8-2023

Topology Optimization for Artificial Neural Networks

Justin Mills

TOPOLOGY OPTIMIZATION FOR ARTIFICIAL NEURAL NETWORKS

A Thesis submitted in partial fulfillment
of the requirements for the degree
Masters of Science

Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

By
Justin Mills
August, 2023

Title: Topology optimization for artificial neural networks

by: Justin Mills

Date Recommended 5/16/2023

DocuSigned by:
1EEB1FD98163449...
Chair

DocuSigned by:
D05CB49207EF4FB...
Committee Member

DocuSigned by:
4CD496AF2FDD4F9...
Committee Member

Committee Member

DocuSigned by:
8F8057903E39448...
Associate Provost for Research and Graduate Education

ABSTRACT

TOPOLOGY OPTIMIZATION FOR ARTIFICIAL NEURAL NETWORKS

This thesis examines the feasibility of implementing two simple optimization methods, namely the Weights Power method (Hagiwara, 1994) and the Tabu Search method (Gupta & Raza, 2020), within an existing framework. The study centers around the generation of artificial neural networks using these methods, assessing their performance in terms of both accuracy and the capacity to reduce components within the Artificial Neural Network's (ANN) topology.

The evaluation is conducted on three classification datasets: Air Quality (Shahane, 2021), Diabetes (Soni, 2021), and MNIST (Deng, 2012). The main performance metric used is accuracy, which measures the network's predictive capability for the classification datasets. The evaluation also considers the reduction of network components achieved by the methods as an indicator of topology optimization.

Python, along with the Scikit-learn framework, is employed to implement the two methods, while the evaluation is conducted in the cloud-based environment of Kaggle Notebooks. The evaluation results are collected and analyzed using the Pandas data analysis framework, with Microsoft Excel used for further analysis and data inspection.

The Weights Power method demonstrates superior performance on the Air Quality and MNIST datasets, whereas the Tabu Search method performs better on the Diabetes dataset. However, the Weights Power method encounters issues with local minima, leading to one of its stop conditions being triggered. On the other hand, the

Tabu Search method faces challenges with the MNIST dataset due to its predetermined limits and restricted scope of changes it can apply to the neural network.

The Weights Power method seems to have reached its optimal performance level within the current implementation and evaluation criteria, implying limited potential for future research avenues. In contrast, to enhance the dynamic nature of the Tabu Search method, further investigation is recommended. This could entail modifying the method's capability to adapt its stop conditions during runtime and incorporating a mechanism to scale the magnitude of changes made during the optimization process. By enabling the method to prioritize larger changes earlier in the process and gradually introducing smaller changes towards the conclusion, its effectiveness could be enhanced.

Keywords: Artificial Neural Networks, Feedforward Neural Network, Topology Optimization, Classification Datasets.

I dedicate this thesis to my parents, Denny and Donna Mills, and my siblings Sandra and Jessica

ACKNOWLEDGMENTS

I am deeply grateful to the Western Kentucky University Computer Science Department for their invaluable support and resources throughout my thesis journey. Their dedication to providing an exceptional educational environment greatly contributed to my academic growth.

Special thanks go to Dr. Ziegler for their unwavering support and guidance throughout the thesis process. Their expertise and understanding played a crucial role in shaping the direction of my research.

I would also like to acknowledge the esteemed faculty members and staff of the Computer Science Department for their commitment to excellence in education. Their passion for teaching has enriched my learning experience.

Furthermore, I want to express my heartfelt appreciation to my family and friends for their unwavering support and encouragement. Their belief in my abilities has been a constant source of motivation.

Lastly, I extend my sincere gratitude to the university as a whole for providing a nurturing academic environment that fosters growth and intellectual development.

The Western Kentucky University Computer Science Department's dedication and support have been invaluable in my thesis journey. It has been an honor to be a part of this exceptional institution.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

Introduction

1.1 Thesis Objective

There are many applications for Artificial Neural Networks (ANNs) across many industries, some examples being self-driving cars, virtual assistants, and automatic machine translation. Some products related to the examples would be Tesla for self-driving cars, Amazon's Alexa and Apple's Siri which are virtual assistants, and Google translate as an example of automatic machine translation (Chatterjee, 2022). ANNs are a heavily researched topic, and their popularity continues to grow. A search on Google Scholar using the search term "Artificial Neural Network" yielded 80,200 hits for the year 2022 alone. Impressively, there have already been 39,100 publications on this subject since the start of 2023.

This thesis will focus on the optimization of the topology of ANN. Topology optimization aims to minimize the number of hidden layers used in an ANN, as well as reducing their corresponding number of neurons and weights, while maximizing the accuracy and performance of the ANN.

There are many advantages to ANN optimization, as it can lead to smaller ANNs by reducing the number of components involved, assuming the performance of the ANN is acceptable. A smaller ANN offers several advantages over a larger one. Firstly, it tends to have better generalization ability since it limits the number of units and weights, preventing overfitting. Additionally, a smaller network is computationally efficient, making it more cost-effective in terms of both learning and practical use. Lastly, there is hope

that a smaller trained network will exhibit behavior that can be described by a simple set of rules, allowing for easier interpretation (Hagiwara, 1994).

Experts in many application areas that use available ANNs often follow generally accepted rules of thumb for building the ANN topology for their work. These rules are explained as when determining the number of hidden neurons in an ANN, there are some general guidelines to follow. Firstly, it is recommended that the number of hidden neurons falls within the range of the input layer size and the output layer size. Another rule suggests that the number of hidden neurons should be approximately 2/3 of the sum of the number of inputs and outputs. Additionally, it is advisable to keep the number of hidden neurons below twice the size of the input layer. These guidelines can help determine an appropriate number of hidden neurons for an ANN (Vujičić et al., 2016).

ANNs are not trained solely to perform well on the given training data; they are trained to learn relationships inherent in the data. If the training data is a representative sample of the overall data for the task, the ANN can generalize and perform well on data it has not encountered before with reasonable accuracy. Improving computational efficiency is a significant advantage of ANN optimization. A reduction in computational cost leads to a decrease in time, which is valuable since time is money, and it also allows for the use of ANNs in small, low-powered devices. While it may be possible to interpret how an ANN makes its predictions, due to its black box nature, this is not always the case and should be considered a bonus if it is possible.

Two methods of ANN topology optimization have been selected that can be understood by non-experts and have similar approaches. These methods aim to find a

reasonably good ANN topology and are from different time periods. The performance of the optimized ANNs using these methods will be compared based on accuracy and other performance metrics for several classification tasks.

The questions that my thesis will answer are:

1. Is it feasible to implement the two simple optimization methods Weights Power (Hagiwara, 1994) & Tabu Search (Gupta & Raza, 2020) using an existing framework. This is important because non-experts typically use ANN frameworks to implement ANN solutions.

2. How do the two selected methods compare to each other for the datasets used, Air Quality (Shahane, 2021), Diabetes (Soni, 2021), and MNIST (Dato-on, 2021). The comparison will cover the performance of each method for the datasets and how many components the method optimized ANNs contains.

1.2 Thesis Outline

Chapter 2 of the thesis covers two main topics: an introduction to ANNs and a detailed description of the two implemented methods. The introduction will cover the essential components of ANN topology such as neurons and network layers, the process of training an ANN, and the performance metrics that will be used in this thesis, along with detailed descriptions of how these metrics function. The description of the two methods will include pseudocode along with a summary of how each method works based on the published record.

Chapter 3 covers the implementation of the two methods such as what predetermined values that are required by the method are and why they were selected. The details for each step are also covered in more depth and highlight any major problems I encountered in their implementation and how they were solved. This is followed by the background information for the evaluation setup, such as what programming language was used along with any frameworks or tools used and why. This section will also cover the datasets used, how they were preprocessed, and why they were selected.

Chapter 4 contains the results from the conducted evaluation. An overview that discusses the quality of the ANNs created and the datasets used for the evaluations. Each result is broken down by the dataset the methods are compared on, and these results cover how well the methods performed for the datasets used. The performance of the methods are compared to one another and which method performed better for each dataset.

Chapter 5 covers the conclusions that were reached from the evaluation results for each of the methods during the completion of the thesis. Following that is any future work that could be continued using this thesis as a starting point.

Chapter 2

Background

This chapter will cover background information and is split into two major sections: an introduction to ANNs and a description of the ANN optimization methods covered in this thesis. The ANN introduction will focus on feed-forward ANNs, providing a detailed description of their topology, including neurons and layers, and the process for the ANN to learn and make predictions. This section will also explain the process of measuring the performance of an ANN. The description of the optimization methods will include the algorithms of both the Weights Power (Hagiwara, 1994) and Tabu Search (Gupta & Raza, 2020) methods.

2.1 Introduction to ANNs

ANNs attempt to mimic the basic functions of the human brain using software. There are different approaches to constructing an ANN, and various ways for them to learn from a given dataset. Topology in the context of artificial neural networks refers to the specific pattern or structure through which individual neurons are interconnected. It encompasses the arrangement and connections between neurons, determining how information flows within the network (Suzuki, 2011). The methods explored in this thesis use a feed-forward artificial neural network architecture, which has the restriction that the network needs to establish a one-way flow of information from the input to the output. This ensures that information is processed and transmitted linearly through the network, without the possibility of circulating back to previous layers or neurons. (Suzuki, 2011). This flow of information in regards to how the ANN makes predictions is

known as feed-forward. Other architectures are outside the scope of this thesis and are not covered.

The type of information contained in the dataset determines how the ANN will learn. Three fundamental approaches to learning that are widely recognized are supervised learning, unsupervised learning, and reinforcement learning (Suzuki, 2011). Supervised learning involves datasets where each sample is labeled with known output values and is often used for classification tasks. The ANN is trained to associate a given input with the corresponding label. During training, this label is often referred to as the "target output". For classification tasks, the label represents the class to which the input belongs. This is the type of learning used in this thesis.

Unsupervised learning uses unlabeled datasets with unknown output values, and it may be used for the ANN to learn how to categorize the data into clusters. The ANN is trained to detect the clusters within the data. Other than an upper limit on the number of clusters, there are usually no restrictions on which inputs belong to which cluster.

Reinforcement learning, also known as semi-supervised learning, acts as a middle ground between supervised and unsupervised learning and is used when the dataset contains samples that are both labeled and unlabeled.

2.1.1 Feed-forward ANN Topology

There are two major components involved in the construction of a feed-forward neural network. The first component, called a ‘neuron,’ includes weights, biases, and an activation function, which together act as a way to generate the neuron’s outputs from its inputs. The second component, called a network ‘layer,’ acts as a way to group

neurons. There are three types of layers: input, output, and hidden layers, each having different functions within the network. Typically, there is only one input layer and one output layer, while the number of hidden layers can vary depending on the problem being solved.

