TECHNISCHE UNIVERSITÄT WIEN
Vienna | Austria

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, 1st 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 (IDSs). 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 **??**, 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

2.1 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* 2.3 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.

2.2 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 $g(x) := f^L(W^L f^{L-1}(W^{L-1}...f^1(W^1 x)...))$ 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) or L1 loss is typically used. Back-propagation 2.3 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

2.3 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

$\hat{y}_i = g(x_i)$ 2.2, 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.

2.4 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 general2.2, 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

2.5 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* $\tilde{C}$, *hidden state* h, *cell state* C, *input gate* i, *forget gate* f and *output gate* o.

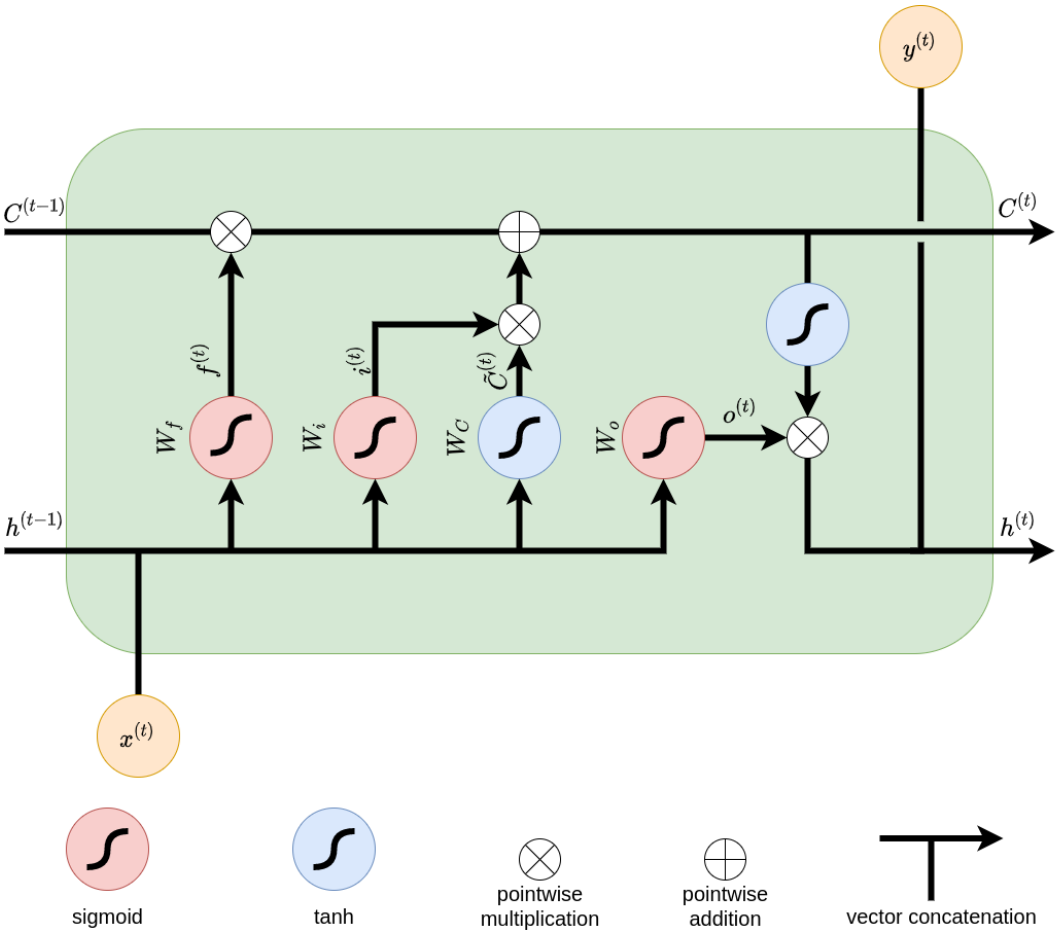

Figure 2.1: One LSTM memory cell [Lip15]

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

2.6 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

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

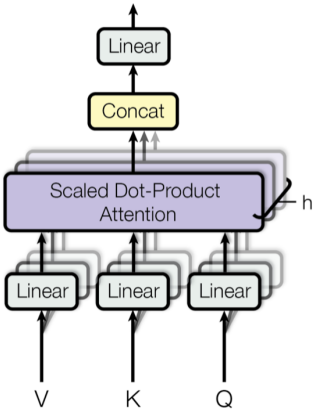

Figure 2.2: Self attention layer of Transformer by [VSP+17b]

"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.

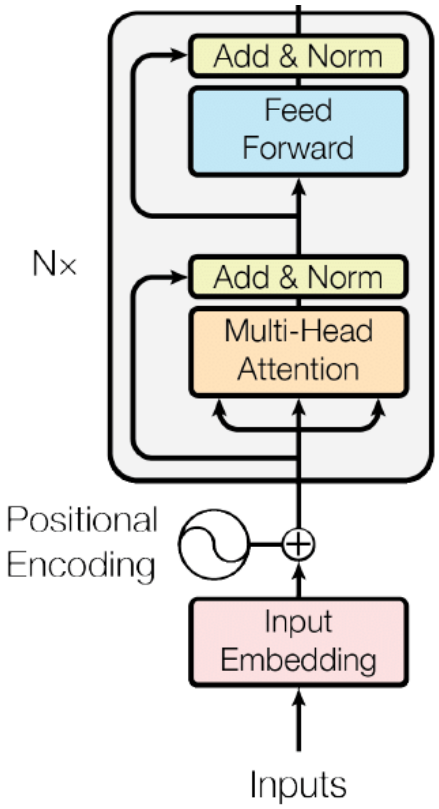

Figure 2.3: Transformer Encoder Model as proposed by [VSP+17b]

2.7 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.

2.8 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.

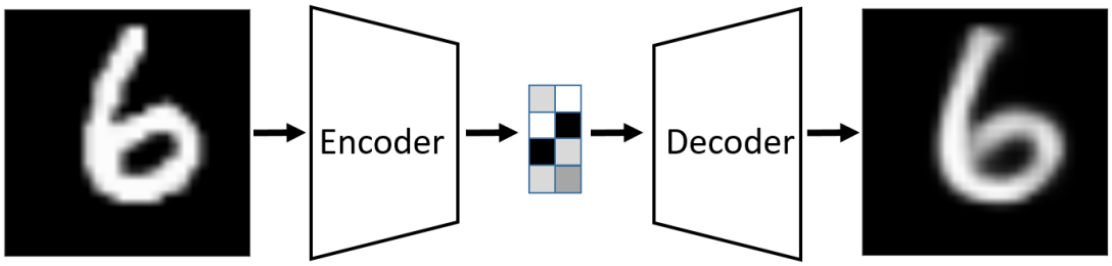

Figure 2.4: Visualization of an auto encoder. The input is encoded and subsequently decoded yielding and approximate reconstruction of the image [BKG20]

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.

2.9 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 billion 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.

2.10 Intrusion Detection System Performance Metrics

To measure the effectiveness of different IDSs a commonly used set of performance metrics has been devised to promote comparison between solutions. For binary classification (attack vs. benign) the basic metrics are

- **True Positive (TP)**: Number of samples correctly classified as attack
- **True Negative (TN)**: Number of samples correctly classified as benign
- **False Positive (FP)**: Number of samples falsely classified as attack
- **False Negative (FN)**: Number of samples falsely classified as benign

From these basic metrics, a variety of semantically more expressive metrics can be derived which describe different performance aspects of the classification task like overall accuracy or the rate of falsely raised alarms. Commonly used metrics are

- **Accuracy** is defined as the ration of correctly classified samples to total samples.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.4}$$

- **Precision** is defined as the ration of true positive samples to predicted positive samples and represents the confidence of attack detection.

$$Precision = \frac{TP}{TP + FP} \tag{2.5}$$

- **Recall or Detection Rate (DR)** is defined as the ration of true positive samples to total positive samples. The metric describes the probability that an attack will be detected by the IDS.

$$Recall = DR = \frac{TP}{TP + FN} \tag{2.6}$$

- **False Negative Rate (FNR) or Missed Alarm Rate (MAR)** is defined as the ratio of false negative samples to total positive samples and describes how many attacks go undetected by the IDS.

$$FNR = MAR = \frac{FN}{TP + FN} \tag{2.7}$$

- **False Positive Rate (FPR) or FAR** is defined as the ratio of false positive samples to predicted positive samples and describes how often the IDS falsely raises an alarm.

