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

07/10/2017

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, leading to the conclusion that more research on the area of applying neural networks to unlabeled data sets is needed.

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 finding suspicious behavior requires a cyber-security expert to look at these logs, while also needing to be processed 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 specializes in 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 paper 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). In this thesis, 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 difference, 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 recently been a very active field of research, becoming even more active as the amount and complexity of the to analyse data increases. In [R+99], a lightweight way 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 $\mathrm{h_{t\text{-}1}}$ 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 σ is a sigmoid function used to squash its inputs into the [0, 1] range:

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

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

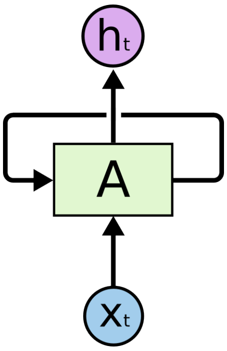

Figure 3.1: A recurrent neural network. From [Ola15]

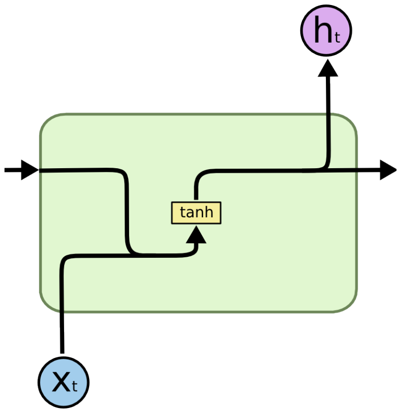

Figure 3.2: A single RNN cell. From [Ola15]

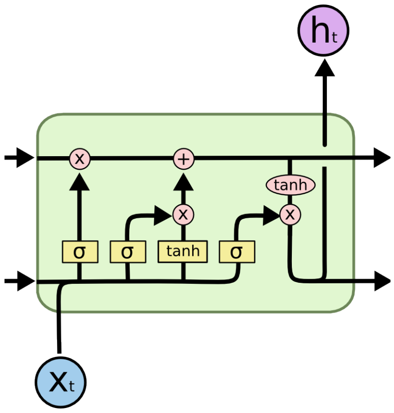

Figure 3.3: A single LSTM cell. From [Ola15]

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. This has the advantage of the network not having to choose what data to keep, as remembering is its default state, seeing as the same state keeps going on to the next iteration.

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 involve 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 running 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

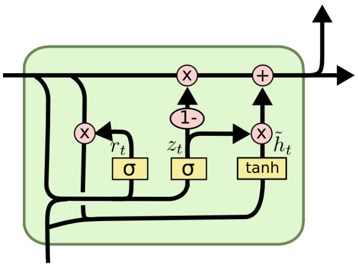

Figure 3.4: A single GRU variation cell. From [Ola15]

iteration's output vector. f_t can be obtained by using the following formula, 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 running the previous forget gate's result as well as the input gate through a matrix addition function. 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 allows them to be better suited for tasks involving series of data. This has lead to the LSTM architecture becoming the dominant RNN architecture. 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 Gated Recurrent Unit (GRU) [CVMG^{+}14] that combines the input and forget gates into a single so-called "update gate" (see figure 3.4). 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 data set that is being used in this thesis, there is a lot of data that is very essential that still requires some processing to extract (such as whether the user connected to a computer they haven'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 users tend to have different behavioral patterns in this data set and every other big network, the decision was made to train one network per user. Another option was explored, as explained in 5. After training on the training set, the network then runs on 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 has 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 amount of 150 actions per user has also been chosen in order for them to be included in the testing/training sets. The amount 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 amount 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 in addition to the time frame at which it was captured not being 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

Due to the size of the data set and the limited amount of time, the decision was made to use only 1% of the users for plots/results in this thesis. This means the first 1% of valid users, meaning only non-anonymous, human users, as explained in the previous section. Users are chosen regardless of the amount of actions they have (as long as this amount is higher than 150), and are simply sorted alphabetically, after which the first 1% are taken.