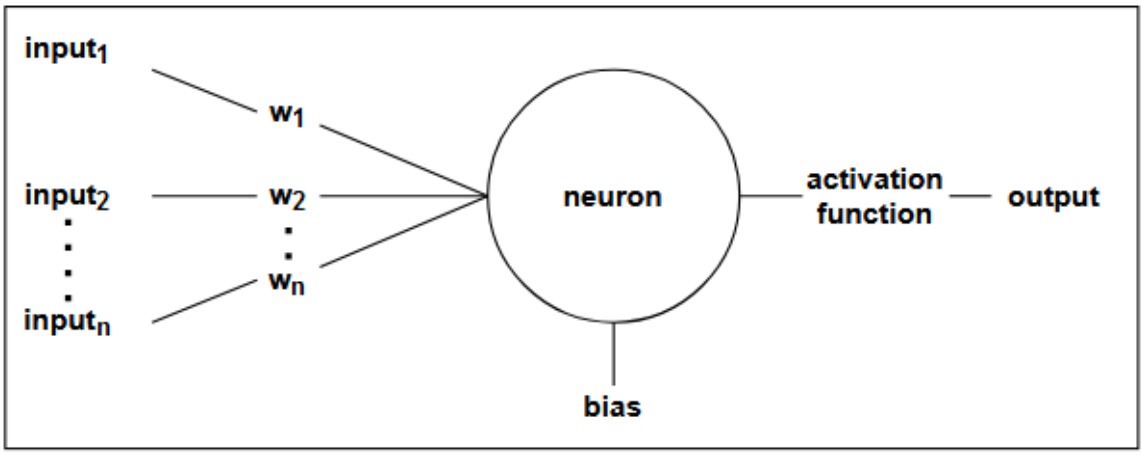

Figure 1: Basic Neuron Topology

An example of a neuron in an ANN is illustrated in **Figure 1**. Information flows from left to right. The weights connect each input to the neuron, and the weight values indicate the contribution of each input value. The bias allows the neuron to have a non-zero output, even if all input values are zero. The total activation reaching a neuron is the sum of the products of each input value and its corresponding weight, plus the bias. The activation function then transforms the total activation into the neuron's output. The process for calculating a neuron's output based on its inputs is shown below in **Formula (1)**.

$$output = activation\ function(bias + \sum_{i=1}^{n} input_i weight_i) \quad (1)$$

It should be noted that a neuron's output value may be passed as input to more than one neuron in the next network layer. The value passed to each of those neurons is the computed output value. It is also important to note that different weights and biases can affect the output value, even if the input stays the same. This means that learning in an ANN involves adjusting the weights and biases.

There are many types of activation functions that are used in ANNs, but the two that are included in the ANN used for the evaluation conducted in this paper are the RELU (rectified linear unit function) and Softmax function in.

$$RELU(x) = max(0, x) \quad (2)$$

$$Softmax(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad (3)$$

The RELU function limits the possible outputs of a hidden layer neuron to positive values and 0. By using 0 as the lower bound, the effect of the neuron's output on the input of the next layer's neurons that map to that output is nullified. This can be observed in **Formula (1)**, as regardless of the associated weight value for the input, the output will always be 0. Another activation function, the Softmax function is employed in the output layer of a classification ANN, constraining the final predictions to values between 0 and 1. It is important to note that when an ANN has multiple potential outputs, the sum of all predictions adds up to 1, allowing for the selection of the largest prediction value as the ANN's prediction for a given data sample.

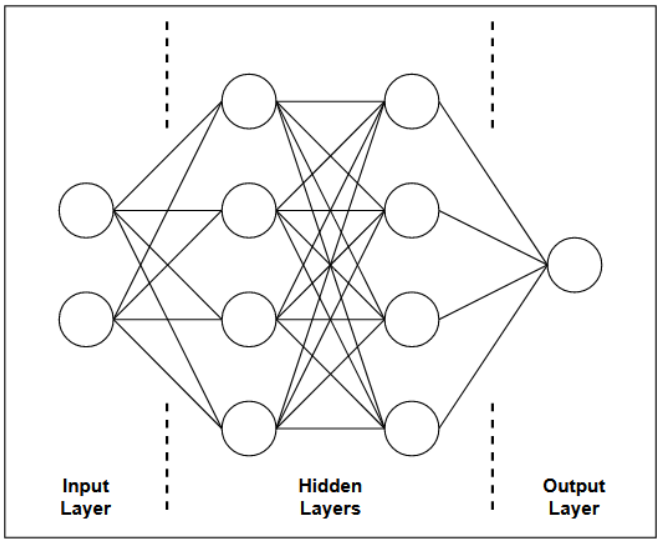

Figure 2: Basic Feed-forward ANN Topology

The data used for tasks in feed-forward ANNs consists of many samples. Each sample contains a fixed number of input values, and in the case of training data, includes the label that the network is intended to learn. The input layer serves as the entry point for the sample data from a dataset, with each data point in the sample having its own input neuron. While input neurons are referred to as neurons, they do not function as such and contain only an output value to pass the input into the hidden layer(s). The hidden layer(s) process the input data and send the processed information to the output layer. For a classification ANN, the number of output neurons is equal to the number of classes, which are the possible output solutions for a given dataset. The activation function of the neurons in the hidden layer(s) for the ANN used in this paper is the ReLU function.

In a classification task, it is customary to translate the output computed by the last hidden layer into something similar to probabilities for each class. This is done using the softmax function, which uses the outputs of all the neurons in the layer. The

output layer neurons also use the softmax function as their activation function. Each neuron in each layer is connected to all the neurons in the previous and next layers, as shown in **Figure 2**. Therefore, the number of outputs of a neuron in a given layer is equal to the number of neurons in the next layer, with each output connected to the neurons in the next layer on a one-to-one basis.

2.1.2 Predictions of a Feed-forward ANN

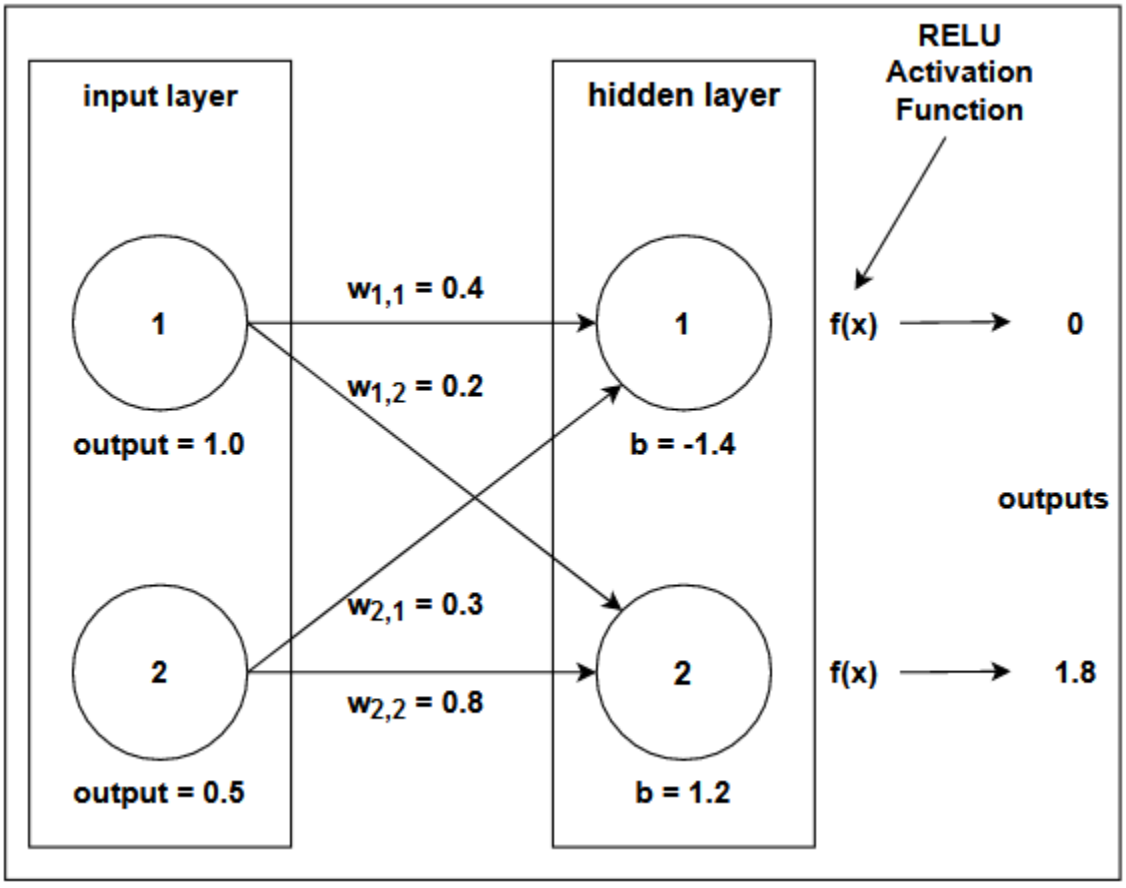

Figure 3: Feed-forward Graphical Representation

An ANN uses its weights and biases to predict an output for a given set of input values. The term 'feedforward' refers to the flow of information from input to output. As

data passes through each layer, the output of each layer is calculated by its associated neurons. Each neuron calculates its individual output using **Formula (1)**. **Figure 3** provides a graphical representation of the feedforward process from the input layer to the output of the hidden layer. The output values of the input layer serve as the inputs of the neurons in the hidden layer. The output of hidden layer neuron 1 is calculated as the RELU of the sum of its weighted inputs plus bias, that is, *RELU(-1.4 + 1.0 * 0.4 + 0.5 * 0.3) = RELU(-0.85)*, which returns an output of *0*. Similarly, the output of hidden layer neuron 2 is calculated using the same method, which is *RELU(1.2 + 1.0 * 0.2 + 0.5 * 0.8) = RELU(1.8)*, returning an output of *1.8*.

