher range, the model has difficulty predicting it accurately.

3.2.5. Summary of DE-DNN experiment

In this work, we present a new approach to finding the optimal structure of deep neural network for predicting the energy consumption of a low-energy house. The DE algorithm is used to determine the optimal architecture of a neural network.

The results showed that our proposed method reduced MSE by 98.32 from the best of the initial architectures to the best architecture found by DE algorithm (with a final MSE of 2271.02) after 17 generations.

Testing error analysis shows that 55% of predictions are higher than the actual value (negative errors), and 45% are lower than the actual value (positive errors). Though most errors are negative, the mean error is 3.23 because the magnitude of the positive errors outweigh the magnitude of the negative errors.

3.3. DNN combined with MA

In this work we combine the MA and backpropagation to optimize DNN's. The MA will be used for optimizing the number of nodes in the hidden layers of an DNN with a predetermined number of hidden layers.

3.3.1. Encoding

We represent DNN architecture as a position vector where the first component represents the number of nodes in the first hidden layer, the second component represents the number of nodes in the second hidden layer, and so on. The input and output layer is not included as this is predetermined and constant.

Here's an example of an DNN with 50 nodes in the first hidden layer and 50 nodes in the second hidden layer, represented as a position vector [50 50].

The positions are then normalized to numbers between 0 and 1, with the goal of faster swarm convergence.

3.3.2. MA search

Table 24 shows the parameters we used in our MA experiment.

Table 24: Overview of the parameter settings used in the MA search

Parameter	Description	Value
npart	Number of particles in the swarm	20
ndim	Number of dimensions in the search space of the swarm	3
m	Max number of swarm generations	30
CR	Probability of a particle in the swarm being used in crossover	0.5
F	Mutation rate	0.8
mode	mode	

Cmode	cmode	
minnodes	Minimum number of nodes in a hidden layer	30
maxnodes	Maximum number of nodes in a hidden layer	80
n_inputs	Number of input nodes	30
n_outputs	Number of output nodes	1
n_hidden	Number of hidden layers	3
batch_size	Number of datapoints to use per parameter update	250
epochs	Number of times the whole dataset is revised by the neural network during the training process	1000
patience	Number of epochs without improvement before stopping the training process of a	50
optimizer	Optimization algorithm used for updating the parameters in the neural network	SGD and Adadelta
lr	Learning rate, how much we take a step in the proposed direction of the optimizer to update DNN parameters	0.001, decreased by 25% if there’s no improvement for the last 30 epochs
Hidden layer activation function	Activation function used in the hidden layer nodes	Softplus
Output layer activation function	Activation function used in the output layer node	Relu

Loss function	The function we use to evaluate the fitness of a particle	MSE(Mean squared error)

Figure 57 shows the change of the worst training and validation MSE in the swarm across 30 generations. We can see that the worst validation MSE and the worst train MSE decreased.

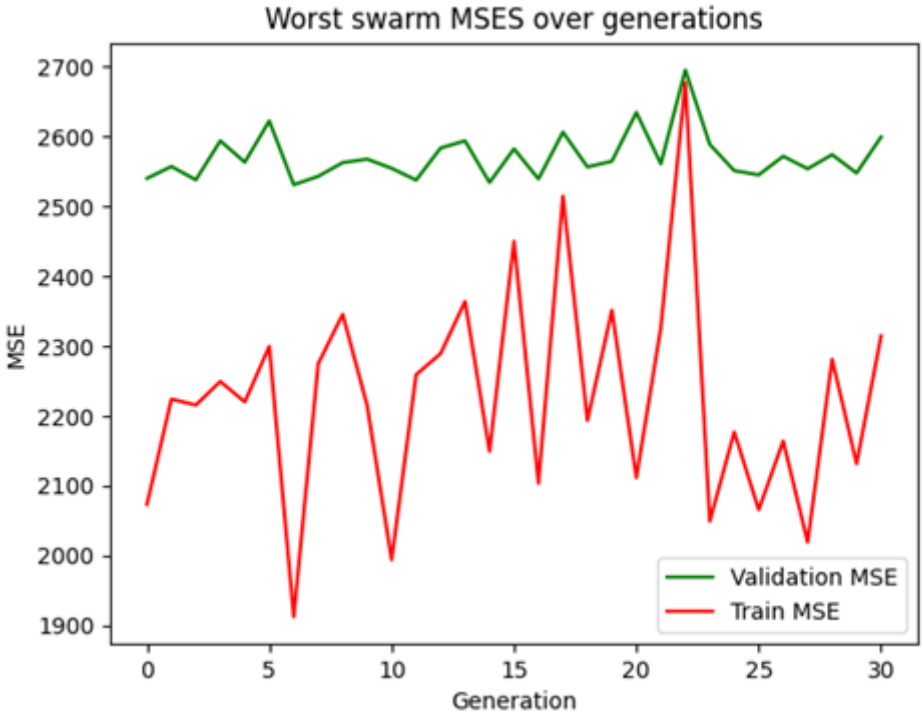


Figure 57: Illustrates the change of the worst swarm MSE over 30 generations

Figure 58 shows the change of the mean training and validation MSE in the swarm across 30 generations. We can see that the mean validation MSE and the mean train MSE decreased.

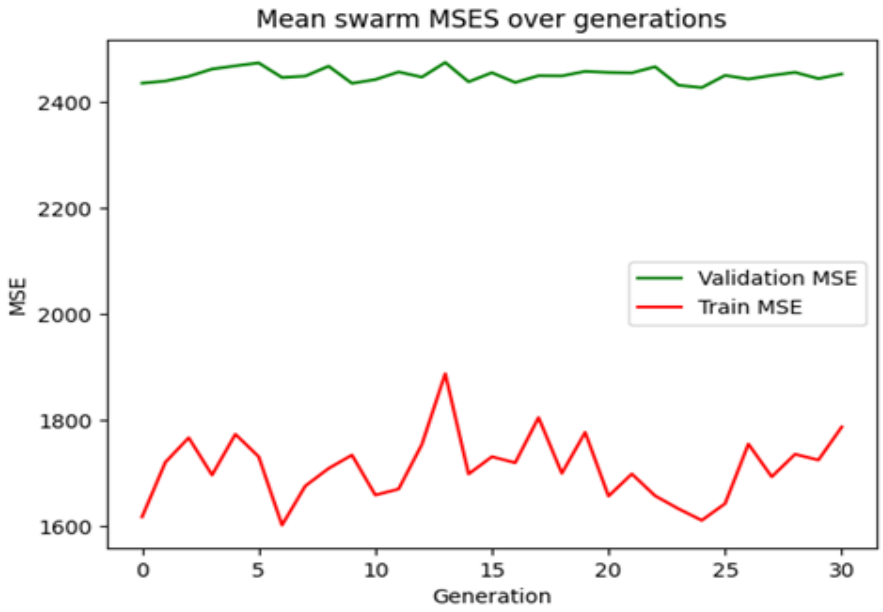


Figure 58: Illustrates the change of the mean swarm MSE over 30 generations

Figure 59 shows the change of the best training and validation MSE in the swarm across 30 generations. We can see that the best validation MSE and the best train MSE decreased, with the best validation MSE being found in generation 30.

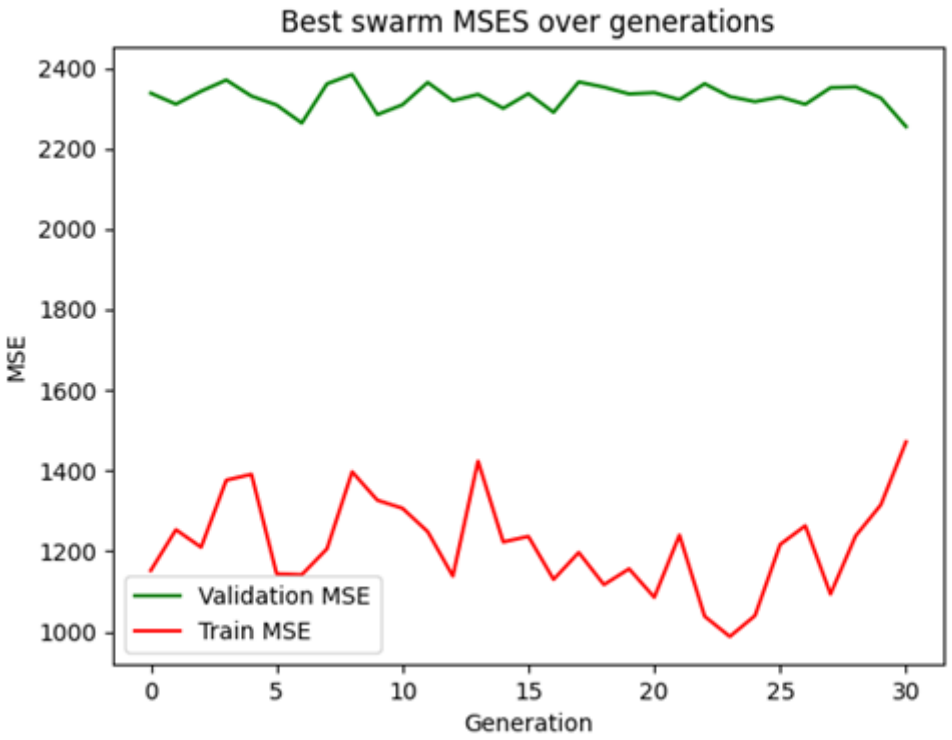


Figure 59: Illustrates the change of the best swarm MSE over 30 generations

3.3.3. Two-phased training process

Figure 60 shows the first training phase of the best architecture found, using SGD optimizer. We can see that both validation and training MSE decreased up until around 350 epochs.

