Solving Data Classification Problems Using Intelligent Agents

Chris R. Adams

KEYWORDS: Neural Computing and Networks, Learning and Reasoning, Data Mining, Java Beans, Intelligent Agents, Data Classification, IBM Agent Building and Learning Environment (ABLE)

Abstract

Computing problems such as pattern recognition in speech or imaging, data classification, or machine learning and reasoning are difficult to solve using traditional programming logic because of the seemingly infinite number of conditions that can occur within related environments. For example, how do you program a computer to recognize a spoken word with unknown background noise using traditional if-else logic? If you don't know the condition, neither will your program. Neural computing, a programming model that emulates the human brain neuron and synapse networks (in a simple form), has proven to be a much better approach to resolving these types of problems. Neural networks are the core of this technology and after 40 years of university-based research they are finding more application in the world of information technology through data classification and prediction (data mining). In this paper, I'll give a quick introduction to neural computing, go into some detail on a specific neural network called back propagation that is commonly used in data classification problems, introduce the IBM Agent Building and Learning Environment (ABLE) toolkit that offers a great environment for developing neural agents and applications and lastly, I will demonstrate how to use a Java-based hybrid intelligent agent to solve a data classification problem. The agent will do the work of a stock analyst by helping to choose companies that are the best financial investment based on a defined set of criteria.

Neural Networks

A neural network is a collection of simple processors or *nodes* that each work on their local data. The nodes communicate by unidirectional *adaptive connections*, which carry numeric (value as

opposed to reference) data. The nodes then sum up their inputs and calculate an output value, or *activation*, and send it to other processing nodes in the neural network.

The behavior of a node in a network can be thought of as follows. Imagine an electrical circuit consisting of a light switch, a dimmer (regulates brightness) and a light bulb. Referencing Figure 1, X_j represents the light switch, Z_{ij} represents the dimmer and X_i represents the light bulb. When you turn on the light (or activate the node) you can control the brightness of the light bulb by adjusting the dimmer. The transfer function is an inherent property of the bulb that also affects its output.

In our neural network, the activation output, X_i, is a function of X_j (the prior node's activation), the connection weight between the two nodes, Z_{ij}, and a transformation function. The index notation suggests that j may represent a large numbers of nodes, in which the combination of all activation outputs and their respected connection weights are summated. An output value, X_i, is then computed by filtering the weighted sum through a nonlinear transfer function (typically sigmoidal). The nonlinear transfer function gives the neural net its ability to model nonlinear relationships.

The connections are called adaptive because they are adjusted during the training of the neural network. This training process usually consists of presenting examples of input/output relationships to the network, which is called S*upervised* training. The *connection weights* are adjusted to minimize the difference between the actual network output value and the desired output value. The weights are a critical component in our network, as I'll show when describing the dynamics of a back propagation network next.

Back Propagation

Of over thirty neural network models, back propagation is one of the most commonly used for data classification problems although it is quite expensive computationally. Its method is based on a gradient decent in error where error means the difference between the actual and target output of a node. It is a feed-forward network meaning that all calculations are done in a "one-way" fashion. There are no backward or lateral calculations made. Figure 2 shows the general architecture of a back propagation neural network.

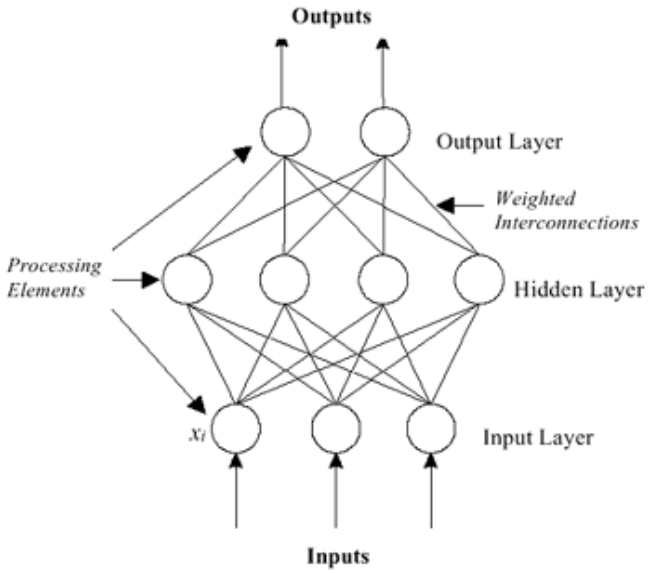

Figure 2: Back Propagation Network

The hidden layer is used to assist in more complex problems and in facilitating network training. We won't require it in our example, but it is easily invoked when needed.

Back propagation learning proceeds in the following way: an input pattern is chosen from a set of input patterns (a pattern can be synonymous with a record from a database table or file). This input pattern determines the activations of the input nodes. Setting the activations of the input layer nodes is followed by the *activation forward propagation* phase: the activation values of first the hidden units and then the output units are computed. This is done by using a sigmoid activation function, like:

$$X_i = \frac{1}{1 + e^{-S\left(\sum_{j=1}^{n} X_j Z_{ji} - I\right)}}$$ where

- X_i is activation of the i^{th} node in the output layer (or hidden layer if used)
- $\sum_{j=1}^{n} X_j Z_{ji}$ is a derived term summing the product of prior active nodes and applied weights
- Z_{ji} is the weight of the connection from the i^{th} node in the previous layer to the j^{th} node in the output (or hidden layer)
- S is a constant which determines the steepness of the sigmoid function
- I is the bias which determines the shift of the sigmoid function

This activation forward propagation phase is followed by the *error backward propagation* phase.