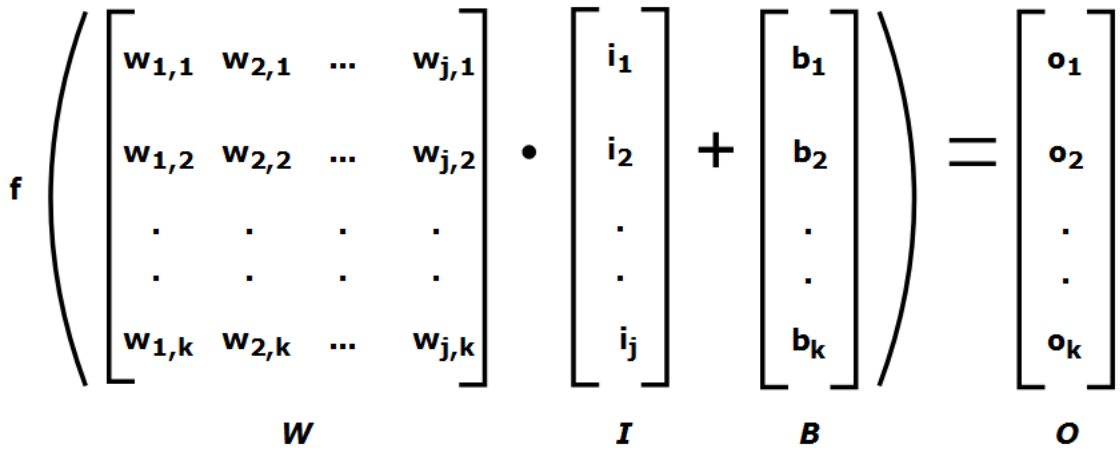

Figure 4: Feedforward Matrix Calculation

As shown in **Figure 4**, calculations for an ANN can be conducted using matrices and vectors. In **Figure 4**, *j* represents the number of output values from the previous layer's neurons, which are combined into the input vector *I* for the hidden layer. The number of neurons in the hidden layer is represented by *k*. Row *k* of the matrix *W* contains the weights w_p for $1 \leq p \leq j$ for hidden layer neuron *k*. *B* is the vector of bias

values b_q for $1 \leq q \leq k$ for each of the k neurons in the hidden layer. Multiplying the matrix W with the input vector I results in a vector S with k elements, representing the sum of weighted inputs for each hidden neuron. Adding the bias vector B and then applying the activation function leads to the output vector for the hidden layer neurons. The result is then passed through the RELU activation function, which leads to the output vector.

2.1.3 Training a Feed-forward ANN

When conducting training for a feed-forward ANN, the available dataset needs to be split into two parts: a training dataset and a testing dataset. This split is commonly an 80/20 split, with 80% of the dataset being used for the training and 20% for the testing. During the training phase, the ANN learns a given task from the training dataset, while the testing dataset is used after training to assess the generalization of the ANN on data it has not been exposed to before.

To avoid training issues, it is important to prevent overfitting or underfitting. Overfitting occurs when an ANN trains too well on the training dataset that it becomes unable to generalize from the given data, which means its prediction performance on unseen data suffers. This can occur when the ANN's topology is too complex and/or the training is not stopped when the ANN's generalization capabilities start to deteriorate. Underfitting occurs when an ANN is unable to correctly predict outputs even for the training data. This normally happens when the ANN is trained on too little data and/or if the ANN's topology is too simple.

The goal of training a feed-forward ANN for a classification task is to minimize the error rate, which is the number of wrong predictions. Reducing the error rate generally means the network makes more correct predictions, which increases an ANN's performance. As seen from **Figure 4** and the related example, an ANN's output is a function of the ANN's weights, biases, and the sample values.

$$ANN\ predicted\ output\ =\ feedforward(input,\ weights,\ biases) \quad (4)$$

Supervised feed-forward ANN training uses the error of the ANN's predictions, which is based on the difference between the target output versus the predicted output. In other words, the error is a function of the target output, weights, biases, and inputs.

$$error\ =\ difference(input,\ weights,\ biases,\ target\ output) \quad (5)$$

The error function is used to modify the weights and biases of each neuron in the ANN. One approach to minimize the error is to use gradient descent. In gradient descent, the slope of the error is estimated with respect to each weight and bias, and the direction and magnitude of the slope are used to determine how much to update each weight or bias to reduce the error slightly. Repeating this many times often leads to a smaller error. To reduce the likelihood of getting stuck in local minima, a momentum term is often added to the computed weight or bias update.

The gradients for the error surface are first computed for the output layer and then for previous layers, one layer at a time. Thus, the computation of weight updates progresses through the network in the opposite direction of the feed-forward predictions, leading to the name 'backpropagation'. The backpropagation process is resource-intensive and responsible for the majority of the computational costs in the

training of an ANN. To mitigate this, training is normally done in batches, where multiple predictions are made over several data samples, and the mean average of the errors of the training batch is used for backpropagation instead of conducting the process for each sample. This is called *stochastic gradient descent*.

There are various error functions that are used for ANNs for different types of tasks. The *log loss function*, calculates the error for both single and multiclass outputs, and used for the backpropagation error in the thesis. The formula is shown below.

$$L_{log}(y,p) = -\frac{1}{N}\sum_{i=0}^{N-1}\sum_{k=0}^{K-1} y_{i,k} log(p_{i,k}) \quad (6)$$

In the above formula, N is the total number of samples in the dataset and K is the number of classes. Furthermore, $p_{i,k}$ is the value predicted by the ANN for the *k-th* output value for the *i-th* sample in the case of datasets with multi outputs; and $y_{i,k}$ is the *k-th* value of the target output for the *i-th* sample from the dataset *y*. This L_{log} is calculated for all N samples in the dataset and returns the negative mean average of the summation. Since the $p_{i,k}$ are between 0 and 1, resulting in a log value less than one, the overall error computed is always positive.

2.1.4 Measurements of ANN Performance

The measurements of performance for an ANN in a classification task can be performed for both the testing and training dataset and involves the relationship between the ANN's predicted output of a sample versus the target output. Measuring the performance on the testing dataset versus the training dataset is useful in

determining if the ANN is under or over fitted and comparing the performance on the 2 datasets shows how generalized the ANN is. This relationship can be broken down into the four types as shown in **Table 1.** Positive indicates the output of a sample would return true, while a negative would be a return of false. For example if a sample contains information about a student's study habits for a final correlating to if that student passed the final or not, an output that the student passed would be considered positive while an output that the student failed would be negative.

		Predicted	
		Negative	Positive
Actual	Negative	TN (True Negative)	FP (False Positive)
	Positive	FN (False Negative)	TP (True Positive)

Table 1: Confusion Matrix

A TN occurs when the ANN correctly predicts a negative value, while a TP is a correct prediction of a positive value. A FN occurs when the ANN wrongly predicts a negative value, while FP is a wrong prediction of a positive value. For example if the true output of samples from a classification dataset is {False, True, False}, and the ANN outputs is {False, False, True}, then the 3 predicted outputs from left to right are a TN, a FN, and a FP. For an entire dataset, TN stands for the total number of samples which are categorized as TN and similar for FN, TP, and FP. These counts of prediction outputs are used to calculate the performance metrics seen in **Table 2**.

Metrics	Formula	Description
Accuracy	$\frac{(TP + TN)}{(TP + FP + FN + TN)}$	Represents the ratio of correctly predicted outputs to the total number of predictions In the case of multiple output values the subset accuracy is taken, where the entire set of predicted values for a sample is compared to its set of true values with the result of a 1 if the sets are an exact match otherwise the result is 0 Worst value: 0 Best value: 1
Precision	$\frac{TP}{TP + FP}$	Represents the ratio of correctly predicted positive values to the total number of (correctly and incorrectly) predicted positive values Intuitively - the value captures how much the ANN can be trusted not to provide false positive predictions In the case of multiple output values the precision value is calculated for each value in the set of outputs values with their resulting values being

		averaged Worst value: 0 Best value: 1
Recall	$\frac{TP}{TP + FN}$	Represents the ratio of correctly predicted positives values to the total number of actual positive values Intuitively - the value captures the fraction of actual positives correctly predicted. In the case of multiple output values the recall value is calculated for each value in the set of outputs values with their resulting values being averaged Worst value: 0 Best value: 1
F1 Score	$\frac{2 * precision * recall}{precision + recall}$	Represents the equilibrium between precision and recall Worst value: 0 Best value: 1

Error	$1 - accuracy$	Represents the ratio of incorrectly predicted outputs to the total number of predictions In the case of multiple output values the subset error is taken, where the entire set of predicted values for a sample is compared to its set of true values with the result of a 1 if the sets are an exact match otherwise the result is 0 Worst value: 1 Best value: 0

Table 2: Performance Metrics

The difference between precision and recall is important, although it may not be intuitive. For example, consider a classification dataset that distinguishes whether a hospital patient has cancer or not. A high precision value indicates that the ANN correctly predicts when a patient has cancer most of the time, while not often predicting that a patient has cancer when they actually don't. However, a high precision value does not necessarily mean that most people who have cancer are correctly identified as such, as FN are not considered for precision. On the other hand, a low precision value would be the reverse, with more patients being predicted to have cancer who don't actually have it.

Similarly, a high recall value indicates that the ANN correctly predicts when a patient has cancer, and it also shows how many patients with cancer were not predicted

to have it. However, a high recall value does not mean that most of the people who are identified as having cancer actually have cancer, since FP are not considered for recall. A low recall value would indicate that many patients with cancer were not predicted to have it.

2.2 Weights Power Method

The Weights Power method generates an optimized ANN with a predetermined level of performance. The method determines the exact ANN topology and the values of all the weights and biases in the network. The method builds an ANN that can learn a classification task for a given dataset by removing components from the ANN that adds little in the completion of the task. It obtains this ANN by first training an initial network while increasing the number of neurons in the hidden layers as needed to reach the required performance level. This base initial network is optimized with an iterative approach of removing neurons followed by an iterative process to remove weights while maintaining the required performance level. These removed neurons and weights are the least important in determining the network's predicted outputs.

2.2.1 Train Initial ANN

Train Initial ANN: Weights Power Method Pseudocode

Input: ANN, data_set
Output: ANN

```
# trains ANN on training dataset
ANN = train ANN (data_set["train"])
reaches_level = reaches_performances_level( ANN, data_set["train"])

while reaches_level is False
        # adds a neuron to each hidden layer
        ANN = add_neurons (ANN)
        # trains ANN on training dataset
        ANN = train ANN (data_set["train"])
        reaches_level = reaches_performances_level (ANN, data_set["train"])

return ANN
```

Algorithm 1: Trains ANN To Required Performance Level

Algorithm 1 first trains an initial ANN so that the method has a base ANN to work from. This initial ANN is created by specifying a predetermined number of hidden layers and a predetermined number of neurons for each of the ANN's hidden layers. Each of the hidden layers for the initial ANN contains the same number of neurons. The ANN is trained on the given training dataset. After the ANN training completes, the method verifies whether it reaches the specified performance level. If it does then the ANN is returned to be used as the base ANN for the remaining steps of the method. Otherwise the method repeatedly adds a neuron to each hidden layer and retrains the ANN until the minimum performance level is reached. This generates an ANN that has learned the given classification task to the specified performance level. This may have

caused the ANN's topology to grow larger than is necessary during **Algorithm 1**. The remaining steps in the method decrease the topology of the ANN.

