The Emerging Role of Defensive Artificial Intelligence in Cybersecurity

A Case Study with Implementation of an A.I.-powered Intrustion Detection System

Trevor J. Judd
Department of Computer Science
Tennessee Technological University
Cookeville, Tennessee
tjjudd21@students.tntech.edu

Justin A. Tatum
Department of Computer Science
Tennessee Technological University
Cookeville, Tennessee
jatatum42@students.tntech.edu

Jordan L. Crowell
Department of Computer Science
Tennessee Technological University
Cookeville, Tennessee
jlcrowell42@students.tntech.edu

Abstract—**The relevance of computing and computer networks has exploded in the twenty-first century, and the need for computer security has rapidly followed suit. As computer systems continue to grow in both size and complexity, human users have begun to turn to automated methods in order to secure valuable data and harden networks against attack. The complexity of these methods has grown in synch with the expansion of cyberspace, and we are now entering an era where computer security will be almost exclusively handled by automated systems powered by artificial intelligence. In this paper, we discuss the current state of automated cyber security, expound upon the challenges facing the current implementation of those systems, and demonstrate the implementation of an A.I.-driven Intrusion Detection System.**

Keywords—Intrusion Detection, Cyber Security, Artificial Intelligence, Neural Networks, Python, Keras, TensorFlow, Pandas

I. Introduction

Artificial intelligence is everywhere. It powers Internet commerce by automating processes like email and data record-keeping, predicting buyers' preferences, personalizing advertising, and providing protection against fraud. We now have communicative A.I.s like Siri and Cortana on our smartphones. IBM's Watson is even a *Jeopardy!* Champion. The applications for artificial intelligence are nearly limitless. According to InfoSec Institute, 63% of organizations are posed to deploy A.I.-based defensive systems in 2020 [6]. This is a necessary adaptation to the growing world of cyberspace, as the sheer amount of data at any given time cannot be handled by humans alone. Automation is necessary.

II. Related Work

A. Intrusion Detection Systems

The application of artificial intelligence to defense in cyberspace has become more prominent in the twenty-first century, as the need for better computer security grows. One of the most common applications is the automation of network intrusion detection. These systems are commonly known as Intrusion Detection Systems (IDS). The primary aim of an Intrusion Detection System is to identify when a malefactor is attempting to compromise the operation of a system [3].

There are two approaches to designing such systems. In a misuse detection-based IDS, intrusions are detected by looking for activities that correspond to known signatures of intrusion or vulnerabilities. On the other hand, anomaly detection-based IDS detects intrusions by searching for abnormal network traffic [1]. The IDS implemented with this paper uses anomaly detection to classify threats. Signature-based systems make up the majority of commercial network intrusion detection systems. Whilst such systems are highly effective against known threats, signature-based detection fails when attack vectors are unknown or known attacks are modified [3]. Anomaly-based detection is still largely a research exercise but shows great promise in compensating for the shortcomings of signature-detection.

B. Classification

Intrusion Detection Systems can be classified into three categories: host-based, network-based, and vulnerability-assessment.

- A *host-based* IDS assesses information found on a single/multiple host systems, including contents of operating systems, system and application files.
- A *network-based* IDS assesses information by analyzing the stream of packets traveling through a network.
- *Vulnerability-Assessment* involves gathering data on internal networks and firewalls.

The IDS implemented for this paper is a network-based IDS.

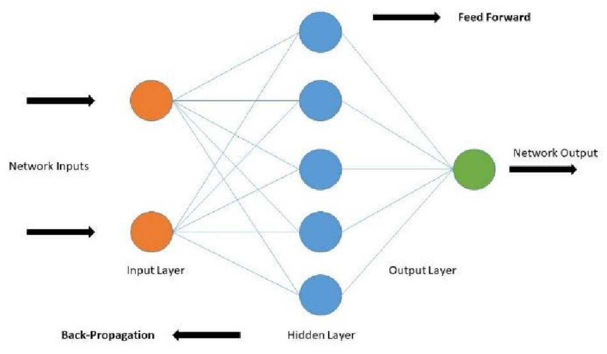
C. Honeypots

Researchers at Purdue University have created a detection system dubbed LIDAR, which stands for lifelong, intelligent, diverse, agile and robust [4]. LIDAR is a flexible A.I. capable of adapting to different threats through a combination of supervised and unsupervised machine learning and rule-based learning. The rule-based learning component takes information from the other two and judges the validity of an attack and coordinates an appropriate response. The most intriguing feature is the honeypot, a lure designed to entice would-be hackers but does not allow malicious actors to infiltrate the system. This gives LIDAR the ability to analyze a potential threat from a position of safety.