We first compute the error for the output layer nodes with the relation:

$$E_i = X_i(Y_i - X_i)(1 - X_i)$$ where

- E_i is the error for the i^{th} node of the output layer
- Y_i is the target activation for the i^{th} node of the output layer
- X_i is the actual activation for the i^{th} node of the output layer

Then we compute the error values for all the hidden layers in the network.

$$E_i = X_i(1 - X_i)\sum_{j=1}^{n} E_j Z_{ji}$$ where

- E_i is the error for the i^{th} node of the hidden layer
- E_j is the error for the j^{th} node in the layer above
- X_i is the actual activation for the i^{th} node of the hidden layer
- Z_{ji} is the weight of the connection between the hidden layer and the j^{th} node in the layer above

At the end of the error backward propagation phase, nodes of the network (except the input layer nodes) will have error values. The error value of a node is used to compute new weights for the connections that lead to the node. New weights are computed using the relation:

$$Z_{ij} = Z_{ij} + \Delta Z_{ij}$$ where

- Z_{ij} is the weight of the connection between the i^{th} node in the previous layer and the j^{th} node in the outer layer (or hidden).
- ΔZ_{ij} is the weight change for the connection between the i^{th} node in the previous layer and the j^{th} node in the outer layer (or hidden).

Up to this point, the structure of the network is in place. What regulates or controls the behavior of the network (helps our values converge) is the weight change relation:

$$\Delta Z_{ij} = \mu \Delta Z_{ij} + \lambda X_i E_j$$ where

- λ is the learning rate (we will set this during the training)

- E_j is the error for the j^{th} node
- X_i is the activation for the i^{th} node in the previous layer from which the connection originates
- Z_{ij} is the previous weight change
- m is the momentum parameter (also set during the training)

I'll talk about the learning rate and momentum a bit later. The important point here is that the weight change becomes our control mechanism to stabilize the network. By effectively altering the weights after each pattern is presented, we control the activation that other nodes will experience. The weight change can occur after each input pattern is presented or after each epoch (one full cycle through the patterns).

A Demonstration of Neural Computing

Now that we have some background on neural computing, we will see how we can apply this technology to solve data classification problems using a set of Java-based agent building tools. In this demonstration, we will create a hybrid intelligent agent that can do the work of a stock analyst in finding quality stocks for us. The agent will be trained on the financial data of companies, analysis results and the selection process used by stock analysts in the past. Once we have successfully trained the agent to know the difference between good and bad investments, we will test its capability to identify good and bad companies for investment. We will also monitor the behavior of the network on a statistical level to verify that the network is stable during operation.

The Fool.com's Rule Maker Strategy Portfolio

I enjoy investing and find it intriguing to determine the best investment strategies for my savings. After a few early years of experimenting on my own (and losing money) I decided to take investing more seriously and focus on a steady, long-term plan rather than trying to make it big on the next Internet or Biotech IPO. The Fool.com investment site gave me a great education on how to do this – simply and wisely. It is a unique place where you can find great information on companies, investment strategies and advice. One of their suggested investment strategies is the "Rule Maker Portfolio" and we will use their criteria to analyze stocks for this example.

The Rule Maker strategy is to invest in dynamic, cash heavy, market leaders. They are the companies that make the 'rules' for their respective industries, hence the name. It's a very sound strategy that has some basic principles and a set of criteria companies must satisfy to participate in the portfolio. The guidelines are of qualitative and quantitative nature, so some of the responses can go either way depending upon the reviewer's point of view. But, generally it is straightforward. There is a lot of good theory behind this strategy, but rather than just reiterating - just go to: http://www.fool.com/portfolios/RuleMaker/RuleMaker.html if you would like more

details.

To set the scenario for our example, let's suppose we are managing the portfolio. In the past we had staff that analyzes the financial data of the companies and once matched against the portfolio criteria, responds with a yes or no for each company reviewed. In our database we have over two years of data of the analysis and decisions made on all companies reviewed. Our goal is to pre-screen all the potential companies prior to the review by an analyst. This will reduce the time they need to spend on companies who clearly will not make the portfolio. Essentially, we will classify the data for the analysts.

I've reduced some of the 'official' Rule Maker criteria to keep the example simple. The following section gives some detail of the criteria we will use and defines some of the terms.

Rule Maker Company Criteria

Five of the main criteria are the following: A company must have

1. **Sales Growth of at Least 10% Year-Over-Year** - Sales growth is the most fundamental indication of an expanding business.
2. **Gross Margins of at Least 50%** - Strong sales growth is great, but only if each dollar of sales is profitable.
3. **Net Profit Margins of at Least 7%** - It is great for a company to be able to sell its products for double the cost of producing them, but it does not mean much if the overhead expenses associated with that product ended up eliminating the profits.
4. **Cash No Less Than 1.5x Total Debt** - True performing companies will have very little or no debt.
5. **Foolish Flow Ratio below 1.25** - In daily company operation, money comes in the front door from sales, and goes out the backdoor from expenses. The Foolish Flow Ratio verifies that the cash stays within the company.

These five criteria require information from both the income statement and balance sheet of a company during a given quarter. The information needed and quantitative descriptions of the ratios are provided below.