Reaches Performances Level: Weights Power Method Pseudocode

Input: **ANN, data_set**
Output: **reaches_level**

```
# maximum training error
training_criterion = 0.2
# calculates training error for ANN
training_error = calculate_error (ANN, data_set["train"])

if training_error < training_criterion
        # ANN has reached correct performance level
        return True
else
        # ANN has not reached correct performance level
        return False
```

Algorithm 2: Checks If ANN Reaches Performances Level

The performance level indicated by the training criterion of **Algorithm 2** is predetermined by specifying a maximum error value for the training dataset. The performance of the trained ANN is determined by calculating the mean absolute error for the ANN, using the formula shown below. If this error value is lower than the predetermined levels for the training dataset then the ANN has reached the required performance level and the method returns true, else it returns false.

$$MAE(\hat{y}, y) = \frac{1}{N} \sum_{i=o}^{N-1} |\hat{y}_i - y_i| \quad (5)$$

Where $\hat{y}_i$ is the predicted value determined by the ANN for the i^{th} sample from the dataset, the y_i is the correct value for the sample, and N is the total number of samples contained in the dataset. This mean absolute error is calculated for all samples N in the training dataset and returns the mean average of the summation. For datasets with multi outputs the absolute error for each output is calculated with the sum of the output error's is averaged by the number of outputs.

2.2.2 Removal of Least Important Neurons

Removal of Least Important Neurons: Weights Power Method Pseudocode

Input: ANN, data_set
Output: ANN

```
reaches_level = True
# holds a saved copy of ANN
saved_ANN = None

while reaches_level is True
        saved_ANN = ANN
        least_neuron = search ANN for least important neuron
        ANN = remove least_neuron from ANN
        reaches_level = reaches_performances_level (ANN, data_set["train"])

return saved_ANN
```

Algorithm 3: Finds And Removes Least Important Neurons

Algorithm 3 takes the ANN trained to the specified performance level during the previous algorithm and iteratively removes the *least_neuron* from its hidden layers. The *least_neuron* is the least important neuron found by iterating over all the neurons in the hidden layers and calculating the importance of each neuron. The importance of the

neuron is determined based on the weights leading to and from the neuron's two adjacent layers. These weights are squared to avoid positive and negative weights from canceling each other when summing the squared weights to compute the importance of the neuron. The smaller the importance value of the neuron the smaller the impact the neuron has on the ANN prediction. The formula for calculating the summed squared weights of the i^{th} neuron of the k^{th} layer is given in **Formula (8)** and is illustrated in **Figure 5**.

$$W_i^k = \sum_m (w_{mi}^{k-1})^2 + \sum_n (w_{in}^k)^2 \quad (8)$$

The m iterates over the weights connecting from the $(k-1)^{st}$ layer to the i^{th} neuron. The n iterates over the weights connecting to the $(k+1)^{st}$ layer from the i^{th} neuron. **Figure 5** is a graphical representation for where the weights used to calculate a neuron's importance are located in respect to the neuron.

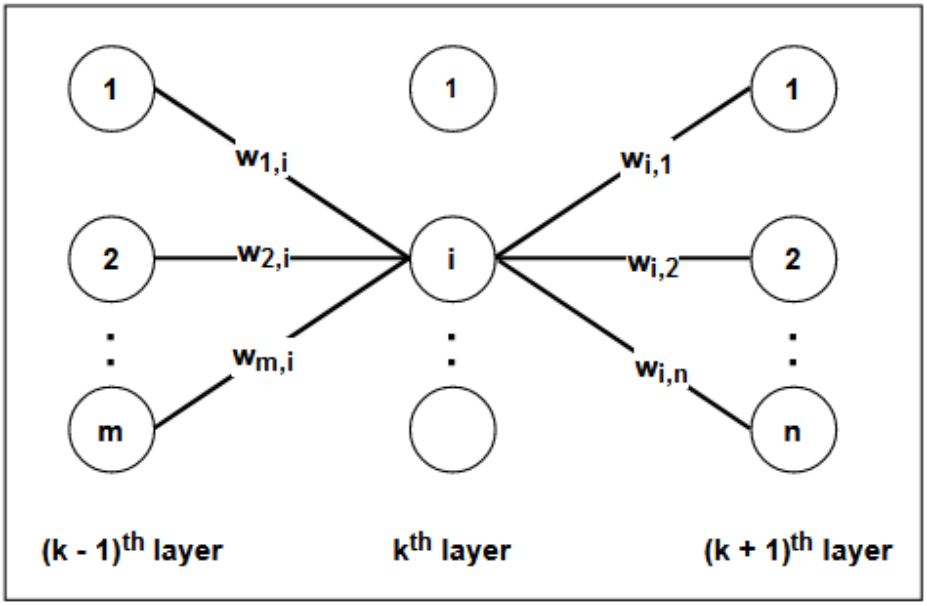

Figure 5: Least Neuron Formula Graphical Representation

The current ANN is set to *saved_ANN* by the algorithm to be used if the modified ANN fails to reach the specified performance level. The algorithm searches for the *least_neuron* and removes it, along with all weights leading to and from the neuron. After the removal of the *least_neuron*, the ANN is retrained. If the ANN still achieves the specified performance level after the neuron removal, the algorithm sets this retrained ANN to *saved_ANN*, and then repeats the process of finding and removing the *least_neuron*. This process continues until the retrained network fails to meet the required performance level. At that point *saved_ANN* is returned.

2.2.3 Removal of Least Important Weights

Algorithm 4 iteratively finds and removes the *least_weight* from the ANN's hidden layers until the ANN fails to reach the specified performance level. The *least_weight* which is the least important that determined by the weight with the smallest associated magnitude. A weight's magnitude is its absolute value.

Removal of Least Important Weights: Weights Power Method Pseudocode

Input: ANN, data_set
Output: ANN

```
reaches_level = True
# holds a saved copy of ANN
saved_ANN = None

while reaches_level isTrue
        saved_ANN = ANN
        least_weight = search ANN for least important weight
        ANN = remove least_weight from ANN
        reaches_level = reaches_performances_level (ANN, data_set["train"])

return saved_ANN
```

Algorithm 4: Removes Least Important Weights From ANN

Like the **Algorithm 3** the saved_ANN which is the current ANN that is saved by **Algorithm 4** to revert to if the modified ANN fails to reach the specified performance level. The algorithm searches for the *least_weight* and removes it before retraining the ANN. While the ANN continues to reach the specified performance level the updated ANN set to the *saved_ANN* and the next *least_weight* is removed. This process stops when the ANN fails to reach the specified performance level at which point it reverts to the ANN contained by *saved_ANN*. This restored ANN is returned as the method's optimized ANN topology.

2.3 Tabu Search Method

The Tabu Search algorithm searches through various network topologies, within the scope given by the search parameters, to find the topology that leads to the best

performance. Unlike the Weights Power method, there is no predetermined performance level to be reached, and the algorithm does not keep track of weights. Instead, it only keeps track of the topology of the ANN it creates, which includes the number of hidden layers and the associated neurons contained by the ANN. The number of hidden layers is restricted by an upper limit, and the number of neurons that can be contained in each hidden layer is bounded by an upper and lower limit. The main idea behind the method is that it randomly generates ANNs, starting with a single hidden layer containing a random number of neurons within the upper and lower bounds. This generated solution is explained in detail in section **2.3.1** and is considered the current best performing ANN topology, referred to as the *local_best*.

This *local_best* is used to create multiple *candidate_solution*, which are variations of the *local_best* topology, with the best performing *candidate_solution* becoming the new *local_best*. The method's process of finding new *local_best* solutions is done for a set number of iterations given by the *iteration_count* parameter. After completing the final iteration, the method increases the number of hidden layers and begins the process described above again. While the method conducts the above process, it keeps track of the best performing *local_best* by storing it in the *global_best* variable, which is the ANN that is returned upon the completion of the method. This process of generating solutions is covered in section **2.3.2**.

The muiltple *candidate_solution* that are created are based on the current *local_best*, as stated above, with the number of neurons increasing or decreasing for each of its current hidden layers. This possibility is determined by the *p* value, which is the percentage chance of change while the amount of change is set by the *K* value, a

percentage of the currently held neurons of the hidden layer. The number of *candidate_solution* created is determined by the *Pmax* value, with half of the *candidate_solution* being created increasing the number of neurons while the other half decreasing the amount. The creation of the mutiple *candidate_solution* is described in detail in section **2.3.3**. When selecting the best candidate that will become the new *local_best* each *candidate_solution* has it topology compared to the *tabu_list*, a list of previously generated topologies. If the topology is found it is restricted from becoming the *local_best* unless its performance is better then the current *global_best*. This process of restricting or making 'taboo' is where the method receives its name and is ment to keep the method from repeatedly checking the same topologies.

2.3.1 Solution Object

Solution Object: Tabu Search Method Pseudocode

Input: n_list[], tabu_tenure, data_set
Initialize: fitness value

```
create_ANN ()
        # trains ANN on training dataset
        ANN = train ANN(data_set['train'])
        generate_metrics (network, data_set['test'])

generate_metrics (ANN, data_set['test']))
        # fitness calculated on testing dataset
        fitness_value = calculate_fitness (ANN, data_set['test']))
```

Algorithm 5: Object For Solution Creation

The concept of a *solution_object*, as represented in **Algorithm 5**, refers to a potential ANN topology to be evaluated during the search process. It includes information on the ANN's topology, as well as its performance metrics post-training. Additionally, it has methods for creating and training an ANN based on the topology

information, as well as for computing the performance metrics. The topology information comprises the number of hidden layers and their respective neuron counts, which are stored in the *n_list*, as well as the *tabu_tenure*, which specifies the number of iterations until a *solution_object* topology is removed from the *tabu_list*. The performance of the ANN after training is saved as its *fitness_value*, which is the accuracy score specified in **Table 3** for the trained ANN over the testing dataset.