III. Core Design

Artificial Neural Networks (ANNs) are a class of machine learning algorithm that simulate the way in which human beings learn. ANNs consist of input and output layers of "neurons" as well as one or more hidden layers that transform the input into something the output layer can use. They are extremely useful in anomaly detection and pattern recognition, especially when applied to data sets that are too complex or too massive for human beings to comprehend. Although neural networks were discovered in the 1940s, they have only recently become a cornerstone of artificial intelligence thanks to a relatively new technique called backpropagation. Backpropagation allows neural networks to adjust their hidden layers to compensate for situations where an outcome doesn't match the prediction. This is done by exposing the ANN to different sections of training data multiple times. Ideally, each iteration moves closer to a minimum error rate, which is determined by an error function.

A network is organized into layers: the input layer is a row from the training dataset, and the first real layer is the hidden layer. The hidden layer is followed by the output layer, which has one neuron for each class value.



The Intrusion Detection System (IDS) implemented for this project utilizes an ANN with backpropagation coded in Python using TensorFlow and Keras libraries and was tested using the KDD99 data set.

A. Backpropagation

The backpropagation algorithm is a supervised learning method for a multi-layer feed-forward networks[5]. The principle of backpropagation involves modeling a given function by adjusting the weights of the inputs so that an expected output is produced. The IDS is trained using a supervised learning method, where the error between the observed output and the expected output is fed back to the system and used to update its internal state.

B. Forward Propagation

The output from a neural network is calculated by propagating an input signal through each layer until the signal reaches the output layer. This is called this forward-propagation. This technique is used to generate predictions during the training phase, and it is also used to make predictions on new data. Forward propagation can be broken down into three parts: neuron activation, neuron transfer, and forward propagation.

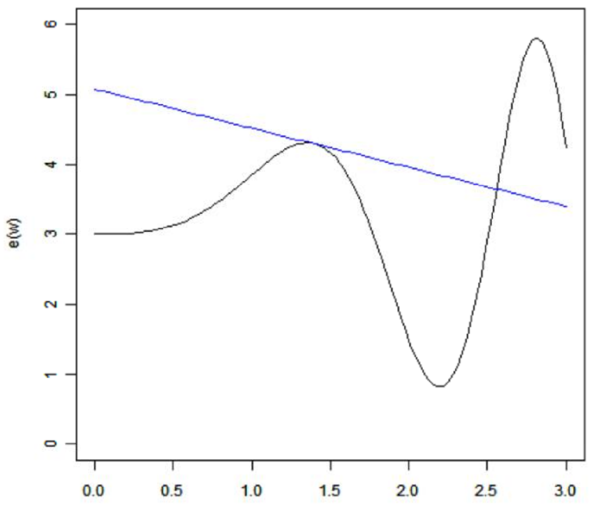
- *Neuron activation*: A neuron is "activated" when it is fed an input. The input can be a row from the training dataset, as in the case of the hidden layer, or it can be the outputs from each neuron in the hidden layer, in the case of the output layer. Neuron activation is calculated as the weighted sum of the inputs, much like linear regression [5].
- *Neuron transfer*: Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is. Different transfer functions can be used, but we chose to apply the softmax function for its utility in classification (the KDD99 dataset must be separated into categorical and continuous variables so that the neural network can be trained). Softmax is a form of logistic regression where an input value is normalized into a vector of values with a probability distribution whose total sums up to 1.
- *Forward propagation*: Forward propagating an input is straightforward: outputs for each neuron are calculated in each layer of the network. All of the outputs from one layer become inputs for the neurons in the next.

