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 29/08/2017

Abstract

Due to the widespread usage and amount of attacks on computer networks, a fast and accurate method to detect attacks on these networks is an ever growing need. Due to their ability to learn and to process data quickly, an Artificial Neural Network (ANN) is an increasingly popular tool for this job. A lot of research has gone into either misuse detection where signatures of known intrusion techniques are found, or simple anomaly detection using labeled data sets. These methods are however not full solutions. New and unknown attacks can not be detected and large labeled data sets require experts to carefully examine every row, leading to very few of them being available.

In this thesis a Recurrent Neural Network (RNN) is applied to unlabeled data, attempting to model a user's behavior and to find any deviations from this model. As RNNs have the ability to process large sequences of data and keep previous data in memory, modeling a user's behavior over longer time series and finding anomalies that span multiple actions (also known as collective anomalies) should be possible as well. In order to this all of a user's actions in a training set are used to teach an RNN to predict a user's actions, after which the RNN predicts the user's next action. Actions that deviate from the predicted action can then be labeled as anomalous actions. Findings indicate that using an RNN for this task is technologically possible, requiring a lot of resources but being able to handle a big network (12000 users) in real-time. However, determining whether the detected anomalies are all anomalies, if some may have gone undetected and what the optimal features are that can be taken from the data set, is something that cannot be determined as the data set is unlabeled. This leads to the conclusion that more research on the area of using ANNs on unlabeled data 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. This leads to network administrators keeping track of everything going on in the network, in an attempt to pick out any weird behavior. The data of what goes on in the network is stored in so-called log files. The problem is that this data needs an expert's opinion, while also needing to be processed very quickly as there tends to be a lot of this data. 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 accurate, 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 pick out any weird behavior as it happens. This can be a very important factor for confidential data. The upcoming field of Deep Neural Networks (DNNs) seems like a perfect fit 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 make good choices.

In 2015, a data set[1] was published in [Ken15b] of around 100GB representing 58 consecutive days of anonymized event data collected from the US based Los Alamos National Laboratory's internal network. This data set consists of a number of different types of data: authentication data, process data, network flow data, DNS data and red team data. The authentication data is by far the biggest at 1,648,275,307 events. The red team data represents a set of simulated intrusions. The red team data is there to train the system on known intrusions (also known as misuse detection) or to validate any found anomalies, however, there is so little red team data that it is not feasible to do this. Because the rest of the data is non-labeled, meaning we do not know whether it actually is or isn't an anomaly, the system needs to be trained to recognize users' behavior. We then attempt try to find any deviations from this behavior (also known as anomalies). Because the data consists of series of actions, sequences of events that are only anomalies when seen 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, which specializes in series of data, is able to find these collective anomalies, making it a perfect fit.

[1]The data itself can be found here [Ken15a]

The main goal of this paper is to apply an LSTM to the data set in an effort to find anomalies. This is done by transforming the data set into features that are then used to train a network to predict the behavior of a user. Then the network is predicts the next action after being given some input, after which the prediction is compared to the actual action made. Any actions that deviate too much are 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 **??** the time taken to run the experiments is discussed; in Chapter 6 the findings are presented and Chapter 7 contains the conclusion.

Chapter 2

Related Work

The field of anomaly detection has always been a very active field of research, becoming even more active as the need for better systems 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 behavior based solely on the users' learned behavior. More advanced techniques like clustering have also been used to find anomalies in unlabeled data sets. Later 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 lookout for new intrusion patterns that felt like a cat-and-mouse game for network administrators, as explored by [PES01].

With the upcoming field of artificial neural networks and deep learning, the interest for using artificial neural networks for anomaly detection and intrusion detection has also increased greatly as they show great potential as explained in [LBH15]. Applying a simple backpropagation neural network to the terminal commands a user executed, in [RLM98], an attempt was made to identify users by these commands in order to find any deviations. 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.

Recurrent neural networks (RNNs) using Long Short Term Memory (LSTM) proved very useful in finding so-called time series anomalies, which are anomalies over a time frame with multiple actions in it instead of single-action anomalies. LSTM networks excel at this are due to their ability to learn which aspects of the "past" should be remembered and which aspects to forget. This is shown in [MVSA15], where a stacked LSTM, trained to recognize regular behavior, was demonstrated performing well on 4 different data sets. 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 significantly anomalous groups of cases.