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 (one of enum)
6	logon type	What type of logon type was used (one of enum)
7	auth orientation	What type of authentication orientation was used (one of enum)
8	success or failure	1 if the login succeeded, 0 if it didn't

Table 4.3: One of encoding

Value	A	B	C
1	1	0	0
2	0	1	0
3	0	0	1

4.2 Features

As the raw data has some unneeded values that need to be transformed such as the domains and the time since the last access, features are constructed from the original rows. 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. 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 enum values (indexes 5,6,7) have been encoded using 1-of-encoding (see table 4.3 for an example), ensuring the values remain nominal. This gives each possible enum 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 enum value instead of possibly assigning them a real-numbered value. This brings the total length of the feature vector up to 33 (11, 9 and 6 possible values for the enums respectively).

The features have been chosen to represent the values that the network will probably be using. For example, the network is unlikely to keep track of a list of every computer the user logged into, but would probably be interested in knowing whether the user logged in to a computer they haven't previously logged in to. Additionally, the 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 split up. 70% is used for training and 30% is used for the test set. This is done separately for every user, making sure that each user has the same 70–30 split. The network expects only real-valued numbers from the range [0,1], however, because the features contain integer values larger than 1, these values have to be normalized to fit into this range. This is done by taking the maximum value for every column and dividing every value in that column by that maximum, linearly scaling every value down to the range [0,1]. This is done by applying the formula below to every column per user, where x is the input column and x' is the output column:

$$x' = x / \max(x) \tag{4.1}$$

This operation is performed on each user's training and test set before they are split up, ensuring that the scaling factor is the same for both sets. 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:

$$x' = \frac{1}{1 + x} \tag{4.2}$$

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 which transforms the data to the correct vector length. All 3 layers use an internal representation vector (and layer output) size of $feature\_size$, which is "between" the network's input and output sizes, both being $feature\_size$ as well (as suggested in [Hea08]). This leads to a total amount of 14,586 trainable weights. The output vector then consists of real values representing each feature separately. 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 function is calculated by using the following formula with n being the length of the input vector, 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 mse function is then applied to the predicted feature vector and the actual feature vector. 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. This value was chosen because increasing this value would introduce overfitting and reducing it would result in higher loss values overall. 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 input vectors, 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). Because of the use of unsupervised learning, no measure of how good the system actually is at detecting attacks exists. Because of this no objective measure of how good the system functions exists, and the system's parameters can not be optimized by using this measure. If a parameter has not been mentioned here, the value of that parameter is keras' default value for that parameter.

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 the template for every other network. Instead of creating a new network for every user, new weights are created that are then applied to the base network. The network is trained by inputting the a feature vector in the training set sequence and having it predict the next feature vector, 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 error value. This is repeated for all batches in the training set.

4.6 Testing

After training, the network is applied to the test set. There is a limitation requiring the use of the same batch size for both training and testing, which would downplay the significance of single anomalies. For example, in a set of *batch_size* losses, 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.

The losses from all of the test data are then collected, after which the interquartile range (IQR) is calculated over the training set's error values. 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.

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

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

In practice the first form of outlier (lower than IQR) will almost never be found, as that would mean an action so perfectly fits the user, it is an anomaly, which is very unlikely to point to an intrusion attempt. Because the training set's error values should represent a regular sequence of actions by the user, any error values in the test set that fall outside of the range calculated with the above functions are classified as anomalies.

After finding anomalies, they have to be translated into actual rows. 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 indexes 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 task can be split over any amount 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, Keras itself will use all CPUs available to it, however when using the GPU, a problem arises. Because Keras only uses a single GPU per neural network, only a single GPU can be used per model, which in this case is the template model. This can be solved 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 amount 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 of these experiments, only 0.1% of the data set has been used. Due to the number of experiments, a higher percentage would take significantly longer. 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 represent a higher amount of the data set than they themselves make up. 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. To summarize, 2 layers of standard LSTMs with a single Dense layer. All using a batch size of 32 and 25 epochs.