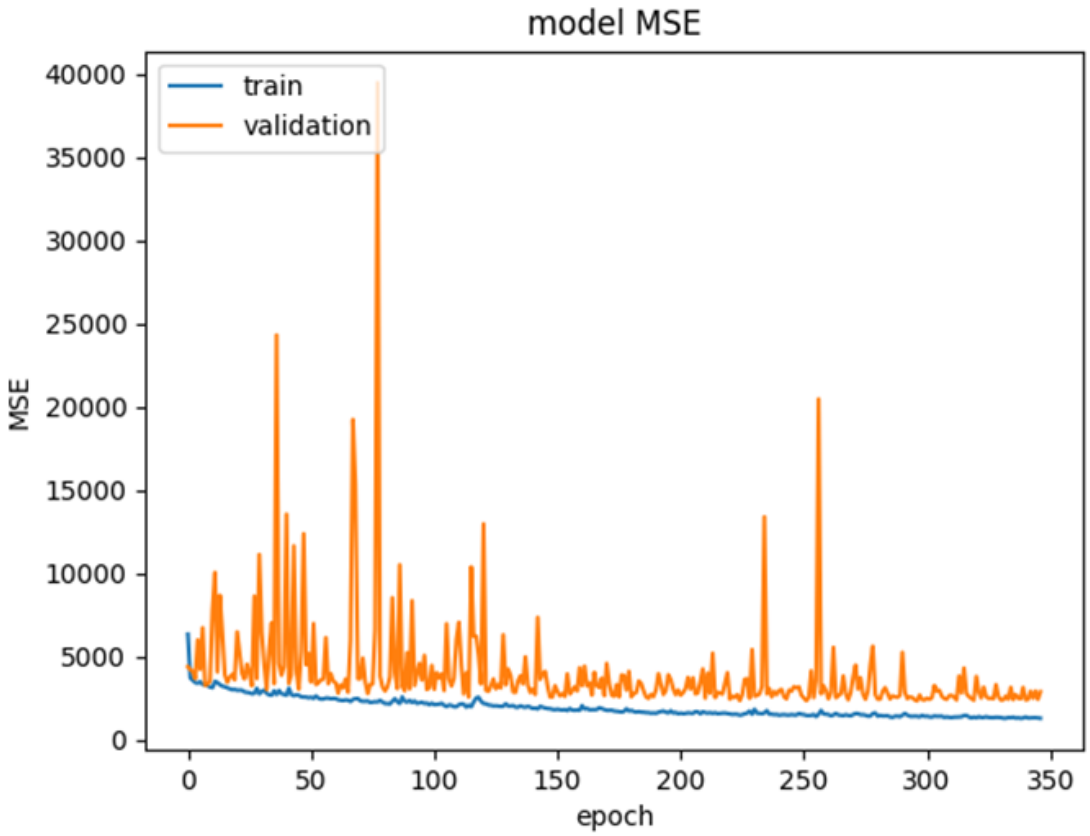


Figure 60: Illustrates the change of the best architecture MSE over epochs in the first training phase using SGD optimizer

Figure 61 shows the second training process of the best architecture found, using Adadelta optimizer. We can see that we were not able to further decrease the validation MSE in the second training phase.

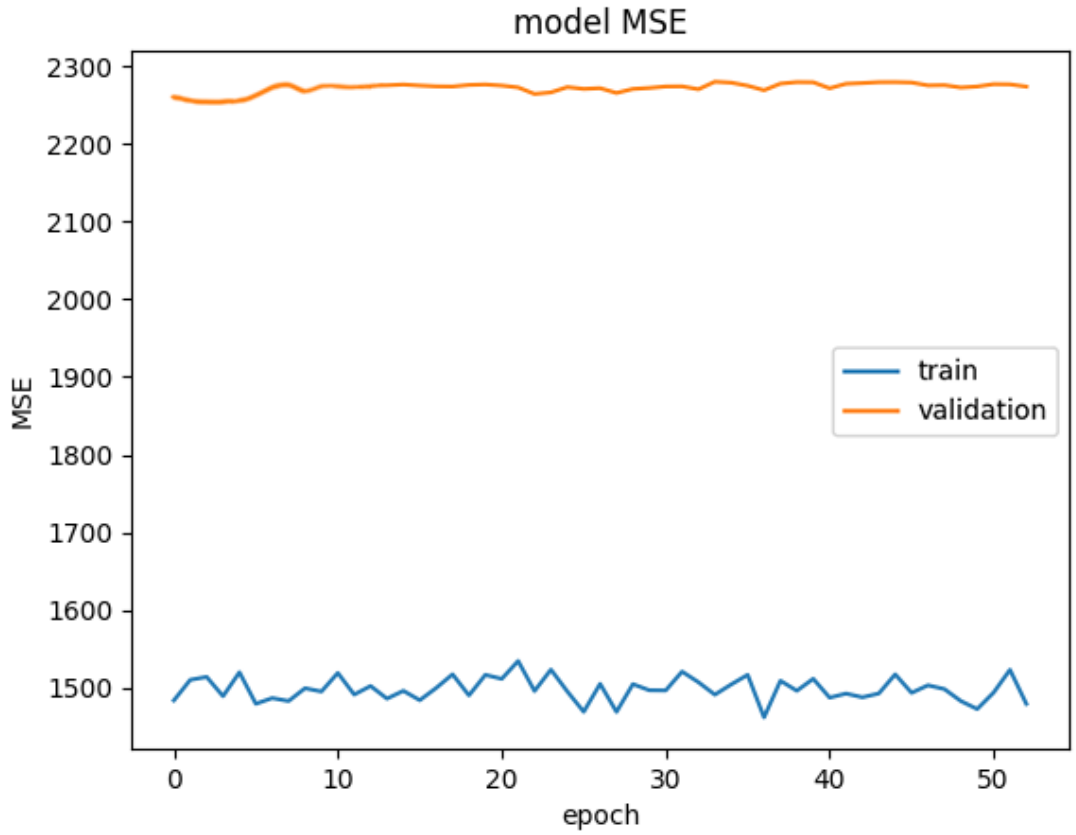


Figure 61: Illustrates the change of the best architecture MSE over epochs in the second training phase using Adadelta optimizer

3.3.4. MA search results

The MA search results are summarized in table 25.

Table 25: Summary of results from the MA search

Best validation MSE in generation 0	2337.43
Best architecture found	30-33-36-50-1
Best architecture validation MSE	2254.41
Generation in which the best architecture was found	30
Swarm MSE reduction	83.02

Figure 62 visualizes the best architecture found by MA. We can see that the layer sizes increase layer by layer.

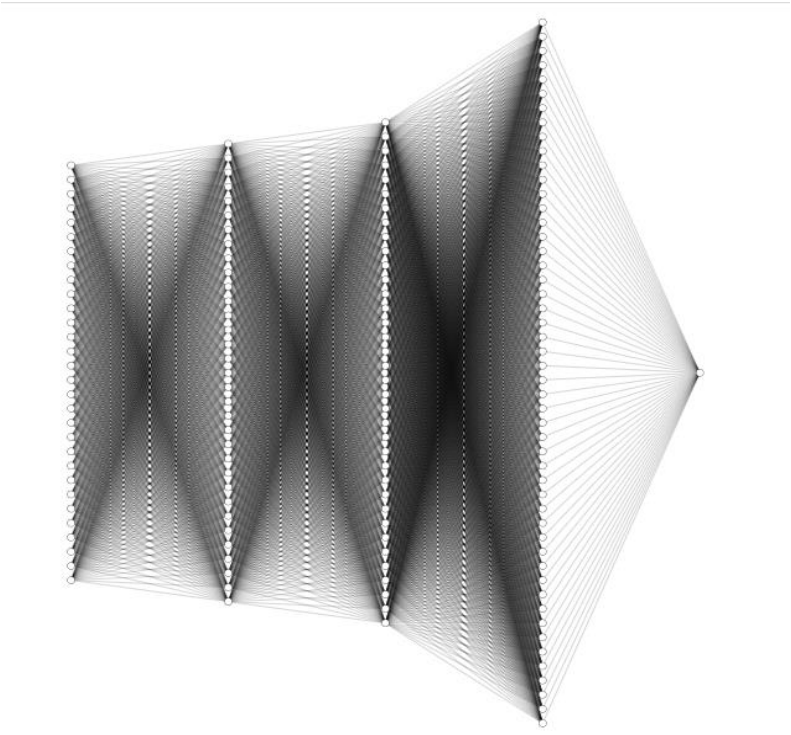


Figure 62: Visualization of the best architecture found by MA (30-33-36-50-1)

Table 26 shows how the best architecture performs on the training, validation and test set in terms of metrics. We can see that the model performs better on the training and validation sets than the test set, which is expected.

Table 26: Table showing the metrics of the best architecture in train, validation, and test sets

Metric results using MA-DNN combination											
Train				Validation				Test			
RMSE	R^2	MAE	MAPE	RMSE	R^2	MAE	MAPE	RMSE	R^2	MAE	MAPE
36.84	0.70	21.14	26.45	47.48	0.49	25.88	32.14	51.38	0.41	25.88	32.55

Statistical analysis of model testing errors

Getting a deeper understanding of the testing error is an important step to get practical knowledge about the model. Without digging into it, we only have the metrics to tell us how it really is performing. The metrics do not give us a clear enough image of which instances we have good accuracy on and which ones we do not. Table 27 shows statistical error measurements to gain a more profound understanding of the inner workings of the proposed model.

Table 27: The statistics of the testing errors defined by the actual value minus the predicted value on every instance in the test set.

Standard Deviation	51.35
Max	332.48
75^{th} percentile	10.06
Mean	1.68
25^{th} percentile	-16.46
Min	-249.56
Proportion of negative errors	0.54
Proportion of positive errors	0.46

Notice that the model usually predicts too high values, indicated by the proportion of negative errors being over 0.5, but the mean error being positive means that the magnitude of too low predictions outweighs the magnitude of too high predictions. The max error is 332 and the minimum error is -249. This means that the worst error was when a prediction was lower than the actual value.

