DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2023

Machine Learning model applied to Reactor Dynamics

SH204X Master Thesis Project Report

Nikitopoulos Dionysios-Dimitrios

KTH ROYAL INSTITUTE OF TECHNOLOGY
NUCLEAR ENERGY ENGINEERING
TRITA-SCI-GRU 2022:042

Author

Nikitopoulos Dionysios-Dimitrios <ddni@kth.se>
Nuclear Energy Engineering
KTH Royal Institute of Technology

Project Company

Westinghouse Electric Sweden AB
Västerås, Sweden

Examiner

Jan Dufek
Stockholm
KTH Royal Institute of Technology

KTH Supervisor

Walter Villanueva
Stockholm
KTH Royal Institute of Technology

Westinghouse Team

Supervisor

Tobias Strömgren

Advisors

Camilla Rotander

Andreas Wikström

Abstract

This project's idea revolved around utilizing the most recent techniques in Machine Learning, Neural Networks, and Data processing to construct a model to be used as a tool to determine stability during core design work. This goal will be achieved by collecting distribution profiles describing the core state from different steady states in five burn-up cycles in a reactor to serve as the dataset for training the model. An additional cycle will be reserved as a blind testing dataset for the trained model to predict. The variables that will be the target for the predictions are the decay ratio and the frequency since they describe the core stability.

The distribution profiles extracted from the core simulator POLCA7 were subjected to many different Data processing techniques to isolate the most relevant variables to stability. The processed input variables were merged with the decay ratio and frequency for those cases, as calculated with POLCA-T. Two different Machine Learning models, one for each output parameter, were designed with Pytorch to analyze those labeled datasets. The goal of the project was to predict the output variables with an error lower than 0.1 for decay ratio and 0.05 for frequency. The models were able to predict the testing data with an RMSE of 0.0767 for decay ratio and 0.0354 for frequency.

Finally, the trained models were saved and tasked with predicting the output parameters for a completely unknown cycle. The RMSE was even better for the unknown cycle, with 0.0615 for decay ratio and 0.0257 for frequency, respectively.

Keywords

Master Thesis, Machine Learning, stability ,Energy distribution profiles, Prediction, frequency, decay ratio, Data processing, POLCA-T, Pytorch, testing data, RMSE.

Acknowledgements

This Thesis was only possible with the initial help and guidance of my KTH supervisor Walter Villanueva. He played a vital role in motivating me to pursue my interest in innovation and groundbreaking technologies in the Nuclear field, despite my limited background. Additionally, he brought me into contact with Westinghouse and kept encouraging me to accept the topic of Machine Learning and focus all my efforts on delving into it.

I would also like to sincerely thank my Westinghouse team consisting of Tobias Strömgren, Camilla Rotander, and Andreas Wikström. Their feedback and suggestions during our weekly meetings pushed me to keep researching and improving the model. Although he participated in only a few meetings, Andreas provided many ideas and, most importantly, the suggestion to undertake data processing, which provided me with a holistic view of data processing. Camilla's expertise in stability and POLCA-T was invaluable in solving many of the problems encountered and pointing in the right direction on the selection of input variables. My immediate Westinghouse supervisor, Tobias Strömgren, deserves special praise for handling everything from technical support to timeline management with efficiency and kindness. Without his constant vigilance and interest, this project would not have been accomplished. I want to also thank Westinghouse Electric Sweden AB for its financial support that enabled traveling and research for this Thesis.

Finally, I would like to thank my colleague and friend from KTH, Richard Härlin, that also undertook a Thesis project for Westinghouse some months prior. By observing his progress, I was able to much more accurately predict the time and effort requirements for future parts of the Thesis. Additionally, discussing the work and problems and enjoying his company while commuting to Västerås helped me remain more energized and committed.

Acronyms

GDC General Design Criteria

ML Machine Learning

AI Artificial Intelligence

NN Neural Networks

FP Forward Propagation

BP Backward Propagation

MSE Mean Square Error

RMSE Root Mean Square Error

LR Learning Rate

GD Gradient Descent

SGD Stochastic Gradient Descent

ReLU Rectified Linear Unit

DL Deep Learning

BN Batch Normalization

TF TensorFlow

MAE Mean Absolute Error

Contents

1 Introduction

1.1 Background

Reactor stability is crucial for safe operation, as large oscillations can negatively impact the fuel cladding integrity. In BWRs, stability problems usually only occur during reactor start-up or significant transients during operation. However, several new generation BWR's characteristics bring stability to the forefront [1]:

1. Operation at high nominal reactor power
2. Natural core cooling
3. Increased core size, resulting in stronger susceptibility to oscillations

The US Regulatory Commission's Standard review plan [2] details the acceptance criteria for constructing nuclear reactors. Specifically, the General Design Criteria GDC-10 and GDC-12 deal with reactor stability issues:

- *"GDC 10, "Reactor Design," requires that specified acceptable fuel design limits not be exceeded during any condition of normal operation, including conditions that result in unstable power oscillations with the scram system available."*
- *"GDC 12, "Suppression of Reactor Power Oscillations," requires that oscillations be either not possible or reliably and readily detected and suppressed."*

∗ Both GDCs are exact quotes from [2].

Stability analysis is crucial in nuclear design to establish the stability margins for all the reactor's operation modes. The design margins detail the interval between the system and the limits to unstable operation. The analysis for this complex phenomenon requires a combination of different fields and techniques [1]:

- transient thermal-hydraulics
- neutron kinetics
- fuel management
- numerical models
- simulations
- 3D modeling

1.1.1 Instabilities in Boiling Water Reactors

Boiling Water Reactors' complex dynamics result in many instability types and different modes of oscillation. Although there is no consensus on the naming of the different instability types, some of the most prevalent ones are:

1. Control System Instabilities: This type of instability is usually related to an actuator of the reactor (typically a valve) following the signals given by the control system to regulate a variable. The unstable control actions are usually negligibly small and these types of instabilities result in low-frequency power or flow oscillations within the normal reactor noise range.
2. Channel Thermohydraulic Instabilities: The most common instability for Boiling Water Reactors is the density-wave/channel-flow instability, characterized by the channel flow becoming unstable and oscillating at some frequency. Channel thermohydraulic oscillations are also called "local" instabilities because they may affect a single channel in the core of a large Boiling Water Reactor. This type of instability is fortunately extremely rare but can prove quite dangerous and go undetected for a long period of time because of its single-channel nature.
3. Coupled Neutronic-Thermohydraulic Instabilities: Neutronic-Thermohydraulic Instabilities arise from the culmination of all the destabilizing effects of the thermohydraulic feedback paired with those of the neutronic feedback.
    - **Neutronic feedback:** Involves the changes in reactivity caused by void fraction fluctuations, leading to neutron flux oscillations with a frequency close to the inverse of the density-wave time constant.
    - **Thermohydraulic feedback:** Affects the inlet flow rate similarly to the effect of the channel flow stability.

Coupled Neutronic-Thermohydraulic Instabilities are the relevant type for this Thesis' investigation, thus their mode of oscillation present more interest:

- **Core-wide/in-phase mode:** The power and inlet flow of the whole core oscillate in phase for all channels.
- **Regional/out-of-phase mode:** The power of one half of the core oscillates out-of-phase with respect to the power of the other half. The inlet flows of both halves are also out-of-phase with respect to each other.

The neutron flux caused by a spatial distribution of reactivity oscillates resembling a string. The dominant modes of oscillation for Coupled Neutronic-Thermohydraulic Instabilities in the radial direction for a cylindrical reactor are presented in Figure 1.1. The mode of oscillation depends on the excitation and the reactor conditions [3].

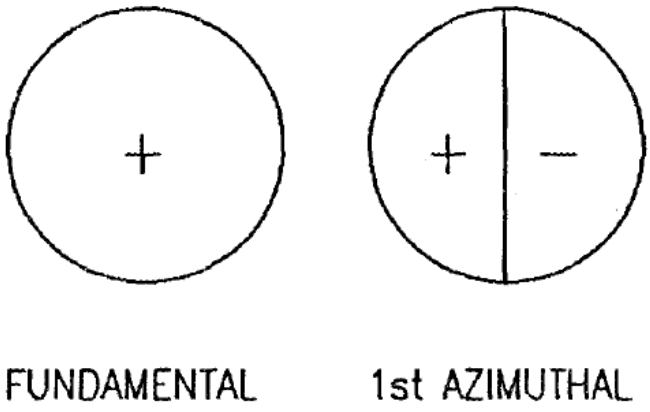


Figure 1.1: Modes of oscillation for the coupled neutronic-thermohyclraulic instability type. [3]

1.1.2 Decay ratio

The decay ratio is widely used for monitoring the stability of boiling water reactors. It is defined as the ratio between two consecutive maxima of the system's impulse response, thus being a unitless decimal value (smaller than 1), as can be seen in Figure 1.2.

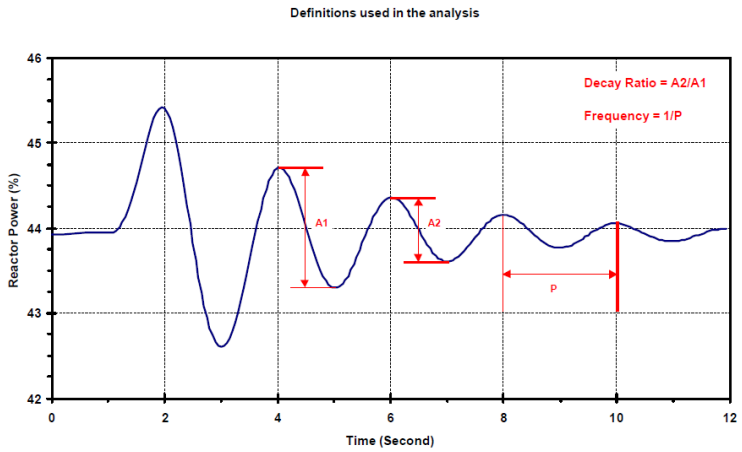


Figure 1.2: Definitions of decay ratio and frequency. [4]

In practical applications, the decay ratio is determined via noise analysis of measured neutron flux signals and can serve as a reliable quantifier of the degree of stability. The perturbations in a reactor with a small decay ratio have proven to be damped more rapidly than those in a reactor with a larger decay ratio [5].

1.1.3 Resonance Frequency

As detailed above, oscillations are directly tied to the appearance of instabilities. Specifically, the oscillation period characterizes how oscillations grow or attenuate. Resonance frequency represents the inverse of the oscillation period, and is expressed in Hz, as can be inferred from Figure 1.2.

1.2 Goal

The importance of the decay ratio and (resonance) frequency set them apart as the most crucial variables to ensure stability. The acceptance criteria for stability demand a decay ratio of 0.65-0.70 (different for different utilities) for a pre-determined state point in the operating domain. The analyses of core stability are performed by utilizing system codes, such as POLCA-T. These codes generate a 3D modeling of the reactor core, coupled with the thermal-hydraulic modeling of every fuel assembly in the core and the thermal mechanic fuel properties.

POLCA-T's calculations and core analyses are precise but demanding in computational power and time. Westinghouse's proposition was to design a Machine Learning model able to predict the core stability as a simplified tool during core design work. A database will be created by collecting data critical to core stability from the steady state core simulator POLCA7, and the calculated decay ratio and resonance frequency. The decay ratio and resonance frequency results were calculated by introducing a (reactivity) disturbance (usually a control rod disturbance) in POLCA-T, resulting in a decaying impulse response. The decay ratio and resonance frequency values are estimated from the average of the consecutive amplitude ratios, as can be observed in Figure 1.3.

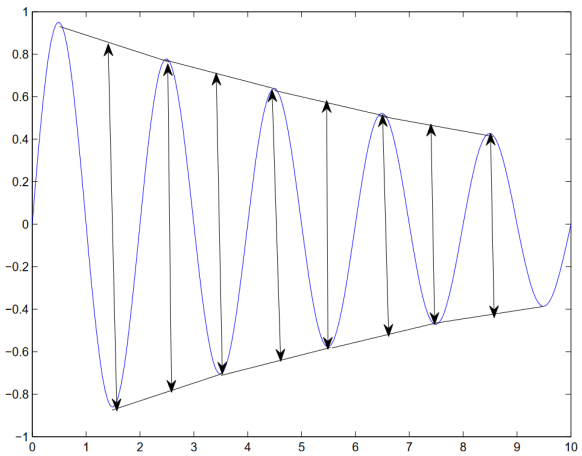


Figure 1.3: Value estimations of decay ratio and frequency.

Then, the model will be trained on that dataset. The trained algorithm needs to be able to predict the output values (decay ratio and frequency) for known and unknown cases. The predicted values need to approximate POLCA-T's results (the target was set to an error lower than 0.1 for decay ratio and 0.05 for frequency) for the algorithm to be considered a valuable tool in core design.

1.3 Outline

The Outline for this paper closely follows the structure of the work for this Thesis:

- Chapter 2 explains the theoretical concepts and selections (with motivations) that served as the backbone for the Machine Learning implementation
- Chapter 3 compares the most popular Machine Learning frameworks, introduces some of their most important features, and motivates the selection of Pytorch
- Chapter 4 briefly illustrates the work time allocation and details the methodology, coding, and graphical consideration for each stage of the work
- Chapter 5 details the prediction operation of the models and its evaluation of unknown data (with illustrations and explanations)
- Chapter 6 reflects on the lessons learned from the entirety of this Thesis, offers some suggestions on future work, and the closing remarks on this experience

2 Machine Learning Theory

2.1 Machine Learning introductory concepts

The term Machine Learning (ML) refers to a specific subset of Artificial Intelligence (AI) focused on developing systems that can autonomously utilize databases to learn. The term ”learn” in this context refers to distinguishing patterns emerging from the data points, making connections between them, and ultimately arriving at conclusions. The conclusions include but are not limited to data classification, output values predictions, and decision-making without the need for additional human input.

The most common methods utilized in ML model creation are supervised and unsupervised ML. These two methods differ in the database format, the main concept for arriving at conclusions, and the techniques implemented. Both of these methods have already been implemented in various industrial sectors, like financing, health care, transportation, retail, and others, providing additional capabilities, often surpassing the pattern recognition capabilities of the human brain and transforming the field of data science.

Supervised ML centers around the availability of labeled data. Labeled data are data that have been processed in some form and attributed to a label. For example, in the case of vehicle image recognition, the images have been pre-attributed with the correct description (car, boat, plane, etc.).The labeled data are used as input for the algorithm, producing outputs and comparing them with the correct ones provided. Then the algorithm will adopt its model to reduce the error based on the comparison and repeat the process. The model goes through the labeled dataset multiple times and starts developing a correlation between characteristics shared among all identical labels, thus learning. In this example, the algorithm would probably conclude that to classify a vehicle as a plane; it has to have wings. Then, after the algorithm has been trained in the labeled database, it can utilize the conclusions to classify/recognize unlabeled data. If the previous connection between plane-wings holds true, it would be able to instantly classify any images of planes where the wings are distinguishable. The main techniques implemented in supervised ML include classification, regression, prediction, and gradient boosting. The typical applications are predictions for future data or data with unknown outputs, like this Thesis’ goal.

Contrarily, unsupervised ML is used for unlabeled data. Since the correct conclusion is not predetermined or available, the algorithm needs to distinguish similarities in the dataset without a starting point. For that reason, unsupervised ML has a more stochastic nature and is mainly applied in fields without established concrete connections between data points, like sales. An unsupervised ML model can be used to import a store's sales database and find underlying connections between purchases and buyer characteristics, for example, younger people buying more electronics. The main techniques implemented in unsupervised ML include self-organizing maps, nearest-neighbor mapping, k-means clustering, and singular value decomposition. The typical applications are marketing recommendations, user preference predictions, and data outlier identification [6].

2.2 Neural Networks

The idea of creating a complex computational system, with many layers of nodes activating similarly to brain neurons, has existed since 1943. However, the limitation of the computational power of the era delayed the construction of the first functional multi-layered Neural Network (NN) until 1975 [7].

In their original inception, NNs were visualized as a computational system mimicking the problem-solving processes of the human brain, see Figure 2.1. However, with the advancement in the fields of AI, researchers diverged from that to compartmentalizing different NNs to specific tasks, including computer vision, speech recognition, social network filtering, and medical diagnosis. Additionally, NNs excel at solving complex problems by modeling the relationships between non-linear inputs and outputs.

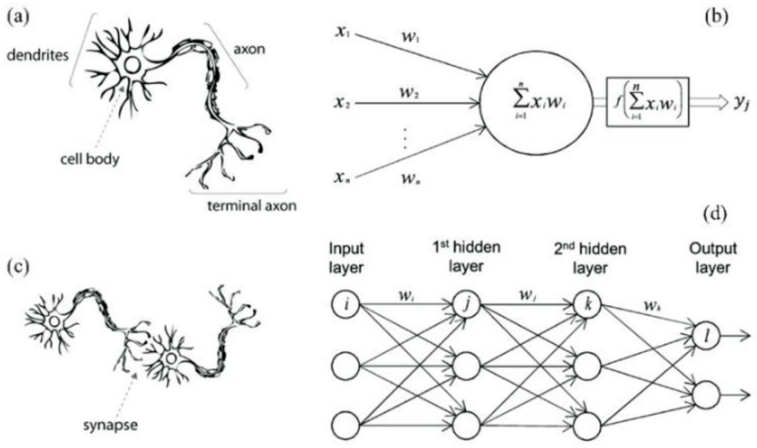


Figure 2.1: A biological neuron in comparison to an artificial neural network. (a) Brain neuron, (b) Artificial neuron, (c) Neuron + biological synapse, (d) Artificial neural network [8]

Every NN comprises an input layer, connected to a number of hidden layers (depending on the requirements of the problem), and finally, an output layer. The layers are connected via their nodes, creating a network of interconnected nodes. Different nodes can be activated simultaneously by the same input, and the activation spreads throughout the different layers of the network until the output. An example of a simple NN is presented in Figure 2.2:

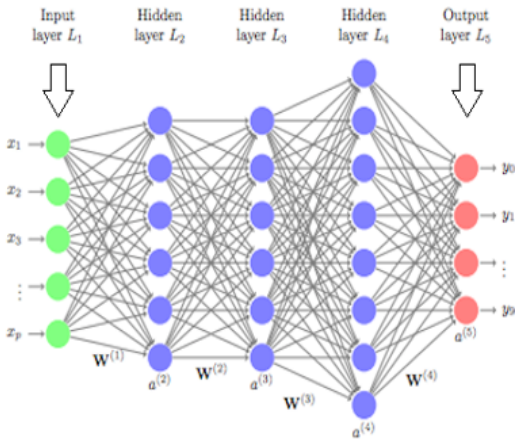


Figure 2.2: Simple Artificial Neural Network with three hidden layers [9]

2.3 Forward and Backward Propagation

The combination of Forward Propagation (FP) and Backward Propagation (BP) enables learning in NNs. During FP, the information flows from input -with the provided weights- to output. Then, BP evaluates the FP and adjusts the weights based on the observed loss. The circle continues with another FP with the new weights until the loss has been reduced to the desired level, as illustrated in Figure 2.3:

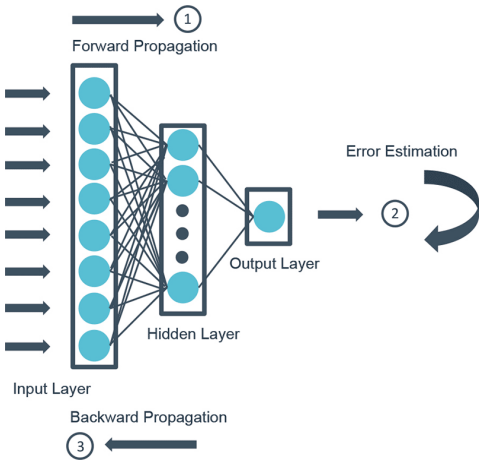


Figure 2.3: Forward and Backward Propagation circle [10]

2.3.1 Forward Propagation

FP flows through the NN by performing a series of operations for each layer:

1. Input layer:
    - The data is inserted into the NN and transferred to the first hidden layer.
2. Hidden layers:
    - Each hidden layer's nodes will perform mathematical calculations to combine the data points from the input from the previous layer with a set of coefficients and assign appropriate weights to the inputs.
    - These input-weight products are then summed up.
    - The sum is passed through a node's activation function, which is tasked with determining whether the signal is impactful enough to progress further through the NN.
    - The output of that hidden layer is transferred to the next layer as its input.
3. Output layer:
    - The hidden layers link to the output layer (output collection) [7].

The connection between the operations, as mentioned above, between consecutive layers is presented in Figure 2.4:

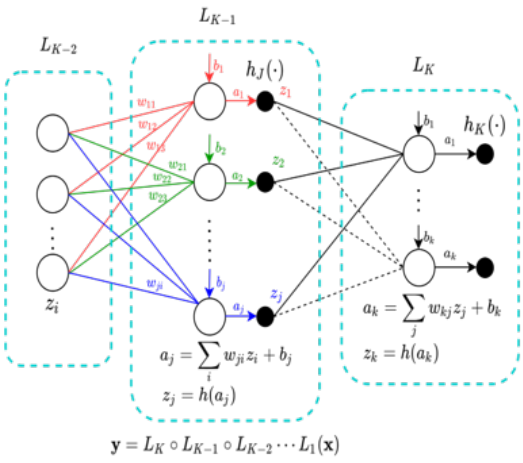


Figure 2.4: Forward propagation between layers [11]

2.3.2 Backward Propagation

FP is completed with the output collection at the output layer. Then, BP begins by evaluating this predicted output s against an expected output y (supervised ML). For this evaluation, a cost function needs to be defined, ranging in complexity from MSE (mean squared error) to cross-entropy. Root Mean Square Error (RMSE) is a standard way to measure the error of a model in predicting quantitative data [12]. This cost function is noted with C and described by Equation (1):

$$C = cost(s, y) \tag{1}$$

BP's goal is to minimize C through a readjustment of the weights and biases used in the preceding FP. The gradients of C control this adjustment for each parameter [13]. The gradients for each parameter at each layer are called local gradients and can be easily calculated with the chain rule starting from the output layer and going back a layer at a time. A representation of the chain nature of BP in a NN is presented in Figure 2.5:

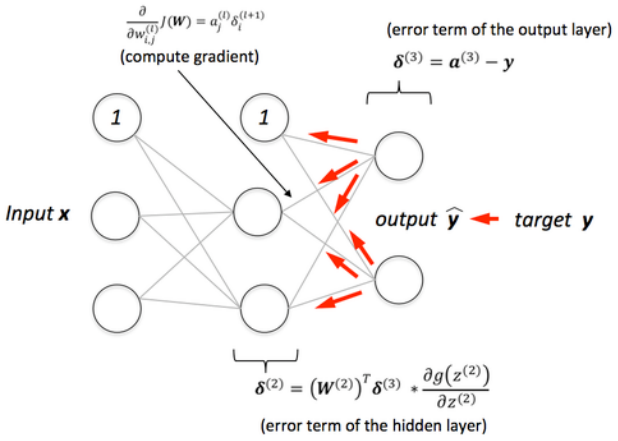


Figure 2.5: Backward propagation between layers [14]

BP is challenging to implement and is hard to make efficient without special optimization libraries. Fortunately, the most popular ML and DL libraries, such as TensorFlow and Pytorch (Section 3), include implemented BP using optimized strategies.

2.4 Learning Rate

The most reliable existing method for the estimation of the weights and their coefficients arises from an optimization procedure called Stochastic Gradient Descent (SGD) [15], presented in Figure 2.6:

- The error gradient is calculated from the training split of the dataset.
- The BP of the errors algorithm indicates the amount of influence the weights have on the error (estimated weight error).
- That amount is scaled by the Learning Rate (LR), a modifiable hyperparameter that usually ranges between 0.01 and 0.1.
- The weights are updated by the product of the LR and the estimated weight error.

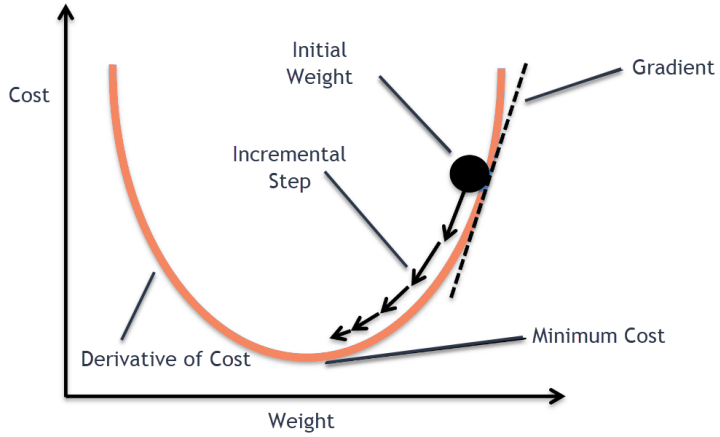


Figure 2.6: SGD Algorithm [16]

The LR is crucial to the NN's performance since it essentially controls the ability of the model to learn and adapt to a given problem. A model with an optimal LR will perform a very close approximation of the required function in the set of training epochs (The number of times that the whole training dataset is passed through). There are benefits to smaller and larger LRs, but they can also lead to significant problems for the whole model.

A larger LR results in more significant changes to the weights with each epoch, thus improving the computational speed of the model since it requires fewer training epochs. However, this can lead to abnormally large weight updates forcing the model to converge in a suboptimal solution and then oscillate over the remaining training epochs. In extreme scenarios, the weight updates may increase uncontrollably, leading to a numerical overflow.

A smaller LR equates to smaller changes to the weights per epoch, allowing the model to converge to the optimal set of weights slowly, but it will require more training epochs to be adequately trained. In the case of the LR being too small for the model, the process may never converge or get stuck on a suboptimal solution [17].

Unfortunately, no analytical calculation exists for the optimal LR. Instead, configuring the LR for a NN requires tuning through a set of different techniques:

- **Diagnostic plots** (line plots of loss over epochs) can visualize many properties, like the rate of learning over epochs and the detection of detrimentally small or large LRs, from shape or oscillations.
- **A grid search** (sensitivity analysis of the LR) can highlight the order of magnitude for optimal LRs and the relationship between LR and performance.
- **"Momentum"** (Section 2.5.3) (exponentially weighted average of the last updates to the weights) can improve the optimization process by directing updates to the correct direction of the curve.
- **A learning rate schedule** (varying the LR over the training process) can allow fine-tuning the weight changes towards the end of the learning process.
- **An adaptive learning rate** (the LR is adjusted by an optimizer based on the performance of the model on the training dataset) can substantially decrease the convergence time [15], as can be observed in Figure 2.7:

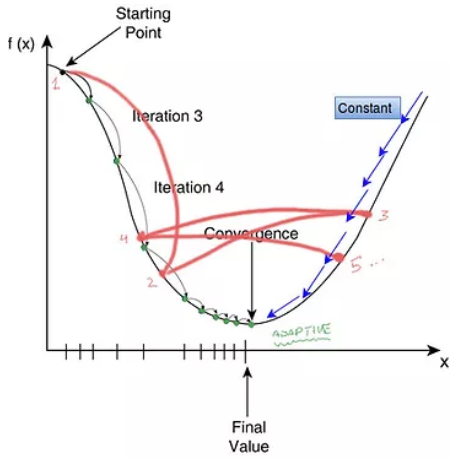


Figure 2.7: Convergence time improvement with adaptive LR [18]

2.5 Optimizers

Optimizers are algorithms specifically tasked with monitoring the learning process in the NN and automatically adjusting the weights and LR to reduce the losses and convergence time and increase the overall accuracy [19]. However, there exists a plethora of optimizers available with their set of advantages and disadvantages. Hence the need arises to deeply understand these algorithms and recognize where their benefits outweigh their disadvantages for the given problem.

2.5.1 Gradient Descent

The most basic and classical optimization algorithm is the Gradient Descent (GD). GD is a first-order optimization algorithm. It is tasked with calculating the loss function's first-order derivative, locating its minima, and calculating the weight change to reach it. Its simplicity and robustness for simple tasks make it the easy choice for linear regression and classification problems.

The GD algorithm is appealing for basic problems because it is easy to understand, implement, and simplifies the computation process. However, as the complexity of the task and the database size increase, GD starts facing serious problems:

- Due to the direct change of weights based on the loss function's first derivative, the GD may trap the process at local minima, as illustrated in Figure 2.8.
- Due to the change of weights resulting from the calculation of the gradient on the whole dataset, for larger datasets, the convergence time increases uncontrollably.
- Memory requirements for larger datasets' gradient calculation are prohibitive.

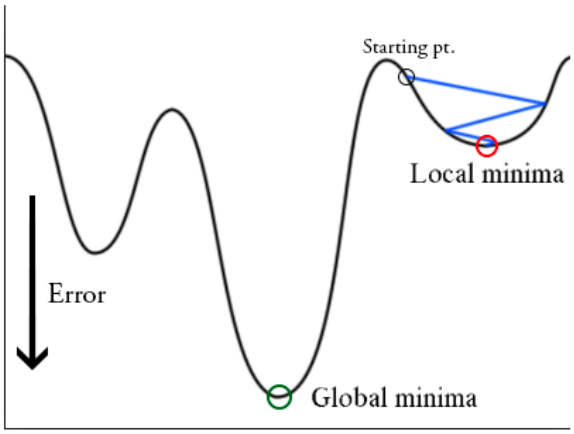


Figure 2.8: Gradient Descent Stuck at Local Minima [20]

2.5.2 Stochastic Gradient Descent

SGD is an improvement on the GD optimizer to counteract large databases' memory and time issues. These issues emerge in GD due to the change of weights resulting from calculating the gradient of the whole dataset. The stochastic nature lies in calculating the gradient of a random batch instead of the whole database. That leads to an increased number of shorter epochs, but the model parameters are altered after each. The frequent updates lead to shorter convergence time, and more importantly, memory is not encumbered by the values of loss functions.

Because only a small batch size of the data is selected in each epoch, the algorithm includes a lot more noise than the GD, leading to a more unstable path, as is observable in Figure 2.9. However, despite the higher number of iterations to reach the local minima, the computation cost is still significantly less than that of the GD. However, the high variance in model parameters can lead to unnecessary iterations even after achieving global minima. [21].