$$FPR = FAR = \frac{FP}{TP + FP} \tag{2.8}$$

2.11 Terminology

Acronyms

ANN	Artificial Neural Network
BCE	Binary Cross Entropy
BERT	Bidirectional Encoder Representations from Transformers
CNN	Convolutional Neural Network
DLSTM	Deep Long Short-Term Memory
FAR	False Alarm Rate
GPU	Graphics Processing Unit
IAT	Interarrival Time
IDS	Intrusion Detection System
LSTM	Long Short-Term Memory
LSTM-AE	LSTM-based Auto-Encoder
LSTM-SAE	LSTM-based Stacked Auto-Encoder
MAE	Mean Absolute Error
ML	Machine Learning
MLM	Masked LM
MTS	Multivariat Time Series
NIDS	Network Intrusion Detection System
NLP	Natural Language Processing
NN	Neural Network

NSP	Next Sentence Prediction
ReLU	Rectified Linear Unit
RMSE	Root Mean Square Error
RNN	Recurrent Neural Network
SEL	Squared Error Loss
SGD	Stochastic Gradient Descent
SMAPE	Symmetric Mean Absolute Percentage Error

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 Machine Learning for Network Intrusion Detection

Machine learning techniques have shown to be competitive to signature based expert systems when it comes to Network Intrusion Detection (NID). Hence there have been many attempts over the past years to implement various types of machine learning algorithms with considerable success. The design space for such systems is vast as Hongyu Liu and Bo Lang show in their 2019 paper [LL19] "Machine Learning and Deep Learning Methods for Intrusion Detection Systems: A Survey" where they enumerate the possible approaches to a machine learning based IDS and discuss their advantages and disadvantages. The authors classify proposed IDSs to date based on data sources and detection methods used to create a comprehensive taxonomy system. When using machine learning, further classification entails the type of machine learning methods used for implementing it. Based on their classification, our approach would be categorized as following: The data source is a network based IDS with flow based detection using deep learning methods. The detection method is machine learning based anomaly detection and the machine learning model is a deep learning model based on LSTM cells, but using both unsupervised and supervised learning methods. As such, our model is not even completely classifiable by the taxonomy proposed by Hongyu Liu et al..

insert image of classification of our approach based on this taxonomy

Congruent with our assumptions, Hongyu Liu and Bo Lang deem a flow based detection to be a suitable way to structuring the raw network data, as it represents the whole

network environment and retains a lot of contextual information but with the obvious drawback that package payload is being ignored. This leads to poor detection rates for User to Root (U2R) and Remote to Local (R2L) attacks like SQL injection or XSS attacks which our results confirm as can be seen in section 6.

Although the design space for ML based is much greater, we are focusing on Deep Learning (DL) based approaches which have shown to be superior to shallow networks in general. A good initial overview of state-of-the-art DL techniques applied to NID can be found in the 2020 paper "Fog-Based Attack Detection Framework for Internet of Things Using Deep Learning" [SYZ20] by Ahmed Samy et al. Although their it has its focus on IDSs for resource and energy constrained Internet of Things (IoT) networks, the authors provides a comprehensive overview of the performance of different up-to-date DL models applied to various NIDS data sets. The goal of their research is to implement powerful DL based IDSs in a resource constrained IoT network by outsourcing the processing to fog nodes which are capable of storing large amounts of data with low latency and are placed at the edge of the IoT network. In the context of their research, they implemented six state of the art DL models:

- **Deep Neural Network (DNN)** with an input layer of 1024 cells, five hidden layers with 512 cells each using ReLU activation functions and one output layer with one cell using a sigmoid activation function.
- **Long Short-Term Memory (LSTM)** with an input layer of 128 cells, three hidden layers with 256 cells each and one output layer with one cell using sigmoid activation
- **Bidirectional Long Short-Term Memory (Bi-LSTM)** with an input layer of 128 cells, three hidden layers with 128 cells each and one output layer with one cell using sigmoid activation
- **Convolutional Neural Network Long Short-Term Memory (CNN-LSTM)** with three convolutional layers with 64 filters, each using ReLU activation functions, three pooling layers, one LSTM layer with 256 cells and one output layer with one cell using sigmoid activation
- **Gated Recurrent Unit (GRU)** with an input layer of 64 cells, three hidden layers with 64 cells each and one output layer with one cell using sigmoid activation
- **Convolutional Neural Network (CNN)** with three convolutional layers with 64 filters each using ReLU activation functions, three pooling layers and one output layer with one cell using sigmoid activation

For multi-class classification the output layer is expanded to match the number of classes in the data set. Ahmed Samy et al. tested their implementations on five different data sets:

- UNSW-NB15 [MS15]
- CICIDS-2017 [SLG18]
- RPL-NIDS17 [VR19]
- N_BaIoT [MBM+18]
- NSL-KDD [TBLG09]

They used a feature representation of the data comprised of 80 network traffic features extracted with CICFLOWMETER. The best results in training where achieved with a learning rate of 0.01, a batch size of 64 using the Adam optimization algorithm. They used accuracy, precision, recall, F1-measure, FAR, DR as metrics corresponding with the definitions in section 2.10 to assess the performance of the different models. This research was especially relevant for as, because the authors used similar models and the same datasets, so a direct comparison with our results can be made. An overview of the relevant results for binary classifications can be found in table 3.1 and for multi-class classification in table 3.2.

cut tables to only include relevant stats

DataSet Name	DL Model	Accuracy (%)	Precision (%)	Recall (%)	F1-Measure (%)	FAR (%)	DR (%)
UNSW-NB15 Dataset	**DNN**	99.67	99.79	99.87	99.83	6.23	99.87
	LSTM	99.96	99.96	99.97	99.98	4.02	99.97
	Bi-LSTM	99.67	99.82	99.83	99.83	5.35	99.83
	GRU	99.58	99.79	99.77	99.78	6.21	99.77
	CNN	99.66	99.86	99.78	99.82	4.24	99.78
	CNN-LSTM	98.95	99.96	98.97	99.46	1.20	98.97
CICIDS-2017 Dataset	**DNN**	98.95	98.57	99.73	99.15	2.29	99.73
	LSTM	99.37	99.28	99.67	99.49	1.15	99.67
	Bi-LSTM	99.35	99.22	99.77	99.48	1.25	99.77
	GRU	99.35	99.21	99.73	99.47	1.26	99.73
	CNN	99.08	98.61	99.92	99.26	2.25	99.92
	CNN-LSTM	98.88	98.41	99.8	99.1	2.57	99.8
RPLNIDS-2017 Dataset	**DNN**	98.01	98.97	98.88	98.93	13.31	98.88
	LSTM	98.15	98.99	99.07	99.12	12.25	99.07
	Bi-LSTM	98.01	98.97	98.88	98.93	13.31	98.88
	GRU	98.01	98.97	98.88	98.92	13.08	98.88
	CNN	97.94	99.02	98.74	98.88	11.38	98.74
	CNN-LSTM	97.01	97.73	98.92	98.40	29.78	98.92
N_BaIoT-2018 Dataset	**DNN**	98.9	91.88	92.43	92.15	0.6	92.43
	LSTM	99.85	98.64	99.81	99.12	0.1	99.81
	Bi-LSTM	99.81	97.32	99.8	98.63	0.2	99.8
	GRU	99.57	96.36	98.28	97.31	0.31	98.28
	CNN	99.41	97.83	94.97	96.38	0.18	94.97
	CNN-LSTM	99.39	99.19	93.64	96.34	0.7	93.64
NSL-KDD Dataset	**DNN**	98.24	98.96	98.63	98.29	10.2	98.63
	LSTM	99.34	99.18	99.59	99.39	0.1	99.59
	Bi-LSTM	99.07	98.61	99.68	99.14	1.6	99.68
	GRU	98.85	98.56	99.31	98.93	1.6	99.31
	CNN	99.03	99.16	99.51	99.13	0.3	99.51
	CNN-LSTM	96.07	94.19	99.77	96.43	7.03	98.77

Figure 3.1: Evaluation metrics of DL models with the five data sets in binary classification. [SYZ20]