Chapter 3

Recurrent Neural Networks

3.1 Recurrent Neural Networks

A recurrent neural network (RNN) is a variation of artificial neural networks that continuously uses the output of its previous layer as the input to the current layer along with the input that was inputted to this iteration (see figure 3.1). Because of this feature, RNNs have the ability to store and keep in memory previous input values as they can be passed along through the previous iteration's output.

As can be seen in figure 3.2, RNNs only have a single value that acts as both the output of the cell and the data passed through to the next iteration. This hidden layer value h_t at time step t is calculated by the following function where h_t is the value of the current hidden state and output and h_{t-1} the value of the previous hidden state, W is the weight matrix for the cell's input, U is the weight matrix for the hidden value, x_t is the input of this cell, and σ is a sigmoid function used to squash its inputs into the [-1, 1] range:

$$h_t = \sigma(Wx + Uh_{t-1})$$

In theory the RNN has the ability to recall previous inputs. In practice, however, standard RNNs seem to be

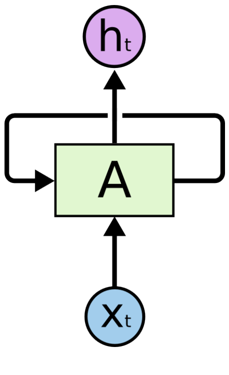

Figure 3.1: A recurrent neural network. From [Ola15]

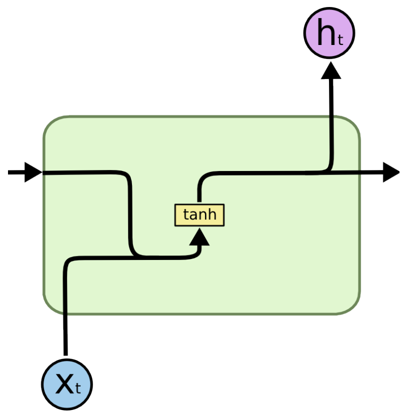

Figure 3.2: A single RNN cell. From [Ola15]

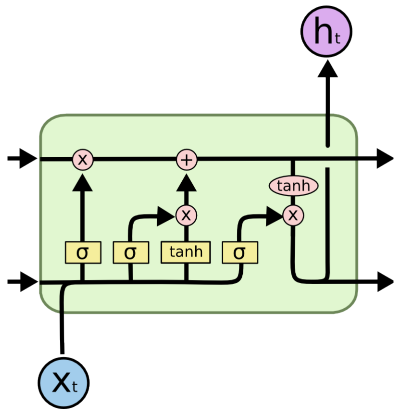

Figure 3.3: A single LSTM cell. From [Ola15]

not that amazing at remembering the past 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 commonly used LSTM RNN, introduced by [HS97].

3.2 LSTMs

Contrary to regular RNNs, LSTMs have 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 never having to forget things, 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 bit more parameters in this cell than a normal RNN cell. The output value and hidden value are composed of a number of different functions. 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. f_t can be obtained by using the following formula, where W contains the weights for the input and U contains the

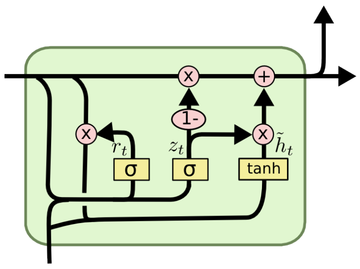

Figure 3.4: A single GRU variation cell. From [Ola15]

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:

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

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 and the input gate through a matrix addition function. The input gate (i_t) can be found by using the following formula:

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

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

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

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

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

The output state h_t can then be obtained:

$$h_t = o_t \circ \sigma(c_t)$$

This results in a version of an RNN that is able to remember more and is more liberal in choosing what it wants to remember and what it wants to discard. This allows them to be used a lot more effectively and has lead to the LSTM 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 Depth Gated RNN [YCV+15], which uses an additional depth gate to connect memory cells of adjacent layers, 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. In [JZS15] , findings show that some architectures did work better than others, however seeing as this study did not include a variation of anomaly detection in its testing problems, we'll be using the standard LSTM architecture. When training we will be using the "adam" optimizer, which was introduced in [KB14]. This is a method of gradient-based optimization of stochastic objective functions.

Chapter 4

Methods

In a data set as big as the one that is being used, there is a lot of information that is very essential but not included by default in a row (such as whether the user connected to an unvisited computer). There is also some data that does not tell a lot when unprocessed (such as text data that the network shouldn't handle as a vector). This is the reason for so-called features being created based on the data. These features are then sent to the network to train after being split into a training and test set. Because users tend to have different behavioral patterns in the data set and every other big network, the decision was made to train one network per user. The possibility of using one network for all users was explored, but this introduced some problems like the total amount of actions for all users not being the same and simple performance reasons. The network also gravitates towards detecting "average" users, always labeling outlying users like sysadmins as anomalies. After training on the training set, the network then runs on the test set. A list is then composed containing the differences between the expected and actual values. Any items in this list that deviate too much from the median are then labeled as anomalies.

