SM03

An Animated Guide: An Introduction to Neural Nets

By Russ Lavery (contractor for Numeric Resources Ltd.)

Abstract

Artificial Neural Networks (ANNs) are a machine learning technique that is modeled on how cells in our brains learn. It is a supervised learning algorithm (we must have a sample data set where we know the values of Y) and can be used to build models where Y is discrete or continuous. This paper is an introduction to the underlying concepts, vocabulary and application of Artificial Neural Networks in SAS® Enterprise Miner and JMP®.

Introduction:

The name Artificial Neural Network is appropriate because, like our brains, it is made up of highly connected nodes and because the underlying process of an ANN uses an algorithm that learns by making adjustments to those connections.

To the right is a rough picture of a Neuron or nerve cell.

It has input paths called dendrites that take information, from the environment or other cells, and sends a signal to the cell body (SOMA).

Inside the cell body, the inputs are summed and, in a part of the cell called the Axon Hillock, the sum of the inputs is compared to a threshold value. It should be noted that some inputs increase the chance of the axon firing and some inputs inhibit the chance of the axon firing.

If the sum of the inputs exceeds a threshold value, the cell fires a signal down the Axon to other cells.

The signal down the Axon is electrical but the signal is transferred to the downstream cells (the signal is transferred across the synapse) via a chemical process.

Figure 1
To right we see how the early machine learning researchers conceptualized an artificial cell that was capable of learning.

The decision making node is called a perceptron.

It takes values for one observation (a row of data) from a data set, creates a weighted sum of the input values then and compares the weighted sum to a threshold. Weights are represented by $W_0$, $W_1$, $W_2$ and $W_3$.

If the weighted summed value is above the threshold the “perceptron fires” and classifies the observation as 1. If it fails to fire it, in effect, classifies the observation as a 0.

![Example of a Perceptron](image)

The history of Neural Networks, and other machine learning processes, is complex. In the late 1940’s Donald Olding Hebb hypothesized an organic learning process by saying “When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some process of metabolic change takes place in one or both cells”. The commonplace phrasing for the Hebb’s Hypothesis is “Cells that fire together, wire together”. Hebb was saying that learning happens as a result of changes in the metabolism of cells in response to the environment. In Figure 2 we see an electronic equivalent of the Hebb’s Hypothesis. Exposing the algorithm to a training data set (each row containing X values and a Y - with known values for the Y) and letting the algorithm determine the optimum values of the weights $W_0$, $W_1$, $W_2$ and $W_3$ is called “training the network”. Adjusting the weights to minimize error (remember that the data set has values for Y in it, so we can compute errors) is how Neural Networks learn. Adjusting the weights is the equivalent to changes in the metabolism of cells.

Research was done on machine learning in the 1950s and 1960s (Vapnick published his paper on Support Vector Machines in 1963 – SVMs are experimental in SAS Enterprise Miner 9.2). In 1969 Papert and Minsk published a book that caused research “interest in machine learning to drop precipitously”. Papert and Minsk were able to prove that a single cell (AKA a single perceptron) could not learn the logical XOR function. This inability to learn the XOR was seen to be a serious, and general, impediment to machines learning by themselves and scholarly interest shifted to other areas.
Later researchers were able to demonstrate that a multi-level/multi cell network could learn the XOR and a graphic of three cells learning the XOR is presented at right.


His algorithm adjusted connections (weights – shown as Ws in graphic at right) between the cells/nodes/perceptions in a manner that mimicked Hebbe.

This adjusting algorithm, and new computing power, created a practical and powerful analytic tool.

### Common Activation functions

While cells, and early research, used step functions (Figure 2) as the activation function, researchers soon abandoned the step function.

The slope of a step function is either zero or infinity and this limits the number of "mathematical tools" that can be applies to the problem of minimizing or maximizing.

At right are some common activation functions (a picture of a mound shaped function will be shown later).

The most commonly used functions are: Linear, Hyperbolic Tangent and Gaussian/Radial Basis function (or mound shaped).
Heuristics drive many decisions in ANN projects and a practice has developed combines different activation functions in multi-layer ANNs.

In a multi-layer network, a linear activation function is sometimes used as a data reduction trick in the layer closest to the input variables.

The linear activation function has an effect of reducing the number of input variables – a bit like performing a multi-variable regression on the X variables at the start of the ANN.

A hyperbolic tangent has a shape similar to a logistic function and both hyperbolic and logistic functions have several interesting properties when applied in an ANN algorithm.

They can be considered a hybrid of step and linear functions because they are, like a step function, insensitive to very large or small values and linear in a range.

Consider the logistic curve in Figure 5. Small values of X map to similar small values of Y. Large values of X map to similar large values of Y. However, in the center of the curve, the response is sensitive to small differences in X.

