Deggendorf Institute of Technology

Department of Electrical Engineering and IT

Project as Elective Course:

Teaching Artificial Neural Networks To Automate the Design of Phase-lead Compensator, Based on Root-Locus approach

Student: Zineddine Bettouche, MET 00809173

Supervisor: Prof. Nikolaus Müller

Deggendorf, 2021

Abstract

Automation has many advantages: higher production rates and increased productivity, more efficient use of materials, better product quality, improved safety, shorter workweeks for labor, and reduced factory lead times. Higher output and increased productivity have been two of the biggest reasons in justifying the use of automation. Despite the claims of high quality from good workmanship by humans, automated systems typically perform the manufacturing process with less variability than human workers do, resulting in greater control and consistency of product quality. In addition, increased process control makes use of materials more efficiently, resulting in less scrap. Furthermore, with the introduction of Machine Learning and Deep Learning, it is even more convenient to not only automate tasks, but also to teach artificial intelligence models to solve certain tasks and make predictions and classifications for new data.

Content

1. Introduction

People learn from experience. The richer our experiences, the more we can learn. In the artificial intelligence (AI) discipline known as deep learning, the same can be said for machines powered by AI hardware and software. The experiences through which machines can learn are defined by the data they acquire, and the quantity and quality of data determine how much they can learn.

Deep learning is a branch of machine learning. Unlike traditional machine learning algorithms, many of which have a finite capacity to learn no matter how much data they acquire, deep learning systems can improve their performance with access to more data: the machine version of more experience. After machines have gained enough experience through deep learning, they can be put to work for specific tasks such as driving a car, detecting weeds in a field of crops, detecting diseases, inspecting machinery to identify faults, and so on.

1.1. Aim and Motivation

As it is established earlier, artificial neural network can learn to modulate relations between the input data and the desired outcome from the training data and their labels.

Although engineering is known to belong to exact sciences, few engineering topics do not bind to a specific set of statements to solve their beheld issues. In such cases, intuition and experience are what engineers rely on to come up with acceptable designs and solutions. One of these topics, is the design of compensators based on Root-Locus, that would be further discussed in details in upcoming chapters. The speed, in which an engineer can design an acceptable compensator, is proportional to his experience. Therefore, this procedure of solving such a task gives rise to the following question: if experience is what it takes for an engineer to become faster at designing compensators, can an artificial neural network be trained to solve the task at hand?

As it is clear by now, this work is about a trial of teaching a neural network how to design a phase-lead compensator, which is again a task that is not bound by defined rules and can result in different solutions that might be equally acceptable.

1.2. Structure of Work

In this paper, the fundamental definitions and ideas of Control Theory and specifically designing a phase-lead compensator based on Root-Locus approach are to be introduced first. Due to lack of data at the beginning of this work, a Python program was built to generate a significant amount of data to later train the designed neural network. The program is going to be explained and its functionality is going to be documented accordingly. Afterwards, the process of building and training an artificial neural network is going to be introduced, which would be the initiation of the experimental part, in which different experiments and their outcomes are going to be discussed. A conclusion at the end is drawn, based on the previously acquired results, and further needed work is going to be suggestively stated.

2. Related Work

Work related to artificial neural network (ANN) design is going to be placed here in the future, as it is going to be a tool that helps the development of this work's experimental part. However, when it comes to specifically using ANN to design compensators, no work was perceived nor found, so far at least.

3. Background

In this chapter of the thesis, some processes, structures and programs that are relevant for the work are dealt with. This important information is presented to facilitate an overview of these topics.

3.1. Control Theory

In engineering and mathematics, control theory deals with the behavior of dynamical systems. The desired output of a system is called the reference. When one or more output variables of a system need to follow a certain reference over time, a controller manipulates the inputs to a system to obtain the desired effect on the output of the system. Rapid advances in digital system technology have radically altered the control design options. It has become routinely practicable to design very complicated digital controllers and to carry out the extensive calculations required for their design. These advances in implementation and design capability can be obtained at low cost because of the widespread availability of inexpensive and powerful digital processing platforms and high-speed analog IO devices.

3.2. Root Locus

The root locus [1] is the path of the roots of the characteristic equation traced out in the s-plane as a system parameter varies from zero to infinity. The relative stability and the transient performance of a closed-loop control system are directly related to the location of the closed-loop roots of the characteristic equation in the s-plane. It is frequently necessary to adjust one or more system parameters in order to obtain suitable root locations.

Therefore, it is worthwhile to determine how the roots of the characteristic equation of a given system migrate about the s-plane as the parameters are varied; that is, it is useful to determine the locus of roots in the s-plane as a parameter is varied. The root locus method was introduced by Evans in 1948 and has been developed and utilized extensively in control engineering practice. The root locus technique is a graphical method for sketching the locus of roots in the s-plane as a parameter is varied. In fact, the root locus method provides the engineer with a measure of the sensitivity of the roots of the system to a variation in the parameter being considered. The root locus technique may be used to great advantage in conjunction with the Routh-Hurwitz criterion. The root locus method provides graphical information, and therefore an approximate sketch can be used to obtain qualitative information concerning the stability and performance of the system. Furthermore, the locus of roots of the characteristic equation of a multi-loop system may be investigated as readily as for a single-loop system. If the root locations are not satisfactory, the necessary parameter adjustments often can be readily ascertained from the root locus.

3.3. The Design of Feedback Control Systems

In control theory [1], stable control systems resulting in an acceptable response to input commands, are less sensitive to system parameter changes, result in a minimum steady-state error for input commands, and, finally, are able to reduce the effect of undesirable disturbances. However, a feedback control system that provides an optimum performance without any necessary adjustments is rare indeed. Usually, it is necessary to compromise among the many conflicting and demanding specifications and to adjust the system parameters to provide a suitable and acceptable performance when it is not possible to obtain all the desired optimum specifications. For instance, if a set of performance measures is desired to be less than some specified values, a conflicting set of requirements is often encountered. Hence, if it is wished to have a system with a percent overshoot less than 20% and a settling time of 3.3 seconds, a conflicting requirement on the system damping-ratio is obtained. If it is impossible to relax these two performance requirements, the system must be altered in some way. The alteration or adjustment of a control system in order to provide a suitable performance is called compensation; that is, compensation is the adjustment of a system in order to make up for deficiencies or inadequacies.

3.3.1 Compensators

A compensator [1] is an additional component or circuit that is inserted into a control system to compensate for a deficient performance. The compensation network function $G_c(s)$ is cascaded with the specified process G(s) in order to provide a suitable loop transfer function

$$L(s) = G_c(s) * G(s)$$

The compensator $G_c(s)$ can be chosen to alter either the shape of the root locus or the frequency response. Consider the first-order compensator with the transfer function $G_c(s) = k * (s+z)/(s+p)$ the design problem then becomes the selection of z, p, and k in order to provide a suitable performance. When $|z| < |p|$, the network is called a phase-lead network and has a pole-zero s plane configuration, which is the type of compensator that this work is going to train ANNs to design.

3.3.2. Design: General Guidelines

1. Analyze (simulate) the uncompensated system, if it meets the specification: stop here.
2. Select the compensator type (the simpler, the better)
3. Tune the compensator parameters (here, by help of the root-locus)
4. Simulate the compensated system, if not satisfactory go back to (3, or 2)

3.4. Artificial Neural Networks

Artificial neural networks (ANN) give machines the ability to process data similar to the human brain and make decisions or take actions based on the data. While there's still more to develop before machines have similar imaginations and reasoning power as humans, ANNs help machines complete and learn from the tasks they perform. Ultimately, ANNs try to replicate how our human brains process information and make decisions. While ANNs are based on

mathematical theory created in the 1940s, it was not until the last couple of decades that it became a focus for artificial intelligence.

When backpropagation was developed to help these networks learn and adjust actions based on outcomes its development and adoption really began to accelerate. When a human brain receives an input, it processes it through a series of neurons. Different neurons of the human brain are responsible for processing different aspects of input in a hierarchical fashion. ANNs try to replicate this through artificial neurons called units that are arranged in layers and connected to each other to create a web-like structure. ANNs have an input layer and output layer. Between these two layers, there are other hidden layers that perform the mathematical computations that help determine the decision or action the machine should take. Ultimately, these hidden layers are in place to transform the input data into something the output unit can use. The data is processed by each hidden layer and then moves on to the next based on connections that are weighted. Think of this process as an assembly line in a factory—raw materials as the input and different stops on the conveyor belt to add an element to the product equate to the hidden layers of an ANN that processes the data until you get to the output. Based on what the machine learns about the data when processed by one layer, it determines how to move it through to the next, more senior layer based on the value it receives when evaluated. Based on the complexity of the issue at hand, it can continue to process through more senior units until delivered to the output layer. Before an ANN can be fully deployed, it must be trained. This training involves comparing an outcome a machine gets with the human-provided description of what outcome is expected. If these do not match, the machine uses this feedback and goes back to adjust the weights of the layers (called backpropagation). These new learning rules are applied and help guide the neural networks in future processing. To illustrate how this works for the human brain, consider how humans might learn how to shoot a basketball so they score more baskets. Over time and with experience, different techniques are tried to improve the odds the shot will make it in the basket—bending legs less or more, adjusting the hand position, shooting force, the angle of the shot, use of backboard, etc. When a shot does not make it in, the brain adjusts based on this feedback and tries something else. Over time, there is enough learning to improve the outcome so that more balls make it through the net than miss it.