4.1 Data

The data set from [Ken15b] contains a number of different types of data, as described in Chapter 1. For this thesis we will only be using the authentication data as that is by far the largest data set at 1,648,275,307 events. The data set has a total of 12,425 users over 17,684 computers and spans 58 consecutive days. 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 5% of the data set for plots/results in this thesis. Keep in mind that this means the first 5% of the file, not 5% of users. Because the file is sorted chronologically this is the same as using only the data of approximately 3 days. This causes many users to not have enough actions to pass the 150 actions baseline. Increasing this percentage will not only increase the actions for existing users but also introduces new users, together leading to an exponential increase in work required.

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

Because the raw data has some unneeded values, some that need to be transformed, and some that need to be added, features are made from the original rows. Because increasing the number of features has a big performance impact, keeping the number of features low, while still making sure the most important features are extracted is very important. The features are taken from the actions of one user over the whole data set. For 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 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 value. This brings the total number of the feature vector up to 33 (11, 9 and 6 possible values for the enums respectively).

The features have been chosen to fill 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 in to, but would probably be interested in knowing whether the user logged in to a new computer. Additionally the network is unlikely to subtract the previous action's time stamp from the current time stamp, but will probably be interested in knowing the time since the last action to see whether the user is doing lots of operations at once, is doing actions at a normal human speed, or if they're barely doing anything.

4.3 Preprocessing

Not all users are can be used for training/testing. This is because some users have so little actions that no reasonable predictions can be made for them. The minimal amount of actions was chosen to be 150. Any users having more than 150 actions are used for training/testing. 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, these values have to be normalized to fit into the 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 row, where x is the old row and x' is the new row:

$$x' = x / \max(x)$$

This operation is performed for the training and test set at the same time 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 originally, ensuring that the input data closely resembles real input data and making maximum 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 max 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 user will have more seconds between their last action than are in a human lifetime). One other method of solving this problem is by applying the following function to all (unscaled) feature values:

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

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]. This would be a good solution to the problem, however, as we are not using real-time data in the experiments this method is not needed.

4.4 Training

After preprocessing, the training data is then used as input for the networks (s). For performance reasons, a single network is created, which is then used as the base 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. This network 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 layers use an internal representation vector (and layer output) size of $feature\_size$, which is "between" the input and output sizes of $feature\_size$ (as suggested in [Hea08]). 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 by a lot. The network is trained first on the supplied training data, always trying to optimize for the lowest loss value using the mean squared error function (mse). This function measures the average of the squares of the deviations, 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 x' being the actual vector:

$$mse = (\sum_{0}^{n} (x_n - x'_n)^2)/n$$

This function is then applied to the predicted feature vector and the actual feature vector, thereby training the network to predict the next feature vector. This 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 network actually is at detecting attacks exists. Because of this no objective measure of how good the network exists, and the network's parameters can not be optimized by using this measure.

4.5 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 (as in a set of $batch\_size$ losses, one anomaly is not that significant), a method needs to be devised to test on a batch size of 1 instead. This is done by creating another network, identical in structure, 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 is then collected, after which the interquartile range (IQR) is calculated. 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, who 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.

$$x < Q1 - 1.5IQR$$

$$x > Q3 + 1.5IQR$$

In practice the first form of outlier 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 actual anomalous behavior.

After finding anomalies, they have to be translated into actual rows. This issue occurs because when inputting solely features, the original input is discarded. As network administrators will want to see the name of the user that is behind a found anomaly and may want to investigate the actions themselves, these anomalies are translated back into source actions. This is done by storing the index of an anomaly and the user associated with the anomaly. The indexes have a 1–to–1 correspondence to the source actions, allowing for easy translation. This step can be skipped if, for example, no anomalies were found or only known anomalies were found.

