Predictive Model Learning of Stochastic Simulations

John Hegstrom, FSA, MAAA

Table of Contents

Table of Figures

Executive Summary

In this paper I will describe how predictive models can be constructed that will “learn” the input-output relationships of stochastic simulations. The usefulness of this technique derives from that fact that the sophisticated and computationally intensive set of stochastic simulations needs to be run only once for a given set of fixed input. Once we are satisfied that the predictive model has learned the desired relationships, the variable inputs to the more efficient and compact predictive model can be varied and output results can be generated quickly for optimization or “what-if” analysis.

I will explore the performance of an actual predictive model that learned the input-output relationships of a personal financial planning model based on stochastic simulations. For this exercise I chose to construct and use a feed-forward neural network. The neural network was successfully trained using output from a spreadsheet based financial planning model that was fed by an economic scenario generator. The trained network was then tested on a previously unseen data set to gauge its level of performance.

The trained network generates an almost instantaneous result, compared to the better part of an hour it took to run each stochastic simulation for a given set of fixed parameter value. This can make a very significant difference when making important decisions in real time.

Choice of Predictive Modeling Techniques

While there are many predictive modeling techniques available, I have chosen to use neural networks in this paper. It has been proven that, given certain conditions, neural network models are universal function approximators[1]. Essentially, a properly constructed neural network can reproduce any continuous functional mapping to any desired degree of accuracy.

Neural Network Basics

Neural networks, as the name implies, use principles analogous to the functioning of the neural structure in the human brain. There are processing units called neurons that are connected together by functional relationships similar to synapses in the brain. Information travels between the neurons as numerical values. The receiving neurons then take the values, process them and pass the output results on to other neurons.

[1] Bishop, Christopher M. (1995) Neural Networks for Pattern Recognition, p. 128. Oxford University Press.

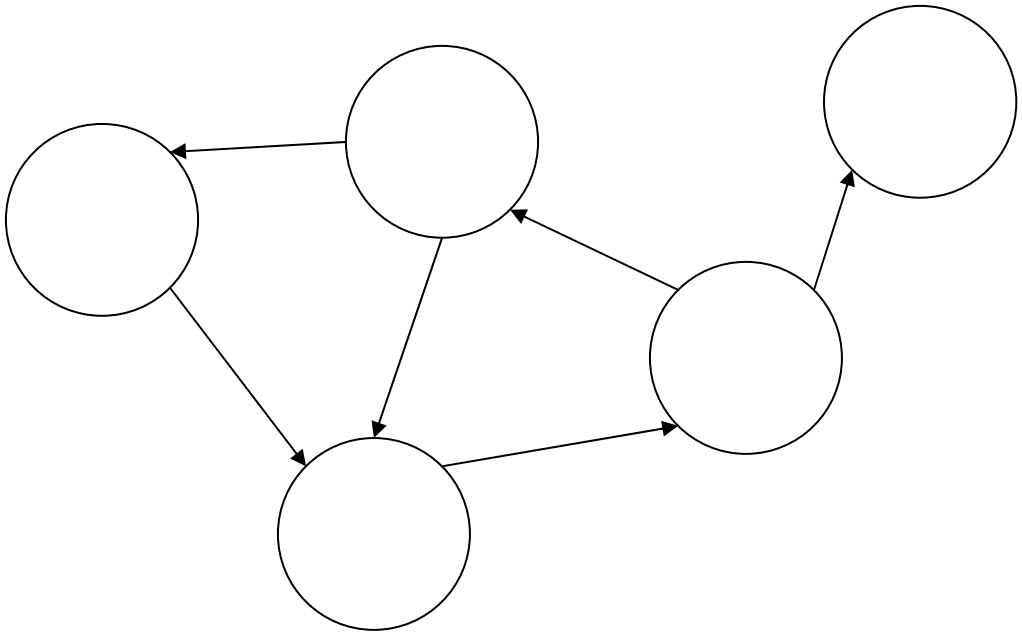

Figure 1. Generalized Neural Network

While generally information can flow in any direction within a neural network, it is more convenient mathematically to have the information flow in one direction. This type of network is called a feed-forward neural network.

