TRAINING AND SOURCE CODE GENERATION FOR ARTIFICIAL NEURAL NETWORKS

BY

BRANDON WINRICH

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2015

MASTER OF SCIENCE THESIS

OF

BRANDON WINRICH

APPROVED:

Thesis Committee:

Major Professor ______________________________

____

____

____
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2015

ABSTRACT

The ideas and technology behind artificial neural networks have advanced considerably since their introduction in 1943 by Warren McCulloch and Walter Pitts. However, the complexity of large networks means that it may not be computationally feasible to retrain a network during the execution of another program, or to store a network in such a form that it can be traversed node by node. The purpose of this project is to design and implement a program that would train an artificial neural network and export source code for it so that the network may be used in other projects.

After discussing some of this history of neural networks, I explain the mathematical principals behind them. Two related training algorithms are discussed: backpropagation and RPROP. I also go into detail about some of the more useful activation functions.

The actual training portion of the project was not self implemented. Instead, a third party external library was used: Encog, developed by Heaton Research. After analyzing how Encog stores the weights of the network, and how the network is trained, I discuss how I used several of the more important classes. There are also details of the slight modifications I needed to make to one of the classes in the library.

The actual implementation of the project consists of five classes, all of which are discussed in the fourth chapter. The program has two inputs by the user (a config file and a training data set), and returns two outputs (a training error report and the source code).

The paper concludes with discussions about additional features that may be implemented in the future. Finally, an example is given, proving that the program works as intended.

TABLE OF CONTENTS

ABSTRACT ii

TABLE OF CONTENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1 Background Information 1

1.1 Predecessors to ANNs 1

1.1.1 Perceptron 2

1.2 What is an Artificial Neural Network? 3

1.3 Justification for Study 5

List of References 6

2 Mathematical Elements of Networks 7

2.1 Training 7

2.1.1 Backpropagation 7

2.1.2 Resilient Propagation 9

2.2 Activation Functions 11

2.2.1 Linear 11

2.2.2 Sigmoid 12

2.2.3 Hyperbolic Tangent 12

2.2.4 Elliott 13

List of References 15

Page

3 Java library Encog 16

3.1 Overview 16

3.2 How Encog Stores Weights 16

3.3 Training 18

3.4 Some Individual Classes 20

3.4.1 TrainingSetUtil 20

3.4.2 BasicNetwork 23

3.4.3 FlatNetwork 24

3.4.4 BasicLayer 25

3.4.5 BasicMLDataSet 25

3.4.6 BasicMLDataPair 26

3.4.7 ActivationFunction 26

4 Implementation 28

4.1 Overview 28

4.2 Assumptions/Design Choices 28

4.3 Inputs 29

4.3.1 Config File 29

4.3.2 Training Data Set 32

4.4 Outputs 32

4.4.1 Training Error Report 33

4.4.2 Source Code 33

4.5 Individual classes 35

4.5.1 NeuralGenerator 35

Page

4.5.2 LayerInfo . . . 36

4.5.3 OutputWriter . . . 37

4.5.4 OutputWriterTxt . . . 38

4.5.5 OutputWriterJava . . . 39

List of References . . . 41

5 Future work and conclusions . . . 42

5.1 Categorical classification . . . 42

5.2 Additional output formats . . . 42

5.3 Non-command line inputs . . . 43

5.4 Normalization . . . 44

5.5 Conclusions . . . 44

List of References . . . 47

APPENDIX

Source Code . . . 49

A.1 NeuralGenerator.java . . . 49

A.2 LayerInfo.java . . . 64

A.3 OutputWriter.java . . . 67

A.4 OutputWriterTxt.java . . . 71

A.5 OutputWriterJava.java . . . 77

A.6 TrainingSetUtil.java (modified) . . . 84

BIBLIOGRAPHY . . . 88

LIST OF TABLES

Table		Page
1	Average number of required epochs	11
2	Elliott vs TANH	15

LIST OF FIGURES

Figure		Page
1	McCulloch-Pitts model of a neuron	1
2	An example of a neural network	4
3	An example of a neural network with bias nodes	5
4	The two sides of a computing unit[1]	7
5	Extended network for the computation of the error function[1]	8
6	Result of the feed-forward step[1]	9
7	Backpropagation path up to output unit j[1]	9
8	Linear activation function	11
9	Sigmoid activation function	12
10	Hyperbolic tangent activation function	13
11	Comparison between Elliott (solid) and sigmoid (dotted) activation functions	14
12	Comparison between Symmetric Elliott (solid) and hyperbolic tangent (dotted) activation functions	14
13	Neural network with labeled weight indexes	16
14	BasicNetwork.getWeight()	17
15	Class hierarchy for training	18
16	Comparison between modified and original code	22
17	Sample from first output file (.txt)	33
18	Sample from first output file (.csv)	33
19	Sample second output file (.txt)	39
20	Sample second output file (.java)	41

Figure		Page

21	test.csv	44
22	output1.csv	45
23	graph of output1.csv	45
24	Results from NeuralGenerator.java	46
25	TestModule.java	46
26	Results from output2.java	46
27	Sample config file	47
28	output2.java	48

CHAPTER 1

Background Information

1.1 Predecessors to ANNs

The history of most neural network research can be traced back to the efforts of Warren McCulloch and Walter Pitts. In their 1943 paper ‘A logical Calculus of Ideas Immanent in Nervous Activity’[1], McCulloch and Pitts introduced the foundation of a neuron, a single piece of the nervous system, which would respond once a certain threshold had been reached. This model of a neuron is still used today.

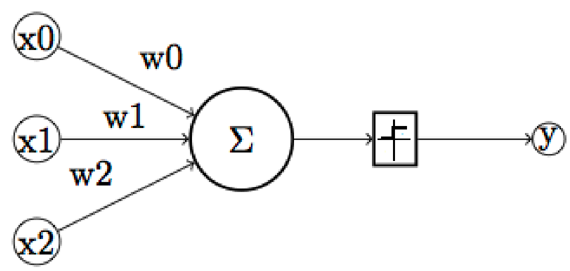

Figure 1: McCulloch-Pitts model of a neuron

The input values to a neuron are all given individual weights. These weighted values are summed together and fed into a threshold function: every value greater than 0 returns a value of 1, and all other values return 0.

In 1949, neuropsychologist Donald Hebb published his book ‘The Organization of Behavior’. Hebb postulated that “When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased.” [2]. Hebbian learning influenced research in the field of machine learning, especially in the area of unsupervised learning.

In 1958, Frank Rosenblatt developed the Perceptron. More information on this will be presented in the following subsection.

In 1959, Bernard Widrow and Marcian Hoff developed a working model they called ADALINE (ADAptive LINEar), as well as a more advanced version known as MADALINE (Multiple ADAptive LINEar)[3]. These models were some of the first to be applied to real world problems (such as eliminating echoes on phone lines), and may still be in use today.[4]

Breakthroughs in neural network research declined starting in 1969, when Marvin Minsky and Seymour Papert published their book 'Perceptrons: an Introduction to Computational Geometry'. In this book, Minsky and Papert claimed Rosenblatt's perceptron wasn't as promising as it was originally believed to be. For example, it was unable to correctly classify an XOR function. While this book did introduce some new ideas about neural networks, it also contributed to what was known as 'the dark age of connectionism' or an AI winter, as there was a lack of major research for over a decade.

Interest in artificial networks declined, and the focus of the community switched to other models such as support vector machines. [5]

1.1.1 Perceptron

In his 1958 paper, Frank Rosenblatt considered 3 questions[6]:

1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

The first question wasn't addressed as he believed it "is in the province of sensory physiology, and is the only one for which appreciable understanding has been achieved." The other two questions became the basis of his concept of a perceptron (which he compared to the retina in an eye).

A perceptron functions as a single neuron, accepting weighted inputs and an unweighted bias, the output of which is passed to a transfer function. This step function evaluates to 1 if the value is positive, and either 0 or -1 if the value is negative (the exact value may vary depending on the model). After iterations with a learning algorithm, the perceptron calculates a decision surface to classify a data set into two categories.

The perceptron learning algorithm is as follows[7]:

1. Initialize the weights and threshold to small random numbers.

2. Present a pattern vector $(x_1, x_2, ..., x_n)^t$ and evaluate the output of the neuron.

3. Update the weights according to $w_j(t+1) = w_j(t) + \eta(d - y)x_j$, where d is the desired output, t is the iteration number, and η $(0.0 < \eta < 1.0)$ is the gain (step size).

Steps two and three are repeated until the data set has been properly classified. Unfortunately, due to the nature of the perceptron, it will only work with data that is linearly separable.

1.2 What is an Artificial Neural Network?

An artificial neural network (sometimes referred to as an ANN, or just a neural network) is a machine learning model inspired by biological neural networks (such as the central nervous system).

Neural networks fall under the supervised learning paradigm. In supervised learning, the network is presented with pairs of data, input and output. The goal is to be able to map the input to the output, training in a way that minimizes the error between the actual output and the desired output. More information may be found in section 2.1.

Each piece of the network is known as a neuron. Neural networks still use the McCulloch-Pitts model of a neuron (see figure 1). The inputs into a node are the values from the previous layer (or the input values, if the layer in question is the input layer). Each value is multiplied by the associated weight, and then those products are summed together ($\sum w_i x_i$). Rather than being fed into a threshold function, an activation function is used. This allows for a wider ranger of potential output values, instead of just 0 or 1. The output value of the neuron can be used as the input into the next layer, or as the output for the network.

Neural networks consist of at least three layers: an input layer, at least one hidden layer, and an output layer:

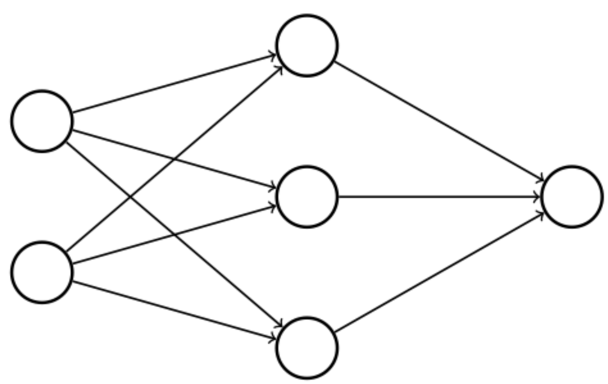

Figure 2: An example of a neural network

It is also possible to have bias nodes. These nodes hold a constant value (often +1), and act only as an input to another neuron (they do not have any inputs or activation functions associated with themselves).

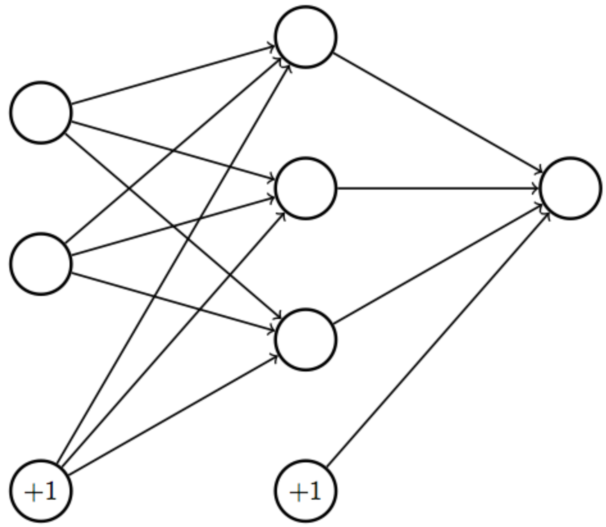

Figure 3: An example of a neural network with bias nodes

1.3 Justification for Study

My main interest in machine learning pertains to the realm of video games. Artificial intelligence is an important aspect of every game (except those which are exclusively multiplayer, with no computer-controlled agents). 5-60% of the CPU is utilized by AI-related processes, and this number has been known climb as high as 100% for turn-based strategy games[8]. While some modern games utilize machine learning, most of this is done before the game is published (rather than the training occurring during runtime). According to Charles and McGlinchey, "online learning means that the AI learns (or continues to learn) whilst the end product is being used, and the AI in games is able to adapt to the style of play of the user. Online learning is a much more difficult prospect because it is a real-time process and many of the commonly used algorithms for learning are therefore not suitable."[9]

The project that I have completed focuses on generating the source code for an artificial neural network, which is directly applicable to the field of gaming. With the actual training occurring during the development phase, it makes sense to have a program that can create the network, separate from the rest of the project. The source code that it outputs then allows the network to be used within the context of a game. The other benefit of such a program is that it allows the neural network to be used without having to maintain the structure of the network. Reducing the results of the network down to mathematical formulas results in faster computation times than having to walk through the nodes of a network (as stored in multiple classes or data structures). The results of this project have been tested in a Quake II environment.

List of References

[1] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943. [Online]. Available: http://dx.doi.org/10.1007/BF02478259

[2] D. Hebb, "The organization of behavior; a neuropsychological theory." 1949.

[3] B. Widrow, M. E. Hoff, *et al.*, "Adaptive switching circuits." 1960.

[4] C. Clabaugh, D. Myszewski, and J. Pang. "History: The 1940's to the 1970's." [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html

[5] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF00994018

[6] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[7] A. K. Jain, J. Mao, and K. Mohiuddin, "Artificial neural networks: A tutorial," 1996.

[8] L. Galway, D. Charles, and M. Black, "Machine learning in digital games: a survey," *Artificial Intelligence Review*, vol. 29, no. 2, pp. 123–161, 2008.

[9] D. Charles and S. Mcglinchey, "The past, present and future of artificial neural networks in digital games," *Proceedings of the 5th international conference on computer games: artificial intelligence, design and education*, pp. 163–169, 2004.

CHAPTER 2

Mathematical Elements of Networks

2.1 Training

2.1.1 Backpropagation

The concept of the backpropagation algorithm was first developed by Paul Werbos in his 1974 PhD thesis, 'Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences'.

Much of the math in this section comes from Raúl Rojas' book 'Neural Networks - A Systematic Introduction'[1]

The backpropagation algorithm works by using the method of gradient decent. In order to do this, it needs to use an activation function which is differentiable. This is a change from the perceptron, which used a step function. One of the more popular activations functions is the sigmoid function. This and other alternatives will be explored in section 2.2.

In order to make the calculations easier, each node is considered in two separate parts. Rojas calls this a *B-diagram* (or backpropagation diagram). As seen in figure 4, the right side calculates the output from the activation function, while the left side computes the derivative.

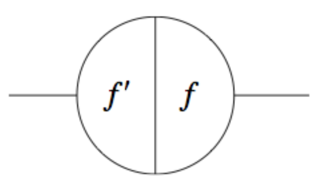

Figure 4: The two sides of a computing unit[1]

Rather than calculating the error function separately, the neural network is extended with an additional layer used to calculate the error internally (as seen

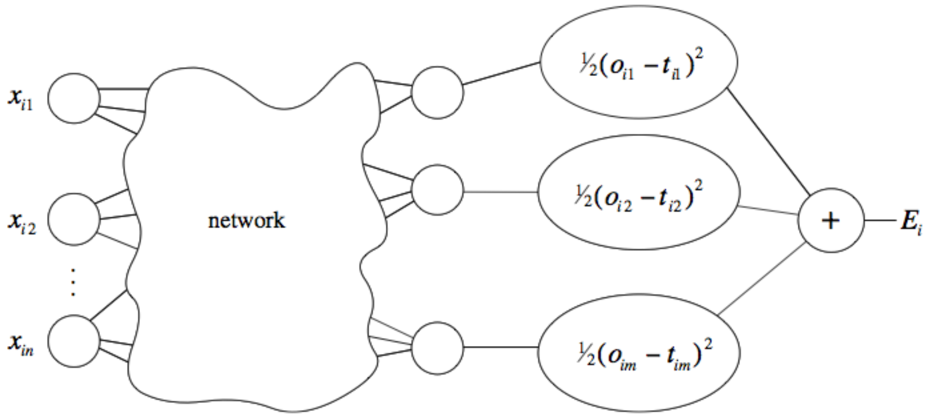

Figure 5: Extended network for the computation of the error function[1]

in figure 5). The equation for the error function is $E = \frac{1}{2}\sum_{i=1}^{p} ||\mathbf{o}_i - \mathbf{t}_i||^2$, where $\mathbf{o}_i$ is the output value from node i, and $\mathbf{t}_i$ is the target value. Keeping in mind the separation of the nodes as previously mentioned, the derivative calculated in the left portion will be $(\mathbf{o}_i - \mathbf{t}_i)$.