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 amount 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 epoch sizes to 15 and 40 has also shown little difference in the results with the current setup. However, some experiments with a number of different epoch sizes have been done on a set of users with

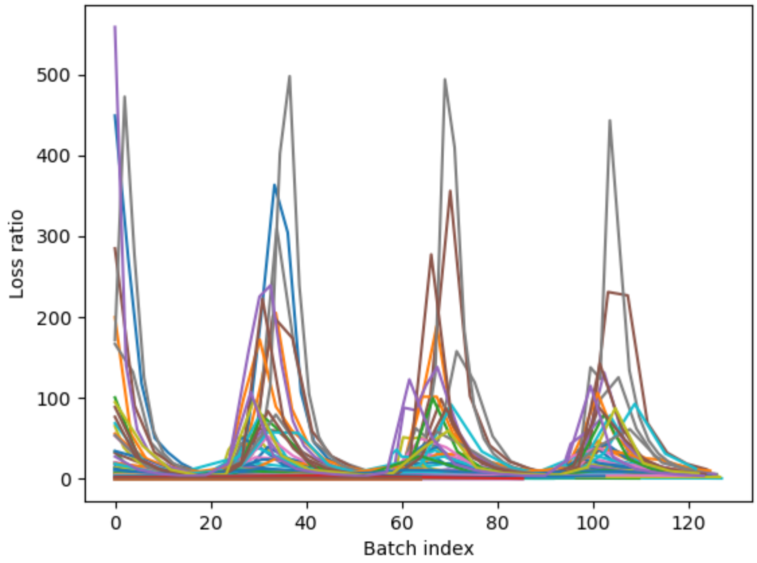

Figure 5.1: The ratio between the highest loss event and the average loss of a batch, using 25 epochs.

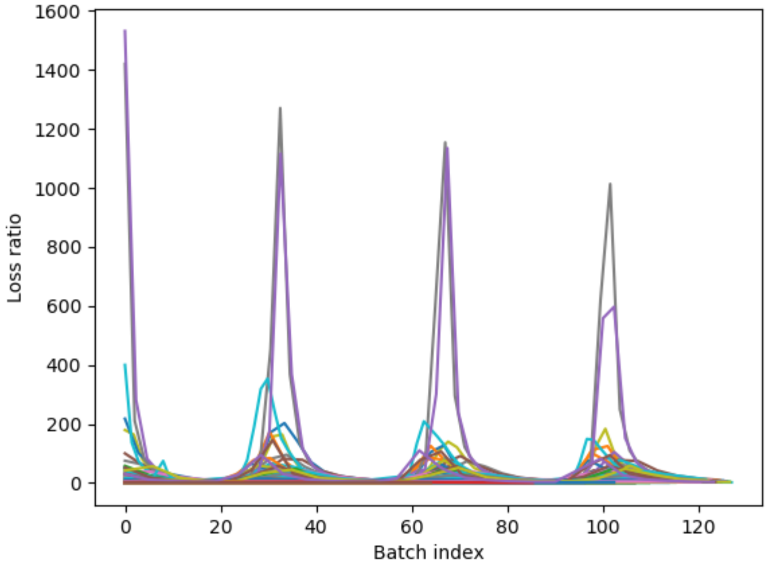

Figure 5.2: The ratio between the highest loss event and the average loss of a batch, using 15 epochs.