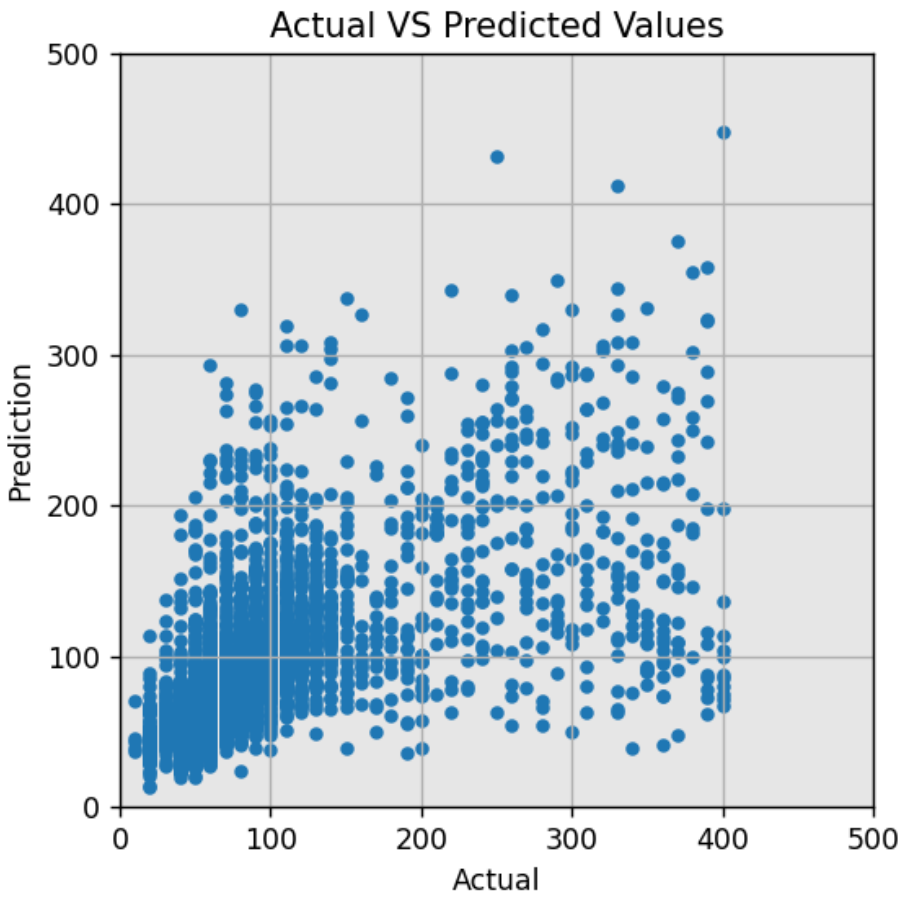


Figure 63: The predicted vs. actual energy use.

Figure 63 shows predictions as a dependent variable of actual values. A perfect model with zero error would show us points following a linear relationship with a slope of 1. When the actual values are in the lower range of approximately 10-150, the slope looks relatively good. However, values above 150 are clearly harder to predict for the model, where it frequently predicts too low values.

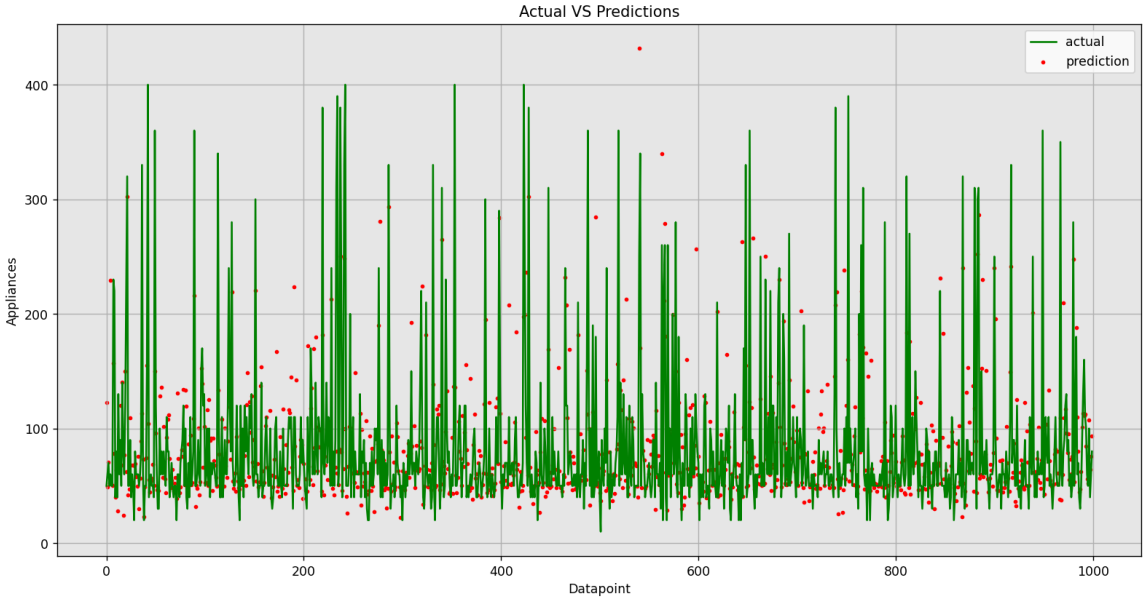


Figure 64: The predicted vs. actual energy use (only the first 1000 instances in test dataset shown here as a sample).

Figure 64 confirms that we have better accuracy when the actual values are in the lower range. When the actual values are in the higher range, the model usually predicts values much lower than the actual values. We can clearly see that when the actual values are higher than 150, the model has more difficulty making accurate predictions.

Figure 65 shows that the majority of errors are negative (too high predictions), however we have many positive errors with greater magnitude.

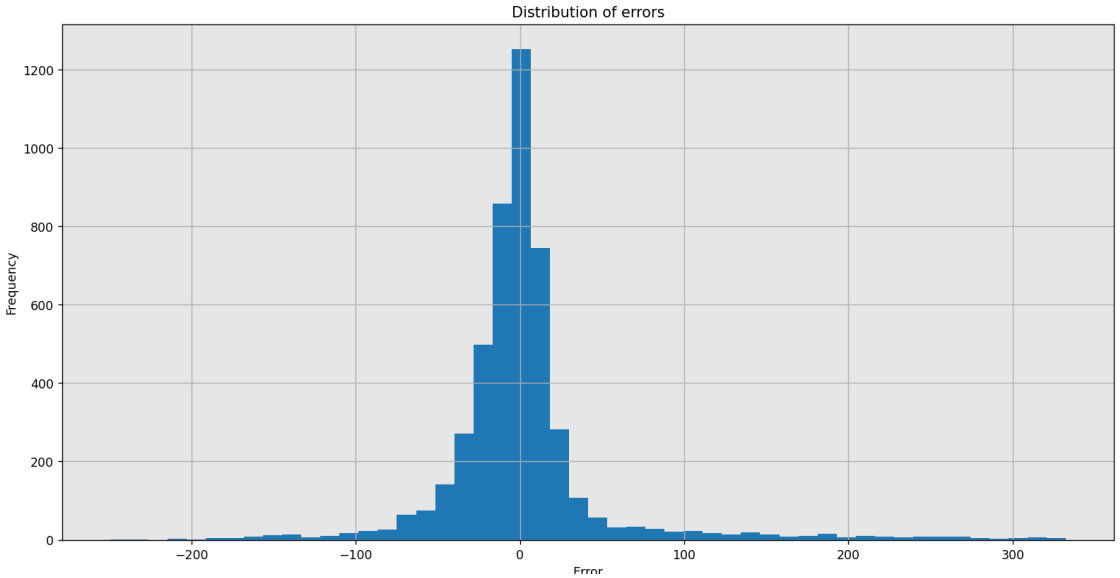


Figure 65: The distribution of the testing errors.

Table 28 shows comparison of errors from too high predictions vs errors from too low predictions (in comparison with the actual values). In this table we use the absolute value of the testing errors, even though too high predictions result in negative errors and too low predictions lead to positive errors.

Table 28: The statistics of the absolute values of the errors due to too low and too high predictions.

	Absolute errors due to too high predictions	Absolute errors due to too low predictions
Standard Deviation	29.24	55.35
Max	249.56	332.48
75^{th} percentile	29.56	25.84
Mean	23.89	31.38
25^{th} percentile	6.34	4.95
Min	0.05	0.004

Notice that 75% of the too high predictions are between 0.05 to 29.56 different from the actual value, while 75% of too low predictions are between 0.004 and 25.84 different from the actual value.

To sum it all up, the test error analysis shows that the model more frequently predicts values higher than the actual value, however this is outweighed by the magnitude of the error made when predicting values lower than the actual value. When the actual value is in the higher range, the model has difficulty predicting it accurately.

3.3.5. Summary of MA-DNN experiment

In this work, we present a new approach to find the optimal structure of a deep neural network for predicting the energy consumption of a low-energy house. The MA algorithm is used to determine the optimal architecture of the neural network.

The results showed that our proposed method reduced MSE by 83.02 from the best of the initial architectures to the best architecture found by MA algorithm (with a final MSE of 2254.41), after 30 generations.

Testing error analysis shows that 54% of predictions are higher than the actual value (negative errors), and 46% are lower than the actual value (positive errors). Though most errors are negative, the mean error is 1.68 because the magnitude of the positive errors outweigh the magnitude of the negative errors.

4. Discussion

In this chapter, we'll look at the testing part of our project and discuss our results in comparison to existing work, with regards to model performance, and computational complexity. In addition to this we will discuss test error analysis and feature importance.

4.1. Comparison with existing work – Accuracy

Table 29 gives an overview of the comparison between the results from the different algorithms. The first four algorithms were used by Candanedo et. al. in their research, and the last three are our algorithms.

Table 29: The results of the algorithms measured by different metrics

Model	**Training**				**Testing**			
	RMSE	**R^2**	**MAE**	**MAPE**	**RMSE**	**R^2**	**MAE**	**MAPE**
LM	93.21	0.18	53.13	61.32	93.18	0.16	51.97	59.93
SVM Radial	39.35	0.85	15.08	15.60	70.74	0.52	31.36	**29.76**
GBM	17.56	0.97	11.97	16.27	66.65	**0.57**	35.22	38.29
RF	29.61	0.92	13.75	13.43	68.48	0.54	31.85	31.39
PSO-DNN	39.02	0.67	21.50	25.74	52.96	0.38	26.64	30.23
DE-DNN	38.49	0.68	21.61	26.80	52.79	0.38	27.63	32.66
MA-DNN	36.84	0.70	21.14	26.45	**51.38**	0.41	**25.88**	32.55

We can see that the algorithms we implemented gave better RMSE and MAE values in the testing set, where the MA-DNN combination performed best. When we look at the R^2 values in the testing set, our algorithms did only perform better than the LM model. Candanedo et. al. claimed that GBM was their best model, and we can see that our combination of algorithms performed better than it in three of four metrics, with R^2 being the only metric where their algorithms gave better results.

Test error analysis shows that the models' predictions are more accurate on datapoints where the actual appliances energy consumption value is in the range of 10-150 Wh. When it comes to higher values, the models generally predicted too low. Looking at Figure 63 for example, we can see that we have different predictions for datapoints with the same actual appliances Wh value. We can also see that we have different actual values for the same predictions. For a neural network to predict two different numbers on two different datapoints, the values of the features cDNNot be the same on those two datapoints. This means that there is room for improvement for the dataset features to predict energy consumption of the appliances more accurately.