The backpropagation algorithm consists of four steps:

1. Feed-forward computation
2. Backpropagation to output layer
3. Backpropagation to hidden layer(s)
4. Weight updating

In the first step, the algorithm is processed in a straight forward manner, with the output from one node being used as the input to the next node, as seen in figure 6.

Generally speaking, backpropagation retraces through the network in reverse. Since the network is being run backwards, we evaluate using the left side of the node (the derivative). Instead of outputs being used as the the inputs to the next node, outputs from a node are multiplied by the output of previous nodes.

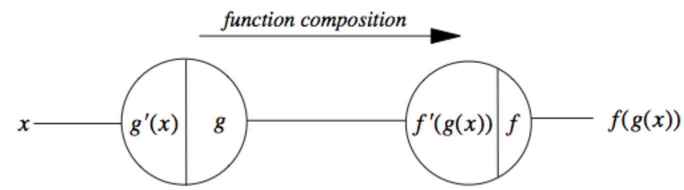

Figure 6: Result of the feed-forward step[1]

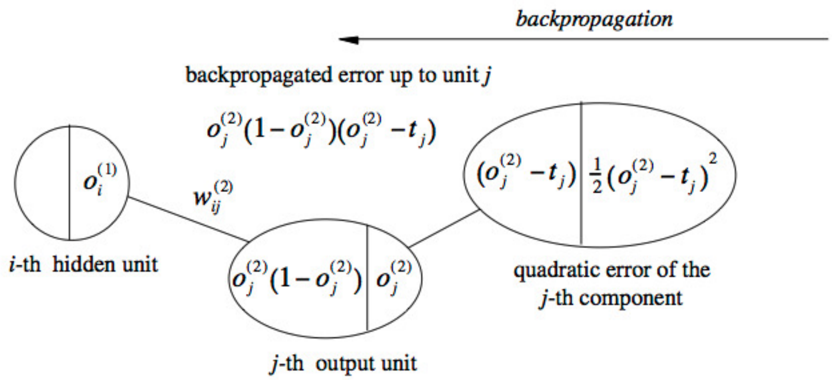

Figure 7: Backpropagation path up to output unit j[1]

We extended the network to calculate the error function, so for the output layer we use that derivative as an input, as seen in figure 7.

Backpropagation for the hidden layer(s) acts in the same way, using the values from the output layer as its input.

The final step is weight updating. The formula for updating the weight w_{ij} (the weight between node i and node j) is $\Delta w_{ij} = -\gamma o_i \delta_j$, where γ is the learning rate, o_i is the output from node i, and δ_j is the error from node j.

A possible variation is the inclusion of a momentum variable η. This can help make the learning rate more stable: $\Delta w_{ij}(t) = -\gamma o_i \delta_j + \eta \Delta w_{ij}(t-1)$

2.1.2 Resilient Propagation

A promising alternative to backpropagation is resilient propagation (often referred to as RPROP), originally proposed by Martin Riedmiller and Heinrich

Braun in 1992. Instead of updating the weights based on how large the partial derivative of the error function is, the weights are updated based on whether the partial derivative is positive or negative.

First, the change for each weight is updated based on if the derivative has changed signs. If such a change has occurred, that means the last update was too large, and the algorithm has passed over a local minimum. To counter this, the update value will be decreased. If the sign stays the same, then the update value is increased.

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^{+} * \Delta_{ij}^{(t-1)}, \text{ if } \frac{\delta E}{\delta w_{ij}}^{(t-1)} * \frac{\delta E}{\delta w_{ij}}^{(t)} > 0 \\ \eta^{-} * \Delta_{ij}^{(t-1)}, \text{ if } \frac{\delta E}{\delta w_{ij}}^{(t-1)} * \frac{\delta E}{\delta w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)}, \text{ else} \end{cases} \quad \text{where } 0 < \eta^{-} < 1 < \eta^{+}.$$

Typically, η^{+} is assigned a value of 1.2, and η^{-} is assigned a value of 0.5.

Once the update value is determined, the sign of the current partial derivative is considered. In order to bring the error closer to 0, the weight is decreased if the partial derivative is positive, and increased if it is negative.

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)}, \text{ if } \frac{\delta E}{\delta w_{ij}}^{(t)} > 0 \\ +\Delta_{ij}^{(t)}, \text{ if } \frac{\delta E}{\delta w_{ij}}^{(t)} < 0 \\ 0, \text{ else} \end{cases}$$

At the end of each epoch, all of the weights are updated:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \Delta w_{ij}^{(t)}$$

The exception to this rule is if the partial derivative has changed signs, then the previous weight change is reversed. According to Reidmiller and Braun, "due to that 'backtracking' weight-step, the derivative is supposed to change its sign once again in the following step. In order to avoid a double punishment of the update-value, there should be no adaptation of the update-value in the succeeding step."

$$\Delta w_{ij}^{(t)} = -\Delta w_{ij}^{(t-1)}, \text{ if } \frac{\delta E}{\delta w_{ij}}^{(t-1)} * \frac{\delta E}{\delta w_{ij}}^{(t)} < 0$$

In most cases, the update value is limited to a specific range, with an upper limit of $\Delta_{max} = 50.0$ and a lower limit of $\Delta_{min} = 1e^{-6}$.

Reidmiller and Braun provide tested RPROP against several other popular algorithms: backpropagation (BP), SuperSAB (SSAB), and Quickprop (QP)[2]:

Problem	10-5-10	12-2-12	9 Men's Morris	Figure Rec.
BP (best)	121	>15000	98	151
SSAB (best)	55	534	34	41
QP (best)	21	405	34	28
RPROP (std)	30	367	30	29
RPROP (best)	19	322	23	28

Table 1: Average number of required epochs

2.2 Activation Functions

2.2.1 Linear

One of the simpler activation functions is the linear function:

$f(x) = x$

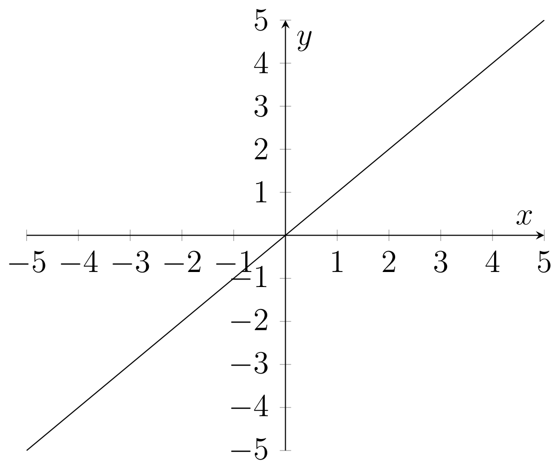

Figure 8: Linear activation function

This activation is very simple, and isn't used very often. The input is directly transferred to the output without being modified at all. Therefore, the output range is $\mathbb{R}$. The derivative of this activation function is $f'(x) = 1$.

A variation on this is the ramp activation function. This function has an upper and lower threshold, where all values below the lower threshold are assigned a certain value and all values above the upper threshold are assigned a different value

(0 and 1 are common). The result is something similar to the step function used in the perceptron, but with a linear portion in the middle instead of a disjuncture.

2.2.2 Sigmoid

One of the more common activation functions is the sigmoid function. A sigmoid function maintains a shape similar to the step function used in perceptrons (with horizontal asymptotes at 0 at 1). However, the smooth curve of the sigmoid means that it is a differentiable function, so it can be used in backpropagation (which requires an activation function to have a derivative).

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

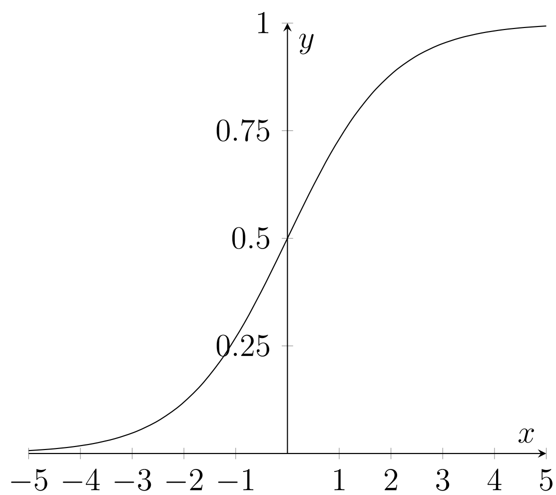

Figure 9: Sigmoid activation function

The output range of this activation function 0 to 1. The derivative of this activation function is $f'(x) = f(x) * (1 - f(x))$.

2.2.3 Hyperbolic Tangent

The hyperbolic tangent function has a similar shape to the sigmoid function. However, its lower horizontal asymptote is at -1 instead of 0. This may be more useful with some data sets, where use of a sigmoid activation function does not produce any negative numbers.

$f(x) = tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$

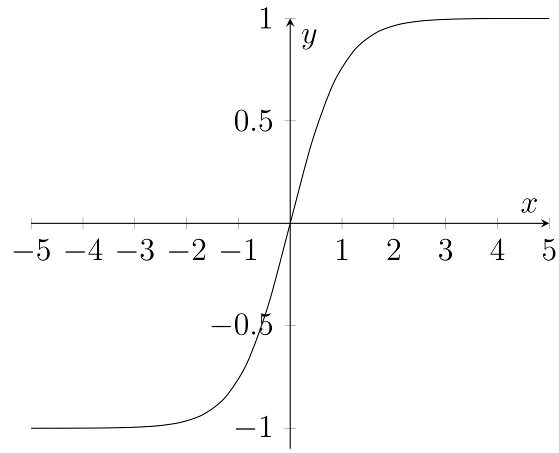

Figure 10: Hyperbolic tangent activation function

The output range of this activation function is -1 to 1. The derivative of this activation function is $f'(x) = 1 - f(x) * f(x)$.

2.2.4 Elliott

Elliott activation functions were originally proposed by David L. Elliott in 1993 as more computationally effective alternatives to the sigmoid and hyperbolic tangent activation functions.[3]

Encog provides two such activation functions: Elliott and Symmetric Elliott. In all of the cases below, s is the slope, which has a default value of 1 (although this can be changed).

The Elliott activation function serves as an alternative to the sigmoid activation function:

$f(x) = \frac{0.5(x*s)}{1+|x*s|} + 0.5$

Just as the sigmoid activation function, this produces an output range of 0 to 1. The derivative of this activation function is $f'(x) = \frac{s}{2*(1+|x*s|)^2}$

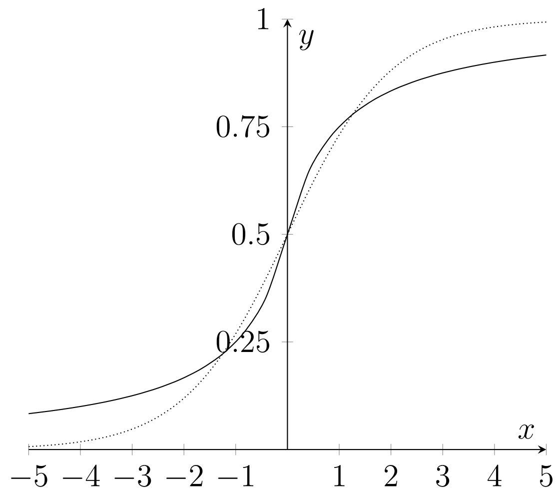

Figure 11: Comparison between Elliott (solid) and sigmoid (dotted) activation functions

The Symmetric Elliott activation functions serves as an alternative to the hyperbolic tangent activation function:

$f(x) = \frac{x*s}{1+|x*s|}$

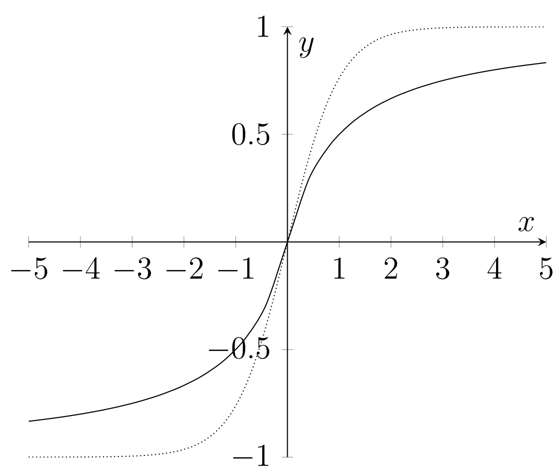

Figure 12: Comparison between Symmetric Elliott (solid) and hyperbolic tangent (dotted) activation functions

Just as the hyperbolic tangent activation function, this produces an output range of -1 to 1. The derivative of this activation function is $f'(x) = \frac{s}{(1+|x*s|)^2}$

Heaton Research (the company that makes the Encog library) provided some interesting statistics on the efficiency of this activation function[4]:

Activation Function	**Total Training Time**	**Avg Iterations Needed**
TANH	6,168ms	474
ElliottSymmetric	2,928ms	557

Table 2: Elliott vs TANH

While the Symmetric Elliot required more iterations of training in order to reach the desired error, the time it took for each training iteration was much less than the hyperbolic tangent, resulting in the network being trained in effectively half the time. Although computational power has increased considerably since David L. Elliott first proposed these activation functions (earlier versions of Encog approximated the value of the hyperbolic tangent because it was faster than Java's built-in TANH function), they can still be useful for training large networks and/or data sets.

According to the javadoc comments for the two classes, these activation functions approach their horizontal asymptotes more slowly than their traditional counterparts, so they "might be more suitable to classification tasks than predictions tasks".

List of References

[1] R. Rojas, "The backpropagation algorithm," in *Neural Networks.* Springer, 1996, pp. 149–182.

[2] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The rprop algorithm," in *Neural Networks, 1993., IEEE International Conference on.* IEEE, 1993, pp. 586–591.

[3] D. L. Elliott, "A better activation function for artificial neural networks," 1993.

[4] J. Heaton. "Elliott activation function." September 2011. [Online]. Available: http://www.heatonresearch.com/wiki/Elliott_Activation_Function

CHAPTER 3

Java library Encog

3.1 Overview

Encog is an external machine learning library. Originally released in 2008, Encog is developed by Heaton Research (run by Jeff Heaton). The current Java version is 3.3 (released on October 12, 2014). Encog is released under an Apache 2.0 license.

3.2 How Encog Stores Weights

In their simplest form, Encog stores the weights for a neural network in an array of doubles inside a `FlatNetwork` object.

