Deconstructing a Neural Network

BY

Patrick Moghames

DigiPen Institute of Technology

5001-150th Ave NE

Redmond, WA USA

E-mail: pmoghames@digipen.edu

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the graduate studies program
of DigiPen Institute of Technology
Redmond, Washington
United States of America

Fall
2007

Thesis Advisor: Dr. Bikramjit Banerjee

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDY PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS OF PATRICK MOGHAMES

HAS BEEN SUCCESSFULLY COMPLETED ON NOVEMBER 27TH 2007

TITLE OF THESIS: PLAYER MODELING USING KNOWLEDGE TRANSFER

MAJOR FILED OF STUDY: COMPUTER SCIENCE.

COMMITTEE:

Michael Aristidou, Chair	Samir Abou Samra
Christopher Erhardt	Dimitri Volper

APPROVED:

Xin Li date Director Graduate Program	Matt Klassen date Dean of Faculty
Samir Abou Samra date Chair of Computer Science Department	Claude Comair date President of DigiPen

THE MATERIAL PRESENTED WITHIN THIS DOCUMENT DOES NOT NECESSARILY REFLECT THE OPINION OF THE COMMITTEE, THE GRADUATE PROGRAM, OR DIGIPEN INSTITUTE OF TECHNOLOGY.

INSTITUTE of DigiPen Institute Of Technology
PROGRAM OF MASTER'S DEGREE

THESIS APPROVAL

Date: November 27th 2007

Based on the CANDIDATE'S successful oral defense, it is recommended that the thesis prepared by

Patrick Moghames

ENTITLED

Deconstructing a Neural Network

Be accepted in partial fulfillment of the requirements for the degree of master of computer science from the program of Master's degree at DigiPen Institute Of Technology.

Dr. Michael Aristidou
Thesis Advisory Committee Chair

Dr. Xin Li
Director of Graduate Program

Dr. Matt Klassen,
Dean of Faculty

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology

Table of Contents

1 Acknowledgements

I would like to thank my advisor Dr. Bikramjit Banerjee in guiding me to reach the results that I have today. I would also like to thank all my teachers that supported me during my college years. A special thank you goes out to Dr. Michael Aristidou and Dr. Dimitri Volper for their help in finalizing the paper.

The biggest thank you goes out to my friends and family. My mother and father, Therese & Abdo, my two sisters, Patricia & Patina, thank you for your support throughout my whole life. Last but not least, thank you Mr. Samir Abou Samra and Mr. Claude Comair for your non-stop push and support.

2 Key Words

Neural Network, Taylor Series, Approximation

3 Abstract

We focus on the problem of constructing an equivalent function for a given neural network, when the training/test data are not available. The unknown function captured by the neural net is represented as the Taylor series expansion in terms of the inputs, and the relevant coefficients are computed from the weights of the network. We argue that such deconstruction of a neural network can be a useful tool in complexity reduction.

4 Introduction

Neural networks (a.k.a. Artificial Neural Networks, or ANNs) are one of the best-known function approximators with wide ranging applications in Artificial Intelligence and Machine Learning. While traditionally ANNs are learned from supervised data, with pre-specified topology, there has also been a significant body of work on evolving the topology besides learning the parameters of the network. This sheds light on one of the main limitations of ANNs, viz.; the proper topology is often unknown (unless significant domain knowledge is assumed/ available). An improper topology can significantly affect the accuracy of the learned function and its capability to generalize (over-fitting), so it is often worthwhile to learn the topology as well. A complementary approach to this problem has been to learn the weights (parameters) for a complex network and then simplify it iteratively while preserving the functionality. Such *constructive* and *destructive approaches* have met with mixed success, primarily due to the added burden of computation that they impose on time-critical applications, such as a video game.

In this paper, we address the problem of simplifying an ANN, but in a radically different way compared to existing literature. We attempt to infer the underlying function captured by an ANN *directly in terms of the weights of the network.* Our main motivation is that a complex network with possibly 100's of hidden units might have actually captured a very simple function that can be expressed simply in terms of 10's of parameters. We intend to acquire this simple representation of the underlying function making the bulky ANN dispensable.

Another possibility is that the number of inputs is large even if the number of hidden units is small, so that the network still has a very large number of weights. In classification tasks with such networks, if the number of examples (training/test) is relatively small (as it is in most real-world applications), the input space is *sparse,* which means the data may be linearly separable. This indicates that the underlying function may be rather simple and that the bulky network with a plethora of weights is rather wasteful.

A second possible application of our approach could be the following: an ANN usually does not yield an idea of the analytic nature of the function it represents. Suppose the training data are unavailable (this is a likely scenario since the whole point of inductive learning is to extract a model from the training data so that the latter is no longer necessary to maintain) or corrupted. How does one recover the analytic function with access to the ANN only? Our method retrieves an analytic approximation that can be made arbitrarily close to the actual function with sufficient computation time, requiring no access to the training data. The two main assumptions behind our work are

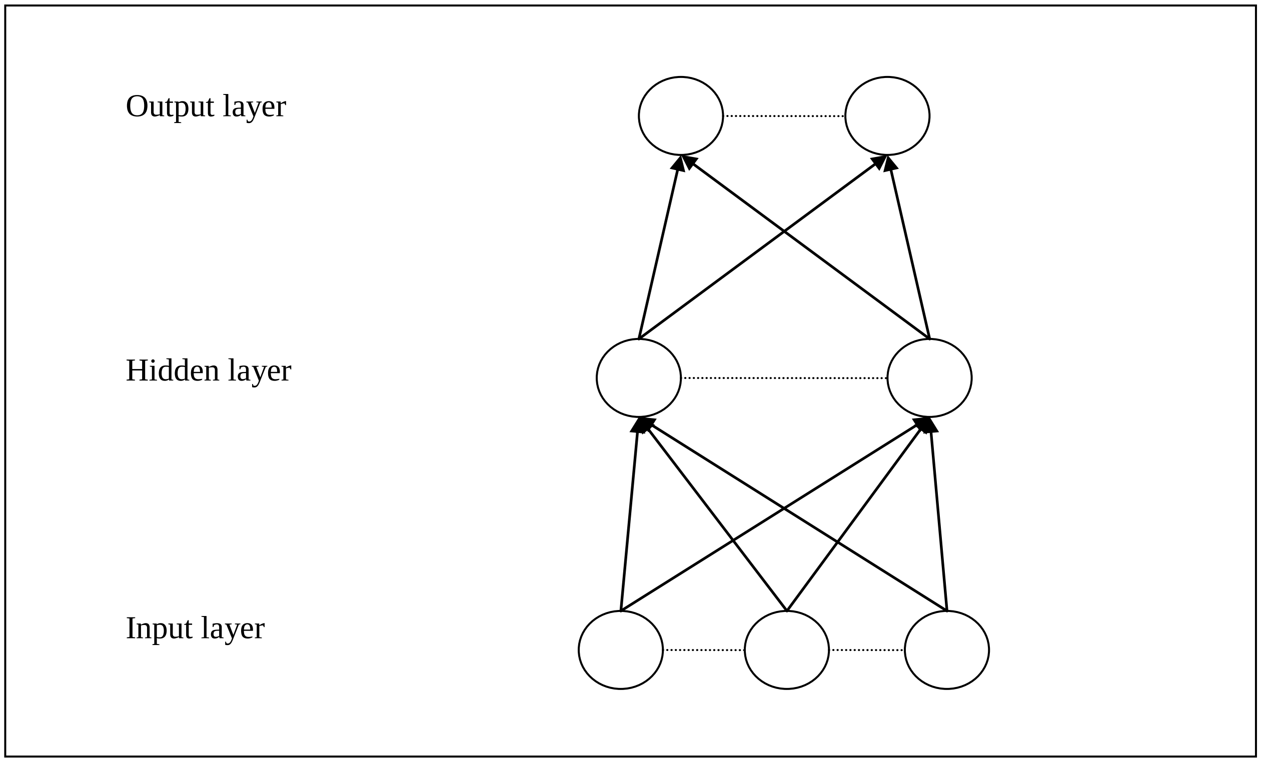
1. The ANN has been learned and the training data are not available any more. We only have access to the weights and the topology of the ANN and can only use this information to infer the underlying function.
2. The underlying function can be expressed as an infinite series using Taylor's series expansion of a function.

We derive the coefficients in the Taylor series expansion of the underlying function in terms of the weights of the given network, and show that the error of approximating this function with a finite number of terms of the infinite series decreases very fast. This means a finite truncation can often produce a very close approximation. We show experiments that the inferred model can exhibit very similar behavior to the ANN that it approximates.

5 What is an Artificial Neural Network?

[8, 13, 14, 15] We will start by giving a brief definition of ANNs. ANNs try to map the way humans think and react to things in life. ANNs start with having inputs or receptors then will go through various calculations to come up with the output or the decision to be made.

ANNs are made from several nodes depending on the problem to be solved. Nodes are connected to each other through connections with different values referred to as weights. The basic structure of a network is having inputs connected to output directly through connections. In several cases, such topology might give you a reasonable solution to your problem. In other cases, we would have to start adding intermediate nodes known as hidden nodes. So, till now, we know that our network is formed from three layers (input, hidden and output) with each layer containing several nodes or neurons. It is worth saying that some problems require having multiple hidden layers.



The figure above illustrates a basic network with one input layer, one hidden layer and one output layer.

5.1 Building an Artificial Neural Network

The process of building an ANN can be fairly complex and exists in several ways. We will focus on one of the most commonly used ways to build an ANN. We will explain briefly how a Neural Network is trained and used.

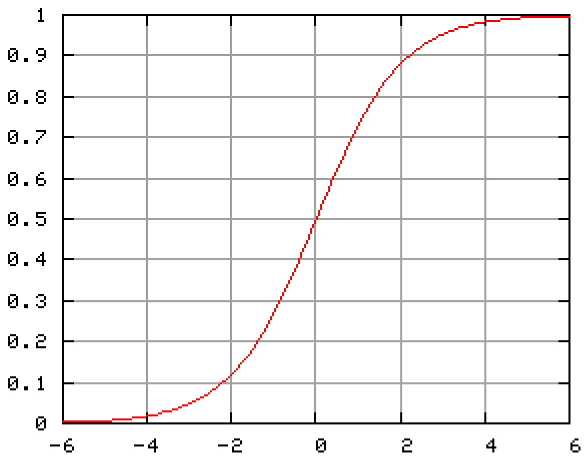
First, we need to start with a set of data called the training set. This set will hold some certain values that we would like our network to learn approximating or imitating. Basically, it will consist with a list of certain inputs and their respective (desired) outputs. The more data we have, the more general network we can build. But, if the range of the data is too wide, we may never reach a reasonable solution.

Second, we initialize all our connection weights to random values. Now, the training will begin to adapt this network to the data. There are several ways to train ANNs; the most commonly training method used the back-propagation method. This method consists of two things: Feed Forwarding and Back Propagating. This training method will start by entering the inputs one by one and feeding them to the network. Then, the output will be computed referred to by current output. We compare our current output with the desired output of the respective inputs. We calculate the error term and then go backward in our network adjusting the connection weights. After that, we move on to the next input in the data set. We pass on the data sets for a certain number of times usually predefined or we can keep on iterating until the total error reaches a certain limit.

The most common way to calculate network outputs is using the sigmoid function. This function is called the activation function. The activation function, σ, and its first derivative are given by

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$



There are several activation functions to be used but sigmoid has proved to be one of the most efficient ones to be used because of its ability to capture and approximate many functions. After the training is complete and the error has become small, we migrate this network with the resulting weights into our program and just use it with any inputs we want.

6 Previous Work

Finding the ultimate and perfect neural network for certain problems has always been a problem for A.I. programmers. Programmers are always wondering if this topology fits the best or maybe the other. A certain number of hidden nodes was chosen by experts and then training techniques were used to fix the Neural Network. However, even with all the experts spending so much time and working to find the best topology for the given Neural Network problem, programmers, in most cases, were not satisfied with the results reached.

Many have tried to come up with the ultimate formula to calculate the number of hidden nodes to be used in the network. All the formulas reached gave us good results, but we always needed great results. Many methods were introduced to create and build Neural Networks. All can be categorized into two main groups: Constructive methods and Destructive methods. Reading these two words, we can build a small idea on how these two methods work. Later in the paper, we will talk about these two methods.

6.1 Destructive Methods

Destructive methods are methods that use the destruction approach to find the ultimate Neural Network desired. What that means is that we take a Neural Network with a certain number of hidden nodes and work on reducing that number to a smaller number. We will talk about two types of Destructive Methods. The first one is related to pruning weights from the network while the other uses node removals in destructing the network.

1- *Weight Pruning*

   This section deals with minimizing network topologies by removing weights from the network.

   a- *Magnitude based pruning algorithms [1]*

      This method deals with removing weights starting with the weights with the smallest value. This method considers that weights with smaller values have the least effect on the network, so removing them must have the same effect on the whole network in general.

b- Optimal Brain Damage (OBD) [1]

This algorithm deals with calculating a certain value named “saliency” of weight *i*, and then the weight with the least saliency value will be removed from the network. The value saliency is calculated according to the following formula:

$$s_i = \frac{W_i^2}{2} \sum_{\mu} \frac{\partial^2 E^{\mu}}{\partial (O^{\mu})^2} \left(\frac{\partial O^{\mu}}{\partial W_i} \right)^2$$

where

$$W_i \ is \ the \ given \ weight$$

$$E^{\mu} = \frac{1}{2} (y^{\mu} - O^{\mu})^2$$

$$y^{\mu} \ is \ the \ desired \ output$$

$$O^{\mu} \ is \ the \ reached \ output$$

$$\mu \ goes \ from \ 0 \ to \ total \ number \ of \ output$$

After calculating the saliency for all the weights in the network, we pick the weight with the smallest saliency number and remove it from the network.

2- Node Pruning [1]

This type of pruning deals with removing nodes. Basically, we need to find out how the network works or performs without a certain node. In other words, we need to calculate the error difference in the network in the presence and absence of that certain node.

6.2 Constructive Methods

Constructive methods have been the main center of attention in evolving networks. Many ideas have been discovered in this field. Some used Genetic Algorithms to evolve the networks, while others found other ways to do so. In the following sections, we will look at some of the techniques used to evolve neural networks.

1- *TWEANN (Topology & Weight Evolving Artificial Neural Networks) [2]*

   In this section, we will see how the idea of evolving neural networks using genetic algorithms evolved through time. Basically, to be able to evolve the topology of a certain neural network using GAs, we had to find the right genetic representation to be used in the process of evolving.

   a- *Binary Encoding*

      In this type of encoding, we use the simplest traditional encoding using the bit string representation used by GAs in general. An algorithm called sGA (Structured Genetic Algorithm) was used in this encoding type. A bit string was used to represent the connection matrix of the network. The limitation of this algorithm was that the size of the connectivity matrix was the number of nodes squared which means that this matrix would explode for really large networks with a big number of nodes. Crossover would not give us good results since it would be hard to apply beneficial crossovers using a linear string to represent the graph structure.

b- Graph Encoding

This type of encoding tried to solve the problem encountered in the above section. A new algorithm was introduced which used the dual representation. This algorithm is called PDGP (Parallel Distributed Genetic Programming). The first representation in this algorithm was a graph structure while the second representation was a linear genome describing the connections between the nodes. In this type of encoding, sub-graph-swapping crossovers and topological mutations use the graph structure representation while point crossovers and connection parameter mutations use the linear genome representation. The problem encountered here resembled the one found in section (a) which is the finite limit set for the number of nodes in the network before it explodes.

c- GNARL (Generalized Acquisition of Recurrent Links)

This kind of evolution is known also as the non-mating type of evolution. This means that founders of this algorithm gave up on crossover by commenting that "the prospect of evolving connectionist networks with crossover appears limited in general". The only problem here is that the founders of this algorithm state that it's better to work on network evolution without using crossover without showing its advantages or disadvantages. They just leave the problem of demonstrating the advantages of crossover to other methods.

d- *Indirect Encoding*

This type of encoding is famously known as CE (Cellular Encoding). In CE, genomes are programs written in a specialized graph transformation language. Unfortunately, this type is not used widely in evolving neural networks because indirect encoding do not map directly to their phenotypes hence they can bias the search in unpredictable ways.