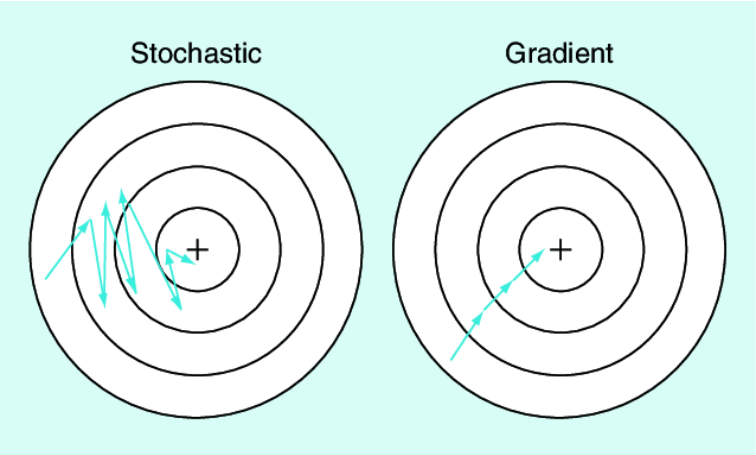


Figure 2.9: Stochastic gradient descent compared with gradient descent. [22]

Despite efficiently dealing with the problems with larger datasets, SGD and all other variations of the GD methodology suffer from the GD method's inherent flaws:

1. The LR cannot be analytically calculated before, and relying on the GD method for it can always cause the progress to get stuck.
2. GD variations operate under a constant LR for all the parameters. However, the ability to configure different LRs for different parameters can be valuable.
3. The risk of getting trapped at local minima is unavoidable with GD variations [19].

2.5.3 Momentum

The addition of momentum dramatically reduces the problem of SGD with high variance in the variables. In addition, momentum can direct the learning process to an easier path, thus accelerating the convergence and reducing fluctuations and oscillations around minima. The significance of momentum in the path to a global minimum is illustrated in Figure 2.10. The momentum term is noted with γ and is usually set to 0.9.

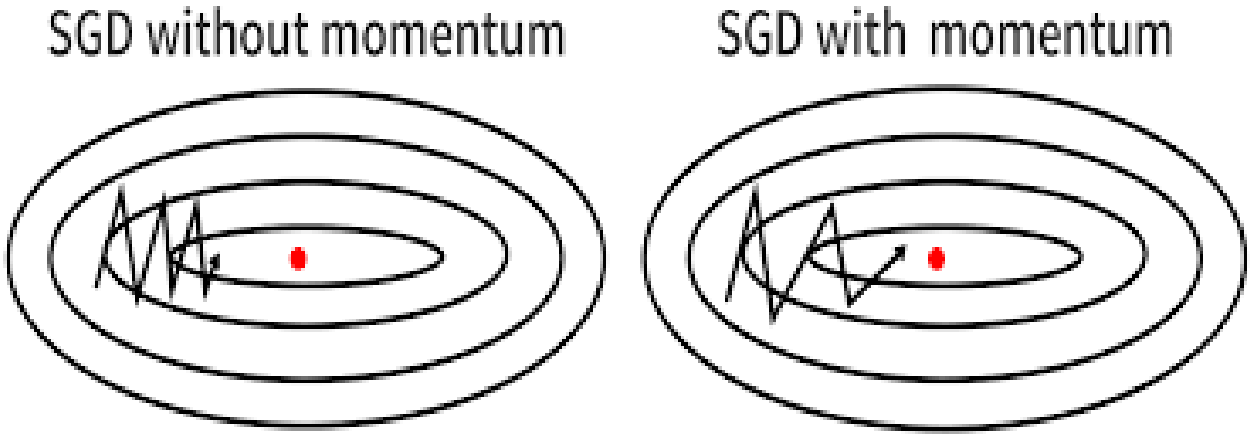


Figure 2.10: SGD with and without momentum comparison [23]

Momentum uses the information from the weights' previous updates to positively influence the current update. A weight that keeps moving in a particular direction will accumulate some momentum related to that direction. This accumulated momentum is retained as a portion of all the previous gradients. When a weight change encounters a shift in direction (usually at local minima), the accumulated momentum will keep steering it in the previous direction, as shown in Figure 2.11. Momentum is significant enough in most cases to overcome the local minima [24].

2.6 Adaptive optimization

Adding momentum at SGD solves the issues of being trapped at local minima but introduces momentum as a new hyperparameter that must be configured correctly. Additionally, the problem with constant hyperparameters cannot be fixed by either of these optimizers alone. Therefore, a new direction in the optimizing method was developed, known as adaptive optimizers. The name results from having the ability to constantly adapt the essential parameters like LR and momentum for every weight in the network, therefore changing along with the weights.

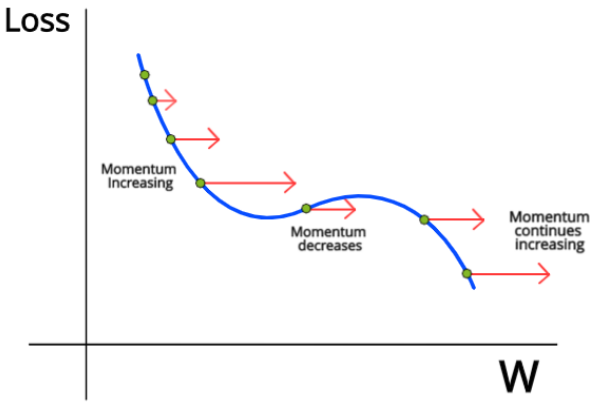


Figure 2.11: Momentum visual example [24]

2.6.1 Adagrad (Adaptive Gradient Descent)

Adagrad (Adaptive gradients) is an optimizer that automatically changes the LR during each update. The complete method centers around dividing the LR with the history of gradient value until that point, meaning that LR depends upon the difference in the parameters during training. Thus, a big parameters change leads to a small LR change. Adagrad is more helpful in applications to datasets that contain both sparse and dense features, where a constant LR for all features is counterproductive.

Non-sparse features accumulate a large history value because of the increased frequency of updates. Therefore, the result from dividing the LR with the large history is a small effective LR. Conversely, sparse features have a smaller gradient history value, resulting in a large effective LR. This is illustrated in Figure 2.12, where the different colored curves represent different optimizers: GD (red), Momentum GD (yellow), and Adagrad (blue).

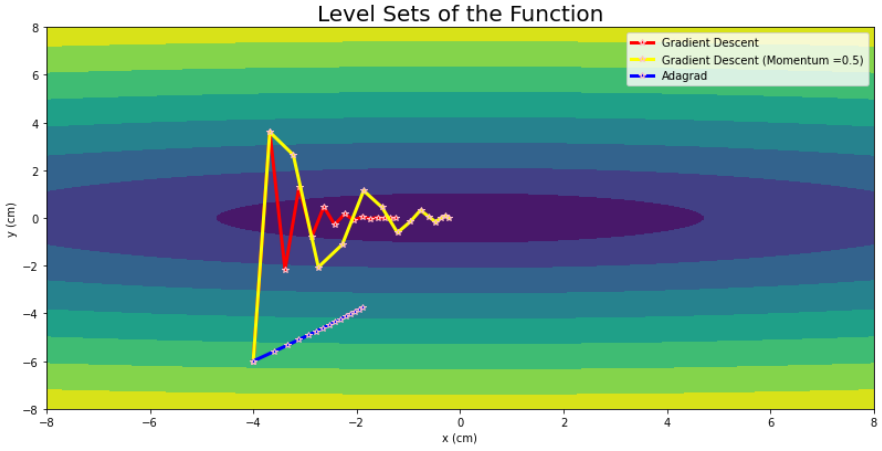


Figure 2.12: GD vs Momentum vs Adagrad [25]

As can be easily observed in Figure 2.12 Adagrad clearly sets a more efficient path without oscillations. However, its convergent ability is insufficient near the end of the learning process. This results from the main downside of Adagrad: an aggressive, monotonic decrease of the LR. As the squared gradients in the denominator sum up, the effective LR keeps decreasing. Once the LR becomes insignificantly small, the model stops being able to learn further. This became known as the problem of varying gradients [24].

2.6.2 RMSprop (Root Mean Square propagation)

RMSprop was designed to solve the problem of varying gradients with great success. RMSprop modified the method of Adagrad by taking the exponential moving average of the gradients rather than their sum. This assigns a higher importance factor to the most recent gradient updates and ensures that the LR constantly changes without the learning rate decaying as quickly. However, one significant disadvantage of RMSprop is that it still requires a manual definition for the LR, which can lead to convergence problems. Additionally, the suggested constant value is unsuitable for every application.

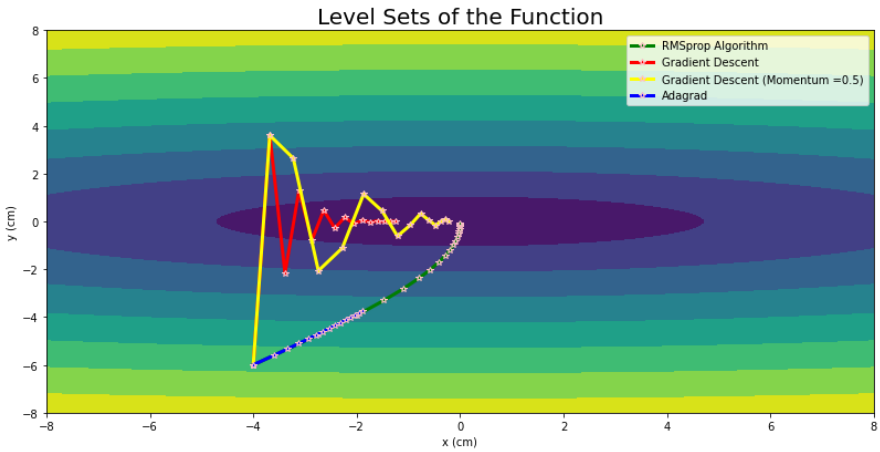


Figure 2.13: GD vs Momentum vs Adagrad vs RMSprop [25]

As illustrated in Figure 2.13, as the GD converges towards the optimal point, it creates considerable oscillations in the vertical direction. On the contrary, RMSprop restricts the movement in the vertical direction and shifts it to the horizontal direction. Therefore, the convergence to the global minimum becomes possible from the point where Adagrad stops learning. Finally, in comparison to momentum, RMSprop provides more reliable and faster convergence.

2.6.3 Adam (Adaptive Moment Estimation)

Adam was designed to combine the advantages of Adagrad and RMSprop, thus outperforming both of them. The idea centers around storing an exponentially decaying average of past squared gradients like AdaDelta and an exponentially decaying average of past gradients M(t) (called uncentered variance). This allows the Adam optimizer to update the LR for each network weight constantly. The analytical methodology behind Adam follows these steps:

1. Adam calculates an exponential moving average of the gradient and the squared gradient.
2. Two new parameters, beta1, and beta2, are introduced that control the decay rates of the moving averages, with initial values close to 1.
3. The initial values of beta1 and beta2 introduce a bias of moment estimates towards zero.
4. The biased estimates are being calculated.
5. The bias-corrected estimates are being calculated to eliminate the bias [26].

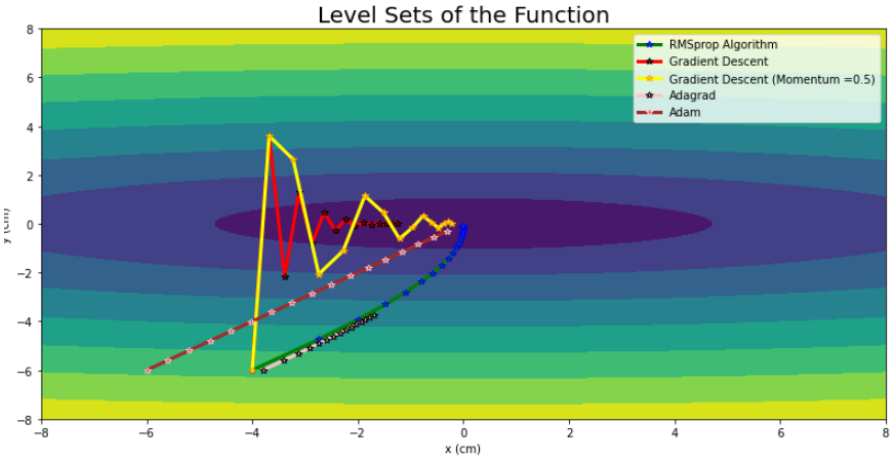


Figure 2.14: GD vs. Momentum vs. Adagrad vs. RMSprop vs. Adam [25]

As seen in Figure 2.14, Adam follows a straight-line path directly to the convergence zone. Adam's accumulation of the sum over previous gradients eliminates the oscillations that exist in all other optimizers. The straight line results from the squared gradient term are also prevalent in RMSprop. The most impressive feat in this Figure is that Adam has the farthest starting point from the minimum (-6,-6), instead of (-6,-4) for all other optimizers, and yet Adam still manages to converge fastest than with minimal iterations [25].

2.7 Motivation for selecting Adam for this Thesis

By analyzing the strengths and weaknesses of the most commonly used Optimizers, it was concluded that Adam possesses various advantages over the alternatives. Adam's unique characteristics have set it since its inception as a benchmark for deep learning papers, and it is recommended as a default optimization algorithm. These include easy implementation, faster convergence time, low memory requirements, and autonomy in parameter estimation and adjustment [26].

Most of the current literature, both in Machine Learning and Deep Learning, highly recommends its use unless a very experienced user has a specific reason to prefer an alternative:

- *"Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances... its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. Insofar, Adam might be the best overall choice."* [27]
- On an analysis on Adam, titled *"ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION"* the authors' collected Figure 2.15 showing the comparison between different optimizers on image recognition, concluding that *"Adam shows better convergence than other methods"* [28].

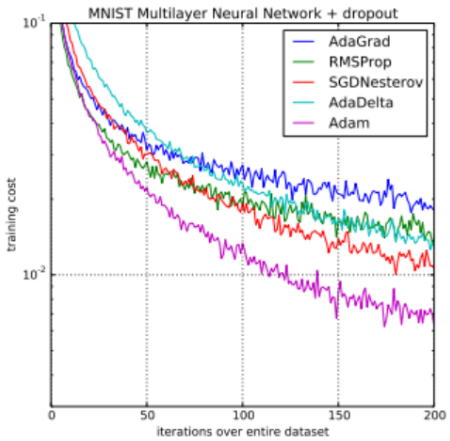


Figure 2.15: Training of multilayer neural networks on MNIST images. [28]

Adam uses a predefined set of starting parameters according to the library's settings. According to [28] *"Good default settings for the tested machine learning problems are* $\alpha = 0.001$, $\beta 1 = 0.9$, $\beta 2 = 0.999$ *and* $\varepsilon = 10^{-8}$ *".*

For Pytorch (Section 3), they are initially set to:

- **α (LR)** =0.001
- **β1 (exponential decay rate for the first moment estimates)** =0.9
- **β2 (exponential decay rate for the second-moment estimates)** =0.999
- **ε (correcting number to prevent any division by zero)** = 10^{-8} [26].

In more recent studies, SGD with momentum has been argued to outperform Adam, however in-depth analyses on the topic concluded:

1. *"…. by tuning all available hyperparameters at scales in deep learning, more general optimizers never underperform their special cases…. fine-tuned Adam is always better than SGD, while there exists a performance gap between Adam and SGD when using default hyperparameters."* [29]
2. *"Trying out all possible optimizers to find the best one for a project is not always possible…. If you have the resources to find a good learning rate schedule, SGD with momentum is a solid choice. If you are in need of quick results without extensive hypertuning, tend towards adaptive gradient methods."* [30]
3. *"Overall, Adam/Nadam have lowest training error/loss, but not validation error/loss" (Figures 2.16 and 2.17) "… this paper suggests that we start training with Adam and switch to SGD when a triggering condition is satisfied. This shrinks the generalization gap between SGD and Adam."* [31]

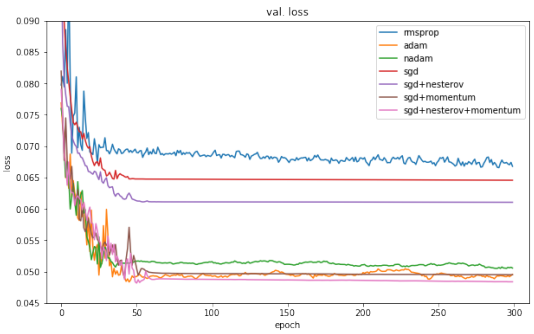


Figure 2.16: Validation loss comparison

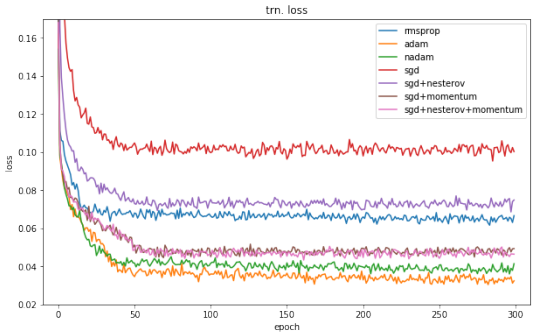


Figure 2.17: Training loss comparison [31]

In conclusion, within the time and knowledge limits for the completion of this Thesis, Adam appears as the most fitting Optimizer. Without extensive hyperparameter fine-tuning, its advantages outweigh its disadvantages and seem to outperform all alternatives.

2.8 Activation functions

As explained in Chapter 2.2, after the weights have been calculated and assigned to the inputs in each hidden layer, their sum is passed through the activation function of that layer. Adam will handle weights, LR, and their updates, but what about the activation functions? How do they operate, and what activation function is the most fitting for this project?

An activation function enables the NN to identify and ”learn” complex patterns in the dataset. Looking back at the analogy of the neurons activations in the human brain, the activation function imitates the operation of an axon, which is responsible for deciding which impulses will be carried to the next neuron, as shown in Figure 2.18.

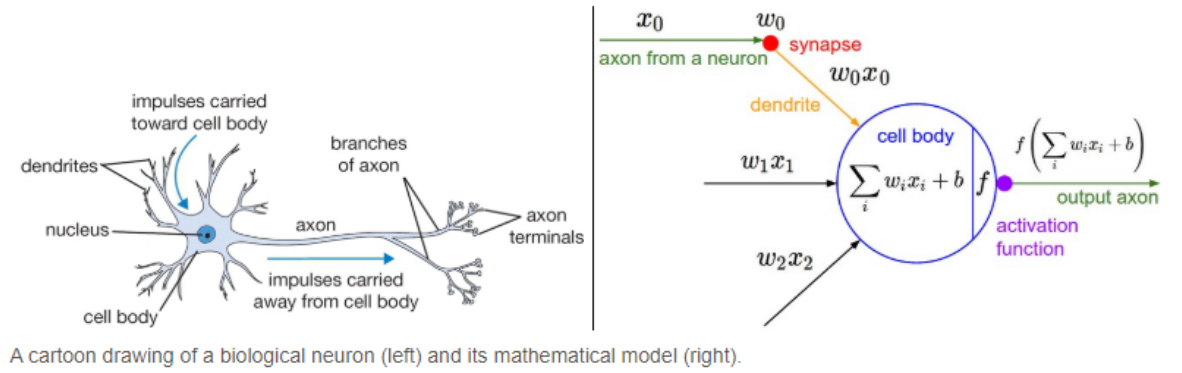


Figure 2.18: A cartoon drawing of a biological neuron (left) and its mathematical model (right) [32]

Activation functions are essential to the learning process because they serve as gatekeepers of the neuron’s output:

- They restrict the value of the output from the neuron to a certain limit. Without an upper limit, the output can go very high in magnitude, especially in the case of very deep neural networks with millions of parameters, leading to computational issues [32].
- They add non-linearity. A NN without an activation function would not be able to combine complex (nonlinear) patterns and make predictions from the data; it would just be a series of linear transformations on the inputs using the weights and biases [33].

2.8.1 Activation functions' criteria

A good activation function fulfills as many as possible of the following criteria:

1. **Be differentiable:** During the GD process in the NN (Section 2.5.1) the layers in the model need to be differentiable or at least differentiable in parts (ReLU has a non-differentiable point at 0, but this is trivial, since the point 0 will never be reached in most cases) to be able to calculate the gradient and move forward.
2. **Be computationally inexpensive:** Activation functions are applied after every layer, which means that they will be calculated multiple times during the training epochs. Therefore, every calculation must add an insignificant computational cost to the NN to avoid getting stuck.
3. **Be zero-centered:** The activation function's outputs should be symmetrical at zero so that the gradients do not shift to a particular direction [32].
4. **Not succumb to the vanishing gradient problem:** The GD process includes the backward propagation step, which is a chain rule to get the change in weights to reduce the loss after every epoch. Therefore, the derivative of an activation function in a layer results from the product of derivatives of activation functions in the path from the final layer to the current layer in a backward way - from the output layer to the input layer. The vanishing gradient problem emerges when the number of hidden layers increases. As this number increases, the partial derivative terms become smaller, or even zero, leading to much slower updates of the weights until they completely stop updating. An illustration of the Sigmoid activation function is presented in Figure 2.19.

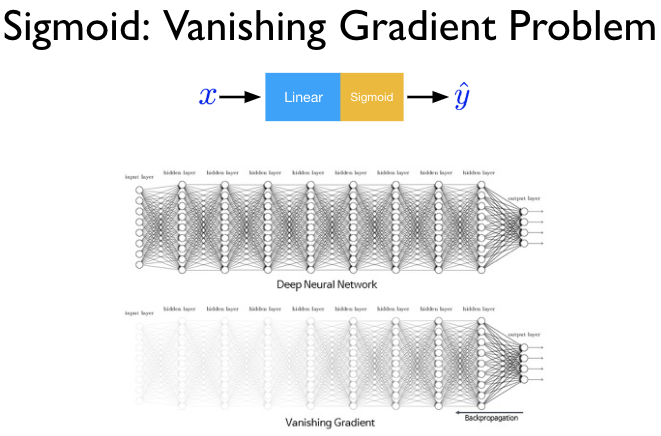


Figure 2.19: Vanishing gradient problem for the Sigmoid activation function [34]

2.8.2 Sigmoid activation function

The Sigmoid function was one of the first activation functions. Sigmoid succeeded in producing output values strictly in the range (0,1) and non-linearity. However, it only fulfils the differentiable criterion, as it is continuously differentiable as can be infered from Figure 2.20.

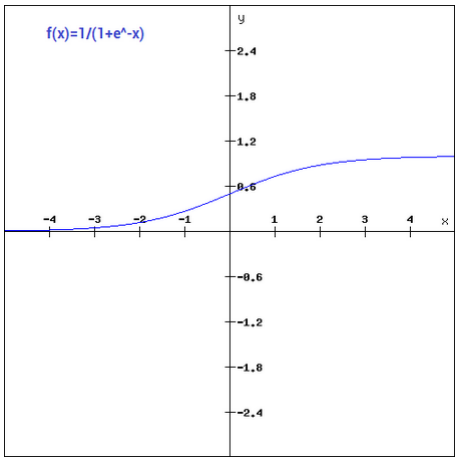


Figure 2.20: The Sigmoid activation function [33]

The derivative of this function is equal to (sigmoid(x)*(1-sigmoid(x)). As seen in Figure 2.21 the gradient values are significantly large in the middle (-3,3) but become increasingly smaller on the edge regions. That leads to very small gradients in these regions leading to the known vanishing gradient problem.

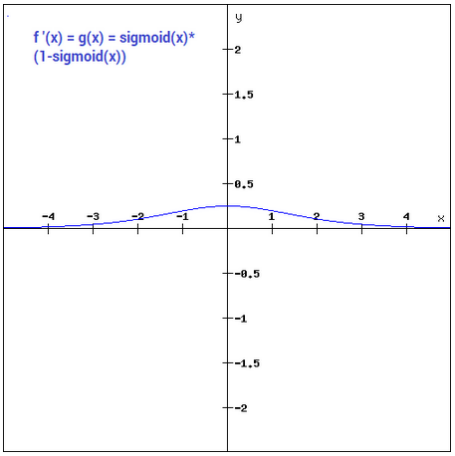
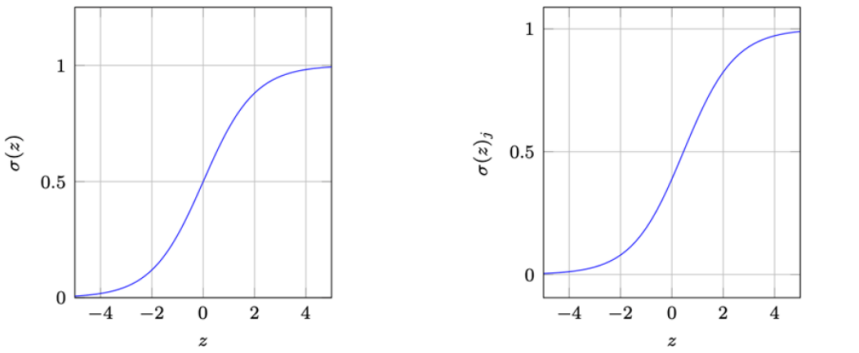


Figure 2.21: The Sigmoid activation function's derivative [33]

Unfortunately, Sigmoid is also both computationally costly and not zero-centered. Therefore, it remains relevant for historical reasons as the pioneer of the activation functions and is generally only used for binary classification problems[32].

2.8.3 Softmax activation function

The softmax is a more generalized form of the sigmoid and is usually preferred as the final layer in multi-class classification problems. The main advantage of softmax over sigmoid is that it assigns an output between 0 and 1 for each input vector, but with the specifications that the sum of all outputs will be 1 (probabilistic outputs). Although Sigmoid and Softmax appear identical in Figure 2.22 they are fundamentally different in the scores they produce.



(a) Sigmoid activation function. (b) Softmax activation function.

Figure 2.22: The Sigmoid and Softmax activation functions [35]

By comparing the scores that sigmoid and Softmax produce (Figures 2.23 and 2.24) we can clearly see that in both functions with increased input value, the score increase with upper limit 1. However, in Figure 2.23, the scores plateau with multiple values in the range of 0.9 to 0.99 (meaning that the sum will be much bigger than 1), while in Figure 2.24, higher input values produce high probability outputs, but the probabilities sum is maintained at 1.

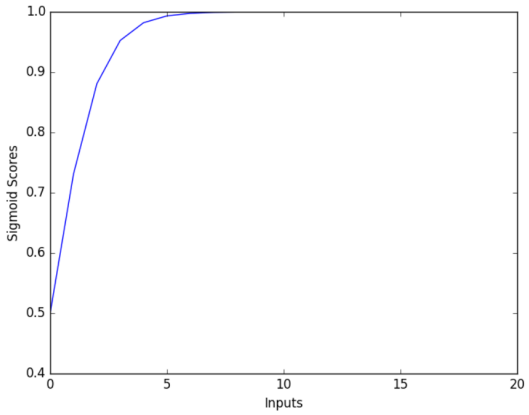


Figure 2.23: Sigmoid scores

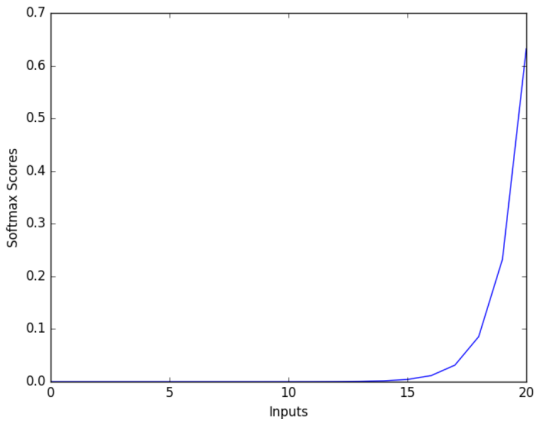


Figure 2.24: Softmax scores [36]

2.8.4 Tanh activation function

The tanh activation function was designed to solve the symmetry problem around zero. Tanh's outputs range from -1 to 1 (instead of 0 to 1), thus producing an output centered around zero, as is observable in Figure 2.25. Unfortunately, tanh shares all the other properties with Sigmoid by being continuous and differentiable at all points but also computationally costly and faces the vanishing gradient problem. For this reasons, it serves only as a marginal upgrade to the Sigmoid.

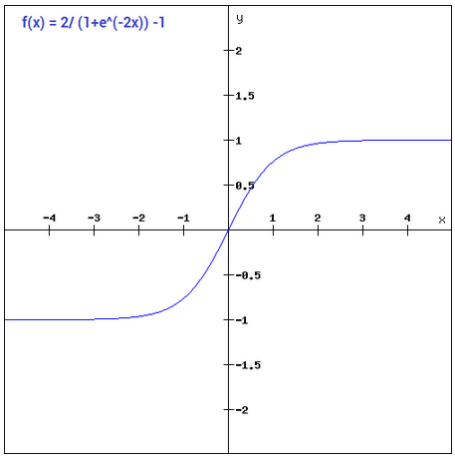


Figure 2.25: The Tanh activation function [33]

2.8.5 ReLU (Rectified Linear Unit) activation function

A shared problem between sigmoid and tanh activation functions is called saturation, which describes the tendency of large input values snapping to the upper limit and smaller input values to the lower limit in the range of output values (0-1 and -1-1, respectively). In addition, the functions are accurately sensitive to changes near the center of their input values (0.5 and 0, respectively). Unfortunately, limited sensitivity and saturation affect all the inputs regardless of the usefulness of the information they may contain. Saturation, in particular, significantly hinders both the learning process and the weights update, leading to poor performance of the model.

Ideally, a NN using SGD with backpropagation would require an activation function that could theoretically act like a linear function while still being a nonlinear function to facilitate learning of complex relationships in the data. The solution to this problem came from the Rectified Linear Unit activation function (ReLU), considered one of the few milestones in the deep learning revolution [37]. This led to ReLU rapidly becoming the default activation function for most types of NNs.

ReLU is very straightforward, as it either returns the exact input value provided if it is a positive number or returns zero if the input value is negative. Since ReLU is linear for all positive values, it encompasses all the desirable properties of a linear activation function (known as a piecewise linear function or a hinge function). At the same time, it fulfills the nonlinear function role for all negative input values. A significant advantage that arises from this duality is the selective neuron activation. In practice, the neurons will only be activated for positive input values. On the contrary, for all negative input values, the neuron does not get activated, as can be observed in Figure 2.26.

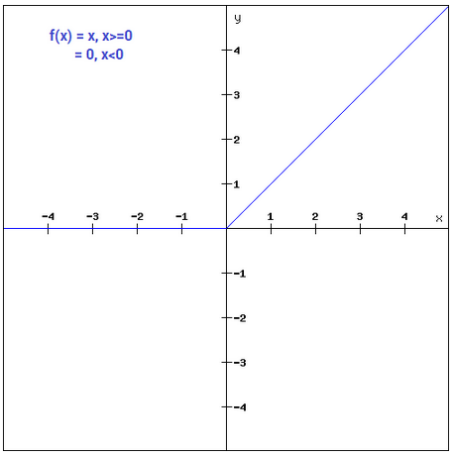


Figure 2.26: The ReLU activation function [33]

ReLU does not suffer from the Vanishing gradient problem. However, as can be seen in Figure 2.27, the gradient value is zero for all the negative inputs. This leads to an inability to update the weights and biases for those neurons during the backpropagation process. The accumulation of many instances creates dead neurons, known as the dying ReLU problem.