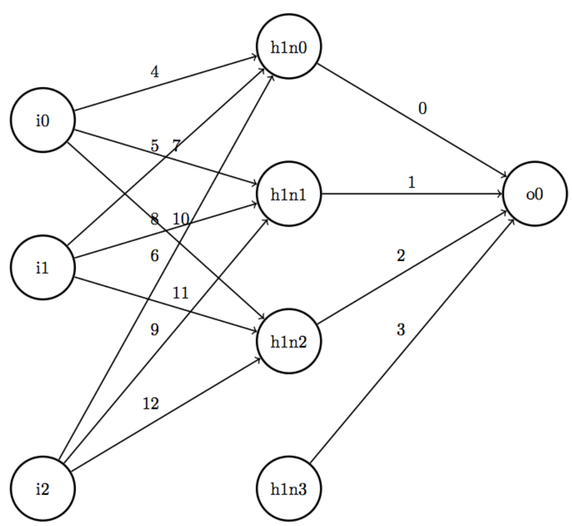

Figure 13: Neural network with labeled weight indexes

As seen in Figure 13, the order of the weights is determined by a combination of the reversed order of the layers and the regular order of the nodes (with the biases being the last node in a layer, if applicable). For example, the network in Figure 13 consists of an input layer of 2 nodes and a bias, a hidden layer of 3 nodes and a bias, and an output layer of 1 node. The first 4 weights in the array are the weights going from the hidden layer to the output layer (`weights[0]` connects h1n0 to o0, `weights[1]` connects h1n1 to o0...). The next 3 weights connect the input layer to the first hidden node (`weights[4]` connects i0 to h1n0, `weights[5]` connects i1 to h1n0...). This continues in this fashion until the final weight in the array, `weights[12]`, which connects the input bias node to the last regular hidden node (i2 to h1n2).

To access all of the weights at once, the `BasicNetwork` class provides a `dumpWeights()` method. It may also be useful to use the `weightIndex` array from the `FlatNetwork`, which indicates where in the weights array each layer starts.

Alternately, the `BasicNetwork` class has a `getWeight()` method, which allows a user to access the weight from one specific node to another. This is the method that I utilized in my implementation:

```
/**
 * Get the weight between the two layers.
 * @param fromLayer The from layer.
 * @param fromNeuron The from neuron.
 * @param toNeuron The to neuron.
 * @return The weight value.
 */
public double getWeight(final int fromLayer,
      final int fromNeuron,
      final int toNeuron) {
   this.structure.requireFlat();
   validateNeuron(fromLayer, fromNeuron);
   validateNeuron(fromLayer + 1, toNeuron);
   final int fromLayerNumber = getLayerCount() - fromLayer - 1;
   final int toLayerNumber = fromLayerNumber - 1;

   if (toLayerNumber < 0) {
      throw new NeuralNetworkError(
            ''The specified layer is not connected to another layer: ''
                  + fromLayer);
   }

   final int weightBaseIndex
      = this.structure.getFlat().getWeightIndex()[toLayerNumber];
   final int count
      = this.structure.getFlat().getLayerCounts()[fromLayerNumber];
   final int weightIndex = weightBaseIndex + fromNeuron
         + (toNeuron * count);

   return this.structure.getFlat().getWeights()[weightIndex];
}
```

Figure 14: BasicNetwork.getWeight()

3.3 Training

Encog has several different ways to train networks. For the purpose of this project, we will focus on on propagation training.

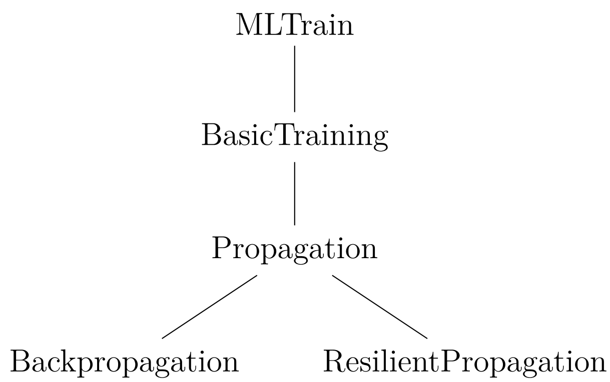

Figure 15: Class hierarchy for training

As seen in figure 15, training in Encog utilizes several different classes. Each method of training that is utilized has its own class (`Backpropagation` and `ResilientPropagation`), with most of the work being done in the parent class `Propagation`. There are other forms of training available, so `Propagation` extends the `BasicTraining` class, and all forms of training must implement the `MLTrain` interface.

Most of the training is done through the `Propagation.iteration()` method, which calls several helper methods. There are two different versions of this method: a default version and a version that accepts the number of iterations as a parameter. In order to do a single iteration, the default form of the method calls the alternate version and passes 1 as a parameter.

The first method to be invoked is `BasicTraining.preIteration()`. This method increments a counter called `iteration`, which keeps track of the current iteration. It also calls upon the `preIteration()` method for any strategies that may be in use.

Strategies are additional methods of training that may be used to enhance the performance of a training algorithm. The ResilientPropagation class doesn't use any, but the Backpropagation class allows for the use of two strategies: SmartLearningRate and SmartMomentum. These strategies will be used to attempt to calculate the learning rate and momentum if they have not been specified upon creation of the network. However, since both of these variables are assigned values by the user (with a default momentum of 0 if the use of that variable is not desired), training strategies are not used in the implementation of this project.

The next method to be invoked is Propagation.rollIteration(). However, the use of this method is superfluous. While the BasicTraining class has a variable which keeps track of the current iteration, the Propagation class has its own copy of that same variable (rather than inheriting the value from its parents class. The rollIteration() method increments this duplicate variable. Unfortunately, where the variable in the BasicTraining class is utilized in accessor and mutator methods, the same variable in the Propagation class is not used anywhere outside of the rollIteration() method.

Following this is the Propagation.processPureBatch() method (large data sets may want to make use of the processBatches() method, which uses a portion of the training set rather than the entire thing). This in turn calls upon Propagation.calculateGradients() and Propagation.learn().

Propagation.calculateGradients() iterates through the network and calculates the gradient at each portion of the network (for more information, see section 2.1.1). This is done through the GradientWorker class. The advantage of this is that it allows for multithreaded calculations. Different portions of the network that don't rely on each other (for example, nodes in the same layer do not have any weights connecting them) can be calculated in parallel using an array of GradientWorkers. This project only uses single threaded calculations, so the array has a size of 1.

`Propagation.learn()` uses the gradients to update the weights for the network. Different algorithms update weights in different ways (see sections 2.1.1 and 2.1.2 for more information), so this is an abstract method, with each child class having its own implementation.

The last method to be used is `BasicTraining.postIteration()`. This method calls upon the `postIteration()` method for any strategies if applicable. The `ResilientPropagation` class has its own `postIteration()` method, which stores the error in the `lastError` variable, because RPROP uses this to check for sign changes in the error during subsequent iterations.

3.4 Some Individual Classes

The following sections will go into detail about how I used some of the classes from the Encog library. It wasn't feasible to describe all of the classes used in the program, but these seven were the most important.

3.4.1 TrainingSetUtil

`TrainingSetUtil` (org.encog.util.simple.TrainingSetUtil) is the only class that I modified.

The main method that I used was `loadCSVTOMemory()`. This method takes a CSV file and loads that into a `ReadCSV` object. Then, that object is converted into something that I could use for training: an `MLDataSet`. There were two problems I was encountering when importing CSV files: incomplete data entries were giving undesired results when training, and an inability to preserve the column headers.

It is not uncommon to have data sets with entries that don't have values for all columns (especially when dealing with data obtained through real world observations). These values can lead to undesired results if used for training, so I wanted to discard those entries in their entirety. Thankfully, attempting to load empty data values throws a CSVError exception (Error:Unparseable number), so I was able to surround that part of the code with a try-catch statement. Inside the catch portion, I decided not to print out the stack trace because that information wasn't very useful. However, I did increment a counter I had created called `ignored`, which would then be printed to the console at the conclusion of the importing process.

For the column headers, I needed to create a new data member:

```
private static List<String> columnNames = new ArrayList<String>();
```

The information from the .CSV file is loaded into a `ReadCSV` object. If the .CSV file has headers (as specified through a boolean), these are stored in an ArrayList of Strings, which can be accessed through a `getColumnNames()` method in that class. However, there is no way to access that `ReadCSV` object after the initial importing process is completed. Thus, I needed to add some additional functionality to the the `TrainingSetUtil` class.

Inside the `loadCSVTOMemory()` method, I added a simple statement to store the headers in the data member that I had defined above:

```
if(headers){
   columnNames = csv.getColumnNames();
}
```

After that, it was just a matter of creating a standard accessor method (similar to the one in the `ReadCSV` class):

```
/**
 * @return the columnNames
 */
public static List<String> getColumnNames() {
```

```
    return columnNames;
}
```

Back in the main part of the program, I wrote another method to change the ArrayList into a standard array because I am more comfortable accessing information in that format.

Modified Code

```
if(headers){
   columnNames = csv.getColumnNames();
}

int ignored = 0;

while (csv.next()) {
   MLData input = null;
   MLData ideal = null;
   int index = 0;
   try{
      input = new BasicMLData(inputSize);
      for (int i = 0; i < inputSize; i++) {
         double d = csv.getDouble(index++);
         input.setData(i, d);
      }

      if (idealSize > 0) {
         ideal = new BasicMLData(idealSize);
         for (int i = 0; i < idealSize; i++) {
            double d = csv.getDouble(index++);
            ideal.setData(i, d);
         }
      }

      MLDataPair pair = new BasicMLDataPair(input,
            ideal);
      result.add(pair);
   }catch (CSVError e){
      ignored++;
      //e.printStackTrace();
   }
}
System.out.println(''Rows ignored: '' + ignored);
return result;
```

Original Code

```
 while (csv.next()) {
    MLData input = null;
    MLData ideal = null;
    int index = 0;

    input = new BasicMLData(inputSize);
    for (int i = 0; i < inputSize; i++) {
       double d = csv.getDouble(index++);
       input.setData(i, d);
    }

    if (idealSize > 0) {
       ideal = new BasicMLData(idealSize);
       for (int i = 0; i < idealSize; i++) {
          double d = csv.getDouble(index++);
          ideal.setData(i, d);
       }
    }

    MLDataPair pair = new BasicMLDataPair(input,
          ideal);
    result.add(pair);

}

return result;
```

Figure 16: Comparison between modified and original code

Figure 16 shows the differences between the original `loadCSVTOMemory()` method and the modified version. The complete code for this class may be found in Appendix A.6

3.4.2 BasicNetwork

`BasicNetwork` (org.encog.neural.networks.BasicNetwork) serves as the main source of interaction between my implementation and the network itself. However, this doesn't necessarily mean that this class does most of the work. Much of the information is stored in related classes (for example, once the format of the network is set up, the majority of information about the network is stored in a `FlatNetwork` object).

Before a network can be used, its structure must be defined. For this purpose, the `BasicNetwork` class uses this data member:

```
/**
 * Holds the structure of the network. This keeps the network from
    having to
 * constantly lookup layers and synapses.
 */
private final NeuralStructure structure;
```

To set up this structure, each layer must be added through the use of the `addLayer()` method. Each layer passed through the parameters will be added to an ArrayList of `Layer` objects. The first layer added will be considered the input layer.

Once all of the layers are added, the network must be finalized by invoking `structure.finalizeStructure()`. Finalizing a neural structure eliminates the intermediate representation of the layers, temporarily storing that information in `FlatLayer` objects, and then creating the `FlatNetwork` object which will be used in the remaining network operations.

Once the network is finalized, the `reset()` method is invoked, which assigns random starting values to the weights.

The actual network training is done through the `Propagation` class (an abstract class which serves as a parent for classes such as `Backpropagation` and `ResilientPropagation`). The `BasicNetwork` object is passed as a parameter, as well as the training data set and any other necessary variables (such as the learning rate and momentum if applicable).

Once the network is fully trained, its effectiveness can be measured by use of the `compute()` method. This is used to compare each ideal output value with the output value the network produces when given the same input.

3.4.3 FlatNetwork

The `FlatNetwork` class (org.encog.neural.flat.FlatNetwork) is a more computationally efficient form of a neural network, designed to store everything in single arrays instead of keeping track of everything in multiple layers. Layers are maintained through the use of index arrays, which indicate where each layer starts in the main arrays. According to the javadoc comments, "this is meant to be a very highly efficient feedforward, or simple recurrent, neural network. It uses a minimum of objects and is designed with one principal in mind-- SPEED. Readability, code reuse, object oriented programming are all secondary in consideration".

In concept, `FlatNetwork` objects act similarly from the standpoint of the user, for they share many of the same methods. However, most of the calculations (such as training) are actually done in this class (the `BasicNetwork` class invokes methods from here). The speed increase comes from the use of single-dimensional arrays of doubles and ints, which have a faster information access time than using accessor and mutator methods with multiple classes.

3.4.4 BasicLayer

The `BasicLayer` class (org.encog.neural.networks.layers.BasicLayer) is an implementation of the `Layer` interface. Its job is to store information about the specific layer it is assigned (input, hidden, or output) during the creation of the network. Once the network has been finalized, specific layers are no longer used.

The class has two constructors: one which has user defined parameters (activation function, bias, and neuron count), and one which just receives the number of neurons in the layer. If the second constructor is used, the default option is to create a layer which has a bias and uses a hyperbolic tangent activation function.

Hidden layers utilize all three variables when being initialized. Input layers do not have activation functions. Bias nodes are stored in the layer prior to where they will have an impact (a bias node which effects the nodes in the hidden layer will be declared as part of the input layer), so output layers should not have a bias.

Each layer also has a data member which indicates which network the layer is a part of.

3.4.5 BasicMLDataSet

The `BasicMLDataSet` class (org.encog.ml.data.basic.BasicMLDataSet) isn't a very complicated class, but it is very important. A child class for the more general `MLDataSet` interface, the main purpose of this class is to maintain an ArrayList of `BasicMLDataPair` objects. This is what the training data set will be stored in. The class contains of several constructors, able to create an object by accepting multidimensional double arrays, an `MLDataSet` object, or an `MLDataPair` ArrayList. The rest of the class contains several add methods, as well as methods to retrieve data entries or information about the data set (such as its size).

3.4.6 BasicMLDataPair

The `BasicMLDataPair` class (org.encog.ml.data.basic.BasicMLDataPair) is a child class of the `MLDataPair` interface. Its purpose is to hold the information of a single data entry. Each `BasicMLDataPair` contains two `MLData` objects, arrays of doubles designed to store the input data and the ideal data respectively. Both values are necessary for supervised learning, but only the input value is required for unsupervised learning (the ideal value should be left null).

3.4.7 ActivationFunction

`ActivationFunction` (org.encog.engine.network.activation.ActivationFunction) is an interface that serves as a parent class for any activation function that would be used with a neural network. The library comes with sixteen activation functions already implemented, but users are free to implement their own as long as they include all of the methods in the interface.

The two most important methods are as follows:

```
void activationFunction(double[] d, int start, int size);
```

This method is the main math portion of the activation function. The input values are stored in the double array `d`, with the range of values specified by the variables `start` and `size`. After some mathematical calculations, the output value from the activation function is stored in the same double array. For example, from the `ActivationSigmoid` class:

```
x[i] = 1.0 / (1.0 + BoundMath.exp(-1 * x[i]));
```

The `ActivationLinear` class actually leaves this method blank. The linear activation function has outputs identical to its inputs, so there is no need to do anything with the array of doubles.

```
double derivativeFunction(double b, double a);
```

This method calculates the derivative of the activation function at a certain point. Not all activation functions have derivatives (there is another method called `hasDerivative()`, which will return true if the specific activation function has a derivative and false otherwise). However, there must be a derivative for an activation function to be used with backpropagation.