The Gaussian activation function is useful when there is an ideal point.

Frogs have two neural networks in their brain that help them decide if something is a fly and good to eat.

One network compares the “possible meal” to an ideal fly size

and the other network compares the objects motion to the motion of an ideal fly.

The object being evaluated can be: be too large or two small to be “lunch” or not moving correctly to be “lunch”

The radial basis function curve handles this well because, like the hyperbolic tangent, the RBF is insensitive to very large or very small values.

However, the value of Y, returned by levels of X, is sensitive to X when X is close to “ideal”.

While a Cardinal and a Moose are very different in size, using the RBF as an activation function makes the decision easy. Both animals are so far from the ideal size that they are (for all practical purposes) assigned the same Y value and immediately classified as a “thing NOT to eat”. They both get very similar Y measures on the characteristic “SIZE NOT AT ALL LIKELY TO BE LUNCH”. The motion evaluating neural network works in a similar manner. A thing may be too still to be lunch or it may be too active to be lunch. The results of these two paths are combined before a decision is made.
Specifying a Neural Net

An analyst will start out by specifying the X and Y variables. Figure 7 was created using JMP and specifies:
- The Y and X variables
- The structure of the model:
  - Two layers, with 2 nodes in each layer
  - Linear activation in nodes near the variables
  - Hyperbolic tangent activation in nodes near Y

Like regression, and other modeling techniques, an ANN model will perform better on the data that was used to train it than on new data.

If the ANN is sufficiently complex (many levels and nodes) it can memorize the training data set and predict the training data set with 100% accuracy.

However, the goal is to create a model that predicts well on new data. Generally an analyst will create a holdout sample and measure the predictive power of his/her model on the holdout sample. This option can easily be selected in JMP (see boxes above the red square).

Evaluating performance using a validation sample forces you to split the data set and use one part of the data set to build a model and the other part of the data set to evaluate the model. Basically, the model built on the training sample is applied to the “hold out sample”. The model created using the training data, and X values in the “holdout data set” are used to predict the Y values in the holdout data set. This technique of holding out some of the data for testing can be implemented in several different ways.

JMP Pro allows you to select either “create a holdout data set” or “use a K-fold holdout”. If you select holdout, JMP will remove a randomly selected proportion of the original rows to be used in the “predict new observations phase”.

If you select K fold, you will select an integer K and cause a more complex process to occur. If we specify k=5, JMP divides the data into 5 parts. It uses 4 of the parts to develop a model and the remaining fifth as validation data. The process repeats K times, with each of the K samples being used one time as validation data.

The hidden layer structure section of the dialog box (see red square) allows you to specify a one, or two, layer Neural network, the number of nodes in each layer and the activation functions to be applied in each layer.

The Fitting Options section, below the red square, allow the user to specify some options that are not closely related. Transform Covariates applies a John DB or Johnson SU transform that will reduce the effects of outliers. The penalty method allows the user to specify several types of error penalty.

There are some hints as to which penalty method to select.

Squared penalty: Useful if an analyst expects all variables to be important.

Absolute (Value) penalty and Weight Decay penalty are useful if the analyst has many variables, and expects a few to be important and to have many variables to be unimportant. Selecting no penalty is useful if the model is large and run time is an issue but is generally considered to be the least “accurate” method.

The number of tours is a way of coping with the issue of local minimums. Neural Networks usually have many local error minimum and one method of handling this is to do many runs (John Sall recommends 20 if you have time) with different starting points for the search (different initial weights). If the 20 runs produce a consensus, then you have found “truth”.
Boosting “boosts the importance of hard to predict points” by running a series of ANNs.

The first ANN predicts Y.

The second ANN uses the same X values but predicts the errors in the first model.

A third ANN, if specified, predicts the errors of the second model.

The models from each step are combined into a final model.

If you Boost two times and:
- you specify a two layer ANN with 3 nodes close to the variables and 2 nodes near the Y
- then you will get a final model with 6 nodes near the variables and four nodes near Y.

Figure 8

Specifying a Neural Net: Fitting Options: Boosting
(Boosting the importance of hard to predict obs.)

A Neural Net Example

Imagine that we want to model the weight of subjects in SAShelp.class. We want to use height and age (as X variables) to predict weight (our Y), and ask for a 2 layer 2 node ANN.

The red Ws, in the graphic in the right part of Figure 9, stand for weights and the ANN will “learn” how to set them to “proper” values so that the error in predicting Y is small. Each weight can, and likely does, have a different value.

Calculating Y, when weights are known, is conceptually simple. Variables are multiplied by their weights and the results are carried to the nodes to the right. The weighted values are summed, in the nodes and the activation function is applied producing a number that is the output of that node. The output is sent to nodes to the right where the process is repeated.