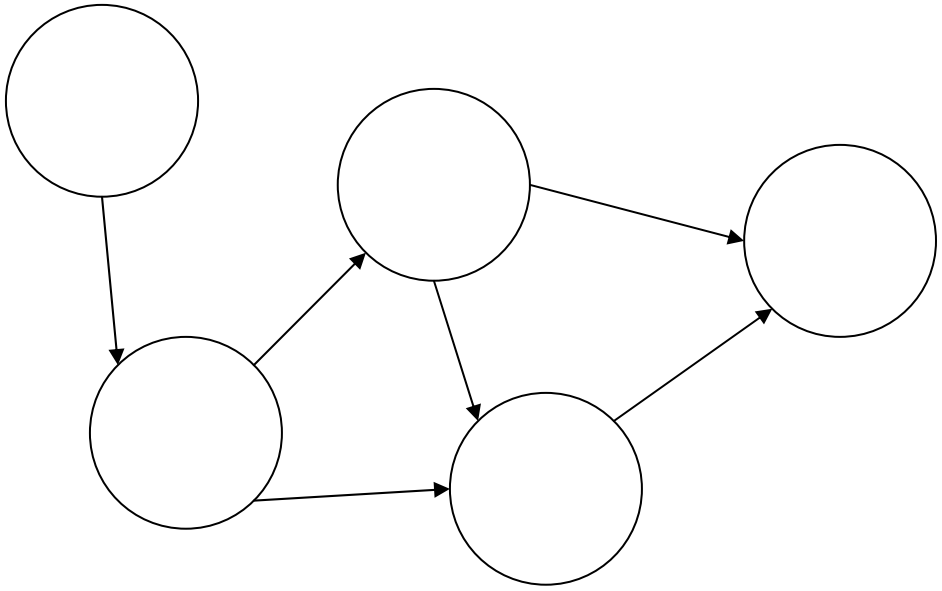

Figure 2. Feedforward Neural Network

The most commonly used type of feed-forward neural network is called a multi-layer perceptron (MLP). In this type of network, each non-input neuron (or unit) is fed into by output from all of the units in the previous level. The units that are not input or output units are called *hidden* units. There can be one or more layers of hidden units.

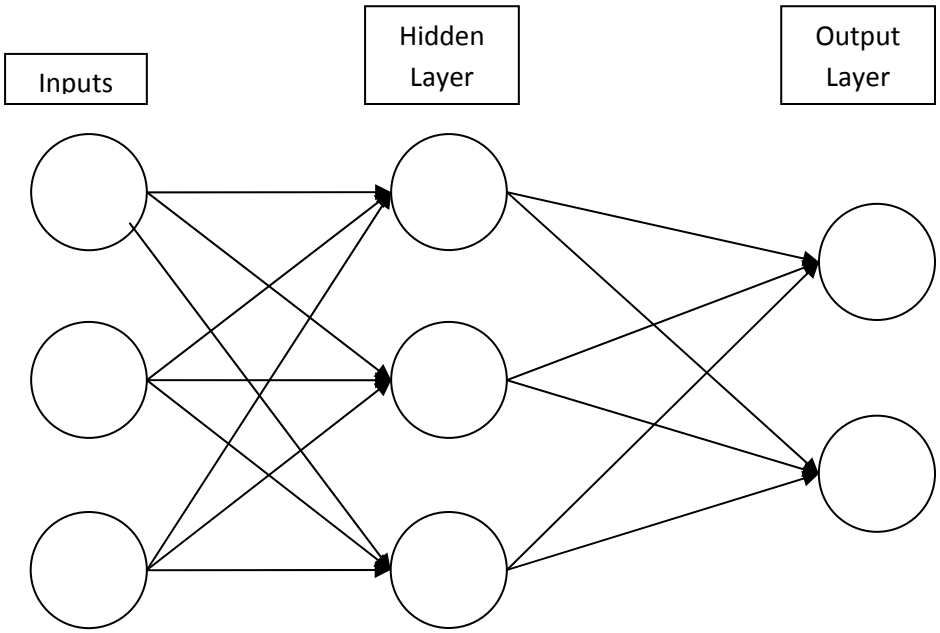

Figure 3 – Multi-Layer Perceptron

The output of hidden unit *j* in the first hidden layer can be described as $a_j = \sum_{i=0}^{d} w_{ji}^{(1)} x_i$

where *d* is the number of input units, $w_{ji}^{(1)}$ is the weight in the first layer for the signal from input *i* to hidden unit *j* and x_i is the input value from input *i*. We will set x_0= 1 so that we have what is called a *bias* unit.

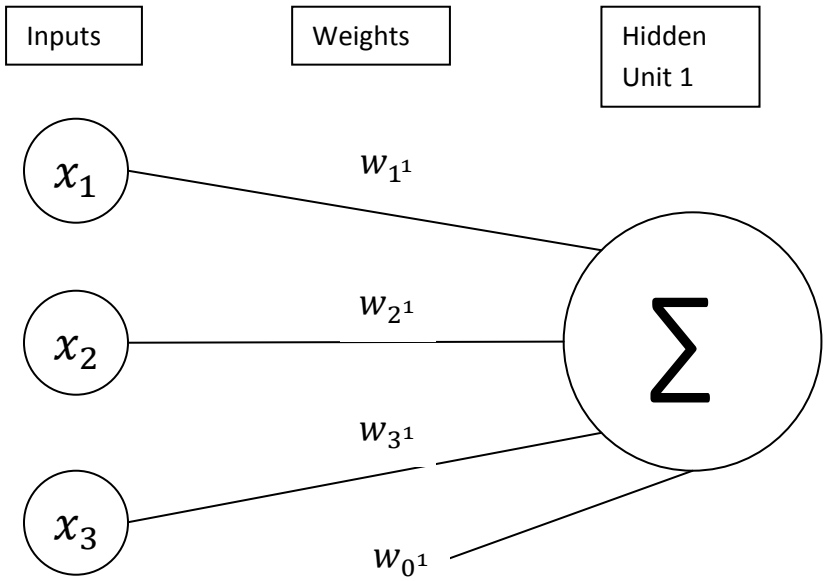

Figure 4. Example of Hidden Unit Processing

With any neural network architecture, processing takes place in each non-input unit after all of the inputs to that unit are weighted and summed together. This processing must ultimately be of a non-linear nature to allow the network to learn non-linear functional relationships. Therefore, we will transform the linear sum a_j using a non-linear *activation function* that we will call *g(x)*. The activation output is defined as $z = g(a)$. The functional form of *g* can vary based on preference, but often is based on the *logistic sigmoid* function where

$$g(a) = \frac{1}{1 + e^{-a}}$$

(see Figure 5)