4.2. Comparison with existing work – Computational Complexity

LM, SVM, GBM and RF are relatively simple machine learning algorithms which do not require a lot of computational power or time. Our method of optimizing DNN architectures using evolutionary algorithms and swarm intelligence algorithms is computationally expensive and can easily crash even on a laptop. To implement our method, we had to rent a powerful server, and our method still required between 6 to14 hours of time to complete. This can be explained by the fact that training DNN is a relatively complex and time-consuming task to begin with. Since we used a swarm size of 20 with 2 trials per DNN run on 30 generations, it resulted in training a total of 20*2*30 = 1200 DNN to be trained.

4.3. Feature importance

As previously mentioned, Figure 26 is the combination of the results from the different algorithms we utilized for feature importance. This is to visualize all the prior results to make it easier for us to see which features were more important than others.

In the same figure we can clearly see “hour” dominating, which makes sense considering that the change in energy usage will be most clear in an hourly timeframe. This is further confirmed by the correlation matrix in Figure 17 where we can see “hour” is the most correlated with the appliances and energy consumption

The second most important feature is “T3” which from Table 10 we can see is the temperature in the laundry room area. This feature being placed so high is logical as the laundry room will have multiple appliances. When these appliances are in use, they will produce heat as a byproduct which will in turn increase the ambient temperature. Logic dictates that an increase in temperature in this area will have come as a result of appliances using more energy.

The third and fourth most important features “RH_1” and “RH_2” are the humidity in the kitchen area and the humidity in the living room area respectively. The living room and the kitchen, where these sensors were placed, are not separated by a wall. You could probably even say it’s the same room, which is the reason we chose not to write about these separately. If one is affected the other will be too, as the humidity in a room does not change much from one end to the other.

Both being placed high in our results makes sense as the humidity in the room will be affected by the fridge constantly being opened and taking in the moisture from the air (Lee, 2020) and subsequently using energy too cool down again. In addition, when food is being cooked on the stove or in the oven it will create steam which will increase the humidity in the air. when these appliances are using energy, they will in turn affect the humidity, therefore both “RH_1” and “RH_2” are placed so high. “RH_1” will be ranked a bit higher as it’s the sensor placed in the kitchen and will notice the change in humidity quicker than “RH_2”.

From these observations we can reasonably conclude that the “Combined average feature importance” is accurate.

4.4. Reliability of results

Our work is based on neural networks, evolutionary algorithms, and swarm intelligence, which make use of randomness. Such algorithms can sometimes have a “bad run” because of bad luck, which can lead to us not seeing the full potential of an algorithm.

Hyperparameter tuning is important when working with AI algorithms. One can never know which hyperparameters are best for a specific problem before testing. In our case we used suggested parameter values from the book Practical Swarm Intelligence and (Kneusel, 2021) were not able to run multiple trials with different hyperparameter settings because our implementations were very computationally expensive and took up to 20 hours. This can lead to us not squeezing out the full potential of our implementations.

The dataset was collected using a wireless sensor network. Sensors are not always accurate and can wear out over time, which can lead to inaccurate data. This can have an impact on the results we achieve when feeding the data into an AI.

5. Conclusions and Future work

In this thesis, we presented our work on a Machine Learning-based solution to reduce energy consumption of a house. We started by choosing few algorithms that can help us achieve better results and combine them with deep neural networks to optimize the structure of an Deep Neural Network (DNN), getting an even better result. We have combined an Evolutionary algorithm known as Differential Evolution (DE) with Deep Neural Networks (DNNs), an optimization algorithm known as Particle Swarm Optimization (PSO) with DNNs, and lastly Memetic algorithm (MA) with DNNs.
The combination of these algorithms gave, as expected, better results than the other stand-alone algorithms that we’ve compared with, but the results were not as easy to achieve.

While working with these combinations of algorithms, we met some challenges that needed to be worked around. A major challenge was computational complexity, meaning that our normal, and somewhat powerful laptops with 16GB RAM and an adequate GPU, cannot run optimal computations to reach optimal results without crashing.

This challenge opens an opportunity for further work on our already established solutions. When working on DNN-architectures, a certain number of trails can be given to each architecture to evaluate its potential. For our proposed solution, we chose two trials per network architecture. The reason for choosing only two trials per network is lack of immense computational power. Given that in the future, if we can increase the computational power of our devices or even just opt for stronger supercomputers, we are able to increase the number of trials per network architecture proportionally and thus get a more accurate and better results than currently presented.

6. References

Aggarwal, S. (2020, May 13th). *Multiple Linear Regression*. Towards Data Science. https://towardsdatascience.com/multiple-linear-regression-8cf3bee21d8b

Ahmad, M.F., Isa, N.A., Lim, W.H. & Ang, K.M. (2022). Differential evolution: A recent review based on state-of-the-art works. *Alexandria Engineering Journal*, *61* (5), 3831-3872. https://doi.org/10.1016/j.aej.2021.09.013

Bangyal, W.H., Nisar, K., Ag. Ibrahim, A.A.B., Haque, M.R., Rodrigues, J.J.P.C. & Rawat, D.B. Comparative Analysis of Low Discrepancy Sequence-Based Initialization Approaches Using Population-Based Algorithms for Solving the Global Optimization Problems. *Applied Science 2021, 11, 7591*. https://doi.org/10.3390/app11167591

Bobbitt, Z. (2021*). RMSE vs. R-Squared: Which Metric Should You Use?* Statology.org https://www.statology.org/rmse-vs-r-squared/

Brownlee, J. (2016, September 9th). *A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning*. Towards Data Science. https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/

Cambridge University Press. (2022, May 24th). Nomenclature. *Cambridge Dictionary*. https://dictionary.cambridge.org/dictionary/english/nomenclature

Candanedo, L. M., Feldheim, V. & Deramix, D. (2017). Data driven prediction models of energy use of appliances in a low-energy house. *Energy and Buildings*, *140*, 81–97. https://doi.org/10.1016/j.enbuild.2017.01.083

Crypto1. (2020, October 2nd). *How Does the Gradient Descent Algorithm Work in Machine Learning?* Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/10/how-does-the-gradient-descent-algorithm-work-in-machine-learning/

Dhirf, H., Giraldo L.G.S., Kubat, M. & Wuchty, S. (2019). *A Stable Combinatorial Particle Swarm Optimization for Scalable Feature Selection in Gene Expression Data* [screenshot]. Researchgate. https://www.researchgate.net/publication/330673022

Dobilas, S. (2021, Jan 17th). *SVM Classifier and RBF Kernel — How to Make Better Models in Python*. Towards Data Science. https://towardsdatascience.com/svm-classifier-and-rbf-kernel-how-to-make-better-models-in-python-73bb4914af5b

Engelbrecht, A.P. (2007). *Computational Intelligence. An Introduction (2. edition).* Wiley. p.31-32 https://web2.qatar.cmu.edu/~gdicaro/15382-Spring18/hw/hw3-files/pso-book-extract.pdf

Fouad, M.M., El-Desouky, A.I., Al-Hajj, R. & El-Kenawy, E.M. (2020). Dynamic Group-Based Cooperative Optimization Algorithm. *IEEE Access*, *8*, 148378-148403. doi:10.1109/ACCESS.2020.3015892

Huang, W., Xu, T., Li, K., He, J. (2019). Multiobjective differential evolution enhanced with principle component analysis for constrained optimization. *Swarm and Evolutionary Computation, 50*(100571), 1-14. https://doi.org/10.1016/j.swevo.2019.100571

Kie Codes. (2020, July 13th). *Genetic Algorithms Explained By Example* [Video]. YouTube. https://www.youtube.com/watch?v=uQj5UNhCPuo

Kneusel R. T. (2021). *Practical Swarm Intelligence in Python. Using Swarm Intelligence and Evolutionary Algorithms to Solve Problems in Engineering and Science*. Independently published.

Kneusel R.T. (2018). *Random Numbers and Computers*. Springer International Publishing AG. https://doi.org/10.1007/978-3-319-77697-2

Kumar, S. (2021). *Wireless Sensor Network (WSN)* [screenshot]. Geeksforgeeks.org https://www.geeksforgeeks.org/wireless-sensor-network-wsn/

Leather, A. (2021, March 16th). *Intel Announces 11th Gen Core i9-11900K, i7-11700K And i5-11600K Desktop CPU Details And Pricing* [screenshot]. Forbes. https://www.forbes.com/sites/antonyleather/2021/03/16/intel-announces-11th-gen-core-i9-11900k-i7-11700k-and-i5-11600k-desktop-cpu-details-and-pricing/?sh=22fe5c8f729f

Lee, S. (2020, March 26th). *What Can Cause Too Much Moisture on the Interior of a Refrigerator*. Hunker. https://www.hunker.com/13409822/what-can-cause-too-much-moisture-on-the-interior-of-a-refrigerator

Liu, J. (2020). Survey of the Image Recognition Based on Deep Learning Network for Autonomous Driving Car, *2020 5th International Conference on Information Science, Computer Technology and Transportation (ISCTT)*, 1-6. doi: 10.1109/ISCTT51595.2020.00007

Macri, N.A. (1982). CHAPTER 8 – STATISTICS. *Study Guide for Applied Finite Mathematics (Third Edition)*. (p.139-158). Academic Press. https://doi.org/10.1016/B978-0-12-059570-9.50010-8

Mallawaarachchi, V. (2017, July 8th). *Introduction to Genetic Algorithms – Including Example Code* [screenshot]. Towards Data Science. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

Masashi, S. (2016). Chapter 3 - Examples of Discrete Probability Distributions. *Introduction to Statistical Machine Learning.* p.25-36. https://doi.org/10.1016/B978-0-12-802121-7.00014-5

Pascual, C. (2018). *Tutorial: Understanding Regression Error Metrics in Python*. Dataquest.io https://www.dataquest.io/blog/understanding-regression-error-metrics/

Plagianakos, V. P., Tasoulis, D. K., Vrahatis, M. N., (2008). A Review of Major Application Areas of Differential Evolution. *Advances in Differential Evolution*. *Studies in Computational Intelligence*, 143. Springer, Berlin, Heidelberg, 197-238. https://doi.org/10.1007/978-3-540-68830-3_8

Ramachandran, P., Zoph, B. & Le, Q. V. (2017). Searching for Activation Functions. https://doi.org/10.48550/arXiv.1710.05941