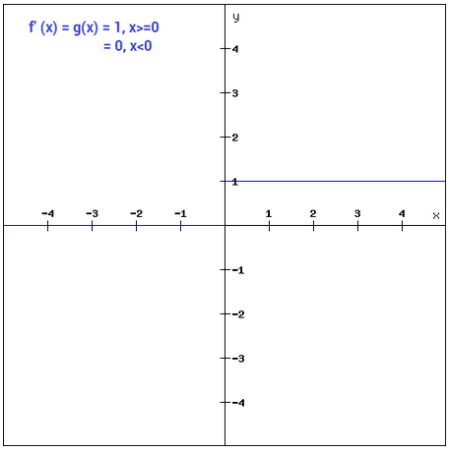


Figure 2.27: The ReLU activation function's derivative [33]

2.9 Motivation for selecting ReLU for this Thesis

The ReLU function is widely considered the default activation function for hidden layers in modern ML applications. This is mainly because of its simplicity and its long list of advantages over older activation functions:

1. **Computational Simplicity:** ReLu is surprisingly easy to understand and implement in a network, adding close to no simplicity by only requiring a max() function.
2. **Computational inexpensiveness:** The calculations are performed much faster with ReLU because no exponential terms are included in the function.
3. **Accelerated convergence:**The convergence of ReLU is estimated to be six times faster than sigmoid and tanh functions. This is due to ReLU's function fixed derivative for one linear component, and a zero derivative for the other linear component, leading to a faster learning process [38].
4. **Representational Sparsity:** In contrast to the tanh and sigmoid activation functions that are limited to approximating a zero output, ReLU is capable of outputting a true zero value. This, in turn, enables the activation of hidden layers in neural networks with one or more true zero values. This feature is called a sparse representation and is highly sought after in representational learning [37].
5. **Not succumb to the vanishing gradient problem:** As aforementioned, ReLU has no issues with the vanishing gradient problem.
6. **Mix of boundedness and unboundedness:** ReLU's duality creates a unique mix, where it is bounded to 0 for negative inputs but unbounded in the positive direction. This feature creates an inbuilt regularization and successfully leads to the training of deep multi-layered networks [39].
7. **Linear behavior:** ReLU looks and acts like a linear activation function, allowing for easier optimization of the NN.

* **Dying ReLU irrelevance:** This is not a general advantage of ReLU, but in this specific problem, all the input data are positive physical units. Therefore the main problem of the dying ReLU that results from negative inputs will not appear in any case.

In addition to its advantages, most of the sources that compared the different available activation functions recommend ReLU:

- *"Start with ReLU in your network... If you think the model has stopped learning, then you can replace it with a LeakyReLU... Activation functions work best in their default hyperparameters that are used in popular frameworks such as Tensorflow and Pytorch."* [32]
- *"The choice is made by considering the performance of the model or convergence of the loss function. Start with the ReLU activation function and if you have a dying ReLU problem, try leaky ReLU...In MLP and CNN neural network models, ReLU is the default activation function for hidden layers."* [38]
- *"Rectified Linear Unit is the rockstar of activation functions. This is the most widely used and a goto activation function for most types of problems....Use: CNN's, RNN's, and other deep neural networks.ReLU is a clear winner for Hidden layer activation functions."* [39]
- *"Easy and fast convergence of the network can be the first criterion.ReLU will be advantageous in terms of speed. You're gonna have to let the gradients die/vanish. It is usually used in intermediate layers rather than an output.....Use: CNN's, RNN's, and other deep neural networks.ReLU is a clear winner for Hidden layer activation functions."* [40]
- *"For hidden layers, a differential nonlinear function is more suitable because it trains the neural network model to work on more complex problems.Always use the ReLu function in hidden layers...If you're confused about which activation function to use, begin with the ReLu function as it's used in most scenarios these days."* [41]
- *"As we have seen above, the ReLU function is simple and it consists of no heavy computation as there is no complicated math. The model can, therefore, take less time to train or run. One more important property that we consider the advantage of using ReLU activation function is sparsity....The activations functions that were used mostly before ReLU such as sigmoid or tanh activation function saturated....ReLU, on the other hand, does not face this problem as its slope doesn't plateau, or "saturate," when the input gets large. Due to this reason models using ReLU activation function converge faster. "* [42]

Finally, in the evaluation of the performance of different activation functions presented in [40], the author collected the results of the validation and training accuracy and loss values of the samples for 20 epoch, that are illustrated in Figures 2.28, 2.29,2.30 and 2.31 respectively.

Figure 2.28: Validation accuracy

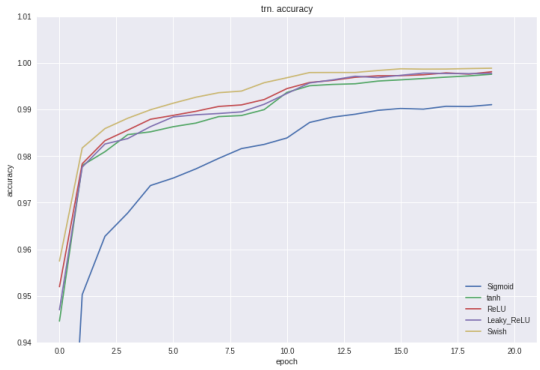


Figure 2.29: Training accuracy [40]

All 4 Figures are in agreement with the points mentioned above since ReLU excels in both training and validation accuracy and minimization of losses.

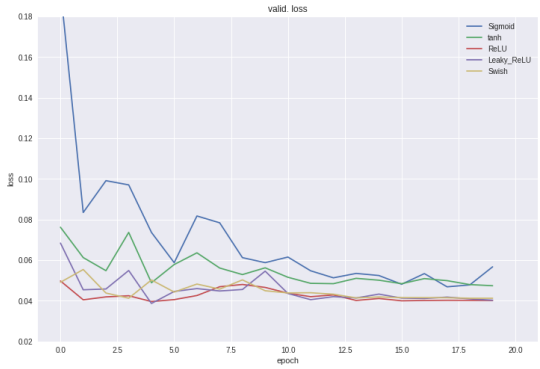


Figure 2.30: Validation Loss

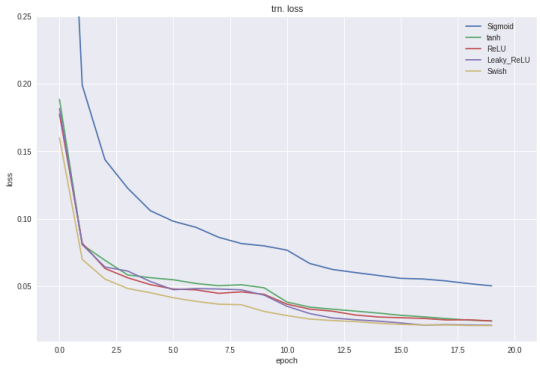


Figure 2.31: Training Loss [40]

In conclusion, the unanimous suggestion of all current literature towards ReLU, its advantages over other available activation functions, and the additional benefit of being unaffected by the dying ReLU problem by the nature of the task set ReLU as the optimal choice for this project.

2.10 Deep Learning

One of the most confusing problems in this Thesis came from the output requirements. Since the algorithm is designed to be able to predict two output values (Frequency and Decay ratio), there were two options:

1. Construction of two single output independent NNs, each tasked with predicting one of the two needed outputs **OR**
2. Multi-output regression requires a specialized machine learning algorithm capable of predicting two or more numerical variables.

Unfortunately, there has yet to be a clear consensus on which method is preferable. Popular examples of algorithms that can support multi-output regression are found in decision trees and ensembles of decision trees. However, both face limitations because the input and output data connections are blocky or highly structured based on the training data.

Alternatively, a NN can be created with the scope of supporting multi-output regression by specifying the number of target output variables in the output layer. The added benefit in this configuration lies in having the NN learn a continuous function capable of establishing useful relationships between input and output variables [43].

Deep learning (DL) NNs are built with the capacity to solve multi-output regression problems. The big difference in performance between traditional ML and DL algorithms is their scalability. While other ML techniques reach a plateau in performance, DL can continuously improve eternally with the increase in data availability, as can be observed in Figure 2.32.

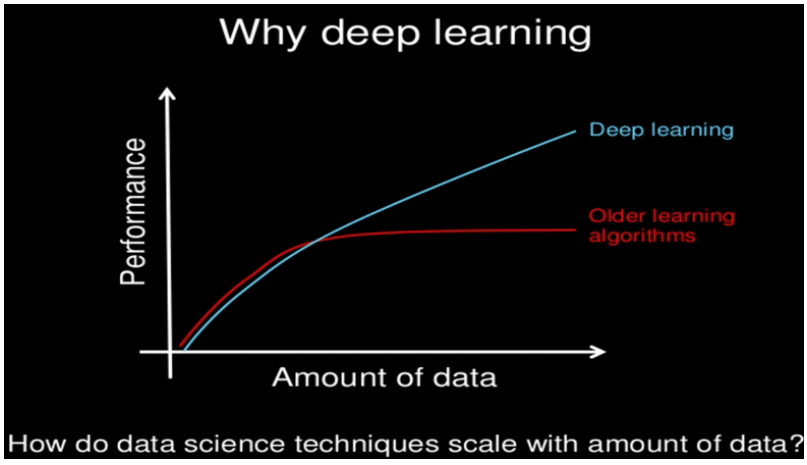


Figure 2.32: Why Deep Learning? Slide by Andrew Ng, all rights reserved. [44]

Although the data availability for this Thesis is quite limited to feed a DL NN properly, the most beneficial features of DL were explored and implemented in the model to facilitate updated versions of the algorithm for future updated databases and future work on multi-output regression. Therefore, the construction of two single-output NNs will be performed.

2.10.1 Sequential layers

In DL algorithms, multiple modules can be sequentially appended in each of the hidden layers. This leads to the model going through each of those modules during each passing through each hidden layer. The addition of every module in the sequential layer will increase the complexity and reduce the computational speed of the NN. Therefore it must fulfill a definite purpose to justify its addition. For this reason, the selected sequence for the sequential layers was the following:

1. **Linear regression:** Establishes linear relationship between a dependent variable and the other given independent variables [45].
2. **Activation function:** Introduces non-linearity by applying ReLU to the Linear regression's results and enables learning (Section 2.8.5).
3. **Batch Normalization (BN):** Normalizes the activation values to reduce the variance of the hidden representation. This process aims to reduce all the activation values to the same scale, resulting in improved training speed [46]. BN follows Equation (2):

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \varepsilon}} * \gamma + \beta \tag{2}$$

   By the Pytorch instrumentation manual [47]: *"The mean and standard-deviation are calculated per-dimension over the mini-batches and γ, and β are learnable parameter vectors of size C (where C is the number of features or channels of the input)."*
4. **Dropout:** Deactivates some of the NN's neurons with the help of Bernoulli distribution to avoid the problem of overfitting [46]. This problem refers to the model finding shortcuts to perfectly navigate through the training data but failing to perform well on a hold-out sample. Overfitting impacts negatively on the NN's ability to efficiently learn and then generalize to unfamiliar data [48].

2.10.2 Batch Normalization

As aforementioned, every module in the sequential layer must fulfill a definite purpose to justify its addition. In Figure 2.33, the distributions of the output values from a specific layer across two networks (with and without BN) are presented:

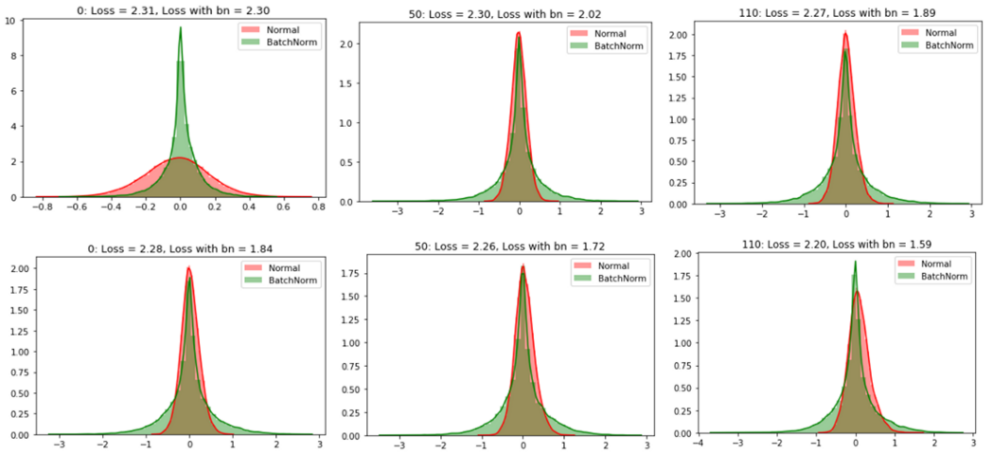


Figure 2.33: First epoch (first row) and second epoch (second row) BN comparison [46]

As can be inferred from Figure 2.33, BN significantly influences the distribution. In the NN without BN the distribution of inputs within each epoch fluctuates wildly, while being almost constant in the NN with BN. This affects the training speed and reduces the number of epochs since all activation values will be comparable in scale.

Additionally, the loss of a NN with BN follows a sharper and steeper reduction of the loss function, as illustrated in Figure 2.34.This arises from BN allowing to implement a higher LR without the fear of compromising convergence, leading to a faster and better converge since more local minima can be avoided.

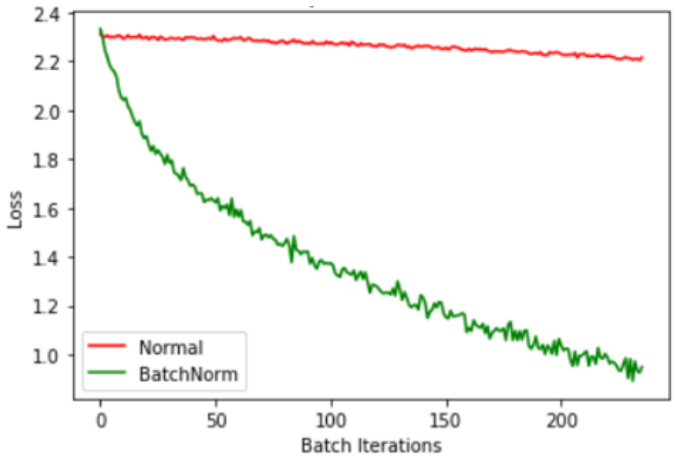


Figure 2.34: Loss function Batch Normalization comparison [46]

2.10.3 Dropout

An NN must perform consistently well on the training dataset and unknown data to make accurate predictions. The learning process is based on establishing patterns from known examples and using them to generalize the knowledge to unknown cases. However, there is a delicate balance in learning:

- **Underfit model:** If the learning is on the low side, the model will not be able to learn and be any more accurate than a randomizer both on the training dataset and on new data. Underfit models suffer from high bias and low variance.
- **Overfit model:** If the learning is on the high side, the model learns to mimic the optimal path for the training data but cannot perform in unseen examples or even statistical noise added to examples in the training dataset. Overfit models suffer from high variance, and low bias [48].

In both cases, the model lacks the ability to generalize and learn efficiently, and the predictions have no value, as illustrated in Figure 2.35:

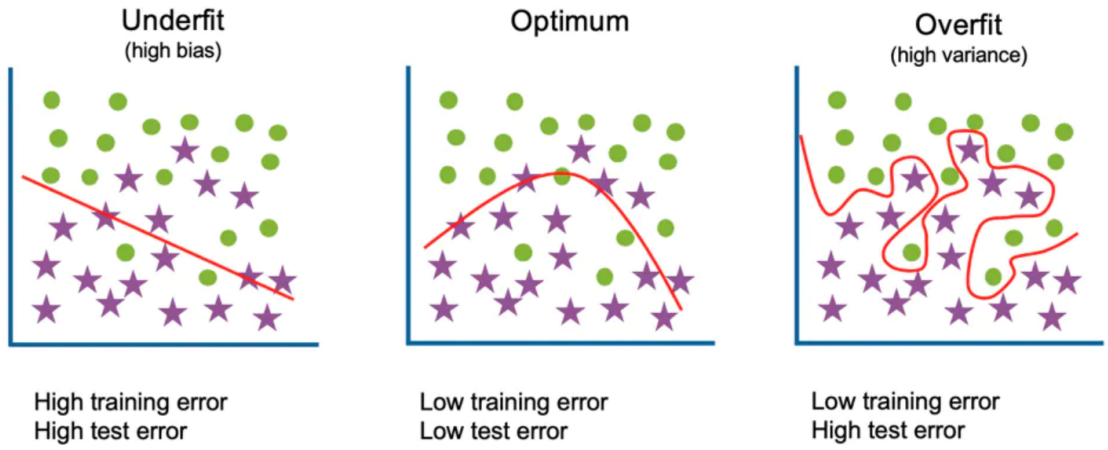


Figure 2.35: Underfitting, optimum fitting, and overfitting with their characteristics [49]

Underfitting can be easily fixed by increasing the capacity of the model. The capacity of a model can be increased by a change in its structure, such as implementing additional layers or nodes.

Overfitting is a more complex problem and requires targeted techniques to fix it. These techniques, called regularization techniques, focus on reducing the generalization error by constraining the size of the weights. A commonly used regularization technique, Dropout, approximates the training of multiple NNs with different architectures in parallel.

Dropout acts as a switch during training, in the sense that during each iteration, it shuts down some of the nodes in each layer randomly. Therefore, each epoch deals with a NN with a different distribution of active nodes and connections between the layers. This effectively mimics the process of training the same dataset in many slightly different NNs and averaging the results. The process of Dropout on a simple NN can be observed in Figure 2.36.

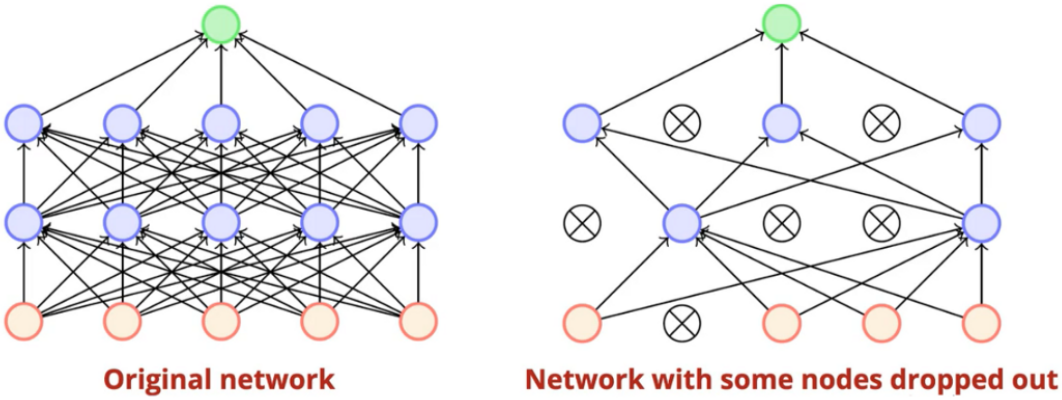


Figure 2.36: Visual representation of Dropout in a NN [46]

NNs without Dropout are prone to learning the noise associated with the data instead of generalizing. Specifically, as the number of epochs increases, the loss also increases. In contrast, Dropout forces the network to actually learn a sparse representation. This consequently leads to the NN learning only the more robust features that are dominant in all the different random subsets of the other neurons [50]. This can be observed in Figure 2.37, where the loss over the epochs is almost constant with Dropout implementation.

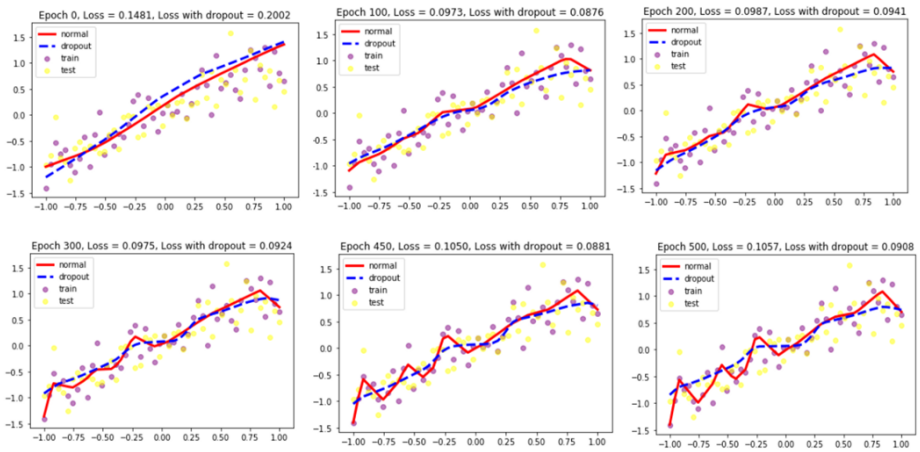


Figure 2.37: Loss comparison over epochs in a NN with and without Dropout [46]

Dropout is applied to each individual layer, and their outputs are, therefore, randomly subsampled. Thus, the network's capacity during training is reduced, and a wider network may be required. All regularization methods are particularly effective on problems with limited training data, where overfitting is more likely to occur. Therefore, in the case of this Thesis, where there is a shortage of case data, but the NN will be fairly large, Dropout appears as a necessary and strictly advantageous module.

2.11 Final model

In conclusion, the final Model selection, based on the collection of different literature sources on the current ML research and favored practices, will be comprised of:

- Adam as the Optimizer responsible for configuring the weights, biases, and LR in all hidden layers
- ReLU as the activation function for all hidden layers
- Sequential modules for each hidden layer, in order of Linear Regression, Activation function, Batch Normalization, and Dropout
- Two different NNs for each of the required output variables

A quick, colored overview of the model is presented in Figure 2.38:

Layer	Value	Specifics	Relevant algorithm
Input layer-Initial data	Input parameters	Axial Power Channel flow Burnup Axial void	Polca-T
Hidden layer-Optimizer	Weights (w)	Automatically adjusted	Adam
	Bias (b)	Automatically adjusted	Adam
	Learning rate	Automatically adjusted	Adam
Hidden layer- Sequential modules	Linear regression	i input neurons to j output neurons	(nn.Linear(n_in,i))
	Activation function	Applies ReLU	(nn.ReLU(inplace=True))
	Batch normalization	Applies BatchNorm1d	(nn.BatchNorm1d(i))
	Dropout	Applies Dropout	(nn.Dropout(p))
Output layer	2 different NNs for each output	Decay ratio Frequency	(Future work) multilayer perceptron (MLP)

Figure 2.38: Overview of the selected Algorithms and hyperparameters in the final model

3 Pytorch

3.1 Pytorch vs TensorFlow

PyTorch is an open-source, Python-based mathematical library and DL framework developed and maintained by Facebook and open-sourced on Github in 2017. Although it is built on Python, PyTorch also has a C++ and CUDA interface.

TensorFlow (TF) is an end-to-end open-source DL framework developed by Google and released in 2015. TF is a symbolic math library and has been praised for its multiple abstraction levels for building and training models

In the recent years of ML development, PyTorch and TF have dominated the selection among the available DL frameworks. Although both frameworks have many supporters, the debate over which framework is superior has yet to be settled, with each framework being superior in specific instances [51].

3.1.1 Tensors

Both Pytorch and TF utilize tensors for quick and efficient data handling. Tensors are arrays comprised of n-dimensions that store the structure of the necessary data. As can be observed in Figure 3.1, with the increase in dimensions, the size of the tensor increases, reaching multi-dimensional matrices.

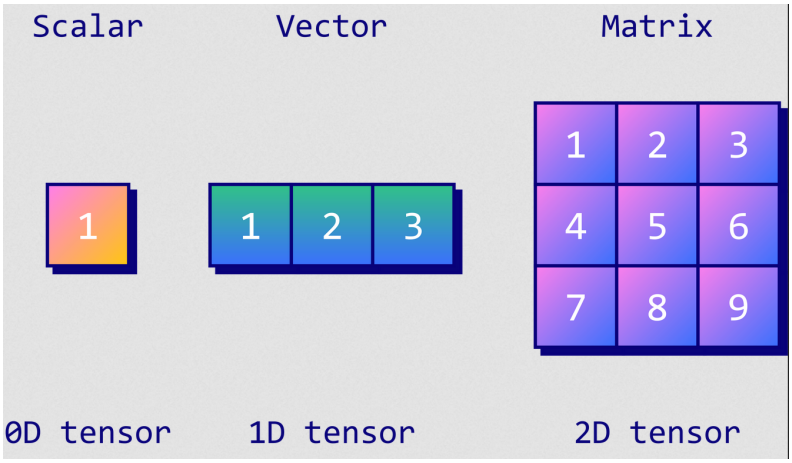


Figure 3.1: Tensors of different dimensions [52]

Tensors function similarly to NumPy's arrays but are designed to run both on GPU and CPU. That greatly increases the training and inference speed as the dimensions increase. For 4D matrices multiplications, tensors complete the task 52 times faster than arrays, making them irreplaceable in ML and especially in DL [52].

3.1.2 Graph construction technique

The main distinction between the two frameworks lies in the computation graphs construction techniques. TF operates on a static graph concept, while Pytorch employs a dynamic approach. The term computational graph refers to a simplistic design that describes the order of computations in the NN as a directed graph, as illustrated in Figure 3.2. The computational graph enables parallelism or dependency-driving scheduling, resulting in a faster and more efficient training process.

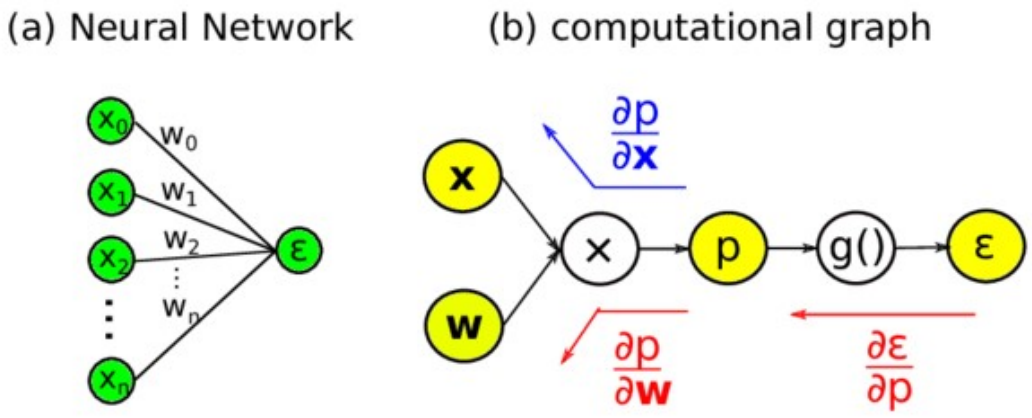


Figure 3.2: A (a) neural network and (b) its computational graph [53]

TF's static graph concept requires the user to define the computation graph of the model before being able to run the ML model. The computational graph is generated statically and requires compilation before its execution.TF implemented dynamic graphs with the addition of the TF Fold library, but PyTorch is contracted organically around the dynamic graph approach.

PyTorch operates by directly interpreting the correspondence between each line of code and its equivalent computational graph area. The implementation directly on Python allows for an automatic alteration of the graphs with every modification to the code, without the user needing to provide special session interfaces or placeholders.

In conclusion, Pytorch's dynamic graph creation process follows the coding logic of Python, giving it a more familiar feeling. On the contrary, TF requires a more complicated coding process, thus adding complexity to it [54].

3.1.3 Distributed training

Another significant advantage PyTorch offers over TF is the in-built data parallelism. Data-parallel training is one of the two types of distributed training, the other being model-parallel training as illustrated in Figure 3.3. Among the two types of training, data-parallel training is the most efficient way of training DL models and the most common practice in all frameworks. With data-parallel training, the entirety of the dataset is divided into a number of subsets arising from the number of nodes. This requires the correct intercommunication between nodes during training to ensure constant sync between them. Data-parallel training is the most efficient way of training DL models and the most common practice [55].

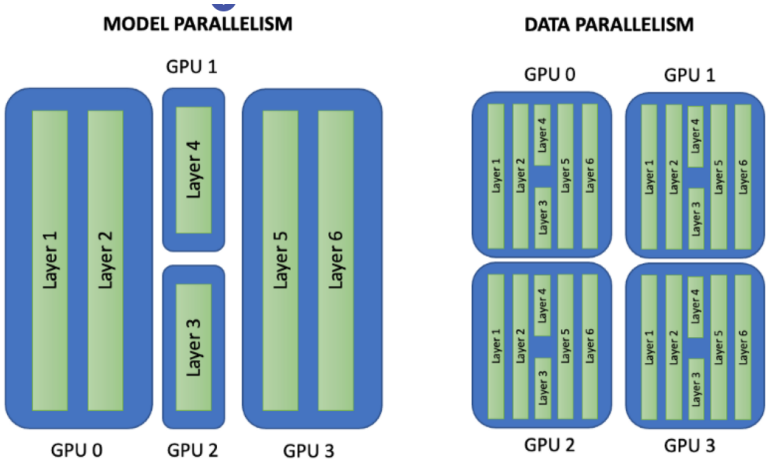


Figure 3.3: Distributed training model parallelism vs data parallelism [55]

PyTorch employs Pythons' implemented asynchronous execution to optimize performance in Data-parallel training. TF does not offer an automatic equivalent and requires manual coding and fine-tuning. However, with experienced and careful coding, all the automatic processes of Pythorch's data parallelism can be replicated in TF. Furthermore, the distributed training on Distributed TensorFlow can be accessed by using the tf.distribute API and selecting the preferred strategy.

3.1.4 Debugging

PyTorch offers great simplicity during debugging by relying on the standard python debugger.

TF's debugging process is more complicated, as the user can either learn to operate the TensorFlow debugger tool or request variables from the session [56].

3.1.5 Visualization

Visualization of the training process can provide great insight into the training process and allow for faster and more efficient debugging. TFlow's visualization library, named TensorBoard, offers a great variety of features:

1. Reliable tracking and visualization of the loss and accuracy of the model.
2. Visualization of the computational graph.
3. Histograms overseeing the changes of weights, biases, or other tensors through the epochs.
4. Display capabilities for images, text, and audio data.
5. Profiling of different TF programs [54].