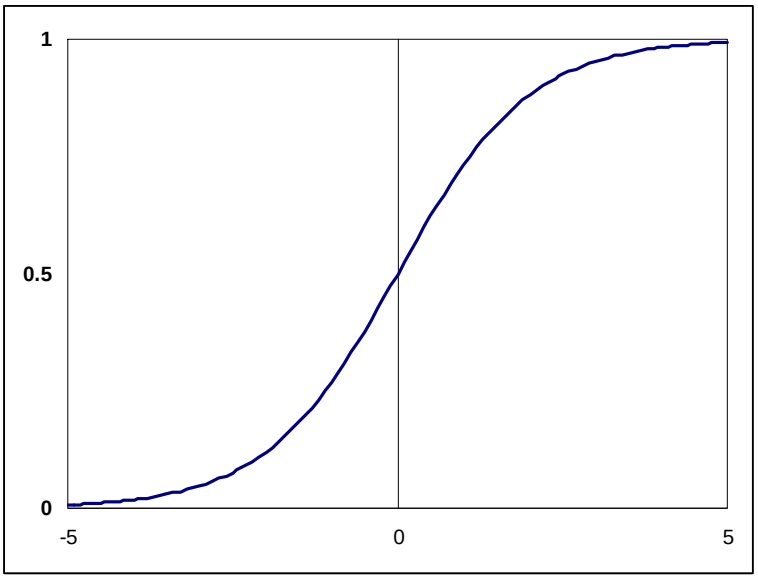

Figure 5. Logisitic Sigmoid Function

The output of *g(a)* is then passed to units in the output layer, or alternatively another hidden layer. In the output layer, the inputs are weighted, summed and transformed just like in the hidden layers. The activation function at the output stage, *h(x),* is often linear but can vary based on the range and nature of desired outputs.

The equation for the output of this network with one hidden layer, for the *kth* output, is:

$$y_k = h\left(\sum_{j=0}^{M} w_{kj}^{(2)}\, g\left(\sum_{i=0}^{d} w_{ji}^{(1)} x_i\right)\right)$$

Once the network architecture has been determined and the data scrubbed and pre-processed as necessary, the next step is to present the network with training examples so that the network weights can be optimized. Optimizing the weights involves minimizing an error function, typically the sum of squared error. The standard sum of squared error function for

training pattern *n* (where *c* is the number of output units and *t* is a training pattern output) is given by:

$$e_n = \frac{1}{2}\sum_{k=1}^{c}(y_k - t_k)^2$$

The aggregated error for the neural network is summed over all of the individual training examples and is:

$$E = \sum_n e_n$$

Optimizing the weights to minimize the error function is not a trivial process due to the complexity and non-linearity of the neural network equations. A process called *back-propagation* was developed. First a random set of weights is computed for the network. The network is then presented with the training examples and the error between the example output and neural net output is computed and propagated backwards through the network and the weights are adjusted to reduce the error. This is done in successive iterations until desired criteria for convergence are met. In this way, the neural network "learns" the underlying functional relationship. For a mathematical treatment of back-propagation, the reader is directed to Bishop[2].

While back-propagation itself is a standard method, the actual determination of the changes in weights can vary based on various non-linear optimization algorithms. The most common method is called the *delta rule* or *gradient descent*. Other methods, that converge faster under

[2] (Bishop) p 140-148.

certain conditions, include *delta-bar-delta, conjugate gradient* and *Levenberg-Marquardt.* Levenberg-Marquardt is used in the example in this paper as it converges very quickly for networks with relatively small amounts of weights.

Financial Planning Model

A crude personal financial planning model was developed in Microsoft Excel. It has the following inputs:

1. Current dollar amount of assets, $Assets_0$.
2. Current annual income, $Income_1$.
3. Current annual expenses, as a ratio to income, *exprat*.
4. Current and planned retirement age, age_c and age_r.
5. Final age for calculation, a_f.
6. Tax rate, *txrt*, which does not vary by age or amount or type of income.
7. Current and retirement asset allocation between cash, bond, small company stocks, and large company stocks. These are denoted as $a_{cc}, a_{cb}, a_{cs}, a_{cl}, a_{rc}, a_{rb}, a_{rs}, a_{rl}$.
8. Short and intermediate Interest rates, large company and small company equity total returns, and nominal inflation , $i_t^s, i_t^i, r_t^l, r_t^s, inf_t$.