Sacco, W.F., Henderson, N., Rios-Coelho, A.C., Ali, M.M. & Pereira, C.M.N.A (2009). Differential evolution algorithms applied to nuclear reactor core design. *Annals of Nuclear Energy*, *36* (8), 1093-1099. https://doi.org/10.1016/j.anucene.2009.05.007

Schneider, S. (2019, November 15th). *Mind design: could you merge with artificial intelligence?* [screenshot] (Picture on main page). Sciencefocus. Copyright by Immediate Media Company Ltd. 2022 https://www.sciencefocus.com/future-technology/mind-design-could-you-merge-with-artificial-intelligence/

Shilov, A. (2019, January 24th). *Colorful Offers GeForce RTX 2080 Ti Without A Cooler* [screenshot]. Anandtech. https://www.anandtech.com/show/13898/colorful-offers-geforce-rtx-2080-ti-without-cooler

Storn, R. & Price, K. (1997). Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimiztion 11*, 341-359. https://doi.org/10.1023/A:1008202821328

Svozil, D., Kvasnicka, V. & Prospichal, J. (1997). Introduction to multi-layer feed-forward neraul networks. *Chemometrics and Intelligent Laboratory Systems. 39*(1). 43-62. https://doi.org/10.1016/S0169-7439(97)00061-0

Tehrani, K. F., Zhang, Y., Shen, P. & Kner, P. (2017). Adaptive optics stochastic optimal reconstruction microscopy (AO-STORM) by particle swarm optimization. *Biomedical Optics Express*, *8*(11). https://doi.org/10.1364/BOE.8.005087

Teja, R. (2019). *Basics of Wireless Sensor Networks (WSN) | Classification, Topologies, Applications*. Electronicshub.org. https://www.electronicshub.org/wireless-sensor-networks-wsn/

Wang, X. & Qiu, X. (2013). Application of particle swarm optimization for enhanced cyclic steam stimulation in a offshore heavy oil reservoir. *International Journal of Information Technology, Modeling and Computing (IJITMC), 1(2),* 37-47. doi:10.5121/ijitmc.2013.1204

Whigham, P.A. & Dick, G. (2010). Implicitly Controlling Bloat in Genetic Programming. *IEEE Transactions on Evolutionary Computation*, *14*(1), 173-190. doi:10.1109/TEVC.2009.2027314

Yiu, T. (2019, Jun 12th). *Understanding Random Forest*. Towards Data Science. https://towardsdatascience.com/understanding-random-forest-58381e0602d2

Appendices

A.1 Glossary

Table 30: Glossary definitions

Term	**Definition**
Bernoulli experiment	Bernoulli experiments are experiments that have only two possible outcomes. The probability for either outcome is the same and does not change the success or failure rate. (Masashi, 2016, p.26) As an example, a coin flipped has the same probability of landing on heads or tails.
Nomenclature	A nomenclature is a system of terms and names that is used by a particular area of science (Cambridge University Press)
Bloat	Bloat in algorithms is the production of a neural network that increases in size but does not consequently improve its fitness and doesn't find a solution. (Whigham & Dick, 2010, p.173)
Search space	The area/space that the particles in our algorithm are moving through. It is essentially the space we are searching through. Our goal is to try and reach the lowest point in the search space. Reaching the lowest point would mean we have found the lowest global function value otherwise known as the optimal solution.

Bounds	Bounds are the limits in our search space that the particles must stay inside of and can't move beyond. If they try to leave the boundaries, they will be forced back inside
Local minimum and global minimum	Local minimum and global minimum are both terms used frequently in our thesis. These are the lowest points in our search space and what the particles are trying to find. We are always trying to reach the global minimum as it is the lowest possible point in our search space. This would be the optimal solution for a problem, and where we get the lowest function value. local minimum points are lower than the average height in our search space but reaching them will still not give us the optimal solution. In some cases, this is fine as we can be satisfied with the results of the local minimum point. In other cases, they may hinder us from finding the global minimum as the particles don't know if there are any lower points and might settle down here.
Metaheuristic	Problem independent technique that can be applied to a broad range of problems. For GA, PSO and DE
Deep Neural Network (DNN)	A Deep neural network is a collection off nodes and networks. DNNs are inspired by biological neural networks but are much simpler. There are a few things you need to create an DNN, you need nodes: input nodes, hidden nodes and output nodes. You need connections between these nodes, and you need weights and biases. (Liu, 2020, p.44)

	A DNN starts off with values for the input nodes. All nodes will get their values from their connections, except for input nodes. Each node will send a “signal” through the connection to a node they are preceding. This “signal” which is just a number will be multiplied by a weight and all other connections leading to the succeeding node will be added up to create a new value. This is the value the succeeding node will have. Then the process is repeated until we get a value for the output nodes. To get a desired DNN you must find the right architecture, this is the structure of the DNN, number of nodes, number of layers, connections. You must also find the optimal weight and bias values; these can be changed any time and will be essential to getting a desired outcome The difference between an artificial neural network (ANN) and a deep neural network is that an ANN can be classified as a DNN if it has more than one hidden layer
Dimensions	Dimensions is a term that will be used repeatedly throughout this thesis as it is deeply rooted in AI and neural networks. When we talk about dimension, we are talking about it in a mathematical sense, therefore we might operate with more than 3 dimensions. In those cases, it will be impossible to visualize the space that the algorithm is working inside of, considering that we humans can only perceive 3 dimensions. If we were to work with particles that move in a 2/3-dimensional space, looking for the optimal solution, we

	would be able to visualize it using plots. We could see where the particles are, how they are moving and if it was obvious, where the optimal solution lies.

A.2 Acronyms

Table 31: Table of acronyms

Acronym	**Definition**
AI	Artificial Intelligence
ANN	Artificial Neural Network
DE	Differential Evolution
DNN	Deep Neural Network
SIA	Swarm Intelligence Algorithm
EA	Evolutionary Algorithm
PSO	Particle Swarm Optimization
MA	Memetic Algorithm
GA	Genetic Algorithm
NEAT	Neuroevolution of Augmenting Topologies
RF	Random Forest
DTR	Decision Tree Regressor
GBM	Gradient Boosting Machine
SVM	Support Vector Machines
ML	Machine Learning
SGD	Stochastic Gradient Descent

A.3 List of symbols

Table 32: Table of symbols

Symbol	Description
α	Learning rate
$\hat{x}_i$	The best position in the search space that particle i has found on its own
x_i	The position of particle i
$\hat{g}_i$	The best position the particle i knows about
g_i	The position of the neighborhood of particle i
p_b	Interaction probability
p	Probability vector
U	D-dimensional random vectors
N	Random value
$\bar{x}$	The mean value between current component of the neighborhood best position and the particle best position
$\hat{g}_{ij}$	The best position the j-th position component of particle i knows about
$\hat{x}_{ij}$	The best position in the search space that the j-th position component of particle i has found on its own
x_{ij}	The position of the j-th position component of particle i
c_1	Cognitive parameter
c_2	Social parameter
w	Base inertia parameter
x_i	Initial particle position
v_i	Initial particle velocity
σ	Sigmoid function
v	Mutation vector
$v_1, v_2\ and\ v_3$	Donor swarm vectors
CR	Crossover probability
F	The amplification factor
u	Candidate vector

A.4 Taxonomy

In the world of biology, taxonomy is a way of classifying living organisms. This classification has a certain criterion for each organism to be part of a specific class. Groups of some living organisms are called kingdoms like bees, while some are called colonies like ants and so on. AI (Artificial Intelligence) is an enormous field with a lot of intricacies. A group of AI-algorithms, which is the group we are working on, is called Global optimization algorithms. Global optimization algorithms are grouped into 5 categories. Table 33 shows a list of the five categories and their algorithms.

Table 33: Algorithms split into categories

Taxonomy category	**Algorithm**	**Description**
Mountaineer	PSO	Particles reach for the highest peak - emphasis on exploitation rather than exploration. This is how a PSO would work if c1(cognitive component) > 0 and c2 = 0.
Sightseer	-	Searching the space looking for an interesting position to exploit by maintaining data gain from the history of functional evaluations.
Team	PSO, DE, MA	A group for ideal population algorithms. Agents (particles) work to both explore and exploit positions in balanced way. This is how an ideal optimization algorithm is supposed to work.
Surveyor	-	Builds an approximate map of the search space to select new regions for exploring.
chimera	DNN combined with either PSO, DE, or MA	Hybrid of two algorithms built from existing algorithms. This is the structural goal of our algorithm where we combine DNN with optimization algorithms.

A.5 WSN (Wireless Sensor Networks)

WSN is distributed networks with dedicated sensors that measure and record the physical conditions of a given space. ZigBee is WSN-technology which is based on IEEE 802.15.4, a wireless technology for low cost, low power, and low data transfer networks and devices.
A typical WSN is divided into 2 components: *Sensor node* and *Network architecture*.
Sensor node has 4 Components:

- Power-supply.
- Sensor (which measures the analog physical data like temperature and converts it to digital by using ADC.)
- Processing unit (for intelligent data processing and data manipulation as needed.)
- Communication system (often short-range radio for communication and data transmission and receiving.)

The network-part of this makes sure that all the sensory data collected, its intelligent processing and manipulation, and transmission is all connected and being done efficiently. When a network is delivering data, it sends it to a single station called “Base Station” as shown in (Kumar, 2021). This Base Station makes it possible to connect a WSN to another WSN or even the internet as shown in the Figure 66.

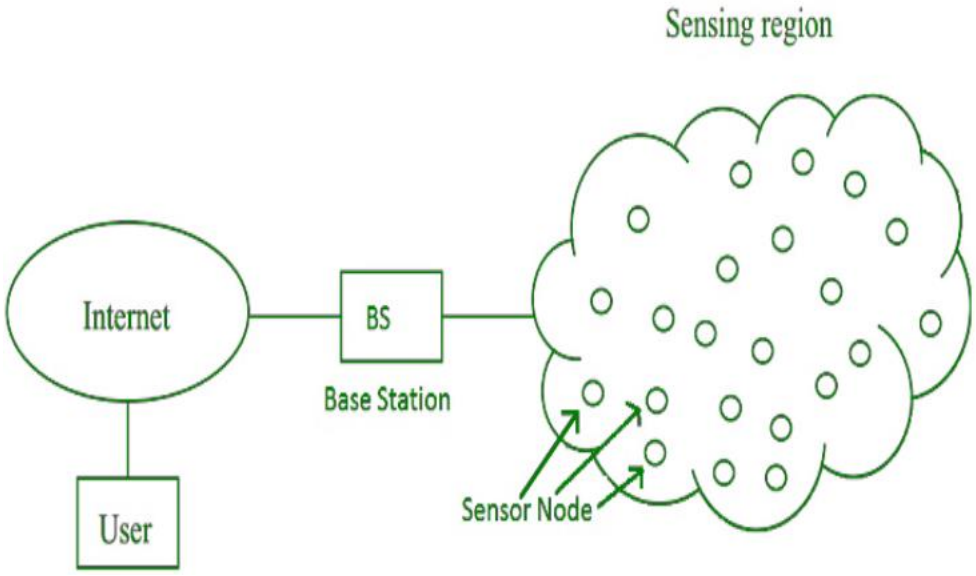


Figure 66: Shows the whole connection of a typical WSN (Kumar, S., 2021). Copyright by GeeksForGeeks.