Pytorch has yet to develop an equally helpful Visualization library but has integrated Tensorboard for its many advantages, which will also be used for this Thesis.

3.1.6 Production deployment

TF allows the user to directly deploy trained models to production by using TensorFlow serving, its built-in high-level API server.

PyTorch does not provide any framework to deploy models directly onto the web but requires an external API back-end server, like Flask or Django. However, most users prefer TensorFlow serving not to compromise performance.

3.1.7 Learning curve and community

TF has established itself over the years among industry professionals due to its optimal visualization, highly modifiable framework, and customizable features suitable for high-level model development. Thus, many established projects, detailed instruction manuals, and an engaged community of users with many years of experience are readily available. However, it is still a complicated framework that requires careful coding, manual adjustments, and familiarization with its system to access its full potential. Additionally, the difficulty in debugging for beginners unfamiliar with its Debugging Tool and static graph concept requiring deep understanding before making minor adjustments to the model significantly increases the learning curve for ML beginners.

PyTorch has a considerably shorter history as a framework; however, the increase in community numbers in this short span is monumental. The great advantage of being built on Python and following its coding logic for all tasks have set it as the golden standard for ML and DL research. Pytorch has almost completely overtaken TF in current research papers thanks to its increased flexibility, short training duration, and debugging capabilities. As can be observed in Figure 3.4, while in 2018 Pytorch and TF seemed to have an equal split among papers, by the first quarter of 2021, 60% of all papers are implemented in PyTorch, with just 11% implemented in TF.

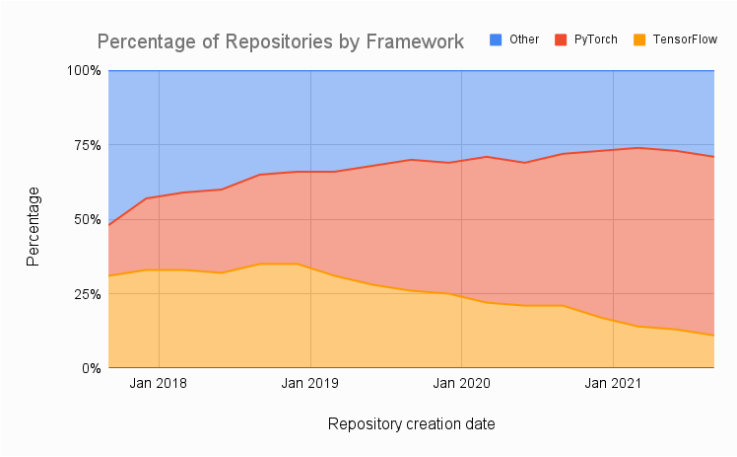


Figure 3.4: Percentage of papers using Pytorch vs. Tensorflow over the years [57]

3.2 Motivation for selecting Pytorch for this Thesis

In conclusion, there exists no clear consensus on which framework is superior, as both are excellent and capable of solving the vast majority of ML and DL problems. For this Thesis, Pytorch was selected as the model's framework for a variety of reasons:

1. Dynamic graph construction approach.
2. Unfamiliarity with TensorFlow debugger tool.
3. Visualization will utilize Tensorboard regardless.
4. No model deployment.
5. Learning curve simplicity and compatibility with Python coding.
6. Having emerged as the stronger choice for research papers.
7. Practical reasons on Westinghouse's part.

3.3 Model's life-cycle

3.3.1 Life-cycle in large scale industrial ML projects

An ML or DL project can range from fairly straightforward to highly complicated and is time and resource-consuming. In an ideal scenario, a model's life cycle would follow the simplistic process of Figure 3.5, and all projects would be successful in time.

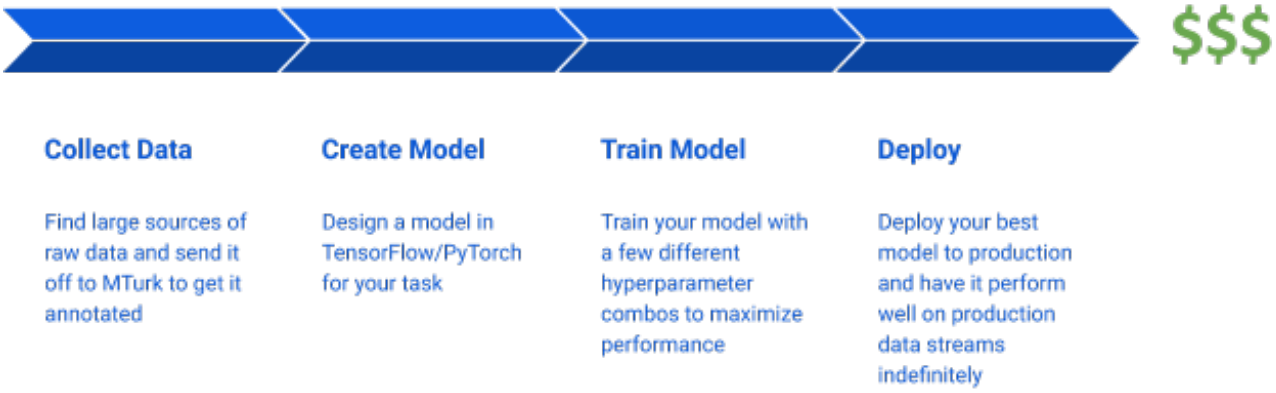


Figure 3.5: Naive assumption of the ML life cycle [58]

However, in reality, most complications and delays arise after the model has been designed and trained on the dataset during the evaluation period. The most significant advantage of ML, especially DL, is the model's dynamic interaction with the dataset, which does not stop after the design and training of the model. The hard work of an experienced ML developer lies in understanding the initial results of the model, evaluating mistakes or outliers in the learning process, and using those to refine both the model and the dataset.

Even the most successful and tested ML models that have been deployed for many years already by large companies like Google, Facebook etc. are constantly monitored and updated by teams of developers to eliminate arising biases, enrich them with newly collected sources of data, and enhance their capabilities to cover additional functions. As illustrated in Figure 3.6, the relationship between the dataset and the model is cyclical, and every improvement in one or the other restarts the evaluation process [58].

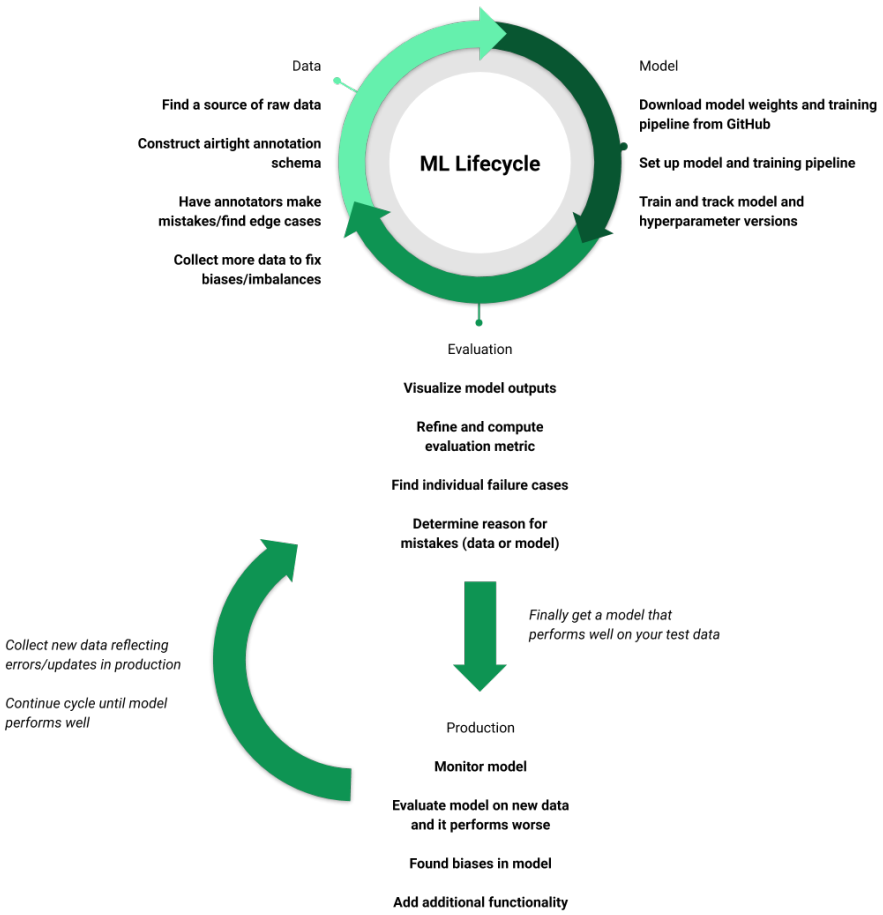


Figure 3.6: Realistic example of ML life cycle [58]

3.3.2 Thesis' model life-cycle

In the scope and constraints of this Thesis, the model's life cycle will be more linear, with potential updates, implementations of additional features, and enrichment with data from more burnup cycles from the reactor (or comparison with other reactors' cycles) left for follow-up projects.

The main steps that were conducted in the model life-cycle followed this chronological order, as proposed in [59] and will be discussed in detail in Chapters 4.3-4.6:

1. Data preparation
2. Model definition
3. Model Training
4. Model evaluation

4 Method

4.1 Thesis timeline

The timeline for this Thesis had to diverge many times from the original plan due to various unforeseen events and obstacles. However, despite some backtracking along the way, there were six distinct major stages, each with a different focus:

1. **Tutorial stage [Late January-Mid March]:** This period was the most unstructured of all since the familiarization with the concept of ML from the start required the combination of many different sources from various backgrounds and a lot of backtracking to understand the more complex terms.
2. **Simple NNs stage [Mid March-Late March]:** This relatively brief period marked the first actual experimentation with ML coding by trying to understand and rewrite simple ML problems of binary classification and simple regression.
3. **Final Model definition stage [Early April-Late May]:** This period consisted mainly of examining successful ML regression models for housing price prediction, comparing the different approaches, and selecting all the hyperparameters for the definition of this final model.
4. **Data preparation stage [Late May-Early July]:** This period was completely unscheduled on my part since the initial description of the project implied that the corrected dataset would be provided externally. I want to take this opportunity to thank Andreas Wikström for suggesting that it would be a great learning opportunity to include the preparation of the dataset from the raw flux distribution profiles in my work. Although it required a lot of effort and research, it gave me a holistic view of data processing.
5. **Thesis stage [Early July-Early September]:** This period was spent in its entirety on composing and editing the largest part of this paper. There was no possibility of working on the dataset and model during this period because of my delay in returning to Sweden, caused by the difficulty in securing housing and confidentially laws disallowing me to bring the dataset through customs.
6. **Model testing and conclusions stage [Early September-Mid October]:** The final period included finishing Data preparation, defining, training, and evaluating the model (Chapter 4), and using it for predictions (Chapter 5).

These six main stages do not directly coincide with the four steps in the life-cycle of a Model as described in Section 3.3.2. This is attributed to my inexperience with the material, leading to many fallbacks but is mainly indicative of the unpredictability of the learning process. Many of the steps could not be completed by themselves in any of the time periods since they required the combination of competencies gained from a later period. On the contrary, some of the steps did not require a period as they arose from knowledge gained in previous periods. The timeline was included to illustrate the process of compartmentalizing the different sub-tasks and this pyramid nature of the learning process, where all the seemingly unrelated knowledge building blocks merged to contribute to the final result.

4.2 Pre-work

Pre-work on any ML problem can vary in knowledge demand and time requirements based on the field, pre-existing knowledge in general coding and ML coding specifically, and the intricacies of ML concepts. In this case, I approached the subject without any familiarity with the ML or DL field, limited Python coding experience, and knowledge from my Master's studies on Nuclear energy engineering.

4.2.1 Theoretical familiarization

Delving into such a deep topic as ML required the combination of a variety of source materials from different professionals since understanding each concept generated questions on different sub-concepts. The most helpful materials in this quest were in combination with all the references in Chapter 2:

- 3Blue1Brown's playlist on Neural Networks [60], was an excellent starting point to gain a basic understanding of NNs, SGD, FP, BP, and their mathematical connections to regression.
- sentex's deep dive into ML concepts [61] proved to be too detailed for this Thesis's scope but invaluable for carefully studying the most important concepts.
- sentex's mini-series into DL with Pytorch [62] provided both an overview of how NNs on Pytorch are constructed with simple examples to experiment on (Section 4.2.2) and led to further research on DL advantages.

4.2.2 Practical familiarization

The research and understanding of the dynamics and intricacies of ML were so fascinating and rewarding that I would have undoubtedly spent much more time trying to get even more knowledge if not for the wise council of my Westinghouse supervisor Tobias Strömgren, that it was time to start ”getting our hands dirty with the task.”

Admittedly, the transition from reading about such complicated processes and the mathematical connections behind the algorithms to implementing them initially seems like an impossible task. However, I would like to devote these lines to ensure everyone that may get discouraged by it that there is an abundance of videos, implemented examples with code walk-throughs, and an endless database included in the Torchvision package. Pytorch and Tensorflow have been highly optimized over the years to automatize most tasks and significantly reduce the knowledge barrier. Additionally, the vast gap from simple NNs for basic tasks to extremely complex DL NNs can be tackled with many small steps over the learning process. I hope that the following explanation of the series of algorithms that I attempted while trying to reach the final model can serve as a small guide for one approach to this process. The complete algorithms with a brief explanation can be found in Appendix A:

1. **Basic classification ML problem:** A simplistic classification problem was proposed in most tutorials as the initial algorithm to understand ML basics.
2. **Simplistic version of a regression problem for housing market price prediction:** The problem of (static) price prediction on housing markets from seemingly unconnected variables is a good analogous to this Thesis' prediction model. Therefore, different approaches to that problem were examined.
3. **Complex version of a regression problem for housing market price prediction:** This solution required a much higher degree of understanding of the concepts of Sequential layers and encoding techniques and approached the problem of predicting a numerical value more holistically. Although some of the features were altered based on the needs of the Thesis (no need for categorical and continuous features split), it finally served as the backbone for the Model definition for this Thesis. Thus, the full code in Appendix A.3 is recommended as an excellent starting point for future work on this model.

4.3 Data preparation

Data preparation is one of the cornerstones of ML and requires much hard work. Data preparation centers around transforming a raw dataset into a more optimized form that can enable learning in the model. The philosophy revolves around finding a path to enable the model to discover the underlying structure of the problem better. The data engineer selects the ”environment” in which the model will operate and predict outputs that lead to the final discovery of the problem’s structure [63].

Data preparation’s complexity stems from the fact that each dataset is different and highly specific to the project and requires a clear understanding of the involved variables and connections. For this part, the expertise of Camilla Rotander on reactor dynamics proved invaluable to help me understand the complex relationships and patterns in the distribution profiles and specifically to highlight and evaluate the variables that are more influential and should be inserted into the model.

Combining the knowledge basis for the specific dataset with some general guidelines on efficient data processing can simplify the path to accurate results. The following steps were applied to this Thesis’ data preparation (full algorithms in Appendix B):

1. **Data Cleaning:** This step involves finding and correcting possible errors or null values in the data. Since all data were extracted directly from POLCA7’s distribution profiles, verifying the absence of errors or null values was easy.
2. **Feature Selection:** This is the most essential step in data preparation since correctly identifying the most influential input variables can significantly increase the convergence speed and accuracy. For this Thesis, it was decided that the input values that should be extracted from the distribution files should be the power, the void, the channel flow, and the burn-up distributions for each assembly of the reactor (700 channels in total), the average reactor power and the inlet temperature, adding up to a total of 2802 input variables for each case.
3. **Data Transformation:** This step required a lot of research and understanding of different data processing techniques. The techniques applied to the raw energy distribution files included separating string variables, selective string-to-number transforms, geometrical transformations, data selection and removal, label creation, and power scaling. The final input table for each case consisted of the 2802 variables in assembly order and adequately scaled.

4.4 Model definition

The model definition was the most vague part of the Method for this Thesis. This is attributed to the novelty nature of any ML project. Despite ML being a deep field with many existing models, each problem requires individual focus and understanding of its specific needs to define the exact model.

4.4.1 ML category

The model definition begins with translating the project's goal into ML concepts to understand the category of the model. Since the data and their correct labels will be extracted from POLCA7 (distributions) and POLCA-T (stability parameters), it is clear that the problem will follow supervised learning. The main categories for supervised ML models are shown in Figure 4.1 [64]:

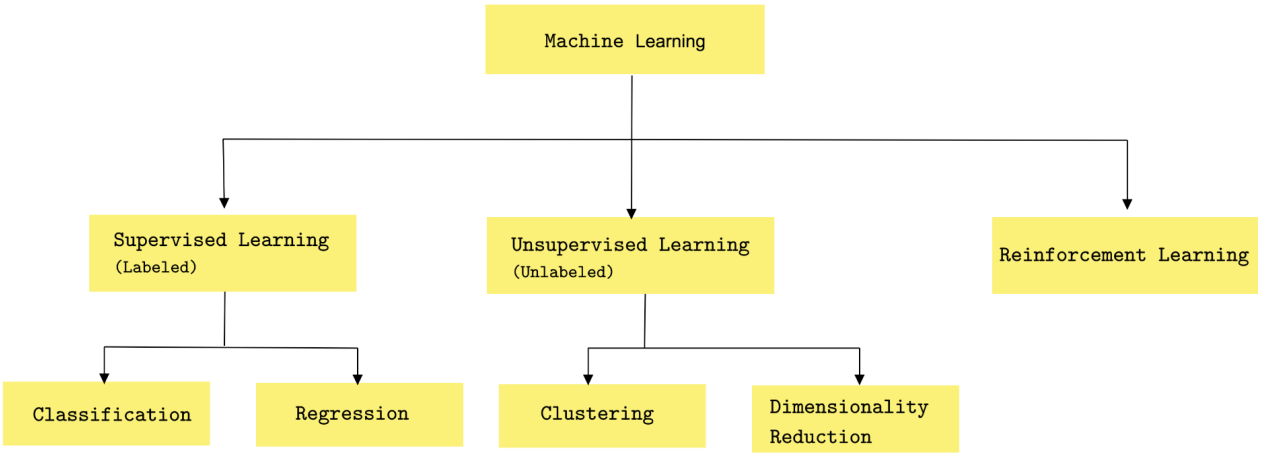


Figure 4.1: ML categories [65]

- **Regression Models:** In regression models, a mathematical relationship exists between the input and output variables (known or unknown). The goal of a regression model is to predict a single continuous output variable.
- **Classification Models:** In classification models, a finite number of options can be assigned to the output variable. The goal of a classification model is to predict strictly categorical variables, and it can be either:
  - **Binary classification Models:** Only two possible outputs
  - **Multiclass classification Models:** More than two possible outputs

Since the end goal of this project is to predict a numerical value from many numerical variables, the model for each output parameter will be a Regression model.

4.4.2 Model's algorithm construction

The final model (Section 2.11) resulted from many discussions and evaluations in brainstorming and discussion meetings with my supervisors. The considerations and motivations presented in Chapters 2 and 3 served as the guide to modify the algorithm from Appendix A.3 according to the specifications of this Thesis' problem.

Initially, the needed functions and libraries must be defined, as can be observed in Figure 4.2. All Data processing can be found in Appendix B, leading to the finished input dataset Merged.csv. After reading the dataset, the algorithm drops the columns that contain the two target outputs. At this point, the two models diverge, storing the output of interest into a new dataframe and then transforming it into a tensor. The processes are identical, except for the output parameter, thus the decay ratio model will be presented as the example. Creating a list of continuous features could have been omitted since all the input data are continuous. Regardless, it was included as an example for future versions of continuous and categorical features split.

```
import torch
import torch.nn as nn
import numpy as np
import pandas as pd
torch.set_printoptions(linewidth=120)
from torch.utils.tensorboard import SummaryWriter

#1) Data Preparation
df=pd.read_csv('C:/Users/dimit/Desktop/Thesis/Codes/Thesis Model/Inputs/Merged.csv')


X=df.drop(['Decay ratio','Frequency'],axis=1)
z=df.drop((X),axis=1)
z1=z.drop(['Frequency'],axis=1)
y=torch.tensor(z1.values,dtype=torch.float).reshape(-1,1)

#create continuous features
cont_features=[]
for i in X.columns:
        cont_features.append(i)

cont_values=np.stack([X[i].values for i in cont_features],axis=1)
cont_values=torch.tensor(cont_values,dtype=torch.float)
```

Figure 4.2: Data preparation Thesis model

The model is defined with the def__init, with arguments the model (*self*), the input (*n_cont*), the output (*y*) and the dropout ratio (*p=0.5*). The n_cont undergoes a BN before being inserted into the input layer (*n_in*). For any future work on multi-output regression techniques, the command (*layerlist.append(nn.Linear(layers[-1], y)*) is crucial for output definition (*y*). The forward definition consists of concatenating the continuous features to be inserted into the model. Finally, a manual seed of 100 is selected, and the NN is created, with initial input (*len(n_cont)*), one output variable (*y*), two Sequential hidden layers (one with 100 the other with 50 neurons) and the dropout ratio, as illustrated in Figure 4.3.

```
#2)creating the Neural Network

class FeedForwardNN(nn.Module):

    def __init__(self,  n_cont, y,  layers, p=0.5):
        super().__init__()

        self.bn_cont = nn.BatchNorm1d(n_cont)
        n_in =  n_cont
        layerlist = []

        for i in layers:
            layerlist.append(nn.Linear(n_in,i))
            layerlist.append(nn.ReLU(inplace=True))
            layerlist.append(nn.BatchNorm1d(i))
            layerlist.append(nn.Dropout(p))
            n_in = i
        layerlist.append(nn.Linear(layers[-1],y))

        self.layers = nn.Sequential(*layerlist)

    def forward(self,  x):
        x = self.bn_cont(x)
        x = torch.cat([x], 1)
        x = self.layers(x)
        return x

torch.manual_seed(100)
model=FeedForwardNN(len(cont_features),1,[100,50],p=0.4)
```

Figure 4.3: Thesis model creation

4.4.3 Model's visual representation

By utilizing Tensorboard, the whole NN was graphically depicted (Figure 4.4) to ensure that the NN abides by our initial specifications:

1. The NN follows the order input-Batch Normalization-Sequential Layers-output.
2. The input corresponds to the input dataset, as seen on the right side of Figure 4.4, which correctly states 650 cases*2802 input variables.
3. The output matches the target, as seen on the right side of Figure 4.4, which correctly states 650 cases*1 output variable.

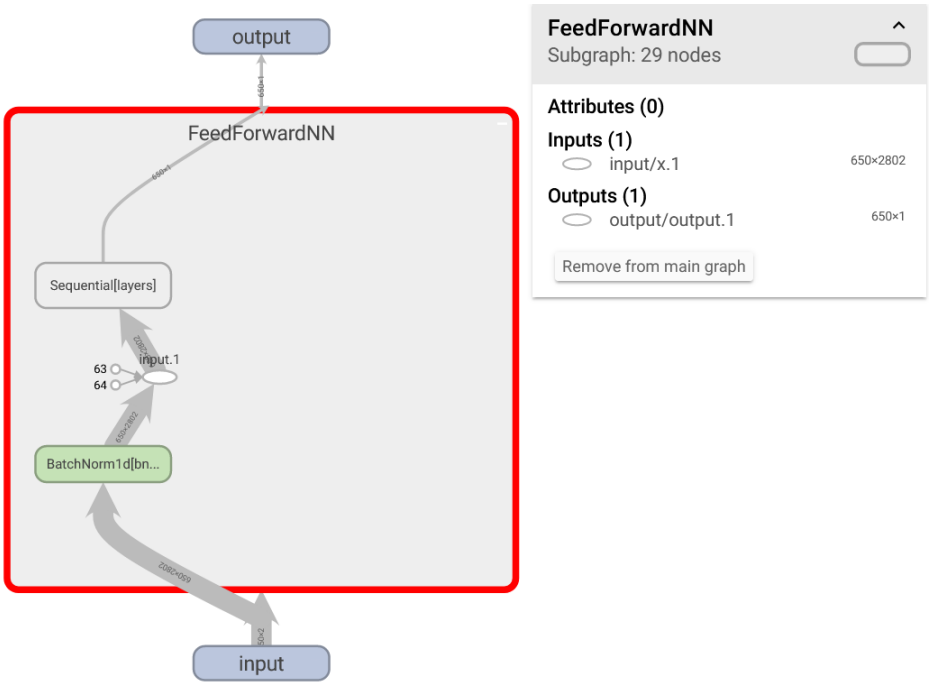


Figure 4.4: Thesis Model NN

Additional attention must be paid to the hidden Sequential layers, as illustrated and magnified in the following Figures, to verify that:

1. There exists two Sequential hidden layers, following the order Linear-ReLu-Batch Normalization-Dropout (Figure 4.5).
2. The first hidden layers have consistency in its layers input/outputs with 650 cases*100 neurons (Figure 4.6).
3. The second hidden layer has consistency in its layers input/outputs with 650 cases*50 neurons (Figure 4.7).

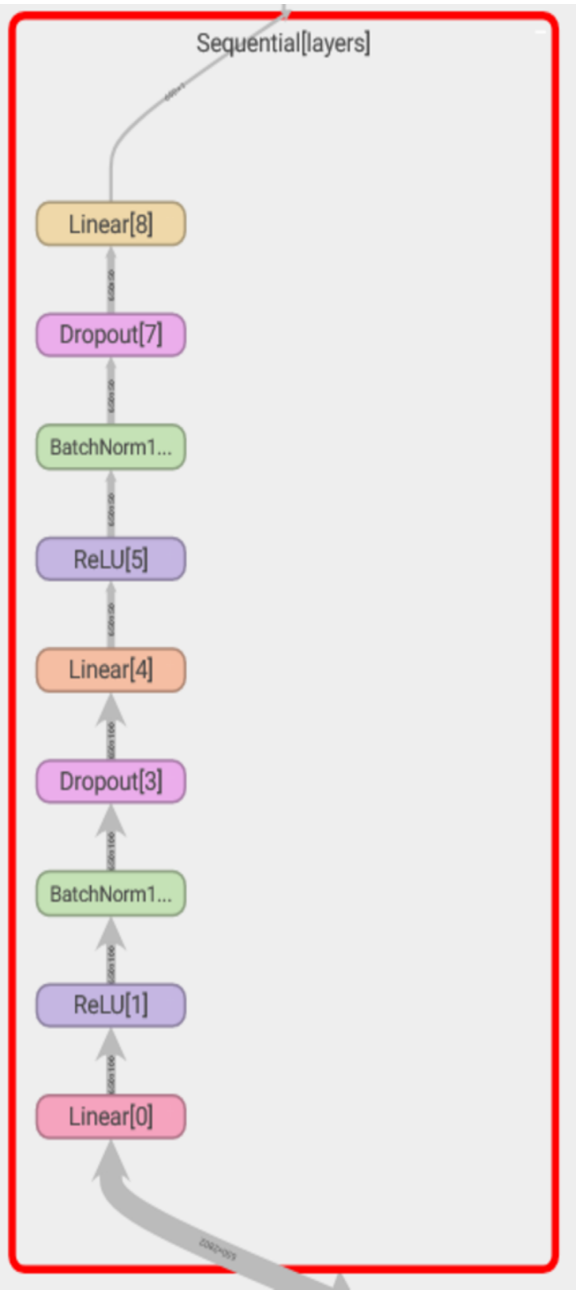


Figure 4.5: Thesis Model Sequential layers

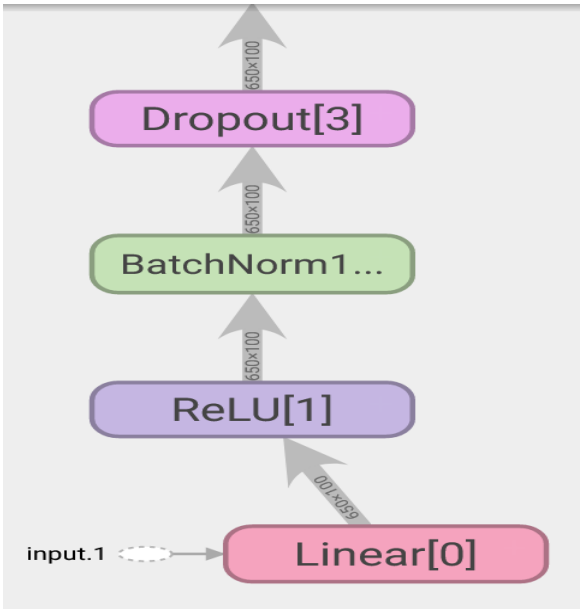


Figure 4.6: First Sequential layer

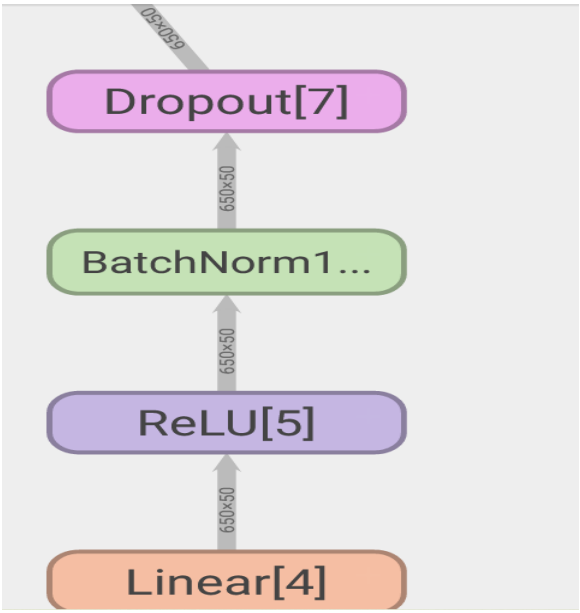


Figure 4.7: Second Sequential layer

4.5 Model training

In supervised ML, model training creates a mathematical representation of the relationship between the data features and a target label. The model's training is directly influenced by the quality of the data and the model's hyperparameters.

4.5.1 Test/train split

In ML, the input dataset is one of the most important resources available and needs to be allocated carefully. The training process consists of two parts of equal importance: testing and training. Both of these parts require a dataset distinct from one another. The common practice of training and validating the model with a single source of data is called the train-test split and involves splitting the initial dataset into two subsets.

The first subset's (training dataset) function trains the model to match the inputs to the correct outputs provided. Subsequently, the efficiency of the training process is tested by employing the input part of the second subset (test dataset) and requesting from the model to predict the unknown output. Then these predictions are compared to the known outputs of the test dataset, and the error is estimated. The train-test procedure requires a sufficiently large dataset to facilitate the split [66].