The model produces year by year asset balances based on the inputs given:

$$Assets_t =$$

$$Assets_{t-1} + (1 - txrt)[Income_t - Expense_t] + (1 - txrt)[Assets_{t-1} + (Income_t - Expense_t)/2][a_{cc} i_t^s + a_b\left(i_t^i + cg_t^i\right) + a_{cs} r_t^s + a_{cl} r_t^l]$$

Where

$$Income_t = (Income_0)\prod_{i=1}^{t}(1 + inf_i) \qquad t = 1, \dots, age_r$$

$$0 \qquad t > age_r$$

$$Expense_t = (exprat)(Income_0)\prod_{i=1}^{t}(1 + inf_i)$$

And

$$cg_t^i = 7\left(i_{t-1}^i - i_t^i\right)$$

where 7 is the assumed effective duration of an intermediate term bond. While the *cg* term is a rough approximation, it will suffice for the purpose of this example.

Economic Scenarios

Economic scenarios are necessary in order to supply input to the financial planning model. While any real-world economic scenario generator could be used, the outputs must include the term structure of interest rates, equity returns for various equity classes, and inflation rates. The generator should be calibrated to produce the range and volatility of results that we expect into the future.

For the example in this paper I have chosen a generator developed by researchers Kevin C. Ahlgrim, Stephen P. D'Arcy, and Richard W. Gorvett under the direction of the SOA's Committee

on Finance Research[3] in July 2004. Details of the model and its development can be found on the SOA website (http://www.soa.org/research/finance/research-modeling-of-economic-series-coordinated-with-interest-rate-scenarios.aspx). The parameter values used in the model are illustrative and were not calibrated to the current environment. The parameter settings are described in Appendix A.

This generator was implemented in Microsoft Excel using the @RISK add-in produced by Palisade Corporation.

Stochastic Simulations

For a given simulation, certain variables are held constant and not subject to change. These are current age, final age, current value of assets, and tax rate. We will call these the *constants* and they are used in the financial planning model. These are variables that will not vary under the control of the subject and are assumed to be otherwise unpredictable. The inclusion of tax rate in this list is somewhat arbitrary.

Other variables under the control of the subject are randomly varied so that the neural network will have a range of training values to learn from. These variables are retirement age, expense to income ratio, and current and retirement asset allocation percentages. *N* random sets of these variables are calculated. We will call these the *user variables* and they are also used in the financial planning model.

Now a set of *M* equally probable economic scenarios are generated and are in put into the financial planning model one-by-one. We then calculate *p,* which represents the probability

[3] (Ahlgrim, D'Arcy, & Gorvett, 2004)

that assets always stay greater than zero prior the final age. We calculate *p* over *M* economic scenarios for a given set of user variable values (set *k* out of *N*) that were generated

$$p_k = \frac{\sum_{j=1}^{M} I_j}{M} \quad k = 1, \ldots, N$$

where I_j is the indicator variable that is set to 1 if assets never go below zero in scenario *j*, 0 otherwise.

Specifying and Training the Neural Network

The following *N* training cases, generated by the stochastic simulations, will be fed into the neural network.

One desired variable (output): p_k the probability that assets last until the final age

Ten input variables:

- Planned retirement age
- Expense ratio
- Current and retirement asset allocation percentages for cash, bond, small company stocks, and large company stocks.

The task for this neural network is to predict p_k, the probability that assets last until the final age, given the input user variables. Each of the *N* training cases for the neural network therefore contains a set of randomly generated user variables as input, and the corresponding value of p_k produced by the financial planning model as desired output.

There is a dizzying array of options to choose from when specifying a neural network. As mentioned previously, a multilevel perceptron architecture was used for this example. However, a decision still needed to be made about the number of hidden layers and hidden units. The guidance on choosing the number of hidden layers is that one layer is usually sufficient[4]. The choice of the number of hidden units to use in each layer is a more crucial matter. Too few units and the network will under-fit the data and will not perform well. Using too many units relative to the number of training cases will cause the network to over-fit the data and it will not perform well on unseen data. Figures 6-9 illustrate this concept using a polynomial function sampled with added noise and then fitted with a polynomial. A neural network works similarly but over *n*-dimensional hyperspace which can be imagined as a higher order version of the two-dimensional curve fits shown. If there are too many free parameters the over-fitting occurs, if there are too few under-fitting will occur. The number of units and thus weights in the model will affect the proper degree of fit.