2- *Cascade Correlation Algorithm [4]*

This type of evolving neural network does not use GA. It is a fairly simple and interesting way in finding the best neural network topology suitable for our problem.

a- *Introduction*

As seen in the previous parts, we encountered drawbacks in the way we used GA in building perfect neural network topologies. So, basically programmers needed another type of algorithm to build networks. The Cascade Correlation Algorithm is an algorithm that approaches the problem of building neural networks from another view using a different approach which does not user Genetic Algorithms. A short description of this algorithm would be that the algorithm starts from minimal structure (no hidden nodes). It then trains the network and adds hidden nodes one by one. Each time it adds a new hidden node, the input side weights are frozen and the system is trained. This process is used until the network reaches a topology with a small margin of error. We will see a more general description of the algorithm later in this paper.

b- *Back-Propagation is Slow?*

One of the main reasons that led into establishing the Cascade Correlation algorithm is finding a problem for the ordinary back-propagation learning algorithms. The problem is that back-propagation algorithms require lots of epochs to reach a solution (An epoch is defined as passing through all the training examples one time). Most of the times, we may reach a solution that does not suite our problem. We may sometimes reach a point where we never reach a solution. Basically, back-propagation is less desired due to two large factors:

i) *Step-Size Problem*

First reason that shows us why the back-propagation learning technique is less desired is the step-size problem. What we mean by this is that back-propagation uses in its calculation the first derivative of the overall error with respect to each weight in the network only. In this way, we might miss the perfect solution while going backwards in the network and fixing the weights.

Many solutions were introduced in order to try and fix this problem. One of the solutions is taking large steps which will help us find the solution directly but still, we are left with missing the right solution problem. Another solution is using a new concept called *momentum*. This concept tracks the weight changes in the network and finds the best solution for the network. A third solution suggested computing the second derivative of the total error with respect to each weight in the network. This solution encountered a major problem. The problem was that the curvature of the error function is not given in the network. One solution to the last problem was calculating explicitly the approximation of the second derivative and using it in the gradient descent. Finally, a solution came up that proved to be the most useful which became known as Fahlman's Quickprop algorithm. Quickprop computes the first derivative $\frac{\partial E}{\partial w}$ as the usual back-propagation algorithm, but here instead of a simple gradient descent, Quickprop uses a second-order method, related to Newton's method, to update the weights.

ii) The Moving Target Problem

The second problem in the back-propagation learning technique is the Moving Target problem. This problem addresses the issue of the always changing weights in a manner that may never lead to a solution. This is due to the fact that in ordinary back-propagation techniques, the hidden nodes do not communicate with each other in the network. In this manner, each weight connecting the hidden nodes to the output and input nodes is updated after each and every element of the training set. In this way, the weights will keep on changing and never settling down for a long time to see if one of the weights changed would be fit for the network or can actually be a solution for the network.

One of the main problems found in the Moving Target problem is what is known by *the herd effect*. This effect occurs in the case where we have for example multiple hidden units and they have two training examples to train against. Let's name those training examples A & B. Suppose that A always receives a greater margin error than B. All the hidden units will work more on fixing A's problem leaving B on this side. Once A is solved, the units will see B's problem and work on fixing it, and before they know, A would have begun having problems again. A solution to this problem would be allowing some of the weights in the network to change at once and other weights to remain constant. This concept was also known as freezing the weights.

c- *The Algorithm*

The Cascade Correlation algorithm relies on two basic ideas. The first idea is called the cascade architecture, in which hidden units are added to the network one by one and these added nodes do not change after being added to the network. The second idea is the learning algorithm which will be used to create and add the hidden nodes needed.

The Cascade Correlation algorithm starts minimally with no hidden nodes. It starts with its input nodes and output nodes. A bias is added as to the input nodes, and it's always set to +1(*Figure 1*).

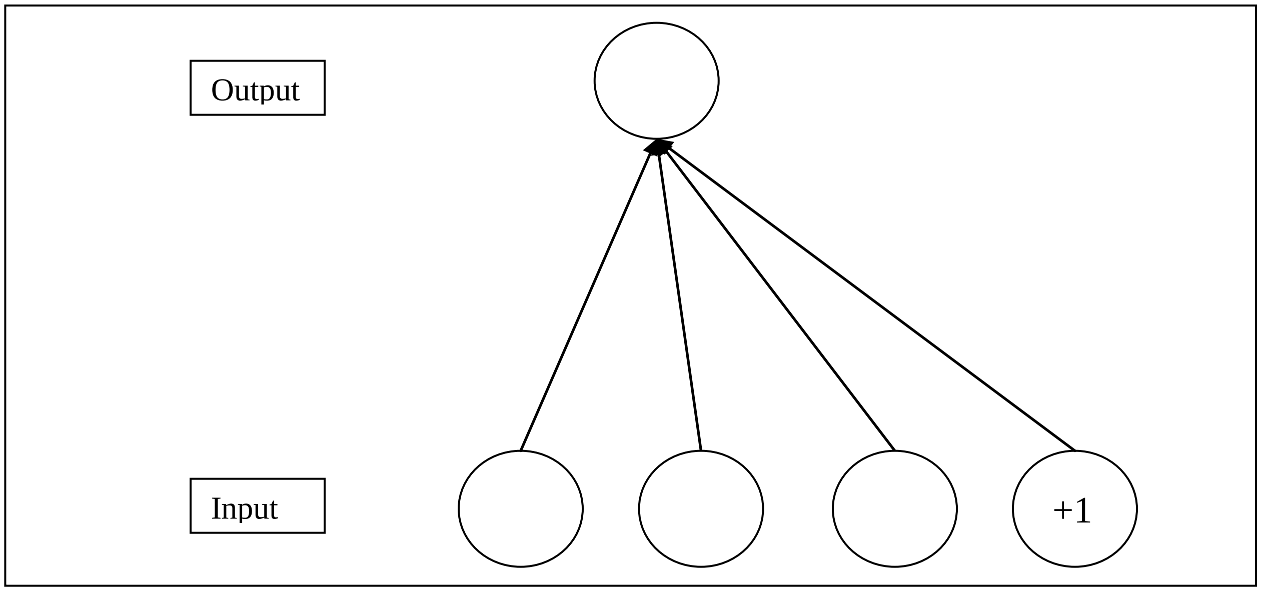


Figure 1

As seen in the above figure, all inputs are connected to the output. The special feature used in this algorithm is that everything is connected to everything.

The output in this algorithm can be calculated linearly by multiplying each input with the weight of the connection that connects it with the output and then adding the result of all those multiplications, or a non-linear activation function can be applied. This algorithm has been tested positively using a symmetric sigmoidal function whose output range lies between -1 and +1.

The algorithm adds hidden nodes one by one. Each time a new hidden node is added, it is connected to all other nodes in the system. What is meant by that is that it will be connected to all input and output nodes. In addition, it will be connected to all the previously added hidden nodes in the system. When this new node is added, all connections going into all hidden nodes are frozen and all the output connections are trained repeatedly. This is the basic idea of the algorithm.

Now comes the question of when to add a new node?

When the system starts minimally with no hidden nodes, we train the weights using any technique desirable. It could be the “delta rule”, the “Perceptron learning algorithm” or any of the well-known learning techniques for single-layered networks. We can also use the Quickprop algorithm described earlier. When used in single-layered networks, Quickprop acts like the “delta rule” algorithm. Regardless of what algorithm we chose to use, we keep on applying that algorithm until our network reaches a point where the error difference is less than a certain threshold that we've set. If that condition is met, we run the algorithm one last time to measure the error of the system. Now we test for this error. If it's less than a certain threshold that we, again, have set, we stop. If not, we add a hidden node to the network. The question how to add a node will be answered in the next paragraph. After adding the new node, as stated before, we connect it to all other nodes including previous hidden nodes (if found). Now we freeze all the connections going from the input nodes to all hidden nodes, and we train the system.

Now, we move on to the process of how to add a new hidden node. We begin with a unit that will be called *a candidate unit*. This unit or node is connected to all input nodes and pre-existing hidden nodes with connections of random weights. At first, we will not connect it to the output nodes. It will be treated now as an output node, and we will apply the used learning algorithm on this little system that we have temporarily created. While fixing the weights of the new connections, we always try to maximize *S* where is *S* is defined by the following formula.

$$S = \sum_{o} \left| \sum_{p} (V_p - \bar{V})(E_{p,o} - \bar{E}_o) \right|$$

where

$$o\ goes\ from\ 0\ to\ the\ number\ of\ outputs - 1$$

$$p\ goes\ from\ 0\ to\ the\ number\ of\ training\ passes$$

$$V_p\ is\ the\ value\ reached\ in\ this\ pass\ at\ the\ candidate\ node$$

$$\bar{V}\ is\ the\ average\ of\ all\ values\ reached$$

$$E_{p,o}\ is\ the\ error\ reached\ from\ this\ pass\ at\ this\ candidate\ node$$

$$\bar{E}_o\ is\ the\ average\ error\ of\ this\ output$$

In order to maximize *S*, we compute $\frac{\partial S}{\partial w_i}$ the derivative of *S* with respect to all the current node's input weights w_i, which can be computed by the following formula.

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o\left(E_{p,o} - \bar{E}_o\right) f'_p I_{i,p}$$

where

$$i\ goes\ from\ 0\ to\ the\ number\ of\ connected\ nodes$$

$$E_{p,o}\ is\ the\ error\ reached\ from\ this\ pass\ at\ this\ candidate\ node$$

$$\bar{E}_o\ is\ the\ average\ error\ of\ this\ candidate\ node$$

$$f'_p\ is\ the\ derivative\ of\ the\ activation\ function$$

$$I_{i,p}\ is\ the\ input\ of\ connection\ i\ at\ patter\ p$$

After computing $\frac{\partial S}{\partial w_i}$ for each input connection at the candidate node, we perform a gradient ascent to maximize *S*. When *S* stops improving, we add the new candidate, freeze the connections that come into the new node, connect it to the output nodes and continue applying the algorithms to train the network and check for the error.

An addition to the algorithm would be using a set of candidate nodes, known as the *pool of candidate units*. Each of these units will have a set of random initial weights. In order to choose the right unit, we perform the steps explained above about maximizing *S*. At the end, we install the candidate with the best correlation score. In this way, we will have multiple candidates to fit into our system, and we get to choose the best of them.

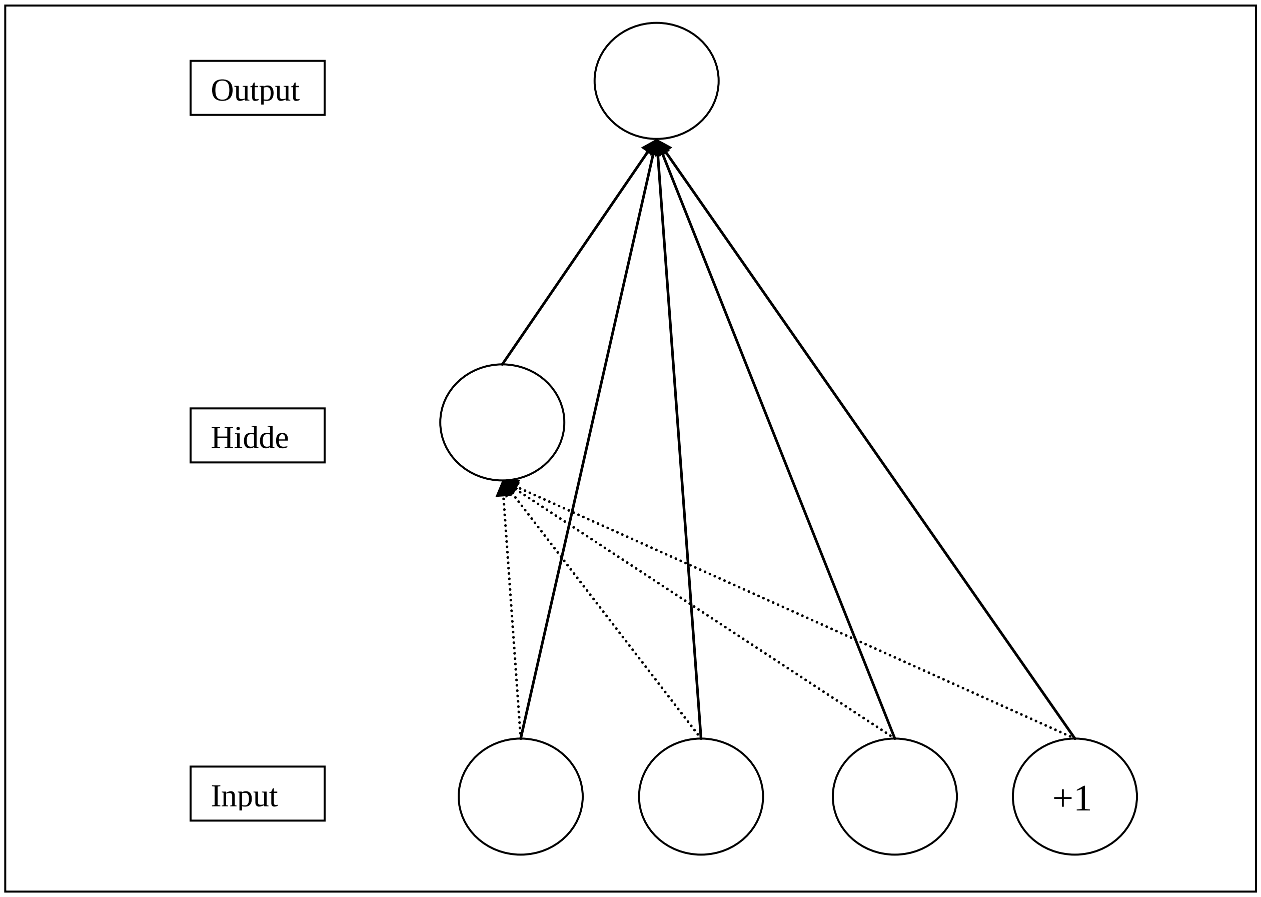


Figure 2

In Figure 2, we can notice how we added a new hidden node to the system.

The connections going from the inputs to the newly added node are dotted meaning that they are frozen.

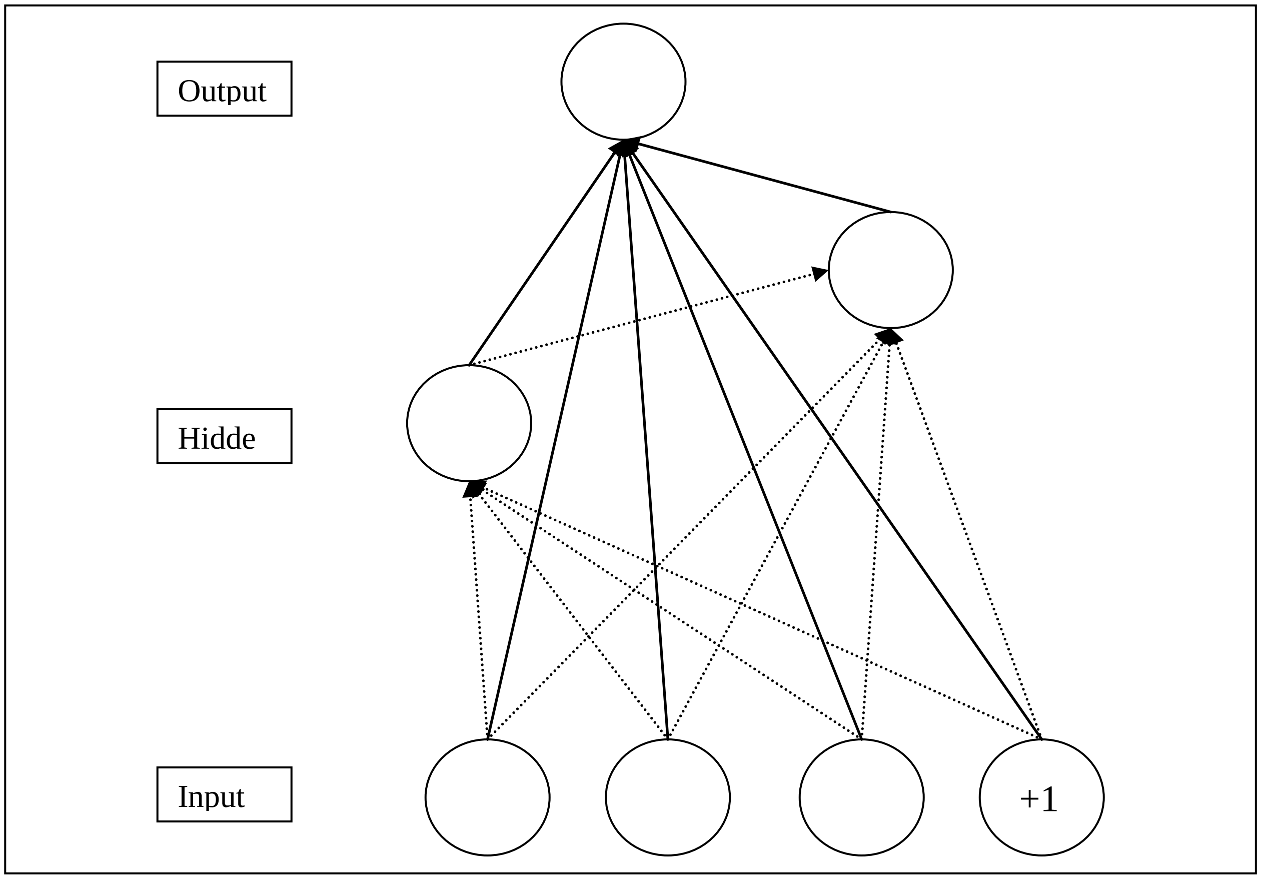


Figure 3

In Figure 3, another, hidden node in the system. Notice how again all connections going to the new node are frozen, even those coming from other hidden nodes.

d- Advantages

- Quick learning
- Network determines size and topology needed for itself
- No back-propagation most importantly

- Retains structure built even if training sets have changed

3- *NEAT (Neuro-Evolution with Augmenting Topologies) [2]*

This type of evolution used Genetic Algorithms to evolve the neural network from minimal topology using a new and improved way. This algorithm came as an answer to previous problems encountered while dealing with the evolution of neural networks using genetic algorithms. The basic three problems that NEAT works on fixing are competing conventions, protecting innovation with speciation and initial population and topological innovation.

a- *Competing Conventions*

There can be lots of ways to represent a graph. This is the competing conventions problem also known as the permutations problem. Basically, it means that we can represent our network in multiple ways thus leading to problems when applying crossover between networks that can be the same but because of their different representation appear to be different.