Income Statement:
Total Sales (Revenue)
Prior-year Sales
Cost of Goods Sold
Net Income

Balance Sheet:
- Cash & Equivalents
- Current Assets
- Current Liabilities
- Short-term Debt
- Long-term Debt

Rule Maker Ratios:

	Rule Maker Standard	Calculation
Sales Growth	> 10%	$\frac{\text{Total Sales}}{\text{Prior Years Sales}}$
Gross Margins	> 50%	$\frac{\text{(Revenue – Cost of Goods Sold)}}{\text{Revenue}}$
Net Margins	> 7%	$\frac{\text{Net Income}}{\text{Revenue}}$
Cash-to-Debt Ratio	> 1.5 **or** No Debt	$\frac{\text{Cash \& Equivalents}}{\text{(Short Term Debt + Long Term Debt)}}$
Foolish Flow Ratio	< 1.25	$\frac{\text{(Current Assets - Cash \& Equivalent)}}{\text{(Current Liability – Short Term Debt)}}$

The Rule Maker standard values specify what that ratio must be in order to qualify for the portfolio. If the ratio is within the limit, an analyst specifies YES. If not, the analyst specifies NO. A company must receive YES for each ratio in order to be considered in the portfolio.

We will use all the data to train our network. The raw numbers from the statements, the ratio values and the Boolean values stating whether or not they qualify. We will use a select number of companies for training the network. These companies with their respective numbers and analysis results can be found in the Appendix of this document. As stated above, an approval is given only when all criteria is met. This approval is the value that will be determined by our network output during the testing phase.

IBM Agent Building and Learning Environment

Now that we have set the stage for what we want to accomplish in our example, lets explore the tools we will use to create achieve this. There are a plethora of available neural network software packages. Some are quite expensive and others free for personal usage and experimentation. The Agent Building and Learning Environment (**ABLE**) package is a great set of tools to experiment with this type of development. It is available through one of my favorite sites of all time, www.alphaworks.ibm.com. Alphaworks offers all of IBM's latest and greatest "alpha" versioned software to experiment with. Payment, of course, is helping them debugs software. I don't mind because they have so much great stuff to play with and ABLE is no exception. They offer some samples as well (one of which I used to help me do this demonstration) that help you come up to speed quickly on the package's depth.

The ABLE package provides a framework of core Java beans (called *AbleBeans*) and an IDE (the *AbleEditor*) for building hybrid intelligent agents. They include both reasoning and learning. The following table is a brief description taken from the documentation of the Java packages that come with ABLE version 1.0a.

Core Packages	
com.ibm.able	Refer to the ABLE package index for more details on using ABLE.
com.ibm.able.beans	The BEANS package provides a set of interfaces and objects which implement "smart" or intelligent components.
com.ibm.able.beans.filter	The FILTER package provides a set of objects implementing a template based scaling and transformation of data for use by neural networks.
com.ibm.able.beans.fuzzy	The Fuzzy System (Fs) package defines a fuzzy rule language, a fuzzy ruleset editor that you can use to create, test, and save source fuzzy rules as well as ready-to-run serialized fuzzy agents (the editor is integrated into the AbleEditor, but also can be run as a stand-alone tool), a fuzzy inference engine, and objects and APIs for creating and running fuzzy rulesets under program control.
com.ibm.able.beans.rules	The RULES package provides a set of objects implementing standard boolean-logic based forward and backward reasoning using if-then rules.
Agent Packages	
com.ibm.able.agents	The AGENTS package provides a set of interfaces and objects for constructing hybrid intelligent agents out of AbleBean components.
com.ibm.able.platform	The PLATFORM package provides a set of objects and APIs for running intelligent agents in a distributed environment.
com.ibm.able.platform.agents	FIPA agents in this package include an AMS, an ACC, and a DF, as well as a default FIPA agent that can be used as a base for building FIPA-compliant agents.
Graphical Editor	
com.ibm.able.editor	The EDITOR package provides a set of objects that implement a GUI development environment for constructing hybrid intelligent agents using AbleBeans and AbleAgents as components.

Table 1: IBM ABLE Java Packages and Editor Description – Version 1.0a

Setup

ABLE requires a recent Java Development Kit (JDK) with a swing library. I first tried to install ABLE on a Windows NT and Windows 2000 machine. I installed the Sun JDK 1.2.2 for

Windows, defined the Classpath for the JDK and ABLE in my environment, but in both cases failed to get the editor up and running. When running the installation scripts I received an error "Exception in thread 'main' java.lang.NoClassDefFoundError: com/ibm/able/editor/AbleEditor. After a lot of wasted time debugging and unanswered support email, I switched to Linux. Not having the Sun JDK for Linux, I used IBM's 1.1.8 JDK (or SDK) and download the Swing library from the web. The IBM Java SDK comes with a classes.zip package that must be included in your Classpath as well as with the swing library, Swingall.jar. This configuration worked fine.

AbleNeuralClassifierAgent

The Able environment provides all the components needed to create this simple example of data classification. We will use the AbleNeuralClassifierAgent that uses a back propagation neural network as the classifier model. Using the Able Editor, we simply drag the Agent icon into the main window. The AbleNeuralClassifierAgent is named NNClassifier Agent in Figure 3 below. The tree view on the left shows the beans that compose the Agent by default, but you are free to elaborate on the structure to introduce your own functionality to the network.