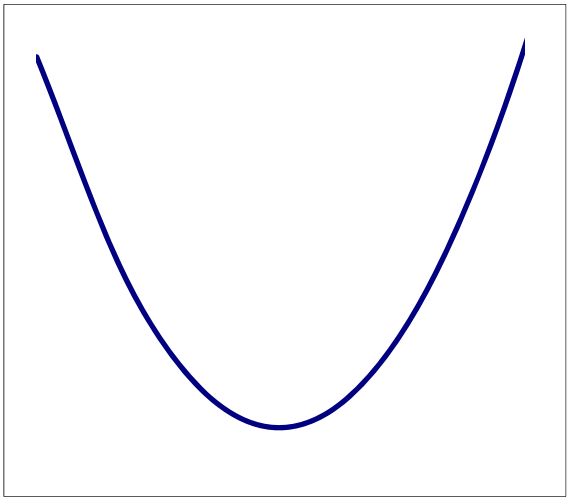

Figure 6 - Underlying function

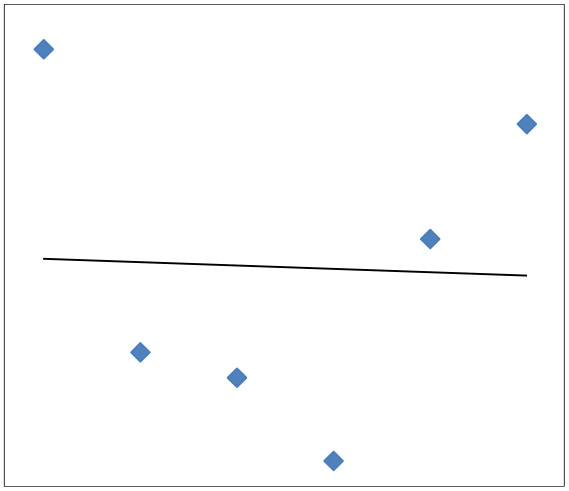

Figure 7 - Under-fitting sampled data

[4] (Gurney, 1997) p. 73-76

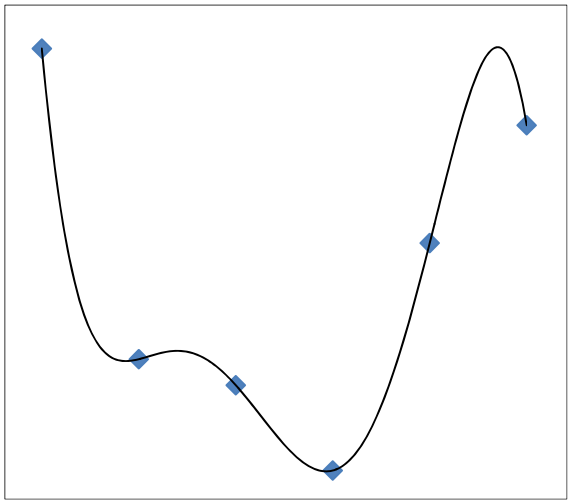

Figure 8 - Over-fitting sampled data

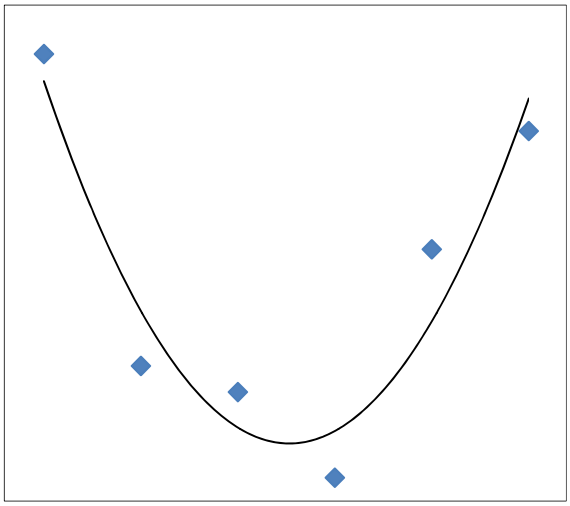

Figure 9 – Proper fitting of sampled data

In order to choose the number of hidden units for this neural network, a test was performed whereby the network was trained on 800 points of preliminary data with varying numbers of hidden units. The error was then computed on the training set and on an unseen 200-point *cross-validation* set of data to gauge the effect of adding more hidden units.

