Self-supervised Pre-training on LSTM and Transformer models for Network Intrusion Detection

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Embedded Systems

by

Jonas Ferdigg, BSc
Registration Number 01226597

to the Faculty of Electrical Engineering and Information Technology

at the TU Wien

Advisor: Univ. Prof. Dipl.-Ing. Dr.-Ing. Tanja Zseby
Assistance: Univ.Ass. Dott.mag. Maximilian Bachl

Vienna, 1^{st} January, 2001

Erklärung zur Verfassung der Arbeit

Jonas Ferdigg, BSc

Hiermit erklÃďre ich, dass die vorliegende Arbeit gemÃďÃ§ dem Code of Conduct âĂŞ Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulÃďssige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt Ãijbernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im InâĂŞ noch im Ausland in gleicher oder in Ãďhnlicher Form in anderen PrÃijfungsverfahren vorgelegt.

Wien, 1. Jänner 2001

Acknowledgements

Enter your text here.

Kurzfassung

Ihr Text hier.

Abstract

Contents

CHAPTER 1

Introduction

1.1 Motivation

With the progressing digitalization of evermore aspects of society, cyber security will always be a relevant issue as no system will ever be fully secure. Preventing possible cyber attacks by developing more robust systems is one way to mitigate the issue, the other is preventing already existing faults from being exploited as not every vulnerability can be patched easily as it is the case with e.g. DoS and bruteforce attacks. To stop such attacks it is necessary to identify them within the vast flow of ordinary network traffic which gives rise to the need of Intrusion Detection Systems (IDS). State-of-the-art IDSs apply two methods to detect occurring attacks: Signature-based detection and statistical anomaly-based detection. Signature-based detection looks for known patterns or signatures within packets and data streams to identify incoming attacks . Statistical anomaly-based detection focuses on differentiating between normal and abnormal behavior in the system and raises an alert if the latter is identified. The problem with signature-based detection is that unknown attacks are ignored and anomaly-based detection is still not sufficiently accurate and prone to false positives . The rise of Machine Learning (ML) gave opportunity to use the mighty pattern recognition capabilities of Neural Networks (NNs) for intrusion detection. As ML is a rapidly developing field its steady improvement fueled the advance of NN based IDSs which start to show promising results . NNs however are still mostly trained in a supervised fashion, namely by providing labeled examples of cyber attacks for the NN to learn from. This again poses the problem, that only known attacks can be identified, but new attacks that are sufficiently similar to old attacks can also be identified, which is not the case with mere signature-based detection. As with every form of supervised training on NNs, labeled data is harder to come by while unlabeled data is often abundant and certainly so for network traffic data. For this reason, self-supervised training/pretraining is seeing increased use in the realm of ML , as unlabeled data can be used to boost the performance without the need

insert reference to state of the art ids

give examples for IDSs lacking accuracy

give examples for NN based IDSs

give examples of self supervised machine learning for expensive labeled data. One of the most noteworthy examples of the effectiveness of self-supervised pre-training for Neural Networks in the realm of Natural Language Processing (NLP) is Bidirectional Encoder Representations from Transformers (BERT) [DCLT18] developed by Jacob Devlin *et alteri* from Google AI Language. BERT is based on the state-of-the-art Transformer architecture [VSP+17a] and uses a series of proxy tasks like word masking and next sentence prediction to teach the network about syntax and grammar in a self-supervised fashion. The pre-trained network can then be fine-tuned for more specific tasks like question answering or text classification. Analogous, it would be highly beneficial if these or similar pre-training mechanisms could be used to bolster performance of ML based IDSs by improving the classification of network flows, at the most basic level, into cyber attack vs. no cyber attack.

As the technologies mentioned above are fairly recent (Transformers Dec 2017, BERT May 2019) and the design space for solutions in the context of ML for cyber security is substantial, there has not yet been sufficient inquiry into the possibilities of these new methods when applied to the problems posed by Intrusion Detection and cyber attack classification. NN performance also improves with the steadily increasing capabilities of modern Graphics Processing Units (GPU) which makes this a promising concept that can be improved upon by future more powerful hardware.

1.2 Research Questions

In this thesis we inspect if the flow classification performance of Long Short-Term Memorys (LSTMs) and Transformer-Encoder networks can be improved with self-supervised pre-training in a scenario where only little labeled and a lot of labeled data is available. In our context this means a ratio of 1:1000 for labeled to unlabeled data. For performance we are mainly looking at the accuracy of classification, but we are also keeping track of the False Alarm Rate (FAR). The problem to solve is a binary classification problem for which the model is to group flows into *attack* and *no-attack*.

- R1: Can self-supervised pre-training improve the flow classification capabilities of an LSTM model?
- R2: Can self-supervised pre-training improve the flow classification capabilities of a Transformer-Encoder model?
- R3: Which pre-training tasks improve accuracy and which do not?
- R4: If improvement is possible, how can it be explained?

1.3 Approach

To answer these questions we conduct a series of experiments. In these experiments we devised different proxy tasks for the model to solve in a self-supervised fashion.

Solving these proxy tasks serves as pre-training for the network during which it learns the structure of the data and to form abstract representations within its latent space. After the pre-training we train the network with very little labeled training data to teach it how it should classify the flows. These experiments show if pre-training can improve accuracy of the model when compared to only training it with the same amount of labeled data but no pre-training. They also show which pre-training methods are more and which are less beneficial for classification accuracy.

1.4 Contribution

- Implementation of a pre-trainable LSTM model and training suite
- Implementation of a pre-trainable Transformer-Encoder model and training suite
- Inquiry into the benefits of pre-training for sequence-to-sequence models in the context of Network Intrusion Detection Systems (NIDSs)
- Development of new pre-training methods for LSTMs and TransformerEncoder models in the context of NIDSs

Here provide a list of the contributions of your work.

Suggestion (especially for dissertations): provide a table with research questions, methods used to answer each, and major findings and the section in which to find details.

1.5 Structure

After this introduction section we will provide some background information and define terminology used throughout the thesis 2. Subsequently we provide an overview of the current state-of-the-art of NNs for sequence-to-sequence modeling 3, pre-training for such models and ML supported NIDSs in general. Reasoning behind our methodology, and other decisions made, can be found in its dedicated section 4. A detailed description of the conducted experiments can be found in the section *Experiments* 5 with the goal to make them as reproducible as possible. A structured comprehension of experiments conducted is provided in the section *Results* 6. Finally, in the sections *Discussion* 7 and *Conclusion* 8 we discuss successes and failures and draw conclusions from our findings, including pointers for future research.

CHAPTER 2

Background

Artificial Neural Networks (ANNs) have shown great improvements over the last years due to increasing compute power, more sophisticated models and smarter training algorithms . ML and ANNs have long found their way into many commercial applications and many scientific fields have successfully applied this relatively new method of data processing to further their own research. It was only logical that researchers and companies have also started to look into the possible benefits this emerging technology could have for Network Security applications . ANNs are especially suited for IDSs due to their capability to classify data with high accuracy. To harness the power of ML for the purpose of Network Security, we made use of existing methods and models which we will summarize in this section.

cite papers

cite papers

cite papers

2.0.1 Machine Learning

Machine learning describes the study of computer algorithms which are *trained* to optimize a given criterion without the need to specifically program them. In this context *trained* means being provided with data which consists of desired input-output pairs. By processing the input data and comparing it with the desired output data the algorithm adjusts its internal parameters, often called *weights*, through the process of *Backpropagation* 2.0.3 to produce better results in the next iteration. To ensure that the algorithm is learning patterns and structure of problem and not only memorizing input-output data pairs, the training process is often split into two phases: *Training* phase and *validation* phase. In the training phase, the algorithm processes the data and tries to improve its performance. As data is often scarce, the same input-output pairs are used multiple times. This often leads to the problem of *overfitting*, meaning the algorithm performs well only for exactly the data it was trained on but not for similar data. E.g.: The algorithm is tasked with detecting dogs in a picture, outputting TRUE if a dog is present and FALSE if not. After training it only recognizes dog breeds that where present in the training dataset because it relied too much on the individual characteristics of different dog breeds for classification instead of features that every dog has. To detect this behavior early, in the validation phase the algorithm is applied on data it was not trained on. To perform well in this phase the algorithm needs to have found the correct patterns in the training data so it can generalize its prediction to new unseen data.

2.0.2 Artificial Neural Networks

ANNs are a type of Machine Learning algorithm used for classification and prediction. Named after their resemblance to neurons in a brain, ANNs are systems comprised of connected nodes called *artificial neurons*. Analogous to synapses, nodes communicate *via* connections called *edges* by sending "signals" to other nodes. Signals are represented as scalar real numbers. The output signal from a sending node is multiplied by the weight of the edge the signal is "traveling" on. Each node calculates its output signal by applying a non-linear function to the sum of its input signals. Signals travel forward through the network from the first to the last layer, but usually not within layers. The resulting computations can be summarized as a combination of function compositions and matrix multiplications $g(x) := f^L(W^L f^{L-1}(W^{L-1}...f^1(W^1x)...))$ where L is the number of layers, $W^l, l \in \{1, ..., L\}$ the weights connecting nodes of the prior layer to layer l and f^l the activation function of the layer. W^l can also be written as series (w^l_{jk}) where w^l_{jk} is the weight between the k-th node in layer $l-1$ and the j-th node in layer l.

There are various types of ANNs like Recurrent Neural Networks (RNNs) or Convolutional Neural Networks (CNNs) which have many derivations themselves but they all operate on the before stated principal of signals traveling through the network which get transformed at each node by a differentiable non-linear function. The most popular non-linear function at this time is the Rectified Linear Unit (ReLU) function. Without training an ANN performs an input transformation that depends on the initialization values of its weights, often called *parameters*. The network is trained to perform a desired transformation by adjusting its weights/parameters through virtue of *back-propagation*. The network produces output $\hat{y}$ at the last layer after processing input x. A scalar cost/loss value is calculated by a *loss function* $C(\hat{y}, y)$ as a measure of difference between the networks output $\hat{y}$ and the target output y. For classification tasks the loss function is usually cross entropy loss and for regression Squared Error Loss (SEL) is typically used. Back-propagation 2.0.3 computes the gradient of the loss function which is then used by a gradient method like Stochastic Gradient Descent (SGD) to iteratively update all weights in order to minimize (or maximize) $C(\hat{y}, y)$.

reference cross entropy loss

find/create graphic

2.0.3 Back-Propagation

Backpropagation is a type of differentiation algorithm used to calculate the gradient of an arbitrary function with relatively low computational effort. During training an input x_i is processed and information is flowing *forward* through the network producing output $\hat{y}_i = g(x_i)$ 2.0.2, hence this is called a *forward-pass* or *forward-propagation*. The model output culminates into a single scalar cost after applying a cost or loss function $C(\hat{y}_i, y_i)$

which can be interpreted as a measure of distance between the model output $\hat{y}_i$ and the target output y_i. For ML the back-propagation algorithm is used to calculate the gradient of the loss function $\nabla_\theta C(\theta)$ with respect to every weight w^l_{kj} in the model. For this purpose, the weights w are deemed parameters of the forward-propagation and input x_i are deemed constant with the effect of $g(w)$ now only being dependent on w. The chain rule for differentiation is applied multiple times to calculate the partial derivative $\frac{\partial C(g(w),y)}{\partial w^l_{jk}}$ for every weight between every layer in the network which ultimately yields the gradient of $C(g(w), y)$ with respect to w.

2.0.4 Recurrent Neural Networks

The broader concept behind all RNNs is a cyclic connection which enables the RNN to update its state based on past states and current input data [YSHZ19]. Typically, an RNN consists of standard tanh nodes with corresponding weights. There are different kinds of RNNs like continuous-time and discrete-time or finite impulse and infinite impulse RNNs. Here we will only look at discrete-time, finite impulse RNNs as we will only be using those. This type of network, e.g. the Elman network [Elm90], is capable of processing sequences of variable length by compressing the information from the whole sequence into the *hidden layer*. The model produces one output token for each input token, so the transformation is sequence-to-sequence where input and output sequences are of equal length. One input sequence consists of a sequence of real valued vectors $x^{(t)} = x^{(1)}, x^{(2)}, ..., x^{(T)}$ where T is the sequence length. From this input sequence, an output sequence of real valued vectors $\hat{y}^{(t)} = \hat{y}^{(1)}, \hat{y}^{(2)}, ..., \hat{y}^{(T)}$ is produced. To train an RNN pairs of input and target sequences $(x^{(t)}, y^{(t)})$ are provided from which, analogous to the training of ANNs in general2.0.2, a differentiable loss function $C(\hat{y}^{(t)}, y^{(t)})$ can be calculated which can again be minimized by applying back-propagation and SGD. In theory, RNNs can process data sequences of arbitrary length, but the longer the sequence, the deeper the network gets i.e. the longer the gradient paths. This leads to complications when relevant tokens are further apart in the sequence as the RNN is not capable of handling such "long-term dependencies" [YSHZ19]. Long gradient paths in RNNs might also cause the gradient to become either very small or very large, which results in the known *vanishing gradient* or *exploding gradient* problems correspondingly and cause training to either stagnate or diverge. The LSTM improves upon RNNs by making the gradient more stable and allowing long-term dependencies to be considered in the learning process.

give a more formal description of RNNs

find/create graphic

2.0.5 Long Short-Term Memory

Introduced by Hochreiter and Schmidhuber in 1997 [HS97], the LSTM model mitigates the vanishing and exploding gradient problem by replacing the tanh nodes in the hidden layer of a conventional RNN with *memory cells* as seen in 2.1. A memory cell is comprised of *input node*, *input gate*, *internal state*, *forget gate* and *output gate*.

In contrast to an ordinary RNN, an LSTM has two memory states: the hidden state $h^{(t)}$

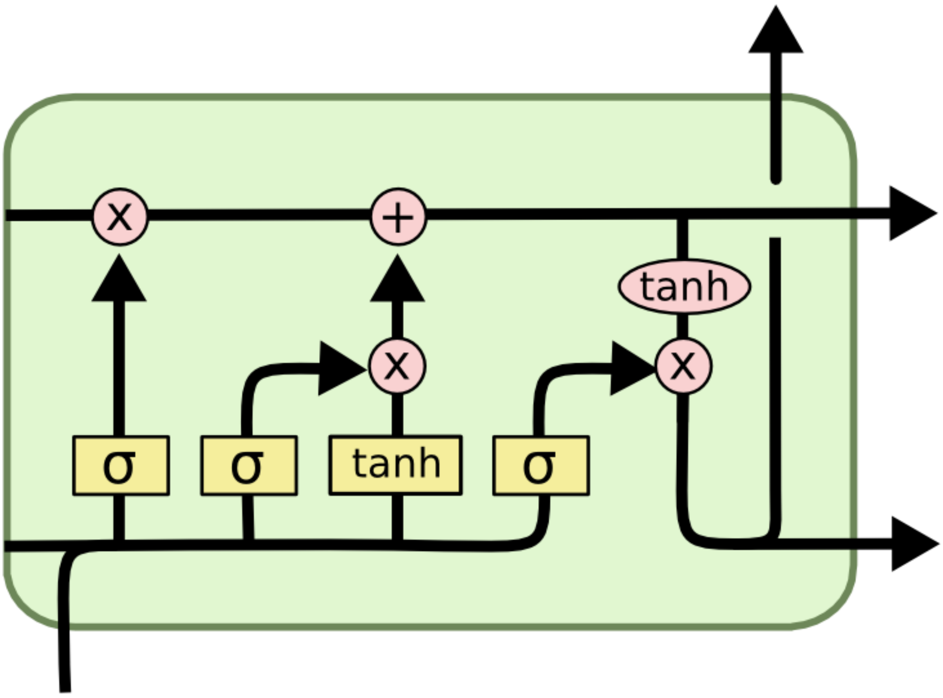

Figure 2.1: One LSTM memory cell [Lip15]

and the *cell state* $C^{(t)}$. Three gates enable the cell to control the flow of information and its effects on the cell state. For this purpose, gates in an LSTM consist of a point-wise multiplication with a vector that holds values between 0 and 1. The three sigma activations seen in 2.1 produce the gate vectors. The input gate $i^{(t)} = \sigma(W^i[h^{(t-1)}, x^{(t)}] + b^i)$ controls whether the memory cell is updated. The forget gate $f^{(t)} = \sigma(W^f[h^{(t-1)}, x^{(t)}] + b^f)$ controls how much of the old state is to be forgotten. The output gate $o^{(t)} = \sigma(W^o[h^{(t-1)}, x^{(t)}] + b^o)$ controls whether the current cell state is made visible. The weight matrices W^i, W^j and W^o decide how information is processed by the cell and are learned parameters. The cell state is updated by addition with the vector $\bar{C} = \tanh(W^C[h^{(t-1)}, x^t] + b^C)$ after multiplication with the input gate vector $i^{(t)}$. The repeated addition of a tanh activation distributes gradients and vanishing/exploding gradients are mitigated.

2.0.6 Adam Optimizer

2.0.7 Attention and Transformers

2017 Vaswani et al. published a paper with the ominous title "Attention is All you Need" [VSP+17b], referring to the already known attention mechanism which is used to model dependencies within a data sequence over longer distances. The authors proposed the Transformer model consisting entirely of self attention mechanisms to model sequences and therefore diverge from the recurrent architectures of RNNs and LSTMs. Attention is a mechanism to capture contextual relations between tokens in a sequence, e.g. words in a sentence. For every token in the input sequence, an attention vector is generated which represents how relevant other tokens in the input sequence are to the token in question. While attention can be implemented in different ways, the authors chose the scaled dot-product attention defined as

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

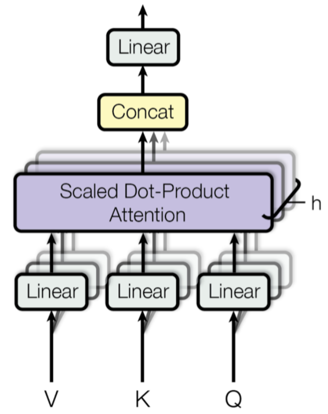

Figure 2.2: Self attention layer of Transformer by [VSP+17b]

"An attention function can be described as mapping a query and a set of key-value pairs to an output" [VSP+17b]. Q, K and V are matrices composed of query, key and value vectors for every token with respect to every other token in the sequence. Vaswani et al. proposed the use of Multi-Head Attention mechanism suggesting the use of multiple independent attention heads which are generated by linear projection of the original Q, K and V matrices by different learned matrices W_i^Q, W_i^K and W_i^V for $i = 1, ..., h$ where h is the number of desired attention heads. The attention vectors of the different attention heads are again concatenated and projected by matrix W^Z again resulting in a single combined attention vector instead of h vectors. This results in the formulation

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V), i = 1, ..., h \tag{2.2}$$

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O \tag{2.3}$$

depicted in figure 2.2. The Multi-Head Attention block from 2.2 is used in the Transformer encoder block 2.3 together with a fully-connected feed forward network. After each sub-layer (Multi-Head Attention, Feed Forward) layer normalization is applied and a residual connection originating from the input to the sub-layer is added as can again be seen in figure 2.3. The output of each sub-layer is hence defined as $LayerNorm(x + Sublayer(x))$ where $Sublayer$ is either a Feed Forward or a Multi-Head Attention function. While there is more to the Transformer model, for our experiments we are only using the parts described here.

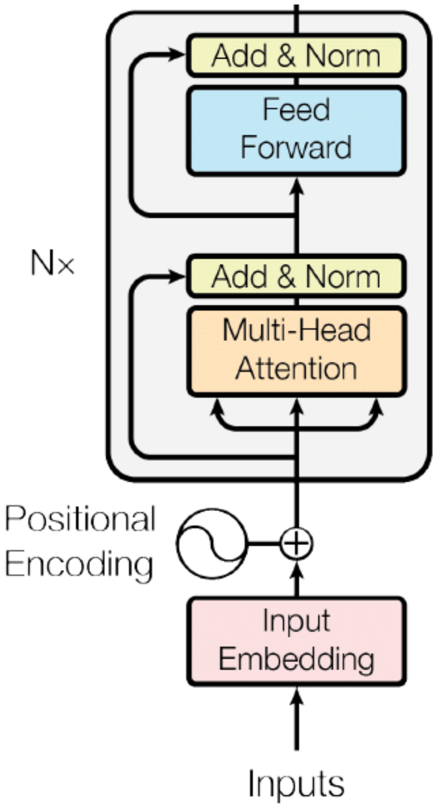

Figure 2.3: Transformer Encoder Model as proposed by [VSP+17b]

2.0.8 Self-supervised Learning

Supervised learning is most effective when teaching a NN what its supposed to do but it is limited by the amount of labeled data that is available. For many use cases, not enough is available and the cost of creating new labeled data is too high to be feasible. In those cases, self-supervised learning or self-supervised pre-training might be an efficient addition.

For supervised learning the target data provides the supervision. For Self-supervised learning the data itself provides the supervision meaning the loss $C(\hat{x}, x)$ is calculate between the reconstructed input $\hat{x}$ and the actual input x. In general this means that some part of an input tensor or an input series is withheld and the model is tasked with reconstructing the unknown information. So instead of being trained for the task we want it to perform, it is first trained on a *proxy task* which serves no purpose on its own but forces the model to learn a semantic representation of the data which will help solve the actual task.

2.0.9 Auto Encoder

The auto encoder is a popular tool for self-supervised learning. The model is composed of an *encoder* and a *decoder* stage as can be seen in figure 2.4. The encoder compresses the input data, artificially causing loss of information. In the next step the decoder tries to reconstruct the compressed data as accurately as possible. The loss $C(\hat{x}, x)$ is then calculated as the difference between the original input and the reconstructed one. The aim of this seemingly nonsensical task is to force the model to form an abstract, more compact representation of the input data in its restricted latent space. To compress data with minimal loss of relevant information the network has to find patterns in the input and ideally learns some semantic of the data.

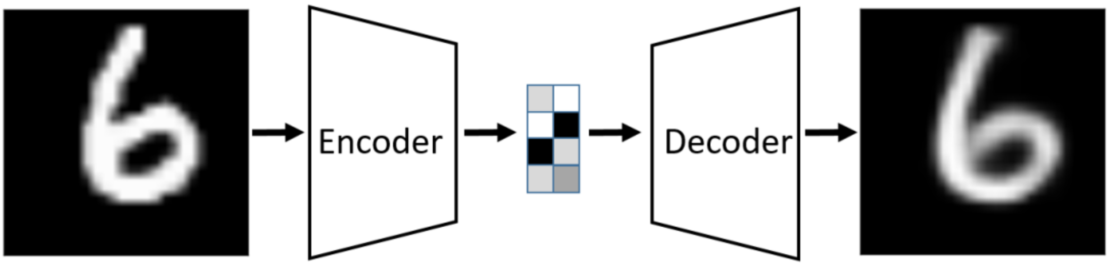

Figure 2.4: Visualization of an auto encoder. The input is encoded and subsequently decoded yielding and approximate reconstruction of the image [BKG20]

After the self-supervised training of the auto encoder is finished, the decoder stage is removed and subsequently the output of the encoder is used as input tensor for a classification or prediction model.

2.0.10 Pre-Training and Fine-Tuning

Pre-training with subsequent *fine-tuning* describes a methodology of training a NN in two separate phases. E.g. Googls BERT for NLP is pre-trained in a self-supervised fashion with vast amounts of text (3.3 million words) [DCLT18]. Depending on the task of the model, i.e. translation, question answering, text generation, the models parameters are then fine-tuned with labeled data fit the given task.

2.0.11 Terminology

In addition: Abbreviations and mathematical notation should be put in a list in the beginning of the thesis

CHAPTER 3

State of the art

As the topic of this thesis is rather specific, comparable research is hard to find. Overall, the thesis works on the two subjects of unsupervised pre-training for NNs and for ML supported NIDS. Here we are looking at state-of-the-art research of both aspects individually.

3.1 Self-supervised Pre-training for LSTMs and Transformer Networks

When it comes to machine learning, rapid progress has been made over the past years. Frameworks such as PyTorch [ea16] and Tensorflow [Tea15] have made the technology accessible to people without a background in computer science. More than 11 thousand papers in the category "Computer Science - Artificial Intelligence (cs.AI)" have been published on arXiv.org [Gin91] within only the last year. With steadily increasing processing capabilities, vast amounts of data can be used to train ever growing NNs within an acceptable timeframe. E.g. the largest variant of Google's BERT algorithm has 340 million parameters and was trained on a dataset of 3.3 Billion words [DCLT18].

3.1.1 BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Google's BERT [DCLT18] by Jacob Devlin et al. effectively uses a Deep Bidirectional Transformer model, often referred to as Transformer Encoder, for various NLP tasks, both on sentence and word level, like question answering, natural language inference, sentiment analysis, paraphrasing and others. At the time it was published, it produced the highest recorded GLUE [WSM$^+$18] score of 80.5% advancing it by 7.7% over the former top scorer. It uses the WordPiece [WSC$^+$16] embedding resulting in a 30,000 token vocabulary. It was pre-trained in a fully unsupervised fashion on all sentences in the English Wikipedia (2,5 Billion words) and the BooksCorpus [ZKZ+15] containing 800 Million word. The pre-training consisted of two proxy tasks: Next Sentence Prediction (NSP) and Masked LM (MLM). For NSP, two sections of text, A and B, separated by a [SEP] token are fed into the model at the same time. 50% of the time, B is the next section that follows A in the original text. 50% of the time it is a random sentence from the corpus. The model is tasked with predicting, if sentence B follows sentence A. For MLM, 15% of the input tokens are hidden from the model by replacing with a [MASK] tokens. The model is tasked with reconstructing the masked tokens. Both of those pre-training tasks are performed at the same time. The pre-trained model is then fine-tuned to perform a specific down-stream task.