Attacks	DNN			LSTM			Bi-LSTM		
	P (%)	R (%)	F1-M (%)	P (%)	R (%)	F1-M (%)	P (%)	R (%)	F1-M (%)
Benign	98.41	99.71	99.06	99.48	99.15	99.31	99.04	99.24	99.14
DDoS	99.18	57.77	73.01	98.89	98.94	98.91	99.49	97.16	98.31
FTP-Patator	84.38	56.41	67.62	91.05	99.14	94.93	93.84	98.94	96.32
Port-Scan	47.66	74.78	58.21	99.42	99.91	99.66	99.21	99.97	99.59
SSH-Patator	39.31	29.27	33.56	100	70.68	82.82	93.73	79.03	85.75
Brute-Force	30.23	4.12	7.26	44.74	84.65	58.54	44.81	85.77	58.86
SQL-Injection	0	0	0	11.11	2.56	4.28	0	0	0
	GRU			**CNN**			**CNN-LSTM**		
	P (%)	R (%)	F1-M (%)	P (%)	R (%)	F1-M (%)	P (%)	R (%)	F1-M (%)
Benign	99.81	98.86	99.33	95.78	99.94	97.82	74.5	99.89	85.35
DDoS	99.06	98.94	99	99.96	98.68	99.32	99.77	98.93	99.35
FTP-Patator	89.13	99.79	94.16	99.76	84.58	91.54	96.42	49.2	65.16
Port-Scan	99.79	99.93	99.86	99.96	91.28	95.42	91	0.57	1.13
SSH-Patator	74.04	98.76	84.63	98.57	50.64	66.91	99.56	50.98	67.43
Brute-Force	43.8	84.65	57.73	26.14	3.16	5.64	0	0	0
SQL-Injection	0	0	0	0	0	0	0	0	0

Figure 3.2: Evaluation metrics of DL models with the five data sets in binary classification. [SYZ20]

As shown by the results and also stated in the paper: LSTM networks outperform other deep learning models consistently in attack detection and accuracy overall. This gives us confidence that our decision of working with LSTM networks was the right call as it adds to the real world relevance of our work. We also use a TransformerEncoder model for classification which was not included in the comparison done by Ahmed Samy et al. Furthermore, the models in their paper were trained fully supervised with at least 70% of the data available in each data set which differs from our approach of using very little labeled data and relying on self-supervised pre-training.

3.2 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]. In the following section we are looking at some of the related papers with regards to our research question. An overview of the methods and models used in these papers can be

found in table 3.2.

[H]

Paper	Model	Pretraining Task(s)	Downstream Task(s)	Val. Dataset(s)	Val. Value
[DCLT18] 3.2.1	Transformer Encoder	next sentence prediction, word masking	various NLP tasks	GLUE score	80.5% acc.
[SMS15] 3.2.2	DLSTM	auto-encoding, sequence prediction	human action recognition	UCF-101, HMDB-51	74.9%, 44.1% acc.
[SMS15] 3.2.2	DLSTM	stacked auto-encoding	MTS forecasting	capital bike sharing PM2.5 concentration in the air of CHINA	11.646, 9.864 SMAPE

3.2.1 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.

3.2.2 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.

Model	UCF-101 RGB	UCF-101 1-frame flow	HMDB-51 RGB
Single Frame	72.2	72.2	40.1
LSTM classifier	74.5	74.3	42.8
Composite LSTM Model + Finetuning	75.8	74.9	44.1

Table 3.1: Summary of results on Action Recognition [SMS15]

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 3.3 where both proxy tasks, reconstructing the input and predicting the future, are combined to produce a single model.

The pre-trained models are then finetuned for their classification task on the mentioned training datasets. With the pre-trained + finetuned composite model, the authors achieved an absolute increase of 1.3% accuracy for both the UCF-101 and the HMDB-51 datasets over a conventional LSTM classifier as can be seen in table 3.1.

For our thesis we used the same Auto-Encoder and composite model for pre-training as explained in sections **??** and **??**. We tried with both conditioned and unconditioned models, comparing results in section 6.

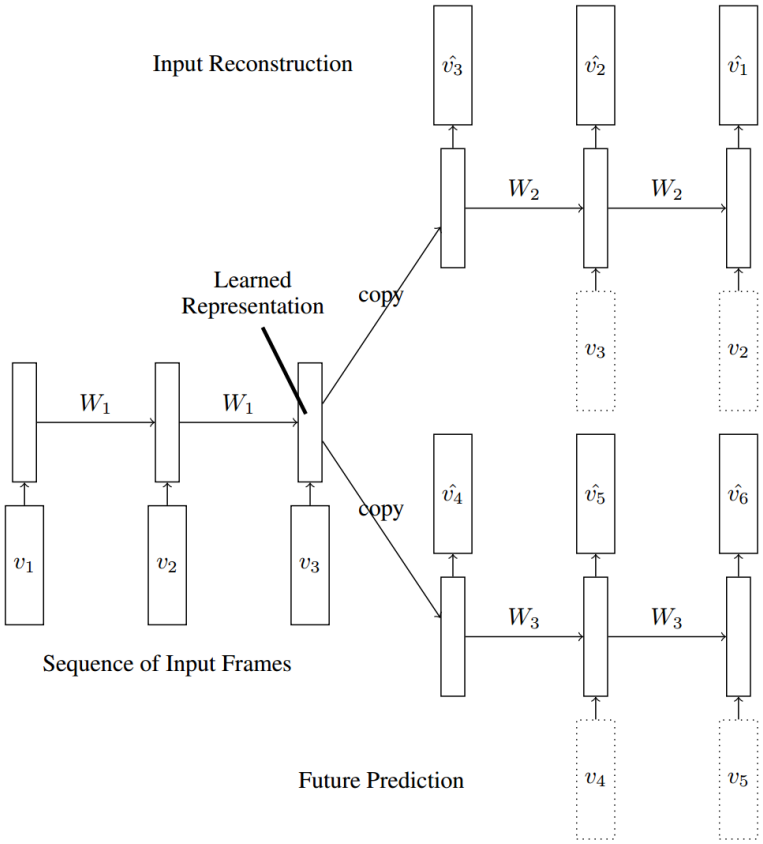

Figure 3.3: Composite model for input reconstruction and future prediction [SMS15]

3.2.3 Unsupervised Pre-training of a Deep LSTM-based Stacked Autoencoder for Multivariate Time Series Forecasting Problems

In their 2019 paper [SK19b] Alaa Sagheer et al. explore the benefits of unsupervised pre-training using stacked LSTM auto encoders with subsequent supervised finetuning. Their goal was to improve the prediction capabilities for Multivariat Time Series (MTS) problems. In their previous paper [SK19a], the authors showed the effectiveness of Deep Long Short-Term Memory (DLSTM) based models for MTS prediction tasks. In their 2019 paper, they showed the improvements resulting from pre-training when compared to an initial random initialization of weights when working with DLSTM models. Compared to shallow LSTM networks, DLSTM networks contain multiple layers of LSTM cells stacked on each other. Information travels the network from left to right and from bottom to top as is depicted in figure 3.4.

For pre-training the network, the authors use a LSTM-based Stacked Auto-Encoder (LSTM-SAE) model. In contrast to a conventional auto encoder like described in 2.8, a stacked auto encoder model uses multiple encoder and decoder layers as can be seen

