Universiteit Leiden
The Netherlands

Opleiding Informatica

Detecting anomalies with recurrent neural networks

Sander Ronde

Supervisors:

Wojztek Kowalczyk & Bas van Stein

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

`www.liacs.leidenuniv.nl` 12/08/2017

Abstract

This is where you write an abstract that concisely summarizes your thesis.

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 problem however 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 is also known as an Intrusion Detection System (IDS). This IDS 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, instead of finding out weeks after the fact which can be a very important factor for confidential data. The upcoming field of artificial neural networks (ANN) seems like a perfect fit for this problem, as it combines both the speed of computers, and attempts to mimic the ability of our brains to learn very quickly, allowing it to make good choices.

In 2015, [Ken15] published a dataset of around 100GB representing 58 consecutive days of de-identified event data collected from the US based Los Alamos National Laboratory's internal network. This dataset consists of a number of different types of data. These types are authentication data, process data, network flow data, DNS data and red team data, where the authentication data is by far the biggest at 1,648,275,307 events. Here 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. Seeing as the rest of the data is non-labeled data, where we do not know whether it actually is or isn't an anomaly, the system needs to be trained to recognize users' behavior and 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 dataset. 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.

It is recommended to end the introduction with an overview of the thesis. This chapter contains the introduction; Chapter 2 discusses related work; Chapter 6 concludes.

Also make a nice sentence with "bachelor thesis", LIACS and the names of the supervisors.

Chapter 2

Related Work

The field of intrusion detection has always been a very active field, 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, while in [LS^+98], simple classifiers were used to find anomalous behavior based on known intrusion techniques 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 techniques and anomalous changes in user behavior using clustering, attempting to go beyond the constant lookout for new intrusion techniques 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. 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, reaching a 96% accuracy in finding unusual activity. Neural networks perform a lot better on noisy data where some fields may be missing or incomplete, as [Can98] shows, applying a neural network 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. LSTMs excel at this are due to their ability to remember past input, as is shown in [MVSA15], where a stacked LSTM, trained to recognize regular behavior, was demonstrated performing well on 4 different datasets. LSTMs 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 off 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 of 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 timestep 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, however in practice standard RNNs seem to be not that amazing at remembering data for long periods of time. This problem was investigated in [BSF94] among

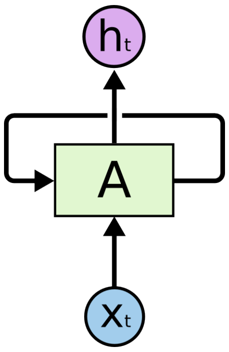

Figure 3.1: A recurrent neural network. From [Chr15c]

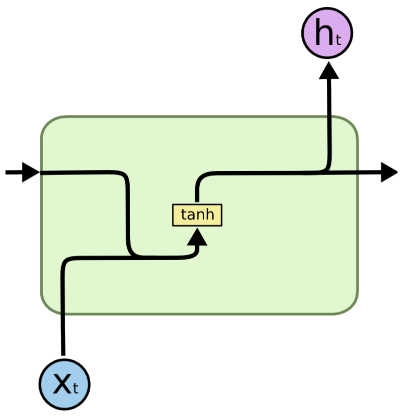

Figure 3.2: A single RNN cell. From [Chr15d]

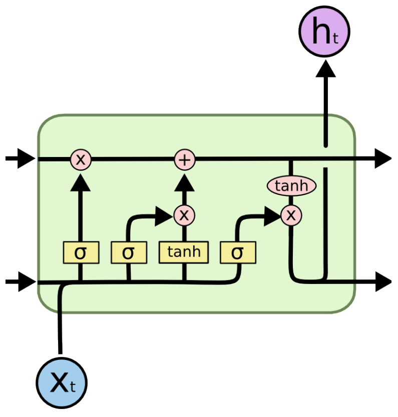

Figure 3.3: A single LSTM cell. From [Chr15b]

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 data. 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 to 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 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 vectors:

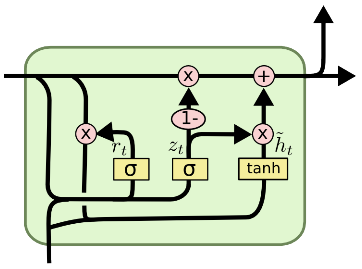

Figure 3.4: A single GRU variation cell. From [Chr15a]

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

The network then determines what to remember from the input values. 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 value of the hidden state (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 matrixes):

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

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

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

The output value 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. [JZS15] found 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.

