Universiteit Leiden
The Netherlands

# Opleiding Informatica

Detecting anomalies with recurrent neural networks

Sander Ronde

Supervisors:

Wojtek Kowalczyk & Bas van Stein

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 16/11/2017

# Abstract

Due to the widespread usage of computer networks and numerous attacks on them, a fast and accurate method to detect these attacks is an ever growing need. In this thesis, a system using a Recurrent Neural Network (RNN) is explored as a method to detect intrusions. This system is applied to an unlabeled cyber-security data set in order to determine its effectiveness. The goal is to train the system on every individual user in this data set in order to learn their behavior and to find any deviations in their behavior. It should be stressed that deviations in behavior (also known as anomalies) cannot be labeled as "intrusions" without the involvement of domain experts. Nevertheless, they can be used for identifying potential attacks and presenting them to cyber-security experts for further evaluation. Several architectures for this system are explored in order to find the optimal one. However, results show that using an unlabeled data set for the training of this network leaves no good measure of the accuracy of the system. This makes finding the optimal architecture a hard task. The system used in this thesis has been shown to find likely anomalies, leading to the conclusion that RNNs are an effective method for flagging anomalous behavior.

# Contents

# Chapter 1

# Introduction

As the presence of computer networks in our day-to-day lives increases, the need for a method to detect attacks or abuse of these networks also increases. A common approach to this problem is the analysis of network log files that contain information about system activities, such as login attempts, transfers of data, etc. The problem is that only a cyber-security expert can conclude whether certain behavior is actually an attack. This also needs to happen very quickly as there tends to be a lot of data in these logs. This calls for a computer system handling this problem as humans simply can't keep up with the amount of data. A system that does this needs to be both fast and good at identifying anomalous behavior, while at the same time being able to adapt to any changes the attackers might make to avoid it. The system should, preferably, also be able to run in real-time, being able to detect any abnormal behavior as it happens. This can be a very important factor in data breaches. The field of Deep Neural Networks (DNNs) seems to present a solution for this problem; it combines both the speed of computers and attempts to mimic the ability of our brains to learn very quickly, which allows it to recognize complex patterns.

In 2015, a data set[1] was published in [Ken15], containing around 100GB of anonymized event data collected from the US-based Los Alamos National Laboratory's internal network over 58 consecutive days. This data set consists of a number of different types of data: authentication, process, network flow, DNS and red team data. The authentication data is by far the biggest with 1,051,430,459 out of 1,648,275,307 total events. The red team data represents a set of simulated intrusions. This type of data is present to train the system on known intrusions (also known as misuse detection) or to validate the system's findings. However, there is so little red team data (749 actions) that it is not feasible to do this. Since the rest of the data is unlabeled, meaning it is not known whether or not they are actually attacks, the system needs to be trained to recognize users' behavior. The next step is to try to find any deviations from this behavior (also known as anomalies). Because the data consists of a series of actions, sequences of events that are only anomalies when observed together (also known as collective anomalies) might also be in the data set. Collective anomalies would go unnoticed when only reading the data one action at a time. However, a recurrent neural network (RNN), which is particularly good at series of data, is able to find these collective anomalies, making it a perfect fit for this purpose.

[1] The data set can be found at https://csr.lanl.gov/data/cyber1/

The main goal of this thesis is to evaluate the effectiveness of using RNNs for finding anomalies in cyber-security related data, in particular with regards to unsupervised learning (learning on unlabeled data). The approach to this goal is to attempt to find anomalies in the previously mentioned data set, experimenting with different parameters to the neural network and different RNN architectures. Finding anomalies is done by transforming the data set into a vector of features that are then used to train a network to predict the behavior of a user. The network then predicts the next feature vector based on the previous feature vectors. This prediction is then compared to the actual features. If the prediction deviates too much from the mean differences calculated over the training set, these features (and the action they were constructed from) are then classified as anomalies.

This thesis is structured as follows: in Chapter 2 related work is discussed; in Chapter 3 the RNN architecture is explained; in Chapter 4 the used methods are described; in Chapter 5 results of the experiments regarding the network's architecture are analyzed; in Chapter 6 the time taken to run the system is discussed; in Chapter 7 the findings are presented and Chapter 8 contains the conclusion.

# Chapter 2

# Related Work

The field of anomaly detection has been a very active field of research in the last 50 years, becoming even more active as the amount and the complexity of the to data sets increases. In [R$^+$99], a lightweight system to scan a network's active data flow and to find possible intrusions based on known attacks was proposed. In [LS$^+$98], simple classifiers were used to find anomalous behavior based on known intrusion patterns and changes in user behavior. More advanced techniques like clustering have also been used to find anomalies in unlabeled data sets. Since 2001 there were attempts to build a system that also detected yet unknown intrusion patterns and anomalous changes in user behavior using clustering, attempting to go beyond the constant search for new intrusion patterns that felt like a cat-and-mouse game for the developers of these systems, as explored by [PES01].

With the upcoming field of deep learning in machine learning, the interest for applying these fields to anomaly and intrusion detection has also increased greatly. They were shown to have great potential, as explained in [LBH15]. In [RLM98], a simple backpropagation neural network was applied to the terminal commands a user executed, an attempt was made to identify users by these commands in order to find any deviations, finding that this is an effective way of detecting intrusions. Contrary to rule-based analysis, neural networks perform a lot better on noisy data where some fields may be missing or incomplete, as [Can98] shows. Here a neural network was applied to noisy computer network metadata in order to detect different methods of attack.

In a recurrent neural network, further explained in [LBH15], the output of one cell is connected to the input of the next cell. As a result processed information from previous cells' inputs should make it through to later cells. The information that is fed forward is selected based on what the network is trained to select. In theory, the RNN has the ability to recall previous inputs. In practice, however, standard RNNs seem to fall off when remembering for longer periods of time, often not remembering the data for more than about 5–6 iterations. This problem was investigated in [BSF94] among others, finding problems with gradient based learning algorithms when applied to RNNs. This then prompted the development of the now commonly used Long Short Term Memory (LSTM) architecture for RNNs, which was introduced in [HS97].

RNNs using the LSTM architecture proved very useful in finding so-called time series anomalies, which are anomalies over a time frame with multiple actions, instead of single-action anomalies. LSTM networks excel at this due to their ability to learn which aspects of the previous input data should be remembered and which aspects to forget. This is shown in [MVSA15], where a stacked LSTM, trained to recognize regular behavior, was demonstrated to perform well on 4 different data sets. Each of these 4 data sets contains some anomalies, ranging from long-term to short-term anomalies, and from data containing a single variable to 12 variables. LSTM networks tend to be able to find so-called collective anomalies where other types of anomaly detection would not find them, being able to link together multiple instances of slightly deviating behavior into a definitive anomaly. This technique was applied in [OH15], where they were able to probabilistically group together the contribution of individual anomalies in order to find significant anomalous groups of cases.

# Chapter 3

# Recurrent Neural Networks

## 3.1 Recurrent Neural Networks

A recurrent neural network is a variation of an artificial neural network that continuously uses the output of its previous cell as the input of the current cell along with the input that was applied to this cell (see Figure 3.1). Due to this feature, RNNs have the ability to store processed information from the previous input in the hidden state as these can be passed along through the previous cell's output.

As can be seen in Figure 3.2, RNNs only have a single vector that functions as the hidden state and the cell's output. The vector of values at the output of the hidden layer that are observed at time $t$, $\mathrm{h_t}$ is calculated by the following function where $\mathrm{h_t}$ is the current hidden state vector and $\mathrm{h_{t-1}}$ the vector of the previous hidden state, $W$ is the weight matrix for the cell's input, $U$ is the weight matrix for the hidden value, $\mathrm{x_t}$ is the input of this cell, and $\sigma$ is a sigmoid function used to squash its inputs into the [0, 1] range:

$$h_t = \sigma(Wx_t + Uh_{\mathrm{t-1}}) \tag{3.1}$$

These standard RNN cells and any other RNN architectures could be stacked on top of each other. These

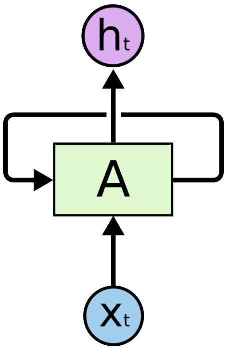

Figure 3.1: A recurrent neural network. From [Ola15]

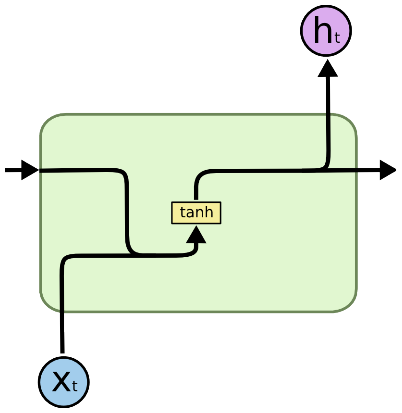

Figure 3.2: A single RNN cell. From [Ola15]

are also known as deep recurrent neural networks. In [PGCB13], deep RNNs were shown to outperform conventional single-layer RNNs at polyphonic music prediction and language modeling. An RNN architecture that takes advantage of the stacking of layers in particular is the Depth Gated RNN, introduced in [YCV$^+$15], which uses an additional depth gate to connect memory cells of adjacent layers.