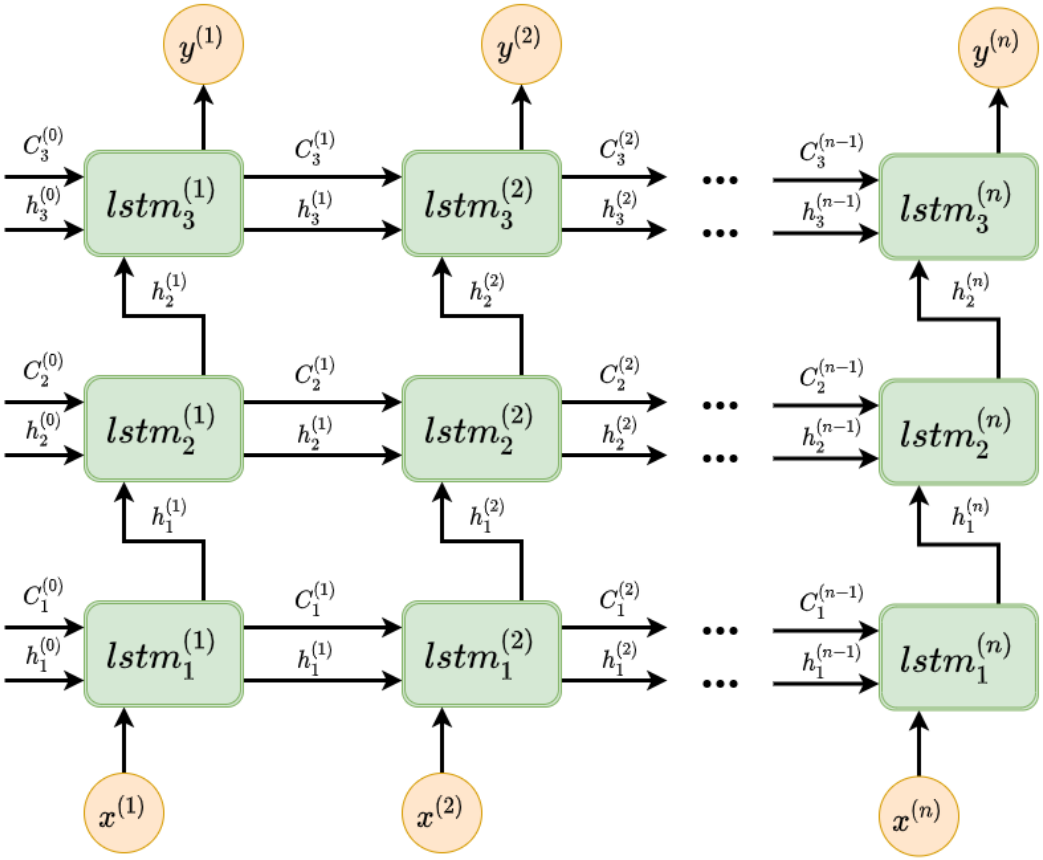

Figure 3.4: Depiction of a three layered LSTM network

in figure 3.5. The different encoder layers are trained individually in a multi phased training procedure: Train the first auto encoder layer like a conventional LSTM-based Auto-Encoder (LSTM-AE) with the target being the original input data. Cut the decoder part of the first LSTM-AE. When training the second LSTM-AE, the input is encoded by both the encoder of the first and second LSTM-AE block and then decoded only by the decoder of the second LSTM-AE. The target data for training the second LSTM-AE is again the original input, and not the reconstructed input of the first LSTM-AE. The training process is depicted in figure 3.5. This process is then repeated for arbitrarily many LSTM-AE layers. The authors tried both one and two stacked layers of LSTM-AE. The trained encoder blocks are then used to initialize a multi layered DLSTM.

To complete the training phase, the parameters of the pre-trained DLSTM are then finetuned in a supervised fashion. This is done by adding an output layer which produces values of the dimension of labels of the training set used and of the variables to be predicted. In the case of the authors, the output layer was a single neuron to predict a single variable. For supervised finetuning and validation they used two datasets:

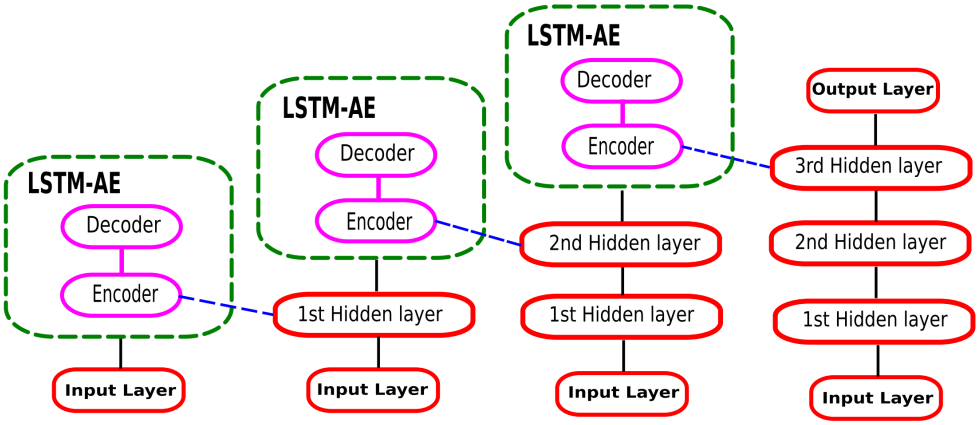

Figure 3.5: Layer-wise pre-training of LSTM-SAE model. [SK19b]

Model	No. of hidden layer	Dropout	lag	batch	RMSE	MAE	SMAPE
DLSTM	1	0.4	20	146	52.062	32.468	12.088
DLSTM	2	0.3	25	219	49.811	31.524	12.183
LSTM-SAE	1	0.1	30	219	49.389	32.192	13.878
LSTM-SAE	2	0.1	30	73	46.927	30.041	11.646

Table 3.2: The results of DLSTM and LSTM-SAE using data set 1 [SK19b]

1. *The capital bike sharing dataset*
2. *The PM2.5 concentration in the air of CHINA dataset*

For the first dataset, the model tried to predict how many bike will be rented on a particular day based on parameters like *Season, Holiday, Weekday, Working day*, etc. For the second dataset, the task was to predict PM2.5 concentrations in the air for various Chinese cities based on parameters like *Dew Point, Temperature, Pressure, Combined Wind Direction,* etc.

As metric of performance, the authors used Root Mean Square Error (RMSE), Mean Absolute Error (MAE) and Symmetric Mean Absolute Percentage Error (SMAPE) (lower is better) which all describe the difference between predicted value and observed value. The results of evaluation with both data sets can be seen in tables 3.2 and 3.3.

The results in tables 3.2 and 3.3 show that unsupervised pre-training improved final accuracy and led to better and faster convergence.

For this thesis, we used only a single LSTM-AE network, but with three layers of LSTM cells making it a DLSTM for both encoder and decoder.

Model	No. of hidden layer	Dropout	lag	batch	RMSE	MAE	SMAPE
DLSTM	1	0.2	30	60	23.993	12.124	10.919
DLSTM	2	0.2	30	73	23.750	12.452	12.181
LSTM-SAE	1	0.1	30	219	23.907	12.509	11.052
LSTM-SAE	2	0.3	25	146	24.041	12.060	9.864

Table 3.3: The results of DLSTM and LSTM-SAE using data set 2 [SK19b]

- 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

As summarized by the survey paper [LL19] of Hongyu Liu and Bo Lang in section **??**, the design space for ML based NIDS is vast and can be hard to navigate at times. Hence, careful consideration of data representation, data pre-processing, ML models, model parameters and training hyperparameters is necessary. The goal of the thesis was to examine the benefit of pre-training for already established DL models in the context of NIDS, hence we wanted to start from the most effective DL models available, which, derived from state-of-the-art research, seems to be a uni-directional multi-layer DLSTM network. Additionally, we also looked at research on DL and self-supervised/unsupervised learning in general. Most of the state-of-the-art research on DL, and especially in self-supervised/unsupervised and transfer learning, is done in the context of NLP [DCLT18], [PNI+18], [VSP+17a].

Network communication is similar to natural language in the sense that it follows a certain set of rules, a grammar so to say, and each token like a word or packet conveys some semantic meaning when in the context of a sentence or flow. Other researchers have made similar observations [RATS18]. We deemed the TransformerEncoder, which is often cited as an advancement of the LSTM in sequence modeling, to be a powerful new tool for network traffic classification.

4.1 Datasets

We used the two NIDS datasets *UNSW-NB15* [MS15] and *CIC-IDS-2017* [SLG18].

The **UNSW-NB15** dataset [MS15], created by Nour Moustafa et al. from the School of Engineering and Information Technology, University of New South Wales at the Australian Defence Force Academy, Australia, was released as an update for the formerly frequently used but deprecated [MS15] KDD dataset family. It was designed to reflect most known low foot print attacks at time of publication. The records are bidirectional flows extracted

from the raw traffic data and grouped by the commonly used *<srcIP, dstIP, srcPort, dstPort, protocol>* tuple. Each flow is described by 47 derived or measured features e.g. total duration, bytes transmitted, the mean of the flow packet size transmitted by destination IP, etc. The dataset contains a total of 2.54 million records of which 2.22 million are natural transaction data labeled as benign and 0.32 million attack records meaning 87.4% of data is classified as benign. Attack records can be further divided into nine attack categories as listed in table 4.1.

Type	**No. Records**	**% w.r.t. majority class**	**% w.r.t. majority attack class**	**% w.r.t. total instances**
Normal	2218761	100.000	1029.678	87.351
Fuzzers	24246	1.093	11.252	0.955
Analysis	2677	0.121	1.242	0.105
Backdoor	2329	0.105	1.081	0.092
DoS	16353	0.737	7.589	0.644
Exploits	44525	2.007	20.663	1.753
Generic	215481	9.712	100.000	8.483
Reconnaissance	13987	0.630	6.491	0.551
Shellcode	1511	0.068	0.701	0.059
Worms	174	0.008	0.081	0.007

Table 4.1: UNSW-NB15 dataset record distribution [MS15].

The dataset was generated from a synthetic environment comprised of 3 networks and 45 distinct IP addresses using the IXIA PerfectStorm (now keysight PerfectStorm) tool.