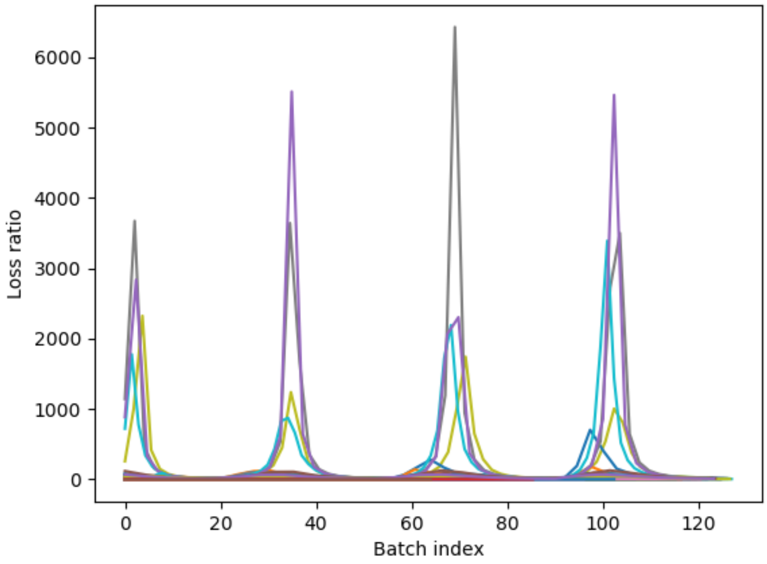

Figure 5.3: The ratio between the highest loss event and the average loss of a batch, using 8 epochs.

a very low amount 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 loss value and the average loss 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.

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 amount 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. 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.

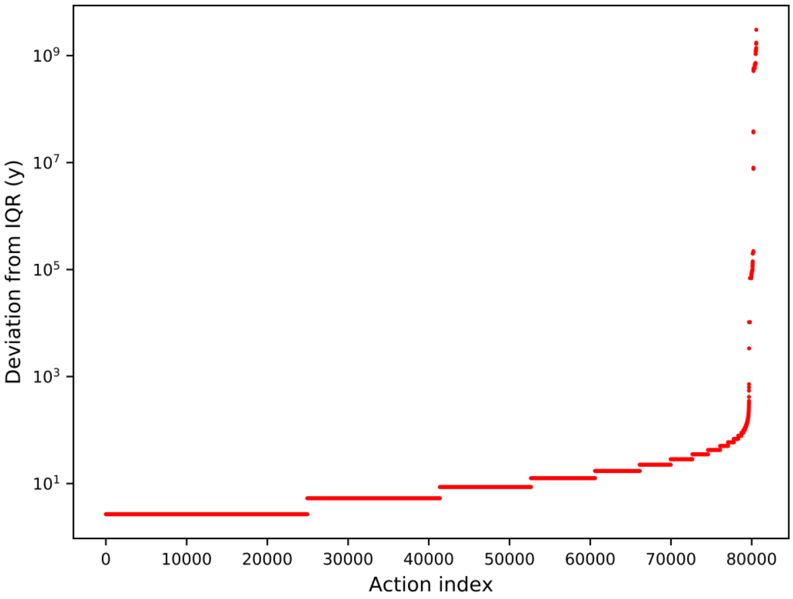

Figure 5.4: The deviations y (function 7.1) of the mean squared errors of all predicted test set feature vectors compared to the actual feature vectors. Using GRU cells.

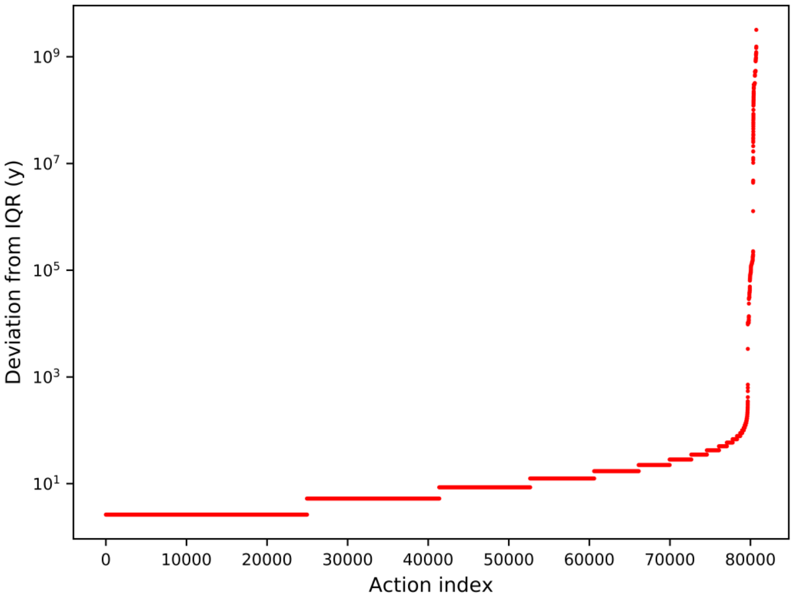

Figure 5.5: The deviations y (function 7.1) of the mean squared errors of all predicted test set feature vectors compared to the actual feature vectors. Using LSTM cells.