In Figure 4, we see an illustrated form of the problem faced. In this figure, both networks are the same but they are represented in a different way. Now, if we apply a crossover between those two networks, we will see that we will lose data from the networks and this data will be a hidden node. As seen in the same figure, the results of the crossover gave us two offspring. In each of these two offspring, there is a node missing. The correct answer must have the same topology as the one of the parents' since both have the same topology. We need a way to keep track of the network's structure to know how to apply crossover between networks.

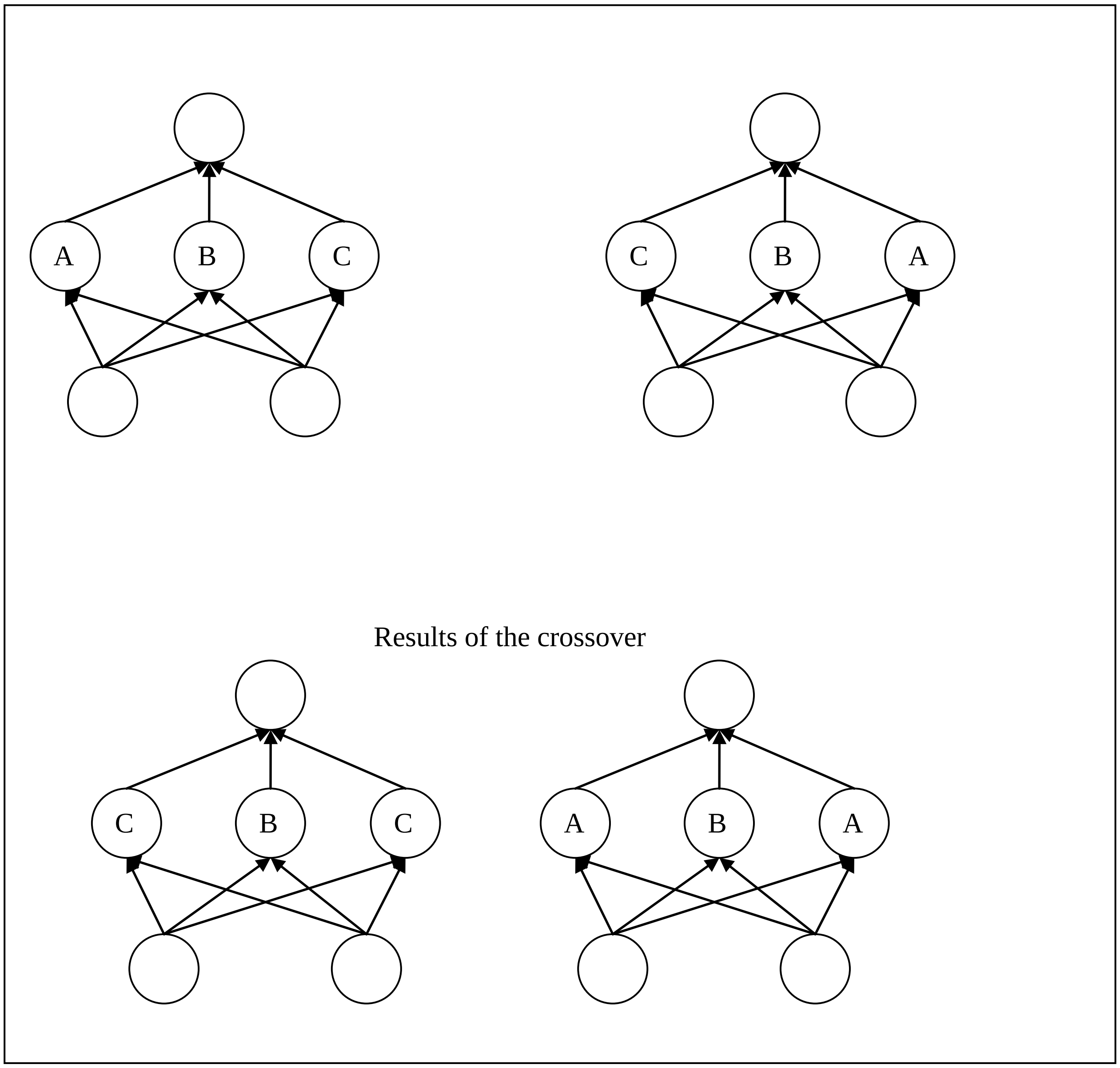


Figure 4

b- *Protecting Innovation with Speciation*

The second problem occurs when adding new structures to the population. The new structures' fitness will be low at first hence they are in danger of being removed from the population in the next generation. So basically, what we need is a way to protect these new networks and give them a chance to improve before we put them under the microscope and test for their fitness.

c- *Initial Population and Topological Innovation*

The last problem relies trying to know how to start. We've seen several ideas on how to begin. What we mean here by how to begin is deciding on the initial topology (structure) of the network. Most of the old algorithms that used the concept of evolution have started with a random number of hidden nodes. NEAT argues that the reason that the reason why most of those algorithms have failed to deliver good results was according to that factor.

Now we reach the part where we begin to explain the NEAT algorithm in detail after having explained the problems that have lead to the rise of NEAT. In the next part, we will address the structure used in NEAT, plus we will talk about how NEAT solved each of the arising problems talked about in the first section.

I. *Genetic Encoding*

NEAT came as an answer to the problems faced while using the concept of evolving neural networks using GAs. One of the problems faced was the genetic representation used in previous algorithms. NEAT uses a simple way of representing genes and genomes, and this way has proved to be very effective in a positive way to the evolution process. NEAT's gene representation was designed in a way to make it easier to apply crossover between two genomes. The NEAT uses two types of genes. These genes make up the genomes. A genome in NEAT is a network structure itself. The two types of genes are connection genes and node genes.

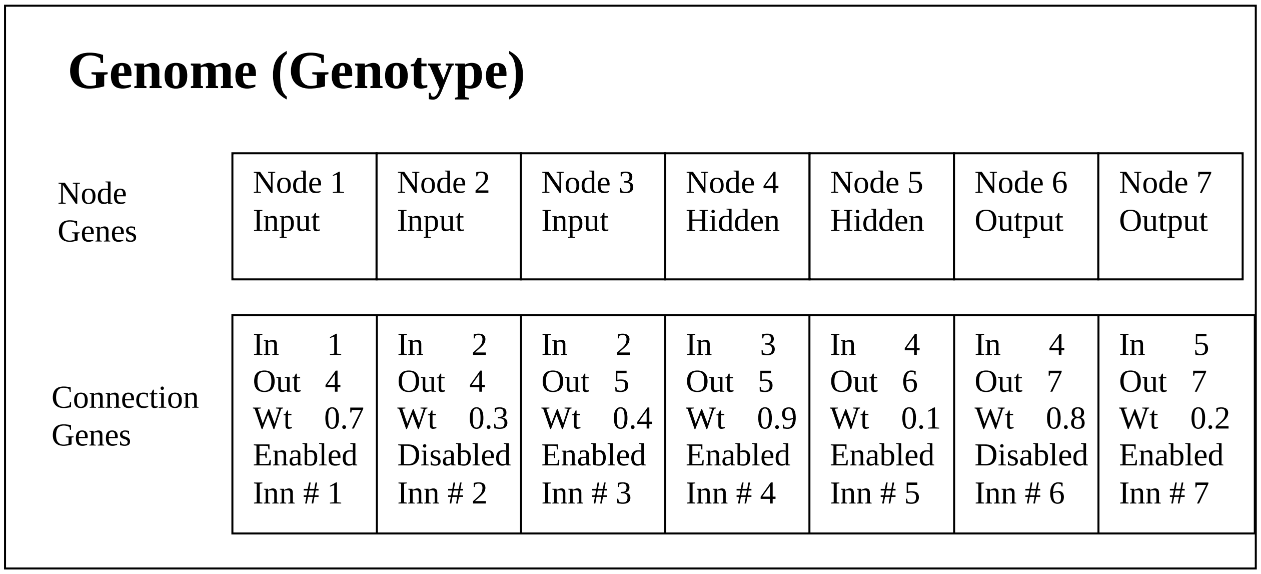
i) *Node Gene*

This gene represents the nodes in the network. Its structure contains mainly two things: its name or ID in the network, and its type (input, hidden or output).

ii) *Connection Gene*

The connection represents the connections that connect two nodes in a network. This gene's structure contains more things than the node gene. It first contains two node pointers: one for in-node, and one for the out-node. Second, it contains the weight of this connection usually a floating point number. It also contains a Boolean that expresses whether this connection is enabled or disabled. Finally, it has a number called the Innovation Number. The Innovation Number acts like an ID for this connection. This is used mostly in crossover function (explained later).

After seeing how genes are represented, let’s see how genomes are formed in the network. Then we’ll see how phenotypes can be acquired from genomes (genotypes). Figure 5 illustrates the general representation of a genome. As you can see, it is made of two lists: node gene list and connection gene list. The node list contains all nodes present in the network. Each node has its ID and it states whether it’s an input, a hidden or an output node. The connection gene list contains all the connection found in the network. Each connection gene has two pointers telling us what nodes this connection connects. Then, we can see the weight, and the Boolean to tell us whether this connection is turned on or off. Finally, we see the innovation number which acts as the ID for this connection. Please note that connections that have same in and out node have the same innovation number even if this connection was added later in a future generation.



Genome (Genotype)

Node Genes	Node 1 Input	Node 2 Input	Node 3 Input	Node 4 Hidden	Node 5 Hidden	Node 6 Output	Node 7 Output
Connection Genes	In 1 Out 4 Wt 0.7 Enabled Inn # 1	In 2 Out 4 Wt 0.3 Disabled Inn # 2	In 2 Out 5 Wt 0.4 Enabled Inn # 3	In 3 Out 5 Wt 0.9 Enabled Inn # 4	In 4 Out 6 Wt 0.1 Enabled Inn # 5	In 4 Out 7 Wt 0.8 Disabled Inn # 6	In 5 Out 7 Wt 0.2 Enabled Inn # 7

Figure 5

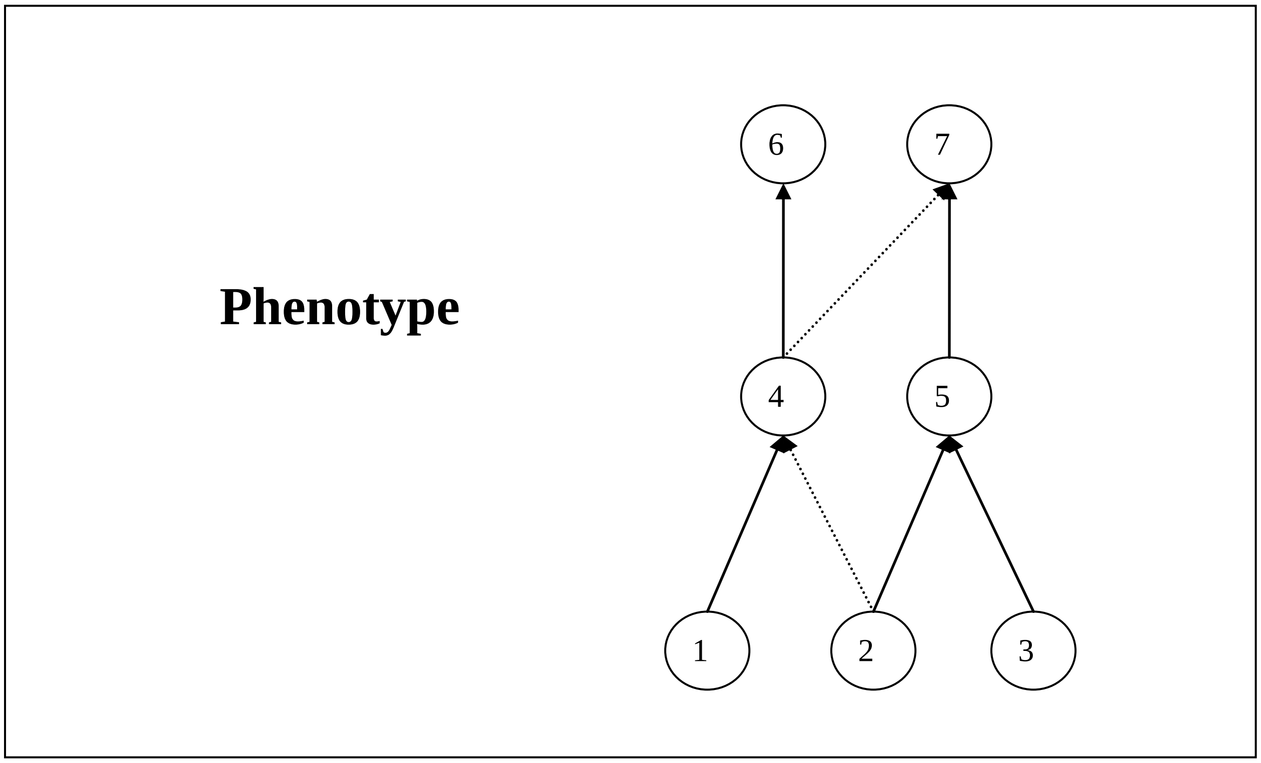


Figure 6

Figure 6 shows us what the phenotype of the above genome will look like. The dotted connections mean that this connection is disabled.

Now, we will talk about mutations and how they occur in NEAT. NEAT has two types of mutations that happen during the evolution process. Mutation can either add a new connection or add a new node to the network. Both mutations will help the network get larger. This shows us how the evolution part of the network occurs.

i) *Add Connection*

This type of mutation adds a new connection to the network. A new connection means that a new innovation number has to be assigned to this new connection. Where to add this new connection can be at a random place and the weight of this connection is also random. Note that this new connection will now connect two nodes that were not connected previously in the network. This process is illustrated in Figure 7. The added connection has the color red. The new connection got the Innovation Number 8.

ii) *Add Node*

This type of mutation adds a new node to the network. For sure, the type of this node is a hidden node (input and output node number is fixed throughout the process). In here, an existing connection is split and the node is added in between the two nodes that were connected by the chosen connection to be split. The old connection (the one split) will be set to disabled. Two new connections will be added to the network. The new connection going into the new node added will receive a weight of +1, while the connection going from the new node to the out-node of the old connection will have the same weight of the old connection. Figure 8 shows us the add node mutation process. The new node added has the ID 8. Notice how connection of Innovation Number 7 is disabled and two new connections are added to the system.