Discovering the optimal test/train split is a heavily discussed topic, with many sources suggesting adopting one of the common practice empirical split ratios [67]:

- Train: 80%, Test: 20%
- Train: 67%, Test: 33%
- Train: 50%, Test: 50%

To try and encompass a more scientific approach to estimating the optimal split, the practice suggested in [68] was followed:

1. Approximate Nu with the number of unique rows in the dataset: **Nu=650**
2. Apply **Nu** to Equation (3):

$$\gamma^* = \frac{1}{N_u^{1/4} + 1} \Rightarrow \boldsymbol{\gamma^* = 0.165} \tag{3}$$

3. Perform a split with ratio Train: 83,5%, Test: 16,5%

4.5.2 Epoch number

An epoch is defined as a full forward, and backward pass of the dataset through the NN [69]. As the number of epochs increases, the weights go in through many iterations, leading from underfitting to an optimal fit and reducing the training error. After this point, every additional epoch leads to overfitting and an increase in the testing error, as illustrated in Figure 4.8.

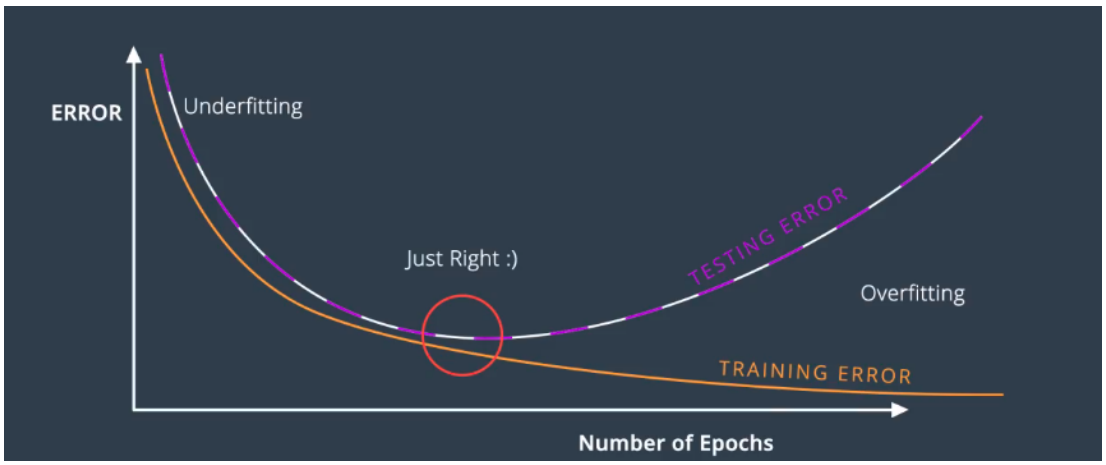


Figure 4.8: Epoch number selection [70]

No analytical formula exists to calculate the optimal number of epochs, as it diverges according to the database's specifications, like size and diversity. Therefore, a graphical evaluation of the specific model is advisable. The Tensorboard graph for the model's testing and training error over a large number of epochs (15000) indicates that the optimal was reached near 3000 epochs, as can be observed in Figure 4.9.

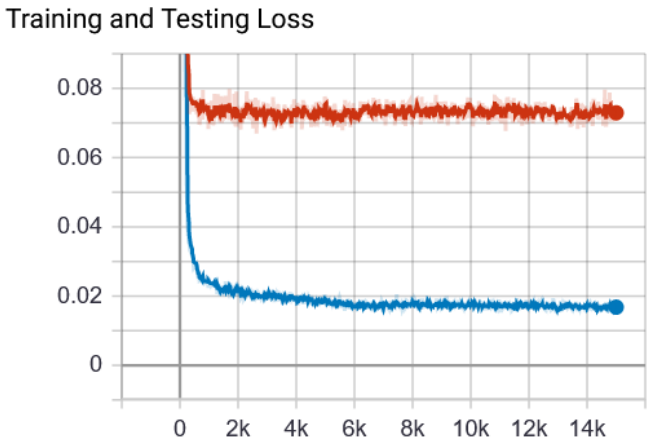


Figure 4.9: Testing and training error over a large number of epochs

4.5.3 Model's training algorithm

Initially, the loss function is defined as MSE (and converted to RMSE later) and Adam as the Optimizer. Additionally, the model's parameters are inserted into Adam, and the recommended LR=0.001 for Adam (Section 2.7). Because of the limited input dataset, a singular batch of 650 cases was created and split into test and train data for the input and output variables with a 0.165 split ratio, as established in Section 4.5.1. An empty list was created to store all the loss values (final_losses). The add_scalar command creates the train losses graph in Tensorboard. Finally, a message was scheduled every ten epochs to display the loss to verify its reduction over the training. The three last lines are necessary during every loop to perform BP and then reset the Optimizer's step. Figure 4.10 presents the code for the training loop.

```
#3) loss calculation and Adam optimizer
loss_function=nn.MSELoss()
optimizer=torch.optim.Adam(model.parameters(),lr=0.001)

#batches creation
batch_size=650 #(depends on cases)
test_size=int(batch_size*0.165)
X_train=cont_values[:batch_size-test_size]
X_test=cont_values[batch_size-test_size:batch_size]
y_train=y[:batch_size-test_size]
y_test=y[batch_size-test_size:batch_size]
print(len(X_train), len(X_test), len(y_train), len(y_test))

#epochs and training loss
epochs=3000
final_losses=[]
for i in range(epochs):
    i=i+1
    y_pred=model(X_train)
    train_loss=torch.sqrt(loss_function(y_pred,y_train)) ### RMSE
    final_losses.append(train_loss)
    tb.add_scalar("Training Loss", train_loss, i)
    if i%10==1:
        print("Epoch number: {} and the Loss : {}".format(i,train_loss.item()))
    optimizer.zero_grad()
    train_loss.backward()
    optimizer.step()
```

Figure 4.10: Thesis' training loop for 3000 epochs

The Tensorboard graphs for the Training loss over the 3000 epochs for both output variables can be observed in Figures 4.11 and 4.12, respectively:

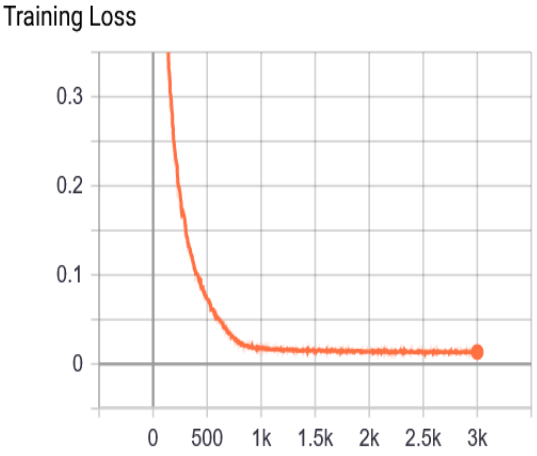


Figure 4.11: Frequency training loss

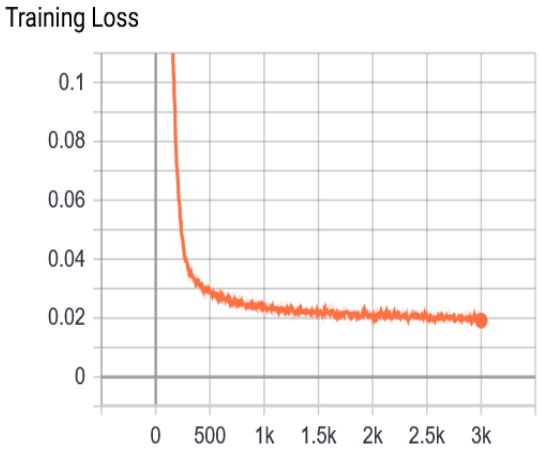


Figure 4.12: Decay ratio training loss

4.6 Model evaluation

4.6.1 Regression metrics

ML models are evaluated on different metrics to monitor the learning, fine-tune the hyperparameters, and approach a better result. Although accuracy is usually tied to model assessment, it can only be used in classification problems. The performance of regression NNs is derived in conjunction with the error between the target values and their predictions. The three most commonly used metrics for regression are [71]:

1. **Mean Absolute Error (MAE):** MAE centers around calculating the absolute Difference between actual and predicted values. MAE's units are the same as the target output. Additionally, MAE's changes are linear (the same weight applies to all errors). Therefore, MAE is a more direct representation of the sum of errors. The calculation of MAE results from Equation 4 [72]:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{4}$$

Advantages	Disadvantages
same unit as the output variable	non-differentiable graph
most robust metric to outliers	need Optimizers to be differentiable

Table 4.1: MAE Advantages and Disadvantages [73]

2. **Mean Squared Error (MSE)/ Root Mean Squared Error (RMSE):** MSE calculates the squared Difference (deviation) between the predicted and true target values. The squaring magnifies the larger errors, leading to the inflation of the average error score. RMSE is simply the square root of MSE and is used more often because MSE values can become too magnified. Additionally, RMSE has the same units as the target, allowing for easier interpretation. RMSE follows Equation 5 [72]:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{5}$$

Advantages	Disadvantages
same unit as the output variable	not that robust to outliers
differentiable graph	increases in magnitude with the scale of the error

Table 4.2: RMSE Advantages and Disadvantages [73]

3. **R Squared/Adjusted R Squared:**

   R Squared score is different from the metrics as mentioned above, by measuring the model's performance of your model, instead of the loss. Essentially, it calculates the variability in the dependent variable can be explained by the model. R Squared is especially prone to overfitting for models with many independent variables. It is called R Squared because it is derived from the Correlation Coefficient(R) square. R Squared's value ranges from 0 to 1, with bigger values indicating a better fit between prediction and actual value, as calculated from Equation 6 [72]:

$$R^2 = 1 - \frac{SS_{Regression}}{SS_{Total}} = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \overline{y})^2} \tag{6}$$

Advantages	Disadvantages
measures the model's performance	prone to overfitting
helpful to explain the model's utility with a percentage	strictly improves with each new variable

Table 4.3: R Squared Advantages and Disadvantages [73]

   Adjusted R Squared is designed to fix the overfitting issue by penalizing newly added independent variables and keeping the metric from growing uncontrollably. Adjusted R Squared modifies R Squared with Equation 7 [73]:

$$R^2_{adj} = 1 - [(\frac{n-1}{n-k-1} * (1 - R^2)] \tag{7}$$

   , where n symbolizes the number of observations and k is the number of independent input variables. Thus, the adjusted R-squared adjusts for the number of terms in the model. Irrelevant input variables will decrease its value, and only good additions will improve its fit.

Since all metrics have their share of advantages and disadvantages, there needs to be a clear consensus on an optimal metric. The most popular metric seems to be the RMSE, therefore it will be included and evaluated for this Thesis' model. Possible future work could be the inclusion and comparison of additional metrics.

4.6.2 Model's testing algorithm

Initially, the gradient calculation needs to be disabled to have predictions. The model predictions for the test data are being called and stored as y_pred, and the RMSE is calculated. A dataframe was created with the Predictions, Test (correct output), and the Difference for all test cases, which will be presented and discussed for both output variables in the next Section 4.6.3. The add_scalar command creates the test losses graph and a combined graph of testing and training losses in Tensorboard and matplotlib to create the scalar plots (Section 4.6.3). Finally, the RMSE (4.6.3) was printed for the final loop. The code for the training is presented in Figure 4.13.

```
#testing
y_pred=""
with torch.no_grad():
    y_pred=model(X_test)
    test_loss=torch.sqrt(loss_function(y_pred,y_test))

data_verify=pd.DataFrame(y_test.tolist(),columns=["Test"])
data_predicted=pd.DataFrame(y_pred.tolist(),columns=["Prediction"])
final_output=pd.concat([data_verify,data_predicted],axis=1)
final_output['Difference']=final_output['Test']-final_output['Prediction']
#    tb.add_scalar("Testing Loss", test_loss, i)
#    tb.add_scalars('Training and Testing Loss', {"Training Loss" : train_loss,
                            # "Testing Loss" : test_loss}, i)
    # plt.figure(figsize=(10,10))
    # plt.scatter(y_test, y_pred, c='crimson')
    # p1 = max(max(y_pred), max(y_test))
    # p2 = min(min(y_pred), min(y_test))
    # plt.plot([p1, p2], [p1, p2], 'b-')
    # plt.xlabel('True Values', fontsize=15)
    # plt.ylabel('Predictions', fontsize=15)
    # plt.show()
print('RMSE: {}'.format(test_loss))
print (final_output)
# tb.close()


# model saving
torch.save(model.state_dict(),'ML_decay_ratio.pth')
```

Figure 4.13: Thesis' testing loop for 3000 epochs

The Tensorboard graphs for the Testing loss over the 3000 epochs for both output variables can be observed in Figures 4.14 and 4.15, respectively:

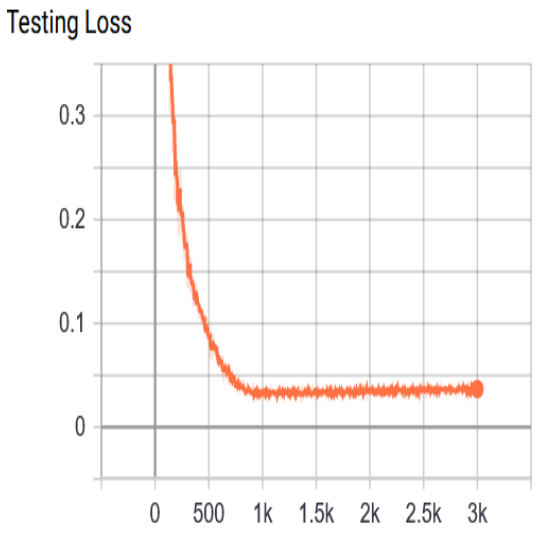


Figure 4.14: Frequency testing loss

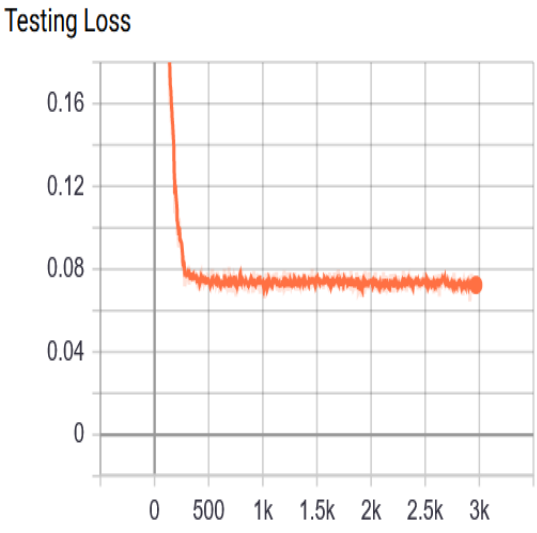


Figure 4.15: Decay ratio testing loss

4.6.3 Model's predictions for the testing dataset

The 0.165 split translates to 543 training cases and 107 testing cases for each model. The Predictions, Test (expected value), and the Difference for all cases are collected in Table 4.4, with the RMSE calculated by the algorithm for each of the outputs.

By comparing the RMSE, one can realize that the testing predictions for Frequency are much more reliable, with minimal outliers. Decay ratio has fourteen predictions that differ from the target output by more than 0,1. For Frequency, only seven predictions differ more than 0,05. To get a better image, the testing predictions for each target variable were plotted against the correct values in Figures 4.16 and 4.17:

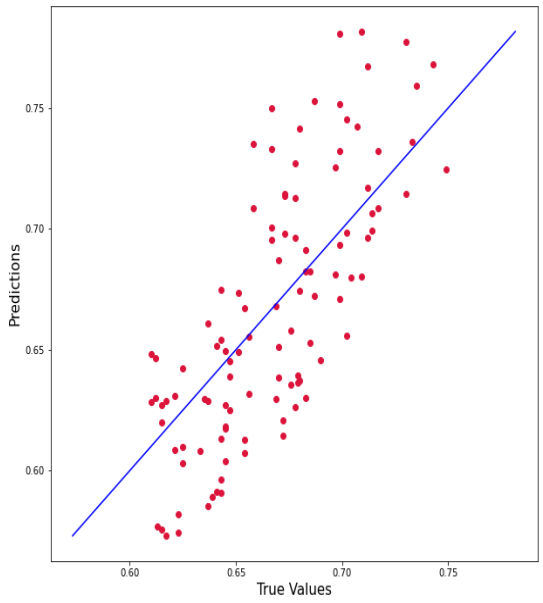


Figure 4.16: Frequency testing predictions against true values

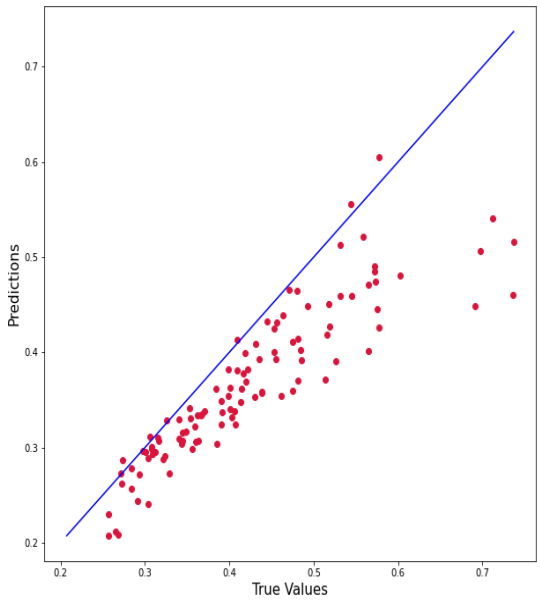


Figure 4.17: Decay ratio testing predictions against true values

By comparing the above Figures, an interesting observation can be made. Frequency seems to be very consistent for all the range of the target outputs, with some minor outliers for the highest frequency values. The trendline of the target values is almost perfectly cutting the scatter of the predictions in the middle.

Decay ratio, however, greatly varies with the values of the target outputs. The predictions for lower decay ratio values are very precise, but as the values increase, the divergence increases dramatically.

Summarizing, the testing predictions show that the model has been trained and produces logical predictions for the testing dataset. The next step is to save it and test its worth by requesting predictions for an unknown burn-up cycle (Chapter 5).

Table 4.4: Predictions for the testing dataset

	Decay Ratio				Frequency		
Index	Test	Prediction	Difference	Index	Test	Prediction	Difference
1	0.308	0.2985	0.0095	1	0.6540	0.6674	-0.0134
2	0.577	0.6049	-0.0279	2	0.6670	0.7498	-0.0828
3	0.445	0.4329	0.0121	3	0.6730	0.7138	-0.0408
4	0.362	0.3336	0.0284	4	0.6830	0.6825	0.0005
5	0.48	0.4643	0.0157	5	0.6870	0.7526	-0.0656
6	0.419	0.3994	0.0196	6	0.7090	0.7815	-0.0725
7	0.415	0.3616	0.0534	7	0.6100	0.6480	-0.0380
8	0.326	0.329	-0.003	8	0.6120	0.6467	-0.0347
9	0.273	0.2865	-0.0135	9	0.6150	0.6271	-0.0121
10	0.471	0.4658	0.0052	10	0.6370	0.6610	-0.0240
11	0.371	0.3385	0.0325	11	0.6430	0.6746	-0.0316
12	0.306	0.3115	-0.0055	12	0.6470	0.6390	0.0080
13	0.544	0.556	-0.012	13	0.6580	0.7349	-0.0769
14	0.431	0.4093	0.0217	14	0.6670	0.7004	-0.0334
15	0.354	0.3303	0.0237	15	0.6780	0.6961	-0.0181
16	0.464	0.4391	0.0249	16	0.6780	0.7129	-0.0349
17	0.409	0.4136	-0.0046	17	0.6990	0.7516	-0.0526
18	0.407	0.3242	0.0828	18	0.6100	0.6283	-0.0183
19	0.322	0.2882	0.0338	19	0.6120	0.6300	-0.0180
20	0.272	0.2615	0.0105	20	0.6150	0.6198	-0.0048
21	0.366	0.3336	0.0324	21	0.6410	0.6518	-0.0108
22	0.304	0.241	0.063	22	0.6430	0.6541	-0.0111
23	0.531	0.4593	0.0717	23	0.6580	0.7084	-0.0504
24	0.422	0.3823	0.0397	24	0.6670	0.6955	-0.0285
25	0.349	0.3167	0.0323	25	0.6730	0.7145	-0.0415
26	0.453	0.4255	0.0275	26	0.6780	0.7271	-0.0491
27	0.401	0.3624	0.0386	27	0.6990	0.7805	-0.0815
28	0.315	0.3099	0.0051	28	0.6170	0.6288	-0.0118
29	0.268	0.2081	0.0599	29	0.6210	0.6088	0.0122
30	0.359	0.3216	0.0374	30	0.6450	0.6496	-0.0046
31	0.298	0.296	0.002	31	0.6470	0.6451	0.0019
32	0.531	0.5132	0.0178	32	0.6670	0.7328	-0.0658
33	0.417	0.3777	0.0393	33	0.6730	0.6981	-0.0251
34	0.343	0.3036	0.0394	34	0.6830	0.6910	-0.0080
35	0.399	0.3819	0.0171	35	0.7070	0.7423	-0.0353
36	0.39	0.3487	0.0413	36	0.6250	0.6423	-0.0173
37	0.308	0.3004	0.0076	37	0.6210	0.6307	-0.0097
38	0.265	0.2121	0.0529	38	0.6250	0.6096	0.0154
39	0.456	0.4316	0.0244	39	0.6510	0.6737	-0.0227
40	0.356	0.2984	0.0576	40	0.6510	0.6492	0.0018
41	0.293	0.2721	0.0209	41	0.6560	0.6553	0.0007
42	0.545	0.4596	0.0854	42	0.6800	0.7416	-0.0616
43	0.34	0.309	0.031	43	0.6850	0.6822	0.0028
44	0.455	0.3925	0.0625	44	0.6970	0.7254	-0.0284
45	0.401	0.3402	0.0608	45	0.7120	0.7673	-0.0553
46	0.39	0.3247	0.0653	46	0.6370	0.6286	0.0084
47	0.301	0.2951	0.0059	47	0.6350	0.6297	0.0053
48	0.257	0.2298	0.0272	48	0.6330	0.6083	0.0247
49	0.484	0.402	0.082	49	0.6690	0.6681	0.0009
50	0.36	0.3057	0.0543	50	0.6700	0.6514	0.0186
51	0.284	0.2568	0.0272	51	0.6690	0.6297	0.0393
52	0.602	0.4811	0.1209	52	0.7020	0.7450	-0.0430
53	0.438	0.3581	0.0799	53	0.6990	0.7320	-0.0330
54	0.344	0.3072	0.0368	54	0.6990	0.6932	0.0058
55	0.406	0.3383	0.0677	55	0.7300	0.7774	-0.0474
56	0.309	0.2931	0.0159	56	0.6450	0.6271	0.0179
57	0.257	0.2073	0.0497	57	0.6410	0.5912	0.0498
58	0.385	0.3038	0.0812	58	0.6830	0.6299	0.0531
59	0.291	0.2441	0.0469	59	0.6800	0.6371	0.0429
60	0.691	0.4487	0.2423	60	0.7170	0.7086	0.0084
61	0.481	0.3707	0.1103	61	0.7140	0.7066	0.0074
62	0.363	0.3071	0.0559	62	0.7120	0.7168	-0.0048
63	0.526	0.3909	0.1351	63	0.7330	0.7361	-0.0031
64	0.43	0.353	0.077	64	0.7490	0.7247	0.0243
65	0.438	0.3579	0.0801	65	0.6470	0.6251	0.0219
66	0.329	0.2729	0.0561	66	0.6430	0.5908	0.0522
67	0.271	0.2732	-0.0022	67	0.6430	0.5962	0.0468
68	0.575	0.4455	0.1295	68	0.6760	0.6578	0.0182
69	0.403	0.3321	0.0709	69	0.6790	0.6396	0.0394
70	0.312	0.2955	0.0165	70	0.6790	0.6365	0.0425
71	0.736	0.4609	0.2751	71	0.7120	0.6963	0.0157
72	0.514	0.3708	0.1432	72	0.7090	0.6803	0.0287
73	0.565	0.4011	0.1639	73	0.7300	0.7143	0.0157
74	0.461	0.3544	0.1066	74	0.7430	0.7679	-0.0249
75	0.453	0.4005	0.0525	75	0.6430	0.6132	0.0298
76	0.344	0.3156	0.0284	76	0.6390	0.5890	0.0500
77	0.284	0.2784	0.0056	77	0.6370	0.5852	0.0518
78	0.565	0.4707	0.0943	78	0.6700	0.6383	0.0317
79	0.413	0.3475	0.0655	79	0.6720	0.6208	0.0512
80	0.324	0.2905	0.0335	80	0.6720	0.6147	0.0573
81	0.737	0.5165	0.2205	81	0.7020	0.6986	0.0034
82	0.519	0.4269	0.0921	82	0.6990	0.6710	0.0280
83	0.391	0.3369	0.0541	83	0.7020	0.6556	0.0464
84	0.577	0.426	0.151	84	0.7170	0.7322	-0.0152
85	0.475	0.3599	0.1151	85	0.7350	0.7591	-0.0241
86	0.475	0.4115	0.0635	86	0.6250	0.6032	0.0218
87	0.367	0.3351	0.0319	87	0.6230	0.5818	0.0412
88	0.304	0.2884	0.0156	88	0.6230	0.5744	0.0486
89	0.559	0.5214	0.0376	89	0.6540	0.6128	0.0412
90	0.42	0.3696	0.0504	90	0.6560	0.6319	0.0241
91	0.34	0.3297	0.0103	91	0.6540	0.6071	0.0469
92	0.712	0.541	0.171	92	0.6800	0.6744	0.0056
93	0.516	0.4183	0.0977	93	0.6850	0.6527	0.0323
94	0.399	0.3541	0.0449	94	0.6900	0.6458	0.0442
95	0.573	0.4743	0.0987	95	0.6970	0.6813	0.0157
96	0.481	0.4146	0.0664	96	0.7140	0.6991	0.0149
97	0.493	0.4483	0.0447	97	0.6170	0.5731	0.0439
98	0.384	0.3613	0.0227	98	0.6130	0.5770	0.0360
99	0.316	0.3072	0.0088	99	0.6150	0.5756	0.0394
100	0.572	0.4901	0.0819	100	0.6450	0.6176	0.0274
101	0.435	0.3925	0.0425	101	0.6450	0.6039	0.0411
102	0.353	0.341	0.012	102	0.6450	0.6182	0.0268
103	0.697	0.5062	0.1908	103	0.6700	0.6870	-0.0170
104	0.518	0.4509	0.0671	104	0.6760	0.6354	0.0406
105	0.409	0.3813	0.0277	105	0.6780	0.6263	0.0517
106	0.572	0.4845	0.0875	106	0.6870	0.6724	0.0146
107	0.485	0.3915	0.0935	107	0.7040	0.6799	0.0241
	RMSE=0.0767				RMSE=0.0354		

5 Results

5.1 Verification algorithms

As aforementioned, one of the complete cycles (c26) was separated from the others during Data preparation and kept hidden from the model to use as a Verification dataset for the trained model. Specifically, this cycle consisted of 82 steady states. Two new Python scripts were written to load the model, read this cycle's csv file and give predictions for each of the two target output variables.

The first part of loading a NN involves rewriting its definition, manual seed, and specifications (input features count, number of output variables, hidden layers, and Dropout ratio). The important distinction here is that since this script did not receive the initial dataset, all of the specifications must be in the form of numbers (not length). Then the model is called with model.load_state_dict(torchload(pth path)) and set into model.eval() mode.

Then, the csv of the cycle is read, and a dataframe without the target outputs is constructed. The predict function is created that converts a row of the dataframe (steady state) to tensors, asks the model to predict on that row, and converts it to a NumPy array. Finally, a loop is created to call the predict function for all rows and append the predictions in a list. This list, along with the true values and the difference, is inserted into a dataframe and saved on the Decay_ratio_Predictions.csv or Frequency_Predictions.csv , respectively. The complete Python script for the predictions can be followed in Figure 5.1.

```
df=pd.read_csv('C:/Users/dimit/Desktop/Thesis/Codes/Thesis Model/Inputs/final_c26.csv')
X=df.drop(['Decay ratio','Frequency'],axis=1)
z=df.drop((X),axis=1)
z1=z.drop(['Frequency'],axis=1)

def predict(row, model):
    # convert row to data
    row = Tensor([row])
    # make prediction
    yhat = model(row)
    # retrieve numpy array
    yhat = yhat.detach().numpy()
    return yhat

list=[]

with torch.no_grad():
    for i in range(0, len(X)):
        yhat = predict(X.iloc[i], model)
        list.append(yhat)

flatten = np.array(list).flatten()
df1 = pd.DataFrame(flatten, columns=['Predictions'])
df2 = pd.concat([df1, z1], axis=1, join='inner')
df2['Difference'] = df2['Predictions'] - df2['Decay ratio']

df2.to_csv('C:/Users/dimit/Desktop/Thesis/Codes/Thesis Model/Decay_ratio_Predictions.csv', index=False)
```

Figure 5.1: Thesis' predicting

5.2 Model's predictions for the Verification dataset

The Predictions, Test (expected value), and the Difference for the unknown cycle are collected in Table 5.1, with the RMSE for each of the outputs.