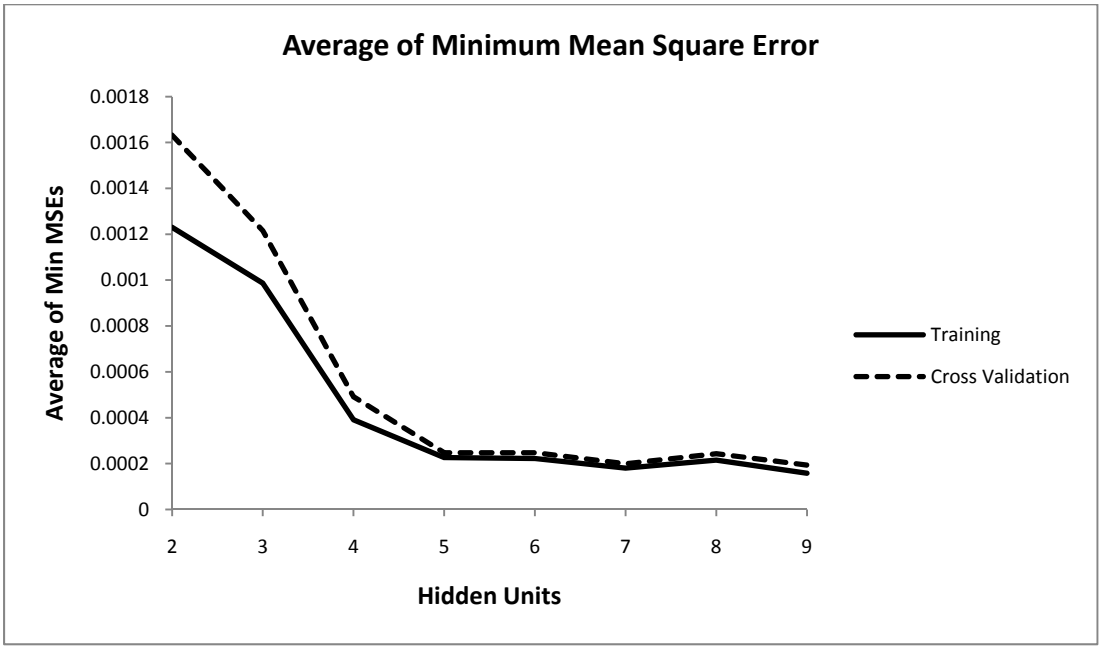

Figure 6 - Error for different numbers of hidden units

Based on the performance of the tests, it was decided to use five hidden units for the neural network, as there was a significantly diminished return associated with additional hidden units.

Neural Net Results Compared to Stochastic Simulation

A set of data points, using *N* = 2000 and *M* = 500, was generated using the financial planning model and economic scenario generator. The current age and final age were fixed at 35 and 85, respectively. The current amount of assets was set to $250,000, the annual pre-tax income to $150,000, and the tax rate was set to 30 percent. The resulting data was subdivided into a training set of 1700 points and a test set of 300 points. The neural network was constructed using the Neurosolutions software package, version 5, produced by NeuroDimension, Inc. The network was first trained using the training set. Then the network was run using the previously unseen test data and the network output of p_k was compared to the desired p_k. The scatter plot (Figure 7) shows the relationship between predicted and actual p_k. The results indicate proper fitting of data. A tighter correlation could be achieved by increasing *N* and *M* at the expense of CPU processing time.

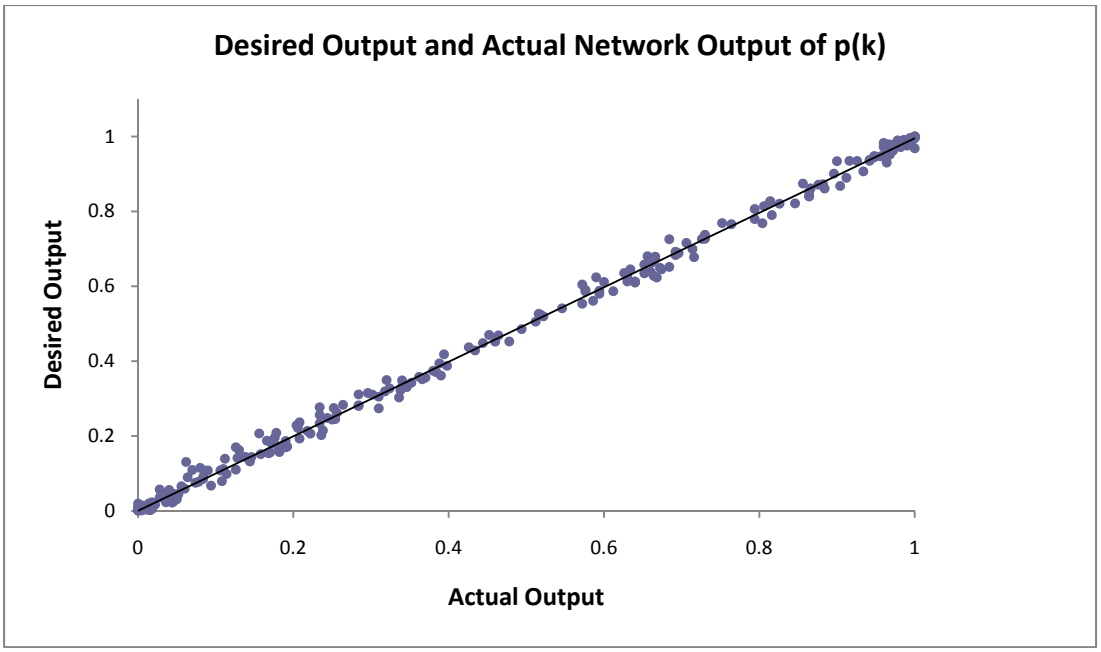

Figure 7 – Comparison of output from stochastic simulation and neural network

The advantage is that the neural network, once trained, can produce a relatively accurate answer in a fraction of a second. The stochastic simulation model, however, can take a great deal of processing time between queries. This is especially true as the models become more complex. In the context of personal financial planning, it is strongly desirable that the system be able to respond to what-if queries within a reasonable time frame. Even this basic stochastic model required several minutes to complete but the neural network can be evaluated in less than 1/10 of a second It is not uncommon to develop significantly more complex stochastic models that take hours, days or even longer to solve, making it impractical to run various "what-if" analysis. Once a valid predictive network has been developed and trained scenarios can be tested in seconds instead of days. This computational leverage opens a whole new frontier in financial modeling.