Connection Genes before Mutation

In 1	In 2	In 2	In 3	In 4	In 4	In 5
Out 4	Out 4	Out 5	Out 5	Out 6	Out 7	Out 7
Wt 0.7	Wt 0.3	Wt 0.4	Wt 0.9	Wt 0.1	Wt 0.8	Wt 0.2
Enabled	Disabled	Enabled	Enabled	Enabled	Disabled	Enabled
Inn # 1	Inn # 2	Inn # 3	Inn # 4	Inn # 5	Inn # 6	Inn # 7

New Connection Gene added after Mutation

In 1
Out 5
Wt 0.7
Enabled
Inn # 8

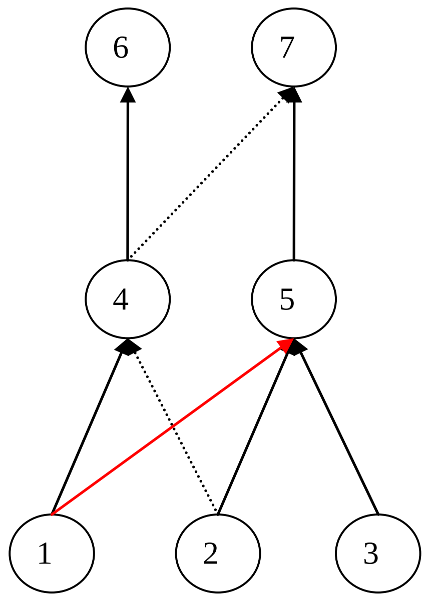


Figure 7

Node Genes before Mutation

Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7
Input	Input	Input	Hidden	Hidden	Output	Output

Connection Genes before Mutation

In 1	In 2	In 2	In 3	In 4	In 4	In 5
Out 4	Out 4	Out 5	Out 5	Out 6	Out 7	Out 7
Wt 0.7	Wt 0.3	Wt 0.4	Wt 0.9	Wt 0.1	Wt 0.8	Wt 0.2
Enabled	Disabled	Enabled	Enabled	Enabled	Disabled	Enabled
Inn # 1	Inn # 2	Inn # 3	Inn # 4	Inn # 5	Inn # 6	Inn # 7

Node Gene added after Mutation

Node 8
Hidden

Old Connection Gene disabled after Mutation

In 5
Out 7
Wt 0.2
Disabled
Inn # 7

New Connection Genes added after Mutation

In 5	In 8
Out 8	Out 7
Wt 1.0	Wt 0.2
Enabled	Enabled
Inn # 8	Inn # 9

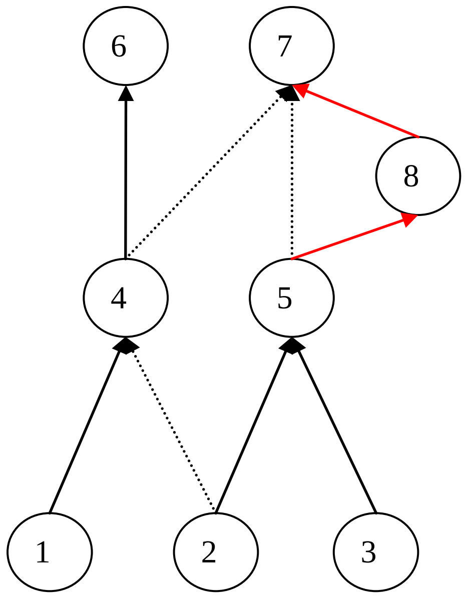


Figure 8

This process of mutation will gradually help the network grow up into larger networks and converge towards the perfect network structure solution.

II. *Tracking Genes through Historical Markings*

This section talks about the way to solve that Competing Conventions problem. The solution was to add a small number of which we now know as the Innovation Number. This number will mostly help us in the crossover process. As illustrated before, the competing conventions problem occurred when we were applying crossover between two nodes of the same structure. Now, using the Innovation Number, the crossover is much easier and much more effective. All we have to do is match up genes with the same Innovation Number and leave them as they are in the new network, while the non matching genes will be added two as is to the network. Now the question is: how to choose what connection genes to take between matching genes. Figure 9 shows us the first of two parents that we’re going to apply crossover to. Figure 10 shows us the second parent.

Parent 1
Connection
Genes

In 1	In 2	In 3	In 4	In 3
Out 4	Out 4	Out 5	Out 5	Out 4
Enabled	Disabled	Enabled	Enabled	Enabled
Inn # 1	Inn # 2	Inn # 3	Inn # 4	Inn # 6

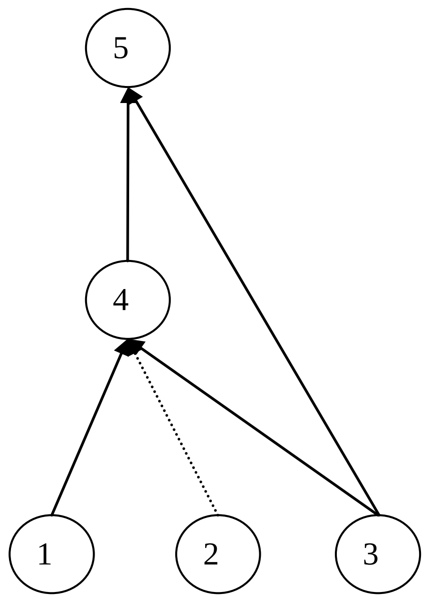


Figure 9

Parent 2 Connection Genes	In 1 Out 4 Enabled Inn # 1	In 2 Out 4 Disabled Inn # 2	In 3 Out 5 Disabled Inn # 3	In 4 Out 5 Enabled Inn # 4	In 2 Out 6 Enabled Inn # 5	In 6 Out 5 Enabled Inn # 7

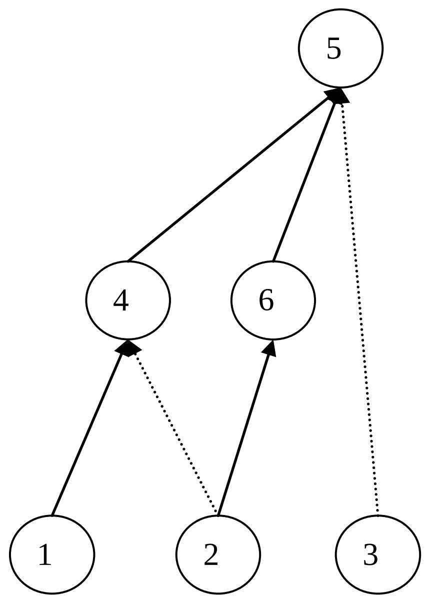
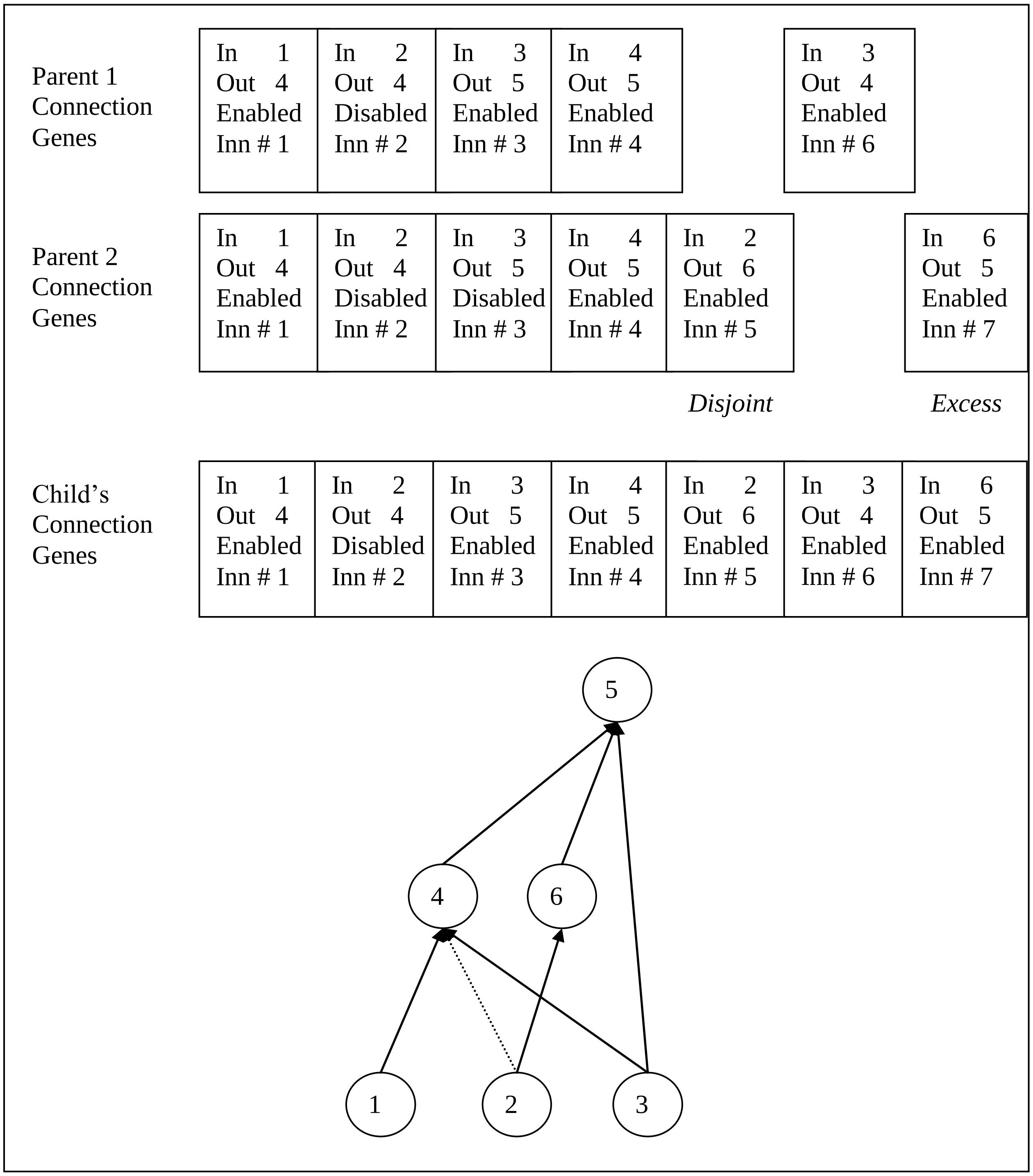


Figure 10

Figure 11 shows us the result of the crossover when it happens between the two parents shown above. First, we lay out all the genes and sort them by their Innovation Number. All genes in Parent 2 whose Innovation Number is less or equal than the maximum Innovation Number in Parent 1 and that are not found in Parent 1 are called *Disjoint* genes. All genes in Parent 2 whose Innovation Number is larger than the maximum Innovation Number in Parent 1 are called *Excess* genes. Matching genes are chosen randomly from the parents to be in the child. Non matching genes are taken from the fittest parent and put in the child. In case, both parents have the same fitness, non matching genes are chosen again randomly.



Parent 1 Connection Genes	In 1 Out 4 Enabled Inn # 1	In 2 Out 4 Disabled Inn # 2	In 3 Out 5 Enabled Inn # 3	In 4 Out 5 Enabled Inn # 4		In 3 Out 4 Enabled Inn # 6	
Parent 2 Connection Genes	In 1 Out 4 Enabled Inn # 1	In 2 Out 4 Disabled Inn # 2	In 3 Out 5 Disabled Inn # 3	In 4 Out 5 Enabled Inn # 4	In 2 Out 6 Enabled Inn # 5		In 6 Out 5 Enabled Inn # 7
					Disjoint		*Excess*
Child's Connection Genes	In 1 Out 4 Enabled Inn # 1	In 2 Out 4 Disabled Inn # 2	In 3 Out 5 Enabled Inn # 3	In 4 Out 5 Enabled Inn # 4	In 2 Out 6 Enabled Inn # 5	In 3 Out 4 Enabled Inn # 6	In 6 Out 5 Enabled Inn # 7

Figure 11

III. *Protecting Innovation through Speciation*

One of the previous problems that we talked about is that when a new child is introduced into the population, its fitness will be very low hence making it a candidate to be deleted in the next generation. We do not want this effect; instead we want this new child to grow up and then compete. In other words, we want to find a way to protect this child while it grows up. This is where the concept of Speciation comes into play. We will divide the population into smaller species. Networks will be grouped into species according to their genotype, meaning that networks of same structure will be in the same group. In this way, we will networks compete in their own niche and grow up in there, and then come back and compete in the population at large.

Networks of different topology can be added to species too by using a compatibility distance function to check how compatible these two networks can be. Below is the compatibility function:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 . \bar{W}$$

where

$$c_1, c_2 and\ c_3\ are\ three\ coefficients\ to\ determine\ the\ importance\ factor$$

$$E\ is\ the\ number\ Excess\ nodes$$

$$D\ is\ the\ number\ Disjoint\ nodes$$

$$\bar{W}\ is\ the\ average\ weight\ difference\ between\ matching\ genes$$

$$N\ is\ the\ number\ genes\ in\ the\ larger\ genome$$

We will specify a compatibility threshold δ_t to check for compatibility between two genomes. If the current genome is compatible with any of the present species, it will be added to that species; else we create a new species and add the current genome to it.

In NEAT, an explicit fitness sharing is used. In this manner, a species cannot become too large and take over the whole population because the solution will converge to whatever species performs best at first. What we do is that we adjust the fitness of the genomes in the species to represent the whole genomes in that species using the function below:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh\big(\delta(i,j)\big)}$$

where

$$i\ goes\ from\ 1\ to\ number\ of\ genomes\ in\ the\ species$$

$$f'_i\ is\ the\ adjusted\ or\ new\ fitness\ of\ the\ genome\ i$$

$$f_i\ is\ the\ fitness\ of\ the\ genome\ i$$

$$n\ is\ the\ number\ genomes\ in\ the\ species$$

$$\delta(i,j)\ is\ the\ compatibility\ distance\ between\ genome\ i\ and\ j$$

$$sh\ is\ the\ sharing\ function$$

$$sh = \begin{cases} 0\ if\ \delta(i,j) > \delta_t \\ 1\ if\ \delta(i,j) \le \delta_t \end{cases}$$

Now species reproduce by first removing the genomes with lowest fitness in the population. Then the population is replaced by the offsprings of the remaining genomes in the population. After that, we perform a technique, also known as clustering, where we divide the offspring into species according to their topologies.

IV. *Minimizing Dimensionality through Incremental Growth from Minimal Structure*

The last problem was finding out how to start with the algorithm. What we mean by that is choosing the initial topology that we start applying the algorithm to. We know that the number of input nodes and output nodes is fixed and cannot change, so our problem is finding the right number of hidden nodes to begin with. Most of the old algorithms have started with a random number of hidden nodes to begin with. Starting randomly might give us a network structure that already passed the perfect topology required. For example, let's say that the random generator gave us 3 hidden nodes. Starting from that number of nodes, we perform our evolution algorithm and the solution returned was 3 hidden nodes. The question asked here is what if we started with 0 number of hidden nodes and the perfect solution was 2 hidden nodes. NEAT states that it's best to start with no hidden nodes and build up. This is the concept of starting minimally to reach the minimum solution size desired.

7 Deconstructing a Neural Network

Neural network structures can grow to be very large and complex, with many connections (and consequently many weights to store). Large networks require large memory space for storage. We have seen several ways to prune networks that try to minimize this complexity without sacrificing accuracy significantly. These methods can be quite complex and often leave us with large networks.

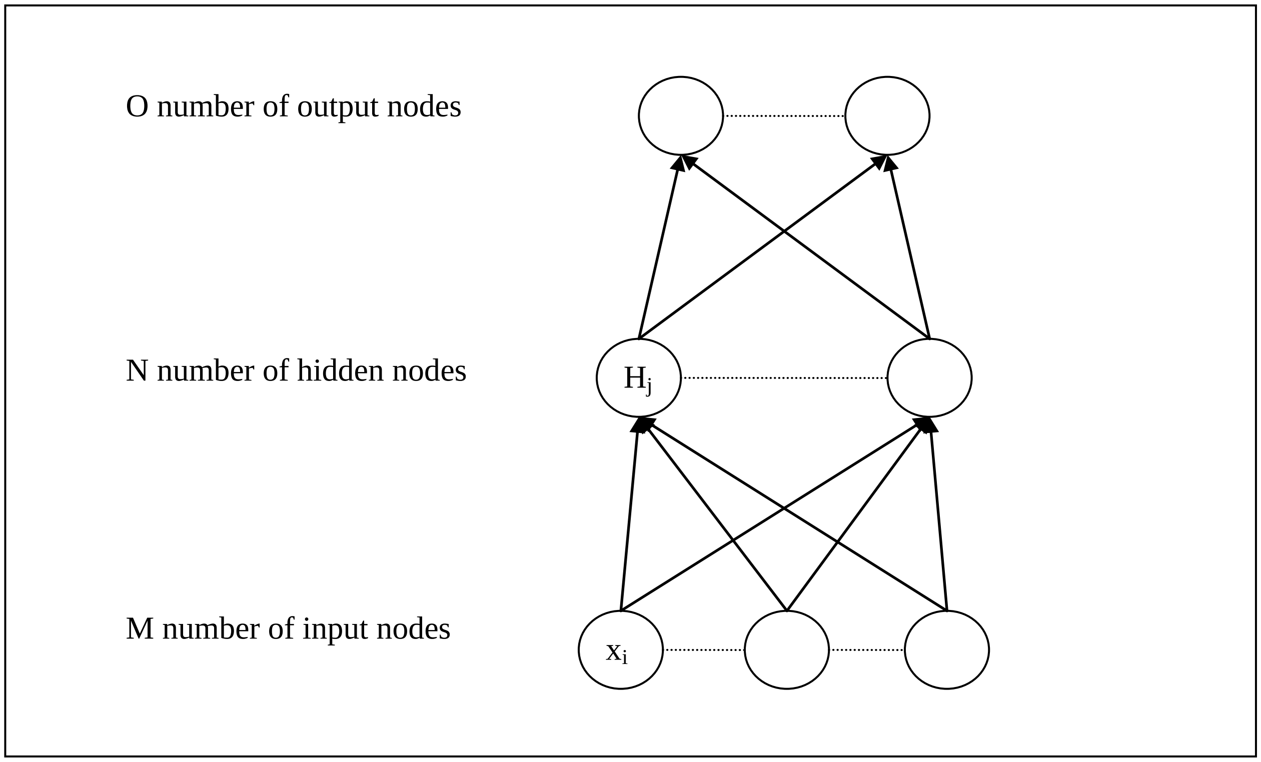
We will propose a different approach to this process of simplification. We extract an analytic function of minimal complexity that has the same behavior as the network, to a certain error-tolerance level. We show that the resulting expression can involve significantly fewer parameters than the given network, leading to a compact equivalent function.