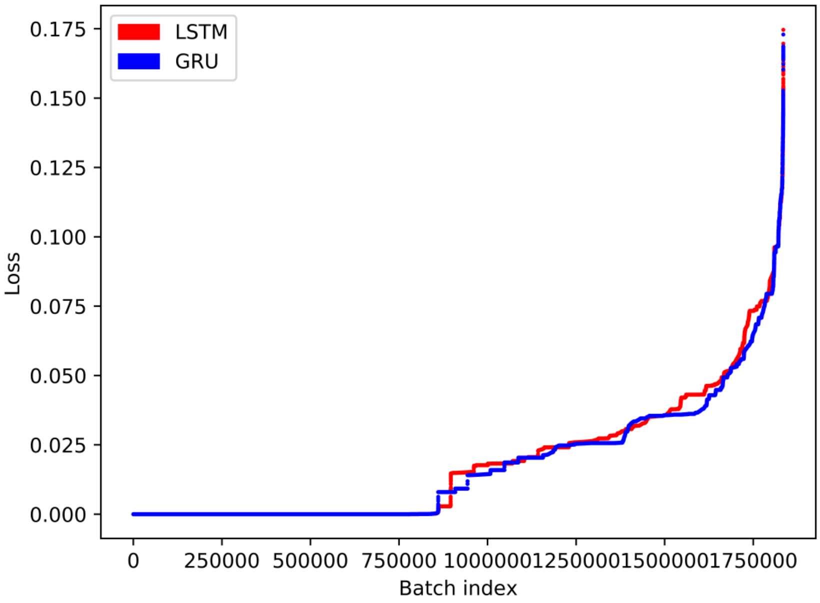

Figure 5.6: The loss calculated over each batch using the mse.

5.4 GRU cell

Using 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 were 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. Figures 5.4 and 5.5 show that the architecture using LSTM cells has more events with a high deviation than the architecture using GRU cells. This can point to it detecting more actual anomalies, or it labeling non-anomalous actions as anomalies. Because the data set is unlabeled there is unfortunately no way to find out. Figures 5.6 again shows that the architecture using LSTMs has a higher loss 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 an LSTM cells showed a significant difference. Just as with the GRU cells, the RNN cells have significantly lower losses, as can be seen when comparing figures 5.5 and 5.7. The RNN cells also flagged less events as anomalous, flagging 21% less as such. Once again no clear conclusion can be drawn from 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 finding

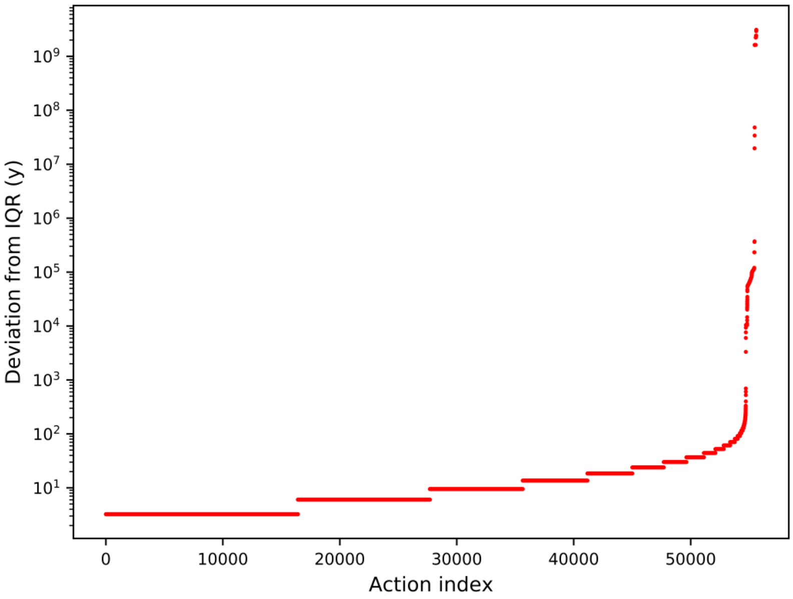

Figure 5.7: The deviations y (function 7.1) of the mean squared errors of all predicted test set feature vectors compared to the actual feature vectors. Using RNN cells.

anomalies.