3.4.1. The Dense layer use in ANNs for Regression

Layers [2] in the deep learning model can be considered as the architecture of the model. There can be various types of layers that can be used in the models. All of these different layers have their own importance based on their features. For example, LSTM layers are mostly used in the time series analysis or in the Natural Language Processing problems, convolutional layers in image processing, etc. A dense layer, also referred to as a fully connected layer, is a layer that is used usually in the final stages of the neural network. This layer helps in changing the dimensionality of the output from the preceding layer so that the model can easily define the relationship between the values of the data in which the model is working. This relationship can come in handy when the task at hand is under the class of regression.

In any neural network, a dense layer is a layer that is deeply connected with its preceding layer, which means the neurons of the layer are connected to every neuron of its preceding layer. The dense layer's neuron in a model receives output from every neuron of its preceding layer, where neurons of the dense layer perform matrix-vector multiplication.

Matrix vector multiplication is a procedure where the row vector of the output from the preceding layers is equal to the column vector of the dense layer. The general rule of matrix-vector multiplication is that the row vector must have as many columns as the column vector.

The general formula for a matrix-vector product is:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

Values under the matrix are the trained parameters of the preceding layers and also can be updated by the backpropagation. Backpropagation is the most commonly used algorithm for training the feedforward neural networks. Generally, backpropagation in a neural network computes the gradient of the loss function with respect to the weights of the network for single input or output. From the above intuition, we can say that the output coming from the dense layer will be an N-dimensional vector. We can see that it is reducing the dimension of the vectors. Finally, one of the most important parameters in ANN layers is the activation functions.

3.4.2. Activation Functions

In neural networks, the activation function [3] is a function that is used for the transformation of the input values of neurons. It introduces the non-linearity into the ANN so that the networks can learn the relationship between the input and output values. An activation function decides whether a neuron should be fired or not. Whether the information that the neuron is receiving is relevant for prediction or should be ignored.

The types of activation functions fall under one of the following:

- **Binary Step:** This activation function is useful when the input pattern can only belong to one or two groups, that is, binary classification. However, the step function would not be able to classify more than one class classifier.

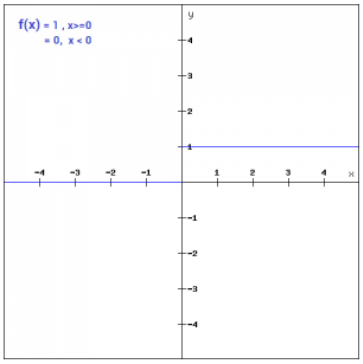


Figure: binary step

- **Sigmoid:** Usually used in output layer of a binary classification, where result is either 0 or 1 as value for the sigmoid function lies between 0 and 1. Therefore, the result can be mapped to be 1 if value is greater than 0.5 or 0 otherwise. However, sigmoid saturates and kills gradients and performs badly in hidden Layers.

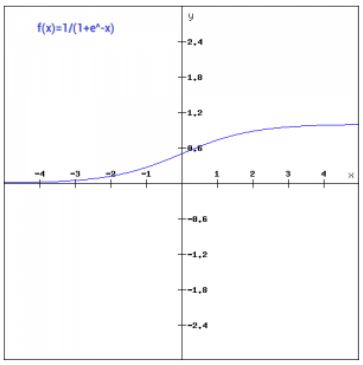


Figure: Sigmoid

- **Tanh:** Usually used in hidden layers of a neural network as its values lie between -1 and 1 hence the mean for the hidden layer comes out be 0 or very close to it, which means, it helps in centering the data by bringing the mean close to 0. This makes learning for the next layer much easier. However, similar to sigmoid, it also has a vanishing gradient problem.

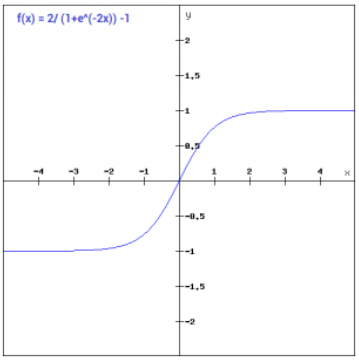


Figure: Tanh

- **ReLU:** The Rectified Linear Activation Unit less computationally expensive than Tanh and sigmoid because it involves simpler mathematical operations. At a time, only a few neurons are activated making the network sparse efficient and easy for computation. ReLU neurons have the tendency to become inactive for all inputs, i.e., they tend to die out. This can be caused by high learning rates, and can thus reduce the model's learning capacity.

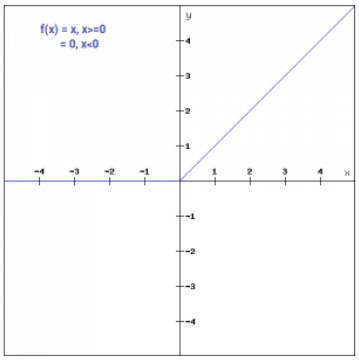


Figure: RelU

- **Softmax:** It is useful when classifying more than two classes. Unlike sigmoid, it can handle multiple classification problem. However, it can only be used in Output Layer.

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

After the quick summary of each of the activation function types, it becomes clear that the most suitable activation function for a regression task, is the linear function.

3.4.2. Epochs and Loss Metrics

In deep learning, the loss is the value that a neural network is trying to minimize: it's the distance between the ground truth and the predictions. In order to minimize this distance, the neural network learns by adjusting weights and biases in a manner that reduces the loss. For instance, in regression tasks, you have a continuous target, e.g., height. What you want to minimize is the difference between the predictions, and the actual height. *mean_absolute_error* can be used for example as loss so the neural network knows this is what it needs to minimize.

In classification, it's a little more complicated, but very similar. Predicted classes are based on probability. The loss is therefore also based on probability. The neural network minimizes the likelihood to assign a low probability to the actual class. The loss is typically *categorical_crossentropy*.

Loss and *val_loss* (validation-loss) differ because the former is applied to the train set, and the latter to the test set. As such, the latter is a good indication of how the model performs on unseen data. It's best to rely on the *val_loss* to prevent overfitting. Overfitting is when the model fits the training data too closely, and the loss keeps decreasing while the val_loss is stable, or increases.

The Mean Squared Error (MSE) loss is the default loss to use for regression problems. Mathematically, it is the preferred loss function under the inference framework of maximum likelihood if the distribution of the target variable is Gaussian. It is the loss function to be evaluated first and only changed if you have a good reason. Mean squared error is calculated as the average of the squared differences between the predicted and actual values. The result is always positive regardless of the sign of the predicted and actual values and a perfect value is 0.0. The squaring means that larger mistakes result in more error than smaller mistakes, meaning that the model is punished for making larger mistakes.

There may be regression problems in which the target value has a spread of values and when predicting a large value, you may not want to punish a model as heavily as mean squared error. Instead, the natural logarithm of each of the predicted values can be calculated first, then the mean squared error. This is called the Mean Squared Logarithmic Error loss, or MSLE for short. It has the effect of relaxing the punishing effect of large differences in large predicted values. As a loss measure, it may be more appropriate when the model is predicting unscaled quantities directly.

On some regression problems, the distribution of the target variable may be mostly Gaussian, but may have outliers, e.g. large or small values far from the mean value. The Mean Absolute Error, or MAE, loss is an appropriate loss function in this case as it is more robust to outliers. It is calculated as the average of the absolute difference between the actual and predicted values.

There are other loss functions for classification ANNs, however that is not in the scope of this work.

Finally, it is worth stating that an experiment is done in this project to determine the most fitting loss function and epochs number for the ANN.