## 3.2 LSTMs

The LSTM architecture, contrary to regular RNNs, has an additional hidden state that is never directly outputted (see Figure 3.3). This additional hidden state can then be used by the network solely for remembering previous relevant information. Instead of having to share its "memory" with its output, these values are now separate. During the training process, an LSTM learns what should be remembered for the future and what should be forgotten, which is achieved by using its internal weights.

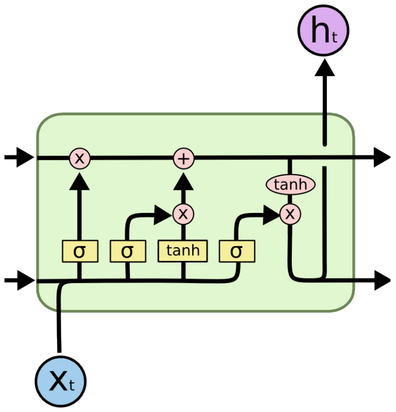

Figure 3.3: A single LSTM cell. From [Ola15]

As can be seen in the figure, there are quite a few more parameters in this cell than in a normal RNN cell. The calculation of the output vector and the hidden vector involves several operations, a full explanation of which can be found in [Ola15]. First of all the network determines how much of the hidden state to forget, also called the forget gate. This is done by pushing both the previous iteration's output ($c_{t-1}$) and the forget gate vector ($f_t$) through a matrix multiplication. This allows the network to forget values at specific indices in the previous iteration's output vector. $f_t$ can be obtained by using formula 3.2, where $W$ contains the weights for the input and $U$ contains the weights for the previous iteration's output vector, $x_t$ refers to the input, $h_{t-1}$ to the previous iteration's output vector and $b$ to a set of bias vectors:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{3.2}$$

The network then determines what to remember from the input vector. This is commonly referred to as the input gate. This is done by pushing the previous forget gate's output as well as the input gate through a matrix addition function. The output of the input gate ($i_t$) can be found by using the following formula:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{3.3}$$

The final hidden state vector ($c_t$) can then be found by using the previous two results as follows, where $\circ$ denotes the Hadamard product (where each value at index $ij$ is the product of the values at the indices $ij$ in the two input matrices):

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma(W_c x_t + U_c h_{t-1} + b_c) \tag{3.4}$$

This vector is then passed on to the next iteration. Now the output gate vector $o_t$ is calculated:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{3.5}$$

The output state $h_t$ can then be obtained:

$$h_t = o_t \circ \sigma(c_t) \tag{3.6}$$

This results in a version of an RNN that is able to remember more and is more liberal in choosing what information it wants to keep in the hidden state and what it wants to discard. This makes LSTM networks better suited for tasks involving series of data. This has lead to the LSTM architecture becoming the dominant RNN architecture.

The total number of trainable weights in an LSTM layer can be calculated as follows, where *input_dim* is the last dimension of the input vector and *units* is the amount of LSTM cells in this layer:

$$weights = (4 * units * input\_dim) + (4 * units * units) + (4 * units) \tag{3.7}$$

In this formula, the number 4 comes from there being 4 different weights for calculating the gates (forget, input, hidden state and output). The first term (also known as the kernel weights) then represents the $W$ matrix in the above functions, the second term (also called the recurrent kernel) represents the $U$ matrix and the third term is a matrix containing biases $b$.

## 3.3 GRUs

Another RNN architecture is the Gated Recurrent Unit (GRU), introduced in [CVMG+14]. This architecture combines the input and forget gates into a single so-called "update gate" and also merges the cell state and hidden state (see Figure 3.4). The calculation of the merged output vector once again consists of several operations. The network first computes the "reset gate" $r_t$ using the following function, where $W_r$ are the weights for the reset gate and $[h_{t-1}, x_t]$ signifies the concatenation of $h_{t-1}$ and $x_t$:

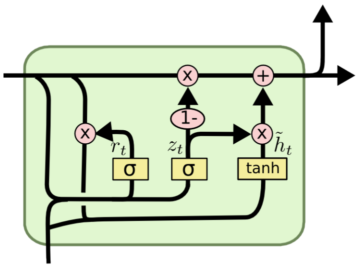

Figure 3.4: A single GRU variation cell. From [Ola15]

$$r_t = \sigma(W_r[h_{t-1}, x_t]) \tag{3.8}$$

After this, the "update gate" $z_t$ is computed as follows, where $W_z$ holds the weights of the update gate:

$$z_t = \sigma(W_z[h_{t-1}, x_t]) \tag{3.9}$$

The output vector $h_t$ (representing both the cell's output and its state) can then be computed by the following function:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h_t} \tag{3.10}$$

Where $\tilde{h_t}$ is computed by:

$$\tilde{h_t} = \tanh(W * [r_t * h_{t-1}, x_t]) \tag{3.11}$$

## 3.4 Architectures

There have been numerous variations of the standard LSTM architecture that was just described, including but not limited to the Peephole LSTM [GSS02] that connects the hidden state with the gate activation functions, and the Neural Turing Machine [GWD14], which extends the capabilities of the neural network by coupling it to an external memory resource.

There has been some research into which architectures are the most efficient. Some architectures are better than others at specific problems, as [JZS15] demonstrates. However, this study did not include a variation of anomaly detection in its testing problems. This means that experiments regarding what architecture is the most promising for anomaly detection still have to be done. Experiments regarding the architecture of the system are done in Chapter 5. The base architecture that will be used in this thesis will be the standard LSTM architecture. Since it is the dominant architecture and has seen a lot of use, testing other architectures against this baseline seems like a good choice. For training the "adam" optimizer will be used, which was introduced in [KB14]. This is a method of gradient-based optimization of stochastic objective functions.

# Chapter 4

# Methods

In the original data set that is being used in this thesis, there is a lot of hidden information is essential, for example whether the user connected to a computer he/she hasn't visited before. This is the reason for so-called features being created based on the data. These features are then sent to the network to train on, after being split into a training set and a test set. Since individual users tend to have different behavioral patterns, the decision was made to train one network per user instead of sharing a network. Another option was explored, as explained in Chapter 5. After training on the training set, the network is then applied to the test set. A list is then composed for the training set and the test set, containing the differences between the expected and actual vectors. Any elements in the list of the test set deviating too much from the median of the training set's list (which should represent normal behavior), are then labeled as anomalies.

## 4.1 Data

The data set from [Ken15] contains a number of different types of data, as described in Chapter 1. In this thesis, only the authentication data will be used since that is by far the largest data set with 1,051,430,459 events. The data set covers a total of 17,684 computers and spans 58 consecutive days. There are 26,301 total users in the data set, 13,875 of which are computer users. These are not tied to specific persons and as such learning their behavior will not be very useful. As such these are not included in the testing/training sets. In addition to these users there is a single "anonymous user" that is also excluded from the testing/training sets. This leaves a total of 12,425 human users. A minimum number of 150 actions per user has also been chosen in order for them to be included in the testing/training sets. The number of users that do not meet this criterion is not known as the system has not been run on 100% of the data set. The number 150 was chosen because, since weights are updated after every batch, with a batch size of in this case 32, the weights can only be updated a few times, leading to a bad approximation of the user's behavior. It could be argued that the minimum number of actions should be even higher. However, it should be fairly easy to determine if a user's behavior was only flagged as anomalous because they have so few actions that the network hasn't learned their behavior yet, making this a small problem. The data was entirely anonymized. Also the true dates over which data were collected has not been disclosed. The authentication data format can be seen in Table 4.1

Table 4.1: The data set structure

| time | source user@domain | destination user@domain | source computer | destination computer | authentication type | logon type | authentication orientation | success/failure |
|---|---|---|---|---|---|---|---|---|
| 1 | C625@DOM1 | U147@DOM1 | C625 | C625 | Negotiate | Batch | LogOn | Success |
| 1 | C625@DOM1 | SYSTEM@C653 | C653 | C653 | Negotiate | Service | LogOn | Success |
| 1 | C625@DOM1 | SYSTEM@C653 | C660 | C660 | Negotiate | Service | LogOn | Success |

Only human users having more than 150 records are used in this thesis. Since anonymous users and computer users do not represent a single person, and the persons that use a given computer may change at any time, attempting to model the behavior of these users is not effective.