The method receives two doubles as parameters. The first double, `b`, is the original input number (in the activationFunction method, this number would have been in the `d` array). The second double, `a`, is the original output value. This is the value the activation function produces if it is given `a` as an input. Depending on the equation for each specific activation function, the derivative will be calculated with whichever value is more computationally efficient. For example, the `ActivationSigmoid` class uses the output value:

```
return a * (1.0 - a);
```

To contrast, the `ActivationElliott` class uses the input value:

```
double s = params[0];
double d = (1.0+Math.abs(b*s));
return (s*1.0)/(d*d);
```

As of v3.3, all activation functions in the Encog library have derivatives with the exception of `ActivationCompetitive`. Attempting to use this activation function in a backpropagation network will throw an EncogError exception ("Can't use the competitive activation function where a derivative is required").

CHAPTER 4

Implementation

4.1 Overview

The purpose of this program is to train an artificial neural network and export source code for it. This will allow the results of the network to be used in other projects without needing to store it in a data structure.

All information is controlled through user input via a config file and a training data set. The program will output two files: a training error report, and the code for the network. The exact format of these outputs will be designated by the user.

4.2 Assumptions/Design Choices

Early in the design process, I decided that I was going to use an external third party library to handle the actual training of the neural network. The purpose of this project was more focused on the source code generation for a neural network, rather than the training itself. Doing the actual implementation for the network training would add additional development time to this project. In addition, unless it were to be made the main focus of the project, a personal implementation would not be as effective as a third party alternative, as the designers of said software have spent years optimizing the code. More information about the java library Encog may be found in the previous chapter.

The only other major design decision was the restriction of only numerical information for the training data set. The program is only designed to be used with numbers for all data entries. Using strings will result in rows being ignored when the .csv file is imported. For more information on this decision, see section 5.1.

The program also assumes that all inputs from the user are valid. As of now, there are very little debugging tools built into the program, so invalid data will result in the program not running.

4.3 Inputs

The program requires two inputs from the user: a config file containing all of the information required by the neural network, and a .csv file containing the training data set.

4.3.1 Config File

The only command line argument is the file path for a config file. This file can have any name, but it must have a .txt file extension. The config file contains the following information:

- The complete file path for the training data set. This file will be described in detail in the next subsection.
- A boolean for whether or not the training data set file has a header row (true for yes, false for no).
- The number of input variables (how many columns in the training data set are independent variables).
- The number of output variables (how many columns in the training data set are dependent variables).
- The number of hidden layers the artificial neural network will be constructed with. There is no theoretical upper limit on the number of hidden layers this program can accommodate, although studies have shown that almost any problem can be solved with the use of at most two hidden layers. [1]
- Attributes for each hidden layer:

- An integer for the type of activation function:
    0. Sigmoid
    1. Hyperbolic Tangent
    2. Linear
    3. Elliott
    4. Gaussian
    5. Logarithmic
    6. Ramp
    7. Sine
    8. Step
    9. Bipolar
    10. Bipolar Sigmoid
    11. Clipped Linear
    12. Elliott Symmetric
    13. Steepened Sigmoid
- A boolean for if the layer has a bias node or not.
- An integer for the number of normal neurons in the layer.

- Attributes for the input layer (only bias information is needed).
- Attributes for the output layer (bias and activation function is needed).
- The file type for the first output file (the training error):
    0. text (.txt)
    1. just numbers (.csv)

- The name of the first output file (not including the file extension; the program will add that information internally).
- The file type for the second output file (the code for the artificial neural network):

  0. equation format(.txt)
  1. java file (standalone)
  2. java file (integrated)

- The name of the second output file (not including the file extension; the program will add that information internally).
- The desired training error. The network will continue to train until the error is less than or equal to this number.
- The maximum number of epochs. If the desired training error has not yet been reached, the network will stop training after this many iterations.
- An integer for the network type:

  0. ResilientPropagation (see section 2.1.2)
  1. Backpropagation (see section 2.1.1)

- The learning rate. This is only applicable for backpropagation networks.
- The momentum. The program will not use momentum if this value is set to 0. This is only applicable for backpropagation networks.

Comments can be made by beginning a line with a percent symbol (%). The methods related to importing the config file will ignore any such lines.

Rather than prompting the user for this information within the program, using a file allows all of the required information to be stored in one place. This also makes multiple uses of the program easier, because the user is able to change the value of a single variable without going through the tedious process of re-inputting all of the other data as well.

4.3.2 Training Data Set

The other primary input is the training data set. As mentioned in the previous subsection, the file path for this file is given as part of the config file rather than as a command line argument.

The training data set must conform to the following specifications:

- It must be in comma-separated values format (a .csv file).
- Headers are optional. If they are included, the code that the program exports will use the column names as identifiers for variables.
- If possible, do not include any rows with blank entries in them. These rows will be discarded when the .csv file is imported, and therefore not used for training purposes.
- The .csv file shall for organized so that the independent variables (input) are on the left, while the dependent variables (output) are on the right.
- All of the data entries must be numerical. At this time the program does not support categorical classification.

Currently, the program uses the same data set for both training and testing.

4.4 Outputs

The program has two separate output files: one file containing the training error report, and one file containing the code of the neural network.

4.4.1 Training Error Report

The first output file contains information about the training error. This is the overall error (how far the actual results are from the ideal results) after each iteration of training.

The exact format of this file can be specified by the user in the config file. Currently, there are two possible formats.

If the user selects option 0, the output will be in a .txt file:

```
Epoch #1 Error:0.3341111047060686
Epoch #2 Error:0.28750265295383065
Epoch #3 Error:0.26142669721283035
```

Figure 17: Sample from first output file (.txt)

If the user selects option 1, the output will be in a .csv file. This will have a header row, and can be loaded into other programs for analysis (such as graphing):

```
Epoch,Error
1,0.3341111047060686
2,0.28750265295383065
3,0.26142669721283035
```

Figure 18: Sample from first output file (.csv)

4.4.2 Source Code

The second output file contains the source code for the trained neural network. Regardless of what file format this output is in, there will be two main sections to it: variable declaration, and network calculation.

The variable declaration section is a list of all the variables that will be used in the network calculation section, as well as any default values (such as 1.0 for biases). I decided upon the following naming conventions for variables:

- **i** - Input layer (starts at 0)

- **h** - Hidden layer (starts at 1)
- **o** - Output layer (starts at 0)
- **n** - Number of the node within a layer (starts at 0)
- **f** - Indicates which node from the previous layer the link originates (starts at 0)
- **t** - Total (the sum of all f nodes leading to a specified nodes), before being fed to the activation function.
- Lack of an f or a t indicates that this value is the output from an activation function (or a bias node).

If there are headers present in the input file, these will be included as input and output variable names.

The network calculation section is where the specific weight values are utilized in order to write equations that map the specified input values to output values. This allows the function of the trained network to be maintained without needing to store the information in a data structure.

The exact format of this file can be specified by the user in the config file. Currently, there are two possible formats.

If the user selects option 0, the output will be in a .txt file. Variable declarations will just consist of names, and the network calculation section will just be mathematical formulas.

If the user selects option 1 or 2, the output will be in a .java file. Variables will all be declared as doubles (original input and final output variables will be public, and all others will be private). The network calculation section will be inside a method. Everything will also be inside of a class (which shares the same name as the file itself, as specified by the user in the config file).

4.5 Individual classes

The program itself (not counting the modified Encog library) currently consists of five classes. This number may grow in the future if more output source code types were to be implemented.

4.5.1 NeuralGenerator

The `NeuralGenerator` class is the largest class in the program. Most of the work happens here.

The variable declarations are mostly self explanatory, so they will not be discussed here. The comments for each variable can be viewed in Appendix A.1.

After the initial setup, the first thing the program does is import data from the config file, through the `validateConfig()` method. This method goes through the config file line by line (through the use of a helper method, `nextValidLine()`, which ignores any lines that are commented out, as designated by a '%' at the beginning of a line). All information from the config file is stored into data members so it can be accessed by other methods, and is then printed out to the console.

The `initializeOutput1()` method is called, which creates the first output file. This file will contain the training error report. For more information, see section 4.4.1.

The next method to be invoked is `createNetwork()`. This method creates a `BasicNetwork`, and populates it with an input layer, hidden layers, and an output layer. The information for each layer (activation function, bias, and number of nodes) is specified by `LayerInfo` objects, which in turn are defined by the information in the config file. Once all of the layers are added, the network is finalized, and the weights are reset to random values.

Next, the training data set is created from the .csv file. If there are headers, these are stored in an ArrayList (the information is then stored in a String array, because I prefer working with that format).

Then, the network is trained. The two current options for training utilize either the `Backpropagation` class or the `ResilientPropagation` class (for more information, see sections 2.1.1 and 2.1.2 respectively). After each iteration of training, the training error is calculated, and written to a file through the `writeOne()` method. This helper method also prints the information to the console. Training will continue until the training error is below the desired amount, or until the maximum number of epochs has been reached.

Once the network is trained, the first file is closed. The program prints the results of the network to the console, comparing the actual value of each input to its ideal value.

Finally, the `initializeOuput2()` method is invoked. This method creates the code output file (see section 4.4.2), and stores the necessary values in variables through accessor methods in the `OutputWriter` class. Finally, the program flow then proceeds to the `writeFile()` method for the desired `OutputWriter` child class, and then the program terminates.

4.5.2 LayerInfo

`LayerInfo` is a small class created to store the information needed to create a layer in an artificial neural network. I had originally planned on using a struct, but java does not support those, so I decided to make a separate class to hold the same information.

The class has 3 main variables:

- `private int activationFunction` - An integer for the type of activation function for that layer.

- `private boolean isBiased` - A boolean for whether or not the layer has a bias node.

- `private int neurons` - An integer for the number of normal (non-bias) neurons in the layer.

All of these variables are set through parameters passed to the constructor. There should not be a need to change these values once they have been set, so there are no mutator methods. Each variable has an accessor method so that its value can be used by the rest of the program.

The only other method is the `toString()` method. This method is used for returning the information from the layer in an easy-to-read format, so that it can be printed. While not essential to the flow of the program, it may be useful for the user to see this information displayed in the console (especially for debugging purposes).

4.5.3 OutputWriter

The `OutputWriter` class serves as a parent class for other OutputWriters. This class holds all of the shared methods required to create a file and output the code/formula for a trained artificial neural network.

Child classes must implement three methods: `createFile()`, `writeFile()`, and `parseActivationFunction()`.

The `createFile()` method creates the file used for output. While the majority of the code in this method is the same in all child classes, I found that it was easier to have each child class add its own file extension to the file name (.txt or .java).

The `writeFile()` method is rather lengthy. This is where the actual program writes the code/formula for the neural network to a file. While similar in terms of basic structure, the actual details of this will vary with each child class.

The `parseActivationFunction()` parses the equation of the activation function and returns it in String form. A series of `if-else` statements allowed for 14 of the 16 currently implemented activation functions to be used (Softmax is rather complicated and would require the code to be reworked, and Competitive is non-differentiable so I didn't see a need to include it).

4.5.4 OutputWriterTxt

The `OutputWriterTxt` class is a child of the `OutputWriter` class. This class will be used if the user selects option 0 for the second output file.

The `createFile()` method creates the second output file, using the filename as specified in the config file and appending a .txt file extension.

The variable declarations section gives the names of all the variables to be used in the network calculation section, in the following order:

- Header names for input (if applicable).
- Input layer.
- Hidden layer(s).
- Output layer.
- Header names for output (if applicable).

If there any are bias nodes present, they are assigned the value of the bias as defined in the network (the default is 1.0, but this value is customizable).

Following the variable declaration is the network calculation section. If there are any variables defined by header names, the values of these are stored into the predefined variables such as `i0`. After that, calculation begins starting with the first hidden layer. The `f` values are calculated first, consisting of the associated node in the previous layer multiplied by the weight of the connection (as defined by the trained network). All `f` values for a specific node are added together, with the resulting sum being stored in a `t` value. This `t` value is then fed through the activation function (the text of which comes from the `parseActivationFunction()` method), and that is stored in the final value for that node. This continues through all of the hidden layers and the output layer. Finally, if applicable, the values of the final outputs are stored in the variables defined by output header names.

The `parseActivationFunction()` method parses the equation of the activation function and returns it in String form. All information with regards to the exact mathematical formulas for each activation function came from the `activationFunction()` method of the associated class.

```
//Variable declarations
//Header Names
x1
x2
xor
//Input layer
i0
i1
i2 = 1.0
//Hidden layer(s)
//Hidden Layer 1
h1n0f0
h1n0f1
h1n0f2
h1n0t
h1n0
h1n1f0
h1n1f1
h1n1f2
h1n1t
h1n1
h1n2 = 1.0
//Output layer
o0f0
o0f1
o0f2
o0t
o0
```

(a)

```
i0 = x1
i1 = x2

h1n0f0 = i0 * 1.367776537359666
h1n0f1 = i1 * 1.2920909769370108
h1n0f2 = i2 * -2.3097219906621054
h1n0t = h1n0f0 + h1n0f1 + h1n0f2
h1n0 = 1.0 / (1.0 + e^(-1 * h1n0t)

h1n1f0 = i0 * 4.975164727376538
h1n1f1 = i1 * 4.634590911948785
h1n1f2 = i2 * -1.1798243603241103
h1n1t = h1n1f0 + h1n1f1 + h1n1f2
h1n1 = 1.0 / (1.0 + e^(-1 * h1n1t)

o0f0 = h1n0 * -15.158855977843674
o0f1 = h1n1 * 10.890253618352084
o0f2 = h1n2 * -3.360679967779563
o0t = o0f0 + o0f1 + o0f2
o0 = 1.0 / (1.0 + e^(-1 * o0t)

xor = o0
```

(b)

Figure 19: Sample second output file (.txt)

4.5.5 OutputWriterJava

The `OutputWriterJava` class is a child of the `OutputWriter` class. This class will be used if the user selects option 1 or 2 for the second output file.

The `createFile()` method creates the second output file, using the filename as specified in the config file and appending a .java file extension.

The format of the .java file was inspired by the output of the program Tiberius. One of the original intents of this program was to be used in a course that currently uses Tiberius, so it made sense to model the output file in a way that it would be compatible.

The first things written to the file is an import statement for `java.lang.Math`, followed by the declaration of the class (with the same name as the file).

The variable declarations section declares all of the variables to be used in the network calculation section, in the same order as specified in the previous subsection. All methods are static so that they can be accessed from the main method without creating a specific object of this class type, so all variables are declared as static doubles. Most variables are private, but the input and output variables (as well as any variable names defined by headers in the training data set) are declared as public so that they can be accessed by the user in other classes. If there any are bias nodes present, they are assigned the value of the bias as defined in the network (the default is 1.0, but this value is customizable). Bias nodes are also always declared as private, even if they are in the input layer.

If the user has chosen to make a standalone java file, there will be two additional methods: `main()` and `initData()`. The `main()` method will call the other two methods (`initData()` and `calcNet()`), and then print the output values to the console. The `initData()` method will provide default values for the input variables (using the header names if applicable). The default values are currently set to 1, although these can be modified by the user. If the user has selected to make an integrated java file, neither of these two methods will be present.

The `calcNet()` method contains the network calculation section of the code. The code generation of this section is almost identical to the equivalent in the `OutputWriterTxt` class, except that every line ends with a semicolon.