2.3.2 Generate Solutions

Generate Method Optimized ANN: Tabu Search Method Pseudocode

Input: n_inputs, n_outputs, max_hidden_layer, iteration_count, data_set
Output: global_best

```
# Note: ← indicates an action being preformed on the given list
#       example: list ← action to be preformed

# contains the global best solution
global_best = None

for current_layers in range 1 to max_hidden_layer
        # initialize tabu list
        tabu_list[] = None
        # avoids short term cycles
        tabu_tenure = 4
        # creates ANN toplogy
        n_list[] = fill_neuron_list (current_layers)
        current_solution = solution_object (n_list[], tabu_tenure, data_set)
        # append current solution to tabu list
        tabu_list[] ← append current_solution

        if current_solution is initial solution
                global_best = current_solution

        for inter in range 1 to iteration_count
                local_best = generate_cadidates (current_solution, current_layer, n_list[], data_set)

                tabu_list[]   ← decrement last four solution's tabu_tenure value
                # Note: if solution's tabu_tenure reaches zero after decrementing pop solution from list

                local_best, tabu_list[] = compare_local_and_global_best (local_best, tabu_list[])
                # Note: if local_best is better than global_best, then global_best is set to local_best in
                #       above method

                current_solution = local_best

return global_best
```

Algorithm 6: Generate Possible Solutions Returning Best Performing Solution

Algorithm 6 describes the process of searching for possible solutions for an ANN with 1 hidden layer, and then applying the same process to topologies with 2, 3, and so on hidden layers up to a predetermined maximum number of hidden layers given by the *max_hidden_layer* variable. At the start of each iteration, a *solution_object* is created and set to the *current_solution*. The algorithm generates multiple *candidate_solution* based on the *current_solution*, and the best performing *candidate_solution* is stored in the *local_best* variable. The performance of the *local_best* is then compared to that of the *global_best*, and the better of the two is stored as the new *global_best*. The *local_best* then replaces the value of the *current_solution* for use in creating the next group of *candidate_solution*. To allow for previously searched topologies to be researched, the last four *solution_object* in the *tabu_list* have their *tabu_tenure* reduced by one. When a *solution_object* in the *tabu_list* has its *tabu_tenure* reduced to zero, it is removed from the *tabu_list*, allowing it to be used in future searches. The *tabu_list* prevents *candidate_solution* from being considered the best performing **candidate_solution** unless their performance is better than the *global_best*. As the algorithm concludes, the *global_best* is returned as the best performing *solution_object*.

Fill Neuron List: Tabu Search Method Pseudocode

Input: current_layers
Output: n_list[]

```
# lower and upper bounds for number of neurons allowed in each hidden layer
lower_bound =  math.ceil((n_inputs, n_outputs)/2)
upper_bound = math.ceil((n_inputs, n_outputs)x2/3)

for layer in range 1 to current_layers
        n_list[layer]   ←  append random whole number between lower_bound and upper_bound

return n_list[]
```

Algorithm 7: Randomly Fills Number Of Neurons For Each Layer

As **Algorithm 6** increases the number of hidden layers a new topology for the number of neurons in each layer needs to be generated. The generation is performed by **Algorithm 7**, which randomly selects a number of neurons for each hidden layer. The selected number is restricted by a *lower_bound* and *upper_bound* which are calculated with the following formulas:

$$lower\ bound\ =\ math.ceil((n_{inputs}\ +\ n_{outputs})/2) \quad (9)$$

$$upper\ bound\ =\ math.ceil((n_{inputs}\ +\ n_{outputs})\ *\ 2/3) \quad (10)$$

For both formulas their results are rounded up to the nearest whole number for the function's solutions.

Compare Local And Global Best: Tabu Search Method Pseudocode

Input: local_best, tabu_list[]
Output: local_best, tabu_list[]

```
# compares fitness value of local and global best
if local_best better than global_best
        # append local_best to tabu_list
        tabu_list[]   ← append local_best
        global_best = local_best
else
        # if local_best is not in tabu_list append it to list
        if local_best_solution not in tabu_list
                tabu_list[]   ← append local_best

return local_best, tabu_list[]
```

Algorithm 8: Compares Local And Global Best Solutions

In **Algorithm 8**, when comparing the performance of the *local_best* and *global_best*, an exception is implemented according to the reference paper. If the *local_best* performs better than the *global_best*, it is appended to the *tabu_list*, even if its topology is already included. This exception is made because the *local_best* is worth

exploring further, even though it is tabu. As a result, multiple versions of the *local_best* may be included in the *tabu_list*, which will increase the number of iterations during which the *local_best* is tabu. Even after the first version of the *local_best* is removed from the *tabu_list* when its *tabu_tenure* is reduced to zero, another version will still need to have its *tabu_tenure* reduced to zero before it can be considered non-tabu.

2.3.3 Generate Candidates Topology

Generate Candidates: Tabu Search Method Pseudocode

Input: current_solution, current_layers, n_list[], data_set, tabu_list[]
Initialize: P_{max}, p, K, candidate_list[]
Output: best_candidate

```
increase_n_list[] = n_list[]
decrease_n_list[] = n_list[]

for candidate_count in range 1 to Pmax/2
        # increase neurons
        change_type = 1
        increase_n_list[] = generate_cadidate_topology (current_layers, increase_n_list[], p, K, change_type)

        if increase_n_list[] not None
                # append candidate list with candidate solution
                candidate_list[]   ← append solution (increase_n_list[], tabu_tenure, data_set)

for candidate_count in range 1 to Pmax/2
        # decrease neurons
        change_type = -1
        decrease_n_list[] = generate_cadidate_ topology (current_layers, decrease_n_list[], p, K, change_type)

        if decrease_n_list[] not None
                # append candidate list with candidate solution
                candidate_list[]   ← append solution (decrease_n_list[], tabu_tenure, data_set)

best_candidate  = determine_best_candidate (candidate_list[])

if best_candidate is None
        return current_soulution
else
        return best_candidate
```

Algorithm 9: Generates Alternative Solutions

Algorithm 9 controls the creation of *candidate_solution*, which means it creates the candidate ANNs, trains them, and collects the post-training metrics. These

candidate_solution are alternative topologies of the *current_solution*. The *candidate_solution* modifies the prior *candidate_solution* by randomly modifying the number of neurons in each of the hidden layers. This modification is not guaranteed, and it is possible that a given hidden layer will remain unchanged from the prior *candidate_solution*. The total number of *candidate_solution* to be created is split by the type of modification performed on them. Half of the *candidate_solution* modify the number of neurons in the hidden layers by increasing the amount, while the other half decrease the amount. The first *candidate_solution* produced by each half uses the *current_solution* for its base topology. The best-performing *candidate_solution* is returned by the algorithm, and it can be selected even if it is tabu, as long as it performs better than the *global_best*.

Generate Candidate Topology: Tabu Search Method Pseudocode

Input: **current_layers, n_list[], p, K, change_type**
Output: **n_list[]**

```
for layer in range 1 to current_layers
        w = random (0, 1)

        # p is the probability of modifying the number of neurons
        if w ≥ p

                # updates hidden layer's neuron is able
                if  n_list[layer] + ceiling (K * change_type) within upper and lower bounds
                        # increases or decreases neurons depenent on change_type
                        n_list[layer] += ceiling (K * change_type)
                else:
                        return None

return n_list[]
```

Algorithm 10: Randomly Increase Or Decrease Hidden Layer Neuron Count

Algorithm 10 covers the modification of *candidate_solution* topologies in more detail. For each hidden layer in the topology, the probability of modifying the number of

its neurons is given by the p value. If the hidden layer is able to be modified, then the layer is increased or decreased depending on the *chance_type*, which is just a -1 or 1 to allow for the algorithm to perform both functions. The *K* value determines the extent of change applied to the layer. Before the modification is accepted as a valid modification amount, it needs to be checked to make sure it is within the range of the *lower_bound* and *upper_bound*. Once accepted, the modification is made, and the algorithm proceeds to the next hidden layer. After all hidden layers have been modified, if possible, the modified topology is returned.

Determine Best Candidate: Tabu Search Method Pseudocode

Input: **candidate_list[], tabu_list[]**
Output: **best_candidate**

```
best_candiate = None

for candidate in candidate_list[]

        if candidate is better then global best or candidate is not in tabu_list[]
                best_candidate = compare_solutions (best_candidate, candidate)

return best_candidate
```

Algorithm 11: Searches Candidate List For Best Performing Candidate Solution

Algorithm 11 compares all the *candidate_solution* produced by **Algorithm 9**. The *candidate_solution* will be compared if it meets the following requirements: it performs better than the *global_best* or it is not labeled as tabu by its topology being in the *tabu_list*. Once all *candidate_solution* that meet the requirements have been compared, the best *candidate_solution* is selected and returned by the algorithm.

Chapter 3

Topology Optimization Method Evaluations

This chapter covers the implementation of the evaluations conducted in this paper. It includes the Python frameworks and tools used in conjunction with the evaluations. It also explains in detail how the datasets used during the evaluation were collected and processed and provides the reasoning behind their selection. Finally, the paper provides a description of the parameters used for each optimization method, along with an explanation of how the parameters of the ANN were initialized.

3.1 Framework and Tools

Different frameworks and tools were used for different phases of the thesis development, from implementing the methods from scratch, to analyzing and visualization of the results of the conducted evaluations.

Implementation of the methods: *Scikit-learn* is a Python framework that is used for predictive analytics, and was selected for its ability to implement ANNs and to process datasets. The other options considered for implementing ANNs were *TensorFlow* and *PyTorch*. Scikit-learn was ultimately selected due to the author's positive experiences during past applications. Several additional Python libraries were needed to successfully use Scikit-learn in executing the ANNs used by the implemented methods. *Pandas*, a data analytics and manipulation tool was used to read the datasets csv files and transform them into something that can be understood by Scikit-learn. *Numpy* is a library that allows the manipulation of large multidimensional arrays and matrices, and it also includes many mathematical functions that can be performed on

arrays and matrices. Scikit-learn leans quite heavily on this library for their implementation of ANNs.

Analysis and visualization of results: *Microsoft Excel* has many options for the analysis and visualization of data, so it was used as a medium for analyzing the evaluations' results. Excel was also used for the creation of charts used to visualize and explain evaluations' findings.

Evaluation execution environment: *Kaggle* is a well-known Data Science website that hosts a large number of resources in terms of datasets for research and practice purposes. One of the resources is *Kaggle Notebooks*, a cloud based remote computational environment which was used in the execution of the evaluations. It allows access to more computational resources then the author has available locally and the ability to conduct concurrent sessions without a reduction of computational resources available. No more than 10 concurrent notebook sessions can run at the same time for a single user. For each notebook session, Kaggle limits the maximal max computational time allowed to 12 hours of execution time. Each session is allocated 20 Gigabytes of disk space to save output files. Based on the technical specification at the time the evaluation was conducted on Kaggle, the CPU specifications were 4 CPU cores per notebook with 16 Gigabytes of RAM. The other cloud computing environments considered were *Microsoft Azure Notebook* and *Google Colab*. Kaggle Notebooks was selected as the environment to use as the platform that hosts the datasets used for the evaluation allowing for straightforward importation of the datasets into the environment.