4.6 Code

All the code was written in Python, using the [C+15] Keras deep learning library's LSTM as the neural network, using all default settings except for those mentioned. 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 can be fully 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 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 5. When using the CPU Keras itself will use all CPUs available to it, however when using the GPU a problem arises. Because Keras requires a single GPU per process, multiple independent processes have to be launched in order to parallelize. This is done by having one root process splitting the to-do job between n processes, where n is the number of 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

Evaluation

Because there are three major stages to the evaluation (preprocessing, training/testing, translating) as explained in Chapter 4, and these three stages having different bottlenecks, not all experiments were run on the same computers. Both the preprocessing and translating stages require the reading of the entire data set file, making it very RAM intensive. They also only require CPU work, shifting the bottleneck to the CPU after reading the file. Preprocessing also allows for CPU parallelization, making more CPUs a very good thing to have. 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 preprocessing reduces the total amount of data, using a computer with less RAM is now 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.

In order to get an idea how long running everything on 100% of the data set would take, three different percentages have been used: 0.1%, 1% and 5%. Doing preprocessing took 38 seconds for 0.1% of the data while it took 40m16s for 5% of the data, both using 10 CPUs. A very rough estimate puts the duration of preprocessing the entire data set at about 40 hours also using 10 CPUs. Doing the training/testing stage with 16 GPUs took 1h51m on 0.1% of the data set, while 1% took about 10 hours and 5% of the data took 62h0m36s, giving a rough estimate of 2000 hours for 100% of the data set. Using 20 CPUs instead takes 55h51m3s on 5% of the data set, giving an estimate of 1800 hours for 100% of the data set. This means that on in our scenario running experiments on the CPU is faster. Results may vary based on the amount of CPUs or GPUs the experiments are executed on, making one of these two the faster one. The anomaly translation part generally only takes roughly 2 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 being done. The biggest time sink for this part is loading the data set file itself at about 2h15m. Putting this all together, the entire process takes 65h10m52s when using GPUs and 59h1m19s when using CPUs both for 5% of the data. The 5% data set contains 50,276,292 actions, meaning the network can handle about 214 actions per second on GPUs and 237 actions per second on CPUs, making this very fit for real-time anomaly detection. The actual testing stage (without training) takes even shorter, generally only taking a few seconds per user for all their actions, which would make a network that doesn't continue learning after the initial training even more feasible to run.

Chapter 6

Results

During the execution of the network, a number of plots have been made in order to visualize any outliers, along with a list of any anomalies the network has found. Keep in mind that these plots and anomalies are all based on 5% of the authentication data.

In order to get an idea of how much an action deviates relative to its containing distribution, a function has been created with which this can be calculated. In the function indicating the max value of x before x becomes an outlier, replacing 1.5 with y gives an indication of this deviation from its containing distribution. This gives the following function:

$$x > Q3 + yIQR$$

In order to find y here, the following function is used:

$$y = (x - Q3)/IQR$$

This y value is then plotted, the result of which can be seen in figure 6.1. As can be seen in the figure, there are quite a number of outliers, some of which having outliers that fall far beyond the cutoff value of 1.5. From this, we can conclude that at least some anomalies are being found.

As can be seen in figure 6.2, 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.

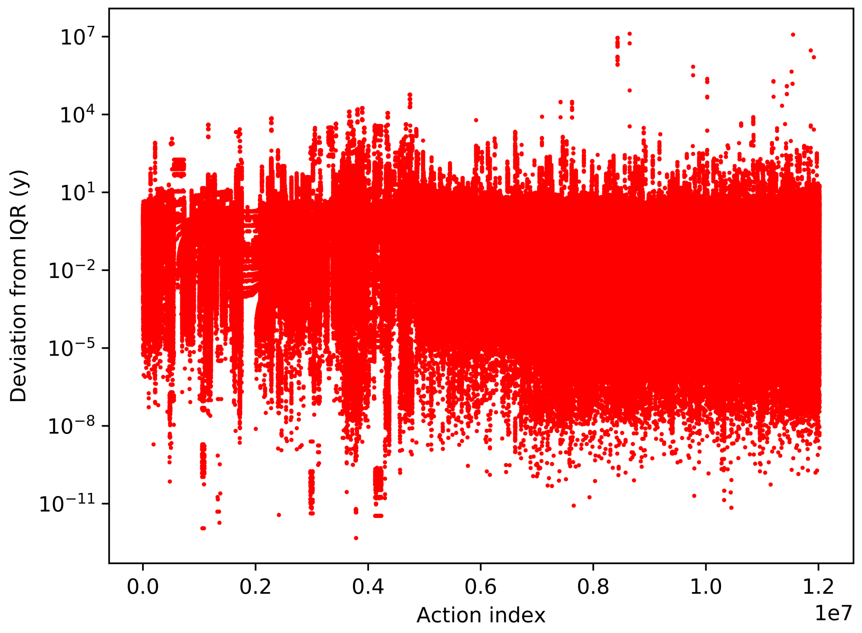

Figure 6.1: All deviations (y) from the IQR.