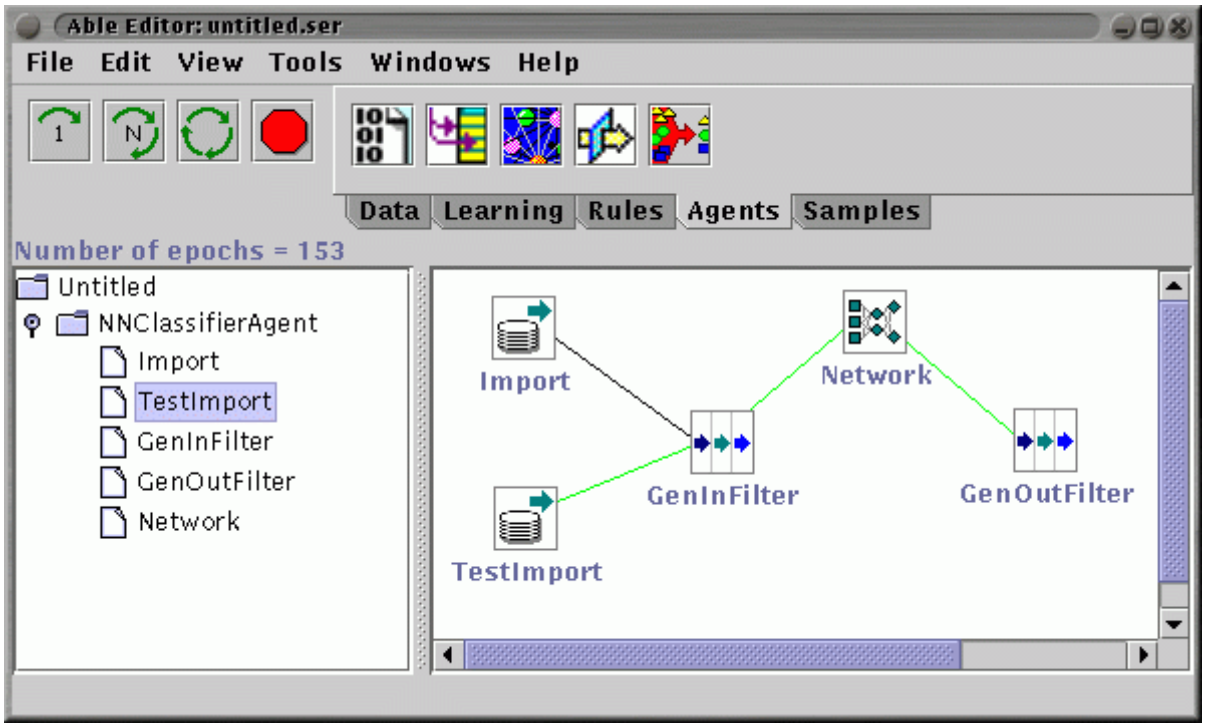

Figure3: Neural Network Classifier Agent and its Java Bean Components

By default, the classifier agent provides import beans (Import and TestImport for training and testing data, respectively) for loading data, translations filter beans, GenInFilter and GenOutFilter, which transform and scale the data for input and output to/from the neural

network. It also provides the core of the agent, the AbleBackPropagation learning bean (Network). Just to confirm the connectivity between the beans, I opened up inspection buffers on the beans to see how data translates through the network – a very cool option.

Training

The goal of training is to, of course, train the agent to understand what we are looking for. In our case, we want to teach the network what a Rule Maker company is. To begin, we need to import the data using the Import bean to load the training data. We select the Import icon and select properties to load a data definition file. The definition file gives identification to the data and defines the architecture of the network. Listing 1 provides the training definition file for our Rule Maker stock training data.

discrete	TotalSales
discrete	LastYearsTotalSales
discrete	TotalCosts
discrete	NetIncome
discrete	CashAndEquivalents
discrete	CurrentAssets
discrete	CurrentLiabilities
discrete	ShortTermDebt
discrete	LongTermDebt
continuous	SalesGrowth
continuous	GrossMargins
continuous	NetMargins
continuous	FoolishFlowRatio
categorical	Condition1
categorical	Condition2

categorical	Condition3
categorical	Condition4
categorical	Condition5
categorical	Class

Listing 1: Rule Maker Stock Training Data Definition File

The *discrete* type means integer and these are the raw values from the financial statements. The *continuous* type means a real number and these are the Rule Maker ratios we described. The *categorical* type means Boolean and these are the 'YES' or 'NO' decisions made as to if the ratio conditions are met. Finally, the *Class* variable on the definition side represents the output of the network. In our example, this variable is the approval or disapproval to list the company as a Rule Maker company.

Once this definition file is loaded, we generate the architecture that defines the network. Our definition file will create architecture of 18 input nodes and one output nodes (or units). It is our option as to create hidden layers as well for accelerated training and difficult data structures, but in for our example using just an input and output layer works just fine. This file will also reference our Rule Maker Stock Training Data listed in the appendix so that it is loaded during training.

Before we start the actual training, there are several parameters related to the back propagation model that need to be set. Although there are about 20 in all, we'll just look the most important.

- The *Learn rate* controls how much the weights are changed during each cycle. The larger the value, the greater the change.
- *Momentum* controls how much the weights are changed during a weight update by considering prior weight changes. Its main purpose is to help obtain convergence in the calculations.
- *Tolerance* defines the acceptable difference between the desired output value and the actual output value.

These values in addition to the architecture of the network are enough to start the training. I have just accepted the default values for each as shown in figure 4.

