Nuclear and Radiation Department, Faculty of Engineering, Alexandria University

July 2024

DEEP LEARNING MODEL APPLIED TO LOSS OF COOLANT ACCIDENT ANALYSIS

By

Youssef Ibrahim Badr

Mai Tarek Hamdy

Ibrahim Mohamed Mohsen

Ahmed Hesham Nazef

Alaa Ahmed El-Gendy

Hassan Ahmed Radwan

Rawda Ayman Mohamed

Reyad Mohamed Reyad

Mohamed Nazeh Abdel-Badee

A Graduation Project submitted

in satisfaction of Bachelor's degree requirements

Under Supervision of

Dr. Tarek Nagla

Acknowledgment

We extend our gratitude to our thesis supervisor, Dr. Tarek Nagla, for his unwavering support, guidance, and provision of invaluable resources throughout our research journey. His dedication has been instrumental in shaping our work.

Our sincere appreciation also goes to the head of the Nuclear and Radiation Engineering department, Dr. Mohamed Yasser Khalil, whose mentorship and insight have illuminated our path and enriched our learning experience.

We are deeply indebted to Dr. Alya Badawi for her exceptional technical support and expertise, particularly in the use of WSC, Inc. Simulator for data acquisition. Her assistance was crucial in the preparation and analysis of our thesis data.

We would be remiss not to acknowledge our teaching assistant, Eng. Ahmed Mostafa, whose provision of valuable resources and guidance in the practical aspects of machine learning has been indispensable to our progress.

Words fall short in expressing our immense gratitude to all the professors in our department. Their passion, knowledge, and unwavering support have not only made this thesis possible but have also ignited within us a fervor for academic excellence and scientific inquiry.

The completion of this thesis stands as a testament to the solid academic and scientific foundation instilled in us by our esteemed faculty. Their collective wisdom and mentorship have been the cornerstone of our academic journey, propelling us towards this significant milestone.

Abstract

The word safety can be defined as "*freedom from hazard or danger*". In any technological society, no activity has zero risk. Hence, no such absolute safety exists. In order to achieve a high level of safety, postulated accidents are considered in the design phase of the reactor to ensure functionality of withstanding such accidents. Those are known as design basis accidents (DBAs). The goal in reactor safety is to prevent accidents from occurring. The reactor systems need to be designed, constructed, and operated so that the chances of a malfunction or operational error are very small. A primary DBA that is extensively studied is the loss of coolant accident (LOCA), where it receives thorough analysis and consideration throughout the work.

Deep learning or deep neural networks have revolutionized several fields, including image recognition, speech recognition, natural language processing, robotic due to their predictability and ability to identify patterns and relations between input parameters through weights assignments and backwards propagations. Deep neural networks, due to their ability to train and detect the pattern through a relatively small dataset, were also considered for this work. The harnessing of such techniques can result in not only enhanced but also more accurate predictions in nuclear field specially designs safety related aspect.

This work explores the deep learning techniques through Long Short-Term Memory (LSTM) architecture, creating a foundation model for time-series dataset for a loss of coolant accident to predict break sizes. The choice of LSTM architecture is because of its abilities to deal with time-series problems. For this work various break sizes, from 0.5% to 100% break, time variant parameters were collected from WSC, Inc 1400 MWe Generic simulator for the Pressurized Light Water Reactors with two circulation loops. On practicing, the neural networks trained on parameters such as loop temperature, pressure, containment pressure and Boron concentration. After training the LSTM model with these parameters and its validation set, the model Mean Absolute Error (MAE) of 5.185, Mean Squared Error (MSE) of 76.50, Root Mean Squared Error (RMSE) of 7.953, R^2 of 0.89 and Accuracy of 80.684% within a tolerance of 15%.

Table of Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
DL	Deep Learning
DBA	Design Basis Accident
NN	Neural Networks
ML	Machine Learning
LSTM	Long Short-Term Memory
LOCA	Loss of Coolant Accident
AOOs	Anticipated operational occurrences
PWR	Pressurized Water Reactor
BWR	Boiling Water Reactor
PCT	Peak Cladding Temperature
ECCS	Emergency Core Cooling System
HPSI	High-Pressure Safety Injection
CCPS	Centrifugal Charging Pumps
CSAU	Code Scaling, Applicability and Uncertainty
PIRT	Phenomena Identification and Ranking Table

SETs	Separate Effect Tests
IETs	Integral Effect Tests
FSLOCA	Full Spectrum LOCA
CNN	Convolutional Neural Networks
MSE	Mean Square Error
CFNN	Cascaded Fuzzy Neural Network
OPR1000	Optimized Power Reactor
MAAP	Modular Accident Analysis Program
NARX	Nonlinear Autoregressive with Exogenous inputs
RNNs	Recurrent Neural Networks
SVM	Support Vector Machine
CUDA	Compute Unified Device Architecture
RMSE	Root Mean Squared Error
MAE	Mean Absolutes Error
ReLU	Rectified Linear Unit
Adam	Adaptive Moment Estimation
LR	Learning Rate

SGD	Stochastic Gradient Descent
FNN	Feedforward Neural Networks
MLP	Multiple-Layer Perceptron
NumPy	Numerical Python
GAN	Generative Adversarial Network
PCTran	Personal Computer Transient Analyzer
GUI	Graphical User Interface
GPWR	Generic PWR Simulator
WSC	Western Services Corporations
ICs	Initial Conditions
CSV	Comma-Separated Values
LOFA	Loss of Flow Accident

Chapter I
Introduction

In the fast-evolving era of Artificial Intelligence (AI), Deep Learning (DL) stands as a cornerstone technology, revolutionizing how machines understand, learn, and interact with complex data. At its essence, Deep Learning AI mimics the intricate and interconnected neural networks (NN) of the human brain, enabling computers to autonomously discover patterns and make decisions from vast amounts of unstructured data. This transformative field has propelled breakthroughs across various domains, from computer vision and natural language processing to healthcare diagnostics and autonomous driving. Deep Learning has reshaped industries, pushing the boundaries of what's possible in AI, and paving the way for a future where intelligent systems can perceive, comprehend, and innovate autonomously (Chollet, 2021).

Deep Learning, being a subset of Machine Learning (ML) and AI, has evolved over the past decade for lots of reasons. One reason in particular is the abundance of lots of data which can be harnessed to train such a model to mimic the human brain behavior of pattern recognitions and complex correlations between multiple inputs.

Deep learning algorithms are neural networks that are modeled after the human brain. For example, a human brain contains millions of interconnected neurons that work together to learn and process information. Similarly, deep learning neural networks, or artificial neural networks, are made of many layers of artificial neurons that work together inside the computer. Artificial neurons are software modules called nodes, which use mathematical calculations to process data. Artificial neural networks are deep learning algorithms that use these nodes to solve complex problems.

Deep Learning has made lots of advancements such as, fraud detection, computer vision, natural language processing, autonomous vehicles and vocal AI. This is because Deep learning algorithms give better results when trained on large amounts of high-quality data. Outliers or mistakes in input dataset can significantly affect the deep learning process. For instance, if we take a model that detects the types of animals in images as an example, the deep learning model might classify a turtle as air-plane if non-animal images were accidentally introduced in the dataset.

Deep Learning algorithms can provide multiple advantages such as:

(1) High Accuracy
(2) Scalability
(3) Flexibility
(4) Continual Improvement
(5) Hidden relationships and pattern discovery
(6) Unsupervised learning
(7) Automated Feature Engineering. Where, Deep Learning algorithms can automatically discover and learn relevant features from data without the need for manual feature engineering.

“Better safety equates to better health”. The term reactor safety describes the task of operating nuclear power plants free of accidents and unintentional releases of radioactive and nuclear material. With safety being the most important factor in reactor design and operation, improvement in every safety aspect becomes a must for a better, efficient and safer reactor operation.

One of the main safety aspects known in reactors design is the Design Basis Accident (DBA), where, a reactor is specifically designed to withstand a postulated accident ensuring reactor safe shutdown, core residual heat removal and continuous cooling and the prevention of any release of nuclear material.

Deep Learning techniques advantages, mentioned above, makes them a strong candidate for safety related problems. This work harness Deep Learning techniques, mainly Long Short-Term Memory (LSTM), to predict LOCA break sizes with time variant input parameters.

1.1 Loss of Coolant Accident (LOCA)

In the design of nuclear power plants, it is required that various operational occurrences are considered and that the consequences of such occurrences are analyzed so that suitable mitigating systems can be designed. Normal operation of the plant can be defined as operation within specified operational limits and conditions. Anticipated operational occurrences (AOOs) are operational processes deviating from normal operation which are expected to occur at least once during the lifetime of a facility but which, in view of appropriate design provision, do not cause any significant damage to items important to safety or lead to accident conditions. The accident condition can be simply defined as deviations from normal operation more severe than anticipated operational occurrences. Accident conditions can be divided into DBAs and severe accidents (Bays et al., 2019).

For the accident conditions, there are acceptance criteria which must be fulfilled. For the design basis accidents, the most fundamental acceptance criterion is typically that there should be no or at most very limited radiological consequences to the public. However, in order to fulfil this criterion, there will be a number of other acceptance criteria related to the different safety systems of the reactor. How these criteria are formulated will depend on the general design of the reactor and the various physical phenomena of importance to the occurrence of a particular design basis accident.

One of the DBAs for nuclear reactors is the loss of coolant caused by the failure of a large coolant pipe. Loss of coolant accident (LOCA) can occur in any water cooled-water moderated reactor. It is postulated in Pressurized Water Reactors (PWR), Boiling Water Reactors (BWR) and CANDUs, and every reactor design essentially.

This work studies closely the LOCA in the PWR designs and its time-series behavior. For PWR, illustrated in Figure I-1, the initiating event of the design basis accident is the double-ended guillotine break of one of the large coolant pipes between the reactor vessel and the main circulation pump. In order to mitigate the consequences of this break, it is necessary that the reactor has several emergency core cooling systems. This postulation assumes that 100% of the pipe is broken, hence, the term guillotine break. This means that the primary coolant can no longer be provided for the reactor core cooling (Glasstone & Sesonske, 1994).

LOCAs fall into several categories depending on the size of the postulated break in the primary coolant circuit. Smaller breaks, which could lead to minor fuel cladding damage at worst, are considered in 1.2.1   For the design basis accident, however, a "guillotine" double ended break is the postulated break, as can be seen in Figure I-2 and Figure I-3. The result of such a break, the primary system pressure would drop and almost all the reactor water would be expelled into the containment. The drop in pressure resulting from such a LOCA would actuate the protection system and the reactor would be tripped. The fission chain reaction in the core would thus be terminated, as it would in any case; because of the loss of coolant, in the case of PWR, is also a loss of moderator. Nevertheless, heat would continue to be released at a high rate from the sources, about 1.5% of the rated thermal power, after an hour of the reactor trip. This is mainly due to the decay heat of the minor actinides contained in the fuel.

The main concern in such an accident is the depressurization of the system and the Average Peak Cladding Temperature (PCT), where the coolant not only leak but it also evaporates leaving the core without heat removal. The various Emergency Core Cooling System (ECCS) subsystems must then provide sufficient cooling in time to minimize overheating and fuel cladding damage in order to maintain reactor core integrity and prevent any fuel failure. Thus, achieving the safety requirements (NRC, n.d.).

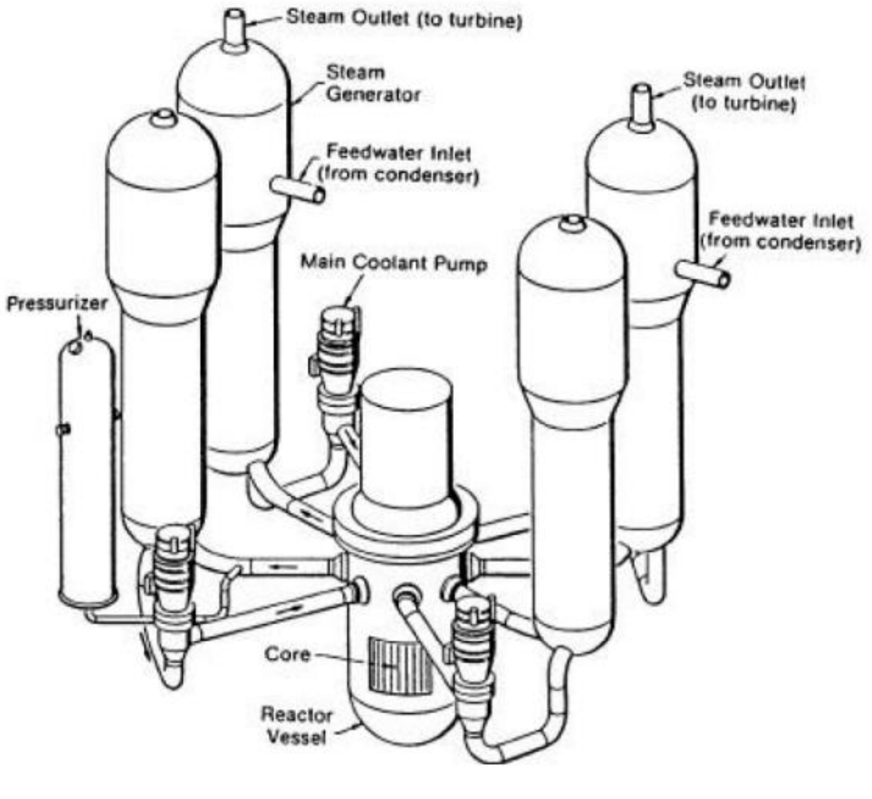

Steam Outlet (to turbine)
Steam Generator
Steam Outlet (to turbine)
Feedwater Inlet (from condenser)
Feedwater Inlet (from condenser)
Main Coolant Pump
Pressurizer
Core
Reactor Vessel

Figure I-1. PWR Arrangement

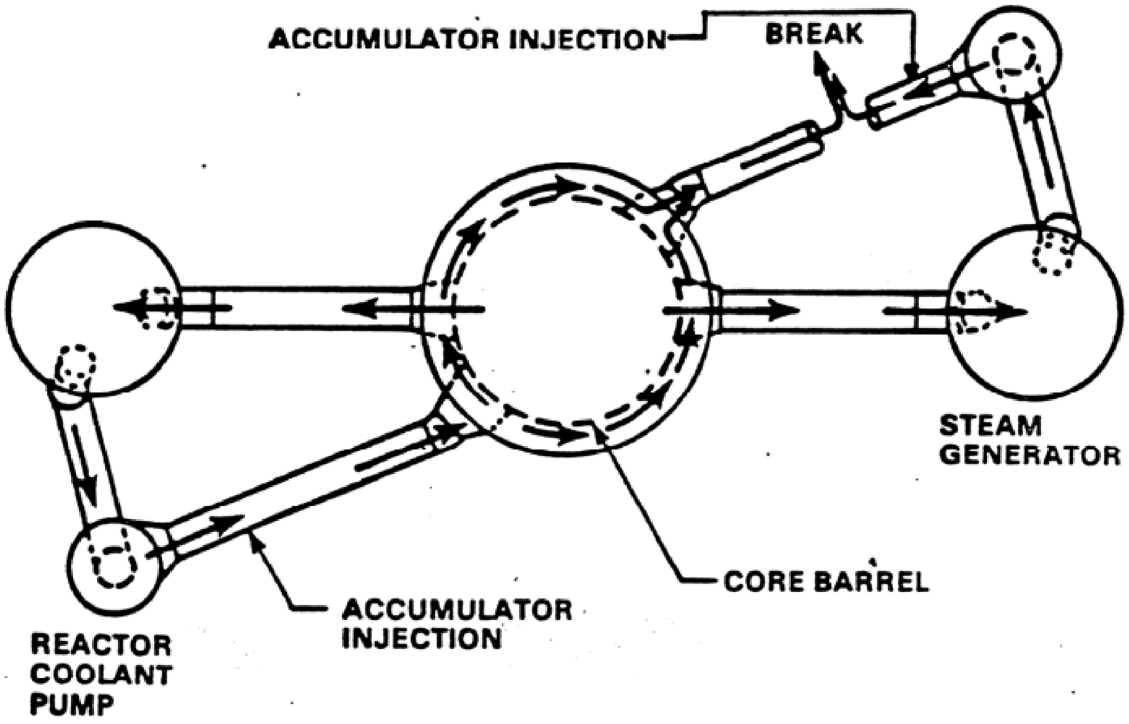


Figure I-2. Cold Leg Break Steam Flow Path Top View. Courtesy of Westinghouse.

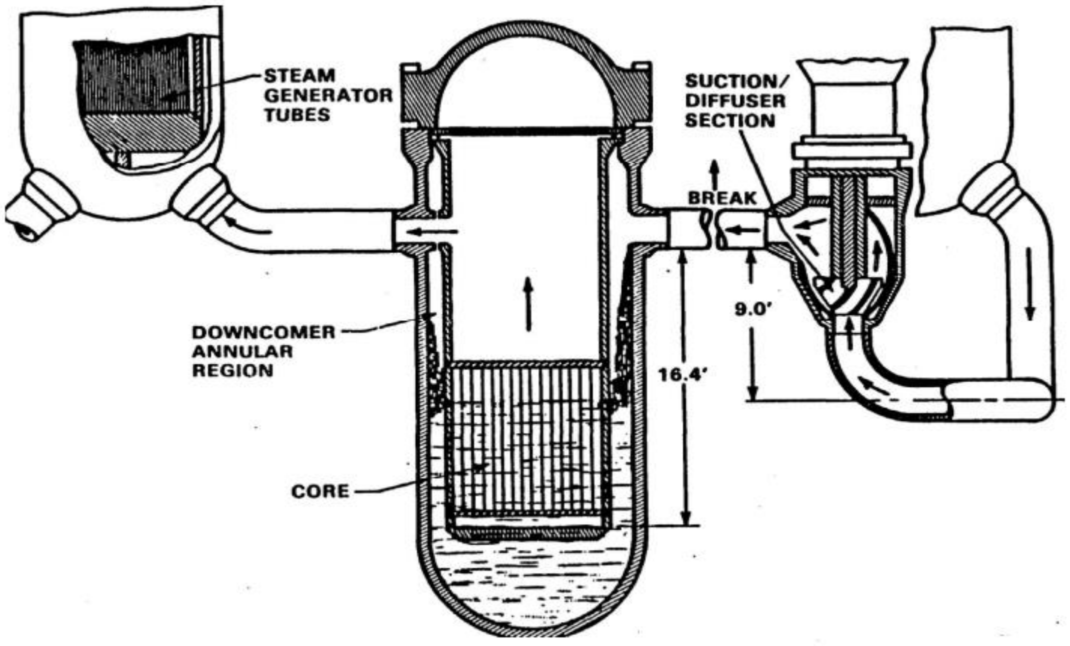


Figure I-3. Cold Leg Break Steam Flow Path Schematic Drawing. Courtesy of Westinghouse.

1.2 PWR Break Categorization

1.2.1 PWR Small Break LOCA

Breaks with flow areas typically less than 1 $\boldsymbol{ft^2}$ and greater than 3/8 $\boldsymbol{in}$. in diameter span the category of small breaks. A small break is sufficiently large that the primary system depressurizes to the high-pressure safety injection set point and a safety injection or "S" signal is generated, automatically starting the High-Pressure Safety Injection (HPSI) system. Breaks smaller than 3/8-inch in diameter do not depressurize the reactor coolant system because the reactor charging flow can replace the lost inventory. This, in fact, is due to the Centrifugal Charging Pumps (CCPs) used in the reactor primary coolant circuit. CCPs are powerful pumps that provide a fluid flow based on a desired pressure. The control rods shut down the reactor such that only decay heat is generated in the core.

The limiting break is one that is large enough that the high-pressure safety injection system cannot make up the mass loss from the reactor system but small enough that the reactor system does not quickly depressurize to the accumulator set point. This combination of circumstances leads to a core uncovering. The limiting small-break LOCA is determined by the inter-play between core power level, the axial power shape, break size, the high-head safety injection performance, and the pressure at which the accumulator begins to inject. For Westinghouse plants, the limiting breaks are typically in the 2–4-inch range.

1.2.2 PWR Large Break LOCA

This is the regarded break size in the design basis accident in the PWR. It is a double-ended guillotine break in a cold leg between the reactor coolant pump and the reactor vessel. The time series of this event can be divided into three main phases (Glasstone & Sesonske, 1994):

(1) Blowdown Phase.
(2) Refill Phase.
(3) Reflood phase.

1.2.2.1 Blowdown Phase

The blowdown period (0-30 s) occurs as a result of a break in the coolant system through which the primary coolant is rapidly expelled, as demonstrated in Figure I-4. Blowdown proceeds in two stages, namely, subcooled and saturated. During the initial subcooled blowdown stage, the primary system pressure drops rapidly to the vapor (or saturation) pressure of the water at the existing temperature. Within a fraction of a second after the break, the core voids and goes through departure from nucleate boiling. The negative void reactivity rapidly shuts down the core. With the diminished cooling and the redistribution of stored energy in the fuel, the cladding heats up. Interactions between the pump and the break dynamics cause intermittent flow reversals. In saturated blowdown stage, steam voids are formed, and the steam-water mixture flows out through the break; this stage would probably continue for roughly 15 to 20 s until the system pressure becomes approximately equal to the containment pressure. The two-phase flow, which is comparable to a steam-water mixture flowing through a pipe is called "choked" flow. The primary system pressure rapidly decreases, and the high-pressure safety injection begins, but most of this flow is lost out of the break. Injection from the cold-leg accumulators begins but much of the injected flow is swept around the downcomer, into the broken-loop cold leg and out the break.

As the blowdown progresses, an increasing amount of the accumulator-injected coolant stays in the downcomer and some water begins to enter the lower plenum. The average PCT during the blowdown phase of a large-break LOCA is approximately 1500°F (815°C) and the PCT at 95% confidence level is about 1750°F (954°C), assuming a loss-of-offsite power and the worst single failure assumption for the emergency core cooling system. As the saturated blowdown proceeds, the fraction of liquid phase in the coolant would decrease until a froth of water and steam remained. The level of the froth would fall, leaving the upper part of the core dry. Heat removal from the core would then be primarily by radiation to the surrounding structure. The temperature of the cladding might then rise to the point (over 980°C) at which failure of some of the hotter rods could occur as a result of the loss of strength of the hot zircaloy. The heat liberated in the zirconium-water reaction would contribute to the temperature increase.

1.2.2.2 Refill Phase

The refill period occurs between 30 and 40 seconds following the start of the LOCA. pressure has decreased to a level at which the low-pressure injection system activates and begins to inject water into the system. The accumulators would inject borated water into the reactor vessel through the intact cold legs or directly into the downcomer. This would provide some cooling of the fuel, but in the initial stage most of the water would probably be expelled as a steam-water mixture through the pipe break. This phenomenon is known as ECCS Water Bypass. In ECCS water bypass, the blowdown flow resists the injected water from the accumulator and other ECCS subsystems forcing the water to flow out of the break.

The lower plenum begins to fill with accumulator water as coolant bypass diminishes. While refilling of the lower plenum is underway, however, the core heats up in a near adiabatic mode due to decay heat. Some fuel rods balloon and burst, causing blockage of some of the flow channels during refill. There is very little cooling of the fuel during the refill phase; such cooling as does occur is mainly by convection to the steam-water mixture.

1.2.2.3 Reflood Phase

The reflood period occurs between 40 and 200 seconds. It begins at the time when the lower plenum has filled, and the core begins to refill. Water injected by the accumulators fills the downcomer and creates the driving head for refilling the core. As the water rose, it would come into contact with the hot fuel cladding. The fuel rods will then be covered with a vapor film layer which water cannot penetrate. This provides a film boiling condition. The temperature near the core bottom soon drops to the point at which water can penetrate the steam film and nucleate boiling could occur. The temperature of the cladding could then decrease fairly rapidly. The accumulator tanks in a PWR would be empty about a minute after the pipe break, but water from other ECCS sources will permit the core to be recovered.

The average reflood PCT during this period is approximately 1680°F (915°C).

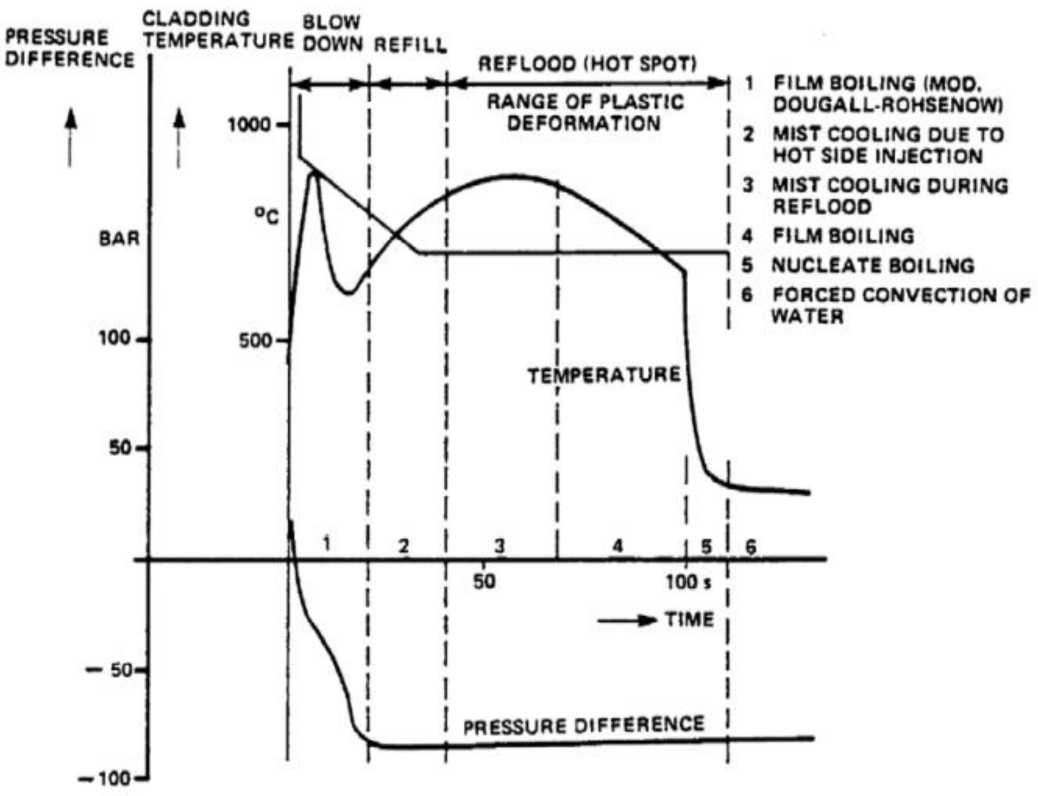


Figure I-4. Double-end Guillotine LOCA Time-Series Behavior through its Phases

Chapter II
Literature Review

2.1 Best Estimate Evaluation Existent Codes

A best-estimate model is a model that should provide a realistic calculation of the important parameters which are associated with a particular phenomenon to the practical degree with the currently available data and knowledge of the phenomenon. Since the 1988 amendment of the 10 CFR 50.46 rule, Westinghouse has been developing and applying realistic or best estimate methods to perform LOCA safety analyses (Frepoli, 2008).

A realistic analysis requires the execution of various realistic LOCA transient simulations where the effect of both model and input uncertainties are ranged and propagated throughout the transients. Most of the implementations, including Westinghouse, follow the Code Scaling, Applicability and Uncertainty (CSAU) methodology (Westinghouse, 2009). A key step in a best-estimate analysis is the assessment of uncertainties associated with physical models, data, and plant initial and boundary condition variabilities.

2.1.1 CSAU Methodology

The Code Scaling, Applicability and Uncertainty (CSAU) methodology process is divided in three main elements:

(1) The scenario is broken down into relevant-time periods (e.g., blowdown, refill, and reflood for large break scenario) and the nuclear power plant broken down into relevant regions (e.g., fuel rod, core, lower plenum). Then potentially important phenomena/processes are identified for each time period and region. An expert's panel performs ranking and document basis for consensus. Results are compiled in the phenomena identification and ranking table (PIRT). The PIRT is a critical element of CSAU-based methodologies. It is designed to focus the prioritization of code assessment and facilitate the decisions on physical model and methodology development.

(2) The assessment of the code. An assessment matrix is established where separate effect tests (SETs) and integral effect tests (IETs) are selected to validate the code against the important

phenomena identified in the PIRT. The code biases and uncertainties are established, and the effect of scale determined. A key output from this element is the establishment of probability distributions and biases for the contributors identified in Element (1). In addition to the generation of probability distributions, and perhaps even more important, this element required a thorough 3 assessment of the code's ability to correctly predict all the dominant physical processes during the transient. This leads to the adequacy decision of the evaluation model.

(3) The actual implementation stage of the methodology. Sensitivity and uncertainty analyses are performed here. This element is probably the most straight forward of all the elements.

While Elements (1) and (2) of the CSAU are generally applied in various form, the techniques used to combine the uncertainties evolved over the last few years. The CSAU originally suggested the use of response surfaces methods, however shortcomings were soon identified in early implementation. Direction in recent years is toward direct Monte Carlo methods and the use of nonparametric statistics. This generated a debate in the industry since the regulations are not directly suited to a statistical framework.

2.1.2 Westinghouse WCOBRA/TRAC Code

The COBRA/TRAC computer program is a thermal-hydraulic that has been developed to predict the thermal-hydraulic response of nuclear reactor primary coolant systems to Large Break LOCA accidents and other anticipated transients (Thurgood et al., n.d.). It is derived from the merging of COBRA-TF and TRAC-PD2.

The COBRA-TF computer code provides a two-fluid, three-field representation of two-phase flow. Each field is treated in three dimensions and is compressible. Continuous vapor, continuous liquid and entrained liquid drop are the three fields. The conservation equations [continuity equation, motion equation and energy conservation (from the first law of thermodynamics).] for each of the three fields and for heat transfer from and within the solid structures in contact with the fluid are solved using a semi-implicit, finite-difference numerical technique on an Eulerian mesh (volume nodes tracker).

COBRA-TF features extremely flexible noding for both the hydrodynamic mesh and the heat transfer solution. This flexibility provides the capability to model the wide variety of geometries encountered in vertical components of nuclear reactor primary systems.

TRAC-PD2 is a systems code designed to model the behavior of the entire reactor primary system/circuit. It features special models for each component in the system. These include accumulators, pumps, valves, pipes, pressurizers, steam generators, and the reactor vessel. With the exception of the reactor vessel, the thermal-hydraulic response of these components to transients is treated with a five-equation drift flux representation of two-phase flow. The vessel component of TRAC-PD2 is somewhat restricted in the geometries that can be modeled and cannot treat the entrainment of liquid drops from the continuous liquid phase directly.

The TRAC-PD2 vessel module has been removed and COBRA-TF has been implemented as the new vessel component in TRAC-PD2. The resulting code is COBRA/TRAC. The vessel component in COBRA/TRAC has the extended capabilities provided by the three-field representation of two-phase flow and the flexible noding. The documentation of the COBRA/TRAC program consists of five separate volumes. Volume 1 contains a description of the basic three-field conservation equations and constitutive models used in the vessel component (COBRA-TF).

Volume 2 contains the finite-difference equations for the vessel. It describes the numerical techniques used to solve these equations and the coupling between the TRAC-PD2 equations and the COBRA-TF vessel equations.

Volume 3 is the Users' Manual. It contains line-by-line input instructions for COBRA/TRAC and user guidance for application of the code. Volume 4, the Applications Manual, contains the results of simulations run to assess the performance of the code. Volume 5 is a Programmers' Guide. In 1993, Improvements and error corrections to COBRA/TRAC through 100 SET/IET tests led to the development of WCOBRA/TRAC.

Recently, Westinghouse has made several upgrades for the purpose of extending the EM applicability to smaller break sizes. Also, the uncertainty methodology was upgraded to reflect a larger coverage of scenarios. The new EM is called Westinghouse Full Spectrum LOCA (FSLOCA) Methodology and is intended to be applicable to a full spectrum of LOCAs, from small to intermediate break as well as large break LOCAs.

2.1.3 Westinghouse NOTRUMP/LOCTA

Just as the WCOBRA/TRAC case, NOTRUMP/LOCTA is derived from merging both NOTRUMP code and LOCTA code. NOTRUMP/LOCTA is used for small breaks LOCA unlike the WCOBRA/TRAC which is employed for large breaks only.

For small breaks (less than 1 ft^2), the NOTRUMP computer code is employed to calculate the transient depressurization of the reactor coolant system (RCS) as well as to describe the fluid flow mass and energy through the break. The NOTRUMP computer code is a state of art 1-D general network code (well written, commented and structured. Functioning perfectly under any condition, including those not considered when it was tested for its original purpose) incorporating a number of advanced features (Shimeck & Hartz, 2000).