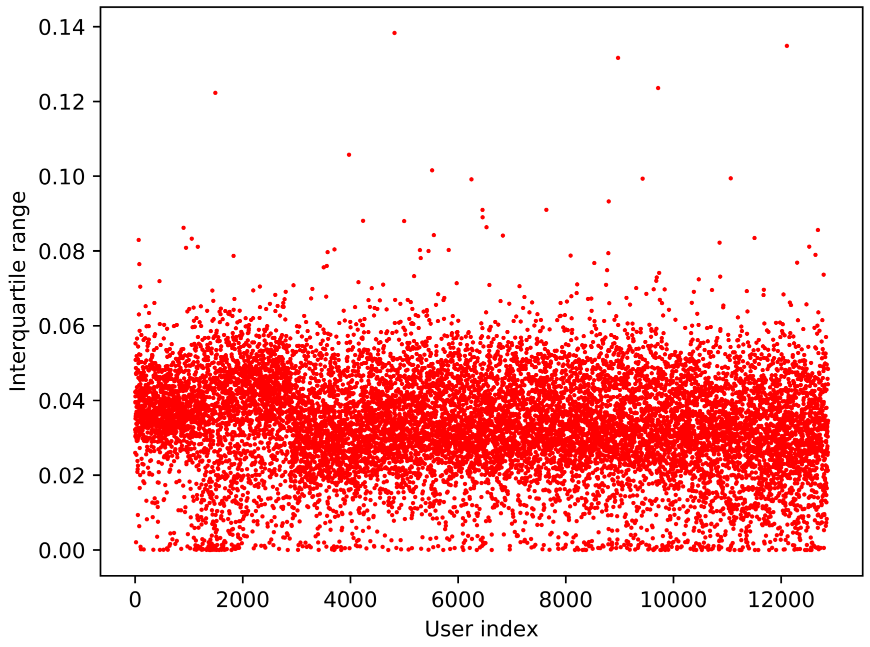

Figure 6.2: All IQR values.

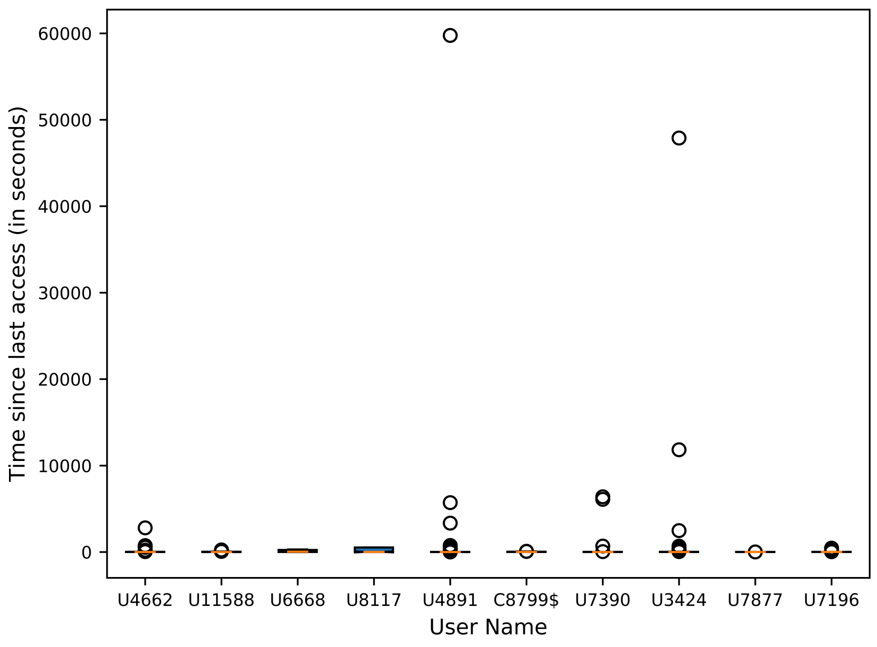

Figure 6.3: The top 10 highest offenders' seconds since last access.

In figure 6.3 we take a closer look 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, U4891 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 are some small and bigger deviations from this average, probably causing the network to flag them as suspicious.