3.2 Dataset Selection and Preprocessing

Three datasets were selected, being sourced from Kaggle. The first dataset reposted to kaggle (Shahane, 2021) from the original source (Gambi, 2020). The dataset covers the air quality of a room using the gas concentration contained within the room, with 1845 collected samples. Each sample contains 6 integer values (250 to 3000), acquired by a series of 6 gas sensors that respond to various classes of chemical compounds. Each sample also includes the activity which took place in the room while the sample was collected. The 4 activities conducted in the room were labeled by the integers 1 through 4:

1. Normal activities which is correlated to clean air
2. Meal preparation
3. Presence of smoke over a short period of time
4. Cleaning being conducted with household cleaning products.

The next dataset is a collection of 768 patients' medical information that is used to train an ANN to determine if a patient has diabetes (Soni, 2021). Each sample contains 8 values, with each sample also being correlated to a diagnosis value of a 0 or 1. A value of 0 means the patient doesn't have diabetes and a 1 means they do. The 8 values for the sample contain 6 integer values that represent the number of times the patient was pregnant (0 to 17), glucose levels (0 to 199), blood pressure (0 to 122), skin thickness (0 to 99), insulin levels (0 to 846), and their age in years (21 to 81). The

sample also contains real values for the patient's BMI (body mass index) (0 to 67.1) and diabetes pedigree function (0.08 to 2.42).

The final dataset used is based on the Modified National Institute of Standards and Technology (MNIST) (Deng, 2012), an image collection of handwritten integer digits (0 to 9), the dataset was converted by Dato-on (2021) from a set of 28x28 grayscale images to a collection of pixel values in csv format. Each sample consists of 784 pixel values (0 to 255) with 0 indicating a pure black pixel and 255 a pure white pixel. Each sample corresponds to an integer value (0 to 9). The dataset was pre-split into training and testing datasets consisting of 60000 and 10000 samples respectively. For use in this thesis only the samples containing digits 0 through 4 were used, which reduces the number of samples used from the training and testing datasets to 30602 and 5139 respectively.

The air quality and diabetes datasets were selected for their low number of samples contained by the datasets. An ANN trained on a lower number of samples can lead to an under-fitted ANN which means that the ANN fails to understand the data from a lack of information. This issue can be solved by having a ANN that has a more complex topology. This data set allows the author to make sure that the topology optimization methods do not reduce the ANN's topology by too much. The MNIST dataset was selected due to its use in the evaluations conducted in the reference paper for the Tabu Search Method (Gupta, 2020). It was also due to the high amount of input values for each sample in the dataset leading to the possibility for an overfitted ANN. This can be solved by reducing the complexity of the topology of an ANN. This dataset

ensures the topology optimization method reduces the ANN's topology by an adequate amount.

For each dataset the following preprocessing steps were taken. Using Scikit-learn's standard scaler method the datasets were normalized so that the respective dataset's mean is equal to 0 and the variability (average squared deviations from the mean) is 1. This will keep large values in a sample from negatively prejudicing the ANN to one prediction over all others. The air quality and diabetes datasets were separated into training and testing datasets with 80% of the original dataset's samples being used for the training dataset with the remaining 20% of the samples being allocated to the testing dataset.

3.3 Initial Evaluation Setup

The metrics collected during the course of testing both optimization methods use the metrics described in **section 2.1.4** *Measurements of ANN Performance* as a base. The metrics were collected using the *Scikit-learn* framework's metrics methods. The following methods were used: *accuracy_score*, *balanced_accuracy_score*, *mean_absolute_error*, *precision_score*, *recall_score*, and *f1_score*. All the methods used the predicted and true output values for the dataset the ANN is being trained for in a given evaluation. The *accuracy_score* function computes the accuracy metric from **section 2.1.4**. The *balanced_accuracy_score* is used when there is a possibility of an imbalanced dataset and is calculated by taking the mean result of the recall value for each of the output neuron results. If the result is close to the accuracy result, it means that the dataset is balanced. The *mean_absolute_error* is the same method used to

assess the performance level of an ANN in **section 2.2.1** *Train Initial ANN*. For the *precision_score*, *recall_score*, and *f1_score* functions, the average for the multiclass dataset is set to compute a type of weighted average.

3.4 Weights Power Method Evaluation Setup

The method parameters used in the Weights Power reference paper were matched wherever possible. These parameters include the convergents criterion for the training and testing datasets, which were set to 0.1 and 0.5 respectively. However, due to the strict criterion that requires the ANN to achieve an accuracy of 0.9 or higher for the initial ANN, some of the parameters were changed from what was used in the paper.

For example, the momentum used by the reference paper was set to 0.0, which was stated to simplify the implementation of their evaluation. However, for the evaluations conducted in this paper, a momentum of 0.9 was used. This value was determined by testing various momentum values ranging from 0 to 1 and selecting the value that had the most positive effect on the accuracy for the datasets used in this paper's evaluation.

Similarly, the learning rate of 0.05 used by the reference paper was also an issue for the accuracy of the ANN. The step size for updating the weights during training the ANN was failing to converge. To address this issue, the default learning rate of 0.001 used by *Scikit-learn* was selected as a replacement value.

Since the reference paper did not specify other parameters used for the ANN, *Scikit-learn* default values were used, with changes made as necessary. The weight optimizer was changed from its default value of the adam optimization algorithm to

stochastic gradient descent to match the one used during the testing of the Tabu Search Method, making the comparisons between the two methods relevant.

The default batch size is 200 or the number of samples, whichever is lower. This value was changed to 10 for air quality, 5 for diabetes, and 250 for MNIST dataset. The batch sizes were determined after testing different numbers of sizes with the diabetes dataset, which was the worst-performing of the three. Then, the ratio between the batch size and the number of samples was calculated, and the ratio was used to determine the batch size for the other datasets. The ratio was calculated by using the number of samples in the training dataset for MNIST which is 30602 samples. This value was rounded down to 30000, when comparing the batch size to this value (250 / 30000 = 0.008) leads to a ratio of 0.008.

Based on evaluationation with different values, adjustments were made to improve the performance of the ANN. The maximum number of iterations for training epochs (a complete cycle through all samples in training dataset) was increased from 200 to 2000, providing the ANN with more training time, which yielded positive results. Similarly, changes were made to the stopping criteria. The maximum number of epochs for training, even if the error does not improve, was raised from 10 to 90. Additionally, the tolerance level, representing the minimum required improvement in error at the end of each training epoch, was adjusted from $1.0x10^{-5}$ to $5.0x10^{-5}$. These adjustments were based on careful evaluation of various values, selecting the ones that demonstrated the most significant positive effects on the ANN's performance.

The implemented approach allows for the removal of a weight or neuron from an ANN without necessitating its complete reconstruction, as devised by the author. Removing a weight involves setting the weight's value to zero, while removing a neuron requires setting all its weight and bias values to zero. As seen in **Formula (1)**, the output of a neuron is calculated by multiplying a given weight value with its associated input value, and adding each of these weight input results together. If a weight value is set to zero, it will have no effect on the calculated total for all weight input results. This applies to neuron removal as well, where all its 0-valued weights and biases will nullify any input values coming into the neuron. This leads to passing the zero value into the activation function, which for this evaluation is RELU as shown in **Formula (2**), resulting in an output of zero for the neuron. This zero output will nullify the weight values associated with its output, causing it to have no effect on the next layer's output results.

As mentioned earlier, setting values to zero can prevent removed weights and neurons from affecting ANN predictions. However, since these zero values are still considered part of the ANN, *Scikit-learn* modifies them during the update step of training. To address this, an override of the method that controls the update process in *Scikit-learn* was needed. This was achieved using a Python technique called *Monkey Patching*, which involves replacing a framework method with a patched method that is called instead of the original method.

The patched method we implemented keeps track of all weight and bias values that were removed and skips these values during the update step, preserving the values set to zero. This ensures that the removed weights and neurons do not interfere with the training process.

Another issue that was encountered with *Scikit-learn* is that by default, when an ANN is trained, its weight and bias values are randomized at the beginning. This would overwrite the manually set values needed to retain for the evaluation, which required retraining after every removal. To overcome this, the warm start parameter was modified when creating the ANN. Setting this parameter from its default value of False to True, which prevents Scikit-learn from randomizing the values and ensures consistency between training sessions. By making these modifications, the zero values were preserved in the ANN's weights and biases throughout the training process.

3.5 Tabu Search Method Evaluation Setup

The reference paper for the Tabu Search method only specified a few parameters to be used for the ANN training. The paper described utilized a stochastic gradient descent technique along with momentum backpropagation. Specifically, the momentum value for backpropagation was set to 0.7.

The reference paper provides more information about the values used in the Tabu Search method parameters and the same values were used for this thesis. For each layer, a number of local bests are created equal to the number of iterations, which was set to 10 for the evaluation conducted in this paper. The chance of either decreasing or increasing the number of neurons from each of the local best layers for a given candidate is set to 0.5, while the percentage change amount is set to 0.03 of the current number of neurons in the local best layer. A total of 20 candidates are generated during each iteration.

In addition, the tabu tenure value was set to 4, which acted as a countdown value for how long to keep a given solution in the tabu list. However, the implementation in the reference paper did not seem to strictly follow the use of the tabu tenure, as it seems to allow the algorithm to generate solutions that match the topology of a solution already contained in the tabu list. This deviates from the tabu search described in a paper by Glover (1986), which was cited in the reference paper. Upon introducing the concept of merging a tabu search with gradient descent, the author recognized that the apparent contradictions were simply a result of oversight during the editing process. Consequently, the author decided to implement the original idea of incorporating a tabu list, as detailed in section 2.3.

Chapter 4

Topology Optimization Method Comparisons

This chapter starts with providing an overview of the results obtained for the datasets used in the evaluations and verifies their balance to ensure that they are not biased towards any particular output result. It also checks that the ANNs created are not prone to predicting one output incorrectly over another. Following this section, there are individual sections for the comparison between the optimization methods for each dataset. The datasets are presented in the order of Air Quality, Diabetes, and then MNIST.

For each method and each dataset 10 final ANNs were trained. The metrics presented in this chapter are the mean values averaged over all 10 runs. As the Tabu Search method creates multiple initial solutions during the optimization process, their metrics were averaged together into a single initial solution.

4.1 Overview

As covered in the previous chapter, the difference between the ANN's balanced accuracy and accuracy score allows inference about the balance of the dataset. The tables below show the balanced accuracy and accuracy score, along with the absolute difference between the two. **Table 3** shows the results produced by the Tabu Search Method, and **Table 4** shows the results produced by the Weights Power Method. It can be observed that the largest difference in both instances involves the datasets created from the Diabetes data, with the difference ranging from 0.0127 to 0.0695 across both

tables. The fact that the largest difference is relatively small is indicative of the datasets being fairly well balanced.