3.5. Python Programming Language

Python is an interpreted, general-purpose programming language, which is one of the most popular choices for modern software development. Python is modular – i.e. it can be easily integrated with other technologies and solutions. Secondly, it is open-source. There is a vibrant community of developers, who contribute to the development of the technology, and the Python Software Foundation oversees the quality and the direction in which the language is going. Thirdly, Python is interpreted, which means it is translated to machine code right before the program is launched. This facilitates writing portable and universal programs, which are easier to use on different operating systems.

AI projects differ from traditional software projects. The differences lie in the technology stack, the skills required for an AI-based project, and the necessity of deep research. To implement certain AI aspirations, the programming language that should be used, must be stable, flexible, and includes practical tools. Python offers all of this, which is why there are many Python AI projects today. From development to deployment and maintenance, Python helps developers be productive and confident about the software they are building. Benefits that make Python the best fit for machine learning and AI-based projects include simplicity and consistency, access to great libraries and frameworks for AI and machine learning (ML), flexibility, platform independence, and a wide community. These add to the overall popularity of the language.

3.5.1. Keras Deep Learning Framework

Keras [2] is a widely used and a very popular deep learning framework. It represents an interface to the TensorFlow backend. The Keras API was developed with the focus on enabling rapid experimentation. “Getting from the idea to the result as quickly as possible is the key to good research” is a principle of the Keras community. TensorFlow is an end-to-end, open-source machine learning platform. You can think of it as an infrastructure layer for differentiable programming. It has now developed into a quasi-standard in the area of programming neural networks and deep learning. TensorFlow is characterized by its high performance and good scalability Keras is TensorFlow's high-level API: an understandable, highly productive interface for solving machine-learning problems with a focus on modern deep learning.

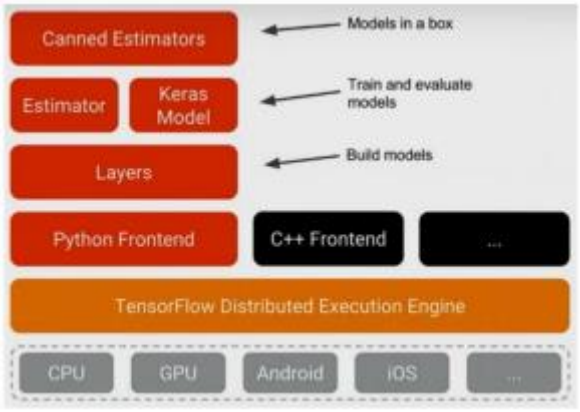


Figure: Keras layout over Tensorflow

It offers essential abstractions and building blocks for the development and integration of machine learning solutions. Keras enables the cross-platform capabilities of TensorFlow to be

exploited: Keras can run on TPU or on large clusters of GPUs. Keras models can be exported in order to run them in the browser or on a mobile device.

4. Methodology

This section describes the structures and pipelines of development. The description includes how the program generates data with which the neural network gets trained and tested.

4.1. Dataset

When the idea for this project was crafted, acquiring enough dataset to train a neural network was the biggest obstacle. Therefore a python program was written to generate data.

4.1.1. Data-Generating Program

As already mentioned, a Python program was written to come up with compensator parameters that satisfy the transient-time conditions. The script executes the following flowchart:

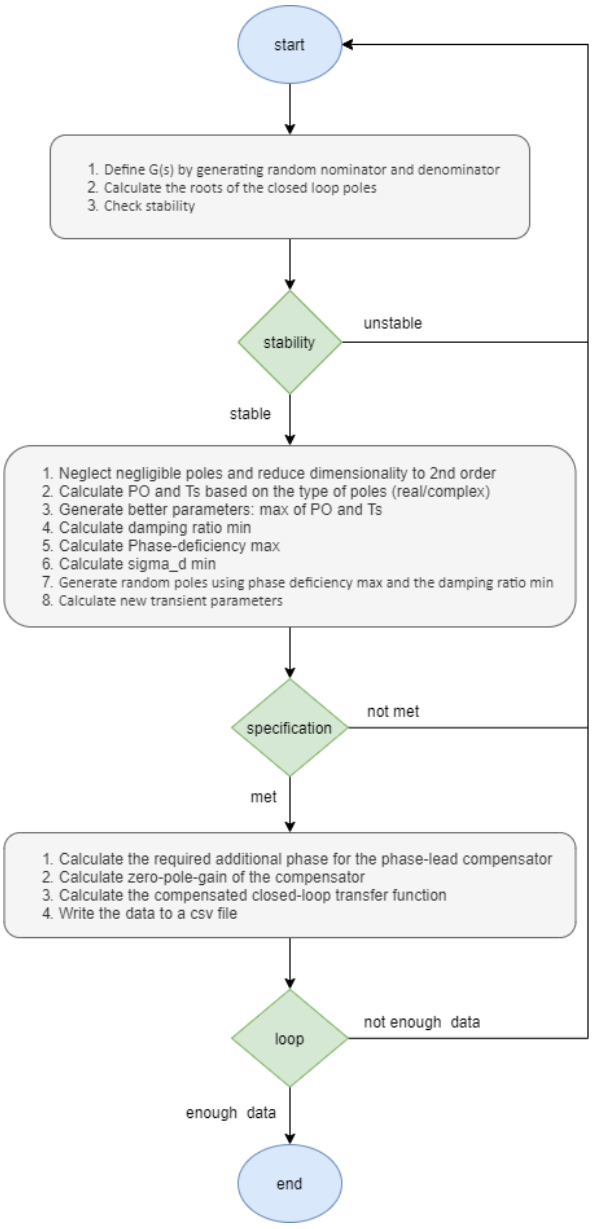


Figure: Execution flow of data-generating program

4.1.2. Data Info

The data-generating program as displayed previously, generates a transfer function with no zeros and 2 poles. This was established to simplify the process and focus on the learning aspect of the neural network. In more details, the program generates the transfer function randomly, checks its poles, and only if they are distinct and on the left hand panel of the s-plane (negative) they are kept. Otherwise the program restarts from ground zero. If the condition is met, it calculates the current specifications of the system, meaning that it calculates the closed-loop TF and then both of PO and Ts. The program then defines better specifications (making them both smaller) and then aims to solve for a transfer function (Gc) which when cascaded to the open loop transfer function would reach the defined specifications. The solving process is semi-random process generating a complex pole. This is implemented in a loop that randomly generates this pole and checks for its fitness in the solving process. If more than a 1000 loops are gone through, and no solution was found, the program simply starts from ground zero.

After letting the program run for some time, it gathered over 100000 rows. There is a total of 41 columns of all the data stored. At first, many of the columns were stored as complex numbers. This can cause issues in the learning process of a neural network. It is more suitable to divide the complex number into its real and imaginary parts and store separately. The data stored was as follows:

- Uncompensated open-loop: nominator (uoln), denominator (uold 0…3)
- Uncompensated closed-loop: denominator(ucld 0…3), poles (uclp 0…2 –r or -i)
- PO max (percentage overshoot), Ts max (settling time), c min (damping factor), phase deficiency max, sigma_d min.
- Compensation pole (pd -r, pd -i, pd phase)
- New c, new PO, new Ts, required additional phase
- Compensation transfer function (Gc-z, Gc-p, Gc-k)
- Compensated closed-loop: nominator (ccln), denominator (ccld 0…3), poles (cclp 0…2 –r or -i)

It is important to note that no column is going to be used, but the input and output columns:

- **Inputs:** PO max, Ts max, and the uncompensated open-loop TF uoln, uold-0…uold-3
- **Outputs:** The compensation TF: Gc-z, Gc-p, Gc-k

4.1.3. Data Standardization and Normalization

Deep learning neural networks learn how to map inputs to outputs from examples in a training dataset. The weights of the model are initialized to small random values and updated via an optimization algorithm in response to estimates of error on the training dataset. Given the use of small weights in the model and the use of error between predictions and expected values, the scale of inputs and outputs used to train the model are an important factor. Unscaled input variables can result in a slow or unstable learning process, whereas unscaled target variables on regression problems can result in exploding gradients causing the learning process to fail. Data preparation involves using techniques [4] such as the normalization and standardization to rescale input and output variables prior to training a neural network model.