Chapter 6

Evaluation

There are three major stages to the evaluation (preprocessing, training/testing, translating) as explained in Chapter 4. These three stages each have different bottlenecks. Because of this not all experiments were run on the same computer. Both the preprocessing and translating stages require the reading of the entire data set file. The whole file needs to be read because it has to be sorted by individual users before being able to output the features for a percentage of those users. This makes it a very RAM intensive task, requiring around 1TB of RAM. Preprocessing and translating only require CPU work after loading in the file, shifting the bottleneck to the CPU after loading it. The preprocessing stage allows for CPU parallelization, making more CPUs an easy way to improve performance. Because of the high RAM requirement, the first and third stages are run on a computer with 1.5TB of ram and 16 Intel Xeon E5–2630v3 CPUs running at 2.40GHz with 32 threads. Because the output of the preprocessing stage is significantly smaller than the data set (about 93% smaller), using a computer with less RAM for the training/testing stage is possible. Because of this the second task was 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. This stage can either be very CPU- or GPU-intensive depending on which method 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%. Doing preprocessing took 2h45m14s for 0.1% of the data while it took 4h57m for 1% of the data, both using 10 CPUs and including the reading of the file (at around 2h15m). 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 amount and architecture of CPUs or GPUs the system is executed on, making either CPUs or GPUs the faster option. The anomaly translation part generally only takes roughly 5 and a half hours, not varying much between data set sizes as all users need to be iterated through regardless and no other heavy CPU work is

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
Translating	5h20m	5h20m	5h20m
Total	51h20m7s	182h42m	∼1690h25m30s

being done. A significant amount of time is spent on loading the data set file again, taking around 2h15m as well. 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 network can handle about 198 rows per second on GPUs and 221 rows per second on CPUs, making this a very good fit for real-time anomaly detection. The actual testing stage (without training) takes even shorter, generally taking about 1/100th of the time the training stage took for that user, which would make a network that does not continue learning after the initial training even more feasible to run.

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 these plots and anomalies are all based on 1% of the users.

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 (equation 4.3) between the predicted feature vector and the actual feature vector, solving for y gives the following function:

$$y = (x - Q3)/IQR \tag{7.1}$$

This y value is then plotted, the result of which can be seen in figure 7.1. As can be seen in the figure, there are quite a number of outliers, some of which having IQR-scores that fall far beyond the cutoff value of 1.5. From this, it can be concluded that at least some outliers are being found.

As can be seen in figure 7.2, the IQRs tend to be fairly close to each other, meaning the mean losses are close to each other as well. This shows that the network is relatively successful at modeling user behavior, as the difference between the expected and actual action calculated by the loss function shows few big spikes. A network that is unsuccessful at this would have inconsistent IQRs as the losses would fluctuate more from user to user and would show higher values indicating 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.

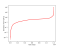

Figure 7.1: The deviations y (function 7.1) of the mean squared errors of all predicted test set feature vectors compared to the actual feature vectors. The IQR has been calculated over the training set using the same method of predicted vs actual mean squared error.