Similarly to the results of Section 4.6.3, the RMSEs still show that the predictions for Frequency are much more reliable for the evaluation dataset, with minimal outliers. However, the predictions are compelling for both output values: decay ratio has only four predictions that differ from the target output by more than 0,1. For Frequency, only five predictions differ more than 0,04. To get a better image, the testing predictions for each target variable were plotted against the correct values in Figures 5.2 and 5.3:

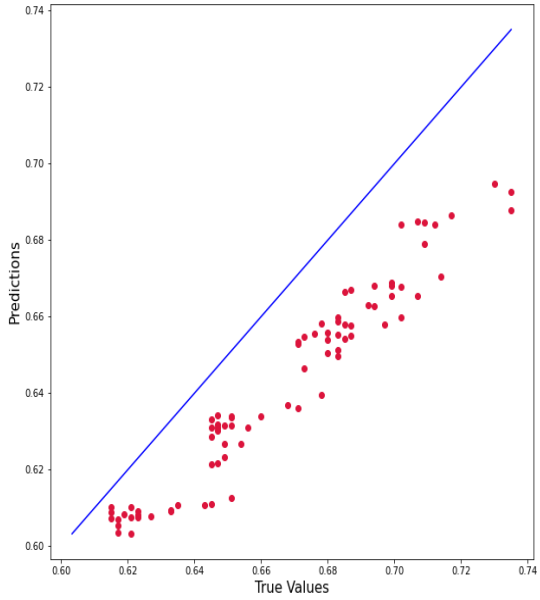


Figure 5.2: Frequency verification predictions against true values

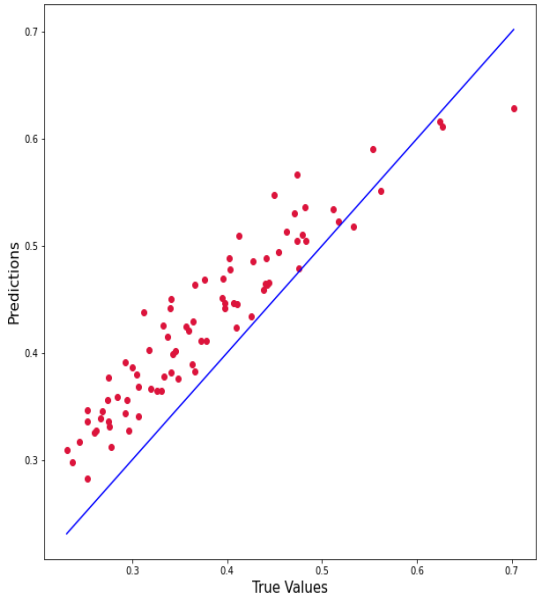


Figure 5.3: Decay ratio verification predictions against true values

By comparing the above Figures, the results seem to be opposite to those of Section 4.6.3. Frequency is the one that varies with the values of the target outputs. The predictions are more precise for lower frequency values, but as the values increase, the divergence increases slightly. That does not mean that Frequency diverges more; it remains more accurate than the decay ratio but is not that consistent for all values.

Decay ratio, however, seems to be very consistent for all the range of the target outputs, with close to zero outliers.

Table 5.1: Predictions for the unknown cycle

	Decay Ratio				Frequency		
Index	**Test**	**Prediction**	**Difference**	**Index**	**Test**	**Prediction**	**Difference**
1	0.3980	0.4469	0.0489	1	0.7300	0.6946	-0.0354
2	0.4540	0.4941	0.0401	2	0.6450	0.6214	-0.0236
3	0.3410	0.3815	0.0405	3	0.6470	0.6217	-0.0253
4	0.2760	0.3315	0.0555	4	0.6490	0.6232	-0.0258
5	0.4110	0.4458	0.0348	5	0.6730	0.6465	-0.0265
6	0.3260	0.3641	0.0381	6	0.6830	0.6498	-0.0332
7	0.6240	0.6157	-0.0083	7	0.6990	0.6654	-0.0336
8	0.4790	0.5108	0.0318	8	0.7020	0.6677	-0.0343
9	0.4400	0.4641	0.0241	9	0.7350	0.6926	-0.0424
10	0.4830	0.5045	0.0215	10	0.6430	0.6107	-0.0323
11	0.3630	0.3896	0.0266	11	0.6450	0.6109	-0.0341
12	0.2930	0.3432	0.0502	12	0.6510	0.6126	-0.0384
13	0.4380	0.4587	0.0207	13	0.6710	0.6362	-0.0348
14	0.3480	0.3756	0.0276	14	0.6780	0.6394	-0.0386
15	0.7020	0.6284	-0.0736	15	0.6970	0.6579	-0.0391
16	0.5170	0.5225	0.0055	16	0.7020	0.6599	-0.0421
17	0.4100	0.4233	0.0133	17	0.7070	0.6654	-0.0416
18	0.5620	0.5513	-0.0107	18	0.7140	0.6704	-0.0436
19	0.4760	0.4793	0.0033	19	0.7350	0.6878	-0.0472
20	0.4410	0.4881	0.0471	20	0.6330	0.6091	-0.0239
21	0.3340	0.3781	0.0441	21	0.6330	0.6093	-0.0237
22	0.2750	0.3363	0.0613	22	0.6350	0.6109	-0.0241
23	0.3980	0.4417	0.0437	23	0.6600	0.6339	-0.0261
24	0.3200	0.3667	0.0467	24	0.6680	0.6368	-0.0312
25	0.6270	0.6117	-0.0153	25	0.6800	0.6558	-0.0242
26	0.4740	0.5048	0.0308	26	0.6850	0.6579	-0.0271
27	0.3780	0.4109	0.0329	27	0.6940	0.6628	-0.0312
28	0.5120	0.5340	0.0220	28	0.6990	0.6690	-0.0300
29	0.4420	0.4641	0.0221	29	0.7170	0.6865	-0.0305
30	0.3960	0.4698	0.0738	30	0.6230	0.6075	-0.0155
31	0.3070	0.3679	0.0609	31	0.6210	0.6076	-0.0134
32	0.2620	0.3272	0.0652	32	0.6230	0.6092	-0.0138
33	0.4710	0.5301	0.0591	33	0.6470	0.6310	-0.0160
34	0.3600	0.4205	0.0605	34	0.6490	0.6316	-0.0174
35	0.2950	0.3564	0.0614	35	0.6510	0.6340	-0.0170
36	0.5540	0.5905	0.0365	36	0.6710	0.6534	-0.0176
37	0.4270	0.4856	0.0586	37	0.6760	0.6557	-0.0203
38	0.3430	0.3986	0.0556	38	0.6830	0.6597	-0.0233
39	0.4630	0.5130	0.0500	39	0.6870	0.6671	-0.0199
40	0.4070	0.4465	0.0395	40	0.7070	0.6849	-0.0221
41	0.3410	0.4504	0.1094	41	0.6170	0.6071	-0.0099
42	0.2740	0.3556	0.0816	42	0.6150	0.6072	-0.0078
43	0.2440	0.3165	0.0725	43	0.6150	0.6090	-0.0060
44	0.4120	0.5096	0.0976	44	0.6470	0.6303	-0.0167
45	0.3180	0.4025	0.0845	45	0.6450	0.6310	-0.0140
46	0.2690	0.3455	0.0765	46	0.6450	0.6331	-0.0119
47	0.4740	0.5662	0.0922	47	0.6710	0.6528	-0.0182
48	0.3660	0.4635	0.0975	48	0.6730	0.6546	-0.0184
49	0.3000	0.3862	0.0862	49	0.6780	0.6583	-0.0197
50	0.4020	0.4880	0.0860	50	0.6850	0.6665	-0.0185
51	0.3640	0.4295	0.0655	51	0.7020	0.6841	-0.0179
52	0.3120	0.4376	0.1256	52	0.6230	0.6082	-0.0148
53	0.2530	0.3466	0.0936	53	0.6190	0.6084	-0.0106
54	0.2310	0.3095	0.0785	54	0.6150	0.6103	-0.0047
55	0.2930	0.3916	0.0986	55	0.6470	0.6319	-0.0151
56	0.2530	0.3362	0.0832	56	0.6470	0.6341	-0.0129
57	0.4500	0.5479	0.0979	57	0.6800	0.6540	-0.0260
58	0.3400	0.4422	0.1022	58	0.6830	0.6554	-0.0276
59	0.2750	0.3764	0.1014	59	0.6830	0.6587	-0.0243
60	0.3760	0.4682	0.0922	60	0.6940	0.6682	-0.0258
61	0.3370	0.4150	0.0780	61	0.7090	0.6847	-0.0243
62	0.3330	0.4251	0.0921	62	0.6270	0.6079	-0.0191
63	0.2670	0.3384	0.0714	63	0.6230	0.6082	-0.0148
64	0.2370	0.2977	0.0607	64	0.6210	0.6103	-0.0107
65	0.4030	0.4782	0.0752	65	0.6560	0.6312	-0.0248
66	0.3050	0.3793	0.0743	66	0.6510	0.6314	-0.0196
67	0.2600	0.3254	0.0654	67	0.6510	0.6336	-0.0174
68	0.4820	0.5359	0.0539	68	0.6850	0.6541	-0.0309
69	0.3570	0.4243	0.0673	69	0.6870	0.6551	-0.0319
70	0.2840	0.3593	0.0753	70	0.6870	0.6576	-0.0294
71	0.3950	0.4512	0.0562	71	0.6990	0.6682	-0.0308
72	0.3460	0.4016	0.0556	72	0.7120	0.6842	-0.0278
73	0.3730	0.4110	0.0380	73	0.6210	0.6033	-0.0177
74	0.2960	0.3272	0.0312	74	0.6170	0.6035	-0.0135
75	0.2530	0.2827	0.0297	75	0.6170	0.6055	-0.0115
76	0.4440	0.4653	0.0213	76	0.6540	0.6269	-0.0271
77	0.3310	0.3641	0.0331	77	0.6490	0.6266	-0.0224
78	0.2780	0.3122	0.0342	78	0.6450	0.6286	-0.0164
79	0.5330	0.5176	-0.0154	79	0.6800	0.6505	-0.0295
80	0.3070	0.3404	0.0334	80	0.6830	0.6513	-0.0317
81	0.4250	0.4337	0.0087	81	0.6920	0.6629	-0.0291
82	0.3660	0.3822	0.0162	82	0.7090	0.6791	-0.0299
	RMSE=0.0615				**RMSE=0.0257**		

6 Conclusions

6.1 Lessons learned

Researching and working on this project taught me a variety of different theoretical and practical skills. However, the most essential lessons resulted from the errors, simplifications, and omissions I encountered in the later parts of the project. So, I would like to dedicate this Section to discuss and offer my suggestions on them.

- As mentioned in Section 2.7, current literature suggests that Adam with default parameters gets outperformed by SGD with Momentum with precise hyperparameters tuning, and the question of whether fine-tuned Adam gets outperformed by fine-tuned SGD is relative to training or validation. This was one of the simplifications attributed to a lack of experience rather than time. I cannot honestly suggest a method as it requires research on fine-tuning. My recommendation for such cases is to fix the pressing issues and allow simplifications when the time/knowledge investment is too high.
- The trial and error with two NNs and a single multi-output NN proved too demanding. While I remain interested in comparing the two strategies, two functional NNs that could predict the target were deemed preferable to restarting and possibly facing a dead end with multi-output regression.
- While graphically comparing the performance during the epochs in Section 4.5.2, the 6000 epoch mark seemed ideal. But after evaluating the saved model with the predictions for Chapter 5, I discovered that while Frequency gave great predictions, the model had overfitted for the Decay ratio and predicted some values near 22, two scales of power higher than the margin. Therefore, I had to backtrack and reexamine the epoch selection. At that point in the Thesis work, a complete examination for each output parameter would be very time-consuming, so I settled for 3000 epochs that seemed to be surprisingly accurate for Decay ratio and accurate enough for Frequency. I believe that Frequency would benefit from more seasons and recommend this examination as follow-up work.
- Before the testing stage of the model, I misjudged the importance of metrics and a graphical depiction of the model's performance. But Tensorboard and RMSE proved invaluable tools for the model's testing and prediction stages, and I am very glad I got at least a glimpse at them.

- Even the most basic decisions in ML are still largely experimental and unsolved. In my research, there was a multitude of problems, where most literature sources recommended trial and error to find what option best fits your specific model:
  - Optimizer fine-tuning
  - LR modification
  - multiple single output against a single multi-output model comparison
  - train/test split ratio
  - epoch number selection
  - regression metrics

  In my opinion, this exploratory nature of ML adds a lot of excitement to the process of discovering the model. Additionally, every programmer gets the opportunity to make personal choices without knowing the correct one, which adds a personal connection to your final model.
- The most popular ML libraries (Pytorch and Tensorflow) have been so optimized over their years of existence that coding and modifying your model requires much more of a logical understanding of the task and the solution process rather than advanced coding skills. Specifically, Pytorch follows Python's simplified and easy-to-understand logic so closely that after understanding the order of operations you want the model to follow, translating it to ML commands is trivial.
- Every problem offers many hidden opportunities for additional learning. Before the addition of Data processing to my tasks, I was just happy to receive a final version of the dataset to insert into the model. Data processing added at least two extra months of work to the project and many hopeless, frustrating nights. But if I had shied away from the opportunity or insisted on getting handed the processed version of the data, I would have missed out on a large and very rewarding part of the process of constructing an ML model.
- It is never too late to challenge your knowledge and delve into a new field. I was extremely anxious and reluctant to accept this Thesis project because my coding/Python background was practically nonexistent. I am very grateful to my KTH supervisor Walter Villanueva, for encouraging and motivating me to accept it and strive to succeed in this new field.

6.2 Future work

Solving this problem provided me with some insight and new ideas for follow-up work on this model or similar but more streamlined versions of it.

- An obvious addition would be to include datasets from various different reactors, separate them by type, construction year and discover similarities between them and possible improvements.
- The dataset that was used as an input to train and test the model consisted of only 650 cases in total. A common rule of thumb in ML suggests a minimum of cases equal to ten times the parameters of the model, resulting in 28020 cases for this model [74]. I would suggest continuing in the range of 20000-200000 cases, as suggested in [75]. However, the model was constructed with the principles of DL, resulting in no threshold to learning even with much larger datasets.
- As mentioned in Section 6.1, some of the techniques utilized were hindered by time or knowledge restraints, which led to simplifications. To improve the model's performance, each of those steps can be replicated with modifications or additional research on the subject. Specifically, the training and testing loss behavior for each output parameter needs to be evaluated in-depth, visually, and through the RMSE/overfitting for the training data. Finally, the construction and comparison with the multi-output model require significant research and testing.
- ML is intertwined with fine-tuning and model improvement. There are always optimizations and hyperparameter tuning that will lead to a better understanding and modification of the model's structure. Specifically, the comparison between the default parameters in Adam, fine-tuned SGD, and fine-tuned Adam is a large project that could reveal a lot about the model's learning process.
- In the evaluated model, some specific steady state distributions were examined for their known impact on core stability properties. In future work, it may prove fruitful to conduct a more detailed evaluation on the input variables selection to narrow down the parameters that are crucial to stability. For example, an uncertainty analysis could be performed to evaluate the impact of transient non-equilibrium, modeling the case of an instability event.

6.3 Final words

In conclusion, I found this project to be a great success in two different aspects:

1. The final model seems to be already quite efficient and reliable in predicting the target output features for full unknown burn-up cycles. The error margin during the design of the project was set to 0.1 for decay ratio and 0.05 for frequency. The models were able to predict the testing data with an RMSE of 0.0767 for decay ratio and 0.0354 for frequency, and the unknown cycle with an RMSE of 0.0615 for decay ratio and 0.0257 for frequency, respectively. In addition, this first version is burdened by multiple handicaps, like the limited dataset, time, and workforce constraints that did not allow it to optimally tune its many hyperparameters, utilize a variety of evaluation metrics and perform modification into the hidden layers. Fortunately, the bibliography and methodology have been researched and presented here at an acceptable depth to facilitate those improvements easily. In that spirit, the model was designed with those changes in mind by adopting ML and DL practices that encourage and help to introduce various improvements.
2. It was made clear both from the theoretical and practical work for this Thesis how useful advanced technologies for data handling and utilization can be in the field of Nuclear Energy. The long history and methodical data-keeping from the leading companies in the field are an enormous resource that can and has to be taken advantage of to keep improving the design and safe operation of older and newer reactors. I hope this Thesis can help and encourage more Nuclear Energy Engineers to delve into the fields of AI, ML, and DL and share my enthusiasm for the bright future that lies ahead by pushing the boundaries of technology.

References

[1] Auria, FD, Lombardi-Costa, A, and Bousbia-Salah, A. “Joint ICTP-IAEA Course on Natural Circulation Phenomena and Passive Safety Systems in Advanced Water Cooled Reactors”. In: (2010).

[2] U.S. NUCLEAR REGULATORY COMMISSION. *STANDARD REVIEW PLAN*. URL: https://www.nrc.gov/docs/ML0705/ML070550017.pdf. (accessed: 08/11 2022).

[3] Oak Ridge National Laboratory. *Density -Wave Instabilities in Boiling Water Reactors*. URL: https://www.osti.gov/servlets/purl/10183139. (accessed: 28/02 2023).

[4] Kruners, Magnus. “Analysis of instability event in Oskarshamn-3, Feb.8, 1998, with SIMULATE-3K”. In: (1998).

[5] T.H.J.J. van der Hagen, R. Zboray and Kruijf, W.J.M. de. “Questioning the use of the decay ratio in BWR stability monitoring”. In: (1999).

[6] SAS Institute. *Machine Learning What it is and why it matters*. URL: https://www.sas.com/en_us/insights/analytics/machine-learning.html. (accessed: 24/06 2022).

[7] SAS Institute. *Neural Networks What they are and why they matter*. URL: https://www.sas.com/en_us/insights/analytics/neural-networks.html. (accessed: 24/06 2022).

[8] Putra Sumari1, Saqib Jamal Syed, Laith Abualigah. *A Novel Deep Learning Pipeline Architecture based on CNN to Detect Covid-19 in Chest X-ray Images*. URL: https://www.researchgate.net/publication/351443166_A_Novel_Deep_Learning_Pipeline_Architecture_based_on_CNN_to_Detect_Covid-19_in_Chest_X-ray_Images. (accessed: 02/08 2022).

[9] Christoph Ostertag. *The Shortest Introduction To Deep Learning You Will Find On The Web*. URL: https://medium.com/analytics-vidhya/the-shortest-introduction-to-deep-learning-you-will-find-on-the-web-25a9975bbe1d. (accessed: 24/06 2022).

[10] RAM-AI. *RAM’s Machine learning: A new dimension to financial data*. URL: https://www.ram-ai.com/machine-learning/#machine. (accessed: 14/09 2022).

[11] Hollan Haule. *Understanding Error Backpropagation*. URL: https://towardsdatascience.com/error-backpropagation-5394d33ff49b. (accessed: 24/06 2022).

[12] James Moody. *What does RMSE really mean?* URL: https://towardsdatascience.com/what-does-rmse-really-mean-806b65f2e48e. (accessed: 14/09 2022).

[13] Simeon Kostadinov. *Understanding Backpropagation Algorithm*. URL: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd. (accessed: 14/09 2022).

[14] Sebastian Raschka. *Machine Learning FAQ:Can you give a visual explanation for the back propagation algorithm for neural networks?* URL: https://sebastianraschka.com/faq/docs/visual-backpropagation.html. (accessed: 14/09 2022).

[15] Jason Brownlee. *How to Configure the Learning Rate When Training Deep Learning Neural Networks*. URL: https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks. (accessed: 30/07 2022).

[16] Divakar Kapil. *Stochastic vs Batch Gradient Descent*. URL: https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1. (accessed: 02/08 2022).

[17] Jason Brownlee. *Understand the Impact of Learning Rate on Neural Network Performance*. URL: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks. (accessed: 30/07 2022).

[18] Sunil Patel. *Artificial Neural Network - Effect of Adaptive Learning Rate*. URL: https://snlpatel0012134.wixsite.com/thinking-machine/single-post/Artificial-Neural-Network-Effect-of-Adaptive-Learning-Rate. (accessed: 14/09 2022).

[19] Sanket Doshi. *Various Optimization Algorithms For Training Neural Network*. URL: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6. (accessed: 02/08 2022).

[20] Hemayet Ahmed Chowdhury, Md. Azizul Haque Imon,Anisur Rahman,Aisha Khatun, Md Saiful Islam. *A Continuous Space Neural Language Model for Bengali Language*. URL: https://www.researchgate.net/publication/338621083_A_Continuous_Space_Neural_Language_Model_for_Bengali_Language. (accessed: 03/08 2022).

[21] Ayush Gupta. *A Comprehensive Guide on Deep Learning Optimizers*. URL: https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/. (accessed: 06/08 2022).

[22] Kristy Ame Carpenter, David S Cohen,Juliet Jarrell, Xudong Huang. *Deep learning and virtual drug screening*. URL: https://www.researchgate.net/publication/328106221_Deep_learning_and_virtual_drug_screening. (accessed: 06/08 2022).

[23] Juan Du. *The Frontier of SGD and Its Variants in Machine Learning*. URL: https://iopscience.iop.org/article/10.1088/1742-6596/1229/1/012046. (accessed: 06/08 2022).

[24] Harsha Bommana. *Understanding Optimizers*. URL: https://medium.com/deep-learning-demystified/https-medium-com-deep-learning-demystified-understanding-optimizers-313b787a69fe. (accessed: 06/08 2022).

[25] Saket Thavanani. *Comparative Performance of Deep Learning Optimization Algorithms Using Numpy*. URL: https://towardsdatascience.com/comparative-performance-of-deep-learning-optimization-algorithms-using-numpy-24ce25c2f5e2. (accessed: 07/08 2022).

[26] Jason Brownlee. *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. (accessed: 08/08 2022).

[27] Sebastian Rude. *An overview of gradient descent optimization algorithms*. URL: https://arxiv.org/pdf/1609.04747.pdf. (accessed: 08/08 2022).

[28] Diederik P. Kingma, Jimmy Lei Ba. *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*. URL: https://arxiv.org/pdf/1412.6980.pdf. (accessed: 08/08 2022).

[29] Sieun Park. *A 2021 Guide to improving CNNs-Optimizers: Adam vs SGD*. URL: https://medium.com/geekculture/a-2021-guide-to-improving-cnns-optimizers-adam-vs-sgd-495848ac6008. (accessed: 08/08 2022).

[30] Philipp Wirth. *Which Optimizer should I use for my ML Project?* URL: https://www.lightly.ai/post/which-optimizer-should-i-use-for-my-machine-learning-project. (accessed: 08/08 2022).

[31] shaoanlu. *SGD > Adam?? Which One Is The Best Optimizer: Dogs-VS-Cats Toy Experiment*. URL: https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/. (accessed: 08/08 2022).

[32] Vandit Jain. *Everything you need to know about "Activation Functions" in Deep learning models*. URL: https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253. (accessed: 09/08 2022).

[33] Dishashree26 Gupta. *Fundamentals of Deep Learning – Activation Functions and When to Use Them?* URL: https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/. (accessed: 09/08 2022).

[34] Satish Varma. *Vanishing Gradient Problem*. URL: https://www.kaggle.com/getting-started/118228. (accessed: 09/08 2022).

[35] Nomidl. *Difference between Sigmoid and Softmax activation function?* URL: https://www.nomidl.com/deep-learning/what-is-the-difference-between-sigmoid-and-softmax-activation-function/. (accessed: 10/08 2022).

[36] Saimadhu Polamuri. *Difference Between Softmax Function and Sigmoid Function*. URL: https://dataaspirant.com/difference-between-softmax-function-and-sigmoid-function/. (accessed: 10/08 2022).

[37] Jason Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU)*. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/. (accessed: 12/08 2022).

[38] Rukshan Pramoditha. *How to Choose the Right Activation Function for Neural Networks*. URL: https://towardsdatascience.com/how-to-choose-the-right-activation-function-for-neural-networks-3941ff0e6f9c. (accessed: 12/08 2022).

[39] Adiamaan Keerthi. *Fantastic activation functions and when to use them*. URL: https://towardsdatascience.com/fantastic-activation-functions-and-when-to-use-them-481fe2bb2bde. (accessed: 12/08 2022).

[40] Ayyüce Kızrak. *Comparison of Activation Functions for Deep Neural Networks*. URL: https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-networks-706ac4284c8a. (accessed: 12/08 2022).

[41] Turing.com. *How to Choose an Activation Function For Deep Learning*. URL: https://www.turing.com/kb/how-to-choose-an-activation-function-for-deep-learning. (accessed: 12/08 2022).

[42] Great Learning Team. *What is Rectified Linear Unit (ReLU)? | Introduction to ReLU Activation Function*. URL: https://www.mygreatlearning.com/blog/relu-activation-function/#sh4. (accessed: 14/09 2022).

[43] Jason Brownlee. *Deep Learning Models for Multi-Output Regression*. URL: https://machinelearningmastery.com/deep-learning-models-for-multi-output-regression/. (accessed: 13/08 2022).

[44] Jason Brownlee. *What is Deep Learning?* URL: https://machinelearningmastery.com/what-is-deep-learning/. (accessed: 13/08 2022).

[45] Gaurav Sharma. *5 Regression Algorithms you should know – Introductory Guide!* URL: https://www.analyticsvidhya.com/blog/2021/05/5-regression-algorithms-you-should-know-introductory-guide/. (accessed: 14/08 2022).

[46] Niranjan Kumar. *Batch Normalization and Dropout in Neural Networks with Pytorch*. URL: https://towardsdatascience.com/batch-normalization-and-dropout-in-neural-networks-explained-with-pytorch-47d7a8459bcd. (accessed: 14/08 2022).

[47] Pytorch. *BatchNorm1d*. URL: https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html. (accessed: 14/08 2022).

[48] Jason Brownlee. *How to Avoid Overfitting in Deep Learning Neural Networks*. URL: https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/. (accessed: 14/08 2022).

[49] IBM Cloud Education. *Overfitting*. URL: https://www.ibm.com/cloud/learn/overfitting. (accessed: 18/08 2022).

[50] Jason Brownlee. *A Gentle Introduction to Dropout for Regularizing Deep Neural Networks*. URL: https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/. (accessed: 18/08 2022).

[51] John Terra. *Keras vs Tensorflow vs Pytorch: Key Differences Among the Deep Learning Framework*. URL: https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article. (accessed: 21/08 2022).

[52] Maxime Labonne. *What is a Tensor in Deep Learning?* URL: https://mlabonne.github.io/blog/what-is-a-tensor/. (accessed: 25/08 2022).

[53] Yuqing Chen, Erdinc Saygin. *Seismic Inversion by Hybrid Machine Learning*. URL: https://www.researchgate.net/publication/344260274_Seismic_Inversion_by_Hybrid_Machine_Learning. (accessed: 25/08 2022).

[54] Vihar Kurama. *PyTorch vs. TensorFlow: Which Framework Is Best for Your Deep Learning Project?* URL: https://builtin.com/data-science/pytorch-vs-tensorflow. (accessed: 25/08 2022).

[55] Nilesh Barla. *Distributed Training: Frameworks and Tools*. URL: https://neptune.ai/blog/distributed-training-frameworks-and-tools. (accessed: 25/08 2022).

[56] Simran Kaur Arora. *PyTorch vs TensorFlow: Difference you need to know*. URL: https://hackr.io/blog/pytorch-vs-tensorflow. (accessed: 25/08 2022).

[57] Ryan O'Connor. *PyTorch vs TensorFlow in 2022*. URL: https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/. (accessed: 26/08 2022).

[58] Eric Hofesmann. *The Machine Learning Lifecycle in 2021*. URL: https://towardsdatascience.com/the-machine-learning-lifecycle-in-2021-473717c633bc. (accessed: 27/08 2022).

[59] Jason Brownlee. *PyTorch Tutorial: How to Develop Deep Learning Models with Python*. URL: https://machinelearningmastery.com/pytorch-tutorial-develop-deep-learning-models/. (accessed: 25/08 2022).

[60] 3Blue1Brown. *Neural networks playlist*. URL: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. (accessed: 28/08 2022).

[61] sentdex. *Machine Learning with Python playlist*. URL: https://www.youtube.com/playlist?list=PLQVvvaa0QuDfKTOs3Keq_kaG2P55YRn5v. (accessed: 28/08 2022).

[62] sentdex. *Pytorch - Deep learning w/ Python playlist*. URL: https://www.youtube.com/playlist?list=PLQVvvaa0QuDdeMyHEYc0gxFpYwHY2Qfdh. (accessed: 28/08 2022).

[63] Jason Brownlee. *What Is Data Preparation in a Machine Learning Project*. URL: https://machinelearningmastery.com/what-is-data-preparation-in-machine-learning/. (accessed: 23/09 2022).

[64] Vijay Kanade. *What Is Machine Learning? Definition, Types, Applications, and Trends for 2022*. URL: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-ml/. (accessed: 07/10 2022).

[65] Packt. *Types of machine learning*. URL: https://subscription.packtpub.com/book/data/9781789537253/13/ch13lvl1sec83/types-of-machine-learning. (accessed: 07/10 2022).

[66] David Weedmark. *Machine Learning Model Training: What It Is and Why It's Important*. URL: https://www.dominodatalab.com/blog/what-is-machine-learning-model-training. (accessed: 23/09 2022).

[67] Jason Brownlee. *Train-Test Split for Evaluating Machine Learning Algorithms*. URL: https://machinelearningmastery.com/train-test-split-for-evaluating-machine-learning-algorithms/. (accessed: 18/10 2022).

[68] V. Roshan Joseph. *Optimal ratio for data splitting*. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/sam.11583. (accessed: 18/10 2022).

[69] Sagar Sharma. *Epoch vs Batch Size vs Iterations*. URL: https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9. (accessed: 18/10 2022).

[70] Upendra Vijay. *What is epoch and How to choose the correct number of epoch*. URL: https://medium.com/@upendravijay2/what-is-epoch-and-how-to-choose-the-correct-number-of-epoch-d170656adaaf. (accessed: 18/10 2022).

[71] Jason Brownlee. *Regression Metrics for Machine Learning*. URL: https://machinelearningmastery.com/regression-metrics-for-machine-learning/. (accessed: 21/10 2022).

[72] Songhao Wu. *3 Best metrics to evaluate Regression Model?* URL: https://towardsdatascience.com/what-are-the-best-metrics-to-evaluate-your-regression-model-418ca481755b. (accessed: 21/10 2022).

[73] Raghav Agrawal. *Know The Best Evaluation Metrics for Your Regression Model !* URL: https://www.analyticsvidhya.com/blog/2021/05/know-the-best-evaluation-metrics-for-your-regression-model/. (accessed: 21/10 2022).

[74] Eugene Dorfman. *How Much Data Is Required for Machine Learning?* URL: https://postindustria.com/how-much-data-is-required-for-machine-learning/. (accessed: 01/01 2023).

[75] Jason Brownlee. *How Much Training Data is Required for Machine Learning?* URL: https://machinelearningmastery.com/much-training-data-required-machine-learning/. (accessed: 01/01 2023).

[76] Gabriel Atkin. *House Sale Price Prediction With PyTorch*. URL: https://www.youtube.com/watch?v=9K3guNrgy-4. (accessed: 07/10 2022).

[77] Krish Naik. *Pytorch Tutorial 5-Live- Kaggle Advance House Price Prediction Using Pytorch Deep Learning*. URL: https://www.youtube.com/watch?v=0fSn9WG8MrA. (accessed: 07/10 2022).

Appendix - Contents

A Training algorithms

The three Training Algorithms briefly discussed in Section 4.2.2 are presented analytically here, with their full code, personal comments, and explanation and source materials for anyone interested in reading further into them.