The `parseActivationFunction()` method parses the equation of the activation function and returns it in String form. Some mathematical expressions are substituted with their java equivalents (for example, $|x|$ becomes `Math.abs(x)`, and e^x becomes `Math.exp(x)`).

```java
import java.lang.Math;

public class output2
{
    //Variable declarations
    //Header Names
    public static double x1;
    public static double x2;
    public static double xor;
    //Input layer
    public static double i0;
    public static double i1;
    private static double i2 = 1.0;
    //Hidden layer(s)
    //Hidden Layer 1
    private static double h1n0f0;
    private static double h1n0f1;
    private static double h1n0f2;
    private static double h1n0t;
    private static double h1n0;
    private static double h1n1f0;
    private static double h1n1f1;
    private static double h1n1f2;
    private static double h1n1t;
    private static double h1n1;
    private static double h1n2 = 1.0;
    //Output layer
    private static double o0f0;
    private static double o0f1;
    private static double o0f2;
    private static double o0t;
    public static double o0;

    public static void main(String[] args)
    {
        initData();
        calcNet();
        System.out.println(o0);
    }
```

(a)

```java
    public static void initData()
    {
        //data is set here
        x1 = 1;
        x2 = 1;
    }

    public static void calcNet()
    {
        i0 = x1;
        i1 = x2;

        h1n0f0 = i0 * -918.089072207438;
        h1n0f1 = i1 * 3.998008385237878;
        h1n0f2 = i2 * -8.631004060244248;
        h1n0t = h1n0f0 + h1n0f1 + h1n0f2;
        h1n0 = 1.0 / (1.0 + Math.exp(-1 * h1n0t));

        h1n1f0 = i0 * 3.0491035435976457;
        h1n1f1 = i1 * -297.45021956037084;
        h1n1f2 = i2 * -6.180183917718583;
        h1n1t = h1n1f0 + h1n1f1 + h1n1f2;
        h1n1 = 1.0 / (1.0 + Math.exp(-1 * h1n1t));

        o0f0 = h1n0 * 507.0365297905731;
        o0f1 = h1n1 * 132.9962652623966;
        o0f2 = h1n2 * -2.1795899783064443;
        o0t = o0f0 + o0f1 + o0f2;
        o0 = 1.0 / (1.0 + Math.exp(-1 * o0t));

        xor = o0;
    }
}
```

(b)

Figure 20: Sample second output file (.java)

List of References

[1] J. Heaton. "The number of hidden layers." September 2008. [Online]. Available: http://www.heatonresearch.com/node/707

CHAPTER 5

Future work and conclusions

5.1 Categorical classification

As of right now, the program only works with data sets with entirely numerical entries. This means that any data sets with categorical entries will need to be changed into a numerical representation before they can be used. For example, with the iris data set, instead of species of setosa, versicolor, and virginica, it would use numbers such as 1, 2, and 3.

My original concept was to start out with numerical classification first, because it is easier to work with, and then expand to include categorical if I had time. However, I discovered late in the implementation period that in order to be able to use categorical data sets, I would have to completely change how the network itself was implemented. Within Encog, categorical classification is done with several different classes than numerical classification.

As of the writing of this paper, I do not know if I can use those classes to work with numerical data sets, or if I would have to make a separate main class for the different types.

5.2 Additional output formats

Originally, this project was going to be written in C++, because of the applicability of that language to the gaming industry (where I want to work)[1]. However, it was changed to Java because of the portability of that language (there is no need to compile the code for different systems, because it is always run within the java virtual machine).

Currently, the second output file from the program can be in either basic equational format (in a .txt file), or in Java code. Given more time, I would have preferred to also allow for C++ code to be exported. Given the nature of the program, it would not be unfeasible to allow other target languages to be implemented as well.

5.3 Non-command line inputs

Currently, the only input into the program is through a config file, which contains all of the necessary data that the program needs to run. While this can be easier for multiple runs (because the user does not need to repeatedly input the information each time), I recognize that it can be hard to set up the config file for the first time. Some users may also prefer to enter the information on a step by step basis.

The basic implementation of an alternate input method is not very complicated. If there are no arguments passed to the program when it is run, a new method would be called instead of the `validateConfig()` method. This method would assign values for all of the necessary variables through a series of input statements utilizing a scanner.

Related to this additional input method would be improved config file debugging. Currently the program assumes that all of the data the user has inputted is valid. There is no checking in that method to see if a number is within a valid range (for example, a value of 17 for the activation function type). These numbers are checked in other places of the code, but it would be more useful for the user to have the numbers validated the first time they are encountered. If there is a major problem (for example, the program is expecting a number, and the user has a string of text instead), a generic exception will be thrown, and the stack trace will be printed to the console.

Although these implementations are not very difficult, they were omitted for because of timing.

5.4 Normalization

Depending on the data set being used, normalization can be an important feature in machine learning. For example, data sets with values for a certain variable that are much larger than other values may not converge as well during training as with a normalized data set. Encog supports several different types of normalization, but I would most likely be using range normalization due to the ability to normalize the data to a specific range (for example, -1 to 1 when using a hyperbolic tangent activation function, or 0 to 1 when using a sigmoid activation function).

$f(x) = \frac{(x-d_L)(n_H-n_L)}{(d_H-d_L)} + n_L$ [2]

In this equation, d_L is the minimum value (low) in the data set, d_H is the maximum value (high) in the data set, n_L is the desired normalized minimum, and n_H is the desired normalized maximum.

5.5 Conclusions

Overall, the program works. The best way to illustrate this is to walk through an example.

In this example, the config file shown in figure 27 was used. This creates a network with a single hidden row of 2 neurons, a sigmoid function for an activation function, and RPROP for a training algorithm. The XOR function was used as a simple data set, as seen in figure 21.

x1	x2	xor
0	0	0
1	0	1
0	1	1
1	1	0

Figure 21: test.csv

Due to the speed of RPROP, this network was able to train to an error of 0.01 within 47 iterations. A .csv format was chosen for the first output file, which can be seen in figure 22.

Epoch	Error				
1	0.27828625	16	0.24331544	32	0.10324324
2	0.26261123	17	0.24145497	33	0.10371875
3	0.25154451	18	0.23910242	34	0.09008032
4	0.25082551	19	0.23688016	35	0.08526512
5	0.2514136	20	0.23280682	36	0.07503029
6	0.24935344	21	0.2280602	37	0.06615781
7	0.25124387	22	0.22064924	38	0.06183121
8	0.24911844	23	0.21124914	39	0.05247069
9	0.24879488	24	0.19860283	40	0.04412141
10	0.24863693	25	0.18573386	41	0.03573549
11	0.24818165	26	0.16598462	42	0.0310084
12	0.24760513	27	0.14295186	43	0.0238079
13	0.24695005	28	0.1266399	44	0.01915438
14	0.24600628	29	0.15851361	45	0.01462404
15	0.24481026	30	0.12725001	46	0.01027347
		31	0.11714096	47	0.00673701

(a) (b) (c)

Figure 22: output1.csv

CSV files can easily be plotted, as seen in figure 23. The spike at epoch 29 most likely illustrates the nature of the algorithm to correct itself after skipping past a local minimum, as denoted by a sign change in the gradient.

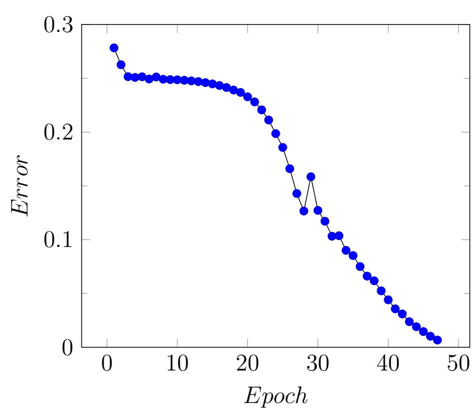

Figure 23: graph of output1.csv

The second output is in a java file, as seen in figure 28. The integrated option was chosen, so there will not be any main method.

The program also printed the results of testing the network, as seen in figure 24.

```
Neural Network Results:
0.0,0.0, actual=0.05226631825448506,ideal=0.0
1.0,0.0, actual=0.9465544658147504,ideal=1.0
0.0,1.0, actual=0.9092496344940679,ideal=1.0
1.0,1.0, actual=0.05052107883471591,ideal=0.0
```

Figure 24: Results from NeuralGenerator.java

In order to test to ensure that the program outputted source code correctly, I wrote the following short program, seen in figure 25. This program tests the code for the network by using all four values of the original training data set, and outputting the results.

```
package test;

public class TestModule {

  private static int[][] tests =
        {{0,0},{1,0},{0,1},{1,1}};

  public static void main(String[] args) {

    for (int i = 0; i < 4; i++){
      System.out.print("\nTest #" + i + ": (" +
            tests[i][0]
          + "," + tests[i][1] + ") \t\t Results: ");
      output2.x1 = tests[i][0];
      output2.x2 = tests[i][1];
      output2.calcNet();
      System.out.print(output2.xor);
    }
  }
}
```

Figure 25: TestModule.java

When this program is run, it produces the output seen in figure 26.

```
Test #0: (0,0)          Results: 0.05226631825448506
Test #1: (1,0)          Results: 0.9465544658147504
Test #2: (0,1)          Results: 0.9092496344940679
Test #3: (1,1)          Results: 0.05052107883471591
```

Figure 26: Results from output2.java

By comparing the results from figures 24 and 26, it is evident that the generated source code produces the same results as the trained network. Therefor, the network has been successfully maintained without the use of additional data structures.

There are always improvements that can be made (for example, those listed in sections 5.1-5.4). However, considering the initial scope of the project, the program that has been created can be considered successful.

```
% Example Config file

% complete file address for dataset, in .csv format
/Users/bwinrich/Documents/workspace/Thesis/test.csv

% Does the cvs file have headers?
true

% Number of input parameters
2

% Number of output parameters
1

% Number of hidden layers
1

% Attributes of layers
% Use the following format: (activationFuction, isBiased, neurons),
% Each layer should be on its own line
% activationFuction is an int, with the following activation functions available:
%% 0 - Sigmoid
%% 1 - Hyperbolic Tangent
%% 2 - Linear
%% 3 - Elliott
%% 4 - Gaussian
%% 5 - Logarithmic
%% 6 - Ramp
%% 7 - Sine
%% 8 - Step
%% 9 - BiPolar
%% 10 - Bipolar Sigmoid
%% 11 - Clipped Linear
%% 12 - Elliott Symmetric
%% 13 - Steepened Sigmoid
% isBiased is a boolean (true if the layer has a bias, false otherwise)
% neurons is an int (the number of neurons in that layer)
(0, true, 2)


% Finally, the attributes of the first and last layer
% The input layer doesn't have an activation function,
% so we only need to know if it is biased
(true)
```

(a)

```
% The output layer has an activation function,
% but we don't need to know how many neurons it has,
% Or if it has a bias
(0)

% Output 1 file type
%% 0 = text (txt)
%% 1 = just numbers (csv)
1

% Output 1 file name (do not include file extension at the end)
output1

% Output 2 file type
%% 0 = equation format (.txt file)
%% 1 = java file (standalone)
%% 2 = java file (integrated)
2

% Output 2 file name (do not include file extension at the end)
output2

% Desired training error (the network will train until
% the error is less than or equal to this)
0.01

% Maximum number of epochs
10000

% Network Type
%% 0 = ResilientPropagation
%% 1 = Backpropagation
0

% Learning Rate
% (only applicable for Backpropagation)
0.7

% Momentum (if you do not want to use momentum, set this to 0)
% (only applicable for Backpropagation)
0
```

(b)

Figure 27: Sample config file

List of References

[1] K. Stuart. "How to get into the games industry – an insiders' guide." March 2014. [Online]. Available: http://www.theguardian.com/technology/2014/mar/20/how-to-get-into-the-games-industry-an-insiders-guide

[2] J. Heaton. "Range normalization." September 2011. [Online]. Available: http://www.heatonresearch.com/wiki/Range_Normalization

```java
import java.lang.Math;

public class output2
{
    //Variable declarations
    //Header Names
    public static double x1;
    public static double x2;
    public static double xor;
    //Input layer
    public static double i0;
    public static double i1;
    private static double i2 = 1.0;
    //Hidden layer(s)
    //Hidden Layer 1
    private static double h1n0f0;
    private static double h1n0f1;
    private static double h1n0f2;
    private static double h1n0t;
    private static double h1n0;
    private static double h1n1f0;
    private static double h1n1f1;
    private static double h1n1f2;
    private static double h1n1t;
    private static double h1n1;
    private static double h1n2 = 1.0;
    //Output layer
    private static double o0f0;
    private static double o0f1;
    private static double o0f2;
    private static double o0t;
    public static double o0;
```

(a)

```java
    public static void calcNet()
    {
        i0 = x1;
        i1 = x2;

        h1n0f0 = i0 * 3.6869200533700797;
        h1n0f1 = i1 * -3.3847620146833983;
        h1n0f2 = i2 * -2.0233813712749544;
        h1n0t = h1n0f0 + h1n0f1 + h1n0f2;
        h1n0 = 1.0 / (1.0 + Math.exp(-1 * h1n0t));

        h1n1f0 = i0 * 8.161271474645059;
        h1n1f1 = i1 * -7.657582392497621;
        h1n1f2 = i2 * 1.8305055444152039;
        h1n1t = h1n1f0 + h1n1f1 + h1n1f2;
        h1n1 = 1.0 / (1.0 + Math.exp(-1 * h1n1t));

        o0f0 = h1n0 * 9.36214064231166;
        o0f1 = h1n1 * -7.281180690756561;
        o0f2 = h1n2 * 2.284144577336713;
        o0t = o0f0 + o0f1 + o0f2;
        o0 = 1.0 / (1.0 + Math.exp(-1 * o0t));

        xor = o0;
    }
}
```

(b)

Figure 28: output2.java

APPENDIX

Source Code

A.1 NeuralGenerator.java

```java
package skynet;
import org.encog.Encog;
import org.encog.engine.network.activation.*;
import org.encog.ml.data.MLData;
import org.encog.ml.data.MLDataPair;
import org.encog.ml.data.MLDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.propagation.Propagation;
import org.encog.neural.networks.training.propagation.back.Backpropagation;
import org.encog.neural.networks.training.propagation.resilient.*;
import org.encog.util.csv.CSVFormat;
import org.encog.util.simple.TrainingSetUtil;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

/**
 * The main class for the program. The purpose of this program is the train
 * an Artificial Neural Network and output source code for it, so that it
 * can be used in other projects.
 * @author bwinrich
 *
 */
public class NeuralGenerator {

   /**
    * A BufferedWriter for our error output
    */
   private BufferedWriter bw1 = null;

   /**
    * An OutputWriter for our code output
    */
   private OutputWriter myOutput;
```

```java

    /**
     * The number of neurons in each layer (including bias neurons)
     */
    private int[] numberOfTotalNeurons;

    /**
     * The number of neurons in each layer (excluding bias neurons)
     */
    private int[] numberOfNormalNeurons;

    /**
     * The location of the .csv file for the training data set
     */
    private String filePath = null;

    /**
     * Does the data set have headers?
     */
    private boolean hasHeaders = false;

    /**
     * The number of input nodes
     */
    private int numOfInput = 0;

    /**
     * The number of output nodes
     */
    private int numOfOutput = 0;

    /**
     * The number of hidden layers in the network
     */
    private int numOfHiddenLayers = 0;

    /**
     * An array holding the information for each layer (activation function,
     * bias, number of neurons)
     */
    private LayerInfo[] allMyLayers = null;

    /**
     * The network will train until the error is this value or lower
     */
    private double desiredTrainingError = 0.01;
```

```java

/**
 * The maximum number of epochs the network will train
 */
private int numOfEpochs = 0;

/**
 * The learning rate for the network (backpropagation only)
 */
private double learningRate = 0;

/**
 * The momentum for the network (backpropagation only)
 */
private double momentum = 0;

/**
 * The type of file the error output will be (0: .txt, 1: .csv)
 */
private int output1Type = 0;

/**
 * The type of file the code output will be (0: .txt, 1/2: .java)
 */
private int output2Type = 0;

/**
 * The name of the error output file
 */
private String output1Name = null;

/**
 * The name of the code output file
 */
private String output2Name = null;

/**
 * The data structure for the ANN
 */
private BasicNetwork network = null;

/**
 * Array to hold the column names (only if the data set has headers)
 */
private String[] columnNames;