Among these advanced features are:

- Flow regime-dependent drift flux calculations with counter-current flooding limitations.
- Mixture level tracking logic in multiple-stacked fluid nodes.
- Regime-dependent drift flux calculations in multiple-stacked fluid nodes and regime-dependent heat transfer correlations.
- Calculation of thermal non-equilibrium in all fluid volumes.

The NOTRUMP small break LOCA emergency core cooling system (ECCS) evaluation model was developed to determine the reactor coolant system (RCS) response to design basis small break LOCAs, and to address NRC concerns. Typically, the analytical model for RCS is nodalized into volumes interconnected by flow paths. The broken loop is modeled while the intact loops are lumped into a second loop. Transient behavior of the system is determined from the governing conservation equations of mass, energy, and momentum

Thermal analyses are performed with a version of the LOCTA code using the NOTRUMP calculated core pressure, fuel rod power history, uncovered core steam flow and mixture heights as boundary conditions.

All of the LOCTA codes, as well as several other codes derived from the LOCTA models, have the fundamental capabilities for performing a 1-D radial heat conduction solution for a nuclear fuel rod geometry, shown in Figure II-1. The main purpose is to predict the behavior of fuel rods under LOCA conditions. Also, LOCTA has the fundamental intrinsic capability to solve the 1-D heat con

duction problem for cylindrical geometry. It is also possible to adapt the code to solve this problem for similar geometries.

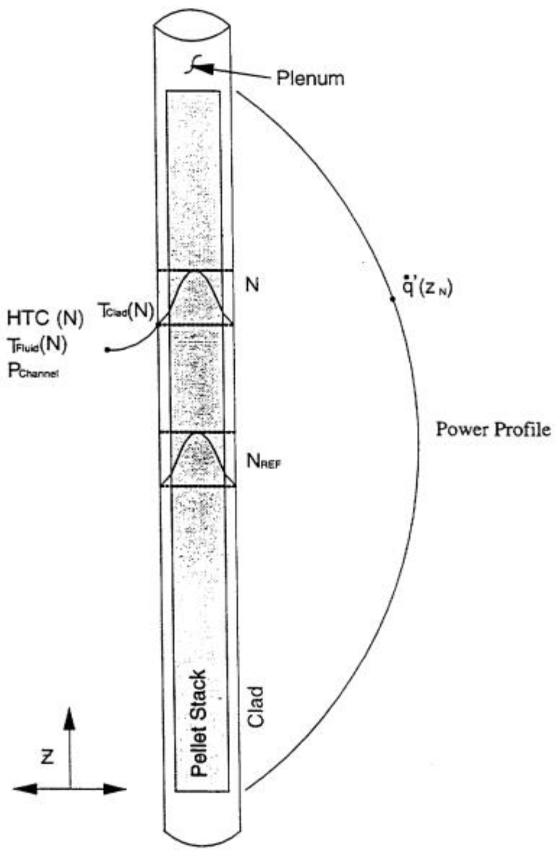


Figure II-1. Fuel rod model for LOCTA-JR.

In order to solve the complete thermal-hydraulic problem for a fuel rod a number of ancillary models are necessary in addition to the heat conduction model. These include models for:

- Thermal mechanical expansion and contraction of the pellet and clad.
- Rod gas (gap) pressure behavior.
- Pellet to-clad gap conductance.
- Models for predicting channel fluid conditions.

To solve a problem, first the transient time-dependent channel fluid conditions of pressure, fluid temperature, clad surface heat transfer coefficient are input for individual axial nodes, along with a normalized rod power curve. Since there are no fluid calculations other than timestep interpolations and obtaining steam properties, the bulk of the problem is simply a standard 1-D radial heat conduction problem which consists of setting up the simultaneous equations for the specified radial nodalization and inverting the matrix to obtain the new temperature distribution and clad temperature.

2.2 Deep Learning Methods Used for LOCA Diagnosis and Analysis

2.2.1 Diagnosis and Prediction for Loss of Coolant Accidents in Nuclear Power Plants Using Deep Learning Methods

As mentioned in literature (She et al., 2021), the hybrid model is proved to be functional, accurate for using post-accident prediction for loss of coolant accident in nuclear power plants, they used a collection of (CNN+ LSTM= ConvLSTM) for this work.

This work has chosen CNN structure to filter LOCA parameters and extract the Important features for prediction, while LSTM has been chosen as LOCA deals with time series databases.

Seventy-five percent of the dataset is used for training purposes. Each LOCA case (0.2, 0.4, 0.6, 0.8, or 1.0 cm^2) is simulated under five kinds of operation status (60, 70, 80, 90, and 100% reactor power levels). With an improved structure, the fault diagnosis model based on ConvLSTM successfully reaches classification accuracy as high as 96%.

Loss values and Mean Square Error (MSE) for all the test cases are kept as low as 10−3, satisfying the accuracy expectation.

Compared to the LSTM model, the CNN-LSTM demonstrated its advantage of multi-feature processing (Choi et al., 2017), which provides a better prediction performance.

2.2.2 Estimation of LOCA Break Size Using Cascaded-Fuzzy Neural Networks

In this study, (Geon Pil Choi, 2017), a cascaded fuzzy neural network (CFNN) model with a machine learning function was utilized to predict the LOCA break size. the CFNN is an extension of the FNN.

The data used were obtained by numerically simulating severe accident scenarios of an optimized power reactor (OPR1000) using Modular Accident Analysis Program (MAAP) code.

In this study, a Takagi-Sugeno type fuzzy model is used in which the “if” part is fuzzy linguistic, while the “then” part is crisp. Therefore, the Takagi-Sugeno type fuzzy inference system does not need a defuzzifier in the output terminal.

The accidents have different break positions and sizes. The inner diameters of the hot-leg and cold-leg pipes are 1.0668 m and 0.762 m, respectively, and the inner diameter of a steam generator tube is 0.0169 m. The break size of the hot leg and cold leg pipes ranges from a minimum of $^{1}/_{400}$ of their guillotine break to a maximum of half of the guillotine break. The break size of SGTR ranges from 11 SGTRs to 210 SGTRs.

The performance results of the CFNN model show that the RMS error decreases as the stage number of the CFNN model increases. In addition, the performance results of the CFNN model produce an RMS error level below 0.7%. Therefore, it is confirmed that the CFNN model can accurately predict the LOCA break size.

2.2.3 Real-time estimation of break sizes during LOCA in nuclear power plants using NARX neural network

This paper, (Saghafi & Ghofrani, 2019), deals with break size estimation of LOCA using a nonlinear autoregressive with exogenous inputs (NARX) neural network.

By adding feedback connections to the architecture of feedforward networks, recurrent neural networks (RNN) are formed, which can deal with temporal input signals. NARX neural networks, as a subcategory of RNNs, have limited feedbacks which come only from the output neurons rather than the hidden neurons.

This database of LOCA scenarios is developed using RELAP5 thermal-hydraulic code. Thirteen scenarios of LOCA with various break sizes (5% - 100% of main pipeline area) in the cold leg of Bushehr NPP, plus a null-transient scenario are modeled using RELAP5 code.

Bushehr NPP is a four loop Russian-designed light water reactor, i.e. VVER-1000. Approximately 70% of transients in the database are selected for training, 15% for validation, and 15% are left for testing of NARX neural network.

The results of NARX neural network for estimation of the untrained cases are in good agreement with the actual break sizes. NARX neural network can accurately find a general solution for break size estimation problem in real-time, using a limited number of training cases.

Chapter III
Machine Learning Theory

Artificial intelligence can be defined as efforts to automate intellectual tasks normally performed by humans. As such, AI is a general field that encompasses both Machine Learning and Deep learning, but that also includes many more approaches that don't involve any learning such as Symbolic AI. Although Symbolic AI proved suitable to solve well-defined, logical problems, such as playing chess, it turned out to be intractable to figure out explicit rules for solving more complex, fuzzy problems, such as image classification, speech recognition, and language translation. A new approach arose to take symbolic AI's place: Machine Learning (Chollet, 2021).

Machine Learning (ML) is subsection of AI-based technologies, as showcased in Figure III-1, that can learn and improve from experience without using programming but accessing data and self-learn. This process starts with previous data or instructions for finding and analyzing the pattern to achieve accurate decisions and predictions. This all happens without human intervention by programming or inserting rules.

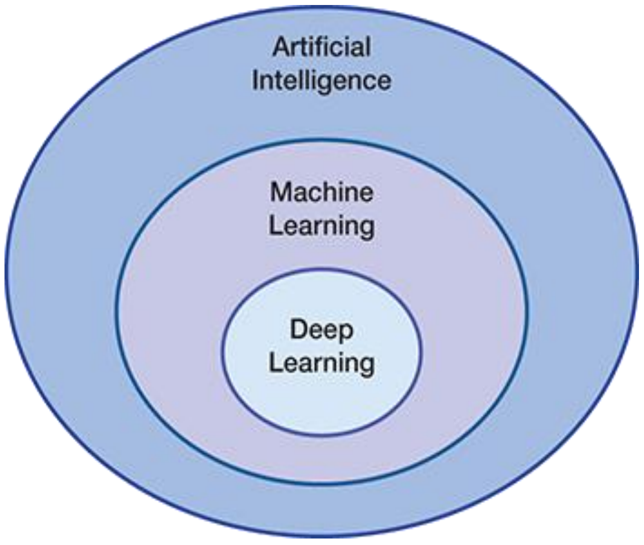


Figure III-1. The relation between AI, Machine Learning (ML) and Deep Learning (DL).

ML is different from computer programming, as computer programs receive rules from the developer to follow during data processing, while a ML system is trained rather than explicitly programmed, as it can distinguish patterns emerging from the data points, making connections between them, and ultimately arriving at conclusions. That’s called “learning process”.

ML models learn by iteratively transforming inputs into outputs and comparing these outputs to expected results. The model adjusts its internal representations based on feedback from this comparison, gradually improving its ability to produce meaningful outputs that align with expectations.

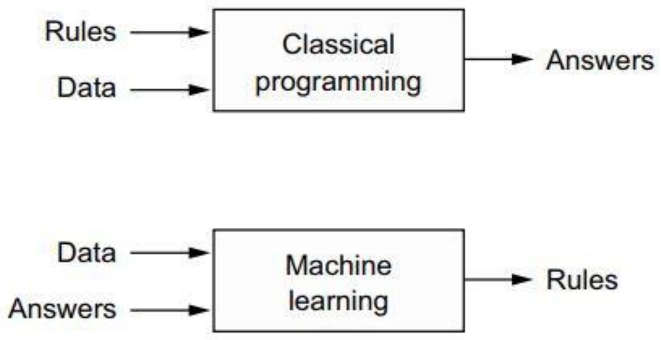


Figure III-2. Deep Learning with Python, François Chollet

ML is divided into three types based on the learning behavior:

(1) **Supervised Learning**

Supervised Learning is where the data is in the form of input-output pairs and the task is to find the map of the main function, so the given data is labeled. Both classification and regression problems are supervised learning problems.
Regression is where the algorithm learns to predict continuous values based on input data. The output labels in regression are continuous values, such as stock prices, and housing prices.
Classification is where the algorithm learns to assign input data to a specific category or class based on input data. The output labels here are discrete values. Classification algorithms can be binary, where the output is one of two possible classes, or multiclass, where the output can be one of several classes.

(2) Unsupervised Learning

Unsupervised Learning is where the task is to draw extrapolation based on data, in form of input, only to assign a pattern in the data. The goal of unsupervised learning is to discover the underlying structure or distribution in the data. Both clustering and dimensionality reduction problems are unsupervised learning problems. Clustering algorithms group similar data points together based on their features. The goal is to identify groups of data points that are like each other, while being distinct from other groups. Dimensionality reduction algorithms reduce the number of input variables in a dataset while preserving as much of the original information as possible. This is useful for reducing the complexity of a dataset and making it easier to visualize and analyze.

(3) Reinforcement Learning

Reinforcement learning is a technique where penalties are added to “agents”, so with each run it learns the process through trial and error. They are most commonly used for auto-driving modern car systems. They are also used to train board games AI such as AlphaGo and AlphaZero. This technique, however, is very computationally expensive as many runs are needed for a model to establish good results.

3.1 Deep Learning

DL is a specific subfield of machine learning. It is a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations. The *deep* in *Deep Learning* isn’t a reference to any kind of deeper understanding achieved by the approach; rather, it stands for this idea of successive layers of representations. How many layers contribute to a model of the data is called the depth of the model (Muhammad et al., 2020).

Other appropriate names for the field could have been layered representations learning and hierarchical representations learning. Modern DL often involves tens or even hundreds of successive layers of representations, and they're all learning automatically from exposure to training data. Meanwhile, other approaches to ML tend to focus on only learning one or two layers of representations of the data.

In DL, these layered representations are, almost always, learned via models called neural networks, structured in literal layers stacked on top of each other. The term neural network is a reference to neurobiology, but although some of the central concepts in deep learning were developed in part by drawing inspiration from our understanding of the brain, deep-learning models are not models of the brain. For our purposes, DL is a mathematical framework for learning representations from data.

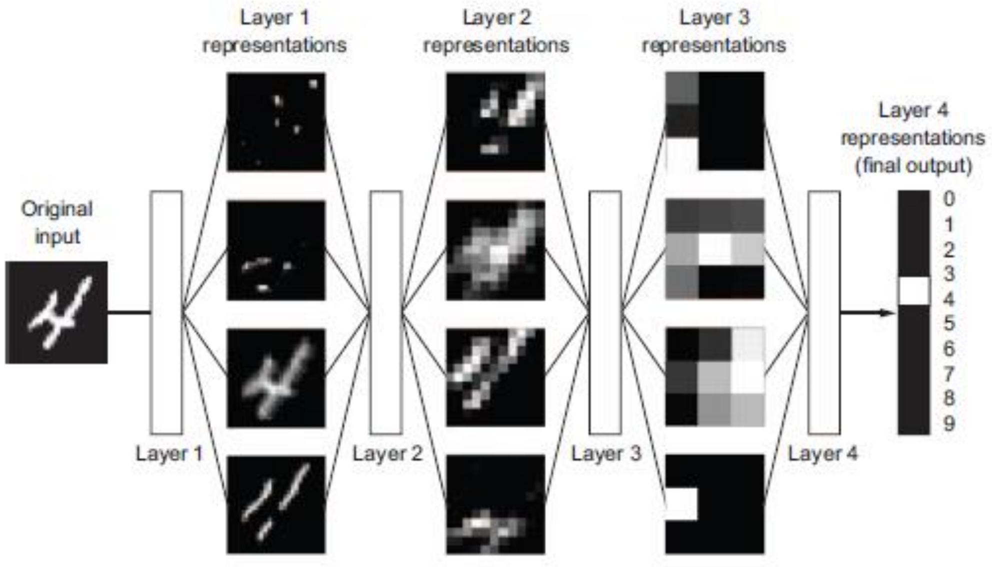


Figure III-3. Deep representations learned by a digit-classification

DL is, technically, a multistage way to learn data representations. It's a simple idea but, as it turns out, very simple mechanisms, sufficiently scaled, can end up looking like magic. ML is about mapping inputs to targets, which is done by observing many examples of input and targets.

Deep neural networks do this input to target mapping via a deep sequence of simple data transformations (layers) and these data transformations are learned by exposure to examples. The specification of what a layer does to its input data is stored in the layer's weights, which in essence are a bunch of numbers. In technical terms, we would say that the transformation implemented by a layer is parameterized by its weights.

Weights are also sometimes called the parameters of a layer. In this context, learning means finding a set of values for the weights of all layers in a network as shown in Figure III-4, such that the network will correctly map example inputs to their associated targets. Since a deep neural network can contain tens of millions of parameters, finding the correct value for all of them may seem like a daunting task. Especially given that modifying the value of one parameter will affect the behavior of all the others. The fundamental trick in DL, thus, is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score.

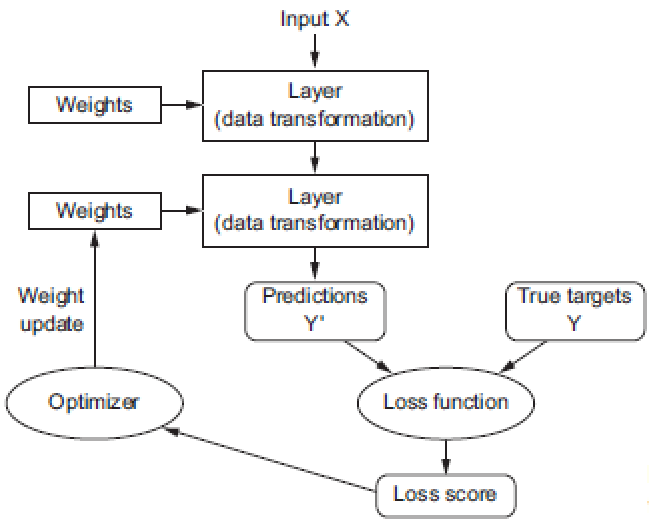


Figure III-4. The loss score is used as a feedback signal to adjust the weights.

This adjustment is the role of the optimizer, which implements what is called the Backpropagation algorithm, the central algorithm in DL.

The weights of a network are assigned random values, so the network merely implements a series of random transformations. Naturally, its output is far from what it should ideally be, and the loss score is accordingly very high. But with every example of the network processes, the weights are adjusted a little in the correct direction, and the loss score decreases. This is the training loop, which, repeated enough times, typically tens of iterations over thousands of examples, yields weight values that minimize the loss function. A network with minimal loss is one for which the outputs are as close as they can be to the targets: a trained network.

The primary reason DL took off so quickly is that it offers better performance on many problems, yet that is not the only reason. DL also makes problem solving much easier, because it completely automates what used to be the most crucial step in a machine-learning workflow: feature engineering.

Machine-learning techniques, shallow learning, only involved transforming the input data into one or two successive representation spaces, usually via simple transformations such as Support Vector Machines (SVMs) or decision trees. The refined representations required by complex problems generally can't be attained by such techniques. As such, humans had to go to great lengths to make the initial input data more amenable to processing by these methods: they had to manually engineer good layers of representations for their data. This is called feature engineering.

DL, on the other hand, completely automates this step. With DL, you learn all features in one pass rather than having to engineer them yourself. This has greatly simplified machine-learning workflows, often replacing sophisticated multistage pipelines with a single, simple, end-to-end deep-learning model.

In practice, there are fast-diminishing returns to successive applications of shallow-learning methods, because the optimal first representation layer in a three-layer model isn't the optimal first layer in a one-layer or two-layer model. What is transformative about DL is that it allows a model to learn all layers of representation jointly, at the same time, rather than in succession. With joint feature learning, whenever the model adjusts one of its internal features, all other features that depend on it automatically adapt to the change, without requiring human intervention. Everything is supervised by a single feedback signal. Every change in the model serves the end goal. This is much more powerful than greedily stacking shallow models, because it allows for complex, abstract representations to be learned by breaking them down into long series of intermediate spaces (layers); each space is only a simple transformation away from the previous one.

These are the two essential characteristics of how DL learns from data, the incremental, layer-by-layer way in which increasingly complex representations are developed, and the fact that these intermediate incremental representations are learned jointly, each layer being updated to follow both the representational needs of the layer above and the needs of the layer below. Together, these two properties have made deep learning vastly more successful than previous approaches to machine learning.

The two key ideas of deep learning for computer vision, Convolutional Neural Networks (CNNs) and backpropagation, were already well understood in 1989. The Long Short-Term Memory (LSTM) algorithm, which is fundamental to deep learning for timeseries, was developed in 1997 and has barely changed since. In general, three technical forces are driving advances in ML:

(1) Computational capabilities.
(2) Datasets and benchmarks.
(3) Algorithmic advances.

Because the field is guided by experimental findings rather than by theory, algorithmic advances only become possible when appropriate data and hardware are available to try new ideas or scale up old ideas, as is often the case. ML isn't mathematics or physics, where major advances can be done with a pen and a piece of paper. It's an engineering science.

AI is sometimes heralded as the new industrial revolution. If deep learning is the steam engine of this revolution, then data is its coal: the raw material that powers our intelligent machines, without which nothing would be possible. When it comes to data, in addition to the exponential progress in storage hardware over the past 20 years, following Moore's law, the game changer has been the rise of the internet, making it feasible to collect and distribute very large datasets for machine learning.

Using only one or two layers of representations, they weren't able to shine against more-refined shallow methods such as SVMs and random forests. The key issue was that of gradient propagation through deep stacks of layers. The feedback signal used to train neural networks would fade away as the number of layers increased.

This changed with the advent of several simple but important algorithmic improvements that allowed for better gradient propagation:

(1) Better activation functions for neural layers
(2) Better weight-initialization schemes, starting with layer-wise pretraining, which was quickly abandoned.
(3) Better optimization schemes, such as RMSProp and Adam.

Only when these improvements began to allow for training models with 10 or more layers did deep learning start to shine. Even more advanced ways to help gradient propagation were discovered, such as batch normalization, residual connections, and depth wise separable convolutions. Today we can train from scratch models that are thousands of layers deep.

Democratization of DL can be defined as the making DL more accessible to non-experts, so that they can use it to solve their own problems without requiring specialized knowledge.

One of the key factors driving this inflow of new faces in DL has been the democratization of the toolsets used in the field. In the early days, doing DL required significant C++ and CUDA expertise, which few people possessed. Nowadays, basic Python scripting skills suffice to do advanced deep-learning research. This has been driven most notably by the development of Theano and then TensorFlow, two symbolic tensor-manipulation frameworks for Python that support auto differentiation, greatly simplifying the implementation of new models, and by the rise of user-friendly libraries such as Keras, which makes deep learning as easy. After its release, Keras quickly became the go-to deep-learning solution for large numbers of new startups, graduate students, and researchers pivoting into the field.

DL has several properties that justify its status as an AI revolution, and it's here to stay. We may not be using neural networks two decades from now, but whatever we use will directly inherit from modern deep learning and its core concepts. These important properties can be broadly sorted into three categories:

(1) Simplicity

DL removes the need for feature engineering, replacing complex, brittle, engineering-heavy pipelines with simple, end-to-end trainable models that are typically built using only five or six different tensor operations.

(2) Scalability

DL is highly amenable to parallelization on GPUs or TPUs, hence, it can take full advantage of Moore's law. In addition, deep-learning models are trained by iterating over small batches of data, allowing them to be trained on datasets of arbitrary size.

(3) Versatility and reusability

Unlike many prior machine-learning approaches, deep-learning models can be trained on additional data without restarting from scratch, making them viable for continuous online learning, serving as an important property for very large production models.

3.2 Neural Networks

The first trial to visualize the neural network in 1943 paved the way to create computational network that can act like human brains. The core idea behind a neural network is to represent the transformation of the input to the output as links between neurons in a sequence of layers.

Each neuron has a weighted connection to the neurons in the layer below, and it applies a nonlinear function to this weighted sum, to determine its own output. The output is connected to the inputs in the layer above or the final output.

So, to get an output, the input passes through these layers or stages, and in every layer the input is represented different than before, yet to be more informative. The layers effect to data is based on bunch of numbers (weights).

In the learning process, it means finding the suitable weights to perform the correct mapping between the input and the target.

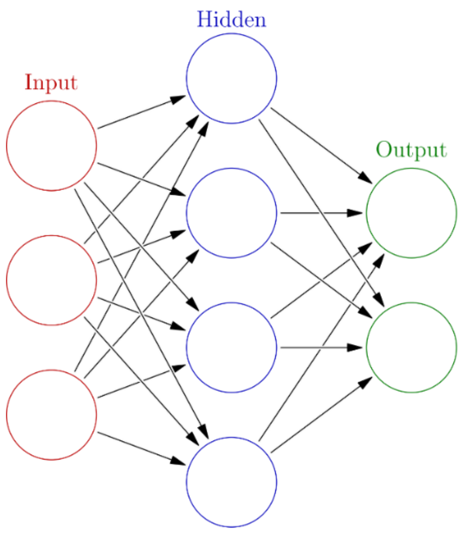


Figure III-5. A scheme for a generic neural network

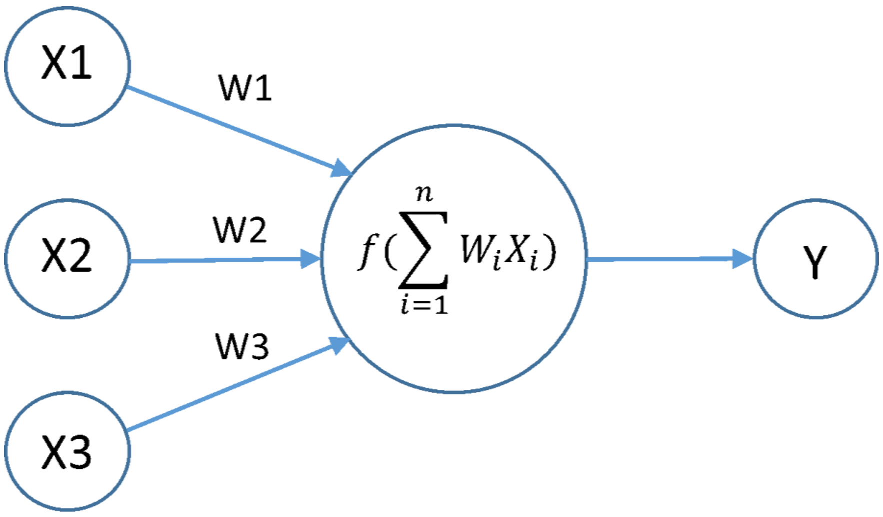


Figure III-6. demonstration of weights for an artificial neuron

3.3 Forward Propagation

The directions of the arrows between two successive nodes represent how the data propagate through the structure passing into different layers:

(1) Input layer

The first layer that receives input data.

(2) Hidden Layer

It can be more than one layer, and each layer consists of a certain number of nodes that can vary between layers. Each node in these layers 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 input, then thy are summed up. The new value passes through the node's activation function before being transferred to the next layer.

(3) Output Layer

The last layer in the model structure produces the output value.

3.4 Backpropagation and Loss Function

To ensure that the predicted value is relatively true, it should be compared with the real value in supervised ML. This comparison shall produce a signal that will be used in the feedback process to update the model parameters to be more accurate. That’s called “Backpropagation”.

One of the important steps in this process is the comparison step, as it requires a way to control the difference between the two outputs, this is the cost function or loss function.

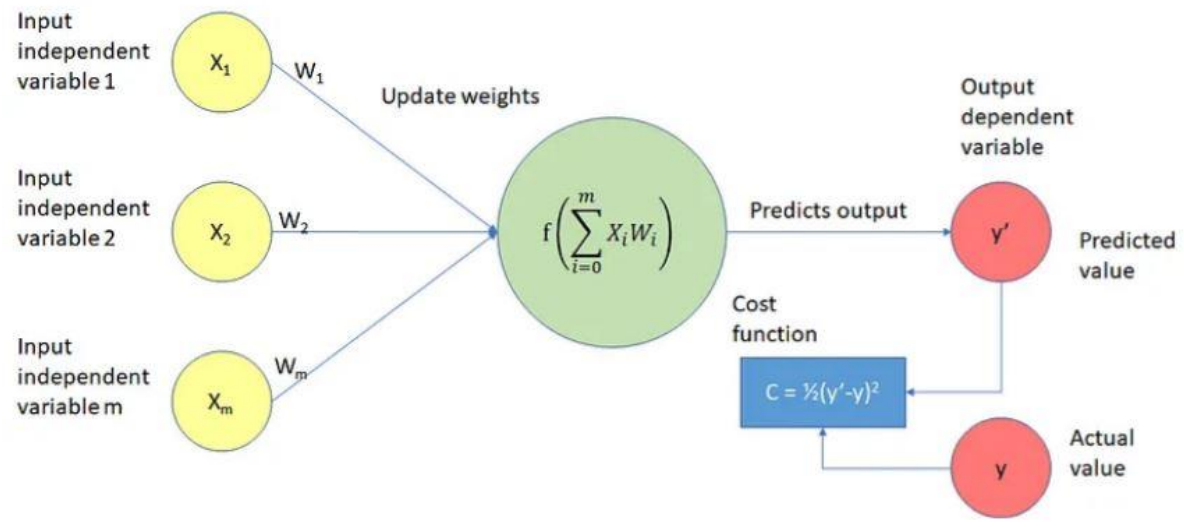
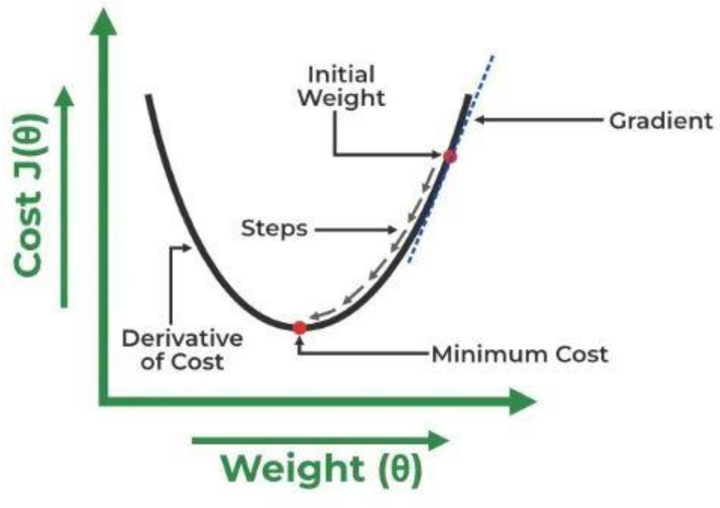

Input independent variable 1
X_1
W_1
Update weights
Output dependent variable
Input independent variable 2
X_2
W_2
$f\left(\sum_{i=0}^{m} X_i W_i\right)$
Predicts output
y'
Predicted value
Cost function
Input independent variable m
X_m
W_m
$C = ½(y'-y)^2$
y
Actual value

Figure III-7. A scheme Showing Cost Function Operation on Weights

For this evaluation, a cost function needs to be defined, ranging in complexity from Mean Squared Error (MSE) to cross-entropy. Root Mean Square Error (RMSE) is a standard way to measure the error of a model in predicting quantitative data.

At the start of the learning process, the weights take random numbers, so the value of the cost function, or the loss, is larger, and this signal goes back as a feedback signal to change the weights to new and suitable ones to decrease the cost until the model output became acceptable. This cycle is the training loop, demonstrated in Figure III-8. 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.

Figure III-8. A Scheme Showing How a Cost Function Reaches a Minimum Loss



Cost functions for Regression problems include:

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)

3.4.1 Mean Absolute Error

Regression metric measures the average magnitude of errors in a group of predictions, without considering their directions. In other words, it's a mean of absolute differences among predictions and expected results where all individual deviations have even importance.

$$MAE = \frac{\overbrace{1}^{\substack{test \\ set}}}{n} \sum_{i=1}^{n} \left| \underbrace{y_i}_{\substack{actual \\ value}} - \underbrace{\hat{y}_i}_{\substack{predicted \\ value}} \right| \qquad \text{(III-1)}$$

Where:

- i: Index of sample
- ŷ: Predicted value.
- y: Expected value.
- n: Number of samples in dataset.

3.4.2 Mean Squared Error

One of the most used and firstly explained regression metrics, which is the average squared difference between the predictions and expected results. In other words, an alteration of MAE where instead of taking the absolute value of differences, they are squared. In MAE, the partial error values were equal to the distances between points in the coordinate system. Regarding MSE, each partial error is equivalent to the area of the square created out of the geometrical distance between the measured points. All region areas are summed up and averaged.

$$MSE = \overbrace{\frac{1}{n}}^{\substack{test\\set}} \sum_{i=1}^{n} \left(\underbrace{y_i}_{\substack{actual\\value}} - \underbrace{\hat{y}_i}_{\substack{predicted\\value}} \right)^2 \tag{III-2}$$

Where:

- i: Index of sample,
- ŷ: Predicted value,
- y: Expected value,
- m: Number of samples in dataset.

There are different forms of MSE formula, where there is no division by two in the denominator, yet its presence makes MSE derivation calculus cleaner.

Calculating derivative of equations using absolute value is problematic. MSE uses exponentiation instead and consequently has good mathematical properties which make the computation of its derivative easier in comparison to MAE. It is relevant when using a model that relies on the *Gradient Descent* algorithm

3.4.3 Root Mean Square error (RMSE)

Root Mean Square error is the extension of MSE. Measuring the average of square root of sum of squared differences between predictions and actual observations.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (\text{ Predicted }_i - \text{ Actual }_i)^2}{N}} \tag{III-3}$$

3.5 Activation Functions

On comparison with a neuron-based model that is in our brains, the activation function is at the end deciding what is to be fired to the next neuron. An activation function is a function that is added into an artificial neural network in order to help the network learn complex patterns in the data (Gupta, 2020). It determines the range of values of activation of an artificial neuron. This is applied to the sum of the weighted input data of the neuron. An activation function is characterized by the non-linearity property. It takes in the output signal from the previous cell and converts it into some form that can be taken as input to the next cell.