Normalization: rescaling of the data from the original range so that all values are within the range of 0 and 1. Normalization requires that we know or are able to accurately estimate the

minimum and maximum observable values. We may be able to estimate these values from the available data. Here y is the normalized value of x:

$$y = \frac{x - min}{\max - min}$$

Standardization: Standardizing a dataset involves rescaling the distribution of values so that the mean of observed values is 0 and the standard deviation is 1. It is sometimes referred to as "whitening." This can be thought of as subtracting the mean value or centering the data. Like normalization, standardization can be useful, and even required in some machine learning algorithms when the data has input values with differing scales. Standardization assumes that the observations fit a Gaussian distribution (bell curve) with a well behaved mean and standard deviation. We can still standardize the data if this expectation is not met, but We may not get reliable results. Standardization requires that we know or are able to accurately estimate the mean and standard deviation of observable values. We may be able to estimate these values from your training data. Here y is the standardized value of x:

$$y = \frac{x - mean}{standard\_deviation}$$

$$mean = \frac{\sum x}{count(x)}$$

$$standard\_deviation = \sqrt{\frac{\sum (x - mean)^2}{count(x)}}$$

In the following sections, an experiment will be conducted to calculate the distribution of the inputs and outputs of our dataset, in order to decide whether it should be standardized or normalized.

4.2. Project Directory Structure

The directory of the project is structured as follows:

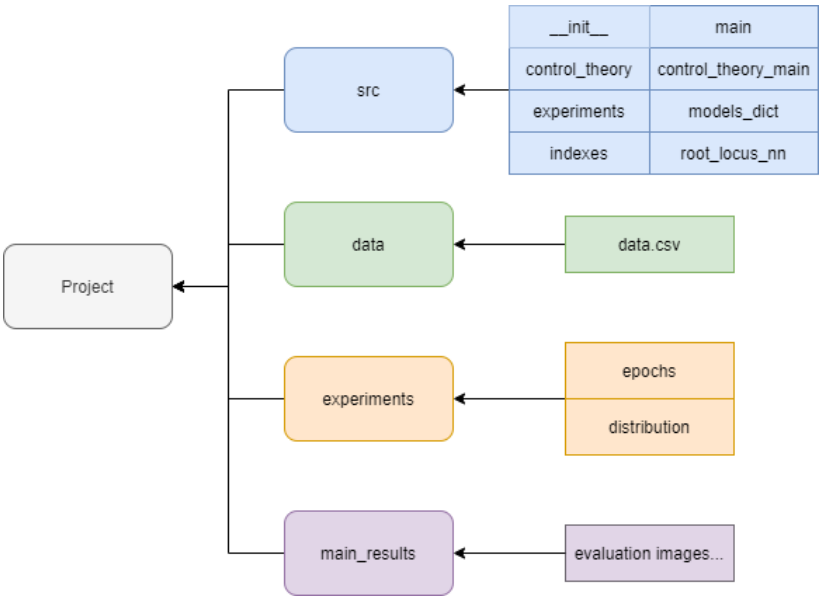


Figure: Project structure

The most important directory is the *src* folder, as it contains all the scripts. These scripts are going to be discussed in a bit of details in the upcoming subsections. The data directory is the directory containing the excel file generated from running the control theory scripts. It acts in this project as the main dataset to train and evaluate the neural networks constructed. The experiments folder contains all of the results of the experiments in its subdirectories. Finally, the main_results folder is the directory that the main script stores its results in.

4.3. Neural Network Structure

With the help of Keras framework, a Python script was defined to train and use the ANN. The functional structure of the neural network will be first discussed with its possible variations. Following that, the developed Python module within the scope of this work will be examined in more details.

4.3.1. Functional Structure

As discussed earlier, there exist many combinations between the input and output columns. This has led to create a Python class that is reusable enough to define an ANN model for every case of inputs-outputs combination.

Having a regression task, the dense layer was used to create the network. The input/output layers depend on how many inputs/outputs there are, whereas the number of hidden layers is three (Nh), and it can be calculated with the following formula:

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))}$$

N_i = number of input neurons.
N_o = number of output neurons.
N_s = number of samples in training data set.
α = an arbitrary scaling factor usually 2-10.

Therefore, our neural network would be of the shape of:

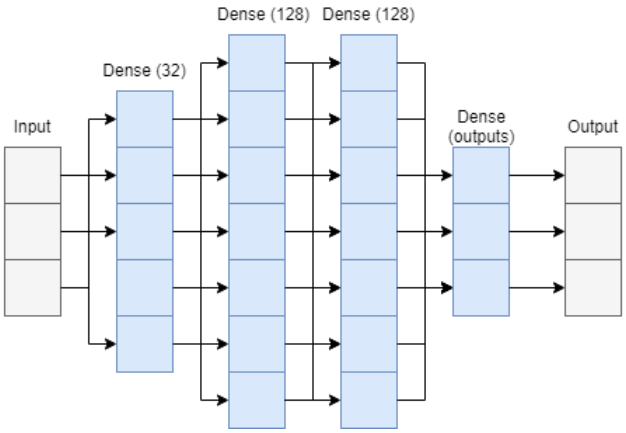


Figure: ANN layers connection

The between brackets represents the number of units in each layer. It was taken conventionally. However, it is suggested that an experiment is done to observe the effect of having different

numbers of hidden-layers on the learning process, which will be presented in the experiments section.

4.3.2. RootLocusNN – Python Class

This Python script, which was created as part of this work, is responsible for training the autoencoder. It was developed into a Python class to be used more efficiently in an object-oriented programming manner. The module was tested with both versions of Python: 3.8.5 and 3.6.8. A module essentially refers to a Python script that provides functionality, which in return can be imported into another script in order to provide more functions in a more robust way that can be beneficial to the speed of development and credited to increase the reusability of the code. Furthermore, to ensure that the package conflicts do not occur when a user tries to run the code, a requirements file was built through pip-tools to contain all of dependencies, so that the user's Python environment operates on the same versions of libraries that the original code was developed on.

It is far more efficient to use a graphics processing unit GPU for training, rather than using a common central processing unit CPU. This is a product of the nature of neural networks, as they rely on passing tensors or more simply matrices, when GPUs are designed to operate on similar tasks. Therefore, it makes more sense to use a GPU as a hardware component, dedicated to training the ANN. In order to accomplish such a task, certain libraries are required depending on the hardware manufacturer.

One of the most widely used and supported libraries is NVidia's CUDA, which ensures the optimization of the interaction between Keras, TensorFlow, and the graphics card, resulting in a faster training process.

The following figure displays the class structure:

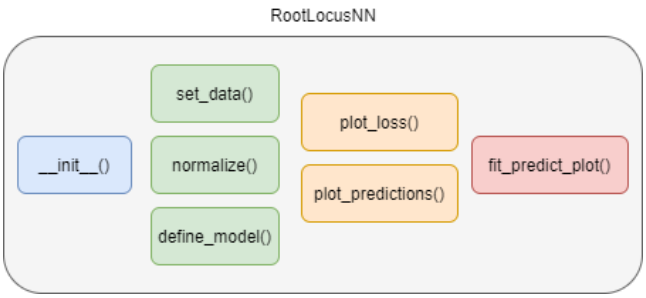


Figure: RootLocusNN class structure

- `normalize(self, df)`: normalize the data.
- `set_data(self, data, scale)`: splits the data into training and testing depending on the input and output columns. If the scale argument is true, it calls the normalize method on the splits.
- `__init__(self, model_name, data, input_keys, output_keys, scale)`: this is the object initializing method. It sets the model name, the input and output keys, and finally calls the *set_data()* method.
- `define_model(self):` defines the model depending on the input and output keys.
- `plot_loss(self, folder_path): plotting loss experienced in training and evaluating phases.`

- `plot_predictions(self, folder_path, titles):` plotting the predicted values next to the testing values for comparison.
- `fit_predict_plot(self, batch_size, epochs):` this method is the most important method in the class. It calls *define_model()* to define the model, it trains the defined model with the passed epochs and batch size. The method then creates a directory and stores the trained model for future use, and calls the two plotting methods which both store their plots in the created directory.

4.4. Python Scripts

4.4.1. Control Theory: *control_theory.py* and *control_theory_main.py*

The two scripts modulate the solution created to generate the dataset used in this project. c*ontrol_theory()* is the script holding all the functions used in *main_theory_main* script, whereas the "main" script in this scope contains the algorithm followed to generate transfer functions, and to try to solve for a compensating transfer function and store all of the obtained data.

4.4.2. Experiments: *experiment.py*

All of the experiments undergone in this work, are written in this script, which has the following functions:

- `plot(title, column, bins, outputName, outputFolder):` plotting a histogram of the passed columns to visualize the occurrence of every value.
- `get_best_distribution(data):` calling plot to get the histogram of occurrence, and then calculating the probability of falling that histogram into one of the most common distributions.
- `distribution_experiment(data):` performing the distribution experiment on the passed data by calling the two previously mentioned functions.
- `epochs_experiment(data):` performs the epoch experiment that can determine the fitting of an epoch in training the model.

4.4.3. Main Script: *main.py*

The main script is only used at the end, after tuning the parameters of the neural network (in the experiments part). The program written in the script, uses the neural network with the tuned parameters, in order to observe the final results of the work, which are the predictions of a compensation TF (Gc) and how close they are to the expected values.