## 4.2 Features

Some variables like the computer's name can not be used as input for a neural network, instead they are used to define features. Since increasing the number of features has a big performance impact, keeping the number of features low, while still making sure the most important data is represented by them is essential. The features are created on a per-user basis, iterating through that user's events and generating features based on them. For an overview of the features see Table 4.2. All categorical values (indices 5,6,7) have been encoded using 1-of-encoding (see Table 4.3 for an example), ensuring the values remain categorical. This gives each possible categorical value a single spot in a vector, setting it to 1 only if that value is true, and setting the rest to 0. This allows the network to make individual predictions for every possible categorical value, instead of having to output a single prediction for all values through one real-numbered output. This brings the total length of the feature vector up to 33 (11, 9 and 6 possible values for the categorical values respectively).

Table 4.2: The features

| Index | Feature | Description |
|---|---|---|
| 0 | domains delta | 1 if a previously unvisited domain was accessed, 0 otherwise |
| 1 | dest users delta | 1 if a previously unvisited destination user was accessed, 0 otherwise |
| 2 | src computers delta | 1 if a previously unvisited source computer was accessed, 0 otherwise |
| 3 | dest computers delta | 1 if a previously unvisited destination computer was accessed, 0 otherwise |
| 4 | time since last access | The time (in seconds) since the last time any network activity occurred |
| 5 | auth type | What type of authentication type was used (categorical data) |
| 6 | logon type | What type of logon type was used (categorical data) |
| 7 | auth orientation | What type of authentication orientation was used (categorical data) |
| 8 | percentage failed logins | The percentage of logins that were unsuccessful |
| 9 | success or failure | 1 if the login succeeded, 0 if it didn't |

Table 4.3: 1-of-encoding

|  | $x_1$ | $x_2$ | $x_3$ |
|---|---|---|---|
| A | 1 | 0 | 0 |
| B | 0 | 1 | 0 |
| C | 0 | 0 | 1 |

Table 4.4: The translation of three values (A, B and C) into a vector representing each one.

The features have been chosen in such a way to contain relevant information that can be used for modeling user behavior. For example, the neural network is unlikely to keep track of every computer the user logged into, instead, the data might contain information about whether the user logged in to a computer they haven't previously logged in to. Additionally, the neural network is unlikely to subtract the previous action's time stamp from the current action's time stamp, but will probably be interested in knowing the time since the last action. This can be helpful for determining whether the user is doing lots of operations at once, doing them at a normal human speed, or if they're barely doing anything at all.

## 4.3 Preprocessing

In order to have both a training and test set for every user, the data is chronologically split into 70% for training and 30% for testing. This is done separately for every user, making sure that each user has the same 70–30 split. As every feature except for time since last access falls in the range [0,1], this feature also needs to be fit into that range. This is done by taking the maximum value for the time since last access column and dividing every value in that column by the maximum value in the column, linearly scaling every value down to the range [0,1].

The data is kept in chronological order, as it was read from the data set file, ensuring that the input data closely resembles the input data as it would appear in a real network, and making use of the LSTM's ability to make sense of sequences.

Keep in mind that in a real-time scenario, scaling can not be done by using the same factor for both the training and test set, as the eventual maximum value is unknown, leading to values that fall above the [0,1] range. This can be solved by taking the maximum possible or reasonable value as a scaling factor for both test sets. For example no user will ever access more computers than are available on the network and no human user will have more seconds between their last action than there are in a human lifetime. Another method of solving this problem is to apply the following function to all (unscaled) feature values:

$$x' = \frac{1}{1+x} \tag{4.1}$$

Instead of continuously increasing, $x'$ shrinks here, fixing the problem of features exceeding the range [0,1]. This also takes care of scaling the feature down to the range [0,1]. As such this is a very good solution to this problem. However, because no real-time training/testing occurs in this thesis, the problem does not have to be dealt with.

## 4.4 Experimental setup

The system consists of 3 layers, with the first two being stateful LSTMs (stateful meaning the state is preserved across batches), and the third layer being a dense layer using the ReLU activation function which was introduced in [HSM+00]. The 3 layers contain $feature\_size$ (33) cells each, which is "between" the network's input and output sizes, both being $feature\_size$ as well (as suggested in [Hea08]). This means that each layer's output vector is also of size $feature\_size$. The dense layer transforms the data to the desired output dimensions of ($feature\_size$, 1) where. All three layers have biases enabled and use the default keras initializer "glorot_uniform", introduced in [GB10], which draws its values from a uniform distribution in the range of [-limit, limit]. Limit is calculated as follows, with fan_in being the weight's input dimension and fan_out being the weight's output dimension (both always equal to $feature\_size$ except for the first layer's kernel weights, where fan_in is 1):

$$limit = sqrt(6/(fan\_in + fan\_out)) \tag{4.2}$$

The first LSTM layer also returns its sequences instead of only last output, allowing the second LSTM to be stateful. The total number of weights for the LSTM layers can be calculated by using formula 3.7. The number of units for both LSTM layers is $feature\_size$ and the input shape for the first layer is (33, 1), leading to a total of 4620 trainable weights for the first layer. Due to the first layer returning its sequences, the second layer's input shape is (33, 33) leading to a total of 8844 for the second layer. The third layer is a dense layer, whose weights consist of a kernel of size $(input\_dim * units)$ and a set of weights of size $units$, making the number of weights for the dense layer 1122. This leads to a total number of trainable weights. The output vector of shape ($feature\_size$, 1) then holds the predicted features, where every value in this vector represents one feature. The network uses a batch size of 32. Increasing the batch size tends to cause the network to converge slower, which can cause problems when dealing with users with few actions. On the other hand reducing the batch size quickly slows down the network significantly. The network is first trained on the supplied training data, always trying to optimize for the lowest loss value, calculated by the mean squared error function (mse). This function measures the average of the squares of the differences between actual and predicted values, giving an approximation of the deviation from the expected value. The mean squared error is calculated by using the following formula with $n$ being the length of the vector that was used as its input, $x$ being the predicted vector and $y$ being the actual vector:

$$mse = (\sum_{i=0}^{n-1} (x_i - y_i)^2)/n \tag{4.3}$$

The weights are adjusted during the training process to minimize the $mse$ over the training set, one batch (of size 32) at a time. This is done by using the "adam" optimizer. The learning rate of this optimizer was set to 0.001. Training is repeated for 25 epochs. Because too many iterations of the optimizer might lead to overfitting, the training process was stopped at 25 epochs. Too little iterations, however, results in higher error values.

This is also the number of epochs that is commonly used, making it a good choice when no other number of epochs was proven to work better. In Chapter 5 some experiments regarding changing the epoch size are done. As another measure to prevent overfitting, a dropout factor of 0.5, and a recurrent dropout factor of 0.2 is used for both LSTM layers. A dropout factor, which randomly drops certain nodes in the network, and a recurrent dropout factor that randomly drops out vectors between states, were shown to prevent overfitting in [SHK$^+$14]. Note that these parameters are not perfect and they are all chosen because they are either standard values in many projects (batch size and epochs) or because they are recommended (dropout). If a parameter has not been mentioned here, the value of that parameter is Keras' default value for that parameter. The system's goal is to identify outliers, which are then labeled as possible cyber-security attacks. Unfortunately, because the data set is unlabeled, no objective measure of how good the system is at finding actual cyber-security attacks exists, disallowing the optimization of the system's parameters based on this performance measure.

Because training LSTM networks requires many experiments with different setups and architectures, the decisions was made to use only 0.1% of the users in the data set for Chapter 5. However, in order to get a clear idea of the system's performance and to measure the effectiveness of the system, 1% of the users in the data set were used in Chapters 6 and 7. The users are sorted alphabetically before taking the first x% of human users.