When viewed as a network, our equivalent function can be very different from the original network. It uses only the polynomial terms (in inputs) that are indispensable in capturing the functionality of the original network. Since it is based on the original network's parameters (and not on the training data), any inaccuracies in the latter will be translated to our representation as well. However, any redundancy in the original network will be eliminated in our representation.

7.1 Our Approach

The function captured by a given neural network can be represented in many ways. We use a Taylor series [9] representation of the function and calculate progressively higher order terms of the series from the weights of the given network. We first illustrate our approach with functions that are quadratic in their inputs. Later, we extend this approach to higher order polynomials, making way to a more general Taylor series representation for an arbitrary unknown function.

In this paper we focus on networks with a single hidden layer, and using a sigmoid activation function. Since such networks can capture any bounded, continuous function, our approach has a wide applicability.



The functional form of the output of the network is

$$Out_o = \sigma\left(\sum_{j=1}^{N} w_{jo} H_j\right) = \sigma_1$$

where

$$Out_o \text{ is the output } o$$

$$o \text{ goes from } 1 \text{ to number of outputs}$$

$$w_{jo} \text{ is the weight connecting hidden node } j \text{ with output node } o$$

$$H_j \text{ is the value of hidden node } j$$

$$H_j = \sigma\left(\sum_{i=1}^{M} w_{ij} x_i\right) = \sigma_2$$

where

$$w_{ij} \text{ is the weight of input } i \text{ and hidden node } j$$

$$x_i \text{ is the value of input } i$$

Our representation of the output using a quadratic function is

$$P_o = a_{00} + \sum_{i=1}^{M} a_{0i} x_i + \sum_{i=1}^{M} \sum_{k=i+1}^{M} a_{ik} x_i x_k + \sum_{i=1}^{M} a_{ii} x_i^2 \qquad (1)$$

where

$$a_{ik} \text{ are all constants}$$

$$x_i \text{ are the inputs}$$

Our task is to compute the values of all the constants in the above equation. From now on, we will assume that the network has a single output node. If there is more than one output node, our approach simply needs to be replicated on each additional output, which is a straightforward extension. In the following, method *i)* stands for the first method of computing the output (of the network) using the sigmoid activation function, and method *ii)* stands for the second method of computing the output (of the equivalent function) from equation (1).

1- $Computing\ a_{00}$

To find this value, we find the output using both functions with all inputs set to zero.

$$H_j = \sigma(0) = \frac{1}{2}$$

$$i)\ Out_o|_{\vec{x}=\vec{0}} = \sigma\left(\frac{\sum_{j=1}^{N} w_{jo}}{2}\right)$$

$$ii)\ P_o|_{\vec{x}=\vec{0}} = a_{00}$$

therefore

$$a_{00} = \sigma\left(\frac{\sum_{j=1}^{N} w_{jo}}{2}\right)$$

2- $Computing\ a_{0i}$

To find this value, we first differentiate the output function with respect to x_i and then set all inputs to zero.

$$i)\ \frac{\partial Out_o}{\partial x_i} = \sigma_1(1-\sigma_1)\sum_{j=1}^{N} w_{jo}\sigma_2(1-\sigma_2)w_{ij}$$

$$=> \frac{\partial Out_o}{\partial x_i}\bigg|_{\vec{x}=\vec{0}} = a_{00}(1-a_{00})\frac{\sum_{j=1}^{N}(w_{jo}w_{ij})}{4}$$

$$ii)\ \frac{\partial P_o}{\partial x_i} = a_{0i} + a_{ik}x_k + 2a_{ii}x_i$$

$$=> \frac{\partial P_o}{\partial x_i}\bigg|_{\vec{x}=\vec{0}} = a_{0i}$$

therefore

$$a_{0i} = a_{00}(1-a_{00})\frac{\sum_{j=1}^{N}(w_{jo}w_{ij})}{4}$$

3- $Computing\ a_{ii}$

To find this value, we find the second derivative of the output function with respect to x_i and then again we set all inputs to zero.

$$i)\ \frac{\partial^2 Out_o}{\partial {x_i}^2}$$

$$= \sigma_1(1-\sigma_1)(1-2\sigma_1)\left(\sum_{j=1}^{N}\big(w_{jo}\sigma_2(1-\sigma_2)w_{ij}\big)\right)^2 + \sigma_1(1$$

$$-\sigma_1)\left(\sum_{j=1}^{N}\big(w_{jo}\sigma_2(1-\sigma_2)(1-2\sigma_2){w_{ij}}^2\big)\right)$$

$$=> \frac{\partial^2 Out_o}{\partial {x_i}^2}\bigg|_{\vec{x}=\vec{0}} = a_{00}(1-a_{00})(1-2a_{00})\left(\frac{\sum_{j=1}^{N}\big(w_{jo}w_{ij}\big)}{4}\right)^2$$

$$ii)\ \frac{\partial^2 P_o}{\partial {x_i}^2} = 2a_{ii}$$

$$=> \frac{\partial^2 P_o}{\partial {x_i}^2}\bigg|_{\vec{x}=\vec{0}} = 2a_{ii}$$

therefore

$$a_{ii} = \frac{1}{2} a_{00}(1 - a_{00})(1 - 2a_{00}) \left(\frac{\sum_{j=1}^{N} \left(w_{jo} w_{ij} \right)}{4} \right)^2$$

4- *Computing* a_{ik}

To find this value, we find the derivative of the output with respect to x_i and then with respect to x_k, and then again we set all inputs to zero.

$$i) \ \frac{\partial^2 Out_o}{\partial x_i \partial x_k} = \sigma_1(1 - \sigma_1)(1 - 2\sigma_1) \left(\sum_{j=1}^{N} \left(w_{jo} \sigma_2 (1 - \sigma_2) w_{ij} \right) \right) \left(\sum_{j=1}^{N} \left(w_{jo} \sigma_2 (1 - \sigma_2) w_{kj} \right) \right) + \sigma_1 (1 - \sigma_1) \left(\sum_{j=1}^{N} \left(w_{jo} \sigma_2 (1 - \sigma_2)(1 - 2\sigma_2) w_{ij} w_{kj} \right) \right)$$

$$=> \ \left. \frac{\partial^2 Out_o}{\partial x_i \partial x_k} \right|_{\vec{x} = \vec{0}} = a_{00}(1 - a_{00})(1 - 2a_{00}) \left(\frac{\sum_{j=1}^{N} \left(w_{jo} w_{ij} \right)}{4} \right) \left(\frac{\sum_{j=1}^{N} \left(w_{jo} w_{kj} \right)}{4} \right)$$

$$ii) \ \frac{\partial^2 P_o}{\partial x_i \partial x_k} = a_{ik}$$

$$=> \ \left. \frac{\partial^2 P_o}{\partial x_i \partial x_k} \right|_{\vec{x} = \vec{0}} = a_{ik}$$

therefore

$$a_{ik} = a_{00}(1 - a_{00})(1 - 2a_{00}) \left(\frac{\sum_{j=1}^{N} \left(w_{jo} w_{ij} \right)}{4} \right) \left(\frac{\sum_{j=1}^{N} \left(w_{jo} w_{kj} \right)}{4} \right)$$

Thus all the coefficients in equation 1 can be computed from the known weights of the network, producing the equivalent quadratic function of the inputs.

7.2 Extending our Approach

It is possible to extend the above method to any polynomial function. We note that the number of constants to be calculated depends on the number of inputs we have and the degree of the polynomial we want to use to capture the network's functionality. The number of constants, *A,* needed for a polynomial of degree *P* given a network of *M* inputs, is:

$$A = \sum_{i=0}^{P} \binom{i+M-1}{M-1}$$

where

$$\binom{a}{b} = \frac{a!}{b!\,(a-b)!}$$

Now suppose we want to compute the coefficient for $x_1^p x_2^q \dots x_M^r$, it is given by

$$a_{12\dots M} = \left(\frac{1}{p!\,q!\dots r!}\right)\left(\frac{\partial^{p+q+\cdots+r} Out_o}{\partial x_1^p x_2^q \dots x_M^r,}\bigg|_{\vec{x}=\vec{0}}\right)$$

$$= \left(\frac{1}{p!\,q!\dots r!}\right)\left(f^{(p+q+\cdots+r)}(a_0)\right)\left(\frac{\sum_{j=1}^{N}(w_{jo}w_{1j})}{4}\right)^p \left(\frac{\sum_{j=1}^{N}(w_{jo}w_{2j})}{4}\right)^q \dots \left(\frac{\sum_{j=1}^{N}(w_{jo}w_{Mj})}{4}\right)^r \quad (2)$$

where

$$Out_o\ is\ our\ normal\ network\ output\ function$$

$$a_0 = \sigma\left(\frac{\sum_{j=1}^{N} w_{jo}}{2}\right)$$

$$f^{(n)}(a) = \sigma^{(n)}(x)\big|_{\sigma(x)=a}$$

7.3 Taylor Series representation

In general, any differentiable function taking a vector as input can be represented as the Taylor series expansion

$$f(x_1, \dots, x_M) = \sum_{j=0}^{\infty} \left\{ \frac{1}{j!} \left[\sum_{i=1}^{M} (x_i - a_i) \frac{\partial}{\partial x_i^{'}} \right]^j f(x_1^{'}, \dots, x_M^{'}) \right\}_{x_1^{'}=a_1 \dots x_n^{'}=a_M} \qquad (3)$$

where

$$M \text{ is the number of inputs}$$

$$j \text{ is the power of the polynomial}$$

To apply this formula to our network, we set *f* to be the network output function which is the same as Out_o. Then we set the vector $a_1 \dots a_n = \vec{0}$. Now, we calculate each term *j = 0, 1, 2…* in the following way:

For *j=0*, we get $f(0, \dots, 0) = Out_o|_{\vec{x}=\vec{0}} = a_0$, as before.

For *j=1*, we get $\sum_{k=1}^{M}\left(x_k \frac{\partial Out_o}{\partial x_k}\Big|_{\vec{x}=\vec{0}} \right)$

For *j=2*, we get

$$\frac{1}{2}\left(\sum_{k=1}^{M} \left(x_k^2 \frac{\partial^2 Out_o}{\partial x_k^2}\bigg|_{\vec{x}=\vec{0}} \right) + 2 * \sum_{k=1}^{M} \sum_{l=k+1}^{M} \left(x_k x_l \frac{\partial^2 Out_o}{\partial x_k x_l}\bigg|_{\vec{x}=\vec{0}} \right) \right)$$

and similarly for *j = 3, 4,*...

In the end, we will get our network output function as a polynomial.

7.4 Multinomial Expansion

In order to compute the *j*th term in the above Taylor series expansion, we use the Multinomial Expansion theorem [12]:

$$\left(\sum_{i=1}^{M} x_i\right)^j = \sum_{k_1,k_2,\dots,k_m} \left(\binom{j}{k_1, k_2, \dots, k_M} x_1^{k_1} x_2^{k_2} \dots x_M^{k_M}\right)$$

where

$$M\ is\ the\ number\ of\ inputs$$

$$j\ is\ the\ power\ of\ the\ polynomial$$

$$k_1 + k_2 + \cdots + k_M = j$$

$$\binom{j}{k_1, k_2, \dots, k_M} = \frac{j!}{k_1!\,k_2!\dots,k_M!}$$

Now, we can replace $\left[\sum_{i=1}^{M}(x_i - a_i)\frac{\partial}{\partial x_i'}\right]^j$ in the formula (3) by:

$$\sum_{k_1,k_2,\dots,k_M} \left(\left(\frac{j}{k_1, k_2, \dots, k_M}\right) x_1^{k_1} x_2^{k_2} \dots x_M^{k_M} \frac{\partial^j}{\partial x_1^{k_1} x_2^{k_2} \dots x_M^{k_M}}\right)$$

It can be shown that the above expression for the j-th term of the Taylor series is the same as equation (2).

The above representation looks so hard and almost impossible to code. This is why we elaborated this formula to reach one that would be simpler to code.

$$\left(\sum_{i=1}^{M} x_i\right)^j = \left[\prod_{i=1}^{M-2}\left(\sum_{r_i=0}^{r_{i-1}}\left(\binom{r_{i-1}}{r_i} x_i^{r_{i-1}-r_i}\right)\right)\right]\left[\sum_{r_{m-1}=0}^{r_{m-2}}\left(\binom{r_{m-2}}{r_{m-1}} x_{m-1}^{r_{m-2}-r_{m-1}} x_m^{r_{m-1}}\right)\right]$$

It can be shown that the above expression for the j-th term of the Taylor series is the same as equation (2), under the following approximation for the k-th derivative of the sigmoid function:

$$\sigma^{(k)} = \sigma \prod_{\tau=1}^{k}(1-\tau\sigma)$$

Although this approximation is valid for lower order terms (first and second), it deviates from the true value increasingly for higher order terms. For instance, the third derivative of s is actually

$$\sigma^{(3)} = \sigma(1-\sigma)(1-6\sigma+6\sigma^2)$$

which is slightly different from the approximation that we use which is

$$\sigma^{(3)} \approx \sigma(1-\sigma)(1-2\sigma)(1-3\sigma)$$

$$= \sigma(1-\sigma)(1-5\sigma+6\sigma^2)$$

Despite this discrepancy, this approximation serves to simplify our algorithm and works well with our intended applications, which seeks mostly lower order terms.

Applications with many inputs seldom require higher order terms (linearly separable), and for those that do, the Taylor series will have many more terms than weights in the network, defeating the purpose of our approach. Consequently, it is important to note that although our approach applies to a wide variety of problems where a neural net has been acquired, it cannot be applied beneficially in many problems where a neural network is a more succinct representation of a function compared to its analytical form.

8 Analysis and Experimentation Plan

The first point of note is that we have ignored bias weights of a neural network in all of the above steps, for simplicity. It is a straightforward matter to incorporate the bias weights in the above equations. Secondly, the equivalent function computed by the above procedure can be very different from the original network. It uses only the polynomial terms (in inputs) that are indispensable in capturing the functionality of the original network, while the latter is not limited to polynomials. For example, consider a neural net for computing the XOR boolean function. For binary inputs *x, y*, our approach yields coefficients that approximately represent *x+y-2xy,* which is equivalent to XOR(*x,y*)*.* Notice that the neural net would need at least 2 hidden units for capturing this function to a high accuracy, necessitating at least 9 weights (including the bias weights). In contrast, the proposed method can capture the same functionality with only 3 non-zero coefficients.

It is also noteworthy that since our method is based on the original network's parameters (and not on the training data), any inaccuracies in the latter will be translated to our representation as well. However, any redundancy in the original network will be eliminated in our representation.