Without the application of an activation function, the only operations in computing the output of a multilayer perceptron would be the linear products between the weights and the input values. The application of the non-linear activation function leads to a non-linearity of the artificial neural network and, thus, to a non-linearity of the function that approximates the neural network. There are multiple reasons for having non-linear activation functions in a network:

(1) They help keeping the value of the output from the neuron restricted to a certain limit as per requirement.

(2) Their ability to add non-linearity into a neural network, hence, mimicking the biological neural network complicity. It can be regarded as the most important feature in an activation function.

Instead of thinking of a neural network as a collection of neurons and connections, it can be thought of simply as a function. Like any ordinary mathematical function, a neural network performs a mathematical mapping from input x to output y. The below equation represents a mathematical simplification for activation functions. a, b and c are chosen arbitrary for demonstration.

$$y = f(x_1, x_2, x_3) = a \cdot x_1 + b \cdot x_2 + c \cdot x_3 \mid a = 2, b = 5, c = 7 \qquad \text{(III-4)}$$

Hence, the goal in training a neural network is to find a particular set of weights or parameters so that, given an input vector x, we can compute a prediction y that corresponds to the actual target value y. In other words, we are trying to create a function that can model our training data.
We can only model the data if there exists a mathematical dependency between the input vector x and the labels y. This mathematical dependence can vary in complexity. And in most cases, we as humans cannot see this relationship with our eyes when we take a look at the data.

However, if there is a mathematical dependency between the input vectors and the labels, it is sure that the neural network shall recognize this dependency during training and adjust the weights so that it can model this dependency in the training data. Or, in other words, so that it can realize a mathematical mapping from input features x to output y. The five most important activation functions in deep learning are discussed below.

3.5.1 Sigmoid Function

The most common activation function was the sigmoid function. The sigmoid function maps the incoming inputs to a range between 0 and 1. Its mathematical definition is

$$f(x) = \frac{1}{1 + e^{-x}} \quad \text{(III- 5)}$$

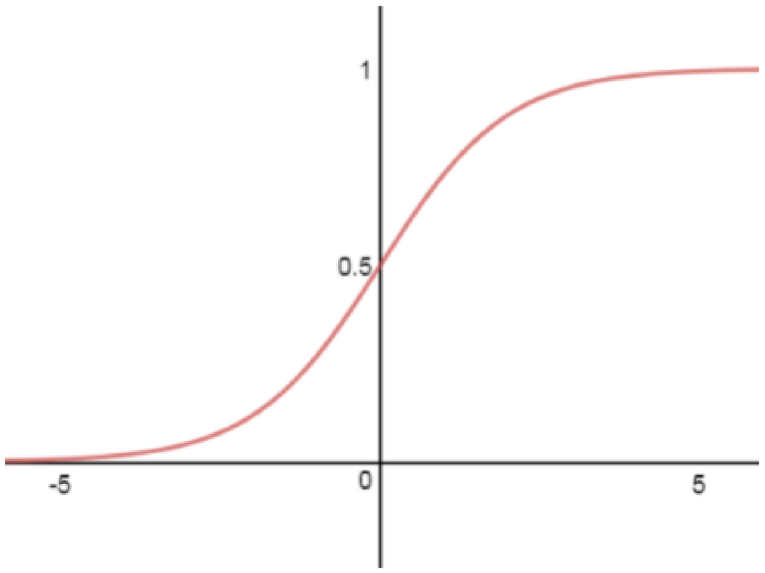


Figure III-9: Graphical Representation of the Sigmoid function

The function takes an input value x and returns the output in the interval [0, 1]. In practice, the sigmoid nonlinearity has recently fallen out of favor and is rarely used. The sigmoid has two disadvantages

1. Gradient Vanishing Problem, where the earlier neurons learn very slowly.
2. Non-Zero Centered. Meaning that the learning more difficult and unstable due to asymmetric sigmoid. we cannot increase and decrease the weights simultaneously; we can only increase or decrease all the weights simultaneously.

3.5.2 Tanh Function

It is very commonly used in Deep Learning. The tangent hyperbolic function is defined mathematically as

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \qquad \text{(III-6)}$$

The function maps a real-valued number to the range [-1, 1]. As with the sigmoid function, neurons saturate at large negative and positive values, and the derivative of the function approaches zero at beyond the interval of [-3,3]. In practice, the tanh function is always preferred to the sigmoid function as its outputs are zero-centered.

3.5.3 Rectified Linear Unit – RELU

The Rectified Linear Unit, or simply ReLU, has become very popular in recent years. The activation is linear for input values greater than zero: R(x) = max(0, x)

Mathematically it can be defined as follows,

$$R(x) = \begin{cases} x \; if \; x > 0 \\ 0 \; otherwise \end{cases} \qquad \text{(III-7)}$$

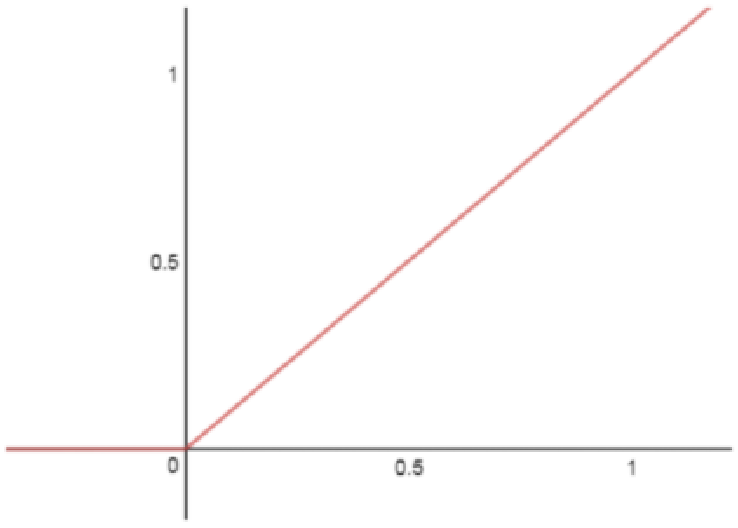


Figure III-10: Graphical Representation of the ReLU function

Advantages of ReLU

- They accelerate the convergence of the gradient descent toward the global minimum of the loss function compared to other activation functions. This is due to its linear, non-saturating property.
- Less computationally expensive. They can be implemented simply by thresholding a value vector at zero.

There is a problem, however, with the ReLU activation function. Because the outputs of this function are zero for input values below zero, the neurons of the network can become very fragile and can “die” during training. This phenomenon happens when the weights are updated; the weights are adjusted in a way that for certain neurons of a hidden layer, the inputs $x < 0$ are always below zero. Hence, the output of the ReLU will always be zero and, therefore, makes no contribution to the training process.

3.5.4 Leaky RELU

This function can be regarded as nothing more than just an improved version of the ReLU function. It is designed to overcome the “death” of some neurons in the ReLU functions.

$$R(x) = \begin{cases} x \; if \; x \geq 0 \\ \propto x \; otherwise \end{cases} \tag{III-8}$$

The advantage of using Leaky ReLU and replacing the horizontal line is that zero gradients are avoided. This is because in this case we no longer have “dead” neurons that are always zero and thus no longer contribute to the training.

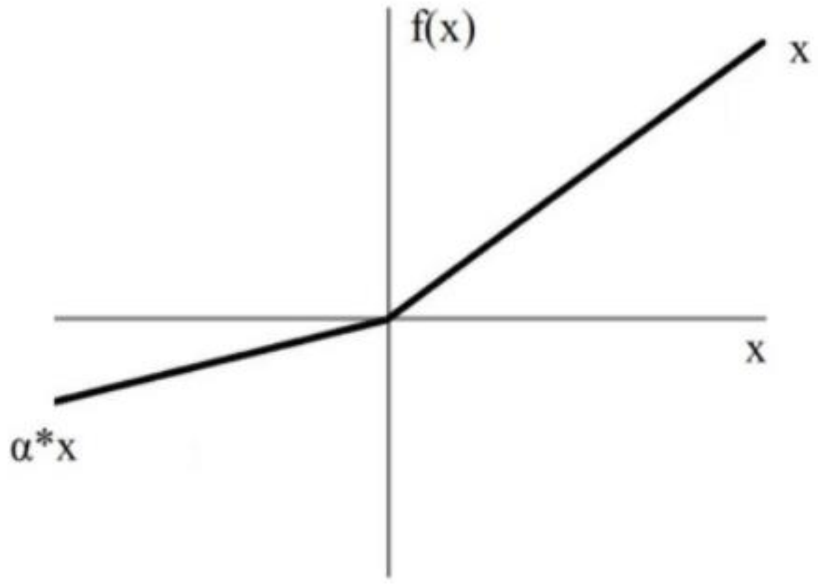


Figure III-11: Graphical Representation of the Leaky ReLU Function

3.5.5 SoftMax Function

SoftMax is only applied in the last layer and only when the neural network is asked to predict probability values in classification tasks. The SoftMax activation function forces the values of the output neurons to take values between zero and one, so that they can represent probability values in the interval [0, 1]. This interval fits the probability distributions making it very useful for multi-class problems. It has its outmost implementations in multi-class problems such as predicting mutually exclusive events. The SoftMax function not only forces the outputs into the range between 0 and 1, but also ensures that the sum of the outputs across all possible classes adds up to one.

There is no precise plot for the SoftMax function as it is a multivariable function. SoftMax isn't a continuous mathematical function such as logistic (Sigmoid), tanh or ReLU. It is used to map outputs of the last layer of a Neural Network into a probability distribution, i.e., summation of SoftMax squashed layer's outputs will be 1 (unity).

As shown below in Figure III-12, the neurons in the output layer receive an input vector z which is the result of a product between a weight matrix of the current layer and the output of the previous layer. A neuron in the output layer with SoftMax activation receives a single value $\boldsymbol{z_1}$, which is an entry in the vector z, and outputs the value $\boldsymbol{y_1}$.

When SoftMax activation is used, each single output of a neuron in the output layer is calculated according to the following equation:

$$y_i = \frac{e^{z_i}}{\sum_{i`} e^{z_{i`}}} \tag{III-9}$$

The choice of a proper activation function solely depends on the problem approached. It also depends heavily on the range of the outputs expected. A rule of thumb is applied to the hidden layers, where their activation function is chosen to be linear.

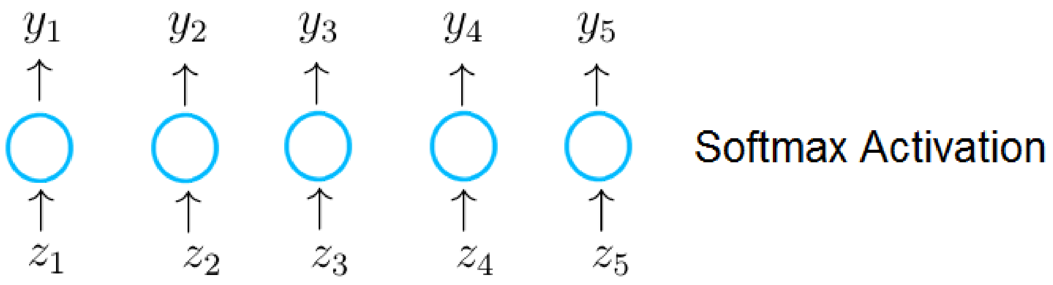


Figure III-12. A scheme of SoftMax Function

3.6 Optimizers

Optimizers are essential algorithms in DL that require a proper and optimized loss function to obtain the ideal desired weights. They update the weights of a neural network during training. They are responsible for finding the set of weights that minimizes the loss function.

Optimizers are the general concept used in neural networks because it involves randomly initializing and manipulating the value of weights for every epoch to increase the model network's accuracy potential. A comparison is made in every epoch between the output from the training data and the actual data, which helps us calculate the errors and find out the loss functions and further updating of the corresponding weights.

There needs to be some way to conclude how the weights should be manipulated to get the most accuracy for which Keras optimizers come into the picture. Keras optimizer helps achieving the ideal weights and get a loss function that is completely optimized. One of the most popular optimizers is gradient descent, which is often used in computer vision classification problems.

Various other Keras optimizers are available and used widely for different practical purposes. There is a provision of various APIs provided by Keras for implementing various optimizers of Keras. The choice of an optimizer depends on the problem addressed and the expected outputs.

An optimizer is class that contains arguments such as:

- learning_rate: A float or a callable that takes no arguments and returns the actual value to use. *The learning rate.*

- beta_1: A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. *The exponential decay rate for the 1st moment estimates.*
- beta_2: A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. *The exponential decay rate for the 2nd moment estimates.*
- epsilon: A small constant for numerical stability.
- amsgrad: Boolean. Whether to apply AMSGrad variant of this algorithm from the paper (Reddi et al., 2019).
- name: String. The name to use for momentum accumulator weights created by the optimizer.
- weight_decay: Float. If set, weight decay is applied.
- clipnorm: Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
- clipvalue: Float. If set, the gradient of each weight is clipped to be no higher than this value.
- global_clipnorm: Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
- use_ema: Boolean. If True, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the

weight values change after each training batch), and periodically overwriting the weights with their moving average.

- ema_momentum: Float. Only used if “use_ema=True”. This is the momentum to use when computing the EMA of the model's weights.
- ema_overwrite_frequency: Int or None, defaults to None. Only used if “use_ema=True”. Every ema_overwrite_frequency steps of iterations, we overwrite the model variable by its moving average. If None, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training.
- loss_scale_factor: Float or None. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training.

- gradient_accumulation_steps: Int or None. If an int, model & optimizer variables will not be updated at every step; instead, they will be updated every “gradient_accumulation_steps” steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Keras provides various types of optimizers:

(1) Adam

This optimizer stands for Adaptive Moment estimation. This makes the Adam algorithm; the gradient descent method is upgraded for the optimization tasks. It requires less memory and is very efficient. This method must go in this scenario when we have a lot of data in bulk quantity and parameters associated with it. It is most popular among developers of neural networks. Adam is discussed more in details below.

(2) Adagrad

This optimizer of Keras uses specific parameters in the learning rates. It has got its base of the frequencies made in the updates by the value of parameters, and accordingly, the working happens. The individual features affect the learning rate and are adjusted accordingly.

(3) Nadam

This optimizer makes use of the Nadam algorithm. It stands for Nesterov and Adam optimizer, and the component of Nesterov is more efficient than the previous implementations. Nesterov component is used for the updating of the gradient by the Nadam optimizer.

(4) Adamax

It is the adaption of the algorithm of Adam optimizer hence the name Adam max. The base of this algorithm is the infinity norm. When using the models that have embeddings, it is considered superior to Adam optimizer in some scenarios.

(5) RMSprop

It stands for Root Mean Square Propagation. The main motive of the RMSprop is to make sure that there is a constant movement in the average calculation of the square of gradients, and the performance of the task of division for gradient upon the root of average also takes place.

(6) FTRL

This optimizer has support shrinkage type L2 and online L2 for loss function.

(7) SGD

This stands for the Keras Stochastic Gradient Descent optimizer and uses momentum and gradient descent. For gradient calculations, a batched subset is used in this type of

Keras optimizer. This Optimizer is mainly used in multi-classification problems and often with the output neuron is a SoftMax activation function.

3.6.1 Adam Optimizer

The Adam optimizer (Adaptive Moment Estimation) is a popular optimization algorithm used in machine learning and deep learning. It combines the strengths of two other extensions of gradient descent: Momentum and RMSprop. This combination makes Adam particularly efficient for large-scale problems involving extensive data and numerous parameters. Adam is known for its low memory requirements and computational efficiency. Adam optimizes the gradient descent process by integrating the principles of Momentum and RMSprop:

- Momentum: This technique accelerates gradient descent by considering the exponentially weighted average of past gradients. It helps the algorithm converge faster by maintaining direction and momentum

$$w_{t+1} = w_t - \alpha m_t \tag{III-10}$$

Where

$$m_t = \beta m_{t-1} + (1-\beta)\left[\frac{\delta L}{\delta w_t}\right] \tag{III-11}$$

Here:

- $\mathbf{m_t}$ = aggregate of gradients at time t [current] (initially, $m_t = 0$)
- $\boldsymbol{m_{t-1}}$ = aggregate of gradients at time t-1 [previous]
- $\boldsymbol{w_t}$ = weights at time t
- $\boldsymbol{w_{t+1}}$ = weights at time t+1
- $\boldsymbol{\alpha_t}$ = learning rate at time t
- $\boldsymbol{\delta L}$ = derivative of Loss Function
- $\boldsymbol{\delta w_t}$ = derivative of weights at time t

- $\boldsymbol{\beta}$ = Moving average parameter (const, 0.9)

- RMSprop: This technique improves upon Adagrad by using an exponentially weighted moving average of squared gradients. It prevents the learning rate from becoming too small.

$$w_{t+1} = w_t - (v_t + \epsilon)^{-1/2} \cdot \alpha_t \cdot \left(\frac{\partial L}{\partial w_t}\right) \quad \text{(III-12)}$$

Where

$$v_t = \beta v_{t-1} + (1 - \beta)\left(\frac{\partial L}{\partial w_t}\right)^2 \quad \text{(III-13)}$$

Here:

- $\boldsymbol{w_t}$ = weights at time t
- $\boldsymbol{w_{t+1}}$ = weights at time t+1
- $\boldsymbol{\alpha_t}$ = learning rate at time t
- $\boldsymbol{\partial L}$ = derivative of Loss Function
- $\boldsymbol{\partial w_t}$ = derivative of weights at time t
- $\boldsymbol{v_t}$ = sum of square of past gradients. [i.e sum(∂L/∂Wt-1)] (initially, V_t = 0)
- $\boldsymbol{\beta}$ = Moving average parameter (const, 0.9)
- $\boldsymbol{\epsilon}$ = A small positive constant (10^{-8})

Adam Optimizer combines the strengths of previous methods to give a more optimized gradient descent. It builds upon these strengths to control the gradient descent rate effectively, minimizing oscillations as it approaches the global minimum demonstrated in Figure III -13.

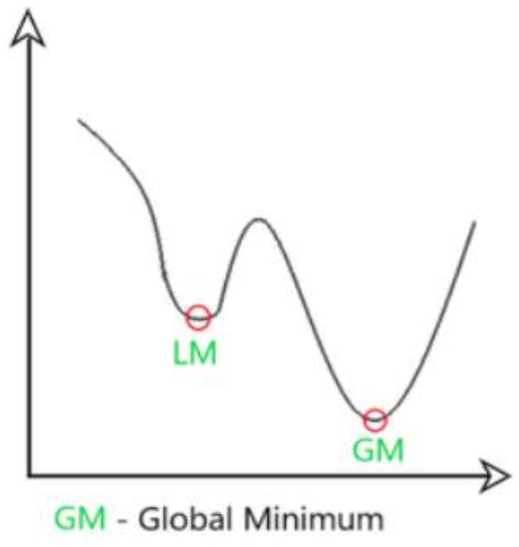


Figure III -13 Global and local Minimums approached by Adam

By taking sufficiently large steps (step-size), it navigates past local minima hurdles along the optimization path. This approach integrates features from momentum and RMSprop methods, which are known for their effectiveness in adjusting learning rates based on the gradient's first and second moments. This adaptive mechanism ensures efficient convergence, particularly beneficial in deep learning applications dealing with large-scale datasets and high-dimensional parameter spaces.

As mentioned earlier, the Adam optimizer integrates formulas from momentum and RMSprop methods to optimize gradient descent effectively. Here are the key formulas uti-lized:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\delta L}{\delta w_t} \qquad \text{(III-14)}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\delta L}{\delta w_t}\right)^2 \qquad \text{(III-15)}$$

Here:

- ϵ: A small positive constant (typically $\mathbf{10^{-8}}$) to prevent division by zero in case $\boldsymbol{v_t}$ approaches zero.
- $\boldsymbol{\beta_2}$ ***and*** $\boldsymbol{\beta_1}$: Decay rates for the exponential moving averages of gradients.
- α: Step size parameter or learning rate (typically around 0.001).

Initially, $\boldsymbol{m_t}$ and $\boldsymbol{v_t}$ tend to be biased towards zero due to $\boldsymbol{\beta_1}$ and $\boldsymbol{\beta_2}$ being close to 1. To correct this bias, Adam computes bias-corrected estimates $\boldsymbol{\widehat{m_t}}$ and $\boldsymbol{\widehat{v_t}}$:

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t} \tag{III -16}$$

$$\widehat{v_t} = \frac{v_t}{1-\beta_2^t} \tag{III- 17}$$

These corrected estimates ensure that $\boldsymbol{m_t}$ and $\boldsymbol{v_t}$ remain unbiased throughout the optimization process, aiding in controlled descent towards the global minimum while reducing oscillations. The update rule for Adam optimizer then becomes:

$$w_{t+1} = w_t - \widehat{m_t}\left(\frac{\alpha}{\left(\sqrt{\widehat{v_t}} + \epsilon\right)}\right) \tag{III - 18}$$

This adaptive learning rate mechanism adjusts the step size for each parameter based on their respective gradients' statistics, leading to efficient convergence in high-dimensional parameter spaces and large-scale datasets.

For the performance, Adam optimizer significantly enhances gradient descent compared to previous methods, achieving superior performance in terms of training cost and convergence speed. Figure III-14 illustrates Adam's superior performance, demonstrating lower training costs and faster convergence rates compared to other optimizers.

3.7 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). The Learning Rate (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. 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.

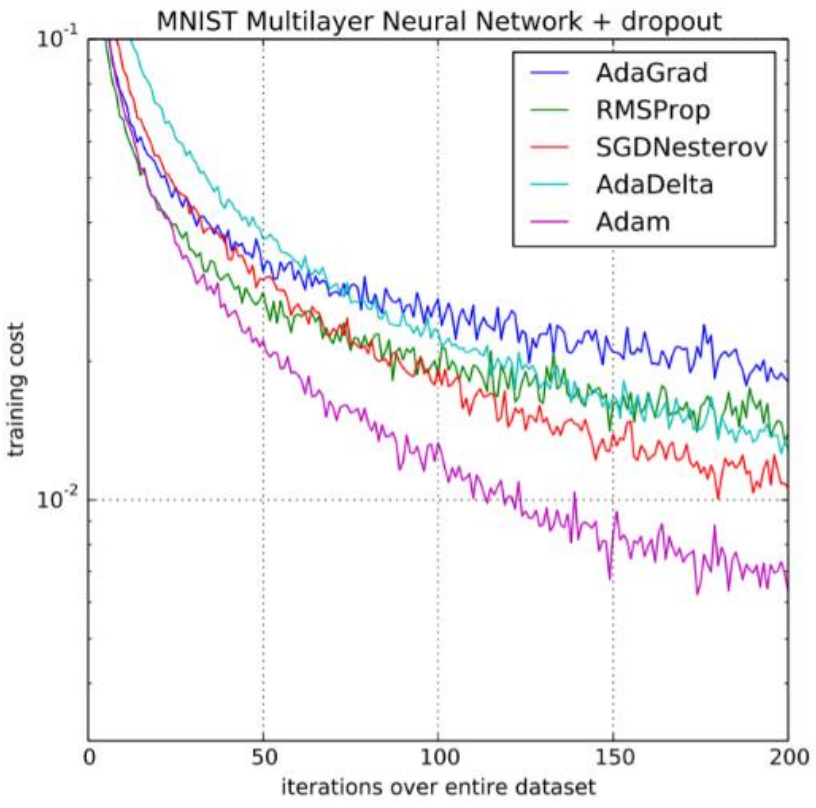


Figure III-14. A Comparison Showing Adam Performance Compared to Other Optimizers.

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.

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

3.8 Regularization

Regularization techniques are used in machine learning to prevent overfitting, which occurs when a model performs well on training data but poorly on unseen data. Regularization adds a penalty to the loss function to constrain the model's complexity, thereby improving its generalization to new data. The addition is done via Keras's L1L2 module, which uses a technique called *Elastic Net Regularization* (Giba, n.d.).

This technique is particularly useful when there are multiple features that are correlated with each other. The loss function for Elastic Net is a convex combination of the L1 and L2 penalties.

$$\text{Loss} = \frac{1}{2m}\sum_{i=1}^{m} (y_i - \hat{y}_i)^2 + \lambda_1 \sum_{j=1}^{n} |w_j| + \frac{\lambda_2}{2}\sum_{j=1}^{n} w_j^2 \tag{III - 19}$$

(1) The first term $\frac{1}{2m}\sum_{i=1}^{m} (y_i - \hat{y}_i)^2$ represents the mean squared error (MSE) loss, which is the standard loss function for regression tasks. It measures the average squared difference between the true values and the predicted values.

(2) The second term $\lambda_1 \sum_{j=1}^{n} |w_j|$ represents the L1 regularization (Lasso), which adds the absolute value of the coefficients as a penalty. It encourages sparsity in the model by driving some coefficients to be exactly zero, effectively performing feature selection.

(3) The third term $\frac{\lambda_2}{2}\sum_{j=1}^{n} w_j^2$ represents the L2 regularization (Ridge), which adds the squared value of the coefficients as a penalty. It discourages large coefficients by shrinking them, thus helping to prevent overfitting.

Where m is the number of training examples, n the number of features, $\boldsymbol{w_j}$ the $\boldsymbol{j_{th}}$ model parameter weight, $\boldsymbol{y_i}$ the true value of the $\boldsymbol{i_{th}}$ training example, $\boldsymbol{\hat{y}_i}$ is the predicted value of the $\boldsymbol{i_{th}}$ training example.

3.9 Types of Neural Networks

3.9.1 Feedforward Neural Networks (FNN)

One of the most basic neural network models is a Feedforward Neural Network (FNN), also known as Multiple-Layer Perceptron (MLP), shown in Figure III-15. It is trained with back-propagation learning algorithms and is of the most popular neural networks available today.

It consists of multiple neurons each receives the output from the neurons of the previous layer and is weighted and processed by certain weights and biases to obtain the output of the neurons of the current layer, serving as input of the neurons of the next layer.

The network information stream is uni-directional and can only move from the input layer to the output layer, so it is called a feedforward neural network.

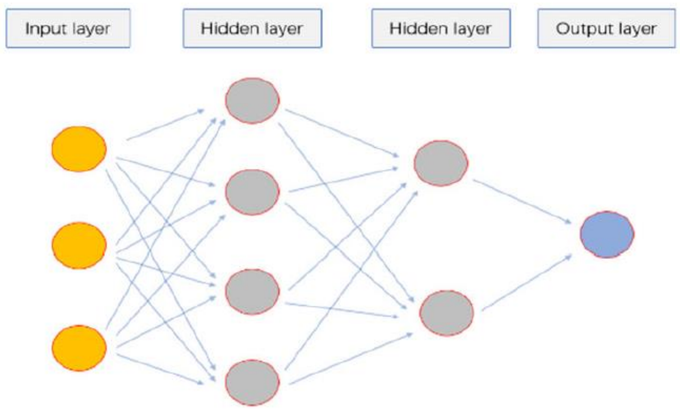


Figure III-15. FeedForward Neural Network Layout

The FNN is usually composed of an input layer, several hidden layers and an output layer, as shown in Figure III-15. When the input layer receives the original data input, the hidden layer transforms it and extracts the feature to a certain extent, and the hidden layer's results are used to predict the model's value in the output layer.

Each layer consists of more than one neuron, and the connections between the neurons are weighted, which can be learned to optimize through backpropagation algorithms. FNN are powerful models that are capable of handling non-linear classification and regression tasks. FNN is an artificial neural network in which the nodes do not form loops. Also, FNNs that have hidden layers can be trained to perform and solve any classification and regression problems. In general, they can be used Include image classification and credit scoring.

3.9.2 Convolutional Neural Networks (CNN)

They are feed-forward neural networks that have a deep structure and include convolutional computation known as Convolutional Neural Networks (CNNs). Networks of convolutional neurons have been proposed by the mechanism of the Receiving Domain in biology. CNNs are designed to process data that has a grid-like structure. For instance, a one-dimensional grid formed by regular sampling on the timeline is the form of time series data and image data is viewed as a grid of pixels in two dimensions.

As shown in Figure III-16, the convolutive layer will dictate the output of the neuron and connect it to the local by calculating the scalar product between its weight and the areas that are connected to the input volume. The pooling layer is primarily responsible for reducing the size of the feature map, which in turn reduces the number of calculations and parameters, and enhances the effectiveness of model training. The latter layer of the CNN, the fully connected layer, is often used to integrate the characteristics extracted by the convolution and pooling layers, and carry out the final classification or regression task.

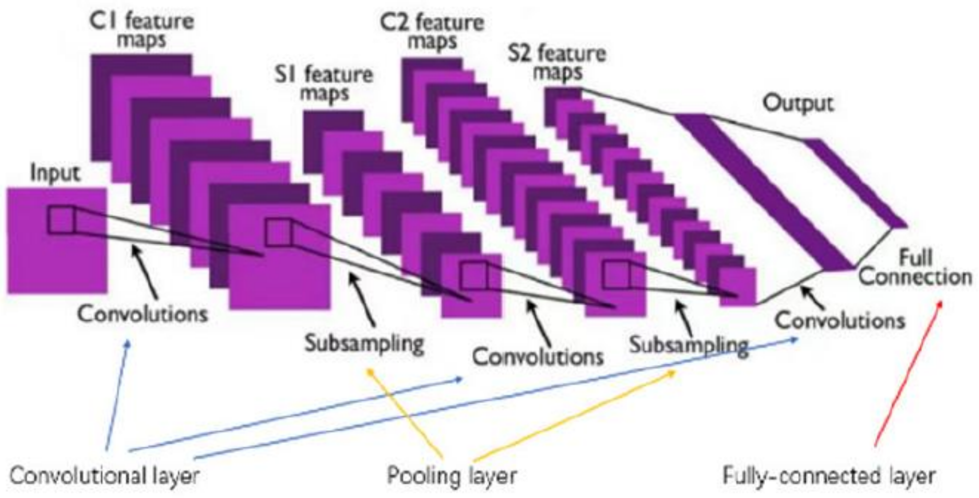


Figure III-16. CNNs Layout

The functions of the fully connected layer include:

(1) **Feature Integration**

- The fully connected layer incorporates the local characteristics of the convolution and pooling layers into a global feature vector to capture the higher-order relationships between features.

(2) **Classification or regression**

- The fully connected layer can be used to implement the final classification or regression task.
- The classification task commonly uses the SoftMax activation function in the output layer to map the feature vector to the probability distribution of each class.
- In a regression task, a linear activation function or other appropriate activation function can be used to predict continuous values.

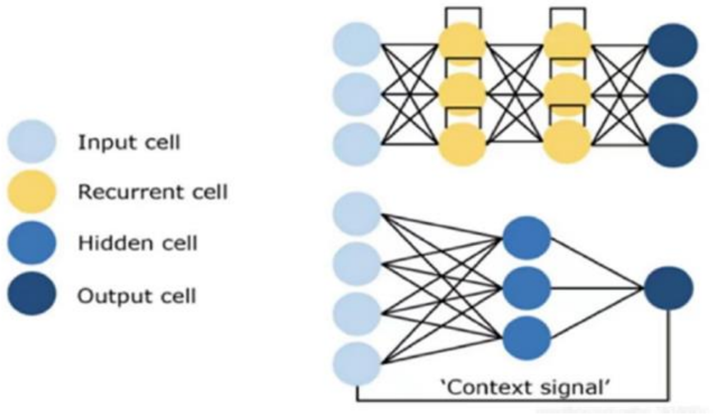
(3) **Non-linear mapping**

- Activation functions in the fully connected layer (such as ReLU, Sigmoid, etc.) introduce nonlinear mapping into the network to improve the expressiveness of the model.
- CNN is a unique feedforward neural network that has a local connection, weight sharing, and pooling, which is suitable for processing data with grid structure (such as images and speech signals). Their application are mainly in Computer vision and analysis of medical imagery.

3.9.3 Recurrent Neural Networks (RNNs)

A network that sends feedback signals to each other is called a Recurrent Neural Network (RNN), shown in Figure III-17. Processing time series data requires a specific type of neural network. The main feature of RNN is the existence of cyclic connections in the network, which allows the network to remember previous information.

Figure III-17. RNNs Layout



The main components of an RNN can be classified into:

(1) **Input layer**

The input cell is responsible for receiving timing data. When processing sequential data (such as text, time series), the network receives data from the input cell for each step. The input cell can be a simple vector or a more complex structure, such as the Embedding Layer, which maps discrete data to a continuous vector space.