## 4.5 Training

After preprocessing, the training data is used as input for the networks. For performance reasons, a single network is created, which is then used as a template for every user. New weights are created for every user, which are then applied to the base network. The network is trained by inputting a feature vector in the training set sequence and having it predict the next feature vector (see Table 4.5), after which the *mse* over these two vectors is calculated. This is done for every feature vector in the training set in batches of 32. After every batch the weights are updated based on the errors observed over that batch. This is repeated for all batches in the training set.

Table 4.5: The network's input vector and the vector it's supposed to predict

| Input | Target Prediction |
|---|---|
| $x_0$ | $x_1$ |
| $x_1$ | $x_2$ |
| $x_2$ | $x_3$ |
| ... | ... |
| $x_{n-2}$ | $x_{n-1}$ |
| $x_{n-1}$ | $x_n$ |

## 4.6 Testing

After training, the network is applied to the test set. In Keras the batch size of the training set and the batch size of the test set have to be the same, this downplays the significance of single anomalies. For example, in a set of *batch_size* errors, one big anomaly is not as significant as 10 small anomalies. As such a method needs to be devised to test using batch size of 1 instead. This is done by creating another network, identical in structure but having a batch size of 1, and transferring the weights and states when tests occur. This has the same effect as changing the original network's batch size to 1 (for this application), but without all the performance losses.

First of all, the interquartile range (IQR) is calculated over the training set's errors in order to get a baseline for the test set. The IQR function attempts to find statistical outliers based on the median values of a distribution. This is done by calculating the medians of both the upper and lower half of a distribution, which are then called Q1 and Q3 respectively. The IQR is then equal to $Q3 - Q1$. Any values that lay outside of the ranges of the following functions, where $x$ is the input value, are then called outliers.

$$x < Q1 - 1.5IQR \tag{4.4a}$$

$$x > Q3 + 1.5IQR \tag{4.4b}$$

The error represents the difference between the predicted vector and the actual vector. Because of this, the higher the error, the bigger this difference. This also means that the first form of outlier (4.4a) is not interesting, as the model simply predicted the behavior very well, not pointing to anomalous behavior. Because the training set's errors should represent a regular sequence of actions by the user, any errors calculated over the test set that fall outside of the range calculated with the above functions are classified as anomalies.

After finding anomalies, the corresponding fragments of the original data should be investigated to find out if they represent cyber-security events. This issue occurs because when inputting solely features, the events they are based on are discarded. As network administrators will want to see the name of the user that is behind a found anomaly and may want to have the events investigated by a cyber-security expert, these anomalies are translated back into source events. This is done by storing the index of an anomaly as well as the user associated with the anomaly. The indices have a 1–to–1 correspondence to the source events, allowing for easy translation. This step can be skipped if, for example, no anomalies were found or only previously known anomalies were found.

## 4.7 Code

All the code was written in Python, using the [C+15] Keras deep learning wrapper's LSTM as the neural network. Default settings were used if not mentioned otherwise. TensorFlow [AAB+15] was used as the underlying library for Keras. Due to both the preprocessing/feature generation and the training/testing stages being very slow (as will be explained later in the evaluation section), especially when using big datasets, both of these operations have been parallelized. The first stage (preprocessing/feature generation) is a very CPU-dependent task, this work can be split over any number of CPU's, handling a single user per CPU at a time until all users have been processed. The second stage (training/testing) can be either run on the CPU (s) or GPU (s). Depending on the hardware of the computer the experiments are executed on, one of these will be faster, as will be discussed in Chapter 6. When using the CPU, TensorFlow itself will use all CPUs available to it, however when using the GPU, a problem arises. Because TensorFlow only requires a single GPU per neural network per process, and increasing the amount of GPUs per network does not speed up training or testing, there is no point in using multiple GPUs for the same network. Because of this, only a single GPU will be used per model in this thesis, which in this case is the template model. In order to make use of the other GPUs and to speed up computation, a method of splitting the work over all GPUs must be found. This can be done by splitting the work into multiple independent processes in order to parallelize the operations. This is done by having one root process splitting the to-do jobs between *n* processes. The total number of users is split over the sub-processes. These *n* processes all produce partial outputs (both anomalies and plots), that have to then be stitched together by the host process. The host process then produces the final output.

# Chapter 5

# Experiments

A number of experiments have been done in order to find the differences in effectiveness between different architectures. For all experiments in this chapter, only 0.1% of the data set has been used. Because this number is so small, it is not a perfect representation of the data set. However, while only 0.1% of users (13 users) are in this data set, these users together represent a total of 6117530 actions (around 0.6% of the total data set), meaning these users are more active than others. Even though these experiments are not done on 100% of the data set, significant differences in effectiveness are likely to persist in bigger data sets.

The base architecture that all the experiments are compared to is the architecture described in Chapter 4, with the base training parameters also being the same as those described in Chapter 4. To summarize, 2 layers of standard LSTMs with dropout enabled with a single Dense layer, using a batch size of 32 and 25 epochs for training.

## 5.1 Batch Size

Experiments regarding increasing or decreasing the batch size were done. Changing the batch sizes to 64 and 16 respectively. No significant differences were observed. The number of anomalies found were the same between all configurations in this experiment. The IQR values were also all the same. From this it can be concluded that changing the batch size to be bigger or smaller has little effect (at least in this portion of the data set and with these numbers). Seeing as increasing the batch size significantly increases performance, increasing it would be an easy way to speed up the system. However, increasing the batch size by too much has been shown to reduce the effectiveness of neural networks.

## 5.2 Epochs

Changing the number of epochs to 15 and 40 has also shown little difference in the results with the current setup. However, some experiments with changing the number of epochs have been done on a larger set of users (100) with a very low number of actions ( 1500 per user). The architecture was also different, having no dropout rate on the LSTM layers. This showed some significant differences in the results. Figures 5.1, 5.2 and 5.3 show significant differences between the highest error value and the average error value of a batch. This is likely because epoch sizes that low tend to have a big effect when given little training data. The effect of a high epoch size is to essentially train multiple times on the same data. When there is a lot of data available, this is not necessary, instead causing the network to overfit on the input data it has. The differences between these two test scenarios can likely be explained by the first scenario having a dropout rate, significantly reducing the amount of overfitting a higher epoch size brings, and also having a lot more input data, making a lower epoch size have less effect as well.

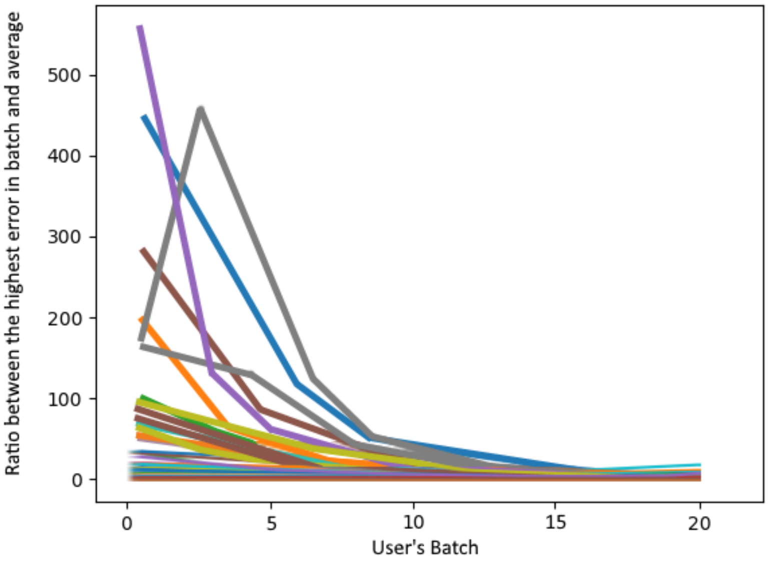

Figure 5.1: The ratio between the highest error event in the batch and the average error in the batch. Every line represents a user. Amount of batches normalized against the user with the most batches and cut off at 20 batches. Using 25 epochs.

## 5.3 Shared weights

The possibility of using one neural network for all users was explored. However, this introduced some problems. One of these is that the total number of actions for all users was not the same. This leads to the system weighing the actions of high-volume users heavier than those of low-volume users, leading to an unfair representation of the "average" user's behavior. Another problem was that performance was significantly worse. Parallelization can not be applied as the single network has to be kept in memory for the entire process.