This two stage approach, pre-training and fine-tuning, produces a reusable pre-trained model which can then be fine-tuned relatively swiftly (Jacob Devlin et al. state that it takes at most an hour of fine-tuning on a GPU to replicate all results in the paper) to solve various NLP tasks. For this thesis, we use the same approach to pre-train our models in an unsupervised fashion and then fine-tune them with a small amount of labeled data to teach them the down-stream task of classifying network flows. We also use the pre-training task of masking parts of the input data for the model to reconstruct for both our LSTM and Transformer networks. The NSP task is not feasible in our situation, as network flows don't have an order other than the time of occurrence, and therefore flows don't have a semantically identifiable successor or predecessor.

3.1.2 Unsupervised Learning of Video Representations using LSTMs

The use of unsupervised learning is not limited to Transformer networks. As early as 2016, before the rise of Transformers, Nitish Srivastava et al. showed in their paper "Unsupervised Learning of Video Representations using LSTMs" [SMS15] that unsupervised learning on LSTMs can have a positive impact on subsequent classification tasks. The authors use video data to train their models in frame prediction and auto encoding as the proxy tasks with the goal of improving accuracy in human action recognition, based on evaluation with the UCF-101 and HMDB-51 datasets. They experimented with two types of video representations: patches of image pixels and high-level representations ("percepts") of video frames extracted by a convolutional net. They used 13,320 videos with an average length of 6.2 seconds belonging to 101 different action categories.

The auto-encoding property of the model is achieved by concatenating two LSTMs, with one performing the function of encoder and one of decoder. The goal is to produce a sequence2sequence model capable of reconstructing the input sequence after being forced to compress the input data. The input sequence is first processed by the encoder LSTM to produce an output of constant length (in their case, the hidden size of the encoder LSTM). The resulting vector is then fed into the decoder which is tasked with reconstructing the input sequence in reverse order. Here, the decoder can be configured to either be *conditioned* or *unconditioned.* A conditioned decoder uses the output of the last LSTM stage as input for the next stage. An unconditioned decoder uses the corresponding input token (ground truth) as input for the next stage. The latter practice is also called *teacher forcing*.

The second unsupervised task to train the LSTM consists of predicting multiple future video frames. For this, again two consecutive LSTM networks are used: an encoder and a predictor LSTM. The first network is fed the frame representation of part of a short video and again produces a fixed sized output vector to be used by the predictor LSTM. The second LSTM is then tasked with producing the remaining frames. Same as with the auto-encoder the predictor LSTM can either be conditioned or unconditioned.

The authors then proposed a composite model as can be seen in figure 3.1 where both proxy tasks, reconstructing the input and predicting the future, are combined to produce a single model.

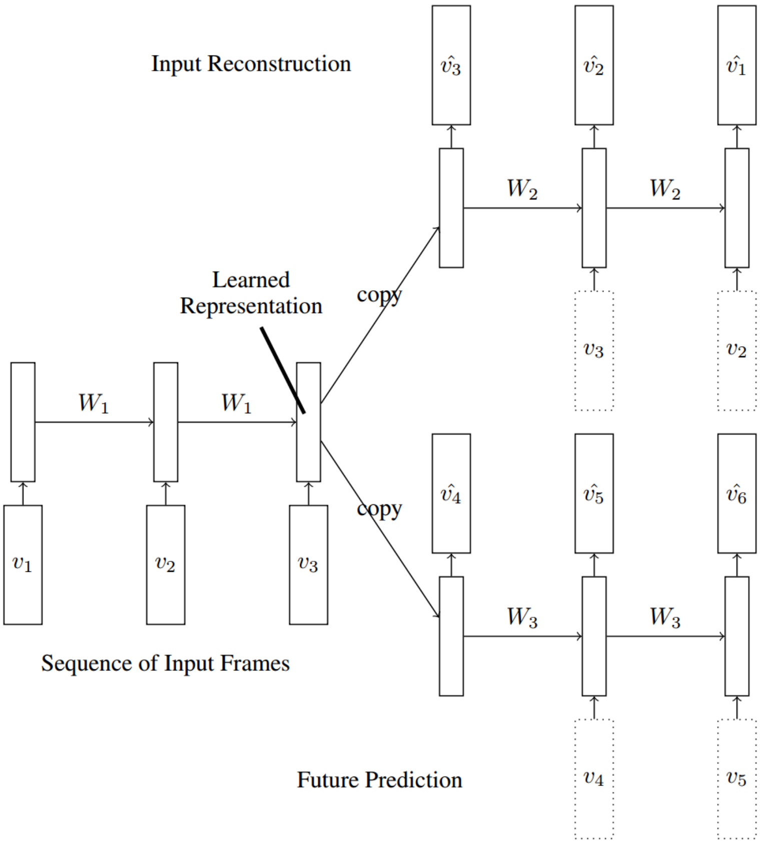

Figure 3.1: Composite model for input reconstruction and future prediction [SMS15]

3.2 Machine Learning for Network Intrusion Detection

Here provide an overview of the related state of art. Look for papers that are closest to the research you are doing Suggestion: make a table with the related papers and compare them wrt to different criteria, for instance

- Findings: What do they claim (main findings)
- Data: What data set they are using
- Methods: Which methods did they use?
- Reproducibility: Is it possible to reproduce the results? (e.g., is the data available? are all parameter settings provided? Is source code provided?)
- Relevance (How relevant is it for your work)

In the last paragraph explain how your work differs from the existing works.

CHAPTER 4

Methodology

- explain why these experiments are used
- explain metric for comparing results (accuracy, false alarm rate)
- short summary of code?

Here describe the methodology you use and why you decided to use it. e.g., theoretical considerations, simualtons, experiments, measurements, testbeds, emulations, etc. What concepts are used.

Also explain which metrics you use to measure success or failure (e.g., detection performance with accuracy, recall, precision, f1 score, RocAUC, etc.)