(2) **Hidden cell**

The hidden cell is the core part of the RNN and contains Recurrent Units. The masked layer is informed of the input data of the current temporal step and the masked state of the previous temporal step at each temporal step.
The hidden state contains information about previous time steps, giving the RNN memory power.

The cyclic units in the hidden cell can adopt different structures, such as simple RNN units, Long Short-Term Memory units (LSTM), or Gated Recurrent Units (GRU). The main functions of the hidden cell are learning and preserving long-term and short-term dependencies in time series data. Also, the information from the previous time step is integrated with the input of the current time step to generate a new hidden state.

(3) **Recurrent cell**

The recurrent cell is the core components of the net-work that processes timing data and retains historical information.

The functions of the recurrent cell are:

- Processing time-series data
- Retention of historical data
- Learning temporal features

(4) **Output cell**

The hidden cell's hidden state is used by the output cell for generation of the final output. The output cell can be a simple linear transformation, an activation function, or a more complex structure such as the SoftMax classifier.

The main functions of the output cell are:

- Map the timing features learned by the RNN to the target space (such as classification labels, regression values, etc.).
- In sequence generation tasks

An RNN is a neural network with an internal state that is capable of processing time series data. In RNNs, certain results are fed back into the network, giving it memory capability. They can be used in various applications such as natural language processing and speech recognition.

3.9.4 Long Short-Term Memory Neural Networks (LSTM)

Long Short-Term Memory (LSTM) networks, a specialized variant of RNNs, have emerged as a cornerstone in the domain of sequence modeling and temporal data analysis. Conceived by Hochreiter and Schmidhuber in 1997. LSTM networks address the limitations of traditional RNNs, particularly the challenges associated with long-term dependencies and the vanishing gradient problem. This essay delves into the structural intricacies, operational mechanisms, and diverse applications of LSTM networks, elucidating their significance in contemporary machine learning.

LSTM networks distinguish themselves through their unique architecture, which incorporates memory cells, input gates, forget gates, and output gates. These components collectively enable LSTM networks to retain and manipulate information over extended sequences, thereby mitigating the gradient decay observed in conventional RNNs. The memory cell serves as the core unit, preserving state information across time steps. The input gate modulates the incorporation of new information into the memory cell, while the forget gate regulates the retention or discarding of existing cell states. The output gate, in turn, determines the extent to which the current cell state influences the output of the LSTM unit.

Mathematically, the operations within an LSTM cell can be delineated through a series of equations. The forget gate, denoted as $\boldsymbol{f_t}$, is computed by applying a sigmoid function to the concatenation of the previous hidden state $\boldsymbol{h_{t-1}}$ and the current input $\boldsymbol{x_t}$, weighted by $\boldsymbol{W_f}$ and biased by $\boldsymbol{b_f}$. This can be expressed as:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{III - 20}$$

Similarly, the input gate $\boldsymbol{i_t}$ and the candidate cell state $\boldsymbol{\tilde{C}_t}$ are defined as:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{III - 21}$$

$$\tilde{C}_t = \tanh\,(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{III - 22}$$

The new cell state $\boldsymbol{C_t}$ is then updated by combining the previous cell state $\boldsymbol{C_{t-1}}$ scaled by the forget gate $\boldsymbol{f_t}$ and the candidate cell state $\boldsymbol{\tilde{C}_t}$ scaled by the input gate $\boldsymbol{i_t}$:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{III - 23}$$

These equations collectively enable LSTM networks to dynamically regulate information flow, thereby preserving relevant data across long temporal horizons and facilitating effective learning in complex sequence tasks. Below is a graphical representation of the architecture of an LSTM network Figure III-18 (Zhang et al., 2023).

The robustness of LSTM networks in managing long-term dependencies has catalyzed their adoption across a spectrum of applications. In natural language processing, LSTMs have been instrumental in tasks such as language modeling, machine translation, and sentiment analysis.

By effectively capturing syntactic and semantic nuances over extended text sequences, LSTMs enhance the performance of language models and translation systems. For instance, in machine translation, LSTM-based encoder-decoder architectures have demonstrated remarkable proficiency in translating between languages with divergent grammatical structures.

LSTMs have found utility in time series forecasting, where their ability to model temporal correlations and seasonality patterns has proven advantageous. In finance, LSTM networks facilitate accurate prediction of stock prices and market trends by analyzing historical data and identifying latent temporal patterns. Similarly, in the domain of healthcare, LSTMs are employed to predict patient outcomes and disease progression by leveraging longitudinal medical records.

The integration of LSTM networks into speech recognition systems has also yielded significant improvements. By capturing temporal dependencies in audio signals, LSTMs enhance the accuracy of speech-to-text conversion and enable more natural interactions in voice-activated applications. Additionally, LSTMs are utilized in the field of music generation, where they assist in composing melodies by learning from existing musical compositions and generating coherent sequences that emulate human creativity.

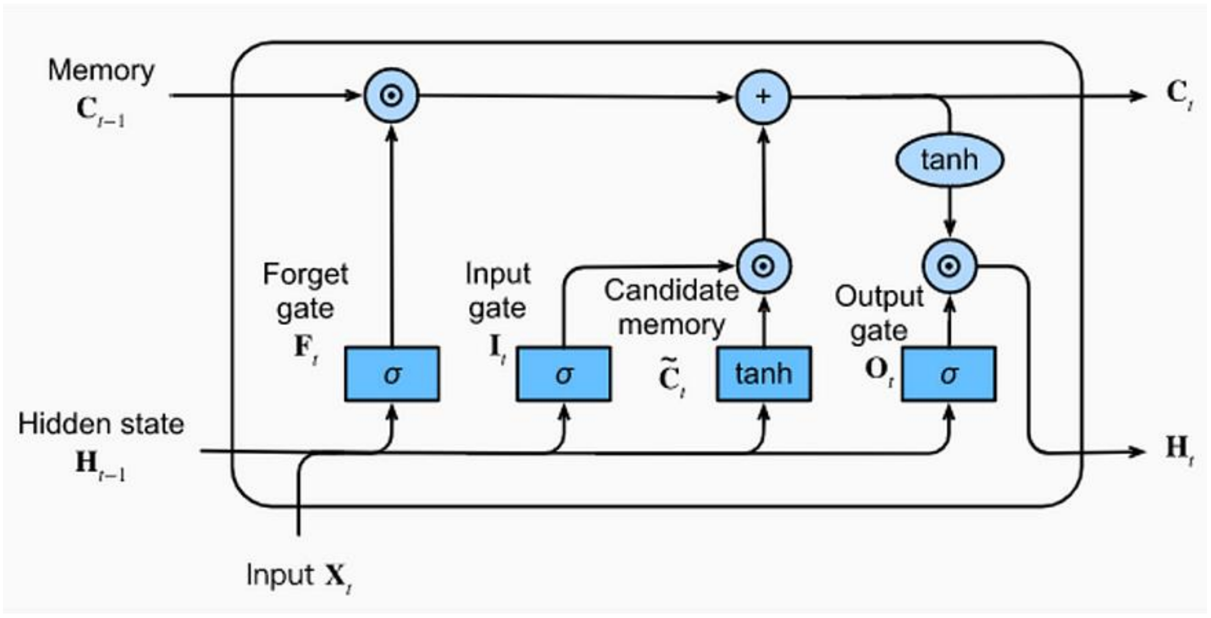

Memory $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
tanh
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
tanh
Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Input $\mathbf{X}_t$

Figure III-18. LSTM Architecture

In this work, LSTM networks are used in what is deemed as a time series regression problem. By utilizing LSTMs ability to model temporal correlations and patterns, a regression model of break sizes can be established.

Despite their numerous advantages, LSTM networks are not devoid of challenges. The complexity of their architecture, characterized by a multitude of gates and memory cells, necessitates substantial computational resources and can lead to prolonged training times. Furthermore, the efficacy of LSTMs is contingent upon the careful tuning of hyperparameters, such as the number of hidden units and learning rate, which can pose practical difficulties in large-scale applications.

3.10 TensorFlow and Python

3.10.1 Python

Python is chosen as the programming language of choice in this work, not only due to its simple syntax, but also due to its rising popularity in the data science community.

The biggest advantage that Python provides is the rather large array of external libraries and support that make complicated processes simpler. An example of this can be illustrated by examining the process of reading input files.

Parsing Input Files

Comma-separated value (CSV) files are rather simple to handle. They're essentially text files written in the format below.

```
hmi_RCSCT5_VALUE,hmi_RRCTP10_VALUE
850.551,3980.49
```

Each variable is separated by a comma from the previous, and each line represents a row. It would be trivial to write an algorithm that parses those files utilizing Python's built-in file reading functions (Rossum, n.d.), a sample of such algorithm is:

(1) **Open the CSV File**:

- Use Python’s built-in open() function to open the CSV file in read mode.

(2) Read the File Line by Line:

- Use a loop to read each line from the file.

(3) **Split Lines into Fields**:

- Split each line into fields based on the comma delimiter.

(4) **Store the Data**:

- Store each list of fields into a larger list representing all the rows.

(5) **Close the CSV File**:

- Close the file after reading to free up system resources.

(6) **Return the Parsed Dat**a:

- Return the list containing all rows and their respective fields.

An implementation of the above algorithm is provided in Appendix B. On the other hand, reading excel files is much harder.

This is due to the nature of what an excel file is. Excel Files (.xlsx, .xlsm, .xltx, and .xltm) are essentially a compressed file containing multiple Extensible Markup Language (xml) files, which can be easily read by a text editor, but contain much more nuanced syntax, making it harder to extract the data. An algorithm for reading Excel files from scratch would look like:

(1) **Open the Excel File**:

- Use Python’s built-in open() function to open the Excel file in binary read mode.

(2) **Unzip the File**:

- Use Python’s zipfile module to unzip the Excel file.

(3) **Read the Workbook XML**:

- Extract the xl/workbook.xml file to understand the structure of the workbook and get the sheet names.

(4) **Read the Worksheet XML**:

- Extract the xl/worksheets/sheet1.xml file (assuming we're reading the first sheet) to get the data.

(5) **Parse the XML Content**:

- Use an XML parser to read the cell data and convert it into Python lists.

(6) **Return the Parsed Data**:

- Return the list containing all rows and their respective cell values.

The above algorithm assumes only one excel sheet for simplicity, and as you may already have noticed, already utilizes two modules built-in python instead of writing the code from complete scratch. An implementation of the above algorithm is also provided in in Appendix B.

3.10.2 Pandas

Due to the complexities of the above algorithm, and thanks to the wide support of the Python programming language, there are multiple frameworks that make the prior process as easy as writing one line of code. The most popular and commonly used in the Machine Learning space being "Pandas" (Pandas Development Team, 2024).

Pandas, a powerful data analysis library for Python, was created by Wes McKinney in 2008. McKinney, while working at AQR Capital Management, recognized the need for a high-performance, flexible data manipulation tool tailored for financial analysis. Inspired by the functionality of data analysis tools in other languages, such as R and SAS, he developed Pandas to offer similar capabilities in Python. The library has since grown rapidly in popularity, becoming a cornerstone for data science and analytics due to its efficiency, ease of use, and robust feature set, supporting a wide range of data manipulation tasks from simple data loading to complex data transformations.

Pandas significantly simplifies and enhances the process of reading CSV and Excel files, providing an efficient and user-friendly interface for data manipulation. When reading CSV files, Pandas uses the *read_csv* function, which internally leverages optimized C-based parsers to handle large files quickly. This function automatically handles a variety of tasks such as detecting data types, handling missing values, parsing dates, and more, which would require extensive manual coding if done from scratch. Additionally, Pandas offers robust support for different delimiters, encodings, and options to read data in chunks, facilitating the processing of very large datasets that may not fit into memory.

For Excel files, Pandas employs the *read_excel* function, which builds on libraries like *openpyxl* and *xlrd* to read Excel 2007+ and older files respectively. This function abstracts the complexities of dealing with the ZIP archive structure of Excel files and parsing XML content. With just a single function call, users can read data from specific sheets, handle merged cells, and even filter rows and columns while reading the data. Pandas efficiently manages the intricate details of data extraction, such as dealing with shared strings and various data types, ensuring that users receive a clean and structured *DataFrame*. This high-level abstraction not only saves time but also reduces the potential for errors, making Pandas an indispensable tool for data analysis and manipulation in Python. Example implementations of the usage of both *read_excel* and *read_csv* functions are in the second two codes provided in Appendix B.

3.10.3 Numpy

NumPy (Harris et al., 2020), short for Numerical Python, is another fundamental library in the Python ecosystem, designed primarily for numerical computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. Here’s a continuation from where we left off:

NumPy complements Pandas in data analysis and manipulation tasks by offering:

(1) **Efficient Array Operations:** NumPy arrays are more compact than Python lists and offer a significant performance boost due to being implemented in C. This efficiency is crucial for handling large datasets commonly encountered in data science and machine learning tasks.

(2) **Mathematical Functions**: NumPy provides a wide range of mathematical functions that operate on entire arrays of data without the need for explicit looping, making code concise and fast. These functions include basic arithmetic operations, statistical functions, linear algebra operations, and more.

(3) **Random Number Generation**: NumPy includes functions for random number generation, which are essential for tasks like simulating data or initializing parameters in statistical models.

(4) **Integration with Pandas**: NumPy arrays are compatible with Pandas, allowing seamless integration between the two libraries. Data can be easily converted between Pandas DataFrames and NumPy arrays, enabling users to leverage the strengths of both libraries in a single workflow.

(5) **Support for Custom Data Types**: NumPy arrays can be created with a specific data type, such as integers or floats of various sizes, ensuring efficient memory usage and compatibility with other libraries that expect specific data types.

3.10.4 TensorFlow

TensorFlow is an end-to-end platform designed for deep learning and machine learning, offering Python-friendly open-source capabilities. This versatile library excels at training deep neural networks for various tasks including image recognition, handwritten digit classification, word embedding, and recurrent neural networks. In addition, it facilitates scalable prediction using trained models. TensorFlow also provides integration with a range of APIs to create deep learning architectures such as RNNs and CNNs.

TensorFlow's capabilities

(1) **Core Functionality**

- Tensor Operations: TensorFlow's core is built around tensors, which are multi-dimensional arrays. The library provides extensive support for tensor operations, allowing for efficient computation and manipulation.
- Graph Computation: TensorFlow allows for the creation of computational graphs, where nodes represent operations and edges represent tensors. This enables efficient computation and optimization.

(2) **Machine Learning Models**

- Deep Learning: TensorFlow excels at building deep learning models, including neural networks like CNNs, RNNs, and GANs.
- Custom Models: Users can build custom machine learning models tailored to specific tasks, benefiting from TensorFlow’s flexibility and comprehensive API.
- Pre-trained Models: TensorFlow Hub provides access to pre-trained models that can be easily integrated and fine-tuned for specific tasks.

(3) **Model Training and Evaluation**

(4) **Deployment and Production**

- TensorFlow offers robust support for deploying machine learning models in production environments, ensuring efficient real-world applications. TensorFlow Serving is a high-performance system designed for production, allowing dynamic model management with gRPC and REST APIs. TensorFlow Lite optimizes models for mobile and edge devices, supporting Android, iOS, and embedded systems with efficient conversion and optimization tools. TensorFlow.js enables model deployment directly in web browsers and Node.js environments, providing pre-trained models and transfer learning capabilities. TensorFlow Extended (TFX) supports end-to-end ML pipelines with components for data validation, model training, and serving, integrating with Apache Beam and Kubernetes for scalable deployment. Model monitoring and management tools detect issues like data drift and degradation, ensuring reliable performance. TensorFlow's integration with Google Cloud AI Platform, AutoML, and Kubeflow simplifies cloud deployment and management, making TensorFlow a powerful tool for delivering machine learning models to production.

(5) **Specialized Libraries**

- TensorFlow Probability: A library for probabilistic reasoning and statistical analysis.
- TensorFlow Extended (TFX): An end-to-end platform for deploying production machine learning pipelines.
- TensorFlow Federated: Enables decentralized, privacy-preserving machine learning.
- TensorFlow Graphics: Provides tools for differentiable rendering and 3D graphics.

(6) **Communities and Ecosystems**

- Community Support: TensorFlow has a vibrant community contributing to tutorials, models, and tools, enhancing learning and collaboration.
- Integration with Other Tools: TensorFlow integrates well with other libraries and frameworks like PyTorch, scikit-learn, and Apache Spark, facilitating diverse workflows.

3.10.5 Keras

Keras is a deep learning API for Python, built on top of TensorFlow, designed to provide a convenient way to define and train deep learning models. Originally developed for research purposes, Keras aims to facilitate rapid deep learning experimentation.

It is hardware compatible. Through TensorFlow, Keras can run on various hardware platforms, including GPUs, TPUs, and CPUs, and can be scaled seamlessly across thousands of machines. Keras prioritizes the developer experience, focusing on creating an API for humans rather than machines. It adheres to best practices for reducing cognitive load by offering consistent and straightforward workflows, minimizing the actions required for common tasks, and providing clear, actionable feedback on user errors. This makes Keras easy to learn for beginners and highly productive for experts.

Keras has a user base of over a million, including academic researchers, engineers, and data scientists at both startups and large companies, as well as graduate students and hobbyists (Chollet, n.d.).

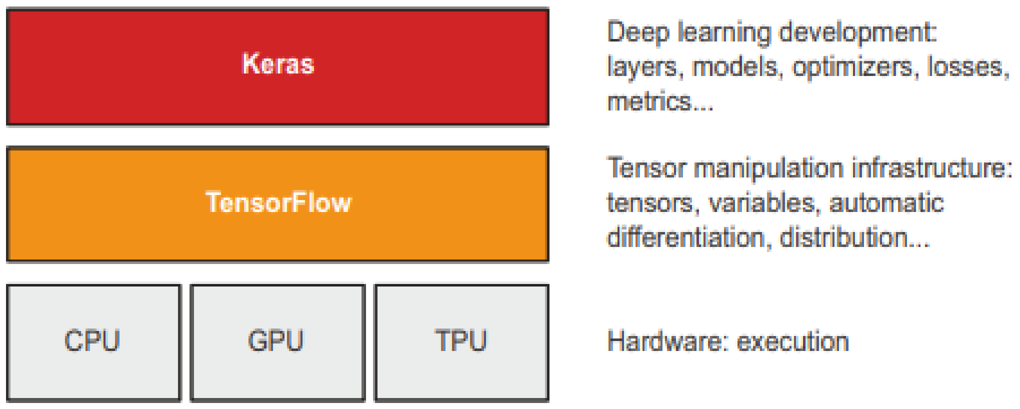

Keras
TensorFlow
CPU
GPU
TPU
Deep learning development: layers, models, optimizers, losses, metrics...
Tensor manipulation infrastructure: tensors, variables, automatic differentiation, distribution...
Hardware: execution

Figure III-19. Keras and TensorFlow: TensorFlow is a low-level tensor computing platform, and Keras is a high-level deep learning API

3.10.5.1 Anatomy of A Neural Network: Understanding Core Keras Apis

Layers form the core data structure in neural networks. A layer processes input tensors and produces output tensors. Some layers are stateless, but most have a state, consisting of weights learned through stochastic gradient descent, representing the network's knowledge (Chollet, 2021).

Different layers are suited to different data types and tensor formats. For example:

- Dense layers (Dense class in Keras): Used for rank-2 tensors (samples, features), ideal for simple vector data.
- Recurrent layers (LSTM) and 1D convolution layers (Conv1D): Used for rank-3 tensors (samples, timesteps, features), suitable for sequence data.
- 2D convolution layers (Conv2D): Used for rank-4 tensors, appropriate for image data.

Layers can be thought of as the LEGO bricks of deep learning, particularly in Keras, where building models involves assembling these layers into functional pipelines.

A deep learning model is essentially a graph of layers, represented by the Model class in Keras. There are more complex network topologies, including:

- Two-branch networks
- Multi-head networks
- Residual connections

An example of a complex topology is the Transformer (Vaswani et al., 2023), used for processing text data. You can build models in Keras either by subclassing the Model class or using the Functional API, which is more concise.

Choosing a network topology defines a hypothesis space, constraining the series of tensor operations mapping input data to output data. This structure encodes assumptions about the problem and prior knowledge the model starts with. For instance, a single Dense layer with no activation assumes the classes are linearly separable (Chollet, n.d.).

Selecting the right architecture is more art than science, relying on best practices and intuition developed through practice. Principles for building neural networks and developing intuition on what works for specific problems are crucial for constructing these networks, selecting the right learning configurations, and tweaking models for optimal results.

After defining the model architecture, you need to choose three additional components:

- Loss function (objective function): The quantity minimized during training, representing the task's measure of success.
- Optimizer: Determines how the network updates based on the loss function, with Adam being a popular choice that combines the benefits of AdaGrad and RMSProp.

- Metrics: Measures of success monitored during training and validation, like classification accuracy. Unlike the loss, training doesn't directly optimize for these metrics, so they don’t need to be differentiable.

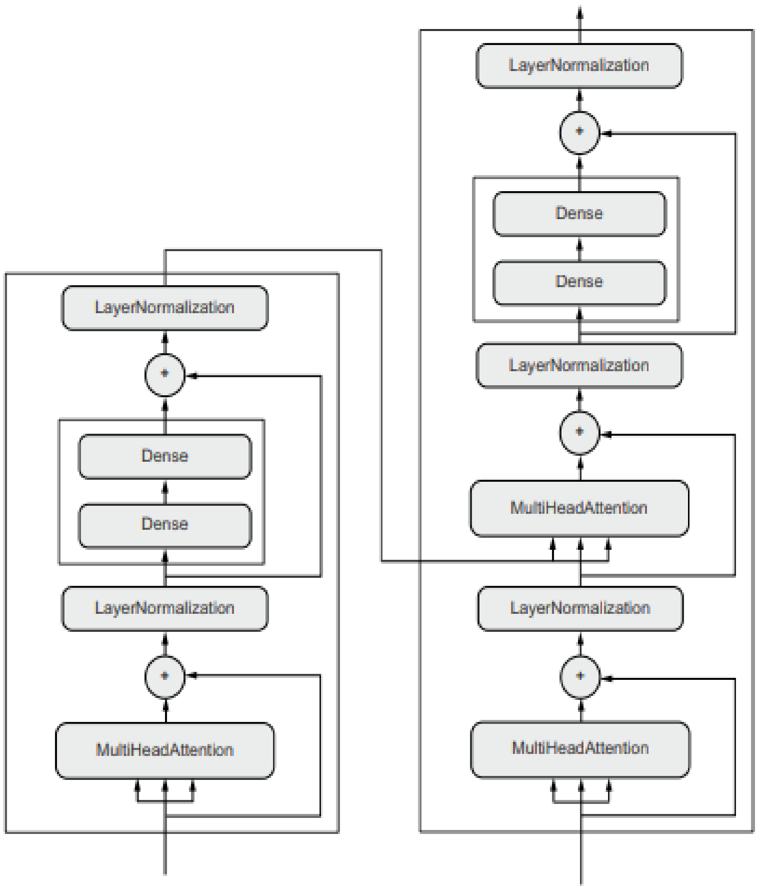


Figure III-20. Transformer Architecture.

3.10.6 Sci-Kit Learn

The development of Scikit-Learn began as part of the Google Summer of Code (GSoC) in 2007 by David Cournapeau. The project was originally called "scikits.learn" and was meant to be a part of the "SciKits" (SciPy Toolkits) collection. The goal was to create a unified and easy-to-use framework for Manning Publications Co. Machine learning in Python. Over the years, it has grown substantially with contributions from a wide community of developers.

It is a powerful, open-source machine learning library for the Python programming language. It provides simple and efficient tools for data mining and data analysis, and it is built on top of other robust libraries such as NumPy, SciPy, and matplotlib. Scikit-Learn is known for its simplicity, consistency, and accessibility, which make it a popular choice for both beginners and experienced practitioners in machine learning.

Scikit-Learn benefits from a vibrant and active community. The development process is transparent and involves contributions from a wide range of individuals, including researchers, industry practitioners, and students. The project is hosted on GitHub, where contributors can submit issues, feature requests, and pull requests. The documentation is comprehensive and includes user guides, tutorials, and API references.

It is mainly used in the project for its metrics module, which is known for its processing capabilities, as well be discussed later.

Chapter IV
Data Acquisition

In today's data-driven environment, any simulation project's success is determined by the quality and comprehensiveness of its data collecting procedure. Data collecting is the foundation of simulations, providing the necessary inputs to generate accurate and dependable models. Hence, the fidelity of simulations is directly proportional to the quantity of the data underpinning them. Data gathering, by capturing real-world occurrences in a structured and accurate manner, allows for the reproduction of complex systems, predictive analytics, and decision-making processes. As we go deeper into the complexities of mimicking real-world events, it becomes clear that solid data collecting is more than just a first step; it is a necessary foundation for gaining meaningful and actionable insights.

Using a pre-existing program, the most prominent example being *Personal Computer Transient Analyzer* (PC-Tran), to generate data instead of building a model from scratch is an efficient and a more reliable way to conserve both effort and computational power.

4.1 PC-Tran

PC-Tran is a reactor transient and accident simulation software designed to run on personal computers. It features a Graphical User Interface (GUI) that adheres to the Microsoft Windows environment specifications. The data input and output are in the MDB format, which is compatible with Microsoft Office Access databases (Qi et al., 2022)

4.1.1 PC-Tran PWR 3-Loop Simulator Description

Following various literatures (Racheal et al., 2022) the break sizes introduced were in between 0% and 100%. This posed a significant problem later upon analysis, since the decline observed at, for instance, 100% break was not in par with the expected values.

After examining the pressure drop obtained in 100% break showcased in Figure IV-1 and comparing it to the graph mentioned above, Figure I-4, the problem is now defined: the pressure at PCTran simulation stabilizes at a given value after its decline, not undergoing any further decrease, which is not entirely realistic.

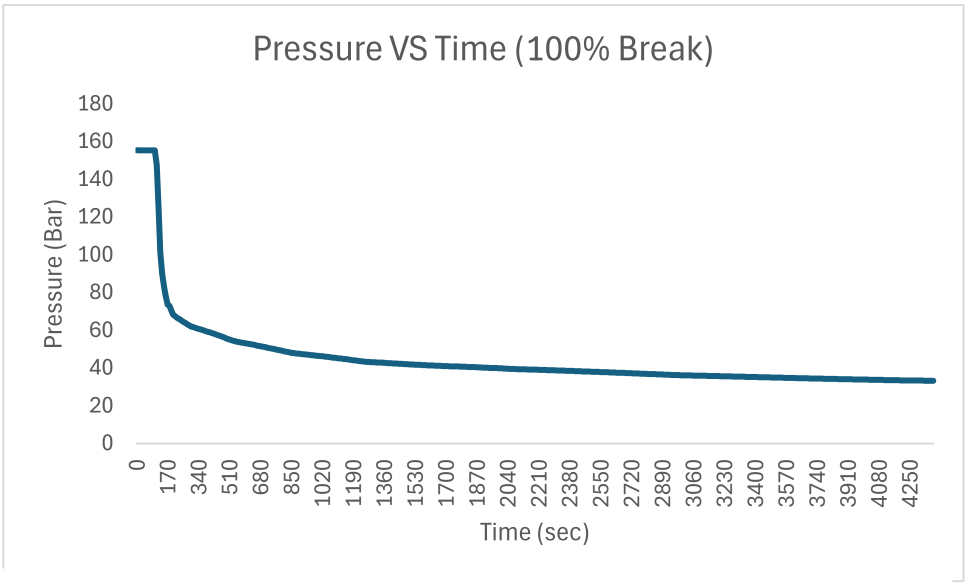


Figure IV-1. Pressure Drop in PC-Tran at 100% Break

Therefore, upon further consideration and literature review (Xiao et al., 2024), it was concluded that the percentages taken into account are of 100 cm^2, and not of the whole area of the pipe. Hence, the diameter of piping of the reference reactor for PCTran Simulator were obtained, and the maximum break size area was calculated (NRC, n.d.).

Table 1. Reactor Coolant Piping Design Parameters

Pipe	Diameter (In Inches)
Inlet piping - inside diameter	27.5"
Inlet piping - nominal thickness	2.69"
Outlet piping - inside diameter	29.0"
Outlet piping - nominal thickness	2.84"
RCP suction piping - inside diameter	31.0"
RCP suction piping - nominal thickness	2.99"

By using the outlet piping inside diameter, the cross-section area for maximum break was calculated.

$$Cross - Section\ Area\ for\ Maximum\ Break = \\ {}^{\pi}/_{4} \times (29 \times 2.5)^2 = 4128.25\ cm^2 \tag{III - 24}$$

By inserting that maximum break into PCTran simulator, the Figure IV-2 for pressure drop is obtained.

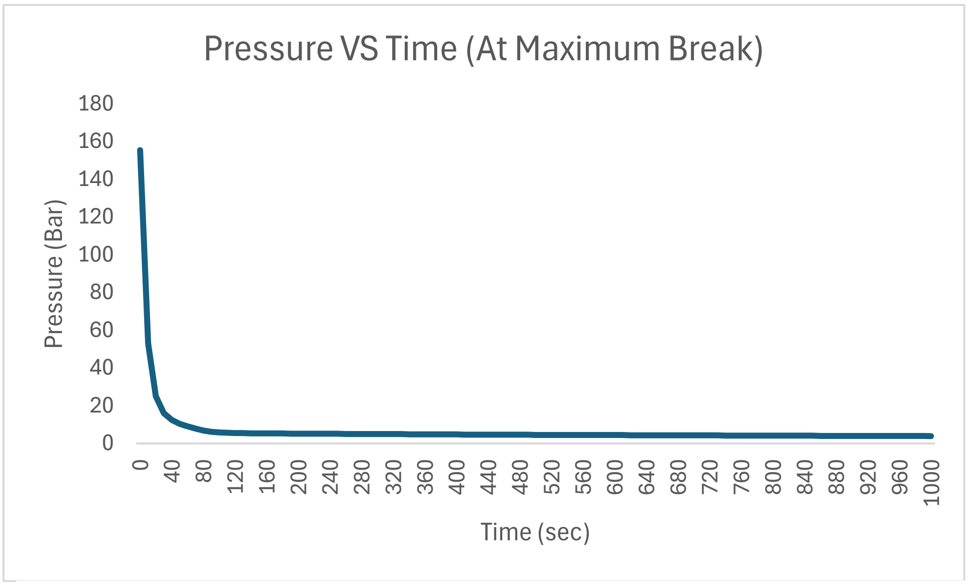


Figure IV-2. Pressure Drop in PC-Tran at Maximum Break

4.1.2 Reasons for Discarding PCTran

While the results of the "pressure drop" was viable after various testing, and despite various literature using PCTRAN, the PCT temperature does not follow the expected trend, discussed in 1.1

Also, PCTran's main company, *Microsim Technologies*, is on hiatus, which leaves the simulator outdated.

PCTRAN data was used as a foundation for model architecture creation, mainly due to their availability and wide usage. It is clear that the data is rather simplistic and is not a metric of real reactor data, but due to their similar structure to real data they are more than enough for building the model, and working on hypertuning the parameters later on.

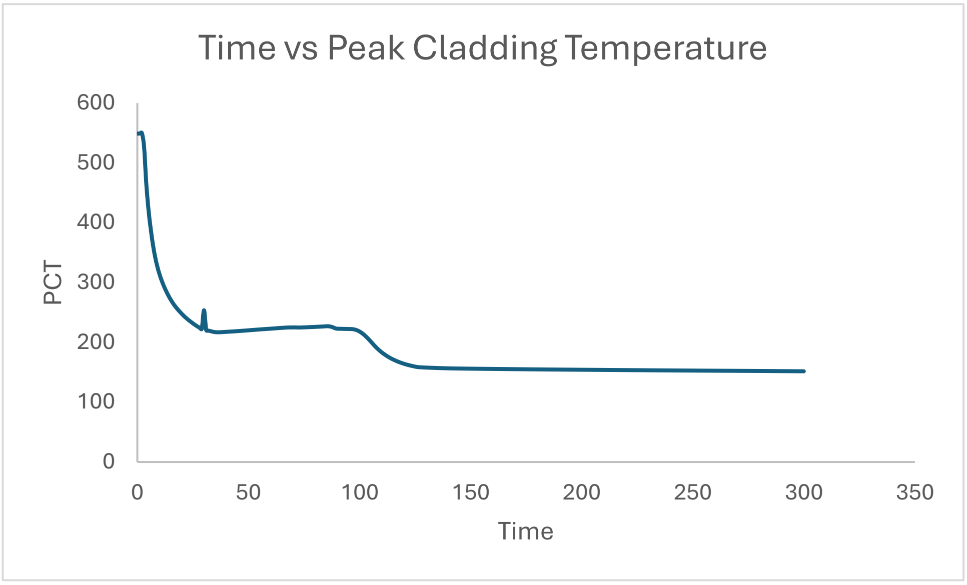


Figure IV-3: PCT Trend in PCTRAN Simulation

4.2 Generic PWR Simulator

The generic simulator (GPWR), developed by Western Services Corporations, whose reference plant is *Palo Verde* in The United States, allows the performance of complete plant startups, shutdowns, and load maneuvers, as well as realistically replicate normal and abnormal plant transients, including malfunction scenarios.