Tabu Search Method – Method Best Solution				
Data	Dataset	Balanced Accuracy	Accuracy	Absolute Difference
Air Quality	Train	0.8444	0.8601	0.0157
	Test	0.8592	0.8629	0.0037
Diabetes	Train	0.6691	0.7336	0.0645
	Test	0.7162	0.7857	0.0695
MNIST	Train	0.9956	0.9956	0.0000
	Test	0.9899	0.9899	0.0000

Table 3: Tabu Search Accuracy Comparison

Weights Power Method - Method Best Solution				
Data	Dataset	Balanced Accuracy	Accuracy	Absolute Difference
Air Quality	Train	0.9203	0.9209	0.0006
	Test	0.8970	0.8967	0.0003
Diabetes	Train	0.8709	0.8836	0.0127
	Test	0.6592	0.6968	0.0376
MNIST	Train	0.9414	0.9418	0.0004
	Test	0.9479	0.9481	0.0002

Table 4: Weights Power Accuracy Comparison

Table 2 provides a detailed explanation of recall, precision, and f1, which are used to evaluate whether an ANN has a propensity for certain prediction errors. Recall indicates the probability that the ANN can correctly predict all actual positive cases, while precision reflects the probability that it can predict positive cases correctly. The f1 score is the ratio between recall and precision and represents the ANN's ability to identify positive cases. **Table 5** and **Table 6** present the performance of ANNs optimized using the Tabu Search and Weights Power methods, respectively, on various datasets. The MNIST datasets had the best performance for both optimization methods, followed by the Air Quality datasets, with the worst performance on the Diabetes datasets. This

trend seems to correspond with the number of samples provided by each training dataset, with MNIST containing 30,602 samples, Air Quality containing 1,476 samples, and Diabetes containing 614 samples. Both tables indicate that the optimized ANNs performed adequately.

Tabu Search Method – Method Best Solution				
Data	**Dataset**	**Recall**	**Precision**	**F1**
Air Quality	Train	0.8601	0.8606	0.8597
	Test	0.8629	0.8639	0.8622
Diabetes	Train	0.7336	0.7292	0.7155
	Test	0.7857	0.7839	0.7728
MNIST	Train	0.9956	0.9956	0.9956
	Test	0.9899	0.9899	0.9899

Table 5: Tabu Search Recall, Precision, & F1

Weights Power Method - Method Best Solution				
Data	**Dataset**	**Recall**	**Precision**	**F1**
Air Quality	Train	0.9209	0.9214	0.9210
	Test	0.8967	0.8975	0.8967
Diabetes	Train	0.8836	0.8871	0.8829
	Test	0.6968	0.7011	0.6971
MNIST	Train	0.9418	0.9417	0.9417
	Test	0.9481	0.9479	0.9479

Table 6: Weights Power Recall, Precision, & F1

4.2 Air Quality Dataset

Table 7 showcases the results data for the Tabu Search Method, while **Table 8** presents the results for the Weights Power Method. These tables provide an overview of the metrics achieved by each method upon the Air Quality dataset. The data presented in the tables is a result of conducting 10 separate runs for each of the methods. Information regarding the initial networks and the best ANN obtained from each of the 10 runs was collected and recorded.

In the Tabu Search Method, five initial solutions were generated for each run, with an increasing number of hidden layers. For each run, the neuron amounts of the hidden layers were averaged. Subsequently, the averaged results for each layer across all runs were calculated. These averaged results, representing the final initial solution, are displayed in **Table 7**. Additionally, the table includes the method's best solution, which represents the most optimal outcome obtained throughout the optimization process, along with its corresponding metrics.

The Weights Power Method follows a procedure of generating the initial solution starting with a single hidden layer and progressively increasing the number if neurons until it reaches the desired performance level. If the number of neurons for a hidden layer reaches the upper bound and another hidden layer is added with a random number of neurons between the lower and upper bound. The metrics presented in **Table 8** represent the average values derived from the last initial solution across all runs. It should be noted that if the average neuron count for a particular hidden layer is zero, it indicates that hidden layer wasn't necessitated for the ANN to reach the desired performance level for any of the 10 runs for the given dataset. Additionally, the table includes the metrics for the method's best solution, indicating the highest performing outcome achieved by the method during the optimization process. Since the method's best solution can only reduce the number of neurons and not add hidden layers, it is implied that the method best solution will never contain more hidden layers or neurons than the initial solution.

The metrics displayed in both tables encompass various aspects, such as the average number of neurons for each corresponding hidden layer (H1-H5), the average

total number of neurons and weights across all hidden layers (Total Neurons, Total Weights), and the average accuracy of both the initial solutions and the method's best solutions (Accuracy). Furthermore, the tables provide information about the specific dataset from which these metrics were obtained, allowing for a comprehensive understanding of the results achieved by each method (Dataset).

The data from the results shows that the Weights Power method resulted in a smaller ANN on the Air Quality dataset, with a single hidden layer and a corresponding low number of neurons while maintaining higher accuracy when compared to the Tabu Search method. This is most likely due to the Weights Power method's approach to the creation of its initial solution as it only adds neurons and hidden layers as needed to reach its required performance level, which increases the likelihood of its initial topology to be more compact. This observation is evident in **Table 8**, where the utilization of a single hidden layer is illustrated for the initial solution. It can be observed that the method best solution achieved is either equal to or less than the number of neurons and hidden layers present in the initial solution. This relationship between the method's approach and the resulting solutions is clearly demonstrated in the table. On the other hand, the Tabu Search method makes use of all its available hidden layers up to the maximum which for this dataset was set to 5, which can be seen in **Table 7** with its use of the maximum number of hidden layers in this search for a topology with the best performance.

Air Quality – Tabu Search Method									
Solution	H1	H2	H3	H4	H5	Total Neurons	Total Weights	Accuracy	Dataset
Initial	5.9	5.8	6.2	5.8	5.9	29.6	199.36	0.5762	Train
								0.5556	Test
Method Best	7	5.4	3.5	1.3	1.4	18.6	110.67	0.8601	Train
								0.8629	Test

Table 7: Tabu Search Initial & Method Best Topology On Air Quality

Air Quality – Weights Power Method									
Solution	H1	H2	H3	H4	H5	Total Neurons	Total Weights	Accuracy	Dataset
Initial	5.2	0	0	0	0	5.2	31.2	0.9513	Train
								0.9293	Test
Method Best	4.9	0	0	0	0	4.9	15.8	0.8967	Train
								0.9209	Test

Table 8: Weights Power Initial & Method Best Topology On Air Quality

The scatter plot in **Figure 6** depicts distinct groupings based on the number of hidden layers in the ANN within the Tabu Search method. As the method progresses and includes more hidden layers, these groupings become more pronounced. It's worth noting that the groupings are defined by averaging the local best solutions obtained from multiple runs of the method.

In each run of the Tabu Search method, 50 local best solutions are generated for a specific dataset. To calculate the average local best solution for a particular index, we'll use the notation *LBS_avg(i, j)*. Here, *i* represents the index of the hidden layer grouping (ranging from 1 to 5), and *j* represents the index of the specific local best solution within that grouping (ranging from 1 to 50). Therefore, *LBS_avg(i, j)* denotes the average of the local best solutions *LBS(i, j)* obtained from each run for a specific index *j* within grouping *i*.

As the number of neurons increases within each grouping, the scatter plot shows that the accuracy values of the average local best solutions, *LBS_avg(i, j)*, become more scattered. This indicates a wider range of accuracy values as more neurons are included, reflecting greater performance variability.

The method's best solution, highlighted in **Table 7**, lies between the 3rd and 4th groupings when considering the scatter plot from the left side. As the number of hidden layers increases, the groupings become more distinct, and the accuracy values of the average local best solutions, *LBS_avg(i, j)*, exhibit greater scattering. However, when the number of neurons decreases, the subsequent groupings only experience a slight loss of cohesion. Therefore, the method best solution in **Table 7**, positioned between the 3rd and 4th groupings, represents a crucial configuration that strikes a balance between performance and the number of neurons in the hidden layers. It demonstrates promising results without significant loss of cohesion observed in the subsequent groupings.

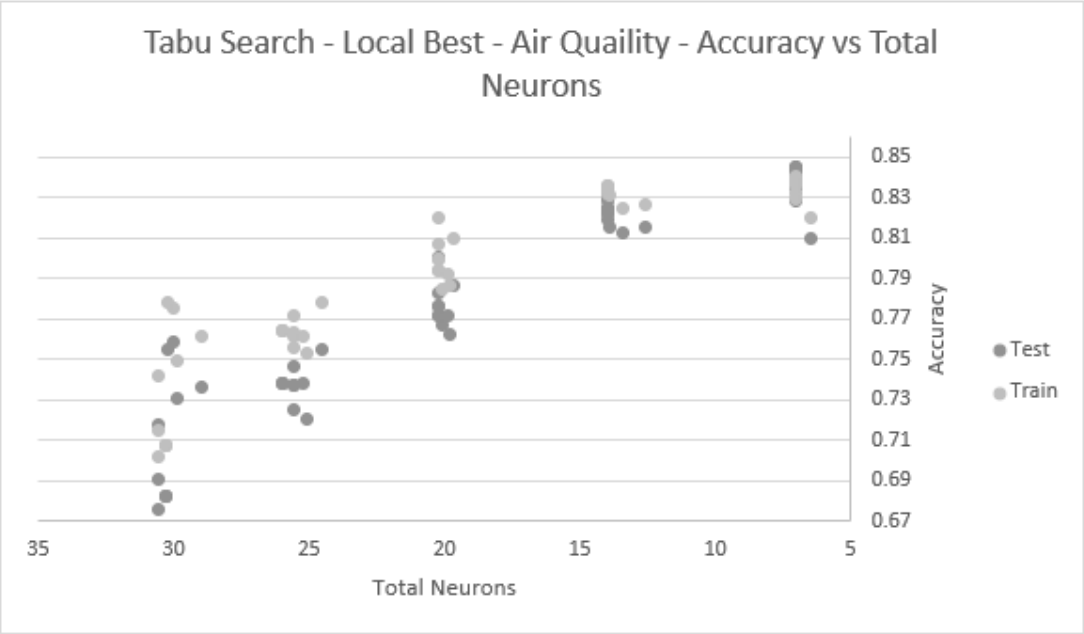

Figure 6: Tabu Search's Local Best Accuracy Vs Total Neurons On Air Quality

4.3 Diabetes Dataset

The Diabetes dataset proved to be more challenging for the Weights Power method, which had difficulty achieving the minimal performance required for an initial solution to be used by the method. As shown in **Table 10**, the ANN created by the Weights Power method had trouble with overfitting to the training data, as evidenced by the significant difference in accuracy score between the training and testing datasets. Additionally, the method was only able to remove an average of 0.8 neurons and had no effect on the total number of neurons. In contrast, the Tabu Search method performed much better, as seen in **Table 9**, and was able to remove an average of 19.3 neurons. In fact, its final neuron count for its method best was less than half the size of the Weights Power method's final count, with 10.6 total neurons compared to 27 neurons. While the Tabu Search method did start with more neurons in its initial solution, this alone does not account for the significant difference in the final neuron count between

the two methods. One possible reason for this difference is that the Weights Power method stops trying to remove neurons and weights after its ANN falls below a certain level of performance. This requirement could prevent the Weights Power method from removing more neurons after it falls into a local minimum, which is not an issue with the Tabu Search method, as it was designed with the ability to escape local minimum traps.