Provide a figure (see example figure 4.1) to describe the processing steps

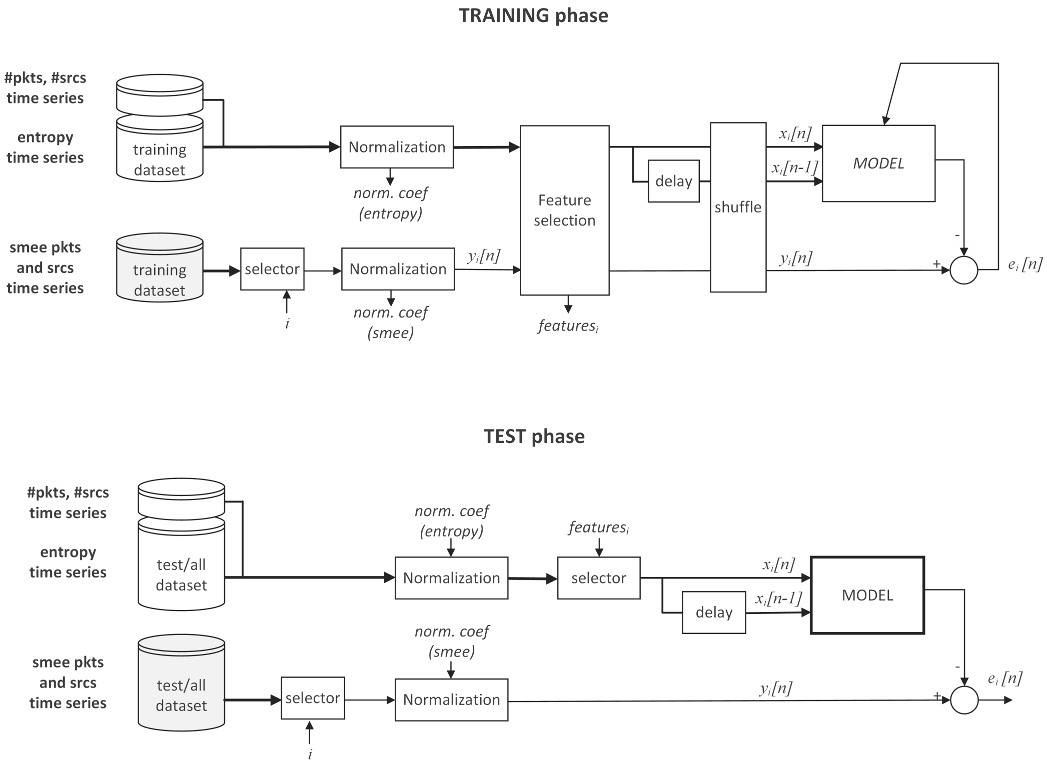

Figure 4.1: Describe in the caption exactly what can be seen in the figure

CHAPTER 5

Experiments

To inspect the potential benefits of self-supervised pre-training for ML-based intrusion detection we chose to take a look at LSTM and Transformer networks as they are suited to process sequences of variable length and have shown promising results in the past . Network traffic data can be looked at from a multitude of perspectives ranging from aggregate statistical data over different time-frames [MDES18] to looking at feature representations of single packets which can be viewed in the context of *flows*. Flows are loosely defined as sequences of packets that share a certain property [HBFZ19]. In our case we define flows as packets that share source and destination IP address, source and destination port, and the network protocol used. This creates the quintuple *<srcIP, dstIP, srcPort, dstPort, protocol>* as the key over which individual packets are aggregated to flows. We used the data pre-processing from [HBFZ19] as it fit the requirements for our experiments and was easily modifiable. The underlying data from which flow data is extracted are the *CIC-IDS-2017* [SLG18] and *UNSW-NB15* [MS15] NIDS datasets. After the data pre-processing from [HBFZ19] each packet is represented by source port, destination port, packet length, Interarrival Time (IAT), packet direction and all TCP flags (SYN, FIN, RST, PSH, ACK, URG, ECE, CWR, NS) resulting in 15 input features to be used in training the NNs.

give examples

The task of the NNs is to classify each flow into either *benign* or *attack* which results in a binary classification problem. Ordinary network traffic that should be ignored by the IDS is labeled as *benign* and flows that constitute or are part of a cyber-attack are labeled as *attack*. As there are only two possible labels, Binary Cross Entropy (BCE) can be used as loss function to determine the distance between the predicted label by the NNs and the actual label . For updating weights we use the *Adam* optimizer [KB14] which is an extension to the commonly used SGD method. Similar to *AdaGrad* [Rud16] and *RMSProp* [Rud16] it maintains separate learning rates for each individual weight instead of using the same learning rate for every weight like in classic SGD. Compared to other optimizers *Adam* was shown to be more effective in improving training efficiency

give more detailed explaination of BCE Loss

Model	Dataset	Batchsize	Subset	Training %	Training Epochs
LSTM	CIC-IDS-2017	128	-	90	20
LSTM	CIC-IDS-2017	128	-	10	100
LSTM	CIC-IDS-2017	128	-	1	500
LSTM	CIC-IDS-2017	128	CIC_10	-	500
LSTM	UNSW-NB15	128	-	90	20
LSTM	UNSW-NB15	128	-	10	100
LSTM	UNSW-NB15	128	-	1	500
LSTM	UNSW-NB15	128	CIC_10	-	500
Transformer	CIC-IDS-2017	512	-	90	20
Transformer	CIC-IDS-2017	512	-	10	100
Transformer	CIC-IDS-2017	512	-	1	500
Transformer	CIC-IDS-2017	512	UNSW_10	-	500
Transformer	UNSW-NB15	512	-	90	20
Transformer	UNSW-NB15	512	-	10	100
Transformer	UNSW-NB15	512	-	1	500
Transformer	UNSW-NB15	512	UNSW_10	-	500

Table 5.1: List of baseline training runs used for comparison later in the thesis.

[KB14] and is appropriate for noisy or sparse gradients which can occur when working with RNNs in general.

As a premise for our research we trained the LSTM and the Transformer network in a solely supervised fashion to get a baseline later results can be compared to. Supervised training was performed for 20, 100 and 500 epochs each for 90%, 10% and 1% respectively of available data on both data-sets and a constant 10% of data for validation which has not been used for training. A full overview of all experiments to establish a comparison baseline can be seen in 5.1. We specifically wanted to know how the networks would perform in a scenario where very little labeled training data was available as this would best describe a scenario where large amounts of unlabeled data are available for self-supervised pre-training and only a small amount of labeled data for fine tuning. To pre-train a NN the network is given a task that is not necessarily connected to the final purpose of the network, often referred to as a *proxy task*. By solving the proxy task the network attempts to find structure in the data and should learn to form a more abstract representation of the data within its latent space. E.g. with BERT pre-training is performed by masking a certain percentage of input tokens and having the NN predict the missing words and additionally letting the network guess whether one sentences precedes another in a text. We defined our own proxy tasks for pre-training the networks as described in the following sections. Pre-training is performed with 80% of available data, supervised fine-tuning with 1% and validation with 10% of data.