The generic simulator is a pressurized water reactor with two circulation loops, two steam generators and four reactor coolant transfer pumps. The Reactor total thermal power is approximately 4000 MW, and the turbine electric power is approximately 1400 MW.

4.2.1 Generic PWR Simulator Specifications

Table 2. Plant Specifications

Plant Item	Description
Reactor Core Power (Nominal)	3983 MW
Pressurizer Pressure (Nominal)	158.2 Kg/cm²
Hot Leg Temperature	325 ºC
Coolant Inlet Temperature	292.6 ºC
Outlet Pipe (Hot Leg) Diameter	106.7 cm
Inlet Pipe (Cold Leg) Diameter	76.2 cm
Average Temperature Rise in Vessel	32.4 ºC
Average temperature in vessel (Nominal)	309.2 ºC
Initial Conditions	Middle of Life (MOC)

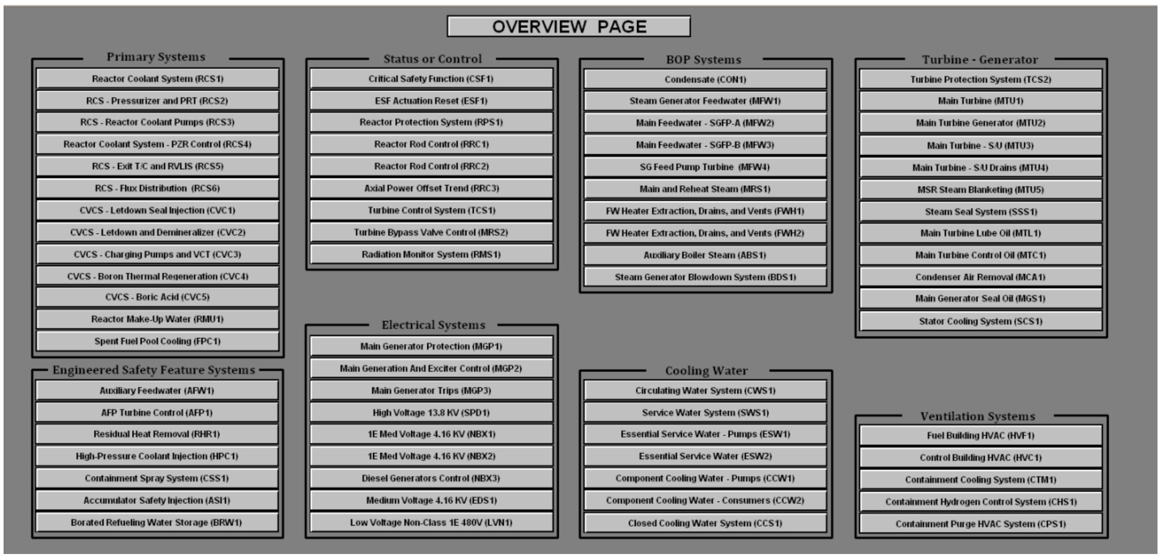

OVERVIEW PAGE
Primary Systems
Reactor Coolant System (RCS1)
RCS - Pressurizer and PRT (RCS2)
RCS - Reactor Coolant Pumps (RCS3)
Reactor Coolant System - PZR Control (RCS4)
RCS - Exit T/C and RVLIS (RCS5)
RCS - Flux Distribution (RCS6)
CVCS - Letdown Seal Injection (CVC1)
CVCS - Letdown and Demineralizer (CVC2)
CVCS - Charging Pumps and VCT (CVC3)
CVCS - Boron Thermal Regeneration (CVC4)
CVCS - Boric Acid (CVC5)
Reactor Make-Up Water (RMU1)
Spent Fuel Pool Cooling (FPC1)
Engineered Safety Feature Systems
Auxiliary Feedwater (AFW1)
AFP Turbine Control (AFP1)
Residual Heat Removal (RHR1)
High-Pressure Coolant Injection (HPC1)
Containment Spray System (CSS1)
Accumulator Safety Injection (ASI1)
Borated Refueling Water Storage (BRW1)
Status or Control
Critical Safety Function (CSF1)
ESF Actuation Reset (ESF1)
Reactor Protection System (RPS1)
Reactor Rod Control (RRC1)
Reactor Rod Control (RRC2)
Axial Power Offset Trend (RRC3)
Turbine Control System (TCS1)
Turbine Bypass Valve Control (MRS2)
Radiation Monitor System (RMS1)
Electrical Systems
Main Generator Protection (MGP1)
Main Generation And Exciter Control (MGP2)
Main Generator Trips (MGP3)
High Voltage 13.8 KV (SPD1)
1E Med Voltage 4.16 KV (NBX1)
1E Med Voltage 4.16 KV (NBX2)
Diesel Generators Control (NBX3)
Medium Voltage 4.16 KV (EDS1)
Low Voltage Non-Class 1E 480V (LVN1)
BOP Systems
Condensate (CON1)
Steam Generator Feedwater (MFW1)
Main Feedwater - SGFP-A (MFW2)
Main Feedwater - SGFP-B (MFW3)
SG Feed Pump Turbine (MFW4)
Main and Reheat Steam (MRS1)
FW Heater Extraction, Drains, and Vents (FWH1)
FW Heater Extraction, Drains, and Vents (FWH2)
Auxiliary Boiler Steam (ABS1)
Steam Generator Blowdown System (BDS1)
Cooling Water
Circulating Water System (CWS1)
Service Water System (SWS1)
Essential Service Water - Pumps (ESW1)
Essential Service Water (ESW2)
Component Cooling Water - Pumps (CCW1)
Component Cooling Water - Consumers (CCW2)
Closed Cooling Water System (CCS1)
Turbine - Generator
Turbine Protection System (TCS2)
Main Turbine (MTU1)
Main Turbine Generator (MTU2)
Main Turbine - S/U (MTU3)
Main Turbine - S/U Drains (MTU4)
MSR Steam Blanketing (MTU5)
Steam Seal System (SSS1)
Main Turbine Lube Oil (MTL1)
Main Turbine Control Oil (MTC1)
Condenser Air Removal (MCA1)
Main Generator Seal Oil (MGS1)
Stator Cooling System (SCS1)
Ventilation Systems
Fuel Building HVAC (HVF1)
Control Building HVAC (HVC1)
Containment Cooling System (CTM1)
Containment Hydrogen Control System (CHS1)
Containment Purge HVAC System (CPS1)

Figure IV-4. Plant Overview in Simulator

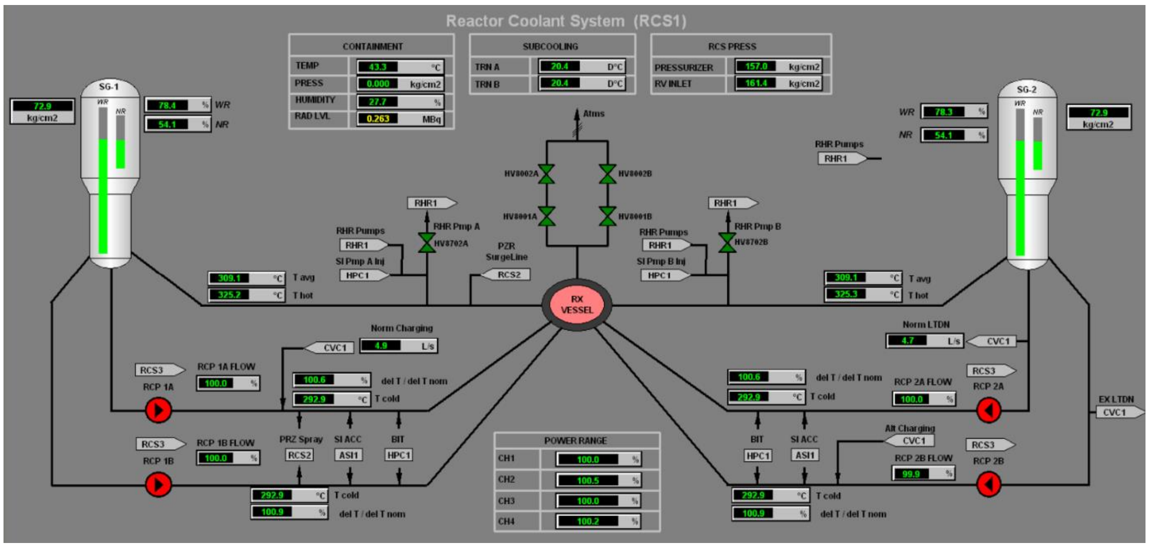


Figure IV-5. Reactor Coolant System Interface

4.2.2 Scenario of LOCA in Generic PWR Simulator

The GPWR simulator employs a straightforward procedure for initiating a Loss of Coolant Accident (LOCA). The desired malfunction *“Loss of Coolant in 1A Cold Leg”* is selected from a predefined list, and the break percentage is then introduced into the system. In contrast to PCTran, the simulator expresses this percentage as a fraction of the total pipe area, rather than as a percentage of 100 cm². This approach simplifies the process considerably.

To optimize time and resource allocation, the data generation process was conducted concurrently across multiple devices. This parallel approach necessitated a rigorous verification process to ensure the congruence of Initial Conditions (ICs) files across all devices utilized. These IC files, which are pre-existing files within the simulator's database, play a crucial role in establishing consistent baseline parameters for each simulation.

The synchronization of ICs files across all devices was paramount to maintaining the integrity and comparability of the generated data sets. This methodical approach to file consistency served to mitigate potential discrepancies that could arise from variations in starting conditions, thereby enhancing the overall accuracy and reliability of the simulated results.

The simulator incorporates a vast array of observable variables. Consequently, the careful selection of relevant parameters was crucial to the experimental process. Initially, seventy-five parameters were identified as potentially significant. This number was subsequently reduced to twenty-three to enhance the model's accuracy and efficiency. The selected parameters were systematically documented and archived in a dedicated file. This file was designed to be compatible with the simulator's monitored parameters list import function. The creation of this standardized parameter set facilitated streamlined operations across multiple devices, enhancing efficiency and consistency in the data collection process.

For each simulation, the system was allowed to run for a duration of five minutes. The resulting data was then archived in a comma-separated values (CSV) file to facilitate further analysis. Approximately 187 files were generated, each representing a different break percentage, with increments of 0.5% between successive simulations.

As detailed in Appendix A, an automation script was developed to expedite the data generation process. However, it should be noted that a significant portion of the data acquisition was still performed manually.

4.2.3 Challenges Encountered in Simulator Operation

While the simulator significantly streamlined the data generation process, alleviating many of the challenges encountered with PC-Tran's operation, it nonetheless presented its own set of operational complexities.

In certain scenarios, the simulator exhibited an anomaly wherein it would maintain the power at its maximum value despite the active malfunction. This issue could generally be resolved by resetting the system to its initial conditions and reinitiating the simulation. However, in some instances, multiple reset attempts were necessary to fully rectify the problem.

In certain instances, upon initializing the simulator on a previously utilized device, an unexpected behavior was observed wherein the simulator would purge the existing parameters from the monitored parameters list, resulting in a blank configuration. This intermittent issue underscored the critical importance of the aforementioned parameter documentation file.

A particularly problematic range of break percentages was identified between 13% and 20%. Within this range, the system generated a cascade of error messages, indicating that either the power had exceeded 110% of its nominal value or that the pressure had surpassed 20.68 MPa. Notably, these errors were often preceded by an anomalous temperature reading in the RCS5 window, where values would plummet to -273 degrees Celsius (absolute zero). Such readings are incongruent with both the inserted malfunction parameters and the expected behavior of a reactor under normal or abnormal operating conditions.

Upon investigation, it was determined that these errors stemmed from a division-by-zero occurrence in the underlying code. However, due to limited technical support resources, this issue remained unresolved. Consequently, data files corresponding to the aforementioned problematic range of break percentages could not be procured.

These operational constraints highlight the importance of robust error handling and comprehensive technical support in simulation environments, particularly when dealing with complex systems such as nuclear reactors.

Chapter V
Feature Selection and Model

5.1 Feature Selection

Feature selection, demonstrated in Figure V-I is a critical step in developing effective AI models. It involves identifying and choosing the most relevant variables or attributes from a larger set of input data to use in training the model. This process helps reduce dimensionality, improve model performance, and enhance interpretability. Effective feature selection can lead to faster training times, reduced overfitting, and more robust models.

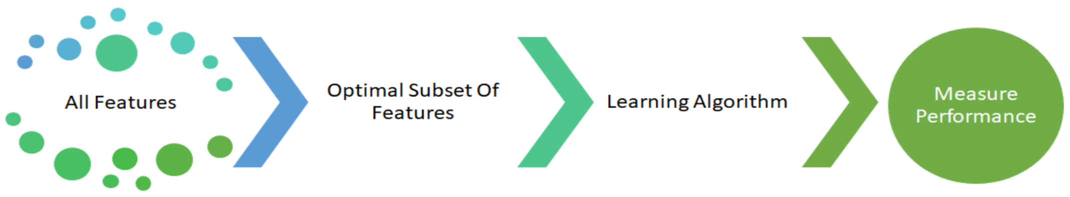


Figure V-1. Demonstration of Feature Selection

In section 4.2.2 a strategic feature selection process was employed to refine the input parameters for the AI model. Initially, the simulator generated seventy-five parameters. However, this number was systematically reduced to twenty-three through a manual selection process. The parameters were first categorized into three groups based on their perceived importance. The selection criteria primarily focused on two key factors: the rate of change of each parameter and its overall significance to the model. Parameters that exhibited repetitive behavior yielding identical results were eliminated. Similarly, parameters that remained constant over time were also discarded, as they provided little discriminative value to the model. This targeted approach to feature selection ensured that only the most informative and dynamic parameters were retained, potentially improving the model's efficiency and predictive power.

For example, the Boron Concentration, in Figure V-2. and Figure V-3., proved to be one of the most important parameters to be fed into the model due to its significance

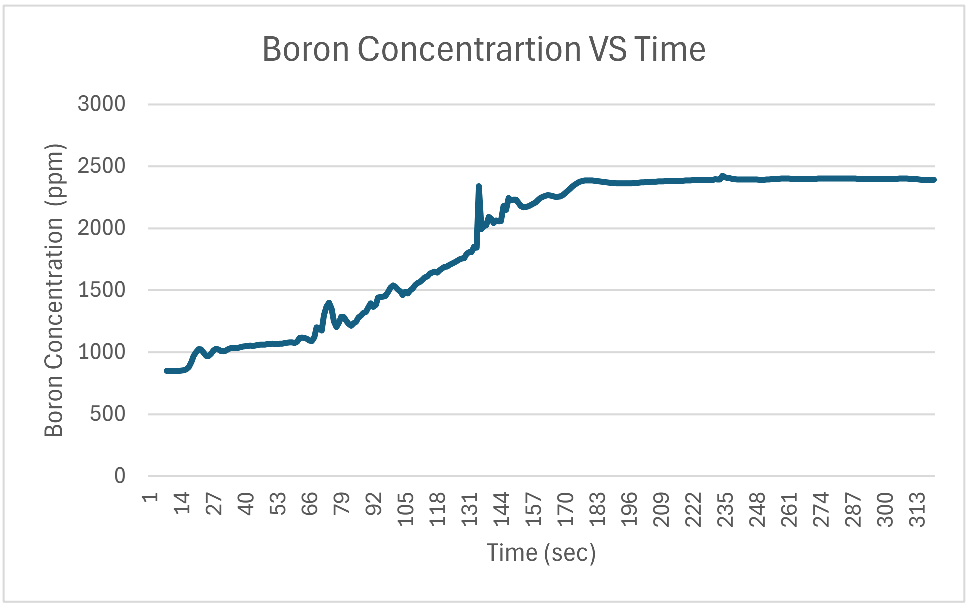


Figure V-2. Boron Concentration VS Time at 10% Break

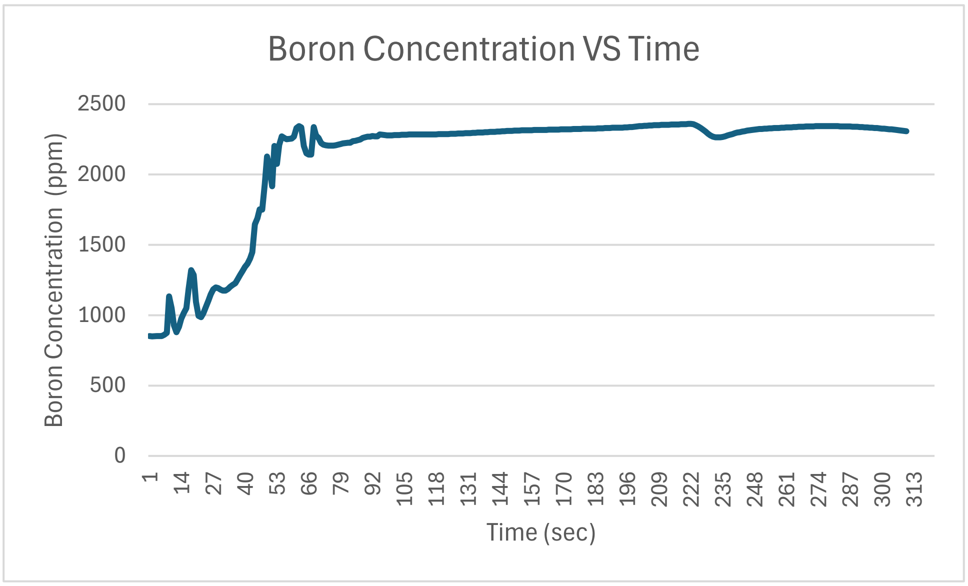


Figure V-3. Boron Concentration VS Time at 100% Break

Conversely, certain parameters were identified as crucial for the model's performance. Specifically, variables related to system dynamics, such as temperatures, pressures, and flow rates within various loops, were determined to be highly beneficial. These parameters were recognized for their ability to provide meaningful, time-varying data that could significantly enhance the model's predictive capabilities. By retaining these variables, the feature selection process ensured that the model would have access to critical information about the system's thermal and hydraulic behavior, potentially improving its accuracy and reliability in simulating or predicting LOCA scenarios.

For example, the Wide Ring Channel 1 Pressure displayed significant change as the LOCA incident progressed as shown in Figure V-4, Figure V-5, making it a valuable parameter.

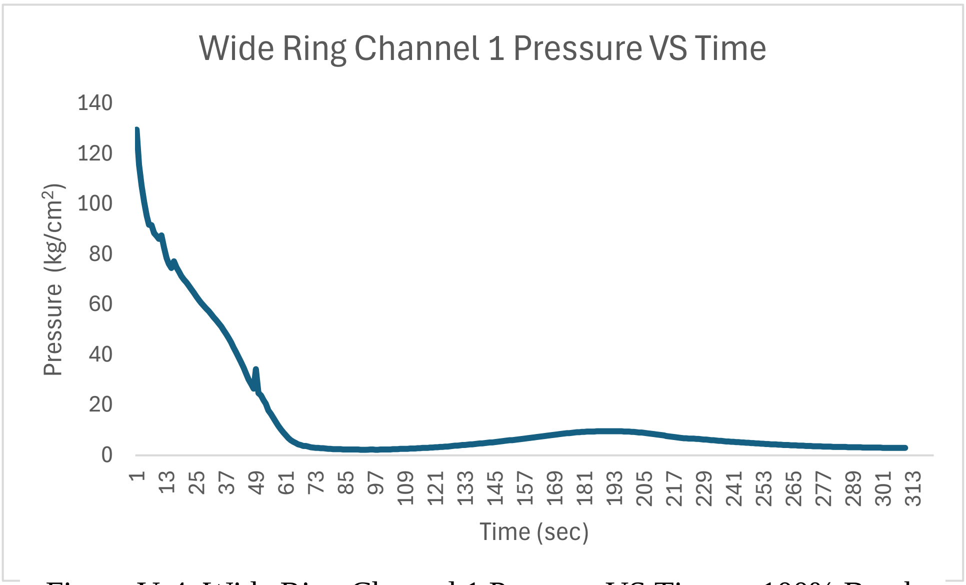


Figure V-4. Wide Ring Channel 1 Pressure VS Time at 100% Break

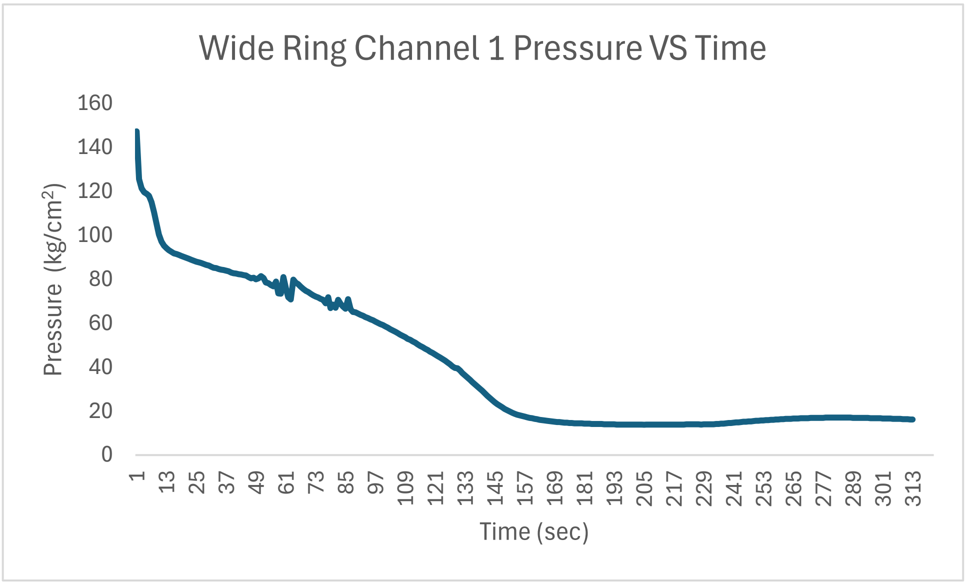


Figure V-5. Wide Ring Channel 1 Pressure VS Time at 10% Break

As previously noted, the feature selection process involved further elimination of parameters based on two key criteria: redundancy and relevance. Some variables were removed due to their repetitive nature, providing duplicate information already captured by other parameters. Others were excluded because they were deemed less significant for the model's objectives.

A notable case in point was the Generated Power Percentage parameter in Figure V-6 and Figure V-7. Despite its apparent importance, this variable was ultimately discarded. The rationale behind this decision was that its behavior—characterized by an extremely rapid transition from maximum to zero—offered little valuable insight to the model. Such an abrupt change provided minimal informative data for the AI to learn from, effectively rendering the parameter non-contributory to the model's predictive capabilities.

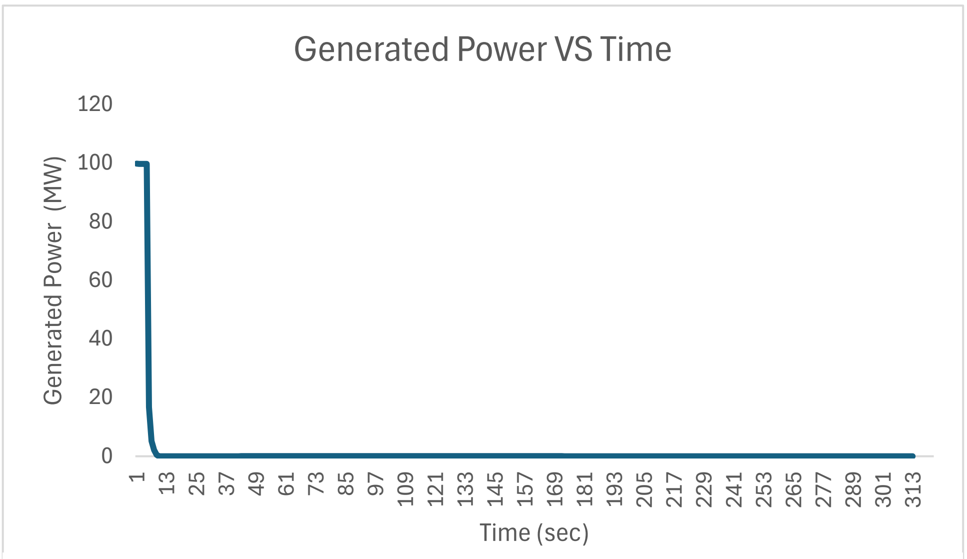


Figure V-6. Generated Power Percentage VS Time at 10% Break

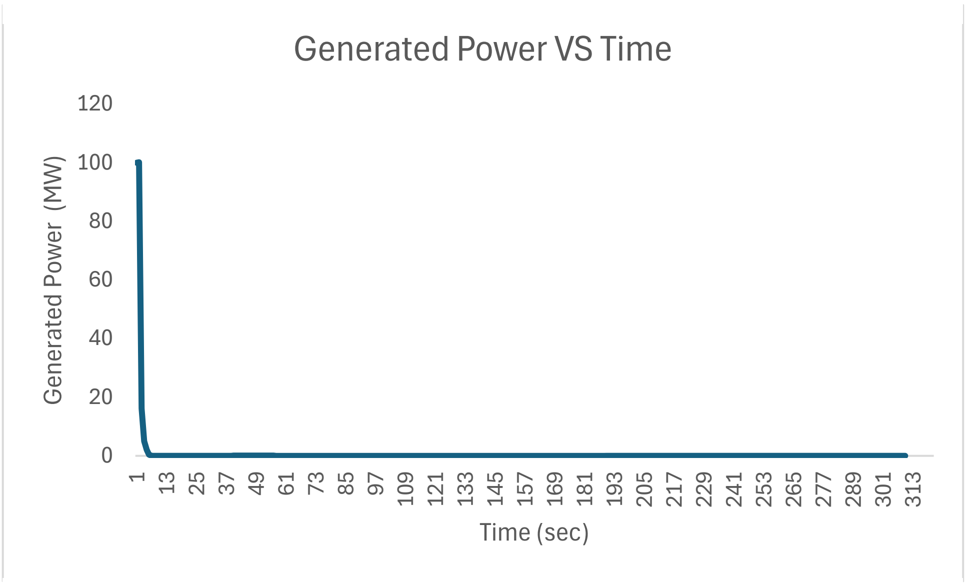


Figure V-7. Generated Power Percentage VS Time at 100% Break

The Control Rod Position was another parameter eliminated during the feature selection process. This decision was based on the predictable behavior of control rods following LOCA initiation. In such scenarios, the reactor undergoes an immediate scram, causing all control rods to fully insert within seconds. Given this rapid and uniform response, the Control Rod Position parameter offered limited unique insights. It was recognized that other parameters could effectively convey similar information while providing additional value to the model.

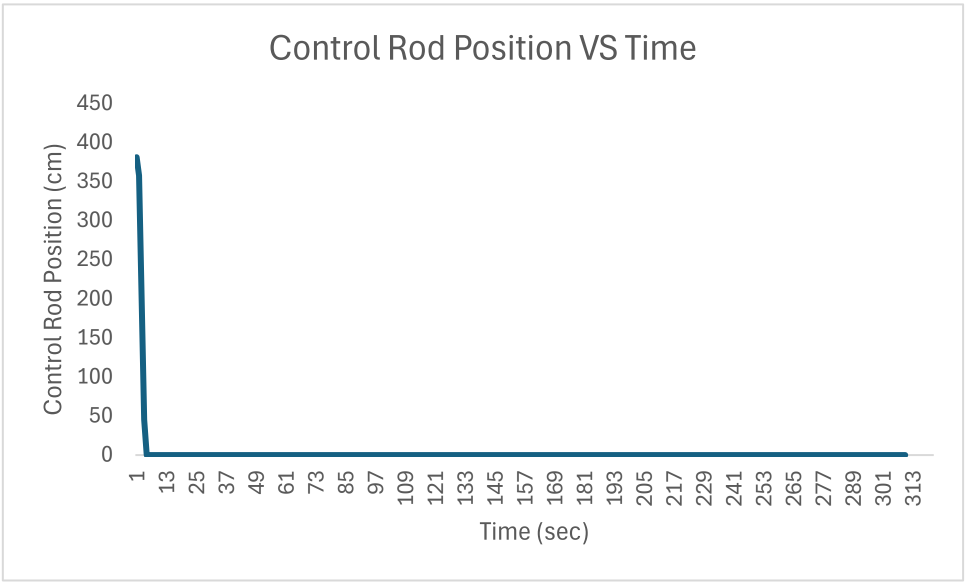


Figure V-8. Control Rod Position VS Time (Regardless of Break Size)

A crucial point to emphasize is the consistency of the Control Rod Position parameter's behavior across various break sizes in LOCA scenarios. Regardless of the magnitude of the break, this parameter exhibited remarkably uniform patterns, producing nearly identical graphical representations in each case. Ultimately, both of the parameters discussed above are to be considered an indicatin of reactor shutdown, and they provide redundant information which is also carried by the Thermal Power.

To ensure the robustness and accuracy of the manual feature selection process, a validation step was implemented. Means and standard deviations for the data were calculated. This computational approach served as a safeguard against potential oversights or biases in the human-driven selection process.

5.2 Domain Knowledge Based Feature Selection Algorithm

Developing an AI model consists of several perquisites. The choice of platform, of course, is a priority as well as the "handling" of the data. It is mentioned in section 4.2.3 that the WSC

Simulator was run by a parameters file that had a 73-column output, corresponding to variables that are affected post-LOCA for around 300 seconds – corresponding to rows - on average. A total of 187 files were generated, excluding the 0% percent file, which is just the reactor running for 5 minutes under normal conditions, that leaves 186 files. Manual feature selection by domain knowledge selected from 73 parameters only 23 (including the Time parameter) as mentioned in detail above. An Algorithm was used to map the simulator variable names to readable-by-experts names. The Algorithm was as follows:

```
(1) Start
(2) Initial Setup
    • Load Excel file containing parameters (A pre-made excel file with 1-to-1 mapping
      of variable names).
    • Define Huge_folder and recieve_folder.
    • List files in Huge_folder.
(3) Create Mapping Dictionary
    • Initialize mapping and Mapping dictionaries.
    • For Each Sheet in Excel File:
        o Load sheet into DataFrame.
        o Identify key and value columns.
        o Populate mapping dictionary.
    • Clean up keys and populate Mapping dictionary.
(4) Process Each File
    • For Each File in list_of_files:
        o Construct file path.
        o Detect file encoding.
        o Read CSV file into DataFrame.
        o Drop first four rows and reset index.
        o Rename 'Variable' column to 'time'.
        o Clean column names.
        o Rename columns using Mapping dictionary.
        o Construct output file name.
        o Save processed DataFrame as Excel file in recieve_folder.
(5) End
```

The above algorithm is implemented in python, as can be seen in Appendix A. The algorithm, however, is not implemented in the data used later as input for the AI model and was only used during the domain-knowledge-based feature selecting phase.

5.3 Pre-preparation of Input Data

It is also desirable to separate the process of reduction of parameters from the "main" code and generate 186 files with constant parameters because they can be reused at will without interfering with the process and will not affect the readability of the data. The Algorithm for this process is simple:

```
(1) Start
(2) Define Relevant Variables - Initialize relevant_variables list.
(3) Set Input/Output Directories - Initialize input_directory and output_directory.
(4) Check Output Directory Existence:
      o If Exists - Proceed to next step.
      o If Not - Create output_directory.
(5) Process Files in Input Directory:
      o For Each File:
          ▪ If File Ends with .csv:
              • Read CSV File - Load file into DataFrame (df).
              • Identify Relevant Columns - Select columns containing any
                of the relevant_variables substrings.
              • Save New CSV File - Write filtered DataFrame to new CSV
                in output_directory.
(6) End
```

The Relevant Variables, of course, refer to the 23 pre-selected ones. The Python Implementation of the above code is provided in Appendix A.
Ultimately, only 10 of those 23 pre-selected variables are used. This is due to testing the model with various parameters, and calculations of standard deviations of multiple columns and comparing with each other. The choice had also one other criteria: None of the parameters are to be repeated, despite being useful, so as not to force the model to find correlations

where the correlation is too trivial.

5.4 Model Building Approach

Now that we have the data input files prepared, and our external python libraries selected, the next step is to build an approach for machine learning.