```

```java
/**
 * The type of network the ANN will be (0: Resilient propagation,
 * 1: backpropagation)
 */
private int networkType;

/**
 * The training data set
 */
private MLDataSet trainingSet;


/**
 * The main method.
 * @param args Should contain the location of the config file
 */
@SuppressWarnings("unused")
public static void main(final String args[]) {
   if (args.length == 0){
      System.out.println("Error: No file");
   }else{
      String configFilePath = args[0];
      NeuralGenerator myThesis = new NeuralGenerator(configFilePath);
   }
}

/**
 * Constructor for the class.
 * @param configFilepath The location of the config file
 */
public NeuralGenerator(String configFilepath){
   newMain(configFilepath);
}

/**
 * The driver method, which will call all other necessary methods
 * required for the execution of the program
 * @param configFilePath The location of the config file
 */
private void newMain(String configFilePath){

   //Import the config file, and the necessary information
   validateConfig(configFilePath);

   //Create the first output file
   System.out.println("Initializing first output file...");
```

```java
        initializeOutput1();

        // create a neural network
        System.out.println("Creating network...");
        createNetwork();

        //Import data set
        System.out.println("Importing csv file...");

        trainingSet = TrainingSetUtil.loadCSVTOMemory(CSVFormat.ENGLISH,
             filePath, hasHeaders, numOfInput, numOfOutput);

        //Just because I prefer working with arrays instead of arrayLists
        if(hasHeaders){
           List<String> columns = TrainingSetUtil.getColumnNames();
           int width = columns.size();
           columnNames = new String[width];
           for (int i = 0; i < width; i++ ){
              columnNames[i] = columns.get(i);
           }
        }

        // train the neural network
        train();

        // Close the first file after we're done with it
        try{
           if(bw1!=null)
              bw1.close();
        }catch(Exception ex){
           System.out.println("Error in closing the BufferedWriter"+ex);
        }

        // test the neural network
        System.out.println("");
        System.out.println("Neural Network Results:");
        for(MLDataPair pair: trainingSet ) {
           final MLData output = network.compute(pair.getInput());

           System.out.println(pair.getInput().getData(0) + ","
                 + pair.getInput().getData(1) + ", actual=" + output.getData(0)
                 + ",ideal=" + pair.getIdeal().getData(0));
        }

        //Some additional numbers that we need
        int layers = network.getLayerCount();
```

```
    numberOfTotalNeurons = new int [layers];
    numberOfNormalNeurons = new int [layers];

    for (int i = 0; i<layers; i++)
    {
       numberOfTotalNeurons[i] = network.getLayerTotalNeuronCount(i);
       numberOfNormalNeurons[i] = network.getLayerNeuronCount(i);
    }

    System.out.println("\n");

    //Initialize the OutputWriter
    System.out.println("Initializing Second Output File...");
    initializeOutput2();

    System.out.println("Writing to file...");

    myOutput.writeFile();

    System.out.println("Done.");

    Encog.getInstance().shutdown();
 }


 /**
  * This method handles the training of the Artificial Neural Network
  */
 private void train() {
    Propagation train = null;

    //Different networks will be created based on the type listed in the
    //config file
    switch(networkType){
    case 0:
       train = new ResilientPropagation(network, trainingSet);
       break;
    case 1:
       train = new Backpropagation(network, trainingSet, learningRate,
            momentum);
       break;
    default:
       break;
    }
```

```
        int epoch = 1;

        System.out.println("");

        System.out.println("Training...");

        System.out.println("");

        //Training the network
        do {
            train.iteration();

            //We write the error to the first output file
            writeOne(epoch, train.getError());

            epoch++;
        } while((train.getError() > desiredTrainingError)
                && (epoch < numOfEpochs));
        //Training will continue until the error is not above the desired
        //error, or until the maximum number of epochs has been reached
        train.finishTraining();
    }

    /**
     * Helped method for creating the ANN and adding layers to it
     */
    private void createNetwork() {
        network = new BasicNetwork();

        for (LayerInfo myLayer:allMyLayers)
        {
            switch (myLayer.getActivationFunction()){
            case -1: //The input layer doesn't have an activation function
                network.addLayer(new BasicLayer(null, myLayer.isBiased(),
                        myLayer.getNeurons()));
                break;
            case 0:
                network.addLayer(new BasicLayer(new ActivationSigmoid(),
                        myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 1:
                network.addLayer(new BasicLayer(new ActivationTANH(),
                        myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 2:
                network.addLayer(new BasicLayer(new ActivationLinear(),
```

```
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 3:
                network.addLayer(new BasicLayer(new ActivationElliott(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 4:
                network.addLayer(new BasicLayer(new ActivationGaussian(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 5:
                network.addLayer(new BasicLayer(new ActivationLOG(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 6:
                network.addLayer(new BasicLayer(new ActivationRamp(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 7:
                network.addLayer(new BasicLayer(new ActivationSIN(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 8:
                network.addLayer(new BasicLayer(new ActivationStep(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 9:
                network.addLayer(new BasicLayer(new ActivationBiPolar(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 10:
                network.addLayer(new BasicLayer(
                    new ActivationBipolarSteepenedSigmoid(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 11:
                network.addLayer(new BasicLayer(new ActivationClippedLinear(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 12:
                network.addLayer(new BasicLayer(new ActivationElliottSymmetric(),
                    myLayer.isBiased(), myLayer.getNeurons()));
                break;
            case 13:
                network.addLayer(new BasicLayer(new ActivationSteepenedSigmoid(),
                    myLayer.isBiased(), myLayer.getNeurons()));
```

```java
                break;
            default:
                //Unimplemented activation function: Softmax (complicated)
                //Unimplemented activation function: Competitive
                //(non-differentiable)
                System.out.println("Error: This activation function is "
                        + "either invalid or not yet implemented");
                break;
            }
        }

        network.getStructure().finalizeStructure();

        network.reset();
    }


    /**
     * This method creates the error output file
     */
    private void initializeOutput1() {
        String output1NameFull = null;

        //File type is specified in the config file
        switch(output1Type){
        case 0:
            output1NameFull = output1Name + ".txt";
            break;
        case 1:
            output1NameFull = output1Name + ".csv";
            break;
        default:
            //More cases can be added at a later point in time
            System.out.println("Invalid output 1 type");
        }

        try{
            File file1 = new File(output1NameFull);

            if (!file1.exists()) {
                file1.createNewFile();
            }

            FileWriter fw1 = new FileWriter(file1);
            bw1 = new BufferedWriter(fw1);

```

```java
            //Header line for a .csv file
            if (output1Type == 1){
               bw1.write("Epoch,Error");
               bw1.newLine();
            }
         }catch (IOException e){
            // TODO Auto-generated catch block
            e.printStackTrace();
         }
      }

      /**
       * This method creates the code output file
       */
      private void initializeOutput2(){

         //File type is specified in the config file
         switch(output2Type){
         case 0:
            myOutput = new OutputWriterTxt();
            break;
         case 1:
            myOutput = new OutputWriterJava(true);
            break;
         case 2:
            myOutput = new OutputWriterJava(false);
            break;
         default:
            //More cases can be added if additional classes are designed
            System.out.println("Invalid output 2 type");
            break;
         }

         //Creating the file
         myOutput.createFile(output2Name);

         //Passing all of the necessary network information to the OutputWriter
         myOutput.setNetwork(network);
         myOutput.setInputCount(numOfInput);
         myOutput.setOutputCount(numOfOutput);
         myOutput.setLayers(numOfHiddenLayers+2);
         myOutput.setNumberOfTotalNeurons(numberOfTotalNeurons);
         myOutput.setNumberOfNormalNeurons(numberOfNormalNeurons);
         myOutput.setHasHeaders(hasHeaders);
         myOutput.setColumnNames(columnNames);
         myOutput.initializeOtherVariables();
```

```java
    }

    /**
     * Helper method for writing to the error output file
     * @param epoch Number of times the network has been trained
     * @param error Training error for that epoch
     */
    private void writeOne(int epoch, double error) {

        String temp = null;

        //Format depends on file type
        switch(output1Type){
        case 0:
            temp = "Epoch #" + epoch + " Error:" + error;
            break;
        case 1:
            temp = "" + epoch + "," + error;
            break;
        default:
            temp = "Invalid output 2 type";
            break;
        }

        //Output the error to the console before writing it to the file
        System.out.println(temp);
        try {
            bw1.write(temp);
            bw1.newLine();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    /**
     * Helper method for retrieving lines from the config file. Comments are
     * not considered valid lines.
     * @param d The BufferedReader for the config file
     * @return The next valid line from the config file
     * @throws IOException
     */
    private String nextValidLine(BufferedReader d) throws IOException
    {
        String validLine = null;
        boolean isValid = false;
```

```java

        if (d.ready()){
            do{
                String str = d.readLine();

                if (str.length() != 0){
                    //Eliminate extra space
                    str = str.trim();

                    //Comments start with %, and are not considered valid
                    if (str.charAt(0) != '%'){

                        validLine = str;
                        isValid = true;
                    }
                }
            }while (!isValid && d.ready());
        }
        return validLine;
    }

    /**
     * A lengthy method for validating the config file. All information from
     * the config file is stored into data members so it can be accessed by
     * other methods.
     * @param configFilepath The location of the config file
     */
    public void validateConfig(String configFilepath)
    {
        try{
            File myFile = new File(configFilepath);
            FileInputStream fis = null;

            BufferedReader d = null;

            fis = new FileInputStream(myFile);

            d = new BufferedReader(new InputStreamReader(fis));

            //First, we store the file path of the .csv file
            if (d.ready()){
                filePath = nextValidLine(d);
            }

            //Next, we store if the csv file has headers or not
            if (d.ready()){
```

```java
            hasHeaders = Boolean.parseBoolean(nextValidLine(d));
         }

         //Next, we store the number of input parameters
         if (d.ready()){
            numOfInput = Integer.valueOf(nextValidLine(d));
         }

         //Next, we store the number of output parameters
         if (d.ready()){
            numOfOutput = Integer.valueOf(nextValidLine(d));
         }

         //Next, we store the number of hidden layers
         if (d.ready()){
            numOfHiddenLayers = Integer.valueOf(nextValidLine(d));
         }

         //Next, we store the information for our hidden layers
         allMyLayers = new LayerInfo[numOfHiddenLayers+2];

         String layer = null;
         int activationFunction;
         boolean isBiased;
         int neurons;

         for (int i = 1; i < numOfHiddenLayers+1; i++){
            if (d.ready()){
               layer = nextValidLine(d);
               layer = layer.trim().toLowerCase();
               layer = layer.substring(1,layer.length()-1);
               String[] layers = layer.split(",");

               for (String l:layers){
                  l = l.trim();
               }

               activationFunction = Integer.valueOf(layers[0].trim());
               isBiased = Boolean.parseBoolean(layers[1].trim());
               neurons = Integer.valueOf(layers[2].trim());

               allMyLayers[i] =
                     new LayerInfo(activationFunction, isBiased, neurons);
            }
         }

```

```
    //Next, we store the information for the input layer
    if (d.ready()){
      layer = nextValidLine(d);
      layer = layer.trim().toLowerCase();
      layer = layer.substring(1,layer.length()-1);

      isBiased = Boolean.parseBoolean(layer.trim());

      allMyLayers[0] = new LayerInfo(-1, isBiased, numOfInput);
    }

    //Finally, we store the information for the output layer
    if (d.ready()){
      layer = nextValidLine(d);
      layer = layer.trim().toLowerCase();
      layer = layer.substring(1,layer.length()-1);

      String[] layers = layer.split(",");

      activationFunction = Integer.valueOf(layers[0].trim());

      allMyLayers[numOfHiddenLayers+1] =
          new LayerInfo(activationFunction, false, numOfOutput);
    }

    //store the information about the output 1 file type
    if (d.ready()){
      output1Type = Integer.valueOf(nextValidLine(d));
    }

    //store the information about the output 1 name
    if (d.ready()){
      output1Name = nextValidLine(d);
    }

    //store the information about the output 2 file type
    if (d.ready()){
      output2Type = Integer.valueOf(nextValidLine(d));
    }

    //store the information about the output 2 name
    if (d.ready()){
      output2Name = nextValidLine(d);
    }

    //Store the information for the desired training error
```

```java
if (d.ready()){
   desiredTrainingError = Double.valueOf(nextValidLine(d));
}

//Store the information for the maximum number of epochs
if (d.ready()){
   numOfEpochs = Integer.valueOf(nextValidLine(d));
}

//Store the information for the desired network type
if (d.ready()){
   networkType = Integer.valueOf(nextValidLine(d));
}

//We need additional variables if we are using Backpropagation
if (networkType == 1){
   //Store the information for the learning rate
   if (d.ready()){
      learningRate = Double.valueOf(nextValidLine(d));
   }

   //Store the information for the momentum
   if (d.ready()){
      momentum = Double.valueOf(nextValidLine(d));
   }
}

//TODO: reorder this
//output the information from the config file
System.out.println("config file validated:");
System.out.println("\tfilePath = " + filePath);
System.out.println("\thasHeaders = " + hasHeaders);
System.out.println("\tnumOfInput = " + numOfInput);
System.out.println("\tnumOfOutput = " + numOfOutput);
System.out.println("\tnumOfHiddenLayers = " + numOfHiddenLayers);
for (LayerInfo l: allMyLayers){
   System.out.println("\t" + l.toString());
}
System.out.println("\tdesiredTrainingError = "
      + desiredTrainingError);
System.out.println("\tnumOfEpochs = " + numOfEpochs);
System.out.println("\tnetworkType = " + networkType);
if (networkType == 1){
   System.out.println("\tlearningRate = " + learningRate);
   System.out.println("\tmomentum = " + momentum);
}
```

```
            System.out.println("\toutput2Type = " + output1Type);
            System.out.println("\toutput2Name = " + output1Name);
            System.out.println("\toutput2Type = " + output2Type);
            System.out.println("\toutput2Name = " + output2Name);

        }
        catch (Exception e){
            //TODO: create more detailed error messages, to see where the error
            //occurred
            System.out.println("Invalid config file");
            e.printStackTrace();
        }
        System.out.println("");
    }
}
```

