A Modified Genetic Algorithm and Switch-Based Neural Network Model Applied to Misuse-Based Intrusion Detection

by:

Ian Stewart

A thesis submitted to the School of Computing
In conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
February 2009

Copyright © Ian Stewart, 2009
Abstract

As our reliance on the Internet continues to grow, the need for secure, reliable networks also increases. Using a modified genetic algorithm and a switch-based neural network model, this thesis outlines the creation of a powerful intrusion detection system (IDS) capable of detecting network attacks.

The new genetic algorithm is tested against traditional and other modified genetic algorithms using common benchmark functions, and is found to produce better results in less time, and with less human interaction. The IDS is tested using the standard benchmark data collection for intrusion detection: the DARPA 98 KDD99 set. Results are found to be comparable to those achieved using ant colony optimization, and superior to those obtained with support vector machines and other genetic algorithms.

Key words: Network security, Intrusion Detection Systems (IDS), data mining, machine learning, real time detection, genetic algorithm, neural networks.
Acknowledgment

My time as a graduate student has been one of the best experiences of my life, and the work which I have completed in this time is something that I will always be very proud of. None of this would have been possible without the assistance of a few special individuals, however.

First off, I would like to thank the Graduate Studies and Computer Science departments at both Trent University and Queen’s University for being so accommodating and helpful throughout this entire process, as well as the Natural Sciences and Engineering Research Council (NSERC) for so generously funding my research.

Next, I would like to thank Dr. Selim Akl, my supervisor from Queen’s University, as well as his entire Parallel Processing Lab Group. Dr. Akl was always there to provide guidance and answers to any questions that I had during my time at Queen’s.

The biggest thank you of all goes to Dr. Wenying Feng, who has been so patient throughout this entire process, dating all the way back to the application phase during my undergraduate days, right through to the completion of this thesis. Without Dr. Feng, I would not be here today.

Finally, I’d like to thank my loving parents and family for always being there for me throughout my entire academic career.
Table of Contents

Abstract ii
Acknowledgements iii
Table of Contents iv
List of Figures ix
List of Tables xi
List of Abbreviations xii

1 Introduction 1
  1.1 Research Motivation ................................................. 1
  1.2 Contributions .......................................................... 2
  1.3 Thesis Outline .......................................................... 3

2 Background 5
  2.1 Network Security ..................................................... 5
    2.1.1 Overview ........................................................... 5
  2.2 Intrusion Prevention vs. Intrusion Detection ......................... 6
    2.2.1 Intrusion Prevention .............................................. 6
    2.2.2 Intrusion Detection .............................................. 7
  2.3 The Genetic Algorithm ................................................. 8
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.2 Fitness Function Limitations</td>
<td>56</td>
</tr>
<tr>
<td>4.4 Improved Neural Network Model</td>
<td>57</td>
</tr>
<tr>
<td>4.4.1 Switch-Based Hidden Nodes</td>
<td>57</td>
</tr>
<tr>
<td>4.4.2 New Fitness Function</td>
<td>58</td>
</tr>
<tr>
<td>4.5 Performance Comparison</td>
<td>60</td>
</tr>
<tr>
<td>4.6 Implications of Results</td>
<td>69</td>
</tr>
<tr>
<td>5 Intrusion Detection System</td>
<td>72</td>
</tr>
<tr>
<td>5.1 System Design and Implementation</td>
<td>72</td>
</tr>
<tr>
<td>5.1.1 Structure of the New IDS</td>
<td>72</td>
</tr>
<tr>
<td>5.2 IDS Performance Evaluation</td>
<td>74</td>
</tr>
<tr>
<td>5.2.1 Evaluation Data Set</td>
<td>74</td>
</tr>
<tr>
<td>5.2.2 Experimental Setup</td>
<td>80</td>
</tr>
<tr>
<td>5.2.3 Testing Results</td>
<td>81</td>
</tr>
<tr>
<td>5.2.4 Support Vector Machines</td>
<td>81</td>
</tr>
<tr>
<td>5.2.5 Ant Colony Optimization</td>
<td>82</td>
</tr>
<tr>
<td>5.2.6 Modified Genetic Algorithm</td>
<td>82</td>
</tr>
<tr>
<td>5.2.7 Improved Genetic Algorithm</td>
<td>84</td>
</tr>
<tr>
<td>5.2.8 Results Comparison</td>
<td>84</td>
</tr>
<tr>
<td>5.2.9 Further Validation With a Different Test Set</td>
<td>90</td>
</tr>
<tr>
<td>6 Conclusion and Future Work</td>
<td>93</td>
</tr>
<tr>
<td>6.1 Research Summary</td>
<td>93</td>
</tr>
<tr>
<td>6.1.1 Contributions of the I-GA</td>
<td>93</td>
</tr>
<tr>
<td>6.1.2 Contributions of the I-NN</td>
<td>94</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>95</td>
</tr>
<tr>
<td>References</td>
<td>97</td>
</tr>
</tbody>
</table>
D.2 KDD99 Data Set .................................................. 131
List of Figures

2.1 The Basic Structure of a Neural Network ......................... 12
2.2 A Sigmoidal Function ............................................ 13
2.3 Self-Organizing Map ............................................ 17
2.4 Support Vectors ................................................. 18
2.5 Support Vector Machine ......................................... 20

3.1 A Geometric Graph of the Meaning of the Crossover Equations ... 34
3.2 Meaning of the Crossover Equations - Different Parents .......... 35
3.3 Benchmark Function 1 ........................................... 43
3.4 Benchmark Function 2 ........................................... 44
3.5 Benchmark Function 3 ........................................... 44
3.6 Benchmark Function 4 ........................................... 45
3.7 Benchmark Function 5 ........................................... 45
3.8 Benchmark Function 6 ........................................... 46
3.9 The Value of the Crossover Weight Parameter ...................... 47

4.1 The Switch-based Neural Network ................................. 52
4.2 Exclusive Or (XOR) Operator .................................... 61
4.3 XOR Performance Results ........................................ 63
4.4 Neural Network Generated by the I-GA for XOR Problem ........ 64
4.5 Neural Network Generated by the M-GA for XOR Problem ........ 64
List of Tables

5.1 Class Distributions of 10% KDD99 Data Set . . . . . . . . . . . . . . 80
5.2 Modified Training Set . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.3 Modified Test Set . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.4 SVM Confusion Matrix . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.5 ACO Confusion Matrix . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.6 M-GA Confusion Matrix . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.7 I-GA Confusion Matrix . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.8 Results Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.9 Detection Rate Comparison . . . . . . . . . . . . . . . . . . . . . . . 86
5.10 False Positive Rate Comparison . . . . . . . . . . . . . . . . . . . . . 87
5.11 False Negative Rate Comparison . . . . . . . . . . . . . . . . . . . . . 88
5.12 IDS Network Size Comparison for I-GA and M-GA . . . . . . . . . . 90
5.13 Second Test Set . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.14 I-GA Confusion Matrix - Different Test Set . . . . . . . . . . . . . . . 91

B.1 Benchmark 1 - Fitness Value After 50 Iterations . . . . . . . . . . . . 115
B.2 Benchmark 2 - Fitness Value After 50 Iterations . . . . . . . . . . . . 116
B.3 Benchmark 3 - Fitness Value After 50 Iterations . . . . . . . . . . . . 117
B.4 Benchmark 4 - Fitness Value After 50 Iterations . . . . . . . . . . . . 118
B.5 Benchmark 5 - Fitness Value After 50 Iterations . . . . . . . . . . . . 119
B.6 Benchmark 6 - Fitness Value After 50 Iterations . . . . . . . . . . . . 120
List of Abbreviations

- ACO: Ant Colony Optimization
- AFRL: The United States Air Force Research Laboratory
- DARPA: The Defense Advanced Research Projects Agency
- GA: Genetic Algorithm
- IDS: Intrusion Detection System
- I-GA: Improved Genetic Algorithm
- I-NN: Improved Neural Network
- KDD: Knowledge Discovery and Data Mining
- LAN: Local Area Network
- M-GA: Modified Genetic Algorithm
- M-NN: Modified Neural Network
- NN: Neural Network
- SVM: Support Vector Machines
- T-GA: Traditional Genetic Algorithm
- T-NN: Traditional Neural Network
Chapter 1

Introduction

1.1 Research Motivation

As malicious computer users become more and more creative, the task of protecting networks continues to become more and more difficult. This thesis aims to create an adaptive intrusion detection system that is capable of identifying attackers in both an accurate and speedy manner.

As this is a real-time detection system, the speed element is an important one. Attackers can cause a lot of damage to a network in a very short period of time, which means that waiting for lengthy classification procedures is not a viable option. A major goal in this thesis will therefore be the development of a simple and quick means of accurately classifying users.

Intrusion detection systems (IDS) are generally divided into two main categories: misuse-based and anomaly-based. Misuse-based IDSs create attacker profiles and then try to fit users into one of those slots during classification, whereas anomaly-
based IDSs simply create a normal user profile and then identify anything that is not within this range as an attack. While anomaly-based IDSs are more capable of identifying new types of attacks than misuse-based system, they also tend to have higher false alarm rates. The IDS proposed in this thesis will be based primarily on the misuse-based approach.

1.2 Contributions

The contributions of this thesis can be divided into three major areas:

1. **New Genetic Algorithm:** One of the main contributions of the thesis is the development of a simple but effective genetic algorithm (GA). The algorithm builds upon previous GAs, but modifies some areas in order to improve performance. As demonstrated in this thesis, the new GA is capable of forming accurate and compact neural networks for the purpose of intrusion detection.

2. **New Neural Network Model:** The new neural network model not only results in the creation of simpler and faster neural networks, but also simplifies the process of actually creating the neural network in the first place. Currently, neural network creators are generally expected to predefine network shape and size based on either domain knowledge or trial and error. This thesis removes this expectation and simplifies the process by applying the new genetic algorithm mentioned in the previous paragraph.

3. **GA/NN-Based IDS:** Using the two constructs listed above and based on the aforementioned goals, a quick and effective IDS is then formulated and compared with other IDSs. The results show that the new algorithm produces results that are comparable to ant colony optimization-based IDSs, and which
are generally better than those produced by previous genetic algorithm and support vector machine-based systems.

1.3 Thesis Outline

The remainder of the thesis is organized as follows:

Chapter 2: An overview of network security, intrusion detection, and data mining techniques; particularly the genetic algorithm and neural network model. Related work in these fields is introduced and discussed in this section.

Chapter 3: A new genetic algorithm is proposed in this chapter, based primarily on the modified GA found in [40]. The new GA is shown to perform more effectively on a set of benchmark functions than either the traditional or modified genetic algorithms.

Chapter 4: A new neural network (NN) model is proposed in this chapter, once again based primarily on a concept laid out in [40]. While neural networks normally use training algorithms to set their weights, size and connectivity are generally set in advance by the network creator. This neural network uses switches on both its connections and hidden nodes in order to allow the genetic algorithm to turn them on and off and change the overall structure of the network. In this manner, an optimal network size can be found algorithmically instead of through manual trial and error.

Chapter 5: This chapter applies the new genetic algorithm and neural network models found in Chapter 3 and Chapter 4 to intrusion detection. Using the standard data mining IDS benchmark, the KDD99 data set, the GA/NN-based IDS identifies attackers and legitimate users with a very high level of accuracy. The system is tested
extensively and the results are compared to other intrusion detection systems.

Chapter 6: This chapter discusses the implications of the findings in this thesis, and then outlines potential future work that it may lead to.
Chapter 2

Background

A significant amount of research has already been conducted on intrusion detection, particularly in relation to data mining. This chapter begins by explaining the basic concepts behind network security, and then discusses a number of past data mining approaches that have been used for the purposes of intrusion detection. Of particular interest in this discussion will be the genetic algorithm and the neural network, which will be expanded upon in Chapters 3 and 4.

2.1 Network Security

2.1.1 Overview

The advent of wireless technology in recent years has left networks more vulnerable than ever. While a number of effective techniques exist for the prevention of attacks, the nature of the data being transmitted over these networks is often very
sensitive, and therefore calls for more security than an initial barricade can provide alone. In other words, it is unreasonable to simply assume that we can prevent all attackers from infiltrating a network, and we therefore must have some sort of device for the monitoring and tracking of users within the network. This type of system is known as intrusion detection, as opposed to intrusion prevention [49] [66].

2.2 Intrusion Prevention vs. Intrusion Detection

2.2.1 Intrusion Prevention

The distinction between intrusion prevention and detection can be thought of in terms of a simple analogy: a castle town. Intrusion prevention is analogous to the walls, moat, turrets, and drawbridge surrounding the castle, designed to prevent any outsiders from gaining access to the town. In networking terms, intrusion prevention generally involves some sort of authentication (using a password or key) on the users’ part in order for them to gain access to the network (e.g. in wireless networks: TKIP, WEP, WPA, WPA2, etc.) [49].

It is unreasonable to assume that no attackers will ever make it past the wall, moat, turrets, and drawbridge, however, and some sort of contingency plan must be in place. Intrusion detection acts as the guards within the castle walls who seek to identify and remove any unwanted guests. It is extremely important for recovery, as without these guards, attackers can do whatever they please once gaining access to the network.
2.2.2 Intrusion Detection

The issue of intrusion detection is not quite as simple as guards identifying attackers in a castle, unfortunately. Human guards can easily identify attackers based on their dress, behaviour, and a variety of other characteristics that human beings innately recognize. Networks are not nearly as naturally gifted when it comes to identifying users, however, which means that developers and administrators must find ways to “teach” the network how to tell the “good guys” from the “bad guys”. If we consider the analogy outlined above, it is clear that the primary goal in the development of such a tool is to create a network capable of learning and identifying in as “human” a manner as possible [66].

Over the years, extensive research has been conducted on the topic of machine learning. As a subfield of artificial intelligence, machine learning is concerned with the development of systems which allow computers to learn over time. Not surprisingly, such algorithms can be applied to intrusion detection systems in order to teach the network what to look for when monitoring for intruders. The goal is to train the network to the point where it is every bit as competent as the human guards in our castle example, and is able to intuitively identify and eject any attackers.

It is important to consider exactly how the network should be trained, however. Two clear options exist: we can teach the system what all the different types of intruders look like, and then it will be capable of identifying any attackers matching these descriptions in the future, or we can simply show it what normal, good users look like [49] (i.e. inhabitants of the castle), and then the system will kick out any users that do not match these criteria. Both approaches have their benefits and drawbacks, as the first results in very few false alarms, but has difficulty identifying new types of attackers, while the second approach is more flexible in finding new attacks, but will
identify good users as attackers from time to time. These two approaches are defined as follows:

1. **Misuse-Based** [66]: Misuse-based involves the training of the detection system based on known attacker profiles. The attributes of previous attacks are used to create attack signatures. During classification, a user’s behaviour is compared with these signatures, and any matches are considered attacks. A connection that does not fit into any of the attack profiles is deemed to be legitimate. While this approach minimizes false alarms, it is obvious that new forms of attacks may not fit into any category and will erroneously be classified as normal.

2. **Anomaly-Based** [49]: Anomaly-based detection requires the creation of a “good user” profile that is used to measure the “goodness” of subsequent users. Unlike misuse-based detection, no attack profiles are used. Instead, a normal-user profile is created. Any connection falling within this range is deemed to be normal, while any users falling out of line with this signature are considered attacks. This approach is more adaptive in that it is far more likely to catch new types of attacks than misuse-based intrusion detection, but it also leads to a higher false alarm rate. A legitimate user acting in an unusual, but unharmful manner may be labeled as an attack, simply because it does not fit within the normal-user profile.

### 2.3 The Genetic Algorithm

The genetic algorithm models real-world evolutionary practices in order to “evolve” optimal solutions to a given problem. In nature, organisms containing their own genetic code battle the elements and one another for survival, and the fittest
members tend to have the best chance of living on. These fit organisms are then more likely to reproduce and pass on their successful genes to future generations. In this manner, the average fitness of each subsequent generation is generally greater than that of the last. Mutation adds randomness to the process by introducing random change to certain members of a population. These mutations can either be beneficial or detrimental.

The computational version works in much the same manner as its real-world counterpart. Individuals in a population are represented by strings, showing the genetic code of the individual in question. These genetic codes represent potential solutions to a given problem, and their effectiveness is represented by a fitness function which takes into account the accuracy of the solution. The most fit members of the population are the most likely to cross-breed and reproduce offspring carrying on their attributes. Weaker members are less likely to reproduce, and their features are therefore more likely to die out as time goes on. In this manner, subsequent generations tend to be more fit than the previous ones, as the fittest are more likely to survive and pass on their genes. Random mutation exists in order to encourage change and prevent the settling into a local minimum. If properly implemented, the algorithm eventually “evolves” a population of highly fit solutions to the given problem [7] [12] [15] [16] [27] [34] [48] [53] [57] [58] [59] [60].

While many different versions of the genetic algorithm exist (using different fitness functions, breeding mechanisms, etc.), most traditional GAs follow this general pattern:

1. **Generate initial population:** An initial population of size \( N \) is generated randomly.
2. **Evaluate fitness:** Evaluate the fitness of every member of the population based on a fitness function.

3. **Select intermediate population:** Selection of individuals is proportional to fitness. In other words, more fit individuals are more likely to be selected for parenthood. These individuals move into the intermediate population, which is essentially the breeding ground for the next population.

4. **Generate crossover offspring:** Crossover is the mechanism by which parents reproduce in the genetic algorithm. Two individuals are selected from the intermediate population, and a random point is chosen to split both strings apart. The substrings from each are then swapped and recombined (with probability $p_c$) in order to form two new children which each contain attributes from both of their parents. In this manner, successful patterns (known as schema) are preserved and passed down to future generations. An example of crossover for the strings 1101001100101101 and yxyxyxxyxyxxyxy, where the strings are to be broken apart following the fifth digit is:

   $11010 \lor 01100101101$

   $yxyyx \land yxyxyxyxyxy$

   which results in two new child strings, 11010yxyxyxyxyxy and yxyyx01100101101 [60].