We can identify the steps for building a successful machine learning model as the following:

(1) Data Loading
(2) Data Pre-processing
(3) Model Creation & Training
(4) Data post-processing

In the subsequent sections we will cover each of these steps in detail.

5.4.1 Data Loading

The Algorithm for reading the 186 input files is simple enough. The one problem with this algorithm is the fact that the data generation process was over a long period time, with multiple contribution from different authors and the WSC Cold Break LOCA automation code. Hence, the nomenclature was slightly different for the files, and the files must be read in order (For example, you can’t parse all the files in the folder using the OS module because their order will change)

The Algorithm for data loading of the input data is therefore:

(1) **Initialize empty list as *sequences***: The list is intended to contain all of the input data as lists within this list.
(2) **Initialize the location of all data as *path***: The path where input data is contained.
(3) **Initialize the maximum number of rows allowed as *max_rows***: This ensures that any input that ran for more than exactly 5 minutes only has 5 minutes of data, so the data is homogenous and 1-to-1.
(4) **Loop through all the 186 files**: This is done through a for loop, with the iterator “*i*” set to be an incrementing value from 0.5 to 100. It’s Important to note that the range

function can only generate integers for the iterator, hence Numpy *arange* function is used to initialize the loop instead.

a. **Check if the iterator is a float** (if modulus of 1 is not 0): This is done because *arange* generates floats. This is convenient for actual floats (eg. *1.5)*, however, numbers such as "*2"* will be treated as "2.0" and this is inconvenient for file reading, hence the need to separate integers from floats.
    i. **Try reading the file**:
        (1) Initialize file path as *file_path:* The file path is the earlier defined *path* plus the iterator number plus "%_processed.csv" to indicate that the file has been subjected to the earlier mentioned feature reduction algorithm. An example of the file name would be: "/directory/1%_processed.csv".
        (2) Read 300 rows of the file using pandas *read_csv(file_path)* function into a dataframe "df" and the selected 10 columns.
        (3) Drop the "hmi_RCSLT501_VALUE_BAR" column if it exists: This is done to ensure that there aren't two readings of "RCSLT501", as the feature selection algorithm depends on the name and there have been instances of a couple of input data with two readings of this parameter with different units.
        (4) Convert df to a python list "*values"*: This returns a list that contains all the rows and columns of the selected csv file. The shape of each list would therefore be (300 rows, 10 columns)
        (5) Append the list "*sequences"* the list "*values"*: This leads to a list with (value of the iterator "*i*", 300, 10).
    ii. If the reading of the file fails due to "*file not found error*":
        (1) **Try reading the file**:
            a. Initialize file path as *file_path:* The only change from the last time is "_processed.csv", without the percentage. Example of the file name would be: "/directory/1_processed.csv".
            b. Repeat step 2, 3, 4 & 5 from i.
        (2) If the reading of the file fails due to "file not found error":

        a. Print "File Not Found": This is done to ensure that the algorithm does not stop if the iterator corresponds to a file that does not exist, mainly due to the fact that a few percentages are missing as mentioned in 4.2.3

b. **Else if the iterator is an integer**:
    i. Make Sure the iterator is treated as an integer.
    ii. Repeat steps i & ii from step "a".

Now we have the input data, the output data is going to be the break size which is just a number from *0.5* to *100* with increments of 0.5 to represent the break sizes, with exception of the non-existing files. A simple for loop can generate the necessary target data. From now on the input data will be referred to as *X_data* and *y_data*. There is still a need to shuffle the data and separate it to training, test and validation. The procedure for shuffling and separating the data is done via a simple algorithm that utilizes NumPy's arrays capabilities:

(1) Convert "*sequences"* to a NumPy array
(2) Convert "*y_data"* to a NumPy array
(3) Find the length of "*sequences"*
(4) Based on that length, generate an array with numbers from 0 to the number of elements minus 1: This array would correspond to indices, which are both equal in number in both "*sequences"* and "*y_data"*.
(5) Shuffle the *"indices"* array using the random module's shuffle function within NumPy.
(6) Rearrange both "*sequences" and* "*y_data"* using the generated shuffled indices.

Separation of data into training, validation and test is done using a 7:2:1 ratio via array slicing, arrays *"X_train", "X_test", "X_val"* are generated with their corresponding y-targets. The python implementation of this code is available the Main Model Algorithm section in Appendix A.

5.4.2 Data Pre-processing

Data pre-processing refers mainly to the process of selecting a suitable scaling method. It's easy to create bias in ML models' learning process by feeding bigger numbers to their train data. There are multiple methods for scaling data so the model "views" them as equal and consequently avoids creating bias. Some of those methods are discussed in this section below.

5.4.2.1 Min-Max Scaling

It keeps all features within a specific range, [0,1] for example, and has multiple advantages when data does not have any particular distribution and a known maximum and minimum value. The most advantageous property of this type of scaling is the fact that it preserves relationships between data points. This makes it very viable for neural networks. The scaling is done by finding a new data point "x'" through the following formula:

$$x' = \frac{x - x_{min}}{x_{\mathrm{max}} - x_{\mathrm{min}}} \tag{III - 25}$$

However, in our case, preserving the relationships between data points is not advantageous, and leads to the model creating bias towards higher varying non-linear features. It is also very sensitive to outliers, which is typical in the WSC simulator data.

5.4.2.2 Standardization (z-score normalization)

It is frequently used in linear regression, logistic regression, SVM, and clustering algorithms like K-Means. The idea is to center the data around 0 with a standard deviation of 1. Hence, its implementation is done via the formula:

$$x' = \frac{x - \mu}{\sigma} \tag{III - 26}$$

Where $\boldsymbol{\mu}$ represents the mean of the feature, and $\boldsymbol{\sigma}$ is the standard deviation of the feature. This method, while viable, presents the issue of the fact that most of the data output of nuclear accidents do not follow a normal distribution. This presents a challenge for this method.

5.4.2.3 Max-Abs scaling

Scaling by dividing over the maximum value of the feature. It is most useful when the feature is already centered around 0. It is most useful for text processing tasks. The formula is simple:

$$x' = \frac{x}{|x_{\max}|} \tag{III - 27}$$

This method would create obvious bias not only towards higher valued features, but also towards features with changing scales in between break sizes. It is therefore not viable.

5.4.2.4 Mean Normalization

Similar *Min-Max scaling* with the added advantage of centering the data around 0. This method is robust and potentially viable, the formula is:

$$x' = \frac{x - \mu}{x_{max} - x_{min}} \tag{III - 28}$$

There are plenty of other methods, but most of them are for more specific purposes.

After careful consideration, the choices were narrowed down to *mean normalization* and *standardization*. Through trial and error, it was found that *standardization* preforms best with the WSC simulator data.

The implementation algorithm is mostly done by using NumPy functions, with exception of one function done manually to implement the equation mentioned in the *standardization* section above. The python code is available in the appendix.

5.4.3 Model Creation

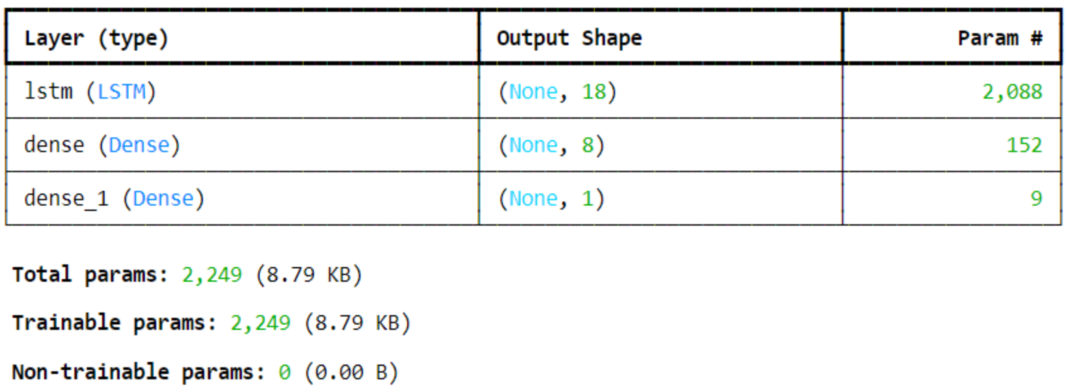
The model creation is done through Keras sequential library. This means that each layer is added in order that the model learns in. The choice of the layers is a single *LSTM* layer with

15 units, and two dense (fully-connected) layers with only 8 and 1 node respectively, which by default uses a tanh activation. This choice of layers has been reached after various trial and error attempts with the goal of reducing the Mean Absolute Error of the results. One other addition to the model was *Elastic Net Regularization*. The model is then compiled with *Adam*, and a learning rate of 0.001. The model is then *fit* with *Keras* on its training data.

A notable addition to the model was the use of a custom loss function. The custom loss function was created to accommodate the fact that post LOCA behavior tends to vary noticeably according to break size as discussed in section 1.2 The loss function is based on MSE, with the addition of a multiplied penalty factor of "9" when the break size is between 80 & 100 as well as a 1.1 factor for small breaks and 1.6 factor in the mid-high section of the breaks. The exact implementation is show in Appendix A. An algorithm summarizing the process is as follows:

(1) **Define a custom loss function**: a. Set penalty factors for different ranges of true values b. Calculate Mean Squared Error (MSE) between true and predicted values c. Apply penalties to MSE based on the range of true values d. Calculate the mean of penalized MSE e. Return the square root of the weighted error
(2) **Import necessary Keras modules and components**.
(3) **Set up regularization strengths for L1 and L2 regularization**.
(4) **Create a Sequential model**: a. Add an *LSTM* layer with 18 units, no return sequences, input shape (300, 10), and L1/L2 regularization b. Add a *Dense* layer with 8 units and linear activation c. Add another Dense layer with 1-unit, linear activation, and L1/L2 regularization.
(5) **Display the model summary**: Shown in Table 3 below.

Table 3. The Model Summary



Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 18)	2,088
dense (Dense)	(None, 8)	152
dense_1 (Dense)	(None, 1)	9

```
Total params: 2,249 (8.79 KB)
Trainable params: 2,249 (8.79 KB)
Non-trainable params: 0 (0.00 B)
```

(6) **Set up a ModelCheckpoint callback to save the best model.**

(7) **Compile the model**: a. Use the custom loss function b. Use Adam optimizer with a learning rate of 0.001 c. Use Root Mean Squared Error as a metric.

(8) **Train the model**: a. Fit the model on training data (X_train, y_train) b. Use a batch size of 32 c. Validate on validation data (X_val, y_val) d. Train for 2000 epochs e. Use the ModelCheckpoint callback

5.4.4 Model Post-Processing

Post-processing is an essential step for model evaluation. This process includes evaluating the model on the test set, de-normalizing the results to display them and calculating errors and losses. An algorithm for post-processing can be as follows:

(1) **Define a postprocess_y function**:
   a. Take an array as input
   c. Reverse the normalization process by multiplying by std_y_train and adding mean_y_train
   d. Return the processed array

(2) **Load the best model from the saved file**

(3) **Make predictions on the test set**:
   a. Use the loaded model to predict on X_test
   b. Postprocess the predictions using postprocess_y function
   c. Postprocess the actual test values (y_test) using postprocess_y function

(4) **Create a DataFrame with test predictions and actual values**

(5) **Calculate and print evaluation metrics**:
   a. Mean Absolute Error (MAE)
   b. Mean Squared Error (MSE)
   c. Root Mean Squared Error (RMSE)
   d. R-squared (R2) score

(6) **Define a function to calculate the percentage of predictions within a tolerance**:
   a. Take actual values, predictions, and tolerance as inputs
   b. Calculate the absolute difference between actual and predicted values
   c. Determine if each prediction is within the acceptable difference
   d. Calculate and return the percentage of accurate predictions

(7) **Calculate the accuracy using the defined function with a 15% tolerance**

(8) **Plot the training and validation loss:**

a. Create a line plot of training and validation loss over epochs
b. Set title, labels, and legend
c. Find the epoch with the lowest validation loss
d. Mark the best model point on the graph with a red dot
e. Annotate the best model point with its epoch and loss value
f. Display the plot

Chapter VI Discussion and Results

The results of the previous code are saved in to an excel file. Particularly, the dataframe created with predictions vs actual values of the test set.

Example results would be as follows:

Table 4: Test Predictions vs Actuals

Test Predictions	Actuals
70.839	70
86.257	93.5
63.342	64
9.554	10.5
82.119	91.5
82.292	90
81.158	95
62.599	57.5
67.243	69
41.368	42.5
4.313	1
29.324	34
87.219	75.5
5.027	3
55.476	51
25.169	28
24.904	24.5
36.229	35
73.265	65

Which can be demonstrated via the Figure VI-1 below:

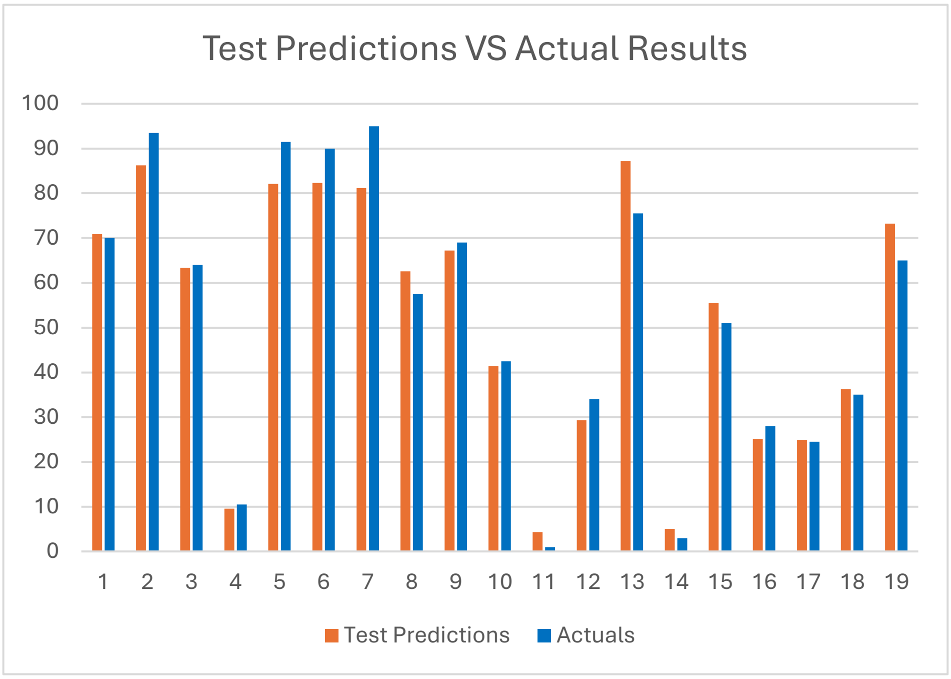


Figure VI-1. Test Predictions VS Actual Results

The metrics of the model's performance are calculated as specified in the previous section. The result are as follows:

Table 5: Model Performance's Metrics

Mean Absolute Error (MAE): 4.607
Mean Squared Error (MSE): 36.640
Root Mean Squared Error (RMSE): 6.053
R-squared: 0.958 Percentage of predictions within tolerance: 84.210%

The validation (red) vs training (blue) loss during each epoch are extracted and graphed in Figure VI-2 as follows:

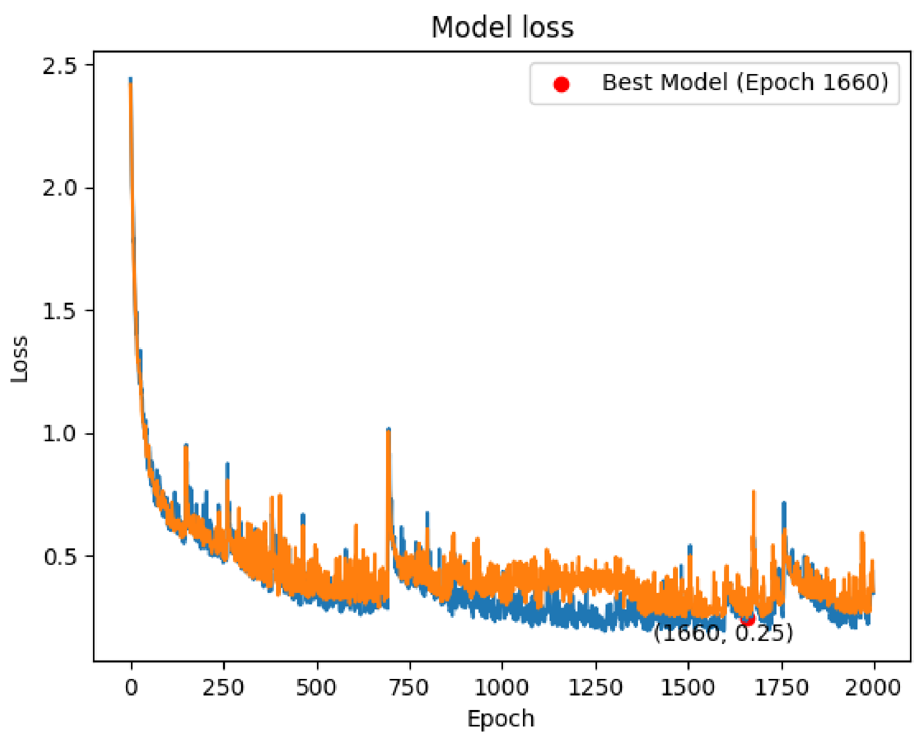


Figure VI-2. Validation Loss VS Training Loss

It can be noted that there are obvious fluctuations in the loss of the training process. This is mainly due to the hard penalties in the custom loss function. Despite this, a smooth model with less fluctuations performs worse.

This can be easily shown by comparing the metrics of a run without the custom loss function with the above.

Mean Absolute Error (MAE): 6.77
Mean Squared Error (MSE): 78.388
Root Mean Squared Error (RMSE): 8.853
R-squared: 0.886
Percentage of predictions within tolerance: 45.0%

The validation vs training loss of this model was as shown in Figure VI-3.

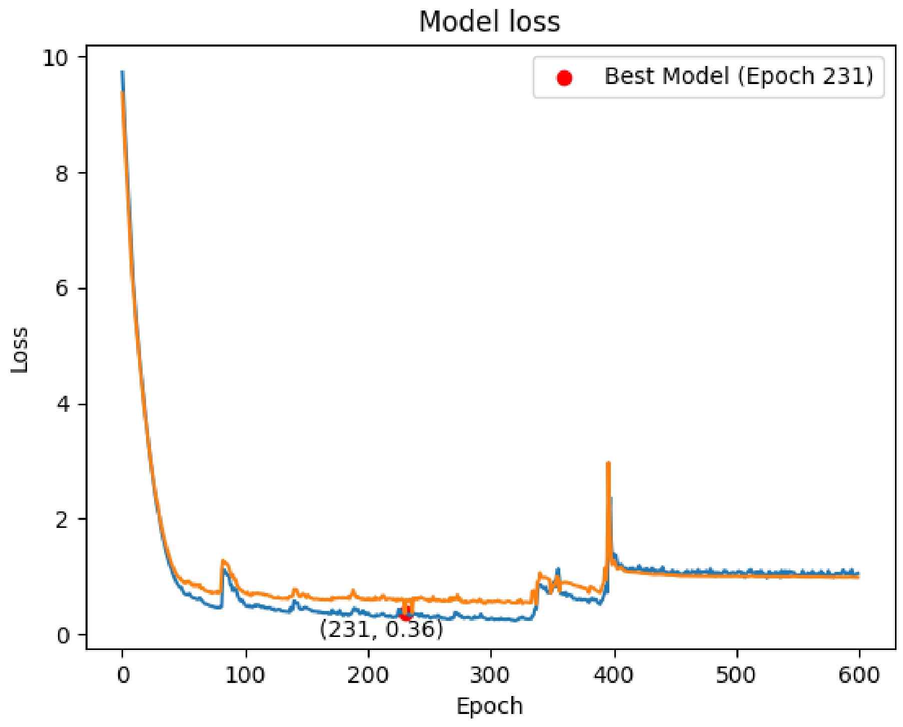


Figure VI-3. Validation Loss VS Training Loss for smooth model

However, it is important to note that this is not a 1-to-1 comparison. Due to the fact that during the data loading process the indices are shuffled, each time the code is run a slightly different result ensues. The custom-loss-function based model is much more consistent in this regard.

However, to avoid this issue, the code is run enough times for the average of the results to converge.

The results of each run are saved in an excel file, and the metrics are calculated for each run, saved in a text file and then averaged. For the python implementation view Appendix A. The average results of 100 runs are as follows:

Table 6: Average of 100 Runs to Determine Model Metrics

Mean Absolute Error (MAE): 5.185
Mean Squared Error (MSE): 76.502
Root Mean Squared Error (RMSE): 7.953
R-squared: 0.889
Percentage of predictions within tolerance 15%: 80.684%

Of course, those are average results that do not represent any real model per se. They are, however, an indication of the algorithm's consistency. The below graphs showcase actual results (smooth) versus the averages (dotted) in the table above.

(1) Cumulative MAE

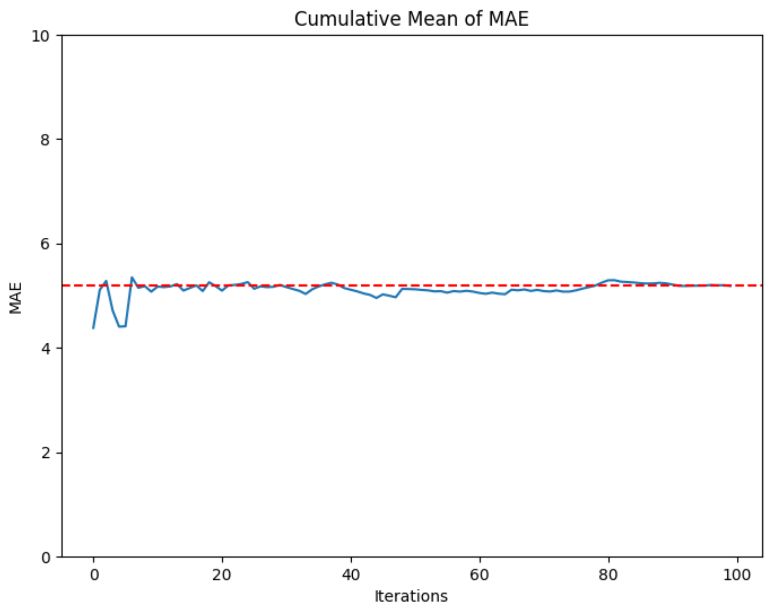


Figure VI-4. Cumulative Mean of Mean Absolute Error

(2) Cumulative MSE

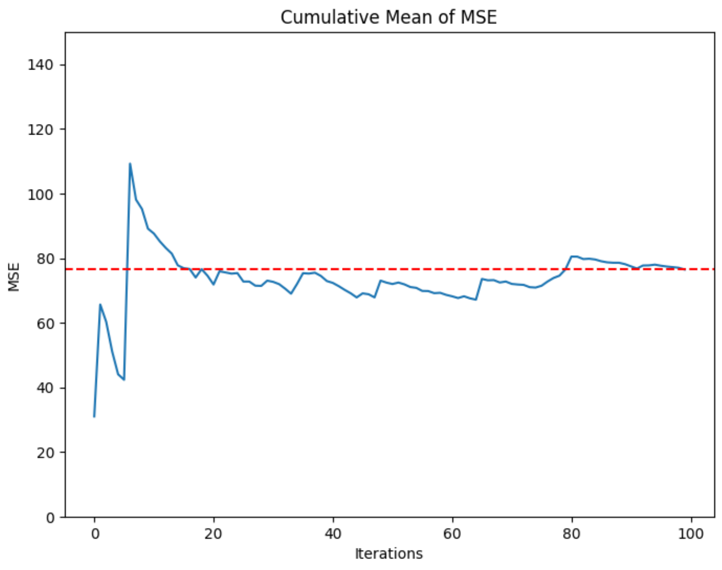


Figure VI-5. Cumulative Mean of Mean Square Error

(3) Cumulative RMSE

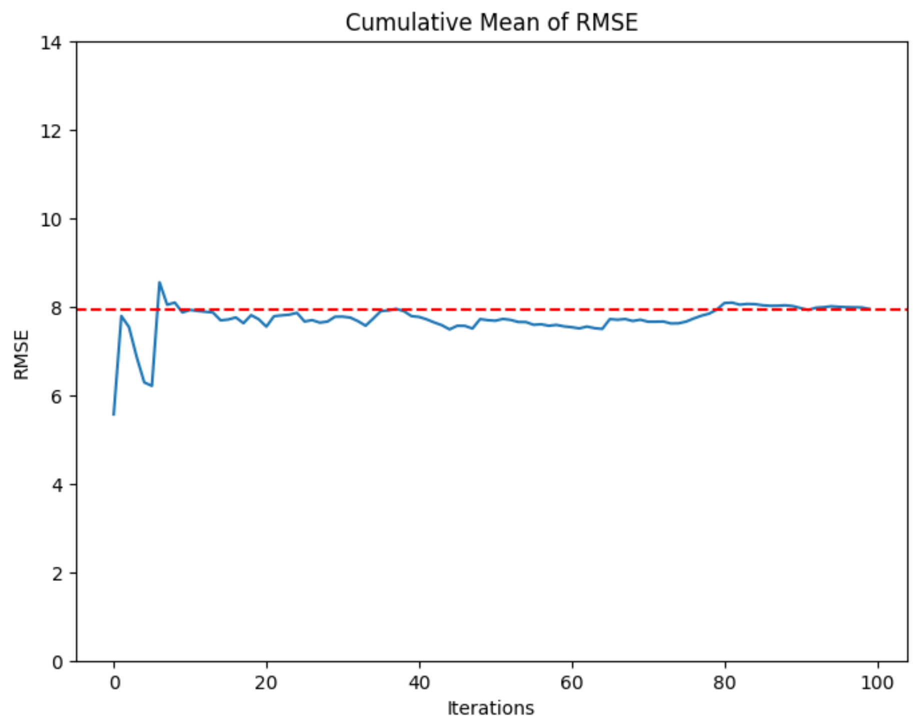


Figure VI-6. Cumulative Mean of Root Mean Square Error

(4) Cumulative R2

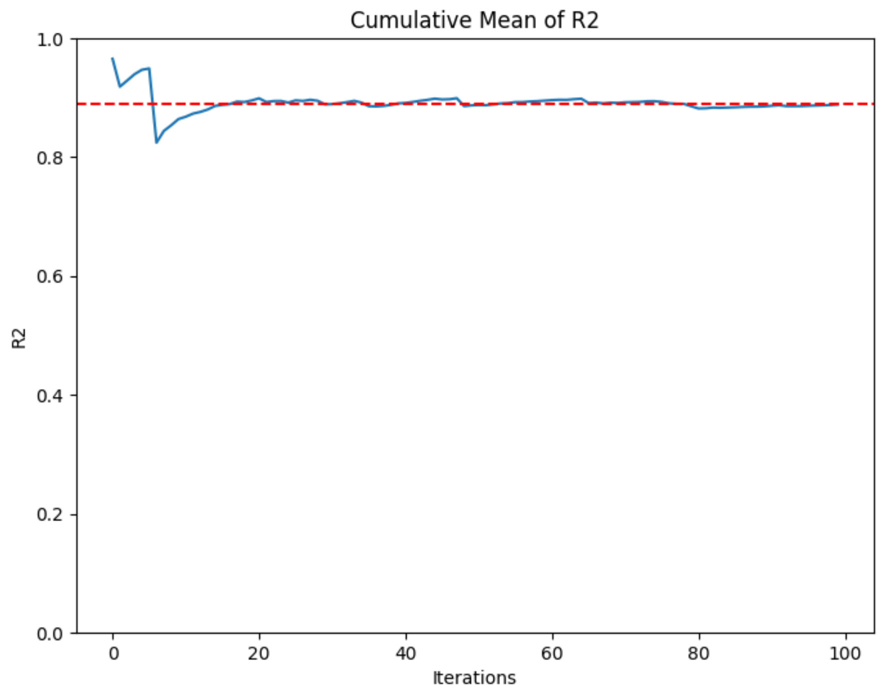


Figure VI-7. Cumulative Mean of R-Squared

(5) Cumulative Accuracy within 15% tolerance

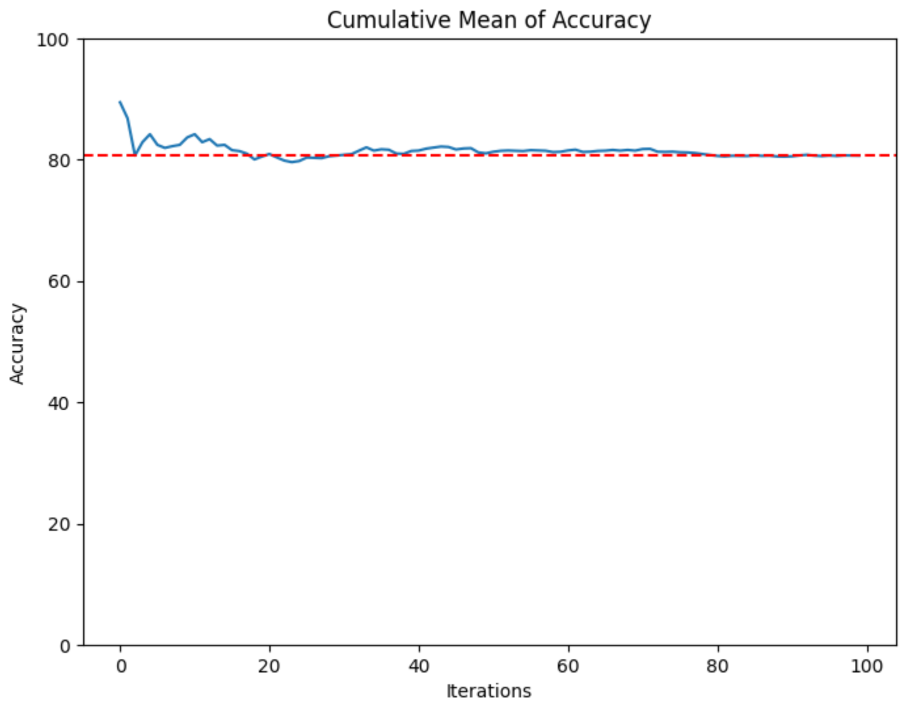


Figure VI-8. Cumulative Mean of Accuracy Within 15% Tolerance

To add to that, the nature of the data significantly impacts the performance of the neural network. An example of this can easily be demonstrated with applying the model to PCTRAN's data discussed in 4.1 The results of the model would be:

Table 7. Model Results Using PC-Tran's Data

Mean Absolute Error (MAE): 0.883
Mean Squared Error (MSE): 1.15
Root Mean Squared Error (RMSE): 1.072
R-squared: 0.997
Within tolerance: 100.0%

This is expected due to the simpler nature of the data. However, it can be noted by inspection that the GPWR data itself does not exhibit noticeable linear difference between a percentage and another. This is especially true for larger break sizes. An average MAE, for example, of

5.185 is reasonable with that in mind, despite it being higher in value than some of the literature discussed before.

Chapter VII
Conclusion and Future Work

The deep learning models discussed in this study demonstrate significant advantages in calculating break sizes almost instantaneously and providing more accurate insights into safety-related problems. As it was shown, the model, through its architecture and the chosen layers, can determine hidden relations between given parameters.

In this work, LSTM Architecture is used to model and predict the break sizes regarding a LOCA. Despite their typical use for time-series prediction, LSTM networks showcase strong results when applied to post-nuclear-accidents' parameters for regression. From experimental results, it was observed that LSTM neural network gives substantial results with high accuracy, with 5.18 MAE on average, and 89% R^2. Those results are comparable to other results of different models discussed in literature.

This work can be regarded as a foundation model for the prediction and assessment of break sizes. The potential for applying such models to actual reactor systems is promising, and the results can be scaled to real reactor accident data for specific reactors. RMSE and MAE errors can be further reduced by hyper tuning parameters, adding more datasets or a combination of both with fine accuracy. Future work may explore ways to expand the proposal here to furthermore reduce RMSE. This work can be fine-tuned for other accident-related problems by changing the dataset, such as; Loss of Flow Accidents (LOFA), given adequate datasets qualitatively and quantitatively.