A.2 LayerInfo.java

```
package skynet;
/**
 * A simple class, designed to hold the information required to create a
 * layer in the neural network
 * @author bwinrich
 */

class LayerInfo{

    /**
     * An integer for the type of activation function (see comments in
     * config file for details)
     */
    private int activationFunction;

    /**
     * A boolean for if the layer has a bias node or not
     */
    private boolean isBiased;

    /**
     * An integer for the number of normal neurons in the layer
     */
    private int neurons;

    /**
     * A constructor with parameters. We have no need for a default
     * constructor
     * @param activationFunction type of activation function
```

```
     * @param isBiased is there a bias node
     * @param neurons number of normal neurons
     */
    public LayerInfo(int activationFunction, boolean isBiased, int neurons){
        this.activationFunction = activationFunction;
        this.isBiased = isBiased;
        this.neurons = neurons;
    }

    /**
     * Accessor method for activationFunction
     * @return the activationFunction
     */
    public int getActivationFunction() {
        return activationFunction;
    }

    /**
     * Accessor method for isBiased
     * @return the isBiased
     */
    public boolean isBiased() {
        return isBiased;
    }

    /**
     * Accessor method for neurons
     * @return the neurons
     */
    public int getNeurons() {
        return neurons;
    }

    /** A method used for returning the information for the layer in an
     * easy-to-read format, so that it can be printed.
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString(){

        String activation = null;

        switch(activationFunction){
        case -1:
            activation = "n/a";
            break;
```

```
        case 0:
            activation = "Sigmoid";
            break;
        case 1:
            activation = "Hyperbolic Tangent";
            break;
        case 2:
            activation = "Linear";
            break;
        case 3:
            activation = "Elliott";
            break;
        case 4:
            activation = "Gaussian";
            break;
        case 5:
            activation = "Logarithmic";
            break;
        case 6:
            activation = "Ramp";
            break;
        case 7:
            activation = "Sine";
            break;
        case 8:
            activation = "Step";
            break;
        case 9:
            activation = "BiPolar";
            break;
        case 10:
            activation = "Bipolar Sigmoid";
            break;
        case 11:
            activation = "Clipped Linear";
            break;
        case 12:
            activation = "Competitive";
            break;
        case 13:
            activation = "Elliott Symmetric";
            break;
        case 14:
            activation = "Softmax";
            break;
        case 15:
```

```
            activation = "Steepened Sigmoid";
            break;
         default:
            activation = "Invalid";
            break;
         }

         return ("Layer: (" + activation + ", " + isBiased + ", " + neurons
               + ")");
      }
   }
```

A.3 OutputWriter.java

```
package skynet;
import java.io.BufferedWriter;
import java.io.File;
import java.io.IOException;
import org.encog.engine.network.activation.*;
import org.encog.neural.flat.FlatNetwork;
import org.encog.neural.networks.BasicNetwork;

/**
 * A parent class for other OutputWriters. This class holds all of the
 * shared methods required to create a file and output the code/formula for
 * a trained Artificial Neural Network.
 * @author bwinrich
 */
public abstract class OutputWriter {

   /**
    * The file to write to
    */
   protected File file2;

   /**
    * A BufferedWriter for file writing
    */
   protected BufferedWriter bw2 = null;

   /**
    * The name of the file
    */
   protected String outputName;

   /**
    * Does the data set have headers?
```

```
     */
    protected boolean hasHeaders;

    /**
     * Array to hold the column names (only if the data set has headers)
     */
    protected String[] columnNames;

    /**
     * The data structure for the ANN
     */
    protected BasicNetwork network;

    /**
     * The number of input nodes
     */
    protected int inputCount;

    /**
     * The number of output nodes
     */
    protected int outputCount;

    /**
     * The number of neurons in each layer (including bias neurons)
     */
    protected int[] numberOfTotalNeurons;

    /**
     * The number of neurons in each layer (excluding bias neurons)
     */
    protected int[] numberOfNormalNeurons;

    /**
     * The value of the biases of each layer, if applicable
     */
    protected double[] biases;

    /**
     * The flattened version of the ANN
     */
    protected FlatNetwork myFlat;

    /**
     * The number of layers in the ANN
     */
```

```java
    protected int layers;


    /**
     * Default constructor
     */
    public OutputWriter(){}

    /**
     * Mutator method for hasHeaders
     * @param hasHeaders the hasHeaders to set
     */
    public void setHasHeaders(boolean hasHeaders) {
        this.hasHeaders = hasHeaders;
    }

    /**
     * Mutator method for columnNames
     * @param columnNames the columnNames to set
     */
    public void setColumnNames(String[] columnNames) {
        this.columnNames = columnNames;
    }

    /**
     * Mutator method for network
     * @param network the network to set
     */
    public void setNetwork(BasicNetwork network) {
        this.network = network;
    }

    /**
     * Mutator method for inputCount
     * @param inputCount the inputCount to set
     */
    public void setInputCount(int inputCount) {
        this.inputCount = inputCount;
    }

    /**
     * Mutator method for outputCount
     * @param outputCount the outputCount to set
     */
    public void setOutputCount(int outputCount) {
        this.outputCount = outputCount;
```

```java
    }

    /**
     * Mutator method for numberOfTotalNeurons
     * @param numberOfTotalNeurons the numberOfTotalNeurons to set
     */
    public void setNumberOfTotalNeurons(int[] numberOfTotalNeurons) {
        this.numberOfTotalNeurons = numberOfTotalNeurons;
    }

    /**
     * Mutator method for numberOfNormalNeurons
     * @param numberOfNormalNeurons the numberOfNormalNeurons to set
     */
    public void setNumberOfNormalNeurons(int[] numberOfNormalNeurons) {
        this.numberOfNormalNeurons = numberOfNormalNeurons;
    }

    /**
     * Mutator method for biases
     * @param biases the biases to set
     */
    private void setBiases(double[] biases) {
        this.biases = biases;
    }

    /**
     * Mutator method for myFlat
     * @param myFlat the myFlat to set
     */
    private void setMyFlat(FlatNetwork myFlat) {
        this.myFlat = myFlat;
    }

    /**
     * Mutator method for layers
     * @param layers the layers to set
     */
    public void setLayers(int layers) {
        this.layers = layers;
    }

    /**
     * Some variables can be initialized using information already passed to
     * the class.
     */
```

```java
    public void initializeOtherVariables(){
        setMyFlat(network.getStructure().getFlat());
        setBiases(myFlat.getBiasActivation());

    }

    /**
     * Creates the file used for output.
     * @param output2Name the name of
     */
    public abstract void createFile(String output2Name);

    /**
     * Writes to the output file. Each String passed as a parameter is
     * written on its own line
     * @param stuff The line to be written to the file
     */
    protected void writeTwo(String stuff){
        try{
            bw2.write(stuff);
            bw2.newLine();
        }catch (IOException e){
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    /**
     * Parses the equation of the activation function and returns it in
     * String form
     * @param af The activation function to parse
     * @param varName The variable passed to the activation function
     * @param targetVarName The variable the result of the activation
     * function will be stored in
     * @return The parsed form of the activation function in String form
     */
    protected abstract String parseActivationFunction(ActivationFunction af,
            String varName, String targetVarName);

    /**
     * A lengthy method for writing the code/formula for the neural network
     * to a file. Each child class will have its own implementation.
     */
    public abstract void writeFile();
}
```

A.4 OutputWriterTxt.java

```java
package skynet;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import org.encog.engine.network.activation.*;

/**
 * A child class of OutputWriter, used for creating .txt files
 * @author bwinrich
 */
public class OutputWriterTxt extends OutputWriter{

   /**
    * Default Constructor
    */
   public OutputWriterTxt(){}

   /* (non-Javadoc)
    * @see OutputWriter#writeFile()
    */
   @Override
   public void writeFile() {

      writeTwo("//Variable declarations");

      //Variables from headers of csv file, if applicable
      if(hasHeaders){
         writeTwo("//Header Names");
         for (String s: columnNames){
            writeTwo(s);
         }
      }

      //variables - input layer
      writeTwo("//Input layer");
      for (int i = 0; i<inputCount; i++)
      {
         writeTwo("i"+ i);
      }
      for (int i = inputCount; i<numberOfTotalNeurons[0]; i++)
      {
         writeTwo("i" + i + " = " + biases[biases.length-1]);
      }

      //variables - hidden layers
```

```
      writeTwo("//Hidden layer(s)");
      for (int i=1; i<layers-1; i++){
         writeTwo("//Hidden Layer " + i);
         for (int j = 0; j<numberOfNormalNeurons[i]; j++){
            for (int k = 0; k < numberOfTotalNeurons[i-1]; k++){
               writeTwo("h" + i + "n" + j + "f" + k);
            }
            writeTwo("h" + i + "n" + j + "t");
            writeTwo("h" + i + "n" + j);
         }
         for (int j = numberOfNormalNeurons[i]; j<numberOfTotalNeurons[i];
               j++){
            writeTwo("h" + i + "n" + j + " = " + biases[biases.length-i-1]);
         }
      }

      //varibles - output layer
      writeTwo("//Output layer");
      for (int i=0; i<outputCount; i++){
         for (int j = 0; j < numberOfTotalNeurons[layers-2]; j++){
            writeTwo("o" + i + "f" + j);
         }
         writeTwo("o" + i + "t");
         writeTwo("o" + i);
      }

      writeTwo("");

      double weight;
      String sum = "";

      //Some extra code if we have headers, to set the default input
      //variables to the header variables
      if(hasHeaders){
         for (int i = 0; i < inputCount; i++){
            writeTwo("i" + i + " = " + columnNames[i]);
         }
      }

      //Hidden layers calculation
      for (int i = 1; i<layers-1; i++){
         for (int j = 0; j<numberOfNormalNeurons[i]; j++){
            writeTwo("");

            sum = "";

```

```java
            for (int k = 0; k < numberOfTotalNeurons[i-1]; k++){
               weight = network.getWeight(i-1,k,j);
               if (i == 1)
               {
                  writeTwo("h" + i + "n" + j + "f" + k + " = i" + k + " * "
                        + weight);
               }else{
                  writeTwo("h" + i + "n" + j + "f" + k + " = h" + (i-1) + "n"
                        + k + " * " + weight);
               }

               if (k == 0){
                  sum = "h" + i + "n" + j + "f" + k;
               }else{
                  sum += " + h" + i + "n" + j + "f" + k;
               }
            }
            writeTwo("h" + i + "n" + j + "t = " + sum);

            String af = parseActivationFunction(network.getActivation(i),
                  "h" + i + "n" + j + "t", "h" + i + "n" + j);
            writeTwo(af.substring(0,af.length()-1));
         }
      }

      //Output layer calculation
      writeTwo("");

      sum = "";

      for (int i = 0; i<outputCount; i++){
         for (int j = 0; j<numberOfTotalNeurons[layers-2]; j++){
            weight = network.getWeight(layers-2,j,i);
            writeTwo ("o" + i + "f" + j + " = h" + (layers-2) + "n" + j
                  + " * " + weight);

            if (j == 0){
               sum = "o" + i + "f" + j;
            }else{
               sum += " + o" + i + "f" + j;
            }
         }
         writeTwo("o" + i + "t = " + sum);

         String af = parseActivationFunction(network.getActivation(layers-1),
               "o" + i + "t", "o" + i);
```

```java
         writeTwo(af.substring(0,af.length()-1));
      }

      //Some extra code if we have headers, to set the default input
      //variables to the header variables
      if (hasHeaders){
         writeTwo("");

         for (int i = 0; i < outputCount; i++){
            writeTwo(columnNames[i + inputCount] + " = o" + i);
         }
      }

      writeTwo("");

      try{
         if(bw2!=null)
            bw2.close();
      }catch(Exception ex){
         System.out.println("Error in closing the BufferedWriter"+ex);
      }
   }

   /* (non-Javadoc)
    * @see OutputWriter#createFile(java.lang.String)
    */
   @Override
   public void createFile(String output2Name){

      outputName = output2Name;

      try{
         file2 = new File(output2Name + ".txt");
         if (!file2.exists()) {
            file2.createNewFile();
         }

         FileWriter fw2 = new FileWriter(file2);
         bw2 = new BufferedWriter(fw2);
      }catch (IOException e){
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
   }

   @Override
```

```java
    protected String parseActivationFunction(ActivationFunction af,
          String varName, String targetVarName){
       String text = null;

       if (af instanceof ActivationSigmoid){
          text = targetVarName + " = 1.0 / (1.0 + e^(-1 * " + varName + "))";
       }else if (af instanceof ActivationTANH){
          text = targetVarName + " = tanh(" + varName + ")";
       }else if (af instanceof ActivationLinear){
          text = targetVarName + " = " + varName;
       }else if (af instanceof ActivationElliott){
          double s = af.getParams()[0];
          text = targetVarName + " = ((" + varName + " * " + s
                + ") / 2) / (1 + |" + varName + " * " + s + "|) + 0.5";
       }else if (af instanceof ActivationGaussian){
          text = targetVarName + " = e^(-(2.5*" + varName + ")^2)";
       }else if (af instanceof ActivationLOG){
          text = "if(" + varName + " >= 0){\n\t" + targetVarName
                + " = log(1 + " + varName + ")\n}else{\n\t" + targetVarName
                + " = -log(1 - " + varName + ")\n}";
       }else if (af instanceof ActivationRamp){
          double paramRampHighThreshold =
                ((ActivationRamp)(af)).getThresholdHigh();
          double paramRampLowThreshold =
                ((ActivationRamp)(af)).getThresholdLow();
          double paramRampHigh = ((ActivationRamp)(af)).getHigh();
          double paramRampLow = ((ActivationRamp)(af)).getLow();
          double slope = (paramRampHighThreshold-paramRampLowThreshold)
                / (paramRampHigh-paramRampLow);

          text = "if(" + varName + " < " + paramRampLowThreshold + ") {\n\t"
                + targetVarName + " = " + paramRampLow + "\n} else if ("
                + varName + " > " + paramRampHighThreshold + ") {\n\t"
                + targetVarName + " = " + paramRampHigh + "\n} else {\n\t"
                + targetVarName + " = (" + slope + " * " + varName + ")";
       }else if (af instanceof ActivationSIN){
          text = targetVarName + " = sin(2.0*" + varName + ")";
       }else if (af instanceof ActivationStep){
          double paramStepCenter = ((ActivationStep)(af)).getCenter();
          double paramStepLow = ((ActivationStep)(af)).getLow();
          double paramStepHigh = ((ActivationStep)(af)).getHigh();

          text = "if (" + varName + ">= " + paramStepCenter + ") {\n\t"
                + targetVarName + " = " + paramStepHigh + "\n} else {\n\t"
                + targetVarName + " = " + paramStepLow + "\n}";
       }else if (af instanceof ActivationBiPolar){
```

```
            text = "if(" + varName + " > 0) {\n\t" + targetVarName
                  + " = 1\n} else {\n\t" + targetVarName + " = -1\n}";
         }else if (af instanceof ActivationBipolarSteepenedSigmoid){
            text = targetVarName + " = (2.0 / (1.0 + e^(-4.9 * " + varName
                  + "))) - 1.0";
         }else if (af instanceof ActivationClippedLinear){
            text = "if(" + varName + " < -1.0) {\n\t" + targetVarName
                  + " = -1.0\n} else if (" + varName + " > 1.0) {\n\t"
                  + targetVarName + " = 1.0\n} else {\n\t" + targetVarName
                  + " = " + varName + "\n}";
         }else if (af instanceof ActivationElliottSymmetric){
            double s = af.getParams()[0];
            text = targetVarName + " = (" + varName + "*" + s + ") / (1 + |"
                  + varName + "*" + s + "|)";
         }else if (af instanceof ActivationSteepenedSigmoid){
            text = targetVarName + " = 1.0 / (1.0 + e^(-4.9 * " + varName
                  + "))";
         }else{
            //Unimplemented activation function: Softmax (complicated)
            //Unimplemented activation function: Competitive (complicated,
            //non-differentiable)
            //in Encog 3.3 there aren't any other activation functions, so
            //unless someone implements their own we shouldn't get to this point
            text = "Error: unknown activation function";
         }
         return text;
      }
   }
```