Chapter 4

Methods

In a dataset 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), while there is also some data that is completely useless by default (text data that the network can't handle as a vector). This is the reason for the data being turned into so-called features. These features are then sent to the network to learn after being split into a training and test set. Because users behave very differently in the dataset (and every 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 amount of actions for all users not being the same, the network moving to recognize the median values, and simple performance reasons. 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 dataset from [Ken15] contains a number of different types of data, as described in the introduction. For this thesis we will only be using the authentication data as that is by far the largest dataset at 1,648,275,307 events. The dataset 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 timeframe at which it was captured not being disclosed. The data format can be seen in table 4.1

Table 4.1: The dataset 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

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 amount of features has a big performance impact, keeping the amount 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 dataset. For the features see table 4.2. All enum values (indexes 5,6,7) have been encoded using 1-of-encoding (see table **??** 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 timestamp from the current timestamp, 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 however 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).

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 consist of 3 layers, with the first two being stateful LSTMs, and the third layer being a dense layer. All using 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 network uses a batch size of 32. This number was chosen because increasing this value would decrease the weight of individual actions when in the testing stage as the network always only accepts input of that size and outputs a single "loss" value. Increasing the batch size would reduce the significance of a single anomaly on the loss value. Reducing the batch size however, 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, which measures the average of the squares of the deviations, giving an approximation of the deviation from the expected value. 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 values are not perfect, seeing as there was no way to objectively compare the results of different parameters.

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 however, 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 of course 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) is a very GPU-dependent task. 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 amount 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

Due to there being three major stages to the evaluation (preprocessing, training/testing, translating) as explained in section 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 dataset 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. As such 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.The training/testing stage however, is a very GPU intensive task. Due to parallelization being highly effective for this task, as explained in section 4, the amount of GPUs is very influential for this task, offering close to linear performance improvements, 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 16 NVIDIA Tesla K80 GPUs each with 11.5GB of memory.

Due to the size of the dataset and the limited amount of time, the decision was made to use only 5% of the dataset. Keep in mind that this means the first 5% of the file, not 5% of users. This means that approximately 3 days of data is being used. 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 (causing an exponential increase in work required) but also introduces new users, also leading to an exponential increase in work.

The preprocessing stage took approximately when only using the first 5% of the file. A very rough estimate puts the duration of preprocessing the entire dataset at about 130 hours.

Chapter 6

Conclusions

Unfortunately there is no knowing whether better features could be chosen as the dataset is unlabeled and there is no way of knowing whether the detected anomalies are actually anomalies or whether we missed some apart from labeling it. As such that would be a good candidate for further research.

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.

[Chr15a] Christopher Olah. Gru variation. http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-var-GRU.png, 2015. [Online; accessed August 8, 2017].

[Chr15b] Christopher Olah. Lstm chain. http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png, 2015. [Online; accessed August 8, 2017].

[Chr15c] Christopher Olah. Rnn rolled. http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/RNN-rolled.png, 2015. [Online; accessed August 8, 2017].

[Chr15d] Christopher Olah. Simple rnn. http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-SimpleRNN.png, 2015. [Online; accessed August 8, 2017].

[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.

[Ken15] Alexander D. Kent. Cybersecurity Data Sources for Dynamic Network Research. In *Dynamic Networks in Cybersecurity*. Imperial College Press, June 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*, page 89. 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.

[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.