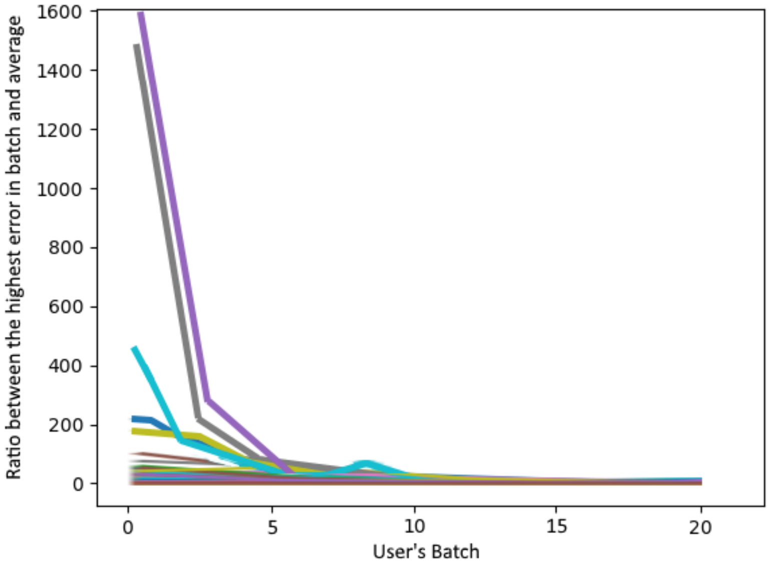

Figure 5.2: The ratio between the highest error event in the batch and the average error in the batch. Every line represents a user. Amount of batches normalized against the user with the most batches and cut off at 20 batches. Using 15 epochs.

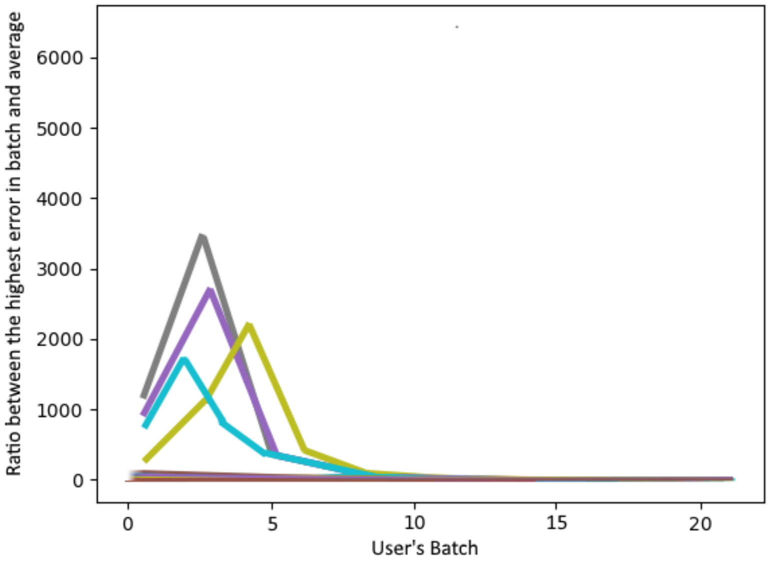

Figure 5.3: The ratio between the highest error event in the batch and the average error in the batch. Every line represents a user. Amount of batches normalized against the user with the most batches and cut off at 20 batches. Using 8 epochs.

The network also gravitates towards learning the behavior of "average" users. It will try to find a middle ground in the behavior of different types of users (sysadmins vs users that rarely log in), never really learning a single user's behavior well enough to find slight deviations. It will then accept users such as sysadmins having a high error value as normal, which could be very dangerous if such an account gets compromised.

## 5.4 GRU cell

Usage of a different RNN cell has also been explored. One of these different RNN cells is the GRU cell, introduced in [CVMG$^{+}$14]. The differences between an architecture using a LSTM cells and an architecture using GRU cells are slight. As [CGCB14] showed, the performance of GRU cells and LSTM cells are very similar when it came to the area of polyphonic music modeling and speech signal modeling. As such, it is likely that their performance on the area of cyber-security is also similar.

In order to get an idea of how much the error between the predicted feature vector and the actual feature vector deviates from that user's mean error, a function has been created with which this can be calculated. When replacing 1.5 with $y$ in equation 4.4b, with $x$ being the mean squared error (function 4.3) between the predicted feature vector and the actual feature vector, solving for $y$ gives the following function:

$$y = (x - Q3)/IQR \tag{5.1}$$

This $y$ value is plotted for the different architectures (GRU and LSTM with the parameters described above), which can be seen in Figure 5.4. This figure shows the GRU and LSTM deviations being fairly close. In the lowest group (from 0 to 18) the LSTM seems to be consistently higher than the GRU, however, at this scale the difference isn't that big. If you draw lines through these points, the lines run parallel (when viewing the plot non-logarithmically), showing that the architectures find similar anomalies. In later groups, the GRU has consistently higher $y$ scores than the LSTM. This can point to it flagging actual anomalies as more anomalous (pointing to it being better at recognizing them), or it labeling non-anomalous actions as more likely to be anomalies. Because the data set is unlabeled there is unfortunately no way to find out. Figure 5.5 again shows that the architecture using LSTMs has a higher error value for most batches, also peaking higher. The LSTM architecture also flagged 0.7% more events as anomalies. Again conclusions can not be drawn regarding which result is more effective but it can be seen that they do differ somewhat.

## 5.5 RNN cell

Using standard RNN cells instead of LSTM cells showed a significant difference. The parameters to the RNN were the same as those to the LSTM network. The RNN cells seem to have significantly higher errors in the middle group (18 to 35), while having significantly lower errors in the last group, as can be seen in Figure 5.6.

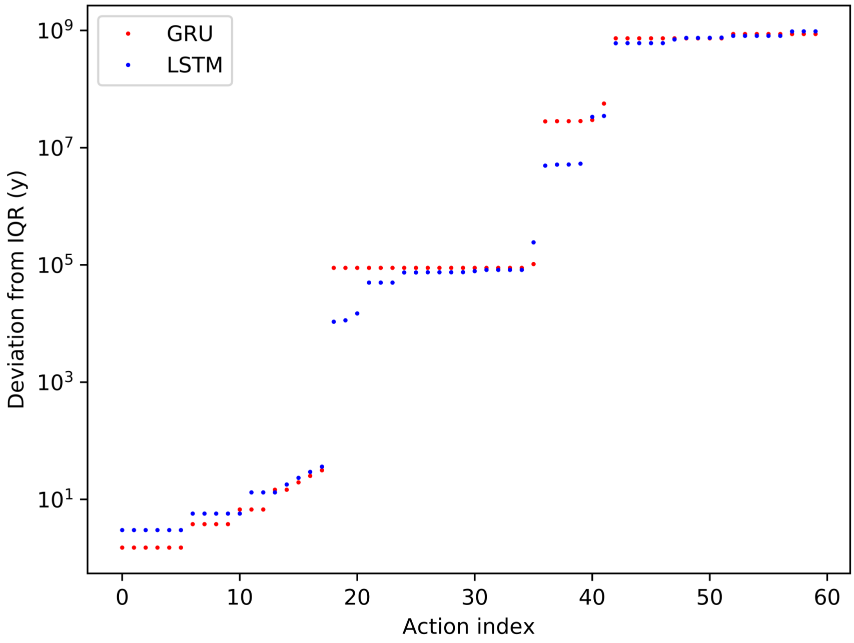

Figure 5.4: The deviations $y$ (function 5.1). Where the IQR is being calculated over the training set and $x$ is the mean squared error between a predicted feature vector and the actual feature vector. Combining the last 5 actions for each user, over 0.1% of all users (sorted). Comparing GRU and LSTM.