A.5 OutputWriterJava.java

```
package skynet;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import org.encog.engine.network.activation.*;

/**
 * A child class out OutputWriter, used for creating .java files.
 * @author bwinrich
 */
public class OutputWriterJava extends OutputWriter{

   private boolean standalone;

   /**
```

```java
     * Default constructor
     */
    public OutputWriterJava(){}

    /**
     * Constructor with parameter
     * @param standalone main method or not
     */
    public OutputWriterJava(boolean standalone){
       this.standalone=standalone;
    }

    /* (non-Javadoc)
     * @see OutputWriter#writeFile()
     */
    @Override
    public void writeFile() {

       writeTwo("import java.lang.Math;");

       writeTwo("");

       writeTwo("public class " + outputName);
       writeTwo("{");

       writeTwo("\t//Variable declarations");

       //Variables from headers of csv file, if applicable
       if(hasHeaders){
          writeTwo("\t//Header Names");
          for (String s: columnNames){
             writeTwo("\tpublic static double " + s + ";");
          }
       }

       //variables - input layer
       writeTwo("\t//Input layer");
       for (int i = 0; i<inputCount; i++)
       {
          writeTwo("\tpublic static double i"+ i + ";");
       }
       for (int i = inputCount; i<numberOfTotalNeurons[0]; i++)
       {
          writeTwo("\tprivate static double i" + i + " = "
                + biases[biases.length-1] + ";");
       }
```

```
        //variables - hidden layers
        writeTwo("\t//Hidden layer(s)");
        for (int i=1; i<layers-1; i++){
          writeTwo("\t//Hidden Layer " + i);
          for (int j = 0; j<numberOfNormalNeurons[i]; j++){
            for (int k = 0; k < numberOfTotalNeurons[i-1]; k++){
              writeTwo("\tprivate static double h" + i + "n" + j + "f" + k
                  +";");
            }
            writeTwo("\tprivate static double h" + i + "n" + j + "t;");
            writeTwo("\tprivate static double h" + i + "n" + j + ";");
          }
          for (int j = numberOfNormalNeurons[i]; j<numberOfTotalNeurons[i];
              j++){
            writeTwo("\tprivate static double h" + i + "n" + j + " = "
                + biases[biases.length-i-1]+";");
          }
        }

        //varibles - output layer
        writeTwo("\t//Output layer");
        for (int i=0; i<outputCount; i++){
          for (int j = 0; j < numberOfTotalNeurons[layers-2]; j++){
            writeTwo("\tprivate static double o" + i + "f" + j +";");
          }
          writeTwo("\tprivate static double o" + i + "t;");
          writeTwo("\tpublic static double o" + i + ";");
        }

        writeTwo("");

        //standalone files will have a main method
        if(standalone){

          //TODO: customize this (from Tiberius)
          writeTwo("\tpublic static void main(String[] args)");
          writeTwo("\t{");
          writeTwo("\t\tinitData();");
          writeTwo("\t\tcalcNet();");
          for (int i = 0; i < outputCount; i++){
            writeTwo("\t\tSystem.out.println(o" + i + ");" );
          }
          writeTwo("\t}");
          writeTwo("");

```

```java
        //TODO: customize this (from Tiberius)
        writeTwo("\tpublic static void initData()");
        writeTwo("\t{");
        writeTwo("\t\t//data is set here");
        for (int i = 0; i < inputCount; i++){
            if(hasHeaders){
                writeTwo("\t\t" + columnNames[i] + " = 1;");
            }else{
                writeTwo("\t\ti" + i + " = 1;");
            }
        }
        writeTwo("\t}");
        writeTwo("");
    }

    double weight;
    String sum = "";

    writeTwo("\tpublic static void calcNet()");
    writeTwo("\t{");

    //Some extra code if we have headers, to set the default input
    //variables to the header variables
    if (hasHeaders){
        for (int i = 0; i < inputCount; i++){
            writeTwo("\t\ti" + i + " = " + columnNames[i] + ";");
        }
    }

    //Hidden layers calculation
    for (int i = 1; i<layers-1; i++){
        for (int j = 0; j<numberOfNormalNeurons[i]; j++){
            writeTwo("");

            sum = "";

            for (int k = 0; k < numberOfTotalNeurons[i-1]; k++){
                weight = network.getWeight(i-1,k,j);
                if (i == 1)
                {
                    writeTwo("\t\th" + i + "n" + j + "f" + k + " = i" + k
                            + " * " + weight + ";");
                }else{
                    writeTwo("\t\th" + i + "n" + j + "f" + k + " = h" + (i-1)
                            + "n" + k + " * " + weight + ";");
                }
```

```
            if (k == 0){
               sum = "h" + i + "n" + j + "f" + k;
            }else{
               sum += " + h" + i + "n" + j + "f" + k;
            }
         }

         writeTwo("\t\th" + i + "n" + j + "t = " + sum + ";");

         String af = parseActivationFunction(network.getActivation(i),
               "h" + i + "n" + j + "t", "h" + i + "n" + j);
         writeTwo("\t\t" + af);
      }
   }

   //Output layer calculation
   writeTwo("");

   sum = "";

   for (int i = 0; i<outputCount; i++){
      for (int j = 0; j<numberOfTotalNeurons[layers-2]; j++){
         weight = network.getWeight(layers-2,j,i);
         writeTwo("\t\to" + i + "f" + j + " = h" + (layers-2) + "n" + j
               + " * " + weight + ";");

         if (j == 0){
            sum = "o" + i + "f" + j;
         }else{
            sum += " + o" + i + "f" + j;
         }
      }
      writeTwo("\t\to" + i + "t = " + sum + ";");

      String af = parseActivationFunction(network.getActivation(layers-1),
            "o" + i + "t", "o" + i);
      writeTwo("\t\t" + af);
   }

   //Some extra code if we have headers, to set the default input
   //variables to the header variables
   if (hasHeaders){
      writeTwo("");
      for (int i = 0; i < outputCount; i++){
         writeTwo("\t\t" + columnNames[i + inputCount] + " = o" + i
```

```java
                        + ";");
            }
        }

        writeTwo("\t}");

        writeTwo("");
        writeTwo("}");

        try{
            if(bw2!=null)
                bw2.close();
        }catch(Exception ex){
            System.out.println("Error in closing the BufferedWriter"+ex);
        }
    }

    /* (non-Javadoc)
     * @see OutputWriter#createFile(java.lang.String)
     */
    @Override
    public void createFile(String output2Name){

        outputName = output2Name;

        try{
            file2 = new File(output2Name + ".java");
            if (!file2.exists()) {
                file2.createNewFile();
            }

            FileWriter fw2 = new FileWriter(file2);
            bw2 = new BufferedWriter(fw2);
        }catch (IOException e){
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    @Override
    protected String parseActivationFunction(ActivationFunction af,
            String varName, String targetVarName){
        String text = null;

        if (af instanceof ActivationSigmoid){
            text = targetVarName + " = 1.0 / (1.0 + Math.exp(-1 * " + varName
```

```
            + "));";
    }else if (af instanceof ActivationTANH){
        text = targetVarName + " = Math.tanh(" + varName + ");";
    }else if (af instanceof ActivationLinear){
        text = targetVarName + " = " + varName + ";";
    }else if (af instanceof ActivationElliott){
        double s = af.getParams()[0];
        text = targetVarName + " = ((" + varName + " * " + s
            + ") / 2) / (1 + Math.abs(" + varName + " * " + s
            + ")) + 0.5;";
    }else if (af instanceof ActivationGaussian){
        text = targetVarName + " = Math.exp(-Math.pow(2.5*" + varName
            + ",2.0));";
    }else if (af instanceof ActivationLOG){
        text = "if(" + varName + " >= 0){\n\t" + targetVarName
            + " = Math.log(1 + " + varName + ");\n}else{\n\t"
            + targetVarName + " = -Math.log(1 - " + varName + ");\n}";
    }else if (af instanceof ActivationRamp){
        double paramRampHighThreshold =
            ((ActivationRamp)(af)).getThresholdHigh();
        double paramRampLowThreshold =
            ((ActivationRamp)(af)).getThresholdLow();
        double paramRampHigh = ((ActivationRamp)(af)).getHigh();
        double paramRampLow = ((ActivationRamp)(af)).getLow();
        double slope = (paramRampHighThreshold-paramRampLowThreshold)
            / (paramRampHigh-paramRampLow);

        text = "if(" + varName + " < " + paramRampLowThreshold + ") {\n\t"
            + targetVarName + " = " + paramRampLow + ";\n} else if ("
            + varName + " > " + paramRampHighThreshold + ") {\n\t"
            + targetVarName + " = " + paramRampHigh + ";\n} else {\n\t"
            + targetVarName + " = (" + slope + " * " + varName + ");";
    }else if (af instanceof ActivationSIN){
        text = targetVarName + " = Math.sin(2.0*" + varName + ");";
    }else if (af instanceof ActivationStep){
        double paramStepCenter = ((ActivationStep)(af)).getCenter();
        double paramStepLow = ((ActivationStep)(af)).getLow();
        double paramStepHigh = ((ActivationStep)(af)).getHigh();

        text = "if (" + varName + ">= " + paramStepCenter + ") {\n\t"
            + targetVarName + " = " + paramStepHigh + ";\n} else {\n\t"
            + targetVarName + " = " + paramStepLow + ";\n}";
    }else if (af instanceof ActivationBiPolar){
        text = "if(" + varName + " > 0) {\n\t" + targetVarName
            + " = 1;\n} else {\n\t" + targetVarName + " = -1;\n}";
    }else if (af instanceof ActivationBipolarSteepenedSigmoid){
```

```java
         text = targetVarName + " = (2.0 / (1.0 + Math.exp(-4.9 * "
              + varName + "))) - 1.0;";
      }else if (af instanceof ActivationClippedLinear){

         text = "if(" + varName + " < -1.0) {\n\t" + targetVarName
              + " = -1.0;\n} else if (" + varName + " > 1.0) {\n\t"
              + targetVarName + " = 1.0;\n} else {\n\t" + targetVarName
              + " = " + varName + ";\n}";
      }else if (af instanceof ActivationElliottSymmetric){
         double s = af.getParams()[0];
         text = targetVarName + " = (" + varName + "*" + s
              + ") / (1 + Math.abs(" + varName + "*" + s + "));";
      }else if (af instanceof ActivationSteepenedSigmoid){
         text = targetVarName + " = 1.0 / (1.0 + Math.exp(-4.9 * " + varName
              + "));";
      }else{
         //Unimplemented activation function: Softmax (complicated)
         //Unimplemented activation function: Competitive (complicated,
         //non-differentiable)
         //in Encog 3.3 there aren't any other activation functions, so
         //unless someone implements their own we shouldn't get to this point
         text = "Error: unknown activation function";
      }
      return text;
   }
}
```

A.6 TrainingSetUtil.java (modified)

```java
/*
 * Encog(tm) Core v3.3 - Java Version
 * http://www.heatonresearch.com/encog/
 * https://github.com/encog/encog-java-core

 * Copyright 2008-2014 Heaton Research, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
```

```java
 *
 * For more information on Heaton Research copyrights, licenses
 * and trademarks visit:
 * http://www.heatonresearch.com/copyright
 */
package org.encog.util.simple;

/*
 * modified: added condition so that it will ignore rows with incomplete
     data
 */

/*
 * Additional modification: added way to retrieve header information
 */

import java.util.ArrayList;
import java.util.List;

import org.encog.ml.data.MLData;
import org.encog.ml.data.MLDataPair;
import org.encog.ml.data.MLDataSet;
import org.encog.ml.data.basic.BasicMLData;
import org.encog.ml.data.basic.BasicMLDataPair;
import org.encog.ml.data.basic.BasicMLDataSet;
import org.encog.util.EngineArray;
import org.encog.util.ObjectPair;
import org.encog.util.csv.CSVError;
import org.encog.util.csv.CSVFormat;
import org.encog.util.csv.ReadCSV;

public class TrainingSetUtil {

  private static List<String> columnNames = new ArrayList<String>();

  /**
   * Load a CSV file into a memory dataset.
   * @param format The CSV format to use.
   * @param filename The filename to load.
   * @param headers True if there is a header line.
   * @param inputSize The input size. Input always comes first in a file.
   * @param idealSize The ideal size, 0 for unsupervised.
   * @return A NeuralDataSet that holds the contents of the CSV file.
   */
  public static MLDataSet loadCSVTOMemory(CSVFormat format,
        String filename, boolean headers, int inputSize, int idealSize) {
```

```java
    MLDataSet result = new BasicMLDataSet();
    ReadCSV csv = new ReadCSV(filename, headers, format);

    if(headers){
      columnNames = csv.getColumnNames();
    }


    int ignored = 0;

    while (csv.next()) {
      MLData input = null;
      MLData ideal = null;
      int index = 0;
      try{
        input = new BasicMLData(inputSize);
        for (int i = 0; i < inputSize; i++) {
          double d = csv.getDouble(index++);
          input.setData(i, d);
        }

        if (idealSize > 0) {
          ideal = new BasicMLData(idealSize);
          for (int i = 0; i < idealSize; i++) {
            double d = csv.getDouble(index++);
            ideal.setData(i, d);
          }
        }

        MLDataPair pair = new BasicMLDataPair(input, ideal);
        result.add(pair);
      }catch (CSVError e){
        ignored++;

        //e.printStackTrace();
      }
    }
    System.out.println("Rows ignored: " + ignored);

    return result;
  }

  public static ObjectPair<double[][], double[][]> trainingToArray(
      MLDataSet training) {
    int length = (int)training.getRecordCount();
    double[][] a = new double[length][training.getInputSize()];
```

```
    double[][] b = new double[length][training.getIdealSize()];

    int index = 0;
    for (MLDataPair pair : training) {
      EngineArray.arrayCopy(pair.getInputArray(), a[index]);
      EngineArray.arrayCopy(pair.getIdealArray(), b[index]);
      index++;
    }

    return new ObjectPair<double[][], double[][]>(a, b);
  }

  /**
   * @return the columnNames
   */
  public static List<String> getColumnNames() {
    return columnNames;
  }


}
```

BIBLIOGRAPHY

Charles, D. and Mcglinchey, S., "The past, present and future of artificial neural networks in digital games," *Proceedings of the 5th international conference on computer games: artificial intelligence, design and education*, pp. 163–169, 2004.

Clabaugh, C., Myszewski, D., and Pang, J. "History: The 1940's to the 1970's." [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html

Cortes, C. and Vapnik, V., "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF00994018

Elliott, D. L., "A better activation function for artificial neural networks," 1993.

Galway, L., Charles, D., and Black, M., "Machine learning in digital games: a survey," *Artificial Intelligence Review*, vol. 29, no. 2, pp. 123–161, 2008.

Heaton, J. "The number of hidden layers." September 2008. [Online]. Available: http://www.heatonresearch.com/node/707

Heaton, J. "Elliott activation function." September 2011. [Online]. Available: http://www.heatonresearch.com/wiki/Elliott_Activation_Function

Heaton, J. "Range normalization." September 2011. [Online]. Available: http://www.heatonresearch.com/wiki/Range_Normalization

Hebb, D., "The organization of behavior; a neuropsychological theory." 1949.

Jain, A. K., Mao, J., and Mohiuddin, K., "Artificial neural networks: A tutorial," 1996.

McCulloch, W. and Pitts, W., "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943. [Online]. Available: http://dx.doi.org/10.1007/BF02478259

Riedmiller, M. and Braun, H., "A direct adaptive method for faster backpropagation learning: The rprop algorithm," in *Neural Networks, 1993., IEEE International Conference on.* IEEE, 1993, pp. 586–591.

Rojas, R., "The backpropagation algorithm," in *Neural Networks.* Springer, 1996, pp. 149–182.

Rosenblatt, F., "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

Stuart, K. "How to get into the games industry – an insiders' guide." March 2014. [Online]. Available: http://www.theguardian.com/technology/2014/mar/20/how-to-get-into-the-games-industry-an-insiders-guide

Widrow, B., Hoff, M. E., *et al.*, "Adaptive switching circuits." 1960.