The highest offending action's predicted vs actual action are shown in Table 6.1. In this table (and following tables) features that are integers and not floats have been depicted as such, as all but the *percentage_failed_logins* features are integers. Any predictions made by the network are trimmed to 1 decimal point.

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 0.0 to *logon_type_4* (interactive logon) and a value of 0.0 to *auth_type_0* (NTLM). When compared to the user's previous logins in Table 6.2, the last action really stands out as different. Many anomalies like this have been found. Often times the user logs in after a really long while or significantly changes their behavior by logging in using methods rarely or never used before. This signals that the network is doing a good job at recognizing the user's behavior and finding anything that deviates from it.

Table 6.1: The predicted features vs the actual features of the top offender (precision set to 2 decimals)

Label	Actual	Predicted
time_since_last_access	0	0.00
domains_delta	0	0.00
dest_users_delta	0	0.00
src_computers_delta	0	0.00
dest_computers_delta	0	0.00
percentage_failed_logins	0.0	0.00
success_failure	1	0.4
auth_type_0	1	0.1
auth_type_1	0	0.19
auth_type_2	0	0.00
auth_type_3	0	0.00
auth_type_4	0	0.00
auth_type_5	0	0.00
auth_type_6	0	0.00
auth_type_7	0	0.00
auth_type_8	0	0.00
auth_type_9	0	0.69
auth_type_10	0	0.01
logon_type_0	0	0.02
logon_type_1	0	0.00
logon_type_2	0	0.00
logon_type_3	0	0.00
logon_type_4	1	0.00
logon_type_5	0	0.05
logon_type_6	0	0.08
logon_type_7	0	0.08
logon_type_8	0	0.08
auth_orientation_0	0	0.58
auth_orientation_1	0	0.06
auth_orientation_2	0	0.00
auth_orientation_3	0	0.01
auth_orientation_4	0	0.00
auth_orientation_5	1	0.28

Table 6.2: The highest offending user's previous logins before the anomaly

time	source user@domain	destination user@domain	source computer	destination computer	authentication type	logon type	authentication orientation	success/failure
212020000	C14012$@DOM1	C14012$@DOM1	C14012	C2106	Network	LogOn	Success	
212020000	C14012$@DOM1	C14012$@DOM1	C14012	C2106	Network	LogOn	Success	
212020000	C14012$@DOM1	C14012$@DOM1	C2106	C2106	Network	LogOff	Success	
212023000	C14012$@DOM1	C14012$@DOM1	C2106	C2106	Network	LogOff	Success	
212029000	C14012$@DOM1	C14012$@DOM1	C2106	C2106	Network	LogOff	Success	
212043000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
212758000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
212772000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
212784000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
212797000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
213233000	C14012$@DOM1	U6147@DOM1	C14012	C14012	Unlock	LogOn	Success	
213233000	C14012$@DOM1	U6147@DOM1	C1521	C14012	Unlock	LogOn	Success	
213657000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
213671000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
213698000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
213707000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
214557000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
214571000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
214598000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
214608000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
215457000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
215471000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
215524000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
215533000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
216357000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
216371000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
216392000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
216405000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	
217145000	C14012$@DOM1	C14012$@DOM1	C14012	C2106	Network	LogOn	Success	
217149000	C14012$@DOM1	C14012$@DOM1	C2106	C2106	Network	LogOff	Success	
217257000	C14012$@DOM1	C14012$@DOM1	C14012	C457	Network	LogOn	Success	
217271000	C14012$@DOM1	C14012$@DOM1	C457	C457	Network	LogOff	Success	

Chapter 7

Conclusions

In this paper we investigated the feasibility of anomaly detection by using recurrent neural networks. Chapter 5 shows that it is possible to run a system in real-time to find anomalies using recurrent neural networks, as well as it being possible to run this system on a previously captured data set. Seeing as the network can handle 214 actions per second by using 16 GPUs and 237 actions per second by using 20 CPUs, 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 Los Alamos National Laboratory has 12,425 employees that generated 1,648,275,307 events in 58 days, this leads to 329 actions per second for 12,425 users. This means that 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 we can conclude 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 comment on. While from Chapter 5 we can conclude that the actions we investigated 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 users that need to be closely investigated, not one that can completely replace them.

Another problem was the tuning of the neural network itself and its parameters. 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 the network can be given, preventing the optimizing and fine tuning of the network, which 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. Lots of big strides are 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

[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.

[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.

[Ken15a] Alexander D. Kent. Comprehensive, Multi-Source Cyber-Security Events. Los Alamos National Laboratory, 2015.

[Ken15b] 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.

[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.