Intrusion Detection System Design: Part III

Jon Ludwig, Brad Israel

1 Introduction

Intrusion Detection Systems (IDS) are designed to classify activity in order to differentiate between legitimate and illegitimate use of a computer or network. An IDS may monitor any type of activity, but a common type of activity to monitor is network traffic. An intrusion detection system which monitors network activity is called a Network-based Intrusion Detection System, or NIDS. Network traffic is relatively easy to capture and any intrusion from a remote machine should be present in a comprehensive log of the network activity. The problem that an NIDS attempts to solve is finding the traffic which should be classified as illegitimate activity and reporting it appropriately.

Finding illegitimate activity is a very difficult problem in most cases. Since a network is used for many diverse tasks, many times illegitimate traffic very closely resembles legitimate traffic. In this project we examine the application of neural networks to the problem of classification of traffic.

2 Neural Networks

Neural networks are systems of computation modeled after intelligent biological systems. The basic functional units of a neural network are the neuron and the synapse. The network is composed of a number of neurons connected together with synapses. A neuron is a function that takes some number of input values and produces an output value. A synapse is a connection between the output of a neuron and the input of a neuron. Synapses are often weighted. Neurons are typically arranged into layers, where two layers form a bipartite graph. In a *feed forward* neural network all values flow in the same direction, from the neurons in one layer to the neurons in the next, as shown in Figure 1. A network requires an *input layer* and an *output layer*, and may have one or more *hidden layers* between them.

Each neuron is essentially a composition of functions. A diagram of a neuron is shown in Figure 2. The inputs to each neuron i can be represented as a multidimensional vector $\mathbf{x}_i = \langle x_{i,1}, x_{i,2}, \ldots, x_{i,n} \rangle$, where $x_{i,j}$ represents the input to neuron i from neuron j. Similarly, $\mathbf{w}_i$ represents the weights associated with the inputs $\mathbf{x}_i$. Typically the neuron performs some type of integration of the input vector $\mathbf{x}_i$, usually a weighted summation as shown in Equation 1. An *activation function* is then applied to determine the magnitude of the neuron firing. In simple cases a hard-limited binary thresholding function is used as shown in Equation

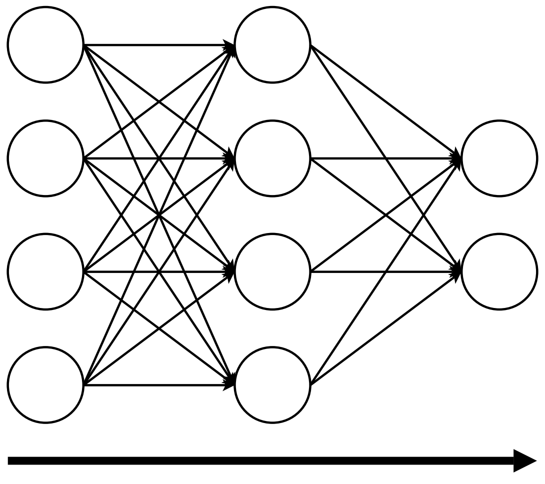

Figure 1: Feed Forward network

2. This function poses some problems as it is non-differentiable and makes adjusting the weights with back-propagation difficult. We have chosen to use a sigmoid function, as shown in Equation 3. Thus the output values y_i for each neuron are clamped in the interval $(0, 1)$.

$$f(\mathbf{x}_i, \mathbf{w}_i) = \sum_j w_{i,j} x_{i,j} \tag{1}$$

$$\phi_t(x) = \begin{cases} 1, & x < t \\ 0, & x \geq t \end{cases} \tag{2}$$

$$\phi(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

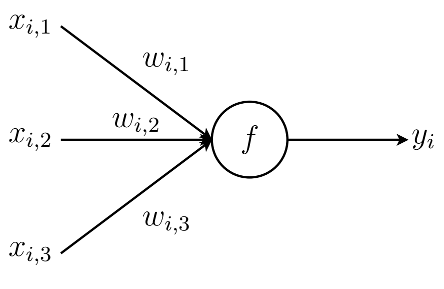

Figure 2: Neuron

2.1 Learning

In order to create a neural network which models a given data set we need to be able to modify the weights of the synapses. The integration and activation functions will be the same for each neuron, but the weights of the synapses will dictate how a neuron fires and thus how the network as a whole operates. In this project we explored several techniques for machine learning.