Figure 9

Neural Diagrams Example

The learning of proper values for the Ws is done through a method called “Back Propagation of Errors”. There are hundreds of algorithms that can do this and we will speak generally about the process. We will see more about how back propagation is done in Figure 10 where we try to predict the weight of Katie.
In figure 10 we see the “initial weights”. Initial weights are randomly assigned to values set close to zero (not on zero because zero is a stable and nonsensical solution).

Starting the weights close to zero puts them in the part of the Hyperbolic Tangent curve where Y is sensitive to small changes in X.

This initial prediction of Y, using these weights, will be, obviously, very bad. If all the weights are close to zero, the predicted Y will be close to zero.

We will only be able to predict Y if we adjust/train the W (weight) values.

Two parameters affect how fast weights are adjusted and how fast the ANN converges to a solution. Both parameters were created to deal with local minimums in the error term. The parameters are: a time varying learning factor and momentum.

The time varying learning parameter says that “at the beginning” your estimates are likely to be bad (remember, initial weights are close to zero) and we should “pass” all (or most of) the error back to the algorithm that adjusts weights.

We do this because, in the beginning, we want to make large adjustments to the weights – so we get fast convergence.

After several steps, we hope to be close to the (a) minimum and we want to make small adjustments so we do not overshoot the “maximum prediction point” (the inverse of an Error minimum).

Imagine a predicting function that has a dimension for every weight and many local minimums as is suggested by Figure 11. In the plot to the right Z is predictive power plotted vs 2 X variables. We can discuss some of the criticisms of ANNs using this figure.

Since high points are easier to see on a graph than low points we flipped our problem in Figure 11 from minimizing error to maximizing – the issues are the same. When an ANN is searching for an optimum, it is often following a ridge line to a “high point” that is a gently rounded curve. We would be, as business people, happy with any solution near the gently rounded “high point” – all solutions in the red oval near the top of the “mound” are, on a practical basis, equally good.

In addition, there is another top of a mound (green oval), that, while far away in x1 and x2, is as good as the red oval solution. ANNs are criticized as un-interpretable. If we study Figure 11 we ask “who cares”. If the solution I pick is good, AND there are OTHER EQUALLY GOOD Solutions, who cares if I can not interpret the details of the one solution I happen to pick. We say to the client “the model predicts Y with good accuracy and is a black box. Live with that situation”.

![Figure 10](image1)

**Neural Diagrams Example**

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Sex</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>KATIE</td>
<td>12</td>
<td>F</td>
<td>59</td>
<td>1</td>
</tr>
</tbody>
</table>

- Randomize weights to values approx. = 0
- Start in the sensitive and linear part of the activation function
- Use two parameters to help converge quickly and avoid local optimums in the error hypersurface
- Time varying Learning Factor (what % of error do you use)
- Momentum (if you made a large adjustment last iteration, make a large one this iteration as well)

![Figure 10](image2)

**Several sets of weights will work equally well**

![Figure 11](image3)
If we projected the error into a simpler picture (Figure 12, where we will try to minimize error) and are looking for minimum error we can discuss the momentum parameter again. Look at the red arrows. As we move to the left, we see we are approaching a local minimum and the momentum parameter was created to help the algorithm force the search up out of the local minimum. Momentum says “if we made a big X adjustment last “step” try to make a big adjustment this step.

Figure 12

Focusing on the black arrows in Figure 12 illustrates another problem with ANNs. If the algorithm happens to be “traversing a long flat section of the error hyper-plane, it can “time out” before it reaches the end of the flat section. One way to deal with the issue of local minimums is to set the number of tours (Figures 7 & 9) to a large number and this will cause JMP run many times with different initial weights (figure 10) and produce a weighted solution.

It is time to return to the example of training the weights in an ANN that will predict student weight (lbs.) from age (years) and height (inches).

We pass to the algorithm Johnson Transformed X variables and the algorithm uses algebra, and the initial weights that were randomly set close to zero, to make a first prediction of the Y value.

The first estimate of the weight in pounds, using randomly assigned network weights, is terrible. It is 1.5 pounds. The error is 93.5 and can be seen in Figure 13.

Since this is the first run, we expect that the calculated error is both large and really is error. By “really is error”, we mean a real difference from a true value and not some function of a local minimum of a complex error hyperplane.

Figure 13
We know that the first estimate of the Y value will be bad and we pass 1/1 (one divided by the iteration number) of the error back to the algorithm to be used in adjusting the weights. We want to have the algorithm, after the first step, make a large adjustments in weights. In the second iteration 1/2 of the error will be passed to the algorithm to be used in adjusting weights.

“Backward Propagation of the Error” assigns to each variable coming into a node a certain percentage of that error in the node. There are hundreds of ways to do this and the procedure can be complex, so I will treat this process as a black box.