5. **Mutate offspring:** Before inserting the crossover-generated offspring into the new population, each bit is mutated (generally with a relatively low probability, known as $p_m$). In other words, each position within the string has a low chance of changing to a different, random value. The offspring are then inserted into the
new generation. This process, along with the crossover operation, is repeated until the new generation is filled.

6. Repeat steps 2-5 until an acceptable solution has been found.

2.4 Neural Networks

While the genetic algorithm described in the previous section does not have much direct use in relation to intrusion detection by itself, it can be used to train powerful feedforward neural networks that are capable of classification in a wide variety of areas. A neural network is a collection of nodes and weighted connections that are meant to loosely represent biologic neurons. Figure 2.1 shows the basic structure of a standard neural network. Input is first passed to the input layer, which contains one node for every input into the network. These nodes then pass their input along their connections to the hidden layer. While neural networks generally have only one input and output layer, there can be any number of hidden layers. The hidden layer accepts input from each of its connections to the input layer, and then applies the following operation:

\[ y_j = K \left( \sum_{i=0}^{n} x_i w_{ji} \right) \]  

(2.1)

where \( y_j \) is the output from the \( j \)th hidden node, \( n \) is the total number of input nodes, \( x_i \) is the \( i \)th input, and \( w_{ji} \) is the weight on the connection between the \( i \)th input and the \( j \)th hidden node. \( K(a) \) is a predefined function which transforms the weighted sum into a given range. The sigmoidal function is often used, since it is a continuous function which transforms any value into a real number between 0 and 1.
In other words, $K(a)$ is often defined by:

$$K(a) = \frac{1}{1 + e^{-a}}$$  \hspace{1cm} (2.2)

Figure 2.2 shows a graphical depiction of the sigmoidal function.

The same process is then repeated in subsequent hidden layers (if there are any) and in the output layer, with the previous layer now acting as the input layer. By using an error function and some sort of training mechanism, the network can eventually be taught to associate given inputs with desired outputs. Inputs and desired output pairs are presented to the network repeatedly, and the amount of error found leads the training mechanism to alter the weights of the connections in a manner that is meant to reduce error in subsequent runs. It is through this alteration of the weights...
Neural networks are simple but very powerful mechanisms which are capable of carrying out a number of prediction, classification, and calculation problems. These networks are trained in a wide variety of supervised and unsupervised manners, including backpropagation and simulated annealing.

As mentioned earlier, the genetic algorithm is also capable of training a neural network. By encoding the weights of the networks into a string, these values can be input into the genetic algorithm as members of the population. Using the inverse of the error rate in the network as the fitness function, the GA then gradually kills off weaker networks and breeds the stronger ones, until an effective, acceptable set of weights has been discovered.

When applied to intrusion detection, the accuracy with which the neural network identifies attackers is used to form the fitness function. The genetic algorithm
then runs until it finds a combination of weights which minimize the number of classification errors made by the neural network [6] [26] [41] [67].

The following chapters present a modified version of the genetic algorithm which is capable of setting not only the connection weights of neural networks for the purposes of intrusion detection, but also their size and structure.

2.5 Other Data Mining Approaches

2.5.1 Overview

Data mining is defined as “the use of data analysis tools to discover previously unknown, valid patterns and relationships in large data sets” [54]. It is essentially the automation of the tasks formally conducted by human analysts and field experts. As data collections continue to increase in complexity and size, it becomes more and more unreasonable to expect these humans to be able to keep up, and so the need for these classification tools has become more and more pressing.

Data mining has become a widely used tool in a number of fields, including business and finance [13], security [39], and medicine [38].

2.5.2 Machine Learning Approaches

Despite the analogies to the human brain made in previous sections, machine learning is really just the intelligent application of statistical processes [29]. The most common types of machine learning algorithms are the following:
• **Supervised learning** [36]: the goal of supervised learning is the creation of a function that is capable of mapping a set of inputs to their corresponding desired outputs. In other words, the classification mechanism is presented with a number of input-output pairs, and parameters are altered until the classifier is capable of accurately mapping the input to the desired output. Neural networks and genetic algorithms are two supervised learning techniques [9] [16]. Other supervised learning algorithms include support vector machines [14], the k-nearest-neighbor classifier [55], and decision trees [63].

• **Unsupervised learning** [20]: unlike supervised learning, unsupervised learning does not work with labeled data. Instead of input-output pairs, these techniques attempt to sort and find structure in a collection of unlabeled data points. Clustering, which is the “process of organizing objects into groups whose members are similar in some way” [64], is the central problem in unsupervised learning.

• **Semi-supervised learning**: a combination of supervised and unsupervised learning. Both labeled and unlabeled data points are used in this method for the creation of the classifier.

2.5.3 **Machine Learning-based IDS Applications**

**Self-Organizing Maps**

Self-organizing maps (SOMs) use unsupervised learning in order to cluster data into meaningful classes. An SOM is a type of neural network in which a number of nodes are used to represent neurons in the human brain. These nodes, or neurons, are initially placed randomly in the centre of a map, with each node having connections
to other nodes. Training then takes place through the use of competitive learning, as training samples are input to the network, and the node with the smallest Euclidean distance from that input is considered to be the winner. In other words, locate the $j$ such that:

$$|x - w_j| \leq |x - w_k| \text{ for all } k \neq j$$  

(2.3)

where $x$ represents the input vector, and $w_i$ represents the $i$th node vector. The winning node then moves closer to the input vector using the following learning rule:

$$\Delta w_j = \alpha (x - w_j)$$  

(2.4)

where $\Delta w_j$ represents the change in node $j$’s position, and $\alpha$ is a small learning rate which prevents drastic changes from being made. Along with the winning node, its neighbours also move closer to the input vector, which results in a stretching of the overall node structure. A neighbour is defined by some threshold $N$, where all prototypes $w_m$ for which $D(m, j) = N$ will be moved along with the winning node. $D(m, j)$ is simply a distance function that measures in terms of distance along the node structure’s indices [42].

After a sufficient number of training trials are presented to the network, the nodes will eventually stretch out into a structure that fits itself around the data set. Figure 2.3 shows an example of how the nodes may look at various stages in the training process. Each node will represent a cluster centroid, and new inputs that are presented to the network will be classified based on their Euclidean distance to these nodes. In other words, the winning node is once again determined based on which neuron is closest to the input vector. In this classification phase, nodes are no longer moved, as training has already been completed. The input vector is instead simply clustered with the winning node. This leads to $n$ different clusters of data, where $n$ is
Figure 2.3: Self-Organizing Map

Four stages of SOM training. (a) Nodes are placed randomly in the centre of the map. (b) and (c) Nodes begin to stretch out based on initial input vectors. (d) Nodes have stretched into a structured pattern after extensive training, and are now ready to classify new inputs.

the number of nodes used in the self-organizing map. In the case of network intrusion detection, these clusters can then be used to classify the input vectors as either being legitimate or illegitimate users [10] [52].

Support Vector Machines

Like any linear classifier, the goal of a support vector machine (SVM) is to find a $p - 1$ dimensional hyperplane to separate two distinct classes in a $p$ dimensional
Figure 2.4: Support Vectors

Three possible decision boundaries that can be drawn between the circles and triangles. Line 2 clearly attains the largest margins, and would therefore be used in an SVM.

feature space (i.e. a line to separate 2D data points, a plane to separate 3D data points, etc.). Support vector machines take this idea one step further, however, as they also stipulate that the decision boundary which provides the maximum margin between classes must be selected. In other words, the technique looks to achieve maximum separation between classes when determining its classification strategy. Figure 2.4 shows an example of three different decision boundaries for a problem, with only one of them achieving this maximum margin separation. The formalization of this idea is as follows:

There is a set of training data with the following form:

$$\{(x_1 c_1) \ (x_2 c_2) \ (x_3 c_3) \ \ldots \ (x_n c_n)\}$$  \hspace{1cm} (2.5)
where \( c_i \) is either +1 or -1, indicating which class the data point \( x_i \) belongs to. Next, we can form any hyperplane as the set of points which satisfies:

\[
\mathbf{w} \cdot \mathbf{x} - b = 0
\]  

(2.6)

where \( \mathbf{w} \) is a normal vector which is perpendicular to the hyperplane, and \( b \) is the offset from the origin. We now need to select the values of \( \mathbf{w} \) and \( b \) which maximize the margin. In other words, we need to find values of these parameters such that two parallel hyperplanes are created which both separate the two data sets, and obtain the greatest separation between one another. These parallel hyperplanes are represented as:

\[
\mathbf{w} \cdot \mathbf{x}_i - b = 1
\]  

(2.7)

and

\[
\mathbf{w} \cdot \mathbf{x}_i - b = -1
\]  

(2.8)

In the case where the two classes are linearly separable (i.e. a hyperplane exists which can completely separate the two classes without error), we simply need to maximize the distance between these two planes. As seen in Figure 2.5, the distance between the two hyperplanes is \( \frac{2}{|\mathbf{w}|} \), which means that we need to minimize \( |\mathbf{w}| \) in order to maximize the total distance [14]. The optimization problem then simply becomes:

Select \( \mathbf{w} \) and \( b \) such that \( |\mathbf{w}| \) is minimized and \( c_i (\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1 \), for all \( 1 \leq i \leq n \).  

(2.9)

Since \( |\mathbf{w}| \) contains an absolute value, it is a non-convex problem and is therefore rather difficult to solve, and so we will instead formulate the rule as follows:

Select \( \mathbf{w} \) and \( b \) such that \( \frac{|\mathbf{w}|^2}{2} \) is minimized and \( c_i (\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1 \), for all \( 1 \leq i \leq n \).

(2.10)
The data points will not always be linearly separable, however. As a result, we can add a slack variable which allows for some errors to be made, but which aims to minimize the number and effect of those errors. The new optimization problem now becomes:

$$\text{Select } w \text{ and } b \text{ such that } \frac{||w||^2}{2} + C \sum \xi_i \text{ is minimized and}$$

$$c_i(w \cdot x_i - b) \geq 1 - \xi_i, \text{ for all } 1 \leq i \leq n.$$  

(2.11)

where $\xi_i$ is a slack variable that measures the degree of misclassification in the data point $x_i$, and $C$ is a constant. By introducing this slack variable, the classifier can now minimize the impact of errors by finding the most successful decision boundary [14].

Other approaches exist for the separation of non-linearly separable data, such as the
use of a kernel function to create a non-linear decision boundary. A kernel function takes a data set and transforms it into a higher dimension through the use of some function (common ones include radial basis functions, Gaussian functions, and sigmoidal functions). The transformed data may become linearly separable in the higher dimension, despite the fact that it remains inseparable in its original space. The result is a decision boundary which appears nonlinear in the data’s original dimension, but which is linear in the higher dimension, and which is capable of separating the two classes [14]. This is a very powerful trick that is used in most realistic, practical implementations of SVMs. In the case of an intrusion detection system, the SVM is used to separate legitimate users from illegitimate users, which then allows for simple classifications based on new input vectors. It should be noted that while one SVM is capable of separating only two distinct classes, the combination of multiple SVMs can successfully separate more than two classes. This allows for more specific classification of attacks, such as “DOS”, “User to Root”, etc., instead of simply “attack” and “legitimate” [33] [61].

**Ant Colony Optimization**

Ant colony optimization (ACO) is, as one would expect, a simulation of real-world ants in a colony. Real ants communicate with one another indirectly through the use of pheromones. Initially, ants wander about randomly searching for food. When an ant finds a food source, it returns to the colony and lays down a pheromone trail along the way. Other ants are attracted to these pheromones, and will therefore follow the trail to the food source. These pheromones fade over time, so shorter trails become more attractive to other ants. As a result, short, optimal paths are eventually formed to the closest food sources to the colony.
In the simulation version, ants wander about a parameter space in search of an optimal solution for a given problem. Like real ants, these agents lay down pheromones in order to direct future ants towards solution paths that worked. Unlike actual ants, however, there is a governing process which is capable of measuring the cost of a given solution, and pheromone updates occur synchronously among ants. Eventually, agents will converge to an optimal path, which is actually capable of changing in real time based on changes in the environment [11]. ACO is a probabilistic method in which ants move from node $i$ to node $j$ with probability $p_{i,j}$, where,

$$p_{i,j} = \frac{(\tau_{ij}^\alpha) (\eta_{ij}^\beta)}{\sum (\tau_{ij}^\alpha) (\eta_{ij}^\beta)}$$

(2.12)

and where $\tau_{ij}$ represents the amount of pheromone that is on arc $ij$, $\alpha$ is a parameter that controls the influence of the pheromones (i.e. $\tau_{ij}$), $\eta_{ij}$ is the desirability of arc $i,j$ (typically $\frac{1}{d_{i,j}}$, where $d_{i,j}$ is the distance from node $i$ to node $j$), and $\beta$ is a parameter that controls the influence of the desirability (i.e. $\eta_{ij}$).

The pheromone update rule is defined by the following:

$$\tau_{ij} = \rho \tau_{ij} + \Delta \tau_{ij}$$

(2.13)

where $\rho$ is the rate of evaporation for pheromones and $\Delta \tau_{ij}$ is the amount of pheromone deposited on the arc $ij$, and the amount of pheromone deposited on a given arc $ij$ is given by:

$$\Delta \tau_{ij}^k = \begin{cases} 
1/L_k & \text{if ant travels on arc } ij \\
0 & \text{otherwise}
\end{cases}$$

(2.14)

where $L_k$ is the cost of the $k$th ant’s search. This cost is generally representative of the distance traveled by an ant, but can also represent some other cost function [11].
After a number of iterations, the ants will eventually converge to one optimal path which can then be used for classification. In the case of intrusion detection, this technique is used to cluster the training samples of legitimate and illegitimate users. The ants then take new input samples to one of the clusters (i.e. the shortest path) based on the pheromone paths that have been laid, and classification is carried out based on the cluster to which the sample is taken [25] [51].
Chapter 3

The Improved Genetic Algorithm

3.1 Introduction

The genetic algorithm is a powerful construct which employs concepts central to evolutionary biology in order to evolve optimal solutions to a given problem. As discussed in Chapter 2, potential applications are quite diverse, ranging from the tuning of a neural network’s parameters for intrusion detection to the prediction of football scores and the solving of complex mathematical equations. While many different versions of the genetic algorithm exist, the work in this chapter will build primarily upon the algorithm discussed in [40]. The reasons for this selection are two-fold: first, this is a relatively new approach presented by Leung et al. that has been shown to achieve better results in less time than the more traditional genetic algorithms; second, the algorithm was specifically designed and tested in [40] for the purposes of training neural network parameters, which will be discussed further in Chapter 4. This algorithm will be called “M-GA” (Modified Genetic Algorithm) throughout this thesis, while the Traditional Genetic Algorithm will be called “T-
GA”, and the newly proposed Improved Genetic Algorithm will be referred to as “I-GA”.

The work completed by Leung et al. is quite impressive when compared to traditional genetic algorithms. The main focus of the new GA (I-GA) is to simplify the process in order to create a more user-friendly algorithm capable of forming more compact neural networks in the coming chapters. Since this thesis is building towards a real-time application that must make quick classifications, the aforementioned compactness and simplicity are of vital importance when considering the modifications to be made to the algorithm. This chapter will first introduce the M-GA, then discuss its primary successes and the areas in which it will be altered, then demonstrate the potential importance and usefulness of the proposed algorithm, and lay out the details of my extensions to the work. The new algorithm has been tested extensively, and results will be presented and analyzed later in the chapter.

3.2 The Modified Genetic Algorithm

3.2.1 Overview

The M-GA employs the same basic concepts as the T-GA introduced in Chapter 2, but tweaks the implementation in hopes of achieving better results. As demonstrated later in the chapter, the M-GA outperforms the T-GA on a number of benchmark functions. The basic algorithm is shown below. The following two sections describe some of the major modifications in greater detail:
begin
    \( \tau = 0 \)  // \( \tau \): number of iterations
    \( \text{initialize} \ P(\tau) \)  // \( P(\tau) \): population for iteration \( \tau \)
    \( \text{evaluate} \ f(P(\tau)) \)  // \( f(P(\tau)) \): fitness function
while (not termination condition) do
    begin
        \( \tau = \tau + 1 \)
        select 2 parents \( p_1 \) and \( p_2 \) from \( P(\tau) \)
        perform crossover operation according to equations 3.5 to 3.8
        perform mutation operation according to equation 3.9 to generate
        three offspring: \( \text{nos}_1, \text{nos}_2, \text{and nos}_3 \)
        // reproduce a new \( P(\tau) \)
        if random number < \( p_a \) \( / / p_a \): probability of acceptance
            The offspring among \( \text{nos}_1, \text{nos}_2, \text{and nos}_3 \) with the largest fitness value
            replaces the least fit member of the population
        else begin
            if \( f(\text{nos}_1) > \) smallest fitness value in the \( P(\tau) \)
                \( \text{nos}_1 \) replaces the least fit member of the population
            end
            if \( f(\text{nos}_2) > \) smallest fitness value in the updated \( P(\tau) \)
                \( \text{nos}_2 \) replaces the least fit member of the population
            end
            if \( f(\text{nos}_3) > \) smallest fitness value in the updated \( P(\tau) \)
                \( \text{nos}_3 \) replaces the least fit member of the population
            end
        Evaluate \( f(P(\tau)) \)
    end
end
Crossover

For each generation, two parent chromosomes are selected based on their fitness value for the crossover operation. This selection process is defined by:

\[
q_i = \frac{f(p_i)}{\sum_{k=1}^{\text{pop.size}} f(p_k)} \tag{3.1}
\]

where \(p_i\) is the \(i\)th chromosome in the current generation, \(f(p_i)\) is the fitness of \(p_i\), \(\text{pop.size}\) is the size of the population, and \(q_i\) is the probability that the \(i\)th chromosome will be selected for the crossover operation. \(p_i\) is defined as:

\[
p_i = [p_{i1}, p_{i2}, \ldots, p_{ij}, \ldots, p_{i\text{no.vars}}] \tag{3.2}
\]

where \(\text{no.vars}\) represents the dimensionality of \(p_i\), \(i = 1, 2, \ldots, \text{pop.size}\), \(j = 1, 2, \ldots, \text{no.vars}\), and:

\[
p_{\text{max}} = [\text{para}_{1\text{max}}, \text{para}_{2\text{max}}, \ldots, \text{para}_{\text{no.vars}}] \tag{3.3}
\]

\[
p_{\text{min}} = [\text{para}_{1\text{min}}, \text{para}_{2\text{min}}, \ldots, \text{para}_{\text{no.vars}}] \tag{3.4}
\]

where \(\text{para}_{1\text{max}}\) and \(\text{para}_{1\text{min}}\) represent the maximum and minimum allowable values for a given \(i\)th parameter (i.e. if \(i\)'s domain was [1..10], then \(\text{para}_{1\text{max}}\) would be 10 and \(\text{para}_{1\text{min}}\) would be 1).