Since we have a data set which we would like to model, we can feed examples from the data set into the neural network, and calculate an error based on the result we desire and the result produced by the network. This is called *supervised learning.* One common error function known as the *mean square error*, or *MSE*, is given in Equation 4, where t_i is the desired output.

Once the error is calculated we would like to modify the weights of the networks such that the error of the modified network should be reduced. A common method to modify the weights of the synapses is a method called *back-propagation.* In back-propagation the weight vectors $\mathbf{w}_i$ are modified proportionally to impact that the synapse had on calculating the original result. There are many techniques for determining how much to modify the weight vector and in what direction. Probably the most common is known as *gradient descent.* Gradient descent calculates the gradient of the error surface and finds the steepest slope "downward" to reduce the error. In neural networks it is common to use the *stochastic gradient descent* algorithm which estimates the gradient of the error surface from one training example at a time. Equations 5 and 6 gives the equation used to calculate the new weights for each synapse connected to the output layer. Here η is the *learning rate.* For synapses connected to the hidden layers, Equation 7 is used instead, where k includes all the nodes which take the output of neuron i as an input (they are in the layer directly after neuron i in the feed forward paradigm). In some cases a *momentum* constant m is added to $\Delta w_{i,j}$ to help get out of local minima by factoring in the previous weight modification. Equation 8 shows how $\Delta w_{i,j}$ is calculated with a momentum parameter, where $\hat{\Delta} w_{i,j}$ is the weight change from the previous time step.

$$E = \frac{1}{2} \sum_i (t_i - y_i)^2 \tag{4}$$

$$\Delta w_{i,j} = -\eta \frac{\partial E}{\partial w_{i,j}} = \eta \delta_i x_{i,j} \tag{5}$$

$$\delta_i = \frac{\partial E}{\partial y_i} = -(t_i - y_i)(1 - y_i) y_i \tag{6}$$

$$\delta_i = (1 - y_i) y_i \sum_k \delta_k w_{k,i} \tag{7}$$

$$\Delta w_{i,j} = \eta \delta_i x_{i,j} + m \hat{\Delta} w_{i,j} \tag{8}$$

Another back-propagation algorithm which uses a heuristic to improve training speed is the *resilient back-propagation* algorithm. Here, only the sign of the partial derivative is examined and the weights are modified according to some update value $\Delta_{i,j}$. Equation 9 shows this method. The update value is modified according to whether or not the sign of the partial derivative changed in the last time step. If the sign did change the update value is multiplied by a constant η^-, if it did not change it is multiplied by a constant η^+.

$$\Delta w_{i,j} = \begin{cases} -\Delta_{i,j}, & \frac{\partial E}{\partial w_{i,j}} > 0 \\ +\Delta_{i,j}, & \frac{\partial E}{\partial w_{i,j}} < 0 \\ 0, & otherwise \end{cases} \tag{9}$$

3 Experiment

In order to evaluate how well neural networks perform in the task of intrusion detection we created several networks with varying parameters and using different training data sets. The neural networks were used to classify activity as either attack or normal. In some cases the networks were trained to identify the type of attack or differentiate between known and unknown attacks.

The training data was derived from the KDDCup 10% dataset [3]. The data was normalized so that all fields contained values in the interval $[0, 1]$ and the symbolic names were ignored. Then we created five sets of formatted data that singled out one type of attack, specifically the Neptune, Portsweep, Satan, Smurf, and Warezclient attacks. Most of the formatted datasets included 1000 attacks mixed in with larger amounts of normal data, except for the Satan attack set, which only had 521 attacks. We also created a mixed dataset that included 8000 known attacks from the training sets and 824 unknown attacks. Finally, we formatted the KDDCup data in the same way so that once the neural network was trained on the different datasets, we could run the network on the full dataset to see what attacks it could find.

To improve our understanding of neural networks, in general, we developed a piece of software in Java to construct a neural network with a set of given parameters, train it using back-propagation and stochastic gradient descent, and use it to classify network activity. We also utilized the Joone toolkit [1] to create a neural network for both training and validation. The data was formatted so that it could be read in on a file input synapse into the input nodes on the network. Each dataset has 33 columns (semicolon delimited) of test data for input and one column of the desired answer. When training the network, the last column is used to figure out the error and support the back-propagation method. From our tests, we found that setting the learning rate to 0.5 and momentum to 0.2 to return the best results. When the real data is run on the trained neural network, the last column is not used until after the network is finished processing. When the network is processing the real data, it outputs the error that the record generates on the output node of the network and stores it in a file. The ouput file is then read in by our "Results" class and compared to the answer column in the real dataset. If the generated error from network was closer to 0, then the record was classified as normal and if it was closer to 1, it was classified as an attack. Then the real answer was compared to the network classification to figure out how many of the attacks were classified properly and how many false negatives and positives there were.