In our example two graphs (Figures 8 and 9) have been easily obtained from the fully trained neural network. They exhibit the relationships, for this hypothetical individual, between long-term financial security, retirement age, and expenses.

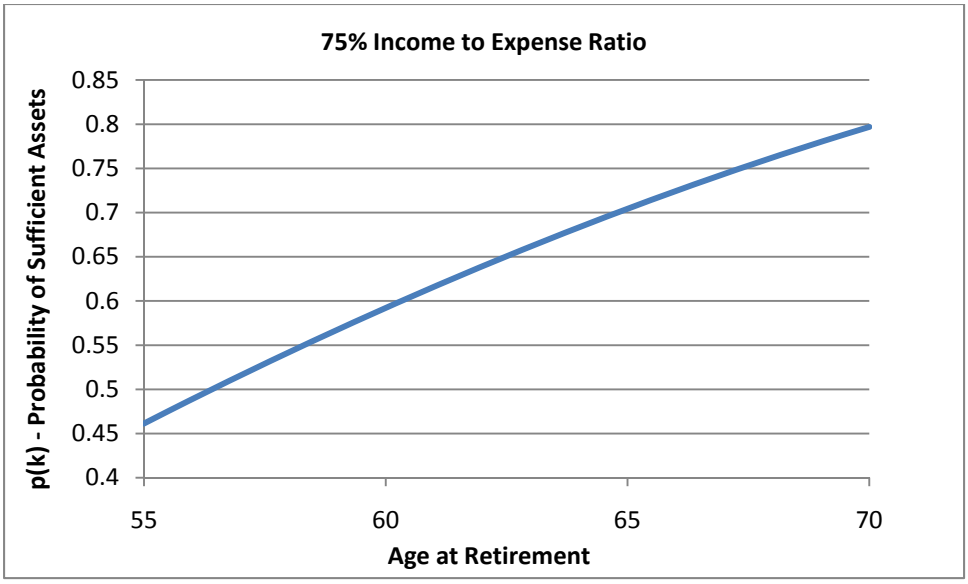

Figure 8 - Relationship between retirement age and financial security

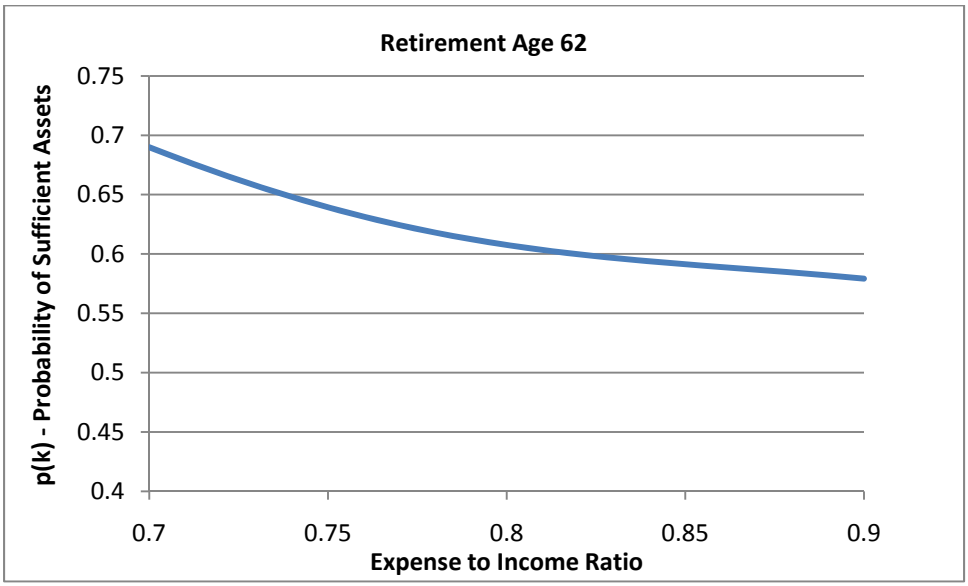

Figure 9 - Relationship between expenses and financial security

Caveats of Using Predictive Models to Learn Stochastic Simulations

Predictive models in general are very good at interpolation; that is they predict well when the input data lay in the regions of hyperspace that have been explored by the training cases. Predictive models, including neural networks, are not quite as good at extrapolation. Care must be taken when using inputs that are outside of the range that was used for training cases. If stress testing is a desired use of the neural network, then training cases should be developed that cover extreme scenarios.

One of the drawbacks of neural networks is that they are relatively opaque as a modeling tool. It is difficult to look “inside” of a neural net to see what it is doing. There are some tools and techniques for exploring sensitivities to changes in input, but in general neural nets have the feel of a “black box” to critical observers.

Conclusion

We have shown that it is possible for a predictive model to learn the functional relationship underlying a stochastic simulation to a desired degree of accuracy.