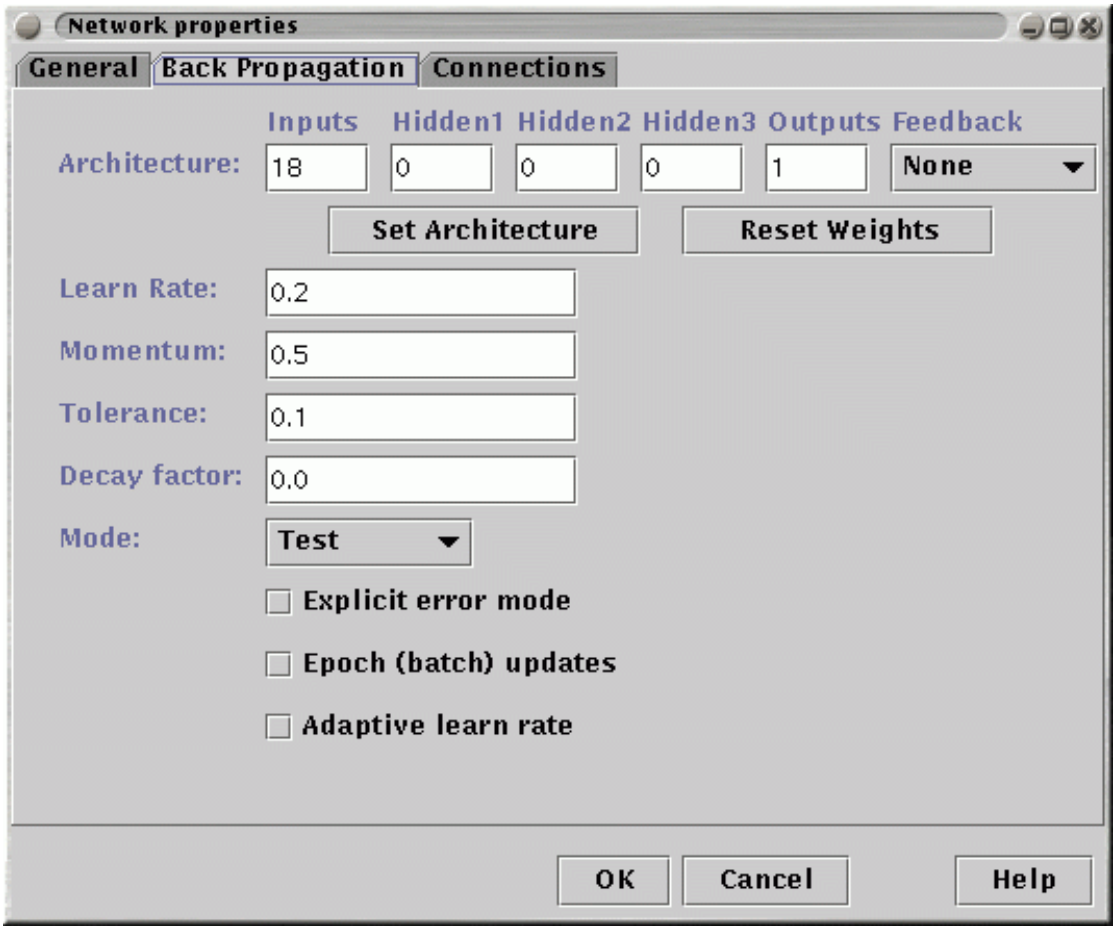

Figure 4: Back Propagation Learning Bean Configuration

There are other metrics related to the mechanics of the back propagation model and these can be used to help study the results of the training (and testing) by using the inspection options for each of the beans. Metrics such as Average Root Mean Square (RMS) Error and Bad Pattern Ratio both are metrics for helping to determine if the network is performing normally (data convergence). The Avg. RMS error is exactly that: the average of the RMS error throughout the network during an epoch. If this value does not decrease over the training cycles, the network is not converging (or learning) and the architecture or configuration parameters must be altered. Typically this is good time to introduce a hidden layer or two. The bad pattern ratio indicates the number of patterns in the previous epoch that were out of the specified tolerance for a single pattern. This helps you realize the number of patterns that are causing the network not to converge. Using other metrics, like Last RMS error, helps you to determine exactly which pattern is causing non-convergence. The Avg. RMS Error and the Bad Pattern Ratio should decrease during the training, but neither of these values should be expected to go to zero.

OK, ready for training. As Figure 3 shows, the Able Editor has a set of tools that allow us to manually control the training session. In our example, I just click the run icon that will iterate repetitively through epochs over the patterns. The key to knowing when training is finished is by the convergence of the Avg. RMS error value toward 0.1. To track these types of values and

monitor the network during operation, Able provides several java *inspection* classes. The inspector fields and methods are taken from the AbleInspector, AbleInspectorContext and AbleInspectorData classes that are part of the com.ibm.able.editor package. Figure 5 shows a plot of the Avg. RMS Error when sampling 1000 data points. This number eventually levels off to about 0.1 after 100+ epochs. You would have doubts about your network architecture if you didn't see a continuous decrease or if you see spikes (like the one in the beginning of this plot) occuring throughout. Such results would prompt reconfiguring of the back propagation network architecture or parameter modification.