Bibliography

1. Bays, S., Abou Jaoude, A., & Borlodan, G. (2019). *Reactor Fundamentals Handbook* (INL/EXT-19-53301-Rev000, 1615634; p. INL/EXT-19-53301-Rev000, 1615634). https://doi.org/10.2172/1615634

2. Choi, G. P., Yoo, K. H., Back, J. H., & Na, M. G. (2017). Estimation of LOCA Break Size Using Cascaded Fuzzy Neural Networks. *Nuclear Engineering and Technology, 49*(3), 495–503. https://doi.org/10.1016/j.net.2016.11.001

3. Chollet, F. (n.d.). *Keras: Deep Learning for humans*. Retrieved June 27, 2024, from https://keras.io/

4. *Ševera, Pavel. (1993). Reconstruction of: Emergency Core Cooling System (ECCS), Confinement Spray System (SSK), and Pressurizer Safety Valves and Relief Line System (PVKO). VUJE Trnava.*

5. Chollet, F. (2021). *Deep learning with Python* (Second edition). Manning.

6. Frepoli, C. (2008). An Overview of Westinghouse Realistic Large Break LOCA Evaluation Model. *Science and Technology of Nuclear Installations, 2008*. https://doi.org/10.1155/2008/498737

7. Giba, L. (n.d.). *Elastic Net Regression Explained, Step by Step*. Retrieved June 27, 2024, from https://machinelearningcompass.com/machine_learning_models/elastic_net_regression/

8. Glasstone, S., & Sesonske, A. (1994). *Nuclear reactor engineering: Reactor systems engineering. Fourth edition, Volume Two*. Chapman and Hall, New York, NY (United States). https://www.osti.gov/biblio/100943

9. Gupta, D. (2020, January 29). Fundamentals of Deep Learning—Activation Functions and When to Use Them? *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/

10. Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

11. Muhammad, W., Ullah, I., & Ashfaq, M. (2020). An Introduction to Deep Convolutional Neural Networks With Keras. In *Machine Learning and Deep Learning in Real-Time Applications* (pp. 231–272). IGI Global. https://doi.org/10.4018/978-1-7998-3095-5.ch011

12. NRC. (n.d.). Westinghouse Technology Systems Manual Section 5.2 Emergency Core Cooling Systems. In *Westinghouse Technology Systems Manual*. https://www.nrc.gov/docs/ML2116/ML21166A218.pdf

13. Pandas Development Team, T. (2024). *pandas-dev/pandas: Pandas* (v2.2.2) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.10957263

14. Qi, B., Xiao, X., Liang, J., Po, L. C., Zhang, L., & Tong, J. (2022). An open time-series simulated dataset covering various accidents for nuclear power plants. *Scientific Data*, *9*(1), 766. https://doi.org/10.1038/s41597-022-01879-1

15. Racheal, S., Liu, Y., & Ayodeji, A. (2022). Evaluation of optimized machine learning models for nuclear reactor accident prediction. *Progress in Nuclear Energy*, *149*, 104263. https://doi.org/10.1016/j.pnucene.2022.104263

16. Reddi, S. J., Kale, S., & Kumar, S. (2019). *On the Convergence of Adam and Beyond* (arXiv:1904.09237). arXiv. https://doi.org/10.48550/arXiv.1904.09237

17. Rossum, G. V. (n.d.). *io—Core tools for working with streams*. Python Documentation. Retrieved June 27, 2024, from https://docs.python.org/3/library/io.html

18. Saghafi, M., & Ghofrani, M. B. (2019). Real-time estimation of break sizes during LOCA in nuclear power plants using NARX neural network. *Nuclear Engineering and Technology*, *51*(3), 702–708. https://doi.org/10.1016/j.net.2018.11.017

19. She, J., Shi, T., Xue, S., Zhu, Y., Lu, S., Sun, P., & Cao, H. (2021). Diagnosis and Prediction for Loss of Coolant Accidents in Nuclear Power Plants Using Deep Learning Methods. *Frontiers in Energy Research*, *9*. https://doi.org/10.3389/fenrg.2021.665262

20. Shimeck, D. J., & Hartz, J. J. (2000). *Description of the Westinghouse LOCTAJR 1-D Heat Conduction Code for LOCA Analysis of Fuel Rods*.

21. Thurgood, P. M. J., Kelly, J. M., Guidotti, T. E., Kohrt, R. J., & Crowell, K. R. (n.d.). *COBRA/TRAC - A Thermal-Hydraulics Code for Transient Analysis of Nuclear Reactor Vessels and Primary Coolant Systems*.

22. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). *Attention Is All You Need* (arXiv:1706.03762). arXiv. https://doi.org/10.48550/arXiv.1706.03762

23. Westinghouse. (2009). *Overview Westinghouse Realistic (Best-Estimate) LOCA Methodology*.

24. Xiao, X., Qi, B., Liang, J., Tong, J., Deng, Q., & Chen, P. (2024). Enhancing LOCA Breach Size Diagnosis with Fundamental Deep Learning Models and Optimized Dataset Construction. *Energies*, *17*(1), Article 1. https://doi.org/10.3390/en17010159

25. Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2023). *Dive into Deep Learning* (arXiv:2106.11342). arXiv. https://doi.org/10.48550/arXiv.2106.11342

26. NEA. (2009). *Nuclear Fuel Behavior in Loss-of-coolant Accident (LOCA) Conditions*. OECD.

Appendix A: Essential Code

Simulator Variables change to Human-Readable Variables

```
#!/usr/bin/env python
# coding: utf-8

import pandas as pd
import os
import chardet

# Initial condition
xls = pd.ExcelFile('/content/PARAMETERS (1).xlsx') # Put the edited lab file here to translate
Huge_folder = '/content/sample_data/raw' # Put the folder containing all breaks
recieve_folder = '/content/sample_data/fried' # Put the folder containing all breaks after edit

# Prepare the great folder
list_of_files = os.listdir(Huge_folder)

# Mapping dictionary
mapping = {}
Mapping = {}

for sheet_name in xls.sheet_names:
    df = pd.read_excel(xls, sheet_name=sheet_name)

    key_column = df.columns[1]
    value_column = df.columns[2]

    for _, row in df.iterrows():
        key = row[key_column]
        value = row[value_column]
        mapping[key] = value

for key, value in mapping.items():
    new_key = str(key).replace('-', '')
    new_key = str(new_key).replace(' ', '')
    new_key = str(new_key).replace('*', '')
    Mapping[new_key] = value

list_of_files
```

```

# Duplicate the edit with new folder
for b in range(len(list_of_files)):
    # Open the file
    path = f'{Huge_folder}/{list_of_files[b]}'
    with open(path, 'rb') as f:
        encoding = chardet.detect(f.read())['encoding']

    df = pd.read_csv(path, skiprows=[0, 1], encoding='ISO-8859-1')

    # Shift names to the right places
    df = df.drop(df.index[0:4])
    df = reset_index(drop=True)

    # Define the real names
    df = df.rename(columns={'Variable': 'time'})

    # Delete the extra strings
    first = []
    sec = []
    for i in range(len(df.columns)):
        first.append(df.columns[i].strip('hmi_'))
    for j in range(len(first)):
        sec.append(first[j].strip('_VALUE'))
    df.columns = sec
    df.rename(columns=Mapping, inplace=True)

    # Extract
    x = list_of_files[b].strip(".csv")
    y = x.strip('.xslx')
    df.to_excel(f'{recieve_folder}/ {y}.xlsx', index=False)

```

Transformation of features to a single file

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


# Important Libraries
import pandas as pd
import os
import chardet
from google.colab import files
import glob



# In[ ]:

```

```
#Initial Condition
xls = pd.ExcelFile('/content/PARAMETERS (1).xlsx')    #put the edited lab file here to translate
Huge_folder='/content/sample_data/raw trial' #put the folder contain all breaks
recieve_folder='/content/sample_data/rare'  #put the folder contain all breaks after edit
paramater_folder='/content/sample_data/fried 3' #put the final folder

list_of_files= os.listdir(Huge_folder)
list_of_sheets_lame= os.listdir(recieve_folder)
list_of_sheets_lame.sort(reverse=False)



# In[ ]:


#Mapping Dictionary

mapping = {}
Mapping={}

for sheet_name in xls.sheet_names:
    df = pd.read_excel(xls, sheet_name=sheet_name)

    key_column = df.columns[1]
    value_column = df.columns[2]

    for _, row in df.iterrows():
        key = row[key_column]
        value = row[value_column]
        mapping[key] = value

for key,value in mapping.items():
  new_key=str(key).replace('-','')
  new_key=str(new_key).replace(' ','')
  new_key=str(new_key).replace('*','')
  Mapping[new_key]=value


# In[ ]:


#duplicate the edit with new folder

for b in range(len(list_of_files)):

    #open the file

  path=f'{Huge_folder}/{list_of_files[b]}'
  with open(path, 'rb') as f:
    encoding = chardet.detect(f.read())['encoding']

  df=pd.read_csv(path,skiprows=[0,1],encoding='ISO-8859-1')


    #shift names to the right places
  df = df.drop(df.index[0:4])
  df = df.reset_index(drop=True)

    #define the real names
  df = df.rename(columns={'Variable': 'time'})

    #delete the extra strings
  first=[]
  sec=[]
  for i in range(len(df.columns)): first.append(df.columns[i].strip('hmi_'))
  for j in range(len(first)): sec.append(first[j].strip('_VALUE'))
  df.columns=sec
  df.rename(columns=Mapping,inplace=True)
```

```
    #extract
  x=list_of_files[b].strip(".csv")
  y=x.strip('.xslx')
  df.to_excel(f'{recieve_folder}/ {y}.xlsx', index=False)



# now, let's create a code to select each parameter in one sheet (with diff breaks)

# In[ ]:


list_of_sheets=[]
list_of_sheets_lame= os.listdir(recieve_folder)
for name in list_of_sheets_lame:
  new_name=name.strip('.xlsx')
  new_name=name.strip('processed')
  list_of_sheets.append(new_name)

list_of_sheets.sort(reverse=False)
haha=pd.read_excel(f'{recieve_folder}/{list_of_sheets_lame[1]}')
parameters=list(haha.columns)
parameters.remove('time')

for p in parameters:
  df=pd.DataFrame()
  df.to_excel(f'{paramater_folder}/{p}.xlsx', index=False)

list_of_out= os.listdir(paramater_folder)
list_of_sheets_lame.sort(reverse=False)
output_file = paramater_folder
input_file = recieve_folder


# In[ ]:


for v in range(len(list_of_sheets_lame)):
  df=pd.read_excel(f'{recieve_folder}/{list_of_sheets_lame[v]}')
  if df.columns[0] !='time': print(list_of_sheets_lame[v])


# In[ ]:


paramater_folder='/content/sample_data/fried 4'
input_file='/content/sample_data/rare'
for a in range(len(list_of_out)): #num= 75
  df_req=pd.read_excel(f'{output_file}/{list_of_out[a]}') #pressure for example
  for num in range(len(list_of_sheets_lame)): #num= 5
    df=pd.read_excel(f'{input_file}/{list_of_sheets_lame[num]}') # break 3% for example
    old_name = df.columns[a+1]
    new_name = f'{old_name} {list_of_sheets[num]}'
    selected_column = df[old_name].rename(new_name)
    df_req[new_name] = selected_column
    df_req.to_excel(f'{paramater_folder}/{old_name}.xlsx', index=False)
```

Feature Reduction Algorithm

```
1. import os
2. import pandas as pd
3.
```

```
# List of relevant variables
relevant_variables = [
    "TSCWT1",
    "RCSCT5",
    "MRSPT514", "MRSPT524", "MRSPT534", "MRSPT544",
    "RCSLT501", "RCSLT502",
    "RCSFT414", "RCSFT424",
    "RCSLT470",
    "RCSPT405",
    "RCSTT413A", "RCSTT413B",
    "RCSTT423B", "RCSTT423C",
    "RCSTT433B",
    "RCSTT443B",
    "RCSTT453A",
    "RCSTT1734",
    "CTMTT60",
    "CTMPT1000A",
    "RRCTP10"
]

# Directory containing CSV files
input_directory = "input_files"

# Output directory for new CSV files
output_directory = "output_files"

# Ensure output directory exists
if not os.path.exists(output_directory):
    os.makedirs(output_directory)

# Process each CSV file
for filename in os.listdir(input_directory):
    if filename.endswith(".csv"):
        input_filepath = os.path.join(input_directory, filename)
        output_filepath = os.path.join(output_directory, filename)

        # Read CSV file
        df = pd.read_csv(input_filepath)

        # Find columns with relevant headers (containing substrings)
        relevant_columns = [col for col in df.columns if any(var in col for var in relevant_variables)]

        # Save new CSV file with only relevant columns
        df[relevant_columns].to_csv(output_filepath, index=False)

```

Main Model Algorithm

• Data Loading Algorithm

```
import tensorflow as tf
import os
import pandas as pd
import numpy as np

#If you dont have the latest keras version, dont run this
os.environ["KERAS_BACKEND"] = "tensorflow"
#Loading a tensor with the initial three columns in each csv files, where each tensor corresponds to an input
path = './gpdatafinalproc/'
sequences1 = list()
max_row = 300  # Change this to the desired maximum row number

for i in np.arange(0.5, 100.5, 0.5):
    if i % 1 != 0:   #Ensure that i is a float not an integer
        i = i
        try:
            file_path = path + str(i) + '%_processed.csv'
            print(file_path)
            # Read only up to the specified maximum row
            df = pd.read_csv(file_path, header=0, nrows=max_row, usecols= [0, 1, 2, 6, 8, 11, 13, 18, 19, 22])
            # Drop column if it exists
            if 'hmi_RCSLT501_VALUE_BAR' in df.columns:
                df.drop(columns=['hmi_RCSLT501_VALUE_BAR'], inplace=True)
            values = df.values
            sequences1.append(values)
        except FileNotFoundError:
            try:
                file_path = path + str(i) + '_processed.csv'
                print(file_path)
                # Read only up to the specified maximum row
                df = pd.read_csv(file_path, header=0, nrows=max_row, usecols = [0, 1, 2, 6, 8, 11, 13, 18, 19, 22])
                # Drop column if it exists
                if 'hmi_RCSLT501_VALUE_BAR' in df.columns:
                    df.drop(columns=['hmi_RCSLT501_VALUE_BAR'], inplace=True)
                values = df.values
                sequences1.append(values)
            except FileNotFoundError:
                print("File Not Found")
    else:
        i = int(i)
        try:
            file_path = path + str(i) + '.0%_processed.csv'
            print(file_path)
            # Read only up to the specified maximum row
            df = pd.read_csv(file_path, header=0, nrows=max_row, usecols=[0, 1, 2, 6, 8, 11, 13, 18, 19, 22])
            # Drop column if it exists
            if 'hmi_RCSLT501_VALUE_BAR' in df.columns:
                df.drop(columns=['hmi_RCSLT501_VALUE_BAR'], inplace=True)
            values = df.values
            sequences1.append(values)
        except FileNotFoundError:
            try:
                file_path = path + str(i) + '%_processed.csv'
                print(file_path)
                # Read only up to the specified maximum row
                df = pd.read_csv(file_path, header=0, nrows=max_row, usecols = [0, 1, 2, 6, 8, 11, 13, 18, 19, 22])
                # Drop column if it exists
                if 'hmi_RCSLT501_VALUE_BAR' in df.columns:
                    df.drop(columns=['hmi_RCSLT501_VALUE_BAR'], inplace=True)
                values = df.values
                sequences1.append(values)
            except FileNotFoundError:
                print("File Not Found")

y_data = [i for i in range(1, 101) if i not in {17, 19, 22, 23}]
y_data += [i + 0.5 for i in range(0, 13)] + [i + 0.5 for i in range(20, 100) if i + 0.5 not in {21.5, 22.5, 27.5}]

y_data.sort()

sequences1= np.array(sequences1)
y_data = np.array(y_data)

num_samples = len(sequences1)
indices = np.arange(num_samples)
np.random.shuffle(indices)

shuffled_data = sequences1[indices]
shuffled_targets = y_data[indices]

# Determine the sizes of each set
train_size = int(0.7 * num_samples)
val_size = int(0.2 * num_samples)
test_size = num_samples - train_size - val_size

# Split the shuffled data and targets
X_train, y_train = shuffled_data[:train_size], shuffled_targets[:train_size]
X_val, y_val = shuffled_data[train_size:train_size + val_size], shuffled_targets[train_size:train_size + val_size]
X_test, y_test = shuffled_data[train_size + val_size:], shuffled_targets[train_size + val_size:]

# Convert data to float arrays
```

```
X_train_float = np.array(X_train, dtype=float)
y_train_float = np.array(y_train, dtype=float)
X_val_float = np.array(X_val, dtype=float)
y_val_float = np.array(y_val, dtype=float)
X_test_float = np.array(X_test, dtype=float)
y_test_float = np.array(y_test, dtype=float)

```

- ## Data Pre-processing Algorithm

```
# Calculate mean and standard deviation
means = np.mean(X_train_float, axis=(0, 1))  # Calculate mean across samples and time steps
stds = np.std(X_train_float, axis=(0, 1))    # Calculate std across samples and time steps
mean_y_train = np.mean(y_train_float)
std_y_train = np.std(y_train_float)
def preprocess(X, means, stds):
    for i in range(X.shape[2]):  # Loop over each feature
        X[:, :, i] = (X[:, :, i] - means[i]) / stds[i]
    return X

def preprocess_output3(y, mean, std):
  y = (y- mean)/std
  return y
# Preprocess all datasets using the same normalization parameters
X_train = preprocess(X_train, means, stds)
X_val = preprocess(X_val, means, stds)
X_test = preprocess(X_test, means, stds)


y_train = preprocess_output3(y_train, mean_y_train, std_y_train)
y_val = preprocess_output3(y_val, mean_y_train, std_y_train)
y_test = preprocess_output3(y_test, mean_y_train, std_y_train)

#This is only because y-data are read as lists within list when they're not
def classes(arr):
    normalized_data = [[x] for x in arr]
    return normalized_data

y_train = np.array(classes(y_train))
y_val = np.array(classes(y_val))
y_test = np.array(classes(y_test))

```

- ## Model Creation and Training

```
#If you dont have the latest keras version, its going to be from tf.keras.models import Sequential
```

```
from keras.models import Sequential
from keras.layers import *
from keras.callbacks import ModelCheckpoint, ReduceLROnPlateau
from keras.losses import MeanAbsoluteError, MeanSquaredLogarithmicError
from keras.metrics import RootMeanSquaredError, MeanSquaredLogarithmicError
from keras.optimizers import Adam
from keras.losses import Huber
from keras.regularizers import *

# Define the regularization strength
l1_reg_strength = 0.013
l2_reg_strength = 0.013

# Create the model
model = Sequential()
model.add(LSTM(units=18, return_sequences=False, input_shape=(300, 10),
               kernel_regularizer=l1_l2(l1=l1_reg_strength, l2=l2_reg_strength)))

model.add(Dense(8, activation='linear'))

model.add(Dense(units=1, activation='linear',
                kernel_regularizer=l1_l2(l1=l1_reg_strength, l2=l2_reg_strength)))



# Display model summary
model.summary()

import keras
@keras.saving.register_keras_serializable()
def custom_loss(y_true, y_pred):
    penalty_factor_1 = 1.1  # Penalty for values between 0 and 10
    penalty_factor_2 = 1.6  # Penalty for values between 70 and 85
    penalty_factor_3 = 9  # Penalty for values between 85 and 100

    mse = tf.square(y_true - y_pred)

    # Apply penalties based on the range of y_true values
    penalized_mse = tf.where((y_true > 0) & (y_true <= 10), penalty_factor_1 * mse, mse)
    penalized_mse = tf.where((y_true > 70) & (y_true <= 80), penalty_factor_2 * mse, mse)
    penalized_mse = tf.where((y_true > 80) & (y_true <= 100), penalty_factor_3 * mse, mse)

    weighted_error = tf.reduce_mean(penalized_mse)

    return tf.sqrt(weighted_error)

cp5 = ModelCheckpoint('model/best_model.keras', save_best_only=True)
```

```
model.compile(loss=custom_loss, optimizer=Adam(learning_rate=0.001), metrics=[RootMeanSquaredError()])

history = model.fit(X_train, y_train, batch_size = 32, validation_data=(X_val, y_val), epochs=2000, callbacks=[cp5])
```

- ## Post-Processing and Results

```
def postprocess_y(arr):
    arr = (arr * std_y_train) + mean_y_train
    return arr
model = keras.models.load_model('model/best_model.keras')
predictions = model.predict(X_test)
predictions = postprocess_y(predictions[:])
actual = postprocess_y(y_test[:, 0])

pd.options.display.float_format = '{:.2f}'.format

train_results = pd.DataFrame(data={'Test Predictions':predictions.flatten(), 'Actuals':actual})
train_results

from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score



# Assuming `actual.flatten()` and `predictions.flatten()` are your test set targets and model predictions as numpy arrays:
actual = actual.flatten()
predictions = predictions.flatten()

# Calculate MAE
mae = mean_absolute_error(actual, predictions)
print(f'Mean Absolute Error (MAE): {mae}')

# Calculate MSE
mse = mean_squared_error(actual, predictions)
print(f'Mean Squared Error (MSE): {mse}')

# Calculate RMSE
rmse = np.sqrt(mse)
print(f'Root Mean Squared Error (RMSE): {rmse}')

# Calculate R-squared
r2 = r2_score(actual, predictions)
print(f'R-squared: {r2}')

def percentage_within_tolerance(actual, predictions, tolerance):
    """
    Calculates the percentage of predictions that are within a certain
    tolerance of the actual values.
    Parameters:
        actual (numpy.ndarray): The actual values.
        predictions (numpy.ndarray): The predicted values.
        tolerance (float): The acceptable tolerance as a decimal (e.g., 0.10 for ±10%).

    Returns:
        float: The percentage of predictions within the tolerance.
    """
    actual = np.array(actual)
    predictions = np.array(predictions)

    # Calculate the absolute difference in terms of the actual values
    difference = np.abs(actual - predictions)
    acceptable_difference = tolerance * actual

    # Find the percentage of predictions within the acceptable difference
    accurate_predictions = np.where(difference <= acceptable_difference, 1, 0)
    accuracy_percentage = np.mean(accurate_predictions) * 100

    return accuracy_percentage



accuracy = percentage_within_tolerance(actual, predictions.flatten(), tolerance=0.15)
print(f"Percentage of predictions within tolerance: {accuracy}%")
```

- ## Plotting Training vs Validation loss

```
import matplotlib.pyplot as plt

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')

# Find the epoch with the lowest validation loss
best_epoch = history.history['val_loss'].index(min(history.history['val_loss']))

# Mark the point of the best model on the graph
best_val_loss = history.history['val_loss'][best_epoch]
plt.scatter(best_epoch, best_val_loss, color='red', marker='o', label=f'Best Model (Epoch {best_epoch+1})')
plt.annotate(f'({best_epoch+1}, {best_val_loss:.2f})', (best_epoch, best_val_loss), textcoords="offset points", xytext=(-10,-10), 
ha='center')
```

```
plt.legend()

plt.show()
```

- ## Average Metrics

```
import numpy as np
import matplotlib.pyplot as plt
import ast

with open('results.txt', 'r') as file:
    arr = file.readlines()



# Step 3: Convert the string representation of the list to an actual list
arr = ast.literal_eval(arr[0])



print(np.mean(arr, axis = 0))



metrics = np.array(arr)
cumulative_means = np.cumsum(metrics, axis=0) / np.arange(1, metrics.shape[0] + 1)[:, None]

# Extract the cumulative means for each metric
cumulative_MAE = cumulative_means[:, 0]
cumulative_MSE = cumulative_means[:, 1]
cumulative_RMSE = cumulative_means[:, 2]
cumulative_R2 = cumulative_means[:, 3]
cumulative_Accuracy = cumulative_means[:, 4]




plt.figure(figsize=(8, 6))
plt.plot(cumulative_MAE)
plt.axhline(y=5.185992, color='r', linestyle='--')
plt.title('Cumulative Mean of MAE')
plt.xlabel('Iterations')
plt.ylabel('MAE')
plt.ylim(0, 10)
plt.savefig('cumulative_MAE.png')
plt.show()

# Plot cumulative mean of MSE
plt.figure(figsize=(8, 6))
plt.plot(cumulative_MSE)
plt.axhline(y=76.50258, color='r', linestyle='--')
plt.title('Cumulative Mean of MSE')
plt.xlabel('Iterations')
plt.ylabel('MSE')
plt.ylim(0, 150)
plt.savefig('cumulative_MSE.png')
plt.show()

# Plot cumulative mean of RMSE
plt.figure(figsize=(8, 6))
plt.plot(cumulative_RMSE)
plt.axhline(y=7.953842, color='r', linestyle='--')
plt.title('Cumulative Mean of RMSE')
plt.xlabel('Iterations')
plt.ylabel('RMSE')
plt.ylim(0, 14)
plt.savefig('cumulative_RMSE.png')
plt.show()

# Plot cumulative mean of R2
plt.figure(figsize=(8, 6))
plt.plot(cumulative_R2)
plt.axhline(y=0.888586, color='r', linestyle='--')
plt.title('Cumulative Mean of R2')
plt.xlabel('Iterations')
plt.ylabel('R2')
plt.ylim(0, 1)
plt.savefig('cumulative_R2.png')
plt.show()

# Plot cumulative mean of Accuracy
plt.figure(figsize=(8, 6))
plt.plot(cumulative_Accuracy)
plt.axhline(y=80.684211, color='r', linestyle='--')
plt.title('Cumulative Mean of Accuracy')
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.ylim(0, 100)
plt.savefig('cumulative_Accuracy.png')
plt.show()
```

Appendix B: Non-Essential Code

Reading CSV files from scratch

```
def parse_csv(file_path):
    # Step 1: Open the CSV file
    with open(file_path, mode='r') as file:
        data = []

        # Step 2: Read the file line by line
        for line in file:
            # Step 3: Split lines into fields
            # Remove any trailing newline characters and split by comma
            fields = line.strip().split(',')

            # Step 4: Store the data
            data.append(fields)

    # Step 5: Return the parsed data
    return data

# Example usage
file_path = 'example.csv'
parsed_data = parse_csv(file_path)
for row in parsed_data:
    print(row)

```

Reading Excel file from scratch

```
import zipfile
from xml.etree import ElementTree as ET

def parse_excel(file_path):
    # Step 1: Open the Excel file
    with open(file_path, 'rb') as file:
        # Step 2: Unzip the file
        with zipfile.ZipFile(file) as z:
            # Step 3: Read the workbook.xml file to get the sheet information
        workbook_xml = z.read('xl/workbook.xml')
        workbook_tree = ET.fromstring(workbook_xml)

        # Step 4: Read the first sheet (assuming it's sheet1.xml)
        sheet_path = 'xl/worksheets/sheet1.xml'
        sheet_xml = z.read(sheet_path)
        sheet_tree = ET.fromstring(sheet_xml)

        # Step 5: Parse the XML content to extract data
        data = []
        for row in sheet_tree.findall('.//{http://schemas.openxmlformats.org/spreadsheetml/2006/main}row'):
            row_data = []
            for cell in row.findall('{http://schemas.openxmlformats.org/spreadsheetml/2006/main}c'):
                cell_value = cell.find('{http://schemas.openxmlformats.org/spreadsheetml/2006/main}v')
                if cell_value is not None:
                    row_data.append(cell_value.text)
                else:
                    row_data.append('')
            data.append(row_data)

    # Step 6: Return the parsed data
    return data

# Example usage
file_path = 'example.xlsx'
parsed_data = parse_excel(file_path)
for row in parsed_data:
    print(row)

```

WSC Simulator Automation Code

```
import subprocess
import os
import pyautogui
from time import sleep
import psutil
import tkinter as tk
from tkinter import filedialog
from decimal import Decimal, ROUND_HALF_UP

def open_application(application_path):
    try:
        sleep(1)
        #pyautogui.click(x=13, y=1061)  # Click somewhere on the screen to ensure the focus is not on the Python shell
        sleep(0.5)
        pyautogui.hotkey('win', 'r')  # Press the Windows key + R to open the Run dialog
        pyautogui.hotkey('win', 'r')  # Press the Windows key + R to open the Run dialog
        sleep(1)
        pyautogui.write(application_path)  # Type the path to the application
        pyautogui.press('enter')  # Press Enter to open the application
        # Wait for the application window to open
        pyautogui.sleep(2)
        # Maximize the application window to full screen
        pyautogui.hotkey('win', 'up')
    except Exception as e:
        print(f"Error opening application: {e}")

print('''
    This program has to be in the same folder as SimClient.exe in the 3keymaster folder

    ''')

answer = True

while answer:
    break_size = float(input('Insert Initial Break Size: '))
    increment = float(input('Increase in break size each run: '))
    sim_time = float(input('a bit more than Half the Simulation time (seconds): '))
    sim_time /= 1.25

    print("""
        Choose directory to save files
        """)
    root = tk.Tk()
    root.withdraw()  # Hide the main window

    directory = filedialog.askdirectory()
    print("Chosen Directory:", directory)

    if break_size <= 100 and increment <= 100:
        answer = False
    else:
        print("Please insert a float between 0 and 100")
# Define the list of available servers
servers = ["SimServerV.exe", "SimServerD.exe", "SimServerW.exe"]

# Prompt the user to choose a server
print("Choose a server:")
for index, server in enumerate(servers):
    print(f"{index + 1}. {server}")

# Get user input for server selection
```

```
choice = input("Enter the number corresponding to the server you want to use: ")

try:
    choice = int(choice)
    if 1 <= choice <= len(servers):
        selected_server = servers[choice - 1]
        print(f"Selected server: {selected_server}")
    else:
        print("Invalid choice. Please enter a valid number.")
except ValueError:
    print("Invalid input. Please enter a number.")


precision = Decimal('0.1')

while break_size <= 100:
    sleep(5)
    # Get the current directory
    break_size = Decimal(f'{break_size}').quantize(precision, rounding=ROUND_HALF_UP)
    break_size = float(break_size)
    file_name = rf'{directory}\{break_size}%.csv'
    current_directory = os.getcwd()
    program_name = selected_server
    # Construct the full path to the program
    program_path = os.path.join(current_directory, program_name)
    # Open the program using subprocess
    #process = subprocess.Popen(program_path)
    open_application(program_path)
    sleep(5)
    pyautogui.write(rf'reset -o "Books\GenericPWR\IC\2"') #Reset for good measure
    pyautogui.press('enter')
    sleep(2)
    pyautogui.write(f'set System.Slow = 0.5;')
    pyautogui.press('enter')
    sleep(1)
    pyautogui.write(f'IMF mf_RCS05 f:{break_size}')
    pyautogui.press('enter')
    sleep(0.4)
    pyautogui.write(f'run')
    pyautogui.press('enter')
    sleep(sim_time) #simulation run-time
    pyautogui.write(f'Freeze') #Freeze simulation
    pyautogui.press('enter')
    sleep(1)
    pyautogui.write(rf'daq 0 301 1.0 "{file_name}" {break_size}%')
    pyautogui.press('enter')
    sleep(2)
```