C. Error and Gradient Descent

Backpropagation is a type of gradient descent, which refers to the calculation of a derivative of the error function at each weight value in the neural network. The error function is derived from the difference between the expected outputs and the observed outputs. The gradient of the error function tells the backpropagation algorithm whether it should increase or decrease the weight [7].

Below is the graph of an example error function. The gradient is the partial derivative of each weight in the neural network with respect to the error function. A weight represents a connection between two neurons, and each weight has a gradient that is the slope of the error function. Calculating the gradient of the error function helps the algorithm to determine whether it should increase or decrease the weight of a particular connection between neuron pairs. Many propagation-training algorithms utilize gradients. In all of them, the sign of the gradient tells the neural network the following information:

- The weight is not contributing to the error of the neural network if the gradient is 0.
- The weight should be increased to achieve a lower error if the gradient is negative.
- The weight should be decreased to achieve a lower error if the gradient is positive.



Training using backpropagation can be summed up thusly: it is a search for the set of weights that will cause the neural network to have the lowest error for a training set, represented by the absolute minima in the graph above.

D. *Train Network*

The neural network is trained using stochastic gradient descent, which involves exposing the neural network to a training dataset across multiple iterations. Each row of data forward propagates the inputs, back-propagates the error, and updates the network weights. This process can be broken down into two sections: update weights and train network.

- *Update weights:* Once errors are calculated for each neuron in the network using backpropagation, they can be used to update the weights of connections between neuron pairs.
- *Train network:* Updating the neural network using stochastic gradient descent involves looping through the network for a fixed number of epochs. Within each epoch, the network is updated for each row in the training dataset.

E. *Predict*

Making predictions with a trained neural network is straightforward. We have already explained how forward propagation of an input signal produces output. Feeding the neural network input is all we need to do to make a prediction.

F. *KDD99*

The KDD99 data set is the data set used for the Third International Knowledge Discovery and Data Mining Tools Competition. It was provided to competitors whose task was to build a network intrusion detector capable of distinguishing between attacks and normal network activity. Although the data set is over twenty years old, it is still widely used in academic research. It consists of forty-seven numeric and string variables, and 494,021 rows. The problem presented to the IDS is one of classification: is an activity normal or is it malicious behavior? Twenty-five percent of KDD99 will be used to train the neural network while the remaining 75% is used to test the system.

IV. Evaluation

We originally intended to use Stochastic Gradient Descent (SGD) as our method of back-propagating error; however, in the course of our research we were exposed to unique tools and libraries available for Python (namely TensorFlow) that allowed us to test different algorithms for backpropagation. We discovered that SGD is actually the least efficient algorithm compared to available alternatives, such as Adaptive Moments Update, or ADAM.

A. *Stochastic Gradient Descent vs. ADAM*

SGD uses a randomly selected batch of training elements to update the weights, and is currently one of the most popular neural network training algorithms. Because the neural network is trained on a random sample of the complete training set each iteration, the error does not make a smooth transition downward. However, the error usually does trend downward [7].

ADAM is an optimization algorithm that, like SGD, uses randomized samples from the training data. It estimates the first and second moments (mean and variance, respectively) to determine the weight corrections, beginning with an exponentially decaying average of past gradients. This procedure is similar to classic momentum update; however, ADAM calculates the value of the average automatically based on the current gradient. The update rule then calculates the second moment.

The plots below show the training effectiveness of ADAM over SGD:

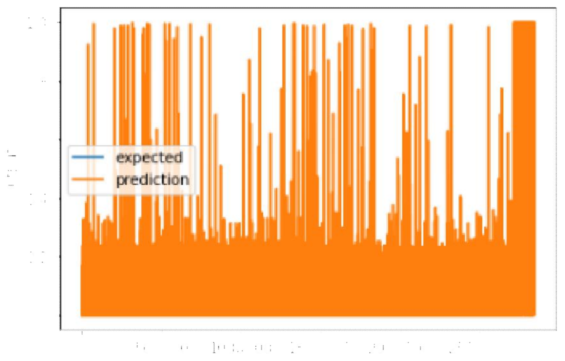


Figure 1

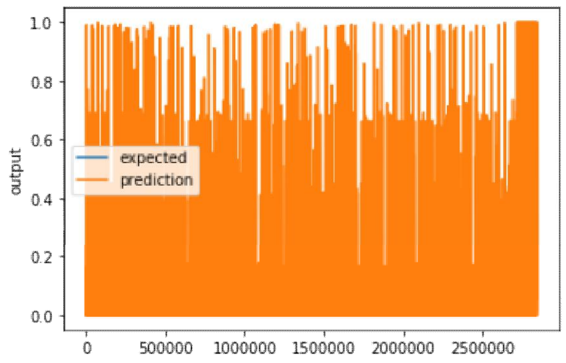


Figure 2

As can be seen from the regression charts, the neural network using ADAM (Figure 2) made more predictions with less training than the neural net using SGD (Figure 1).

B. Keras, TensorFlow, and Pandas

We began coding the Intrusion Detection System in Python, beginning with the functions needed to support the functionality of an Artificial Neural Network. It wasn't until we tried testing the IDS with KDD99 that we realized we had overlooked an obvious obstacle: the KDD99 dataset contains both continuous and categorical variables. We were used to working with datasets containing purely numerical data, and we had coded our first neural network accordingly. This led us to seek out supplementary libraries that could help us in processing the KDD99 data. We discovered three Python libraries that made the task of coding the IDS and training it with KDD99 immeasurably easier: Keras, TensorFlow, and Pandas.

Pandas is a Python software library written for data manipulation and analysis. Using Pandas, we were easily able to encode numeric columns as z-scores and non-numeric data as dummy variables. This preprocessing was key to getting the neural network to accept input properly. TensorFlow is an open-source symbolic math library written for Python with applications for deep learning neural networks. Keras is an open-source machine learning library designed to run on top of TensorFlow as a more user-friendly supplement.

Our exposure to Keras and TensorFlow allowed us to easily experiment using different training methods, like ADAM and SGD.

C. Results

Stochastic Gradient Descent yielded an accuracy score of 98.9%, so it is very good at distinguishing between normal and abnormal network activity. However, ADAM yielded an accuracy score of 99.5% and required less training than SGD. Perhaps in a future paper, we will do an analysis of the effectiveness of different backpropagation algorithms.

V. Contributions

Trevor Judd did the primary research into the implementation and functionality of the Intrusion Detection System, did the coding, and wrote Sections III, IV, and VI of this paper.

Justin Tatum helped in debugging code and contributed to Section II of this paper.

Jordan Crowell did research and contributed to Section II of this paper.

VI. Conclusion

It's tempting to imagine a time in the not-too-distant future where all security tasks are automated by artificial intelligences. There is definitely a growing trend toward automation in cyber security, and artificial intelligence is certainly a driving force to be reckoned with. Where exactly do humans fit into the picture?

It would be a mistake to completely disregard the human element in computer security. At this time, the best an automated security system can do is alert real-world humans to the possibility of a threat. The systems themselves cannot take action; although, they can recommend action to the human security analysts that monitor them. When machine learning models are added to the existing set of detection strategies (signatures, blacklists, and rules), the generation of this view can remain for the most part unchanged [2]. There are also other obstacles preventing artificial intelligence from completely replacing human security analysts, namely memory, data, and raw computing power, all of which might be prohibitively expensive for most companies. Another drawback is that hackers can also use A.I. themselves to test their malware and improve and enhance it to potentially become A.I.-proof [8]. Malware trained by machine learning algorithms can be hardened against detection by A.I.-powered security systems that use the same tools.

In the end, the human element cannot be completely removed. Artificially intelligent computer security systems will always need human programmers to teach them new tricks to find the bad guys and human security monitors to see what the A.I. itself might miss. Despite its power to make systems and networks more secure, artificial intelligence is ultimately limited by the fact that it is only *artificial*.