When the algorithm has adjusted the weights it again calculates Y and calculates an error. In general, the process repeats, and weights are adjusted to reduce error—though details of the algorithm vary.

The process repeats until it converges.

Training methods can back propagate: 1) after each obs. is read or 2) after all obs. are read.

In the first method we look at Katie and try to adjust the weights to fit her data. Then we use those weights as starting points for adjusting weights as we process the second person.

Alternatively, we can read all obs., collect error information and try to adjust weights then after we have looked at everyone (and, of course, make several passes through the data).

Adjusting weights after each obs. seems to converge faster and seems to be the most commonly used method.

Figure 15 contains the final weights for all students and the error for Katie. This output is hard to interpret and this difficulty in interpreting weights is why people say that ANNs are black boxes.

**Business rules of thumb**

ANNs predict well but do not have the solid theoretical foundation that we can find under other techniques (notably Support Vector Machines). Accordingly practitioners must be satisfied with general comments and suggestions. I have collected several heuristics from web searches and books and present them below:
N or size of data set

The source data set should have several observations per weight and the number of weights can be large. Think in terms of tens of thousands of obs.

X variables should be numeric and in the range of -1 to +1 (or -5 to +5).

Dummy variable coding of categories can create too many variables. Consider something like thermometer coding. Eg. To a company selling cat food, the attractiveness of households is not linear with the number of cats (1,2,3,4,5,6,7 cats) they own.

Linear activation functions are often used in the layer next to the variables (to reduce the number of variables), and combined with a non-linear transform in the layer near Y.

Check that your training data contains all the levels of X and Y that you see in the “real world”. Check that your model produces all the levels of Y you see in the “real world”.

The more x variables you have the larger the network need to be.

The larger the network the greater the danger of over fitting and non-optimal weights.

Smaller networks often generalize better than large networks and will run faster.

The layer next to the variables should not have more nodes than you have variables.

Start with “few” nodes and change the number of nodes up and down.

for classification, start with one node per level of Y and hope that nodes will “specialize” in modeling different levels of Y.

Removing outliers can make for easier learning.

Recode important outliers to reflect their importance (see thermometer coding above).

Sometimes is it the combination of X values that make a point an outlier (a 6 foot tall ten year old).

Reduce the number of X variables before you start the ANN part of the process.

Look at correlations, t test of X vs Y, partial leverage plots, Decision trees, Proc Varclus Proc Factor

Swingler technique for pre-analysis of variables.

Build an ANN with many variables and few nodes.

Initialize the starting weights to zero (as always) and train the net.

If an variable is not important the weight well stay close to zero.

Drop unimportant X variables, re-run and compare results.

Scale the X variables – consider Softmax squishing.

Beware of using your ANN on a new data set where the ranges of X differ from the training set.

An evenly distributed data set is important for training the network. You should balance the numbers of cases for different values/levels of Y and scaling X values makes this more difficult.

You can sound profound if you say “Use complex networks for accuracy and simple networks for generalizability”.

How to think about, and adjust, ANN structure:

If training error is low and test error is high:
ANN is too complex and you have too many nodes-weights.
ANN has memorized the training data set quirks.

If error is high for both training and validation data sets:
  NN is too simple and not predicting well.
  You have too few nodes-weights and need to add nodes.

If all weights are large your ANN is too simple and not predicting well

Adding nodes-weights does not solve every problem:
  You could be using the wrong activation function.
  You could have a small N, or data that does not predict Y.

When do you stop training and looking for a good model?
  You run out of time or money.
  Your accuracy reaches and acceptable level for the business to use the model.
  You start to see the error in the training data set increase.

Analyzing weights:
  Take the weights, break the range into 10 buckets and count the weights in each bucket
  A plot of frequency should be mound shaped.
  A U-shaped frequency plot indicates a poor model.
  Running the final model topology/structure with different initial weights and comparing
  the changes in weight values gives an indication of model reliability.

If after training, no weights move from Zero, it is likely that the model has not learned anything.

ANNS become obsolete as the conditions that were modeled (possibly consumer confidence) changes but ANNS fail slowly
and gracefully. The timing of when to replace an old ANN with a new one can be difficult to get right (and to know if you get
wrong).

Summary:

ANNs are very powerful tools for predicting both categorical and continuous Y values. They are often used in conjunction with
other techniques like Decision trees and have gained widespread use. ANNS do not have the theoretical foundation of
Support vector Machines and are difficult to understand.

CONTACT INFORMATION:

Your comments and questions are valued and encouraged. Contact the author at:

Russ Lavery  Bryn Mawr, PA

Email: Russ.Lavery@verizon.net

Web page for all papers: Russ lavery.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in
the USA and other countries. ® indicates USA registration.
Other brand and product names are registered trademarks or trademarks of their respective companies.

References:


