Discriminating if a network flow could have been created from a given sequence of network packets

Bachelor thesis by Jens Keim
Date of submission: September 11, 2020

1. Review: Prof. Dr. Max Mühlhäuser
2. Review: Dr. Carlos G. Cordero
3. Review: Aidmar Wainakh
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Informatik
Telecooperation
SPIN

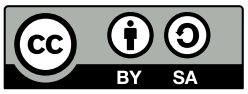
Discriminating if a network flow could have been created from a given sequence of network packets

Bachelor thesis by Jens Keim

1. Review: Prof. Dr. Max Mühlhäuser
2. Review: Dr. Carlos G. Cordero
3. Review: Aidmar Wainakh

Date of submission: September 11, 2020

Darmstadt



This thesis is dedicated to my loving family.
To my mother Marlene Keim as well as my father Joachim Keim, without whose support and patience my bachelor's degree would not have been possible.
To my sister Angelika Keim, who always motivated me during my studies, even if it was difficult. I would also like to thank her for taking care of our mother, when I could not.
I also owe special thanks to my supervisor Carlos Garcia Cordero, as his guidance helped me a lot during this thesis.

Diese Arbeit ist meiner liebenden Familie gewidment.
Meiner Mutter Marlene Keim sowie meinem Vater Joachim Keim, ohne deren Unterstützung und Geduld mein Bachelorstudium nicht möglich gewesen wäre.
Meiner Schwester Angelika Keim, die mich stehts für das Studium motiviert hat, selbst wenn es schwierig war. Auch möchte ich ihr dafür danken, dass sie für unsere Mutter gesorgt hat, als ich es nicht konnte.
Außerdem gilt besonderer Dank meinem Betreuer Carlos Garcia Cordero, da seine Anleitung eine große Hilfe während meiner Ausarbeitung war.

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Jens Keim, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, September 11, 2020

J. Keim

Contents

1. Abstract

This thesis aims to design a neural network (NN), that is capable of discriminating if a network flow could have been created based on a sequence of packets and can be used as a discriminative network (DN) for a Generative Adversarial Network (GAN) in future work.

For this, we first determined the features of network flows and packets alike, which are relevant to this task. We then created a dataset by extracting the relevant features from well-known network traffic datasets from the field of network intrusion detection (NID), as well as falsifying said datapoints to provide negative samples. We also provide a pipeline for the process of creating such datasets.

For our NN model we compared available architectures of recurrent neural networks (RNNs): simple RNN (simpleRNN), Long Short Term Memory (LSTM), and Gated Recurrent Units (GRUs). Furthermore our model uses a special kind of RNN called a conditional RNN (condRNN) [25], which already has provided good results for a mixture of conditional and sequential input in the field of image region classification [15] [33]. This is necessary as a flow is the conditional counterpart to a sequence of packets. We aim to test the effectiveness of the different RNN architectures in regards to our problem and in the context of condRNNs.

2. Introduction

This work focuses on the creation of a NN model that is capable of deciding if a network flow could have been created by a sequence of network packets. In this chapter, we will explain the motivation and idea behind this task, as well as research questions and objectives associated with it.

2.1. Motivation

Network intrusion detection systems (NIDSs) require datasets to be trained and tested. However, appropriate datasets for NIDSs are hard to come by since creating authentic synthetic datasets is difficult and time-consuming and most real traffic is rarely shared due to privacy concerns [26] or copyright [7]. This is why researchers in the field of NIDSs are restricted to use datasets with known defects or to create their own datasets.

A researcher that chooses to create their own datasets may use real network traffic, synthetic network traffic, or a mixture of both. Using real traffic to create a dataset, although desirable, has many disadvantages. The capture might be too old to represent current networks or it may contain artifacts. Synthetic network traffic is hard to create because network traffic is diverse. It is influenced by many factors, like the countless terminals communicating, the interim devices, e.g. switches, and their bandwidth, gateways, subnets, and churn of terminals. To generate appropriate datasets one would need to simulate all those factors and their interactions. Since this is particularly complicated, this is where the mixture of real and synthetic traffic comes in. With this method, one takes an already existing network packet capture and modifies it to fit their needs. This modification can range from adding specific packets, e.g. of a network attack [7], to modifying existing packets to fit a new network topology or altering other significant features of the network behavior by modifying each packet. Even though this method is quite effective, its design and implementation are also time-consuming and might need to be repeated for each

use case. Since it is difficult to create realistic network packet captures from scratch, one could propose to create these traffic captures based on network flows, as those are more commonly availabilable.

The paper ”Flow-based Network Traffic Generation using Generative Adversarial Networks” by Markus Ring et al. [26] provides a method to create synthetic network flows that mimic a given set of network flows. Building on top of this method one could synthetically create labeled datasets of authentic network flows for NIDSs. However, it is still desirable to have traffic based datasets, since they provide more information than flows. Instead one could create synthetic network flows like proposed by Ring et al. [26] and use these already authentic flows to create authentic synthetic network traffic at the packet level. One approach to this could be a GAN that generates network packet captures based on network flows.

2.2. Problem

If we would want to create a GAN that is able to create the packets from which a network flow was constructed, three major challenges need to be addressed. One, the DN of the GAN needs to be defined such that it can determine if a network flow could have been created by a sequence of network packets. Two, the generative network (GN) of the GAN needs to be able to create network packet captures that are indistinguishable (in some statistical sense) from the packets the network flow was created from. Three, the training of the GAN and the required datasets. To create a GAN, both the DN and GN need to perform well on their own.

In this research, we focus on building a NN that can be used as the DN of a GAN in future work. It is not obvious, however, how to create a DN that can verify that a network flow was created from a sequence of packets. This leads to the research questions presented next.

2.3. Research Questions

This thesis will attempt to answer the following research questions.

- How can we develop a neural network that acts as a discriminator for a GAN that can determine if a network flow could have been created by a sequence of network packets.
- Which features of network flows and packets are (most) significant to determine if a network flow could have been created by a sequence of network packets?
- Which NN architecture would be most suitable to distinguish if network packets could have created some network flow?

2.4. Goals and Objectives

This thesis focuses on creating a NN model that can be used as the DN of a GAN that takes network flows as input and produces network packets as output. The NN gets an array of network packets and at least one network flow as input and provides the probability that a specific network flow was created from the given sequence of packets. To achieve this goal, we might need to consider additional input, such as labels for either the network flows, the network packets, or both. Therefore the goal of this thesis is to create a NN model that distinguishes if a specific network flow could have been created from a given sequence of packets, or not.

It will not only be necessary to create the NN architecture, but also the datasets needed to train and test the system. For this, we need suitable representations of network flows and packets that a NN can process. To test the reliability of our model, diverse datasets from sufficiently distinct networks need to be constructed. So to reach the goal, we need to reach the following objectives:

- Collecting diverse packet captures, representing a wide range of networks.
- Extracting flows from said packet captures.
- Extracting features from packets and flows alike, which can be used to distinguish if packets belong to a certain flow.
- Creating fake flows by modifying the extracted ones based on those features.
- Creating a NN model that is able to compute those features.
- Testing the effectiveness of NN architectures to work as discriminators.
- Deciding which NN architecture is capable of making correct decisions.

3. Background

In this chapter we explain background knowledge and terminology required in order to understand the following contributions.

3.1. Network Flows v.s. Network Packets

If we want to extract features from network flows we first need to understand the difference between network flows and network traffic captures. A network packet capture is a recording of network traffic at the packet level. It is limited by a start and end time. Each packet within the capture contains all header information it contained during actual transmission. The payload of the packets is omitted if its size gets too large to be reasonably stored, otherwise the packets are unaltered. A network flow on the other hand is an artificial logical equivalent to only one network connection [5].

This connection may be between two terminals, a multicast group or a terminal and a broadcast address [23]. Like a network packet capture, a flow is limited by a start and end time, but it does not contain information on individual packets. It holds aggregated information of the network packets within the connection. For a packet to classify as belonging to a flow the packets typically need to share properties like transport protocol, source and destination IP as well as source and destination port [12] [6]. So it describes the connection on a more abstract level and provides less information than a network packet capture.

For example, a flow in the YAF format will only keep track of a minimum of header information, like source and destination IP address and port. It will contain the flags used by the first and last packet of the communication, but it will not contain the flags of each individual packet within the communication. It will know the duration and average round trip time of the communicating, but not the time intervals between two packets.

Additionally, it might have information the packets themselves do not provide, e.g. the reason for the end of the communication [36].

3.1.1. YAF

Yet Another Flowmeter (YAF) is a tool to create network flows based on network traffic. It is capable of saving bidirectional flow data of an active link or extracting flows from an offline capture file. It uses IPFIX-based data structures and file formats, which can also be exported and used by IPFIX compliant toolchains. We will use the later functionality will help us to create a dataset for our use case. With `yafscii` YAF provides a tool to export YAF network flow data as plain text. This will also come in handy for us, as we try to keep our dataset sampling simple by using csv files.

3.1.2. Wireshark

Wireshark is probably the most well-known network traffic tool out there. It is capable of capturing, displaying, sorting, and filtering network traffic captures, as well as analysing network packet captures. It also comes with further functionality, however in the scope of this thesis, we will mostly be utilizing the filter feature. Wireshark comes equipped with a GUI but also provides a CLI tool called `tshark`. `tshark` includes the same set of features as the GUI does while being slightly more performant. We will be using this CLI tool for Wireshark henceforth within this thesis.

3.2. Network traffic datasets

This thesis aims to lay the groundwork for a new method of the creation of authentic synthetic network packet captures that can be used for labeled datasets for training and testing NIDSs. Anomaly-based NIDSs (ANIDS) need network traffic datasets for testing and training. Those datasets need to be appropriate to the threat and labeled accordingly. However appropriate datasets are hard to come by. Most real traffic recordings are rarely shared due to privacy concerns [26] or copyright [7]. The datasets including network attacks, which are shared publicly, tend to either be snapshots from real attacks or synthetically created traces from an isolated network. The later does not provide realistic non-attack traffic, or background traffic. The former present the problem of non-attack

traffic being of a unique network in a unique time frame and therefore not representative of every network. Also, they most likely get anonymized and are therefore do not represent the ground truth. Thus these datasets are not sufficient to train or test NIDS. And no matter the origin the dataset will only describe the network and the behavior of its traffic at the time of recording or creation, hence it might not be applicable in future networks [26].

3.3. Recurrent Neural Networks

To solve the problem stated in this thesis we are required to find an appropriate NN architecture. Our NN model should be able to decide if a network flow could have been created from a sequence of network packets. The most commonly used NN architecture for sequential data are recurrent neural networks (RNNs). A feed-forward NN usually processes each input independently from previous inputs. RNNs store a hidden state obtained from the previous result of the NN and uses them to process the next input. Natural language processing usually benefits from this, as this allows to draw connections between the words within a sequence. In theory, this ”memory” can hold information about all previous calculations, but in practice, it can only keep track of the last few steps. This is due to the vanishing gradient problem [14].

Vanishing Gradient Problem

The vanishing gradient problem describes an RNN's tendency to forget earlier timestamps of a longer sequence. It is also known as the short term memory problem. To understand how this problem manifests one needs to understand how an RNN operates. To ensure a single big value within one timestamp does not make other values less significant, but values also do not lose their relation to each other, RNNs use the hyperbolic tangent function to ensure the values stay between -1 and 1. The values in the middle of the range of data are now closest to 0. However this also means that those values become so small that they do not contribute to the learning process of the RNN anymore. Thus vanishing from the weights.

3.3.1. Long Short Term Memory & Gated Recurrent Units

This is where Long Short Term Memory (LSTM) networks [14] or Gated Recurrent Units (GRUs), formerly known as gated recursive convolutional neural network (grConv) [2], come in. Both similarly tackle the short-term memory issue, also known as the vanishing gradient problem [14], where the weights of the memory are changed so little, that it becomes insignificant. Both models provide a gated approach. They also both rely on the sigmoid function in addition to the hyperbolic tangent function a SimpleRNN uses. While the hyperbolic tangent function produces values between -1 and 1, the sigmoid function produces values from 0 to 1. This benefits the memory since low impact data is multiplied by 0 and therefore left out of the equation. If the value is multiplied by 1 it stays the same. So the internal gates help the NN to learn what data is important and what data can be forgotten, while keeping the important data in memory. The difference between LSTMs and GRUs lies in how they use the sigmoid function in their internal gates.

Long Short Term Memory

LSTMs have three gates, the forget gate, the input gate and the output gate, and two states, the cell state and the hidden states. [22] The forget gate performs a sigmoid operation on the previous hidden state and the current input. The result is then used as a multiplier for the cell state, effectively letting the NN forget any information that is no longer relevant after taking the new input into account. The input gate gets the same input, but uses the hyperbolic tangent and sigmoid functions in parallel and then multiplies their results. The sigmoid function helps to decide which of the current data, provided by the hyperbolic tangent function, is important enough to keep, as this result will be added to the cell state. The output gate decides what information the next hidden state should carry. It calculates the sigmoid function on the concatenation of the previous hidden state and the current input, just as the other gates. The results gets used as weights for a hyperbolic tangent operation on the new cell state, meaning simple multiplication. The newly calculated cell state and hidden state get passed to the next timestamp.

Gated Recurrent Units

GRUs have only two gates, the reset gate and the update gate, as well as the cell state. [22] The update gate uses a sigmoid activation over the current input and the previous cell state, as the GRU has no hidden state. The inverted values are then used to weight

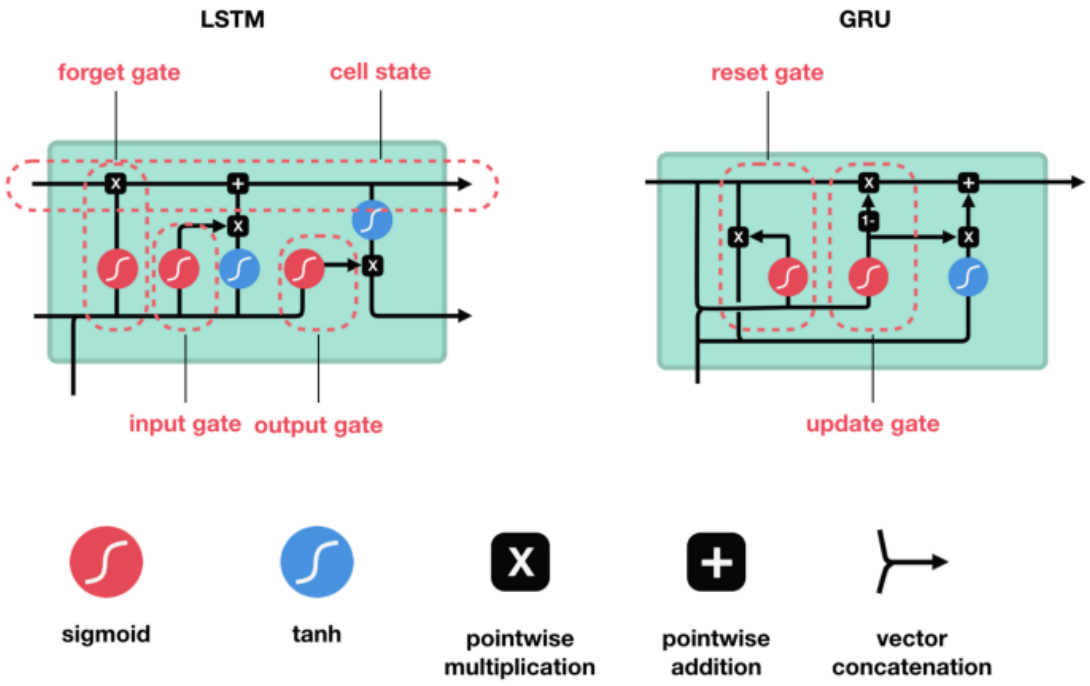


Figure 3.1.: Schematic comparison of LSTMs and GRUs, created by Micheal Phi [22]

the cell state. The reset gate again uses the same sigmoid unit and data as the update gate. This result gets multiplied with the current cell state and concatenated with the current input. This concatenation is processed by a hyperbolic tangent and multiplied by the non-inverted values of the update gate, before being added to the current cell state. A GRU has fewer activations than an LSTM and therefore requires less computation. This is illustrated in the schematic comparison in figure 3.1.

Now we have seen how different RNNs operate and how LSTMs and GRUs solve the vanishing gradient problem for sequential input. In our case however, we do provide more than one input and not all of our inputs are sequential.

3.3.2. Conditional Recurrent Neural Network

This is where the conditional recurrent neural network (condRNN) [25] comes in. Compared to the earlier explained RNN architectures, it allows a second input, a non-sequential input, called conditional input, in addition to the sequential input. Just what we need for our flow data. The conditional input gets used to set the initial state of the RNN for each datapoint. This allows the conditional data to be taken into account without appending it to each timestamp in the sequence. Also see 4.2.1 Image Description Generation in Related Work.

3.4. Masking

The problem presented in this thesis involves the classification of network packets related to their flows. As network flows vary in sequence length and different protocols contain a different amount of header fields and therefore features, one problem to solve is the inconsistency of our data sequence. Masking is a common practice in the field of NN models to process sequences with missing data. The missing data gets filled with a unique value before it is processed by the NN. This value is called masking value and should not appear anywhere else in the data. Once the completed sequence is fed into the NN, a masking capable layer processes the input data and creates a vector containing only ones and zeros (or true and false). This vector is called the mask. The mask is passed as a separate input to a mask supporting layer of the NN that should process the original input. The layer processing the input data now can refer to the mask and only takes into account the data that is marked as valid in the provided mask. With this procedure holes in sequence can be closed and sequences of non-uniform length can be padded without changing the requirements for the model.

3.5. Generative Adversarial Networks

To create a NN that could later be used as a DN we first need to understand how a Generative Adversarial Network (GAN) operates. A GAN consists of two NNs that compete against each other to maximize the quality of the generated data: a generator network (GN) and a discriminative network (DN). The GN creates synthetic samples and the DN gets real or synthetic samples at random. The DN then needs to decide if the sample it just

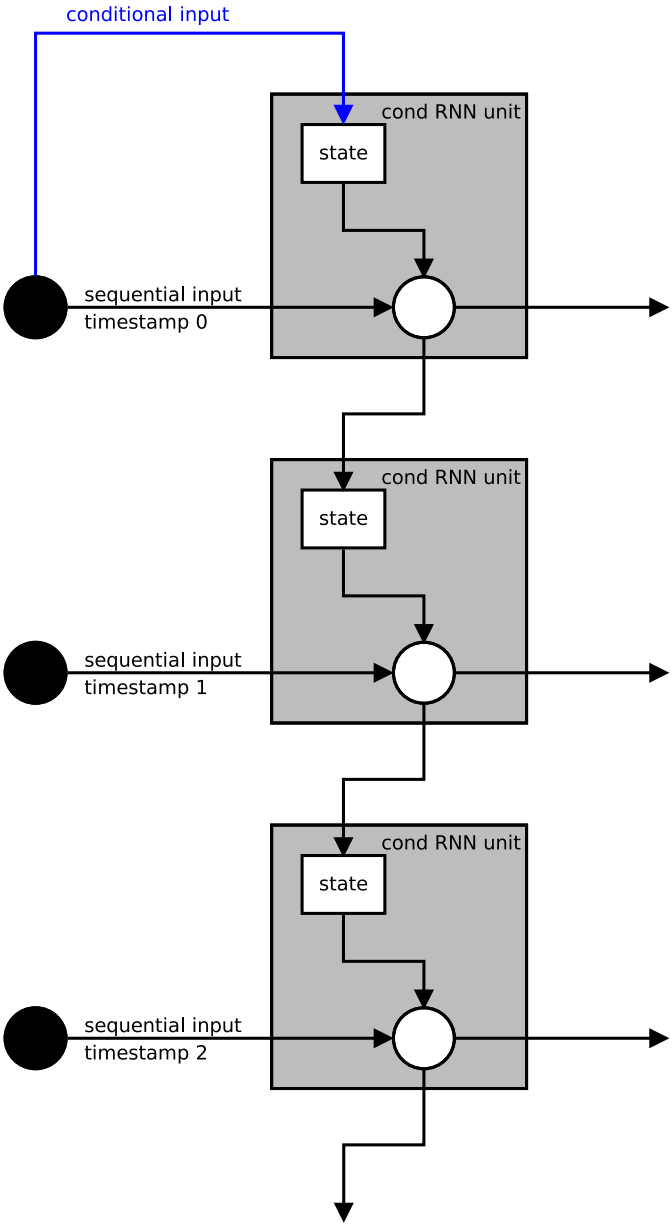


Figure 3.2.: Schematic sketch of the input for a condRNN unit over time, the areas marked blue present the difference to the normal RNN architecture.

saw is real or fake. Afterward, the decisions of the DN gets evaluated and both the DN and GN get fine-tuned based on the results. If the decision of the DN was correct than the GN is not good enough too fool the DN and therefore gets a greater loss, while the DN gets a smaller loss. If the DN was not able to make correct decisions it gets a greater loss and the DN a smaller loss. The GN is constantly trying to fool the DN with data it generates synthetically, while is constantly trying to figure out if the data it was given is real or synthetic. If correctly optimized, a GAN may create synthetic data indistinguishable from real data.

3.6. Terminology

During this work, we will use certain phrases and wording that may seem vague but simplifies more complex expressions.

When we say *a sequence of (network) packets belongs to a (network) flow*, we mean that *the (network) flow could have been created from this sequence of (network) packets*.

A *network flow* might also be called *network packet flow*, *packet flow* or just *flow*.

A *network packet capture* might be called *network capture*, *packet capture*, or even *pcap (file)*.

When we talk about positive or negative *samples*, the sample means datapoint.

When we talk about *errors* in regards to datapoints, we refer to the alterations made to a positive datapoint so that it becomes a negative datapoint.

A *dead feature* is a feature, which is not properly taken into account by a NN during the learning process, for example because it never changes. In other words, such a feature gets ignored.

4. Related Work

4.1. Flow-based Network Traffic Generation using Generative Adversarial Networks

Ring et al. [26] introduce a GAN to generate synthetic network flows. Not only are their synthetically created network flows interesting as the source of input for the proposed GAN in future work, but their choice and classification of network flow features might be a valuable resource for this thesis. They classify transport protocol, IP addresses, and ports as categorical features, duration as continuous, the number of bytes, and packets as numeric. TCP Flags could either by seen as binary attributes or one categorical value. Even though Ring et al. [26] do not create network packet captures, most of their classification is applicable to our problem. In addition, they reference and apply a novel approach for the conversion of categorical features of network flows. They call it IP2Vec.

4.1.1. IP2Vec

Network flows consist of multiple features, some of which are categorical, e.g. IP addresses, ports, and protocols. However, NNs work best with continuous and numerical data. To effectively train a NNs, which work best with numerical and continuous data, we need to represent those categorical features as numerical data. For this we planned on using IP2Vec proposed by Ring et al. [27]. This solution produces similarity values for IP addresses based on the behavior of the host.

They based their work on the concept of word2Vec from the field of natural language processing. There the word is observed in the context of its neighbors instead of alone. Similarly, in IP2Vec IPs are not observed on their own, but in the context of other features of the flow, they correlate with. So instead of words IP2Vec processes network flow attributes, specifically source IP, destination IP, destination port and protocol. However,

one could change those attributes and determine the difference between IPs based on different attributes. With this approach they are able to determine the differences between IPs based on their communication patterns and behavior within one dataset. For example, a destination IP that correlates with port 80 often can be easily classified as an HTTP server. Their experiments show that provided the right dataset, the data produced by IP2Vec can be used to distinguish between infected and non-infected hosts, as well as between servers, clients and even printers.

Conclusion on IP2Vec

After due consideration, we decided not to use the IP2Vec approach. This is mainly because of the fact that addresses do not influence the flow behavior too much. The only feature of them that might influence a flow is if they are public or local (private). Additionally, IP2Vec sees IPs in the context of a series of packet flows, which represent a network traffic capture, meaning that it relates the IPs to one another depending on the behavior of their flows within one network capture.

This does not apply in our use case, since we use the traffic of multiple network packet capture datasets. To use IP2Vec we would either have to apply it before extracting the flows and mixing the results, or mix the flows and then apply IP2Vec to it. The former has the problem that we would risk introducing false correlations between IPs from different datasets. As an IP address could appear in more than one network capture and have a completely different behavior in each. Mixing the information IP2Vec extracted from both of the captures would distort the results of the NN regarding the affected flows. The later IP2Vec approach has the problem, that IPs will correlate to each other without ever having been in the same network. IP2Vec does not account for problems that might result from this. The best-case scenario is that two IPs from different captures will be classified as being unknown to each other. But even this holds the issue that a later flow may have communication between those IP addresses and the GAN will have problems generating packet data that will satisfy the DN, because to the DN this will look like an anomaly.

Furthermore, we would assume that for our GAN in the future work, we have a whole set of flows for each IP address, that could provide the data necessary to apply IP2Vec before using the flow as input to the DN. This will most likely not be the case, as the GN of the GAN would try to create a sequence of packets based on the input of one flow. So it would not have the sufficient data to create IP2Vec vectors. Since both possible approaches have the problem with IPs that might be present in multiple source datasets, and we can not

assume that the future work GAN is able to produce IP2Vec vectors from one flow, we concluded that IP2Vec is not suitable for our problem.

4.2. Related Neural Network Models

We researched related work regarding the generation of synthetic network traffic using NN. But besides Ring et al. there was nothing of interest to this thesis. However, we also researched RNNs and GANs. Since some approaches to solutions for other problems might be applicable to our problem of the discrimination if a network flow could have been created from a given sequence of network packets. In addition to a GAN utilizing federated learning (FL) by by Rasouli et al. [24] that could be interesting to the future work GAN, we found an interesting concept for a RNN. Once the feature sets of network flows and packets were defined it became clearer which papers where to be considered. This is when we found Vinyals [33] and Karpathy [15] works on image description generation.

4.2.1. Image Description Generation

On first glance the field of image description may look out of the scope of this thesis. However, in computer science and machine learning in particular, it is not unlikely for two very separate problems to have the same solution. The process of teaching a NN to describe an image or even different regions of the image using natural language is a big and complex problem. The papers "Show and Tell: A Neural Image Caption Generator" by Vinyals et al. [33] and "Deep Visual-Semantic Alignments for Generating Image Descriptions" by Karpathy and Fei-Fei [15] both solved this problem in 2015. However, the problem we were interested in is much smaller.

How did both Vinyals and Karpathy solve the problem of multiple inputs for RNNs? They had the image data and the descriptions in natural language. While the sequence of descriptions is sequential the image data is not, but it still needed to be included as input for the RNN. This is where both parties independently create a condRNN, that took the image as conditional input in the beginning and the descriptions as sequential input. The conditional input conditions the state of the RNN before the sequence is read. For more details on condRNNs, see also section 3.3.2.

Deep Visual-Semantic Alignments for Generating Image Descriptions

Karpathy et al. extract pixel data of the whole image and the 19 locations within the image. They use a convolutional neural network (CNN) to convert each set of pixels into a vector of h dimensions. So each image is represented by a set of 20 vectors with h dimensions. They use a bidirectional recurrent neural network (BRNN) that takes a sequence of N words and transforms each word again into an h-dimensional vector. This vector got adjusted with 300-dimensional word2vec weights based on its context within the sentence. They then condition their RNN on the first step with the image date via bias interactions. Then the RNN processes the sequence of word vectors in order to learn how to predict the next word in the description based on the image data and previous words. Their word vectors began with a designated START word and ended with a STOP word. This way they were able to counteract the unequal length of their sequences. However, it is not obvious how they were able to use SGD with mini-batches of 100 image-sequences, with variable sequence lengths.

Show and Tell: A Neural Image Caption Generator

Vinyals et al. also use a CNN to create a proper representation of the image that can be passed to the RNN. For the RNN they chose the LSTM model with the adjustment of a conditional input. The also used 512 units per RNN layer. Over all, their work is quite similar to Karpathy et al..

5. Methodology

For our NN model, we assume that some form of Recurrent Neural Network (RNN) will be suitable as a discriminator of a GAN that can create synthetic network packet captures from network flows. Since simpleRNNs have the problem of short-term memory, we propose the use of either Long Short Term Memory (LSTM) networks [14] or Gated Recurrent Units (GRUs) [2]. GRUs tend to use fewer operations and states to produce similar or better results than LSTMs. But depending on the use case, LSTMs might still produce better results. It is to be determined, which of the two neural networks provides better results for the problem in this thesis.

For this we will run experiments with the **simpleRNN**, **LSTM** and **GRU** layers provided in Tensorflow. **simpleRNN** will provide the baseline for the real contenders **LSTM** and **GRU** during the experiments, as it will most likely provide the least optimal results. Before we can pursue the experiments however we need data to train and test with.

5.1. Dataset

For our dataset, we will extract packet sequences and corresponding network flow from publicly available and well-known network packet captures and create network flows based on these sequences. Each network packet sequence will be paired with its network flow so that our NN can learn the connection between the two data structures. Since the end goal in the future work GAN is to generate the complete packet sequences based of the flows, each sequence of packet features will contain as much of the original header information as needed to reconstruct a packet sequence from them. This way, no crucial information gets lost and the DN is prepared to handle the complete data of a packet capture, compare them to flows and make an informed decision if a network flow belongs to a packet sequence.

Each flow attribute and packet header field is converted into an appropriate representation depending on its data type and meaning for the sequence of packets. Since the NNs are most efficient with continuous and numerical data, we will convert all fields accordingly, wherever possible. We will also implement a pipeline to create negative samples from the ground truth, which builds our positive samples.

5.1.1. Dataset Preparation

The NN needs to be trained with a significant amount of both positive and negative datapoints so that it can make informed decisions. First, we collect samples that represent the ground truth and will be our positive datapoints. We sample flows from multiple packet captures we extracted from popular intrusion detection datasets with network traffic packets. Then in order to create negative datapoints, we randomly apply modifications to the previously collected positive samples.

Sampling

Instead of collecting all flows from our source captures, we sample sequences of packets from only one packet capture of each dataset. This may not seem like much but as we only plan on creating a small proof of concept the data should be sufficient. The sampled packet sequences and flows are stored separately as csv files for further preprocessing, as network packet captures are usually slow to process. For our purpose, we considered two approaches for sampling a network packet capture for flows. Either by sampling through the packet sequences within the captures or by splitting the capture into intervals and sampling through those partial captures. Splitting the capture into intervals is less compute-heavy and therefore less time-intensive since the resulting packet captures are smaller and much quicker to parse. Although this might truncate some flows, it should not be a problem, since a capture itself is truncated at the beginning and end anyway and we also use timeouts (20 seconds) for the flow collecting, which will truncate the flows once again. Additionally, we do not require flows longer than the intervals themselves, as we want to keep our packet sequences short enough to be reasonably processed by an RNN. Later during the further preprocessing, we drop any flow longer than a maximum number of packets (500), to ensure an equal length for all our packet sequences.

The question however is the length of the intervals. Should they be determined relative to the size of the capture or should they have a fixed length? A fixed-length may provide

better comparability between the flows across different captures, but the variation between packet captures regarding packet rate (frequency) may prove to be problematic. As setting the fixed length to low may yield the problem that some captures return empty intervals and setting it too high might result in too large intervals and therefore in less efficiently parsable partial captures. An interval length based on the percentage of each capture avoids these issues. Even though it might shrink the flow duration of smaller captures, it appears to be the better approach. In addition, extracting packet sequences with a relative offset within a capture might provide sequences with a wider range of flow behavior. Now that we know we want to split the capture first, it is to be decided, how we sample the packet sequences within an interval. We can make an educated guess that randomly selected samples will yield better results than sequential sampling, as it again provides a wider range of flow behavior.

So for our dataset, we decided that splitting the capture into intervals first would be the right approach. We split the captures into intervals relative to its size. This way we will preserve the behavior of the sampled flows within the larger network within each specific interval. Afterward, we will randomly sample up to 250 of TCP and UDP packet sequences alike within those intervals. Please note not all source files provide enough flows per-protocol though, specifically UDP packets seem to be less prominent. Each sampled packet sequence will be exported as a capture first using Wireshark's `editcap`. The header fields of the packets belonging to the sequence will be stored as a csv file for easier parsing, exported by Wireshark's `tshark`. Then we use YAF to extract the flow information and `yafscii` to create flow-based csv files form the exported capture.

A Padding supported Conversion from UDP to TCP

Each datapoint in our dataset can either be a TCP or a UDP packet sequence. Compared to UDP packets, TCP packets have more header fields and therefore more features get derived from TCP packets compared to UDP packets. This results in an uneven timestamp size or length of packet data between two datapoints, if one is based on a UDP packet and the other is a TCP packet. To make packets of both protocols comparable, we will add padding to the UDP packets. We add the TCP flags and options as well as the sequence number and window size at the same positions where they would be in a TCP feature vector, more on that in section 5.1.2. They get assigned our padding value (-15), which will be used as a masking value during the input into the NN. To round it up we convert the UDP length to TCP header length and TCP payload length, see section 5.1.2 paragraph **derivation**. The last part is especially important to keep the datapoints comparable and

not only at the same length. This way, the NN can make the right decisions based on the data and is not held back by matching single features between the two protocols. As our positive datapoints are ready for processing now, all that is left to do to complete our dataset is the creation of the negative samples.

Creation of negative samples

To create negative samples we use four different methods. The selection of ground truth samples that will be falsified is randomized in all methods. The first method is **altering packet and flow features** by either sampling features **from another datapoint** of the dataset (**method 1b**) or generating **random values** (**method 1a**), which lie within a rough frame of expectation for the feature. For the frame of expectation, we used the number of bits available to the header fields, as they provide the hard limit to the range of the feature's value. **Method 1** is able to produce three different types of negative samples:

- a datapoint, which has altered packets
- a datapoint, which has an altered flow
- a datapoint, which has both altered

Please note that we later decided not to alter the flow as the RNN should learn if a packet sequence could have been created by a given flow and also detecting altered flows would lead the classifier to learn unnecessary information. Furthermore in the scenario of our future GAN, the flow input is authentic and does not need to be classified itself for authenticity. However dropping altered flows did not change the performance of our model, so the model would likely perform the same through the later experiments if the altered flows were present in our dataset.

The second method is randomizing the **packet order** (**method 2**) within the packet sequence of a given datapoint. The third is to **decrease** or **increase** the number of packets of the sequence in a given datapoint (**method 3**). To decrease the number of packets we just remove a random number of packets at a random position. To increase the number of packets we again sample from another datapoint in the dataset and add some of its packets to the current sequence. And the fourth and final method is **mismatching** network flows with a different sequence of packets (**method 4**).

Each negative sample creation method has one or more parameters, which decide the degree to which it alters the ground truth sample. The degree of **method 1** is defined

by the number of errors assigned to every packet and the number of altered packets. The degree of **method 3** is defined by the number of packets added to or removed from the sequence. While adding packets was done by a total amount, removing packets was done relative to the sequence size, as having a lot of negative samples with sequences of small size might cause the NN to learn that small sequences are per default negative. The **methods 2 and 4** have no degree, they are completely random and have no parameters adjusting the severity of the error.

The parameters used to change the degrees stay not completely static either. We randomize them within a given range that seemed appropriate to the parameter. For example, adding packets using **method 3** adds $+/-10$ packets, so using the parameter 0 packets equals adding 1 to 10 packets at random, while using the parameter 15 would add between 5 and 25 packets. **Method 1** uses a range of $+/-5$ for the number of errors per packet. Initially, the number of packets altered was a static one, but later we did the same $+/-5$ range for the numbers of altered packets, for the creation of the **20 packets** dataset, see section 6.1.

Using different methods to create negative datapoints provides us with a good mix of negative datapoints. This is important in order to effectively train the NN, as it will face many different errors in packet sequences in a real-world application. However, it is important to falsify the data realistically, so that the NN is efficiently tested for future use within a GAN. For this, we need to provide a good representation of each feature from packets and flows alike.

5.1.2. Datapoint Features

In this section we discuss, which features will be part of our dataset and how they need to be derived, so that they could be used as input for the NN. We will generally focus only on IPv4 traffic to keep it simple. Also, we will focus on the TCP and UDP protocols for now. All network traffic features influencing network flow behavior will be of importance to our NN. This property will therefore decide the datapoint features. Network flow attributes and sequences of packet header fields will be the source for our datapoints.

Packet sequence features

Each packet contains a variety of header fields depending on its protocol, the position within a communication stream, and the options chosen by the communicating endpoints.

It is to be decided, which of those header fields are of importance to our goals. If we just wanted to validate that a sequence of packets belongs to a flow, we could argue that the only header fields interesting are the ones that are present in the flow representation as well. But for the future work GAN, this would not be enough. Instead, we want our NN to learn the correlation between all packet header fields and the limited flow features. This will enable the NN to be used as a DN for the proposed GAN later. One could argue that to properly correlate between flows and packets we need all of the header fields of each packet. However, most of the header fields can be trivially verified by means of direct comparison to the flow features (e.g. addresses), order checking (e.g. acknowledgement numbers) or similar methods. For this reason, some header fields are insubstantial for our use case for example, the addresses of the communicating endpoints (source and destination). This would include MAC and IP addresses for example.

Neither extreme would be the right approach. Instead, we suggest sorting the packet header fields and flow attributes into the following different categories: **Not Used** (gray), useful but **Trivial** (red), **Dropped** (purple), **Feature Derivations** (yellow), and raw **Features** (blue). We color-coded the five categories in order to give a more direct overview of membership for the packet header fields in tables 5.1 to 5.5 present. In the following paragraphs we briefly explain the meaning of each category and explain the membership of packet header fields.

Not Used (gray) are header fields that do not change ever or at least not in our scenario. This includes the Version of the IP Header since we only take IPv4 traffic into account, and the reserved bits in the IP flags and the TCP header.

Trivial (red) header fields might be useful, but should not be fed to the NN. Either because they can easily be validated outside the NN and might slow the NN down for no benefit, or they do not alter the behavior of the network flow. This would include header lengths, urgent pointer, as well as MAC and IP addresses. The acknowledgement number order can be easily checked outside the NN and can be omitted as well.

Dropped (purple) fields could have been **Feature Derivations** (yellow), but were omitted due to a lack of data. A NN needs a large amount of data to make proper decisions, if the data is not sufficient and it encounters a feature for the first time it would at best classify the datapoint correctly with a 50% chance. The best case of insufficient

data is no data, so the NN would ignore the feature, but as such features would just be a **dead feature**, it is better still to omit it. For IP options and the TCP flag NS, also called the ECN-nonce, the data is inconclusive, as our dataset does not contain any flows with IP options and only three with an ECN-nonce. This however is not unusual, as those fields are rarely used in practice anyway. Our dataset supports our research results that this is indeed just normal network traffic behavior, since it is constructed from numerous network packet captures from statistically different intrusion detection datasets. Luckily network flows do not contain either of those header fields in their attributes, meaning it should not impact our results to omit the two. IP options would influence the behavior of a flow, but as our dataset does not contain any IP options, omitting them should not result in bad training results. It only means that in order to take IP options into account in the future our NN and preprocessing needs to be expanded. However, most of the implementation was already done and could be easily adjusted to fit this need. The ECN-nonce is the more acute problem since less than 0.0009% of datapoints use it, which would add risk for our NN to misinterpret the data, once it comes across such a datapoint. Omitting the ECN-nonce should not be a problem either, as the feature does not impact the behavior of the flow in any way that is not already indicated by the ECN flag and value, see later in this section.

Feature Derivations (yellow) are fields, which are important to our probkem, but can not be used as direct input to the NN. They need to be **derived** in some way beforehand. The raw data representation of those features is not optimal as NN input and needs to be abstracted first. The most prominent type of feature with this property is the categorical feature, but simple byte or string conversion can also already qualify for this category of header fields. The type of derivation can vary from header field to header field. The protocol for example can be represented as a single boolean as our dataset only contains TCP and UDP, where 1 stands for TCP and 0 for UDP. Take note that UDP length, which describes the total length of the UDP frame, will be derived to fit the TCP length fields. It is simple split into `tcp.len`, a Wireshark filter describing the payload size of TCP packets, and the data offset, which is equivalent to the TCP header length. This way packets of the two different protocols are comparable.

The checksums are a perfect example for a feature derivation that can be abstracted quite a bit. It does not matter to the NN what the checksum reads as the future work GAN should not have to generate checksums at all. It should generate the header fields that change the behavior of the flow. The checksum for the packets of the output sequence could be easily generated after the generation of the header fields and therefore outside

the GAN. But an invalid checksum may influence the flow of packets, in the sense that the packet might get requested again and will be resend. So for our DN it only matters if the checksum matches the header. So instead of feeding the NN the raw checksum and therefore teaching it how checksums work, we will provide its status in regards to validity with a simple boolean.

The ports are categorical features and need to be converted to a continuous data representation. This is done by sorting the port into the following categories based on port ranges defined by the IANA [28]:

- **system** ports (0-1023)
- **registered** or user ports (1024-49151)
- **ephemeral** or dynamic ports (49152-65535)

Additionally the ephemeral category is split into the following subcategories:

- **IANA** (49152-65535): the range of ports assigned as dynamic by the Internet Assigned Numbers Authority (IANA) [28]
- **linux** (32768-60999): ephemeral ports accessable on linux based systems through `/proc/sys/net/ipv4/ip_local_port_range`
- **bsdold** (1024-5000): ephemeral ports used by older (Free)BSD implementations (version < 4.6) [31]
- **winserver** (1025–65535): dynamic ports used by Windows Server 2000 [30] [34]
- **winxp** (1025–5000): dynamic ports used by Windows XP [29] and Server 2003 (until update MS08-037) [37]

Newer versions of Windows use the IANA range [30], so the we can spare a new feature for it. Only Windows Server 2008 with Exchange Server 2007 uses 1025–60000, which is still within the **winserver** category [30]. In future work one could expand this binary representation by adding subcategories to the system and registered ports. But this is out of the scope of this thesis.

It is important that we derivate TCP options to remove dead features and preserve the information that is important to our problem, as TCP options such as maximum segment size (MSS) and selective acknowledgment (SACK) may impact a network flows behavior, while others may never occur, just like the IP options described above. We collect the MSS, SACK and the timestamp options, as they are the most prominent and frequent options used and our dataset also does not contain any other TCP options. The timestamps

get converted to the relative timestamp based on the earliest timestamp in the packet sequence in order to compare them to the `frame.time_relative` filter option from Wireshark.

The Explicit Congestion Notification (ECN) [13] header field has a simple derivation, although an important one. It consists of two bits that are interpreted as an integer by Wireshark. However, the bits are a categorical feature comparable with the IP and TCP Flags. If neither bit is set, ECN is not supported by the sender. If either one bit is set by itself, ECN is supported by the sender. If both bits are set, there is network congestion on the route between the sender and the receiver. This leaves us with two boolean features as input for our NN: ECN capable transport and congestion encountered. If we look into the documentation of ECN, we will see that those are the only attributes the ECN provides [10] and our derivation in fact matches the information ECN supporting hard- and software extracts from the header fields.

The Differentiated Services Code Point (DSCP), formerly Type of Service (ToS), header field has a more complicated derivation. Its specification is open for experimental and future standard values. For this reason, it is divided into 3 pools of values, specified by the values last two bits [10]. The first pool (xxxxx0) contains standard codepoints and still unassigned values, reserved for later standards. The second pool (xxxx11) is reserved for local use or experimental codepoints. The third pool (xxxx01) contains codepoints for local use or experimental, but will be used for standard codepoints once the first pool is exhausted. We abstract this behavior with two features: local (or experimental) and standard. This allows us to keep the parameters for the pool category minimal. But the pools are not the only scheme by which DSCPs are categorized. There are five major categories of standardized DSCPs, of which two have ranked subcategories. The five categories **Low Effort** [4], **Class Selector (CS)** [20], **Assured Forwarding (AF)** [35], **Voice Admit** [3], and **Expedited Forwarding** [11] will again be used as boolean features for the NN input. This way we complete the binary representation of DSCPs. However, CS and AF are again ranked within their category. Both have an IP precedence class, which is ranked from 1 to 7, where packets of class 7 have the highest priority, and packets with class 1 have the lowest. AF only allows the first four classes (1-4) though. AF also has the drop precedence probability, which is ranked "Low", "Medium", and "High", or in our abstraction 1 to 3, and is used within one class, so to not overwrite the priority set by the IP precedence, but to rank within one such class. As the name suggests the higher the precedence probability class the more likely the packet is dropped, so here the higher number has the least priority. Both of these classifications will be represented as an integer value.

Features (blue) are fields that we can use as unmodified or raw input. They are numerical (integer, float) or truth based (boolean) and can therefore be interpreted by a NN. But not all numerical or boolean fields automatically qualify as features. For example the transport protocol is represented as a numerical value, but it would be ineffective to train a NN based on that number. It would try to perform operations on it like on any integer value, but that would lead to wrong results. In the best case it would to no result, while creating overhead during computation. The fields qualifying for this category need to be continuous data. Additionally to the packet header fields shown in the table we also considered some meta data. As mentioned before we use the relative time offset of the packets, this feature should match with the duration of the flow, and the TCP payload length, provided by Wireshark's filter `tcp.len`.

The identification IP header field uniquely identifies a network packet compared to other packets from the same source. As it is just counting up, one could suggest, that it is not more of a feature than the acknowledgment number. However in a network packet capture it is possible that a source's packets are not monitored continuously, for example behind a NAT, thus their intermediate packets would not be recorded. This would require a more sophisticated approach for generating IDs for synthetic packets. Therefore we suggest letting the future GAN learn to create such an ID itself in this case, our NN needs to learn the relationship between those IDs too.

The IP and TCP flags are already in binary representation and can be fed to the NN as is. The TCP flags are of more importance than the IP flags though, as our flow representation also includes them. As fragment offset, the time to live (TTL) value and the window size are as important to flow behavior as the flags and are all numerical, we include them as features as well. The sequence number could just as easily be compared to the initial sequence number contained in the flow data. However, the position of the sequence number in the packet sequence can decide if a flow is valid or not.

The data offset and IP header length (IHL) are indicating that TCP and IP options should be present, this provides interesting context between packet (header) length and other features. Combined with the total length it also gives us information about the payload size. As the UDP length will have been converted to TCP length and data offset, packets of both protocols will benefit from this. Now that we know the network packet features and their derivations, we need to apply the same principles to network flows, the counterpart to a packet sequence.

Network flow features

Network flow features will be classified in the same categories as the packet header fields. Table 5.6 shows the color-coded flow features. In this work, we need to abstract flow behavior from a sequence of packets and compare it to the features of a given flow. As a flow has a higher abstraction from network traffic than packets, we can omit fewer features. Please note however that the reverse features, which a YAF flow representation provides, will be omitted anyway due to the fact that it does not relate to the packet sequence of the flow, but to the counter part of it. Meaning the flow from destination to source.

IP addresses and timestamps might mislead our classifier, as a flow might have almost the same behavior even though it has two different endpoints communicating or was sent at another time. As mentioned in the previous section (IP) addresses can be easily compared outside of the NN and are therefore **trivial** (red). The same applies to the timestamps. Although timestamps have an additional problem. As they are large floating-point numbers, they might slow the NN down.

Duration, round trip time, as well as packet, and byte count are already present as numerical continuous data, therefore, they can be used as raw **features** (blue). The initial sequence number falls under the same category just as their header field counterpart did.

The rest of the features are to be **feature derivations** (yellow) and will be derived in a way that fits the derivations of their packet field counterparts. The protocol and ports are derived the same way the packet derivations did it. A YAF flow contains the TCP flags of the first and last packet of the sequence of packets is was created from. The flags are provided as strings by yaf containing the initial letter of each active flag: **F**IN, **S**YN, **R**ST, **P**SH, **A**CK, **U**RG, **E**CE, **C**WR. While a **0** means no flags were active. To conform with our previous binary representation, we create a boolean for each flag in the same order a TCP packet contains them. For further details on the derivation of protocol, ports, and TCP flags, see paragraph derivations in section 5.1.2. The end-reason describes the reason for the end of the packet sequence. It is given as a string and can have five different values:

- **idle** timeout
- **active** timeout
- **eof** interrupted by YAF's timeout or end of file
- **rsrc** premature flush if YAF's flow table maximum is reached

bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	Desitnation							Source					Type	

Table 5.1.: MAC Header (Ethernet II)

bit	0			8			16			19
0	Version	IHL		DSCP	ECN		Total Length			
32	Identification						0	DF	MF	Fragment Offset
64	Time To Live			Protocol			Header Checksum			
96	Source IP Address									
128	Destination IP Address									
160	Options (if IHL > 5)									
192										
224										
256										

Table 5.2.: IPv4 Header

- **force** for single UDP packet flow records that belong to a bigger flow

As this is a categorical feature and each end reason mutually excludes all others, we decided to use a one-hot encoding for the binary representation. Please note that the last two end-reasons do not appear in our dataset, so their derivations are dead features. This was only discovered after most experiments had already been conducted. However as stated before, dead features do not effect the results of a NN, but only introduce overhead. This can be adjusted in the future.

bit	0			4			7			16
0	Source Port									Desitnation Port
32	Sequence Number									
64	Acknowledgement Number									
96	Data Offset			Reserved			Flags			Window Size
128	Checksum									Urgent Pointer (if URG set)
160	Options (if data offset > 5. Padded at the end with "0" bytes if necessary.)									
...										

Table 5.3.: TCP Header

NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN

Table 5.4.: TCP Flags

bit	0	16
0	Source Port	Desitnation Port
32	Length	Checksum

Table 5.5.: UDP Header

Start Timestamps
End Timestamps
Duration
round trip time
protocol
source IP
source port
destination IP
destination port
TCP flags (forward only)
initial TCP sequence number (forward only)
first-packet 802.1q VLAN tag (forward only)
packet count (forward only)
octet/byte count (forward only)
the reason for the end of the packet sequence.

Table 5.6.: YAF IPFIX Template

5.1.3. Network Packet Captures used

For the creation of our datasets, we need network packet captures as sources. For this, we take a look at publically available intrusion detection datasets. Since the goal of our future work is to produce network packet captures, that can be used in the field of network intrusion detection. As stated above we will extracted data from multiple popular datasets, such as: **CDX**, **DARPA**, **MACCDC**, **MAWI**, **Simpleweb**, and **UNSW**. We chose these datasets because of their significant statistical differences, which has been shown in the paper "On generating network traffic datasets with synthetic attacks for intrusion detection" by Cordero et al. [8]. This is important as we need to provide the NN with enough data to distinguish flow behavior in order to match packets to them.

CDX The data capture from the U.S. National Security Agency (NSA) published in 2009 [9]. `CDX westpoint/2009-04-21-04-06-19.dmp12`

DARPA The Intrusion Detection Data Sets from 1998 and 1999 [1]. This dataset provides PCAP files that were captured in a simulation network. We chose a training capture that does not contain any attacks from Friday of week one, specifically the outside capture [19].

MACCDC The U.S. National CyberWatch Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC) dataset. We sampled from PCAP 03 from 2012 [21].

MAWI The MAWILab dataset is backbone traffic with labeled anomalies from the Fukuda lab in Japan [18]. They provide fifteen minutes of traffic every day since 2010. The capture we used is from June 2015 [17].

Simpleweb This dataset contains mostly flow data, but also provides some network captures. The traffic was collected at a honeypot hosted in the network of the University of Twente in September 2008 [16]. Sadly the source of the capture we used is no longer online. It was the `Simpleweb/traces_TCP-IP_location1/loc1-20020523-1835` tcpdump.

UNSW-NB15 A NIDS dataset with over 100GB of network traffic at packet level that includes multiple types of attacks, such as Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode and Worms [32]. We sampled through a 1.9 GB network packet capture from this dataset. Pcap number 9 from 17.02.2015.

5.2. Neural Network

Our NN model is build around the structure of our data. This section explains the reasoning behind the input for the NN and the resulting NN model.

5.2.1. Input

To use our dataset with a NN we need to convert the datapoints into a data structure that is suitable as NN input. As NNs work best with continuous data we face the following problem when converting the data. The packet and flow features only available as bytes or a string need to be converted to an integer, a float or a binary representation, a sequence of boolean values that describes a categorical feature, fully or at least partially. This is already solved in the dataset creation. Afterward each datapoint has a general structure of a list of flow features and a sequence of packets, where each packet consists of a list of attributes derived from the data.

As RNNs work with sequential data and in our case only the sequence of packets is sequential, we need a more sophisticated solution to take our flow data into account. There are multiple approaches to this problem. One is to append the flow data to each packet in the sequence. But this creates an enormous amount of overhead and has also proven to be an ineffective solution in the past. Both Vinyals et al. [33] and Karpathy and Fei-Fei [15] claim that passing the condition to the NN with each timestamp of the sequence was less effective in providing results. Another approach would be to use two different NN layers in parallel to learn about the flow attributes and the packet sequence and merge the results afterward. In our case the packet sequence can only be verified as belonging to a given flow if the layer knows the flow attributes. Therefore the flow data should directly impact the packet sequence, so separating them would not be a good idea. The remaining and only liable approach is to modify the initial state of the RNN with the flow data as proposed by Vinyals et al. [33] and by Karpathy and Fei-Fei [15]. Both used the output of a CNN, that provided a continuous vector representation of an image, as a

conditional input for their RNN. We won't have to do this as our flow data is already in a continuous state after the dataset creation. However our data preprocessing is not done yet. As any RNN our condRNN requires datapoints in uniform shape to perform optimal.

The batch size can be a powerful tool to adjust the training of a NN. The datapoints within one batch need to be even in shape however. So when facing datasets with non-uniform shaped datapoints, such as ours, one would need to create batches based on shape instead of randomly. Even worse if a datapoint has a unique shape it would need to be processed alone. So an uneven datapoint shape leads forces us to use only one sample per batch or create batches of samples with similar sizes To prevent this each sequence and the number of features per input shall be the same for each datapoint. The shape of our datapoints varies on two accounts: first the uneven number of features within a UDP or a TCP packet, second the length of a packet sequence. So we need to transform our datapoints to a uniform shape. Since UDP and TCP packets have a different number of features associated with them UDP packets need to be padded in order to coexist with TCP packets in our dataset. This is done in during dataset creation. It allows us to compare the behavior of UDP and TCP packets and provides a uniform number of features per input. To provide a uniform sequence length we need to add padding to the packet sequences of each datapoint to the maximum sequence size. For this we must limit our datapoints to a reasonable packet sequence length. After investigating our dataset we chose 500 packets as the limit, as this only excludes 0.58% of the datapoints. To understand our data after this padding our NN model needs two masking layers. One for ignoring the padding of UDP packets described in the previous section on dataset preparation. And one for ignoring the padding of the packet sequence length.

5.2.2. Model

For our proof of concept NN model we propose a straigth forward model. As mentioned above we need two masking layers, which filter the UDP feature padding and packet sequence padding. The masks of both layers are concatenated with a logical AND operation. The fully-formed mask will then be passed to the first condRNN layer in addition to the inputs. The number of condRNN layers and their units is to be determined in our experiments. The output of the last condRNN layer provides gets compressed by a dense layer with half as many rectified linear units (ReLUs). For the output layer we use a dense layer with one unit and a sigmoid activation as we are looking for a categorical result of 0 or 1. A schematic of our model can be found in figure 5.1. Now that we know the

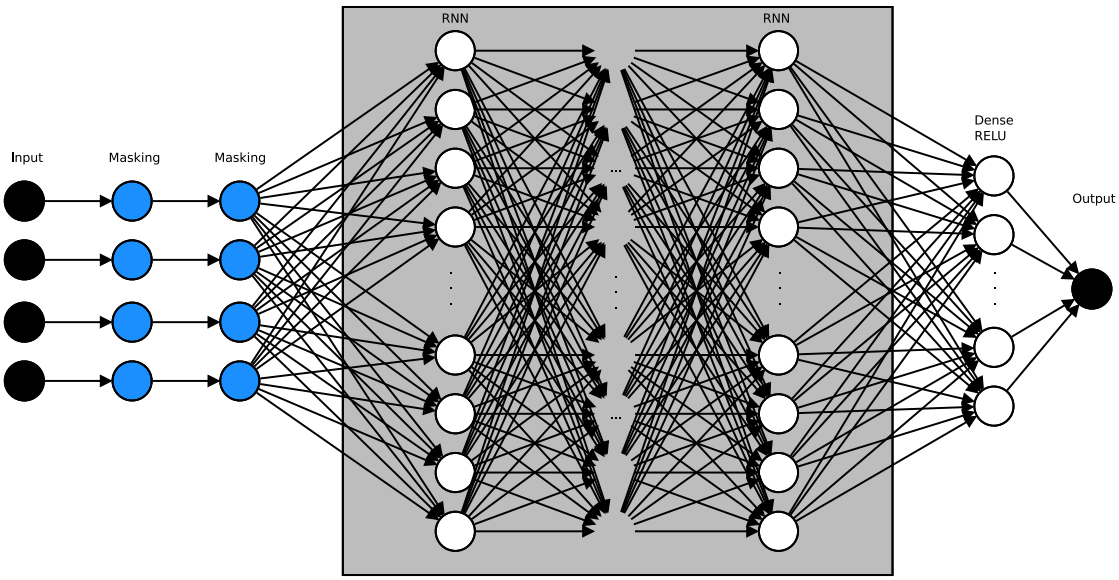


Figure 5.1.: Our NN Model has two masking units per input, a number X of RNN units on Y layers, pedenting on the experiment, as well as half as much units of Dense RELUs and one Dense unit with the sigmoid activation on the output layer.

structure of our model as well as which layers to use, we need to figure out which NN architectures will be used in our experiments.

For our RNN layers, we will compare different architectures of RNNs. Using RNNs will ensure previous data points will be taken into account during learning and in the case of LSTMs and GRUs all previous datapoints will be taken into account. However, it is important to establish a baseline, which will help us establish a measure of progress. Therefore our baseline will be simpleRNNs. Afterward, we will compare our baseline results to the results of LSTMs and GRUs. As described in the Input section above, we will use a NN model, which accepts flows as conditional input and packets as sequential input. The `cond-rnn` library [25] written for Keras and Tensorflow and supports simpleRNNs, LSTMs and GRUs as cells for conditional RNNs. This library empowers us to compare simpleRNNs, LSTMs and GRUs for our problem.

6. Experiments

In this chapter, we discuss the experiments conducted on the dataset(s) and the NN models described in the previous chapters. We initially planned to compare the different RNN architectures, simpleRNN, LSTM and GRU. However as our model performed suboptimally, we first shifted our focus to debugging of the model, solving our problem with one of the architectures and finding the hyperparameters, for which our model achieved the best performance. Even though we still aimed to compare the three RNNs, we first needed to figure out, how one of the RNNs could be trained effectively. For this reason, most of our experiments are only done with an **LSTM**-based condRNN model, or conditional LSTM (condLSTM) model. We later compare the model to simpleRNNs and GRUs. All our model iterations are condRNNs.

So when we talk about simpleRNN, LSTM, and GRUs, we mean conditional simpleRNN, condLSTM, and conditional GRU (condGRU). When we talk about accuracy or loss in this chapter we generally talk about validation accuracy and validation loss, unless stated otherwise. As they were very similar to the training accuracy and loss, in most experiments slightly better and give a clearer understanding about the model's performance than test accuracy and test loss. Since the validation is repeated for each epoch and the testing is only done once after the model was trained, the validation can provide a look at the performance of the model over time.

6.1. Dataset statistics

For our experiments, we generated multiple datasets from our source packet captures. For testing on our workstation, we first generated a **base** dataset that was constructed from 5% of our available data. For the experiments on the compute server, we generated a **full** dataset from the complete set of data we sampled from our source packet captures. The **full** dataset has 378410 total datapoints, including 227499 positive datapoints and

150911 negative datapoints. To keep the compute time reasonable we used only 10% (**full10**) and 25% (**full25**) of the complete dataset at random. After achieving only poor performance during our experiments we generated three additional datasets.

Those were based on the same data as the **base** dataset, but as it had only minimal errors in the negative samples, we now generated datasets with more errors. First, we increased the number of errors per packet from 1 to **9 errors** per altered packet. Then, we increased the negative samples by a factor of three. This dataset we call **more negatives**. Alternatively, we generated a dataset that altered not just one but up to **20 packets** an average with random selection in the range of 15 to 25. See table 6.1 for the detailed breakdown of parameters used to create these datasets. Each dataset had an equal share of each method of negative sample generation described in section 5.1.1. The difference between the datasets is the degree of errors per negative sample and the number of negative samples added.

The datasets **base**, **9 errors**, and **20 packets** did not differ too much in regards to the number of datapoints and therefore in the ratio of positive to negative datapoints, as can be seen in table 6.2. The datasets **9 errors**, **more negatives**, and **20 packets** have 9 errors per packet, but only **20 packets** has more than one packet altered per negative datapoint. And the **more negatives** dataset is the only of the smaller datasets with significantly more datapoints, as more negative samples are generated to provide a greater sample pool per negative sample creation method.

We also generated a pure dataset for each method of negative sample creation (5.1.1). They all use the parameter configuration the base dataset used, which applies to their method. And except for **method 3**, which had only 16890 datapoints with 6606 negative datapoints, they all had 20568 datapoints total and an even split of positive and negative datapoints. We used these datasets to compare the different methods to create negative samples for out dataset. We named the datasets method X, where X is the number of the method used to create it. Note that **method 1a** refers to the creation of false packets by random data and **method 1b** refers to the creation of false packets by extraction from within the dataset, read more in section 5.1.1.

6.2. Experimental Setup

For the initial training of our model, we used only around 5% (base) and 10% (full10) of the data available to us as not to exceed the resources, specifically memory, of the systems

name	**errors** per packet	false **packets**	**reduce** percentages	**add** packets	sample **factor**
base	3 (1-8)	1	10%, 25%, 50%	0 (1-10)	1
9 errors	9 (4-14)	1	50%, 75%, 90%	50 (40-60)	1
more negatives	9 (4-14)	1	50%, 75%, 90%	50 (40-60)	3
20 packets	9 (4-14)	20 (15-25)	50%, 75%, 90%	50 (40-60)	1
full	3 (1-8)	1	10%, 25%, 50%	0 (1-10)	1

Table 6.1.: Negative sample statistics from datasets.

dataset **name**	**total** datapoints	**positive** datapoints	**negative** datapoints	**percentage used**	datapoints **total used**
base	18977	10284	8693	100%	18977
9 errors	18086	10284	7802	100%	18086
more negatives	29550	10284	19266	100%	29550
20 packets	16693	10284	6409	100%	16693
full10	378410	227499	150911	10%	37841
full25	378410	227499	150911	25%	94602

Table 6.2.: Datapoints per dataset.

we used and reduce compute times. The systems we used are a workstation with an AMD Ryzen 5 3600 6-Core @ 12x 3.6GHz and 32GB of ECC memory and a compute server with the Tesla V100S GPU also with 32GB memory. As our experiments progressed and we required more memory, we had to switch to CPU processing on the compute server. Giving us a share of the 48 threads @ 3.0GHz and 504GB of ECC memory. Even though we had longer compute times, this at least allowed us to run experiments with 25% of our dataset (full25) and use more than 96 units within an RNN layer. We later hoped to increase this even further to 50%, but the computation time would not allow it, as too much time was already spend on debugging and optimizing the model, see section 6.3.1 and 6.3.

Our model for the experiments consisted of two masking layers, which provided the RNN layer with the mask for UDP and packet padding, up to four condRNN (LSTM) layers with up to 128 units, a dense layer with ReLU activation and half as many units as an RNN layer, and a dense layer with a sigmoid activation unit for the output. This was manually implemented as a `keras.Model`, since the condRNN layer, which accepts two inputs, is incompatible with `keras.Sequential` at this point in time, as it only supports a single input. For the condRNN layer we used the `cond-rnn` library [25].

In our earlier experiments we used **sparse categorical cross-entropy** as a loss function, but this function was ill-suited for our problem. Once we figured this out we switched to the **binary cross-entropy** loss function, which is best suited for binary classification problems. Combined with a **sigmoid activation** for our output layer this helped us obtain better results. We also compared our results to the **hinge** loss function in combination with the **hyperbolic tangent** (tanh) function on the output and a conversion of the input targets from $[0, 1]$ to $[-1, 1]$ hoping for a steeper learning curve, as it provides a greater punishment for misclassification with the wrong signage. We thought this might be beneficial to our cause, but as it turns out it only led to worse results, see figure A.1.

Throughout our experiments, we always used 70%, 10%, and 20% of the used dataset for training, validation, and testing respectively, as it is the most common segmentation of data. We started with 5 to 20 epochs for initial testing of hyperparameter combinations and went up to 100. Once a combination of hyperparameters looked promising, we went up to as far as 1000 epochs. In each epoch, the whole training dataset is used and processed in batches of size 100 to 1000. Afterward, the model gets validated by the validation dataset. During training the weights get updated after each batch. For training we used Tensorflows `model.fit()` method. After all training and validation epochs are finished the model was evaluated on our test dataset. We soon came to realize that there is not much change to be seen after 100 epochs.

6.3. Evaluation of Results

At the beginning of our experiments, our model did not achieve more than 60% accuracy. The accuracy was most often pending around 57% over multiple experiments. As our **base** dataset (section 6.2) had the same ratio of positive to negative datapoints, we assumed the true accuracy of our model to be 50% at best, so not better as a coin toss. Usually, the validation accuracy increases over the first three to 20 epochs, depending on the experiment and its hyperparameters, then just hovers close to its average value, e.g. 57%, and flattens slightly above it. Sometimes it continued to increase although slowly, but even 1000 epochs weren't enough to reach a decent accuracy and resulted in the same behavior, see figure A.2. The validation loss behaved similarly it decreased quickly during the first epochs than slower and trends toward flattening close to its average.

We only reached an accuracy of 59% at best with one RNN layer of up to 64 units. During our earlier experiments, we concluded three or four layers to produce the best results again going any higher than four usually did not improve the accuracy and only produced additional overhead in computing time. With three layers and 64 units per layer, we were able to produce an accuracy of 60% to 63% within the first 15 epochs, while the loss was consistently decreasing, although slowly. Since the switch to binary cross-entropy, we saw the same behavior, only this time one or two layers seem to achieve better results than three or four layers, see figure A.4. As three and four layers did produce worse results with the new loss function, we did not increase the number of layers further, as we could make the educated guess that the performance would not benefit from additional layers.

We tried RNN layers with 32, 64, 96, 128, 192 or 256 units. However, we found anything above 128 did not provide better results in relation to the compute time. We believe the optimal number of units to be between 64 and 128. If we compare their performance in figures A.5 and A.6, we see that 96 units produce the best results on average and good validation accuracy over-all with a steadily decreasing loss. This goes for our model with both one and two layers as well as our base and full25 datasets.

We found the batch size of 350 to be most effective to achieve a good decrease of loss. As we can see in figure A.7 the batch size of 1000 provides the worst results by far. While 500 provides better results, it was not able to crack 63% accuracy. Everything under 500 shows more promising results, especially 350, which as an average validation accuracy of 63%. But going lower than that we see slightly worse results again, as we can see in figure A.7.

For the dataset(s) we conducted two types of comparisons. One the number of datapoints through the percentage of the complete dataset. Two the number of negative datapoints

and their error degree. For the first, we used our smaller dataset with 18977 datapoints, which is roughly 5% of our total data, and compared it to 25% of our total dataset, which amounts to 94602 datapoints. We saw an improvement of both accuracy and loss here, that manifested as a shift of the curve along the y-axis, see A.8. Only the general behavior of the model did not change. However, it is to be noted that for the larger dataset (full25) the number of layers and units seems to make a bigger difference in performance compared to the dataset with fewer datapoints. Increasing the units from 32 to 64 increases the average accuracy by 3.5 percentage points with the full25 dataset, while the base dataset only shows an increase of around 1 percentage points. The same goes for the increase of layers from one to two, although it is not as much of a difference, the ratio appears the same. For the second, we used the datasets with different configurations of negative samples described in section 6.1. As we can see in figure A.9 the more negative samples are in a dataset the better the average accuracy and loss. However, it does cause more fluctuation in the accuracy over time and the convergence toward the average worsens. We conclude that the improvement is merely based on the additional data available and was not on the increase of negative samples. Adding more altered packets to our first method of negative sample creation (**20 packets**) did improve accuracy and loss. Even though it gave a significant shift along the y-axis it again did not change the behavior of the model overall. Increasing the number of errors per altered packet from 3 to 9 did increase the model's accuracy a bit, but again only by shifting the curve along the y-axis.

Regarding different error types, we also experimented with datasets with negative datapoints created by only one of the methods described in section 5.1.1. The results can be seen in figure A.10. As we can see, **method 1a** and **method 2** caused the most problems for our model. This is of no surprise, as **method 1a** generates random numerical values within the scope of the header fields, meaning its size in bit, so the model might only see those numbers in those negative samples and **method 2** randomizes the order of packets and not their contents, which is hard to detect on the bases of a flow, especially for UDP. In fact, it might be impossible, see later in discussion section 6.4.
Method 1b on the other hand performs a bit better than average in comparison with the general results of the model. This again can be easily explained, as the method takes a number of fields from other datapoints in the dataset at random and adds them to an otherwise positive sample. The model has to struggle as it knows all values from other positive datapoints, but it also is able to pick up to which datapoints the values belong.
Method 3 and **method 4** perform the best in our model with up to 75% validation accuracy and an average validation accuracy of above 70%. This is interesting to note. However, they have samples they struggle with and drop the validation accuracy as low as 52.5% (method 3) and 56% (method 4), although mostly they stay above 65%. Their behavior

is the most complex compared to the methods. The good performance of both is most likely due to the lack of unknown data, meaning data that is not present in the positive datapoints. One could argue that **method 1b** should benefit from this as well, and it does, to a certain degree. However, **method 1b** has fewer and less obvious changes. It alters header fields of a few packets, in most cases - like in this comparison - just one. **Method 3** and **method 4** alter the sequence of packets, which is way more impactful, especially in regards to the recurrent architecture of our model. As for the problematic samples, we assume that they had only small changes. For example, they could have been created from positive samples with a small packet count and therefore only got a small reduction of packets (method 3), or two very similar flows got their packet sequence switched (method 4). As we have a lot of short TCP flows, we deem this likely.

When we compared our LSTM-based model to the same model with simpleRNN layers, we see very similar accuracy and loss (figure A.11). However, the difference lies in the detail here. If we look at the graphs, we can see that simpleRNN has a lot more fluctuation than our LSTM. This could be traced back to the vanishing gradient problem. The model using simpleRNN can practically only hold a limited amount of previous results in its memory and therefore struggles with data the LSTM has no problems with. Unfortunately, our model was not able to produce any results using GRUs since we switched to binary cross-entropy for the loss function. The loss with GRUs was not computed correctly, therefore the accuracy did not change. We instead ran the GRU with the hinge loss function, as we described earlier in this section, which at least was able to compute loss values. However, the accuracy did not even reach 25%, see figure A.12.

6.3.1. Debugging

To debug our model we performed multiple sanity checks. We tried a dataset using data from only one of the aforementioned NIDS dataset captures with no improvements. Changing the architecture from the conditional RNNs to a normal RNNs did not change the overall behavior of the model too much, only created slightly worse accuracy, so a problem with the `cond-rnn` libraries implementation can be excluded for the most part. This holds also for the manual model implementation, as a simple `keras.Sequential` implementation yields the same results. We also tried running without the masking layers, anticipating worse results, as our masking values -10 and -15 were no longer excluded from the NN layer operations. However, we saw no noticeable changes to accuracy. Possibly our NN model was able to notice the marking values as not relevant data on its own, without the masking layers. A higher number of ReLUs in the Dense layer also did not

provide better results. The model without the ReLU layer performed slightly better than with the ReLU layer, at least regarding the validation accuracy and loss, but the flattened curves remained. So we returned to our original ReLU implementation.

6.4. Discussion

The results of our experiments and debugging made us believe that the architecture of our model is at fault. It might still be a bug in the implementation, however it seems unlikely. One thing that might be worth changing is the masking solution. For the UDP to TCP conversion padding, we do not see a better solution. And at the time of the original design of the model, it seemed plausible to use the same approach for the packet sequence padding. However, after the experiments, we conclude that the approach of Karpathy [15] with the designated start and stop element at the beginning and end of the sequence might be a better solution. As long as it does allow larger batch sizes than one. The condRNN architectures should in theory be the best approach to the problem and as we need to handle sequential data with the packet sequences, one will not get around a recurrent model. However, there is a lot of depth to NN model implementation and we could not explore all of them. It might be that a different learning approach or optimizer would turn the page.

At this moment we do not believe our chosen flow and packet features to be at fault. However, as this was the first approach to classify if a sequence of network packets could have been created from a given network flow, it is possible that we missed something. And there are most likely improvements to be made in the future. But we doubt that they are so significant, that they will be the sole or even a major factor in the improvement of the model or the search for a different solution. We believe our research regarding network packet and flow features to be thorough.

As for the negative datapoints. It is no wonder that increasing the number of errors in negative samples increases the performance of the model. However, it can not be the solution to boost the model's performance by giving it easier samples to learn from. In order to perform as a DN of a GAN proposed in the problem statement 2.2 it would need to detect much smaller errors than we provided in our dataset(s).

The model has shown that with enough data and time, it will very slowly trend toward a higher accuracy. So the possibility remains that the data was not enough to obtain good results. Especially as we could only use up to 25% of our already small dataset. But as it

stands it is too inefficient to deliver good results in a reasonable amount of time and with reasonable resources.

7. Conclusion

7.1. Summary

We discussed the features of network packets and flows and their importance to the flow behavior and therefore to a classifier, as proposed by this thesis. We created a pipeline that generates datasets for use with NNs, that aim to classify if a network flow could have been created from a sequence of network packets. The datapoints of said datasets consist of a pair of one network flow and sequence of network packets, both are represented as a vector of features derived from the original data. Said pipeline is capable of sampling random flows from network packet captures, extracting flow and packet features, and creating negative datapoints.

We designed a NN classifier that attempts to distinguish if a flow could have been created by a sequence of packets. Our NN model utilizes conditional recurrent neural networks (condRNNs) in order to handle the input shape of our data. This NN model was intended be used as a DN for a GAN that creates network packet sequences from a network flow. And even though its performance is not sufficient yet, we believe that the model can be adjusted for this purpose in the future. As it stands now condLSTMs perform best for this problem. However GRUs may shine in more complex implementations.

7.2. Research Contributions

Even though the results were not optimal this work provides a prototype condRNN model that tries to distinguish if a network flow could have been created by a sequence of network packets. However, this is not the only contribution. The condRNN model was debugged and experiments regarding the hyperparameters for more effective training were conducted. These results should help build a NN model that is capable of solving our

problem in the future. The most valuable research contribution would be the discussion on network packet and flow features we take into account, as well as theory and practice of the derivation of said data. Furthermore, we will publish the pipeline for effectively sampling network packet captures and created negative samples, so that future research can benefit from it.

7.2.1. Answering the Research Questions

How can we develop a neural network that acts as a discriminator for a GAN that can determine if a network flow could have been created by a sequence of network packets? Once our NN model is efficiently capable of discriminating if a network flow was created by a sequence of network packets. It should be able to be used as a DN for the proposed GAN.

Which features of network flows and packets are (most) significant to determine if a network flow could have been created by a sequence of network packets? This question was answered in-depth in section 5.1.2.

Which NN architecture would be most suitable to distinguish if network packets could have created some network flow? Based on our experiments LSTMs provide the best results, more specifically conditional LSTMs (condLTMs). However, at best it could achieve 69% accuracy, so more experiments and research are needed regarding this question.

7.3. Future Work

This work sets the foundation for a variety of future work. Most of all the GAN proposed in the introduction.

7.3.1. Generation of network traffic captures with GANs

As stated many times throughout this thesis, we created a NN that when optimized could serve as a DN for a GAN that creates synthetic and authentic network packets from a network flow. The next step for this would be to implement a GN that can be used in

combination with our DN. It is also advised to gather more data to be more representative of the real world. The dataset we created is limited by its source data, which was fine for a proof of concept, but will not be sufficient when applied to real-world problems.

7.3.2. Federated Learning

Once the GAN proposed above produces good results, one could combine it with federated learning (FL). In the last couple of years, FL has become a widely used method to protect user data while training ML models. So one real-world application for the GAN could be to deploy it with NIDSs, routers of big ISPs, or the routers of personal users. This way, the GAN could be trained with real-world traffic without compromising the privacy of the data. The paper "FedGAN: Federated Generative Adversarial Networks for Distributed Data" by Rasouli et al. [24] introduces a federated GAN (FedGAN) that uses local generators and discriminators. This way, multiple users could add their traffic datasets without exposing their data.

FL in general still has flaws regarding privacy, but multiple researchers are working on improving the privacy of FL for example, in the form of differential privacy (DP). DP tries to preserve the user's privacy by compromising accuracy. The most common methods to achieve DP are clipping the weights, which will be sent to the server, and the addition of noise before averaging weights in order to anonymize the single contributions and minimize the extractable information about the user. Some libraries, e.g. PySyft, also provide homomorphic encryption, so the end-user might not have to train on their data themselves. Instead, they could send the encrypted data to an intermediate server, which computes the model with the encrypted data. Therefore also never seeing the data itself, while the computation heavy part is not done locally. This is specifically interesting if we consider using a FedGAN to train on data collected by consumer routers, as those usually do not have much computation power, to begin with.

With a GAN that also respects appropriately labeled intrusion detection network captures, combining this approach with FL could have the potential to solve the issue of creating synthetic network traffic for a self-learning anomaly-based NIDSs.

7.3.3. Possible Model Adjustments

As described a condRNN uses the conditional input to initiate the state of the RNN layer before the sequential input gets processed. This behavior was appropriate to the task of

image description generation [33] [15] as the sequence data only consisted of one or two sentences. At most those were 50 words long. Even with the addition of word2vec weights, this would most likely never be as much as the data we used. Our sequential data is only limited to 500 packets and each packet has around 66 features. The RNN may have forgotten all the flow information from the conditional input by the time it reaches the end of the sequence. One could propose an advanced condRNN that feeds the conditional input into the state every X number of timestamps or even with every timestamp of the sequence. In the future, we might explore this and look into how it affects the results of our NN model.

Our NN handling both the UDP and the TCP protocol allows us to have negative samples, in which the transport protocol was changed. We view this as an advantage of our design, as it also would allow the future additions of other protocols. However, in future work it might be interesting to compare our results to a NN model that consists of two RNNs, where one is handling UDP and one is handling TCP packet sequences.

7.3.4. Feature Improvements

As this thesis was a proof of concept, it leaves opportunity for future work regarding improvements and optimizations. Besides improvements to the NN model additions or removal of packet and flow features might yield interesting results. We see three categories where the work could be improved. Tailoring an IP2Vec-like approach for MAC addresses, IPs and ports, at least for public addresses. Especially in conjunction with FL.

Other possible improvements might be: Adding some of the trivial features as a binary representation with the purpose of indicating the validity of the feature. Bringing in more meta data extracted by Wireshark, or more network flow statistics. Exploring other network flow representations. Or as stated before additional subcategories for system and registered ports, for example categories for different services or the importance of said services. Also the deletion of the two end reasons that are not present in our dataset and other dead features, if any.

Bibliography

[1] *1999 DARPA Intrusion Detection Evaluation Dataset | MIT Lincoln Laboratory*. https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *arXiv:1409.0473 [cs, stat]* (May 2016). arXiv: 1409.0473 [cs, stat].

[3] Fred Baker, James Polk, and Martin Dolly. *A Differentiated Services Code Point (DSCP) for Capacity-Admitted Traffic*. en. https://tools.ietf.org/html/rfc5865.

[4] Roland Bless <roland.bless@kit.edu>. *A Lower-Effort Per-Hop Behavior (LE PHB) for Differentiated Services*. en. https://tools.ietf.org/html/rfc8622.

[5] Nevil Brownlee, Cyndi Mills, and Greg Ruth. *Traffic Flow Measurement: Architecture*. en. https://tools.ietf.org/html/rfc2722.

[6] Benoit Claise <bclaise@cisco.com>. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information*. en. https://tools.ietf.org/html/rfc5101.

[7] Carlos Garcia Cordero et al. "ID2T: A DIY Dataset Creation Toolkit for Intrusion Detection Systems". In: *2015 IEEE Conference on Communications and Network Security (CNS)*. Sept. 2015, pp. 739–740. DOI: 10.1109/CNS.2015.7346912.

[8] Carlos Garcia Cordero et al. "On Generating Network Traffic Datasets with Synthetic Attacks for Intrusion Detection". In: *arXiv:1905.00304 [cs]* (May 2019). arXiv: 1905.00304 [cs].

[9] *Cyber Research Center - Data Sets | United States Military Academy West Point*. https://www.westpoint.edu/centers-and-research/cyber-research-center/data-sets.

[10] *Differentiated Services Field Codepoints (DSCP)*. https://www.iana.org/assignments/dscp-registry/dscp-registry.xhtml.

[11] Victor Firoiu et al. *An Expedited Forwarding PHB (Per-Hop Behavior)*. en. https://tools.ietf.org/html/rfc3246.

[12] *Flow-Based Network Traffic Generation Using Generative Adversarial Networks - ScienceDirect*. https://www.sciencedirect.com/science/article/pii/S0167404818308393.

[13] Sally Floyd, K. K. Ramakrishnan, and David L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. en. https://tools.ietf.org/html/rfc3168.

[14] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[15] Andrej Karpathy and Li Fei-Fei. "Deep Visual-Semantic Alignments for Generating Image Descriptions". In: *arXiv:1412.2306 [cs]* (Apr. 2015). arXiv: 1412.2306 [cs].

[16] *Labeled Dataset for Intrusion Detection - SimpleWiki*. https://www.simpleweb.org/wiki/index.php/

[17] *MAWILab - Data Set - 2015/06/21*. http://www.fukuda-lab.org/mawilab/v1.1/2015/06/21/201

[18] *MAWILab - Home*. http://www.fukuda-lab.org/mawilab/.

[19] *MIT Lincoln Laboratory: DARPA Intrusion Detection Evaluation*. https://archive.ll.mit.edu/ideval/d

[20] Kathleen Nichols et al. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. en. https://tools.ietf.org/html/rfc2474.

[21] *PCAP Files from the US National CyberWatch Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC)*. https://www.netresec.com/?page=MACCDC.

[22] Michael Phi. *Illustrated Guide to LSTM's and GRU's: A Step by Step Explanation*. en. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21. June 2020.

[23] Jarno Rajahalme et al. *IPv6 Flow Label Specification*. en. https://tools.ietf.org/html/rfc3697.

[24] Mohammad Rasouli, Tao Sun, and Ram Rajagopal. "FedGAN: Federated Generative Adversarial Networks for Distributed Data". In: *arXiv:2006.07228 [cs, stat]* (June 2020). arXiv: 2006.07228 [cs, stat].

[25] Philippe Rémy. *Philipperemy/Cond_rnn*. Aug. 2020.

[26] Markus Ring et al. "Flow-Based Network Traffic Generation Using Generative Adversarial Networks". en. In: *Computers & Security* 82 (May 2019), pp. 156–172. ISSN: 0167-4048. DOI: 10.1016/j.cose.2018.12.012.

[27] Markus Ring et al. "IP2Vec: Learning Similarities Between IP Addresses". In: *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*. Nov. 2017, pp. 657–666. DOI: 10.1109/ICDMW.2017.93.

[28] *Service Name and Transport Protocol Port Number Registry*. https://www.iana.org/assignments/ser names-port-numbers/service-names-port-numbers.xhtml.

[29] *The Cable Guy - December 2005*. en-us. https://docs.microsoft.com/en-us/previous-versions/bb878133(v=technet.10).

[30] *The Default Dynamic Port Range for TCP/IP Has Changed in Windows Vista and in Windows Server 2008*. https://support.microsoft.com/en-us/help/929851/the-default-dynamic-port-range-for-tcp-ip-has-changed-in-windows-vista.

[31] *The Ephemeral Port Range*. https://www.ncftp.com/ncftpd/doc/misc/ephemeral_ports.html.

[32] *The UNSW-NB15 Data Set Description*. https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-NB15-Datasets/.

[33] Oriol Vinyals et al. "Show and Tell: A Neural Image Caption Generator". In: *arXiv:1411.4555 [cs]* (Apr. 2015). arXiv: 1411.4555 [cs].

[34] *When You Try to Connect from TCP Ports Greater than 5000 You Receive the Error 'WSAENOBUFS (10055)'*. https://support.microsoft.com/en-us/help/196271/when-you-try-to-connect-from-tcp-ports-greater-than-5000-you-receive-t.

[35] John Wroclawski et al. *Assured Forwarding PHB Group*. en. https://tools.ietf.org/html/rfc2597.

[36] *YAF - Documentation*. https://tools.netsa.cert.org/yaf/yafscii.html.

[37] *You Experience Issues with UDP-Dependent Network Services after You Install DNS Server Service Security Update 953230 (MS08-037)*. https://support.microsoft.com/en-us/help/956188/you-experience-issues-with-udp-dependent-network-services-after-you-in.

Appendices

A. Experimental Plots

In this section, we append all experiment graphs.

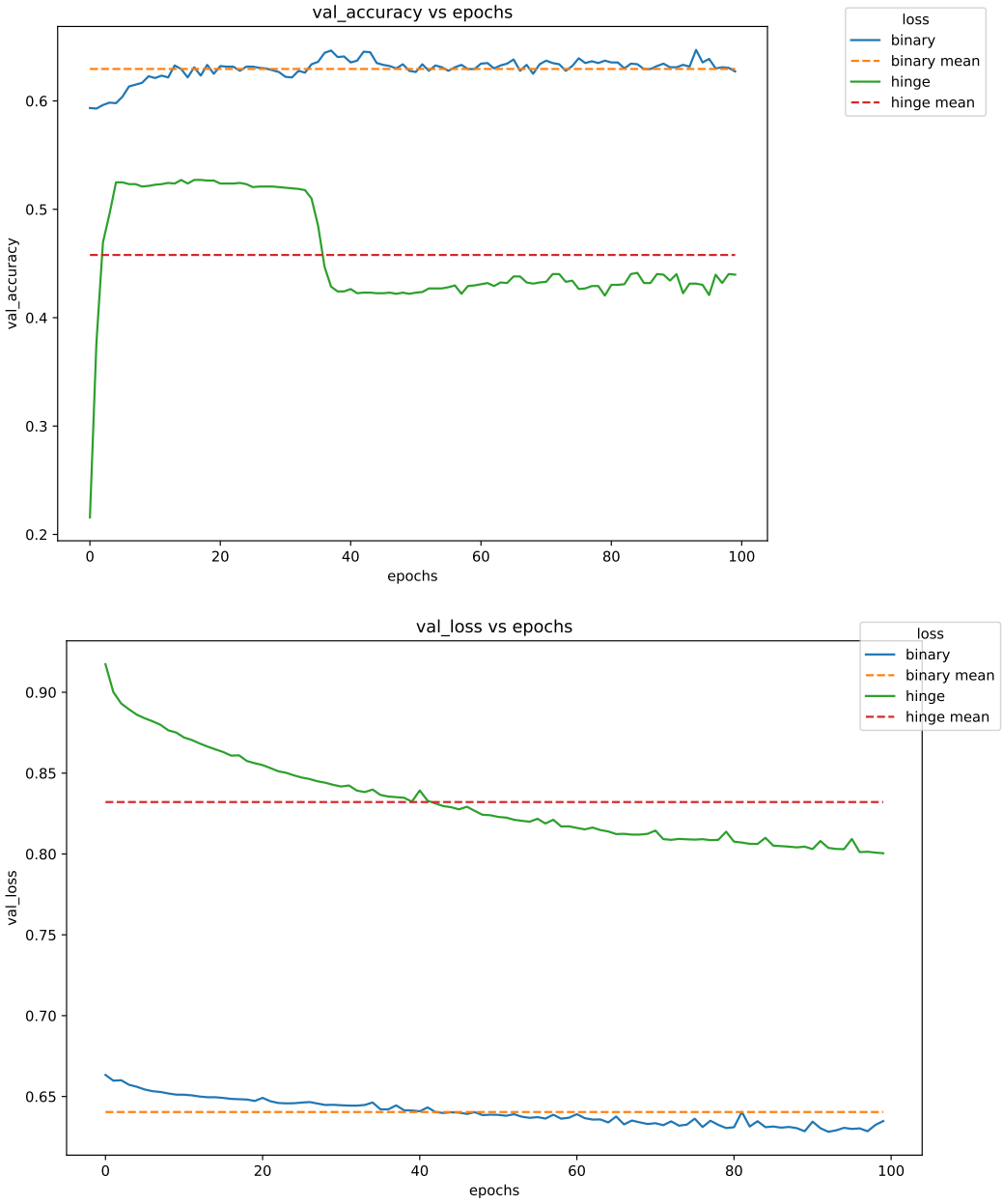


Figure A.1.: Comparison of both validation accuracy and loss between **binary crossentropy** and **hinge loss**. Our model with 2 LSTM layers, and 64 units per layer was trained over 100 epochs with a batch size of 350 and the **base** dataset.

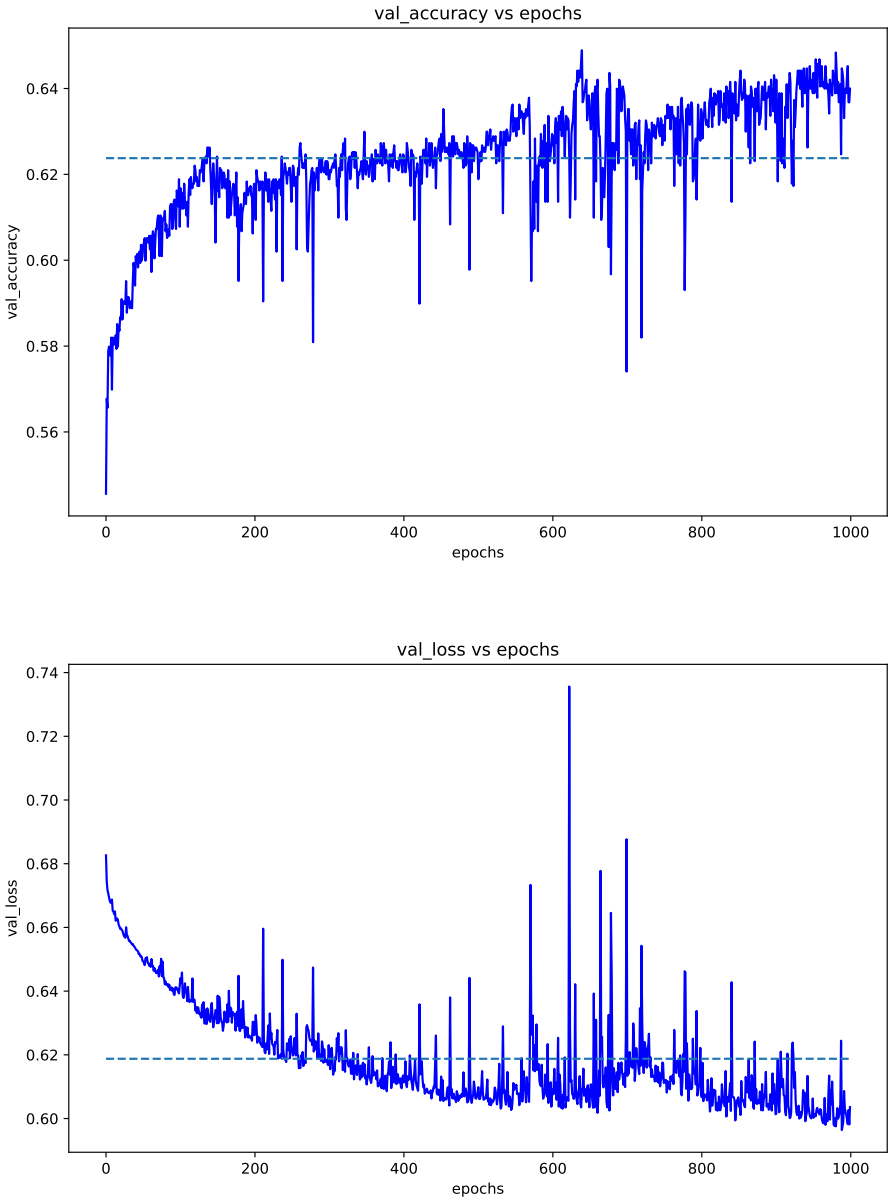


Figure A.2.: Validation accuracy and loss over 1000 epochs. Our model with 2 LSTM layers, and 64 units per layer was trained over 1000 epochs with a batch size of 350 and the **base** dataset.

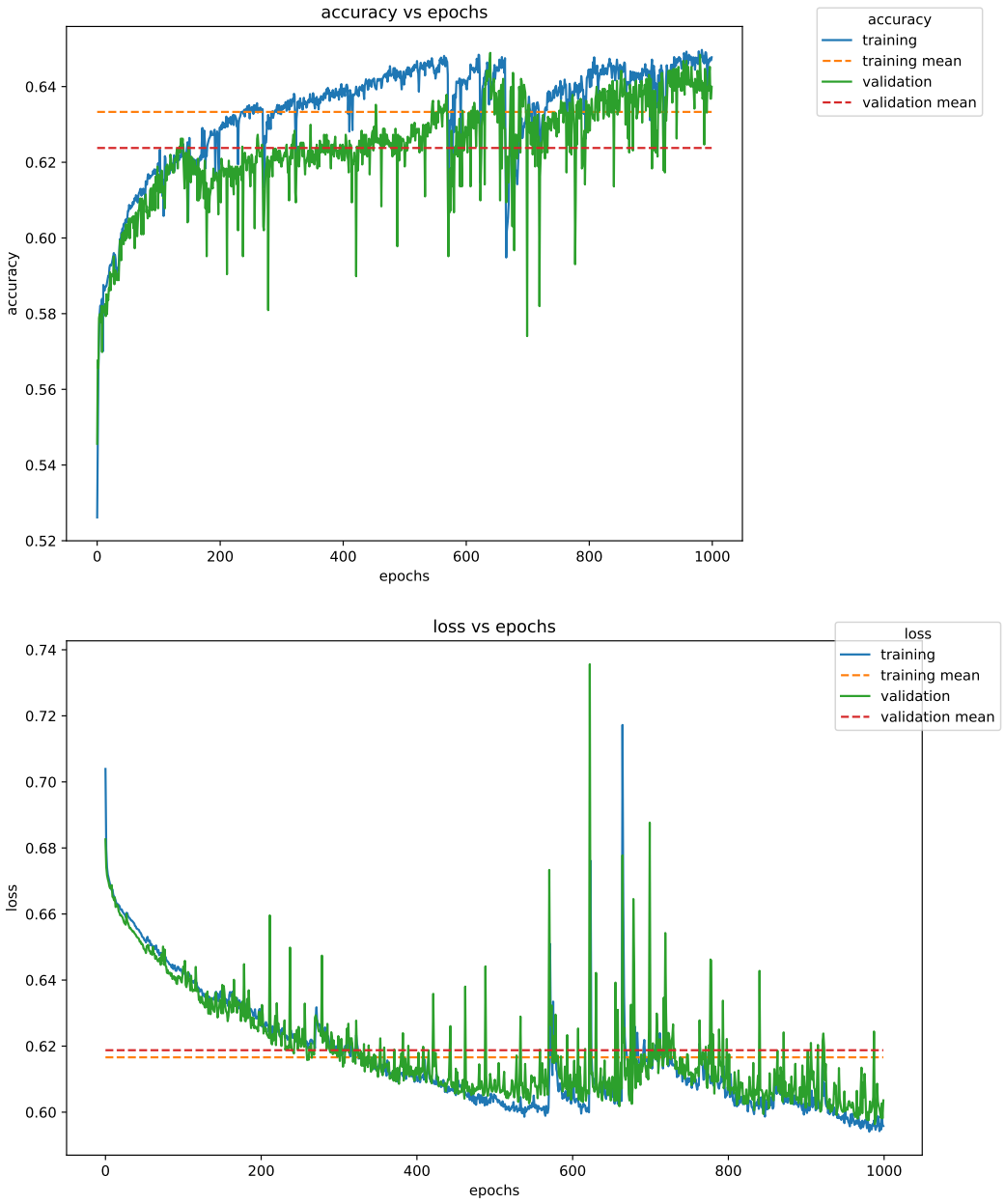


Figure A.3.: comparison of accuracy and loss over 1000 epochs between training and validation. Our model with 2 LSTM layers, and 64 units per layer was trained over 1000 epochs with a batch size of 350 and the **base** dataset.

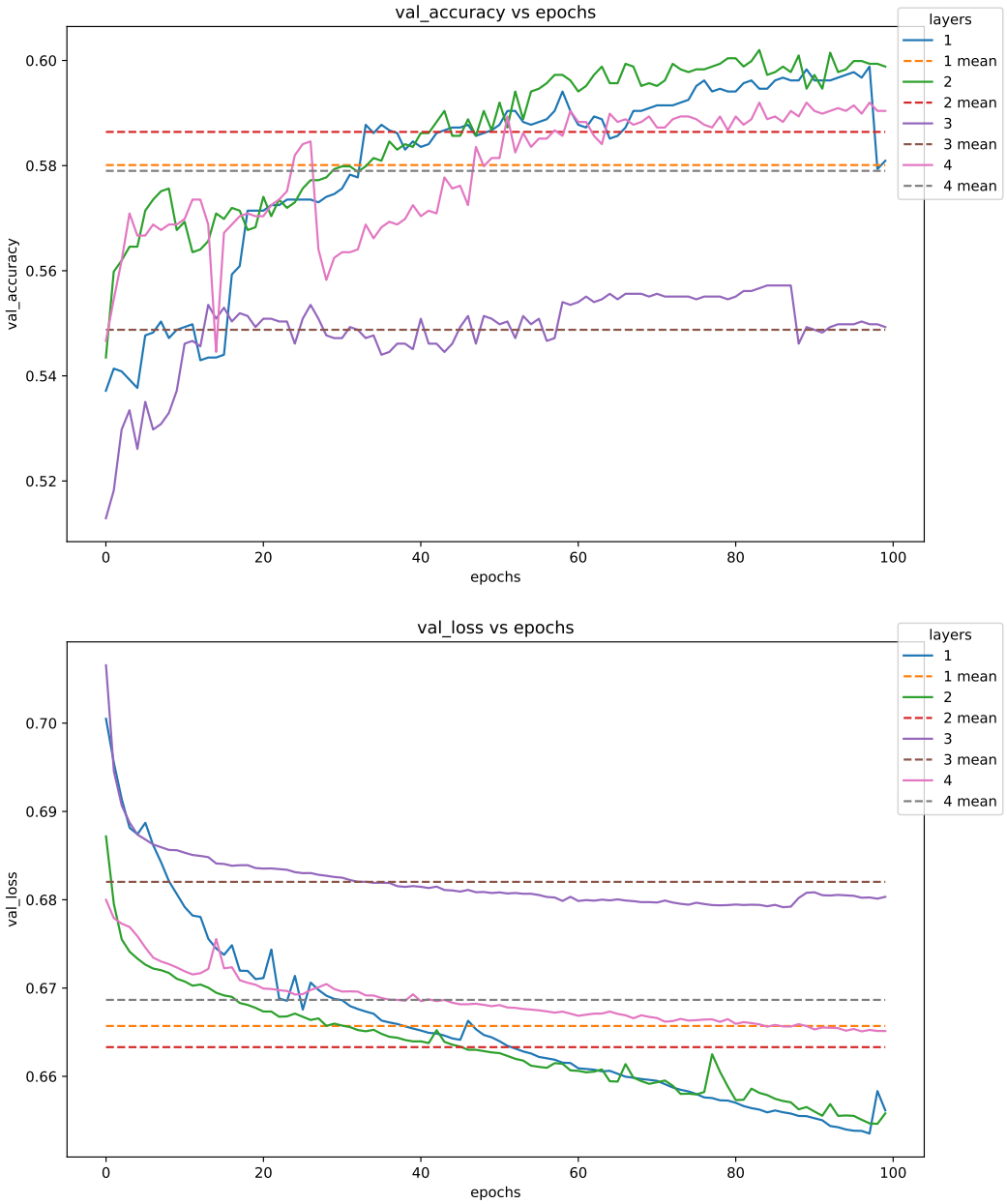


Figure A.4.: Comparison of both validation accuracy and loss between 1 to 4 layers. Our model with 64 LSTM units per layer was trained over 100 epochs with a batch size of 500 and the **base** dataset.

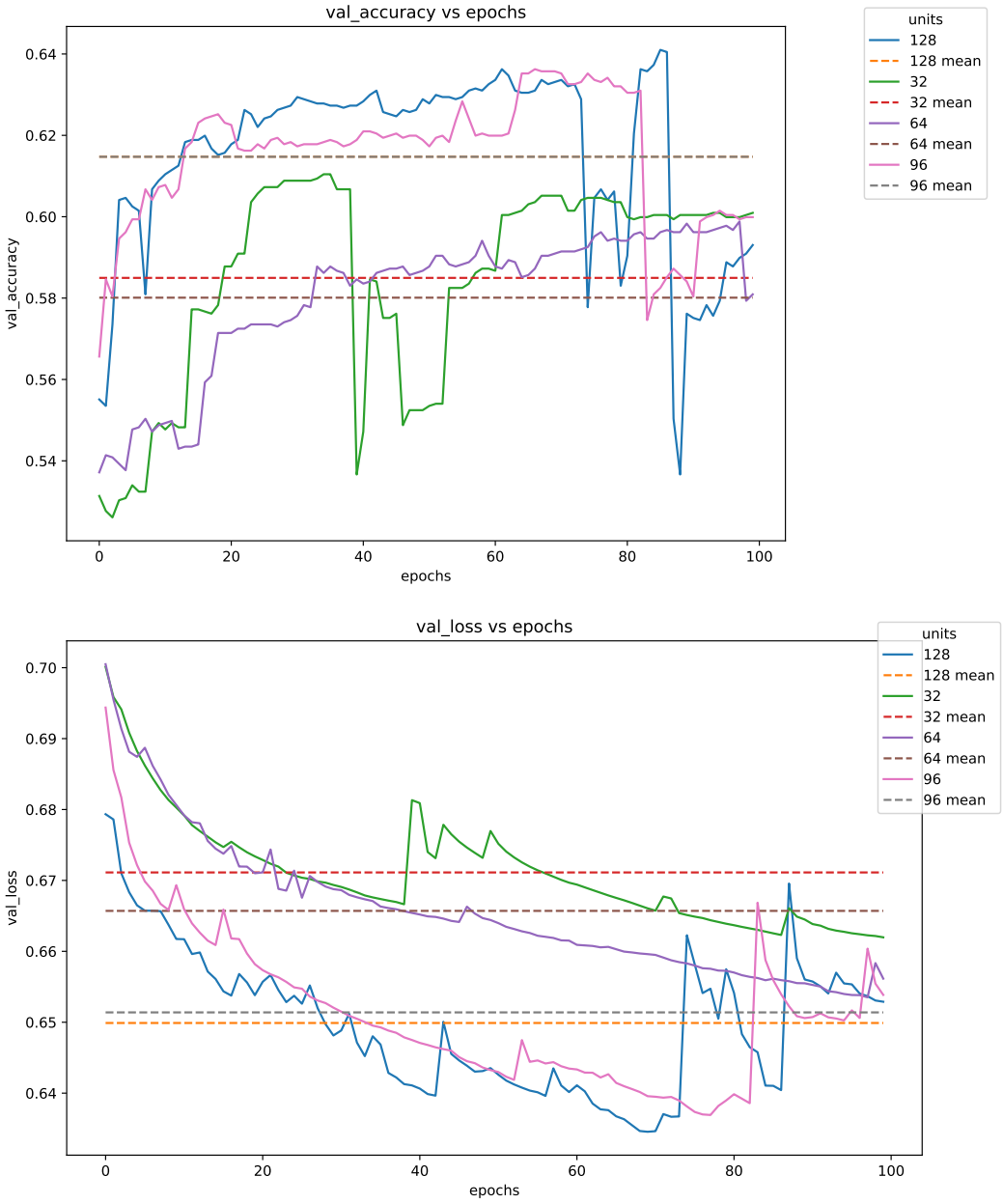


Figure A.5.: Comparison of both validation accuracy and loss between different numbers of units. For this experiment we used one RNN layer and trained with a batch size of 500 over 500 epochs.

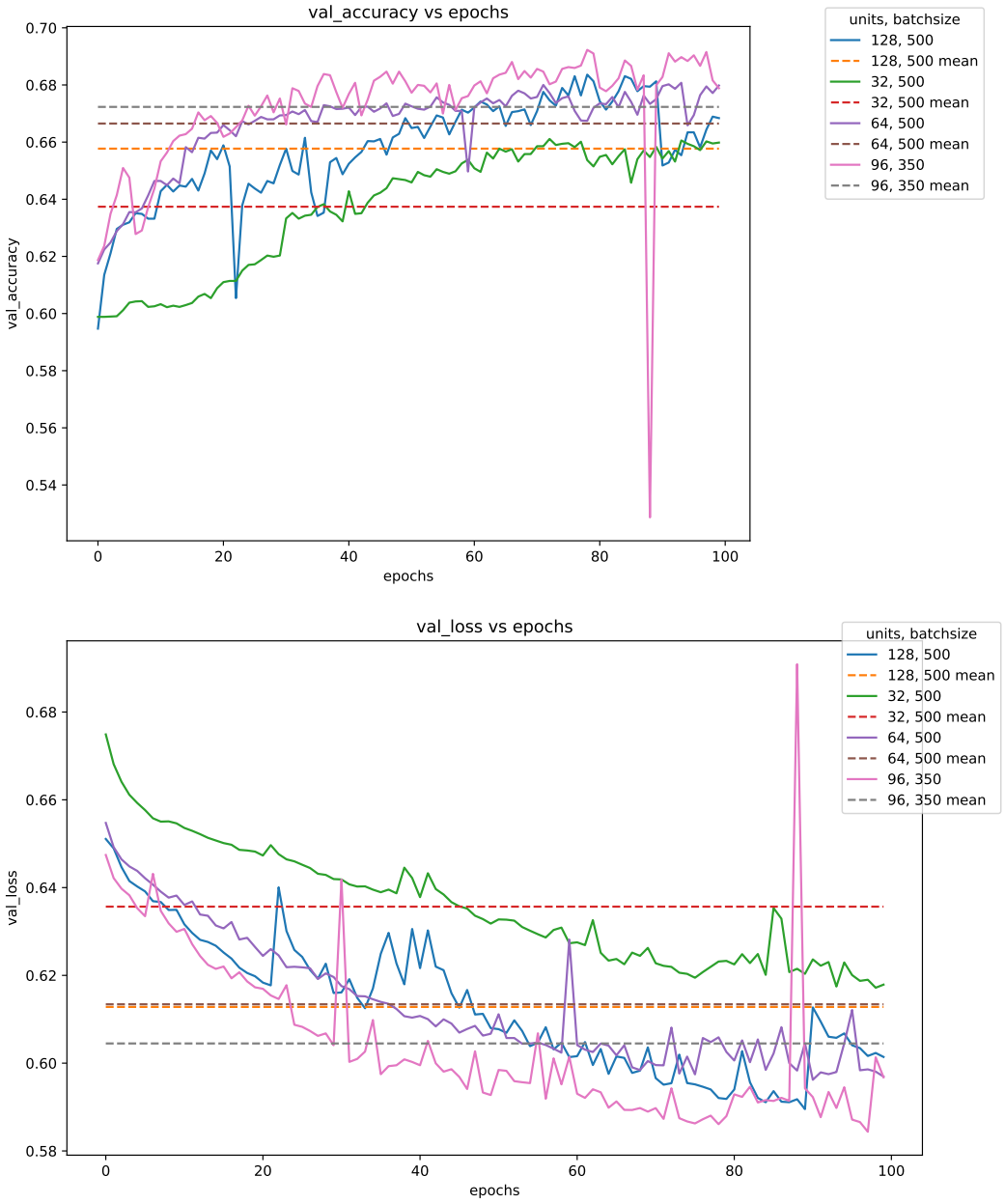


Figure A.6.: Comparison of both validation accuracy and loss between different numbers of units. For this experiment we used two RNN layer and trained with a batch sizes of 350 and 500 over 500 epochs.

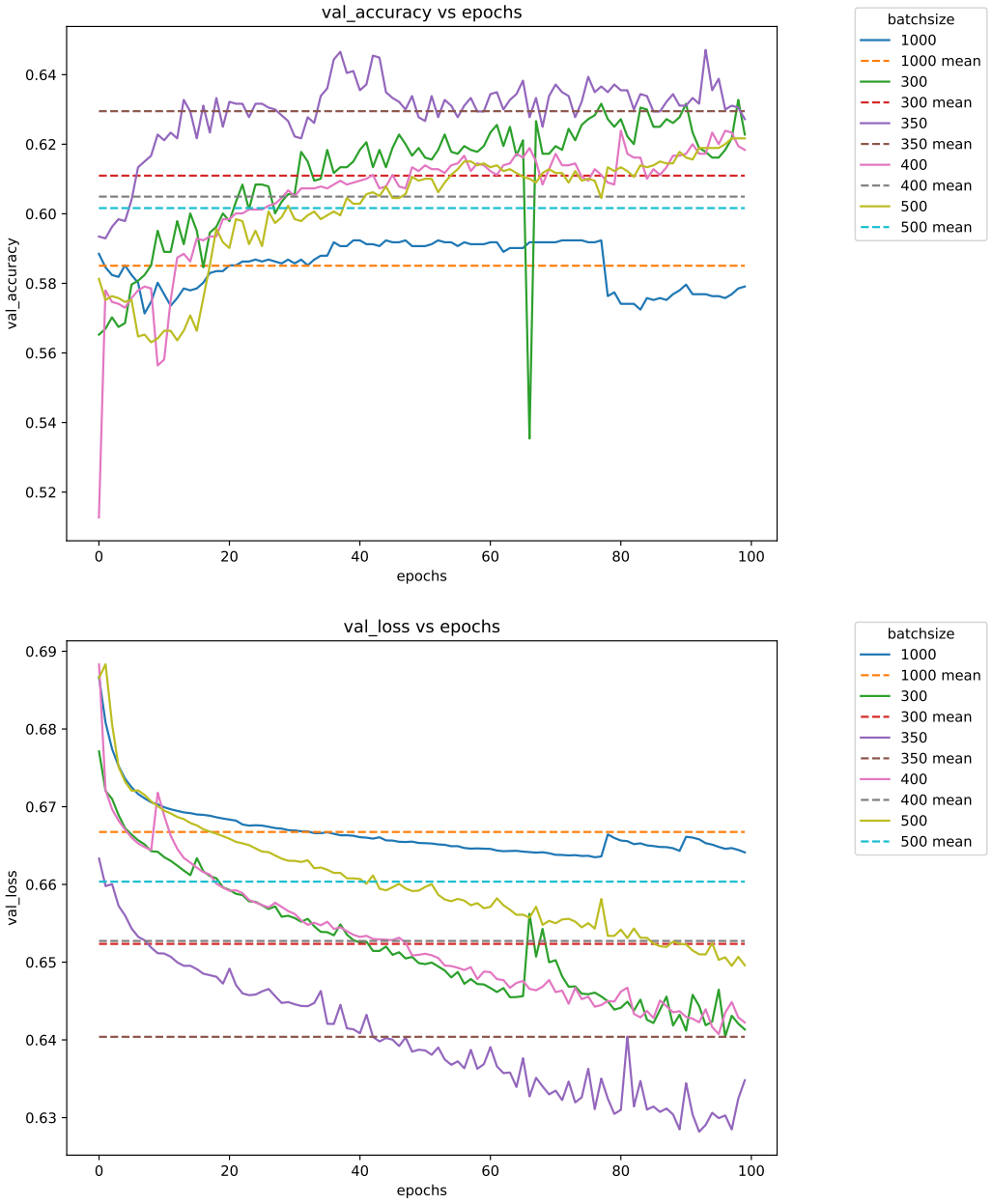


Figure A.7.: Comparison of both validation accuracy and loss between batch sizes. Our model with 2 LSTM layers, and 64 units per layer was trained over 100 epochs with the **9 errors** dataset.

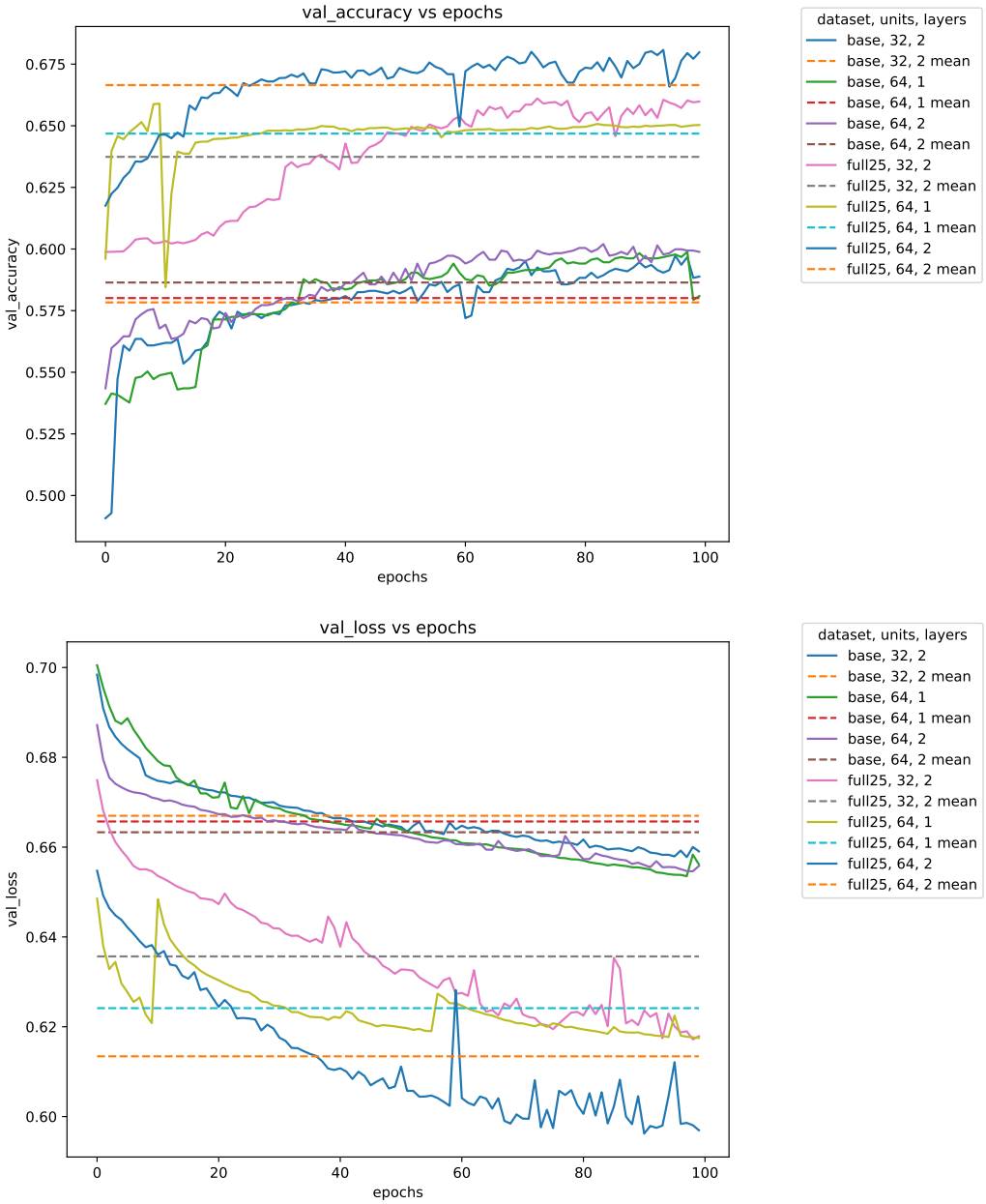


Figure A.8.: Comparison of both validation accuracy and loss between datasets **base** and **full25**. Our model with various combinations of number of layers and units was trained over 100 epochs with a batch size of 500 and the **base** dataset.

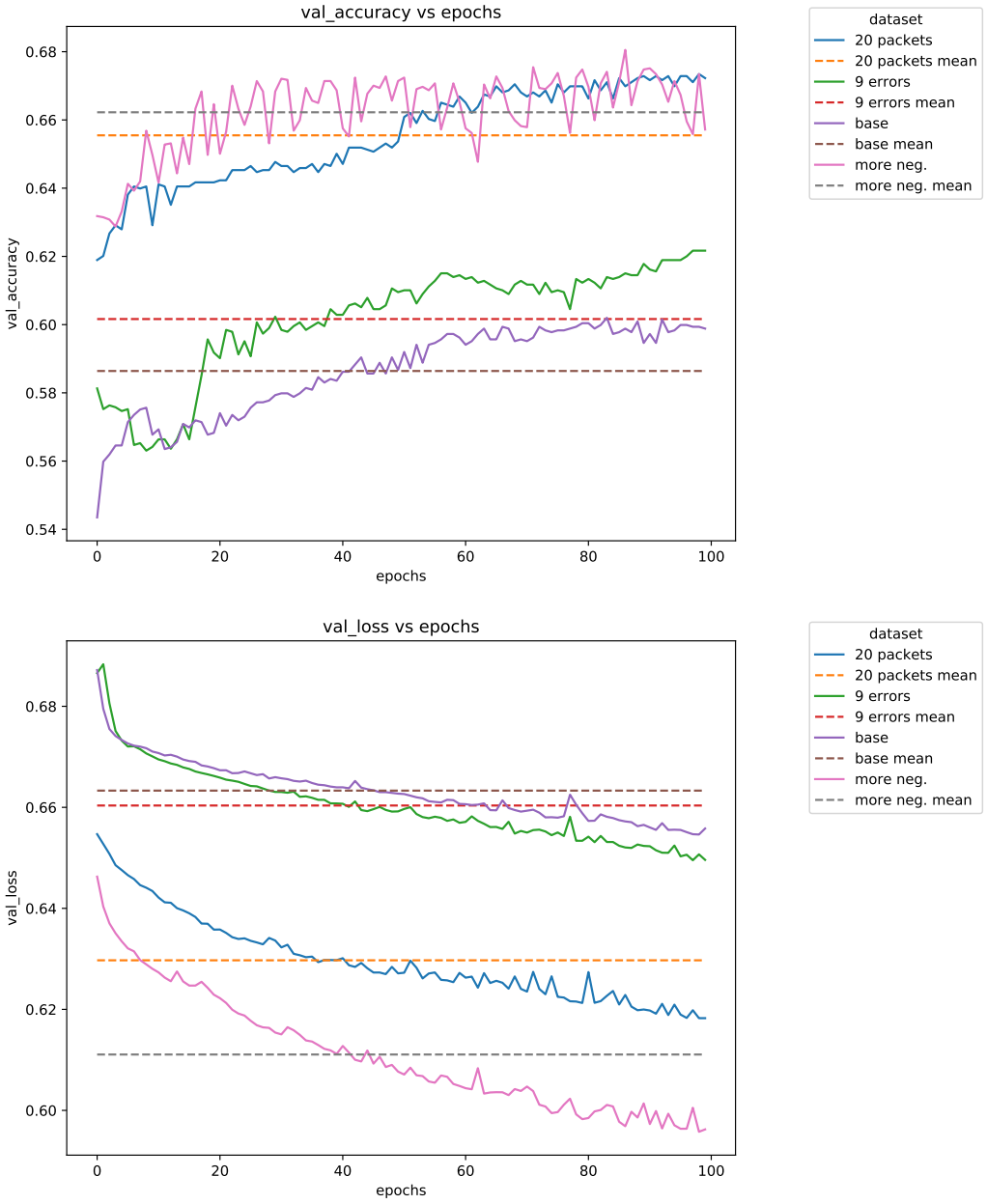


Figure A.9.: Comparison of both validation accuracy and loss between datasets with various negative sample configurations, see 6.1. Our model with 2 LSTM layers, and 64 units per layer was trained over 100 epochs with a batch size of 500.

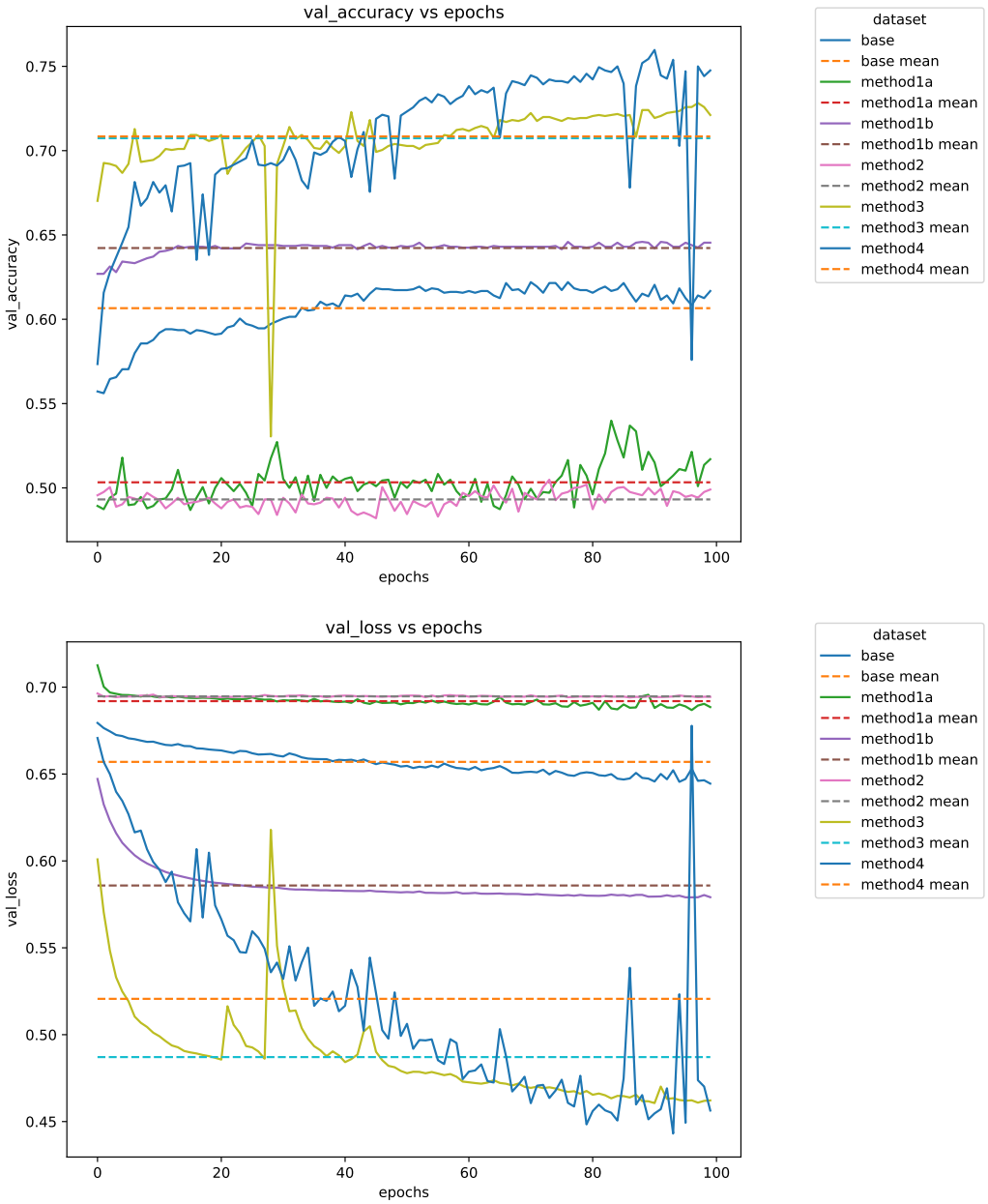


Figure A.10.: Comparison of both validation accuracy and loss between datasets created with only one method of negative sample creation. Our model with 2 LSTM layers, and 96 units per layer was trained over 100 epochs with a batch size of 350.

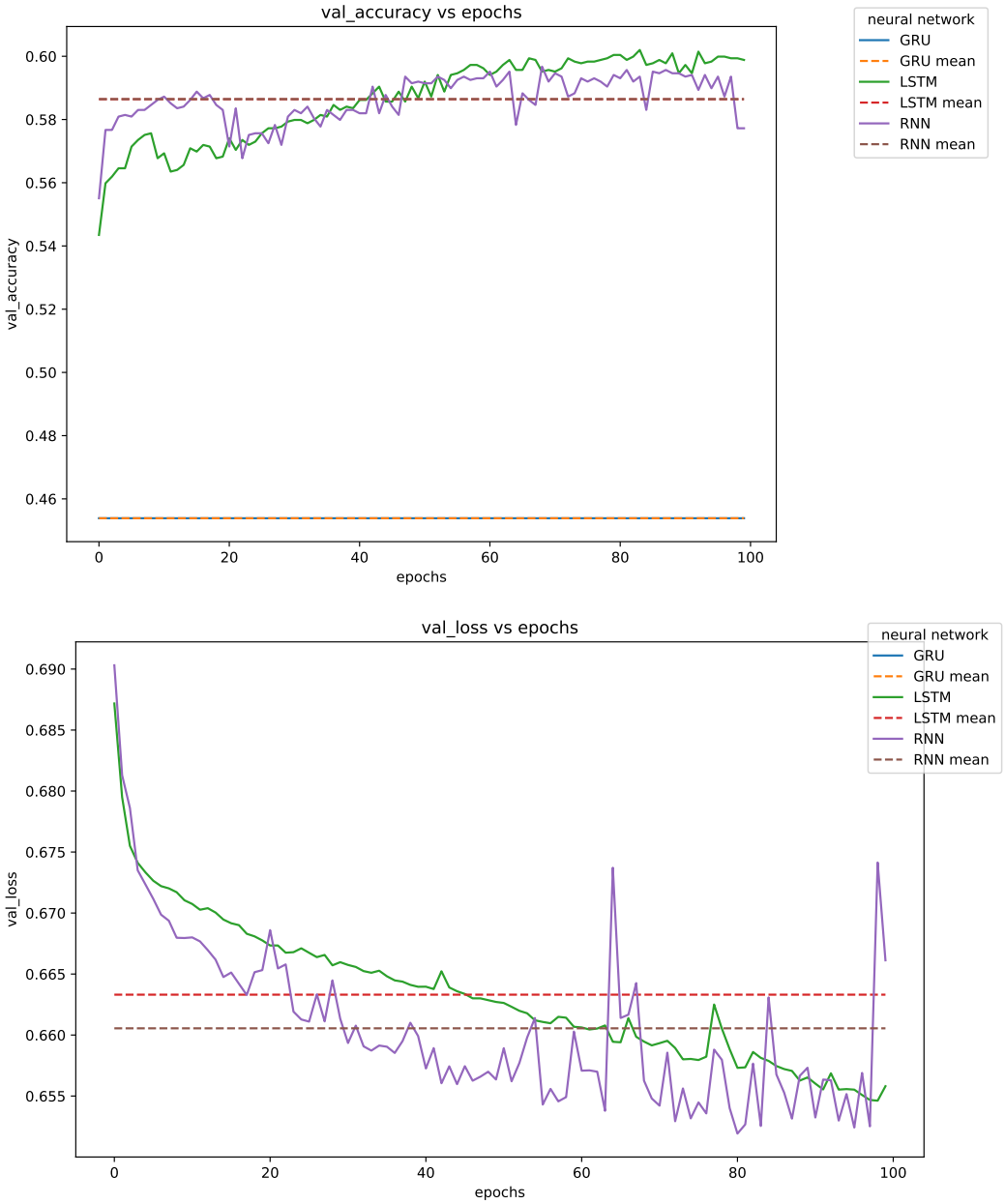


Figure A.11.: Comparison of both validation accuracy and loss between simpleRNN, LSTM and GRU using 2 condRNN layers, 64 units per layer, and a batch size of 500 over 100 epochs with the **base** dataset.

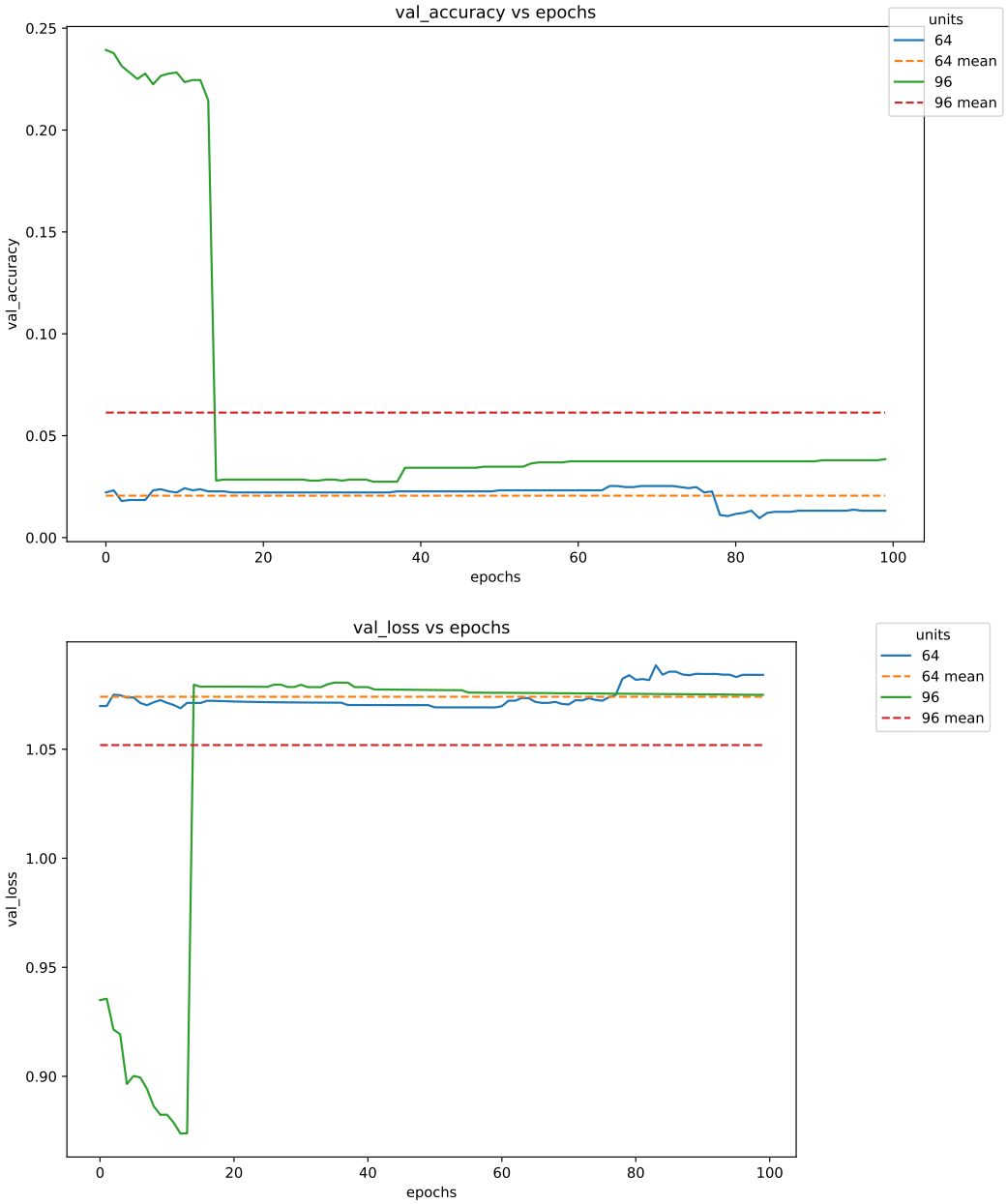


Figure A.12.: Comparison of both validation accuracy and loss between 64 and 96 units per layer. For this our model was trained with 1 layer of GRU units, the batch size of 500 over 100 epochs with the **base** dataset.