The **CIC-IDS-2017** dataset [SLG18], created by Iman Sharafaldin et. al from Canadian Institute for Cybersecurity (CIC), University of New Brunswick (UNB), Canada constitutes another updated representation of known attack types at the time of publication. Compared to the UNSW-NB15 dataset it contains records of more modern cyber attacks like Heartbleed, HULK DoS but leaves out e.g. worm attacks. It contains a total of 2.83 million records of which 2.36 million are labeled as benign and 0.47 million are attack records. In other words, 83.35% of the dataset are benign records and 16.61% attack records. Attack records can are further classified as shown in table 4.2.

Type	No. Records	% w.r.t. majority class	% w.r.t. majority attack class	% w.r.t. total instances
Benign	2359087	100.000	1020.932	83.363
Bot	1966	0.083	0.851	0.069
DDoS	41835	1.773	18.105	1.478
DoS GoldenEye	10293	0.436	4.454	0.364
DoS Hulk	231072	9.795	100.000	8.165
DoS Slow – httptest	5499	0.233	2.380	0.194
DoS slowloris	5796	0.246	2.508	0.205
FTP-Patator	7938	0.336	3.435	0.281
Heartbleed	11	0.000	0.005	0.000
Infiltration	36	0.002	0.016	0.001
PortScan	158930	6.737	68.779	5.616
SSH-Patator	5897	0.250	2.552	0.208
Web Atk. – Brute Force	1507	0.064	0.652	0.053
Web Atk. – Sql Injection	21	0.001	0.009	0.001

Table 4.2: CIC-IDS-2017 dataset record distribution [PB18].

In contrast to the UNSW-NB15 network simulation environment, the network topology consists of two networks: A highly secured victim network with firewall, router, switches and most common operating systems. The attack network is a separate network, containing a set of PCs with public IPs and running Windows 8.1 and Kali Linux.

To reduce variance in results and to keep comparability high we use the same random seed for all experiments and use stratified sampling when we divide the data sets into smaller chunks for pretraining, training and validation to assure the each attack category is represented in the subset proportionally to the whole dataset. This is especially important if we use very small subsets (10 flows per attack category) for finetuning as described in section 5.

4.2 Data Representation

Network traffic data can be viewed from a multitude of perspectives ranging from aggregate statistical data over different time-frames [MDES18] to looking at feature representations of single packets. These 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 sequences of 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, which is a very common approach [WZA06], [MS15], [MZIV18]. We chose a flow representation since this approach has shown good results frequently, is easy to obtain, requires no domain knowledge, and is feasible for encrypted traffic [MZIV18]. The downside of this approach is that the packet payload is ignored, which leads to poor classification performance for U2R and R2L [TBLG09] like SQL injection, XSS and other payload based attacks. This is also shown by our results as can be seen in section 6. Commonly, flows are represented as a single feature vector in the dataset, containing aggregated statistical data of the completed flow like e.g. the mean of the flow packet size transmitted by the source IP, source to destination packet count or bits per second or total duration of the transmission. The downside of this is that flows can be processed only once they are completed [HBFZ19]. In a real world scenario, this approach has major downsides since the IDS can only inspect communications in retrospect and never in real time. This was one of the reasons we decided to represent our flows as a sequence of packets instead of a single aggregated feature vector. Each packet is represented by the 15 features in table 4.3.

#	Name	Type	Constant over flow	Description
0	srcPort	Int	yes	Source port number
1	dstPort	Int	yes	Destination port number
2	protocol	Int	yes	Protocol identifier
3	pktLength	Int	no	Destination port number
4	pktIat	Int	no	Interarrival Time (IAT)
5	pktDirection	Bool	no	Sending direction of the packet
6	synFlag	Bool	no	TCP SYN Flag
7	finFlag	Bool	no	TCP FIN Flag
8	rstPort	Bool	no	TCP RST Flag
9	pshFlag	Bool	no	TCP PSH Flag
10	ackFlag	Bool	no	TCP ACK Flag
11	urgFlag	Bool	no	TCP URG Flag
12	eceFlag	Bool	no	TCP ECE Flag
13	cwrFlag	Bool	no	TCP CWR Flag
14	nsFlag	Bool	no	TCP NS Flag

Table 4.3: Packet features [PB18].

Another reason why we used a packet sequence instead of an aggregate flow representation is its similarity to natural language as mentioned before, which enables us to apply state-of-the-art methods from DL based NLP. We used the data pre-processing from [HBFZ19] as it fit the requirements for our experiments and was easily modifiable.

To keep gradient paths shorter and to improve training stability, packet sequences are truncated to be at most 100 packets long. As last step of pre-processing features are normalized to be within a range of $[-1, 1]$ to make gradient descent converge quicker.

4.3 Machine Learning Models

To inspect the potential benefits of self-supervised pretraining 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 in the domains of IDS and/or NLP . Both types of networks are generally susceptible to improvements through self-supervised pretraining as prior research has shown [DCLT18]

give examples

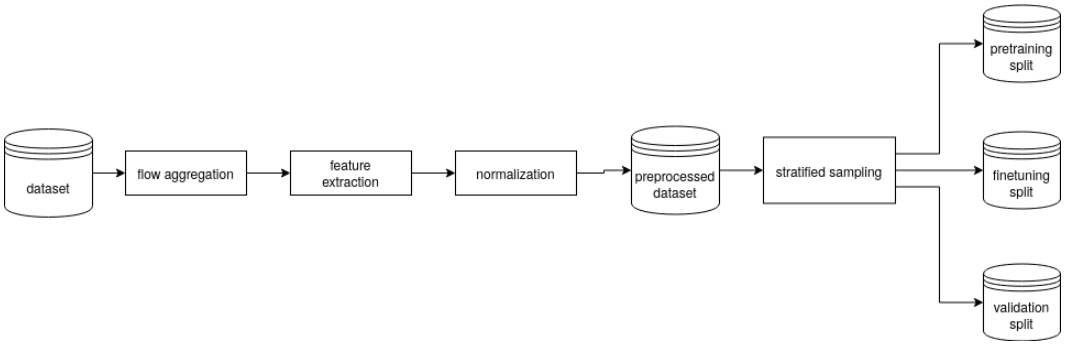

Figure 4.1: All steps performed in dataset preprocessing to yield pretraining, training and validation splits.

[SMS15] [SK19b]. Whether pretraining improves performance in the context of NIDS remains to be shown and is subject of this thesis.

For our **LSTM network** we chose a three layer DLSTM with a *hidden size* of 512. While a larger network might be slightly more effective, this configuration proved to be swiftly trainable while also producing results close to those achieved by other researchers using LSTM models applied to the same datasets [SYZ20]. Since our main focus is on comparisons between different training methods applied to the same model, it is not necessary 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. Independent of the context, LSTM models have shown to be sensible to initialization of their weights and hidden state. This can be seen as a drawback but also as an opportunity to increase performance or decrease learning time. While there are many ways to initialize the parameters of an LSTM network, the most common ones are 0-initialization, random initialization and some form of transfer learning which in our case is self-supervised pretraining. For pretraining, the output layer is a linear layer with 15 nodes, equal to the number of features, and no activation function. For binary classification the output layer is replaced with a linear layer with a single node and no activation function because the objective function binary classification Binary Cross Entropy (BCE) loss operates on logits. This results in a sequence2sequence model which generates output sequences equal to the length of the input sequence. For pre-training this is the desired result, but for binary classification would only require a sequence2scalar model. So for classification, only the output of the last stage is looked at because at that point, the whole input sequence was processed by the model and information in a (uni-directional) LSTM only flows in one direction. The output of this stage should therefore be most accurate.

The Transformer model as proposed by Vaswani et. al [VSP+17a] and its derivative score

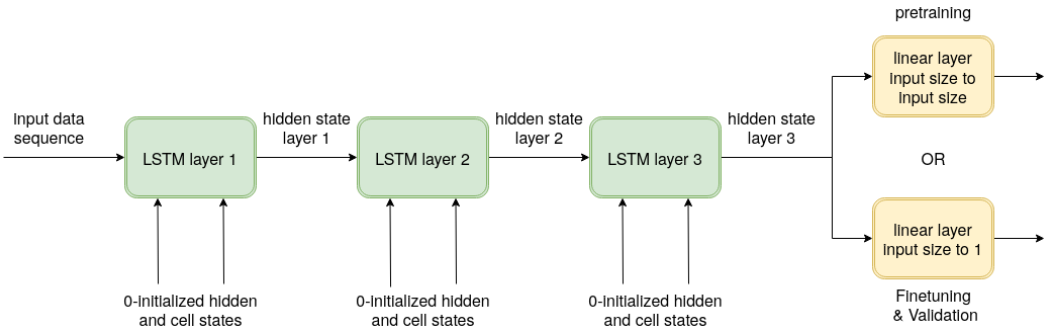

Figure 4.2: Depiction of the LSTM model.

high on the leader boards of NLP benchmarks like GLUE [WSM+18] or SQuAD [**?**] and are still considered state-of-the-art in many regards. Following the example of BERT we only used the encoder part of the *Transformer network* since the decoder does not provide any benefits 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 2, meaning an internal representation size double to the number of features per packet. For pretraining, the output layer is a linear layer with 15 nodes, equal to the number of features, and no activation function. For binary classification the output layer is replaced with a linear layer with a single node and no activation function because the objective function for binary classification BCE loss operates on logits. This again results in a sequence2sequence model which produces output sequences of length equal to the input sequence. For binary classification, we only need a single value, but the Transformer Encoder model produces a sequence of values. In contrast to the LSTM model, information in the Transformer does not aggregate at a specific stage and therefore we can't identify an output token which has more information or is more likely to be accurate than others. Google solved this problem for BERT by prepending a classification token to every input sequence and the model learns to aggregate information regarding classification at the corresponding output token. We tried this approach with no success, perhaps due to insufficient training. We opted for an average pooling layer over all output tokens to get from a sequence of variable length to a single value and this approach also works as can be seen later in the results section 6.2.

insert image of transformer model

4.4 Framework and Training

To conduct our experiments we used PyTorch [ea16] to implement and train our proposed models. To save labor and to keep results comparable we used the standard implementations *torch.nn.LSTM* and *torch.nn.TransformerEncoder*. Training is conducted by a standard training loop going through forward and back propagation, calculating losses and updating weights for each batch. Noteworthy is the use of gradient clipping to a

maximum of 1 and the use of a learning rate scheduler witch decreases the learning rate by a factor of 0.1 if mean batch loss is plateauing during training. As optimization criterion for pretraining we use L1 loss, i.e. MAE (*nn.L1Loss*). For supervised training we use BCE loss with mean reduction on logits directly (*nn.BCEWithLogitsLoss*). 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 [KB14] and is appropriate for noisy or sparse gradients which can occur when working with RNNs in general. We developed and implemented a framework in Python to automate the experiments, generate statistics, plots and metrics.

The models were trained on two NVidia GeForce RTX 2070 Super GPUs with a combined performance of 19.4 Teraflops per second for 32bit float (FP32) values. We tried using 16bit float (FP16) values during training together with the *torch.cuda.amp.GradScaler*. This significantly decreased training time but introduced numerical instability in some cases and we went back to training with FP32 values. Training was performed with a batch size of 128 for the LSTM model and 1024 for the TransformerEncoder model. Further increasing batch size did neither improve final accuracy nor did it decrease the time in which training converges.

- 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 4.3) to describe the processing steps

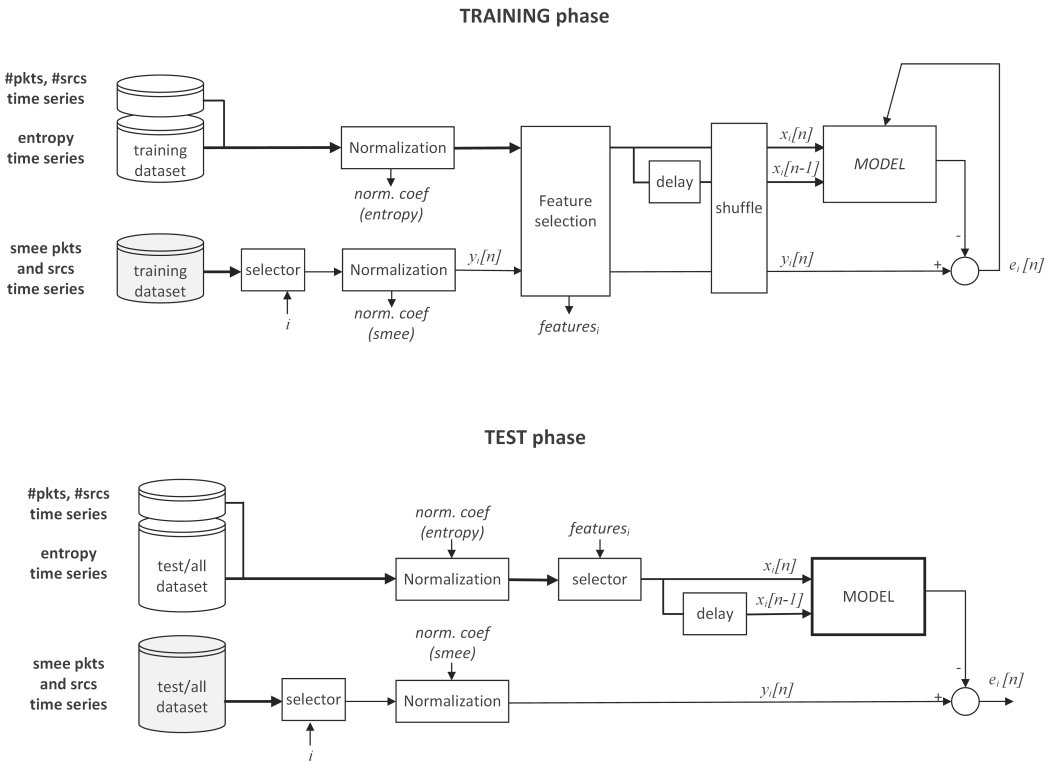

Figure 4.3: Describe in the caption exactly what can be seen in the figure

CHAPTER 5

Experiments

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 10%, 1% and with even smaller subsets. The dataset specific subsets, here labeled *CIC17_10* and *UNSW15_10* contain 10 flows per attack category and an adequate amount of benign samples to retain the ratio of attack to benign samples from the original dataset to keep some comparability. For the CIC-IDS-2017 dataset, the subset contains 10 records for each of the 13 attack categories and 651 benign records. For UNSW-NB15 it is 10 records for each of the 9 attack categories and 621 benign records. For both datasets, this constitutes just 0.028% of the total number of samples included. Experiments are numbered so they can be referenced in later sections.

As pretraining method we devised a list of proxy tasks which should challenge the model to build an abstract representation of the data within its hidden space or at least learn

#	Model	Dataset	Batch size	Subset	Training %	Training Eps.
1.1.1	LSTM	CIC-IDS-2017	128	-	90	40
1.1.2	LSTM	CIC-IDS-2017	128	-	10	40
1.1.3	LSTM	CIC-IDS-2017	128	-	1	200
1.1.4	LSTM	CIC-IDS-2017	128	CIC17_10	-	600
1.1.5	LSTM	UNSW-NB15	128	-	90	40
1.1.6	LSTM	UNSW-NB15	128	-	10	40
1.1.7	LSTM	UNSW-NB15	128	-	1	200
1.1.8	LSTM	UNSW-NB15	128	CIC17_10	-	600
1.2.1	Transformer	CIC-IDS-2017	512	-	90	40
1.2.2	Transformer	CIC-IDS-2017	512	-	10	40
1.2.3	Transformer	CIC-IDS-2017	512	-	1	200
1.2.4	Transformer	CIC-IDS-2017	512	UNSW15_10	-	600
1.2.5	Transformer	UNSW-NB15	512	-	90	40
1.2.6	Transformer	UNSW-NB15	512	-	10	40
1.2.7	Transformer	UNSW-NB15	512	-	1	200
1.2.8	Transformer	UNSW-NB15	512	UNSW15_10	-	600

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

which features are more important than others and correct weights accordingly. A list of all proxy tasks can be seen in table **??**. Each of them will be explained in detail in the sections below.

Section(s)	Label	Name	Description
5.1.1	IDENTITY	Identity Function	Reconstruct exact input
5.1.2	PREDICT	Predict Packet	Predict the next packet at each stage of the LSTM
5.1.3, 5.2.3	MASKP	Mask Packets	Reconstruct masked packets in the sequence
5.1.4, 5.2.1	MASKF	Mask Features	Reconstruct masked features
5.1.5, 5.2.2	AUTO	Auto-Encoder	Encode and decode input with minimal loss
5.1.6	COMPOSITE	Composite Task	Combination of prediction and auto-encoding

Table 5.2: Devised proxy tasks for pretraining of DL models.

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

For pre-training the LSTM we devised six 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, a sequence2sequence Auto Encoder and a composite task comprised of part auto encoding and part prediction. The 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 somewhat robust to changes on the initial learning rate. For every proxy task, the model has been trained with the different parameters in table 5.3 to establish comparable results.

- 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 this 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.