It is also important to state that the number of constants in the equation is proportional to the number of inputs and outputs in the network and has nothing to do with the number of hidden nodes. Taking the XOR example again, it has 2 inputs and one output resulting in 6 constants in case we are using a second degree polynomial which proved sufficient to come up with a decent solution. On the other hand, in the regular representation that would need approximately 9 hidden nodes resulting in 20 values that need to be stored. The number of floating point numbers saved in this small example is quite high. Given a network with 5 inputs and 3 outputs and n hidden nodes:

Number Of Hidden Nodes	**Number Of Float Numbers (Weights)**	**Number Of Constants Using power 2**
10	80	21*3
20	160	21*3
30	240	21*3
40	320	21*3
50	400	21*3

On the other hand, the approximation will require much more constants with every power increment. Given now the same network (5 inputs and 3 outputs) but this time 10 hidden nodes (80 float numbers or weights):

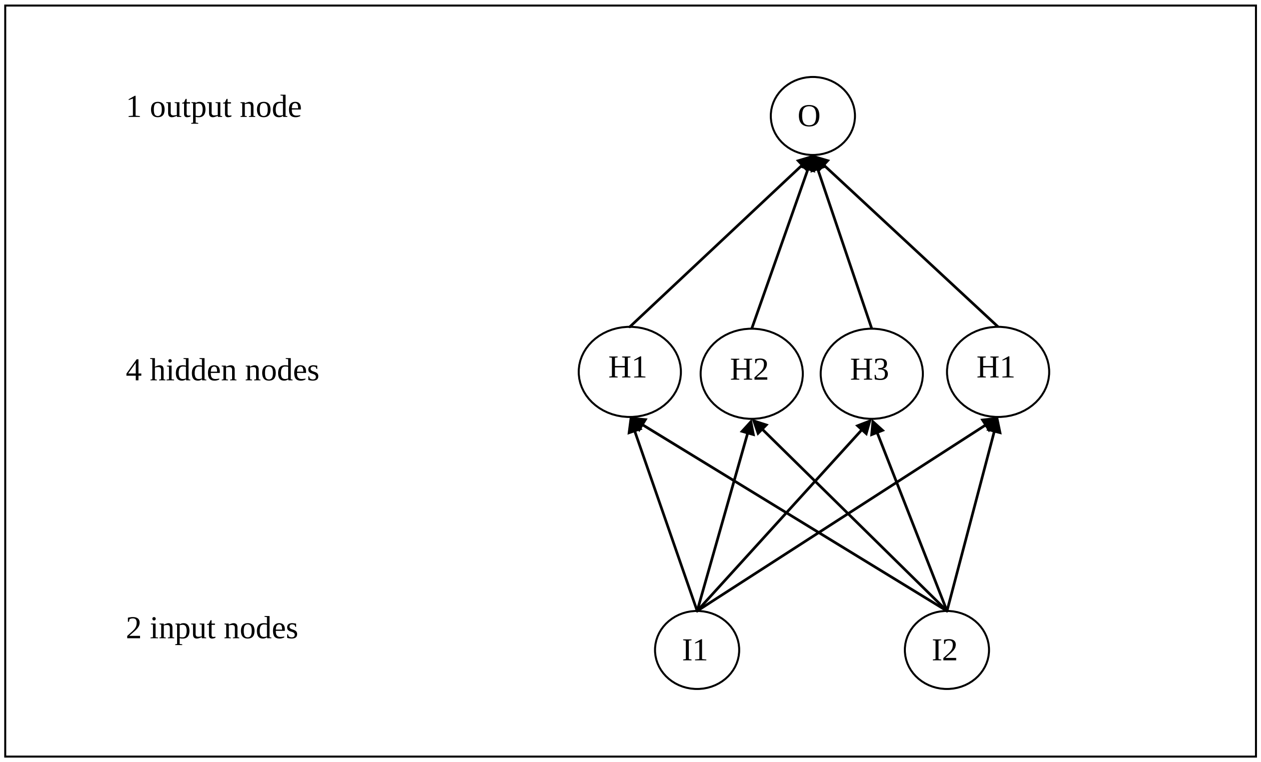
Power	**Number Of Constants**
1	6*3
2	21*3
3	56*3
4	126*3
5	252*3

As you can see, the number of constants grew quickly every time we increased the power (degree) of the polynomial. In most cases, a second degree polynomial gave us acceptable results.

We tested our approach in another way. We took an arbitrary neural network with M input nodes, N hidden nodes and O output nodes. We initialized all weights between all nodes to a random value ranging between -1 & 1. We used the resultant network as input to our algorithm and tested for results for random values to check the error margin between the outputs of the two algorithms. The average margin of error was around 0.0001 using a 2nd degree polynomial. At times, the error was as low as 0.0000001. A drawback occurred with every high degree polynomials i.e. degree 10. While using high degree polynomials, we noticed that at time the network will sometimes generate results of an error margin of 10 or 100 at times. After looking at the problem and analyzing it, we figured out that precision for small values multiplied at high degrees can result in such abnormalities especially that the derivative approximation can result in errors. Further studies for the program can be done as future work to improve the stability of the program. Until now, the program has proved useful in many cases.

8.1 Example

Take an example of a network with 2 input nodes, 4 hidden nodes and one output node. This network results in 12 float numbers to store. Using a second degree equation, we need 6 constants to represent the network.



Given the trained network weights:

a) I1H1 = -1.0; I1H2 = 0.5; I1H3 = -0.6; I1H4 = 0.4

b) I2H1 = 1.0; I2H2 = -0.7; I2H3 = 0.9; I2H4 = 0.25

c) H1O = 0.35; H2O = -0.95; H3O = 0.65; H4O = -0.15

$$P_o = a_{00} + a_{01}x_1 + a_{02}x_2 + a_{12}x_1x_2 + a_{11}x_1^2 + a_{22}x_2^2$$

$$i)\ a_{00} = \sigma\left(\frac{\sum_{j=1}^{4} HjO}{2}\right) = \sigma\left(\frac{H1O + H2O + H3O + H4O}{2}\right) = \sigma\left(\frac{-0.1}{2}\right) = 0.487503$$

$$ii)\ a_{01} = a_{00}(1 - a_{00})\frac{\sum_{j=1}^{4}\big((HjO)(I1Hj)\big)}{4}$$

$$= a_{00}(1 - a_{00})\frac{\big((H1O)(I1H1) + (H2O)(I1H2) + (H2O)(I1H2) + (H2O)(I1H2)\big)}{4}$$

$$= \sigma\left(\frac{-0.1}{2}\right)\left(1 - \sigma\left(\frac{-0.1}{2}\right)\right)\left(\frac{-1.275}{4}\right) = -0.079637721$$

$$iii)\ a_{02} = a_{00}(1 - a_{00})\frac{\sum_{j=1}^{4}\big((HjO)(I2Hj)\big)}{4}$$

$$= a_{00}(1 - a_{00})\frac{\big((H1O)(I2H1) + (H2O)(I2H2) + (H2O)(I2H2) + (H2O)(I2H2)\big)}{4}$$

$$= \sigma\left(\frac{-0.1}{2}\right)\left(1 - \sigma\left(\frac{-0.1}{2}\right)\right)\left(\frac{1.5625}{4}\right) = 0.097595237$$

$$iv)\ a_{11} = \frac{1}{2}a_{00}(1 - a_{00})(1 - 2a_{00})\left(\frac{\sum_{j=1}^{4}\big((HjO)(I1Hj)\big)}{4}\right)^2$$

$$= \frac{1}{2}a_{00}(1 - a_{00})(1$$

$$- 2a_{00})\left(\frac{\big((H1O)(I1H1) + (H2O)(I1H2) + (H2O)(I1H2) + (H2O)(I1H2)\big)}{4}\right)^2$$

$$= \sigma\left(\frac{-0.1}{2}\right)\left(1 - \sigma\left(\frac{-0.1}{2}\right)\right)\left(1 - 2\sigma\left(\frac{-0.1}{2}\right)\right)\left(\frac{-1.275}{4}\right)^2 = 0.00031724045$$

$$v)\ a_{22}=\frac{1}{2}a_{00}(1-a_{00})(1-2a_{00})\left(\frac{\sum_{j=1}^{4}\big((HjO)(I2Hj)\big)}{4}\right)^2$$

$$=\frac{1}{2}a_{00}(1-a_{00})(1$$

$$-2a_{00})\left(\frac{\big((H1O)(I2H1)+(H2O)(I2H2)+(H2O)(I2H2)+(H2O)(I2H2)\big)}{4}\right)^2$$

$$=\frac{1}{2}\sigma\left(\frac{-0.1}{2}\right)\left(1-\sigma\left(\frac{-0.1}{2}\right)\right)\left(1-2\sigma\left(\frac{-0.1}{2}\right)\right)\left(\frac{1.5625}{4}\right)^2=0.00047643989$$

$$vi)\ a_{12}$$

$$=\frac{1}{2}a_{00}(1-a_{00})(1-2a_{00})\left(\frac{\sum_{j=1}^{4}\big((HjO)(I1Hj)\big)}{4}\right)\left(\frac{\sum_{j=1}^{4}\big((HjO)(I2Hj)\big)}{4}\right)$$

$$=\frac{1}{2}a_{00}(1-a_{00})(1$$

$$-2a_{00})\frac{\big((H1O)(I1H1)+(H2O)(I1H2)+(H2O)(I1H2)+(H2O)(I1H2)\big)}{4}$$

$$\frac{\big((H1O)(I2H1)+(H2O)(I2H2)+(H2O)(I2H2)+(H2O)(I2H2)\big)}{4}$$

$$=\frac{1}{2}\sigma\left(\frac{-0.1}{2}\right)\left(1-\sigma\left(\frac{-0.1}{2}\right)\right)\left(1-2\sigma\left(\frac{-0.1}{2}\right)\right)\left(\frac{-1.275}{4}\right)\left(\frac{1.5625}{4}\right)$$

$$=-0.00077755004$$

So the polynomial is:

$$P_o=\ 0.487503-0.079637721x_1+0.097595237x_2-0.00077755004\,x_1x_2$$

$$+\,0.00031724045x_1^2+0.00047643989x_2^2$$

Inputs (I1, I2)	**Sigmoid**	**2nd degree equation**	**Error**
(0,0)	0.487503	0.487503	0.0
(1,0)	0.411762	0.408182	0.00358
(0,1)	0.578944	0.585574	0.00663
(1,1)	0.505544	0.505476	0.000068
(0.5,0.5)	0.496495	0.496485	0.000010
(0.25,0.75)	0.539744	0.540932	0.001188
(0.75,0.25)	0.452628	0.452236	0.000392

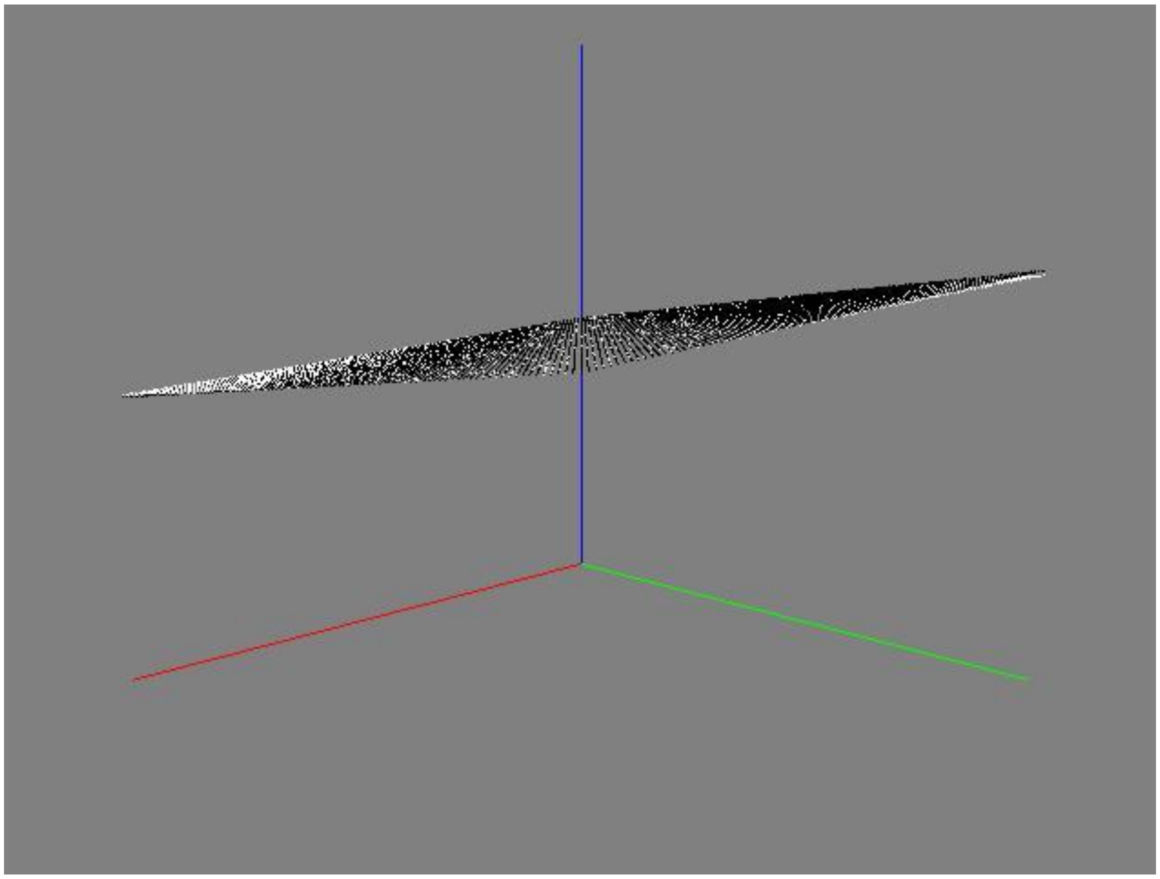
Two tests were made on point going from 0 to 1; one with an increment of 0.01 (a total of 10000 points) and the other with an increment of 0.001 (a total of 1000000 points). The results came as follows:

	0.01	**0.001**
Largest Error	0.006445	0.006612
Error < 0.00001	944	93994
Error < 0.000001	228	22625
Error < 0.0000001	26	2454
Error < 0.00000001	3	246

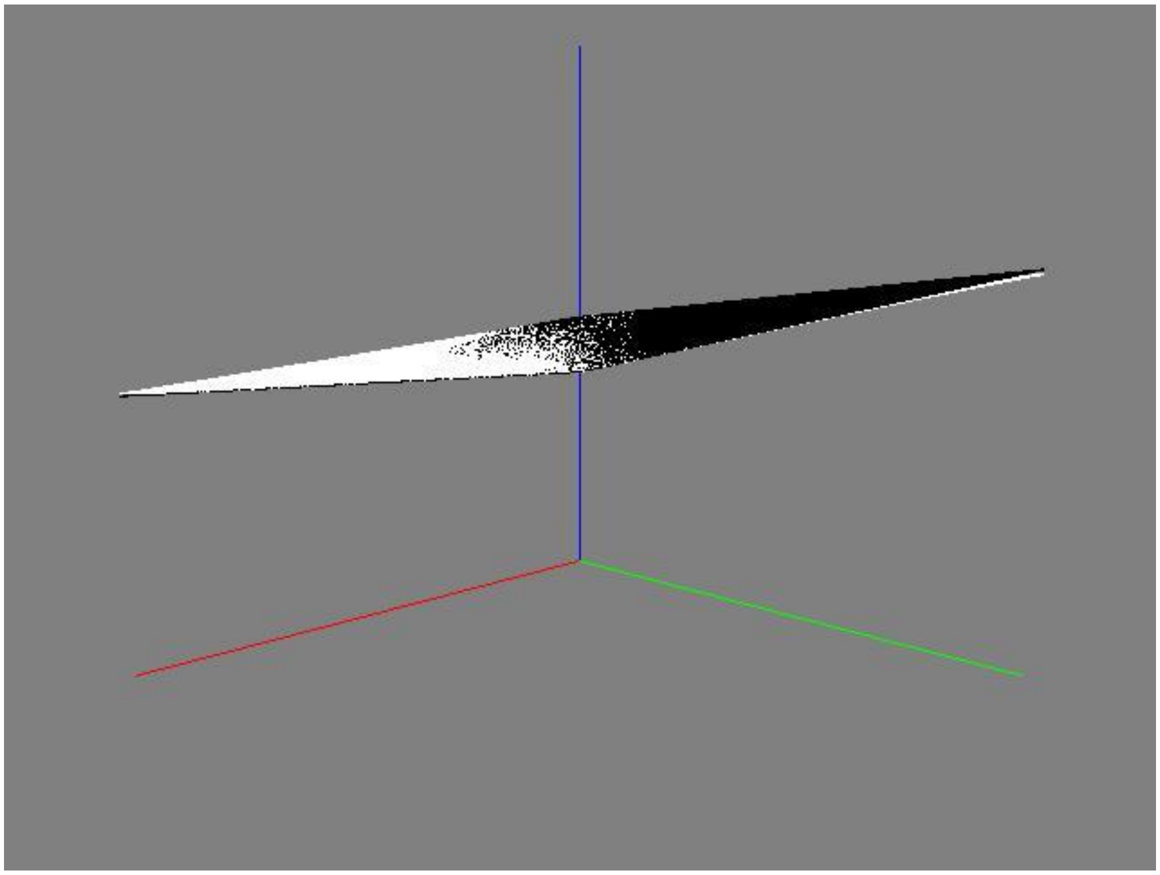
The tables above show some good results. The error term is minimal in many of the cases but still, there are some cases where the error is large but as shown the maximum error was about 0.6%.