```
pyautogui.write(f'exit')
pyautogui.press('enter')
break_size += increment
```

Appendix C: Trends of Parameters Used in Model

Boron Concentration (0[1])

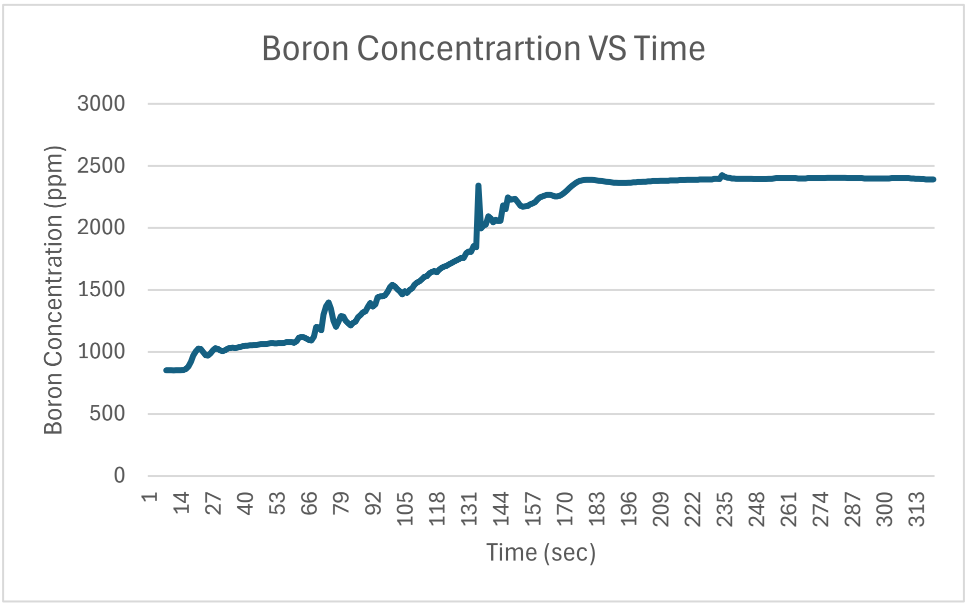


Figure C-1. Boron Concentration (At 10% Break)

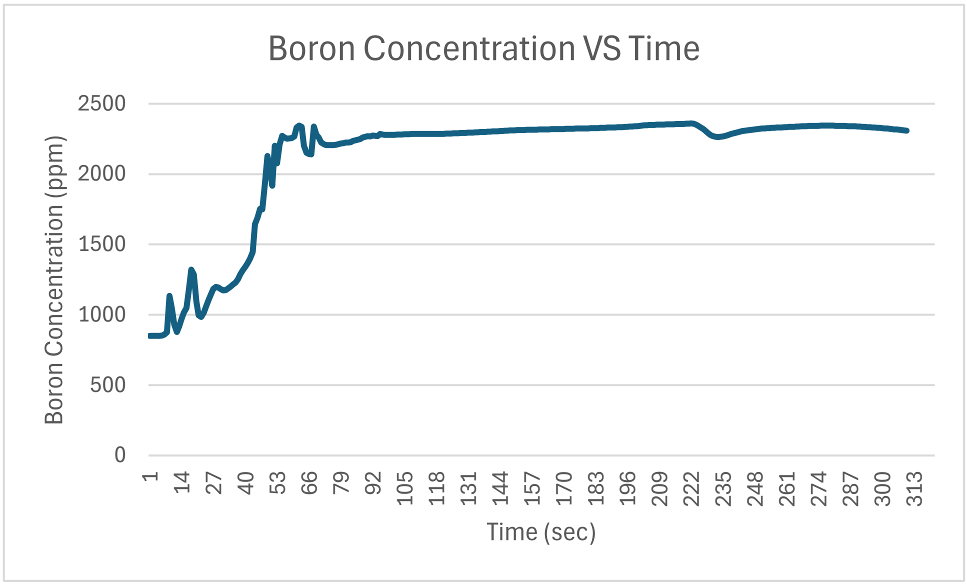


Figure C-2. Boron Concentration (At 100% Break)

[1] The number of column used in the python code implementation

Wide Ring Channel 1 Pressure (11)

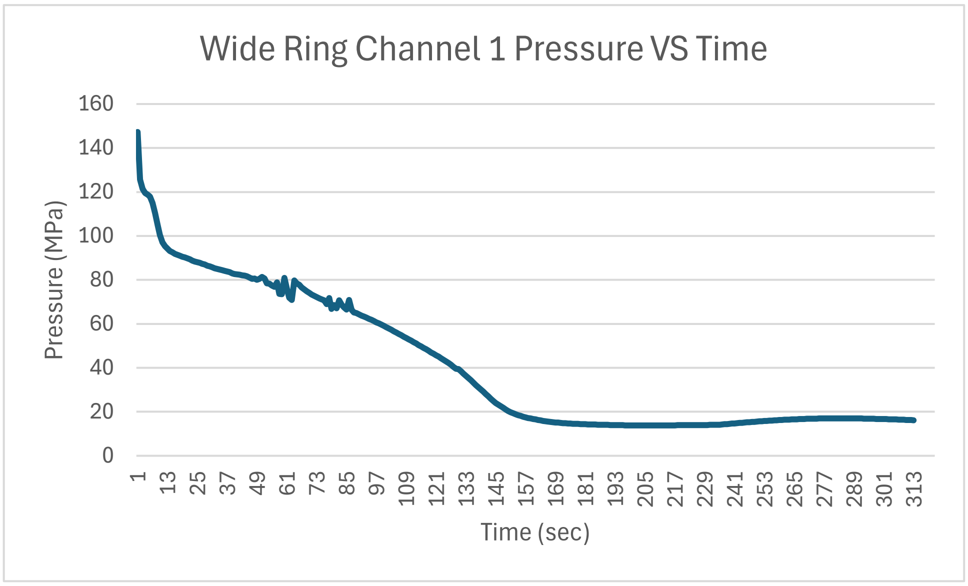


Figure C-3. Wide Ring Channel 1 Pressure (At 10% Break)

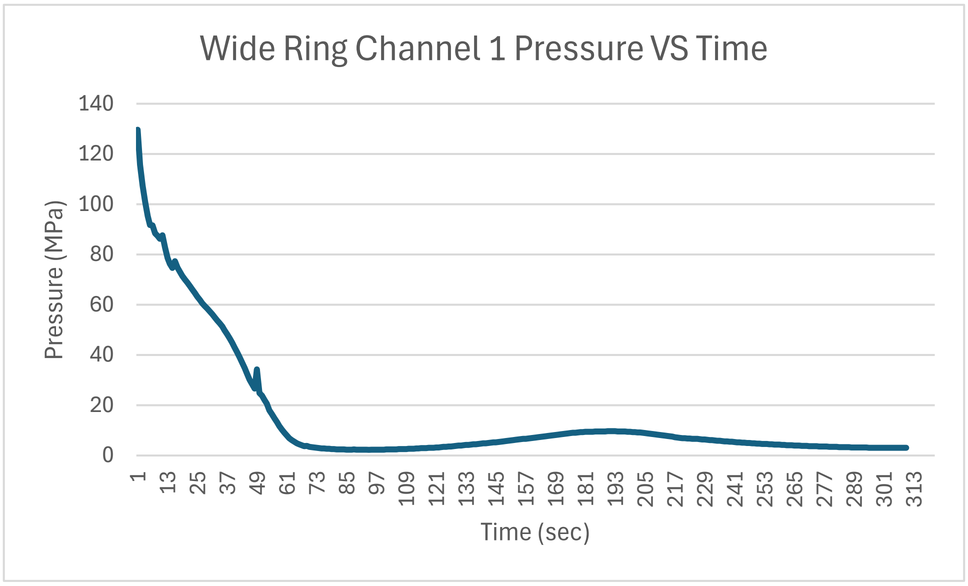


Figure C-4. Wide Ring Channel 1 Pressure (At 100% Break)

Thermal Power (1)

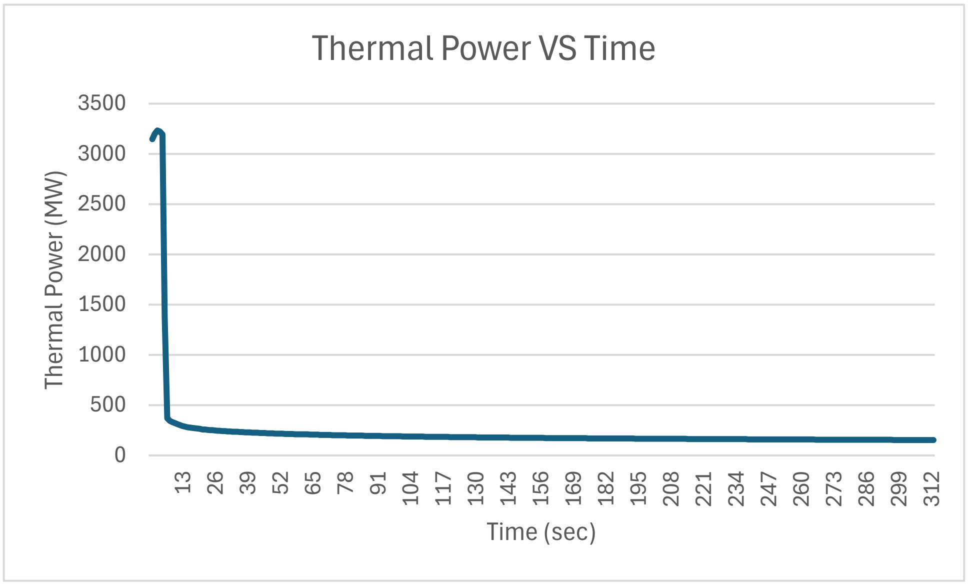


Figure C-6. Thermal Power (At 10% Break)

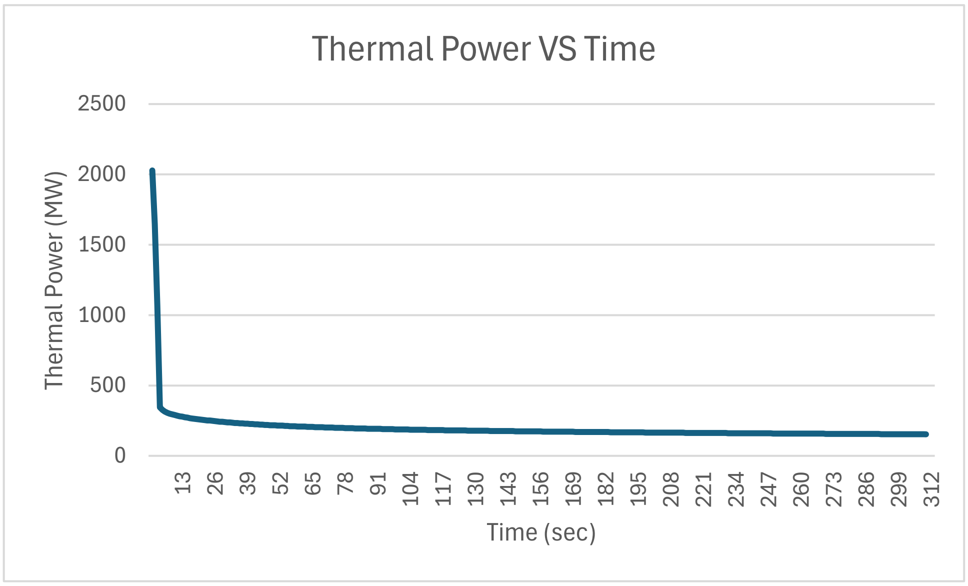


Figure C-5. Thermal Power (At 100% Break)

Pressure of SG1 Loop 1A (2)

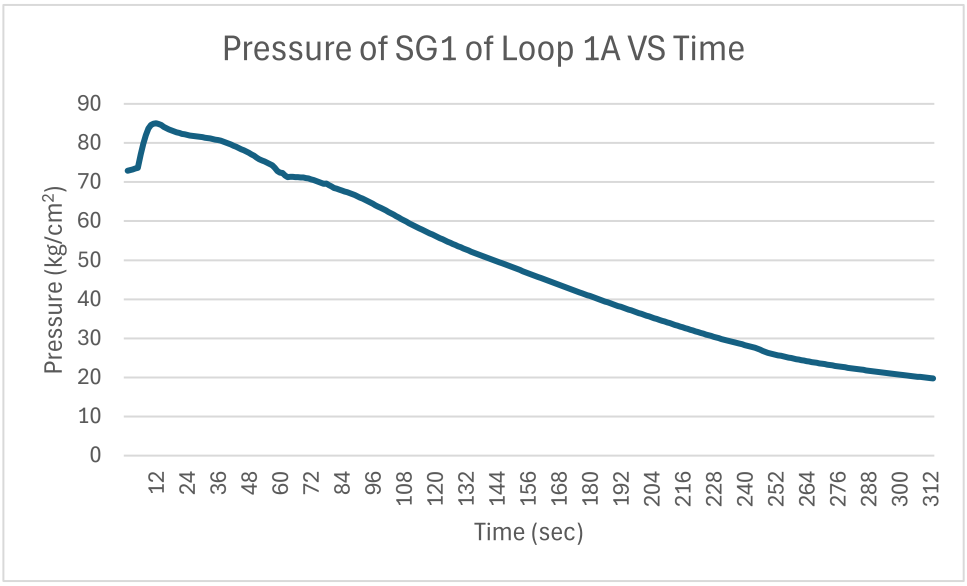


Figure C-7. Pressure of SG1 Loop 1A (At 10% Break)

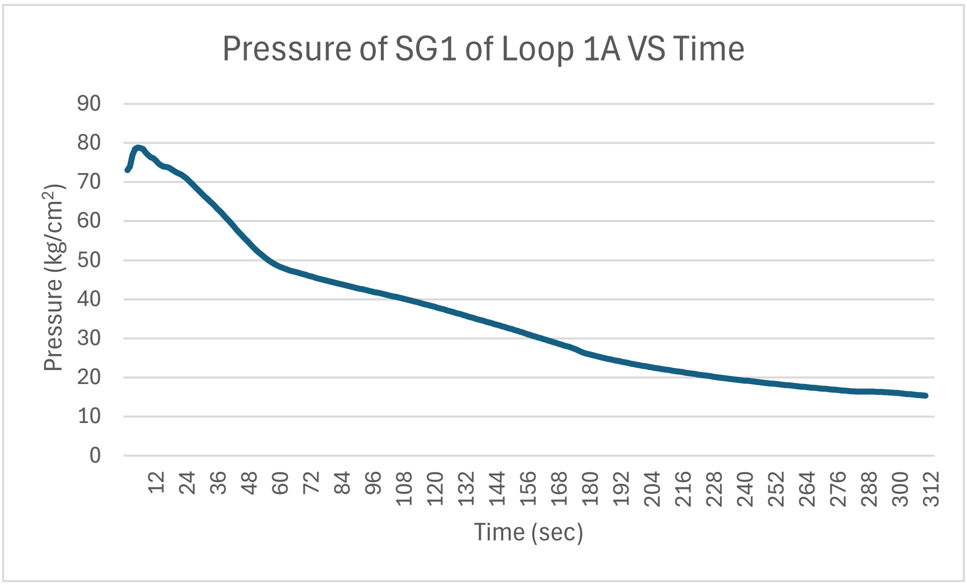


Figure C-8. Pressure of SG1 Loop 1A (At 100% Break)

SG Wide Ring Level (6)

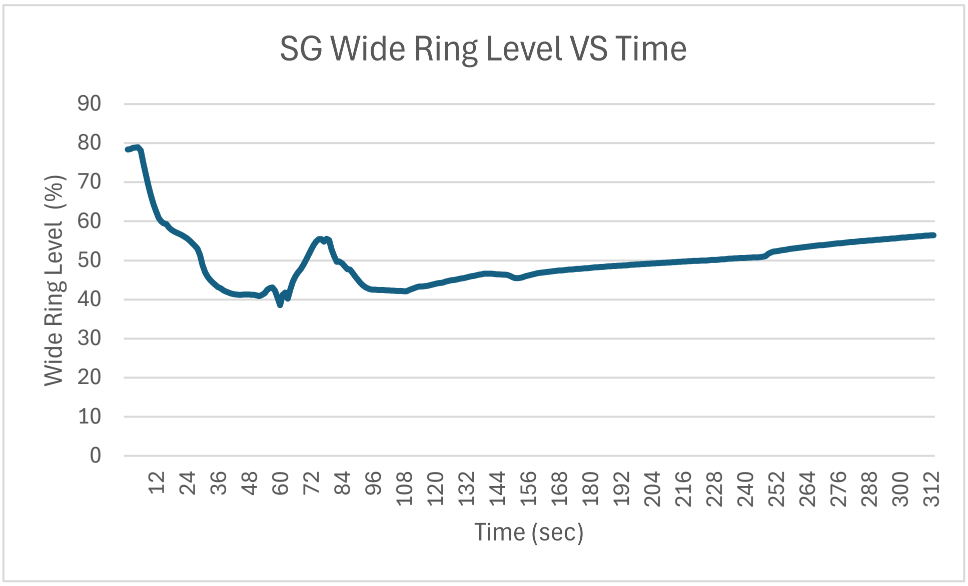


Figure C-9: SG Wide Ring Level (At 10% Break)

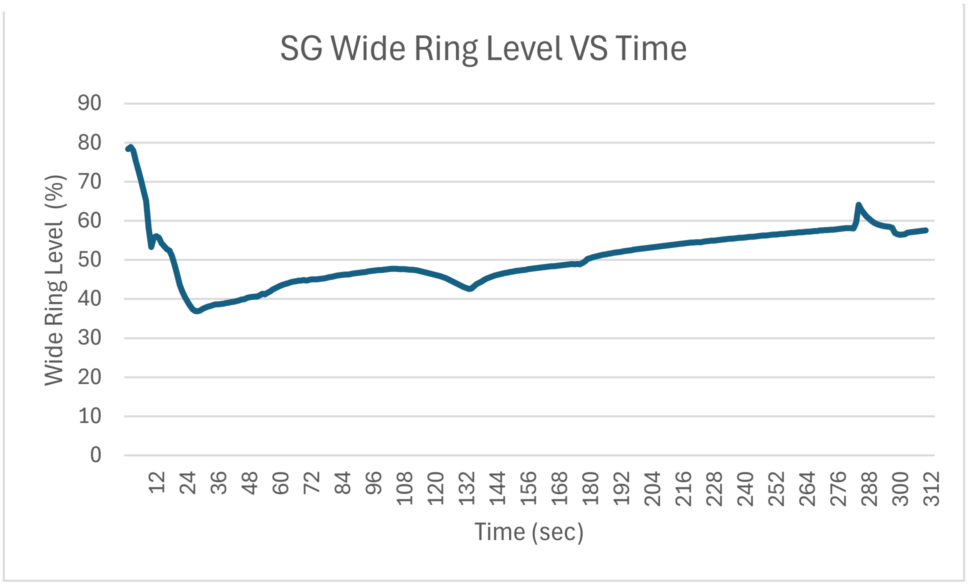


Figure C-10. SG Wide Ring Level (At 100% Break)

Flow of Cold Leg of Loop 1A (8)

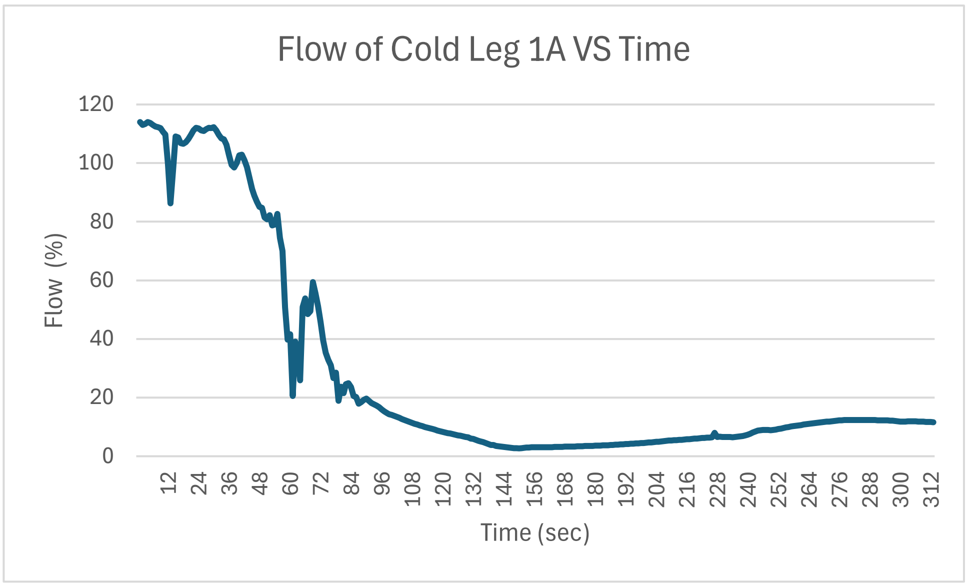


Figure C-11. Flow of Cold Leg 1A (At 10% Break)

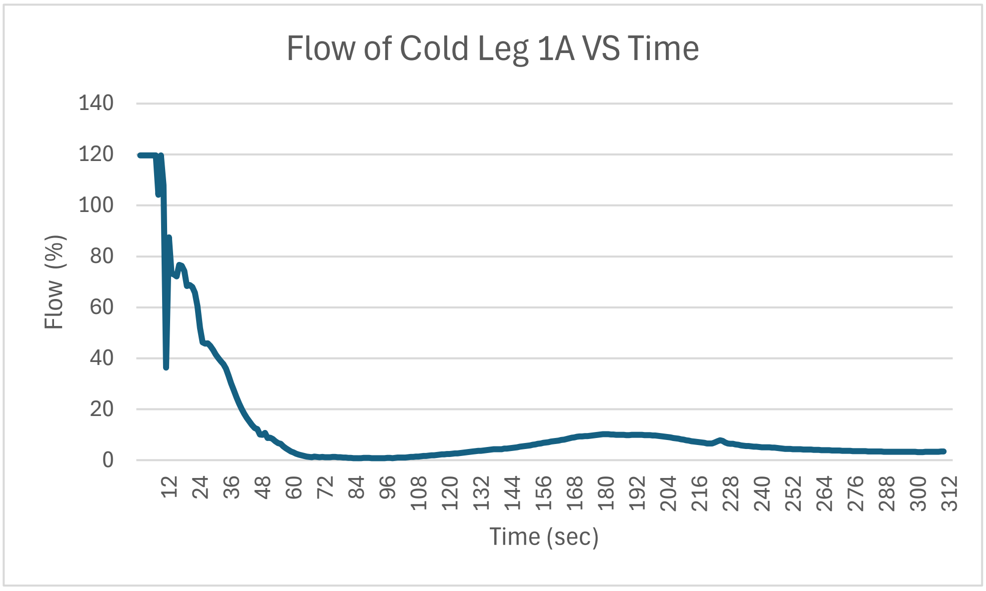


Figure C-12. Flow of Cold Leg 1A (At 100% Break)

Wide Ring Cold Leg 1A Temperature (13)

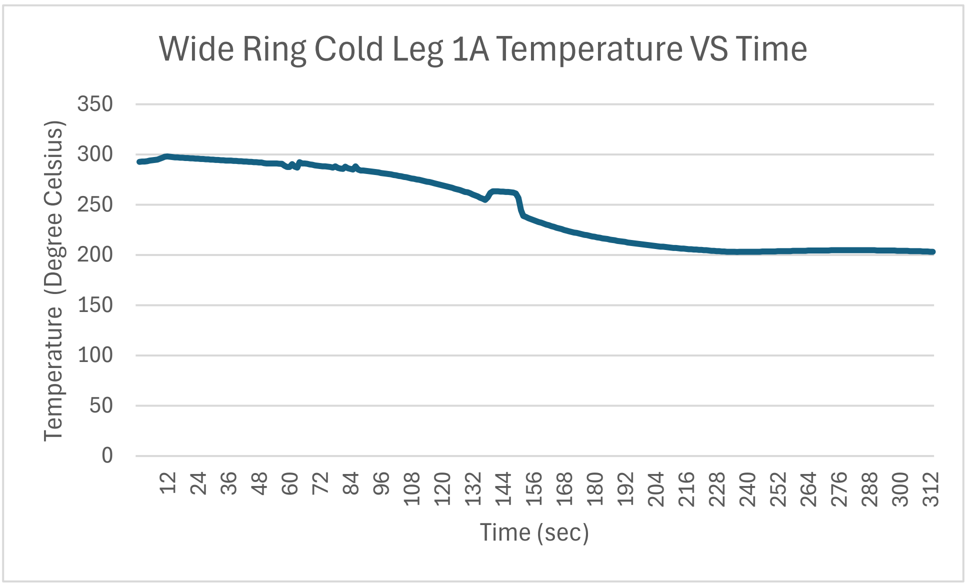


Figure C-13. Wide Ring Cold Leg 1A Temperature (At 10% Break)

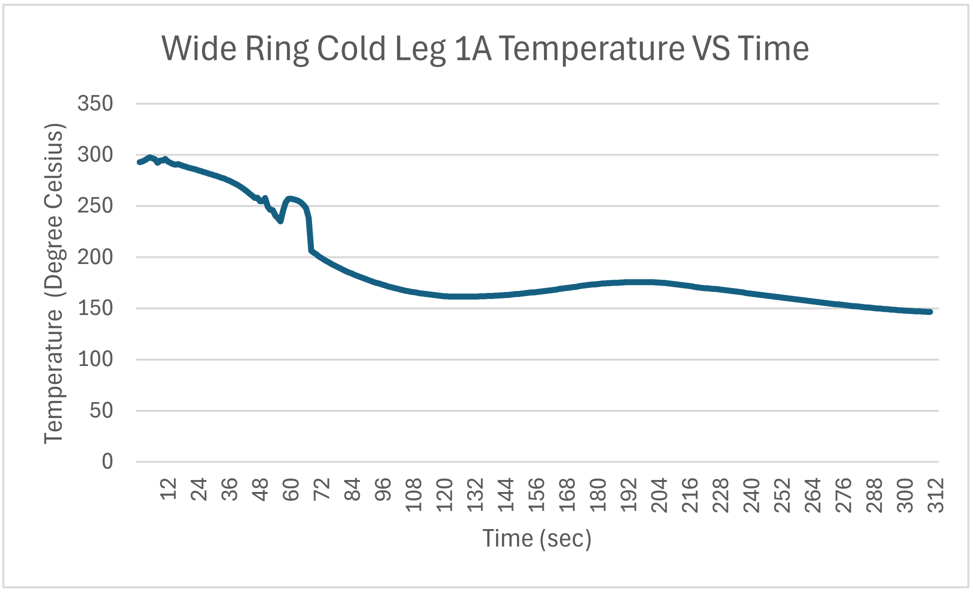


Figure C-14. Wide Ring Cold Leg 1A Temperature (At 100% Break)

Wide Ring Cold Leg 2A Temperature (18)

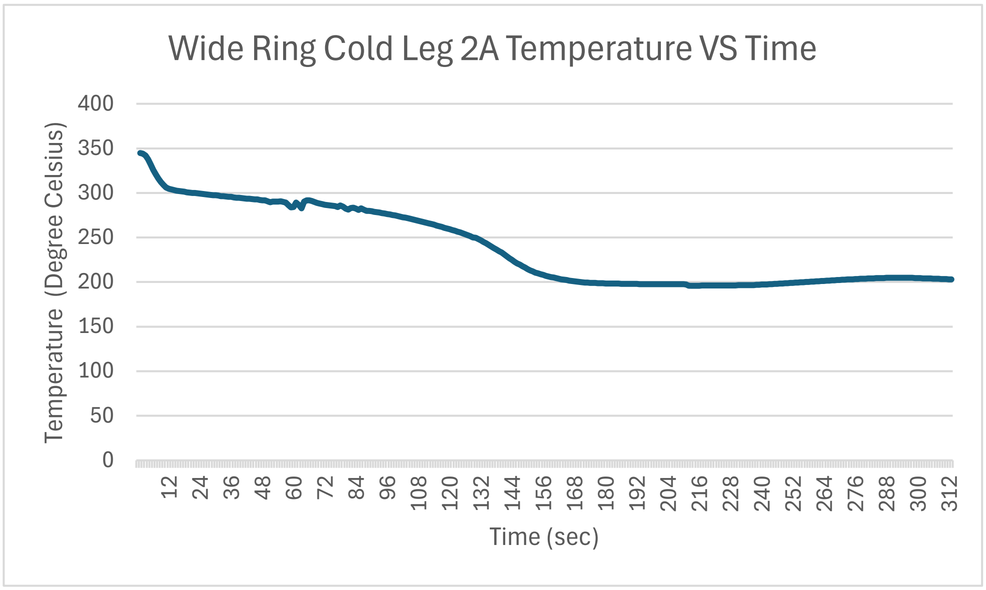


Figure C-15: Wide Ring Cold Leg 2A Temperature (At 10% Break)

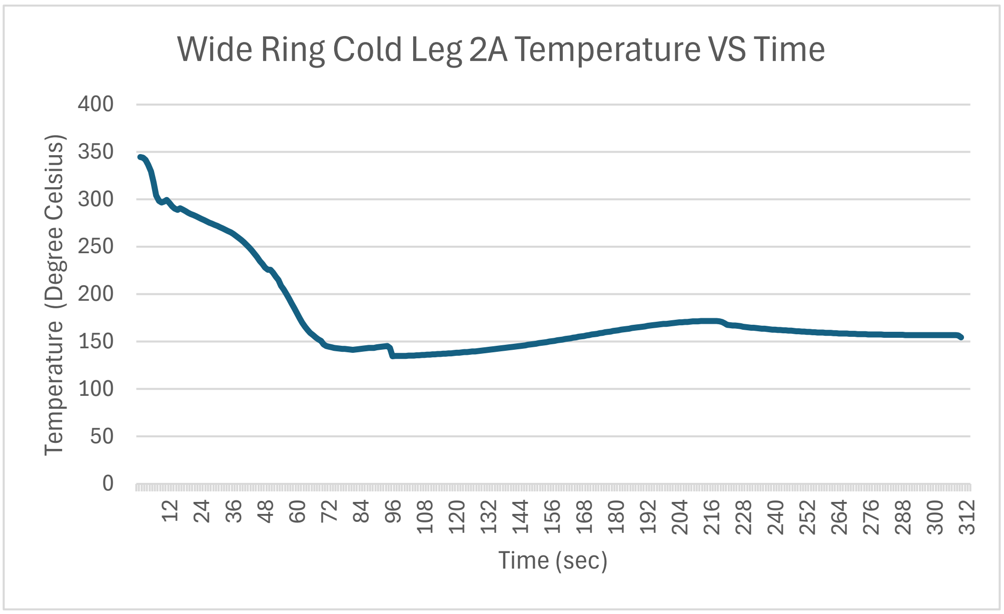


Figure C-16. Wide Ring Cold Leg 2A Temperature (At 100% Break)

Margin To Saturation (19)

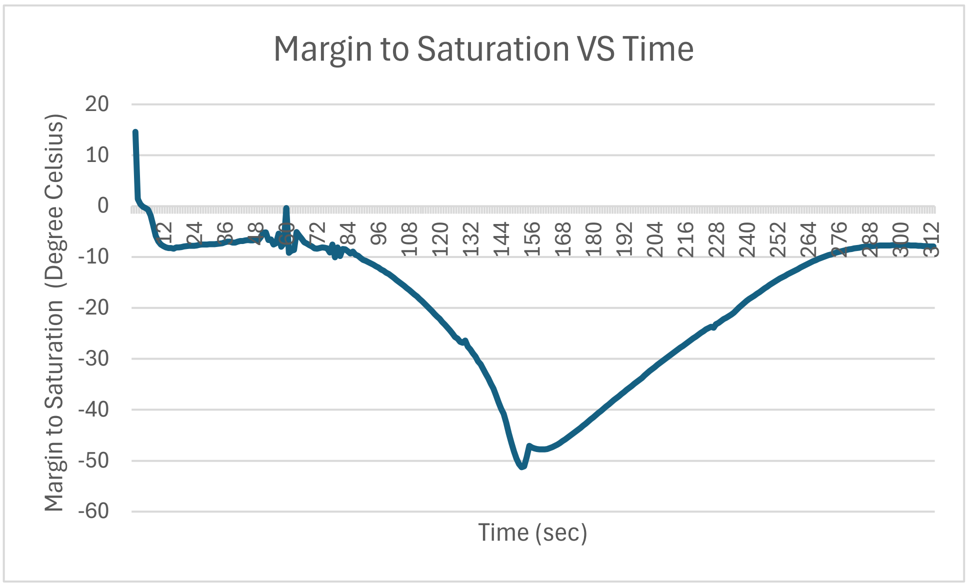


Figure C-18. Margin to Saturation (At 10% Break)

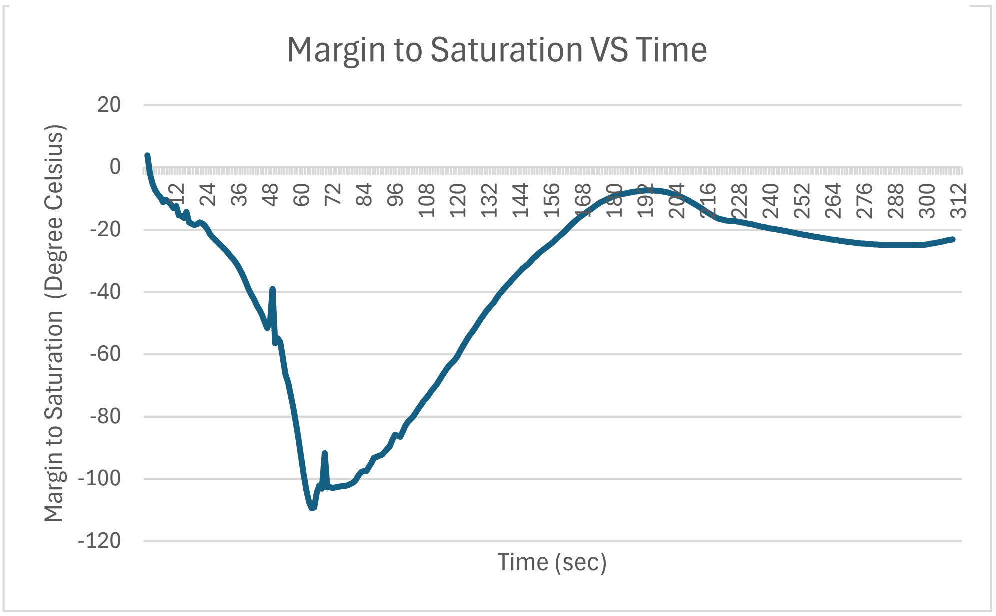


Figure C-17. Margin to Saturation (At 100% Break)