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 **??**. The key difference however, is 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 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.

5.1.4 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

5.1.5 Auto-Encoder

As explained in section 2.8, 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. As template we used the model proposed by Nitish Srivastava et al. in their paper "Unsupervised Learning of Video Representations using LSTMs" [SMS15], but similar proposals for Auto-Encoders with LSTMs can be found in [SK19b] or [YLZ20].

The encoder LSTM compresses the whole input sequence $x_e^{(t)} = (x_e^{(1)}, x_e^{(2)}, ..., x_e^{(n)})$ into the hidden 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, either the output of the current stage $\hat{y}^{(t)}$ or the target token of the current stage $x^{(t)}$, the ground truth, 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

5.1.6 Composite model

For the composite model we recreated the network proposed by Nitish Srivastava et al. in their paper "Unsupervised Learning of Video Representations using LSTMs" [SMS15] as summarized in section 3.2.2. As a self-supervised pre-training proxy task, the model is fed half the packet sequence of a flow and is tasked with both reconstructing the part of the sequence it had access to, and predicting the missing part of the flow which it had no access to. The output of the model is a sequence $\hat{y}^{(t)}$ of length equal to the original input sequence $x^{(t)}$ of which the first half is reconstructed and the second half is predicted by the model. The loss is again calculated as the MAE (*L1Loss* with *mean* reduction) between the original input and the output sequence of the model. The model consists of three LSTMs which can be labeled *encoder*, *decoder* and *predictor* as can be seen in figure 5.1. The encoder processes the first half of the original input sequence, constructing an abstract representation in its hidden state. The hidden state of the last stage of the encoder LSTM is copied to both the decoder and predictor LSTMs as initial hidden state. Exactly like the Auto-Encoder from the previous section **??** the decoder LSTM tries to recreate the input sequence. Initialized with the final hidden state of the encoder, the predictor LSTM tries foretell future packets of the flow. At every stage of the predictor LSTM (except the first), either the output $\hat{y^{(t-1)}}$ of the previous stage or the target token $x^{(t-1)}$ of the previous stage is used as input token for the next stage. The authors of [SMS15] label those two methods *conditioned* and *uncondition* as is further explained in section 3.2.2.

make own graphic with labels encoder, decoder, predictor

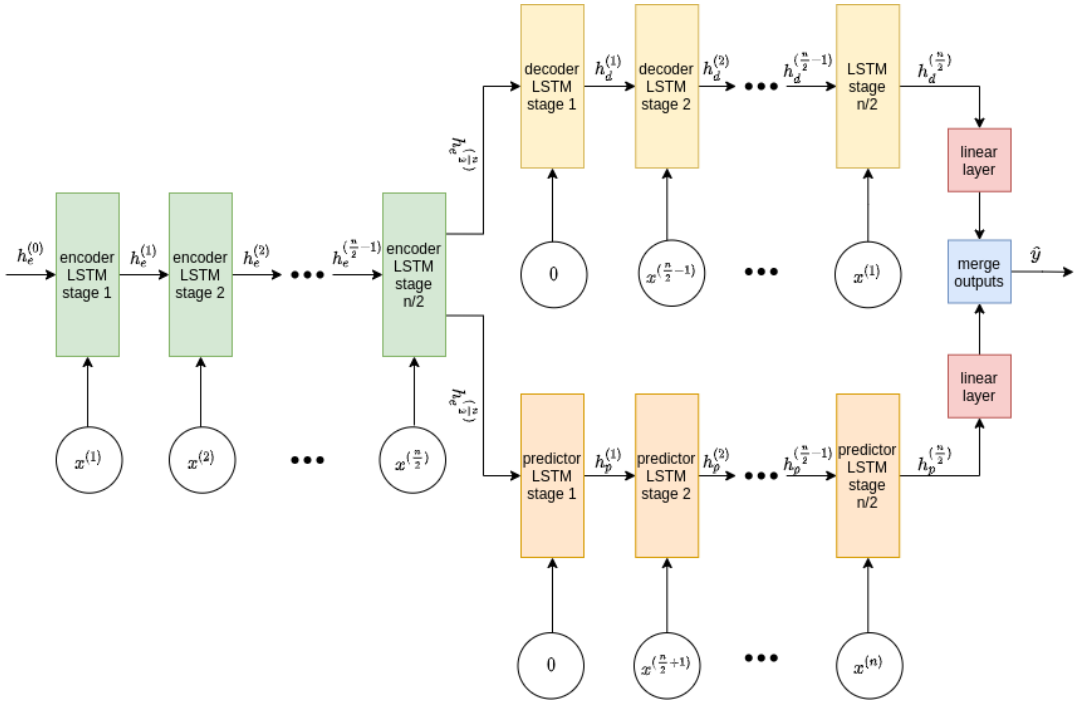

Figure 5.1: Composite model for input reconstruction and future prediction

5.2 Self-supervised Pre-training for Transformer Networks

Like with the LSTM we devised a series of proxy tasks for 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.

#	Dataset	Subset	Training %	Training Eps.	Proxy Task	Pretr. %	Pretr. Eps.
3.1.1	CIC-IDS-2017	-	10	40	NONE	0	0
3.1.2	CIC-IDS-2017	-	10	40	AUTO	80	10
3.1.3	CIC-IDS-2017	-	10	40	MASKF	80	10
3.1.4	CIC-IDS-2017	-	10	40	MASKP	80	10
3.2.1	CIC-IDS-2017	-	1	200	NONE	0	0
3.2.2	CIC-IDS-2017	-	1	200	AUTO	80	10
3.2.3	CIC-IDS-2017	-	1	200	MASKF	80	10
3.2.4	CIC-IDS-2017	-	1	200	MASKP	80	10
3.3.1	CIC-IDS-2017	CIC17_10	-	600	NONE	0	0
3.3.2	CIC-IDS-2017	CIC17_10	-	600	AUTO	80	10
3.3.3	CIC-IDS-2017	CIC17_10	-	600	MASKF	80	10
3.3.4	CIC-IDS-2017	CIC17_10	-	600	MASKP	80	10
3.4.1	UNSW-NB15	-	10	40	NONE	0	0
3.4.2	UNSW-NB15	-	10	40	AUTO	80	10
3.4.3	UNSW-NB15	-	10	40	MASKF	80	10
3.4.4	UNSW-NB15	-	10	40	MASKP	80	10
3.5.1	UNSW-NB15	-	1	200	NONE	0	0
3.5.2	UNSW-NB15	-	1	200	AUTO	80	10
3.5.3	UNSW-NB15	-	1	200	MASKF	80	10
3.5.4	UNSW-NB15	-	1	200	MASKP	80	10
3.6.1	UNSW-NB15	UNSW15_10	-	600	NONE	0	0
3.6.2	UNSW-NB15	UNSW15_10	-	600	AUTO	80	10
3.6.3	UNSW-NB15	UNSW15_10	-	600	MASKF	80	10
3.6.4	UNSW-NB15	UNSW15_10	-	600	MASKP	80	10

Table 5.4: Training and pretraining configurations for TransformerEncoder model with different proxy tasks.

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 Auto-Encoder

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].