5. Experiments

In this section, the experiments done in this work are discussed. The thought behind them more importantly is rationalized and explained in details. As mentioned in the previous sections, the experiments are all implemented in the *experiments.py* script.

5.1. Distribution Experiment

As it was explained in the section "4.1.3. Data Standardization and Normalization" it is important to determine the distributions of the inputs and outputs to decide which of the two preprocessing algorithms is more suitable for the date as a step to enhance the quality of the training of the neural network.

In this experiment, the distribution of every column of the inputs/outputs is plotted. A probability of the distribution falling into one of the most common data distribution is then calculated.

The most common data distributions are as following:

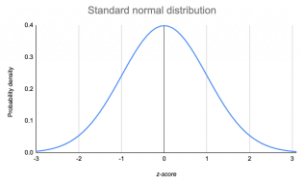
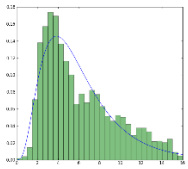
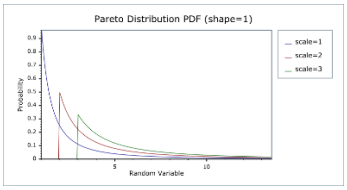
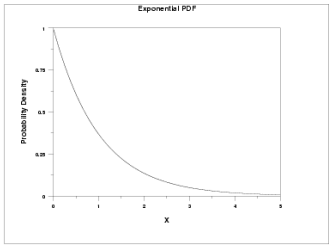
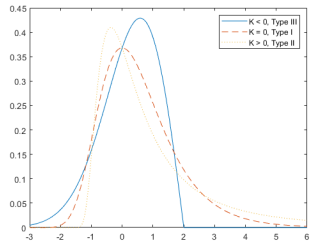
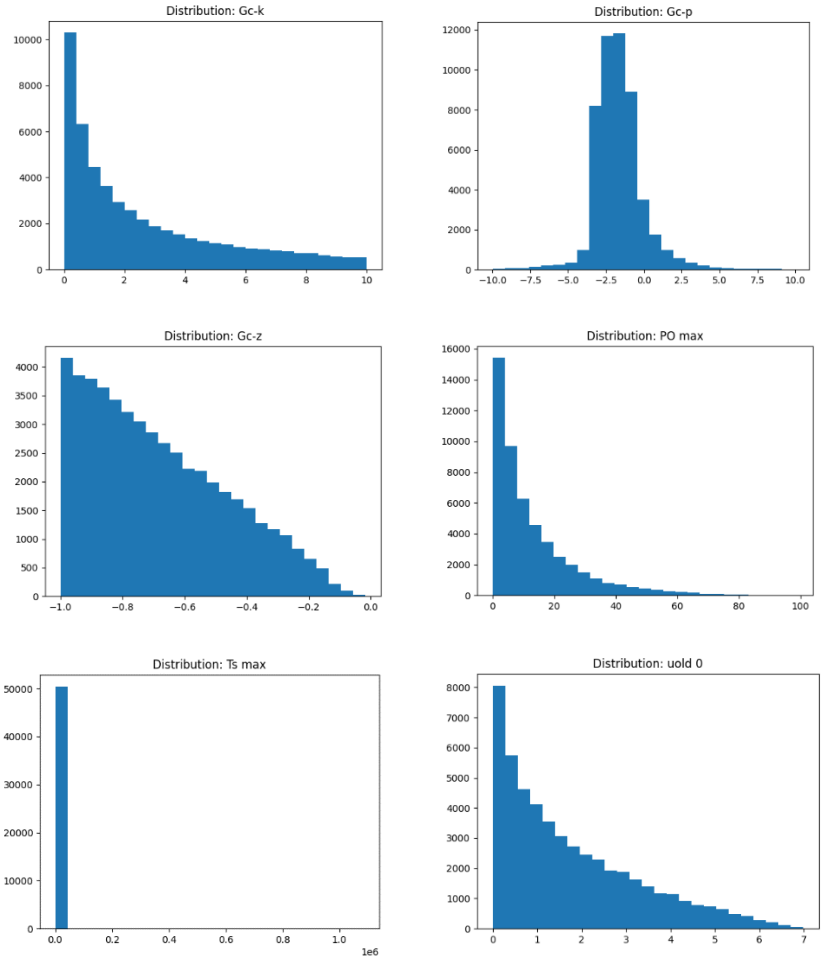
Normal	Weibull	Pareto
		
Exponential	**Generalize extreme value**	
		

Table: Common data distribution functions

The resulted distribution graphs and their probabilities are as following:



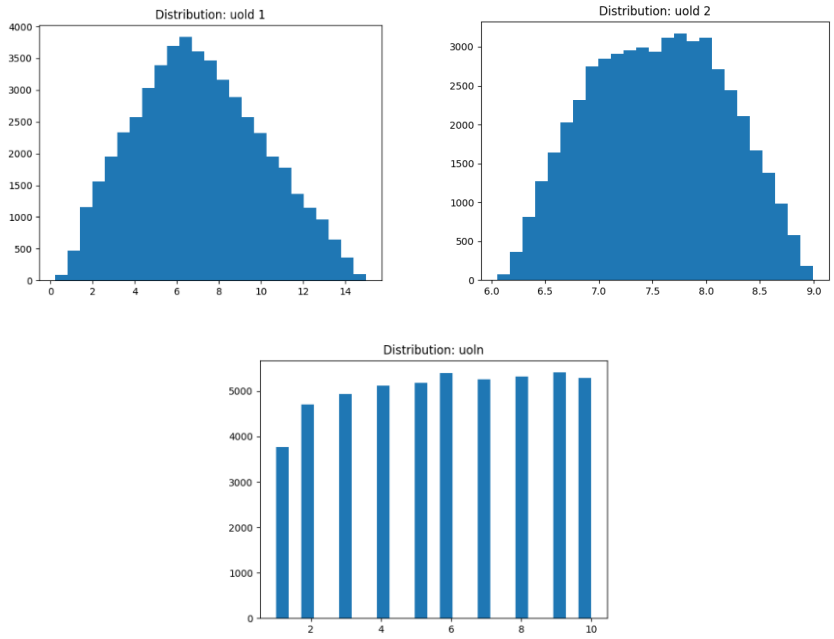


Figure: data distribution of input and output columns

After having calculated, the probability of a distribution being a common one, is calculated by applying the Kolmogorov-Smirnov test. This test is a hypothesis test procedure for determining if two samples of data are from the same distribution. The test is non-parametric and entirely agnostic to what this distribution actually is. The fact that we never have to know the distribution the samples come from is incredibly useful, especially in software and operations where the distributions are hard to express and difficult to calculate with.

The results are as follows:

	norm	exponweib	Weibull-max	Weibull-min	pareto	genextreme
uoln	0	0	0	0	0	0
uold 0	0	**7.45e-18**	1.35e-266	2.77e-60	0	5.32e-164
uold 1	2.80e-30	**8.49e-10**	7.83e-16	9.45e-11	0	7.89e-16
uold 2	1.59e-56	**3.99e-30**	4.20e-53	3.56e-49	0	4.04e-53
Ts max	0	0	0	0	0	0
PO max	0	**2.55e-32**	0	1.13e-63	0	9.65e-130
Gc-z	1.29e-277	**7.17e-24**	2.26e-108	1.84e-79	4.532e-280	3.52e-112
Gc-p	1.73e-289	**2.74e-218**	0	0	0	0
Gc-k	0	**1.53e-58**	0	1.16e-209	0	

Table: probabilities of distributions calculated, falling under one of the common distribution

As apparent, the columns do not fall under one specific distribution, and more importantly rarely are normally distributed.

A result of the experiment would be: the standardization of the data is not suitable in this case due to not meeting the condition for it, which is, as stated earlier, the data need to be in a normal distribution. Therefore, the program normalizes the data before proceeding to training and testing phases.

5.2. Epoch Experiment

The second experiment done in this project aims to determine the number epochs in training that leads to the best results in terms of training and testing losses.

The program has a list of possible epochs: [10, 20, 50, 100, 200, 500]. A model of the ANN is created, and trained for every epoch number. Finally, The loss of every training/testing is recorded and plotted.

The results of loss and accuracy are as follows:

Table: Loss of different epoch numbers

The first remark that can be made is that once the ANN is trained with more than 50 epochs, the validation error starts rising. This is known as overfitting. Although the ANN gets better at dealing with the training set, when it comes to predicting the output of new data (testing), it fails due to its exaggerated specificity with the training data, i.e. failing to generalize the input-output relationship. Therefore epochs over 100 are eliminated for all.