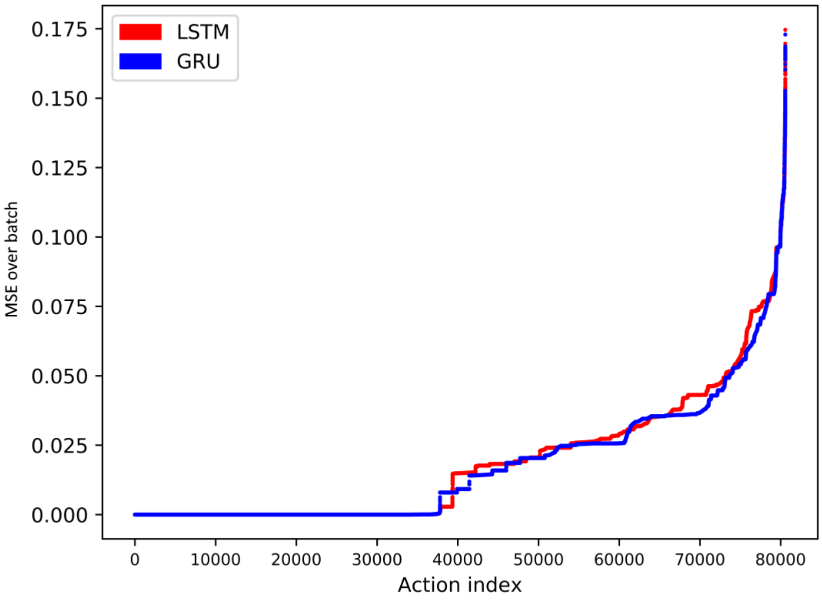

Figure 5.5: The error of all users' actions combined, calculated using the mse (sorted).

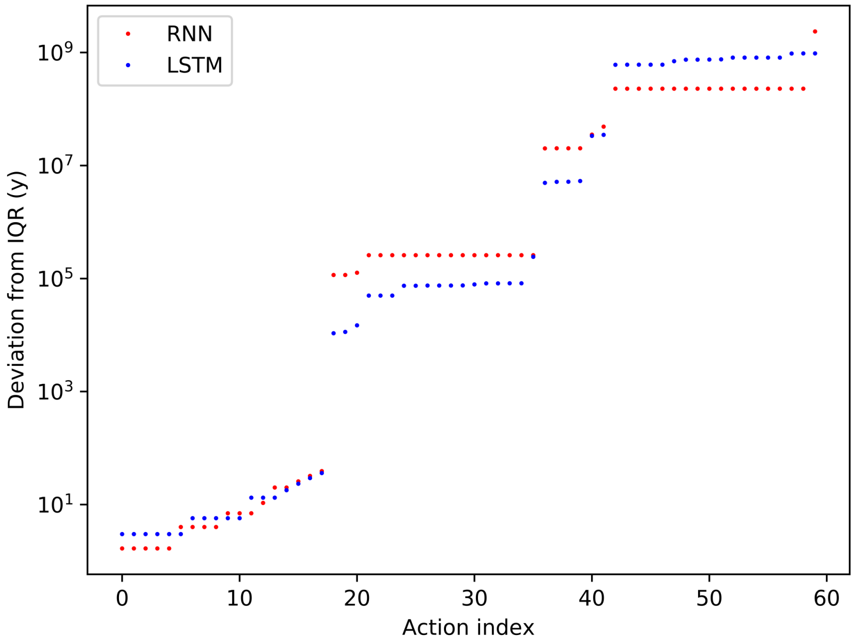

Figure 5.6: The deviations $y$ (function 5.1). Where the IQR is being calculated over the training set and $x$ is the mean squared error between a predicted feature vector and the actual feature vector. Combining the last 5 actions for each user, over 0.1% of all users (sorted). Comparing plain RNN and LSTM.

The RNN cells also flagged less events as anomalous, flagging 21% less as such. Once again no clear conclusion can be drawn as to which one is more effective, however, LSTMs have been shown to be superior to regular RNNs in many fields, suggesting that this may also be a case of the LSTM cells outperforming the RNN cells at flagging anomalies.

# Chapter 6

# Performance Evaluation

There are three major stages to the evaluation (preprocessing, training/testing, post-processing) as explained in Chapter 4. The bottleneck in all stages is the CPU, because of this all tasks were ran on a computer with 1TB of ram, 20 Intel Xeon E5–2650v3 CPUs running at 2.30GHz with 40 threads and 8 dual-gpu boards containing two NVIDIA Tesla K80 GPUs each with 11.5GB of memory. The training/testing stage can be ran on the GPU as well, shifting the bottleneck to the GPU. Because of this, this stage is either very CPU- or GPU-intensive depending on which of the two is chosen.

In order to get an idea how long running everything on 100% of the data set would take, two different percentages have been used: 0.1% and 1%. Keep in mind that a lot depends on the way things are programmed and code optimizations might significantly reduce the amount of time the process takes. The code used in this thesis was not optimized in that aspect. It was only used to support the hypothesis that real-time anomaly detection is feasible on the hardware available to big computer networks. Doing preprocessing took 2h45m14s for 0.1% of the data while it took 4h57m for 1% of the data, both using 10 CPUs. Scaling this up linearly would put the duration of preprocessing the entire data set at about 250 hours, also using 10 CPUs. Doing the training/testing stage with 16 GPUs took 43h14m53s on 0.1% of the data set, while 1% took 172h25m. Since 1% of the dataset contains 126,322,330 feature vectors, and 100% of the data set should contain 1,051,430,459 rows, doing training/testing for 100% of the data set should take approximately 1435h5m30s using GPUs. Using 20 CPUs instead takes about 152h35m on 1% of the data set. This means that on in our scenario running the system on the CPU is approximately 12% faster. Results may vary based on the number and architecture of CPUs or GPUs the system is executed on, making either CPUs or GPUs the faster option. The post-processing stage generally only takes roughly 5h30m, not varying much between data set sizes as all users need to be iterated through regardless and no other heavy CPU work is being done. Adding all of these times together leads to the following results. The entire process takes 48h30m7s for 0.1% of the data set, 179h52m for 1% of the data set and approximately 1617h23m40s for 100% of the data set. See Table 6.1 for an overview. 1% of the data set contains 126,322,330 feature vectors, meaning the system can handle training at about 198 rows per second on GPUs and 221 rows per second on CPUs, making this a very good fit for real-time training. The actual testing stage (without training) takes even shorter, generally taking about 1/100th of the time the training stage took for that user. This makes a system that learns and detects anomalies in real-time feasible. A system that does not continue learning after the initial training would taken up even less resources. This means a system that learns periodically instead of in real-time is even more technically feasible.

Table 6.1: The time every stage takes, by data set size

| | 0.1% | 1% | 100% |
|---|---|---|---|
| Preprocessing | 2h45m14s | 4h57m | ∼250h |
| Training/Testing | 43h14m53s | 172h25m | ∼1435h5m30s |
| Post-processing | 5h20m | 5h20m | 5h20m |
| Total | 51h20m7s | 182h42m | ∼1690h25m30s (∼70 days) |

# Chapter 7

# Results

During the training and testing stages, a number of plots have been made in order to visualize any outliers, along with a list of any anomalies the network has found. Note that all plots in this chapter are based on 1% of the users.

The *y* value resulting from function 5.1, where the IQR is calculated over the training set, can be seen in Figure 7.1. As can be seen in the figure, there are quite a few outliers, some of which having errors that fall far beyond the cutoff value of 1.5. From this, it can be concluded that at least some outliers are being found. The *y* values also tend to be very low, with around 90% being 1.0 or lower. This shows that the network is relatively successful at modeling user behavior, as its predictions' errors are very close to the training set's errors. A network that is unsuccessful at this would have a lot of high *y* as a result of the bad predictions.

Focussing on the highest offending users allows us to see more clearly why the network thought certain users were deemed anomalies and whether the network may have been right. The highest offending users are the users responsible for the highest offending actions, which are the actions with the highest errors compared to the predictions the network made.

In Figure 7.2 a closer look is taken at the time since the last network access for the top 10 highest offending users. This clearly shows some very big deviations from users' times since their last network access. For example, U41 consistently has a very low time since last access, as can be seen from the box plot being very small and concentrated around that area. However, there is a significant deviation from this user's normal behavior, which the network has probably flagged as a deviation. The same goes for other users like U102 and U72, changing their behavior a lot, while "normal" users like U34, U107 and U119 keep their time since last access consistent. U41's pause only took 2 weeks, which can easily be explained by a vacation, a period of sickness or any other reason for taking a short break. However, this is something that is still worth investigating, as this could also be an abandoned account being picked up by a different (potentially malicious) user.