Once two parents, \(p_1\) and \(p_2\), have been selected, four different crossover operations will be performed in order to generate four child chromosomes. The four operators used are defined by:
\[ os_c^1 = [os_1^1, os_1^2, \ldots, os_{n_{vars}}^1] = \frac{P_1 + P_2}{2} \quad (3.5) \]
\[ os_c^2 = [os_1^2, os_2^2, \ldots, os_{n_{vars}}^2] = p_{max}(1 - w) + \max(p_1, p_2)w \quad (3.6) \]
\[ os_c^3 = [os_1^3, os_2^3, \ldots, os_{n_{vars}}^3] = p_{min}(1 - w) + \min(p_1, p_2)w \quad (3.7) \]
\[ os_c^4 = [os_1^4, os_2^4, \ldots, os_{n_{vars}}^4] = \frac{(p_{max} + p_{min})(1 - w) + (p_1 + p_2)w}{2} \quad (3.8) \]

where \( c \) stands for “crossover”, \( os^m_n \) is the \( n \)th crossover-generated offspring, \( os^m_n \) is the \( m \)th element of the \( n \)th crossover-generated offspring, \( 0 \leq w \leq 1 \) and is a predetermined weight, \( \max(p_1, p_2) \) is the vector obtained by taking the maximum of each corresponding element in \( p_1 \) and \( p_2 \), \( \min(p_1, \text{ and } p_2) \) is the vector obtained by taking the minimum of each corresponding element in \( p_1 \) and \( p_2 \).

The purpose of these four modified operators is to spread the search across the entire domain, by stretching the offspring over the diagonal from \( p_{min} \) to \( p_{max} \). The first and fourth operators tend to search near the centre of the domain, while the second and third operators conduct their searches at the domain’s boundaries. The location of these searches can be tuned further through the altering of the \( w \) weight parameter. This allows for a broader range of values to be searched, and prevents settling into local minima. Once the four offspring have been generated, the child with the highest fitness value is selected for mutation.

**Mutation**

In the mutation stage, three new offspring are spawned from the child chosen in the crossover phase (i.e. the crossover child with the highest fitness value). The
mutation for all three is defined by:

\[ \text{nos}_j = [os_1, os_2, \ldots, os_{\text{no}_\text{vars}}] + [b_1 \Delta nos_1, b_2 \Delta nos_2, \ldots, b_{\text{no}_\text{vars}} \Delta nos_{\text{no}_\text{vars}}] \]

(3.9)

where \( j = 1, 2, 3, b_i, \) and \( i = 1, 2, \ldots, \text{no}_\text{vars} \) can only take on a value of 0 or 1, and \( \Delta nos_i, i = 1, 2, \ldots, \text{no}_\text{vars} \) are randomly generated variables such that

\[ \text{para}_i^{\text{min}} \leq os_i + \Delta nos_i \leq \text{para}_i^{\text{max}} \]

The first mutated offspring is obtained by setting one randomly selected \( b_i \) value to 1, and setting the rest to 0. In this manner, just one element of the chromosome is altered. For the second offspring, a random number of \( b_i \) values are set to 1, and once again, the rest are set to 0. In this mutation, a variable number of elements are altered. In the third and final mutation, all values of \( b_i \) are set to 1, which results in all elements being altered.

Once the three mutated chromosomes have been created, a random number between 0 and 1 is generated, and if it is below a certain predetermined threshold, the mutated offspring with the highest fitness value replaces the member of the last generation with the lowest fitness value, regardless of which of the two is more fit. This predetermined threshold represents the probability of accepting a less successful chromosome in order to find a global minimum on the error curve (or a maximum on the fitness curve). If the randomly generated number is larger than this threshold, then each offspring is checked against the least fit member of the population and is only accepted if it has a higher fitness value.
3.2.2 Objectives of the M-GA/I-GA

The primary objective of the M-GA and its improved version, the I-GA, is the development of an improved genetic algorithm which is capable of finding optimal solutions to a variety of different problems with as little human interaction required as possible. The algorithm’s secondary goal is to optimize the structure of neural networks using a switch mechanism. While the goals are not dependent upon one another (a T-GA could be used to train the switch-based network, and the modified genetic algorithm is capable of solving a number of other problems), the paper [40] intertwines them by testing the M-GA on the switch-based neural network.

The overall objective therefore becomes the tuning of both the weights and structure of a neural network using this I-GA with as little human interaction as possible. The human aspect is an important one, because the primary goal is the creation of an algorithm that is capable of self-sufficiently tuning everything about a neural network, and so any dependencies on expert knowledge from the user should be considered unacceptable.

This chapter ignores the neural network component of the algorithm and evaluates only the genetic algorithm portion. The primary objective to consider is therefore the ability for the algorithm to discover optimal solutions for any given problem (not just the tuning of neural networks) more quickly and with less human interaction than previous genetic algorithms.

3.2.3 Successes of the M-GA

The M-GA contains three main modifications: the crossover operation, the mutation phase, and the acceptance probability. All three changes are for the positive,
although, as the next subsection will show, they are not without their limitations either.

The crossover phase creates four different offspring using four very different formulae. Two of these operations search at either end of the possible solution set, and the other two search everything in between. In this manner, parents with high fitness values from previous generations can be combined in multiple ways in order to hone in on areas with high potential. The use of multiple crossover operations is therefore a major strength of the algorithm which allows for a wider variety of offspring to be generated than just one operator would, yet does so in a very strategic manner.

The fittest member of this crossover population is then selected for mutation, which also uses multiple operators in order to generate a variety of children. The mutation phase produces one child with just one mutated element, another child with a random number of changed elements, and a third child with all of its elements altered. The second and third operators are generally most useful in the beginning and middle phases of training, as initial chromosomes are not very fit and require a large amount of change in order to reach their final destination. The first operator simply fine tunes one aspect of the solution, which becomes more useful as the process nears the end and small alterations are preferable. The use of these three operators once again provides flexibility and adaptability, as the algorithm is able to alter its behaviour as the training process moves along.

The final major addition to the M-GA is the inclusion of an acceptance probability which allows for a small percentage of less fit solutions to survive in hopes that it will lead to a more universal search. While the method with which it is implemented is severely flawed (which will be discussed in the following section), the concept is an
important one which helps the algorithm to avoid local minima and eventually find an optimal solution.

### 3.2.4 Limitations of the M-GA

While the paper introduces many interesting ideas that show some definite potential, there are areas in which it can be improved, particularly for the purposes of this thesis. As stated earlier, one of the primary objectives of the M-GA is to take the need for human interaction out of the setting of network structure. Although major steps are made towards that goal, the following list shows three major aspects of the M-GA that can be expanded upon. I will explain the issues in this section, and will then discuss the manner in which they have been modified in the following section.

1. User-defined parameters
2. Static acceptance probability
3. Wasted crossover offspring

**User-defined Parameters**

The first limitation of the M-GA is the fact that the user is required to define parameters before the training process which affect the final result. The most important of these values is a weight variable which is used to define the behaviour of the crossover operations. This weight, $w$, ranges from 0 to 1 and biases the search area of the algorithm towards either the centre or the boundaries of the search space. The four crossover formulae defined in the previous section (equations 3.5-3.8) are used to create the crossover children, and the last three use the $w$ parameter.
While a dynamic value of $w$ would seem appropriate, whereby the weight alters itself towards the fitter solutions, the paper simply uses a static value that must be set ahead of time by the user. This requirement places unneeded pressure on the person using the algorithm, as prior experimentation must be completed in order to obtain an appropriate value for this $w$ variable.

Figure 3.1 and Figure 3.2 show the effect that the alteration of $w$ has on the crossover-generated offspring in the M-GA. It can be seen that while $w$ has no effect on $os^1_c$, larger values of $w$ bring $os^2_c$, $os^3_c$, and $os^4_c$ closer to the centre of the search area formed by the selected parents, while smaller values of $w$ bring them closer to the boundaries, or in the case of offspring #4, the centre of the entire parameter space. While this may seem somewhat trivial in this simple, two-parameter example, high-dimensional cases are far more expansive and require well-tuned parameters in order to hone in on the more promising search areas.

It is important to note that the weight value is set to very specific values in the examples shown in [40]. The M-GA is tested through the approximation of six functions, and the following $w$ values are employed, obtained through experimentation: 0.5, 0.99, 0.1, 0.5, 0.01, and 0.01. Clearly the value ranges greatly (it is not simply the case that most problems use a value near 0.5, for example), and the performance of the algorithm would obviously be greatly affected if a user chose a value of 0.99 for the fifth function shown above (by assuming that it would be similar to the second function), when a value of 0.01 is actually called for. Finding a manner in which this value can be automatically generated by the network would not only make the user’s life easier, but would also ensure better performance on the part of the M-GA.
Figure 3.1: A Geometric Graph of the Meaning of the Crossover Equations

The graph shows a very simple, two-parameter example of the meaning behind the crossover offspring generated by the M-GA. The bounding box represents the search area, based upon $p_{\text{max}}$ and $p_{\text{min}}$. Within the bounding box are the two selected parents, and then the four crossover children generated. Lines between points show possible ranges that change based on the value of $w$. 
Figure 3.2: Meaning of the Crossover Equations - Different Parents
The graph shows the same concept as Figure 3.1, but uses different parents.
Static Acceptance Probability

One of the modifications found in the M-GA is the addition of an acceptance probability which allows for the survival of some less fit solution sets in hopes that they may lead to the discovery of a global minimum for the error rate. A random number is uniformly generated between 0 and 1, and if this number is below the acceptance threshold, the most fit offspring generated in the mutation phase will replace the least fit member of the previous generation, regardless of which one is more fit.

The inclusion of an acceptance probability is actually one of the M-GA’s best additions, but the fact that the threshold remains constant throughout the entire training process is a definite shortcoming. A static acceptance probability implies that it is just as probable that a weak solution will be selected on the last training epoch as it is on the very first, despite the fact that the algorithm has already focused in on its desired minimum at this point, and the acceptance of weaker solutions is actually counterproductive. A threshold which decreases as training progresses, as in simulated annealing [35], will be discussed in the following section.

Wasted Crossover Offspring

The majority of genetic algorithms use some combination of crossover and mutation operations for the creation of new offspring. The exact logistics of these operations is generally what distinguishes one version of the genetic algorithm from the next. In the case of the M-GA, each new generation begins by producing four new
offspring using the crossover operations discussed earlier in the chapter. The most fit of these children is then selected to create three more children using the mutation operations previously outlined. The idea behind such a setup is based on the fact that the crossover operations will produce results using the attributes of earlier generations, and then the mutation phase will add randomness to the equation in order to prevent the algorithm from sinking into local minima. While this is generally true, the M-GA only accepts mutated children into future generations, and completely ignores those generated in the crossover phase. While implementing and testing the M-GA, it was apparent that the most fit member of a generation would often come from the crossover operation, yet only its mutated, less fit offspring would be accepted into future generations.

Clearly the inclusion of a mutation phase is very important, as it provides the randomness that drives evolution, both in the real-world, biological version, and the algorithmic implementation discussed here. It is unreasonable to stipulate that every single generation must produce mutated children however, as more fit, unmutated children created using the crossover operation will often be more beneficial to the population as a whole.

3.3 The Improved Genetic Algorithm

With the targeted areas of the M-GA in mind, a new GA (I-GA) is proposed based on some modifications to the M-GA in this section. Results presented later in the chapter demonstrate that the I-GA is capable of attaining superior results to those produced by the M-GA and T-GA, and with less user-interaction required.
3.3.1 User-defined Parameters

Instead of requiring the user to set the crossover weight value in advance, the I-GA will simply set the value at 0.5 at the start, and will then modify \( w \) after every training epoch. For the first epoch, the direction (positive or negative) that \( w \) first moves is selected randomly. If the fittest crossover offspring found in that generation is more fit than the fittest from the previous generation, the area being searched is clearly promising, and so \( w \) once again moves in that direction. Conversely, if the fittest member of the new generation is less fit than that of the previous epoch, \( w \) reverses directions and begins to backtrack towards the more promising results from earlier. The amount by which \( w \) is altered decreases with every training epoch, which allows the weight to eventually settle into an effective value. In this manner, the parameters can be set to their proper values without the need for any prior experimentation on the part of the user.

In summary, the new weight-setting algorithm is as follows:

1. \( w \), a weight value between 0 and 1, is initially set at 0.5.
2. A random integer (0 or 1) is generated to determine the initial sign for \( \Delta w \). 1 indicates positive, and 0 indicates negative.
3. The initial population is generated.
4. The first generation of crossover offspring is generated using a \( w \) value of 0.5.
5. \( w \) is modified by an amount \( \Delta w \).
6. \(|\Delta w|\) reduces in magnitude. It reduces at a rate that ensures that it will have settled into a final value by 1/10th of the way through training, based on the maximum number of epochs allowed.
7. A new generation of crossover offspring is created using the new \( w \) value. If the most fit member of the new population is more fit than the most fit member of the previous generation, \( \Delta w \) maintains the same sign. If not, \( \Delta w \) switches its sign.

8. \( w \) is modified by an amount \( \Delta w \).

9. \( |\Delta w| \) is reduced in magnitude.

10. Repeat steps 7 through 10 until \( w \) settles into a final value (which occurs when \( \Delta w \) reaches 0, 1/10th of the way through training).

### 3.3.2 Acceptance Probability

The I-GA will remove the static element of the acceptance probability. In other words, the acceptance threshold will start at a certain value, and will then gradually drop as training progresses. This addition ensures randomness early on in the process, but allows for the algorithm to hone in on one single minimum later on in training.

For example, if the I-GA is only being run for a maximum of 500 iterations, and we begin with a 10% probability of accepting a less fit solution, the acceptance probability \( (p_a) \) will drop at a rate that ensures that it will hit 0% by the final epoch (i.e. the 500th). Therefore, the rate at which \( p_a \) falls \( (\Delta p_a) \) will be \( 0.1 / 500 = 0.0002 = 0.02\% \), and the acceptance probability will be altered in the following manner:

- **1st Epoch:** \( p_a = 10\% \)
- **2nd Epoch:** \( p_a = 9.98\% \)
- **3rd Epoch:** \( p_a = 9.96\% \)
100th Epoch: $p_a = 8.00\%$

200th Epoch: $p_a = 6.00\%$

500th Epoch: $p_a = 0\%$

3.4 Performance Comparison

To evaluate the performance of the newly developed I-GA, we use C++ to implement the T-GA, M-GA, and I-GA. Select portions of the I-GA code is provided in Appendix A.

The program is tested using six commonly used benchmark equations [40]. In [40], the authors use these six formulae to show that their M-GA is able to achieve better results in less time than a T-GA. The six test functions, $f_i(\mathbf{x})$, $i = 1, 2, 3, 4, 5, 6$, where $\mathbf{x} = [x_1 \ x_2 \ldots \ x_n]^T$ is a vector with $n$ dimensions, are defined as the following:

$$f_1(\mathbf{x}) = \sum_{i=1}^{n} x_i^2, \quad -5.12 \leq x_i \leq 5.12$$  \hspace{1cm} (3.10)

where $n = 3$ and the minimum point is at $f_1(0, 0, 0) = 0$.

$$f_2(\mathbf{x}) = \sum_{i=1}^{n-1} \left(100(x_{i+1} - x_i)^2 + (x_i - 1)^2\right), \quad -2.048 \leq x_i \leq 2.048$$  \hspace{1cm} (3.11)
where \( n = 2 \) and the minimum point is at \( f_2(0, 0) = 0 \).

\[
f_3(x) = 6n + \sum_{i=1}^{n} \text{floor}(x_i), \quad -5.12 \leq x_i \leq 5.12 \tag{3.12}
\]

where \( n = 5 \) and the minimum point is at \( f_3(-5.12, -5.12, -5.12, -5.12, -5.12) = 0 \). The function \( \text{floor}(y) \) rounds a real number \( y \) down to the nearest integer that is less than \( y \).

\[
f_4(x) = \sum_{i=1}^{n} ix_i^4 + \text{Gauss}(0, 1), \quad -1.28 \leq x_i \leq 1.28 \tag{3.13}
\]

where \( n = 3 \) and the minimum point is at \( f_4(0, 0, 0) = 0 \) and \( \text{Gauss} = \frac{1}{1+e^x} \), with \( \alpha \) being a random real number. This \( \text{Gauss} \) function uniformly generates a floating point number between 0 and 1.

\[
f_5(x) = \frac{1}{k} + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i + a_{ij})^6}, \quad -65.356 \leq x_i \leq 65.356 \tag{3.14}
\]

where \( k = 500 \), \( a \) is defined as:

\[
a = \{a_{ij}\} =
\begin{bmatrix}
-32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 \\
32 & 32 & 32 & 32 & 32 & -16 & -16 & -16 & -16 & -16 \\
& & 0 & 0 & 0 & 0 & 0 & 16 & 16 & 16 & 16 & 32 & 32 & 32 & 32 & 32 & 32 & 32 & 32
\end{bmatrix}
\]

and the maximum point is at \( f_5(-32, -32) \approx 1 \).

\[
f_6(x) = \sum_{i=1}^{n} \left[ x_i^2 - 10 \cos(2\pi x_i) + 10 \right], \quad -5.12 \leq x_i \leq 5.12 \tag{3.15}
\]

where \( n = 3 \) and the minimum point is at \( f_6(0, 0, 0) = 0 \).
Functions $f_1$ through $f_4$ and $f_6$ are all to be minimized, and use the following fitness function:

$$fitness = \frac{1}{1 + f_i(x)}, \quad i = 1, 2, 3, 4, 6$$  \hspace{1cm} (3.16)

Function $f_5$, on the other hand, is to be maximized, and therefore uses the fitness function:

$$fitness = f_5(x)$$  \hspace{1cm} (3.17)

For consistency, the experiments are set up identically to those found in [40]. The I-GA goes through these six test functions, attempting to maximize the fitness functions shown above as quickly as possible. The results are compared with those obtained by the M-GA and T-GA (with arithmetic crossover and nonuniform mutation) [16] [28] [40] [57] [59]. For each test function, the simulation lasts for 500 iterations and the population size is set to ten. For the T-GA, the probability of crossover is set at 0.8 for all functions and the probability of mutation for functions $f_1$ to $f_6$ are 0.8, 0.8, 0.7, 0.8, 0.8, and 0.35, respectively. The M-GA and I-GA do not require those parameters. The shape parameters $b$ of the T-GA for nonuniform mutation, which are selected by trial and error through experimentation, are set at $b = 5$ for $f_1$, $f_2$ and $f_5$, $b = 0.1$ for $f_3$, and $b = 1$ for $f_4$ and $f_6$. This parameter is not required for either the M-GA or I-GA. For the M-GA, the values of $w$ are set to be 0.5, 0.99, 0.1, 0.5, 0.01, and 0.01 for the six test functions, respectively. This value is originally set at 0.5 in the I-GA, and then sets itself based on the algorithm discussed earlier in the chapter. The probability of acceptance is set initially at 0.1 for all functions. The initial values of $x$ are set to be the same for all three versions of the genetic algorithm [40].

For tests 1 to 6, the initial values are $[1 \; 1 \; 1]$, $[0.5 \; 0.5]$, $[1 \; \ldots \; 1]$, $[0.5 \; \ldots \; 0.5]$, $[10 \; \ldots \; 10]$ and $[1 \; 1 \; 1]$, respectively. The results for the six benchmark functions are shown in Figures 3.3 through 3.8. Because results can vary in the genetic algorithms
based on the random setting of parameters, the graphs shows the average results achieved over five separate test runs, thus giving a more accurate and consistent portrayal of the algorithm’s abilities. The I-GA outperforms the others quite significantly for the first five test functions. For the sixth function, it performs far better than the T-GA, but not quite as well as the M-GA. Figure 3.9 shows the value of \( w \) in the I-GA throughout the first test run.

Figure 3.3: Benchmark Function 1
A comparison of the three GAs using the first benchmark function. The I-GA clearly outperforms both the M-GA and T-GA for this function.
Figure 3.4: Benchmark Function 2
A comparison of the three GAs using the second benchmark function. The I-GA significantly outperforms both the M-GA and T-GA for this function.

Figure 3.5: Benchmark Function 3
A comparison of the three GAs using the third benchmark function. The I-GA clearly outperforms both the M-GA and T-GA for this function.
Figure 3.6: Benchmark Function 4
A comparison of the three GAs using the fourth benchmark function. The I-GA significantly outperforms both the M-GA and T-GA for this function.

Figure 3.7: Benchmark Function 5
A comparison of the three GAs using the fifth benchmark function. The I-GA performs comparably to the M-GA and T-GA for this function.
Figure 3.8: Benchmark Function 6
A comparison of the three GAs using the sixth benchmark function. This is the only one of the six benchmark functions in which the M-GA outperforms the I-GA. Both end with a similar final result, but the M-GA attains its solution much more quickly in this case. The I-GA once again outperforms the T-GA.

3.5 Implications of Results

The results obtained using the six benchmark functions outlined in [40] are clearly quite encouraging. In five of the six cases, the I-GA achieved results that were significantly better than those found by the T-GA and Leung et al’s M-GA. In the sixth case, the solution was still found, but it took more iterations than the M-GA. The T-GA was unable to find a correct solution for this particular function, so both modified algorithms outperformed it in this instance. Exact solutions to the first four functions were found within 10 to 30 iterations of the I-GA.

Even more impressive is the fact that the new algorithm was able to achieve these superior results using less human interaction than the other two GAs. As demonstrated in the experimental setup outlined in the previous section, the T-GA and M-GA require a number of different parameters to be set by the human programmer in advance, which is not only time-consuming and demanding on the domain knowl-
Figure 3.9: The Value of the Crossover Weight Parameter
The value of the $w$ parameter using the I-GA’s self-setting approach.
(a) $f_1(x)$; (b) $f_2(x)$; (c) $f_3(x)$; (d) $f_4(x)$; (e) $f_5(x)$; (f) $f_6(x)$.
edge of the user, but also leads to the potential for human error. By eliminating most human-set parameters and implementing the self-setting nature of $w$, the new algorithm is quicker and easier to setup, achieves better results more quickly, and eliminates a significant amount of human error.

Figure 3.9 shows the value of $w$ throughout the experiment. In all cases, the weight parameter hones in on a specific value and slowly settles into it, until it remains stagnant after the 1/10th mark. In some cases, the final values are not the same as those chosen by Leung et al. in their experimental setup. This could be indicative of the fact that they selected incorrect values for their simulations, or simply a sign that some of the problems are capable of achieving good results using a few different $w$ values. Either way, the fact that superior results are achieved using this method of setting $w$ indicates that it is an effective algorithm.

Following the development of this improved GA, the next step is its use for more complicated and useful problems than the simple benchmark equations. The next chapter will propose a new type of neural network that uses this improved genetic algorithm to tune its parameters, connectivity, and size.
Chapter 4

Switch-based Neural Networks

4.1 Introduction

While the I-GA outlined in Chapter 3 is clearly quite effective, its true power is demonstrated when used for the purpose of training neural networks. The examples shown in the previous chapter demonstrated the I-GA finding one-time, final solutions to problems. Although certainly useful in some situations, there are clearly a number of areas that require much more than the minimization of a simple mathematical equation. Problems involving classification or prediction, for example, require a quick and simple construct for the solving of a large number of different inputs. It is here that the power of the neural network comes into play [7] [12] [27] [34] [53] [58] [59]. The I-GA can be used to tune the parameters of a neural network in such a way as to minimize the errors that it makes on a given data set [40]. By inputting the weights of the neural network’s connections to the genetic algorithm, they can be fine-tuned to the point where the network is capable of finding mathematical patterns in the input and seemingly “learning” them in the process.
The concept of tuning a neural network’s connection weights using the genetic algorithm is not a new one, however. While not necessarily the most common approach, it is one that has been used fairly extensively since its creation, as was discussed in Chapter 2. The weights of the connections are input into the GA, and, using a fitness function that involves the error rate of the network, they are tuned until a robust, accurate neural network is achieved.

The genetic algorithm is capable of optimizing more than just the weighting of a given neural network, however. While a less common practice, it is also possible to tune the topology of a network using a GA. By placing switches on connections and/or hidden nodes, these components of the network can be turned on or off, and the size and shape of the network can therefore be determined by the algorithm itself. Despite the fact that some research has been completed on this particular area, the majority of cases see only the weights of neural networks being trained by genetic algorithms, and topology decisions are left strictly to the network creator. This particular practice requires domain knowledge and a time-consuming trial and error approach, however, and is far from ideal.

This chapter will once again use a concept put forth in [40] as a starting point. The paper introduces a neural network with a fixed number of hidden nodes, but switch-based connections, thus allowing for the creation of partially connected neural networks. The following sections will first explain the modifications made by Leung et al. in [40], then discuss the strengths of these design decisions, then once again identify areas to expand upon, and then finally propose an architecture which improves upon their switch-based neural network. The newly proposed neural network will then be trained using the I-GA from Chapter 3, and will be tested using some simple logic problems. Following the notation used in Chapter 3, traditional, non-switch-based neural networks are referred to as “T-NN”, Leung et al.’s modified switch-based
network is called “M-NN”, and the new and improved neural network model is referred to as “I-NN”.

4.2 The Switch-Based Neural Network

4.2.1 Overview

The M-GA has been used to tune the parameters of a feedforward neural network using switches, including the number of connections and the weights of those connections [40]. These switches are simply binary functions that lie on the connections between nodes. In other words, if some parameter \( \alpha \) is greater than or equal to 0, the switch is turned on and a connection exists between the two nodes, but if \( \alpha \) is less than 0, the switch is turned off and the connection does not exist. Let \( \delta \) denote the binary function, defined by:

\[
\delta(\alpha) = \begin{cases} 
0 & \text{if } \alpha < 0 \\
1 & \text{if } \alpha \geq 0
\end{cases}, \quad \alpha \in \mathbb{R} \tag{4.1}
\]

Figure 4.1 shows a general depiction of this network’s structure. The overall input-output relationship of the network, for any output node \( k \), is:

\[
y_k(t) = \sum_{j=1}^{n_h} \delta(s_{jk}^2)w_{jk} \log \sigma \left[ \sum_{i=1}^{n_{in}} \delta(s_{ij}^1)v_{ij}z_i(t) - \delta(s_j^1)b_{j1} \right] - \delta(s_k^2) \log \sigma(b_k^2),
\]

\[
k = 1, 2, \ldots, n_{out} \tag{4.2}
\]

where \( z_i(t), \ i = 1, 2, \ldots, n_{in} \) are inputs which are functions of the variable \( t \), \( n_{in} \) represents the number of inputs, \( n_h \) represents the number of hidden nodes, \( w_{jk}, j = \)
The $v$ and $w$ variables represent weights, the $z$ variables represent the inputs, the $y$ variables the outputs, the $b$ variables are the biases, and the $s$ variables represent the parameters used to turn on and off the switches [40].

$1, 2, \ldots, n_h, k = 1, 2, \ldots, n_{out}$ represents the weight of the connection between the $j$th hidden node and the $k$th output node, $v_{ij}$ represents the weight of the connection between the $i$th input node and the $j$th hidden node, $s^1_{ij}$ represents the parameter of the connection switch from the $i$th input to the $j$th hidden node, $s^2_{jk}$ represents the parameter of the connection switch from the $j$th hidden node to the $k$th output node, $n_{out}$ represents the number of output nodes, $b^1_j$ and $b^2_k$ represent the biases for the hidden and output nodes, respectively, $s^1_j$ and $s^2_k$ represent the parameters of the connection switches of the biases to the hidden and output layers, respectively, and $\log \text{sig}(\cdot)$ denotes the logarithmic sigmoid function:

$$\log \text{sig}(\alpha) = \frac{1}{1 + e^{-\alpha}}$$ (4.3)
By forming chromosomes based on the weight, threshold, and step parameter ($\alpha$) vectors and using the network’s performance rate (i.e. the inverse of the error rate) as a fitness function, it is then possible to tune the neural network with the modified genetic algorithm in order to find the optimal combination of the aforementioned parameters. In [40], Leung et al demonstrate that this algorithm performs as well as, and often better than the normal genetic algorithm, but more notably, it manages to do so using smaller networks. The modified algorithm is capable of eliminating any unnecessary connections, therefore converting an initially large, fully-connected network into a potentially smaller, partially-connected neural network. This results in leaner, simpler networks that are capable of classifying more quickly and with less cost than those tuned using more traditional means.

4.2.2 Objectives of the M-NN/I-NN

As mentioned in Chapter 3, the overall objective of the M-GA/M-NN combination is the tuning of both the weights and structure of a neural network using a modified genetic algorithm with as little human interaction as possible. The human interaction factor is a very important one, since the entire idea behind the algorithm is the creation of a GA that is capable of self-sufficiently tuning everything about a neural network, and so any dependencies on expert knowledge from the user should be considered a minus. The primary advantage of the M-GA and M-NN combination is its simplicity over other similar architectures, and so this lack of complexity becomes a central part of its objective.
4.2.3 Successes of the M-NN

While a number of algorithms are capable of tuning the weights in a neural network, there are few that are able to adjust the network’s size and structure [8] [40] [62]. The inclusion of the switches on the connections of the network, and then the use of the M-GA to turn these switches on or off, is a very significant proposition which vastly increases the genetic algorithm’s ability to create an optimal network. In fact, the main drawback with this element is the fact that the authors did not use it more, which will once again be discussed in the following subsection.

While some other implementations have also been capable of modifying topology and weighting, one of the main advantages of the M-GA/M-NN combination is its simplicity and user friendliness. Yao and Liu create a powerful GA for the purpose of modifying topology and weighting in [62], for example, but the hybrid GA (dubbed as EPNet) uses only mutation and works alongside a backpropagation algorithm for the tuning of the weights in the neural network. As a result, it is more of a hybrid GA/BP implementation than a true genetic algorithm, and is more complex and difficult to set up than Leung et al.’s algorithm as a result.

Other similar topology-based algorithms exist as well [21] [24] [45] [46] [56], including the ANNA ELEONORA algorithm proposed in [43]. Much like [40], the paper introduces a system in which genetic strings represent both the weights and the switches of the network. In [8], another similar algorithm was proposed, in which the number of hidden nodes and connection links for each network is first randomly chosen within a predefined range. Offspring is then generated using three steps: first, the parents are copied, then, the type of mutation to be performed is determined, and then the copy is mutated. Mutation is different for the weight portion of the string than it is for the connection-switch portion.
While all of the aforementioned algorithms are rather similar to the M-GA found in [40], Leung et al.’s algorithm is not only more current, but its simplicity and impressive results make it the obvious choice to build upon.

4.3 Limitations of the M-NN

While Chapter 3 discussed some limitations in the M-GA, it left out any issues that pertained to the training of neural networks. As mentioned above, the M-NN has a lot of potential, but there are some areas that can be further improved. The following list outlines two potential changes that can be made in order to improve the M-NN model. The manner in which they are modified and implemented is then explained later in the chapter.

1. Switch-based network limitations

2. Fitness function limitations

4.3.1 Switch-Based Network Limitations

The switch-based network proposed in the M-NN model is one of the most interesting components of the entire algorithm. However, the switches are located on the connections only, and as a result, can do little more than form partially connected networks. While this can be useful, it would seem obvious that the same type of mechanism could be used on the hidden nodes in order to automate not only the connectivity of the network, but also its size. In other words, the entire topology of the network may be manipulated using this algorithm with some changes to the implementation.
In the M-GA/M-NN, users must set the number of hidden nodes in advance, and then train and test the network for each reasonable number of these nodes, since the algorithm itself is incapable of eliminating any nodes on its own. This once again results in the user being forced to do an unnecessary amount of work in the training process, and it is something that can be improved with an extension to the structure of the network.

4.3.2 Fitness Function Limitations

Like most genetic algorithms, the M-GA uses the inverse of the error rate as its fitness function for determining reproductive and survival probabilities. While this is an effective measure, particularly with regards to obtaining an accurate neural network, there is no incentive for the network to shrink and simplify using this formula. The switch-based network allows for the removal of connections, but without any motivation for it to do so, it is quite possible that the algorithm will simply produce a fully-connected network that looks exactly the same as one generated using the traditional genetic algorithm would, which defeats the entire purpose of the switches in the first place.

The following section will discuss the idea of including the size of the network in the fitness value, which is meant to lead to networks which are both simple and effective.
4.4 Improved Neural Network Model

4.4.1 Switch-Based Hidden Nodes

While the switch-based network is one of the most important and powerful mechanisms proposed in the M-NN, an extension of the principle may make its impact much greater. Instead of only allowing for the removal of connections, switches have been placed on the hidden nodes within the network, thus allowing for the tuning of both connectivity and size. Users no longer have to train and test the network for various numbers of nodes, as one run-through using a rather large number of hidden nodes will allow the algorithm to turn off any unnecessary nodes and find the optimal size on its own.

The node switches behave in the same manner as those found on the connections. A parameter $\alpha$ will be tuned using the I-GA, and will then be compared to a threshold value of 0. If the parameter is greater than or equal to the threshold, the node is turned on; if not, it is switched off. In other words, the node switch $\delta_n$ is defined as:

$$
\delta_n(\alpha) = \begin{cases} 
0 & \text{if } \alpha < 0 \\
1 & \text{if } \alpha \geq 0
\end{cases}
$$

(4.4)

where $\alpha \in \mathbb{R}$, and the connection switch $\delta_c$ is defined as:

$$
\delta_c(\beta) = \begin{cases} 
0 & \text{if } \beta < 0 \\
1 & \text{if } \beta \geq 0
\end{cases}
$$

(4.5)

where $\beta \in \mathbb{R}$. 

57
As a result, the overall input-output relationship of the network, for any output node \( k \), becomes:

\[
y_k(t) = \sum_{j=1}^{n_h} \delta_n(h_j)\delta_c(s^2_{jk})w_{jk}\log \sigma \left[ \sum_{i=1}^{n_in} \delta_c(s^1_{ij})v_{ij}z_i(t) - \delta_c(s^1_{j})b^1_j \right] - \delta_c(s^2_k\log \sigma(b^2_k),
\]

\[k = 1, 2, \ldots, n_{out}\]  \hspace{1cm} (4.6)

where \( h_j \) represents the parameter for the \( j \)th hidden node switch.

### 4.4.2 New Fitness Function

The second major modification to the M-NN does not actually alter the neural network itself, but rather the manner in which the GA trains it. The fitness function of [40] is the same as that used by any T-GA in charge of training a neural network’s weights: the inverse of the error rate. In this manner, neural networks with lower error rates have a better chance of surviving and propagating their successful genes than those that make a number of misclassifications on the training set. While this is obviously a logical method for creating accurate networks, the objective of the algorithm is to not only produce accurate results, but to also reduce the size and complexity of neural networks in the process. Therefore, it logically follows that the size of the network should somehow be included within the fitness function in order to encourage the formation of smaller, less-connected networks.