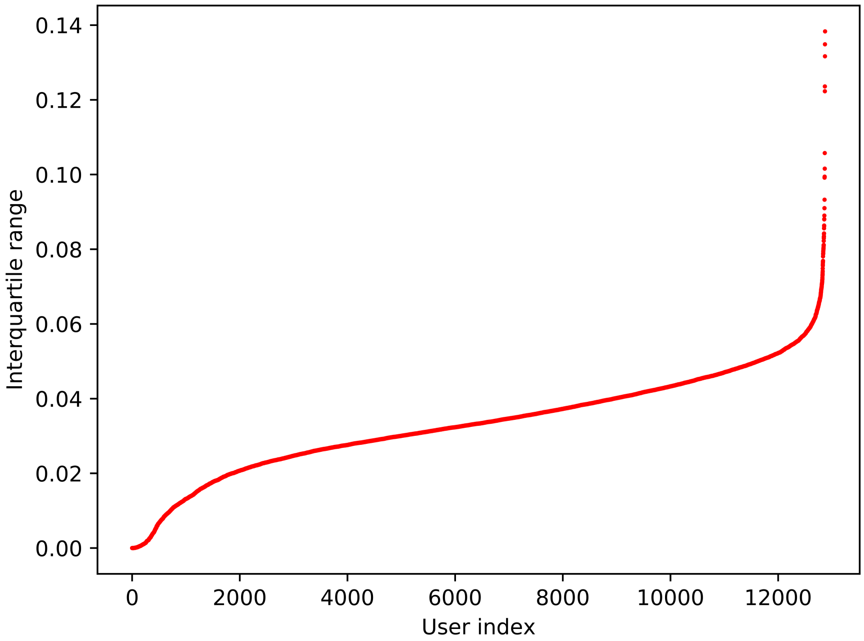

Figure 7.2: The IQR values calculated over the mean squared errors of all test set feature vectors and their predictions, by user. Note that for actual testing, the IQR is calculated over the training set, this is simply done to show the distribution of the losses of all users.

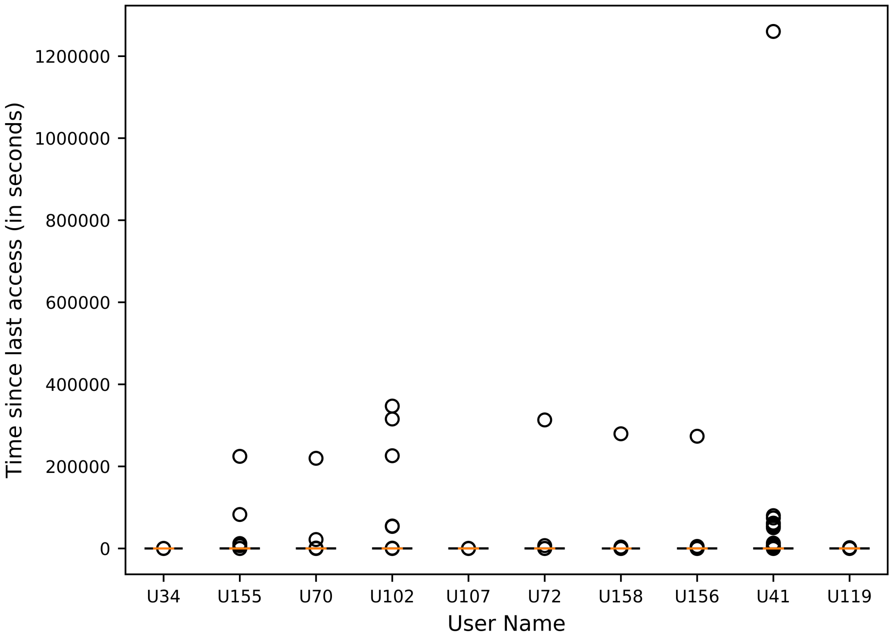

Figure 7.3: The top 10 highest offenders' seconds since last access.

In figure 7.3 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.

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 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 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.

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$_{A}UTHENTICATION_{P}A$	REMOTEINTERACTIVE	TGS	Success

As can be seen, the action 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. 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.

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. Seeing as the system can handle 198 actions per second by using 16 GPUs and 221 actions per second by using 20 CPUs in the setup that was used in this thesis, most networks will be able to run this system 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 should be enough to handle both training and testing in real-time for the Los Alamos 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.

However, knowing whether the found anomalies are actually intrusion attempts 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 (7.1, 7.2) do indeed look like anomalies, there is no certainty over whether the found anomalies are all anomalies and whether all anomalous actions have in fact been detected. Because of this, the network 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.

Another 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 measure of the accuracy of a network architecture can be made, making it very hard to find the optimal network architecture and also preventing 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.

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.

[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.

[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.

[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.