The highest offending (highest mean squared error value) feature vector's predicted vs actual action are shown in Table 7.1. In this table (and following tables) features that are integers and not floats have been depicted as

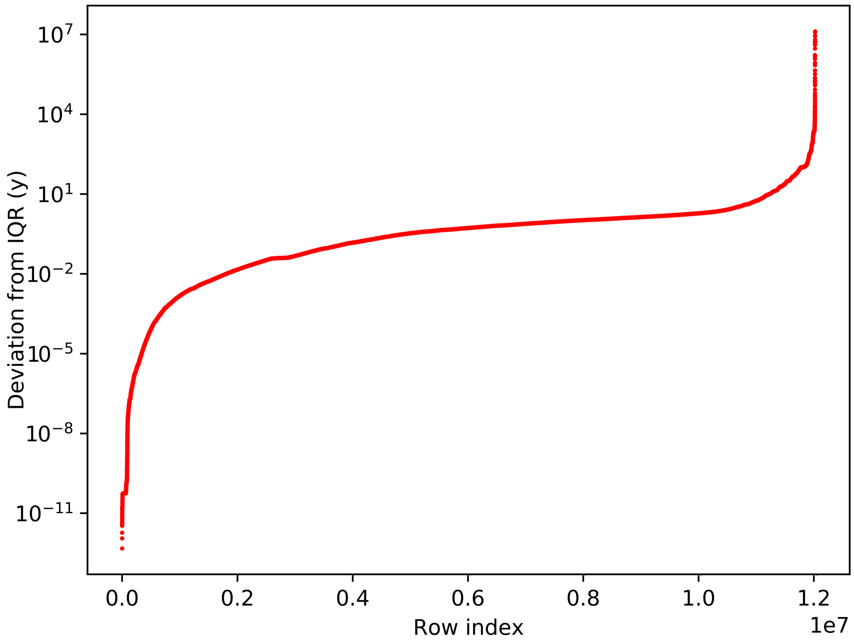

Figure 7.1: The deviations $y$ (function 5.1) of the mean squared errors of all predicted test set feature vectors compared to the actual feature vectors. The IQR has been calculated over the training set using the same method of predicted vs actual mean squared error.

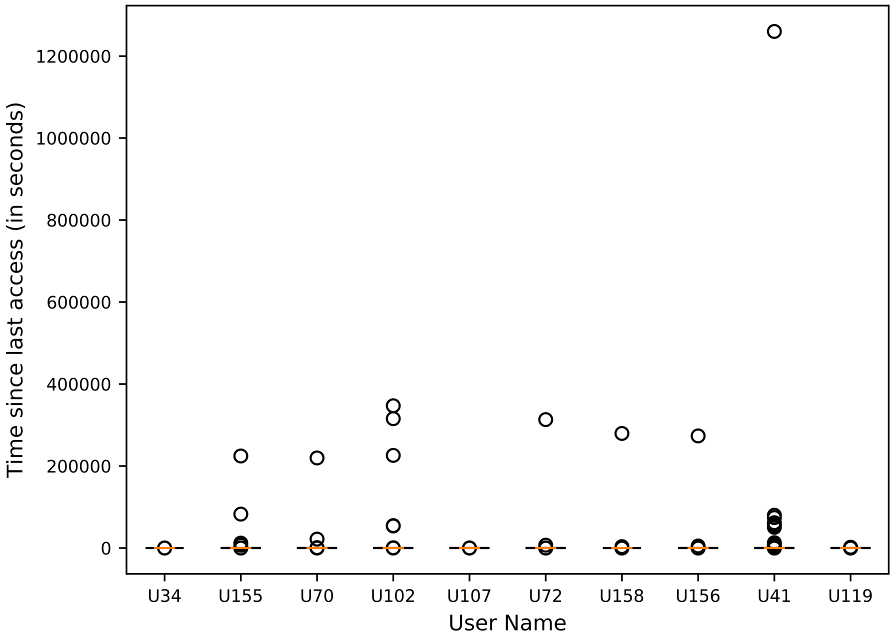

Figure 7.2: The top 10 highest offenders' seconds since last access. Actions with the highest errors aren't necessarily high because of the time since last access metric (U34, U107).

integers, as all but the *percentage_failed_logins* and *time_since_last_access* features are integers. Any predictions made by the network are trimmed to 3 decimal points.

As can be seen, the action in Table 7.1 itself isn't very weird, simply using a different method of authentication, a different method of logging in and a different authentication orientation. These methods themselves are not inherently anomalies, but the network learned that these actions are rarely made by the user, assigning a value of 0.000 to *auth_type_0* (NTLM), *logon_type_7* (REMOTEINTERACTIVE) and *auth_orientation_2* (TGS). The network also assigned values of 0.999 or higher to a single action in all of these enums, pointing to the user almost always logging in using these exact methods. When compared to the user's previous logins in Table 7.2, the last action really stands out as different.

Table 7.1: The predicted features vs the actual features of the top offending feature set by U399 (precise to 3 decimals)

| Label | Actual | Predicted |
|---|---|---|
| time_since_last_access | 0.060 | 0.011 |
| domains_delta | 0 | 0.000 |
| dest_users_delta | 1 | 0.000 |
| src_computers_delta | 1 | 0.000 |
| dest_computers_delta | 1 | 0.000 |
| percentage_failed_logins | 0.001 | 0.000 |
| success_failure | 1 | 0.999 |
| auth_type_0 | 0 | 0.000 |
| auth_type_1 | 0 | 1.000 |
| auth_type_2 | 0 | 0.000 |
| auth_type_3 | 0 | 0.000 |
| auth_type_4 | 0 | 0.000 |
| auth_type_5 | 0 | 0.000 |
| auth_type_6 | 0 | 0.000 |
| auth_type_7 | 0 | 0.000 |
| auth_type_8 | 0 | 0.000 |
| auth_type_9 | 1 | 0.000 |
| auth_type_10 | 0 | 0.000 |
| logon_type_0 | 0 | 0.000 |
| logon_type_1 | 0 | 0.000 |
| logon_type_2 | 0 | 0.000 |
| logon_type_3 | 0 | 0.999 |
| logon_type_4 | 0 | 0.000 |
| logon_type_5 | 0 | 0.000 |
| logon_type_6 | 0 | 0.000 |
| logon_type_7 | 1 | 0.000 |
| logon_type_8 | 0 | 0.000 |
| auth_orientation_0 | 0 | 0.999 |
| auth_orientation_1 | 0 | 0.000 |
| auth_orientation_2 | 1 | 0.000 |
| auth_orientation_3 | 0 | 0.000 |
| auth_orientation_4 | 0 | 0.000 |
| auth_orientation_5 | 0 | 0.000 |

The number two highest offending action can be seen in Table 7.3. This action by itself is again not very strange, however, the actual column looks a lot like the one in Table 7.1. They only differ in *percentage_failed_logins*, a feature that is calculated over previous actions as well, which explains the difference. The fact that the top two highest offending actions are (almost) exactly the same is definitely worth investigating. This could point to an

Table 7.2: The highest offending user's last 30 logins before the anomaly and the anomaly last

| time (ms) | source user@domain | destination user@domain | source computer | destination computer | authentication type | logon type | authentication orientation | success/failure |
|---|---|---|---|---|---|---|---|---|
| 3769690 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769706 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769720 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769728 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769732 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769734 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769750 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769753 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769754 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769755 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769769 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769811 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769849 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3769861 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770055 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770057 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770058 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770066 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770112 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770113 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770114 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770115 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770116 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770144 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770146 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770148 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770152 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770154 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770156 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770158 | U399@DOM5 | U400@C832 | C832 | C832 | Negotiate | Interactive | LogOn | Success |
| 3770158 | U399@DOM5 | U400@C832 | C832 | C832 | MICROSOFT_AUTHENTICATION_PA | REMOTEINTERACTIVE | TGS | Success |

actual cyber-security attack that infects different users and uses the same method of attack. Once again this action differs a lot from what the network predicted. Only the *auth_type* feature was predicted as a possibility. However, it is probably just a coincidence that the possible cyber-security attack uses the same authentication type that user U155 regularly uses. When compared to the user's previous actions, which can be seen in Table 7.4, the offending action really stands out. The user's previous actions also seem strange on their own, attempting to log in to the same computer 24 times in a row every few seconds. The user then switches to a different computer and tries the same thing 4 more times before giving up and finally logging in successfully. Many anomalies like this have been found. A user often logs in after a really long time relative to their other login times or significantly changes their behavior by logging in using methods rarely or never used before. This indicates that the network is doing a good job at recognizing the user's behavior and finding anything that deviates from it.