4 Results

The numerical results of our experiments are shown in Table 1. For the individual attacks, the high number of false negatives is a little misleading because when the results were compiled by comparing the network output and the real dataset, it was looking at all the records that were known to be attacks, not just that type of attack. The false positives were still correct estimates of the amount of normal traffic that was mistakenly classified as an attack. Figure 3 shows the results of the individually trained neural networks along with the number of attacks that they correctly classified.

Attack Name	Number of Attacks	Total Records	Correctly Classified	False Positives	False Negatives	NN Training Time (ms)
Neptune	494021	396743	183997	5	310019	83288
Portsweep	494021	396743	110142	5	383874	3717743
Satan	494021	396743	98814	3	395204	1362517
Smurf	494021	396743	377203	870	115948	6095
Warezclient	494021	396743	97850	293	395878	1245717
Mixed	494021	396743	484853	5592	3576	191540

Table 1: Results Data Table

The results table also records the amount of time taken to train the neural network for each type of attack. Some of the data took up to an hour to train, which is obviously the downside to using a neural network to try and detect attacks in real-time.

When the neural network was trained on the mixed dataset of 8000 known attacks and 821 unknown attacks, the results were much better than expected. As shown in Table 1 and Figure 4, the trained neural network was able to correctly classify 98% of the records and had much fewer false negatives. The increase in false positives was probably because of increased complexity in the training required for multiple attacks.

5 Conclusions

The problems that we found with using neural networks to detect attacks is that it takes too much time to train the network to be viable for a real-time system. However, once the network is trained, it actually had much better performance than our anomaly detection IDS from Project 2, which was only able to correctly classify 78% of the records. Our training and real datasets and full java source code used in the project can be found in the respective zip and jar files.