In both graphs, x-axis is in red, y-axis is in green, z-axis is in blue, original's network output in white and polynomial output in black.

The graph representing the reached polynomial (in black) using a 0.01 increment on points going from 0 to 1 is compared to the original network's function (in white) in the following figure:



The graph representing the reached polynomial (in black) using a 0.001 increment on points going from 0 to 1 is compared to the original network's function (in white) in the following figure:



Both graphs show that our result and the result generated from the network are really close in values in this particular case.

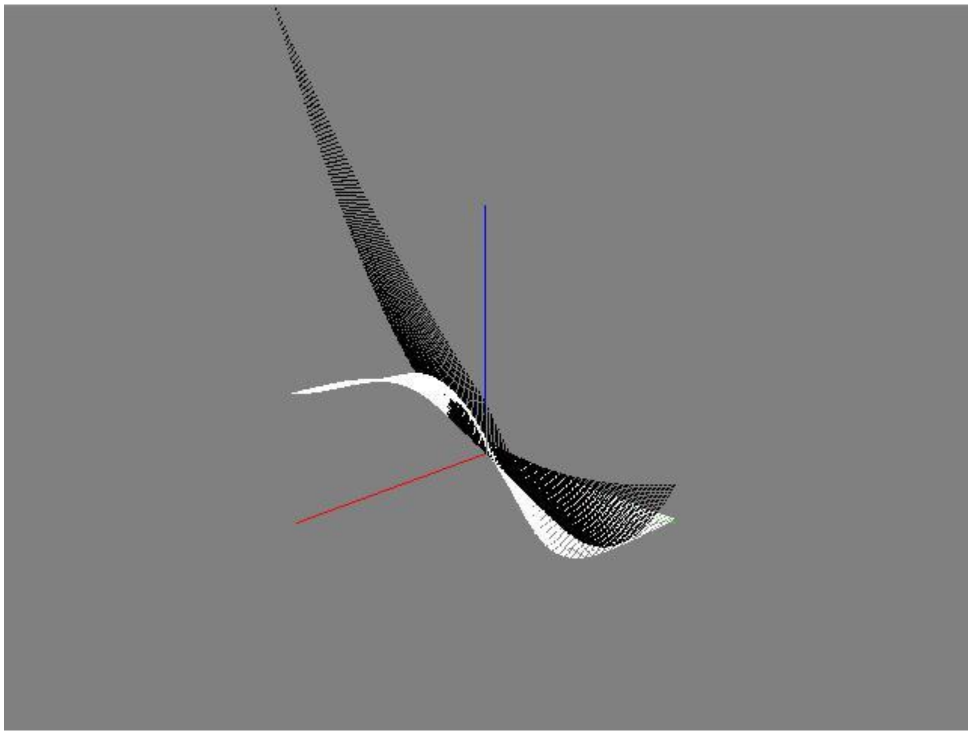
8.2 More Results

Several other results were reached using random weights for the connections on different networks with different topologies. For each network topology, 1000 samples were taken randomly and the points were incremented from 0 to 1 using a 0.01 increment. The table below shows the average error in each case:

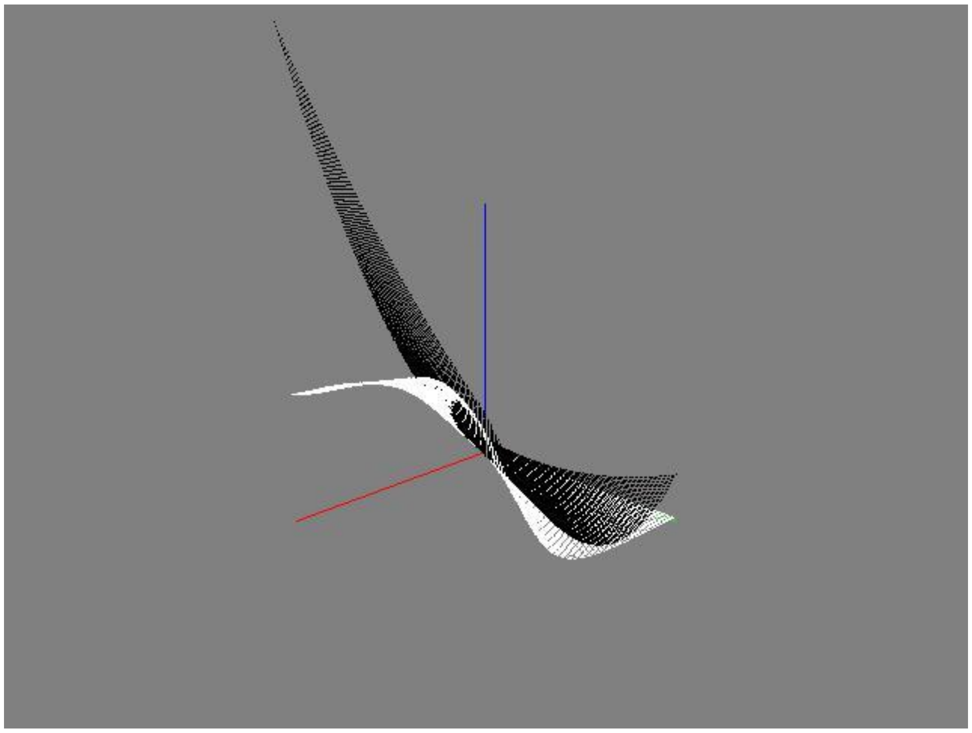
Network Topology	**Average Error**
2 inputs, 4 hidden, 1 output	0.00514368
2 inputs, 6 hidden, 1 output	0.00663347
3 inputs, 6 hidden, 1 output (with one of the inputs fixed to 0.0)	0.00655637
3 inputs, 6 hidden, 1 output (with one of the inputs fixed to 0.5)	0.00818654
3 inputs, 6 hidden, 1 output (with one of the inputs fixed to 1.0)	0.012859

8.3 Drawbacks

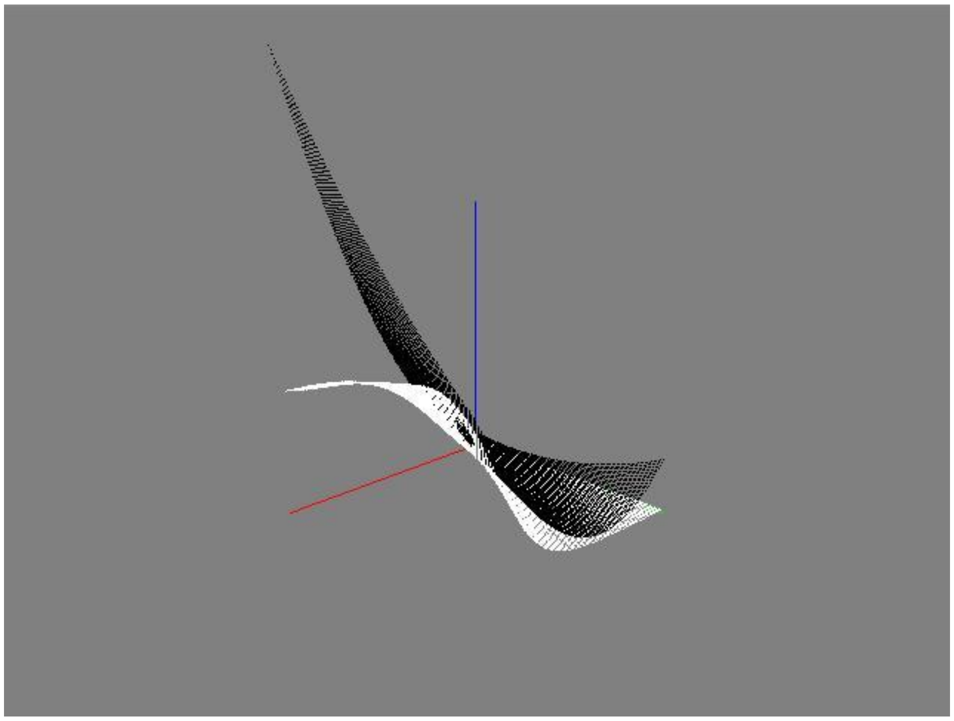
In this section, we will focus on the drawbacks of our algorithm; that is illustrating the points where our algorithm has failed. It is noticeable that the algorithm fails every time we compute results that are far from the origin of approximation (in our case $\vec{0}$). The following example shows clearly the flaws of this algorithm at some of its points. We considered a more complicated neural network with 3 inputs, 10 hidden nodes and 1 output node. We will show 11 examples (figures) each having one of the inputs fixed at a point incrementing by 0.1 from 0 to 1 and the other 2 varying from 0 to 1 using a 0.01 increment.



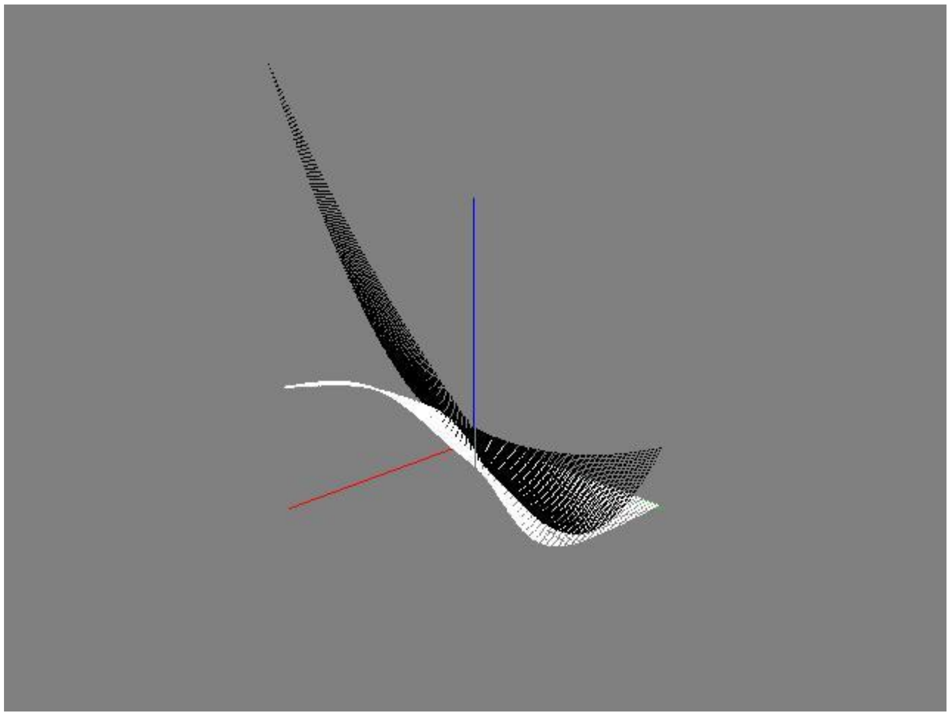
One of the weights is fixed at 0.0. Largest error is 1.228560



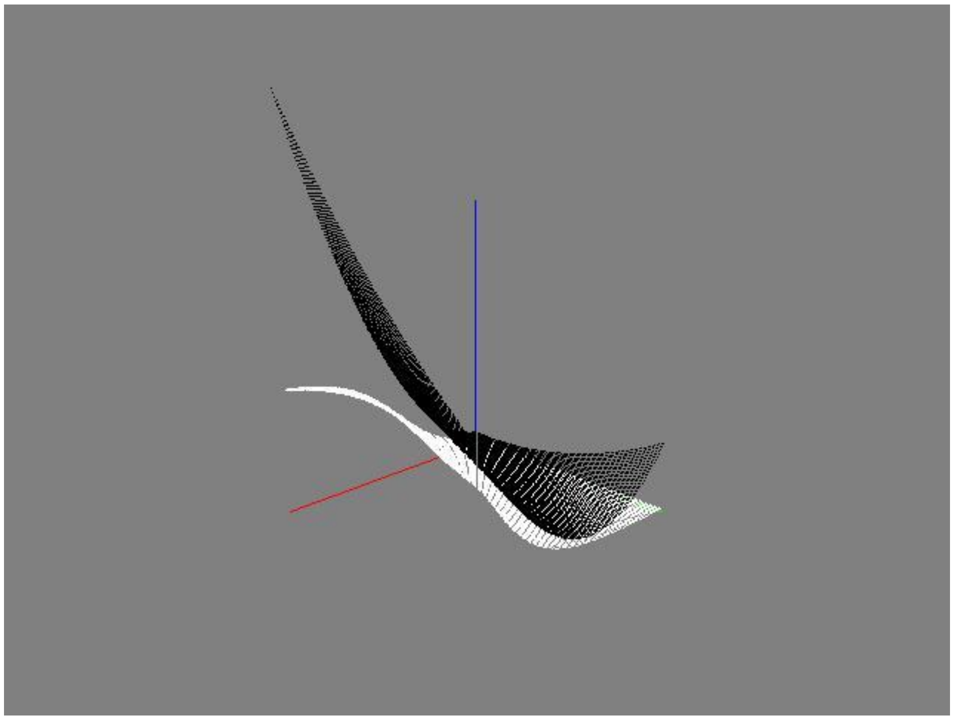
One of the weights is fixed at 0.1. Largest error is 1.172808



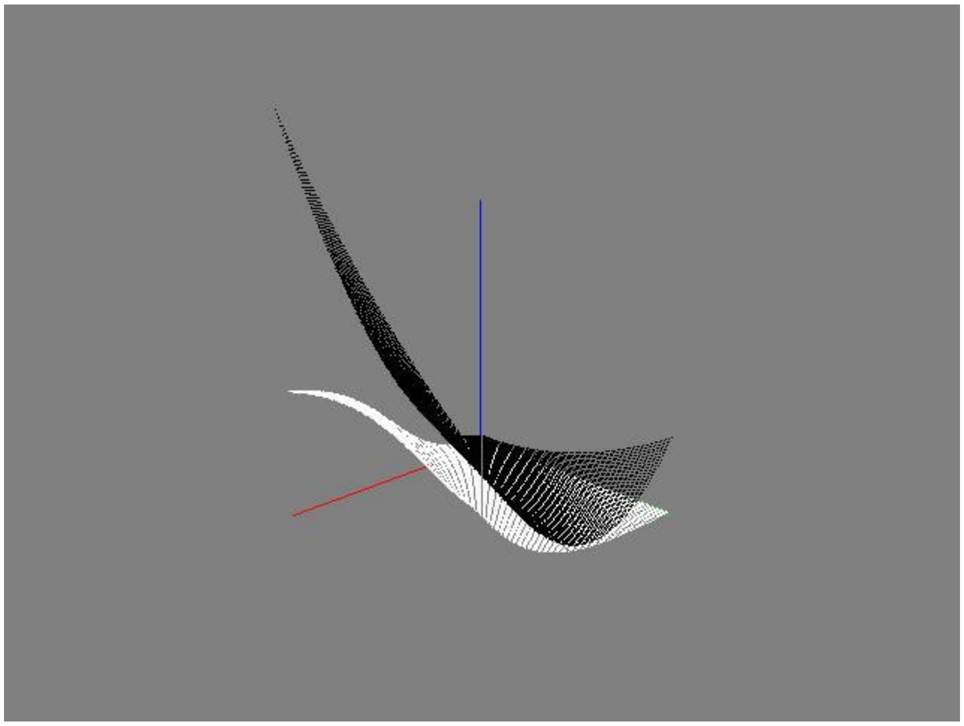
One of the weights fixed to 0.2. Largest error is 1.111509



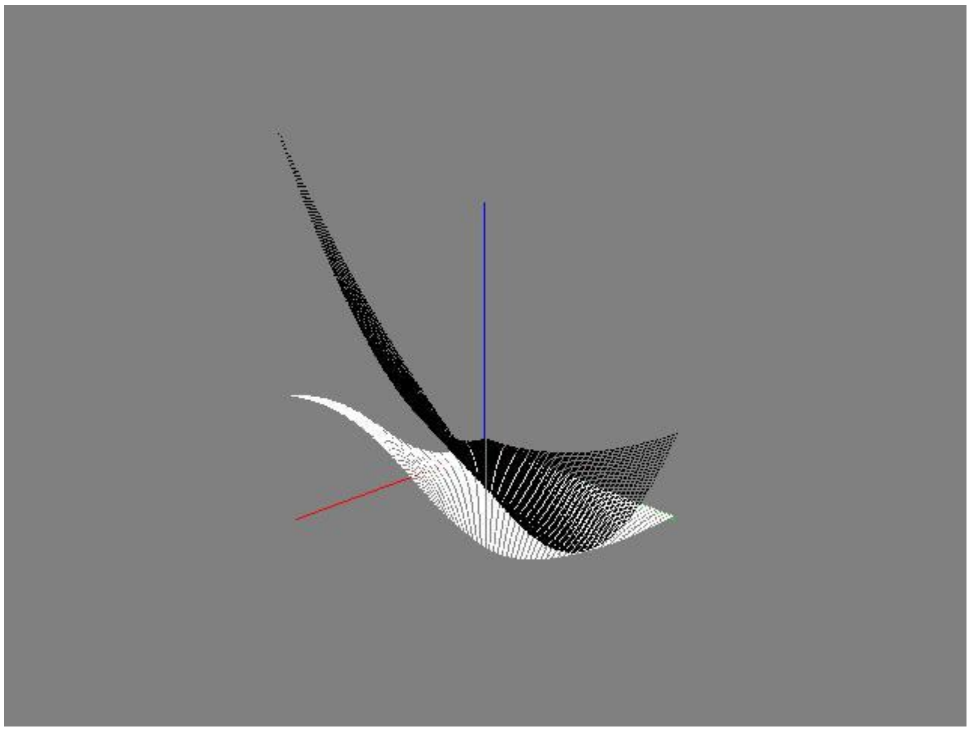
One of the weights fixed to 0.3. Largest error is 1.045970