Acknowledgment

The authors would like to acknowledge Dr. Jeff Heaton, Ph.D., of Washington University, St. Louis, whose lectures on the applications of deep learning and neural networks were immensely valuable to the implementation, understanding, and explanation of the Intrusion Detection System described in this paper.

References

[1] E. Kesavulu Reddy, "Neural Networks for Intrusion Detection and Its Applications," Proceedings of the World Congress on Engineering, vol. 2, pp. 529–551, July 2013.

[2] I. Arnaldo and K. Veeramachaneni. (2019). *The Holy Grail of "Systems for Machine Learning"* [Online]. Available: https://www.kdd.org

[3] A. Shenfield, D. Day and A. Ayesh. (2018). *Intelligent Intrustion Detection Systems Using Artificial Neural Networks* [Online]. Available: https://www.sciencedirect.com

[4] C. Adam. (2020, Feb 6). *Intrustion Alert: System Uses machine learning, curiosity-driven 'honeypots' to stop cyber attackers* [Online]. Available: https://www.purdue.edu

[5] J. Brownlee. (2019, Dec. 1). *How to Code a Neural Network with Backpropagation in Python (from scratch)* [Online]. Available: https://www.machinelearningmaterty.com

[6] P. Paganini. (2019, Dec. 30). *Top Cybersecurity Predictions for 2020* [Online]. Available: https://www.resources.infosecinstitute.com

[7] J. Heaton. (2019). *T81-558: Applications of Deep Neural Networks Module 4 Part 4* [Online]. Available: https://github.com/jeffheaton

[8] A. Laurence (2019, Aug. 22). *The Impact of Artificial Intelligence on Cyber Security* [Online]. Available: https://www.cpomagazine.com