Diabetes – Tabu Search Method									
Solution	H1	H2	H3	H4	H5	Total Neurons	Total Weights	Accuracy	Dataset
Initial	5.9	6	6.1	5.9	6	29.9	196.59	0.6469	Train
								0.6773	Test
Method Best	6.6	3.3	0.4	0.3	0	10.6	76.32	0.7336	Train
								0.7857	Test

Table 9: Tabu Search Initial & Method Best Topology On Diabetes

Diabetes – Weights Power Method									
Solution	H1	H2	H3	H4	H5	Total Neurons	Total Weights	Accuracy	Dataset
Initial	6	6	6	5.9	3.9	27.8	178.8	0.9062	Train
								0.6974	Test
Method Best	6	6	5.9	5.5	3.6	27	170.8	0.8836	Train
								0.6968	Test

Table 10: Weights Power Initial & Method Best Topology On Diabetes

4.4 MNIST Dataset

The MNIST dataset was the largest used during this evaluation with the most amount of inputs for the ANNs to learn from and showed the limitations of the Tabu Search method. Since the Tabu Search is limited in the amount of changes that it can make to an ANN due to its set number of iterations, there is a hard cap on the number of neurons that can be removed. This is a limitation that the Weights Power method doesn't have as it removes neurons and weights until its performance drops too far. **Table 11** shows that the Tabu Search method was only able to remove 23.4 neurons,

while the Weights Power method shown in **Table 12** removed 389 neurons, which is a massive difference. The Weights Power method didn't just stop at removing neurons, it also removed a significant number of weights, with its final total of weights at 63.9. For a single hidden layer constructed of 5 neurons with 784 input values, there would be 3920 weight values normally. The final total of neurons and weights leads to a massive reduction in the complexity of the ANN. The Weights Power method was able to obtain this result while maintaining a comparable accuracy score to the ANN resulting from the Tabu Search method.

It should be noted that although this evaluation utilized a subset of the MNIST dataset consisting of only 5 digits, the resulting best solution of the method included a single hidden layer with an average of 418.3 neurons. In contrast, the reference paper employed the entire MNIST dataset, including all 10 digits, and reported results from a single hidden layer containing 518 neurons (Gupta & Raza, 2020). The increase in the total number of neurons in the reference paper is likely attributed to the added complexity of capturing a wider range of patterns and variations introduced by the inclusion of all digits. Despite the differences in dataset subsets and the total number of neurons, the evaluation results remain comparable to those presented in the reference paper, indicating the effectiveness of the Tabu Search method in both scenarios.

MNIST – Tabu Search Method									
Solution	H1	H2	H3	H4	H5	Total Neurons	Total Weights	Accuracy	Dataset
Initial	441.7	0	0	0	0	441.7	348501.3	0.9958	Train
								0.9878	Test
Method Best	418.3	0	0	0	0	418.3	204548.7	0.9956	Train
								0.9899	Test

Table 11: Tabu Search Initial & Method Best Topology On MNIST

MNIST – Weights Power Method									
Solution	H1	H2	H3	H4	H5	Total Neurons	Total Weights	Accuracy	Dataset
Initial	394	0	0	0	0	394	308896	0.9998	Train
								0.9900	Test
Method Best	5	0	0	0	0	5	63.9	0.9418	Train
								0.9481	Test

Table 12: Weights Power Initial & Method Best Topology On MNIST

Chapter 5

Conclusion & Future Work

5.1 Conclusion

This paper aims to investigate the feasibility of implementing the Weights Power and Tabu Search optimization methods using the existing framework and compare their ability to reduce the complexity of the ANN while maintaining acceptable performance. The two methods were implemented using Python's *Scikit-learn* framework and evaluated using the Air Quality, Diabetes, and MNIST datasets obtained from *Kaggle*, a data science website. The evaluations were conducted on *Kaggle's* cloud servers.

From the conducted evaluation, the Weights Power method appeared to result in a more compact ANN while maintaining acceptable performance levels for two out of the three datasets. The following results were concluded from the datasets.

On running the methods on the Air Quality dataset, it was found that the Tabu Search method's initial solution can result in a larger ANN and has constant pressure to improve its performance, only decreasing the number of neurons if the resulting ANN has better performance. The Weights Power method, on the other hand, focuses more on simplifying the ANN, and performance is only a factor for ensuring it maintains its minimal performance levels.

For the Diabetes dataset, the Tabu Search method was able to reduce the ANN more than the Weights Power method. However, the Weights Power method seemed to face the issue of falling into a local minimum without the ability to overcome this hurdle to keep removing components. In contrast, the Tabu Search method was built with the ability to deal with the issue of local minima.

The MNIST dataset showed the benefit of the Weights Power method with the ability to keep reducing the number of ANN components until it fails to attain its minimal performance. However, the Tabu Search method had hard-set limits for its reduction ability. While the Weights Power method performed better in its optimization task, the Tabu Search seemed to have the capacity to improve beyond what the Weights Power method could obtain.

Therefore, if the evaluation were extended, it would focus on improving the Tabu Search method. The possible improvements will be covered in the next section.

5.2 Future Work

With the decision to focus on the Tabu Search method for any future work, some areas of further study would consist of implementing a Tabu Search that is more in line with the original implementation and allows the tabu list to restrict the creation of ANNs toplogies that are still contained in the list. Additionally, changing the number of neurons that can be removed or added for candidate solutions would be beneficial. This amount should start with larger amounts and

decrease as candidates fall below a set level of performance, helping to address the hard limits inherent in the method. It would also be interesting to increase the chance of reducing the number of neurons compared to increasing them, which would give the Tabu Search method pressure to decrease complexity while retaining the ability to escape local minima that the ability to both increase and decrease the number of neurons gives Tabu Search. Finally, allowing the Tabu Search a way to dynamically increase the limits of how many ANNs the method can search would be intriguing, although the parameters that would allow this would require further research.

References

Chatterjee, M. (2022, January 20). *Top 20 Applications of Deep Learning in 2022 Across Industries*. *GreatLearning Blog: Free Resources What Matters to Shape Your Career!* Retrieved February 23, 2022, from https://www.mygreatlearning.com/blog/deep-learning-applications/

Dato-on, D. (2018). *MNIST in CSV* [Dataset; CSV]. In *Kaggle* (Version V2). https://www.kaggle.com/datasets/oddrationale/mnist-in-csv

Gambi, E. (2020). *Air Quality dataset for ADL classification* (Version V1) [Dataset; Csv]. Mendeley Data. https://doi.org/10.17632/kn3x9rz3kd.1

Glover, F. (1986). *Future paths for integer programming and links to artificial intelligence*. Computers & *Operations Research*, 13(5), 533–549. https://doi.org/10.1016/0305-0548(86)90048-1

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. *MIT Press*. http://www.deeplearningbook.org

Gupta, T. K., & Raza, K. (2020). *Optimizing Deep Feedforward Neural Network Architecture: A Tabu Search Based Approach*. *Neural Processing Letters*, 51(3), 2855–2870. https://doi.org/10.1007/s11063-020-10234-7

Hagiwara, M. (1994). *A simple and effective method for removal of hidden units and weights*. *Neurocomputing*, 6(2), 207–218. https://doi.org/10.1016/0925-2312(94)90055-8

Li Deng. (2012). *The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]*. IEEE Signal Processing Magazine, 29(6), 141–142. https://doi.org/10.1109/msp.2012.2211477

Rashid, T. (2016). *Make Your Own Neural Network: A Gentle Journey Through the Mathematics of Neural Networks, and Making Your Own Using the Python Computer Language (1st ed.)*. Createspace Independent Publishing Platform.

Shahane, S. (2021). *Air Quality Dataset* [Dataset; CSV]. In *Kaggle* (Version V1). https://www.kaggle.com/datasets/saurabhshahane/adl-classification

Sharma, S., Sharma, S., & Athaiya, A. (2020). *ACTIVATION FUNCTIONS IN NEURAL NETWORKS*. *International Journal of Engineering Applied Sciences and Technology*, 04(12), 310–316. https://doi.org/10.33564/ijeast.2020.v04i12.054

Soni, A. (2020). *diabetes_Dataset_clean* [Dataset; CSV]. In *Kaggle* (Version V1). https://www.kaggle.com/datasets/adityasoni2000/diabetes-dataset-clean

Suzuki, K. (Ed.). (2011). *Artificial Neural Networks - Methodological Advances and Biomedical Applications*. *InTech*. https://doi.org/10.5772/644

Vujičić, T. M., Matijević, T., Ljucović, J., Balota, A., & Sevarac, Z. (2016). *Comparative Analysis of Methods for Determining Number of Hidden Neurons in Artificial Neural Network. Central European Conference on Information and Intelligent Systems*, 219–223. https://www.proquest.com/docview/1833969586?pq-origsite=gscholar&fromopenview=true

Appendix A

Source Code & Dataset Links

Weights Power Method

https://www.kaggle.com/code/justinmills/weight-power-method-evaluation

Tabu Search Method

https://www.kaggle.com/code/justinmills/tabu-search-method-evaluation

Air Quaility Dataset

https://www.kaggle.com/datasets/saurabhshahane/adl-classification

Diabetes Dataset

https://www.kaggle.com/datasets/adityasoni2000/diabetes-dataset-clean

MNIST Dataset

https://www.kaggle.com/datasets/oddrationale/mnist-in-csv

Copyright Permission

Name: Mills, Justin Harold

Email (to receive future readership statistics): justin.mills921@topper.wku.edu

Type of document: ['Thesis']

Title: Topology Optimization For Artificial Neural Networks

Keywords (3-5 keywords not included in the title that uniquely describe content): Feedforward Neural Network, Classification Datasets

Committee Chair: Dr. Uta Ziegler

Additional Committee Members: Dr. Mustafa Atici Dr. Michael Galloway

Select 3-5 TopSCHOLAR® disciplines for indexing your research topic in TopSCHOLAR®: Physical Sciences and Mathematics Computer Science Artificial Intelligence and Robotics