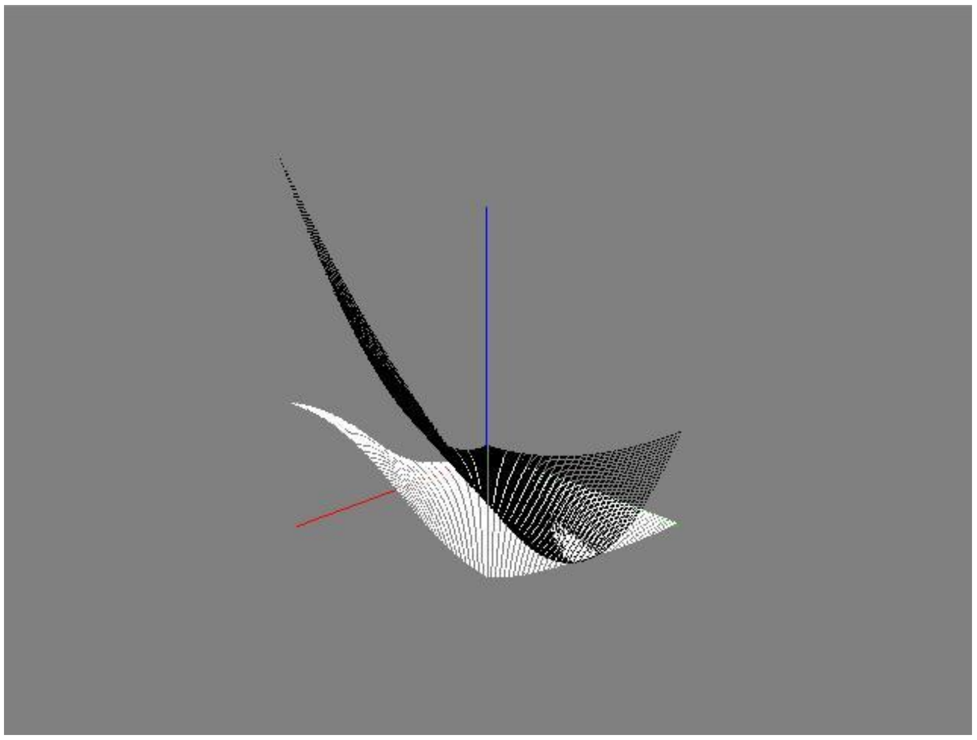
One of the weights fixed to 0.4. Largest error is 0.978146



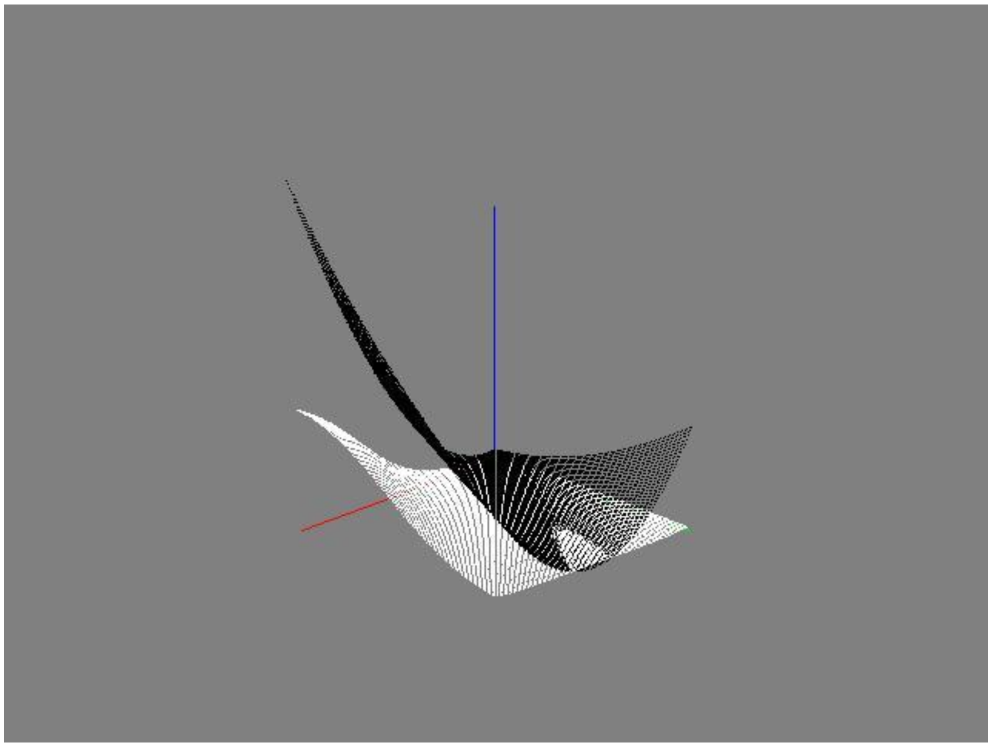
One of the weights fixed to 0.5. Largest error is 0.910438



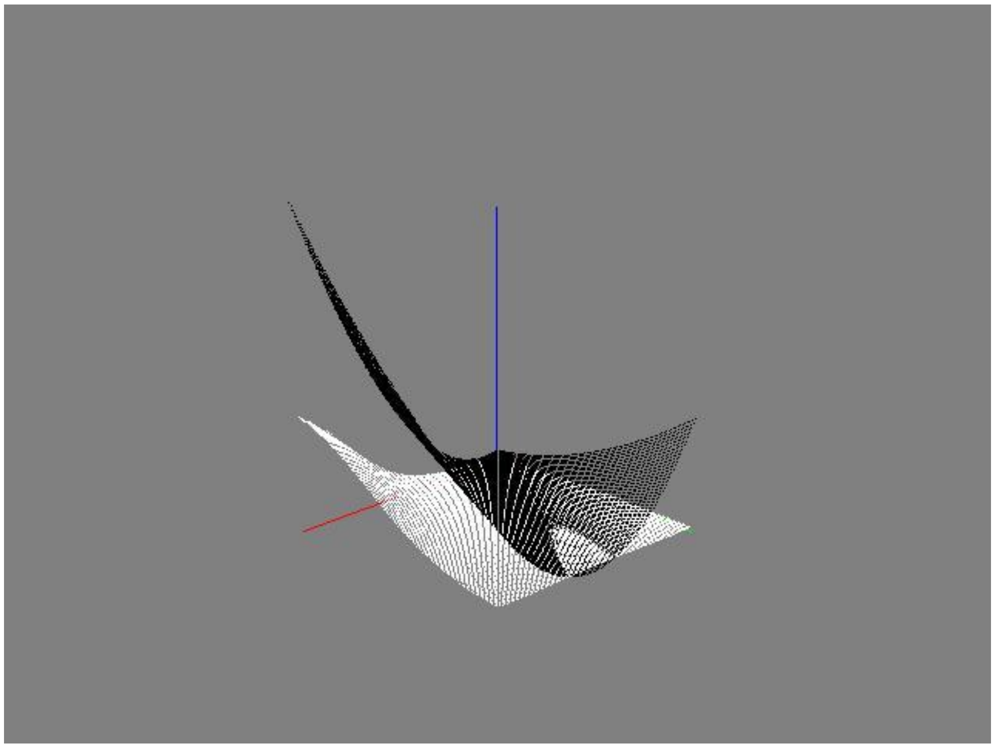
One of the weights fixed to 0.6. Largest error is 0.845444



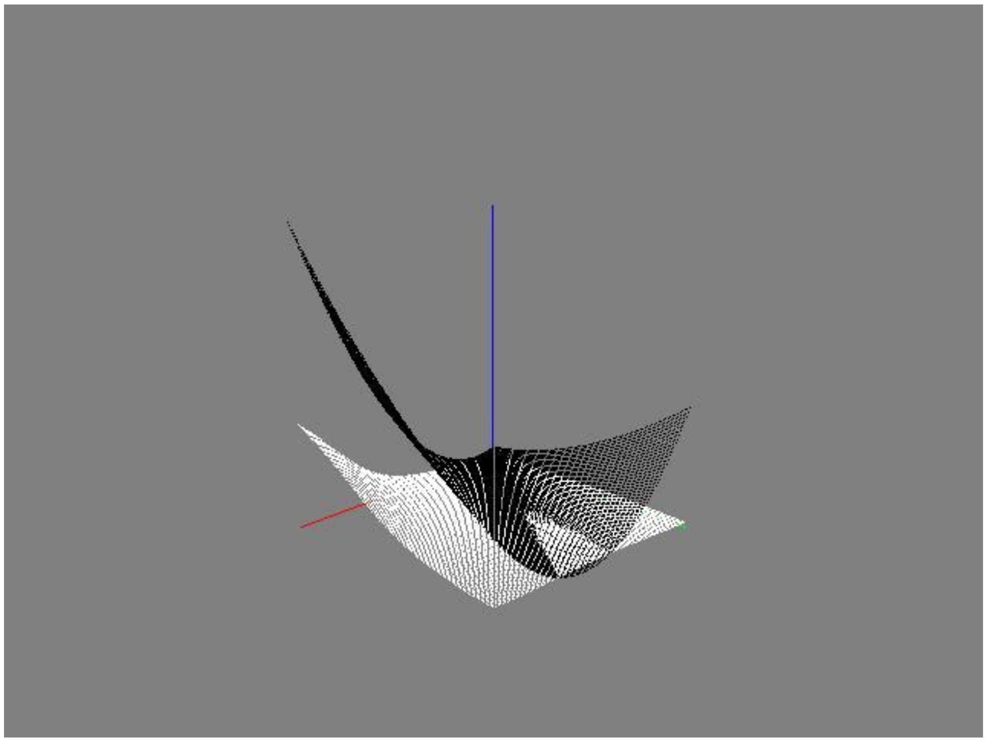
One of the weights fixed to 0.7. Largest error is 0.785721



One of the weights fixed to 0.8. Largest error is 0.733625



One of the weights fixed to 0.9. Largest error is 0.691170



One of the weights fixed to 1.0. Largest error is 0.659660

As you can see from this example, our approximation failed at several points in the network thus resulting in large errors. This can be due to the fact that the degree of the polynomial used cannot approximate the network's original function. We should have used a higher order polynomial.

A solution to this problem may be dissecting our space into smaller partitions where each partition has its own approximation function at a certain point. Currently, our algorithm approximates the neural network at the point $\vec{0}$. This has resulted in all the errors in our program. We think that if we approximate the network at several points in space and then for each given point, we calculate its result using the function that gives us the best result. In other words, we group our points into partitions, and we associate to each partition a polynomial that best represents the network within this partition. The drawback of this method is storage space since we now have to store more constants to represent all polynomials around our space.

9 Conclusion

In conclusion, we find that our algorithm holds a new and nice representation of the neural network. It proved to be memory efficient for those interested in saving memory space. It proved to give results close enough to the actual neural network with a tolerance to the error resulted.

As you can notice, the error term gradually increased when we started getting farther from the origin. This is due to the fact that the Taylor Series approximation was done at the vector $\vec{0}$. This means that the function will lose precision far from that point.

Nevertheless this algorithm has proved to be useful in several cases where the problem concentrates around the origin.

If we go back to the example in 8.1, its computational needs are as follows:

$$Out_o = \sigma\left(\sum_{j=1}^{4} w_{jo}\sigma\left(\sum_{i=1}^{2} w_{ij}x_i\right)\right)$$

i) 12 multiplications

ii) 12 additions

iii) 5 divisions

iv) 5 exponentials

In our approach this computational needs will be reduced to:

$$P_o = a_{00} + \sum_{i=1}^{2} a_{0i}x_i + \sum_{i=1}^{2}\sum_{k=i+1}^{2} a_{ik}x_ix_k + \sum_{i=1}^{2} a_{ii}x_i^2$$

i) 8 multiplications

ii) 5 additions

The difference is clearly seen now. Not only our approach is memory efficient, but it needs fewer computations to achieve an acceptable result. It's still true that our approach will fail once we get far from the origin where we originally approximated our function.

Throughout this paper, we have looked at several methods to better improve the representation of our neural networks as well as find better ways to find the best topology for any given network. Our approach dealt with the problem from another window where we tried simplifying the network to a simple polynomial that can be read by anyone. Our experiments succeeded at times with flying numbers but also fell into problems in several other cases. Further work will continue to improve more the efficiency of this algorithm.

10 Future Directions

This approach approximates the sigmoid function using the Taylor series at the origin. Our future work includes expanding the representation and test among differences in approximations across other points.

This approach deals with approximating a neural network whose activation function is the regular sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. An expansion to that approximation could done by using an approximation for any sigmoid function $\sigma(x) = \frac{1}{1+e^{-nx}}$, where n is a scaling value for the network.

Another thing to do is apply this algorithm to a more complex problem such as a game. Currently, neural networks are not included much in games due to their complexity and high storage space that they require when modeling complex agent behaviors. This approach could be used to help enter the neural networks more into the gaming world because of its simplicity to program and the results would still be the same.

Another approach to be added to the program would be working on neural networks that use different activation functions. But, since the sigmoid function is the most commonly used as an activation function for neural networks, we have decided to use it here.

As mentioned in section 8.3, one of the most innovative ideas to be touched in the future is dissecting our space (any dimension) into several partitions. Since in our approach the error tends to increase when we get farther from the origin of approximation, we can create an area around this origin where this polynomial will be used. Then we approximate the network at another point in space being in the center of the next partition. In this way, we think we can guarantee good results along our entire network's space. The drawback of this method is storage space because then we will have to save multiple polynomials representing each region in space, but run-time computations will be the same all over the regions.

11 References

[1] Colin Campbell, "*Constructive learning techniques for designing neural network systems*," in Neural Network Systems, Techniques and Applications ed. C. T. Leondes (Academic Press, San Diego, 1997)

[2] Kenneth O. Stanley, Risto Miikkulainen, "*Evolving neural networks through augmenting topologies*", Evolutionary Computation 10 (2002) 99--127

[3] Kenneth O. Stanley, Bobby D. Bryant & Risto Miikkulainen, "*Evolving Neural Network Agents in the NERO Video Game*". In Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05). 2005

[4] Scott E. Fahlman & Christian Lebiere, "*The Cascade-Correlation Learning Architecture*". CMU-CS-90-100. August 29, 1991

[5] Babak Hassibi, David G. Stork & Gregpry J. Wolff, "*Optimal Brain Surgeon and General Network Pruning*". 1993

[6] Peter Morgan, Bruce Curry & Malcom Beynon, “Pruning neural networks by minimization of the estimated variance” *European Journal of Economic and Social Systems* **14** N° 1 (2000) 1-16

[7] Rudy Setiono, Wee Kheng Leow & James Y. L. Thong, “*Opening the Neural Network Black Box: An Algorithm For Extracting Rules From Function Approximating Artificial Neural Networks*”. International Conference on Information Systems 2000

[8] Tom Mitchell, *Machine Learning* (McGraw Hill, 1997).

[9] Al Shenk, *Calculus and Analytic Geometry* (Pearson Scott Foresman; 4th edition (July 2000))

[10] Geoffrey Stephenson, *Worked Examples in Mathematics for Scientists and Engineers* (Longman, March 1985)

[11] E. Drougge, E. & J. Wroldsen, "*A Robust Algorithm for Pruning Neural Networks*", Gjovik College Preprint , November, (1994).

[12] George Chrystal, *Algebra* (A. & C. Black 1889)

[13] Carlos Gershenson, “*Aritificial Neural Networks for Beginners*”, August 2003

[14] Weiyu Yi, "*Artificial Neural Networks*", October 7, 2005

[15] Martin Anthony & Peter L. Barlett, *Neural Network Learning: Theoretical Foundations* (Cambridge University Press; 1 edition (January 15, 1999))