A.1 Basic classification ML problem

The first part of most ML algorithms deals with data preparation. In all three of the algorithms here, the dataset is loaded from an existing database, making the data preparation much simpler than in reality. A realistic version of the data preparation that was required for this Thesis is illustrated in Appendix B. As can be observed in Figure A.1 after loading the dataset, the shape of the dataset is displayed for illustration, and then an 80-20 test-train split is applied with a random seat.

A Standard Scaler is then applied to normalize the data since no preprocessing was conducted. The needed transformation to tensors will always be a cornerstone in Ml programs for the reasons discussed in Section 3.1.1. Moreover, finally, a reshaping of the target variable (y).

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split



#Step 1: Data preparation
bc = datasets.load_breast_cancer()
X,y = bc.data, bc.target

n_samples, n_features = X.shape
print (n_samples, n_features)

X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2, random_state=1234)

#Data Scaling
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

#Converting to tensors
X_train = torch.from_numpy(X_train.astype(np.float32))
X_test = torch.from_numpy(X_test.astype(np.float32))
y_train = torch.from_numpy(y_train.astype(np.float32))
y_test = torch.from_numpy(y_test.astype(np.float32))

# #y Reshaping
y_train = y_train.view(y_train.shape[0], 1 )
y_test = y_test.view(y_test.shape[0], 1 )
```

Figure A.1: A1-Data Preparation

For the purpose of the problem, both the activation function and the layers are as simplified as possible, comprised of a Sigmoid and a simple Linear regression layer. Similarly, the LR starts at a predefined 0.02 and follows the preset SGD optimizer, as can be followed in the code of Figure A.2. These practices vary from the recommendations in the literature but are sufficient for an introductory algorithm.

```python
#Step 2: Model creation
class LogisticRegression(nn.Module):
    def __init__(self, n_input_features):
        super(LogisticRegression, self).__init__()
        #Layer creation
        self.linear = nn.Linear(n_input_features, 1)

    def forward(self, x):
        y_predicted = torch.sigmoid(self.linear(x))
        return y_predicted

model = LogisticRegression(n_features)



#Step 3: Loss calculation and optimizer
learning_rate = 0.02
criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate)
```

Figure A.2: A1-Model creation

Finally, in the training and evaluation part, a conservative loop of 2000 epochs was performed, with a loss tracker and an update on the loss every ten epochs. After the training is completed, an accuracy check against the verification data is performed by a simple equation of the sum of the correct predictions over the total of predictions, as illustrated in Figure A.3.

```python
#Step 4: Training loop
num_epochs = 2000
for epoch in range(num_epochs):

    #forward pass
    y_predicted = model(X_train)

    #loss calculation
    loss = criterion(y_predicted, y_train)

    #backward pass
    loss.backward()

    #update optimizer
    optimizer.step()

    #zero gradients
    optimizer.zero_grad()

    #return epoch info every 10 steps
    if (epoch+1) % 10 ==0:
        print(f'epoch: {epoch+1}, Loss = {loss.item():.4f} ')

#Step 5: Model Evaluation
with torch.no_grad():
    y_predicted = model(X_test)
    y_predicted_cls = y_predicted.round()
    acc = y_predicted_cls.eq(y_test).sum() / float(y_test.shape[0])
    print(f'The accuracy is {acc:.4f}')
```

Figure A.3: A1-Training

After completing this simplistic algorithm, I was pleasantly surprised by the results. Neither the activation function nor the optimizer was up to the standards of established ML projects, there was no tuning in the hyperparameters, and the database was relatively small (568 cases with 29 input parameters each). Despite all that, the program could predict the tumor category (benign or malignant corresponding to 0 or 1) with an accuracy of over 0.95 (given the probabilistic nature of the algorithm).

A.2 Simplistic version of a regression problem for housing market price prediction

The two solutions to the house pricing market problem that will be presented here were thoroughly examined as bases for this Thesis' final model. The solution in Appendix A.3 is more intricate and includes many features that can be used to develop the model in future work further and introduce new capabilities. Despite that, this algorithm served as a good introduction to regression problems. The code for this algorithm was provided by the very educative video of Gabriel Atkin [76].

The dataset, in this case, is loaded from a database but is not in a usable format yet, and contains both irrelevant variables and variables in the wrong format. For that reason, some simple transformations are performed to drop the date columns, isolate the unique zip codes and perform one-hot encoding. Encoding is applied on mixed data, containing both categorical and continuous features, for memory conservation and functionality, and although very interesting, proved to be outside the scope of this Thesis. Regardless, I spent a considerable amount of time understanding the different encoding methods for future knowledge. After the dataset has been converted into a more refined version, the usual feature selection, scaling, train-test split, and tensor transformation is performed, as illustrated in Figure A.4.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

data = pd.read_csv('C:/Users/dimit/Desktop/Price prediction/First case/kc_house_data.csv')

#1) Preprocessing

#dropping irrelevant data
data = data.drop('id' , axis=1)
data['year'] = data['date'].apply(lambda x: x[0:4])
data['month'] = data['date'].apply(lambda x: x[4:6])
data = data.drop('date', axis=1)
len(data['zipcode'].unique())

def onehot_encode(df, column, prefix):
    df = df.copy()
    dummies = pd.get_dummies(df[column], prefix=prefix)
    df = pd.concat([df, dummies], axis=1)
    df = df.drop(column, axis=1)
    return df

data = onehot_encode(data, 'zipcode' , 'zip')
data = data.drop('yr_renovated' , axis=1)
print (data)

#Setting features and labels
y = data['price'].copy()
X = data.drop('price', axis=1).copy()

#Scaling
scaler = StandardScaler()
X = scaler.fit_transform(X)

#Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, random_state=1)

#Converting to tensors
X_train = torch.tensor(X_train).type(torch.float32)
X_test = torch.tensor(X_test).type(torch.float32)
y_train = torch.tensor(np.array(y_train)).type(torch.float32)
y_test = torch.tensor(np.array(y_test)).type(torch.float32)
```

Figure A.4: A2-Data Preparation

On the model definition, the NN is far more complicated than the previous one, possessing two linear layers with ReLU activation functions and employing Adam as the optimizer (with initial LR 0.01). Moreover, the loss is observed by the MSE. The model definition is presented in Figure A.5.

```python
#2) Model creation

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.layer1 = nn.Linear(88, 64)
        self.layer2 = nn.Linear(64, 64)
        self.out = nn.Linear(64, 1)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = self.out(x)
        return x

net = Net()

#Optimizing
optimizer = torch.optim.Adam(net.parameters(), lr=0.01)
criterion = nn.MSELoss()
```

Figure A.5: A2-Model creation

The training of this algorithm confused me profoundly and led me to search for alternatives since it diverged significantly from the knowledge I had accumulated until this point. The nonexistence of epoch definition and the zip loop still remain a mystery even after rewatching the video while writing the Thesis. It may be that I am still missing some part of the information or that there is something wrong with the Pytorch equivalent of the code on the video since the programmer admits to just starting Pytorch and is himself confused about the accuracy discrepancy with Tensorflow. Regardless, I include the original code for training in Figure A.6.

```python
#3)Model training
for x, target in zip(X_train, y_train):
    optimizer.zero_grad()
    output = net(x)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()

#Loss calculation

total_loss = 0
for x, target in zip(X_train, y_train):
    output = net(x)
    loss = criterion(output, target)
    total_loss += loss
print (total_loss)

avg_loss = total_loss/len(X_test)
print (avg_loss)
RMSE = torch.sqrt(avg_loss).detach().numpy()

print ("RMSE:", RMSE)
```

Figure A.6: A2-Training

The results of this algorithm are rather disappointing. Despite utilizing an impressive dataset of more than twenty thousand cases, the RMSE is on the same power scale as the houses' actual selling prices. Unfortunately, there are no extra metrics to the code other than the RMSE to identify the problem, which served as a great lesson in the importance of being able to observe the learning through the epochs.

A.3 Complex version of a regression problem for housing market price prediction

This is the algorithm that impressed me the most amongst those I examined, both for the agreement of its practices with the most recent literature and for various extra features that broadened my horizons into the depth available in ML and DL. The programmer's Youtube channel, Krish Naik, possesses an extensive knowledge pool covering most topics regarding ML and Data processing, which I highly recommend, especially for beginners like myself. This algorithm is part of his Pytorch tutorial series with detailed explanation [77]. Since I do not believe I can explain it better and many parts were already discussed in my version in Chapter 4, I will only briefly comment on the parts I did not include in my model and found very interesting for future projects.

The dataset includes a mix of categorical and continuous input variables in this problem. The obvious first step is to create three different lists, one with the categorical, one with the continuous, and one with the output features, and convert it to a tensor list, as can be observed in Figure A.7.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import datetime

#1) Data Preparation
df=pd.read_csv('C:/Users/dimit/Desktop/Thesis/Codes/Price prediction/Second case/houseprice.csv',usecols=["SalePric
                                        "Street", "YearBuilt", "LotShape", "1stFlrSF", "2ndFlrSF"]).dropna()

df['Total Years'] = datetime.datetime.now().year-df['YearBuilt']
df.drop("YearBuilt",axis=1,inplace=True)
df.columns

#create categorical features and output features
cat_features=["MSSubClass", "MSZoning", "Street", "LotShape"]
out_feature="SalePrice"

lbl_encoders={}
for feature in cat_features:
    lbl_encoders[feature]=LabelEncoder()
    df[feature]=lbl_encoders[feature].fit_transform(df[feature])
print (df)

#stacking and converting to tensors for cat features
cat_features=np.stack([df['MSSubClass'],df['MSZoning'],df['Street'],df['LotShape']],1)
cat_features=torch.tensor(cat_features,dtype=torch.int64)
#print (cat_features)
```

Figure A.7: A3-Data prep first part

The more complex process is to create embedding layers for each category, which allows for denser, more information-rich vectors that are processed and manipulated more efficiently by the algorithm, as illustrated in Figure A.8. This encoding method was excessive for the Thesis since all data was continuous, but there exists a lot of potential to enhance the model's capabilities.

```
#create continuous variable
cont_features=[]
for i in df.columns:
    if i in ["MSSubClass", "MSZoning", "Street", "LotShape","SalePrice"]:
        pass
    else:
        cont_features.append(i)

#stacking and converting to tensors for continuous variables
cont_values=np.stack([df[i].values for i in cont_features],axis=1)
cont_values=torch.tensor(cont_values,dtype=torch.float)


#create dependent feature
y=torch.tensor(df['SalePrice'].values,dtype=torch.float).reshape(-1,1)
#print (y)

#embedding size for categorical columns
cat_dims=[len(df[col].unique()) for col in ["MSSubClass", "MSZoning", "Street", "LotShape"]]
embedding_dim= [(x, min(50, (x + 1) // 2)) for x in cat_dims]

embed_representation=nn.ModuleList([nn.Embedding(inp,out) for inp,out in embedding_dim])
#print (embed_representation)

cat_featuresz=cat_features[:4]
pd.set_option('display.max_rows', 500)
embedding_val=[]
for i,e in enumerate(embed_representation):
    embedding_val.append(e(cat_features[:,i]))
z = torch.cat(embedding_val, 1)
#print (z)

#implement dropupout
droput=nn.Dropout(.4)
final_embed=droput(z)
print (final_embed)
```

Figure A.8: A3-Data prep second part

The training and evaluation (presented in Figures A.9 and A.10 for completion) are similar to this Thesis'. The difference arises from the fact that training and evaluation are performed separately for each feature category (categorical and continuous).

```
#2)creating a Feed Forward Neural Network

class FeedForwardNN(nn.Module):

    def __init__(self, embedding_dim, n_cont, out_sz, layers, p=0.5):
        super().__init__()
        self.embeds = nn.ModuleList([nn.Embedding(inp,out) for inp,out in embedding_dim])
        self.emb_drop = nn.Dropout(p)
        self.bn_cont = nn.BatchNorm1d(n_cont)

        layerlist = []
        n_emb = sum((out for inp,out in embedding_dim))
        n_in = n_emb + n_cont

        for i in layers:
            layerlist.append(nn.Linear(n_in,i))
            layerlist.append(nn.ReLU(inplace=True))
            layerlist.append(nn.BatchNorm1d(i))
            layerlist.append(nn.Dropout(p))
            n_in = i
        layerlist.append(nn.Linear(layers[-1],out_sz))
        self.layers = nn.Sequential(*layerlist)

    def forward(self, x_cat, x_cont):
        embeddings = []
        for i,e in enumerate(self.embeds):
            embeddings.append(e(x_cat[:,i]))
        x = torch.cat(embeddings, 1)
        x = self.emb_drop(x)

        x_cont = self.bn_cont(x_cont)
        x = torch.cat([x, x_cont], 1)
        x = self.layers(x)
        return x

torch.manual_seed(100)
model=FeedForwardNN(embedding_dim,len(cont_features),1,[100,50],p=0.4)
```

Figure A.9: A3-Model creation

```
#3) loss calculation and Adam optimizer
loss_function=nn.MSELoss()
optimizer=torch.optim.Adam(model.parameters(),lr=0.01)

#batches creation
batch_size=1200
test_size=int(batch_size*0.15)
train_categorical=cat_features[:batch_size-test_size]
test_categorical=cat_features[batch_size-test_size:batch_size]
train_cont=cont_values[:batch_size-test_size]
test_cont=cont_values[batch_size-test_size:batch_size]
y_train=y[:batch_size-test_size]
y_test=y[batch_size-test_size:batch_size]

#epochs and loss reduction
epochs=5000
final_losses=[]
for i in range(epochs):
    i=i+1
    y_pred=model(train_categorical,train_cont)
    loss=torch.sqrt(loss_function(y_pred,y_train)) ### RMSE
    final_losses.append(loss)
    if i%10==1:
        print("Epoch number: {} and the loss : {}".format(i,loss.item()))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(range(epochs), final_losses)
plt.ylabel('RMSE Loss')
plt.xlabel('epoch');

#data validation
y_pred=""
with torch.no_grad():
    y_pred=model(test_categorical,test_cont)
    loss=torch.sqrt(loss_function(y_pred,y_test))
print('RMSE: {}'.format(loss))

data_verify=pd.DataFrame(y_test.tolist(),columns=["Test"])
data_predicted=pd.DataFrame(y_pred.tolist(),columns=["Prediction"])
final_output=pd.concat([data_verify,data_predicted],axis=1)
final_output['Difference']=final_output['Test']-final_output['Prediction']
print (final_output.head())
```

Figure A.10: A3-Training

B Data processing algorithms

The series of algorithms presented here detail the complete order of operations performed before the dataset was ready to be inserted into the ML model. The automated processes created to generate the raw data from POLCA-T for all cases will be explained as well as the different parts of the data processing procedure from the raw data extracted from POLCA-T to the final dataset and external verification dataset. In total, the distribution files from 5 different burn-up cycles were collected for the input data and an additional (c26) to be used for testing on a fully unknown cycle. Camilla Rotander provided the distribution files for each state sorted in cycles c25-c30. The order of operations follows the same basis, included in parenthesis.

B.1 Automatization to extract the raw dataset (run_c25-30)

The scripts get_polut_inputsC25-30 automate the commands to extract the data and will not be included here for proprietary reasons. The .txt files generated from this script were stored in the folder Polut_inputfiles. The folders run_c25-30 include copies of the .dat distribution files and the python script run_polut_inputsC25-30, with the example for c30 presented in Figure B.1. The only issue encountered in this part was the restriction in the name of the .txt file inserted in polut. After trying many techniques to accept a variable-stored name without any result, my solution was to isolate all the .txt from the .dat files and initiate a loop that renamed them in sequence to polutinputdata.txt and run polut on each under this same name. To avoid confusion, the output from polut retained its original .dat name and was saved in this folder.

```
import pandas as pd
import os
import sys
import subprocess as s


path='/san/jobA/job1/2022p0045/Dimitris/c30/run_c30'

cycles=[]
filename=[]
for root, dirs, files in os.walk(path):
       for file in files:
           if file.endswith('.txt'):
               filename.append(file)
filename=filename[0:]

for filenames in filename:
    old_name = filenames
    new_name = '/san/jobA/job1/2022p0045/Dimitris/c30/run_c30/polutindata.txt'
    os.renames(old_name, new_name)
    p1=s.run(["p7 -v 4.24.1 polut < polutindata.txt"], shell= True)
```

Figure B.1: run_polut_inputsC30

B.2 Complete Data processing

This ended up being the most challenging part of the Thesis and required incredible amounts of research and coding attempts. It will be split into two parts, the first located in folder Data_performs a channel-mapping function to be able to instantly assign the correct reactor assembly to each value, and the other located in folders fix_c25-30, applying a series of data transformation techniques to extract them as a csv table.

B.2.1 Channel-mapping

The main idea for this process revolves around the geometry in which the raw data is presented. The assemblies are presented in a complicated Y/X coordinates system in a non-intuitive order. By using the assembly numbers map from polut, the script presented in Figure B.2, uses Pythons iloc function to determine which positions represent reflectors (having a zero value in this mode), then drops them and rearranges the rest by attributing the assembly number to the channel indicated by polut, and save this correlation in a map csv, that will be used later to easily distribute all values from raw data to the correct assembly.

```
import pandas as pd

df=pd.read_csv('C:/Users/dimit/Desktop/Data processing/channels.txt')
df2=df.drop(df.index[0:82])
df2=df2.drop(df2.index[30:36])
df2=df2.drop(df2.index[60])

df2=pd.melt(df2)
df3= df2['value'].str.split(' ', expand=True)
df3=df3.drop(0, axis=1)
df3=df3.drop(2, axis=1)
df3=df3.drop(3, axis=1)
df3=df3.drop(4, axis=1)
df4=df3.T

new_header = df4.iloc[0] #grab the first row for the header
df4 = df4[1:] #take the data less the header row
df4.columns = new_header #set the header row as the df header
df4.fillna("",inplace=True)

df4=pd.melt(df4)
df4['value'] = pd.to_numeric(df4['value'])
df4.dropna(subset=['value'], inplace=True)

# index=df4.index
# zeros=df4['value']==0
# print(zeros)
# zeros_indices=index[zeros]
# zeros_indices_list=zeros_indices.tolist()
# print(zeros_indices_list)

df4=df4.drop([19, 90, 94, 98, 102, 106, 110, 180, 184, 258, 262, 336, 340, 413, 417, 479, 554, 629, 704, 7

df4=df4.rename(columns = {df4.columns[0]:'XChannel'})
df4=df4.rename(columns = {df4.columns[1]:'Xvalue'})

df4=df4.reset_index(drop=True)
print(df4)
df4.to_csv('channelmap.csv')
```

Figure B.2: Channel-mapping

B.2.2 Data transformations (fix_c25-30)

The first part of this process was selective data cleaning. Polut's .txt output files included many lines with unnecessary messages from polut and irrelevant data. Additionally, all the input variables super-categories (power, void, channel-flow, and burn-up for each reactor assembly) were not separated, and the average reactor power and the inlet temperature were mixed among other irrelevant values.

As shown in Figure B.3, the polut outputs are selected and looped through to locate and drop the unnecessary lines in each of them (fortunately, the .txt geometry is identical for all outputs). The relevant lines are stored in 6 dataframes; (df1-4) for the main categories and df5, dfP for the temperature and average power, respectively.

```
import pandas as pd
import os
import sys

path='/san/jobA/job1/2022p0045/Dimitris/c30/fix_c30'


cycles=[]
filename=[]
for root, dirs, files in os.walk(path):
        for file in files:
            if file.startswith('output'):
                filename.append(file)


for filenames in filename:
    df=pd.read_csv(filenames,  header=None, sep='\n')


    # 1)split into 5 dataframes

    df1 = df.iloc[:77,:]
    # print(df1)

    df2 = df.iloc[79:156,:]
    # print(df2)

    df3 = df.iloc[157:234,:]
    # print(df3)

    df4 = df.iloc[235:312,:]
    # print(df4)

    df5 = df.iloc[313:,:]
    # print(df5)

    dfP = df.iloc[1]
    # print(dfP)
```

Figure B.3: fix_c30 variable category separation

The next step is creating a loop through df1-4. The pd.melt function reshapes them by separating the values into columns. The numerical values are recognized as a long string, so a str.split is applied, with spacebar as the criterion. A transpose is applied to switch rows to columns, and the channel number row is set as the header, followed by another melt. The value column is transformed into numbers with pd.to_numeric and set to round up to five decimals. The Channel number column is converted to numeric as well. The index axis is updated to ensure that the correct amount of values is retained. The four dataframes are appended in the df_new list and reassigned df0-df3 names to be handled separately again. This process is presented in Figure B.4.

```
    # 2)drop needless lines in each dataframe

new_dfs=[]
for df in [df1, df2, df3, df4]:
    df_new= df.reset_index(drop=True)
    df_new=df_new.drop([0,1,2,3,4,5,6,7,8,9,10,41,42,43,44,45,46])
    df_new=pd.melt(df_new)
    df_new= df_new['value'].str.split(' ', expand=True)
    df_new=df_new.drop(0, axis=1)
    df_new=df_new.drop(2, axis=1)
    df_new=df_new.drop(3, axis=1)
    df_new=df_new.drop(4, axis=1)
    df_new=df_new.T
    new_header = df_new.iloc[0] #grab the first row for the header
    df_new = df_new[1:] #take the data less the header row
    df_new.columns = new_header #set the header row as the df header
    df_new.fillna("",inplace=True)
    df_new=pd.melt(df_new)
    df_new['value'] = pd.to_numeric(df_new['value'])
    df_new.dropna(subset=['value'], inplace=True)
    df_new=df_new.reset_index(drop=True)
    df_new=df_new.drop(index=[4, 9, 10, 11, 12, 13, 14, 24, 25, 36, 37, 49, 50, 63, 64, 78, 92, 106

    # 3)Fix indexes and split into dataframes

    df_new['value'] =df_new['value'].round(decimals=5)
    df_new=df_new.reset_index(drop=True)

    df_new=df_new.rename(columns = {df_new.columns[0]:'Channel'})
    df_new['Channel'] = pd.to_numeric(df_new['Channel'])

    df_new=df_new.rename_axis('index')
    df_new=df_new.reset_index(drop=True)
    new_dfs.append(df_new)

for i in range(len(new_dfs),):
    globals()[f"df{i}"] = new_dfs[i]
```

Figure B.4: fix_c30 dataframe transformation for each variable category

The polut outputs include a scaling factor for each variable category, which is then applied to its corresponding variable noted by the df_new names. Then, the script reads the channel-mapping csv created in B.2.1 and uses the shared Channel columns in both dataframes to assign the correct assembly number to the values, sort them by assembly descending order (new Channel columns), as illustrated in Figure B.5.

```
    # 4)Apply multipliers and channelmap individually

df0['value']=10**-3*df0['value']
df1['value']=10**-2*df1['value']
df2['value']=10**-2*df2['value']
df3['value']=10**-4*df3['value']

df6=pd.read_csv('/san/jobA/job1/2022p0045/Data_processing/channelmap.csv')

df0= df6.join(df0)
df0=df0.drop(['Channel', 'XChannel', 'Unnamed: 0'], axis=1)
df0 = df0.sort_values(by = ['Xvalue'])
df0=df0.rename(columns = {df0.columns[0]:'Channel'})
df0=df0.reset_index(drop=True)

df1= df6.join(df1)
df1=df1.drop(['Channel', 'XChannel', 'Unnamed: 0'], axis=1)
df1 = df1.sort_values(by = ['Xvalue'])
df1=df1.rename(columns = {df1.columns[0]:'Channel'})
df1=df1.reset_index(drop=True)

df2= df6.join(df2)
df2=df2.drop(['Channel', 'XChannel', 'Unnamed: 0'], axis=1)
df2 = df2.sort_values(by = ['Xvalue'])
df2=df2.rename(columns = {df2.columns[0]:'Channel'})
df2=df2.reset_index(drop=True)

df3= df6.join(df3)
df3=df3.drop(['Channel', 'XChannel', 'Unnamed: 0'], axis=1)
df3 = df3.sort_values(by = ['Xvalue'])
df3=df3.rename(columns = {df3.columns[0]:'Channel'})
df3=df3.reset_index(drop=True)
```

Figure B.5: fix_c30 power-scaling and channel-mapping for each variable category

The last part of the fix_c30 script utilizes the two dataframes reserved for power and temperature to perform the same routine of melt, split the string variables by the blank space, locate the cells holding the value for each, respectively, and create two new dataframes holding those values and assigning them the names ”Temperature” and ”Power.” All six dataframes are merged with pd.concat. Finally, the distribution number in the name of the output file is inserted as the value in a new first column (”Distribution”), and the merged dataframe is saved as a csv with that name, as can be followed in Figure B.6.

```
    # 5)Add temperature and power

df5=df5.reset_index(drop=True)
df5=pd.melt(df5)
df5= df5['value'].str.split(' ', expand=True)
temp=df5.iloc[10][8]
power=df5.iloc[7][28]
dft=['Temperature',temp]
dfP=['Power',power]
df4 = pd.DataFrame(dft, index= ['Channel', 'value'])
df5 = pd.DataFrame(dfP, index= ['Channel', 'value'])
df4=df4.T
df5=df5.T


    # 7)Merge together and fix formatting

df=pd.concat([df5, df4, df0, df1, df2, df3])
df=df.reset_index(drop=True)
df=df.T
new_header = df.iloc[0]
df = df[1:]
df.columns = new_header


f1=filenames.replace('output_','')
f1=f1.replace('.dat.txt', '')
df.insert(0, "Distribution", [f1], True)
df.to_csv(f1+'.csv', index=False)
```

Figure B.6: fix_c30 merge and renaming

B.3 Adding decay ratio and frequency (add_decay _c25-30)

The process of adding the target output variables (decay ratio and frequency) was also split into two parts, located in folder add_decay _c25-30, one devoted to reading and cleaning the Resultat_c25-30 files provided by Camilla Rotander and one to merge the csv files for all cases of each cycle with the processed data created in B.2.

B.3.1 Reading decay ratio and frequency (read_decay _c25-30)

The Resultat_c25-30 files had two issues: the distribution number decay ratio and frequency were in that order vertically, and some of the distributions had run into an error with a string starting with Westinghouse below the distribution number.

For the error part, after a pd.melt, the iloc command was utilized for appearances of ”West”. However, they had to be dropped along with the previous line of the dataframe (distribution number). This was solved manually by putting the number of the iloc line and the previous number and saving the error-less dataframe as df1.

For the vertical issue, the rows that contained the different variables were separated by searching for the ”_” (distribution number rows), ”DEC” (decay ratio rows), and ”FREQ” (frequency rows) and creating df2-4, each with only one of the variables present. Specifically, in dataframes 3 and 4, the Frequency and decay ratio values were part of a string. pd.to_numeric was producing only NaN values. Therefore the following command was utilized to keep only strings containing the numbers 0-9 and the decimal places: astype(’str’).str.extract(’([0-9]+[,./]*[0-9]*)’). Finally, the three dataframes were merged in one horizontally with pd.concat and saved to the c25-30_final folder as decay_ratio.csv, as can be observed in Figure B.7.

```
import pandas as pd
import os
import sys

path='/san/jobA/job1/2022p0045/Dimitris/c30/add_decay_c30'
# path='\\svstsmb\job1\2022p0045\Dimitris\c30\add_decay_c30'

df=pd.read_csv('/san/jobA/job1/2022p0045/Dimitris/c30/add_decay_c30/Resultat_c30.txt',  header=None)
# df=pd.read_csv('//svstsmb/job1/2022p0045/Dimitris/c30/add_decay_c30/Resultat_c30.txt',  header=None)


df=pd.melt(df)
df1=df.loc[df.value.str.contains('West')]
df=df.drop([24,25,110,111,133,134,141,142,158,159,184,185,222,223,227,228,235,236,282,283])

df2=df[df.value.str.contains('_')]
df2=df2.drop(['variable'], axis=1)
df2=df2.rename(columns = {df2.columns[0]:'Distribution'})
df3=df[df.value.str.contains('DEC')]
df3=df3.drop(['variable'], axis=1)
df3=df3.rename(columns = {df3.columns[0]:'Decay ratio'})
df3['Decay ratio'] = df3['Decay ratio'].astype('str').str.extract('([0-9]+[,./]*[0-9]*)')
df4=df[df.value.str.contains('FREQ')]
df4=df4.drop(['variable'], axis=1)
df4=df4.rename(columns = {df4.columns[0]:'Frequency'})
df4['Frequency'] = df4['Frequency'].astype('str').str.extract('([0-9]+[,./]*[0-9]*)')


list_dfs=[df2, df3, df4]

[df.reset_index(drop=True, inplace=True) for df in list_dfs]

merged= pd.concat([df2, df3, df4], axis=1, join='inner')
merged.to_csv('/san/jobA/job1/2022p0045/Dimitris/c30/c30_final/decay_ratio.csv', index=False)
```

Figure B.7: Reading decay ratio and frequency (read_decay _c25-30)

B.3.2 Merging all processed data (merge _c25-30)

The csv files for all cases of each cycle with the processed data created in B.2 were copied to the folders add_decay _c25-30 respectively, to enable merging all cases for each cycle in a single dataframe. A loop through the folder located all csv files (this is why the results of the previous steps were saved to a different file) and then assigns them to a list. A dataframe is created from the first row in the list and another loop passes through the rest of the list, dropping the header for all dataframes to not have the labels reappear after every data line. Then they are sequentially appended to the first row's dataframe and then pd.melted and expanded and saved to c25-30_final folder as merge.csv, as illustrated in Figure B.8.

```
import pandas as pd
import os
import sys

path='/san/jobA/job1/2022p0045/Dimitris/c30/add_decay_c30'


df=pd.read_csv('/san/jobA/job1/2022p0045/Dimitris/c30/add_decay_c30/Resultat_c30.txt',  header=None)




cycles=[]
filename=[]
for root, dirs, files in os.walk(path):
        for file in files:
            if file.endswith('.csv'):
                filename.append(file)


dataframes=pd.read_csv(filename[0],  header=None, sep='\n')
for filenames in filename[1:]:
    df=pd.read_csv(filenames,  header=None, sep='\n')
    df= df.iloc[1:].reset_index(drop=True)
    dataframes=pd.concat([dataframes,df], axis=0, )
df=pd.melt(dataframes)
df= df['value'].str.split(',', expand=True)
df.columns=df.iloc[0]
df=df[1:]


df.to_csv('/san/jobA/job1/2022p0045/Dimitris/c30/c30_final/merge.csv', index=False)
```

Figure B.8: Merging all processed data (merge _c25-30)

B.4 Removing distributions without decay ratio and frequency (c25-30_final)

All the folders c25-30_final include their merge.csv. and decay_ratio.csv generated in the previous step. The script final_c25-30 was designed to read those two files and merge them horizontally based on the ”Distribution” column. Since the ”Distribution” column of the decay_ratio.csv files is missing some distributions that were removed because of the error, only the ones present in both are kept in the merged dataframe. This very handily automatically removes all the distributions without decay ratio and frequency. Finally, the ”Distribution” column serves no more purpose and can be dropped, and the merged dataframe for each cycle was saved as final_c25-30.csv, respectively. This simple process can be followed in Figure B.9.