With that in mind, the previous fitness function of a chromosome \( p_i \), of the form

\[
\text{fitness} = f(p_i) = \frac{1}{1 + \text{error}}
\]
will instead take the form:

\[
\text{fitness} = f(p_i) = \frac{1}{1 + (a)(\text{error}) + (1 - a)(\text{size})}
\]  

(4.7)

where

\[
\text{size} = \frac{N_h + N_c}{N_{\text{max},h} + N_{\text{max},c}}
\]

(4.8)

and \(a, 0 \leq a \leq 1\), is a real number which will be used to determine the emphasis that is placed on minimizing the size of the network, \(N_h\) is the number of hidden node switches turned on, \(N_c\) is the number of connection switches turned on, \(N_{\text{max},h}\) is the total number of hidden nodes (both turned on and off), \(N_{\text{max},c}\) is the total number of connections (both turned on and off), and size is a value between 0 and 1 which represents the size of the network as a percentage of the node and connection switches that are turned on. The maximum number of hidden nodes and connections refers to the number of hidden nodes and total connections in the initial, fully connected network that the algorithm starts off with. This particular measure was chosen for two reasons: (a) higher values indicate a larger network, and (b) its range of values is the same as that of error, which ensures that no extra emphasis is placed on either variable (aside from the setting of \(a\), of course).

The next factor to consider is the value of \(a\). Clearly a value that is too high (i.e. \(a = 1\)) will simply revert the formula back to its original form, ignoring network size in the process. A value of \(a\) that is too low (i.e. \(a = 0\)) is even more problematic though, as highly inaccurate neural networks will be produced by the algorithm. It is unfortunately unfeasible to have the algorithm set the parameter during processing (much like the crossover weight parameter \(w\) was), as a constant fitness function is required in order to accurately compare the strength of neural networks created in different generations. A fitness function which varies greatly from generation to
generation would therefore provide skewed results, and is clearly undesirable. Since
different users have different expectations from the algorithm (i.e. in real time ap-
lications, the user may be more concerned with speed and simplicity than accuracy,
so a lower value of \( a \) would be selected, whereas in an application that is more con-
cerned with accuracy than quickness, a higher value of \( a \) is clearly desirable), it is
therefore necessary to make this parameter a user-defined element of the new genetic
algorithm.

While adding another parameter for the user to set is clearly not helpful in in-
creasing the simplicity of the algorithm, the parameter \( a \) is much easier to understand
than the \( w \) variable discussed in Chapter 3. Even the most experienced users in a
given field would really have no idea how to set \( w \) before running trials to determine
its value. The value of \( a \), on the other hand, relies simply on the user’s preference for
speed and simplicity vs. overall accuracy. This is a classic trade-off in any computa-
tional problem, and this newly introduced parameter allows the user to define his or
her exact needs.

For the purposes of this research, we use the value of 0.8 for the parameter \( a \),
aside from the comparison of results based on different \( a \) values found in the following
section. This provides a fitness function which still primarily stresses the accuracy
of the network, but which also takes into account its size and prevents the GA from
simply selecting a fully connected, large network unless it absolutely has to.

4.5 Performance Comparison

The I-NN model, like the I-GA from Chapter 3, was also implemented in C++. Portions of
the model’s code are provided in Appendix A. A visualization tool was
also created in Visual Basic, which simply takes the output produced by the genetic algorithms and shows a graphic of the neural network that was created. The results of this tool can be seen in the neural network figures shown later in this section.

The algorithm was once again tested against both the M-GA and T-GA, both using the same parameters as defined in the experiments conducted in Chapter 3, with the exception of the crossover \( w \) parameter in the M-GA, which was set to 0.1 in the first example, and 0.3 in the second (based on trial-and-error). All three algorithms begin the training process with one layer of ten hidden nodes. As discussed in the previous section, the parameter \( a \) used in tuning the I-GA’s fitness function is set to 0.8.

Two separate simple logic statements were used to train and test the network. The first test involved the binary XOR (exclusive or) operator, which is actually a linearly inseparable classification problem. In other words, no straight line can be drawn

![Figure 4.2: Exclusive Or (XOR) Operator](image)

The graph shows the four possible outputs from the XOR operator. No straight line can be used to separate the two classes (i.e. true, represented by the triangles, and false, represented by the circles).
between the points of class 1 and class 2 in Figure 4.2 that will ensure 100% accuracy. The XOR operator is defined as follows:

\[
f(x_1, x_2) = x_1 \text{ XOR } x_2
\]  

(4.9)

where \(x_i, i = 1, 2\), are either true or false (i.e. 1 or 0), and \(f(x_1, x_2) = \text{true}\) if and only if one, and only one, of \(x_1\) and \(x_2\) is true. In other words, the operator returns false when both inputs are the same, and true otherwise. The neural network for this function takes two inputs, \(x_1\) and \(x_2\), and outputs one value, \(f(x_1, x_2)\). As a result, from equation 4.6, \(n_{in} = 2\), \(n_h = 10\), and \(k = 1\), and the overall input-output relationship for the network’s only output node is the following:

\[
y(t) = \sum_{j=1}^{10} \delta_n(h_{ij})\delta_c(s_{jk}^2)w_{jk}\log\sigma \left[ \sum_{i=1}^{2} \delta_c(s_{ij}^1)v_{ij}z_i(t) - \delta_c(s_j^1)b_j^1 \right] - \delta_c(s_k^2)\log\sigma(b_k^2),
\]

Figure 4.3 shows the fitness values obtained over 5000 iterations for each of the three GAs (once again based on the average fitness values attained over five test runs in order to offset the effects of randomness). It should be noted that, while the new fitness function was used in the actual processing for the I-GA, this graph just uses

\[
\text{fitness} = \frac{1}{1 + \text{error}}
\]

as its comparison value, simply to provide a consistent measure to compare with. Since there are only four possible sets of inputs to the XOR function (0 and 0, 0 and 1, 1 and 0, and 1 and 1), the neural network is trained with all four on each iteration. As a result, the fitness graph shown indicates the network’s accuracy on all possible inputs, and therefore no additional test set is needed once training is complete.

Figure 4.4 shows the network formed by the I-GA, Figure 4.5 shows the network
Figure 4.3: XOR Performance Results
The results obtained using the I-GA, T-GA, and M-GA to tune neural networks for
the modeling of the XOR operator. The average results of five separate test runs are
used in order to combat the effects of randomness.

formed by the M-GA, and Figure 4.6 shows the network formed by the T-GA (all
for the first test runs). Figure 4.7 compares the number of relevant connections and
hidden nodes in the final versions of the neural networks created by these three GAs.
The term “relevant connections” refers simply to connections in the M-NN that actu-
ally have an effect on the final result. While the I-GA uses a simple clean-up method
at the end of processing to remove any extra connections or nodes that either lead
nowhere or are cut off from any useful input, the M-GA does not possess such a
feature. As a result, some dead-ends exist in the neural networks presented by the
M-NN. While the connections still technically exist for that algorithm, they have no
impact on any final output, and are therefore not to be counted in the graph shown
in Figure 4.7.
Figure 4.4: Neural Network Generated by the I-GA for XOR Problem

Figure 4.5: Neural Network Generated by the M-GA for XOR Problem

Figure 4.6: Neural Network Generated by the T-GA for XOR Problem
Figure 4.7 shows the accuracy of networks formed using \( a \) values of 0.2, 0.4, 0.6, 0.8, and 1.0. As the graph demonstrates, the test runs using \( a \) values of 0.8 and 1.0 produced similar results, with both being more accurate than the other runs. However, Figure 4.9 shows that these higher \( a \) values also led to networks that were generally larger in size than those produced by lower \( a \) values. These results demonstrate the classic tradeoff between simplicity and accuracy discussed earlier in the chapter. For the purposes of this thesis however, \( a = 0.8 \) seems to provide the best compromise between the two factors; yielding results comparable to \( a = 1.0 \), but doing so with much smaller networks. With this in mind, \( a = 0.8 \) for the remainder of the examples discussed.
Figure 4.8: The $a$ Parameter - Accuracy
The results obtained using the I-GA to solve the XOR problem with different values of $a$.

Figure 4.9: The $a$ Parameter - Size
The network sizes obtained using the I-GA to solve the XOR problem with different values of $a$. 
The second formula used to test the neural network model is defined as follows:

\[ f(x_1, x_2, x_3, x_4) = (x_1 \lor x_2) \land (x_3 \lor x_4) \quad (4.10) \]

where \( x_i \), \( i = 1, 2, 3, 4 \), are either true or false (1 or 0). The neural network for this function takes four inputs, \( x_1, x_2, x_3, \) and \( x_4 \), and outputs one value, \( f(x_1, x_2, x_3, x_4) \). As a result, from equation 4.6, \( n_{in} = 4 \), \( n_h = 10 \), and \( k = 1 \), and the overall input-output relationship for the network’s only output node is the following:

\[
y(t) = \sum_{j=1}^{10} \delta_n(h_j)\delta_c(s^2_{jk})w_{jk}\log \text{sig} \left[ \sum_{i=1}^{4} \delta_c(s^1_{ij})v_{ij}z_i(t) - \delta_c(s^1_j)b^1_j \right] - \delta_c(s^2_k)\log \text{sig}(b^2_k),
\]

Figure 4.10 shows the fitness values obtained over 5000 iterations for each of the three GAs, based on the same fitness function used in the previous example. Once again, there is a relatively small number of possible inputs to the formula (sixteen different possibilities in this case), and so the neural network is again trained with all sixteen on each iteration, and the fitness graph shown indicates the network’s accuracy on all possible inputs.

Figure 4.11 shows the network formed by the I-GA, Figure 4.12 shows the network formed by the M-GA, and Figure 4.13 shows the network formed by the T-GA. Figure 4.14 compares the number of relevant connections and hidden nodes in the final versions of the neural networks created by these three GAs. In the first example, the new model provides comparable results to those obtained by the other two GAs, as all three end up finding an optimal solution by the end of training. In the second example, the new model once again outperforms both of the other GAs. More significant, however, is the fact that the I-GA produces networks that are quite a bit smaller than those created by the other two GAs, as demonstrated in Figure 4.7 and Figure 4.14. This finding demonstrates the fact that the modifications made to the
The results obtained using the I-GA, T-GA, and M-GA to tune neural networks for the modeling of the logical formula: $f(x_1, x_2, x_3, x_4) = (x_1 \lor x_2) \land (x_3 \lor x_4)$, where $x_i$, $i = 1, 2, 3, 4$, are either true or false (1 or 0). The average results of five separate test runs are used in order to combat the effects of randomness.

GA and neural network model in this chapter lead to the creation of simpler networks, without sacrificing any accuracy or training time in the process.
4.6 Implications of Results

Although the results obtained from these two tests are once again quite encouraging, they are simple logic equations that don’t show the full potential of the
Figure 4.14: Network Size for Second Logic Example

algorithm. The fact that the I-GA produced smaller networks that gave comparable or, in the case of the second example, better results than the other two GAs is extremely encouraging. Chapter 3 has already demonstrated the power of the I-GA itself, but the modified fitness function presented in this chapter clearly had a major impact on the size of the neural networks produced by the algorithm. The true power of the node switches is not fully realized in these examples, as there is only one hidden layer and one output node, so the switches on the output connections basically have the same effect as the hidden node switches. This addition will come into play more in Chapter 5, when the algorithm is applied to far more complicated problems.

In summary, up to this point, two primary points have now been established:

1. The I-GA is capable of outperforming the M-GA and the T-GA on a variety of different problems, and with less human interaction required.

2. The I-GA and I-NN model combine to form smaller, simpler neural networks which don’t compromise accuracy or training time, making them perfect for
real-time applications.

With this in mind, Chapter 5 will apply the research conducted in Chapters 3 and 4 to a much more complicated classification problem: network intrusion detection.
Chapter 5

Intrusion Detection System

This chapter outlines the implementation of the work laid out in Chapter 3 and Chapter 4 for the purposes of intrusion detection. Using the improved genetic algorithm for the tuning of the size and parameters of the improved neural network, a compact and effective IDS is developed. This intrusion detection system is then tested using the KDD99 data set [2], the standard benchmark data set used in the field of machine learning-based intrusion detection.

5.1 System Design and Implementation

5.1.1 Structure of the New IDS

The newly proposed intrusion detection system is based upon the I-GA/I-NN combination discussed in Chapter 4. The I-GA is used to train the size, connectivity, and parameters of a number of neural networks, which are judged according to
their ability to classify network data. The primary difference in this application of the algorithm is the fact that an extensive data set must be used, both for training and testing. In the examples shown in Chapter 4, the neural networks were simply modeling a mathematical function with only a few inputs, and so the training process was much simpler. In order to train the networks for intrusion detection, extensive network data must first be converted into a form that the neural network will understand, and then compiled into a usable training set. Then, in order to validate the classifying power of the NN, a second data collection containing new entries must be formed and fed to the neural network. This second collection of data will be known as the algorithm’s test set.

Figure 5.1 outlines the processes that the IDS carries out, and Figure 5.2 shows the general structure of the system itself. Descriptions of the main modules and their interactions are given below:

- **Raw Training Data Processing:** A file called “Training Set.txt” is input to the IDS and converted into a multidimensional array. The file contains raw network data from the KDD99 intrusion detection data set [2], which will be examined later in the chapter. This array is the input to the neural network used to attain fitness values.

- **Genetic:** This module represents the improved genetic algorithm, and uses the training data to train neural networks. This module works in the same manner as the I-GA laid out in Chapters 3 and 4.

- **Neural Network Training:** A number of neural networks are trained in order to find the best IDS. The number of errors made on the training set will be used to form the fitness value.
• **NeuralNet**: Identical to the I-NN found in Chapter 4. The neural network consists of input nodes, output nodes, hidden switch-based nodes, and switch-based connections between layers.

• **IDS Testing**: This phase is treated differently in this chapter than in previous ones. Because potential input is infinite in this case, the neural network must be validated to ensure its usefulness. In order to prove that the I-NN is not only successful on the training set, a separate test set with new data is used to test the neural network. This data also comes from the KDD99 set.

• **Network Visualizer**: While the attributes of the chosen neural network are stored in a text file for later use, it is difficult to visualize it in this format. A simple Visual Basic program is therefore developed to take this data and display the neural network as nodes and connections.

---

5.2 **IDS Performance Evaluation**

5.2.1 **Evaluation Data Set**

In order to train the neural networks for intrusion detection, input must be based on actual network connection records. A connection is defined as “a sequence
The IDS acts in a similar manner to the I-GA/I-NN found in Chapter 4. The two major differences are: (1) the requirement for raw network data to be processed for the formation of a training set, and (2) the need for a separate test set for the validation of the neural networks at the conclusion of training. The rest of the process is identical to the algorithm found in previous chapters.
While the actual Genetic and NeuralNet portions of the system are identical to those found in Chapter 4, the rest of the diagram represents newly added components. The processing of raw training data is necessitated by the use of real-world neural network data. The test set portion is required in order to validate the usefulness of the chosen neural network. The network visualizer is a tool that was developed in order to display the final network. It was used in Chapter 4, but will be discussed more in depth in this chapter.
of TCP packets starting and ending at some well defined times, between which data flows to and from a source IP address to a target IP address under some well defined protocol” [3]. In order to obtain accurate results which can be compared to other similar intrusion detection systems, the KDD99 data set [2] is used for evaluation. This set, which is a version of the DARPA98 data collection, is a standard benchmark set used in the intrusion detection field [32] [51] [61].

**DARPA IDS Set**

The first standard benchmark for the evaluation of network intrusion detection systems was collected by The Information Systems Technology Group of MIT Lincoln Laboratory. The project was carried out in 1998 and 1999 by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), and was known as the 1998 DARPA Intrusion Detection Evaluation. Intrusion detection systems were first tested offline using network traffic and audit logs collected on a simulation network and were then delivered to the AFRL for the real-time portion of the evaluation [4]. The data set provided for this project consisted of seven weeks worth of training data and two weeks worth of testing data. These data collections were laid out in the following manner [4]:

Training Data - Weeks 1-2: Initial training and development of the intrusion detection systems is carried out. Attacks and additional traffic types are added until traffic patterns reach a final steady state by the end of this phase.

Training Data - Weeks 3-7: Final training of the intrusion detection systems is carried out. This final stage consists of the addition of variations on attacks and other background traffic.
Test Data - 2 Weeks: Data is collected to test the abilities of the intrusion detection. This data is collected in the same manner as that found in the final five weeks of the training data collection phase.

All attacks found in the training and testing sets fall into four major categories [5]:

- **Denial of Service (DoS):** The attacker floods the network in an attempt to make some resource too busy to handle legitimate requests.

- **User to Root (U2R):** The attacker enters the system as a normal user, but exploits a vulnerability in the network in order to move up to a higher access level.

- **Remote to Local (R2L):** An attacker gains local access to a machine from a remote location without actually having an account on that machine.

- **Probing:** An attacker scopes the network to find any weaknesses that can be used for future attacks (comparable to robbers casing a bank before they rob it).

**KDD99 Data Set**

The KDD99 data set, which uses a version of the 1998 DARPA set, has become the standard benchmark data set for intrusion detection systems [2]. The set consists of a large number of network connections, both normal and attacks. Attacks belong to the four main categories discussed above, but can also be broken down into smaller subcategories. Each connection consists of 41 features which can be broken down into the following four categories [3]:

78
1. **Basic features:**

   This category consists of the basic features common to all network connections, such as duration, the protocol used, the size of the packet, etc. See Table D.2.

2. **Traffic features based on a time window:**

   This section of features contains traffic attributes which cannot be measured using a static snapshot of the connection. DoS and probing attacks are not easily identified, and are characterized by a large number of connections in a very short period of time. As a result, these features are measured using a two-second time window. The window is used to examine the connections in the past two seconds which have the same destination host as that of the current connection, as well as the connections that have the same service as that of the current connection [3]. See Table D.3.

3. **Host-based traffic features:**

   Since many probing attacks scan the hosts (or ports) over a long period of time, a two second time window is not always enough. As a result, the same features shown in the previous section also exist using a window of 100 connections. See Table D.4.

4. **Connection-based content features based on domain knowledge:**

   The final category of features is less obvious than those found in the previous sections, as it is based on domain knowledge and uses features that may or may not be useful in detecting suspicious behaviour. This category is generally most useful in detecting R2L and U2R attacks. See Table D.5.

   Each connection in the KDD99 data set is labeled as either normal or as one of the four attack types described earlier. There are 23 specific types of attacks within these four categories (See Table D.6, Table D.7, Table D.8, and Table D.9).
Table 5.1 shows the distribution of connections in the 10% KDD99 data set, which is the standard data set for the evaluation of intrusion detection systems [65]. It will form the basis of the training and testing data for the experiments in the next section.

Table 5.1: Class Distributions of 10% KDD99 Data Set

<table>
<thead>
<tr>
<th>Class</th>
<th>Number of Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>97277</td>
</tr>
<tr>
<td>DoS</td>
<td>391458</td>
</tr>
<tr>
<td>U2R</td>
<td>52</td>
</tr>
<tr>
<td>R2L</td>
<td>1126</td>
</tr>
<tr>
<td>Probe</td>
<td>4107</td>
</tr>
<tr>
<td>Total</td>
<td>494021</td>
</tr>
</tbody>
</table>

5.2.2 Experimental Setup

The I-GA is most effective when training with relatively small sets of data, as many training iterations are required before the system is ready. As Table 5.1 shows, there are nearly 500,000 total entries in the KDD99 data set, a number which is unnecessarily high for the I-GA. Training would take hours or even days using a set that large, and would not lead to any better results than those yielded by a more concise, compact training set. With that in mind, the training set used here is shown in Figure 5.2. The connections found in this modified training set are all taken from the original KDD99 data set [2]. Similarly, the test set has also been reduced to a more realistic and manageable size, as is shown in Figure 5.3. The training and test sets are based on the distributions used in [65]. As a result, SVM and ACO test results are also taken from [65] and used in the comparison below.
Table 5.2: Modified Training Set

<table>
<thead>
<tr>
<th>Class</th>
<th>Number of Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>200</td>
</tr>
<tr>
<td>DoS</td>
<td>60</td>
</tr>
<tr>
<td>U2R</td>
<td>30</td>
</tr>
<tr>
<td>R2L</td>
<td>60</td>
</tr>
<tr>
<td>Probe</td>
<td>40</td>
</tr>
<tr>
<td>Total</td>
<td>390</td>
</tr>
</tbody>
</table>

Table 5.3: Modified Test Set

<table>
<thead>
<tr>
<th>Class</th>
<th>Number of Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>1000</td>
</tr>
<tr>
<td>DoS</td>
<td>500</td>
</tr>
<tr>
<td>U2R</td>
<td>52</td>
</tr>
<tr>
<td>R2L</td>
<td>1000</td>
</tr>
<tr>
<td>Probe</td>
<td>500</td>
</tr>
<tr>
<td>Total</td>
<td>3052</td>
</tr>
</tbody>
</table>

5.2.3 Testing Results

5.2.4 Support Vector Machines

The SVM classifier used in these experiments implements a one-vs.-all approach. In other words, five different SVMs are trained, and each one labels a connection as either belonging or not belonging to a certain class. A voting scheme is then used to acquire a final classification. If there is no clear-cut winner after the vote, then the connection is labeled as “Unknown”.

The results of the SVM testing are shown in Table 5.4.
5.2.5 Ant Colony Optimization

The only parameter which needs to be set for the ACO algorithm before experimentation is the swarm similarity coefficient $\beta$ [65], which is set to 0.25 for all ACO experiments shown here.

Table 5.5 shows the results of the ACO testing.

5.2.6 Modified Genetic Algorithm

Through trial and error, the M-GA’s $w$ parameter is set to 0.8 in the following experiments. As in Chapter 3 and Chapter 4, the acceptance probability is set to 0.1.
While the KDD99 data set contains 41 features, discrete, text-based features such as protocol, service, and flags cannot be accurately represented by real numbers in a neural network setup. While each possible textual input could be associated with a numerical value, this approach would imply similarities between some inputs that are not necessarily similar at all. As a result, each possible input for these discrete, multi-class features is given its own input node. Only one node for each of these features will be set to 1 for any given input, and the rest will be set to 0. As a result, instead of 41 input nodes, 108 are used in these experiments, for both the M-GA and I-GA. The maximum number of hidden nodes allowed is set to 216. As a result, if every node and connection switch is turned on, there will be a total of 329 nodes and 24629 connections in the network.

Table 5.6 shows the results of the M-GA testing.

<table>
<thead>
<tr>
<th>Classified Class</th>
<th>Normal</th>
<th>DoS</th>
<th>U2R</th>
<th>R2L</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normal</td>
<td>892</td>
<td>47</td>
<td>9</td>
<td>33</td>
<td>19</td>
</tr>
<tr>
<td>DoS</td>
<td>1</td>
<td>483</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>U2R</td>
<td>0</td>
<td>0</td>
<td>36</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>R2L</td>
<td>46</td>
<td>177</td>
<td>216</td>
<td>556</td>
<td>5</td>
</tr>
<tr>
<td>Probe</td>
<td>7</td>
<td>214</td>
<td>15</td>
<td>0</td>
<td>264</td>
</tr>
</tbody>
</table>
5.2.7 Improved Genetic Algorithm

As in Chapter 4, the acceptance probability begins at 0.1 and then gradually decreases as training progresses, and $w$ begins at 0.5. The new fitness function parameter $a$ is once again set to 0.8.

Table 5.6 shows the results of the I-GA testing.

<table>
<thead>
<tr>
<th>Classified Class</th>
<th>Normal</th>
<th>DoS</th>
<th>U2R</th>
<th>R2L</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>909</td>
<td>34</td>
<td>10</td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>DoS</td>
<td>0</td>
<td>485</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>U2R</td>
<td>0</td>
<td>0</td>
<td>41</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>R2L</td>
<td>46</td>
<td>342</td>
<td>16</td>
<td>591</td>
<td>5</td>
</tr>
<tr>
<td>Probe</td>
<td>9</td>
<td>213</td>
<td>15</td>
<td>5</td>
<td>258</td>
</tr>
</tbody>
</table>

5.2.8 Results Comparison

In order to accurately and objectively compare the four algorithms tested above, four more test runs were conducted. Using the five sets of results, the algorithms were then compared using one-factor designs, which “are used to compare several alternatives of a single categorical variable. For example, one could use such a design for comparing several processors, several computer systems, or several caching schemes.” [31]

The measured data consists of $r$ observations for each of the $a$ alternatives. In this
Table 5.8: Results Analysis

<table>
<thead>
<tr>
<th></th>
<th>SVM</th>
<th>ACO</th>
<th>M-GA</th>
<th>I-GA</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_{11}$</td>
<td>$y_{12}$</td>
<td>$y_{13}$</td>
<td>$y_{14}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y_{21}$</td>
<td>$y_{22}$</td>
<td>$y_{23}$</td>
<td>$y_{24}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y_{31}$</td>
<td>$y_{32}$</td>
<td>$y_{33}$</td>
<td>$y_{34}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y_{41}$</td>
<td>$y_{42}$</td>
<td>$y_{43}$</td>
<td>$y_{44}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y_{51}$</td>
<td>$y_{52}$</td>
<td>$y_{53}$</td>
<td>$y_{54}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column sum</td>
<td>$\sum y_{1}$</td>
<td>$\sum y_{2}$</td>
<td>$\sum y_{3}$</td>
<td>$\sum y_{4}$</td>
<td>$\sum y_{..}$</td>
</tr>
<tr>
<td>Column mean</td>
<td>$\bar{y}_{1}$</td>
<td>$\bar{y}_{2}$</td>
<td>$\bar{y}_{3}$</td>
<td>$\bar{y}_{4}$</td>
<td>$\mu = \bar{y}_{..}$</td>
</tr>
<tr>
<td>Column effect</td>
<td>$\alpha_{1} = \bar{y}<em>{1} - \bar{y}</em>{..}$</td>
<td>$\alpha_{2} = \bar{y}<em>{2} - \bar{y}</em>{..}$</td>
<td>$\alpha_{3} = \bar{y}<em>{3} - \bar{y}</em>{..}$</td>
<td>$\alpha_{4} = \bar{y}<em>{4} - \bar{y}</em>{..}$</td>
<td></td>
</tr>
</tbody>
</table>

Case, $r$ is 5 (since 5 test runs have been conducted for each algorithm) and $a$ is 4 (because we are comparing four different algorithms). In Table 5.8 [31], $y_{ij}$ is the $j$th alternative of the $i$th observation.

After computing the mean and column effect in Table 5.8, the standard deviation of errors is calculated as follows [31]:

$$\text{Standard deviation of errors } S_{e} = \sqrt{\frac{\sum_{i=1}^{r} \sum_{j=1}^{a} y_{ij}^2 - ar\mu^2 - r \sum_{j=1}^{a} \alpha_{j}^2}{a(r - 1)}} \quad (5.1)$$

In order to determine the significant difference between two algorithms, we will compute the confidence intervals of the difference between two effect values (for example, $\alpha_{1} - \alpha_{2}$).

The mean value and standard deviation of $\alpha_{1} - \alpha_{2}$ can be computed by the following two equations:

$$\text{Mean value of } \alpha_{1} - \alpha_{2} = \mu_{\alpha_{1} - \alpha_{2}} = \bar{y}_{1} - \bar{y}_{2} \quad (5.2)$$

$$\text{Standard deviation of } \alpha_{1} - \alpha_{2} = \sigma_{\alpha_{1} - \alpha_{2}} = S_{e} \sqrt{\frac{2}{ar}} \quad (5.3)$$
90% Confidence interval for $\alpha_1 - \alpha_2 = \mu_{\alpha_1 - \alpha_2} \pm t_{[1-\delta/2;a(r-1)]} \times \sigma_{\alpha_1 - \alpha_2}$ (5.4)

where $t_{[1-\delta/2;a(r-1)]}$ is the $(1-\delta/2)$-quantile of a t-variate with $n-1$ degrees of freedom, and $\delta$ is the level of significance, defined as:

$$\delta = 1 - \text{confidence}$$ (5.5)

In other words, when trying to determine whether or not an algorithm is superior to another with a certainty of 95%, $\delta$ would be equal to $1 - 0.95 = 0.05$. The confidence intervals are computed using $t$-values read at $a(r - 1)$ degrees of freedom. In this case, we will use a confidence level of 90%, which means that $\delta = 1 - 0.90 = 0.10$ and $t_{[0.95,16]}$ is 1.746 (see Appendix D for a table containing these values). As long as the confidence interval does not contain 0, we can state with 90% confidence that one algorithm is significantly better than the other. Otherwise, no such statement can be made [31] [65].

Comparing Detection Rate

<table>
<thead>
<tr>
<th>Test Run</th>
<th>SVM</th>
<th>ACO</th>
<th>M-GA</th>
<th>I-GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>63.13</td>
<td>75.24</td>
<td>73.1</td>
<td>74.84</td>
</tr>
<tr>
<td>2</td>
<td>65.62</td>
<td>83.94</td>
<td>77.35</td>
<td>79.42</td>
</tr>
<tr>
<td>3</td>
<td>68.43</td>
<td>74.5</td>
<td>72.92</td>
<td>84.2</td>
</tr>
<tr>
<td>4</td>
<td>69</td>
<td>83.35</td>
<td>78.87</td>
<td>81.76</td>
</tr>
<tr>
<td>5</td>
<td>67.33</td>
<td>83.45</td>
<td>74.01</td>
<td>78.14</td>
</tr>
</tbody>
</table>

Column mean | 66.702 | 80.10 | 75.25 | 79.672 |
Mean         | 75.43  | 75.43 | 75.43 | 75.43  |
Column effect| -8.728 | 4.666 | -0.18 | 4.242  |
Standard deviation of errors = 3.481,
Mean value of $\alpha_4 - \alpha_1 = 12.97$,
Mean value of $\alpha_2 - \alpha_4 = 0.424$, 
Mean value of $\alpha_4 - \alpha_3 = 4.422$,
Standard deviation of $\alpha_4 - \alpha_1 = 1.1$, 
Standard deviation of $\alpha_2 - \alpha_4 = 1.1$, 
Standard deviation of $\alpha_4 - \alpha_3 = 1.1$, 

90% Confidence interval for $\alpha_4 - \alpha_1 = 12.97 \pm 1.746 \times 1.1 = (11.048, 14.892)$,
90% Confidence interval for $\alpha_2 - \alpha_4 = 0.424 \pm 1.746 \times 1.1 = (-1.498, 2.346)$,
90% Confidence interval for $\alpha_4 - \alpha_3 = 4.422 \pm 1.746 \times 1.1 = (2.5, 6.344)$.

Therefore, from Table 5.9, the detection rate of the I-GA is significantly better than that of the SVM and M-GA, and comparable to that of the ACO.

Comparing False Positive Rate

<table>
<thead>
<tr>
<th>Test Run</th>
<th>SVM</th>
<th>ACO</th>
<th>M-GA</th>
<th>I-GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.6</td>
<td>2.39</td>
<td>3.54</td>
<td>2.98</td>
</tr>
<tr>
<td>2</td>
<td>6.09</td>
<td>2.68</td>
<td>2.81</td>
<td>2.76</td>
</tr>
<tr>
<td>3</td>
<td>5.8</td>
<td>2.45</td>
<td>3.92</td>
<td>2.32</td>
</tr>
<tr>
<td>4</td>
<td>4.59</td>
<td>3.37</td>
<td>3.01</td>
<td>2.49</td>
</tr>
<tr>
<td>5</td>
<td>5.6</td>
<td>3.34</td>
<td>3.78</td>
<td>2.90</td>
</tr>
<tr>
<td>Column mean</td>
<td>5.536</td>
<td>2.846</td>
<td>3.412</td>
<td>2.69</td>
</tr>
<tr>
<td>Mean</td>
<td>3.621</td>
<td>3.621</td>
<td>3.621</td>
<td>3.621</td>
</tr>
<tr>
<td>Column effect</td>
<td>1.915</td>
<td>-0.775</td>
<td>-0.209</td>
<td>-0.931</td>
</tr>
</tbody>
</table>

Standard deviation of errors = 0.463,
Mean value of $\alpha_1 - \alpha_4 = 2.846$, 

87
Mean value of $\alpha_2 - \alpha_4 = 0.156$,
Mean value of $\alpha_3 - \alpha_4 = 0.722$,
Standard deviation of $\alpha_1 - \alpha_4 = 0.146$,
Standard deviation of $\alpha_4 - \alpha_2 = 0.146$,
Standard deviation of $\alpha_3 - \alpha_4 = 0.146$,
90% Confidence interval for $\alpha_1 - \alpha_4 = 2.846 \pm 1.746 \times 0.146 = (2.59, 3.10)$,
90% Confidence interval for $\alpha_2 - \alpha_4 = 0.156 \pm 1.746 \times 0.146 = (-0.1, 0.412)$,
90% Confidence interval for $\alpha_3 - \alpha_4 = 0.722 \pm 1.746 \times 0.146 = (0.466, 0.978)$.

Therefore, from Table 5.10, the false positive rate of the I-GA is significantly better than that of the SVM and M-GA, and comparable to that of the ACO.

Comparing False Negative Rate

<table>
<thead>
<tr>
<th>Test Run</th>
<th>SVM</th>
<th>ACO</th>
<th>M-GA</th>
<th>I-GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.89</td>
<td>1.5</td>
<td>1.77</td>
<td>1.8</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>0.6</td>
<td>1.64</td>
<td>0.76</td>
</tr>
<tr>
<td>3</td>
<td>23.59</td>
<td>0.3</td>
<td>1.91</td>
<td>0.45</td>
</tr>
<tr>
<td>4</td>
<td>26.13</td>
<td>0</td>
<td>1.12</td>
<td>1.23</td>
</tr>
<tr>
<td>5</td>
<td>15.89</td>
<td>0</td>
<td>1.83</td>
<td>1.12</td>
</tr>
<tr>
<td>Column mean</td>
<td>21.9</td>
<td>0.36</td>
<td>1.654</td>
<td>1.072</td>
</tr>
<tr>
<td>Mean</td>
<td>6.277</td>
<td>6.277</td>
<td>6.277</td>
<td>6.277</td>
</tr>
<tr>
<td>Column effect</td>
<td>15.624</td>
<td>-5.797</td>
<td>-4.623</td>
<td>-5.205</td>
</tr>
</tbody>
</table>

Standard deviation of errors = 2.885,
Mean value of $\alpha_1 - \alpha_4 = 20.828$,
Mean value of $\alpha_4 - \alpha_2 = 0.592$,
Mean value of $\alpha_3 - \alpha_4 = 0.582$, 88
Standard deviation of $\alpha_1 - \alpha_4 = 0.912$,
Standard deviation of $\alpha_4 - \alpha_2 = 0.912$,
standard deviation of $\alpha_3 - \alpha_4 = 0.912$,
90% Confidence interval for $\alpha_1 - \alpha_4 = 20.828 \mp 1.746 \times 0.912 = (19.235, 22.421)$,
90% Confidence interval for $\alpha_4 - \alpha_2 = 0.592 \mp 1.746 \times 0.912 = (-1.001, 2.185)$,
90% Confidence interval for $\alpha_3 - \alpha_4 = 0.582 \mp 1.746 \times 0.912 = (-1.011, 2.175)$.

Therefore, from Table 5.11, the false negative rate of the I-GA is significantly better than that of the SVM, and comparable to that of the ACO and M-GA.

**Results Analysis Summary**

Figure 5.3 shows a summary of the results yielded by the four algorithms in the tests described above. It is clear, based on the one-factor designs carried out in this section, that the I-GA outperforms the M-GA and SVM, and performs quite similarly to the ACO algorithm.

Table 5.12 shows the average size of the neural networks produced by the I-GA and M-GA (rounded to the nearest whole number). Using the new fitness function, the I-GA produces significantly smaller networks than those generated by the M-GA, which is important in a real-time application that begins with a very large network structure (216 hidden nodes and 24629 connections).
5.2.9 Further Validation With a Different Test Set

In order to further validate the improved genetic algorithm, a second test set is also constructed. The distribution of this second set is taken from [65], and is based on the test set used in the KDD99 Cup; a competition in which a number of teams competed to create the most effective IDS using the KDD99 data set. The second test set’s distribution is shown in Table 5.13. This set contains more connections than the one used earlier in the chapter, and is actually a far less realistic representation of everyday network traffic, as far more attacks than normal connections are present.
However, its distribution closely resembles that used in the KDD99 Cup, and these test results therefore allow for a comparison between the Cup winner and the I-GA.

<table>
<thead>
<tr>
<th>Class</th>
<th>Number of Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>10000</td>
</tr>
<tr>
<td>DoS</td>
<td>40000</td>
</tr>
<tr>
<td>U2R</td>
<td>52</td>
</tr>
<tr>
<td>R2L</td>
<td>100</td>
</tr>
<tr>
<td>Probe</td>
<td>400</td>
</tr>
<tr>
<td>Total</td>
<td>50552</td>
</tr>
</tbody>
</table>

Table 5.13: Second Test Set

Table 5.14 shows the I-GA’s performance using the second test set. The detection rate is much higher than in previous tests because of the fact that DoS attacks make up a very large portion of the overall traffic, and the classifier works very well on these types of attacks. Figure 5.4 shows a comparison between the I-GA and the KDD99 Cup winner [22] [65]. Results obtained by the two are actually quite comparable, with the Cup winner having a slight edge in detection rate and false positives, but the I-GA actually having a small advantage in false negatives.

<table>
<thead>
<tr>
<th>Classified Class</th>
<th>Normal</th>
<th>DoS</th>
<th>U2R</th>
<th>R2L</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>9701</td>
<td>79</td>
<td>24</td>
<td>140</td>
<td>56</td>
</tr>
<tr>
<td>DoS</td>
<td>301</td>
<td>37766</td>
<td>74</td>
<td>822</td>
<td>1037</td>
</tr>
<tr>
<td>U2R</td>
<td>1</td>
<td>7</td>
<td>26</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>R2L</td>
<td>2</td>
<td>0</td>
<td>11</td>
<td>79</td>
<td>8</td>
</tr>
<tr>
<td>Probe</td>
<td>2</td>
<td>25</td>
<td>3</td>
<td>172</td>
<td>198</td>
</tr>
</tbody>
</table>

Table 5.14: I-GA Confusion Matrix - Different Test Set
Figure 5.4: Test Results Using a Different Testing Set
Chapter 6

Conclusion and Future Work

6.1 Research Summary

Through modifications to pre-existing genetic algorithm and neural network models, this thesis proposes a new GA and NN combination which is capable of tuning not only the weighting of a neural network, but also its size and connectivity. The new algorithm is shown to converge more quickly than previous algorithms when tested with a collection of benchmark functions. The specific improvements that the I-GA and I-NN models bring are discussed below.

6.1.1 Contributions of the I-GA

1. Reduced User Interaction:

   One of the major goals of this work was the reduction of the need for user interaction. By adding the algorithm for the automatic setting of the $w$ crossover parameter, the I-GA requires less work on the part of the user than the M-GA,
which in turn required less interaction than the T-GA. As a result, the potential for human error is also reduced.

2. **Faster Convergence:**

   As demonstrated by the test results shown in Chapter 3 and Appendix B, the I-GA outperforms the M-GA and T-GA on a wide variety of benchmark functions, not only attaining comparable or superior results, but doing so in fewer iterations than the previous algorithms required.

3. **Improved Acceptance Probability:**

   By introducing concepts used in simulated annealing to the acceptance probability introduced by the M-GA, the I-GA maintains the randomness necessary to avoid local maxima in the fitness curve while still preserving its evolutionary, “survival of the fittest” approach.

### 6.1.2 Contributions of the I-NN

1. **Reduced User Interaction:**

   Like the I-GA, the I-NN also helps to reduce the workload placed on the user. While the majority of neural network models require users to set the number of nodes in the network before training, the introduction of switches to not only the connections, but also to hidden nodes, allows the algorithm to handle this aspect of training all by itself, and thereby frees the user from even more responsibility, and further reduces the potential for human error.

2. **Smaller, Simpler Networks:**

   Using node/connection switches and a new fitness function in the I-GA based on network accuracy, size, and connectivity, the I-GA/I-NN combination is capable
of creating smaller and simpler neural networks than the M-GA or T-GA, which is an important factor to consider when being used in a real-time environment.

6.2 Future Work

1. Input Node Switches:

In this work, switches were introduced to the hidden nodes in order to turn them off. A possible method for feature reduction in the future would be the inclusion of these same switches, only attached to input nodes as well. The I-GA could then turn on and off these inputs in order to determine which features are the most useful and necessary, thereby further reducing network size and complexity.

2. Self-Setting a Parameter:

The inclusion of a new fitness function in the I-GA introduced a new parameter for users to set. This value, $a$, determines the emphasis that is placed on network size versus classification accuracy. At the moment, no algorithm exists in the I-GA for the automatic setting of this variable, but it would be beneficial to find a method for this in the future.

3. Application to Other Domains:

The I-GA/I-NN combination was tested on a collection of benchmark functions and intrusion detection data in this work. Neural networks are very diverse in their potential applications however, so the use of this algorithm for other purposes in the future is the next natural step in determining just how useful and multi-faceted it is.
4. Combination With Other Algorithms:

While the I-GA/I-NN algorithm produces impressive results on its own, its combination with other machine-learning techniques could further increase its potential. Combining it with the SVM and ACO algorithms via a “majority rules” voting scheme is a possible approach, for example.
References


Appendix A

A.1 Code Listing

In this appendix, code is presented for some of the more interesting and vital portions of the algorithm. An extensive amount of code was created for this project, so large, uninteresting sections are omitted from this discussion.

A.1.1 I-NN Code Listing

/*********************************************************
Method: Run()
Type: Double
Purpose: To determine the output from the neural network
        based on the given input.
Inputs: *in - an array containing the network input.
Outputs: A double between 0 and 1 indicating the value of
        the single output node.
*********************************************************/
double NeuralNet::Run(double *in)
{
// An array containing the values calculated by hidden layer
// nodes.
double *net = new double[NUMHNODES];
// An array containing the final values calculated by output
// layer nodes.
double *finalNet = new double[NUMOUTPUT];

// Calculate the net values for the hidden layer neurons.
for (int i = 0; i < NUMHNODES; i++)
{
    net[i] = 1 * hWeights[0][i] * hOn[0][i];
    for (int j = 1; j <= NUMINPUT; j++)
    {
        net[i] += in[j-1] * hWeights[j][i] * hOn[j][i];
    }
    net[i] = Sigmoid(net[i]) * nodeOn[i];
}

// Calculate the net for the final output layer.
for (i = 0; i < NUMOUTPUT; i++)
{
    finalNet[i] = 1 * hWeights[0][i] * hOn[0][i];
    for (int j = 1; j <= NUMHNODES; j++)
    {
        finalNet[i] += net[j-1] * hWeights[j][i] * hOn[j][i];
    }
}
}
finalNet[i] = Sigmoid(finalNet[i]);

// Used when only output node is used. This portion is changed for problems
// involving three or more classes. In that case, a function is called to find
// which output node has the highest value, and then that number is returned.
// The Test and Train methods below are also altered in this case. For
// simplicity’s sake here, we’ll simply examine the one node case, which is used
// in all of the benchmark functions found in Chapters 3 and 4.
double result = finalNet[0];

delete[] net;
delete[] finalNet;

return result;
}

/*********************************************************
Method: Test()
Type: Integer
Purpose: To return the neural network’s classification
(0 if the output is less than 0.5, and 1
otherwise).
Inputs: *in - an array containing the network input.
Outputs: An integer, either 0 or 1, indicating the
network’s final classification.
int NeuralNet::Test(double *in)
{
    double result = Run(in);

    if (result < 0.5)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

Method: Train()
Type: Double
Purpose: To return the amount of error during training.
Inputs: *in - an array containing the network input.
        d - an integer containing the desired output.
Outputs: A double between 0 and 1 indicating the amount
        of error for this input.

double NeuralNet::Train(double *in, int d)
{
    return fabs((double) d - Run(in));
}
A.1.2 I-GA Code

The code for the I-GA is more than 1700 lines long, and is more complex than that of the I-NN, so only two portions of code will be presented: the basic training process and the basic testing phase. This section shows the code used for the IDS, and not for the benchmark functions from Chapter 3 and Chapter 4.

A.1.3 Training Process

```java
for (int i = 0; i < POPULATION_SIZE; i++) {
    for (int k = 0; k < EPOCH_SIZE; k++) {
        // Get next training input.
        in2 = in[k];

        // Keep track of errors for each type (attack or legitimate).
        errors[desired[k]] += (double) population[i]->Train(in2,desired[k]);
        totalClass[desired[k]]++;
    }

    // Calculate average error for each type.
    for (k = 0; k < NUM_TYPES; k++) {
    }
}
```
if (totalClass[k] > 0)
{
    errors[k] = (double) totalClass[k];
}
else
{
    errors[k] = 0;
}

totalErrors += (errors[k] * (1 / (double) NUM_TYPES));
}

nodes = 0;
connects = 0;

for (k = 0; k < NUMHNODES; k++)
{
    nodes += n1[k];
}

for (k = 0; k < NUMCONNECT; k++)
{
    connects += s1[k];
}

fitness[i] = 1 / (1 + (0.8 * totalErrors) + (0.2 * (nodes + connects)))(i);
totalErrors = 0;

for (k = 0; k < NUM_TYPES; k++)
{
    errors[k] = 0;
    totalClass[k] = 0;
}

A.1.4 Testing Process

outfile.open(tempStr);
ifstream infile2 (tempStr);

int numTotal = 0;
int guess = -1;

while (!infile2.eof())
{
    getline (infile2, line);
    SortTestInput(line);

    // Get neural network's classification.
    guess = population[0]->Test(in2);
// Correct
if (guess == desiredTest)
{
    totalCorrect++;
    classCorrect[desiredTest]++;
}

numClass[desiredTest]++;
numTotal++;
umGuesses[guess]++;
}

infile2.close();

// Output results.
outfile << "Total Correct: " << totalCorrect << endl;
outfile << "Total Tested: " << numTotal << endl;
outfile << "Test Accuracy: " << ((double) totalCorrect / (double) numTotal) << endl;
outfile << endl;
outfile << endl;
outfile << "Normal Correct: " << classCorrect[0] << endl;
outfile << "Attack Correct: " << classCorrect[1] << endl;
outfile << endl;
outfile << endl;
outfile << "Total Normal: " << numClass[0] << endl;
outfile << "Total Attack: " << numClass[1] << endl;
outfile << endl;
outfile << endl;
outfile << "Normal Guesses: " << numGuesses[0] << endl;
outfile << endl;
outfile << endl;
outfile << "Normal Accuracy: " << ((double) classCorrect[0] / (double) numClass[0]) << endl;
outfile << "Attack Accuracy: " << ((double) classCorrect[1] / (double) numClass[1]) << endl;

outfile.close();
Appendix B

In this appendix, the M-GA and I-GA will be compared using the six benchmark functions. Based on the results shown in Figures 3.3 through 3.8, this appendix will demonstrate the level of confidence with which it can be said that one algorithm is superior to the other. The two algorithms are compared based on the fitness value after 50 iterations in five separate test runs, and a 90% confidence interval is used in each comparison. [31]

B.1 GA Benchmark Comparison

B.1.1 Benchmark Test 1

The performance differences constitute a sample of five observations, \{0.01, 0, 0.06, 0.039996, 0.03\}. For this sample:

\[
\text{Sample mean} = 0.027999
\]
Table B.1: Benchmark 1 - Fitness Value After 50 Iterations

<table>
<thead>
<tr>
<th>Test Run</th>
<th>I-GA Fitness</th>
<th>M-GA Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.99</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.94</td>
</tr>
<tr>
<td>4</td>
<td>0.999996</td>
<td>0.96</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Sample variance = 0.000569976

Sample standard deviation = 0.02387417

Confidence interval for mean =

\[ 0.027999 \pm t \sqrt{\frac{0.000569976}{5}} = 0.027999 \pm t(0.010676853) \]

The 0.95-quantile of a t-variate with five degrees of freedom is 2.015:

\[ 90\% \text{ confidence interval} = 0.027999 \pm (2.015)(0.010676853) = (0.00648534, 0.04951306) \]

Since the confidence interval does not include zero, we can say with 90% confidence that the I-GA performs better on this benchmark function than the M-GA does over the course of the first 50 iterations.

### B.1.2 Benchmark Test 2

The performance differences constitute a sample of five observations, \{0.258584, 0.141072, 0.172592, 0.137737, 0.206571\}. For this sample:
Table B.2: Benchmark 2 - Fitness Value After 50 Iterations

<table>
<thead>
<tr>
<th>Test Run</th>
<th>I-GA Fitness</th>
<th>M-GA Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.998584</td>
<td>0.74</td>
</tr>
<tr>
<td>2</td>
<td>0.974402</td>
<td>0.83333</td>
</tr>
<tr>
<td>3</td>
<td>0.962592</td>
<td>0.79</td>
</tr>
<tr>
<td>4</td>
<td>0.997737</td>
<td>0.86</td>
</tr>
<tr>
<td>5</td>
<td>0.976571</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Sample mean = 0.183311

Sample variance = 0.002545768

Sample standard deviation = 0.050455603

Confidence interval for mean =

\[ 0.183311 \pm t\sqrt{0.002545768/5} = 0.183311 \pm t(0.022564432) \]

The 0.95-quantile of a t-variate with five degrees of freedom is 2.015:

90% confidence interval =

\[ 0.183311 \pm (2.015)(0.022564432) = (0.00648534, 0.04951306) \]

Since the confidence interval does not include zero, we can say with 90% confidence that the I-GA performs better on this benchmark function than the M-GA does over the course of the first 50 iterations.

B.1.3 Benchmark Test 3
Table B.3: Benchmark 3 - Fitness Value After 50 Iterations

<table>
<thead>
<tr>
<th>Test Run</th>
<th>I-GA Fitness</th>
<th>M-GA Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.93</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.90005</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.97</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.89</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.989</td>
</tr>
</tbody>
</table>

The performance differences constitute a sample of five observations, \{0.07, 0.09995, 0.03, 0.11, 0.011\}. For this sample:

- Sample mean = 0.064190
- Sample variance = 0.001852306
- Sample standard deviation = 0.043038419

Confidence interval for mean =

\[0.064190 \pm t \sqrt{\frac{0.001852306}{5}} = 0.064190 \pm t(0.019247366)\]

The 0.95-quantile of a t-variate with five degrees of freedom is 2.015:

90% confidence interval =

\[0.064190 \pm (2.015)(0.019247366) = (0.025406557, 0.102973443)\]

Since the confidence interval does not include zero, we can say with 90% confidence that the I-GA performs better on this benchmark function than the M-GA does over the course of the first 50 iterations.
B.1.4 Benchmark Test 4

Table B.4: Benchmark 4 - Fitness Value After 50 Iterations

<table>
<thead>
<tr>
<th>Test Run</th>
<th>I-GA Fitness</th>
<th>M-GA Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.6666</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.72</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.773</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.82</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.64</td>
</tr>
</tbody>
</table>

The performance differences constitute a sample of five observations, \{0.3334, 0.28, 0.227, 0.18, 0.36\}. For this sample:

Sample mean = 0.276080

Sample variance = 0.005495932

Sample standard deviation = 0.074134553

Confidence interval for mean =

\[0.276080 \pm t\sqrt{0.005495932/5} = 0.276080 \pm t(0.03315398)\]

The 0.95-quantile of a t-variate with five degrees of freedom is 2.015:

90% confidence interval =

\[0.276080 \pm (2.015)(0.03315398) = (0.20927473, 0.34288527)\]

Since the confidence interval does not include zero, we can say with 90% confidence that the I-GA performs better on this benchmark function than the M-GA does over the course of the first 50 iterations.
B.1.5 Benchmark Test 5

Table B.5: Benchmark 5 - Fitness Value After 50 Iterations

<table>
<thead>
<tr>
<th>Test Run</th>
<th>I-GA Fitness</th>
<th>M-GA Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.36</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.24</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.3333</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.2978</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.3333</td>
</tr>
</tbody>
</table>

The performance differences constitute a sample of five observations, \{0.497426, 0.705227, 0.666675, 0.702175, 0.607001\}. For this sample:

\[
\text{Sample mean} = 0.635701
\]

\[
\text{Sample variance} = 0.007538928
\]

\[
\text{Sample standard deviation} = 0.086827
\]

\[
\text{Confidence interval for mean =}
0.635701 \pm t\left(\frac{0.007538928}{5}\right) = 0.635701 \pm t(0.038830215)
\]

The 0.95-quantile of a t-variate with five degrees of freedom is 2.015:

\[
90\% \text{ confidence interval =}
0.635701 \pm (2.015)(0.038830215) = (0.557457917, 0.713943683)
\]

Since the confidence interval does not include zero, we can say with 90% confidence that the I-GA performs better on this benchmark function than the M-GA does over the course of the first 50 iterations.
B.1.6 Benchmark Test 6

Table B.6: Benchmark 6 - Fitness Value After 50 Iterations

<table>
<thead>
<tr>
<th>Test Run</th>
<th>I-GA Fitness</th>
<th>M-GA Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.332986</td>
<td>0.987</td>
</tr>
<tr>
<td>2</td>
<td>0.319327</td>
<td>0.999</td>
</tr>
<tr>
<td>3</td>
<td>0.27169</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0.256763</td>
<td>0.967</td>
</tr>
<tr>
<td>5</td>
<td>0.33332</td>
<td>1</td>
</tr>
</tbody>
</table>

The performance differences constitute a sample of five observations, \{-0.654014, -0.679673, -0.728310, -0.710237, -0.666680\}. For this sample:

Sample mean = -0.687783

Sample variance = 0.000949518

Sample standard deviation = 0.030814258

Confidence interval for mean =

\[-0.687783 \pm t\sqrt{(0.000949518/5)} = -0.687783 \pm t(0.013780555)\]

The 0.95-quantile of a t-variate with five degrees of freedom is 2.015:

90% confidence interval =

\[-0.687783 \pm (2.015)(0.013780555) = (-0.715550618, -0.660014982)\]

Since the confidence interval is less than zero, we can say with 90% confidence that the M-GA performs better on this benchmark function than the I-GA does over the course of the first 50 iterations.
B.1.7 Summary of Results

The I-GA is proven to outperform the M-GA (over the course of the first 50 iterations) in benchmark functions 1 through 5, but is outperformed by the M-GA in the sixth benchmark.

B.2 NN Benchmark Comparison

B.2.1 Benchmark Test 1

<table>
<thead>
<tr>
<th>Test Run</th>
<th>I-GA/I-NN Fitness</th>
<th>M-GA/M-NN Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.973695</td>
<td>0.934</td>
</tr>
<tr>
<td>2</td>
<td>0.999</td>
<td>0.946</td>
</tr>
<tr>
<td>3</td>
<td>0.959</td>
<td>0.937</td>
</tr>
<tr>
<td>4</td>
<td>0.96666</td>
<td>0.91</td>
</tr>
<tr>
<td>5</td>
<td>0.97</td>
<td>0.9266</td>
</tr>
</tbody>
</table>

The performance differences constitute a sample of five observations, \{0.039695, 0.053000, 0.022000, 0.056660, 0.043400\}. For this sample:

Sample mean = 0.042951

Sample variance = 0.000184667

Sample standard deviation = 0.013589211
Confidence interval for mean =

\[ 0.042951 \mp t \sqrt{(0.000184667/5)} = 0.042951 \mp t(0.00607728) \]

The 0.95-quantile of a t-variate with five degrees of freedom is 2.015:

90% confidence interval =

\[ 0.042951 \mp (2.015)(0.00607728) = (0.030705281, 0.055196719) \]

Since the confidence interval does not include zero, we can say with 90% confidence that the I-GA/I-NN combination performs better on this benchmark function than the M-GA/M-NN does over the course of 5000 iterations.

### B.2.2 Benchmark Test 2

#### Table B.8: Benchmark 2 - Fitness Value After 5000 Iterations

<table>
<thead>
<tr>
<th>Test Run</th>
<th>I-GA/I-NN Fitness</th>
<th>M-GA/M-NN Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.9696556</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.964</td>
<td>0.99</td>
</tr>
<tr>
<td>3</td>
<td>0.99</td>
<td>0.97</td>
</tr>
<tr>
<td>4</td>
<td>0.96666</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0.975</td>
<td>0.9999</td>
</tr>
</tbody>
</table>

The performance differences constitute a sample of five observations, \{-0.030344, -0.026000, 0.020000, -0.033340, -0.024900\}. For this sample:

Sample mean = 0.000484777

Sample variance = 0.022017643
Sample standard deviation = 0.009846589

Confidence interval for mean =

\[ 0.000484777 \pm t \sqrt{\frac{0.022017643}{5}} = 0.000484777 \pm t(0.00607728) \]

The 0.95-quantile of a t-variate with five degrees of freedom is 2.015:

90% confidence interval =

\[ 0.000484777 \pm (2.015)(0.00607728) = (-0.038757758, 0.000923998) \]

Since the confidence interval includes zero, we cannot say that either the I-GA/I-NN or the M-GA/M-NN perform better than the other over the course of 5000 iterations with a confidence of 90

### B.2.3 Summary of Results

The I-GA/I-NN is proven to outperform the M-GA/I-NN in the first benchmark function, but neither function is appreciably better than the other in the second benchmark. Even comparable results between the two are still impressive, as the I-GA/I-NN combination attains similar results with much less human interaction, and manages to set the network size and connectivity by itself in the process, while the M-GA/M-NN was only able to set the connectivity on its own.
Appendix C

In this appendix, various phases of the I-GA/I-NN algorithms will be validated using a simple, two-dimensional case. The purpose of this exercise is to test and validate that individual sections of the code work as they are expected to.

C.1 Data Processing

Input data:

Table C.1: Sample Training Data

<table>
<thead>
<tr>
<th>feature 1</th>
<th>feature 2</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Parameters:

Maximum number of iterations: 0
Number of training samples: 4
Number of data dimensions: 2
Number of classes: 2
Normalization: Off

Results:

Table C.2: Resulting Training Array

<table>
<thead>
<tr>
<th>in[0]</th>
<th>in[1]</th>
<th>desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

The processing of training data is carried out as expected.

C.2 Training Phase

Input data:

The data set shown in Table C.1 will be used to train the I-GA.
Parameters:

Maximum number of iterations: 100
Desired fitness: 0.99
Number of training samples: 4
Number of data dimensions: 2
Number of classes: 2

Results:

Number of iterations: 34
Errors: 0.000624639
Fitness: 0.999376

<table>
<thead>
<tr>
<th>in[0]</th>
<th>in[1]</th>
<th>desired</th>
<th>Training Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The training phase works as expected.
C.3 Testing Phase

Testing data:

<table>
<thead>
<tr>
<th>Table C.4: Test Set 1 - Correct labeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature 1</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table C.5: Test Set 2 - Incorrect Labeling - 0 becomes 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature 1</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table C.6: Test Set 3 - Incorrect Labeling - 1 becomes 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature 1</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
Parameters:

Maximum number of iterations: 100
Desired fitness: 0.99
Number of training samples: 4
Number of test samples: 4
Number of data dimensions: 2
Number of classes: 2

Results:

<table>
<thead>
<tr>
<th>in[0]</th>
<th>in[1]</th>
<th>desired</th>
<th>Test Result</th>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Accuracy = 100%
False Positive Rate = 0%
False Negative Rate = 0%
Table C.8: Test 2 Results

<table>
<thead>
<tr>
<th>in[0]</th>
<th>in[1]</th>
<th>desired</th>
<th>Test Result</th>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Accuracy = 75%
False Positive Rate = 0%
False Negative Rate = 25%

Table C.9: Test 3 Results

<table>
<thead>
<tr>
<th>in[0]</th>
<th>in[1]</th>
<th>desired</th>
<th>Test Result</th>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Accuracy = 75%
False Positive Rate = 25%
False Negative Rate = 0%

The testing phase works as expected.
Appendix D

D.1 Quantities of the \( t \) Distribution

Table D.1 lists the values of \( t_{[p,n]} \), based on the \( t \)-distribution [31], where \( p \) represents probability and \( n \) represents degrees of freedom.

<table>
<thead>
<tr>
<th>( n )</th>
<th>0.60</th>
<th>0.75</th>
<th>0.90</th>
<th>0.95</th>
<th>0.96</th>
<th>0.975</th>
<th>0.98</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.325</td>
<td>1.000</td>
<td>3.078</td>
<td>6.314</td>
<td>7.916</td>
<td>12.706</td>
<td>15.894</td>
<td>31.821</td>
</tr>
<tr>
<td>2</td>
<td>0.289</td>
<td>0.816</td>
<td>1.886</td>
<td>2.920</td>
<td>3.320</td>
<td>4.303</td>
<td>4.849</td>
<td>6.965</td>
</tr>
<tr>
<td>3</td>
<td>0.277</td>
<td>0.765</td>
<td>1.638</td>
<td>2.353</td>
<td>2.605</td>
<td>3.182</td>
<td>3.492</td>
<td>4.541</td>
</tr>
<tr>
<td>4</td>
<td>0.271</td>
<td>0.741</td>
<td>1.533</td>
<td>2.132</td>
<td>2.333</td>
<td>2.776</td>
<td>2.999</td>
<td>3.747</td>
</tr>
<tr>
<td>5</td>
<td>0.267</td>
<td>0.727</td>
<td>1.476</td>
<td>2.015</td>
<td>2.191</td>
<td>2.571</td>
<td>2.757</td>
<td>3.365</td>
</tr>
<tr>
<td>6</td>
<td>0.265</td>
<td>0.718</td>
<td>1.440</td>
<td>1.943</td>
<td>2.048</td>
<td>2.447</td>
<td>2.612</td>
<td>3.143</td>
</tr>
<tr>
<td>7</td>
<td>0.263</td>
<td>0.711</td>
<td>1.415</td>
<td>1.895</td>
<td>2.006</td>
<td>2.365</td>
<td>2.517</td>
<td>2.998</td>
</tr>
<tr>
<td>8</td>
<td>0.262</td>
<td>0.706</td>
<td>1.397</td>
<td>1.860</td>
<td>1.964</td>
<td>2.262</td>
<td>2.398</td>
<td>2.821</td>
</tr>
<tr>
<td>9</td>
<td>0.261</td>
<td>0.703</td>
<td>1.383</td>
<td>1.833</td>
<td>1.933</td>
<td>2.132</td>
<td>2.303</td>
<td>2.639</td>
</tr>
<tr>
<td>10</td>
<td>0.260</td>
<td>0.700</td>
<td>1.372</td>
<td>1.812</td>
<td>1.912</td>
<td>2.048</td>
<td>2.235</td>
<td>2.568</td>
</tr>
<tr>
<td>11</td>
<td>0.260</td>
<td>0.697</td>
<td>1.363</td>
<td>1.796</td>
<td>1.899</td>
<td>2.006</td>
<td>2.132</td>
<td>2.474</td>
</tr>
<tr>
<td>12</td>
<td>0.259</td>
<td>0.695</td>
<td>1.356</td>
<td>1.782</td>
<td>1.873</td>
<td>1.912</td>
<td>1.912</td>
<td>1.812</td>
</tr>
<tr>
<td>13</td>
<td>0.259</td>
<td>0.694</td>
<td>1.350</td>
<td>1.771</td>
<td>1.859</td>
<td>1.899</td>
<td>1.873</td>
<td>1.782</td>
</tr>
<tr>
<td>14</td>
<td>0.258</td>
<td>0.692</td>
<td>1.345</td>
<td>1.761</td>
<td>1.845</td>
<td>1.873</td>
<td>1.859</td>
<td>1.771</td>
</tr>
<tr>
<td>15</td>
<td>0.258</td>
<td>0.691</td>
<td>1.341</td>
<td>1.753</td>
<td>1.833</td>
<td>1.845</td>
<td>1.825</td>
<td>1.749</td>
</tr>
<tr>
<td>16</td>
<td>0.258</td>
<td>0.690</td>
<td>1.337</td>
<td>1.746</td>
<td>1.825</td>
<td>1.833</td>
<td>1.812</td>
<td>1.734</td>
</tr>
</tbody>
</table>
D.2 KDD99 Data Set

The following tables show the distribution of both features and attack-types in the KDD99 data set [2] [3] [65].

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>Length (number of seconds) of the connection.</td>
<td>continuous</td>
</tr>
<tr>
<td>protocol_type</td>
<td>Type of protocol used.</td>
<td>discrete</td>
</tr>
<tr>
<td>service</td>
<td>Network service on the destination.</td>
<td>discrete</td>
</tr>
<tr>
<td>src_bytes</td>
<td>Number of data bytes from source to destination.</td>
<td>continuous</td>
</tr>
<tr>
<td>dst_bytes</td>
<td>Number of data bytes from destination to source.</td>
<td>continuous</td>
</tr>
<tr>
<td>flag</td>
<td>Normal or error status of the connection.</td>
<td>discrete</td>
</tr>
<tr>
<td>land</td>
<td>1, if connection is from/to the same host/port; 0, otherwise.</td>
<td>discrete</td>
</tr>
<tr>
<td>wrong_fragment</td>
<td>Number of incorrect fragments.</td>
<td>continuous</td>
</tr>
<tr>
<td>urgent</td>
<td>Number of urgent packets.</td>
<td>continuous</td>
</tr>
</tbody>
</table>
Table D.3: Traffic Features Using Two-Second Time Windows

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of connections in the past two seconds that connect to the same host as the current connection.</td>
<td>continuous</td>
</tr>
<tr>
<td>serror_rate</td>
<td>% of connections that have SYN errors among the same destination host connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>rerror_rate</td>
<td>% of connections that have REJ errors among the same destination host connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>same_srv_rate</td>
<td>% of connections that get the same service as current connection among the same destination host connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>diff_srv_rate</td>
<td>% of connections that get different services from current connection among the same destination host connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>srv_count</td>
<td>% number of connections that have the same service as the current connection in the past two seconds.</td>
<td>continuous</td>
</tr>
<tr>
<td>srv_serror_rate</td>
<td>% of connections that have SYN errors among the same service connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>srv_rerror_rate</td>
<td>% of connections that have REJ errors among the same service connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>srv_diff_host_rate</td>
<td>% of connections that connect to different hosts from the current connection among the same service connections.</td>
<td>continuous</td>
</tr>
</tbody>
</table>
Table D.4: Host Based Traffic Features

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of connections that connect to the same host as the current connection within the window of 100 connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>serror_rate</td>
<td>% of connections that have SYN errors among the same destination host connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>rerror_rate</td>
<td>% of connections that have REJ errors among the same destination host connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>same_srv_rate</td>
<td>% of connections that get the same service as the current connection among the same destination host connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>diff_srv_rate</td>
<td>% of connections that get different services from current connection among the same destination host connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>srv_count</td>
<td>% number of connections that have the same service as the current connection within the window of 100 connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>srv_serror_rate</td>
<td>% of connections that have SYN errors among the same service connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>srv_rerror_rate</td>
<td>% of connections that have REJ errors among the same service connections.</td>
<td>continuous</td>
</tr>
<tr>
<td>srv_diff_host_rate</td>
<td>% of connections that connect to different hosts among the same service connections.</td>
<td>continuous</td>
</tr>
</tbody>
</table>
Table D.5: Connection-Based Content Features Based on Domain Knowledge

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hot</td>
<td>Number of hot indicators.</td>
<td>continuous</td>
</tr>
<tr>
<td>num_failed_logins</td>
<td>Number of failed login attempts.</td>
<td>continuous</td>
</tr>
<tr>
<td>logged_in</td>
<td>1, if successfully logged in; 0, otherwise.</td>
<td>discrete</td>
</tr>
<tr>
<td>num_compromised</td>
<td>Number of compromised conditions.</td>
<td>continuous</td>
</tr>
<tr>
<td>root_shell</td>
<td>1, if root shell is obtained; 0, otherwise.</td>
<td>discrete</td>
</tr>
<tr>
<td>su_attempted</td>
<td>% 1, if su root command attempted; 0, otherwise.</td>
<td>discrete</td>
</tr>
<tr>
<td>num_root</td>
<td>Number of root accesses.</td>
<td>continuous</td>
</tr>
<tr>
<td>num_file_creatons</td>
<td>Number of file creation operations.</td>
<td>continuous</td>
</tr>
<tr>
<td>num_shells</td>
<td>Number of shell prompts.</td>
<td>continuous</td>
</tr>
<tr>
<td>num_access_files</td>
<td>Number of operations on access control files.</td>
<td>continuous</td>
</tr>
<tr>
<td>num_outbound_cmds</td>
<td>Number of outbound commands in an ftp. session.</td>
<td>continuous</td>
</tr>
<tr>
<td>is_hot_login</td>
<td>1, if the login belongs to the hot list; 0, otherwise.</td>
<td>discrete</td>
</tr>
<tr>
<td>is_guest_login</td>
<td>1, if the login is a guest login; 0, otherwise.</td>
<td>discrete</td>
</tr>
</tbody>
</table>

Table D.6: DoS Attack Types

<table>
<thead>
<tr>
<th>Attack Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>back</td>
<td>Attack against apache web server where a client requests a URL containing many backslashes.</td>
</tr>
<tr>
<td>land</td>
<td>Denial of service where a remote host is sent a UDP packet with the same source and destination.</td>
</tr>
<tr>
<td>neptune</td>
<td>Syn flood denial of service on one or more ports.</td>
</tr>
<tr>
<td>pod</td>
<td>Denial of service ping of death.</td>
</tr>
<tr>
<td>smurf</td>
<td>Denial of service icmp echo reply flood.</td>
</tr>
<tr>
<td>teardrop</td>
<td>Denial of service where mis-fragmented UDP packets cause some systems to reboot.</td>
</tr>
</tbody>
</table>
### Table D.7: R2L Attack Types

<table>
<thead>
<tr>
<th>Attack Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftp_write</td>
<td>Remote FTP user creates .rhost file in anonymous FTP directory and obtains local login.</td>
</tr>
<tr>
<td>guess_passwd</td>
<td>Try to guess password via telnet for guest account.</td>
</tr>
<tr>
<td>imap</td>
<td>Remote buffer overflow using imap port leads to root shell.</td>
</tr>
<tr>
<td>multihop</td>
<td>Multi-day scenario in which a user first breaks into one machine.</td>
</tr>
<tr>
<td>phf</td>
<td>Exploitable CGI script which allows a client to execute arbitrary commands on a machine with a misconfigured web server.</td>
</tr>
<tr>
<td>spy</td>
<td>Multi-day scenario in which a user breaks into a machine with the purpose of finding important information where the user tries to avoid detection. Use several different exploit methods to gain access.</td>
</tr>
<tr>
<td>warezclient</td>
<td>Users downloading illegal software which was previously posted via anonymous FTP by the warezmaster.</td>
</tr>
<tr>
<td>warezmaster</td>
<td>Anonymous FTP upload of Warez (usually illegal copies of copyrighted software) onto FTP server.</td>
</tr>
</tbody>
</table>

### Table D.8: U2R Attack Types

<table>
<thead>
<tr>
<th>Attack name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer_overflow</td>
<td>Buffer overflow using various methods leads to root shell.</td>
</tr>
<tr>
<td>loadmodule</td>
<td>Resets IFS for a normal user and creates a root shell.</td>
</tr>
<tr>
<td>perl</td>
<td>Sets the user ID to root in a perl script and creates a root shell</td>
</tr>
<tr>
<td>rootkit</td>
<td>Multi-day scenario where a user installs one or more components of a rootkit.</td>
</tr>
<tr>
<td>Attack Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ipsweep</td>
<td>Surveillance sweep performing either a port sweep or ping on multiple host addresses.</td>
</tr>
<tr>
<td>nmap</td>
<td>Network mapping using the nmap tool. Mode of exploring network will vary-options include SYN.</td>
</tr>
<tr>
<td>portsweep</td>
<td>Surveillance sweep through many ports to determine which services are supported on a single host.</td>
</tr>
<tr>
<td>satan</td>
<td>Network probing tool which looks for well-known weaknesses. Operate at three different levels. Level 0 is light.</td>
</tr>
</tbody>
</table>