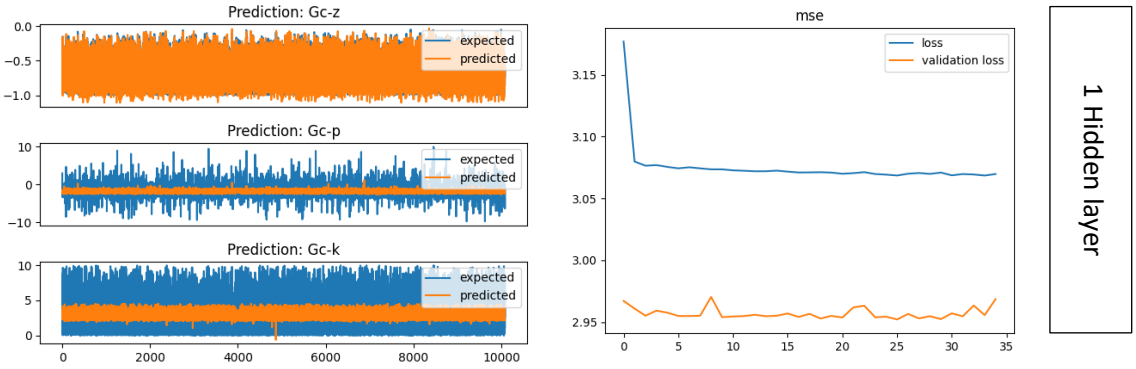
Concerning the range 10-50, the results can be acceptable. Note that the graphs may not do the fluctuations justice, as the y-scale is on a small range, making them appear larger at first glance. However, in fact, the fluctuations in both loss and validation-loss are in the scale of 0.001.

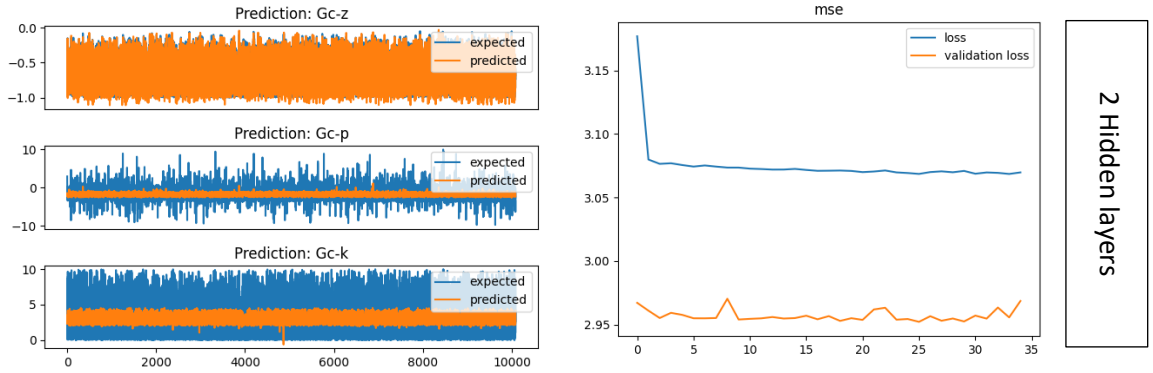
A general rule would be to take the highest number of epochs that does not result in overfitting, therefore the best epochs number is between 20 and 50. A reasonable decision would be to take an average value of the range of 20-50, rather than sticking to one end. This results in having a better training loss than the lower end, and also a better testing loss (validation) than the higher end.

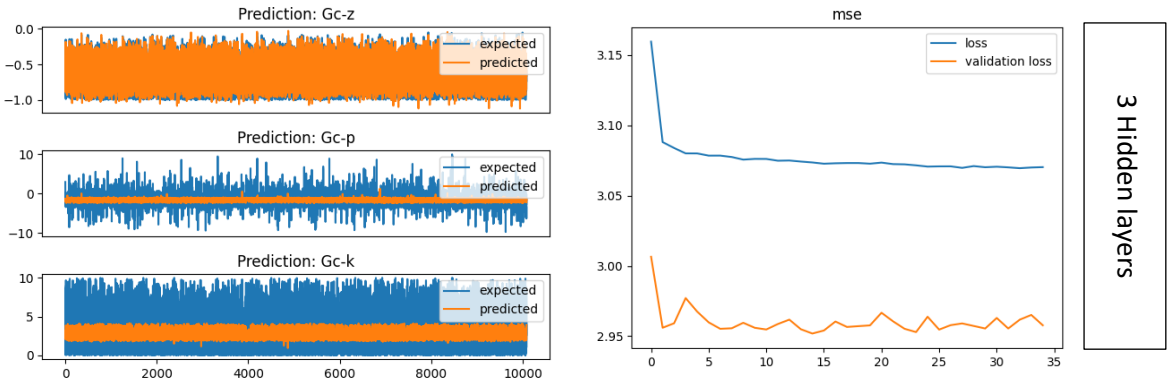
Therefore, this experiments resulted in taking 35 as the epochs number for ANN training.

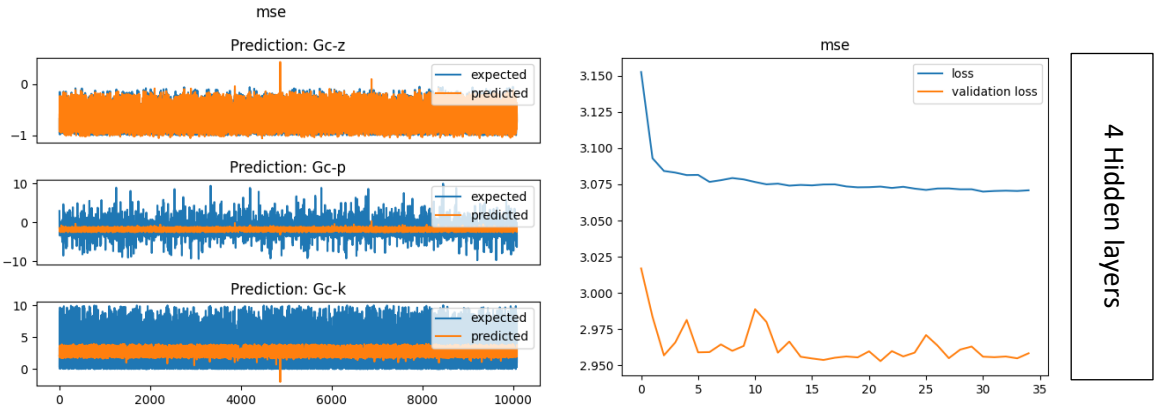
5.3. Hidden-layers Experiment

In this experiment, the neural network is going to be built with different numbers of hidden-layers. This is done to observe the effect of the number of hidden-layers on the learning process.