```python
import pandas as pd
import os
import sys

path='/san/jobA/job1/2022p0045/Dimitris/c30/c30_final'
# path='//svstsmb/job12022p0045/Dimitris/c30/c30_final'

df=pd.read_csv('/san/jobA/job1/2022p0045/Dimitris/c30/c30_final/decay_ratio.csv',  header=None)
# df=pd.read_csv('//svstsmb/job1/2022p0045/Dimitris/c30/c30_final/decay_ratio.csv',  header=None)
df.columns=df.iloc[0]
df=df[1:]


df2=pd.read_csv('/san/jobA/job1/2022p0045/Dimitris/c30/c30_final/merge.csv',  header=None)
# df2=pd.read_csv('//svstsmb/job1/2022p0045/Dimitris/c30/c30_final/merge.csv',  header=None)
df2.columns=df2.iloc[0]
df2=df2[1:]



final=pd.merge(df,df2, on="Distribution", how="left")
final=final.drop("Distribution", axis=1)
final.to_csv('final_c30.csv', index=False)
```

Figure B.9: Removing distributions without decay ratio and frequency (c25-30_final)

B.5 Merged and Verification datasets for the ML Model

The final_c26.csv was copied to the Verification folder and used in Section 5.2. The final_c25,27,28,29 and 30.csv were copied to the Merge folder, where the Merge script looped through all of them, concatenated them one after another, and saved the final input dataset (Merged.csv) for the ML comprised of 650 cases with 2804 variables each. The Merge script is presented in Figure B.10.

```python
import pandas as pd
import os
import glob



files=glob.glob("/san/jobA/job1/2022p0045/Dimitris/Merge/*.csv")

df=[]
for file in files:
    df2=pd.read_csv(file)
    df.append(df2)
df=pd.concat(df)
df= df.reset_index(drop=True)

df.to_csv('Merged.csv', index=False)
```

Figure B.10: Merged dataset creation for the ML Model

C Text scripts for all used algorithms

C.1 Thesis Model script

```
import torch
import torch.nn as nn
import numpy as np
import pandas as pd
torch.set_printoptions(linewidth=120)
from torch.utils.tensorboard import SummaryWriter

#1) Data Preparation
df=pd.read_csv('C:/Users/dimit/Desktop/Thesis/Codes/Thesis Model/Inputs/Merged.csv')

X=df.drop(['Decay ratio','Frequency'],axis=1)
z=df.drop((X),axis=1)
z1=z.drop(['Frequency'],axis=1)
y=torch.tensor(z1.values,dtype=torch.float).reshape(-1,1)

#create continuous features
cont_features=[]
for i in X.columns:
        cont_features.append(i)

cont_values=np.stack([X[i].values for i in cont_features],axis=1)
cont_values=torch.tensor(cont_values,dtype=torch.float)

#2)creating the Neural Network

class FeedForwardNN(nn.Module):

    def __init__(self,  n_cont, y,  layers, p=0.5):
        super().__init__()

        self.bn_cont = nn.BatchNorm1d(n_cont)
        n_in =  n_cont
        layerlist = []

        for i in layers:
            layerlist.append(nn.Linear(n_in,i))
            layerlist.append(nn.ReLU(inplace=True))
            layerlist.append(nn.BatchNorm1d(i))
            layerlist.append(nn.Dropout(p))
            n_in = i
        layerlist.append(nn.Linear(layers[-1],y))

        self.layers = nn.Sequential(*layerlist)

    def forward(self,  x):
        x = self.bn_cont(x)
        x = torch.cat([x], 1)
        x = self.layers(x)
        return x

torch.manual_seed(100)
model=FeedForwardNN(len(cont_features),1,[100,50],p=0.5)

#3) loss calculation and Adam optimizer
loss_function=nn.MSELoss()
optimizer=torch.optim.Adam(model.parameters(),lr=0.001)

#batches creation
batch_size=650 #(depends on cases)
test_size=int(batch_size*0.165)
X_train=cont_values[:batch_size-test_size]
X_test=cont_values[batch_size-test_size:batch_size]
y_train=y[:batch_size-test_size]
y_test=y[batch_size-test_size:batch_size]
print(len(X_train), len(X_test), len(y_train), len(y_test))
```

```
#epochs and training loss
epochs=3000
final_losses=[]
for i in range(epochs):
    i=i+1
    y_pred=model(X_train)
    train_loss=torch.sqrt(loss_function(y_pred,y_train)) ### RMSE
    final_losses.append(train_loss)
    tb.add_scalar("Training Loss", train_loss, i)
    if i%10==1:
        print("Epoch number: {} and the loss : {}".format(i,train_loss.item()))
    optimizer.zero_grad()
    train_loss.backward()
    optimizer.step()


#4)testing
y_pred=""
with torch.no_grad():
    y_pred=model(X_test)
    test_loss=torch.sqrt(loss_function(y_pred,y_test))
    tb.add_scalar("Testing Loss", test_loss, i)
data_verify=pd.DataFrame(y_test.tolist(),columns=["Test"])
data_predicted=pd.DataFrame(y_pred.tolist(),columns=["Prediction"])
final_output=pd.concat([data_verify,data_predicted],axis=1)
final_output['Difference']=final_output['Test']-final_output['Prediction']
    # tb.add_scalar("Testing Loss", test_loss, i)
#   tb.add_scalars('Training and Testing Loss', {"Training Loss" : train_loss,
                           # "Testing Loss" : test_loss}, i)
    #plt.figure(figsize=(10,10))
    #plt.scatter(y_test, y_pred, c='crimson')
    #p1 = max(max(y_pred), max(y_test))
    #p2 = min(min(y_pred), min(y_test))
    #plt.plot([p1, p2], [p1, p2], 'b-')
    #plt.xlabel('True Values', fontsize=15)
    #plt.ylabel('Predictions', fontsize=15)
    #plt.show()
print('RMSE: {}'.format(test_loss))
print (final_output)
# tb.close()


#5) model saving
torch.save(model.state_dict(),'ML_decay_ratio.pth')
```

C.2 Thesis Predicting script

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import pandas as pd
from torch import Tensor

  # Loading the saved model

class FeedForwardNN(nn.Module):

    def __init__(self,  n_cont, y,  layers, p=0.5):
        super().__init__()

        self.bn_cont = nn.BatchNorm1d(n_cont)
        n_in =  n_cont
        layerlist = []

        for i in layers:
            layerlist.append(nn.Linear(n_in,i))
            layerlist.append(nn.ReLU(inplace=True))
            layerlist.append(nn.BatchNorm1d(i))
            layerlist.append(nn.Dropout(p))
            n_in = i
        layerlist.append(nn.Linear(layers[-1],y))

        self.layers = nn.Sequential(*layerlist)

    def forward(self,  x):
        x = self.bn_cont(x)
        x = torch.cat([x], 1)
        x = self.layers(x)
        return x

torch.manual_seed(100)
model=FeedForwardNN(2802,1,[100,50],p=0.5)

model.load_state_dict(torch.load('ML_decay_ratio.pth'))
model.eval()


df=pd.read_csv('C:/Users/dimit/Desktop/Thesis/Codes/Thesis Model/Inputs/final_c26.csv')
X=df.drop(['Decay ratio','Frequency'],axis=1)
z=df.drop((X),axis=1)
z1=z.drop(['Frequency'],axis=1)

def predict(row, model):
    # convert row to data
    row = Tensor([row])
    # make prediction
    yhat = model(row)
    # retrieve numpy array
    yhat = yhat.detach().numpy()
    return yhat

list=[]

with torch.no_grad():
    for i in range(0, len(X)):
        yhat = predict(X.iloc[i], model)
        list.append(yhat)

flatten = np.array(list).flatten()
df1 = pd.DataFrame(flatten, columns=['Predictions'])
df2 = pd.concat([df1, z1], axis=1, join='inner')
df2['Difference'] = df2['Predictions'] - df2['Decay ratio']

df2.to_csv('C:/Users/dimit/Desktop/Thesis/Codes/Thesis Model/Decay_ratio_Predictions.csv', index=False)
```

C.3 Basic classification ML problem script

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

#Step 1: Data preparation
bc = datasets.load_breast_cancer()
X,y = bc.data, bc.target
n_samples, n_features = X.shape
print (n_samples, n_features)

X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2, random_state=1234)

#Data Scaling
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
#Converting to tensors
X_train = torch.from_numpy(X_train.astype(np.float32))
X_test = torch.from_numpy(X_test.astype(np.float32))
y_train = torch.from_numpy(y_train.astype(np.float32))
y_test = torch.from_numpy(y_test.astype(np.float32))
# #y Reshaping
y_train = y_train.view(y_train.shape[0], 1 )
y_test = y_test.view(y_test.shape[0], 1 )

#Step 2: Model creation
class LogisticRegression(nn.Module):
    def __init__(self, n_input_features):
        super(LogisticRegression, self).__init__()
        #Layer creation
        self.linear = nn.Linear(n_input_features, 1)
    def forward(self, x):
        y_predicted = torch.sigmoid(self.linear(x))
        return y_predicted
model = LogisticRegression(n_features)

#Step 3: Loss calculation and optimizer
learning_rate = 0.02
criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate)

#Step 4: Training loop
num_epochs = 2000
for epoch in range(num_epochs):

    #forward pass
    y_predicted = model(X_train)
    #loss calculation
    loss = criterion(y_predicted, y_train)
    #backward pass
    loss.backward()
    #update optimizer
    optimizer.step()
    #zero gradients
    optimizer.zero_grad()
    #return epoch info every 10 steps
    if (epoch+1) % 10 ==0:
        print(f'epoch: {epoch+1}, loss = {loss.item():.4f} ')

#Step 5: Model Evaluation
with torch.no_grad():
    y_predicted = model(X_test)
    y_predicted_cls = y_predicted.round()
    acc = y_predicted_cls.eq(y_test).sum() / float(y_test.shape[0])
    print(f'The accuracy is {acc:.4f}')
```

C.4 Simplistic version of a regression problem for housing market price prediction script

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

data = pd.read_csv('C:/Users/dimit/Desktop/Thesis/Codes/Price prediction/First case/kc_house_data.csv')

#1) Preprocessing
#dropping irrelevant data
data = data.drop('id' , axis=1)
data['year'] = data['date'].apply(lambda x: x[0:4])
data['month'] = data['date'].apply(lambda x: x[4:6])
data = data.drop('date', axis=1)
len(data['zipcode'].unique())

def onehot_encode(df, column, prefix):
    df = df.copy()
    dummies = pd.get_dummies(df[column], prefix=prefix)
    df = pd.concat([df, dummies], axis=1)
    df = df.drop(column, axis=1)
    return df

data = onehot_encode(data, 'zipcode' , 'zip')
data = data.drop('yr_renovated' , axis=1)
print (data)

#Setting features and labels
y = data['price'].copy()
X = data.drop('price', axis=1).copy()

#Scaling
scaler = StandardScaler()
X = scaler.fit_transform(X)

#Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, random_state=1)

#Converting to tensors
X_train = torch.tensor(X_train).type(torch.float32)
X_test = torch.tensor(X_test).type(torch.float32)
y_train = torch.tensor(np.array(y_train)).type(torch.float32)
y_test = torch.tensor(np.array(y_test)).type(torch.float32)

#2) Model creation

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.layer1 = nn.Linear(88, 64)
        self.layer2 = nn.Linear(64, 64)
        self.out = nn.Linear(64, 1)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = self.out(x)
        return x

net = Net()

#Optimizing
optimizer = torch.optim.Adam(net.parameters(), lr=0.01)
criterion = nn.MSELoss()
```

```
#3)Model training
for x, target in zip(X_train, y_train):
    optimizer.zero_grad()
    output = net(x)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()

#Loss calculation

total_loss = 0
for x, target in zip(X_train, y_train):
    output = net(x)
    loss = criterion(output, target)
    total_loss += loss
print (total_loss)

avg_loss = total_loss/len(X_test)
print (avg_loss)
RMSE = torch.sqrt(avg_loss).detach().numpy()

print ("RMSE:", RMSE)
```

C.5 Complex version of a regression problem for housing market price prediction script

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import datetime

#1) Data Preparation
df=pd.read_csv('C:/Users/dimit/Desktop/Thesis/Codes/Price prediction/Second case/houseprice.csv',usecols=["SalePrice", "MSSubClass", "MSZoning", "LotFrontage", "LotA
                                        "Street", "YearBuilt", "LotShape", "1stFlrSF", "2ndFlrSF"]).dropna()

df['Total Years'] = datetime.datetime.now().year-df['YearBuilt']
df.drop("YearBuilt",axis=1,inplace=True)
df.columns

#create categorical features and output features
cat_features=["MSSubClass", "MSZoning", "Street", "LotShape"]
out_feature="SalePrice"
lbl_encoders={}
for feature in cat_features:
    lbl_encoders[feature]=LabelEncoder()
    df[feature]=lbl_encoders[feature].fit_transform(df[feature])
print (df)

#stacking and converting to tensors for cat features
cat_features=np.stack([df['MSSubClass'],df['MSZoning'],df['Street'],df['LotShape']],1)
cat_features=torch.tensor(cat_features,dtype=torch.int64)
#print (cat_features)

#create continuous variable
cont_features=[]
for i in df.columns:
    if i in ["MSSubClass", "MSZoning", "Street", "LotShape","SalePrice"]:
        pass
    else:
        cont_features.append(i)

#stacking and converting to tensors for continuous variables
cont_values=np.stack([df[i].values for i in cont_features],axis=1)
cont_values=torch.tensor(cont_values,dtype=torch.float)

#create dependent feature
y=torch.tensor(df['SalePrice'].values,dtype=torch.float).reshape(-1,1)
#print (y)

#embedding size for categorical columns
cat_dims=[len(df[col].unique()) for col in ["MSSubClass", "MSZoning", "Street", "LotShape"]]
embedding_dim= [(x, min(50, (x + 1) // 2)) for x in cat_dims]
embed_representation=nn.ModuleList([nn.Embedding(inp,out) for inp,out in embedding_dim])
#print (embed_representation)

cat_featuresz=cat_features[:4]
pd.set_option('display.max_rows', 500)
embedding_val=[]
for i,e in enumerate(embed_representation):
    embedding_val.append(e(cat_features[:,i]))
z = torch.cat(embedding_val, 1)
#print (z)

#implement dropupout
droput=nn.Dropout(.4)
final_embed=droput(z)
print (final_embed)
```

```
#2)creating a Feed Forward Neural Network
class FeedForwardNN(nn.Module):
    def __init__(self, embedding_dim, n_cont, out_sz, layers, p=0.5):
        super().__init__()
        self.embeds = nn.ModuleList([nn.Embedding(inp,out) for inp,out in embedding_dim])
        self.emb_drop = nn.Dropout(p)
        self.bn_cont = nn.BatchNorm1d(n_cont)
        layerlist = []
        n_emb = sum((out for inp,out in embedding_dim))
        n_in = n_emb + n_cont
        for i in layers:
            layerlist.append(nn.Linear(n_in,i))
            layerlist.append(nn.ReLU(inplace=True))
            layerlist.append(nn.BatchNorm1d(i))
            layerlist.append(nn.Dropout(p))
            n_in = i
        layerlist.append(nn.Linear(layers[-1],out_sz))
        self.layers = nn.Sequential(*layerlist)
    def forward(self, x_cat, x_cont):
        embeddings = []
        for i,e in enumerate(self.embeds):
            embeddings.append(e(x_cat[:,i]))
        x = torch.cat(embeddings, 1)
        x = self.emb_drop(x)
        x_cont = self.bn_cont(x_cont)
        x = torch.cat([x, x_cont], 1)
        x = self.layers(x)
        return x
torch.manual_seed(100)
model=FeedForwardNN(embedding_dim,len(cont_features),1,[100,50],p=0.4)
#3) loss calculation and Adam optimizer
loss_function=nn.MSELoss()
optimizer=torch.optim.Adam(model.parameters(),lr=0.01)
#batches creation
batch_size=1200
test_size=int(batch_size*0.15)
train_categorical=cat_features[:batch_size-test_size]
test_categorical=cat_features[batch_size-test_size:batch_size]
train_cont=cont_values[:batch_size-test_size]
test_cont=cont_values[batch_size-test_size:batch_size]
y_train=y[:batch_size-test_size]
y_test=y[batch_size-test_size:batch_size]
#epochs and loss reduction
epochs=5000
final_losses=[]
for i in range(epochs):
    i=i+1
    y_pred=model(train_categorical,train_cont)
    loss=torch.sqrt(loss_function(y_pred,y_train)) ### RMSE
    final_losses.append(loss)
    if i%10==1:
        print("Epoch number: {} and the loss : {}".format(i,loss.item()))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(range(epochs), final_losses)
plt.ylabel('RMSE Loss')
plt.xlabel('epoch');
#data validation
y_pred=""
with torch.no_grad():
    y_pred=model(test_categorical,test_cont)
    loss=torch.sqrt(loss_function(y_pred,y_test))
print('RMSE: {}'.format(loss))
data_verify=pd.DataFrame(y_test.tolist(),columns=["Test"])
data_predicted=pd.DataFrame(y_pred.tolist(),columns=["Prediction"])
final_output=pd.concat([data_verify,data_predicted],axis=1)
final_output['Difference']=final_output['Test']-final_output['Prediction']
print (final_output.head())
```

C.6 Automatization to extract the raw dataset (run_c25-30) script

```
import pandas as pd
import os
import sys
import subprocess as s

path='/san/jobA/job1/2022p0045/Dimitris/c30/run_c30'

cycles=[]
filename=[]
for root, dirs, files in os.walk(path):
       for file in files:
           if file.endswith('.txt'):
               filename.append(file)
filename=filename[0:]

for filenames in filename:
    old_name = filenames
    new_name = '/san/jobA/job1/2022p0045/Dimitris/c30/run_c30/polutindata.txt'
    os.renames(old_name, new_name)
    p1=s.run(["p7 -v 4.24.1 polut < polutindata.txt"], shell= True)
```

C.7 Channel-mapping script

```
import pandas as pd

df=pd.read_csv('C:/Users/dimit/Desktop/Data processing/channels.txt')
df2=df.drop(df.index[0:82])
df2=df2.drop(df2.index[30:36])
df2=df2.drop(df2.index[60])

df2=pd.melt(df2)
df3= df2['value'].str.split(' ', expand=True)
df3=df3.drop(0, axis=1)
df3=df3.drop(2, axis=1)
df3=df3.drop(3, axis=1)
df3=df3.drop(4, axis=1)
df4=df3.T

new_header = df4.iloc[0] #grab the first row for the header
df4 = df4[1:] #take the data less the header row
df4.columns = new_header #set the header row as the df header
df4.fillna("",inplace=True)

df4=pd.melt(df4)
df4['value'] = pd.to_numeric(df4['value'])
df4.dropna(subset=['value'], inplace=True)

# index=df4.index
# zeros=df4['value']==0
# print(zeros)
# zeros_indices=index[zeros]
# zeros_indices_list=zeros_indices.tolist()
# print(zeros_indices_list)

df4=df4.drop([19, 90, 94, 98, 102, 106, 110, 180, 184, 258, 262, 336, 340, 413, 417, 479, 554, 629, 704, 779, 783, 787, 858, 933, 1008, 1083, 1158, 1233,
1308, 1383, 1454, 1458, 1462, 1529, 1604, 1679, 1754, 1827, 1831, 1900, 1904, 1973, 1977, 2046, 2050, 2111, 2115, 2119, 2123, 2127, 2131, 2186, 2308, 2358,
2362, 2366, 2370, 2374, 2378, 2428, 2432, 2498, 2502, 2568, 2572, 2638, 2642, 2713, 2788, 2863, 2938, 3003, 3007, 3011, 3078, 3153, 3228, 3303, 3378, 3453,
3528, 3603, 3678, 3682, 3686, 3763, 3838, 3913, 3988, 4063, 4067, 4143, 4147, 4223, 4227, 4303, 4307, 4383, 4387, 4391, 4395, 4399, 4403, 4483])

df4=df4.rename(columns = {df4.columns[0]:'XChannel'})
df4=df4.rename(columns = {df4.columns[1]:'Xvalue'})

df4=df4.reset_index(drop=True)
print(df4)
df4.to_csv('channelmap.csv')
```

C.8 Data transformations (fix_c25-30) script

```
import pandas as pd
import os
import sys

path='/san/jobA/job1/2022p0045/Dimitris/c30/fix_c30'

cycles=[]
filename=[]
for root, dirs, files in os.walk(path):
       for file in files:
           if file.startswith('output'):
               filename.append(file)

for filenames in filename:
    df=pd.read_csv(filenames,  header=None, sep='\n')

    # 1)split into 5 dataframes

    df1 = df.iloc[:77,:]
    # print(df1)
    df2 = df.iloc[79:156,:]
    # print(df2)
    df3 = df.iloc[157:234,:]
    # print(df3)
    df4 = df.iloc[235:312,:]
    # print(df4)
    df5 = df.iloc[313:,:]
    # print(df5)
    dfP = df.iloc[1]
    # print(dfP)

        # 2)drop needless lines in each dataframe

    new_dfs=[]
    for df in [df1, df2, df3, df4]:
        df_new= df.reset_index(drop=True)
        df_new=df_new.drop([0,1,2,3,4,5,6,7,8,9,10,41,42,43,44,45,46])
        df_new=pd.melt(df_new)
        df_new= df_new['value'].str.split(' ', expand=True)
        df_new=df_new.drop(0, axis=1)
        df_new=df_new.drop(2, axis=1)
        df_new=df_new.drop(3, axis=1)
        df_new=df_new.drop(4, axis=1)
        df_new=df_new.T
        new_header = df_new.iloc[0] #grab the first row for the header
        df_new = df_new[1:] #take the data less the header row
        df_new.columns = new_header #set the header row as the df header
        df_new.fillna("",inplace=True)
        df_new=pd.melt(df_new)
        df_new['value'] = pd.to_numeric(df_new['value'])
        df_new.dropna(subset=['value'], inplace=True)
        df_new=df_new.reset_index(drop=True)
        df_new=df_new.drop(index=[4, 9, 10, 11, 12, 13, 14, 24, 25, 36, 37, 49, 50, 63, 64, 78, 92, 106, 120, 134, 135, 136, 152, 168, 184, 200, 216, 232, 248,
        264, 278, 279, 280, 294, 308, 322, 336, 349, 350, 362, 363, 374, 375, 385, 386, 391, 392, 393, 394, 395, 396, 401, 402, 407, 408, 409, 410, 411, 412,
        417, 418, 428, 429, 440, 441, 453, 454, 467, 481, 495, 509, 523, 524, 525, 539, 555, 571, 587, 603, 619, 635, 651, 667, 668, 669, 683, 697, 711, 725,
        739, 740, 753, 754, 766, 767, 778, 779, 789, 790, 791, 792, 793, 794, 799])

        # 3)Fix indexes and split into dataframes

        df_new['value'] ==df_new['value'].round(decimals=5)
        df_new=df_new.reset_index(drop=True)
        df_new=df_new.rename(columns = {df_new.columns[0]:'Channel'})
        df_new['Channel'] = pd.to_numeric(df_new['Channel'])
        df_new=df_new.rename_axis('index')
        df_new=df_new.reset_index(drop=True)
        new_dfs.append(df_new)

    for i in range(len(new_dfs),):
        globals()[f"df{i}"] = new_dfs[i]
```

```
    # 4)Apply multipliers and channelmap individually

df0['value']=10**-3*df0['value']
df1['value']=10**-2*df1['value']
df2['value']=10**-2*df2['value']
df3['value']=10**-4*df3['value']

df6=pd.read_csv('/san/jobA/job1/2022p0045/Data_processing/channelmap.csv')

df0= df6.join(df0)
df0=df0.drop(['Channel', 'XChannel', 'Unnamed: 0'], axis=1)
df0 = df0.sort_values(by = ['Xvalue'])
df0=df0.rename(columns = {df0.columns[0]:'Channel'})
df0=df0.reset_index(drop=True)

df1= df6.join(df1)
df1=df1.drop(['Channel', 'XChannel', 'Unnamed: 0'], axis=1)
df1 = df1.sort_values(by = ['Xvalue'])
df1=df1.rename(columns = {df1.columns[0]:'Channel'})
df1=df1.reset_index(drop=True)

df2= df6.join(df2)
df2=df2.drop(['Channel', 'XChannel', 'Unnamed: 0'], axis=1)
df2 = df2.sort_values(by = ['Xvalue'])
df2=df2.rename(columns = {df2.columns[0]:'Channel'})
df2=df2.reset_index(drop=True)

df3= df6.join(df3)
df3=df3.drop(['Channel', 'XChannel', 'Unnamed: 0'], axis=1)
df3 = df3.sort_values(by = ['Xvalue'])
df3=df3.rename(columns = {df3.columns[0]:'Channel'})
df3=df3.reset_index(drop=True)


    # 5)Add temperature and power

df5=df5.reset_index(drop=True)
df5=pd.melt(df5)
df5= df5['value'].str.split(' ', expand=True)
temp=df5.iloc[10][8]
power=df5.iloc[7][28]
dft=['Temperature',temp]
dfP=['Power',power]
df4 = pd.DataFrame(dft, index= ['Channel', 'value'])
df5 = pd.DataFrame(dfP, index= ['Channel', 'value'])
df4=df4.T
df5=df5.T


    # 7)Merge together and fix formatting

df=pd.concat([df5, df4, df0, df1, df2, df3])
df=df.reset_index(drop=True)
df=df.T
new_header = df.iloc[0]
df = df[1:]
df.columns = new_header


f1=filenames.replace('output_','')
f1=f1.replace('.dat.txt', '')
df.insert(0, "Distribution", [f1], True)
df.to_csv(f1+'.csv', index=False)
```

C.9 Reading decay ratio and frequency (read_decay _c25-30)script

```
import pandas as pd
import os
import sys

path='/san/jobA/job1/2022p0045/Dimitris/c30/add_decay_c30'
# path='\\svstsmb\job1\2022p0045\Dimitris\c30\add_decay_c30'

df=pd.read_csv('/san/jobA/job1/2022p0045/Dimitris/c30/add_decay_c30/Resultat_c30.txt',  header=None)
# df=pd.read_csv('//svstsmb/job1/2022p0045/Dimitris/c30/add_decay_c30/Resultat_c30.txt',  header=None)


df=pd.melt(df)
df1=df.loc[df.value.str.contains('West')]
df=df.drop([24,25,110,111,133,134,141,142,158,159,184,185,222,223,227,228,235,236,282,283])

df2=df[df.value.str.contains('_')]
df2=df2.drop(['variable'], axis=1)
df2=df2.rename(columns = {df2.columns[0]:'Distribution'})
df3=df[df.value.str.contains('DEC')]
df3=df3.drop(['variable'], axis=1)
df3=df3.rename(columns = {df3.columns[0]:'Decay ratio'})
df3['Decay ratio'] = df3['Decay ratio'].astype('str').str.extract('([0-9]+[,./]*[0-9]*)')
df4=df[df.value.str.contains('FREQ')]
df4=df4.drop(['variable'], axis=1)
df4=df4.rename(columns = {df4.columns[0]:'Frequency'})
df4['Frequency'] = df4['Frequency'].astype('str').str.extract('([0-9]+[,./]*[0-9]*)')


list_dfs=[df2, df3, df4]

[df.reset_index(drop=True, inplace=True) for df in list_dfs]

merged= pd.concat([df2, df3, df4], axis=1, join='inner')
merged.to_csv('/san/jobA/job1/2022p0045/Dimitris/c30/c30_final/decay_ratio.csv', index=False)
```

C.10 Merging all processed data (merge _c25-30) script

```
import pandas as pd
import os
import sys

path='/san/jobA/job1/2022p0045/Dimitris/c30/add_decay_c30'


df=pd.read_csv('/san/jobA/job1/2022p0045/Dimitris/c30/add_decay_c30/Resultat_c30.txt',  header=None)

cycles=[]
filename=[]
for root, dirs, files in os.walk(path):
        for file in files:
            if file.endswith('.csv'):
                filename.append(file)

dataframes=pd.read_csv(filename[0],  header=None, sep='\n')
for filenames in filename[1:]:
    df=pd.read_csv(filenames,  header=None, sep='\n')
    df= df.iloc[1:].reset_index(drop=True)
    dataframes=pd.concat([dataframes,df], axis=0, )
df=pd.melt(dataframes)
df= df['value'].str.split(',', expand=True)
df.columns=df.iloc[0]
df=df[1:]

df.to_csv('/san/jobA/job1/2022p0045/Dimitris/c30/c30_final/merge.csv', index=False)
```

C.11 Removing distributions without decay ratio and frequency (c25-30_final) script

```
import pandas as pd
import os
import sys

path='/san/jobA/job1/2022p0045/Dimitris/c30/c30_final'
# path='//svstsmb/job12022p0045/Dimitris/c30/c30_final'

df=pd.read_csv('/san/jobA/job1/2022p0045/Dimitris/c30/c30_final/decay_ratio.csv',  header=None)
# df=pd.read_csv('//svstsmb/job1/2022p0045/Dimitris/c30/c30_final/decay_ratio.csv',  header=None)
df.columns=df.iloc[0]
df=df[1:]


df2=pd.read_csv('/san/jobA/job1/2022p0045/Dimitris/c30/c30_final/merge.csv',  header=None)
# df2=pd.read_csv('//svstsmb/job1/2022p0045/Dimitris/c30/c30_final/merge.csv',  header=None)
df2.columns=df2.iloc[0]
df2=df2[1:]



final=pd.merge(df,df2, on="Distribution", how="left")
final=final.drop("Distribution", axis=1)
final.to_csv('final_c30.csv', index=False)
```

C.12 Merged and Verification datasets for the ML Model script

```
import pandas as pd
import os
import glob

files=glob.glob("/san/jobA/job1/2022p0045/Dimitris/Merge/*.csv")

df=[]
for file in files:
    df2=pd.read_csv(file)
    df.append(df2)
df=pd.concat(df)
df= df.reset_index(drop=True)

df.to_csv('Merged.csv', index=False)
```