Model	Dataset	B.s.	Subset	Tr. %	Tr. Eps.	Pretr. %	Pretr. Eps.
LSTM	CIC-IDS-2017	128	-	10	20	80	10
LSTM	CIC-IDS-2017	128	-	1	100	80	10
LSTM	CIC-IDS-2017	128	CIC_10	-	500	80	10
LSTM	UNSW-NB15	128	-	10	20	80	10
LSTM	UNSW-NB15	128	-	1	100	80	10
LSTM	UNSW-NB15	128	UNSW_10	-	500	80	10

Table 5.2: Training and pretraining configurations for LSTM model trainings with different proxy tasks.

5.1 Self-supervised Pre-training for Long Short-Term Memory Networks

For our LSTM network we chose a three layer LSTM with a *hidden size* and *cell size* of 512. While a larger network might be more effective, this configuration proved to be swiftly trainable while also producing results close to the optimum . Since we are only interested in comparisons between different training methods applied to the same model, it is not necessary to increase model size to achieve optimal results as this would unnecessarily increase the training time needed until the model converges. For training the LSTM model, each flow is considered one sample and each packet is one token. The tokens are processed by the model in chronological order, meaning packets with an earlier timestamp will be processed first. The timestamp however is not part of the feature representation but is considered for data pre-processing to order packets within flows. For pre-training the LSTM we devised five different proxy tasks for the model to solve in a self-supervised fashion: Predicting the next packet in the flow, predicting masked features of randomly chosen packets and predicting randomly masked packets, the identity function and an AutoEncoder. The Mean Absolute Error (MAE) is used to determine the divergence between prediction and target data. Translating to PyTorch this means we used *L1Loss* with *mean* reduction as the loss function for pre-training. We tuned the hyper-parameters of training for both supervised and self-supervised training to an initial *learning rate* of 10^{-3} and a *batch size* of 128. Over the training process, the learning rate will be adjusted by Adam so the model is robust to changes on the initial learning rate. For every proxy task, the model has been trained with the different parameters in table 5.2 to establish comparable results.

provide numbers

- different parameterization of LSTM
- two consecutive 3-layered LSTMs
- orthogonal initialization
- CrossEntropy Loss instead of BCE

5.1.1 Identity Function

The simplest form of a proxy-task for pre-training is having the model learn the identity function. In practice that means that input sequence $x^{(t)}$ and target sequence $y^{(t)}$ are the same $x^{(t)} = y^{(t)} = x^{(1)}, x^{(2)}, ..., x^{(n)}$ where n is the sequence length. The model learns to convey the information through the network at each time step. For this task, the model does not need to derive any meaningful hidden representation of the data, but as our experiments show it still moves the weights of the model into a favorable direction when compared to a 0-initialization or an orthogonal initialization.

5.1.2 Predict Packet

For this proxy task, the model has to predict the next packet in the flow. We started by predicting only the last packet in each flow but then moved to predicting all packets in a flow except the first. This means having a *sequence-to-sequence* model where the inputs are all tokens in one flow with length n except the last, because it has no successor: $x^{(t)} = (x^{(1)}, x^{(2)}, ..., x^{(n-1)})$. The target data are all tokens in the same flow except the first, because it has no predecessor: $y^{(t)} = (x^{(2)}, x^{(3)}, ..., x^{(n)})$. LSTMs process data in sequential order so at each time step, the model only has information of packets in the past and is to predict what the next packet in the flow will be. This results in two comparable tensors $y^{(t)}$ and the model output sequence $\hat{y}^{(t)} = (\hat{y}^{(1)}, \hat{y}^{(2)}, ..., \hat{y}^{(n-1)})$ of equal length $n-1$ between which a differentiable loss $C(y^{(t)}, \hat{y}^{(t)})$ can be calculated. This way, a lot of information is conveyed to the network when compared to only predicting the last packet in a flow. At first glance, this looks similar to the identity function in 5.1.1. The key difference is however, that the token which is to be predicted is not yet available as an input token to the model, meaning it has to derive the features by other means than conveying the requested input token to the output. The loss is calculated as the MAE (*L1Loss* with *mean* reduction) between the predicted logits and the target data sequences.

5.1.3 Mask Features

For this pre-training task, the model is to predict masked features of some packets in the sequence. We have tried multiple masking values but -1 produces the best results out of the values we tried . This proxy task in particular can be parameterized in different ways. E.g. the number of features and which features to mask, if always the same features are masked or if the selection is random for each packet or for each flow, if every packet in the sequence has some masked features or if there is only a chance that a packet is selected for masking. Those are only some examples of how this task can be set up in different ways. To be completely exhaustive was not possible, so we compiled a selection of some of the variations as an overview of the parameter space. For pre-training the model is provided masked data as input sequence and the unmasked data is the target. The loss is calculated as the MAE (*L1Loss* with *mean* reduction) between the predicted *logits* and the target data sequences.

give a comparison of values

enumerate all parameter combinations used 22

5.1.4 Mask Packets

Similar to the pre-training in BERT, all features of random packets in the sequence are masked with a value of -1 and the model is to predict the masked tokens. Again, MAE is used as the loss function. Unlike to BERT, we don't only look at the masked tokens when calculating the loss but compare every feature of every packet, also the non-masked ones, which adds an auto-encoding property to the pre-training. We found this to have more beneficial effect on the results than only looking at the masked packets. The most important parameter here is the ratio of how many packets per sequence are to be masked compared to its sequence length. To work with an absolute number of masked packets is not feasible as sequence length varies from 1 to a set max sequence length which in our case was 100. If an absolute number was used to determine how many packets should be masked some sequences would be completely masked out which would not be beneficial for training.