Base Station is an integral part of the network architecture. In a Flat architecture, the base emits a signal to all the sensor nodes but only the sensors with equivalent queries will answer and activate the actuators. While in a Hierarchical architecture, the base station thinks of all the sensors as being a member of different clusters of sensor nodes. Every cluster has a Cluster head, which then relays information to the rest of the cluster members (Teja, 2019).

A.6 Information of the metrics used in the results

When the performance of the results of certain algorithm is being measured, we take advantage of specific metrics that allow us to evaluate the results in different ways. Candanedo et. al. used 4 different metrics to evaluate the results achieved by their algorithms. The metrics are RMSE, R^2, MAE, and MAPE.

Root Mean Squared Error (RMSE) describes how a regression model can predict the value of the response variable in whole numbers. It does this by telling us about the distance between the predicted value made by a regression model and the actual value. The lower the RMSE, the better a model and a dataset fit each other. Equation 13 shows the formula for RMSE.

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

Equation 13: Formula for RMSE.

In Equation 13, $\hat{y}_i$ represents the predicted values, y_i represents the observed values and n is the number of observations.

R^2, also called the "coefficient of determination", is a value between 0 and 1 that measures how well the regression line fits the datapoints. Though both RMSE and R^2 represent how good a dataset and a regression model fit together and they both predict the value of the response variable, R^2 does the prediction in percentage terms. It is also usual to present this percentage in a decimal-form. The closer the R^2-value is to 1 or 100%, the better a model and a dataset fit each other. It is useful to calculate both R^2 and RMSE, since they both describe the fit of a model to a dataset in percentage and absolute as we can observe from almost all the research papers that evaluate the fitness of a dataset. The formula for R^2 is shown in Equation 14, where $\hat{y}_i$ represents the prediction or a given point on the regression line, $\bar{y}_i$ represents the mean of all the values and y_i represents the actual values.

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{\bar{y}}_i)^2}$$

Equation 14: Formula for R^2

An issue that effects RMSE calculations are the outliers. Outliers are datapoint that overshoot or undershoot too much according to the other datapoints. It means that if we, for example, have 5 datapoints for the average temperatures of 5 days in the spring in Oslo, which have the values 4,6,2,8, and 98, we can observe that 98 is clearly an outlier since 98 is just an impossible temperature for our data. So, if we have a data that has an outlier, we must downplay it by using feature engineering, since the error contributed by this outlier can throw off our predictions.

Mean Absolute Error (MAE) takes the absolute value of each datapoint, finds the difference (residual) between the datapoint and the regression line (our model prediction), and lastly

takes the average of all these residuals as outlined in Equation 15 and Figure 67. Since MAE is calculating with absolute values, outliers are not seen as a big issue because both MAE and MAPE are robust to outlier. The lower the MAE, the better the model predicts.

$$MAE = \frac{1}{n}\sum|y - \hat{y}|$$

Equation 15: Formula for MAE

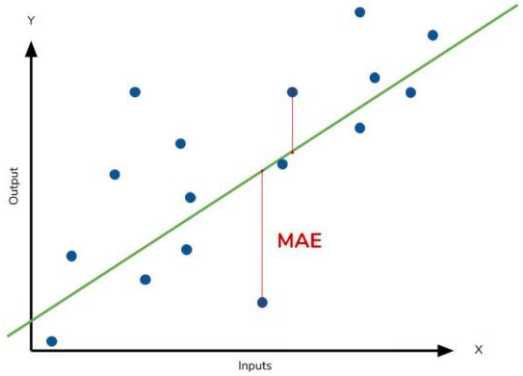


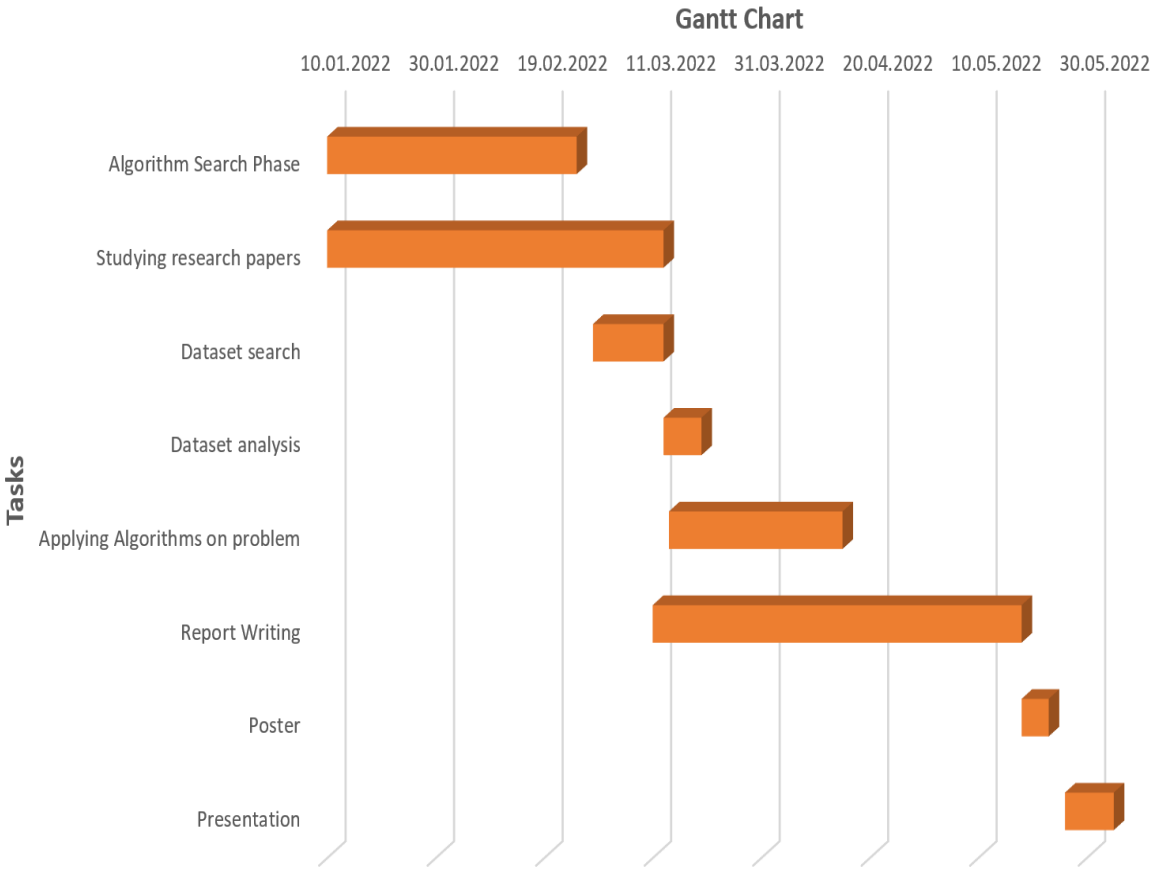
Figure 67: Shows MAE between a datapoint and the regression line (Pascual, C., 2018). Copyright 2022 by Dataquest.

Mean Absolute Percentage Error (MAPE) is the percentage version of MAE. And the equation for MAPE is an updated equation of the MAE with adjustments to make everything calculating a percentage-value as seen from Equation 16. MAPE can also be represented as a decimal number.