Table 7.3: The predicted features vs the actual features of the second highest offending feature set by U155 (precise to 3 decimals)

| Label | Actual | Predicted |
|---|---|---|
| time_since_last_access | 0.000 | 0.000 |
| domains_delta | 0 | 0.000 |
| dest_users_delta | 1 | 0.000 |
| src_computers_delta | 1 | 0.002 |
| dest_computers_delta | 1 | 0.000 |
| percentage_failed_logins | 0.993 | 0.000 |
| success_failure | 1 | 0.997 |
| auth_type_0 | 0 | 0.000 |
| auth_type_1 | 0 | 0.455 |
| auth_type_2 | 0 | 0.000 |
| auth_type_3 | 0 | 0.000 |
| auth_type_4 | 0 | 0.003 |
| auth_type_5 | 0 | 0.000 |
| auth_type_6 | 0 | 0.004 |
| auth_type_7 | 0 | 0.000 |
| auth_type_8 | 0 | 0.000 |
| auth_type_9 | 1 | 0.547 |
| auth_type_10 | 0 | 0.000 |
| logon_type_0 | 0 | 0.002 |
| logon_type_1 | 0 | 0.000 |
| logon_type_2 | 0 | 0.003 |
| logon_type_3 | 0 | 0.000 |
| logon_type_4 | 0 | 0.000 |
| logon_type_5 | 0 | 0.000 |
| logon_type_6 | 0 | 0.000 |
| logon_type_7 | 1 | 0.009 |
| logon_type_8 | 0 | 0.988 |
| auth_orientation_0 | 0 | 0.540 |
| auth_orientation_1 | 0 | 0.008 |
| auth_orientation_2 | 1 | 0.000 |
| auth_orientation_3 | 0 | 0.000 |
| auth_orientation_4 | 0 | 0.000 |
| auth_orientation_5 | 0 | 0.453 |

Table 7.4: The highest offending user's last 30 logins before the anomaly and the anomaly last. A question mark means the value was not in the dataset, possibly part of the anonymization process.

| time (ms) | source user@domain | destination user@domain | source computer | destination computer | authentication type | logon type | authentication orientation | success/failure |
|---|---|---|---|---|---|---|---|---|
| 4175329000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175330000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175359000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175360000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175389000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175390000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175420000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175450000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175480000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175498000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175510000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175540000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175570000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175571000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175600000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175629000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175630000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175637000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175660000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175672000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175673000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175690000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175720000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175750000 | U155@? | U155@? | C1746 | C1746 | ? | TGT | ? | Fail |
| 4175759000 | U155@? | U155@? | C529 | C529 | ? | TGT | ? | Fail |
| 4175759000 | U155@? | U155@? | C922 | C922 | ? | TGT | ? | Fail |
| 4175761000 | U155@? | U155@? | C529 | C529 | ? | TGT | ? | Fail |
| 4175761000 | U155@? | U155@? | C922 | C922 | ? | TGT | ? | Fail |
| 4175765000 | U155@DOM1 | C922$@DOM1 | C922 | C922 | ? | AuthMap | ? | Success |
| 4175765000 | U155@DOM1 | U155@DOM1 | C922 | C922 | Network | LogOff | ? | Success |

# Chapter 8

# Conclusions

In this thesis the feasibility of anomaly detection by using recurrent neural networks was investigated. Chapter 6 shows that it is possible to build a monitoring system that detects anomalies in real-time using recurrent neural networks, as well as it being possible to run the same system on a previously captured data set. As discussed in Chapter 6, the network can handle training at 198 actions per second by using 16 GPUs and 221 actions per second by using 20 CPUs. Most network's administrators will be able to run this system in real-time as it requires minimal resources, especially when not training in real-time. Especially since adding more or faster GPUs/CPUs is an easy way to increase the capacity of the network. Knowing that the 26,301 users (including computer users) at Los Alamos National Laboratory that contributed to the data set generated 1,051,430,459 events in 58 days, this leads to 329 actions per second for 26,301 users. In this scenario the system would have to store the weights of 26,301 users. Each of the 14,586 trainable weights is being represented by a python float32. This data type is 4 bytes per float32, which means that the total size of the weights in memory is 4 * 14,586 = 58344 bytes or around 58 KB per user. In the Los Alamos data set the total size of all users' networks would be a very manageable 1.4 GB. Adding around 8 GPUs or 8 CPUs to the setup described in Chapter 6 should be enough to handle both training and testing in real-time for the Los Alamos computer network, making this a very feasible method of anomaly detection. From this it can be concluded that using a recurrent neural network for anomaly detection is technologically feasible at least in this scenario.

One problem is finding the optimal network architecture and training it. Very few parameters can be chosen with absolute certainty or with results backing them up as the best parameters. As the data set is unlabeled, no objective measure of how good the system is at finding actual cyber-security attacks exists, making it very hard to find the optimal network architecture. This also prevents the choosing of the best possible features. Even though this will be something that will (most likely) always remain an issue when it comes to unlabeled data sets, more research could be done into the best configurations for a given problem without involving labels. This area (meta learning) is an area that is very active at the moment. There is some progress being made when it comes to supervised learning such as in [ZL16], however, no such advancements have been made yet in the area of unsupervised learning. For example into the area of generating a good model and setup for a given problem or for selecting good features for training from given data.

Another problem is knowing whether the flagged actions are actually intrusion attempts. This is and always will be something that only domain experts are able to verify. While from Chapter 7 it can be concluded that the actions that have been investigated (Table 7.1 and Table 7.3) do indeed look like anomalies, there is no certainty over whether the actions the network flagged as anomalous are all anomalies and whether all anomalous actions have in fact been detected. Because of this, the system can only be a tool for domain experts to reduce the number of cases that need to be closely investigated, not one that can completely replace them. However, the system is shown to pick out very anomalous behavior (and a possible multi-user cyber-security attack), showing that this approach is very effective on top of being technologically feasible.

# Bibliography

[AAB$^+$15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[C$^+$15] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[Can98] James Cannady. Artificial neural networks for misuse detection. In *National information systems security conference*, pages 368–81, 1998.

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[CVMG$^+$14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[GSS02] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3(Aug):115–143, 2002.

[GWD14] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[Hea08] Jeff Heaton. *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HSM$^+$00] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951, 2000.

[JZS15] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350, 2015.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Ken15] Alexander D. Kent. Cybersecurity Data Sources for Dynamic Network Research. In *Dynamic Networks in Cybersecurity*. Imperial College Press, June 2015.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LS$^+$98] Wenke Lee, Salvatore J Stolfo, et al. Data mining approaches for intrusion detection. In *USENIX Security Symposium*, pages 79–93. San Antonio, TX, 1998.

[MVSA15] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. Long short term memory networks for anomaly detection in time series. In *Proceedings*, pages 89–94. Presses universitaires de Louvain, 2015.

[OH15] Tomas Olsson and Anders Holst. A probabilistic approach to aggregating anomalies for unsupervised anomaly detection with industrial applications. In *FLAIRS Conference*, pages 434–439, 2015.

[Ola15] Christopher Olah. Understanding lstm networks. *GITHUB blog, posted on August*, 27:2015, 2015.

[PES01] Leonid Portnoy, Eleazar Eskin, and Sal Stolfo. Intrusion detection with unlabeled data using clustering. In *In Proceedings of ACM CSS Workshop on Data Mining Applied to Security (DMSA-2001*, pages 5–8, 2001.

[PGCB13] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013.

[R$^+$99] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, pages 229–238, 1999.

[RLM98] Jake Ryan, Meng-Jang Lin, and Risto Miikkulainen. Intrusion detection with neural networks. In *Advances in neural information processing systems*, pages 943–949, 1998.

[SHK$^+$14] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[YCV$^+$15] Kaisheng Yao, Trevor Cohn, Katerina Vylomova, Kevin Duh, and Chris Dyer. Depth-gated lstm. *arXiv preprint arXiv:1508.03790*, 2015.

[ZL16] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

# Appendices

# Appendix A

# Neural Network Code

```
import keras

model = keras.models.Sequential()
model.add(keras.layers.recurrent.LSTM(33, input_shape=(33, 1) batch_size=32,
        return_sequences=True, stateful=True, dropout=0.5,
        recurrent_dropout=0.2))
model.add(keras.layers.recurrent.LSTM(33, batch_size=32,
        return_sequences=False, stateful=True, dropout=0.5,
        recurrent_dropout=0.2))
model.add(keras.layers.core.Dense(33, activation="relu"))

optimizer = keras.optimizers.adam(lr=0.001)
model.compile(loss=’mean_squared_error’, optimizer=optimizer)
```