Appendix

(see attached for source code)

```
%matplotlib inline

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.utils import get_file

# Regression chart
def chart_regression(pred, y, sort=True):
	t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
	if sort:
		t.sort_values(by=['y'], inplace=True)
	plt.plot(t['y'].tolist(), label='expected')
	plt.plot(t['pred'].tolist(), label='prediction')
	plt.ylabel('output')
	plt.legend()
	plt.show()

try:
	path = get_file('kddcup.data_10_percent.gz', origin='http://kdd.ics.uci.edu/dat
abases/kddcup99/kddcup.data_10_percent.gz')
except:
	print('Error downloading')
	raise

print(path)

df = pd.read_csv(path, header=None)

print("Read {} rows.".format(len(df)))
df.dropna(inplace=True,axis=1)

# Add column heads to the CSV file
df.columns = [
	'duration',
	'protocol_type',
	'service',
	'flag',
	'src_bytes',
	'dst_bytes',
	'land',
	'wrong_fragment',
	'urgent',
	'hot',
	'num_failed_logins',
	'logged_in',
	'num_compromised',
	'root_shell',
	'su_attempted',
	'num_root',
	'num_file_creations',
	'num_shells',
	'num_access_files',
	'num_outbound_cmds',
ENCODING = 'utf-8'

def expand_categories(values):
        result = []
        s = values.value_counts()
        t = float(len(values))
        for v in s.index:
                result.append("{}:{}%".format(v,round(100*(s[v]/t),2)))
        return "[{}]".format(",".join(result))

def analyze(df):
        print()
        cols = df.columns.values
        total = float(len(df))

        print("{} rows".format(int(total)))
        for col in cols:
                uniques = df[col].unique()
                unique_count = len(uniques)
                if unique_count>100:
                        print("** {}:{} ({}%)".format(col,unique_count,int(((unique_count)
                / total
l)*100)))
                else:
                        print("** {}:{}".format(col,expand_categories(df[col])))
                        expand_categories(df[col])

import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore

analyze(df)

# Feature engineering
# Encode numeric variables as z-scores
def encode_numeric_zscore(df, name, mean=None, sd=None):
        if mean is None:
                mean = df[name].mean()

        if sd is None:
                sd = df[name].std()

        df[name] = (df[name] - mean) / sd

# Encode text values to dummy variables
def encode_text_dummy(df, name):
        dummies = pd.get_dummies(df[name])
        for x in dummies.columns:
                dummy_name = f"{name}-{x}"
                df[dummy_name] = dummies[x]
        df.drop(name, axis=1, inplace=True)
```

```
# Encode the feature vector

encode_numeric_zscore(df, 'duration')
encode_text_dummy(df, 'protocol_type')
encode_text_dummy(df, 'service')
encode_text_dummy(df, 'flag')
encode_numeric_zscore(df, 'src_bytes')
encode_numeric_zscore(df, 'dst_bytes')
encode_text_dummy(df, 'land')
encode_numeric_zscore(df, 'wrong_fragment')
encode_numeric_zscore(df, 'urgent')
encode_numeric_zscore(df, 'hot')
encode_numeric_zscore(df, 'num_failed_logins')
encode_text_dummy(df, 'logged_in')
encode_numeric_zscore(df, 'num_compromised')
encode_numeric_zscore(df, 'root_shell')
encode_numeric_zscore(df, 'su_attempted')
encode_numeric_zscore(df, 'num_root')
encode_numeric_zscore(df, 'num_file_creations')
encode_numeric_zscore(df, 'num_shells')
encode_numeric_zscore(df, 'num_access_files')
encode_numeric_zscore(df, 'num_outbound_cmds')
encode_text_dummy(df, 'is_host_login')
encode_text_dummy(df, 'is_guest_login')
encode_numeric_zscore(df, 'count')
encode_numeric_zscore(df, 'srv_count')
encode_numeric_zscore(df, 'serror_rate')
encode_numeric_zscore(df, 'srv_serror_rate')
encode_numeric_zscore(df, 'rerror_rate')
encode_numeric_zscore(df, 'srv_rerror_rate')
encode_numeric_zscore(df, 'same_srv_rate')
encode_numeric_zscore(df, 'diff_srv_rate')
encode_numeric_zscore(df, 'srv_diff_host_rate')
encode_numeric_zscore(df, 'dst_host_count')
encode_numeric_zscore(df, 'dst_host_srv_count')
encode_numeric_zscore(df, 'dst_host_same_srv_rate')
encode_numeric_zscore(df, 'dst_host_diff_srv_rate')
encode_numeric_zscore(df, 'dst_host_same_src_port_rate')
encode_numeric_zscore(df, 'dst_host_srv_diff_host_rate')
encode_numeric_zscore(df, 'dst_host_serror_rate')
encode_numeric_zscore(df, 'dst_host_srv_serror_rate')
encode_numeric_zscore(df, 'dst_host_rerror_rate')
encode_numeric_zscore(df, 'dst_host_srv_rerror_rate')

# Convert to numpy
x_columns = df.columns.drop('outcome')
x = df[x_columns].values
dummies = pd.get_dummies(df['outcome']) # Classification
outcomes = dummies.columns
num_classes = len(outcomes)
y = dummies.values

df.groupby('outcome')['outcome'].count()
```

```
# Split test and train data. 25% test, 75% train
x_train, x_test, y_train, y_test = train_test_split(
x, y, test_size=0.25, random_state=42)

# Create neural network
model = Sequential()
model.add(Dense(10, input_dim=x.shape[1], kernel_initializer='normal', activation='
relu'))
model.add(Dense(50, input_dim=x.shape[1], kernel_initializer='normal', activation='
relu'))
model.add(Dense(10, input_dim=x.shape[1], kernel_initializer='normal', activation='
relu'))
model.add(Dense(1, kernel_initializer='normal'))
model.add(Dense(y.shape[1],activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='sgd')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, verbose=1,
mode='auto')
model.fit(x_train,y_train,validation_data=(x_test,y_test),callbacks=[monitor],verbo
se=2,epochs=1000)

pred = model.predict(x_test)
chart_regression(pred.flatten(),y_test)

# Measure accuracy
pred = model.predict(x_test)
pred = np.argmax(pred,axis=1)
y_eval = np.argmax(y_test,axis=1)
score = metrics.accuracy_score(y_eval, pred)
print("Validation score: {}".format(score))
```