5.1.5 Auto-Encoder

As explained in section **??**, for the Auto-Encoder the model is tasked with compressing and decompressing the data as lossless as possible. With an LSTM model, this means having two consecutive LSTM models where the first is to encode the sequence and the second is to decode the sequence. For the encoder LSTM this means compressing the whole input sequence $x_e^{(t)} = (x_e^{(1)}, x_e^{(2)}, ..., x_e^{(n)})$ into the hidden state (and cell state) of the last stage $h_e^{(n)}$ $(C_e^{(n)})$ where n is the length of the input sequence. The decoder LSTM is then initialized with the hidden and cell state of the last stage from the encoder LSTM $h_d^{(1)} = h_e^{(n)}, C_d^{(1)} = C_e^{(n)}$ trying to reconstruct the input sequence. After every stage of the decoder, the output of the current stage $\hat{y}^{(t)}$ is then fed into the model as input token $x_d^{(t+1)} = \hat{y}^{(t)}$ for the next stage to calculate the next time step. The first input token for the decoder is a zero vector which functions as a start-of-sequence token $x^{(1)} = 0$. This way, the encoder is forced to store as much information about the sequence as possible in the hidden state and as the size of the hidden state is constrained, it has to find an abstract representation of the sequence. For supervised fine-tuning and validation, only the encoder part of the model is used.

insert graphic

5.2 Self-supervised Pre-training for Transformer Networks

Following the example of BERT we only used the encoder part of the transformer since the decoder does not provide any benefit for classification problems. We tuned the model parameters to be 10 Transformer layers, each layer consisting of a 3-headed Multi-Head Attention block and a feed-forward network with a forward expansion of 20 times the input size, i.e. the number of features per packet. Since we did not observe any over-fitting during training, we set the drop-out rate to zero (except for training with the Auto-Encoder 5.2.2). Like with the LSTM we devised a series of proxy tasks for

Model	Dataset	B.s.	Subset	Tr. %	Tr. Eps.	Pretr. %	Pretr. Eps.
Transformer	CIC-IDS-2017	512	-	10	20	80	10
Transformer	CIC-IDS-2017	512	-	1	100	80	10
Transformer	CIC-IDS-2017	512	CIC_10	-	500	80	10
Transformer	UNSW-NB15	512	-	10	20	80	10
Transformer	UNSW-NB15	512	-	1	100	80	10
Transformer	UNSW-NB15	512	UNSW_10	-	500	80	10

Table 5.3: Training and pretraining configurations for Transformer model trainings with different proxy tasks.

pre-training the model in self-supervised fashion. Since the information flow is different in Transformers than it is in LSTMs, the pre-training task *Predict Packets* 5.1 we used for the LSTM is no longer feasible. While the LSTM at each stage has only access to all the tokens it processed up to this point, the Transformer has access to all input tokens at each stage of the execution which is one of the benefits of self-attention [VSP+17a]. Contrary to our expectations, supervised training (with 90% auf the dataset) on the Transformer takes longer than on the LSTM to achieve the observed optimal accuracy of 99,65%. In other words, when training the LSTM and the Transformer network with the same amount of data for the same amount of time, the LSTM produces better results. In the following sections we describe the pre-training methods we used to pre-train the Transformer network.

write

- two consecutive TransformerEncoders, one for pre-training, one for fine-tuning
- Classification (CLS) Token
- Resetting last Layers 1,...,5 of Transformer after pre-training
- Use decoder also
- different dropout rates
- different number of attention heads

5.2.1 Mask Features

Analogous to the *Mask Features* proxy task for the LSTM, we used the same method for pre-training the Transformer.

5.2.2 Autoencoder

Autoencoder are an established concept when it comes to self-supervised learning **??**. With this method input and target data are the same and the network is tasked with reconstructing the input data at the output. To prevent the network from simply "transporting" the input tokens through the network without having to learn anything, a form of regularization is introduced to force the network into learning an abstract representation of the data [BKG21]. In our case, we used the dropout rate to introduce artificial noise into the input data.

5.2.3 Mask Packet

For this proxy task, random packets in the flow are masked with a value of -1 and the model is to predict only the masked packets. Since a packet in a flow can be seen as a word in a sentence, and the feature representation of a packet is similar to an embedded word vector, this pre-training task is analogous to the method used in BERT [DCLT18].

CHAPTER 6

Results

- maximum accuracy with 0-90-10 pre-sup-val training
- comparison between pretraining accuracy with different proxy tasks for 10-80-10 pre-sup-val training
- comparison between pretraining accuracy with different proxy tasks for 1-89-10 pre-sup-val training
- comparison between pretraining accuracy with different proxy tasks for subset 10_flows subset pre-sup-val training
- comparison of performance improvements for different amounts of supervised training
- comparison of performance improvements for different compositions of pretraining data
- comparison between multiple datasets
- comparison to orthogonal initialization

6.1 Long Short-Term Memory Network

- provide data for maximum results including class stats for both datasets to establish a feel for the maximally possible accuracy with supervised training and 90% of data
- show results for different amounts of supervised data and discuss results between different proxy tasks by showing loss progression and validation accuracy over training time and comparing class stats

- highlight the improvement in accuracy when comparing to supervised training only
- look closely at differences in loss progression and validation accuracy over time between different proxy tasks

6.2 Transformer Network

6.3 Explainability

- close look at differences in performance for different attack classes
- partial dependency plots
- neuron activation

CHAPTER 7

Discussion

Discuss any open issues and give a critical reflection of your work. E.g., what could be problems to deploy your method or do you have an idea how your findings could be generalized or what could be a hindrance for generalization?

Also discuss strange things you observed or results you could not completely explain.

CHAPTER 8

Conclusion

Conclude your work. Stress again what was the contribution. Provide an outlook what could be further improvements and what could future research do to continue your work.

List of Figures

List of Tables

List of Algorithms

Bibliography

[BKG20] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders, 03 2020.

[BKG21] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders, 2021.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[ea16] Adam Paszke et al. Pytorch, 2016.

[Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[Gin91] Paul Ginsparg. arxiv, 1991.

[HBFZ19] Alexander Hartl, Maximilian Bachl, Joachim Fabini, and Tanja Zseby. Explainability and adversarial robustness for rnns. *CoRR*, abs/1912.09855, 2019.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[Lip15] Zachary Chase Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[MDES18] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. *CoRR*, abs/1802.09089, 2018.

[MS15] Nour Moustafa and Jill Slay. Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set). 11 2015.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[SLG18] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *Proceedings of the 4th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pages 108–116. INSTICC, SciTePress, 2018.

[SMS15] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. Unsupervised learning of video representations using lstms. *CoRR*, abs/1502.04681, 2015.

[Tea15] Google Brain Team. Tensorflow, 2015.

[VSP+17a] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[VSP+17b] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[WSC+16] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[WSM+18] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *CoRR*, abs/1804.07461, 2018.

[YSHZ19] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures. *Neural Computation*, 31(7):1235–1270, 07 2019.

[ZKZ+15] Yukun Zhu, Ryan Kiros, Richard S. Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *CoRR*, abs/1506.06724, 2015.