#	Dataset	Subset	Training %	Training Eps.	Proxy Task	Pretr. %	Pretr. Eps.
2.1.1	CIC-IDS-2017	-	10	40	NONE	0	0
2.1.2	CIC-IDS-2017	-	10	40	PREDICT	80	10
2.1.3	CIC-IDS-2017	-	10	40	MASKF	80	10
2.1.4	CIC-IDS-2017	-	10	40	MASKP	80	10
2.1.5	CIC-IDS-2017	-	10	40	AUTO	80	10
2.1.6	CIC-IDS-2017	-	10	40	IDENTITY	80	10
2.1.7	CIC-IDS-2017	-	10	40	COMPOSITE	80	10
2.2.1	CIC-IDS-2017	-	1	200	NONE	0	0
2.2.2	CIC-IDS-2017	-	1	200	PREDICT	80	10
2.2.3	CIC-IDS-2017	-	1	200	MASKF	80	10
2.2.4	CIC-IDS-2017	-	1	200	MASKP	80	10
2.2.5	CIC-IDS-2017	-	1	200	AUTO	80	10
2.2.6	CIC-IDS-2017	-	1	200	IDENTITY	80	10
2.2.7	CIC-IDS-2017	-	1	200	COMPOSITE	80	10
2.3.1	CIC-IDS-2017	CIC17_10	-	600	NONE	0	0
2.3.2	CIC-IDS-2017	CIC17_10	-	600	PREDICT	80	10
2.3.3	CIC-IDS-2017	CIC17_10	-	600	MASKF	80	10
2.3.4	CIC-IDS-2017	CIC17_10	-	600	MASKP	80	10
2.3.5	CIC-IDS-2017	CIC17_10	-	600	AUTO	80	10
2.3.6	CIC-IDS-2017	CIC17_10	-	600	IDENTITY	80	10
2.3.7	CIC-IDS-2017	CIC17_10	-	600	COMPOSITE	80	10
2.4.1	UNSW-NB15	-	10	40	NONE	0	0
2.4.2	UNSW-NB15	-	10	40	PREDICT	80	10
2.4.3	UNSW-NB15	-	10	40	MASKF	80	10
2.4.4	UNSW-NB15	-	10	40	MASKP	80	10
2.4.5	UNSW-NB15	-	10	40	AUTO	80	10
2.4.6	UNSW-NB15	-	10	40	IDENTITY	80	10
2.4.7	UNSW-NB15	-	10	40	COMPOSITE	80	10
2.5.1	UNSW-NB15	-	1	200	NONE	0	0
2.5.2	UNSW-NB15	-	1	200	PREDICT	80	10
2.5.3	UNSW-NB15	-	1	200	MASKF	80	10
2.5.4	UNSW-NB15	-	1	200	MASKP	80	10
2.5.5	UNSW-NB15	-	1	200	AUTO	80	10
2.5.6	UNSW-NB15	-	1	200	IDENTITY	80	10
2.5.7	UNSW-NB15	-	1	200	COMPOSITE	80	10
2.6.1	UNSW-NB15	UNSW15_10	-	600	NONE	0	0
2.6.2	UNSW-NB15	UNSW15_10	-	600	PREDICT	80	10
2.6.3	UNSW-NB15	UNSW15_10	-	600	MASKF	80	10
2.6.4	UNSW-NB15	UNSW15_10	-	600	MASKP	80	10
2.6.5	UNSW-NB15	UNSW15_10	-	600	AUTO	80	10
2.6.6	UNSW-NB15	UNSW15_10	-	600	IDENTITY	80	10
2.6.7	UNSW-NB15	UNSW15_10	-	600	COMPOSITE	80	10

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

CHAPTER 6

Results

In this chapter, we discuss results and try to explain them based on neuron activation and Partial Dependency (PD) plots.

- 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/random initialization/0-initialization
- comparison of validation loss and accuracy convergence for different pretraining tasks

	NONE	AUTO	ID	COMPOSITE	OBSCURE	PREDICT
Hyperparameters						
Epochs Supervised	50	50	50	50	50	50
Epochs Pretraining	50	10	10	10	10	10
Batch size	128	128	128	128	128	128
Proxy task	NONE	AUTO	ID	COMPOSITE	OBSCURE	PREDICT
Pretraining percentage	0.00 %	80.00 %	80.00 %	80.00 %	80.00 %	80.00 %
Training percentage	10.00 %	10.00 %	10.00 %	10.00 %	10.00 %	10.00 %
Validation percentage	10.00 %	10.00 %	10.00 %	10.00 %	10.00 %	10.00 %
Specialized subset						
Learning rate	0.001	0.001	0.001	0.001	0.001	0.001
Random Seed	500	500	500	500	500	500
Modelparameters						
Hidden size	512	512	512	512	512	512
# Layers	3	3	3	3	3	3
Training metrics						
Best epoch	49	46	48	47	49	47
Time to best epoch	1h 34m	1h 59m	1h 43m	3h 13m	1h 40m	1h 35m
Performance metrics						
Accuracy	99.632 %	99.630 %	99.727 %	99.733 %	99.620 %	99.705 %
False alarm rate	0.289 %	0.337 %	0.229 %	0.229 %	0.384 %	0.278 %
Missed alarm rate	1.172 %	1.129 %	0.854 %	0.829 %	1.123 %	0.892 %
Detection rate	98.828 %	98.871 %	99.146 %	99.171 %	98.877 %	99.108 %
Precision	99.711 %	99.663 %	99.771 %	99.771 %	99.616 %	99.722 %
Specificity	99.903 %	99.887 %	99.923 %	99.923 %	99.871 %	99.907 %
Recall	99.711 %	99.663 %	99.771 %	99.771 %	99.616 %	99.722 %
F1-Measure	99.711 %	99.663 %	99.771 %	99.771 %	99.616 %	99.722 %

6.1 Long Short-Term Memory Network

6.1.1 Identity Function

- camparison identity function with both datasets

6.1.2 Predict Packet

6.1.3 Mask Features

- compare differences strategies for masking features

6.1.4 Mask Packets

6.1.5 Auto-Encoder

- compare differences between conditioned and unconditioned model

6.1.6 Composite model

- compare differences between conditioned and unconditioned model
- 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

6.2.1 Mask Features

6.2.2 Autoencoder

6.2.3 Mask Packet

6.3 Explainability

- close look at differences in performance for different attack classes
- partial dependency plots
- neuron activation
- DLSTMs and transformer encoder already very effective, so improvement is hard

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

Acronyms

ANN	Artificial Neural Network
BCE	Binary Cross Entropy
BERT	Bidirectional Encoder Representations from Transformers
CNN	Convolutional Neural Network
DLSTM	Deep Long Short-Term Memory
FAR	False Alarm Rate
GPU	Graphics Processing Unit
IAT	Interarrival Time
IDS	Intrusion Detection System
LSTM	Long Short-Term Memory
LSTM-AE	LSTM-based Auto-Encoder
LSTM-SAE	LSTM-based Stacked Auto-Encoder
MAE	Mean Absolute Error
ML	Machine Learning
MLM	Masked LM
MTS	Multivariat Time Series
NIDS	Network Intrusion Detection System
NLP	Natural Language Processing
NN	Neural Network

NSP	Next Sentence Prediction
ReLU	Rectified Linear Unit
RMSE	Root Mean Square Error
RNN	Recurrent Neural Network
SEL	Squared Error Loss
SGD	Stochastic Gradient Descent
SMAPE	Symmetric Mean Absolute Percentage Error

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.

[LL19] Hongyu Liu and Bo Lang. Machine learning and deep learning methods for intrusion detection systems: A survey. *Applied Sciences*, 9(20), 2019.

[MBM+18] Yair Meidan, Michael Bohadana, Yael Mathov, Yisroel Mirsky, Dominik Breitenbacher, Asaf Shabtai, and Yuval Elovici. N-baiot: Network-based detection of iot botnet attacks using deep autoencoders. *CoRR*, abs/1805.03409, 2018.

[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.

[MZIV18] Fares Meghdouri, Tanja Zseby, and Félix Iglesias Vázquez. Analysis of lightweight feature vectors for attack detection in network traffic. *Applied Sciences*, 8:2196, 11 2018.

[PB18] Ranjit Panigrahi and Samarjeet Borah. A detailed analysis of cicids2017 dataset for designing intrusion detection systems. 7:479–482, 01 2018.

[PNI+18] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018.

[RATS18] Benjamin J. Radford, Leonardo M. Apolonio, Antonio J. Trias, and Jim A. Simpson. Network traffic anomaly detection using recurrent neural networks. *CoRR*, abs/1803.10769, 2018.

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

[SK19a] Alaa Sagheer and Mostafa Kotb. Time series forecasting of petroleum production using deep lstm recurrent networks. *Neurocomputing*, 323:203–213, 2019.

[SK19b] Alaa Sagheer and Mostafa Kotb. Unsupervised pre-training of a deep lstm-based stacked autoencoder for multivariate time series forecasting problems. *Scientific Reports*, 9:19038, 12 2019.

[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.

[SYZ20] Ahmed Samy, Haining Yu, and Hongli Zhang. Fog-based attack detection framework for internet of things using deep learning. *IEEE Access*, PP:1–1, 04 2020.

[TBLG09] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A. Ghorbani. A detailed analysis of the kdd cup 99 data set. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pages 1–6, 2009.

[Tea15] Google Brain Team. Tensorflow, 2015.

[VR19] Abhishek Verma and Virender Ranga. Evaluation of network intrusion detection systems for rpl based 6lowpan networks in iot. *Wireless Personal Communications*, 108:1571–1594, 10 2019.

[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.

[WZA06] Nigel Williams, Sebastian Zander, and Grenville Armitage. A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. *Computer Communication Review*, 36:5–16, 10 2006.

[YLZ20] Lun-Pin Yuan, Peng Liu, and Sencun Zhu. Recomposition vs. prediction: A novel anomaly detection for discrete events based on autoencoder. *CoRR*, abs/2012.13972, 2020.

[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.