Personal financial planning is just one possible use of this concept. Business financial planning and risk management also require fast turnarounds to “what-if” queries. One promising application is in the area of searching for optimum investment strategies or allocations that meet risk constraints. A neural network, once properly trained, can be quickly searched for optimal solutions or strategies.

Bibliography

Ahlgrim, K. C., D'Arcy, S. P., & Gorvett, R. W. (2004, July). *SOA - Modeling of Economic Series Coordinated with Interest Rate Scenarios*. Retrieved from SOA - Society of Actuaries: http://www.soa.org/research/finance/research-modeling-of-economic-series-coordinated-with-interest-rate-scenarios.aspx (Date website was last accessed was 2/1/2010.)

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition.* Oxford: Oxford University Press.

Gurney, K. (1997). *An Introduction to Neural Networks.* London: UCL Press Limited.

Kautt, G. G. (2001). *Stochastic Modeling: The New Way to Predict Your Financial Future.* Fairfax, VA: Monitor Publishing Company.

Appendix A – Economic Scenario Generator Parameters

TERM STRUCTURE MODEL - TWO-FACTOR VASICEK

	Required Input - Current Market Conditions
0.01	Level of inflation (.01 = 1%)
0.01	Current 3-mo T-bill rate
0.015	Current 1-yr T-bond yield
0.033	Current 2-yr T-bond yield
0.04	Current 5-yr T-bond yield
0.05	Current 10-yr T-bond yield (round UP to the nearest 1/2%)
0.055	Current 30-yr T-bond yield (round UP to the nearest 1/2%)

Model Parameters

	Real interest (r)
1	rk1 - mean reversion speed for short rate process
0.01	rs1 - volatility of short rate process
0.1	rk2 - mean reversion speed for long rate process
0.0165	rs2 - volatility of long rate process
0.028	rm2 - long-term mean reversion level for long rate
-0.05	rlow - lower bound for short-term interest
0	rinit1 - initial short-term real interest rate
0.007	rinit2 - initial mean reversion level for real interest rate
0.5	rcorr - correlation between long and short processes

	Inflation (q)
0.4	qk1 - mean reversion speed for inflation process
0.04	qs1 - volatility of inflation process
0.048	qm2 - long-term mean reversion level for inflation (per time step)
-0.02	qlow - lower bound for short-rate inflation
0.01	qinit1 - initial inflation level
-0.3	qcorr - correlation between inflation and (short factor) real interest

Negative Nominal interest rates not allowed

Appendix A (continued)

EQUITY MODEL - REGIME SWITCHING

Large Stocks

0.008	emu0 - mean equity excess return in state 0 (one month)
0.039	evol0 - volatility of equity return in state 0 (one month)
-0.011	emu1 - mean equity excess return in state 1 (one month)
0.113	evol1 - volatility of equity return in state 1 (one month)

Regime Switching Transition Probabilities

State	No Switch	Switch
0	0.989	0.011
1	0.941	0.059

Small Stocks

0.01	emus0 - mean equity excess return in state 0 (one month)
0.052	evols0 - volatility of equity return in state 0 (one month)
0.003	emus1 - mean equity excess return in state 1 (one month)
0.166	evols1 - volatility of equity return in state 1 (one month)

Regime Switching Transition Probabilities

State	No Switch	Switch
0	0.976	0.024
1	0.9	0.1

0.9	ereg - correlation between large and small regime shifts
0.95	ecorr - correlation between large and small excess returns

EQUITY DIVIDENDS

NOTE: These parameters may not be immediately interpreted since the process is based on LN of yields.

0	dyk - dividend (log) yield reversion speed
0	dym - dividend (log) yield reversion level
0.13	dys - dividend (log) yield volatility
0.015	dyinit - initial dividend yield (NOT log)
-0.25	dcorr - correlation of (log) dividend yield with stock returns

Appendix B – Neural Network Parameters for Sample Model

Training cases:

Retirement Age: 55 – 70

Expense to Income Ratio: 70% - 95%

Allocation percentages: 0% to 100% in increments of 10% for each class

W_{ij} = weight matrix applied to normalized inputs

i \ j	1	2	3	4	5
1	-0.2547	-0.3150	0.1305	0.4070	0.5264
2	0.5482	-0.7488	-0.6709	0.8062	-0.6836
3	-0.3466	-0.2923	-0.6386	-0.5539	-2.1562
4	-2.3312	1.7418	1.7529	-0.4271	-0.1517
5	-0.9058	-1.0541	0.7204	0.3137	-0.6866
6	-0.6592	0.7587	0.7168	-1.0645	0.4449
7	-0.1425	-0.1894	0.2932	0.0039	-1.1187
8	-1.1593	0.7800	0.7572	-0.9473	0.8244
9	0.6436	0.7992	-0.5849	-0.2025	0.8737
10	0.8819	-0.8602	-0.7946	0.9611	-0.7109

W_{j1} = weight matrix applied to hidden units to compute output

j	1	2	3	4	5
	-1.1590	2.1396	-2.0441	-1.4649	-3.5084