$$MAPE = \frac{100\%}{n}\sum\left|\frac{y - \hat{y}}{y}\right|$$

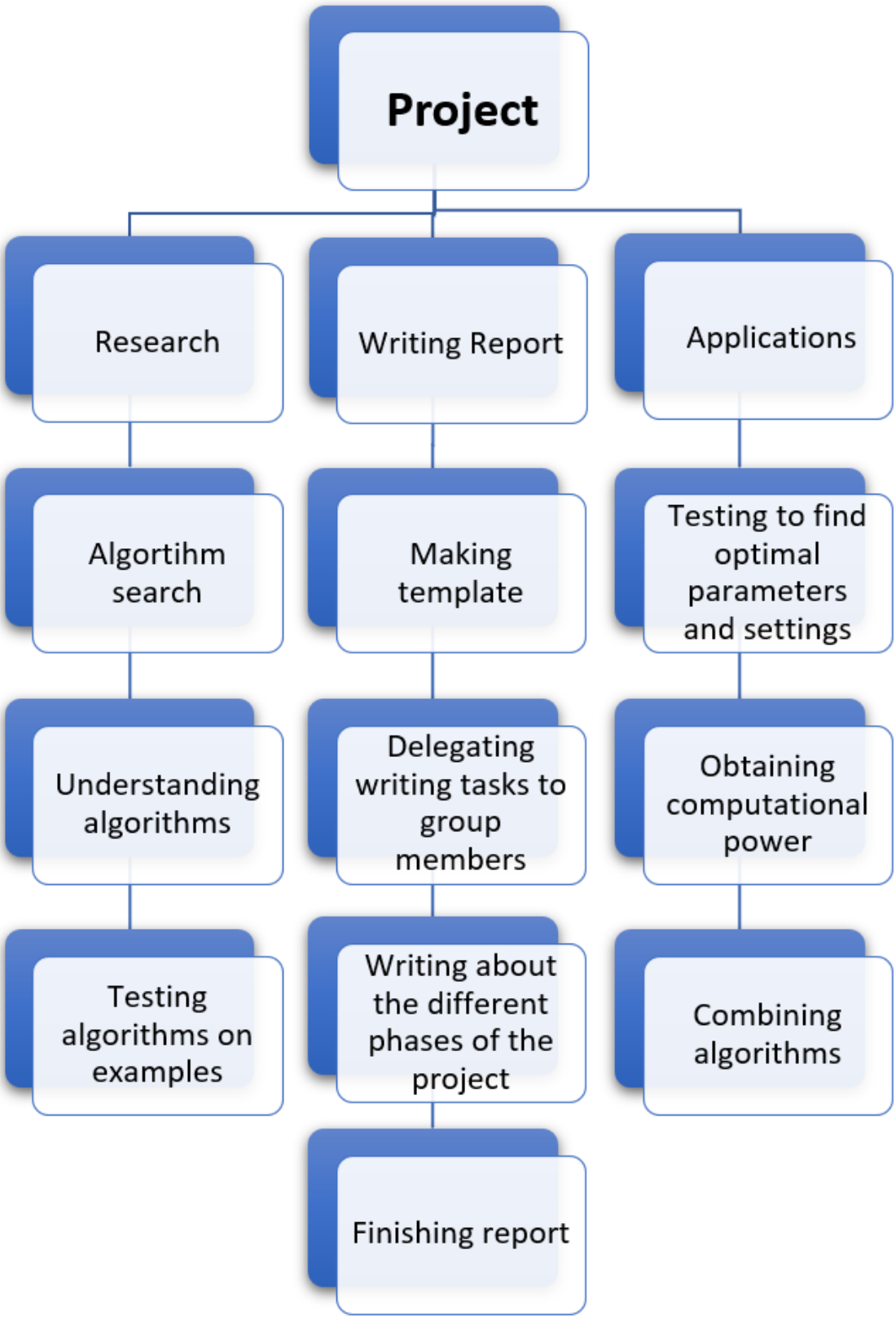
Equation 16: Formula for MAPE

A.7 Gantt-Chart



Tasks	**Start date**	**Duration (Days)**	**End date**	**Responsible**
Algorithm search & development phase	10.01.2022	46	25.02.2022	All
Studying research papers	10.01.2022	62	13.03.2022	Ahmed, Mohamed, Zishan
Dataset search	28.02.2022	13	13.03.2022	All
Dataset analysis	13.03.2022	7	20.03.2022	Alexander, Mikael
Applying algorithms on problem	14.03.2022	32	15.04.2022	Alexander, Mikael
Report Writing	11.03.2022	68	18.05.2022	All
Poster	18.05.2022	5	23.05.2022	**All**
Presentation	26.05.2022	11	06.06.2022	All

A.8 WBS – Work Breakdown Structure

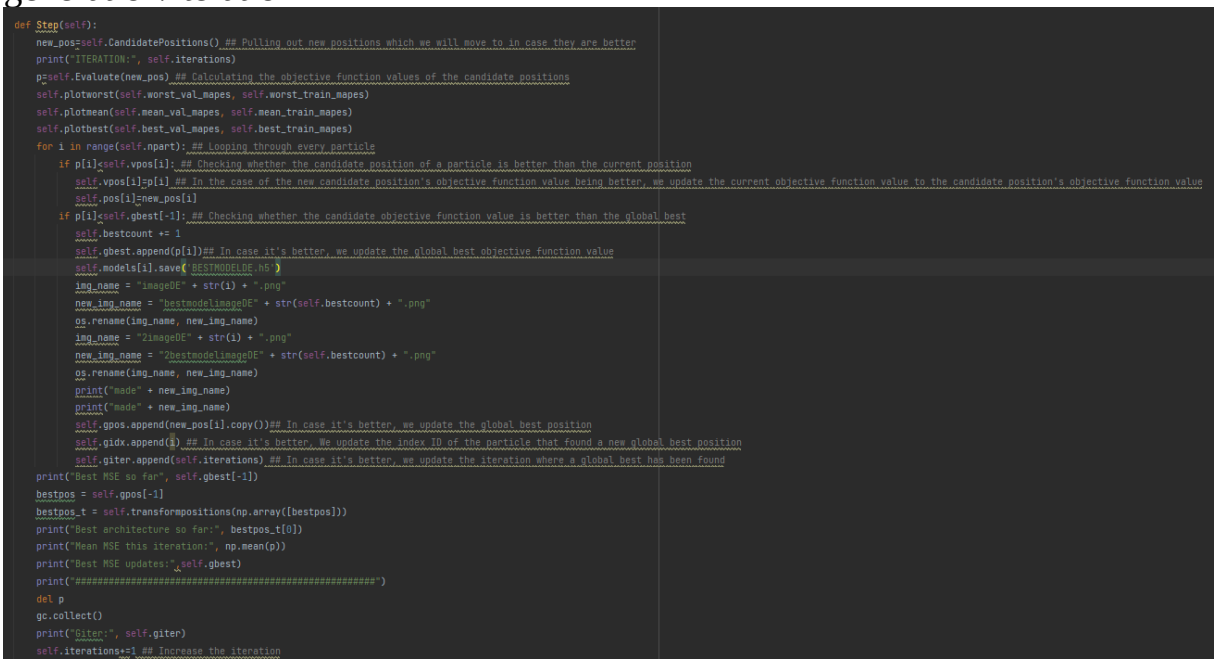


A.9 Most important parts of our code

A.9.1 Step function from PSO, summarizing what happens in a generation/iteration in PSO

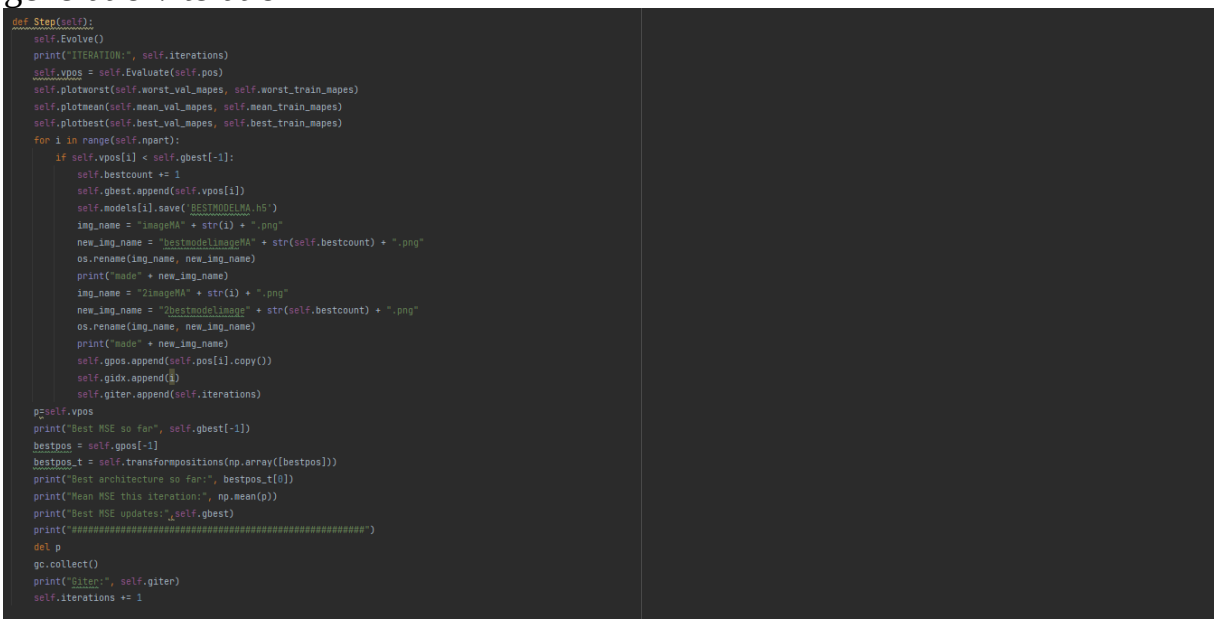
```
def Step(self):
    ## The step function represents a single swarm iteration, we start by calculating W for the iteration
    if (self.inertia!=None):
        w=self.inertia.CalculateW(self.w,self.iterations,self.max_iter)
        ## If we have an inertia to schedule the W value over the iterations, it's calculated every iteration
    else:
        w=self.w
    if self.bare:
        ## In the case of barebones PSO
        self.pos=self.BareBonesUpdate()
    else: ## If not barebones PSO, it's canonical PSO
        for i in range(self.npart): ## Looping through every particle
            lbest,lpos=self.NeighborhoodBest(i)
            c1=self.c1*np.random.random(self.ndim) ## Cognical/individual constant, multiplied by a random number in the range [0,1)
            c2=self.c2*np.random.random(self.ndim) ## Social constant, multiplied by a random number in the range [0,1)
            self.vel[i]=w*self.vel[i]+ c1*(self.xpos[i]-self.pos[i])+c2*(lpos-self.pos[i]) ## Calculating the velocity/position update for each particle

        if (self.vbounds!=None):## In case of velocity bounds, we change the velocity to be within the velocity bounds
            self.vel=self.vbounds.Limits(self.vel)
        self.pos=self.pos+self.vel ## Updating the particle positions by adding the velocity/position update vector to each position
    if (self.bounds!=None): ## In the case of positional bounds, we set the positions to be within the bounds
        self.pos=self.bounds.Limits(self.pos)
    print("ITERATION:", self.iterations)
    p=self.Evaluate(self.pos) ## Calculating the objective function values at all the positions
    self.plotworst(self.worst_val_MSEs,self.worst_train_MSEs)
    self.plotmean(self.mean_val_MSEs,self.mean_train_MSEs)
    self.plotbest(self.best_val_MSEs,self.best_train_MSEs)
    for i in range(self.npart): ##Looping through all the particles
        if (p[i]<self.xbest[i]):## Checking if the current objective function values of the particles are better than their own personal best objective function values
            self.xbest[i]=p[i]
            self.xpos[i]=self.pos[i]
        if (p[i]<self.gbest[-1]): ## Checking if any of the current objective function values are better than the current global best, if so:
            self.bestcount+=1
            self.gbest.append(p[i]) ## Update the global best objective function value
            self.models[i].save('BESTMODELPSO.h5')
            img_name="image"+str(i)+".png"
            new_img_name="bestmodelimage"+str(self.bestcount)+".png"
            os.rename(img_name,new_img_name)
            print("made"+new_img_name)
            img_name = "2image" + str(i) + ".png"
            new_img_name = "2bestmodelimage" + str(self.bestcount) + ".png"
            os.rename(img_name, new_img_name)
            print("made" + new_img_name)
            self.gpos.append(self.pos[i].copy()) ## Update the global best position
        self.gidx.append(i) ## Update the global best particle ID
        self.giter.append(self.iterations) ## Update the global best position found iteration
del self.models
print("Best MSE so far",self.gbest[-1])
bestpos=self.gpos[-1]
bestpos_t = self.transformpositions(np.array([bestpos]))
print("Best architecture so far:",bestpos_t[0])
print("Mean MSE this iteration:",np.mean(p))
print("Best MSE updates",self.gbest)
print("####################################################")
del p
gc.collect()
print("Giter:",self.giter)
self.iterations+=1 ## Increase the iterator
```