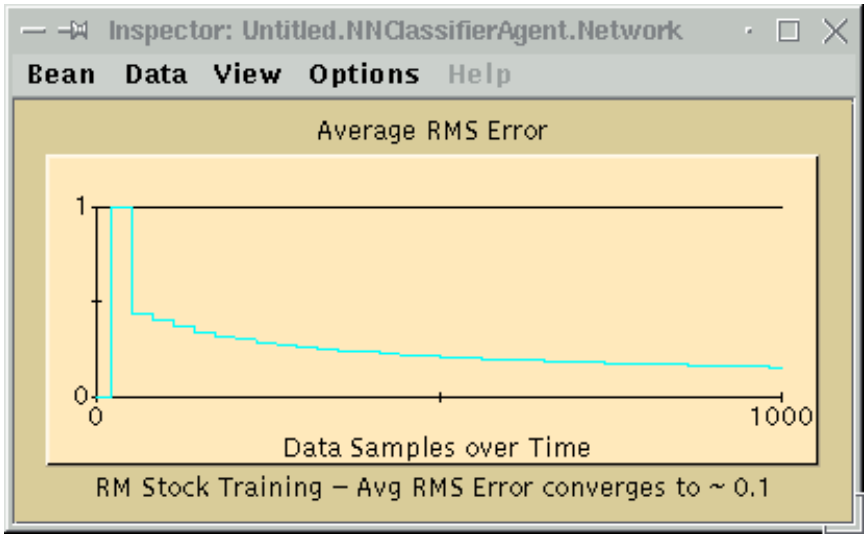

Figure 5: Average RMS Error vs. Time During Training

Understanding if the network is appropriately trained is the next step. We use the testing data afterward to confirm the network can classify the companies as we expect. We will run our test data through the network and determine how well the network can classify data it has never seen before.

Testing

Setting up for testing is similar to training. We load a definition file like the one in Listing 1 with the only difference being that we change the 'class' value to 'Approved'. This way, the network knows what value we are looking for and the specific status. We also load our Rule Maker Stock test data. A listing of this data is also in the appendix.

Our point in testing is to determine if the network is capable of classifying other sets of data it has yet seen. Therefore, our approach for this simple demo will be to analyze the results of the output bean and match up those results to those in the testing data file. If we see an approval corresponding for those stocks that are considered a Rule Maker, than the network was able to classify our data. In the same light, if a company fails the Rule Maker criteria, the network should say so. We will also pay attention to the Avg. RMS Error and Bad Pattern Ratio to get a sense for how the network handles this new data.

To start, we load our testing definition file in the TestImport bean. And we also manually load the testing data file name in the Neural Classifier Tab of the Agent itself. Why we need to load the data file during testing and not during training is beyond me, but again, this is alphaware. Then we switch the Agent Mode (from default, Train) to Test. Before we start, we want to open a few inspection windows to monitor performance and to confirm the network is successfully classifying our data.

Our testing data has nine patterns. The results of the first record are below in Figure 6. The left window shows the TestImport buffer (referenced output buffer). These values are those from the Rule Maker Test Data file for Applied Materials Inc. The window on the right is the inspection window for the GenOutFilter bean. The inputBuffer is the network output layer activation result and the outputBuffer is the decision made by the Network on if this is a Rule Maker company. The output either states 'YES' or 'Unknown'. As we would expect, Applied Materials Inc. qualifies.

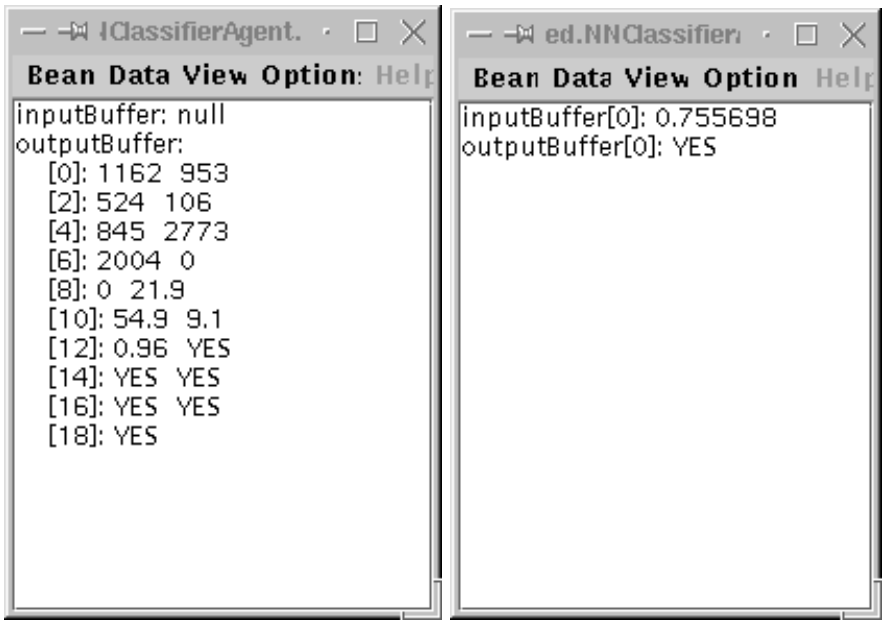

Figure 6: Inspector results for Applied Materials Inc.

The next pattern was randomly chosen just to show a negative result. The left window in Figure 7 shows the TestImport buffer for Proctor & Gamble and the right window shows their respective results.