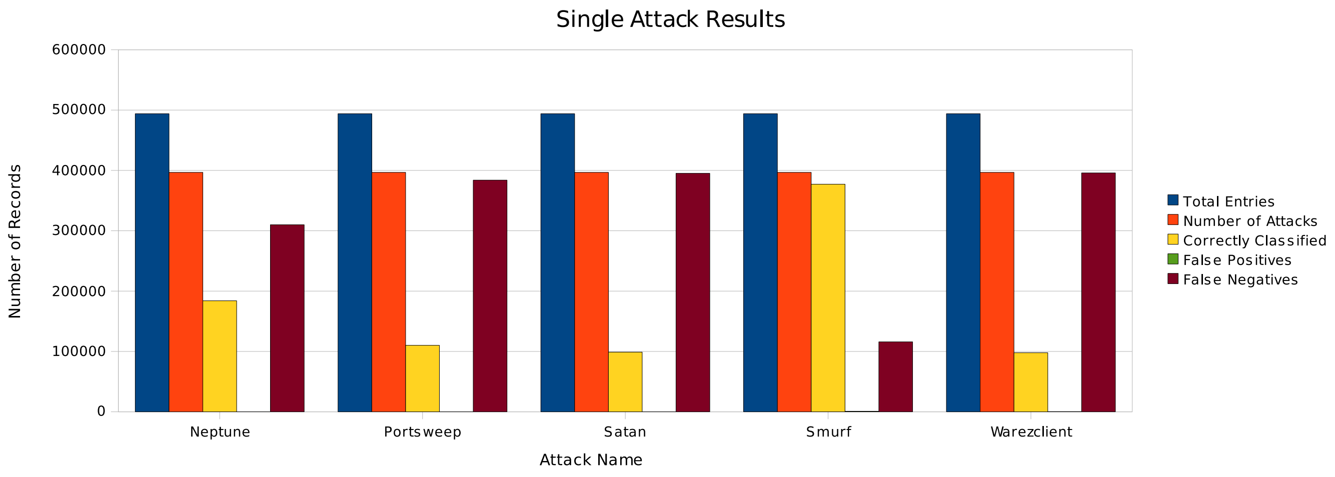

Figure 3: Results when training for single attacks

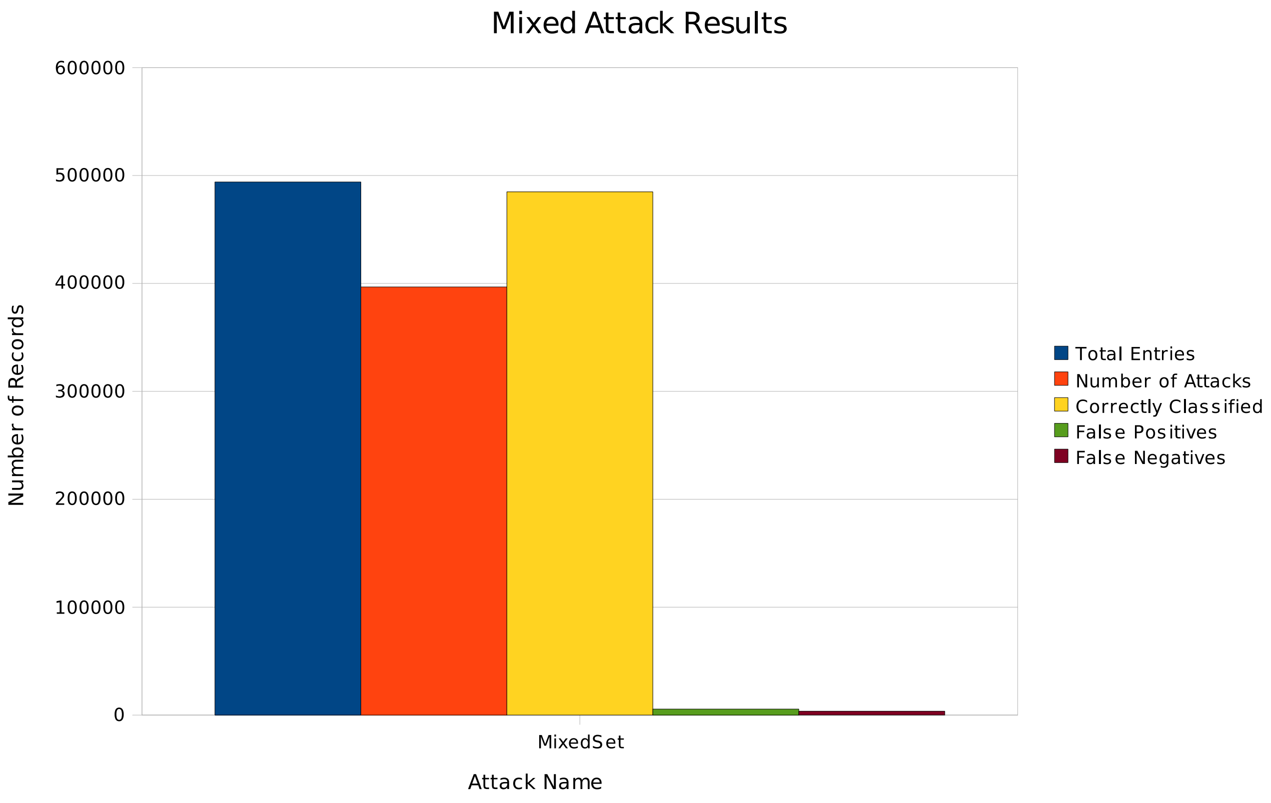

Figure 4: Results when training for mixed known and unknown attacks

References

[1] Joone: Java object oriented neural engine, January 2008.

[2] G. Brightwell, C. Kenyon, and Hélène Paugam-Moisy. Multilayer neural networks: One or two hidden layers? In Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, page 148. The MIT Press, 1997.

[3] S. Hettich and S. Bay. Kddcup 1999 dataset. *UCI KDD archive*, 1999.

[4] H. Lari-Najafi, M. Nasiruddin, and T. Samad. Effect of initial weights on back-propagation and its variations. *Systems, Man and Cybernetics, 1989. Conference Proceedings., IEEE International Conference on*, pages 218–219 vol.1, 14-17 Nov 1989.

[5] M. Riedmiller. Rprop-Description and Implementation Details. *University of Karlsruhe*, 1994.

[6] Raúl Rojas. *Neural networks: a systematic introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.