A.9.2 Step function from DE, summarizing what happens in a generation/iteration in DE



```
def Step(self):
    new_pos=self.CandidatePositions() ## Pulling out new positions which we will move to in case they are better
    print("ITERATION:", self.iterations)
    p=self.Evaluate(new_pos) ## Calculating the objective function values of the candidate positions
    self.plotworst(self.worst_val_mapes, self.worst_train_mapes)
    self.plotmean(self.mean_val_mapes, self.mean_train_mapes)
    self.plotbest(self.best_val_mapes, self.best_train_mapes)
    for i in range(self.npart): ## Looping through every particle
        if p[i]<self.vpos[i]: ## Checking whether the candidate position of a particle is better than the current position
            self.vpos[i]=p[i] ## In the case of the new candidate position's objective function value being better, we update the current objective function value to the candidate position's objective function value
            self.pos[i]=new_pos[i]
        if p[i]<self.gbest[-1]: ## Checking whether the candidate objective function value is better than the global best
            self.bestcount += 1
            self.gbest.append(p[i])## In case it's better, we update the global best objective function value
            self.models[i].save('BESTMODELDE.h5')
            img_name = "imageDE" + str(i) + ".png"
            new_img_name = "bestmodelimageDE" + str(self.bestcount) + ".png"
            os.rename(img_name, new_img_name)
            img_name = "2imageDE" + str(i) + ".png"
            new_img_name = "2bestmodelimageDE" + str(self.bestcount) + ".png"
            os.rename(img_name, new_img_name)
            print("made" + new_img_name)
            print("made" + new_img_name)
            self.gpos.append(new_pos[i].copy())## In case it's better, we update the global best position
            self.gidx.append(i) ## In case it's better, We update the index ID of the particle that found a new global best position
            self.giter.append(self.iterations) ## In case it's better, we update the iteration where a global best has been found
    print("Best MSE so far", self.gbest[-1])
    bestpos = self.gpos[-1]
    bestpos_t = self.transformpositions(np.array([bestpos]))
    print("Best architecture so far:", bestpos_t[0])
    print("Mean MSE this iteration:", np.mean(p))
    print("Best MSE updates:",self.gbest)
    print("##########################################################")
    del p
    gc.collect()
    print("Giter:", self.giter)
    self.iterations+=1 ## Increase the iteration
```

A.9.3 Step function from MA, summarizing what happens in a generation/iteration in MA



```
def Step(self):
    self.Evolve()
    print("ITERATION:", self.iterations)
    self.vpos = self.Evaluate(self.pos)
    self.plotworst(self.worst_val_mapes, self.worst_train_mapes)
    self.plotmean(self.mean_val_mapes, self.mean_train_mapes)
    self.plotbest(self.best_val_mapes, self.best_train_mapes)
    for i in range(self.npart):
        if self.vpos[i] < self.gbest[-1]:
            self.bestcount += 1
            self.gbest.append(self.vpos[i])
            self.models[i].save('BESTMODELMA.h5')
            img_name = "imageMA" + str(i) + ".png"
            new_img_name = "bestmodelimageMA" + str(self.bestcount) + ".png"
            os.rename(img_name, new_img_name)
            print("made" + new_img_name)
            img_name = "2imageMA" + str(i) + ".png"
            new_img_name = "2bestmodelimage" + str(self.bestcount) + ".png"
            os.rename(img_name, new_img_name)
            print("made" + new_img_name)
            self.gpos.append(self.pos[i].copy())
            self.gidx.append(i)
            self.giter.append(self.iterations)
    p=self.vpos
    print("Best MSE so far", self.gbest[-1])
    bestpos = self.gpos[-1]
    bestpos_t = self.transformpositions(np.array([bestpos]))
    print("Best architecture so far:", bestpos_t[0])
    print("Mean MSE this iteration:", np.mean(p))
    print("Best MSE updates:",self.gbest)
    print("##########################################################")
    del p
    gc.collect()
    print("Giter:", self.giter)
    self.iterations += 1
```

A.9.4 Objective class, used for converting position vector into an DNN for training and evaluation

```
class Objective:
    def __init__(self,
                 x_train=None,
                 y_train=None,
                 x_valid=None,
                 y_valid=None,
                 patience=50,
                 batch_size=1,
                 epochs=100,
                 loss=None,
                 dropout_prob=0.2,
                 verbose=0):
        self.x_train=x_train,
        self.y_train=y_train,
        self.x_valid = x_valid,
        self.y_valid = y_valid,
        self.patience=patience
        self.batch_size=batch_size
        self.epochs=epochs
        self.dropout_prob=dropout_prob
        self.loss=loss
        self.maxseed=0
        self.verbose=verbose

    def create_model(self, pos):
        activation_met="softplus"
        ann = models.Sequential()
        ann.add(layers.Dense(pos[0],kernel_initializer="he_uniform", activation=activation_met, input_shape=(30,)))
        ann.add(layers.BatchNormalization())
        for i in range(1,len(pos)):
            ann.add(layers.Dense(pos[i], activation=activation_met,kernel_initializer="he_uniform"))
            ann.add(layers.BatchNormalization())
        ann.add(layers.Dense(1, activation="relu",kernel_initializer="he_uniform"))
        ann.compile(optimizer=optimizers.SGD(0.001), loss=self.loss)
        return ann
```

```
    def Evaluate(self, pos, queue, i):
        model = self.create_model(pos)
        lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.75, patience=30)
        early_stopping_cb = keras.callbacks.EarlyStopping(patience=50, restore_best_weights=True)
        history = model.fit(self.x_train, self.y_train, validation_data=(self.x_valid, self.y_valid),
                            batch_size=self.batch_size, epochs=self.epochs, callbacks=[lr_scheduler,early_stopping_cb], verbose=self.verbose)
        val_losses = np.array(history.history["val_loss"])
        train_losses = np.array(history.history["loss"])
        index_min = np.argmin(val_losses)
        val=val_losses[index_min]
        train=train_losses[index_min]
        if len(val_losses)<self.epochs:
            model.compile(optimizer=optimizers.Adadelta(), loss=self.loss, metrics=["MSE"])
            history2 = model.fit(self.x_train, self.y_train, validation_data=(self.x_valid, self.y_valid),
                                 batch_size=self.batch_size, epochs=100, callbacks=[early_stopping_cb], verbose=self.verbose)
            val_losses = np.array(history2.history["val_loss"])
            train_losses = np.array(history2.history["loss"])
            index_min = np.argmin(val_losses)
            val = val_losses[index_min]
            train = train_losses[index_min]
        else:
            history2=history
        queue.put([val, train, i,model,history,history2])
        print("Particle",pos,"validation MSE",val)
        del model, history,val,train,lr_scheduler
```

A.9.5 Using the gpustat command to ensure that all three GPUs are being utilized. The green numbers are the utilization in percentage of each GPU.

```
root@C.4595708:~$ gpustat
341bb48df652                  Sat May 21 00:28:46 2022  515.43.04
[0] NVIDIA GeForce RTX 3090 | 52'C,  43 % | 23312 / 24576 MB |
[1] NVIDIA GeForce RTX 3090 | 58'C,  44 % | 24572 / 24576 MB |
[2] NVIDIA GeForce RTX 3090 | 55'C,  43 % | 23313 / 24576 MB |
```

A.9.6 Assigning a specific GPU to be used by tensorflow, using os library in python

```
import os
os.environ["CUDA_VISIBLE_DEVICES"]="0"
```

A.9.7 Assigning a specific set of CPU cores to a task, using the taskset command

```
root@C.4595708:~$ taskset -c 0-6 python3 main.py
```

A.9.8 Assuring that the assigned CPU cores (0-20) are being utilized, using the top command.

We can see the utilization in percentage by looking at the second column.

```
%Cpu0  : 49.6 us, 10.4 sy,  0.0 ni, 40.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 49.5 us,  9.7 sy,  0.0 ni, 40.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  : 48.7 us,  9.5 sy,  0.0 ni, 41.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  : 46.5 us, 11.8 sy,  0.0 ni, 41.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  : 28.6 us,  2.1 sy,  0.0 ni, 69.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  : 27.5 us,  2.8 sy,  0.0 ni, 69.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  : 27.2 us,  1.1 sy,  0.0 ni, 71.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  : 26.4 us,  4.4 sy,  0.0 ni, 69.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  : 34.9 us,  8.5 sy,  0.0 ni, 56.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  : 35.8 us,  7.4 sy,  0.0 ni, 56.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 : 36.2 us,  6.5 sy,  0.0 ni, 57.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 : 36.3 us,  7.2 sy,  0.0 ni, 56.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 : 32.5 us,  7.3 sy,  0.0 ni, 60.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 : 31.3 us,  7.8 sy,  0.0 ni, 60.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 : 31.1 us,  1.7 sy,  0.0 ni, 67.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 : 32.7 us,  2.0 sy,  0.0 ni, 65.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu16 : 46.7 us,  7.7 sy,  0.0 ni, 45.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu17 : 45.6 us, 10.9 sy,  0.0 ni, 43.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu18 : 44.5 us,  9.6 sy,  0.0 ni, 46.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu19 : 45.1 us,  9.8 sy,  0.0 ni, 45.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu20 : 22.6 us,  4.5 sy,  0.0 ni, 72.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu21 :  0.3 us,  0.3 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu22 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu23 :  0.3 us,  2.3 sy,  0.0 ni, 97.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu24 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu25 :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu26 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu27 :  0.7 us,  0.7 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu28 :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu29 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu30 :  1.0 us,  0.3 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu31 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu32 :  0.3 us,  0.3 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu33 :  0.0 us,  3.9 sy,  0.0 ni, 96.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu34 :  0.3 us,  0.7 sy,  0.0 ni, 99.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu35 :  0.7 us,  2.9 sy,  0.0 ni, 96.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu36 :  0.0 us,  0.7 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu37 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu38 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu39 :  1.3 us,  0.3 sy,  0.0 ni, 98.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu40 :  1.0 us,  4.3 sy,  0.0 ni, 94.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu41 :  0.7 us,  0.3 sy,  0.0 ni, 99.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu42 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu43 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
```