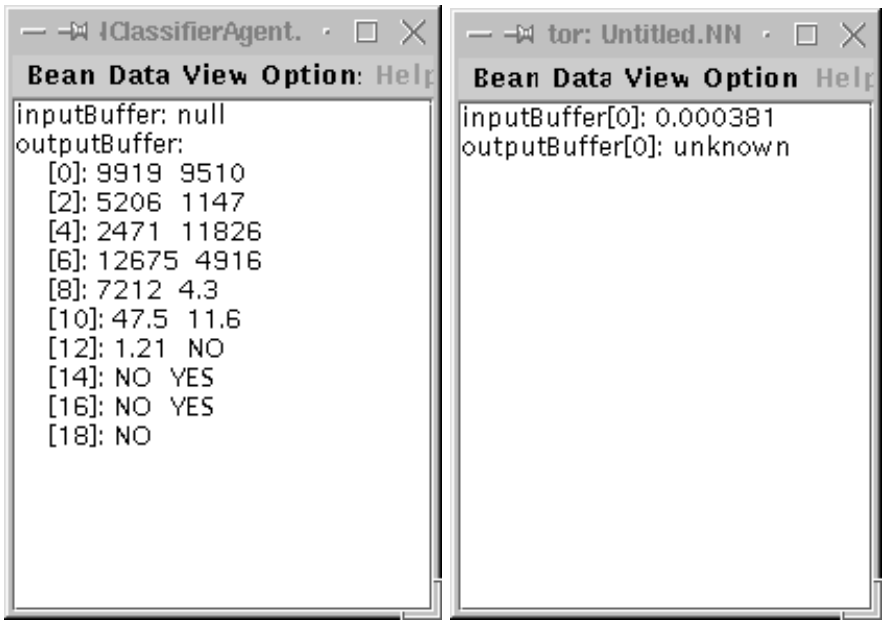

Figure 7: Inspector Results for Proctor & Gamble

Although P&G have a tremendous market share, the summation of their short-term and long-term debt is greater than their total sales! This is not a wise investment, and by not confirming YES, our network told us so.

Lastly, we'll look at SAP in Figure 8. The result confirms SAP is considered a Rule Maker company as well.

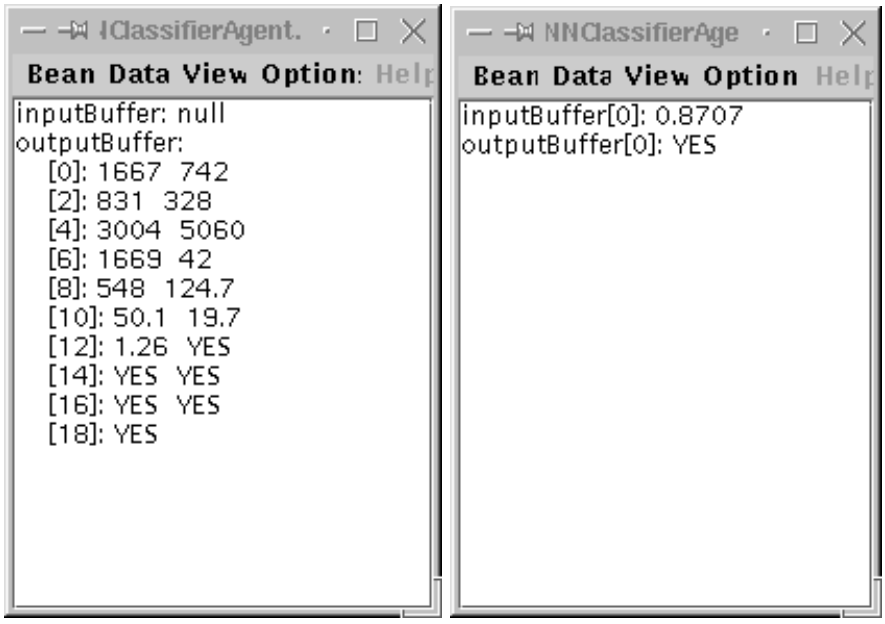

Figure 8: Inspector Results for SAP

Although not presented here, all other companies received a negative response, as we would

expect.

Lastly, we would want to confirm that these results were not just luck and we can do that by looking at some statistical metrics of the network. Figure 9 shows the Avg. RMS error and the Bad Pattern Ratio of the test epoch. The Avg. RMS Error is within the same range of the 0.1 value we saw during training the network. This means the network didn't have much trouble in making its decision. If the value was much greater, say towards 0.5, it would be telling us that it has not been trained for that type of data. In that case we would want to further train the network on more types of data. This is one of the great advantages of neural computing, it can continue to learn without re-programming new conditions!

Also, the bad pattern ratio value shows that about 25% of the patterns had errors above the tolerance we defined, 0.1. This might be higher than we would want in a real application, but the results prove the network is stable enough to solve our simple classification problem.

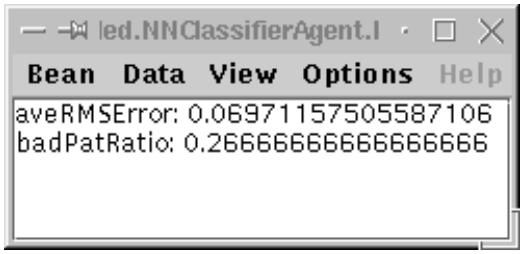

Figure 9: Avg. RMS Error & Bad Pattern Ratio for the Test Epoch

Conclusion

Overall the results were good. Our network was able to classify 100% of our data correctly. We successfully trained a network to look for certain patterns in data and once it learned what we were looking for, it was able to identify what we consider a Rule Maker company from one that does not meet the criteria.

If you made it this far and are thinking why I couldn't have just used the traditional if-else logic to determine if a company satisfies the Rule Maker criteria or not – the answer is I could have. But what happens when the data becomes more complex or perhaps you don't even know the criteria and the conditions are unknown. Neural computing gives us a way to solve these problems just by knowing the nature of the data. This concept is completely alien to conditional coding. This technique is imperative for larger sets of data like in speech or imaging or huge databases storing genetic structures. Here, we are taking a statistical approach to come up with the same results that traditional coding would give us with considerably less work.

The IBM Able package is a great tool set for getting more familiar with neural computing. As mentioned earlier, there are many existing software packages available to help you build neural programs, but this is one of the best I've seen so far using the Java platform and offering a wealth of available classes to build your own neural tools.

Appendix

Rule Maker Stock Training Portfolio Data

Company	Sales	PY-Sales	Cost	etIncome	Cash&eq	CurrAssts	CurrLia	ST-debt	LT-debt	SG	GM	NM
Cisco (Q2-00)	4919	3171	1748	1026	4653	9080	5099	0	0	55.2	64.5	20.9
Yahoo (Q1-00)	228	104	34	63	992	1072	235	0	0	119.2	85.1	27.6
Qualcomm (Q1-00)	1120	941	648	177	1391	2875	907	3	0	19	42.1	1.5
Sun Micro (Q1-00)	3553	2802	1719	353	4460	8146	3344	9	2154	26.8	51.6	9.9
Microsoft (Q2-00)	6112	5195	756	2486	17843	22020	10540	0	0	17.7	87.6	40.7
Coca-Cola (Q3-99)	5195	4747	1706	787	1637	6124	9686	5218	1108	9.4	67.2	15.1
Nike (Q2-99)	2177	2224	1365	124	128	3342	1591	557	389	-2.1	37.3	5.7
IBM (Q1-99)	20317	17618	13059	1470	5356	81751	63445	13708	16285	15.3	35.7	7.2
EMC (Q4-99)	1876	1556	843	337	1824	4320	1398	0	673	20.6	55.1	18
APC (Q1-00)	309	277	163	47	498	948	191	0	0	11.6	47.2	15.2
Oracle (Q3-00)	2449	2079	707	498	2768	5145	2809	2	305	17.8	71.1	20.3
Broadcom (Q4-99)	161	75	66	37	260	391	86	1	1	114.7	59	23
J&J (Q1-99)	6638	5783	2038	1128	2505	11618	7866	2489	1346	14.8	69.3	17
Aber. & Fitch (Q3-99)	287	230	165	39	141	251	159	0	0	24.8	42.5	13.6
Intel (Q1-00)	8021	7103	2964	2474	11216	17461	7107	373	868	12.9	63	30.8
PMC-Sierra (Q1-00)	103	50	21	23	223	297	106	2	1	106	79.6	22.3
WorldCom (Q3-99)	9186	3758	3887	1107	307	9707	15553	3679	13245	144.4	57.7	12.1
Disney (Q1-99)	6589	6339	3525	622	1194	10794	7229	2028	11077	3.9	46.5	9.4
Circuit City (Q4-98)	3403	2849	2627	85	266	2395	964	8	427	19.4	22.8	2.5
Hewlett-Packard (Q4-99)	11362	10296	8102	760	5411	21642	14321	3105	1764	10.4	28.7	6.7
CompUSA (Q1-99)	1691	1233	1475	-5	242	1340	1003	1	1	37.1	12.8	-0.3
Best Buy (Q4-98)	3458	3290	2841	110	786	2063	1347	30	31	5.1	17.8	3.2
Dell (Q3-99)	6784	5390	5414	289	5857	9709	5324	0	508	25.9	20	4.3
Lucent (Q1-99)	8220	6184	4327	457	792	17774	11562	3185	3716	32.9	47.4	5.6
3COM (Q1-99)	1410	1370	743	90	982	3585	1208	0	30	2.9	47.3	6.4

Rule Maker Stock Test Portfolio Data

Company	Sales	PY-Sales	Cost	etIncome	Cash&eq	CurrAssts	CurrLia	ST-debt	LT-debt	SG	GM	NM
Applied Material (Q1-00)	1667	742	831	328	3004	5060	1669	42	548	124.7	50.1	19.7
SAP (Q1-99)	1162	953	524	106	845	2773	2004	0	0	21.9	54.9	9.1
Proctor & Gamble (Q3-99)	9919	9510	5206	1147	2471	11826	12675	4916	7212	4.3	47.5	11.6
Office Depot (Q1-99)	2623	2399	1894	101	910	2904	1578	5	484	9.3	27.8	3.9
Merck (Q2-99)	8018	6470	4370	1478	3541	10427	6545	612	3213	23.9	45.5	18.4
McDonalds (Q2-99)	3407	3180	1985	518	408	1421	2230	2230	6203	7.1	41.7	15.2
Colgate-Palmolive (Q3-99)	2314	2265	1060	240	272	2431	2361	622	2187	2.2	54.2	10.4
Mattel (Q1-99)	692	705	375	-20	50	1789	1089	293	983	-1.8	45.8	-2.9
Mobil (Q4-98)	29139	35062	10557	1460	1461	17593	19412	4248	4530	-16.9	63.8	5

References

[1] Nigrin, Albert, *Neural Networks for Pattern Recognition*, The MIT Press, 1993, Massachusetts Institute of Technology, Cambridge, Massachusetts

[2] Bigus, Joseph P. *et al*. *IBM Agent Building and Learning Environment* (Online Documentation), IBM T.J. Watson Research Center, 2000, Armonk, NY

[3] *An Introduction to Neural Computing*, Computer Associates International, Inc. 1999, Islandia, NY

[4] Gardner, Tom, *Rule Maker Investment Portfolio* http://www.fool.com/portfolios/RuleMaker/RuleMaker.html , 1998

[5] Grossberg, Stephen, http://.cns.bu.edu, The Department of Cognitive and Neural Systems,

Boston University, Boston Massachusetts, 2000

AmazonTech White Paper Series