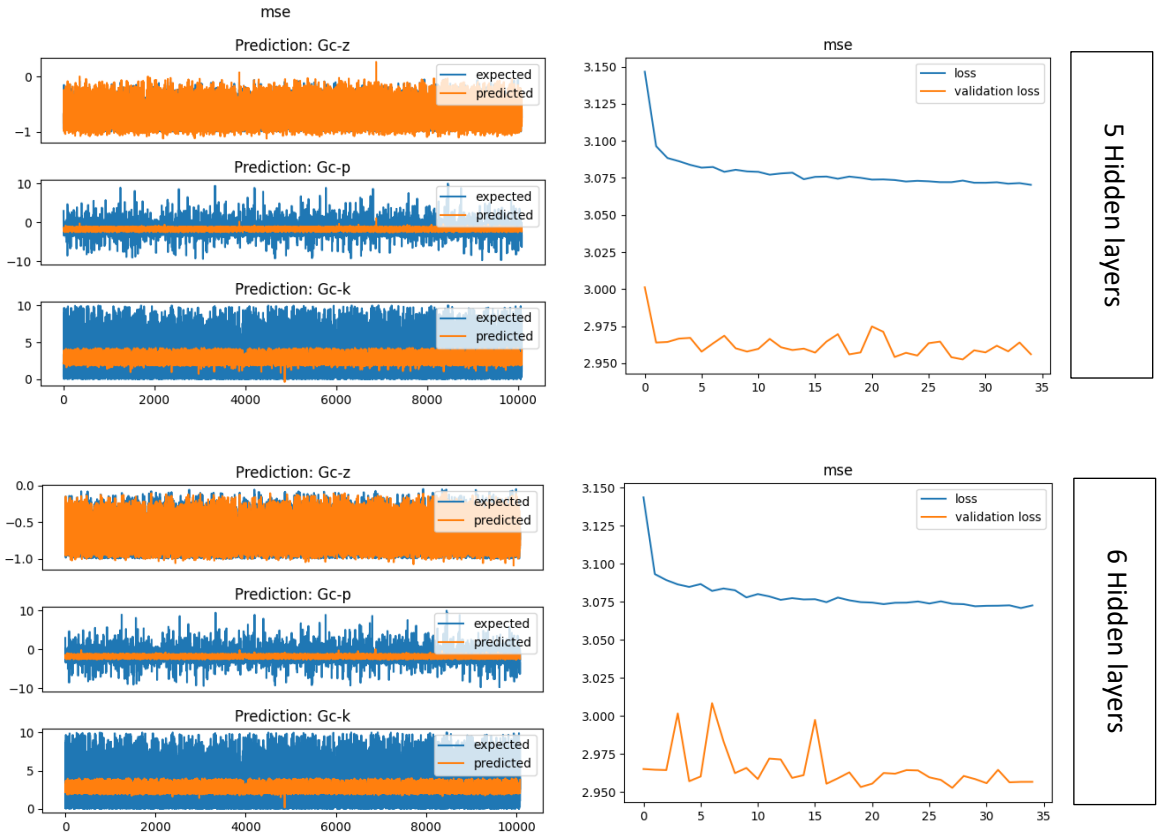


Figure: Effect of hidden layers on losses

For a clearer comparison, the total deviation of the predicted values from the expected ones was calculated for each case, and the results are as follows:

Hidden-layers	1	2	3	4	5	6
Deviation	7.402	**7.131**	7.294	7.451	7.334	7.368

Table: Effect of hidden layers on prediction accuracy

As it can be seen from the calculations, having 2 hidden layers is optimal for the learning process.

5.4. Loss Experiment

To evaluate which of the previously introduced loss functions is optimal for the ANN, this experiment was done. The program for this experiment uses the neural network with the different loss functions, and then calculates the total deviation in order to perform a clear comparison. The results are as follows:

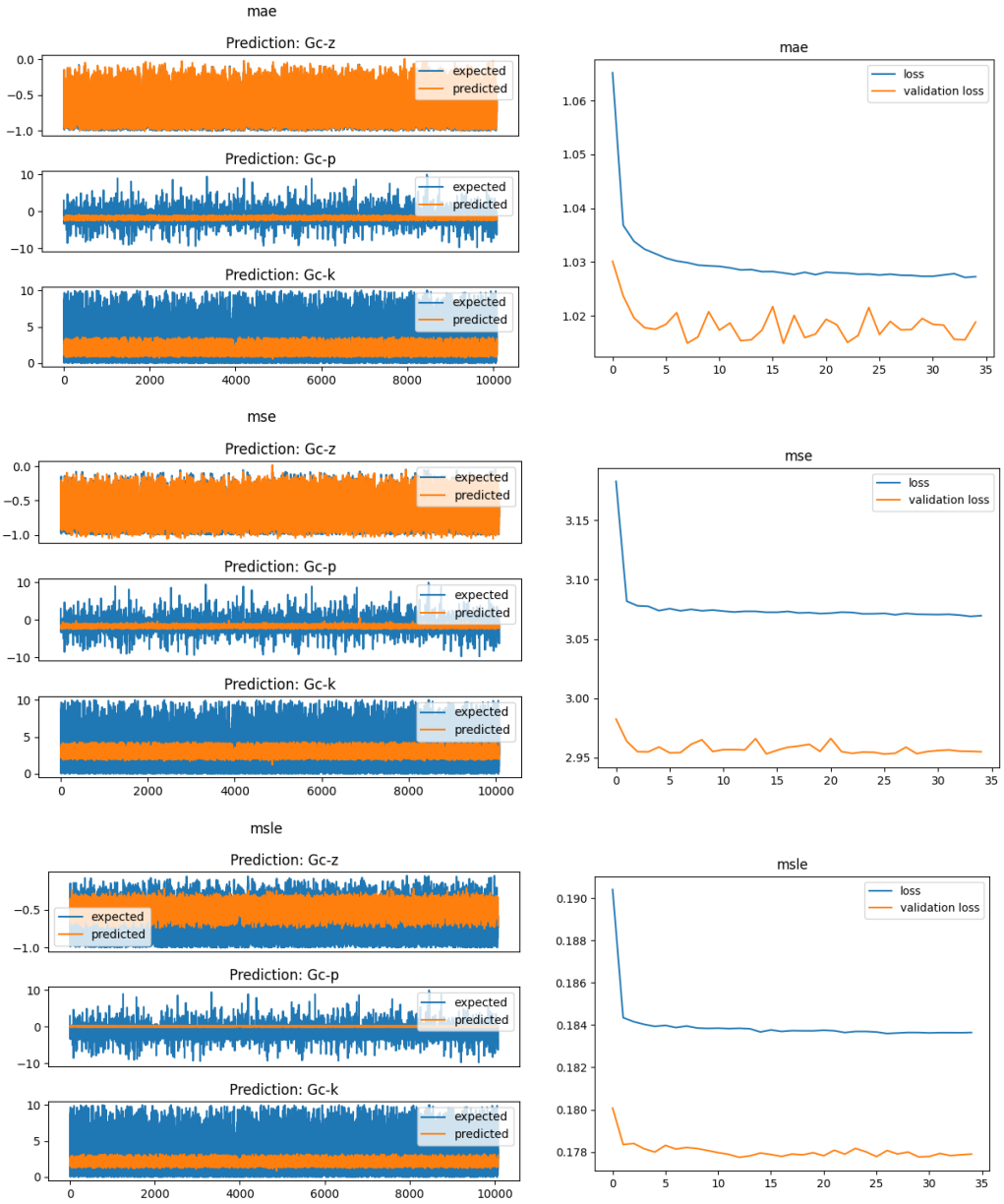


Figure: Effect of different loss functions on the prediction accuracy

As it can be seen from the plots, although the MSLE has the lowest losses, the predictions do not match their expectations as good as when the ANN uses the other two loss functions. Therefore it is safe to say that the loss values depend on the function used, and are not as relevant as the rate they drop with.

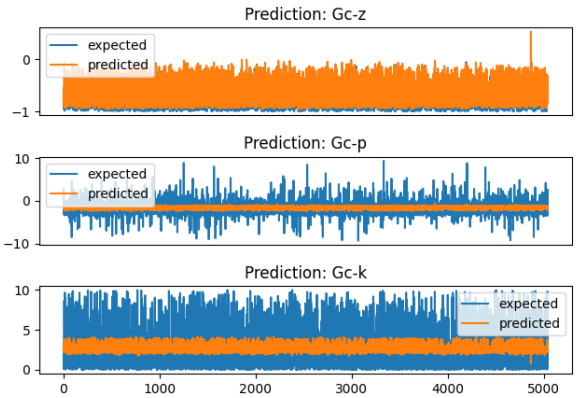
For MSE and MAE functions, there seem to be a very close similarity in results. However, the MSE function is chosen from this experiment. This is due to the fact that the MSE is the most commonly used loss function.

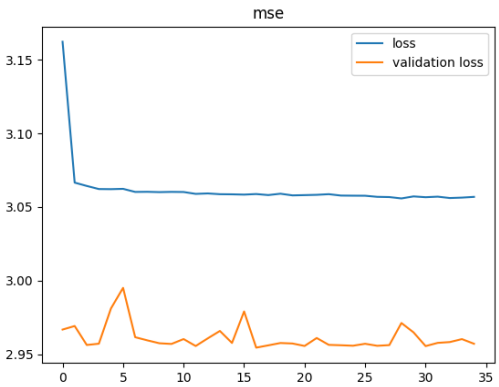
5.5. Data-split Experiment

Throughout the examples and tutorials available online and in books, there seems to be a verity of choices for the data split. This split is just the percentage of dividing the data into training and test data. This means that a split of 0.5 implies that the data is split in half between the two types mentioned. This experiment, although not seeming very crucial, can be straight-forwardly

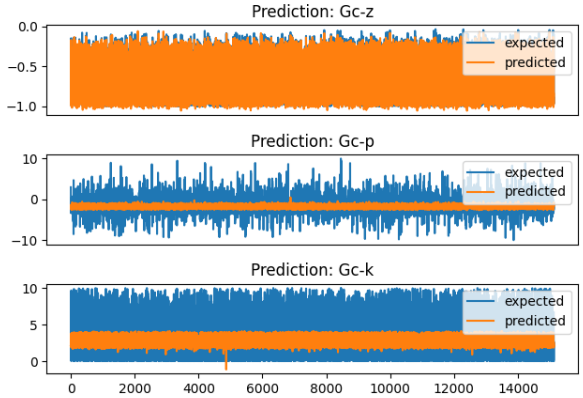
implemented, therefore offering an observation on the behavior of the neural network when different data-splits are used. The results are as follows:

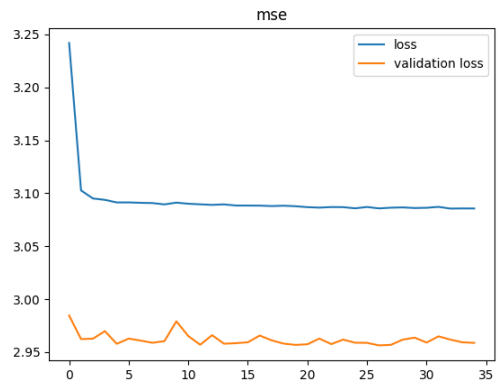
0.1



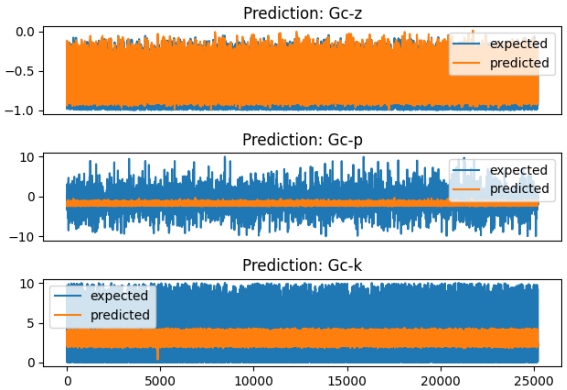


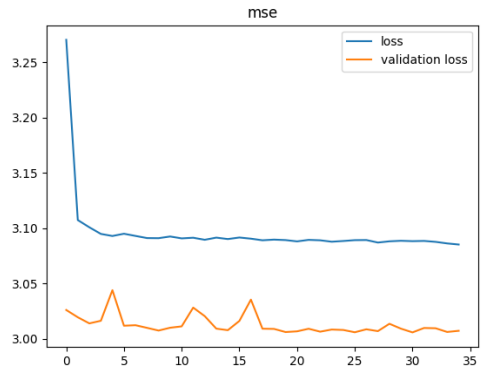
0.3



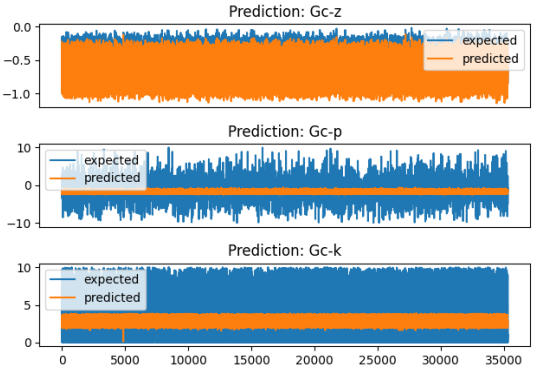


0.5





0.7



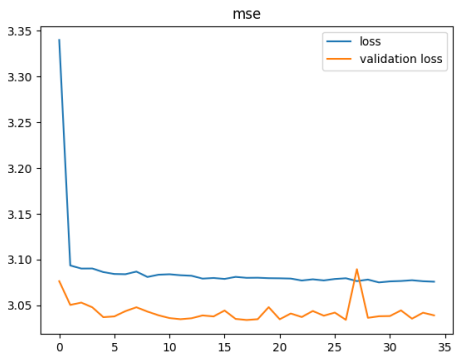


Figure: Effect of data-split on the predictions and losses

It can be seen that the losses experience the least fluctuations in the splits between 0.1 and 0.3. Conventionally, the data-split is 80% for training and 20% for testing, and this matches the average of our chosen interval. Therefore, a split of 0.2 is chosen.

6. Evaluation

The final evaluation is the use of the ANN with the tuned parameters, which are as follows:

- The data is normalized
- The epoch number is chosen to be 35
- The number of hidden layers is 2
- The loss function is MSE
- The testing split is 20% of the data

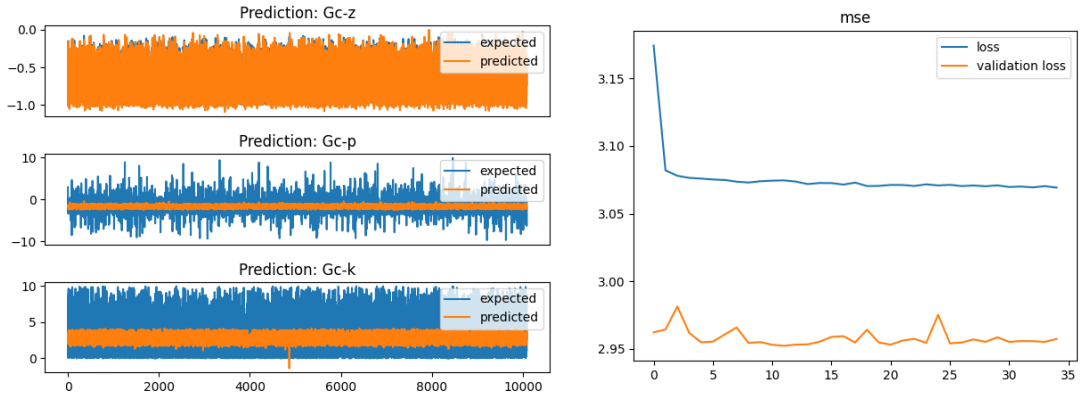


Figure: Predictions and losses of the main program

7. Conclusion & Outlooks

At the beginning of this work, there was the question “can an artificial neural network learn to predict values that engineers learn to suggest based on their experience and intuition?”. The aim was to design an ANN and train it to compensate a transfer function based on Root-Locus method. The theoretical background of the relevant topics in control theory were discussed, then the artificial neural network subject was introduced. The framework Keras was presented as it is used mainly in ANN design.

First was the problem of data gathering. A python program was created to tackle the issue by mimicking the procedure of designing a compensator, in a way that the program keeps semi-randomly looping until falling into a satisfactory result.

After gathering enough data, the following procedure was to design an appropriate ANN and optimize it for the training phase. The dense layers were chosen to build the neural network, and their parameters such as activation functions, and units, were chosen according to best practices while taking into consideration the nature of this work, i.e. regression task.

The experiments were done as an elimination or decision-making stage, in which it was decided whether it is more appropriate to standardize the dataset or to normalize it. A function was

written to execute the experiment by calculating the distribution of every column, plot it, and calculate finally the probability of that distribution falling into one of the most common data distributions. This experiment concluded in the decision of normalizing the dataset due to the fact that the most columns have a distribution different than the normal (Gaussian) distribution. The second experiment was about deciding the number of epochs that the ANN goes through in the training process. This was modulated also in a programming function that trains the ANN with different epochs number. The results led to choosing the epochs to be 35. The third experiment was about deciding the number of hidden layers appropriate for the neural network. Models with different numbers of hidden layers were built and a plot of their results was done. Finally, a comparison between their deviations was calculated, and the experiment resulted into choosing 2 hidden layers for the network for optimal results. The fourth experiment dealt with the different loss functions that can be used for regression. A simple plot of the predictions and loss rates of each function resulted into taking the MSE as the loss function. The fifth and final experiment presented the effect of having different split percentages of the data on the learning rate of the network. The split of 20% as testing data concluded the experiments section.

The last section was about evaluating the ANN by training it with the best results of the experiments and comparing its predictions to the expected values. It was seen that the tuned parameters of the data have further enhanced the quality of predictions and lessened the fluctuations of the losses. And this concludes the work done in this project.

A future outlook for this work would be experimenting with other types of compensators other than the phase-lead. In addition, the data of different orders of transfer functions. The data gathering part can be more selective to observe the effects on the accuracy of the ANN. The neural network design can be subjected to further development for more optimization. Finally, as new research is being published on having a neural network that learns from complex numbers, it is only intuitive to attempt the integration of such research in this topic as the raw data contains originally complex numbers that had to be split into different columns.

In conclusion, the results of this work can be labeled as successful, although it can be regarded only as the beginning of the development of such field.

8. References

[1] Dorf & Bishop, Modern Control Systems, 13th Edition

[2] Keras (https://keras.io/)

[3] Keras Layers (https://keras.io/api/layers/)

[4] Keras Activation Functions (https://keras.io/api/layers/activations/)

[5] Normalization VS Standardization (https://medium.com/swlh/difference-between-standardization-normalization-99be0320c1b1)